ПРОФЕССИОНАЛЬНО О SPRING

4-е издание

# Spring 4

## ДЛЯ ПРОФЕССИОНАЛОВ

*РУКОВОДСТВО ПО SPRING FRAMEWORK 4, СООТВЕТСТВУЮЩЕЕ ОТРАСЛЕВЫМ СТАНДАРТАМ*

Крис Шеффер, Кларенс Хо и Роб Харроп

ВИЛЬЯМС
www.williamspublishing.com

Apress®
www.apress.com

# Pro Spring 4

Fourth Edition

**Chris Schaefer**
**Clarence Ho**
**Rob Harrop**

Apress®

# Spring 4

## ДЛЯ ПРОФЕССИОНАЛОВ

4-е издание

**Крис Шеффер**
**Кларенс Хо**
**Роб Харроп**

Москва • Санкт-Петербург • Киев
2015

ББК 32.973.26-018.2.75
Ш53
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского *Ю.Н. Артеменко*
Под редакцией *Ю.Н. Артеменко*
По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

**Шефер,** Крис, **Хо,** Кларенс, **Харроп,** Роб.

Ш53 Spring 4 для профессионалов, 4-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2015. — 752 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1992-2 (рус.)

**ББК 32.973.26-018.2.75**



*Научно-популярное издание*

**Крис Шефер, Кларенс Хо, Роб Харроп**

# Spring 4 для профессионалов
# 4-е издание

Верстка *Т.Н. Артеменко*
Художественный редактор *В.Г. Павлютин*

Подписано в печать 24.06.2015. Формат 70×100/16.
Гарнитура Times.
Усл. печ. л. 60,63. Уч.-изд. л. 41,2.
Тираж 500 экз. Заказ № 3180.

Отпечатано способом ролевой струйной печати
в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1

ООО, "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1992-2 (рус.)
ISBN 978-1-4302-6151-3 (англ.)

# Оглавление

# Содержание

*Я посвящаю эту книгу моей жене, сыну, семье, друзьям и, конечно же, моей кошке Салли, кто делил со мной домашний офис в течение многих часов бодрствования, сохраняя мне рассудок.*

Крис Шефер

# Об авторах

**Крис Шефер** — разработчик программного обеспечения, ориентирующийся преимущественно на технологии Java и JVM. Он живет в Венисе (Флорида) со своей женой, сыном и кошкой. Помимо рабочих технологий увлекается ездой на велосипеде, разнообразными мероприятиями на свежем воздухе и астрономией.

**Кларенс Хо** — ведущий Java-архитектор в фирме SkywideSoft Technology Limited (`www.skywidesoft.com`), занимающейся консультациями по программному обеспечению и расположенной в Гонконге. Работая в сфере информационных технологий более 20 лет, он был главой команды во многих проектах по разработке приложений на дому, а также предоставлял консультационные услуги по производственным решениям своим клиентам. Кларенс начал программировать на Java в 2001 году, а с 2005 года был вовлечен в проектирование и разработку JEE-приложений с помощью таких технологий, как EJB, Spring Framework, Hibernate, JMS и WS. С тех пор он выступал в качестве Java-архитектора корпоративных приложений.

В настоящее время Кларенс работает консультантом в международном финансовом учреждении, будучи задействованным в разнообразных областях, в числе которых архитектурное проектирование Java EE, обучение, предоставление рекомендаций по технологическим решениям и рекомендуемые приемы разработки приложений.

В свободное от работы время Кларенс занимается спортом (бегом трусцой, плаванием, футболом, пешеходным туризмом), чтением, просмотром фильмов, а также общением с друзьями.

**Роб Харроп** является соучредителем SpringSource — компании, которая стоит за успешным проектом Spring Framework. В настоящее время он занимает должность руководителя технического отдела в First Banco. До SpringSource Роб был соучредителем и техническим директором в Cake Solutions (Манчестер, Соединенное Королевство). Он специализируется на крупных масштабируемых корпоративных системах.

Роб был автором и соавтором в пяти книгах. Он доступен в Твиттере как `@robertharrop`.

# О техническом рецензенте

**Мануэль Джордан Элера** — разработчик-самоучка и исследователь, которому нравится изучать новые технологии для проведения собственных экспериментов и создавать из них новые сочетания.

Мануэль выиграл звание 2010 Springy Award — Community Champion. В свободное от работы время (которого не особенно много) он читает Библию и сочиняет музыку на своей гитаре. Мануэль известен как `dr_pompeii`.

Мануэль выступал техническим рецензентом в следующих книгах, выпущенных издательством Apress:

- *Pro SpringSource dm Server* (2009 г.)
- *Spring Enterprise Recipes* (2009 г.)

- *Spring Recipes, Second Edition* (2010 г.)
- *Pro Spring Integration* (2011 г.)
- *Pro Spring Batch* (2011 г.)
- *Pro Spring 3* (2012 г.)
- *Pro Spring MVC: With Web Flow* (2012 г.)
- *Pro Spring Security* (2013 г.)
- *Pro Hibernate and MongoDB* (2013 г.)
- *Pro JPA 2, Second Edition* (2013 г.)
- *Practical Spring LDAP* (2013 г.)

Найти его можно в собственном блоге (`www.manueljordanelera.blogspot.com`), а также в Твиттере (`@dr_pompeii`).

# От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: `info@williamspublishing.com`
WWW: `http://www.williamspublishing.com`

Информация для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1
Украины: 03150, Киев, а/я 152

# Введение

Раскрывая версию Spring Framework 4, эта книга представляет собой наиболее исчерпывающее справочное и учебное руководство по Spring, которое позволит задействовать всю мощь этой лидирующей платформы, предназначенной для разработки корпоративных приложений на языке Java.

В этом издании рассматривается ядро платформы Spring и ее интеграция с другими ведущими технологиями Java, такими как Hibernate, JPA 2 и WebSocket. Мы поделимся с вами собственным опытом и реальными практическими приемами, применяемыми во время разработки приложений уровня предприятия, в числе которых обеспечение удаленного взаимодействия, создание транзакций, построение веб-уровня и уровня презентаций, а также многое другое.

Благодаря этой книге, вы изучите следующие темы.

- Использование инверсии управления (IoC) и внедрения зависимостей (DI).
- Использование аспектно-ориентированного программирования (АОП) вместе с платформой Spring и понимание причин его важности.
- Построение веб-приложений на основе Spring с применением Spring MVC и WebSocket.
- Использование нового синтаксиса лямбда-выражений Java 8.
- Работа со сценарными языками наподобие Groovy для снабжения создаваемых приложений расширенной функциональностью.

Вооружитесь знаниями для построения сложных Spring-приложений сверху донизу. Настоящая книга ориентирована на опытных Java-разработчиков, которые могут впервые изучать Spring или обладают минимальным представлением об этой платформе. Она предназначена для тех, кто уже занимается или планирует погрузиться в мир разработки корпоративных приложений на языке Java.

# ГЛАВА 1

# Введение в Spring

Когда мы думаем о сообществе разработчиков на языке Java, мы вспоминаем полчища старателей, которые в конце 1840-х годов неистово прочесывали реки Северной Америки в поисках золотых самородков. Наши "реки" — реки Java-разработчиков — изобилуют проектами с открытым кодом, но, в отличие от истории со старателями, нахождение действительно полезного проекта может оказаться длительным по времени и трудным процессом.

Многим проектам с открытым кодом Java характерна общая особенность — они были призваны просто заполнить пробел в реализации последней "модной" технологии или шаблона. С учетом сказанного, многие высококачественные и полезные проекты предназначены для решения реальных потребностей в реальных приложениях, и в этой книге вы встретите ряд таких проектов. С одним из них вы ознакомитесь достаточно подробно — и это, как не трудно догадаться, Spring.

На протяжении всей книги вы будете сталкиваться со многими применениями разнообразных технологий с открытым кодом, причем все они объединены в платформу Spring Framework. Благодаря Spring, разработчик приложений может пользоваться широким спектром инструментов с открытым кодом, не занимаясь написанием больших объемов кода и не привязывая создаваемое приложение слишком тесно к какому-то конкретному инструменту.

В этой главе вы ознакомитесь с платформой Spring Framework без анализа крупных примеров или изучения подробных объяснений. Если вы уже знакомы с проектом Spring, можете сразу переходить к чтению главы 2.

## Что такое Spring?

Пожалуй, наиболее трудной частью объяснения технологии Spring является точная классификация того, что она собой представляет. Обычно *Spring* описывают как облегченную платформу для построения Java-приложений, но с этим утверждением связаны два интересных момента.

Во-первых, Spring можно применять для построения любого приложения на языке Java (например, автономных, веб-приложений или приложений Java Enterprise Edition (JEE)) в отличие от многих других платформ (таких как Apache Struts, которая ограничена созданием только веб-приложений).

Во-вторых, характеристика *облегченная* в действительности не имеет никакого отношения к количеству классов или размеру дистрибутива, а вместо этого определяет принцип всей философии Spring — минимальное воздействие. Платформа Spring является облегченной в том смысле, что для использования всех преимуществ ядра

Spring вы должны вносить минимальные (если вообще какие-либо) изменения в код своего приложения, а если в какой-то момент вы решите прекратить пользоваться Spring, то и это сделать очень просто.

Обратите внимание, что речь идет только о ядре Spring — многие дополнительные компоненты Spring, такие как доступ к данным, требуют более тесной привязки к Spring Framework. Однако польза от такой привязки вполне очевидна, к тому же повсеместно в книге мы будем представлять приемы для сведения к минимуму ее влияния на разрабатываемые приложения.

## Эволюция Spring Framework

Платформа Spring Framework берет начало из книги Рода Джонсона *Expert One-on-One: J2EE Design and Development* (Wrox, 2002 г.). За прошедшее десятилетие Spring Framework значительно выросла в плане основной функциональности, связанных проектов и поддержки со стороны сообщества. Теперь, когда доступен новый крупный выпуск Spring Framework, полезно взглянуть на важные средства, которые появлялись в каждом промежуточном выпуске Spring, в конечном итоге приведя к выходу Spring Framework 4.0.

- Spring 0.9
  - Первый публичный выпуск платформы, основанный на книге *Expert One-on-One: J2EE Design and Development*
- Spring 1.*x*
  - Spring Core: контейнер бинов и поддерживающие утилиты
  - Spring Context: `ApplicationContext`, пользовательский интерфейс, проверка достоверности, JNDI, Enterprise JavaBeans (EJB), удаленная обработка и почтовая поддержка
  - Spring DAO: поддержка инфраструктуры транзакций, Java Database Connectivity (JDBC) и объектов доступа к данным (data access object — DAO)
  - Spring ORM: поддержка Hibernate, iBATIS и Java Data Objects (JDO)
  - Spring AOP: реализация аспектно-ориентированного программирования (АОП), согласованного с альянсом AOP Alliance
  - Spring Web: базовые средства интеграции, такие как функциональность множественного содержимого, инициализация контекста посредством прослушивателей сервлетов и контекст веб-ориентированных приложений
  - Spring Web MVC: инфраструктура "модель-представление-контроллер" (Model-View-Controller — MVC) для построения веб-приложений
- Spring 2.*x*
  - Более простое конфигурирование XML за счет применения новой конфигурации, основанной на схеме XML (XML Schema), а не на формате DTD. Заметные области, подверженные усовершенствованиям, включают определения бинов, АОП и декларативные транзакции
  - Новые области действия бинов для использования в веб-приложениях и порталах (на уровне запроса, сеанса и глобального сеанса)

- Поддержка аннотаций `@AspectJ` для разработки с применением АОП
- Уровень абстракции Java Persistence API (JPA)
- Полная поддержка асинхронных объектов POJO (Plain Old Java Object — простой старый объект Java), управляемых сообщениями JMS
- Упрощения JDBC, в том числе `SimpleJdbcTemplate`, когда применяется Java 5+
- Поддержка именованных параметров JDBC (`NamedParameterJdbcTemplate`)
- Библиотека дескрипторов форм для Spring MVC
- Введение инфраструктуры Portlet MVC
- Поддержка динамических языков: бины могут быть написаны на JRuby, Groovy и BeanShell
- Поддержка уведомлений и управляемой регистрации MBean в JMX
- Абстракция `TaskExecutor`, введенная для планирования задач
- Поддержка аннотаций Java 5, в частности для `@Transactional` и `@Required` в дополнение к `@AspectJ`

- Spring 2.5.x
  - Новые конфигурационные аннотации `@Autowired` и поддержка аннотаций JSR-250 (`@Resource`, `@PostConstruct`, `@PreDestroy`)
  - Новые стереотипные аннотации: `@Component`, `@Repository`, `@Service`, `@Controller`
  - Поддержка автоматического сканирования путей классов для автоматического обнаружения и связывания классов, снабженных стереотипными аннотациями
  - Обновления АОП: появление элемента среза (pointcut) вида `bean(...)` и привязывания во время загрузки (load-time weaving) AspectJ
  - Полная поддержка управления транзакциями WebSphere
  - В дополнение к аннотации `@Controller` из Spring MVC добавлены аннотации `@RequestMapping`, `@RequestParam` и `@ModelAttribute` для поддержки обработки запросов посредством конфигурации с помощью аннотаций
  - Поддержка Tiles 2
  - Поддержка JSF 1.2
  - Поддержка JAX-WS 2.0/2.1
  - Введение инфраструктуры Spring TestContext Framework, которая поддерживает управляемое аннотациями тестирование взаимодействия, не зависящее от применяемой инфраструктуры тестирования
  - Возможность развертывания контекста приложения Spring как адаптера JCA

- Spring 3.0.x
  - Поддержка средств Java 5 — обобщений, аргументов переменной длины и других усовершенствований
  - Первоклассная поддержка адаптеров `Callables`, `Futures`, `ExecutorService`, а также интеграция `ThreadFactory`
  - Модули платформы теперь управляются отдельно, с одним деревом исходного кода на модуль JAR
  - Введение языка выражений Spring (Spring Expression Language — SpEL)
  - Интеграция основных средств и аннотаций Java Config
  - Универсальная система преобразования типов и система форматирования полей
  - Всеобъемлющая поддержка REST
  - Новое пространство имен MVC XML и дополнительные аннотации наподобие `@CookieValue` и `@RequestHeaders` для Spring MVC
  - Улучшения проверки достоверности и поддержка JSR-303 ("Проверка достоверности бинов")
  - Начальная поддержка для Java EE 6: аннотация `@Async/@Asynchronous`, JSR-303, JSF 2.0, JPA 2.0 и т.д.
  - Поддержка встроенных баз данных, таких как HSQL, H2 и Derby
- Spring 3.1.x
  - Новая абстракция Cache
  - Профили определения бинов могут определяться в XML; кроме того, имеется поддержка аннотации `@Profile`
  - Абстракция среды для унифицированного управления свойствами
  - Эквиваленты в форме аннотаций для распространенных элементов пространства имен Spring XML, такие как `@ComponentScan`, `@EnableTransactionManagement`, `@EnableCaching`, `@EnableWebMvc`, `@EnableScheduling`, `@EnableAsync`, `@EnableAspectJAutoProxy`, `@EnableLoadTimeWeaving` и `@EnableSpringConfigured`
  - Поддержка Hibernate 4
  - Поддержка инфраструктурой Spring TestContext Framework классов `@Configuration` и профилей определения бинов
  - Пространство имен `c:` для упрощенного внедрения через конструктор
  - Поддержка конфигурации на основе кода Servlet 3 контейнера типа сервлета
  - Возможность начальной загрузки `EntityManagerFactory` интерфейса JPA без `persistence.xml`
  - Добавление в Spring MVC средств Flash и `RedirectAttributes`, которые позволяют атрибутам продолжать существование после перенаправления за счет использования сеанса HTTP

- Усовершенствования переменных шаблонов URI
- Возможность снабжать аннотацией `@Valid` аргументы методов контроллеров `@RequestBody` в Spring MVC
- Возможность снабжать аннотацией `@RequestPart` аргументы методов контроллеров в Spring MVC

- Spring 3.2.x
  - Поддержка асинхронной обработки запросов на основе Servlet 3
  - Новая инфраструктура тестирования Spring MVC
  - Новые аннотации `@ControllerAdvice` и `@MatrixVariable` в Spring MVC
  - Поддержка обобщенных типов в `RestTemplate` и в аргументах `@RequestBody`
  - Поддержка Jackson JSON 2
  - Поддержка Tiles 3
  - Аргумент `@RequestBody` или `@RequestPart` теперь может сопровождаться аргументом `Errors`, делая возможной обработку ошибок проверки достоверности
  - Возможность исключения шаблонов URL за счет применения пространства имен MVC и параметров конфигурации Java Config
  - Поддержка `@DateTimeFormat` без Joda Time
  - Глобальное форматирование даты и времени
  - Усовершенствования параллелизма во всей платформе, сводящие к минимуму блокировки и в целом улучшающие параллельное создание ограниченных/прототипированных бинов
  - Новая система сборки, основанная на Gradle
  - Переход на GitHub: `https://github.com/SpringSource/spring-framework`
  - Усовершенствованная поддержка Java SE 7 / OpenJDK 7 внутри платформы и сторонних зависимостей. Теперь CGLIB и ASM являются частью Spring. В дополнение к AspectJ 1.6 поддерживается версия AspectJ 1.7

- Spring 4.0
  - Улучшенная практика начала работы посредством набора руководств Getting Started на веб-сайте `www.spring.io/guides`
  - Удаление устаревших пакетов и методов, относящихся к предыдущей версии Spring 3
  - Поддержка Java 8 с поднятием минимальной версии до Java 6 Update 18
  - Базовым уровнем для Spring Framework 4.0 теперь считается версия Java EE 6 и выше
  - Язык, специфичный для предметной области (domain-specific language — DSL), с помощью которого определяются бины Groovy, позволяет их конфигурировать посредством синтаксиса Groovy

- Усовершенствования, касающиеся основных контейнеров, тестирования и общей разработки веб-приложений
- Обмен сообщениями WebSocket, SockJS и STOMP

## Инверсия управления или внедрение зависимостей?

Ядро Spring Framework основано на принципе *инверсии управления* (Inversion of Control — IoC), когда создание и управление зависимостями между компонентами становятся внешними. Рассмотрим пример, в котором класс `Foo` при выполнении обработки определенного вида зависит от экземпляра класса `Bar`. Традиционно `Foo` создает экземпляр `Bar`, используя операцию `new` или получая нужный экземпляр от какой-то разновидности фабричного класса. Согласно подходу IoC, экземпляр класса `Bar` (или его подкласса) предоставляется классу `Foo` во время выполнения некоторым внешним процессом. Такое поведение, т.е. внедрение зависимостей во время выполнения, приводящее к инверсии управления, было переименовано Мартином Фаулером в более описательное — *внедрение зависимостей* (Dependency Injection — DI). Точная природа зависимостей, управляемых DI, обсуждается в главе 3.

---

**На заметку!** Как будет показано в главе 3, применение термина *внедрение зависимостей* при ссылке на инверсию управления всегда корректно. В контексте Spring эти термины можно использовать взаимозаменяемо, совершенно не теряя при этом смысла.

---

Реализация DI в Spring основана на двух ключевых концепциях Java — компонентах JavaBean и интерфейсах. При использовании Spring в качестве поставщика DI вы получаете гибкость определения конфигурации зависимостей внутри своих приложений разнообразными путями (например, XML-файлы, конфигурационные классы Java, аннотации в коде или новый метод определения бинов Groovy). Компоненты JavaBean (POJO) предоставляют стандартный механизм для создания ресурсов Java, которые поддаются конфигурированию многими способами вроде конструкторов и методов установки. В главе 3 вы увидите, как Spring использует спецификацию JavaBean для формирования ядра своей модели конфигурации DI; в действительности любой ресурс, управляемый Spring, называется *бином* (bean). На тот случай, если вы еще не знакомы с технологией JavaBean, в начале главы 3 будет приведен краткий пример.

Интерфейсы и DI — это взаимовыгодные технологии. Очевидно, что проектирование и написание кода интерфейсов предназначено для гибких приложений, но сложность связывания вместе приложения, использующего интерфейсы, довольно высока и требует от разработчиков дополнительных усилий по кодированию. За счет применения DI объем кода, который необходим при проектировании приложения на основе интерфейсов, снижается почти до нуля. Кроме того, с помощью интерфейсов можно получить максимальную отдачу от DI, потому что бины могут использовать любую реализацию интерфейса для удовлетворения их зависимости. Применение интерфейсов также позволяет Spring задействовать динамические прокси JDK (шаблон Proxy), чтобы предоставить мощные концепции вроде АОП для сквозной функциональности.

В контексте внедрения зависимостей Spring действует больше подобно контейнеру, чем платформе, предоставляя экземпляры классов вашего приложения со всеми необходимыми зависимостями, но делает это гораздо менее навязчивым способом. Использование Spring для DI основано всего лишь на следовании внутри своих классов соглашениям об именовании, принятым для компонентов JavaBean; ни специальных классов, предназначенных для наследования, ни собственных схем именования, которые должны соблюдаться, не предусмотрено. Во всяком случае, единственным изменением, которое делается в приложении, применяющем DI, является открытие доступа к дополнительным свойствам ваших компонентов JavaBean, что позволяет внедрять больше зависимостей во время выполнения.

## Эволюция внедрения зависимостей

Благодаря распространению Spring и других платформ DI, за последние несколько лет технология внедрения зависимостей обрела широкое признание в сообществе разработчиков на языке Java. В то же время разработчики были уверены, что использование DI было наилучшей практикой в разработке приложений, и преимущества применения DI были также хорошо изучены.

Популярность DI также была подтверждена официальным принятием процессом сообщества Java (Java Community Process — JCP) документ JSR-330 ("Dependency Injection for Java" — "Внедрение зависимостей для Java") в 2009 году. Документ JSR-330 стал формальным запросом спецификации Java (Java Specification Request — JSR) и одним из ее лидирующих авторов был Род Джонсон, основатель Spring Framework.

В JEE 6 документ JSR-330 стал одной из спецификаций, входящих в полный комплект технологий. За это время архитектура EJB (начиная с версии 3.0) также была существенно модернизирована; с целью упрощения разработки разнообразных приложений Enterprise JavaBeans была принята модель DI.

Несмотря на то что подробное обсуждение DI откладывается до главы 3, полезно взглянуть на преимущества, которые обеспечивает использование DI по сравнению с более традиционным подходом.

- **Сокращение объема связующего кода.** Одним из самых больших плюсов DI является возможность значительного сокращения объема кода, который должен быть написан для связывания вместе компонентов приложения. Часто этот код тривиален, так что создание зависимости сводится просто к созданию нового экземпляра объекта. Тем не менее, связующий код может стать довольно сложным, когда нужно искать зависимости в репозитории JNDI или если зависимости не могут вызываться напрямую, как в случае с удаленными (дистанционно) ресурсами. В таких ситуациях DI может действительно упростить связующий код, предоставляя автоматический поиск в JNDI и автоматическое создание прокси для удаленных ресурсов.
- **Упрощенная конфигурация приложения.** За счет применения DI процесс конфигурирования приложения значительно упрощается. Для конфигурирования классов, которые могут быть внедрены в другие классы, можно использовать разнообразные варианты. Тот же самый подход можно применять для формулирования требований зависимостей для внедряющего объекта при внедре-

нии соответствующего экземпляра бина или свойства. Вдобавок DI намного упрощает обмен одной реализации зависимости с другой. Рассмотрим случай, когда есть компонент DAO, который выполняет операции над данными в базе данных PostgreSQL, и его нужно обновить до Oracle. С помощью DI соответствующую зависимость бизнес-объектов можно легко переконфигурировать, чтобы она использовала реализацию Oracle вместо PostgreSQL.

- **Возможность управления общими зависимостями в единственном репозитории.** При традиционном подходе к управлению зависимостями общих служб, таких как подключение к источнику данных, транзакция и удаленные службы, вы создаете экземпляры (или получаете их из определенных фабричных классов) зависимостей там, где они нужны (внутри зависимого класса). Это приводит к распространению зависимостей по множеству классов в приложении, что может затруднить их изменение. В случае использования DI вся информация об общих зависимостях содержится в единственном репозитории, существенно упрощая управление зависимостями и уменьшая подверженность ошибкам.
- **Улучшенная возможность тестирования.** Когда классы проектируются для DI, становится возможной простая замена зависимостей. Это особенно полезно при тестировании приложения. Рассмотрим бизнес-объект, который производит определенную сложную обработку; в рамках этого он использует DAO для доступа к данным, хранящимся в реляционной базе данных. При тестировании вас не интересует проверка DAO — вам нужно протестировать бизнес-объект с разнообразными наборами данных. При традиционном подходе, когда бизнес-объект самостоятельно отвечает за получение экземпляра DAO, тестирование довольно затруднительно, поскольку нет возможности заменить реализацию DAO имитированной реализацией, которая возвращала бы наборы тестовых данных. Вместо этого нужно будет удостовериться, что тестовая база данных содержит корректные данные, и применять для прогона тестов полную реализацию DAO. В случае использования DI можно создать имитированную реализацию объекта DAO, которая возвращает наборы тестовых данных, и затем передавать их бизнес-объекту с целью проверки. Этот механизм может быть расширен для тестирования любого уровня приложения и особенно полезен при тестировании веб-компонентов, для которых создаются имитированные реализации `HttpServletRequest` и `HttpServletResponse`.
- **Стимулирование качественных проектных решений для приложений.** Вообще говоря, проектирование для DI означает проектирование с использованием интерфейсов. Типовое приложение, ориентированное на внедрение зависимостей, построено так, что все основные компоненты определяются как интерфейсы, после чего создаются конкретные реализации этих интерфейсов и затем связываются вместе с применением контейнера DI. Такая разновидность проектного решения была возможна в Java еще до появления DI и основанных на DI контейнеров, подобных Spring. Но за счет использования Spring вы получаете в свое распоряжение целый ряд средств DI и можете сосредоточиться на построении логики приложения, а не на поддерживающей DI платформе.

Как видите, технология DI предоставляет немало преимуществ приложению, однако она отнюдь не лишена недостатков. В частности, DI может затруднить вы-

яснение, какая реализация конкретной зависимости к каким объектам привязана, для тех, кто не особенно хорошо ориентируется в коде. Как правило, эта проблема возникает только при отсутствии опыта работы с DI. С обретением опыта и следуя рекомендациям по программированию для DI (например, помещение всех встраиваемых классов на каждом уровне приложения в один и тот же пакет), разработчики будут способны легко получить представление о картине в целом. Преимущества почти всегда перевешивают этот небольшой недостаток, тем не менее, о нем следует помнить при планировании приложения.

## За пределами внедрения зависимостей

Само по себе ядро Spring со своими развитыми возможностями DI является вполне достойным инструментом, но чем Spring действительно выделяется — так это широчайшим набором дополнительных средств, которые все элегантно спроектированы и построены с использованием принципов DI. Платформа Spring предлагает средства для всех уровней приложения, от вспомогательных API-интерфейсов для доступа к данным до расширенных возможностей MVC. Упомянутые возможности Spring хороши тем, что хотя Spring часто предоставляет собственный подход, они легко интегрируются с другими инструментами Spring, становясь полноправными членами семейства Spring.

### *Поддержка Java 8*

Версия Java 8 привносит множество захватывающих возможностей, поддерживаемых в Spring Framework 4, наиболее примечательными из которых являются лямбда-выражения и ссылки на методы с интерфейсами обратного вызова Spring. Другая функциональность Java 8 включает первоклассную поддержку `java.time` (JSR-310) и обнаружение имен параметров. Несмотря на то что Spring Framework 4.0 поддерживает Java 8, по-прежнему обеспечивается обратная совместимость с JDK 6 Update 18. При разработке новых проектов рекомендуется применять самую последнюю версию Java, такую как 7 или 8.

### *Аспектно-ориентированное программирование с использованием Spring*

Аспектно-ориентированное программирование (АОП) предоставляет возможность реализации *сквозной логики*, т.е. логики, которая применяется к множеству частей приложения, в одном месте и обеспечивает ее автоматическое применение по всему приложению.

Подход Spring к АОП заключается в создании *динамических прокси* для целевых объектов и *привязывании* объектов к сконфигурированному совету для выполнения сквозной логики. Согласно природе динамических прокси JDK, целевые объекты должны реализовывать интерфейс, объявляющий метод, в котором будет применен совет АОП.

Еще одной популярной библиотекой АОП является проект Eclipse AspectJ (`www.eclipse.org/aspectj`), который предоставляет более мощные средства, включая конструирование объектов, загрузку классов и большую возможность выделения сквозной функциональности.

Хорошая новость для разработчиков, применяющих Spring и АОП, заключается в том, что начиная с версии 2.0, в Spring поддерживается более тесная интеграция с AspectJ.

Ниже перечислено несколько особенностей.

- Поддержка выражений со срезами в стиле AspectJ.
- Поддержка стиля аннотаций `@AspectJ` при использовании для привязывания АОП из Spring.
- Поддержка аспектов, реализованных в AspectJ для DI.
- Поддержка привязывания во время загрузки внутри объекта `ApplicationContext` в Spring.

---

**На заметку!** Начиная с версии Spring Framework 3.2, поддержка аннотации `@AspectJ` может быть включена с помощью Java Config.

---

Оба вида АОП имеют свои области применения, и в большинстве случаев АОП из Spring достаточно для удовлетворения требований сквозной функциональности в приложении. Однако при наличии более сложных требований может использоваться AspectJ, причем в одном приложении Spring можно смешивать оба вида АОП.

АОП имеет много применений. Типичное применение, демонстрируемое во множестве традиционных примеров АОП, связано с выполнением некоторого вида регистрации, но возможности АОП выходят далеко за рамки тривиальных приложений регистрации. На самом деле внутри самой платформы Spring Framework АОП используется для многих целей, в частности, при управлении транзакциями. АОП в Spring подробно рассматривается в главе 5, где демонстрируются типичные применения АОП внутри Spring Framework и в примерах приложений, обсуждаются вопросы производительности АОП, а также области, в которых традиционные технологии подходят лучше, чем АОП.

### Язык выражений Spring (SpEL)

Язык выражений (Expression Language — EL) — это технология, позволяющая приложению манипулировать объектами Java во время выполнения. Однако с EL связана одна проблема: разные технологии предлагают собственные реализации и синтаксис EL. Например, Java Server Pages (JSP) и Java Server Faces (JSF) имеют собственные языки EL, синтаксис которых отличается друг от друга. Для решения этой проблемы и был создан унифицированный язык выражений (Unified Expression Language).

Поскольку платформа Spring Framework развивалась слишком быстро, существовала потребность в стандартном языке выражений, который мог бы совместно использоваться всеми модулями Spring Framework, а также другими проектами Spring. Поэтому, начиная с версии 3.0, в Spring появился язык выражений Spring (Spring Expression Language — SpEL), который предоставляет мощные средства для вычисления выражений, а также для доступа к объектам Java и бинам Spring во время выполнения. Результаты вычислений могут применяться в приложении или внедряться в другие компоненты JavaBean.

### Проверка достоверности в Spring

*Проверка достоверности* — еще одна обширная тема в приложениях любого вида. В идеальном сценарии правила проверки достоверности для атрибутов внут-

ри компонентов JavaBean, содержащих данные, могут быть применены согласованным образом вне зависимости от того, где инициирован запрос на манипуляцию данными — в пользовательском интерфейсе, пакетном задании или же удаленно (например, веб-службой, веб-службой REST или удаленным вызовом процедуры (Remote Procedure Call — RPC)).

Для решения указанных проблем в Spring предлагается встроенный API-интерфейс проверки достоверности в виде интерфейса `Validator`. Этот интерфейс предоставляет простой и лаконичный механизм, который позволяет инкапсулировать логику проверки достоверности в класс, ответственный за проверку достоверности целевого объекта. В дополнение к целевому объекту метод проверки достоверности принимает объект `Errors`, который используется для сбора любых возникающих ошибок при проверке.

В Spring также имеется полезный служебный класс `ValidationUtils`, который предлагает удобные методы для вызова других валидаторов, проверки наличия распространенных проблем вроде пустых строк и сообщения об ошибках указанному объекту `Errors`.

Руководствуясь необходимостью, JCP была также разработана спецификация Bean Validation API (JSR-303), которая предоставляет стандартный путь определения правил проверки достоверности для бинов. Например, когда к свойству бина применяется аннотация `@NotNull`, она указывает, что атрибут не должен содержать нулевое значение перед тем, как он может быть сохранен в базе данных.

Начиная с версии 3.0, в Spring доступна встроенная поддержка спецификации JSR-303. Для использования этого API-интерфейса нужно просто объявить объект `LocalValidatorFactoryBean` и внедрить интерфейс `Validator` в любые бины, управляемые Spring. Платформа Spring найдет лежащую в основе реализацию. По умолчанию Spring сначала ищет Hibernate Validator (`hibernate.org/subprojects/validator`), который представляет собой популярную реализацию JSR-303. Многие технологии пользовательских интерфейсов (например, JSF 2 и Google Web Toolkit), включая Spring MVC, также поддерживают применение проверки достоверности JSR-303 в пользовательском интерфейсе. Прошли те времена, когда разработчикам приходилось писать одну и ту же логику проверки достоверности как в пользовательском интерфейсе, так и на уровне взаимодействия с базой данных. Более подробно об этом пойдет речь в главе 10.

---

**На заметку!** Начиная с версии Spring Framework 4.0, поддерживается версия 1.1 спецификации Bean Validation API (JSR-349).

---

### *Доступ к данным в Spring*

Доступ к данным и постоянство являются, пожалуй, наиболее обсуждаемыми темами в мире Java. Платформа Spring обеспечивает великолепную интеграцию с любым выбранным инструментом доступа к данным. Вдобавок для многих проектов Spring предлагает в качестве жизнеспособного решения простое средство JDBC (Java Database Connectivity — подключение к базам данных Java) с его упрощенными API-интерфейсами, представляющими собой оболочки для стандартного API-интерфейса.

**На заметку!** Начиная с версии Spring Framework 4.0, поддержка iBATIS была удалена. Проект MyBatis-Spring предлагает интеграцию с платформой Spring; дополнительные сведения о нем можно получить по адресу `http://mybatis.github.io/spring/`.

Однако по причине безудержного роста Интернета и облачных вычислений в последние несколько лет, помимо реляционных было разработано много других баз данных "специального назначения". Примерами могут служить базы данных, основанные на парах "ключ-значение" и предназначенные для обработки исключительно больших объемов данных (в общем случае на них ссылаются как на NoSQL), графовые базы данных и документные базы данных. Чтобы помочь разработчикам в поддержке таких баз данных и не усложнять модель доступа к данным Spring, был создан отдельный проект под названием Spring Data (`http://projects.spring.io/spring-data`). Этот проект в дальнейшем был разделен на различные категории с целью поддержки более специфических требований по доступу к данным.

**На заметку!** Поддержка нереляционных баз данных в Spring в этой книге не рассматривается. Для тех, кто интересуется этой темой, упомянутый ранее проект Spring Data будет хорошей отправной точкой. На странице проекта указаны нереляционные базы данных, которые он поддерживает, а также предоставлены ссылки на домашние страницы для этих баз данных.

Поддержка JDBC в Spring делает построений приложения на основе JDBC вполне реальным, даже в особо сложных случаях. Поддержка Hibernate, JDO и JPA еще более упрощает и без того простые API-интерфейсы, сокращая затраты на кодирование со стороны разработчиков. При использовании API-интерфейсов Spring для доступа к данным через любой инструмент имеется возможность получить все преимущества великолепной поддержки транзакций, предлагаемой Spring. Более подробные сведения будут предоставлены в главе 9.

Одним из самых полезных средств Spring является возможность простого комбинирования технологий доступа к данным в рамках приложения. Например, приложение может работать с базой данных Oracle, но использовать Hibernate для большей части логики доступа к данным. Однако если требуются некоторые специфичные для Oracle средства, очень просто реализовать соответствующую часть уровня доступа к данным с помощью API-интерфейсов JDBC, поддерживаемых Spring.

### *Поддержка Object to XML Mapping (OXM) в Spring*

Многие приложения должны интегрироваться или предоставлять службы для других приложений. Одним из распространенных требований является обмен данными с другими системами, либо на регулярной основе, либо в реальном времени. Наиболее распространенным форматом данных считается XML. В результате возникает необходимость в преобразовании компонента JavaBean в формат XML и наоборот.

Платформа Spring поддерживает много общепринятых инфраструктур отображения Java в XML и, как обычно, устраняет потребность в непосредственной привязке к любой специфической реализации. Spring предоставляет общие интерфейсы для маршализации (преобразования компонентов JavaBean в XML) и демаршализации (преобразования XML в объекты Java) для внедрения зависимостей в любые бины Spring. Поддерживаются такие распространенные библиотеки, как Java Architecture

for XML Binding (JAXB), Castor, XStream, JiBX и XMLBeans. При рассмотрении в главе 12 удаленного доступа приложения Spring к бизнес-данным в формате XML будет показано, как использовать в своих приложениях поддержку OXM (Object to XML Mapping — отображение объектов в XML), предлагаемую Spring.

### Управление транзакциями

Платформа Spring предлагает удобный уровень абстракции для управления транзакциями, который позволяет осуществлять программный и декларативный контроль над транзакциями. За счет применения уровня абстракции для транзакций упрощается смена лежащего в основе протокола транзакций и диспетчеров ресурсов. Вы можете начать с простых, локальных, специфичных для ресурса транзакций и затем перейти к глобальным, мультиресурсным транзакциям, не изменяя уже написанного кода.

Транзакции подробно рассматриваются в главе 9.

### Упрощение и интеграция с JEE

Благодаря растущему принятию инфраструктур DI, таких как Spring, многие разработчики решили строить приложения, используя инфраструктуры DI для поддержки подхода EJB из JEE. В результате сообщество JCP также столкнулось со сложностью EJB. Начиная с версии 3.0 спецификации EJB, доступный API-интерфейс был упрощен и теперь включает многие концепции DI.

Тем не менее, для приложений, которые были построены на EJB или должны развертывать Spring-приложения в контейнере JEE и пользоваться корпоративными службами сервера приложений (например, диспетчер транзакций JTA (Java Transaction API — API-интерфейс транзакций Java), пул подключений к источникам данных и фабрики подключений JMS), Spring также предлагает упрощенную поддержку указанных технологий. Для EJB платформа Spring предоставляет простое объявление, позволяющее выполнять поиск JNDI и внедрять в бины Spring. На противоположной стороне Spring также обеспечивает простую аннотацию для внедрения бинов Spring в компоненты EJB.

Для любых ресурсов, сохраненных в местоположении, которое доступно через JNDI, платформа Spring позволяет избавиться от сложного кода поиска и внедрять ресурсы, управляемые JNDI, как зависимости в другие объекты во время выполнения. В качестве побочного эффекта приложение становится не привязанным к JNDI, увеличивая степень многократного использования кода в будущем.

### MVC на веб-уровне

Хотя платформа Spring может применяться практически в любых конфигурациях, от настольной системы до веб, она предлагает широкий спектр классов, предназначенных для поддержки создания веб-приложений. Благодаря Spring, вы получаете максимальную гибкость при выборе способа реализации пользовательского интерфейса для веб-приложения.

Наиболее популярным шаблоном, который используется при разработке веб-приложений, является MVC. В последних версиях Spring постепенно развилась от простой веб-платформы до полноценной реализации MVC.

Прежде всего, следует отметить обширную поддержку представлений в Spring MVC. В дополнение к стандартной поддержке JSP и JSTL (Java Standard Tag

Library — стандартная библиотека дескрипторов Java), которая значительно подкреплена библиотеками дескрипторов Spring, в распоряжении разработчиков имеется полностью интегрированная поддержка Apache Velocity, FreeMarker, Apache Tiles и XSLT. Кроме того, доступен набор базовых классов представлений, которые упрощают добавление к приложениям вывода Microsoft Excel, PDF и JasperReports.

Во многих случаях вы сочтете поддержку Spring MVC вполне достаточной для удовлетворения потребностей ваших веб-приложений. Однако Spring может также интегрироваться с другими популярными веб-платформами, такими как Struts, JSF, Atmosphere, Google Web Toolkit (GWT) и т.д.

В последние годы технология веб-платформ стремительно развивалась. Пользователи требовали более быстрого отклика и высокой интерактивности, и это привело к появлению Ajax — широко распространенной технологии для разработки насыщенных Интернет-приложений (Rich Internet Application — RIA). С другой стороны, пользователи также хотят иметь доступ к своим приложениям на любом устройстве, включая смартфоны и планшеты. Это порождает необходимость в веб-платформах, которые поддерживают HTML5, JavaScript и CSS3. В главе 16 мы обсудим разработку веб-приложений с применением Spring MVC.

### Поддержка WebSocket

Начиная с версии Spring Framework 4.0, доступна поддержка интерфейса Java API для WebSocket (JSR-356). В WebSocket определен API-интерфейс для создания постоянного подключения между клиентом и сервером, обычно реализованного в веб-браузерах и серверах. Разработка в стиле WebSocket открывает простор для эффективных двухсторонних коммуникаций, которые делают возможным обмен сообщениями в реальном времени для быстрореагирующих приложений. Использование поддержки WebSocket подробно рассматривается в главе 16.

### Поддержка удаленных технологий

Доступ или отображение удаленных компонентов в Java никогда не было простой работой. Имея дело с платформой Spring, вы можете получить в свое распоряжение обширную поддержку большого диапазона удаленных технологий для быстрого отображения и доступа к удаленным службам.

Платформа Spring предлагает поддержку разнообразных механизмов удаленного доступа, в том числе RMI (Java Remote Method Invocation — удаленный вызов методов Java), JAX-WS, протоколы Hessian и Burlap от Caucho, JMS, AMQP (Advanced Message Queuing Protocol — расширенный протокол очередизации сообщений) и REST. В дополнение к этим удаленным протоколам, Spring также предоставляет собственный вызывающий объект на основе HTTP, базирующийся на стандартной сериализации Java. За счет применения возможностей динамического создания прокси, предлагаемых платформой Spring, прокси для удаленного ресурса внедряется в качестве зависимости в ваши классы, что устраняет необходимость привязки приложения к специфической реализации удаленной технологии и сокращает объем кода, который должен быть написан для приложения. Поддержка удаленных технологий в Spring рассматривается в главе 12.

### Поддержка электронной почты

Отправка электронной почты является типичным требованием для многих видов приложений и полноценно поддерживается в Spring Framework. Платформа Spring предоставляет упрощенный API-интерфейс для отправки сообщений электронной почты, который хорошо согласуется с возможностями Spring DI. В Spring поддерживается стандартный интерфейс JavaMail API.

Spring позволяет создать сообщение-прототип в контейнере DI и использовать его в качестве основы для всех сообщений, отправляемых из приложения. Это упрощает настройку параметров почтового сообщения, таких как тема и адрес отправителя. Вдобавок для настройки тела сообщения платформа Spring интегрируется с механизмами шаблонов вроде Apache Velocity, которые дают возможность вынести содержимое сообщений за пределы Java-кода.

### Поддержка планирования заданий

Большинство нетривиальных приложений требуют определенных возможностей планирования. Возможность планирования запуска кода в заранее определенный момент времени представляет собой очень полезное средство для разработчиков, для чего бы это не делалось — для отправки обновлений заказчикам или для выполнения служебных задач.

Платформа Spring предлагает поддержку планирования, которая удовлетворяет требованиям большинства распространенных сценариев. Задача может быть запланирована на запуск либо с фиксированным интервалом, либо с применением выражения для Unix-утилиты `cron`.

С другой стороны, для выполнения и планирования задач платформа Spring также интегрируется с другими библиотеками планирования. Например, в среде сервера приложений Spring может делегировать выполнение библиотеке CommonJ, которая используется многими серверами приложений. Для планирования задач Spring также поддерживает несколько библиотек, в числе которых JDK Timer API и Quartz, представляющие собой известные библиотеки планирования с открытым кодом.

Поддержка планирования в Spring обсуждается в главе 11.

### Поддержка динамических сценариев

Начиная с JDK 6, в Java появилась поддержка динамических языков, которая позволяет запускать сценарии, написанные на других языках, в среде JVM. Примерами таких языков могут служить Groovy, JRuby и JavaScript.

Платформа Spring также поддерживает выполнение динамических сценариев в Spring-приложении. Кроме того, можно определить Spring-бин, написанный на языке динамических сценариев, и внедрить его в нужные компоненты JavaBean. В число языков динамических сценариев, поддерживаемых Spring, входят Groovy, JRuby и BeanShell. Более детально поддержка динамических сценариев в Spring рассматривается в главе 14.

### Упрощенная обработка исключений

Существует область, в которой Spring действительно помогает сократить объем повторяющегося рутинного кода — обработка исключений. Основой философии Spring в этом отношении является тот факт, что проверяемые исключения используются в Java чрезмерно, и платформа не должна принуждать к перехвату любого

исключения, после которого вряд ли будет возможность провести восстановление — мы полностью разделяем такую точку зрения.

В действительности многие платформы спроектированы так, чтобы сократить потребность в написании кода для обработки проверяемых исключений. Тем не менее, во многих таких платформах принято придерживаться проверяемых исключений, но искусственно снижать степень детализации в иерархии классов исключений. В отношении Spring вы отметите интересный момент: вследствие удобства, которое получают разработчики благодаря применению непроверяемых исключений, иерархия классов исключений является в высшей степени детальной.

Повсеместно в этой книге вы будете встречать примеры, в которых механизмы обработки исключений Spring могут уменьшать объем необходимого кода и в то же время совершенствовать возможности идентификации, классификации и диагностики ошибок внутри приложения.

# Проект Spring

Одной из самых подкупающих особенностей проекта Spring является уровень активности, наблюдаемой в сообществе, и степень взаимодействия между Spring и другими проектами, такими как CGLIB, Apache Geronimo и AspectJ. Наиболее расхваливаемое преимущество открытого кода связано с тем, что если даже проект будет неожиданно свернут, у вас останется исходный код; но давайте посмотрим правде в глаза — вряд ли вы захотите остаться один на один с кодовой базой масштаба Spring и впоследствии поддерживать и улучшать ее. По этой причине отрадно знать, что дела идут хорошо, а сообщество Spring сохраняет высокую активность.

## Происхождение Spring

Как отмечалось ранее в этой главе, истоки Spring корнями уходят в книгу *Expert One-to-One J2EE Design and Development*. В этой книге Род Джонсон представил собственную платформу под названием Interface 21 Framework, которую разработал для использования в своих приложениях. После выпуска в мир открытого кода эта платформа сформировала основу Spring Framework, которую мы знаем сейчас.

Платформа Spring быстро прошла через стадии бета-тестирования и предвыпускной версии, и ее первый официальный выпуск 1.0 стал доступным 24 марта 2004 года. С тех пор платформа Spring значительно разрослась и на время написания этой книги она доступна в виде версии Spring Framework 4.0.

## Сообщество Spring

Сообщество Spring — одно из лучших сообществ из всех проектов с открытым кодом, с которыми мы встречались. Списки рассылки и форумы всегда активны, а скорость добавления новых средств, как правило, высока. Команда разработчиков действительно сосредоточена на том, чтобы сделать Spring самой успешной платформой для построения Java-приложений, и об этом можно судить по качеству производимого ими кода.

Как уже упоминалось, Spring также получает преимущества от тесных взаимоотношений с другими проектами с открытым кодом, и этот факт чрезвычайно важен, если вы полагаетесь на значительную зависимость от полного дистрибутива Spring.

С точки зрения пользователя, пожалуй, самыми выдающимися особенностями Spring являются великолепная документация и тестовый комплект, сопровождающие дистрибутив. В документации описаны практически все возможности Spring, и это упрощает освоение платформы новыми пользователями. Тестовый комплект Spring является всеобъемлющим — команда разработчиков пишет тесты для любого аспекта. Когда они обнаруживают ошибку, то исправляют ее, сначала написав тест, который выявляет ошибку, и затем обеспечив его успешное прохождение.

Исправление ошибок и создание новых средств не ограничивается одной лишь командой разработчиков! Вы можете внести свой вклад в код, отреагировав на запросы в любом перечне проектов Spring внутри официальных репозиториев GitHub (`http://github.com/spring-projects`). Кроме того, можно формулировать проблемы и отслеживать их с помощью системы JIRA для Spring (`https://jira.springsource.org/secure/Dashboard.jspa`).

Что все это означает для вас? Выражаясь просто, это значит, что вы можете быть уверены в высоком качестве кода Spring Framework, а также в том, что в обозримом будущем команда разработчиков Spring продолжит совершенствовать и без того замечательную платформу.

## Комплект Spring Tool Suite

Для упрощения процесса разработки основанных на Spring приложений в Eclipse в рамках Spring создан проект Spring IDE. Вскоре после этого компания SpringSource, которую Род Джонсон основал после Spring, создала интегрированный инструмент под названием Spring Tool Suite (STS), доступный для загрузки по адресу `www.spring.io/tools`. Хотя он был платным продуктом, теперь данный инструмент доступен бесплатно. Инструмент STS интегрирован в Eclipse IDE, Spring IDE, Mylyn (среда разработки в Eclipse, основанная на задачах), Maven for Eclipse, AspectJ Development Tools и множество других полезных подключаемых модулей Eclipse внутри единственного пакета. В каждой новой версии появляются дополнительные средства, такие как поддержка языка сценариев Groovy, графический редактор конфигурации Spring, инструменты визуальной разработки для проектов вроде Spring Batch и Spring Integration, а также поддержка сервера приложений Pivotal tc Server.

---

**На заметку!** Компания SpringSource была приобретена VMWare и включена в состав Pivotal Software, Inc.

---

В дополнение к комплекту, основанному на Java, доступен комплект Groovy/Grails Tool Suite, который обладает аналогичными возможностями, но ориентирован на разработку Groovy и Grails (`www.spring.io/tools`).

## Проект Spring Security

Проект Spring Security (`http://projects.spring.io/spring-security`), ранее называвшийся Acegi Security System for Spring — это еще один важный проект в рамках Spring. Проект Spring Security предоставляет всеобъемлющую поддержку безопасности на уровне веб-приложения и отдельных методов. Он тесно интегрирован с платформой Spring Framework и другими распространенными механизмами аутентификации, такими как базовая аутентификация HTTP, вход с помощью форм,

сертификаты X.509 и продукты с реализацией единого входа (single sign-on — SSO), например, CA SiteMinder. Проект Spring Security предлагает управление доступом на основе ролей к ресурсам приложения, а в приложениях с более сложными требованиями к безопасности (скажем, разделение данных) поддерживается использование списка управления доступом (Access Control List — ACL). Тем не менее, Spring Security в основном применяется в защищенных веб-приложениях, как будет показано в главе 16.

### Проекты Spring Batch и Spring Integration

Не стоит и говорить, что выполнение пакетных заданий и интеграция являются распространенными сценариями использования в приложениях. Чтобы удовлетворить данные требования и упростить их решение для разработчиков в этих областях, в рамках Spring созданы проекты Spring Batch и Spring Integration. Проект Spring Batch предоставляет общую инфраструктуру и разнообразные политики для реализации пакетных заданий, значительно сокращая объем рутинного кодирования. За счет реализации шаблонов интеграции корпоративных приложений (Enterprise Integration Patterns — EIP) проект Spring Integration помогает упростить интеграцию Spring-приложений с внешними системами.

### Другие проекты

Выше мы описали ключевые модули Spring и некоторые важные проекты внутри Spring, но существует множество других проектов, предназначенных для удовлетворения других требований сообщества. В качестве примеров можно упомянуть Spring Boot, Spring XD, Spring for Android, Spring Mobile, Spring Social и Spring AMQP. За дополнительной информацией обращайтесь на веб-сайт Spring by Pivotal (`www.spring.io/projects`).

## Альтернативы Spring

Учитывая предыдущие замечания относительно количества проектов с открытым кодом, не должен вызывать удивление тот факт, что Spring — далеко не единственная платформа, которая предлагает средства внедрения зависимостей (DI) или полноценные сквозные решения для построения приложений. На самом деле проектов настолько много, что их сложно даже упомянуть вскользь. Соблюдая дух открытости, мы включили здесь краткое описание нескольких таких платформ, но убеждены, что ни одна из них не предлагает настолько исчерпывающее решение, как это делает Spring.

### JBoss Seam Framework

Основанная Гэвином Кингом (создателем библиотеки Hibernate ORM), Seam Framework (`www.seamframework.org`) представляет собой еще одну зрелую платформу, базирующуюся на DI. Она поддерживает разработку пользовательских интерфейсов веб-приложений (JSF), уровень бизнес-логики (EJB 3) и JPA для реализации постоянства. Основное отличие между Seam и Spring связано с тем, что платформа Seam Framework построена полностью на стандартах JEE. Платформа JBoss также способствует развитию идей Seam Framework в JCP и появлению спецификации JSR-299 ("Contexts and Dependency Injection for the Java EE Platform" — "Контексты и внедрение зависимостей для платформы Java EE").

### Google Guice

Следующей популярной платформой DI является Google Guice (`http://code.google.com/p/google-guice`). Поддерживаемая поисковым гигантом Google, Guice представляет собой облегченную платформу, которая сосредоточена на обеспечении DI для управления конфигурацией приложений. Она также была опорной реализацией спецификации JSR-330 ("Dependency Injection for Java" — "Внедрение зависимостей для Java").

### PicoContainer

PicoContainer (`http://picocontainer.com`) — это исключительно малый контейнер DI, который позволяет использовать DI для своего приложения, не вводя никаких зависимостей помимо PicoContainer. Поскольку PicoContainer является всего лишь контейнером DI, вы можете обнаружить, что по мере роста приложения приходится вводить другую платформу, такую как Spring; в этом случае лучше применять Spring с самого начала. Тем не менее, если все, что требуется — это небольшой контейнер DI, то PicoContainer будет хорошим вариантом, но поскольку пакеты Spring для контейнера DI отделены от остальной части платформы, вы можете легко воспользоваться им и сохранить гибкость на будущее.

### Контейнер JEE 7

Как упоминалось ранее, концепция DI получила широкое распространение и также реализована JCP. Когда вы разрабатываете приложение для серверов приложений, совместимых с JEE 7 (JSR-342), стандартные приемы DI можно применять на всех уровнях.

## Резюме

В этой главе мы представили высокоуровневый обзор платформы Spring Framework с обсуждением всех ее основных возможностей, а также указали на те части книги, где эти возможности рассматриваются более подробно. После чтения этой главы вы должны понимать, что платформа Spring может предложить, осталось лишь увидеть, *как* она это делает.

В следующей главе мы посмотрим, что необходимо для построения и запуска базового Spring-приложения. Мы покажем, как получить Spring Framework, а также обсудим варианты упаковки, тестовый комплект и документацию. Кроме того, в главе 2 будет представлен некоторый базовый код Spring, включая вездесущий пример "Hello World!", реализованный в контексте DI.

# ГЛАВА 2

# Начало работы

Часто самым сложным аспектом при освоении нового инструмента разработки является выяснение того, с чего следует начать. Как правило, проблема усугубляется, если инструмент предлагает слишком много опций, как это делает Spring. К счастью, начало работы с платформой Spring в действительности не так трудно, если знать, где и что искать в первую очередь. В этой главе приведены все основные сведения о том, как выйти на старт. В частности, будут рассмотрены следующие вопросы.

- **Получение Spring.** Первый логический шаг заключается в получении и в сборке JAR-файлов Spring. Если вы хотите сделать все быстро, воспользуйтесь фрагментами кода для управления зависимостями в своей системе сборки согласно примерам, предоставленным по адресу `http://projects.spring.io/spring-framework`. Однако если вы хотите находиться на переднем крае разработки с помощью Spring, поищите последнюю версию исходного кода в репозитории GitHub для Spring (`http://github.com/spring-projects/spring-framework`).
- **Варианты упаковки Spring.** Упаковка Spring является модульной; разрешено выбирать компоненты, которые должны использоваться в приложении, и при распространении готового приложения включать только эти компоненты. Платформа Spring содержит много модулей, но вам понадобится только их подмножество, которое зависит от нужд приложения. Каждый модуль имеет свой скомпилированный двоичный код в JAR-файле наряду с соответствующей документацией Javadoc и исходными JAR-файлами.
- **Руководства Spring.** Новый веб-сайт Spring включает раздел Guides (Руководства), находящийся по адресу `www.spring.io/guides`. Под руководствами понимаются быстрые практические инструкции по построению начального примера для любой задачи разработки с помощью Spring. Эти руководства также отражают последние выпуски проектов и технологий Spring, предоставляя в ваше распоряжение наиболее актуальные примеры.
- **Тестовый комплект и документация.** Одним из предметов особой гордости членов сообщества Spring является всеобъемлющий тестовый комплект и набор документации. Тестирование занимает значительную часть работы команды. Набор документации, входящий в состав стандартного дистрибутива, также великолепен.

- **Пример приложения "Hello World!" в Spring.** Как бы то ни было, мы считаем, что наилучшим способом начала работы с любым новым инструментом, предназначенным для программирования, является написание какого-то кода. Мы представим простой пример, который является полноценной реализацией, основанной на DI, хорошо известного приложения "Hello World!". Не переживайте, если с первого раза не поймете весь его код; далее в книге будут приведены исчерпывающие пояснения.

Если вы уже знакомы с основами Spring Framework, можете переходить прямо к главе 3, где рассматривается реализация IoC и DI в Spring. Тем не менее, даже зная основы Spring, вы наверняка найдете в настоящей главе интересные сведения, особенно касающиеся упаковки и зависимостей.

# Получение Spring Framework

Прежде чем вы сможете приступить к разработке с помощью Spring, необходимо получить код самой платформы. Для этого есть две возможности: воспользоваться своей системой сборки для установки желаемых модулей либо извлечь и построить код из репозитория Spring в GitHub. Применение инструмента для управления зависимостями, такого как Maven или Gradle, часто представляет собой самый простой подход, поскольку все, что необходимо сделать — это объявить зависимость в конфигурационном файле и позволить инструменту получить требуемые библиотеки самостоятельно.

## Быстрое начало

Зайдите на страницу проекта Spring Framework (`http://projects.spring.io/spring-framework`), чтобы получить фрагмент кода управления зависимостями для системы сборки, который позволит включить в ваш проект последнюю версию RELEASE платформы Spring. Можно также использовать промежуточные образы предстоящих выпусков или предыдущие версии платформы.

## Извлечение Spring из GitHub

Если вы хотите иметь доступ к новым средствам еще до того, как они будут отражены в образах, можете извлечь исходный код непосредственно из репозитория GitHub в Pivotal. Для получения последней версии кода Spring сначала установите средство Git, которое можно загрузить по адресу `http://git-scm.com/`, затем откройте окно командной строки и введите следующую команду:

```
git clone git://github.com/spring-projects/spring-framework.git
```

В корневой папке проекта находится файл `README.md` с полным описанием сборки платформы на основе исходного кода.

# Упаковка Spring

*Модуль Spring* — это просто JAR-файл, в котором упакован код, требуемый для данного модуля. После того, как вы поймете назначение каждого модуля, вы сможете выбрать модули, необходимые для проекта, и включить их в свой код.

## Модули Spring

В версии 4.0.2.RELEASE платформа Spring поступает с 20 модулями, упакованными в 20 файлов JAR. Эти JAR-файлы и соответствующие им модули описаны в табл. 2.1. Действительный формат имен JAR-файлов выглядит подобно `spring-aop-4.0.2.RELEASE.jar`, но для простоты в табл. 2.1 приводится только часть, которая специфична для модуля (вроде `aop`).

**Таблица 2.1. Модули Spring**

| JAR-файл | Описание |
|---|---|
| `aop` | Этот модуль содержит все классы, необходимые для применения в приложении средств аспектно-ориентированного программирования (АОП) из Spring. Этот JAR-файл также должен быть включен в приложение, если планируется работать с другими использующими АОП средствами Spring, такими как декларативное управление транзакциями. Кроме того, в этот модуль упакованы и классы, поддерживающие интеграцию с AspectJ |
| `aspects` | Этот модуль содержит все классы, предназначенные для расширенной интеграции с библиотекой АОП в AspectJ. Например, данный модуль понадобится, если вы применяете Java-классы для своей конфигурации Spring и нуждаетесь в управлении транзакциями с помощью аннотаций в стиле AspectJ |
| `beans` | Этот модуль содержит все классы, предназначенные для поддержки манипуляций с бинами Spring. Большинство классов здесь поддерживают реализацию фабрики бинов Spring. Например, в этот модуль упакованы классы, требуемые для обработки XML-файла конфигурации Spring и Java-аннотаций |
| `context` | Этот модуль содержит классы, которые предоставляют многие расширения для ядра Spring. Вы увидите, что все классы должны использовать средство `ApplicationContext` из Spring (описанное в главе 5), а также классы для интеграции с EJB, Java Naming and Directory Interface (JNDI) и Java Management Extensions (JMX). В этом модуле также содержатся удаленные классы Spring, классы для интеграции с языками динамических сценариев (например, JRuby, Groovy, BeanShell), классы API-интерфейса Beans Validation (JSR-303), классы для планирования и выполнения задач и т.д. |
| `context-support` | Этот модуль содержит дополнительные расширения для модуля `spring-context`. На стороне пользовательского интерфейса имеются классы для поддержки электронной почты и интеграции с механизмами шаблонов, такими как Velocity, FreeMarker и JasperReports. Кроме того, здесь упакованы классы для интеграции с различными библиотеками выполнения и планирования задач, в числе которых CommonJ и Quartz |
| `core` | Это основной модуль, необходимый для каждого приложения Spring. В данном JAR-файле находятся все классы, которые совместно используются всеми остальными модулями Spring (например, классы для доступа к файлам конфигурации). В этом JAR-файле вы также найдете наборы исключительно полезных служебных классов, которые применяются по всей кодовой базе Spring и которые можно использовать в собственных приложениях |
| `expression` | Этот модуль содержит все классы поддержки для языка SpEL (Spring Expression Language — язык выражений Spring) |
| `instrument` | Этот модуль включает агент инструментирования Spring для начальной загрузки виртуальной машины Java (Java Virtual Machine — JVM). Данный JAR-файл обязателен для использования в приложении Spring привязывания во время загрузки AspectJ |

*Окончание табл. 2.1*

| JAR-файл | Описание |
|---|---|
| instrument-tomcat | Этот модуль включает агент инструментирования Spring для начальной загрузки JVM на сервере Tomcat |
| jdbc | Этот модуль включает все классы, предназначенные для поддержки JDBC. Данный модуль необходим для всех приложений, которым требуется доступ к базам данных. В модуль упакованы классы для поддержки источников данных, типов данных JDBC, шаблонов JDBC, собственных подключений JDBC и т.д. |
| jms | Этот модуль включает все классы, предназначенные для поддержки JMS |
| messaging | Этот модуль содержит ключевые абстракции, заимствованные из проекта Spring Integration, которые служат основой для приложений на базе сообщений, и добавляет поддержку сообщений STOMP |
| orm | Этот модуль расширяет стандартный набор средств JDBC платформы Spring поддержкой популярных инструментов ORM, в числе которых Hibernate, JDO, JPA и средство отображения данных iBATIS. Многие классы в данном JAR-файле зависят от классов, содержащихся в JAR-файле spring-jdbc, поэтому вы определенно должны включать его в свое приложение |
| oxm | Этот модуль предоставляет поддержку OXM (Object/XML Mapping — отображение объектов на XML). В данный модуль упакованы классы, предназначенные для абстрагирования маршализации и демаршализации XML, а также для поддержки популярных инструментов, таких как Castor, JAXB, XMLBeans и XStream |
| test | Как упоминалось ранее, Spring предоставляет набор имитированных классов для помощи в тестировании приложений. Многие из этих имитированных классов используются внутри тестового комплекта Spring, так что они хорошо проверены и существенно упрощают тестирование разрабатываемых приложений. Безусловно, в модульных тестах для веб-приложений интенсивно применяются имитированные классы HttpServletRequest и HttpServletResponse. С другой стороны, Spring обеспечивает тесную интеграцию с инфраструктурой модульного тестирования JUnit, и в этом модуле предоставлены многие классы, которые поддерживают разработку тестовых сценариев JUnit; например, класс SpringJUnit4ClassRunner предлагает простой способ начальной загрузки ApplicationContext в среду модульного тестирования |
| tx | Этот модуль предоставляет все классы, предназначенные для поддержки инфраструктуры транзакций Spring. Здесь вы найдете классы из уровня абстракции транзакций, поддерживающие Java Transaction API (JTA) и интеграцию с серверами приложений от ведущих производителей |
| web | Этот модуль содержит основные классы для использования Spring в веб-приложениях, в том числе классы для автоматической загрузки средства ApplicationContext, классы для поддержки загрузки файлов и набор полезных классов для выполнения повторяющихся задач, таких как извлечение целочисленных значений из строки запроса |
| webmvc | Этот модуль содержит все классы для собственной инфраструктуры MVC платформы Spring. В случае если применяется отдельная инфраструктура MVC, классы из этого JAR-файла не нужны. Spring MVC рассматривается более подробно в главе 16 |
| web-portlet | Этот модуль предоставляет поддержку использования Spring MVC при разработке портлетов для развертывания в среде сервера портала |
| websocket | Этот модуль предоставляет поддержку Java API для WebSocket (JSR-356) |

**На заметку!** Явная зависимость от модуля ASM больше не требуется, т.к. он теперь упакован вместе с ядром Spring.

## Выбор модулей для разрабатываемого приложения

Без инструмента управления зависимостями, подобного Maven или Gradle, выбор модулей для использования в приложении может оказаться затруднительным. Например, если требуется только фабрика бинов Spring и поддержка DI, по-прежнему необходимы такие модули, как `spring-core`, `spring-beans`, `spring-context` и `spring-aop`. Если нужна поддержка веб-приложений Spring, потребуется также добавить модуль `spring-web` и т.д. Благодаря возможностям инструментов сборки, таким как поддержка транзитивных зависимостей Maven, все обязательные библиотеки третьих сторон будут включены автоматически.

## Доступ к модулям Spring в репозитории Maven

Созданный Apache Software Foundation, проект Maven (`http://maven.apache.org`) стал одним из наиболее популярных инструментов управления зависимостями для Java-приложений, которые охватывают как среды с открытым кодом, так и корпоративные среды.

Maven представляет собой мощный инструмент для сборки, упаковки и управления зависимостями приложений. Он поддерживает полный цикл сборки приложения, начиная с обработки ресурсов и компиляции и заканчивая тестированием и упаковкой. Вдобавок существует большое количество подключаемых модулей Maven для решения разнообразных задач, таких как обновление баз данных и развертывание упакованного приложения на специфическом сервере (например, Tomcat, JBoss или WebSphere).

Практически все проекты с открытым кодом поддерживают распространение своих библиотек через репозиторий Maven. Наиболее популярным является репозиторий Maven Central, размещенный на сервере apache.org; на веб-сайте Maven Central (`http://search.maven.org`) можно выполнять поиск на предмет существования артефакта и получать связанную с ним информацию. После загрузки и установки Maven на машине разработки автоматически появляется доступ к репозиторию Maven Central. Ряд других сообществ открытого кода (например, JBoss и Spring от Pivotal) также предоставляют своим пользователям доступ к собственным репозиториям Maven. Тем не менее, для доступа к таким репозиториям их придется добавить в файл настроек Maven или файл POM (Project Object Model — объектная модель проекта) проекта.

Детальное обсуждение Maven выходит за рамки настоящей книги, но вы всегда можете обратиться к онлайновой документации или книгам, которые предоставят подробные сведения о Maven. Однако по причине настолько широкого распространения Maven полезно упомянуть структуру упаковки Spring в репозитории Maven.

С каждым артефактом Maven связан идентификатор группы, идентификатор артефакта, тип упаковки и версия. Например, для `log4j` идентификатором группы является `log4j`, идентификатором артефакта — `log4j` и типом упаковки — `jar`. Далее следует версия. Скажем, для версии 1.2.16 файл артефакта будет иметь имя `log4j-1.2.16.jar` и располагаться в папке для конкретного идентификатора группы, идентификатора артефакта и версии.

# Использование документации Spring

Один из аспектов Spring, который делает ее настолько полезной платформой для разработчиков, строящих реальные приложения, связан с изобилием хорошо написанной и точной документации. В каждом выпуске Spring Framework команда, отвечающая за подготовку документации, прилагает все усилия, чтобы документация была исчерпывающей и согласованной с командой разработки. Это значит, что каждое функциональное средство Spring не только полностью документировано в Javadoc, но также описано в справочном руководстве, включаемом в каждый выпуск. Если вы еще не знакомы с документацией Spring Javadoc и справочным руководством, самое время сделать это. Эта книга не является заменой какого-либо из упомянутых ресурсов; напротив, она призвана служить дополняющим справочным пособием, демонстрирующим построение приложений Spring с самого начала.

# Помещение Spring в приложение "Hello World!"

К этому моменту вы уже должны понимать, что Spring является серьезным, хорошо поддерживаемым проектом, который обладает всем необходимым для того, чтобы быть великолепным инструментом для разработки приложений. Тем не менее, мы пока еще не приводили какой-либо код. Наверняка вы с нетерпением ждете того момента, когда платформа Spring будет продемонстрирована в действии, и поскольку это невозможно сделать, не прибегая к коду, давайте займемся этим сейчас. Не переживайте, если не полностью поймете код, приводимый в этом разделе; по мере продвижения по материалу настоящей книги будут представлены дополнительные подробности.

## Построение примера приложения "Hello World!"

Вы определенно должны быть знакомы с традиционным примером "Hello World!", но на всякий случай в листинге 2.1 приведен код его Java-версии во всей своей красе.

**Листинг 2.1. Типовой пример "Hello World!"**

```
package com.apress.prospring4.ch2;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Этот пример очень прост — он выполняет работу, но не является особо расширяемым. Что если необходимо изменить сообщение? А что если нужно выводить сообщение разными способами, возможно, в стандартный вывод ошибок вместо стандартного вывода, или заключить его в HTML-дескрипторы, а не представлять как простой текст?

Мы собираемся переопределить требования для этого примера приложения и укажем, что должен поддерживаться простой и гибкий механизм изменения выво-

димого сообщения, а также несложный способ изменения поведения визуализации. В базовом примере "Hello World!" оба изменения можно произвести быстро и легко, внеся соответствующие правки в код. Однако более крупное приложение требует времени на перекомпиляцию и повторное тестирование. Поэтому более удачное решение предусматривает вынесение содержимого сообщения наружу и его чтение во время выполнения, возможно из аргументов командной строки (листинг 2.2).

**Листинг 2.2. Использование аргументов командной строки в "Hello World!"**

```
package com.apress.prospring4.ch2;

public class HelloWorldWithCommandLine {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.out.println(args[0]);
        } else {
            System.out.println("Hello World!");
        }
    }
}
```

В этом примере достигнуто то, чего мы хотели — теперь можно изменять сообщение, не меняя код. Однако в данном приложении по-прежнему остается проблема: компонент, отвечающий за визуализацию сообщения, отвечает также и за его получение. Изменение способа получения сообщения означает изменение кода визуализации. Добавьте к этому тот факт, что мы все еще не можем легко изменить код визуализации; для этого понадобится изменять класс, который запускает приложение.

В плане дальнейшего совершенствования приложения следует отметить, что лучшее решение предполагает проведение рефакторинга с целью вынесения логики визуализации и логики получения сообщений в отдельные компоненты. Кроме того, чтобы сделать приложение действительно гибким, эти компоненты должны реализовывать интерфейсы, с использованием которых будут определяться взаимозависимости между компонентами и запускающим классом.

За счет рефакторинга логики получения сообщений мы можем определить простой интерфейс `MessageProvider` с единственным методом `getMessage()`, как показано в листинге 2.3.

**Листинг 2.3. Интерфейс `MessageProvider`**

```
package com.apress.prospring4.ch2;

public interface MessageProvider {
    String getMessage();
}
```

В листинге 2.4 интерфейс `MessageRenderer` реализован всеми компонентами, которые могут визуализировать сообщения.

**Листинг 2.4. Интерфейс `MessageRenderer`**

```
package com.apress.prospring4.ch2;

public interface MessageRenderer {
    void render();
    void setMessageProvider(MessageProvider provider);
    MessageProvider getMessageProvider();
}
```

Как видите, интерфейс `MessageRenderer` имеет метод `render()` и также метод `setMessageProvider()` в стиле JavaBean. Любые реализации `MessageRenderer` отделены от получения сообщений и делегируют ответственность за это интерфейсу `MessageProvider`, с которым они поставляются. Здесь `MessageProvider` представляет собой зависимость от `MessageRenderer`. Создавать простые реализации этих интерфейсов довольно легко (листинг 2.5).

**Листинг 2.5. Класс `HelloWorldMessageProvider`**

```
package com.apress.prospring4.ch2;

public class HelloWorldMessageProvider implements MessageProvider {
    @Override
    public String getMessage() {
        return "Hello World!";
    }
}
```

В листинге 2.5 мы создаем простой интерфейс `MessageProvider`, который всегда возвращает в качестве сообщения строку "Hello World!". Класс `StandardOutMessageRenderer` (приведенный в листинге 2.6) также прост.

**Листинг 2.6. Класс `StandardOutMessageRenderer`**

```
package com.apress.prospring4.ch2;

public class StandardOutMessageRenderer implements MessageRenderer {
    private MessageProvider messageProvider;

    @Override
    public void render() {
        if (messageProvider == null) {
            throw new RuntimeException(
                "You must set the property messageProvider of class:"
                + StandardOutMessageRenderer.class.getName());
                // Вы должны установить свойство messageProvider класса
        }

        System.out.println(messageProvider.getMessage());
    }

    @Override
    public void setMessageProvider(MessageProvider provider) {
        this.messageProvider = provider;
    }
```

```
    @Override
    public MessageProvider getMessageProvider() {
        return this.messageProvider;
    }
}
```

Теперь осталось только переписать метод `main()` входного класса, как показано в листинге 2.7.

**Листинг 2.7. Приложение "Hello World!" после рефакторинга**

```
package com.apress.prospring4.ch2;

public class HelloWorldDecoupled {
    public static void main(String[] args) {
        MessageRenderer mr = new StandardOutMessageRenderer();
        MessageProvider mp = new HelloWorldMessageProvider();
        mr.setMessageProvider(mp);
        mr.render();
    }
}
```

Приведенный код довольно прост: мы создаем экземпляры `HelloWorldMessageProvider` и `StandardOutMessageRenderer`, хотя объявленными типами являются, соответственно, `MessageProvider` и `MessageRenderer`. Причина в том, что нам нужно взаимодействовать в коде только с методами, предоставленными этими интерфейсами, а классы `HelloWorldMessageProvider` и `StandardOutMessageRenderer` уже реализуют эти интерфейсы. Затем мы передаем `MessageProvider` экземпляру `MessageRenderer` и вызываем метод `MessageRenderer.render()`. Скомпилировав и запустив эту программу, мы получим вполне ожидаемый вывод строки "Hello World!".

Теперь пример больше похож на то, к чему мы стремимся, но осталась одна небольшая проблема. Изменение реализации либо интерфейса `MessageRenderer`, либо интерфейса `MessageProvider` означает модификацию кода. Чтобы обойти данную проблему, мы можем создать простой фабричный класс, который читает имена классов реализации из файла свойств и создает их экземпляры в интересах приложения (листинг 2.8).

**Листинг 2.8. Класс `MessageSupportFactory`**

```
package com.apress.prospring4.ch2;

import java.io.FileInputStream;
import java.util.Properties;

public class MessageSupportFactory {
    private static MessageSupportFactory instance;

    private Properties props;
    private MessageRenderer renderer;
    private MessageProvider provider;

    private MessageSupportFactory() {
        props = new Properties();
```

```
        try {
            props.load(new FileInputStream(
                "com/apress/prospring4/ch2/msf.properties"));

            String rendererClass = props.getProperty("renderer.class");
            String providerClass = props.getProperty("provider.class");

            renderer = (MessageRenderer)
                Class.forName(rendererClass).newInstance();
            provider = (MessageProvider)
                Class.forName(providerClass).newInstance();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    static {
        instance = new MessageSupportFactory();
    }

    public static MessageSupportFactory getInstance() {
        return instance;
    }

    public MessageRenderer getMessageRenderer() {
        return renderer;
    }

    public MessageProvider getMessageProvider() {
        return provider;
    }
}
```

Приведенная здесь реализация элементарна и несколько наивна, обработка ошибок упрощена, а имя конфигурационного файла жестко закодировано, но мы уже имеем значимый объем кода. Конфигурационный файл для этого класса очень прост:

```
renderer.class=com.apress.prospring4.ch2.StandardOutMessageRenderer
provider.class=com.apress.prospring4.ch2.HelloWorldMessageProvider
```

Осталось внести небольшое изменение в метод `main()`, как показано в листинге 2.9.

**Листинг 2.9. Использование `MessageSupportFactory`**

```
package com.apress.prospring4.ch2;

public class HelloWorldDecoupledWithFactory {
    public static void main(String[] args) {
        MessageRenderer mr = MessageSupportFactory.getInstance().
getMessageRenderer();
        MessageProvider mp = MessageSupportFactory.getInstance().
getMessageProvider();
        mr.setMessageProvider(mp);
        mr.render();
    }
}
```

Перед тем как переходить к добавлению Spring в это приложение, давайте вспомним, что было сделано. Начав с простого приложения "Hello World!", мы определили два дополнительных требования, которым приложение должно удовлетворять. Первое требование: изменение сообщения должно осуществляться просто. Второе требование: изменение механизма визуализации также должно быть простым. Для удовлетворения этих требований мы ввели два интерфейса: `MessageProvider` и `MessageRenderer`. В плане возможности получения сообщения для визуализации интерфейс `MessageRenderer` полагается на реализацию интерфейса `MessageProvider`. Наконец, мы добавили простой фабричный класс для извлечения имен классов реализации и создания их экземпляров должным образом.

### *Рефакторинг с использованием Spring*

Показанный ранее окончательный пример соответствует целям, намеченным для нашего приложения, но с ним по-прежнему связаны проблемы. Первая проблема состоит в том, что мы должны написать немалый объем связующего кода для соединения всех частей в одно приложение, одновременно сохраняя компоненты слабо связанными. Вторая проблема заключается в том, что мы все еще должны вручную предоставлять реализацию `MessageRenderer` с экземпляром реализации `MessageProvider`. С применением Spring мы можем решить обе эти проблемы.

Для решения проблемы со слишком большим объемом связующего кода мы можем полностью удалить класс `MessageSupportFactory` из приложения и заменить его интерфейсом Spring по имени `ApplicationContext`. Не переживайте особо по поводу этого интерфейса; пока вполне достаточно знать, что он используется Spring для сохранения всей информации о среде, относящейся к приложению, которое управляется Spring. Данный интерфейс расширяет другой интерфейс, `ListableBeanFactory`, который действует в качестве поставщика для любого экземпляра бинов, управляемых Spring (листинг 2.10).

**Листинг 2.10. Использование интерфейса `ApplicationContext` из Spring**

```
package com.apress.prospring4.ch2;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class HelloWorldSpringDI {
    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext
          ("META-INF/spring/app-context.xml");

        MessageRenderer mr = ctx.getBean("renderer", MessageRenderer.class);
        mr.render();
    }
}
```

В листинге 2.10 вы видите, что метод `main()` получает экземпляр `ClassPathXmlApplicationContext` (информация о конфигурации приложения загружается из файла `META-INF/spring/app-context.xml`, расположенного в пути классов проекта), типизированный как `ApplicationContext`, и получает из него экземпляры `MessageRenderer` с применением метода `ApplicationContext.getBean()`. Сейчас не стоит беспокоиться о методе `getBean()`; достаточно помнить, что он

читает конфигурацию приложения (в рассматриваемом случае из XML-файла), инициализирует среду ApplicationContext из Spring и затем возвращает сконфигурированный экземпляр бина. Этот XML-файл (app-context.xml) служит тем же целям, что и аналогичный файл, который используется для MessageSupportFactory (листинг 2.11).

**Листинг 2.11. Конфигурация приложения в Spring, представленная с помощью XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="provider"
         class="com.apress.prospring4.ch2.HelloWorldMessageProvider"/>
   <bean id="renderer"
         class="com.apress.prospring4.ch2.StandardOutMessageRenderer"
         p:messageProvider-ref="provider"/>
</beans>
```

В листинге 2.11 приведена типичная конфигурация ApplicationContext в Spring. Сначала объявляется пространство имен Spring, и стандартным пространством имен является beans. Пространство имен beans применяется для объявления бинов, которые должны управляться Spring, а также их требований к зависимостям (в предыдущем примере свойство messageProvider бина renderer ссылается на бин provider) для Spring, чтобы распознать и внедрить эти зависимости.

Далее объявляется бин с идентификатором provider и соответствующий класс реализации. Когда платформа Spring видит это определение бина во время инициализации ApplicationContext, она создает экземпляр класса и сохраняет его с указанным идентификатором.

Затем объявляется бин renderer с соответствующим классом реализации. Вспомните, что при получении сообщения для визуализации этот бин полагается на интерфейс MessageProvider. Чтобы информировать Spring о таком требовании DI, мы используем атрибут пространства имен p. Атрибут дескриптора p:messageProvider-ref="provider" сообщает Spring, что свойство messageProvider бина должно быть внедрено с помощью другого бина. Внедряемый в это свойство бин должен иметь идентификатор provider. Когда платформа Spring встречает это определение, она создает экземпляр класса, находит в бине свойство по имени messageProvider и внедряет его с применением экземпляра бина, имеющего идентификатор provider.

Как видите, во время инициализации ApplicationContext метод main() теперь просто получает бин MessageRenderer с использованием безопасного к типам метода getBean() (передавая ему идентификатор и ожидаемый возвращаемый тип, которым является интерфейс MessageRenderer) и вызывает render(); Spring создает реализацию MessageProvider и внедряет ее в реализацию MessageRenderer. Обратите внимание, что мы не должны вносить какие-либо изменения в классы, которые связаны вместе с применением Spring. На самом деле эти классы не имеют никакого отношения к платформе Spring и совершенно не обращают внимания на ее

существование. Тем не менее, это не всегда так. Ваши классы могут реализовывать интерфейсы Spring для взаимодействия с контейнером DI разнообразными путями.

Давайте посмотрим, как действуют новая конфигурация Spring и модифицированный метод `main()`. Используя Maven, введите следующую команду в своем терминале, чтобы собрать проект и корень вашего исходного кода:

```
mvn clean package dependency:copy-dependencies
```

Единственным обязательным модулем Spring, который должен быть объявлен в Maven POM, является `spring-context`. Инструмент Maven автоматически удовлетворит любые транзитивные зависимости, требуемые для данного модуля. Цель `dependency:copy-dependencies` скопирует все обязательные зависимости в подкаталог по имени `dependency` внутри каталога `target`. Это значение пути будет также применяться в качестве префикса для библиотечных файлов, добавляемых в `MANIFEST.MF` при построении JAR-файла. Если вы незнакомы с конфигурацией сборки JAR-файлов и самим процессом в Maven, просмотрите исходный код для этой главы (доступный для загрузки на веб-сайте издательства), в особенности файл `pom.xml`. Наконец, чтобы запустить пример реализации Spring DI, введите следующие команды:

```
cd target ; java -jar hello-world-4.0-SNAPSHOT.jar
```

После этого вы должны увидеть несколько сообщений, генерируемых процессом начального запуска контейнера Spring, а за ними наш ожидаемый вывод "Hello World!".

---

**На заметку!** Некоторые интерфейсы и классы, определенные в примере "Hello World", могут использоваться в последующих главах. Хотя в данном примере мы привели исходный код полностью, в других главах могут быть показаны сжатые версии этого кода, чтобы соблюсти краткость особенно в тех случаях, когда в код вносятся дополнительные модификации. Кроме того, в целях демонстрации код помещен в плоскую пакетную структуру, тогда как в реальном приложении он должен быть организован соответствующим образом.

---

# Резюме

В этой главе мы представили основные сведения, необходимые для подготовки и запуска Spring. Было показано, как начать работу со Spring посредством инструментов управления зависимостями и получить текущую разрабатываемую версию прямо из GitHub. Мы описали, каким образом упакована платформа Spring, и перечислили зависимости, необходимые для ее функциональных средств. Располагая этими сведениями, вы можете принимать обоснованные решения о том, какие JAR-файлы Spring нужны для приложения и какие зависимости должны распространяться вместе с приложением. Документация Spring, руководства и тестовый комплект обеспечивают пользователям Spring идеальную основу для начала разработки в Spring, поэтому мы посвятили определенное время на анализ того, что именно делается доступным благодаря Spring. Наконец, мы рассмотрели пример применения Spring DI, в котором традиционное приложение "Hello World!" было превращено в слабо связанное и расширяемое приложение визуализации сообщений.

Важно понимать, что в этой главе мы только слегка коснулись поверхности Spring DI и в целом платформы Spring. В следующей главе мы обсудим IoC и DI в Spring.

ГЛАВА 3

# Введение в IoC и DI в Spring

В главе 2 были раскрыты базовые принципы инверсии управления (Inversion of Control — IoC) и внедрения зависимостей (Dependency Injection — DI). С практической точки зрения DI является специализированной формой IoC, хотя вы часто будете обнаруживать, что эти два термина используются взаимозаменяемо. В данной главе мы более подробно рассмотрим IoC и DI, формализуя отношения между этими двумя концепциями и демонстрируя, каким образом платформа Spring вписывается в общую картину.

После определения IoC и DI и их взаимоотношений с платформой Spring мы рассмотрим концепции, которые являются существенными для реализации DI в Spring. В этой главе раскрываются только основы реализации DI в Spring; мы обсудим более сложные возможности DI в главе 4. Итак, в настоящей главе рассматриваются следующие темы.

- **Концепции инверсии управления.** В этой части описаны разновидности IoC, включая пассивную инверсию зависимостей, или внедрение зависимостей (Dependency Injection), и активную инверсию зависимостей, или поиск зависимостей (Dependency Lookup). Здесь мы объясним отличия между разными подходами IoC и представим доводы за и против для каждого из них.
- **Инверсия управления в Spring.** В этой части рассмотрены возможности IoC, доступные в Spring, и показано, как они реализованы. В частности, вы увидите службы Dependency Injection, предлагаемые Spring, включая Setter Injection (Внедрение через метод установки), Constructor Injection (Внедрение через конструктор) и Method Injection (Внедрение через метод).
- **Внедрение зависимостей в Spring.** В этой части раскрывается реализация контейнера IoC в Spring. Для определения бинов и требований DI главным интерфейсом, с которым будет взаимодействовать приложение, является `BeanFactory`. Однако за исключением нескольких начальных примеров все остальные примеры кода в этой главе будут сосредоточены на использовании интерфейса `ApplicationContext` платформы Spring, который представляет собой расширение `BeanFactory` и обеспечивает намного более мощные возможности. Мы объясним отличие между `BeanFactory` и `ApplicationContext` в последующих разделах.

- **Конфигурирование контекста приложения Spring.** Финальная часть этой главы посвящена применению двух подходов к конфигурированию `ApplicationContext` — на основе XML и с помощью аннотаций. Конфигурация Groovy и Java более подробно обсуждаются в главе 4. Сначала мы рассмотрим конфигурацию DI, а затем перейдем к описанию дополнительных служб, предоставляемых `BeanFactory`, таких как наследование, управление жизненным циклом и автосвязывание.

# Инверсия управления и внедрение зависимостей

По своей сути IoC и, следовательно, DI направлены на то, чтобы предложить простой механизм для предоставления зависимостей компонента (часто называемых *коллабораторами* объекта) и управления этими зависимостями на протяжении всего их жизненного цикла. Компонент, который требует определенных зависимостей, часто называют *зависимым объектом* или, в случае IoC, *целевым объектом*. Вообще говоря, инверсия управления может быть разделена на два подтипа: внедрение зависимостей (Dependency Injection) и поиск зависимостей (Dependency Lookup). Эти подтипы в дальнейшем подразделяются на конкретные реализации служб IoC. В этом определении можно ясно видеть, что когда речь идет о DI, мы всегда говорим об IoC, но когда речь ведется об IoC, не всегда имеется в виду DI (например, Dependency Lookup — это также форма IoC).

# Типы инверсии управления

Вас может интересовать, почему существуют два типа IoC, которые к тому же дополнительно разделены на различные реализации. Похоже, что четкий ответ на этот вопрос отсутствует; конечно, разные типы обеспечивают определенный уровень гибкости, но нам кажется, что IoC — в большей степени смесь старых и новых идей; это и представляют два типа IoC.

Тип *Dependency Lookup* является намного более традиционным подходом и на первый взгляд выглядит более знакомым Java-программистам. Тип *Dependency Injection*, хотя поначалу кажется нелогичным, в действительности обеспечивает более высокую гибкость и удобство в использовании по сравнению с Dependency Lookup.

В случае применения IoC в стиле Dependency Lookup компонент должен получить ссылку на зависимость, тогда как в стиле Dependency Injection зависимости внедряются в компонент контейнером IoC. Вариант Dependency Lookup доступен в виде двух типов: Dependency Pull (Извлечение зависимостей) и Contextualized Dependency Lookup (Контекстуализированный поиск зависимостей), или CDL. Вариант Dependency Injection также имеет две разновидности: Constructor Dependency Injection (Внедрение зависимостей через конструктор) и Setter Dependency Injection (Внедрение зависимостей через метод установки).

---

**На заметку!** При обсуждениях в этом разделе мы не заботимся о том, каким образом вымышленный контейнер IoC узнает обо всех разных зависимостях, а только о том, что в определенной точке он выполняет действия, описанные для каждого механизма.

---

## Тип Dependency Pull

Для Java-разработчика Dependency Pull является самым узнаваемым типом IoC. В Dependency Pull зависимости извлекаются из реестра по мере необходимости. Любой, кто хотя бы раз писал код для доступа к EJB (2.1 или предшествующих версий), использовал Dependency Pull (т.е. через API-интерфейс JNDI для поиска компонента EJB). На рис. 3.1 показан сценарий Dependency Pull через механизм поиска.

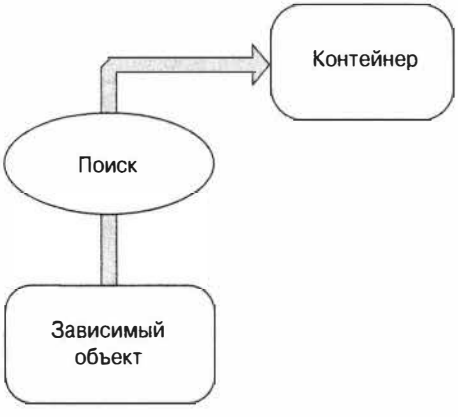


**Рис. 3.1.** Извлечение зависимостей через поиск JNDI

Платформа Spring также предлагает Dependency Pull в качестве механизма для извлечения компонентов, которыми она управляет; вы видели это в действии в главе 2. В листинге 3.1 показан типовой поиск Dependency Pull в приложении, основанном на Spring.

**Листинг 3.1. Dependency Pull в Spring**

```
package com.apress.prospring4.ch3;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class DependencyPull {
    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext
          ("META-INF/spring/app-context.xml");
        MessageRenderer mr = ctx.getBean("renderer", MessageRenderer.class);
        mr.render();
    }
}
```

Этот тип IoC не только преобладает в JEE-приложениях (использующих EJB 2.1 или предшествующие версии), в которых широко применяются поиски JNDI для получения зависимостей из реестра, но является центральным аспектом для работы с платформой Spring во многих средах.

## Тип Contextualized Dependency Lookup

Тип Contextualized Dependency Lookup (CDL) в некоторых отношениях похож на Dependency Pull, но в CDL поиск осуществляется в контейнере, который управляет ресурсом, а не только в каком-то центральном реестре, и обычно он производится в установленной точке. Механизм CDL проиллюстрирован на рис. 3.2.

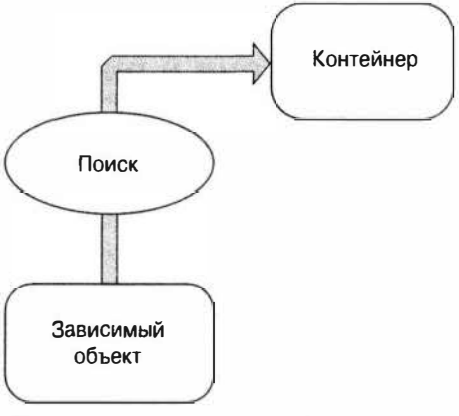


**Рис. 3.2.** Контекстуализированный поиск зависимостей

Тип CDL работает за счет реализации компонентом интерфейса, подобного показанному в листинге 3.2.

**Листинг 3.2. Интерфейс компонента для CDL**

```
package com.apress.prospring4.ch3;

public interface ManagedComponent {
    void performLookup(Container container);
}
```

Реализуя этот интерфейс, компонент сигнализирует контейнер о том, что он желает получить зависимость. Контейнер обычно предоставляется лежащим в основе сервером приложений (например, Tomcat или JBoss) или платформой (скажем, Spring). В листинге 3.3 приведен простой интерфейс `Container`, который предоставляет службу Dependency Lookup.

**Листинг 3.3. Простой интерфейс `Container`**

```
package com.apress.prospring4.ch3;

public interface Container {
    Object getDependency(String key);
}
```

Когда контейнер готов передать зависимости компоненту, он вызывает метод `performLookup()` на каждом компоненте по очереди. Затем компонент может искать свои зависимости с использованием интерфейса `Container`, как показано в листинге 3.4.

**Листинг 3.4. Получение зависимостей в CDL**

```
package com.apress.prospring4.ch3;

public class ContextualizedDependencyLookup implements ManagedComponent {
    private Dependency dependency;

    @Override
    public void performLookup(Container container) {
      this.dependency = (Dependency) container.getDependency("myDependency");
    }

    @Override
    public String toString() {
      return dependency.toString();
    }
}
```

В листинге 3.4 обратите внимание на то, что `Dependency` — это пустой класс.

## Тип Constructor Dependency Injection

Тип Constructor Dependency Injection относится к ситуации, когда зависимости предоставляются компоненту в его конструкторе (или конструкторах). Компонент объявляет конструктор или набор конструкторов, получающих в качестве аргументов его зависимости, и контейнер IoC передает зависимости компоненту при создании его экземпляра (листинг 3.5).

**Листинг 3.5. Внедрение зависимостей через конструктор**

```
package com.apress.prospring4.ch3;

public class ConstructorInjection {
    private Dependency dependency;

    public ConstructorInjection(Dependency dependency) {
        this.dependency = dependency;
    }

    @Override
    public String toString() {
        return dependency.toString();
    }
}
```

## Тип Setter Dependency Injection

В случае Setter Dependency Injection контейнер IoC внедряет зависимости компонента через методы установки в стиле JavaBean. Методы установки компонента отражают зависимости, которыми контейнер IoC может управлять. В листинге 3.6 показан типичный компонент, основанный на Setter Dependency Injection.

**Листинг 3.6. Внедрение зависимостей через метод установки**

```
package com.apress.prospring4.ch3;
public class SetterInjection {
    private Dependency dependency;
    public void setDependency(Dependency dependency) {
        this.dependency = dependency;
    }
    @Override
    public String toString() {
        return dependency.toString();
    }
}
```

Внутри контейнера на требование зависимости, отраженное методом `setDependency()`, ссылаются по имени в стиле JavaBean — `dependency`. На практике Setter Injection является наиболее широко используемым механизмом внедрения, и он относится к одному из простейших в реализации механизмов IoC.

## Выбор между внедрением и поиском

Выбор используемого стиля IoC — внедрение или поиск — не всегда является сложной задачей. Во многих случаях применяемый тип IoC определяется используемым контейнером. Например, в случае EJB 2.1 или предшествующих версий вы должны выбрать IoC в стиле поиска (через JNDI), чтобы получать EJB из контейнера JEE. Помимо первоначальных поисков бинов компоненты и их зависимости в Spring всегда связываются друг с другом с помощью IoC в стиле внедрения.

**На заметку!** При использовании Spring доступ к ресурсам EJB можно производить без необходимости в явном поиске. Spring может действовать в качестве адаптера между системами IoC в стиле поиска и внедрения, таким образом позволяя управлять всеми ресурсами с применением внедрения.

Реальный вопрос заключается в следующем: когда есть выбор, какой метод должен использоваться — внедрение или поиск? Ответ: определенно внедрение. Взглянув на код в листингах 3.4 и 3.5, вы четко увидите, что применение внедрения оказывает нулевое воздействие на код компонентов. С другой стороны, код Dependency Pull должен активно получать ссылку на реестр и взаимодействовать с ним при получении зависимостей, а использование CDL требует от классов реализации специфичного интерфейса и поиска зависимостей вручную. Когда применяется внедрение, все, что классы должны сделать — это позволить зависимостям внедряться либо через конструкторы, либо через методы установки.

В случае внедрения вы можете пользоваться своими классами полностью отдельно от контейнера IoC, который поставляет зависимые объекты с их коллабораторами вручную, тогда как в случае поиска ваши классы всегда зависят от классов и интерфейсов, определенных контейнером. Еще один недостаток поиска связан с чрезмерно трудным тестированием классов в изоляции от контейнера. При использовании внедрения тестирование компонентов становится тривиальным, поскольку вы можете просто предоставить зависимости самостоятельно с помощью подходящего конструктора или метода установки.

**На заметку!** Более полное обсуждение тестирования с применением Dependency Injection и Spring приведено в главе 13.

Решения на основе поиска неизбежно оказываются более сложными, чем решения, основанные на внедрении. Хотя такой сложности не следует бояться, мы ставим под сомнение обоснованность добавления ненужной сложности к процессу, поскольку считаем этот аспект центральным при управлении зависимостями в приложении.

Оставив все второстепенные причины в стороне, главной причиной выбора внедрения, а не поиска, является значительное упрощение работы. При использовании внедрения приходится писать существенно меньше кода, этот код прост и в общем случае может быть автоматизирован с помощью хорошей IDE-среды. Вы заметите, что весь код в примерах внедрения является пассивным в том смысле, что он не пытается активно выполнять какую-то задачу. Самым замечательным моментом, который вы отметите в коде внедрения, будет то, что получаемые объекты хранятся только в полях; при извлечении зависимости из любого реестра или контейнера никакой другой код не задействуется. Таким образом, код оказывается намного более простым и менее подверженным ошибкам. Пассивный код легче сопровождать, чем активный код, поскольку в нем мало что может пойти не так. Взгляните на следующий фрагмент кода из листинга 3.4:

```
public void performLookup(Container container) {
  this.dependency = (Dependency) container.getDependency("myDependency");
}
```

В этом коде многое может пойти не так, как было задумано: ключ зависимости может измениться, экземпляр контейнера может быть `null` или возвращенная зависимость может относиться к неподходящему типу. Мы привели этот код, т.к. он содержит большое количество *перемещающихся частей*, и здесь много чего может быть нарушено. Применение Dependency Lookup может разъединить компоненты приложения, но при этом добавляется сложность в дополнительный код, который требуется для связывания этих компонентов вместе с целью выполнения каких-то полезных задач.

## Выбор между Setter Injection и Constructor Injection

Теперь, когда мы прояснили, какой метод IoC является предпочтительным, осталось выбрать применяемую разновидность внедрения — Setter Injection или Constructor Injection. Тип Constructor Injection особенно удобен, когда перед использованием компонента обязательно должен существовать экземпляр класса зависимости. Многие контейнеры, включая Spring, предоставляют механизм проверки, все ли зависимости определены, который предназначен для случая применения Setter Injection, но за счет использования Constructor Injection вы формально задаете требование для зависимости в независимой от контейнера манере. Кроме того, Constructor Injection помогает достичь применения неизменяемых объектов.

Тип Setter Injection полезен в разнообразных случаях. Если компонент отражает свои зависимости контейнеру, но готов предоставить для них стандартные настройки, то обычно лучшим способом достичь этого является Setter Injection. Еще одно достоинство Setter Injection заключается в том, что данный тип внедрения

позволяет объявлять зависимости в интерфейсе, хотя это не настолько удобно, как может показаться на первый взгляд. Предположим, что есть типичный бизнес-интерфейс с одним бизнес-методом по имени `defineMeaningOfLife()`. Если в дополнение к этому методу вы определите метод установки для внедрения, такой как `setEncyclopedia()`, то тем самым обяжете все реализации использовать или хотя бы учитывать наличие этой зависимости. Тем не менее, вы не должны определять `setEncyclopedia()` в бизнес-интерфейсе. Вместо этого вы можете определить данный метод в классах, реализующих бизнес-интерфейс. При таком подходе все современные контейнеры IoC, в том числе и Spring, могут работать с компонентом в терминах бизнес-интерфейса, но по-прежнему предоставлять зависимости реализованного класса. Несколько прояснить сказанное поможет пример. Взгляните на бизнес-интерфейс, приведенный в листинге 3.7.

**Листинг 3.7. Интерфейс `Oracle`**

```
package com.apress.prospring4.ch3;

public interface Oracle {
    String defineMeaningOfLife();
}
```

Обратите внимание, что этот бизнес-интерфейс не определяет каких-либо методов установки для Dependency Injection. Реализовать такой интерфейс можно так, как показано в листинге 3.8.

**Листинг 3.8. Реализация интерфейса `Oracle`**

```
package com.apress.prospring4.ch3;

public class BookwormOracle implements Oracle {
    private Encyclopedia encyclopedia;

    public void setEncyclopedia(Encyclopedia encyclopedia) {
        this.encyclopedia = encyclopedia;
    }
    @Override
    public String defineMeaningOfLife() {
        return "Encyclopedias are a waste of money - use the Internet";
    }
}
```

Как видите, класс `BookwormOracle` не только реализует интерфейс `Oracle`, но также определяет метод установки для Dependency Injection. Платформа Spring более чем удобна при работе со структурой подобного рода, и нет никакой необходимости определять зависимости в бизнес-интерфейсе. Возможность использовать интерфейсы для определения зависимостей — часто рекламируемое преимущество Setter Injection, однако в действительности вы должны стремиться к применению методов установки исключительно для внедрения из своих интерфейсов. Если только вы не имеете абсолютной уверенности в том, что все реализации отдельного бизнес-интерфейса требуют заданной зависимости, позвольте каждому классу реализации определить собственные зависимости и поддерживать в бизнес-интерфейсе только бизнес-методы.

Хотя вы не всегда должны помещать методы установки для зависимостей в бизнес-интерфейс, добавление в него методов установки и извлечения для параметров конфигурации является удачным решением, которое делает Setter Injection полезным инструментом. Мы рассматриваем параметры конфигурации как специальный случай зависимостей. Безусловно, компоненты зависят от данных конфигурации, однако конфигурационные данные значительно отличаются от тех типов зависимостей, которые вы видели до сих пор. Мы вскоре обсудим отличия, а пока взгляните на бизнес-интерфейс, представленный в листинге 3.9.

**Листинг 3.9. Интерфейс `NewsletterSender`**

```
package com.apress.prospring4.ch3;
public interface NewsletterSender {
    void setSmtpServer(String smtpServer);
    String getSmtpServer();
    void setFromAddress(String fromAddress);
    String getFromAddress();
    void send();
}
```

Интерфейс `NewsletterSender` реализуется классами, которые отправляют по электронной почте набор информационных бюллетеней. Единственным бизнес-методом является `send()`, но обратите внимание на определение в интерфейсе двух свойств JavaBean. Почему это было сделано, ведь только что говорилось о том, что не следует определять зависимости в бизнес-интерфейсе? Причина в том, что эти значения — адрес SMTP-сервера и почтовый адрес, по которому отправляются сообщения — в практическом смысле не являются зависимостями; они представляют собой конфигурационные детали, влияющие на функционирование всех реализаций `NewsletterSender`. Здесь возникает вопрос: в чем отличие параметра конфигурации от любого другого вида зависимости? В большинстве случаев можно легко распознать, должна ли зависимость классифицироваться как параметр конфигурации, но если вы не уверены, то проверьте следующие три характеристики, которые указывают на параметр конфигурации.

- **Параметры конфигурации являются пассивными.** В интерфейсе `NewsletterSender`, показанном в листинге 3.9, параметр SMTP-сервера может служить примером пассивной зависимости. Пассивные зависимости не используются напрямую для выполнения какого-то действия; для запуска своих действий они применяются внутренне или другими зависимостями. В примере `MessageRenderer` из главы 2 зависимость `MessageProvider` не является пассивной — она выполняет функцию, которая была необходима `MessageRenderer` для завершения своей задачи.
- **Параметры конфигурации — это обычно информация, а не другие компоненты.** Под этим мы понимаем, что параметр конфигурации представляет собой определенную порцию информации, которая необходима компоненту для завершения своей работы. Очевидно, что SMTP-сервер — порция информации, требуемая `NewsletterSender`, однако `MessageProvider` — это в действительности другой компонент, который необходим для корректного функционирования `MessageRenderer`.

- **Параметры конфигурации обычно представляют собой простые значения или коллекции простых значений.** В действительности это является побочным продуктом первых двух характеристик, но параметры конфигурации — обычно простые значения. В Java это означает, что они относятся к элементарному типу (или соответствующему классу-оболочке), к типу `String` или к коллекции таких значений. Простые значения, как правило, пассивны. Другими словами, со значениями `String` мало что можно делать кроме манипулирования данными, которые они представляют; почти всегда эти значения используются для информационных целей: например, значение `int` может представлять номер порта для прослушивания сетевым сокетом, а значение `String` — SMTP-сервер, через который будет производиться отправка почтовых сообщений.

При рассмотрении вопроса, определять ли конфигурационные настройки в бизнес-интерфейсе, также подумайте о том, применим ли конкретный параметр конфигурации ко всем реализациям этого бизнес-интерфейса или же только к одной. Например, в случае `NewsletterSender` очевидно, что все реализации должны знать, какой SMTP-сервер использовать для отправки почтовых сообщений. Однако такую конфигурационную настройку, как признак защищенной отправки сообщений, имеет смысл не включать в бизнес-интерфейс, потому что данную возможность поддерживают не все почтовые API-интерфейсы, и вполне уместно предположить, что многие реализации вообще не будут принимать во внимание эту настройку.

---

**На заметку!** Вспомните, что в главе 2 мы решили определять зависимости на уровне бизнес-логики. Это делалось только в целях иллюстрации и ни в коем случае не должно рассматриваться как рекомендуемый подход.

---

Внедрение зависимостей через метод установки также позволяет менять зависимости для различных реализаций на лету, не создавая нового экземпляра родительского компонента. Это становится возможным благодаря поддержке JMX в Spring. Пожалуй, самое значительное преимущество типа Setter Injection в том, что он представляет собой наименее навязчивый механизм внедрения из всех существующих.

Вообще говоря, вы должны выбирать тип внедрения на основе своего сценария использования. Внедрение зависимостей через метод установки позволяет менять местами зависимости, не создавая новые объекты, и также разрешает классу выбирать подходящие стандартные настройки без необходимости в явном внедрении объекта. Внедрение через конструктор будет удачным выбором, когда нужно гарантировать передачу зависимостей компоненту, и в случае проектирования для неизменяемых объектов. Имейте в виду, что в то время как внедрение через конструктор гарантирует предоставление компоненту всех зависимостей, большинство контейнеров предлагают также механизм для обеспечения этого, но ценой может стать привязка вашего кода к инфраструктуре.

# Инверсия управления в Spring

Как упоминалось ранее, инверсия управления — это крупная часть того, делает Spring. Ядро реализации Spring основано на Dependency Injection, хотя также

обеспечиваются и возможности Dependency Lookup. Когда платформа Spring предоставляет коллабораторы зависимому объекту автоматически, она делает это с использованием Dependency Injection. В приложении, основанном на Spring, всегда предпочтительнее применять Dependency Injection для передачи коллабораторов зависимым объектам, а не заставлять зависимые объекты получать коллабораторы через поиск. Механизм Dependency Injection в Spring иллюстрируется на рис. 3.3 (а механизм Dependency Lookup был представлен на рис. 3.2).

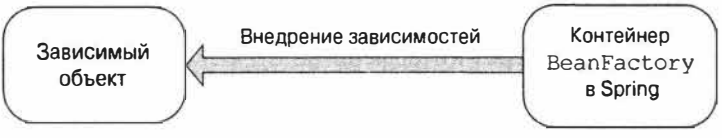


**Рис. 3.3.** Механизм внедрения зависимостей в Spring

Несмотря на то что Dependency Injection является предпочтительным механизмом связывания вместе коллабораторов и зависимых объектов, для доступа к зависимым объектам понадобится Dependency Lookup. Во многих средах Spring не может автоматически связать все компоненты приложения с использованием Dependency Injection, поэтому для доступа к начальному набору компонентов вы должны применять Dependency Lookup. Например, в автономных Java-приложениях необходимо выполнить начальную загрузку контейнера Spring в методе `main()` и получить зависимости (через интерфейс `ApplicationContext`) для программной обработки. Однако при построении веб-приложений с помощью поддержки MVC в Spring этого можно избежать за счет автоматического связывания всего приложения. Всегда, когда возможно использовать Dependency Injection вместе с платформой Spring, это следует делать; в противном случае можно прибегнуть к средствам Dependency Lookup. В этой главе вы увидите примеры обоих подходов в действии, и мы укажем на них, как только они встретятся впервые.

Интересной функцией контейнера IoC в Spring является возможность действовать в качестве адаптера между его собственным контейнером Dependency Injection и внешними контейнерами Dependency Lookup. Эта функция рассматривается далее в главе.

Платформа Spring поддерживает Constructor Injection и Setter Injection, укрепляя стандартный набор средств IoC рядом полезных дополнений и упрощая разработку в целом.

В оставшейся части главы приводятся основы контейнера DI в Spring с множеством иллюстративных примеров.

# Внедрение зависимостей в Spring

Поддержка Dependency Injection в Spring является всеобъемлющей и, как вы увидите в главе 4, выходит за рамки стандартного набора средств IoC, обсуждавшегося до сих пор. В оставшейся части текущей главы рассматриваются основы контейнера Dependency Injection в Spring, в том числе Setter Injection, Constructor Injection и Method Injection, вместе с детальным описанием способов конфигурирования Dependency Injection в Spring.

## Бины и фабрики бинов

Ядром контейнера Dependency Injection в Spring является интерфейс фабрики бинов по имени `BeanFactory`. Этот интерфейс отвечает за управление компонентами, в том числе их зависимостями и жизненными циклами. Термин *бин* в Spring используется для ссылки на любой компонент, управляемый контейнером. Обычно бины на определенном уровне придерживаются спецификации JavaBean, но это не обязательно, особенно если для связывания бинов друг с другом планируется применять Constructor Injection.

Если приложению требуется только поддержка DI, то с контейнером DI в Spring можно взаимодействовать через интерфейс `BeanFactory`. В этом случае приложение должно создать экземпляр класса, реализующего интерфейс `BeanFactory`, и сконфигурировать его в соответствие с информацией о бине и зависимостях. После того, как это сделано, приложение может получать доступ к бинам через `BeanFactory` и пользоваться их обработкой. В ряде случаев вся настройка подобного рода производится автоматически (например, в веб-приложении экземпляр `ApplicationContext` будет загружаться веб-контейнером во время начальной загрузки приложения с помощью класса `ContextLoaderListener`, предоставляемого Spring, который объявлен в файле дескрипторов `web.xml`). Но во многих случаях кодировать настройку приходится самостоятельно. Все примеры, приведенные в этой главе, требуют ручной настройки реализации `BeanFactory`.

Хотя `BeanFactory` можно сконфигурировать программно, более распространен подход с внешним конфигурированием, при котором используется какая-то разновидность файла конфигурации. Внутренне конфигурация бина представлена экземплярами классов, которые реализуют интерфейс `BeanDefinition`. Конфигурация бина хранит информацию не только о самом бине, но и о бинах, от которых он зависит. Для любых классов реализации `BeanFactory`, которые также реализуют интерфейс `BeanDefinitionReader`, данные `BeanDefinition` можно читать из файла конфигурации с применением или `PropertiesBeanDefinitionReader`, или `XmlBeanDefinitionReader`. Класс `PropertiesBeanDefinitionReader` читает определение бина из файла свойств, а `XmlBeanDefinitionReader` — из XML-файла.

Итак, бины можно идентифицировать внутри `BeanFactory`, и каждому бину может быть назначен идентификатор, имя или то и другое. Бин можно также создать без идентификатора либо имени (получив *анонимный бин*) или как внутренний бин в рамках другого бина. Каждый бин имеет, по крайней мере, одно имя, но может располагать любым их количеством (дополнительные имена разделяются запятыми). Все имена после первого считаются псевдонимами того же самого бина. Для извлечения бина из `BeanFactory`, а также для установки отношений зависимости (т.е. когда бин A зависит от бина B), используются идентификаторы или имена.

## Реализации `BeanFactory`

Описание интерфейса `BeanFactory` может выглядеть слишком сложным, но на практике это не так. Давайте рассмотрим простой пример.

Пусть имеется реализация, которая имитирует оракула, рассказывающего о смысле жизни. В листингах 3.10 и 3.11 приведено определение интерфейса и простая реализация.

**Листинг 3.10. Интерфейс `Oracle`**

```
package com.apress.prospring4.ch3;

public interface Oracle {
   String defineMeaningOfLife();
}
```

**Листинг 3.11. Простая реализация интерфейса `Oracle`**

```
package com.apress.prospring4.ch3;

public class BookwormOracle implements Oracle {
   @Override
   public String defineMeaningOfLife() {
      return "Encyclopedias are a waste of money - use the Internet";
   }
}
```

А теперь давайте посмотрим, как в автономной Java-программе можно инициализировать `BeanFactory` и получить бин `oracle` для работы с ним (листинг 3.12).

**Листинг 3.12. Использование `BeanFactory`**

```
package com.apress.prospring4.ch3;

import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;
import org.springframework.core.io.ClassPathResource;

public class XmlConfigWithBeanFactory {
   public static void main(String[] args) {
      DefaultListableBeanFactory factory = new DefaultListableBeanFactory();

      XmlBeanDefinitionReader rdr = new XmlBeanDefinitionReader(factory);
      rdr.loadBeanDefinitions(new
         ClassPathResource("META-INF/spring/xml-bean-factory-config.xml"));

      Oracle oracle = (Oracle) factory.getBean("oracle");

      System.out.println(oracle.defineMeaningOfLife());
   }
}
```

В листинге 3.12 мы используем `DefaultListableBeanFactory` — одну из двух основных реализаций `BeanFactory`, поставляемых платформой Spring — и читаем информацию `BeanDefinition` из XML-файла с применением `XmlBeanDefinitionReader`. После того, как реализация `BeanFactory` создана и сконфигурирована, мы извлекаем бин `Oracle`, используя его имя `oracle`, которое указано в XML-файле конфигурации. В листинге 3.13 приведено содержимое XML-файла для начальной загрузки `BeanFactory` (`xml-bean-factory-config.xml`).

**Листинг 3.13. Простая XML-конфигурация для Spring**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="oracle" name="wiseworm" class="com.apress.prospring4.ch3.
BookwormOracle"/>
</beans>
```

**На заметку!** При объявлении местоположения XSD-файла Spring рекомендуется не включать номер версии. Платформа Spring уже поддерживает его распознавание как XSD-файла с версией, сконфигурированного посредством указателя в файле `spring.schemas`. Этот файл находится в модуле `spring-beans`, определенном как зависимость в вашем проекте. Вам также не придется модифицировать все файлы бинов при модернизации до новой версии Spring.

В показанном выше файле объявлен бин Spring, которому назначен идентификатор `oracle` и имя `wiseworm`, а также указан лежащий в основе класс реализации `com.apress.prospring4.ch3.BookwormOracle`. Пока что не переживайте особо по поводу этой конфигурации; мы обсудим детали в последующих разделах.

Определив такую конфигурацию, запустите программу из листинга 3.12, и в выводе консоли вы увидите фразу, возвращаемую методом `defineMeaningOfLife()`.

В дополнение к `XmlBeanDefinitionReader` платформа Spring также предлагает класс `PropertiesBeanDefinitionReader`, который позволяет управлять конфигурацией бина с использованием свойств, отличающихся от XML. Хотя свойства идеальны для маленьких и простых приложений, они могут быстро стать громоздкими, когда приходится иметь дело с большим количеством бинов. По этой причине предпочтительнее применять XML-формат конфигурации для всех приложений кроме самых тривиальных.

Разумеется, можно объявлять собственные реализации `BeanFactory`, но следует помнить о том, что это требует довольно много работы; чтобы получить тот же уровень функциональности, который доступен в готовых реализациях `BeanFactory`, понадобится реализовать намного больше интерфейсов, чем просто `BeanFactory`. Если все, что требуется — это определение нового механизма конфигурации, создайте свое средство чтения определений, разработав класс, который расширяет класс `DefaultListableBeanFactory`, реализующий интерфейс `BeanFactory`.

## Интерфейс `ApplicationContext`

Интерфейс `ApplicationContext` в Spring представляет собой расширение `BeanFactory`. В дополнение к службам DI интерфейс `ApplicationContext` также предлагает другие службы, такие как служба транзакций и АОП, источник сообщений для интернационализации (i18n), обработка событий приложения и т.д.

При разработке приложений, основанных на Spring, рекомендуется взаимодействовать с платформой Spring через интерфейс `ApplicationContext`. Начальная загрузка `ApplicationContext` поддерживается в Spring за счет ручного кодирования

(создание экземпляра вручную и загрузка подходящей конфигурации) или в среде веб-контейнера через `ContextLoaderListener`. Начиная с этого места, во всех примерах кода будет использоваться `ApplicationContext`.

# Конфигурирование `ApplicationContext`

Обсудив базовые концепции IoC и DI, а также ознакомившись с простым примером применения интерфейса `BeanFactory` в Spring, давайте углубимся в детали конфигурирования Spring-приложения.

В последующих разделах мы рассмотрим различные аспекты конфигурирования Spring-приложений. В частности, мы сосредоточим внимание на интерфейсе `ApplicationContext`, который предоставляет намного больше вариантов конфигурации, чем традиционный интерфейс `BeanFactory`.

## Варианты конфигурации Spring

Прежде чем заняться деталями конфигурирования `ApplicationContext`, давайте посмотрим, какие варианты доступны для определения конфигурации приложения в рамках Spring.

Изначально платформа Spring поддерживала определение бинов либо через свойства, либо через XML-файл. После выхода JDK 5 и появления в Spring поддержки Java-аннотаций платформа Spring (начиная с версии Spring 2.5) также позволяет применять при конфигурировании `ApplicationContext` аннотации Java.

Так что же лучше — XML или аннотации? На эту тему проводилось множество дебатов, и часть из них можно найти в Интернете (для примера почитайте форумы сообщества Spring по адресу `http://forum.spring.io`). Строго определенного ответа на этот вопрос нет, и каждый из подходов характеризуется как положительными, так и отрицательными чертами. Использование XML-файла позволяет вынести всю конфигурацию за пределы Java-кода, в то время как аннотации дают разработчику возможность определять и видеть настройку DI внутри кода. Платформа Spring также допускает смешивание этих двух подходов в одном `ApplicationContext`. Один из распространенных подходов предусматривает определение инфраструктуры приложения (например, источника данных, диспетчера транзакций, фабрики подключений JMS или JMX) в XML-файле, а конфигурации DI (внедряемых бинов и зависимостей для бинов) — в аннотациях. Тем не менее, независимо от того, какой вариант выбран, его необходимо придерживаться и довести это до сведения всех членов команды разработки. Следование избранному стилю и согласованное его применение во всем приложении намного упростит последующие действия по разработке и сопровождению.

Чтобы облегчить понимание обоих подходов к конфигурации, мы по возможности будем приводить примеры кода XML и аннотаций рядом друг с другом.

### *Обзор базовой конфигурации*

Для конфигурации XML понадобится объявить обязательное базовое пространство имен, предоставляемое Spring, которое требуется приложению.

В листинге 3.14 показан простой пример, в котором объявлено только пространство имен `beans` для определения бинов Spring. В примерах мы ссылаемся на этот файл `app-context-xml.xml` как на конфигурацию в стиле XML.

**Листинг 3.14. Простая XML-конфигурация Spring**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

</beans>
```

Кроме `beans` платформа Spring предоставляет большое количество других пространств имен, предназначенных для разнообразных целей. В их число входят `context` для конфигурации `ApplicationContext`, `aop` для поддержки АОП и `tx` для поддержки транзакций. Описание этих пространств имен можно найти в соответствующих главах.

Для использования поддержки аннотаций Spring внутри своего приложения в конфигурации XML необходимо объявить дескрипторы, представленные в листинге 3.15. В примерах мы ссылаемся на этот файл `app-context-annotation.xml` как на конфигурацию XML с поддержкой аннотаций.

**Листинг 3.15. XML-конфигурация Spring с поддержкой аннотаций**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="com.apress.prospring4.ch3.annotation" />
</beans>
```

Дескриптор `<context:component-scan>` сообщает Spring о необходимости сканирования кода на предмет внедряемых бинов, аннотированных с помощью `@Component`, `@Controller`, `@Repository` и `@Service`, а также поддерживающих аннотации `@Autowired` и `@Inject` в указанном пакете (и всех его внутренних пакетах). В дескрипторе `<context:component-scan>` можно определить множество пакетов, используя в качестве разделителя запятую, точку запятой или пробел. Кроме того, для более детализированного управления этот дескриптор поддерживает включение и исключение сканирования компонентов. Например, взгляните на конфигурацию в листинге 3.16.

**Листинг 3.16. Сканирование компонентов в XML-конфигурации Spring**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">
  <context:component-scan base-package="com.apress.prospring4.ch3.annotation" >
      <context:exclude-filter type="assignable"
          expression="com.example.NotAService"/>
   </context:component-scan>
</beans>
```

Приведенный выше дескриптор сообщает Spring о необходимости сканирования указанного пакета, но с пропуском классов, которым был назначен тип, заданный в выражении (им может быть либо класс, либо интерфейс). Помимо исключающего фильтра можно применять включающий фильтр. В качестве типа критерия фильтрации (`type`) можно указывать `annotation`, `regex`, `assignable`, `aspectj` или `custom` (с собственным классом фильтра, который реализует `org.springframework.core.type.filter.TypeFilter`). Формат выражения (`expression`) зависит от заданного типа.

## Объявление компонентов Spring

После того, как вы разработали класс службы какого-то вида и желаете его использовать в приложении, основанном на Spring, необходимо сообщить платформе Spring о том, что эти бины пригодны для внедрения в другие бины, и позволить ей управлять ими. Вспомните пример из главы 2, в котором интерфейс `MessageRender` выводил сообщение и полагался на интерфейс `MessageProvider` в получении сообщения, предназначенного для визуализации. В листинге 3.17 показаны интерфейсы и реализации этих двух служб.

**Листинг 3.17. `MessageRenderer` и `MessageProvider`**

```
package com.apress.prospring4.ch3;

public interface MessageRenderer {
    void render();
    void setMessageProvider(MessageProvider provider);
    MessageProvider getMessageProvider();
}

package com.apress.prospring4.ch3;

import com.apress.prospring4.ch3.MessageProvider;
import com.apress.prospring4.ch3.MessageRenderer;

public class StandardOutMessageRenderer implements MessageRenderer {
    private MessageProvider messageProvider;

    @Override
    public void render() {
        if (messageProvider == null) {
            throw new RuntimeException(
            "You must set the property messageProvider of class:"
            + StandardOutMessageRenderer.class.getName());
            // Вы должны установить свойство messageProvider класса
        }
```

```
        System.out.println(messageProvider.getMessage());
    }
    @Override
    public void setMessageProvider(MessageProvider provider) {
        this.messageProvider = provider;
    }
    @Override
    public MessageProvider getMessageProvider() {
        return this.messageProvider;
    }
}
```

```
package com.apress.prospring4.ch3;

public interface MessageProvider {
    String getMessage();
}
```

```
package com.apress.prospring4.ch3.xml;

import com.apress.prospring4.ch3.MessageProvider;
public class HelloWorldMessageProvider implements MessageProvider {
    @Override
    public String getMessage() {
        return "Hello World!";
    }
}
```

Для объявления бинов в XML-файле к базовой конфигурации в `app-context-xml.xml` (см. листинг 3.14) добавляются дескрипторы `<bean>`, что можно видеть в листинге 3.18.

**Листинг 3.18. Объявление бинов Spring (стиль XML)**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="messageRenderer"
        class="com.apress.prospring4.ch3.xml.StandardOutMessageRenderer"/>

    <bean id="messageProvider"
        class="com.apress.prospring4.ch3.xml.HelloWorldMessageProvider"/>
</beans>
```

С помощью дескрипторов, приведенных в листинге 3.18, объявляются два бина — один с идентификатором `messageProvider` и реализацией `HelloWorldMessageProvider`, а другой с идентификатором `messageRenderer` и реализацией `StandardOutMessageRenderer`.

Для определения бинов Spring через аннотацию модифицировать XML-файл конфигурации (`app-context-annotation.xml`) больше не нужно; вам просто необходимо добавить соответствующую аннотацию к классам реализации служб в пакете `com.apress.prospring4.ch3.annotation` (листинг 3.19).

**Листинг 3.19. Объявление бинов Spring (стиль аннотаций)**

```
package com.apress.prospring4.ch3.annotation;

import org.springframework.stereotype.Service;
import com.apress.prospring4.ch3.MessageRenderer;

@Service("messageRenderer")
public class StandardOutMessageRenderer implements MessageRenderer {
    private MessageProvider messageProvider;

    @Override
    public void render() {
        if (messageProvider == null) {
            throw new RuntimeException(
            "You must set the property messageProvider of class:"
            + StandardOutMessageRenderer.class.getName());
            // Вы должны установить свойство messageProvider класса
        }

        System.out.println(messageProvider.getMessage());
    }

    @Override
    public void setMessageProvider(MessageProvider provider) {
        this.messageProvider = provider;
    }

    @Override
    public MessageProvider getMessageProvider() {
        return this.messageProvider;
    }
}

package com.apress.prospring4.ch3.annotation;

import org.springframework.stereotype.Service;
import com.apress.prospring4.ch3.MessageProvider;

@Service("messageProvider")
public class HelloWorldMessageProvider implements MessageProvider {
    @Override
    public String getMessage() {
        return "Hello World!";
    }
}
```

В предыдущем примере кода аннотация `@Service` применяется для указания на то, что этот бин предоставляет службы, которые могут требоваться другим бинам; в качестве параметра аннотации передается имя бина. Во время начальной загрузки `ApplicationContext` с XML-конфигурацией, показанной в листинге 3.15, платформа Spring будет искать эти компоненты и создавать экземпляры бинов с указанными именами.

Выбранный подход никак не влияет на способ получения бинов из `ApplicationContext`. В листинге 3.20 приведен пример кода для получения поставщика сообщений.

**Листинг 3.20. Объявление бинов Spring (тестирование)**

```
package com.apress.prospring4.ch3;

import org.springframework.context.support.GenericXmlApplicationContext;

public class DeclareSpringComponents {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:app-context-xml.xml");
      ctx.refresh();
      MessageProvider messageProvider = ctx.getBean("messageProvider",
          MessageProvider.class);
      System.out.println(messageProvider.getMessage());
   }
}
```

Вместо `DefaultListableBeanFactory` создается экземпляр `GenericXmlApplicationContext`. Класс `GenericXmlApplicationContext` реализует интерфейс `ApplicationContext` и способен выполнить начальную загрузку `ApplicationContext` из конфигураций, определенных в XML-файлах.

Если в предоставленном исходном коде для этой главы поменять местами файлы `app-context-xml.xml` и `app-context-annotation.xml`, то можно обнаружить, что оба случая дают один и тот же результат — вывод строки "Hello World!".

В листингах 3.21 (`app-context-xml.xml`) и 3.22 (`app-context-annotation.xml`) показано содержимое конфигурационных файлов для конфигурации в стиле XML и в стиле аннотаций, которые обсуждались ранее.

**Листинг 3.21. Конфигурация в стиле XML (`app-context-xml.xml`)**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="messageProvider"
        class="com.apress.prospring4.ch3.xml.HelloWorldMessageProvider"/>
</beans>
```

**Листинг 3.22. Конфигурация в стиле аннотаций (`app-context-annotation.xml`)**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">
   <context:component-scan
        base-package="com.apress.prospring4.ch3.annotation"/>
</beans>
```

## Использование внедрения через метод установки

Для конфигурирования Setter Injection с применением XML необходимо указать дескрипторы <property> внутри <bean> для каждого дескриптора <property>, в который должна быть внедрена зависимость. Например, чтобы присвоить бин поставщика сообщений свойству messageProvider бина messageRenderer, понадобится просто изменить дескриптор <bean> для бина messageRenderer, как показано в листинге 3.23.

**Листинг 3.23. Внедрение через метод установки (стиль XML)**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="messageRenderer"
      class="com.apress.prospring4.ch3.xml.StandardOutMessageRenderer">
      <property name="messageProvider" ref="messageProvider"/>
   </bean>

   <bean id="messageProvider"
      class="com.apress.prospring4.ch3.xml.HelloWorldMessageProvider"/>
</beans>
```

В этом коде мы присваиваем бин messageProvider свойству messageProvider. С помощью атрибута ref свойству можно присвоить ссылку на бин (чуть позже мы обсудим это более подробно).

Если используется Spring 2.5 или последующая версия и в XML-файле конфигурации объявлено пространство имен p, то объявить внедрение можно так, как представлено в листинге 3.24.

**Листинг 3.24. Внедрение через метод установки с применением пространства имен p (стиль XML)**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:p="http://www.springframework.org/schema/p"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="messageRenderer"
      class="com.apress.prospring4.ch3.xml.StandardOutMessageRenderer"
      p:messageProvider-ref="messageProvider"/>

   <bean id="messageProvider"
      class="com.apress.prospring4.ch3.xml.HelloWorldMessageProvider"/>
</beans>
```

Пространство имен p предоставляет упрощенный способ объявления внедрения через метод установки.

**На заметку!** Пространство имен `p` не определено в файле XSD и существует только в ядре Spring; следовательно, в атрибуте `schemaLocation` никакие файлы XSD не объявляются.

В случае аннотаций все даже проще. Нужно только добавить аннотацию `@Autowired` к методу установки, как показано в листинге 3.25.

**Листинг 3.25. Внедрение через метод установки (стиль аннотаций)**

```
package com.apress.prospring4.ch3.annotation;

import org.springframework.stereotype.Service;
import com.apress.prospring4.ch3.MessageRenderer;
import com.apress.prospring4.ch3.MessageProvider;

@Service("messageRenderer")
public class StandardOutMessageRenderer implements MessageRenderer {
    private MessageProvider messageProvider;

    @Override
    public void render() {
        if (messageProvider == null) {
            throw new RuntimeException(
            "You must set the property messageProvider of class:"
            + StandardOutMessageRenderer.class.getName());
            // Вы должны установить свойство messageProvider класса
        }

        System.out.println(messageProvider.getMessage());
    }

    @Override
    @Autowired
    public void setMessageProvider(MessageProvider provider) {
        this.messageProvider = provider;
    }

    @Override
    public MessageProvider getMessageProvider() {
        return this.messageProvider;
    }
}
```

После объявления дескриптора `<context:component-scan>` в XML-файле конфигурации во время инициализации `ApplicationContext` платформа Spring обнаружит эти аннотации `@Autowired` и внедрит зависимость по мере надобности.

**На заметку!** Для получения того же результата вместо `@Autowired` можно также использовать `@Resource(name="messageProvider")`. Аннотация `@Resource` входит в состав стандарта JSR-250, определяющего общий набор Java-аннотаций для применения в платформах JSE и JEE. В отличие от `@Autowired`, аннотация `@Resource` поддерживает параметр `name` для указания более точных требований DI. Кроме того, платформа Spring поддерживает использование аннотации `@Inject`, которая является частью спецификации JSR-299 ("Contexts and Dependency Injection for the Java EE Platform" — "Контексты и внедрение зависимостей для платформы Java EE"). Аннотация `@Inject` эквивалентна по поведению аннотации `@Autowired` в Spring.

А теперь давайте проверим результат с помощью кода, приведенного в листинге 3.26.

**Листинг 3.26. Использование внедрения через метод установки (тестирование)**

```
package com.apress.prospring4.ch3;

import org.springframework.context.support.GenericXmlApplicationContext;

public class DeclareSpringComponents {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        MessageRenderer messageRenderer = ctx.getBean("messageRenderer",
            MessageRenderer.class);

        messageRenderer.render();
    }
}
```

Как и ранее, файлы `app-context-xml.xml` и `app-context-annotation.xml` в исходном коде для главы можно поменять местами; в обоих случаях получается один и тот же результат — вывод строки "Hello World!".

## Использование внедрения через конструктор

В предыдущем примере реализация `MessageProvider`, `HelloWorldMessageProvider`, возвращает одно и то же жестко закодированное сообщение для каждого вызова метода `getMessage()`. В файле конфигурации Spring можно просто создать конфигурируемый класс `MessageProvider`, который позволит определять сообщение внешне (листинг 3.27).

**Листинг 3.27. Класс `ConfigurableMessageProvider` (стиль XML)**

```
package com.apress.prospring4.ch3.xml;

import com.apress.prospring4.ch3.MessageProvider;

public class ConfigurableMessageProvider implements MessageProvider {
    private String message;

    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }

    @Override
    public String getMessage() {
        return message;
    }
}
```

Как видите, создать экземпляр класса `ConfigurableMessageProvider`, не предоставив значение для сообщения, невозможно (если только не указать `null`). Это в точности то, что требуется, и данный класс идеально подходит для использования с Constructor Injection. В листинге 3.28 показано, как переделать определение бина

messageProvider для создания экземпляра ConfigurableMessageProvider, внедряя сообщение с помощью Constructor Injection.

**Листинг 3.28. Использование внедрения через конструктор (стиль XML)**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="messageProvider"
       class="com.apress.prospring4.ch3.xml.ConfigurableMessageProvider">
       <constructor-arg value="Configurable message"/>
    </bean>
</beans>
```

В этом коде вместо дескриптора <property> применяется дескриптор <constructor-arg>. Поскольку на этот раз другой бин не передается, а только литерал String, для указания значения аргумента конструктора используется атрибут value вместо ref.

При наличии более одного аргумента конструктора или нескольких конструкторов в классе каждому дескриптору <constructor-arg> понадобится предоставить атрибут index для указания индекса аргумента, начинающегося с 0, в сигнатуре конструктора. Имея дело с конструкторами, принимающими несколько аргументов, всегда лучше применять атрибут index, чтобы избежать путаницы между параметрами и обеспечить выбор платформой Spring корректного конструктора.

В дополнение к пространству имен p в версии Spring 3.1 можно также использовать пространство имен c, как показано ниже:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:c="http://www.springframework.org/schema/c"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="messageProvider"
       class="com.apress.prospring4.ch3.xml.ConfigurableMessageProvider"
         c:message="This is a configurable message"/>
</beans>
```

**На заметку!** Пространство имен c не определено в файле XSD и существует только в ядре Spring; следовательно, в атрибуте schemaLocation никакие файлы XSD не объявляются.

Для применения аннотации вместе с Constructor Injection мы также используем аннотацию @Autowired в методе конструктора целевого бина (листинг 3.29), что является альтернативой варианту Setter Injection, который демонстрировался в листинге 3.24.

**Листинг 3.29. Использование внедрения через конструктор (стиль аннотаций)**

```
package com.apress.prospring4.ch3.annotation;

import org.springframework.stereotype.Service;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.beans.factory.annotation.Autowired;
import com.apress.prospring4.ch3.MessageProvider;

@Service("messageProvider")
public class ConfigurableMessageProvider implements MessageProvider {
   private String message;

   @Autowired
   public ConfigurableMessageProvider(@Value("Configurable message")
     String message) {
       this.message = message;
   }

   @Override
   public String getMessage() {
       return this.message;
   }
}
```

В листинге 3.29 применяется еще одна аннотация, `@Value`, для определения значения, подлежащего внедрению в конструктор. Таким способом в Spring внедряются значения в бин. Кроме простых строк можно использовать мощный язык SpEL для динамического внедрения значений (об этом пойдет речь далее в этой главе).

Однако жесткое кодирование значения не является удачной идеей, т.к. его изменение влечет за собой перекомпиляцию программы. Даже если выбран вариант DI с аннотациями, рекомендуется выносить за пределы кода значения, предназначенные для внедрения. Чтобы вынести сообщение за пределы кода, определим его как бин Spring в конфигурационном файле для случая аннотаций (листинг 3.30).

**Листинг 3.30. Определение сообщения как бина Spring при внедрении через конструктор (стиль аннотаций)**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:c="http://www.springframework.org/schema/c"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

   <context:component-scan
        base-package="com.apress.prospring4.ch3.annotation"/>

   <bean id="message" class="java.lang.String"
         c:_0="This is a configurable message"/>
</beans>
```

В листинге 3.30 определен бин с идентификатором `message` и типом `java.lang.String`. Обратите внимание, что мы также применяем пространство имен `c` для Constructor Injection при установке строкового значения, и конструкция `_0` указывает индекс аргумента в конструкторе.

Имея объявленный бин, можно избавиться от аннотации `@Value` в целевом бине, как показано в листинге 3.31.

**Листинг 3.31. Избавление от аннотации `@Value` при внедрении через конструктор (стиль аннотаций)**

```
package com.apress.prospring4.ch3.annotation;

package com.apress.prospring4.ch3.annotation;

import org.springframework.stereotype.Service;
import org.springframework.beans.factory.annotation.Autowired;
import com.apress.prospring4.ch3.MessageProvider;

@Service("messageProvider")
public class ConfigurableMessageProvider implements MessageProvider {
    private String message;
    @Autowired
    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }
    @Override
    public String getMessage() {
        return this.message;
    }
}
```

Поскольку объявленный бин `message` и его идентификатор совпадают с именем аргумента в конструкторе, Spring обнаружит аннотацию и внедрит значение в метод конструктора. Запустив тест посредством кода, приведенного в листинге 3.32, с конфигурациями посредством XML (`app-context-xml.xml`) и аннотаций (`app-context-annotation.xml`), мы получаем в обоих случаях вывод сконфигурированного сообщения:

```
This is a configurable message
```

**Листинг 3.32. Использование внедрения через конструктор (тестирование)**

```
package com.apress.prospring4.ch3;

import org.springframework.context.support.GenericXmlApplicationContext;

public class DeclareSpringComponents {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();
        MessageProvider messageProvider = ctx.getBean("messageProvider",
            MessageProvider.class);
        System.out.println(messageProvider.getMessage());
    }
}
```

В некоторых случаях Spring не имеет возможности выяснить, какой конструктор вы хотите использовать для Constructor Injection. Обычно это происходит, когда есть два конструктора с одинаковым количеством аргументов, типы которых представлены в точности тем же самым способом. Взгляните на код в листинге 3.33.

**Листинг 3.33. Путаница с конструкторами**

```
package com.apress.prospring4.ch3.xml;

import org.springframework.context.support.GenericXmlApplicationContext;

public class ConstructorConfusion {
   private String someValue;

   public ConstructorConfusion(String someValue) {
      System.out.println("ConstructorConfusion(String) called");
      this.someValue = someValue;
   }
   public ConstructorConfusion(int someValue) {
      System.out.println("ConstructorConfusion(int) called");
      this.someValue = "Number: " + Integer.toString(someValue);
   }
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx =
        new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-xml.xml");
      ctx.refresh();
      ConstructorConfusion cc =
        (ConstructorConfusion) ctx.getBean("constructorConfusion");
      System.out.println(cc);
   }
   public String toString() {
      return someValue; Constructor Injection:constructor confusion
   }
}
```

Здесь четко видно, что этот код делает — он просто извлекает бин типа `ConstructorConfusion` из `ApplicationContext` и записывает значение в консольный вывод. Теперь рассмотрим код в листинге 3.34 (`app-context-xml.xml`).

**Листинг 3.34. Путаница с конструкторами (файл `app-context-xml.xml`)**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:c="http://www.springframework.org/schema/c"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="messageProvider"
      class="com.apress.prospring4.ch3.xml.ConfigurableMessageProvider"
      c:message="This is a configurable message"/>

   <bean id="constructorConfusion"
      class="com.apress.prospring4.ch3.xml.ConstructorConfusion">
```

```
        <constructor-arg>
            <value>90</value>
        </constructor-arg>
    </bean>
</beans>
```

Какой конструктор будет вызван в этом случае? Запуск примера дает следующий вывод:

```
ConstructorConfusion(String) called
90
```

Вывод показывает, что был вызван конструктор с аргументом `String`. Учитывая то, что любое целочисленное значение, переданное с применением Constructor Injection, предваряется префиксом `Number:`, как видно в коде конструктора с аргументом `int`, это не тот результат, который ожидался. Чтобы обойти данную проблему, понадобится внести небольшое изменение в конфигурацию, как показано в листинге 3.35 (`app-context-xml.xml`).

**Листинг 3.35. Устранение путаницы с конструкторами**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="messageProvider"
        class="com.apress.prospring4.ch3.xml.ConfigurableMessageProvider"
        c:message="This is a configurable message"/>
    <bean id="constructorConfusion"
        class="com.apress.prospring4.ch3.xml.ConstructorConfusion">
        <constructor-arg type="int">
            <value>90</value>
        </constructor-arg>
    </bean>
</beans>
```

Обратите внимание, что дескриптор `<constructor-arg>` имеет дополнительный атрибут `type`, указывающий тип аргумента, который должна искать платформа Spring. Запуск примера теперь дает корректный вывод:

```
ConstructorConfusion(int) called
Number: 90
```

При внедрении через конструктор с использованием аннотаций такой путаницы можно избежать, применяя аннотацию непосредственно к целевому методу конструктора, что и сделано в листинге 3.36.

**Листинг 3.36. Путаница с конструкторами (стиль аннотаций)**

```
package com.apress.prospring4.ch3.annotation;

import org.springframework.stereotype.Service;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.support.GenericXmlApplicationContext;

@Service("constructorConfusion")
public class ConstructorConfusion {
   private String someValue;
   public ConstructorConfusion(String someValue) {
      System.out.println("ConstructorConfusion(String) called");
      this.someValue = someValue;
   }
   @Autowired
   public ConstructorConfusion(@Value("90") int someValue) {
      System.out.println("ConstructorConfusion(int) called");
      this.someValue = "Number: " + Integer.toString(someValue);
   }
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx =
         new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();

      ConstructorConfusion cc =
         (ConstructorConfusion) ctx.getBean("constructorConfusion");
      System.out.println(cc);
   }
   public String toString() {
      return someValue;
   }
}
```

За счет применения аннотации `@Autowired` к целевому методу конструктора платформа Spring будет использовать этот метод для создания экземпляра бина и внедрения указанного значения. Как и ранее, это значение должно быть вынесено во внешнюю конфигурацию.

**На заметку!** Аннотация `@Autowired` может быть применена только к одному из методов конструкторов. В случае ее применения к нескольким методам конструкторов платформа Spring выдаст соответствующее сообщение во время начальной загрузки `ApplicationContext`.

## Использование параметров внедрения

В предыдущих двух примерах было показано, как внедрять другие компоненты и значения в бин с помощью Setter Injection и Constructor Injection. Платформа Spring поддерживает огромное количество вариантов для параметров внедрения, позволяя внедрять не только другие компоненты и простые значения, но также коллекции Java, внешне определенные свойства и даже бины из другой фабрики. Все эти типы параметров внедрения могут применяться в Setter Injection и Constructor Injection за счет использования соответствующего дескриптора внутри дескрипторов `<property>` и `<constructor-args>`.

### *Внедрение простых значений*

Внедрять простые значения в бины легко. Для этого нужно лишь указать в конфигурации значение, заключенное в дескриптор `<value>`. По умолчанию дескриптор `<value>` может не только читать значения `String`, но также преобразовывать их в любой элементарный класс или оболочку для такого класса. В листинге 3.37 приведен простой бин, который имеет разнообразные свойства, доступные для внедрения.

**Листинг 3.37. Внедрение простых значений (стиль XML)**

```
package com.apress.prospring4.ch3.xml;

import org.springframework.context.support.GenericXmlApplicationContext;

public class InjectSimple {
   private String name;
   private int age;
   private float height;
   private boolean programmer;
   private Long ageInSeconds;

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-xml.xml");
      ctx.refresh();

      InjectSimple simple = (InjectSimple)ctx.getBean("injectSimple");

      System.out.println(simple);
   }

   public void setAgeInSeconds(Long ageInSeconds) {
      this.ageInSeconds = ageInSeconds;
   }

   public void setProgrammer(boolean programmer) {
      this.programmer = programmer;
   }

   public void setAge(int age) {
      this.age = age;
   }
   public void setHeight(float height) {
      this.height = height;
   }

   public void setName(String name) {
      this.name = name;
   }

   public String toString() {
      return "Name :" + name + "\n"
         + "Age:" + age + "\n"
         + "Age in Seconds: " + ageInSeconds + "\n"
         + "Height: " + height + "\n"
         + "Is Programmer?: " + programmer;
   }
}
```

В дополнение к свойствам класс `InjectSimple` также определяет метод `main()`, который создает `ApplicationContext` и затем извлекает бин `InjectSimple` из Spring. После этого значения свойств этого бина записываются в консольный вывод. В листинге 3.38 представлена конфигурация (`app-context-xml.xml`) для этого бина.

**Листинг 3.38. Конфигурирование внедрения простых значений**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="injectSimple" class="com.apress.prospring4.ch3.xml.
InjectSimple"
      p:name="Chris Schaefer" p:age="32" p:height="1.778"
p:programmer="true"
      p:ageInSeconds="1009843200"/>
</beans>
```

В листингах 3.37 и 3.38 видно, что допускается определять свойства бина, которые принимают значения `String`, элементарные значения или оболочки для элементарных значений, и затем внедрять значения для этих свойств с использованием дескриптора `<value>`. Ниже показан вывод, полученный в результате запуска этого примера:

```
Name: Chris Schaefer
Age: 32
Age in Seconds: 1009843200
Height: 1.778
Is Programmer?: true
```

Для внедрения простых значений в стиле аннотаций можно применять аннотацию `@Value` к свойствам бина. В этот раз вместо метода установки мы применим указанную аннотацию к оператору объявления свойства, как показано в листинге 3.39. (Платформа Spring поддерживает аннотации либо в методе установки, либо в свойствах.)

**Листинг 3.39. Внедрение простых значений (стиль аннотаций)**

```
package com.apress.prospring4.ch3.annotation;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.stereotype.Service;

@Service("injectSimple")
public class InjectSimple {
   @Value("Chris Schaefer")
   private String name;

   @Value("32")
   private int age;
```

```
    @Value("1.778")
    private float height;

    @Value("true")
    private boolean programmer;

    @Value("1009843200")
    private Long ageInSeconds;

    public static void main(String[] args) {
        GenericXmlApplicationContext ctx =
          new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();

        InjectSimple simple = (InjectSimple) ctx.getBean("injectSimple");
        System.out.println(simple);
    }

    public String toString() {
        return "Name: " + name + "\n"
            + "Age: " + age + "\n"
            + "Age in Seconds: " + ageInSeconds + "\n"
            + "Height: " + height + "\n"
            + "Is Programmer?: " + programmer;
    }
}
```

Полученный результат будет в точности таким же, как при использовании конфигурации в стиле XML.

### *Внедрение значений с использованием SpEL*

В Spring 3 появилось мощное функциональное средство — язык выражений Spring (Spring Expression Language — SpEL). Язык SpEL позволяет вычислять выражения динамически и затем применять их в `ApplicationContext`. Результат вычисления можно использовать для внедрения в бины Spring. В этом разделе мы посмотрим, как применять SpEL для внедрения свойств из других бинов, на основе примера из предыдущего раздела.

Предположим, что теперь мы хотим вынести значения, предназначенные для внедрения в бин Spring, в конфигурационный класс (листинг 3.40).

**Листинг 3.40. Внедрение значений с использованием SpEL (стиль XML)**

```
package com.apress.prospring4.ch3.xml;

public class InjectSimpleConfig {
    private String name = "Chris Schaefer";
    private int age = 32;
    private float height = 1.778f;
    private boolean programmer = true;
    private Long ageInSeconds = 1009843200L;

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public float getHeight() {
        return height;
    }
    public void setHeight(float height) {
        this.height = height;
    }
    public boolean isProgrammer() {
        return programmer;
    }
    public void setIsProgrammer(boolean programmer) {
        this.programmer = programmer;
    }
    public Long getAgeInSeconds() {
        return ageInSeconds;
    }
    public void setAgeInSeconds(Long ageInSeconds) {
        this.ageInSeconds = ageInSeconds;
    }
}
```

Затем можно определить бин в XML-конфигурации и с помощью SpEL внедрить свойства бина в зависимый бин, как сделано в листинге 3.41 (`app-context-xml.xml`).

**Листинг 3.41. Внедрение значений с использованием SpEL (стиль XML)**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="injectSimpleConfig"
        class="com.apress.prospring4.ch3.xml.InjectSimpleConfig"/>
    <bean id="injectSimpleSpel"
        class="com.apress.prospring4.ch3.xml.InjectSimpleSpel"
        p:name="#{injectSimpleConfig.name}"
        p:age="#{injectSimpleConfig.age}"
        p:height="#{injectSimpleConfig.height}"
        p:programmer="#{injectSimpleConfig.programmer}"
        p:ageInSeconds="#{injectSimpleConfig.ageInSeconds}"/>
</beans>
```

Обратите внимание на использование SpEL-выражения `#{injectSimpleConfig.name}` в ссылке на свойство другого бина. Для возраста (`age`) мы добавляем 1 к значению из бина, чтобы продемонстрировать возможность применения SpEL для манипулирования свойством и внедрения в зависимый бин. Давайте протестируем конфигурацию с помощью кода, приведенного в листинге 3.42.

**Листинг 3.42. Тестирование внедрения значений с использованием SpEL**

```
package com.apress.prospring4.ch3.xml;

import org.springframework.context.support.GenericXmlApplicationContext;

public class InjectSimpleSpel {
    private String name;
    private int age;
    private float height;
    private boolean programmer;
    private Long ageInSeconds;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return this.age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public float getHeight() {
        return this.height;
    }

    public void setHeight(float height) {
        this.height = height;
    })

    public boolean isProgrammer() {
        return this.programmer;
    }

    public void setProgrammer(boolean programmer) {
        this.programmer = programmer;
    }

    public Long getAgeInSeconds() {
        return this.ageInSeconds;
    }

    public void setAgeInSeconds(Long ageInSeconds) {
        this.ageInSeconds = ageInSeconds;
    }

    public String toString() {
        return "Name: " + name + "\n"
```

```
            + "Age: " + age + "\n"
            + "Age in Seconds: " + ageInSeconds + "\n"
            + "Height: " + height + "\n"
            + "Is Programmer?: " + programmer;
    }
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx =
            new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();
        InjectSimpleSpel simple =
            (InjectSimpleSpel)ctx.getBean("injectSimpleSpel");
        System.out.println(simple);
    }
}
```

Вывод этой программы выглядит так:

```
Name: Chris Schaefer
Age:33
Age in Seconds: 1009843200
Height: 1.778
Is Programmer?: true
```

При использовании внедрения значений в стиле аннотаций понадобится только добавить аннотации `@Value` с выражениями SpEL (листинг 3.43).

**Листинг 3.43. Внедрение значений с использованием SpEL (стиль аннотаций)**

```
package com.apress.prospring4.ch3.annotation;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.stereotype.Service;

@Service("injectSimpleSpel")
public class InjectSimpleSpel {
    @Value("#{injectSimpleConfig.name}")
    private String name;

    @Value("#{injectSimpleConfig.age + 1}")
    private int age;

    @Value("#{injectSimpleConfig.height}")
    private float height;

    @Value("#{injectSimpleConfig.programmer}")
    private boolean programmer;

    @Value("#{injectSimpleConfig.ageInSeconds}")
    private Long ageInSeconds;

    public String toString() {
        return "Name: " + name + "\n"
            + "Age: " + age + "\n"
            + "Age in Seconds: " + ageInSeconds + "\n"
            + "Height: " + height + "\n"
            + "Is Programmer?: " + programmer;
    }
```

```
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx =
          new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        InjectSimpleSpel simple =
          (InjectSimpleSpel)ctx.getBean("injectSimpleSpel");
        System.out.println(simple);
    }
}
```

В листинге 3.44 показана версия класса `InjectSimpleConfig`, в которой применяются аннотации.

**Листинг 3.44. Класс `InjectSimpleConfig` (версия с аннотациями)**

```
package com.apress.prospring4.ch3.annotation;

import org.springframework.stereotype.Component;

@Component("injectSimpleConfig")
public class InjectSimpleConfig {
    private String name = "John Smith";
    private int age = 35;
    private float height = 1.78f;
    private boolean programmer = true;
    private Long ageInSeconds = 1103760000L;
}
```

В листинге 3.44 вместо аннотации `@Service` используется аннотация `@Component`. В основном применение `@Component` дает тот же самый эффект, что и `@Service`. Обе аннотации инструктируют Spring о том, что аннотированный класс является кандидатом на автоматическое обнаружение с использованием конфигурации, основанной на аннотациях, и сканирования в пути классов. Однако поскольку класс `InjectSimpleConfig` хранит конфигурацию приложения, а не предоставляет бизнес-службу, применение `@Component` имеет больший смысл.

На практике `@Service` является специализацией `@Component`, отражающей тот факт, что аннотированный класс предоставляет бизнес-службу другим уровням внутри приложения.

Тестирование полученной программы даст тот же самый результат. С использованием SpEL можно получать доступ к любым управляемым Spring бинам и свойствам, а также манипулировать ими в приложении за счет поддержки в Spring развитых языковых средств и синтаксиса.

### *Внедрение бинов в одной и той же единице XML*

Как вы уже видели, один бин можно внедрять в другой с применением дескриптора `<ref>`. В листинге 3.45 приведен класс, который предлагает метод установки, чтобы разрешить внедрение бина.

**Листинг 3.45. Внедрение бинов**

```
package com.apress.prospring4.ch3.xml;

import org.springframework.context.support.GenericXmlApplicationContext;
import com.apress.prospring4.ch3.Oracle;
import com.apress.prospring4.ch3.BookwormOracle;

public class InjectRef {
   private Oracle oracle;

   public void setOracle(Oracle oracle) {
      this.oracle = oracle;
   }

   public static void main(String[] args) {
      GenericXmlApplicationContext ctx =
        new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-xml.xml");
      ctx.refresh();

      InjectRef injectRef = (InjectRef) ctx.getBean("injectRef");
      System.out.println(injectRef);
   }

   public String toString() {
      return oracle.defineMeaningOfLife();
   }
}
```

Чтобы настроить Spring для внедрения одного бина в другой, сначала понадобится сконфигурировать два бина: один в качестве внедряемого и еще один в качестве цели внедрения. После того, как это сделано, останется только сконфигурировать внедрение с использованием дескриптора `<ref>` для целевого бина. В листинге 3.46 показан пример такой конфигурации (`app-context-xml.xml`).

**Листинг 3.46. Конфигурирование внедрения бина**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="oracle"
       name="wiseworm" class="com.apress.prospring4.ch3.BookwormOracle"/>

   <bean id="injectRef" class="com.apress.prospring4.ch3.xml.InjectRef">
      <property name="oracle">
         <ref local="oracle"/>
      </property>
   </bean>
</beans>
```

Запуск класса из листинга 3.46 даст следующий вывод:

```
Encyclopedias are a waste of money - use the Internet
```

Здесь важно отметить один момент: внедряемый тип не должен в точности совпадать с тем типом, который определен в качестве целевого; эти типы просто должны быть совместимыми. *Совместимость* означает, что если объявленный в качестве целевого тип является интерфейсом, то внедряемый тип должен реализовывать этот интерфейс. Если объявленный тип является классом, то внедряемый тип должен относиться либо к тому же самому типу, либо к его подтипу. В этом примере класс `InjectRef` определяет метод `setOracle()`, предназначенный для получения экземпляра `Oracle`, который представляет собой интерфейс, а внедряемым типом является `BookwormOracle` — класс, реализующий `Oracle`. Этот момент может привести в замешательство некоторых разработчиков, хотя в действительности здесь нет ничего сложного. Внедрение подчиняется тем же правилами типизации, что и любой код Java, поэтому если вы знаете, как работает типизация в Java, то должны понимать, что типизация при внедрении довольно проста.

В предыдущем примере идентификатор (`id`) бина, предназначенного для внедрения, был указан с использованием атрибута `local` дескриптора `<ref>`. Как будет показано позже в разделе "Именование бинов", бину можно назначать более одного имени и ссылаться на него с помощью множества псевдонимов. Применение атрибута `local` означает, что дескриптор `<ref>` всегда просматривает только идентификатор бина и никогда не принимает во внимание его псевдонимы. Более того, определение бина должно существовать в том же самом XML-файле конфигурации. Чтобы внедрить бин по любому имени или импортировать бин из другого XML-файла конфигурации, вместо атрибута `local` в дескрипторе `<ref>` необходимо применять атрибут `bean`. В листинге 3.47 показана альтернативная конфигурация для предыдущего примера, в которой для внедряемого бина используется альтернативное имя.

**Листинг 3.47. Внедрение с использованием псевдонимов бина**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="oracle" name="wiseworm" class="com.apress.prospring4.ch3.
BookwormOracle"/>

   <bean id="injectRef" class="com.apress.prospring4.ch3.xml.InjectRef">
      <property name="oracle">
         <ref bean="wiseworm"/>
      </property>
   </bean>
</beans>
```

В этом примере бину `oracle` с помощью атрибута `name` назначается псевдоним, после чего этот бин внедряется в бин `injectRef` с указанием псевдонима в атрибуте `bean` дескриптора `<ref>`. Пока что не следует особо переживать по поводу семантики именования — мы подробно обсудим ее далее в этой главе. Запуск класса `InjectRef` (листинг 3.45) даст тот же результат, что и предыдущий пример.

### *Внедрение и вложение `ApplicationContext` друг в друга*

До сих пор внедряемые бины располагались в том же самом контексте `ApplicationContext` (и, следовательно, в той же самой фабрике `BeanFactory`), что и бины, в которые производилось внедрение. Тем не менее, Spring поддерживает иерархическую структуру для `ApplicationContext`, когда один контекст (и связанная с ним фабрика `BeanFactory`) является родительским для другого контекста. Разрешая вложение `ApplicationContext`, платформа Spring делает возможным разнесение конфигурации по отдельным файлам — настоящая находка для крупных проектов, содержащих множество бинов.

При вложении `ApplicationContext` платформа Spring позволяет бинам из контекста, который считается дочерним, ссылаться на бины в родительском контексте. Вложение `ApplicationContext` с применением `GenericXmlApplicationContext` осуществляется очень просто. Чтобы вложить один контекст `GenericXmlApplicationContext` в другой, необходимо просто вызвать метод `setParent()` в дочернем контексте `ApplicationContext`, как показано в листинге 3.48.

**Листинг 3.48. Вложение `GenericXmlApplicationContext`**

```
package com.apress.prospring4.ch3;

import org.springframework.context.support.GenericXmlApplicationContext;

public class HierarchicalAppContextUsage {
   public static void main(String[] args) {
      GenericXmlApplicationContext parent =
        new GenericXmlApplicationContext();
      parent.load("classpath:META-INF/spring/parent.xml");
      parent.refresh();

      GenericXmlApplicationContext child =
         new GenericXmlApplicationContext();
      child.load("classpath:META-INF/spring/app-context-xml.xml");
      child.setParent(parent);
      child.refresh();

      SimpleTarget target1 = (SimpleTarget) child.getBean("target1");
      SimpleTarget target2 = (SimpleTarget) child.getBean("target2");
      SimpleTarget target3 = (SimpleTarget) child.getBean("target3");

      System.out.println(target1.getVal());
      System.out.println(target2.getVal());
      System.out.println(target3.getVal());
   }
}
```

В листинге 3.49 представлен класс `SimpleTarget`.

**Листинг 3.49. Класс `SimpleTarget`**

```
package com.apress.prospring4.ch3;

public class SimpleTarget {
   private String val;
```

```
    public void setVal(String val) {
        this.val = val;
    }

    public String getVal() {
        return val;
    }
}
```

Внутри конфигурационного файла для дочернего контекста ApplicationContext ссылка на какой-нибудь бин из родительского контекста ApplicationContext работает в точности как ссылка на бин из дочернего контекста ApplicationContext, если только в дочернем контексте ApplicationContext не присутствует бин с тем же самым именем. В этом случае необходимо просто заменить атрибут bean дескриптора <ref> атрибутом parent. В листинге 3.50 приведен пример конфигурационного файла для родительского контекста BeanFactory (parent.xml).

**Листинг 3.50. Конфигурация родительского контекста ApplicationContext**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="injectBean" class="java.lang.String" c:_0="Bean In Parent"/>

    <bean id="injectBeanParent" class="java.lang.String" c:_0="Bean In Parent"/>
</beans>
```

Как видите, в конфигурации просто определены два бина: injectBean и injectBeanParent. Оба они являются объектами String со значениями Bean In Parent (Бин в родительском контексте). В листинге 3.51 представлен пример конфигурации для дочернего контекста ApplicationContext (app-context-xml.xml).

**Листинг 3.51. Конфигурация дочернего контекста ApplicationContext**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="target1" class="com.apress.prospring4.ch3.SimpleTarget"
        p:val-ref="injectBeanParent"/>

    <bean id="target2" class="com.apress.prospring4.ch3.SimpleTarget"
        p:val-ref="injectBean"/>

    <bean id="target3" class="com.apress.prospring4.ch3.SimpleTarget">
        <property name="val">
```

```
        <ref parent="injectBean"/>
      </property>
   </bean>
   <bean id="injectBean" class="java.lang.String" c:_0="Child In Bean"/>
</beans>
```

Обратите внимание, что здесь определены четыре бина. Бин `injectBean` в этом листинге похож на бин в `injectBean` в родительском контексте за исключением отличия в значении `String`, которое отражает его размещение в дочернем контексте `ApplicationContext` (строка `Bean In Child`).

Бин `target1` использует атрибут `val-ref` дескриптора `<bean>` для ссылки на бин по имени `injectBeanParent`. Поскольку этот бин существует только в родительской фабрике `BeanFactory`, бин `target1` получает ссылку на него. Здесь интересно отметить два момента. Первый: атрибут `bean` можно применять для ссылки на бины как в дочернем, так и в родительском контекстах `ApplicationContext`. Это позволяет ссылаться на бины прозрачным образом, позволяя перемещать бины между конфигурационными файлами по мере роста приложения. Второй момент связан с невозможностью применения атрибута `local` для ссылки на бины в родительском контексте `ApplicationContext`. Анализатор XML проверяет, существует ли в том же самом файле допустимый элемент, который соответствует значению, указанному в атрибуте `local`, и это предотвращает его использование для ссылки на бины в родительском контексте.

Бин `target2` применяет атрибут `val-ref` дескриптора `<bean>` для ссылки на `injectBean`. Поскольку этот бин определен в обоих контекстах `ApplicationContext`, бин `target2` получает ссылку на `injectBean` в собственном контексте `ApplicationContext`.

Бин `target3` использует дескриптор `<ref>` для непосредственной ссылки на бин `injectBean` из родительского контекста `ApplicationContext`. Так как в `target3` применяется атрибут `parent` дескриптора `<ref>`, бин `injectBean`, объявленный в дочернем контексте `ApplicationContext`, полностью игнорируется.

**На заметку!** Вы могли заметить, что в отличие от `target1` и `target2`, бин `target3` не использует пространство имен `p`. Несмотря на то что пространство имен `p` предлагает удобные сокращения, оно не предоставляет все возможности, доступные в случае применения дескриптора `property`, такие как ссылка на родительский бин. Хотя мы привели его в качестве примера, для определения бинов лучше выбрать что-то одно — либо пространство имен `p`, либо дескрипторы `property`, а не смешивать оба стиля (если только в этом нет крайней необходимости).

Ниже показан вывод, полученный в результате запуска класса `HierarchicalAppContextUsage` (листинг 3.48):

```
Bean In Parent
Bean In Child
Bean In Parent
```

Как и ожидалось, бины `target1` и `target3` получают ссылку на бины из родительского контекста `ApplicationContext`, тогда как бин `target2` — ссылку на бин из дочернего контекста `ApplicationContext`.

### Использование коллекций для внедрения

Часто бинам необходим доступ к коллекциям объектов, а не только к отдельным бинам или значениям. Таким образом, не должно стать сюрпризом, что Spring позволяет внедрять коллекцию объектов в бины. Применять коллекцию несложно: необходимо выбрать дескриптор <list>, <map>, <set> или <props> для представления экземпляра List, Map, Set или Properties и затем передать индивидуальные элементы, как это делается при любом другом внедрении. Дескриптор <props> позволяет передавать в качестве значений только String, потому что класс Properties разрешает только значения свойств типа String. Для <list>, <map> или <set> можно использовать любой дескриптор при внедрении в свойство, даже еще один дескриптор коллекции. Это позволяет передавать List из Map, Map из Set или даже List, состоящий из элементов Map, каждый из которых состоит из элементов Set, а те, в свою очередь, из элементов List! В листинге 3.52 показан класс, в который можно внедрять все четыре типа коллекций.

**Листинг 3.52. Внедрение коллекции (стиль XML)**

```
package com.apress.prospring4.ch3.xml;

import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

import org.springframework.context.support.GenericXmlApplicationContext;

public class CollectionInjection {
    private Map<String, Object> map;
    private Properties props;
    private Set set;
    private List list;
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx =
            new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();
        CollectionInjection instance =
            (CollectionInjection) ctx.getBean("injectCollection");
        instance.displayInfo();
    }
    public void setList(List list) {
        this.list = list;
    }
    public void setSet(Set set) {
        this.set = set;
    }
    public void setMap(Map <String, Object> map) {
        this.map = map;
    }
    public void setProps(Properties props) {
        this.props = props;
    }
```

```
    public void displayInfo() {
        System.out.println("Map contents:\n");
        for (Map.Entry<String, Object> entry: map.entrySet()) {
            System.out.println("Key: " + entry.getKey() + " - Value: "
                               + entry.getValue());
        }
        System.out.println("\nProperties contents:\n");)
        for (Map.Entry<Object, Object> entry: props.entrySet()) {
            System.out.println("Key: " + entry.getKey() + " - Value: "
                               + entry.getValue());
        }
        System.out.println("\nSet contents:\n");
        for (Object obj: set) {
            System.out.println("Value: " + obj);
        }
        System.out.println("\nList contents:\n");
        for (Object obj: list) {
            System.out.println("Value: " + obj);
        }
    }
}
```

Здесь довольно много кода, но в действительности он мало что делает. Метод `main()` извлекает бин `CollectionInjection` из Spring и затем вызывает метод `displayInfo()`. Этот метод просто выводит содержимое экземпляров `Map`, `Properties`, `Set` и `List`, которые будут внедрены из Spring. В листинге 3.53 приведена конфигурация, требуемая для внедрения значений каждого свойства класса `CollectionInjection`.

**Листинг 3.53. Конфигурирование внедрения коллекций (стиль XML)**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="oracle" name="wiseworm"
          class="com.apress.prospring4.ch3.xml.BookwormOracle"/>
    <bean id="injectCollection"
          class="com.apress.prospring4.ch3.xml.CollectionInjection">
        <property name="map">
            <map>
                <entry key="someValue">
                    <value>Hello World!</value>
                </entry>
                <entry key="someBean">
                    <ref local="oracle"/>
                </entry>
            </map>
        </property>
```

```
      <property name="props">
         <props>
            <prop key="firstName">Chris</prop>
            <prop key="secondName">Schaefer</prop>
         </props>
      </property>
      <property name="set">
         <set>
            <value>Hello World!</value>
            <ref local="oracle"/>
         </set>
      </property>
      <property name="list">
         <list>)
            <value>Hello World!</value>
            <ref local="oracle"/>
         </list>
      </property>
   </bean>
</beans>
```

Обратите внимание на объявление свойства `Map<String,Object>`. Для JDK 5 и более новых версий платформа Spring также поддерживает строго типизированное определение `Collection`, и будет выполнять преобразование из XML-конфигурации в указанный соответствующим образом тип (`app-context-xml.xml`).

В этом коде значения внедряются во все четыре метода установки, доступные в классе `CollectionInjection`. Для свойства `map` мы внедряем экземпляр `Map` с применением дескриптора `<map>`. Каждый элемент карты указывается с помощью дескриптора `<entry>`, и каждый элемент имеет ключ `String` и значение. Этим значением может быть любое значение, которое можно внедрить в свойство отдельно; в приведенном примере демонстрируется использование дескрипторов `<value>` и `<ref>` для добавления к `Map` значения `String` и ссылки на бин. Для свойства `props` мы применяем дескриптор `<props>`, чтобы создать экземпляр `java.util.Properties` и заполнить его с помощью дескрипторов `<prop>`. Обратите внимание, что хотя дескриптор `<prop>` снабжается ключами подобно дескриптору `<entry>`, указывать значение `String` можно только для каждого свойства, относящегося к экземпляру `Properties`.

Оба дескриптора, `<list>` и `<set>`, работают совершенно одинаково: каждый элемент указывается с применением любых дескрипторов значений, таких как `<value>` и `<ref>`, которые предназначены для внедрения одиночного значения в свойство. В листинге 3.52 к экземплярам `List` и `Set` мы добавили значение `String` и ссылку на бин.

Ниже показан вывод, сгенерированный классом из листинга 3.52. Как и можно было ожидать, он просто выводит список элементов, добавленных в коллекцию в конфигурационном файле.

```
Map contents:

Key: someValue - Value: Hello World!
Key: someBean - Value: com.apress.prospring4.ch3.xml.
BookwormOracle@6a4f787b
```

```
Properties contents:
Key: secondName - Value: Schaefer
Key: firstName - Value: Chris
Set contents: )
Value: Hello World!
Value: com.apress.prospring4.ch3.xml.BookwormOracle@6a4f787b
List contents:
Value: Hello World!
Value: com.apress.prospring4.ch3.xml.BookwormOracle@6a4f787b
```

Вспомните, что с элементами <list>, <map> и <set> можно задействовать любой из дескрипторов, которые используются при установке значений свойств, отличных от коллекций, для указания значения одного из элементов в коллекции. Это очень мощная концепция, поскольку вы не ограничены одним лишь внедрением коллекций элементарных значений; вы можете также внедрять коллекции бинов или другие коллекции.

Посредством такой функциональности разрабатываемое приложение намного проще разбивать на модули и предоставлять различные выбираемые пользователем реализации ключевых частей логики приложения. Подумайте о системе, которая позволяет корпоративным сотрудникам создавать, корректировать и заказывать персонализированные канцелярские принадлежности в онлайновом режиме. В этой системе завершенное изображение каждого заказа отправляется на подходящий принтер, когда он готов к работе. Единственная сложность связана с тем, что одни принтеры ожидают получения изображений по электронной почте, другие — по FTP, а третьи имеют дело с протоколом защищенного копирования (Secure Copy Protocol — SCP). Применяя внедрение коллекций Spring, можно создать стандартный интерфейс для этой функциональности, который приведен в листинге 3.54.

**Листинг 3.54. Интерфейс `ArtworkSender`**

```
package com.apress.prospring4.ch3;

public interface ArtworkSender {
    void sendArtwork(String artworkPath, Recipient recipient);
    String getFriendlyName();
    String getShortName();
}
```

В листинге 3.54 класс `Recipient` является пустым. Для интерфейса `ArtworkSender` можно создавать множество реализаций, каждая из которых способна описать себя самостоятельно вроде показанной в листинге 3.55.

**Листинг 3.55. Класс `FtpArtworkSender`**

```
package com.apress.prospring4.ch3;

public class FtpArtworkSender implements ArtworkSender {
    @Override
    public void sendArtwork(String artworkPath, Recipient recipient) {
        // Логика работы с FTP...
    }
```

```
    @Override
    public String getFriendlyName() {
        return "File Transfer Protocol";
    }
    @Override
    public String getShortName() {
        return "ftp";
    }
}
```

Предположим, что затем разрабатывается класс `ArtworkManager`, который поддерживает все доступные реализации интерфейса `ArtworkSender`. Имея такие реализации, вы просто передаете `List` в класс `ArtworkManager`. Сконфигурировав каждый шаблон канцелярских принадлежностей, с помощью метода `getFriendlyName()` вы можете отобразить для системного администратора список вариантов доставки. Вдобавок ваше приложение может оставаться полностью непривязанным к индивидуальным реализациям, если вы просто пишете код для интерфейса `ArtworkSender`. Реализацию класса `ArtworkManager` мы оставляем в качестве упражнения для самостоятельной проработки.

Кроме XML-конфигурации для внедрения коллекций можно использовать аннотации. Однако для простоты сопровождения имеет смысл также вынести значения коллекций в конфигурационный файл. В листинге 3.56 приведена конфигурация для четырех разных бинов Spring, которые имитируют те же свойства коллекций, что и предыдущий пример (`app-context-annotation.xml`).

**Листинг 3.56. Конфигурирование внедрения коллекций (стиль аннотаций)**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:util="http://www.springframework.org/schema/util"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util.xsd">
    <context:annotation-config/>
    <context:component-scan
        base-package="com.apress.prospring4.ch3.annotation"/>
    <util:map id="map" map-class="java.util.HashMap">
        <entry key="someValue">
            <value>Hello World!</value>
        </entry>
        <entry key="someBean">
            <ref bean="oracle"/>
        </entry>
    </util:map>)
```

```
    <util:properties id="props">
        <prop key="firstName">Chris</prop>
        <prop key="secondName">Schaefer</prop>
    </util:properties>

    <util:set id="set">
        <value>Hello World!</value>
        <ref bean="oracle"/>
    </util:set>

    <util:list id="list">
        <value>Hello World!</value>
        <ref bean="oracle"/>
    </util:list>
</beans>
```

Давайте также разработаем версию класса `BookwormOracle` с аннотациями. Код класса показан в листинге 3.57.

**Листинг 3.57. Класс `BookwormOracle`**

```
package com.apress.prospring4.ch3.annotation;

import org.springframework.stereotype.Service;

import com.apress.prospring4.ch3.Oracle;

@Service("oracle")
public class BookwormOracle implements Oracle {
    @Override
    public String defineMeaningOfLife() {
        return "Encyclopedias are a waste of money - use the Internet";
    }
}
```

В конфигурации из листинга 3.56 мы применяем пространство имен `util`, предоставляемое Spring, для объявления бинов, хранящих свойства коллекций. По сравнению с предшествующими версиями Spring конфигурация значительно упростилась. В тестовом классе мы внедряем ранее показанные бины и используем аннотации `@Resource` стандарта JSR-250 с указанием в них имен (листинг 3.58).

**Листинг 3.58. Применение аннотации `@Resource` при внедрении коллекций**

```
package com.apress.prospring4.ch3.annotation;

import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

import org.springframework.stereotype.Service;
import org.springframework.context.support.GenericXmlApplicationContext;

import javax.annotation.Resource;

@Service("injectCollection")
```

```
public class CollectionInjection {
   @Resource(name="map")
   private Map<String, Object> map;
   @Resource(name="props")
   private Properties props;
   @Resource(name="set")
   private Set set;
   @Resource(name="list")
   private List list;
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();
      CollectionInjection instance =
          (CollectionInjection) ctx.getBean("injectCollection");
      instance.displayInfo();
   }
   public void displayInfo() {
      System.out.println("Map contents:\n");
      for (Map.Entry<String, Object> entry: map.entrySet()) {
         System.out.println("Key: " + entry.getKey() + " - Value: "
             + entry.getValue());
      }
      System.out.println("\nProperties contents:\n");
      for (Map.Entry<Object, Object> entry: props.entrySet()) {
         System.out.println("Key: " + entry.getKey() + " - Value: "
             + entry.getValue());
      }
      System.out.println("\nSet contents:\n");
      for (Object obj: set) {
         System.out.println("Value: " + obj); ";
      }
      System.out.println("\nList contents:\n");
      for (Object obj: list) {
         System.out.println("Value: " + obj);
      }
   }
}
```

Запуск тестовой программы дает тот же самый результат, что и пример, в котором используется XML-конфигурация.

