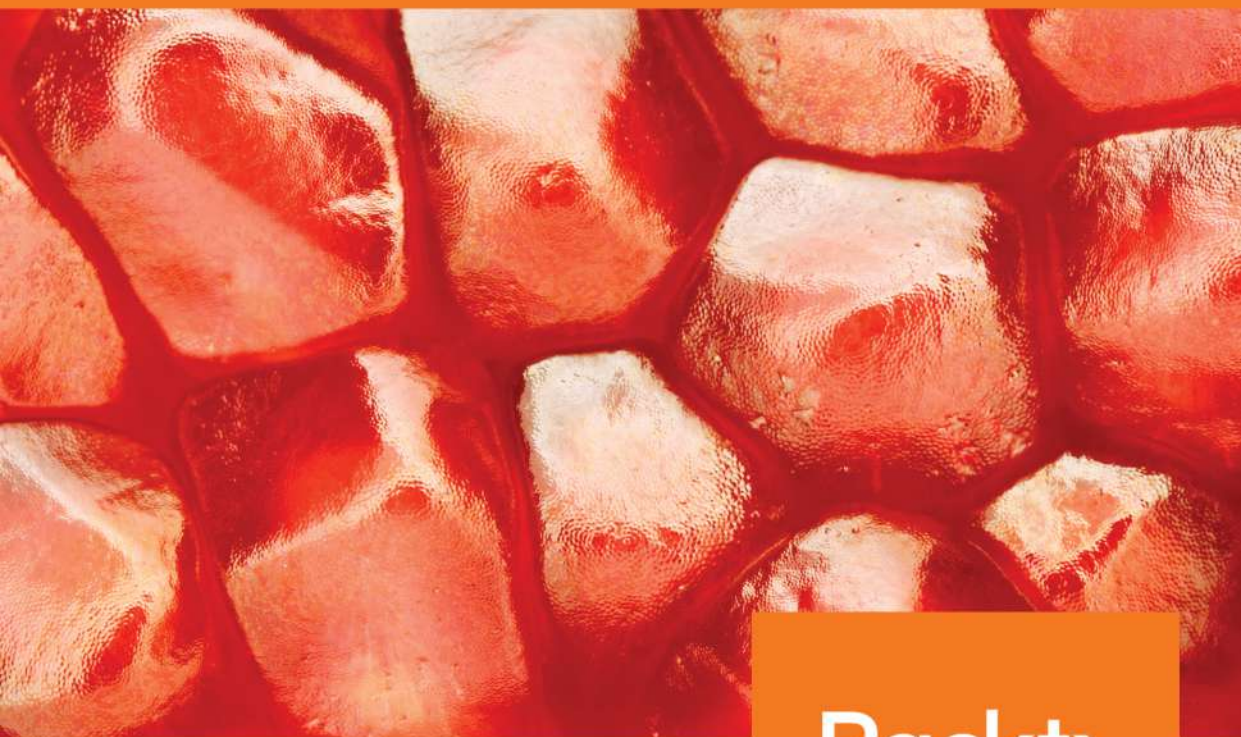


Spring

ВСЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Динеш Раджпут



Packt>

Spring 5 Design Patterns

Master efficient application development with patterns such as proxy, singleton, the template method, and more

Dinesh Rajput



BIRMINGHAM - MUMBAI

Spring

ВСЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Динеш Раджпут



Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2019

Динеш Раджпут
Spring. Все паттерны проектирования
Серия «Библиотека программиста»

Перевели с английского *Е. Иконникова, И. Пальти*

Заведующая редакцией	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>О. Сивченко</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Н. Хлебина</i>
Художественный редактор	<i>С. Заматевская</i>
Корректоры	<i>Е. Павлович, Т. Радецкая</i>
Верстка	<i>Г. Блинов</i>

ББК 32.988.02-018
УДК 004.738.2

Раджпут Динеш

P15 Spring. Все паттерны проектирования. — СПб.: Питер, 2019. — 320 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-0935-7

В этой книге дается обзор фреймворка Spring 5 и паттернов проектирования для него. Объясняется принцип внедрения зависимостей (dependency injection), играющий ключевую роль при создании слабосвязанного кода во фреймворке Spring. Затем рассматриваются классические паттерны «Банды четырех» при проектировании приложений на Spring. В следующих частях книги автор рассматривает паттерны аспектно-ориентированного программирования (AOP), шаблоны JDBC, позволяющие абстрагировать доступ к базе данных. В заключительных главах книги автор исследует работу с MVC, реактивные шаблоны проектирования и паттерны проектирования, применяемые при конкурентном и параллельном программировании в Spring.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1788299459 англ.

© Packt Publishing 2017. First published in the English language under the title «Spring 5 Design Patterns — (9781788299459)»

ISBN 978-5-4461-0935-7

© Перевод на русский язык ООО Издательство «Питер», 2019

© Издание на русском языке, оформление ООО Издательство «Питер», 2019

© Серия «Библиотека программиста», 2019

Права на издание получены по соглашению с Packt Publishing. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 12.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —

Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 07.12.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 25,800. Тираж 700. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Краткое содержание

Об авторе	12
О рецензенте.....	13
Предисловие.....	14
Глава 1. Знакомство с Spring Framework 5.0 и паттернами проектирования.....	18
Глава 2. Обзор паттернов проектирования GoF: базовые паттерны проектирования ...	45
Глава 3. Соображения по поводу структурных и поведенческих паттернов	68
Глава 4. Связывание компонентов с помощью паттерна внедрения зависимостей...	105
Глава 5. Жизненный цикл компонентов и используемые паттерны.....	138
Глава 6. Аспектно-ориентированное программирование в Spring с помощью паттернов «Заместитель» и «Декоратор»	159
Глава 7. Доступ к базе данных с помощью фреймворка Spring и JDBC-реализаций паттерна «Шаблонный метод»	184
Глава 8. Доступ к базе данных с помощью паттернов ORM и транзакций	206
Глава 9. Улучшение производительности приложения с помощью паттернов кэширования	227
Глава 10. Реализация паттерна MVC в веб-приложениях с помощью фреймворка Spring	245
Глава 11. Реализация реактивных паттернов проектирования	288
Глава 12. Реализация конкурентных паттернов.....	312

Оглавление

Об авторе	12
О рецензенте	13
Предисловие	14
Темы, рассматриваемые в книге	14
Что нужно для чтения книги	16
Принятые обозначения	16
Скачивание кода примеров	17
Об ошибках	17
Глава 1. Знакомство с Spring Framework 5.0 и паттернами проектирования	18
Знакомство с фреймворком Spring	18
Упрощение разработки приложений благодаря применению Spring и паттернов	20
Использование широчайших возможностей паттерна POJO	21
Внедрение зависимостей между POJO	22
Использование паттерна внедрения зависимостей для зависимых компонентов	25
Применение объектов для сквозных задач	29
Применение шаблонов для устранения стереотипного кода	33
Использование контейнеров Spring для управления компонентами с помощью паттерна «Фабрика»	36
Фабрики компонентов	36
Контексты приложений	37
Создание контейнера с контекстом приложения	37
Жизнь компонента в контейнере	38
Модули фреймворка Spring	40
Core Container Spring	41
Модуль AOP	42
Spring DAO — доступ к данным и интеграция	42
ORM	42
Web MVC	42
Новые возможности Spring Framework 5.0	43
Резюме	44
Глава 2. Обзор паттернов проектирования GoF: базовые паттерны проектирования ...	45
Возможности паттернов проектирования	46
Обзор часто используемых паттернов проектирования GoF	47

Порождающие паттерны проектирования	48
Паттерн проектирования «Фабрика»	49
Паттерн проектирования «Абстрактная фабрика»	52
Паттерн проектирования «Одиночка»	58
Паттерн проектирования «Прототип»	60
Паттерн проектирования «Строитель»	63
Резюме	66
Глава 3. Соображения по поводу структурных и поведенческих паттернов	68
Базовые паттерны проектирования	68
Структурные паттерны проектирования	69
Поведенческие паттерны проектирования	94
Паттерны проектирования J2EE	103
Резюме	104
Глава 4. Связывание компонентов с помощью паттерна внедрения зависимостей...	105
Паттерн внедрения зависимостей	106
Решение проблем с помощью паттерна внедрения зависимостей	106
Виды внедрения зависимостей	111
Внедрение зависимостей через конструктор	111
Внедрение зависимости через сеттер	113
Сравнение внедрений через конструктор и сеттер, а также рекомендуемые практики	115
Описание конфигурации паттерна внедрения зависимостей с помощью Spring	115
Использование паттерна внедрения зависимостей с Java-конфигурацией	117
Создание класса Java-конфигурации: AppConfig.java	117
Объявления компонентов Spring в классе конфигурации	117
Внедрение компонентов Spring	118
Оптимальный подход к настройке паттерна внедрения зависимостей с помощью Java	119
Использование паттерна внедрения зависимостей с XML-конфигурацией	120
Создание файла XML-конфигурации	120
Объявление компонентов Spring в XML-файле	121
Внедрение компонентов Spring	121
Использование паттерна внедрения зависимостей с конфигурацией на основе аннотаций	124
Что такое стереотипные аннотации	124
Автосвязывание зависимостей и неоднозначности	131
Рекомендуемые практики для конфигураций паттерна DI	135
Резюме	137
Глава 5. Жизненный цикл компонентов и используемые паттерны	138
Жизненный цикл компонента Spring и его фазы	139
Фаза инициализации	140
Фаза использования компонентов	150
Фаза уничтожения компонента	151

Области видимости компонентов	153
Одиночная область видимости	154
Прототипная область видимости компонента	155
Сеансовая область видимости компонента	155
Запросная область видимости компонента	155
Другие области видимости в Spring	156
Резюме	158
Глава 6. Аспектно-ориентированное программирование в Spring с помощью паттернов «Заместитель» и «Декоратор»	159
Паттерн «Заместитель» в Spring	160
Что такое сквозная функциональность	162
Что такое аспектно-ориентированное программирование	162
Проблемы, решаемые с помощью AOP	163
Как AOP решает проблемы	166
Основные понятия и терминология AOP	167
Совет	167
Точка соединения	168
Срез	169
Аспект	169
Вплетение	169
Задание срезов	170
Создание аспектов	172
Реализация советов	174
Тип совета: до	174
Тип совета: после возврата	175
Тип совета: после исключения	176
Тип совета: после	177
Тип совета: везде	178
Описание аспектов с помощью XML-конфигурации	180
AOP-прокси	181
Резюме	183
Глава 7. Доступ к базе данных с помощью фреймворка Spring и JDBC-реализаций паттерна «Шаблонный метод»	184
Оптимальный подход к проектированию доступа к данным	185
Задача управления ресурсами	187
Реализация паттерна проектирования «Шаблонный метод»	188
Настройка источника данных и паттерн «Пул объектов»	192
Задание настроек источника данных с помощью JDBC-драйвера	193
Конфигурирование источника данных с помощью пула соединений	194
Реализация паттерна «Строитель» для создания встроенного источника данных ..	196
Абстрагирование доступа к базе данных с помощью паттерна DAO	196
Реализация паттерна DAO с помощью фреймворка Spring	197
Работа с JdbcTemplate	198
Когда использовать JdbcTemplate	199

Рекомендуемые практики JDBC и настройки JdbcTemplate	204
Резюме	205
Глава 8. Доступ к базе данных с помощью паттернов ORM и транзакций	206
Фреймворки ORM и используемые в них паттерны.....	207
Управление ресурсами и транзакциями	209
Единообразная обработка и трансляция исключений	209
Паттерн «Объект доступа к данным»	210
Создание объектов DAO в Spring с помощью паттерна проектирования «Фабрика»	211
Паттерн «Отображение данных»	212
Паттерн «Модель предметной области»	213
Прокси для паттерна «Отложенная загрузка»	214
Паттерн «Шаблонный метод» для поддержки Hibernate в Spring	214
Интеграция Hibernate со Spring	214
Задание настроек объекта SessionFactory фреймворка Hibernate в контейнере Spring	214
Реализация объектов DAO на основе простого API Hibernate	216
Стратегии управления транзакциями в Spring.....	217
Декларативное задание границ и реализация транзакций.....	219
Развертывание диспетчера транзакций	220
Программное задание границ и реализация транзакций.....	223
Рекомендуемые практики для ORM Spring и модуля транзакций приложения	225
Резюме	226
Глава 9. Улучшение производительности приложения с помощью паттернов кэширования	227
Что такое кэш.....	228
Абстракция кэша	228
Включение возможности кэширования посредством паттерна «Заместитель»	229
Включение прокси для кэширования с помощью аннотаций	230
Включение прокси для кэширования с помощью пространства имен XML.....	231
Декларативное кэширование с помощью аннотаций.....	232
Аннотация @Cacheable	232
Аннотация @CachePut	233
Аннотация @CacheEvict	235
Аннотация @Caching	236
Аннотация @CacheConfig.....	236
Декларативное кэширование с помощью XML.....	237
Настройка хранилища кэша	240
Сторонние диспетчеры кэша.....	240
EhCache.....	240
XML-конфигурация	241
Создание пользовательских аннотаций кэширования	242
Лучшие рекомендуемые практики для веб-приложений.....	242
Резюме	244

Глава 10. Реализация паттерна MVC в веб-приложениях с помощью фреймворка Spring	245
Реализация паттерна MVC в веб-приложении	246
Архитектура «Модель 2» паттерна MVC в Spring	247
Паттерн проектирования «Единая точка входа»	248
Включение возможностей MVC Spring	257
Реализация контроллеров	259
Отображение запросов с помощью аннотации @RequestMapping	260
Передача данных модели представлению	264
Принятие параметров запроса	265
Обработка форм веб-страницы	268
Реализация контроллера обработки форм	270
Привязка данных с помощью паттерна проектирования «Команда»	272
Проверка корректности входных параметров форм	275
Реализация компонента «Представление» в паттерне MVC	277
Описание арбитра представлений в MVC Spring	278
Паттерн «Вспомогательный компонент представления»	281
Паттерн «Составное представление» и использование арбитра представлений фреймворка Apache Tiles	283
Рекомендуемые практики проектирования веб-приложений	285
Резюме	286
Глава 11. Реализация реактивных паттернов проектирования	288
Изменение требований к приложениям с течением времени	288
Паттерн «Реактивность»	290
Отличительные признаки паттерна «Реактивность»	290
Блокирующие вызовы	296
Неблокирующие вызовы	296
Контроль обратного потока данных	297
Реализация реактивности с помощью фреймворка Spring 5.0	298
Реактивный веб-модуль Spring	299
Реализация реактивного веб-приложения на стороне сервера	300
Модель программирования на основе аннотаций	301
Функциональная модель программирования	303
Реализация реактивного приложения на стороне клиента	308
Преобразование типов тела запроса и ответа	310
Резюме	311
Глава 12. Реализация конкурентных паттернов	312
Паттерн «Активный объект»	313
Паттерн проектирования «Монитор»	314
Паттерны «Полусинхронность» и «Полуасинхронность»	315
Паттерн «Ведущий/ведомые»	316
Паттерн «Реактор»	317
Паттерн «Локальная память потока выполнения»	319
Резюме	320

Посвящается моим родителям, жене и сыну Арнаву.

Отдельное посвящение — моему покойному деду Аржуну Сингху.

Об авторе

Динеш Раджпут — главный редактор сайта Dineshonjava, технического блога, посвященного технологиям Java и Spring. На сайте размещены статьи на тему Java-технологий. Динеш — блогер, автор книг, с 2008 года энтузиаст Spring, сертифицированный специалист компании Pivotal (Pivotal Certified Spring Professional). Обладает более чем десятилетним опытом проектирования и разработки с использованием Java и Spring. Специализируется на работе с последней версией Spring Framework, Spring Boot, Spring Security, на создании REST API, архитектуре микросервисов, реактивном программировании, аспектно-ориентированном программировании с применением Spring, паттернах проектирования, Struts, Hibernate, веб-сервисах, Spring Batch, Cassandra, MongoDB, архитектуре веб-приложений.

В настоящее время Динеш работает менеджером по технологиям в одной из компаний, лидирующих в области создания программного обеспечения (ПО). Был разработчиком и руководителем команды в Bennett, Coleman & Co. Ltd, а до этого — ведущим разработчиком в Paytm. Динеш с восторгом относится к новейшим технологиям Java и любит писать о них в технических блогах. Является активным участником Java- и Spring-сообществ на различных форумах. Динеш — один из лучших специалистов по Java и Spring.