**На заметку!** Вас может удивить, почему вместо `@Autowired` применяется аннотация `@Resource`. Причина в том, что аннотация `@Autowired` семантически определена так, что она всегда трактует массивы, коллекции и карты как наборы соответствующих бинов, с целевым типом бина, производным от объявленного типа значений в коллекции. Таким образом, например, если класс имеет атрибут типа `List<Oracle>` и определена аннотация `@Autowired`, то платформа Spring попытается внедрить все бины типа `Oracle` из текущего контекста `ApplicationContext` в этот атрибут (вместо `<util:list>`, объявленного в конфигурационном файле), что приведет либо к внедрению неожиданных зависимостей, либо к генерации платформой Spring исключения при отсутствии бинов типа `Oracle`. В результате для внедрения типа коллекции понадобится явно сообщить Spring о необходимости выполнить внедрение, указав имя бина, что и поддерживает аннотация `@Resource`.

## Использование внедрения через метод

Кроме внедрения через конструктор и метод установки Spring предоставляет еще одно менее часто используемое средство DI, которое называется внедрением через метод (Method Injection). Возможности Method Injection доступны в двух слабо связанных формах — Lookup Method Injection (Внедрение через метод поиска) и Method Replacement (Замена метода). Форма Lookup Method Injection поддерживает еще один механизм, с помощью которого бин может получить одну из своих зависимостей. Форма Method Replacement позволяет заменять реализацию любого метода бина произвольным образом, без необходимости в изменении первоначального исходного кода. Для обеспечения этих двух средств Spring применяет возможности динамического расширения байт-кода, обеспечиваемые библиотекой CGLIB.

### *Тип Lookup Method Injection*

Тип Lookup Method Injection был добавлен в Spring версии 1.1 для преодоления проблем, возникающих, когда один бин зависит от другого бина, который имеет отличающийся жизненный цикл — в частности, когда одиночный объект (singleton) зависит от неодиночного объекта. В такой ситуации Setter Injection и Constructor Injection приводят к тому, что одиночный бин поддерживает единственный экземпляр того, что должно быть неодиночным бином. В некоторых случаях необходимо, чтобы одиночный бин при возникновении потребности получал новый экземпляр неодиночного бина.

Рассмотрим сценарий, в котором класс `LockOpener` предоставляет службу открытия любого шкафчика. Для открытия шкафчика `LockOpener` полагается на класс `KeyHelper`, который внедрен в `LockOpener`. Однако проектное решение класса `KeyHelper` предусматривает наличие ряда внутренних состояний, что делает этот класс непригодным для многократного использования. Каждый раз, когда вызывается метод `openLock()`, требуется новый экземпляр `KeyHelper`. В этом случае `LockOpener` будет одиночным объектом. Однако при внедрении класса `KeyHelper` с применением нормального механизма будет повторно использоваться тот же самый экземпляр класса `KeyHelper` (который был создан, когда платформа Spring выполняла внедрение в первый раз). Чтобы обеспечить передачу нового экземпляра `KeyHelper` методу `openLock()` при каждом его вызове, необходимо применять внедрение типа Lookup Method Injection.

Обычно этого можно добиться, заставив одиночный бин реализовать интерфейс `ApplicationContextAware` (который рассматривается в следующей главе). После этого, используя экземпляр `ApplicationContext`, одиночный бин может искать новый экземпляр неодиночной зависимости каждый раз, когда он в нем нуждается. Внедрение типа Lookup Method Injection позволяет одиночному бину объявить, что ему требуется неодиночная зависимость, и что он будет получать новый экземпляр неодиночного бина всегда, когда ему нужно взаимодействовать с ним, без необходимости в реализации любого специфичного для Spring интерфейса.

Внедрение Lookup Method Injection работает за счет объявления в одиночном бине метода поиска, который возвращает экземпляр неодиночного бина. При получении ссылки на одиночный бин внутри приложения в действительности получается ссылка на динамически созданный подкласс, в котором Spring реализует метод поиска. Типичная реализация включает определение метода поиска как абстрак-

тного. Это предотвращает возникновение странных ошибок в ситуации, когда вы забыли сконфигурировать Method Injection и работаете напрямую с классом бина, имеющим пустую реализацию метода, а не с подклассом, расширенным Spring. Это достаточно сложная тема, которую лучше проиллюстрировать с помощью примера.

В этом примере мы создаем неодиночный бин и два одиночных бина, реализующие тот же самый интерфейс. Первый одиночный бин получает экземпляр неодиночного бина, применяя "традиционный" подход с внедрением Setter Injection, а второй одиночный бин использует Method Injection. В листинге 3.59 показан бин `MyHelper`, который в данном примере играет роль неодиночного бина.

**Листинг 3.59. Бин `MyHelper`**

```
package com.apress.prospring4.ch3;

public class MyHelper {
    public void doSomethingHelpful() {
        // Какая-то обработка.
    }
}
```

Этот бин определенно не впечатляет, но он прекрасно подходит для целей рассматриваемого примера. В листинге 3.60 приведен интерфейс `DemoBean`, который реализуют оба одиночных бина.

**Листинг 3.60. Интерфейс `DemoBean`**

```
package com.apress.prospring4.ch3;

public interface DemoBean {
    MyHelper getMyHelper();
    void someOperation();
}
```

Интерфейс `DemoBean` имеет два метода: `getMyHelper()` и `someOperation()`. В примере приложения метод `getMyHelper()` применяется для получения ссылки на экземпляр `MyHelper`, а в случае бина с методом поиска — для выполнения действительного поиска. `someOperation()` — это простой метод, который при своей работе полагается на класс `MyHelper`.

В листинге 3.61 показан класс `StandardLookupDemoBean`, который использует Setter Injection для получения экземпляра класса `MyHelper`.

**Листинг 3.61. Класс `StandardLookupDemoBean`**

```
package com.apress.prospring4.ch3;

public class StandardLookupDemoBean implements DemoBean {
    private MyHelper myHelper;
    public void setMyHelper(MyHelper myHelper) {
        this.myHelper = myHelper;
    }
    @Override
    public MyHelper getMyHelper() {
        return this.myHelper;
    }
```

```
    @Override
    public void someOperation() {
        myHelper.doSomethingHelpful();
    }
}
```

Этот код должен выглядеть знакомым, но обратите внимание, что метод `someOperation()` для выполнения своей обработки применяет сохраненный экземпляр `MyHelper`. В листинге 3.62 приведен класс `AbstractLookupDemoBean`, в котором используется Method Injection для получения экземпляра класса `MyHelper`.

**Листинг 3.62. Класс `AbstractLookupDemoBean`**

```
package com.apress.prospring4.ch3;

public abstract class AbstractLookupDemoBean implements DemoBean {
    public abstract MyHelper getMyHelper();

    @Override
    public void someOperation() {
        getMyHelper().doSomethingHelpful();
    }
}
```

Метод `getMyHelper()` объявлен как абстрактный, и он вызывается методом `someOperation()` для получения экземпляра `MyHelper`. В листинге 3.63 показана конфигурация, необходимая для этого примера (`app-context-xml.xml`).

**Листинг 3.63. Конфигурирование внедрения через метод поиска**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="helper"
        class="com.apress.prospring4.ch3.MyHelper" scope="prototype"/>

    <bean id="abstractLookupBean"
        class="com.apress.prospring4.ch3.AbstractLookupDemoBean">
        <lookup-method name="getMyHelper" bean="helper"/>
    </bean>

    <bean id="standardLookupBean"
        class="com.apress.prospring4.ch3.StandardLookupDemoBean">
        <property name="myHelper">
            <ref bean="helper"/>
        </property>
    </bean>
</beans>
```

Конфигурация для вспомогательного бина и бина `standardLookupBean` должна теперь быть знакомой. Для `abstractLookupBean` необходимо сконфигурировать

метод поиска с применением дескриптора <lookup-method>. Атрибут name дескриптора <lookup-method> сообщает платформе Spring имя метода бина, который ей следует переопределить. Этот метод не должен принимать аргументы, а возвращаемым типом должен быть бин, который нужно вернуть из метода. В данном случае метод должен возвращать класс MyHelper или какой-то из его подклассов. Атрибут bean указывает Spring, какой бин должен возвращать метод поиска.

Оставшаяся часть кода для этого примера приведена в листинге 3.64.

**Листинг 3.64. Класс LookupDemo**

```
package com.apress.prospring4.ch3;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.util.StopWatch;

public class LookupDemo {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        DemoBean abstractBean = (DemoBean) ctx.getBean("abstractLookupBean");
        DemoBean standardBean = (DemoBean) ctx.getBean("standardLookupBean");

        displayInfo(standardBean);
        displayInfo(abstractBean);
    }

    public static void displayInfo(DemoBean bean) {
        MyHelper helper1 = bean.getMyHelper();
        MyHelper helper2 = bean.getMyHelper();

        System.out.println("Helper Instances the Same?: "
            + (helper1 == helper2));

        StopWatch stopWatch = new StopWatch();
        stopWatch.start("lookupDemo");

        for (int x = 0; x < 100000; x++) {
            MyHelper helper = bean.getMyHelper();
            helper.doSomethingHelpful();
        }

        stopWatch.stop();

        System.out.println("100000 gets took "
            + stopWatch.getTotalTimeMillis() + " ms");
    }
}
```

В этом коде мы извлекаем abstractLookupBean (создание экземпляра абстрактного класса поддерживается только при использовании внедрения типа Lookup Method Injection, при котором Spring будет применять CGLIB для генерации подкласса AbstractLookupDemoBean, динамически переопределяющего этот метод) и standardLookupBean из GenericXMLApplicationContext, после чего передаем каждую ссылку методу displayInfo(). В первой части метода displayInfo() создаются две локальные переменные MyHelper и каждой при-

сваивается значение путем вызова метода `getMyHelper()` на переданном бине. Используя эти две переменные, он записывает в `stdout` сообщение, которое позволяет понять, указывают ли две ссылки на один и тот же объект. В случае класса `abstractLookupBean` новый экземпляр `MyHelper` должен извлекаться для каждого вызова `getMyHelper()`, поэтому ссылки не должны быть одинаковыми. В случае класса `standardLookupBean` бину передается одиночный экземпляр `MyHelper` с помощью Setter Injection, при этом данный экземпляр сохраняется и возвращается для каждого вызова `getMyHelper()`, так что две ссылки должны совпадать.

---

**На заметку!** Класс `StopWatch`, применяемый в предыдущем примере, является служебным классом, доступным в Spring. Вы найдете этот класс очень полезным в ситуациях, когда необходимо проводить простые оценки производительности и тестировать разрабатываемые приложения.

---

В финальной части метода `displayInfo()` выполняется простой тест производительности для выяснения, какой из бинов быстрее. Очевидно, что `standardLookupBean` должен быть быстрее, потому что каждый раз он возвращает один и тот же экземпляр, но нам интересно увидеть эту разницу.

Запуск класса `LookupDemo` (листинг 3.64) для тестирования дает следующий вывод:

```
Helper Instances the Same?: true
100000 gets took 8 ms
Helper Instances the Same?: false
100000 gets took 1039 ms
```

Легко заметить, что экземпляры вспомогательного бина, как и ожидалось, одинаковы в случае использования `standardLookupBean` и отличаются в случае применения `abstractLookupBean`. Разница в производительности заметна, но это также ожидалось.

### *Соображения по поводу внедрения через метод поиска*

Внедрение через метод поиска (Lookup Method Injection) предназначено для использования в ситуации, когда нужно работать с двумя бинами, имеющими разные жизненные циклы. Не поддавайтесь искушению применять Lookup Method Injection, если бины разделяют один и тот же жизненный цикл, особенно когда они являются одиночными. В листинге 3.64 демонстрируется заметная разница в производительности между использованием Lookup Method Injection для получения новых экземпляров зависимости и применением стандартного DI для получения одиночного экземпляра зависимости. Кроме того, внедрением Lookup Method Injection не следует пользоваться без нужды, даже если имеются бины с разными жизненными циклами.

Представим ситуацию, когда есть три одиночных бина, которые разделяют общую зависимость. Каждый одиночный бин должен иметь собственный экземпляр зависимости, поэтому зависимость создается как неодиночный объект, но требуется, чтобы каждый одиночный бин использовал один и тот же экземпляр коллаборатора на протяжении всего времени жизни. В этом случае идеальным решением будет Setter Injection, а Lookup Method Injection просто привнесет нежелательные накладные расходы.

Во время применения Lookup Method Injection существует ряд принципов проектирования, которые должны быть приняты во внимание при построении классов. В показанных ранее примерах мы объявляли метод поиска в интерфейсе. Единственная причина такого решения заключалась в том, что мы не хотели дублировать метод `displayInfo()` в двух разных типах бинов. Как уже упоминалось, в общем случае вы не должны засорять бизнес-интерфейс ненужными определениями, которые предназначены исключительно для целей IoC. Есть еще один момент, который следует иметь в виду. Хотя вы не обязаны делать метод поиска абстрактным, это поможет не забыть сконфигурировать метод поиска, чтобы случайно не пользоваться его пустой реализацией.

### *Тип Method Replacement*

Несмотря на то что в документации Spring замена метода (Method Replacement) классифицируется как форма внедрения, этот прием сильно отличается от того, что было показано до сих пор. Ранее внедрение применялось исключительно для поставки бинов с их коллабораторами. Используя Method Replacement, можно заменить реализацию любого метода любого бина произвольным образом, без необходимости в изменении исходного кода этого бина. Например, пусть в приложении используется библиотека третьей стороны, и требуется изменить логику определенного метода. Однако вы не в состоянии изменить исходный код, поскольку он был предоставлен третьей стороной, так что единственное решение — применить Method Replacement для замены логики данного метода собственной реализацией.

Внутренне это достигается созданием подкласса для класса бина динамическим образом. Вы используете библиотеку CGLIB и перенаправляете обращения к методу, подлежащему замене, другому бину, который реализует интерфейс `MethodReplacer`.

В листинге 3.65 показан простой бин, в котором объявлены две перегруженных версии метода `formatMessage()`.

**Листинг 3.65. Класс `ReplacementTarget`**

```
package com.apress.prospring4.ch3;

public class ReplacementTarget {
    public String formatMessage(String msg) {
        return "<h1>" + msg + "</h1>";
    }

    public String formatMessage(Object msg) {
        return "<h1>" + msg + "</h1>";
    }
}
```

С помощью функциональности Method Replacement, предлагаемой Spring, можно заменить любой метод класса `ReplacementTarget`. В этом примере мы покажем, как заменить метод `formatMessage(String)`, а также сравним производительность замененного и исходного методов.

Чтобы заменить метод, сначала понадобится создать реализацию интерфейса `MethodReplacer` (листинг 3.66).

**Листинг 3.66. Реализация интерфейса `MethodReplacer`**

```
package com.apress.prospring4.ch3;

import java.lang.reflect.Method;

import org.springframework.beans.factory.support.MethodReplacer;

public class FormatMessageReplacer implements MethodReplacer {
    @Override
    public Object reimplement(Object arg0, Method method, Object[] args)
            throws Throwable {
        if (isFormatMessageMethod(method)) {
            String msg = (String) args[0];

            return "<h2>" + msg + "</h2>";
        } else {
            throw new IllegalArgumentException("Unable to reimplement method "
                    + method.getName());
        }
    }

    private boolean isFormatMessageMethod(Method method) {
        if (method.getParameterTypes().length != 1) {
            return false;
        }

        if (!("formatMessage".equals(method.getName()))) {
            return false;
        }

        if (method.getReturnType() != String.class) {
            return false;
        }

        if (method.getParameterTypes()[0] != String.class) {
            return false;
        }

        return true;
    }
}
```

Интерфейс `MethodReplacer` имеет единственный метод `reimplement()`, который вы должны реализовать. Этому методу передаются три аргумента: бин, на котором вызывается исходный метод, экземпляр `Method`, представляющий переопределяемый метод, и массив аргументов, передаваемых методу. Метод `reimplement()` должен возвращать результат замененной логики и, очевидно, что тип возвращаемого значения должен быть совместим с возвращаемым типом заменяемого метода.

В листинге 3.66 класс `FormatMessageReplacer` сначала проверяет, переопределяется ли метод `formatMessage(String)`; если это так, он вызывает заменяющую логику (в данном случае она окружает сообщение парой `<h2>` и `</h2>`) и возвращает сформатированное сообщение вызывающему коду. Проверять правильность сообщения не обязательно, но это полезно в случае использования нескольких реализаций интерфейсов `MethodReplacer` с похожими аргументами. Применение проверки помогает избежать ситуации, при которой случайно используется другой `MethodReplacer` с совместимыми аргументами и возвращаемыми типами.

В листинге 3.67 приведен код ApplicationContext, в котором определены два бина типа ReplacementTarget — один имеет замененный метод formatMessage(String), а другой нет (app-context-xml.xml).

**Листинг 3.67. Конфигурирование замены метода**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="methodReplacer"
     class="com.apress.prospring4.ch3.FormatMessageReplacer"/>

   <bean id="replacementTarget"
      class="com.apress.prospring4.ch3.ReplacementTarget">
      <replaced-method name="formatMessage" replacer="methodReplacer">
         <arg-type>String</arg-type>
      </replaced-method>
   </bean>

   <bean id="standardTarget"
      class="com.apress.prospring4.ch3.ReplacementTarget"/>
</beans>
```

Как видите, реализация MethodReplacer объявлена как бин в ApplicationContext. Затем с применением дескриптора <replaced-method> производится замена метода formatMessage(String) бина replacementTargetBean.

В атрибуте name дескриптора <replaced-method> указывается имя заменяемого метода, а в атрибуте replacer — имя бина MethodReplacer, который должен заменить реализацию метода. При наличии перегруженных методов, как в классе ReplacementTarget, с помощью дескриптора <arg-type> можно указать необходимую сигнатуру метода. Дескриптор <arg-type> поддерживает сопоставление по образцу, поэтому String будет соответствовать java.lang.String и также java.lang.StringBuffer.

В листинге 3.68 приведено простое демонстрационное приложение, которое извлекает бины standardTarget и replacementTarget из ApplicationContext, запускает их методы formatMessage(String) и затем прогоняет несложный тест производительности, чтобы определить, какой из методов быстрее.

**Листинг 3.68. Замена метода в действии**

```
package com.apress.prospring4.ch3;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.util.StopWatch;

public class MethodReplacementExample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
```

```
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        ReplacementTarget replacementTarget = (ReplacementTarget) ctx
                .getBean("replacementTarget");
        ReplacementTarget standardTarget = (ReplacementTarget) ctx
                .getBean("standardTarget");

        displayInfo(replacementTarget);
        displayInfo(standardTarget);
    }

    private static void displayInfo(ReplacementTarget target) {
        System.out.println(target.formatMessage("Hello World!"));

        StopWatch stopWatch = new StopWatch();
        stopWatch.start("perfTest");

        for (int x = 0; x < 1000000; x++) {
            String out = target.formatMessage("foo");
        }

        stopWatch.stop();

        System.out.println("1000000 invocations took: "
                + stopWatch.getTotalTimeMillis() + " ms");
    }
}
```

Этот код должен выглядеть очень знакомым, поэтому мы не будем вдаваться в какие-либо детали. На нашей машине запуск этого примера дал следующий вывод:

```
<h2>Hello World!</h2>
1000000 invocations took: 396 ms
<h1>Hello World!</h1>
1000000 invocations took: 18 ms
```

Как и ожидалось, вывод из бина `replacementTarget` показывает, что выполнялась переопределенная реализация, предоставленная `MethodReplacer`. Однако интересно, что динамически замененный метод функционирует во много раз медленнее, чем метод, определенный статически. Удаление проверки правильности метода из `MethodReplacer` приводит лишь к незначительной разнице при множестве запусков, поэтому можно заключить, что большая часть накладных расходов относится к подклассу CGLIB.

### *Когда использовать замену метода*

Замена метода может оказаться весьма полезной в различных обстоятельствах, особенно когда нужно переопределить только отдельный метод в единственном бине, а не во всех бинах одного типа. Тем не менее, для переопределения методов мы по-прежнему предпочитаем пользоваться стандартными механизмами Java, а не полагаться на динамическое расширение байт-кода.

Если вы собираетесь применять замену метода как часть своего приложения, мы рекомендуем использовать один интерфейс `MethodReplacer` на метод или группу перегруженных методов. Не поддавайтесь искушению применять единственную реализацию интерфейса `MethodReplacer` для множества несвязанных друг с другом методов; это приведет к выполнению дополнительных ненужных сравнений `String`

с целью выяснения, какой метод должен получить новую реализацию. Мы обнаружили, что проведение простых проверок того, что `MethodReplacer` работает с правильным методом, является полезным и не добавляет слишком много накладных расходов. Если вас действительно заботит вопрос производительности, можете предусмотреть в `MethodReplacer` булевское свойство, которое позволит включать и отключать такую проверку с использованием Dependency Injection.

## Именование бинов

Платформа Spring поддерживает довольно сложную структуру именования бинов, которая предоставляет в ваше распоряжение высокую гибкость в обработке множества ситуаций. Каждый бин должен иметь, по крайней мере, одно имя, которое является уникальным внутри содержащего бин контекста `ApplicationContext`. При определении имени бина платформа Spring следует простому процессу распознавания. Если в дескрипторе `<bean>` предусмотрен атрибут `id`, его значение применяется в качестве имени. Когда атрибут `id` не указан, Spring ищет атрибут `name`, и если он определен, то используется первое имя, заданное в атрибуте `name`. (Мы говорим *первое имя*, потому что в атрибуте `name` допускается определять множество имен; этот момент вскоре будет раскрыт более подробно.) Если не указан ни атрибут `id`, ни атрибут `name`, в качестве имени платформа Spring применяет имя класса бина, разумеется, при условии, что другие бины не используют то же самое имя класса. В случае если множество бинов с отсутствующими атрибутами `id` и `name` имеют одно и то же имя класса, Spring сгенерирует исключение (типа `org.springframework.beans.factory.NoSuchBeanDefinitionException`) при внедрении во время инициализации `ApplicationContext`. В листинге 3.69 показан пример конфигурации, в которой применяются все три схемы именования.

**Листинг 3.69. Именование бинов**

```
<bean id="string1" class="java.lang.String"/>
<bean name="string2" class="java.lang.String"/>
<bean class="java.lang.String"/>
```

Формально все эти подходы одинаково допустимы, но какой из них лучше выбрать для приложения? Прежде всего, избегайте использования автоматических имен, создаваемых на основе поведения классов. Это существенно снижает гибкость в случае определения множества бинов одного типа, потому гораздо лучше назначать им имена самостоятельно. В таком случае, если стандартное поведение Spring в будущем изменится, ваше приложение продолжит нормально функционировать. Выбирая, какой атрибут лучше применять, `id` или `name`, всегда отдавайте предпочтение `id` для указания стандартного имени бина. До выхода версии Spring 3.1 атрибут `id` совпадал с идентификатором XML (т.е. `xsd:ID`), что накладывало ограничение на символы, которые можно было использовать. Начиная с версии Spring 3.1, для атрибута `id` применяется тип `xsd:String`, поэтому существовавшее ранее ограничение относительно символов устранено. Тем не менее, платформа Spring продолжает контролировать уникальность `id` в рамках всего контекста `ApplicationContext`. В общем случае вы должны назначать своему бину имя с использованием атрибута `id` и затем ассоциировать этот бин с другими именами, создавая псевдонимы имен, как объясняется в следующем разделе.

### *Создание псевдонимов для имен бинов*

Платформа Spring разрешает бину иметь более одного имени. Для этого в атрибуте `name` дескриптора `<bean>` бина необходимо указать список имен, разделенных пробелами, запятыми или точками с запятой. Это можно сделать вместо применения атрибута `id` или в сочетании с ним.

Чтобы определить псевдонимы для имен бинов Spring, кроме атрибута `name` можно воспользоваться дескриптором `<alias>`. В листинге 3.70 приведена простая конфигурация `<bean>`, в которой для единственного бина определено множество имен (`app-context-xml.xml`).

**Листинг 3.70. Конфигурирование множества имен для бина**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="name1" name="name2 name3,name4;name5" class="java.lang.String"/>
   <alias name="name1" alias="name6"/>
</beans>
```

Как видите, для бина определены шесть имен. Первое имя указано в атрибуте `id` и еще четыре имени заданы в атрибуте `name` в виде списка, в котором применяются все разрешенные разделители (это сделано только в демонстрационных целях; в реальных приложениях так поступать не рекомендуется). При реальной разработке следует стандартизировать разделитель для имен бинов в объявлениях. Еще один псевдоним определен с помощью дескриптора `<alias>`. В листинге 3.71 представлен пример Java-программы, которая извлекает тот же самый бин из `ApplicationContext` шесть раз, используя разные имена и проверяя идентичность полученного бина.

**Листинг 3.71. Доступ к бинам с применением псевдонимов**

```
package com.apress.prospring4.ch3.xml;

import org.springframework.context.support.GenericXmlApplicationContext;

public class BeanNameAliasing {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-xml.xml");
      ctx.refresh();

      String s1 = (String) ctx.getBean("name1");
      String s2 = (String) ctx.getBean("name2");
      String s3 = (String) ctx.getBean("name3");
      String s4 = (String) ctx.getBean("name4");
      String s5 = (String) ctx.getBean("name5");
      String s6 = (String) ctx.getBean("name6");

      System.out.println((s1 == s2));
      System.out.println((s2 == s3));
```

```
        System.out.println((s3 == s4));
        System.out.println((s4 == s5));
        System.out.println((s5 == s6));
    }
}
```

Для конфигурации, показанной в листинге 3.70, этот код выводит на консоль значение `true` пять раз, подтверждая то, что бины, доступ к которым осуществляется посредством разных имен, в действительности являются одним и тем же бином.

Извлечь список псевдонимов бина можно с помощью вызова метода `ApplicationContext.getAliases(String)` с передачей ему любого из имен бина или его идентификатора. Этот метод возвращает в виде массива `String` список псевдонимов, содержащий все псевдонимы кроме указанного при вызове.

Создание псевдонимов для имен бинов не является средством, которое будет обязательно использоваться при построении приложения. Если заранее планируется, что в приложении будет множество бинов, которые внедряются в другой бин, то внедряемые бины могут для доступа к целевому бину применять одно и то же имя. Однако при эксплуатации приложения и его сопровождении могут вноситься модификации, и вот тут может пригодиться создание псевдонимов для имен.

Рассмотрим следующий сценарий: имеется приложение, в котором все 50 бинов, сконфигурированные с использованием Spring, требуют реализации интерфейса `Foo`. При этом 25 бинов пользуются реализацией `StandardFoo` с именем бина `standardFoo`, а другие 25 — реализацией `SuperFoo` с именем бина `superFoo`. Через полгода после передачи приложения в эксплуатацию принимается решение о переводе первых 25 бинов на реализацию `SuperFoo`. Есть три варианта, как сделать это.

- Первый вариант — изменение реализации класса бина `standardFoo` на `SuperFoo`. Недостаток этого подхода в том, что появляются два экземпляра класса `SuperFoo`, когда в действительности необходим только один. Вдобавок, когда изменится конфигурация, придется вносить изменения в два бина, а не в один.
- Второй вариант — обновление конфигурации внедрения для 25 изменяемых бинов, в которой имена бинов меняются с `standardFoo` на `superFoo`. Это не самый элегантный подход: можно воспользоваться поиском и заменой, но последующий откат изменений, если решение окажется неприемлемым, означает необходимость извлечения старой версии кода из системы управления версиями.
- Третий и наиболее идеальный вариант — удаление (или комментирование) определения бина `standardFoo` и назначение `standardFoo` в качестве псевдонима для `superFoo`. Такое изменение требует минимальных усилий и упрощает восстановление системы в ее прежней конфигурации.

## Режим создания экземпляров бинов

По умолчанию все бины в Spring являются одиночными. Это значит, что Spring обслуживает одиночный экземпляр бина, все зависимые объекты используют один и тот же экземпляр и все вызовы `ApplicationContext.getBean()` возвращают тот же самый экземпляр. Мы демонстрировали это в примере, показанном в листинге 3.71, где для проверки идентичности бинов была возможность применять операцию сравнения `==`, а не `equals()`.

Термин *одиночный* (singleton) используется в Java для ссылки на две отличающиеся концепции: объект, который имеет единственный экземпляр в рамках приложения, и шаблон проектирования Singleton. Первую концепцию мы будем называть одиночным объектом, а вторую без изменений — шаблоном проектирования Singleton. Этот шаблон проектирования стал популярным благодаря книге *Design Patterns: Elements of Reusable Object Oriented Software* Эриха Гаммы и др. (Addison-Wesley, 1995 г.). Проблема возникает, когда начинают путать потребность в одиночных экземплярах и необходимость в применении шаблона проектирования Singleton. Типовая реализация упомянутого шаблона в Java приведена в листинге 3.72.

**Листинг 3.72. Шаблон проектирования Singleton**

```
package com.apress.prospring4.ch3;

public class Singleton {
    private static Singleton instance;

    static {
        instance = new Singleton();
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```

Данный шаблон достигает своей цели, позволяя поддерживать и оперировать единственным экземпляром класса по всему приложению, но делает это за счет увеличения степени связанности. Код приложения должен всегда явно знать о классе `Singleton`, чтобы получить его экземпляр, и это приводит к полному исчезновению возможности вынесения кода в интерфейсы. В действительности шаблон Singleton представляет собой два шаблона проектирования в одном. Первый, и он же желательный, шаблон включает обслуживание единственного экземпляра объекта. Второй, менее желательный, шаблон касается поиска объекта, и это полностью устраняет возможность использования интерфейсов. Применение шаблона Singleton также существенно затрудняет замену реализаций произвольным образом, поскольку большинство объектов, которым требуется экземпляр `Singleton`, получают доступ к объекту `Singleton` напрямую. Это приведет к возникновению разнообразных сложностей при попытке модульного тестирования приложения, т.к. заменить объект Singleton имитированным объектом в целях тестирования не получится.

К счастью, в Spring можно воспользоваться моделью создания одиночного экземпляра, не прибегая непосредственно к шаблону проектирования Singleton. Все бины в Spring по умолчанию создаются как экземпляры `Singleton`, и для обработки всех запросов к конкретному бину Spring применяется один и тот же экземпляр. Естественно, платформа Spring не ограничивается использованием только одиночного экземпляра; можно по-прежнему создавать новый экземпляр бина для удовлетворения каждой зависимости и каждого обращения к методу `getBean()`. Все это делается без какого-либо влияния на код приложения, и по этой причине мы предпочитаем говорить, что платформа Spring является *независимой от режима создания экземпляров*. На самом деле это очень мощная концепция. Если в начале разработки

вы считаете, что какой-то объект является одиночным, но впоследствии обнаруживаете, что он не подходит для многопоточного доступа, можете спокойно изменить объект на неодиночный, не повлияв на код приложения.

**На заметку!** Хотя изменение режима создания экземпляров не влияет на код приложения, оно может привести к возникновению ряда проблем, если вы полагаетесь на интерфейсы жизненного цикла Spring. Более детально об этом пойдет речь в главе 4.

Изменить режим создания экземпляров с одиночного на неодиночный довольно просто, как демонстрируется в листинге 3.73 (`app-context-xml.xml`).

**Листинг 3.73. Конфигурация неодиночного бина**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:c="http://www.springframework.org/schema/c"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="nonSingleton" class="java.lang.String" scope="prototype"
      c:_0="Chris Schaefer"/>
</beans>
```

Как видите, единственным отличием между этим объявлением бина и объявлениями, которые приводились ранее, является добавление атрибута `scope` (область действия) и установка его значения в `prototype` (прототип). По умолчанию Spring устанавливает атрибут `scope` в `singleton` (одиночный). Область действия на уровне прототипа заставляет Spring создавать новый экземпляр бина каждый раз, когда он запрашивается приложением. В листинге 3.74 можно видеть, какой эффект эта установка оказывает на приложение.

**Листинг 3.74. Неодиночные бины в действии**

```
package com.apress.prospring4.ch3;

import org.springframework.context.support.GenericXmlApplicationContext;

public class NonSingleton {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-xml.xml");
      ctx.refresh();

      String s1 = (String) ctx.getBean("nonSingleton");
      String s2 = (String) ctx.getBean("nonSingleton");

      System.out.println("Identity Equal?: " + (s1 ==s2));
      System.out.println("Value Equal:? " + s1.equals(s2));
      System.out.println(s1);
      System.out.println(s2);
   }
}
```

Выполнение этого примера дает следующий вывод:

```
Identity Equal?:  false
Value Equal:? true
Chris Schaefer
Chris Schaefer
```

Вывод показывает, что хотя значения двух объектов `String` одинаковы, сами сущности различаются, несмотря на то, что оба экземпляра были получены с использованием одного и того же имени бина.

### *Выбор режима создания экземпляров*

В большинстве сценариев довольно просто выяснить, какой режим создания экземпляров является для них подходящим. Обычно мы считаем одиночный режим стандартным для своих бинов. В общем случае одиночные экземпляры должны применяться в следующих сценариях.

- **Совместно используемый объект без состояния.** Имеется объект, не поддерживающий состояние, и множество зависимых от него объектов. Поскольку состояние не поддерживается и, следовательно, синхронизация не нужна, не потребуется создавать новый экземпляр бина каждый раз, когда он необходим зависимому объекту для какой-то обработки.
- **Совместно используемый объект с состоянием только для чтения.** Этот сценарий похож на предыдущий, но здесь поддерживается состояние, предназначенное только для чтения. В данном случае синхронизация по-прежнему не нужна, поэтому создание экземпляра для каждого запроса бина просто увеличивает накладные расходы.
- **Совместно используемый объект с совместно используемым состоянием.** Если нужен бин, состояние которого должно использоваться совместно, то одиночный экземпляр будет идеальным выбором. В этом случае потребуется обеспечить как можно более детализированную синхронизацию при записи состояния.
- **Объекты с высоким коэффициентом использования и записываемым состоянием.** Если есть бин, который интенсивно применяется в приложении, вы можете обнаружить, что сохранение его одиночным и синхронизация всего доступа к нему по записи позволяют обеспечить более высокую производительность, чем постоянное создание сотен экземпляров этого объекта. Используя такой подход, старайтесь обеспечить как можно более детализированную синхронизацию без ущерба для согласованности. Этот подход особенно полезен в ситуациях, когда приложение создает большое количество экземпляров в течение длительного периода времени, когда совместно используемый объект имеет лишь небольшую часть записываемого состояния или когда создание нового экземпляра сопряжено с большими расходами ресурсов.

В перечисленных ниже сценариях должны применяться неодиночные экземпляры.

- **Объекты с записываемым состоянием.** Если имеется бин с большим объемом записываемого состояния, может выясниться, что затраты на синхронизацию превышают затраты на создание нового экземпляра для обработки каждого запроса от зависимого объекта.

- **Объекты с закрытым состоянием.** В ряде случаев зависимым объектам нужен бин, который имеет закрытое состояние, так что зависимые объекты могут проводить свою обработку отдельно друг от друга. В такой ситуации одиночный экземпляр точно не подходит, поэтому необходим неодиночный экземпляр.

Главное преимущество, которое вы получаете от управления созданием экземпляров Spring, связано с уменьшением потребления памяти приложениями благодаря одиночным экземплярам, причем с вашей стороны для этого понадобятся совсем небольшие усилия. Кроме того, если окажется, что режим одиночных экземпляров не соответствует потребностям приложения, довольно легко перенастроить конфигурацию на использование режима неодиночных экземпляров.

### *Реализация областей действия бинов*

В дополнение к областям действия на уровне одиночного экземпляра и прототипа при определении бина Spring, предназначенного для более специфичных целей, доступны другие области действия. Можно также реализовать специальную область действия и зарегистрировать ее в `ApplicationContext`. Ниже приведен список областей действия, которые поддерживаются в версии Spring 4.

- **Одиночный экземпляр.** Стандартная область действия. Будет создаваться только один объект на контейнер Spring IoC.
- **Прототип.** Платформа Spring будет создавать новый экземпляр, когда он запрашивается приложением.
- **Запрос.** Для применения в веб-приложениях. Когда для построения веб-приложений используется Spring MVC, бины с областью действия на уровне запроса будут создаваться для каждого HTTP-запроса и уничтожаться по завершении его обработки.
- **Сеанс.** Для применения в веб-приложениях. Когда для построения веб-приложений используется Spring MVC, бины с областью действия на уровне сеанса будут создаваться для каждого HTTP-сеанса и уничтожаться после его завершения.
- **Глобальный сеанс.** Для использования в веб-приложениях, основанных на портлетах. Бины с областью действия на уровне глобального сеанса могут совместно применяться всеми портлетами внутри портального приложения Spring MVC.
- **Поток.** Платформа Spring будет создавать новый экземпляр бина, когда он запрашивается новым потоком, и возвращать один тот же экземпляр бина при поступлении запроса из того же самого потока. Обратите внимание, что по умолчанию эта область действия не зарегистрирована.
- **Специальная.** Специальная область действия бина, которую можно создать за счет реализации интерфейса `org.springframework.beans.factory.config.Scope` и регистрации области действия в конфигурации Spring (для стиля XML используйте класс `org.springframework.beans.factory.config.CustomScopeConfigurer`).

# Распознавание зависимостей

Во время нормального функционирования платформа Spring способна распознавать зависимости, просто просматривая конфигурационный файл или аннотации в классах. Подобным образом Spring может гарантировать, что все бины сконфигурированы в правильном порядке, так что каждый бин имеет корректно настроенные зависимости. Если платформа Spring не выполнит этого, а просто создаст бины и сконфигурирует их в произвольном порядке, может оказаться, что какой-нибудь бин будет создан и настроен до его зависимостей. Очевидно, что при этом в приложении могут возникать различные проблемы.

К сожалению, Spring ничего не известно о зависимостях, которые существуют между бинами в коде. Например, пусть один бин по имени `beanA` получает экземпляр другого бина под названием `beanB` в своем конструкторе через вызов `getBean()`. Например, в конструкторе бина `beanA` получается экземпляр бина `beanB` посредством вызова `ctx.getBean("beanB")` без уведомления Spring о необходимости внедрения зависимости. В этом случае платформа Spring не знает о том, что `beanA` зависит от `beanB`, в результате чего она может создать экземпляр `beanA` перед созданием экземпляра `beanB`. Предоставить Spring дополнительную информацию о зависимостях между бинами можно с помощью атрибута `depends-on` дескриптора `<bean>`. В листинге 3.75 показано, как можно было бы сконфигурировать `beanA` и `beanB`.

**Листинг 3.75. Ручное определение зависимостей**

```
<bean id="beanA" class="com.apress.prospring4.ch3.BeanA" depends-on="beanB"/>
<bean id="beanB" class="com.apress.prospring4.ch3.BeanB"/>
```

В этой конфигурации мы утверждаем, что бин по имени `beanA` зависит от бина под названием `beanB`. При создании экземпляров бинов платформа Spring примет это во внимание и обеспечит создание экземпляра `beanB` раньше экземпляра `beanA`.

Во время разработки приложений такого подхода лучше избегать; вместо этого определяйте зависимости с помощью контрактов Setter Injection и Constructor Injection. Однако если Spring интегрируется с унаследованным кодом, может оказаться, что определенные в нем зависимости требуют предоставления дополнительной информации для Spring Framework.

# Автосвязывание бина

Во всех показанных до сих пор примерах с помощью конфигурационного файла явно определялось, каким образом отдельные бины связаны друг с другом. Если вы не желаете делать это самостоятельно, можете предоставить Spring возможность попытаться связать компоненты автоматически. По умолчанию автосвязывание отключено. Чтобы включить его, в атрибуте `autowire` для бина понадобится указать используемый режим автосвязывания.

## Режимы автосвязывания

Платформа Spring поддерживает следующие режимы автосвязывания: `byName`, `byType`, `constructor`, `default` и `no` (автосвязывание отключено; устанавливается по умолчанию). Когда применяется автосвязывание `byName`, платформа Spring пытается связать каждое свойство с бином, имеющим такое же имя. Таким образом, если целевой бин имеет свойство по имени `foo` и в `ApplicationContext` определен бин `foo`, то этот бин `foo` назначается свойству `foo` целевого бина.

При использовании автосвязывания `byType` платформа Spring пытается связать каждое свойство целевого бина с бином того же самого типа в `ApplicationContext`. Это значит, что если в целевом бине присутствует свойство типа `String` и в `ApplicationContext` определен некоторый бин типа `String`, то Spring свяжет бин `String` со свойством `String` целевого бина. Если в `ApplicationContext` определено более одного бина типа `String`, то Spring не сможет решить, какой из них выбрать для автосвязывания, и сгенерирует исключение (типа `org.springframework.beans.factory.NoSuchBeanDefinitionException`).

Режим автосвязывания `constructor` функционирует подобно режиму `byType` за исключением того, что для выполнения внедрения он применяет конструкторы, а не методы установки. Платформа Spring пытается найти соответствие с как можно большим числом аргументов в конструкторе. Таким образом, если бин имеет два конструктора, первый из которых принимает аргумент `String`, а второй — аргументы `String` и `Integer`, и при этом в `ApplicationContext` определены бины `String` и `Integer`, Spring будет использовать конструктор с двумя аргументами.

В режиме `default` платформа Spring будет автоматически производить выбор между режимами `constructor` и `byType`. Если бин имеет стандартный конструктор (без аргументов), то Spring применяет режим `byType`, а в противном случае — режим `constructor`.

В листинге 3.76 показана простая конфигурация, которая автоматически связывает три бина одного и того же типа с использованием всех описанных режимов (`app-context-xml.xml`).

**Листинг 3.76. Конфигурирование автосвязывания**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="foo" class="com.apress.prospring4.ch3.Foo"/>
   <bean id="bar1" class="com.apress.prospring4.ch3.Bar"/>
   <bean id="targetByName" autowire="byName"
     class="com.apress.prospring4.ch3.xml.Target"
      lazy-init="true"/>
   <bean id="targetByType" autowire="byType"
      class="com.apress.prospring4.ch3.xml.Target"
      lazy-init="true"/>
   <bean id="targetConstructor" autowire="constructor"
      class="com.apress.prospring4.ch3.xml.Target" lazy-init="true"/>
</beans>
```

Эта конфигурация должна выглядеть очень знакомой. Здесь `Foo` и `Bar` — пустые классы. Обратите внимание, что каждый бин `Target` имеет отличающееся значение в атрибуте `autowire`. Кроме того, атрибут `lazy-init` установлен в `true`, чтобы информировать Spring о необходимости создания экземпляра бина только при первом его запросе, а не во время начальной загрузки, поэтому мы можем выводить результаты в правильном месте внутри тестовой программы. В листинге 3.77 приведен код простого Java-приложения, которое извлекает каждый бин `Target` из `ApplicationContext`.

**Листинг 3.77. Автосвязывание коллабораторов**

```
package com.apress.prospring4.ch3.xml;

import org.springframework.context.support.GenericXmlApplicationContext;
import com.apress.prospring4.ch3.Foo;
import com.apress.prospring4.ch3.Bar;

public class Target {
    private Foo foo;
    private Foo foo2;
    private Bar bar;

    public Target() {
    }

    public Target(Foo foo) {
        System.out.println("Target(Foo) called");
    }

    public Target(Foo foo, Bar bar) {
        System.out.println("Target(Foo, Bar) called");
    }

    public void setFoo(Foo foo) {
        this.foo = foo;
        System.out.println("Property foo set");
    }

    public void setFoo2(Foo foo) {
        this.foo2 = foo;
        System.out.println("Property foo2 set");
    }
    public void setBar(Bar bar) {
        this.bar = bar;
        System.out.println("Property bar set");
    }
    public static void main(String[] args) {
       GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
       ctx.load("classpath:META-INF/spring/app-context-xml.xml");
       ctx.refresh();

       Target t = null;
       System.out.println("Using byName:\n");
       t = (Target) ctx.getBean("targetByName");
```

```
        System.out.println("\nUsing byType:\n");
        t = (Target) ctx.getBean("targetByType");

        System.out.println("\nUsing constructor:\n");
        t = (Target) ctx.getBean("targetConstructor");

    }
}
```

В этом коде класс `Target` имеет три конструктора: конструктор без аргументов, конструктор, который принимает экземпляр `Foo`, и конструктор, принимающий экземпляры `Foo` и `Bar`. В дополнение к перечисленным конструкторам бин `Target` располагает тремя свойствами, из которых два имеют тип `Foo` и одно тип `Bar`. Все свойства и конструкторы при вызове выводят сообщения на консоль. Метод `main()` просто извлекает все бины `Target`, объявленные в `ApplicationContext`, и тем самым инициирует процесс автосвязывания. Ниже показан вывод, полученный в результате выполнения этого примера:

```
Using byName:

Property foo set

Using byType:

Property bar set
Property foo set
Property foo2 set

Using constructor:

Target(Foo, Bar) called
```

В выводе видно, что когда Spring применяет режим `byName`, единственным устанавливаемым свойством является `foo`, т.к. одно это свойство имеет соответствующую запись бина в конфигурационном файле. При использовании режима `byType` платформа Spring устанавливает значения трех свойств. Свойства `foo` и `foo2` установлены бином `foo`, а свойство `bar` — бином `bar1`. В случае применения режима `constructor` платформа Spring выбирает конструктор с двумя аргументами, потому что она может предоставить бины для обоих аргументов и не нуждается в обращении к другому конструктору.

## Когда используется автосвязывание

В большинстве случаев ответ на вопрос, должно ли использоваться автосвязывание, будет определенно отрицательным. Автосвязывание может сэкономить время в небольших приложениях, но во многих ситуациях оно приводит к неудачным решениям и утере гибкости в крупных приложениях. Применение режима `byName` выглядит неплохой идеей, но может потребовать назначения свойствам в классах искусственных имен, чтобы удалось задействовать функциональность автосвязывания. Вся идея Spring заключается в том, что вы создаете свои классы так, как вам нравится, и позволяете Spring работать с ними, а не наоборот. Искушение использовать режим `byType` будет возникать до тех пор, пока вы не поймете, что в `ApplicationContext` может существовать только один бин для каждого типа — ограничение, которое создаст проблемы, когда необходимо обслуживать бины одно-

го и того же типа, но с различными конфигурациями. Аналогичное замечание касается также применения автосвязывания `constructor`.

В ряде случаев автосвязывание может сберечь время, однако явное определение связывания не требует настолько много дополнительных усилий, чтобы отказываться от преимуществ явной семантики и полной свободы в отношении именования свойств и произвольного количества экземпляров бина одного и того же типа, которыми можно управлять. При разработке любого нетривиального приложения по возможности держитесь от автосвязывания подальше.

## Настройка наследования бинов

В некоторых ситуациях может понадобиться множество определений бинов одного и того же типа или реализация совместно используемого интерфейса. Это может стать проблематичным, если бины должны совместно использовать одни параметры конфигурации и не делать этого в отношении других. Процесс сохранения совместно используемых параметров конфигурации в синхронизированном состоянии подвержен ошибкам, а в крупных проектах еще и может отнимать довольно много времени. Чтобы обойти это, Spring позволяет создать определение `<bean>`, которое наследует настройки своих свойств от другого бина в том же самом контексте `ApplicationContext`. При необходимости значения любых свойств дочернего бина можно переопределить, что позволяет иметь над ними полный контроль, но родительский бин может предоставлять каждому дочернему бину базовую конфигурацию. В листинге 3.78 приведена простая конфигурация с двумя бинами, один из которых является дочерним по отношению к другому (`app-context-xml.xml`).

**Листинг 3.78. Конфигурирование наследования бинов**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="inheritParent" class="com.apress.prospring4.ch3.xml.SimpleBean"
      p:name="Chris Schaefer" p:age="32"/>
   <bean id="inheritChild" class="com.apress.prospring4.ch3.xml.SimpleBean"
      parent="inheritParent" p:age="33"/>
</beans>
```

В этом коде дескриптор `<bean>` для бина `inheritChild` имеет дополнительный атрибут `parent`, который указывает, что платформа Spring должна считать бин `inheritParent` родительским для определяемого бина. В случае если определение родительского бина не должно быть доступным для поиска из `ApplicationContext`, к дескриптору `<bean>`, объявляющему родительский бин, можно добавить атрибут `abstract="true"`. Поскольку бин `inheritChild` имеет собственное значение для свойства `age`, Spring передает это значение бину. Однако `inheritChild` не устанавливает значение для свойства `name`, поэтому Spring использует значение, заданное в бине `inheritParent`. В листинге 3.79 показан код класса `SimpleBean`, применяемого в предыдущей конфигурации.

**Листинг 3.79. Класс SimpleBean**

```
package com.apress.prospring4.ch3.xml;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SimpleBean {
    private String name;
    private int age;

    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        SimpleBean parent = (SimpleBean) ctx.getBean("inheritParent");
        SimpleBean child = (SimpleBean) ctx.getBean("inheritChild");

        System.out.println("Parent:\n" + parent);
        System.out.println("Child:\n" + child);
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String toString() {
        return "Name: " + name + "\n" + "Age: " + age;
    }
}
```

Как видите, метод `main()` класса `SimpleBean` извлекает бины `inheritChild` и `inheritParent` из контекста `ApplicationContext` и записывает содержимое их свойств в `stdout`. Ниже представлен вывод, полученный в результате выполнения этого примера:

```
Parent:
Name: Chris Schaefer
Age: 32
Child:
Name: Chris Schaefer
Age: 33
```

Бин `inheritChild` вполне ожидаемо унаследовал от бина `inheritParent` значение для своего свойства `name`, но предоставил собственное значение для свойства `age`.

Дочерние бины наследуют аргументы конструкторов и значения свойств от родительских бинов, поэтому при наследовании бинов можно использовать оба стиля внедрения. Такой уровень гибкости превращает наследование бинов в мощный инструмент для построения приложений с большим количеством бинов. При объявлении множества бинов с одинаковыми значениями в совместно используемых

свойствах избегайте соблазна применять копирование и вставку для дублирования этих значений; вместо этого настройте в своей конфигурации подходящую иерархию наследования.

Помните, что наследование бинов не обязательно должно совпадать с иерархией наследования классов Java. Вполне допустимо использовать наследование бинов, скажем, для пяти бинов одного и того же типа. Наследование бинов больше похоже на средство создания шаблонов, чем на собственно наследование. Однако при изменении типа дочернего бина имейте в виду, что этот тип должен быть совместимым с типом родительского бина.

# Резюме

В этой главе мы раскрыли много общих понятий, связанных с ядром Spring и IoC. Были продемонстрированы примеры различных типов IoC и приведены обсуждения достоинств и недостатков применения каждого механизма в разрабатываемых приложениях. Мы посмотрели, какие механизмы IoC предлагает Spring, и объяснили, когда их использовать в приложениях, а когда нет. Во время исследования IoC был представлен интерфейс `BeanFactory`, являющийся ключевым компонентом для средств IoC в Spring, а также интерфейс `ApplicationContext`, который расширяет `BeanFactory` и предоставляет дополнительную функциональность. Во время рассмотрения `ApplicationContext` мы сосредоточили внимание на классе `GenericXmlApplicationContext`, который делает возможным применение внешней конфигурации Spring с помощью XML. Также обсуждался еще один способ объявления требований DI для `ApplicationContext` — использование Java-аннотаций.

В главе были представлены основы набора функциональных средств IoC в Spring, включая Setter Injection, Constructor Injection, Method Injection, автосвязывание и наследование бинов. При обсуждении конфигурации мы продемонстрировали настройку свойств бинов с применением широкого разнообразия значений, в том числе других бинов, конфигураций в стиле XML и аннотаций, а также класса `GenericXmlApplicationContext`.

В данной главе мы лишь слегка коснулись поверхности Spring и встроенного в Spring контейнера IoC. В следующей главе мы рассмотрим некоторые связанные с IoC возможности, специфичные для Spring, и взглянем более внимательно на другую функциональность, доступную в ядре Spring.

ГЛАВА 4

# Детальные сведения о конфигурации Spring

В предыдущей главе мы представили концепцию инверсии управления (Inversion of Control — IoC) и показали, как она вписывается в инфраструктуру Spring Framework. Тем не менее, мы только слегка коснулись того, что способно делать ядро Spring. Платформа Spring предоставляет широкий спектр служб, которые дополняют и расширяют ее базовые возможности IoC. В этой главе мы собираемся взглянуть на все это детально. В частности, в главе будут рассмотрены следующие темы.

- **Управление жизненным циклом бинов.** Все бины, которые вы видели до сих пор, были довольно простыми и совершенно не привязанными к контейнеру Spring. В этой части главы вы ознакомитесь со стратегиями, которыми можно воспользоваться, чтобы позволить бинам получать уведомления от контейнера Spring в различных точках их жизненного цикла. Это можно сделать за счет реализации специфических интерфейсов, предлагаемых Spring, путем указания методов, которые Spring может вызывать через рефлексию, или с применением аннотаций жизненного цикла компонентов JavaBean, описанных в спецификации JSR-250.
- **Превращение бинов в "осведомленные о платформе Spring".** В некоторых случаях необходимо, чтобы бин имел возможность взаимодействовать со сконфигурировавшим его контекстом `ApplicationContext`. Для этих целей Spring предлагает два интерфейса, `BeanNameAware` и `ApplicationContextAware`, которые позволяют бину получать назначенное ему имя и ссылку на его контекст `ApplicationContext`, соответственно. В данной части главы раскрываются реализации этих интерфейсов и приводятся некоторые соображения по поводу их практического использования в приложении.
- **Использование фабрик бинов.** Интерфейс `FactoryBean` предназначен для реализации любым бином, который действует в качестве фабрики для других бинов. Интерфейс `FactoryBean` предоставляет механизм, с помощью которого легко интегрировать собственные фабрики с интерфейсом `BeanFactory` из Spring.
- **Работа с редакторами свойств для компонентов JavaBean.** Интерфейс `PropertyEditor` — это стандартный интерфейс, предоставляемый пакетом `java.beans`. Редакторы свойств (реализующие `PropertyEditor`) используются для преобразования значений свойств в и из представлений `String`.

Платформа Spring применяет редакторы свойств повсеместно, в основном для чтения значений, указанных в конфигурации `BeanFactory`, и преобразования их в корректные типы. В этой части главы мы обсудим набор редакторов свойств, поставляемых Spring, и покажем, как их использовать в рамках приложения. Также будет кратко объясняться реализация собственных редакторов свойств.

- **Дополнительные сведения о расширении `ApplicationContext`.** Как вы уже знаете, `ApplicationContext` — это расширение `BeanFactory`, предназначенное для применения в полнофункциональных приложениях. Интерфейс `ApplicationContext` предоставляет полезный набор дополнительной функциональности, включая поддержку интернационализированных сообщений, загрузку ресурсов и публикацию событий. В данной части главы мы детально рассмотрим средства, дополняющие возможности IoC, которые предлагает `ApplicationContext`. Мы также забежим немного вперед и покажем, каким образом `ApplicationContext` упрощает использование Spring при построении веб-приложений.

- **Использование Java-классов в конфигурации.** До версии 3.0 платформа Spring поддерживала только базовую конфигурацию XML с аннотациями для конфигурирования бинов и зависимостей. Начиная с версии 3.0, Spring предлагает разработчикам еще один вариант конфигурирования `ApplicationContext` с применением Java-классов. Мы кратко рассмотрим этот новый вариант конфигурации Spring-приложений.

- **Использование расширений конфигурации.** Мы представим средства, упрощающие конфигурирование приложений, такие как управление профилями, абстракция среды и источников свойств и т.д. В этой части главы мы раскроем эти средства и покажем, как их применять для решения специфических потребностей конфигурации.

- **Использование Groovy для конфигурирования.** В версии Spring 4.0 появилась новая возможность конфигурирования определений бинов на языке Groovy, который может применяться в качестве альтернативы или дополнения существующих методов конфигурации посредством XML и Java.

# Влияние Spring на переносимость приложений

Большинство функциональных средств, обсуждаемых в этой главе, являются специфичными для Spring, и во многих случаях они не доступны в других контейнерах IoC. Хотя многие контейнеры IoC поддерживают функциональность управления жизненным циклом, возможно, они делают это через набор интерфейсов, которые отличаются от предлагаемых в Spring. Если переносимость приложения между разными контейнерами IoC действительно важна, вы можете решить избегать использования тех средств, которые привязывают приложение к платформе Spring.

Однако помните, что за счет установки ограничения, связанного с переносимостью приложения между контейнерами IoC, вы теряете все богатство функциональности, обеспечиваемой Spring. Поскольку вы наверняка приняли стратегическое решение в пользу применения платформы Spring, имеет смысл извлечь максимум из ее возможностей.

Остерегайтесь создавать требование о переносимости на пустом месте. Во многих случаях конечных пользователей не волнует тот факт, что приложение может выполняться в трех разных контейнерах IoC; им просто нужно его запустить. Согласно нашему опыту, частой ошибкой является попытка построить приложение на основе "наименьшего общего знаменателя" для функциональных средств, доступных в рамках выбранной технологии. Это может поставить создаваемое приложение в невыгодные условия с самого начала. Тем не менее, если приложение должно быть переносимым между контейнерами IoC, не следует трактовать это требование как недостаток — это нормальное требование, которому, помимо прочих, приложение должно удовлетворять. В книге *Expert One-on-One: J2EE Development without EJB* (Wrox, 2004 г.) такие типы требований называются фантомными требованиями; там же можно найти развернутое обсуждение их влияния на разрабатываемые проекты.

Несмотря на то что использование этих функциональных средств может привязывать приложение к платформе Spring Framework, на самом деле переносимость приложения увеличивается в гораздо большей степени. Подумайте о том, что вы имеете дело со свободно доступной платформой с открытым кодом, не принадлежащей какому-то конкретному поставщику. Приложение, построенное с применением контейнера IoC из Spring, выполняется везде, где функционирует машина Java. Для корпоративных Java-приложений Spring открывает новые возможности в плане переносимости. Платформа Spring предоставляет многие из средств, предлагаемых JEE, и также поддерживает классы для абстрагирования и упрощения множества других аспектов JEE. В большинстве ситуаций с помощью Spring можно построить веб-приложение, которое выполняется в простом сервлетном контейнере, но с таким же уровнем сложности, как у приложения, ориентированного на полномасштабный сервер приложений JEE. Привязываясь к Spring, вы увеличиваете степень переносимости своего приложения, заменяя многие функциональные средства, которые являются либо специфическими для поставщика, либо зависят от специфической для поставщика конфигурации, эквивалентными возможностями Spring.

## Управление жизненным циклом бинов

Важным аспектом любого контейнера IoC, включая Spring, является возможность конструирования бинов так, что они будут получать уведомления в определенных точках их жизненного цикла. Это позволяет бинам выполнять подходящую обработку в этих точках. В общем случае непосредственное отношение к бинам имеют два события жизненного цикла: событие после инициализации и событие перед уничтожением.

В контексте Spring событие *после инициализации* генерируется, когда Spring завершает установку всех значений свойств бина и все проверки зависимостей, которые были сконфигурированы для выполнения. Событие *перед уничтожением* инициируется непосредственно перед тем, как Spring приступает к уничтожению экземпляра бина. Однако для бинов с областью действия на уровне прототипа событие перед уничтожением Spring генерировать не будет. Проектное решение, лежащее в основе платформы Spring, предполагает выполнение методов обратного вызова жизненного цикла, связанных с инициализацией, для объектов независимо от области действия бинов, но для бинов с областью действия на уровне прототипа методы обратного вызова жизненного цикла, относящиеся к уничтожению, запускаться не будут.

Платформа Spring предоставляет три механизма, которые бин может использовать для привязки к каждому из указанных событий и выполнения дополнительной обработки: основанный на интерфейсах, основанный на методах и основанный на аннотациях.

В рамках механизма, основанного на интерфейсах, бин реализует интерфейс, специфичный для типа уведомлений, которые необходимо получать, и Spring уведомляет бин через метод обратного вызова, определенный в этом интерфейсе. В случае механизма, основанного на методах, Spring позволяет указать в конфигурации `ApplicationContext` имя метода, который должен быть вызван при инициализации бина, и имя метода, который должен быть вызван при уничтожении бина. Механизм, основанный на аннотациях, предполагает применение аннотаций JSR-250 для указания методов, которые платформа Spring должна вызывать после конструирования и перед уничтожением.

Для обоих событий эти механизмы достигают в точности одной и той же цели. Механизм, основанный на интерфейсах, широко используется в самой платформе Spring, поэтому при работе с компонентами Spring нет необходимости помнить о том, что нужно указывать инициализацию и уничтожение. Тем не менее, в собственных бинах может быть лучше применять механизм, основанный на методах или аннотациях, поскольку ваши бины не обязаны реализовывать интерфейсы, специфичные для Spring. Хотя мы утверждаем, что переносимость часто не является настолько важной, как это заявлено во многих книгах, это не означает, что вы должны жертвовать переносимостью, когда существует вполне хорошая альтернатива. Другими словами, если вы привязали свое приложение к Spring другими путями, то использование метода интерфейса позволит определить обратный вызов только один раз, после чего забыть о нем. В ситуации, когда определяется множество однотипных бинов, которым необходимы уведомления жизненного цикла, применение механизма, основанного на интерфейсах, позволяет избежать необходимости в указании методов обратного вызова жизненного цикла для каждого бина в XML-файле конфигурации. Использование аннотаций JSR-250 — еще один приемлемый вариант, т.к. это стандарт, определенный JCP, и вдобавок вы не привязываетесь к аннотациям, специфичным для Spring. Просто удостоверьтесь, что контейнер IoC, в котором выполняется приложение, поддерживает стандарт JSR-250.

В целом выбор механизма, который будет применяться для получения уведомлений жизненного цикла, зависит от требований вашего приложения. Если вас интересует переносимость либо вы просто определяете один или два бина заданного типа, которым нужны обратные вызовы, используйте механизм, основанный на методах. Если вы применяете конфигурацию в стиле аннотаций и уверены, что целевой контейнер IoC поддерживает стандарт JSR-250, используйте механизм, основанный на аннотациях. Если вы не слишком обеспокоены переносимостью или определяете множество однотипных бинов, которым необходимы уведомления жизненного цикла, то применение механизма, основанного на интерфейсах, будет наилучшим способом обеспечить получение бинами ожидаемых уведомлений. Если же вы планируете использовать какой-то бин во множестве разных проектов Spring, то наверняка хотите, чтобы функциональность этого бина была как можно более самодостаточной, поэтому в таком случае определенно понадобится применять механизм, основанный на интерфейсах.

На рис. 4.1 приведен высокоуровневый обзор того, как Spring управляет жизненным циклом бинов внутри их контейнера.

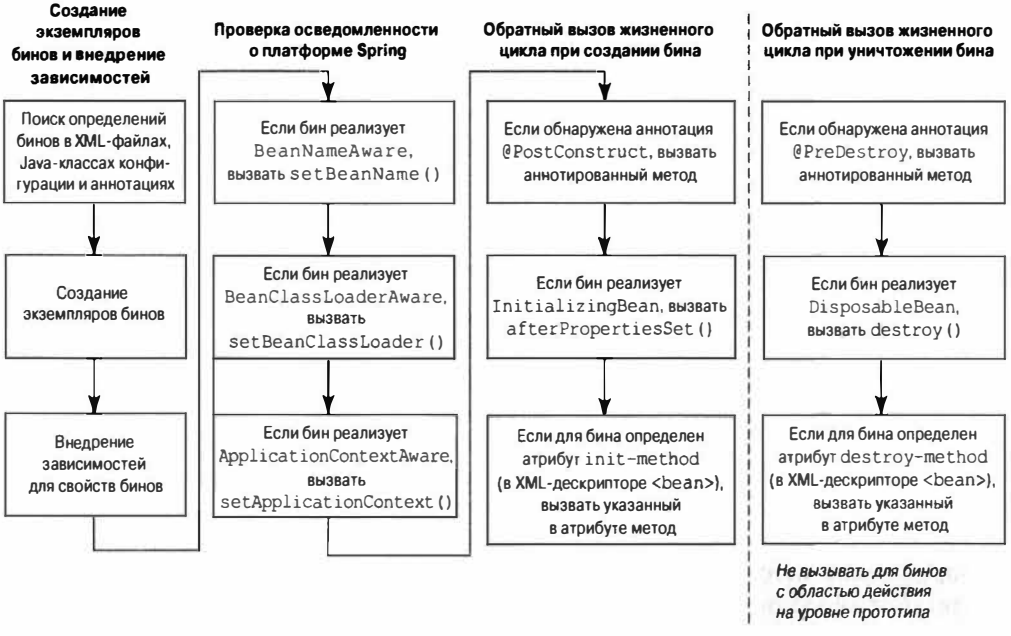


**Рис. 4.1.** Жизненный цикл бинов Spring

## Привязка к созданию бина

Будучи осведомленным о том, когда он инициализирован, бин может проверить, все ли требуемые зависимости удовлетворены. Хотя платформа Spring способна автоматически проверять зависимости, используемый ею подход можно охарактеризовать как "все или ничего", и она не предлагает каких-либо возможностей для применения дополнительной логики в процедуре распознавания зависимостей. Рассмотрим бин, который имеет четыре зависимости, объявленные как методы установки, причем две из них являются обязательными, а одна поддерживает подходящее стандартное значение на случай, когда зависимость не предоставлена. Используя обратный вызов инициализации, бин может проверить требующиеся ему зависимости и при необходимости сгенерировать исключение либо предоставить стандартное значение.

Бин не может выполнить такие проверки в своем конструкторе, потому что в этой точке Spring не имеет возможности предоставить значения для зависимостей. Обратный вызов инициализации инициируется после того, как Spring завершает предоставление зависимостей и может выполнить любые запрашиваемые проверки зависимостей.

Обратный вызов инициализации не ограничивается одной лишь проверкой зависимостей; в нем можно делать все, что угодно, но наиболее полезен этот обратный вызов как раз для той цели, которую мы описали выше. Во многих случаях обрат-

ный вызов инициализации также является местом для запуска любых действий, которые бин должен предпринимать автоматически в ответ на свою конфигурацию. Например, если вы строите бин для выполнения запланированных задач, то обратный вызов инициализации будет идеальным местом для запуска планировщика — в конце концов, конфигурационные данные для бина уже установлены.

---

**На заметку!** Вам не придется разрабатывать бин для запуска запланированных задач, поскольку Spring может это делать автоматически с помощью встроенного средства планирования или через интеграцию с планировщиком Quartz. Более детально эти вопросы рассматриваются в главе 11.

---

### *Выполнение метода, когда бин создается*

Как упоминалось ранее, один из способов получения обратного вызова инициализации предусматривает назначение некоторого метода бина в качестве метода инициализации и сообщение об этом платформе Spring. Этот механизм обратного вызова полезен, когда имеется всего один бин определенного типа или когда необходимо, чтобы приложение оставалось непривязанным к Spring. Еще одна причина применения этого механизма заключается в том, чтобы позволить приложению Spring работать с бинами, которые были построены ранее или предоставлены независимыми поставщиками.

Определение метода обратного вызова сводится просто к указанию имени в атрибуте `init-method` дескриптора `<bean>` для бина. В листинге 4.1 показан код бина с двумя зависимостями.

**Листинг 4.1. Класс `SimpleBean`**

```
package com.apress.prospring4.ch4;

import org.springframework.beans.factory.BeanCreationException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;

public class SimpleBean {
    private static final String DEFAULT_NAME = "Luke Skywalker";

    private String name;
    private int age = Integer.MIN_VALUE;

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void init() {
        System.out.println("Initializing bean"); // инициализация бина

       if (name == null) {
            System.out.println("Using default name");
                               // использование стандартного имени
            name = DEFAULT_NAME;
        }
```

```
        if (age == Integer.MIN_VALUE) {
            throw new IllegalArgumentException(
                    "You must set the age property of any beans of type "
                  // Должно быть установлено свойство age любого бина типа
                    + SimpleBean.class);
        }
    }
    public String toString() {
        return "Name: " + name + "\nAge: " + age;
    }
    public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-xml.xml");
      ctx.refresh();

      SimpleBean simpleBean1 = getBean("simpleBean1", ctx);
      SimpleBean simpleBean2 = getBean("simpleBean2", ctx);
      SimpleBean simpleBean3 = getBean("simpleBean3", ctx);
    }
    private static SimpleBean getBean(String beanName, ApplicationContext ctx)
    {
        try {
            SimpleBean bean = (SimpleBean) ctx.getBean(beanName);
            System.out.println(bean);
            return bean;
        } catch (BeanCreationException ex) {
            System.out.println("An error occured in bean configuration: "
                    + ex.getMessage()); // В конфигурации бина произошла ошибка
            return null;
        }
    }
}
```

Обратите внимание, что здесь определен метод `init()`, который будет служить обратным вызовом инициализации. Метод `init()` проверяет, установлено ли свойство `name`, и если это не так, то использует стандартное значение, хранящееся в константе `DEFAULT_NAME`. Кроме того, этот метод также проверяет, установлено ли свойство `age`, и генерирует исключение `IllegalArgumentException`, если оно не установлено.

Метод `main()` класса `SimpleBean` пытается получить три бина из контекста `GenericXmlApplicationContext`, имеющие тип `SimpleBean`, с применением собственного метода `getBean()`. Если бин получен успешно, то в методе `getBean()` на консоль выводится детальная информация о нем. Когда в методе `init()` возникает исключение, как и будет в случае, когда свойство `age` не установлено, Spring помещает его в оболочку `BeanCreationException`. Метод `getBean()` перехватывает такие исключения и записывает в консольный вывод сообщение, информирующее о возникшей ошибке, а также возвращает значение `null`.

В листинге 4.2 приведена конфигурация `ApplicationContext`, в которой определены бины, используемые в листинге 4.1 (`app-context-xml.xml`).

**Листинг 4.2. Конфигурирование бинов `SimpleBean`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd"
   default-lazy-init="true">
   <bean id="simpleBean1"
      class="com.apress.prospring4.ch4.SimpleBean"
      init-method="init" p:name="Chris Schaefer" p:age="32"/>
   <bean id="simpleBean2"
      class="com.apress.prospring4.ch4.SimpleBean"
      init-method="init" p:age="32"/>
   <bean id="simpleBean3"
      class="com.apress.prospring4.ch4.SimpleBean"
      init-method="init" p:name="Chris Schaefer"/>
</beans>
```

Как видите, дескрипторы `<bean>` для всех трех бинов содержат атрибут `init-method`, который сообщает платформе Spring, что она должна вызвать метод `init()`, как только завершит конфигурирование бина. Бин `simpleBean1` имеет значения для обоих свойств `name` и `age`, поэтому он проходит через метод `init()` совершенно без изменений. В бине `simpleBean2` не задано значение для свойства `name`, и это приводит к тому, что в методе `init()` свойство `name` получит стандартное значение. Наконец, в бине `simpleBean3` не указано значение для свойства `age`. Логика, реализованная в методе `init()`, трактует это как ошибку, поэтому генерируется исключение `IllegalArgumentException`. Также обратите внимание, что в дескриптор `<beans>` был добавлен атрибут `default-lazy-init="true"`, который инструктирует Spring о том, что экземпляры бинов, определенных в конфигурационном файле, должны создаваться только при их запросе приложением. Если не указать этот атрибут, то Spring попытается инициализировать все бины во время начальной загрузки `ApplicationContext` и потерпит неудачу при инициализации `simpleBean3`. Выполнение рассматриваемого примера дает следующий вывод:

```
Initializing bean
Name: Chris Schaefer
Age: 32
Initializing bean
Using default name
Name: Luke Skywalker
Age: 32
An error occured in bean configuration: Error creating bean with name
'simpleBean3' defined in class path resource [META-INF/spring/
app-context-xml.xml]: Invocation of init method failed; nested exception
is java.lang.IllegalArgumentException: You must set the age property of
any beans of type class com.apress.prospring4.ch4.SimpleBean
В конфигурации бина обнаружена ошибка: Ошибка создания бина с именем
simpleBean3, отпределенного в ресурсе пути классов [META-INF/spring/
app-context-xml.xml]: Сбой при вызове метода init; вложенным исключением
является java.lang.IllegalArgumentException: Должно быть установлено
свойство age любого бина типа com.apress.prospring4.ch4.SimpleBean
```

По этому выводу можно судить, что бин simpleBean1 был сконфигурирован корректно с применением значений, указанных в конфигурационном файле. В бине simpleBean2 для свойства name было использовано стандартное значение, поскольку в конфигурации никакого значения не было задано. Наконец, экземпляр бина simpleBean3 не был создан, т.к. метод init() сгенерировал исключение из-за отсутствия значения для свойства age.

Как видите, применение метода инициализации является идеальным способом удостовериться в правильности конфигурации бинов. Благодаря этому механизму вы можете получить все преимущества IoC, не теряя контроля, который дает определение зависимостей вручную. Единственное ограничение метода инициализации связано с тем, что он не может принимать аргументы. Можно определять любой возвращаемый тип, хотя он игнорируется Spring, и можно даже использовать статический метод, но этот метод не должен принимать аргументы.

Когда применяется статический метод инициализации, преимущества этого механизма сводятся на нет, поскольку отсутствует возможности получить доступ к состоянию любого бина для его проверки. Если бин использует статическое состояние как средство экономии памяти, и для его проверки определен статический метод инициализации, вы должны рассмотреть возможность перемещения статического состояния в состояние экземпляра и применения нестатического метода инициализации. Когда используются средства управления одиночными экземплярами, встроенные в Spring, конечный эффект будет таким же, но вы получаете бин, который намного проще тестировать, и также имеете дополнительную возможность создавать несколько экземпляров бина с собственными состояниями, если в этом возникнет необходимость. Конечно, в некоторых случаях требуется статическое состояние, совместно используемое множеством экземпляров бина, и вы всегда можете применять статический метод инициализации.

### *Реализация интерфейса `InitializingBean`*

Интерфейс InitializingBean, предлагаемый в Spring, позволяет определять внутри бина код, который будет выполняться, когда бин получает уведомление о том, что платформа Spring завершила его конфигурирования. Подобно методу инициализации, это дает возможность проверить допустимость конфигурации бина, предоставляя наряду с этим любые стандартные значения. В интерфейсе InitializingBean определен единственный метод afterPropertiesSet(), который служит той же цели, что и метод init() в листинге 4.1. В листинге 4.3 показан переделанный предыдущий пример, в котором вместо метода инициализации используется интерфейс InitializingBean.

**Листинг 4.3. Использование интерфейса `InitializingBean`**

```
package com.apress.prospring4.ch4;

import org.springframework.beans.factory.BeanCreationException;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;

public class SimpleBeanWithInterface implements InitializingBean {
   private static final String DEFAULT_NAME = "Luke Skywalker";
```

```
    private String name;
    private int age = Integer.MIN_VALUE;

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void myInit() {
        System.out.println("My Init");
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("Initializing bean");

        if (name == null) {
            System.out.println("Using default name");
            name = DEFAULT_NAME;
        }

        if (age == Integer.MIN_VALUE) {
            throw new IllegalArgumentException(
                    "You must set the age property of any beans of type " +
                    SimpleBeanWithInterface.class);
        }
    }

    public String toString() {
        return "Name: " + name + "\nAge: " + age;
    }

    public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-xml.xml");
      ctx.refresh();

      SimpleBeanWithInterface simpleBean1 = getBean("simpleBean1", ctx);
      SimpleBeanWithInterface simpleBean2 = getBean("simpleBean2", ctx);
      SimpleBeanWithInterface simpleBean3 = getBean("simpleBean3", ctx);
    }

    private static SimpleBeanWithInterface getBean(String beanName,
            ApplicationContext ctx) {
        try {
            SimpleBeanWithInterface bean =
                (SimpleBeanWithInterface) ctx.getBean(beanName);
            System.out.println(bean);
            return bean;
        } catch (BeanCreationException ex) {
            System.out.println("An error occured in bean configuration: "
                    + ex.getMessage());
            return null;
        }
    }
}
```

Как видите, изменений в этом примере не особенно много. Кроме очевидного изменения имени класса, единственная разница в том, что этот класс реализует интерфейс `InitializingBean`, а логика инициализации перемещена в метод `InitializingBean.afterPropertiesSet()`. В листинге 4.4 приведена конфигурация для данного примера (`app-context-xml.xml`).

**Листинг 4.4. Конфигурация для примера использования интерфейса `InitializingBean`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd"
   default-lazy-init="true">
   <bean id="simpleBean1"
      class="com.apress.prospring4.ch4.SimpleBeanWithInterface"
      p:name="Chris Schaefer" p:age="32"/>
   <bean id="simpleBean2"
      class="com.apress.prospring4.ch4.SimpleBeanWithInterface"
      p:age="32"/>
   <bean id="simpleBean3"
      class="com.apress.prospring4.ch4.SimpleBeanWithInterface"
      p:name="Chris Schaefer"/>
</beans>
```

И снова между конфигурациями в листингах 4.4 и 4.2 отличий не очень много. Заметная разница связана с отсутствием атрибута `init-method`. Поскольку класс `SimpleBeanWithInterface` реализует интерфейс `InitializingBean`, платформе Spring известно, какой метод инициировать в качестве обратного вызова инициализации, поэтому отпадает потребность в любой дополнительной конфигурации. Ниже показан вывод, полученный в результате запуска этого примера:

```
Initializing bean
Name: Chris Schaefer
Age: 32
Initializing bean
Using default name
Name: Luke Skywalker
Age: 32
An error occured in bean configuration: Error creating bean with name
'simpleBean3' defined in class path resource [META-INF/spring/app-
context-xml.xml]: Invocation of init method failed; nested exception is
java.lang.IllegalArgumentException: You must set the age property of any
beans of type class com.apress.prospring4.ch4.SimpleBeanWithInterface
В конфигурации бина обнаружена ошибка: Ошибка создания бина с именем
simpleBean3, отпределенного в ресурсе пути классов [META-INF/
spring/app-context-xml.xml]: Сбой при вызове метода init; вложенным
исключением является java.lang.IllegalArgumentException: Должно быть
установлено свойство age любого бина типа com.apress.prospring4.ch4.
SimpleBeanWithInterface
```

### Использование аннотации @PostConstruct стандарта JSR-250

Еще одним методом, с помощью которого можно достичь той же самой цели, является аннотация жизненного цикла `@PostConstruct`, определенная в стандарте JSR-250. Начиная с версии Spring 2.5, поддерживаются также и аннотации JSR-250 для указания метода, который платформа Spring должна вызвать, если соответствующая аннотация, связанная с жизненным циклом бина, существует в классе. В листинге 4.5 представлен код программы, в котором применяется аннотация `@PostConstruct`.

**Листинг 4.5. Использование аннотации `@PostConstruct`**

```
package com.apress.prospring4.ch4;

import javax.annotation.PostConstruct;
import org.springframework.beans.factory.BeanCreationException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;

public class SimpleBeanWithJSR250 {
    private static final String DEFAULT_NAME = "Luke Skywalker";

    private String name;
    private int age = Integer.MIN_VALUE;

    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }

    @PostConstruct
    public void init() throws Exception {
        System.out.println("Initializing bean");
       if (name == null) {
            System.out.println("Using default name");
            name = DEFAULT_NAME;
        }
        if (age == Integer.MIN_VALUE) {
            throw new IllegalArgumentException(
                  "You must set the age property of any beans of type " +
                   SimpleBeanWithJSR250.class);
        }
    }
    public String toString() {
        return "Name: " + name + "\nAge: " + age;
    }
    public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();

      SimpleBeanWithJSR250 simpleBean1 = getBean("simpleBean1", ctx);
      SimpleBeanWithJSR250 simpleBean2 = getBean("simpleBean2", ctx);
      SimpleBeanWithJSR250 simpleBean3 = getBean("simpleBean3", ctx);
    }
```

```
    private static SimpleBeanWithJSR250
          getBean(String beanName, ApplicationContext ctx) {
        try {
            SimpleBeanWithJSR250 bean =
                (SimpleBeanWithJSR250) ctx.getBean(beanName);
            System.out.println(bean);
            return bean;
        } catch (BeanCreationException ex) {
            System.out.println("An error occured in bean configuration: "
                  + ex.getMessage());
            return null;
        }
    }
}
```

Здесь используется точно такой же код, как и в случае подхода с `init-method`; просто к методу `init()` применяется аннотация `@PostConstruct`. Следует отметить, что этому методу можно назначить любое имя.

Что касается конфигурации, то поскольку используются аннотации, в конфигурационный файл понадобится добавить дескриптор `<context:annotation-config>` из пространства имен `context`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd"
    default-lazy-init="true">

    <context:annotation-config/>

    <bean id="simpleBean1"
        class="com.apress.prospring4.ch4.SimpleBeanWithJSR250"
        p:name="Chris Schaefer" p:age="32"/>

    <bean id="simpleBean2"
        class="com.apress.prospring4.ch4.SimpleBeanWithJSR250"
        p:age="32"/>

    <bean id="simpleBean3"
        class="com.apress.prospring4.ch4.SimpleBeanWithJSR250"
        p:name="Chris Schaefer"/>
</beans>
```

Запустив программу, вы увидите тот же самый вывод, что и производимый в случае применения других механизмов:

```
Initializing bean
Name: Chris Schaefer
Age: 32
Initializing bean
Using default name
```

```
Name: Luke Skywalker
Age: 32
Initializing bean
An error occured in bean configuration: Error creating bean with name
'simpleBean3': Invocation of init method failed; nested exception is
java.lang.IllegalArgumentException: You must set the age property of any
beans of type class com.apress.prospring4.ch4.SimpleBeanWithJSR250
```

*В конфигурации бина обнаружена ошибка: Ошибка создания бина с именем simpleBean3: Сбой при вызове метода init; вложенным исключением является java.lang.IllegalArgumentException: Должно быть установлено свойство age любого бина типа com.apress.prospring4.ch4.SimpleBeanWithJSR250*

Все три подхода обладают своими достоинствами и недостатками. Используя метод инициализации, вы получаете преимущество отсутствия привязки приложения к Spring, но при этом вы должны не забыть сконфигурировать метод инициализации для каждого бина, который в нем нуждается. Работа с интерфейсом `InitializingBean` дает преимущество наличия возможности указывать обратный вызов инициализации один раз для всех экземпляров класса бина, но при этом приложение привязывается к Spring. Аннотации требуют применения к методам, а также проведения проверки, поддерживает ли контейнер IoC стандарт JSR-250. В итоге вы должны позволить требованиям вашего приложения управлять решением относительно используемого подхода. Если проблемой является переносимость, применяйте подход с инициализацией или аннотациями; в противном случае используйте интерфейс `InitializingBean`, чтобы сократить объем потребностей в конфигурировании для приложения и снизить вероятность возникновения ошибок в приложении из-за некорректной конфигурации.

### Порядок распознавания

К одному и тому же экземпляру бина можно применять все механизмы. В этом случае Spring вызывает метод, аннотированный `@PostConstruct`, затем метод `InitializingBean.afterPropertiesSet()` и, наконец, ваш метод инициализации, указанный в конфигурационном файле. Это полезно в ситуации, когда существующий бин выполняет некоторую инициализацию в определенном методе, но необходимо предусмотреть дополнительную инициализацию, если используется Spring.

## Привязка к уничтожению бина

Когда применяется реализация `ApplicationContext`, являющаяся оболочкой интерфейса `DefaultListableBeanFactory` (такая как `GenericXmlApplicationContext` через метод `getDefaultListableBeanFactory()`), с помощью вызова `ConfigurableBeanFactory.destroySingletons()` объекту `BeanFactory` можно сообщить о необходимости уничтожения всех одиночных экземпляров. Обычно это делается при прекращении работы приложения и позволяет очистить любые ресурсы, которые бины могут удерживать открытыми, аккуратно завершая приложение. Такой обратный вызов является удобным местом для сброса данных, находящихся в памяти, в постоянное хранилище, а также для завершения длительно выполняющихся процессов, которые могли быть запущены ранее.

Разрешить бинам получать уведомления о факте вызова метода `destroySingletons()` можно тремя способами, которые похожи на механизмы, предназначенные для получения обратного вызова инициализации. Обратный вызов уничтожения часто используется в сочетании с обратным вызовом инициализации. Во многих случаях в обратном вызове инициализации создается и конфигурируется некоторый ресурс, а в обратном вызове уничтожения этот ресурс освобождается.

### *Запуск метода, когда бин уничтожается*

Чтобы назначить метод для вызова во время уничтожения бина, нужно просто указать имя этого метода в атрибуте `destroy-method` дескриптора `<bean>` для бина. Платформа Spring вызывает этот метод непосредственно перед уничтожением одиночного экземпляра бина (она не будет вызывать этот метод для бинов с областью действия на уровне прототипа). В листинге 4.6 приведен пример применения обратного вызова `destroy-method`.

**Листинг 4.6. Использование обратного вызова `destroy-method`**

```
package com.apress.prospring4.ch4;

import java.io.IOException;
import java.io.File;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.context.support.GenericXmlApplicationContext;

public class DestructiveBean implements InitializingBean {
   private File file;
   private String filePath;

   @Override
   public void afterPropertiesSet() throws Exception {
      System.out.println("Initializing Bean"); // инициализация бина
      if (filePath == null) {
         throw new IllegalArgumentException(
               "You must specify the filePath property of "
              // Свойство filePath должно быть установлено в классе
               + DestructiveBean.class);
      }
      this.file = new File(filePath);
      this.file.createNewFile();
      System.out.println("File exists: " + file.exists());
                     // файл существует
   }

   public void destroy() {
      System.out.println("Destroying Bean"); // уничтожение бина
      if(!file.delete()) {
         // ОШИБКА: не удается удалить файл
         System.err.println("ERROR: failed to delete file.");
      }
      System.out.println("File exists: " + file.exists());
                     // файл существует
   }
```

```
    public void setFilePath(String filePath) {
        this.filePath = filePath;
    }

    public static void main(String[] args) throws Exception {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        DestructiveBean bean = (DestructiveBean) ctx.getBean("destructiveBean");

        System.out.println("Calling destroy()"); // вызов метода destroy()
        ctx.destroy();
        System.out.println("Called destroy()");  // метод destroy() вызван
    }
}
```

В этом коде определен метод `destroy()`, в котором удаляется ранее созданный файл. Метод `main()` извлекает бин типа `DestructiveBean` из `GenericXmlApplicationContext` и затем вызывает его метод `destroy()` (который, в свою очередь, вызывает `ConfigurableBeanFactory.destroySingletons()`, находящийся в оболочке `ApplicationContext`), сообщая платформе Spring о том, что нужно уничтожить все управляемые ею одиночные экземпляры. Обратные вызовы инициализации и уничтожения выводят на консоль сообщение, информирующее о том, что они были вызваны. В листинге 4.7 показана конфигурация для бина `destructiveBean` (`app-context-xml.xml`).

**Листинг 4.7. Конфигурирование обратного вызова `destroy-method`**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="destructiveBean"
        class="com.apress.prospring4.ch4.DestructiveBean"
        destroy-method="destroy"
        p:filePath=
        "#{systemProperties['java.io.tmpdir']}#{systemProperties[
'file.separator']}test.txt"/>
</beans>
```

Обратите внимание, что мы указали метод `destroy()` в качестве обратного вызова уничтожения с помощью атрибута `destroy-method`. Значение для атрибута `filePath` строится с использованием выражения SpEL, в котором перед именем файла `test.txt` помещен результат конкатенации системных свойств `java.io.tmpdir` и `file.separator`, чтобы обеспечить совместимость между платформами. В результате выполнения этого примера получается следующий вывод:

```
Initializing Bean
File exists: true
Calling destroy()
Destroying Bean
File exists: false
Called destroy()
```

Как видите, Spring сначала запускает обратный вызов инициализации, после чего экземпляр `DestructiveBean` создает экземпляр `File` и сохраняет его. Затем во время вызова `destroy()` платформа Spring проходит по набору управляемых ею одиночных экземпляров (в этом случае есть только один такой экземпляр) и инициирует любые обратные вызовы уничтожения, которые были определены. Именно здесь экземпляр `DestructiveBean` удаляет созданный файл и выводит на экран сообщение о том, что он больше не существует.

### *Реализация интерфейса `DisposableBean`*

Как и для обратных вызовов инициализации, Spring предоставляет интерфейс, в данном случае `DisposableBean`, который бины могут реализовать в качестве механизма для получения обратных вызовов уничтожения. В интерфейсе `DisposableBean` определен единственный метод `destroy()`, который вызывается непосредственно перед уничтожением бина. Этот интерфейс применяется подобно интерфейсу `InitializingBean`. В листинге 4.8 приведен модифицированный код класса `DestructiveBean`, реализующего интерфейс `DisposableBean`.

**Листинг 4.8. Реализация интерфейса `DisposableBean`**

```
package com.apress.prospring4.ch4;

import java.io.IOException;
import java.io.File;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.context.support.GenericXmlApplicationContext;

public class DestructiveBeanWithInterface implements InitializingBean,
DisposableBean {
    private File file;
    private String filePath;

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("Initializing Bean"); // инициализация бина

        if (filePath == null) {
            throw new IllegalArgumentException(
                    "You must specify the filePath property of "
                   // Свойство filePath должно быть установлено в классе
                    + DestructiveBeanWithInterface.class);
        }
        this.file = new File(filePath);
        this.file.createNewFile();

        System.out.println("File exists: " + file.exists());
                           // файл существует
    }
```

```
    @Override
    public void destroy() {
        System.out.println("Destroying Bean"); // уничтожение бина
        if(!file.delete()) {
            // ОШИБКА: не удается удалить файл
            System.err.println("ERROR: failed to delete file.");
        }
        System.out.println("File exists: " + file.exists());
                        // файл существует
    }
    public void setFilePath(String filePath) {
        this.filePath = filePath;
    }
    public static void main(String[] args) throws Exception {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-xml.xml");
      ctx.refresh();
      DestructiveBeanWithInterface bean =
          (DestructiveBeanWithInterface) ctx.getBean("destructiveBean");
      System.out.println("Calling destroy()"); // вызов метода destroy()
      ctx.destroy();
      System.out.println("Called destroy()");  // метод destroy() вызван
    }
}
```

Разница между кодом, использующим метод обратного вызова, и кодом, в котором применяется интерфейс обратного вызова, незначительна. В этом случае мы даже использовали одинаковые имена методов. В листинге 4.9 показана уточненная конфигурация для данного примера (`app-context-xml.xml`).

**Листинг 4.9. Конфигурация для примера применения интерфейса `DisposableBean`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="destructiveBean"
        class="com.apress.prospring4.ch4.DestructiveBeanWithInterface"
        p:filePath=
        "#{systemProperties['java.io.tmpdir']}#{systemProperties[
'file.separator']}test.txt"/>
</beans>
```

Кроме отличающегося имени класса единственное отличие связано с отсутствием атрибута `destroy-method`.

Запуск этого примера дает следующий вывод:

```
Initializing Bean
File exists: true
Calling destroy()
Destroying Bean
File exists: false
Called destroy()
```

### *Использование аннотации `@PreDestroy` стандарта JSR-250*

Третий способ предусматривает применение аннотации жизненного цикла `@PreDestroy`, определенной в стандарте JSR-250, которая является противоположностью `@PostConstruct`. В листинге 4.10 представлена версия класса `DestructiveBean`, в которой используются аннотации `@PostConstruct` и `@PreDestroy` для выполнения действий, связанных с инициализацией и уничтожением.

**Листинг 4.10. Реализация `DisposableBean` с использованием `@PreDestroy` и `@PostDestroy`**

```
package com.apress.prospring4.ch4;

import java.io.File;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.context.support.GenericXmlApplicationContext;

public class DestructiveBeanWithJSR250 {
    private File file;
    private String filePath;
    @PostConstruct
    public void afterPropertiesSet() throws Exception {
        System.out.println("Initializing Bean");

        if (filePath == null) {
            throw new IllegalArgumentException(
                    "You must specify the filePath property of " +
                    DestructiveBeanWithJSR250.class);
        }

        this.file = new File(filePath);
        this.file.createNewFile();

        System.out.println("File exists: " + file.exists());
    }

    @PreDestroy
    public void destroy() {
        System.out.println("Destroying Bean");

        if(!file.delete()) {
            System.err.println("ERROR: failed to delete file.");
        }

        System.out.println("File exists: " + file.exists());
    }

    public void setFilePath(String filePath) {
        this.filePath = filePath;
    }
```

```
    public static void main(String[] args) throws Exception {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();

        DestructiveBeanWithJSR250 bean =
            (DestructiveBeanWithJSR250) ctx.getBean("destructiveBean");

        System.out.println("Calling destroy()");
        ctx.destroy();
        System.out.println("Called destroy()");
    }
}
```

В листинге 4.11 показана XML-конфигурация для этого бина, в которую добавлен дескриптор `<context:annotation-config>` (`app-context-annotation.xml`).

**Листинг 4.11. Конфигурация для примера использования `DisposableBean` с аннотацией JSR-250**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>
    <bean id="destructiveBean"
        class="com.apress.prospring4.ch4.DestructiveBeanWithJSR250"
        p:filePath=
        "#{systemProperties['java.io.tmpdir']}#{systemProperties[
'file.separator']}test.txt"/>
</beans>
```

Обратный вызов при уничтожении — это идеальный механизм для обеспечения аккуратного завершения приложений, при котором ресурсы гарантированно не останутся в открытом или несогласованном состоянии. Однако по-прежнему приходится решать, какой подход использовать: метод обратного вызова при уничтожении, интерфейс `DisposableBean` или аннотацию `@PreDestroy`. Как и ранее, позвольте требованиям приложения управлять решением этого вопроса; применяйте метод обратного вызова, если важна переносимость, и интерфейс `DisposableBean` или аннотацию JSR-250, чтобы снизить объем необходимой конфигурации.

### *Порядок распознавания*

Как и в случае с созданием бина, при уничтожении бина можно использовать все механизмы на одном и том же экземпляре бина. В такой ситуации Spring сначала вызывает метод, аннотированный `@PreDestroy`, затем метод `DisposableBean.destroy()` и, наконец, ваш метод уничтожения, сконфигурированный в определении XML.

### *Использование перехватчика завершения*

Единственный недостаток обратных вызовов уничтожения в Spring связан с тем, что они не запускаются автоматически, т.е. перед закрытием приложения нужно не забыть вызвать `AbstractApplicationContext.destroy()`. Когда приложение выполняется как сервлет, указанный метод `destroy()` можно вызвать в методе `destroy()` сервлета. Однако в автономном приложении все не так просто, особенно при наличии множества точек выхода из приложения. К счастью, решение есть. Java позволяет создать перехватчик завершения (shutdown hook) — поток, который выполняется непосредственно перед завершением приложения. Это отличный способ вызвать метод `destroy()` вашего класса `AbstractApplicationContext` (который расширяется всеми конкретными реализациями `ApplicationContext`). Задействовать данный механизм проще всего с применением метода `registerShutdownHook()` класса `AbstractApplicationContext`. Этот метод автоматически инструктирует Spring о необходимости регистрации перехватчика завершения лежащей в основе исполняющей среды JVM. В листинге 4.12 приведен пример.

**Листинг 4.12. Регистрация перехватчика завершения**

```
package com.apress.prospring4.ch4;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import java.io.File;
import org.springframework.context.support.GenericXmlApplicationContext;

public class DestructiveBeanWithInterface {
   private File file;
   private String filePath;

   @PostConstruct
   public void afterPropertiesSet() throws Exception {
      System.out.println("Initializing Bean");

      if (filePath == null) {
         throw new IllegalArgumentException(
               "You must specify the filePath property of " +
               DestructiveBeanWithInterface.class);
      }

      this.file = new File(filePath);
      this.file.createNewFile();

      System.out.println("File exists: " + file.exists());
   }

   @PreDestroy
   public void destroy() {
      System.out.println("Destroying Bean");

      if(!file.delete()) {
         System.err.println("ERROR: failed to delete file.");
      }

      System.out.println("File exists: " + file.exists());
   }
```

```
    public void setFilePath(String filePath) {
        this.filePath = filePath;
    }

    public static void main(String[] args) throws Exception {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.registerShutdownHook();
        ctx.refresh();

        DestructiveBeanWithInterface bean =
            (DestructiveBeanWithInterface) ctx.getBean("destructiveBean");
    }
}
```

Ниже показана соответствующая XML-конфигурация (`app-context-annotation.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

    <bean id="destructiveBean"
        class="com.apress.prospring4.ch4.DestructiveBeanWithInterface"
        p:filePath=
        "#{systemProperties['java.io.tmpdir']}
         #{systemProperties['file.separator']}test.txt"/>
</beans>
```

Выполнение этого примера дает в результате следующий вывод:

```
Initializing Bean
File exists: true
Destroying Bean
File exists: false
```

Как видите, метод `destroy()` был вызван, несмотря на то, что мы не написали ни единой строки кода для его явного вызова при завершении приложения.

# Превращение бинов в "осведомленные о платформе Spring"

Одно из крупнейших преимуществ внедрения зависимостей перед поиском зависимостей как механизма инверсии управления заключается в том, что бины не обязаны знать детали реализации контейнера, который ими управляет. Для бина, который использует внедрение зависимостей через конструктор или метод установки, контейнер Spring ничем не отличается от контейнера, предоставляемого Google

Guice, или, скажем, контейнера PicoContainer. Однако при определенных обстоятельствах может потребоваться бин, который применяет внедрение зависимостей для получения своих зависимостей, так что он может взаимодействовать с контейнером по какой-то другой причине. Примером может служить бин, который автоматически конфигурирует перехватчик завершения, поэтому ему необходим доступ к `ApplicationContext`. В других случаях бину может понадобиться узнать свое имя (т.е. имя бина, назначенное внутри текущего `ApplicationContext`), чтобы на основе этого предпринимать то или иное действие.

Вообще говоря, это средство предназначено для внутреннего употребления Spring. Предоставление имени бина некоторого бизнес-смысла в общем случае является неудачной идеей и может привести к проблемам конфигурирования, когда именами бинов придется искусственно манипулировать с целью поддержки их бизнес-смысла. Тем не менее, мы считаем, что возможность бина выяснить свое имя во время выполнения действительно полезна для целей регистрации в журналах. Представьте ситуацию, в которой есть множество бинов одного и того же типа, выполняющихся с различными конфигурациями. Имя бина может быть включено в журнальные сообщения, что поможет отличать бины, генерирующие ошибки, от бинов, которые функционировали нормально, когда что-то пошло не так, как было задумано.

## Использование интерфейса `BeanNameAware`

Интерфейс `BeanNameAware`, который бин может реализовать, чтобы получить свое имя, имеет единственный метод: `setBeanName(String)`. Платформа Spring вызывает метод `setBeanName()` после завершения конфигурирования бина, но перед любыми обратными вызовами жизненного цикла (инициализации или уничтожения), как было показано на рис. 4.1. В большинстве случаев реализация метода `setBeanName()` сводится к одной строке, в которой значение, переданное контейнером, сохраняется в каком-то поле для дальнейшего использования. В листинге 4.13 приведен код простого бина, который получает свое имя с помощью `BeanNameAware` и затем выводит его на консоль.

**Листинг 4.13. Реализация `BeanNameAware`**

```
package com.apress.prospring4.ch4;

import org.springframework.beans.factory.BeanNameAware;

public class BeanNamePrinter implements BeanNameAware {
    private String beanName;

    @Override
    public void setBeanName(String beanName) {
        this.beanName = beanName;
    }

    public void someOperation() {
        System.out.println("Bean [" + beanName + "] - someOperation()");
    }
}
```

Представленная реализация довольно проста. Вспомните, что метод `BeanNameAware.setBeanName()` вызывается перед тем, как первый экземпляр бина будет возвращен приложению через вызов `ApplicationContext.getBean()`, поэтому внутри метода `someOperation()` нет необходимости проводить проверку, доступно ли имя бина. В листинге 4.14 показана простая конфигурация для этого примера (`app-context-xml.xml`).

**Листинг 4.14. Конфигурация для примера `LoggingBean`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="beanNamePrinter" class="com.apress.prospring4.ch4.BeanNamePrinter"/>
</beans>
```

Легко заметить, что никакой специальной конфигурации для использования интерфейса `BeanNameAware` не требуется. В листинге 4.15 приведен код простого приложения, которое извлекает экземпляр `BeanNamePrinter` из `ApplicationContext` и затем вызывает метод `someOperation()`.

**Листинг 4.15. Класс `LoggingBeanExample`**

```
package com.apress.prospring4.ch4;

import org.springframework.context.support.GenericXmlApplicationContext;

public class BeanNamePrinterExample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-xml.xml");
      ctx.refresh();

      BeanNamePrinter bean = (BeanNamePrinter) ctx.getBean("beanNamePrinter");
      bean.someOperation();
   }
}
```

Этот пример генерирует следующий вывод; обратите внимание на наличие имени бина в сообщении о вызове `someOperation()`:

```
Loading XML bean definitions from class path resource
  [META-INF/spring/app-context-xml.xml]
...
Bean [beanNamePrinter] - someOperation()
```

Работать с интерфейсом `BeanNameAware` действительно просто, и он очень полезен, когда нужно улучшить качество журнальных сообщений. Не поддавайтесь искушению придавать именам бинов бизнес-смысл просто потому, что к ним возможен доступ; поступая так, вы привязываете классы к Spring, но взамен получаете функциональность, которая приносит незначительную пользу. Если бины внутренне

нуждаются в имени, обеспечьте реализацию ими интерфейса наподобие Nameable (специфичный для приложения), имеющего метод setName(), и затем назначьте каждому бину имя с применением внедрения зависимостей. В этом случае можно сохранить краткими имена, которые используются при конфигурировании, и вынуждено не манипулировать конфигурацией, чтобы привнести в имена бинов бизнес-смысл.

## Использование интерфейса ApplicationContextAware

За счет применения интерфейса ApplicationContextAware бины имеют возможность получать ссылку на контекст ApplicationContext, который их сконфигурировал. Основной причиной, по которой созданы эти интерфейсы, была необходимость предоставить бину доступ к ApplicationContext внутри приложения, например, для получения других бинов Spring программным образом, используя getBean(). Однако вы должны избегать подобной практики и применять внедрение зависимостей для предоставления своих бинов, а также их коллабораторов. Если вы используете основанный на поиске подход getBean() для получения зависимостей, когда можно применять внедрение зависимостей, то тем самым привносите ненужную сложность в свои бины и привязываете их к Spring Framework без веских причин.

Конечно, контекст ApplicationContext используется не только для поиска бинов; он решает множество других задач. Как упоминалось ранее, одной из задач является уничтожение всех одиночных экземпляров с предварительным уведомлением их по очереди. В предыдущем разделе было показано, как создать перехватчик завершения, чтобы сообщить ApplicationContext о необходимости уничтожения всех одиночных экземпляров перед завершением приложения. Применяя интерфейс ApplicationContextAware, несложно построить бин, который может быть сконфигурирован в ApplicationContext на автоматическое создание и настройку перехватчика завершения. Код этого бина приведен в листинге 4.16.

**Листинг 4.16. Класс ShutdownHookBean**

```
package com.apress.prospring4.ch4;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.support.GenericApplicationContext;

public class ShutdownHookBean implements ApplicationContextAware {
    private ApplicationContext ctx;

    @Override
    public void setApplicationContext(ApplicationContext ctx)
        throws BeansException {

        if (ctx instanceof GenericApplicationContext) {
            ((GenericApplicationContext) ctx).registerShutdownHook();
        }
    }
}
```

Большая часть кода должна выглядеть хорошо знакомой. В интерфейсе `ApplicationContextAware` определен единственный метод `setApplicationContext(ApplicationContext)`, который Spring вызывает для передачи вашему бину ссылки на его контекст `ApplicationContext`. В листинге 4.16 класс `ShutdownHookBean` проверяет, относится ли `ApplicationContext` к типу `GenericApplicationContext`, что означает его поддержку метода `registerShutdownHook()`; если это так, перехватчик завершения для `ApplicationContext` будет зарегистрирован. В листинге 4.17 показано, как сконфигурировать этот бин для работы с бином `DestructiveBeanWithInterface` (`app-context-annotation.xml`).

**Листинг 4.17. Конфигурирование `ShutdownHookBean`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

   <context:annotation-config/>

   <bean id="destructiveBean"
      class="com.apress.prospring4.ch4.DestructiveBeanWithInterface"
      p:filePath=
      "#{systemProperties['java.io.tmpdir']}
       #{systemProperties['file.separator']}test.txt"/>

   <bean id="shutdownHook"
      class="com.apress.prospring4.ch4.ShutdownHookBean"/>
</beans>
```

Обратите внимание, что никакой специальной конфигурации не требуется. В листинге 4.18 приведен простой пример приложения, в котором `ShutdownHookBean` используется для управления уничтожением одиночных экземпляров бинов.

**Листинг 4.18. Использование `ShutdownHookBean`**

```
package com.apress.prospring4.ch4;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import java.io.File;
import org.springframework.context.support.GenericXmlApplicationContext;

public class DestructiveBeanWithInterface {
   private File file;
   private String filePath;

   @PostConstruct
   public void afterPropertiesSet() throws Exception {
      System.out.println("Initializing Bean");
```

```
        if (filePath == null) {
            throw new IllegalArgumentException(
                    "You must specify the filePath property of " +
                    DestructiveBeanWithInterface.class);
        }
        this.file = new File(filePath);
        this.file.createNewFile();

        System.out.println("File exists: " + file.exists());
    }

    @PreDestroy
    public void destroy() {
        System.out.println("Destroying Bean");

        if(!file.delete()) {
            System.err.println("ERROR: failed to delete file.");
        }

        System.out.println("File exists: " + file.exists());
    }

    public void setFilePath(String filePath) {
        this.filePath = filePath;
    }

    public static void main(String[] args) throws Exception {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.registerShutdownHook();
      ctx.refresh();

      DestructiveBeanWithInterface bean =
          (DestructiveBeanWithInterface) ctx.getBean("destructiveBean");
    }
}
```

Показанный код также должен выглядеть знакомым. Когда платформа Spring производит начальную загрузку `ApplicationContext`, и бин `destructiveBean` определен в конфигурации, она передает ссылку на `ApplicationContext` бину `shutdownHook` для регистрации перехватчика завершения. Выполнение этого примера, как и ожидалось, дает следующий вывод:

```
Initializing Bean
File exists: true
Destroying Bean
File exists: false
```

Как видите, даже при отсутствии обращений к `destroy()` в главном приложении бин `ShutdownHookBean` зарегистрирован как перехватчик завершения, и он вызывает метод `destroy()` непосредственно перед завершением приложения.

# Использование фабрик бинов

При работе с платформой Spring вы столкнетесь с проблемой создания и внедрения зависимостей, которые нельзя создать просто за счет использования операции `new`. Для решения этой проблемы Spring предоставляет интерфейс фабрики бинов

FactoryBean, действующий в качестве адаптера для объектов, которые не могут создаваться и управляться с помощью стандартной семантики Spring. Обычно фабрики бинов применяются для создания бинов, которые невозможно создать с использованием операции new, таких как бины, доступные посредством статических фабричных методов, хотя это не всегда так. Попросту говоря, FactoryBean — это бин, который действует как фабрика для других бинов. Фабрики бинов конфигурируются внутри ApplicationContext подобно обычным бинам, но когда платформа Spring применяет интерфейс FactoryBean для удовлетворения запроса зависимости или поиска, она не возвращает экземпляр FactoryBean, а взамен вызывает метод FactoryBean.getObject() и возвращает результат этого вызова.

Фабрики бинов используются в Spring для решения важных задач, наиболее заметными из которых является создание прокси для транзакций, как будет показано в главе 9, и автоматическое извлечение ресурсов из контекста JNDI. Тем не менее, фабрики бинов полезны не только для реализации внутренней функциональности Spring; вы сочтете их по-настоящему удобными и при построении собственных приложений, т.к. они позволяют управлять намного большим числом ресурсов с применением IoC, чем это было бы в противном случае.

## Пример специальной фабрики бинов: класс MessageDigestFactoryBean

Часто разрабатываемые проекты требуют той или иной разновидности криптографической обработки; как правило, это связано с генерацией дайджеста сообщения или хеша пользовательского пароля для его сохранения в базе данных. В Java имеется класс MessageDigest, который предлагает функциональность для создания дайджеста из произвольных данных. Класс MessageDigest сам по себе является абстрактным; его конкретные реализации получаются вызовом метода MessageDigest.getInstance() с передачей ему имени алгоритма построения дайджеста, который должен использоваться. Например, если для создания дайджеста необходимо применять алгоритм MD5, то получить экземпляр MessageDigest можно с помощью следующего кода:

```
MessageDigest md5 = MessageDigest.getInstance("MD5");
```

Если для управления созданием объекта MessageDigest должна использоваться платформа Spring, то лучшее, что можно сделать без FactoryBean — предусмотреть в бине определенное свойство, algorithmName, и затем применять обратный вызов инициализации для обращения к MessageDigest.getInstance(). Используя FactoryBean, эту логику можно инкапсулировать внутри бина. После этого любые бины, которым требуется экземпляр MessageDigest, могут просто объявлять свойство, скажем, messageDigest, и с помощью FactoryBean получать нужный экземпляр. В листинге 4.19 показана реализация FactoryBean, которая делает как раз то, что описано выше.

**Листинг 4.19. Класс MessageDigestFactoryBean**

```
package com.apress.prospring4.ch4;
import java.security.MessageDigest;
import org.springframework.beans.factory.FactoryBean;
import org.springframework.beans.factory.InitializingBean;
```

```
public class MessageDigestFactoryBean implements
     FactoryBean<MessageDigest>, InitializingBean {
   private String algorithmName = "MD5";

   private MessageDigest messageDigest = null;

   @Override
   public MessageDigest getObject() throws Exception {
      return messageDigest;
   }

   @Override
   public Class<MessageDigest> getObjectType() {
      return MessageDigest.class;
   }

   @Override
   public boolean isSingleton() {
      return true;
   }

   @Override
   public void afterPropertiesSet() throws Exception {
      messageDigest = MessageDigest.getInstance(algorithmName);
   }

   public void setAlgorithmName(String algorithmName) {
      this.algorithmName = algorithmName;
   }
}
```

В интерфейсе `FactoryBean` объявлены три метода: `getObject()`, `getObjectType()` и `isSingleton()`. Платформа Spring вызывает метод `getObject()` для извлечения объекта, созданного `FactoryBean`. Это действительный объект, который передается другим бинам, применяющим `FactoryBean` в качестве коллаборатора. В листинге 4.19 видно, что `MessageDigestFactoryBean` передает клон сохраненного экземпляра `MessageDigest`, который создан в обратном вызове `InitializingBean.afterPropertiesSet()`.

Метод `getObjectType()` позволяет сообщить Spring, какой тип объекта будет возвращать фабрика бинов. В качестве типа можно указать `null`, если тип заранее не известен (например, фабрика бинов создает объекты разных типов в зависимости от конфигурации, что может быть определено только после инициализации фабрики), но если тип задан, то Spring может использовать его для автосвязывания. Возвращаемым типом является `MessageDigest` (в данной ситуации это класс, но можно попробовать вернуть тип интерфейса и заставить фабрику бинов создавать экземпляр конкретного класса реализации, если только это необходимо). Причина в том, что нам не известно, какой конкретный тип будет возвращен (это не играет роли, поскольку в любом случае все бины будут определять свои зависимости с применением `MessageDigest`).

Свойство `isSingleton()` позволяет информировать Spring о том, что фабрика бинов управляет одиночным экземпляром. Не забывайте, что за счет установки атрибута `singleton` дескриптора `<bean>` для фабрики бинов вы сообщаете Spring о поддержке одиночных экземпляров самой фабрикой, но не возвращаемыми ею объектами.

А теперь давайте посмотрим, как задействовать фабрику бинов в приложении. В листинге 4.20 приведен код простого бина, который обслуживает два экземпляра `MessageDigest` и отображает дайджесты сообщения, передаваемого методу `digest()`.

**Листинг 4.20. Класс `MessageDigester`**

```
package com.apress.prospring4.ch4;

import java.security.MessageDigest;

public class MessageDigester {
    private MessageDigest digest1;
    private MessageDigest digest2;

    public void setDigest1(MessageDigest digest1) {
        this.digest1 = digest1;
    }

    public void setDigest2(MessageDigest digest2) {
        this.digest2 = digest2;
    }

    public void digest(String msg) {
        System.out.println("Using digest1");
        digest(msg, digest1);

        System.out.println("Using digest2");
        digest(msg, digest2);
    }

    private void digest(String msg, MessageDigest digest) {
        System.out.println("Using alogrithm: " + digest.getAlgorithm());
        digest.reset();
        byte[] bytes = msg.getBytes();
        byte[] out = digest.digest(bytes);
        System.out.println(out);
    }
}
```

В листинге 4.21 показан пример конфигурации для двух бинов `MessageDigestFactoryBean`, из которых один использует алгоритм SHA1, а другой — стандартный (MD5) алгоритм (`app-context-xml.xml`).

**Листинг 4.21. Конфигурирование бинов `MessageDigestFactoryBean`**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="shaDigest" class="com.apress.prospring4.ch4.MessageDigestFactoryBean"
     p:algorithmName="SHA1"/>

   <bean id="defaultDigest"
        class="com.apress.prospring4.ch4.MessageDigestFactoryBean"/>
```

```
    <bean id="digester"
        class="com.apress.prospring4.ch4.MessageDigester"
        p:digest1-ref="shaDigest"
        p:digest2-ref="defaultDigest"/>
</beans>
```

Как видите, мы сконфигурированы не только два бина `MessageDigestFactoryBean`, но также и `MessageDigester`, при этом бины `MessageDigestFactoryBean` предоставляют значения свойствам `digest1` и `digest2`. Поскольку в бине `defaultDigest` свойство `algorithmName` не указано, внедрение не произойдет, и будет применяться стандартный алгоритм (MD5), закодированный в классе. В листинге 4.22 приведен базовый пример класса, который извлекает бин `MessageDigester` из фабрики бинов и создает дайджест простого сообщения.

**Листинг 4.22. Использование `MessageDigester`**

```
package com.apress.prospring4.ch4;

import org.springframework.context.support.GenericXmlApplicationContext;

public class MessageDigestExample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        MessageDigester digester = (MessageDigester) ctx.getBean("digester");
        digester.digest("Hello World!");
    }
}
```

Запуск этого примера дает в результате следующий вывод:

```
Using digest1
Using alogrithm: SHA1
[B@77cd7a0
Using digest2
Using alogrithm: MD5
[B@204f30ec
```

Здесь видно, что бин `MessageDigest` снабжен двумя реализациями `MessageDigest`, SHA1 и MD5, несмотря на то, что бины `MessageDigest` не были сконфигурированы в фабрике бинов. Именно так функционирует фабрика бинов.

Фабрики бинов не являются идеальным решением, когда вы работаете с классами, экземпляры которых не могут быть созданы с использованием операции `new`. Если вы работаете с объектами, созданными с помощью фабричного метода, и хотите применять эти классы в приложении Spring, создайте фабрику бинов для действия в качестве адаптера, позволив классам пользоваться всеми преимуществами встроенных в Spring средств IoC.

## Доступ к фабрике бинов напрямую

Учитывая тот факт, что Spring автоматически реагирует на любые ссылки на фабрику бинов выдачей объектов, созданных этой фабрикой, возникает вопрос: можно ли получить доступ к фабрике бинов напрямую? Ответ: да, можно.

Доступ к фабрике бинов осуществляется просто: в вызове метода `getBean()` необходимо предварить имя бина амперсандом, как показано в листинге 4.23.

**Листинг 4.23. Доступ к фабрикам бинов напрямую**

```
package com.apress.prospring4.ch4;

import java.security.MessageDigest;

import org.springframework.context.support.GenericXmlApplicationContext;

public class AccessingFactoryBeans {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-xml.xml");
      ctx.refresh();

      MessageDigest digest = (MessageDigest) ctx.getBean("shaDigest");

      MessageDigestFactoryBean factoryBean =
          (MessageDigestFactoryBean) ctx.getBean("&shaDigest");

      try {
          MessageDigest shaDigest = factoryBean.getObject();
          System.out.println(shaDigest.digest("Hello world".getBytes()));
      } catch (Exception ex) {
          ex.printStackTrace();
      }
   }
}
```

Выполнение этой программы генерирует такой вывод:

```
[B@77cd7a0
```

Это средство используется в нескольких местах кода Spring, но для его применения в приложении не должно возникать каких-либо причин. Интерфейс `FactoryBean` предназначен для использования в качестве части поддерживающей инфраструктуры, чтобы дать возможность применять большее число классов приложения в настройке IoC. Избегайте прямого доступа к `FactoryBean` и затем ручного вызова метода `getObject()`, позволив это делать платформе Spring; выполняя это вручную, вы делаете излишнюю работу и неизбежно привязываете свое приложение к специфическим деталям реализации, которые в будущем могут измениться.

## Использование атрибутов `factory-bean` и `factory-method`

Иногда нужно создавать экземпляры компонентов JavaBean, которые были предоставлены приложением третьей стороны, не поддерживающим Spring. Вы не знаете, как создавать экземпляр этого класса, но вам известно, что приложение третьей стороны предлагает класс, который можно использовать для получения экземпляра JavaBean, необходимого вашему приложению Spring. В этом случае также могут

применяться атрибуты factory-bean и factory-method дескриптора <bean> для бина Spring.

Чтобы продемонстрировать, как это работает, в листинге 4.24 приведена еще одна версия класса MessageDigestFactory, которая предоставляет метод для возвращения бина MessageDigest.

**Листинг 4.24. Класс MessageDigestFactory**

```
package com.apress.prospring4.ch4;

import java.security.MessageDigest;

public class MessageDigestFactory {
   private String algorithmName = "MD5";

   public MessageDigest createInstance() throws Exception {
      return MessageDigest.getInstance(algorithmName);
   }

   public void setAlgorithmName(String algorithmName) {
      this.algorithmName = algorithmName;
   }
}
```

В листинге 4.25 показано, каким образом сконфигурировать фабричный метод для получения соответствующего экземпляра бина MessageDigest (app-context-xml.xml).

**Листинг 4.25. Конфигурирование MessageDigestFactory**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="shaDigestFactory"
      class="com.apress.prospring4.ch4.MessageDigestFactory"
      p:algorithmName="SHA1"/>

   <bean id="defaultDigestFactory"
      class="com.apress.prospring4.ch4.MessageDigestFactory"/>

   <bean id="shaDigest"
        factory-bean="shaDigestFactory"
        factory-method="createInstance">
   </bean>

   <bean id="defaultDigest"
        factory-bean="defaultDigestFactory"
        factory-method="createInstance"/>

   <bean id="digester"
      class="com.apress.prospring4.ch4.MessageDigester"
      p:digest1-ref="shaDigest"
      p:digest2-ref="defaultDigest"/>
</beans>
```

Обратите внимание, что в конфигурации определены два бина `MessageDigestFactory`, из которых один использует SHA1, а другой — стандартный алгоритм. Затем для бинов `shaDigest` и `defaultDigest` с помощью атрибута `factory-bean` мы инструктируем Spring о необходимости создавать их экземпляры с применением соответствующего бина `MessageDigestFactory` и указываем в атрибуте `factory-method` метод, который должен применяться для получения экземпляра бина.

В листинге 4.26 приведен код класса, предназначенного для тестирования.

**Листинг 4.26. Класс `MessageDigestFactory` в действии**

```
package com.apress.prospring4.ch4;

import org.springframework.context.support.GenericXmlApplicationContext;

public class MessageDigestFactoryExample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-xml.xml");
      ctx.refresh();

      MessageDigester digester = (MessageDigester) ctx.getBean("digester");
      digester.digest("Hello World!");
   }
}
```

Запуск этой программы сгенерирует следующий вывод:

```
Using digest1
Using alogrithm: SHA1
[B@1e397ed7
Using digest2
Using alogrithm: MD5
[B@490ab905
```

# Редакторы свойств для компонентов JavaBean

Если вы не очень хорошо знакомы с концепциями компонентов JavaBean, то знайте, что `PropertyEditor` (редактор свойств) — это интерфейс, который преобразует значение свойства в и из представления внутреннего типа String. Первоначально это задумывалось как способ ввода значений свойств в виде строк (`String`) в редакторе с последующим их преобразованием в корректный тип. Однако, поскольку редакторы свойств по своей природе являются облегченными классами, они нашли применение во многих ситуациях, включая Spring.

Из-за того, что изрядная часть значений свойств в приложении, основанном на Spring, задается в файле конфигурации `BeanFactory`, по существу они являются строками. Тем не менее, свойства, для которых указываются эти значения, могут не относиться к типу `String`. Таким образом, платформа Spring позволяет вместо искусственного создания множества свойств типа `String` определить редакторы свойств, которые будут управлять преобразованием значений `String`, заданных для свойств, в подходящие типы.

## Встроенные редакторы свойств

В версии Spring 4 доступно 13 встроенных реализаций `PropertyEditor`, предварительно зарегистрированных в `BeanFactory`. В листинге 4.27 показан код простого бина, в котором объявлено 13 свойств, по одному для каждого типа, поддерживаемого встроенными редакторами свойств.

**Листинг 4.27. Использование встроенных редакторов свойств**

```
package com.apress.prospring4.ch4;

import java.io.File;
import java.io.InputStream;
import java.net.URL;
import java.util.Date;
import java.util.List;
import java.util.Locale;
import java.util.Properties;
import java.util.regex.Pattern;
import java.text.SimpleDateFormat;
import org.springframework.beans.PropertyEditorRegistrar;
import org.springframework.beans.PropertyEditorRegistry;
import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.beans.propertyeditors.StringTrimmerEditor;

import org.springframework.context.support.GenericXmlApplicationContext;
public class PropertyEditorBean {
    private byte[] bytes;                   // ByteArrayPropertyEditor
    private Class cls;                      // ClassEditor
    private Boolean trueOrFalse;            // CustomBooleanEditor
    private List<String> stringList;        // CustomCollectionEditor
    private Date date;                      // CustomDateEditor
    private Float floatValue;               // CustomNumberEditor
    private File file;                      // FileEditor
    private InputStream stream;             // InputStreamEditor
    private Locale locale;                  // LocaleEditor
    private Pattern pattern;                // PatternEditor
    private Properties properties;          // PropertiesEditor
    private String trimString;              // StringTrimmerEditor
    private URL url;                        // URLEditor
    public void setCls(Class cls) {
        // Установка класса
        System.out.println("Setting class: " + cls.getName());
        this.cls = cls;
    }
    public void setFile(File file) {
        // Установка файла
        System.out.println("Setting file: " + file.getName());
        this.file = file;
    }
    public void setLocale(Locale locale) {
        // Установка локали
        System.out.println("Setting locale: " + locale.getDisplayName());
        this.locale = locale;
    }
```

```
public void setProperties(Properties properties) {
    // Вывод количества загруженных свойств
    System.out.println("Loaded " + properties.size() + " properties");
    this.properties = properties;
}
public void setUrl(URL url) {
    // Установка URL
    System.out.println("Setting URL: " + url.toExternalForm());
    this.url = url;
}
public void setBytes(byte[] bytes) {
    // Вывод количества добавленных байтов
    System.out.println("Adding " + bytes.length + " bytes");
    this.bytes = bytes;
}
public void setTrueOrFalse(Boolean trueOrFalse) {
    // Установка булевского значения
    System.out.println("Setting Boolean: " + trueOrFalse);
    this.trueOrFalse = trueOrFalse;
}
public void setStringList(List<String> stringList) {
    // Установка списка строк
    System.out.println("Setting string list with size: "
       + stringList.size());
    this.stringList = stringList;
    for (String string: stringList) {
       // Член типа String
       System.out.println("String member: " + string);
    }
}
public void setDate(Date date) {
    // Установка даты
    System.out.println("Setting date: " + date);
    this.date = date;
}
public void setFloatValue(Float floatValue) {
    // Установка значения с плавающей точкой
    System.out.println("Setting float value: " + floatValue);
    this.floatValue = floatValue;
}
public void setStream(InputStream stream) {
    // Установка потока
    System.out.println("Setting stream: " + stream);
    this.stream = stream;
}
public void setPattern(Pattern pattern) {
    // Установка шаблона
    System.out.println("Setting pattern: " + pattern);
    this.pattern = pattern;
}
```

```
    public void setTrimString(String trimString) {
        // Установка усеченной строки
        System.out.println("Setting trim string: " + trimString);
        this.trimString = trimString;
    }

    public static class CustomPropertyEditorRegistrar
            implements PropertyEditorRegistrar {
        @Override
        public void registerCustomEditors(PropertyEditorRegistry registry) {
          SimpleDateFormat dateFormatter = new SimpleDateFormat("MM/dd/yyyy");
          registry.registerCustomEditor(Date.class,
                  new CustomDateEditor(dateFormatter, true));

          registry.registerCustomEditor(String.class,
                  new StringTrimmerEditor(true));
        }
    }

    public static void main(String[] args) throws Exception {
        File file = File.createTempFile("test", "txt");
        file.deleteOnExit();
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();
        PropertyEditorBean bean =
            (PropertyEditorBean) ctx.getBean("builtInSample");
    }
}
```

В листинге 4.27 видно, что `PropertyEditorBean` имеет 13 свойств, каждое из которых соответствует одному встроенному редактору свойств. В листинге 4.28 представлен пример конфигурации, задающей значения для всех этих свойств (`app-config-xml.xml`).

**Листинг 4.28. Конфигурация, использующая редакторы свойств**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util.xsd">

    <bean id="customEditorConfigurer"
      class="org.springframework.beans.factory.config.CustomEditorConfigurer"
      p:propertyEditorRegistrars-ref="propertyEditorRegistrarsList"/>

    <util:list id="propertyEditorRegistrarsList">
        <bean class="com.apress.prospring4.ch4.
PropertyEditorBean$CustomPropertyEditorRegistrar"/>
    </util:list>
```

```
  <bean id="builtInSample" class="com.apress.prospring4.ch4.PropertyEditorBean"
      p:bytes="Hello World"
      p:cls="java.lang.String"
      p:trueOrFalse="true"
      p:stringList-ref="stringList"
      p:stream="test.txt"
      p:floatValue="123.45678"
      p:date="05/03/13"
  p:file="#{systemProperties['java.io.tmpdir']}
          #{systemProperties['file.separator']}test.txt"
      p:locale="en_US"
      p:pattern="a*b"
      p:properties="name=Chris age=32"
      p:trimString="   String need trimming   "
      p:url="http://www.springframework.org"
   />

   <util:list id="stringList">
      <value>String member 1</value>
      <value>String member 2</value>
   </util:list>
</beans>
```

Как видите, хотя все свойства `PropertyEditorBean` не относятся к типу `String`, значения для свойств указаны как простые строки. Также обратите внимание на регистрацию `CustomDateEditor` и `StringTrimmerEditor`, поскольку эти два редактора по умолчанию в Spring не зарегистрированы. Выполнение данного примера дает следующий вывод:

```
Adding 11 bytes
Setting class: java.lang.String
Setting date: Wed May 03 00:00:00 EST 13
Setting file: test.txt
Setting float value: 123.45678
Setting locale: English (United States)
Setting pattern: a*b
Loaded 1 properties
Setting stream:
    sun.net.www.protocol.jar.JarURLConnection$JarURLInputStream@57e1b0c
Setting string list with size: 2
String member: String member 1
String member: String member 2
Setting trim string: String need trimming
Setting Boolean: true
Setting URL: http://www.springframework.org
```

Согласно этому выводу, Spring применяет встроенные редакторы свойств для преобразования строковых представлений различных свойств в корректные типы. В табл. 4.1 приведена сводка по встроенным редакторам свойств, которые доступны в Spring.

**Таблица 4.1. Встроенные редакторы свойств Spring**

| Редактор свойств | Описание |
|---|---|
| `ByteArrayPropertyEditor` | Преобразует значение `String` в массив байтов |
| `ClassEditor` | Преобразует полностью определенное имя класса в экземпляр `Class`. При использовании этого редактора свойств следите за тем, чтобы не включить лишние пробелы перед или после имени класса, когда применяется `GenericXmlApplicationContext`; в противном случае будет сгенерировано исключение `ClassNotFoundException` |
| `CustomBooleanEditor` | Преобразует строку в Java-тип `Boolean` |
| `CustomCollectionEditor` | Преобразует исходную коллекцию (например, представленную через пространство имен `util` в Spring) в целевой тип `Collection` |
| `CustomDateEditor` | Преобразует строковое представление даты в значение `java.util.Date`. Этот редактор с желаемым форматом даты необходимо зарегистрировать в `ApplicationContext` |
| `CustomNumberEditor` | Преобразует строку в числовое значение, которым может быть `Integer`, `Long`, `Float` или `Double` |
| `FileEditor` | Преобразует строковый путь к файлу в экземпляр `File`. Платформа Spring не проверяет существование файла |
| `InputStreamEditor` | Преобразует строковое представление ресурса (например, файлового ресурса вида `file:D:/temp/test.txt` или `classpath:test.txt`) в свойство входного потока |
| `LocaleEditor` | Преобразует строковое представление локали, такое как `en-GB`, в экземпляр `java.util.Locale` |
| `Pattern` | Преобразует строку в JDK-объект `Pattern` или наоборот |
| `PropertiesEditor` | Преобразует строку в формате `ключ1=значение1 ключ2=значение2 ключn=значениеn` в экземпляр `java.util.Properties` с настройкой соответствующих свойств |
| `StringTrimmerEditor` | Выполняет усечение строковых значений перед внедрением. Этот редактор свойств должен быть явно зарегистрирован |
| `URLEditor` | Преобразует строковое представление URL в экземпляр `java.net.URL` |

Этот набор редакторов свойств обеспечивает хорошую основу для работы с платформой Spring и существенно упрощает конфигурирование приложения с такими общими компонентами, как файлы и URL.

## Создание специального редактора свойств

Хотя встроенные редакторы свойств покрывают ряд стандартных случаев преобразования типов свойств, временами требуется создавать собственный редактор свойств, который предназначен для поддержки класса либо их набора в приложении. Платформа Spring полностью поддерживает регистрацию специальных редакторов свойств; единственный недостаток заключается в том, что в интерфейсе

`java.beans.PropertyEditor` присутствует большое количество методов, многие из которых не имеют отношения к решаемой задаче — преобразованию типов свойств. К счастью, в JDK 5 или более новых версиях предлагается класс `PropertyEditorSupport`, который могут расширять специальные редакторы свойств, оставляя вам реализацию только одного метода `setAsText()`.

Давайте рассмотрим простой пример реализации специального редактора свойств. Предположим, что имеется класс `Name` с двумя свойствами — именем и фамилией. Код этого класса показан в листинге 4.29.

**Листинг 4.29. Класс `Name`**

```
package com.apress.prospring4.ch4;

public class Name {
    private String firstName;
    private String lastName;

    public Name(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String toString() {
        return "First name: " + firstName + " - Last name: " + lastName;
    }
}
```

Для упрощения конфигурации приложения мы разработаем специальный редактор, который преобразует строку с разделителем-пробелом в имя и фамилию для класса `Name`. Код этого специального редактора свойств приведен в листинге 4.30.

**Листинг 4.30. Класс `NamePropertyEditor`**

```
package com.apress.prospring4.ch4;

import java.beans.PropertyEditorSupport;

public class NamePropertyEditor extends PropertyEditorSupport {
    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        String[] name = text.split("\\s");
```

```
        setValue(new Name(name[0], name[1]));
    }
}
```

Редактор очень прост. Он расширяет класс `PropertyEditorSupport` из JDK и реализует метод `setAsText()`. В этом методе мы просто расщепляем `String` на массив строк, используя в качестве разделителя символ пробела. Затем создается экземпляр класса `Name` с передачей для имени части строки, находящейся до пробела, а для фамилии — части строки после пробела. Наконец, преобразованное значение возвращается с помощью вызова метода `setValue()`.

Чтобы класс `NamePropertyEditor` можно было применять в приложении, его понадобится зарегистрировать в `ApplicationContext`. В листинге 4.31 представлена конфигурация `ApplicationContext`, в которой настраиваются бины типов `CustomEditorConfigurer` и `NamePropertyEditor` (`app-context-xml.xml`).

**Листинг 4.31. Использование `CustomEditorConfigurer`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean name="customEditorConfigurer"
        class="org.springframework.beans.factory.config.CustomEditorConfigurer">
        <property name="customEditors">
            <map>
                <entry key="com.apress.prospring4.ch4.Name"
                       value="com.apress.prospring4.ch4.NamePropertyEditor"/>
            </map>
        </property>
    </bean>
  <bean id="exampleBean" class="com.apress.prospring4.ch4.CustomEditorExample"
      p:name="Chris Schaefer"/>
</beans>
```

Относительно этой конфигурации следует отметить два момента. Во-первых, специальные редакторы свойств внедряются в класс `CustomEditorConfigurer` с использованием свойства `customEditors` типа `Map`. Во-вторых, каждая запись в `Map` представляет одиночный редактор свойств с ключом записи — именем класса, для которого этот редактор применяется. В коде видно, что ключом для `NamePropertyEditor` является `com.apress.prospring4.ch4.Name`, т.е. класс, для которого этот редактор должен использоваться.

В листинге 4.32 приведен код класса `CustomEditorExample`, который регистрировался в качестве бина в листинге 4.31.

**Листинг 4.32. Класс `CustomEditorExample`**

```
package com.apress.prospring4.ch4;
import org.springframework.context.support.GenericXmlApplicationContext;
```

```
public class CustomEditorExample {
    private Name name;

    public Name getName() {
        return name;
    }

    public void setName(Name name) {
        this.name = name;
    }

    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        CustomEditorExample bean =
            (CustomEditorExample) ctx.getBean("exampleBean");

        System.out.println(bean.getName());
    }
}
```

В этом коде нет ничего необычного. Запуск примера дает следующий вывод:

```
First name: Chris - Last name: Schaefer
```

Это вывод из метода `toString()`, который был реализован в классе `Name`; он показывает, что имя и фамилия правильно заполнены Spring с помощью сконфигурированного редактора свойств `NamePropertyEditor`.

Начиная с версии 3, платформа Spring предлагает средства Converter SPI (service provider interface — интерфейс поставщика служб) и Formatter SPI, которые предоставляют простой и хорошо структурированный API-интерфейс для выполнения преобразований типов и форматирования полей. Это особенно полезно при разработке веб-приложений. Интерфейсы Converter SPI и Formatter SPI подробно рассматриваются в главе 10.

# Дополнительные сведения о конфигурации `ApplicationContext`

Хотя мы уже обсуждали интерфейс `ApplicationContext`, по большому счету, мы касались возможностей, связанных с интерфейсом `BeanFactory`, оболочкой которого является `ApplicationContext`. В Spring различные реализации интерфейса `BeanFactory` отвечают за создание экземпляров бина, предоставляя поддержку внедрения зависимостей и жизненного цикла для бинов, которые управляются Spring. Тем не менее, как утверждалось ранее, помимо расширения `BeanFactory`, интерфейс `ApplicationContext` предлагает также и другую полезную функциональность.

Главная функция `ApplicationContext` заключается в предоставлении намного более развитой инфраструктуры, на основе которой строятся приложения. Интерфейс `ApplicationContext` гораздо больше (по сравнению с `BeanFactory`) осведомлен о бинах, которые сконфигурированы внутри него. В случае многих классов и интерфейсов инфраструктуры Spring, таких как `BeanFactoryPostProcessor`,

он взаимодействует с бинами от вашего имени, сокращая объем кода, который пришлось бы писать для использования Spring.

Самое большое преимущество от работы с интерфейсом `ApplicationContext` связано с тем, что он позволяет конфигурировать и управлять платформой Spring и контролируемыми ею ресурсами в полностью декларативной манере. Это означает, что по мере возможности Spring предоставляет классы поддержки для автоматической загрузки `ApplicationContext` в приложение, устраняя необходимость в написании какого-либо кода для доступа к `ApplicationContext`. На практике это средство доступно пока что только при построении веб-приложений с помощью Spring, позволяя инициализировать `ApplicationContext` в дескрипторе развертывания веб-приложения. В автономном приложении инициализация `ApplicationContext` требует простого кодирования.

Кроме предоставления модели, которая больше сосредоточена на декларативной конфигурации, интерфейс `ApplicationContext` поддерживает следующие дополнительные средства:

- интернационализация;
- публикация событий;
- управление и доступ к ресурсам;
- дополнительные интерфейсы жизненного цикла;
- улучшенное автоматическое конфигурирование компонент инфраструктуры.

В последующих разделах мы обсудим некоторые наиболее важные средства в `ApplicationContext` кроме DI.

## Интернационализация с помощью интерфейса `MessageSource`

Одной из областей, в которых Spring действительно превосходит другие платформы, является поддержка интернационализации (i18n). Посредством интерфейса `MessageSource` приложение может получать доступ к строковым ресурсам, называемым *сообщениями*, которые хранятся на разных языках. Для каждого языка, который должен поддерживаться в приложении, обеспечивается список сообщений, имеющих ключи для соответствия сообщениям на других языках. Например, если нужно отобразить английскую фразу-панграмму, которая дословно переводится, как "Шустрая бурая лиса прыгает через ленивую собаку", на английском и чешском языках, понадобится создать два сообщения, имеющих ключ `msg`; сообщение на английском выглядит как "The quick brown fox jumped over the lazy dog", а на чешском — "Příšerne žlutoucký kun úpel dábelské ódy".

Хотя для работы с `MessageSource` нет необходимости в использовании `ApplicationContext`, этот интерфейс расширяет `MessageSource` и предоставляет специальную поддержку для загрузки сообщений и для обеспечения их доступности в рамках среды. Автоматическая загрузка сообщений доступна в любой среде, но автоматический доступ предоставляется только в определенных управляемых Spring сценариях, таких как применение инфраструктуры Spring MVC для построения веб-приложения. Несмотря на то что любой класс может реализовать `ApplicationContextAware` и таким образом получить доступ к автоматически за-

груженным сообщениям, в разделе "Использование MessageSource в автономных приложениях" далее в этой главе мы предложим лучшее решение.

Если вы еще не знакомы с поддержкой i18n в Java, то прежде чем продолжить, рекомендуется просмотреть документацию Javadoc (http://download.java.net/jdk8/docs/api/index.html).

### Использование ApplicationContext и MessageSource

Помимо ApplicationContext платформа Spring предлагает три реализации MessageSource: ResourceBundleMessageSource, ReloadableResourceBundleMessageSource и StaticMessageSource.

Реализация StaticMessageSource не должна применяться в производственном приложении, поскольку ее нельзя конфигурировать внешне, а это, как правило, одно из главных требований при добавлении возможностей i18n к приложению. Реализация ResourceBundleMessageSource загружает сообщения с использованием Java-класса ResourceBundle.

Реализация ReloadableResourceBundleMessageSource в основном такая же, но поддерживает запланированную перезагрузку лежащих в основе исходных файлов.

Все три реализации MessageSource также реализуют еще один интерфейс по имени HierarchicalMessageSource, который позволяет вкладывать друг в друга экземпляры MessageSource. Это ключ к тому, как ApplicationContext работает с множеством экземпляров MessageSource.

Чтобы задействовать поддержку MessageSource, предоставляемую ApplicationContext, в конфигурации должен быть определен бин типа MessageSource с именем messageSource. Контекст ApplicationContext берет этот MessageSource и вкладывает его внутрь себя самого, разрешая доступ к сообщениям с применением ApplicationContext. Для лучшего понимания происходящего имеет смысл обратиться к примеру.

В листинге 4.33 показан код простого приложения, которое имеет доступ к набору сообщений для английской и чешской локалей.

**Листинг 4.33. Исследование MessageSource**

```
package com.apress.prospring4.ch4;

import java.util.Locale;

import org.springframework.context.support.GenericXmlApplicationContext;

public class MessageSourceDemo {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();
        Locale english = Locale.ENGLISH;
        Locale czech = new Locale("cs", "CZ");

        System.out.println(ctx.getMessage("msg", null, english));
        System.out.println(ctx.getMessage("msg", null, czech));

        System.out.println(ctx.getMessage("nameMsg", new Object[] { "Chris",
                "Schaefer" }, english));
    }
}
```

Пока что не переживайте по поводу вызовов метода `getMessage()`; вскоре мы к этому вернемся. Сейчас достаточно знать, что они извлекают сообщения с ключами для указанной локали. В листинге 4.34 представлена конфигурация, используемая этим приложением (`app-context-xml.xml`).

**Листинг 4.34. Конфигурирование бина `MessageSource`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:util="http://www.springframework.org/schema/util"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/util
       http://www.springframework.org/schema/util/spring-util.xsd">

   <bean id="messageSource"
     class="org.springframework.context.support.ResourceBundleMessageSource"
    p:basenames-ref="basenames"/>

   <util:list id="basenames">
      <value>buttons</value>
      <value>labels</value>
   </util:list>
</beans>
```

Здесь мы определяем бин `ResourceBundleMessageSource` по имени `messageSource`, как и требовалось, и конфигурируем его с набором имен для формирования основы набора файлов. Класс `ResourceBundle` из Java, который применяется реализацией `ResourceBundleMessageSource`, работает с набором файлов свойств, указанных с помощью базовых имен. При поиске сообщения для конкретной локали `ResourceBundle` ищет файл, имя которого состоит из комбинации базового имени и имени локали. Например, если базовым именем является `foo` и производится поиск сообщения для локали `en-GB` (британский английский), то `ResourceBundle` ищет файл по имени `foo_en_GB.properties`.

Содержимое файлов свойств для английского (`labels_en.properties`) и чешского (`labels_cs_CZ.properties`) языков, используемых в предыдущем примере, приведено в листингах 4.35 и 4.36.

**Листинг 4.35. Файл `labels_en.properties`**

```
msg=The quick brown fox jumped over the lazy dog
nameMsg=My name is {0} {1}
```

**Листинг 4.36. Файл `labels_cs_CZ.properties`**

```
msg=Příšerne žlutoucký kun úpel dábelské ódy
```

Выполнение класса `MessageSourceDemo` из листинга 4.33 дает следующий вывод:

```
The quick brown fox jumped over the lazy dog
Příšerne žlutoucký kun úpel dábelské ódy
My name is Chris Schaefer
```

Теперь этот пример стал вызывать даже больше вопросов. Что означают эти вызовы `getMessage()`? Почему мы применяли `ApplicationContext.getMessage()`, а не получали доступ к бину `ResourceBundleMessageSource` напрямую? Мы ответим на эти вопросы по очереди.

### *Использование метода `getMessage()`*

В интерфейсе `MessageSource` определены три перегруженных версии метода `getMessage()`, которые описаны в табл. 4.2.

**Таблица 4.2. Перегруженные версии метода `MessageSource.getMessage()`**

| Сигнатура метода | Описание |
|---|---|
| `getMessage(String, Object[], Locale)` | Это стандартный метод `getMessage()`. Аргумент типа `String` представляет собой ключ сообщения, соответствующий ключу в файле свойств. В листинге 4.33 первый вызов `getMessage()` использует в качестве ключа `msg`, и это соответствует следующей записи в файле свойств для локали en: `msg=The quick brown fox jumped over the lazy dog`. Аргумент типа массива `Object[]` предназначен для хранения замен в сообщении. В третьем вызове `getMessage()` в листинге 4.33 мы передаем массив из двух элементов `String`. Сообщением, имеющим ключ `nameMsg`, было `My name is {0} {1}`. Числа в фигурных скобках являются заполнителями, каждый из которых замещается соответствующим элементом в массиве, переданном как аргумент. Последний аргумент, `Locale`, указывает `ResourceBundleMessageSource`, какой файл свойств искать. Несмотря на то что в первом и втором вызовах `getMessage()` в примере применялся один и тот же ключ, вызовы возвращали разные сообщения, соответствующие значению `Locale`, которое было передано `getMessage()` |
| `getMessage(String, Object[], String, Locale)` | Эта перегруженная версия работает аналогично `getMessage(String, Object[], Locale)`, но принимает второй аргумент типа `String`, который позволяет передать стандартное значение на случай, если указанный ключ не доступен для локали, заданной в `Locale` |
| `getMessage(MessageSourceResolvable, Locale)` | Эта перегруженная версия представляет собой специальный случай, который детально обсуждается в разделе "Интерфейс `MessageSourceResolvable`" |

### *Причина использования `ApplicationContext` в качестве `MessageSource`*

Чтобы объяснить причину применения `ApplicationContext` в качестве `MessageSource`, мы должны немного забежать вперед и взглянуть на поддержку веб-приложений в Spring. В общем, причина связана с тем, что вы не должны использовать `ApplicationContext` как `MessageSource`, поскольку это приводит к нежелательной привязке вашего бина к `ApplicationContext` (более под-

робное обсуждение вы найдете в следующем разделе). Вы должны применять ApplicationContext при построении веб-приложения с помощью инфраструктуры MVC, поддерживаемой Spring.

Ключевым интерфейсом в Spring MVC является Controller. В отличие от платформ, подобных Struts, которые требуют реализации контроллеров путем наследования от конкретного класса, в Spring просто необходимо реализовать интерфейс Controller (либо снабдить разрабатываемый класс контроллера аннотацией @Controller). С учетом сказанного, Spring предоставляет коллекцию полезных базовых классов, которые вы будете использовать для реализации собственных контроллеров. Все эти базовые классы сами являются подклассами (прямо или косвенно) класса ApplicationObjectSupport, представляющего собой удобный суперкласс для любых объектов приложения, которые должны быть осведомлены о контексте ApplicationContext.

Не забывайте, что в конфигурации веб-приложения контекст ApplicationContext загружается автоматически. Класс ApplicationObjectSupport обращается к этому контексту ApplicationContext, помещает его в оболочку объекта MessageSourceAccessor и делает доступным контроллеру через защищенный метод getMessageSourceAccessor(). Класс MessageSourceAccessor предлагает широкий спектр удобных методов для работы с MessageSource. Эта форма *автоматического внедрения* исключительно полезна; она устраняет необходимость в наличии во всех контроллерах открытого свойства MessageSource.

Однако это не самая веская причина применения ApplicationContext в качестве MessageSource в веб-приложении. Главная причина использования ApplicationContext вместо ручного определения бина MessageSource заключается в том, что Spring, где возможно, открывает ApplicationContext как MessageSource для уровня презентаций. Это означает, что когда применяется библиотека дескрипторов JSP (JSTL), поддерживаемая Spring, дескриптор <spring:message> автоматически читает сообщения из ApplicationContext, и в случае использования JSTL дескриптор <fmt:message> делает то же самое.

Все указанные преимущества означают, что при разработке веб-приложения лучше применять поддержку MessageSource в ApplicationContext, чем управлять экземпляром MessageSource отдельно. Это особенно верно, если принять во внимание, что для работы с упомянутым средством нужно всего лишь сконфигурировать бин MessageSource с именем messageSource.

## Использование MessageSource в автономных приложениях

Когда бины MessageSource используются в автономных приложениях, в которых Spring не предлагает никакой дополнительной поддержки кроме автоматического вложения бина MessageSource в ApplicationContext, делать доступными бины MessageSource лучше с помощью внедрения зависимостей. Вы можете отдать предпочтение варианту, при котором бины реализуют интерфейс ApplicationContextAware, но это воспрепятствует их применению в контексте BeanFactory. Добавьте к этому факт усложнения тестирования без какой-либо видимой пользы, и необходимость придерживаться внедрения зависимостей для доступа к объектам MessageSource в автономных приложениях станет очевидной.

## Интерфейс MessageSourceResolvable

При поиске сообщения в MessageSource на месте ключа можно использовать объект, реализующий интерфейс MessageSourceResolvable, и набор аргументов. Наиболее широко этот интерфейс применяется в библиотеках проверки достоверности Spring для связывания объектов Error с их интернационализированными сообщениями об ошибках.

## События приложений

Еще одним средством ApplicationContext, отсутствующим в BeanFactory, является возможность публиковать и получать события, используя ApplicationContext в качестве брокера. В этом разделе мы посмотрим, как это делается.

### *Использование событий приложения*

Событие — это класс, производный от ApplicationEvent, который сам является производным от java.util.EventObject. Любой бин может прослушивать события, реализовав интерфейс ApplicationListener<T>; при этом ApplicationContext автоматически регистрирует любой сконфигурированный бин, который реализует данный интерфейс, в качестве прослушивателя. События публикуются с помощью метода ApplicationEventPublisher.publishEvent(), поэтому публикующий их класс должен быть осведомлен об интерфейсе ApplicationContext (расширяющем интерфейс ApplicationEventPublisher). В веб-приложении это достигается просто, т.к. многие классы являются производными от классов Spring Framework, которые делают возможным доступ к ApplicationContext через защищенный метод. Чтобы публиковать события в автономном приложении, бин должен реализовать интерфейс ApplicationContextAware.

В листинге 4.37 показан пример базового класса события.

**Листинг 4.37. Создание класса события**

```
package com.apress.prospring4.ch4;

import org.springframework.context.ApplicationEvent;

public class MessageEvent extends ApplicationEvent {
    private String msg;
    public MessageEvent(Object source, String msg) {
        super(source);
        this.msg = msg;
    }
    public String getMessage() {
        return msg;
    }
}
```

Код довольно прост; единственным моментом, который в нем следует отметить, является наличие в ApplicationEvent одного конструктора, принимающего ссылку на источник события. Это отражено в конструкторе MessageEvent.

В листинге 4.38 приведен код для прослушивателя.

**Листинг 4.38. Класс `MessageEventListener`**

```
package com.apress.prospring4.ch4;

import org.springframework.context.ApplicationListener;

public class MessageEventListener implements ApplicationListener<MessageEvent> {
    @Override
    public void onApplicationEvent(MessageEvent event) {
        MessageEvent msgEvt = (MessageEvent) event;
        System.out.println("Received: " + msgEvt.getMessage());
    }
}
```

В интерфейсе `ApplicationListener` определен единственный метод `onApplicationEvent()`, который вызывается Spring, когда событие сгенерировано. Класс `MessageEventListener` заинтересован только в событиях типа `MessageEvent` (или его подклассов), для чего он реализует строго типизированный интерфейс `ApplicationListener`. При получении `MessageEvent` он выводит сообщение в `stdout`. Публиковать события легко; понадобится только создать экземпляр класса события и передать его методу `ApplicationEventPublisher.publishEvent()`, как показано в листинге 4.39.

**Листинг 4.39. Публикация события**

```
package com.apress.prospring4.ch4;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Publisher implements ApplicationContextAware {
    private ApplicationContext ctx;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
            throws BeansException {
        this.ctx = applicationContext;
    }

    public void publish(String message) {
        ctx.publishEvent(new MessageEvent(this, message));
    }

    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext(
                "classpath:META-INF/spring/app-context-xml.xml");

        Publisher pub = (Publisher) ctx.getBean("publisher");
        pub.publish("Hello World!");
        pub.publish("The quick brown fox jumped over the lazy dog");
    }
}
```

Здесь класс Publisher извлекает из ApplicationContext экземпляр самого себя и затем посредством метода publish() публикует два события MessageEvent для ApplicationContext. Экземпляр бина Publisher получает доступ к ApplicationContext за счет реализации ApplicationContextAware. В листинге 4.40 показана конфигурация для этого примера (app-context-xml.xml).

**Листинг 4.40. Конфигурирование бинов ApplicationListener**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="publisher" class="com.apress.prospring4.ch4.Publisher"/>
   <bean id="messageEventListener"
      class="com.apress.prospring4.ch4.MessageEventListener"/>
</beans>
```

Обратите внимание, что для регистрации MessageEventListener с ApplicationContext никакой специальной конфигурации не требуется; платформа Spring делает это автоматически. Запуск примера дает следующий вывод:

```
Received: Hello World!
Received: The quick brown fox jumped over the lazy dog
```

### *Соображения по поводу использования событий*

Во многих случаях в приложении необходимо уведомлять определенные компоненты о тех или иных событиях. Часто это делается путем написания кода для явного уведомления каждого компонента или за счет применения какой-то технологии обмена сообщениями, подобной JMS. Недостаток приема с написанием кода для уведомления каждого компонента по очереди состоит в том, что вы привязываете компоненты к публикующему классу, что во многих случаях нежелательно.

Представьте ситуацию, когда вы кешируете подробную информацию о товаре внутри приложения, чтобы избежать лишних обращений к базе данных. Еще один компонент позволяет модифицировать подробную информацию о товаре и сохранять в базе данных. Чтобы данные в кеше не стали недействительными, компонент обновления явно уведомляет кеш о том, что информация о товаре изменилась. В этом примере компонент обновления связан с компонентом, который в действительности не имеет никакого отношения к его бизнес-ответственности. Более удачное решение может выглядеть следующим образом: компонент обновления публикует событие всякий раз, когда подробная информация о товаре изменяется, а заинтересованные компоненты, такие как кеш, прослушивают это событие. Преимущество этого решения заключается в том, что компоненты сохраняются несвязанными, а это упрощает удаление кеша, когда он больше не нужен, или добавление другого прослушивателя, который заинтересован в получении сведений об изменении подробной информации о товаре.

Применение технологии JMS в этом случае было бы излишним, поскольку процесс объявления недействительной записи о товаре в кеше является быстрым и не-

критичным. Использование инфраструктуры для работы с событиями в Spring приводит к добавлению в приложение весьма незначительных накладных расходов.

Как правило, события применяются для логики реагирования, которая выполняется быстро и не является частью главной логики приложения. В предшествующем примере объявление записи о товаре в кеше недействительной происходит в ответ на обновление подробной информации о товаре, это осуществляется быстро (во всяком случае, должно) и не относится к главной функции приложения. Для процессов, которые выполняются долго и формируют часть основной бизнес-логики, рекомендуется использовать JMS или аналогичную систему обмена сообщениями, такую как RabbitMQ. Основные преимущества применения технологии JMS связаны с тем, что она больше подходит для длительно выполняющихся процессов, и по мере роста системы управляемую JMS обработку сообщений, которые содержат бизнес-информацию, можно при необходимости вынести на отдельную машину.

## Доступ к ресурсам

Часто приложение нуждается в доступе к ресурсам в разнообразных формах. Может понадобиться доступ к конфигурационным данным, хранящимся в файле внутри файловой системы, данным образа, записанным в JAR-файле в пути классов, или каким-нибудь данным, расположенным на произвольном сервере. Платформа Spring предоставляет унифицированный механизм для доступа к ресурсам независимым от протокола способом. Это означает, что приложение может работать с файловым ресурсом одинаковым образом, где бы он ни находился: в файловой системе, в пути классов или на удаленном сервере.

В основе всей поддержки ресурсов Spring находится интерфейс `org.springframework.core.io.Resource`. В интерфейсе `Resource` определены десять самоочевидных методов: `contentLength()`, `exists()`, `getDescription()`, `getFile()`, `getFileName()`, `getURI()`, `getURL()`, `isOpen()`, `isReadable()` и `lastModified()`. В дополнение к этим десяти методам имеется еще один, не столь самоочевидный: `createRelative()`. Метод `createRelative()` создает новый экземпляр `Resource`, используя путь относительно экземпляра, на котором он вызывается. Можно построить собственные реализации `Resource`, хотя это и выходит за рамки материала этой главы, но в большинстве случаев будет применяться одна из встроенных реализаций для доступа к файлу (класс `FileSystemResource`), пути классов (класс `ClassPathResource`) или URL-ресурсам (класс `UrlResource`).

Для поиска и создания экземпляров `Resource` платформа Spring внутренне использует другой интерфейс, `ResourceLoader`, и его стандартную реализацию `DefaultResourceLoader`. Однако обычно вы не будете взаимодействовать с `DefaultResourceLoader`, а вместо этого работать с другой реализацией `ResourceLoader` — `ApplicationContext`.

В листинге 4.41 приведен пример приложения, которое обращается к трем ресурсам посредством `ApplicationContext`.

**Листинг 4.41. Доступ к ресурсам**

```
package com.apress.prospring4.ch4;

import java.io.File;
import org.springframework.context.ApplicationContext;
```

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.Resource;

public class ResourceDemo {
   public static void main(String[] args) throws Exception{
      ApplicationContext ctx = new ClassPathXmlApplicationContext();

      File file = File.createTempFile("test", "txt");
      file.deleteOnExit();

      System.out.println(file.getPath());
      Resource res1 = ctx.getResource("file://" + file.getPath());
      displayInfo(res1);

      Resource res2 = ctx.getResource("classpath:test.txt");
      displayInfo(res2);

      Resource res3 = ctx.getResource("http://www.google.com");
      displayInfo(res3);
   }

   private static void displayInfo(Resource res) throws Exception{
      System.out.println(res.getClass());
      System.out.println(res.getURL().getContent());
      System.out.println("");
   }
}
```

Обратите внимание, что в каждом вызове `getResource()` передается URI для каждого ресурса. Вы должны узнать общепринятые протоколы `file:` и `http:`, указанные для `res1` и `res3`. Протокол `classpath:`, применяемый для `res2`, является специфичным для Spring и указывает, что `ResourceLoader` должен искать ресурс в пути классов. Запуск этого примера дает в результате следующий вывод:

```
class org.springframework.core.io.UrlResource
java.io.BufferedInputStream@6a024a67

class org.springframework.core.io.ClassPathResource
sun.net.www.content.text.PlainTextInputStream@4de8b406

class org.springframework.core.io.UrlResource
sun.net.www.protocol.http.HttpURLConnection$HttpInputStream@48eff760
```

Как видите, для протоколов `file:` и `http:` платформа Spring возвращает экземпляр `UrlResource`. Она включает класс `FileSystemResource`, но `DefaultResourceLoader` вообще не использует этот класс. Причина в том, что стандартная стратегия загрузки ресурсов в Spring трактует URL и файл как один и тот же тип ресурса, но с отличающимися протоколами (т.е. `file:` и `http:`). Если экземпляр `FileSystemResource` обязателен, применяйте `FileSystemResourceLoader`. Получив экземпляр `Resource`, вы можете работать с содержимым ресурса по своему усмотрению, используя `getFile()`, `getInputStream()` или `getURL()`. В ряде случаев, например, когда применяется протокол `http:`, вызов `getFile()` генерирует исключение `FileNotFoundException`. По этой причине для доступа к содержимому ресурсов рекомендуется использовать метод `getInputStream()`, потому что он, скорее всего, будет функционировать со всеми возможными типами ресурсов.

# Конфигурация, использующая Java-классы

Кроме XML для конфигурирования `ApplicationContext` можно также применять Java-классы. Ранее для этого использовался отдельный проект Spring JavaConfig, но, начиная с версии Spring 3.0, его основные средства, касающиеся конфигурирования с помощью Java-классов, были объединены с ядром Spring Framework.

В этом разделе мы посмотрим, каким образом применять Java-классы для конфигурирования `ApplicationContext`, и приведем эквивалентную XML-конфигурацию.

## Конфигурирование `ApplicationContext` в Java

Мы рассмотрим конфигурирование `ApplicationContext` с использованием Java-классов на том же самом примере поставщика и визуализатора сообщений, который был представлен в главе 3. Для удобства в листинге 4.42 повторно приведен код интерфейса поставщика сообщений и класса конфигурируемого поставщика сообщений.

**Листинг 4.42. `MessageProvider` и `ConfigurableMessageProvider`**

```
package com.apress.prospring4.ch4;

public class ConfigurableMessageProvider implements MessageProvider {
    private String message = "Default message";

    public ConfigurableMessageProvider() {
    }

    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    @Override
    public String getMessage() {
        return message;
    }
}
```

В листинге 4.43 показаны интерфейс `MessageRenderer` и реализация `StandardOutMessageRenderer`.

**Листинг 4.43. `MessageRenderer` и `StandardOutMessageRenderer`**

```
package com.apress.prospring4.ch4;

public interface MessageRenderer {
    void render();
    void setMessageProvider(MessageProvider provider);
    MessageProvider getMessageProvider();
}

package com.apress.prospring4.ch4;
```

```
public class StandardOutMessageRenderer implements MessageRenderer {
   private MessageProvider messageProvider;

   @Override
   public void render() {
      if (messageProvider == null) {
         throw new RuntimeException(
            "You must set the property messageProvider of class:"
               + StandardOutMessageRenderer.class.getName());
      }
      System.out.println(messageProvider.getMessage());
   }

   @Override
   public void setMessageProvider(MessageProvider provider){
      this.messageProvider = provider;
   }

   @Override
   public MessageProvider getMessageProvider() {
      return this.messageProvider;
   }
}
```

В листинге 4.44 представлена XML-конфигурация для `ApplicationContext` (`app-context-xml.xml`).

**Листинг 4.44. XML-конфигурация**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:c="http://www.springframework.org/schema/c"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="messageRenderer"
      class="com.apress.prospring4.ch4.StandardOutMessageRenderer"
      p:messageProvider-ref="messageProvider"/>

   <bean id="messageProvider"
      class="com.apress.prospring4.ch4.ConfigurableMessageProvider"
      c:message="This is a configurable message"/>
</beans>
```

Наконец, в листинге 4.45 приведен код тестовой программы.

**Листинг 4.45. Тестовая программа для XML-конфигурации**

```
package com.apress.prospring4.ch4;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class JavaConfigXMLExample {
```

```
    public static void main(String[] args) {
        ApplicationContext ctx = new
            ClassPathXmlApplicationContext(
              "classpath:META-INF/spring/app-context-xml.xml");

        MessageRenderer renderer =
            ctx.getBean("messageRenderer", MessageRenderer.class);

        renderer.render();
    }
}
```

Выполнение этой программы дает следующий вывод:

```
This is a configurable message
```

В случае применения Java-класса вместо XML для конфигурирования показанных выше поставщика и визуализатора сообщений необходимо просто реализовать обычный компонент JavaBean, но с соответствующими аннотациями для Java-конфигурации в Spring. В листинге 4.46 представлен Java-класс, который эквивалентен XML-конфигурации из листинга 4.44.

**Листинг 4.46. Java-конфигурация**

```
package com.apress.prospring4.ch4;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
    @Bean
    public MessageProvider messageProvider() {
        return new ConfigurableMessageProvider();
    }

    @Bean
    public MessageRenderer messageRenderer() {
        MessageRenderer renderer = new StandardOutMessageRenderer();
        renderer.setMessageProvider(messageProvider());

        return renderer;
    }
}
```

В классе `AppConfig` сначала используется аннотация `@Configuration` для информирования платформы Spring о том, что это конфигурационный файл, основанный на Java. После этого для объявления бина Spring и требований DI применяется аннотация `@Bean`. Аннотация `@Bean` эквивалентна дескриптору `<bean>`, а имя метода — атрибуту `id` дескриптора `<bean>`. При создании экземпляра бина `MessageRender` внедрение зависимости через метод установки достигается вызовом соответствующего метода для получения поставщика сообщений, что дает такой же результат, как использование атрибута `ref` в конфигурации XML. В листинге 4.47 показано, как инициализировать `ApplicationContext` из Java-конфигурации.

**Листинг 4.47. Тестирование Java-конфигурации**

```
package com.apress.prospring4.ch4;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationC
ontext;

public class JavaConfigSimpleExample {
   public static void main(String[] args) {
      ApplicationContext ctx = new
         AnnotationConfigApplicationContext(AppConfig.class);

      MessageRenderer renderer =
         ctx.getBean("messageRenderer", MessageRenderer.class);

      renderer.render();
   }
}
```

Здесь применяется класс `AnnotationConfigApplicationContext`, конструктору которого в качестве аргумента передается класс конфигурации (ему также можно передавать множество классов конфигурации с помощью аргументов переменной длины из JDK). После этого возвращенный экземпляр `ApplicationContext` можно использовать обычным образом.

Запуск этой программы дает следующий результат:

```
Default message
```

После ознакомления с базовым применением Java-класса конфигурации давайте перейдем к рассмотрению других вариантов конфигурации. Пусть для поставщика сообщений необходимо вынести сообщение в файл свойств (`message.properties`) и затем внедрять его в `ConfigurableMessageProvider` с помощью Constructor Injection. Содержимое файла `message.properties` таково:

```
message=Hello from Spring Java Configuration
```

Взглянем на переделанную тестовую программу, которая загружает файлы свойств, используя аннотацию `@PropertySource`, и затем внедряет их в реализацию поставщика сообщений. В листинге 4.48 также добавлено много разнообразных аннотаций, которые Spring поддерживает для базовой Java-конфигурации. Обратите внимание, что для работы аннотации `@EnableTransactionManagement` необходимо добавить к проекту зависимость от модуля `spring-tx` (табл. 4.3).

**Таблица 4.3. Зависимость для поддержки транзакций в Spring**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.springframework` | `spring-tx` | 4.0.2.RELEASE | Модуль Spring для поддержки транзакций |

**Листинг 4.48. Java-класс конфигурации `AppConfig` (переделанный)**

```
package com.apress.prospring4.ch4;

import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.DependsOn;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.ImportResource;
import org.springframework.context.annotation.Lazy;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.annotation.Scope;
import org.springframework.core.env.Environment;
import org.springframework.transaction.annotation.
EnableTransactionManagement;

@Configuration
@ImportResource(value="classpath:META-INF/spring/app-context-xml.xml")
@PropertySource(value="classpath:message.properties")
@ComponentScan(basePackages={"com.apress.prospring4.ch4"})
@EnableTransactionManagement
public class AppConfig {
    @Autowired
    Environment env;

    @Bean
    @Lazy(value=true)
    public MessageProvider messageProvider() {
        return new ConfigurableMessageProvider(env.getProperty("message"));
    }

    @Bean(name="messageRenderer")
    @Scope(value="prototype")
    @DependsOn(value="messageProvider")
    public MessageRenderer messageRenderer() {
        MessageRenderer renderer = new StandardOutMessageRenderer();
        renderer.setMessageProvider(messageProvider());

        return renderer;
    }
}
```

В листинге 4.48 можно заметить множество распространенных аннотаций для конфигурирования с помощью Java-классов. Ниже перечислены основные моменты, связанные с этим листингом.

- Аннотация `@PropertySource` применяется для загрузки файлов свойств в `ApplicationContext` и принимает в качестве аргумента местоположение (допускается указывать более одного местоположения). В XML той же самой цели служит дескриптор `<context:property-placeholder>`.
- Можно также использовать аннотацию `@ImportResource` для импортирования конфигурации из XML-файлов, что означает возможность совместного применения XML и Java-классов конфигурации, хотя поступать подобным образом не рекомендуется. Смешивание конфигураций XML и Java усложнит сопровождение приложения, поскольку в поисках специфического бина придется просматривать и XML-файлы, и Java-классы.

- Кроме `@ImportResource` для импортирования других классов конфигурации можно использовать аннотацию `@Import`, а это значит, что можно иметь множество Java-классов конфигурации для разных конфигураций (например, один класс может быть выделен для объявления бинов DAO, один — для объявления бинов служб и т.д.).
- Аннотация `@ComponentScan` определяет пакеты, которые платформа Spring должна сканировать на предмет аннотаций для определений бинов. Аналогичную роль играет дескриптор `<context:component-scan>` в XML-конфигурации.
- Остальные аннотации самоочевидны. Аннотация `@Lazy` заставляет Spring создавать экземпляры бина, только когда он запрашивается (то же, что и `lazy-init="true"` в XML), а `@DependsOn` сообщает Spring о том, что определенный бин зависит от других бинов, поэтому Spring должна обеспечить создание их экземпляров первыми. Аннотация `@Scope` определяет область действия бина.
- Службы инфраструктуры приложения также могут быть определены в Java-классах. Например, `@EnableTransactionManagement` указывает, что будет применяться средство управления транзакциями Spring, которое подробно рассматривается в главе 9.
- Вы могли также заметить аннотацию `@Autowired` у переменной `env` типа `Environment`. Это средство абстрагирования `Environment`, предоставляемое Spring. Мы обсудим его далее в этой главе.

Выполнение тестовой программы дает следующий вывод:

```
Hello from Spring Java Configuration
```

Это сообщение, которое определено в файле `message.properties`.

### Выбор между конфигурациями Java и XML

Как вы уже знаете, использование Java-классов может обеспечить тот же самый уровень конфигурирования `ApplicationContext`, что и XML. Так какой же подход выбрать? Соображение по этому поводу очень похоже на то, что принималось во внимание при выборе конфигурации DI — в стиле XML или в стиле аннотаций Java-аннотаций. Ясно, что каждый подход имеет свои плюсы и минусы. Тем не менее, рекомендация остается такой же: решив в команде разработчиков, какой подход применять, придерживайтесь его и сохраняйте стиль конфигурирования неизменным, не разбрасываясь между Java-классами и XML-файлами. Использование одного подхода существенно упростит работу по сопровождению.

## Профили

Еще одной интересной возможностью, предлагаемой Spring, является концепция профилей конфигурации. В сущности, *профиль* заставляет Spring конфигурировать только тот контекст `ApplicationContext`, который определен, когда указанный профиль становится активным. В этом разделе мы продемонстрируем применение профилей в простой программе.

## Пример использования средства профилей Spring

Пусть имеется служба по имени FoodProviderService, которая отвечает за обеспечение едой учебных заведений, включая детский сад (kindergarten) и среднюю школу (high school). В интерфейсе FoodProviderService определен только один метод provideLunchSet(), который формирует обеденный набор для каждого учащегося вызываемого учебного заведения. Обеденный набор представляет собой список объектов типа Food — очень простого класса, который имеет единственный атрибут name. Код класса Food показан в листинге 4.49.

**Листинг 4.49. Класс Food**

```
package com.apress.prospring4.ch4;

public class Food {
    private String name;

    public Food() {
    }

    public Food(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

В листинге 4.50 приведен интерфейс FoodProviderService.

**Листинг 4.50. Интерфейс FoodProviderService**

```
package com.apress.prospring4.ch4;

import java.util.List;

public interface FoodProviderService {
    List<Food> provideLunchSet();
}
```

Теперь предположим, что существуют два поставщика обеденных наборов, один для детского сада и один для средней школы. Хотя производимые обеденные наборы отличаются, предлагаемая ими служба является той же самой — поставка обедов учащимся. Итак, давайте создадим две разных реализации FoodProviderService, назначив им одно и то же имя, но поместив их в разные пакеты для идентификации целевых учебных заведений. Код для этих двух классов показан в листингах 4.51 и 4.52.

**Листинг 4.51. Реализация FoodProviderService для детского сада**

```
package com.apress.prospring4.ch4.kindergarten;

import java.util.ArrayList;
```

```
import java.util.List;

import com.apress.prospring4.ch4.Food;
import com.apress.prospring4.ch4.FoodProviderService;

public class FoodProviderServiceImpl implements FoodProviderService {
    @Override
    public List<Food> provideLunchSet() {
        List<Food> lunchSet = new ArrayList<Food>();
        lunchSet.add(new Food("Milk"));
        lunchSet.add(new Food("Biscuits"));
        return lunchSet;
    }
}
```

**Листинг 4.52. Реализация `FoodProviderService` для средней школы**

```
package com.apress.prospring4.ch4.highschool;

import java.util.ArrayList;
import java.util.List;

import com.apress.prospring4.ch4.Food;
import com.apress.prospring4.ch4.FoodProviderService;

public class FoodProviderServiceImpl implements FoodProviderService {
    @Override
    public List<Food> provideLunchSet() {
        List<Food> lunchSet = new ArrayList<Food>();
        lunchSet.add(new Food("Coke"));
        lunchSet.add(new Food("Hamburger"));
        lunchSet.add(new Food("French Fries"));

        return lunchSet;
    }
}
```

В этих листингах видно, что две реализации предоставляют один и тот же интерфейс `FoodProviderService`, но производят отличающиеся комбинации еды в обеденных наборах. А теперь предположим, что в детском саду хотят, чтобы поставщик доставлял обеденные наборы их воспитанникам; давайте посмотрим, как для достижения этого можно воспользоваться конфигурацией профиля Spring. Сначала мы рассмотрим XML-конфигурацию.

Мы создадим два XML-файла конфигурации, один для профиля детского сада и один для профиля средней школы. В листингах 4.53 и 4.54 приведена конфигурация поставщиков еды для детского сада и средней школы, соответственно.

**Листинг 4.53. XML-конфигурация для детского сада (`kindergarten-config.xml`)**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd"
        profile="kindergarten">
```

```
    <bean id="foodProviderService"
     class="com.apress.prospring4.ch4.kindergarten.FoodProviderServiceImpl"/>
</beans>
```

**Листинг 4.54. XML-конфигурация для средней школы (highschool-config.xml)**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd"
        profile="highschool">

    <bean id="foodProviderService"
      class="com.apress.prospring4.ch4.highschool.FoodProviderServiceImpl"/>
</beans>
```

В предыдущих двух конфигурациях обратите внимание на использование атрибутов `profile="kindergarten"` и `profile="highschool"` в дескрипторе `<beans>`. Это сообщает Spring о том, что экземпляры бинов должны создаваться, только если активен указанный профиль. Давайте посмотрим, как активизировать корректный профиль, когда `ApplicationContext` применяется в автономном приложении. Тестовая программа приведена в листинге 4.55.

**Листинг 4.55. Пример XML-конфигурации профиля**

```
package com.apress.prospring4.ch4;

import java.util.List;
import org.springframework.context.support.GenericXmlApplicationContext;

public class ProfileXmlConfigExample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/*-config.xml");
        ctx.refresh();

        FoodProviderService foodProviderService =
            ctx.getBean("foodProviderService", FoodProviderService.class);

        List<Food> lunchSet = foodProviderService.provideLunchSet();

        for (Food food: lunchSet) {
            System.out.println("Food: " + food.getName());
        }
    }
}
```

Метод `ctx.load()` загрузит оба файла `kindergarten-config.xml` и `highschool-config.xml`, т.к. ему передается групповой символ в виде префикса. В этом примере на основе атрибута `profile` платформа Spring создаст экземпляры только бинов из файла `kindergarten-config.xml`, который активизируется JVM-аргументом `-Dspring.profiles.active="kindergarten"`.

Выполнение программы дает следующий вывод:

```
Food: Milk
Food: Biscuits
```

Это в точности то, что реализация поставщика для детского сада сформирует для обеденного набора. А теперь изменим аргумент профиля в предыдущем листинге на `highschool` (`-Dspring.profiles.active="highschool"`); вывод станет таким:

```
Food: Coke
Food: Hamburger
Food: French Fries
```

Итак, теперь вы знаете, как применять профили. Можно также программно устанавливать используемый профиль, вызвав `ctx.getEnvironment().setActiveProfiles("kindergarten")`. Вдобавок можно зарегистрировать классы, которые должны стать доступными посредством профилей, с помощью JavaConfig, просто снабдив класс аннотацией `@Profile`.

### Соображения по поводу использования профилей

Средство профилей в Spring предлагает разработчикам еще один способ управления конфигурацией запуска приложений, который обычно реализуется в инструментах сборки (например, поддержка профилей в Maven). Инструменты сборки на основе переданных им аргументов упаковывают нужные файлы конфигурации/свойств в Java-архив (JAR или WAR в зависимости от типа приложения) и затем развертывают его в целевой среде. Средство профилей Spring позволяет разработчикам приложений самостоятельно определять профили и активизировать их либо программно, либо с помощью аргумента JVM. За счет использования поддержки профилей Spring можно иметь один архив приложения и развертывать его во всех средах, передавая подходящий профиль в качестве аргумента во время начальной загрузки JVM. Скажем, могут существовать приложения с разными профилями, такими как (`dev`, `hibernate`), (`prd`, `jdbc`) и т.д., при этом каждая комбинация представляет среду запуска (среда разработки или производственная среда) и применяемую библиотеку доступа к данным (Hibernate или JDBC). В итоге управление профилями приложения переносится на сторону программирования.

Но этому подходу присущи также и недостатки. Некоторые могут возразить, что помещение всей конфигурации для различных сред в конфигурационные файлы приложения или в Java-классы с последующим связыванием их вместе будет подвержено ошибкам в случае небрежной обработки (например, администратор может забыть об установке нужного аргумента JVM в среде сервера приложений). Упаковка файлов сразу для всех профилей также приводит к увеличению размера пакета. Здесь снова следует напомнить: позвольте требованиям к приложению и конфигурации управлять вашим выбором подхода, который наилучшим образом соответствует разрабатываемому проекту.

## Абстракция `Environment` и `PropertySource`

Для установки активного профиля необходимо обратиться к интерфейсу `Environment`. Этот интерфейс представляет собой уровень абстракции, предназначенный для инкапсуляции среды выполняющегося приложения Spring.

Кроме профилей интерфейс Environment инкапсулирует и другие ключевые порции информации — свойства. Свойства служат для сохранения лежащей в основе приложения конфигурации среды, куда входит местоположение папки приложения, параметры подключения к базе данных и т.д.

Возможности абстракции Environment и PropertySource в Spring содействуют разработчикам в доступе к разнообразной конфигурационной информации, связанной с платформой запуска. В рамках этой абстракции все свойства системы, переменные среды и свойства приложения обслуживаются интерфейсом Environment, который Spring наполняет во время начальной загрузки ApplicationContext.

В листинге 4.56 показан простой пример.

**Листинг 4.56. Пример работы с абстракцией Environment**

```
package com.apress.prospring4.ch4;

import java.util.HashMap;
import java.util.Map;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.MapPropertySource;
import org.springframework.core.env.MutablePropertySources;

public class EnvironmentSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.refresh();

      ConfigurableEnvironment env = ctx.getEnvironment();
      MutablePropertySources propertySources = env.getPropertySources();
      Map appMap = new HashMap();
      appMap.put("application.home", "application_home");
      propertySources.addLast(new MapPropertySource("PROSPRING4_MAP", appMap));

      System.out.println("user.home: " + System.getProperty("user.home"));
      System.out.println("JAVA_HOME: " + System.getenv("JAVA_HOME"));

      System.out.println("user.home: " + env.getProperty("user.home"));
      System.out.println("JAVA_HOME: " + env.getProperty("JAVA_HOME"));
      System.out.println("application.home: "
         + env.getProperty("application.home"));
   }
}
```

В листинге 4.56 после инициализации ApplicationContext мы получаем ссылку на интерфейс ConfigurableEnvironment. Через этот интерфейс извлекается обработчик MutablePropertySources (стандартная реализация интерфейса PropertySources, которая позволяет манипулировать содержащимися источниками свойств). Затем мы создаем карту, помещаем в нее свойства приложения и конструируем экземпляр класса MapPropertySource (подкласс PropertySource, который читает ключи и значения из экземпляра Map) для этой карты. Наконец, мы добавляем класс MapPropertySource к MutablePropertySources с помощью метода addLast().

Запустив программу, мы получаем следующий вывод:

```
user.home: /home/chris
JAVA_HOME: /home/chris/bin/java
user.home: /home/chris
JAVA_HOME: /home/chris/bin/java
application.home: application_home
```

В первых двух строках свойство системы JVM по имени `user.home` и переменная среды `JAVA_HOME` извлекаются, как и ранее (с использованием класса `System` из JVM). Тем не менее, в последних трех строках видно, что доступ ко всем свойствам системы, переменным среды и свойствам приложения может быть получен через интерфейс `Environment`. Это является хорошей иллюстрацией того, как абстракция `Environment` может помочь в управлении и доступе ко всему многообразию свойств внутри среды запуска приложения.

В рамках абстракции `PropertySource` платформа Spring обращается к свойствам в следующем стандартном порядке:

- свойства системы для выполняющейся машины JVM;
- переменные среды;
- свойства, определяемые приложением.

Для примера предположим, что вы определили то же самое свойство приложения, `user.home`, и добавили его к интерфейсу `Environment` через класс `MutablePropertySources`. Запустив программу, вы увидите, что по-прежнему извлекается свойство `user.home` из JVM, а не то, которое было определено вами. Однако Spring позволяет управлять порядком, в соответствии с которым интерфейс `Environment` извлекает свойства. Давайте немного изменим код в листинге 4.56 и посмотрим, как это работает. Модифицированная версия показана в листинге 4.57 (отличия выделены полужирным).

**Листинг 4.57. Модифицированный пример работы с абстракцией `Environment`**

```
package com.apress.prospring4.ch4;

import java.util.HashMap;
import java.util.Map;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.MapPropertySource;
import org.springframework.core.env.MutablePropertySources;

public class EnvironmentSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.refresh();

        ConfigurableEnvironment env = ctx.getEnvironment();
        MutablePropertySources propertySources = env.getPropertySources();

        Map appMap = new HashMap();
        appMap.put("user.home", "application_home");

        propertySources.addFirst(new MapPropertySource("PROSPRING4_MAP", appMap));
```

```
        System.out.println("user.home: " + System.getProperty("user.home"));
        System.out.println("JAVA_HOME: " + System.getenv("JAVA_HOME"));
        System.out.println("user.home: " + env.getProperty("user.home"));
        System.out.println("JAVA_HOME: " + env.getProperty("JAVA_HOME"));
    }
}
```

В листинге 4.57 мы определили свойство приложения по имени `user.home` и добавили его как первое свойство для поиска с помощью метода `addFirst()` класса `MutablePropertySources`. В результате выполнения программы получается следующий вывод:

```
user.home: /home/chris
JAVA_HOME: /home/chris/bin/java
user.home: application_home
JAVA_HOME: /home/chris/bin/java
```

Первые две строки остались теми же самыми, т.к. для извлечения показанных в них значений по-прежнему применялись методы `getProperty()` и `getenv()` класса `System` из JVM. Однако при использовании интерфейса `Environment` видно, что определенное нами свойство `user.home` получает преимущество, поскольку оно определено как первое для поиска значений свойств.

В реальных приложениях необходимость непосредственного взаимодействия с интерфейсом `Environment` возникает редко. Чаще всего будет применяться заполнитель свойства в форме `${}` (например, `${application.home}`) и производиться внедрение полученного значения в бины Spring. Давайте посмотрим на это в действии.

Предположим, что есть класс `AppProperty`, который предназначен для хранения всех свойств приложения, загруженных из файла свойств. Код этого класса показан в листинге 4.58.

**Листинг 4.58. Пример использования заполнителей свойств Spring**

```
package com.apress.prospring4.ch4;

public class AppProperty {
    private String applicationHome;
    private String userHome;

    public String getApplicationHome() {
        return applicationHome;
    }

    public void setApplicationHome(String applicationHome) {
        this.applicationHome = applicationHome;
    }

    public String getUserHome() {
        return userHome;
    }

    public void setUserHome(String userHome) {
        this.userHome = userHome;
    }
}
```

В листинге 4.59 представлено содержимое файла application.properties, в котором хранятся свойства выполняющегося приложения.

**Листинг 4.59. Файл application.properties**

```
application.home=application_home
user.home=/home/chris-new
```

Обратите внимание, что в этом файле свойств также объявлено свойство user.home. Давайте взглянем на XML-конфигурацию Spring, приведенную в листинге 4.60 (app-context-xml.xml).

**Листинг 4.60. Конфигурация для примера использования заполнителей свойств Spring**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

   <context:property-placeholder location="classpath:application.properties"/>

   <bean id="appProperty" class="com.apress.prospring4.ch4.AppProperty">
       <property name="applicationHome" value="${application.home}"/>
       <property name="userHome" value="${user.home}"></property>
   </bean>
</beans>
```

Мы используем дескриптор <context:property-placeholder> для загрузки свойств в Spring-интерфейс Environment, который помещен в оболочку интерфейса ApplicationContext. Кроме того, мы применяем заполнители для внедрения значений в бин AppProperty. В листинге 4.61 показан код тестовой программы.

**Листинг 4.61. Тестирование использования заполнителей свойств Spring**

```
package com.apress.prospring4.ch4;

import org.springframework.context.support.GenericXmlApplicationContext;

public class PlaceHolderSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        AppProperty appProperty = ctx.getBean("appProperty", AppProperty.class);

        System.out.println("application.home: " + appProperty.getApplicationHome());
        System.out.println("user.home: " + appProperty.getUserHome());
    }
}
```

Запуск этой программы приводит к получению следующего вывода:

```
application.home: application_home
user.home: /home/chris
```

В выводе видно, что заполнитель `application.home` был соответствующим образом распознан, в то время как свойство `user.home` по-прежнему извлекалось из свойств JVM, что вполне корректно, поскольку таково стандартное поведение абстракции `PropertySource`. Чтобы заставить Spring назначить более высокий приоритет значениям из файла `application.properties`, к дескриптору `<context:property-placeholder>` добавляется атрибут `local-override="true"`:

```
<context:property-placeholder local-override="true"
   location="classpath:env/application.properties"/>
```

Атрибут `local-override` сообщает Spring о необходимости переопределить существующие свойства с использованием свойств, определенных в этом заполнителе. Запустив программу, вы увидите, что теперь извлекается свойство `user.home` из файла `application.properties`:

```
application.home: application_home
user.home: /home/chris-new
```

# Конфигурация, использующая аннотации JSR-330

Как было указано в главе 1, в JEE 6 предлагается поддержка спецификации JSR-330 ("Dependency Injection for Java" — "Внедрение зависимостей для Java"), которая представляет собой коллекцию аннотаций для выражения конфигурации DI приложения внутри контейнера JEE или другой совместимой инфраструктуры IoC. Платформа Spring также поддерживает и распознает эти аннотации, так что хотя вы не можете запускать приложение в контейнере JEE 6, все равно имеете возможность применять аннотации JSR-330 в рамках Spring. Использование аннотаций JSR-330 помогает упростить перенос приложения из Spring в контейнер JEE 6 или другой совместимый контейнер IoC (например, Google Guice).

Давайте снова обратимся к примеру поставщика и визуализатора сообщений и реализуем его с применением аннотаций JSR-330. Для поддержки аннотаций JSR-330 понадобится добавить к проекту зависимость, описанную в табл. 4.4.

**Таблица 4.4. Зависимость для поддержки JSR-330**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `javax.inject` | `javax.inject` | 1 | Стандартная библиотека JSR-330 |

В листинге 4.62 представлена реализация `MessageProvider` и `Configurable MessageProvider`.

**Листинг 4.62. Класс `ConfigurableMessageProvider` (JSR-330)**

```
package com.apress.prospring4.ch4;

public interface MessageProvider {
   String getMessage();
}
```

```
package com.apress.prospring4.ch4;

import javax.inject.Inject;
import javax.inject.Named;

@Named("messageProvider")
public class ConfigurableMessageProvider implements MessageProvider {
    private String message = "Default message";

    public ConfigurableMessageProvider() {
    }

    @Inject
    @Named("message")
    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    @Override
    public String getMessage() {
        return message;
    }
}
```

Вы заметите, что все аннотации принадлежат пакету `javax.inject`, который является реализацией стандарта JSR-330. В этом классе аннотация `@Named` используется в двух местах. Во-первых, она применяется для объявления внедряемого бина (аналогично аннотации `@Component` или `@Service` в Spring). В листинге аннотация `@Named("messageProvider")` указывает, что `ConfigurableMessageProvider` является внедряемым бином, и назначает ему имя `messageProvider`, т.е. делает то же самое, что и атрибут `name` дескриптора `<bean>` в Spring. Во-вторых, мы используем внедрение через конструктор, помещая аннотацию `@Inject` перед конструктором, который принимает строковое значение. Затем мы применяем аннотацию `@Named` для указания на то, что нужно внедрять значение, имеющее имя `message`. Взгляните на интерфейс `MessageRenderer` и реализацию `StandardOutMessageRenderer` в листинге 4.63.

**Листинг 4.63. Класс `StandardOutMessageRenderer` (JSR-330)**

```
package com.apress.prospring4.ch4;

public interface MessageRenderer {
    void render();
    void setMessageProvider(MessageProvider provider);
    MessageProvider getMessageProvider();
}

package com.apress.prospring4.ch4;

import javax.inject.Inject;
import javax.inject.Named;
import javax.inject.Singleton;

@Named("messageRenderer")
```

```
@Singleton
public class StandardOutMessageRenderer implements MessageRenderer {
   @Inject
   @Named("messageProvider")
   private MessageProvider messageProvider = null;

   @Override
   public void render() {
      if (messageProvider == null) {
         throw new RuntimeException(
            "You must set the property messageProvider of class:"
            + StandardOutMessageRenderer.class.getName());
      }
      System.out.println(messageProvider.getMessage());
   }
   @Override
   public void setMessageProvider(MessageProvider provider) {
      this.messageProvider = provider;
   }
   @Override
   public MessageProvider getMessageProvider() {
      return this.messageProvider;
   }
}
```

В листинге 4.63 мы используем аннотацию `@Named` для определения внедряемого бина. Обратите внимание на аннотацию `@Singleton`. Полезно отметить, что в стандарте JSR-330 по умолчанию бин является неодиночным, что похоже на область действия на уровне прототипа в Spring. Таким образом, если вы хотите, чтобы в среде JSR-330 ваш бин был одиночным, то должны применять аннотацию `@Singleton`. Однако использование этой аннотации в Spring не дает никакого эффекта, потому что стандартным режимом создания экземпляров бинов в Spring является одиночный. Мы поместили эту аннотацию только в демонстрационных целях, чтобы подчеркнуть отличие между Spring и другими контейнерами, совместимыми с JSR-330.

На этот раз в свойстве `messageProvider` мы применяем аннотацию `@Inject` для внедрения через метод установки и указываем, что для такого внедрения должен использоваться бин с именем `messageProvider`.

В листинге 4.64 определена простая XML-конфигурация Spring для приложения (`app-context-annotation.xml`).

**Листинг 4.64. XML-конфигурация Spring (JSR-330)**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context.xsd">
```

```
    <context:component-scan base-package="com.apress.prospring4.ch4"/>
    <bean id="message" class="java.lang.String">
        <constructor-arg value="You are running JSR330!"/>
    </bean>
</beans>
```

Для работы с JSR-330 не нужны какие-то специальные дескрипторы; просто сконфигурируйте свое приложение подобно обычному приложению Spring. Дескриптор `<context:component-scan>` заставляет Spring искать относящиеся к DI аннотации, и Spring распознает заданные аннотации JSR-330. Кроме того, здесь объявлен бин Spring по имени `message` для внедрения через конструктор в класс `ConfigurableMessageProvider`. В листинге 4.65 приведен код тестовой программы.

**Листинг 4.65. Тестирование примера использования JSR-330**

```
package com.apress.prospring4.ch4;
import org.springframework.context.support.GenericXmlApplicationContext;
public class Jsr330Example {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();
        MessageRenderer renderer
            = ctx.getBean("messageRenderer", MessageRenderer.class);
        renderer.render();
    }
}
```

Запуск этой программы приводит к генерации следующего вывода:

```
You are running JSR330!
```

За счет применения спецификации JSR-330 можно упростить переход на другие контейнеры IoC, совместимые с JSR-330 (например, серверы приложений, совместимые с JEE 6, или контейнеры DI наподобие Google Guice). Тем не менее, аннотации Spring обладают намного большими возможностями и гибкостью, чем аннотации JSR-330. Ниже описаны некоторые основные отличия между ними.

- При использовании аннотации `@Autowired` из Spring можно задавать атрибут `required`, указывающий на то, что DI должно быть выполнено (для объявления этого требования можно также применять аннотацию `@Required` из Spring), тогда как для аннотации `@Inject` из JSR-330 эквивалент отсутствует. Более того, Spring предоставляет аннотацию `@Qualifier`, которая обеспечивает более точное управление выполнением автосвязывания зависимостей на основе имени квалификатора.
- Спецификация JSR-330 поддерживает только одиночный и неодиночный режимы создания экземпляров бинов, в то время как Spring поддерживает большее число режимов, что очень полезно для веб-приложений.
- В Spring можно использовать аннотацию `@Lazy`, чтобы заставить Spring создавать экземпляры бина только при запрашивании приложением. В JSR-330 эквивалентная возможность отсутствует.

Аннотации Spring и JSR-330 можно также смешивать в одном приложении. Тем не менее, рекомендуется придерживаться какого-то одного вида аннотаций, чтобы обеспечить согласованный стиль приложения. Один из возможных подходов предусматривает применение аннотаций JSR-330, когда только возможно, и аннотаций Spring при необходимости. Однако это даст лишь незначительные выгоды, поскольку придется выполнять большой объем работы при переходе на другой контейнер DI.

В заключение отметим, что рекомендуется отдавать предпочтение аннотациям Spring перед аннотациями JSR-330, т.к. аннотации Spring являются более мощными, если только не существует требования относительно того, что приложение должно быть независимым от контейнеров IoC.

# Конфигурация, использующая Groovy

В версии Spring Framework 4.0 появилась возможность конфигурирования определений бинов и контекста `ApplicationContext` с применением языка Groovy. Это предоставляет разработчикам еще один вариант для либо замены, либо дополнения конфигурации бинов на основе XML и/или аннотаций. Контекст `ApplicationContext` может создаваться непосредственно в сценарии Groovy или загружаться из кода Java посредством класса `GenericGroovyApplicationContext` в обоих случаях. Для начала давайте детально рассмотрим способ создания определений бинов из внешнего сценария Groovy и последующей их загрузки в коде Java. В листинге 4.66 приведен пример POJO-класса `Contact`.

**Листинг 4.66. Пример POJO-класса `Contact`**

```
package com.apress.prospring4.ch4;

public class Contact {
    private String firstName;
    private String lastName;
    private int age;

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int getAge() {
        return age;
    }
```

```
    @Override
    public String toString() {
        return "First name: " + firstName + ", Last name: " + lastName
            + ", Age: " + age;
    }
}
```

Как видите, это всего лишь обычный Java-класс с парой свойств, касающихся контактной информации. Мы используем здесь простой Java-класс для демонстрации того, что конфигурирование бинов в Groovy вовсе не означает необходимость переписывания на языке Groovy всей кодовой базы. Имея класс, давайте займемся сценарием Groovy (`beans.groovy`), который будет применяться для создания определения бинов (листинг 4.67).

**Листинг 4.67. Внешний сценарий Groovy для конфигурирования бинов**

```
package com.apress.prospring4.ch4

beans {
    contact(Contact, firstName: 'Chris', lastName: 'Schaefer', age: 32)
}
```

Сценарий Groovy начинается с замыкания верхнего уровня по имени `beans`, которое предоставляет определения бинов платформе Spring. Первым делом мы указываем имя бина (`contact`), после чего в качестве аргументов передаем тип класса (`Contact`), за которым следуют имена свойств и значения для установки этих свойств. Теперь давайте создадим простую тестовую программу на Java, загружающую определения бинов из сценария Groovy, как показано в листинге 4.68.

**Листинг 4.68. Загрузка определений бинов в коде Java посредством сценария Groovy**

```
package com.apress.prospring4.ch4;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.GenericGroovyApplicationContext;

public class GroovyBeansFromJava {
    public static void main(String[] args) {
        ApplicationContext context
            = new GenericGroovyApplicationContext("classpath:beans.groovy");
        Contact contact = context.getBean("contact", Contact.class);
        System.out.println(contact);
    }
}
```

Здесь видно, что создание `ApplicationContext` осуществляется в обычной манере, но за счет использования класса `GenericGroovyApplicationContext` и предоставления ранее написанного сценария Groovy, который строит определения бинов.

Прежде чем можно будет запускать примеры, приведенные в этом разделе, понадобится добавить зависимость, указанную в табл. 4.5.

**Таблица 4.5. Зависимость для поддержки Groovy**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| org.codehaus.groovy | groovy-all | 2.2.2 | Библиотека Groovy |

Выполнение примера кода из листинга 4.68 дает следующий вывод:

```
First name: Chris, Last name: Schaefer, Age: 32
```

Теперь, когда вы увидели, как загружать определения бинов из кода Java через внешний сценарий Groovy, возникает вопрос: каким образом создать ApplicationContext и определения бинов только в сценарии Groovy? Давайте рассмотрим код Groovy (GroovyConfig.groovy), приведенный в листинге 4.69.

**Листинг 4.69. Создание ApplicationContext и определений бинов в Groovy**

```
package com.apress.prospring4.ch4

import org.springframework.context.support.GenericApplicationContext
import org.springframework.beans.factory.groovy.GroovyBeanDefinitionReader

def ctx = new GenericApplicationContext()
def reader = new GroovyBeanDefinitionReader(ctx)

reader.beans {
   contact(Contact, firstName: 'Chris', lastName: 'Schaefer', age: 32)
}

ctx.refresh()

println ctx.getBean("contact")
```

Запустив этот пример, мы получаем уже знакомый вывод:

```
First name: Chris, Last name: Schaefer, Age: 32
```

На этот раз мы создаем экземпляр типичного контекста GenericApplicationContext, но применяем GroovyBeanDefinitionReader, который будет использоваться для передачи определений бинов. Затем, как и предыдущем примере, мы создаем бин из нашего простого POJO-класса, обновляем контекст ApplicationContext и выводим строковое представление бина Contact. Как видите, проще уже некуда!

Понятно, что мы только слегка коснулись поверхности того, что можно делать с помощью поддержки Groovy в Spring 4. Поскольку в распоряжении разработчика имеется вся мощь языка Groovy, при создании определений бинов он может предпринимать множество интересных действий. Имея полный доступ к ApplicationContext, вы можете не только конфигурировать бины, но также работать с поддержкой профилей, файлами свойств и т.д. Просто помните, что с большой мощью приходит и высокая ответственность.

# Резюме

В этой главе вы ознакомились с широким спектром средств Spring, которые дополняют ключевые возможности IoC. Вы узнали, как подключаться к жизненному

циклу бина и делать его осведомленным о среде Spring. Были представлены фабрики бинов в качестве решения для включения IoC в обширный набор классов. Было также показано, как применять редакторы свойств для упрощения подготовки конфигурации приложения и устранения потребности в искусственных свойствах типа `String`. Кроме того, в главе было завершено подробное рассмотрение ряда дополнительных средств, предлагаемых `ApplicationContext`, среди которых i18n, публикация событий и доступ к ресурсам.

Мы также раскрыли такие возможности, как использование Java-классов и нового синтаксиса Groovy вместо XML-конфигурации, поддержка профилей, а также уровень абстракции среды и источников свойств. Наконец, мы обсудили применение стандартных аннотаций JSR-330 в Spring.

Итак, мы рассмотрели основные концепции платформы Spring Framework и ее средства как контейнера DI, а также другие службы, которые предоставляет ядро Spring Framework. В следующей главе и далее мы взглянем на использование Spring в специфичных областях, таких как АОП, доступ к данным, поддержка транзакций и поддержка веб-приложений.

ГЛАВА 5

# Введение в аспектно-ориентированное программирование с использованием Spring

Помимо внедрения зависимостей (Dependency Injection — DI), платформа Spring Framework предлагает поддержку аспектно-ориентированного программирования (АОП). На АОП часто ссылаются как на инструмент для реализации сквозной функциональности. Понятие *сквозная функциональность* имеет отношение к логике, которая не может быть отделена от остальной части приложения и в результате приводит к дублированию кода и тесной связанности. За счет использования АОП для разбиения на модули отдельных порций логики, которые называются *функциональностью*, ее можно применять ко многим частям приложения, не дублируя код и не создавая жесткие зависимости. Типичными примерами сквозной функциональности являются средства регистрации в журнале и обеспечения безопасности, которые присутствуют во многих приложениях. Представьте себе приложение, которое в целях отладки регистрирует в журнале начало и завершение каждого метода. Возможно, вы вынесете код регистрации в журнале в специальный класс, однако впоследствии для выполнения регистрации все равно понадобится вызывать методы этого класса по два раза для каждого метода приложения. Используя АОП, вы можете просто указать, что методы регистрирующего класса должны вызываться перед и после вызова каждого метода в приложении.

Важно понимать, что АОП дополняет объектно-ориентированное программирование (ООП), а не соперничает с ним. ООП очень хорошо подходит для решения широкого разнообразия задач, с которыми сталкиваются программисты. Однако если взглянуть на пример с регистрацией в журнале еще раз, то становится очевидным, что когда дело доходит до реализации сквозной функциональности в больших масштабах, возможностей ООП не хватает. Применение для разработки всего приложения одного лишь АОП практически нереально, учитывая, что функции АОП основаны на ООП. Аналогично, хотя вполне возможно строить полные приложения с использованием ООП, более разумный подход предусматривает применение АОП для решения определенных задач, связанных со сквозной функциональностью.

Ниже перечислены темы, которые рассматриваются в данной главе.

- **Основы АОП.** Прежде чем обсуждать реализацию АОП в Spring, мы раскроем основы АОП как технологии. Большинство концепций, описанных в этой части главы, не являются специфичными для Spring и могут быть найдены в любой реализации АОП. Если вы уже знакомы с другими реализациями АОП, можете спокойно пропустить эту часть.
- **Типы АОП.** Различают два типа АОП: статическое и динамическое. При статическом АОП, таком как предоставляемое механизмами связывания во время компиляции AspectJ (`http://eclipse.org/aspectj/`), сквозная функциональность применяется к коду на этапе компиляции, и ее нельзя изменить без модификации кода и повторной компиляции. При динамическом АОП, подобном АОП в Spring, сквозная функциональность применяется динамически во время выполнения. Это позволяет вносить изменения в конфигурацию АОП, не проводя повторно компиляцию приложения. Эти типы АОП дополняют друг друга и в случае совместного использования формируют мощное сочетание, которое можно применять в своих приложениях.
- **Архитектура АОП в Spring.** Реализация АОП в Spring представляет собой только подмножество полного набора функциональных средств АОП, доступного в других реализациях, таких как AspectJ. В этой части главы мы приведем высокоуровневый обзор средств, присутствующих в Spring, и особенностей их реализации, а также объясним, почему некоторые возможности исключены из реализации Spring.
- **Прокси АОП в Spring.** Прокси — это крупная часть рабочей среды АОП в Spring, и вы должны хорошо понимать их, чтобы извлечь максимум из этой реализации АОП. Мы рассмотрим два вида прокси: динамический прокси JDK и прокси CGLIB. В частности, мы представим различные сценарии использования каждого вида прокси, оценим производительность двух типов прокси и дадим ряд простых рекомендаций, следование которым в приложении позволит добиться максимально эффективной работы с АОП в Spring.
- **Использование АОП в Spring.** В этой части главы будут приведены некоторые практические примеры применения АОП. Мы начнем с простого примера "Hello World!", чтобы упростить понимание кода АОП, и продолжим детальным описанием функциональных средств АОП, доступных в Spring, сопровождая их подходящими примерами.
- **Расширенное использование срезов.** Мы исследуем классы `ComposablePointcut` и `ControlFlowPointcut`, введения и соответствующие технологии, которые вы должны задействовать в случае использования срезов в приложении.
- **Службы инфраструктуры АОП.** Платформа Spring полностью поддерживает конфигурирование АОП в прозрачной и декларативной манере. Мы рассмотрим три способа (класс `ProxyFactoryBean`, пространство имен `aop` и аннотации в стиле `@AspectJ`) внедрения декларативно определенных прокси АОП в объекты приложения в качестве коллабораторов, делая приложение полностью не осведомленным о том, что оно работает с объектами, которые снабжены советами.

- **Интеграция с AspectJ.** Расширение AspectJ является полнофункциональной реализацией АОП. Основное отличие между AspectJ и АОП в Spring заключается в том, что AspectJ применяет совет к целевым объектам через связывание (либо во время компиляции, либо во время выполнения), тогда как АОП в Spring основано на прокси. Набор функциональных средств AspectJ намного шире, чем в АОП в Spring, но им значительно сложнее пользоваться по сравнению со Spring. Расширение AspectJ будет подходящим решением, когда обнаруживается, что АОП в Spring не поддерживает средство, которое необходимо для приложения.

# Концепции АОП

Как и большинство технологий, АОП имеет собственный специфичный набор концепций и терминов, смысл которых важно понимать. В следующем списке приведены ключевые концепции АОП.

- **Точки соединения.** Точка соединения (joinpoint) — это четко определенная точка во время выполнения приложения. Типовые примеры точек соединения включают обращение к методу, собственно вызов метода (Method Invocation), инициализацию класса и создание экземпляра объекта. Точки соединения являются ключевой концепцией АОП и определяют места в приложении, в которые можно вставлять дополнительную логику с применением АОП.
- **Советы.** Фрагмент кода, который должен выполняться в отдельной точке соединения, представляет собой совет (advice), определенный методом в классе. Существует много типов советов, среди которых перед, когда совет выполняется до точки соединения, и после, когда совет выполняется после точки соединения.
- **Срезы.** Срез (pointcut) — это коллекция точек соединения, которая используется для определения ситуации, когда совет должен быть выполнен. Создавая срезы, вы получаете точный контроль над тем, как применять совет к компонентам приложения. Как упоминалось ранее, типичной точкой соединения является вызов метода или коллекция всех вызовов методов в отдельном классе. Часто между срезами можно устанавливать сложные отношения, чтобы дополнительно ограничить то, когда будет выполнен совет.
- **Аспекты.** Аспект (aspect) — это комбинация совета и срезов, инкапсулированных в классе. Такая комбинация дает в результате определение логики, которая должна быть включена в приложение, и указание мест, где она должна выполняться.
- **Связывание.** Связывание (weaving) представляет собой процесс вставки аспектов в определенную точку внутри кода приложения. Для решений АОП времени компиляции связывание обычно делается на этапе сборки. Подобным же образом для решений АОП времени выполнения связывание происходит динамически во время выполнения. В AspectJ поддерживается еще один механизм связывания, называемый связыванием во время загрузки (load-time weaving — LTW), при котором перехватывается лежащий в основе загрузчик классов JVM и обеспечивается связывание с байт-кодом, когда он загружается загрузчиком классов.

- **Цель.** Цель (target) — это объект, поток выполнения которого изменяется каким-то процессом АОП. На целевой объект часто ссылаются как на объект, снабженный советом.
- **Введение.** Введение (introduction) представляет собой процесс, посредством которого можно изменить структуру объекта за счет помещения в него дополнительных методов или полей. Введение АОП можно использовать для обеспечения реализации любым объектом определенного интерфейса без необходимости в том, чтобы класс этого объекта реализовывал такой интерфейс явно.

Не переживайте, если эти концепции показались вам запутанными; они станут яснее, когда мы представим ряд примеров. Кроме того, имейте в виду, что в реализации АОП, предлагаемой Spring, вам не придется сталкиваться напрямую со многими из перечисленных концепций, а из-за особенностей реализации АОП в Spring некоторые концепции не важны. В ходе материала главы мы обсудим все эти средства в контексте платформы Spring.

# Типы АОП

Как упоминалось ранее, различают два типа АОП: статическое и динамическое. Разница между ними касается точки, в которой происходит процесс связывания, а также того, каким образом этот процесс осуществляется.

## Использование статического АОП

При статическом АОП процесс связывания формирует еще один шаг внутри процесса сборки приложения. В терминах Java процесс связывания при статической реализации АОП предусматривает модификацию действительного байт-кода приложения, должным образом изменяя и расширяя код приложения. Это хороший способ выполнения процесса связывания, поскольку конечным результатом является просто байт-код Java, и нет необходимости предпринимать специальные трюки во время выполнения, чтобы определить, когда должен быть применен совет.

Недостаток такого механизма связан с тем, что любые изменения, вносимые в аспекты, даже если они касаются всего лишь добавления еще одной точки соединения, требуют перекомпиляции всего приложения. Прекрасным примером статической реализации АОП может служить связывание во время компиляции в AspectJ.

## Использование динамического АОП

Динамические реализации АОП, подобные АОП в Spring, отличаются от статических реализаций в том, что процесс связывания происходит динамически во время выполнения. Способ достижения этого зависит от реализации, но, как вы увидите, применяемый в Spring подход заключается в создании прокси для всех целевых объектов, позволяя совету вызываться требуемым образом. Недостаток динамического АОП связан с тем, что оно обычно не выполняется настолько хорошо, как статическое АОП, однако производительность неуклонно растет. Основным преимуществом применения динамических реализаций АОП является простота изменения целого набора аспектов в приложении без необходимости в повторной компиляции кода приложения.

### Выбор типа АОП

Выбор между статическим и динамическим АОП — довольно трудное решение. Обе реализации обладают своими достоинствами, и вы не ограничены использованием только какого-то одного типа АОП. В общем случае статические реализации АОП несколько длиннее и обычно обладают более широким набором функциональных средств и большим количеством доступных точек соединения. Как правило, если производительность особенно критична или нужно средство АОП, которое в Spring не реализовано, необходимо применять AspectJ. В большинстве других ситуаций АОП в Spring является идеальным выбором. Помните, что многие решения, основанные на АОП, такие как управление транзакциями, предлагаются Spring в готовом виде, поэтому проверьте наличие внутри платформы того или иного средства, прежде чем заняться его созданием самостоятельно.

Как обычно, позвольте требованиям приложения управлять выбором реализации АОП и не ограничивайтесь единственной реализацией, если для приложения больше подходит комбинация технологий. В целом реализация АОП в Spring отличается меньшей сложностью, чем AspectJ, поэтому она должна рассматриваться в качестве первого варианта.

## АОП в Spring

Реализацию АОП в Spring можно представлять как состоящую из двух логических частей. Первая часть — это ядро АОП, которое обеспечивает совершенно несвязанную, чисто программную функциональность АОП (также называемую *Spring AOP API*). Вторая часть реализации АОП — это набор служб платформы, которые упрощают использование АОП в приложениях. Сверх того другие компоненты Spring, такие как диспетчер транзакций и вспомогательные классы EJB, предоставляют основанные на АОП службы, упрощающие разработку приложений.

### Альянс АОП

Альянс АОП (`http://aopalliance.sourceforge.net/`) является результатом совместных усилий представителей многих проектов АОП с открытым кодом по определению стандартного набора интерфейсов для реализаций АОП. Там, где возможно, в Spring вместо определения собственных интерфейсов применяются интерфейсы, определенные Альянсом АОП. Это позволяет многократно использовать отдельный совет во множестве реализаций АОП, которые поддерживают интерфейсы от Альянса АОП.

### Пример "Hello World!" в АОП

Перед тем, как погрузиться в детали реализации АОП в Spring, давайте обратимся к примеру. Мы возьмем простой класс, выводящий сообщение "World", и с применением АОП трансформируем экземпляр этого класса во время выполнения, чтобы он выводил сообщение "Hello World!". Базовый класс `MessageWriter` показан в листинге 5.1.

**Листинг 5.1. Класс `MessageWriter`**

```
package com.apress.prospring4.ch5;

public class MessageWriter {
    public void writeMessage() {
        System.out.print("World");
    }
}
```

Имея реализованный метод вывода сообщения, давайте добавим к этому классу совет, чтобы `writeMessage()` взамен выводил "Hello World!".

Чтобы сделать это, нам необходимо перед выполнением существующего тела метода выполнить один код (для вывода строки "Hello "), а после выполнения тела — другой код (для вывода "!"). В терминах АОП нам требуется совет *вокруг*, который выполняется вокруг точки соединения. В данном случае точкой соединения является вызов метода `writeMessage()`. В листинге 5.2 приведен код класса `MessageDecorator`, действующего в качестве реализации совета "вокруг".

**Листинг 5.2. Реализация совета "вокруг"**

```
package com.apress.prospring4.ch5;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class MessageDecorator implements MethodInterceptor {
    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.print("Hello ");
        Object retVal = invocation.proceed();
        System.out.println("!");
        return retVal;
    }
}
```

Интерфейс `MethodInterceptor` — это стандартный интерфейс Альянса АОП для реализации совета "вокруг" для точек соединения вызовов методов. Объект `MethodInvocation` представляет вызов метода, снабжаемый советом, и с помощью этого объекта мы управляем тем, когда вызову метода разрешено продолжаться. Поскольку это совет "вокруг", мы способны выполнять действия перед вызовом метода и после его вызова, но до того, как произойдет возврат из метода.

В листинге 5.2 мы просто записываем в консольный вывод строку "Hello ", вызываем метод с помощью `MethodInvocation.proceed()` и после этого записываем в консольный вывод строку "!".

Финальный шаг этого примера заключается в связывании совета `MessageDecorator` (а точнее — метода `invoke()`) с кодом. Для этого мы создаем экземпляр `MessageWriter`, т.е. цель, и затем создаем прокси этого экземпляра, инструктируя фабрику прокси относительно связывания с советом `MessageDecorator`. В листинге 5.3 показан необходимый код.

**Листинг 5.3. Связывание с советом `MessageDecorator`**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.framework.ProxyFactory;

public class HelloWorldAOPExample {
   public static void main(String[] args) {
      MessageWriter target = new MessageWriter();

      ProxyFactory pf = new ProxyFactory();
      pf.addAdvice(new MessageDecorator());
      pf.setTarget(target);

      MessageWriter proxy = (MessageWriter) pf.getProxy();

      target.writeMessage();
      System.out.println("");
      proxy.writeMessage();
   }
}
```

Важным моментом, который следует отметить в коде, является использование класса `ProxyFactory` для создания прокси целевого объекта и одновременного его связывания с советом. Совет `MessageDecorator` передается в `ProxyFactory` с помощью вызова `addAdvice()`, а цель для связывания указывается посредством вызова `setTarget()`. После того, как цель установлена, и некоторый совет добавлен к `ProxyFactory`, с помощью вызова `getProxy()` мы генерируем прокси. Наконец, мы вызываем `writeMessage()` на исходном целевом объекте и на прокси-объекте.

Выполнение кода из листинга 5.3 дает в результате следующий вывод:

```
World
Hello World!
```

Как видите, вызов метода `writeMessage()` на незатронутом целевом объекте приводит к стандартному обращению без выдачи на консоль дополнительной информации. Однако при вызове прокси выполняется код в `MessageDecorator`, генерируя желаемый вывод сообщения "Hello World!". В приведенном примере целевой класс не имеет никаких зависимостей от Spring или от интерфейсов Альянса АОП; изящество АОП в Spring и, следовательно, в целом АОП заключается в том, что советом можно снабдить почти любой класс, даже если этот класс создавался без учета АОП. Единственное ограничение, по крайней мере, в АОП из Spring, связано с невозможностью добавления советов к финальным классам, поскольку они не могут быть переопределены и, таким образом, для них нельзя создавать прокси.

## Архитектура АОП в Spring

Ключевая архитектура АОП в Spring основана на прокси. Когда вы хотите создать экземпляр класса, снабженный советом, то должны использовать класс `ProxyFactory` для создания прокси этого экземпляра, первым делом предоставив `ProxyFactory` со всеми аспектами, которые необходимо связать с прокси. Применение `ProxyFactory` — это чисто программный подход к созданию прокси АОП. По большей части использовать такой подход в своем приложении не обязательно; вместо этого можно положиться на механизмы декларативной конфигура-

ции АОП, предоставляемые Spring (класс `ProxyFactoryBean`, пространство имен `aop` и аннотации в стиле `@AspectJ`), которые обеспечат декларативное создание прокси. Тем не менее, важно понимать, как работает создание прокси, поэтому сначала мы будем применять программный подход к созданию прокси, а затем приступим к исследованию декларативных конфигураций АОП в Spring.

Во время выполнения платформа Spring анализирует сквозную функциональность, определенную для бинов в `ApplicationContext`, и динамически генерирует прокси-бины (которые являются оболочками для лежащих в основе целевых бинов). Вместо обращения к целевому бину напрямую вызывающие объекты внедряют прокси-бин. Прокси-бин затем анализирует текущие условия (т.е. точку соединения, срез или совет) и соответствующим образом связывает подходящий совет. На рис. 5.1 показано высокоуровневое представление прокси АОП в Spring в действии.

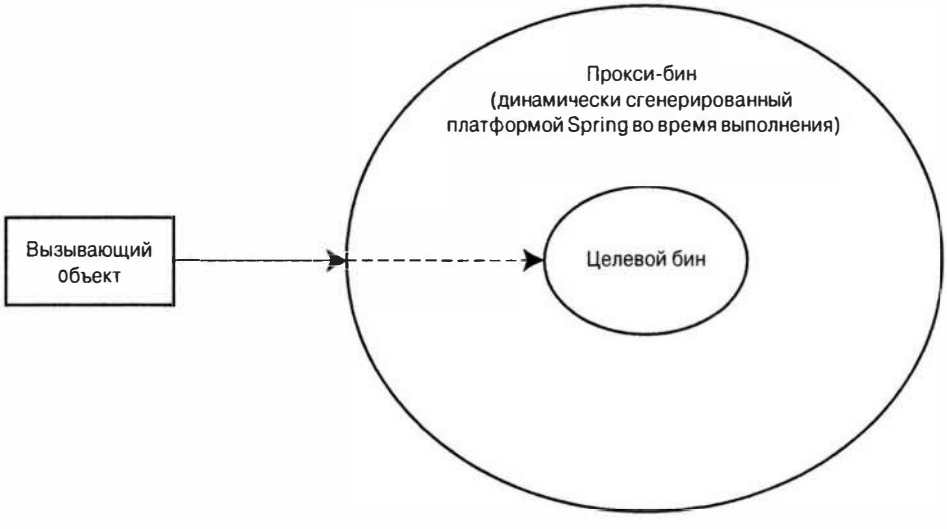


**Рис. 5.1.** Работа прокси АОП в Spring

Внутренне Spring поддерживает две реализации прокси: динамические прокси JDK и прокси CGLIB. По умолчанию, когда целевой объект, оснащаемый советом, реализует какой-то интерфейс, для создания экземпляров прокси целевого объекта Spring будет использовать динамический прокси JDK. Однако если целевой объект, оснащаемый советом, не реализует интерфейс (например, из-за того, что представляет собой конкретный класс), то для создания экземпляров прокси будет применяться CGLIB. Одна из основных причин заключается в том, что динамический прокси JDK поддерживает создание прокси только для интерфейсов. Прокси подробно обсуждаются в разделе "Что собой представляют прокси" далее в главе.

### *Точки соединения в Spring*

Одним из наиболее заметных упрощений АОП в Spring следует считать поддержку только одного типа точек соединения: вызов метода. На первый взгляд, это может показаться серьезным ограничением, если вы знакомы с другими реализациями АОП, такими как AspectJ, где поддерживается намного больше типов точек соединения, но в действительности это делает платформу Spring более доступной.

Точка соединения типа вызов метода (Method Invocation) на сегодняшний день является наиболее часто используемой из всех доступных точек соединения, и с ее применением можно решить многие из задач, которые делают АОП полезным в повседневной разработке. Не забывайте, что если нужно снабдить советом некоторый код в точке соединения, отличающейся от вызова метода, то всегда можно использовать Spring и AspectJ вместе.

### *Аспекты в Spring*

В реализации АОП в Spring аспект представляется экземпляром класса, который реализует интерфейс `Advisor`. Платформа Spring предлагает удобные реализации `Advisor`, которые можно применять в своих приложениях, устраняя необходимость в создании специальных реализаций `Advisor`. Существуют два подчиненных интерфейса `Advisor`: `IntroductionAdvisor` и `PointcutAdvisor`.

Интерфейс `PointcutAdvisor` реализован всеми реализациями `Advisor`, которые используют срезы для управления применением совета к точкам соединения. В Spring введение трактуется как специальный вид совета, и с помощью интерфейса `IntroductionAdvisor` вы можете управлять классами, к которым применяется введение.

Различные реализации `PointcutAdvisor` подробно обсуждаются в разделе "Советы и срезы в Spring" далее в этой главе.

## Класс `ProxyFactory`

Класс `ProxyFactory` управляет процессом связывания и создания прокси АОП в Spring. Прежде чем вы сможете создать прокси, вы должны указать снабженный советом или целевой объект. Как было показано ранее, это можно делать с использованием метода `setTarget()`. Внутренне `ProxyFactory` делегирует процесс создания прокси экземпляру `DefaultAopProxyFactory`, который, в свою очередь, делегирует его либо `Cglib2AopProxy`, либо `JdkDynamicAopProxy`, в зависимости от параметров приложения. Создание прокси более подробно рассматривается далее в этой главе.

Класс `ProxyFactory` предоставляет метод `addAdvice()`, который был задействован в листинге 5.3, для случаев, когда совет должен применяться к вызовам всех методов в классе, а не только к избранным. Внутри метод `addAdvice()` помещает переданный ему совет в экземпляр `DefaultPointcutAdvisor`, который является стандартной реализацией `PointcutAdvisor`, и конфигурирует его со срезом, по умолчанию включающим все методы. Если необходим дополнительный контроль над созданием `Advisor` или нужно добавить введение к прокси, создайте реализацию `Advisor` самостоятельно и используйте метод `addAdvisor()` класса `ProxyFactory`.

Один и тот же экземпляр `ProxyFactory` можно применять для создания множества прокси, каждый из которых имеет отличающийся аспект. Чтобы помочь в этом, в `ProxyFactory` предусмотрены методы `removeAdvice()` и `removeAdvisor()`, позволяющие удалять из `ProxyFactory` любой совет или реализации `Advisor`, которые ранее были добавлены. Для проверки, имеет ли `ProxyFactory` конкретный присоединенный к нему совет, вызовите метод `adviceIncluded()`, передав ему проверяемый объект совета.

## Создание совета в Spring

Платформа Spring поддерживает шесть разновидностей советов, которые описаны в табл. 5.1.

**Таблица 5.1. Типы советов в Spring**

| Название совета | Интерфейс | Описание |
|---|---|---|
| Перед (before) | `org.springframework.aop.MethodBeforeAdvice` | Используя совет "перед", можно осуществлять специальную обработку перед входом в точку соединения. Поскольку в Spring точка соединения — всегда вызов метода, по существу это позволяет реализовать предварительную обработку до выполнения метода. Совет "перед" имеет полный доступ к цели вызова метода, а также к аргументам, переданным методу, но не обладает никаким контролем над выполнением самого метода. В случае если совет "перед" генерирует исключение, дальнейшее выполнение цепочки перехватчиков (а также целевого метода) прекращается, и исключение распространяется обратно по цепочке перехватчиков |
| После возврата (after returning) | `org.springframework.aop.AfterReturningAdvice` | Совет "после возврата" выполняется после завершения выполнения вызова метода в точке соединения и возврата значения. Совет "после возврата" имеет доступ к цели вызова метода, к аргументам, переданным методу, а также к возвращаемому значению. Поскольку когда вызывается совет этого типа, метод уже выполнен, совет не имеет никакого контроля над вызовом метода. Если целевой метод генерирует исключение, совет "после возврата" выполняться не будет, а исключение распространится вверх по стеку вызовов обычным образом |
| После (after (finally)) | `org.springframework.aop.AfterAdvice` | Совет "после возврата" выполняется только в случае нормального завершения метода, снабженного советом. Однако совет "после" будет выполняться вне зависимости от результата метода, снабженного этим советом. Совет данного типа выполняется, даже когда метод, снабженный советом, дает сбой или когда генерируется исключение |
| Вокруг (around) | `org.aopalliance.intercept.MethodInterceptor` | В Spring совет "вокруг" моделируется с использованием стандарта Альянса АОП для перехватчика метода. Совету разрешено выполняться перед и после вызова метода, и есть возможность управления точкой, в которой вызов метода может быть продолжен. При необходимости можно вообще пропустить выполнение метода, предоставив собственную реализацию его логики |

*Окончание табл. 5.1*

| Название совета | Интерфейс | Описание |
|---|---|---|
| Перехват (throws) | `org.springframework.aop.ThrowsAdvice` | Совет "перехват" выполняется после возврата из вызова метода, но только в случае, если во время вызова было сгенерировано исключение. Совет этого типа может перехватывать только специфичные исключения, и тогда возможен доступ к методу, сгенерировавшему исключение, к аргументам, переданным вызову, и к цели вызова |
| Введение (introduction) | `org.springframework.aop.IntroductionInterceptor` | Платформа Spring моделирует введения как специальные типы перехватчиков. Используя перехватчик введения, можно указать реализацию методов, которые должны быть введены советом |

### *Интерфейсы для совета*

Вспомните из предыдущего обсуждения класса `ProxyFactory`, что совет добавляется к прокси либо прямо с использованием метода `addAdvice()`, либо косвенно с применением `Advisor` посредством метода `addAdvisor()`. Основная разница между `Advice` и `Advisor` заключается в том, что `Advisor` содержит в себе `Advice` и связанный `Pointcut`, что обеспечивает более точный контроль над тем, какие точки соединения `Advice` будет перехватывать. В Spring предусмотрена четко определенная иерархия интерфейсов советов. Эта иерархия основана на интерфейсах Альянса АОП и показана на рис. 5.2.

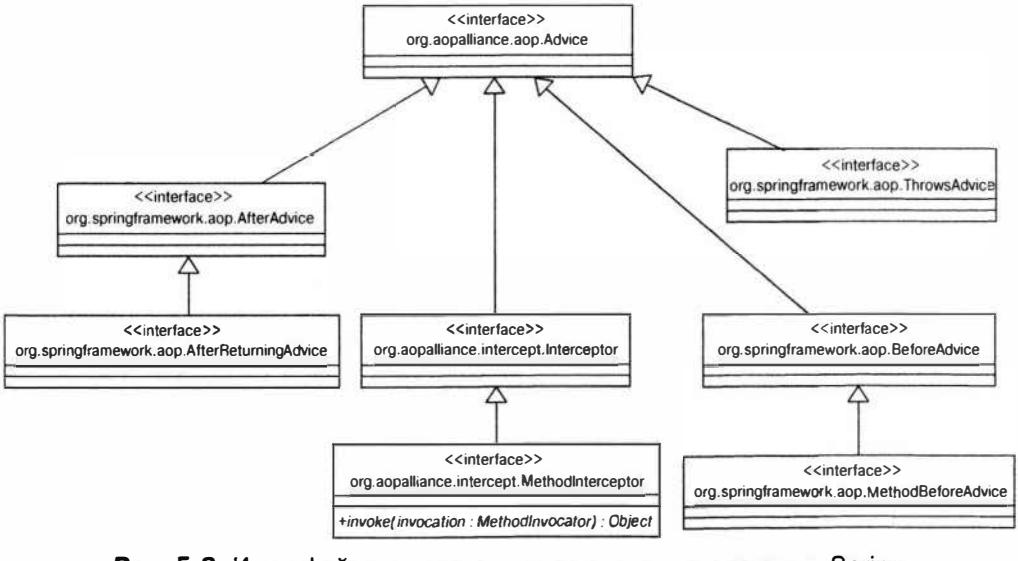


**Рис. 5.2.** Интерфейсы, предназначенные для типов советов Spring

Такая иерархия обладает не только преимуществом своей объектно-ориентированной природы, но также и возможностью работы с типами советов обобщенным образом, например, используя единственный метод `addAdvice()` класса `ProxyFactory`, и простого добавления новых типов советов без необходимости в модификации класса `ProxyFactory`.

### Создание совета "перед"

Совет "перед" является одним из наиболее часто применяемых типов советов, доступных в Spring. Этот совет может изменять аргументы, переданные методу, и предотвращать выполнение метода путем генерации исключения. В данном разделе мы покажем два простых примера использования совета "перед": для записи в консольный вывод сообщения, содержащего имя метода, перед его выполнением, и для ограничения доступа к методам объекта.

В листинге 5.4 показан код класса `SimpleBeforeAdvice`.

**Листинг 5.4. Класс `SimpleBeforeAdvice`**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;
import org.springframework.aop.framework.ProxyFactory;

public class SimpleBeforeAdvice implements MethodBeforeAdvice {
    public static void main(String[] args) {
        MessageWriter target = new MessageWriter();

        ProxyFactory pf = new ProxyFactory();
        pf.addAdvice(new SimpleBeforeAdvice());
        pf.setTarget(target);

        MessageWriter proxy = (MessageWriter) pf.getProxy();
        proxy.writeMessage();
    }

    @Override
    public void before(Method method, Object[] args, Object target)
            throws Throwable {
        System.out.println("Before method: " + method.getName());
    }
}
```

В этом коде мы снабжаем советом созданный ранее экземпляр класса `MessageWriter` с помощью экземпляра класса `SimpleBeforeAdvice`. В интерфейсе `MethodBeforeAdvice`, который реализует класс `SimpleBeforeAdvice`, определен единственный метод `before()`, вызываемый инфраструктурой АОП перед запуском метода в точке соединения. Вспомните, что в данный момент мы используем срез по умолчанию, предоставляемый методом `addAdvice()`, который соответствует всем методам класса. Методу `before()` передаются три аргумента: метод для запуска, аргументы, которые будут переданы этому методу, и экземпляр `Object`, служащий целью вызова. Класс `SimpleBeforeAdvice` применяет аргумент `Method` метода `before()` для записи в консольный вывод сообщения, которое содержит имя запускаемого метода.

В результате выполнения этого примера генерируется следующий вывод:

```
Before method: writeMessage
World
```

Как видите, вывод, полученный из вызова `writeMessage()`, присутствует, но перед ним находится вывод, сгенерированный `SimpleBeforeAdvice`.

### *Защита доступа к методам с использованием совета "перед"*

В этом разделе мы собираемся реализовать совет "перед", который проверяет учетные данные пользователя перед тем, как разрешить вызову метода продолжаться. Если учетные данные пользователя некорректны, совет генерирует исключение, предотвращая выполнение метода. Пример, рассматриваемый в данном разделе, несколько упрощен. Он позволяет пользователям аутентифицироваться с произвольным паролем и предоставляет доступ к защищенным методам только одному жестко закодированному пользователю. Тем не менее, пример демонстрирует простоту применения АОП для реализации сквозной функциональности, такой как средства безопасности.

---

**На заметку!** Это всего лишь пример демонстрации использования совета "перед". Всеобъемлющая поддержка защиты запуска методов, относящихся к бинам Spring, уже предлагается проектом Spring Security, поэтому самостоятельно реализовывать такие средства не придется.

---

В листинге 5.5 показан код класса `SecureBean`. Этот класс будет защищен с применением АОП.

**Листинг 5.5. Класс `SecureBean`**

```
package com.apress.prospring4.ch5;

public class SecureBean {
    public void writeSecureMessage() {
        System.out.println("Every time I learn something new, "
            + "it pushes some old stuff out of my brain");
    }
}
```

Класс `SecureBean` отображает только один перл премудростей Гомера Симпсона, который мы не хотим показывать абсолютно всем. Поскольку в этом примере требуется аутентификация пользователей, так или иначе, мы должны предусмотреть хранение ряда деталей. В листинге 5.6 приведен код класса `UserInfo`, предназначенного для хранения учетных данных пользователя.

**Листинг 5.6. Класс `UserInfo`**

```
package com.apress.prospring4.ch5;

public class UserInfo {
    private String userName;
    private String password;
```

```
    public UserInfo(String userName, String password) {
        this.userName = userName;
        this.password = password;
    }
    public String getPassword() {
        return password;
    }
    public String getUserName() {
        return userName;
    }
}
```

Этот класс просто хранит данные о пользователе, с которыми впоследствии можно делать что-нибудь полезное. В листинге 5.7 представлен код класса `SecurityManager`, отвечающего за аутентификацию пользователей и сохранение их учетных данных с целью извлечения в будущем.

**Листинг 5.7. Класс `SecurityManager`**

```
package com.apress.prospring4.ch5;

public class SecurityManager {
    private static ThreadLocal<UserInfo> threadLocal =
        new ThreadLocal<UserInfo>();

    public void login(String userName, String password) {
        threadLocal.set(new UserInfo(userName, password));
    }

    public void logout() {
        threadLocal.set(null);
    }

    public UserInfo getLoggedOnUser() {
        return threadLocal.get();
    }
}
```

Приложение использует класс `SecurityManager` для аутентификации пользователя и последующего извлечения деталей, связанных с аутентифицированным пользователем. Пользователь аутентифицируется с помощью метода `login()`. В реальном приложении метод `login()` мог бы проверять учетные данные по базе данных или каталогу LDAP, но здесь мы предполагаем, что все пользователи аутентифицированы. Метод `login()` создает для пользователя объект `UserInfo` и сохраняет его в текущем потоке с применением `ThreadLocal`. Метод `logout()` устанавливает в `null` любые значения, которые могут быть сохранены в `ThreadLocal`. Наконец, метод `getLoggedOnUser()` возвращает объект `UserInfo` для текущего аутентифицированного пользователя. Если аутентифицированные пользователи отсутствуют, этот метод возвращает `null`.

Чтобы проверить, аутентифицирован ли пользователь, и если это так, то разрешен ли ему доступ к методам SecureBean, мы должны создать совет, который выполняется перед методом и сравнивает объект UserInfo, возвращенный SecurityManager.getLoggedOnUser(), с набором учетных данных для разрешенных пользователей. Код класса SecurityAdvice, представляющего этот совет, показан в листинге 5.8.

**Листинг 5.8. Класс SecurityAdvice**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class SecurityAdvice implements MethodBeforeAdvice {
    private SecurityManager securityManager;

    public SecurityAdvice() {
        this.securityManager = new SecurityManager();
    }

    @Override
    public void before(Method method, Object[] args, Object target)
            throws Throwable {
        UserInfo user = securityManager.getLoggedOnUser();

        if (user == null) {
            System.out.println("No user authenticated");
            throw new SecurityException(
                "You must login before attempting to invoke the method: "
                + method.getName());
        } else if ("chris".equals(user.getUserName())) {
            System.out.println("Logged in user is chris - OKAY!");
        } else {
            System.out.println("Logged in user is " + user.getUserName()
                + " NOT GOOD :(");
            throw new SecurityException("User " + user.getUserName()
                + " is not allowed access to method " + method.getName());
        }
    }
}
```

Класс SecurityAdvice создает экземпляр SecurityManager в своем конструкторе и затем сохраняет этот экземпляр в поле. Следует отметить, что приложение и SecurityAdvice не нуждаются в совместном использовании одного и того же экземпляра SecurityManager, т.к. все данные сохранены в текущем потоке с использованием ThreadLocal. В методе before() мы предпринимаем простую проверку, является ли chris именем аутентифицированного пользователя. Если это так, мы предоставляем пользователю доступ, а в противном случае генерируем исключение. Также обратите внимание на то, что объект UserInfo проверяется на равенство значению null, которое означает, что текущий пользователь не аутентифицирован.

В листинге 5.9 приведен пример приложения, в котором применяется класс SecurityAdvice для защиты класса SecureBean.

**Листинг 5.9. Класс `SecurityExample`**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.framework.ProxyFactory;

public class SecurityExample {
    public static void main(String[] args) {
        SecurityManager mgr = new SecurityManager();

        SecureBean bean = getSecureBean();

        mgr.login("chris", "pwd");
        bean.writeSecureMessage();
        mgr.logout();

        try {
            mgr.login("invaliduser", "pwd");
            bean.writeSecureMessage();
        } catch(SecurityException ex) {
            System.out.println("Exception Caught: " + ex.getMessage());
        } finally {
            mgr.logout();
        }

        try {
            bean.writeSecureMessage();
        } catch(SecurityException ex) {
            System.out.println("Exception Caught: " + ex.getMessage());
        }
    }

    private static SecureBean getSecureBean() {
        SecureBean target = new SecureBean();

        SecurityAdvice advice = new SecurityAdvice();

        ProxyFactory factory = new ProxyFactory();
        factory.setTarget(target);
        factory.addAdvice(advice);

        SecureBean proxy = (SecureBean)factory.getProxy();

        return proxy;
    }
}
```

В методе `getSecureBean()` мы создаем прокси для класса `SecureBean`, который снабжается советом с применением экземпляра `SecurityAdvice`. Этот прокси возвращается вызывающему объекту. Когда вызывающий объект обращается к любому методу на этом прокси, данный вызов сначала направляется экземпляру `SecurityAdvice` для проведения проверки, связанной с безопасностью. В методе `main()` мы тестируем три сценария, вызывая метод `SecureBean.writeSecureMessage()` с двумя наборами пользовательских учетных данных и затем без учетных данных. Поскольку `SecurityAdvice` разрешает вызовам методов продолжаться, только если текущим аутентифицированным пользователем является `chris`, мы ожидаем, что единственным успешным сценарием в листинге 5.9 будет первый. Запуск этого примера дает следующий вывод:

```
Logged in user is chris - OKAY!
Every time I learn something new, it pushes some old stuff out of my brain
Logged in user is invaliduser NOT GOOD :(

Exception Caught: User invaliduser is not allowed access to method
writeSecureMessage
No user authenticated

Перехвачено исключение: Пользователю invaliduser не разрешен доступ к
методу writeSecureMessage
Аутентифицированные пользователи отсутствуют

Exception Caught: You must login before attempting to invoke the method:
writeSecureMessage

Перехвачено исключение: Вы должны войти, прежде чем пытаться вызвать
метод: writeSecureMessage
```

Как видите, только первому вызову `SecureBean.writeSecureMessage()` было разрешено продолжаться. Остальные вызовы были прекращены по исключению `SecurityException`, сгенерированному `SecurityAdvice`.

Несмотря на простоту, этот пример подчеркивает полезность совета "перед". Безопасность — типичный пример совета "перед", но мы также находим его удобным в сценарии, который требует модификации аргументов, передаваемых методу.

### *Создание совета "после возврата"*

Совет "после возврата" выполняется после того, как произошел возврат из вызова метода в точке соединения. Учитывая, что метод уже выполнен, переданные ему аргументы модифицировать невозможно. Хотя эти аргументы можно прочитать, путь выполнения изменить нельзя, равно как и предотвратить само выполнение метода. Это вполне ожидаемые ограничения; однако, несколько неожиданным является тот факт, что в совете "после возврата" невозможно модифицировать возвращаемое значение. При использовании совета "после возврата" вы ограничены только дополнительной обработкой. Несмотря на то что совет "после возврата" не позволяет изменять возвращаемое значение вызова метода, можно сгенерировать исключение, которое будет передано вверх по стеку вместо возвращаемого значения.

В этом разделе мы рассмотрим два примера применения совета "после возврата" в приложении. В первом примере после вызова метода просто осуществляется запись сообщения в консольный вывод. Во втором примере показано, как можно использовать совет "после возврата" для добавления к методу проверки ошибок. Предположим, что имеется класс `KeyGenerator`, который генерирует ключи для криптографических целей. Многие криптографические алгоритмы страдают из-за проблемы, заключающейся в том, что небольшое количество ключей считаются слабыми. *Слабый ключ* — это любой ключ, характеристики которого значительно упрощают выведение исходного сообщения, не зная этот ключ. Для алгоритма DES всего существуют 256 возможных ключей. В этом пространстве ключей 4 ключа считаются слабыми, а еще 12 ключей — полуслабыми. Хотя шансы получить один из этих ключей во время их случайной генерации невелики (1 из 252), проверка ключей настолько проста, что выполнять ее, безусловно, полезно. Во втором примере этого раздела мы построим совет "после возврата", который проверяет ключи, сгенерированные `KeyGenerator`, на предмет слабости и генерирует исключение при обнаружении такого ключа.

**На заметку!** Для получения более полных сведений о слабых ключах и криптографии в целом рекомендуем обратиться к книге *Applied Cryptography* Брюса Шнайера (Wiley, 1996 г.).

В листинге 5.10 приведен код класса `SimpleAfterReturningAdvice`, который демонстрирует применение совета "после возврата" путем записи сообщения в консольный вывод после возвращения из метода.

**Листинг 5.10. Класс `SimpleAfterReturningAdvice`**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;
import org.springframework.aop.framework.ProxyFactory;

public class SimpleAfterReturningAdvice implements AfterReturningAdvice {
    public static void main(String[] args) {
        MessageWriter target = new MessageWriter();

        ProxyFactory pf = new ProxyFactory();

        pf.addAdvice(new SimpleAfterReturningAdvice());
        pf.setTarget(target);

        MessageWriter proxy = (MessageWriter) pf.getProxy();
        proxy.writeMessage();
    }

    @Override
    public void afterReturning(Object returnValue, Method method,
            Object[] args, Object target) throws Throwable {
        System.out.println("");
        System.out.println("After method: " + method.getName());
    }
}
```

Обратите внимание, что в интерфейсе `AfterReturningAdvice` объявлен единственный метод `afterReturning()`, которому передается возвращаемое значение из вызова метода, ссылка на вызванный метод, аргументы, переданные этому методу, и цель вызова. Запуск этого примера дает в результате следующий вывод:

```
World
After method: writeMessage
```

Вывод очень похож на результат, полученный в примере с советом "перед", за исключением того, что сообщение, записываемое советом, находится после сообщения из метода `writeMessage()`.

Совет "после возврата" удобно применять при проведении дополнительной проверки ошибок, когда это возможно, для метода, возвращающего недопустимое значение. В описанном ранее сценарии генератор криптографических ключей может выдать ключ, который считается слабым для определенного алгоритма. В идеальном случае генератор ключей должен самостоятельно проверять ключи на предмет слабости, но поскольку шансы получения слабых ключей очень малы, многие генераторы такой проверки не содержат. Используя совет "после возврата", мы можем обеспечить эту дополнительную проверку для метода генерации ключей.

В листинге 5.11 показан очень простой генератор ключей.

**Листинг 5.11. Класс KeyGenerator**

```
package com.apress.prospring4.ch5;

import java.util.Random;

public class KeyGenerator {
   protected static final long WEAK_KEY = 0xFFFFFFF0000000L;
   protected static final long STRONG_KEY = 0xACDF03F590AE56L;

   private Random rand = new Random();

   public long getKey() {
      int x = rand.nextInt(3);

      if (x == 1) {
         return WEAK_KEY;
      }

      return STRONG_KEY;
   }
}
```

Этот генератор ключей не должен считаться защищенным. Он умышленно сделан простым для данного примера, и в одном случае из трех генерирует слабый ключ. В листинге 5.12 приведен код класса `WeakKeyCheckAdvice`, который проверяет, не является ли результат, возвращаемый методом `getKey()`, слабым ключом.

**Листинг 5.12. Проверка на слабые ключи**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;

public class WeakKeyCheckAdvice implements AfterReturningAdvice {
   @Override
   public void afterReturning(Object returnValue, Method method,
         Object[] args,Object target) throws Throwable {

      if ((target instanceof KeyGenerator)
            && ("getKey".equals(method.getName()))) {
         long key = ((Long) returnValue).longValue();

         if (key == KeyGenerator.WEAK_KEY) {
            throw new SecurityException(
               "Key Generator generated a weak key. Try again");
               // Генератор выдал слабый ключ. Требуется повторный вызов.
         }
      }
   }
}
```

В методе `afterReturning()` мы сначала проверяем, является ли метод, который был выполнен в точке соединения, методом `getKey()`. Если это так, мы проверяем результирующее значение на предмет слабого ключа. Если обнаружива-

ется, что результат метода `getKey()` — слабый ключ, мы генерируем исключение `SecurityException`, чтобы информировать об этом вызывающий код. В листинге 5.13 показано простое приложение, демонстрирующее использование этого совета.

**Листинг 5.13. Тестирование класса `WeakKeyCheckAdvice`**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.framework.ProxyFactory;

public class AfterAdviceExample {
    private static KeyGenerator getKeyGenerator() {
        KeyGenerator target = new KeyGenerator();

        ProxyFactory factory = new ProxyFactory();
        factory.setTarget(target);
        factory.addAdvice(new WeakKeyCheckAdvice());

        return (KeyGenerator)factory.getProxy();
    }

    public static void main(String[] args) {
        KeyGenerator keyGen = getKeyGenerator();
        for(int x = 0; x < 10; x++) {
            try {
                long key = keyGen.getKey();
                System.out.println("Key: " + key);
            } catch(SecurityException ex) {
                // Сгенерирован слабый ключ
                System.out.println("Weak Key Generated!");
            }
        }
    }
}
```

После создания снабженного советом прокси для цели `KeyGenerator` класс `AfterAdviceExample` пытается сгенерировать десять ключей. Если во время одиночной генерации произошло исключение `SecurityException`, на консоль выводится сообщение, информирующее пользователя о том, что был получен слабый ключ; иначе отображается сгенерированный ключ. Однократный запуск этого приложения на нашей машине дал следующий вывод:

```
Key: 48658904092028502
Key: 48658904092028502
Key: 48658904092028502
Weak Key Generated!
Key: 48658904092028502
Key: 48658904092028502
Key: 48658904092028502
Weak Key Generated!
Key: 48658904092028502
Key: 48658904092028502
```

Вывод показывает, что класс `KeyGenerator`, как и ожидалось, временами генерирует слабые ключи, а `WeakKeyCheckAdvice` гарантирует, что при каждом обнаружении слабого ключа будет инициировано исключение `SecurityException`.

### Создание совета "вокруг"

Совет "вокруг" функционирует подобно комбинации советов "перед" и "после", но с одним большим отличием — имеется возможность модифицировать возвращаемое значение. Кроме того, можно предотвратить действительное выполнение метода. Это значит, что за счет применения совета "вокруг" по существу можно заменить всю реализацию метода новым кодом. Совет "вокруг" в Spring моделируется как перехватчик с использованием интерфейса `MethodInterceptor`. Существует множество применений совета данного типа, и вы обнаружите, что многие функциональные средства Spring, такие как поддержка удаленных прокси и управление транзакциями, реализованы с участием перехватчиков методов. Перехватчики методов также являются удобным механизмом для построения профилей выполнения приложений, и они формируют основу примеров в этом разделе.

Мы не собираемся строить простой пример для перехвата метода, а сошлемся на первый пример в листинге 5.2, который демонстрирует использование базового перехватчика метода для записи сообщения с обеих сторон вызова метода. Обратите внимание на то, что метод `invoke()` класса `MethodInterceptor` не имеет такой же набор аргументов, как у соответствующих методов в интерфейсах `MethodBeforeAdvice` и `AfterReturningAdvice`, т.е. ему не передаются цель вызова, вызываемый метод и аргументы для вызываемого метода. Тем не менее, доступ к этим данным можно получить посредством объекта `MethodInvocation`, который передается `invoke()`. Все сказанное иллюстрируется в следующем примере.

В этом примере мы хотим получить базовую информацию о производительности методов класса, в частности, узнать, сколько времени выполняется тот или иной метод. Для достижения этой цели мы воспользуемся классом `StopWatch`, включенным в Spring, и нам, безусловно, нужен класс, реализующий `MethodInterceptor`, т.к. необходимо запустить `StopWatch` перед вызовом метода и остановить `StopWatch` сразу после вызова.

В листинге 5.14 показан код класса `WorkerBean`, который мы собираемся профилировать с применением класса `StopWatch` и совета "вокруг".

**Листинг 5.14. Класс `WorkerBean`**

```
package com.apress.prospring4.ch5;

public class WorkerBean {
    public void doSomeWork(int noOfTimes) {
        for(int x = 0; x < noOfTimes; x++) {
            work();
        }
    }

    private void work() {
        System.out.print("");
    }
}
```

Как видите, класс `WorkerBean` довольно прост. Метод `doSomeWork()` принимает единственный аргумент `noOfTimes` и вызывает метод `work()` указанное в `noOfTimes` количество раз. Метод `work()` просто содержит фиктивный вызов

System.out.print() с передачей ему пустой строки. Это предотвращает проведение компилятором оптимизации, заключающейся в отбрасывании кода метода work() и, следовательно, его вызова.

В листинге 5.15 приведен код класса ProfilingInterceptor, который использует класс StopWatch для получения времени вызова метода. Этот перехватчик применяется для профилирования класса WorkerBean, показанного в листинге 5.14.

**Листинг 5.15. Класс ProfilingInterceptor**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.springframework.util.StopWatch;

public class ProfilingInterceptor implements MethodInterceptor {
    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        StopWatch sw = new StopWatch();
        sw.start(invocation.getMethod().getName());

        Object returnValue = invocation.proceed();

        sw.stop();
        dumpInfo(invocation, sw.getTotalTimeMillis());
        return returnValue;
    }

    private void dumpInfo(MethodInvocation invocation, long ms) {
        Method m = invocation.getMethod();
        Object target = invocation.getThis();
        Object[] args = invocation.getArguments();

        System.out.println("Executed method: " + m.getName());
                           // выполняемый метод
        System.out.println("On object of type: " +
              target.getClass().getName());        // класс объекта

        System.out.println("With arguments:");     // аргументы
        for (int x = 0; x < args.length; x++) {
            System.out.print("      > " + args[x]);
        }
        System.out.print("\n");

        System.out.println("Took: " + ms + " ms");  // время выполнения
    }
}
```

В методе invoke(), который является единственным методом интерфейса MethodInterceptor, мы создаем экземпляр StopWatch и сразу же запускаем его, а затем позволяем вызову метода продолжаться с помощью MethodInvocation.proceed(). После завершения вызова метода и получения возвращаемого значения мы останавливаем StopWatch и передаем общее количество миллисекунд вместе с объектом MethodInvocation методу dumpInfo(). Наконец, мы возвращаем экземпляр Object, полученный из MethodInvocation.proceed(), чтобы вызывающий

код мог извлечь корректное возвращаемое значение. В этом случае мы не хотим разрушать стек вызовов каким-либо образом; код просто действует как перехватчик вызова метода. При желании можно было бы изменить стек вызовов полностью, переадресовав вызов метода другому объекту или удаленной службе, либо просто заменить реализацию логики метода внутри перехватчика и возвратить другое значение.

Метод `dumpInfo()` просто записывает в консольный вывод некоторую информацию о вызове метода, а также время, затраченное на выполнение метода. В первых трех строках `dumpInfo()` видно, как можно использовать объект `MethodInvocation` для выяснения метода, который был вызван, исходной цели вызова и переданных аргументов.

В листинге 5.16 приведен код класса `ProfilingExample`, который сначала снабжает советом экземпляр `WorkerBean` с помощью `ProfilingInterceptor` и затем профилирует метод `doSomeWork()`.

**Листинг 5.16. Класс `ProfilingExample`**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.framework.ProxyFactory;

public class ProfilingExample {
    public static void main(String[] args) {
        WorkerBean bean = getWorkerBean();
        bean.doSomeWork(10000000);
    }

    private static WorkerBean getWorkerBean() {
        WorkerBean target = new WorkerBean();

        ProxyFactory factory = new ProxyFactory();
        factory.setTarget(target);
        factory.addAdvice(new ProfilingInterceptor());

        return (WorkerBean)factory.getProxy();
    }
}
```

Запуск этого примера на нашей машине дал следующий вывод:

```
Executed method: doSomeWork
On object of type: com.apress.prospring4.ch5.WorkerBean
With arguments:
   > 10000000
Took: 477 ms
```

Вывод позволяет видеть, какой метод был выполнен, какой класс служил целью, какие аргументы передавались, и сколько времени заняло выполнение.

### *Создание совета "перехват"*

Совет "перехват" похож на совет "после возврата" тем, что выполняется после точки соединения, которая всегда является вызовом метода, но совет "перехват" инициируется, только если метод генерирует исключение. Совет "перехват" также подобен совету "после возврата" в том, что он имеет небольшой контроль над выполнением программы. При использовании совета "перехват" нельзя проигнориро-

вать возникшее исключение и взамен возвратить какое-то значение. Единственная модификация, которую можно внести в поток управления программы, заключается в изменении типа сгенерированного исключения. Это довольно мощная концепция, которая существенно упрощает разработку приложений. Представьте себе ситуацию, когда имеется API-интерфейс, который генерирует целое множество плохо определенных исключений. С применением совета "перехват" вы можете снабдить им все классы в этом API-интерфейсе и превратить иерархию исключений во что-то более управляемое и описательное. Разумеется, совет "перехват" можно также использовать для обеспечения централизованной регистрации ошибок в приложении, снижая объем кода регистрации ошибок, который разбросан по всему приложению.

Как было показано на рис. 5.2, совет "перехват" реализован интерфейсом `ThrowsAdvice`. В отличие от интерфейсов, с которыми вы сталкивались до сих пор, в `ThrowsAdvice` не определено ни одного метода; он просто является маркерным интерфейсом, применяемым платформой Spring. Причина подобного решения состоит в том, что в Spring разрешен типизированный совет "перехват", позволяющий определять точные типы исключений, которые совет должен перехватывать. Это достигается в Spring обнаружением методов с заданными сигнатурами с помощью рефлексии. Платформа Spring ищет две отличающихся сигнатуры методов. Сказанное лучше всего продемонстрировать на простом примере. В листинге 5.17 представлен простой бин с двумя методами, которые генерируют исключения разных типов.

**Листинг 5.17. Класс `ErrorBean`**

```
package com.apress.prospring4.ch5;

public class ErrorBean {
    public void errorProneMethod() throws Exception {
        throw new Exception("Foo");
    }

    public void otherErrorProneMethod() throws IllegalArgumentException {
        throw new IllegalArgumentException("Bar");
    }
}
```

В листинге 5.18 показан код класса `SimpleThrowsAdvice`, отображающего обе сигнатуры методов, которые Spring ищет для совета "перехват".

**Листинг 5.18. Класс `SimpleThrowsAdvice`**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;

import org.springframework.aop.ThrowsAdvice;
import org.springframework.aop.framework.ProxyFactory;

public class SimpleThrowsAdvice implements ThrowsAdvice {
    public static void main(String[] args) throws Exception {
        ErrorBean errorBean = new ErrorBean();

        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(errorBean);
        pf.addAdvice(new SimpleThrowsAdvice());
```

```
        ErrorBean proxy = (ErrorBean) pf.getProxy();
        try {
            proxy.errorProneMethod();
        } catch (Exception ignored) {
        }
        try {
            proxy.otherErrorProneMethod();
        } catch (Exception ignored) {
        }
    }
    @Override
    public void afterThrowing(Exception ex) throws Throwable {
        System.out.println("***");
        System.out.println("Generic Exception Capture");
                        // перехват обобщенного исключения
        System.out.println("Caught: " + ex.getClass().getName());
                        // имя исключения
        System.out.println("***\n");
    }
    @Override
    public void afterThrowing(Method method, Object[] args, Object target,
            IllegalArgumentException ex) throws Throwable {
        System.out.println("***");
        System.out.println("IllegalArgumentException Capture");
                        // перехват исключения IllegalArgumentException
        System.out.println("Caught: " + ex.getClass().getName());
                        // имя исключения
        System.out.println("Method: " + method.getName());    // имя метода
        System.out.println("***\n");
    }
}
```

Первое, что Spring ищет для совета "перехват" — это один или более открытых методов по имени `afterThrowing()`. Возвращаемый тип этих методов не важен, хотя мы считаем, что лучше указывать для них тип `void`, поскольку такие методы не могут возвращать значащие данные. Первый метод `afterThrowing()` в классе `SimpleThrowsAdvice` принимает единственный аргумент типа `Exception`. В нем можно указывать любой тип исключения, и этот метод идеально подходит, когда вас не интересует метод, сгенерировавший исключение, или переданные ему аргументы. Обратите внимание, что этот метод перехватывает `Exception` и любые подтипы `Exception`, если только для них не предусмотрены собственные методы `afterThrowing()`.

Во втором методе `afterThrowing()` мы объявили четыре аргумента для указания метода, сгенерировавшего исключение, аргументов, переданных этому методу, и цели вызова метода. Порядок следования аргументов в этом методе важен, и они должны быть указаны все четыре. Обратите внимание, что второй метод `afterThrowing()` перехватывает исключения типа `IllegalArgumentException` (или его подтипа). Выполнение этого примера дает следующий вывод:

```
***
Generic Exception Capture
Caught: java.lang.Exception
***

***
IllegalArgumentException Capture
Caught: java.lang.IllegalArgumentException
Method: otherErrorProneMethod
***
```

Как видите, когда генерируется простое исключение типа `Exception`, вызывается первый метод `afterThrowing()`, но при генерации `IllegalArgumentException` вызывается второй метод `afterThrowing()`. Платформа Spring вызывает одиночный метод `afterThrowing()` по одному разу для каждого исключения, к тому же, как было показано в примере из листинга 5.18, Spring использует метод, сигнатура которого в наибольшей степени соответствует типу исключения. В ситуации, когда совет "перехват" имеет дело с двумя методами `afterThrowing()`, причем оба объявлены с тем самым типом `Exception`, но один принимает единственный аргумент, а другой — четыре аргумента, платформа Spring вызывает метод `afterThrowing()` с четырьмя аргументами.

Совет "перехват" полезен во многих ситуациях; он позволяет классифицировать заново целые иерархии исключений, а также строить централизованную регистрацию исключений для приложения. В частности, мы обнаружили, что совет "перехват" удобен при отладке готовых приложений, т.к. он позволяет добавлять дополнительный код регистрации в журнале, не изменяя код самого приложения.

### *Выбор типа совета*

В общем случае выбор типа совета управляется требованиями к приложению, но вы должны выбирать наиболее специфический тип совета. Другими словами, не используйте совет "вокруг", если подходит совет "перед". В большинстве ситуаций совет "вокруг" может обеспечить все то, что поддерживают остальные три типа советов, однако он может оказаться излишним для реализации задуманного. Применяя наиболее специфический тип совета, вы делаете назначение кода более ясным, а также сокращаете возможности для возникновения ошибок. Предположим, что в совете должно подсчитываться количество вызовов метода. Если использовать для этого совет "перед", то понадобится лишь соответствующим образом закодировать счетчик, но в случае совета "вокруг" вы должны не забыть о вызове самого метода и о возврате значения вызывающему коду. Такие мелочи способствуют проникновению ошибок в приложение. Поддерживая тип совета максимально целенаправленным, вы уменьшаете возможность возникновения ошибок.

## Советы и срезы в Spring

Во всех примерах, показанных до сих пор, при конфигурировании совета для прокси вызывался метод `ProxyFactory.addAdvice()`. "За кулисами" этот метод делегирует свою работу методу `addAdvisor()`, создающему экземпляр `DefaultPointcutAdvisor` и конфигурирующему его со срезом, который указывает на все методы. В такой ситуации совет считается применяемым ко всем методам целевого объекта. В ряде случаев, как, например, при использовании АОП для регис-

трации в журнале, это может быть желательно, но в других обстоятельствах может понадобиться ограничить круг методов, к которым применяется данный совет.

Конечно, можно было бы просто выполнять проверку, подходит ли метод, внутри самого совета, но такому подходу присущи недостатки. Один из них заключается в том, что жесткое кодирование принимаемых методов в совете снижает возможности его многократного использования. С помощью срезов можно конфигурировать перечень методов, к которым будет применяться совет, не помещая этот код внутрь совета; это очевидным образом увеличит степень повторного использования совета. Другие недостатки жесткого кодирования списка методов в совете относятся к производительности. Если снабжаемый советом метод проверяется внутри совета, это происходит каждый раз, когда вызывается любой метод целевого объекта. В результате производительность приложения снижается. Когда применяются срезы, проверка выполняется по одному разу для каждого метода, а результаты кешируются для дальнейшего использования. Еще один недостаток отказа от применения срезов для ограничения списка снабжаемых советом методов, также связанный с производительностью, касается того, что при создании прокси Spring может оптимизировать методы, не снабженные советом, и это ускоряет их вызовы. Более подробно вопросы, касающиеся такой оптимизации, будут рассматриваться далее в этой главе.

Мы настоятельно рекомендуем не поддаваться искушению и не кодировать жестко проверки методов внутри совета, а взамен этого, где только возможно, пользоваться срезами для управления применимостью совета к методам целевого объекта. Тем не менее, в некоторых случаях жесткое кодирование проверок в совете является необходимым. Вспомните рассмотренный ранее пример совета "после возврата", который был предназначен для перехвата слабых ключей, генерируемых классом `KeyGenerator`. Такой вид совета тесно связан с классом, для которого он назначен, и вполне разумно внутри него проверять, к правильному ли типу он применяется. Подобное связывание совета и целевого объекта мы называем *родственностью цели*. В общем случае срезы должны использоваться, когда совет имеет небольшую родственность цели или вообще ее не имеет — т.е. может быть применен к любому типу или к широкому диапазону типов. Если совет имеет строгую родственность цели, попробуйте проверить корректность использования совета в нем самом; это поможет уменьшить ошибки, связанные с неправильным применением совета. Мы также рекомендуем избегать снабжения методов советами без особой нужды. Как вы увидите, это приведет к заметному снижению скорости вызовов, что может оказать серьезное влияние на общую производительность приложения.

## Интерфейс `Pointcut`

Срезы в Spring создаются путем реализации интерфейса `Pointcut`, который показан в листинге 5.19.

**Листинг 5.19. Интерфейс `Pointcut`**

```
package org.springframework.aop;

public interface Pointcut {
   ClassFilter getClassFilter ();
   MethodMatcher getMethodMatcher();
}
```

В этом коде видно, что в интерфейсе `Pointcut` определены два метода, `getClassFilter()` и `getMethodMatcher()`, которые возвращают экземпляры `ClassFilter` и `MethodMatcher`. Очевидно, что если вы решили реализовать интерфейс `Pointcut`, то должны реализовать эти методы. К счастью, как будет показано в следующем разделе, обычно это не обязательно, поскольку Spring предоставляет на выбор реализации `Pointcut`, покрывающие большинство, если не все, сценарии использования.

При выяснении, применим ли интерфейс `Pointcut` к конкретному методу, Spring сначала проверяет, применим ли `Pointcut` к классу этого метода, используя экземпляр `ClassFilter`, который возвращается вызовом `Pointcut.getClassFilter()`. Интерфейс `ClassFilter` приведен в листинге 5.20.

**Листинг 5.20. Интерфейс `ClassFilter`**

```
org.springframework.aop;
public interface ClassFilter {
   boolean matches(Class<?> clazz);
}
```

Как видите, в интерфейсе `ClassFilter` определен единственный метод `matches()`, принимающий экземпляр `Class`, который представляет класс, предназначенный для проверки. Метод `matches()` возвращает `true`, если срез применим к классу, и `false` — в противном случае.

Интерфейс `MethodMatcher` сложнее интерфейса `ClassFilter`; он показан в листинге 5.21.

**Листинг 5.21. Интерфейс `MethodMatcher`**

```
package org.springframework.aop;
public interface MethodMatcher {
   boolean matches(Method m, Class<?> targetClass);
   boolean isRuntime();
   boolean matches(Method m, Class<?> targetClass, Object[] args);
}
```

В Spring поддерживаются два типа `MethodMatcher`, статический и динамический, что определяется по возвращаемому значению метода `isRuntime()`. Перед использованием `MethodMatcher` платформа Spring вызывает `isRuntime()` для выяснения, является ли `MethodMatcher` статическим, на что указывает возвращенное значение `false`, или же динамическим, что отражается значением `true`.

Для статического среза Spring вызывает метод `matches(Method, Class<T>)` интерфейса `MethodMatcher` по одному разу для каждого метода целевого объекта, кешируя возвращаемое значение для последующих обращений к этому методу. Таким образом, проверка применимости метода производится только однократно для каждого метода, и последующие обращения к методу не приводят к вызову `matches()`.

Для динамических срезов Spring по-прежнему выполняет статическую проверку с помощью `matches(Method, Class<T>)` при вызове метода в первый раз, чтобы определить общую применимость этого метода. Однако в дополнение к этому и при условии, что статическая проверка возвратила `true`, платформа Spring прово-

дит дальнейшую проверку для каждого вызова метода, используя `matches(Method, Class<T>, Object[])`. Таким образом, динамический `MethodMatcher` может выяснить, должен ли применяться срез, на основе конкретного вызова метода, а не только самого метода. Например, срез необходимо применять, только если аргумент представляет собой значение типа `Integer`, которое больше 100. В этом случае в методе `matches(Method, Class<T>, Object[])` может быть предусмотрен код для дополнительной проверки аргумента при каждом вызове.

Очевидно, что статические срезы выполняются намного быстрее динамических, т.к. не требуют дополнительной проверки при каждом вызове. Динамические срезы обеспечивают больший уровень гибкости при решении о том, применять ли совет. В общем случае мы рекомендуем использовать статические срезы везде, где только возможно. Однако в случаях, когда совет добавляет значительные накладные расходы, может быть разумно избегать любых необязательных обращений к совету за счет применения динамического среза.

В целом вам редко придется создавать собственные реализации интерфейса `Pointcut` с нуля, потому что Spring предоставляет абстрактные базовые классы и для статических, и для динамических срезов. Эти базовые классы, а также другие реализации `Pointcut`, рассматриваются в последующих разделах.

### *Доступные реализации интерфейса `Pointcut`*

В версии Spring 4.0 предлагаются восемь реализаций интерфейса `Pointcut`: два абстрактных класса, служащие вспомогательными классами для создания статических и динамических срезов, и шесть конкретных классов, которые предназначены для решения следующих задач:

- объединение множества срезов в одно целое;
- поддержка срезов потока управления;
- выполнение простых сопоставлений на основе имени;
- определение срезов с использованием регулярных выражений;
- определение срезов с применением выражений AspectJ;
- определение срезов, которые ищут специфические аннотации на уровне классов или методов.

В табл. 5.2 приведена сводка по восьми реализациям интерфейса `Pointcut`.

**Таблица 5.2. Сводка по реализациям интерфейса `Pointcut`**

| Класс реализации | Описание |
|---|---|
| `org.springframework.aop.support.annotation.AnnotationMatchingPointcut` | Срез, который ищет специфическую Java-аннотацию в классе или методе. Этот класс требует JDK 5 или более новой версии |
| `org.springframework.aop.aspectj.AspectJExpressionPointcut` | Срез, который использует средство связывания AspectJ для оценки выражения среза, представленного с помощью синтаксиса AspectJ |
| `org.springframework.aop.support.ComposablePointcut` | Класс `ComposablePointcut` применяется для объединения двух и более срезов с помощью таких операций, как `union()` и `intersection()` |

*Окончание табл. 5.2*

| Класс реализации | Описание |
|---|---|
| `org.springframework.aop.support.ControlFlowPointcut` | Класс `ControlFlowPointcut` представляет срез, предназначенный для специального случая, который соответствует всем методам в потоке управления другого метода — т.е. любому методу, который вызван прямо или косвенно в результате выполнения другого метода |
| `org.springframework.aop.support.DynamicMethodMatcherPointcut` | Класс `DynamicMethodMatcherPointcut` служит базовым классом для построения динамических срезов |
| `org.springframework.aop.support.JdkRegexpMethodPointcut` | Класс `JdkRexepMethodPointcut` позволяет определять срезы с использованием поддержки регулярных выражений JDK 1.4. Этот класс требует JDK 1.4 или более новой версии |
| `org.springframework.aop.support.NameMatchMethodPointcut` | С помощью класса `NameMatchMethodPointcut` можно создать срез, который выполняет простое сопоставление со списком имен методов |
| `org.springframework.aop.support.StaticMethodMatcherPointcut` | Класс `StaticMethodMatcherPointcut` служит базовым классом для построения статических срезов |

На рис. 5.3 показана диаграмма UML для классов реализации `Pointcut`.

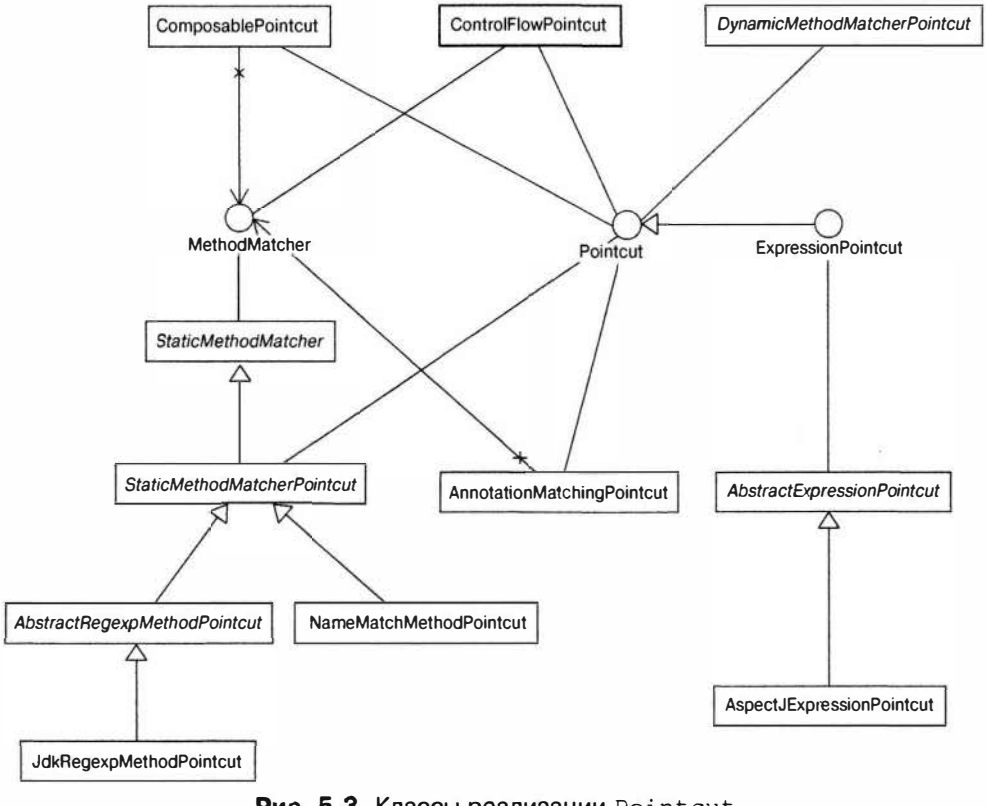


**Рис. 5.3.** Классы реализации `Pointcut`

### Использование DefaultPointcutAdvisor

Прежде чем можно будет использовать любую реализацию Pointcut, потребуется реализовать интерфейс Advisor или, точнее говоря, интерфейс PointcutAdvisor. Как упоминалось ранее (в разделе "Аспекты в Spring"), интерфейс Advisor предназначен для представления аспекта в Spring — связки совета и срезов, управляющей тем, какие методы должны снабжаться советом, и как это должно делаться. Платформа Spring предлагает несколько реализаций PointcutAdvisor, но сейчас мы займемся только одной из них — DefaultPointcutAdvisor. Класс DefaultPointcutAdvisor — это просто PointcutAdvisor для связывания одного Pointcut с одним Advice.

### Создание статического среза с использованием класса StaticMethodMatcherPointcut

В этом разделе мы создадим простой статический срез, расширив абстрактный класс StaticMethodMatcherPointcut. Поскольку класс StaticMethodMatcherPointcut расширяет класс StaticMethodMatcher (также абстрактный), который реализует интерфейс MethodMatcher, потребуется реализовать метод matches(Method, Class<?>); остальная часть реализации Pointcut поддерживается автоматически.

Хотя это единственный метод, который должен быть реализован (при расширении класса StaticMethodMatcherPointcut), может также понадобиться переопределить метод getClassFilter(), что и сделано в этом примере, чтобы гарантировать снабжение советом только методов подходящего типа.

В рассматриваемом примере мы имеем два класса, BeanOne и BeanTwo, в которых определены идентичные методы. В листинге 5.22 показан код класса BeanOne.

**Листинг 5.22. Класс BeanOne**

```
package com.apress.prospring4.ch5;

public class BeanOne {
    public void foo() {
        System.out.println("foo");
    }

    public void bar() {
        System.out.println("bar");
    }
}
```

Класс BeanTwo имеет те же методы, что и класс BeanOne. В этом примере нам нужна возможность создать прокси для обоих классов, используя тот же самый класс DefaultPointcutAdvisor, но применить совет только к методу foo() класса BeanOne. Чтобы сделать это, мы создали класс SimpleStaticPointcut, код которого приведен в листинге 5.23.

**Листинг 5.23. Класс SimpleStaticPointcut**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;
import org.springframework.aop.ClassFilter;
import org.springframework.aop.support.StaticMethodMatcherPointcut;

public class SimpleStaticPointcut extends StaticMethodMatcherPointcut {
    @Override
    public boolean matches(Method method, Class<?> cls) {
        return ("foo".equals(method.getName()));
    }

    @Override
    public ClassFilter getClassFilter() {
        return new ClassFilter() {
            public boolean matches(Class<?> cls) {
                return (cls == BeanOne.class);
            }
        };
    }
}
```

Здесь мы реализовали метод `matches(Method, Class<?>)` согласно требованию абстрактного класса `StaticMethodMatcher`. Реализация просто возвращает `true`, если именем метода является `foo`, и `false` — в противном случае. Обратите внимание, что метод `getClassFilter()` также был переопределен; он возвращает экземпляр `ClassFilter`, метод `matches()` которого возвращает `true` только для класса `BeanOne`. Благодаря этому статическому срезу обеспечивается соответствие только для методов класса `BeanOne` и, более того, только для метода `foo()` этого класса.

В листинге 5.24 показан код класса `SimpleAdvice`, который просто выводит сообщения с обеих сторон вызова метода.

**Листинг 5.24. Класс SimpleAdvice**

```
package com.apress.prospring4.ch5;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class SimpleAdvice implements MethodInterceptor {
    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
      System.out.println(">> Invoking " + invocation.getMethod().getName());
      Object retVal = invocation.proceed();
      System.out.println(">> Done");
      return retVal;
    }
}
```

В листинге 5.25 представлен код простого тестового приложения для этого примера, которое создает экземпляр класса `DefaultPointcutAdvisor` с применением классов `SimpleAdvice` и `SimpleStaticPointcut`.

**Листинг 5.25. Класс StaticPointcutExample**

```
package com.apress.prospring4.ch5;

import org.aopalliance.aop.Advice;
import org.springframework.aop.Advisor;
import org.springframework.aop.Pointcut;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;

public class StaticPointcutExample {
   public static void main(String[] args) {
      BeanOne one = new BeanOne();
      BeanTwo two = new BeanTwo();

      BeanOne proxyOne;
      BeanTwo proxyTwo;

      Pointcut pc = new SimpleStaticPointcut();
      Advice advice = new SimpleAdvice();
      Advisor advisor = new DefaultPointcutAdvisor(pc, advice);

      ProxyFactory pf = new ProxyFactory();
      pf.addAdvisor(advisor);
      pf.setTarget(one);
      proxyOne = (BeanOne)pf.getProxy();

      pf = new ProxyFactory();
      pf.addAdvisor(advisor);
      pf.setTarget(two);
      proxyTwo = (BeanTwo)pf.getProxy();

      proxyOne.foo();
      proxyTwo.foo();

      proxyOne.bar();
      proxyTwo.bar();
   }
}
```

Обратите внимание, что экземпляр `DefaultPointcutAdvisor` затем используется при создании двух прокси: для экземпляра `BeanOne` и для экземпляра `BeanTwo`. Наконец, на этих двух прокси вызываются методы `foo()` и `bar()`.

Выполнение этого примера дает в результате следующий вывод:

```
>> Invoking foo
foo
>> Done
foo
bar
bar
```

Как видите, единственным методом, к которому применялся `SimpleAdvice`, был `foo()` класса `BeanOne`, что и ожидалось. Ограничение методов, к которым применяется совет, реализуется довольно просто и, как будет показано при обсуждении вариантов прокси, является ключевым фактором обеспечения наилучшей производительности приложения.

### Создание динамического среза с использованием класса `DynamicMethodMatcherPointcut`

Создание динамического среза не особенно отличается от создания статического среза, поэтому в рассматриваемом примере мы создаем динамический срез для класса, код которого приведен в листинге 5.26.

**Листинг 5.26. Класс `SampleBean`**

```
package com.apress.prospring4.ch5;

public class SampleBean {
    public void foo(int x) {
        System.out.println("Invoked foo() with: " + x);
    }

    public void bar() {
        System.out.println("Invoked bar()");
    }
}
```

В этом примере мы хотим снабдить советом только метод `foo()`, но в отличие от предыдущего примера, сделать это тогда, когда передаваемый ему аргумент `int` имеет значение, которое больше или меньше 100.

Как и в случае со статическими срезами, для создания динамических срезов Spring предлагает удобный базовый класс — `DynamicMethodMatcherPointcut`. Этот класс имеет единственный абстрактный метод `matches(Method, Class<?>, Object[])` (благодаря реализуемому им интерфейсу `MethodMatcher`), который должен быть реализован, но, как вы увидите, разумно также реализовать метод `matches(Method, Class<?>)` для управления поведением статических проверок. Код класса `SimpleDynamicPointcut` показан в листинге 5.27.

**Листинг 5.27. Класс `SimpleDynamicPointcut`**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;

import org.springframework.aop.ClassFilter;
import org.springframework.aop.support.DynamicMethodMatcherPointcut;

public class SimpleDynamicPointcut extends DynamicMethodMatcherPointcut {
    @Override
    public boolean matches(Method method, Class<?> cls) {
        // Выполнить статическую проверку
        System.out.println("Static check for " + method.getName());
        return ("foo".equals(method.getName()));
    }

    @Override
    public boolean matches(Method method, Class<?> cls, Object[] args) {
        // Выполнить динамическую проверку
        System.out.println("Dynamic check for " + method.getName());
        int x = ((Integer) args[0]).intValue();
        return (x != 100);
    }
```

```
    @Override
    public ClassFilter getClassFilter() {
        return new ClassFilter() {
            public boolean matches(Class<?> cls) {
                return (cls == SampleBean.class);
            }
        };
    }
}
```

В коде, приведенном в листинге 5.27, легко заметить, что мы переопределяем метод `getClassFilter()` подобно предыдущему примеру из листинга 5.23. Это устраняет необходимость в проверке класса в методах сопоставления имен методов — то, что особенно важно для динамической проверки. Хотя обязательной является только реализация динамической проверки, мы также реализуем и статическую проверку. Причина заключается в том, что, как нам известно, метод `bar()` никогда не будет снабжаться советом. Если указать на данный факт с использованием статической проверки, то Spring никогда не будет выполнять динамическую проверку для этого метода. Это объясняется тем, что когда метод статической проверки реализован, Spring сначала воспользуется им, и если результат проверки покажет несоответствие, то дальнейшая динамическая проверка не предпринимается. Кроме того, результат статической проверки кешируется для улучшения производительности. Но если пренебречь статической проверкой, то Spring будет выполнять динамическую проверку при каждом вызове метода `bar()`. Рекомендуется выполнять проверку класса в методе `getClassFilter()`, проверку метода — в методе `matches(Method, Class<?>)` и проверку аргумента — в методе `matches(Method, Class<?>, Object[])`. Это сделает срез намного проще для понимания и сопровождения, а также улучшит показатели производительности.

В методе `matches(Method, Class<?>, Object[])` мы возвращаем `true`, если значение, переданное в аргументе `int` методу `foo()`, не равно 100, и `false` — в противном случае. Обратите внимание, что при выполнении динамической проверки известно, что мы имеем дело с методом `foo()`, поскольку никакие другие методы не прошли статической проверки.

В листинге 5.28 приведен пример этого среза в действии.

**Листинг 5.28. Класс `DynamicPointcutExample`**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;

public class DynamicPointcutExample {
    public static void main(String[] args) {
        SampleBean target = new SampleBean();

        Advisor advisor = new DefaultPointcutAdvisor(
            new SimpleDynamicPointcut(), new SimpleAdvice());

        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        SampleBean proxy = (SampleBean)pf.getProxy();
```

```
        proxy.foo(1);
        proxy.foo(10);
        proxy.foo(100);

        proxy.bar();
        proxy.bar();
        proxy.bar();
    }
}
```

Обратите внимание, что мы используем тот же самый класс совета, что и в примере со статическим срезом. Однако в текущем примере должны быть снабжены советом только первые два вызова `foo()`. Динамическая проверка предотвращает снабжением советом третий вызов `foo()`, а статическая проверка не позволяет снабдить советом метод `bar()`. Запуск этого примера дает следующий вывод:

```
Static check for foo
Static check for bar
Static check for toString
Static check for clone
Static check for foo
Dynamic check for foo
>> Invoking foo
Invoked foo() with: 1
>> Done
Dynamic check for foo
>> Invoking foo
Invoked foo() with: 10
>> Done
Dynamic check for foo
Invoked foo() with: 100
Static check for bar
Invoked bar()
Invoked bar()
Invoked bar()
```

Как и ожидалось, советом были снабжены только первые два обращения к методу `foo()`. Обратите внимание, что благодаря статической проверке метода `bar()`, ни один из вызовов `bar()` не подвергался динамической проверке. Интересно отметить также и то, что метод `foo()` участвовал в двух статических проверках: во время начальной фазы, когда проверялись все методы, и при его вызове в первый раз.

Пример показал, что динамические проверки обеспечивают более высокую гибкость, чем статические проверки, но из-за дополнительных накладных расходов во время выполнения они должны использоваться только в случае абсолютной необходимости.

### *Использование простого сопоставления имен*

Часто при создании среза требуется выполнять сопоставление на основе лишь имени метода, игнорируя его сигнатуру и возвращаемый тип. В таком случае можно избежать создания подкласса `StaticMethodMatcherPointcut` и применять вместо него `NameMatchMethodPointcut` (подкласс `StaticMethodMatcherPointcut`) для сопоставления со списком имен методов.

Когда используется NameMatchMethodPointcut, никакого внимания сигнатуре метода не уделяется, поэтому если есть, например, методы foo() и foo(int), то они оба соответствуют имени foo.

В листинге 5.29 показан простой класс с четырьмя методами.

**Листинг 5.29. Класс NameBean**

```
package com.apress.prospring4.ch5;

public class NameBean {
    public void foo() {
        System.out.println("foo");
    }
    public void foo(int x) {
        System.out.println("foo " + x);
    }
    public void bar() {
        System.out.println("bar");
    }
    public void yup() {
        System.out.println("yup");
    }
}
```

В рассматриваемом примере мы хотим реализовать сопоставление с методами foo(), foo(int) и bar() с помощью NameMatchMethodPointcut; это сводится к сопоставлению с именами foo и bar. Код приведен в листинге 5.30.

**Листинг 5.30. Использование NameMatchMethodPointcut**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.aop.support.NameMatchMethodPointcut;

public class NamePointcutExample {
    public static void main(String[] args) {
        NameBean target = new NameBean();

        NameMatchMethodPointcut pc = new NameMatchMethodPointcut();
        pc.addMethodName("foo");
        pc.addMethodName("bar");
        Advisor advisor = new DefaultPointcutAdvisor(pc, new SimpleAdvice());

        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        NameBean proxy = (NameBean)pf.getProxy();

        proxy.foo();
        proxy.foo(999);
        proxy.bar();
        proxy.yup();
    }
}
```

Потребность в построении класса для среза отсутствует; достаточно просто создать экземпляр `NameMatchMethodPointcut`. Обратите внимание, что к срезу было добавлено два имени, `foo` и `bar`, с использованием метода `addMethodName()`. Запуск этого примера дает следующий вывод:

```
>> Invoking foo
foo
>> Done
>> Invoking foo
foo 999
>> Done
>> Invoking bar
bar
>> Done
yup
```

Как и ожидалось, благодаря срезу методы `foo()`, `foo(int)` и `bar()` были снабжены советами, но метод `yup()` остался незатронутым.

### *Создание срезов с помощью регулярных выражений*

В предыдущем разделе мы обсуждали, как выполнить простое сопоставление с предварительно определенным списком методов. Но как быть в ситуации, если все имена методов заранее не известны, но существует шаблон, которому эти имена следуют? Например, что если необходимо обеспечить соответствие всем методам, имена которых начинаются на `get`? В этом случае можно воспользоваться срезом с регулярным выражением `JdkRegexpMethodPointcut`, который позволяет выполнять сопоставление на основе регулярного выражения.

В листинге 5.31 показан код простого класса с тремя методами.

**Листинг 5.31. Класс `RegexpBean`**

```
package com.apress.prospring4.ch5;

public class RegexpBean {
    public void foo1() {
        System.out.println("foo1");
    }
    public void foo2() {
        System.out.println("foo2");
    }
    public void bar() {
        System.out.println("bar");
    }
}
```

Используя срез с регулярным выражением, можно реализовать соответствие всех методов этого класса, имена которых начинаются с `foo`. Необходимый код приведен в листинге 5.32.

**Листинг 5.32. Использование регулярных выражений для срезов**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.Advisor;
```

```
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.aop.support.JdkRegexpMethodPointcut;

public class RegexpPointcutExample {
   public static void main(String[] args) {
      RegexpBean target = new RegexpBean();

      JdkRegexpMethodPointcut pc = new JdkRegexpMethodPointcut();
      pc.setPattern(".*foo.*");
      Advisor advisor = new DefaultPointcutAdvisor(pc, new SimpleAdvice());

      ProxyFactory pf = new ProxyFactory();
      pf.setTarget(target);
      pf.addAdvisor(advisor);
      RegexpBean proxy = (RegexpBean) pf.getProxy();

      proxy.foo1();
      proxy.foo2();
      proxy.bar();
   }
}
```

Обратите внимание, что мы не обязаны создавать класс для среза; вместо этого мы просто создаем экземпляр `JdkRegexpMethodPointcut` и указываем шаблон для сопоставления. Что интересно отметить, так это шаблон. При сопоставлении имен методов Spring использует полностью определенное имя метода, т.е. для `foo1()` производится сопоставление с `com.apress.prospring4.ch5.RegexpBean.foo1`, чем и объясняется наличие конструкции `.*` в начале шаблона. Это мощная концепция, поскольку она позволяет производить сопоставление со всеми методами внутри заданного пакета без необходимости в знании точного перечня классов в пакете, а также имен имеющихся методов. Запуск этого примера дает следующий вывод:

```
>> Invoking foo1
foo1
>> Done
>> Invoking foo2
foo2
>> Done
bar
```

Как и можно было ожидать, советом были снабжены только методы `foo1()` и `foo2()`, т.к. метод `bar()` не соответствовал шаблону регулярного выражения.

### *Создание срезов с помощью выражения для срезов AspectJ*

Кроме регулярных выражений JDK срезы можно также объявлять с применением языка выражений для срезов AspectJ. Позже в этой главе вы увидите, что при объявлении среза в XML-конфигурации с использованием пространства имен `aop` платформа Spring по умолчанию применяет язык срезов AspectJ. Кроме того, когда задействована предлагаемая Spring поддержка АОП в стиле аннотаций `@AspectJ`, также необходимо использовать язык срезов AspectJ. Таким образом, при объявлении срезов с помощью языка выражений лучше всего подойдут выражения срезов AspectJ. Платформа Spring предоставляет класс `AspectJExpressionPointcut` для определения срезов посредством языка выражений AspectJ. Для работы с выраже-

ниями срезов AspectJ в Spring понадобится включить в путь классов проекта два библиотечных файла AspectJ — `aspectjrt.jar` и `aspectjweaver.jar`. Просто добавьте к проекту зависимости, показанные в табл. 5.3.

**Таблица 5.3. Зависимости Maven для AspectJ**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.aspectj` | `aspectjrt` | 1.7.4 (1.8.0.M1 для Java 8) | Библиотека времени выполнения AspectJ |
| `org.aspectj` | `aspectjweaver` | 1.7.4 (1.8.0.M1 для Java 8) | Библиотека связывания AspectJ |

Давайте снова обратимся к предыдущему примеру применения регулярных выражений JDK и посмотрим, как получить тот же самый результат с использованием выражения AspectJ. В листинге 5.33 показан бин, который в точности такой же, как в листинге 5.31; отличается только имя.

**Листинг 5.33. Класс `AspectjexpBean`**

```
package com.apress.prospring4.ch5;

public class AspectjexpBean {
    public void foo1() {
        System.out.println("foo1");
    }
    public void foo2() {
        System.out.println("foo2");
    }
    public void bar() {
        System.out.println("bar");
    }
}
```

С помощью среза, основанного на выражении AspectJ, можно также обеспечить сопоставление со всеми методами этого класса, которые имеют имена, начинающиеся с `foo`. Соответствующий код приведен в листинге 5.34.

**Листинг 5.34. Использование выражений AspectJ для срезов**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.Advisor;
import org.springframework.aop.aspectj.AspectJExpressionPointcut;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;

public class AspectjexpPointcutExample {
    public static void main(String[] args) {
        AspectjexpBean target = new AspectjexpBean();

        AspectJExpressionPointcut pc = new AspectJExpressionPointcut();
        pc.setExpression("execution(* foo*(..))");
        Advisor advisor = new DefaultPointcutAdvisor(pc, new SimpleAdvice());
```

```
        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        AspectjexpBean proxy = (AspectjexpBean) pf.getProxy();

        proxy.foo1();
        proxy.foo2();
        proxy.bar();
    }
}
```

Обратите внимание на вызов метода `setExpression()` класса `AspectJExpressionPointcut` для установки критерия совпадения.

Выражение `"execution(* foo*(..))"` означает, что совет должен применяться к выполнению любых методов, которые имеют имена, начинающиеся с `foo`, принимают любые аргументы и возвращают значение любого типа. Запуск программы даст те же результаты, что и предыдущий пример использования регулярных выражений JDK.

### *Создание срезов, соответствующих аннотациям*

Если приложение основано на аннотациях, может возникнуть желание задействовать для определения срезов собственные аннотации, т.е. применять логику совета ко всем методам или типам со специфическими аннотациями. В Spring доступен класс `AnnotationMatchingPointcut` для определения срезов с использованием аннотаций. Мы снова прибегнем к предыдущему примеру и посмотрим, как в нем применить аннотацию в качестве среза.

Первым делом определим интерфейс аннотации по имени `AdviceRequired`, который представляет аннотацию, используемую для объявления среза. Код показан в листинге 5.35.

**Листинг 5.35. Использование аннотации для срезов**

```
package com.apress.prospring4.ch5;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface AdviceRequired {
}
```

В предыдущем листинге мы объявили интерфейс как аннотацию с указанием `@interface` в качестве типа, при этом аннотация `@Target` определяет, что аннотация `@AdviceRequired` может применяться либо на уровне типа, либо на уровне метода.

В листинге 5.36 приведен код простого бина с аннотацией `@AdviceRequired`.

**Листинг 5.36. Класс SampleAnnotationBean**

```
package com.apress.prospring4.ch5;

public class SampleAnnotationBean {
    @AdviceRequired
    public void foo(int x) {
        System.out.println("Invoked foo() with: "  +x);
    }
    public void bar() {
        System.out.println("Invoked bar()");
    }
}
```

В представленном выше бине метод `foo()` аннотирован с помощью `@AdviceRequired` и к нему должен быть применен совет.

В листинге 5.37 показан код тестовой программы.

**Листинг 5.37. Тестирование среза, использующего аннотацию**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.aop.support.annotation.
AnnotationMatchingPointcut;
public class AnnotationPointcutExample {
    public static void main(String[] args) {
        SampleAnnotationBean target = new SampleAnnotationBean();

        AnnotationMatchingPointcut pc = AnnotationMatchingPointcut
            .forMethodAnnotation(AdviceRequired.class);
        Advisor advisor = new DefaultPointcutAdvisor(pc, new SimpleAdvice());

        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        SampleAnnotationBean proxy = (SampleAnnotationBean) pf.getProxy();
        proxy.foo(100);
        proxy.bar();
    }
}
```

В этом коде запрашивается экземпляр класса `AnnotationMatchingPointcut` за счет вызова его статического метода `forMethodAnnotation()`, которому передается тип аннотации. Это указывает, что мы хотим применить совет ко всем методам, аннотированным заданной аннотацией. Также возможно указать аннотации на уровне типа, вызвав метод `forClassAnnotation()`. Ниже представлен вывод, полученный в результате запуска этой программы:

```
>> Invoking foo
Invoked foo() with: 100
>> Done
Invoked bar()
```

Как видите, поскольку метод `foo()` аннотирован, совет применился только к нему.

### Удобные реализации `Advisor`

Для многих реализаций `Pointcut` в Spring также предлагается удобная реализация `Advisor`, которая действует в качестве среза.

Например, вместо использования в предыдущем примере `NameMatchMethodPointcut` совместно с `DefaultPointcutAdvisor` можно было бы просто применить `NameMatchMethodPointcutAdvisor` (листинг 5.38).

**Листинг 5.38. Использование `NameMatchMethodPointcutAdvisor`**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.NameMatchMethodPointcutAdvisor;

import com.apress.prospring4.ch5.staticpc.SimpleAdvice;

public class NamePointcutUsingAdvisor {
    public static void main(String[] args) {
        NameBean target = new NameBean();

        NameMatchMethodPointcutAdvisor advisor = new
            NameMatchMethodPointcutAdvisor(new SimpleAdvice());
        advisor.addMethodName("foo");
        advisor.addMethodName("bar");

        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        NameBean proxy = (NameBean) pf.getProxy();

        proxy.foo();
        proxy.foo(999);
        proxy.bar();
        proxy.yup();
    }
}
```

Обратите внимание, что вместо создания экземпляра `NameMatchMethodPointcut` мы конфигурируем детали среза с помощью экземпляра `NameMatchMethodPointcutAdvisor`. В этом случае `NameMatchMethodPointcutAdvisor` действует и как `Advisor`, и как `Pointcut`.

Узнать подробные сведения о различных реализациях `Advisor` можно в документации Javadoc по пакету `org.springframework.aop.support`. Между этими двумя подходами нет заметной разницы в производительности, и помимо чуть меньшего объема кода во втором примере отличия при кодировании незначительны. Мы предпочитаем придерживаться первого подхода, поскольку считаем, что он дает более ясный код. В конце концов, выбор стиля является исключительно персональным предпочтением.

# Что собой представляют прокси

До сих пор мы только вскользь упоминали о прокси, которые генерируются `ProxyFactory`. Мы указывали, что в Spring доступны два типа прокси: прокси JDK, создаваемые с использованием класса `Proxy` из JDK, и прокси на основе CGLIB, создаваемые с помощью класса `Enhancer` из CGLIB. Вас может интересовать, в чем заключаются отличия между этими двумя типами прокси, и для чего они применяются в Spring. В этом разделе мы детально рассмотрим отличия между упомянутыми прокси.

Основным предназначением прокси является перехват вызовов методов и, где это необходимо, выполнение цепочек совета, который применяется к отдельному методу. Управление и вызов совета в основном не зависит от прокси, и этим занимается инфраструктура АОП в Spring. Однако прокси отвечает за перехват вызовов всех методов и передачу их при необходимости инфраструктуре АОП для применения совета.

Вдобавок к этой ключевой функциональности прокси должен также поддерживать набор дополнительных средств. Прокси можно сконфигурировать так, чтобы он был доступен через класс `AopContext` (который является абстрактным); это позволит извлечь прокси и вызвать оснащенные советом методы из целевого объекта. Прокси отвечает за обеспечение того, что если эта опция включена с помощью `ProxyFactory.setExposeProxy()`, то класс прокси становится соответствующим образом доступным. В дополнение к этому все классы прокси по умолчанию реализуют интерфейс `Advised`, который, помимо прочего, позволяет изменять цепочку совета после того, как прокси был создан. Прокси должен также гарантировать, чтобы любой метод, который возвращает `this` (т.е. возвращает цель с прокси), фактически возвращал прокси, а не целевой объект.

Как видите, типичный прокси должен выполнять немало работы, и вся нужная логика реализована в прокси JDK и CGLIB.

## Использование динамических прокси JDK

Прокси JDK представляют собой наиболее базовый тип прокси, доступный в Spring. В отличие от прокси CGLIB, прокси JDK могут генерироваться только для интерфейсов, но не классов. Таким образом, любой объект, для которого необходим прокси, должен реализовывать хотя бы один интерфейс. В общем случае использование интерфейсов для классов является удачным проектным решением, однако это не всегда возможно, особенно если приходится работать с унаследованным кодом или кодом от третьих сторон. В таком случае должен применяться прокси CGLIB.

При использовании прокси JDK все вызовы методов перехватываются JVM и направляются методу `invoke()` прокси. Затем `invoke()` выясняет, снабжен ли вызываемый метод советом (согласно правилам, определяемым срезом), и если это так, вызывает цепочку совета и сам метод с помощью рефлексии. Вдобавок метод `invoke()` выполняет всю логику, которая обсуждалась в предыдущем разделе.

Прокси JDK не разделяет методы на снабженные и не снабженные советом вплоть до вызова метода `invoke()`. Это означает, что для методов, не снабженных советом, метод `invoke()` на прокси по-прежнему вызывается, все проверки выполняются, и методы запускаются с применением рефлексии. Очевидно, что при каж-

дом вызове метода возникают накладные расходы времени выполнения, хотя часто прокси не реализует никакой дополнительной обработки кроме вызова через рефлексию метода, не снабженного советом.

Чтобы сообщить `ProxyFactory` о необходимости использования прокси JDK, необходимо указать список интерфейсов для прокси с помощью метода `setInterfaces()` (в классе `AdvisedSupport`, который класс `ProxyFactory` косвенно расширяет).

## Использование прокси CGLIB

В случае прокси JDK все решения относительно обработки конкретного вызова метода принимаются во время выполнения при каждом таком вызове. Библиотека CGLIB динамически генерирует байт-код нового класса для каждого прокси, по возможности повторно используя ранее сгенерированные классы.

Когда *прокси CGLIB* создается в первый раз, библиотека CGLIB запрашивает Spring о том, как желательно поддерживать каждый метод. Это означает, что многие решения, принимаемые при каждом обращении к `invoke()` в прокси JDK, в случае прокси CGLIB производятся только один раз. Поскольку CGLIB генерирует действительный байт-код, появляется намного больше гибкости в том, как поддерживать методы. Например, прокси CGLIB генерирует соответствующий байт-код для вызова любых методов, не снабженных советом, напрямую, сокращая накладные расходы, привносимые прокси. Вдобавок прокси CGLIB определяет, может ли метод возвращать `this`, и если нет, то позволяет вызову метода выполняться напрямую, снова сокращая накладные расходы времени выполнения.

Прокси CGLIB также поддерживает цепочки фиксированных советов отличающимся от прокси JDK образом. Цепочка фиксированного совета — это такая цепочка, которая не будет изменяться после генерации прокси. По умолчанию имеется возможность изменить средства снабжения советом и сам совет даже после создания прокси, хотя такое требуется редко. Прокси CGLIB обрабатывает цепочки фиксированных советов специальным образом, сокращая накладные расходы времени выполнения, которые связаны с цепочкой совета.

## Сравнение производительности прокси

До сих пор мы занимались обсуждением в свободной форме отличий в реализации разных типов прокси. В этом разделе мы собираемся прогнать простой тест для сравнения производительности прокси CGLIB и прокси JDK.

Давайте создадим интерфейс `SimpleBean` и реализующий его класс `DefaultSimpleBean`, который будет использоваться в качестве целевого объекта при создании прокси. Код интерфейса `SimpleBean` и класса `DefaultSimpleBean` показан в листингах 5.39 и 5.40.

**Листинг 5.39. Интерфейс `SimpleBean`**

```
package com.apress.prospring4.ch5;

public interface SimpleBean {
    void advised();
    void unadvised();
}
```

**Листинг 5.40. Класс `DefaultSimpleBean`**

```
package com.apress.prospring4.ch5;

public class DefaultSimpleBean implements SimpleBean {
    private long dummy = 0;

    @Override
    public void advised() {
        dummy = System.currentTimeMillis();
    }

    @Override
    public void unadvised() {
        dummy = System.currentTimeMillis();
    }
}
```

В листинге 5.41 приведен код класса `TestPointcut`, который обеспечивает статическую проверку метода, снабжаемого советом.

**Листинг 5.41. Класс `TestPointcut`**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;

import org.springframework.aop.support.StaticMethodMatcherPointcut;

public class TestPointcut extends StaticMethodMatcherPointcut {
    @Override
    public boolean matches(Method method, Class cls) {
        return ("advised".equals(method.getName()));
    }
}
```

В листинге 5.42 показан код класса `NoOpBeforeAdvice`, который представляет собой простой совет "перед", не имеющий операций.

**Листинг 5.42. Класс `NoOpBeforeAdvice`**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class NoOpBeforeAdvice implements MethodBeforeAdvice {
    @Override
    public void before(Method method, Object[] args, Object target)
        throws Throwable {
        // операции отсутствуют
    }
}
```

Наконец, в листинге 5.43 приведен код для теста производительности.

**Листинг 5.43. Тестирование производительности прокси**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.Advisor;
import org.springframework.aop.framework.Advised;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;

public class ProxyPerfTest {
   public static void main(String[] args) {
      SimpleBean target = new DefaultSimpleBean();

      Advisor advisor = new DefaultPointcutAdvisor(new TestPointcut(),
            new NoOpBeforeAdvice());

      runCglibTests(advisor, target);
      runCglibFrozenTests(advisor, target);
      runJdkTests(advisor, target);
   }

   private static void runCglibTests(Advisor advisor, SimpleBean target) {
      ProxyFactory pf = new ProxyFactory();
      pf.setProxyTargetClass(true);
      pf.setTarget(target);
      pf.addAdvisor(advisor);

      SimpleBean proxy = (SimpleBean)pf.getProxy();
      // Запуск тестов для стандарного прокси CGLIB
      System.out.println("Running CGLIB (Standard) Tests");
      test(proxy);
   }

   private static void runCglibFrozenTests(Advisor advisor, SimpleBean target)
   {
      ProxyFactory pf = new ProxyFactory();
      pf.setProxyTargetClass(true);
      pf.setTarget(target);
      pf.addAdvisor(advisor);
      pf.setFrozen(true);

      SimpleBean proxy = (SimpleBean)pf.getProxy();
      // Запуск тестов для прокси CGLIB с цепочкой фиксированного совета
      System.out.println("Running CGLIB (Frozen) Tests");
      test(proxy);
   }

   private static void runJdkTests(Advisor advisor, SimpleBean target) {
      ProxyFactory pf = new ProxyFactory();
      pf.setTarget(target);
      pf.addAdvisor(advisor);
      pf.setInterfaces(new Class[]{SimpleBean.class});

      SimpleBean proxy = (SimpleBean)pf.getProxy();
      // Запуск тестов для прокси JDK
      System.out.println("Running JDK Tests");
      test(proxy);
   }
```

```
    private static void test(SimpleBean bean) {
        long before = 0;
        long after = 0;

        // Тестирование метода, снабженного советом
        System.out.println("Testing Advised Method");
        before = System.currentTimeMillis();
        for(int x = 0; x < 500000; x++) {
            bean.advised();
        }
        after = System.currentTimeMillis();

        // Вывод времени выполнения метода
        System.out.println("Took " + (after - before) + " ms");

        // Тестирование метода, не снабженного советом
        System.out.println("Testing Unadvised Method");
        before = System.currentTimeMillis();
        for(int x = 0; x < 500000; x++) {
            bean.unadvised();
        }
        after = System.currentTimeMillis();

        // Вывод времени выполнения метода
        System.out.println("Took " + (after - before) + " ms");

        // Тестирование метода equals()
        System.out.println("Testing equals() Method");
        before = System.currentTimeMillis();
        for(int x = 0; x < 500000; x++) {
            bean.equals(bean);
        }
        after = System.currentTimeMillis();

        // Вывод времени выполнения метода
        System.out.println("Took " + (after - before) + " ms");

        // Тестирование метода hashCode()
        System.out.println("Testing hashCode() Method");
        before = System.currentTimeMillis();
        for(int x = 0; x < 500000; x++) {
            bean.hashCode();
        }
        after = System.currentTimeMillis();

        // Вывод времени выполнения метода
        System.out.println("Took " + (after - before) + " ms");

        Advised advised = (Advised)bean;

        // Тестирование метода Advised.getProxyTargetClass()
        System.out.println("Testing Advised.getProxyTargetClass() Method");
        before = System.currentTimeMillis();
        for(int x = 0; x < 500000; x++) {
            advised.getTargetClass();
        }
        after = System.currentTimeMillis();

        // Вывод времени выполнения метода
        System.out.println("Took " + (after - before) + " ms");

        System.out.println(">>>\n");
    }
}
```

В этом коде мы тестируем три вида прокси: стандартный прокси CGLIB, прокси CGLIB с цепочкой фиксированного совета (т.е. когда прокси зафиксирован вызовом метода `setFrozen()` в классе `ProxyConfig`, который класс `ProxyFactory` косвенно расширяет, библиотека CGLIB предпримет дальнейшую оптимизацию, но последующее изменение совета не разрешено) и прокси JDK. Для каждого типа прокси запускаются следующие пять тестовых сценариев.

- **Метод, снабженный советом (тест 1).** Метод, который снабжен советом. В этом тесте используется совет "перед", не предусматривающий какую-либо обработку, что снижает влияние совета на тесты производительности.
- **Метод, не снабженный советом (тест 2).** Метод прокси, который не снабжен советом. Часто прокси имеет много методов, не снабженных советом. Этот тест позволяет выяснить, насколько хорошо методы, не снабженные советом, выполняются для различных прокси.
- **Метод `equals()` (тест 3).** Данный тест позволяет выяснить накладные расходы, связанные с вызовом метода `equals()`. Это особенно важно, когда прокси применяются в качестве ключей в `HashMap` или подобной коллекции.
- **Метод `hashcode()` (тест 4).** Как и `equals()`, метод `hashCode()` важен, когда используются `HashMap` или похожие коллекции.
- **Выполнение методов интерфейса `Advised` (тест 5).** Как упоминалось ранее, прокси по умолчанию реализует интерфейс `Advised`, позволяя модифицировать прокси после создания и запрашивать информацию, связанную с прокси. Этот тест показывает, насколько быстро происходит обращение к методам интерфейса `Advised` с применением различных типов прокси.

Результаты прогона этих тестов приведены в табл. 5.4.

**Таблица 5.4. Результаты тестирования производительности прокси (значения указаны в миллисекундах)**

| | Прокси CGLIB (стандартный) | Прокси CGLIB (фиксированный) | Прокси JDK |
|---|---|---|---|
| Метод, снабженный советом | 245 | 135 | 224 |
| Метод, не снабженный советом | 92 | 42 | 78 |
| `equals()` | 9 | 6 | 77 |
| `hashCode()` | 29 | 13 | 23 |
| `Advised.getProxyTargetClass()` | 9 | 6 | 15 |

Как видите, производительность стандартного прокси CGLIB и динамического прокси JDK для методов, снабженных и не снабженных советом, не сильно отличается. Как обычно, получаемые результаты будут варьироваться в зависимости от оборудования и версии JDK.

Однако есть заметная разница при использовании прокси CGLIB с цепочкой фиксированного совета. Похожая ситуация имеет место также с методами `equals()` и `hashCode()`, которые оказываются значительно быстрее, когда применяется прокси CGLIB. Методы интерфейса `Advised` также быстрее в случае фиксирован-

ного прокси CGLIB. Это объясняется тем, что методы `Advised` обрабатываются раньше в методе `intercept()`, поэтому для них не выполняется большая часть логики, которая требуется остальным методам.

### Выбор прокси для использования

Принимать решение о том, какой прокси использовать, обычно легко. Прокси CGLIB предназначен как для классов, так и для интерфейсов, тогда как прокси JDK — только для интерфейсов. В плане производительности нет заметной разницы между прокси JDK и прокси CGLIB в стандартном режиме (во всяком случае, при выполнении методов, снабженных и не снабженных советом). В фиксированном режиме прокси CGLIB цепочка совета не может быть изменена и CGLIB проводит дополнительную оптимизацию. Когда создается прокси для класса, по умолчанию устанавливается прокси CGLIB, т.к. это единственный прокси, который может быть сгенерирован для класса. Чтобы использовать прокси CGLIB для интерфейса, понадобится установить флаг `optimize` класса `ProxyFactory` в `true` с помощью метода `setOptimize()`.

## Расширенное использование срезов

Ранее в этой главе мы взглянули на шесть базовых реализаций интерфейса `Pointcut`, предоставляемых Spring; по большей части, они удовлетворяют потребностям многих приложений. Тем не менее, иногда возникает необходимость в большей гибкости при определении срезов. В Spring доступны две дополнительных реализации `Pointcut`, `ComposablePointcut` и `ControlFlowPointcut`, которые как раз и предоставляют нужную гибкость.

### Использование срезов потока управления

Срезы потока управления Spring, реализованные классом `ControlFlowPointcut`, подобны конструкции `cflow`, доступной во многих других реализациях АОП, хотя они не настолько мощные. В сущности, срез потока управления в Spring перехватывает все вызовы методов, выполненные из заданного метода либо из всех методов в классе. Это довольно трудно представить визуально, поэтому рассмотрим, как работает такой срез, на простом примере.

В листинге 5.44 показан код класса `SimpleBeforeAdvice`, который выводит описание метода, снабжаемого советом.

**Листинг 5.44. Класс `SimpleBeforeAdvice`**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class SimpleBeforeAdvice implements MethodBeforeAdvice {
    @Override
    public void before(Method method, Object[] args, Object target)
            throws Throwable {
        System.out.println("Before method: " + method);
    }
}
```

Этот класс совета позволяет увидеть, какой метод попадает в срез `ControlFlowPointcut`. В листинге 5.45 представлен простой класс с одним методом — методом, который необходимо снабдить советом.

**Листинг 5.45. Класс `TestBean`**

```
package com.apress.prospring4.ch5;

public class TestBean {
    public void foo() {
        System.out.println("foo()");
    }
}
```

Вы видите простой метод `foo()`, который нужно снабдить советом. Однако при этом должно быть удовлетворено специальное требование — данный метод необходимо снабдить советом, только когда он вызывается из другого указанного метода. В листинге 5.46 приведен код простой тестовой программы для рассматриваемого примера.

**Листинг 5.46. Использование класса `ControlFlowPointcut`**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.Advisor;
import org.springframework.aop.Pointcut;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.ControlFlowPointcut;
import org.springframework.aop.support.DefaultPointcutAdvisor;

public class ControlFlowExample {
    public static void main(String[] args) {
        ControlFlowExample ex = new ControlFlowExample();
        ex.run();
    }
    public void run() {
        TestBean target = new TestBean();

        Pointcut pc = new ControlFlowPointcut(ControlFlowExample.class, "test");
        Advisor advisor = new DefaultPointcutAdvisor(pc, new SimpleBeforeAdvice());
        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        TestBean proxy = (TestBean) pf.getProxy();
        // Попытка нормального вызова
        System.out.println("Trying normal invoke");
        proxy.foo();
        // Попытка вызова из ControlFlowExample.test()
        System.out.println("Trying under ControlFlowExample.test()");
        test(proxy);
    }
    private void test(TestBean bean) {
        bean.foo();
    }
}
```

В листинге 5.46 снабженный советом прокси связывается с `ControlFlowPointcut` и затем метод `foo()` вызывается дважды: один раз напрямую из метода `run()` и один раз из метода `test()`. Особый интерес представляет следующая строка кода:

```
Pointcut pc = new ControlFlowPointcut(ControlFlowExample.class, "test");
```

Здесь создается экземпляр `ControlFlowPointcut` для метода `test()` класса `ControlFlowExample`. В сущности, она говорит: "Включить в срез все методы, вызванные из метода `ControlFlowExample.test()`". Обратите внимание, что хотя формулировка выглядит как "Включить в срез все методы", на самом деле она означает "Включить в срез все методы объекта прокси, который снабжен советом с использованием `Advisor`, соответствующего этому экземпляру `ControlFlowPointcut`".

Запуск на выполнение примера из листинга 5.46 дает следующий вывод:

```
Trying normal invoke
foo()
Trying under ControlFlowExample.test()
Before method: public void com.apress.prospring4.ch5.TestBean.foo()
foo()
```

Как видите, когда метод `foo()` вызывается в первый раз за пределами потока управления метода `test()`, он не снабжен советом. Когда `foo()` вызывается во второй раз, уже внутри потока управления метода `test()`, экземпляр `ControlFlowPointcut` показывает, что связанный с ним совет применяется к `foo()`, поэтому метод превращается в снабженный советом. Обратите внимание, что если бы мы вызвали внутри метода `test()` еще один метод, не относящийся к прокси, он не был бы снабжен советом.

Срезы потока управления могут быть исключительно полезны, позволяя снабжать объект советом выборочно, только когда объект выполняется в контексте другого объекта. Однако помните, что применение среза потока управления связано с существенным снижением производительности по сравнению с другими видами срезов.

Давайте рассмотрим пример. Предположим, что имеется система обработки транзакций, которая поддерживает интерфейсы `TransactionService` и `AccountService`. Мы хотим применить совет "после", чтобы в ситуации, когда метод `AccountService.updateBalance()` вызывается методом `TransactionService.reverseTransaction()`, после обновления баланса на счету клиенту отправлялось уведомление по электронной почте. Однако ни при каких других обстоятельствах уведомление посылаться не должно. В этом случае удобно воспользоваться срезом потока управления. На рис. 5.4 показана диаграмма последовательностей UML для описанного сценария.

## Использование компонуемого среза

В предыдущих примерах создания срезов мы применяли по одному срезу для каждого экземпляра `Advisor`. В большинстве случаев этого вполне достаточно, но в некоторых ситуациях для достижения намеченной цели может понадобиться скомпоновать вместе два среза или даже больше.

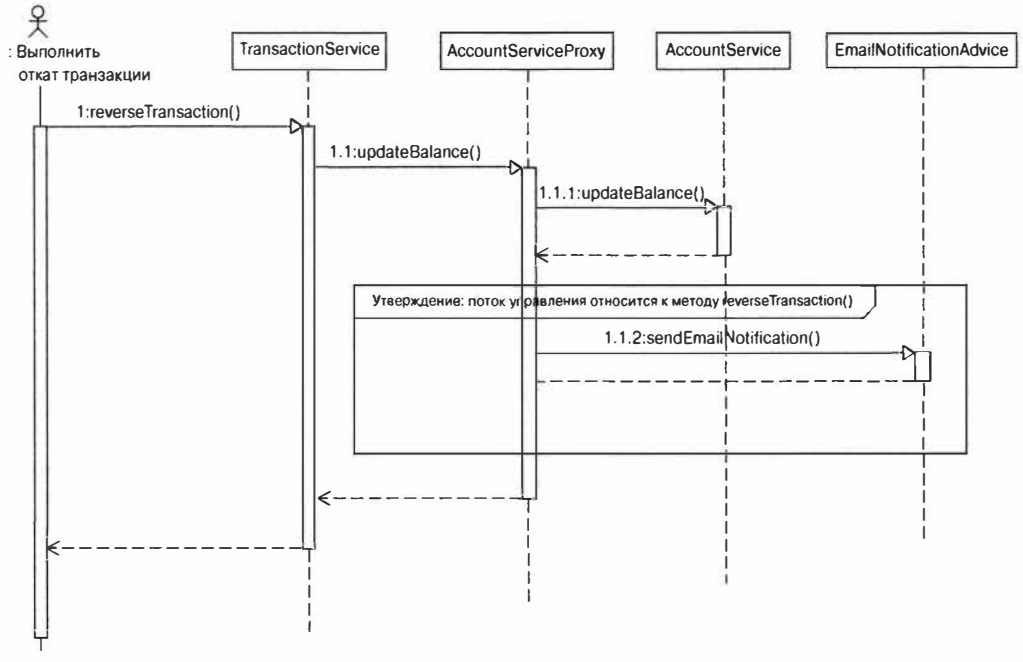


**Рис. 5.4.** Диаграмма последовательностей UML для среза потока управления

Предположим, что в срез необходимо включить все методы получения и установки из класса бина. У вас есть срез для методов получения и срез для методов установки, но нет среза для обеих разновидностей методов. Разумеется, можно было бы создать еще один срез с новой логикой, но более удачный подход предполагает комбинирование упомянутых двух срезов в единый срез с использованием `ComposablePointcut`.

Класс `ComposablePointcut` поддерживает два метода: `union()` и `intersection()`. По умолчанию `ComposablePointcut` создается с реализацией интерфейса `ClassFilter`, которая соответствует всем классам, и реализацией интерфейса `MethodMatcher`, которая соответствует всем методам, хотя при конструировании можно указать собственные начальные `ClassFilter` и `MethodMatcher`. Методы `union()` и `intersection()` перегружены для принятия аргументов `ClassFilter` и `MethodMatcher`.

Метод `ComposablePointcut.union()` может быть вызван за счет передачи экземпляра реализации одного из интерфейсов `ClassFilter`, `MethodMatcher` или `Pointcut`. Результат операции объединения заключается в том, что `ComposablePointcut` добавит условие "или" в свою цепочку вызова для поиска соответствия срезам.

То же касается и метода `ComposablePointcut.intersection()`, но в этом случае вместо условия "или" добавляется условие "и", которое означает, что все определения `ClassFilter`, `MethodMatcher` и `Pointcut` внутри `ComposablePointcut` должны давать соответствие, чтобы совет был применен. Это удобно представлять как конструкцию `WHERE` в SQL-запросе, причем метод `union()` похож на операцию `OR`, а метод `intersection()` — на операцию `AND`.

Как и в случае со срезами потока управления, это сложно воспроизвести визуально, так что намного проще рассмотреть на примере. В листинге 5.47 показан простой бин с тремя методами.

**Листинг 5.47. Класс `SampleBean`**

```
package com.apress.prospring4.ch5;

public class SampleBean {
    public String getName() {
        return "Chris Schaefer";
    }

    public void setName(String name) {
    }

    public int getAge() {
        return 32;
    }
}
```

В этом примере мы собираемся сгенерировать три разных прокси, используя один и тот же экземпляр `ComposablePointcut`, но каждый раз модифицировать `ComposablePointcut` с применением метода `union()` или `intersection()`. После этого мы вызовем все три метода на прокси `SampleBean` и посмотрим, какие из них будут снабжены советом. В листинге 5.48 приведен необходимый код.

**Листинг 5.48. Исследование `ComposablePointcut`**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;

import org.springframework.aop.Advisor;
import org.springframework.aop.ClassFilter;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.ComposablePointcut;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.aop.support.StaticMethodMatcher;

public class ComposablePointcutExample {
    public static void main(String[] args) {
        SampleBean target = new SampleBean();

        ComposablePointcut pc = new ComposablePointcut(ClassFilter.TRUE,
            new GetterMethodMatcher());

        // Тест 1
        System.out.println("Test 1");
        SampleBean proxy = getProxy(pc, target);
        testInvoke(proxy);

        // Тест 2
        System.out.println("Test 2");
        pc.union(new SetterMethodMatcher());
        proxy = getProxy(pc, target);
        testInvoke(proxy);
```

```
        // Тест 3
        System.out.println("Test 3");
        pc.intersection(new GetAgeMethodMatcher());
        proxy = getProxy(pc, target);
        testInvoke(proxy);
    }

    private static SampleBean getProxy(ComposablePointcut pc,
            SampleBean target) {
        Advisor advisor = new DefaultPointcutAdvisor(pc,
            new SimpleBeforeAdvice());

        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        return (SampleBean) pf.getProxy();
    }

    private static void testInvoke(SampleBean proxy) {
        proxy.getAge();
        proxy.getName();
        proxy.setName("Chris Schaefer");
    }

    private static class GetterMethodMatcher extends StaticMethodMatcher {
        @Override
        public boolean matches(Method method, Class<?> cls) {
            return (method.getName().startsWith("get"));
        }
    }

    private static class GetAgeMethodMatcher extends StaticMethodMatcher {
        @Override
        public boolean matches(Method method, Class<?> cls) {
            return "getAge".equals(method.getName());
        }
    }

    private static class SetterMethodMatcher extends StaticMethodMatcher {
        @Override
        public boolean matches(Method method, Class<?> cls) {
            return (method.getName().startsWith("set"));
        }
    }
}
```

Первый момент, который следует отметить в этом примере, касается набора из трех закрытых реализаций интерфейса `MethodMatcher`. Класс `GetterMethodMatcher` соответствует всем методам, имена которых начинаются на `get`. Он является стандартной реализацией `MethodMatcher`, которую мы используем для сборки `ComposablePointcut`. В связи с этим мы ожидаем, что первый цикл обращений к методам `SampleBean` приведет к снабжению советом только методов `getAge()` и `getName()`.

Класс `SetterMethodMatcher` соответствует всем методам, имена которых начинаются с `set`, и он комбинируется с `ComposablePointcut` посредством `union()`

для второго цикла обращений. В этой точке мы имеем объединение двух реализаций MethodMatcher: одна соответствует всем методам, начинающимся на get, а другая — всем методам, начинающимся на set. Здесь мы ожидаем, что будут снабжены советом все вызовы методов.

Класс GetAgeMethodMatcher является специализированным и соответствует только методу getAge(). Этот MethodMatcher комбинируется с ComposablePointcut с помощью intersection() для третьего цикла обращений. Поскольку GetAgeMethodMatcher скомпонован с применением intersection(), в третьем цикле обращений мы ожидаем снабжения советом только метода getAge(), т.к. он является единственным методом, который соответствует всем скомпонованным MethodMatcher.

Запуск этого примера на выполнение дает в результате следующий вывод:

```
Test 1
Before method: public int com.apress.prospring4.ch5.SampleBean.getAge()
Before method: public java.lang.String com.apress.prospring4.ch5.
SampleBean.getName()
Test 2
Before method: public int com.apress.prospring4.ch5.SampleBean.getAge()
Before method: public java.lang.String com.apress.prospring4.ch5.
SampleBean.getName()
Before method: public void com.apress.prospring4.ch5.SampleBean.
setName(java.lang.String)
Test 3
Before method: public int com.apress.prospring4.ch5.SampleBean.getAge()
```

Как и ожидалось, при первом цикле обращений к методам прокси советом были снабжены только методы getAge() и getName(). На втором цикле, когда SetterMethodMatcher был скомпонован посредством union(), советом оказались снабжены все методы. На третьем цикле в результате пересечения GetAgeMethodMatcher был снабжен советом один лишь метод getAge().

Хотя этот пример демонстрирует использование реализаций MethodMatcher только в процессе компоновки, применять ClassFilter при построении среза столь же просто. На самом деле при построении составного среза можно использовать комбинацию реализаций MethodMatcher и ClassFilter.

## Компоновка и интерфейс Pointcut

В предыдущем разделе было показано, как создавать составной срез с использованием множества реализаций MethodMatcher и ClassFilter. Создавать составные срезы можно также с помощью других объектов, которые реализуют интерфейс Pointcut.

Другой способ построения составного среза предполагает применение класса org.springframework.aop.support.Pointcuts. Этот класс предоставляет три статических метода. Методы intersection() и union() принимают два среза в качестве аргументов и конструируют составной срез. С другой стороны, доступен также метод matches(Pointcut, Method, Class, Object[]), предназначенный для выполнения быстрой проверки, соответствует ли срез предоставленному методу, классу и аргументам метода.

Класс `Pointcuts` поддерживает операции только с двумя срезами. Таким образом, если нужно скомбинировать `MethodMatcher` и `ClassFilter` с `Pointcut`, следует использовать класс `ComposablePointcut`. Однако когда необходимо скомбинировать два среза, класс `Pointcuts` более удобен.

### Резюме по созданию срезов

Платформа Spring предлагает мощный набор реализаций `Pointcut`, которые должны удовлетворять большинству, если не всем, требованиям приложения. Не забывайте, что когда не удается найти срез, подходящий для ваших потребностей, вы можете создать собственную реализацию с нуля, реализовав интерфейсы `Pointcut`, `MethodMatcher` и `ClassFilter`.

Для комбинирования срезов и советов предусмотрены два подхода. Первый подход, который применялся до сих пор, предполагает отделение реализации среза от совета. В показанном до настоящего момента коде мы создавали экземпляры реализаций `Pointcut` и затем использовали `DefaultPointcutAdvisor` для добавления совета вместе с `Pointcut` к прокси.

При втором подходе, который задействован во многих примерах в документации по Spring, `Pointcut` инкапсулируется внутри собственной реализации `Advisor`. Таким образом, имеется класс, который реализует `Pointcut` и `PointcutAdvisor`, с методом `PointcutAdvisor.getPointcut()`, просто возвращающим `this`. Этот подход используется многими классами Spring, такими как `StaticMethodMatcherPointcutAdvisor`.

Мы считаем первый подход более гибким, т.к. он позволяет применять разные реализации `Pointcut` с разными реализациями `Advisor`. Однако второй подход удобен в ситуациях, когда вы собираетесь использовать одну и ту же комбинацию `Pointcut` и `Advisor` в различных частях приложения или даже между разными приложениями. Второй подход также удобен в случае, если каждый `Advisor` должен иметь отдельный экземпляр `Pointcut`; вы достигаете этого, делая `Advisor` ответственным за создание `Pointcut`.

Вспомните из обсуждения производительности прокси, что методы, не снабженные советом, выполняются намного эффективнее, чем методы, снабженные советом. По данной причине вы должны удостовериться, что при использовании `Pointcut` вы снабжаете советом только те методы, для которых это абсолютно необходимо. Тем самым вы сократите объем ненужных накладных расходов, появляющихся в приложении из-за применения АОП.

## Начало работы с введениями

Введения представляют собой важную часть набора функциональных возможностей АОП, доступных в Spring. За счет использования введений можно динамически добавлять новую функциональность к существующему объекту. Платформа Spring позволяет вводить в существующий объект реализацию любого интерфейса. Возникает вполне закономерный вопрос: для чего может понадобиться добавление функциональности динамически во время выполнения, если ее легко добавить при проектировании? Ответ на этот вопрос прост. Функциональность добавляется динамически, если она является сквозной и ее довольно сложно реализовать с применением традиционного совета.

## Основы введений

В Spring *введения* трактуются как специальный тип совета, точнее — как специальный тип совета "вокруг". Поскольку введения применяются исключительно на уровне классов, использовать срезы с введениями нельзя; они не совпадают семантически. Введение добавляет к классу новые реализации интерфейсов, а срез определяет, к каким методам применяется совет. Введение создается путем реализации интерфейса `IntroductionInterceptor`, который расширяет интерфейсы `MethodInterceptor` и `DynamicIntroductionAdvice`. На рис. 5.5 показана эта структура вместе с методами обоих интерфейсов.

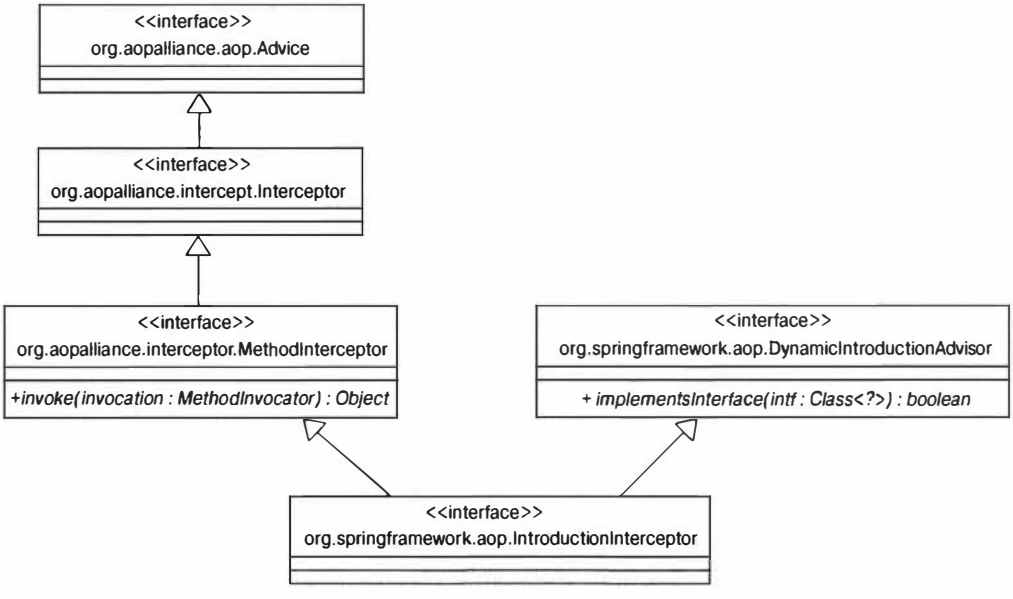


**Рис. 5.5.** Структура интерфейсов для введений

Как видите, в интерфейсе `MethodInterceptor` определен метод `invoke()`. С помощью этого метода вы предоставляете реализацию для вводимых интерфейсов и выполняете перехват любых дополнительных методов по мере необходимости. Реализация всех методов для интерфейса внутри одного метода может оказаться затруднительной и, скорее всего, приведет к большому объему кода, с которым придется иметь дело при решении, какой из методов вызывать. К счастью, Spring предоставляет стандартную реализацию `IntroductionInterceptor` по имени `DelegatingIntroductionInterceptor`, которая намного упрощает создание введений. Чтобы построить введение с использованием `DelegatingIntroductionInterceptor`, вы создаете класс, который унаследован от `DelegatingIntroductionInterceptor` и реализует интерфейсы, предназначенные для введения. Класс `DelegatingIntroductionInterceptor` затем просто делегирует все вызовы введенных методов соответствующему методу внутри себя. Не беспокойтесь, если это выглядит не особенно понятно; в следующем разделе будет приведен пример.

Точно так же, как необходимо использовать `PointcutAdvisor` при работе с советом среза, потребуется применять `IntroductionAdvisor` для добавления введений к прокси. Стандартной реализацией `IntroductionAdvisor` является `DefaultIntroductionAdvisor`, которая должна удовлетворять большинству, если не всем, потребностям введения. Учтите, что добавление введения с помощью `ProxyFactory.addAdvice()` не разрешено и приводит к генерации исключения `AopConfigException`. Вместо него должен использоваться метод `addAdvisor()` с передачей ему экземпляра реализации интерфейса `IntroductionAdvisor`.

В случае применения стандартного совета — т.е. не введений — возможно использование одного и того же экземпляра для множества разных объектов. В документации по Spring это называется *жизненным циклом на основе классов*, хотя одиночный экземпляр совета можно применять для многих классов. Совет типа введения формирует часть состояния объекта, снабженного советом, в результате чего для каждого такого объекта должен быть предусмотрен отдельный экземпляр совета. В документации это называется *жизненным циклом на основе экземпляров*. Так как необходимо гарантировать, что каждый снабженный советом объект имеет отдельный экземпляр введения, часто предпочтительнее создавать подкласс `DefaultIntroductionAdvisor`, отвечающий за создание совета типа введения. Таким образом, понадобится только обеспечить создание нового экземпляра класса совета для каждого объекта, потому что это автоматически создает новый экземпляр введения.

Например, пусть требуется применить совет "перед" к методу `setFirstName()` на всех экземплярах класса `Contact`. На рис. 5.6 показан один и тот же совет, который применяется ко всем объектам типа `Contact`.

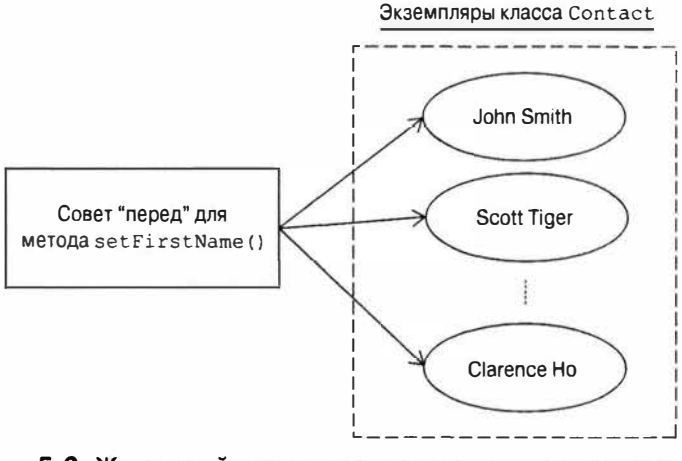


**Рис. 5.6.** Жизненный цикл на основе экземпляров для совета

А теперь предположим, что необходимо смешать введение во все экземпляры класса `Contact`, и это введение будет сопровождаться информацией для каждого экземпляра `Contact` (например, атрибутом `isModified`, который указывает, был ли экземпляр модифицирован). В таком случае введение будет создаваться для каждого экземпляра класса `Contact` и затем привязываться к этому экземпляру, как показано на рис. 5.7.

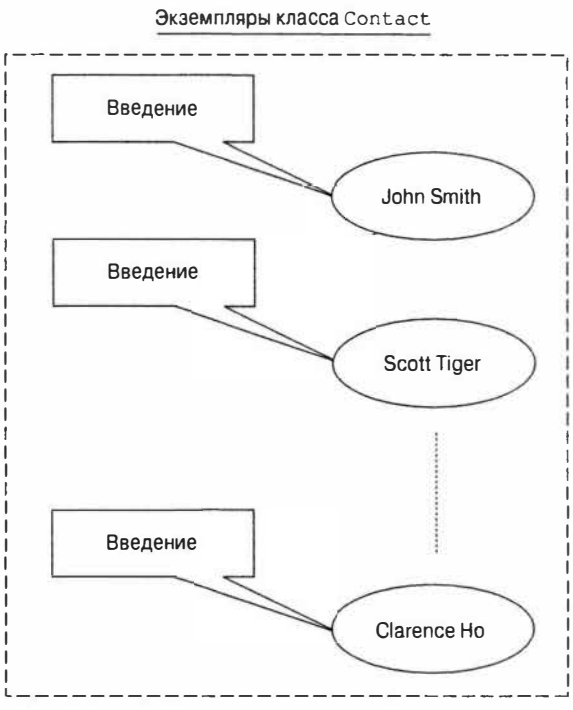


**Рис. 5.7.** Введение на основе экземпляров

На этом рассмотрение основ введений завершено. Далее мы обсудим, как использовать введения для решения задачи с обнаружением модификации объекта.

## Обнаружение модификации объекта с помощью введений

*Обнаружение модификации объекта* — полезный прием по многим причинам. Обычно обнаружение модификации применяется для предотвращения излишнего доступа к базе данных во время сохранения содержимого объекта. Если объект передается методу для модификации, но возвращается обратно без изменений, то не имеет смысла выполнять оператор обновления в базе данных. Использование проверки модификации подобным образом может увеличить пропускную способность приложения, особенно когда база данных уже работает с высокой нагрузкой или расположена в удаленной сети, увеличивая стоимость коммуникаций.

К сожалению, такой вид функциональности трудно реализовать вручную, потому что он требует добавления к каждому методу, который может модифицировать состояние объекта, кода для проверки, действительно ли состояние было изменено. Если подсчитать все проверки на `null` и проверки, изменилось ли значение, получится примерно восемь строк кода на метод. Конечно, можно вынести этот код в отдельный метод, но все равно данный метод придется вызывать каждый раз, когда необходимо выполнить проверку. Распространение такого кода по всему приложению с множеством классов, требующих проверки модификации — своего рода мина замедленного действия.

В описанной ситуации помогут введения. Мы не хотим, чтобы каждый класс, требующий проверок модификации, наследовался от какой-то базовой реализации, в результате теряя свой единственный шанс на наследование, равно как и не хотим добавлять код проверки ко всем методам, которые могут изменять состояние объекта. С помощью введений мы можем обеспечить гибкое решение задачи обнаружения модификации, не требующее написания множества повторяющегося и подверженного ошибкам кода.

В рассматриваемом примере мы собираемся построить полноценную инфраструктуру проверок модификации, используя введения. Логика проверки модификации инкапсулирована в интерфейсе `IsModified`, реализация которого будет введена в соответствующие объекты вместе с логикой перехвата для автоматического выполнения проверок модификации. Для целей данного примера мы применяем соглашения JavaBean, по которым считается, что модификация происходит при любом обращении к методу установки. Разумеется, мы не трактуем все вызовы метода установки как модификацию — мы проверяем, отличается ли значение, переданное методу установки, от текущего значения, хранящегося в объекте. Единственным недостатком этого решения является то, что возврат объекта в первоначальное состояние будет считаться модификацией, если какое-то одно из значений объекта изменилось во время обработки. Например, пусть имеется объект `Contact` с атрибутом `firstName`. Предположим, что во время обработки атрибут `firstName` изменил свое значение с Peter на John. В результате объект помечается как модифицированный. Однако он все равно будет помечен как модифицированный, даже если при последующей обработке значение атрибута вернется обратно к исходному состоянию Peter. Один из способов учета такой ситуации предусматривает сохранение полной хронологии изменений в рамках жизненного цикла объекта. Тем не менее, предлагаемая нами реализация нетривиальна и удовлетворяет большинство требований. Реализация более полного решения могла бы привести к чрезмерному усложнению примера.

### *Использование интерфейса `IsModified`*

Центральным компонентом в решении проверки модификаций является интерфейс `IsModified`, который в рассматриваемом примере приложения применяется для принятия интеллектуальных решений о сохранении содержимого объектов. Мы не будем касаться того, как приложение могло бы использовать `IsModified`, а вместо этого сосредоточим внимание на реализации введения. Код интерфейса `IsModified` показан в листинге 5.49.

**Листинг 5.49. Интерфейс `IsModified`**

```
package com.apress.prospring4.ch5;

public interface IsModified {
   boolean isModified();
}
```

Здесь нет ничего особо интересного — просто один метод `isModified()`, отражающий, был ли объект модифицирован.

### Создание смеси

Следующий шаг заключается в создании кода, который реализует интерфейс `IsModified` и будет введен в объекты; это называется *смесью* (mixin). Как упоминалось ранее, создавать смеси намного проще путем определения подклассов `DelegatingIntroductionInterceptor`, чем с помощью непосредственной реализации интерфейса `IntroductionInterceptor`. Наш класс смеси, `IsModifiedMixin`, является подклассом `DelegatingIntroductionInterceptor` и также реализует интерфейс `IsModified`. Код класса `IsModifiedMixin` приведен в листинге 5.50.

**Листинг 5.50. Класс `IsModifiedMixin`**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.Map;

import org.aopalliance.intercept.MethodInvocation;
import org.springframework.aop.support.DelegatingIntroductionInterceptor;

public class IsModifiedMixin extends DelegatingIntroductionInterceptor
        implements IsModified {
    private boolean isModified = false;

    private Map<Method, Method> methodCache = new HashMap<Method, Method>();

    @Override
    public boolean isModified() {
        return isModified;
    }

    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        if (!isModified) {
            if ((invocation.getMethod().getName().startsWith("set"))
                && (invocation.getArguments().length == 1)) {

                Method getter = getGetter(invocation.getMethod());

                if (getter != null) {
                    Object newVal = invocation.getArguments()[0];
                    Object oldVal = getter.invoke(invocation.getThis(), null);

                    if((newVal == null) && (oldVal == null)) {
                        isModified = false;
                    } else if((newVal == null) && (oldVal != null)) {
                        isModified = true;
                    } else if((newVal != null) && (oldVal == null)) {
                        isModified = true;
                    } else {
                        isModified = (!newVal.equals(oldVal));
                    }
                }
            }
        }
        return super.invoke(invocation);
    }
```

```
    private Method getGetter(Method setter) {
        Method getter = null;

        getter = (Method) methodCache.get(setter);

        if (getter != null) {
            return getter;
        }

        String getterName = setter.getName().replaceFirst("set", "get");

        try {
            getter = setter.getDeclaringClass().getMethod(getterName, null);

            synchronized (methodCache) {
                methodCache.put(setter, getter);
            }

            return getter;
        } catch (NoSuchMethodException ex) {
            return null;
        }
    }
}
```

Первым моментом, который следует отметить в коде, является реализация интерфейса `IsModified`, которая состоит из закрытого поля `isModified` и метода `isModified()`. Этот пример подчеркивает главную причину, по которой должен существовать только один экземпляр смеси на объект, снабженный советом — смесь вводит в объект не только методы, но также и состояние. Совместное использование единственного экземпляра этой смеси множеством объектов приведет также к совместному использованию его состояния, и в результате окажется, что после первого же изменения какого-то одного объекта модифицированными будут считаться все объекты.

В действительности реализация метода `invoke()` для смеси не является обязательной, но в данном случае она позволяет обнаруживать модификацию автоматически. Мы начинаем с выполнения проверки, только если объект пока еще не изменялся; проверять на предмет модификаций объект, в отношении которого известно, что он изменялся, необходимости нет. Затем мы выясняем, является ли метод установщиком, и если это так, извлекаем соответствующий метод получателя. Обратите внимание, что пара получателя/установщика кешируется для ускорения будущих извлечений. Наконец, мы сравниваем значение, возвращенное получателем, со значением, переданным установщику, чтобы определить, произошла ли модификация. При этом выполняются проверки с возможными комбинациями значения `null` с соответствующей установкой `isModified`. Важно помнить, что в случае использования `DelegatingIntroductionInterceptor` при переопределении метода `invoke()` потребуется вызвать `super.invoke()`, потому что `DelegatingIntroductionInterceptor` направляет обращения корректному адресату — либо объекту, снабженному советом, либо самой смеси.

В смеси можно реализовать произвольное количество интерфейсов, и каждый из них будет автоматически введен в объект, снабженный советом.

### Создание аспекта

На следующем шаге создается `Advisor`, служащий оболочкой для создания класса смеси. Этот шаг не обязателен, но он поможет обеспечить применение нового экземпляра класса смеси для каждого объекта, снабженного советом. Код класса `IsModifiedAdvisor` показан в листинге 5.51.

**Листинг 5.51. Создание совета для смеси**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.support.DefaultIntroductionAdvisor;

public class IsModifiedAdvisor extends DefaultIntroductionAdvisor {
    public IsModifiedAdvisor() {
        super(new IsModifiedMixin());
    }
}
```

Обратите внимание, что для создания `IsModifiedAdvisor` был расширен класс `DefaultIntroductionAdvisor`. Реализация совета довольно проста и не требует дополнительных пояснений.

### Собираем все вместе

Теперь, имея класс смеси и класс `Advisor`, мы можем протестировать инфраструктуру проверки модификации. В листинге 5.52 приведен простой класс, предназначенный для тестирования `IsModifiedMixin`.

**Листинг 5.52. Класс `TargetBean`**

```
package com.apress.prospring4.ch5;

public class TargetBean {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Этот бин имеет единственное свойство `name`, которое мы используем при тестировании смеси проверки модификации. В листинге 5.53 показано, как собрать прокси для совета и затем провести тестирование кода проверки модификаций.

**Листинг 5.53. Использование `IsModifiedMixin`**

```
package com.apress.prospring4.ch5;

import org.springframework.aop.IntroductionAdvisor;
import org.springframework.aop.framework.ProxyFactory;

public class IntroductionExample {
```

```
    public static void main(String[] args) {
        TargetBean target = new TargetBean();
        target.setName("Chris Schaefer");

        IntroductionAdvisor advisor = new IsModifiedAdvisor();

        ProxyFactory pf = new ProxyFactory();
        pf.setTarget(target);
        pf.addAdvisor(advisor);
        pf.setOptimize(true);

        TargetBean proxy = (TargetBean) pf.getProxy();
        IsModified proxyInterface = (IsModified)proxy;

        System.out.println("Is TargetBean?: " + (proxy instanceof TargetBean));
        System.out.println("Is IsModified?: " + (proxy instanceof IsModified));

        System.out.println("Has been modified?: " +
            proxyInterface.isModified());

        proxy.setName("Chris Schaefer");

        System.out.println("Has been modified?: " +
            proxyInterface.isModified());
        proxy.setName("Chris Schaefer");
        System.out.println("Has been modified?: " +
            proxyInterface.isModified());
    }
}
```

Обратите внимание, что при создании прокси флаг оптимизации устанавливается в `true`, чтобы обеспечить применение прокси CGLIB. Причина в том, что если выбрать для введения смеси прокси JDK, то результирующий прокси не будет экземпляром класса (в данном случае `TargetBean`); такой прокси реализует только интерфейсы смеси, но не исходный класс. В случае прокси CGLIB исходный класс реализуется прокси наряду с интерфейсами смеси.

В коде сначала производится проверка, что прокси является экземпляром `TargetBean`, а затем — что это экземпляр `IsModified`. Обе проверки возвращают `true`, если используется прокси CGLIB, но для прокси JDK значение `true` дает только проверка `IsModified`. Наконец, мы тестируем код проверки модификаций, сначала устанавливая свойство `name` в его текущее значение, а потом — в новое значение, каждый раз проверяя значение флага `isModified`. Запуск этого примера дает в результате следующий вывод:

```
Is TargetBean?: true
Is IsModified?: true
Has been modified?: false
Has been modified?: false
Has been modified?: true
```

Как и ожидалось, обе проверки `instanceof` возвращают `true`. Обратите внимание, что первый вызов `isModified()`, до возникновения всех модификаций, возвращает `false`. Следующий вызов, после установки свойства `name` в то же самое значение, также возвращает `false`. Однако финальный вызов метода `isModified()`, после того, как `name` установлено в новое значение, возвращает `true`, указывая на то, что объект на самом деле был изменен.

### Резюме по введениям

Введения являются одним из наиболее мощных средств АОП в Spring; они позволяют не только расширять функциональность существующих методов, но также динамически расширять набор интерфейсов и реализаций объектов. Использование введений — это великолепный способ реализации сквозной логики, с которой приложение взаимодействует через четко определенные интерфейсы. В общем, это такая разновидность логики, которую желательно применять декларативно, а не программно. Используя класс `IsModifiedMixin`, построенный в рассмотренном примере, и службы платформы, обсуждаемые в следующем разделе, мы можем декларативно определить, какие объекты обладают возможностью проверки модификаций, не изменяя реализации этих объектов.

Очевидно, что поскольку введения работают через прокси, они добавляют определенный объем накладных расходов. Все методы прокси считаются снабженными советом, т.к. применять срезы в сочетании с введениями не допускается. Тем не менее, учитывая многочисленность служб, которые можно реализовать с помощью введений (включая проверку модификации объекта), накладные расходы, связанные с производительностью, являются лишь небольшой платой за сокращение объема кода, требуемого для реализации служб, а также за повышенную устойчивость и улучшенные возможности сопровождения, обеспечиваемые централизацией логики служб.

## Службы платформы, предназначенные для АОП

К этому моменту мы написали немало кода, снабжающего объекты советами и генерирующего для них прокси. Хотя само по себе это не является большой проблемой, но все же означает, что вся конфигурация советов жестко закодирована в приложении, ликвидируя некоторые преимущества возможности прозрачного снабжения советом реализации метода. К счастью, Spring предоставляет дополнительные службы платформы, позволяющие создать прокси для совета в конфигурации приложения и затем внедрить этот прокси в целевой бин подобно любым другим зависимостям.

Использование декларативного подхода к конфигурации АОП предпочтительнее ручного программного подхода. Применение декларативного механизма не только позволяет вынести наружу конфигурацию совета, но и снижает шансы на допущение ошибок при кодировании. Можно также получить преимущество от совместного использования внедрения зависимостей и АОП, обеспечивая, в конечном счете, полностью прозрачную среду.

### Декларативное конфигурирование АОП

Для декларативного конфигурирования АОП в Spring доступны три опции.

- **Использование `ProxyFactoryBean`.** В АОП, реализованном платформой Spring, класс `ProxyFactoryBean` предоставляет декларативный способ конфигурирования `ApplicationContext` (и, следовательно, лежащего в основе `BeanFactory`) при создании прокси АОП на основе определенных бинов Spring.

- **Использование пространства имен aop в Spring.** Появившееся в версии Spring 2.0 пространство имен aop предоставляет упрощенный (по сравнению с ProxyFactoryBean) способ определения аспектов и их требований DI в Spring-приложениях. Тем не менее, пространство имен aop "за кулисами" использует ProxyFactoryBean.
- **Использование аннотация в стиле @AspectJ.** Кроме пространства имен aop, основанного на XML, для конфигурирования АОП внутри классов можно также применять аннотации в стиле @AspectJ. Хотя используемый в этом случае синтаксис основан на AspectJ и понадобится включить ряд библиотек AspectJ, платформа Spring по-прежнему применяет механизм прокси (т.е. создает прокси-объекты для целей) во время начальной загрузки ApplicationContext.

## Использование ProxyFactoryBean

Класс ProxyFactoryBean — это реализация FactoryBean, позволяющая указать целевой бин и предоставляющая для этого бина набор советов и аспектов, которые впоследствии объединяются в прокси АОП. Поскольку с ProxyFactoryBean можно использовать совет и аспект, декларативное конфигурирование доступно не только для совета, но также и для срезов.

Класс ProxyFactoryBean разделяет общий интерфейс (org.springframework.aop.framework.Advised) с ProxyFactory (оба класса косвенно расширяют класс org.springframework.aop.framework.AdvisedSupport, который реализует интерфейс Advised), в результате чего в нем доступны многие из тех же самых флагов, такие как frozen, optimize и exposeProxy. Значения для этих флагов передаются прямо лежащему в основе классу ProxyFactory, который также позволяет декларативно конфигурировать фабрику.

### Класс ProxyFactoryBean в действии

Работать с классом ProxyFactoryBean очень просто. Вы определяете бин, который будет целевым, затем используете ProxyFactoryBean и определяете бин, к которому приложение будет получать доступ, применяя целевой бин в качестве цели прокси. Где только возможно, определяйте целевой бин как анонимный бин внутри объявления бина прокси. Это предотвратит приложение от случайного доступа к бину, не снабженному советом. Однако в ряде случаев (вроде рассматриваемого здесь примера) может потребоваться создание более одного прокси для того же самого бина, и тогда должен использоваться нормальный бин верхнего уровня. В листингах 5.54 и 5.55 показаны два класса, один из которых зависит от другого.

**Листинг 5.54. Класс MyDependency**

```
package com.apress.prospring4.ch5;
public class MyDependency {
    public void foo() {
        System.out.println("foo()");
    }
    public void bar() {
        System.out.println("bar()");
    }
}
```

**Листинг 5.55. Класс MyBean**

```
package com.apress.prospring4.ch5;

public class MyBean {
   private MyDependency dep;

   public void execute() {
      dep.foo();
      dep.bar();
   }

   public void setDep(MyDependency dep) {
      this.dep = dep;
   }
}
```

В этом примере мы собираемся создать два прокси для одного экземпляра MyDependency, оба с тем же самым базовым советом, который приведен в листинге 5.56.

**Листинг 5.56. Класс MyAdvice**

```
package com.apress.prospring4.ch5;

import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class MyAdvice implements MethodBeforeAdvice {
   @Override
   public void before(Method method, Object[] args, Object target)
         throws Throwable {
      System.out.println("Executing: " + method);
   }
}
```

Первый прокси будет просто снабжать целевой объект советом напрямую; таким образом, будут снабжены все методы. Для второго прокси мы сконфигурируем AspectJExpressionPointcut и DefaultPointcutAdvisor так, чтобы советом снабжался только метод foo() класса MyDependency. Чтобы протестировать совет, мы создадим два определения бинов типа MyBean, каждое из которых будет внедрено с помощью отличающегося прокси. Затем мы вызовем метод execute() на каждом из этих бинов и посмотрим, что происходит при вызове методов, снабженных советом, на зависимости. В листинге 5.57 показана конфигурация для этого примера (app-context-xml.xml).