При написании данной книги я общался со многими людьми, помогавшими мне разобраться в неочевидных деталях реактивного программирования и паттернов «банды четырех».

В первую очередь я хотел бы поблагодарить рецензента, Раджива Кумара Мохана, технического консультанта и преподавателя. Особая благодарность Навину Джайну, который помог мне придумать реалистичные ситуации применения всех паттернов проектирования «банды четырех», приведенные в примерах.

Конечно же, благодарю и мою дорогую жену Анамику и сына Арнава, помогавшего мне отдохнуть за мобильными играми.

Наконец, эта книга обязана итоговым видом труду редакторов издательства Packt, Лоуренсу Вейгасу и Карану, которые помогли мне при ее написании, и Суприи, присоединившейся в процессе подготовки к печати и внесшей много предложений по улучшению издания.

О рецензенте

Раджив Кумар Мохан обладает большим опытом разработки ПО и корпоративного обучения. На протяжении 18 лет он работал в таких крупнейших IT-компаниях, как IBM, Pentasoft, Sapient и Deft Infosystems. Начал карьеру как программист, руководил многими проектами.

Является экспертом в предметной области Java, J2EE и родственных фреймворках, Android, UI-технологиях. Сертифицирован компанией Sun в качестве Java-программиста (SCJP, Sun Certified Java Programmer) и веб-разработчика на Java (Sun Certified Web Component Developer, SCWCD). Раджив имеет четыре высших образования: в области информатики (Computer Science), прикладной информатики (Computer Applications), органической химии и делового администрирования (MBA). Является консультантом по подбору персонала и экспертом по обучению в HCL, Amdocs, Steria, TCS, Wipro, Oracle University, IBM, CSC, Genpact, Sapient Infosys и Capgemini.

Раджив — основатель фирмы SNS Infotech, расположенной в городе Большая Нойда. Кроме того, он работал в Национальном институте технологий моды (National Institute Of Fashion Technology, NIFT).

Хотел бы поблагодарить Бога за возможность рецензировать эту книгу, а моих детей Сану и Саину и жену Нилам — за их помощь и поддержку, позволившие мне закончить работу в срок.

Предисловие

Эта книга предназначена для всех Java-разработчиков, желающих изучить Spring в целях применения в корпоративном ПО. В частности, она позволит улучшить понимание паттернов проектирования, используемых в фреймворке Spring, и того, как с помощью Spring решать часто встречающиеся в корпоративных приложениях задачи проектирования. Несомненно, специалисты оценят приведенные в книге примеры.

Предполагается, что читатель обладает базовыми знаниями Core Java, JSP, Servlet и XML.

Фреймворк Spring 5 недавно выпущен компанией Pivotal и поддерживает парадигму реактивного программирования. По сравнению с предыдущей версией содержит много новых возможностей — все они будут обсуждаться в этой книге, что позволит лучше понять работу фреймворка.

Spring сейчас приобрел особое значение потому, что все компании стали применять его как основной фреймворк для создания корпоративных приложений. Приступать к работе со Spring можно, не прибегая к внешним корпоративным серверам.

Цель этой книги — обсуждение всех паттернов проектирования, используемых фреймворком Spring, и их реализации в Spring. Автор также описывает наилучшие практики, которые желательно применять при проектировании и разработке приложений.

Книга содержит 12 глав, охватывающих темы от азов до описания сложных методов проектирования, таких как реактивное программирование.

Данная книга делится на три части. Первая освещает основы паттернов проектирования и фреймворка Spring. Вторая уходит от клиентской стороны к внутренней реализации приложения и показывает, как Spring применяется там. Третья часть продолжает вторую, демонстрируя, как с помощью фреймворка создавать веб-приложения, и вводя концепцию реактивного программирования. Здесь же показано, как организовывать распараллеливание в корпоративных приложениях.

Темы, рассматриваемые в книге

Глава 1 «Знакомство с Spring Framework 5.0 и паттернами проектирования» дает обзор фреймворка Spring 5 и всех его новых возможностей, включая простейшие примеры внедрения зависимостей и аспектно-ориентированного программирования. Кроме того, освещает модули Spring.

Глава 2 «Обзор паттернов проектирования GoF: базовые паттерны проектирования» представляет базовые паттерны, описанные «бандой четырех», включая некоторые наилучшие практики проектирования приложений. Вдобавок содержит обзор решения стандартных задач с использованием паттернов проектирования.

Глава 3 «Соображения по поводу структурных и поведенческих паттернов» дает информацию о структурных паттернах и паттернах поведения, описанных «бандой четырех», включая некоторые наилучшие практики проектирования приложений. Кроме того, представляет решения стандартных задач с помощью паттернов проектирования.

Глава 4 «Связывание компонентов с помощью паттерна внедрения зависимостей» посвящена рассмотрению паттерна внедрения зависимостей и подробному рассказу о конфигурировании Spring в приложении. Показаны различные варианты конфигурирования приложения, а именно через применение XML, аннотаций, Java и смешанного подхода.

Глава 5 «Жизненный цикл компонентов и используемые паттерны» дает обзор жизненного цикла компонентов Spring, управляемого контейнером Spring, в том числе рассказывает о контейнерах Spring и IoC. Кроме того, описывает обработчики обратного вызова и постпроцессоры в жизненном цикле компонентов Spring.

Глава 6 «Аспектнo-ориентированное программирование на Spring с помощью паттернов “Заместитель” и “Декоратор”» рассказывает об использовании Spring AOP для отделения сквозных задач от обслуживаемых ими объектов. Она также выступает в роли подготовительной перед последующими главами, где Spring AOP будет применяться для предоставления декларативных услуг, таких как транзакции, безопасность и кэширование.

Глава 7 «Доступ к базе данных с помощью фреймворка Spring и JDBC-реализаций паттерна “Шаблонный метод”» рассматривает доступ к данным с использованием Spring и JDBC. Из нее вы узнаете, как, задействуя абстракции операций с JDBC в Spring и шаблонов JDBC, выполнять запросы к реляционным базам данных (РБД) более простым способом, чем через родной JDBC.

Глава 8 «Доступ к базе данных с помощью паттернов ORM и транзакций» показывает, как интегрировать Spring с фреймворками объектно-реляционного отображения (ORM), такими как Hibernate и другие реализации Java Persistence API (JPA), с помощью инструментов управления транзакциями. Кроме того, глава рассказывает о чудесах Spring Data JPA в генерации динамических запросов.

Глава 9 «Улучшение производительности приложения с помощью паттернов кэширования» демонстрирует, как повысить производительность приложения, вовсе избегая использования баз данных, если есть доступ к необходимой информации. Таким образом, покажет, как Spring поддерживает кэширование данных.

Глава 10 «Реализация паттерна MVC в веб-приложениях с помощью фреймворка Spring» дает краткий обзор разработки веб-приложений с помощью Spring

MVC. Вы познакомитесь с паттернами MVC (Model — View — Controller (Модель — Представление — Контроллер)), «Единая точка входа» (Front Controller), «Сервлет-диспетчер» (Dispatcher Servlet) и основами Spring MVC, веб-фреймворка, базирующегося на принципах Spring. Вы узнаете, как писать контроллеры для обработки веб-запросов, и увидите, как связывать параметры запросов и нагрузку с прикладными объектами, одновременно обеспечивая проверку данных и обработку ошибок. Эта глава также знакомит с представлениями и арбитрами представлений в Spring MVC.

Глава 11 «Реализация реактивных паттернов проектирования» посвящена модели реактивного программирования, то есть программирования с асинхронными потоками данных. Вы увидите, как реактивная модель реализована в веб-модуле Spring.

Глава 12 «Реализация конкурентных паттернов» подробно рассматривает параллельную обработку нескольких соединений на веб-сервере. Как и принято в используемой модели архитектуры, обработка запросов отделена от логики приложения.

Что нужно для чтения книги

Этот текст можно читать и без компьютера или ноутбука, не используя ничего, кроме собственно книги. Однако для выполнения приведенных примеров понадобится установить Java 8, что можно сделать, перейдя по ссылке <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>. Для примеров потребуется и ваша любимая среда разработки. Я использовал Spring Tool Suite (STS), скачать ее последнюю версию можно по адресу <https://spring.io/tools/sts/all>, выбрав свою операционную систему. Java 8 и STS работают на различных платформах: Windows, macOS и Linux.

Принятые обозначения

В этой книге разные типы информации выделены разными шрифтами. Ниже приведены примеры и объяснения.

Участки кода внутри текста, названия таблиц баз данных, имена папок и файлов, расширения файлов, пути, веб-адреса и пользовательский ввод выглядят так: «В нашем коде имеется класс `TransferServiceImpl`, конструктор которого принимает два аргумента».

Блок кода выглядит так:

```
public class JdbcTransferRepository implements TransferRepository{
    JdbcTemplate jdbcTemplate;
    public setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
    // ...
}
```


Новые термины и ключевые слова выделяются курсивом.



Так обозначаются предупреждения и важные замечания.



Таким образом обозначаются советы.

Скачивание кода примеров

Примеры кода для выполнения упражнений из этой книги доступны для скачивания по адресу <https://github.com/PacktPublishing/Spring5-Design-Patterns>.

Скачать файлы можно, выполнив следующие шаги.

1. Перейдите по адресу github.com/PacktPublishing/Spring5-Design-Patterns.
2. Нажмите кнопку Clone or Download (Клонировать или скачать).
3. На открывшейся панели щелкните на ссылке Download Zip (Скачать Zip).

После того как файлы загружены, распакуйте их с помощью последней версии:

- ❑ WinRAR/7-Zip для Windows;
- ❑ Zipeg/iZip/UnRarX для Mac;
- ❑ 7-Zip/PeaZip для Linux.

Об ошибках

Хотя мы прикладываем все возможные усилия, чтобы обеспечить достоверность содержания, ошибки случаются. Если вы нашли ошибку в одной из наших книг (в тексте или в коде), то мы будем благодарны вам за сообщение об этом. Так вы избавите от неудобств других читателей и поможете нам улучшить следующие издания книги. Обнаружив ошибку, сообщите о ней на <https://www.packtpub.com/books/info/packt/errata-submission-form-0>, выбрав книгу, перейдя по ссылке Errata Submission Form и введя описание ошибки. Когда ваше сообщение проверят, ошибка будет указана на сайте и добавлена к списку ошибок в разделе **Errata**, относящемуся к этой книге.

Посмотреть список уже найденных ошибок можно на <https://www.packtpub.com/books/content/support>, введя в строке поиска название книги и обратившись к разделу Errata.

1

Знакомство с Spring Framework 5.0 и паттернами проектирования

Эта глава даст вам представление о фреймворке Spring, его модулях и об использовании паттернов проектирования, обусловивших успех Spring. Здесь будут описаны все главные модули фреймворка. Мы начнем с его основ, рассмотрим новые возможности, появившиеся в Spring 5.0, а также разберемся в паттернах проектирования главных модулей фреймворка.

Прочитав эту главу, вы поймете, как Spring работает и решает распространенные проблемы проектирования корпоративных приложений с помощью паттернов проектирования. Вы узнаете, как улучшить слабую связанность между компонентами приложения и упростить создание приложений, используя Spring и паттерны проектирования.

В этой главе:

- ❑ знакомство с фреймворком Spring;
- ❑ упрощение разработки приложений благодаря применению Spring и паттернов;
 - использование широчайших возможностей паттерна POJO;
 - внедрение зависимостей;
 - применение аспектов в сквозных задачах;
 - использование паттерна «Шаблонный метод» для избавления от стереотипного кода;
- ❑ создание контейнера Spring для компонентов с помощью паттерна «Фабрика»;
 - разработка контейнера с контекстом приложения;
 - жизнь компонента в контейнере;
- ❑ модули Spring;
- ❑ новые возможности Spring Framework 5.0.

Знакомство с фреймворком Spring

В первые годы существования Java было разработано множество тяжеловесных технологий для создания корпоративных приложений. Однако поддерживать последние было нелегко ввиду их тесной связи с конкретным фреймворком. Пару лет

назад все Java-технологии, кроме Spring, были достаточно тяжеловесны, как EJB. В то время Spring предлагался как альтернативная технология, специально созданная для EJB, так как Spring предоставлял очень простую, гораздо более гибкую и легкую, по сравнению с другими существовавшими Java-технологиями, модель программирования.

Широкие возможности Spring достигаются благодаря использованию множества паттернов проектирования, но главной стала *POJO-модель программирования* (Plain Old Java Object, «старый добрый объект Java»). Она обеспечила простоту фреймворка Spring и, кроме того, предоставила функционал таких концепций, как паттерн *внедрения зависимостей* (DI) и *аспектно-ориентированное программирование* (AOP), благодаря использованию паттернов «Заместитель» и «Декоратор».

Spring Framework — это фреймворк с открытым исходным кодом и основанная на Java платформа, предоставляющая полную поддержку инфраструктуры для создания корпоративных Java-приложений. Таким образом, разработчики не должны думать об инфраструктуре приложения и могут сконцентрироваться на его бизнес-логике, а не конфигурации. Все файлы инфраструктуры, конфигурации и метаконфигурации, использующие Java или XML, обрабатываются фреймворком Spring. Так, при создании приложения с помощью модели программирования POJO он обеспечивает большую гибкость, чем при использовании неагрессивной модели программирования.

IoC-контейнер Spring (Inversion of Control, инверсия управления) — ядро всего фреймворка. Он позволяет объединять в единое целое разные части приложения, обеспечивая согласованность архитектуры. Компоненты Spring MVC могут использоваться для формирования очень гибкого веб-уровня. IoC-контейнер облегчает разработку на бизнес-уровне с помощью POJO.

Spring упрощает создание приложений и часто устраняет зависимость от других API. Рассмотрим несколько ситуаций, в которых вы как разработчик можете выиграть от использования платформы Spring.

- ❑ Все классы в приложении являются простыми POJO-классами — Spring неагрессивен. В большинстве ситуаций он не требует от вас наследоваться от классов фреймворка или реализовывать его интерфейсы.
- ❑ Приложения, использующие Spring, не требуют наличия сервера приложений Java EE, но могут быть развернуты на нем.
- ❑ В транзакции с обращением к базе данных методы могут выполняться с помощью управления транзакциями Spring, без использования сторонних транзакционных API.
- ❑ Благодаря Spring можно использовать методы Java как обработчики запросов или удаленные методы, такие как метод `service()` из API сервлетов, без взаимодействия с API контейнера сервлетов.
- ❑ Spring позволяет использовать локальные методы `java` как обработчики сообщений без применения в приложении API сервиса сообщений *Java Message Service (JMS)*.

- ❑ Spring также позволяет использовать локальные методы `java` как операции управления без применения в приложении API управленческих расширений *Java Management Extensions (JMX)*.
- ❑ Spring выступает в роли контейнера для объектов приложения. Объекты не должны сами находить и *устанавливать* связи между собой.
- ❑ Spring создает компоненты и внедряет зависимости между объектами приложения, таким образом управляя жизненным циклом компонентов.

Упрощение разработки приложений благодаря применению Spring и паттернов

При разработке коммерческого приложения на традиционной платформе Java, когда речь заходит об организации повторного использования «блоков»-компонентов, возникает большое количество ограничений. Нельзя не учитывать, что повторное применение компонентов основной и общей функциональности — общепринятый стандарт проектирования. Для решения задачи повторного использования кода можно задействовать различные паттерны проектирования, такие как «Фабрика» (Factory), «Абстрактная фабрика» (Abstract Factory), «Строитель» (Builder), «Декоратор» (Decorator), «Локатор служб» (Service Locator), и составлять из базовых «кирпичиков» согласованное целое, класс или объект, таким образом обеспечивая повторное использование составляющих частей. Эти паттерны, решающие общие и рекурсивные задачи, реализованы внутри фреймворка Spring. Таким образом, он предоставляет инфраструктуру с четко определенным способом применения.

Написание корпоративных приложений — непростое дело, но Spring, специально созданный для борьбы с этими сложностями, упрощает задачу разработчиков. Применение Spring не ограничивается стороной сервера — он упрощает также сборку проекта, тестирование и слабое сцепление. Spring использует концепцию POJO, то есть компонент Spring может быть Java-объектом любого типа. Компонент — самодостаточный участок кода, который в идеале можно использовать повторно в разных приложениях.