**Листинг 5.57. Декларативная конфигурация АОП**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
    <bean id="myBean1" class="com.apress.prospring4.ch5.MyBean">
        <property name="dep">
            <ref bean="myDependency1"/>
        </property>
    </bean>

    <bean id="myBean2" class="com.apress.prospring4.ch5.MyBean">
        <property name="dep">
            <ref bean="myDependency2"/>
        </property>
    </bean>

    <bean id="myDependencyTarget"
          class="com.apress.prospring4.ch5.MyDependency"/>

    <bean id="myDependency1"
          class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target">
            <ref bean="myDependencyTarget"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>advice</value>
            </list>
        </property>
    </bean>

    <bean id="myDependency2"
          class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target">
            <ref bean="myDependencyTarget"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>advisor</value>
            </list>
        </property>
    </bean>

    <bean id="advice" class="com.apress.prospring4.ch5.MyAdvice"/>

    <bean id="advisor"
          class="org.springframework.aop.support.DefaultPointcutAdvisor">
        <property name="advice">
            <ref bean="advice"/>
        </property>
        <property name="pointcut">
            <bean class="org.springframework.aop.aspectj.AspectJExpressionPointcut">
                <property name="expression">
                    <value>execution(* foo*(..))</value>
                </property>
            </bean>
        </property>
    </bean>
</beans>
```

В этом примере мы просто устанавливаем свойства, которые устанавливали в коде с применением возможностей DI в Spring. Единственный примечательный момент связан с использованием анонимного бина для среза и класса `ProxyFactoryBean`. Мы предпочитаем применять анонимные бины для срезов, когда они не являются совместно используемыми, поскольку это позволяет удерживать набор бинов, к которым разрешен прямой доступ, насколько возможно малым и имеющим отношение к приложению. Применяя `ProxyFactoryBean`, важно понимать, что объявление `ProxyFactoryBean` будет видно приложению, и оно будет использоваться при удовлетворении зависимостей. Объявление лежащего в основе целевого бина не снабжается советом, так что данный бин должен применяться, только когда нужно обойти инфраструктуру АОП, хотя в общем случае приложение не обязано быть осведомленным об инфраструктуре АОП и, следовательно, пытаться обойти ее. По этой причине везде, где только возможно, должны использоваться анонимные бины, чтобы избежать случайного доступа к ним со стороны приложения.

В листинге 5.58 приведен код простого класса, который получает два экземпляра `MyBean` из `ApplicationContext` и затем вызывает для каждого из них метод `execute()`.

**Листинг 5.58. Класс `ProxyFactoryBeanExample`**

```
package com.apress.prospring4.ch5;

import org.springframework.context.support.GenericXmlApplicationContext;

public class ProxyFactoryBeanExample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx =
            new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        MyBean bean1 = (MyBean)ctx.getBean("myBean1");
        MyBean bean2 = (MyBean)ctx.getBean("myBean2");

        System.out.println("Bean 1");
        bean1.execute();

        System.out.println("\nBean 2");
        bean2.execute();
    }
}
```

Запуск примера из листинга 5.58 дает в результате следующий вывод:

```
Bean 1
Executing: public void com.apress.prospring4.ch5.MyDependency.foo()
foo()
Executing: public void com.apress.prospring4.ch5.MyDependency.bar()
bar()

Bean 2
Executing: public void com.apress.prospring4.ch5.MyDependency.foo()
foo()
bar()
```

Как и ожидалось, методы `foo()` и `bar()` в первом прокси были снабжены советом, т.к. в их конфигурации никакие срезы не применялись. Однако во втором прокси был снабжен советом только метод `foo()`, потому что в конфигурации использовался срез.

### *Использование класса `ProxyFactoryBean` для введений*

Применение класса `ProxyFactoryBean` не ограничивается одним лишь снабжением объекта советом; он также может использоваться для введения смесей в объекты. Вспомните из предшествующего обсуждения введений, что для добавления введения должен применяться интерфейс `IntroductionAdvisor`; добавлять введение напрямую нельзя. То же самое правило касается использования `ProxyFactoryBean` с введениями. Когда применяется `ProxyFactoryBean`, прокси намного проще конфигурировать, если создана специальная реализация `Advisor` для смеси. В листинге 5.59 показан пример конфигурации для введения `IsModifiedMixin`, которое рассматривалось ранее в главе (`app-context-xml.xml`).

**Листинг 5.59. Конфигурирование введений с помощью `ProxyFactoryBean`**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="bean"
      class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="target">
         <bean class="com.apress.prospring4.ch5.TargetBean">
            <property name="name">
               <value>Chris Schaefer</value>
            </property>
         </bean>
      </property>
      <property name="interceptorNames">
         <list>
            <value>advisor</value>
         </list>
      </property>
      <property name="proxyTargetClass">
         <value>true</value>
      </property>
   </bean>

   <bean id="advisor" class="com.apress.prospring4.ch5.IsModifiedAdvisor"/>
</beans>
```

В конфигурации видно, что класс `IsModifiedAdvisor` используется в качестве аспекта для `ProxyFactoryBean`, и поскольку не нужно создавать другой прокси для того же самого целевого объекта, мы применяем анонимное объявление для целевого бина. В листинге 5.60 представлено изменение предыдущего примера введения, в котором прокси получается из `ApplicationContext`.

**Листинг 5.60. Класс `IntroductionConfigExample`**

```
package com.apress.prospring4.ch5;

import org.springframework.context.support.GenericXmlApplicationContext;

public class IntroductionConfigExample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx =
            new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        TargetBean bean = (TargetBean) ctx.getBean("bean");
        IsModified mod = (IsModified) bean;

        System.out.println("Is TargetBean?: " + (bean instanceof TargetBean));
        System.out.println("Is IsModified?: " + (bean instanceof IsModified));

        System.out.println("Has been modified?: " + mod.isModified());
        bean.setName("Chris Schaefer");

        System.out.println("Has been modified?: " + mod.isModified());
        bean.setName("Chris Schaefer");

        System.out.println("Has been modified?: " + mod.isModified());
    }
}
```

Запуск этого примера на выполнение дает в точности такой же вывод, как и предыдущий пример введения, но на этот прокси получается из `ApplicationContext` и в коде приложения отсутствует какая-либо конфигурация.

### *Резюме по классу `ProxyFactoryBean`*

При использовании класса `ProxyFactoryBean` можно конфигурировать прокси АОП, что обеспечивает гибкость программного метода без необходимости в привязке кода приложения к конфигурации АОП. Если только не требуется принимать решения во время выполнения относительно того, как должны создаваться прокси, то лучше придерживаться декларативного, а не программного подхода к конфигурированию прокси. А теперь давайте перейдем к рассмотрению еще двух вариантов для декларативного конфигурирования АОП в Spring, которые являются рекомендуемыми для приложений, основанных на Spring 2.0 и выше с JDK 5 или более новой версии.

## Использование пространства имен `aop`

Пространство имен `aop` предлагает существенно упрощенный синтаксис для декларативного конфигурирования АОП в Spring. Чтобы продемонстрировать его работу, мы воспользуемся предыдущим примером с `ProxyFactoryBean`, слегка модифицировав его.

В листингах 5.61 и 5.62 показаны несколько измененные классы `MyDependency` и `MyBean`.

**Листинг 5.61. Класс MyDependency**

```
package com.apress.prospring4.ch5;

public class MyDependency {
    public void foo(int intValue) {
        System.out.println("foo(int): " + intValue);
    }

    public void bar() {
        System.out.println("bar()");
    }
}
```

**Листинг 5.62. Класс MyBean**

```
package com.apress.prospring4.ch5;

public class MyBean {
    private MyDependency dep;

    public void execute() {
        dep.foo(100);
        dep.foo(101);
        dep.bar();
    }

    public void setDep(MyDependency dep) {
        this.dep = dep;
    }
}
```

В предыдущем листинге метод `foo()` класса `MyDependency` был модифицирован для принятия целочисленного значения в качестве аргумента. Кроме того, в классе `MyBean` метод `foo()` был вызван два раза с разными параметрами.

Давайте посмотрим, как выглядит класс совета. В листинге 5.63 приведен пересмотренный код класса `MyAdvice`.

**Листинг 5.63. Класс MyAdvice**

```
package com.apress.prospring4.ch5;

import org.aspectj.lang.JoinPoint;

public class MyAdvice {
    public void simpleBeforeAdvice(JoinPoint joinPoint) {
        System.out.println("Executing: " +
              joinPoint.getSignature().getDeclaringTypeName() + " "
              + joinPoint.getSignature().getName());
    }
}
```

Вы увидите, что класс совета больше не нуждается в реализации интерфейса `MethodBeforeAdvice`. Кроме того, совет "перед" принимает в качестве аргумента `JoinPoint`, но не метод, объект и аргументы. В действительности для класса совета этот аргумент является необязательным, так что данный метод может быть остав-

лен без аргументов. Тем не менее, если в совете необходим доступ к деталям среза (в этом случае мы хотим создать копию информации о вызывающем типе и имени метода), то аргумент должен приниматься. Когда аргумент для метода определен, Spring будет автоматически передавать методу срез для дальнейшей обработки.

В листинге 5.64 показана XML-конфигурация АОП в Spring с пространством имен `aop` (`app-context-xml.xml`).

**Листинг 5.64. Конфигурирование АОП в Spring с помощью пространства имен `aop`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:aop="http://www.springframework.org/schema/aop"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">
   <aop:config>
      <aop:pointcut id="fooExecution"
         expression="execution(* com.apress.prospring4.ch5..foo*(int))"/>
      <aop:aspect ref="advice">
         <aop:before pointcut-ref="fooExecution"
            method="simpleBeforeAdvice"/>
      </aop:aspect>
   </aop:config>
   <bean id="advice" class="com.apress.prospring4.ch5.MyAdvice"/>
   <bean id="myDependency"
      class="com.apress.prospring4.ch5.MyDependency"/>
   <bean id="myBean" class="com.apress.prospring4.ch5.MyBean">
      <property name="dep" ref="myDependency"/>
   </bean>
</beans>
```

Во-первых, мы должны объявить пространство имен `aop` в дескрипторе `<beans>`. Во-вторых, вся конфигурация АОП в Spring помещена в дескриптор `<aop:config>`. Внутри этого дескриптора мы можем определить срез, аспекты, советы и т.д., а также ссылаться на другие бины Spring обычным образом.

В предыдущей конфигурации был определен срез с идентификатором `fooExecution`. Выражение `"execution(* com.apress.prospring4.ch5..foo*(int))"` означает, что необходимо снабдить советом все методы с префиксом `foo` в классах, определенных в пакете `com.apress.prospring4.ch5` (включая все его подпакеты). Кроме того, метод `foo()` должен принимать один аргумент целочисленного типа. Далее в дескрипторе `<aop:aspect>` был объявлен аспект со ссылкой на класс совета как на бин Spring с идентификатором `advice`, который представляет собой класс `MyAdvice`. В атрибуте `pointcut-ref` указана ссылка на определенный срез с идентификатором `fooExecution`, а совет "перед" (объявленный с помощью дескриптора `<aop:before>`) реализован методом `simpleBeforeAdvice()` внутри бина совета.

В листинге 5.65 приведена тестовая программа.

**Листинг 5.65. Класс `AopNamespaceExample`**

```
package com.apress.prospring4.ch5;

import org.springframework.context.support.GenericXmlApplicationContext;

public class AopNamespaceExample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        MyBean myBean = (MyBean) ctx.getBean("myBean");
        myBean.execute();
    }
}
```

В этом примере мы просто инициализируем `ApplicationContext`, извлекаем бин обычным образом и вызываем его метод `execute()`. Запуск этой программы на выполнение дает следующий вывод:

```
Executing: com.apress.prospring4.ch5.MyDependency foo
foo(int): 100
Executing: com.apress.prospring4.ch5.MyDependency foo
foo(int): 101
bar()
```

Как видите, два вызова метода `foo()` были снабжены советом, но вызов метода `bar()` — нет. Пример работает в точности так, как ожидалось, и вы должны заметить, что его конфигурация значительно упрощена по сравнению с конфигурацией `ProxyFactoryBean`.

Давайте снова займемся переделкой предыдущего примера для более сложного случая. Предположим, что необходимо снабдить советом только методы бинов Spring с идентификаторами, начинающимися с `myDependency`, которые принимают целочисленный аргумент со значением, не равным 100. Чтобы запускать совет только когда значение аргумента не равно 100, мы должны слегка модифицировать совет. Переделанный класс `MyAdvice` показан в листинге 5.66.

**Листинг 5.66. Класс `MyAdvice` (переделанный для проверки аргумента)**

```
package com.apress.prospring4.ch5;

import org.aspectj.lang.JoinPoint;

public class MyAdvice {
    public void simpleBeforeAdvice(JoinPoint joinPoint, int intValue) {
        if (intValue != 100) {
            System.out.println("Executing: " +
                joinPoint.getSignature().getDeclaringTypeName() + " "
                + joinPoint.getSignature().getName()
                + " argument: " + intValue);
        }
    }
}
```

Изменения внесены в два места. Во-первых, к сигнатуре совета "перед" добавлен аргумент `intValue`. Во-вторых, внутри совета производится проверка и логика запускается, только если значение аргумента не равно 100.

Для передачи аргумента совету понадобится также немного подкорректировать XML-конфигурацию. В этом случае нужно модифицировать выражение среза. Измененное выражение среза показано ниже:

```
<aop:pointcut id="fooExecution" expression="execution(* foo*(int))
    and args(intValue) and bean(myDependency*)"/>
```

К выражению среза были добавлены две дополнительных директивы. Первая из них, `args(intValue)`, сообщает Spring о необходимости передачи аргумента по имени `intValue` в совет "перед". Вторая директива, `bean(myDependency*)`, инструктирует Spring относительно снабжения советом только бинов, имеющих префикс идентификатора `myDependency`. Это весьма мощное средство; при наличии четко определенной структуры имен бинов Spring, вы можете очень легко снабжать советом только требуемые объекты. Например, используя `bean(*DAO*)`, совет можно применять ко всем бинам DAO, а с помощью `bean(*Service*)` применять его ко всем бинам, относящимся к уровням всех служб, не указывая полностью определенные имена классов для сопоставления.

Запуск той же самой тестовой программы (`AopNamespaceExample`) даст следующий вывод:

```
foo(int): 100
Executing: com.apress.prospring4.ch5.MyDependency foo argument: 101
foo(int): 101
bar()
```

В выводе видно, что советом был снабжен только вызов метода `foo()` с аргументом, значение которого не равно 100.

Давайте рассмотрим еще один пример применения пространства имен `aop` для совета "вокруг". Взамен создания другого класса для реализации интерфейса `MethodInterceptor` мы можем просто добавить новый метод к классу `MyAdvice`. Переделанный класс `MyAdvice` с новым методом `simpleAroundAdvice()` показан в листинге 5.67.

**Листинг 5.67. Класс `MyAdvice` (переделанный для проверки аргумента) с методом `simpleAroundAdvice()`**

```
package com.apress.prospring4.ch5;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;

public class MyAdvice {
    public void simpleBeforeAdvice(JoinPoint joinPoint, int intValue) {
        if (intValue != 100) {
            // Выполнять, только если intValue не равно 100
            System.out.println("Executing: " +
                joinPoint.getSignature().getDeclaringTypeName() + " "
                + joinPoint.getSignature().getName()
                + " argument: " + intValue);
        }
    }
```

```
    public Object simpleAroundAdvice(ProceedingJoinPoint pjp, int intValue)
            throws Throwable {
        // Вывести информацию перед выполнением
        System.out.println("Before execution: " +
            pjp.getSignature().getDeclaringTypeName() + " "
            + pjp.getSignature().getName()
            + " argument: " + intValue);

        Object retVal = pjp.proceed();

        // Вывести информацию после выполнения
        System.out.println("After execution: " +
            pjp.getSignature().getDeclaringTypeName() + " "
            + pjp.getSignature().getName() + " argument: " + intValue);

        return retVal;
    }
}
```

Вновь добавленный метод `simpleAroundAdvice()` должен получить, по крайней мере, один аргумент типа `ProceedingJoinPoint`, чтобы он смог продолжить обращением к целевому объекту. Кроме того, к нему также добавлен аргумент `intValue` для отображения его значения внутри совета.

В листинге 5.68 представлена XML-конфигурация.

**Листинг 5.68. Конфигурирование АОП в Spring с использованием пространства имен aop (совет "вокруг")**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <aop:config>
        <aop:pointcut id="fooExecution"
            expression="execution(* com.apress.prospring4.ch5..foo*(int))"/>

        <aop:aspect ref="advice">
            <aop:before pointcut-ref="fooExecution"
                method="simpleBeforeAdvice"/>
            <aop:around pointcut-ref="fooExecution"
                method="simpleAroundAdvice"/>
        </aop:aspect>
    </aop:config>

    <bean id="advice" class="com.apress.prospring4.ch5.MyAdvice"/>
    <bean id="myDependency"
        class="com.apress.prospring4.ch5.MyDependency"/>
    <bean id="myBean" class="com.apress.prospring4.ch5.MyBean">
        <property name="dep" ref="myDependency"/>
    </bean>
</beans
```

Мы только добавили новый дескриптор <aop:around> для объявления совета "вокруг" и ссылки на тот же самый срез. Запуск тестовой программы на этот раз дает следующий вывод:

```
Before execution: com.apress.prospring4.ch5.MyDependency foo argument: 100
foo(int): 100
After execution: com.apress.prospring4.ch5.MyDependency foo argument: 100
Executing: com.apress.prospring4.ch5.MyDependency foo argument: 101
Before execution: com.apress.prospring4.ch5.MyDependency foo argument: 101
foo(int): 101
After execution: com.apress.prospring4.ch5.MyDependency foo argument: 101
bar()
```

В выводе необходимо отметить два интересных момента. Во-первых, совет "вокруг" был применен к обоим вызовам метода foo(), поскольку он не проверяет аргумент. Во-вторых, для вызова метода foo() с передачей значения 101 в качестве аргумента были выполнены оба совета, "перед" и "вокруг", причем по умолчанию совет "перед" получает преимущество.

---

**На заметку!** При использовании пространства имен aop или аннотаций в стиле @AspectJ доступны два типа совета "после". *Совет "после возврата"* (дескриптор <aop:after-returning>) применяется, только если целевой метод завершился нормально. Другой тип, обычный *совет "после"* (дескриптор <aop:after>), выполняется как в случае нормального завершения метода, так и в ситуации, когда в методе возникла ошибка, из-за которой было сгенерировано исключение. Если необходимо, чтобы совет применялся независимо от результата выполнения целевого метода, следует использовать обычный совет "после".

---

## Использование аннотаций в стиле @AspectJ

В случае применения АОП в Spring с JDK 5 или последующей версии для объявления совета можно также использовать аннотации в стиле @AspectJ. Однако, как было указано ранее, для снабжения советами целевых методов Spring по-прежнему применяет собственный механизм создания прокси, а не механизм связывания AspectJ.

В этом разделе мы посмотрим, как реализовать такие же аспекты, как построенные с помощью пространства имен aop, но с использованием аннотаций в стиле @AspectJ. В примерах данного раздела аннотации будут применяться также и для других бинов Spring.

В листингах 5.69 и 5.70 показаны классы MyDependency и MyBean с аннотациями DI из Spring.

**Листинг 5.69. Класс MyDependency**

```
package com.apress.prospring4.ch5;
import org.springframework.stereotype.Component;
@Component("myDependency")
public class MyDependency {
    public void foo(int intValue) {
        System.out.println("foo(int): " + intValue);
    }
```

```
    public void bar() {
        System.out.println("bar()");
    }
}
```

**Листинг 5.70. Класс MyBean**

```
package com.apress.prospring4.ch5;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component("myBean")
public class MyBean {
    private MyDependency myDependency;

    public void execute() {
        myDependency.foo(100);
        myDependency.foo(101);
        myDependency.bar();
    }

    @Autowired
    public void setDep(MyDependency myDependency) {
        this.myDependency = myDependency;
    }
}
```

Оба класса аннотированы с помощью аннотации `@Component` с назначением соответствующих имен. В классе `MyBean` метод установки свойства `myDependency` аннотирован посредством `@Autowired` для автоматического внедрения платформой Spring.

А теперь посмотрим, как класс `MyAdvice` может использовать аннотации в стиле `@AspectJ`. Мы реализуем срезы, а также советы "перед" и "вокруг". Код класса `MyAdvice` приведен в листинге 5.71.

**Листинг 5.71. Класс MyAdvice**

```
package com.apress.prospring4.ch5;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class MyAdvice {
    @Pointcut("execution(* com.apress.prospring4.ch5..foo*(int)) &&
              args(intValue)")
    public void fooExecution(int intValue) {
    }
```

```
    @Pointcut("bean(myDependency*)")
    public void inMyDependency() {
    }
    @Before("fooExecution(intValue) && inMyDependency()")
    public void simpleBeforeAdvice(JoinPoint joinPoint, int intValue) {
        if (intValue != 100) {
        // Выполнять, только если intValue не равно 100
        System.out.println("Executing: " +
            joinPoint.getSignature().getDeclaringTypeName() + " "
            + joinPoint.getSignature().getName() + " argument: " + intValue);
        }
    }
    @Around("fooExecution(intValue) && inMyDependency()")
    public Object simpleAroundAdvice(ProceedingJoinPoint pjp, int intValue)
            throws Throwable {
        // Вывести информацию перед выполнением
        System.out.println("Before execution: " +
            pjp.getSignature().getDeclaringTypeName() + " "
            + pjp.getSignature().getName()
            + " argument: " + intValue);
        Object retVal = pjp.proceed();
        // Вывести информацию после выполнения
        System.out.println("After execution: " +
            pjp.getSignature().getDeclaringTypeName() + " "
            + pjp.getSignature().getName()
            + " argument: " + intValue);
        return retVal;
    }
}
```

Вы заметите, что структура кода очень похожа на ту, что использовалась в примере с пространством имен `aop`, только в этом случае применяются аннотации. Тем не менее, все равно важно отметить несколько моментов.

- **Для аннотирования класса `MyAdvice` использовались аннотации `@Component` и `@Aspect`.** Аннотация `@Aspect` объявляет его как класс аспекта. Чтобы позволить Spring сканировать компонент, когда в XML-конфигурации применяется дескриптор `<context:component-scan>`, класс также необходимо аннотировать с помощью `@Component`.
- **Срезы определены как методы, возвращающие `void`.** В классе было определено два среза; оба они аннотированы посредством `@Pointcut`. Мы намеренно разделили выражение среза в примере с пространством имен `aop` на два. Первое выражение (указанное методом `fooExecution(int intValue)`) определяет срез для выполнения методов `foo*()` с целочисленным аргументом внутри всех классов из пакета `com.apress.prospring4.ch5`, и этот аргумент (`intValue`) будет также передан в совет. Второе выражение (указанное методом `inMyDependency()`) определяет другой срез, предназначенный для выполнения всех методов из бинов Spring, имена которых имеют префикс `myDependency`. Также обратите внимание, что для представления условия "и" в выражении среза применяется `&&`, тогда как в случае пространства имен `aop` для этого использовалась операция `and`.

- **Метод совета "перед" аннотирован с помощью `@Before`, а метод совета "вокруг" — посредством `@Around`.** Для обоих советов передается значение, которое участвует в двух срезах, определенных в классе. Конструкция `"fooExecution(intValue) && inMyDependency()"` означает, что для применения совета должны быть удовлетворены условия обоих срезов, и это то же самое, что и операция пересечения в `ComposablePointcut`.
- **Логика советов "перед" и "вокруг" совпадает с логикой этих же советов в примере с пространством имен `aop`.**

При наличии всех этих аннотаций XML-конфигурация становится очень простой. В листинге 5.72 показана конфигурация для рассматриваемого примера (`app-config-annotation.xml`).

**Листинг 5.72. Конфигурирование АОП в Spring посредством аннотаций `@AspectJ`**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop.xsd
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context.xsd">

   <aop:aspectj-autoproxy/>

   <context:component-scan base-package="com.apress.prospring4.ch5"/>
</beans>
```

Здесь объявлено только два дескриптора. Дескриптор `<aop:aspect-autorpoxy>` предназначен для сообщения Spring о том, что нужно сканировать аннотации в стиле `@AspectJ`, а дескриптор `<context:component-scan>` по-прежнему требуется Spring для сканирования бинов Spring внутри пакета, в котором находится совет. Также необходимо аннотировать класс совета с помощью аннотации `@Component`, указывая, что он является компонентом Spring.

Дескриптор `<aop:aspectj-autoproxy>` имеет атрибут `proxy-target-class`. По умолчанию он установлен в `false`, а это означает, что Spring будет создавать стандартные прокси, основанные на интерфейсах, с использованием динамического прокси JDK. В случае установки `proxy-target-class` в `true` платформа Spring будет применять библиотеку CGLIB для создания прокси, которые основаны на классах. В листинге 5.73 показана тестовая программа в форме класса `AspectJAnnotationExample`.

**Листинг 5.73. Тестирование аннотаций AspectJ**

```
package com.apress.prospring4.ch5;

import org.springframework.context.support.GenericXmlApplicationContext;
```

```
public class AspectJAnnotationExample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        MyBean myBean = (MyBean) ctx.getBean("myBean");
        myBean.execute();
    }
}
```

Запуск этой программы на выполнение дает те же самые результаты, что и пример с пространством имен `aop`:

```
Before execution: com.apress.prospring4.ch5.MyDependency foo argument: 100
foo(int): 100
After execution: com.apress.prospring4.ch5.MyDependency foo argument: 100
Before execution: com.apress.prospring4.ch5.MyDependency foo argument: 101
Executing: com.apress.prospring4.ch5.MyDependency foo argument: 101
foo(int): 101
After execution: com.apress.prospring4.ch5.MyDependency foo argument: 101
bar()
```

## Соображения по поводу декларативного конфигурирования АОП в Spring

К этому моменту мы обсудили три способа определения конфигурации АОП в Spring, включая класс `ProxyFactoryBean`, пространство имен `aop` и аннотации в стиле `@AspectJ`. Мы уверены, вы согласитесь с тем, что применять пространство имен `aop` намного проще, чем класс `ProxyFactoryBean`. Итак, остается главный вопрос: что использовать — пространство имен `aop` или аннотации в стиле `@AspectJ`?

Если приложение Spring основано на XML-конфигурации, то применение пространства имен `aop` является естественным выбором, поскольку оно обеспечит согласованность стилей конфигураций АОП и DI. С другой стороны, если приложение в основном базируется на аннотациях, то следует использовать аннотации `@AspectJ`. Как обычно, позвольте требованиям к приложению и конфигурации управлять вашим выбором подхода, который наилучшим образом соответствует разрабатываемому проекту.

Кроме того, существуют и другие отличия между подходами с пространством имен `aop` и аннотациями `@AspectJ`.

- Синтаксис для выражений срезов имеет небольшие отличия (например, как упоминалось ранее, необходимо применять `and` при подходе с пространством имен `aop`, но `&&` при подходе с аннотациями `@AspectJ`).
- Подход с пространством имен `aop` поддерживает только модель создания одиночных экземпляров аспектов.
- При подходе с пространством имен `aop` нельзя "комбинировать" несколько выражений срезов. Например, в рассмотренном ранее примере использования аннотаций `@AspectJ` мы могли комбинировать два определения срезов

(т.е. `fooExecution(intValue) && inMyDependency()`) в советах "перед" и "вокруг". Однако поступить так в случае применения пространства имен `aop` не получится, поэтому придется создать новое выражение среза, комбинирующее условия сопоставления.

# Интеграция с AspectJ

АОП предлагает мощное решение для многих общих задач, которые возникают при объектно-ориентированной разработке приложений. При использовании АОП в Spring в вашем распоряжении имеется подмножество функциональности АОП, с помощью которого в большинстве случаев можно решить задачи, появляющиеся во время построения приложений. Однако в ряде ситуаций могут понадобиться средства АОП, которые выходят за рамки доступных в реализации АОП из Spring.

С позиции точки соединения АОП в Spring поддерживает только срезы, соответствующие выполнению открытых нестатических методов. Тем не менее, в некоторых случаях совет необходимо применять к защищенным/закрытым методам, во время создания объекта или доступа к полям и т.д.

В таких ситуациях следует обратиться к реализации АОП с более полным набором функциональных возможностей. Мы предпочитаем использовать AspectJ, и поскольку теперь можно конфигурировать аспекты AspectJ с помощью Spring, AspectJ становится великолепным дополнением АОП в Spring.

## Что собой представляет AspectJ

AspectJ — это полнофункциональная реализация АОП, которая использует процесс связывания (либо во время компиляции, либо во время выполнения) для введения аспектов в код. В AspectJ аспекты и срезы построены с применением Java-подобного синтаксиса, который упрощает его изучение разработчиками на языке Java. Мы не собираемся уделять слишком много времени на исследование работы AspectJ, потому что это выходит за рамки контекста настоящей книги. Взамен мы представим ряд простых примеров AspectJ и покажем, как их сконфигурировать в Spring. За дополнительными сведениями по AspectJ обращайтесь к книге *AspectJ in Action: Enterprise AOP with Spring Applications,* 2-е издание (Manning, 2009 г.).

---

**На заметку!** Мы не будет описывать, каким образом связывать аспекты AspectJ с приложением. За соответствующими деталями обращайтесь к документации по AspectJ или просмотрите пример кода `aspectj-aspects` для главы 5.

---

## Использование одиночных экземпляров аспектов

По умолчанию аспекты AspectJ являются одиночными экземплярами в том смысле, что вы получаете единственный экземпляр на один загрузчик класса. Проблема, которая возникает в платформе Spring с каждым аспектом AspectJ, состоит в том, что она не может создать экземпляр аспекта, поскольку это обрабатывается самой инфраструктурой AspectJ. Однако в каждом аспекте доступен метод `org.aspectj.lang.Aspects.aspectOf()`, который можно использовать для доступа к экземпляру аспекта. Применяя метод `aspectOf()` и специальную возможность конфигурирования Spring, вы можете поручить конфигурирование аспекта платформе Spring.

Благодаря этой поддержке, вы получаете полную отдачу от мощного набора функциональных средств АОП, предлагаемого AspectJ, не теряя великолепных возможностей DI и конфигурирования Spring. Это также означает, что в приложении вам не понадобятся два отдельных метода конфигурирования; один и тот же подход с `ApplicationContext` можно использовать для всех бинов, управляемых Spring, и для аспектов AspectJ.

В листинге 5.74 приведен код базового класса `MessageWriter`, который будет снабжен советом с применением AspectJ.

**Листинг 5.74. Класс `MessageWriter`**

```
package com.apress.prospring4.ch5;

public class MessageWriter {
    public void writeMessage() {
        System.out.println("foobar!");
    }
    public void foo() {
        System.out.println("foo");
    }
}
```

В этом примере мы собираемся использовать AspectJ для снабжения советом метода `writeMessage()` и вывода сообщения до и после вызова метода. Выводимые сообщения можно будет конфигурировать с помощью Spring.

В листинге 5.75 показан код аспекта `MessageWrapper` (именем файла является `MessageWrapper.aj`, т.е. это файл AspectJ, а не стандартный Java-класс).

**Листинг 5.75. Аспект `MessageWrapper`**

```
package com.apress.prospring4.ch5;

public aspect MessageWrapper {
    private String prefix;
    private String suffix;
    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }
    public String getPrefix() {
        return this.prefix;
    }
    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }
    public String getSuffix() {
        return this.suffix;
    }
    pointcut doWriting() :
        execution(*
          com.apress.prospring4.ch5.MessageWriter.writeMessage());
    before() : doWriting() {
        System.out.println(prefix);
    }
```

```
    after() : doWriting()
        System.out.println(suffix);
    }
}
```

По существу мы создаем аспект по имени `MessageWrapper` и, как в случае нормального Java-класса, предоставляем ему два свойства, `suffix` и `prefix`, которые будут использоваться при снабжении советом метода `writeMessage()`. Затем мы определяем именованный срез `doWriting()` для единственной точки соединения, в этом случае — для выполнения метода `writeMessage()`. Инфраструктура AspectJ поддерживает большое количество точек соединения, но перечисление их всех не входит в круг задач данного примера. Наконец, мы определяем советы двух видов: один выполняется перед срезом `doWriting()`, а другой — после него. В листинге 5.76 показано, как этот аспект конфигурируется в Spring (`app-config-xml.xml`).

**Листинг 5.76. Конфигурирование аспекта AspectJ**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="aspect" class="com.apress.prospring4.ch5.MessageWrapper"
        factory-method="aspectOf">
        <property name="prefix">
            <value>The Prefix</value>
        </property>
        <property name="suffix">
            <value>The Suffix</value>
        </property>
    </bean>
</beans>
```

Как видите, большая часть конфигурации бина аспекта очень похожа на конфигурацию стандартного бина. Единственное отличие заключается в использовании атрибута `factory-method` дескриптора `<bean>`. Этот атрибут позволяет классам, следующим традиционному шаблону "Фабрика" (Factory), плавно интегрироваться в Spring. Например, если есть класс `Foo` с закрытым конструктором и статическим фабричным методом `getInstance()`, то применение атрибута `factory-method` позволяет бину этого класса быть управляемым со стороны Spring. Метод `aspectOf()`, имеющийся в каждом аспекте AspectJ, предоставляет доступ к экземпляру аспекта, позволяя Spring устанавливать свойства этого аспекта. В листинге 5.77 приведен код простого тестового приложения для рассматриваемого примера.

**Листинг 5.77. Конфигурация AspectJ в действии**

```
package com.apress.prospring4.ch5;

import org.springframework.context.support.GenericXmlApplicationContext;

public class AspectJExample {
```

```
    public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load(new String[] {"classpath:META-INF/spring/app-context-xml.xml"});
      ctx.refresh();
      MessageWriter writer = new MessageWriter();
      writer.writeMessage();
      writer.foo();
    }
}
```

Обратите внимание, что сначала мы загружаем `ApplicationContext`, чтобы дать возможность Spring сконфигурировать аспект. Затем мы создаем экземпляр `MessageWriter` и вызываем методы `writeMessage()` и `foo()`. Запуск этого примера на выполнение дает следующий вывод:

```
The Prefix
foobar!
The Suffix
foo
```

Здесь видно, что совет в аспекте `MessageWrapper` был применен к методу `writeMessage()`, а значения свойств `prefix` и `suffix`, указанные в конфигурации `ApplicationContext`, использовались советом при выводе сообщений до и после вызова метода.

# Резюме

В этой главе мы раскрыли множество ключевых концепций АОП и показали, как эти концепции были воплощены в реализации АОП в Spring. Мы обсудили функциональные средства, которые реализованы (и не реализованы) в рамках АОП в Spring, и указали на AspectJ как на решение АОП с возможностями, отсутствующими в реализации Spring. Некоторое время было уделено объяснению деталей типов советов, доступных в Spring, и рассмотрению примеров четырех типов советов в действии. Мы также показали, как с помощью срезов ограничить методы, к которым применяется совет. В частности, были продемонстрированы шесть базовых реализаций срезов, предлагаемых Spring. Вдобавок мы объяснили детали конструирования прокси АОП, различные их варианты, а также отличия между ними. Мы сравнили производительность трех типов прокси и подчеркнули основные отличия и ограничения при выборе между прокси JDK и CGLIB. Мы взглянули на дополнительные возможности для создания срезов и показали, как расширять набор интерфейсов, реализуемых объектом, с использованием введений. Мы исследовали службы Spring Framework для декларативного конфигурирования АОП, которые позволяют избежать жесткого кодирования логики конструирования прокси АОП. Также мы кратко рассмотрели интеграцию Spring и AspectJ, которая дает возможность получить всю мощь AspectJ, не теряя преимуществ гибкости Spring. Это был определенно огромный материал, посвященный АОП.

В следующей главе мы перейдем к совершенно другой теме — использование поддержки JDBC, предлагаемой Spring, для радикального упрощения процесса кодирования доступа к данным на основе JDBC.

ГЛАВА 6

# Поддержка JDBC в Spring

К этому моменту вы уже видели, насколько просто строить приложение, полностью управляемое Spring. Вы получили хорошее представление о конфигурации бинов и об аспектно-ориентированном программировании (АОП). Тем не менее, отсутствует еще одна часть головоломки: как получить данные, которые управляют приложением?

Если отбросить простейшие одноразовые утилиты командной строки, то почти каждое приложение нуждается в сохранении данных в хранилище какого-то вида. Наиболее часто используемым и удобным хранилищем является реляционная база данных.

Самыми примечательными системами управления реляционными базами данных (СУРБД) с открытым кодом можно считать MySQL (`www.mysql.com`) и PostgreSQL (`www.postgresql.org`). Как правило, MySQL более широко используется при разработке веб-приложений, особенно на платформе Linux. С другой стороны, система PostgreSQL является более дружественной для разработчиков под Oracle, поскольку ее процедурный язык PL/pgSQL очень близок к языку PL/SQL в Oracle.

Даже в случае выбора самой быстрой и надежной СУРБД приложение может потерять в скорости и гибкости, если в нем присутствует плохо спроектированный и реализованный уровень доступа к данным. Приложения, как правило, работают с уровнем доступа к данным весьма интенсивно, в результате чего любые узкие места в коде доступа к данным влияют на все приложение независимо от того, насколько хорошо спроектированы остальные уровни.

В этой главе мы покажем, как с помощью Spring упростить реализацию кода доступа к данным, используя JDBC. Мы начнем с рассмотрения огромного объема повторяющегося кода, который пришлось бы писать, не имея Spring, и затем сравним его с классом доступа к данным, реализованным с применением Spring. Результат действительно впечатляет: платформа Spring позволяет использовать всю мощь настроенных человеком SQL-запросов, сводя к минимуму объем кода поддержки, который придется написать. В частности, в главе будут рассматриваться следующие темы.

- **Сравнение традиционного кода JDBC и поддержки JDBC в Spring.** Мы исследуем, каким образом Spring упрощает код JDBC старого стиля, обеспечивая в то же самое время аналогичную функциональность. Вы также увидите, каким

образом Spring обращается к низкоуровневому API-интерфейсу JDBC и как этот API-интерфейс отображается на удобные классы вроде `JdbcTemplate`.

- **Подключение к базе данных.** Хотя мы не собираемся вникать в каждую мелкую деталь управления подключениями к базам данных, мы продемонстрируем фундаментальные отличия между `Connection` и `DataSource`. Естественно, мы обсудим, каким образом Spring управляет источниками данных (`DataSource`), и какие источники данных можно использовать в приложениях.
- **Извлечение и отображение данных на Java-объекты.** Мы покажем, как извлекать данные и затем эффективно отображать выбранные данные на Java-объекты. Вы также узнаете, что Spring JDBC является жизнеспособной альтернативой инструментам объектно-реляционного отображения (object-relational mapping — ORM).
- **Вставка, обновление и удаление данных.** Наконец, мы обсудим, каким образом реализовать операции вставки, обновления и удаления за счет применения Spring для выполнения запросов указанных типов.

# Введение в лямбда-выражения

В выпуске Java 8 наряду с множеством других функциональных средств появилась поддержка лямбда-выражений. Лямбда-выражения представляют собой великолепную замену использованию внутренних анонимных классов, а также идеальный кандидат для работы с поддержкой JDBC в Spring. Для применения лямбда-выражений наличие версии Java 8 *обязательно*. Эта книга была написана во времена предварительных выпусков Java 8 и выхода первой версии General Availability, поэтому мы отдаем себе отчет в том, что пока далеко не каждый разработчик имеет дело с Java 8. С учетом сказанного, в примерах настоящей главы приводятся оба варианта там, где это применимо. Лямбда-выражения подходят в большинстве мест Spring API, где используются шаблоны или обратные вызовы, так что они совершенно не ограничены JDBC. В этой главе сами лямбда-выражения не раскрываются, поскольку они являются средством языка Java, и вы должны быть знакомы с концепциями и синтаксисом лямбда-выражений. За дополнительной информацией по этой теме обращайтесь к руководству по ссылке `http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`.

# Модель данных для кода примеров

Перед тем, как продолжить, мы должны представить простую модель данных, которая используется во всех примерах этой главы, а также в нескольких последующих главах при обсуждении других технологий доступа к данным (мы будем соответствующим образом расширять модель для удовлетворения потребностей, связанных с каждой новой темой).

Модель включает простую базу данных контактов с двумя таблицами. Первая из них, таблица `CONTACT`, хранит информацию о контактной персоне, а вторая, таблица `CONTACT_TEL_DETAIL`, содержит подробности о телефонах этой персоны. Каждый контакт может иметь ноль или более телефонных номеров; другими словами, между таблицами `CONTACT` и `CONTACT_TEL_DETAIL` существует отношение "один ко мно-

гим". Информация о контакте включает имя и фамилию, дату рождения, тип телефона и соответствующий телефонный номер. На рис. 6.1 показана диаграмма "сущность-отношение" (entity-relationship — ER) для этой базы данных.

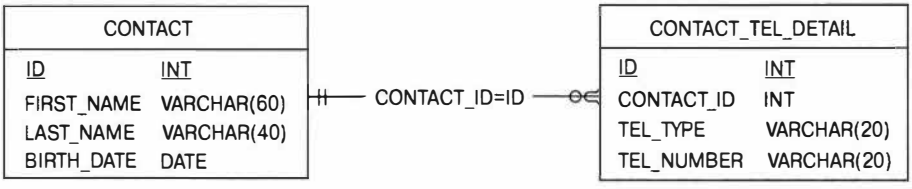


**Рис. 6.1.** Простая модель данных для кода примеров

Как видите, в обеих таблицах присутствует столбец `ID`, значение которого автоматически устанавливается базой данных во время вставки.

Таблица `CONTACT_TEL_DETAIL` имеет отношение внешнего ключа с таблицей `CONTACT`, которое поддерживает связь по столбцу `CONTACT_ID` с первичным ключом таблицы `CONTACT` (т.е. столбцом `ID`).

---

**На заметку!** В ряде примеров настоящей главы для демонстрации взаимодействия с реальной базой данных мы используем СУРБД с открытым кодом MySQL. Это требует наличия у вас доступного экземпляра MySQL. Мы не рассматриваем процесс установки MySQL. При желании вы можете применять другую СУРБД, но тогда должны соответствующим образом модифицировать определения схемы и функций. Кроме того, мы покажем, как использовать встроенную базу данных, что устраняет необходимость в базе данных MySQL.

---

В листинге 6.1 приведен сценарий для создания базы данных (совместимый с MySQL).

**Листинг 6.1. Простой сценарий для создания модели данных (`schema.sql`)**

```
CREATE TABLE CONTACT (
      ID INT NOT NULL AUTO_INCREMENT
    , FIRST_NAME VARCHAR(60) NOT NULL
    , LAST_NAME VARCHAR(40) NOT NULL
    , BIRTH_DATE DATE
    , UNIQUE UQ_CONTACT_1 (FIRST_NAME, LAST_NAME)
    , PRIMARY KEY (ID)
);

CREATE TABLE CONTACT_TEL_DETAIL (
      ID INT NOT NULL AUTO_INCREMENT
    , CONTACT_ID INT NOT NULL
    , TEL_TYPE VARCHAR(20) NOT NULL
    , TEL_NUMBER VARCHAR(20) NOT NULL
    , UNIQUE UQ_CONTACT_TEL_DETAIL_1 (CONTACT_ID, TEL_TYPE)
    , PRIMARY KEY (ID)
    , CONSTRAINT FK_CONTACT_TEL_DETAIL_1 FOREIGN KEY (CONTACT_ID)
                REFERENCES CONTACT (ID)
);
```

В листинге 6.2 показан сценарий для наполнения таблиц CONTACT и CONTACT_TEL_DETAIL тестовыми данными.

**Листинг 6.2. Простой сценарий для наполнения данными (test-data.sql)**

```
insert into contact (first_name, last_name, birth_date) values
  ('Chris', 'Schaefer', '1981-05-03');
insert into contact (first_name, last_name, birth_date) values
  ('Scott', 'Tiger', '1990-11-02');
insert into contact (first_name, last_name, birth_date) values
  ('John', 'Smith', '1964-02-28');
insert into contact_tel_detail (contact_id, tel_type, tel_number) values
  (1, 'Mobile', '1234567890');
insert into contact_tel_detail (contact_id, tel_type, tel_number) values
  (1, 'Home', '1234567890');
insert into contact_tel_detail (contact_id, tel_type, tel_number) values
  (2, 'Home', '1234567890');
```

В последующих разделах этой главы будут приведены примеры извлечения данных из базы данных через JDBC и отображения результирующего набора непосредственно на Java-объекты (т.е. объекты POJO). В листингах 6.3 и 6.4 представлены классы предметной области Contact и ContactTelDetail.

**Листинг 6.3. Класс предметной области Contact**

```
package com.apress.prospring4.ch6;

import java.io.Serializable;
import java.sql.Date;
import java.util.List;

public class Contact implements Serializable {
    private Long id;
    private String firstName;
    private String lastName;
    private Date birthDate;
    private List<ContactTelDetail> contactTelDetails;

    public void setId(Long id) {
        this.id = id;
    }

    public Long getId() {
        return this.id;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getFirstName() {
        return this.firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
```

```
    public String getLastName() {
        return this.lastName;
    }
    public void setContactTelDetails(List<ContactTelDetail> contactTelDetails)
    {
        this.contactTelDetails = contactTelDetails;
    }
    public List<ContactTelDetail> getContactTelDetails() {
        return contactTelDetails;
    }
    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }
    public Date getBirthDate() {
        return birthDate;
    }
    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate;
    }
}
```

**Листинг 6.4. Класс предметной области `ContactTelDetail`**

```
package com.apress.prospring4.ch6;
import java.io.Serializable; }
public class ContactTelDetail implements Serializable {
    private Long id;
    private Long contactId;
    private String telType;
    private String telNumber;
    public void setId(Long id) {
        this.id = id;
    }
    public Long getId() {
        return this.id;
    }
    public void setContactId(Long contactId) {
        this.contactId = contactId;
    }
    public Long getContactId() {
        return this.contactId;
    }
    public void setTelType(String telType) {
        this.telType = telType;
    }
    public String getTelType() {
        return this.telType;
    }
```

```
    public void setTelNumber(String telNumber) {
        this.telNumber = telNumber;
    }
    public String getTelNumber() {}
        return this.telNumber;
    }
    @Override
    public String toString() {
        return "Contact Tel Detail - Id: " + id + ", Contact id: " + contactId
            + ", Type: " + telType + ", Number: " + telNumber;
    }
}
```

Давайте начнем с простого интерфейса `ContactDao`, который инкапсулирует все службы доступа к данным для информации о контакте. Этот интерфейс показан в листинге 6.5.

**Листинг 6.5. Интерфейс `ContactDao`**

```
package com.apress.prospring4.ch6;

import java.util.List;

public interface ContactDao {
    List<Contact> findAll();
    List<Contact> findByFirstName(String firstName);
    String findLastNameById(Long id);
    String findFirstNameById(Long id);
    void insert(Contact contact);
    void update(Contact contact);
    void delete(Long contactId);
}
```

В интерфейсе `ContactDao` определено несколько методов поиска, а также методы вставки, обновления и удаления, которые вместе объединяются в термин CRUD (create, read, update, delete — создание, чтение, обновление и удаление).

Наконец, чтобы упростить тестирование, мы модифицируем свойства `log4j`, указав `DEBUG` в качестве уровня регистрации в журнале для всех классов. При уровне `DEBUG` модуль JDBC в Spring будет выводить все лежащие в основе SQL-операторы, выполняемые в базе данных, поэтому вы будете знать, что точно происходит; это особенно удобно во время поиска и устранения синтаксических ошибок в SQL-операторах. В листинге 6.6 приведено содержимое файла `log4j.properties` (находящегося в папке `src/main/resources` файлов исходного кода проекта для главы 8) с включенным уровнем `DEBUG`.

**Листинг 6.6. Файл `log4j.properties`**

```
log4j.rootCategory=DEBUG, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %40.40c:%4L - %m%n
```

# Исследование инфраструктуры JDBC

Инфраструктура JDBC предоставляет Java-приложениям стандартный способ доступа к данным, хранящимся в базе данных. В основе JDBC лежит драйвер, специфичный для каждой базы данных; именно этот драйвер позволяет Java-коду получать доступ к базе данных.

После загрузки драйвер регистрирует себя с помощью класса `java.sql.DriverManager`. Этот класс управляет списком драйверов и предоставляет статические методы для установления подключений к базе данных. Метод `getConnection()` класса `DriverManager` возвращает интерфейс `java.sql.Connection`, реализованный драйвером. Этот интерфейс позволяет запускать SQL-операторы в базе данных.

Инфраструктура JDBC сложна и хорошо протестирована; тем не менее, присущая ей сложность затрудняет разработку. Первый уровень сложности связан с обеспечением в коде средств управления подключениями к базе данных. Подключение является дефицитным ресурсом, который к тому же сопровождается довольно высокими расходами при установлении. Обычно база данных создает поток или порождает дочерний процесс для каждого подключения. Однако количество параллельных подключений, как правило, ограничено, и большое число открытых подключений замедлит работу базы данных.

Мы покажем, как Spring помогает справиться с этой сложностью, но сначала нужно посмотреть, каким образом выбираются, удаляются и обновляются данные в чистом JDBC.

Давайте создадим простую форму реализации интерфейса `ContactDao` для взаимодействия с базой данных посредством чистого JDBC. Памятуя о том, что уже известно о подключениях к базе данных, мы возьмем всю ответственность на себя и применим дорогостоящий (в смысле производительности) подход с созданием подключения для каждого оператора. Это приведет к значительному снижению производительности Java и добавит дополнительную нагрузку на базу данных, поскольку подключение должно устанавливаться для каждого запроса. Однако если оставлять подключение открытым, то сервер базы данных может даже прекратить работу. В листинге 6.7 приведен код, необходимый для управления подключением JDBC, в котором в качестве примера используется MySQL.

**Листинг 6.7. Управление подключением JDBC**

```
package com.apress.prospring4.ch6;
public class PlainContactDao implements ContactDao {
   static {
      try {
         Class.forName("com.mysql.jdbc.Driver");
      } catch (ClassNotFoundException ex) {
         ex.printStackTrace();
      }
   }
   private Connection getConnection() throws SQLException {
      return DriverManager.getConnection(
         "jdbc:mysql://localhost:3306/prospring4_ch6",
         "prospring4", "prospring4");
   }
   ...
```

```
    private void closeConnection(Connection connection) {
        if (connection == null) {
            return;
        }
        try {
            connection.close();
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
    }
}
```

Хотя этот код далек от завершения, он дает представление о том, какие действия должны предприниматься для управления подключением JDBC. Код даже не работает с пулом подключений, который является распространенным способом более эффективного управления подключениями к базе данных. Мы пока не обсуждаем организацию пула подключений (это будет сделано в разделе "Подключения к базе данных и источники данных" далее в главе), а взамен приводим в листинге 6.8 реализацию методов `findAll()`, `insert()` и `delete()` интерфейса `ContactDao` с использованием простого JDBC.

**Листинг 6.8. Реализация DAO с использованием простого JDBC**

```
package com.apress.prospring4.ch6;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

public class PlainContactDao implements ContactDao {
    static {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException ex) {
            ex.printStackTrace();
        }
    }

    private Connection getConnection() throws SQLException {
        return DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/prospring4_ch6",
            "prospring4", "prospring4");
    }

    private void closeConnection(Connection connection) {
        if (connection == null) {
            return;
        }
```

```
    try {
        connection.close();
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
}
@Override
public List<Contact> findAll() {
    List<Contact> result = new ArrayList<Contact>();
    Connection connection = null;

    try {
        connection = getConnection();
        PreparedStatement statement =
            connection.prepareStatement("select * from contact");
        ResultSet resultSet = statement.executeQuery();

        while (resultSet.next()) {
            Contact contact = new Contact();
            contact.setId(resultSet.getLong("id"));
            contact.setFirstName(resultSet.getString("first_name"));
            contact.setLastName(resultSet.getString("last_name"));
            contact.setBirthDate(resultSet.getDate("birth_date"));

            result.add(contact);
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
    } finally {
        closeConnection(connection);
    }
    return result;
}
@Override
public void insert(Contact contact) {
    Connection connection = null;
    try {
        connection = getConnection();
        PreparedStatement statement = connection.prepareStatement(
"insert into Contact (first_name, last_name, birth_date) values (?, ?, ?)"
          , Statement.RETURN_GENERATED_KEYS);
        statement.setString(1, contact.getFirstName());
        statement.setString(2, contact.getLastName());
        statement.setDate(3, contact.getBirthDate());
        statement.execute();
        ResultSet generatedKeys = statement.getGeneratedKeys();

        if (generatedKeys.next()) {
            contact.setId(generatedKeys.getLong(1));
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
    } finally {
        closeConnection(connection);
    }
}
```

```
    @Override
    public void delete(Long contactId) {
        Connection connection = null;
        try {
            connection = getConnection();
            PreparedStatement statement =
                connection.prepareStatement("delete from contact where id=?");
            statement.setLong(1, contactId);
            statement.execute();
        } catch (SQLException ex) {
            ex.printStackTrace();
        } finally {
            closeConnection(connection);
        }
    }
    @Override
    public List<Contact> findByFirstName(String firstName) {
        return null;
    }
    @Override
    public String findFirstNameById(Long id) {
        return null;
    }
    @Override
    public String findLastNameById(Long id) {
        return null;
    }
    @Override
    public void update(Contact contact) {
    }
}
```

В листинге 6.9 приведена главная тестовая программа, демонстрирующая показанную выше реализацию DAO в действии.

**Листинг 6.9. Тестирование реализации, использующей чистый JDBC**

```
package com.apress.prospring4.ch6;

import java.sql.Date;
import java.util.GregorianCalendar;
import java.util.List;

public class PlainJdbcSample {
    private static ContactDao contactDao = new PlainContactDao();

    public static void main(String[] args) {
        // Вывести начальный список контактов
        System.out.println("Listing initial contact data:");

        listAllContacts();

        System.out.println();
        // Вставить новый контакт
        System.out.println("Insert a new contact");
```

```
        Contact contact = new Contact();
        contact.setFirstName("Jacky");
        contact.setLastName("Chan");
        contact.setBirthDate(new Date(
            (new GregorianCalendar(2001, 10, 1)).getTime().getTime()));
        contactDao.insert(contact);

        // Вывести список контактов после создания нового контакта
        System.out.println("Listing contact data after new contact created:");
        listAllContacts();

        System.out.println();
        // Удалить только что созданный контакт
        System.out.println("Deleting the previous created contact");
        contactDao.delete(contact.getId());.

        // Вывести список контактов после удаления ранее созданного контакта
        System.out.println("Listing contact data after new contact deleted:");
        listAllContacts();
    }

    private static void listAllContacts() {
        List<Contact> contacts = contactDao.findAll();

        for (Contact contact: contacts) {
            System.out.println(contact);
        }
    }
}
```

Кроме того, в проект необходимо добавить зависимость для MySQL Java (табл. 6.1).

**Таблица 6.1. Зависимость для MySQL**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| mysql | mysql-connector-java | 5.1.29 | Java-библиотека драйвера MySQL |

Запуск программы из листинга 6.9 дает следующие результаты (предполагается, что вы имеете локально базу данных MySQL по имени prospring4_ch6, а также располагаете именем пользователя и паролем, которые установлены в prospring4; вы должны иметь возможность доступа к схеме базы данных и запустить в базе данных сценарии schema.sql и test-data.sql для создания таблиц и наполнения их начальными данными):

```
Listing initial contact data:
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Insert a new contact
Listing contact data after new contact created:
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
Contact - Id: 4, First name: Jacky, Last name: Chan, Birthday: 2001-11-01
```

```
Deleting the previous created contact
Listing contact data after new contact deleted:
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

В первом блоке строк внутри вывода отображаются начальные данные. Второй блок строк показывает, что была добавлена новая запись. Финальный блок строк отражает факт удаления ранее созданной записи.

Как видно в листинге 6.8, большой объем кода нуждается в перемещении во вспомогательный класс или, что еще хуже, дублируется в каждом классе DAO. Это главный недостаток JDBC с точки зрения разработчика приложений — ему просто не хватит времени повторять один и тот код в каждом классе DAO. Вместо этого лучше сосредоточиться на написании кода, который действительно делает то, что должен делать класс DAO: выборка, обновление и удаление данных. Чем больше вспомогательного кода нужно написать, тем больше проверяемых исключений придется проверить и тем больше ошибок можно внести в код.

Именно здесь вступает в игру инфраструктура DAO и Spring. Платформа исключает код, не выполняющий какую-либо специальную логику, и позволяет не думать о вспомогательных действиях, которые должны быть выполнены. Кроме того, широкая поддержка JDBC в Spring также значительно упрощает решение задач.

# Инфраструктура JDBC в Spring

Код, который обсуждался в первой части этой главы, не был особенно сложным, но мало того, что писать его утомительно, при его вводе высока вероятность допущения ошибок. Самое время взглянуть, каким образом Spring упрощает реализацию и делает ее более элегантной.

## Обзор пакетов JDBC в Spring

Поддержка JDBC в Spring разделена на пять пакетов, которые описаны в табл. 6.2; каждый пакет обрабатывает те или иные аспекты доступа JDBC.

**Таблица 6.2. Пакеты JDBC в Spring**

| Пакет | Описание |
|---|---|
| `org.springframework.jdbc.core` | Содержит ядро для классов JDBC в Spring. Пакет включает класс JDBC по имени `JdbcTemplate`, который упрощает программирование операций для базы данных с помощью JDBC. Множество его подпакетов предоставляют поддержку доступа к данным JDBC с более специализированным назначением (например, класс `JdbcTemplate`, который поддерживает именованные параметры), а также связанные служебные классы |
| `org.springframework.jdbc.datasource` | Содержит вспомогательные классы и реализации `DataSource`, которые можно использовать для запуска кода JDBC вне контейнера JEE. Множество подпакетов предоставляют поддержку для встроенных баз данных, инициализации баз данных и разнообразных механизмов поиска в источниках данных |

*Окончание табл. 6.2*

| Пакет | Описание |
|---|---|
| `org.springframework.jdbc.object` | Содержит классы, которые помогают преобразовывать данные, возвращаемые из базы, в объекты или списки объектов. Эти объекты и списки являются простыми Java-объектами, следовательно, они отключены от базы данных |
| `org.springframework.jdbc.support` | Наиболее важным средством в этом пакете является поддержка трансляции `SQLException`. Это позволяет Spring распознавать коды ошибок, используемые базой данных, и отображать их на исключения более высокого уровня |
| `org.springframework.jdbc.config` | Содержит классы, которые поддерживают конфигурацию JDBC внутри `ApplicationContext`. Например, этот пакет включает класс обработчика для пространства имен `jdbc` (скажем, для дескрипторов `<jdbc:embedded-database>`) |

Давайте начнем обсуждение поддержки JDBC в Spring с рассмотрения функциональности самого низкого уровня. Первое, что необходимо сделать перед запуском SQL-запросов — установить подключение к базе данных.

## Подключения к базе данных и источники данных

Для управления подключением к базе данных можно использовать платформу Spring, определив бин, который реализует интерфейс `javax.sql.DataSource`. Отличие между `DataSource` и `Connection` состоит в том, что `DataSource` предоставляет и управляет набором реализаций `Connection`.

Простейшей реализацией `DataSource` является `DriverManagerDataSource` (из пакета `org.springframework.jdbc.datasource`). Глядя на имя класса, можно предположить, что он просто обращается к `DriverManager` для получения подключения. Тот факт, что `DriverManagerDataSource` не поддерживает пул подключений к базе данных, делает этот класс неподходящим ни для каких других целей кроме тестирования. Конфигурация `DriverManagerDataSource` довольно проста, как легко заметить в листинге 6.10; нужно лишь указать имя класса драйвера, URL подключения, имя пользователя и пароль (`datasource-drivermanager.xml`).

**Листинг 6.10. Бин `dataSource` типа `DriverManagerDataSource`, управляемый Spring**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">
```

```
    <bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      p:driverClassName="${jdbc.driverClassName}"
      p:url="${jdbc.url}"
      p:username="${jdbc.username}"
      p:password="${jdbc.password}"/>

    <context:property-placeholder
            location="classpath:META-INF/config/jdbc.properties"/>
</beans>
```

Вы наверняка распознали свойства в этом листинге. Они представляют значения, которые обычно передаются JDBC для получения реализации интерфейса `Connection`. Информация подключения к базе данных обычно хранится в файле свойств для упрощения обслуживания и модификации под нужды разных сред развертывания. В листинге 6.11 показано содержимое файла `jdbc.properties`, из которого заполнитель свойств Spring будет загружать информацию о подключении.

**Листинг 6.11. Файл `jdbc.properties`**

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/prospring4_ch6
jdbc.username=prospring4
jdbc.password=prospring4
```

В реальных приложениях можно применять доступный в рамках проекта Apache Commons класс `BasicDataSource` (`http://commons.apache.org/dbcp/`) либо класс источника данных, реализованный сервером приложений JEE (например, JBoss, WebSphere, WebLogic или GlassFish), который может дополнительно увеличить производительность приложения. Источник данных можно было бы использовать в простом коде JDBC и получить те же преимущества организации пула; однако в большинстве случаев возможность централизованного конфигурирования источника данных будет по-прежнему отсутствовать. С другой стороны, Spring позволяет объявлять бин `dataSource` и устанавливать свойства подключения в файлах определений `ApplicationContext` (в листинге 6.12 приведено содержимое файла `datasource-dbcp.xml`).

**Листинг 6.12. Бин `dataSource`, управляемый Spring**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close"
```

```
    p:driverClassName="${jdbc.driverClassName}"
    p:url="${jdbc.url}"
    p:username="${jdbc.username}"
    p:password="${jdbc.password}"/>

  <context:property-placeholder
          location="classpath:META-INF/config/jdbc.properties"/>
</beans>
```

**На заметку!** Кроме класса `BasicDataSource` из Apache Commons доступны и другие популярные библиотеки с открытым кодом для поддержки пула подключений к базам данных, в числе которых C3P0 (`www.mchange.com/projects/c3p0/index.html`) и BoneCP (`http://jolbox.com/`).

Показанный источник данных, управляемый Spring, реализован в `org.apache.commons.dbcp.BasicDataSource`. Наиболее важный момент заключается в том, что бин `dataSource` реализует интерфейс `javax.sql.DataSource`, и его можно непосредственно применять в своих классах доступа к данным.

Другой способ конфигурирования бина `dataSource` предусматривает использование JNDI. Если разрабатываемое приложение должно выполняться в контейнере JEE, можно извлечь преимущества от пула подключений, управляемого контейнером. Для работы с источником данных, который основан на JNDI, понадобится изменить объявление бина `dataSource`, как показано в листинге 6.13 (`datasource-jndi.xml`).

**Листинг 6.13. Бин `dataSource`, основанный на JNDI и управляемый Spring**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean").
     p:jndiName="java:comp/env/jdbc/prospring4ch6"/>
</beans>
```

В предыдущем примере мы применяли бин `JndiObjectFactoryBean` из Spring для получения источника данных с помощью поиска JNDI. Начиная с версии 2.5, платформа Spring предоставляет пространство имен `jee`, которое дополнительно упрощает конфигурирование. В листинге 6.14 приведена та же самая конфигурация источника данных JNDI, использующая пространство имен `jee` (`datasource-jee.xml`).

**Листинг 6.14. Бин `dataSource`, основанный на JNDI и управляемый Spring (используется пространство имен `jee`)**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jee="http://www.springframework.org/schema/jee"
```

```
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee.xsd">

    <jee:jndi-lookup jndi-name="java:comp/env/jdbc/prospring4ch6"/>
</beans>
```

В показанном листинге мы объявляем пространство имен `jee` в дескрипторе `<beans>` и затем посредством дескриптора `<jee:jndi-lookup>` объявляем источник данных.

Принимая подход JNDI, нужно не забыть о добавлении ссылки на ресурс (`resource-ref`) в файл описателя приложения (листинг 6.15).

**Листинг 6.15. Ссылка на ресурс в файле описателя приложения**

```
<root-node>
    <resource-ref>
        <res-ref-name>jdbc/prospring4ch6</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
</root-node>
```

Дескриптор `<root-node>` является заполнителем; его необходимо изменить в зависимости от того, как упакован ваш модуль. Например, он становится `<web-app>` в описателе веб-развертывания (`WEB-INF/web.xml`), если приложение представляет собой веб-модуль. Скорее всего, потребуется также сконфигурировать `resource-ref` в файле описателя, специфичном для сервера приложений. Однако обратите внимание, что в элементе `resource-ref` указано имя ссылки `jdbc/prospring4ch6`, а свойство `jndiName` бина `dataSource` установлено в `java:comp/env/jdbc/prospring4ch6`.

Как видите, платформа Spring позволяет конфигурировать источник данных практически любым желаемым образом, и она скрывает действительную реализацию или местоположение источника данных от остальной части кода приложения. Другими словами, ваши классы DAO не знают, да и не обязаны знать, на что указывает источник данных.

Управление подключениями также делегировано бину `dataSource`, который, в свою очередь, осуществляет управление самостоятельно либо использует для выполнения всей работы контейнер JEE.

## Поддержка встроенной базы данных

Начиная с версии 3.0, платформа Spring также предлагает поддержку встроенной базы данных, которая автоматически запускает такую базу данных и делает ее доступной приложению в виде источника данных. В листинге 6.16 показана конфигурация встроенной базы данных (`app-context-xml.xml`).

**Листинг 6.16. Поддержка встроенной базы данных в Spring**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc.xsd">
   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:META-INF/sql/schema.sql"/>
      <jdbc:script location="classpath:META-INF/sql/test-data.sql"/>
   </jdbc:embedded-database>
</beans>
```

В предыдущем листинге мы сначала объявляем пространство имен `jdbc` в дескрипторе `<beans>`. После этого мы используем дескриптор `<jdbc:embedded-database>` для объявления встроенной базы данных и назначаем ей идентификатор `dataSource`. Внутри этого дескриптора мы также сообщаем Spring о необходимости выполнения указанных сценариев, которые предназначены для создания схемы базы данных и наполнения ее тестовыми данными. Обратите внимание, что порядок следования сценариев является важным: файл с командами DDL (Data Definition Language — язык определения данных) всегда должен указываться первым, а за ним — файл с командами DML (Data Manipulation Language — язык манипулирования данными). В атрибуте `type` задается тип используемой встроенной базы данных. В версии Spring 4.0 поддерживаются типы HSQL (стандартный), H2 и DERBY.

Поддержка встроенной базы данных исключительно полезна при локальной разработке или модульном тестировании. В оставшейся части этой главы мы будем применять встроенную базу данных для запуска кода примеров, так что устанавливать какую-либо СУБД на вашей машине не понадобится.

Вы можете не только задействовать поддержку встроенной базы данных через пространство имен JDBC, но также инициализировать экземпляр базы данных, функционирующий где-то в другом месте, такой как MySQL, Oracle и т.д. Вместо указания `type` и `embedded-database` просто используйте `initialize-database`, и ваши сценарии будут выполняться в отношении заданного источника данных, как если бы речь шла о встроенной базе данных.

# Использование источников данных в классах DAO

Для реализации примера давайте создадим интерфейс `ContactDao`, как показано в листинге 6.17.

**Листинг 6.17. Интерфейс `ContactDao`**

```
package com.apress.prospring4.ch6;
public interface ContactDao {
   String findLastNameById(Long id);
}
```

В качестве простой реализации мы добавим свойство dataSource в класс JdbcContactDao. Причина, по которой мы хотим добавить свойство dataSource в класс реализации, а не в интерфейс, должна быть достаточно очевидной: интерфейс не обязан знать, каким образом данные будут извлекаться и обновляться. Добавляя методы, модифицирующие источник данных, в интерфейс, мы в лучшем случае вынуждаем реализации объявлять заглушки для методов получения и установки. Понятно, что такой подход нельзя считать удачным. Взгляните на простой класс JdbcContactDao в листинге 6.18.

**Листинг 6.18. Класс JdbcUserDao со свойством dataSource**

```
package com.apress.prospring4.ch6;

import javax.sql.DataSource;

public class JdbcContactDao implements ContactDao {
   private DataSource dataSource;

   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
   }

   ...
}
```

Теперь мы можем сообщить Spring о необходимости конфигурирования бина contactDao с использованием реализации JdbcContactDao и установки свойства dataSource (содержимое файла app-context-xml.xml показано в листинге 6.19).

**Листинг 6.19. Файл контекста приложения Spring с бинами dataSource и contactDao**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:META-INF/sql/schema.sql"/>
      <jdbc:script location="classpath:META-INF/sql/test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="contactDao" class="com.apress.prospring4.ch6.JdbcContactDao"
     p:dataSource-ref="dataSource"/>
</beans>
```

Для поддержки базы данных H2 необходимо добавить к проекту соответствующую зависимость (табл. 6.3).

**Таблица 6.3. Зависимость для базы данных H2**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| com.h2database | h2 | 1.3.172 | Java-библиотека для базы данных H2 |

Платформа Spring теперь создает бин `contactDao` за счет получения экземпляра класса `JdbcContactDao` со свойством `dataSource`, установленным в бин `dataSource`.

Передовой опыт предусматривает обеспечение установки всех обязательных свойств бина. Проще всего это сделать, реализовав интерфейс `InitializingBean` и предоставив реализацию для метода `afterPropertiesSet()` (листинг 6.20). В таком случае вы гарантируете, что все обязательные свойства `JdbcContactDao` корректным образом установлены. За дополнительными сведениями об инициализации бинов обращайтесь в главу 4.

**Листинг 6.20. Класс `JdbcContactDao` с реализацией интерфейса `InitializingBean`**

```
package com.apress.prospring4.ch6;

import javax.sql.DataSource;

import org.springframework.beans.factory.BeanCreationException;
import org.springframework.beans.factory.InitializingBean;

public class JdbcContactDao implements ContactDao, InitializingBean {
    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    ...

    @Override
    public void afterPropertiesSet() throws Exception {
        if (dataSource == null) {
            // Свойство dataSource в ContactDao должно быть установлено
            throw new BeanCreationException("Must set dataSource on ContactDao");
        }
    }
}
```

В коде, показанном до сих пор, для управления источником данных применялась платформа Spring, а также вводился интерфейс `ContactDao` и его реализация JDBC. Мы также устанавливали свойство `dataSource` класса `JdbcContactDao` в файле для `ApplicationContext`. Теперь мы расширим код, добавив к интерфейсу и реализации действительные операции DAO.

# Обработка исключений

Поскольку в Spring рекомендуется использовать исключения времени выполнения, а не проверяемые исключения, необходим механизм трансляции проверяемого исключения `SQLException` в исключение времени выполнения Spring JDBC.

Так как исключения, связанные с SQL в Spring, представляют собой исключения времени выполнения, они могут быть намного более детализированными, чем проверяемые исключения. По определению это не является характеристикой исключений времени выполнения, но объявлять длинный список проверяемых исключений в конструкции `throws` весьма неудобно; следовательно, проверяемые исключения, как правило, менее детализированы, чем их эквиваленты времени выполнения.

Платформа Spring предоставляет стандартную реализацию интерфейса `SQLExceptionTranslator`, которая берет на себя обязанности по трансляции обобщенных кодов ошибок SQL в исключения Spring JDBC. В большинстве случаев этой реализации вполне достаточно, но мы можем расширить стандартную реализацию Spring и указать новую реализацию `SQLExceptionTranslator` для использования в `JdbcTemplate`, как показано в листинге 6.21.

**Листинг 6.21. Специальный класс `SQLExceptionTranslator`**

```
package com.apress.prospring4.ch6;

import java.sql.SQLException;

import org.springframework.dao.DataAccessException;
import org.springframework.dao.DeadlockLoserDataAccessException;
import org.springframework.jdbc.support.SQLErrorCodeSQLExceptionTranslator;

public class MySQLErrorCodesTranslator extends
                                SQLErrorCodeSQLExceptionTranslator {
    @Override
    protected DataAccessException customTranslate(String task,
            String sql, SQLException sqlex) {
        if (sqlex.getErrorCode() == -12345) {
            return new DeadlockLoserDataAccessException(task, sqlex);
        }
        return null;
    }
}
```

В то же время к проекту необходимо добавить зависимость для `spring-jdbc` (табл. 6.4).

**Таблица 6.4. Зависимость для `spring-jdbc`**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.springframework` | `spring-jdbc` | 4.0.2.RELEASE | Модуль Spring JDBC |

Чтобы использовать специальный транслятор, его необходимо передать `JdbcTemplate` в классах DAO. В листинге 6.22 представлен фрагмент кода из расширенного метода `JdbcContactDao.setDataSource()` для иллюстрации его применения.

**Листинг 6.22. Использование специального класса `SQLExceptionTranslator` в Spring JDBC**

```
public void setDataSource(DataSource dataSource) {
   this.dataSource = dataSource;

   JdbcTemplate jdbcTemplate = new JdbcTemplate();
   jdbcTemplate.setDataSource(dataSource);

   MySQLErrorCodesTranslator errorTranslator =
      new MySQLErrorCodesTranslator();

   errorTranslator.setDataSource(dataSource);

   jdbcTemplate.setExceptionTranslator(errorTranslator);

   this.jdbcTemplate = jdbcTemplate;
}
```

Платформа Spring будет вызывать этот специальный транслятор исключений SQL при обнаружении исключений SQL во время выполнения SQL-операторов в базе данных, и трансляция произойдет, когда код ошибки равен –12345. Для других ошибок Spring применит свой стандартный механизм трансляции исключений.

Очевидно, ничто не мешает создать `SQLExceptionTranslator` в виде управляемого Spring бина и использовать бин `JdbcTemplate` в своих классах DAO. Не переживайте, если не помните прочитанное о классе `JdbcTemplate`; мы собираемся обсудить его более подробно.

# Класс `JdbcTemplate`

Этот класс представляет ядро поддержки JDBC в Spring. Он способен выполнять все типы SQL-операторов. Выражаясь упрощенно, операторы могут быть разделены на определяющие данные и манипулирующие данными. *Операторы определения данных* отвечают за создание разнообразных объектов базы данных (таблиц, представлений, хранимых процедур и т.п.). *Операторы манипулирования данными* взаимодействуют с данными и могут быть классифицированы на операторы выборки и операторы обновления. *Оператор выборки* в общем случае возвращает набор строк, каждая из которых содержит один и тот же набор столбцов. *Оператор обновления* модифицирует данные в базе, но никакого результата не возвращает.

Класс `JdbcTemplate` позволяет отправлять базе данных SQL-оператор любого типа и возвращать результат также любого типа.

В этом разделе мы рассмотрим общие сценарии применения поддержки JDBC в Spring с помощью класса `JdbcTemplate`.

## Инициализация `JdbcTemplate` в классе DAO

Перед обсуждением использования `JdbcTemplate` давайте посмотрим, как подготовить `JdbcTemplate` для применения в классе DAO. Делается это очень просто; в большинстве случаев нужно лишь сконструировать экземпляр класса, передав объект источника данных (который должен быть внедрен платформой Spring в класс DAO). В листинге 6.23 приведен фрагмент кода, инициализирующего объект `JdbcTemplate`.

**Листинг 6.23. Инициализация JdbcTemplate**

```
private JdbcTemplate jdbcTemplate;

private DataSource dataSource;

public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    this.jdbcTemplate = new JdbcTemplate(dataSource);
}
```

Общая практика предусматривает инициализацию JdbcTemplate внутри метода установки источника данных, следовательно, как только источник данных внедряется Spring, JdbcTemplate будет также инициализирован и готов к использованию.

После конфигурирования объект JdbcTemplate является безопасным в отношении потоков. Это означает, что можно также инициализировать одиночный экземпляр JdbcTemplate в XML-файле конфигурации Spring и внедрить его во все бины DAO.

**На заметку!** В Spring-модуле Jdbc имеется класс по имени JdbcDaoSupport. Он является оболочкой для класса JdbcTemplate и ваши классы DAO могут расширять класс JdbcDaoSupport. В этом случае после внедрения такого класса DAO с источником данных экземпляр JdbcTemplate будет инициализирован автоматически.

## Извлечение одиночного значения с использованием класса JdbcTemplate

Начнем с простого запроса, который возвращает одиночное значение. Например, мы хотим иметь возможность извлекать имя контакта по его идентификатору. Извлекать значение с применением JdbcTemplate несложно. В листинге 6.24 показана реализация метода findFirstNameById() в классе JdbcContactDao. Для других методов предоставлены пустые реализации.

**Листинг 6.24. Использование JdbcTemplate для извлечения одиночного значения**

```
package com.apress.prospring4.ch6;

import javax.sql.DataSource;
import java.util.List;

import org.springframework.beans.factory.BeanCreationException;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcContactDao implements ContactDao, InitializingBean {
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplate;

    @Override
    public String findFirstNameById(Long id) {
        return jdbcTemplate.queryForObject(
                "select first_name from contact where id = ?",
                new Object[]{id}, String.class);
    }
```

```
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;

        JdbcTemplate jdbcTemplate = new JdbcTemplate();
        jdbcTemplate.setDataSource(dataSource);

        MySQLErrorCodesTranslator errorTranslator =
            new MySQLErrorCodesTranslator();

        errorTranslator.setDataSource(dataSource);

        jdbcTemplate.setExceptionTranslator(errorTranslator);

        this.jdbcTemplate = jdbcTemplate;
    }
    @Override
    public void afterPropertiesSet() throws Exception {
        if (dataSource == null) {
          throw new BeanCreationException("Must set dataSource on ContactDao");
        }
        if (jdbcTemplate == null) {
          throw new BeanCreationException("Null JdbcTemplate on ContactDao");
        }
    }
}
```

В предыдущем листинге для извлечения значения имени используется метод `queryForObject()` класса `JdbcTemplate`. Первый аргумент — это строка с SQL-оператором, а второй — параметры, передаваемые SQL-оператору для связывания, в формате массива объектов. Последний аргумент представляет возвращаемый тип, которым в рассматриваемом случае является `String`. Кроме `Object` можно также запрашивать другие типы, такие как `Long` и `Integer`. Давайте посмотрим на результат. В листинге 6.25 приведен код тестовой программы.

**Листинг 6.25. Выборка данных с применением `JdbcTemplate`**

```
package com.apress.prospring4.ch6;

import org.springframework.context.support.GenericXmlApplicationContext;

public class JdbcContactDaoSample {
    public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-xml.xml");
      ctx.refresh();

      ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

      System.out.println("First name for contact id 1 is: " +
          contactDao.findFirstNameById(1l));
    }
}
```

Как и можно было ожидать, запуск этой программы дает следующий вывод:

```
First name for contact id 1 is: Chris
```

## Использование именованных параметров с NamedParameterJdbcTemplate

В предыдущем примере мы использовали обычный заполнитель (символ ?) в качестве параметров запроса и должны передавать значения параметров в виде массива Object. Когда применяется обычный заполнитель, порядок становится важным, причем порядок помещения параметров в массив должен быть таким же, как и порядок указания параметров в запросе.

Некоторые разработчики предпочитают использовать именованные параметры, чтобы гарантировать точную привязку каждого параметра. Соответствующую поддержку в Spring обеспечивает разновидность класса JdbcTemplate по имени NamedParameterJdbcTemplate (из пакета org.springframework.jdbc.core.namedparam).

Инициализация NamedParameterJdbcTemplate совпадает с инициализацией JdbcTemplate, так что нужно просто объявить переменную типа NamedParameterJdbcTemplate и создать новый ее экземпляр в методе setDataSource(), как показано в листинге 6.26.

**Листинг 6.26. Использование NamedParameterJdbcTemplate для нахождения фамилии по идентификатору**

```
package com.apress.prospring4.ch6;

import javax.sql.DataSource;
import java.util.List;
import java.util.Map;
import java.util.HashMap;

import org.springframework.beans.factory.BeanCreationException;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;

public class JdbcContactDao implements ContactDao, InitializingBean {
    private DataSource dataSource;
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    @Override
    public String findLastNameById(Long id) {
        String sql = "select last_name from contact where id = :contactId";

        Map<String, Object> namedParameters = new HashMap<String, Object>();
        namedParameters.put("contactId", id);

        return namedParameterJdbcTemplate.queryForObject(sql,
            namedParameters, String.class);
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;

        NamedParameterJdbcTemplate namedParameterJdbcTemplate =
            new NamedParameterJdbcTemplate(dataSource);

        this.namedParameterJdbcTemplate = namedParameterJdbcTemplate;
    }
```

```
    @Override
    public void afterPropertiesSet() throws Exception {
        if (dataSource == null) {
            throw new BeanCreationException("Must set dataSource on ContactDao");
        }
        if (namedParameterJdbcTemplate == null) {
            throw new BeanCreationException("Null NamedParameterJdbcTemplate on
ContactDao");
        }
    }
}
```

Вы видите, что вместо заполнителя ? применяется именованный параметр (предваряемый двоеточием). В листинге 6.27 приведен модифицированный код тестовой программы.

**Листинг 6.27. Модифицированный код тестовой программы `JdbcContactDaoSample`, в которой используются именованные параметры**

```
package com.apress.prospring4.ch6;

import org.springframework.context.support.GenericXmlApplicationContext;

public class JdbcContactDaoSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

        System.out.println("Last name for contact id 1 is: " +
            contactDao.findLastNameById(1l));
    }
}
```

Запустив программу снова, на этот раз с применением именованных параметров, мы получим следующий вывод:

```
Last name for contact id 1 is: Schaefer
```

## Извлечение объектов предметной области с помощью `RowMapper<T>`

Большую часть времени вместо извлечения одиночного значения вам требуется запрашивать одну или более строк и затем трансформировать каждую строку в соответствующий объект предметной области. Интерфейс `RowMapper<T>` в Spring (из пакета `org.springframework.jdbc.core`) поддерживает простой способ отображения результирующего набора JDBC на объекты POJO. Давайте посмотрим на него в действии, реализовав метод `findAll()` интерфейса `ContactDao` с использованием интерфейса `RowMapper<T>`. Реализация метода `findAll()` показана в листинге 6.28.

**Листинг 6.28. Использование RowMapper<T> для запроса объектов предметной области**

```
package com.apress.prospring4.ch6;

import javax.sql.DataSource;
import java.sql.SQLException;
import java.sql.ResultSet;

import java.util.List;
import java.util.Map;
import java.util.HashMap;

import org.springframework.beans.factory.BeanCreationException;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

public class JdbcContactDao implements ContactDao, InitializingBean {
    private DataSource dataSource;
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    @Override
    public String findLastNameById(Long id) {
        String sql = "select last_name from contact where id = :contactId";

        Map<String, Object> namedParameters = new HashMap<String, Object>();
        namedParameters.put("contactId", id);

        return namedParameterJdbcTemplate.queryForObject(sql,
            namedParameters, String.class);
    }

    @Override
    public List<Contact> findAll() {
      String sql = "select id, first_name, last_name, birth_date from contact";
      return namedParameterJdbcTemplate.query(sql, new ContactMapper());
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;

        NamedParameterJdbcTemplate namedParameterJdbcTemplate =
            new NamedParameterJdbcTemplate(dataSource);

        this.namedParameterJdbcTemplate = namedParameterJdbcTemplate;
    }

    @Override
    public void afterPropertiesSet() throws Exception {>
        if (dataSource == null) {
          throw new BeanCreationException("Must set dataSource on ContactDao");
        }

        if (namedParameterJdbcTemplate == null) {
            throw new BeanCreationException("Null NamedParameterJdbcTemplate
on ContactDao");
        }
    }

    private static final class ContactMapper implements RowMapper<Contact> {
        @Override
```

```
        public Contact mapRow(ResultSet rs, int rowNum) throws SQLException {
            Contact contact = new Contact();
            contact.setId(rs.getLong("id"));
            contact.setFirstName(rs.getString("first_name"));
            contact.setLastName(rs.getString("last_name"));
            contact.setBirthDate(rs.getDate("birth_date"));

            return contact;
        }
    }
}
```

В предыдущем листинге мы определяем статический внутренний класс по имени `ContactMapper`, реализующий интерфейс `RowMapper<T>`. Класс должен предоставить реализацию `mapRow()`, которая трансформирует значения конкретной записи результирующего набора в желаемый объект предметной области. Объявление его как статического внутреннего класса позволяет совместно использовать `RowMapper<T>` множеством методов поиска.

### Рефакторинг с использованием лямбда-выражения

При работе с Java 8 вместо создания класса `ContactMapper`, как было показано ранее, можно применить лямбда-выражение:

```
@Override
public List<Contact> findAll() {
  String sql = "select id, first_name, last_name, birth_date from contact";
  return namedParameterJdbcTemplate.query(sql, (rs, rowNum) -> {
        Contact contact = new Contact();
        contact.setId(rs.getLong("id"));
        contact.setFirstName(rs.getString("first_name"));
        contact.setLastName(rs.getString("last_name"));
        contact.setBirthDate(rs.getDate("birth_date"));

        return contact; >
  });
}
```

Затем метод `findAll()` просто должен вызвать метод запроса, передав ему строку запроса и экземпляр отображения строки. На случай, когда запрос требует параметры, для метода `query()` предоставляется перегруженная версия, которая принимает параметры запроса.

Давайте создадим класс `JdbcContactDaoSample` для нахождения всех контактов (листинг 6.29).

**Листинг 6.29. Вывод списка контактов**

```
package com.apress.prospring4.ch6;

import java.util.List;
import org.springframework.context.support.GenericXmlApplicationContext;
```

```
public class JdbcContactDaoSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();>
        ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);
        List<Contact> contacts = contactDao.findAll();
        for (Contact contact: contacts) {
            System.out.println(contact);
            if (contact.getContactTelDetails() != null) {
                for (ContactTelDetail contactTelDetail:
                    contact.getContactTelDetails()) {
                    System.out.println("---" + contactTelDetail);
                }
            }
            System.out.println();
        }
    }
}
```

Запуск этой программы дает следующий результат:

```
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

## Извлечение вложенных объектов предметной области с помощью `ResultSetExtractor`

Давайте перейдем к рассмотрению более сложного примера, в котором необходимо извлечь данные из родительской (`CONTACT`) и дочерней (`CONTACT_TEL_DETAIL`) таблиц с помощью соединения, а затем соответствующим образом трансформировать данные во вложенный объект (`ContactTelDetail` внутри `Contact`).

Упомянутый ранее интерфейс `RowMapper<T>` подходит только для отображения строки на одиночный объект предметной области. Для более сложной объектной структуры должен использоваться интерфейс `ResultSetExtractor`. Чтобы продемонстрировать его применение, мы добавим к интерфейсу `ContactDao` еще один метод по имени `findAllWithDetail()`. Этот метод должен наполнить список контактов информацией о телефонах.

В листинге 6.30 демонстрируется добавление метода `findAllWithDetail()` к интерфейсу и реализация этого метода с использованием `ResultSetExtractor`.

**Листинг 6.30. Использование `ResultSetExtractor` для запроса объектов предметной области**

```
package com.apress.prospring4.ch6;

import java.util.List;

public interface ContactDao {
    String findLastNameById(Long id);
    List<Contact> findAllWithDetail();
}
```

```
package com.apress.prospring4.ch6;

import javax.sql.DataSource;
import java.util.List;
import java.util.Map;
import java.util.HashMap;
import java.util.ArrayList;
import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.beans.factory.BeanCreationException;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;

public class JdbcContactDao implements ContactDao, InitializingBean {
    private DataSource dataSource;
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    @Override
    public String findLastNameById(Long id) {
        String sql = "select last_name from contact where id = :contactId";

        Map<String, Object> namedParameters = new HashMap<String, Object>();
        namedParameters.put("contactId", id);

        return namedParameterJdbcTemplate.queryForObject(sql,
            namedParameters, String.class);
    }

    @Override
    public List<Contact> findAllWithDetail() {
      String sql = "select c.id, c.first_name, c.last_name, c.birth_date" +
          ", t.id as contact_tel_id, t.tel_type, t.tel_number from contact c " +
          "left join contact_tel_detail t on c.id = t.contact_id";

      return namedParameterJdbcTemplate.query(sql, new ContactWithDetailExtractor());
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;

        NamedParameterJdbcTemplate namedParameterJdbcTemplate =
            new NamedParameterJdbcTemplate(dataSource);

        this.namedParameterJdbcTemplate = namedParameterJdbcTemplate;
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        if (dataSource == null) {
          throw new BeanCreationException("Must set dataSource on ContactDao");
        }

        if (namedParameterJdbcTemplate == null) {
            throw new BeanCreationException("Null NamedParameterJdbcTemplate
on ContactDao");
        }
    }
```

```
    private static final class ContactWithDetailExtractor implements
                                        ResultSetExtractor<List<Contact>> {
        @Override
        public List<Contact> extractData(ResultSet rs) throws SQLException,
                                                    DataAccessException {
            Map<Long, Contact> map = new HashMap<Long, Contact>();
            Contact contact = null;

            while (rs.next()) {
                Long id = rs.getLong("id");
                contact = map.get(id);

                if (contact == null) {
                  contact = new Contact();
                  contact.setId(id);
                  contact.setFirstName(rs.getString("first_name"));
                  contact.setLastName(rs.getString("last_name"));
                  contact.setBirthDate(rs.getDate("birth_date"));
                  contact.setContactTelDetails(new ArrayList<ContactTelDetail>());
                  map.put(id, contact);
                }

                Long contactTelDetailId = rs.getLong("contact_tel_id");

                if (contactTelDetailId > 0) {
                  ContactTelDetail contactTelDetail = new ContactTelDetail();
                  contactTelDetail.setId(contactTelDetailId);
                  contactTelDetail.setContactId(id);
                  contactTelDetail.setTelType(rs.getString("tel_type"));
                  contactTelDetail.setTelNumber(rs.getString("tel_number"));
                  contact.getContactTelDetails().add(contactTelDetail);
                }
            }

            return new ArrayList<Contact> (map.values());
        }
    }
}
```

Код похож на пример `RowMapper`, но на этот раз мы объявляем внутренний класс, который реализует `ResultSetExtractor`. Затем мы реализуем метод `extractData()` для трансформации результирующего набора в список объектов `Contact`. В методе `findAllWithDetail()` запрос использует левое соединение двух таблиц, так что контакты без телефонов также будут извлечены. Результатом является декартово произведение двух таблиц. Наконец, мы вызываем метод `JdbcTemplate.query()`, передавая ему строку запроса и экземпляр реализации `ResultSetExtractor`.

### Рефакторинг с использованием лямбда-выражения

При работе с Java 8 вместо создания класса `ContactWithDetailExtractor`, как было показано ранее, можно применить лямбда-выражение:

```
@Override
public List<Contact> findAllWithDetail() {
```

```
    String sql = "select c.id, c.first_name, c.last_name, c.birth_date" +
      ", t.id as contact_tel_id, t.tel_type, t.tel_number from contact c " +
      "left join contact_tel_detail t on c.id = t.contact_id";

    return namedParameterJdbcTemplate.query(sql, (ResultSet rs) -> {
        Map<Long, Contact> map = new HashMap<Long, Contact>();
        Contact contact = null;

        while (rs.next()) {
            Long id = rs.getLong("id");
            contact = map.get(id);

            if (contact == null) {
              contact = new Contact();
              contact.setId(id);
              contact.setFirstName(rs.getString("first_name"));
              contact.setLastName(rs.getString("last_name"));
              contact.setBirthDate(rs.getDate("birth_date"));
              contact.setContactTelDetails(new ArrayList<ContactTelDetail>());
              map.put(id, contact);
            }

            Long contactTelDetailId = rs.getLong("contact_tel_id");

            if (contactTelDetailId > 0) {
                ContactTelDetail contactTelDetail = new ContactTelDetail();
                contactTelDetail.setId(contactTelDetailId);
                contactTelDetail.setContactId(id);
                contactTelDetail.setTelType(rs.getString("tel_type"));
                contactTelDetail.setTelNumber(rs.getString("tel_number"));
                contact.getContactTelDetails().add(contactTelDetail);
            }
        }

        return new ArrayList<Contact> (map.values());
    });
}
```

В листинге 6.31 приведен новый код класса `JdbcContactDaoSample`, который выводит список контактов с детальной информацией.

**Листинг 6.31. Вывод списка контактов с применением `ResultSetExtractor`**

```
package com.apress.prospring4.ch6;

import java.util.List;
import org.springframework.context.support.GenericXmlApplicationContext;

public class JdbcContactDaoSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-xml.xml");
        ctx.refresh();

        ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

        List<Contact> contactsWithDetail = contactDao.findAllWithDetail();

        for (Contact contact: contactsWithDetail) {
            System.out.println(contact);
```

```
            if (contact.getContactTelDetails() != null) {
              for (ContactTelDetail contactTelDetail: contact.getContactTelDetails()) {
                  System.out.println("---" + contactTelDetail);
              }
            }
            System.out.println();
        }
    }
}
```

Запустив тестовую программу еще раз, получаем следующий вывод:

```
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
---Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
---Contact Tel Detail - Id: 2, Contact id: 1, Type: Home, Number: 1234567890
---Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
```

Как видите, контакты и сведения о связанных с ними телефонах выведены должным образом. Данные были сгенерированы с помощью сценария из листинга 6.2.

До сих пор вы видели, как применять `JdbcTemplate` для выполнения ряда общих операций запросов. Класс `JdbcTemplate` (а также класс `NamedParameterJdbcTemplate`) предлагает несколько перегруженных методов `update()`, которые поддерживают операции обновления данных, включая вставку, обновление, удаление и т.д. Однако поскольку метод `update()` вполне самоочевиден, мы оставляем его вам для самостоятельного исследования. С другой стороны, как будет показано в последующих разделах, для выполнения операций обновления данных мы будем использовать класс `SqlUpdate`, предоставляемый Spring.

# Классы Spring, моделирующие операции JDBC

В предыдущем разделе вы видели, что `JdbcTemplate` и связанные служебные классы для отображения данных существенно упрощают программную модель при разработке логики доступа к данным с помощью JDBC. Построенная поверх `JdbcTemplate` платформа Spring также предлагает множество полезных классов, которые моделируют операции над данными JDBC и позволяют разработчикам реализовывать логику запроса и трансформаций результирующего набора в объекты предметной области в более объектно-ориентированной манере. В частности, в этом разделе мы обсудим перечисленные ниже классы.

- **`MappingSqlQuery<T>`.** Класс `MappingSqlQuery<T>` позволяет поместить строку запроса вместе с методом `mapRow()` в оболочку единственного класса.
- **`SqlUpdate`.** Класс `SqlUpdate` позволяет поместить внутрь себя любой SQL-оператор обновления. Он также предоставляет множество полезных функций для привязки SQL-параметров, извлечения сгенерированного СУРБД ключа после вставки новой записи и т.д.
- **`BatchSqlUpdate`.** Класс `BatchSqlUpdate` позволяет выполнять пакетные операции обновления. Например, можно проходить в цикле по Java-объекту

List и с помощью BatchSqlUpdate заносить в очередь записи, для которых затем будут выданы операторы обновления в пакете. Можно устанавливать размер пакета и сбрасывать операцию в любой момент.

- **SqlFunction<T>.** Класс SqlFunction<T> позволяет вызывать хранимые функции в базе данных с аргументами и возвращаемым типом. Существует еще один класс, StoredProcedure, который помогает вызывать хранимые процедуры.

## Настройка DAO-классов JDBC с использованием аннотаций

Первым делом давайте посмотрим, как настроить класс реализации DAO, используя аннотации. В следующем примере кода интерфейс ContactDao реализуется метод за методом до тех пор, пока не будет получена полная его реализация. В листинге 6.32 представлен интерфейс ContactDao с завершенным списком предлагаемых им служб доступа к данным.

**Листинг 6.32. Интерфейс ContactDao**

```
package com.apress.prospring4.ch6;

import java.util.List;

public interface ContactDao {
    List<Contact> findAll();
    List<Contact> findByFirstName(String firstName);
    String findFirstNameById(Long id);
    List<Contact> findAllWithDetail();
    void insert(Contact contact);
    void insertWithDetail(Contact contact);
    void update(Contact contact);
}
```

В листинге 6.33 показано начальное объявление и внедрение свойства источника данных с применением аннотаций JSR-250.

**Листинг 6.33. Объявление JdbcContactDao с использованием аннотаций**

```
package com.apress.prospring4.ch6;

import javax.annotation.Resource;
import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Repository;

@Repository("contactDao")
public class JdbcContactDao implements ContactDao {
    private Log log = LogFactory.getLog(JdbcContactDao.class);

    private DataSource dataSource;

    @Resource(name="dataSource")
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

```
    public DataSource getDataSource() {
        return dataSource;
    }
    ...
}
```

В предыдущем листинге мы используем аннотацию `@Repository` для объявления бина Spring по имени `contactDao`, и поскольку класс содержит код доступа к данным, `@Repository` также инструктирует Spring о необходимости транслировать исключения SQL, специфичные для базы данных, в более дружественную к приложению иерархию `DataAccessException`, поддерживаемую платформой. Мы также объявляем переменную `log` с использованием пакета `commons-logging` из Apache для записи сообщений в журнал внутри программы. Наконец, мы применяем к свойству источника данных аннотацию `@Resource` стандарта JSR-250, чтобы позволить Spring внедрить источник данных по имени `dataSource`.

В листинге 6.34 представлена XML-конфигурация для Spring, использующая аннотации (`app-context-annotation.xml`).

**Листинг 6.34. XML-конфигурация для Spring, использующая аннотации**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/spring-jdbc.xsd">

    <context:component-scan base-package="com.apress.prospring4.ch6"/>

    <jdbc:embedded-database id="dataSource" type="H2">
        <jdbc:script location="classpath:META-INF/sql/schema.sql"/>
        <jdbc:script location="classpath:META-INF/sql/test-data.sql"/>
    </jdbc:embedded-database>
</beans>
```

В этой конфигурации мы объявляем встроенную базу данных типа H2 и применяем дескриптор `<context:component-scan>` для автоматического обнаружения бинов Spring. Построив такую инфраструктуру, мы можем переходить к реализации операций JDBC.

## Запрашивание данных с использованием `MappingSqlQuery<T>`

Для моделирования операций запросов платформа Spring предоставляет класс `MappingSqlQuery<T>`. В сущности, мы конструируем класс `MappingSqlQuery<T>`, используя источник данных и строку запроса. Затем мы реализуем метод `mapRow()` для отображения каждой записи результирующего набора в соответствующий объект предметной области.

Давайте начнем с создания класса SelectAllContacts (который представляет операцию запроса для выборки всех контактов), расширяющего абстрактный класс MappingSqlQuery<T>. Код класса SelectAllContacts приведен в листинге 6.35.

**Листинг 6.35. Класс SelectAllContacts**

```
package com.apress.prospring4.ch6;

import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

import org.springframework.jdbc.object.MappingSqlQuery;

public class SelectAllContacts extends MappingSqlQuery<Contact> {
   private static final String SQL_SELECT_ALL_CONTACT =
      "select id, first_name, last_name, birth_date from contact";

   public SelectAllContacts(DataSource dataSource) {
      super(dataSource, SQL_SELECT_ALL_CONTACT);
   }

   protected Contact mapRow(ResultSet rs, int rowNum) throws SQLException {
      Contact contact = new Contact();

      contact.setId(rs.getLong("id"));
      contact.setFirstName(rs.getString("first_name"));
      contact.setLastName(rs.getString("last_name"));
      contact.setBirthDate(rs.getDate("birth_date"));

      return contact;
   }
}
```

Внутри класса SelectAllContacts объявляется SQL-оператор для выборки всех контактов. В конструкторе класса вызывается метод super() для создания экземпляра класса, используя источник данных и этот SQL-оператор. Кроме того, реализован метод MappingSqlQuery<T>.mapRow(), предназначенный для отображения результирующего набора на объект предметной области Contact.

Имея класс SelectAllContacts, мы можем реализовать метод findAll() в классе JdbcContactDao, как показано в листинге 6.36.

**Листинг 6.36. Реализация метода findAll()**

```
package com.apress.prospring4.ch6;

import java.util.List;

import javax.annotation.Resource;
import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Repository;

@Repository("contactDao")
public class JdbcContactDao implements ContactDao {
   private static final Log LOG = LogFactory.getLog(JdbcContactDao.class);
```

```
    private DataSource dataSource;
    private SelectAllContacts selectAllContacts;

    @Override
    public List<Contact> findAll() {
        return selectAllContacts.execute();
    }

    @Override
    public List<Contact> findByFirstName(String firstName) {
        return null;
    }

    @Override
    public String findFirstNameById(Long id) {.
        return null;
    }

    @Override
    public void insert(Contact contact) {
    }

    @Override
    public void update(Contact contact) {
    }

    @Resource(name="dataSource")
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
        this.selectAllContacts = new SelectAllContacts(dataSource);
    }
    public DataSource getDataSource() {
        return dataSource;
    }
}
```

В методе `setDataSource()` при внедрении источника данных создается экземпляр класса `SelectAllContacts`. В методе `findAll()` мы просто вызываем метод `SelectAllContacts.execute()`, который непрямо унаследован из абстрактного класса `SqlQuery<T>`. Это все, что нужно было сделать. В листинге 6.37 приведен пример программы для тестирования полученной логики.

**Листинг 6.37. Тестирование `MappingSqlQuery`**

```
package com.apress.prospring4.ch6;

import java.util.List;
import org.springframework.context.support.GenericXmlApplicationContext;

public class AnnotationJdbcDaoSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();
        ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);
        List<Contact> contacts = contactDao.findAll();
        listContacts(contacts);
    }
```

```
    private static void listContacts(List<Contact> contacts) {
        for (Contact contact: contacts) {
            System.out.println(contact);
            if (contact.getContactTelDetails() != null) {
                for (ContactTelDetail contactTelDetail: contact.getContactTelDetails()) {
                    System.out.println("---" + contactTelDetail);
                }
            }
            System.out.println();
        }
    }
}
```

Запуск тестовой программы дает следующий вывод:

```
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

Из-за того, что мы указали для свойств регистрации в журнале уровень DEBUG, в выводе консоли будет также присутствовать запрос, отправленный Spring:

```
JdbcTemplate: 663 - Executing prepared SQL query
JdbcTemplate: 597 - Executing prepared SQL statement [select id,
    first_name, last_name, birth_date from contact]
```

Давайте перейдем к реализации метода `findByFirstName()`, принимающего один именованный параметр. Подобно предыдущему примеру, мы создаем для операции класс `SelectContactByFirstName`, код которого показан в листинге 6.38.

**Листинг 6.38. Класс `SelectContactByFirstName`**

```
package com.apress.prospring4.ch6;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;
import javax.sql.DataSource;
import org.springframework.jdbc.object.MappingSqlQuery;
import org.springframework.jdbc.core.SqlParameter;

public class SelectContactByFirstName extends MappingSqlQuery<Contact> {
    private static final String SQL_FIND_BY_FIRST_NAME =
        "select id, first_name, last_name, birth_date from contact where
first_name = :first_name";
    public SelectContactByFirstName(DataSource dataSource) {
        super(dataSource, SQL_FIND_BY_FIRST_NAME);
        super.declareParameter(new SqlParameter("first_name", Types.VARCHAR));
    }
    protected Contact mapRow(ResultSet rs, int rowNum) throws SQLException {
        Contact contact = new Contact();
        contact.setId(rs.getLong("id"));
        contact.setFirstName(rs.getString("first_name"));
        contact.setLastName(rs.getString("last_name"));
        contact.setBirthDate(rs.getDate("birth_date"));
        return contact;
    }
}
```

Класс SelectContactByFirstName похож на SelectAllContacts. В нем используется другой SQL-оператор с именованным параметром first_name. В методе конструктора вызывается метод declareParameter() (который непрямо унаследован от абстрактного класса org.springframework.jdbc.object.RdbmsOperation). Теперь займемся реализацией метода findByFirstName() в классе JdbcContactDao. Соответствующий код приведен в листинге 6.39.

**Листинг 6.39. Реализация метода findByFirstName()**

```
package com.apress.prospring4.ch6;
import java.util.List;
import java.util.Map;
import java.util.HashMap;
import javax.annotation.Resource;
import javax.sql.DataSource;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Repository;

@Repository("contactDao")
public class JdbcContactDao implements ContactDao {
    private static final Log LOG = LogFactory.getLog(JdbcContactDao.class);
    private DataSource dataSource;
    private SelectAllContacts selectAllContacts;
    private SelectContactByFirstName selectContactByFirstName;
    @Override
    public List<Contact> findAll() {
        return selectAllContacts.execute();
    }
    @Override
    public List<Contact> findByFirstName(String firstName) {
        Map<String, Object> paramMap = new HashMap<String, Object>();
        paramMap.put("first_name", firstName);
        return selectContactByFirstName.executeByNamedParam(paramMap);
    }
    @Override
    public String findFirstNameById(Long id) {
        return null;
    }
    @Override
    public void insert(Contact contact) {
    }
    @Override
    public void update(Contact contact) {.
    }
    @Resource(name="dataSource")
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
       this.selectAllContacts = new SelectAllContacts(dataSource);
       this.selectContactByFirstName = new SelectContactByFirstName(dataSource);
    }
    public DataSource getDataSource() {
        return dataSource;
    }
}
```

При внедрении источника данных создается экземпляр SelectContactByFirstName. После этого в методе findByFirstName() конструируется экземпляр HashMap с именованными параметрами и значениями.

Наконец, вызывается метод executeByNamedParam() (непрямо унаследованный от абстрактного класса SqlQuery<T>). Давайте протестируем этот метод посредством кода, показанного в листинге 6.40.

**Листинг 6.40. Тестирование метода findByFirstName()**

```
package com.apress.prospring4.ch6;

import java.util.List;
import org.springframework.context.support.GenericXmlApplicationContext;

public class AnnotationJdbcDaoSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();

      ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

      List<Contact> contacts = contactDao.findByFirstName("Chris");
      listContacts(contacts);
   }

   private static void listContacts(List<Contact> contacts) {
      for (Contact contact: contacts) {
         System.out.println(contact);

         if (contact.getContactTelDetails() != null) {
            for (ContactTelDetail contactTelDetail:
                    contact.getContactTelDetails()) {
               System.out.println("---" + contactTelDetail);
            }
         }

         System.out.println();
      }
   }
}
```

Запуск этой программы дает следующий вывод из метода findByFirstName():

```
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
```

Здесь важно отметить один момент: MappingSqlQuery<T> подходит только для отображения одиночной строки на объект предметной области. Для вложенного объекта по-прежнему нужно применять JdbcTemplate с ResultSetExtractor, как это делалось в примере метода findAllWithDetail(), представленного в классе JdbcTemplate.

## Обновление данных с использованием SqlUpdate

Для обновления данных Spring предоставляет класс SqlUpdate. В листинге 6.41 приведен код класса UpdateContact, который расширяет класс SqlUpdate поддержкой операций обновления.

**Листинг 6.41. Класс `UpdateContact`**

```
package com.apress.prospring4.ch6;

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class UpdateContact extends SqlUpdate {
    private static final String SQL_UPDATE_CONTACT =
        "update contact set first_name=:first_name,
              last_name=:last_name, birth_date=:birth_date where id=:id";

    public UpdateContact(DataSource dataSource) {
        super(dataSource, SQL_UPDATE_CONTACT);
        super.declareParameter(new SqlParameter("first_name", Types.VARCHAR));
        super.declareParameter(new SqlParameter("last_name", Types.VARCHAR));
        super.declareParameter(new SqlParameter("birth_date", Types.DATE));
        super.declareParameter(new SqlParameter("id", Types.INTEGER));
    }
}
```

Код в предыдущем листинге должен выглядеть знакомым. Здесь создается экземпляр класса `SqlUpdate` с запросом, а также объявляются именованные параметры. В листинге 6.42 показана реализация метода `update()` в классе `JdbcContactDao`.

**Листинг 6.42. Использование `SqlUpdate`**

```
package com.apress.prospring4.ch6;

import java.util.List;
import java.util.Map;
import java.util.HashMap;

import javax.annotation.Resource;
import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Repository;

@Repository("contactDao")
public class JdbcContactDao implements ContactDao {
    private static final Log LOG = LogFactory.getLog(JdbcContactDao.class);

    private DataSource dataSource;
    private SelectAllContacts selectAllContacts;
    private SelectContactByFirstName selectContactByFirstName;
    private UpdateContact updateContact;

    @Override
    public List<Contact> findAll() {
        return selectAllContacts.execute();
    }

    @Override
```

```
    public List<Contact> findByFirstName(String firstName) {
        Map<String, Object> paramMap = new HashMap<String, Object>();
        paramMap.put("first_name", firstName);

        return selectContactByFirstName.executeByNamedParam(paramMap);
    }

    @Override
    public String findFirstNameById(Long id) {
        return null;
    }

    @Override
    public void insert(Contact contact) {
    }

    @Override
    public void update(Contact contact) {
        Map<String, Object> paramMap = new HashMap<String, Object>();
        paramMap.put("first_name", contact.getFirstName());
        paramMap.put("last_name", contact.getLastName());
        paramMap.put("birth_date", contact.getBirthDate());
        paramMap.put("id", contact.getId());

        updateContact.updateByNamedParam(paramMap);

        LOG.info("Existing contact updated with id: " + contact.getId());
    }

    @Resource(name="dataSource")
    public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
      this.selectAllContacts = new SelectAllContacts(dataSource);
      this.selectContactByFirstName = new SelectContactByFirstName(dataSource);
      this.updateContact = new UpdateContact(dataSource);
    }

    public DataSource getDataSource() {
        return dataSource;
    }
}
```

Во время внедрения источника данных конструируется экземпляр `UpdateContact`. В методе `update()` на основе переданного объекта `Contact` создается `HashMap` из именованных параметров, после чего вызывается метод `updateByNamedParam()` для обновления записи о контакте. Чтобы протестировать операцию обновления, модифицируем класс `AnnotationJdbcDaoSample`, как показано в листинге 6.43.

**Листинг 6.43. Тестирование метода `update()`**

```
package com.apress.prospring4.ch6;

import java.util.List;
import java.util.GregorianCalendar;
import java.sql.Date;
import org.springframework.context.support.GenericXmlApplicationContext;
```

```
public class AnnotationJdbcDaoSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();

      ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

      Contact contact = new Contact();
      contact.setId(1l);
      contact.setFirstName("Chris");
      contact.setLastName("John");
      contact.setBirthDate(new Date(
          (new GregorianCalendar(1977, 10, 1)).getTime().getTime()));

       contactDao.update(contact);

      listContacts(contactDao.findAll());
   }

   private static void listContacts(List<Contact> contacts) {
       for (Contact contact: contacts) {
          System.out.println(contact);

          if (contact.getContactTelDetails() != null) {
             for (ContactTelDetail contactTelDetail:
                    contact.getContactTelDetails()) {
                System.out.println("---" + contactTelDetail);
             }
          }

          System.out.println();
       }
   }
}
```

Здесь мы просто создаем объект `Contact` и затем вызываем метод `update()`. Запуск этой программы дает следующий вывод из последнего метода `listContacts()`:

```
INFO com.apress.prospring4.ch6.JdbcContactDao:  60 - Existing contact
updated with id: 1
Contact - Id: 1, First name: Chris, Last name: John, Birthday: 1977-11-01

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

В выводе видно, что контакт с идентификатором 1 был соответствующим образом обновлен.

## Вставка данных и извлечение сгенерированного ключа

Для вставки данных мы также можем использовать класс `SqlUpdate`. Интересно отметить, каким образом генерируется первичный ключ (которым обычно является столбец с идентификатором). Это значение доступно только после завершения оператора вставки, когда СУРБД сгенерирует значение, идентифицирующее вставляемую запись. Столбец `ID` объявлен с атрибутом `AUTO_INCREMENT` и является пер-

вичным ключом, так что значение ему присваивается СУРБД во время выполнения операции вставки. Если вы работаете с Oracle, то, скорее всего, сначала получите уникальный идентификатор из последовательности Oracle и затем выполните оператор вставки с результатом запроса.

В старых версиях JDBC сделать подобное не так-то просто. Например, в случае MySQL необходимо запустить SQL-оператор `select last_insert_id()`, а в Microsoft SQL Server — `select @@IDENTITY`.

К счастью, начиная с версии JDBC 3.0, появилась новая функциональная возможность, которая позволяет извлекать сгенерированные СУРБД ключи унифицированным образом. В листинге 6.45 приведена реализация метода `insert()`, которая также извлекает сгенерированный ключ для вставленной записи контакта. Она будет работать с большинством баз данных (если только не со всеми); просто удостоверьтесь, что располагаете драйвером JDBC, который совместим с JDBC 3.0 и более новыми версиями.

Мы начинаем с создания класса `InsertContact` для операции вставки, который расширяет класс `SqlUpdate`. Код класса `InsertContact` показан в листинге 6.44.

**Листинг 6.44. Класс `InsertContact`**

```
package com.apress.prospring4.ch6;

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class InsertContact extends SqlUpdate {
    private static final String SQL_INSERT_CONTACT =
        "insert into contact (first_name, last_name, birth_date) values
(:first_name, :last_name, :birth_date)";

    public InsertContact(DataSource dataSource) {
        super(dataSource, SQL_INSERT_CONTACT);
        super.declareParameter(new SqlParameter("first_name", Types.VARCHAR));
        super.declareParameter(new SqlParameter("last_name", Types.VARCHAR));
        super.declareParameter(new SqlParameter("birth_date", Types.DATE));
        super.setGeneratedKeysColumnNames(new String[] {"id"});
        super.setReturnGeneratedKeys(true);
    }
}
```

Класс `InsertContact` в основном похож на класс `UpdateContact`; понадобится лишь сделать две дополнительных вещи.

При конструировании экземпляра класса `InsertContact` мы вызываем метод `SqlUpdate.setGeneratedKeysColumnNames()` для объявления имени столбца идентификации. Метод `SqlUpdate.setReturnGeneratedKeys()` заставляет лежащий в основе драйвер JDBC извлечь сгенерированный ключ.

В листинге 6.45 приведена реализация метода `insert()` в классе `JdbcContactDao`.

**Листинг 6.45. Использование `SqlUpdate` для операции вставки**

```
package com.apress.prospring4.ch6;

import java.util.List;
import java.util.Map;
import java.util.HashMap;

import javax.annotation.Resource;
import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Repository;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;

@Repository("contactDao")
public class JdbcContactDao implements ContactDao {
    private static final Log LOG = LogFactory.getLog(JdbcContactDao.class);

    private DataSource dataSource;
    private SelectAllContacts selectAllContacts;
    private SelectContactByFirstName selectContactByFirstName;
    private UpdateContact updateContact;
    private InsertContact insertContact;

    @Override
    public List<Contact> findAll() {
        return selectAllContacts.execute();
    }

    @Override
    public List<Contact> findByFirstName(String firstName) {
        Map<String, Object> paramMap = new HashMap<String, Object>();
        paramMap.put("first_name", firstName);

        return selectContactByFirstName.executeByNamedParam(paramMap);
    }

    @Override
    public String findFirstNameById(Long id) {
        return null;
    }

    @Override
    public void insert(Contact contact) {
        Map<String, Object> paramMap = new HashMap<String, Object>();
        paramMap.put("first_name", contact.getFirstName());
        paramMap.put("last_name", contact.getLastName());
        paramMap.put("birth_date", contact.getBirthDate());

        KeyHolder keyHolder = new GeneratedKeyHolder();

        insertContact.updateByNamedParam(paramMap, keyHolder);

        contact.setId(keyHolder.getKey().longValue());

        LOG.info("New contact inserted with id: " + contact.getId());
    }

    @Override
```

```
    public void update(Contact contact) {
        Map<String, Object> paramMap = new HashMap<String, Object>();
        paramMap.put("first_name", contact.getFirstName());
        paramMap.put("last_name", contact.getLastName());
        paramMap.put("birth_date", contact.getBirthDate());
        paramMap.put("id", contact.getId());

        updateContact.updateByNamedParam(paramMap);

        LOG.info("Existing contact updated with id: " + contact.getId());
    }

    @Resource(name="dataSource")
    public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
      this.selectAllContacts = new SelectAllContacts(dataSource);
      this.selectContactByFirstName = new SelectContactByFirstName(dataSource);
      this.updateContact = new UpdateContact(dataSource);
      this.insertContact = new InsertContact(dataSource);
    }

    public DataSource getDataSource() {
        return dataSource;
    }
}
```

При внедрении источника данных создается экземпляр `InsertContact`. Внутри `insert()` мы также используем метод `SqlUpdate.updateByNamedParam()`. Кроме того, мы передаем методу экземпляр `KeyHolder`, который будет хранить сгенерированный идентификатор. После вставки данных сгенерированный ключ можно будет извлечь из `KeyHolder`.

В листинге 6.46 показан модифицированный код класса `AnnotationJdbcDaoSample`.

**Листинг 6.46. Тестирование метода `insert()`**

```
package com.apress.prospring4.ch6;

import java.util.List;
import java.util.GregorianCalendar;
import java.sql.Date;
import org.springframework.context.support.GenericXmlApplicationContext;

public class AnnotationJdbcDaoSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new
GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();

        ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

        Contact contact = new Contact();
        contact.setFirstName("Rod");
        contact.setLastName("Johnson");
        contact.setBirthDate(new Date((new GregorianCalendar(2001, 10, 1))
                .getTime().getTime()));
```

```
        contactDao.insert(contact);
        listContacts(contactDao.findAll());
    }

    private static void listContacts(List<Contact> contacts) {
        for (Contact contact: contacts) {
            System.out.println(contact);

            if (contact.getContactTelDetails() != null) {
                for (ContactTelDetail contactTelDetail:
                        contact.getContactTelDetails()) {
                    System.out.println("---" + contactTelDetail);
                }
            }

            System.out.println();
        }
    }
}
```

В результате выполнения программы получается следующий вывод из последнего метода `listContacts()`:

```
INFO com.apress.prospring4.ch6.JdbcContactDao:  62 - New contact
inserted with id: 4
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
Contact - Id: 4, First name: Rod, Last name: Johnson, Birthday: 2001-11-01
```

Как видите, новый контакт с идентификатором 4 был вставлен и затем корректно извлечен.

## Объединение операций в пакеты с помощью `BatchSqlUpdate`

Для пакетных операций используется класс `BatchSqlUpdate`. Новый метод `insertWithDetail()` будет вставлять в базу данных информацию о контакте и его телефонах.

Чтобы можно было вставить запись о телефоне, понадобится создать класс `InsertContactTelDetail`, код которого показан в листинге 6.47.

**Листинг 6.47. Класс `InsertContactTelDetail`**

```
package com.apress.prospring4.ch6;

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.BatchSqlUpdate;

public class InsertContactTelDetail extends BatchSqlUpdate {
    private static final String SQL_INSERT_CONTACT_TEL =
        "insert into contact_tel_detail (contact_id, tel_type, tel_number)
values (:contact_id, :tel_type, :tel_number)";
```

```
    private static final int BATCH_SIZE = 10;

    public InsertContactTelDetail(DataSource dataSource) {
        super(dataSource, SQL_INSERT_CONTACT_TEL);

        declareParameter(new SqlParameter("contact_id", Types.INTEGER));
        declareParameter(new SqlParameter("tel_type", Types.VARCHAR));
        declareParameter(new SqlParameter("tel_number", Types.VARCHAR));

        setBatchSize(BATCH_SIZE);
    }
}
```

Обратите внимание, что в конструкторе с помощью вызова метода `BatchSqlUpdate.setBatchSize()` мы устанавливаем размер пакета для операции вставки JDBC.

В листинге 6.48 приведена реализация метода `insertWithDetail()` в классе `JdbcContactDao`.

**Листинг 6.48. Пакетная SQL-операция обновления**

```
package com.apress.prospring4.ch6;

import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;

import java.sql.ResultSet;
import java.sql.SQLException;

import javax.annotation.Resource;
import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Repository;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.dao.DataAccessException;

@Repository("contactDao")
public class JdbcContactDao implements ContactDao {
    private static final Log LOG = LogFactory.getLog(JdbcContactDao.class);

    private DataSource dataSource;
    private SelectAllContacts selectAllContacts;
    private SelectContactByFirstName selectContactByFirstName;
    private UpdateContact updateContact;
    private InsertContact insertContact;
    private InsertContactTelDetail insertContactTelDetail;

    @Override
    public List<Contact> findAll() {
        return selectAllContacts.execute();
    }
```

```
@Override
public List<Contact> findByFirstName(String firstName) {
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("first_name", firstName);

    return selectContactByFirstName.executeByNamedParam(paramMap);
}

@Override
public String findFirstNameById(Long id) {
    return null;
}

@Override
public void insert(Contact contact) {
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("first_name", contact.getFirstName());
    paramMap.put("last_name", contact.getLastName());
    paramMap.put("birth_date", contact.getBirthDate());

    KeyHolder keyHolder = new GeneratedKeyHolder();

    insertContact.updateByNamedParam(paramMap, keyHolder);

    contact.setId(keyHolder.getKey().longValue());

    LOG.info("New contact inserted with id: " + contact.getId());
}

@Override
public void insertWithDetail(Contact contact) {
    insertContactTelDetail = new InsertContactTelDetail(dataSource);

    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("first_name", contact.getFirstName());
    paramMap.put("last_name", contact.getLastName());
    paramMap.put("birth_date", contact.getBirthDate());

    KeyHolder keyHolder = new GeneratedKeyHolder();

    insertContact.updateByNamedParam(paramMap, keyHolder);

    contact.setId(keyHolder.getKey().longValue());

    LOG.info("New contact inserted with id: " + contact.getId());

    List<ContactTelDetail> contactTelDetails =
        contact.getContactTelDetails();

    if (contactTelDetails != null) {
        for (ContactTelDetail contactTelDetail: contactTelDetails) {
            paramMap = new HashMap<String, Object>();
            paramMap.put("contact_id", contact.getId());
            paramMap.put("tel_type", contactTelDetail.getTelType());
            paramMap.put("tel_number", contactTelDetail.getTelNumber());
            insertContactTelDetail.updateByNamedParam(paramMap);
        }
    }

    insertContactTelDetail.flush();
}

@Override
```

```
    public List<Contact> findAllWithDetail() {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(getDataSource());

        String sql = "select c.id, c.first_name, c.last_name, c.birth_date" +
", t.id as contact_tel_id, t.tel_type, t.tel_number from contact c " +
"left join contact_tel_detail t on c.id = t.contact_id";

        return jdbcTemplate.query(sql, new ContactWithDetailExtractor());
    }

    @Override
    public void update(Contact contact) {
        Map<String, Object> paramMap = new HashMap<String, Object>();
        paramMap.put("first_name", contact.getFirstName());
        paramMap.put("last_name", contact.getLastName());
        paramMap.put("birth_date", contact.getBirthDate());
        paramMap.put("id", contact.getId());

        updateContact.updateByNamedParam(paramMap);

        LOG.info("Existing contact updated with id: " + contact.getId());
    }

    @Resource(name="dataSource")
    public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
      this.selectAllContacts = new SelectAllContacts(dataSource);
      this.selectContactByFirstName = new SelectContactByFirstName(dataSource);
      this.updateContact = new UpdateContact(dataSource);
      this.insertContact = new InsertContact(dataSource);
    }

    public DataSource getDataSource() {
        return dataSource;
    }

    private static final class ContactWithDetailExtractor
        implements ResultSetExtractor<List<Contact>> {

        @Override
        public List<Contact> extractData(ResultSet rs) throws
            SQLException, DataAccessException {

            Map<Long, Contact> map = new HashMap<Long, Contact>();
            Contact contact = null;

            while (rs.next()) {
                Long id = rs.getLong("id");
                contact = map.get(id);

                if (contact == null) {
                    contact = new Contact();
                    contact.setId(id);
                    contact.setFirstName(rs.getString("first_name"));
                    contact.setLastName(rs.getString("last_name"));
                    contact.setBirthDate(rs.getDate("birth_date"));
                    contact.setContactTelDetails(new ArrayList<ContactTelDetail>());
                     map.put(id, contact);
                }
```

```
            Long contactTelDetailId = rs.getLong("contact_tel_id");
            if (contactTelDetailId > 0) {
                ContactTelDetail contactTelDetail = new ContactTelDetail();
                contactTelDetail.setId(contactTelDetailId);
                contactTelDetail.setContactId(id);
                contactTelDetail.setTelType(rs.getString("tel_type"));
                contactTelDetail.setTelNumber(rs.getString("tel_number"));
                contact.getContactTelDetails().add(contactTelDetail);
            }
        }
        return new ArrayList<Contact> (map.values());
    }
}
}
```

При каждом вызове метода `insertWithDetail()` создается новый экземпляр `InsertContactTelDetail`, т.к. класс `BatchSqlUpdate` не является безопасным в отношении потоков. Далее он используется подобно `SqlUpdate`. Основное отличие заключается в том, что класс `BatchSqlUpdate` будет помещать операции вставки в очередь и затем отправлять их базе данных в виде пакета. Каждый раз, когда количество записей становится равным размеру пакета, Spring запускает в базе данных операцию групповой вставки для ожидающих записей. С другой стороны, по завершении мы вызываем метод `BatchSqlUpdate.flush()`, чтобы заставить Spring сбросить все ожидающие операции (т.е. операции вставки, находящиеся в очереди, количество которых пока не достигло размера пакета). Наконец, мы проходим в цикле по списку объектов `ContactTelDetail` внутри объекта `Contact` и вызываем метод `BatchSqlUpdate.updateByNamedParam()`.

Для упрощения тестирования также реализован метод `findAllWithDetail()`.

### Рефакторинг с использованием лямбда-выражения

При работе с Java 8 вместо создания класса `ContactWithDetailExtractor`, как было показано ранее, можно применить лямбда-выражение:

```
@Override
public List<Contact> findAllWithDetail() {
    JdbcTemplate jdbcTemplate = new JdbcTemplate(getDataSource());
    String sql = "select c.id, c.first_name, c.last_name, c.birth_date" +
", t.id as contact_tel_id, t.tel_type, t.tel_number from contact c " +
"left join contact_tel_detail t on c.id = t.contact_id";
    return jdbcTemplate.query(sql, (ResultSet rs) -> {
        Map<Long, Contact> map = new HashMap<Long, Contact>();
        Contact contact = null;
        while (rs.next()) {
            Long id = rs.getLong("id");
            contact = map.get(id);
            if (contact == null) {
                contact = new Contact();
                contact.setId(id);
```

```
            contact.setFirstName(rs.getString("first_name"));
            contact.setLastName(rs.getString("last_name"));
            contact.setBirthDate(rs.getDate("birth_date"));
            contact.setContactTelDetails(new ArrayList<ContactTelDetail>());

              map.put(id, contact);
            }

            Long contactTelDetailId = rs.getLong("contact_tel_id");

            if (contactTelDetailId > 0) {
               ContactTelDetail contactTelDetail = new ContactTelDetail();
               contactTelDetail.setId(contactTelDetailId);
               contactTelDetail.setContactId(id);
               contactTelDetail.setTelType(rs.getString("tel_type"));
               contactTelDetail.setTelNumber(rs.getString("tel_number"));
               contact.getContactTelDetails().add(contactTelDetail);
            }
         }

         return new ArrayList<Contact> (map.values());
      });
   }
```

В листинге 6.49 приведен модифицированный код класса `AnnotationJdbcDaoSample`, предназначенного для тестирования пакетной операции вставки.

**Листинг 6.49. Тестирование метода `insertWithDetail()`**

```
package com.apress.prospring4.ch6;

import java.util.List;
import java.util.ArrayList;
import java.util.GregorianCalendar;
import java.sql.Date;
import org.springframework.context.support.GenericXmlApplicationContext;

public class AnnotationJdbcDaoSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();

      ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

      Contact contact = new Contact();
      contact.setFirstName("Michael");
      contact.setLastName("Jackson");
      contact.setBirthDate(new Date((new GregorianCalendar(1964, 10, 1))
            .getTime().getTime()));

      List<ContactTelDetail> contactTelDetails = new ArrayList<ContactTelDetail>();

      ContactTelDetail contactTelDetail = new ContactTelDetail();
      contactTelDetail.setTelType("Home");
      contactTelDetail.setTelNumber("11111111");

      contactTelDetails.add(contactTelDetail);

      contactTelDetail = new ContactTelDetail();
```

```
        contactTelDetail.setTelType("Mobile");
        contactTelDetail.setTelNumber("22222222");

        contactTelDetails.add(contactTelDetail);
        contact.setContactTelDetails(contactTelDetails);
        contactDao.insertWithDetail(contact);

        listContacts(contactDao.findAllWithDetail());
    }
    private static void listContacts(List<Contact> contacts) {
        for (Contact contact: contacts) {
            System.out.println(contact);

            if (contact.getContactTelDetails() != null) {
              for (ContactTelDetail contactTelDetail:
                      contact.getContactTelDetails()) {
                  System.out.println("---" + contactTelDetail);
              }
            }

            System.out.println();
        }
    }
}
```

Запуск этой программы даст следующий вывод из последнего метода `listContacts()`:

```
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
---Contact Tel Detail - Id: 2, Contact id: 1, Type: Home, Number: 1234567890
---Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
---Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Contact - Id: 4, First name: Michael, Last name: Jackson, Birthday: 1964-11-01
---Contact Tel Detail - Id: 4, Contact id: 4, Type: Home, Number: 11111111
---Contact Tel Detail - Id: 5, Contact id: 4, Type: Mobile, Number: 22222222
```

Как видите, все новые контакты со сведениями о телефонах были вставлены в базу данных.

## Вызов хранимых функций с использованием `SqlFunction`

Платформа Spring также предлагает классы, которые упрощают выполнение хранимых процедур/функций с применением JDBC. В этом разделе мы посмотрим, как запустить простую функцию с использованием класса `SqlFunction`. Мы будем работать с базой данных MySQL, создадим хранимую функцию и вызовем ее с помощью класса `SqlFunction<T>`.

Мы предполагаем, что имеется база данных MySQL со схемой `prospring4_ch6`, а также именем пользователя и паролем, установленными в `prospring4` (та же база данных, которая применялась в примере из раздела "Исследование инфраструктуры JDBC"). Давайте создадим хранимую функцию по имени `getFirstNameById()`, которая принимает идентификатор контакта и возвращает имя контакта.

В листинге 6.50 показан сценарий для создания этой хранимой функции в MySQL (`stored-function.sql`). Запустите этот сценарий в указанной базе данных MySQL.

**Листинг 6.50. Хранимая функция для базы данных MySQL**

```
DELIMITER //
CREATE FUNCTION getFirstNameById(in_id INT)
   RETURNS VARCHAR(60)
BEGIN
   RETURN (SELECT first_name FROM contact WHERE id = in_id);
END //
DELIMITER ;
```

Хранимая функция просто принимает идентификатор и возвращает имя из записи контакта с этим идентификатором.

Далее мы создадим класс `StoredFunctionFirstNameById`, предназначенный для представления операции вызова хранимой функции, который расширяет класс `SqlFunction<T>`. Код этого класса приведен в листинге 6.51.

**Листинг 6.51. Класс `StoredFunctionFirstNameById`**

```
package com.apress.prospring4.ch6;

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlFunction;

public class StoredFunctionFirstNameById extends SqlFunction<String> {
   private static final String SQL = "select getfirstnamebyid(?)";

   public StoredFunctionFirstNameById (DataSource dataSource) {
      super(dataSource, SQL);
      declareParameter(new SqlParameter(Types.INTEGER));
      compile();
   }
}
```

Класс расширяет `SqlFunction<T>` и передает тип `String`, который указывает возвращаемый тип функции. Затем мы объявляем строку `SQL` для вызова хранимой функции в базе данных MySQL. После этого в конструкторе объявляется параметр и формируется операция. Теперь класс готов для использования в классе реализации. В листинге 6.52 показан класс `JdbcContactDao`, в котором применяется хранимая функция.

**Листинг 6.52. Класс `JdbcContactDao`, использующий хранимую функцию**

```
package com.apress.prospring4.ch6;

import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;
```

```
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.annotation.Resource;
import javax.sql.DataSource;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Repository;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.dao.DataAccessException;

@Repository("contactDao")
public class JdbcContactDao implements ContactDao {
   private static final Log LOG = LogFactory.getLog(JdbcContactDao.class);

   private DataSource dataSource;
   private SelectAllContacts selectAllContacts;
   private SelectContactByFirstName selectContactByFirstName;
   private UpdateContact updateContact;
   private InsertContact insertContact;
   private InsertContactTelDetail insertContactTelDetail;
   private StoredFunctionFirstNameById storedFunctionFirstNameById;

   @Override
   public List<Contact> findAll() {
      return selectAllContacts.execute();
   }

   @Override
   public List<Contact> findByFirstName(String firstName) {
      Map<String, Object> paramMap = new HashMap<String, Object>();
      paramMap.put("first_name", firstName);

      return selectContactByFirstName.executeByNamedParam(paramMap);
   }

   @Override
   public String findFirstNameById(Long id) {
      List<String> result = storedFunctionFirstNameById.execute(id);
      return result.get(0);
   }

   @Override
   public void insert(Contact contact) {
      Map<String, Object> paramMap = new HashMap<String, Object>();
      paramMap.put("first_name", contact.getFirstName());
      paramMap.put("last_name", contact.getLastName());
      paramMap.put("birth_date", contact.getBirthDate());

      KeyHolder keyHolder = new GeneratedKeyHolder();

      insertContact.updateByNamedParam(paramMap, keyHolder);

      contact.setId(keyHolder.getKey().longValue());

      LOG.info("New contact inserted with id: " + contact.getId());
   }
```

```
    @Override
    public void insertWithDetail(Contact contact) {
        insertContactTelDetail = new InsertContactTelDetail(dataSource);

        Map<String, Object> paramMap = new HashMap<String, Object>();
        paramMap.put("first_name", contact.getFirstName());
        paramMap.put("last_name", contact.getLastName());
        paramMap.put("birth_date", contact.getBirthDate());

        KeyHolder keyHolder = new GeneratedKeyHolder();

        insertContact.updateByNamedParam(paramMap, keyHolder);

        contact.setId(keyHolder.getKey().longValue());

        LOG.info("New contact inserted with id: " + contact.getId());

        List<ContactTelDetail> contactTelDetails =
            contact.getContactTelDetails();

        if (contactTelDetails != null) {
            for (ContactTelDetail contactTelDetail: contactTelDetails) {
                paramMap = new HashMap<String, Object>();
                paramMap.put("contact_id", contact.getId());
                paramMap.put("tel_type", contactTelDetail.getTelType());
                paramMap.put("tel_number", contactTelDetail.getTelNumber());
                insertContactTelDetail.updateByNamedParam(paramMap);
            }
        }

        insertContactTelDetail.flush();
    }

    @Override
    public List<Contact> findAllWithDetail() {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(getDataSource());

        String sql = "select c.id, c.first_name, c.last_name, c.birth_date" +
", t.id as contact_tel_id, t.tel_type, t.tel_number from contact c " +
"left join contact_tel_detail t on c.id = t.contact_id";

        return jdbcTemplate.query(sql, new ContactWithDetailExtractor());
    }

    @Override
    public void update(Contact contact) {
        Map<String, Object> paramMap = new HashMap<String, Object>();
        paramMap.put("first_name", contact.getFirstName());
        paramMap.put("last_name", contact.getLastName());
        paramMap.put("birth_date", contact.getBirthDate());
        paramMap.put("id", contact.getId());

        updateContact.updateByNamedParam(paramMap);

        LOG.info("Existing contact updated with id: " + contact.getId());
    }

    @Resource(name="dataSource")
    public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
      this.selectAllContacts = new SelectAllContacts(dataSource);
      this.selectContactByFirstName = new SelectContactByFirstName(dataSource);
```

```
    this.updateContact = new UpdateContact(dataSource);
    this.insertContact = new InsertContact(dataSource);
    this.storedFunctionFirstNameById = new StoredFunctionFirstNameById(dataSource);
  }
  public DataSource getDataSource() {
     return dataSource;
  }
  private static final class ContactWithDetailExtractor
     implements ResultSetExtractor<List<Contact>> {
     @Override
     public List<Contact> extractData(ResultSet rs) throws
        SQLException, DataAccessException {
        Map<Long, Contact> map = new HashMap<Long, Contact>();
        Contact contact = null;
        while (rs.next()) {
           Long id = rs.getLong("id");
           contact = map.get(id);

           if (contact == null) {
             contact = new Contact();
             contact.setId(id);
             contact.setFirstName(rs.getString("first_name"));
             contact.setLastName(rs.getString("last_name"));
             contact.setBirthDate(rs.getDate("birth_date"));
             contact.setContactTelDetails(new ArrayList<ContactTelDetail>());

             map.put(id, contact);
           }
           Long contactTelDetailId = rs.getLong("contact_tel_id");
           if (contactTelDetailId > 0) {
              ContactTelDetail contactTelDetail = new ContactTelDetail();
              contactTelDetail.setId(contactTelDetailId);
              contactTelDetail.setContactId(id);
              contactTelDetail.setTelType(rs.getString("tel_type"));
              contactTelDetail.setTelNumber(rs.getString("tel_number"));
              contact.getContactTelDetails().add(contactTelDetail);
           }
        }
        return new ArrayList<Contact> (map.values());
     }
  }
}
```

При внедрении источника данных создается экземпляр `StoredFunctionFirstNameById`. После этого внутри метода `findFirstNameById()` вызывается метод `execute()`, которому передается идентификатор контакта. Этот вызов возвратит список `String`, из которого нужен лишь первый элемент, поскольку результирующий набор должен содержать только одну запись.

### Рефакторинг с использованием лямбда-выражения

При работе с Java 8 вместо создания класса `ContactWithDetailExtractor`, как было показано ранее, можно применить лямбда-выражение:

```
@Override
public List<Contact> findAllWithDetail() {
    JdbcTemplate jdbcTemplate = new JdbcTemplate(getDataSource());

    String sql = "select c.id, c.first_name, c.last_name, c.birth_date" +
", t.id as contact_tel_id, t.tel_type, t.tel_number from contact c " +
"left join contact_tel_detail t on c.id = t.contact_id";

    return jdbcTemplate.query(sql, (ResultSet rs) -> {
       Map<Long, Contact> map = new HashMap<Long, Contact>();
       Contact contact = null;

       while (rs.next()) {
          Long id = rs.getLong("id");
          contact = map.get(id);

          if (contact == null) {
             contact = new Contact();
             contact.setId(id);
             contact.setFirstName(rs.getString("first_name"));
             contact.setLastName(rs.getString("last_name"));
             contact.setBirthDate(rs.getDate("birth_date"));
             contact.setContactTelDetails(new ArrayList<ContactTelDetail>());

             map.put(id, contact);
          }

          Long contactTelDetailId = rs.getLong("contact_tel_id");

          if (contactTelDetailId > 0) {
             ContactTelDetail contactTelDetail = new ContactTelDetail();
             contactTelDetail.setId(contactTelDetailId);
             contactTelDetail.setContactId(id);
             contactTelDetail.setTelType(rs.getString("tel_type"));
             contactTelDetail.setTelNumber(rs.getString("tel_number"));
             contact.getContactTelDetails().add(contactTelDetail);
          }
       }

       return new ArrayList<Contact> (map.values());
    });
}
```

В листинге 6.53 приведено измененное содержимое файла конфигурации Spring для подключения к MySQL (`app-context-annotation.xml`).

**Листинг 6.53. Конфигурация Spring для MySQL**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <import resource="classpath:META-INF/spring/datasource-dbcp.xml"/>

    <context:component-scan base-package="com.apress.prospring4.ch6"/>
</beans>
```

Здесь импортируется файл `datasource-dbcp.xml`, который содержит конфигурацию источника данных для базы данных MySQL. Для запуска программы к проекту должна быть добавлена зависимость `commons-dbcp`, как показано в табл. 6.5.

**Таблица 6.5. Зависимость для `commons-dbcp`**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `commons-dbcp` | `commons-dbcp` | 1.4 | Библиотека поддержки пула подключений к базе данных `commons-dbcp` из Apache |

В листинге 6.54 показана модифицированная тестовая программа.

**Листинг 6.54. Тестирование хранимой функции в базе данных MySQL**

```
package com.apress.prospring4.ch6;

import org.springframework.context.support.GenericXmlApplicationContext;

public class AnnotationJdbcDaoSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();

        ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

        System.out.println(contactDao.findFirstNameById(1l));
    }
}
```

В тестовой программе мы передаем хранимой функции идентификатор, равный 1. В результате возвратится Chris в качестве имени, если ранее для базы данных MySQL запускался сценарий `test-data.sql`. Выполнение программы дает следующий вывод:

```
JdbcTemplate: 663 - Executing prepared SQL query
JdbcTemplate: 597 - Executing prepared SQL statement [select getfirstnamebyid(?)]
DataSourceUtils: 110 - Fetching JDBC Connection from DataSource
DataSourceUtils: 327 - Returning JDBC Connection to DataSource
Chris
```

В выводе видно, что имя было извлечено корректно. Здесь был представлен лишь простой пример для демонстрации возможностей функций модуля Spring JDBC. Платформа Spring также предлагает и другие классы (например, `StoredProcedure`), предназначенные для вызова сложных хранимых процедур, которые возвращают составные типы данных. Если вас интересует доступ к хранимым процедурам с использованием JDBC, рекомендуем обратиться к справочному руководству по Spring.

# Проект Spring Data: расширения JDBC

В последние годы технологии баз данных развивались настолько быстрыми темпами, порождая огромное количество специализированных баз данных, что в наши дни СУРБД не является единственным выбором при разработке приложений. В ответ на эту эволюцию технологий баз данных, а также с целью удовлетворения потребностей сообщества разработчиков, был организован проект Spring Data (`www.springsource.org/spring-data`). Главная цель этого проекта — предоставление удобных расширений к ключевой функциональности доступа к данным Spring для взаимодействия с базами данных, отличающимися от реляционных.

Проект Spring Data предлагает разнообразные расширения. Мы упомянем здесь одно из них — JDBC Extensions (`www.springsource.org/spring-data/jdbc-extensions`). Как вы уже наверняка догадались, это расширение предоставляет ряд усовершенствованных средств для упрощения разработки приложений JDBC с использованием Spring.

Ниже перечислены основные функциональные возможности JDBC Extensions.

- **Поддержка QueryDSL.** Инструмент QueryDSL (`www.querydsl.com`) — это специфичный для предметной области язык, который предоставляет инфраструктуру для разработки запросов, безопасных к типам. Расширение JDBC Extensions в проекте Spring Data предлагает класс `QueryDslJdbcTemplate`, упрощающий написание приложений JDBC за счет применения QueryDSL вместо SQL-операторов.
- **Расширенная поддержка Oracle Database.** Расширение JDBC Extensions делает доступными многочисленные развитые средства для пользователей Oracle Database. С точки зрения подключений к базе данных оно поддерживает параметры сеансов, специфичные для Oracle, а также технологию быстрого обхода отказа подключения (Fast Connection Failover) при работе с Oracle RAC. Кроме того, предлагаются классы, которые интегрируются с расширенной поддержкой очередей Oracle (Oracle Advanced Queueing). С точки зрения типов данных обеспечивается собственная поддержка XML-типов Oracle, `STRUCT` и `ARRAY` и т.д.

Если вы занимаетесь разработкой приложений JDBC, используя Spring с Oracle Database, то определенно стоит взглянуть на JDBC Extensions.

# Соображения по поводу использования JDBC

Благодаря этому богатому набору функциональных средств, платформа Spring позволяет существенно упростить разработку, когда для взаимодействия с лежащей в основе СУРБД применяется JDBC. Однако по-прежнему приходится писать довольно много кода, особенно при трансформации результирующих наборов в соответствующие объекты предметной области.

На основе JDBC создано множество библиотек с открытым кодом, призванных ликвидировать разрыв между структурой реляционных данных и объектно-ориентированной моделью языка Java. Например, iBATIS является популярной инфраструктурой отображения данных, которая также основана на SQL-отображении. Инфраструктура iBatis дает возможность отображать объекты с хранимыми процеду-

рами или запросами на XML-файл описателя. Подобно Spring, iBATIS поддерживает декларативный способ отображения запросов на объекты, значительно сокращая время на сопровождение SQL-запросов, которые могут быть разбросаны по разнообразным классам DAO.

Доступно также множество других инфраструктур объектно-реляционного отображения, которые ориентированы на объектную модель, а не на запрос. В число популярных инфраструктур такого рода входят Hibernate, EclipseLink (также известная под названием TopLink) и OpenJPA. Все они соответствуют спецификации JPA в JCP.

По прошествии лет инструменты и инфраструктуры объектно-реляционного отображения стали намного более зрелыми, поэтому большинство разработчиков будут пользоваться одним из таких инструментов, а не напрямую JDBC. Однако в ситуациях, когда для обеспечения максимальной производительности необходим полный контроль над запросом, отправляемым в базу данных (к примеру, когда применяется иерархический запрос в Oracle), Spring JDBC оказывается по-настоящему жизнеспособным вариантом. Вдобавок при использовании Spring появляется преимущество в том, что можно смешивать различные технологии доступа к данным. Например, можно применять Hibernate в качестве главного инструмента объектно-реляционного отображения, а JDBC — как дополнение для некоторой логики сложных запросов или пакетных операций; их допускается сочетать в рамках одиночной бизнес-операции и затем помещать в одну и ту же транзакцию базы данных. Платформа Spring помогает легко справляться с ситуациями подобного рода.

# Резюме

В этой главе было показано, как использовать Spring для упрощения программирования JDBC. Вы научились подключаться к базе данных и выполнять операции выборки, обновления, удаления и вставки, а также обращаться к хранимым функциям базы данных. Подробно обсуждался ключевой класс Spring JDBC по имени `JdbcTemplate`. Кроме того, были рассмотрены другие классы Spring, построенные на основе `JdbcTemplate`, которые помогают моделировать различные операции JDBC. Мы также продемонстрировали применение в подходящих ситуациях нового средства лямбда-выражений, доступного в версии Java 8. В следующих двух главах речь пойдет об использовании Spring совместно с популярными технологиями объектно-реляционного отображения для разработки логики доступа к данным.

# ГЛАВА 7

# Использование Hibernate в Spring

В предыдущей главе было показано, как использовать JDBC в Spring-приложениях. Тем не менее, хотя Spring обеспечивает существенное упрощение разработки для JDBC, по-прежнему приходится писать большой объем кода. В этой главе мы рассмотрим одну из библиотек объектно-реляционного отображения (object-relational mapping — ORM), которая называется Hibernate.

Если вы обладаете опытом разработки приложений с доступом к данным посредством сущностных бинов EJB (до версии EJB 3.0), то наверняка должны помнить этот довольно болезненный процесс. Утомительное конфигурирование отображения, установление границ транзакций и наличие в каждом бине большого объема стереотипного кода, предназначенного для управления его жизненным циклом, значительно снижало продуктивность разработки корпоративных Java-приложений.

Поскольку платформа Spring была создана для охвата разработки на основе POJO и поддержки декларативной конфигурации вместо трудной и неуклюжей настройки бинов EJB, в сообществе разработчиков начали понимать, что более простая, облегченная и основанная на POJO инфраструктура может снизить сложность написания логики доступа к данным. С тех пор появилось множество библиотек, которые все вместе называют *библиотеками ORM*. Главное предназначение библиотеки ORM — ликвидация разрыва между структурой реляционных данных в СУРБД и объектно-ориентированной моделью в Java, что позволит разработчикам сосредоточиться на программировании с применением объектной модели и в то же самое время легко выполнять действия, касающиеся постоянства.

Среди всех библиотек ORM, доступных в сообществе открытого кода, Hibernate является одной из наиболее успешных. Ее функциональные возможности, такие как подход на основе POJO, простота разработки и поддержка определений сложных отношений, завоевали сердца многих участников сообщества передовых разработчиков на Java.

Популярность Hibernate также повлияла на процесс JCP, в рамках которого была разработана спецификация Java-объектов данных (Java Data Object — JDO) в качестве одной из стандартных технологий ORM в Java EE. Начиная с EJB 3.0, сущностный бин EJB был даже заменен интерфейсом Java Persistence API (JPA), на многие концепции которого оказали влияние популярные библиотеки ORM, в том числе Hibernate, TopLink и JDO.

Отношения между Hibernate и JPA очень тесные. Гэвин Кинг, основатель Hibernate, представил JBoss, будучи одним из членов экспертной группы JCP по определению спецификации JPA. Начиная с версии 3.2, Hibernate предоставляет реализацию JPA. Это значит, что при разработке приложений с использованием Hibernate для поставщика службы постоянства можно выбирать между собственным API-интерфейсом Hibernate и JPA с Hibernate.

После обсуждения истории развития Hibernate в этой главе будет показано, как применять Spring с Hibernate при разработке логики доступа к данным. Hibernate является настолько обширной библиотекой ORM, что раскрытие каждого ее аспекта в одной главе попросту невозможно; исследованиям Hibernate посвящены отдельные книги. В этой главе рассматриваются базовые идеи и основные сценарии использования Hibernate в Spring. В частности, мы обсудим следующие темы.

- **Конфигурирование фабрики сеансов Hibernate.** Ключевая концепция Hibernate связана с интерфейсом `Session`, который управляется `SessionFactory`. Мы покажем, как конфигурировать фабрику сеансов Hibernate для работы в Spring-приложении.
- **Основные концепции объектно-реляционного отображения с использованием Hibernate.** Мы рассмотрим основные концепции, связанные с применением Hibernate для отображения POJO на структуру лежащей в основе реляционной базы данных. Также будут обсуждаться некоторые часто используемые отношения, включая "один ко многим" и "многие ко многим".
- **Операции над данными.** Мы представим примеры выполнения операций над данными (запроса, вставки, обновления, удаления) с применением Hibernate в среде Spring. При работе с библиотекой Hibernate мы в основном будем взаимодействовать с ее основным интерфейсом `Session`.

---

**На заметку!** При определении объектно-реляционных отображений Hibernate поддерживает два стиля конфигурирования. Один из них предусматривает помещение информации отображения в XML-файлы, а другой предполагает использование Java-аннотаций внутри сущностных классов (в мире ORM или JPA класс Java, который отображается на структуру лежащей в основе реляционной базы данных, называется *сущностным классом* (entity class)). В этой главе мы сосредоточим внимание на применении подхода с аннотациями для объектно-реляционного отображения. Для аннотирования отображений мы используем стандарты JPA (например, определенные в пакете `javax.persistence`), поскольку они взаимозаменяемы с собственными аннотациями Hibernate, что поможет при будущей миграции в среду JPA.

---

# Модель данных для кода примера

Модель данных, применяемая в этой главе, показана на рис. 7.1.

В модель данных были добавлены две новых таблицы, `HOBBY` и `CONTACT_HOBBY_DETAIL` (таблица соединения), которые моделируют отношения "многие ко многим" между таблицами `CONTACT` и `HOBBY`. Кроме того, в таблицы `CONTACT` и `CONTACT_TEL_DETAIL` был добавлен столбец `VERSION`, предназначенный для оптимистичной блокировки, которая подробно обсуждается позже в главе.

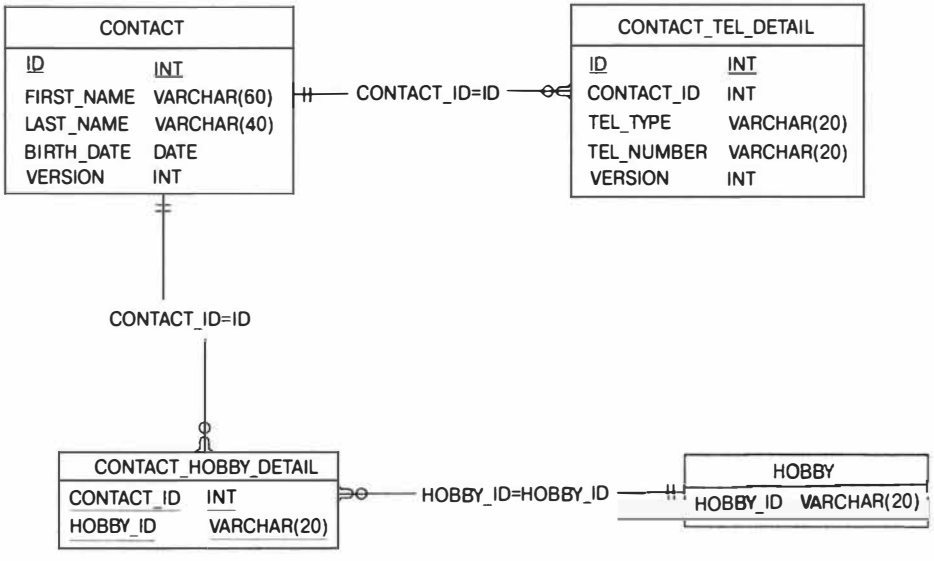


**Рис. 7.1.** Модель данных для демонстрации работы с Hibernate

В примерах этой главы мы используем встроенную базу данных H2, поэтому имя базы данных не требуется.

В листингах 7.1 и 7.2 показаны сценарии для создания схемы и наполнения тестовыми данными.

**Листинг 7.1. Сценарий для создания примера модели данных (`schema.sql`)**

```
CREATE TABLE CONTACT (
    ID INT NOT NULL AUTO_INCREMENT
    , FIRST_NAME VARCHAR(60) NOT NULL
    , LAST_NAME VARCHAR(40) NOT NULL
    , BIRTH_DATE DATE
    , VERSION INT NOT NULL DEFAULT 0
    , UNIQUE UQ_CONTACT_1 (FIRST_NAME, LAST_NAME)
    , PRIMARY KEY (ID)
);
CREATE TABLE HOBBY (
     HOBBY_ID VARCHAR(20) NOT NULL
     , PRIMARY KEY (HOBBY_ID)
);
CREATE TABLE CONTACT_TEL_DETAIL (
ID INT NOT NULL AUTO_INCREMENT
     , CONTACT_ID INT NOT NULL
     , TEL_TYPE VARCHAR(20) NOT NULL
     , TEL_NUMBER VARCHAR(20) NOT NULL
     , VERSION INT NOT NULL DEFAULT 0
     , UNIQUE UQ_CONTACT_TEL_DETAIL_1 (CONTACT_ID, TEL_TYPE)
     , PRIMARY KEY (ID)
     , CONSTRAINT FK_CONTACT_TEL_DETAIL_1 FOREIGN KEY (CONTACT_ID)
        REFERENCES CONTACT (ID)
);
```

```
CREATE TABLE CONTACT_HOBBY_DETAIL (
    CONTACT_ID INT NOT NULL
    , HOBBY_ID VARCHAR(20) NOT NULL
    , PRIMARY KEY (CONTACT_ID, HOBBY_ID)
    , CONSTRAINT FK_CONTACT_HOBBY_DETAIL_1 FOREIGN KEY (CONTACT_ID)
       REFERENCES CONTACT (ID) ON DELETE CASCADE
    , CONSTRAINT FK_CONTACT_HOBBY_DETAIL_2 FOREIGN KEY (HOBBY_ID)
    REFERENCES HOBBY (HOBBY_ID)
);
```

**Листинг 7.2. Сценарий для наполнения тестовыми данными (`test-data.sql`)**

```
insert into contact (first_name, last_name, birth_date)
       values ('Chris', 'Schaefer', '1981-05-03');
insert into contact (first_name, last_name, birth_date)
       values ('Scott', 'Tiger', '1990-11-02');
insert into contact (first_name, last_name, birth_date)
       values ('John', 'Smith', '1964-02-28');

insert into contact_tel_detail (contact_id, tel_type, tel_number)
       values (1, 'Mobile', '1234567890');
insert into contact_tel_detail (contact_id, tel_type, tel_number)
       values (1, 'Home', '1234567890');
insert into contact_tel_detail (contact_id, tel_type, tel_number)
       values (2, 'Home', '1234567890');

insert into hobby (hobby_id) values ('Swimming');
insert into hobby (hobby_id) values ('Jogging');
insert into hobby (hobby_id) values ('Programming');
insert into hobby (hobby_id) values ('Movies');
insert into hobby (hobby_id) values ('Reading');

insert into contact_hobby_detail(contact_id, hobby_id) values (1, 'Swimming');
insert into contact_hobby_detail(contact_id, hobby_id) values (1, 'Movies');
insert into contact_hobby_detail(contact_id, hobby_id) values (2, 'Swimming');
```

# Конфигурирование фабрики сеансов Hibernate

Как упоминалось ранее, ключевая концепция Hibernate основана на интерфейсе `Session`, который получается из `SessionFactory`. В Spring предлагаются классы для поддержки конфигурирования фабрики сеансов Hibernate в качестве бина Spring с желаемыми свойствами. Чтобы работать с Hibernate, необходимо добавить зависимость Hibernate, как показано в табл. 7.1.

**Таблица 7.1. Зависимость для Hibernate**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.hibernate` | `hibernate-entitymanager` | 4.2.3.Final | Библиотека Hibernate 4 |

В листинге 7.3 приведено содержимое соответствующего XML-файла конфигурации (app-context-annotation.xml).

**Листинг 7.3. Конфигурация Spring для фабрики сеансов Hibernate**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xmlns:p="http://www.springframework.org/schema/p"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:util="http://www.springframework.org/schema/util"
   xsi:schemaLocation="
     http://www.springframework.org/schema/jdbc
     http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/tx
     http://www.springframework.org/schema/tx/spring-tx.xsd
     http://www.springframework.org/schema/util
     http://www.springframework.org/schema/util/spring-util.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:META-INF/sql/schema.sql"/>
      <jdbc:script location="classpath:META-INF/sql/test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionManager"
     class="org.springframework.orm.hibernate4.HibernateTransactionManager"
       p:sessionFactory-ref="sessionFactory"/>

   <tx:annotation-driven/>

   <context:component-scan base-package="com.apress.prospring4.ch7"/>

   <bean id="sessionFactory"
     class="org.springframework.orm.hibernate4.LocalSessionFactoryBean"
        p:dataSource-ref="dataSource"
        p:packagesToScan="com.apress.prospring4.ch7"
        p:hibernateProperties-ref="hibernateProperties"/>

   <util:properties id="hibernateProperties">
      <prop key="hibernate.dialect">org.hibernate.dialect.H2Dialect</prop>
      <prop key="hibernate.max_fetch_depth">3</prop>
      <prop key="hibernate.jdbc.fetch_size">50</prop>
      <prop key="hibernate.jdbc.batch_size">10</prop>
      <prop key="hibernate.show_sql">true</prop>
   </util:properties>
</beans>
```

В предыдущей конфигурации было объявлено несколько бинов для обеспечения поддержки фабрики сеансов Hibernate.

Ниже перечислены основные элементы конфигурации.

- **Бин `dataSource`.** Мы объявляем источник данных со встроенной базой данных H2.
- **Бин `transactionManager`.** Фабрика сеансов Hibernate требует диспетчера транзакций для транзакционного доступа к данным. Платформа Spring предоставляет диспетчер транзакций, специфичный для Hibernate 4 (`org.springframework.orm.hibernate4.HibernateTransactionManager`). Бин объявлен с идентификатором `transactionManager`. По умолчанию всегда, когда требуется диспетчер транзакций, Spring будет искать внутри `ApplicationContext` бин с именем `transactionManager`. Транзакции подробно рассматриваются в главе 9. Вдобавок мы объявляем дескриптор `<tx:annotation-driven>` для поддержки требований к установлению границ транзакций, используя аннотации.
- **Дескриптор `<context:component-scan>`.** Этот дескриптор должен быть хорошо знаком. Мы указываем Spring на необходимость сканирования компонентов в пакете `com.apress.prospring4.ch7`.
- **Бин `sessionFactory`.** Этот бин является самой важной частью. Внутри него определено множество свойств. Во-первых, мы должны внедрить бин источника данных в фабрику сеансов. Во-вторых, мы инструктируем Hibernate относительно сканирования объектов предметной области в пакете `com.apress.prospring4.ch7`. В-третьих, свойство `hibernateProperties` предоставляет детали конфигурации для Hibernate. Существует множество конфигурационных параметров, но мы определяем только несколько важных свойств, которые должны предоставляться для каждого приложения. В табл. 7.2 перечислены главные конфигурационные параметры для фабрики сеансов Hibernate.

**Таблица 7.2. Главные конфигурационные параметры**

| Свойство | Описание |
|---|---|
| `hibernate.dialect` | Диалект базы данных для запросов, которые должны использоваться Hibernate. Библиотека Hibernate поддерживает диалекты SQL для многих баз данных. Эти диалекты являются подклассами `org.hibernate.dialect.Dialect`. В число основных диалектов входят `H2Dialect`, `Oracle10gDialect`, `PostgreSQLDialect`, `MySQLDialect`, `SQLServerDialect` и т.д. |
| `hibernate.max_fetch_depth` | Объявляет "глубину" для внешних соединений, когда отображаемые объекты имеют ассоциации с другими отображенными объектами. Этот параметр позволяет предотвратить выборку Hibernate слишком большого объема данных при наличии множества вложенных ассоциаций. Обычно применяется значение 3 |
| `hibernate.jdbc.fetch_size` | Количество записей из лежащего в основе результирующего набора JDBC, который библиотека Hibernate должна использовать для извлечения записей из базы данных в каждой выборке. Например, запрос был отправлен в базу данных и результирующий набор содержит 500 записей. Если размер выборки равен 50, то для получения всех данных Hibernate понадобится выполнить 10 выборок |

*Окончание табл. 7.2*

| Свойство | Описание |
|---|---|
| `hibernate.jdbc.batch_size` | Указывает Hibernate количество операций обновления, которые должны быть сгруппированы в пакет. Это очень удобно для выполнения пакетных операций в Hibernate. Очевидно, когда выполняется пакетное задание, обновляющее сотни записей, мы хотели бы, чтобы библиотека Hibernate сгруппировала запросы в пакеты, а не отправляла запросы обновления по одному за раз |
| `hibernate.show_sql` | Указывает, должна ли библиотека Hibernate выводить SQL-запросы в файл журнала или на консоль. Этот режим имеет смысл включать в среде разработки, потому что он помогает в тестировании и устранении ошибок |

За полным списком свойств, поддерживаемых Hibernate, обращайтесь в справочное руководство по конфигурации Hibernate (`http://docs.jboss.org/hibernate/core/4.3/manual/en-US/html/ch03.html`).

# Объектно-реляционное отображение с использованием аннотаций Hibernate

Имея показанную ранее конфигурацию, следующий шаг заключается в моделировании сущностных Java-классов POJO и их отображение на структуру лежащих в основе реляционных данных.

Отображение можно реализовать с помощью двух подходов. При первом подходе сначала проектируется объектная модель, а затем на ее основе генерируются сценарии для базы данных. Например, для конфигурации фабрики сеансов можно передать свойство `hibernate.hbm2ddl.auto`, чтобы заставить Hibernate автоматически экспортировать DDL-сценарий схемы в базу данных. Второй подход заключается в том, чтобы начать с модели данных, а затем построить объекты POJO для нужного отображения. Мы отдаем предпочтение второму подходу, поскольку он обеспечивает больший контроль над моделью данных, что очень полезно для оптимизации производительности доступа к данным. Предложенной ранее модели данных соответствует объектно-ориентированная модель, диаграмма классов которой показана на рис. 7.2.

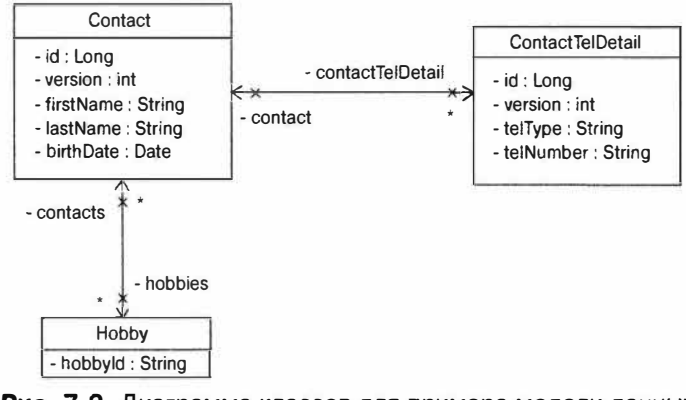


**Рис. 7.2.** Диаграмма классов для примера модели данных

Как видите, здесь имеется отношение "один ко многим" между объектами `Contact` и `ContactTelDetail`, а также отношение "многие ко многим" между объектами `Contact` и `Hobby`.

## Простое отображение

Давайте начнем с отображения простых атрибутов класса. В листинге 7.4 представлен код класса `Contact` с аннотациями отображения.

**Листинг 7.4. Класс `Contact`**

```
package com.apress.prospring4.ch7;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
import javax.persistence.Version;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name = "contact")
public class Contact implements Serializable {
    private Long id;
    private int version;
    private String firstName;
    private String lastName;
    private Date birthDate;

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Version
    @Column(name = "VERSION")
    public int getVersion() {
        return this.version;
    }

    public void setVersion(int version) {
        this.version = version;
    }
```

```
    @Column(name = "FIRST_NAME")
    public String getFirstName() {
        return this.firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    @Column(name = "LAST_NAME")
    public String getLastName() {
        return this.lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    @Temporal(TemporalType.DATE)
    @Column(name = "BIRTH_DATE")
    public Date getBirthDate() {
        return this.birthDate;
    }
    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }
    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate;
    }
}
```

Сначала мы аннотируем тип с помощью `@Entity`, указывая на то, что это отображенный сущностный класс. Аннотация `@Table` определяет имя таблицы в базе данных, на которую отображается эта сущность. Каждый отображенный атрибут снабжается аннотацией `@Column` с указанием имени столбца. В случае если имена типа и атрибутов совпадают с именами таблицы и столбцов, аннотации `@Table` и `@Column` можно опустить.

Относительно отображения необходимо отметить несколько моментов.

- **Атрибут даты рождения аннотирован посредством `@Temporal` со значением `TemporalType.DATE`.** Причина в том, что мы хотим отобразить тип даты Java (`java.util.Date`) на тип даты SQL (`java.sql.Date`). Это позволит обращаться к атрибуту `birthDate` в объекте `Contact`, используя тип `java.util.Date`, как обычно делается в приложении.
- **Атрибут `id` аннотирован с помощью `@Id`.** Это означает, что он является первичным ключом объекта. Инфраструктура Hibernate будет применять его как уникальный идентификатор при управлении экземплярами сущностей контактов в рамках сеанса. Кроме того, аннотация `@GeneratedValue` сообщает Hibernate, каким образом было сгенерировано значение `id`. Стратегия `IDENTITY` говорит о том, что идентификатор был сгенерирован СУРБД во время вставки.

- **Атрибут `version` аннотирован посредством `@Version`.** Это сообщает Hibernate о том, что мы хотим использовать механизм оптимистичной блокировки, применяя для управления атрибут `version`. Каждый раз, когда инфраструктура Hibernate обновляет запись, она сравнивает версию экземпляра сущности с версией записи в базе данных. Если версии совпадают, значит, данные ранее не обновлялись, поэтому Hibernate обновит данные и увеличит значение в столбце версии. Однако если версии отличаются, это значит, что кто-то обновил запись, и Hibernate сгенерирует исключение `StaleObjectStateException`, которое Spring будет транслировать в `HibernateOptimisticLockingFailure Exception`. В приведенном примере для управления версиями используется целочисленное значение. Помимо целого числа, Hibernate также поддерживает метку времени. Тем не менее, рекомендуется применять для управления версиями именно целочисленное значение, т.к. в этом случае Hibernate будет всегда увеличивать значение версии на 1 после каждого обновления. Когда же используется временная метка, после каждого обновления Hibernate будет заменять значение этой метки текущим показанием времени. Метки времени менее безопасны, поскольку две параллельных транзакции могут загрузить и обновить один и тот же элемент данных практически одновременно в пределах миллисекунды.

В листинге 7.5 показан еще один отображенный объект — `ContactTelDetail`.

**Листинг 7.5. Класс `ContactTelDetail`**

```
package com.apress.prospring4.ch7;
import static javax.persistence.GenerationType.IDENTITY;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
import javax.persistence.Version;
@Entity
@Table(name = "contact_tel_detail")
public class ContactTelDetail implements Serializable {
    private Long id;
    private int version;
    private String telType;
    private String telNumber;
    public ContactTelDetail() {
    }
    public ContactTelDetail(String telType, String telNumber) {
        this.telType = telType;
        this.telNumber = telNumber;
    }
    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return this.id;
    }
```

```
    public void setId(Long id) {
        this.id = id;
    }
    @Version
    @Column(name = "VERSION")
    public int getVersion() {
        return this.version;
    }
    public void setVersion(int version) {
        this.version = version;
    }
    @Column(name = "TEL_TYPE")
    public String getTelType() {
        return this.telType;
    }
    public void setTelType(String telType) {
        this.telType = telType;
    }
    @Column(name = "TEL_NUMBER")
    public String getTelNumber() {
        return this.telNumber;
    }
    public void setTelNumber(String telNumber) {
        this.telNumber = telNumber;
    }
}
```

В листинге 7.6 представлен код класса `Hobby`.

**Листинг 7.6. Класс `Hobby`**

```
package com.apress.prospring4.ch7;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Column;
import javax.persistence.Id;

@Entity
@Table(name = "hobby")
public class Hobby implements Serializable {
    private String hobbyId;
    @Id
    @Column(name = "HOBBY_ID")
    public String getHobbyId() {
        return this.hobbyId;
    }
    public void setHobbyId(String hobbyId) {
        this.hobbyId = hobbyId;
    }
    public String toString() {
        return "Hobby :" + getHobbyId();
    }
}
```

## Отображение "один ко многим"

Инфраструктура Hibernate обладает возможностью моделировать много видов ассоциаций. Наиболее распространенными ассоциациями являются "один ко многим" и "многие ко многим". Каждый контакт будет иметь ноль или больше телефонных номеров, так что здесь получается ассоциация "один ко многим" (в терминах ORM ассоциация "один ко многим" используется для моделирования отношений "ноль ко многим" и "один ко многим" в структуре данных). В листинге 7.7 показан модернизированный код класса `Contact` для ассоциации с классом `ContactTelDetail`.

**Листинг 7.7. Ассоциация "один ко многим"**

```
package com.apress.prospring4.ch7;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
import javax.persistence.Version;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.CascadeType;
import javax.persistence.OneToMany;

@Entity
@Table(name = "contact")
public class Contact implements Serializable {
    private Long id;
    private int version;
    private String firstName;
    private String lastName;
    private Date birthDate;
    private Set<ContactTelDetail> contactTelDetails = new
HashSet<ContactTelDetail>();

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Version
    @Column(name = "VERSION")
```

```
    public int getVersion() {
        return this.version;
    }
    public void setVersion(int version) {
        this.version = version;
    }
    @Column(name = "FIRST_NAME")
    public String getFirstName() {
        return this.firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    @Column(name = "LAST_NAME")
    public String getLastName() {
        return this.lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    @Temporal(TemporalType.DATE)
    @Column(name = "BIRTH_DATE")
    public Date getBirthDate() {
        return this.birthDate;
    }
    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }
    @OneToMany(mappedBy = "contact", cascade=CascadeType.ALL,
        orphanRemoval=true)
    public Set<ContactTelDetail> getContactTelDetails() {
        return this.contactTelDetails;
    }
    public void setContactTelDetails(Set<ContactTelDetail> contactTelDetails)
    {
     this.contactTelDetails = contactTelDetails;
    }
    public void addContactTelDetail(ContactTelDetail contactTelDetail) {
        contactTelDetail.setContact(this);
        getContactTelDetails().add(contactTelDetail);
    }
    public void removeContactTelDetail(ContactTelDetail contactTelDetail) {
        getContactTelDetails().remove(contactTelDetail);
    }
    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate;
    }
}
```

Метод извлечения атрибута `contactTelDetails` снабжен аннотацией `@OneToMany`, которая указывает на наличие отношения "один ко многим" с классом `ContactTelDetail`. Этой аннотации передается несколько атрибутов. Атрибут `mappedBy` задает свойство в классе `ContactTelDetail`, которое обеспечивает ассоциацию (т.е. связывает с определением внешнего ключа в таблице `CONTACT_TEL_DETAIL`). Атрибут `cascade` означает, что операция обновления должна распространяться на дочерние записи. Атрибут `orphanRemoval` указывает, что после того, как детали телефонных номеров контакта обновлены, записи, которые больше не существуют в наборе, должны быть удалены из базы данных.

В листинге 7.8 приведен измененный код класса `ContactTelDetail` с целью отображения ассоциации.

**Листинг 7.8. Отображение "один ко многим" в `ContactTelDetail`**

```
package com.apress.prospring4.ch7;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
import javax.persistence.Version;
import javax.persistence.ManyToOne;
import javax.persistence.JoinColumn;

@Entity
@Table(name = "contact_tel_detail")
public class ContactTelDetail implements Serializable {
    private Long id;
    private int version;
    private String telType;
    private String telNumber;
    private Contact contact;

    public ContactTelDetail() {
    }

    public ContactTelDetail(String telType, String telNumber) {
        this.telType = telType;
        this.telNumber = telNumber;
    }

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }
```

```
    @Version
    @Column(name = "VERSION")
    public int getVersion() {
        return this.version;
    }
    public void setVersion(int version) {
        this.version = version;
    }
    @Column(name = "TEL_TYPE")
    public String getTelType() {
        return this.telType;
    }
    public void setTelType(String telType) {
        this.telType = telType;
    }
    @Column(name = "TEL_NUMBER")
    public String getTelNumber() {
        return this.telNumber;
    }
    public void setTelNumber(String telNumber) {
        this.telNumber = telNumber;
    }
    @ManyToOne
    @JoinColumn(name = "CONTACT_ID")
    public Contact getContact() {
        return this.contact;
    }
    public void setContact(Contact contact) {
        this.contact = contact;
    }
    public String toString() {
    return "Contact Tel Detail - Id: " + id + ", Contact id: "
           + getContact().getId() + ", Type: "
           + telType + ", Number: " + telNumber;
    }
}
```

Мы снабдили метод извлечения атрибута `contact` аннотацией `@ManyToOne`, которая задает другую сторону ассоциации с `Contact`. Мы также указали аннотацию `@JoinColumn` с именем столбца внешнего ключа. Наконец, мы переопределили метод `toString()` для упрощения тестирования кода примера в будущем.

## Отображение "многие ко многим"

Каждый контакт имеет ноль или большее количество хобби, и каждое хобби связано с нулем или большим числом контактов, т.е. получается отображение "многие ко многим". Такое отображение требует таблицу соединения, которой является таблица `CONTACT_HOBBY_DETAIL`, показанная на рис. 7.2. В листинге 7.9 представлен модифицированный код класса `Contact`, моделирующий это отношение.

**Листинг 7.9. Ассоциация "многие ко многим"**

```
package com.apress.prospring4.ch7;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
import javax.persistence.Version;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.OneToMany;
import javax.persistence.ManyToMany;
import javax.persistence.JoinTable;
import javax.persistence.JoinColumn;
import javax.persistence.CascadeType;

@Entity
@Table(name = "contact")
public class Contact implements Serializable {
    private Long id;
    private int version;
    private String firstName;
    private String lastName;
    private Date birthDate;
    private Set<ContactTelDetail> contactTelDetails = new
HashSet<ContactTelDetail>();
    private Set<Hobby> hobbies = new HashSet<Hobby>();

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Version
    @Column(name = "VERSION")
    public int getVersion() {
        return this.version;
    }

    public void setVersion(int version) {
        this.version = version;
    }

    @Column(name = "FIRST_NAME")
```

```
    public String getFirstName() {
        return this.firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    @Column(name = "LAST_NAME")
    public String getLastName() {
        return this.lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    @Temporal(TemporalType.DATE)
    @Column(name = "BIRTH_DATE")
    public Date getBirthDate() {
        return this.birthDate;
    }
    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }
    @OneToMany(mappedBy = "contact", cascade=CascadeType.ALL,
        orphanRemoval=true)
    public Set<ContactTelDetail> getContactTelDetails() {
        return this.contactTelDetails;
    }
    public void setContactTelDetails(Set<ContactTelDetail> contactTelDetails) {
     this.contactTelDetails = contactTelDetails;
    }
    public void addContactTelDetail(ContactTelDetail contactTelDetail) {
        contactTelDetail.setContact(this);
        getContactTelDetails().add(contactTelDetail);
    }
    public void removeContactTelDetail(ContactTelDetail contactTelDetail) {
        getContactTelDetails().remove(contactTelDetail);
    }
    @ManyToMany
    @JoinTable(name = "contact_hobby_detail",
        joinColumns = @JoinColumn(name = "CONTACT_ID"),
        inverseJoinColumns = @JoinColumn(name = "HOBBY_ID"))
    public Set<Hobby> getHobbies() {
        return this.hobbies;
    }
    public void setHobbies(Set<Hobby> hobbies) {
        this.hobbies = hobbies;
    }
    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate;
    }
}
```

Метод извлечения атрибута hobbies в классе Contact аннотирован посредством @ManyToMany. Мы также предоставляем аннотацию @JoinTable для указания таблицы соединения, которую должна просматривать инфраструктура Hibernate. В атрибуте name задается имя таблицы соединения, в joinColumns определяется столбец, являющийся внешним ключом в таблице CONTACT, а в inverseJoinColumns указывается столбец, который представляет внешний ключ на другой стороне ассоциации (т.е. в таблице HOBBY). В листинге 7.10 показан измененный код класса Hobby.

**Листинг 7.10. Отображение "многие ко многим" в классе Hobby**

```
package com.apress.prospring4.ch7;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Column;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.JoinTable;
import javax.persistence.JoinColumn;
import java.util.Set;
import java.util.HashSet;

@Entity
@Table(name = "hobby")
public class Hobby implements Serializable {
    private String hobbyId;
    private Set<Contact> contacts = new HashSet<Contact>();

    @Id
    @Column(name = "HOBBY_ID")
    public String getHobbyId() {
        return this.hobbyId;
    }

    public void setHobbyId(String hobbyId) {
        this.hobbyId = hobbyId;
    }

    @ManyToMany
    @JoinTable(name = "contact_hobby_detail",
        joinColumns = @JoinColumn(name = "HOBBY_ID"),
        inverseJoinColumns = @JoinColumn(name = "CONTACT_ID"))
    public Set<Contact> getContacts() {
        return this.contacts;
    }

    public void setContacts(Set<Contact> contacts) {
        this.contacts = contacts;
    }

    @Override
    public String toString() {
        return "Hobby :" + getHobbyId();
    }
}
```

Это отображение напоминает приведенное в листинге 7.9, но здесь атрибуты joinColumns и inverseJoinColumns обращены, отражая данную ассоциацию.

# Интерфейс Session в Hibernate

В Hibernate при взаимодействии с базой данных приходится иметь дело с интерфейсом Session, который получается из фабрики сеансов.

В листинге 7.11 показан код класса ContactDaoImpl, который используется в примерах этой главы и имеет внедренную сконфигурированную реализацию SessionFactory.

**Листинг 7.11. Внедрение SessionFactory**

```
package com.apress.prospring4.ch7;

import org.springframework.transaction.annotation.Transactional;
import org.springframework.stereotype.Repository;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.SessionFactory;
import javax.annotation.Resource;

@Transactional
@Repository("contactDao")
public class ContactDaoImpl implements ContactDao {
    private static final Log LOG = LogFactory.getLog(ContactDaoImpl.class);

    private SessionFactory sessionFactory;

    public SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    ...

    @Resource(name="sessionFactory")
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
}
```

Как обычно, мы объявляем класс DAO как бин Spring с применением аннотации @Repository. Аннотация @Transactional определяет требования к транзакциям, которые будут обсуждаться в главе 9. Атрибут sessionFactory внедрен с использованием аннотации @Resource.

# Выполнение операций базы данных с помощью Hibernate

В листинге 7.12 показан интерфейс ContactDao, который определяет контракт для служб доступа к данным контактов, которые мы собираемся предоставлять.

**Листинг 7.12. Интерфейс `ContactDao`**

```
package com.apress.prospring4.ch7;

import java.util.List;

public interface ContactDao {
    List<Contact> findAll();
    List<Contact> findAllWithDetail();
    Contact findById(Long id);
    Contact save(Contact contact);
    void delete(Contact contact);
}
```

Интерфейс очень прост; он имеет три метода поиска, один метод сохранения и один метод удаления. Метод `save()` будет выполнять операции вставки и обновления.

## Запрашивание данных с использованием языка запросов Hibernate

Инфраструктура Hibernate, как и другие инструменты ORM вроде JDO и JPA, спроектирована на основе объектной модели. Это значит, что после определения всех отображений конструировать SQL-операторы для взаимодействия с базой данных не придется. Вместо этого для определения запросов в Hibernate используется язык запросов Hibernate (Hibernate Query Language — HQL). Во время взаимодействия с базой данных Hibernate транслирует HQL-запросы в SQL-операторы.

Синтаксис HQL очень похож на синтаксис SQL. Тем не менее, при этом нужно мыслить категориями объектов, а не базы данных. В последующих разделах будет приведено несколько примеров.

### *Простой запрос с отложенной выборкой*

Давайте начнем с реализации метода `findAll()`, который просто извлекает из базы данных все контакты. В листинге 7.13 показан модифицированный код с этой функциональностью.

**Листинг 7.13. Реализация метода `findAll()`**

```
package com.apress.prospring4.ch7;

import org.springframework.transaction.annotation.Transactional;
import org.springframework.stereotype.Repository;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.SessionFactory;
import javax.annotation.Resource;
import java.util.List;

@Transactional
@Repository("contactDao")
public class ContactDaoImpl implements ContactDao {
    private static final Log LOG = LogFactory.getLog(ContactDaoImpl.class);

    private SessionFactory sessionFactory;

    @Override
```

```
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return sessionFactory.getCurrentSession()
            .createQuery("from Contact c").list();
    }
    @Override
    public List<Contact> findAllWithDetail() {
        return null;
    }
    @Override
    public Contact findById(Long id) {
        return null;
    }
    @Override
    public Contact save(Contact contact) {
        return null;
    }
    @Override
    public void delete(Contact contact) {
    }
    public SessionFactory getSessionFactory() {
        return sessionFactory;
    }
    @Resource(name="sessionFactory")
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
}
```

Метод `SessionFactory.getCurrentSession()` получает реализацию интерфейса `Session` из Hibernate. Затем вызывается метод `Session.createQuery()`, которому передается оператор HQL. Оператор `from Contact c` просто извлекает все контакты из базы данных. Альтернативный синтаксис этого оператора выглядит как `select c from Contact c`. Аннотация `@Transactional(readOnly=true)` означает, что транзакция должна быть установлена как предназначенная только для чтения. Установка этого атрибута для методов, производящих только чтение, обеспечит более высокую производительность.

В листинге 7.14 приведена простая тестовая программа для `ContactDaoImpl`.

**Листинг 7.14. Тестирование класса `ContactDaoImpl`**

```
package com.apress.prospring4.ch7;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringHibernateSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();
```

```
        ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);
        listContacts(contactDao.findAll());
    }
    private static void listContacts(List<Contact> contacts) {
        System.out.println("");
        System.out.println("Listing contacts without details:");
        for (Contact contact: contacts) {
            System.out.println(contact);
            System.out.println();
        }
    }
}
```

Запуск тестовой программы дает в результате следующий вывод:

```
Listing contacts without details:
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

Записи о контактах были извлечены, но что насчет деталей, связанных с телефонами и хобби? Давайте изменим тестовый класс с целью вывода детальной информации. В листинге 7.15 показан модифицированный код.

**Листинг 7.15. Тестирование класса `ContactDaoImpl`, выводящего детальную информацию**

```
package com.apress.prospring4.ch7;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringHibernateSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();

        ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

        listContactsWithDetail(contactDao.findAll());
    }

    private static void listContactsWithDetail(List<Contact> contacts) {
        System.out.println("");
        System.out.println("Listing contacts with details:");

        for (Contact contact: contacts) {
            System.out.println(contact);

            if (contact.getContactTelDetails() != null) {
                for (ContactTelDetail contactTelDetail:
                        contact.getContactTelDetails()) {
                    System.out.println(contactTelDetail);
                }
            }
```

```
            if (contact.getHobbies() != null) {
                for (Hobby hobby: contact.getHobbies()) {
                    System.out.println(hobby);
                }
            }
            System.out.println();
        }
    }
}
```

Повторный запуск программы приводит к исключению:

```
Listing contacts with details:
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
Exception in thread "main" org.hibernate.LazyInitializationException:
Исключение в потоке "main" org.hibernate.LazyInitializationException:
...
```

При попытке доступа к ассоциации инфраструктура Hibernate генерирует исключение `LazyInitializationException`. Причина в том, что по умолчанию Hibernate выбирает ассоциацию отложенным ("ленивым") образом, а это означает, что соединение для наполнения записями таблицы ассоциации (т.е. `CONTACT_TEL_DETAIL`) осуществляться не будет. Причина принятия такого подхода связана с производительностью, поскольку если для запроса, извлекающего тысячи записей, извлекать также все ассоциации, то большой объем передаваемых данных приведет к снижению производительности.

### Запрос с выборкой ассоциаций

Для того чтобы инфраструктура Hibernate извлекала данные из ассоциаций, можно воспользоваться одним из двух способов. При первом способе ассоциация определяется с режимом выборки `EAGER`, например: `@ManyToMany(fetch=FetchType.EAGER)`. Это сообщает Hibernate о необходимости выборки связанных данных в каждом запросе. Однако, как уже упоминалось, такой подход оказывает влияние на производительность извлечения данных.

Второй способ предполагает обеспечение выборки связанных данных в запросе только по требованию. Если используется запрос `Criteria`, можно вызвать метод `Criteria.setFetchMode()` и заставить Hibernate незамедлительно выполнить выборку ассоциации. В случае применения `NamedQuery` для инструктирования Hibernate о необходимости незамедлительной выборки ассоциации можно использовать операцию `fetch`.

Давайте взглянем на реализацию метода `findAllWithDetail()`, который будет извлекать все контакты вместе с телефонами и хобби. В этом примере мы воспользуемся подходом с именованным запросом (`NamedQuery`). Именованный запрос может быть вынесен в XML-файл или объявлен с применением аннотации в сущностном классе. В листинге 7.16 показан пересмотренный объект предметной области `Contact` с именованным запросом, который определен посредством аннотаций.

**Листинг 7.16. Использование `NamedQuery`**

```
package com.apress.prospring4.ch7;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
import javax.persistence.Version;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.OneToMany;
import javax.persistence.ManyToMany;
import javax.persistence.JoinTable;
import javax.persistence.JoinColumn;
import javax.persistence.CascadeType;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;

@Entity
@Table(name = "contact")
@NamedQueries({
@NamedQuery(name="Contact.findAllWithDetail",
query="select distinct c from Contact c left join fetch c.contactTelDetails t
left join fetch c.hobbies h")
})
public class Contact implements Serializable {
    private Long id;
    private int version;
    private String firstName;
    private String lastName;
    private Date birthDate;
    private Set<ContactTelDetail> contactTelDetails =
         new HashSet<ContactTelDetail>();
    private Set<Hobby> hobbies = new HashSet<Hobby>();

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Version
    @Column(name = "VERSION")
```

```
public int getVersion() {
    return this.version;
}
public void setVersion(int version) {
    this.version = version;
}
@Column(name = "FIRST_NAME")
public String getFirstName() {
    return this.firstName;
}
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
@Column(name = "LAST_NAME")
public String getLastName() {
    return this.lastName;
}
public void setLastName(String lastName) {
    this.lastName = lastName;
}
@Temporal(TemporalType.DATE)
@Column(name = "BIRTH_DATE")
public Date getBirthDate() {
    return this.birthDate;
}
public void setBirthDate(Date birthDate) {
    this.birthDate = birthDate;
}
@OneToMany(mappedBy = "contact", cascade=CascadeType.ALL,
    orphanRemoval=true)
public Set<ContactTelDetail> getContactTelDetails() {
    return this.contactTelDetails;
}
public void setContactTelDetails(Set<ContactTelDetail> contactTelDetails)
{
   this.contactTelDetails = contactTelDetails;
}
public void addContactTelDetail(ContactTelDetail contactTelDetail) {
    contactTelDetail.setContact(this);
    getContactTelDetails().add(contactTelDetail);
}
public void removeContactTelDetail(ContactTelDetail contactTelDetail) {
    getContactTelDetails().remove(contactTelDetail);
}
@ManyToMany
@JoinTable(name = "contact_hobby_detail",
    joinColumns = @JoinColumn(name = "CONTACT_ID"),
    inverseJoinColumns = @JoinColumn(name = "HOBBY_ID"))
```

```
    public Set<Hobby> getHobbies() {
        return this.hobbies;
    }

    public void setHobbies(Set<Hobby> hobbies) {
        this.hobbies = hobbies;
    }

    @Override
    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate;
    }
}
```

В коде сначала определяется именованный запрос `Contact.findAllWithDetail`. Затем определяется собственно запрос на языке HQL. Обратите внимание на конструкцию `left join fetch`, которая сообщает Hibernate о необходимости незамедлительно произвести выборку ассоциации. Также должна использоваться конструкция `select distinct`, иначе Hibernate возвратит дублированные объекты (например, если какой-то контакт имеет два телефона, будут возвращены два объекта `Contact`).

В листинге 7.17 приведена реализация метода `findAllWithDetail()`.

**Листинг 7.17. Реализация метода `findAllWithDetail()`**

```
package com.apress.prospring4.ch7;

import org.springframework.transaction.annotation.Transactional;
import org.springframework.stereotype.Repository;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.SessionFactory;
import javax.annotation.Resource;
import java.util.List;

@Transactional
@Repository("contactDao")
public class ContactDaoImpl implements ContactDao {
    private static final Log LOG = LogFactory.getLog(ContactDaoImpl.class);

    private SessionFactory sessionFactory;

    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return sessionFactory.getCurrentSession()
          .createQuery("from Contact c").list();
    }

    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAllWithDetail() {
        return sessionFactory.getCurrentSession().
            getNamedQuery("Contact.findAllWithDetail").list();
    }
```

```
    @Override
    public Contact findById(Long id) {
        return null;
    }
    @Override
    public Contact save(Contact contact) {
        return null;
    }
    @Override
    public void delete(Contact contact) {

    }
    public SessionFactory getSessionFactory() {
        return sessionFactory;
    }
    @Resource(name="sessionFactory")
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
}
```

На этот раз мы используем метод `Session.getNamedQuery()`, передавая ему имя `NamedQuery`. После модификации тестовой программы (`SpringHibernateSample`) с целью вызова `ContactDao.findAllWithDetail()` и ее запуска получается следующий вывод:

```
Listing contacts with details:
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
Contact Tel Detail - Id: 2, Contact id: 1, Type: Home, Number: 1234567890
Hobby :Movies
Hobby :Swimming

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming
```

Теперь все контакты со своими деталями извлекаются корректно. Давайте рассмотрим еще один пример с именованным запросом, принимающим параметры. На этот раз мы реализуем метод `findById()`, который также должен производить выборку ассоциации. В листинге 7.18 показан класс `Contact` с добавленным новым именованным запросом.

**Листинг 7.18. Использование именованного запроса с параметрами**

```
package com.apress.prospring4.ch7;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
```

```
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
import javax.persistence.Version;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.OneToMany;
import javax.persistence.ManyToMany;
import javax.persistence.JoinTable;
import javax.persistence.JoinColumn;
import javax.persistence.CascadeType;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;

@Entity
@Table(name = "contact")
@NamedQueries({
@NamedQuery(name="Contact.findById",
query="select distinct c from Contact c left join fetch c.contactTelDetails
t left join fetch c.hobbies h where c.id = :id"),
@NamedQuery(name="Contact.findAllWithDetail",
query="select distinct c from Contact c left join fetch c.contactTelDetails
t left join fetch c.hobbies h")
})
public class Contact implements Serializable {
    private Long id;
    private int version;
    private String firstName;
    private String lastName;
    private Date birthDate;
    private Set<ContactTelDetail> contactTelDetails =
        new HashSet<ContactTelDetail>();
    private Set<Hobby> hobbies = new HashSet<Hobby>();

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Version
    @Column(name = "VERSION")
    public int getVersion() {
        return this.version;
    }

    public void setVersion(int version) {
        this.version = version;
    }
```

```
    @Column(name = "FIRST_NAME")
    public String getFirstName() {
        return this.firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    @Column(name = "LAST_NAME")
    public String getLastName() {
        return this.lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    @Temporal(TemporalType.DATE)
    @Column(name = "BIRTH_DATE")
    public Date getBirthDate() {
        return this.birthDate;
    }
    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }
    @OneToMany(mappedBy = "contact", cascade=CascadeType.ALL,
        orphanRemoval=true)
    public Set<ContactTelDetail> getContactTelDetails() {
        return this.contactTelDetails;
    }
    public void setContactTelDetails(Set<ContactTelDetail> contactTelDetails)
    {
        this.contactTelDetails = contactTelDetails;
    }
    public void addContactTelDetail(ContactTelDetail contactTelDetail) {
        contactTelDetail.setContact(this);
        getContactTelDetails().add(contactTelDetail);
    }
    public void removeContactTelDetail(ContactTelDetail contactTelDetail) {
        getContactTelDetails().remove(contactTelDetail);
    }
    @ManyToMany
    @JoinTable(name = "contact_hobby_detail",
        joinColumns = @JoinColumn(name = "CONTACT_ID"),
        inverseJoinColumns = @JoinColumn(name = "HOBBY_ID"))
    public Set<Hobby> getHobbies() {
        return this.hobbies;
    }
    public void setHobbies(Set<Hobby> hobbies) {
        this.hobbies = hobbies;
    }
    @Override
    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate;
    }
}
```

В именованном запросе `Contact.findById` мы объявляем именованный параметр `:id`. В листинге 7.19 приведена реализация метода `findById()` в классе `ContactDaoImpl`.

**Листинг 7.19. Реализация метода `findById()`**

```
package com.apress.prospring4.ch7;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.stereotype.Repository;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.SessionFactory;
import javax.annotation.Resource;
import java.util.List;
@Transactional
@Repository("contactDao")
public class ContactDaoImpl implements ContactDao {
    private static final Log LOG = LogFactory.getLog(ContactDaoImpl.class);
    private SessionFactory sessionFactory;
    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return sessionFactory.getCurrentSession()
          .createQuery("from Contact c").list();
    }
    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAllWithDetail() {
        return sessionFactory.getCurrentSession().
            getNamedQuery("Contact.findAllWithDetail").list();
    }
    @Override
    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        return (Contact) sessionFactory.getCurrentSession().
            getNamedQuery("Contact.findById").
            setParameter("id", id).uniqueResult();
    }
    @Override
    public Contact save(Contact contact) {
        return null;
    }
    @Override
    public void delete(Contact contact) {

    }
    public SessionFactory getSessionFactory() {
        return sessionFactory;
    }
    @Resource(name="sessionFactory")
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
}
```

В этом листинге мы используем тот же самый метод `Session.getNameQuery()`. Кроме того, мы вызываем метод `setParameter()`, передавая ему именованный параметр и значение для него. В случае нескольких параметров можно применять метод `setParameterList()` или `setParameters()` интерфейса `Query`.

Существуют также и более сложные методы запросов, такие как собственный запрос или запрос с критерием, которые мы обсудим в следующей главе при рассмотрении JPA.

Чтобы протестировать метод `findById()`, изменим код класса `SpringHibernateSample`, как показано в листинге 7.20.

**Листинг 7.20. Тестирование метода `findById()`**

```
package com.apress.prospring4.ch7;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringHibernateSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();

        ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

        Contact contact = contactDao.findById(11);
        System.out.println("");
        System.out.println("Contact with id 1:" + contact);
        System.out.println("");
    }
}
```

Запуск тестовой программы дает следующий вывод:

```
Contact with id 1:Contact - Id: 1, First name: Chris,
Last name: Schaefer, Birthday: 1981-05-03
```

## Вставка данных

Вставка данных в Hibernate осуществляется очень просто. Единственным нестандартным действием является извлечение первичного ключа, сгенерированного базой данных. Как было показано в предыдущей главе, в JDBC нужно было явно заявлять, что мы хотим извлечь сгенерированный ключ, передавая экземпляр `KeyHolder` и получая из него ключ после выполнения оператора вставки. В Hibernate все эти действия не требуются. Инфраструктура Hibernate извлечет сгенерированный ключ и заполнит объект предметной области после вставки. Реализация метода `save()` представлена в листинге 7.21.

**Листинг 7.21. Реализация метода `save()`**

```
package com.apress.prospring4.ch7;

import org.springframework.transaction.annotation.Transactional;
import org.springframework.stereotype.Repository;
import org.apache.commons.logging.Log;
```

```
import org.apache.commons.logging.LogFactory;
import org.hibernate.SessionFactory;
import javax.annotation.Resource;
import java.util.List;

@Transactional
@Repository("contactDao")
public class ContactDaoImpl implements ContactDao {
    private static final Log LOG = LogFactory.getLog(ContactDaoImpl.class);

    private SessionFactory sessionFactory;

    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return sessionFactory.getCurrentSession()
          .createQuery("from Contact c").list();
    }

    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAllWithDetail() {
        return sessionFactory.getCurrentSession().
            getNamedQuery("Contact.findAllWithDetail").list();
    }

    @Override
    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        return (Contact) sessionFactory.getCurrentSession().
            getNamedQuery("Contact.findById").
            setParameter("id", id).uniqueResult();
    }

    @Override
    public Contact save(Contact contact) {
        sessionFactory.getCurrentSession().saveOrUpdate(contact);
        LOG.info("Contact saved with id: " + contact.getId());
        return contact;
    }

    @Override
    public void delete(Contact contact) {

    }

    public SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    @Resource(name="sessionFactory")
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
}
```

Мы должны лишь вызвать метод `Session.saveOrUpdate()`, который может использоваться для операций вставки и обновления. Мы также фиксируем в журнале идентификатор сохраненного объекта контакта, который будет заполняться

Hibernate после помещения объекта в базу данных. В листинге 7.22 показан код для вставки новой записи контакта в классе `SpringHibernateSample`.

**Листинг 7.22. Тестирование операции вставки**

```
package com.apress.prospring4.ch7;

import java.util.List;
import java.util.Date;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringHibernateSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();

      ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

      Contact contact = new Contact();
      contact.setFirstName("Michael");
      contact.setLastName("Jackson");
      contact.setBirthDate(new Date());

      ContactTelDetail contactTelDetail =
          new ContactTelDetail("Home", "1111111111");

      contact.addContactTelDetail(contactTelDetail);

      contactTelDetail = new ContactTelDetail("Mobile", "2222222222");

      contact.addContactTelDetail(contactTelDetail);

      contactDao.save(contact);

      listContactsWithDetail(contactDao.findAllWithDetail());
   }

   private static void listContactsWithDetail(List<Contact> contacts) {
      System.out.println("");
      System.out.println("Listing contacts with details:");
      for (Contact contact: contacts) {
         System.out.println(contact);

         if (contact.getContactTelDetails() != null) {
            for (ContactTelDetail contactTelDetail:
                   contact.getContactTelDetails()) {
               System.out.println(contactTelDetail);
            }
         }

         if (contact.getHobbies() != null) {
            for (Hobby hobby: contact.getHobbies()) {
               System.out.println(hobby);
            }
         }
         System.out.println();
      }
   }
}
```

В листинге 7.22 мы создаем новый объект контакта, добавляем сведения о двух телефонах и сохраняем этот объект. Затем мы выводим список всех контактов. Запуск этой программы дает следующий вывод:

```
Hibernate: insert into contact_tel_detail (ID, CONTACT_ID, TEL_NUMBER,
TEL_TYPE, VERSION) values (null, ?, ?, ?, ?)
18:04:00,240  INFO com.apress.prospring4.ch7.ContactDaoImpl:
38 - Contact saved with id: 4

Listing contacts with details:
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
Contact Tel Detail - Id: 2, Contact id: 1, Type: Home, Number: 1234567890
Hobby :Movies
Hobby :Swimming

Contact - Id: 4, First name: Michael, Last name: Jackson, Birthday: 2013-07-25
Contact Tel Detail - Id: 4, Contact id: 4, Type: Mobile, Number: 2222222222
Contact Tel Detail - Id: 5, Contact id: 4, Type: Home, Number: 1111111111

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming
```

В записи INFO журнала видно, что идентификатор вновь сохраненного контакта заполнен корректно. Инфраструктура Hibernate также выводит все SQL-операторы, выполняемые в базе данных, поэтому вы получаете хорошее представление о том, что происходит "за кулисами".

## Обновление данных

Обновление контакта производится так же просто, как и вставка данных. Предположим, что для контакта с идентификатором 1 необходимо обновить имя и удалить запись с домашним телефоном. Чтобы протестировать операцию обновления, внесите в код класса SpringHibernateSample изменения, приведенные в листинге 7.23.

**Листинг 7.23. Тестирование операции обновления**

```
package com.apress.prospring4.ch7;

import java.util.List;
import java.util.Date;
import java.util.Set;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringHibernateSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();

      ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

      Contact contact = contactDao.findById(1l);
       contact.setFirstName("Kim Fung");
```

```
        Set<ContactTelDetail> contactTels = contact.getContactTelDetails();
        ContactTelDetail toDeleteContactTel = null;
        for (ContactTelDetail contactTel: contactTels) {
            if (contactTel.getTelType().equals("Home")) {
                toDeleteContactTel = contactTel;
            }
        }
        contact.removeContactTelDetail(toDeleteContactTel);
        contactDao.save(contact);

        listContactsWithDetail(contactDao.findAllWithDetail());
    }
    private static void listContactsWithDetail(List<Contact> contacts) {
        System.out.println("");
        System.out.println("Listing contacts with details:");
        for (Contact contact: contacts) {
            System.out.println(contact);
            if (contact.getContactTelDetails() != null) {
                for (ContactTelDetail contactTelDetail:
                        contact.getContactTelDetails()) {
                    System.out.println(contactTelDetail);
                }
            }
            if (contact.getHobbies() != null) {
                for (Hobby hobby: contact.getHobbies()) {
                    System.out.println(hobby);
                }
            }
            System.out.println();
        }
    }
}
```

Как показано в предшествующем листинге, мы сначала извлекаем запись с идентификатором 1, а затем изменяем имя. После этого мы проходим в цикле по объектам с деталями телефонов, извлекаем объект с типом "Home" и удаляем его из контакта. Наконец, мы вызываем метод `ContactDao.save()`. Запуск программы дает в результате следующий вывод:

```
Listing contacts with details:
Contact - Id: 1, First name: Kim Fung, Last name: Schaefer, Birthday: 1981-05-03
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
Hobby :Movies
Hobby :Swimming

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming
```

Как видите, имя контакта было обновлено, а сведения о домашнем телефоне удалены. Запись о телефоне может быть удалена, поскольку мы передаем ассоциации "один ко многим" атрибут `orphanRemoval=true`, который инструктирует Hibernate о том, что все висячие (т.е. не связанные с контактами) записи в базе данных должны быть удалены.

## Удаление данных

Удаление данных также выполняется просто. Необходимо лишь вызвать метод `Session.delete()` и передать ему объект контакта. В листинге 7.24 показан код реализации этого метода.

**Листинг 7.24. Реализация метода `delete()`**

```
package com.apress.prospring4.ch7;

import org.springframework.transaction.annotation.Transactional;
import org.springframework.stereotype.Repository;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.SessionFactory;
import javax.annotation.Resource;
import java.util.List;

@Transactional
@Repository("contactDao")
public class ContactDaoImpl implements ContactDao {
    private static final Log LOG = LogFactory.getLog(ContactDaoImpl.class);

    private SessionFactory sessionFactory;
    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return sessionFactory.getCurrentSession()
          .createQuery("from Contact c").list();
    }
    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAllWithDetail() {
        return sessionFactory.getCurrentSession().
            getNamedQuery("Contact.findAllWithDetail").list();
    }
    @Override
    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        return (Contact) sessionFactory.getCurrentSession().
            getNamedQuery("Contact.findById").
            setParameter("id", id).uniqueResult();
    }
    @Override
    public Contact save(Contact contact) {
        sessionFactory.getCurrentSession().saveOrUpdate(contact);
        LOG.info("Contact saved with id: " + contact.getId());
        return contact;
    }
```

```
    @Override
    public void delete(Contact contact) {
        sessionFactory.getCurrentSession().delete(contact);
        LOG.info("Contact deleted with id: " + contact.getId());
    }

    public SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    @Resource(name="sessionFactory")
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
}
```

Операция удаления удалит запись контакта, а также всю связанную с ней информацию, включая детали о телефонах и хобби, т.к. в отображении было определено `cascade=CascadeType.ALL`. В листинге 7.25 приведен код для тестирования метода удаления в классе `SpringHibernateSample`.

**Листинг 7.25. Тестирование операции удаления**

```
package com.apress.prospring4.ch7;

import java.util.List;
import java.util.Date;
import java.util.Set;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringHibernateSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();

        ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

        Contact contact = contactDao.findById(1l);

        contactDao.delete(contact);

        listContactsWithDetail(contactDao.findAllWithDetail());
    }

    private static void listContactsWithDetail(List<Contact> contacts) {
        System.out.println("");
        System.out.println("Listing contacts with details:");

        for (Contact contact: contacts) {
            System.out.println(contact);

            if (contact.getContactTelDetails() != null) {
                for (ContactTelDetail contactTelDetail:
                        contact.getContactTelDetails()) {
                    System.out.println(contactTelDetail);
                }
            }
```

```
            if (contact.getHobbies() != null) {
                for (Hobby hobby: contact.getHobbies()) {
                    System.out.println(hobby);
                }
            }
            System.out.println();
        }
    }
}
```

В показанном выше листинге мы извлекаем контакт с идентификатором 1 и затем вызываем метод `delete()` для удаления информации о контакте. Запуск этой программы даст следующий вывод:

```
Listing contacts with details:
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming
```

Как видите, контакт с идентификатором 1 был успешно удален.

# Соображения по поводу использования Hibernate

Примеры в этой главе продемонстрировали, что при правильном определении объектно-реляционного отображения, ассоциаций и запросов инфраструктура Hibernate может обеспечить среду, которая позволит сосредоточить внимание на программировании с использованием объектной модели, а не на построении SQL-операторов для каждой операции. В последние несколько лет Hibernate быстро развивалась и была принята широким сообществом Java-разработчиков в качестве библиотеки для уровня доступа к данным.

Тем не менее, необходимо помнить о нескольких важных моментах. Поскольку вы не имеете никакого контроля над генерируемыми SQL-операторами, то должны очень тщательно определять отображения, особенно ассоциации и связанную с ними стратегию выборки. Затем понаблюдайте за SQL-операторами, которые генерирует Hibernate, чтобы удостовериться в их корректном поведении.

Понимание внутреннего механизма, согласно которому Hibernate управляет своим сеансом, также играет очень важную роль, особенно в пакетных операциях. Инфраструктура Hibernate хранит управляемые объекты внутри сеанса и регулярно сбрасывает их и очищает. Неудачно спроектированная логика доступа к данным может привести к тому, что Hibernate будет сбрасывать сеанс слишком часто, из-за чего существенно снизится производительность. Если нужен полный контроль над запросом, можно воспользоваться собственным запросом, который рассматривается в следующей главе.

Наконец, важную роль в настройке производительности Hibernate играют и параметры (размер пакета, размер выборки и т.д.). Вы должны определить их в фабрике сеансов и корректировать во время нагрузочного тестирования приложения для нахождения оптимальных значений.

В конечном счете, инфраструктура Hibernate и ее великолепная поддержка JPA, которую мы обсудим в следующей главе, представляют собой естественное решение для Java-разработчиков, нуждающихся в объектно-ориентированном подходе при реализации логики доступа к данным.

# Резюме

В этой главе мы рассмотрели базовые концепции Hibernate и показали, как они конфигурируются в рамках приложения Spring. Мы также представили общие приемы для определения отображений ORM, раскрыли понятие ассоциаций и продемонстрировали использование класса `HibernateTemplate` для выполнения разнообразных операций в базе данных.

В настоящей главе мы смогли охватить лишь небольшую часть функциональности и возможностей Hibernate. Мы настоятельно рекомендуем изучить стандартную документацию по Hibernate, если вы планируете применять Hibernate вместе с платформой Spring. Кроме того, есть немало книг, посвященных этой теме, в частности, *Beginning Hibernate*, 3-е изд. (Apress, 2014 г.) и *Pro JPA 2* (Apress, 2013 г.).

В следующей главе мы обсудим интерфейс JPA и его использование в Spring. Инфраструктура Hibernate предлагает великолепную поддержку JPA, поэтому в примерах следующей главы мы продолжим применять Hibernate в качестве поставщика службы постоянства. Для операций выборки и обновления JPA действует подобно Hibernate. В следующей главе мы рассмотрим несколько дополнительных тем, включая собственный запрос и запрос с критерием, а также использование Hibernate вместе с поддержкой JPA.

ГЛАВА 8

# Доступ к данным в Spring с использованием JPA 2

В предыдущей главе было показано, как использовать Hibernate в Spring при реализации логики доступа к данным с помощью подхода ORM. Мы продемонстрировали возможности настройки фабрики сеансов Hibernate в конфигурации Spring и объяснили, как применять интерфейс `Session` для выполнения разнообразных операций доступа к данным. Однако это только один способ использования Hibernate. Другой прием эксплуатации Hibernate в приложении Spring предусматривает применение Hibernate в качестве поставщика службы постоянства, соответствующего стандартному API-интерфейсу постоянства Java (Java Persistence API — JPA).

Отображение POJO в Hibernate и мощный язык запросов HQL добились большого успеха и также повлияли на разработку стандартов, определяющих технологии доступа к данным в мире Java. После Hibernate в рамках JCP был подготовлен стандарт JDO (Java Data Objects — объекты данных Java) и затем JPA.

На момент написания этой книги интерфейс JPA достиг версии 2.1 и поддерживает такие стандартизованные концепции, как `PersistenceContext`, `EntityManager` и JPQL (Java Persistence Query Language — язык запросов постоянства Java). Эта стандартизация предоставляет разработчикам способ переключения между поставщиками постоянства JPA, к которым относятся Hibernate, EclipseLink, Oracle TopLink и Apache OpenJPA. В результате большинство новых JEE-приложений используют JPA в качестве уровня доступа к данным.

Платформа Spring также предлагает великолепную поддержку JPA. Например, имеется несколько бинов `EntityManagerFactoryBean` для начальной загрузки диспетчера сущностей JPA с поддержкой всех упомянутых ранее поставщиков JPA. Внутри проекта Spring Data также ведется подпроект Spring Data JPA, который ориентирован на расширенную поддержку использования JPA в Spring-приложениях. В число основных возможностей проекта Spring Data JPA входят концепции `Repository` и `Specification`, а также поддержка языка запросов, специфичного для предметной области (Query Domain Specific Language — QueryDSL).

В этой главе мы обсудим применение JPA 2.1 в Spring, используя Hibernate в качестве лежащего в основе поставщика службы постоянства. Вы узнаете, каким образом реализовать разнообразные операции базы данных с применением интерфейса `EntityManager` из JPA и языка JPQL. Затем мы покажем, как проект Spring Data JPA может дополнительно упростить разработку с помощью JPA. Наконец, мы рассмотрим более сложные темы, относящиеся к ORM, в том числе собственные запросы и запросы с критериями.

В частности, в этой главе мы обсудим следующие темы.

- **Ключевые концепции JPA.** Мы раскроем некоторые главные концепции JPA.
- **Конфигурирование диспетчера сущностей JPA.** Мы обсудим типы `EntityManagerFactory`, поддерживаемые Spring, и способы конфигурирования наиболее часто используемого из них — класса `LocalContainerEntityManagerFactoryBean` — в XML-конфигурации Spring.
- **Операции с данными.** Мы покажем, каким образом реализовать элементарные операции базы данных в JPA, которые концептуально похожи на операции, создаваемые с помощью Hibernate.
- **Расширенные операции запросов.** Мы рассмотрим применение собственных запросов в JPA и строго типизированный API-интерфейс критериев для реализации более гибких операций запросов.
- **Введение в Spring Data JPA.** Мы обсудим проект Spring Data JPA и покажем, каким образом он позволяет упростить процесс разработки логики доступа к данным.
- **Отслеживание изменений в сущностях и аудит.** Общим требованием в операциях обновления базы данных является отслеживание дат создания и обновления сущностей, а также того, кто производил изменение. Кроме того, всегда требуется такая критически важная информация, как пользователь и таблица хронологии, хранящая все версии сущности. Мы покажем, как Spring Data JPA и Hibernate Envers (Hibernate Entity Versioning System — система управления версиями сущностей Hibernate) могут упростить разработку логики подобного рода.

---

**На заметку!** Подобно Hibernate, JPA поддерживает определение отображений либо в XML, либо в Java-аннотациях. В этой главе мы сосредоточимся на отображении с помощью аннотаций, поскольку такой стиль более популярен, чем вариант с XML.

---

# Введение в JPA 2.1

Подобно другим запросам спецификации Java (Java Specification Request — JSR), цель спецификации JPA 2.1 (JSR-338) заключается в стандартизации программной модели ORM в средах JSE и JEE. В ней определен общий набор концепций, аннотаций, интерфейсов и других служб, которые поставщик постоянства JPA должен реализовывать. В случае программирования в соответствие со стандартом JPA разработчики имеют возможность смены лежащего в основе поставщика по своему желанию, что похоже на переключение на другой JEE-совместимый сервер приложений для приложений, построенных согласно стандартам JEE.

В основе JPA лежит интерфейс `EntityManager`, который поступает из фабрик типа `EntityManagerFactory`. Главной задачей `EntityManager` является поддержка контекста постоянства, внутри которого будут храниться все экземпляры сущностей, управляемые этим контекстом. Конфигурация `EntityManager` определена как единица постоянства, и в приложении может существовать более одной такой единицы. Если вы используете Hibernate, то можете считать контекст постоянства тем же, что и интерфейс `Session`, а `EntityManagerFactory` — тем же, что и `SessionFactory`. В Hibernate управляемые сущности сохраняются в сеансе, с которым можно взаимодействовать напрямую через Hibernate-интерфейс `SessionFactory` или `Session`. Однако в JPA непосредственно взаимодействовать с контекстом постоянства нельзя. Вместо этого для выполнения всей необходимой работы следует полагаться на `EntityManager`.

Язык JPQL очень похож на HQL, поэтому если вы ранее работали с HQL, то перейти на JPQL должно быть легко. Тем не менее, в JPA 2 появился строго типизированный API-интерфейс критериев, который при конструировании запроса полагается на метаданные отображенных сущностей. Учитывая это, любые ошибки будут обнаруживаться во время компиляции, а не во время выполнения.

За более детальными сведениями о спецификации JPA 2 рекомендуем обратиться к книге *Pro JPA 2* (Apress, 2013 г.).

В этом разделе мы обсудим базовые концепции JPA, пример модели данных, который будет рассматриваться в главе, и способы конфигурирования `ApplicationContext` для поддержки JPA.

## Использование модели данных для кода примеров

В этой главе мы будем пользоваться той же самой моделью данных, что и в главе 7. Однако при обсуждении реализации средств аудита в целях демонстрации мы добавим несколько столбцов и таблицу хронологии. Таким образом, мы начнем с тех же сценариев создания базы данных, которые приводились в предыдущей главе. Если вы по каким-то причинам пропустили главу 7, взгляните на модель данных, представленную в разделе "Модель данных для кода примера"; это поможет понять пример кода в настоящей главе.

## Конфигурирование `EntityManagerFactory` из JPA

Ранее в этой главе упоминалось, что для применения JPA в Spring необходимо сконфигурировать `EntityManagerFactory`, как это делалось для `SessionFactory` в Hibernate.

В Spring поддерживаются три типа конфигурации `EntityManagerFactory`.

В первом типе конфигурации используется класс `LocalEntityManagerFactoryBean`. Это простейший случай, который требует только указания имени единицы постоянства. Однако поскольку он не поддерживает внедрение источника данных и, следовательно, не имеет возможности участвовать в глобальных транзакциях, такой тип конфигурации подходит только во время разработки.

Второй тип конфигурации применяется для совместимого с JEE 6 контейнера, в который сервер приложений производит начальную загрузку единицы постоянства JPA на основе информации в дескрипторах развертывания. Это позволяет Spring находить диспетчер сущностей с помощью поиска JNDI. В листинге 8.1 приведен фрагмент кода для поиска диспетчера сущностей через JNDI.

**Листинг 8.1. Поиск диспетчера сущностей через JNDI**

```
<beans>
   <jee:jndi-lookup id="prospring4Emf"
      jndi-name="persistence/prospring4PersistenceUnit"/>
</beans>
```

В спецификации JPA единица постоянства должна быть определена в конфигурационном файле `META-INF/persistence.xml`. Тем не менее, в Spring 3.1 была добавлена возможность, которая устраняет потребность в этом; позже в главе мы покажем, как ею пользоваться.

Третий тип конфигурации, который является наиболее распространенным и применяется в этой главе — класс `LocalContainerEntityManagerFactoryBean`. Он поддерживает внедрение источника данных и может принимать участие как в локальных, так и в глобальных транзакциях. В листинге 8.2 показано содержимое соответствующего XML-файла конфигурации (`app-context-annotation.xml`).

**Листинг 8.2. Конфигурация Spring для `LocalContainerEntityManagerFactoryBean`**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc=http://www.springframework.org/schema/jdbc
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:META-INF/sql/schema.sql"/>
      <jdbc:script location="classpath:META-INF/sql/test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionManager"
         class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <tx:annotation-driven transaction-manager="transactionManager" />

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
        />
      </property>
      <property name="packagesToScan" value="com.apress.prospring4.ch8"/>
```

```
    <property name="jpaProperties">
      <props>
        <prop key="hibernate.dialect">
          org.hibernate.dialect.H2Dialect</prop>
        <prop key="hibernate.max_fetch_depth">3</prop>
        <prop key="hibernate.jdbc.fetch_size">50</prop>
        <prop key="hibernate.jdbc.batch_size">10</prop>
        <prop key="hibernate.show_sql">true</prop>
      </props>
    </property>
  </bean>

  <context:component-scan base-package="com.apress.prospring4.ch8" />
</beans>
```

В приведенной выше конфигурации объявлено несколько бинов для поддержки конфигурации `LocalContainerEntityManagerFactoryBean` с Hibernate в качестве поставщика постоянства. Ниже описаны основные элементы этой конфигурации.

- **Бин `dataSource`.** Мы объявляем источник данных со встроенной базой данных типа H2. Поскольку это встроенная база данных, указывать имя для нее не требуется.
- **Бин `transactionManager`.** Фабрика диспетчера сущностей требует диспетчера транзакций для транзакционного доступа к данным. В Spring предусмотрен диспетчер транзакций специально для JPA (`org.springframework.orm.jpa.JpaTransactionManager`). Этот бин объявлен с идентификатором `transactionManager`. Транзакции более подробно рассматриваются в главе 9. Дескриптор `<tx:annotation-driven>` предназначен для поддержки объявления требований к установлению границ транзакций с использованием аннотаций.
- **Дескриптор `component-scan`.** Этот дескриптор уже должен быть знаком. Мы указываем Spring на необходимость сканирования компонентов в пакете `com.apress.prospring4.ch8`.
- **Бин фабрики диспетчера сущностей JPA.** Бин `emf` является наиболее важным элементом конфигурации. Первым делом мы объявляем, что этот бин использует класс `LocalContainerEntityManagerFactoryBean`. Внутри бина предоставляются различные свойства. Во-первых, как и можно было ожидать, необходимо внедрить бин источника данных. Во-вторых, мы указываем в свойстве `jpaVendorAdapter` класс `HibernateJpaVendorAdapter`, т.к. применяется Hibernate. В-третьих, мы инструктируем фабрику сущностей относительно поиска объектов предметной области с аннотациями ORM в пакете `com.apress.prospring4.ch8` (указанном с помощью дескриптора `<property name="packagesToScan">`). Обратите внимание, что эта возможность доступна только начиная с версии Spring 3.1, и за счет поддержки сканирования классов предметной области можно опустить определение единицы постоянства в файле `META-INF/persistence.xml`. В-четвертых, свойство `jpaProperties` предоставляет детали конфигурации для поставщика постоянства, т.е. Hibernate. Вы увидите, что здесь применяется та же самая конфигурация, которая использовалась в главе 7, поэтому мы не приводим ее пояснения снова.

## Использование аннотаций JPA для отображения ORM

Инфраструктура Hibernate оказала большое влияние на проектное решение, положенное в основу JPA. Аннотации отображения в JPA очень близки к тем аннотациям, которые применялись в главе 7 для отображения объектов предметной области на базу данных. Если вы взглянете на исходный код классов предметной области в главе 7, то заметите, что все аннотации отображения находятся в пакете `javax.persistence`, а это говорит об их совместимости с JPA.

После того как `EntityManagerFactory` соответствующим образом сконфигурирован, его внедрение в классы осуществляется очень просто. В листинге 8.3 приведен код класса `ContactServiceImpl`, который будет использоваться в качестве примера для выполнения операций базы данных с применением JPA.

**Листинг 8.3. Внедрение диспетчера сущностей**

```
package com.apress.prospring4.ch8;

import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    private Log log = LogFactory.getLog(ContactServiceImpl.class);
    @PersistenceContext
    private EntityManager em;
    @Override
    public List<Contact> findAll() {
        return null;
    }
    @Override
    public List<Contact> findAllWithDetail() {
        return null;
    }
    @Override
    public Contact findById(Long id) {
        return null;
    }
    @Override
    public Contact save(Contact contact) {
        return null;
    }
    @Override
    public void delete(Contact contact) {
    }
}
```

К классу применено несколько аннотаций. Аннотация `@Service` предназначена для идентификации класса как компонента Spring, который предоставляет бизнес-службы другому уровню; этот бин Spring получает имя `jpaContactService`. Аннотация `@Repository` указывает, что класс содержит логику доступа к данным и заставляет Spring транслировать исключения, специфичные для поставщика, в иерархию `DataAccessException`, определенную Spring. Как вам уже должно быть известно, аннотация `@Transactional` используется для определения требований к транзакции.

Для внедрения `EntityManager` мы используем аннотацию `@PersistenceContext`, которая представляет собой стандартную аннотацию JPA для внедрения диспетчера сущностей. Вас может удивить, почему для внедрения диспетчера сущностей применяется имя `@PersistenceContext`, но если принять во внимание, что сам контекст постоянства управляется посредством `EntityManager`, то именование аннотаций обретает большой смысл. При наличии в приложении множества единиц постоянства можно также добавить к аннотации атрибут `unitName` для указания единицы постоянства, подлежащей внедрению. Обычно единица постоянства представляет отдельный источник данных.

# Выполнение операций базы данных с помощью JPA

В этом разделе мы покажем, как выполнять операции базы данных в JPA. В листинге 8.4 приведен интерфейс `ContactService`, отражающий службы информации о контактах, которые мы собираемся предоставлять.

**Листинг 8.4. Интерфейс `ContactService`**

```
package com.apress.prospring4.ch8;

import java.util.List;

public interface ContactService {
    List<Contact> findAll();
    List<Contact> findAllWithDetail();
    Contact findById(Long id);
    Contact save(Contact contact);
    void delete(Contact contact);
}
```

Интерфейс очень прост; он имеет три метода поиска, один метод сохранения и один метод удаления. Метод `save()` будет выполнять операции вставки и обновления.

## Использование языка JPQL для запрашивания данных

Синтаксис JPQL и HQL очень похож и на самом деле все запросы HQL, которые применялись в главе 7, можно использовать повторно для реализации трех методов поиска в интерфейсе `ContactService`. Для применения JPA и Hibernate в проект понадобится добавить зависимости, описанные в табл. 8.1.

**Таблица 8.1. Зависимости для Hibernate и JPA**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| org.hibernate | hibernate-entitymanager | 4.2.3.Final | Библиотека Hibernate 4 |
| org.hibernate.javax.persistence | hibernate-jpa-2.1-api | 1.0.0.Final | Библиотека JPA |

Для удобства в листинге 8.5 воспроизведен код классов модели предметной области из главы 7.

**Листинг 8.5. Классы модели предметной области**

```
package com.apress.prospring4.ch8;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
import javax.persistence.Version;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.OneToMany;
import javax.persistence.ManyToMany;
import javax.persistence.JoinTable;
import javax.persistence.JoinColumn;
import javax.persistence.CascadeType;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;

@Entity
@Table(name = "contact")
@NamedQueries({
    @NamedQuery(name="Contact.findAll", query="select c from Contact c"),
    @NamedQuery(name="Contact.findById",
        query="select distinct c from Contact c left join fetch
c.contactTelDetails t left join fetch c.hobbies h where c.id = :id"),
    @NamedQuery(name="Contact.findAllWithDetail",
        query="select distinct c from Contact c left join fetch
c.contactTelDetails t left join fetch c.hobbies h")
})
public class Contact implements Serializable {
    private Long id;
    private int version;
    private String firstName;
    private String lastName;
```

```
private Date birthDate;
private Set<ContactTelDetail> contactTelDetails =
    new HashSet<ContactTelDetail>();
private Set<Hobby> hobbies = new HashSet<Hobby>();

@Id
@GeneratedValue(strategy = IDENTITY)
@Column(name = "ID")
public Long getId() {
    return this.id;
}

public void setId(Long id) {
    this.id = id;
}

@Version
@Column(name = "VERSION")
public int getVersion() {
    return this.version;
}

public void setVersion(int version) {
    this.version = version;
}

@Column(name = "FIRST_NAME")
public String getFirstName() {
    return this.firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

@Column(name = "LAST_NAME")
public String getLastName() {
    return this.lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

@Temporal(TemporalType.DATE)
@Column(name = "BIRTH_DATE")
public Date getBirthDate() {
    return this.birthDate;
}

public void setBirthDate(Date birthDate) {
    this.birthDate = birthDate;
}

@OneToMany(mappedBy = "contact", cascade=CascadeType.ALL,
    orphanRemoval=true)
public Set<ContactTelDetail> getContactTelDetails() {
    return this.contactTelDetails;
}
```

```
    public void setContactTelDetails(Set<ContactTelDetail> contactTelDetails)
    {
     this.contactTelDetails = contactTelDetails;
    }

    public void addContactTelDetail(ContactTelDetail contactTelDetail) {
        contactTelDetail.setContact(this);
        getContactTelDetails().add(contactTelDetail);
    }

    public void removeContactTelDetail(ContactTelDetail contactTelDetail) {
        getContactTelDetails().remove(contactTelDetail);
    }

    @ManyToMany
    @JoinTable(name = "contact_hobby_detail",
        joinColumns = @JoinColumn(name = "CONTACT_ID"),
        inverseJoinColumns = @JoinColumn(name = "HOBBY_ID"))
    public Set<Hobby> getHobbies() {
        return this.hobbies;
    }

    public void setHobbies(Set<Hobby> hobbies) {
        this.hobbies = hobbies;
    }

    @Override
    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate;
    }
}
```

```
package com.apress.prospring4.ch8;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
import javax.persistence.Version;
import javax.persistence.ManyToOne;
import javax.persistence.JoinColumn;

@Entity
@Table(name = "contact_tel_detail")
public class ContactTelDetail implements Serializable {
    private Long id;
    private int version;
    private String telType;
    private String telNumber;
    private Contact contact;

    public ContactTelDetail() {
    }
```

```
    public ContactTelDetail(String telType, String telNumber) {
        this.telType = telType;
        this.telNumber = telNumber;
    }

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Version
    @Column(name = "VERSION")
    public int getVersion() {
        return this.version;
    }

    public void setVersion(int version) {
        this.version = version;
    }

    @Column(name = "TEL_TYPE")
    public String getTelType() {
        return this.telType;
    }

    public void setTelType(String telType) {
        this.telType = telType;
    }

    @Column(name = "TEL_NUMBER")
    public String getTelNumber() {
        return this.telNumber;
    }

    public void setTelNumber(String telNumber) {
        this.telNumber = telNumber;
    }

    @ManyToOne
    @JoinColumn(name = "CONTACT_ID")
    public Contact getContact() {
        return this.contact;
    }

    public void setContact(Contact contact) {
        this.contact = contact;
    }

    @Override
    public String toString() {
        return "Contact Tel Detail - Id: " + id + ", Contact id: "
            + getContact().getId() + ", Type: "
            + telType + ", Number: " + telNumber;
    }
}
```

```
package com.apress.prospring4.ch8;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Column;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.JoinTable;
import javax.persistence.JoinColumn;
import java.util.Set;
import java.util.HashSet;

@Entity
@Table(name = "hobby")
public class Hobby implements Serializable {
    private String hobbyId;
    private Set<Contact> contacts = new HashSet<Contact>();

    @Id
    @Column(name = "HOBBY_ID")
    public String getHobbyId() {
        return this.hobbyId;
    }

    public void setHobbyId(String hobbyId) {
        this.hobbyId = hobbyId;
    }

    @ManyToMany
    @JoinTable(name = "contact_hobby_detail",
        joinColumns = @JoinColumn(name = "HOBBY_ID"),
        inverseJoinColumns = @JoinColumn(name = "CONTACT_ID"))
    public Set<Contact> getContacts() {
        return this.contacts;
    }

    public void setContacts(Set<Contact> contacts) {
        this.contacts = contacts;
    }

    @Override
    public String toString() {
        return "Hobby :" + getHobbyId();
    }
}
```

Если сравнить запросы в листинге 8.5 с запросами, которые применялись в главе 7, то никаких отличий не обнаружится. Таким образом, в случае использования Hibernate переход на JPA производится относительно легко.

Давайте начнем с метода `findAll()`, который просто извлекает все контакты из базы данных. В листинге 8.6 приведен модифицированный код.

**Листинг 8.6. Реализация метода `findAll()`**

```
package com.apress.prospring4.ch8;

import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    private Log log = LogFactory.getLog(ContactServiceImpl.class);

    @PersistenceContext
    private EntityManager em;

    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAll() {
        List<Contact> contacts = em.createNamedQuery("Contact.findAll",
            Contact.class).getResultList();
        return contacts;
    }

    @Override
    public List<Contact> findAllWithDetail() {
        return null;
    }

    @Override
    public Contact findById(Long id) {
        return null;
    }

    @Override
    public Contact save(Contact contact) {
        return null;
    }
    @Override
    public void delete(Contact contact) {

    }
}
```

В этом листинге видно, что мы используем метод `EntityManager.createNamedQuery()`, передавая ему имя запроса и ожидаемый тип возврата. В данном случае `EntityManager` возвратит реализацию интерфейса `TypedQuery<X>`. Затем вызывается метод `TypedQuery.getResultList()` для извлечения контактов.

В листинге 8.7 приведен код простой программы для тестирования класса `ContactServiceImpl`.

**Листинг 8.7. Тестирование класса `ContactServiceImpl`**

```
package com.apress.prospring4.ch8;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringJPASample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx =
        new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();

      ContactService contactService = ctx.getBean(
         "jpaContactService", ContactService.class);

      listContacts(contactService.findAll());
   }

   private static void listContacts(List<Contact> contacts) {
      System.out.println("");
      // Вывести список контактов без подробных сведений.
      System.out.println("Listing contacts without details:");
      for (Contact contact: contacts) {
         System.out.println(contact);
         System.out.println();
      }
   }
```

Запуск предыдущего кода на выполнение дает следующий вывод:

```
Listing contacts without details:
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

**На заметку!** Для ассоциаций в спецификации JPA утверждается, что по умолчанию поставщики постоянства должны производить выборку ассоциации незамедлительно. Однако в реализации JPA, предлагаемой Hibernate, по умолчанию по-прежнему используется стратегия отложенной выборки. Поэтому в таком случае явное определение ассоциации с отложенной выборкой не требуется. Стандартная стратегия выборки Hibernate отличается от описанной в спецификации JPA.

А теперь реализуем метод `findAllWithDetail()`, который будет выбирать все связанные телефоны и хобби. В листинге 8.8 показан модифицированный код класса `ContactServiceImpl`.

**Листинг 8.8. Реализация метода `FindAllWithDetail()`**

```
package com.apress.prospring4.ch8;

import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    private Log log = LogFactory.getLog(ContactServiceImpl.class);

    @PersistenceContext
    private EntityManager em;

    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAll() {
        List<Contact> contacts = em.createNamedQuery("Contact.findAll",
            Contact.class).getResultList();
        return contacts;
    }

    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAllWithDetail() {
        List<Contact> contacts = em.createNamedQuery(
            "Contact.findAllWithDetail", Contact.class).getResultList();
        return contacts;
    }
    @Override
    public Contact findById(Long id) {
        return null;
    }
    @Override
    public Contact save(Contact contact) {
        return null;
    }
    @Override
    public void delete(Contact contact) {
    }
}
```

Метод `findAllWithDetail()` похож на `findAll()`, но использует другой именованный запрос с включенной конструкцией `left join fetch`. В листинге 8.9 приведена переделанная тестовая программа для вывода деталей, связанных с контактами.

**Листинг 8.9. Тестирование класса `ContactServiceImpl`**

```
package com.apress.prospring4.ch8;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringJPASample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();

      ContactService contactService = ctx.getBean(
          "jpaContactService", ContactService.class);

      List<Contact> contacts = contactService.findAllWithDetail();
      listContactsWithDetail(contacts);
   }

   private static void listContactsWithDetail(List<Contact> contacts) {
       System.out.println("");
       System.out.println("Listing contacts with details:");
       for (Contact contact: contacts) {
          System.out.println(contact);
          if (contact.getContactTelDetails() != null) {
             for (ContactTelDetail contactTelDetail:
                 contact.getContactTelDetails()) {
                System.out.println(contactTelDetail);
             }
          }
          if (contact.getHobbies() != null) {
             for (Hobby hobby: contact.getHobbies()) {
                System.out.println(hobby);
             }
          }
          System.out.println();
       }
   }
}
```

Снова запустив эту программу, вы получите следующий вывод:

```
Listing contacts with details:
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
Contact Tel Detail - Id: 2, Contact id: 1, Type: Home, Number: 1234567890
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
Hobby :Movies
Hobby :Swimming

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming
```

Давайте следующим рассмотрим метод `findById()`, который демонстрирует применение именованного запроса с именованными параметрами в JPA. Кроме того,

будет произведена выборка ассоциаций. Скорректированная реализация показана в листинге 8.10.

**Листинг 8.10. Реализация метода `findById()`**

```
package com.apress.prospring4.ch8;

package com.apress.prospring4.ch8;

import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    private Log log = LogFactory.getLog(ContactServiceImpl.class);
    @PersistenceContext
    private EntityManager em;
    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAll() {
        List<Contact> contacts = em.createNamedQuery("Contact.findAll",
            Contact.class).getResultList();
        return contacts;
    }
    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAllWithDetail() {
        List<Contact> contacts = em.createNamedQuery(
            "Contact.findAllWithDetail", Contact.class).getResultList();
        return contacts;
    }
    @Transactional(readOnly=true)
    @Override
    public Contact findById(Long id) {
        TypedQuery<Contact> query = em.createNamedQuery(
            "Contact.findById", Contact.class);
        query.setParameter("id", id);
        return query.getSingleResult();
    }
    @Override
    public Contact save(Contact contact) {
        return null;
    }
    @Override
    public void delete(Contact contact) {
    }
}
```

Метод `EntityManager.createNamedQuery(java.lang.String name, java.lang.Class<T> resultClass)` вызывается для получения экземпляра реализации интерфейса `TypedQuery<T>`, который гарантирует, что результат запроса должен относиться к типу `Contact`. Затем с помощью метода `TypedQuery<T>.setParameter()` устанавливаются значения именованных параметров внутри запроса, после чего вызывается метод `getSingleResult()`, т.к. результат должен содержать только одиночный объект `Contact` с указанным идентификатором. Тестирование этого метода мы оставляем в качестве упражнения для самостоятельного выполнения.

### *Запрос с нетипизированными результатами*

Во многих случаях возникает необходимость отправить запрос базе данных и манипулировать результатами по собственному усмотрению, а не сохранять их в отображенном сущностном классе. Типичным примером может быть веб-отчет, содержащий список заданного количества столбцов из множества таблиц.

Предположим, что имеется веб-страница, которая отображает итоговую информацию по всем контактам. Для каждого контакта итоговая информация включает имя, фамилию и домашний телефон. Контакты, не имеющие домашнего телефона, не отображаются. Этот сценарий использования мы можем реализовать с помощью запроса и затем вручную манипулировать результирующим набором.

Давайте создадим новый класс `ContactSummaryUntypeImpl` с методом `displayAllContactSummary()`. В листинге 8.11 приведена типовая реализация этого метода.

**Листинг 8.11. Реализация метода `displayAllContactSummary()` в классе `ContactSummaryUntypeImpl`**

```
package com.apress.prospring4.ch8;

import org.springframework.stereotype.Service;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;

import java.util.List;
import java.util.Iterator;

@Service("contactSummaryUntype")
@Repository
@Transactional
public class ContactSummaryUntypeImpl {
    @PersistenceContext
    private EntityManager em;
    @Transactional(readOnly=true)
    public void displayAllContactSummary() {
       List result = em
             .createQuery("select c.firstName, c.lastName, t.telNumber "
                + "from Contact c left join c.contactTelDetails t "
                + " where t.telType='Home'").getResultList();
       int count = 0;
       for (Iterator i = result.iterator(); i.hasNext();) {
          Object[] values = (Object[]) i.next();
```

```
            System.out.println(++count + ": " + values[0] + ", "
              + values[1] + ", " + values[2]);
        }
    }
}
```

Как показано в листинге 8.11, мы используем метод `EntityManager.createQuery()` для создания запроса, передавая ему оператор JPQL, и затем получаем результирующий список.

В случае явного указания выбираемых столбцов в JPQL-операторе JPA возвращает итератор, а каждый элемент внутри этого итератора представляет собой массив объектов. После этого посредством итератора организуется цикл и для каждого элемента массива объектов отображается значение. Каждый массив объектов соответствует записи в результирующем наборе. В листинге 8.12 приведена тестовая программа.

**Листинг 8.12. Тестирование метода `displayAllContactSummary()`**

```
package com.apress.prospring4.ch8;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringJPASample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();

        ContactSummaryUntypeImpl contactSummaryUntype =
            ctx.getBean("contactSummaryUntype",
                ContactSummaryUntypeImpl.class);

        contactSummaryUntype.displayAllContactSummary();
    }
}
```

Запуск тестовой программы дает следующий вывод:

```
1: Chris, Schaefer, 1234567890
2: Scott, Tiger, 1234567890
```

В JPA имеется более элегантное решение, нежели обработка массива объектов, возвращаемого запросом; мы обсудим его в следующем разделе.

### *Запрос со специальным типом результата и конструирующим выражением*

При запрашивании специального результата, подобного рассмотренному в предыдущем разделе, можно заставить JPA напрямую конструировать POJO из каждой записи. Продолжая пример из предыдущего раздела, давайте создадим объект POJO по имени `ContactSummary`, который хранит результаты запроса итоговой информации о контакте. Код класса `ContactSummary` приведен в листинге 8.13.

**Листинг 8.13. Класс `ContactSummary`**

```
package com.apress.prospring4.ch8;

import java.io.Serializable;

public class ContactSummary implements Serializable {
    private String firstName;
    private String lastName;
    private String homeTelNumber;

    public ContactSummary(String firstName, String lastName,
            String homeTelNumber) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.homeTelNumber = homeTelNumber;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getHomeTelNumber() {
        return homeTelNumber;
    }

    public String toString() {
        return "First name: " + firstName + " Last Name: " + lastName
            + " Home Phone: " + homeTelNumber;
    }
}
```

Класс `ContactSummary` имеет свойства для итоговой информации о контакте, а также метод конструктора, который принимает все эти свойства.

Имея класс `ContactSummary`, можно модифицировать метод и воспользоваться конструирующим выражением внутри запроса, чтобы инструктировать поставщик JPA о необходимости отображения результирующего набора на класс `ContactSummary`. Для начала создадим интерфейс `ContactSummary`. Его код представлен в листинге 8.14.

**Листинг 8.14. Интерфейс `ContactSummaryService`**

```
package com.apress.prospring4.ch8;

import java.util.List;

public interface ContactSummaryService {
    List<ContactSummary> findAll();
}
```

В листинге 8.15 приведена реализация метода `ContactSummaryService.findAll()`, в которой используется конструирующее выражение для отображения результирующего набора.

**Листинг 8.15. Реализация метода `findAll()` с использованием конструирующего выражения**

```
package com.apress.prospring4.ch8;

import org.springframework.stereotype.Service;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;

import java.util.List;

@Service("contactSummaryService")
@Repository
@Transactional
public class ContactSummaryServiceImpl implements ContactSummaryService {
    @PersistenceContext
    private EntityManager em;

    @Transactional(readOnly=true)
    @Override
    public List<ContactSummary> findAll() {
        List<ContactSummary> result = em.createQuery(
            "select new com.apress.prospring4.ch8.ContactSummary("
            + "c.firstName, c.lastName, t.telNumber) "
            + "from Contact c left join c.contactTelDetails t "
            + "where t.telType='Home'",
            ContactSummary.class).getResultList();

        return result;
    }
}
```

В операторе JPQL указывается ключевое слово `new` вместе с полностью определенным именем класса POJO, который будет сохранять результаты, и передаются выбираемые атрибуты как аргумент конструктора каждого класса `ContactSummary`. Наконец, класс `ContactSummary` передается методу `createQuery()` для указания типа результата.

Чтобы протестировать метод `findAll()`, модифицируем код класса `SpringJPASample` (листинг 8.16).

**Листинг 8.16. Тестирование метода `findAll()`, использующего конструирующее выражение**

```
package com.apress.prospring4.ch8;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringJPASample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();
```

```
        ContactSummaryService contactSummaryService =
            ctx.getBean("contactSummaryService",ContactSummaryService.class);
        List<ContactSummary> contacts = contactSummaryService.findAll();
        for (ContactSummary contactSummary: contacts) {
            System.out.println(contactSummary);
        }
    }
}
```

Запуск на выполнение этой программы сгенерирует вывод для каждого объекта `ContactSummary` внутри списка, как показано ниже:

```
First name: Chris Last Name: Schaefer Home Phone: 1234567890
First name: Scott Last Name: Tiger Home Phone: 1234567890
```

Как видите, конструирующее выражение очень удобно для отображения результата специального запроса на объекты POJO с целью дальнейшей обработки.

## Вставка данных

Вставка данных с применением JPA осуществляется очень просто. Подобно Hibernate, JPA также поддерживает извлечение первичного ключа, генерируемого базой данных. В листинге 8.17 показан модифицированный код с реализацией метода `save()`.

**Листинг 8.17. Реализация метода `save()`**

```
package com.apress.prospring4.ch8;

import org.springframework.stereotype.Service;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    private Log log = LogFactory.getLog(ContactServiceImpl.class);

    @PersistenceContext
    private EntityManager em;

    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAll() {
        List<Contact> contacts = em.createNamedQuery("Contact.findAll",
            Contact.class).getResultList();
        return contacts;
    }
```

```
    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAllWithDetail() {
        List<Contact> contacts = em.createNamedQuery(
            "Contact.findAllWithDetail", Contact.class).getResultList();
        return contacts;
    }

    @Transactional(readOnly=true)
    @Override
    public Contact findById(Long id) {
        TypedQuery<Contact> query = em.createNamedQuery(
            "Contact.findById", Contact.class);
        query.setParameter("id", id);
        return query.getSingleResult();
    }

    @Override
    public Contact save(Contact contact) {
        if (contact.getId() == null) {
            log.info("Inserting new contact");
            em.persist(contact);
        } else {
            em.merge(contact);
            log.info("Updating existing contact");
        }
        log.info("Contact saved with id: " + contact.getId());
        return contact;
    }

    @Override
    public void delete(Contact contact) {
    }
}
```

В листинге 8.17 видно, что метод `save()` сначала по значению `id` проверяет, является ли объект новым экземпляром сущности. Если `id` равно `null` (т.е. идентификатор еще не назначен), то это новый экземпляр сущности, и будет вызван метод `EntityManager.persist()`. Когда вызывается метод `persist()`, диспетчер сущностей (`EntityManager`) сохраняет сущность и делает ее управляемым экземпляром в рамках контекста постоянства. Если значение `id` существует, значит, производится обновление, и тогда будет вызван метод `EntityManager.merge()`. При вызове метода `merge()` диспетчер сущностей объединяет состояние сущности с текущим контекстом постоянства.

В листинге 8.18 приведен модифицированный код для вставки новой записи контакта.

**Листинг 8.18. Тестирование операции вставки**

```
package com.apress.prospring4.ch8;

import java.util.List;
import java.util.Date;
```

```
import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringJPASample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();

      ContactService contactService = ctx.getBean(
          "jpaContactService", ContactService.class);

      Contact contact = new Contact();
      contact.setFirstName("Michael");
      contact.setLastName("Jackson");
      contact.setBirthDate(new Date());

      ContactTelDetail contactTelDetail =
          new ContactTelDetail("Home", "1111111111");

      contact.addContactTelDetail(contactTelDetail);

      contactTelDetail = new ContactTelDetail("Mobile", "2222222222");

      contact.addContactTelDetail(contactTelDetail);

      contactService.save(contact);

      listContactsWithDetail(contactService.findAllWithDetail());
   }
   private static void listContactsWithDetail(List<Contact> contacts) {
      System.out.println("");
      System.out.println("Listing contacts with details:");

      for (Contact contact: contacts) {
         System.out.println(contact);
         if (contact.getContactTelDetails() != null) {
            for (ContactTelDetail contactTelDetail:
               contact.getContactTelDetails()) {
               System.out.println(contactTelDetail);
            }
         }

         if (contact.getHobbies() != null) {
            for (Hobby hobby: contact.getHobbies()) {
               System.out.println(hobby);
            }
         }

         System.out.println();
      }
   }
}
```

Как здесь показано, мы создаем новый контакт, добавляем к нему два телефона и сохраняем объект. После этого мы снова выводим список всех контактов. Запуск программы дает следующий вывод:

```
INFO apress.prospring4.ch8.ContactServiceImpl:  50 - Inserting new contact
Hibernate: insert into contact (ID, BIRTH_DATE, FIRST_NAME, LAST_NAME,
VERSION) values (null, ?, ?, ?, ?)
Hibernate: insert into contact_tel_detail (ID, CONTACT_ID, TEL_NUMBER,
TEL_TYPE, VERSION) values (null, ?, ?, ?, ?)
Hibernate: insert into contact_tel_detail (ID, CONTACT_ID, TEL_NUMBER,
TEL_TYPE, VERSION) values (null, ?, ?, ?, ?)
INFO apress.prospring4.ch8.ContactServiceImpl:  57 - Contact saved with id: 4

Listing contacts with details:
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
Contact Tel Detail - Id: 2, Contact id: 1, Type: Home, Number: 1234567890
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number:
1234567890
Hobby :Movies
Hobby :Swimming

Contact - Id: 4, First name: Michael, Last name: Jackson, Birthday: 2013-08-12
Contact Tel Detail - Id: 4, Contact id: 4, Type: Home, Number: 1111111111
Contact Tel Detail - Id: 5, Contact id: 4, Type: Mobile, Number: 2222222222

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming
```

В журнальной записи INFO видно, что значение id вновь сохраненного контакта установлено корректно. Инфраструктура Hibernate также выводит SQL-операторы, запускаемые в базе данных.

## Обновление данных

Обновление контактов реализуется так же просто, как и вставка данных. Давайте рассмотрим пример. Предположим, что для контакта с идентификатором 1 необходимо обновить имя и удалить запись для домашнего телефона. Чтобы протестировать операцию обновления, модифицируем код класса SpringJPASample (листинг 8.19).

**Листинг 8.19. Тестирование операции обновления**

```
package com.apress.prospring4.ch8;

import java.util.List;
import java.util.Date;
import java.util.Set;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringJPASample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();

      ContactService contactService = ctx.getBean(
          "jpaContactService", ContactService.class);

      Contact contact = contactService.findById(1l);
```

```
        System.out.println("");
        System.out.println("Contact with id 1:" + contact);
        System.out.println("");

        contact.setFirstName("Justin");

        Set<ContactTelDetail> contactTels = contact.getContactTelDetails();
        ContactTelDetail toDeleteContactTel = null;
        for (ContactTelDetail contactTel: contactTels) {
            if (contactTel.getTelType().equals("Home")) {
                toDeleteContactTel = contactTel;
            }
        }
        contactTels.remove(toDeleteContactTel);
        contactService.save(contact);
        listContactsWithDetail(contactService.findAllWithDetail());
    }
    private static void listContactsWithDetail(List<Contact> contacts) {
        System.out.println("");
        System.out.println("Listing contacts with details:");
        for (Contact contact: contacts) {
            System.out.println(contact);
            if (contact.getContactTelDetails() != null) {
                for (ContactTelDetail contactTelDetail:
                    contact.getContactTelDetails()) {
                    System.out.println(contactTelDetail);
                }
            }
            if (contact.getHobbies() != null) {
                for (Hobby hobby: contact.getHobbies()) {
                    System.out.println(hobby);
                }
            }
            System.out.println();
        }
    }
}
```

Сначала мы извлекаем запись с идентификатором 1 и затем изменяем имя. Далее мы проходим в цикле по объектам телефонов, извлекаем объект с типом "Home" и удаляем его из свойства сведений о телефоне для контакта. Наконец, мы вызываем метод `ContactService.save()` снова. В результате запуска этой программы получается следующий вывод:

```
Listing contacts with details:
Contact - Id: 1, First name: Justin, Last name: Schaefer, Birthday: 1981-05-03
Contact Tel Detail - Id: 1, Contact id: 1, Type: Mobile, Number: 1234567890
Hobby :Swimming
Hobby :Movies

Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28

Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming
```

Как видите, имя было обновлено, а домашний телефон удален. Удаление телефона стало возможным по причине определения в ассоциации "один ко многим" атрибута `orphanRemoval=true`, который указывает поставщику JPA (Hibernate) на необходимость удаления всех висячих записей, существующих в базе данных, но больше не принадлежащих какому-либо сохраненному объекту:

```
@OneToMany(mappedBy =
      "contact", cascade=CascadeType.ALL, orphanRemoval=true)
```

## Удаление данных

Удаление данных производится просто. Нужно лишь вызвать метод `EntityManager.remove()` и передать ему объект контакта. В листинге 8.20 показан модифицированный код для удаления контакта.

**Листинг 8.20. Реализация метода `delete()`**

```
package com.apress.prospring4.ch8;

import org.springframework.stereotype.Service;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    private Log log = LogFactory.getLog(ContactServiceImpl.class);

    @PersistenceContext
    private EntityManager em;

    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAll() {
        List<Contact> contacts = em.createNamedQuery("Contact.findAll",
            Contact.class).getResultList();
        return contacts;
    }

    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAllWithDetail() {
        List<Contact> contacts = em.createNamedQuery(
            "Contact.findAllWithDetail", Contact.class).getResultList();
        return contacts;
    }

    @Transactional(readOnly=true)
```

```
    @Override
    public Contact findById(Long id) {
        TypedQuery<Contact> query = em.createNamedQuery(
            "Contact.findById", Contact.class);
        query.setParameter("id", id);
        return query.getSingleResult();
    }
    @Override
    public Contact save(Contact contact) {
        if (contact.getId() == null) {
            log.info("Inserting new contact");
            em.persist(contact);
        } else {
            em.merge(contact);
            log.info("Updating existing contact");
        }
        log.info("Contact saved with id: " + contact.getId());
        return contact;
    }
    @Override
    public void delete(Contact contact) {
      Contact mergedContact = em.merge(contact);
      em.remove(mergedContact);
      log.info("Contact with id: " + contact.getId()  + " deleted successfully");
    }
}
```

Первым делом вызывается метод `EntityManager.merge()` для объединения состояния сущности с текущим контекстом постоянства. Метод `merge()` возвращает управляемый экземпляр сущности. Затем вызывается метод `EntityManager.remove()`, которому передается управляемый экземпляр сущности контакта. Метод `remove()` удаляет запись контакта, а также всю связанную информацию, включая телефоны и хобби, поскольку в отображении было определено `cascade=CascadeType.ALL`. Для тестирования операции удаления внесите в код класса `SpringJPASample` изменения, приведенные в листинге 8.21.

**Листинг 8.21. Тестирование операции удаления**

```
package com.apress.prospring4.ch8;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringJPASample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new
GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();
        ContactService contactService = ctx.getBean(
            "jpaContactService", ContactService.class);
```

```
        Contact contact = contactService.findById(11);
        contactService.delete(contact);

        listContactsWithDetail(contactService.findAllWithDetail());
    }
    private static void listContactsWithDetail(List<Contact> contacts) {
        System.out.println("");
        System.out.println("Listing contacts with details:");
        for (Contact contact: contacts) {
            System.out.println(contact);
            if (contact.getContactTelDetails() != null) {
                for (ContactTelDetail contactTelDetail:
                    contact.getContactTelDetails()) {
                    System.out.println(contactTelDetail);
                }
            }
            if (contact.getHobbies() != null) {
                for (Hobby hobby: contact.getHobbies()) {
                    System.out.println(hobby);
                }
            }
            System.out.println();
        }
    }
}
```

В предыдущем листинге извлекается контакт с идентификатором 1, после чего вызывается метод `delete()` для удаления информации о контакте. Запуск программы дает следующий вывод:

```
Listing contacts with details:
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact Tel Detail - Id: 3, Contact id: 2, Type: Home, Number: 1234567890
Hobby :Swimming
```

Как видите, контакт с идентификатором 1 был удален.

## Использование собственного запроса

Обсудив выполнение элементарных операций базы данных с применением JPA, давайте перейдем к некоторым более сложным темам. Временами требуется полный контроль над запросом, который будет отправлен базе данных. Примером может служить иерархический запрос в базе данных Oracle. Такая разновидность запроса является специфичной для базы данных и называется *собственным запросом* (native query).

В JPA поддерживается выполнение собственных запросов; `EntityManager` отправит такой запрос базе данных в том виде, как он есть, не производя никакого отображения или трансформации. Главным преимуществом применения собственных запросов JPA является отображение результирующего набора на сущностные классы ORM.

В следующих двух разделах будет показано, как использовать собственный запрос для извлечения всех контактов и прямого отображения результирующего набора на объекты Contact.

### Простой собственный запрос

Для демонстрации работы с собственным запросом мы реализуем новый метод, который будет извлекать все контакты из базы данных. В листинге 8.22 показано, что к интерфейсу ContactService добавлен новый метод findAllByNativeQuery().

**Листинг 8.22. Метод findAllByNativeQuery()**

```
package com.apress.prospring4.ch8;
import java.util.List;
public interface ContactService {
    List<Contact> findAll();
    List<Contact> findAllWithDetail();
    Contact findById(Long id);
    Contact save(Contact contact);
    void delete(Contact contact);
    List<Contact> findAllByNativeQuery();
}
```

В листинге 8.23 приведена реализация метода findAllByNativeQuery().

**Листинг 8.23. Реализация метода findAllByNativeQuery()**

```
package com.apress.prospring4.ch8;

import org.springframework.stereotype.Service;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    final static String ALL_CONTACT_NATIVE_QUERY =
        "select id, first_name, last_name, birth_date, version from contact";
    private Log log = LogFactory.getLog(ContactServiceImpl.class);
    @PersistenceContext
    private EntityManager em;
    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAll() {
        List<Contact> contacts = em.createNamedQuery("Contact.findAll",
            Contact.class).getResultList();
        return contacts;
    }
```

```
    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAllWithDetail() {
        List<Contact> contacts = em.createNamedQuery(
            "Contact.findAllWithDetail", Contact.class).getResultList();
        return contacts;
    }

    @Transactional(readOnly=true)
    @Override
    public Contact findById(Long id) {
        TypedQuery<Contact> query = em.createNamedQuery(
            "Contact.findById", Contact.class);
        query.setParameter("id", id);
        return query.getSingleResult();
    }

    @Override
    public Contact save(Contact contact) {
        if (contact.getId() == null) {
            log.info("Inserting new contact");
            em.persist(contact);
        } else {
            em.merge(contact);
            log.info("Updating existing contact");
        }
        log.info("Contact saved with id: " + contact.getId());
        return contact;
    }

    @Override
    public void delete(Contact contact) {
      Contact mergedContact = em.merge(contact);
      em.remove(mergedContact);
      log.info("Contact with id: " + contact.getId()  + " deleted successfully");
    }

    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAllByNativeQuery() {
        return em.createNativeQuery(ALL_CONTACT_NATIVE_QUERY,
            Contact.class).getResultList();
    }
}
```

Как видите, собственный запрос — это всего лишь простой SQL-оператор для извлечения всех столбцов из таблицы `CONTACT`. Чтобы создать и запустить запрос, сначала вызывается метод `EntityManager.createNativeQuery()`, которому передается строка запроса и тип результата. Типом результата должен быть отображенный сущностный класс (в этом случае класс `Contact`). Возвращаемым типом метода `createNativeQuery()` является интерфейс `Query`, который предоставляет операцию `getResultList()` для получения результирующего списка. Поставщик JPA выполнит запрос и преобразует результирующий набор в экземпляры сущностей, основываясь на отображениях JPA, которые были определены в сущностном

классе. Запуск приведенного выше метода даст тот же самый результат, что и запуск метода `findAll()`.

### *Собственный запрос с отображением результирующего набора SQL*

Кроме отображенного объекта предметной области можно передавать строку, указывающую имя отображения результирующего набора SQL. Отображение результирующего набора SQL определяется на уровне сущностного класса за счет применения аннотации `@SqlResultSetMapping`. Отображение результирующего набора SQL может иметь один или более отображений сущностей и столбцов.

Давайте определим простое отображение результирующего набора SQL в сущностном классе `Contact` (листинг 8.24).

**Листинг 8.24. Использование отображения результирующего набора SQL**

```
package com.apress.prospring4.ch8;

import static javax.persistence.GenerationType.IDENTITY;
import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
import javax.persistence.Version;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.OneToMany;
import javax.persistence.ManyToMany;
import javax.persistence.JoinTable;
import javax.persistence.JoinColumn;
import javax.persistence.CascadeType;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.SqlResultSetMapping;
import javax.persistence.EntityResult;
@Entity
@Table(name = "contact")
@NamedQueries({
   @NamedQuery(name="Contact.findAll", query="select c from Contact c"),
   @NamedQuery(name="Contact.findById",
      query="select distinct c from Contact c left join fetch
c.contactTelDetails t left join fetch c.hobbies h where c.id = :id"),
   @NamedQuery(name="Contact.findAllWithDetail",
      query="select distinct c from Contact c left join fetch
c.contactTelDetails t left join fetch c.hobbies h")
})
@SqlResultSetMapping(
    name="contactResult",
    entities=@EntityResult(entityClass=Contact.class)
)
```

```
public class Contact implements Serializable {
   private Long id;
   private int version;
   private String firstName;
   private String lastName;
   private Date birthDate;
   private Set<ContactTelDetail> contactTelDetails =
     new HashSet<ContactTelDetail>();
   private Set<Hobby> hobbies = new HashSet<Hobby>();
   @Id
   @GeneratedValue(strategy = IDENTITY)
   @Column(name = "ID")
   public Long getId() {
      return this.id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   @Version
   @Column(name = "VERSION")
   public int getVersion() {
      return this.version;
   }
   public void setVersion(int version) {
      this.version = version;
   }
   @Column(name = "FIRST_NAME")
   public String getFirstName() {
      return this.firstName;
   }
   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }
   @Column(name = "LAST_NAME")
   public String getLastName() {
      return this.lastName;
   }
   public void setLastName(String lastName) {
      this.lastName = lastName;
   }
   @Temporal(TemporalType.DATE)
   @Column(name = "BIRTH_DATE")
   public Date getBirthDate() {
      return this.birthDate;
   }
   public void setBirthDate(Date birthDate) {
      this.birthDate = birthDate;
   }
   @OneToMany(mappedBy = "contact", cascade=CascadeType.ALL,
      orphanRemoval=true)
   public Set<ContactTelDetail> getContactTelDetails() {
      return this.contactTelDetails;
   }
```

```
    public void setContactTelDetails(Set<ContactTelDetail> contactTelDetails) {
     this.contactTelDetails = contactTelDetails;
    }
    public void addContactTelDetail(ContactTelDetail contactTelDetail) {
       contactTelDetail.setContact(this);
       getContactTelDetails().add(contactTelDetail);
    }
    public void removeContactTelDetail(ContactTelDetail contactTelDetail) {
       getContactTelDetails().remove(contactTelDetail);
    }
    @ManyToMany
    @JoinTable(name = "contact_hobby_detail",
       joinColumns = @JoinColumn(name = "CONTACT_ID"),
       inverseJoinColumns = @JoinColumn(name = "HOBBY_ID"))
    public Set<Hobby> getHobbies() {
       return this.hobbies;
    }
    public void setHobbies(Set<Hobby> hobbies) {
       this.hobbies = hobbies;
    }
    @Override
    public String toString() {
       return "Contact - Id: " + id + ", First name: " + firstName
          + ", Last name: " + lastName + ", Birthday: " + birthDate;
    }
}
```

Для сущностного класса определено отображение результирующего набора SQL по имени `contactResult` с указанием в атрибуте `entityClass` самого класса `Contact`. В JPA поддерживается более сложное отображение для множества сущностей, а также отображение вплоть до уровня отдельных столбцов.

После определения отображения результирующего набора SQL метод `findAllByNativeQuery()` может вызываться с применением имени этого отображения. В листинге 8.25 показан модифицированный код класса `ContactServiceImpl`.

**Листинг 8.25. Реализация метода `findAllByNativeQuery()` с использованием отображения результирующего набора SQL**

```
package com.apress.prospring4.ch8;

import org.springframework.stereotype.Service;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

@Service("jpaContactService")
@Repository
@Transactional
```

```
public class ContactServiceImpl implements ContactService {
    final static String ALL_CONTACT_NATIVE_QUERY =
        "select id, first_name, last_name, birth_date, version from contact";
    private Log log = LogFactory.getLog(ContactServiceImpl.class);
    @PersistenceContext
    private EntityManager em;
    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAll() {
        List<Contact> contacts = em.createNamedQuery("Contact.findAll",
            Contact.class).getResultList();
        return contacts;
    }
    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAllWithDetail() {
        List<Contact> contacts = em.createNamedQuery(
            "Contact.findAllWithDetail", Contact.class).getResultList();
        return contacts;
    }
    @Transactional(readOnly=true)
    @Override
    public Contact findById(Long id) {
        TypedQuery<Contact> query = em.createNamedQuery(
            "Contact.findById", Contact.class);
        query.setParameter("id", id);
        return query.getSingleResult();
    }
    @Override
    public Contact save(Contact contact) {
        if (contact.getId() == null) {
            log.info("Inserting new contact");
            em.persist(contact);
        } else {
            em.merge(contact);
            log.info("Updating existing contact");
        }
        log.info("Contact saved with id: " + contact.getId());
        return contact;
    }
    @Override
    public void delete(Contact contact) {
        Contact mergedContact = em.merge(contact);
        em.remove(mergedContact);
        log.info("Contact with id: " + contact.getId()  + " deleted successfully");
    }
    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAllByNativeQuery() {
        return em.createNativeQuery(ALL_CONTACT_NATIVE_QUERY,
            "contactResult").getResultList();
    }
}
```

Как видите, JPA также поддерживает запуск собственных запросов с гибкими возможностями отображения результирующих наборов SQL.

## Использование API-интерфейса критериев JPA 2 для запроса с критерием

Большинство приложений предоставляют пользовательские интерфейсы для поиска информации. Обычно они отображают большое количество поисковых полей, но пользователи будут вводить информацию только в некоторых из них и затем запускать поиск. Очень трудно подготовить запросы для каждой комбинации параметров, которые пользователи могут вводить, ввиду слишком большого их числа. В такой ситуации на помощь приходит средство запросов с API-интерфейсом критериев.

В JPA 2 крупным нововведением стала функциональность запросов, использующих строго типизированный API-интерфейс критериев (Criteria API). В этом новом Criteria API передаваемый в запрос критерий основан на метамодели отображенных сущностных классов. В результате каждый указанный критерий является строго типизированным, и ошибки обнаруживаются на этапе компиляции, а не во время выполнения.

В API-интерфейсе критериев JPA метамодель сущностного класса представляется именем сущностного класса с суффиксом в виде символа подчеркивания (_). Например, класс метамодели для сущностного класса `Contact` имеет имя `Contact_`. Код класса `Contact_` приведен в листинге 8.26.

**Листинг 8.26. Строго типизированный API-интерфейс критериев JPA 2: метамодель**

```
package com.apress.prospring4.ch8;

import java.util.Date;
import javax.persistence.metamodel.SetAttribute;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.StaticMetamodel;

@StaticMetamodel(Contact.class)
public abstract class Contact_ {
    public static volatile SingularAttribute<Contact, Long> id;
    public static volatile SetAttribute<Contact, ContactTelDetail>
        contactTelDetails;
    public static volatile SingularAttribute<Contact, String> lastName;
    public static volatile SingularAttribute<Contact, Date> birthDate;
    public static volatile SetAttribute<Contact, Hobby> hobbies;
    public static volatile SingularAttribute<Contact, String> firstName;
    public static volatile SingularAttribute<Contact, Integer> version;
}
```

Класс метамодели снабжен аннотацией `@StaticMetamodel` с указанием в атрибуте отображенного сущностного класса. Внутри класса присутствуют объявления всех атрибутов и связанных с ними типов.

Кодирование и сопровождение таких классов метамоделей были бы довольно утомительными. К счастью, доступны инструменты, которые генерируют клас-

сы метамоделей автоматически на основе отображений JPA в рамках сущностных классов. Инфраструктура Hibernate предлагает такой инструмент, который называется Hibernate Metamodel Generator (генератор метамоделей Hibernate; `http://www.hibernate.org/subprojects/jpamodelgen.html`).

Способ генерации классов метамоделей зависит от того, какой инструмент применяется для разработки и построения проекта. Мы рекомендуем ознакомиться с главой "Usage" ("Использование") документации по Hibernate (`http://docs.jboss.org/hibernate/jpamodelgen/1.3/reference/en-US/html_single/#chapter-usage`), в которой приводятся соответствующие детали. В сопровождающих книгу примерах кода для генерации классов метамоделей применяется Maven. Это требует зависимости для генерации классов метамоделей, описанной в табл. 8.2.

**Таблица 8.2. Зависимость для генерации классов метамоделей**

| JAR-файл | Описание |
|---|---|
| `hibernate-jpamodelgen-1.3.0.Final.jar` | Главная библиотека для генерации классов метамоделей. Вы найдете нужный файл в `hibernate-jpamodelgen-1.3.0.Final` после распаковки загруженного пакета или посредством инструмента построения, такого как Maven |

Располагая стратегией генерации классов метамоделей, давайте определим запрос, который принимает имя и фамилию для поиска контактов. В листинге 8.27 приведено определение нового метода `findByCriteriaQuery()` в интерфейсе `ContactService`.

**Листинг 8.27. Метод `findByCriteriaQuery()`**

```
package com.apress.prospring4.ch8;
import java.util.List;
public interface ContactService {
    List<Contact> findAll();
    List<Contact> findAllWithDetail();
    Contact findById(Long id);
    Contact save(Contact contact);
    void delete(Contact contact);
    List<Contact> findAllByNativeQuery();
    List<Contact> findByCriteriaQuery(String firstName, String lastName);
}
```

В листинге 8.28 показана реализация метода `findByCriteriaQuery()`, использующая запрос с API-интерфейсом критериев JPA 2.

**Листинг 8.28. Реализация метода `findByCriteriaQuery()`**

```
package com.apress.prospring4.ch8;

import org.springframework.stereotype.Service;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;
```

```
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;
import javax.persistence.criteria.JoinType;
import javax.persistence.criteria.Predicate;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

@Service("jpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    final static String ALL_CONTACT_NATIVE_QUERY =
        "select id, first_name, last_name, birth_date, version from contact";

    private Log log = LogFactory.getLog(ContactServiceImpl.class);

    @PersistenceContext
    private EntityManager em;

    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAll() {
        List<Contact> contacts = em.createNamedQuery("Contact.findAll",
            Contact.class).getResultList();
        return contacts;
    }
    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAllWithDetail() {
        List<Contact> contacts = em.createNamedQuery(
            "Contact.findAllWithDetail", Contact.class).getResultList();
        return contacts;
    }

    @Transactional(readOnly=true)
    @Override
    public Contact findById(Long id) {
        TypedQuery<Contact> query = em.createNamedQuery(
            "Contact.findById", Contact.class);
        query.setParameter("id", id);

        return query.getSingleResult();
    }

    @Override
    public Contact save(Contact contact) {
        if (contact.getId() == null) {
            log.info("Inserting new contact");
            em.persist(contact);
        } else {
            em.merge(contact);
            log.info("Updating existing contact");
        }
```

```
        log.info("Contact saved with id: " + contact.getId());
        return contact;
    }
    @Override
    public void delete(Contact contact) {
        Contact mergedContact = em.merge(contact);
        em.remove(mergedContact);
        log.info("Contact with id: " + contact.getId()  + " deleted
successfully");
    }
    @Transactional(readOnly=true)
    @Override
    public List<Contact> findAllByNativeQuery() {
        return em.createNativeQuery(ALL_CONTACT_NATIVE_QUERY,
            "contactResult").getResultList();
    }
    @Transactional(readOnly=true)
    @Override
    public List<Contact> findByCriteriaQuery(String firstName, String lastName) {
        log.info("Finding contact for firstName: " + firstName
                + " and lastName: " + lastName);
        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<Contact> criteriaQuery = cb.createQuery(Contact.class);
        Root<Contact> contactRoot = criteriaQuery.from(Contact.class);
        contactRoot.fetch(Contact_.contactTelDetails, JoinType.LEFT);
        contactRoot.fetch(Contact_.hobbies, JoinType.LEFT);
        criteriaQuery.select(contactRoot).distinct(true);
        Predicate criteria = cb.conjunction();
        if (firstName != null) {
            Predicate p = cb.equal(contactRoot.get(Contact_.firstName),
                firstName);
             criteria = cb.and(criteria, p);
        }
        if (lastName != null) {
            Predicate p = cb.equal(contactRoot.get(Contact_.lastName),
                lastName);
            criteria = cb.and(criteria, p);
        }
        criteriaQuery.where(criteria);
        return em.createQuery(criteriaQuery).getResultList();
    }
}
```

Рассмотрим основные моменты, связанные с листингом 8.28.

- Для извлечения экземпляра `CriteriaBuilder` вызывается метод `EntityManager.getCriteriaBuilder()`.
- С использованием метода `CriteriaBuilder.createQuery()`, которому передается `Contact` в качестве результирующего типа, создается типизированный запрос.

- Вызывается метод `CriteriaQuery.from()` с передачей ему сущностного класса. Результатом будет объект корня запроса (интерфейс `Root<Contact>`), соответствующий указанной сущности. Объект корня запроса формирует основу для путевых выражений внутри запроса.
- Два вызова метода `Root.fetch()` обеспечивают незамедлительную выборку ассоциаций, относящихся к телефонам и хобби. Аргумент `JoinType.LEFT` задает внешнее соединение. Вызов метода `Root.fetch()` со значением `JoinType.LEFT` во втором аргументе эквивалентен указанию операции соединения `left join fetch` в JPQL.
- Вызывается метод `CriteriaQuery.select()` с передачей ему объекта корня запроса в качестве результирующего типа. Вызов метода `distinct()` со значением `true` означает, что дублированные записи должны быть устранены.
- С помощью вызова метода `CriteriaBuilder.conjunction()` получается экземпляр `Predicate`; этот вызов означает, что было сделано объединение одного или более ограничений. Экземпляр `Predicate` может быть как простым, так и сложным предикатом; здесь предикат — это ограничение, которое указывает критерий выборки, определенный выражением.
- Производится проверка аргументов имени и фамилии. Если аргумент не равен `null`, создается новый экземпляр `Predicate` с применением метода интерфейса `CriteriaBuilder` (т.е. метода `CriteriaBuilder.and()`). Метод `equal()` предназначен для указания ограничения равенства; внутри него вызывается `Root.get()` с передачей соответствующего атрибута метамодели сущностного класса, к которому будет применяться ограничение. Сконструированный предикат затем объединяется с существующим предикатом (хранящимся в переменной `criteria`) с помощью вызова метода `CriteriaBuilder.and()`.
- Экземпляр `Predicate` создается со всеми критериями и ограничениями, после чего передается в виде конструкции `where` запросу посредством вызова метода `CriteriaQuery.where()`.
- Наконец, `CriteriaQuery` передается в `EntityManager`. Затем `EntityManager` конструирует запрос на основе переданного `CriteriaQuery`, выполняет его и возвращает результат.

Чтобы протестировать запрос с критерием, модифицируем код класса `SpringJPASample`, как показано в листинге 8.29.

**Листинг 8.29. Тестирование метода `findByCriteriaQuery()`**

```
package com.apress.prospring4.ch8;

import java.util.List;
import java.util.Date;
import java.util.Set;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringJPASample {
   public static void main(String[] args) {
     GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
     ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
```

```
        ctx.refresh();
        ContactService contactService = ctx.getBean(
                "jpaContactService", ContactService.class);

        List<Contact> contacts = contactService.findByCriteriaQuery("John", "Smith");
        listContactsWithDetail(contacts);
    }
    private static void listContactsWithDetail(List<Contact> contacts) {
        System.out.println("");
        System.out.println("Listing contacts with details:");
        for (Contact contact: contacts) {
            System.out.println(contact);
            if (contact.getContactTelDetails() != null) {
                for (ContactTelDetail contactTelDetail:
                        contact.getContactTelDetails()) {
                    System.out.println(contactTelDetail);
                }
            }
            if (contact.getHobbies() != null) {
                for (Hobby hobby: contact.getHobbies()) {
                    System.out.println(hobby);
                }
            }
            System.out.println();
        }
    }
}
```

Запуск этой программы дает следующий вывод (который показан не полностью):

```
INFO apress.prospring4.ch8.ContactServiceImpl:  91 - Finding contact for
firstName: John and lastName: Smith
...
Listing contacts with details:
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

Можете испробовать различные комбинации аргументов или передать значение `null` в любом из аргументов и понаблюдать за выводом.

# Введение в проект Spring Data JPA

Проект Spring Data JPA является подпроектом в рамках проекта Spring Data. Главное назначение Spring Data JPA состоит в предоставлении дополнительных возможностей для упрощения разработки приложений с помощью JPA.

Проект Spring Data JPA предлагает множество основных средств. Два из них мы обсудим в этом разделе. Первое средство — это абстракция `Repository`, а второе — прослушиватель сущностей, предназначенный для отслеживания базовой информации аудита по сущностным классам.

## Добавление библиотечных зависимостей Spring Data JPA

Для использования Spring Data JPA понадобится добавить к проекту зависимости, информация о которых приведена в табл. 8.3.

**Таблица 8.3. Зависимости Maven для Spring Data JPA**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.springframework.data` | `spring-data-jpa` | 1.3.0.RELEASE | Библиотека Spring Data JPA |
| `com.google.guava` | `guava` | 14.0.1 | Полезные вспомогательные классы для поддержки коллекций и т.д. |
| `org.springframework` | `spring-aop` | 4.0.2.RELEASE | Пакет Spring AOP для поддержки аспектно-ориентированного программирования |

## Использование абстракции Repository из проекта Spring Data JPA для выполнения операций базы данных

Одной из главных концепций проекта Spring Data и всех его подпроектов является абстракция `Repository` (репозиторий), которая принадлежит проекту Spring Data Commons (`https://github.com/spring-projects/spring-data-commons`). На время написания этой книги актуальной была версия 1.5.2.RELEASE. В Spring Data JPA абстракция репозитория представляет собой оболочку вокруг JPA-интерфейса `EntityManager` и предлагает более простой интерфейс для доступа к данным на основе JPA. Центральным интерфейсом в рамках Spring Data является `org.springframework.data.repository.Repository<T,ID extends Serializable>`, который представляет собой маркерный интерфейс, принадлежащий дистрибутиву Spring Data Commons. В проекте Spring Data предоставляются различные расширения интерфейса `Repository`; одно из них — это интерфейс `org.springframework.data.repository.CrudRepository` (также относящийся к проекту Spring Data Commons), который мы обсудим в настоящем разделе.

Интерфейс `CrudRepository` предлагает несколько часто используемых методов. В листинге 8.30 показано объявление этого интерфейса, взятое из исходного кода проекта Spring Data Commons.

**Листинг 8.30. Интерфейс `CrudRepository`**

```
package org.springframework.data.repository;

import java.io.Serializable;

@NoRepositoryBean
public interface CrudRepository<T, ID extends Serializable> extends
Repository<T, ID> {
   long count();
   void delete(ID id);
```

```
    void delete(Iterable<? extends T> entities);
    void delete(T entity);
    void deleteAll();
    boolean exists(ID id);
    Iterable<T> findAll();
    T findOne(ID id);
    Iterable<T> save(Iterable<? extends T> entities);
    T save(T entity);
}
```

Хотя имена методов самоочевидны, работу абстракции `Repository` лучше исследовать на простом примере. Давайте немного переделаем интерфейс `ContactService`, в частности, его три метода поиска. Модифицированный интерфейс `ContactService` показан в листинге 8.31. После такого изменения интерфейса `ContactService` в классах `SpringJpaSample` и `ContactServiceImpl` появятся ошибки. Просто удалите эти два класса и продолжайте реализацию интерфейса с использованием Spring Data JPA.

**Листинг 8.31. Пересмотренный интерфейс `ContactService`**

```
package com.apress.prospring4.ch8;

import java.util.List;

public interface ContactService {
  List<Contact> findAll();
  List<Contact> findByFirstName(String firstName);
  List<Contact> findByFirstNameAndLastName(String firstName, String lastName);
}
```

Следующий шаг заключается в подготовке интерфейса `ContactRepository`, который расширяет интерфейс `CrudRepository`. Интерфейс `ContactRepository` представлен в листинге 8.32.

**Листинг 8.32. Интерфейс `ContactRepository`**

```
package com.apress.prospring4.ch8;

import java.util.List;

import org.springframework.data.repository.CrudRepository;

public interface ContactRepository extends CrudRepository<Contact, Long> {
  List<Contact> findByFirstName(String firstName);
  List<Contact> findByFirstNameAndLastName(String firstName, String lastName);
}
```

В этом интерфейсе нам необходимо просто объявить два метода, т.к. метод `findAll()` уже доступен как `CrudRepository.findAll()`. В листинге 8.32 видно, что интерфейс `ContactRepository` расширяет интерфейс `CrudRepository`, передавая ему сущностный класс (`Contact`) и тип идентификатора (`Long`). Один причудливый аспект абстракции `Repository` из Spring Data состоит в том, что если соблюдать общее соглашение об именовании, такое как `findByFirstName` и `findByFirstNameAndLastName`, то предоставлять Spring Data JPA именованный

запрос не понадобится. Вместо этого Spring Data JPA самостоятельно "выведет" и сконструирует запрос на основе имени метода.

Например, для метода `findByFirstName()` будет автоматически подготовлен запрос `select c from Contact c where c.firstName = :firstName` и установлен именованный параметр `firstName` из аргумента.

Чтобы пользоваться абстракцией `Repository`, необходимо определить ее в конфигурации Spring. В листинге 8.33 приведен конфигурационный файл (`app-context-annotation.xml`).

**Листинг 8.33. Конфигурация репозитория JPA в Spring**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
      http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:META-INF/sql/schema.sql"/>
      <jdbc:script location="classpath:META-INF/sql/test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <tx:annotation-driven transaction-manager="transactionManager" />

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
        />
      </property>
      <property name="packagesToScan"
          value="com.apress.prospring4.ch8"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.H2Dialect
            </prop>
```

```
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>
<context:component-scan base-package="com.apress.prospring4.ch8"/>
<jpa:repositories base-package="com.apress.prospring4.ch8"
                  entity-manager-factory-ref="emf"
                  transaction-manager-ref="transactionManager"/>
</beans>
```

Сначала в конфигурационный файл должно быть добавлено пространство имен `jpa`. Затем с помощью дескриптора `<jpa:repositories>` конфигурируется абстракция `Repository` из Spring Data JPA. Мы указываем Spring на необходимость сканирования пакета `com.apress.prospring4.ch8` на предмет интерфейсов репозитория, а также передачи `EntityManagerFactory` и диспетчера транзакций соответственно.

В листинге 8.34 показана реализация трех методов поиска из интерфейса `ContactService`.

**Листинг 8.34. Класс `ContactServiceImpl`**

```
package com.apress.prospring4.ch8;

import org.springframework.stereotype.Service;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import org.springframework.beans.factory.annotation.Autowired;

import java.util.List;
import java.util.Lists;

@Service("springJpaContactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    @Autowired
    private ContactRepository contactRepository;
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return Lists.newArrayList(contactRepository.findAll());
    }
    @Transactional(readOnly=true)
    public List<Contact> findByFirstName(String firstName) {
        return contactRepository.findByFirstName(firstName);
    }
    @Transactional(readOnly=true)
    public List<Contact> findByFirstNameAndLastName(
      String firstName, String lastName) {
      return contactRepository.findByFirstNameAndLastName(firstName, lastName);
    }
}
```

Как видите, вместо EntityManager в класс службы мы должны внедрить интерфейс ContactRepository, и Spring Data JPA самостоятельно сделает всю низкоуровневую работу. В листинге 8.35 приведена модифицированная тестовая программа.

**Листинг 8.35. Класс SpringJpaSample**

```
package com.apress.prospring4.ch8;

import java.util.List;
import java.util.Date;
import java.util.Set;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringJPASample {
   public static void main(String[] args) {
     GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
     ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
     ctx.refresh();

     ContactService contactService = ctx.getBean(
          "springJpaContactService", ContactService.class);

     listContacts("Find all:", contactService.findAll());
     listContacts("Find by first name:", contactService.findByFirstName("Chris"));
     listContacts("Find by first and last name:",
          contactService.findByFirstNameAndLastName("Chris", "Schaefer"));
   }

   private static void listContacts(String message, List<Contact> contacts)
{
      System.out.println("");
      System.out.println(message);
      for (Contact contact: contacts) {
         System.out.println(contact);
         System.out.println();
      }
   }
}
```

Запуск программы, как и ожидалось, приведет к выводу списка контактов.

Итак, вы увидели, каким образом Spring Data JPA помогает упростить разработку. Мы не должны создавать именованный запрос, вызывать метод EntityManager.createQuery() и т.д.

Мы рассмотрели только простейшие примеры. Абстракция Repository поддерживает множество функциональных возможностей, включая определение специальных запросов с помощью аннотации @Query, средство Specification (принадлежащее версии 1.4.4.RELEASE проекта Spring Data JPA) и т.п. За дополнительными сведениями обращайтесь в документацию SpringSource:

- Проект Spring Data Commons: http://docs.spring.io/spring-data/data-commons/docs/current/reference/html/
- Проект Spring Data JPA: http://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/

## Отслеживание изменений в сущностном классе

В большинстве приложений требуется отслеживать основную информацию аудита для бизнес-данных, поддерживаемых пользователями. Информация аудита обычно включает имя пользователя, создавшего порцию данных, дату создания, дату последней модификации, а также имя пользователя, который провел эту последнюю модификацию.

Проект Spring Data JPA предлагает такую функцию в форме прослушивателя сущностей JPA, который помогает отслеживать информацию аудита автоматически. Чтобы пользоваться упомянутым средством, сущностный класс должен реализовывать интерфейс `org.springframework.data.domain.Auditable<U, ID extends Serializable>` (принадлежащий Spring Data Commons) или расширять любой класс, который реализует этот интерфейс. В листинге 8.36 показан код интерфейса `Auditable`, взятый из документации по Spring Data.

**Листинг 8.36. Интерфейс `Auditable`**

```
package org.springframework.data.domain;
import java.io.Serializable;
import org.joda.time.DateTime;
public interface Auditable<U, ID extends Serializable>
   extends Persistable<ID> {
   U getCreatedBy();
   void setCreatedBy(final U createdBy);
   DateTime getCreatedDate();
   void setCreated(final DateTime creationDate);
   U getLastModifiedBy();
   void setLastModifiedBy(final U lastModifiedBy);
   DateTime getLastModifiedDate();
   void setLastModified(final DateTime lastModifiedDate);
}
```

Чтобы посмотреть, как это работает, давайте создадим в схеме базы данных новую таблицу по имени `CONTACT_AUDIT`, основанную на таблице `CONTACT`, в которую добавлены четыре столбца, связанные с аудитом. В листинге 8.37 приведен сценарий создания таблицы (`schema.sql`).

**Листинг 8.37. Таблица `CONTACT_AUDIT`**

```
CREATE TABLE CONTACT_AUDIT (
   ID INT NOT NULL AUTO_INCREMENT
   , FIRST_NAME VARCHAR(60) NOT NULL
   , LAST_NAME VARCHAR(40) NOT NULL
   , BIRTH_DATE DATE
   , VERSION INT NOT NULL DEFAULT 0
   , CREATED_BY VARCHAR(20)
   , CREATED_DATE TIMESTAMP
   , LAST_MODIFIED_BY VARCHAR(20)
   , LAST_MODIFIED_DATE TIMESTAMP
   , UNIQUE UQ_CONTACT_AUDIT_1 (FIRST_NAME, LAST_NAME)
   , PRIMARY KEY (ID)
);
```

Полужирным выделены четыре столбца, связанные с аудитом. Далее создается сущностный класс по имени `ContactAudit`, код которого представлен в листинге 8.38.

**Листинг 8.38. Класс `ContactAudit`**

```
package com.apress.prospring4.ch8;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
import javax.persistence.Version;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.Transient;
import org.hibernate.annotations.Type;
import org.springframework.data.domain.Auditable;
import org.joda.time.DateTime;

@Entity
@Table(name = "contact_audit")
public class ContactAudit implements Auditable<String, Long>, Serializable {
    private Long id;
    private int version;
    private String firstName;
    private String lastName;
    private Date birthDate;
    private String createdBy;
    private DateTime createdDate;
    private String lastModifiedBy;
    private DateTime lastModifiedDate;

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Version
    @Column(name = "VERSION")
    public int getVersion() {
        return this.version;
    }

    public void setVersion(int version) {
        this.version = version;
    }
```

```
@Column(name = "FIRST_NAME")
public String getFirstName() {
   return this.firstName;
}

public void setFirstName(String firstName) {
   this.firstName = firstName;
}

@Column(name = "LAST_NAME")
public String getLastName() {
   return this.lastName;
}

public void setLastName(String lastName) {
   this.lastName = lastName;
}

@Temporal(TemporalType.DATE)
@Column(name = "BIRTH_DATE")
public Date getBirthDate() {
   return this.birthDate;
}

public void setBirthDate(Date birthDate) {
   this.birthDate = birthDate;
}

@Column(name="CREATED_BY")
public String getCreatedBy() {
   return createdBy;
}

public void setCreatedBy(String createdBy) {
   this.createdBy = createdBy;
}

@Column(name="CREATED_DATE")
@Type(type="org.jadira.usertype.dateandtime.joda.PersistentDateTime")
public DateTime getCreatedDate() {
   return createdDate;
}

public void setCreatedDate(DateTime createdDate) {
   this.createdDate = createdDate;
}

@Column(name="LAST_MODIFIED_BY")
public String getLastModifiedBy() {
   return lastModifiedBy;
}

public void setLastModifiedBy(String lastModifiedBy) {
   this.lastModifiedBy = lastModifiedBy;
}

@Column(name="LAST_MODIFIED_DATE")
@Type(type="org.jadira.usertype.dateandtime.joda.PersistentDateTime")
public DateTime getLastModifiedDate() {
   return lastModifiedDate;
}
```

```
    public void setLastModifiedDate(DateTime lastModifiedDate) {
        this.lastModifiedDate = lastModifiedDate;
    }

    @Transient
    public boolean isNew() {
        if (id == null) {
            return true;
        } else {
            return false;
        }
    }

    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate
            + ", Create by: " + createdBy + ", Create date: " + createdDate
            + ", Modified by: " + lastModifiedBy + ", Modified date: "
            + lastModifiedDate;
    }
}
```

Сущностный класс `ContactAudit` реализует интерфейс `Auditable` и его методы путем отображения четырех столбцов аудита. Для их отображения на действительные столбцы в таблице применяются аннотации `@Column`. Для двух атрибутов даты (`createdDate` и `lastModifiedDate`) применяется аннотация специального пользовательского типа Hibernate по имени `@Type` с указанием класса реализации `org.joda.time.contrib.hibernate.PersistentDateTime`. Подобно тому, как интерфейс `Auditable` из Spring Data JPA использует тип `DateTime` из библиотеки Joda-Time, библиотека `joda-time-hibernate` предоставляет этот специальный пользовательский тип для применения с Hibernate при сохранении атрибута в столбце `TIMESTAMP` базы данных. Метод `isNew()` интерфейса `Auditable` (унаследованного от интерфейса `org.springframework.data.domain.Persistable<ID extends Serializable>`) также реализован. Аннотация `@Transient` означает, что поле не нуждается в сохранении. Данная функция используется Spring Data JPA для идентификации, является ли это новой сущностью, при определении необходимости установки атрибутов `createdBy` и `createdDate`. В реализации мы просто проверяем идентификатор, и если его значение равно `null`, то возвращаем `true`, указывая на новый экземпляр сущности.

В листинге 8.39 показан интерфейс `ContactAuditService`, в котором мы определяем лишь несколько методов для демонстрации возможностей аудита.

**Листинг 8.39. Интерфейс `ContactAuditService`**

```
package com.apress.prospring4.ch8;

import java.util.List;

public interface ContactAuditService {
    List<ContactAudit> findAll();
    ContactAudit findById(Long id);
    ContactAudit save(ContactAudit contact);
}
```

Следующий шаг заключается в создании интерфейса `ContactAuditRepository`, который приведен в листинге 8.40.

**Листинг 8.40. Интерфейс `ContactAuditRepository`**

```
package com.apress.prospring4.ch8;

import org.springframework.data.repository.CrudRepository;

public interface ContactAuditRepository extends CrudRepository<ContactAudit, Long>
{
}
```

Интерфейс `ContactAuditRepository` просто расширяет интерфейс `CrudRepository`, располагающий реализациями всех методов, которые мы собираемся использовать для `ContactAuditService`. Метод `findById()` реализован посредством метода `CrudRepository.findOne()`.

В листинге 8.41 представлен класс реализации `ContactAuditServiceImpl`.

**Листинг 8.41. Класс `ContactAuditServiceImpl`**

```
package com.apress.prospring4.ch8;

import org.springframework.stereotype.Service;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.beans.factory.annotation.Autowired;

import java.util.List;
import com.google.common.collect.Lists;

@Service("contactAuditService")
@Repository
@Transactional
public class ContactAuditServiceImpl implements ContactAuditService {
    @Autowired
    private ContactAuditRepository contactAuditRepository;

    @Transactional(readOnly=true)
    public List<ContactAudit> findAll() {
        return Lists.newArrayList(contactAuditRepository.findAll());
    }
    public ContactAudit findById(Long id) {
        return contactAuditRepository.findOne(id);
    }
    public ContactAudit save(ContactAudit contact) {
        return contactAuditRepository.save(contact);
    }
}
```

Нам также необходимо выполнить определенную работу по конфигурированию. Первая задача связана с объявлением `AuditingEntityListener<T>` — прослушивателя сущностей JPA, который предоставляет службу аудита. Для объявления прослушивателя создайте файл по имени `/src/main/resources/META-INF/orm.xml` (использовать такое имя файла обязательно, и это указано в спецификации JPA) в корневой папке проекта и объявите прослушиватель, как показано в листинге 8.42.

**Листинг 8.42. Объявление прослушивателя сущностей**

```
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
      http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
      version="2.0">
   <description>JPA</description>
   <persistence-unit-metadata>
      <persistence-unit-defaults>
         <entity-listeners>
            <entity-listener
class="org.springframework.data.jpa.domain.support.AuditingEntityListener" />
      </entity-listeners>
      </persistence-unit-defaults>
   </persistence-unit-metadata>
</entity-mappings>
```

Поставщик JPA выберет этот прослушиватель во время выполнения операций, связанных с постоянством (события сохранения и обновления), для обработки полей аудита.

Мы также должны определить прослушиватель в конфигурации Spring. Требуемая конфигурация показана в листинге 8.43 (`app-context-annotation.xml`).

**Листинг 8.43. Объявление прослушивателя сущностей в Spring**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
      http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx.xsd">
   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:META-INF/sql/schema.sql"/>
      <jdbc:script location="classpath:META-INF/sql/test-data.sql"/>
   </jdbc:embedded-database>
   <bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>
```

```
  <tx:annotation-driven transaction-manager="transactionManager" />

  <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
        />
      </property>
      <property name="packagesToScan"
          value="com.apress.prospring4.ch8"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
         </props>
      </property>
  </bean>

   <context:component-scan base-package="com.apress.prospring4.ch8"/>

   <jpa:repositories base-package="com.apress.prospring4.ch8"
                  entity-manager-factory-ref="emf"
                  transaction-manager-ref="transactionManager"/>

  <jpa:auditing auditor-aware-ref="auditorAwareBean"/>

  <bean id="auditorAwareBean" class="com.apress.prospring4.ch8.AuditorAwareBean"/>
</beans>
```

Дескриптор `<jpa:auditing>` предназначен для включения функции аудита Spring Data JPA, а `auditorAwareBean` — это бин, предоставляющий пользовательскую информацию. В листинге 8.44 приведен класс `AuditorAwareBean`.

**Листинг 8.44. Класс `AuditorAwareBean`**

```
package com.apress.prospring4.ch8;

import org.springframework.data.domain.AuditorAware;

public class AuditorAwareBean implements AuditorAware<String> {
   public String getCurrentAuditor() {
      return "prospring4";
   }
}
```

Класс `AuditorAwareBean` реализует интерфейс `AuditorAware<T>`, передавая ему тип `String`. В реальных ситуациях это должен быть экземпляр класса с пользовательской информацией, например, класс `User`, который представляет вошедшего в систему пользователя, выполнившего действие обновления данных. Мы используем здесь `String` только ради простоты. В классе `AuditorAwareBean` реализован

метод `getCurrentAuditor()`, и значение жестко закодировано как `prospring4`. В реальности сведения о пользователе должны извлекаться из лежащей в основе инфраструктуры безопасности. Например, в Spring Security информация о пользователе может быть получена из класса `SecurityContextHolder`.

На этом вся работа по реализации завершена. В листинге 8.45 показана тестовая программа `SpringJPASample`.

**Листинг 8.45. Тестирование функции аудита Spring Data JPA**

```
package com.apress.prospring4.ch8;

import java.util.List;
import java.util.Date;
import java.util.Set;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringJPASample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
      ctx.refresh();

      ContactAuditService contactService = ctx.getBean(
          "contactAuditService", ContactAuditService.class);

      List<ContactAudit> contacts = contactService.findAll();
      listContacts(contacts);

      System.out.println("Add new contact");
      ContactAudit contact = new ContactAudit();
      contact.setFirstName("Michael");
      contact.setLastName("Jackson");
      contact.setBirthDate(new Date());
      contactService.save(contact);
      contacts = contactService.findAll();
      listContacts(contacts);

      contact = contactService.findById(1l);
      System.out.println("");
      System.out.println("Contact with id 1:" + contact);
      System.out.println("");

      System.out.println("Update contact");
      contact.setFirstName("Tom");
      contactService.save(contact);
      contacts = contactService.findAll();
      listContacts(contacts);
   }
   private static void listContacts(List<ContactAudit> contacts) {
       System.out.println("");
       System.out.println("Listing contacts without details:");
       for (ContactAudit contact: contacts) {
          System.out.println(contact);
          System.out.println();
       }
   }
}
```

В листинге 8.45 мы выводим информацию по аудиту контактов после вставки нового контакта и после его обновления. Запуск этой программы дает следующий вывод:

```
Add new contact
Listing contacts without details:
Contact - Id: 1, First name: Michael, Last name: Jackson, Birthday:
2013-08-11, Create by: prospring4, Create date: 2013-08-11T05:29:37.660-
04:00, Modified by: prospring4, Modified date: 2013-08-11T05:29:37.660-04:00

Update contact
Listing contacts without details:
Contact - Id: 1, First name: Tom, Last name: Jackson, Birthday: 2013-08-11,
Create by: prospring4, Create date: 2013-08-11T05:29:37.660-04:00,
Modified by: prospring4, Modified date: 2013-08-11T05:29:37.864-04:00
```

В приведенном выводе видно, что после создания нового контакта даты создания и последней модификации совпадают. Однако после обновления дата последней модификации изменяется. Аудит — это еще одно удобное средство, предоставляемое Spring Data JPA, так что самостоятельно реализовывать его логику не понадобится.

# Отслеживание версий сущностей с использованием Hibernate Envers

В корпоративном приложении для критически важных бизнес-данных всегда существует требование по сохранению *версий* каждой сущности. Например, в системе управления взаимосвязями с клиентами (customer relationship management — CRM) каждый раз, когда запись клиента вставляется, обновляется или удаляется, ее предыдущая версия должна быть сохранена в таблице хронологии или аудита, чтобы удовлетворять требованиям аудита или другим принятым стандартам.

Для достижения указанных целей существуют два распространенных варианта. Первый предусматривает построение триггеров базы данных, которые будут создавать копию записи в состоянии до обновления в таблице хронологии перед любой операцией обновления, а второй — разработку соответствующей логики на уровне доступа к данным (например, с применением АОП). Тем не менее, обоим вариантам присущи недостатки. Подход с триггерами привязан к платформе базы данных, тогда как реализация логики вручную довольно неуклюжа и сопряжена с ошибками.

Hibernate Envers (Entity Versioning System — система управления версиями сущностей) — это модуль Hibernate, специально спроектированный для автоматизации управления версиями сущностей. В этом разделе мы обсудим использование Hibernate Envers для реализации поддержки версий сущности `ContactAudit`.

---

**На заметку!** Hibernate Envers не является функциональным средством JPA. Мы упоминаем здесь этот модуль, поскольку считаем, что его более уместно рассматривать после обсуждения ряда базовых средств аудита, которые доступны в Spring Data JPA.

---

В Hibernate Envers поддерживаются две стратегии аудита, которые описаны в табл. 8.4.

**Таблица 8.4. Стратегии аудита Hibernate Envers**

| Стратегия аудита | Описание |
|---|---|
| Стандартная | Hibernate Envers поддерживает столбец для номера версии записи. Каждый раз, когда запись вставляется или обновляется, в таблицу хронологии вставляется новая запись с номером версии, извлеченным из последовательности базы данных или таблицы |
| Аудит достоверности | При такой стратегии сохраняются начальная и конечная версии каждой записи хронологии. Когда запись вставляется или обновляется, в таблицу хронологии вставляется новая запись с номером начальной версии. В то же самое время предыдущая запись обновляется номером конечной версии. Также возможно сконфигурировать Hibernate Envers на запись метки времени, когда конечная версия было обновлена, в предыдущей записи хронологии |

В этом разделе мы продемонстрируем работу стратегии аудита достоверности. Хотя она приводит к запуску большего числа обновлений базы данных, извлечение записей хронологии становится намного быстрее. Поскольку метка времени конечной версии также записывается в записи хронологии, при запросе данных упрощается идентификация снимка записи в специфический момент времени.

## Добавление таблиц для отслеживания версий сущностей

Для поддержки отслеживания версий сущностей необходимо добавить несколько таблиц. Прежде всего, для каждой таблицы, в которой будут отслеживаться версии сущностей (в этом случае сущностный класс `ContactAudit`), нужно создать соответствующую таблицу хронологии. Для отслеживания версий записей в таблице `CONTACT_AUDIT` мы создадим таблицу по имени `CONTACT_AUDIT_H`.

В листинге 8.46 приведен сценарий создания этой таблицы (`schema.sql`).

**Листинг 8.46. Таблица `CONTACT_AUDIT_H`**

```
CREATE TABLE CONTACT_AUDIT_H (
   ID INT NOT NULL
   , FIRST_NAME VARCHAR(60) NOT NULL
   , LAST_NAME VARCHAR(40) NOT NULL
   , BIRTH_DATE DATE
   , VERSION INT NOT NULL DEFAULT 0
   , CREATED_BY VARCHAR(20)
   , CREATED_DATE TIMESTAMP
   , LAST_MODIFIED_BY VARCHAR(20)
   , LAST_MODIFIED_DATE TIMESTAMP
   , AUDIT_REVISION INT NOT NULL
   , ACTION_TYPE INT
   , AUDIT_REVISION_END INT
   , AUDIT_REVISION_END_TS TIMESTAMP
   , UNIQUE UQ_CONTACT_AUDIT_H_1 (FIRST_NAME, LAST_NAME)
   , PRIMARY KEY (ID, AUDIT_REVISION)
);
```

Для поддержки стратегии аудита достоверности в каждую таблицу хронологии должны быть добавлены четыре столбца (в листинге 8.46 они выделены полужирным). Эти столбцы кратко описаны в табл. 8.5.

**Таблица 8.5. Столбцы, требуемые для таблицы хронологии**

| Имя столбца | Тип данных | Описание |
|---|---|---|
| AUDIT_REVISION | INT | Начальная версия записи хронологии |
| AUDIT_TYPE | INT | Тип действия со следующими возможными значениями: 0 — добавление, 1 — модификация, 2 — удаление |
| AUDIT_REVISION_END | INT | Конечная версия записи хронологии |
| AUDIT_REVISION_END_TS | TIMESTAMP | Метка времени, когда была обновлена конечная версия |

Hibernate Envers требует наличия еще одной таблицы, предназначенной для отслеживания номеров версий и меток времени, когда каждая версия была создана. Таблица должна называться REVINFO. В листинге 8.47 представлен сценарий создания этой таблицы (schema.sql).

**Листинг 8.47. Таблица REVINFO**

```
CREATE TABLE REVINFO (
   REVTSTMP BIGINT NOT NULL
   , REV INT NOT NULL AUTO_INCREMENT
   , PRIMARY KEY (REVTSTMP, REV)
);
```

Столбец REV предназначен для сохранения каждого номера версии, который будет автоматически инкрементироваться, когда создается новая запись хронологии. Столбец REVTSTMP хранит метку времени (в числовом формате), соответствующую моменту создания версии.

## Конфигурирование EntityManagerFactory для отслеживания версий сущностей

Модуль Hibernate Envers реализован в форме прослушивателей EJB. Эти прослушиватели могут быть сконфигурированы в бине LocalContainerEntityManagerFactory. В листинге 8.48 показана конфигурация бина (app-context-annotation.xml).

**Листинг 8.48. Конфигурирование Hibernate Envers в Spring**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xmlns:tx="http://www.springframework.org/schema/tx"
```

```
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
        http://www.springframework.org/schema/data/jpa
        http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd">

    <jdbc:embedded-database id="dataSource" type="H2">
        <jdbc:script location="classpath:META-INF/sql/schema.sql"/>
        <jdbc:script location="classpath:META-INF/sql/test-data.sql"/>
    </jdbc:embedded-database>

    <bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="emf"/>
    </bean>

    <tx:annotation-driven transaction-manager="transactionManager" />

    <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter">
         <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"/>
        </property>
        <property name="packagesToScan"
            value="com.apress.prospring4.ch8"/>
        <property name="jpaProperties">
            <props>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.H2Dialect
                </prop>
                <prop key="hibernate.max_fetch_depth">3</prop>
                <prop key="hibernate.jdbc.fetch_size">50</prop>
                <prop key="hibernate.jdbc.batch_size">10</prop>
                <prop key="hibernate.show_sql">true</prop>

                <!-- Свойства для Hibernate Envers -->
                <prop key="org.hibernate.envers.audit_table_suffix">_H</prop>
                <prop key="org.hibernate.envers.revision_field_name">
                    AUDIT_REVISION
                </prop>
                <prop key="org.hibernate.envers.revision_type_field_name">
                    ACTION_TYPE
                </prop>
                <prop key="org.hibernate.envers.audit_strategy">
                  org.hibernate.envers.strategy.ValidityAuditStrategy
                </prop>
                <prop key="org.hibernate.envers.audit_strategy_validity_end_
rev_field_name">
                    AUDIT_REVISION_END
                </prop>
```

```
            <prop key="org.hibernate.envers.audit_strategy_validity_store_
revend_timestamp">
                True
            </prop>
            <prop key="org.hibernate.envers.audit_strategy_validity_
revend_timestamp_field_name">
                AUDIT_REVISION_END_TS
            </prop>
        </props>
    </property>
</bean>

<context:component-scan base-package="com.apress.prospring4.ch8"/>

<jpa:repositories base-package="com.apress.prospring4.ch8"
                  entity-manager-factory-ref="emf"
                  transaction-manager-ref="transactionManager"/>

<jpa:auditing auditor-aware-ref="auditorAwareBean"/>

<bean id="auditorAwareBean" class="com.apress.prospring4.ch8.AuditorAwareBean"/>
</beans>
```

Прослушиватель событий аудита Hibernate Envers (`org.hibernate.envers.event.AuditEventListener`) присоединяется к разнообразным событиям постоянства. Прослушиватель перехватывает события после вставки, после обновления или после удаления и помещает в таблицу хронологии копию снимка сущностного класса в состоянии перед обновлением. Прослушиватель также присоединяется к событиям обновления ассоциации (`pre-collection-update`, `pre-collection-remove` и `pre-collection-recreate`) для обработки операций обновления ассоциаций сущностного класса. Hibernate Envers имеет возможность отслеживать хронологию сущностей внутри ассоциации (например, "один ко многим" или "многие ко многим"). Для Hibernate Envers также определено несколько свойств, которые кратко описаны в табл. 8.6 (для ясности префикс свойств, `org.hibernate.envers`, не показан).

**Таблица 8.6. Свойства для Hibernate Envers**

| Имя свойства | Описание |
|---|---|
| `audit_table_suffix` | Суффикс имени таблицы для сущности с отслеживаемыми версиями. Например, для сущностного класса `ContactAudit`, который отображается на таблицу `CONTACT_AUDIT`, Hibernate Envers будет сохранять хронологию в таблице `CONTACT_AUDIT_H`, т.к. свойство `audit_table_suffix` установлено в `_H` |
| `revision_field_name` | Столбец таблицы хронологии для сохранения номера версии для каждой записи хронологии |
| `revision_type_field_name` | Столбец таблицы хронологии для сохранения типа действия обновления |
| `audit_strategy` | Стратегия аудита, используемая для отслеживания версий сущностей |

*Окончание табл. 8.6*

| Имя свойства | Описание |
|---|---|
| `audit_strategy_validity_end_rev_field_name` | Столбец таблицы хронологии для сохранения номера конечной версии для каждой записи хронологии. Требуется только в случае применения стратегии аудита достоверности |
| `audit_strategy_validity_store_revend_timestamp` | Указывает, следует ли сохранять метки времени при обновлении номера конечной версии для каждой записи хронологии. Требуется только в случае применения стратегии с аудитом достоверности |
| `audit_strategy_validity_revend_timestamp_field_name` | Столбец таблицы хронологии для сохранения метки времени, когда обновляется номер конечной версии для каждой записи хронологии. Требуется только в случае применения стратегии с аудитом достоверности и при условии, что предыдущее свойство установлено в `true` |

## Включение отслеживания версий сущностей и извлечения хронологии

Чтобы активизировать отслеживание версий для некоторой сущности, необходимо аннотировать сущностный класс с помощью `@Audited`. В листинге 8.49 приведен сущностный класс `ContactAudit` с примененной аннотацией `@Audited`.

**Листинг 8.49. Класс `ContactAudit`**

```
package com.apress.prospring4.ch8;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;
import javax.persistence.Version;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.Transient;
import org.hibernate.annotations.Type;
import org.springframework.data.domain.Auditable;
import org.joda.time.DateTime;
import org.hibernate.envers.Audited;

@Entity
@Audited
@Table(name = "contact_audit")
public class ContactAudit implements Auditable<String, Long>, Serializable {
    private Long id;
    private int version;
    private String firstName;
```

```
private String lastName;
private Date birthDate;
private String createdBy;
private DateTime createdDate;
private String lastModifiedBy;
private DateTime lastModifiedDate;

@Id
@GeneratedValue(strategy = IDENTITY)
@Column(name = "ID")
public Long getId() {
    return this.id;
}

public void setId(Long id) {
    this.id = id;
}

@Version
@Column(name = "VERSION")
public int getVersion() {
    return this.version;
}

public void setVersion(int version) {
    this.version = version;
}

@Column(name = "FIRST_NAME")
public String getFirstName() {
    return this.firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

@Column(name = "LAST_NAME")
public String getLastName() {
    return this.lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

@Temporal(TemporalType.DATE)
@Column(name = "BIRTH_DATE")
public Date getBirthDate() {
    return this.birthDate;
}

public void setBirthDate(Date birthDate) {
    this.birthDate = birthDate;
}

@Column(name="CREATED_BY")
public String getCreatedBy() {
    return createdBy;
}
```

```
    public void setCreatedBy(String createdBy) {
        this.createdBy = createdBy;
    }

    @Column(name="CREATED_DATE")
    @Type(type="org.jadira.usertype.dateandtime.joda.PersistentDateTime")
    public DateTime getCreatedDate() {
        return createdDate;
    }

    public void setCreatedDate(DateTime createdDate) {
        this.createdDate = createdDate;
    }

    @Column(name="LAST_MODIFIED_BY")
    public String getLastModifiedBy() {
        return lastModifiedBy;
    }

    public void setLastModifiedBy(String lastModifiedBy) {
        this.lastModifiedBy = lastModifiedBy;
    }

    @Column(name="LAST_MODIFIED_DATE")
    @Type(type="org.jadira.usertype.dateandtime.joda.PersistentDateTime")
    public DateTime getLastModifiedDate() {
        return lastModifiedDate;
    }

    public void setLastModifiedDate(DateTime lastModifiedDate) {
        this.lastModifiedDate = lastModifiedDate;
    }

    @Transient
    public boolean isNew() {
        if (id == null) {
            return true;
        } else {
            return false;
        }
    }

    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate
            + ", Create by: " + createdBy + ", Create date: " + createdDate
            + ", Modified by: " + lastModifiedBy + ", Modified date: "
            + lastModifiedDate;
    }
}
```

Этот сущностный класс снабжен аннотацией `@Audited`, которую прослушиватели Hibernate Envers обнаруживают и после этого отслеживают версии обновляемых сущностей. По умолчанию модуль Hibernate Envers также будет пытаться отслеживать хронологию ассоциаций.

Для извлечения записей хронологии в Hibernate Envers предоставляется интерфейс `org.hibernate.envers.AuditReader`, который можно получить из клас-

са AuditReaderFactory. Давайте добавим в интерфейс ContactAuditService новый метод по имени findAuditByRevision(), предназначенный для извлечения записи хронологии ContactAudit по номеру версии. Интерфейс ContactAuditService с упомянутым методом показан в листинге 8.50.

**Листинг 8.50. Интерфейс ContactAuditService с методом findAuditByRevision()**

```
package com.apress.prospring4.ch8;

import java.util.List;

public interface ContactAuditService {
   List<ContactAudit> findAll();
   ContactAudit findById(Long id);
   ContactAudit save(ContactAudit contact);
   ContactAudit findAuditByRevision(Long id, int revision);
}
```

Один из вариантов извлечения записи хронологии предусматривает передачу идентификатора сущности и номера версии. В листинге 8.51 представлена реализация метода findAuditByRevision().

**Листинг 8.51. Реализация метода findAuditByRevision()**

```
package com.apress.prospring4.ch8;

import org.springframework.stereotype.Service;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.beans.factory.annotation.Autowired;

import java.util.List;
import com.google.common.collect.Lists;

import org.hibernate.envers.AuditReader;
import org.hibernate.envers.AuditReaderFactory;

import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;

@Service("contactAuditService")
@Repository
@Transactional
public class ContactAuditServiceImpl implements ContactAuditService {
   @Autowired
   private ContactAuditRepository contactAuditRepository;

   @PersistenceContext
   private EntityManager entityManager;

   @Transactional(readOnly=true)
   public List<ContactAudit> findAll() {
      return Lists.newArrayList(contactAuditRepository.findAll());
   }

   public ContactAudit findById(Long id) {
      return contactAuditRepository.findOne(id);
   }
```

```
    public ContactAudit save(ContactAudit contact) {
        return contactAuditRepository.save(contact);
    }

    @Transactional(readOnly=true)
    @Override
    public ContactAudit findAuditByRevision(Long id, int revision) {
        AuditReader auditReader = AuditReaderFactory.get(entityManager);
        return auditReader.find(ContactAudit.class, id, revision);
    }
}
```

Экземпляр `EntityManager` внедряется в класс и затем передается `AuditReaderFactory` для извлечения экземпляра `AuditReader`. После этого можно вызвать метод `AuditReader.find()`, чтобы извлечь экземпляр сущности `ContactAudit` с заданной версией.

## Тестирование отслеживания версий сущностей

Давайте посмотрим, как работает отслеживание версий сущностей. В листинге 8.52 приведен фрагмент кода, предназначенный для тестирования; код для начальной загрузки `ApplicationContext` и для метода `listContacts()` в листинге 8.52 совпадает с соответствующим кодом в классе `SpringJpaSample`.

**Листинг 8.52. Тестирование отслеживания версий сущностей**

```
package com.apress.prospring4.ch8;

import java.util.List;
import java.util.Date;
import java.util.Set;

import org.springframework.context.support.GenericXmlApplicationContext;

public class SpringJPASample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context-annotation.xml");
        ctx.refresh();

        ContactAuditService contactService = ctx.getBean(
            "contactAuditService", ContactAuditService.class);

        System.out.println("Add new contact");
        ContactAudit contact = new ContactAudit();
        contact.setFirstName("Michael");
        contact.setLastName("Jackson");
        contact.setBirthDate(new Date());
        contactService.save(contact);
        listContacts(contactService.findAll());

        System.out.println("Update contact");
        contact.setFirstName("Tom");
        contactService.save(contact);
        listContacts(contactService.findAll());

        ContactAudit oldContact = contactService.findAuditByRevision(11, 1);
        System.out.println("");
        System.out.println("Old Contact with id 1 and rev 1:" + oldContact);
        System.out.println("");
```

```
        oldContact = contactService.findAuditByRevision(11, 2);
        System.out.println("");
        System.out.println("Old Contact with id 1 and rev 2:" + oldContact);
        System.out.println("");
    }
    private static void listContacts(List<ContactAudit> contacts) {
        System.out.println("");
        System.out.println("Listing contacts without details:");
        for (ContactAudit contact: contacts) {
            System.out.println(contact);
            System.out.println();
        }
    }
}
```

Сначала мы создаем новый контакт и затем обновляем его. После этого мы извлекаем сущности `ContactAudit` с версиями 1 и 2 соответственно. Запуск этого кода дает следующий вывод:

```
Listing contacts without details:
Contact - Id: 1, First name: Tom, Last name: Jackson, Birthday: 2013-08-11,
Create by: prospring4, Create date: 2013-08-11T06:06:21.216-04:00,
Modified by: prospring4, Modified date: 2013-08-11T06:06:21.556-04:00

Old Contact with id 1 and rev 1:Contact - Id: 1, First name: Michael,
Last name: Jackson, Birthday: 2013-08-11, Create by: prospring4, Create
date: 2013-08-11T06:06:21.216-04:00, Modified by: prospring4, Modified
date: 2013-08-11T06:06:21.216-04:00

Old Contact with id 1 and rev 2:Contact - Id: 1, First name: Tom, Last name:
Jackson, Birthday: 2013-08-11, Create by: prospring4, Create date:
2013-08-11T06:06:21.216-04:00, Modified by: prospring4, Modified date:
2013-08-11T06:06:21.556-04:00
```

В показанном выводе видно, что после операции обновления имя в `ContactAudit` было изменено на `Tom`. Однако при просмотре хронологии для версии 1 именем является `Michael`. В версии 2 именем становится `Tom`. Также обратите внимание, что дата последней модификации для версии 2 корректно отражает обновленное значение даты и времени.

# Соображения по поводу того, когда использовать JPA

Несмотря на относительно большой объем материала, мы подробно раскрыли только малую часть JPA. Например, не обсуждалось применение JPA для вызова хранимых процедур базы данных. JPA представляет собой завершенный и мощный стандарт доступа к данным ORM, и благодаря набору библиотек третьих сторон, таких как Spring Data JPA и Hibernate Envers, можно относительно просто реализовать разнообразную сквозную функциональность.

JPA — это стандарт JEE, который поддерживается большинством крупных сообществ открытого кода, а также коммерческими поставщиками (например, JBoss, GlassFish, WebSphere, WebLogic и т.д.). Таким образом, использование JPA в качестве стандарта доступа к данным является вполне обоснованным вариантом. Если требуется абсолютный контроль над запросом, вместо работы с JDBC напрямую можно применять поддержку собственных запросов JPA.

В заключение отметим, что для разработки JEE-приложений с помощью Spring мы рекомендуем использовать JPA для реализации уровня доступа к данным. При желании для реализации некоторых специальных потребностей доступа к данным по-прежнему допускается смешивание с кодом JDBC. Всегда помните, что Spring позволяет легко сочетать технологии доступа к данным с прозрачным управлением транзакциями.

# Резюме

В этой главе мы раскрыли базовые концепции JPA и показали, как конфигурировать `EntityManagerFactory` из JPA в Spring, используя Hibernate в качестве поставщика службы постоянства. Затем мы обсудили применение JPA для выполнения элементарных операций в базе данных. Дополнительные темы включали использование собственных запросов и строго типизированного API-интерфейса критериев JPA.

Кроме того, мы показали, каким образом абстракция `Repository` в Spring Data JPA может упростить разработку приложений JPA, и как применять прослушиватели сущностей Spring Data JPA в целях отслеживания базовой информации аудита для сущностных классов. Также рассматривалось использование модуля Hibernate Envers для полного отслеживания версий сущностных классов.

В следующей главе мы обсудим управление транзакциями в Spring.

# ГЛАВА 9

# Управление транзакциями

Транзакции представляют собой одну из самых важных частей при построении надежного корпоративного приложения. Наиболее общим типом транзакции является операция базы данных. В рамках типовой операции обновления базы данных начинается транзакция базы данных, данные обновляются, а затем по результатам выполнения операции базы данных транзакция фиксируется или откатывается. Однако во многих случаях в зависимости от требований приложения и ресурсов серверной части (такой как СУРБД, промежуточное ПО для обработки сообщений, система ERP и т.д.), с которыми должно взаимодействовать приложение, управление транзакциями может оказаться намного более сложным.

В ранние дни разработки Java-приложений (после появления JDBC, но до того, как стал доступным стандарт JEE или платформа для создания приложений, подобная Spring), разработчики программно контролировали и управляли транзакциями внутри кода приложения. Когда стал доступным стандарт JEE, точнее — стандарт EJB, разработчики получили возможность использовать транзакции, управляемые контейнером (container-managed transaction — CMT), для управления транзакциями декларативным путем. Но сложное объявление транзакций в дескрипторе развертывания EJB было трудно поддерживать, и оно привносило излишнюю сложность в обработку транзакций. Некоторые разработчики предпочитают иметь больший контроль над транзакцией и работают с транзакциями, управляемыми бином (bean-managed transaction — BMT), чтобы управлять ими программным путем. Тем не менее, сложность программирования с использованием API-интерфейса транзакций Java (Java Transaction API — JTA) также препятствует высокой продуктивности разработчиков.

Как обсуждалось в главе 5, посвященной АОП, управление транзакциями является сквозной функциональностью и не должно кодироваться внутри бизнес-логики. Самый подходящий способ реализации управления транзакциями — дать возможность разработчикам определить требования к транзакциям декларативным путем и позволить такой инфраструктуре, как Spring, JEE или АОП, самостоятельно связать логику управления транзакциями. В этой главе мы посмотрим, каким образом Spring может упростить реализацию логики обработки транзакций. Платформа Spring предлагает поддержку как декларативного, так и программного способа управления транзакциями.

В Spring имеется великолепная поддержка декларативных транзакций, а это значит, что загромождать бизнес-логику кодом управления транзакциями не придется. Все, что потребуется сделать — это объявить методы (внутри классов либо уровней), которые должны принимать участие в транзакции, вместе с конфигурацией транзакции, а Spring позаботится о поддержке управления транзакциями. В частности, в этой главе рассматриваются следующие темы.

- **Уровень абстракции транзакций Spring.** Мы обсудим базовые классы абстракции транзакций Spring и объясним, как применять эти классы для управления свойствами транзакций.
- **Декларативное управление транзакциями.** Мы покажем, как использовать Spring и простые Java-объекты для реализации декларативного управления транзакциями. Мы предложим примеры декларативного управления транзакциями, в которых применяются XML-файлы конфигурации, а также Java-аннотации.
- **Программное управление транзакциями.** Несмотря на то что программное управление транзакциями используется очень редко, мы покажем, как работать с предоставляемым Spring классом `TransactionTemplate`, который дает полный контроль над кодом управления транзакциями.
- **Глобальные транзакции с помощью JTA.** Для глобальных транзакций, которые должны охватывать несколько ресурсов серверной части, мы покажем, каким образом конфигурировать и реализовать глобальные транзакции в Spring с применением JTA.

# Исследование уровня абстракции транзакций Spring

Независимо от того, используется Spring или нет, при разработке приложения вы должны совершить фундаментальный выбор типа используемых транзакций — локальных или глобальных. *Локальные транзакции* специфичны для отдельного транзакционного ресурса (к примеру, подключения JDBC), тогда как *глобальные транзакции* управляются контейнером и могут охватывать несколько транзакционных ресурсов.

## Типы транзакций

Локальные транзакции просты в управлении, и если все операции в приложении должны взаимодействовать только с одним транзакционным ресурсом (таким как транзакция JDBC), то локальных транзакций будет вполне достаточно. Однако когда при разработке приложений не используется платформа наподобие Spring, придется написать много кода для управления транзакциями, а если в будущем область действия транзакции понадобится расширить, то код управления локальными транзакциями нужно будет переписать для работы с глобальными транзакциями.

В мире Java глобальные транзакции были реализованы с помощью API-интерфейса транзакций Java (Java Transaction API — JTA). При таком сценарии совместимый с JTA диспетчер транзакций подключается к множеству транзакционных ресурсов через соответствующие диспетчеры ресурсов, которые способны взаимо-

действовать с диспетчером транзакций по протоколу XA (открытый стандарт, определяющий распределенные транзакции). Кроме того, с помощью механизма двухфазной фиксации (2 Phase Commit — 2PC) обеспечивается корректное обновление или откат всех участвующих источников данных серверной части. В случае отказа любого из ресурсов серверной части будет выполнен откат всей транзакции, следовательно, откат будет совершен и в отношении обновлений других ресурсов.

На рис. 9.1 показано высокоуровневое представление глобальных транзакций, реализуемых посредством JTA.

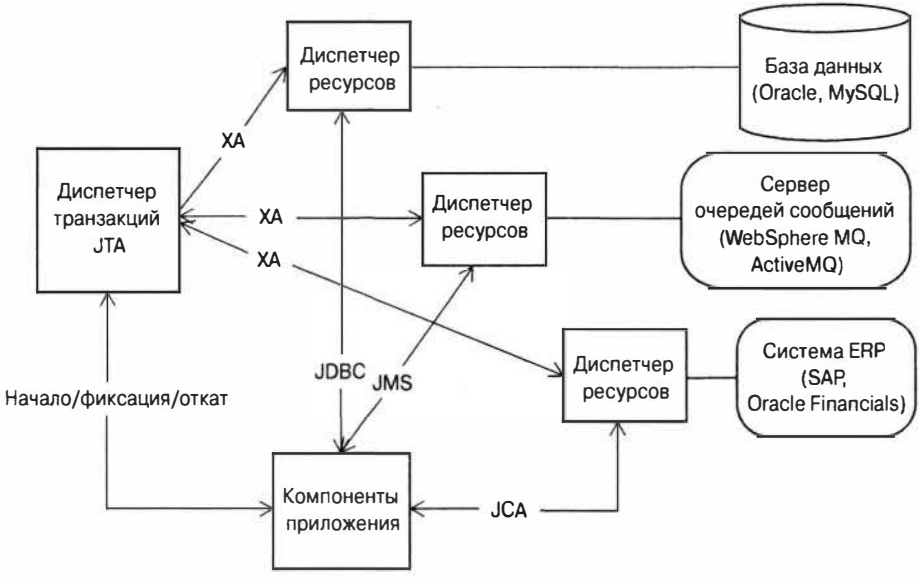


**Рис. 9.1.** Обзор глобальных транзакций, реализуемых с помощью JTA

На рис. 9.1 видно, что в глобальной транзакции (также в общем случае называемой *распределенной транзакцией*) принимают участие четыре основных части. Первый участник — это ресурс серверной части, такой как СУРБД, промежуточное ПО для обмена сообщениями, система планирования ресурсов предприятия (enterprise resource planning — ERP) и т.д.

Вторым участником является диспетчер ресурсов, который обычно предоставляется производителем ресурса серверной части и отвечает за взаимодействие с этим ресурсом. Например, при подключении к базе данных MySQL мы должны взаимодействовать с классом `MysqlXADataSource`, предоставленным Java-коннектором MySQL. Другие ресурсы серверной части (MQ, ERP и т.д.) также предлагают свои диспетчеры ресурсов.

Третий участник — это диспетчер транзакций JTA, который отвечает за управление, координацию и синхронизацию состояния транзакции со всеми диспетчерами ресурсов, принимающими участие в транзакции. При этом будет применяться протокол XA, который представляет собой открытый стандарт, широко используемый для обработки распределенных транзакций. Диспетчер транзакций JTA также поддерживает 2PC, поэтому все изменения будут фиксироваться вместе, и если обновление любого ресурса дает сбой, производится откат всей транзакции, в результате

чего ни один из ресурсов обновляться не будет. Весь механизм целиком описан в спецификации службы транзакций Java (Java Transaction Service — JTS).

Четвертым участником является приложение. Транзакцией управляет (начинает, фиксирует, открывает и т.д.) либо само приложение, либо лежащий в основе контейнер, либо инфраструктура Spring, в рамках которой запускается приложение. В то же самое время приложение взаимодействует с ресурсами серверной части через различные стандарты, определенные JEE. Как показано на рис. 9.1, приложение подключается к СУРБД через JDBC, к MQ через JMS и к системе ERP через архитектуру коннекторов Java EE (Java EE Connector Architecture — JCA).

Интерфейс JTA поддерживается всеми полномасштабными серверами приложений, совместимыми с JEE (например, JBoss, WebSphere, WebLogic и GlassFish), внутри которых транзакция доступна через поиск JNDI. Как и для автономных приложений или веб-контейнеров (скажем, Tomcat и Jetty), существуют также коммерческие решения и решения с открытым кодом, предлагающие поддержку JTA/XA в упомянутых средах (к примеру, Atomikos, JOTM (Java Open Transaction Manager — открытый диспетчер транзакций Java) и Bitronix).

### Реализации интерфейса `PlatformTransactionManager`

В Spring интерфейс `PlatformTransactionManager` использует интерфейсы `TransactionDefinition` и `TransactionStatus` для создания и управления транзакциями. Действительная реализация этих интерфейсов требует глубоких знаний диспетчера транзакций.

На рис. 9.2 показаны реализации `PlatformTransactionManager` в Spring.

Платформа Spring предлагает обширный набор реализаций интерфейса `PlatformTransactionManager`. Класс `CciLocalTransactionManager` поддерживает JEE, JCA и CCI (Common Client Interface — интерфейс общего клиента).

Класс `DataSourceTransactionManager` предназначен для обобщенных подключений JDBC.

Для объектно-реляционного отображения предусмотрено несколько реализаций, включая JDO (класс `JdoTransactionManager`), JPA (класс `JpaTransactionManager`), а также Hibernate 3 и Hibernate 4 (класс `HibernateTransactionManager` с разными именами пакетов).

В отношении JMS доступны реализации для JMS 1.1 и более новых версий (класс `JmsTransactionManager`). Для JTA имеется обобщенный класс реализации `JtaTransactionManager`.

В Spring также предоставляется ряд классов диспетчеров транзакций JTA, специфичных для конкретных серверов приложений. Эти классы обеспечивают встроенную поддержку для WebSphere (класс `WebSphereUowTransactionManager`), WebLogic (класс `WebLogicJtaTransactionManager`) и Oracle OC4J (класс `OC4JJtaTransactionManager`).

## Анализ свойств транзакций

В этом разделе мы обсудим свойства транзакций, поддерживаемые Spring, уделяя основное внимание взаимодействию с СУРБД в качестве ресурса серверной части.

Транзакции обладают четырьмя хорошо известными свойствами ACID (atomicity, consistency, isolation, durability — атомарность, согласованность, изолированность,

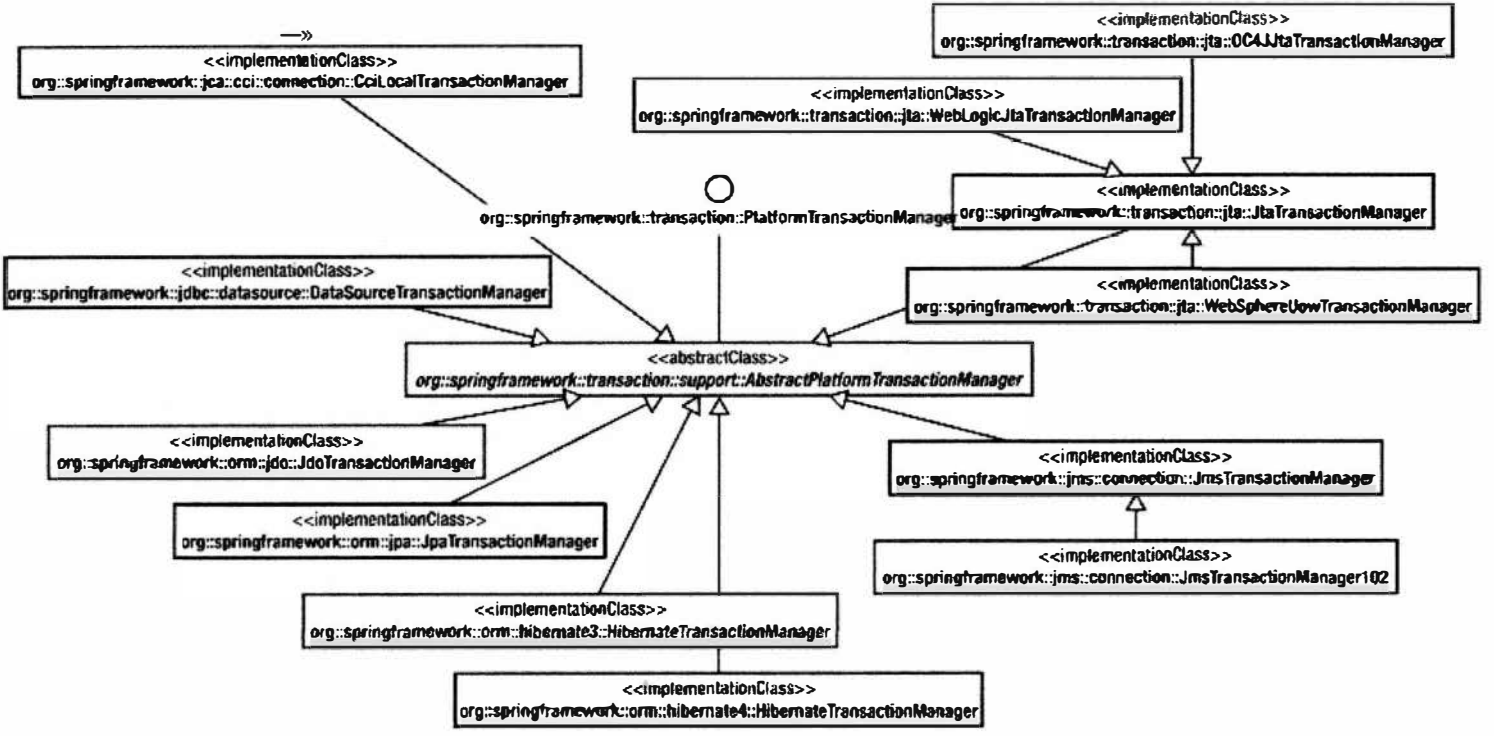


**Рис. 9.2.** Реализации `PlatformTransactionManager` в Spring

долговечность), и за поддержку этих аспектов транзакции отвечают транзакционные ресурсы. Возможность управления атомарностью, согласованностью и долговечностью транзакции отсутствует, но можно управлять ее распространением и тайм-аутом, а также конфигурировать, должна ли транзакция быть предназначена только для чтения, и задавать уровень изоляции.

Все эти параметры инкапсулированы Spring в интерфейсе `TransactionDefinition`. Этот интерфейс используется в ключевом интерфейсе поддержки транзакций `PlatformTransactionManager`, реализации которого отвечают за управление транзакциями на конкретной платформе, такой как JDBC или JTA. Основной метод, `PlatformTransactionManager.getTransaction()`, получает интерфейс `TransactionDefinition` в качестве аргумента и возвращает интерфейс `TransactionStatus`. Интерфейс `TransactionStatus` применяется для управления выполнением транзакции, а конкретнее — для установки результата транзакции и проверки, завершена ли транзакция, или является ли транзакция новой.

## Интерфейс `TransactionDefinition`

Как упоминалось ранее, интерфейс `TransactionDefinition` управляет свойствами транзакции. Давайте рассмотрим этот интерфейс более детально (листинг 9.1) и опишем его методы.

**Листинг 9.1. Интерфейс `TransactionDefinition`**

```
package org.springframework.transaction;

import java.sql.Connection;

public interface TransactionDefinition {

    // Объявления переменных не показаны

    int getPropagationBehavior();
    int getIsolationLevel();
    int getTimeout();
    boolean isReadOnly();
    String getName();
}
```

Простыми и очевидными методами этого интерфейса являются `getTimeout()`, возвращающий промежуток времени (в секундах), в течение которого транзакция должна быть завершена, и `isReadOnly()`, указывающий, предназначена ли транзакция только для чтения. Реализация диспетчера транзакций может использовать эти значения для оптимизации выполнения и проверки того, что транзакция производит только операции чтения. Метод `getName()` возвращает имя транзакции.

Оставшиеся два метода, `getPropagationBehavior()` и `getIsolationLevel()`, заслуживают более пристального внимания.

Мы начнем с метода `getIsolationLevel()`, который управляет тем, какие изменения в данных видны другим транзакциям. В табл. 9.1 перечислены уровни изоляции транзакции вместе с описаниями, какие изменения, сделанные текущей транзакцией, доступны другим транзакциям.

**Таблица 9.1. Уровни изоляции транзакции**

| Уровень изоляции | Описание |
|---|---|
| `TransactionDefinition.ISOLATION_DEFAULT` | Стандартный уровень изоляции лежащего в основе хранилища данных |
| `TransactionDefinition.ISOLATION_READ_UNCOMMITTED` | Самый низкий уровень изоляции; это с трудом можно назвать транзакцией, поскольку при таком уровне разрешено видеть данные, модифицированные другими незафиксированными транзакциями |
| `TransactionDefinition.ISOLATION_READ_COMMITTED` | Стандартный уровень изоляции в большинстве баз данных; он гарантирует, что другие транзакции не имеют возможности читать данные, которые не были зафиксированы текущей транзакцией. Однако данные, прочитанные одной транзакцией, могут быть обновлены другими транзакциями |
| `TransactionDefinition.ISOLATION_REPEATABLE_READ` | Более строгий уровень, чем `ISOLATION_READ_COMMITED`; он гарантирует, что после выборки данных можно произвести выборку, по крайней мере, того же набора данных снова. Если другие транзакции вставили новые данные, эти новые данные могут быть извлечены |
| `TransactionDefinition.ISOLATION_SERIALIZABLE` | Наиболее дорогостоящий и надежный уровень изоляции; все транзакции трактуются как выполняемые последовательно, друг за другом |

Выбор подходящего уровня изоляции очень важен для согласованности данных, но оказывает сильное влияние на производительность. Самый высокий уровень изоляции, `TransactionDefinition.ISOLATION_SERIALIZABLE`, является особенно дорогостоящим в обслуживании.

Метод `getPropagationBehavior()` определяет, что произойдет с транзакционным вызовом в зависимости от того, существует ли активная транзакция. Значения, с которыми работает этот метод, описаны в табл. 9.2.

**Таблица 9.2. Значения, описывающие поведение распространения**

| Поведение распространения | Описание |
|---|---|
| `TransactionDefinition.PROPAGATION_REQUIRED` | Поддерживает транзакцию, если она уже существует. Если транзакций нет, начинается новая транзакция |
| `TransactionDefinition.PROPAGATION_SUPPORTS` | Поддерживает транзакцию, если она уже существует. Если транзакций нет, выполнение происходит без транзакций |
| `TransactionDefinition.PROPAGATION_MANDATORY` | Поддерживает транзакцию, если она уже существует. Если нет активной транзакции, генерируется исключение |
| `TransactionDefinition.PROPAGATION_REQUIRES_NEW` | Всегда начинает новую транзакцию. Если активная транзакция уже существует, она приостанавливается |

*Окончание табл. 9.2*

| Поведение распространения | Описание |
|---|---|
| `TransactionDefinition.PROPAGATION_NOT_SUPPORTED` | Не поддерживает выполнение с активной транзакцией. Всегда выполняется без транзакций и приостанавливает любые существующие транзакции |
| `TransactionDefinition.PROPAGATION_NEVER` | Всегда выполняется без транзакций, даже если имеется активная транзакция. Если активная транзакция существует, генерируется исключение |
| `TransactionDefinition.PROPAGATION_NESTED` | Выполняется во вложенной транзакции, если существует активная транзакция. Если активной транзакции нет, поведение соответствует `TransactionDefinition.PROPAGATION_REQUIRED` |

## Интерфейс `TransactionStatus`

Интерфейс `TransactionStatus`, показанный в листинге 9.2, позволяет диспетчеру транзакций управлять выполнением транзакции. Код может проверить, является ли транзакция новой или предназначенной только для чтения, и способен инициировать откат.

**Листинг 9.2. Интерфейс `TransactionStatus`**

```
package org.springframework.transaction;

public interface TransactionStatus extends SavepointManager {
   boolean isNewTransaction();
   boolean hasSavepoint();
   void setRollbackOnly();
   boolean isRollbackOnly();
   void flush();
   boolean isCompleted();
}
```

Методы интерфейса `TransactionStatus` самоочевидны; наиболее примечательным из них является `setRollbackOnly()`, который вызывает откат и завершает активную транзакцию.

Метод `hasSavePoint()` возвращает признак наличия точки сохранения внутри транзакции (т.е. эта транзакция была создана как вложенная на основе точки сохранения). Метод `flush()` сбрасывает сеанс в хранилище данных, если это применимо (например, когда используется Hibernate). Метод `isCompleted()` возвращает признак завершения транзакции (т.е. фиксация или откат).

# Модель данных и инфраструктура для кода примеров

В этом разделе представлен обзор модели данных и инфраструктуры, которые будут использоваться в примерах управления транзакциями. Мы будем применять JPA с Hibernate в качестве уровня постоянства для реализации логики доступа к данным.

Вдобавок, в целях упрощения разработки основных операций базы данных будет также использоваться проект Spring Data JPA и его абстракция репозитория.

## Создание простого проекта Spring Data JPA с зависимостями

Давайте начнем с создания проекта. Поскольку используется JPA, мы также должны добавить к проекту обязательные зависимости, требуемые для примеров этой главы. Все эти дополнительные зависимости перечислены в табл. 9.3.

**Таблица 9.3. Зависимости Maven для Spring Data JPA**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.springframework.data` | `spring-data-jpa` | 1.5.0.RELEASE | Библиотека Spring Data JPA |
| `org.springframework` | `spring-core` | 4.0.2.RELEASE | Модуль ядра Spring |
| `org.springframework` | `spring-context` | 4.0.2.RELEASE | Модуль контекста Spring |
| `org.hibernate.javax.persistence` | `hibernate-jpa-2.1-api` | 1.0.0.Final | API-интерфейс JPA из Hibernate |
| `org.hibernate` | `hibernate-entitymanager` | 4.2.3.Final | Диспетчер сущностей Hibernate |
| `com.google.guava` | `guava` | 14.0.1 | Полезные вспомогательные классы |
| `log4j` | `log4j` | 1.2.17 | Реализация регистрации в журнале log4j |
| `com.h2database` | `h2` | 1.3.172 | Встроенная база данных H2 |
| `org.slf4j` | `slf4j-log4j12` | 1.7.6 | Библиотека, которая соединяет регистрацию в журнале SLF4J с библиотекой log4j |
| `org.aspectj` | `aspectjrt` | 1.7.2 (1.8.0.M1 для Java 8) | Библиотека времени выполнения AspectJ |
| `org.aspectj` | `aspectjweaver` | 1.7.2 (1.8.0.M1 для Java 8) | Библиотека связывания AspectJ |

Чтобы исследовать поведение кода примера при изменении атрибутов транзакции, мы установим уровень ведения журнала log4j в `DEBUG`. В листинге 9.3 приведено содержимое файла `log4j.properties`.

**Листинг 9.3. Включение уровня ведения журнала `DEBUG` в файле `log4j.properties`**

```
log4j.rootCategory=DEBUG, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - <%m>%n

log4j.category.org.springframework.transaction=INFO
log4j.category.org.hibernate.SQL=DEBUG
```

## Модель данных и общие классы

Для упрощения примеров мы будем использовать всего одну таблицу по имени CONTACT, которая применялась в главах, посвященных доступу к данным. В листингах 9.4 и 9.5 показаны сценарии для создания этой таблицы (schema.sql) и наполнения ее тестовыми данными (test-data.sql).

**Листинг 9.4. Сценарий для создания таблицы**

```
DROP TABLE IF EXISTS CONTACT;
CREATE TABLE CONTACT (
    ID INT NOT NULL AUTO_INCREMENT
    , FIRST_NAME VARCHAR(60) NOT NULL
    , LAST_NAME VARCHAR(40) NOT NULL
    , BIRTH_DATE DATE
    , VERSION INT NOT NULL DEFAULT 0
    , UNIQUE UQ_CONTACT_1 (FIRST_NAME, LAST_NAME)
    , PRIMARY KEY (ID)
);
```

**Листинг 9.5. Сценарий для наполнения таблицы тестовыми данными**

```
insert into contact (first_name, last_name, birth_date)
   values ('Chris', 'Schaefer', '1981-05-03');
insert into contact (first_name, last_name, birth_date)
   values ('Scott', 'Tiger', '1990-11-02');
insert into contact (first_name, last_name, birth_date)
   values ('John', 'Smith', '1964-02-28');
```

Сущностный класс также прост; код класса Contact приведен в листинге 9.6.

**Листинг 9.6. Класс Contact**

```
package com.apress.prospring4.ch9;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.Version;
import java.io.Serializable;
import java.util.Date;

@Entity
@Table(name = "contact")
@NamedQueries({
  @NamedQuery(name="Contact.findAll", query="select c from Contact c"),
  @NamedQuery(name="Contact.countAll", query="select count(c) from Contact c")
})
```

```
public class Contact implements Serializable {
   private Long id;
   private int version;
   private String firstName;
   private String lastName;
   private Date birthDate;
   public Contact() {
   }
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "ID")
   public Long getId() {
      return this.id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   @Version
   @Column(name = "VERSION")
   public int getVersion() {
      return this.version;
   }
   public void setVersion(int version) {
      this.version = version;
   }
   @Column(name = "FIRST_NAME")
   public String getFirstName() {
      return this.firstName;
   }
   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }
   @Column(name = "LAST_NAME")
   public String getLastName() {
      return this.lastName;
   }
   public void setLastName(String lastName) {
      this.lastName = lastName;
   }
   @Temporal(TemporalType.DATE)
   @Column(name = "BIRTH_DATE")
   public Date getBirthDate() {
      return this.birthDate;
   }
   public void setBirthDate(Date birthDate) {
      this.birthDate = birthDate;
   }
   @Override
   public String toString() {
      return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate;
   }
}
```

Для использования абстракции репозитория Spring Data JPA мы также должны определить интерфейс `ContactRepository`, который расширяет интерфейс `CrudRepository<T,ID extends Serializable>` из проекта Spring Data Common. Интерфейс `ContactRepository` представлен в листинге 9.7.

**Листинг 9.7. Интерфейс `ContactRepository`**

```
package com.apress.prospring4.ch9;

import org.springframework.data.repository.CrudRepository;

public interface ContactRepository extends CrudRepository<Contact, Long> {

}
```

В листинге 9.7 видно, что никаких дополнительных методов не требуется, поскольку методов, предоставляемых интерфейсом `CrudRepositoy`, уже достаточно для примеров, рассматриваемых в этой главе.

Наконец, давайте взглянем на интерфейс `ContactService`, который определяет всю бизнес-логику, относящуюся к сущностному классу `Contact`. Код интерфейса `ContactService` приведен в листинге 9.8.

**Листинг 9.8. Интерфейс `ContactService`**

```
package com.apress.prospring4.ch9;

import java.util.List;

public interface ContactService {
    List<Contact> findAll();
    Contact findById(Long id);
    Contact save(Contact contact);
    long countAll();
}
```

Методы этого интерфейса не требуют дополнительных пояснений. В следующем разделе мы обсудим, как обеспечить управление транзакциями различными путями, реализуя интерфейс `ContactService`.

# Декларативные и программные транзакции в Spring

В Spring есть три варианта управления транзакциями. Два из них предназначены для декларативного управления транзакциями, причем в одном варианте используются Java-аннотации, а в другом — XML-конфигурация. Третий вариант предусматривает программное управление транзакциями. Все три варианта рассматриваются в последующих разделах.

## Использование аннотаций для управления транзакциями

В настоящее время применение аннотаций является общепринятым способом определения требований к транзакциям в Spring. Главное достоинство такого подхода состоит в том, что требования к транзакциям вместе с их свойствами (тайм-аут,

уровень изоляции, поведение распространения и т.д.) определены внутри кода, упрощая отладку и сопровождение приложения.

Чтобы включить поддержку аннотаций для управления транзакциями в Spring, необходимо добавить дескриптор `<tx:annotation-driven>` в XML-файл конфигурации. Содержимое этого файла (`tx-annotation-app-context.xml`) показано в листинге 9.9.

**Листинг 9.9. Конфигурация Spring для поддержки аннотаций при управлении транзакциями**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xsi:schemaLocation="http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx.xsd
      http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:META-INF/config/schema.sql"/>
      <jdbc:script location="classpath:META-INF/config/test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionManager"
         class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <tx:annotation-driven/>

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
      />
      </property>
      <property name="packagesToScan" value="com.apress.prospring4.ch9 "/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
```

```
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>

<context:component-scan
    base-package="com.apress.prospring4.ch9" />

<jpa:repositories base-package="com.apress.prospring4.ch9"
    entity-manager-factory-ref="emf"
    transaction-manager-ref="transactionManager"/>
</beans>
```

Сначала определяется встроенная база данных H2 с помощью сценариев для ее создания и наполнения тестовыми данными. Затем, поскольку используется JPA, определяется бин `JpaTransactionManager`. Дескриптор `<tx:annotation-driven>` указывает, что для управления транзакциями применяются аннотации. После этого определяется бин `EntityManagerFactory`, за которым следует дескриптор `<context:component-scan>`, предназначенный для сканирования классов уровня обслуживания. Наконец, с помощью дескриптора `<jpa:repositories>` включается абстракция репозитория Spring Data JPA.

Для класса реализации интерфейса `ContactService` мы начинаем с пустых реализаций всех методов интерфейса `ContactService`.

Давайте реализуем метод `ContactService.findAll()` первым. В листинге 9.10 приведен код класса `ContactServiceImpl` с реализованным методом `findAll()`.

**Листинг 9.10. Класс `ContactServiceImpl` с реализованным методом `findAll()`**

```
package com.apress.prospring4.ch9;

import com.google.common.collect.Lists;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    private ContactRepository contactRepository;

    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return Lists.newArrayList(contactRepository.findAll());
    }

    @Autowired
    public void setContactRepository(ContactRepository contactRepository) {
        this.contactRepository = contactRepository;
    }
}
```

При использовании управления транзакциями на основе аннотаций нам придется иметь дело с единственной аннотацией — `@Transactional`. В листинге 9.10 аннотация `@Transactional` применялась на уровне класса, а это значит, что по умолчанию Spring будет гарантировать существование транзакции перед выполнением каждого метода этого класса. Аннотация `@Transactional` поддерживает несколько атрибутов, с помощью которых можно переопределять ее стандартное поведение. В табл. 9.4 описаны доступные атрибуты вместе с их возможными и стандартными значениями.

**Таблица 9.4. Атрибуты для аннотации `@Transactional`**

| Имя атрибута | Стандартное значение | Возможные значения |
|---|---|---|
| `propagation` | `Propagation.REQUIRED` | `Propagation.REQUIRED`<br>`Propagation.SUPPORTS`<br>`Propagation.MANDATORY`<br>`Propagation.REQUIRES_NEW`<br>`Propagation.NOT_SUPPORTED`<br>`Propagation.NEVER`<br>`Propagation.NESTED` |
| `isolation` | `Isolation.DEFAULT` (стандартный уровень изоляции лежащего в основе ресурса) | `Isolation.DEFAULT`<br>`Isolation.READ_UNCOMMITTED`<br>`Isolation.READ_COMMITTED`<br>`Isolation.REPEATABLE_READ`<br>`Isolation.SERIALIZABLE` |
| `timeout` | `TransactionDefinition.TIMEOUT_DEFAULT` (стандартный тайм-аут транзакции в секундах лежащего в основе ресурса) | Целочисленное значение больше нуля; задает количество секунд для тайм-аута транзакции |
| `readOnly` | `false` | `true`<br>`false` |
| `rollbackFor` | Классы исключений, для которых будет осуществлен откат транзакции | – |
| `rollbackForClassName` | Имена классов исключений, для которых будет осуществлен откат транзакции | – |
| `noRollbackFor` | Классы исключений, для которых не будет осуществлен откат транзакции | – |
| `noRollbackForClassName` | Имена классов исключений, для которых не будет осуществлен откат транзакции | – |
| `value` | `""` (значение квалификатора для указанной транзакции) | – |

Согласно табл. 9.4, аннотация `@Transactional` без атрибутов означает, что распространение транзакции является обязательным, уровень изоляции — стандартным, тайм-аут имеет значение по умолчанию, и установлен режим чтения-записи.

Метод `findAll()` в листинге 9.10 аннотирован посредством `@Transactional(readOnly=true)`. Это переопределяет аннотацию по умолчанию, примененную на уровне класса, и оставляет неизмененными все атрибуты кроме режима только для чтения.

В листинге 9.11 показана программа для тестирования метода `findAll()`.

**Листинг 9.11. Тестирование метода `findAll()`**

```
package com.apress.prospring4.ch9;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

public class TxAnnotationSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/tx-annotation-app-context.xml");
      ctx.refresh();

      ContactService contactService = ctx.getBean("contactService",
         ContactService.class);

      List<Contact> contacts = contactService.findAll();

      for (Contact contactTemp: contacts) {
         System.out.println(contactTemp);
       }
   }
}
```

Запуск этой программы дает следующий вывод, приведенный с сокращениями (для получения дополнительной информации просмотрите вывод отладки на консоли):

```
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday: 1981-05-03
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday: 1990-11-02
Contact - Id: 3, First name: John, Last name: Smith, Birthday: 1964-02-28
```

Для ясности в показанном выше выводе операторы, не относящиеся к делу, не показаны. В первой строке перед запуском метода `findAll()` экземпляр `JpaTransactionManager` из Spring создает новую транзакцию (имя соответствует полностью определенному имени класса с именем метода) со стандартными атрибутами, но транзакция предназначена только для чтения, как определено в аннотации `@Transactional` на уровне метода. Затем запрос отправляется и после его завершения без ошибок транзакция фиксируется. Операции создания и фиксации обрабатываются `JpaTransactionManager`.

Теперь перейдем к реализации операции обновления. Необходимо реализовать методы `findById()` и `save()` в интерфейсе `ContactService`. Их реализация показана в листинге 9.12.

**Листинг 9.12. Класс `ContactServiceImpl` с реализованными методами `findById()` и `save()`**

```
package com.apress.prospring4.ch9;

import com.google.common.collect.Lists;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    private ContactRepository contactRepository;

    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return Lists.newArrayList(contactRepository.findAll());
    }

    @Override
    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        return contactRepository.findOne(id);
    }

    @Override
    public Contact save(Contact contact) {
        return contactRepository.save(contact);
    }

    @Override
    public long countAll() {
        return 0;
    }

    @Autowired
    public void setContactRepository(ContactRepository contactRepository) {
        this.contactRepository = contactRepository;
    }
}
```

Метод `findById()` также аннотирован с помощью `@Transactional(readOnly=true)`. В общем случае атрибут `readOnly=true` должен применяться ко всем методам поиска. Главная причина такого подхода связана с тем, что большинство поставщиков постоянства будут проводить определенную оптимизацию транзакций, предназначенных только для чтения. Например, Hibernate не будет поддерживать снимки управляемых экземпляров, извлекаемых из базы данных с включенным режимом только для чтения.

В методе `save()` мы просто вызываем метод `CrudRepository.save()` и не предоставляем никаких аннотаций. Это значит, что будет использоваться аннотация уровня класса, определяющая транзакцию только для чтения.

Давайте модифицируем класс `TxAnnotationSample` для тестирования метода `save()`, как показано в листинге 9.13.

**Листинг 9.13. Тестирование метода `save()`**

```
package com.apress.prospring4.ch9;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

public class TxAnnotationSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/tx-annotation-app-context.xml");
        ctx.refresh();

        ContactService contactService = ctx.getBean("contactService",
                ContactService.class);

        List<Contact> contacts = contactService.findAll();

        for (Contact contactTemp: contacts) {
            System.out.println(contactTemp);
        }
        Contact contact = contactService.findById(1L);
        contact.setFirstName("Peter");
        contactService.save(contact);
        System.out.println("Contact saved successfully: " + contact);
    }
}
```

Здесь был извлечен объект `Contact` с идентификатором 1, его имя было изменено, после чего объект был сохранен в базе данных. Ниже приведен вывод, имеющий отношение к этому методу:

```
Contact saved successfully: Contact - Id: 1, First name: Peter,
Last name: Schaefer, Birthdate: 1981-05-03
```

Метод `save()` получает стандартные атрибуты, унаследованные от аннотации `@Transactional` уровня класса. После завершения операции обновления `JpaTransactionManager` запускает фиксацию транзакции, что заставляет Hibernate сбросить контекст постоянства и закрыть лежащее в основе JDBC-подключение к базе данных.

Наконец, рассмотрим метод `countAll()`. В этом методе мы исследуем две конфигурации для транзакции. Хотя метод `CrudRepository.count()` вполне способен решить задачу, мы не будем его использовать. Взамен для демонстрационных целей мы реализуем другой метод. В основном это объясняется тем, что методы, определенные интерфейсом `CrudRepository` в Spring Data, уже помечены соответствующими атрибутами транзакций.

В листинге 9.14 показан новый метод `countAllContacts()`, определенный в интерфейсе `ContactRepository`.

**Листинг 9.14. Интерфейс `ContactRepository` с методом `countAllContacts()`**

```
package com.apress.prospring4.ch9;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
public interface ContactRepository extends CrudRepository<Contact, Long> {
    @Query("select count(c) from Contact c")
    Long countAllContacts();
}
```

К новому методу `countAllContacts()` применена аннотация `@Query` со значением, задающим JPQL-оператор, который подчитывает количество контактов. Реализация метода `countAll()` в классе `ContactServiceImpl` представлена в листинге 9.15.

**Листинг 9.15. Класс `ContactServiceImpl` с реализованным методом `countAll()`**

```
package com.apress.prospring4.ch9;

import com.google.common.collect.Lists;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;
@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    private ContactRepository contactRepository;
    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return Lists.newArrayList(contactRepository.findAll());
    }
    @Override
    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        return contactRepository.findOne(id);
    }
    @Override
    public Contact save(Contact contact) {
        return contactRepository.save(contact);
    }
    @Override
    @Transactional(readOnly=true)
    public long countAll() {
        return contactRepository.countAllContacts();
    }
    @Autowired
    public void setContactRepository(ContactRepository contactRepository) {
        this.contactRepository = contactRepository;
    }
}
```

Здесь используется та же самая аннотация, что и в других методах поиска. В листинге 9.16 приведен код для тестирования нового метода.

**Листинг 9.16. Тестирование метода `countAll()`**

```
package com.apress.prospring4.ch9;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

public class TxAnnotationSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/tx-annotation-app-context.xml");
      ctx.refresh();

      ContactService contactService = ctx.getBean("contactService",
            ContactService.class);

      List<Contact> contacts = contactService.findAll();

      for (Contact contactTemp: contacts) {
          System.out.println(contactTemp);
      }

      Contact contact = contactService.findById(1L);
      contact.setFirstName("Peter");
      contactService.save(contact);
      System.out.println("Contact saved successfully: " + contact);
      System.out.println("Contact count: " + contactService.countAll());
   }
}
```

Запуск этой программы дает следующий вывод:

```
Contact count: 3
```

В этом выводе видно, что для метода `countAll()` была создана транзакция, предназначенная только для чтения, чего и следовало ожидать.

Но мы не хотим, чтобы в транзакции участвовала функция `countAll()`. Причина в том, что ее результат не должен управляться диспетчером сущностей JPA. Вместо этого нужно просто получить счетчик и забыть о нем. В таком случае можно переопределить поведение распространения транзакции на `Propagation.NEVER`. В листинге 9.17 показан переделанный метод `countAll()`.

**Листинг 9.17. Класс `ContactServiceImpl` с переделанной реализацией метода `countAll()`**

```
package com.apress.prospring4.ch9;

import com.google.common.collect.Lists;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;
```

```
@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    private ContactRepository contactRepository;

    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return Lists.newArrayList(contactRepository.findAll());
    }

    @Override
    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        return contactRepository.findOne(id);
    }

    @Override
    public Contact save(Contact contact) {
        return contactRepository.save(contact);
    }

    @Override
    @Transactional(propagation= Propagation.NEVER)
    public long countAll() {
        return contactRepository.countAllContacts();
    }

    @Autowired
    public void setContactRepository(ContactRepository contactRepository) {
        this.contactRepository = contactRepository;
    }
}
```

Снова запустив тестовый код из листинга 9.16, вы увидите в выводе отладки, что для метода `countAll()` транзакция создаваться не будет.

В этом разделе рассматривались основные конфигурации, с которыми вам придется иметь дело при повседневной обработке транзакций. В специальных случаях может понадобиться определять тайм-аут, уровень изоляции, выполнять ли откат (или нет) для специфических исключений, а также другие параметры.

**На заметку!** Класс `JpaTransactionManager` в Spring не поддерживает специальный уровень изоляции. Вместо этого он всегда использует стандартный уровень изоляции для лежащего в основе хранилища данных. В случае применения Hibernate в качестве поставщика постоянства для поддержки специального уровня изоляции существует обходной путь, заключающийся в расширении класса `HibernateJpaDialect`.

## Использование XML-конфигурации для управления транзакциями

Еще один распространенный подход к декларативному управлению транзакциями предусматривает применение поддержки АОП в Spring. До выхода версии Spring 2 для определения требований к транзакциям с бинами Spring нужно было исполь-

зовать класс TransactionProxyFactoryBean. Однако, начиная с версии Spring 2, появился более простой способ определения таких требований за счет введения пространства имен aop и применения общих приемов конфигурирования АОП.

В этом разделе будет использоваться тот же самый пример, с которым мы работали при объяснении подхода с аннотациями. Мы просто приведем его к стилю XML-конфигурации. В листинге 9.18 показано содержимое XML-файла конфигурации для управления транзакциями (tx-declarative-app-context.xml).

**Листинг 9.18. XML-конфигурация для управления транзакциями**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xsi:schemaLocation="http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx.xsd
      http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop.xsd">

   <aop:config>
      <aop:pointcut id="serviceOperation" expression=
         "execution(* com.apress.prospring4.ch9.*ServiceImpl.*(..))"/>
      <aop:advisor pointcut-ref="serviceOperation" advice-ref="txAdvice"/>
   </aop:config>

   <tx:advice id="txAdvice">
      <tx:attributes>
         <tx:method name="find*" read-only="true"/>
         <tx:method name="count*" propagation="NEVER"/>
         <tx:method name="*"/>
      </tx:attributes>
   </tx:advice>

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:META-INF/config/schema.sql"/>
      <jdbc:script location="classpath:META-INF/config/test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionManager"
         class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>
```

```
    <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter">
         <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
         />
        </property>
        <property name="packagesToScan" value="com.apress.prospring4"/>
        <property name="jpaProperties">
           <props>
              <prop key="hibernate.dialect">
                 org.hibernate.dialect.H2Dialect
              </prop>
              <prop key="hibernate.max_fetch_depth">3</prop>
              <prop key="hibernate.jdbc.fetch_size">50</prop>
              <prop key="hibernate.jdbc.batch_size">10</prop>
              <prop key="hibernate.show_sql">true</prop>
           </props>
        </property>
    </bean>

    <context:component-scan
       base-package="com.apress.prospring4.ch9" />

    <jpa:repositories base-package="com.apress.prospring4.ch9"
       entity-manager-factory-ref="emf"
       transaction-manager-ref="transactionManager"/>
</beans>
```

Конфигурация очень похожа на вариант с аннотациями из листинга 9.9. По существу дескриптор `<tx:annotation-driven>` был удален, а в дескрипторе `<context:component-scan>` изменено имя пакета, применяемого для декларативного управления транзакциями. Наиболее важными дескрипторами являются `<aop:config>` и `<tx:advice>`.

В дескрипторе `<aop:config>` был определен срез для всех операций внутри уровня обслуживания (т.е. все классы реализации из пакета `com.apress.prospring4.ch9`). Совет ссылается на бин с идентификатором `txAdvice`, который определен посредством дескриптора `<tx:advice>`. В дескрипторе `<tx:advice>` мы конфигурируем атрибуты транзакции для различных методов, которые должны участвовать в транзакции. Как видно в этом дескрипторе, мы указываем, что все методы поиска (методы с префиксом `find`) будут предназначены только для чтения, а методы подсчета (методы с префиксом `count`) не будут принимать участие в транзакции. К остальным методам будет применяться стандартное поведение транзакции. Это та же самая конфигурация, которая использовалась в примере с аннотациями. В листинге 9.19 показан класс реализации для декларативного управления транзакциями в стиле XML.

**Листинг 9.19. Класс `ContactServiceImpl` для XML-конфигурации транзакций**

```
package com.apress.prospring4.ch9;

import com.google.common.collect.Lists;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;

import java.util.List;

@Service("contactService")
@Repository
public class ContactServiceImpl implements ContactService {
    private ContactRepository contactRepository;

    @Override
    public List<Contact> findAll() {
        return Lists.newArrayList(contactRepository.findAll());
    }

    @Override
    public Contact findById(Long id) {
        return contactRepository.findOne(id);
    }

    @Override
    public Contact save(Contact contact) {
        return contactRepository.save(contact);
    }

    @Override
    public long countAll() {
        return contactRepository.countAllContacts();
    }

    @Autowired
    public void setContactRepository(ContactRepository contactRepository) {
        this.contactRepository = contactRepository;
    }
}
```

Код в основном такой же, как в примере с аннотациями, только удалены аннотации `@Transactional`, потому что теперь связывание транзакций осуществляется с помощью АОП в Spring на основе XML-конфигурации. В листинге 9.20 приведена тестовая программа.

**Листинг 9.20. Тестирование XML-конфигурации транзакций**

```
package com.apress.prospring4.ch9;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

import com.apress.prospring4.ch9.domain.Contact;
import com.apress.prospring4.ch9.service.ContactService;

public class TxDeclarativeSample {

    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:tx-declarative-app-context.xml");
        ctx.refresh();

        ContactService contactService = ctx.getBean("contactService",
            ContactService.class);
```

```
        // Тестирование метода findAll()
        List<Contact> contacts = contactService.findAll();

        for (Contact contact: contacts) {
            System.out.println(contact);
        }

        // Тестирование метода save()
        Contact contact = contactService.findById(11);
        contact.setFirstName("Peter");
        contactService.save(contact);
        System.out.println("Contact saved successfully");

        // Тестирование метода countAll()
        System.out.println("Contact count: " + contactService.countAll());

    }
}
```

Мы оставляем запуск тестовой программы и наблюдение за выводом операций, связанных с транзакциями, которые выполняются Spring и Hibernate, для самостоятельной проработки. По большому счету, все должно выглядеть почти так же, как в примере с аннотациями.

## Использование программных транзакций

Третий вариант предусматривает управление поведением транзакций программным образом. В этом случае доступны две возможности. Первая возможность — внедрение экземпляра `PlatformTransactionManager` в бин и взаимодействие с диспетчером транзакций напрямую. Вторая возможность заключается в использовании класса `TransactionTemplate`, предоставляемого Spring, который значительно упрощает работу. В этом разделе мы продемонстрируем взаимодействие с классом `TransactionTemplate`. Для простоты мы сосредоточим внимание только на реализации метода `ContactService.countAll()`.

В листинге 9.21 приведена XML-конфигурация для использования программных транзакций (`tx-programmatic-app-context.xml`).

**Листинг 9.21. Конфигурация Spring для программного управления транзакциями**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:jpa="http://www.springframework.org/schema/data/jpa"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/data/jpa
        http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
```

```
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">
   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:META-INF/config/schema.sql"/>
      <jdbc:script location="classpath:META-INF/config/test-data.sql"/>
   </jdbc:embedded-database>
   <bean id="transactionTemplate"
      class="org.springframework.transaction.support.TransactionTemplate">
      <property name="propagationBehaviorName" value="PROPAGATION_NEVER"/>
      <property name="timeout" value="30"/>
      <property name="transactionManager" ref="transactionManager"/>
   </bean>
   <bean id="transactionManager" class="org.springframework.orm.jpa.
JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
       <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
       />
      </property>
      <property name="packagesToScan"
         value="com.apress.prospring4.ch9"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
         </props>
      </property>
   </bean>
   <context:component-scan
      base-package="com.apress.prospring4.ch9" />
   <jpa:repositories base-package="com.apress.prospring4.ch9"
      entity-manager-factory-ref="emf"
      transaction-manager-ref="transactionManager"/>
</beans>
```

Здесь совет АОП для транзакции был удален. Вдобавок был определен бин `transactionTemplate` с использованием класса `org.springframework.transaction.support.TransactionTemplate`, для которого атрибуты транзакции определены в разделе свойств. Давайте взглянем на реализацию метода `countAll()`, которая показана в листинге 9.22.

**Листинг 9.22. Реализация программного управления транзакциями**

```
package com.apress.prospring4.ch9;

import com.google.common.collect.Lists;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionTemplate;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;

@Service("contactService")
@Repository
public class ContactServiceImpl implements ContactService {
    private ContactRepository contactRepository;
    private TransactionTemplate transactionTemplate;

    @PersistenceContext
    private EntityManager em;

    @Override
    public List<Contact> findAll() {
        return Lists.newArrayList(contactRepository.findAll());
    }

    @Override
    public Contact findById(Long id) {
        return contactRepository.findOne(id);
    }

    @Override
    public Contact save(Contact contact) {
        return contactRepository.save(contact);
    }

    @Override
    public long countAll() {
        return transactionTemplate.execute(new TransactionCallback<Long>() {
            public Long doInTransaction(TransactionStatus transactionStatus) {
                return em.createNamedQuery("Contact.countAll",
                        Long.class).getSingleResult();
            }
        });
    }

    @Autowired
    public void setContactRepository(ContactRepository contactRepository) {
        this.contactRepository = contactRepository;
    }

    @Autowired
    public void setTransactionTemplate(TransactionTemplate transactionTemplate){
        this.transactionTemplate = transactionTemplate;
    }
}
```

В коде сначала был внедрен класс `TransactionTemplate`. Затем в методе `countAll()` был вызван метод `TransactionTemplate.execute()` с передачей ему объявления внутреннего класса, реализующего интерфейс `TransactionCallback<T>`.

Метод `doInTransaction()` был переопределен с применением желаемой логики. Эта логика будет работать с атрибутами, как определено бином `transactionTemplate`. Тестовая программа представлена в листинге 9.23.

**Листинг 9.23. Тестирование программных транзакций**

```
package com.apress.prospring4.ch9;

import org.springframework.context.support.GenericXmlApplicationContext;

public class TxProgrammaticSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/tx-programmatic-app-context.xml");
      ctx.refresh();

      ContactService contactService = ctx.getBean("contactService",
          ContactService.class);

      System.out.println("Contact count: " + contactService.countAll());
   }
}
```

Запуск тестовой программы и наблюдение за результатами мы оставляем в качестве самостоятельной работы. Попробуйте настроить атрибуты транзакции и посмотрите, что произойдет при обработке транзакции для метода `countAll()`.

## Соображения по поводу управления транзакциями

Итак, обсудив различные способы реализации управления транзакциями, осталось выяснить, какой из них должен использоваться. Декларативный подход рекомендуется применять во всех случаях, и по возможности следует избегать реализации управления транзакциями в коде. Когда возникает необходимость в написании логики управления транзакциями внутри приложения, чаще всего это объясняется неудачным проектным решением, и в такой ситуации нужно вынести логику в управляемые фрагменты и определить с их помощью требования к транзакциям декларативно.

При декларативном подходе стили с XML и аннотациями обладают как достоинствами, так и недостатками. Одни разработчики предпочитают не объявлять требования к транзакциям в коде, в то время как другие отдают предпочтение аннотациям за их простое сопровождение, поскольку в коде можно видеть все объявления требований к транзакциям. Здесь снова следует позволить требованиям к приложению управлять принимаемым решением, а если в команде или компании существует стандартный подход, то придерживаться соответствующего ему стиля конфигурации.

# Глобальные транзакции в Spring

Многим корпоративным Java-приложениям необходим доступ к разнообразным ресурсам серверной части. Например, порция информации о заказчике, полученная от внешнего бизнес-партнера, будет требовать обновления баз данных сразу в нескольких системах (CRM, ERP и т.д.). В ряде случаев может понадобиться формирование сообщения и его отправка серверу MQ через JMS для всех остальных приложений в компании, которые заинтересованы в информации о заказчиках. Транзакции, которые охватывают множество ресурсов серверной части, называются *глобальными* (или распределенными).

Главной характеристикой глобальной транзакции является гарантия атомарности, а это значит, что задействованные ресурсы будут либо все обновлены, либо ни один из них обновляться не будет. Диспетчер транзакций должен поддерживать сложную логику координации и синхронизации. В мире Java стандартом де-факто для реализации глобальных транзакций является JTA.

Платформа Spring поддерживает транзакции JTA так же хорошо, как и локальные транзакции, и скрывает эту логику от бизнес-кода. В этом разделе мы покажем, как реализовать глобальные транзакции, используя JTA и Spring.

## Инфраструктура для реализации примера применения JTA

Мы будем использовать ту же таблицу, что и в предыдущих примерах этой главы. Однако встроенная база данных H2 не полностью поддерживает XA (во всяком случае, на момент написания этой книги), поэтому в настоящем примере в качестве базы данных серверной части будет применяться MySQL.

Мы также хотим показать, каким образом реализовать глобальные транзакции с помощью JTA в автономном приложении или в среде веб-контейнера. Таким образом, в этом примере мы будем использовать продукт Atomikos (`http://www.atomikos.com/Main/TransactionsEssentials`), который представляет собой широко применяемый диспетчер транзакций JTA с открытым кодом, предназначенный для сред, отличных от JEE.

Чтобы посмотреть, как работают глобальные транзакции, необходимо иметь, по крайней мере, два ресурса серверной части. Для простоты мы будем использовать одну базу данных MySQL, но два диспетчера сущностей JPA. Требование наличия нескольких ресурсов будет достигнуто, т.к. имеется несколько единиц постоянства JPA для различения баз данных серверной части.

В базе данных MySQL мы создадим две схемы, описанные в табл. 9.5.

**Таблица 9.5. Схемы базы данных MySQL**

| Имя схемы | Информация подключения | Сценарий для наполнения данными |
|---|---|---|
| `prospring4_ch9a` | Имя пользователя: `prospring4_ch9a`<br>Пароль: `prospring4_ch9a` | `schema.sql`<br>`test-data.sql` |
| `prospring4_ch9b` | Имя пользователя: `prospring4_ch9b`<br>Пароль: `prospring4_ch9b` | `schema.sql`<br>`test-data.sql` |

После этого к проекту потребуется добавить обязательные зависимости для MySQL и Atomikos. Эти зависимости перечислены в табл. 9.6.

**Таблица 9.6. Зависимости Maven для MySQL и Atomikos**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| mysql | mysql-connector-java | 5.1.29 | Java-библиотека для MySQL 5 |
| com.atomikos | transactions-jdbc | 3.9.3 | Библиотека транзакций JTA для Atomikos |

После завершения всех настроек можно приступать к конфигурированию и реализации.

## Реализация глобальных транзакций с помощью JTA

Сначала давайте рассмотрим конфигурацию Spring. В листинге 9.24 показано содержимое файла `tx-jta-app-context.xml`.

**Листинг 9.24. Конфигурация Spring для JTA**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">
   <bean id="dataSourceA" class="com.atomikos.jdbc.AtomikosDataSourceBean"
        init-method="init" destroy-method="close">
      <property name="uniqueResourceName" value="XADBMSA"/>
      <property name="xaDataSourceClassName"
                value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"/>
      <property name="xaProperties">
         <props>
            <prop key="databaseName">prospring4_ch9a</prop>
            <prop key="user">prospring4_ch9a</prop>
            <prop key="password">prospring4_ch9a</prop>
         </props>
      </property>
      <property name="poolSize" value="1"/>
   </bean>
   <bean id="dataSourceB" class="com.atomikos.jdbc.AtomikosDataSourceBean"
        init-method="init" destroy-method="close">
      <property name="uniqueResourceName" value="XADBMSB"/>
      <property name="xaDataSourceClassName"
                value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"/>
      <property name="xaProperties">
         <props>
            <prop key="databaseName">prospring4_ch9b</prop>
            <prop key="user">prospring4_ch9b</prop>
            <prop key="password">prospring4_ch9b</prop>
```

```
        </props>
      </property>
      <property name="poolSize" value="1"/>
   </bean>

   <bean id="emfBase"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
        abstract="true">
      <property name="jpaVendorAdapter">
       <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"/>
      </property>
      <property name="packagesToScan" value="com.apress.prospring4.ch9"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.transaction.factory_class">
             org.hibernate.engine.transaction.internal.jta.CMTTransactionFactory
            </prop>
            <prop key="hibernate.transaction.manager_lookup_class">
              com.atomikos.icatch.jta.hibernate3.TransactionManagerLookup
            </prop>
            <prop key="hibernate.dialect">
               org.hibernate.dialect.MySQL5Dialect</prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
         </props>
      </property>
   </bean>
   <bean id="emfA" parent="emfBase">
      <property name="dataSource" ref="dataSourceA"/>
      <property name="persistenceUnitName" value="emfA"/>
   </bean>
   <bean id="emfB" parent="emfBase">
      <property name="dataSource" ref="dataSourceB"/>
      <property name="persistenceUnitName" value="emfB"/>
   </bean>
   <tx:annotation-driven transaction-manager="transactionManager" />
   <bean id="atomikosTransactionManager"
         class="com.atomikos.icatch.jta.UserTransactionManager"
         init-method="init" destroy-method="close">
      <property name="forceShutdown" value="true"/>
   </bean>
   <bean id="atomikosUserTransaction"
         class="com.atomikos.icatch.jta.UserTransactionImp">
      <property name="transactionTimeout" value="300"/>
   </bean>
   <bean id="transactionManager"
         class="org.springframework.transaction.jta.JtaTransactionManager">
      <property name="transactionManager" ref="atomikosTransactionManager"/>
      <property name="userTransaction" ref="atomikosUserTransaction"/>
   </bean>
   <context:component-scan base-package="com.apress.prospring4.ch9 " />
</beans>
```

Конфигурация обширна, но не особо сложна. Первым делом, определены два бина для источников данных, указывающие два разных ресурса баз данных. Бины имеют имена dataSourceA и dataSourceB и подключаются к схемам prospring4_ch9a и prospring4_ch9b. Оба бина для источников данных используют класс com.atomikos.jdbc.AtomikosDataSourceBean, который поддерживает совместимый с XA источник данных, и внутри определений этих бинов определен класс реализации источника данных XA типа MySQL (com.mysql.jdbc.jdbc2.optional.MysqlXADataSource); он представляет собой диспетчер ресурсов для MySQL. Затем предоставляется информация подключения к базе данных. Обратите внимание, что атрибут poolSize задает количество подключений в пуле подключений, который должен поддерживаться Atomikos. Это не является обязательным. Если этот атрибут не указан, Atomikos будет применять стандартное значение, равное 1.

В части, связанной с Atomikos, определены два бина, atomikosTransactionManager и atomikosUserTransaction. Классы реализации предоставляются Atomikos и реализуют интерфейсы TransactionManager и UserTransaction из стандарта JEE. Указанные бины предлагают службы по координации и синхронизации транзакций, требуемые JTA, и взаимодействуют с диспетчерами ресурсов по протоколу XA при поддержке 2PC.

После этого определяется Spring-бин transactionManager (с классом реализации JtaTransactionManager) и внедряются два бина транзакций, предоставляемые Atomikos. Это информирует Spring о необходимости использования Atomikos JTA для управления транзакциями.

Далее определяются три бина EntityManagerFactory с именами emfBase, emfA и emfB. Бин emfBase — это абстрактный родительский бин, который является оболочкой для общих свойств JPA. Бин emfBase реализован с помощью класса LocalContainerEntityManagerFactoryBean из Spring. Бины emfA и emfB наследуют конфигурацию от родительского бина emfBase и отличаются только тем, что в них внедряются разные источники данных (т.е. dataSourceA внедряется в emfA, а dataSourceB — в emfB). Следовательно, emfA будет подключаться к MySQL-схеме prospring4_ch9a через бин dataSourceA, а emfB — подключаться к схеме prospring4_ch9b через бин dataSourceB.

Взгляните на свойства hibernate.transaction.factory_class и hibernate.transaction.manager_lookup_class в бине emfBase. Эти два свойства очень важны, т.к. они применяются Hibernate для поиска интерфейсов UserTransaction и TransactionManager, чтобы участвовать в контексте постоянства, который управляется в глобальной транзакции.

В листинге 9.25 приведен код класса ContactServiceImpl для JTA. Обратите внимание, что для простоты реализован только метод save().

**Листинг 9.25. Класс реализации ContactService для JTA**

```
package com.apress.prospring4.ch9;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.PersistenceException;
```

```
import org.springframework.orm.jpa.JpaSystemException;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
   @PersistenceContext(unitName="emfA")
   private EntityManager emA;

   @PersistenceContext(unitName="emfB")
   private EntityManager emB;

   @Override
   @Transactional(readOnly=true)
   public List<Contact> findAll() {
      return null;
   }

   @Override
   @Transactional(readOnly=true)
   public Contact findById(Long id) {
      return null;
   }

   @Override
   public Contact save(Contact contact) {
      Contact contactB = new Contact();
      contactB.setFirstName(contact.getFirstName());
      contactB.setLastName(contact.getLastName());
      if (contact.getId() == null) {
         emA.persist(contact);
         emB.persist(contactB);
         // throw new JpaSystemException(new PersistenceException());
      } else {
         emA.merge(contact);
         emB.merge(contact);
      }

      return contact;
   }

   @Override
   public long countAll() {
      return 0;
   }
}
```

В коде определены два диспетчера сущностей, которые внедрены в класс `ContactServiceImpl`. В методе `save()` мы сохраняем объект контакта в две схемы. Пока что проигнорируйте оператор `throw` для исключения; мы будем использовать его позже для проверки, произошел ли откат транзакции в случае сбоя сохранения в схему `prospring4_ch9b`. В листинге 9.26 представлен код тестовой программы.

**Листинг 9.26. Тестирование транзакции JTA**

```
package com.apress.prospring4.ch9;

import org.springframework.context.support.GenericXmlApplicationContext;

public class TxJtaSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/tx-jta-app-context.xml");
      ctx.refresh();
      ContactService contactService = ctx.getBean("contactService",
          ContactService.class);
      Contact contact = new Contact();
      contact.setFirstName("Jta");
      contact.setLastName("Manager");
      contactService.save(contact);
      System.out.println("Contact saved successfully");
   }
}
```

Эта программа создает новый объект контакта и вызывает метод `ContactService.save()`. Реализация попытается сохранить один и тот же объект в двух базах данных. При условии, что не произойдет ничего непредвиденного, запуск программы дает следующий вывод (который показан не полностью):

```
Contact saved successfully
```

Продукт Atomikos создает составную транзакцию, взаимодействует с источником данных XA (в рассматриваемом случае MySQL), выполняет синхронизацию, фиксирует транзакцию и т.д.

Новый контакт сохраняется в обеих схемах базы данных.

А теперь посмотрим, каким образом работает откат. Как было показано в листинге 9.25, вместо вызова `emB.persist()` можно просто сгенерировать исключение. Соответствующий фрагмент кода показан в листинге 9.27.

**Листинг 9.27. Тестирование отката транзакции JTA**

```
//emB.persist(contactB);
throw new JpaSystemException(new PersistenceException());
```

Чтобы протестировать сценарий с откатом, сначала удалите новые записи, вставленные предыдущим примером, из двух баз данных MySQL (т.е. запись контакта с полем `FIRST_NAME`, имеющим значение `Jta`, и полем `LAST_NAME`, установленным в `Manager`). Запуск программы теперь дает следующие результаты:

```
Exception in thread "main" org.springframework.orm.jpa.JpaSystemException:
nested exception is javax.persistence.PersistenceException
...
Caused by: javax.persistence.PersistenceException
В потоке main возникло исключение org.springframework.orm.jpa.
JpaSystemException: вложенным исключением является javax.persistence.
PersistenceException
...
Вызвано: javax.persistence.PersistenceException
```

Первый контакт был сохранен, но при сохранении во втором источнике данных генерируется исключение, из-за чего Atomikos производит откат всей транзакции. Можете просмотреть схему `prospring4_ch9a` и удостовериться, что новый контакт не был вставлен.

### Соображения по поводу использования диспетчера транзакций JTA

По поводу необходимости использования JTA для управления глобальными транзакциями, ведутся бурные обсуждения. Например, команда разработчиков Spring не рекомендует применять JTA для глобальных транзакций.

Общий принцип состоит в том, что когда приложение разворачивается на полномасштабном сервере приложений JEE, то нет никаких причин не применять JTA, поскольку все производители популярных серверов приложений JEE оптимизируют собственные реализации JTA для своих платформ. Это одно из главных функциональных средств, за которые вы платите.

В случае автономной версии или развертывания в веб-контейнере позвольте требованиям приложения управлять принятием решения. Проводите нагрузочное тестирование как можно раньше, удостоверяясь, что использование JTA не приводит к снижению производительности.

Хорошая новость заключается в том, что Spring гладко работает с локальными и глобальными транзакциями в большинстве основных веб- и JEE-контейнеров, поэтому при переходе от одной стратегии управления транзакциями к другой обычно модификация кода не требуется. Если вы решите применять JTA в своем приложении, то используйте `JtaTransactionManager` из Spring.

## Резюме

Управление транзакциями является ключевым аспектом обеспечения целостности данных в практически любом типе приложения. В этой главе мы обсудили, как использовать Spring для управления транзакциями, почти не влияя на исходный код приложений. Было также показано, как работать с локальными и глобальными транзакциями.

Мы рассмотрели множество примеров реализации транзакций, в том числе декларативные способы применения XML-конфигурации и аннотаций, а также программный подход.

Локальные транзакции поддерживаются внутри или за пределами сервера приложений JEE, а для включения поддержки локальных транзакций в Spring требуется лишь простое конфигурирование. Тем не менее, настройка среды для глобальной транзакции предусматривает больше работы и сильно зависит от того, с каким поставщиком JTA и соответствующими ресурсами серверной части должно взаимодействовать приложение.

ГЛАВА 10

# Проверка достоверности с преобразованием типов и форматированием

В корпоративном приложении *проверка достоверности* критически важна. Проверка достоверности предназначена для верификации того, что обрабатываемые данные удовлетворяют всем заранее определенным бизнес-требованиям, а также для обеспечения целостности и пригодности данных на других уровнях приложения.

При разработке приложений проверка достоверности данных всегда упоминается вместе с преобразованием и форматированием. Причина в том, что формат источника данных, скорее всего, отличается от формата, используемого в приложении. Например, в веб-приложении пользователь вводит информацию внутри пользовательского интерфейса веб-браузера. Когда пользователь сохраняет эти данные, они отправляются серверу (после завершения локальной проверки достоверности). На стороне сервера выполняется процесс привязки данных, при котором данные извлекаются из HTTP-запроса, преобразуются и привязываются к соответствующим объектам предметной области (к примеру, пользователь вводит контактную информацию в HTML-форме, которая затем привязывается к объекту `Contact` на сервере) на основе правил форматирования, определенных для каждого атрибута (шаблон формата даты может выглядеть, скажем, как `yyyy-MM-dd`). После завершения привязки данных к объекту предметной области применяются правила проверки достоверности для поиска любого нарушения ограничений. Если все проходит нормально, данные сохраняются, а пользователю отображается сообщение об успехе операции. В противном случае пользователю отображаются сообщения об ошибках проверки достоверности.

Вы узнаете, что Spring предлагает развитую поддержку для преобразования типов, форматирования полей и проверки достоверности. В частности, в главе рассматриваются следующие темы.

- **Система преобразования типов Spring и интерфейс поставщика службы форматировщиков.** Мы обсудим обобщенную систему преобразования типов и интерфейс поставщика службы (service provider interface — SPI) форматировщиков (Formatter SPI). Мы покажем, как эти службы можно использовать для замены предшествующей поддержки редакторов свойств, и каким образом они выполняют преобразования между любыми Java-типами.
- **Проверка достоверности в Spring.** Мы посмотрим, как Spring поддерживает проверку достоверности объектов предметной области. Сначала мы предложим краткое введение в собственный интерфейс Spring по имени `Validator`. После этого мы сосредоточим внимание на поддержке спецификации JSR-349: Bean Validation (JSR-349: Проверка достоверности бинов).

# Зависимости

Как и в предыдущих главах, примеры кода, представленные в этой главе, требуют ряда зависимостей, которые перечислены в табл. 10.1. Вы можете обратить внимание на зависимость `joda-time`. На тот случай, если вы работаете с Java 8, версия Spring 4 также поддерживает спецификацию JSR-310, которая описывает API-интерфейс `javax.time`.

**Таблица 10.1. Зависимости Maven для проверки достоверности**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `javax.validation` | `validation-api` | 1.1.0.Final | Библиотека API-интерфейса JSR-349 |
| `org.hibernate` | `hibernate-validator` | 5.1.0.Final | Библиотека Hibernate Validator, которая поддерживает спецификацию JSR-349: Bean Validation |
| `joda-time` | `joda-time` | 2.3 | API-интерфейс даты и времени, предназначенный для упрощения взаимодействия со встроенными библиотеками Java, которые работают с датой и временем. В этой главе мы будем применять его в объектах предметной области |

# Система преобразования типов Spring

В версии Spring 3 появилась новая система преобразования типов, предоставляющая мощный метод для выполнения преобразований между любыми Java-типами внутри Spring-приложений. В этом разделе мы покажем, что эта новая служба реализует такую же функциональность, как и ранее использовавшиеся редакторы свойств, и объясним ее поддержку преобразований между любыми Java-типами. Мы также продемонстрируем реализацию специального преобразователя типов с применением Converter SPI.

## Преобразование строковых значений с использованием редакторов свойств

В главе 4 было показано, что Spring обрабатывает преобразование значений типа `String`, заданных внутри файлов свойств, в свойства объектов POJO за счет поддержки редакторов свойств (`PropertyEditor`). Мы приведем здесь краткий обзор, а затем рассмотрим интерфейс Converter SPI (доступный, начиная с версии Spring 3.0), который предлагает более мощную альтернативу.

Пусть имеется класс `Contact` с несколькими атрибутами, показанный в листинге 10.1.

**Листинг 10.1. Класс `Contact`**

```
package com.apress.prospring4.ch10;

import java.net.URL;
import org.joda.time.DateTime;

public class Contact {
    private String firstName;
    private String lastName;
    private DateTime birthDate;
    private URL personalSite;
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public DateTime getBirthDate() {
        return birthDate;
    }
    public void setBirthDate(DateTime birthDate) {
        this.birthDate = birthDate;
    }
    public URL getPersonalSite() {
        return personalSite;
    }
    public void setPersonalSite(URL personalSite) {
        this.personalSite = personalSite;
    }
    public String toString() {
        return "First name: " + getFirstName()
                + " - Last name: " + getLastName()
                + " - Birth date: " + getBirthDate()
                + " - Personal site: " + getPersonalSite();
    }
}
```

Для атрибута дня рождения (birthDate) используется класс DateTime из Joda-Time. Вдобавок имеется поле типа URL, указывающее персональный веб-сайт для контакта, если это применимо.

Теперь предположим, что необходимо конструировать объекты Contact в ApplicationContext с использованием значений, хранящихся либо в конфигурационном файле Spring, либо в файле свойств. В листинге 10.2 приведено содержимое XML-файла конфигурации Spring (prop-editor-app-context.xml).

**Листинг 10.2. Конфигурация Spring для редактора свойств**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:util="http://www.springframework.org/schema/util"
      xmlns:p="http://www.springframework.org/schema/p"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util.xsd">

   <context:annotation-config/>

   <context:property-placeholder location="classpath:application.properties"/>

   <bean id="customEditorConfigurer"
     class="org.springframework.beans.factory.config.CustomEditorConfigurer"
     p:propertyEditorRegistrars-ref="propertyEditorRegistrarsList"/>

   <util:list id="propertyEditorRegistrarsList">
      <bean class="com.apress.prospring4.ch10.DateTimeEditorRegistrar">
         <constructor-arg value="${date.format.pattern}"/>
      </bean>
   </util:list>

   <bean id="chris" class="com.apress.prospring4.ch10.Contact"
        p:firstName="Chris"
        p:lastName="Schaefer"
        p:birthDate="1981-05-03"
        p:personalSite="http://www.dtzq.com"/>

   <bean id="myContact" class="com.apress.prospring4.ch10.Contact"
        p:firstName="${myContact.firstName}"
        p:lastName="${myContact.lastName}"
        p:birthDate="${myContact.birthDate}"
        p:personalSite="${myContact.personalSite}"/>
</beans>
```

Здесь мы конструируем два разных бина из класса Contact. Бин chris создан со значениями, предоставленными в конфигурационном файле, в то время как атрибуты бина myContact вынесены во внешний файл свойств. Кроме того, определен специальный редактор, предназначенный для преобразования значений String в тип DateTime из Joda-Time, а шаблон формата даты и времени также вынесен во

внешний файл свойств. Содержимое файла свойств (`application.properties`) показано в листинге 10.3.

**Листинг 10.3. Файл `application.properties`**

```
date.format.pattern=yyyy-MM-dd

myContact.firstName=Scott
myContact.lastName=Tiger
myContact.birthDate=1984-6-30
myContact.personalSite=http://www.somedomain.com
```

В листинге 10.4 представлен специальный редактор для преобразования значений `String` в тип `DateTime` из Joda-Time.

**Листинг 10.4. Специальный редактор для типа `DateTime` из Joda-Time**

```
package com.apress.prospring4.ch10;

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
import org.springframework.beans.PropertyEditorRegistrar;
import org.springframework.beans.PropertyEditorRegistry;

import java.beans.PropertyEditorSupport;

public class DateTimeEditorRegistrar implements PropertyEditorRegistrar {
    private DateTimeFormatter dateTimeFormatter;

    public DateTimeEditorRegistrar(String dateFormatPattern) {
        dateTimeFormatter = DateTimeFormat.forPattern(dateFormatPattern);
    }

    @Override
    public void registerCustomEditors(PropertyEditorRegistry registry) {
        registry.registerCustomEditor(DateTime.class,
                                      new DateTimeEditor(dateTimeFormatter));
    }

    private static class DateTimeEditor extends PropertyEditorSupport {
        private DateTimeFormatter dateTimeFormatter;

        public DateTimeEditor(DateTimeFormatter dateTimeFormatter) {
            this.dateTimeFormatter = dateTimeFormatter;
        }

        @Override
        public void setAsText(String text) throws IllegalArgumentException {
            setValue(DateTime.parse(text, dateTimeFormatter));
        }
    }
}
```

В листинге 10.4 мы реализуем интерфейс `PropertyEditorRegister` для регистрации специального класса `PropertyEditor`. Затем мы создаем внутренний класс по имени `DateTimeEditor`, который обрабатывает преобразование значения типа `String` в `DateTime`. В этом примере применяется внутренний класс, т.к. доступ к

нему производится только нашей реализацией PropertyEditorRegistrar. Теперь давайте его протестируем. Тестовая программа приведена в листинге 10.5.

**Листинг 10.5. Тестирование редактора свойств**

```
package com.apress.prospring4.ch10;

import org.springframework.context.support.GenericXmlApplicationContext;

public class PropEditorExample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/prop-editor-app-context.xml");
      ctx.refresh();

      Contact chris = ctx.getBean("chris", Contact.class);
      System.out.println("Chris info: " + chris);
      Contact myContact = ctx.getBean("myContact", Contact.class);
      System.out.println("My contact info: " + myContact);
   }
}
```

Как показано в листинге 10.5, из ApplicationContext извлекаются два бина Contact, которые затем отображаются. Запуск этой программы дает следующий вывод:

```
Chris info: First name: Chris - Last name: Schaefer - Birth date:
1981-05-03T00:00:00.000-04:00 - Personal site: http://www.dtzq.com
My contact info: First name: Scott - Last name: Tiger - Birth date:
1984-06-30T00:00:00.000-04:00 - Personal site: http://www.somedomain.com
```

В выводе видно, что свойства были преобразованы и применены к бинам Contact.

## Введение в систему преобразования типов Spring

Начиная с версии Spring 3.0, стала доступной обобщенная система преобразования типов, которая находится в пакете org.springframework.core.convert. В дополнение к предоставлению альтернативы редакторам свойств, систему преобразования типов можно сконфигурировать для выполнения преобразований между Java-типами и объектами POJO (тогда как редакторы свойств ориентированы на преобразование в Java-типы строковых представлений из файла свойств).

### *Реализация специального преобразователя*

Чтобы посмотреть на систему преобразования типов в действии, давайте вернемся к предыдущему примеру и воспользуемся тем же самым классом Contact. На этот раз предположим, что систему преобразования типов нужно применять для преобразования даты в формате String в свойство birthDate класса Contact, которое имеет тип DateTime из Joda-Time. Для поддержки такого преобразования вместо специального редактора свойства мы создаем специальный преобразователь, реализуя интерфейс org.springframework.core.convert.converter.Converter<S,T>. Код этого специального преобразователя приведен в листинге 10.6.

**Листинг 10.6. Специальный преобразователь `DateTime`**

```
package com.apress.prospring4.ch10;

import javax.annotation.PostConstruct;

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.convert.converter.Converter;

public class StringToDateTimeConverter implements Converter<String, DateTime>
{
    private static final String DEFAULT_DATE_PATTERN = "yyyy-MM-dd";
    private DateTimeFormatter dateFormat;

    private String datePattern = DEFAULT_DATE_PATTERN;

    public String getDatePattern() {
        return datePattern;
    }

    @Autowired(required=false)
    public void setDatePattern(String datePattern) {
        this.datePattern = datePattern;
    }

    @PostConstruct
    public void init() {
        dateFormat = DateTimeFormat.forPattern(datePattern);
    }

    @Override
    public DateTime convert(String dateString) {
        return dateFormat.parseDateTime(dateString);
    }
}
```

Мы реализуем интерфейс `Converter<String, DateTime>`, а это значит, что преобразователь отвечает за преобразование типа `String` (исходный тип `S`) в тип `DateTime` (целевой тип `T`). Внедрение шаблона даты-времени является необязательным за счет снабжения его аннотацией `@Autowired(required=false)`. Если внедрение шаблона не производилось, то используется стандартный шаблон `yyyy-MM-dd`. После этого в методе инициализации (метод `init()`, аннотированный посредством `@PostConstruct`) конструируется экземпляр класса `DateTimeFormat` из Joda-Time, который выполнит преобразование на основе указанного шаблона. Наконец, в коде реализован метод `convert()`, содержащий логику преобразования.

### *Конфигурирование `ConversionService`*

Для применения службы преобразования вместо редактора свойства понадобится сконфигурировать экземпляр интерфейса `org.springframework.core.convert.ConversionService` в `ApplicationContext`. Содержимое конфигурационного файла (`conv-service-app-context.xml`) приведено в листинге 10.7.

**Листинг 10.7. Конфигурация службы преобразования**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">
   <context:annotation-config/>
   <bean id="conversionService"
class="org.springframework.context.support.ConversionServiceFactoryBean">
      <property name="converters">
         <list>
          <bean class="com.apress.prospring4.ch10.StringToDateTimeConverter"/>
         </list>
      </property>
   </bean>
   <bean id="chris" class="com.apress.prospring4.ch10.Contact"
      p:firstName="Chris"
      p:lastName="Schaefer"
      p:birthDate="1981-05-03"
      p:personalSite="http://www.dtzq.com"/>
</beans>
```

Здесь мы указываем Spring, что необходимо использовать систему преобразования типов, объявив бин `conversionService` с классом `ConversionServiceFactoryBean`. Если ни одного бина службы преобразования не определено, Spring будет работать с системой на основе редакторов свойств.

По умолчанию система преобразования типов поддерживает преобразование между общими типами, такими как строки, числа, перечисления, коллекции, карты и т.д. Кроме того, поддерживается преобразование значений `String` в Java-типы внутри системы, основанной на редакторах свойств.

В бине `conversionService` сконфигурирован специальный преобразователь из `String` в `DateTime`. В листинге 10.8 показана тестовая программа.

**Листинг 10.8. Тестирование службы преобразования**

```
package com.apress.prospring4.ch10;

import org.springframework.context.support.GenericXmlApplicationContext;

public class ConvServExample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/conv-service-app-context.xml");
      ctx.refresh();
      Contact chris = ctx.getBean("chris", Contact.class);
      System.out.println("Contact info: " + chris);
   }
}
```

Запуск этой программы дает следующий вывод:

```
Contact info: First name: Chris - Last name: Schaefer - Birth date:
1981-05-03T00:00:00.000-04:00 - Personal site: http://www.dtzq.com
```

Как видите, результат преобразования свойства бина `chris` такой же, как в случае применения редакторов свойств.

### *Преобразование между произвольными типами*

Реальная мощь системы преобразования типов заключена в ее возможности выполнять преобразования между произвольными типами. Чтобы увидеть это в действии, создадим еще один класс по имени `AnotherContact`, который совпадает с классом `Contact`. Его код представлен в листинге 10.9.

**Листинг 10.9. Класс `AnotherContact`**

```
package com.apress.prospring4.ch10;
import java.net.URL;
import org.joda.time.DateTime;
public class AnotherContact {
    private String firstName;
    private String lastName;
    private DateTime birthDate;
    private URL personalSite;
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public DateTime getBirthDate() {
        return birthDate;
    }
    public void setBirthDate(DateTime birthDate) {
        this.birthDate = birthDate;
    }
    public URL getPersonalSite() {
        return personalSite;
    }
    public void setPersonalSite(URL personalSite) {
        this.personalSite = personalSite;
    }
    public String toString() {
        return "First name: " + getFirstName()
                + " - Last name: " + getLastName()
                + " - Birth date: " + getBirthDate()
                + " - Personal site: " + getPersonalSite();
    }
}
```

Нам нужна возможность преобразования любого экземпляра класса `Contact` в экземпляр класса `AnotherContact`.

При преобразовании атрибуты `firstName` и `lastName` из `Contact` станут атрибутами `lastName` и `firstName` в `AnotherContact`. Давайте реализуем для этого специальный преобразователь. Его код показан в листинге 10.10.

**Листинг 10.10. Класс `ContactToAnotherContactConverter`**

```
package com.apress.prospring4.ch10;

import org.springframework.core.convert.converter.Converter;

public class ContactToAnotherContactConverter
    implements Converter<Contact, AnotherContact> {

    @Override
    public AnotherContact convert(Contact contact) {
        AnotherContact anotherContact = new AnotherContact();
        anotherContact.setFirstName(contact.getLastName());
        anotherContact.setLastName(contact.getFirstName());
        anotherContact.setBirthDate(contact.getBirthDate());
        anotherContact.setPersonalSite(contact.getPersonalSite());

        return anotherContact;
    }
}
```

Этот класс очень прост; он лишь меняет местами значения атрибутов `firstName` и `lastName` в классах `Contact` и `AnotherContact`. Чтобы зарегистрировать специальный преобразователь в `ApplicationContext`, замените определение бина `conversionService` в файле `conv-service-app-context.xml` фрагментом кода, приведенным в листинге 10.11.

**Листинг 10.11. Добавление специального преобразователя к службе преобразования**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">
    <context:annotation-config/>
    <bean id="conversionService"
  class="org.springframework.context.support.ConversionServiceFactoryBean">
        <property name="converters">
            <list>
             <bean class="com.apress.prospring4.ch10.StringToDateTimeConverter"/>
                <bean class="com.apress.prospring4.ch10.
                    ContactToAnotherContactConverter"/>
            </list>
        </property>
    </bean>
```

```
    <bean id="chris" class="com.apress.prospring4.ch10.Contact"
        p:firstName="Chris"
        p:lastName="Schaefer"
        p:birthDate="1981-05-03"
        p:personalSite="http://www.dtzq.com"/>
</beans>
```

Порядок следования бинов в свойстве `converters` не важен.

Для тестирования преобразования мы воспользуемся тестовой программой из предыдущего примера, т.е. классом `ConvServExample`. В листинге 10.12 показан модифицированный метод `main()`.

**Листинг 10.12. Тестирование службы преобразования**

```
package com.apress.prospring4.ch10;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.core.convert.ConversionService;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class ConvServExample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/conv-service-app-context.xml");
        ctx.refresh();

        Contact chris = ctx.getBean("chris", Contact.class);

        System.out.println("Contact info: " + chris);

        ConversionService conversionService = ctx.getBean(ConversionService.class);

        AnotherContact anotherContact =
            conversionService.convert(chris, AnotherContact.class);
        System.out.println("Another contact info: " + anotherContact);

        String[] stringArray = conversionService.convert("a,b,c", String[].class);

        System.out.println("String array: " + stringArray[0] +
            stringArray[1] + stringArray[2]);

        List<String> listString = new ArrayList<String>();
        listString.add("a");
        listString.add("b");
        listString.add("c");

        Set<String> setString = conversionService.convert(listString, HashSet.class);

        for (String string: setString)
            System.out.println("Set: " + string);
        }
}
```

В этом коде из `ApplicationContext` получается обработчик интерфейса `ConversionService`. Поскольку мы уже зарегистрировали `ConversionService` в

ApplicationContext со специальными преобразователями, его можно применять для преобразования объекта Contact, а также для преобразований между другими типами, которые поддерживает служба преобразования. Для демонстрационных целей в код были добавлены примеры преобразования из String (разделенных запятыми) в Array и из List в Set.

Запуск этой программы дает следующий вывод:

```
Contact info: First name: Chris - Last name: Schaefer - Birth date:
1981-05-03T00:00:00.000-04:00 - Personal site: http://www.dtzq.com
Another contact info: First name: Schaefer - Last name: Chris - Birth date:
1981-05-03T00:00:00.000-04:00 - Personal site: http://www.dtzq.com
String array: abc
Set: a
Set: b
Set: c
```

В выводе видно, что преобразования между Contact и AnotherContact выполнены корректно, и то же самое можно сказать о преобразованиях String в Array и List в Set. Для службы преобразования типов Spring можно легко создавать специальные преобразователи и осуществлять преобразования на любом уровне приложения. Возможный сценарий использования предусматривает наличие двух систем с одной и той же информацией о контакте, которую необходимо обновить. Однако структуры баз данных в этих системах отличаются (например, фамилия в одной системе означает имя в другой и т.п.). Службу преобразования типов можно применять для преобразования объектов перед их сохранением в каждой отдельной системе.

Начиная с версии Spring 3.0, инфраструктура Spring MVC широко использует службу преобразования (а также интерфейс Formatter SPI, который обсуждается в следующем разделе).

В конфигурации контекста веб-приложения объявление дескриптора <mvc:annotation-driven/> автоматически регистрирует все стандартные преобразователи (к примеру, StringToArrayConverter, StringToBooleanConverter и StringToLocaleConverter из пакета org.springframework.core.convert.support) и форматировщики (например, CurrencyFormatter, DateFormatter и NumberFormatter из различных подпакетов внутри пакета org.springframework.format). Дополнительные сведения по этому поводу будут представлены в главе 16, когда будет обсуждаться разработка веб-приложений в Spring.

# Форматирование полей в Spring

Помимо системы преобразования типов, разработчикам Spring-приложений доступно также другое великолепное средство — Formatter SPI. Как и можно было ожидать, этот SPI-интерфейс помогает настраивать все аспекты, связанные с форматированием полей.

Главным интерфейсом в Formatter SPI, предназначенным для реализации форматировщиков, является org.springframework.format.Formatter<T>. Платформа Spring предлагает несколько его реализаций для часто применяемых типов, среди которых CurrencyFormatter, DateFormatter, NumberFormatter и PercentFormatter.

## Реализация специального форматировщика

Реализация специального форматировщика довольно проста. Мы снова воспользуемся классом `Contact` и реализуем специальный форматировщик для преобразования типа `DateTime` атрибута `birthDate` в и из `String`.

Однако в этот раз мы применим другой подход — расширим класс `org.springframework.format.support.FormattingConversionServiceFactoryBean` из Spring и предоставим специальный форматировщик.

Класс `FormattingConversionServiceFactoryBean` — это фабричный класс, обеспечивающий удобный доступ к лежащему в основе классу `FormattingConversionService`, который поддерживает систему преобразования типов, а также форматирование полей в соответствии с правилами форматирования, определенными для каждого типа поля.

В листинге 10.13 приведен код специального класса, расширяющего класс `FormattingConversionServiceFactoryBean`, со специальным форматировщиком, который определен для форматирования типа `DateTime` из Joda-Time.

**Листинг 10.13. Класс `ApplicationConversionServiceFactoryBean`**

```
package com.apress.prospring4.ch10;

import java.text.ParseException;
import java.util.HashSet;
import java.util.Locale;
import java.util.Set;

import javax.annotation.PostConstruct;

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.format.Formatter;
import org.springframework.format.support.FormattingConversionServiceFactoryBean;

public class ApplicationConversionServiceFactoryBean extends
    FormattingConversionServiceFactoryBean {
    private static final String DEFAULT_DATE_PATTERN = "yyyy-MM-dd";
    private DateTimeFormatter dateFormat;
    private String datePattern = DEFAULT_DATE_PATTERN;
    private Set<Formatter<?>> formatters = new HashSet<Formatter<?>>();
    public String getDatePattern() {
        return datePattern;
    }
    @Autowired(required=false)
    public void setDatePattern(String datePattern) {
        this.datePattern = datePattern;
    }
    @PostConstruct
    public void init() {
        dateFormat = DateTimeFormat.forPattern(datePattern);
        formatters.add(getDateTimeFormatter());
        setFormatters(formatters);
    }
```

```
    public Formatter<DateTime> getDateTimeFormatter() {
        return new Formatter<DateTime>() {
            @Override
            public DateTime parse(String dateTimeString, Locale locale)
                    throws ParseException {
                System.out.println("Parsing date string: " + dateTimeString);
                return dateFormat.parseDateTime(dateTimeString);
            }
            @Override
            public String print(DateTime dateTime, Locale locale) {
                System.out.println("Formatting datetime: " + dateTime);
                return dateFormat.print(dateTime);
            }
        };
    }
}
```

В предыдущем листинге код специального форматировщика выделен полужирным. Он реализует интерфейс `Formatter<DateTime>` и два метода, определенные этим интерфейсом. Метод `parse()` разбирает формат `String` и строит тип `DateTime` (для поддержки локализации также передается нужная локаль), а метод `print()` форматирует экземпляр `DateTime` в `String`. Шаблон даты может быть внедрен в бин (или же будет использоваться шаблон по умолчанию вида `yyyy-MM-dd`). Кроме того, в методе `init()` производится регистрация специального форматировщика с помощью вызова метода `setFormatters()`. Можно добавлять столько форматировщиков, сколько требуется в приложении.

## Конфигурирование ConversionServiceFactoryBean

Чтобы сконфигурировать `ApplicationConversionServiceFactoryBean` в `ApplicationContext`, нужно всего лишь объявить бин с этим классом в качестве поставщика. В листинге 10.14 показано содержимое конфигурационного файла `conv-format-service-app-context.xml`.

**Листинг 10.14. Файл conv-format-service-app-context.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">
    <context:annotation-config/>
    <bean id="conversionService"
class="com.apress.prospring4.ch10.ApplicationConversionServiceFactoryBean"/>
    <bean id="chris" class="com.apress.prospring4.ch10.Contact"
        p:firstName="Chris"
        p:lastName="Schaefer"
        p:birthDate="1981-05-03"
        p:personalSite="http://www.dtzq.com"/>
</beans>
```

В листинге 10.15 приведена тестовая программа.

**Листинг 10.15. Тестирование специального форматировщика**

```
package com.apress.prospring4.ch10;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.core.convert.ConversionService;

public class ConvFormatServExample {
    public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/conv-format-service-app-context.xml");
      ctx.refresh();

      Contact chris = ctx.getBean("chris", Contact.class);

      System.out.println("Contact info: " + chris);

      ConversionService conversionService =
           ctx.getBean("conversionService", ConversionService.class);
      System.out.println("Birthdate of contact is : " +
           conversionService.convert(chris.getBirthDate(), String.class));
    }
}
```

Запуск этой программы дает следующий вывод:

```
Parsing date string: 1981-05-03
Contact info: First name: Chris - Last name: Schaefer - Birth date:
1981-05-03T00:00:00.000-04:00 - Personal site: http://www.dtzq.com
Formatting datetime: 1981-05-03T00:00:00.000-04:00
Birthdate of contact is : 1981-05-03
```

В выводе видно, что Spring применяет метод `parse()` нашего специального форматировщика для преобразования свойства из `String` в тип `DateTime` атрибута `birthDate`. При вызове метода `ConversionService.convert()` с передачей ему атрибута `birthDate` будет вызван метод `print()` для форматирования вывода.

# Проверка достоверности в Spring

Проверка достоверности является критически важной частью любого приложения. Правила проверки достоверности, применяемые к объектам предметной области, гарантируют, что все бизнес-данные хорошо структурированы и удовлетворяют всем бизнес-требованиям. В идеальном случае все правила проверки достоверности поддерживаются в централизованном местоположении, причем один и тот же набор правил применяется к одному и тому же типу данных вне зависимости от того, из какого источника данные поступили (например, через пользовательский ввод в веб-приложении, из удаленного приложения через веб-службы, из сообщения JMS либо из файла).

Когда речь идет о проверке достоверности, функции преобразования и форматирования также важны, потому что перед тем, как порция данных может быть проверена, она должна быть преобразована в желаемый объект POJO согласно правилам форматирования, определенным для каждого типа. Предположим, что пользователь

вводит некоторую контактную информацию в веб-приложении внутри браузера и затем отправляет данные серверу. На стороне сервера, если веб-приложение было разработано с применением инфраструктуры Spring MVC, то Spring извлечет данные из HTTP-запроса и выполнит преобразование `String` в желаемый тип на основе правила форматирования (например, строка `String`, представляющая дату, будет преобразована в поле типа `Date` с помощью правила форматирования `yyyy-MM-dd`). Этот процесс называется ***привязкой данных***. Когда привязка данных завершена, а объект предметной области сконструирован, объект подвергается проверке достоверности, и возникшие ошибки возвращаются и отображаются пользователю. Если проверка достоверности прошла успешно, объект сохраняется в базе данных.

В Spring поддерживаются два основных вида проверки достоверности. Один из них предоставляется самой платформой Spring и предусматривает создание специальных валидаторов за счет реализации интерфейса `org.springframework.validation.Validator`. Другой вид проверки достоверности использует поддержку JSR-349, т.е. API-интерфейса проверки достоверности бинов, встроенную в Spring. В последующих разделах мы рассмотрим оба типа.

## Использование интерфейса `Validator` в Spring

С применением интерфейса `Validator` из Spring можно разрабатывать логику проверки достоверности, создавая класс для реализации этого интерфейса. Давайте посмотрим, как это работает. Предположим, что в классе `Contact`, с которым мы имели дело до сих пор, имя не должно быть пустым. Чтобы проверить объекты `Contact` на предмет соблюдения этого правила, можно создать специальный валидатор. Класс такого валидатора показан в листинге 10.16.

**Листинг 10.16. Класс `ContactValidator`**

```
package com.apress.prospring4.ch10;

import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;
@Component("contactValidator")
public class ContactValidator implements Validator {
    @Override
    public boolean supports(Class<?> clazz) {
        return Contact.class.equals(clazz);
    }
    @Override
    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "firstName", "firstName.empty");
    }
}
```

Класс валидатора реализует интерфейс `Validator` и два его метода. Метод `supports()` указывает, поддерживает ли валидатор проверку достоверности для передаваемого ему класса. Метод `validate()` выполняет проверку достоверности для передаваемого ему объекта. Результат будет сохранен в экземпляре реализации интерфейса `org.springframework.validation.Errors`. В методе `validate()`

мы выполняем проверку только атрибута `firstName` и с помощью удобного метода `ValidationUtils.rejectIfEmpty()` гарантируем, что имя контакта не является пустым. Последний аргумент — это код ошибки, который может использоваться для поиска в пакетах ресурсов сообщений, связанных с проверкой достоверности, и последующего отображения локализованных сообщений об ошибках.

В листинге 10.17 приведено содержимое конфигурационного файла Spring (`spring-validator-app-context.xml`).

**Листинг 10.17. Конфигурация для проверки достоверности в Spring**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">
   <context:component-scan base-package="com.apress.prospring4.ch10"/>
</beans>
```

В листинге 10.18 показана программа для тестирования класса валидатора.

**Листинг 10.18. Тестирование класса валидатора**

```
package com.apress.prospring4.ch10;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.validation.BeanPropertyBindingResult;
import org.springframework.validation.ObjectError;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

public class SpringValidatorSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/spring-validator-app-context.xml");
      ctx.refresh();
      Contact contact = new Contact();
      contact.setFirstName(null);
      contact.setLastName("Schaefer");
      Validator contactValidator =
          ctx.getBean("contactValidator", Validator.class);
       BeanPropertyBindingResult result =
          new BeanPropertyBindingResult(contact, "Chris");
       ValidationUtils.invokeValidator(contactValidator, contact, result);
       List<ObjectError> errors = result.getAllErrors();
       System.out.println("No of validation errors: " + errors.size());
       for (ObjectError error: errors) {
          System.out.println(error.getCode());
       }
   }
}
```

Здесь конструируется объект `Contact` с именем, установленным в `null`. Затем из `ApplicationContext` извлекается валидатор.

Для сохранения результата проверки достоверности создается экземпляр класса `BeanPropertyBindingResult`. Для выполнения проверки достоверности вызывается метод `ValidationUtils.invokeValidator()`. После этого осуществляется проверка на предмет наличия ошибок.

Запуск этой программы дает следующий вывод:

```
No of validation errors: 1
firstName.empty
```

В процессе проверки достоверности произошла одна ошибка, код которой был корректно отображен.

## Использование спецификации JSR-349: Bean Validation

Платформа Spring 4 располагает полноценной поддержкой спецификации JSR-349, описывающей API-интерфейс проверки достоверности бинов (Bean Validation API). Этот интерфейс определяет в пакете `javax.validation.constraints` набор ограничений в форме Java-аннотаций (например, `@NotNull`), которые могут быть применены к объектам предметной области. Вдобавок могут быть разработаны специальные валидаторы (скажем, валидаторы на уровне класса) и также применены с использованием аннотаций.

Применение Bean Validation API освобождает вас от привязки к специфическому поставщику службы проверки достоверности. Благодаря Bean Validation API, для реализации логики проверки достоверности в объектах предметной области вы можете использовать стандартные аннотации и API-интерфейс, не зная лежащего в основе поставщика службы проверки достоверности. Например, популярным таким поставщиком, совместимым с JSR-349, является Hibernate Validator версии 5 (`http://hibernate.org/subprojects/validator`).

Платформа Spring предлагает гладкую поддержку Bean Validation API. Основные функциональные возможности включают поддержку стандартных аннотаций JSR-349 для определения ограничений проверки достоверности, специальные валидаторы и средства конфигурирования проверки достоверности JSR-349 в `ApplicationContext`. В последующих разделах все эти возможности рассматриваются по очереди. В Spring по-прежнему обеспечивается поддержка спецификации JSR-303, когда используется Hibernate Validator версии 4, а в пути классов указан API-интерфейс проверки достоверности версии 1.0.

### *Определение ограничений проверки достоверности для свойств объектов*

Начнем с применения ограничений проверки достоверности к свойствам объектов предметной области. В листинге 10.19 показан класс `Customer` с ограничениями проверки достоверности, примененными к атрибутам `firstName` и `customerType`.

**Листинг 10.19. Класс `Customer`**

```
package com.apress.prospring4.ch10;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
```

```
public class Customer {
    @NotNull
    @Size(min=2, max=60)
    private String firstName;

    private String lastName;

    @NotNull
    private CustomerType customerType;

    private Gender gender;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public CustomerType getCustomerType() {
        return customerType;
    }

    public void setCustomerType(CustomerType customerType) {
        this.customerType = customerType;
    }

    public Gender getGender() {
        return gender;
    }

    public void setGender(Gender gender) {
        this.gender = gender;
    }

    public boolean isIndividualCustomer() {
        return this.customerType.equals(CustomerType.INDIVIDUAL);
    }
}
```

К атрибуту `firstName` применены два ограничения. Первое из них управляется аннотацией `@NotNull`, которая указывает, что значение не должно быть равно `null`. Кроме того, аннотация `@Size` управляет длиной атрибута `firstName`. Ограничение `@NotNull` также применяется и к атрибуту `customerType`. В листингах 10.20 и 10.21 представлены классы `CustomerType` и `Gender`.

**Листинг 10.20. Класс `CustomerType`**

```
package com.apress.prospring4.ch10;

public enum CustomerType {
    INDIVIDUAL("I"), CORPORATE("C");
```

```
    private String code;
    private CustomerType(String code) {
        this.code = code;
    }
    @Override
    public String toString() {
        return this.code;
    }
}
```

**Листинг 10.21. Класс `Gender`**

```
package com.apress.prospring4.ch10;

public enum Gender {
    MALE("M"), FEMALE("F");

    private String code;

    private Gender(String code) {
        this.code = code;
    }
    @Override
    public String toString() {
        return this.code;
    }
}
```

Тип заказчика (класс `CustomerType`) указывает, является ли заказчик физическим лицом или компанией, а пол (класс `Gender`) должен применяться только к заказчикам — физическим лицам. В случае компаний пол должен быть `null`.

### *Конфигурирование поддержки проверки достоверности бинов в Spring*

Чтобы сконфигурировать поддержку Bean Validation API в `ApplicationContext`, мы определяем экземпляр класса `org.springframework.validation.beanvalidation.LocalValidatorFactoryBean` в конфигурации Spring. Содержимое конфигурационного файла `jsr349-app-context.xml` показано в листинге 10.22.

**Листинг 10.22. Конфигурирование Bean Validation API в Spring**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan
        base-package="com.apress.prospring4.ch10"/>

    <bean id="validator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
</beans>
```

Требуется только указать объявление бина с классом `LocalValidatorFactoryBean`. По умолчанию Spring будет искать библиотеку Hibernate Validator в пути классов. А теперь создадим служебный класс для обеспечения проверки достоверности в классе `Customer`. Этот служебный класс по имени `MyBeanValidationService` представлен в листинге 10.23.

**Листинг 10.23. Класс `MyBeanValidationService`**

```
package com.apress.prospring4.ch10;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validator;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("myBeanValidationService")
public class MyBeanValidationService {
    @Autowired
    private Validator validator;

    public Set<ConstraintViolation<Customer>> validateCustomer(Customer customer)
    {
        return validator.validate(customer);
    }
}
```

Здесь внедряется экземпляр `javax.validation.Validator` (обратите внимание на его отличие от интерфейса `Validator`, предоставляемого Spring, которым является `org.springframework.validation.Validator`).

После определения бина `LocalValidatorFactoryBean` можно создавать обработчик для интерфейса `Validator` в любом месте приложения. Для проведения проверки достоверности объекта POJO вызывается метод `Validator.validate()`. Результаты проверки достоверности возвращаются в виде списка (`List`) интерфейсов `ConstraintViolation<T>`.

Тестовая программа приведена в листинге 10.24.

**Листинг 10.24. Класс `Jsr349Sample`**

```
package com.apress.prospring4.ch10;

import java.util.HashSet;
import java.util.Set;

import javax.validation.ConstraintViolation;

import org.springframework.context.support.GenericXmlApplicationContext;

public class Jsr349Sample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/jsr349-app-context.xml");
        ctx.refresh();

        MyBeanValidationService myBeanValidationService =
            ctx.getBean("myBeanValidationService", MyBeanValidationService.class);
```

```
        Customer customer = new Customer();
        customer.setFirstName("C");
        customer.setLastName("Schaefer");
        customer.setCustomerType(null);
        customer.setGender(null);

        validateCustomer(customer, myBeanValidationService);
    }

    private static void validateCustomer(Customer customer,
        MyBeanValidationService myBeanValidationService) {

        Set<ConstraintViolation<Customer>> violations =
            new HashSet<ConstraintViolation<Customer>>();
        violations = myBeanValidationService.validateCustomer(customer);

        listViolations(violations);
    }

    private static
      void listViolations(Set<ConstraintViolation<Customer>> violations) {
        System.out.println("No. of violations: " + violations.size());

        for (ConstraintViolation<Customer> violation: violations) {
            System.out.println("Validation error for property: " +
                violation.getPropertyPath()
                + " with value: " + violation.getInvalidValue()
                + " with error message: " + violation.getMessage());
        }
    }
}
```

Как показано в этом листинге, объект `Customer` конструируется со значениями `firstName` и `customerType`, нарушающими ограничения.

В методе `validateCustomer()` вызывается метод `MyBeanValidationService.validateCustomer()`, который, в свою очередь, обращается к JSR-349 Bean Valiadation API.

Запуск этой программы дает следующий вывод:

```
No. of violations: 2
Validation error for property: customerType with value:
null with error message: may not be null
Validation error for property: firstName with value:
C with error message: size must be between 2 and 60
```

В этом выводе видно, что были обнаружены два нарушения, для которых отображены сообщения. Кроме того, библиотека Hibernate Validator на основе аннотации сформировала стандартные сообщения об ошибках проверки достоверности. Можно также предоставить собственные сообщения об ошибках, что демонстрируется в следующем разделе.

### *Создание специального валидатора*

Кроме проверки достоверности на уровне атрибутов можно применять аналогичную проверку на уровне классов. Например, в случае класса `Customer` мы хотим гарантировать для отдельных заказчиков, что атрибуты `lastName` и `gender` не равны `null`.

В такой ситуации для выполнения упомянутой проверки мы можем разработать специальный валидатор. В рамках Bean Validation API разработка специального валидатора является двухшаговым процессом. Первый шаг заключается в создании типа аннотации для валидатора, как показано в листинге 10.25. Второй шаг предусматривает написание класса, который реализует логику проверки достоверности.

**Листинг 10.25. Аннотация `CheckIndividualCustomer`**

```
package com.apress.prospring4.ch10;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Constraint(validatedBy=IndividualCustomerValidator.class)
@Documented
public @interface CheckIndividualCustomer {
    String message() default "Individual customer should have gender and last
name defined";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Аннотация `@Target(ElementType.TYPE)` означает, что аннотация должна применяться только на уровне классов. Аннотация `@Constraint` указывает, что это валидатор, а в атрибуте `validatedBy` задается класс, предоставляющий логику проверки достоверности. Внутри тела определены три атрибута (в форме методов), которые описаны ниже.

- Атрибут `message` определяет сообщение (или код ошибки) для возврата, когда ограничение нарушено. В аннотации также предоставляется стандартное сообщение.
- Атрибут `groups` указывает группу проверки достоверности, если это применимо. Валидаторы допускается назначать в разные группы и выполнять проверку в конкретной группе.
- Атрибут `payload` задает дополнительные объекты полезной нагрузки (класса, реализующего интерфейс `javax.validation.Payload`). Этот атрибут позволяет присоединять дополнительную информацию к ограничению (например, объект полезной нагрузки может указывать серьезность нарушения ограничения).

В листинге 10.26 представлен класс `IndividualCustomerValidator`, который предоставляет логику проверки достоверности.

**Листинг 10.26. Класс `IndividualCustomerValidator`**

```
package com.apress.prospring4.ch10;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
public class IndividualCustomerValidator implements
    ConstraintValidator<CheckIndividualCustomer, Customer> {
    @Override
    public void initialize(CheckIndividualCustomer constraintAnnotation) {
    }
    @Override
    public boolean isValid(Customer customer,
        ConstraintValidatorContext context) {
        boolean result = true;
        if (customer.getCustomerType() != null &&
          (customer.isIndividualCustomer() && (customer.getLastName() == null
          || customer.getGender() == null))) {
            result = false;
        }
        return result;
    }
}
```

Класс `IndividualCustomerValidator` реализует интерфейс `ConstraintValidator<CheckIndividualCustomer, Customer>`, а это значит, что валидатор ищет аннотацию `CheckIndividualCustomer` в классах `Customer`.

Метод `isValid()` реализован, и лежащий в основе поставщик службы проверки достоверности (к примеру, Hibernate Validator) будет передавать этому методу проверяемый экземпляр. В коде метода мы удостоверяемся, что пользователь является физическим лицом, а затем в том, что свойства `lastName` и `gender` не равны `null`. Результатом является булевское значение, указывающее, как завершилась проверка.

Чтобы включить проверку достоверности, примените аннотацию `@CheckIndividualCustomer` к классу `Customer`, как показано в листинге 10.27.

**Листинг 10.27. Применение специальной проверки достоверности к классу `Customer`**

```
package com.apress.prospring4.ch10;
@CheckIndividualCustomer
public class Customer {
    ...
}
```

Для тестирования специальной проверки достоверности добавьте код, приведенный в листинге 10.28, в метод `main()` класса `Jsr349Sample` из листинга 10.24.

**Листинг 10.28. Тестирование применения специальной проверки достоверности к классу `Customer`**

```
        customer.setFirstName("Chris");
        customer.setLastName("Schaefer");
        customer.setCustomerType(CustomerType.INDIVIDUAL);
        customer.setGender(null);
        validateCustomer(customer, myBeanValidationService);
```

Запуск этой программы дает следующий вывод (показан не полностью):

```
No. of violations: 1
Validation error for property:  with value: com.apress.prospring4.ch10.
Customer@5ae50ce6 with error message: Individual customer should have
gender and last name defined
```

В этом выводе видно, что проверяемое значение (которое является объектом `Customer`) нарушает правило проверки достоверности для заказчиков — физических лиц, т.к. атрибут `gender` равен `null`. Обратите также внимание, что путь к свойству пуст, поскольку это ошибка проверки достоверности на уровне класса.

### Использование аннотации `@AssertTrue` для специальной проверки достоверности

Помимо реализации специального валидатора специальную проверку достоверности в Bean Validation API можно также применять посредством аннотации `@AssertTrue`. Давайте посмотрим, как это работает.

В классе `Customer` удалите аннотацию `@CheckIndividualCustomer` и приведите код в соответствие с листингом 10.29.

**Листинг 10.29. Применение аннотации `@AssertTrue` к классу `Customer`**

```
public class Customer {
   ...
   @AssertTrue(message="ERROR! Individual customer should have gender and
last name defined")
   public boolean isIndividualCustomer() {
      boolean result = true;
      if (getCustomerType() != null &&
            (this.customerType.equals(CustomerType.INDIVIDUAL)
              && (gender == null || lastName == null))) {
         result = false;
      }
      return result;
   }
   ...
}
```

К классу `Customer` добавлен метод `isValidIndividualCustomer()`, аннотированный с помощью `@AssertTrue` (из пакета `javax.validation.constraints`). При запуске проверки достоверности поставщик будет инициировать проверку и удостоверяться в том, что результатом является `true`. Спецификация JSR-349 также предоставляет аннотацию `@AssertFalse` для проверки условий, которые должны быть `false`. Запустив тестовую программу (`Jsr349Sample`) еще раз, вы получите тот же самый вывод, что и при работе со специальным валидатором.

### Соображения по поводу специальной проверки достоверности

Итак, какой подход должен использоваться для реализации специальной проверки достоверности в JSR-349: специальный валидатор или аннотация `@AssertTrue`? В общем случае метод с `@AssertTrue` проще в реализации, к тому же правила проверки можно видеть прямо в коде объектов предметной области. Тем не менее, при более сложной логике проверки достоверности (например, нужно внедрить служеб-

ный класс, получить доступ в базу данных и выполнить проверку с рядом допустимых значений) следует избрать подход с реализацией специального валидатора, поскольку внедрять объекты уровня обслуживания в объекты предметной области совершенно не желательно. Кроме того, специальные валидаторы могут многократно использоваться с похожими объектами предметной области.

### Выбор API-интерфейса проверки достоверности для использования

После обсуждения собственного интерфейса `Validator` в Spring и Bean Validation API осталось выяснить, что из них следует применять в приложении? Безусловно, необходимо избрать JSR-349 по следующим причинам.

- JSR-349 — это стандарт JEE, который широко поддерживается многими платформами для построения интерфейсной и серверной частей (например, Spring, JPA 2, Spring MVC и GWT).
- Спецификация JSR-349 предоставляет стандартный API-интерфейс проверки достоверности, который скрывает лежащего в основе поставщика, поэтому вы не привязываетесь к конкретному поставщику.
- Начиная с версии 4, платформа Spring тесно интегрирована с JSR-349. Например, в веб-контроллере Spring MVC аргумент метода можно снабдить аннотацией `@Valid` (из пакета `javax.validation`), и Spring будет запускать проверку достоверности JSR-349 во время процесса привязки данных. Более того, в конфигурации веб-приложения Spring MVC простой дескриптор `<mvc:annotation-driven/>` сконфигурирует Spring на автоматическое включение системы преобразования типов и форматирования полей Spring, а также поддержки спецификации JSR-349: Bean Validation.
- Если вы используете JPA 2, то поставщик автоматически выполнит проверку достоверности JSR-349 для сущности перед ее сохранением, предоставляя еще один уровень защиты.

За более подробными сведениями по использованию JSR-349: Bean Validation с Hibernate Validator в качестве поставщика реализации обращайтесь на страницу документации по Hibernate Validator (`http://docs.jboss.org/hibernate/validator/5.1/reference/en-US/html`).

## Резюме

В этой главе была рассмотрена система преобразования типов Spring, а также SPI-интерфейс форматировщиков полей. Вы узнали, как использовать новую систему преобразования типов для преобразований между произвольными типами в дополнение к поддержке редакторов свойств.

Кроме того, мы обсудили поддержку в Spring проверки достоверности, интерфейса `Validator` и рекомендуемой спецификации JSR-349: Bean Validation.

# ГЛАВА 11

# Планирование задач в Spring

*Планирование задач* является общей функциональной возможностью в корпоративных приложениях. Планирование задач состоит из трех частей: задачи (представляющей собой порцию бизнес-логики, которая должна запускаться в определенное время или на регулярной основе), триггера (указывающего условие, при удовлетворении которого задача должна выполняться) и планировщика (который запускает задачу на основе информации, полученной от триггера).

В частности, в этой главе будут рассматриваться следующие темы.

- **Планирование задач в Spring.** Мы раскроем поддержку планирования задач в Spring, акцентируя внимание на абстракции `TaskScheduler`, которая появилась в Spring 3. Мы также опишем сценарии планирования, такие как планирование с фиксированным интервалом и с помощью выражения `cron`.
- **Асинхронное выполнение задач.** Мы покажем, как использовать аннотацию `@Async` в Spring для выполнения задач асинхронным образом.
- **Выполнение задач в Spring.** Мы кратко обсудим интерфейс `TaskExecutor` в Spring и особенности выполнения задач.

## Зависимости для примеров планирования задач

В табл. 11.1 перечислены зависимости, которые необходимы для примеров планирования задач, рассматриваемых в этой главе.

**Таблица 11.1. Зависимости Maven для планирования задач**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.springframework.data` | `spring-data-jpa` | 1.5.0.RELEASE | Библиотека Spring Data JPA |
| `joda-time` | `joda-time` | 2.3 | Библиотека Joda-Time (`http://joda-time.sourceforge.net/`) |
| `org.jadira.usertype` | `usertype.core` | 3.0.0.GA | Библиотека для интеграции с Hibernate, обеспечивающая сохранение данных типа даты и времени |

*Окончание табл. 11.1*

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| com.google.guava | guava | 14.0.1 | Полезные вспомогательные классы для работы с коллекциями, поддержки параллелизма, обработки строк и т.д. |
| org.slf4j | slf4j-log4j12 | 1.7.6 | Библиотека ведения журналов (www.slf4j.org), которая используется в примерах этой главы. Данная библиотека помогает соединить ведение журнала SLF4J с библиотекой log4j |

# Реализация планирования задач в Spring

Корпоративные приложения часто нуждаются в планировании задач. Во многих приложениях разнообразные задачи (вроде отправки заказчикам уведомлений по электронной почте, запуска заданий в конце дня, обслуживания данных, обновления данных в пакетах и т.п.) должны планироваться к запуску на регулярной основе, либо через фиксированные интервалы (например, каждый час), либо по заданному расписанию (скажем, ежедневно в 20:00 с понедельника по пятницу). Как упоминалось ранее, планирование задач состоит из трех частей: определение расписания (триггер), выполнение задачи (планировщик) и сама задача.

Для инициирования запуска задачи в Spring-приложении доступно много способов. Один из них предусматривает запуск задания внешне из системы планирования, которая уже существует в среде развертывания приложений. Например, на многих предприятиях для планирования задач используются коммерческие системы, такие как Control-M или CA AutoSys. Если приложение функционирует на платформе Linux/Unix, можно применять планировщик crontab. Запуск заданий можно осуществлять отправкой запроса веб-службе REST, связанной с приложением Spring, в результате чего контроллер Spring MVC инициирует выполнение задания.

Еще один способ предполагает использование поддержки планирования задач, встроенной в Spring. Для планирования задач в Spring доступны три варианта.

- **Поддержка JDK-объекта `Timer`.** Для планирования задач платформа Spring поддерживает объект `Timer` из JDK.
- **Интеграция с Quartz.** Платформа Spring интегрирована с Quartz Scheduler (`www.quartz-scheduler.org`) — популярной библиотекой планирования с открытым кодом.
- **Абстракция `TaskScheduler`, встроенная в Spring.** В версии Spring 3 появилась абстракция `TaskScheduler`, которая предлагает простой способ планирования задач и поддерживает наиболее типичные требования.

В этом разделе мы сосредоточим внимание на применении абстракции `TaskScheduler` для планирования задач.

## Введение в абстракцию TaskScheduler

В абстракции TaskScheduler платформы Spring задействованы три главных участника.

- **Интерфейс Trigger.** Интерфейс org.springframework.scheduling.Trigger предоставляет поддержку для определения механизма запуска. В Spring доступны две реализации Trigger. Класс CronTrigger поддерживает запуск на базе выражения cron, а класс PeriodicTrigger — запуск на основе начальной задержки и затем фиксированного интервала.
- **Задача.** Задача — это порция бизнес-логики, запуск которой необходимо запланировать. В Spring задача может быть указана как метод внутри любого бина Spring.
- **Интерфейс TaskScheduler.** Интерфейс org.springframework.scheduling.TaskScheduler предоставляет поддержку для планирования задач. В Spring доступны три класса реализации интерфейса TaskScheduler. Класс TimerManagerTaskScheduler (из пакета org.springframework.scheduling.commonj) является оболочкой для интерфейса commonj.timers.TimerManager из CommonJ, который обычно используется в коммерческих серверах приложений JEE, таких как WebSphere и WebLogic. Классы ConcurrentTaskScheduler и ThreadPoolTaskScheduler (оба из пакета org.springframework.scheduling.concurrent) представляют собой оболочки для класса java.util.concurrent.ScheduledThreadPoolExecutor. Оба класса поддерживают запуск задач из общего пула потоков.

На рис. 11.1 показаны отношения между интерфейсом Trigger, интерфейсом TaskScheduler и задачей (которая реализует интерфейс java.lang.Runnable).

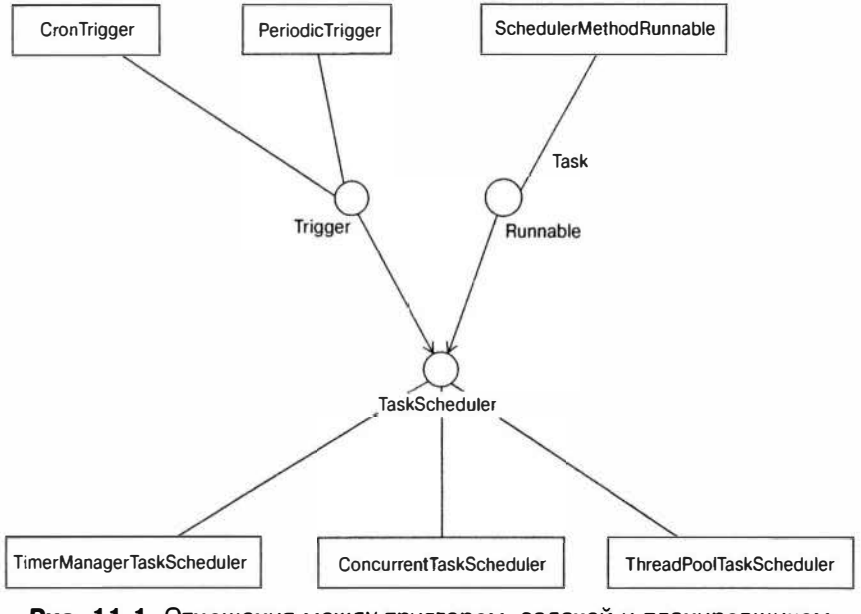


**Рис. 11.1.** Отношения между триггером, задачей и планировщиком

Планировать задачи с применением абстракции `TaskScheduler` из Spring можно двумя способами. Один из них предусматривает использование пространства имен `task` в XML-конфигурации Spring, а другой — аннотаций. Давайте рассмотрим оба эти способа.

## Пример задачи

В целях демонстрации планирования задач в Spring мы реализуем сначала простое задание, а именно — приложение, обслуживающее базу данных с информацией об автомобилях. В листинге 11.1 показан класс `Car`, который реализован как сущностный класс JPA.

**Листинг 11.1. Класс `Car`**

```
package com.apress.prospring4.ch11;

import static javax.persistence.GenerationType.IDENTITY;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Version;

import org.hibernate.annotations.Type;
import org.joda.time.DateTime;

@Entity
@Table(name="car")
public class Car {
    private Long id;
    private String licensePlate;
    private String manufacturer;
    private DateTime manufactureDate;
    private int age;
    private int version;
    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return id;
    }
    @Column(name="LICENSE_PLATE")
    public String getLicensePlate() {
        return licensePlate;
    }
    @Column(name="MANUFACTURER")
    public String getManufacturer() {
        return manufacturer;
    }
    @Column(name="MANUFACTURE_DATE")
    @Type(type="org.jadira.usertype.dateandtime.joda.PersistentDateTime")
    public DateTime getManufactureDate() {
        return manufactureDate;
    }
```

```
    @Column(name="AGE")
    public int getAge() {
        return age;
    }
    @Version
    public int getVersion() {
        return version;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public void setLicensePlate(String licensePlate) {
        this.licensePlate = licensePlate;
    }
    public void setManufacturer(String manufacturer) {
        this.manufacturer = manufacturer;
    }
    public void setManufactureDate(DateTime manufactureDate) {
        this.manufactureDate = manufactureDate;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setVersion(int version) {
        this.version = version;
    }
    @Override
    public String toString() {
        return "License: " + licensePlate + " - Manufacturer: " + manufacturer
               + " - Manufacture Date: " + manufactureDate + " - Age: " + age;
    }
}
```

В листингах 11.2 и 11.3 приведен код сценариев для создания таблицы (`schema.sql`) и заполнения ее тестовыми данными (`test-data.sql`) для сущностного класса `Car`.

**Листинг 11.2. Сценарий для создания таблицы**

```
DROP TABLE IF EXISTS CONTACT;

CREATE TABLE CAR (
    ID INT NOT NULL AUTO_INCREMENT
    , LICENSE_PLATE VARCHAR(20) NOT NULL
    , MANUFACTURER VARCHAR(20) NOT NULL
    , MANUFACTURE_DATE DATE NOT NULL
    , AGE INT NOT NULL DEFAULT 0
    , VERSION INT NOT NULL DEFAULT 0
    , UNIQUE UQ_CAR_1 (LICENSE_PLATE)
    , PRIMARY KEY (ID)
);
```

**Листинг 11.3. Сценарий для заполнения таблицы тестовыми данными**

```
insert into car (license_plate, manufacturer, manufacture_date)
       values ('LICENSE-1001', 'Ford', '1980-07-30');
insert into car (license_plate, manufacturer, manufacture_date)
       values ('LICENSE-1002', 'Toyota', '1992-12-30');
insert into car (license_plate, manufacturer, manufacture_date)
       values ('LICENSE-1003', 'BMW', '2003-1-6');
```

А теперь определим уровень обслуживания для сущности `Car`. Мы будем использовать проект Spring Data JPA и его поддержку абстракции репозитория.

В листинге 11.4 представлен интерфейс `CarRepository`.

**Листинг 11.4. Интерфейс `CarRepository`**

```
package com.apress.prospring4.ch11;

import org.springframework.data.repository.CrudRepository;

public interface CarRepository extends CrudRepository<Car, Long> {
}
```

Здесь нет ничего особенного; мы просто реализовали интерфейс `CrudRepository<Car,Long>`. В листингах 11.5 и 11.6 показан интерфейс `CarService` и класс реализации `CarServiceImpl`.

**Листинг 11.5. Интерфейс `CarService`**

```
package com.apress.prospring4.ch11;

import java.util.List;

public interface CarService {
   List<Car> findAll();
   Car save(Car car);
   void updateCarAgeJob();
}
```

**Листинг 11.6. Класс `CarServiceImpl`**

```
package com.apress.prospring4.ch11;

import com.google.common.collect.Lists;
import org.joda.time.DateTime;
import org.joda.time.Years;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service("carService")
@Repository
@Transactional
```

```
public class CarServiceImpl implements CarService {
   final Logger logger = LoggerFactory.getLogger(CarServiceImpl.class);

   @Autowired
   CarRepository carRepository;

   @Override
   @Transactional(readOnly=true)
   public List<Car> findAll() {
      return Lists.newArrayList(carRepository.findAll());
   }

   @Override
   public Car save(Car car) {
      return carRepository.save(car);
   }

   @Override
   public void updateCarAgeJob() {
      List<Car> cars = findAll();

      DateTime currentDate = DateTime.now();
      logger.info("Car age update job started");

      for (Car car: cars) {
        int age =
          Years.yearsBetween(car.getManufactureDate(), currentDate).getYears();

        car.setAge(age);
        save(car);
        logger.info("Car age update--- " + car);
      }

      logger.info("Car age update job completed successfully");
   }
}
```

В коде определены два метода; один из них извлекает информацию обо всех автомобилях, а другой сохраняет обновленный объект `Car`. Третий метод, `updateCarAgeJob()`, представляет собой задание, которое должно запускаться на регулярной основе и обновлять возраст автомобиля, имея дату его изготовления и текущую дату.

В листинге 11.7 представлена конфигурация Spring для поддержки рассматриваемого примера приложения с автомобилями (`car-job-app-context.xml`).

**Листинг 11.7. Файл `car-job-app-context.xml`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:jdbc="http://www.springframework.org/schema/jdbc"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xmlns:jpa="http://www.springframework.org/schema/data/jpa"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
```

```
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/data/jpa
        http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

    <jdbc:embedded-database id="dataSource" type="H2">
        <jdbc:script location="classpath:META-INF/config/schema.sql"/>
        <jdbc:script location="classpath:META-INF/config/test-data.sql"/>
    </jdbc:embedded-database>

    <bean id="transactionManager"
          class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="emf"/>
    </bean>

    <tx:annotation-driven transaction-manager="transactionManager"/>

    <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="jpaVendorAdapter">
         <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
          />
        </property>
        <property name="packagesToScan" value="com.apress.prospring4.ch11"/>
        <property name="jpaProperties">
            <props>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.H2Dialect
                </prop>
                <prop key="hibernate.max_fetch_depth">3</prop>
                <prop key="hibernate.jdbc.fetch_size">50</prop>
                <prop key="hibernate.jdbc.batch_size">10</prop>
                <prop key="hibernate.show_sql">true</prop>
            </props>
        </property>
    </bean>

    <jpa:repositories base-package="com.apress.prospring4.ch11"
                      entity-manager-factory-ref="emf"
                      transaction-manager-ref="transactionManager"/>

    <context:component-scan base-package="com.apress.prospring4.ch11"/>
</beans>
```

Эта конфигурация должна выглядеть знакомой. Давайте теперь займемся планированием задания по обновлению возраста автомобилей в Spring.

## Использование пространства имен `task` для планирования задач

Подобно другим пространствам имен в Spring, пространство имен `task` предоставляет упрощенную конфигурацию для планируемых задач за счет применения абстракции `TaskScheduler`, доступной в Spring.

Пространство имен `task` для планирования задач используется очень просто. В листинге 11.8 приведено содержимое конфигурационного файла `task-namespace-app-context.xml`.

**Листинг 11.8. Конфигурация Spring, использующая пространство имен `task`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:task="http://www.springframework.org/schema/task"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/task
       http://www.springframework.org/schema/task/spring-task.xsd">

   <import resource="car-job-app-context.xml"/>

   <task:scheduler id="myScheduler" pool-size="10"/>

   <task:scheduled-tasks scheduler="myScheduler">
      <task:scheduled ref="carService" method="updateCarAgeJob"
                      fixed-delay="10000"/>
   </task:scheduled-tasks>
</beans>
```

В этом коде импортируется контекст для приложения, работающего с автомобилями. Обнаружив дескриптор `<task:scheduler>`, платформа Spring создает экземпляр класса `ThreadPoolTaskScheduler`, при этом атрибут `pool-size` задает размер пула потоков, который планировщик может использовать. Внутри дескриптора `<task:scheduled-tasks>` допускается планировать одну или большее число задач. В дескрипторе `<task:scheduled>` задача может ссылаться на бин Spring (в данном случае это бин `carService`) и специфический метод этого бина (в рассматриваемом примере это метод `updateCarAgeJob()`).

Атрибут `fixed-delay` указывает Spring на необходимость создания `PeriodicTrigger` как реализации `Trigger` для `TaskScheduler`.

В листинге 11.9 приведен код программы для тестирования планирования задач.

**Листинг 11.9. Тестирование планирования задач в Spring**

```
package com.apress.prospring4.ch11;

import org.springframework.context.support.GenericXmlApplicationContext;

public class ScheduleTaskSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/task-namespace-app-context.xml");
      ctx.refresh();

      while (true) {
      }
   }
}
```

Класс ScheduleTaskSample довольно прост; он лишь загружает Application Context и организует бесконечный цикл. Запуск этой программы даст следующий вывод, получаемый каждые 10 секунд:

```
<Car age update job started>
<Car age update--- License: LICENSE-1001 - Manufacturer:
Ford - Manufacture Date: 1980-07-29T20:00:00.000-04:00 - Age: 33>
<Car age update--- License: LICENSE-1002 - Manufacturer:
Toyota - Manufacture Date: 1992-12-29T19:00:00.000-05:00 - Age: 21>
<Car age update--- License: LICENSE-1003 - Manufacturer:
BMW - Manufacture Date: 2003-01-05T19:00:00.000-05:00 - Age: 11>
<Car age update job completed successfully>
```

В выводе видно, что атрибуты age автомобилей были успешно обновлены.

Кроме фиксированного интервала доступен более гибкий механизм планирования, предусматривающий использование выражения cron. Замените следующую строку в листинге 11.8:

```
<task:scheduled ref="carService" method="updateCarAgeJob"
                fixed-delay="10000"/>
```

показанной ниже строкой:

```
<task:scheduled ref="carService" method="updateCarAgeJob"
                cron="0 * * * * *"/>
```

После этого изменения снова запустите тестовую программу с классом ScheduleTaskSample и вы увидите, что задание будет выполняться каждую минуту.

## Использование аннотаций для планирования задач

Еще один способ планирования задач с применением абстракции TaskScheduler из Spring предполагает применение аннотаций. Для этой цели в Spring предусмотрена аннотация @Scheduled.

Чтобы включить поддержку аннотаций для планирования задач, необходимо предоставить дескриптор <task:annotation-driven> в XML-конфигурации Spring. Содержимое конфигурационного файла task-annotation-app-context.xml показано в листинге 11.10.

**Листинг 11.10. Конфигурация Spring для планирования на основе аннотаций**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:task="http://www.springframework.org/schema/task"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/task
      http://www.springframework.org/schema/task/spring-task.xsd">

   <import resource="car-job-app-context.xml"/>

   <task:scheduler id="myScheduler" pool-size="10"/>

   <task:annotation-driven scheduler="myScheduler"/>
</beans>
```

Здесь с помощью дескриптора <task:annotation-driven> включается поддержка планирования на основе аннотаций, с указанием в атрибуте scheduler ссылки на бин myScheduler.

Чтобы запланировать выполнение конкретного метода в бине Spring, просто снабдите метод аннотацией @Scheduled и передайте требования к планированию. В листинге 11.11 приведен модифицированный код метода updateCarAgeJob() класса CarServiceImpl.

**Листинг 11.11. Переделанный код метода updateCarAgeJob() класса CarServiceImpl**

```
@Scheduled(fixedDelay=10000)
public void updateCarAgeJob() {
    ...
}
```

Тестовая программа представлена в листинге 11.12.

**Листинг 11.12. Тестирование планирования на основе аннотаций**

```
package com.apress.prospring4.ch11;

import org.springframework.context.support.GenericXmlApplicationContext;
public class ScheduleTaskAnnotationSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/task-annotation-app-context.xml");
        ctx.refresh();
        while (true) {
        }
    }
}
```

Выполнение этой программы даст тот же самый вывод, что и пример использования пространства имен task. Меняя атрибут в аннотации @Scheduled, можно опробовать разные механизмы запуска (т.е. fixedDelay, fixedRate, cron). Протестируйте это самостоятельно.

# Асинхронное выполнение задач в Spring

Начиная с версии Spring 3.0, платформа также поддерживает применение аннотаций для выполнения задачи асинхронным образом. Для этого необходимо лишь аннотировать метод с помощью @Async.

Чтобы увидеть это в действии, рассмотрим простой пример. В листингах 11.13 и 11.14 показан интерфейс AsyncService и класс реализации AsyncServiceImpl.

**Листинг 11.13. Интерфейс AsyncService**

```
package com.apress.prospring4.ch11;

import java.util.concurrent.Future;
public interface AsyncService {
    void asyncTask();
    Future<String> asyncWithReturn(String name);
}
```

**Листинг 11.14. Класс AsyncServiceImpl**

```
package com.apress.prospring4.ch11;

import java.util.concurrent.Future;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.AsyncResult;
import org.springframework.stereotype.Service;

@Service("asyncService")
public class AsyncServiceImpl implements AsyncService {
    final Logger logger = LoggerFactory.getLogger(AsyncServiceImpl.class);
    @Async
    @Override
    public void asyncTask() {
        logger.info("Start execution of async. task");
        try {
            Thread.sleep(10000);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        logger.info("Complete execution of async. task");
    }
    @Async
    @Override
    public Future<String> asyncWithReturn(String name) {
        logger.info("Start execution of async. task with return");
        try {
            Thread.sleep(5000);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        logger.info("Complete execution of async. task with return");
        return new AsyncResult<String>("Hello: " + name);
    }
}
```

В интерфейсе `AsyncService` определены два метода. Метод `asyncTask()` — это простая задача, которая записывает информацию в журнал.

Метод `asyncWithReturn()` принимает аргумент типа `String` и возвращает экземпляр интерфейса `java.util.concurrent.Future<V>`. После завершения `asyncWithReturn()` результат хранится в экземпляре класса `org.springframework.scheduling.annotation.AsyncResult<V>`, который реализует интерфейс `Future<V>` и может использоваться для извлечения результата выполнения в более позднее время. В листинге 11.15 показано содержимое файла конфигурации Spring (`async-app-context.xml`).

**Листинг 11.15. Конфигурация Spring для асинхронной задачи**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:task="http://www.springframework.org/schema/task"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/task
        http://www.springframework.org/schema/task/spring-task.xsd">
    <context:component-scan base-package="com.apress.prospring4.ch11"/>
    <task:annotation-driven />
</beans>
```

Дескриптор `<task:annotation-driven />` в листинге 11.15 необходим для поддержки аннотации `@Async`. Тестовая программа представлена в листинге 11.16.

**Листинг 11.16. Тестирование асинхронной задачи**

```
package com.apress.prospring4.ch11;

import java.util.concurrent.Future;

import org.springframework.context.support.GenericXmlApplicationContext;

public class AsyncTaskSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/async-app-context.xml");
        ctx.refresh();
        AsyncService asyncService = ctx.getBean("asyncService", AsyncService.class);

        for (int i = 0; i < 5; i++) {
            asyncService.asyncTask();
        }
        Future<String> result1 = asyncService.asyncWithReturn("Chris");
        Future<String> result2 = asyncService.asyncWithReturn("John");
        Future<String> result3 = asyncService.asyncWithReturn("Robert");
        try {
            Thread.sleep(6000);
            System.out.println("Result1: " + result1.get());
            System.out.println("Result2: " + result2.get());
            System.out.println("Result3: " + result3.get());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Мы вызываем метод `asyncTask()` пять раз и метод `asyncWithReturn()` три раза с разными аргументами, после чего извлекаем результат после паузы в шесть секунд. Запуск этой программы дает следующий вывод:

```
2014-03-13 12:40:06,474 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Start execution of async. task>
2014-03-13 12:40:06,474 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Start execution of async. task>
```

```
2014-03-13 12:40:06,475 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Start execution of async. task>
2014-03-13 12:40:06,475 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Start execution of async. task>
2014-03-13 12:40:06,475 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Start execution of async. task>
2014-03-13 12:40:06,475 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Start execution of async. task with return>
2014-03-13 12:40:06,475 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Start execution of async. task with return>
2014-03-13 12:40:06,475 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Start execution of async. task with return>
2014-03-13 12:40:11,477 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Complete execution of async. task with return>
2014-03-13 12:40:11,477 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Complete execution of async. task with return>
2014-03-13 12:40:11,477 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Complete execution of async. task with return>
Result1: Hello: Chris
Result2: Hello: John
Result3: Hello: Robert
2014-03-13 12:40:16,477 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Complete execution of async. task>
2014-03-13 12:40:16,477 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Complete execution of async. task>
2014-03-13 12:40:16,477 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Complete execution of async. task>
2014-03-13 12:40:16,477 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Complete execution of async. task>
2014-03-13 12:40:16,477 INFO [com.apress.prospring4.ch11.AsyncServiceImpl]
- <Complete execution of async. task>
```

В выводе легко заметить, что все вызовы начались в одно и то же время. Три вызова со значениями возврата завершились первыми и отобразились в консольном выводе. Наконец, были завершены пять вызовов метода `asyncTask()`.

# Выполнение задач в Spring

Начиная с версии Spring 2.0, платформа предоставляет абстракцию для выполнения задач в виде интерфейса `TaskExecutor`. Интерфейс `TaskExecutor` делает именно то, о чем и можно было догадаться: он выполняет задачу, представленную Java-реализацией `Runnable`. Платформа Spring предлагает несколько готовых реализаций `TaskExecutor`, которые ориентированы на разные потребности. С полным списком реализаций `TaskExecutor` можно ознакомиться по ссылке `http://docs.spring.io/spring/docs/4.0.2.RELEASE/javadoc-api/org/springframework/core/task/TaskExecutor.html`.

Ниже приведен перечень распространенных реализаций `TaskExecutor`.

- **`SimpleAsyncTaskExecutor`.** Создает новые потоки при каждом вызове; существующие потоки повторно не используются.
- **`SyncTaskExecutor`.** Не выполняется асинхронным образом, и вызов происходит в вызывающем потоке.

- **`SimpleThreadPoolTaskExecutor`.** Подкласс класса `SimpleThreadPool` из библиотеки Quartz, который применяется, когда пул потоков необходимо совместно использовать компонентами, входящими в состав библиотеки Quartz и не входящими в нее.
- **`ThreadPoolTaskExecutor`.** Реализация `TaskExecutor` предоставляет возможность конфигурирования экземпляра класса `ThreadPoolExecutor` через свойства бина и открытия к нему доступа как к реализации `TaskExecutor` из Spring.

Каждая реализация `TaskExecutor` служит собственным целям, при этом применяются те же самые соглашения по вызову. Единственной вариацией является конфигурирование того, какую реализацию `TaskExecutor` нужно использовать, а также ее свойств, если они есть. Давайте взглянем на простой пример, который выводит несколько сообщений. В качестве реализации `TaskExecutor` мы будем применять `SimpleAsyncTaskExecutor`. Для начала создадим класс бина, содержащий логику выполнения задачи (листинг 11.17).

**Листинг 11.17. Пример бина `TaskToExecute`**

```
package com.apress.prospring4.ch11;
import org.springframework.core.task.TaskExecutor;
public class TaskToExecute {
   private TaskExecutor taskExecutor;
   public void executeTask() {
      for(int i=0; i < 10; i++) {
         taskExecutor.execute(new Runnable() {
            @Override
            public void run() {
               System.out.println("Hello from thread: "
                     + Thread.currentThread().getName());
            }
         });
      }
   }
   public void setTaskExecutor(TaskExecutor taskExecutor) {
      this.taskExecutor = taskExecutor;
   }
}
```

Этот класс представляет собой всего лишь обычный объект POJO, который принимает `TaskExecutor` как свойство и определяет метод `executeTask()`. Метод `executeTask()` вызывает метод `execute()` предоставленного экземпляра реализации `TaskExecutor` за счет создания нового экземпляра реализации `Runnable`, содержащего логику, которую мы хотим выполнить для данной задачи. Теперь давайте сконфигурируем бин (`app-context.xml`), как показано в листинге 11.18.

**Листинг 11.18. Конфигурация бина для выполнения задач**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:p="http://www.springframework.org/schema/p"
```

```
    xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="taskExecutorSample"
          class="com.apress.prospring4.ch11.TaskToExecute"
          p:taskExecutor-ref="taskExecutor"/>
    <bean id="taskExecutor"
          class="org.springframework.core.task.SimpleAsyncTaskExecutor"/>
</beans>
```

Здесь мы определяем новый бин по имени `taskExecutor` с использованием реализации `SimpleAsyncTaskExecutor` интерфейса `TaskExecutor`. Затем мы внедряем его в наш бин `TaskToExecute`. Давайте протестируем результирующий код (листинг 11.19).

**Листинг 11.19. Тестирование выполнения задач**

```
package com.apress.prospring4.ch11;
import org.springframework.context.support.GenericXmlApplicationContext;
public class TaskExecutorSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/app-context.xml");
        ctx.refresh();
        TaskToExecute taskToExecute = ctx.getBean(TaskToExecute.class);
        taskToExecute.executeTask();
    }
}
```

Выполнение примера приводит к получению вывода, подобного приведенному ниже:

```
Hello from thread: SimpleAsyncTaskExecutor-1
Hello from thread: SimpleAsyncTaskExecutor-3
Hello from thread: SimpleAsyncTaskExecutor-2
Hello from thread: SimpleAsyncTaskExecutor-5
Hello from thread: SimpleAsyncTaskExecutor-4
Hello from thread: SimpleAsyncTaskExecutor-6
Hello from thread: SimpleAsyncTaskExecutor-7
Hello from thread: SimpleAsyncTaskExecutor-8
Hello from thread: SimpleAsyncTaskExecutor-9
Hello from thread: SimpleAsyncTaskExecutor-10
```

Как можно заметить в выводе, каждая задача (выводимое сообщение) отображается по мере ее выполнения. Мы выводим сообщение вместе с именем потока, которым по умолчанию является имя класса (`SimpleAsyncTaskExecutor`), а также номер потока.

# Резюме

В этой главе была рассмотрена поддержка планирования задач в Spring. Мы сосредоточили внимание на встроенной в Spring абстракции `TaskScheduler` и продемонстрировали ее применение для удовлетворения потребностей планирования задач на примере с обновлением данных. Мы также показали, как в Spring поддерживается аннотация для асинхронного выполнения задач. Вдобавок мы кратко раскрыли доступный в Spring интерфейс `TaskExecutor` и его распространенные реализации.

ГЛАВА 12

# Использование удаленной обработки в Spring

Корпоративное приложение обычно нуждается во взаимодействии с другими приложениями. Возьмем, к примеру, компанию, продающую товары; когда пользователь размещает заказ, система обработки заказов обрабатывает его и генерирует транзакцию. Во время обработки заказа производится запрос к складской системе для проверки, доступен ли нужный товар на складе. После подтверждения заказа системе исполнения заказов отправляется уведомление о необходимости доставки товара заказчику. Наконец, соответствующая информация посылается системе бухгалтерского учета; генерируется счет-фактура и обрабатывается платеж.

Большую часть времени этот бизнес-процесс не может быть реализован каким-то одним приложением, поэтому для удовлетворения всех его требований вместе функционируют несколько приложений. Некоторые из таких приложений могут быть разработаны самостоятельно, а другие — приобретены у независимых поставщиков. Кроме того, приложения могут выполняться на отдельных машинах, расположенных в разных местах, и могут быть реализованы с помощью различных технологий и языков программирования (например, Java, .NET и C++). Подтверждение установления связи между приложениями для построения эффективного бизнес-процесса всегда является критически важной задачей при построении архитектуры и реализации приложения. В результате приложению требуется поддержка удаленной обработки через разнообразные протоколы и технологии, чтобы оно могло принимать активное участие в корпоративной среде.

В мире Java поддержка удаленной обработки существует с момента зарождения этого языка. На раннем этапе (Java 1.*x*) большинство требований удаленной обработки были реализованы с применением традиционных сокетов TCP или протокола RMI (Remote Method Invocation — удаленный вызов методов). С появлением J2EE обычным выбором для взаимодействия между приложениями на сервере стали EJB и JMS. Быстрое развитие XML и Интернета привело к возникновению удаленной поддержки с использованием XML через HTTP, включая Java API для протокола RPC на основе XML (JAX-RPC), Java API для веб-служб XML (JAX-RPC) и технологий, основанных на HTTP (например, Hessian и Burlap). Платформа Spring также предлагает собственную поддержку удаленной обработки на основе HTTP, которая называется

HTTP-активатором Spring (Spring HTTP invoker). Чтобы справиться с бурным ростом Интернета и требованиями большей отзывчивости веб-приложений (например, с помощью средств Ajax) в последние годы, критически важной для успеха предприятия становится более облегченная и эффективная удаленная поддержка приложений. В результате был создан интерфейс Java API для веб-служб REST (JAX-RS), который быстро обрел популярность. Немалую часть разработчиков привлекают и другие протоколы, такие как Comet и HTML5 WebSocket. Нечего и говорить, что технологии удаленной обработки продолжают развиваться быстрыми темпами.

Как уже было указано, Spring предоставляет собственную поддержку удаленной обработки (через HTTP-активатор Spring), а также поддержку множества технологий, упоминавшихся ранее (например, RMI, EJB, JMS, Hessian, Burlap, JAX-RPC, JAX-WS и JAX-RS). Раскрыть их все в одной главе невозможно, поэтому мы сосредоточим внимание только на самых часто используемых технологиях. В частности, в этой главе будут рассмотрены следующие темы.

- **HTTP-активатор Spring.** Если приложения, которые должны взаимодействовать, основаны на Spring, то HTTP-активатор Spring предоставляет простой и эффективный способ для обращения к службам, предлагаемым другими приложениями. Мы покажем, как использовать HTTP-активатор Spring для открытия службы на уровне обслуживания и как вызывать службы, предоставляемые удаленным приложением.
- **Использование JMS в Spring.** Служба обмена сообщениями Java (Java Messaging Service — JMS) предлагает еще один асинхронный и слабо связный способ обмена сообщениями между приложениями. Мы объясним, как Spring упрощает разработку приложений с помощью JMS.
- **Использование веб-служб REST в Spring.** Спроектированные специально для протокола HTTP, веб-службы REST являются наиболее часто применяемой технологией для предоставления удаленной поддержки приложениям, а также для поддержки интерактивных пользовательских интерфейсов веб-приложений, в которых используется Ajax. Мы продемонстрируем комплексную поддержку открытия служб с применением JAX-RS в инфраструктуре Spring MVC и объясним, как обращаться к службам с помощью класса `RestTemplate`. Мы также обсудим вопросы защиты служб от неавторизованного доступа.
- **Использование AMQP в Spring.** Родственный проект Spring Advanced Message Queuing Protocol (AMQP) предоставляет типичную Spring-образную абстракцию протокола AMQP наряду с реализацией RabbitMQ. Этот проект предлагает обширный набор возможностей, но в настоящей главе мы сосредоточим внимание на его функциональности удаленной обработки посредством поддержки RPC.

# Добавление обязательных зависимостей для серверной части JPA

В проект должны быть добавлены обязательные зависимости. В табл. 12.1 описаны зависимости, требуемые для реализации уровня обслуживания с применением JPA 2 и Hibernate в качестве поставщика постоянства. Кроме того, будет использоваться Spring Data JPA.

**Таблица 12.1. Зависимости Maven для уровня обслуживания**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| org.springframework | spring-context | 4.0.2.RELEASE | Модуль контекста Spring |
| org.springframework | spring-orm | 4.0.2.RELEASE | Модуль Spring ORM |
| org.springframework | spring-tx | 4.0.2.RELEASE | Модуль поддержки транзакций Spring |
| org.springframework | spring-web | 4.0.2.RELEASE | Веб-модуль Spring |
| org.springframework | spring-webmvc | 4.0.2.RELEASE | Веб-модуль Spring MVC |
| org.springframework.data | spring-data-jpa | 1.5.0.RELEASE | Библиотека Spring Data JPA |
| org.springframework | spring-oxm | 4.0.2.RELEASE | Модуль Spring OXM |
| org.hibernate | hibernate-entitymanager | 4.2.3.Final | Диспетчер сущностей Hibernate с поддержкой JPA 2 |
| org.hibernate.javax.persistence | hibernate-jpa-2.1-api | 1.0.0.Final | API-интерфейс JPA библиотеки Hibernate |
| com.h2database | h2 | 1.3.172 | База данных H2 для встроенного доступа JDBC |
| joda-time | joda-time | 2.3 | JodaTime — это API-интерфейс даты и времени, предназначенный для упрощения взаимодействия со встроенной библиотекой работы с датой и временем, которая входит в состав Java. В случае использования Java 8 платформа Spring также предоставляет полную поддержку API-интерфейса `javax.time`. В этой главе мы будем применять его в объектах предметной области |
| org.jadira.usertype | usertype.core | 3.0.0.GA | Библиотека JodaTime для интеграции с Hibernate, обеспечивающая сохранение данных типа даты и времени |
| com.google.guava | guava | 14.0.1 | Содержит полезные вспомогательные классы |
| log4j | log4j | 1.2.17 | Инфраструктура регистрации в журнале Log4j |
| org.codehaus.castor | castor-xml | 1.3.3 | Инфраструктура отображения Castor OXM |

*Окончание табл. 12.1*

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| com.fasterxml.jackson.core | jackson-core | 2.3.2 | Библиотека ядра Jackson |
| com.fasterxml.jackson.core | jackson-databind | 2.3.2 | Библиотека Jackson DataBind |
| org.springframework.security | spring-security-web | 3.2.1.RELEASE | Веб-модуль Spring Security |
| org.springframework.security | spring-security-config | 3.2.1.RELEASE | Модуль конфигурации Spring Security |
| org.apache.httpcomponents | httpclient | 4.3 | Библиотека Apache HTTPClient |

## Модель данных для примеров

В примерах этой главы мы будем использовать простую модель данных, которая содержит единственную таблицу CONTACT для хранения информации о контактах. В листинге 12.1 показан сценарий для создания схемы базы данных (schema.sql).

**Листинг 12.1. Сценарий для создания схемы базы данных**

```
DROP TABLE IF EXISTS CONTACT;

CREATE TABLE CONTACT (
    ID INT NOT NULL AUTO_INCREMENT
    , FIRST_NAME VARCHAR(60) NOT NULL
    , LAST_NAME VARCHAR(40) NOT NULL
    , BIRTH_DATE DATE
    , VERSION INT NOT NULL DEFAULT 0
    , UNIQUE UQ_CONTACT_1 (FIRST_NAME, LAST_NAME)
    , PRIMARY KEY (ID)
);
```

Как видите, таблица CONTACT содержит только несколько базовых полей информации о контакте. В листинге 12.2 приведен код сценария для наполнения таблицы тестовыми данными (test-data.sql).

**Листинг 12.2. Сценарий для наполнения тестовыми данными**

```
insert into contact (first_name, last_name, birth_date)
  values ('Chris', 'Schaefer', '1981-05-03');
insert into contact (first_name, last_name, birth_date)
  values ('Scott', 'Tiger', '1990-11-02');
insert into contact (first_name, last_name, birth_date)
  values ('John', 'Smith', '1964-02-28');
```

## Реализация и конфигурирование интерфейса ContactService

Добавив упомянутые выше зависимости, мы можем приступить к реализации и конфигурированию уровня обслуживания для примеров этой главы. В последующих разделах мы обсудим реализацию интерфейса ContactService с использованием JPA 2, Spring Data JPA и Hibernate в качестве поставщика постоянства. Затем мы покажем, как сконфигурировать уровень обслуживания в проекте Spring.

### Реализация интерфейса ContactService

В примерах этой главы мы будем открывать удаленным клиентам службы для разнообразных операций над контактной информацией. Прежде всего, необходимо создать сущностный класс Contact, который представлен в листинге 12.3.

**Листинг 12.3. Сущностный класс Contact**

```
package com.apress.prospring4.ch12;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;

import org.hibernate.annotations.Type;
import org.joda.time.DateTime;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Version;

@Entity
@Table(name = "contact")
public class Contact implements Serializable {
    private Long id;
    private int version;
    private String firstName;
    private String lastName;
    private DateTime birthDate;

    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "ID")
    public Long getId() {
        return id;
    }

    @Version
    @Column(name = "VERSION")
    public int getVersion() {
        return version;
    }

    @Column(name = "FIRST_NAME")
    public String getFirstName() {
        return firstName;
    }
```

```
    @Column(name = "LAST_NAME")
    public String getLastName() {
        return lastName;
    }
    @Column(name = "BIRTH_DATE")
    @Type(type="org.jadira.usertype.dateandtime.joda.PersistentDateTime")
    public DateTime getBirthDate() {
        return birthDate;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public void setVersion(int version) {
        this.version = version;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public void setBirthDate(DateTime birthDate) {
        this.birthDate = birthDate;
    }
    @Override
    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
               + ", Last name: " + lastName + ", Birthday: " + birthDate;
    }
}
```

В этом листинге видно, что применяются стандартные аннотации JPA. Мы также используем класс `DateTime` из Joda-Time для атрибута `birthDate`.

Давайте продолжим, реализовав уровень обслуживания; в листинге 12.4 приведен интерфейс `ContactService` со службами, которые должны быть открыты.

**Листинг 12.4. Интерфейс `ContactService`**

```
package com.apress.prospring4.ch12;

import java.util.List;

public interface ContactService {
    List<Contact> findAll();
    List<Contact> findByFirstName(String firstName);
    Contact findById(Long id);
    Contact save(Contact contact);
    void delete(Contact contact);
}
```

Методы этого интерфейса самоочевидны. Поскольку мы будем пользоваться поддержкой репозитория Spring Data JPA, реализуем интерфейс `ContactRepository` (листинг 12.5).

**Листинг 12.5. Интерфейс ContactRepository**

```
package com.apress.prospring4.ch12;
import java.util.List;
import org.springframework.data.repository.CrudRepository;
public interface ContactRepository extends CrudRepository<Contact, Long> {
    List<Contact> findByFirstName(String firstName);
}
```

За счет расширения интерфейса `CrudRepository<T, ID extends Serializable>` в `ContactService` понадобится явно объявить только метод `findByFirstName()`.

В листинге 12.6 приведен класс реализации интерфейса `ContactService`.

**Листинг 12.6. Класс ContactServiceImpl**

```
package com.apress.prospring4.ch12;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.google.common.collect.Lists;
@Service("contactService")
@Repository
@Transactional
public class ContactServiceImpl implements ContactService {
    @Autowired
    private ContactRepository contactRepository;
    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return Lists.newArrayList(contactRepository.findAll());
    }
    @Override
    @Transactional(readOnly=true)
    public List<Contact> findByFirstName(String firstName) {
        return contactRepository.findByFirstName(firstName);
    }
    @Override
    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        return contactRepository.findOne(id);
    }
    @Override
    public Contact save(Contact contact) {
        return contactRepository.save(contact);
    }
    @Override
    public void delete(Contact contact) {
        contactRepository.delete(contact);
    }
}
```

Реализация в основном завершена, и следующий шаг заключается в конфигурировании службы в ApplicationContext внутри проекта веб-приложения, что будет темой следующего раздела.

### Конфигурирование службы

Для доступа JPA мы создаем отдельный конфигурационный файл по имени datasource-tx-jpa.xml, содержимое которого показано в листинге 12.7.

**Листинг 12.7. Конфигурационный файл datasource-tx-jpa.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:jdbc="http://www.springframework.org/schema/jdbc"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd
      http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
      http://www.springframework.org/schema/data/jpa
      http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:META-INF/config/schema.sql"/>
      <jdbc:script location="classpath:META-INF/config/test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionManager" class="org.springframework.orm.jpa.
JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <tx:annotation-driven transaction-manager="transactionManager" />

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
       <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
       />
      </property>
      <property name="packagesToScan" value="com.apress.prospring4.ch12"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
```

```
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>

<context:annotation-config/>

<jpa:repositories base-package="com.apress.prospring4.ch12"
                  entity-manager-factory-ref="emf"
                  transaction-manager-ref="transactionManager"/>
</beans>
```

Поскольку мы открываем HTTP-активатор через Spring MVC, то должны импортировать конфигурацию в корневой контекст `WebApplicationContext`. Для проекта Spring MVC таким файлом является `/src/main/webapp/WEB-INF/spring/root-context.xml`. Модифицированное содержимое этого файла приведено в листинге 12.8.

**Листинг 12.8. Файл `root-context.xml`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <import resource="classpath:META-INF/spring/datasource-tx-jpa.xml" />

    <context:component-scan base-package="com.apress.prospring4.ch12" />
</beans>
```

Сначала в этот конфигурационный файл добавляется пространство имен `context`. Затем файл `datasource-tx-jpa.xml` импортируется в `WebApplicationContext` и, наконец, платформа Spring инструктируется о необходимости сканирования указанного пакета на предмет бинов Spring. Теперь уровень обслуживания завершен и готов к открытию и использованию удаленными клиентами.

# Использование HTTP-активатора Spring

Если приложение, с которым планируется взаимодействовать, также построено с помощью Spring, то удобным вариантом будет применение HTTP-активатора Spring. Он предлагает исключительно простой способ открытия служб внутри `WebApplicationContext` удаленным клиентам. Процедуры для открытия и доступа к службе описаны в следующих разделах.

## Открытие службы

Чтобы открыть службу, добавьте в конфигурационный файл `root-context.xml` определение бина, представленное в листинге 12.9.

**Листинг 12.9. Бин для открытия службы выдачи информации о контакте с использованием HTTP-активатора Spring**

```
   <bean name="contactExporter"
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
      <property name="service" ref="contactService" />
      <property name="serviceInterface"
         value="com.apress.prospring4.ch12.ContactService" />
   </bean>
```

Бин `contactExporter` определен с помощью класса `HttpInvokerServiceExporter`, который предназначен для экспорта любого бина Spring как службы через HTTP-активатор. Внутри бина определены два свойства. Первое из них, свойство `service`, указывает бин, предоставляющий службу. В это свойство внедрен бин `contactService`. Второе свойство, `serviceInterface`, задает тип открываемого интерфейса, которым является `com.apress.prospring4.ch12.ContactService`.

Далее мы должны определить сервлет для службы внутри дескриптора развертывания веб-приложения (`/src/main/webapp/WEB-INF/web.xml`). Определение сервлета внутри файла `web.xml` приведено в листинге 12.10.

**Листинг 12.10. Определение сервлета для HTTP-активатора**

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
       version="3.0">

   <display-name>Spring HTTP Invoker Sample</display-name>

   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/root-context.xml</param-value>
   </context-param>

   <listener>
     <listener-class>org.springframework.web.context.ContextLoaderListener
     </listener-class>
   </listener>

   <servlet>
      <servlet-name>contactExporter</servlet-name>
      <servlet-class>
         org.springframework.web.context.support.HttpRequestHandlerServlet
      </servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>contactExporter</servlet-name>
      <url-pattern>/remoting/ContactService</url-pattern>
   </servlet-mapping>
</web-app>
```

В листинге 12.10 определен сервлет с классом `HttpRequestHandlerServlet`, который используется для открытия экспортируемого бина Spring, определенного в `WebApplicationContext`. Обратите внимание, что имя сервлета (`contactExporter`) должно совпадать с именем экспортируемого бина (см. листинг 12.9; там именем бина также является `contactExporter`). Затем сервлет отображается на URL вида `/remoting/ContactService` в рамках веб-контекста (т.е. `http://localhost:8080/ch12/remoting/ContactService`) приложения.

К этому моменту разработка на стороне сервера завершена. Поскольку это веб-приложение Spring MVC, необходимо создать WAR-файл и развернуть его в контейнере сервлетов. Существует несколько способов сделать это — например, автономный контейнер наподобие Tomcat, экземпляр Tomcat, запускаемый из IDE-среды, или встроенный экземпляр Tomcat, который запускается с помощью инструмента построения, такого как Maven. Выбор подходящего варианта регламентируется вашими потребностями, но для локальной среды разработки рекомендуется применять встроенный экземпляр, запускаемый инструментом построения или напрямую из IDE-среды. В коде, сопровождаемом эту книгу, мы используем встроенный экземпляр посредством Maven. За дополнительными сведениями обращайтесь к исходному коду для книги.

Теперь вы должны построить веб-приложение и развернуть его через метод, выбранный по своему усмотрению. Если в результате перехода на URL сервлета `ContactService` (`http://localhost:8080/ch12/remoting/ContactService`) вы получаете ошибку с кодом 500, не беспокойтесь, т.к. необходимо еще иметь клиент, который сделает запрос.

## Вызов службы

Вызов службы через HTTP-активатор Spring осуществляется просто. Сначала нужно сконфигурировать `ApplicationContext`, как показано в листинге 12.11 (`http-invoker-app-context.xml`).

**Листинг 12.11. Конфигурация `ApplicationContext` для клиента HTTP-активатора**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="remoteContactService"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
      <property name="serviceUrl"
         value="http://localhost:8080/ch12/remoting/ContactService" />
      <property name="serviceInterface"
         value="com.apress.prospring4.ch12.ContactService" />
   </bean>
</beans>
```

В листинге 12.11 для клиентской стороны объявляется бин типа `HttpInvoker ProxyFactoryBean` и устанавливаются два его свойства.

Свойство `serviceUrl` указывает местоположение удаленной службы, которым является `http://localhost:8080/ch12/remoting/ContactService`. Свойство `serviceInterface` задает интерфейс службы (т.е. `ContactService`). Если для клиента разрабатывается отдельный проект, то интерфейс `ContactService` и сущностный класс `Contact` должны присутствовать в пути классов этого клиентского приложения.

В листинге 12.12 представлен основной класс для вызова удаленной службы.

**Листинг 12.12. Класс `HttpInvokerClientSample`**

```
package com.apress.prospring4.ch12;

import java.util.List;

import org.springframework.context.support.GenericXmlApplicationContext;

public class HttpInvokerClientSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/http-invoker-app-context.xml");
      ctx.refresh();

      ContactService contactService =
          ctx.getBean("remoteContactService", ContactService.class);
      System.out.println("Finding all contacts");
      List<Contact> contacts = contactService.findAll();
      listContacts(contacts);

      System.out.println("Finding contact with first name equals Chris");
      contacts = contactService.findByFirstName("Chris");
      listContacts(contacts);
   }
   private static void listContacts(List<Contact> contacts) {
       for (Contact contact: contacts) {
          System.out.println(contact);
       }
       System.out.println("");
   }
}
```

В листинге 12.12 видно, что эта программа очень похожа на любое другое автономное приложение Spring. Сначала инициализируется `ApplicationContext`, после чего извлекается бин `contactService`. Затем просто вызываются его методы, как в локальном приложении. Запуск программы дает следующий вывод:

```
Finding all contacts
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday:
1981-05-02T20:00:00.000-04:00
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday:
1990-11-01T19:00:00.000-05:00
Contact - Id: 3, First name: John, Last name: Smith, Birthday:
1964-02-27T19:00:00.000-05:00

Finding contact with first name equals Chris
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday:
1981-05-02T20:00:00.000-04:00
```

В выводе легко заметить, что методы `findAll()` и `findByFirstName()` были успешно вызваны и результаты возвращены.

# Использование JMS в Spring

Использование промежуточного программного обеспечения, ориентированного на сообщения (которое в общем случае называют *сервером MQ*) — это еще один популярный способ поддержки взаимодействия между приложениями. Основное преимущество сервера очереди сообщений (message queue — MQ) заключается в том, что он предлагает асинхронный и слабо связный путь для интеграции приложений. В мире Java стандартом для подключения к серверу MQ с целью отправки или получения сообщений является JMS. Сервер MQ поддерживает список очередей и тем, для которых приложения могут подключаться, отправлять и получать сообщения. Ниже приведено краткое описание отличий между очередью и темой.

- **Очередь.** Очередь применяется для поддержки модели обмена сообщениями вида "точка-точка". Когда генератор отправляет сообщение в очередь, сервер MQ сохраняет это сообщение в очереди и доставляет его одному и только одному потребителю при следующем его подключении.
- **Тема.** Тема используется для поддержки модели обмена сообщениями вида "публикация-подписка". На сообщение внутри темы может подписываться любое количество клиентов. Когда сообщение поступает в заданную тему, сервер MQ доставляет его всем клиентам, которые на него подписаны. Эта модель особенно удобна в ситуации, когда имеется множество приложений, заинтересованных в получении одной и той же порции информации (например, лента новостей).

В стандарте JMS генератор подключается к серверу MQ и отправляет сообщение в очередь или в тему. Потребитель также подключается к серверу MQ и прослушивает очередь или темы на предмет наличия интересующих его сообщений. В JMS 1.1 был унифицирован API-интерфейс, так что генератор и потребитель не должны были иметь дело с разными API-интерфейсами при взаимодействии с очередями и темами. В этом разделе мы сосредоточим внимание на модели "точка-точка" работы с очередями, которая представляет собой более часто применяемый шаблон в корпоративной среде. Для разработки и тестирования приложения JMS требуется сервер MQ.

В данном разделе мы будем пользоваться сервером Apache ActiveMQ (`activemq.apache.org`) — популярным сервером MQ с открытым кодом.

Пример требует нескольких новых зависимостей Maven, которые описаны в табл. 12.2. Добавьте их в проект.

**Таблица 12.2. Зависимости Maven для JMS и ActiveMQ**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.springframework` | `spring-jms` | 4.0.2.RELEASE | Модуль Spring JMS |
| `org.apache.activemq` | `activemq-core` | 5.7.0 | Java-библиотека ActiveMQ |
| `javax.jms` | `jms` | 1.1 | API-интерфейс JMS 1.1 |

## Установка сервера ActiveMQ

Установка сервера ActiveMQ для использования при разработке производится легко. Подобно Tomcat в предыдущем примере, ActiveMQ может функционировать как автономный сервер или встраиваться и запускаться из инструмента построения (вроде Maven). Для целей разработки мы применяем встроенную версию, как делали это с Tomcat. Если вы хотите установить автономный сервер, обратитесь за дополнительной информацией на страницу загрузки веб-сайта ActiveMQ (`http://activemq.apache.org/download.html`). Чтобы увидеть, каким образом сконфигурирован встроенный экземпляр ActiveMQ в системе построения, просмотрите исходный код примеров, доступный для этой книги.

## Реализация прослушивателя JMS в Spring

Для разработки прослушивателя сообщений понадобится создать класс, который реализует интерфейс `javax.jms.MessageListener` и его метод `onMessage()`. Этот класс показан в листинге 12.13.

**Листинг 12.13. Класс прослушивателя сообщений JMS**

```
package com.apress.prospring4.ch12;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SimpleMessageListener implements MessageListener {
    private static final Logger logger =
        LoggerFactory.getLogger(SimpleMessageListener.class);

    @Override
    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage) message;

        try {
            logger.info("Message received: " + textMessage.getText());
        } catch (JMSException ex) {
            logger.error("JMS error", ex);
        }
    }
}
```

При поступлении сообщения методу `onMessage()` передается экземпляр интерфейса `javax.jms.Message`. Внутри этого метода сообщение приводится к экземпляру интерфейса `javax.jms.TextMessage`, а затем с помощью метода `TextMessage.getText()` извлекается тело сообщения. Список возможных форматов сообщения приведен в онлайновой документации по JEE.

После построения прослушивателя сообщений следующий шаг заключается в определении конфигурации `ApplicationContext`. В листинге 12.14 представлено содержимое файла `jms-listener-app-context.xml`.

**Листинг 12.14. Конфигурация прослушивателя сообщений JMS**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jms="http://www.springframework.org/schema/jms"
   xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/jms
      http://www.springframework.org/schema/jms/spring-jms.xsd">

   <bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory"
      p:brokerURL="tcp://localhost:61616" />
   <bean id="simpleMessageListener"
         class="com.apress.prospring4.ch12.SimpleMessageListener"/>
   <jms:listener-container container-type="default"
      connection-factory="connectionFactory" acknowledge="auto">
      <jms:listener destination="prospring4"
          ref="simpleMessageListener" method="onMessage" />
   </jms:listener-container>
</beans>
```

В коде сначала мы объявляем интерфейс `javax.jms.ConnectionFactory`, предоставленный Java-библиотекой ActiveMQ (класс `ActiveMQConnectionFactory`). Затем мы объявляем бин типа `SimpleMessageListener`. Наконец, мы используем удобный дескриптор `<jms:listener-container>`, предлагаемый пространством имен `jms` в Spring, для объявления прослушивателя сообщений с указанием получателя (т.е. очереди `prospring4`), ссылки на бин и метода, вызываемого при поступлении сообщения.

Теперь давайте посмотрим, как отправить сообщение в очередь `prospring4`.

## Отправка сообщений JMS в Spring

В этом разделе мы покажем, каким образом отправлять сообщения с применением JMS в Spring. Для этой цели мы будем использовать удобный класс `org.springframework.jms.core.JmsTemplate`. Первым делом, мы разработаем интерфейс `MessageSender` и класс его реализации `SimpleMessageSender`. Соответствующий код приведен в листингах 12.15 и 12.16.

**Листинг 12.15. Интерфейс `MessageSender`**

```
package com.apress.prospring4.ch12;

public interface MessageSender {
   void sendMessage(String message);
}
```

**Листинг 12.16. Класс `SimpleMessageSender`**

```
package com.apress.prospring4.ch12;
import javax.jms.JMSException;
```

```
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import org.springframework.stereotype.Component;

@Component("messageSender")
public class SimpleMessageSender implements MessageSender {
    @Autowired
    private JmsTemplate jmsTemplate;

    @Override
    public void sendMessage(final String message) {
        this.jmsTemplate.send(new MessageCreator() {
            @Override
            public Message createMessage(Session session)
                throws JMSException {
                return session.createTextMessage(message);
            }
        });
    }
}
```

Несложно заметить, что был внедрен экземпляр `JmsTemplate`.

В методе `sendMessage()` мы вызываем метод `JmsTemplate.send()` с конструированием на месте экземпляра интерфейса `org.springframework.jms.core.MessageCreator`.

В экземпляре `MessageCreator` реализован метод `createMessage()`, создающий новый экземпляр `TextMessage`, который будет отправлен ActiveMQ.

В листинге 12.17 показана конфигурация Spring для отправки сообщений JMS (`jms-sender-app-context.xml`).

**Листинг 12.17. Конфигурация Spring для отправки сообщений**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">
    <bean id="connectionFactory"
        class="org.apache.activemq.ActiveMQConnectionFactory"
        p:brokerURL="tcp://localhost:61616" />
    <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
        <constructor-arg name="connectionFactory" ref="connectionFactory"/>
        <property name="defaultDestinationName" value="prospring4"/>
    </bean>
    <context:component-scan base-package="com.apress.prospring4.ch12"/>
</beans>
```

Здесь, как обычно, определяется бин `connectionFactory`. Кроме того, объявляется экземпляр `JmsTemplate` с аргументом конструктора `connectionFactory` и свойством `defaultDestinationName`, установленным в очередь `prospring4`.

А теперь давайте увяжем вместе отправку и получение, чтобы посмотреть на JMS в действии. В листинге 12.18 представлена программа для тестирования отправки и получения сообщений.

**Листинг 12.18. Тестирование отправки и получения сообщений**

```
package com.apress.prospring4.ch12;

import org.springframework.context.support.GenericXmlApplicationContext;

public class JmsSample {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/jms-sender-app-context.xml",
            "classpath:META-INF/spring/jms-listener-app-context.xml");
      ctx.refresh();

      MessageSender messageSender =
         ctx.getBean("messageSender", MessageSender.class);

      for(int i=0; i < 10; i++) {
         messageSender.sendMessage("Test message: " + i);
      }
   }
}
```

Тестовая программа проста. После ее запуска сообщения отправляются в очередь. Программа с классом `JmsListenerSample` получает эти сообщения, в результате чего вы увидите на консоли следующий вывод:

```
INFO ss.prospring4.ch12.SimpleMessageListener:  19 - Message received:
Test message: 0
INFO ss.prospring4.ch12.SimpleMessageListener:  19 - Message received:
Test message: 1
INFO ss.prospring4.ch12.SimpleMessageListener:  19 - Message received:
Test message: 2
INFO ss.prospring4.ch12.SimpleMessageListener:  19 - Message received:
Test message: 3
INFO ss.prospring4.ch12.SimpleMessageListener:  19 - Message received:
Test message: 4
INFO ss.prospring4.ch12.SimpleMessageListener:  19 - Message received:
Test message: 5
INFO ss.prospring4.ch12.SimpleMessageListener:  19 - Message received:
Test message: 6
INFO ss.prospring4.ch12.SimpleMessageListener:  19 - Message received:
Test message: 7
INFO ss.prospring4.ch12.SimpleMessageListener:  19 - Message received:
Test message: 8
INFO ss.prospring4.ch12.SimpleMessageListener:  19 - Message received:
Test message: 9
```

В реальности сообщение, скорее всего, будет иметь формат XML и представлять порцию бизнес-информации (например, онлайновый заказ, транзакцию или запрос на резервирование). Как было показано, реализация отправки и получения сообщений посредством Spring JMS довольно проста. В этом разделе были продемонстрированы только базовые сценарии использования JMS. За дополнительными сведениями обращайтесь к онлайновой документации по JEE.

## Работа с JMS 2.0

В версии Spring Framework 4.0 была реализована поддержка JMS 2.0. Функциональность JMS 2.0 доступна из-за наличия JAR-файла JMS 2.0 в пути классов и сохраняет обратную совместимость с версиями JMS 1.*x*. На момент написания этой книги сервер ActiveMQ не поддерживал JMS 2.0, поэтому в рассматриваемом примере мы задействуем систему HornetQ (которая включает поддержку JMS 2.0, начиная с версии 2.4.0.Final) в качестве брокера сообщений и будем применять автономный сервер. Загрузка и установка HornetQ выходит за рамки настоящей книги; документация по HornetQ доступна по ссылке `http://docs.jboss.org/hornetq/2.4.0.Final/docs/quickstart-guide/html/index.html`.

В этом примере мы используем код из примера JMS 1.1 / ActiveMQ, но с небольшими изменениями в конфигурации, предназначенными для HornetQ. Нам также понадобится зависимость для JMS 2.0, а не для JMS 1.1, как показано в табл. 12.3.

**Таблица 12.3. Зависимости Maven для примера JMS 2.0**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.springframework` | `spring-jms` | 4.0.2.RELEASE | Модуль Spring JMS |
| `org.slf4j` | `slf4j-log4j12` | 1.7.6 | Адаптер log4j из библиотеки ведения журналов SLF4J |
| `javax.jms` | `javax.jms-api` | 2.0 | API-интерфейс JMS 2.0 |
| `org.hornetq` | `hornet-jms-client` | 2.4.0.Final | Клиентский API-интерфейс HornetQ JMS |

Сначала внутри конфигурационного файла HornetQ JMS необходимо создать очередь. Этот файл находится в каталоге, куда вы извлекли HornetQ. Местоположение файла выглядит как `config/stand-alone/non-clustered/hornetq-jms.xml`, и нам потребуется добавить определение очереди, как продемонстрировано в листинге 12.19.

**Листинг 12.19. Определение очереди HornetQ**

```
<configuration xmlns="urn:hornetq"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="urn:hornetq /schema/hornetq-jms.xsd">
   ...
   <queue name="prospring4">
      <entry name="/queue/prospring4"/>
   </queue>
</configuration>
```

Теперь запустите сервер HornetQ с помощью сценария `run.sh` (это зависит от операционной системы) и удостоверьтесь в том, что процедура запуска прошла без ошибок. Затем понадобится модифицировать конфигурационные файлы для работы с фабрикой подключений HornetQ вместо ActiveMQ. Прежде всего, внесем изменения в конфигурацию Spring для отправки сообщений (`jms-sender-app-context.xml`), как показано в листинге 12.20.

**Листинг 12.20. Конфигурация Spring для отправки сообщений HornetQ**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">

   <bean id="connectionFactory"
         class="org.hornetq.jms.client.HornetQJMSConnectionFactory">
      <constructor-arg name="ha" value="false" />
      <constructor-arg>
         <bean class="org.hornetq.api.core.TransportConfiguration">
            <constructor-arg
              value="org.hornetq.core.remoting.impl.netty.NettyConnectorFactory"
            />
            <constructor-arg>
              <map key-type="java.lang.String" value-type="java.lang.Object">
                   <entry key="host" value="127.0.0.1"/>
                   <entry key="port" value="5445"/>
              </map>
            </constructor-arg>
         </bean>
      </constructor-arg>
   </bean>

   <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
      <constructor-arg name="connectionFactory" ref="connectionFactory"/>
      <property name="defaultDestinationName" value="prospring4"/>
   </bean>

   <context:component-scan base-package="com.apress.prospring4.ch12"/>
</beans>
```

Изменения касаются только бина `connectionFactory` и связаны с HornetQ. За дополнительными сведениями обращайтесь к документации, URL которой был приведен в начале этого раздела. Теперь давайте модифицируем клиентскую конфигурацию Spring (`jms-listener-app-context.xml`) согласно листингу 12.21.

**Листинг 12.21. Клиентская конфигурация HornetQ**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
      xmlns:jms="http://www.springframework.org/schema/jms"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/jms
       http://www.springframework.org/schema/jms/spring-jms.xsd">

   <bean id="connectionFactory"
         class="org.hornetq.jms.client.HornetQJMSConnectionFactory">
      <constructor-arg name="ha" value="false" />
      <constructor-arg>
         <bean class="org.hornetq.api.core.TransportConfiguration">
            <constructor-arg
             value="org.hornetq.core.remoting.impl.netty.NettyConnectorFactory"
             />
            <constructor-arg>
              <map key-type="java.lang.String" value-type="java.lang.Object">
                   <entry key="host" value="127.0.0.1"/>
                   <entry key="port" value="5445"/>
              </map>
            </constructor-arg>
         </bean>
      </constructor-arg>
   </bean>

   <bean id="simpleMessageListener"
         class="com.apress.prospring4.ch12.SimpleMessageListener"/>

   <jms:listener-container container-type="default"
                           connection-factory="connectionFactory"
                           acknowledge="auto">
      <jms:listener destination="prospring4" ref="simpleMessageListener"
         method="onMessage" />
   </jms:listener-container>
</beans>
```

Наконец, изменим код класса `SimpleMessageSender`, который имеет доступ к `JmsTemplate` (листинг 12.22). Мы установим свойство задержки доставки (Delivery Delay) экземпляра `JmsTemplate`, которое является особенностью JMS 2.0. Это свойство задает минимальный промежуток времени в миллисекундах, который должен пройти после отправки сообщения, прежде чем брокер JMS сможет доставить сообщение потребителю. Если вы располагаете JAR-файлом не JMS 2.0, а JMS 1.*x*, то при попытке применения этого метода Spring сгенерирует исключение, указывающее на необходимость наличия JMS 2.0.

**Листинг 12.22. Установка задержки доставки в генераторе сообщений JMS 2.0**

```
package com.apress.prospring4.ch12;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import org.springframework.stereotype.Component;
```

```
@Component("messageSender")
public class SimpleMessageSender implements MessageSender {
    @Autowired
    private JmsTemplate jmsTemplate;

    @Override
    public void sendMessage(final String message) {
        jmsTemplate.setDeliveryDelay(5000L);

        this.jmsTemplate.send(new MessageCreator() {
            @Override
            public Message createMessage(Session session)
                    throws JMSException {
                return session.createTextMessage(message);
            }
        });
    }
}
```

Запустив пример снова, вы увидите те же самые результаты, но только доставка сообщений будет производиться с задержкой.

# Использование веб-служб REST в Spring

В наши дни веб-службы REST (RESTful-WS) являются, вероятно, самой широко используемой технологией для удаленного доступа. Веб-службы REST интенсивно применяются везде, от вызова удаленных служб через HTTP до поддержки интерактивного пользовательского интерфейса веб-приложения в стиле Ajax.

Популярность веб-служб REST объясняется несколькими причинами.

- **Простота понимания.** Веб-службы REST спроектированы на основе протокола HTTP. Адрес URL вместе с HTTP-методом определяют намерение запроса. Например, URL вида `http://somedomain.com/restful/customer/1` с HTTP-методом `GET` означает, что клиент желает извлечь информацию о заказчике, идентификатор которого равен 1.
- **Облегченность.** Веб-службы REST являются более легковесными по сравнению с веб-службами на основе SOAP, которые требуют большого объема метаданных для описания того, какую службу клиент должен вызывать. Запрос и ответ REST представляют собой просто запрос и ответ HTTP, как в любом другом веб-приложении.
- **Дружественность по отношению к брандмауэру.** Поскольку веб-службы REST предназначены для доступа по протоколу HTTP (или HTTPS), приложение становится более дружественным к брандмауэру и легко доступным для удаленных клиентов.

В этом разделе мы обсудим базовые концепции веб-служб REST и их поддержку в Spring посредством модуля Spring MVC.

## Введение в веб-службы REST

Аббревиатура *REST* означает *RE*presentational *S*tate *T*ransfer (передача состояния представления). Технология REST определяет набор архитектурных ограничений, которые вместе описывают *унифицированный интерфейс* для доступа к ресурсам.

Основные концепции этого унифицированного интерфейса включает идентификацию ресурсов и манипулирование ресурсами через представления.

Для идентификации ресурсов порция информации должна быть доступна через унифицированный идентификатор ресурса (Uniform Resource Identifier — URI). Например, URL вида `www.somedomain.com/api/contact/1` — это URI, представляющий ресурс, который является порцией информации о контакте с идентификатором 1. Если контакт с идентификатором 1 не существует, клиент получит HTTP-ошибку 404, как и в случае, когда на веб-сайте отсутствует запрошенная страница. Еще одним примером может служить `www.somedomain.com/api/contacts` — URI, представляющий ресурс, который является списком информации о контактах.

Такими идентифицируемыми ресурсами можно управлять посредством различных представлений, которые описаны в табл. 12.4.

**Таблица 12.4. Представления для манипулирования ресурсами**

| Представление | Описание |
|---|---|
| `GET` | Извлекает представление ресурса |
| `HEAD` | Идентично `GET`, но без тела ответа; обычно используется для получения заголовка |
| `POST` | Создает новый ресурс |
| `PUT` | Обновляет ресурс |
| `DELETE` | Удаляет ресурс |
| `OPTIONS` | Извлекает разрешенные HTTP-методы |

Детальное описание веб-служб REST можно найти в книге *Ajax and REST Recipes: A Problem-Solution Approach* (Apress, 2006 г.).

## Добавление обязательных зависимостей для примеров

Для разработки примеров этого раздела требуются зависимости, которые описаны в табл. 12.5. Добавьте их в свой проект.

**Таблица 12.5. Зависимости Maven для веб-служб REST**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.springframework` | `spring-oxm` | 4.0.2.RELEASE | Модуль отображения объектов Spring на XML |
| `org.codehaus.jackson` | `jackson-mapper-lgpl` | 1.9.13 | Процессор Jackson JSON для поддержки данных в формате JSON |

*Окончание табл. 12.5*

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| org.codehaus.castor | castor-xml | 1.3.3 | Библиотека Castor XML, которая будет использоваться для маршализации и демаршализации XML-данных |
| org.springframework.security | spring-security-core | 3.2.1.RELEASE | Модуль ядра Spring Security |
| org.springframework.security | spring-security-web | 3.2.1.RELEASE | Веб-модуль Spring Security для защиты веб-служб REST |
| org.springframework.security | spring-security-config | 3.2.1.RELEASE | Модуль конфигурации Spring Security |
| org.apache.httpcomponents | httpclient | 4.3 | Проект Apache HTTP Components. Библиотека клиента HTTP будет использоваться для вызова веб-служб REST |

## Проектирование веб-службы REST для контактной информации

Первый шаг при разработке приложения RESTful-WS заключается в проектировании структуры служб, которая включает поддерживаемые HTTP-методы и целевые URL для различных операций. В веб-службах REST для контактов мы хотим поддерживать операции запроса, создания, обновления и удаления. В рамках операции запроса нам нужно поддерживать извлечение всех контактов и одиночного контакта по идентификатору. Службы будут реализованы в виде контроллера Spring MVC. Именем класса будет `ContactController` из пакета `com.apress.prospring4.ch12`. Шаблон URL, HTTP-метод, описание и соответствующие методы контроллера показаны в табл. 12.6. Все URL используют префикс `http://localhost:8080/ch12/restful`. В качестве форматов данных будут поддерживаться XML и JSON. Подходящий формат будет предоставляться согласно атрибуту `Accept` в заголовке клиентского HTTP-запроса.

**Таблица 12.6. Проектное решение для веб-служб REST контактов**

| URL | HTTP-метод | Описание | Метод контроллера |
|---|---|---|---|
| /contact/listdata | GET | Извлечение всех контактов | listData(...) |
| /contact/{id} | GET | Извлечение одиночного контакта с указанным идентификатором | findContactById(...) |
| /contact | POST | Создание нового контакта | create(...) |
| /contact/{id} | PUT | Обновление существующего контакта с указанным идентификатором | update(...) |
| /contact | DELETE | Удаление контакта с указанным идентификатором | delete(...) |

## Использование Spring MVC для открытия веб-служб REST

В этом разделе мы покажем, как использовать Spring MVC для открытия служб, связанных с контактами, в качестве веб-служб REST согласно проектному решению, представленному в предыдущем разделе. Этот пример построен на основе ряда классов `ContactService`, которые применялись в примере HTTP-активатора Spring.

Для начала мы создадим еще один объект предметной области — класс `Contacts`. Его код приведен в листинге 12.23.

**Листинг 12.23. Класс `Contacts`**

```
package com.apress.prospring4.ch12;

import java.io.Serializable;
import java.util.List;

public class Contacts implements Serializable {
    private List<Contact> contacts;
    public Contacts() {
    }
    public Contacts(List<Contact> contacts) {
        this.contacts = contacts;
    }
    public List<Contact> getContacts() {
        return contacts;
    }
    public void setContacts(List<Contact> contacts) {
        this.contacts = contacts;
    }
}
```

Класс `Contacts` имеет единственное свойство, представляющее собой список объектов `Contact`. Целью является поддержка преобразования списка контактов (возвращаемого методом `listData()` класса `ContactController`) в формат XML или JSON.

### *Конфигурирование библиотеки Castor XML*

Для преобразования возвращаемой информации о контактах в формат XML мы будем пользоваться библиотекой Castor XML (`http://castor.codehaus.org`). Эта библиотека поддерживает множество режимов преобразования между объектами POJO и XML, и в рассматриваемом примере для определения отображений мы будем применять XML-файл. Содержимое файла отображений (`oxm-mapping.xml`) приведено в листинге 12.24.

**Листинг 12.24. Определение отображений для Castor XML**

```
<mapping>
    <class name="com.apress.prospring4.ch12.Contacts">
        <field name="contacts" type="com.apress.prospring4.ch12.Contact"
               collection="arraylist">
            <bind-xml name="contact" />
        </field>
    </class>
```

```
    <class name="com.apress.prospring4.ch12.Contact" identity="id">
        <map-to xml="contact" />
        <field name="id" type="long">
            <bind-xml name="id" node="element"/>
        </field>
        <field name="firstName" type="string">
            <bind-xml name="firstName" node="element" />
        </field>
        <field name="lastName" type="string">
            <bind-xml name="lastName" node="element" />
        </field>
        <field name="birthDate" type="string" handler="dateHandler">
            <bind-xml name="birthDate" node="element" />
        </field>
        <field name="version" type="integer">
            <bind-xml name="version" node="element" />
        </field>
    </class>
    <field-handler name="dateHandler"
            class="com.apress.prospring4.ch12.DateTimeFieldHandler">
        <param name="date-format" value="yyyy-MM-dd" />
    </field-handler>
</mapping>
```

Здесь определены два отображения. Первый дескриптор `<class>` отображает класс `Contacts`, а внутри него свойство `contacts` (список объектов `Contact`) отображается с использованием дескриптора `<bind-xml name="contact" />`. Затем отображается объект `Contact` (с помощью дескриптора `<map-to xml="contact" />` внутри второго дескриптора `<class>`). Кроме того, для поддержки преобразования из типа `DateTime`, определенного в Joda-Time (для атрибута `birthDate` в `Contact`), мы реализуем специальный обработчик полей Castor XML. Этот обработчик полей представлен в листинге 12.25.

**Листинг 12.25. Специальный обработчик полей для типа `DateTime` в Castor XML**

```
package com.apress.prospring4.ch12;

import java.util.Properties;

import org.exolab.castor.mapping.GeneralizedFieldHandler;
import org.exolab.castor.mapping.ValidityException;
import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;

public class DateTimeFieldHandler extends GeneralizedFieldHandler {
    private static String dateFormatPattern;

    @Override
    public void setConfiguration(Properties config) throws ValidityException
    {
        dateFormatPattern = config.getProperty("date-format");
    }
```

```
    @Override
    public Object convertUponGet(Object value) {
        DateTime dateTime = (DateTime) value;

        return format(dateTime);
    }
    @Override
    public Object convertUponSet(Object value) {
        String dateTimeString = (String) value;

        return parse(dateTimeString);
    }
    @Override
    public Class<DateTime> getFieldType() {
        return DateTime.class;
    }
    protected static String format(final DateTime dateTime) {
        String dateTimeString = "";

        if (dateTime != null) {
            DateTimeFormatter dateTimeFormatter =
                    DateTimeFormat.forPattern(dateFormatPattern);
            dateTimeString = dateTimeFormatter.print(dateTime);
        }

        return dateTimeString;
    }
    protected static DateTime parse(final String dateTimeString) {
        DateTime dateTime = new DateTime();

        if (dateTimeString != null) {
            DateTimeFormatter dateTimeFormatter =
                    DateTimeFormat.forPattern(dateFormatPattern);
            dateTime = dateTimeFormatter.parseDateTime(dateTimeString);
        }

        return dateTime;
    }
}
```

Мы расширяем класс `org.exolab.castor.mapping.GeneralizedFieldHandler` из Castor XML и реализуем методы `convertUponGet()`, `convertUponSet()` и `getFieldType()`. Внутри этих методов мы реализуем логику преобразования между `DateTime` и `String`, предназначенную для Castor XML.

Вдобавок мы также определяем файл свойств для использования с Castor XML. Содержимое этого файла (`castor.properties`) приведено в листинге 12.26.

**Листинг 12.26. Файл `castor.properties`**

```
org.exolab.castor.indent=true
```

Это свойство указывает библиотеке Castor XML на необходимость генерации XML-кода с отступами, что намного упрощает его чтение во время тестирования.

### *Реализация класса `ContactController`*

Следующий шаг заключается в реализации класса контроллера, `ContactController`. В листинге 12.27 показан класс `ContactController`, в котором реализованы все методы из табл. 12.6.

**Листинг 12.27. Класс `ContactController`**

```
package com.apress.prospring4.ch12;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@RequestMapping(value="/contact")
public class ContactController {
    final Logger logger = LoggerFactory.getLogger(ContactController.class);

    @Autowired
    private ContactService contactService;

    @RequestMapping(value = "/listdata", method = RequestMethod.GET)
    @ResponseBody
    public Contacts listData() {
        return new Contacts(contactService.findAll());
    }

    @RequestMapping(value="/{id}", method=RequestMethod.GET)
    @ResponseBody
    public Contact findContactById(@PathVariable Long id) {
        return contactService.findById(id);
    }

    @RequestMapping(value="/", method=RequestMethod.POST)
    @ResponseBody
    public Contact create(@RequestBody Contact contact) {
        logger.info("Creating contact: " + contact);
        contactService.save(contact);
        logger.info("Contact created successfully with info: " + contact);
        return contact;
    }

    @RequestMapping(value="/{id}", method=RequestMethod.PUT)
    @ResponseBody
    public void update(@RequestBody Contact contact,
        @PathVariable Long id) {
        logger.info("Updating contact: " + contact);
        contactService.save(contact);
        logger.info("Contact updated successfully with info: " + contact);
    }
```

```
    @RequestMapping(value="/{id}", method=RequestMethod.DELETE)
    @ResponseBody
    public void delete(@PathVariable Long id) {
        logger.info("Deleting contact with id: " + id);
        Contact contact = contactService.findById(id);
        contactService.delete(contact);
        logger.info("Contact deleted successfully");
    }
}
```

Ниже перечислены основные моменты, связанные с листингом 12.27.

- Класс снабжен аннотацией `@Controller`, указывающей на то, что он является контроллером Spring MVC.
- Аннотация уровня класса `@RequestMapping(value="/contact")` определяет, что этот контроллер будет отображаться на все URL в главном веб-контексте. В рассматриваемом примере данный контроллер будет обрабатывать все URL, начинающиеся с `http://localhost:8080/ch12/contact`.
- Реализованный ранее в этой главе класс `ContactService` на уровне обслуживания автоматически связан с контроллером.
- С помощью аннотаций `@RequestMapping` в каждом методе класса `ContactController` указываются шаблон URL и соответствующий HTTP-метод, на который будет производиться отображение. Например, метод `listData()` будет отображаться на URL вида `http://localhost:8080/ch12/contact/listdata` с HTTP-методом `GET`. Метод `update()` будет отображаться на URL вида `http://localhost:8080/ch12/contact/{id}` с HTTP-методом `PUT`.
- Аннотация `@ResponseBody` применяется ко всем методам. Она обеспечивает запись всех возвращаемых методами значений напрямую в поток HTTP-ответа.
- В методах, принимающих переменные пути (например, `findContactById()`), эта переменная пути аннотирована посредством `@PathVariable`. Это заставляет Spring MVC связывать переменную пути внутри URL (к примеру, `http://localhost:8080/ch12/contact/1`) с аргументом `id` метода `findContactById()`. Обратите внимание, что аргумент `id` имеет тип `Long`, но система преобразования типов Spring автоматически выполнит проеобразование из `String` в `Long`.
- В методах `create()` и `update()` аргумент `Contact` аннотирован с помощью `@RequestBody`. Это заставляет Spring автоматически связывать содержимое тела HTTP-запроса с объектом предметной области `Contact`. Преобразование будет осуществляться объявленными экземплярами интерфейса `HttpMessageConverter<Object>` (из пакета `org.springframework.http.converter`) для поддержки форматов, которые обсуждаются далее в этой главе.

### *Конфигурирование сервлета REST*

После завершения разработки контроллера его можно определить в Spring MVC. Сначала понадобится определить сервлет `DispatcherServlet` (из пакета `org.springframework.web.servlet`) для указания Spring MVC на необходимость диспетчеризации всех запросов REST к контроллеру `ContactController`. Чтобы объявить сервлет, добавьте фрагмент кода из листинга 12.28 в файл дескриптора развертывания веб-приложений (`src/main/webapp/WEB-INF/web.xml`).

**Листинг 12.28. Сервлет диспетчера для веб-служб REST**

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

   <display-name>Spring REST Sample</display-name>

   <listener>
     <listener-class>org.springframework.web.context.ContextLoaderListener
     </listener-class>
   </listener>

   <servlet>
      <servlet-name>restful</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>/WEB-INF/spring/rest-context.xml</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>restful</servlet-name>
      <url-pattern>/restful/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Мы объявили сервлет по имени `restful`, имеющий тип `DispatcherServlet` (класс `DispatcherServlet` обсуждается в главе 16). В Spring MVC каждый экземпляр `DispatchServlet` будет иметь собственный контекст `WebApplicationContext` (однако все бины уровня обслуживания, определенные в файле `root-context.xml`, который вызывается корневым `WebApplicationContext`, будут доступны также и для контекста `WebApplicationContext` каждого сервлета).

Дескриптор `<servlet-mapping>` указывает веб-контейнеру (например, Tomcat), что все URL в шаблоне `/restful/*` (скажем, `http://localhost:8080/ch12/restful/contact`) будут обрабатываться сервлетом `restful`.

Как показано в листинге 12.28, для сервлета `restful` мы также указываем, что контекст `WebApplicationContext` данного `DispatcherServlet` должен быть загружен из файла `rest-context.xml`. Содержимое этого конфигурационного файла представлено в листинге 12.29.

**Листинг 12.29. Конфигурация WebApplicationContext для веб-служб REST**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-context.xsd
         http://www.springframework.org/schema/mvc
         http://www.springframework.org/schema/mvc/spring-mvc.xsd">

   <mvc:annotation-driven>
      <mvc:message-converters>
         <bean class="org.springframework.http.converter.json.
MappingJackson2HttpMessageConverter"/>
         <bean class="org.springframework.http.converter.xml.
MarshallingHttpMessageConverter">
            <property name="marshaller" ref="castorMarshaller"/>
            <property name="unmarshaller" ref="castorMarshaller"/>
         </bean>
      </mvc:message-converters>
   </mvc:annotation-driven>

   <context:component-scan base-package="com.apress.prospring4.ch12"/>

   <bean id="castorMarshaller"
         class="org.springframework.oxm.castor.CastorMarshaller">
      <property name="mappingLocation"
                value="classpath:META-INF/spring/oxm-mapping.xml"/>
   </bean>
</beans>
```

Ниже отмечены важные моменты, связанные с листингом 12.26.

- Дескриптор `<mvc:annotation-driven>` включает поддержку аннотаций для Spring MVC (т.е. аннотацию `@Controller`), а также регистрирует систему преобразования типов и форматирования Spring. Кроме того, с определением этого дескриптора также включается поддержка проверки достоверности JSR-349.
- Внутри `<mvc:annotation-driven>` дескриптор `<mvc:message-converters>` объявляет экземпляры `HttpMessageConverter`, которые будут использоваться для преобразований в поддерживаемые форматы. Обратите внимание на то, что дескриптор `<mvc:message-converters>` появился в версии Spring 3.1. Поскольку мы планируем поддерживать форматы данных JSON и XML, объявляются два преобразователя. Первый из них, `MappingJackson2HttpMessageConverter`, предлагается библиотекой Jackson JSON (`http://jackson.codehaus.org`). Второй преобразователь, `MarshallingHttpMessageConverter`, предоставляется модулем `spring-oxm` и предназначен для маршализации/демаршализации XML. Внутри `MarshallingHttpMessageConverter` мы должны определить применяемые маршализатор и демаршализатор, которые в данном случае обеспечиваются библиотекой Castor XML.

- Для бина `castorMarshaller` используется Spring-класс `org.springframework.oxm.castor.CastorMarshaller`, который интегрируется с Castor, и мы указываем местоположение файла отображения, требуемого для работы Castor.
- Дескриптор `<context:component-scan>` указывает Spring пакеты для сканирования на предмет классов контроллеров.

Итак, создание службы серверной стороны завершено. В этот момент вы должны построить WAR-файл, содержащий веб-приложение, или же, если вы работаете в IDE-среде вроде STS, то запустить экземпляр сервера Tomcat.

## Использование `curl` для тестирования веб-служб REST

Давайте проведем быстрый тест реализованных нами веб-служб REST. Один из простых способов тестирования предполагает применение `curl` (`http://curl.haxx.se`) — инструмента командной строки для передачи данных в синтаксисе URL. Для использования этого инструмента загрузите его из указанного веб-сайта и распакуйте на жестком диске своего компьютера.

Например, чтобы протестировать извлечение всех контактов, откройте окно командной строки в Windows или окно терминала в Unix/Linux, запустите веб-службу REST и введите команду, показанную в листинге 12.30.

**Листинг 12.30. Команда `curl` для тестирования веб-службы REST и вывод в формате JSON**

```
curl -v -H "Accept: application/json"
  http://localhost:8080/ch12/restful/contact/listdata
...
{
  "contacts": [
    {
      "id": 1,
      "version": 0,
      "firstName": "Chris",
      "lastName": "Schaefer",
      "birthDate": {
        "year": 1981,
        "dayOfMonth": 2,
        "dayOfWeek": 6,
        "era": 1,
        "dayOfYear": 122,
        "weekyear": 1981,
        "weekOfWeekyear": 18,
        "monthOfYear": 5,
        "yearOfEra": 1981,
        "yearOfCentury": 81,
        "centuryOfEra": 19,
        "millisOfSecond": 0,
        "millisOfDay": 72000000,
        "secondOfMinute": 0,
        "secondOfDay": 72000,
        "minuteOfHour": 0,
        "minuteOfDay": 1200,
        "hourOfDay": 20,
        "chronology": {
          "zone": {
```

```
        "fixed": false,
        "uncachedZone": {
          "fixed": false,
          "cachable": true,
          "id": "America\/New_York"
        },
        "id": "America\/New_York"
      }
    },
    "zone": {
      "fixed": false,
      "uncachedZone": {
        "fixed": false,
        "cachable": true,
        "id": "America\/New_York"
      },
      "id": "America\/New_York"
    },
    "millis": 357696000000,
    "afterNow": false,
    "beforeNow": true,
    "equalNow": false
  }
},
{
  "id": 2,
  "version": 0,
  "firstName": "Scott",
  "lastName": "Tiger",
  "birthDate": {
    "year": 1990,
    "dayOfMonth": 1,
    "dayOfWeek": 4,
    "era": 1,
    "dayOfYear": 305,
    "weekyear": 1990,
    "weekOfWeekyear": 44,
    "monthOfYear": 11,
    "yearOfEra": 1990,
    "yearOfCentury": 90,
    "centuryOfEra": 19,
    "millisOfSecond": 0,
    "millisOfDay": 68400000,
    "secondOfMinute": 0,
    "secondOfDay": 68400,
    "minuteOfHour": 0,
    "minuteOfDay": 1140,
    "hourOfDay": 19,
    "chronology": {
      "zone": {
        "fixed": false,
        "uncachedZone": {
          "fixed": false,
          "cachable": true,
          "id": "America\/New_York"
        },
```

```
      "id": "America\/New_York"
    }
  },
  "zone": {
    "fixed": false,
    "uncachedZone": {
      "fixed": false,
      "cachable": true,
      "id": "America\/New_York"
    },
    "id": "America\/New_York"
  },
  "millis": 657504000000,
  "afterNow": false,
  "beforeNow": true,
  "equalNow": false
 }
},
{
 "id": 3,
 "version": 0,
 "firstName": "John",
 "lastName": "Smith",
 "birthDate": {
  "year": 1964,
  "dayOfMonth": 27,
  "dayOfWeek": 4,
  "era": 1,
  "dayOfYear": 58,
  "weekyear": 1964,
  "weekOfWeekyear": 9,
  "monthOfYear": 2,
  "yearOfEra": 1964,
  "yearOfCentury": 64,
  "centuryOfEra": 19,
  "millisOfSecond": 0,
  "millisOfDay": 68400000,
  "secondOfMinute": 0,
  "secondOfDay": 68400,
  "minuteOfHour": 0,
  "minuteOfDay": 1140,
  "hourOfDay": 19,
  "chronology": {
    "zone": {
      "fixed": false,
      "uncachedZone": {
        "fixed": false,
        "cachable": true,
        "id": "America\/New_York"
      },
      "id": "America\/New_York"
    }
  },
  "zone": {
    "fixed": false,
```

```
        "uncachedZone": {
          "fixed": false,
          "cachable": true,
          "id": "America\/New_York"
        },
        "id": "America\/New_York"
      },
      "millis": -184377600000,
      "afterNow": false,
      "beforeNow": true,
      "equalNow": false
    }
  }
]
}
```

Эта команда отправляет HTTP-запрос серверной веб-службе REST; в данном случае будет вызван метод `listData()` класса `ContactController` для извлечения и возврата всех контактов. Опция `-H` объявляет атрибут `Accept` заголовка HTTP, который указывает, что клиент ожидает получения данных в формате JSON. В результате выполнения этой команды будет получен вывод в формате JSON для изначально заполненных контактов. А теперь давайте взглянем на формат XML; команда представлена в листинге 12.31.

**Листинг 12.31. Команда `curl` для тестирования веб-службы REST и вывод в формате XML**

```
curl -v -H "Accept: application/xml"
  http://localhost:8080/ch12/restful/contact/listdata
...
<?xml version="1.0" encoding="UTF-8"?>
<contacts>
   <contact>
      <id>1</id>
      <firstName>Chris</firstName>
      <lastName>Schaefer</lastName>
      <birthDate>1981-05-02</birthDate>
      <version>0</version>
   </contact>
   <contact>
      <id>2</id>
      <firstName>Scott</firstName>
      <lastName>Tiger</lastName>
      <birthDate>1990-11-01</birthDate>
      <version>0</version>
   </contact>
   <contact>
      <id>3</id>
      <firstName>John</firstName>
      <lastName>Smith</lastName>
      <birthDate>1964-02-27</birthDate>
      <version>0</version>
   </contact>
</contacts>
```

Как видите, по сравнению с листингом 12.30 здесь имеется всего лишь одно отличие. Атрибут `Accept` изменен с JSON на XML. Запуск этой команды приводит к получению вывода в формате XML. Это связано с объявлением экземпляров `HttpMessageConverter` в контексте `WebApplicationContext` сервлета REST; инфраструктура Spring MVC будет вызывать соответствующий преобразователь сообщений на основе атрибута `Accept`, указанного в HTTP-заголовке клиента, и записывать результат в HTTP-ответ.

## Использование класса `RestTemplate` для доступа к веб-службам REST

Для Spring-приложений спроектирован класс `RestTemplate`, предназначенный для доступа к веб-службам REST. В этом разделе мы покажем, как использовать этот класс для доступа к службе контакта на сервере.

Прежде всего, давайте взглянем на базовую конфигурацию `ApplicationContext` для Spring-класса `RestTemplate`, которая представлена в листинге 12.32 (`restful-client-app-context.xml`).

**Листинг 12.32. Файл `restful-client-app-context.xml`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="restTemplate"
         class="org.springframework.web.client.RestTemplate">
      <property name="messageConverters">
         <list>
            <bean
class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter">
               <property name="marshaller" ref="castorMarshaller"/>
               <property name="unmarshaller" ref="castorMarshaller"/>
               <property name="supportedMediaTypes">
                  <list>
                     <bean class="org.springframework.http.MediaType">
                        <constructor-arg index="0" value="application"/>
                        <constructor-arg index="1" value="xml"/>
                     </bean>
                  </list>
               </property>
            </bean>
         </list>
      </property>
   </bean>

   <bean id="castorMarshaller"
         class="org.springframework.oxm.castor.CastorMarshaller">
      <property name="mappingLocation"
               value="classpath:META-INF/spring/oxm-mapping.xml"/>
   </bean>
</beans>
```

В коде объявлен бин `restTemplate` с классом `RestTemplate`. Этот класс использует библиотеку Castor для внедрения свойства `messageConverters` с экземпляром `MarshallingHttpMessageConverter`, который является тем же самым экземпляром, что и применяемый на стороне сервера.

Файл отображений будет совместно использоваться на серверной и клиентской сторонах. Кроме того, в бине `restTemplate` внутри анонимного бина класса `MarshallingHttpMessageConverter` внедрено свойство `supportedMediaTypes` с объявлением анонимного бина класса `MediaType`, указывающее единственный поддерживаемый формат XML. В результате клиент всегда ожидает XML в качестве формата возвращаемых данных, и библиотека Castor XML поможет выполнять преобразования между POJO и XML.

Давайте опробуем службу для получения всех контактов. Соответствующая тестовая программа приведена в листинге 12.33.

**Листинг 12.33. Тестирование `RestTemplate` для операции извлечения всех контактов**

```
package com.apress.prospring4.ch12;

import org.springframework.context.support.GenericXmlApplicationContext;
import org.springframework.web.client.RestTemplate;

public class RestfulClientSample {
    private static final String URL_GET_ALL_CONTACTS =
            "http://localhost:8080/ch12/restful/contact/listdata";
    private static final String URL_GET_CONTACT_BY_ID =
            "http://localhost:8080/ch12/restful/contact/{id}";
    private static final String URL_CREATE_CONTACT =
            "http://localhost:8080/ch12/restful/contact/";
    private static final String URL_UPDATE_CONTACT =
            "http://localhost:8080/ch12/restful/contact/{id}";
    private static final String URL_DELETE_CONTACT =
            "http://localhost:8080/ch12/restful/contact/{id}";

    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/restful-client-app-context.xml");
        ctx.refresh();

        Contact contact;
        RestTemplate restTemplate = ctx.getBean("restTemplate", RestTemplate.class);

        System.out.println("Testing retrieve all contacts:");
        Contacts contacts =
            restTemplate.getForObject(URL_GET_ALL_CONTACTS, Contacts.class);
        listContacts(contacts);
    }

    private static void listContacts(Contacts contacts) {
        for (Contact contact: contacts.getContacts()) {
            System.out.println(contact);
        }
        System.out.println("");
    }
}
```

Здесь объявлены URL для доступа к различным операциям, которые будут применяться в последующих примерах. В методе `main()` извлекается экземпляр `RestTemplate` и затем вызывается метод `RestTemplate.getForObject()` (соответствующий HTTP-методу `GET`), которому передается URL и ожидаемый тип возврата — класс `Contacts`, содержащий полный список контактов.

Удостоверьтесь, что сервер приложений функционирует. Запуск этой программы даст следующий вывод:

```
Testing retrieve all contacts:
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday:
1981-05-02T00:00:00.000-04:00
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday:
1990-11-01T00:00:00.000-05:00
Contact - Id: 3, First name: John, Last name: Smith, Birthday:
1964-02-27T00:00:00.000-05:00
```

Как видите, экземпляр `MarshallingHttpMessageConverter`, зарегистрированный внутри `RestTemplate`, автоматически преобразует сообщение в объект POJO.

А теперь попробуем извлечь контакт по идентификатору. Добавьте фрагмент кода из листинга 12.34 в метод `main()` класса `RestfulClientSample`.

**Листинг 12.34. Тестирования `RestTemplate` для операции извлечения контакта по идентификатору**

```
System.out.println("Testing retrieve a contact by id :");
contact = restTemplate.getForObject(URL_GET_CONTACT_BY_ID, Contact.class, 1);
System.out.println(contact);
System.out.println("");
```

В показанном листинге 12.34 используется версия метода `RestTemplate.getForObject()`, которой также передается идентификатор контакта для извлечения в качестве переменной пути внутри URL (переменная пути `{id}` в `URL_GET_CONTACT_BY_ID`). Если URL содержит более одной переменной пути, то для передачи этих переменных можно применять экземпляр `Map<String,Object>` или поддержку переменного количества аргументов. В последнем случае переменные пути должны следовать в порядке их объявления в URL. Запуск программы теперь даст такой вывод (показан не полностью):

```
Testing retrieve a contact by id :
Contact - Id: 1, First name: Chris, Last name: Schaefer, Birthday:
1981-05-02T00:00:00.000-04:00
```

Как видите, извлечен корректный контакт. Наступила очередь обновления. Добавьте фрагмент кода из листинга 12.35 в метод `main()` класса `RestfulClientSample`.

**Листинг 12.35. Тестирование `RestTemplate` для операции обновления контакта**

```
contact = restTemplate.getForObject(URL_UPDATE_CONTACT, Contact.class, 1);
contact.setFirstName("John Doe");
System.out.println("Testing update contact by id :");
restTemplate.put(URL_UPDATE_CONTACT, contact, 1);
System.out.println("Contact update successfully: " + contact);
System.out.println("");
```

Сначала мы извлекаем контакт, подлежащий обновлению. После обновления объекта контакта мы используем метод `RestTemplate.put()`, соответствующий HTTP-методу `PUT`, и передаем ему URL обновления, модифицированный объект контакта и идентификатор обновляемого контакта. Запуск программы дает следующий вывод (показан не полностью):

```
Testing update contact by id :
Contact update successfully: Contact - Id: 1, First name: John Doe,
Last name: Schaefer, Birthday: 1981-05-02T00:00:00.000-04:00
```

Далее мы протестируем операцию удаления. Добавьте фрагмент кода из листинга 12.36 в метод `main()` класса `RestfulClientSample`.

**Листинг 12.36. Тестирование `RestTemplate` для операции удаления контакта**

```
restTemplate.delete(URL_DELETE_CONTACT, 1);
System.out.println("Testing delete contact by id :");
contacts = restTemplate.getForObject(URL_GET_ALL_CONTACTS, Contacts.class);
listContacts(contacts);
```

Здесь вызывается метод `RestTemplate.delete()`, соответствующий HTTP-методу `DELETE`, которому передается URL и идентификатор. После этого извлекаются все контакты и еще раз отображаются для подтверждения удаления. Запуск программы дает следующий вывод (показан не полностью):

```
Testing delete contact by id :
Contact - Id: 2, First name: Scott, Last name: Tiger, Birthday:
1990-11-01T00:00:00.000-05:00
Contact - Id: 3, First name: John, Last name: Smith, Birthday:
1964-02-27T00:00:00.000-05:00
```

Легко заметить, что контакт с идентификатором 1 был успешно удален. Наконец, давайте проверим операцию вставки. Добавьте фрагмент кода из листинга 12.37 в метод `main()` класса `RestfulClientSample`.

**Листинг 12.37. Тестирование `RestTemplate` для операции вставки контакта**

```
System.out.println("Testing create contact :");
Contact contactNew = new Contact();
contactNew.setFirstName("James");
contactNew.setLastName("Gosling");
contactNew.setBirthDate(new DateTime());
contactNew =
 restTemplate.postForObject(URL_CREATE_CONTACT, contactNew, Contact.class);
System.out.println("Contact created successfully: " + contactNew);
```

Первым делом, создается новый экземпляр объекта `Contact`. Затем вызывается метод `RestTemplate.postForObject()`, соответствующий HTTP-методу `POST`, которому передается URL, созданный экземпляр `Contact` и тип класса. Выполнение тестовой программы даст в результате такой вывод:

```
Testing create contact :
Contact created successfully: Contact - Id: null, First name: James,
Last name: Gosling, Birthday: 2014-03-13T00:00:00.000-04:00
```

Новый контакт был создан на сервере и возвращен клиенту.

## Защита веб-служб REST с помощью Spring Security

Любая удаленная служба требует защиты от неавторизованного доступа, а также извлечения бизнес-информации или взаимодействия с ней. Веб-службы REST не являются исключением. В этом разделе мы покажем, как использовать проект Spring Security для защиты веб-служб REST на сервере. В рассматриваемом примере мы используем версию Spring Security 3.2 (на момент написания книги последним выпуском был 3.2.1.RELEASE), которая предлагает удобную поддержку веб-служб REST.

Применение Spring Security для защиты веб-служб REST производится в три этапа. На первом этапе в дескрипторе развертывания веб-приложения (`web.xml`) понадобится объявить фильтр; соответствующий фрагмент кода приведен в листинге 12.38.

**Листинг 12.38. Объявление фильтра Spring Security в `web.xml`**

```
<filter>
   <filter-name>springSecurityFilterChain</filter-name>
   <filter-class>org.springframework.web.filter.DelegatingFilterProxy
   </filter-class>
</filter>

<filter-mapping>
   <filter-name>springSecurityFilterChain</filter-name>
   <url-pattern>/restful/*</url-pattern>
</filter-mapping>
```

Здесь объявлен фильтр, который позволяет Spring Security перехватывать HTTP-запрос для выполнения проверки, связанной с аутентификацией и авторизацией. Поскольку мы хотим защитить только веб-службы REST, фильтр применен только к шаблону URL вида `/restful/*` (в дескрипторе `<filter-mapping>`).

На втором этапе нужно создать конфигурацию Spring Security, которая будет располагаться в корневом контексте `WebApplicationContext`. Содержимое файла `web-security.xml` показано в листинге 12.39.

**Листинг 12.39. Конфигурация Spring Security**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
   xmlns:beans="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/security
      http://www.springframework.org/schema/security/spring-security.xsd">

   <http pattern="/restful/**" create-session="stateless">
      <intercept-url pattern='/**' access='ROLE_REMOTE' />
      <http-basic />
   </http>
```

```
    <authentication-manager>
        <authentication-provider>
            <user-service>
              <user name="remote" password="remote" authorities="ROLE_REMOTE" />
            </user-service>
        </authentication-provider>
    </authentication-manager>
</beans:beans>
```

Мы объявляем пространство имен `security` (обратите внимание на строку `xmlns="http://www.springframework.org/schema/security"`) и затем применяем его в качестве стандартного пространства имен для конфигурационного файла. В дескрипторе `<http>` мы указываем, что должны защищаться ресурсы под URL вида `/restful/**`. Атрибут `create-session`, появившийся в Spring Security 3.1.0, позволяет сконфигурировать, будет ли при аутентификации создаваться HTTP-сеанс. Так как используемые веб-службы REST не поддерживают состояние, мы устанавливаем значение этого атрибута в `stateless`, сообщая Spring Security о том, что создавать HTTP-сеанс для всех запросов REST не нужно. В результате можно улучшить производительность веб-служб REST.

В дескрипторе `<intercept-url>` мы указываем, что получать доступ к веб-службам REST могут только пользователи с назначенной ролью `ROLE_REMOTE`. Дескриптор `<http-basic/>` определяет, что для веб-служб REST поддерживается только базовая аутентификация HTTP.

Дескриптор `<authentication-manager>` задает информацию, связанную с аутентификацией. Здесь мы определяем простой поставщик аутентификации с жестко закодированными именем пользователя и паролем (оба установлены в `remote`) и назначенной ролью `ROLE_REMOTE`. В корпоративной среде аутентификация, скорее всего, будет производиться либо через базу данных, либо за счет поиска LDAP. На третьем этапе мы импортируем конфигурацию Spring Security в корневой контекст `WebApplicationContext`. Добавьте в файл `root-context.xml` следующую строку:

```
<import resource="web-security.xml" />
```

На этом настройка защиты завершена. Запустив приложение с классом `RestfulClientSample` еще раз, вы увидите следующий вывод (показан не полностью):

```
Exception in thread "main" org.springframework.web.cient.
HttpClientErrorException: 401 Unauthorized
```

```
В потоке main возникло исключение org.springframework.web.cient.
HttpClientErrorException: 401 Неавторизованный
```

Вы получите код состояния HTTP 401, который означает отсутствие авторизации для доступа к службе. Теперь давайте сконфигурируем `RestTemplate` клиента для предоставления учетных данных серверу.

Для начала понадобится модифицировать конфигурацию клиентского приложения REST. В листинге 12.40 показано измененное содержимое конфигурационного файла `restful-client-app-context.xml`.

**Листинг 12.40. Модифицированный файл `restful-client-app-context.xml`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="restTemplate"
      class="org.springframework.web.client.RestTemplate">
      <constructor-arg ref="httpRequestFactory"/>
      <property name="messageConverters">
        <!-- Настройки остались такими же, как ранее, поэтому они не показаны -->
      </property>
   </bean>

   <bean id="castorMarshaller"
      class="org.springframework.oxm.castor.CastorMarshaller">
      <property name="mappingLocation"
                value="classpath:META-INF/spring/oxm-mapping.xml"/>
   </bean>

   <bean id="httpRequestFactory"
class="org.springframework.http.client.HttpComponentsClientHttpRequestFactory">
      <constructor-arg>
         <bean class="org.apache.http.impl.client.DefaultHttpClient">
            <property name="credentialsProvider">
               <bean
class="com.apress.prospring4.ch12.CustomCredentialsProvider">
                  <property name="credentials">
                     <bean
class="org.apache.http.auth.UsernamePasswordCredentials">
                        <constructor-arg name="userName" value="remote"/>
                        <constructor-arg name="password" value="remote"/>
                     </bean>
                  </property>
               </bean>
            </property>
         </bean>
      </constructor-arg>
   </bean>
</beans>
```

В бин `restTemplate` внедрен аргумент конструктора со ссылкой на бин `httpRequestFactory`. Для бина `httpRequestFactory` применяется класс `HttpComponentsClientHttpRequestFactory`, который представляет собой поддержку Spring для библиотеки HttpClient проекта Apache HttpComponents; эта библиотека нужна для конструирования экземпляра `DefaultHttpClient`, хранящего учетные данные для клиента. В целях поддержки внедрения учетных данных мы реализуем простой класс `CustomCredentialsProvider`, приведенный в листинге 12.41.

**Листинг 12.41. Класс `CustomCredentialsProvider`**

```
package com.apress.prospring4.ch12;

import org.apache.http.auth.AuthScope;
import org.apache.http.auth.Credentials;
import org.apache.http.impl.client.BasicCredentialsProvider;

public class CustomCredentialsProvider extends BasicCredentialsProvider {
   public void setCredentials(Credentials credentials) {
      this.setCredentials(AuthScope.ANY, credentials);
   }
}
```

Класс `CustomCredentialsProvider` расширяет класс `BasicCredentials Provider` из библиотеки HttpComponents, и в нем реализован новый метод установки для поддержки внедрения учетных данных. Возвратившись к листингу 12.40, вы увидите, что учетные данные внедрены в этот класс с использованием экземпляра класса `UsernamePasswordCredentials`. Класс `UsernamePasswordCredentials` сконструирован с именем пользователя и паролем, установленными в `remote`. За счет внедрения сконструированного `httpRequestFactory` в `RestTemplate` все запросы REST, отправленные с применением этого шаблона, будут содержать учетные данные. Если теперь снова запустить приложение с классом `RestfulClientSample`, можно заметить, что службы вызываются обычным образом.

# Использование AMQP в Spring

Удаленную обработку можно также реализовать с использованием коммуникаций в стиле RPC (remote procedure call — удаленный вызов процедуры) по протоколу AMQP (Advanced Message Queuing Protocol — расширенный протокол организации очередей сообщений) в качестве транспорта. Подобно JMS, для обмена сообщениями протокол AMQP также применяет брокер сообщений. В этом примере мы используем для сервера AMQP продукт RabbitMQ (`www.rabbitmq.org`). Платформа Spring не предоставляет возможности удаленной обработки в своем ядре. Взамен поддерживается родственный проект под названием Spring AMQP (`http://projects.spring.io/spring-amqp`), который мы применяем как лежащий в основе API-интерфейс для коммуникаций. Проект Spring AMQP предоставляет базовую абстракцию для AMQP и реализацию, предназначенную для взаимодействия с помощью RabbitMQ. В этой главе мы не собираемся раскрывать абсолютно все возможности AMQP или Spring AMQP, а рассмотрим только функциональность удаленной обработки посредством коммуникаций RPC.

Прежде всего, понадобится получить продукт RabbitMQ из `www.rabbitmq.com/download.html` и запустить сервер. Никаких изменений в конфигурацию RabbitMQ вносить не придется, т.к. изначально все работает должным образом.

Когда RabbitMQ уже выполняется, необходимо создать интерфейс службы. В рассматриваемом примере мы создаем простую метеослужбу (Weather Service), которая возвращает прогноз погоды для штата с указанным кодом. Давайте начнем с создания интерфейса `WeatherService` (листинг 12.42).

**Листинг 12.42. Интерфейс WeatherService**

```
package com.apress.prospring4.ch12;

public interface WeatherService {
   String getForecast(String stateCode);
}
```

Далее мы создаем реализацию интерфейса WeatherService, которая просто в ответ на предоставленный код штата выдает прогноз погоды или сообщение о недоступности в случае, если прогноз отсутствует (листинг 12.43).

**Листинг 12.43. Реализация интерфейса WeatherService**

```
package com.apress.prospring4.ch12;

public class WeatherServiceImpl implements WeatherService {
   @Override
   public String getForecast(String stateCode) {
      if ("FL".equals(stateCode)) {
         return "Hot";  // Жарко
      } else if ("MA".equals(stateCode)) {
         return "Cold"; // Холодно
      }

      return "Not available at this time";
           // В данный момент прогноз недоступен
   }
}
```

Имея код метеослужбы, давайте построим конфигурационный файл (amqp-rpc-app-context.xml), который будет настраивать подключение AMQP и открывать доступ к реализации WeatherService, как показано в листинге 12.44.

**Листинг 12.44. Конфигурация WeatherService**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:rabbit="http://www.springframework.org/schema/rabbit"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/rabbit
       http://www.springframework.org/schema/rabbit/spring-rabbit.xsd">

   <rabbit:connection-factory id="connectionFactory" host="localhost" />

   <rabbit:template id="amqpTemplate"
                connection-factory="connectionFactory"
                reply-timeout="2000" routing-key="forecasts"
                exchange="weather" />

   <rabbit:admin connection-factory="connectionFactory" />

   <rabbit:queue name="forecasts" />
```

```
    <rabbit:direct-exchange name="weather">
        <rabbit:bindings>
            <rabbit:binding queue="forecasts" key="forecasts" />
        </rabbit:bindings>
    </rabbit:direct-exchange>

    <bean id="weatherServiceProxy"
      class="org.springframework.amqp.remoting.client.AmqpProxyFactoryBean">
        <property name="amqpTemplate" ref="amqpTemplate" />
        <property name="serviceInterface"
                  value="com.apress.prospring4.ch12.WeatherService" />
    </bean>

    <rabbit:listener-container connection-factory="connectionFactory">
      <rabbit:listener ref="weatherServiceExporter" queue-names="forecasts" />
    </rabbit:listener-container>

    <bean id="weatherServiceExporter"
      class="org.springframework.amqp.remoting.service.AmqpInvokerServiceExporter">
        <property name="amqpTemplate" ref="amqpTemplate" />
        <property name="serviceInterface"
                  value="com.apress.prospring4.ch12.WeatherService" />
        <property name="service">
            <bean class="com.apress.prospring4.ch12.WeatherServiceImpl"/>
        </property>
    </bean>
</beans>
```

Мы конфигурируем подключение RabbitMQ наряду с информацией обмена и очереди. Затем мы создаем бин с использованием класса `AmqpProxyFactoryBean`, который наш клиент применяет в качестве прокси при выполнении запроса RPC. Для запроса мы используем класс `AmqpInvokerServiceExporter`, который связывается с контейнером прослушивателя. Контейнер прослушивателя отвечает за перехват сообщений AMQP и передачу их нашей метеослужбе. Вы можете заметить, что эта конфигурация похожа на конфигурацию JMS в плане подключений, очередей, контейнеров прослушивателей и т.д. Несмотря на подобие в конфигурации, JMS и AMQP являются совершенно разными транспортами, и рекомендуется посетить веб-сайт AMQP (`www.amqp.org`), чтобы ознакомиться с подробными сведениями о данном протоколе. При наличии конфигурации осталось только создать пример класса для выполнения вызовов RPC (листинг 12.45).

**Листинг 12.45. Класс `AmqpRpcSample`**

```
package com.apress.prospring4.ch12;

import org.springframework.context.support.GenericXmlApplicationContext;

public class AmqpRpcSample {
    public static void main(String[] args) {
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
        ctx.load("classpath:META-INF/spring/amqp-rpc-app-context.xml");
        ctx.refresh();

        WeatherService weatherService = ctx.getBean(WeatherService.class);
```

```
        System.out.println("Forecast for FL: "
            + weatherService.getForecast("FL"));
        System.out.println("Forecast for MA: "
            + weatherService.getForecast("MA"));
        System.out.println("Forecast for CA: "
            + weatherService.getForecast("CA"));

        ctx.close();
    }
}
```

Запуск примера должен привести к получению следующего вывода:

```
Forecast for FL: Hot
Forecast for MA: Cold
Forecast for CA: Not available at this time
```

# Резюме

В этой главе были раскрыты наиболее часто используемые технологии удаленной обработки в Spring-приложениях.

Если взаимодействующие приложения основаны на Spring, то удобным вариантом будет применение HTTP-активатора Spring. Если требуется асинхронный режим или слабая связность, то общепринятым выбором будет JMS. В главе также было показано, как использовать веб-службы REST в Spring для открытия служб или доступа к ним с помощью класса `RestTemplate`. Наконец, мы обсудили применение проекта Spring AMQP для удаленной обработки в стиле RPC посредством RabbitMQ.

В следующей главе мы рассмотрим использование Spring для тестирования приложений.

# ГЛАВА 13

# Тестирование в Spring

При разработке корпоративных приложений тестирование является важным способом обеспечения того, что завершенное приложение выполняется ожидаемым образом и удовлетворяет всем видам требований (архитектурным, пользовательским и т.д.). Каждый раз, когда вносится изменение, вы должны удостовериться, что это изменение не повлияло на существующую логику. Поддержка среды построения и тестирования критически важна для обеспечения высокого качества приложений. Воспроизводимые тесты с большой степенью покрытия кода позволяют развертывать новые приложения и изменения к ним с высоким уровнем доверия.

В среде разработки корпоративных приложений существует много видов тестирования, нацеленных на каждый уровень приложения, и каждый вид тестирования обладает собственными характеристиками и требованиями. В этой главе мы обсудим базовые концепции, связанные с тестированием различных уровней приложения, особенно при тестировании Spring-приложений. Мы также представим способы, которыми платформа Spring упрощает разработчикам реализацию тестовых сценариев. В частности, в главе рассматриваются следующие темы.

- **Корпоративная инфраструктура тестирования.** Мы кратко опишем корпоративную инфраструктуру тестирования. Мы обсудим различные виды тестирования и их назначение. Внимание будет сосредоточено на модульном тестировании, ориентированном на разные уровни приложения.
- **Модульное тестирование логики.** Самый мелкий модульный тест предполагает проверку только логики методов внутри какого-то класса, а для всех остальных зависимостей "имитируется" корректное поведение. В этой главе мы обсудим реализацию модульного тестирования логики для классов контроллеров Spring MVC с применением Java-библиотеки имитации для имитирования корректного поведения зависимостей классов.
- **Модульное тестирование взаимодействия.** В корпоративной инфраструктуре тестирования проверка взаимодействия осуществляется в отношении группы классов внутри различных уровней приложения для конкретной порции бизнес-логики. Обычно в среде тестирования взаимодействия уровень обслуживания должен проверяться вместе с уровнем постоянства, т.е. с доступной базой данных серверной части. Однако с развитием архитектуры приложения и обретения зрелости облегченных баз данных, расположенных в памяти, общей практикой теперь становится "модульное тестирование" уровня обслуживания

с уровнем постоянства и базой данных серверной части как единого целого. Например, в этой главе мы будем использовать JPA 2 с Hibernate и Spring Data JPA в качестве поставщика постоянства и H2 в качестве базы данных. В такой архитектуре менее важно "имитировать" Hibernate и Spring Data JPA при тестировании уровня обслуживания. В результате мы здесь будем обсуждать тестирование уровня обслуживания вместе с уровнем постоянства и базой данных H2 в памяти. Такой вид тестирования в общем случае называется модульным тестированием взаимодействия, которое находится посредине между модульным тестированием и полноценным тестированием взаимодействия.

- **Модульное тестирование интерфейсной части.** Даже если мы тестируем каждый уровень приложения, после его развертывания мы по-прежнему должны удостовериться, что все приложение функционирует ожидаемым образом. В частности, в случае веб-приложения, после его развертывания в среде непрерывной сборки мы должны прогнать тестирование интерфейсной части, чтобы проверить, корректно ли работает пользовательский интерфейс. Например, для приложения контактов нужно удостовериться, что каждый шаг в рамках обычной функциональности работает правильно, а также протестировать исключительные случаи (скажем, как приложение функционирует, когда введенная пользователем информация не проходит фазу проверки достоверности). В этой главе мы кратко обсудим применение инфраструктуры тестирования интерфейсной части.

# Введение в корпоративную инфраструктуру тестирования

Под *корпоративной инфраструктурой тестирования* понимаются действия по тестированию в полном жизненном цикле приложения. На различных фазах будут выполняться разные действия тестирования, цель которых заключается в проверке, работает ли функциональность приложения ожидаемым образом в соответствии с определенными бизнес-правилами и техническими требованиями.

На каждой фазе будут прогоняться разные тестовые сценарии. Некоторые из них будут автоматизированы, тогда как другие — выполняться вручную. В каждом сценарии полученные результаты будут проверяться соответствующим персоналом (например, бизнес-аналитиками, пользователями приложения и т.д.).

В табл. 13.1 описаны характеристики и цели каждого типа тестирования, а также общие инструменты и библиотеки, используемые для реализации тестовых сценариев.

В этой главе мы сосредоточим внимание на реализации трех видов модульного тестирования (модульное тестирование логики, модульное тестирование взаимодействия и модульное тестирование интерфейсной части) и покажем, как инфраструктура `TestContext` в Spring и другие поддерживающие инструменты помогают в разработке тестовых сценариев.

Вместо представления детальных сведений и списка классов, которые предлагаются в Spring Framework для области тестирования, мы обсудим применение наиболее часто используемых шаблонов, а также поддерживающих интерфейсов и классов внутри инфраструктуры `TestContext`, реализуя примеры тестовых сценариев в этой главе.

**Таблица 13.1. Описание корпоративной инфраструктуры тестирования**

| Категория тестирования | Описание | Общие инструменты |
|---|---|---|
| Модульное тестирование логики | Модульное тестирование логики применяется к одиночному объекту, не принимая во внимание его роль в окружающей системе | Модульное тестирование: JUnit, TestNG<br>Объекты-имитации: Mockito, EasyMock |
| Модульное тестирование взаимодействия | Модульное тестирование взаимодействия нацелено на проверку взаимодействия между компонентами в среде, "близкой к реальной". Такие тесты будут осуществлять взаимодействия с контейнером (встроенной базой данных, веб-контейнером и т.д.) | Встроенная база данных: H2<br>Тестирование баз данных: DbUnit<br>Веб-контейнер внутри памяти: Jetty |
| Модульное тестирование интерфейсной части | Модульное тестирование интерфейсной части ориентировано на проверку пользовательского интерфейса. Его целью является обеспечение того, что каждый пользовательский интерфейс реагирует на действия пользователей и генерирует ожидаемый вывод | Selenium |
| Непрерывная сборка и тестирование качества | Кодовая база приложения должна собираться на регулярной основе, чтобы обеспечивать соответствие качества кода существующему стандарту (например, наличие комментариев, отсутствие пустых блоков перехвата исключений и т.п.). Кроме того, степень покрытия тестами должна быть как можно более высокой, чтобы гарантировать проверку всех строк разработанного кода | Качество кода: PMD, Checkstyle, FindBugs, Sonar<br>Покрытие тестами: Cobertura, EclEmma<br>Инструменты сборки: Gradle, Maven<br>Непрерывная сборка: Hudson, Jenkins |
| Тестирование взаимодействия в системе | Тестирование взаимодействия в системе проверяет правильность взаимодействия между всеми программами в новой системе и между новой системой и всеми внешними интерфейсами. Тестирование взаимодействия должно также доказать, что новая система работает в соответствии с функциональными спецификациями и эффективно функционирует в операционной среде, не оказывая отрицательного влияния на другие системы | IBM Rational Systems Tester |

*Окончание табл. 13.1*

| Категория тестирования | Описание | Общие инструменты |
|---|---|---|
| Тестирование функциональности | Тестирование функциональности проверяет сценарии использования и бизнес-правила. Цель таких тестов — удостовериться в том, что ввод правильно принимается, а вывод правильно генерируется, где *правильно* означает соответствие и спецификациям сценариев использования, и бизнес-правилам. Это тестирование "черного ящика", осуществляемое путем взаимодействия с приложением через графический пользовательский интерфейс и последующего анализа результатов | IBM Rational Functional Tester, HP Unified Functional Testing |
| Тестирование качества системы | Тестирование качества системы нацелено на проверку того, что разработанное приложение удовлетворяет ряду нефункциональных требований. По большей части при этом выполняется тестирование производительности приложения, чтобы удостовериться в удовлетворении требований относительно параллельно работающих пользователей в системе и рабочей нагрузки. Другие нефункциональные требования включают безопасность, высокую доступность и т.п. | Apache JMeter, HP LoadRunner |
| Тестирование приемлемости использования | Тестирование приемлемости использования эмулирует действительные рабочие условия новой системы, в том числе руководства пользователя и процедуры работы в системе. Широкое привлечение пользователей на этом этапе тестирования предоставит им неоценимый опыт работы ней. Это также позволит программистам и проектировщикам взглянуть на эффективность новых программ. Такое совместное участие способствует более быстрому переходу пользователей и технического персонала на новую систему | IBM Rational TestManager, HP Quality Center |

# Использование аннотаций тестирования Spring

Прежде чем переходить к тестам логики и взаимодействия, полезно отметить, что в дополнение к стандартным аннотациям (вроде `@Autowired` и `@Resource`) платформа Spring предлагает аннотации, специфичные для тестирования. Эти аннотации могут применяться в тестах логики и взаимодействия, предоставляя раз-

нообразную функциональность, такую как упрощенная загрузка файла контекста, профили, координация выполнения тестов и многое другое. В табл. 13.2 приведена сводка по аннотациям и случаям их использования.

**Таблица 13.2. Аннотации тестирования Spring**

| Аннотация | Описание |
|---|---|
| `@ContextConfiguration` | Аннотация уровня классов, которая позволяет определить способ загрузки и конфигурирования `ApplicationContext` для тестов взаимодействия |
| `@WebAppConfiguration` | Аннотация уровня классов, применяемая для указания того, что загружаемый `ApplicationContext` должен быть `WebApplicationContext` |
| `@ContextHierarchy` | Аннотация уровня классов, предоставляющая возможность определения иерархии объектов `ApplicationContext` для заданного теста. Эта аннотация содержит одну или большее число аннотаций `@ContextConfiguration`, представляющих конфигурационные файлы |
| `@ActiveProfiles` | Аннотация уровня классов, которая указывает, какой профиль бина должен быть активным |
| `@DirtiesContext` | Аннотация уровней классов и методов, используемая для указания на то, что во время выполнения теста контекст был каким-то образом модифицирован или поврежден, поэтому он должен быть закрыт и построен повторно для последующих тестов |
| `@TestExecutionListeners` | Аннотация уровня классов, предназначенная для конфигурирования экземпляров `TestExecutionListener`, которые должны быть зарегистрированы с помощью `TestContextManager` |
| `@TransactionConfiguration` | Аннотация уровня классов, применяемая для указания конфигурации транзакции, такой как параметры отката и диспетчер транзакций (если только желаемый диспетчер транзакций не имеет имя бина `transactionManager`) |
| `@Rollback` | Аннотация уровней классов и методов, которая используется для указания, должен ли быть произведен откат транзакции для аннотированного тестового метода. Аннотация уровня классов применяется для стандартных параметров тестирования класса |
| `@BeforeTransaction` | Аннотация уровня методов, указывающая на то, что аннотированный ею метод должен быть вызван перед началом транзакции для тестовых методов, которые помечены с помощью аннотации `@Transactional` |
| `@AfterTransaction` | Аннотация уровня методов, указывающая на то, что аннотированный ею метод должен быть вызван после завершения транзакции для тестовых методов, которые помечены с помощью аннотации `@Transactional` |

*Окончание табл. 13.2*

| Аннотация | Описание |
|---|---|
| `@IfProfileValue` | Аннотация уровней классов и методов, используемая для указания на то, что тестовый метод должен быть разрешен посредством специфичного набора условий среды |
| `@ProfileValueSource-Configuration` | Аннотация уровня классов, которая применяется для указания объекта `ProfileValueSource`, используемого аннотацией `@IfProfileValue`. Если эта аннотация не объявлена в тесте, по умолчанию применяется `SystemProfileValueSource` |
| `@Timed` | Аннотация уровня методов, используемая для указания на то, что тест должен быть завершен в пределах заданного периода времени |
| `@Repeat` | Аннотация уровня методов, которая применяется для указания на то, что аннотированный ею тестовый метод должен быть повторен заданное количество раз |

# Реализация модульных тестов логики

Как обсуждалось ранее, модульный тест логики является самым мелким уровнем тестирования. Его цель заключается в проверке поведения отдельного класса, тогда как для всех зависимостей этих классов ожидаемое поведение имитируется. В этом разделе мы продемонстрируем применение модульного теста логики, реализовав тестовые сценарии для класса `ContactController`, с имитацией ожидаемого поведения для уровня обслуживания. Для имитации ожидаемого поведения уровня обслуживания мы будем использовать популярную инфраструктуру имитации под названием Mockito (`http://code.google.com/p/mockito`).

## Добавление обязательных зависимостей

Первым делом к проекту необходимо добавить зависимости, описанные в табл. 13.3. Мы также будем полагаться на такие классы и интерфейсы, созданные в предшествующих главах, как `Contact`, `ContactService` и т.д.

**Таблица 13.3. Зависимости Maven для модульного тестирования логики**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.mockito` | `mockito-core` | 1.9.5 | Библиотека ядра инфраструктуры имитации Mockito |
| `org.springframework` | `spring-context` | 4.0.2.RELEASE | Модуль контекста Spring |
| `org.springframework` | `spring-test` | 4.0.2.RELEASE | Модуль тестирования Spring |
| `org.springframework` | `spring-web` | 4.0.2.RELEASE | Веб-модуль Spring |
| `junit` | `junit` | 4.11 | Инфраструктура тестирования JUnit |

*Окончание табл. 13.3*

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| org.slf4j | slf4j-log4j12 | 1.7.6 | Адаптер log4j из библиотеки ведения журналов SLF4J |
| joda-time | joda-time | 2.3 | Библиотека для работы с датой и временем JodaTime |
| org.jadira.usertype | usertype.core | 3.0.0.GA | Пользовательские типы данных даты и времени для Hibernate/JodaTime |

## Модульное тестирование контроллеров Spring MVC

На уровне презентаций классы контроллеров обеспечивают интеграцию пользовательского интерфейса и уровня обслуживания.

Методы в классах контроллеров будут отображаться на HTTP-запросы. Внутри такого метода запрос обрабатывается, привязывается к объектам модели и взаимодействует с уровнем обслуживания (внедренным в классы контроллеров с помощью DI из Spring) для обработки данных. По завершении в зависимости от результата класс контроллера обновляет модель и состояние представления (например, пользовательские сообщения, объекты для служб REST и т.п.) и возвращает логическое представление (или модель вместе с представлением) инфраструктуре Spring MVC для распознавания конкретного представления с целью его отображения пользователю.

Основная цель проведения модульного тестирования для классов контроллеров заключается в проверке того, что методы контроллеров правильно обновляют модель и состояния представлений и возвращают корректные представления. Поскольку мы хотим протестировать только поведение классов контроллеров, то должны имитировать уровень обслуживания с корректным поведением.

В случае класса `ContactController` мы собираемся разработать тестовые сценарии для методов `listData()` и `create()`. В последующих разделах мы обсудим необходимые шаги.

### *Тестирование метода `listData()`*

Давайте создадим первый тестовый сценарий для метода `ContactController.listData()`. В этом тестовом сценарии мы хотим удостовериться, что когда метод вызван, после извлечения списка контактов из уровня обслуживания информация правильно сохраняется в модели, а также возвращаются корректные объекты. Тестовый сценарий показан в листинге 13.1.

**Листинг 13.1. Код для тестирования метода `listData()`**

```
package com.apress.prospring4.ch13;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
import java.util.ArrayList;
```

```
import java.util.List;
import org.junit.Before;
import org.junit.Test;
import org.mockito.invocation.InvocationOnMock;
import org.mockito.stubbing.Answer;
import org.springframework.test.util.ReflectionTestUtils;
import org.springframework.ui.ExtendedModelMap;

public class ContactControllerTest {
   private final List<Contact> contacts = new ArrayList<Contact>();

   @Before
   public void initContacts() {
      Contact contact = new Contact();
      contact.setId(11);
      contact.setFirstName("Chris");
      contact.setLastName("Schaefer");
      contacts.add(contact);
   }
   @Test
   public void testList() throws Exception {
      ContactService contactService = mock(ContactService.class);
      when(contactService.findAll()).thenReturn(contacts);

      ContactController contactController = new ContactController();

      ReflectionTestUtils.setField(contactController,
            "contactService", contactService);

      ExtendedModelMap uiModel = new ExtendedModelMap();
      uiModel.addAttribute("contacts", contactController.listData());

      Contacts modelContacts = (Contacts) uiModel.get("contacts");

      assertEquals(1, modelContacts.getContacts().size());
   }
}
```

Во-первых, в тестовом сценарии вызывается метод `initContacts()`, к которому применена аннотация `@Before`, указывающая JUnit, что этот метод должен выполняться перед запуском каждого тестового сценария (в случае, если определенную логику нужно выполнять перед самим тестовым классом, используйте аннотацию `@BeforeClass`). В методе `initContacts()` список контактов инициализируется жестко закодированными значениями.

Во-вторых, метод `testList()` применяется посредством аннотации `@Test`, которая указывает JUnit на то, что это тестовый сценарий, который должен быть запущен. Внутри тестового сценария закрытая переменная `contactService` (типа `ContactService`) имитируется с использованием метода `Mockito.mock()` из библиотеки Mockito (обратите внимание на оператор `import static`). Инфраструктура Mockito также предоставляет метод `when()` для имитации метода `ContactService.findAll()`, который будет применяться классом `ContactController`.

В-третьих, создается экземпляр `ContactController`, и его переменная `contactService`, внедряемая Spring в нормальных ситуациях, устанавливается в имитированный экземпляр с использованием метода `setField()` класса `ReflectionTestUtils`, предоставляемого Spring. Класс `ReflectionTestUtils`

предлагает набор служебных методов, основанных на рефлексии, которые предназначены для применения в сценариях модульного тестирования и тестирования взаимодействия. Кроме того, конструируется экземпляр класса `ExtendedModelMap` (реализующего интерфейс `org.springframework.ui.Model`).

В-четвертых, вызывается метод `ContactController.listData()`. После вызова результат проверяется с помощью различных методов утверждений (предоставляемых JUnit), чтобы удостовериться в корректном сохранении списка контактов в модели, используемой представлением.

Теперь тестовый сценарий можно запустить, и он должен выполниться успешно. В этом можно удостовериться с помощью системы построения или IDE-среды.

Давайте перейдем к тестированию метода `create()`.

### *Тестирование метода `create()`*

В листинге 13.2 приведен фрагмент кода, предназначенный для тестирования метода `create()`.

**Листинг 13.2. Тестирование метода `create()`**

```
package com.apress.prospring4.ch13;
...
import org.mockito.stubbing.Answer;
import org.mockito.invocation.InvocationOnMock;
...
public class ContactControllerTests {
    ...
    @Test
    public void testCreate() {
        final Contact newContact = new Contact();
        newContact.setId(999l);
        newContact.setFirstName("Rod");
        newContact.setLastName("Johnson");

        ContactService contactService = mock(ContactService.class);
        when(contactService.save(newContact)).
              thenAnswer(new Answer<Contact>() {
            public Contact answer(InvocationOnMock invocation)
                    throws Throwable {
                contacts.add(newContact);
                return newContact;
            }
        });

        ContactController contactController = new ContactController();
        ReflectionTestUtils.setField(contactController, "contactService",
                contactService);

        Contact contact = contactController.create(newContact);
        assertEquals(Long.valueOf(999l), contact.getId());
        assertEquals("Rod", contact.getFirstName());
        assertEquals("Johnson", contact.getLastName());

        assertEquals(2, contacts.size());
    }
}
```

Метод `ContactService.save()` имитируется для эмуляции добавления нового объекта `Contact` в список контактов. Обратите внимание на использование интерфейса `org.mockito.stubbing.Answer<T>`, который обеспечивает имитацию метода посредством ожидаемой логики и возвращает значение.

Затем вызывается метод `ContactController.create()`, после чего с помощью операций утверждений проверяется результат. Запустите тестовый сценарий снова и просмотрите результаты его выполнения.

Для метода `create()` мы должны создать дополнительные тестовые сценарии, чтобы проверить разнообразные ситуации. Например, нужно протестировать случай, когда во время операции сохранения возникают ошибки доступа к данным.

# Реализация тестирования взаимодействия

В этом разделе мы реализуем тестирование взаимодействия для уровня обслуживания. Основной службой в приложении контактов является класс `com.apress.prospring4.ch13.ContactServiceImpl`, который представляет собой JPA-реализацию интерфейса `com.apress.prospring4.ch13.ContactService`.

При выполнении модульного тестирования уровня обслуживания мы будем использовать для хранения модели и тестовой информации базу данных в памяти типа H2 с поставщиками JPA (Hibernate и абстракция репозитория Spring Data JPA). Цель заключается в том, чтобы удостовериться, что класс `ContactServiceImpl` корректно выполняет свои бизнес-функции.

В последующих разделах мы покажем, как протестировать некоторые методы поиска и операцию сохранения в классе `ContactServiceImpl`.

## Добавление обязательных зависимостей

Для реализации тестовых сценариев с базой данных нам нужна библиотека, которая поможет заполнить базу данных желаемой тестовой информацией перед запуском тестового сценария и затем легко выполнить необходимые операции в базе данных. Кроме того, для упрощения подготовки тестовых данных мы будем применять формат Microsoft Excel. Чтобы удовлетворить указанным целям, понадобятся дополнительные библиотеки. Для стороны базы данных распространенной библиотекой, которая помогает реализовать тестирование, связанное с базой данных, является DbUnit (`http://dbunit.sourceforge.net`). Вдобавок будет использоваться библиотека проекта Apache POI (`http://poi.apache.org`), с помощью которой будет производиться разбор тестовых данных, подготовленных в формате Microsoft Excel.

В табл. 13.4 описаны обязательные зависимости.

**Таблица 13.4. Зависимости Maven для тестирования взаимодействия**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.dbunit` | `dbunit` | 2.4.9 | Библиотека DbUnit |
| `org.apache.poi` | `poi` | 3.2-FINAL | Библиотека Apache POI, которая поддерживает чтение и запись файлов в формате Microsoft Office. Версия 3.2-FINAL используется из-за несовместимостей последней версии с DbUnit |

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| org.springframework.data | spring-data-jpa | 1.5.0 | Модуль Spring Data JPA |
| org.springframework | spring-orm | 4.0.2.RELEASE | Модуль Spring ORM |
| com.h2database | h2 | 1.3.172 | База данных H2 |
| javax.validation | validation-api | 1.1.0.Final | API-интерфейс проверки достоверности JSR-349 |
| org.hibernate | hibernate-validator | 5.1.0.Final | Реализация API-интерфейса проверки достоверности JSR-349 в Hibernate |
| javax.el | el-api | 2.2 | API-интерфейс библиотеки EL |

## Конфигурирование профиля для тестирования уровня обслуживания

Средство профилей для определений бинов, появившееся в версии Spring 3.1, очень удобно для реализации тестового сценария с соответствующей конфигурацией тестируемых компонентов. Чтобы упростить тестирование уровня обслуживания, для конфигурации `ApplicationContext` мы также будем использовать средство профилей.

Для приложения контактов нам понадобятся два профиля.

- **Профиль разработки (`dev`).** Профиль с конфигурацией для среды разработки. Например, в среде разработки для базы данных серверной части H2 будет выполняться как сценарий создания, так и сценарий наполнения начальными данными.
- **Профиль тестирования (`test`).** Профиль с конфигурацией для среды тестирования. Например, в среде тестирования для базы данных серверной части H2 будет выполняться только сценарий создания, а наполнение данными будет осуществляться тестовым сценарием.

Давайте сконфигурируем среду профилей для приложения работы с контактами. В этом приложении конфигурация серверной части (т.е. источника данных, JPA, транзакции и т.д.) была определена в XML-файле конфигурации `datasource-tx-jpa.xml`. Мы хотели бы конфигурировать источник данных в этом файле только для профиля `dev`. Чтобы сделать это, бин источника данных понадобится поместить в конфигурацию профиля. В листинге 13.3 приведено содержимое файла `datasource-tx-jpa.xml` с необходимыми изменениями.

**Листинг 13.3. Конфигурирование профиля для источника данных**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:jdbc="http://www.springframework.org/schema/jdbc"
```

```
      xmlns:jpa="http://www.springframework.org/schema/data/jpa"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/jdbc
       http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
       http://www.springframework.org/schema/data/jpa
       http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx.xsd">
   <bean id="transactionManager"
         class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <tx:annotation-driven transaction-manager="transactionManager" />

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
         <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan" value="com.apress.prospring4.ch13"/>
      <property name="jpaProperties">
         <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.H2Dialect
            </prop>
            <prop key="hibernate.max_fetch_depth">3</prop>
            <prop key="hibernate.jdbc.fetch_size">50</prop>
            <prop key="hibernate.jdbc.batch_size">10</prop>
            <prop key="hibernate.show_sql">true</prop>
         </props>
      </property>
   </bean>

   <context:annotation-config/>

   <jpa:repositories base-package="com.apress.prospring4.ch13"
                     entity-manager-factory-ref="emf"
                     transaction-manager-ref="transactionManager"/>

   <beans profile="dev">
      <jdbc:embedded-database id="dataSource" type="H2">
         <jdbc:script location="classpath:META-INF/config/schema.sql"/>
         <jdbc:script location="classpath:META-INF/config/test-data.sql"/>
      </jdbc:embedded-database>
   </beans>
</beans>
```

Как показано в этом коде, бин `dataSource` помещен внутрь дескриптора `<beans>` с атрибутом `profile`, имеющим значение `dev`, которое указывает, что источник данных применим только для системы разработки. Вспомните, что профили

можно активизировать, например, с помощью передачи машине JVM системного параметра `-Dspring.profiles.active=dev`.

## Реализация классов инфраструктуры

Перед реализацией индивидуального тестового сценария мы должны реализовать ряд классов для поддержки наполнения тестовыми данными из файла Excel. Вдобавок, чтобы упростить разработку тестового сценария, мы хотим ввести специальную аннотацию по имени `@DataSets`, которая принимает имя файла Excel в качестве аргумента. Мы разработаем специальный прослушиватель выполнения тестов (средство, поддерживаемое инфраструктурой тестирования Spring) для проверки существования этой аннотации и соответствующей загрузки данных.

В последующих разделах мы обсудим реализацию различных классов инфраструктуры и специального прослушивателя, который загружает данные из файла Excel.

### *Реализация специального прослушивателя выполнения тестов*

Интерфейс `org.springframework.test.context.TestExecutionListener` из модуля `spring-test` определяет API-интерфейс прослушивателя, который может перехватывать события на различных фазах выполнения тестового сценария (например, перед и после тестируемого класса, перед и после тестируемого метода и т.д.). При тестировании уровня обслуживания мы реализуем специальный прослушиватель для вновь введенной аннотации `@DataSets`. Цель заключается в поддержке наполнения тестовыми данными с помощью простой аннотации в тестовом сценарии. К примеру, для тестирования метода `ContactService.findAll()` мы хотим применять код, подобный показанному в листинге 13.4.

**Листинг 13.4. Использование аннотации `@DataSets`**

```
@DataSets(setUpDataSet="/com/apress/prospring4/ch13/ContactServiceImplTest.xls")
@Test
public void testFindAll() throws Exception {
    List<Contact> result = contactService.findAll();
    ...
}
```

Применение аннотации `@DataSets` к тестовому сценарию отражает тот факт, что перед запуском теста тестовые данные должны быть загружены в базу из указанного файла Excel.

Первым делом мы должны объявить специальную аннотацию, как показано в листинге 13.5.

**Листинг 13.5. Специальная аннотация (`@DataSets`)**

```
package com.apress.prospring4.ch13;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
```

```
public @interface DataSets {
   String setUpDataSet() default "";
}
```

Специальная аннотация `@DataSets` является аннотацией уровня методов. Кроме того, за счет реализации интерфейса `TestExecutionListener` будет разработан специальный класс прослушивателя выполнения тестов (листинг 13.6).

**Листинг 13.6. Специальный прослушиватель выполнения тестов**

```
package com.apress.prospring4.ch13;

import org.dbunit.IDatabaseTester;
import org.dbunit.dataset.IDataSet;
import org.dbunit.util.fileloader.XlsDataFileLoader;
import org.springframework.test.context.TestContext;
import org.springframework.test.context.TestExecutionListener;

public class ServiceTestExecutionListener implements TestExecutionListener {
   private IDatabaseTester databaseTester;
   @Override
   public void afterTestClass(TestContext arg0) throws Exception {
   }
   @Override
   public void afterTestMethod(TestContext arg0) throws Exception {
      if (databaseTester != null) {
         databaseTester.onTearDown();
      }
   }
   @Override
   public void beforeTestClass(TestContext arg0) throws Exception {
   }
   @Override
   public void beforeTestMethod(TestContext testCtx) throws Exception {
      DataSets dataSetAnnotation =
        testCtx.getTestMethod().getAnnotation(DataSets.class);
      if (dataSetAnnotation == null ) {
         return;
      }
      String dataSetName = dataSetAnnotation.setUpDataSet();
      if (!dataSetName.equals("") ) {
         databaseTester = (IDatabaseTester)
               testCtx.getApplicationContext().getBean("databaseTester");
         XlsDataFileLoader xlsDataFileLoader = (XlsDataFileLoader)
             testCtx.getApplicationContext().getBean("xlsDataFileLoader");
         IDataSet dataSet = xlsDataFileLoader.load(dataSetName);
         databaseTester.setDataSet(dataSet);
         databaseTester.onSetup();
      }
   }
   @Override
   public void prepareTestInstance(TestContext arg0) throws Exception {
   }
}
```

После реализации интерфейса `TestExecutionListener` осталось реализовать несколько методов. Однако в рассматриваемом случае нас интересуют только методы `beforeTestMethod()` и `afterTestMethod()`, в которых производится наполнение и очистка тестовых данных перед и после выполнения каждого тестового метода. Обратите внимание, что каждому методу среда Spring будет передавать экземпляр класса `TestContext`, поэтому метод может получать доступ к `ApplicationContext`, загруженному Spring Framework.

Метод `beforeTestMethod()` представляет особый интерес. Во-первых, он проверяет существование аннотации `@DataSets` для тестового метода. Если аннотация существует, тестовые данные будут загружены из указанного файла Excel. В этом случае интерфейс `IDatabaseTester` (с классом реализации `org.dbunit.DataSourceDatabaseTester`, который будет обсуждаться далее) получается из `TestContext`. Интерфейс `IDatabaseTester` предоставлен DbUnit и поддерживает операции базы данных на основе имеющегося подключения к базе данных либо источника данных.

Во-вторых, экземпляр класса `XlsDataFileLoader` получается из `TestContext`. Класс `XlsDataFileLoader` — это поддержка DbUnit загрузки данных из файла Excel. Внутренне для чтения файлов в формате Microsoft Office она использует библиотеку Apache POI. После этого для загрузки данных из файла вызывается метод `XlsDataFileLoader.load()`, который возвращает экземпляр реализации интерфейса `IDataSet`, представляющий загруженный набор данных.

Наконец, в-третьих, для установки тестовых данных вызывается метод `IDatabaseTester.setDataSet()`, а для запуска наполнения данными — метод `IDatabaseTester.onSetup()`.

В методе `afterTestMethod()` для очистки данных вызывается метод `IDatabaseTester.onTearDown()`.

### *Реализация класса конфигурации*

Теперь давайте реализуем класс конфигурации для среды тестирования. В листинге 13.7 представлен код, использующий конфигурацию в стиле Java Config.

**Листинг 13.7. Класс `ServiceTestConfig`**

```
package com.apress.prospring4.ch13;

import javax.sql.DataSource;

import org.dbunit.DataSourceDatabaseTester;
import org.dbunit.util.fileloader.XlsDataFileLoader;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.datasource.embedded.
EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

@Configuration
@ImportResource("classpath:META-INF/spring/datasource-tx-jpa.xml")
@ComponentScan(basePackages={"com.apress.prospring4.ch13"})
@Profile("test")
```

```
public class ServiceTestConfig {
    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2)
            .addScript("classpath:schema.sql").build();
    }
    @Bean(name="databaseTester")
        public DataSourceDatabaseTester dataSourceDatabaseTester() {
        DataSourceDatabaseTester databaseTester =
            new DataSourceDatabaseTester(dataSource());
        return databaseTester;
    }
    @Bean(name="xlsDataFileLoader")
    public XlsDataFileLoader xlsDataFileLoader() {
        return new XlsDataFileLoader();
    }
}
```

Здесь класс `ServiceTestConfig` определяет `ApplicationContext` для тестирования уровня обслуживания. Во-первых, импортируется XML-файл конфигурации `datasource-tx-jpa.xml`, в котором определена транзакция и конфигурация JPA, повторно используемые для тестирования. Затем применяется аннотация `@ComponentScan`, сообщающая Spring о необходимости сканировать бины уровня обслуживания, которые мы хотим тестировать. Аннотация `@Profile` указывает, что бины, определенные в этом классе, принадлежат профилю `test`.

Во-вторых, внутри класса объявляется еще один бин `dataSource`, который выполняет только сценарий `schema.sql` в базе данных H2, не наполняя ее информацией. Для загрузки данных из файла Excel специальный прослушиватель выполнения тестов использует бины `databaseTester` и `xlsDataFileLoader`. Обратите внимание, что бин `dataSourceDatabaseTester` сконструирован с применением бина `dataSource`, определенного для среды тестирования.

## Модульное тестирование уровня обслуживания

Начнем с модульного тестирования методов поиска, включая `ContactService.findAll()` и `ContactService.findByFirstNameAndLastName()`. Прежде всего, необходимо подготовить тестовые данные в формате Excel. Общепринятая практика предусматривает помещение файла в папку, в которой находится класс тестового сценария, и назначение ему того же самого имени. Таким образом, в рассматриваемом случае именем файла будет `/src/test/java/com/apress/prospring4/ch13/data/ContactServiceImplTest.xls`.

Тестовые данные подготовлены на рабочем листе. Именем рабочего листа является имя таблицы (`CONTACT`), и в первой строке указаны имена столбцов этой таблицы. Начиная со второй строки, вводятся данные для имени, фамилии и даты рождения. Мы указываем столбец идентификатора `ID`, но не предоставляем для него значение. Причина в том, что идентификаторы будут заполняться самой базой данных. С примером такого файла Excel можно ознакомиться в рамках файлов исходного кода, связанных с этой книгой.

В листинге 13.8 показан тестовый класс с тестовыми сценариями для двух методов поиска.

**Листинг 13.8. Тестирование методов поиска**

```
package com.apress.prospring4.ch13;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNull;

import java.util.List;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.TestExecutionListeners;
import org.springframework.test.context.junit4.AbstractTransactionalJUnit4S
pringContextTests;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner. class)
@ContextConfiguration(classes = {ServiceTestConfig.class})
@TestExecutionListeners({ServiceTestExecutionListener.class})
@ActiveProfiles("test")
public class ContactServiceImplTest
       extends AbstractTransactionalJUnit4SpringContextTests {
   @Autowired
   ContactService contactService;

   @DataSets(setUpDataSet=
      "/com/apress/prospring4/ch13/ContactServiceImplTest.xls")
   @Test
   public void testFindAll() throws Exception {
      List<Contact> result = contactService.findAll();

      assertNotNull(result);
      assertEquals(1, result.size());
   }

   @DataSets(setUpDataSet=
      "/com/apress/prospring4/ch13/ContactServiceImplTest.xls")
   @Test
   public void testFindByFirstNameAndLastName_1() throws Exception {
      Contact result =
          contactService.findByFirstNameAndLastName("Chris", "Schaefer");
      assertNotNull(result);
   }

   @DataSets(setUpDataSet=
      "/com/apress/prospring4/ch13/ContactServiceImplTest.xls")
   @Test
   public void testFindByFirstNameAndLastName_2() throws Exception {
      Contact result =
          contactService.findByFirstNameAndLastName("Peter", "Chan");
      assertNull(result);
   }
}
```

Здесь используется та же самая аннотация `@RunWith`, что и при тестировании класса контроллера. Аннотация `@ContextConfiguration` указывает, что конфигурация `ApplicationContext` должна загружаться из класса `ServiceTestConfig`. Аннотация `@TestExecutionListeners` обозначает то, что класс `ServiceTestExecutionListener` должен применяться для перехвата жизненного цикла выполнения тестовых сценариев. Аннотация `@ActiveProfiles` задает используемый профиль. Таким образом, в этом случае будет загружаться бин `dataSource`, определенный в классе `ServiceTestConfig`, а не бин, который определен в файле `datasource-tx-jpa.xml`, поскольку он принадлежит профилю `dev`.

Кроме того, класс расширяет Spring-класс `AbstractTransactionalJUnit4SpringContextTests`, который относится к поддержке JUnit в Spring, с наличием DI и механизма управления транзакциями. Обратите внимание, что в среде тестирования Spring будет производить откат транзакции после выполнения каждого тестового метода, так что произойдет откат всех операций обновления базы данных. Для управления поведением отката можно пользоваться аннотацией `@Rollback` на уровне методов.

В коде предусмотрен один тестовый сценарий для метода `findAll()` и два тестовых сценария для метода `testFindByFirstNameAndLastName()` (один извлекает результат, а другой — нет). Ко всем методам поиска применяется аннотация `@DataSets` с указанным файлом Excel, содержащим тестовые данные. Вдобавок служба `ContactService` автоматически связывается с тестовым сценарием из `ApplicationContext`. Остальной код должен быть самоочевидным. В каждом тестовом сценарии используются разнообразные операторы утверждений, которые проверяют, является ли результат ожидаемым.

Запустите этот тестовый сценарий и удостоверьтесь в его успешном прохождении. Теперь давайте протестируем операцию сохранения. Мы планируем тестировать два случая. Один из них представляет собой нормальную ситуацию, при которой допустимый контакт сохраняется успешно, а другой — ситуацию, когда контакт содержит ошибку, которая приводит к генерации корректного исключения. В листинге 13.9 приведен дополнительный фрагмент кода для этих двух тестовых сценариев.

**Листинг 13.9. Тестирование операции сохранения**

```
package com.apress.prospring4.ch13;

...

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import javax.validation.ConstraintViolationException;

...

public class ContactServiceImplTest extends AbstractServiceImplTest {
    @PersistenceContext
    private EntityManager em;

    ...

    @Test
```

```
    public void testAddContact() throws Exception {
        deleteFromTables("CONTACT");

        Contact contact = new Contact();
        contact.setFirstName("Rod");
        contact.setLastName("Johnson");

        contactService.save(contact);
        em.flush();

        List<Contact> contacts = contactService.findAll();
        assertEquals(1, contacts.size());
    }

    @Test(expected=ConstraintViolationException.class)
    public void testAddContactWithJSR349Error() throws Exception {
        deleteFromTables("CONTACT");

        Contact contact = new Contact();

        contactService.save(contact);
        em.flush();

        List<Contact> contacts = contactService.findAll();
        assertEquals(0, contacts.size());
    }
}
```

В показанном выше листинге обратите внимание на метод `testAddContact()`. Внутри этого метода для гарантии того, что в таблице `CONTACT` нет никаких данных, мы вызываем удобный метод `deleteFromTables()`, предоставляемый классом `AbstractTransactionalJUnit4SpringContextTests`, который очищает таблицу. После вызова операции сохранения мы должны явно вызвать метод `EntityManager.flush()`, чтобы заставить Hibernate сбросить контекст постоянства в базу данных, и тогда метод `findAll()` сможет корректно извлечь информацию из базы данных.

Во втором тестовом методе, `testAddContactWithJSR349Error()`, мы тестируем операцию сохранения контакта, содержащего ошибку проверки достоверности. Здесь в аннотации `@Test` передается атрибут `expected`, который указывает, что тестовый сценарий ожидает генерации исключения заданного типа, в данном случае — класса `ConstraintViolationException`.

Запустите тестовый класс снова и проверьте, успешен ли результат.

Следует отметить, что здесь были рассмотрены только наиболее часто используемые классы в инфраструктуре тестирования Spring. Эта инфраструктура предлагает множество поддерживающих классов и аннотаций, которые делают возможным точный контроль на протяжении жизненного цикла тестового сценария. Например, аннотации `@BeforeTransaction` и `@AfterTransaction` позволяют запускать определенную логику перед тем, как Spring инициирует транзакцию, или после того, как транзакция завершена для тестового сценария. За более подробными сведениями о разнообразных аспектах инфраструктуры тестирования Spring обращайтесь в справочную документацию по Spring.

# Реализация модульного тестирования интерфейсной части

Еще одной интересной областью тестирования является проверка поведения интерфейсной части как единого целого после развертывания веб-приложения в веб-контейнере, подобном Apache Tomcat.

Основная причина такого тестирования заключается в том, что хотя мы тестируем каждый уровень внутри приложения, мы по-прежнему должны удостовериться, что представления ведут себя корректно при различных действиях со стороны пользователей. Автоматизация тестирования интерфейсной части очень важна для экономии времени разработчиков и пользователей при наличии в тестовом сценарии повторяющихся действий в интерфейсной части.

Тем не менее, разработка тестового сценария для интерфейсной части — задача непростая, особенно в приложениях с большим количеством интерактивных, функционально насыщенных и основанных на Ajax компонентов.

## Введение в Selenium

Selenium — это мощный и комплексный инструмент, а также инфраструктура, которые вместе предназначены для автоматизации тестирования интерфейсной части. Главная особенность связана с тем, что посредством Selenium можно "управлять" браузерами, эмулируя взаимодействие пользователей с приложением, и контролировать состояние представлений.

Инфраструктура Selenium поддерживает такие распространенные браузеры, как Firefox, IE и Chrome. Что касается языков, то поддерживаются Java, C#, PHP, Perl, Ruby и Python. Инструмент Selenium также спроектирован с учетом Ajax-приложений и насыщенных Интернет-приложений (RIA), делая возможным тестирование современных веб-приложений.

На тот случай, когда приложение имеет множество пользовательских интерфейсов, и для него необходимо прогонять большое количество тестов интерфейсной части, модуль `selenium-server` предлагает встроенную функциональность сетки, которая поддерживает выполнение тестов интерфейсной части в группе компьютеров.

Selenium IDE — это подключаемый модуль для браузера Firefox, который помогает "записывать" взаимодействие пользователя с веб-приложением. Он также поддерживает повторное воспроизведение и экспортирует сценарии в различные форматы, упрощая разработку тестовых сценариев.

Начиная с версии 2.0, Selenium интегрируется с интерфейсом WebDriver API, который снимает множество ограничений и предлагает альтернативный и более простой программный интерфейс. Результатом является всеобъемлющий объектно-ориентированный API-интерфейс, который предлагает дополнительную поддержку для многочисленных браузеров наряду с улучшенной поддержкой решения проблем, связанных с тестированием современных сложных веб-приложений.

Тестирование интерфейсной части веб-приложений является сложной темой и выходит за рамки материала настоящей книги. Благодаря этому простому введению, вы видите, что Selenium помогает автоматизировать взаимодействие пользователя с интерфейсной частью веб-приложения, совместимое с множеством браузеров.

За дополнительными сведениями обращайтесь к онлайновой документации по Selenium (`http://seleniumhq.org/docs`).

# Резюме

В этой главе мы показали, как разрабатывать различные виды модульного тестирования в Spring-приложениях с помощью часто используемых инфраструктур, библиотек и инструментов, включая JUnit, DbUnit и Mockito.

Во-первых, мы представили высокоуровневое описание корпоративной инфраструктуры тестирования, в котором показано, какие тесты должны выполняться на каждой фазе жизненного цикла разработки приложения. Во-вторых, мы разработали два типа тестов — модульный тест логики и модульный тест взаимодействия. Кроме того, мы кратко затронули тему инфраструктуры тестирования интерфейсной части, которая называется Selenium.

Тестирование корпоративного приложения — это очень крупная тема, и если вы хотите более детально изучить библиотеку JUnit, рекомендуем почитать книгу *JUnit in Action, Second Edition* (Manning, 2010 г.).

ГЛАВА 14

# Поддержка написания сценариев в Spring

В предшествующих главах вы узнали, каким образом Spring Framework помогает Java-разработчикам создавать JEE-приложения. За счет использования механизма внедрения зависимостей Spring Framework и его интеграции с каждым уровнем (через библиотеки внутри собственных модулей Spring Framework или путем взаимодействия с библиотеками третьих сторон) можно упростить реализацию и сопровождение бизнес-логики.

Однако для всей логики, разработанной до сих пор, применялся язык Java. Хотя Java является одним из наиболее успешных языков программирования за всю историю отрасли, его по-прежнему критикуют за некоторые недостатки, такие как структура языка и отсутствие всесторонней поддержки в областях, подобных массивной параллельной обработке.

Например, одна из характеристик языка Java заключается в том, что все переменные являются статически типизированными. Другими словами, в Java-программе каждая объявленная переменная должна иметь связанный с ней статический тип (`String`, `int`, `Object`, `ArrayList` и т.д.). Тем не менее, в ряде сценариев более предпочтительной может оказаться динамическая типизация, которая поддерживается динамическими языками вроде JavaScript.

Для решения таких проблем были разработаны многочисленные языки сценариев. В число самых популярных языков сценариев входят JavaScript, Groovy, Scala, Ruby и Erlang. Почти все эти языки поддерживают динамическую типизацию и спроектированы на предоставление средств, которые не доступны в Java, а также для удовлетворения других специальных целей. Например, язык Scala (`www.scala-lang.org`) комбинирует шаблоны функционального программирования с объектно-ориентированными шаблонами и поддерживает более развитую и масштабируемую модель параллельного программирования с концепциями акторов и передачи сообщений. Вдобавок язык Groovy (`www.groovy-lang.org`) предлагает упрощенную модель программирования и поддерживает реализацию языков, специфичных для предметной области (domain-specific language — DSL), которые делают код приложения проще в чтении и сопровождении.

Еще одной важной концепцией, которую языки сценариев предоставляют Java-разработчикам, являются замыкания (мы обсудим их более подробно позже в этой главе). Говоря упрощенно, *замыкание* (closure) — это фрагмент (или блок) кода, по-

мещенный в объект. Подобно Java-методу, замыкание является исполняемым компонентом и может получать параметры, а также возвращать объекты и значения. Кроме того, оно также представляет собой нормальный объект, который может передаваться с помощью ссылки куда угодно внутри приложения, как и любой объект POJO в Java.

В этой главе мы рассмотрим некоторые основные концепции, связанные с языками сценариев, уделяя особое внимание Groovy; вы увидите, что Spring Framework может гладко работать с языком сценариев, предлагая специфическую функциональность Spring-приложениям. В частности, в этой главе будут освещены следующие темы.

- **Поддержка написания сценариев в Java.** В рамках JCP спецификация JSR-223 ("Scripting for the Java Platform" — "Написание сценариев для платформы Java") регламентирует поддержку языков сценариев в Java; она доступна в Java, начиная с версии 6. Мы предложим обзор поддержки написания сценариев в Java.
- **Язык Groovy.** Мы представим высокоуровневое введение в язык Groovy, который является одним из наиболее популярных языков сценариев, используемых вместе с Java, особенно при работе с платформой Spring Framework.
- **Использование Groovy в Spring.** Платформа Spring Framework обеспечивает комплексную поддержку для языков сценариев. Начиная с версии 3.1, имеется готовая поддержка языков Groovy, JRuby и BeanShell.

Эта глава не предназначена служить детальным справочником по применению языков сценариев. По каждому языку написана одна или несколько книг, в которых подробно описана их структура и способы использования. Основная цель этой главы состоит в том, чтобы продемонстрировать главную идею поддержки языков сценариев в Spring Framework и предоставить характерный пример преимуществ применения того или иного языка сценариев в дополнение к Java при разработке Spring-приложений.

# Работа с поддержкой написания сценариев в Java

Начиная с версии Java 6, в JDK имеется встроенная поддержка API-интерфейса написания сценариев для платформы Java (спецификация JSR-223). Цель заключается в предоставлении стандартного механизма для выполнения кода, написанного на других языках сценариев, в среде JVM. В состав JDK 6 входит механизм под названием Mozilla Rhino, который способен анализировать JavaScript-программы. В этом разделе мы предоставим введение в поддержку JSR-223 внутри JDK 6.

В JDK 6 классы поддержки сценариев находятся в пакете `javax.script`. Давайте начнем с разработки простой программы для извлечения списка механизмов сценариев. Соответствующий класс показан в листинге 14.1.

**Листинг 14.1. Вывод списка механизмов сценариев**

```
package com.apress.prospring4.ch14;

import javax.script.ScriptEngineFactory;
import javax.script.ScriptEngineManager;
```

```
public class ListScriptEngines {
   public static void main(String[] args) {
      ScriptEngineManager mgr = new ScriptEngineManager();

      for (ScriptEngineFactory factory : mgr.getEngineFactories()) {
         String engineName= factory.getEngineName();
         String languageName = factory.getLanguageName();
         String version = factory.getLanguageVersion();

         // Вывод имени механизма, языка и версии
         System.out.println("Engine name: " + engineName + " Language: " +
                            languageName  + "version: " + version);
      }
   }
}
```

В коде создается экземпляр класса `ScriptEngineManager`, который будет обнаруживать и обрабатывать список механизмов (другими словами, классов, реализующих интерфейс `javax.script.ScriptEngine`) в пути классов. Затем с помощью вызова метода `ScriptEngineManager.getEngineFactories()` извлекается список интерфейсов `ScriptEngineFactory`. Интерфейс `ScriptEngineFactory` используется для описания и создания экземпляров механизмов сценариев. Из каждого интерфейса `ScriptEngineFactory` можно извлечь информацию о поддерживаемом языке сценариев. В зависимости от настроек системы, выполнение программы может дать варьирующийся вывод, который должен выглядеть подобно приведенному ниже:

```
Engine name: AppleScriptEngine Language: AppleScript Version: 2.3.1
Engine name: Oracle Nashorn Language: ECMAScript Version:
ECMA - 262 Edition 5.1
```

А теперь напишем простую программу для оценки базового выражения JavaScript. Код программы показан в листинге 14.2.

**Листинг 14.2. Оценка выражения JavaScript**

```
package com.apress.prospring4.ch14;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class JavaScriptTest {
   public static void main(String[] args) {
      ScriptEngineManager mgr = new ScriptEngineManager();
      ScriptEngine jsEngine = mgr.getEngineByName("JavaScript");

      try {
         jsEngine.eval("print('Hello JavaScript in Java')");
      } catch (ScriptException ex) {
         ex.printStackTrace();
      }
   }
}
```

Здесь экземпляр реализации интерфейса `ScriptEngine` извлекается из класса `ScriptEngineManager` с применением имени `JavaScript`. Затем вызывается метод `ScriptEngine.eval()` с передачей ему аргумента типа `String`, который содержит выражение JavaScript. Обратите внимание, что аргументом также может быть класс `java.io.Reader`, который способен читать JavaScript-код из файла. Выполнение этой программы приводит к получению следующего вывода:

```
Hello JavaScript in Java
```

Это должно дать вам представление о способе запуска сценариев в Java. Тем не менее, простое дублирование вывода, полученного из сценариев на других языках, не особенно интересно. В следующем разделе мы предложим введение в Groovy — мощный и развитый язык сценариев.

# Введение в Groovy

Проект Groovy был начат в 2003 году Джеймсом Стрэченом (James Strachan). Основной целью этого проекта является предоставление гибкого и динамического языка для JVM с возможностями, характерными для других популярных языков сценариев, включая Python, Ruby и Smalltalk. Язык Groovy построен поверх Java, расширяет Java и ликвидирует некоторые недостатки Java.

В последующих разделах мы обсудим ряд основных средств и концепций, связанных с языком Groovy, и посмотрим, как он дополняет Java с целью решения специфичных нужд в приложениях. Следует отметить, что многие упоминаемые здесь возможности также доступны в других языках сценариев (например, Scala, Erlang, Python и Clojure).

## Динамическая типизация

Одно из основных отличий Groovy (и многих других языков сценариев) от Java связано с поддержкой динамической типизации переменных. В Java все свойства и переменные должны быть статически типизированными. Другими словами, в операторе объявления для них должен быть указан тип. Однако Groovy поддерживает динамическую типизацию переменных. Динамически типизированные переменные в Groovy объявляются с помощью ключевого слова `def`.

Давайте посмотрим на это в действии, разработав простой сценарий Groovy. Суффиксом файла с классом или сценарием Groovy является `groovy`. В листинге 14.3 приведен простой сценарий Groovy, в котором демонстрируется динамическая типизация.

**Листинг 14.3. Динамическая типизация в Groovy**

```
class Contact {
    def firstName
    def lastName
    def birthDate

    String toString() {
        "($firstName,$lastName,$birthDate)"
    }
}
```

```
Contact contact =
  new Contact(firstName: 'Chris', lastName: 'Schaefer', birthDate: new Date())
Contact anotherContact =
  new Contact(firstName: 42, lastName: 'Schaefer', birthDate: new Date())

println contact
println anotherContact

println contact.firstName + 42
println anotherContact.firstName + 42
```

Этот сценарий Groovy может быть запущен непосредственно в IDE-среде, выполнен без компиляции (Groovy предоставляет инструмент командной строки под названием `groovy`, который выполняет сценарии Groovy напрямую) или же скомпилирован в байт-код Java и затем запущен подобно другим Java-классам. Сценарии Groovy для своего выполнения не требуют наличия метода `main()`. Кроме того, объявление класса, имя которого совпадает с именем файла, не обязательно.

В данном примере определяется класс `Contact` со свойствами, динамически типизированными с помощью ключевого слова `def`. Объявлены три таких свойства. Затем метод `toString()` переопределяется с использованием замыкания, которое возвращает строку.

Далее конструируются два объекта `Contact` посредством сокращенного синтаксиса, предоставляемого Groovy для определения свойств. Для первого объекта `Contact` атрибут `firstName` задается как значение `String`, в то время как для второго объекта `Contact` — в виде целого числа. Наконец, с помощью оператора `println` (его действие аналогично вызову метода `System.out.println()`) выводятся два объекта контактов. Чтобы посмотреть, как Groovy обрабатывает динамическую типизацию, предусмотрены два оператора `println` для вывода результата выполнения операции `firstName + 20`. Обратите внимание, что в Groovy при передаче аргумента методу круглые скобки не обязательны.

Запуск этой программы дает следующий вывод:

```
(Chris,Schaefer,Thu Mar 13 14:01:38 EDT 2014)
(42,Schaefer,Thu Mar 13 14:01:38 EDT 2014)
Chris42
84
```

Как видно в выводе, поскольку свойство определено с динамической типизацией, объект успешно конструируется при передаче значения либо типа `String`, либо целочисленного. Кроме того, в последних двух операторах `println` операция сложения была корректно применена к свойству `firstName` обоих объектов. В первом случае из-за того, что `firstName` имеет тип `String`, к значению свойства дописывается строка `20`. Во втором случае, т.к. `firstName` имеет целочисленный тип, к нему добавляется целое число `20` и результатом оказывается `40`.

Поддержка динамической типизации в Groovy обеспечивает высокую гибкость при манипулировании свойствами класса и переменными в коде приложения.

## Упрощенный синтаксис

Язык Groovy также предоставляет упрощенный синтаксис, поэтому одна и та же логика при реализации на Groovy будет занимать меньший объем кода, чем в случае Java. Ниже перечислены некоторые особенности этого синтаксиса.

- Для завершения оператора точка с запятой не требуется.
- Ключевое слово `return` в методах является необязательным.
- Все методы и классы по умолчанию определены как `public`. Таким образом, использовать ключевое слово `public` в объявлениях методов не обязательно.
- Внутри класса Groovy автоматически генерирует методы извлечения/установки для объявленных свойств. Это значит, что в классе Groovy нужно только объявить тип и имя свойства (например, `String firstName` или `def firstName`), а затем получать доступ к нему в любом другом классе Groovy/Java, и методы извлечения/установки будут применяться автоматически. Вдобавок к свойству можно обращаться без префикса `get/set` (например, `contact.firstName = 'Chris'`). Groovy обработает это должным образом.

Кроме того, Groovy предлагает упрощенный синтаксис и много удобных методов для работы с API-интерфейсом коллекций Java. В листинге 14.4 демонстрируется использование некоторых распространенных операций Groovy для манипулирования списками.

**Листинг 14.4. Манипулирование списками в Groovy**

```
def list = ['This', 'is', 'Chris']
assert list.size() == 3
assert list.class == ArrayList
assert list.reverse() == ['Chris', 'is', 'This']
assert list.sort{ it.size() } == ['is', 'This', 'Chris']
assert list[0..1] == ['is', 'This']
```

В листинге представлена только очень малая часть возможностей, поддерживаемых Groovy. За более подробными сведениями обращайтесь в онлайновую документацию по адресу `http://www.groovy-lang.org/documentation.html`.

## Замыкание

Одним из наиболее важных средств, которые Groovy добавляет к Java, является поддержка замыканий. *Замыкание* позволяет поместить порцию кода в объект и свободно передавать в рамках приложения. Замыкание — это мощное средство, которое делает возможным интеллектуальное и динамическое поведение. Добавление поддержки замыканий к языку Java запрашивалось на протяжении длительного времени. Спецификация JSR-335 ("Lambda Expressions for the Java Programming Language" — "Лямбда-выражения для языка программирования Java"), которая направлена на поддержку программирования в многоядерной среде за счет добавления замыканий и связанных возможностей в язык Java, была включена в Java 8 и поддерживается в новой версии Spring Framework 4.

В листинге 14.5 приведен простой пример применения замыканий (файл SimpleClosure.groovy) в Groovy.

**Листинг 14.5. Простой пример замыкания**

```
def names = ['Chris', 'Johnny', 'Mary']
names.each {println 'Hello: ' + it}
```

Здесь объявляется список. Затем с помощью удобного метода each() выполняется операция над каждым элементом в списке. Аргументом метода each() является замыкание, которое в Groovy заключается в фигурные скобки. В результате логика из замыкания будет применена к каждому элементу списка. Внутри замыкания it — это специальная переменная, которая используется Groovy для представления элемента, находящегося в контексте. Таким образом, замыкание обеспечит добавление к каждому элементу списка префикса в виде строки "Hello: " и его вывод. Выполнение сценария даст следующий вывод:

```
Hello: Chris
Hello: Johnny
Hello: Mary
```

Как уже упоминалось, замыкание можно объявить как переменную и затем применять по мере необходимости. В листинге 14.6 показан еще один пример.

**Листинг 14.6. Определение замыкания как переменной**

```
def map = ['a': 10, 'b': 50]
Closure square = {key, value -> map[key] = value * value}
map.each square
println map
```

В этом примере определяется карта (map) и объявляется переменная square типа Closure. Замыкание принимает в качестве аргументов ключ и значение элемента карты, а логика вычисляет квадрат значения, связанного с ключом. Запуск программы дает следующий вывод:

```
[a:100, b:2500]!
```

Мы предоставили лишь простое введение в замыкания. В следующем разделе мы разработаем простой процессор правил (rule engine), используя Groovy и Spring; при этом также будут применяться замыкания. За более подробными сведениями об использовании замыканий в Groovy обращайтесь в онлайновую документацию по адресу http://www.groovy-lang.org/documentation.html.

# Использование Groovy в Spring

Основное преимущество, которое Groovy и другие языки сценариев привносят в Java-приложения — это поддержка динамического поведения. С использованием замыкания бизнес-логику можно упаковать внутрь объекта и передавать в рамках приложения подобно любой другой переменной.

Еще одной важной возможностью Groovy является поддержка разработки языков DSL с применением упрощенного синтаксиса и замыканий. Как уже упоминалось, DSL — это язык, ориентированный на конкретную предметную область с очень специфичными целями в проектировании и реализации. Задача состоит в том, чтобы построить язык, который будет понятен не только разработчикам, но также бизнес-аналитикам и пользователям. Большую часть времени предметной областью будет выступать сфера бизнеса. Например, языки DSL могут быть определены для классификации заказчиков, калькуляции налога с продаж, расчета заработной платы и т.д.

В этом разделе мы продемонстрируем использование Groovy для реализации простого процессора правил с помощью поддержки DSL в Groovy.

Реализация повторяет пример из великолепной статьи по этой теме, доступной по адресу `www.pleus.net/articles/grules/grules.pdf`, но с некоторыми изменениями. Кроме того, мы покажем, каким образом поддержка обновляемых бинов в Spring делает возможным обновление правил на лету, не требуя компиляции, упаковки и развертывания приложения.

В этом примере мы реализуем правило, применяемое для классификации специфического контакта по различным категориям, основанным на возрасте, который вычисляется с использованием свойства даты рождения.

## Добавление обязательных зависимостей

В рассматриваемом примере мы будем применять язык Groovy и библиотеку Joda-Time, поэтому нам понадобится добавить в проект обязательные зависимости, описанные в табл. 14.1.

**Таблица 14.1. Зависимости Maven для Groovy и Joda-Time**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.codehaus.groovy` | `groovy-all` | 2.2.2 | Библиотека Groovy |
| `joda-time` | `joda-time` | 2.3 | Библиотека для работы с типами даты и времени Joda-Time |

## Разработка предметной области, связанной с контактами

Как упоминалось ранее, язык DSL ориентирован на специфическую предметную область, которая в большинстве случаев имеет отношение к некоторому виду бизнес-данных. Правило, которое мы собираемся реализовать, предназначено для применения к предметной области, связанной с информацией о контактах.

Таким образом, первый шаг связан с разработкой объектной модели предметной области, к которой должно применяться правило. В этом примере модель очень проста и содержит единственный сущностный класс `Contact`, код которого приведен в листинге 14.7. Обратите внимание, что это класс POJO, похожий на классы подобного рода, которые встречались в предшествующих главах.

**Листинг 14.7. Предметная область `Contact`**

```
package com.apress.prospring4.ch14;
import org.joda.time.DateTime;
```

```
public class Contact {
   private Long id;
   private String firstName;
   private String lastName;
   private DateTime birthDate;
   private String ageCategory;

   public Long getId() {
      return id;
   }

   public void setId(Long id) {
      this.id = id;
   }

   public String getFirstName() {
      return firstName;
   }

   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }

   public String getLastName() {
      return lastName;
   }

   public void setLastName(String lastName) {
      this.lastName = lastName;
   }

   public DateTime getBirthDate() {
      return birthDate;
   }

   public void setBirthDate(DateTime birthDate) {
      this.birthDate = birthDate;
   }

   public String getAgeCategory() {
      return ageCategory;
   }

   public void setAgeCategory(String ageCategory) {
      this.ageCategory = ageCategory;
   }

   @Override
   public String toString() {
      return "Contact - Id: " + id + ", First name: " + firstName
            + ", Last name: " + lastName + ", Birthday: " + birthDate
            + ", Age category: " + ageCategory;
   }
}
```

Класс `Contact` представляет простую информацию о контакте. Для свойства `ageCategory` мы хотим разработать динамическое правило, которое можно использовать для проведения классификации. Это правило будет вычислять возраст на основе свойства `birthDate` и затем устанавливать свойство `ageCategory` в зависимости от результата (например, `kid`, `youth` или `adult`).

## Реализация процессора правил

Следующий шаг заключается в разработке простого процессора правил, предназначенного для применения правил к объекту предметной области. Для начала необходимо определить, какую информацию должно содержать правило.

В листинге 14.8 показан класс `Rule`, который представляет собой класс Groovy (файл `Rule.groovy`).

**Листинг 14.8. Класс `Rule`**

```
package com.apress.prospring4.ch14

class Rule {
    private boolean singlehit = true
    private conditions = new ArrayList()
    private actions = new ArrayList()
    private parameters = new ArrayList()
}
```

Каждое правило имеет несколько свойств. Свойство `conditions` определяет различные условия, которые процессор правил должен проверять для обрабатываемого объекта предметной области. Свойство `actions` определяет действия, которые должны предприниматься в случае удовлетворения условий. Свойство `parameters` определяет поведение правила, которое является результатом действий для разных условий. Наконец, свойство `singlehit` определяет, должно ли правило немедленно завершать свое выполнение всякий раз, когда условия удовлетворены.

На следующем шаге строится процессор для выполнения правил. В листинге 14.9 приведен интерфейс `RuleEngine` (обратите внимание, что это Java-интерфейс).

**Листинг 14.9. Интерфейс `RuleEngine`**

```
package com.apress.prospring4.ch14;

public interface RuleEngine {
    void run(Rule rule, Object object);
}
```

В этом интерфейсе определен единственный метод `run()`, который предназначен для применения правила к аргументу — объекту предметной области.

Мы предоставим реализацию процессора правил на языке Groovy.

В листинге 14.10 показан код класса Groovy по имени `RuleEngineImpl` (файл `RuleEngineImpl.groovy`).

**Листинг 14.10. Groovy-класс `RuleEngineImpl`**

```
package com.apress.prospring4.ch14

import org.springframework.stereotype.Component

@Component("ruleEngine")
class RuleEngineImpl implements RuleEngine {
    public void run(Rule rule, Object object) {
        println "Executing rule"
```

```
        def exit=false
        rule.parameters.each{ArrayList params ->
            def paramIndex=0
            def success=true
            if(!exit){
                rule.conditions.each{
                    println "Condition Param index: " + paramIndex
                    success = success && it(object,params[paramIndex])
                    println "Condition success: " + success
                    paramIndex++
                }
                if(success && !exit){
                    rule.actions.each{
                        println "Action Param index: " + paramIndex
                        it(object,params[paramIndex])
                        paramIndex++
                    }
                    if (rule.singlehit){
                        exit=true
                    }
                }
            }
        }
    }
}
```

Класс `RuleEngineImpl` реализует Java-интерфейс `RuleEngine`, и к нему применяется аннотация Spring, как к любому другому объекту POJO. Внутри метода `run()` параметры, определенные в правиле, по очереди передаются замыканию на обработку. Для каждого параметра (который является списком значений) проверяется удовлетворение условий (каждое условие представляет собой замыкание) друг за другом; в проверке участвует соответствующий элемент в списке параметра и объект предметной области. Индикатор успешности проверки (`success`) становится равным `true`, только если были удовлетворены все условия. В этом случае действия (каждое действие — это также замыкание), определенные в правиле, будут применены к объекту с использованием соответствующего значения в списке параметра. Наконец, если для специфического параметра условие удовлетворено и переменная `singlehit` равна `true`, то выполнение правила немедленно завершается.

Чтобы обеспечить более гибкий способ извлечения правила, определим интерфейс `RuleFactory` (листинг 14.11). Обратите внимание, что это Java-интерфейс.

**Листинг 14.11. Интерфейс `RuleFactory`**

```
package com.apress.prospring4.ch14;

public interface RuleFactory {
    Rule getAgeCategoryRule();
}
```

Поскольку имеется только одно правило для классификации контактов по возрастным категориям, в интерфейсе `RuleFactory` определен единственный метод для извлечения правила.

Чтобы сделать процессор правил прозрачным для потребителя, разработаем простой уровень обслуживания и поместим в него процессор. В листингах 14.12 и 14.13 представлен интерфейс `ContactService` и класс `ContactServiceImpl`. Обратите внимание, что оба они являются Java-реализациями.

**Листинг 14.12. Интерфейс `ContactService`**

```
package com.apress.prospring4.ch14;

public interface ContactService {
    void applyRule(Contact contact);
}
```

**Листинг 14.13. Класс `ContactServiceImpl`**

```
package com.apress.prospring4.ch14;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.stereotype.Service;

@Service("contactService")
public class ContactServiceImpl implements ContactService {
    @Autowired
    ApplicationContext ctx;

    @Autowired
    private RuleFactory ruleFactory;

    @Autowired
    private RuleEngine ruleEngine;

    public void applyRule(Contact contact) {
        Rule ageCategoryRule = ruleFactory.getAgeCategoryRule();
        ruleEngine.run(ageCategoryRule, contact);
    }
}
```

В листинге 14.13 требуемые бины Spring автоматически связываются с классом реализации службы. В методе `applyRule()` правило получается из фабрики правил и затем применяется к объекту `Contact`. В результате свойство `ageCategory` для объекта `Contact` будет выведено на основе условий, действий и параметров, определенных в правиле.

## Реализация фабрики правил в виде обновляемого бина Spring

Теперь мы можем реализовать фабрику правил и правило для классификации по возрастным категориям. Мы хотим иметь возможность обновлять правило на лету. Среда Spring должна отслеживать изменение правила и применять самую актуальную логику. В Spring Framework предлагается великолепная поддержка для бинов Spring, которые написаны на языках сценариев и называются *обновляемыми бинами*.

Мы покажем, как сконфигурировать сценарий Groovy в виде бина Spring и указать Spring на необходимость его обновления через регулярные интервалы. Для начала давайте рассмотрим реализацию фабрики правил в Groovy. Чтобы сделать возможным динамическое обновление, мы помещаем класс во внешнюю папку, позволяя его модифицировать. Мы назовем эту папку `rules`. В нее будет помещен класс `RuleFactoryImpl` (который является классом Groovy и находится в файле `RuleFactoryImpl.groovy`). Код этого класса показан в листинге 14.14.

**Листинг 14.14. Класс `RuleFactoryImpl`**

```
package com.apress.prospring4.ch14

import org.joda.time.DateTime
import org.joda.time.Years
import org.springframework.stereotype.Component;

@Component
class RuleFactoryImpl implements RuleFactory {
   Closure age =
      { birthDate ->
         return Years.yearsBetween(birthDate, new DateTime()).getYears() }
   public Rule getAgeCategoryRule() {
      Rule rule = new Rule()
      rule.singlehit=true
      rule.conditions=[ {object, param -> age(object.birthDate) >= param},
            {object, param -> age(object.birthDate) <= param}]
      rule.actions=[{object, param -> object.ageCategory = param}]
      rule.parameters=[
            [0,10,'Kid'],
            [11,20,'Youth'],
            [21,40,'Adult'],
            [41,60,'Middle-aged'],
            [61,120,'Old']
      ]
      return rule
   }
}
```

Класс `RuleFactoryImpl` реализует интерфейс `RuleFactory`, а метод `getAgeCategoryRule()` предназначен для предоставления правила. Внутри правила определено замыкание (`Closure`) по имени `age`, предназначенное для вычисления возраста на основе свойства `birthDate` (типа `DateTime` из Joda-Time) объекта `Contact`.

В рамках правила определены два условия. Первое условие проверяет, больше или равен возраст контакта предоставленному значению параметра, а второе — меньше или равен.

Затем определяется одно действие для присваивания значения, предоставлено в параметре, свойству `ageCategory` объекта `Contact`.

Параметры определяют значения для проверки условий и действия. Например, первый параметр означает, что если возраст находится между 0 и 10, то свойству `ageCategory` объекта `Contact` будет присвоено значение `Kid`, и т.д. Таким образом, первые два значения в каждом параметре будут использоваться двумя услови-

ями для проверки возрастного диапазона, а последнее значение будет применяться для установки свойства `ageCategory`.

Следующий шаг предусматривает определение `ApplicationContext`. Содержимое конфигурационного файла `app-context.xml` приведено в листинге 14.15.

**Листинг 14.15. Конфигурация Spring**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:lang="http://www.springframework.org/schema/lang"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/lang
       http://www.springframework.org/schema/lang/spring-lang.xsd">

   <context:component-scan base-package="com.apress.prospring4.ch14" />

   <lang:groovy id="ruleFactory" refresh-check-delay="5000"
              script-source="file:rules/RuleFactoryImpl.groovy"/>
</beans>
```

Конфигурация довольно проста. Для определения бинов Spring на языке сценариев необходимо использовать пространство имен `lang`. Дескриптор `<lang:groovy>` применяется для объявления бина Spring со сценарием Groovy.

В атрибуте `script-source` указано местоположение сценария Groovy, который будет загружаться Spring. Для обновляемого бина должен быть предоставлен атрибут `refresh-check-delay`. В этом случае мы указываем в нем значение 5000 миллисекунд, что заставляет Spring проверять изменения в файле, если с момента последнего вызова прошло более 5 секунд. Обратите внимание, что Spring не будет проверять файл каждые 5 секунд. Вместо этого проверка будет производиться только при обращении к соответствующему бину.

## Тестирование правила возрастной категории

Теперь все готово для тестирования созданного правила. Тестовая программа показана в листинге 14.16 и представляет собой Java-класс.

**Листинг 14.16. Тестирование процессора правил**

```
package com.apress.prospring4.ch14;

import org.joda.time.format.DateTimeFormat;
import org.springframework.context.support.GenericXmlApplicationContext;

public class RuleEngineTest {
   public static void main(String[] args) {
      GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();
      ctx.load("classpath:META-INF/spring/app-context.xml");
      ctx.refresh();
```

```
        ContactService contactService = ctx.getBean("contactService",
                                                    ContactService.class);

        Contact contact = new Contact();
        contact.setId(11);
        contact.setFirstName("Chris");
        contact.setLastName("Schaefer");
        contact.setBirthDate(DateTimeFormat.forPattern(
          "yyyy-MM-dd").parseDateTime("1981-05-03"));

        contactService.applyRule(contact);
        System.out.println("Contact: " + contact);

        try {
            System.in.read();
        } catch (Exception ex) {
            ex.printStackTrace();
        }

        contactService.applyRule(contact);
        System.out.println("Contact: " + contact);
    }
}
```

После инициализации `GenericXmlApplicationContext` из Spring конструируется экземпляр объекта `Contact`. Затем получается экземпляр реализации интерфейса `ContactService` для применения правила к объекту `Contact`; результат выводится на консоль. Программа останавливается в ожидании пользовательского ввода, после чего правило будет применено во второй раз. В течение этой паузы можно модифицировать класс `RuleFactoryImpl.groovy`, чтобы среда Spring обновила бин и мы увидели измененное правило в действии.

Запуск тестовой программы дает следующий вывод:

```
Executing rule
Condition Param index: 0
Condition success: true
Condition Param index: 1
Condition success: false
Condition Param index: 0
Condition success: true
Condition Param index: 1
Condition success: false
Condition Param index: 0
Condition success: true
Condition Param index: 1
Condition success: true
Action Param index: 2
Contact: Contact - Id: 1, First name: Chris, Last name: Schaefer,
Birthday: 1981-05-03T00:00:00.000-04:00, Age category: Adult
```

Поскольку возраст контакта составляет 32, мы видим, что в правиле обнаружено совпадение для третьего параметра (т.е. `[21,40,'Adult']`). В результате свойство `ageCategory` устанавливается в `Adult`. Теперь, когда программа приостановлена, давайте изменим параметры в классе `RuleFactoryImpl.groovy`. В листинге 14.17 приведен соответствующий фрагмент кода.

**Листинг 14.17. Модификация параметров правила**

```
rule.parameters=[
[0,10,'Kid'],
[11,20,'Youth'],
[21,30,'Adult'],
[31,60,'Middle-aged'],
[61,120,'Old']
]
```

Изменения выделены полужирным. Нажмите клавишу <Enter> в области консоли, чтобы инициировать второе применение правила к тому же самому объекту. На консоли появится следующий вывод:

```
Executing rule
Condition Param index: 0
Condition success: true
Condition Param index: 1
Condition success: false
Condition Param index: 0
Condition success: true
Condition Param index: 1
Condition success: false
Condition Param index: 0
Condition success: true
Condition Param index: 1
Condition success: false
Condition Param index: 0
Condition success: true
Condition Param index: 1
Condition success: true
Action Param index: 2
Contact: Contact - Id: 1, First name: Chris, Last name: Schaefer,
Birthday: 1981-05-03T00:00:00.000-04:00, Age category: Middle-aged
```

Как видите, выполнение правила остановилось на четвертом параметре (т.е. `[31,60,'Middle-aged']`), в результате чего свойству `ageCategory` было присвоено значение `Middle-aged`.

Если вы просмотрите статью, на которую мы ссылались при подготовке этого примера (`www.pleus.net/articles/grules/grules.pdf`), то найдете в ней объяснения, как вынести параметр правила во внешний файл Microsoft Excel, чтобы пользователи могли самостоятельно формировать и обновлять файл параметров.

Конечно, рассмотренное выше правило было простым, однако оно продемонстрировало, каким образом язык сценариев вроде Groovy может содействовать в дополнении приложений Java EE, основанных на Spring, средствами из специфических областей, таких как процессор правил с собственным языком DSL.

Может возникнуть вопрос о том, реально ли сохранить правило в базе данных и поручить средству обновляемых бинов Spring обнаружение изменений, внесенных в базу данных? В этом случае можно было бы еще больше упростить сопровождение правила за счет предложения пользователям интерфейсной (или административной) части, позволяющей обновлять правило в базе данных на лету, а не загружать внешний файл. В действительности существует JIRA-проблема в Spring Framework, в

которой это обсуждается (https://jira.springsource.org/browse/SPR-5106). Будьте аккуратны с описанным подходом. В то же самое время предоставление пользовательского интерфейса для загрузки класса правила также является приемлемым решением. Разумеется, в этом случае необходимо проявлять особую осторожность, и вводимое пользователем правило должно быть тщательно протестировано перед его загрузкой в производственную среду.

# Встраивание кода на динамическом языке

Код на динамическом языке можно не только выполнять из внешних файлов, но также встраивать такой код напрямую в конфигурацию бинов. Хотя подход подобного рода может оказаться удобным в ряде сценариев, таких как быстрая проверка концепций и т.д., с точки зрения сопровождения его использование для построения целиком всего приложения не рекомендуется. Давайте возьмем предыдущий процессор правил в качестве примера и удалим файл RuleEngineImpl.groovy, встроив находящийся в нем код внутрь определения бина (app-context.xml), как показано в листинге 14.18.

**Листинг 14.18. Встроенный код RuleEngineImpl.groovy в app-context.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:lang="http://www.springframework.org/schema/lang"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/lang
       http://www.springframework.org/schema/lang/spring-lang.xsd">
   <context:component-scan base-package="com.apress.prospring4.ch14" />
   <lang:groovy id="ruleFactory" refresh-check-delay="5000"
                script-source="file:rules/RuleFactoryImpl.groovy"/>
   <lang:groovy id="ruleEngine">
      <lang:inline-script>
         <![CDATA[
package com.apress.prospring4.ch14
import org.springframework.stereotype.Component
class RuleEngineImpl implements RuleEngine {
   public void run(Rule rule, Object object) {
      println "Executing rule"
      def exit=false
      rule.parameters.each{ArrayList params ->
         def paramIndex=0
         def success=true
         if(!exit){
            rule.conditions.each{
               println "Condition Param index: " + paramIndex
               success = success && it(object,params[paramIndex])
               println "Condition success: " + success
               paramIndex++
            }
```

```
            if(success && !exit){
               rule.actions.each{
                  println "Action Param index: " + paramIndex
                  it(object,params[paramIndex])
                  paramIndex++
               }
               if (rule.singlehit){
                  exit=true
               }
            }
         }
      }
   }
}
         ]]>
      </lang:inline-script>
   </lang:groovy>
</beans>
```

В коде видно, что мы добавили дескриптор `lang:groovy` с идентификатором `ruleEngine`, представляющим имя бина. После этого мы применили дескриптор `lang:inline-script` для инкапсуляции кода Groovy из файла `RuleEngineImpl.groovy`. Код Groovy помещен внутрь дескриптора `CDATA`, что предотвращает его синтаксический разбор анализатором XML. Теперь запустите пример процессора правил еще раз. Вы заметите, что он работает, как и ранее, но в данном случае мы встроили код Groovy прямо в определение бина, а не сохранили его во внешнем файле. Мы намеренно использовали код из файла `RuleEngineImpl.groovy`, чтобы продемонстрировать, насколько громоздким может стать приложение при встраивании больших объемов кода.

# Резюме

В этой главе была раскрыта тема применения языков сценариев в Java-приложениях и описана поддержка платформой Spring Framework языков сценариев, которые помогают добавлять к приложению динамическое поведение.

Первым делом, мы обсудили спецификацию JSR-223, которая была встроена в Java 6 и поддерживает выполнение сценариев на языке JavaScript. Затем мы представили Groovy — популярный в сообществе Java-разработчиков язык сценариев. Мы также показали некоторые его основные возможности в сравнении с традиционным языком Java.

Наконец, мы рассмотрели поддержку языков сценариев в Spring Framework, продемонстрировав ее в действии за счет проектирования и реализации очень простого процессора правил, использующего поддержку DSL в Groovy. Мы также объяснили, как модифицировать правило и заставить Spring Framework автоматически обнаруживать изменения с применением средства обновляемых бинов, что не требует компиляции, упаковки и развертывания приложения. Кроме того, мы показали, каким образом встраивать код Groovy напрямую в конфигурационный файл для определения кода реализации бина.

# ГЛАВА 15

# Мониторинг приложений Spring

Типичное JEE-приложение содержит несколько уровней и компонентов, таких как уровень презентаций, уровень обслуживания, уровень постоянства и источник данных серверной части. На протяжении стадии разработки либо после развертывания приложения в среде контроля качества (quality assurance — QA) или в производственной среде нам нужно удостовериться, что приложение находится в работоспособном состоянии и не содержит никаких потенциальных проблем либо узких мест.

В Java-приложении существует много аспектов, которые могут вызвать проблемы с производительностью или перегрузкой серверных ресурсов (наподобие центрального процессора, памяти и подсистемы ввода-вывода). В качестве примеров следует упомянуть неэффективный Java-код, утечку памяти (когда в Java-коде выделяются новые объекты без освобождения ссылок на них, что препятствует очистке памяти JVM-машиной во время процесса сборки мусора), параметры JVM, параметры пула потоков, конфигурация источников данных (например, количество разрешенных одновременных подключений к базе данных), настройка базы данных и наличие длительно выполняющихся SQL-запросов.

Следовательно, нам необходимо понимать поведение приложения во время выполнения и идентифицировать любые потенциально узкие места либо имеющиеся проблемы. В мире Java доступно множество инструментов, которые помогают проводить мониторинг поведения JEE-приложений во время выполнения. Большинство из них построены на основе технологии JMX (Java Management Extensions — управляющие расширения Java). В этой главе мы представим несколько общих приемов мониторинга JEE-приложений, основанных на Spring. В частности, в главе будут рассматриваться следующие темы.

- **Поддержка JMX в Spring.** Мы обсудим комплексную поддержку JMX в Spring и покажем, как сделать бины Spring открытыми для мониторинга с помощью инструментов JMX. В качестве инструмента мониторинга приложений мы будем использовать VisualVM (`http://visualvm.java.net/index.html`).
- **Мониторинг статистики Hibernate.** Многие макеты, в том числе Hibernate, предоставляют поддерживающие классы и инфраструктуру для получения доступа к рабочему состоянию и метрикам производительности с применением JMX. Мы покажем, как включить мониторинг посредством JMX для часто используемых компонентов JEE-приложений Spring.

Имейте в виду, что настоящая глава не предназначена служить введением в JMX, поэтому предполагается наличие у вас базового понимания этой технологии. За детальными сведениями по данной теме обращайтесь к онлайновому ресурсу Oracle по адресу `www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`.

# Поддержка JMX в Spring

В технологии JMX классы, которые открыты для мониторинга и управления, называются *управляемыми бинами* (managed bean), или *MBean-компонентами*. Платформа Spring Framework поддерживает несколько механизмов для открытия MBean-компонентов. Основное внимание в этой главе будет уделено открытию бинов Spring (разработанных в виде простых объектов POJO) как MBean-компонентов для проведения мониторинга JMX.

В последующих разделах мы обсудим процедуру открытия бина, содержащего связанную с приложением статистику, в качестве MBean-компонента для мониторинга JMX. Темы включают реализацию бина Spring, открытие бина Spring как MBean-компонента в `ApplicationContext` и применение инструмента VisualVM для мониторинга MBean-компонента.

## Экспортирование бина Spring в JMX

В качестве примера мы будем использовать веб-службу REST, построенную в главе 12. Еще раз просмотрите код примера приложения в указанной главе или обратитесь непосредственно к коду примеров для этой книги, где доступен исходный код, от которого мы и будем отталкиваться. За счет добавления JMX мы хотим открыть количество контактов в базе данных в целях мониторинга JMX. Итак, давайте реализуем интерфейс и класс, как показано в листингах 15.1 и 15.2.

**Листинг 15.1. Интерфейс `AppStatistics`**

```
package com.apress.prospring4.ch15;
public interface AppStatistics {
    int getTotalContactCount();
}
```

**Листинг 15.2. Класс `AppStatisticsImpl`**

```
package com.apress.prospring4.ch15;

import org.springframework.beans.factory.annotation.Autowired;

public class AppStatisticsImpl implements AppStatistics {
    @Autowired
    private ContactService contactService;

    @Override
    public int getTotalContactCount() {
        return contactService.findAll().size();
    }
}
```

В этом примере определен метод для извлечения общего количества записей контактов в базе данных. Чтобы открыть бин Spring для JMX, понадобится добавить конфигурацию в `ApplicationContext`. В листинге 15.3 показан код, необходимый для открытия бина Spring для JMX в конфигурационном файле (`rest-context.xml`).

**Листинг 15.3. Открытие бина Spring для JMX**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:mvc="http://www.springframework.org/schema/mvc"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc.xsd">

   <mvc:annotation-driven>
      <mvc:message-converters>
         <bean class="org.springframework.http.converter.json.
MappingJackson2HttpMessageConverter"/>
         <bean class="org.springframework.http.converter.xml.
MarshallingHttpMessageConverter">
            <property name="marshaller" ref="castorMarshaller"/>
            <property name="unmarshaller" ref="castorMarshaller"/>
         </bean>
      </mvc:message-converters>
   </mvc:annotation-driven>

   <context:component-scan base-package="com.apress.prospring4.ch15"/>

   <bean id="castorMarshaller"
         class="org.springframework.oxm.castor.CastorMarshaller">
      <property name="mappingLocation"
                value="classpath:META-INF/spring/oxm-mapping.xml"/>
   </bean>

   <bean id="appStatisticsBean"
         class="com.apress.prospring4.ch15.AppStatisticsImpl"/>
   <bean id="jmxExporter"
         class="org.springframework.jmx.export.MBeanExporter">
      <property name="beans">
         <map>
            <entry key="bean:name=ProSpring4ContactApp"
                   value-ref="appStatisticsBean"/>
         </map>
      </property>
   </bean>
</beans>
```

Во-первых, в коде объявляется бин для POJO со статистикой (`AppStatisticsImpl`), который мы хотим открыть. Во-вторых, объявляется бин `jmxExporter` с классом реализации `MBeanExporter`.

Класс `MBeanExporter` — это ключевой класс в рамках поддержки JMX платформой Spring Framework. Он отвечает за регистрацию бинов Spring на MBean-сервере JMX (сервер, реализующий JDK-интерфейс `javax.management.MBeanServer`, который существует в большинстве распространенных веб- и JEE-контейнеров, таких как Tomcat и WebSphere). При открытии бина Spring как MBean-компонента среда Spring попытается найти на сервере выполняющийся экземпляр `MBeanServer` и с его помощью зарегистрировать этот MBean-компонент. Например, для сервера Tomcat экземпляр `MBeanServer` будет создаваться автоматически, поэтому никакого дополнительного конфигурирования не требуется.

Внутри бина `jmxExporter` свойство `beans` определяет бины Spring, которые необходимо открыть. Это свойство типа `Map`, поэтому в нем можно указать любое количество MBean-компонентов. В нашем случае нужно открыть бин `appStatisticsBean`, содержащий информацию о приложении контактов, которую мы хотим показать администраторам. В определении MBean-компонента ключ будет использоваться как `ObjectName` (класс `javax.management.ObjectName` в JDK) для бина Spring, на который ссылается соответствующее значение записи. В приведенной выше конфигурации MBean-компонент `appStatisticsBean` будет открыт с `ObjectName` вида `bean:name=Prospring4ContactApp`. По умолчанию все открытые свойства бина видны как атрибуты, а все открытые методы — как операции.

Теперь MBean-компонент доступен для мониторинга через JMX. А теперь займемся настройкой VisualVM и применением клиента JMX в целях мониторинга.

## Настройка VisualVM для мониторинга JMX

VisualVM — очень полезный инструмент, помогающий проводить мониторинг разнообразных аспектов Java-приложений. Это бесплатный инструмент, который находится в папке `bin` внутри папки установки JDK. На веб-сайте проекта также доступна для загрузки автономная версия VisualVM (`http://visualvm.java.net/download.html`). В этой главе мы будем пользоваться автономной версией, которой на момент написания книги была 1.3.8.

Для поддержки различных функций мониторинга в VisualVM применяется система подключаемых модулей. Чтобы обеспечить мониторинг MBean-компонентов Java-приложений, потребуется установить подключаемый модуль VisualVM-MBeans. Выполните следующие шаги.

1. Выберите в меню VisualVM пункт Tools⇨Plugins (Сервис⇨Подключаемые модули).
2. В открывшемся диалоговом окне Plugins (Подключаемые модули) перейдите на вкладку Available Plugins (Доступные подключаемые модули).
3. Щелкните на кнопке Check for Newest (Проверить наличие самых новых).
4. Отметьте флажок возле подключаемого модуля VisualVM-MBeans и щелкните на кнопке Install (Установить).

По завершении установки удостоверьтесь, что сервер Tomcat функционирует, а пример приложения запущен. Затем в представлении Applications (Приложения) в левой части экрана VisualVM вы должны увидеть выполняющийся процесс Tomcat.

По умолчанию VisualVM ищет Java-приложения, которые функционируют под управлением платформы JDK. Двойной щелчок на желаемом узле приводит к отображению экрана мониторинга.

После установки подключаемого модуля VisualVM-MBeans появляется вкладка MBeans (MBean-компоненты). Перейдя на нее, можно просмотреть доступные MBean-компоненты. Слева должен быть виден узел под названием bean (бин). Если развернуть его, появится MBean-компонент `Prospring4ContactApp`, который был открыт.

В правой части показан метод, реализованный в бине, с атрибутом `TotalContactCount` (который был автоматически выведен методом `getTotal ContactCount()` внутри бина). Значением является 3, что соответствует количеству записей, добавленных в базу данных при запуске приложения. В обычном приложении это число изменялось бы на основе количества контактов, которые добавляются во время выполнения приложения.

## Мониторинг статистики Hibernate

В Hibernate также поддерживается управление и открытие для JMX метрик, связанных с постоянством. Чтобы сделать это доступным, понадобится добавить в конфигурацию JPA (файл `datasource-tx-jpa.xml`) два дополнительных свойства Hibernate, как показано в листинге 15.4.

**Листинг 15.4. Включение статистики Hibernate**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:jdbc="http://www.springframework.org/schema/jdbc"
      xmlns:jpa="http://www.springframework.org/schema/data/jpa"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/jdbc
       http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
       http://www.springframework.org/schema/data/jpa
       http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx.xsd">

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script location="classpath:META-INF/config/schema.sql"/>
      <jdbc:script location="classpath:META-INF/config/test-data.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionManager"
         class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="emf"/>
   </bean>

   <tx:annotation-driven transaction-manager="transactionManager" />

   <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
```

```
      <property name="jpaVendorAdapter">
         <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
      </property>
      <property name="packagesToScan" value="com.apress.prospring4.ch15"/>
      <property name="jpaProperties">
         <props>
           <prop key="hibernate.dialect">org.hibernate.dialect.H2Dialect
           </prop>
           <prop key="hibernate.max_fetch_depth">3</prop>
           <prop key="hibernate.jdbc.fetch_size">50</prop>
           <prop key="hibernate.jdbc.batch_size">10</prop>
           <prop key="hibernate.show_sql">true</prop>
           <!-- Свойства статистики Hibernate -->
           <prop key="hibernate.generate_statistics">true</prop>
           <prop key="hibernate.session_factory_name">sessionFactory</prop>
         </props>
      </property>
   </bean>

   <context:annotation-config/>

   <jpa:repositories base-package="com.apress.prospring4.ch15"
                     entity-manager-factory-ref="emf"
                     transaction-manager-ref="transactionManager"/>
</beans>
```

Свойство `hibernate.generate_statistics` инструктирует Hibernate о необходимости генерации статистики для его поставщика постоянства JPA, тогда как свойство `hibernate.session_factory_name` определяет имя фабрики сеансов, требуемое MBean-компонентом статистики Hibernate. Оба свойства находятся ниже комментария "Свойства статистики Hibernate".

Наконец, этот MBean-компонент нужно добавить в конфигурацию `MBeanExporter` из Spring. В листинге 15.5 приведена модифицированная конфигурация, которая была создана ранее в файле `rest-context.xml`.

**Листинг 15.5. MBean-компонент для статистики Hibernate**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:mvc="http://www.springframework.org/schema/mvc"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc.xsd">

   <mvc:annotation-driven>
      <mvc:message-converters>
         <bean class="org.springframework.http.converter.json.
MappingJackson2HttpMessageConverter"/>
```

```
        <bean class="org.springframework.http.converter.xml.
MarshallingHttpMessageConverter">
            <property name="marshaller" ref="castorMarshaller"/>
            <property name="unmarshaller" ref="castorMarshaller"/>
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>

<context:component-scan base-package="com.apress.prospring4.ch15"/>

<bean id="castorMarshaller"
      class="org.springframework.oxm.castor.CastorMarshaller">
    <property name="mappingLocation"
              value="classpath:META-INF/spring/oxm-mapping.xml"/>
</bean>
<bean id="appStatisticsBean"
      class="com.apress.prospring4.ch15.AppStatisticsImpl"/>
<bean id="jmxExporter"
      class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
        <map>
            <entry key="bean:name=ProSpring4ContactApp"
                   value-ref="appStatisticsBean"/>
            <entry key="bean:name=Prospring4ContactApp-hibernate"
                   value-ref="statisticsBean"/>
        </map>
    </property>
</bean>
<bean id="statisticsBean" class="org.hibernate.jmx.StatisticsService">
    <property name="statisticsEnabled" value="true"/>
    <property name="sessionFactoryJNDIName" value="sessionFactory"/>
</bean>
</beans>
```

Здесь объявлен новый бин `statisticsBean` с классом `StatisticsService` из Hibernate в качестве реализации. С помощью этого класса Hibernate поддерживает открытие статистики для JMX.

Обратите внимание на свойство `sessionFactoryJNDIName`, которое должно соответствовать свойству, определенному в листинге 15.4 (`hibernate.session_factory_name`). Внутри бина `jmxExporter` объявлен еще один бин с `ObjectName` вида `bean:name=Prospring4ContactApp-hibernate`, который ссылается на бин `statisticsBean`.

Теперь статистика Hibernate включена и доступна через JMX. Перезагрузите приложение, и после обновления VisualVM можно будет видеть MBean-компонент статистики Hibernate. Щелчок на узле `ProSpring3ContactApp-hibernate` приводит к отображению подробной статистики с правой стороны. На полях с информацией, не относящейся к элементарным типам Java (например, `List`), можно щелкать, чтобы раскрывать их для просмотра содержимого.

В VisualVM можно видеть множество других метрик, в том числе `EntityNames`, `SessionOpenCount`, `SecondCloseCount` и `QueryExecutionMaxTime`. Их значения полезны для понимания поведения постоянства внутри приложения и могут помочь при поиске и устранении неполадок, а также при настройке производительности.

# Резюме

В этой главе были раскрыты высокоуровневые темы, связанные с мониторингом JEE-приложений на основе на Spring.

Прежде всего, мы обсудили поддержку JMX в Spring (JMX — это стандарт для мониторинга Java-приложений). Затем мы продемонстрировали реализацию специальных MBean-компонентов для открытия информации, связанной с приложением, а также открытие статистических данных по общим компонентам, таким как Hibernate.

# ГЛАВА 16

# Разработка веб-приложений в Spring

В корпоративном приложении уровень презентаций существенно влияет на степень принятия приложения пользователями. Уровень презентаций — это своего рода "парадный вход" в приложение. Он позволяет пользователям выполнять бизнес-функции, предоставляемые приложением, а также обеспечивает визуальное представление информации, которой управляет приложение. Эффективность пользовательского интерфейса вносит значительный вклад в успех приложения в целом.

Из-за бурного роста Интернета (особенно в наши дни, учитывая появление различных типов мобильных устройств) разработка уровня презентаций в приложении становится довольно непростой задачей. Ниже приведено несколько основных соображений, которые должны приниматься во внимание во время разработки веб-приложений.

- **Производительность.** Производительность всегда была главным требованием к веб-приложению. Если пользователи после выбора какой-то функции или щелчка на ссылке вынуждены слишком долго ожидать выполнения (в мире Интернета уже 3 секунды подобны столетию), то они определенно не будет удовлетворены таким приложением.
- **Дружественность к пользователю.** Приложение должно быть простым в использовании и навигации с очевидными инструкциями, не вводящими пользователя в заблуждение.
- **Интерактивность и широкие возможности.** Пользовательский интерфейс должен быть интерактивным и отзывчивым. Кроме того, уровень презентаций должен быть обогащенным в визуальном смысле, включая диаграммы, интерфейс в виде управляющих панелей и т.п.
- **Доступность.** В наши дни пользователи требуют, чтобы приложение было доступным из любого места на любом устройстве. На работе они будут использовать для доступа к приложению настольные компьютеры. В дороге пользователи будут осуществлять доступ к приложению из разнообразных мобильных устройств, таких как ноутбуки, планшеты и смартфоны.

Разработка веб-приложения, удовлетворяющего указанным выше требованиям, не является простой, однако эти требования рассматриваются бизнес-пользователями как обязательные. К счастью, для удовлетворения этих потребностей также было создано множество новых технологий и инфраструктур. В последнее время многие платформы и библиотеки для построения веб-приложений, среди которых Spring MVC, Struts, Tapestry, Java Server Faces (JSF), Google Web Toolkit (GWT), jQuery и Dojo (этот список далеко не полон), предлагают инструменты и развитые библиотеки компонентов, помогающие строить интерактивные пользовательские интерфейсы для веб-приложений. Кроме того, многие инфраструктуры предоставляют инструменты или соответствующие библиотеки виджетов (графических компонентов), которые предназначены для мобильных устройств, включая смартфоны и планшеты. Развитие стандартов HTML5 и CSS3 вместе с их поддержкой большинством веб-браузеров и производителей мобильных устройств также способствует упрощению разработки веб-приложений, которые должны быть доступными откуда угодно и на любом устройстве.

Платформа Spring предлагает широкую и интенсивную поддержку разработки веб-приложений. Модуль Spring MVC обеспечивает надежную инфраструктуру и архитектуру MVC (Model View Controller — модель-представление-контроллер), предназначенную для построения веб-приложений. Вместе с модулем Spring MVC можно применять разнообразные технологии представлений (например, JSP или Velocity). Вдобавок Spring MVC интегрируется со многими известными веб-платформами и наборами инструментов (в том числе Struts и GWT). Другие проекты Spring помогают решать специфические потребности веб-приложений. Например, проект Spring MVC, скомбинированный с проектом Spring Web Flow и его модулем Spring Faces, предлагает развернутую поддержку разработки веб-приложений со сложными потоками и использованием JSF в качестве технологии представлений. Проще говоря, для построения уровня презентаций доступно очень много вариантов.

Основное внимание в этой главе сосредоточено на Spring MVC; в ней будет показано, как задействовать мощные средства Spring MVC для разработки высокопроизводительных веб-приложений. В частности, в главе рассматриваются следующие темы.

- **Spring MVC.** Мы обсудим основные концепции шаблона MVC и предоставим введение в Spring MVC. Мы объясним ключевые концепции Spring MVC, включая иерархию `WebApplicationContext` и жизненный цикл обработки запроса.
- **Интернационализация, локаль и оформление темами.** В Spring MVC предлагается расширенная поддержка для удовлетворения общих требований к веб-приложениям, включая интернационализацию (i18n), локаль и оформление темами. Мы обсудим применение Spring MVC для разработки веб-приложений, которые поддерживают эти требования.
- **Поддержка технологий представления и Ajax.** В Spring MVC поддерживается множество технологий представлений. В этой главе мы сосредоточимся на использовании JavaServer Pages (JSP) и Tiles в качестве представлений для веб-приложения. Для реализации расширенных возможностей вместе с JSP будет применяться JavaScript. Доступно немало выдающихся и популярных JavaScript-библиотек вроде jQuery и Dojo. В этой главе мы покажем, как работать с jQuery — подпроектом библиотеки jQuery UI, которая поддерживает построение интерактивных веб-приложений.

- **Поддержка разбиения на страницы и загрузки файлов.** При разработке примеров в этой главе мы обсудим использование Spring Data JPA и компонента пользовательского интерфейса jQuery для обеспечения поддержки разбиения на страницы, когда производится просмотр данных с помощью экранной сетки. Кроме того, будет показано, как в Spring MVC реализовать загрузку файлов. Вместо интеграции с Apache Commons FileUpload мы рассмотрим, как применять для загрузки файлов платформу Spring MVC со встроенной в контейнер Servlet 3.1 поддержкой множественного содержимого.
- **Безопасность.** Безопасность — это крупная тема в области построения веб-приложений. Мы обсудим, как использовать Spring Security для защиты приложения и обработки входа и выхода из него.

# Реализация уровня обслуживания для примеров

На уровне обслуживания, предназначенном для этой главы, мы по-прежнему будем пользоваться приложением контактов. В этом разделе мы обсудим модель данных и реализацию уровня обслуживания, которые будут применяться далее в главе.

## Использование модели данных для примеров

Модель данных для примеров этой главы очень проста и содержит единственную таблицу `CONTACT`, предназначенную для хранения информации о контактах. В листинге 16.1 показан сценарий для создания схемы базы данных (`schema.sql`).

**Листинг 16.1. Сценарий для создания схемы базы данных**

```
DROP TABLE IF EXISTS CONTACT;

CREATE TABLE CONTACT (
      ID INT NOT NULL AUTO_INCREMENT
    , FIRST_NAME VARCHAR(60) NOT NULL
    , LAST_NAME VARCHAR(40) NOT NULL
    , BIRTH_DATE DATE
    , DESCRIPTION VARCHAR(2000)
    , PHOTO BLOB
    , VERSION INT NOT NULL DEFAULT 0
    , UNIQUE UQ_CONTACT_1 (FIRST_NAME, LAST_NAME)
    , PRIMARY KEY (ID)
);
```

Как видите, таблица `CONTACT` содержит лишь несколько базовых полей контактной информации. Единственным моментом, стоящим упоминания, является столбец `PHOTO` с типом данных `BLOB` (binary large object — большой двоичный объект), который будет использоваться для хранения фотографии контакта с применением загрузки файлов. В листинге 16.2 приведен код сценария для наполнения таблицы тестовыми данными (`test-data.sql`).

**Листинг 16.2. Сценарий для наполнения тестовыми данными**

```
insert into contact (first_name, last_name, birth_date)
  values ('Chris', 'Schaefer', '1981-05-03');
```

```
insert into contact (first_name, last_name, birth_date)
       values ('Scott', 'Tiger', '1990-11-02');
insert into contact (first_name, last_name, birth_date)
       values ('John', 'Smith', '1964-02-28');
insert into contact (first_name, last_name, birth_date)
       values ('Peter', 'Jackson', '1944-1-10');
insert into contact (first_name, last_name, birth_date)
       values ('Jacky', 'Chan', '1955-10-31');
insert into contact (first_name, last_name, birth_date)
       values ('Susan', 'Boyle', '1970-05-06');
insert into contact (first_name, last_name, birth_date)
       values ('Tinner', 'Turner', '1967-04-30');
insert into contact (first_name, last_name, birth_date)
       values ('Lotus', 'Notes', '1990-02-28');
insert into contact (first_name, last_name, birth_date)
       values ('Henry', 'Dickson', '1997-06-30');
insert into contact (first_name, last_name, birth_date)
       values ('Sam', 'Davis', '2001-01-31');
insert into contact (first_name, last_name, birth_date)
       values ('Max', 'Beckham', '2002-02-01');
insert into contact (first_name, last_name, birth_date)
       values ('Paul', 'Simon', '2002-02-28');
```

На этот раз объем тестовых данных больше, потому что планируется демонстрация поддержки разбиения на страницы.

## Реализация и конфигурирование интерфейса `ContactService`

В последующих разделах мы обсудим реализацию интерфейса `ContactService` с использованием JPA 2, Spring Data JPA и Hibernate в качестве поставщика постоянства. Затем мы покажем, как сконфигурировать уровень обслуживания в проекте Spring.

### *Реализация интерфейса `ContactService`*

В примерах этой главы мы сделаем доступными уровню презентаций службы для разнообразных операций над контактной информацией. Первым делом необходимо создать сущностный класс `Contact`, который приведен в листинге 16.3.

**Листинг 16.3. Сущностный класс `Contact`**

```
package com.apress.prospring4.ch16;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;

import org.hibernate.annotations.Type;
import org.joda.time.DateTime;
import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.format.annotation.DateTimeFormat.ISO;

import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
```

```
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Lob;
import javax.persistence.Table;
import javax.persistence.Transient;
import javax.persistence.Version;

@Entity
@Table(name = "contact")
public class Contact implements Serializable {
   private Long id;
   private int version;
   private String firstName;
   private String lastName;
   private DateTime birthDate;
   private String description;
   private byte[] photo;

   @Id
   @GeneratedValue(strategy = IDENTITY)
   @Column(name = "ID")
   public Long getId() {
      return id;
   }

   public void setId(Long id) {
      this.id = id;
   }

   @Version
   @Column(name = "VERSION")
   public int getVersion() {
      return version;
   }

   public void setVersion(int version) {
      this.version = version;
   }

   @Column(name = "FIRST_NAME")
   public String getFirstName() {
      return firstName;
   }

   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }

   @Column(name = "LAST_NAME")
   public String getLastName() {
      return lastName;
   }

   public void setLastName(String lastName) {
      this.lastName = lastName;
   }

   @Column(name = "BIRTH_DATE")
   @Type(type="org.jadira.usertype.dateandtime.joda.PersistentDateTime")
```

```
    @DateTimeFormat(iso=ISO.DATE)
    public DateTime getBirthDate() {
        return birthDate;
    }
    public void setBirthDate(DateTime birthDate) {
        this.birthDate = birthDate;
    }
    @Column(name = "DESCRIPTION")
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }

    @Basic(fetch= FetchType.LAZY)
    @Lob
    @Column(name = "PHOTO")
    public byte[] getPhoto() {
        return photo;
    }
    public void setPhoto(byte[] photo) {
        this.photo = photo;
    }
    @Transient
    public String getBirthDateString() {
        String birthDateString = "";
        if (birthDate != null)
            birthDateString = org.joda.time.format.DateTimeFormat
                    .forPattern("yyyy-MM-dd").print(birthDate);
        return birthDateString;
    }
    @Override
    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
               + ", Last name: " + lastName +·", Birthday: " + birthDate
               + ", Description: " + description;
    }
}
```

Как показано в листинге 16.3, в классе применяются стандартные аннотации. Мы также используем класс `DateTime` из Joda-Time для атрибута `birthDate`. Кроме того, обратите внимание на следующие моменты.

- Добавлено новое переходное свойство (за счет применения аннотации `@Transient` к методу извлечения) по имени `birthDateString`, которое будет использоваться для представления пользовательского интерфейса в последующих примерах.

- Для атрибута `photo` в качестве типа данных Java применяется байтовый массив, который соответствует типу данных `BLOB` в СУРБД. Вдобавок метод извлечения аннотирован посредством `@Lob` и `@Basic(fetch=FetchType.LAZY)`. Первая аннотация указывает поставщику JPA, что это столбец большого объекта, а вторая — что атрибут должен извлекаться отложенным (ленивым) образом, чтобы не оказывать влияния на производительность, когда загружается класс, не требующий фотографии.

Давайте продолжим разработку уровня обслуживания. В листинге 16.4 показан интерфейс `ContactService` со службами, которые должны быть отображены.

**Листинг 16.4. Интерфейс `ContactService`**

```
package com.apress.prospring4;

import java.util.List;

public interface ContactService {
    List<Contact> findAll();
    Contact findById(Long id);
    Contact save(Contact contact);
}
```

Методы этого интерфейса самоочевидны. Поскольку мы будем пользоваться поддержкой репозитория Spring Data JPA, то реализуем интерфейс `ContactRepository` (листинг 16.5).

**Листинг 16.5. Интерфейс `ContactRepository`**

```
package com.apress.prospring4.ch16;

import org.springframework.data.repository.CrudRepository;

public interface ContactRepository extends CrudRepository<Contact, Long> {
}
```

В листинге 16.6 приведен класс реализации интерфейса `ContactService`.

**Листинг 16.6. Класс `ContactServiceImpl`**

```
package com.apress.prospring4.ch16;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.google.common.collect.Lists;

@Repository
@Transactional
@Service("contactService")
public class ContactServiceImpl implements ContactService {
    private ContactRepository contactRepository;

    @Override
```

```
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return Lists.newArrayList(contactRepository.findAll());
    }
    @Override
    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        return contactRepository.findOne(id);
    }
    @Override
    public Contact save(Contact contact) {
        return contactRepository.save(contact);
    }
    @Autowired
    public void setContactRepository(ContactRepository contactRepository) {
        this.contactRepository = contactRepository;
    }
}
```

Реализация в основном завершена, и дальнейший шаг заключается в конфигурировании службы в `ApplicationContext` проекта веб-приложения, как обсуждается в следующем разделе.

### *Конфигурирование уровня обслуживания*

Для настройки уровня обслуживания в проекте Spring MVC мы создаем отдельный конфигурационный файл по имени `datasource-tx-jpa.xml`. Содержимое этого файла показано в листинге 16.7.

**Листинг 16.7. Конфигурационный файл `datasource-tx-jpa.xml`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:jdbc="http://www.springframework.org/schema/jdbc"
      xmlns:jpa="http://www.springframework.org/schema/data/jpa"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/jdbc
       http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
       http://www.springframework.org/schema/data/jpa
       http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx.xsd">
    <jdbc:embedded-database id="dataSource" type="H2">
        <jdbc:script location="classpath:META-INF/sql/schema.sql"/>
        <jdbc:script location="classpath:META-INF/sql/test-data.sql"/>
    </jdbc:embedded-database>
```

```
    <bean id="transactionManager"
          class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="emf"/>
    </bean>

    <tx:annotation-driven transaction-manager="transactionManager" />

    <bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter">
            <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
        </property>
        <property name="packagesToScan" value="com.apress.prospring4"/>
        <property name="jpaProperties">
            <props>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.H2Dialect
                </prop>
                <prop key="hibernate.max_fetch_depth">3</prop>
                <prop key="hibernate.jdbc.fetch_size">50</prop>
                <prop key="hibernate.jdbc.batch_size">10</prop>
                <prop key="hibernate.show_sql">true</prop>
            </props>
        </property>
    </bean>

    <context:annotation-config/>

    <jpa:repositories base-package="com.apress.prospring4.ch16"
                      entity-manager-factory-ref="emf"
                      transaction-manager-ref="transactionManager"/>
</beans>
```

Теперь нужно импортировать конфигурацию в корневой контекст `WebApplication Context`. Содержимое конфигурационного файла приведено в листинге 16.8.

**Листинг 16.8. Файл `root-context.xml`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

    <import resource="classpath:META-INF/spring/datasource-tx-jpa.xml" />

    <context:component-scan base-package="com.apress.prospring4.ch16" />
</beans>
```

Сначала в этот конфигурационный файл добавляется пространство имен `context`. Затем файл `datasource-tx-jpa.xml` импортируется в `WebApplicationContext`

и, наконец, платформе Spring указывается на необходимость сканирования заданного пакета в поисках бинов Spring.

Итак, уровень обслуживания завершен и готов к отображению и использованию удаленными клиентами.

# Введение в MVC и Spring MVC

Перед тем, как заняться реализацией уровня презентаций, давайте рассмотрим некоторые основные концепции MVC как шаблона проектирования веб-приложений и выясним, каким образом Spring MVC обеспечивает развитую поддержку в этой области.

В последующих разделах мы по очереди обсудим эти высокоуровневые концепции. Во-первых, мы предложим краткое введение в MVC. Во-вторых, мы дадим высокоуровневое представление инфраструктуры Spring MVC и ее иерархии `WebApplicationContext`. В-третьих, мы рассмотрим жизненный цикл обработки запроса в Spring MVC.

## Введение в MVC

MVC — это шаблон, который часто используется при реализации уровня презентаций в приложении. Главный принцип шаблона MVC состоит в определении архитектуры с четкими ответственностями для каждого компонента. В шаблоне MVC присутствуют три участника.

- **Модель.** Модель представляет бизнес-данные, а также "состояние" приложения внутри контекста конкретного пользователя. Например, на веб-сайте электронной коммерции модель будет включать информацию профиля пользователя, данные корзины для покупок и данные заказа, если пользователи приобретают товары на этом сайте.
- **Представление.** Представление отображает данные пользователю в желаемом формате, поддерживает взаимодействие с пользователями и выполняет проверку достоверности клиентской стороны, интернационализацию, оформление стилями и т.д.
- **Контроллер.** Контроллер обрабатывает запросы к действиям, осуществляемые пользователями в пользовательском интерфейсе, взаимодействуя с уровнем обслуживания, обновляя модель и направляя пользователей на соответствующие представления в зависимости от результатов выполнения.

В связи с появлением веб-приложений на основе Ajax шаблон MVC был расширен для предоставления более отзывчивого и развитого пользовательского интерфейса. Например, когда применяется JavaScript, представление может "прослушивать" события или действия, предпринимаемые пользователем, и затем отправлять запрос `XMLHttpRequest` серверу. На стороне контроллера вместо возврата целого представления производится возврат низкоуровневых данных (скажем, в формате XML или JSON), после чего JavaScript-код осуществляет "частичное" обновление представления с применением полученных данных. Эта концепция изображена на рис. 16.1.

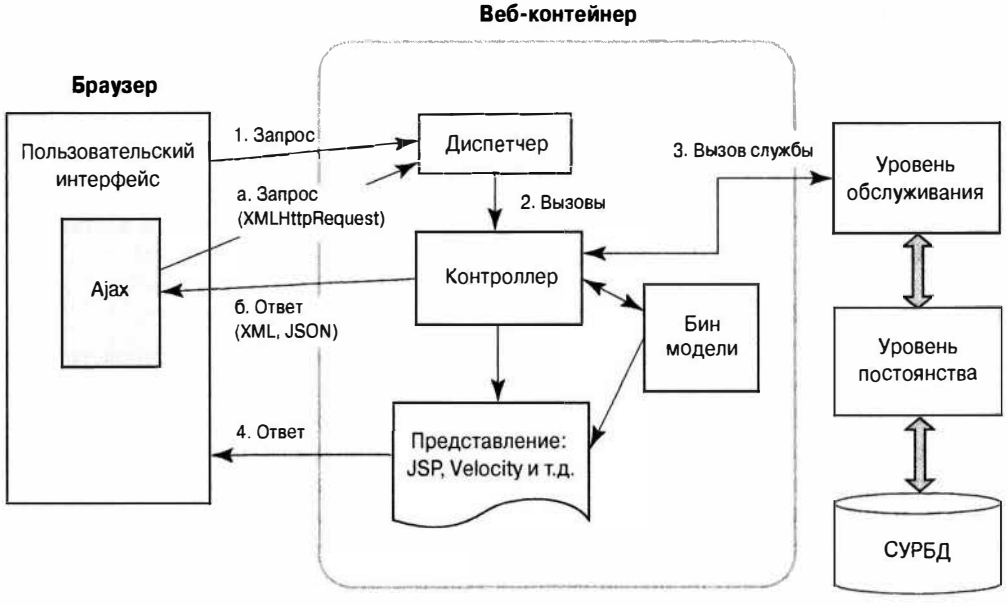


**Рис. 16.1.** Шаблон MVC

На рис. 16.1 проиллюстрирован распространенный шаблон веб-приложения, который можно трактовать как расширение традиционного шаблона MVC. Обычный запрос представления обрабатывается следующим образом.

1. *Запрос.* Запрос отправляется серверу. На стороне сервера в большинстве инфраструктур (например, Spring MVC или Struts) для обработки запроса предусмотрен диспетчер (в форме сервлета).
2. *Вызовы.* Диспетчер направляет запрос соответствующему контроллеру на основе информации HTTP-запроса и конфигурации веб-приложения.
3. *Вызов службы.* Контроллер взаимодействует с уровнем обслуживания.
4. *Ответ.* Контроллер обновляет модель и в зависимости от результата выполнения возвращает пользователю соответствующее представление.

Кроме того, внутри представления происходят вызовы Ajax. Например, предположим, что пользователь просматривает данные в экранной сетке. Когда он щелкает на ссылке для перехода к следующей странице, вместо обновления всей страницы будут инициированы перечисленные ниже действия.

1. *Запрос.* Запрос `XMLHttpRequest` подготавливается и отправляется серверу. Диспетчер направит запрос соответствующему контроллеру.
2. *Ответ.* Контроллер взаимодействует с уровнем обслуживания, и данные ответа будут сформатированы и отправлены браузеру. В этом случае никакие представления не задействованы. Браузер получает данные и выполняет частичное обновление существующего представления.

## Введение в Spring MVC

В Spring Framework модуль Spring MVC предоставляет развернутую поддержку шаблона MVC, а также других средств (например, оформление темами, интернационализацию, проверку достоверности, преобразование типов и форматирование), которые упрощают реализацию уровня презентаций.

В последующих разделах мы обсудим основные концепции Spring MVC, в том числе иерархию `WebApplicationContext` в Spring MVC, типичный жизненный цикл обработки запроса и конфигурирование.

### *Иерархия `WebApplicationContext` в Spring MVC*

В Spring MVC класс `DispatcherServlet` является центральным сервлетом, который получает запросы и направляет их соответствующим контроллерам. В приложении Spring MVC может существовать произвольное количество экземпляров `DispatcherServlet`, предназначенных для разных целей (например, для обработки запросов к пользовательскому интерфейсу и веб-службам REST).

Каждый экземпляр `DispatcherServlet` имеет собственную конфигурацию `WebApplicationContext`, которая определяет характеристики уровня сервлета, такие как контроллеры, поддерживающие сервлет, отображение обработчиков, распознавание представлений, интернационализация, оформление темами, проверка достоверности, преобразование типов и форматирование.

Помимо конфигураций `WebApplicationContext` уровня сервлетов, в Spring MVC также поддерживается корневая конфигурация `WebApplicationContext`, которая включает конфигурации уровня приложения, в том числе для источников данных серверной стороны, безопасности, уровней обслуживания и постоянства. Корневая конфигурация `WebApplicationContext` будет доступна всем конфигурациям `WebApplicationContext` уровня сервлетов.

Давайте рассмотрим пример. Предположим, что в приложении имеются два экземпляра `DispatcherServlet`. Один сервлет предназначен для поддержки пользовательского интерфейса (назовем его сервлетом приложения), а другой — для предоставления служб в форме веб-служб REST другим приложениям (назовем его сервлетом REST).

В Spring MVC мы определим корневую конфигурацию `WebApplicationContext` и конфигурации `WebApplicationContext` для двух `DispatcherServlet`. На рис. 16.2 показана иерархия `WebApplicationContext`, которая будет поддерживаться Spring MVC для описанного сценария.

### *Жизненный цикл запроса Spring MVC*

А теперь посмотрим, каким образом Spring MVC обрабатывает запрос. На рис. 16.3 показаны основные компоненты, вовлеченные в обработку запроса внутри Spring MVC.

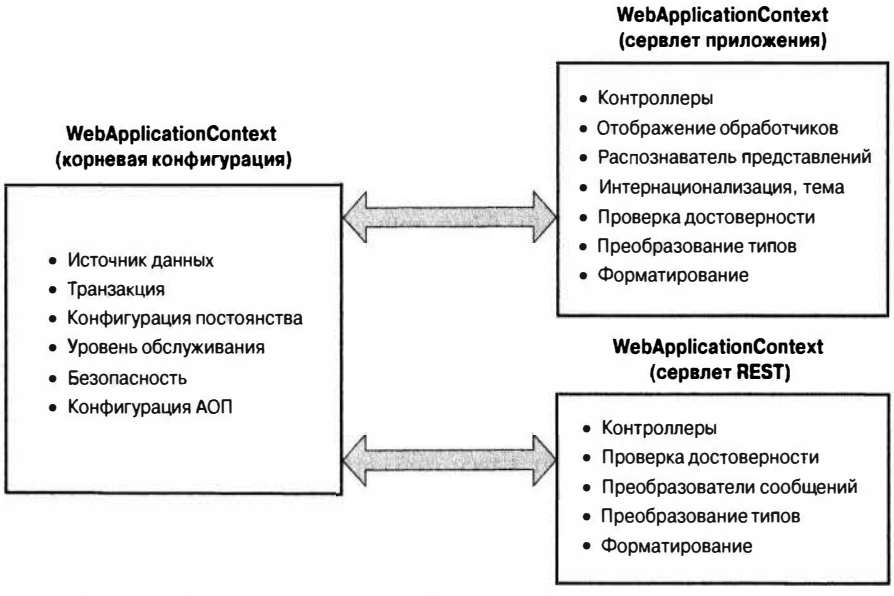


**Рис. 16.2.** Иерархия `WebApplicationContext` в Spring MVC

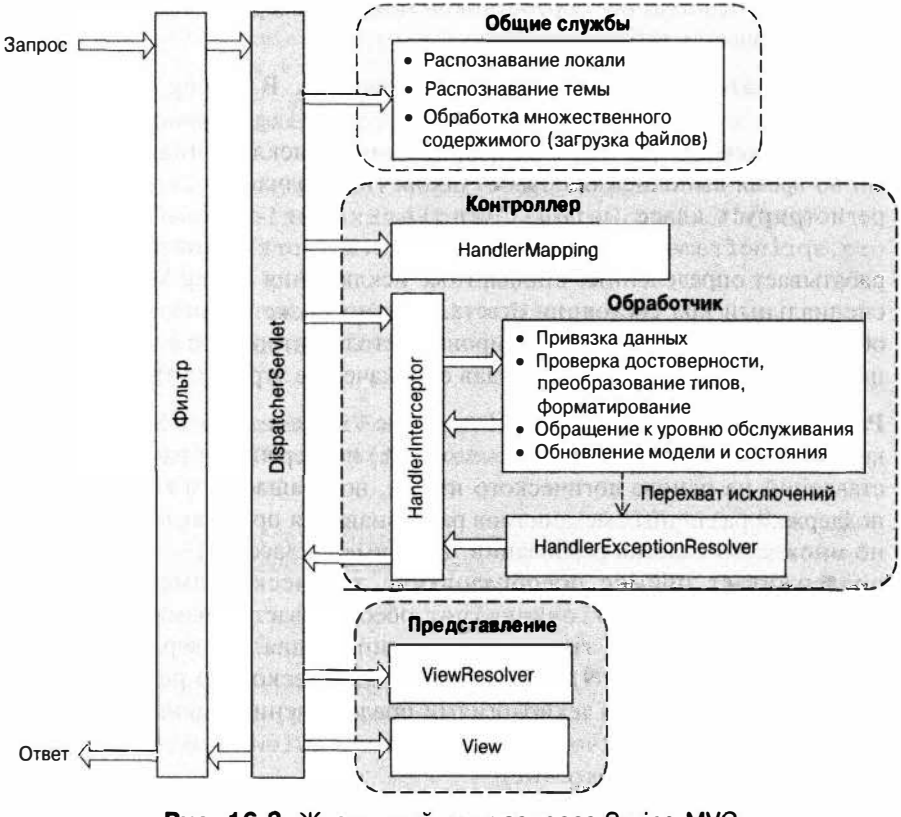


**Рис. 16.3.** Жизненный цикл запроса Spring MVC

Ниже перечислены основные компоненты с указанием их назначения.

- **Фильтр.** Фильтр применяется к каждому запросу. Часто используемые фильтры вместе с их предназначением описаны в следующем разделе.
- **Сервлет диспетчера.** Этот сервлет анализирует запросы и направляет их на обработку соответствующему контроллеру.
- **Общие службы.** Общие службы будут применяться к каждому запросу для предоставления поддержки интернационализации, оформления темами и загрузки файлов. Их конфигурация определена в `WebApplicationContext` сервлета диспетчера.
- **Отображение обработчиков.** Отображает запрос на обработчик (метод внутри класса контроллера Spring MVC). Начиная с версии Spring 2.5, в большинстве ситуаций конфигурация отображения не требуется, поскольку Spring MVC автоматически регистрирует класс `org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping`, который отображает обработчики на основе путей HTTP, выраженных через аннотацию `@RequestMapping` уровня типа или метода внутри классов контроллеров.
- **Перехватчик обработчиков.** В Spring MVC можно зарегистрировать перехватчики для обработчиков, чтобы реализовать общую проверку или логику. Например, перехватчик обработчиков может проверять, вызываются ли обработчики только в рабочие часы.
- **Распознаватель исключений в обработчиках.** В Spring MVC интерфейс `HandlerExceptionResolver` (из пакета `org.springframework.web.servlet`) предназначен для работы с непредвиденными исключениями, возникающими во время выполнения обработчиков. По умолчанию `DispatcherServlet` регистрирует класс `DefaultHandlerExceptionResolver` (из пакета `org.springframework.web.servlet.mvc.support`). Этот распознаватель обрабатывает определенные стандартные исключения Spring MVC, устанавливая специальный код состояния ответа. Можно также реализовать собственный обработчик исключений, аннотировав метод контроллера с помощью аннотации `@ExceptionHandler` и передав ей в качестве атрибута тип исключения.
- **Распознаватель представлений.** Интерфейс `ViewResolver` в Spring MVC (из пакета `org.springframework.web.servlet`) поддерживает распознавание представлений на основе логического имени, возвращаемого контроллером. Для поддержки различных механизмов распознавания представлений предусмотрено множество классов реализации. Например, класс `UrlBasedViewResolver` поддерживает прямое преобразование логических имен в URL. Класс `ContentNegotiatingViewResolver` обеспечивает динамическое распознавание представлений в зависимости от типа медиа, поддерживаемого клиентом (XML, PDF и JSON). Существует также несколько реализаций для интеграции с различными технологиями представлений, такими как FreeMarker (`FreeMarkerViewResolver`), Velocity (`VelocityViewResolver`) и JasperReports (`JasperReportsViewResolver`).

Приведенные описания касаются только нескольких распространенных обработчиков и распознавателей. За полным описанием обращайтесь в справочную документацию по Spring Framework.

### *Конфигурация Spring MVC*

Чтобы сделать доступным модуль Spring MVC внутри веб-приложения, требуется начальная конфигурация, особенно для дескриптора веб-развертывания `web.xml`. Начиная с версии Spring 3.1, стала доступной поддержка конфигурации на основе кода в веб-контейнере Servlet 3.0, которую мы обсудим в разделе "Поддержка конфигурации на основе кода для Servlet 3" далее в главе.

Чтобы настроить поддержку Spring MVC для веб-приложений, понадобится выполнить следующее конфигурирование в дескрипторе веб-развертывания:

- конфигурирование корневого контекста `WebApplicationContext`;
- конфигурирование фильтров сервлетов, требуемых Spring MVC;
- конфигурирование сервлетов диспетчеров внутри приложения.

Для начала давайте создадим файл `web.xml`, использующий Servlet 3.0. Его содержимое приведено в листинге 16.9.

**Листинг 16.9. Дескриптор веб-развертывания для Spring MVC**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
     http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/root-context.xml</param-value>
   </context-param>

   <filter>
     <filter-name>CharacterEncodingFilter</filter-name>
     <filter-class>org.springframework.web.filter.CharacterEncodingFilter
     </filter-class>
     <init-param>
          <param-name>encoding</param-name>
          <param-value>UTF-8</param-value>
     </init-param>
     <init-param>
          <param-name>forceEncoding</param-name>
          <param-value>true</param-value>
     </init-param>
   </filter>

   <filter>
     <filter-name>HttpMethodFilter</filter-name>
     <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter
     </filter-class>
   </filter>
```

```
    <filter>
        <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
        <filter-class>org.springframework.orm.jpa.support.
OpenEntityManagerInViewFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>CharacterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter-mapping>
        <filter-name>HttpMethodFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter-mapping>
        <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <listener>
      <listener-class>org.springframework.web.context.ContextLoaderListener
      </listener-class>
    </listener>

    <servlet>
        <servlet-name>appServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring/appServlet/servlet-context.xml
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>appServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

Ниже отмечены основные моменты, связанные с листингом 16.9.

- В дескрипторе `<web-app>` атрибут `version` и соответствующий URL служат для указания веб-контейнеру, что веб-приложение будет пользоваться Servlet 3.0.
- В дескрипторе `<context-param>` указан параметр `contextConfigLocation`, который определяет местоположение корневого файла конфигурации `WebApplicationContext` для Spring.
- Определено несколько фильтров сервлетов, предоставляемых Spring MVC, и все фильтры отображены на URL корневого контекста веб-приложения. Эти фильтры часто применяются в веб-приложениях. В табл. 16.1 перечислены сконфигурированные фильтры и указано их назначение.

**Таблица 16.1. Часто используемые фильтры сервлетов Spring MVC**

| Имя класса фильтра | Описание |
|---|---|
| `org.springframework.web.filter.CharacterEncodingFilter` | Этот фильтр применяется для указания кодировки символов в запросе |
| `org.springframework.web.filter.HiddenHttpMethodFilter` | Этот фильтр предоставляет поддержку для HTTP-методов, отличных от `GET` и `POST` (например, `PUT`) |
| `org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter` | Этот фильтр связывает диспетчер сущностей JPA (`EntityManager`) с потоком для полной обработки запроса. Он может помочь в восстановлении того же диспетчера сущностей для последующих запросов, поступающих от того же самого пользователя, что позволит функционировать таким средствам JPA, как ленивая выборка |

- Определен прослушиватель класса `org.springframework.web.context.ContextLoaderListener`. Он позволяет Spring загрузить и завершить корневой `WebApplicationContext`.
- Определен один сервлет диспетчера (по имени `appServlet`). Контекст `WebApplicationContext` для сервлета диспетчера расположен в `/src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml`.

# Создание первого представления в Spring MVC

Теперь, имея уровень обслуживания и готовую конфигурацию Spring MVC, можно приступать к реализации нашего первого представления. В этом разделе мы реализуем простое представление для отображения всех контактов, которые были изначально занесены в базу данных сценарием `test-data.sql`.

Как упоминалось ранее, для реализации представления мы будем использовать формат JSPX, который является форматом JSP с правильно построенным XML. Ниже перечислены основные преимущества JSPX по сравнению с JSP.

- JPSX требует более строгого отделения кода от уровня презентаций. Например, в документ JSPX не могут быть помещены "скриптлеты" Java.
- Доступны инструменты для выполнения моментальной проверки (синтаксиса XML), поэтому ошибки могут быть обнаружены на ранних этапах.

Нам потребуется добавить к проекту зависимости Maven, перечисленные в табл. 16.2.

## Конфигурирование сервлета диспетчера

Следующий шаг заключается в конфигурировании сервлета диспетчера. В листинге 16.10 представлена модифицированная конфигурация (файл `/src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml`).

**Таблица 16.2. Зависимости Maven для представления Spring MVC**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| org.springframework | spring-webmvc | 4.0.2.RELEASE | Модуль Spring Web MVC для поддержки MVC |
| org.hibernate.javax.persistence | hibernate-jpa-2.1-api | 1.0.0.Final | Реализация JPA из Hibernate |
| joda-time | joda-time | 2.2 | API-интерфейс для работы с датой и временем Joda-Time |
| org.jadira.usertype | usertype.core | 3.0.0.GA | Вспомогательный API-интерфейс с пользовательскими типами для работы с сохраненными данными даты и времени |
| org.springframework.data | spring-data-jpa | 1.5.0.RELEASE | API-интерфейс Spring Data JPA |
| com.google.guava | guava | 14.0.1 | Вспомогательный API-интерфейс для работы с коллекциями и т.д. |
| com.h2database | h2 | 1.3.172 | Встроенная база данных H2 |
| joda-time | joda-time-jsptags | 1.1.1 | Библиотека дескрипторов JSP, которая поддерживает форматирование типов Joda-Time в представлениях |

**Листинг 16.10. Конфигурация сервлета диспетчера**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:beans="http://www.springframework.org/schema/beans"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/mvc
     http://www.springframework.org/schema/mvc/spring-mvc.xsd
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context.xsd">

   <annotation-driven />

   <resources mapping="/resources/**" location="/resources/" />

   <beans:bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <beans:property name="prefix" value="/WEB-INF/views/" />
      <beans:property name="suffix" value=".jspx" />
   </beans:bean>

   <context:component-scan base-package="com.apress.prospring4.ch16" />
</beans:beans>
```

Пространство имен `mvc` объявлено в качестве стандартного. Дескриптор `<annotation-driven>` включает поддержку конфигурирования посредством аннотаций для контроллеров Spring MVC, а также поддержку преобразования типов и форматирования Spring. Кроме того, этим же дескриптором включается поддержка проверки достоверности бинов JSR-349. Дескриптор `<resources>` определяет статические ресурсы (например, CSS, JavaScript и графические изображения) вместе с их местоположениями, так что Spring MVC сможет улучшить производительность при обслуживании этих файлов. Дескриптор `<context:component-scan>` уже должен быть знакомым. Для интерфейса `ViewResolver` мы продолжим использовать класс `InternalResourceViewResolver` в качестве реализации. Однако мы изменим суффикс на `.jspx`.

## Реализация класса `ContactController`

После конфигурирования `WebApplicationContext` сервлета диспетчера займемся реализацией класса контроллера. Класс `ContactController` приведен в листинге 16.11.

**Листинг 16.11. Класс `ContactController`**

```
package com.apress.prospring4.ch16;
import java.util.List;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@RequestMapping("/contacts")
@Controller
public class ContactController {
  private final Logger logger = LoggerFactory.getLogger(ContactController.class);
  private ContactService contactService;
  @RequestMapping(method = RequestMethod.GET)
  public String list(Model uiModel) {
      logger.info("Listing contacts");
      List<Contact> contacts = contactService.findAll();
      uiModel.addAttribute("contacts", contacts);
      logger.info("No. of contacts: " + contacts.size());
      return "contacts/list";
  }
  @Autowired
  public void setContactService(ContactService contactService) {
      this.contactService = contactService;
  }
}
```

К классу `ContactController` применена аннотация `@Controller`, которая указывает, что это контроллер Spring MVC. Аннотация `@RequestMapping` на уровне класса задает корневой URL, который будет обрабатываться контроллером.

В этом случае все URL с префиксом `/ch16/contacts` будут направляться данному контроллеру. К методу `list()` также применена аннотация `@RequestMapping`, но на этот раз метод отображается на HTTP-метод `GET`. Это значит, что с помощью `list()` будет обработан URL вида `/ch16/contacts` с HTTP-методом `GET`. Внутри метода `list()` извлекается список контактов, который сохраняется в интерфейсе `Model`, экземпляр реализации которого передается методу средой Spring MVC. Наконец, возвращается логическое имя представления `contacts/list`. В конфигурации сервлета диспетчера в качестве распознавателя представлений указан `InternalResourceViewResolver`, а файл имеет префикс `/WEB-INF/views/` и суффикс `.jspx`. В результате Spring MVC выберет для представления файл `/WEB-INF/views/contacts/list.jspx`.

## Реализация представления для списка контактов

На следующем шаге мы должны реализовать страницу представления для отображения контактной информации, которой соответствует файл `/src/main/webapp/WEB-INF/views/contacts/list.jspx`. Содержимое этой страницы показано в листинге 16.12.

**Листинг 16.12. Представление списка контактов**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:joda="http://www.joda.org/joda/time/tags"
    version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8"/>
   <jsp:output omit-xml-declaration="yes"/>
   <h1>Contact Listing</h1>

   <c:if test="${not empty contacts}">
      <table>
         <thead>
            <tr>
               <th>First Name</th>
               <th>Last Name</th>
               <th>Birth Date</th>
            </tr>
         </thead>
         <tbody>
            <c:forEach items="${contacts}" var="contact">
               <tr>
                  <td>${contact.firstName}</td>
                  <td>${contact.lastName}</td>
                  <td><joda:format value="${contact.birthDate}"
                                   pattern="yyyy-MM-dd"/></td>
               </tr>
            </c:forEach>
         </tbody>
      </table>
   </c:if>
</div>
```

Если вы занимались разработкой JSP ранее, то код в листинге 16.12 должен выглядеть знакомым. Но поскольку это страница JSPX, ее содержимое встроено в дескриптор `<div>`. Вдобавок используемые библиотеки дескрипторов объявлены в виде пространство имен XML.

Во-первых, дескриптор `<jsp:directive.page>` определяет атрибуты, которые применяются ко всей странице JSPX, тогда как дескриптор `<jsp:output>` управляет свойствами вывода JSPX-документа.

Во-вторых, дескриптор `<c:if>` выясняет, пуст ли атрибут модели `contacts`. Так как мы уже заполнили базу данных тестовой информацией о контактах, атрибут `contacts` должен содержать данные. В результате дескриптор `<c:forEach>` визуализирует контактную информацию из таблицы на странице. Обратите внимание, что дескриптор `<joda:format>` форматирует атрибут `birthDate`, имеющий тип `DateTime` из Joda-Time.

### Тестирование представления списка контактов

Теперь все готово к тестированию представления списка контактов. Для начала постройте и разверните приложение. Чтобы протестировать представление списка контактов, откройте веб-браузер и посетите URL вида `http://localhost:8080/contact-webapp/contacts`. Вы должны увидеть страницу со списком контактов.

Итак, мы имеем первое работающее представление. В последующих разделах мы расширим приложение дополнительными представлениями и включим поддержку интернационализации, оформления темами и т.д.

## Обзор структуры проекта Spring MVC

Прежде чем погружаться в реализацию разнообразных аспектов веб-приложения, давайте посмотрим, каким образом выглядит структура проекта в примере веб-приложения, которое разрабатывается в этой главе.

Как правило, для поддержки различных функциональных возможностей в веб-приложении требуется большое количество файлов. Например, имеется множество файлов статических ресурсов, таких как стилевые таблицы, файлы кода JavaScript, изображения и библиотеки компонентов. Кроме того, есть файлы, которые поддерживают представление пользовательского интерфейса на разных языках. И, конечно же, существуют страницы представлений, которые будут проанализированы и визуализированы веб-контейнером, а также файлы компоновки и определений, применяемые шаблонной инфраструктурой (например, Apache Tiles) для обеспечения согласованного внешнего вида и поведения приложения.

Рекомендуемый подход предусматривает хранение файлов, которые служат разным целям, в хорошо структурированной иерархии папок, что позволит получить четкую картину использования различных ресурсов в приложении и обеспечить простоту последующего сопровождения.

В табл. 16.3 описана структура папок в веб-приложении, которое будет разработано в этой главе вместе с назначением каждой папки. Обратите внимание, что представленная здесь структура не является обязательной, но она часто используется в сообществе разработчиков веб-приложений.

**Таблица 16.3. Описание папок в примере веб-проекта**

| Имя папки | Назначение | Примечание |
|---|---|---|
| ckeditor | CKEditor (http://ckeditor.com) — это библиотека компонентов JavaScript, которая предоставляет редактор форматированного текста для формы ввода. Мы будем использовать эту библиотеку для редактирования описания контакта | Скопируйте содержимое этой папки из исходного кода примеров в папку своего проекта |
| jqgrid | jqGrid (www.trirand.com/blog) — это библиотека компонентов, построенная поверх jQuery, которая предоставляет различные основанные на сетке компоненты для представления данных. Мы будем использовать эту библиотеку для реализации сетки, отображающей контакты, а также для поддержки разбиения на страницы в стиле Ajax | Скопируйте содержимое этой папки из исходного кода примеров в папку своего проекта |
| scripts | Это папка для всех общих файлов JavaScript. В примерах данной главы будут использоваться JavaScript-библиотеки jQuery (http://jquery.org) и jQuery UI (http://jqueryui.com) для реализации обогащенного пользовательского интерфейса. Сценарии будут помещены в эту папку. Сюда также попадут и самостоятельно разработанные JavaScript-библиотеки | Скопируйте содержимое этой папки из исходного кода примеров в папку своего проекта |
| styles | Хранит файлы стилевых таблиц и связанные со стилями файлы изображений | Скопируйте содержимое этой папки из исходного кода примеров в папку своего проекта |
| WEB-INF/i18n | Файлы для поддержки интернационализации. Файл application*.properties хранит текст, связанный с компоновкой (например, заголовки страниц, метки полей и заголовки меню). Файл message*.properties содержит разнообразные сообщения (например, сообщения об успехе, сообщения об ошибках и сообщения проверки достоверности). В рассматриваемом примере будут поддерживаться английский язык (США) и китайский язык (Гонконг) | Скопируйте содержимое этой папки из исходного кода примеров в папку своего проекта |
| WEB-INF/layouts | Эта папка хранит представления и определения компоновки. Файлы в ней будут использоваться шаблонной инфраструктурой Apache Tiles (http://tiles.apache.org) | |
| WEB-INF/spring | Эта папка хранит конфигурации WebApplicationContext для Spring MVC. Здесь находятся конфигурации контекстов и корневого уровня, и уровня сервлетов диспетчеров | |
| WEB-INF/views | Эта папка хранит представления (в нашем случае файлы JSPX), которые будут использоваться приложением | |

В приведенных далее разделах для поддержки реализации нам понадобятся различные файлы (например, файлы CSS, файлы JavaScript и файлы изображений). Исходный код файлов CSS и JavaScript здесь приводиться не будет. С учетом этого рекомендуется загрузить архив с исходным кодом для этой главы и распаковать его во временную папку, чтобы требуемые для проекта файлы можно было копировать напрямую.

# Включение интернационализации

При разработке веб-приложений рекомендуется включать интернационализацию на самых ранних этапах. Основная работа связана с вынесением текста пользовательского интерфейса и сообщений в файлы свойств.

Даже если требования по интернационализации в настоящее время отсутствуют, все равно полезно вынести строки, зависящие от языка, за пределы приложения, чтобы в будущем было проще поддерживать дополнительные языки.

В Spring MVC интернационализация включается очень просто. Для начала вынесите параметры, зависящие от языка, в разнообразные файлы свойств внутри папки `/WEB-INF/i18n`, как было описано в табл. 16.3. Поскольку мы будем поддерживать английский язык (США) и китайский язык (Гонконг), понадобятся четыре файла. Файлы `application.properties` и `message.properties` хранят параметры для стандартной локали, которой в данном случае является английский язык (США). Кроме того, файлы `application_zh_HK.properties` и `message_zh_HK.properties` содержат параметры для китайского языка (Гонконг).

## Конфигурирование интернационализации в сервлете диспетчера

Теперь, располагая параметрами языков, можно сконфигурировать `WebApplicationContext` сервлета диспетчера для поддержки интернационализации.

В листинге 16.13 показано модифицированное содержимое конфигурационного файла с поддержкой интернационализации (`servlet-context.xml`).

**Листинг 16.13. Модифицированная конфигурация контекста сервлета**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

    <annotation-driven />

    <resources location="/, classpath:/META-INF/web-resources/"
               mapping="/resources/**"/>

    <default-servlet-handler/>
```

```
    <beans:bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <beans:property name="prefix" value="/WEB-INF/views/" />
        <beans:property name="suffix" value=".jspx" />
    </beans:bean>

    <context:component-scan base-package="com.apress.prospring4.ch16" />

    <interceptors>
        <beans:bean
class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
            p:paramName="lang"/>
    </interceptors>

    <beans:bean class="org.springframework.context.support.ReloadableResource
BundleMessageSource"
        id="messageSource"
        p:basenames="WEB-INF/i18n/messages,WEB-INF/i18n/application"
        p:fallbackToSystemLocale="false"/>

    <beans:bean
        class="org.springframework.web.servlet.i18n.CookieLocaleResolver"
        id="localeResolver" p:cookieName="locale"/>
</beans:beans>
```

Во-первых, в листинге 16.13 добавлено пространство имен `p`, а определение ресурсов пересмотрено для отражения новой структуры папок, описанной в табл. 16.3. Дескриптор `<resources>` принадлежит пространству имен `mvc`; он был введен в Spring 3. Этот дескриптор определяет местоположения файлов статических ресурсов, что позволяет Spring MVC обрабатывать файлы в этих папках более эффективно. Атрибут `location` внутри этого дескриптора задает папки для статических ресурсов. Первый путь, `/`, указывает корневую папку для веб-приложения, которой является `/src/main/webapp`, а второй путь, `classpath:/META-INF/web-resources/` — ресурсные файлы для включенных библиотек. Это окажется удобным, если вы включите модуль Spring JavaScript, который имеет поддерживающие ресурсные файлы в папке `/META-INF/web-resources`.

Атрибут `mapping` определяет URL для отображения на статические ресурсы; в качестве примера для URL вида `http://localhost:8080/contact-webapp/resources/styles/standard.css` модуль Spring MVC будет извлекать файл `standard.css` из папки `/src/main/webapp/styles`.

Дескриптор `<default-servlet-handler/>` разрешает отображение сервлета диспетчера на URL корневого контекста веб-приложения, по-прежнему позволяя запросам статических ресурсов обрабатываться сервлетом, который установлен в контейнере в качестве стандартного.

Во-вторых, здесь определен перехватчик Spring MVC с классом `LocaleChangeInterceptor`, который перехватывает все запросы к сервлету диспетчера. Перехватчик поддерживает переключение локалей с помощью конфигурируемого параметра запроса. В конфигурации перехватчика определен параметр URL по имени `lang`, предназначенный для изменения локали, которая применяется в приложении.

В-третьих, определен бин класса `ReloadableResourceBundleMessageSource`. Класс `ReloadableResourceBundleMessageSource` реализует интерфейс `MessageSource`, который загружает сообщения из определенных файлов (в данном случае это файлы `messages*.properties` и `application*.properties` в папке `/WEB-INF/i18n`) для поддержки интернационализации. Обратите внимание на свойство `fallbackToSystemLocale`. Это свойство указывает Spring MVC, нужно ли возвращаться к локали системы, в которой выполняется приложение, если специфический пакет ресурсов для клиентской локали не найден.

Наконец, в-четвертых, определен бин класса `CookieLocaleResolver`. Этот класс поддерживает хранение и извлечение параметров локали из cookie-набора пользовательского браузера.

## Модификация представления списка контактов для поддержки интернационализации

Теперь мы можем изменить страницу JSPX для отображения интернационализированных сообщений. В листинге 16.14 показано пересмотренное представление списка контактов (`list.jspx`).

**Листинг 16.14. Модифицированное представление списка контактов для поддержки интернационализации**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:c="http://java.sun.com/jsp/jstl/core"
   xmlns:joda="http://www.joda.org/joda/time/tags"
   xmlns:spring="http://www.springframework.org/tags"
   version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8"/>
   <jsp:output omit-xml-declaration="yes"/>

   <spring:message code="label_contact_list" var="labelContactList"/>
   <spring:message code="label_contact_first_name"
                   var="labelContactFirstName"/>
   <spring:message code="label_contact_last_name"
                   var="labelContactLastName"/>
   <spring:message code="label_contact_birth_date"
                   var="labelContactBirthDate"/>

   <h1>${labelContactList}</h1>
   <c:if test="${not empty contacts}">
      <table>
         <thead>
            <tr>
               <th>${labelContactFirstName}</th>
               <th>${labelContactLastName}</th>
               <th>${labelContactBirthDate}</th>
            </tr>
         </thead>
         <tbody>
            <c:forEach items="${contacts}" var="contact">
```

```
                <tr>
                    <td>${contact.firstName}</td>
                    <td>${contact.lastName}</td>
                    <td><joda:format value="${contact.birthDate}"
                                     pattern="yyyy-MM-dd"/></td>
                </tr>
            </c:forEach>
        </tbody>
    </table>
</c:if>
</div>
```

Первым делом к странице добавлено пространство имен `spring`. Затем с применением дескриптора `<spring:message>` в соответствующие переменные загружаются сообщения, требуемые представлением. Наконец, заголовок страницы и метки изменяются для использования интернационализированных сообщений.

Теперь постройте и повторно разверните проект, откройте браузер и перейдите на URL вида `http://localhost:8080/contact-webapp/contacts?lang=zh_HK`. Вы увидите страницу для китайской локали (Гонконг).

Поскольку мы определили распознаватель `localeResolver` в `WebApplicationContext` сервлета диспетчера, Spring MVC будет сохранять параметр локали в cookie-наборе браузера (с именем `locale`) и по умолчанию cookie-набор будет сохраняться для сеанса пользователя. Если вы хотите удерживать cookie-набор на протяжении более длительного периода времени, в определении бина `localeResolver` из листинга 16.13 необходимо установить свойство `cookieMaxAge`, которое унаследовано от класса `org.springframework.web.util.CookieGenerator`.

Чтобы переключиться на английский язык (США), можете изменить URL в браузере, указав `?lang=en_US`, в результате чего отобразится страница для английского языка (США). Хотя мы не предусмотрели файл свойств с именем `application_en_US.properties`, Spring MVC возвратится к применению файла `application.properties`, в котором хранятся свойства на стандартном языке, т.е. английском.

# Использование шаблонов и оформления темами

Помимо интернационализации веб-приложение требует обеспечения подходящего внешнего вида (например, бизнес-сайт должен выглядеть профессионально строго, в то время как социальный веб-сайт может иметь более свободный стиль), а также согласованной компоновки, чтобы пользователи не путались при работе с ним.

В веб-приложении стили должны быть вынесены во внешние стилевые таблицы, а не жестко закодированы на странице представления. Кроме того, имена стилей также должны быть согласованными, чтобы можно было подготовить различные *темы*, просто переключая файл стилевой таблицы. В Spring MVC предлагается широкая поддержка оформления темами для веб-приложений.

Вдобавок для предоставления согласованной компоновки требуется шаблонная инфраструктура. В этом разделе для поддержки применения шаблонов к представлениям мы будем пользоваться популярной шаблонной инфраструктурой под названием Apache Tiles (`http://tiles.apache.org`).

Платформа Spring MVC тесно интегрирована с инфраструктурой Apache Tiles. Кроме того, Spring изначально поддерживает Velocity и FreeMarker — шаблонные системы более общего характера, пригодные для применения за рамками веб-приложений, а также шаблоны электронной почты и т.д.

В последующих разделах мы обсудим включение поддержки оформления темами в Spring MVC и способы использования Apache Tiles при определении компоновки страницы.

## Поддержка оформления темами

В Spring MVC предлагается комплексная поддержка оформления темами, которая легко включается в веб-приложениях.

Например, в приложении работы с контактами, рассматриваемом в этой главе, мы хотим создать тему по имени `standard`. Для начала в папке `/src/main/resources` создадим файл по имени `standard.properties` с содержимым, показанным в листинге 16.15.

**Листинг 16.15. Файл `standard.properties`**

```
styleSheet=resources/styles/standard.css
```

Этот файл свойств содержит свойство `styleSheet`, которое указывает стилевую таблицу для использования в теме `standard`. Данный файл свойств является пакетом ресурсов (`ResourceBundle`) для темы, и к теме можно добавлять столько компонентов, сколько необходимо (скажем, местоположение изображения логотипа, местоположение фонового изображения и т.д.).

Следующий шаг состоит в конфигурировании контекста `WebApplicationContext` сервлета диспетчера с целью поддержки оформления темами, для чего понадобится модифицировать конфигурационный файл `servlet-context.xml`. Во-первых, в определение `<interceptors>` необходимо добавить еще один бин перехватчика (листинг 16.16).

**Листинг 16.16. Конфигурирование перехватчика темы**

```
<interceptors>
   <beans:bean
     class="org.springframework.web.servlet.theme.ThemeChangeInterceptor"/>
   <beans:bean
      class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
      p:paramName="lang"/>
</interceptors>
```

Здесь добавлен новый перехватчик `ThemeChangeInterceptor`, который перехватывает каждый запрос для изменения темы.

Во-вторых, в файл конфигурации `servlet-context.xml` нужно добавить определения бинов, показанные в листинге 16.17.

**Листинг 16.17. Конфигурирование поддержки темы**

```
<beans:bean class="org.springframework.ui.context.support.
ResourceBundleThemeSource" id="themeSource"/>

<beans:bean class="org.springframework.web.servlet.theme.CookieThemeResolver"
id="themeResolver" p:cookieName="theme" p:defaultThemeName="standard"/>
```

В листинге 16.17 определены два бина. Первый бин, реализованный с помощью класса `ResourceBundleThemeSource`, отвечает за загрузку пакета ресурсов (`ResourceBundle`) активной темы. Например, если активная тема называется `standard`, то этот бин будет искать файл `standard.properties` как пакет ресурсов данной темы. Второй бин, реализованный посредством класса `CookieThemeResolver`, распознает активную тему для пользователей. Свойство `defaultThemeName` определяет тему, используемую по умолчанию, которой является `standard`. Обратите внимание, что для сохранения темы пользователю в классе `CookieThemeResolver` применяются cookie-наборы. Существует также класс `SessionThemeResolver`, который обеспечивает сохранение атрибута темы в пользовательском сеансе.

Итак, тема `standard` сконфигурирована и готова для использования в наших представлениях. В листинге 16.18 показано модифицированное представление списка контактов (`/WEB-INF/views/contacts/list.jspx`) с поддержкой темы.

**Листинг 16.18. Представление списка контактов с поддержкой темы**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:joda="http://www.joda.org/joda/time/tags"
    xmlns:spring="http://www.springframework.org/tags"
    version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8"/>
   <jsp:output omit-xml-declaration="yes"/>

   <spring:message code="label_contact_list" var="labelContactList"/>
   <spring:message code="label_contact_first_name"
                   var="labelContactFirstName"/>
   <spring:message code="label_contact_last_name"
                   var="labelContactLastName"/>
   <spring:message code="label_contact_birth_date"
                   var="labelContactBirthDate"/>

   <head>
      <spring:theme code="styleSheet" var="app_css" />
      <spring:url value="/${app_css}" var="app_css_url" />
      <link rel="stylesheet" type="text/css" media="screen"
            href="${app_css_url}" />
   </head>

   <h1>${labelContactList}</h1>

   <c:if test="${not empty contacts}">
      <table>
         <thead>
```

```
            <tr>
               <th>${labelContactFirstName}</th>
               <th>${labelContactLastName}</th>
               <th>${labelContactBirthDate}</th>
            </tr>
            </thead>
            <tbody>
            <c:forEach items="${contacts}" var="contact">
               <tr>
                  <td>${contact.firstName}</td>
                  <td>${contact.lastName}</td>
                  <td><joda:format value="${contact.birthDate}"
                        pattern="yyyy-MM-dd"/></td>
               </tr>
            </c:forEach>
            </tbody>
         </table>
      </c:if>
   </div>
```

К представлению добавлен раздел `<head>`, а дескриптор `<spring:theme>` используется для извлечения из пакета ресурсов темы свойства `styleSheet`, которое представляет собой файл стилевой таблицы `standard.css`. Наконец, к представлению добавлена ссылка на эту стилевую таблицу.

После повторного построения и развертывания приложения на сервере откройте браузер и посетите URL представления списка контактов снова (`http://localhost:8080/contact-webapp/contacts`). Вы увидите, что стиль, определенный в файле `standard.css`, был применен.

Используя поддержку оформления темами в Spring MVC, можно легко добавлять новые темы либо изменять существующую тему внутри приложения.

## Применение шаблонов представлений с помощью Apache Tiles

Из числа шаблонных инфраструктур, использующих технологию JSP, наиболее популярной является Apache Tiles (`http://tiles.apache.org`). Платформа Spring MVC тесно интегрирована с Apache Tiles.

Для работы с Apache Tiles и реализации проверки достоверности данных понадобится добавить в проект обязательные зависимости, которые описаны в табл. 16.4.

**Таблица 16.4. Зависимости Maven для Apache Tiles и проверки достоверности**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.apache.tiles` | `tiles-core` | 3.0.4 | Библиотека ядра для Apache Tiles |
| `org.apache.tiles` | `tiles-jsp` | 3.0.4 | Поддержка Apache Tiles для файлов представлений JSP |
| `javax.validation` | `validation-api` | 1.1.0.Final | API-интерфейс проверки достоверности JSR-349 |
| `org.hibernate` | `hibernate-validator` | 5.1.0.Final | Реализация проверки достоверности JSR-349 |

В последующих разделах мы обсудим, как реализовать шаблоны страниц, включая дизайн компоновки страниц, определение и реализацию компонентов внутри компоновки.

### *Дизайн компоновки шаблона*

Прежде всего, мы должны определить количество шаблонов, требуемых в приложении, и компоновку для каждого шаблона.

В приложении работы с контактами, рассматриваемом в этой главе, нам необходим только один шаблон. Как показано на рис. 16.4, шаблон довольно тривиален.

Как видите, шаблон требует следующих страничных компонентов.

- `/WEB-INF/views/header.jspx`. Эта страница предоставляет область заголовка.
- `/WEB-INF/views/menu.jspx`. Эта страница предоставляет область левого меню, а также форму входа, которая будет разработана позже в настоящей главе.
- `/WEB-INF/views/footer.jspx`. Эта страница предоставляет область нижнего колонтитула.

Для определения шаблона мы будем использовать Apache Tiles, и нам понадобится разработать файл шаблона страницы, а также файл определений компоновки, которые описаны ниже.

- `/WEB-INF/layouts/default.jspx`. Эта страница предоставляет полную компоновку для конкретного шаблона.
- `/WEB-INF/layouts/layouts.xml`. Эта страница содержит определения компоновки, требуемые Apache Tiles.

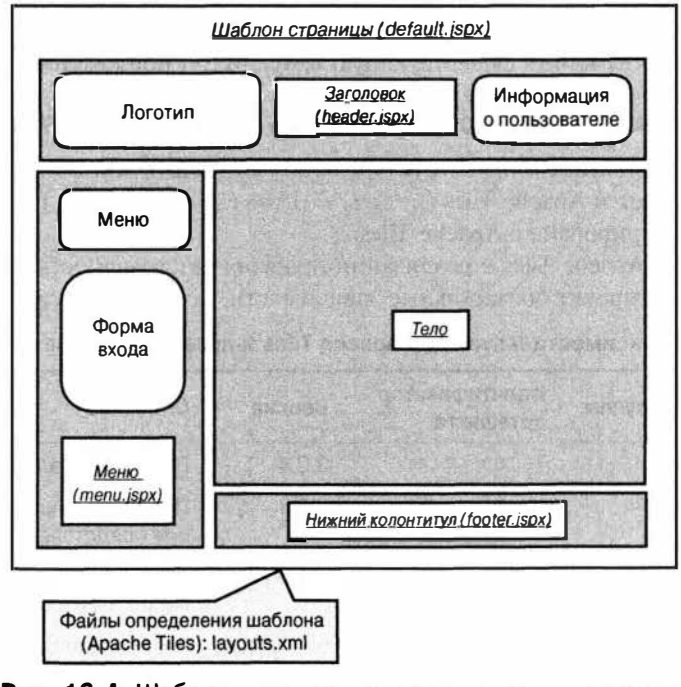


**Рис. 16.4.** Шаблон страницы с компонентами компоновки

### *Реализация компонентов компоновки страницы*

Имея определение компоновки, можно реализовать страничные компоненты. Сначала мы разработаем файл шаблона страницы и файлы определений компоновки, требуемые Apache Tiles.

В листинге 16.19 приведено содержимое файла определений Apache Tiles (`src/main/webapp/WEB-INF/layouts/layouts.xml`).

**Листинг 16.19. Файл определений Apache Tiles**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tiles-definitions PUBLIC
      "-//Apache Software Foundation//DTD Tiles Configuration 2.1//EN"
      "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">

<tiles-definitions>
   <definition name="default" template="/WEB-INF/layouts/default.jspx">
      <put-attribute name="header" value="/WEB-INF/views/header.jspx" />
      <put-attribute name="menu" value="/WEB-INF/views/menu.jspx" />
      <put-attribute name="footer" value="/WEB-INF/views/footer.jspx" />
   </definition>
</tiles-definitions>
```

Код в этом файле должен быть прост для понимания. Здесь имеется одно определение шаблона страницы по имени `default`. Код шаблона находится в файле `default.jspx`. Внутри страницы определены три компонента с именами `header`, `menu` и `footer`. Содержимое этих компонентов будет загружаться из файлов, указанных в атрибутах `value`. За детальным описанием определения Apache Tiles обращайтесь в документацию по этому проекту (`http://tiles.apache.org/`).

В листинге 16.20 представлен стандартный шаблон страницы (`default.jspx`).

**Листинг 16.20. Файл стандартного шаблона**

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:fn="http://java.sun.com/jsp/jstl/functions"
     xmlns:tiles="http://tiles.apache.org/tags-tiles"
     xmlns:spring="http://www.springframework.org/tags">

   <jsp:output doctype-root-element="HTML"
               doctype-system="about:legacy-compat" />

   <jsp:directive.page contentType="text/html;charset=UTF-8" />
   <jsp:directive.page pageEncoding="UTF-8" />

   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <meta http-equiv="X-UA-Compatible" content="IE=8" />

      <spring:theme code="styleSheet" var="app_css" />
      <spring:url value="/${app_css}" var="app_css_url" />
      <link rel="stylesheet" type="text/css" media="screen"
            href="${app_css_url}" />

      <!-- Извлечь пользовательскую локаль из контекста страницы
          (она была установлена распознавателем локалей Spring MVC) -->
```

```
      <c:set var="userLocale">
         <c:set var="plocale">${pageContext.response.locale}</c:set>
         <c:out value="${fn:replace(plocale, '_', '-')}" default="en" />
      </c:set>
      <spring:message code="application_name" var="app_name"
                      htmlEscape="false"/>
      <title><spring:message code="welcome_h3" arguments="${app_name}" />
      </title>
   </head>
   <body class="tundra spring">
      <div id="headerWrapper">
         <tiles:insertAttribute name="header" ignore="true" />
      </div>
      <div id="wrapper">
         <tiles:insertAttribute name="menu" ignore="true" />
         <div id="main">
            <tiles:insertAttribute name="body"/>
            <tiles:insertAttribute name="footer" ignore="true"/>
         </div>
      </div>
   </body>
</html>
```

По большому счету это страница JSP. Основные моменты, на которые следует обратить внимание, описаны ниже.

- В шаблон помещен дескриптор `<spring:theme>`, который поддерживает оформление темами на уровне шаблона.
- Дескриптор `<tiles:insertAttribute>` используется для указания страничных компонентов, которые должны загружаться из других файлов, как отражено в файле `layouts.xml`.

Теперь давайте реализуем компоненты заголовка, меню и нижнего колонтитула. Код для этих компонентов приведен в листингах 16.21, 16.22 и 16.23.

**Листинг 16.21. Компонент заголовка (`header.jspx`)**

```
<div id="header" xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:spring="http://www.springframework.org/tags"
   version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8" />
   <jsp:output omit-xml-declaration="yes" />
   <spring:message code="header_text" var="headerText"/>
   <div id="appname">
      <h1>${headerText}</h1>
   </div>
</div>
```

**Листинг 16.22. Компонент меню (`menu.jspx`)**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div id="menu" xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:c="http://java.sun.com/jsp/jstl/core"
   xmlns:spring="http://www.springframework.org/tags"
   version="2.0">
      <jsp:directive.page contentType="text/html;charset=UTF-8" />
      <jsp:output omit-xml-declaration="yes" />

      <spring:message code="menu_header_text" var="menuHeaderText"/>
      <spring:message code="menu_add_contact" var="menuAddContact"/>
      <spring:url value="/contacts?form" var="addContactUrl"/>

   <h3>${menuHeaderText}</h3>
   <a href="${addContactUrl}"><h3>${menuAddContact}</h3></a>
</div>
```

**Листинг 16.23. Компонент нижнего колонтитула (`footer.jspx`)**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div id="footer" xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:spring="http://www.springframework.org/tags" version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8" />
   <jsp:output omit-xml-declaration="yes" />

   <spring:message code="home_text" var="homeText"/>
   <spring:message code="label_en_US" var="labelEnUs"/>
   <spring:message code="label_zh_HK" var="labelZhHk"/>
   <spring:url value="/contacts" var="homeUrl"/>

   <a href="${homeUrl}">${homeText}</a> |
   <a href="${homeUrl}?lang=en_US">${labelEnUs}</a> |
   <a href="${homeUrl}?lang=zh_HK">${labelZhHk}</a>
</div>
```

Теперь можно изменить представление списка контактов, чтобы оно удовлетворяло шаблону. Главным образом, придется только удалить раздел <head>, т.к. он уже присутствует на шаблонной странице (`default.jspx`). Модифицированное представление списка контактов показано в листинге 16.24.

**Листинг 16.24. Модифицированное представление списка контактов (`/views/contacts/list.jspx`)**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:joda="http://www.joda.org/joda/time/tags"
    xmlns:spring="http://www.springframework.org/tags"
    version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8"/>
   <jsp:output omit-xml-declaration="yes"/>

   <spring:message code="label_contact_list" var="labelContactList"/>
   <spring:message code="label_contact_first_name"
                   var="labelContactFirstName"/>
```

```
    <spring:message code="label_contact_last_name"
                    var="labelContactLastName"/>
    <spring:message code="label_contact_birth_date"
                    var="labelContactBirthDate"/>

    <h1>${labelContactList}</h1>

    <c:if test="${not empty contacts}">
        <table>
            <thead>
                <tr>
                    <th>${labelContactFirstName}</th>
                    <th>${labelContactLastName}</th>
                    <th>${labelContactBirthDate}</th>
                </tr>
            </thead>
            <tbody>
                <c:forEach items="${contacts}" var="contact">
                    <tr>
                        <td>${contact.firstName}</td>
                        <td>${contact.lastName}</td>
                        <td><joda:format value="${contact.birthDate}"
                                         pattern="yyyy-MM-dd"/></td>
                    </tr>
                </c:forEach>
            </tbody>
        </table>
    </c:if>
</div>
```

Итак, шаблон, определение и компоненты готовы; следующий шаг заключается в конфигурировании Spring MVC для интеграции с Apache Tiles.

### *Конфигурирование Apache Tiles в Spring MVC*

Конфигурирование поддержки Apache Tiles в Spring MVC производится просто. В конфигурацию сервлета диспетчера (`servlet-context.xml`) потребуется внести изменение, заменив `InternalResourceViewResolver` классом `UrlBasedViewResolver`. Содержимое пересмотренного файла конфигурации приведено в листинге 16.25.

**Листинг 16.25. Конфигурирование поддержки Apache Tiles в Spring MVC**

```
<?xml version="1.0" encoding="UTF-8"?>

    <!-- Остальной код не показан -->
    <!-- Удалите следующий бин -->
    <beans:bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <beans:property name="prefix" value="/WEB-INF/views/" />
        <beans:property name="suffix" value=".jspx" />
    </beans:bean>

    <!-- Остальной код не показан -->

    <!-- Добавьте следующий бин -->
```

```
    <!-- Конфигурация Apache Tiles -->
    <beans:bean
       class="org.springframework.web.servlet.view.UrlBasedViewResolver"
       id="tilesViewResolver">
       <beans:property name="viewClass"
           value="org.springframework.web.servlet.view.tiles3.TilesView"/>
    </beans:bean>

    <beans:bean
       class="org.springframework.web.servlet.view.tiles3.TilesConfigurer"
       id="tilesConfigurer">
       <beans:property name="definitions">
          <beans:list>
             <beans:value>/WEB-INF/layouts/layouts.xml</beans:value>
             <beans:value>/WEB-INF/views/**/views.xml</beans:value>
          </beans:list>
       </beans:property>
    </beans:bean>

</beans:beans>
```

Бин, который должен быть удален, обозначен комментарием, а после него следуют определения новых бинов. Сначала удаляется исходный бин `ViewResolver` (имеющий класс `InternalResourceViewResolver`). Затем определяется бин `ViewResolver` с классом `UrlBasedViewResolver` и свойством `viewClass`, установленным в класс `TilesView`, который обеспечивает поддержку Apache Tiles в Spring MVC. Наконец, определяется бин `tilesConfigurer`, который предоставляет конфигурации компоновки, требуемые для Apache Tiles.

Финальный конфигурационный файл, который понадобится подготовить — это `/WEB-INF/views/contacts/views.xml`, где определяются представления для приложения работы с контактами. Содержимое этого файла показано в листинге 16.26.

**Листинг 16.26. Файл `views.xml`**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software Foundation//DTD Tiles
Configuration 3.0//EN" "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">
<tiles-definitions>
   <definition extends="default" name="contacts/list">
     <put-attribute name="body" value="/WEB-INF/views/contacts/list.jspx" />
   </definition>
</tiles-definitions>
```

В листинге 16.26 логическое имя представления отображается на соответствующий атрибут `body` представления. Как и в классе `ContactController` из листинга 16.11, метод `list()` возвращает логическое имя представления `contacts/list`, так что Apache Tiles сможет отобразить имя представления на правильный шаблон и тело представления.

Теперь можно протестировать страницу. Удостоверьтесь, что проект повторно построен и развернут на сервере. После загрузки представления списка контактов снова (`http://localhost:8080/contact-webapp/contacts`) будет отображено представление, основанное на шаблоне.

# Реализация представлений для информации о контактах

Продолжим реализацию представлений, которые позволят пользователям просматривать детальную информацию о контакте, создавать новый контакт и обновлять сведения о существующем контакте.

В последующих разделах мы обсудим отображение URL на различные представления, а также реализацию этих представлений. Мы также покажем, как включить поддержку проверки достоверности JSR-349 в Spring MVC для представления редактирования.

## Отображение URL на представления

Первым делом мы должны спроектировать отображение различных URL на соответствующие представления. Одним из рекомендуемых подходов в Spring MVC является применение для отображения представлений URL в стиле REST. В табл. 16.5 описано отображение URL на представления, а также указаны имена методов контроллера, которые будут обрабатывать соответствующие действия.

**Таблица 16.5. Отображение URL на представления**

| URL | HTTP-метод | Метод контроллера | Описание |
|---|---|---|---|
| `/contacts` | `GET` | `list()` | Выводит список контактов |
| `/contacts/{id}` | `GET` | `show()` | Отображает информацию об одном контакте |
| `/contacts/{id}?form` | `GET` | `updateForm()` | Отображает форму редактирования для обновления существующего контакта |
| `/contacts/{id}?form` | `POST` | `update()` | Пользователи обновляют информацию о контакте и отправляют форму. Здесь будут обрабатываться данные |
| `/contacts?form` | `GET` | `createForm()` | Отображает форму редактирования для создания нового контакта |
| `/contacts?form` | `POST` | `create()` | Пользователи вводят информацию о контакте и отправляют форму. Здесь будут обрабатываться данные |
| `/contacts/photo/{id}` | `GET` | `downloadPhoto()` | Загружает фотографию для контакта |

## Реализация представления для просмотра контакта

А теперь давайте реализуем представление для просмотра информации о контакте.

Реализация этого представления состоит из трех шагов.

- Реализация метода контроллера.
- Реализация представления для просмотра контакта (`/views/contacts/show.jspx`).
- Изменение файла определения представлений (`/views/contacts/views.xml`) для данного представления.

В листинге 16.27 приведена реализация метода `show()` класса `ContactController`, предназначенного для отображения информации о контакте.

**Листинг 16.27. Метод `show()` класса `ContactController`**

```
package com.apress.prospring4.ch16;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@RequestMapping("/contacts")
@Controller
public class ContactController {
    private final Logger logger =
        LoggerFactory.getLogger(ContactController.class);

    private ContactService contactService;

    @RequestMapping(method = RequestMethod.GET)
    public String list(Model uiModel) {
        logger.info("Listing contacts");

        List<Contact> contacts = contactService.findAll();
        uiModel.addAttribute("contacts", contacts);

        logger.info("No. of contacts: " + contacts.size());

        return "contacts/list";
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public String show(@PathVariable("id") Long id, Model uiModel) {
        Contact contact = contactService.findById(id);
        uiModel.addAttribute("contact", contact);

        return "contacts/show";
    }

    @Autowired
    public void setContactService(ContactService contactService) {
        this.contactService = contactService;
    }
}
```

Аннотация `@RequestMapping`, примененная к методу `show()`, указывает, что этот метод предназначен для обработки URL вида `/contacts/{id}` с HTTP-методом `GET`. К аргументу `id` данного метода применена аннотация `@PathVariable`, которая заставляет Spring MVC извлекать `id` из URL и помещать его в этот аргумент. Затем контакт извлекается и добавляется к модели, после чего возвращается логическое имя представления `contacts/show`. Следующий шаг заключается в реализации представления просмотра контакта (`/views/contacts/show.jspx`), которое показано в листинге 16.28.

**Листинг 16.28. Представление просмотра контакта**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:spring="http://www.springframework.org/tags"
    xmlns:form="http://www.springframework.org/tags/form"
    xmlns:joda="http://www.joda.org/joda/time/tags"
    version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8"/>
   <jsp:output omit-xml-declaration="yes"/>

   <spring:message code="label_contact_info" var="labelContactInfo"/>
   <spring:message code="label_contact_first_name"
                   var="labelContactFirstName"/>
   <spring:message code="label_contact_last_name"
                   var="labelContactLastName"/>
   <spring:message code="label_contact_birth_date"
                   var="labelContactBirthDate"/>
   <spring:message code="label_contact_description"
                   var="labelContactDescription"/>
   <spring:message code="label_contact_update" var="labelContactUpdate"/>
   <spring:message code="date_format_pattern" var="dateFormatPattern"/>
   <spring:url value="/contacts" var="editContactUrl"/>

   <h1>${labelContactInfo}</h1>

   <div id="contactInfo">

      <c:if test="${not empty message}">
        <div id="message" class="${message.type}">${message.message}</div>
      </c:if>

      <table>
         <tr>
            <td>${labelContactFirstName}</td>
            <td>${contact.firstName}</td>
         </tr>
         <tr>
            <td>${labelContactLastName}</td>
            <td>${contact.lastName}</td>
         </tr>
         <tr>
            <td>${labelContactBirthDate}</td>
            <td><joda:format value="${contact.birthDate}"
                             pattern="${dateFormatPattern}"/></td>
         </tr>
```

```
            <tr>
               <td>${labelContactDescription}</td>
               <td>${contact.description}</td>
            </tr>
         </table>

         <a href="${editContactUrl}/${contact.id}?form">${labelContactUpdate}
         </a>
      </div>
   </div>
```

Эта страница проста; она лишь отображает внутри себя атрибут `contact` модели. Последний шаг связан с модификацией файла определения представлений (`/views/contacts/views.xml`) для отображения логического имени представления `contacts/show`. Просто добавьте в этот файл фрагмент кода из листинга 16.29, поместив его в дескриптор `<tiles-definitions>`.

**Листинг 16.29. Отображение для представления просмотра контакта**

```
   <definition extends="default" name="contacts/show">
      <put-attribute name="body" value="/WEB-INF/views/contacts/show.jspx" />
   </definition>
```

На этом представление просмотра контакта завершено. Теперь необходимо добавить ссылку на представление просмотра контакта (`/views/contacts/list.jspx`) для каждого контакта. В листинге 16.30 приведено пересмотренное содержимое указанного файла.

**Листинг 16.30. Добавление ссылки на представление просмотра контакта**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:joda="http://www.joda.org/joda/time/tags"
    xmlns:spring="http://www.springframework.org/tags"
    version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8"/>
   <jsp:output omit-xml-declaration="yes"/>

   <spring:message code="label_contact_list" var="labelContactList"/>
   <spring:message code="label_contact_first_name"
var="labelContactFirstName"/>
   <spring:message code="label_contact_last_name"
var="labelContactLastName"/>
   <spring:message code="label_contact_birth_date"
var="labelContactBirthDate"/>

   <spring:url value="/contacts" var="showContactUrl"/>

   <h1>${labelContactList}</h1>

   <c:if test="${not empty contacts}">
      <table>
         <thead>
```

```
            <tr>
               <th>${labelContactFirstName}</th>
               <th>${labelContactLastName}</th>
               <th>${labelContactBirthDate}</th>
            </tr>
            </thead>
            <tbody>
            <c:forEach items="${contacts}" var="contact">
               <tr>
                  <td><a href="${showContactUrl}/${contact.id}">
                     ${contact.firstName}</a></td>
                  <td>${contact.lastName}</td>
                  <td><joda:format value="${contact.birthDate}"
                                   pattern="yyyy-MM-dd"/></td>
               </tr>
            </c:forEach>
            </tbody>
         </table>
      </c:if>
</div>
```

В листинге 16.30 с использованием дескриптора `<spring:url>` мы объявляем переменную URL и добавляем ссылку для атрибута `firstName`. Чтобы протестировать представление просмотра контакта, после повторного построения и развертывания откройте представление списка контактов снова. Список теперь должен включать гиперссылки для отображения представления с информацией о контакте. Щелчок на любой такой ссылке приводит к перемещению на представление просмотра контакта.

## Реализация представления для редактирования контакта

Давайте реализуем представление для редактирования контакта. Это то же самое представление, что и применяемое при просмотре контакта; для начала мы добавим в класс `ContactController` методы `updateForm()` и `update()`. В листинге 16.31 показан модифицированный код контроллера с двумя новыми методами.

**Листинг 16.31. Код контроллера, измененный для обновления контакта**

```
package com.apress.prospring4.ch16;

import java.util.List;
import java.util.Locale;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
```

```
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import javax.servlet.http.HttpServletRequest;
@RequestMapping("/contacts")
@Controller
public class ContactController {
   private final Logger logger =
        LoggerFactory.getLogger(ContactController.class);
   private ContactService contactService;
   private MessageSource messageSource;
   @RequestMapping(method = RequestMethod.GET)
   public String list(Model uiModel) {
      logger.info("Listing contacts");
      List<Contact> contacts = contactService.findAll();
      uiModel.addAttribute("contacts", contacts);
      logger.info("No. of contacts: " + contacts.size());
      return "contacts/list";
   }
   @RequestMapping(value = "/{id}", method = RequestMethod.GET)
   public String show(@PathVariable("id") Long id, Model uiModel) {
      Contact contact = contactService.findById(id);
      uiModel.addAttribute("contact", contact);
      return "contacts/show";
   }
   @RequestMapping(value = "/{id}", params = "form", method = RequestMethod.POST)
   public String update(Contact contact, BindingResult bindingResult,
          Model uiModel,  HttpServletRequest httpServletRequest,
          RedirectAttributes redirectAttributes, Locale locale) {
      logger.info("Updating contact");
      if (bindingResult.hasErrors()) {
         uiModel.addAttribute("message", new Message("error",
               messageSource.getMessage("contact_save_fail",
                                        new Object[]{}, locale)));
         uiModel.addAttribute("contact", contact);
         return "contacts/update";
      }
      uiModel.asMap().clear();
      redirectAttributes.addFlashAttribute("message",
          new Message("success",
            messageSource.getMessage("contact_save_success",
               new Object[]{}, locale)));
      contactService.save(contact);
      return "redirect:/contacts/"
         + UrlUtil.encodeUrlPathSegment(contact.getId().toString(),
                                        httpServletRequest);
   }
   @RequestMapping(value = "/{id}", params = "form", method = RequestMethod.GET)
   public String updateForm(@PathVariable("id") Long id, Model uiModel) {
      uiModel.addAttribute("contact", contactService.findById(id));
      return "contacts/update";
   }
```

```
    @Autowired
    public void setContactService(ContactService contactService) {
        this.contactService = contactService;
    }
    @Autowired
    public void setMessageSource(MessageSource messageSource) {
        this.messageSource = messageSource;
    }
}
```

Ниже перечислены основные моменты, связанные с листингом 16.31.

- Интерфейс `MessageSource` автоматически связан с контроллером для извлечения сообщений с поддержкой интернационализации.
- В методе `updateForm()` контакт извлекается и сохраняется в модели, после чего возвращается логическое имя представления `contacts/update`, которое отобразит представление редактирования контакта.
- Метод `update()` будет запускаться, когда пользователь обновляет информацию о контакте и щелкает на кнопке Save (Сохранить). Этот метод требует некоторых пояснений. Сначала Spring MVC попытается связать отправленные данные с объектом предметной области `Contact` и автоматически выполнит преобразование типов и форматирование. В случае обнаружения ошибок привязки (например, дата рождения была введена в неверном формате) эти ошибки будут сохранены в интерфейсе `BindingResult` (из пакета `org.springframework.validation`), а сообщения об ошибках — в модели, приводя к повторному отображению представления редактирования. Если привязка прошла успешно, данные сохраняются, и возвращается логическое имя представления просмотра контакта с указанием `redirect:` в качестве префикса. Обратите внимание, что нам нужно отобразить сообщение после перенаправления, поэтому мы должны использовать метод `RedirectAttributes.addFlashAttribute()` (из интерфейса в пакете `org.springframework.web.servlet.mvc.support`) для отображения сообщения об успехе в представлении просмотра контакта. Flash-атрибуты в Spring MVC временно сохраняются перед перенаправлением (обычно в сеансе) и будут доступны запросу после перенаправления, а затем они немедленно удаляются.
- Класс `Message` — это специальный класс, который хранит сообщение, извлеченное из `MessageSource`, и тип сообщения (т.е. успех или ошибка) для отображения в области сообщений представления. Код класса `Message` приведен в листинге 16.32.
- Класс `UrlUtil` — это служебный класс, кодирующий URL для перенаправления. Код этого класса показан в листинге 16.33.

**Листинг 16.32. Класс `Message`**

```
package com.apress.prospring4.ch16;

public class Message {
    private String type;
    private String message;
```

```
    public Message(String type, String message) {
        this.type = type;
        this.message = message;
    }
    public String getType() {
        return type;
    }
    public String getMessage() {
        return message;
    }
}
```

**Листинг 16.33. Класс `UrlUtil`**

```
package com.apress.prospring4.ch16;

import java.io.UnsupportedEncodingException;

import javax.servlet.http.HttpServletRequest;

import org.springframework.web.util.UriUtils;
import org.springframework.web.util.WebUtils;

public class UrlUtil {
    public static String encodeUrlPathSegment(String pathSegment,
HttpServletRequest httpServletRequest) {
        String enc = httpServletRequest.getCharacterEncoding();
        if (enc == null) {
            enc = WebUtils.DEFAULT_CHARACTER_ENCODING;
        }
        try {
            pathSegment = UriUtils.encodePathSegment(pathSegment, enc);
        } catch (UnsupportedEncodingException uee) {
            //
        }
        return pathSegment;
    }
}
```

А теперь рассмотрим собственно представление редактирования контакта (`/views/contacts/edit.jspx`), которое будет использоваться как при обновлении существующего, так и при создании нового контакта. Код этого представления приведен в листинге 16.34.

**Листинг 16.34. Представление редактирования контакта**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:spring="http://www.springframework.org/tags"
    xmlns:form="http://www.springframework.org/tags/form"
    version="2.0">
    <jsp:directive.page contentType="text/html;charset=UTF-8"/>
    <jsp:output omit-xml-declaration="yes"/>
```

```
<spring:message code="label_contact_new" var="labelContactNew"/>
<spring:message code="label_contact_update" var="labelContactUpdate"/>
<spring:message code="label_contact_first_name"
                var="labelContactFirstName"/>
<spring:message code="label_contact_last_name"
                var="labelContactLastName"/>
<spring:message code="label_contact_birth_date"
                var="labelContactBirthDate"/>
<spring:message code="label_contact_description"
                var="labelContactDescription"/>
<spring:message code="label_contact_photo" var="labelContactPhoto"/>

<spring:eval expression="contact.id == null ?
            labelContactNew:labelContactUpdate"
        var="formTitle"/>

<h1>${formTitle}</h1>

<div id="contactUpdate">
<form:form modelAttribute="contact" id="contactUpdateForm" method="post">

   <c:if test="${not empty message}">
     <div id="message" class="${message.type}">${message.message}</div>
   </c:if>

   <form:label path="firstName">
      ${labelContactFirstName}*
   </form:label>
   <form:input path="firstName" />
   <div>
      <form:errors path="firstName" cssClass="error" />
   </div>
   <p/>

   <form:label path="lastName">
      ${labelContactLastName}*
   </form:label>
   <form:input path="lastName" />
   <div>
      <form:errors path="lastName" cssClass="error" />
   </div>
   <p/>

   <form:label path="birthDate">
      ${labelContactBirthDate}
   </form:label>
   <form:input path="birthDate" id="birthDate"/>
   <div>
      <form:errors path="birthDate" cssClass="error" />
   </div>
   <p/>

   <form:label path="description">
      ${labelContactDescription}
   </form:label>
   <form:textarea cols="60" rows="8" path="description"
                  id="contactDescription"/>
```

```
        <div>
            <form:errors path="description" cssClass="error" />
        </div>
        <p/>

        <form:hidden path="version" />

        <button type="submit">Save</button>
        <button type="reset">Reset</button>

    </form:form>
    </div>
</div>
```

Ниже описаны основные аспекты, касающиеся листинга 16.34.

- Внутри дескриптора `<spring:eval>` используется язык SpEL (Spring Expression Language — язык выражений Spring) для проверки, равен ли идентификатор контакта (`contact.id`) значению `null`. Если это так, значит, создается новый контакт, а если нет, то обновляется существующий контакт. Будет отображен соответствующий заголовок формы.
- Внутри формы с помощью различных дескрипторов `<form>` из Spring MVC отображаются метка, поле ввода и сообщения об ошибках, если при отправке формы привязка завершилась сбоем.

Далее мы добавим отображение представления в файл определения представлений (`/views/contacts/views.xml`). Соответствующий фрагмент кода показан в листинге 16.35.

**Листинг 16.35. Отображение для представления редактирования контакта**

```
<definition extends="default" name="contacts/update">
  <put-attribute name="body" value="/WEB-INF/views/contacts/edit.jspx" />
</definition>
```

На этом представление редактирования контакта завершено. Повторно постройте и разверните проект. После щелчка на ссылке для редактирования отобразится представление редактирования контакта. Обновите информацию о контакте и щелкните на кнопке Save (Сохранить). Если привязка прошла успешно, вы увидите сообщение об успешном сохранении информации о контакте, а затем отобразится представления просмотра контакта.

## Реализация представления для добавления контакта

Реализация представления для добавления контакта очень похожа на реализацию представления для редактирования. Поскольку мы будем повторно использовать страницу `edit.jspx`, понадобится только добавить методы в класс `ContactController` и определение представления.

В листинге 16.36 приведен пересмотренный код класса `ContactController`.

**Листинг 16.36. Пересмотренный код класса `ContactController` с методами для добавления контакта**

```
package com.apress.prospring4.ch16;

import java.util.List;
import java.util.Locale;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import javax.servlet.http.HttpServletRequest;

@RequestMapping("/contacts")
@Controller
public class ContactController {
    private final Logger logger = LoggerFactory.getLogger(ContactController.class);

    private ContactService contactService;
    private MessageSource messageSource;

    @RequestMapping(method = RequestMethod.GET)
    public String list(Model uiModel) {
        logger.info("Listing contacts");

        List<Contact> contacts = contactService.findAll();
        uiModel.addAttribute("contacts", contacts);

        logger.info("No. of contacts: " + contacts.size());

        return "contacts/list";
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public String show(@PathVariable("id") Long id, Model uiModel) {
        Contact contact = contactService.findById(id);
        uiModel.addAttribute("contact", contact);
        return "contacts/show";
    }
    @RequestMapping(value = "/{id}", params = "form", method = RequestMethod.POST)
    public String update(Contact contact, BindingResult bindingResult,
                         Model uiModel,HttpServletRequest httpServletRequest,
                         RedirectAttributes redirectAttributes, Locale locale) {
        logger.info("Updating contact");
        if (bindingResult.hasErrors()) {
            uiModel.addAttribute("message", new Message("error",
                    messageSource.getMessage("contact_save_fail",
                                             new Object[]{}, locale)));
            uiModel.addAttribute("contact", contact);
            return "contacts/update";
        }
```

```
        uiModel.asMap().clear();
        redirectAttributes.addFlashAttribute("message",
                new Message("success",
                messageSource.getMessage("contact_save_success",
                new Object[]{}, locale)));
        contactService.save(contact);
        return "redirect:/contacts/" +
                UrlUtil.encodeUrlPathSegment(contact.getId().toString(),
                httpServletRequest);
    }
    @RequestMapping(value = "/{id}", params = "form", method = RequestMethod.GET)
    public String updateForm(@PathVariable("id") Long id, Model uiModel) {
        uiModel.addAttribute("contact", contactService.findById(id));
        return "contacts/update";
    }
    @RequestMapping(params = "form", method = RequestMethod.POST)
    public String create(Contact contact, BindingResult bindingResult,
          Model uiModel, HttpServletRequest httpServletRequest,
          RedirectAttributes redirectAttributes, Locale locale) {
        logger.info("Creating contact");
        if (bindingResult.hasErrors()) {
            uiModel.addAttribute("message", new Message("error",
                    messageSource.getMessage("contact_save_fail",
                    new Object[]{}, locale)));
            uiModel.addAttribute("contact", contact);
            return "contacts/create";
        }
        uiModel.asMap().clear();
        redirectAttributes.addFlashAttribute("message",
                new Message("success",
                messageSource.getMessage("contact_save_success",
                new Object[]{}, locale)));
        logger.info("Contact id: " + contact.getId());
        contactService.save(contact);
        return "redirect:/contacts/" +
                UrlUtil.encodeUrlPathSegment(contact.getId().toString(),
                httpServletRequest);
    }
    @RequestMapping(params = "form", method = RequestMethod.GET)
    public String createForm(Model uiModel) {
        Contact contact = new Contact();
        uiModel.addAttribute("contact", contact);
        return "contacts/create";
    }
    @Autowired
    public void setContactService(ContactService contactService) {
        this.contactService = contactService;
    }
    @Autowired
    public void setMessageSource(MessageSource messageSource) {
        this.messageSource = messageSource;
    }
}
```

Далее добавим отображение представления в файл определения представлений (`/views/contacts/views.xml`). В листинге 16.37 приведен необходимый фрагмент кода.

**Листинг 16.37. Отображение представления для добавления контакта**

```
<definition extends="default" name="contacts/create">
  <put-attribute name="body" value="/WEB-INF/views/contacts/edit.jspx" />
</definition>
```

На этом представление добавления контакта завершено. После повторного построения и развертывания проекта щелкните на ссылке для создания контакта в области меню. Отобразится представление добавления контакта, позволяя ввести сведения о новом контакте.

## Включение проверки достоверности бинов JSR-349

Давайте сконфигурируем поддержку проверки достоверности бинов JSR-349 для действий создания и обновления контакта. Прежде всего, применим ограничения проверки к объекту предметной области `Contact`. В рассматриваемом примере мы определяем ограничения только для атрибутов `firstName` и `lastName`. В листинге 16.38 представлен модифицированный код класса `Contact` с аннотациями, примененными к атрибутам `firstName` и `lastName`.

**Листинг 16.38. Применение ограничений к объекту предметной области `Contact`**

```
package com.apress.prospring4.ch16;

import static javax.persistence.GenerationType.IDENTITY;

import java.io.Serializable;

import org.hibernate.annotations.Type;
import org.hibernate.validator.constraints.NotEmpty;
import org.joda.time.DateTime;
import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.format.annotation.DateTimeFormat.ISO;

import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Lob;
import javax.persistence.Table;
import javax.persistence.Transient;
import javax.persistence.Version;
import javax.validation.constraints.Size;

@Entity
@Table(name = "contact")
public class Contact implements Serializable {
    private Long id;
    private int version;
```

```
private String firstName;
private String lastName;
private DateTime birthDate;
private String description;
private byte[] photo;

@Id
@GeneratedValue(strategy = IDENTITY)
@Column(name = "ID")
public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

@Version
@Column(name = "VERSION")
public int getVersion() {
    return version;
}

public void setVersion(int version) {
    this.version = version;
}

@NotEmpty(message="{validation.firstname.NotEmpty.message}")
@Size(min=3, max=60, message="{validation.firstname.Size.message}")
@Column(name = "FIRST_NAME")
public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

@NotEmpty(message="{validation.lastname.NotEmpty.message}")
@Size(min=1, max=40, message="{validation.lastname.Size.message}")
@Column(name = "LAST_NAME")
public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

@Column(name = "BIRTH_DATE")
@Type(type="org.jadira.usertype.dateandtime.joda.PersistentDateTime")
@DateTimeFormat(iso=ISO.DATE)
public DateTime getBirthDate() {
    return birthDate;
}

public void setBirthDate(DateTime birthDate) {
    this.birthDate = birthDate;
}
```

```
    @Column(name = "DESCRIPTION")
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    @Basic(fetch= FetchType.LAZY)
    @Lob
    @Column(name = "PHOTO")
    public byte[] getPhoto() {
        return photo;
    }
    public void setPhoto(byte[] photo) {
        this.photo = photo;
    }
    @Transient
    public String getBirthDateString() {
        String birthDateString = "";
        if (birthDate != null)
            birthDateString = org.joda.time.format.DateTimeFormat
                    .forPattern("yyyy-MM-dd").print(birthDate);
        return birthDateString;
    }
    @Override
    public String toString() {
        return "Contact - Id: " + id + ", First name: " + firstName
                + ", Last name: " + lastName + ", Birthday: " + birthDate
                + ", Description: " + description;
    }
}
```

Ограничения применены к соответствующим методам. Обратите внимание, что для представления сообщений проверки достоверности используется код в фигурных скобках. Это приводит к извлечению таких сообщений из пакета ресурсов и обеспечивает поддержку интернационализации.

Чтобы включить проверку достоверности JSR-349 во время процесса привязки данных, нужно просто применить аннотацию `@Valid` к аргументу методов `create()` и `update()` в классе `ContactController`. Фрагмент кода с обоими методами приведен в листинге 16.39.

**Листинг 16.39. Включение проверки достоверности JSR-349 в контроллере**

```
public String update(@Valid Contact contact, ...
public String create(@Valid Contact contact, ...
```

Также требуется, чтобы для сообщения проверки достоверности JSR-349 использовался тот же пакет ресурсов, что и в представлениях. Для этого необходимо сконфигурировать проверку в сервлете диспетчера (файл `servlet-context.xml`). В листинге 16.40 показано, как должен быть изменен код.

**Листинг 16.40. Конфигурирование поддержки JSR-349 в Spring MVC**

```
<?xml version="1.0" encoding="UTF-8"?>
   <!-- Остальной код не показан -->
   <annotation-driven validator="validator"/>
   <!-- Остальной код не показан -->
   <beans:bean id="validator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
      <beans:property name="validationMessageSource" ref="messageSource"/>
   </beans:bean>
</beans:beans>
```

Вначале для поддержки JSR-349 определяется бин `validator` с классом `LocalValidatorFactoryBean`. Обратите внимание на установку свойства `validationMessageSource` для ссылки на определенный бин `messageSource`, который обеспечивает поиск при проверке JSR-349 сообщений по кодам из бина `messageSource`. Затем в дескрипторе `<annotation-driven>` атрибут `validator` явно ссылается на ранее определенный бин `validator`.

Теперь можно протестировать проверку достоверности. В представлении добавления контакта щелкните на кнопке Save (Сохранить). На возвращенной странице отобразится сообщение об ошибке при проверке достоверности.

Переключитесь на китайский язык (для Гонконга) и сделайте то же самое. На этот раз сообщения будут отображаться на китайском языке.

На этом представления в основном завершены, за исключением действия по удалению. Мы оставляем его реализацию в качестве упражнения для самостоятельного выполнения. А теперь приступим к расширению нашего приложения.

# Использование jQuery и jQuery UI

Хотя представления в нашем приложении контактов функционируют вполне нормально, пользовательский интерфейс несколько сыроват. Например, было бы намного удобнее предусмотреть для поля с датой рождения средство выбора даты, а не заставлять пользователя вводить строку даты вручную.

Если только вы не имеете дело с технологиями RIA (Rich Internet Application — насыщенное Интернет-приложение), которые требуют наличия в клиентском веб-браузере специальных исполняющих сред (например, Adobe Flex требует Flash, JavaFX — JRE, а Microsoft Silverlight — Silverlight), то для предоставления пользователям веб-приложения более развитого интерфейса желаемые средства придется реализовывать с помощью JavaScript.

Однако разработка пользовательских интерфейсов для веб-приложений на чистом языке JavaScript является непростой задачей. Синтаксис языка JavaScript значительно отличается от синтаксиса Java, к тому же придется решать проблемы межбраузерной совместимости. Тем не менее, доступно множество JavaScript-библиотек с открытым кодом, которые существенно упрощают данный процесс; среди них можно отметить jQuery и Dojo Toolkit.

В последующих разделах мы обсудим применение jQuery и jQuery UI для разработки более отзывчивых и интерактивных пользовательских интерфейсов. Мы также

рассмотрим некоторые часто используемые подключаемые модули jQuery, предназначенные для специфичных целей, таких как поддержка редактирования форматированного текста, и представим ряд компонентов, основанных на сетке, которые применяются для просмотра данных.

## Введение в jQuery и jQuery UI

jQuery (`http://jquery.org`) — это одна из наиболее популярных JavaScript-библиотек, используемых при разработке пользовательских интерфейсов для веб-приложений. Библиотека jQuery предлагает обширную поддержку основных функциональных возможностей, включая надежный синтаксис "селектора" для выбора элементов DOM-модели внутри документа, развитую модель событий и мощную поддержку Ajax.

Построенная поверх jQuery библиотека jQuery UI (`http://jqueryui.com`) предоставляет широкий набор виджетов (графических элементов) и эффектов. Основные средства включают виджеты для часто применяемых компонентов пользовательского интерфейса (выбор даты, автозавершение, раскрывающийся список и т.д.), перетаскивания, визуальных эффектов и анимации, оформления темами и многих других функций.

Существует также многообразие подключаемых модулей jQuery, разработанных сообществом jQuery для специальных целей, и два из них мы обсудим в этой главе.

В главе мы только слегка коснемся поверхности jQuery. За более подробными сведениями о jQuery обращайтесь к книге *jQuery 2.0 для профессионалов* (ИД "Вильямс", 2014 г.).

## Активизация jQuery и jQuery UI в представлении

Чтобы можно было пользоваться компонентами jQuery и jQuery UI в представлении, необходимо включить требуемые файлы стилевых таблиц и JavaScript-кода.

Если вы проработали раздел "Обзор структуры проекта Spring MVC" ранее в этой главе, то обязательные файлы уже должны быть скопированы в проект. Ниже перечислены основные файлы, которые должны быть включены в представление.

- `/src/main/webapp/scripts/jquery-1.11.1.js`. Это JavaScript-библиотека ядра jQuery. В настоящей главе применяется ее версия 1.11.1. Обратите внимание, что это полная исходная версия. В производственной среде должна использоваться уменьшенная версия библиотеки (т.е. `jquery-1.11.1.min.js`), которая оптимизирована и сжата с целью сокращения времени загрузки и улучшения производительности выполнения.
- `/src/main/webapp/scripts/jquery-ui-1.10.4.custom.min.js`. Это библиотека jQuery UI, упакованная со стилевой таблицей темы, которая может быть настроена и загружена со страницы Themeroller (`http://jqueryui.com/themeroller`). Здесь мы работаем с версией jQuery UI 1.10.4. Обратите внимание, что это уменьшенная версия JavaScript.
- `/src/main/webapp/styles/custom-theme/jquery-ui-1.10.4.custom.css`. Это стилевая таблица для специальной темы, которая будет использоваться jQuery UI для поддержки оформления темами.

Описанные выше файлы должны быть включены только в страницу шаблона (т.е. `/layouts/default.jspx`). В листинге 16.41 показан фрагмент кода, который понадобится добавить к этой странице.

**Листинг 16.41. Включение jQuery в страницу шаблона**

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
   <!-- Остальной код не показан -->

   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <meta http-equiv="X-UA-Compatible" content="IE=8" />
      <spring:theme code="styleSheet" var="app_css" />
      <spring:url value="/${app_css}" var="app_css_url" />
      <link rel="stylesheet" type="text/css" media="screen"
        href="${app_css_url}" />
      <!-- jQuery и jQuery UI -->
      <spring:url value="/resources/scripts/jquery-1.11.1.js"
        var="jquery_url" />
      <spring:url value="/resources/scripts/jquery-ui-1.10.4.custom.min.js"
         var="jquery_ui_url" />
      <spring:url
        value="/resources/styles/custom-theme/jquery-ui-1.10.4.custom.css"
        var="jquery_ui_theme_css" />
      <link rel="stylesheet" type="text/css" media="screen"
        href="${jquery_ui_theme_css}" />
      <script src="${jquery_url}" type="text/javascript"><jsp:text/>
      </script>
      <script src="${jquery_ui_url}" type="text/javascript"><jsp:text/>
      </script>
   <!-- Остальной код не показан -->
</html>
```

Сначала с помощью дескриптора `<spring:url>` определяются URL для файлов и сохраняются в переменных. Затем в разделе `<head>` добавляется ссылка на файлы CSS и JavaScript. Обратите внимание на использование дескриптора `<jsp:text/>` внутри `<script>`. Это объясняется тем, что JSPX будет автоматически сворачивать дескрипторы, не имеющие тела. Таким образом, дескриптор `<script ...></script>` в файле превратится в `<script .../>` в браузере, что приведет к неопределенному поведению страницы. Добавление `<jsp:text/>` гарантирует, что дескриптор `<script>` не будет визуализирован на странице, поскольку это позволит избежать неожиданных проблем.

Благодаря включению этих сценариев, мы можем добавить к представлению некоторые интересные вещи. Давайте в представлении редактирования контакта немного улучшим внешний вид кнопок и включим компонент выбора даты в поле ввода даты рождения. В листинге 16.42 показаны изменения, которые необходимо внести в представление `/views/contacts/edit.jspx`.

**Листинг 16.42. Декорирование кнопок и добавление компонента выбора даты в представление редактирования контакта**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:spring="http://www.springframework.org/tags"
    xmlns:form="http://www.springframework.org/tags/form"
    version="2.0">
   <script type="text/javascript">
   $(function(){
      $('#birthDate').datepicker({
         dateFormat: 'yy-mm-dd',
         changeYear: true
      });
   });
   </script>
   <!-- Остальной код не показан -->
   <button type="submit"
class="ui-button ui-widget ui-state-default ui-corner-all ui-button-text-only">
      <span class="ui-button-text">Save</span>
   </button>
   <button type="reset"
class="ui-button ui-widget ui-state-default ui-corner-all ui-button-text-only">
      <span class="ui-button-text">Reset</span>
   </button>
   <!-- Остальной код не показан -->
</div>
```

Синтаксис `$(function(){})` указывает библиотеке jQuery на то, что сценарий должен запускаться, когда документ готов. Внутри этой функции поле ввода даты рождения (с идентификатором `birthDate`) декорируется с применением функции `datepicker()` из jQuery UI. Затем к кнопкам добавляются разнообразные классы стилей.

Развернув приложение повторно, вы увидите новый стиль кнопок, и при щелчке на поле ввода даты рождения отображается компонент выбора даты.

## Редактирование форматированного текста с помощью CKEditor

Для поля описания в рамках информации о контакте мы используем дескриптор `<form:textarea>` из Spring MVC, поддерживающий многострочный ввод. Предположим, что мы хотим разрешить редактирование форматированного текста, которое является распространенным требованием при вводе длинных текстов, таких как пользовательские комментарии.

Для поддержки этой возможности мы будем применять библиотеку компонентов форматированного текста CKEditor (`http://ckeditor.com`), которая предлагает общий JavaScript-компонент форматированного текста и поддерживает интеграцию с jQuery UI. Файлы находятся в папке `/src/main/webapp/ckeditor` исходного кода примера.

Сначала необходимо включить обязательные JavaScript-файлы в страницу шаблона (`default.jspx`). В листинге 16.43 показан фрагмент кода, который должен быть добавлен к странице.

**Листинг 16.43. Добавление CKEditor к странице шаблона**

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:fn="http://java.sun.com/jsp/jstl/functions"
     xmlns:tiles="http://tiles.apache.org/tags-tiles"
     xmlns:spring="http://www.springframework.org/tags">
      <!-- Остальной код не показан -->
      <!-- jQuery и jQuery UI -->
      <!-- CKEditor -->
      <spring:url value="/resources/ckeditor/ckeditor.js" var="ckeditor_url" />
      <spring:url value="/resources/ckeditor/adapters/jquery.js"
         var="ckeditor_jquery_url" />
      <script type="text/javascript" src="${ckeditor_url}"><jsp:text/>
      </script>
      <script type="text/javascript" src="${ckeditor_jquery_url}">
      <jsp:text/></script>
      <!-- Остальной код не показан -->
</html>
```

В листинге 16.43 включены два сценария — сценарий ядра CKEditor и адаптер с jQuery.

Следующий шаг связан с добавлением CKEditor к представлению редактирования контакта. В листинге 16.44 приведены изменения, которые потребуется внести в код страницы `edit.jspx`.

**Листинг 16.44. Добавление CKEditor к представлению редактирования контакта**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
   <!-- Остальной код не показан -->
   <script type="text/javascript">
   $(function(){
      $('#birthDate').datepicker({
         dateFormat: 'yy-mm-dd',
         changeYear: true
      });
      $("#contactDescription").ckeditor(
         {
            toolbar : 'Basic',
            uiColor : '#CCCCCC'
         }
      );
   });
   </script>
   <!-- Остальной код не показан -->
</div>
```

Поле для ввода описания контакта декорируется посредством CKEditor, когда документ готов. Повторно развернув приложение и перейдя на страницу добавления контакта, вы увидите, что в поле описания стала доступной поддержка редактирования форматированного текста.

За подробными сведениями о применении и конфигурировании CKEditor обращайтесь на сайт с документацией по проекту (`http://docs.cksource.com/`).

## Использование jqGrid для построения сетки данных, поддерживающей разбиение на страницы

Текущее представление списка контактов хорошо работает, только если в системе существует относительно небольшое число контактов. Однако с увеличением количества записей до нескольких тысяч и более возникают проблемы с производительностью.

Общее решение предусматривает реализацию компонента сетки данных с поддержкой разбиения на страницы, в котором пользователь просматривает только определенное количество записей; это уменьшает объем данных, передаваемых между браузером и веб-контейнером. В этом разделе мы продемонстрируем реализацию сетки данных с помощью jqGrid (`www.trirand.com/blog`) — популярного компонента сетки данных, основанного на JavaScript. Здесь используется версия 4.6.0 упомянутого компонента.

Кроме того, будет применяться встроенная в jqGrid поддержка разбиения на страницы Ajax, которая инициирует запрос `XMLHttpRequest` для каждой страницы и принимает формат JSON для данных на странице. Таким образом, мы должны добавить в проект зависимость от библиотеки JSON, как описано в табл. 16.6.

**Таблица 16.6. Зависимость Maven для JSON**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.codehaus.jackson` | `jackson-mapper-lgpl` | 1.9.13 | Процессор Jackson JSON для поддержки данных в формате JSON |

В последующих разделах мы покажем, как реализовать поддержку разбиения на страницы на серверной и клиентской стороне. Сначала мы рассмотрим реализацию компонента jqGrid в представлении списка контактов. Затем мы обсудим реализацию разбиения на страницы на стороне сервера, используя поддержку такого разбиения в модуле Spring Data Commons.

### *Активизация jqGrid в представлении списка контактов*

Чтобы сделать доступным компонент jqGrid в представлениях, понадобится включить требуемые файлы JavaScript-кода и стилевых таблиц в страницу шаблона (`default.jspx`). Необходимый фрагмент кода приведен в листинге 16.45.

**Листинг 16.45. Добавление jqGrid в страницу шаблона**

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
   <!-- Остальной код не показан -->
```

```
    <!-- CKEditor -->
    <!-- jqGrid -->
    <spring:url value="/resources/jqgrid/css/ui.jqgrid.css"
          var="jqgrid_css" />
    <spring:url value="/resources/jqgrid/js/il8n/grid.locale-en.js"
          var="jqgrid_locale_url" />
    <spring:url value="/resources/jqgrid/js/jquery.jqGrid.min.js"
          var="jqgrid_url" />
    <link rel="stylesheet" type="text/css" media="screen" href="${jqgrid_css}" />
       <script type="text/javascript" src="${jqgrid_locale_url}"><jsp:text/>
       </script>
    <script type="text/javascript" src="${jqgrid_url}"><jsp:text/></script>
    <!-- Остальной код не показан -->
</html>
```

Первым делом загружается нужный CSS-файл, затем два JavaScript-файла. Первый из них — это сценарий локали (в данном случае используется английский язык), а второй — библиотека ядра jqGrid (`jquery.jqGrid.min.js`).

Следующий шаг связан с изменением представления списка контактов (`list.jspx`) с целью применения jqGrid. Модифицированный код страницы приведен в листинге 16.46.

**Листинг 16.46. Страница списка контактов с jqGrid**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:spring="http://www.springframework.org/tags"
    version="2.0">
   <jsp:directive.page contentType="text/html;charset=UTF-8"/>
   <jsp:output omit-xml-declaration="yes"/>
   <spring:message code="label_contact_list" var="labelContactList"/>
   <spring:message code="label_contact_first_name"
                   var="labelContactFirstName"/>
   <spring:message code="label_contact_last_name"
                   var="labelContactLastName"/>
   <spring:message code="label_contact_birth_date"
                   var="labelContactBirthDate"/>
   <spring:url value="/contacts/" var="showContactUrl"/>
   <script type="text/javascript">
   $(function(){
     $("#list").jqGrid({
       url:'${showContactUrl}/listgrid',
       datatype: 'json',
       mtype: 'GET',
       colNames:['${labelContactFirstName}', '${labelContactLastName}',
          '${labelContactBirthDate}'],
       colModel :[
         {name:'firstName', index:'firstName', width:150},
         {name:'lastName', index:'lastName', width:100},
         {name:'birthDateString', index:'birthDate', width:100}
       ],
```

```
        jsonReader : {
            root:"contactData",
            page: "currentPage",
            total: "totalPages",
            records: "totalRecords",
            repeatitems: false,
            id: "id"
        },
        pager: '#pager',
        rowNum:10,
        rowList:[10,20,30],
        sortname: 'firstName',
        sortorder: 'asc',
        viewrecords: true,
        gridview: true,
        height: 250,
        width: 500,
        caption: '${labelContactList}',
        onSelectRow: function(id){
            document.location.href ="${showContactUrl}/" + id;
        }
      });
    });
    </script>

    <c:if test="${not empty message}">
        <div id="message" class="${message.type}">${message.message}</div>
    </c:if>
    <h2>${labelContactList}</h2>
    <div>
        <table id="list"><tr><td/></tr></table>
    </div>
    <div id="pager"></div>
</div>
```

Для отображения данных сетки мы объявляем дескриптор `<table>` с идентификатором `list`. Под таблицей определен раздел `<div>` с идентификатором `pager`, который представляет собой часть jqGrid, отвечающую за разбиение на страницы.

В коде JavaScript, когда документ готов, мы указываем jqGrid на необходимость декорирования таблицы с идентификатором `list` как сетки, а также предоставляем конфигурационную информацию. Ниже отмечены некоторые важные аспекты этих сценариев.

- Атрибут `url` указывает ссылку для отправки запроса `XMLHttpRequest`, который получает данные для текущей страницы.
- Атрибут `datatype` указывает формат данных, которым в этом случае является JSON. Компонент jqGrid также поддерживает формат XML.
- Атрибут `mtype` определяет используемый HTTP-метод, в этом случае `GET`.
- Атрибут `colNames` определяет заголовок столбца для данных, который должен отображаться в сетке, тогда как атрибут `colModel` устанавливает детали для каждой строки данных.

- Атрибут `jsonReader` определяет формат данных JSON, который будет возвращаться сервером.
- Атрибут `pager` включает поддержку разбиения на страницы.
- Атрибут `onSelectRow` определяет действие, которое должно выполняться при выборе строки. В этом случае мы направляем пользователя на представление, отображающее контакт с заданным идентификатором.

За подробным описанием конфигурирования и использования компонента jqGrid обращайтесь на сайт с документацией по проекту (`www.trirand.com/jqgridwiki/doku.php?id=wiki:jqgriddocs`).

### *Включение разбиения на страницы на стороне сервера*

На стороне сервера для реализации разбиения на страницы потребуется выполнить несколько шагов. Мы будем использовать поддержку разбиения на страницы из модуля Spring Data Commons. Для ее включения нужно только изменить интерфейс `ContactRepository`, чтобы он расширял интерфейс `PagingAndSortingRepository<T,ID extends Serializable>` вместо `CrudRepository<T,ID extends Serializable>`. В листинге 16.47 показан пересмотренный код интерфейса `ContactRepository`.

**Листинг 16.47. Модифицированный интерфейс `ContactRepository`**

```
package com.apress.prospring4;

import org.springframework.data.repository.PagingAndSortingRepository;

public interface ContactRepository
   extends PagingAndSortingRepository<Contact, Long> {
}
```

Далее в интерфейс `ContactService` необходимо добавить новый метод для поддержки постраничного извлечения данных. Измененный код интерфейса приведен в листинге 16.48.

**Листинг 16.48. Пересмотренный интерфейс `ContactService`**

```
package com.apress.prospring4.ch16;

import java.util.List;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

public interface ContactService {
    List<Contact> findAll();
    Contact findById(Long id);
    Contact save(Contact contact);
    Page<Contact> findAllByPage(Pageable pageable);
}
```

В листинге 16.48 видно, что был добавлен метод `findAllByPage()`, принимающий экземпляр интерфейса `Pageable` в качестве аргумента. В листинге 16.49 представлена реализация метода `findAllByPage()` в классе `ContactServiceImpl`.

Этот метод возвращает экземпляр реализации интерфейса `Page<T>` (который принадлежит Spring Data Commons и находится в пакете `org.springframework.data.domain`).

**Листинг 16.49. Модифицированный класс `ContactServiceImpl`**

```
package com.apress.prospring4.ch16;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.google.common.collect.Lists;

@Repository
@Transactional
@Service("contactService")
public class ContactServiceImpl implements ContactService {
    private ContactRepository contactRepository;

    @Override
    @Transactional(readOnly=true)
    public List<Contact> findAll() {
        return Lists.newArrayList(contactRepository.findAll());
    }

    @Override
    @Transactional(readOnly=true)
    public Contact findById(Long id) {
        return contactRepository.findOne(id);
    }

    @Override
    public Contact save(Contact contact) {
        return contactRepository.save(contact);
    }

    @Autowired
    public void setContactRepository(ContactRepository contactRepository) {
        this.contactRepository = contactRepository;
    }

    @Override
    @Transactional(readOnly=true)
    public Page<Contact> findAllByPage(Pageable pageable) {
        return contactRepository.findAll(pageable);
    }

}
```

В методе `findAllByPage()` мы просто вызываем метод `findAll(Pageable)`, который предоставляется интерфейсом `PagingAndSortingRepository<T,ID extends Serializable>`.

Следующий шаг связан с реализацией в классе ContactController метода, получающего от компонента jqGrid запрос Ajax для страничных данных. Реализация этого метода приведена в листинге 16.50.

**Листинг 16.50. Пересмотренный класс ContactController**

```
package com.apress.prospring4;
// Операторы импорта опущены
@RequestMapping("/contacts")
@Controller
public class ContactController {
    // Остальной код не показан
    @RequestMapping(value = "/listgrid", method = RequestMethod.GET,
       produces="application/json")
    @ResponseBody
    public ContactGrid listGrid(@RequestParam(value = "page",
          required = false) Integer page,
       @RequestParam(value = "rows", required = false) Integer rows,
       @RequestParam(value = "sidx", required = false) String sortBy,
       @RequestParam(value = "sord", required = false) String order) {
       logger.info("Listing contacts for grid with page: {}, rows: {}",
           page, rows);
       logger.info("Listing contacts for grid with sort: {}, order: {}",
           sortBy, order);
       // Обработать поле, по которому производится сортировка
       Sort sort = null;
       String orderBy = sortBy;
       if (orderBy != null && orderBy.equals("birthDateString"))
          orderBy = "birthDate";
       if (orderBy != null && order != null) {
          if (order.equals("desc")) {
             sort = new Sort(Sort.Direction.DESC, orderBy);
          } else
             sort = new Sort(Sort.Direction.ASC, orderBy);
       }
       // Сконструировать страничный запрос для текущей страницы.
       // Примечание: нумерация страниц для Spring Data JPA начинается с 0,
       // тогда как в jqGrid - с 1
       PageRequest pageRequest = null;
       if (sort != null) {
          pageRequest = new PageRequest(page - 1, rows, sort);
       } else {
          pageRequest = new PageRequest(page - 1, rows);
       }
       Page<Contact> contactPage =
          contactService.findAllByPage(pageRequest);
       // Сконструировать сетку, которая вернет данные в формате JSON
       ContactGrid contactGrid = new ContactGrid();
       contactGrid.setCurrentPage(contactPage.getNumber() + 1);
       contactGrid.setTotalPages(contactPage.getTotalPages());
       contactGrid.setTotalRecords(contactPage.getTotalElements());
       contactGrid.setContactData(Lists.newArrayList(contactPage.iterator()));
       return contactGrid;
    }
}
```

Метод `listGrid()` метод обрабатывает запрос Ajax, читает параметры (номер страницы, количество записей на страницу, поле, по которому производится сортировка, порядок сортировки) из запроса (имена параметров в примере кода соответствуют принятым по умолчанию в jqGrid), конструирует экземпляр класса `PageRequest`, реализующий интерфейс `Pageable`, и затем вызывает метод `ContactService.findAllByPage()` для получения страничных данных. После этого создается экземпляр класса `ContactGrid`, который возвращает компонент jqGrid в формате JSON. Код класса `ContactGrid` показан в листинге 16.51.

**Листинг 16.51. Класс `ContactGrid`**

```
package com.apress.prospring4.ch16;

import java.util.List;

public class ContactGrid {
    private int totalPages;
    private int currentPage;
    private long totalRecords;
    private List<Contact> contactData;

    public int getTotalPages() {
        return totalPages;
    }

    public void setTotalPages(int totalPages) {
        this.totalPages = totalPages;
    }

    public int getCurrentPage() {
        return currentPage;
    }

    public void setCurrentPage(int currentPage) {
        this.currentPage = currentPage;
    }

    public long getTotalRecords() {
        return totalRecords;
    }

    public void setTotalRecords(long totalRecords) {
        this.totalRecords = totalRecords;
    }

    public List<Contact> getContactData() {
        return contactData;
    }

    public void setContactData(List<Contact> contactData) {
        this.contactData = contactData;
    }
}
```

Теперь все готово для тестирования нового представления списка контактов. Выполните повторное построение и развертывание проекта и затем загрузите представление списка контактов. Вы должны увидеть расширенное представление с сеткой, отображающей список контактов.

Можете поработать с сеткой, перемещаясь по страницам, устанавливая разное количество записей на странице, изменяя порядок сортировки щелчками на заголовках столбцов и т.д. Интернационализация также поддерживается, поэтому при желании вы можете видеть сетку с метками на китайском языке.

Компонент jqGrid поддерживает фильтрацию данных. Например, данные можно фильтровать по имени, которое содержит строку "Chris", или по дате рождения, попадающей в заданный диапазон.

# Обработка загрузки файлов

Внутри контактной информации имеется поле типа `BLOB`, предназначенное для хранения фотографии, которая может быть загружена на стороне клиента. В этом разделе мы покажем, как реализовать загрузку файлов в Spring MVC.

На протяжении длительного времени стандартная спецификация сервлетов не поддерживала загрузку файлов. В результате для этой цели модуль Spring MVC использовал другие библиотеки (из которых наиболее часто применялась библиотека Apache Commons FileUpload (`http://commons.apache.org/proper/commons-fileupload/`)). Spring MVC имеет встроенную поддержку Apache Commons FileUpload. Однако, начиная с версии Servlet 3.0, загрузка файлов стала встроенной функциональной возможностью веб-контейнера. Сервер Tomcat 7 поддерживает Servlet 3.0, а в версии Spring 3.1 также появилась поддержка загрузки файлов Servlet 3.0.

В последующих разделах мы покажем, как реализовать загрузку файлов с использованием Spring MVC и Servlet 3.0. В табл. 16.7 приведены требуемые зависимости.

**Таблица 16.7. Зависимости Maven для загрузки файлов**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `javax` | `javaee-web-api` | 7.0 | Интерфейс JEE 7.0 Web Profile API, который содержит библиотеку для Servlet 3.1. В конфигурации построения используйте для этой зависимости область видимости `provided` |
| `commons-io` | `commons-io` | 2.4 | Модуль Apache Commons IO предоставляет множество удобных функций для упрощения обработки ввода-вывода в Java |

## Конфигурирование поддержки загрузки файлов

В веб-контейнере, совместимом с Servlet 3.0, и модулем Spring MVC конфигурирование поддержки загрузки файлов выполняется за два шага.

Во-первых, в дескрипторе веб-развертывания (`web.xml`) для определения сервлета диспетчера необходимо добавить раздел `<multipart-config>`. Соответствующий фрагмент кода представлен в листинге 16.52.

**Листинг 16.52. Добавление поддержки загрузки файлов в `web.xml`**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
```

```
    <!-- Остальной код не показан -->
    <!-- Обработка запросов приложения -->
    <servlet>
        <servlet-name>appServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring/appServlet/servlet-context.xml
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        <multipart-config>
            <max-file-size>5000000</max-file-size>
        </multipart-config>
    </servlet>
    <!-- Остальной код не показан -->
</web-app>
```

В версии Servlet 3.0 сервлет, который поддерживает загрузку файлов, должен быть сконфигурирован с помощью дескриптора `<multipart-config>`. Дескриптор `<max-file-size>` управляет максимальным размером файла, разрешенным для загрузки, который составляет 5 Мбайт.

Во-вторых, в `WebApplicationContext` сервлета диспетчера понадобится сконфигурировать бин, реализующий интерфейс `MultipartResolver`. В листинге 16.53 показано определение бина, которое нужно добавить в файл `servlet-context.xml`.

**Листинг 16.53. Конфигурирование поддержки `MultipartResolver` в Spring MVC**

```
    <beans:bean
class="org.springframework.web.multipart.support.StandardServletMultipartResolver"
   id="multipartResolver"/>
```

Обратите внимание на указание класса реализации `StandardServlet MultipartResolver`, который применяется для поддержки собственной загрузки файлов в контейнере Servlet 3.0.

## Изменение представлений для поддержки загрузки файлов

Для добавления поддержки загрузки файлов мы должны модифицировать два представления. Первое из них — представление редактирования (`edit.jspx`) — будет поддерживать загрузку фотографии для контакта, а второе — представление просмотра (`show.jspx`) — будет отображать эту фотографию.

В листинге 16.54 показаны изменения, которые потребуется внести в представление редактирования.

**Листинг 16.54. Представление редактирования контакта с поддержкой загрузки файлов**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
    <!-- Остальной код не показан -->
```

```
   <form:form modelAttribute="contact" id="contactUpdateForm" method="post"
      enctype="multipart/form-data">
      <!-- Остальной код не показан -->
      <form:label path="description">
         ${labelContactDescription}
      </form:label>
      <form:textarea cols="60" rows="8" path="description"
            id="contactDescription"/>
      <div>
         <form:errors path="description" cssClass="error" />
      </div>
      <p/>

      <label for="file">
         ${labelContactPhoto}
      </label>
      <input name="file" type="file"/>
      <p/>

   <!-- Остальной код не показан -->
</div>
```

В листинге 16.54 внутри дескриптора `<form:form>` необходимо включить поддержку загрузки файлов с множественным содержимым, указав атрибут `enctype`. Затем к форме понадобится добавить поле для загрузки файла.

Также потребуется модифицировать представление просмотра для отображения фотографии, связанной с контактом. Изменения, которые должны быть внесены в это представление (`show.jspx`), показаны в листинге 16.55.

**Листинг 16.55. Представление просмотра контакта с поддержкой вывода фотографии**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
   <!-- Остальной код не показан -->

   <spring:message code="label_contact_photo" var="labelContactPhoto"/>
   <spring:url value="/contacts/photo" var="contactPhotoUrl"/>
   <!-- Остальной код не показан -->
         <tr>
            <td>${labelContactDescription}</td>
            <td>${contact.description}</td>
         </tr>
         <tr>
            <td>${labelContactPhoto}</td>
            <td><img src="${contactPhotoUrl}/${contact.id}"></img></td>
         </tr>
   <!-- Остальной код не показан -->
</div>
```

Здесь в таблицу добавлена новая строка для отображения фотографии, в которой указывается URL загрузки фотографии.

## Изменение контроллера для поддержки загрузки файлов

Финальный шаг предусматривает модификацию кода контроллера. Потребуется внести два изменения. Первое изменение касается метода `create()`, который должен принимать загруженный файл в качестве обязательного параметра. Второе изменение связано с реализацией нового метода для загрузки фотографии на основе указанного идентификатора контакта. Пересмотренный код класса `ContactController` приведен в листинге 16.56.

**Листинг 16.56. Модифицированный класс контроллера с поддержкой загрузки файлов**

```
package com.apress.prospring4.ch16;

import javax.servlet.http.Part;

// Остальные операторы импорта опущены
@RequestMapping("/contacts")
@Controller
public class ContactController {

    @RequestMapping(method = RequestMethod.POST)
    public String create(@Valid Contact contact, BindingResult bindingResult,
            Model uiModel, HttpServletRequest httpServletRequest,
            RedirectAttributes redirectAttributes, Locale locale,
            @RequestParam(value="file", required=false) Part file) {
        logger.info("Creating contact");
        if (bindingResult.hasErrors()) {
            uiModel.addAttribute("message", new Message("error",
              messageSource.getMessage("contact_save_fail",
                                       new Object[]{}, locale)));
            uiModel.addAttribute("contact", contact);
            return "contacts/create";
        }
        uiModel.asMap().clear();
        redirectAttributes.addFlashAttribute("message",
            new Message("success",
            messageSource.getMessage("contact_save_success",
                                     new Object[]{}, locale)));
        logger.info("Contact id: " + contact.getId());
        // Обработка загруженного файла
        if (file != null) {
            logger.info("File name: " + file.getName());
            logger.info("File size: " + file.getSize());
            logger.info("File content type: " + file.getContentType());
            byte[] fileContent = null;
            try {
                InputStream inputStream = file.getInputStream();
                if (inputStream == null) logger.info("File inputstream is null");
                    fileContent = IOUtils.toByteArray(inputStream);
                    contact.setPhoto(fileContent);
                } catch (IOException ex) {
                    logger.error("Error saving uploaded file");
                }
            contact.setPhoto(fileContent);
        }
```

```
        contactService.save(contact);
        return "redirect:/contacts/" + UrlUtil.encodeUrlPathSegment(
            contact.getId().toString(), httpServletRequest);
    }

    @RequestMapping(value = "/photo/{id}", method = RequestMethod.GET)
    @ResponseBody
    public byte[] downloadPhoto(@PathVariable("id") Long id) {
        Contact contact = contactService.findById(id);

        if (contact.getPhoto() != null) {
            logger.info("Downloading photo for id: {} with size: {}",
                contact.getId(),
                contact.getPhoto().length);
        }

        return contact.getPhoto();
    }
}
```

Прежде всего, в метод `create()` добавлен аргумент для нового параметра запроса с типом интерфейса `javax.servlet.http.Part`, который Spring MVC предоставит, основываясь на загруженном в запросе содержимом. Затем метод получит содержимое, сохраненное в свойстве `photo` объекта `Contact`.

Кроме того, добавлен новый метод `downloadPhoto()`, предназначенный для обработки загрузки файла. Этот метод просто извлекает поле `photo` из объекта `contact` и напрямую записывает его в поток ответа, что соответствует дескриптору `<img>` в представлении просмотра.

Чтобы протестировать функцию загрузки, разверните повторно приложение и добавьте новый контакт с фотографией. По завершении вы сможете увидеть эту фотографию в представлении просмотра контакта.

Еще потребуется модифицировать функцию редактирования для изменения фотографии, но мы оставляем это в качестве упражнения для самостоятельной проработки.

# Защита веб-приложения с помощью Spring Security

Предположим, что теперь мы хотим защитить наше приложение работы с контактами. Возможностью добавления новых контактов и обновления существующих контактов должны обладать только пользователи, которые вошли в приложение с допустимым пользовательским идентификатором. Другие пользователи, называемые анонимными, могут только просматривать информацию о контактах.

Модуль Spring Security является наилучшим вариантом для защиты приложений, основанных на Spring. Несмотря на использование в основном на уровне презентаций, Spring Security может помочь защитить все уровни в приложении, включая уровень обслуживания. В последующих разделах мы продемонстрируем применение Spring Security для защиты приложения работы с контактами.

В табл. 16.8 перечислены зависимости, требуемые для Spring Security.

**Таблица 16.8. Зависимости Maven для Spring Security**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| org.springframework.security | spring-security-core | 3.2.1.RELEASE | Модуль ядра Spring Security |
| org.springframework.security | spring-security-web | 3.2.1.RELEASE | Веб-модуль Spring Security |
| org.springframework.security | spring-security-config | 3.2.1.RELEASE | Модуль конфигурации Spring Security |
| org.springframework.security | spring-security-taglibs | 3.2.1.RELEASE | Библиотека JSP-дескрипторов Spring Security |

## Конфигурирование Spring Security

Чтобы сконфигурировать Spring Security, сначала понадобится сконфигурировать фильтр в дескрипторе веб-развертывания (`web.xml`). В листинге 16.57 показан фрагмент кода, который должен быть добавлен в файл `web.xml`.

**Листинг 16.57. Конфигурирование фильтра Spring Security**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
   <!-- Остальной код не показан -->

   <!-- Конфигурация Spring Security -->
   <filter>
      <filter-name>springSecurityFilterChain</filter-name>
      <filter-class>org.springframework.web.filter.DelegatingFilterProxy
       </filter-class>
   </filter>

   <filter-mapping>
      <filter-name>springSecurityFilterChain</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>

   <!-- Остальной код не показан -->
</web-app>
```

В листинге 16.57 приведен код фильтра для Spring Security. Следующий шаг заключается в определении контекста Spring Security, который будет импортирован корневым файлом конфигурации `WebApplicationContext`. Содержимое конфигурационного файла `/WEB-INF/spring/security-context.xml` представлено в листинге 16.58.

**Листинг 16.58. Конфигурация контекста Spring Security**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
   xmlns:beans="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/security
     http://www.springframework.org/schema/security/spring-security.xsd">

   <http use-expressions="true">
      <intercept-url pattern='/*' access='permitAll' />
      <form-login login-page="/contacts"
                  authentication-failure-url="/security/loginfail"
         default-target-url="/contacts" />
      <logout logout-success-url="/contacts"/>
   </http>

   <authentication-manager>
      <authentication-provider>
         <user-service>
            <user name="user" password="user" authorities="ROLE_USER" />
         </user-service>
      </authentication-provider>
   </authentication-manager>
</beans:beans>
```

Первым делом с помощью дескриптора `<http>` определяется конфигурация безопасности для HTTP-запросов. Атрибут `use-expressions` означает, что мы планируем использовать для выражений язык SpEL. Дескриптор `<intercept-url>` указывает, что входить в приложение разрешено всем пользователям. Мы покажем, как можно защитить функцию, скрывая опции редактирования в представлении с применением библиотеки дескрипторов Spring Security и защиты методов контроллера. В дескрипторе `<form-login>` определена поддержка формы входа. Как упоминалось при обсуждении компоновки, форма входа будет отображаться слева. Кроме того, будет предоставлена ссылка для выхода из приложения.

Дескриптор `<authentication-manager>` определяет механизм аутентификации. В примере конфигурации жестко закодирован единственный пользователь с назначенной ему ролью `ROLE_USER`. В производственной среде пользователи должны аутентифицироваться с помощью базы данных, LDAP или механизма SSO (single sign-on — единый вход).

В листинге 16.59 приведено пересмотренное содержимое файла `root-context.xml` для импорта файла конфигурации безопасности.

**Листинг 16.59. Модифицированная конфигурация контекста Spring Security**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

   <import resource="classpath:META-INF/spring/datasource-tx-jpa.xml" />
   <import resource="classpath:META-INF/spring/security-context.xml"/>

   <context:component-scan base-package="com.apress.prospring4.ch16" />
</beans>
```

## Добавление к приложению функций входа

Нам потребуется изменить два страничных компонента: заголовок (`header.jspx`) и меню (`menu.jspx`).

В листинге 16.60 показано модифицированное содержимое файла `header.jspx`, который теперь отображает информацию о пользователе, если он вошел в приложение.

**Листинг 16.60. Отображение информации о вошедшем пользователе**

```
<div id="header" xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:spring="http://www.springframework.org/tags"
      xmlns:sec="http://www.springframework.org/security/tags"
      version="2.0">
      <jsp:directive.page contentType="text/html;charset=UTF-8" />
      <jsp:output omit-xml-declaration="yes" />

   <spring:message code="header_text" var="headerText"/>
   <spring:message code="label_logout" var="labelLogout"/>
   <spring:message code="label_welcome" var="labelWelcome"/>
   <spring:url var="logoutUrl" value="/j_spring_security_logout" />

   <div id="appname">
      <h1>${headerText}</h1>
   </div>

   <div id="userinfo">
      <sec:authorize access="isAuthenticated()">${labelWelcome}
         <sec:authentication property="principal.username" />
         <br/>
         <a href="${logoutUrl}">${labelLogout}</a>
      </sec:authorize>
   </div>
</div>
```

Вначале добавляется библиотека дескрипторов Spring Security с префиксом `sec`. Затем добавляется раздел `<div>` с дескриптором `<sec:authorize>` для обнаружения, вошел ли пользователь в приложение. Если это так (т.е. выражение `isAuthenticated()` возвращает `true`), будет отображено имя пользователя и ссылка для выхода из приложения.

В листинге 16.61 приведено пересмотренное содержимое файла `menu.jspx` с добавленной формой входа; ссылка New Contact (Новый контакт) будет отображаться, только если пользователь вошел в приложение.

**Листинг 16.61. Отображение формы входа**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<div id="menu" xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:spring="http://www.springframework.org/tags"
    xmlns:sec="http://www.springframework.org/security/tags"
    version="2.0">
      <jsp:directive.page contentType="text/html;charset=UTF-8" />
      <jsp:output omit-xml-declaration="yes" />

      <spring:message code="menu_header_text" var="menuHeaderText"/>
      <spring:message code="menu_add_contact" var="menuAddContact"/>
      <spring:url value="/contacts?form" var="addContactUrl"/>
   <spring:message code="label_login" var="labelLogin"/>
   <spring:url var="loginUrl" value="/j_spring_security_check" />

   <h3>${menuHeaderText}</h3>
   <sec:authorize access="hasRole('ROLE_USER')">
      <a href="${addContactUrl}"><h3>${menuAddContact}</h3></a>
   </sec:authorize>

   <sec:authorize access="isAnonymous()">
   <div id="login">
     <form name="loginForm" action="${loginUrl}" method="post">
        <table>
           <caption align="left">Login:</caption>
           <tr>
              <td>User Name:</td>
              <td><input type="text" name="j_username"/></td>
           </tr>
           <tr>
              <td>Password:</td>
              <td><input type="password" name="j_password"/></td>
           </tr>
           <tr>
              <td colspan="2" align="center"><input name="submit"
                                                    type="submit"
                                                    value="Login"/>
              </td>
           </tr>
        </table>
     </form>
   </div>
   </sec:authorize>
</div>
```

Во-первых, пункт меню для добавления нового контакта будет отображаться, только если пользователь вошел и имеет назначенную ему роль `ROLE_USER` (как указано в первом дескрипторе `<sec:authorize>`). Во-вторых, если пользователь не вошел (второй дескриптор `<sec:authorize>`, когда выражение `isAnonymous()` возвращает `true`), будет отображена форма входа.

Повторно развернув приложение, вы увидите, что оно отображает форму входа; обратите внимание, что ссылка New Contact не видна.

Введите в полях для имени пользователя и пароля строку `user` и щелкните на кнопке Login (Вход). В области заголовка отобразится информация о пользователе. Кроме того, появится ссылка New Contact.

Потребуется также изменить представление просмотра (`show.jspx`), чтобы отображать ссылку Edit Contact (Редактировать контакт) только для вошедших в приложение пользователей, но мы оставляем это в качестве упражнения.

Как было определено в листинге 16.58, предоставление некорректной информации при входе обрабатывается с помощью URL вида `/security/loginfail`. Таким образом, мы должны реализовать контроллер для поддержки такого сценария. В листинге 16.62 показан класс `SecurityController`.

**Листинг 16.62. Класс `SecurityController`**

```
package com.apress.prospring4.ch16;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.Locale;

@Controller
@RequestMapping("/security")
public class SecurityController {
  private final Logger logger = LoggerFactory.getLogger(SecurityController.class);

  private MessageSource messageSource;

  @RequestMapping("/loginfail")
  public String loginFail(Model uiModel, Locale locale) {
    logger.info("Login failed detected");
    uiModel.addAttribute("message", new Message("error",
      messageSource.getMessage("message_login_fail", new Object[]{}, locale)));
    return "contacts/list";
  }

  @Autowired
  public void setMessageSource(MessageSource messageSource) {
    this.messageSource = messageSource;
  }
}
```

Приведенный выше класс контроллера будет обрабатывать все URL с префиксом `security`, а метод `loginFail()` — сценарий некорректного входа. В этом методе мы сохраняем сообщение о некорректном входе в модели и затем направляем пользователя на домашнюю страницу. Теперь повторно разверните приложение и введите некорректную информацию о пользователе; домашняя страница отобразится снова с сообщением о некорректном входе.

## Использование аннотаций для защиты методов контроллера

Сокрытия ссылки New Contact в меню недостаточно. Например, если ввести соответствующий URL (`http://localhost:8080/contact-webapp/contacts?form`) в браузере напрямую, можно по-прежнему видеть страницу добавления контакта, несмотря на то, что вход в приложение не был совершен. Причина в том, что мы не защитили приложение на уровне URL. Один из способов защиты страницы предусматривает конфигурирование цепочки фильтров Spring Security (в файле `security-context.xml`) для перехвата URL только для аутентифицированных пользователей. Однако в этом случае остальные пользователи не смогут видеть представление списка контактов.

Альтернативный способ решения данной проблемы заключается в применении защиты на уровне методов контроллера, используя поддержку аннотаций Spring Security.

Чтобы включить защиту на уровне методов, мы должны изменить конфигурацию сервлета диспетчера (`servlet-context.xml`), как показано в листинге 16.63.

**Листинг 16.63. Включение защиты на уровне методов**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:security="http://www.springframework.org/schema/security"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd
      http://www.springframework.org/schema/security
      http://www.springframework.org/schema/security/spring-security.xsd">
    <!-- Включение модели программирования @Controller в Spring MVC -->
    <annotation-driven validator="validator"/>

    <!-- Включение защиты на уровне методов контроллера -->
    <security:global-method-security pre-post-annotations="enabled"/>
    <!-- Остальной код не показан -->

</beans:beans>
```

В листинге 16.63 сначала добавляется пространство имен `security`. Затем с помощью дескриптора `<security:global-method-security>` включается защита на уровне методов Spring Security, а посредством атрибута `pre-post-annotations` обеспечивается поддержка аннотаций.

Теперь можно воспользоваться аннотацией `@PreAuthorize` для защиты необходимого метода контроллера. В листинге 16.64 приведен пример защиты метода `createForm()`.

**Листинг 16.64. Применение аннотации Spring Security к методу класса `ContactController`**

```
@PreAuthorize("isAuthenticated()")
@RequestMapping(params = "form", method = RequestMethod.GET)
public String createForm(Model uiModel) {
    Contact contact = new Contact();
    uiModel.addAttribute("contact", contact);
    return "contacts/create";
}
```

Здесь мы применяем аннотацию `@PreAuthorize` (из пакета `org.springframework.security.access.prepost`) для защиты метода `createForm()` с аргументом, который является выражением для требований безопасности.

Теперь можно попробовать напрямую ввести URL страницы добавления нового контакта, и если вы не вошли в приложение, то Spring Security перенаправит на страницу входа, которой является представление списка контактов, как было сконфигурировано в файле `security-context.xml`.

# Поддержка конфигурации на основе кода для Servlet 3

Платформа Spring также поддерживает конфигурацию на основе кода для Servlet 3.0, которая представляет собой альтернативу XML-конфигурации, требуемой в файле дескриптора веб-развертывания (`web.xml`). В этом разделе мы покажем, как использовать Java-код для загрузки контекста `WebApplicationContext` сервлета диспетчера вместо его конфигурирования в файле `web.xml`.

Для применения конфигурации на основе кода необходимо только разработать класс, реализующий интерфейс `org.springframework.web.WebApplicationInitializer`. Все классы, реализующие этот интерфейс, будут автоматически обнаруживаться классом `org.springframework.web.SpringServletContainerInitializer` (реализующим интерфейс `javax.servlet.ServletContainerInitializer` из Servlet 3.0), который автоматически загружается в любом контейнере Servlet 3.0.

Давайте рассмотрим простой пример использования конфигурации на основе кода для загрузки контекста `WebApplicationContext` сервлета диспетчера вместо ее объявления в файле `web.xml`.

Для начала удалим из файла `web.xml` объявления сервлета и отображений для него (листинг 16.65).

**Листинг 16.65. Удаление из файла `web.xml` объявлений сервлета и отображений для него**

```
<servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/appServlet/servlet-context.xml
        </param-value>
    </init-param>
```

```
      <load-on-startup>1</load-on-startup>
      <multipart-config>
         <max-file-size>5000000</max-file-size>
      </multipart-config>
   </servlet>

   <servlet-mapping>
      <servlet-name>appServlet</servlet-name>
      <url-pattern>/</url-pattern>
   </servlet-mapping>
```

После этого создадим класс, который реализует интерфейс `WebApplicationInitializer`. Класс называется `MyWebAppInitializer`, а его код приведен в листинге 16.66.

**Листинг 16.66. Класс `MyWebAppInitializer`**

```
package com.apress.prospring4.ch16;

import javax.servlet.MultipartConfigElement;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.support.XmlWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

public class MyWebAppInitializer implements WebApplicationInitializer {
   @Override
   public void onStartup(ServletContext container) throws ServletException {
     XmlWebApplicationContext appContext = new XmlWebApplicationContext();

     appContext.setConfigLocation(
        "/WEB-INF/spring/appServlet/servlet-context.xml");

     ServletRegistration.Dynamic dispatcher =
       container.addServlet("appServlet", new DispatcherServlet(appContext));

     MultipartConfigElement multipartConfigElement =
        new MultipartConfigElement(null, 5000000, 5000000, 0);
     dispatcher.setMultipartConfig(multipartConfigElement);

     dispatcher.setLoadOnStartup(1);
     dispatcher.addMapping("/");
   }
}
```

Как здесь показано, метод `WebApplicationInitializer.onStartup()` был переопределен за счет реализации логики для загрузки контекста `WebApplicationContext` сервлета диспетчера. Вызывая метод `ServletContext.addServlet()`, мы можем добавить сервлет к веб-контейнеру. Этот метод возвратит экземпляр реализации интерфейса `javax.servlet.ServletRegistration.Dynamic`, и с помощью этого интерфейса можно сконфигурировать различные атрибуты сервлета, такие как отображение сервлета на URL, поддержка множественного содержимого при загрузке файлов и т.д. После повторного построения и раз-

вертывания проекта контекст `WebApplicationContext` сервлета диспетчера будет загружаться так же, как и в случае его конфигурирования в файле `web.xml`.

Используя такой подход в комбинации с конфигурированием Spring в Java-коде, можно реализовать чистую конфигурацию на основе Java-кода для веб-приложения Spring, без необходимости в объявлении любой конфигурации Spring в `web.xml` или других XML-файлах конфигурации Spring.

# Резюме

В этой главе были раскрыты многие темы, связанные с разработкой веб-приложений с помощью Spring MVC.

Сначала мы обсудили высокоуровневые концепции шаблона MVC. Затем мы рассмотрели архитектуру Spring MVC, включая иерархию `WebApplicationContext`, жизненный цикл обработки запросов и конфигурирование.

После этого мы разработали пример приложения контактов, используя Spring MVC и JSPX в качестве технологии представлений. Во время разработки были исследованы разнообразные области. Главные темы касались интернационализации, оформления темами и поддержки шаблонов с применением Apache Tiles. Кроме того, для совершенствования пользовательского интерфейса мы использовали jQuery, jQuery UI и другие JavaScript-библиотеки. В число примеров входили компонент выбора даты, редактор форматированного текста и сетка данных с поддержкой разбиения на страницы. Также обсуждались вопросы защиты веб-приложений посредством Spring Security.

Мы также продемонстрировали функциональные средства веб-контейнеров, совместимых с Servlet 3.0, такие как конфигурирование на основе кода вместо файла `web.xml`. Мы показали, как обрабатывать загрузку файлов внутри среды Servlet 3.0.

В следующей главе мы раскроем дополнительные функциональные возможности, которые Spring предлагает для разработки веб-приложений, рассмотрев введение в WebSocket.

# ГЛАВА 17

# Протокол WebSocket

Для обеспечения коммуникаций между клиентом и сервером в веб-приложениях традиционно использовалась стандартная функциональность запроса/ответа HTTP. По мере развития Интернета возникала потребность в более интерактивных возможностях, часть которых предполагала активное или принудительное получение обновлений от сервера либо выполнение обновлений в реальном времени. Постепенно были реализованы разнообразные методы, такие как непрерывный опрос, длительный опрос и технология Comet. Каждый метод обладал своими достоинствами и недостатками, и протокол WebSocket является результатом попытки извлечь урок из существующих потребностей и нехватки средств за счет создания более простого и надежного способа построения интерактивных приложений.

В этой главе предлагается высокоуровневый обзор протокола WebSocket и основной функциональности, которая предоставляется платформой Spring Framework. В частности, в главе рассматриваются следующие темы.

- **Введение в WebSocket.** Мы представим общее введение в протокол WebSocket. Эта часть главы не предназначена служить подробным справочником по WebSocket, а является лишь высокоуровневым обзором. За детальной информацией о протоколе WebSocket обращайтесь к документу RFC-6455, который доступен по ссылке `http://tools.ietf.org/html/rfc6455`.
- **Использование WebSocket совместно с платформой Spring.** В этой части главы мы углубимся в некоторые детали работы с WebSocket в рамках платформы Spring Framework. Здесь мы рассмотрим применение WebSocket API из Spring, использование SockJS в качестве альтернативного варианта для браузеров, не поддерживающих WebSocket, и организацию отправки сообщений с помощью протокола STOMP (Simple (или Streaming) Text Oriented Message Protocol — простой (или потоковый) протокол текстовых сообщений) поверх SockJS/WebSocket.

## Введение в WebSocket

Протокол WebSocket представляет собой спецификацию, разработанную как часть инициативы HTML5 и делающую возможным полнодуплексное подключение с использованием единственного сокета, при котором сообщения могут пересылаться между клиентом и сервером. В прошлом веб-приложения, требующие функциональности обновлений в реальном времени, для получения таких данных должны были периодически опрашивать компонент серверной стороны, открывая множество подключений или применяя длительный опрос.

Использование протокола WebSocket при взаимодействии в двух направлениях позволяет избегать потребности в опросе HTTP для двунаправленных коммуникаций между клиентом (например, веб-браузером) и HTTP-сервером. Протокол WebSocket предназначен для замены всех существующих методов взаимодействия в двух направлениях, применяющих HTTP в качестве транспорта. Модель с единственным сокетом WebSocket в результате дает более простое решение, устраняя потребность в наличии множества подключений для каждого клиента и снижая накладные расходы (скажем, исчезает необходимость в отправке HTTP-заголовка с каждым сообщением).

Протокол WebSocket использует HTTP во время начального установления связи, что в свою очередь делает возможным его применение на стандартных портах HTTP (80) и HTTPS (443). Для обозначения незащищенных и защищенных подключений в спецификации WebSocket определены схемы `ws://` и `wss://`.

Протокол WebSocket состоит из двух частей: квитирование (установление связи) между клиентом и сервером, а также последующая передача данных. Подключение WebSocket устанавливается путем отправки запроса на обновление из HTTP протоколу WebSocket во время фазы начального квитирования между клиентом и сервером через то же самое лежащее в основе подключение TCP/IP. На протяжении фазы передачи данных как клиент, так и сервер могут посылать сообщения друг другу одновременно, что, как несложно догадаться, открывает возможность добавления к приложениям функциональности более надежного взаимодействия в реальном времени.

# Использование WebSocket совместно с платформой Spring

Начиная с версии 4.0, платформа Spring Framework поддерживает обмен сообщениями в стиле WebSocket, а также STOMP в качестве подпротокола уровня приложения. Внутри платформы поддержка WebSocket находится в модуле `spring-websocket`, который совместим со стандартом WebSocket API для Java (JSR-356).

Разработчики приложений должны также осознавать, что хотя протокол WebSocket привносит новые захватывающие возможности, он поддерживается далеко не всеми браузерами. С учетом того, что приложение должно продолжать работу с пользователями, в браузерах которых поддержка WebSocket отсутствует, понадобится задействовать какую-то запасную технологию для максимально близкой эмуляции желаемой функциональности. Для такого случая в Spring Framework предоставляются прозрачные альтернативные варианты через протокол SockJS, которые рассматриваются позже в этой главе.

В отличие от приложений, основанных на REST, где службы представлены различными URL, в протоколе WebSocket применяется единственный URL для установления начального квитирования, а данные перемещаются по одному и тому же подключению. Такой тип функциональности передачи сообщений в большей степени присущ традиционным системам обмена сообщениями. В версии Spring Framework 4 основные интерфейсы для работы с сообщениями, такие как Message, были перенесены из проекта Spring Integration в новый модуль под названием `spring-messaging`, предназначенный для поддержки приложений обмена сообщениями в стиле WebSocket.

Когда мы упоминаем об использовании STOMP в качестве подпротокола уровня приложения, то имеем в виду протокол, который транспортируется через WebSocket. Сам по себе WebSocket является низкоуровневым протоколом, который просто трансформирует байты в сообщения. Приложение должно быть осведомлено о том, что именно передается по сети, и как раз здесь в игру вступает подпротокол, подобный STOMP. Во время начального квитирования для определения применяемого подпротокола клиент и сервер могут использовать заголовок `Sec-WebSocket-Protocol`. Хотя Spring Framework предлагает поддержку STOMP, протокол WebSocket не регламентирует что-либо специфическое.

Теперь, когда вы понимаете, что собой представляет протокол WebSocket и какую поддержку предоставляет Spring, давайте посмотрим, где можно пользоваться этой технологией. Учитывая односокетную природу протокола WebSocket и его способность обеспечивать непрерывный двунаправленный поток данных, WebSocket идеально подходит для приложений, которые характеризуются высокой частотой передачи сообщений и требуют коммуникаций с низкой задержкой. Приложения, которые могут стать хорошими кандидатами для применения WebSocket, включают игры, инструменты группового сотрудничества в реальном времени, системы обмена сообщениями, чувствительные ко времени службы выдачи ценовой информации, такой как финансовые обновления, и т.д. Когда вы проектируете приложение с учетом использования WebSocket, то должны принимать во внимание как частоту появления сообщений, так и требования к задержке. Это поможет в определении, применять протокол WebSocket или, скажем, длительный опрос HTTP.

## Использование WebSocket API

Как упоминалось ранее в главе, WebSocket просто трансформирует байты в сообщения и транспортирует их между клиентом и сервером. Эти сообщения по-прежнему должны распознаваться самим приложением, и именно здесь начинают действовать подпротоколы вроде STOMP. На тот случай, если вы предпочитаете работать с низкоуровневым интерфейсом WebSocket API напрямую, платформа Spring Framework предоставляет API-интерфейс, с которым можно взаимодействовать. Имея дело с WebSocket API из Spring, вы обычно будете реализовывать интерфейс `WebSocketHandler` или применять удобные подклассы, такие как `BinaryWebSocketHandler` для двоичных сообщений, `SockJsWebSocketHandler` — для сообщений SockJS либо `TextWebSocketHandler` — для сообщений, основанных на `String`. В рассматриваемом примере в целях упрощения мы будем использовать `TextWebSocketHandler` для передачи сообщений `String` через WebSocket. Давайте начнем с того, что посмотрим, каким образом можно получать и работать с сообщениями WebSocket на низком уровне, задействовав WebSocket API из Spring.

При желании каждый пример в этой главе также может быть сконфигурирован посредством Java Config. По нашему мнению, пространство имен XML представляет аспекты конфигурации в сжатой форме, поэтому оно будет применяться повсеместно в этой главе. За дополнительной информацией по Java Config обращайтесь в справочное руководство.

Давайте начнем с добавления требуемых зависимостей, как показано в табл. 17.1.

**Таблица 17.1. Зависимости Maven для примера использования WebSocket API**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| org.springframework | spring-context | 4.0.2 | Модуль Spring Context |
| org.springframework | spring-websocket | 4.0.2 | Модуль Spring WebSocket |
| org.springframework | spring-webmvc | 4.0.2 | Модуль Spring Web MVC |
| org.apache.tomcat | tomcat-websocket-api | 7.0.54 | Интерфейс Apache Tomcat WebSocket API. Поскольку в примере мы используем сервер Tomcat, это является требованием контейнера |
| org.apache.tomcat.embed | tomcat-embed-websocket | 7.0.54 | Интерфейс Apache Tomcat WebSocket API для встроенного сервера Tomcat, который применяется для запуска примера |

В листинге 17.1 приведено содержимое конфигурационного файла `web.xml` (`src/main/webapp/WEB-INF/web.xml`), которое позволяет использовать WebSocket со стандартным сервлетом диспетчера (`DispatcherServlet`) из Spring MVC.

**Листинг 17.1. Конфигурация `web.xml` для сервлета диспетчера**

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
       version="3.0">

   <display-name>Spring WebSocket API Sample</display-name>

   <servlet>
      <servlet-name>websocket</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>/WEB-INF/spring/root-context.xml</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>websocket</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Показанная конфигурация должна выглядеть знакомой. Первым делом мы создаем наше определение сервлета с применением Spring-класса `DispatcherServlet`, предоставляя ему конфигурационный файл (`/WEB-INF/spring/root-context.xml`).

Затем мы определяем отображение сервлета (`servlet-mapping`), указывая, что все запросы должны направляться этому сервлету диспетчера.

А теперь создадим файл `root-context`, содержащий конфигурацию WebSocket (листинг 17.2).

**Листинг 17.2. Файл `root-context`**

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:websocket="http://www.springframework.org/schema/websocket"
      xmlns:mvc="http://www.springframework.org/schema/mvc"
      xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/websocket
       http://www.springframework.org/schema/websocket/spring-websocket.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc.xsd">

   <mvc:resources mapping="/index.html" location="/static/" />

   <websocket:handlers>
      <websocket:mapping path="/echoHandler" handler="echoHandler"/>
   </websocket:handlers>

   <bean id="echoHandler" class="com.apress.prospring4.ch17.EchoHandler"/>
</beans>
```

Сначала мы конфигурируем статический ресурс по имени `index.html`. Этот файл содержит статическую разметку HTML и код JavaScript, который используется для взаимодействия со службой WebSocket серверной стороны. Затем с применением пространства имен `websocket` мы конфигурируем наши обработчики и соответствующий бин для обработки запроса. В данном примере мы определяем единственное отображение обработчика, который получает запросы с путем `/echoHandler` и использует бин с идентификатором `echoHandler` для приема сообщения и ответа посредством отправки предоставленного сообщения обратно клиенту.

Теперь мы готовы к реализации подкласса класса `TextWebSocketHandler` (`src/main/java/com/apress/prospring4/ch17/EchoHandler.java`), который помогает иметь дело с сообщениями, основанными на `String`, удобным способом (листинг 17.3).

**Листинг 17.3. Реализация подкласса класса `TextWebSocketHandler` для обработки сообщений WebSocket, основанных на `String`**

```
package com.apress.prospring4.ch17;

import org.springframework.web.socket.TextMessage;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.handler.TextWebSocketHandler;

import java.io.IOException;

public class EchoHandler extends TextWebSocketHandler {
   @Override
```

```
    public void handleTextMessage(WebSocketSession session,
                                  TextMessage textMessage)
throws IOException {
        session.sendMessage(new TextMessage(textMessage.getPayload()));
    }
}
```

Как видите, это базовый обработчик, который принимает сообщение и отправляет его обратно клиенту без изменений. Содержимое полученного сообщения WebSocket находится в методе `getPayload()`.

Мы сделали практически все, что необходимо для серверной стороны. Учитывая, что `EchoHandler` является типичным бином Spring, вы можете предпринимать любые действия, допустимые в нормальном Spring-приложении, такие как внедрение служб, чтобы позаботится о любых функциях, в которых может нуждаться этот обработчик.

Давайте создадим простого клиента, который сможет взаимодействовать со службой WebSocket серверной стороны. Клиент будет представлять собой обычную HTML-страницу с небольшой добавкой кода JavaScript, который с помощью API-интерфейса браузера создает подключение WebSocket, и кода jQuery, предназначенного для обработки событий щелчков на кнопках и отображения данных. Клиентское приложение будет иметь возможность подключаться, отключаться, отправлять сообщение и отображать обновления состояния на экране. Код клиентской страницы показан в листинге 17.4 (`src/main/webapp/static/index.html`).

**Листинг 17.4. Код HTML и JavaScript для клиентской страницы**

```
<html>
<head>
    <meta charset="UTF-8">
    <title>WebSocket Tester</title>
    <script language="javascript" type="text/javascript"
            src="http://code.jquery.com/jquery-2.1.1.min.js"></script>
    <script language="javascript" type="text/javascript">
        var ping;
        var websocket;

        jQuery(function ($) {
            function writePing(message) {
                $('#pingOutput').append(message + '\n');
            }

            function writeStatus(message) {
              $("#statusOutput").val($("#statusOutput").val() + message + '\n');
            }

            function writeMessage(message) {
                $('#messageOutput').append(message + '\n')
            }

            $('#connect')
                    .click(function doConnect() {
                        websocket = new WebSocket($("#target").val());

                        websocket.onopen = function (evt) {
                            writeStatus("CONNECTED");
```

```
                var ping = setInterval(function () {
                    if (websocket != "undefined") {
                        websocket.send("ping");
                    }
                }, 3000);
            };

            websocket.onclose = function (evt) {
                writeStatus("DISCONNECTED");
            };

            websocket.onmessage = function (evt) {
                if (evt.data === "ping") {
                    writePing(evt.data);
                } else {
                    writeMessage('ECHO: ' + evt.data);
                }
            };

            websocket.onerror = function (evt) {
                onError(writeStatus('ERROR:' + evt.data))
            };
        });

    $('#disconnect')
        .click(function () {
            if(typeof websocket != 'undefined') {
                websocket.close();
                websocket = undefined;
            } else {
                alert("Not connected.");
            }
        });

    $('#send')
        .click(function () {
            if(typeof websocket != 'undefined') {
                websocket.send($('#message').val());
            } else {
                alert("Not connected.");
            }
        });
});
</script>
</head>

<body>
<h2>WebSocket Tester</h2>
Target:
<input type="text" id="target" size="40" value="ws://localhost:8080/
websocket-api/echoHandler"/>
<br/>
<button id="connect">Connect</button>
<button id="disconnect">Disconnect</button>
<br/>
<br/>Message:
<input type="text" id="message" value=""/>
```

```
<button id="send">Send</button>
<br/>
<p>Status output:</p>
<pre><textarea id="statusOutput" rows="10" cols="50"></textarea></pre>
<p>Message output:</p>
<pre><textarea id="messageOutput" rows="10" cols="50"></textarea></pre>
<p>Ping output:</p>
<pre><textarea id="pingOutput" rows="10" cols="50"></textarea></pre>
</body>
</html>
```

В листинге 17.4 представлен пользовательский интерфейс, который позволяет производить обратные вызовы WebSocket API и в реальном времени наблюдать на экране результаты.

Постройте проект и разверните его внутри веб-контейнера. Затем перейдите по ссылке `http://localhost:8080/websocket-api/index.html`, чтобы отобразить пользовательский интерфейс. После щелчка на кнопке Connect (Подключиться) в текстовой области Status Output (Вывод состояния) вы заметите сообщение `CONNECTED` (Подключено), а в текстовой области Ping Output (Вывод `ping`) каждые 3 секунды будет отображаться сообщение `ping`. Введите сообщение в текстовом поле Message (Сообщение) и щелкните на кнопке Send (Отправить). Это сообщение будет отправлено службе WebSocket серверной стороны и отображено в текстовой области Message Output (Вывод сообщения). Завершив отправлять сообщения, щелкните на кнопке Disconnect (Отключиться), после чего в текстовой области Status Output вы увидите сообщение `DISCONNECTED` (Отключено). Вы не сможете отправлять любые дополнительные сообщения, равно как и отключаться повторно, пока снова не подключитесь к службе WebSocket.

Несмотря на то что в этом примере задействована абстракция Spring, определенная поверх низкоуровневого интерфейса WebSocket API, вы четко видите великолепные возможности, которые данная технология способна привнести в ваши приложения. А теперь давайте посмотрим, как предложить такую же функциональность, когда браузер не поддерживает WebSocket и требуется запасной вариант. Проверить свой браузер на предмет совместимости с WebSocket можно с помощью сайта, такого как `www.websocket.org/echo.html`.

## Использование SockJS

Из-за того, что не все браузеры поддерживают WebSocket, но приложение должно корректно функционировать для любого конечного пользователя, в Spring Framework предлагается запасной вариант, предусматривающий использование SockJS. Применение SockJS обеспечит максимально схожее с WebSocket поведение во время выполнения, но без необходимости в изменении кода на стороне приложения.

Протокол SockJS используется на клиентской стороне посредством библиотек JavaScript. Модуль `spring-websocket` платформы Spring Framework содержит соответствующие компоненты SockJS серверной стороны. Когда для предоставления гибкой альтернативы применяется SockJS, клиент сначала отправит серверу запрос `GET` с использованием пути `/info`, чтобы получить информацию о транспорте от

сервера. Протокол SockJS первым делом попробует воспользоваться WebSocket, затем потоковым режимом HTTP и, наконец, длительным опросом HTTP как крайним случаем. Дополнительные сведения о протоколе SockJS и связанных с ним проектах доступны по ссылке `https://github.com/sockjs`.

Включение поддержки SockJS через пространство имен `websocket` осуществляется довольно просто и требует только дополнительной директивы внутри блока `<websocket:handlers>`. Давайте построим приложение, которое подобно приложению, работающему с низкоуровневым интерфейсом WebSocket API, но использует SockJS. Прежде всего, необходимо добавить зависимость, описанную в табл. 17.2.

**Таблица 17.2. Зависимость Maven для примера применения SockJS**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `com.fasterxml.jackson.core` | `jackson-databind` | 2.4.1.1 | Библиотека привязки данных Jackson |

Затем следует создать файл `src/main/webapp/WEB-INF/spring/root-context.xml`, как показано в листинге 17.5.

**Листинг 17.5. Изменение файла `root-context.xml` для включения поддержки SockJS**

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:websocket="http://www.springframework.org/schema/websocket"
      xmlns:mvc="http://www.springframework.org/schema/mvc"
      xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/websocket
       http://www.springframework.org/schema/websocket/spring-websocket.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc.xsd">

   <mvc:resources mapping="/index.html" location="/static/" />

   <websocket:handlers>
      <websocket:mapping path="/echoHandler" handler="echoHandler"/>
      <websocket:sockjs/>
   </websocket:handlers>

   <bean id="echoHandler" class="com.apress.prospring4.ch17.EchoHandler"/>
</beans>
```

Обратите внимание на добавление дескриптора `<websocket:sockjs>`. На самом базовом уровне это все, что требуется для включения SockJS. Мы можем повторно воспользоваться классом `EchoHandler` из примера применения WebSocket API, т.к. планируем предоставить ту же самую функциональность.

Дескриптор пространства имен `<websocket:sockjs/>` поддерживает также и другие атрибуты, которые предназначены для установки таких параметров конфигурации, как обработка cookie-наборов сеанса (по умолчанию включена), местоположения для загрузки специальных клиентских библиотек (на время написания книги

стандартным местоположением было `https://d1fxtkz8shb9d2.cloudfront.net/sockjs-0.3.4.min.js`), конфигурация сигналов работоспособности, предельные размеры сообщений и т.д. Эти параметры потребуется просмотреть и настроить должным образом в зависимости от нужд приложения и типов транспорта.

Теперь давайте создадим файл `web.xml` для отражения нашего сервлета SockJS (листинг 17.6).

**Листинг 17.6. Файл `web.xml` для SockJS**

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
       version="3.0">

   <display-name>Spring SockJS API Sample</display-name>

   <servlet>
      <servlet-name>sockjs</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>/WEB-INF/spring/root-context.xml</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
      <async-supported>true</async-supported>
   </servlet>

   <servlet-mapping>
      <servlet-name>sockjs</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Далее нам необходимо создать HTML-страницу, как мы поступали в примере с интерфейсом WebSocket API, но на этот раз задействовать протокол SockJS, чтобы позаботиться о согласовании транспорта. Наиболее заметные отличия связаны с использованием библиотеки SockJS вместо работы с WebSocket напрямую и применением для подключения к конечной точке типичной схемы `http://` взамен `ws://`. В листинге 17.7 приведен простой клиентский код HTML.

**Листинг 17.7. Клиентская HTML-страница, используемая для отправки сообщений через SockJS**

```
<html>
<head>
   <meta charset="UTF-8">
   <title>SockJS Tester</title>
   <script language="javascript" type="text/javascript"
          src="https://d1fxtkz8shb9d2.cloudfront.net/sockjs-0.3.4.min.js">
   </script>
   <script language="javascript" type="text/javascript"
          src="http://code.jquery.com/jquery-2.1.1.min.js"></script>
```

```
<script language="javascript" type="text/javascript">
   var ping;
   var sockjs;

   jQuery(function ($) {
      function writePing(message) {
         $('#pingOutput').append(message + '\n');
      }

      function writeStatus(message) {
        $("#statusOutput").val($("#statusOutput").val() + message + '\n');
      }

      function writeMessage(message) {
         $('#messageOutput').append(message + '\n')
      }

      $('#connect')
            .click(function doConnect() {
               sockjs = new SockJS($("#target").val());

               sockjs.onopen = function (evt) {
                  writeStatus("CONNECTED");

                  var ping = setInterval(function () {
                     if (sockjs != "undefined") {
                        sockjs.send("ping");
                     }
                  }, 3000);
               };

               sockjs.onclose = function (evt) {
                  writeStatus("DISCONNECTED");
               };

               sockjs.onmessage = function (evt) {
                  if (evt.data === "ping") {
                     writePing(evt.data);
                  } else {
                     writeMessage('ECHO: ' + evt.data);
                  }
               };

               sockjs.onerror = function (evt) {
                  onError(writeStatus('ERROR:' + evt.data))
               };
            });

      $('#disconnect')
            .click(function () {
               if(typeof sockjs != 'undefined') {
                  sockjs.close();
                  sockjs = undefined;
               } else {
                  alert("Not connected.");
               }
            });

      $('#send')
            .click(function () {
```

```
                if(typeof sockjs != 'undefined') {
                    sockjs.send($('#message').val());
                } else {
                    alert("Not connected.");
                }
            });
    });
  </script>
</head>

<body>
<h2>SockJS Tester</h2>
Target:
<input type="text" id="target" size="40"
       value="http://localhost:8080/sockjs/echoHandler"/>
<br/>
<button id="connect">Connect</button>
<button id="disconnect">Disconnect</button>
<br/>
<br/>Message:
<input type="text" id="message" value=""/>
<button id="send">Send</button>
<br/>
<p>Status output:</p>
<pre><textarea id="statusOutput" rows="10" cols="50"></textarea></pre>
<p>Message output:</p>
<pre><textarea id="messageOutput" rows="10" cols="50"></textarea></pre>
<p>Ping output:</p>
<pre><textarea id="pingOutput" rows="10" cols="50"></textarea></pre>
</body>
</html>
```

Реализовав новый код SockJS, проект можно построить и развернуть в контейнере, после чего перейти к пользовательскому интерфейсу, расположенному в `http://localhost:8080/sockjs/index.html`, который обладает всеми теми же характеристиками и функциональностью, что и пользовательский интерфейс в примере работы с WebSocket. Чтобы протестировать альтернативную функциональность SockJS, отключите поддержку WebSocket в своем браузере. Например, в Firefox перейдите на страницу `about:config` и найдите настройку `network.websocket.enabled`. Переключите ее на `false`, перезагрузите пользовательский интерфейс примера и подключитесь повторно. С помощью инструмента, подобного Live HTTP Headers, можно инспектировать трафик, который проходит от браузера к серверу. После верификации поведения переключите настройку `network.websocket.enabled` в Firefox обратно в `true`, перезагрузите страницу и подключитесь повторно. Понаблюдав за трафиком посредством Live HTTP Headers, вы заметите наличие квитирования WebSocket. В нашем простом примере все должно работать в точности так, как если бы использовался интерфейс WebSocket API.

## Отправка сообщений с помощью STOMP

При работе с WebSocket в качестве общего формата между клиентом и сервером обычно будет применяться подпротокол вроде STOMP, поэтому обеим сторонам известно, чего ожидать и каким образом реагировать. Подпротокол STOMP поддерживается платформой Spring Framework, и мы будем использовать его в рассматриваемом примере.

Простой, основанный на кадрах, протокол обмена сообщениями STOMP, который моделируется на базе HTTP, может применяться поверх любого надежного двунаправленного потокового протокола сети, такого как WebSocket. При этом STOMP имеет стандартный формат протокола; существует поддержка JavaScript клиентской стороны для отправки и получения сообщений в браузере и дополнительно для подключения к традиционным брокерам обмена сообщениями, которые поддерживают STOMP, например, RabbitMQ и ActiveMQ. Изначально платформа Spring Framework поддерживает простой брокер, который обрабатывает запросы подписки и широковещательную рассылку сообщений подключенным клиентам в памяти. В этом примере мы задействуем простой брокер, оставив настройку полнофункционального брокера в качестве упражнения для самостоятельной проработки. За дополнительными деталями обращайтесь по ссылке `http://docs.spring.io/spring/docs/current/spring-framework-reference/html/websocket.html#websocket-stomp-handle-broker-relay-configure`.

**На заметку!** Полное описание протокола STOMP доступно по ссылке `http://stomp.github.io/stomp-specification-1.2.html`.

В примере использования STOMP мы создадим простое приложение биржевых котировок, которое отображает несколько заранее определенных биржевых символов, их текущие цены и метки времени момента изменения цен. Пользовательский интерфейс позволит добавлять новые биржевые символы и их начальные цены. Любые подключающиеся клиенты (т.е. другие браузеры внутри сети или совершенно новые клиенты из других сетей) будут видеть одни и те же данные при условии, что они подписаны на широковещательные рассылки сообщений. Ежесекундно каждая цена акции будет обновляться новым случайным значением вместе с обновлением соответствующей метки времени.

Чтобы обеспечить возможность взаимодействия клиентов с приложением биржевых котировок, даже если их браузеры не поддерживают WebSocket, мы снова применим SockJS для прозрачной обработки переключения на любой транспорт. Давайте обратимся к коду. Первым делом необходимо добавить зависимость, описанную в табл. 17.3.

**Таблица 17.3. Зависимость Maven для примера применения STOMP**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.springframework` | `spring-messaging` | 4.0.2 | Модуль `spring-messaging` для поддержки отправки сообщений в примере использования STOMP |

Теперь создадим объект предметной области `Stock`, который будет хранить информацию о котировке (код и цену), как показано в листинге 17.8.

**Листинг 17.8. Объект предметной области `Stock`**

```
package com.apress.prospring4.ch17;

import java.util.Date;
import java.io.Serializable;
import java.text.DateFormat;
import java.text.SimpleDateFormat;

public class Stock implements Serializable {
    private static final long serialVersionUID = 1L;
    private static final String DATE_FORMAT = "MMM dd yyyy HH:mm:ss";

    private String code;
    private double price;
    private Date date = new Date();
    private DateFormat dateFormat = new SimpleDateFormat(DATE_FORMAT);

    public Stock() { }

    public Stock(String code, double price) {
        this.code = code;
        this.price = price;
    }

    public String getCode() {
        return code;
    }

    public void setCode(String code) {
        this.code = code;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public String getDateFormatted() {
        return dateFormat.format(date);
    }
}
```

Далее понадобится добавить контроллер MVC, предназначенный для обработки входящих запросов (листинг 17.9).

**Листинг 17.9. Контроллер MVC для обработки входящих запросов котировок**

```
package com.apress.prospring4.ch17;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.messaging.simp.SimpMessagingTemplate;
import org.springframework.scheduling.TaskScheduler;
import org.springframework.stereotype.Controller;

import javax.annotation.PostConstruct;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Random;

@Controller
public class StockController {
    private TaskScheduler taskScheduler;
    private SimpMessagingTemplate simpMessagingTemplate;

    private List<Stock> stocks = new ArrayList<Stock>();
    private Random random = new Random(System.currentTimeMillis());

    public StockController() {
        stocks.add(new Stock("VMW", 1.00d));
        stocks.add(new Stock("EMC", 1.00d));
        stocks.add(new Stock("GOOG", 1.00d));
        stocks.add(new Stock("IBM", 1.00d));
    }

    @MessageMapping("/addStock")
    public void addStock(Stock stock) throws Exception {
        stocks.add(stock);
        broadcastUpdatedPrices();
    }

    @Autowired
    public void setSimpMessagingTemplate(
      SimpMessagingTemplate simpMessagingTemplate) {
        this.simpMessagingTemplate = simpMessagingTemplate;
    }

    @Autowired
    public void setTaskScheduler(TaskScheduler taskScheduler) {
        this.taskScheduler = taskScheduler;
    }

    private void broadcastUpdatedPrices() {
        for(Stock stock : stocks) {
            stock.setPrice(stock.getPrice()
                + (getUpdatedStockPrice() * stock.getPrice()));
            stock.setDate(new Date());
        }

        simpMessagingTemplate.convertAndSend("/topic/price", stocks);
    }

    private double getUpdatedStockPrice() {
        double priceChange = random.nextDouble() * 5.0;
```

```
        if (random.nextInt(2) == 1) {
            priceChange = -priceChange;
        }

        return priceChange / 100.0;
    }

    @PostConstruct
    private void broadcastTimePeriodically() {
        taskScheduler.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                broadcastUpdatedPrices();
            }
        }, 1000);
    }
}
```

В контроллере делаются две вещи. Прежде всего, в целях демонстрации мы добавляем к списку несколько заранее определенных биржевых символов, а также их начальные цены. Затем мы определяем метод `addStock()`, который принимает объект `Stock`, добавляем его в список биржевых символов и выполняем широковещательную рассылку котировок всем подписчикам. При широковещательной рассылке котировок мы проходим по всем добавленным биржевым символам, обновляем цену каждого и отправляем их всем подписчикам URL вида `/topic/price`, используя связанный шаблон `SimpMessagingTemplate`. Мы также применяем `TaskExecutor` для непрерывной широковещательной рассылки подписанным клиентам обновленного списка цен на акции каждую секунду.

Располагая контроллером, давайте теперь создадим пользовательский интерфейс HTML, предназначенный для отображения клиентам (`src/main/webapp/static/index.html`), как показано в листинге 17.10.

**Листинг 17.10. Пользовательский интерфейс HTML для отображения цен на акции**

```
<html>
<head>
  <title>Stock Ticker</title>
  <script src="https://d1fxtkz8shb9d2.cloudfront.net/sockjs-0.3.4.min.js">
  </script>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3.2/stomp.min.js">
  </script>
  <script src="http://code.jquery.com/jquery-2.1.1.min.js"></script>
  <script>
      var stomp = Stomp.over(new SockJS("/stomp/ws"));

      function displayStockPrice(frame) {
          var prices = JSON.parse(frame.body);

          $('#price').empty();

          for (var i in prices) {
              var price = prices[i];

              $('#price').append(
                      $('<tr>').append(
```

```
                        $('<td>').html(price.code),
                        $('<td>').html(price.price.toFixed(2)),
                        $('<td>').html(price.dateFormatted)
                    )
            );
        }
    }

    var connectCallback = function () {
        stomp.subscribe('/topic/price', displayStockPrice);
    };

    var errorCallback = function (error) {
        alert(error.headers.message);
    };

    stomp.connect("guest", "guest", connectCallback, errorCallback);

    $(document).ready(function () {
        $('.addStockButton').click(function (e) {
            e.preventDefault();

            var jsonstr = JSON.stringify({ 'code': $('.addStock .code').val(),
                'price': Number($('.addStock .price').val()) });

            stomp.send("/app/addStock", {}, jsonstr);

            return false;
        });
    });
  </script>
</head>
<body>
<h1><b>Stock Ticker</b></h1>
<table border="1">
  <thead>
  <tr>
    <th>Code</th>
    <th>Price</th>
    <th>Time</th>
  </tr>
  </thead>
  <tbody id="price"></tbody>
</table>
<p class="addStock">
  Code: <input type="text" class="code"/><br/>
  Price: <input type="text" class="price"/><br/>
  <button class="addStockButton">Add Stock</button>
</p>
</body>
</html>
```

Подобно предшествующим примерам, здесь присутствует HTML-разметка, смешанная с кодом JavaScript для обновления экрана. Мы используем jQuery для обновления HTML-данных, SockJS для предоставления выбора транспорта и JavaScript-библиотеку STOMP под названием `stomp.js` для взаимодействия с сервером.

Данные, отправленные в сообщениях STOMP, кодируются в формате JSON и при необходимости извлекаются. После подключения STOMP мы производим подписку на `/topic/price` для получения обновления цен на акции.

Далее мы сконфигурируем встроенный брокер STOMP в файле `root-context.xml` (`src/main/webapp/WEB-INF/spring/root-context.xml`), содержимое которого приведено в листинге 17.11.

**Листинг 17.11. Конфигурация брокера STOMP**

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:websocket="http://www.springframework.org/schema/websocket"
      xmlns:mvc="http://www.springframework.org/schema/mvc"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/websocket
       http://www.springframework.org/schema/websocket/spring-websocket.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">

   <mvc:annotation-driven />

   <mvc:resources mapping="/index.html" location="/static/" />

   <context:component-scan base-package="com.apress.prospring4.ch17" />

   <websocket:message-broker application-destination-prefix="/app">
      <websocket:stomp-endpoint path="/ws">
         <websocket:sockjs/>
      </websocket:stomp-endpoint>
      <websocket:simple-broker prefix="/topic"/>
   </websocket:message-broker>

   <bean id="taskExecutor"
         class="org.springframework.core.task.SimpleAsyncTaskExecutor"/>
</beans>
```

По большей части эта конфигурация должна выглядеть знакомой. В настоящем примере мы конфигурируем `message-broker` с применением пространства имен `websocket`, определяем конечную точку STOMP и включаем SockJS. Мы также конфигурируем префикс, который подписчики будут использовать для получения сообщений. Сконфигурированный бин `taskExecutor` предназначен для предоставления биржевых котировок через определенный интервал в классе контроллера. Когда применяется поддержка пространства имен, шаблон `SimpMessagingTemplate` создается автоматически и доступен для внедрения в бины.

Осталось лишь сконфигурировать файл `web.xml` (`src/main/webapp/WEB-INF/web.xml`), как показано в листинге 17.12.

**Листинг 17.12. Конфигурация `web.xml` для примера использования STOMP**

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
       version="3.0">

   <display-name>Spring STOMP Sample</display-name>

   <servlet>
      <servlet-name>stomp</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>/WEB-INF/spring/root-context.xml</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
      <async-supported>true</async-supported>
   </servlet>

   <servlet-mapping>
      <servlet-name>stomp</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Теперь постройте проект и разверните его в веб-контейнере, после чего направьте браузер на URL вида `http://localhost:8080/stomp/index.html`. Когда страница загрузится, вы увидите список предварительно определенных биржевых символов, а также их колеблющиеся цены вместе с метками времени для измененных цен. Кроме того, вы заметите на странице область для добавления кода и цены новой котировки. После добавления новой котировки страница немедленно начнет обновление. Изменение цен на акции в реальном времени можно будет наблюдать не только на текущей странице, но также и в любом другом открытом браузере. Проверка достоверности внутри формы не выполняется; она оставлена вам в качестве упражнения.

Вот и все, что требовалось для создания основы приложения WebSocket, управляемого SockJS/STOMP.

# Резюме

В этой главе мы раскрыли общие концепции протокола WebSocket. Мы обсудили поддержку платформой Spring Framework низкоуровневого интерфейса WebSocket API и затем взглянули на применение SockJS как альтернативного варианта для выбора подходящего транспорта в зависимости от клиентского браузера. Наконец, мы представили STOMP в качестве подпротокола WebSocket, предназначенного для передачи сообщений между клиентом и сервером.

В следующей главе мы рассмотрим подпроекты Spring, которые можно использовать в своих приложениях для получения еще более надежной функциональности.

ГЛАВА 18

# Проекты Spring: Spring Batch, Spring Integration, Spring XD и Spring Boot

В этой главе представлен высокоуровневый обзор нескольких проектов, входящих в состав набора Spring, из которых самыми примечательными являются Spring Batch, Spring Integration, Spring XD и Spring Boot. Данная глава не предназначена служить подробным руководством по всем проектам, а дает лишь минимально необходимые сведения и примеры, позволяющие эффективно приступить к работе. Набор Spring содержит намного больше проектов, чем перечисленные в этой главе, но рассмотренные здесь проекты используются наиболее широко, тогда как другие появились недавно либо их начало планируется в будущем. Полный список проектов Spring доступен по ссылке `http://spring.io/projects`. В главе рассматриваются следующие темы.

- **Проект Spring Batch.** Будут раскрыты основные концепции инфраструктуры пакетной обработки Spring, включая то, что она предлагает разработчикам, а также кратко представлена новая поддержка спецификации JSR-352 в версии Spring Batch 3.0.
- **Проект Spring Integration.** Шаблоны интеграции применяются во многих корпоративных приложениях, и проект Spring Integration предоставляет надежную инфраструктуру для реализации таких шаблонов. На основе примера пакета мы продемонстрируем использование Spring Integration как части рабочего потока для запуска пакетного задания.
- **Проект Spring XD.** Проект Spring XD связывает вместе многие существующие проекты Spring, чтобы предоставить унифицированную и расширяемую систему для приложений, работающих с большими данными (big data). Он является распределенной системой, предназначенной для получения данных, анализа в реальном времени, пакетной обработки и экспортирования данных. Мы покажем, как воспроизвести то, что достигнуто в примерах применения

Spring Batch и Spring Integration, с использованием Spring XD посредством простого кода на языке DSL в интерфейсе оболочки.

- **Проект Spring Boot.** Проект Spring Boot направлен на упрощение процесса разработки приложений, позволяя разработчику создавать приложения с минимальным конфигурированием и настройкой безо всякой генерации кода. Мы представим проект Spring Boot с помощью типового веб-приложения "Hello World!" и посмотрим, что именно он предлагает приложению, не требуя никаких усилий с вашей стороны.

Поскольку любая из перечисленных тем заслуживает написания отдельной главы или даже книги, раскрытие абсолютно всех деталей каждого проекта и предоставляемых им функциональных средств в единственной главе попросту невозможно. Мы надеемся, что предложенные здесь введения и базовые примеры подтолкнут вас к дополнительному исследованию всех упомянутых инфраструктур.

# Проект Spring Batch

Проект *Spring Batch* — инфраструктура для пакетной обработки — входит в набор проектов Spring. Эта инфраструктура является легковесной, гибкой и ориентированной на предоставление разработчикам возможности создавать надежные пакетные приложения с минимальными усилиями. Spring Batch поступает с несколькими готовыми компонентами для разнообразных технологий, и в большинстве случаев нужное пакетное приложение удается построить с применением исключительно предлагаемых компонентов.

Типовые пакетные приложения включают ежедневную генерацию накладных, системы расчета заработной платы и процессы ETL (Extract, Transform, Load — извлечение, трансформация, загрузка). Несмотря на то что это элементарные примеры, Spring Batch можно использовать для любого процесса, который нуждается в автоматическом запуске, а не только в рассмотренных здесь сценариях. Как и все другие проекты, Spring Batch построен на основе ядра платформы Spring, и вы имеете полный доступ ко всем его возможностям.

С высокоуровневой точки зрения пакетное задание содержит один или большее количество шагов. Каждый шаг (реализация интерфейса `Step`) предоставляет возможность либо выполнения единицы работы, которая представлена реализацией `Tasklet`, либо участия в том, что называется *обработкой, ориентированной на порции* (chunk-oriented processing). При такой обработке шаг применяет реализацию интерфейса `ItemReader` для чтения данных в некоторой форме, дополнительную реализацию интерфейса `ItemProcessor` для выполнения над данными любых требуемых трансформаций и, наконец, реализацию интерфейса `ItemWriter` для записи результирующих данных. Реализация интерфейса `Step` также поддерживает различные атрибуты конфигурации, такие как размер порции (объем обрабатываемых данных на транзакцию), разрешение многопоточного выполнения, пределы пропуска данных и т.д. На уровнях шагов и заданий могут использоваться прослушиватели для получения уведомлений о событиях, которые происходят на протяжении жизненного цикла пакетного задания (например, перед началом шага, после завершения шага, ловушки для разнообразных отказов при чтении/обработке/записи в рамках сценария обработки, ориентированной на порции, и т.п.).

Хотя большинство заданий можно вполне успешно выполнять посредством единственного потока и одного процесса, в Spring Batch также предлагаются варианты для масштабирования и параллельной обработки заданий. В настоящее время Spring Batch предоставляет в готовом виде следующие возможности масштабирования.

- **Многопоточные шаги.** Это простейший способ сделать шаг многопоточным. Просто добавьте к конфигурации шага реализацию `TaskExecutor` по своему выбору, после чего каждая порция элементов в настройке обработки, ориентированной на порции, будет обрабатываться в собственном потоке выполнения.
- **Параллельные шаги.** Предположим для примера, что в начале задания необходимо прочитать два крупных файла с отличающимися данными. На первых порах вы можете создать два шага и выполнять их один за другим. Если загружаемые данные в обоих файлах не зависят друг от друга, то почему бы ни обработать их одновременно? В таком случае Spring Batch позволяет определить реализацию интерфейса `Split`, содержащего элементы потока, которые инкапсулируют задачи, подлежащие параллельному выполнению.
- **Удаленное разделение на порции.** Эта возможность масштабируемости позволяет удаленно распределить работу по нескольким дистанционным исполнителям и взаимодействовать с помощью какого-нибудь надежного промежуточного программного обеспечения, подобного AMQP или JMS. Удаленное разделение на порции обычно применяется, когда чтение данных не является узким местом внутри процесса, но запись и дополнительная обработка порции данных — наоборот, являются. Порции данных посредством промежуточного программного обеспечения отправляются на обработку подчиненным узлам, которые затем сообщают главному узлу состояние обработки конкретной порции.
- **Разбиение на части.** Эта возможность масштабируемости, как правило, используется при необходимости обработки диапазонов данных, для каждого из которых предусмотрен отдельный поток. Типичным сценарием может служить таблица базы данных, наполненная данными, которые имеют столбец с числовым идентификатором. С помощью разбиения на части вы можете "разнести" обрабатываемые данные по отдельным потокам с определенным количеством записей. Проект Spring Batch предоставляет разработчикам возможность вмешиваться в эту схему разбиения на части, т.к. она существенно зависит от конкретного сценария использования. Разбиение на части можно делать в локальных потоках или доверить это удаленным исполнителям (что похоже на вариант удаленного разделения на порции).

Один из распространенных сценариев применения пакетной обработки предусматривает чтение файла определенного вида, обычно плоского файла в формате с разделителями (например, запятыми), который впоследствии должен быть загружен в базу данных, с предварительной обработкой каждой записи по мере необходимости. Давайте посмотрим, как можно было бы реализовать такой сценарий в Spring Batch.

Первым делом понадобится добавить обязательные зависимости, которые описаны в табл. 18.1.

**Таблица 18.1. Зависимости для пакетного задания**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| org.springframework | spring-jdbc | 4.0.2.RELEASE | Библиотека Spring JDBC |
| org.springframework | spring-context | 4.0.2.RELEASE | Библиотека Spring Context |
| org.springframework.batch | spring-batch-core | 3.0.1.RELEASE | Библиотека Spring Batch Core |
| org.springframework.batch | spring-batch-infrastructure | 3.0.1.RELEASE | Библиотека Spring Batch Infrastructure |
| commons-dbcp | commons-dbcp | 1.4 | Библиотека пула подключений к базе данных Apache Commons DBCP |
| commons-io | commons-io | 2.4 | Пакет Apache Commons IO для вспомогательных классов ввода-вывода |
| com.h2database | h2 | 1.3.172 | Библиотека встроенной базы данных H2 |
| log4j | log4j | 1.2.17 | Библиотека инфраструктуры регистрации в журнале log4j |

После добавления зависимостей можно приступать к исследованию кода. Для начала мы создаем объект предметной области, который представляет лицо, основываясь на данных из читаемого файла (листинг 18.1).

**Листинг 18.1. Объект предметной области `Person`**

```
package com.apress.prospring4.ch18;

public class Person {
    private String firstName;
    private String lastName;

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getLastName() {
        return lastName;
    }
    @Override
    public String toString() {
        return "firstName: " + firstName + ", lastName: " + lastName;
    }
}
```

Следующей мы создадим реализацию интерфейса `ItemProcessor`, которая будет использоваться для преобразования в верхний регистр имени и фамилии каждого лица, представленного объектом `Person` (листинг 18.2). Обратите внимание, что реализации `ItemProcessor` не требуются в сценарии обработки, ориентированной на порции; нужны только `ItemReader` и `ItemWriter`. Мы применяем здесь `ItemProcessor` в качестве демонстрации того, каким образом можно трансформировать данные перед их записью.

**Листинг 18.2. Реализация интерфейса `ItemProcessor`, предназначенная для преобразования в верхний регистр имени и фамилии лица**

```
package com.apress.prospring4.ch18;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.batch.item.ItemProcessor;

public class PersonItemProcessor implements ItemProcessor<Person, Person> {
  private static Log LOG = LogFactory.getLog(PersonItemProcessor.class);

  @Override
  public Person process(Person person) throws Exception {
    String firstName = person.getFirstName().toUpperCase();
    String lastName = person.getLastName().toUpperCase();
    Person transformedPerson = new Person();
    transformedPerson.setFirstName(firstName);
    transformedPerson.setLastName(lastName);
    LOG.info("Transformed person: " + person + " Into: " + transformedPerson);
    return transformedPerson;
  }
}
```

Далее мы создадим реализацию `StepExecutionListener`, которая находится на уровне шагов и сообщает, сколько записей было сохранено после завершения шага (листинг 18.3). Кроме того, `StepExecutionListener` позволяет при необходимости модифицировать возвращаемое значение `ExitStatus`; чтобы оставить его неизменным, просто возвращается `null`.

**Листинг 18.3. Реализация `StepExecutionListener` для вывода количества сохраненных записей**

```
package com.apress.prospring4.ch18;

import org.springframework.batch.core.ExitStatus;
import org.springframework.batch.core.StepExecution;
import org.springframework.batch.core.listener.
StepExecutionListenerSupport;
public class StepExecutionStatsListener extends StepExecutionListenerSupport
{
   @Override
   public ExitStatus afterStep(StepExecution stepExecution) {
      System.out.println("Wrote: " + stepExecution.getWriteCount()
            + " items in step: " + stepExecution.getStepName());
      return null;
   }
}
```

К этому моменту мы имеем связанные друг с другом основные компоненты, но прежде чем переходить к конфигурированию и вызову кода, давайте взглянем на модель данных и сами данные. Модель данных для задания очень проста (src/main/resources/META-INF/spring/jobs/personJob/support/person.sql) и показана в листинге 18.4.

**Листинг 18.4. Модель данных для примера задания**

```
DROP TABLE people IF EXISTS;

CREATE TABLE people  (
      person_id BIGINT IDENTITY NOT NULL PRIMARY KEY,
      first_name VARCHAR(20),
      last_name VARCHAR(20)
);
```

В листинге 18.5 приведены данные, которые мы загрузим для примера (src/main/resources/META-INF/spring/jobs/personJob/support/test-data.csv).

**Листинг 18.5. Данные для примера задания**

```
Jill,Doe
Joe,Doe
Justin,Matthews
Jane,Matthews
```

Теперь займемся конфигурированием, чтобы определить задание, а также настроить встроенную базу данных и связанные компоненты задания (листинг 18.6). Файл, содержимое которого представлено в листинге 18.6, расположен в src/main/resources/META-INF/spring/jobs/personJob/personJob.xml.

**Листинг 18.6. Конфигурация для примера задания**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:batch="http://www.springframework.org/schema/batch"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="
       http://www.springframework.org/schema/batch
       http://www.springframework.org/schema/batch/spring-batch.xsd
       http://www.springframework.org/schema/jdbc
       http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

   <batch:job id="personJob">
      <batch:step id="step1">
         <batch:tasklet>
            <batch:chunk reader="itemReader" processor="itemProcessor"
                         writer="itemWriter"
                         commit-interval="10"/>
```

```
            <batch:listeners>
               <batch:listener ref="stepExecutionStatsListener"/>
            </batch:listeners>
         </batch:tasklet>
         <batch:fail on="FAILED"/>
         <batch:end on="*"/>
      </batch:step>
   </batch:job>

   <jdbc:embedded-database id="dataSource" type="H2">
     <jdbc:script
       location="classpath:/org/springframework/batch/core/schema-h2.sql"/>
     <jdbc:script
       location="classpath:/META-INF/spring/jobs/personJob/support/person.sql"/>
   </jdbc:embedded-database>

   <bean id="transactionManager"
     class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
     p:dataSource-ref="dataSource"/>

   <batch:job-repository id="jobRepository"/>

   <bean id="jobLauncher"
     class="org.springframework.batch.core.launch.support.SimpleJobLauncher"
     p:jobRepository-ref="jobRepository"/>

   <bean id="stepExecutionStatsListener"
     class="com.apress.prospring4.ch18.StepExecutionStatsListener"/>

   <bean id="itemReader"
         class="org.springframework.batch.item.file.FlatFileItemReader">
      <property name="resource"
        value="classpath:/META-INF/spring/jobs/personJob/support/test-data.csv"/>
      <property name="lineMapper">
        <bean
     class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
               <bean
class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
                  <property name="names" value="firstName,lastName"/>
               </bean>
            </property>
            <property name="fieldSetMapper">
               <bean
class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper">
                  <property name="targetType"
                            value="com.apress.prospring4.ch18.Person"/>
               </bean>
            </property>
         </bean>
      </property>
   </bean>

   <bean id="itemProcessor"
        class="com.apress.prospring4.ch18.PersonItemProcessor"/>

   <bean id="itemWriter"
       class="org.springframework.batch.item.database.JdbcBatchItemWriter">
```

```
      <property name="itemSqlParameterSourceProvider">
         <bean
   class="org.springframework.batch.item.database.BeanPropertyItemSqlParam
eterSourceProvider"/>
      </property>
      <property name="sql"
 value="INSERT INTO people (first_name, last_name) VALUES (:firstName, :lastName)"/>
      <property name="dataSource" ref="dataSource"/>
   </bean>
</beans>
```

В этой конфигурации сначала определяется пакетное задание с использованием схемы `batch`. Мы создаем шаг `step1`, конфигурируем его для обработки, ориентированной на порции, а также определяем реализации `ItemReader`, `ItemProcessor` и `ItemWriter` наряду с прослушивателем. Мы настраиваем встроенную базу данных, определяем бины, связанные с пакетом, и конфигурируем бины обработки.

Наконец, для запуска задания нам необходима тестовая программа, код которой показан в листинге 18.7.

**Листинг 18.7. Тестовая программа для запуска задания**

```
package com.apress.prospring4.ch18;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.Date;

public class PersonJob {
   public static void main(String[] args) throws Exception {
     ApplicationContext applicationContext
             = new ClassPathXmlApplicationContext(
                  "/META-INF/spring/jobs/personJob/personJob.xml");

     Job job = applicationContext.getBean(Job.class);
     JobLauncher jobLauncher = applicationContext.getBean(JobLauncher.class);

     JobParameters jobParameters = new JobParametersBuilder()
             .addDate("date", new Date())
             .toJobParameters();

     jobLauncher.run(job, jobParameters);
   }
}
```

Этот код должен быть вам знаком, т.к. в нем создается контекст, получается ряд бинов и производится вызов их методов. Здесь следует обратить внимание на объект `JobParameters`. Он инкапсулирует параметры, которые применяются для различения одного экземпляра `job` от другого. Идентификация заданий важна при определении последнего состояния задания, если оно играет роль, и это состояние также влияет на другие моменты, такие как возможность перезапуска задания.

В настоящем примере для параметра задания мы просто используем текущую дату. В JobParameters поддерживаются многие типы, и параметры доступны в задании как ссылочные данные.

Теперь все готово к тестированию нашего нового задания. Скомпилируйте код и запустите класс PersonJob. На экране отобразятся журнальные сообщения, самые интересные из которых приведены ниже:

```
[org.springframework.batch.core.launch.support.SimpleJobLauncher] -
<Job: [FlowJob: [name=personJob]] launched with the following
parameters: [{date=1406078297879}]>
[org.springframework.batch.core.job.SimpleStepHandler] -
<Executing step: [step1]>
[com.apress.prospring4.ch18.PersonItemProcessor] - <Transformed person:
firstName: Jill, lastName: Doe Into: firstName: JILL, lastName: DOE>
[com.apress.prospring4.ch18.PersonItemProcessor] - <Transformed person:
firstName: Joe, lastName: Doe Into: firstName: JOE, lastName: DOE>
[com.apress.prospring4.ch18.PersonItemProcessor] - <Transformed person:
firstName: Justin, lastName: Matthews Into: firstName: JUSTIN,
lastName: MATTHEWS>
[com.apress.prospring4.ch18.PersonItemProcessor] - <Transformed person:
firstName: Jane, lastName: Matthews Into: firstName: JANE,
lastName: MATTHEWS>
Wrote: 4 items in step: step1
[org.springframework.batch.core.launch.support.SimpleJobLauncher] -
<Job: [FlowJob: [name=personJob]] completed with the following
parameters: [{date=1406078297879}] and the following status: [COMPLETED]>
```

Это все, что можно сказать о примере. Мы получили возможность построить простое пакетное задание, которое читает данные из файла CSV, трансформирует их посредством ItemProcessor, переводя имя и фамилию лица в верхний регистр, и затем записывает результаты в базу данных. Кроме того, с помощью StepListener осуществляется вывод количества элементов, которые были записаны в рамках шага.

За дополнительной информацией по Spring Batch обращайтесь на страницу проекта по ссылке http://projects.spring.io/spring-batch/.

## Спецификация JSR-352

Проект Spring Batch оказал значительное влияние на спецификацию JSR-352 ("Batch Applications for the Java Platform" — "Пакетные приложения для платформы Java"). Если вы решили задействовать спецификацию JSR-352 в своих заданиях, то будете отмечать все большее и большее количество сходных черт между ними, поэтому должны себя чувствовать вполне комфортно при условии, что ранее пользовались Spring Batch. По большей части Spring Batch и JSR-352 располагают похожими конструкциями, и в версии Spring Batch 3.0 спецификация JSR-352 полностью поддерживается. Подобно Spring Batch, задания JSR-352 конфигурируются через схему XML в том, что называют языком спецификации заданий (Job Specification Language — JSL). Поскольку в JSR-352 определена спецификация и API-интерфейс, никаких готовых компонентов инфраструктуры не предоставляется — в отличие от ситуации, когда применяется Spring Batch. Если вы строго придерживаетесь API-интерфейса JSR-352, то реализация интерфейсов JSR-352 и написание всех компо-

нентов инфраструктуры, таких как `ItemReader` и `ItemWriter`, возлагается полностью на вас.

В этом примере мы преобразуем предыдущий пример пакета для использования языка JSL из JSR-352, но вместо продвижения собственных компонентов инфраструктуры мы задействуем те же самые `ItemReader`, `ItemProcessor` и `ItemWriter`, а также воспользуемся Spring для внедрения зависимостей и т.д. Реализацию задания, которое на 100% совместимо со спецификацией JSR-352, мы оставляем вам в качестве упражнения для самостоятельной проработки.

Как было указано, в этом примере мы будем применять бо́льшую часть кода из примера с чистым проектом Spring Batch, внеся в него небольшие изменения. Если вы этого еще не сделали, то самое время привести пример Spring Batch в рабочее состояние и затем производить в нем модификации, приводимые в данном разделе.

Первым делом поменяем базу данных H2 на HSQLDB; соответствующая зависимость описана в табл. 18.2.

**Таблица 18.2. Зависимость для задания JSR-352**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.hsqldb` | `hsqldb` | 2.3.1 | Библиотека встроенной базы данных HSQLDB |

После этого необходимо переместить файл `personJob.xml` в новый каталог внутри `src/main/resources/META-INF/batch-jobs/`. Это является требованием спецификации JSR-352, и при запуске задания понадобится только имя файла без расширения `.xml`. Путь не понадобится, т.к. файл находится в стандартном каталоге. Затем мы изменим содержимое конфигурационного файла, как показано в листинге 18.8.

**Листинг 18.8. Модифицированный конфигурационный файл для соответствия спецификации JSR-352**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="
       http://www.springframework.org/schema/jdbc
       http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://xmlns.jcp.org/xml/ns/javaee
       http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd">

  <job id="personJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
      <step id="step1">
         <listeners>
            <listener ref="stepExecutionStatsListener"/>
         </listeners>
```

```
        <chunk item-count="10">
            <reader ref="itemReader"/>
            <processor ref="itemProcessor"/>
            <writer ref="itemWriter"/>
        </chunk>
        <fail on="FAILED"/>
        <end on="*"/>
    </step>
</job>

<jdbc:embedded-database id="dataSource" type="HSQL">
    <jdbc:script
location="classpath:/META-INF/spring/jobs/personJob/support/person.sql"/>
</jdbc:embedded-database>

<bean id="stepExecutionStatsListener"
    class="com.apress.prospring4.ch18.StepExecutionStatsListener"/>
<bean id="itemReader"
      class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="resource"
    value="classpath:/META-INF/spring/jobs/personJob/support/test-data.csv"/>
    <property name="lineMapper">
      <bean
  class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
          <property name="lineTokenizer">
            <bean
class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
              <property name="names" value="firstName,lastName"/>
            </bean>
          </property>
          <property name="fieldSetMapper">
            <bean
class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper">
              <property name="targetType"
                        value="com.apress.prospring4.ch18.Person"/>
            </bean>
          </property>
        </bean>
    </property>
</bean>

<bean id="itemProcessor"
      class="com.apress.prospring4.ch18.PersonItemProcessor"/>

<bean id="itemWriter"
     class="org.springframework.batch.item.database.JdbcBatchItemWriter">
    <property name="itemSqlParameterSourceProvider">
      <bean
  class="org.springframework.batch.item.database.
BeanPropertyItemSqlParameterSourceProvider"/>
    </property>
    <property name="sql"
value="INSERT INTO people (first_name, last_name) VALUES (:firstName, :lastName)"/>
    <property name="dataSource" ref="dataSource"/>
</bean>
</beans>
```

Несложно заметить, что конфигурация выглядит очень похожей на конфигурацию из примера Spring Batch за исключением того, что в определении задания используется язык JSL из JSR-352 и можно удалить несколько бинов (`transactionManager`, `jobRepository`, `jobLauncher`), поскольку они уже тем или иным образом предоставлены. Вы также заметите определение дополнительной схемы `jobXML_1.0.xsd`. Поддержка этой схемы получается посредством JAR-файла API-интерфейса JSR-352 и добавляется автоматически в случае применения инструмента построения вроде Maven. Если зависимость необходимо получить вручную, зайдите на страницу проекта, указанную в конце этого раздела. Второй частью, подлежащей изменению, является класс `PersonJob`, т.к. теперь для запуска задания мы используем код, соответствующий спецификации JSR-352 (листинг 18.9).

**Листинг 18.9. Тестовый класс `PersonJob` для примера применения JSR-352**

```
package com.apress.prospring4.ch18;

import org.springframework.batch.core.jsr.launch.JsrJobOperator;

import javax.batch.runtime.BatchStatus;
import javax.batch.runtime.JobExecution;
import java.util.Properties;

public class PersonJob {
    public static void main(String[] args) throws Exception {
        JsrJobOperator jobOperator = new JsrJobOperator();

        long executionId = jobOperator.start("personJob", new Properties());
        JobExecution jobExecution = jobOperator.getJobExecution(executionId);

        waitForJob(jobOperator, jobExecution);
    }

    private static void waitForJob(
            JsrJobOperator jobOperator, JobExecution jobExecution) {
        BatchStatus batchStatus = jobExecution.getBatchStatus();

        while(true) {
            if(batchStatus == BatchStatus.STOPPED ||
                    batchStatus == BatchStatus.COMPLETED
                    || batchStatus == BatchStatus.FAILED) {
                return;
            }

            jobExecution =
              jobOperator.getJobExecution(jobExecution.getExecutionId());
            batchStatus = jobExecution.getBatchStatus();
        }
    }
}
```

Код несколько отличается от других примеров, где мы работали с контекстом `ApplicationContext` и бинами напрямую. Для запуска и управления заданием JSR-352 мы применяем класс `JsrJobOperator`. Чтобы предоставить параметры заданию, вместо объекта `JobParameters` используется объект `Properties`. Применяемый объект `Properties` относится к стандартному классу `java.util.Properties`, а параметры задания должны быть созданы с ключом `String` и значением. Еще одно

интересное изменение связано с методом `waitForJob()`. Спецификация JSR-352 по умолчанию предусматривает запуск всех заданий асинхронно. Таким образом, перед завершением нашей автономной программы мы должны подождать, пока задание не перейдет в приемлемое состояние. Если код выполняется в контейнере, таком как сервер приложений того или иного вида, этот код может не понадобиться. Компиляция и последующий запуск класса `PersonJob` дает следующий журнальный вывод:

```
[org.springframework.batch.core.job.SimpleStepHandler] -
<Executing step: [step1]>
[com.apress.prospring4.ch18.PersonItemProcessor] - <Transformed person:
firstName: Jill, lastName: Doe Into: firstName: JILL, lastName: DOE>
[com.apress.prospring4.ch18.PersonItemProcessor] - <Transformed person:
firstName: Joe, lastName: Doe Into: firstName: JOE, lastName: DOE>
[com.apress.prospring4.ch18.PersonItemProcessor] - <Transformed person:
firstName: Justin, lastName: Matthews Into: firstName: JUSTIN,
lastName: MATTHEWS>
[com.apress.prospring4.ch18.PersonItemProcessor] - <Transformed person:
firstName: Jane, lastName: Matthews Into: firstName: JANE,
lastName: MATTHEWS>
Wrote: 4 items in step: step1
```

Журнальный вывод выглядит практически таким же, как и ранее, но на этот раз для определения и запуска задания мы использовали средства JSR-352, а для внедрения зависимостей — функциональность Spring; кроме того, вместо написания собственных компонентов инфраструктуры мы применяли компоненты, предлагаемые проектом Spring Batch.

За дополнительной информацией по JSR-352 обращайтесь на страницу проекта по ссылке `https://jcp.org/en/jsr/detail?id=352`.

# Проект Spring Integration

Проект *Spring Integration* предлагает готовые реализации хорошо известных шаблонов интеграции корпоративных приложений (Enterprise Integration Pattern — EIP). Этот проект сосредоточен на архитектурах, управляемых сообщениями, предоставляет простую модель для решений интеграции, возможности асинхронного выполнения и слабо связные компоненты; вдобавок он спроектирован с учетом расширяемости и тестируемости.

Центральную роль в инфраструктуре играет сообщение (message). Эта обобщенная оболочка вокруг Java-объекта комбинируется с метаданными, используемыми инфраструктурой (полезной нагрузкой и заголовками), и применяется для определения способа обработки данного объекта.

Канал сообщений (message channel) является *каналом* в архитектуре каналов и фильтров, в рамках которой генераторы отправляют сообщения в канал, а потребители получают их из канала. С другой стороны, конечные точки сообщений (message endpoint) представляют *фильтр* в архитектуре каналов и фильтров, а также соединяют код приложения с инфраструктурой обмена сообщениями. Проект Spring Integration предлагает ряд готовых конечных точек сообщений, среди которых преобразователи (transformer), фильтры (filter), маршрутизаторы (router) и разделители (splitter); каждая конечная точка сообщения предоставляет собственные роли и обязанности.

В Spring Integration имеется также изобилие конечных точек интеграции (свыше 20 на момент написания этой книги), которые описаны в разделе "Endpoint Quick Reference Table" ("Краткий справочник по конечным точкам") документации, доступном по ссылке `http://docs.spring.io/spring-integration/reference/htmlsingle/#endpoint-summary`. Эти конечные точки позволяют подключаться к разнообразным ресурсам, таким как AMQP, файлы, HTTP, JMX, Syslog и Twitter. Выходя за пределы даже того, что предлагает Spring Integration, по ссылке `https://github.com/spring-projects/spring-batch-extensions` доступен еще один проект под названием Spring Integration Extensions, который содержит дополнительные возможности интеграции, в том числе с AWS (Amazon Web Services — вебслужбы Amazon), Apache Kafka, SMPP (Short Message Peer-to-Peer — одноранговый обмен короткими сообщениями) и Voldemort. Находясь между готовыми компонентами и компонентами проекта расширений, Spring Integration предоставляет обилие стандартных компонентов, поэтому вероятность того, что возникнет необходимость в написании собственных компонентов, значительно снижается.

В настоящем примере мы собираемся взять за основу предыдущие примеры пакетов, но на этот раз задействовать проект Spring Integration, чтобы продемонстрировать его использование для проведения мониторинга каталога через заданные интервалы. Когда в каталог поступает файл, мы обнаруживаем его и запускаем пакетное задание для его обработки.

Здесь мы снова планируем построить "чистый" проект Spring Batch, с которого начиналась данная глава. Прежде чем продолжить, освежите его в памяти, т.к. далее мы будем рассматривать только новые классы и изменения в конфигурации.

Для начала понадобится добавить несколько новых зависимостей для самого проекта Spring Integration (табл. 18.3).

**Таблица 18.3. Зависимости для проекта Spring Integration**

| Идентификатор группы | Идентификатор артефакта | Версия | Описание |
|---|---|---|---|
| `org.springframework.batch` | `spring-batch-integration` | 3.0.1.RELEASE | Библиотека Spring Batch Integration |
| `org.springframework.integration` | `spring-integration-file` | 4.0.3.RELEASE | Библиотека Spring Integration File |

Теперь давайте создадим класс, который действует в качестве *преобразователя* Spring Integration. Этот преобразователь будет получать из входящего канала сообщение, представляющее найденный файл, и запускать с ним пакетное задание (листинг 18.10).

**Листинг 18.10. Преобразователь Spring Integration**

```
package com.apress.prospring4.ch18;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.integration.launch.JobLaunchRequest;
import org.springframework.messaging.Message;
```

```
import java.io.File;

public class MessageToJobLauncher {
   private Job job;
   private String fileNameKey;

   public MessageToJobLauncher(Job job, String fileNameKey) {
      this.job = job;
      this.fileNameKey = fileNameKey;
   }

   public JobLaunchRequest toRequest(Message<File> message) {
     JobParametersBuilder jobParametersBuilder =
          new JobParametersBuilder();
     jobParametersBuilder.addString(fileNameKey,
                                    message.getPayload().getAbsolutePath());

     return new JobLaunchRequest(job, jobParametersBuilder.toJobParameters());
   }
}
```

После этого мы модифицируем конфигурационный файл `src/main/resources/META-INF/spring/jobs/personJob/personJob.xml`, добавив в него фрагменты, которые связаны с интеграцией (листинг 18.11).

**Листинг 18.11. Измененная конфигурация для интеграции Spring Integration File**

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:batch="http://www.springframework.org/schema/batch"
    xmlns:batch-int="http://www.springframework.org/schema/batch-integration"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:int="http://www.springframework.org/schema/integration"
    xmlns:int-file="http://www.springframework.org/schema/integration/file"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="
       http://www.springframework.org/schema/batch
       http://www.springframework.org/schema/batch/spring-batch.xsd
       http://www.springframework.org/schema/batch-integration
       http://www.springframework.org/schema/batch-integration/
spring-batch-integration.xsd
       http://www.springframework.org/schema/jdbc
       http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/integration
       http://www.springframework.org/schema/integration/
spring-integration.xsd
       http://www.springframework.org/schema/integration/file
       http://www.springframework.org/schema/integration/file/
spring-integration-file.xsd">
```

```
   <batch:job id="personJob">
      <batch:step id="step1">
         <batch:tasklet>
            <batch:chunk reader="itemReader" processor="itemProcessor"
                         writer="itemWriter"
                         commit-interval="10"/>
            <batch:listeners>
               <batch:listener ref="stepExecutionStatsListener"/>
            </batch:listeners>
         </batch:tasklet>
         <batch:fail on="FAILED"/>
         <batch:end on="*"/>
      </batch:step>
   </batch:job>

   <jdbc:embedded-database id="dataSource" type="H2">
      <jdbc:script
        location="classpath:/org/springframework/batch/core/schema-h2.sql"/>
      <jdbc:script
  location="classpath:/META-INF/spring/jobs/personJob/support/person.sql"/>
   </jdbc:embedded-database>
   <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
         p:dataSource-ref="dataSource"/>
   <batch:job-repository id="jobRepository"/>
   <bean id="jobLauncher"
class="org.springframework.batch.core.launch.support.SimpleJobLauncher"
         p:jobRepository-ref="jobRepository"/>
   <bean id="stepExecutionStatsListener"
        class="com.apress.prospring4.ch18.StepExecutionStatsListener"/>
   <bean id="itemReader"
        class="org.springframework.batch.item.file.FlatFileItemReader"
        scope="step">
      <property name="resource"
                value="file://#{jobParameters['file.name']}"/>
      <property name="lineMapper">
         <bean
class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
               <bean
class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
                  <property name="names" value="firstName,lastName"/>
               </bean>
            </property>
            <property name="fieldSetMapper">
               <bean
class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper">
                  <property name="targetType"
                            value="com.apress.prospring4.ch18.Person"/>
               </bean>
            </property>
         </bean>
      </property>
   </bean>
```

```
    <bean id="itemProcessor"
          class="com.apress.prospring4.ch18.PersonItemProcessor"/>
    <bean id="itemWriter"
class="org.springframework.batch.item.database.JdbcBatchItemWriter">
       <property name="itemSqlParameterSourceProvider">
          <bean class="org.springframework.batch.item.database.
BeanPropertyItemSqlParameterSourceProvider"/>
       </property>
       <property name="sql"
  value="INSERT INTO people (first_name, last_name) VALUES (:firstName,
:lastName)"/>
       <property name="dataSource" ref="dataSource"/>
    </bean>
    <int:channel id="inbound"/>
    <int:channel id="outbound"/>
    <int:channel id="loggingChannel"/>
    <int-file:inbound-channel-adapter
              id="inboundFileChannelAdapater" channel="inbound"
                directory="file:/tmp/people/" filename-pattern="*.csv">
       <int:poller fixed-rate="1000"/>
    </int-file:inbound-channel-adapter>
    <int:transformer input-channel="inbound"
                     output-channel="outbound">
       <bean class="com.apress.prospring4.ch18.MessageToJobLauncher">
          <constructor-arg ref="personJob"/>
          <constructor-arg value="file.name"/>
       </bean>
    </int:transformer>
    <batch-int:job-launching-gateway request-channel="outbound"
              reply-channel="loggingChannel"/>
    <int:logging-channel-adapter channel="loggingChannel"/>
</beans>
```

Основные добавления к конфигурации касаются разделов, имеющих префиксы в виде пространств имен `int:` и `batch-int:`. Сначала мы создаем несколько именованных каналов для передачи по ним данных. Затем мы конфигурируем адаптер входящего канала `inbound-channel-adapter` специально для наблюдения за указанным каталогом с интервалом в 1 секунду. Далее мы настраиваем бин преобразователя, который получает файлы как стандартные объекты `java.io.File`, помещенные внутрь сообщения. После этого мы конфигурируем шлюз запуска заданий `job-launching-gateway`, который принимает запросы на запуск заданий от преобразователя для действительной активизации пакетного задания. Наконец, мы создаем адаптер канала регистрации в журнале `logging-channel-adapter`, который будет выводить информационные уведомления после завершения задания. По сконфигурированным атрибутам канала можно заметить, что каждый компонент либо потребляет сообщения, либо их генерирует или же делает то и другое через канал.

А теперь давайте создадим простой тестовый класс, загружающий наш конфигурационный файл. Тестовый класс всего лишь загружает контекст приложения с нашей конфигурацией и остается в функционирующем состоянии до тех пор, пока вы не уничтожите процесс, т.к. он непрерывно опрашивает указанный каталог. Код класса `FileWatcher` приведен в листинге 18.12.

**Листинг 18.12. Тестовый класс для наблюдения за файлами**

```
package com.apress.prospring4.ch18;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class FileWatcher {
    public static void main(String[] args) throws Exception {
        ApplicationContext applicationContext
                = new ClassPathXmlApplicationContext(
                    "/META-INF/spring/jobs/personJob/personJob.xml");
    }
}
```

Скомпилируйте код и запустите класс `FileWatcher`. Когда приложение начинает свою работу, вы можете заметить, что на экран выводятся некоторые журнальные сообщения, но больше ничего не происходит. Причина в том, что адаптер Spring Integration File ожидает помещения файлов в сконфигурированное местоположение в рамках интервала опроса, и до тех пор, пока он не обнаружит там файл, ничего происходить не будет. Создайте простой файл CSV с содержимым вроде показанного ниже:

```
Chris, Schaefer
```

Затем переместите этот файл в сконфигурированное местоположение (`/tmp/people/`), назначив ему любое имя и расширение `.csv` (например, `people.csv`). После переноса этого файла в подходящее место переключитесь обратно на консоль, в которой выполняется приложение `FileWatcher`, и вы увидите вывод, похожий на приведенный далее:

```
[org.springframework.integration.file.FileReadingMessageSource] -
<Created message: [[Payload File content=/tmp/people/b.
csv][Headers={id=0c407bf8-d16d-adbc-c8d7-af3023d404a9,
timestamp=1406080790546}]]>
[org.springframework.batch.core.launch.support.SimpleJobLauncher] -
<Job: [FlowJob: [name=personJob]] launched with the following
parameters: [{file.name=/tmp/people/b.csv}]>
[org.springframework.batch.core.job.SimpleStepHandler] -
<Executing step: [step1]>
[com.apress.prospring4.ch18.PersonItemProcessor] - <Transformed person:
firstName: Chris, lastName: Schaefer Into: firstName: CHRIS,
lastName: SCHAEFER>
Wrote: 1 items in step: step1
[org.springframework.batch.core.launch.support.SimpleJobLauncher] -
<Job: [FlowJob: [name=personJob]] completed with the following parameters:
[{file.name=/tmp/people/b.csv}] and the following status: [COMPLETED]>
[org.springframework.integration.handler.LoggingHandler] -
<JobExecution: id=1, version=2, startTime=Tue Jul 22 21:59:50 EDT 2014,
endTime=Tue Jul 22 21:59:50 EDT 2014, lastUpdated=Tue Jul 22 21:59:50 EDT
2014, status=COMPLETED, exitStatus=exitCode=COMPLETED;
exitDescription=, job=[JobInstance: id=1, version=0, Job=[personJob]],
jobParameters=[{file.name=/tmp/people/b.csv}]>
```

На основании журнальных сообщений можно сделать вывод, что Spring Integration удалось обнаружить файл, создать сообщение, запустить пакетное зада-

ние, трансформирующее содержимое файла CSV, и записать это содержимое в базу данных, находящуюся в памяти. Хотя этот пример прост, он демонстрирует способ построения сложных и несвязанных рабочих потоков между разнообразными типами приложений с применением Spring Integration.

За дополнительной информацией по Spring Integration обращайтесь на страницу проекта по ссылке `http://projects.spring.io/spring-integration/`.

# Проект Spring XD

Проект *Spring XD* — это расширяемая служба времени выполнения, предназначенная для получения распределенных данных, анализа в реальном времени, пакетной обработки и экспортирования данных. Spring XD построен на основе многих существующих проектов, входящих в набор Spring, наиболее значимыми из которых являются сама платформа Spring Framework, Spring Batch и Spring Integration. Цель Spring XD заключается в обеспечении унифицированного способа интеграции множества систем в связное решение больших данных, которое помогает преодолеть сложность распространенных сценариев использования.

Служба Spring XD может функционировать в автономном режиме с единственным узлом, который обычно предназначен для целей разработки и тестирования, а также в полностью распределенном режиме, предоставляя возможность иметь высокодоступные ведущие узлы наряду с несколькими рабочими узлами.

Проект Spring XD позволяет управлять этими узлами через интерфейс оболочки (с применением оболочки Spring), а также графический веб-интерфейс. Посредством упомянутых интерфейсов можно определять метод сборки различных компонентов для удовлетворения потребностей обработки данных либо с использованием синтаксиса типа DSL в приложении оболочки, либо за счет ввода данных в веб-интерфейсе, который самостоятельно построит нужные определения.

Язык DSL в Spring XD основан на нескольких концепциях, таких как потоки, модули, источники, процессоры, приемники и задания. Объединяя эти компоненты вместе с применением лаконичного синтаксиса, легко создавать потоки для соединения друг с другом разнообразных технологий, которые позволяют получать данные, обрабатывать их и в конечном итоге выводить данные во внешний источник или даже запускать пакетное задание для дальнейшей обработки данных. Давайте кратко рассмотрим эти концепции.

- Поток (stream) определяет способ протекания данных от источника к приемнику и возможного их прохождения через любое количество процессоров. Для определения потока применяется язык DSL; например, базовое определение "источник-приемник" может выглядеть как `http | file`.
- Модуль (module) инкапсулирует многократно используемую единицу работы, из которых состоят потоки. Модули объединяются в категории по типам, основанным на их ролях. На время написания этой книги проект Spring XD содержал модули типа источника, процессора, приемника и задания.
- Источник (source) в Spring XD либо опрашивает внешний ресурс, либо активизируется каким-то событием. Источники предоставляют вывод только компонентам, находящимся по ходу движения данных, и первым модулем в потоке должен быть источник.

- Процессор (processor) по своей природе похож на то, что мы видели в Spring Batch. Роль процессора заключается в получении ввода, выполнении трансформации или бизнес-логики в отношении предоставленного объекта и возвращении вывода.
- На другой стороне источника расположен приемник (sink), который принимает входной источник и выводит данные в свой целевой ресурс. Приемник является заключительной остановкой в потоке.
- Задание (job) — это модули, которые определяют задание Spring Batch. Такие задания определяются точно так же, как было описано в начале этой главы, и разворачиваются в Spring XD для обеспечения возможностей пакетной обработки.
- Ответвление (tap) прослушивает данные, проходящие через поток, и также позволяет обрабатывать ответвленные данные в отдельном потоке. Концепция ответвления подобна шаблонам интеграции корпоративных приложений WireTap.

Как и можно было ожидать, проект Spring XD предоставляет несколько готовых источников, процессоров, приемников, заданий и ответвлений. Кроме того, разработчики не ограничены только тем, что предлагается в готовом виде, и могут также строить собственные модули и компоненты. За дополнительной информацией, касающейся создания собственных модулей и компонентов, обращайтесь в справочное руководство:

- Модули — `http://docs.spring.io/spring-xd/docs/current/reference/html/#_creating_a_module`
- Источники — `http://docs.spring.io/spring-xd/docs/current/reference/html/#creating-a-source-module`
- Процессоры — `http://docs.spring.io/spring-xd/docs/current/reference/html/#creating-a-processor-module`
- Приемники — `http://docs.spring.io/spring-xd/docs/current/reference/html/#creating-a-sink-module`
- Задания — `http://docs.spring.io/spring-xd/docs/current/reference/html/#creating-a-job-module`

В рассмотренном далее примере мы покажем, как применять готовые компоненты Spring XD для воспроизведения того, что было создано в примерах демонстрации Spring Batch и Spring Integration, причем все это будет делаться с помощью простой конфигурации командной строки, используя оболочку Spring XD и язык DSL.

Прежде чем начать, потребуется установить Spring XD. Различные методы установки Spring XD описаны в разделе "Getting Started" ("Начало работы") руководства пользователя по ссылке `http://docs.spring.io/spring-xd/docs/1.0.0.RC1/reference/html/#getting-started`. Выбор метода установки зависит от личных предпочтений и не оказывает влияния на пример. Установив Spring XD, запустите исполняемую среду в режиме одиночного узла, как указано в документации.

Для воспроизведения посредством Spring XD того, что было создано в примерах применения Spring Batch и Spring Integration, понадобится выполнить лишь не-

сколько базовых задач. Для начала необходимо создать импортируемый файл CSV с показанным ниже содержимым и поместить его в `/tmp/people.csv`:

```
Jill,Doe
Joe,Doe
Justin,Matthews
Jane,Matthews
```

В консоли оболочки Spring XD введите следующую команду:

```
job create myjob --definition "filejdbc
--resources=file:///tmp/people.csv --names=firstname,lastname
--tableName=people --initializeDatabase=true" --deploy
```

После нажатия клавиши <Enter> должно появиться сообщение `Successfully created and deployed job 'myjob'` (Задание `myjob` успешно создано и развернуто). Если такое сообщение не отобразилось, просмотрите консольный вывод на терминале, где запускался контейнер Spring XD с единственным узлом, чтобы получить дополнительные сведения.

К этому моменту новое определения задания создано, но пока еще ничего не произошло, т.к. задание не было запущено. Для его запуска введите в оболочке приведенную ниже команду:

```
job launch myjob
```

Оболочка должна отреагировать выдачей сообщения о том, что запрос на запуск задания `myJob` успешно отправлен.

Анализируя введенный код DSL, среда Spring XD определяет, что мы хотим создать пакетное задание, которое читает данные из файла и выводит их в базу данных через JDBC с помощью оператора `job create` и источника `filejdbc`. Она также автоматически создает таблицу с именем, полученным из параметра `tableName`, и именами столбцов, извлеченными из параметра `names`, и читает данные, используя параметр `resources`, в котором был предоставлен путь к нашему файлу CSV.

Чтобы проверить импортированные данные, подключитесь к базе данных, указанной во время установки (либо встроенной, либо настоящей СУРБД) с применением излюбленного инструмента и выберите для просмотра записи из таблицы `People`. Если вы не видите данных, исследуйте журнальные записи на консоли, где функционирует контейнер Spring XD с единственным узлом.

Итак, мы ввели в оболочке две команды, но нам не пришлось писать какой-то код или заниматься сложным конфигурированием. Тем не менее, мы импортировали содержимое нашего файла CSV в базу данных, приложив минимальные усилия. Это было достигнуто за счет использования предварительно построенного пакетного задания Spring XD, которое определялось с помощью простого синтаксиса командной строки на основе языка DSL, и последующего запуска задания в оболочке.

Как видите, Spring XD предлагает широкий спектр функциональности в готовом виде и устраняет необходимость в реализации распространенных сценариев использования. В целях дальнейших исследований проекта Spring XD мы оставляем добавление преобразования имени и фамилии, которое выполнялось в предшествующих примерах, в качестве упражнения для самостоятельной проработки.

Дополнительную информацию о проекте Spring XD можно получить на странице проекта по ссылке `http://projects.spring.io/spring-xd/`.

# Проект Spring Boot

Проект *Spring Boot* предназначен для упрощения начального освоения принципов, в соответствие с которыми строится приложение с применением Spring. Он делает предположения на основе вручную введенных зависимостей и предлагает ряд самых распространенных средств, необходимых для большинства приложений, в том числе метрики и проверки работоспособности.

Для достижения цели по упрощению разработки в Spring Boot принят подход с предоставлением первоначальных файлов POM (Project Object Model — объектная модель проекта), описывающих разнообразные типы приложений, которые уже содержат подходящие зависимости и версии. Это означает меньшие затраты времени перед тем, как можно будет приступить к работе. Проект Spring Boot не требует какой-либо конфигурации, которая должна быть написана на XML.

В рамках рассматриваемого примера мы создадим традиционное веб-приложение "Hello World!". Вас может удивить, насколько мало кода требуется для этого по сравнению с типовым веб-приложением, реализованным на языке Java.

Обычно мы начинали примеры с определения зависимостей, необходимых для добавления к проекту. Частью модели упрощения Spring Boot является подготовка всех зависимостей, и в случае использования Maven, например, эту функциональность можно получить с применением родительского файла POM.

Давайте первым делом создадим сам файл POM для фактического проекта (листинг 18.13).

**Листинг 18.13. Файл POM для примера веб-приложения**

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.apress.prospring4.ch18</groupId>
   <artifactId>boot</artifactId>
   <version>4.0-SNAPSHOT</version>
   <name>boot</name>
   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>1.1.4.RELEASE</version>
   </parent>
   <dependencies>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-web</artifactId>
      </dependency>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-actuator</artifactId>
      </dependency>
   </dependencies>
```

```
    <properties>
        <start-class>com.apress.prospring4.ch18.Application</start-class>
    </properties>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Этот файл POM позаботится о распознавании всех зависимостей, которые нужны для разрабатываемого проекта веб-приложения. Нам понадобится только создать два простых класса. Сначала мы создадим класс веб-контроллера, как показано в листинге 18.14.

**Листинг 18.14. Класс `HelloWorldController`**

```
package com.apress.prospring4.ch18;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class HelloWorldController {
    @RequestMapping("/")
    public String helloWorld() {
        return "Hello world";
    }
}
```

Это типичный класс контроллера Spring MVC, с которым мы имели дело в главе 16, и он должен выглядеть хорошо знакомым. Вы можете заметить, что вместо `@Controller` используется `@RestController`. Аннотация `@RestController` применяется для удобства, т.к. она уже является стереотипом `@Controller`, к тому же методы `@RequestMapping` по умолчанию ожидают семантику `@ResponseBody`. Затем мы создаем класс начальной загрузки за счет использования простого метода `main()`; соответствующий код приведен в листинге 18.15.

**Листинг 18.15. Класс начальной загрузки приложения**

```
package com.apress.prospring4.ch18;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

В этот момент может возникнуть вопрос: а где же конфигурационный файл `web.xml` и все остальные компоненты, которые должны быть созданы для базового веб-приложения? Дело в том, что в предыдущих листингах вы уже определили все необходимое! Не верите? Скомпилируйте проект и запустите класс `Application` до тех пор, пока не увидите журнальное сообщение, которое указывает, что приложение успешно стартовало.

Если вы заглянете в сгенерированные журнальные файлы, то увидите, что при таком небольшом объеме кода в приложении происходит очень многое. Наиболее примечательным является то, что код функционирует под управлением сервера Tomcat, к тому же определены разнообразные конечные точки, такие как проверки работоспособности, выходная информация о среде и метрики. Переход на URL вида `http://localhost:8080` приводит к вполне ожидаемому отображению веб-страницы "Hello World!". А теперь взгляните на заранее сконфигурированные конечные точки (например, `http://localhost:8080/health`), которые возвращают JSON-представление состояния приложения. Вдобавок загрузите страницу `http://localhost:8080/metrics`, чтобы лучше понять различные собираемые метрики, такие как размер кучи, сборка мусора и т.д.

Как вы уже наверняка заметили в этом простом примере, Spring Boot радикально упрощает процесс создания веб-приложений (или приложений любого типа, если уж на то пошло). Ушли в прошлое те дни, когда для построения простого веб-приложения приходилось конфигурировать многочисленные файлы, и благодаря встроенному контейнеру сервлетов, готовому обслуживать ваше веб-приложение, все необходимое "просто работает".

Хотя мы продемонстрировали очень простой пример, имейте в виду, что проект Spring Boot не ограничивает вас применением только вариантов, которые он предлагает; Spring Boot всего лишь выбирает стандартные настройки. Если вы хотите использовать не встроенный сервер Tomcat, а Jetty, модифицируйте файл POM, исключив модуль запуска Tomcat из зависимости `spring-boot-starter-web`. С помощью инструмента, подобного дереву зависимостей Maven, легко визуализировать зависимости, задействованные в проекте. Кроме того, Spring Boot предоставляет много других зависимостей запуска для других типов приложений, поэтому рекомендуется внимательно почитать документацию по Spring Boot.

За дополнительной информацией по проекту Spring Boot обращайтесь на страницу проекта по ссылке `http://projects.spring.io/spring-boot/`.

# Резюме

В этой главе мы представили высокоуровневый обзор нескольких проектов из набора Spring. Были кратко рассмотрены проекты Spring Batch, JSR-352, Spring Integration, Spring XD и Spring Boot, каждый из которых предлагает свои уникальные возможности, предназначенные для упрощения решения специфичных задач разработчиками. Некоторые проекты являются новыми, тогда как другие доказали свою надежность и согласованность, выступая в качестве фундамента для других инфраструктур. Мы рекомендуем изучить эти проекты более глубоко, поскольку уверены в том, что в целом они значительно упростят работу над вашими проектами Java.

# Предметный указатель