Поскольку эта книга в основном посвящена всем *паттернам проектирования*, применяемым в Spring для упрощения разработки на Java, я объясняю или хотя бы описываю реализацию и задачи паттернов и наилучшие способы использования инфраструктуры для создания приложений. Фреймворк Spring использует следующие методы для упрощения разработки и тестирования:

- ❑ применяет *паттерн POJO* для легкой и минимально агрессивной разработки корпоративных приложений;
- ❑ задействует *паттерн внедрения зависимостей (DI)* для слабого сцепления и делает систему интерфейс-ориентированной;

- ❑ прибегает к *паттернам* «Декоратор» и «Заместитель» для декларативного программирования с помощью аспектов и общепринятых соглашений;
- ❑ применяет *паттерн* «Шаблонный метод» для устранения стереотипного кода с помощью аспектов и шаблонов.

В текущей главе я объясню все эти понятия и приведу конкретные примеры того, как Spring упрощает разработку на Java. Сначала разберемся, как Spring остается минимально агрессивным, поощряя проектирование, ориентированное на использование паттерна POJO.

Использование широчайших возможностей паттерна POJO

Существует множество фреймворков для разработки на Java, ограничивающих вас, заставляя расширять некоторые имеющиеся классы или реализовывать определенные интерфейсы. В частности, такого подхода придерживались Struts, Tapestry и ранние версии EJB. Эти фреймворки основаны на агрессивной модели программирования, которая усложняет поиск ошибок в системе, а порой делает код просто нечитаемым. Однако при работе с фреймворком Spring нет необходимости расширять готовые классы (реализовывать готовые интерфейсы) — реализация основана на паттерне POJO и следует неагрессивной модели программирования. Таким образом, искать ошибки в системе легче, а код остается понятным.

Spring позволяет программировать, задействуя простые не-Spring классы, и не заставляет применять специфичные для этого фреймворка классы и интерфейсы, так что все классы в Spring-приложении будут POJO. Таким образом, файлы могут быть скомпилированы и исполнены независимо от установленных библиотек Spring. Невозможно даже понять, что эти классы используются этим фреймворком. В худшем случае — при конфигурировании, основанном на Java, — вам придется прибегнуть к аннотациям Spring.

Поясню сказанное на следующем примере:

```
package com.packt.chapter1.spring;
public class HelloWorld {
    public String hello() {
        return "Hello World";
    }
}
```

Это просто POJO-класс без каких-либо указаний или особенностей реализации, относящихся к фреймворку и делающих его компонентом Spring. Он будет одинаково работать в приложениях, использующих и не использующих Spring. Этим и прекрасна неагрессивная модель программирования, применяемая Spring. Spring расширяет возможности POJO другим способом, организуя их взаимодействие с другими POJO с помощью паттерна внедрения зависимостей. Рассмотрим, как внедрение зависимостей позволяет разделить компоненты.

Внедрение зависимостей между POJO

Термин «внедрение зависимостей» не нов — он использовался еще в PicoContainer. Внедрение зависимостей — это паттерн проектирования, обеспечивающий слабую связанность между компонентами Spring, то есть между различными взаимодействующими POJO. Применение внедрения зависимостей к вашим сложным задачам позволит упростить код для написания, понимания и тестирования.

В вашем приложении множество объектов совместно обеспечивают некую требуемую вам функциональность. Подобная совместная работа объектов известна как внедрение зависимостей. Такое внедрение между работающими компонентами помогает при модульном тестировании каждого компонента приложения без сильной связанности.

В реальном приложении конечный пользователь хочет видеть результат. Для получения результата несколько объектов приложения работают совместно и иногда связаны между собой. Поэтому при написании классов в сложных приложениях подумайте о повторном использовании этих классов и сделайте их максимально независимыми. Это наилучшая практика в программировании, позволяющая модульно тестировать такие классы по отдельности.

Как внедрение зависимостей работает и как упрощает разработку и тестирование

Рассмотрим реализацию паттерна внедрения зависимостей в вашем приложении. Он способствует понятности, слабой связанности и тестируемости всего приложения. Предположим, у нас имеется простое приложение (но посложнее, чем пример *Hello World*, который вы писали на первом курсе). Все классы работают совместно для ожидаемого от них решения некой бизнес-задачи. Это значит, что у каждого класса в приложении есть своя доля ответственности за задачу, которую он делит с сотрудничающими объектами (зависимостями). Посмотрите на рис. 1.1. Такая зависимость между объектами может усложнять приложение и создавать сильную связанность.

Типичное приложение состоит из нескольких частей, работающих совместно для выполнения пользовательской задачи. Например, рассмотрим класс `TransferService`, показанный ниже.

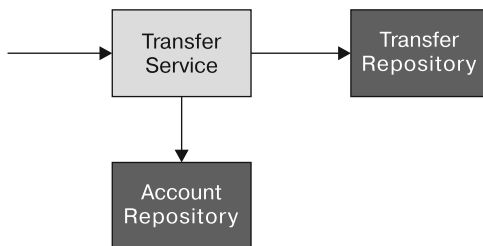


Рис. 1.1. Компонент `TransferService` традиционно зависит от двух других компонентов: `TransferRepository` и `AccountRepository`

Класс `TransferService` использует прямое инстанцирование.

```
package com.packt.chapter1.bankapp.transfer;
public class TransferService {
    private AccountRepository accountRepository;
    public TransferService () {
        this.accountRepository = new AccountRepository();
    }
    public void transferMoney(Account a, Account b) {
        accountRepository.transfer(a, b);
    }
}
```

Объекту `TransferService` нужен объект `AccountRepository`, чтобы перевести деньги со счета `a` на счет `b`. Поэтому он создает экземпляр `AccountRepository` и использует его. Но прямое инстанцирование усиливает связанность и приводит к тому, что создающий объекты код разбросан по всему приложению. При этом становится труднее поддерживать код и писать модульные тесты для `TransferService`, так как в этом случае при необходимости протестировать метод `transferMoney()` класса `TransferService`, используя для модульного тестирования `assert`, метод `transfer()` класса `AccountRepository` вряд ли вызовется тестом. Но разработчик не знает о зависимости `AccountRepository` от класса `TransferService`; по крайней мере он не может протестировать метод `transferMoney()` класса `TransferService` с помощью модульного тестирования.

В корпоративных приложениях связанность очень опасна и приводит к ситуации, когда невозможно развивать приложение в будущем, поскольку любые дальнейшие изменения приводят к появлению большого количества ошибок, а при их исправлении возникают новые ошибки. Сильно связанные компоненты — одна из причин серьезных проблем в подобных приложениях. Сильно связанный код, когда в нем нет необходимости, делает приложение неподдерживаемым, и с течением времени код перестанет использоваться повторно, поскольку не будет понятен другим разработчикам. Однако порой некая связанность необходима в корпоративном приложении, так как полностью не связанные компоненты в реальных задачах применяться не могут. Каждый компонент в приложении несет некую ответственность за свою роль и бизнес-требования, вплоть до того, что все компоненты должны знать об ответственности других приложений. То есть связанность иногда необходима, но мы должны быть очень осторожны при управлении ею, когда она нужна.

Использование вспомогательного паттерна «Фабрика» для зависимых компонентов

Попробуем другой способ обращения с зависимыми объектами, использующий паттерн «Фабрика». Он основан на паттерне «Фабрика» «банды четырех», создающем с помощью фабричного метода экземпляры класса. Таким образом, этот метод централизует применение оператора `new`. Он создает экземпляры класса, основываясь на информации, предоставленной клиентским кодом. Данный паттерн широко используется в стратегии внедрения зависимостей.

Класс `TransferService` использует вспомогательный паттерн «Фабрика»:

```
package com.packt.chapter1.bankapp.transfer;
public class TransferService {
    private AccountRepository accountRepository;
    public TransferService() {
        this.accountRepository =
            AccountRepositoryFactory.getInstance("jdbc");
    }
    public void transferMoney(Account a, Account b) {
        accountRepository.transfer(a, b);
    }
}
```

В данном коде мы используем паттерн «Фабрика» для создания объекта `AccountRepository`. При создании программного обеспечения одной из лучших практик при проектировании и разработке является *program-to-interface (P2I)*. Согласно этой практике конкретные классы должны реализовывать интерфейс, применяемый в клиентском коде для вызывающей функции, а не конкретный класс. Используя P2I, можно улучшать имеющийся код. Мы можем легко заменить его другой реализацией интерфейса, не затрагивая клиентский код. Таким образом, *program-to-interface* обеспечивает слабую связанность. Иначе говоря, отсутствует зависимость от конкретной реализации, приводящая к зависимости на низком уровне. Рассмотрим следующий код. В нем `AccountRepository` — интерфейс, а не класс:

```
public interface AccountRepository{
    void transfer();
    // другие методы
}
```

Мы можем реализовать его в соответствии с нашими требованиями, и он зависит от инфраструктуры клиента. Предположим, что `AccountRepository` понадобился нам в фазе разработки с использованием JDBC API. Мы можем предоставить конкретную реализацию `JdbcAccountRepository` интерфейса `AccountRepository`, как показано ниже:

```
public class JdbcAccountRepository implements AccountRepository{
    // ...реализация методов, определенных в AccountRepository
    // ...реализация остальных методов
}
```

В данном паттерне объекты создаются фабричными классами, чтобы код было легче поддерживать. Это предотвращает разбросанный среди других бизнес-компонентов код, создающий объекты. Вспомогательный класс фабрики также позволяет сделать создание объектов конфигурируемым. Эта техника решает проблему сильной связанности, но мы по-прежнему добавляем фабричные классы к бизнес-компоненту для получения сотрудничающих компонентов. Так что рассмотрим в следующем подразделе паттерн внедрения зависимостей и то, как он решает указанную проблему.

Использование паттерна внедрения зависимостей для зависимых компонентов

Согласно паттерну внедрения зависимостей зависимости присваиваются объектам при их создании фабрикой или третьей стороной. Эта фабрика координирует объекты в системе так, что от зависимого объекта не ожидается создание своих зависимостей. Таким образом, нам надо сосредоточиться на определении зависимостей вместо разрешения зависимостей сотрудничающих объектов в корпоративном приложении. Посмотрим на рис. 1.2. Обратите внимание, что зависимости внедряются в объекты, которым они необходимы.

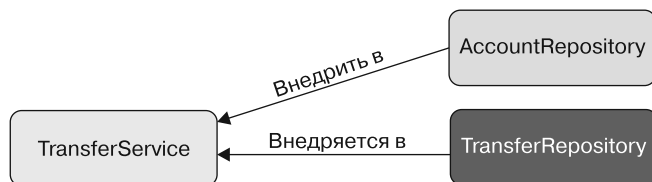


Рис. 1.2. Внедрение зависимостей между различными сотрудничающими компонентами приложения

Для иллюстрации этого утверждения рассмотрим класс `TransferService` — у него есть зависимости от `AccountRepository` и `TransferRepository`. Здесь `TransferService` способен переводить деньги при использовании любой реализации интерфейса `TransferRepository`, то есть мы можем применить `JdbcTransferRepository` или `JpaTransferRepository` в зависимости от того, что получаем на входе из окружающей среды.

Класс `TransferServiceImpl` достаточно гибок, чтобы принимать любой данный ему `TransferRepository`:

```
package com.packt.chapter1.bankapp;
public class TransferServiceImpl implements TransferService {
    private TransferRepository transferRepository;
    private AccountRepository accountRepository;
    public TransferServiceImpl(TransferRepository transferRepository,
        AccountRepository accountRepository) {
        this.transferRepository =
            transferRepository; // TransferRepository is injected
        this.accountRepository = accountRepository;
        // AccountRepository внедрен
    }
    public void transferMoney(Long a, Long b, Amount amount) {
        Account accountA = accountRepository.findById(a);
        Account accountB = accountRepository.findById(b);
        transferRepository.transfer(accountA, accountB, amount);
    }
}
```

Можно видеть, что `TransferServiceImpl` не создает собственные реализации репозитория. Вместо этого реализация предоставляется как аргумент конструктора. Такой тип внедрения зависимостей известен как *внедрение через конструкторы*. Здесь мы передали тип интерфейса репозитория как аргумент конструктора при создании. Теперь `TransferServiceImpl` может использовать свою реализацию репозитория, например JDBC, JPA или mock-объекты. Смысл в том, что `TransferServiceImpl` не связан ни с какой конкретной реализацией репозитория. Не имеет значения, какой тип репозитория используется для перевода суммы с одного счета на другой, до тех пор пока он реализует интерфейс репозитория. Если вы используете паттерн внедрения зависимостей фреймворка Spring, то слабая связанность — одно из ключевых преимуществ. Паттерн внедрения зависимостей всегда поддерживает P2I, в связи с чем каждый объект знает свои зависимости как интерфейсы, а не как реализации, поэтому зависимость можно легко заменить на другую реализацию того же интерфейса вместо того, чтобы заменять класс.

Spring поддерживает такую сборку системы из частей, что:

- ❑ задачей частей не является найти друг друга;
- ❑ любую часть легко заменить.

Способ сборки приложения с помощью создания связей между частями или компонентами называется *привязыванием*. В Spring существует много способов привязывать сотрудничающие компоненты друг к другу для создания системы приложения. Например, мы можем использовать как XML-файл конфигурации, так и Java-файл конфигурации.

Теперь посмотрим, как внедрить зависимости `TransferRepository` и `AccountRepository` в класс `TransferService` с использованием 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="transferService"
    class="com.packt.chapter1.bankapp.service.TransferServiceImpl">
    <constructor-arg ref="accountRepository"/>
    <constructor-arg ref="transferRepository"/>
  </bean>
  <bean id="accountRepository" class="com.
    packt.chapter1.bankapp.repository.JdbcAccountRepository"/>
  <bean id="transferRepository" class="com.
    packt.chapter1.bankapp.repository.JdbcTransferRepository"/>
</beans>
```

Здесь `TransferServiceImpl`, `JdbcAccountRepository` и `JdbcTransferRepository` объявлены как компоненты Spring. Компонент `TransferServiceImpl` создается с по-

мощью передачи ссылок на компоненты `AccountRepository` и `TransferRepository` как аргументы конструктора. Полезно знать, что Spring позволяет задать ту же самую конфигурацию, задействуя Java.

Как альтернативу XML Spring предлагает конфигурирование с использованием Java:

```
package com.packt.chapter1.bankapp.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.packt.chapter1.bankapp.repository.AccountRepository;
import com.packt.chapter1.bankapp.repository.TransferRepository;
import com.packt.chapter1.bankapp.repository.jdbc.JdbcAccountRepository;
import com.packt.chapter1.bankapp.repository.jdbc.JdbcTransferRepository;
import com.packt.chapter1.bankapp.service.TransferService;
import com.packt.chapter1.bankapp.service.TransferServiceImpl;

@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService(){
        return new TransferServiceImpl(accountRepository(), transferRepository());
    }
    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository();
    }
    @Bean
    public TransferRepository transferRepository() {
        return new JdbcTransferRepository();
    }
}
```

Преимущества паттерна внедрения зависимостей не зависят от того, используете вы конфигурирование с помощью XML или Java:

- ❑ внедрение зависимостей способствует слабой связанности. Можно убрать жесткие зависимости с использованием P2I и задать зависимости извне приложения, применяя паттерн «Фабрика» и его встроенную заменяемую и подключаемую извне реализацию;
- ❑ паттерн внедрения зависимостей в объектно-ориентированном программировании способствует композиционному проектированию, а не основанному на наследовании.

Хотя `TransferService` зависит от `AccountRepository` и `TransferRepository`, ему неважно, какой тип (JDBC или JPA) реализаций `AccountRepository` и `TransferRepository` используется в приложении. Только Spring благодаря своей конфигурации (основанной на XML или Java) знает, как соединяются все компоненты,

как они создаются вместе с их требуемыми зависимостями, используя паттерн внедрения зависимостей. Внедрение зависимостей позволяет менять зависимости, не меняя зависимые классы, то есть можно использовать как JDBC-, так и JPA-реализацию, не прибегая к изменению реализации `AccountService`.

В приложении Spring реализация контекста приложения (Spring предлагает `AnnotationConfigApplicationContext` для реализаций, основанных на Java, и `ClassPathXmlApplicationContext` для основанных на XML) загружает определения компонентов и привязывает их в один контейнер Spring. Контекст приложения в Spring создает и привязывает компоненты Spring при запуске приложения. Рассмотрим реализацию контекста приложения в Spring с конфигурированием, основанным на Java. Она загружает конфигурационные файлы Spring (`AppConfig.java` для Java и `Spring.xml` для XML), расположенные в путях приложения. В следующем коде метод `main()` класса `TransferMain` использует класс `AnnotationConfigApplicationContext`, чтобы загрузить класс конфигурации `AppConfig.java` и получить объект класса `AccountService`.

Spring предлагает конфигурирование с использованием Java как альтернативу XML:

```
package com.packt.chapter1.bankapp;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.
    AnnotationConfigApplicationContext;

import com.packt.chapter1.bankapp.config.AppConfig;
import com.packt.chapter1.bankapp.model.Amount;
import com.packt.chapter1.bankapp.service.TransferService;

public class TransferMain {

    public static void main(String[] args) {
        // Загружаем контекст Spring
        ConfigurableApplicationContext applicationContext =
            new AnnotationConfigApplicationContext(AppConfig.class);
        // Получаем бин TransferService
        TransferService transferService =
            applicationContext.getBean(TransferService.class);
        // Используем метод transfer
        transferService.transferAmmount(1001, 2001,
            new Amount(2000.0));
        applicationContext.close();
    }
}
```

Это было краткое знакомство с паттерном внедрения зависимостей. Вы узнаете о нем намного больше в следующих главах книги. Теперь рассмотрим другой способ упростить разработку на Java: с использованием декларативной модели программирования в Spring с помощью аспектов и паттерна «Заместитель».

Применение объектов для сквозных задач

В приложении Spring паттерн внедрения зависимостей обеспечивает слабую связанность между взаимодействующими компонентами, но *аспектно-ориентированное программирование* в Spring (Spring AOP) позволяет выделять повторяющуюся в приложении функциональность. Таким образом, можно сказать, что Spring AOP способствует слабой связанности и позволяет решать сквозные задачи, перечисленные далее, более изящным способом. Оно позволяет этим сервисам прозрачно применяться через объявление. С помощью Spring AOP можно создавать пользовательские аспекты и декларировать их конфигурацию.

Общая функциональность, необходимая во многих местах в приложении, это:

- ☐ журналирование;
- ☐ управление транзакциями;
- ☐ безопасность;
- ☐ кэширование;
- ☐ обработка ошибок;
- ☐ отслеживание производительности;
- ☐ пользовательские бизнес-правила.

Перечисленные здесь компоненты не входят в ядро приложения, но имеют дополнительные функции, обычно называемые сквозными задачами, поскольку те относятся ко многим компонентам, но не касаются их основных обязанностей. Если объединить эти компоненты с функциональностью ядра, таким образом реализуя сквозные задачи без разбиения на модули, то возникнут две серьезные проблемы.

- ☐ *Запутанность кода*: связанность задач означает, что сквозные задачи, такие как задачи безопасности, транзакций и журналирования, связаны с кодом бизнес-объектов приложения.
- ☐ *Разбросанность кода*: данная проблема состоит в том, что код для одной задачи разбросан по разным модулям. Таким образом, код задач безопасности, транзакций и журналирования разбрасывается по всем модулям системы. Иными словами, код для одной и той же задачи в системе повторяется.

Приведенная ниже схема (рис. 1.3) иллюстрирует эту сложность. Бизнес-объекты слишком тесно связаны со сквозными задачами. Каждый объект не только знает, что его операции должны фиксироваться в журнале, соответствовать требованиям безопасности и выполняться в рамках транзакции, но еще и отвечает за свое использование этих сервисов.

Spring AOP делает возможным выделение сквозных задач в отдельные модули, чтобы избежать запутанности и разбросанности кода. Эти модули можно явно применять к основным бизнес-компонентам приложения, не влияя на них. Аспекты гарантируют, что POJO остаются просто Java-объектами. Эти чудеса Spring AOP возможны благодаря использованию паттерна проектирования «Заместитель», который мы будем подробнее обсуждать в следующих главах книги.

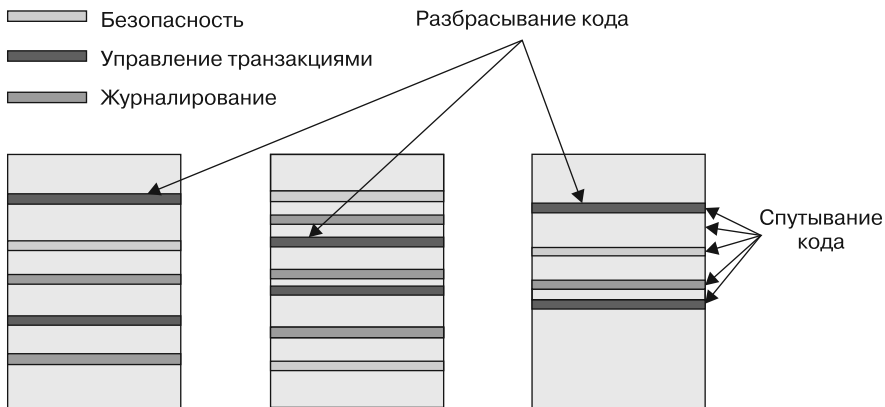


Рис. 1.3. Сквозные задачи, такие как журналирование, безопасность и транзакции, часто разбросаны по модулям, где не являются основными функциями

Как работать со Spring AOP. Это можно описать как такую последовательность действий.

1. *Реализуйте основную логику приложения.* При написании бизнес-логики приложения сосредоточьтесь на главной задаче — вам не нужно беспокоиться о дополнительной функциональности, такой как журналирование, безопасность и транзакции, и включать ее в код бизнес-логики, об этом позаботится Spring AOP.
2. *Напишите аспекты для реализации сквозных задач.* Spring содержит множество готовых аспектов, так что можно добавить дополнительную функциональность в виде независимых аспектов Spring AOP. Дополнительными обязанностями этих аспектов будут сквозные задачи, не входящие в код логики приложения.
3. *Включите аспекты в приложение.* Добавьте выполнение сквозных задач в нужные места — после написания аспектов для дополнительных обязанностей их можно декларативно применить там, где это необходимо в коде логики приложения.

Рассмотрим иллюстрацию к вопросу аспектно-ориентированного программирования в Spring (рис. 1.4).

На рис. 1.4 Spring AOP отделяет сквозные задачи, например безопасность, транзакции и журналирование, от бизнес-модулей, то есть `BankService`, `CustomerService` и `ReportingService`. Эти сквозные задачи применяются в заранее определенных точках бизнес-модулей (показанных на схеме полосками) во время работы приложения.

Предположим, вы хотите журналировать сообщения до и после вызова метода `transferAmount()` класса `TransferService`, используя службы `LoggingAspect`. Следующий код содержит класс `LoggingAspect`, который, возможно, вы будете применять.

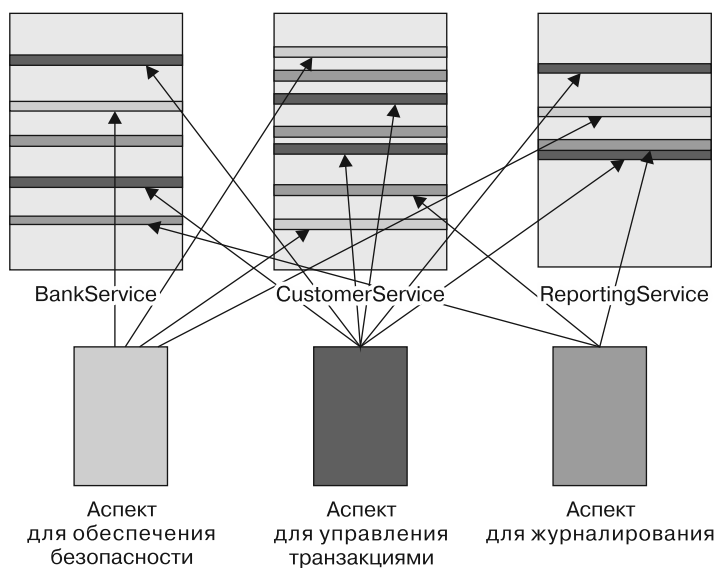


Рис. 1.4. Эволюция системы, основанной на аспектно-ориентированном программировании, — компоненты приложения остаются сосредоточенными на собственной бизнес-функциональности

Класс `LoggingAspect` вызывается для журналирования действий `TransferService`:

```
package com.packt.chapter1.bankapp.aspect;
```

```
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
```

```
@Aspect
```

```
public class LoggingAspect {
    @Before("execution(* *.transferAmount(..)")
    public void logBeforeTransfer(){
        System.out.println("####Вызов метода LoggingAspect.logBeforeTransfer()
        перед переводом суммы ####");
    }
    @After("execution(* *.transferAmount(..)")
    public void logAfterTransfer(){
        System.out.println("####Вызов метода LoggingAspect.logBeforeTransfer()
        после перевода суммы####");
    }
}
```

Для превращения `LoggingAspect` в компонент-аспект нужно объявить его таковым в файле конфигурации Spring. Кроме того, понадобится добавить к классу

аннотацию `@Aspect`. Ниже приведен файл `AppConfig.java` с внесенными изменениями, объявляющими `LoggingAspect` аспектом.

В этом примере также показана возможность использования прокси в Spring AOP:

```
package com.packt.chapter1.bankapp.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

import com.packt.chapter1.bankapp.aspect.LoggingAspect;
import com.packt.chapter1.bankapp.repository.AccountRepository;
import com.packt.chapter1.bankapp.repository.TransferRepository;
import com.packt.chapter1.bankapp.repository.jdbc.JdbcAccountRepository;
import com.packt.chapter1.bankapp.repository.jdbc.JdbcTransferRepository;
import com.packt.chapter1.bankapp.service.TransferService;
import com.packt.chapter1.bankapp.service.TransferServiceImpl;

@Configuration
@EnableAspectJAutoProxy
public class AppConfig {
    @Bean
    public TransferService transferService(){
        return new TransferServiceImpl(accountRepository(),
            transferRepository());
    }
    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository();
    }
    @Bean
    public TransferRepository transferRepository() {
        return new JdbcTransferRepository();
    }
    @Bean
    public LoggingAspect loggingAspect() {
        return new LoggingAspect();
    }
}
```

Здесь мы осуществляем конфигурирование Spring AOP с помощью Java, чтобы объявить компонент `LoggingAspect` аспектом. Сначала мы объявляем `LoggingAspect` компонентом, потом аннотируем этот компонент аннотацией `@Aspect`.

Мы аннотируем метод `logBeforeTransfer()` класса `LoggingAspect` аннотацией `@Before`, так что он будет вызываться перед выполнением `transferAmount()`. Это называется *объявлением предварительной операции* (before advice). Затем мы аннотируем другой метод `LoggingAspect` с помощью `@After`, чтобы объявить: ме-

тод `logAfterTransfer()` должен вызываться после выполнения `transferAmount()`. Это называется *объявлением последующей операции* (after advice).

Аннотация `@EnableAspectJAutoProxy` используется для добавления возможностей Spring AOP в приложение. Она фактически заставляет вас применять прокси к некоторым компонентам, определенным в конфигурационном файле Spring. Мы еще поговорим про Spring AOP в главе 6. На данный момент достаточно знать, что мы попросили Spring вызывать методы `logBeforeTransfer()` и `logAfterTransfer()` класса `LoggingAspect` до и после метода `transferAmount()` класса `TransferService`. Сейчас важно сделать из этого примера два вывода.

- ❑ Класс `LoggingAspect` — все еще POJO (если игнорировать аннотацию `@Aspect` или использовать конфигурирование с помощью XML) — ничто в нем не указывает на использование в качестве аспекта.
- ❑ Важно помнить, что `LoggingAspect` можно применять к `TransferService` без явного вызова со стороны `TransferService`. Фактически `TransferService` ничего не знает о существовании `LoggingAspect`.

Перейдем к другому способу, которым Spring упрощает разработку на Java.

Применение шаблонов для устранения стереотипного кода

Если где-то в нашем корпоративном приложении мы видим код, похожий на уже написанный ранее, это стереотипный код. Нам часто приходится писать такой код снова и снова в разных частях одного и того же приложения, чтобы выполнять одинаковые задачи. К сожалению, во многих местах Java API включают стереотипный код. Типичный пример такого кода можно встретить в работе с JDBC при запросах к базе данных. Если вы когда-нибудь работали с JDBC, то наверняка писали код, делающий что-то из перечисленного ниже:

- ❑ получение соединения из пула соединений;
- ❑ создание объекта `PreparedStatement`;
- ❑ связывание параметров SQL;
- ❑ выполнение `PreparedStatement`;
- ❑ получение данных из объекта `ResultSet` и заполнение объектов-контейнеров данными;
- ❑ освобождение ресурсов базы данных.

Рассмотрим следующий пример, содержащий стереотипный код из JDBC API:

```
public Account getAccountById(long id) {
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
```

```

try {
    conn = dataSource.getConnection();
    stmt = conn.prepareStatement(
        "select id, name, amount from " +
        "account where id=?");
    stmt.setLong(1, id);
    rs = stmt.executeQuery();
    Account account = null;
    if (rs.next()) {
        account = new Account();
        account.setId(rs.getLong("id"));
        account.setName(rs.getString("name"));
        account.setAmount(rs.getString("amount"));
    }
    return account;
} catch (SQLException e) {
} finally {
    if(rs != null) {
        try {
            rs.close();
        } catch(SQLException e) {}
    }
    if(stmt != null) {
        try {
            stmt.close();
        } catch(SQLException e) {}
    }
    if(conn != null) {
        try {
            conn.close();
        } catch(SQLException e) {}
    }
}
return null;
}

```

В приведенном выше примере мы видим, что код JDBC запрашивает из базы имя счета и сумму. Для столь простой задачи нам нужно создать соединение, а затем определить и совершить запрос. Нам также надо ловить проверяемые исключения `SQLException`, хотя при брошенном исключении мы мало что можем сделать. Наконец, нам нужно «убрать за собой», закрыв соединение, освободив объект запроса и набор результатов. Это может вынудить нас также обрабатывать исключения JDBC, причем снова придется ловить `SQLException`. Стереотипный код такого рода очень мешает повторному использованию.

Spring JDBC решает проблему стереотипного кода, применяя паттерн «Шаблонный метод», и сильно упрощает жизнь, избавляя от общего кода. Это делает код доступа к данным ясным и предотвращает неприятные проблемы, такие как утечки соединений, поскольку фреймворк Spring гарантирует, что все ресурсы базы данных освобождаются корректно.

Шаблоны в Spring. Рассмотрим, как их использовать.

Основные шаги алгоритма таковы.

1. Оставить на потом конкретные детали реализации.
2. Скрыть большие куски стереотипного кода.

Spring предоставляет множество шаблонных классов:

- ❑ JdbcTemplate;
- ❑ JmsTemplate;
- ❑ RestTemplate;
- ❑ WebServiceTemplate.

В большинстве из них низкоуровневое управление ресурсами скрыто.

Рассмотрим тот же код, который мы ранее использовали с JdbcTemplate, как пример устранения стереотипного кода.

Мы используем JdbcTemplate, чтобы код в первую очередь выполнял основную задачу:

```
public Account getAccountById(long id) {
    return jdbcTemplate.queryForObject(
        "select id, name, amoount" +
        "from account where id=?",
        new RowMapper<Account>() {
            public Account mapRow(ResultSet rs,
                int rowNum) throws SQLException {
                account = new Account();
                account.setId(rs.getLong("id"));
                account.setName(rs.getString("name"));
                account.setAmount(rs.getString("amount"));
                return account;
            }
        },
        id);
}
```

Как можно видеть из приведенного примера, новая версия `getAccountById()` по сравнению со стереотипным кодом гораздо проще и метод сосредоточен на выборе счета из базы, а не создании соединения с базой, определении и выполнении запроса, обработке исключений SQL и закрытии соединения в конце. При использовании шаблона вы задаете запрос SQL и объект `RowMapper`, отображающий набор полученных данных в объект предметной области, в методе шаблона `queryForObject()`. Шаблон отвечает за выполнение всех необходимых для данной операции действий наподобие соединения с базой. Он также скрывает внутри фреймворка стереотипный код.

В этом разделе мы видели, как Spring справляется со сложностями Java-разработки, ориентируясь на работу с POJO и задействуя внедрение зависимостей, прокси с аспектами и шаблонные методы.

В следующем разделе мы увидим, как применять контейнеры Spring для создания компонентов Spring в приложении и управления ими.

Использование контейнеров Spring для управления компонентами с помощью паттерна «Фабрика»

Spring предоставляет одноименный контейнер, в котором «живут» объекты нашего приложения. Как показано на рис. 1.5, этот контейнер отвечает за создание объектов и управление ими.

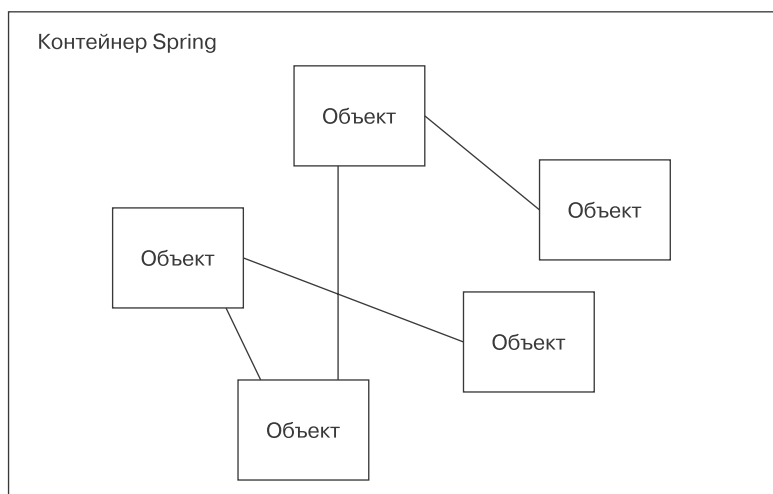


Рис. 1.5. В приложении Spring объекты «живут» в контейнере Spring

Контейнер Spring также связывает множество *объектов* в соответствии с конфигурацией. Он конфигурируется с некоторыми задаваемыми параметрами и управляет всем жизненным циклом объектов.

Существует два основных типа контейнеров Spring:

- ❑ фабрики компонентов;
- ❑ контексты приложения.

Фабрики компонентов

Во фреймворке Spring интерфейс `org.springframework.beans.factory.BeanFactory` определяет фабрику компонентов — она представляет собой контейнер обратного управления. Класс `XmlBeanFactor` реализует указанный интерфейс. Контейнер

читает метаданные конфигурации из файла XML. Он основан на паттерне фабричного метода «банды четырех» — сложными способами создает, кэширует, связывает объекты приложения и управляет ими. Фабрика компонентов — просто пул объектов, в котором объекты создаются и управляются с помощью конфигурации. Для маленьких приложений этого достаточно, но корпоративные приложения требуют большего, так что Spring предоставляет и контейнеры с дополнительными возможностями.

Из следующего подраздела вы узнаете о контекстах приложений и о том, как их создает Spring.

Контексты приложений

Во фреймворке Spring интерфейс `org.springframework.context.ApplicationContext` также определяет контейнер обратного управления Spring. Это просто оболочка для фабрики компонентов, предоставляющая некоторые дополнительные возможности контекста приложения, например поддержку аспектно-ориентированного программирования (и, как следствие, декларативных механизмов для транзакций, обеспечения безопасности и некоторых инструментов).

Создание контейнера с контекстом приложения

Spring предоставляет несколько видов контекстов приложений, используемых в качестве контейнеров компонентов. Есть много реализаций интерфейса `ApplicationContext`, включая следующие:

- ❑ `FileSystemXmlApplicationContext` — загружает определение контекста приложения из конфигурационных XML-файлов в файловой системе;
- ❑ `ClassPathXmlApplicationContext` — загружает определение контекста приложения из конфигурационных XML-файлов, указанных в библиотеке классов приложения;
- ❑ `AnnotationConfigApplicationContext` — загружает определение контекста приложения из конфигурационных классов Java, указанных в библиотеке классов приложения.

Spring предоставляет также веб-реализации интерфейса `ApplicationContext`, например:

- ❑ `XmlWebApplicationContext` — загружает определение контекста приложения из конфигурационных XML-файлов, содержащихся внутри веб-приложения;
- ❑ `AnnotationConfigWebApplicationContext` — загружает определение контекста приложения из конфигурационных классов Java.

Мы можем использовать любую из этих реализаций для загрузки компонентов в фабрику компонентов в зависимости от местоположения конфигурационных файлов. Например, если вы хотите загрузить конфигурационный файл `spring.xml`

из определенного места файловой системы, то Spring предоставляет для этого класс `FileSystemXmlApplicationContext`, который ищет указанный конфигурационный файл в заданном месте файловой системы:

```
ApplicationContext context = new  
    FileSystemXmlApplicationContext("D:/spring.xml");
```

Аналогично можно загрузить конфигурационный файл `spring.xml` из библиотеки классов приложения, используя класс `ClassPathXmlApplicationContext`. Класс будет искать этот конфигурационный файл по всей библиотеке (включая JAR-файлы):

```
ApplicationContext context = new  
    ClassPathXmlApplicationContext("spring.xml");
```

При конфигурировании с помощью Java, а не XML, можно использовать класс `AnnotationConfigApplicationContext`:

```
ApplicationContext context = new  
    AnnotationConfigApplicationContext(AppConfig.class);
```

Когда загружены конфигурационные файлы и получен контекст приложения, мы можем извлечь компонент из контейнера, вызвав метод `getBean()` контекста приложения:

```
TransferService transferService =  
    context.getBean(TransferService.class);
```

В следующем разделе мы поговорим о жизненном цикле компонента и о том, как контейнер Spring управляет им.

Жизнь компонента в контейнере

Контекст приложения Spring использует паттерн фабричного метода для создания компонентов Spring в контейнере в правильном порядке в соответствии с заданной конфигурацией. Таким образом, контейнер Spring отвечает за управление жизненным циклом компонента Spring от его появления до уничтожения. В нормальном приложении Java для инициализации компонента служит ключевое слово `new`, и он сразу готов к использованию. Когда компонент больше не применяется, он может быть собран сборщиком мусора. Но жизненный цикл компонента внутри контейнера Spring сложнее. На рис. 1.6 показана жизнь типичного компонента Spring.

Жизненный цикл компонента в контейнере выглядит следующим образом.

1. Загружаются все определения компонентов, тем самым создается ориентированный граф.
2. Создаются и запускаются экземпляры `BeanFactoryPostProcessor` (при этом можно обновить определения компонентов).

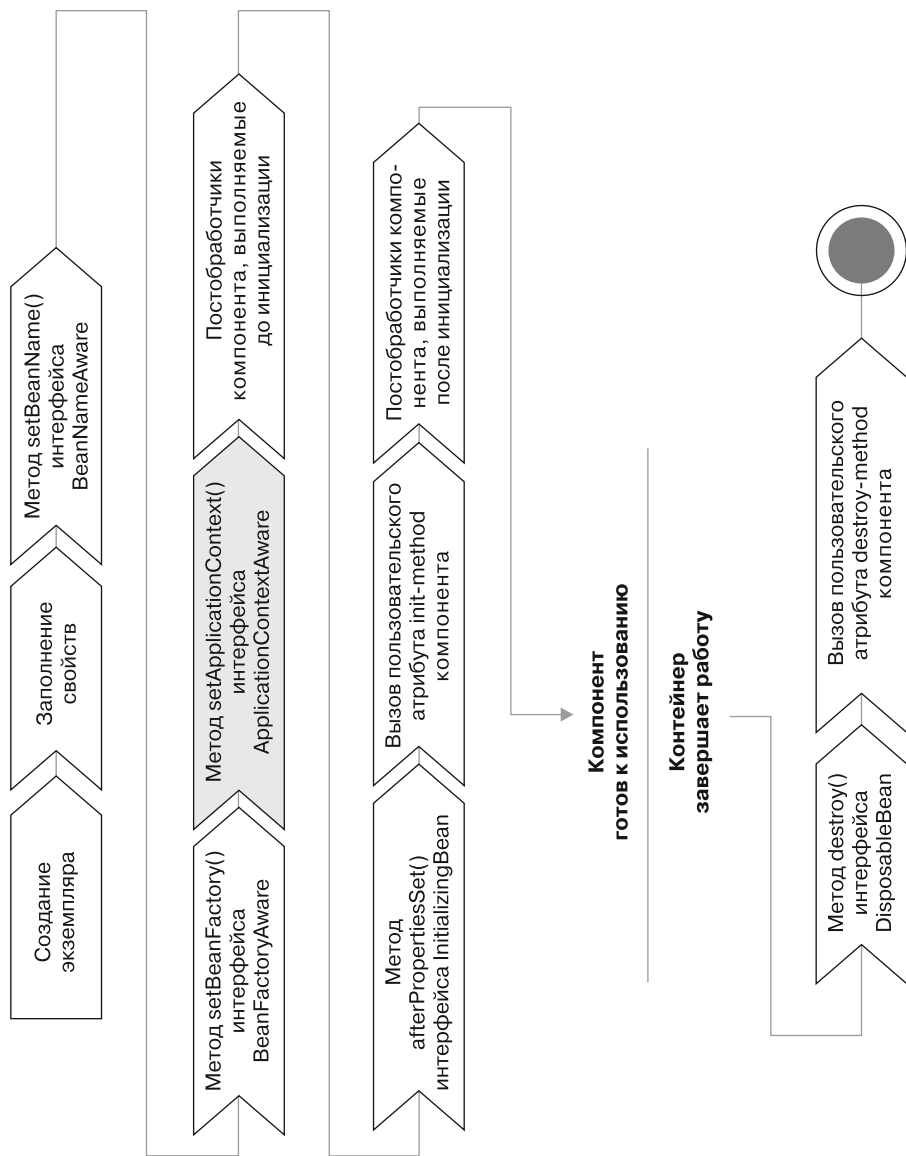


Рис. 1.6. Жизнь типичного компонента Spring

3. Создается экземпляр каждого компонента.
4. Spring внедряет значения и ссылки в свойства компонентов.
5. Spring передает идентификатор компонента методу `setBeanName()` интерфейса `BeanNameAware`, если некий компонент реализует его.
6. Spring передает ссылку на фабрику компонентов методу `setBeanFactory()` класса `BeanFactoryAware`, если некий компонент реализует его.
7. Spring передает ссылку на контекст приложения методу `setApplicationContext()` класса `ApplicationContextAware`, если некий компонент реализует его.
8. Если какие-либо из компонентов реализуют интерфейс `BeanPostProcessor`, то Spring модифицирует их экземпляры перед вызовом инициализации в контейнере, вызывая их методы `postProcessBeforeInitialization()`.
9. При реализации компонентом интерфейса `InitializingBean` Spring вызывает метод `afterPropertiesSet()` для инициализации процесса или загрузки ресурсов для приложения. Его работа зависит от выбранного метода реализации. Существуют и другие методы, выполняющие этот шаг, например `init-method` тега `<bean>`, атрибут `initMethod` аннотации `@Bean` и аннотация `@PostConstruct` из JSR 250.
10. Если какие-либо компоненты реализуют интерфейс `BeanPostProcessor`, то Spring модифицирует их экземпляры после вызова инициализации в контейнере, вызывая их методы `postProcessAfterInitialization()`.
11. Теперь ваш компонент готов к использованию и приложение может обращаться к нему, вызывая метод `getBean()` контекста приложения. Компоненты остаются жить в контексте приложения, пока он не будет закрыт вызовом метода `close()`.
12. Если компонент реализует интерфейс `DisposableBean`, то Spring вызывает его метод `destroy()` для завершения любого процесса или очистки ресурсов приложения. Есть и другие методы для этой цели — например, `destroy-method` тега `<bean>`, атрибут `destroyMethod` аннотации `@Bean` или аннотация `@PreDestroy` из JSR 250.

Следующий раздел описывает модули, предоставляемые фреймворком Spring.

Модули фреймворка Spring

Во фреймворке Spring предусмотрено несколько модулей для отдельных типов задач, более или менее независимых друг от друга. Это очень гибкая система, поэтому разработчик может выбрать только необходимые для приложения модули. Например, применять лишь модуль Spring DI и строить остальное приложение из компонентов, не являющихся компонентами Spring. Таким образом, Spring предоставляет возможности для интеграции с другими фреймворками и API — на-

пример, использовать паттерн внедрения зависимостей только для работы со Struts. Если проектировщики привыкли задействовать Struts, то можно заменить им Spring MVC, в то время как остальное приложение будет использовать компоненты и возможности Spring, например JDBC и транзакции. Так что, пока разработчикам нужно применять Struts, нет необходимости добавлять весь фреймворк Spring.

На рис. 1.7 представлен полный обзор модульной структуры.

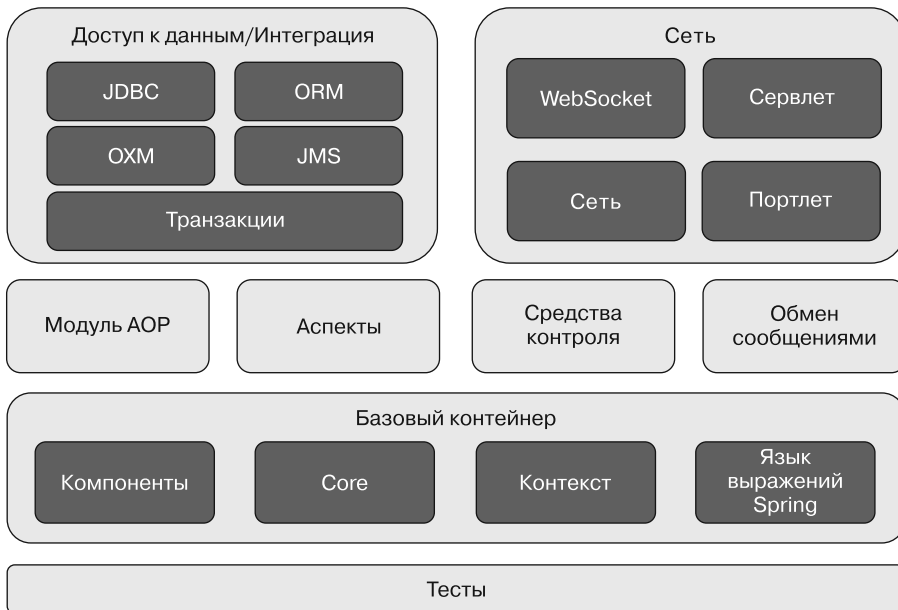


Рис. 1.7. Различные модули фреймворка Spring

Рассмотрим все модули и то, каким образом каждый из них вписывается в общую картину.

Core Container Spring

Данный модуль использует многие паттерны проектирования, в том числе «Фабричный метод», «Абстрактная фабрика», «Одиночка», «Прототип», паттерн внедрения зависимостей. Все остальные модули Spring зависят от него. Вы будете неявно применять его классы для конфигурирования приложений. Его также называют контейнером с обратным управлением, и он является ключевым для поддержки внедрения зависимостей, для создания, конфигурирования компонентов и управления ими. Можно создавать контейнер Spring, реализуя либо `BeanFactory`, либо `ApplicationContext`. Этот модуль содержит фабрику компонентов Spring, которая и обеспечивает внедрение зависимостей.

Модуль AOP

Spring AOP — основанный на Java фреймворк аспектно-ориентированного программирования, интегрированный с AspectJ. Он использует динамические прокси для внедрения аспектов и ориентирован на применение AOP для решения корпоративных задач. Этот модуль основан на паттернах проектирования «Заместитель» и «Декоратор». Он делает возможной модульность для сквозных задач, чтобы избежать запутывания и разбросанности. Как и внедрение зависимостей, такая модульность поддерживает слабую связанность между бизнес-логикой и сквозными задачами. Можно реализовывать собственные аспекты и декларативно конфигурировать их в приложении, не затрагивая код бизнес-объектов, что обеспечивает гибкость кода — можно удалить или изменить логику аспекта, не меняя бизнес-объекты. Это важный модуль фреймворка Spring, так что он будет подробно обсуждаться в главе 6.

Spring DAO — доступ к данным и интеграция

Spring DAO и Spring JDBC упрощают жизнь, устраняя общий код с помощью шаблонов. Последние реализуют паттерн шаблонного метода «банды четырех» и позволяют включать в нужные точки свой код. При работе с традиционным приложением JDBC приходится писать много стереотипного кода, чтобы, например, создать соединение с базой данных, определить и выполнить запрос, получить набор результатов, обработать исключения SQL и закрыть соединение. При работе с фреймворком Spring JDBC с уровнем DAO, в отличие от традиционного приложения JDBC, не приходится писать стереотипный код. Таким образом, Spring позволяет сохранять код приложения простым и понятным.

ORM

Spring также поддерживает решения ORM и предоставляет интеграцию с инструментами последнего. Этот модуль фактически является расширением модуля DAO. Как и шаблоны, основанные на JDBC, Spring предоставляет шаблоны ORM для работы с главными продуктами ORM, например Hibernate, JPA, OpenJPA, TopLink, iBATIS и т. д.

Web MVC

Spring предоставляет веб-модуль для корпоративных веб-приложений, позволяющий создавать очень гибкие веб-приложения, которые используют все возможности контейнера с обратным управлением. Он задействует такие паттерны, как MVC, «Единая точка входа» и «Сервлет-диспетчер», и легко интегрируется с сервлетным API. Веб-модуль Spring очень гибок и легко встраивается. Мы можем добавлять любые технологии просмотра, такие как JSP, FreeMarker, Velocity и т. д.

Мы также можем интегрировать его с другими фреймворками, например Struts, Webwork и JSF, использовать контейнеры с обратным управлением и внедрение зависимостей.

Новые возможности Spring Framework 5.0

Spring 5.0 — самая свежая версия фреймворка. В ней доступно много новых возможностей, включая описанные ниже.

- ❑ Поддерживается JDK 8 + 9 и Java EE 7 Baseline.

Java 8 — минимальное требование для Spring 5.0.

Фреймворк Spring требует как минимум Java EE 7 для работы приложений Spring Framework 5.0. Это значит, что требуется Servlet 3.1, JMS 2.0, JPA 2.1.

- ❑ Удалены устаревшие пакеты, классы и методы.

В Spring 5.0 некоторые пакеты были объявлены устаревшими или удалены. Например, пакет `mock.static` был удален из модуля `spring-aspects`, так что исчезла поддержка аспекта `AnnotationDrivenStaticEntityMockingControl`.

Такие пакеты, как `web.view.tiles2` и `orm.hibernate3/hibernate4`, тоже были удалены из Spring 5.0. В последней версии фреймворка используются Tiles 3 и Hibernate 5.

Фреймворк Spring 5.0 больше не поддерживает Portlet, Velocity, JasperReports, XMLBeans, JDO, Guava (и т. д.).

Некоторые классы и методы, объявленные устаревшими в более ранних версиях, были удалены из Spring 5.0.

- ❑ Добавлена новая модель реактивного программирования.

Эта модель программирования была введена в Spring 5.0. Напомню несколько фактов о модели реактивного программирования.

Spring 5 вводит модуль `DataBuffer` и абстракции кодирования-декодирования с неблокирующей семантикой в модель реактивного программирования.

Используя реактивную модель, Spring 5.0 предоставляет веб-модуль Spring для реализации кодеков сообщений HTTP с поддержкой *JSON (Jackson)* и *XML (JAXB)*.

Модель реактивного программирования в Spring добавляет новый модуль `spring-web-reactive` для реактивной поддержки модели программирования `@Controller`, адаптируя реактивные потоки к контейнерам Servlet 3.1, как и, например, к Netty и Undertow.

В Spring 5.0 также появляется интерфейс `WebClient` с реактивной поддержкой на стороне клиента.

Данный перечень позволяет увидеть, что в Spring 5 есть множество новых замечательных возможностей. В этой книге мы рассмотрим многие из них вместе с примерами и используемыми паттернами проектирования.

Резюме

Прочитав эту главу, вы смогли составить представление о фреймворке Spring и его наиболее часто используемых паттернах проектирования. Я акцентировал внимание на проблемах традиционных приложений J2EE и том, как Spring решает эти проблемы и упрощает разработку на Java, задействуя для создания приложения различные паттерны проектирования и лучшие практики.

Цель Spring — облегчение разработки корпоративных приложений на Java иощрение слабой связанности. Мы также рассмотрели аспектно-ориентированное программирование в Spring для сквозных задач и паттерн внедрения зависимостей для использования со слабой связанностью и подключаемыми компонентами Spring, так что объектам не надо знать, откуда приходят их зависимости и как они реализованы.

Фреймворк Spring позволяет использовать лучшие практики и эффективно проектировать объекты. Spring имеет два важных инструмента: во-первых, контейнеры для создания компонентов и управления их жизнью; во-вторых, поддержку нескольких модулей и интеграции для упрощения разработки.

2

Обзор паттернов проектирования GoF: базовые паттерны проектирования

В этой главе вы найдете обзор паттернов проектирования GoF, включая некоторые рекомендуемые практики создания приложений, а также узнаете, какие часто встречающиеся проблемы решаются с помощью паттернов проектирования.

Я расскажу о паттернах проектирования, часто используемых фреймворком Spring для улучшения архитектуры. Мы живем в эпоху глобализации, так что если какие-то сервисы вообще доступны на рынке, то они доступны во всем мире. Проще говоря, наступила эпоха распределенных вычислительных систем.

Прежде всего, что такое распределенная система? Это приложение, разбитое на несколько частей, работающих одновременно на различных компьютерах и взаимодействующих по сети, обычно с помощью сетевых протоколов. Эти меньшие части носят название *уровней* (tiers).

Для создания распределенного приложения многоуровневая архитектура оптимальна. Но создание многоуровневого распределенного приложения — непростая задача. Разбиение обработки на отдельные уровни ведет к оптимальному использованию ресурсов, а также позволяет распределять задачи между экспертами, лучше всего подходящими для проектировки конкретного уровня. При разработке распределенных приложений возникает множество сложных задач, в том числе:

- ❑ интеграция уровней;
- ❑ управление транзакциями;
- ❑ конкурентная обработка корпоративных данных;
- ❑ безопасность приложения и т. д.

В этой книге мы сосредоточим внимание на упрощении архитектуры и разработки приложений Java EE за счет применения паттернов проектирования и внедрения рекомендуемых практик с помощью фреймворка Spring. В данной главе будут рассмотрены некоторые распространенные паттерны проектирования GoF, а также их применение в Spring для оптимального решения вышеупомянутых проблем корпоративных приложений, ведь проектирование распределенных

объектов — чрезвычайно сложная задача даже для опытных профессионалов. Необходимо учесть многие критически важные вопросы, например масштабируемость, производительность, транзакции и т. д., прежде чем создать окончательный вариант проекта. Готовый вариант можно рассматривать в качестве паттерна.

К концу этой главы вы поймете, что паттерны проектирования оптимальным образом решают любые проблемы, связанные с созданием и разработкой, и узнаете, как в процессе воспользоваться рекомендуемыми практиками. Вы почерпнете много новой информации о паттернах проектирования GoF и увидите примеры их применения на практике. Вы получите сведения о внутренней реализации этих паттернов во фреймворке Spring для получения оптимального корпоративного решения.

В этой главе:

- ❑ возможности паттернов проектирования;
- ❑ обзор распространенных паттернов проектирования GoF;
 - базовые паттерны проектирования:
 - порождающие паттерны проектирования;
 - структурные паттерны проектирования;
 - поведенческие паттерны проектирования;
 - паттерны проектирования J2EE:
 - паттерны проектирования на уровне визуализации;
 - паттерны проектирования на бизнес-уровне;
 - паттерны проектирования на уровне интеграции;
- ❑ некоторые приемы, рекомендуемые для использования при разработке приложений Spring.

Возможности паттернов проектирования

Так что же такое паттерн проектирования? На самом деле паттерн проектирования не связан с каким-либо языком программирования и не обеспечивает никаких относящихся к конкретному языку решений задач. Паттерн проектирования предоставляет решение часто повторяющихся задач. Например, если какая-либо задача возникает часто, то применение для ее решения паттерна оправданно. Неиспользуемое повторно решение проблемы нельзя рассматривать как паттерн, проблема должна встречаться часто, чтобы имело смысл создать для нее используемое повторно решение, которое можно будет считать паттерном. Итак, *паттерн проектирования* — понятие инженерии программного обеспечения, описывающее часто применяемые решения для распространенных проблем проектирования ПО. Паттерны также отражают приемы, рекомендуемые опытными разработчиками объектно-ориентированного ПО.

При проектировании приложения следует анализировать все возможные решения часто встречающихся задач, именуемые паттернами проектирования. Команда

разработчиков должна хорошо понимать используемые паттерны, чтобы иметь возможность эффективно общаться друг с другом. На самом деле, вероятно, вам уже знакомы некоторые паттерны проектирования, хотя вы и не используете для их описания общепринятых наименований. В этой книге мы шаг за шагом продемонстрируем примеры на Java параллельно с изучением основ паттернов.

Паттерн проектирования имеет три отличительных признака.

- ❑ *Относится к конкретному сценарию*, а не к платформе. Его контекстом являются окружающие условия, в которых существует проблема. Контекст должен описываться в паттерне.
- ❑ Паттерны *разработаны для поиска оптимального решения* определенных задач, возникающих при создании программного обеспечения. Эти решения должны ограничиваться контекстом, в котором они рассматриваются.
- ❑ Паттерны *представляют собой решение только тех проблем, для которых предназначены*.

Например, если разработчик говорит о паттерне проектирования «Одиночка» из GoF и отмечает, что используется один объект, то все вовлеченные в процесс разработчики понимают, что необходимо спроектировать класс, у которого в приложении будет только один экземпляр. Одиночка состоит из одного объекта, и все разработчики смогут говорить друг другу, что программа следует паттерну «Одиночка».

Обзор часто используемых паттернов проектирования GoF

Эриха Гамму, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса часто называют GoF (Gang of Four, «банда четырех»). Они написали книгу «Приемы объектно-ориентированного проектирования: паттерны проектирования»¹, с которой началось применение этих паттернов в разработке программного обеспечения.

Из текущей главы вы узнаете, что такое паттерны GoF и как они помогают в решении задач, часто встречающихся при объектно-ориентированном проектировании.

Паттерны GoF — это 23 классических паттерна проектирования ПО, предлагающих используемые повторно решения распространенных задач проектирования ПО. Эти паттерны описаны в книге «Приемы объектно-ориентированного проектирования: паттерны проектирования» и делятся на две основные категории:

- ❑ базовые паттерны проектирования;
- ❑ паттерны проектирования J2EE.

¹ Гамма Э., Хелм Р., Джонсон Р. и Влиссидес Дж. Приемы объектно-ориентированного проектирования: паттерны проектирования. — СПб.: Питер, 2015.

Базовые паттерны проектирования делятся на три основные категории паттернов.

- ❑ *Порождающие*. Паттерны из этой категории позволяют конструировать объекты в тех случаях, когда одних конструкторов недостаточно. Логика создания объектов скрывается. Программы, в основе которых лежат эти паттерны, обеспечивают больше гибкости при создании объектов в соответствии с вашими нуждами и сценариями использования приложения.
- ❑ *Структурные*. Паттерны из этой категории имеют дело с композицией классов или объектов. В корпоративных приложениях существует две основные методики повторного использования функциональности в объектно-ориентированных системах: наследование классов и композиция объектов. Понятие наследования применяется для композиции интерфейсов и определения способов композиции объектов в целях создания новой функциональности.
- ❑ *Поведенческие*. Паттерны из этой категории касаются способов взаимодействия и распределения обязанностей между классами или объектами. Особенно тесно связаны с обменом информацией между объектами. Служат для контроля сложных потоков данных в корпоративных приложениях и их сокращения.

Теперь взглянем на другую основную категорию: *паттерны проектирования J2EE*. С их помощью можно существенно упростить процесс создания приложения. Паттерны проектирования Java EE были описаны в документации Java Blueprints компании Sun, которая представляет собой руководство по проверенным временем готовым решениям и рекомендуемым приемам взаимодействия объектов на различных уровнях приложений на платформе Java EE. Эти паттерны связаны со следующими уровнями приложения:

- ❑ уровнем визуализации;
- ❑ бизнес-уровнем;
- ❑ уровнем интеграции.

В следующем разделе мы изучим порождающие паттерны проектирования.

Порождающие паттерны проектирования

Обсудим основные паттерны из этой категории; то, как с их помощью фреймворк Spring обеспечивает слабое сцепление между компонентами, а также создание компонентов Spring и управление их жизненным циклом. Порождающие паттерны проектирования связаны со способом создания объекта. Логика создания объекта скрыта от обращающейся к объекту подпрограммы.

Все знают, как создать объект в Java с помощью ключевого слова `new`:

```
Account account = new Account();
```

Но для некоторых случаев такой вариант жестко «зашитого» способа создания объекта не подходит. Создавать объект подобным образом не рекомендуется, по-

сколько он может меняться в соответствии со структурой программы. А порождающий паттерн проектирования обеспечивает нужную гибкость в создании объекта.

Рассмотрим теперь различные паттерны проектирования из этой категории.

Паттерн проектирования «Фабрика»

Определяет интерфейс для создания объекта, оставляя подклассам выбор того, экземпляр какого класса создавать. Паттерн «Фабричный метод» позволяет классу делегировать создание объектов подклассам.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

«Фабрика» (Factory) — порождающий паттерн проектирования. Известен также как паттерн проектирования «Фабричный метод». Согласно ему для получения объекта класса можно не раскрывать клиенту нижележащую логику. Новый объект назначается вызывающей стороне с помощью общего интерфейса или абстрактного класса. Это значит, что данный паттерн проектирования скрывает фактическую логику реализации объекта, способ его создания и то, какой класс он воплощает. В результате клиента не будет заботить создание объекта, управление им и его уничтожение — паттерн «Фабрика» позаботится обо всем этом. Данный паттерн — один из наиболее используемых паттернов проектирования Java.

Ниже представлены преимущества паттерна «Фабрика»:

- ❑ способствует слабому сцеплению между взаимодействующими компонентами или классами благодаря использованию интерфейсов вместо привязывания классов, относящихся к конкретному приложению, к коду последнего;
- ❑ позволяет получить вариант объекта класса, реализующего интерфейс, во время выполнения программы;
- ❑ жизненный цикл объекта управляется фабрикой, реализуемой этим паттерном.

Паттерн проектирования «Фабрика» желательно использовать для решения следующих часто встречающихся проблем:

- ❑ чтобы избавить разработчика от бремени создания объектов и управления ими;
- ❑ во избежание тесного сцепления между взаимодействующими компонентами, поскольку компонент не может заранее знать, какие подклассы ему понадобятся создать;
- ❑ чтобы избежать жестко зашитого кода создания объекта класса.

Реализация паттерна во фреймворке Spring

Этот паттерн прозрачно используется во фреймворке Spring для реализации контейнеров Spring с помощью интерфейсов `BeanFactory` и `ApplicationContext`. Контейнеры Spring на основе паттерна «Фабрика» создают компоненты Spring для

приложений фреймворка и управляют жизненным циклом каждого из них. Интерфейсы `BeanFactory` и `ApplicationContext` являются фабричными, а в Spring есть множество классов реализаций. Метод `getBean()` — фабричный, возвращающий соответствующие компоненты Spring.

Рассмотрим пример реализации паттерна проектирования «Фабрика».

Пример реализации

Интерфейс `Account` реализуют два класса: `SavingAccount` и `CurrentAccount`. Так, можно создать класс `Factory` с методом, принимающим один или несколько аргументов и возвращающим объект типа `Account`. Такой метод называется фабричным, поскольку создает экземпляры или класса `CurrentAccount`, или класса `SavingAccount`. Интерфейс `Account` применяется для слабого сцепления. В соответствии с передаваемыми в фабричный метод аргументами он выбирает, экземпляр какого подкласса создать. Возвращаемым типом фабричного метода будет их суперкласс (рис. 2.1).

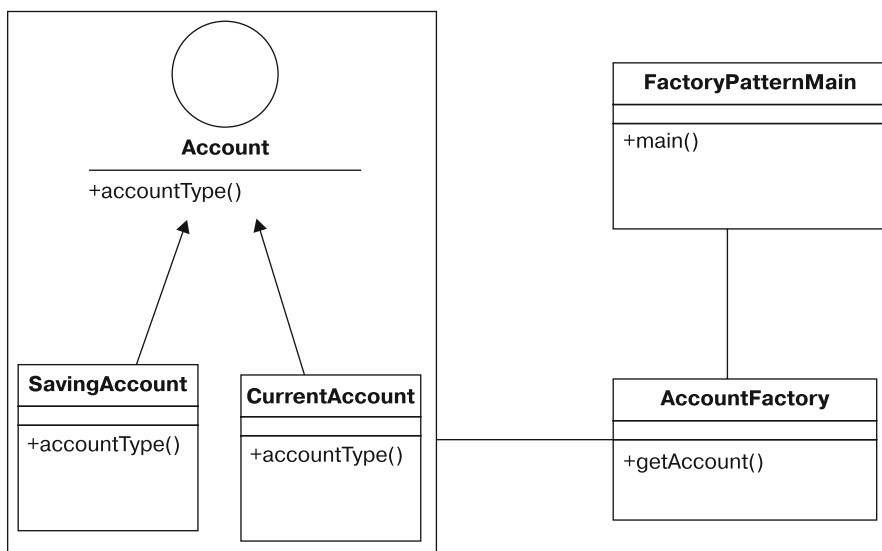


Рис. 2.1. UML-диаграмма для паттерна проектирования «Фабрика»

Рассмотрим этот паттерн проектирования на следующем примере. В нем я создам интерфейс `Account` и несколько конкретных классов, реализующих его:

```
package com.packt.patterninspring.chapter2.factory;
public interface Account {
    void accountType();
}
```

Теперь создадим класс `SavingAccount.java`, реализующий интерфейс `Account`:

```
package com.packt.patterninspring.chapter2.factory;
public class SavingAccount implements Account{
    @Override
    public void accountType() {
        System.out.println("SAVING ACCOUNT");
    }
}
```

Аналогично создаем класс `CurrentAccount.java`, который тоже реализует интерфейс `Account`:

```
package com.packt.patterninspring.chapter2.factory;
public class CurrentAccount implements Account {
    @Override
    public void accountType() {
        System.out.println("CURRENT ACCOUNT");
    }
}
```

Теперь нужно описать класс фабрики `AccountFactory`. Он генерирует объект конкретного класса — `CurrentAccount` или `SavingAccount`, в зависимости от типа счета, передаваемого в фабричный метод.

Фабрика `AccountFactory.java` генерирует объекты типа `Account`:

```
package com.packt.patterninspring.chapter2.factory.pattern;
import com.packt.patterninspring.chapter2.factory.Account;
import com.packt.patterninspring.chapter2.factory.CurrentAccount;
import com.packt.patterninspring.chapter2.factory.SavingAccount;
public class AccountFactory {
    final String CURRENT_ACCOUNT = "CURRENT";
    final String SAVING_ACCOUNT = "SAVING";
    // Используем метод getAccount для получения объекта типа Account
    // Фабричный метод для объектов типа Account
    public Account getAccount(String accountType){
        if(CURRENT_ACCOUNT.equals(accountType)) {
            return new CurrentAccount();
        }
        else if(SAVING_ACCOUNT.equals(accountType)){
            return new SavingAccount();
        }
        return null;
    }
}
```

Класс `FactoryPatternMain` — основной, обращающийся к классу `AccountFactory` для получения объектов `Account`. Он передает фабричному методу аргумент, содержащий информацию о типе счета, например `SAVING` («сберегательный») или `CURRENT` («текущий»). Класс `AccountFactory` возвращает объект типа, переданного фабричному методу.

Создадим демонстрационный класс `FactoryPatterMain.java`, чтобы попробовать на деле паттерн проектирования «Фабричный метод»:

```
package com.packt.patterninspring.chapter2.factory.pattern;
import com.packt.patterninspring.chapter2.factory.Account;
public class FactoryPatterMain {
    public static void main(String[] args) {
        AccountFactory accountFactory = new AccountFactory();
        // Получаем объект класса SavingAccount и вызываем
        // его метод accountType()
        Account savingAccount = accountFactory.getAccount("SAVING");
        // Вызываем метод accountType класса SavingAccount
        savingAccount.accountType();
        // Получаем объект класса CurrentAccount и вызываем
        // его метод accountType()
        Account currentAccount = accountFactory.getAccount("CURRENT");
        // Вызываем метод accountType класса CurrentAccount
        currentAccount.accountType();
    }
}
```

Можете запустить этот файл и посмотреть на результаты выполнения в консоли (рис. 2.2).

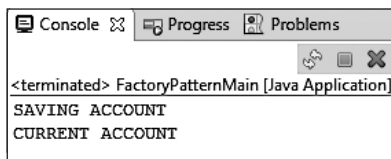


Рис. 2.2. Работа демонстрационного класса `FactoryPatterMain.java`

Паттерн проектирования «Абстрактная фабрика»

Обеспечивает интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов без указания конкретных классов.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

Паттерн проектирования «Абстрактная фабрика» (Abstract Factory) относится к порождающим. Это паттерн более высокого уровня, чем «Фабрика». Согласно ему необходимо просто описать интерфейс или абстрактный класс для создания взаимосвязанных или взаимозависимых объектов без указания конкретных классов. Итак, абстрактная фабрика возвращает фабрику классов. Позвольте пояснить это. Вы берете набор фабричных методов и объединяете эти фабрики в фабрику с помощью паттерна проектирования «Фабрика», получая фабрику фабрик. И вам

не требуется знать подробности обо всех фабриках в этой фабрике — для написания программы достаточно информации о фабрике верхнего уровня.

В паттерне «Абстрактная фабрика» интерфейс создает фабрику взаимосвязанных объектов без указания явным образом их классов. Каждая сгенерированная фабрика может генерировать объекты аналогично паттерну «Фабрика».

Преимущества паттерна «Абстрактная фабрика»:

- ❑ слабое сцепление между семействами компонентов, а также изоляция кода клиента от конкретных классов;
- ❑ это паттерн проектирования более высокого уровня, чем «Фабрика»;
- ❑ лучшая согласованность в масштабах приложения во время конструирования объектов;
- ❑ легкость замены семейств компонентов.

Распространенные задачи, для решения которых имеет смысл использовать паттерн «Абстрактная фабрика»

При проектировании паттерна «Фабрика» для создания объектов в приложении бывает необходимо скомпоновать определенный набор взаимосвязанных объектов с конкретными ограничениями и использовать в них нужную логику. Для этого можно создать внутри фабрики еще одну фабрику для набора взаимосвязанных объектов и наложить на них требуемые ограничения, а затем написать логику для данного набора.

Этот паттерн можно использовать, если требуется адаптировать логику создания объектов к конкретной задаче.

Реализация паттерна во фреймворке Spring

В основе интерфейса `FactoryBean` из фреймворка Spring лежит паттерн проектирования «Абстрактная фабрика». Spring предоставляет множество реализаций данного интерфейса, например `ProxyFactoryBean`, `IndiFactoryBean`, `LocalSessionFactoryBean`, `LocalContainerEntityManagerFactoryBean` и т. д. Интерфейс `FactoryBean` также помогает Spring конструировать те объекты, которые фреймворк не может легко сконструировать сам. Часто он служит для создания сложных объектов со множеством зависимостей. Его можно также использовать, когда логика конструирования объекта слишком изменчива и зависит от конфигурации.

Например, одна из реализаций интерфейса `FactoryBean` во фреймворке Spring — класс `LocalSessionFactoryBean`, используемый для получения ссылки на компонент, связанный с конфигурацией Hibernate. Это настройки, относящиеся конкретно к источнику данных. Их нужно применять до получения объекта `SessionFactory`. Класс `LocalSessionFactoryBean` можно задействовать для согласованного применения настроек конкретного источника данных. Результат, возвращаемый методом `getObject()` класса `FactoryBean`, можно внедрить в любое другое свойство.

Пример реализации

Я собираюсь создать интерфейсы `Bank` и `Account`, а также несколько конкретных классов, которые эти интерфейсы реализуют. Кроме того, я создам здесь класс абстрактной фабрики, `AbstractFactory`. Вдобавок у нас есть классы-фабрики, `BankFactory` и `AccountFactory`, расширяющие класс `AbstractFactory`. Опишу и класс `FactoryProducer` для создания фабрик.

Посмотрим на схему этого примера паттерна проектирования (рис. 2.3).

Создадим демонстрационный класс `AbstractFactoryPatternMain`, который использует класс `FactoryProducer` для получения объекта `AbstractFactory`. Я передаю в `AbstractFactory` следующую информацию: `ICICI`, `YES` для получения объекта класса `Bank`, а также передаю в `AbstractFactory` информацию `SAVING`, `CURRENT` для получения объекта типа `Account`.

Так выглядит код интерфейса `Bank.java`:

```
package com.packt.patterninspring.chapter2.model;
public interface Bank {
    void bankName();
}
```

Затем создадим класс `ICICIBank.java`, реализующий интерфейс `Bank`:

```
package com.packt.patterninspring.chapter2.model;
public class ICICIBank implements Bank {
    @Override
    public void bankName() {
        System.out.println("ICICI Bank Ltd.");
    }
}
```

Теперь создадим класс `YesBank.java`, тоже реализующий интерфейс `Bank`:

```
package com.packt.patterninspring.chapter2.model;
public class YesBank implements Bank{
    @Override
    public void bankName() {
        System.out.println("Yes Bank Pvt. Ltd.");
    }
}
```

В этом примере я использую тот же интерфейс `Account` и реализующие его классы, что и в вышеприведенном примере паттерна «Фабрика».

Класс `AbstractFactory.java` — абстрактный, с его помощью мы получаем фабрики объектов `Bank` и `Account`:

```
package com.packt.patterninspring.chapter2.abstractfactory.pattern;
import com.packt.patterninspring.chapter2.model.Account;
import com.packt.patterninspring.chapter2.model.Bank;
public abstract class AbstractFactory {
    abstract Bank getBank(String bankName);
    abstract Account getAccount(String accountType);
}
```

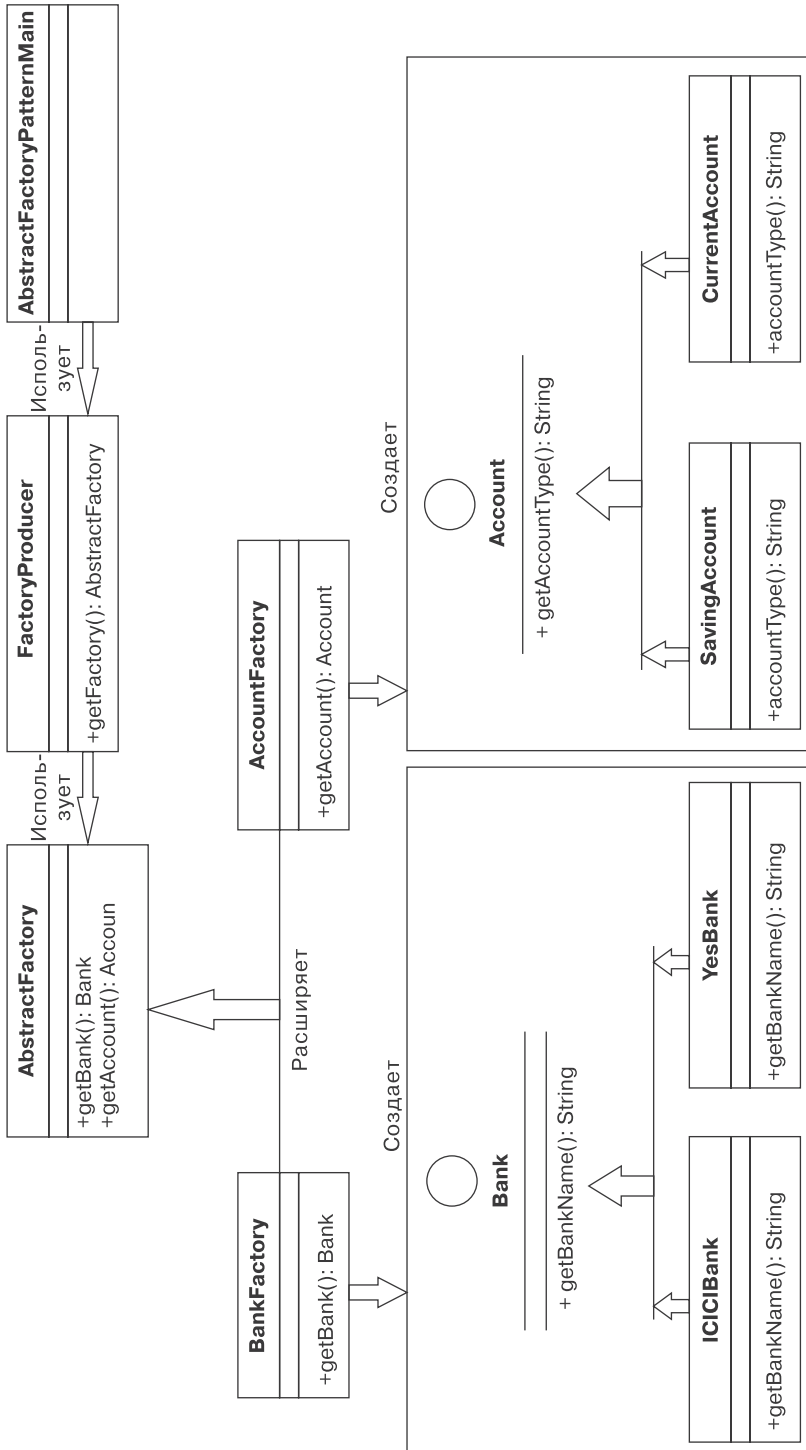


Рис. 2.3. UML-диаграмма для паттерна проектирования «Абстрактная фабрика»

Класс `BankFactory.java` — класс фабрики, расширяющий абстрактный класс `AbstractFactory` в целях генерации объектов конкретного класса по заданной информации:

```
package com.packt.patterninspring.chapter2.abstractfactory.pattern;
import com.packt.patterninspring.chapter2.model.Account;
import com.packt.patterninspring.chapter2.model.Bank;
import com.packt.patterninspring.chapter2.model.ICICIBank;
import com.packt.patterninspring.chapter2.model.YesBank;
public class BankFactory extends AbstractFactory {
    final String ICICI_BANK = "ICICI";
    final String YES_BANK = "YES";
    // Получаем объект указанного банка с помощью метода getBank
    // Это фабричный метод для создания объекта указанного банка
    @Override
    Bank getBank(String bankName) {
        if(ICICI_BANK.equalsIgnoreCase(bankName)){
            return new ICICIBank();
        }
        else if(YES_BANK.equalsIgnoreCase(bankName)){
            return new YesBank();
        }
        return null;
    }
    @Override
    Account getAccount(String accountType) {
        return null;
    }
}
```

`AccountFactory.java` — класс фабрики, расширяющий абстрактный класс `AbstractFactory` в целях генерации объектов конкретного класса по заданной информации:

```
package com.packt.patterninspring.chapter2.abstractfactory.pattern;
import com.packt.patterninspring.chapter2.model.Account;
import com.packt.patterninspring.chapter2.model.Bank;
import com.packt.patterninspring.chapter2.model.CurrentAccount;
import com.packt.patterninspring.chapter2.model.SavingAccount;
public class AccountFactory extends AbstractFactory {
    final String CURRENT_ACCOUNT = "CURRENT";
    final String SAVING_ACCOUNT = "SAVING";
    @Override
    Bank getBank(String bankName) {
        return null;
    }
    // Получаем объект указанного типа счета с помощью метода getAccount
    // Это фабричный метод для создания объектов указанного типа счета
    @Override
    public Account getAccount(String accountType){
```



```

        if(CURRENT_ACCOUNT.equals(accountType)) {
            return new CurrentAccount();
        }
        else if(SAVING_ACCOUNT.equals(accountType)){
            return new SavingAccount();
        }
        return null;
    }
}

```

Класс `FactoryProducer.java` служит для создания генератора фабрик, позволяющего передавать информацию и получать в ответ соответствующую фабрику, например объектов класса `Bank` или `Account`:

```

package com.packt.patterninspring.chapter2.abstractfactory.pattern;
public class FactoryProducer {
    final static String BANK = "BANK";
    final static String ACCOUNT = "ACCOUNT";
    public static AbstractFactory getFactory(String factory){
        if(BANK.equalsIgnoreCase(factory)){
            return new BankFactory();
        }
        else if(ACCOUNT.equalsIgnoreCase(factory)){
            return new AccountFactory();
        }
        return null;
    }
}

```

Класс `FactoryPatternMain.java` демонстрирует паттерн проектирования «Абстрактная фабрика». Класс `FactoryProducer` позволяет получить объект `AbstractFactory` для передачи ему информации, например типа счета, и получения от него фабрик конкретных классов:

```

package com.packt.patterninspring.chapter2.factory.pattern;
import com.packt.patterninspring.chapter2.model.Account;
public class FactoryPatternMain {
    public static void main(String[] args) {
        AccountFactory accountFactory = new AccountFactory();
        // Получаем объект класса SavingAccount и вызываем его метод accountType()
        Account savingAccount = accountFactory.getAccount("SAVING");
        // Вызываем метод accountType класса SavingAccount
        savingAccount.accountType();
        // Получаем объект класса CurrentAccount и вызываем его метод accountType()
        Account currentAccount = accountFactory.getAccount("CURRENT");
        // Вызываем метод accountType класса CurrentAccount
        currentAccount.accountType();
    }
}

```

Можете проверить этот код в действии (рис. 2.4).

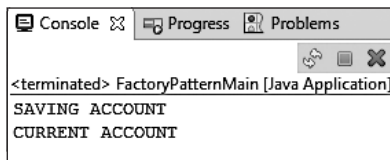


Рис. 2.4. Результаты работы класса FactoryPatternMain.java

Паттерн проектирования «Одиночка»

Гарантирует наличие только одного экземпляра класса в приложении и предоставляет глобальную точку доступа к нему.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

Паттерн проектирования «Одиночка» (Singleton) — один из порождающих паттернов проектирования и один из простейших паттернов в языке Java. Согласно этому паттерну класс использует для всех вызовов один и тот же объект, то есть создание экземпляров класса ограничивается одним объектом с предоставлением глобальной точки доступа к данному классу. Это значит следующее: класс отвечает за создание объекта, а также гарантирует, что для всех клиентских обращений к этому объекту будет создан лишь один объект. Данный класс не допускает непосредственного создания своих объектов. Получить его экземпляр можно только с помощью предоставляемого им статического метода.

Этот паттерн проектирования удобен, когда необходим ровно один объект для согласования действий в масштабах всей системы. Создать экземпляр можно двумя способами:

- ❑ *немедленное создание экземпляра* (early instantiation) — экземпляр создается во время загрузки;
- ❑ *отложенное создание экземпляра* (lazy instantiation) — экземпляр создается при необходимости.

Достоинства паттерна «Одиночка»:

- ❑ предоставляет контроллеру доступ к важнейшим (обычно с «тяжелыми» объектами) классам, например к классу соединения с базой данных и к классу SessionFactory в Hibernate;
- ❑ экономит уйму памяти;
- ❑ представляет собой весьма эффективную архитектуру для многопоточных сред;
- ❑ обеспечивает повышенную гибкость за счет того, что класс контролирует процесс создания экземпляра и может вносить в этот процесс изменения;
- ❑ обеспечивает низкое значение задержки.

Распространенные задачи, для решения которых имеет смысл использовать паттерн «Одиночка»

Паттерн «Одиночка» решает только одну задачу: если допустим только один экземпляр какого-либо ресурса и вы хотели бы управлять этим одним экземпляром — вам нужен одиночка. В обычных обстоятельствах при создании соединения с БД с заданными настройками в распределенной и многопоточной среде решение не следовать паттерну «Одиночка» может привести к ситуации, когда каждый поток выполнения создает по новому соединению с базой данных с другим объектом конфигурации. А благодаря паттерну «Одиночка» у всех потоков выполнения в системе будет один и тот же объект соединения с базой с одним и тем же объектом конфигурации. Чаще всего этот паттерн используется в многопоточных приложениях и приложениях БД. Применяется и при журналировании, кэшировании, для пулов потоков выполнения, параметров конфигурации и т. д.

Реализация паттерна во фреймворке Spring

Фреймворк Spring предлагает в качестве реализации паттерна «Одиночка» единичную область видимости для компонентов. Она схожа с данным паттерном, но не идентична таковому в языке Java. Компонент с единичной областью видимости в Spring означает один экземпляр компонента в расчете на компонент и в расчете на контейнер. Если описать один компонент определенного класса в отдельном контейнере Spring, то последний создаст один и только один экземпляр класса с данным описанием компонента.

Пример реализации

В следующем примере кода я представлю класс с методом, создающим экземпляр этого класса при отсутствии такового. Если экземпляр уже существует, то упомянутый метод просто вернет ссылку на него. Я также учел вопрос потокобезопасности и использовал синхронизированную блокировку, прежде чем создавать объект класса.

```
package com.packt.patterninspring.chapter2.singleton.pattern;
public class SingletonClass {
    private static SingletonClass instance = null;
    private SingletonClass() {
    }
    public static SingletonClass getInstance() {
        if (instance == null) {
            synchronized(SingletonClass.class){
                if (instance == null) {
                    instance = new SingletonClass();
                }
            }
        }
        return instance;
    }
}
```

В этом коде я объявил конструктор класса `SingletonClass` как приватный, чтобы не существовало никакого способа создать объект данного класса. В основе этого примера лежит отложенная инициализация, а значит, программа создает экземпляр при первой потребности в нем. Можно также с легкостью создать объект немедленно, чтобы повысить производительность приложения во время его выполнения. Взглянем на тот же класс `SingletonClass` с незамедлительной инициализацией:

```
package com.packt.patterninspring.chapter2.singleton.pattern;  
public class SingletonClass {  
    private static final SingletonClass INSTANCE =  
        new SingletonClass();  
    private SingletonClass() {}  
    public static SingletonClass getInstance() {  
        return INSTANCE;  
    }  
}
```

Паттерн проектирования «Прототип»

Задаёт вид создаваемых объектов с помощью экземпляра-прототипа и создаёт новые объекты путем его копирования.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

Паттерн проектирования «Прототип» (Prototype) относится к семейству порождающих из числа паттернов разработки программного обеспечения GoF. Он используется для создания объектов путем клонирования на основе экземпляра-прототипа. В корпоративных приложениях создание объекта требует значительных затрат ресурсов в смысле непосредственного создания объектов и инициализации начальных значений их свойств. В случае подобного объекта имеет смысл воспользоваться паттерном проектирования «Прототип». При этом просто копируется уже существующий подобный объект вместо требующего временных затрат создания нового.

Этот паттерн включает реализацию прототипного интерфейса в целях создания клона существующего объекта. Используется в случаях, когда непосредственное создание объекта требует значительных затрат ресурсов, например, когда объект создается после дорогостоящей операции с базой данных. Объект можно закэшировать, вернуть его клон при следующем запросе и обновлять базу по мере необходимости, уменьшая таким образом количество требуемых обращений к ней.

Преимущества паттерна

Преимущества использования паттерна проектирования «Прототип» таковы:

- ❑ уменьшает время, необходимое для создания объектов, благодаря использованию прототипа;

- ❑ сокращает создание производных классов;
- ❑ добавляет и удаляет объекты во время выполнения;
- ❑ динамически задает настройки классов приложения.

Структура классов в виде UML-диаграммы

UML-диаграмма на рис. 2.5 показывает все компоненты паттерна проектирования «Прототип».

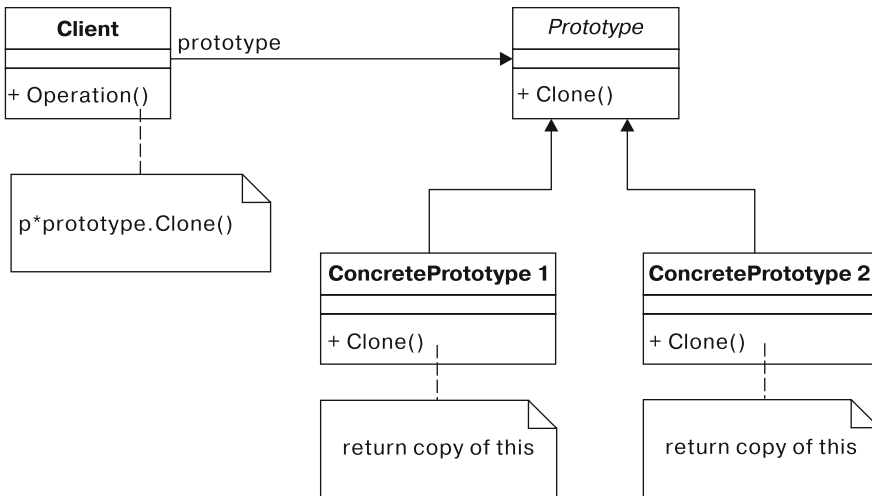


Рис. 2.5. UML-диаграмма для паттерна проектирования «Прототип»

Рассмотрим эти компоненты.

- ❑ **Prototype** (Прототип). Это интерфейс. Объявляет метод `clone()` для клонирования себя.
- ❑ **ConcretePrototype** (Конкретный прототип). Конкретный класс интерфейса **Prototype**, реализующий операцию клонирования.
- ❑ **Client** (Клиент). Класс `caller` предназначен для создания нового объекта с помощью вызова метода `clone` интерфейса-прототипа.

Пример реализации

Я хочу создать абстрактный класс **Account** и расширяющие его конкретные классы. Описываемый далее класс **AccountCache** сохраняет объекты счетов в объекте **HashMap** и возвращает их клоны при необходимости. Создайте абстрактный класс, реализующий интерфейс **Clonable**:

```
package com.packt.patterninspring.chapter2.prototype.pattern;
public abstract class Account implements Clonable{
    abstract public void accountType();
}
```

```

public Object clone() {
    Object clone = null;
    try {
        clone = super.clone();
    }
    catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return clone;
}
}

```

Теперь создадим конкретные классы, расширяющие предыдущий.

Вот файл `CurrentAccount.java`:

```

package com.packt.patterninspring.chapter2.prototype.pattern;
public class CurrentAccount extends Account {
    @Override
    public void accountType() {
        System.out.println("CURRENT ACCOUNT");
    }
}

```

А вот так должен выглядеть файл `SavingAccount.java`:

```

package com.packt.patterninspring.chapter2.prototype.pattern;
public class SavingAccount extends Account{
    @Override
    public void accountType() {
        System.out.println("SAVING ACCOUNT");
    }
}

```

Теперь создадим класс для получения конкретных классов в файле `AccountCache.java`:

```

package com.packt.patterninspring.chapter2.prototype.pattern;
import java.util.HashMap;
import java.util.Map;
public class AccountCache {
    public static Map<String, Account> accountCacheMap =
        new HashMap<>();
    static{
        Account currentAccount = new CurrentAccount();
        Account savingAccount = new SavingAccount();
        accountCacheMap.put("SAVING", savingAccount);
        accountCacheMap.put("CURRENT", currentAccount);
    }
}

```

Класс `PrototypePatternMain.java` демонстрационный, им мы воспользуемся для проверки паттерна проектирования `AccountCache` — создания объекта `Account`

в соответствии с переданной информацией в виде типа счета путем вызова метода `clone()`:

```
package com.packt.patterninspring.chapter2.prototype.pattern;  
public class PrototypePatternMain {  
    public static void main(String[] args) {  
        Account currentAccount = (Account)  
            AccountCache.accountCacheMap.get("CURRENT").clone();  
        currentAccount.accountType();  
        Account savingAccount = (Account)  
            AccountCache.accountCacheMap.get("SAVING").clone();  
        savingAccount.accountType();  
    }  
}
```

Паттерн проектирования «Строитель»

Отделяет конструирование сложного объекта от его представления, чтобы с помощью одного процесса формирования можно было создавать различные представления.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

Паттерн проектирования «Строитель» (Builder) используется для пошагового конструирования сложных объектов, возвращая в результате законченный объект. Логика и весь процесс создания объекта должны быть обобщенными, чтобы можно было использовать их для создания различных конкретных реализаций одного объектного типа. Данный паттерн упрощает конструирование сложных объектов и скрывает нюансы конструирования объекта от вызывающего клиентского кода. При использовании этого паттерна не забывайте, что процесс должен быть пошаговым, то есть вы должны разбить логику конструирования объекта на несколько этапов, в отличие от абстрактной фабрики и фабричного метода, где объект создается за один шаг.

Преимущества паттерна

- ❑ Полная изоляция процесса конструирования объекта от его представления.
- ❑ Возможность конструирования объекта в несколько этапов, что позволяет получить больше контроля над процессом конструирования.
- ❑ Широкие возможности варьирования внутреннего представления объекта.

Структура классов в виде UML-диаграммы

На следующей UML-диаграмме (рис. 2.6) показаны все компоненты паттерна проектирования «Строитель».

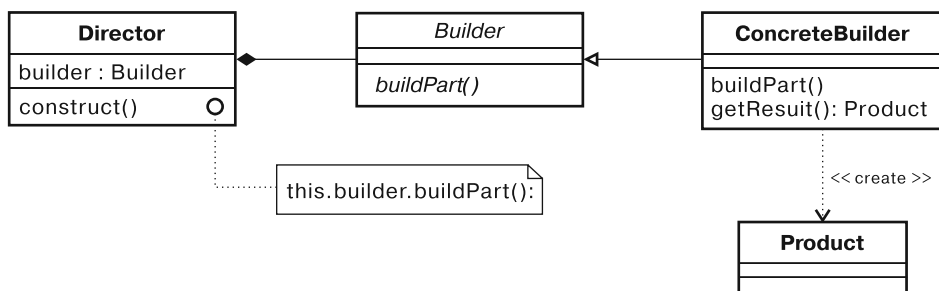


Рис. 2.6. UML-диаграмма для паттерна проектирования «Строитель»

UML-диаграмма для паттерна проектирования «Строитель» включает следующие элементы.

- ❑ **Builder** (Строитель) — **AccountBuilder**. Абстрактный класс или интерфейс для создания частей объекта **Account**.
- ❑ **ConcreteBuilder** (Конкретный строитель). Конструирует и собирает воедино части счета с помощью реализации интерфейса **Builder**.
- ❑ **Director** (Распорядитель). Конструирует объект с помощью интерфейса **Builder**.
- ❑ **Product** (Продукт). Представляет конструируемый сложный объект. **AccountBuilder** формирует внутреннее представление счета и задает процесс его сборки.

Реализация паттерна во фреймворке Spring

Фреймворк Spring прозрачным образом реализует паттерн проектирования «Строитель» для некоторых функциональностей. В Spring на паттерне проектирования «Строитель» основаны следующие классы:

- ❑ `EmbeddedDatabaseBuilder;`
- ❑ `AuthenticationManagerBuilder;`
- ❑ `UriComponentsBuilder;`
- ❑ `BeanDefinitionBuilder;`
- ❑ `MockMvcWebClientBuilder.`

Распространенные задачи, для решения которых имеет смысл использовать паттерн «Строитель»

В корпоративных приложениях паттерн проектирования «Строитель» имеет смысл использовать, если процесс создания объекта состоит из нескольких шагов. На каждом из шагов выполняется часть процесса. В ходе создания объекта задаются необходимые и дополнительные (необязательные) параметры, и в результате получается сложный объект.

Паттерн «Строитель» представляет собой паттерн создания объектов. Его цель состоит в абстрагировании этапов конструирования объекта, чтобы с помощью их различных реализаций можно было конструировать разные представления объектов. Строитель часто используется для создания продуктов в соответствии с паттерном проектирования «Составной объект».

Пример реализации

В следующем примере кода я собираюсь создать класс `Account` с внутренним классом `AccountBuilder`. В последнем имеется метод для создания его экземпляра:

```
package com.packt.patterninspring.chapter2.builder.pattern;
public class Account {
    private String accountName;
    private Long accountNumber;
    private String accountHolder;
    private double balance;
    private String type;
    private double interest;
    private Account(AccountBuilder accountBuilder) {
        super();
        this.accountName = accountBuilder.accountName;
        this.accountNumber = accountBuilder.accountNumber;
        this.accountHolder = accountBuilder.accountHolder;
        this.balance = accountBuilder.balance;
        this.type = accountBuilder.type;
        this.interest = accountBuilder.interest;
    }
    // Сеттеры и геттеры
    public static class AccountBuilder {
        private final String accountName;
        private final Long accountNumber;
        private final String accountHolder;
        private double balance;
        private String type;
        private double interest;
        public AccountBuilder(String accountName,
            String accountHolder, Long accountNumber) {
            this.accountName = accountName;
            this.accountHolder = accountHolder;
            this.accountNumber = accountNumber;
        }
        public AccountBuilder balance(double balance) {
            this.balance = balance;
            return this;
        }
        public AccountBuilder type(String type) {
            this.type = type;
            return this;
        }
    }
}
```

```

    public AccountBuilder interest(double interest) {
        this.interest = interest;
        return this;
    }
    public Account build() {
        Account user = new Account(this);
        return user;
    }
}
public String toString() {
    return "Account [accountName=" + accountName + ",
        accountNumber=" + accountNumber + ", accountHolder="
        + accountHolder + ", balance=" + balance + ", type="
        + type + ", interest=" + interest + "]";
}
}

```

Для проверки работы паттерна мы воспользуемся демонстрационным классом `AccountBuilderTest.java`. Взглянем на то, как создать объект `Account` путем передачи ему исходных данных:

```

package com.packt.patterninspring.chapter2.builder.pattern;
public class AccountBuilderTest {
    public static void main(String[] args) {
        Account account = new Account.AccountBuilder("Saving
            Account", "Dinesh Rajput", 1111)
            .balance(38458.32)
            .interest(4.5)
            .type("SAVING")
            .build();
        System.out.println(account);
    }
}

```

Можете запустить этот файл. Результат будет примерно таким, как на рис. 2.7.

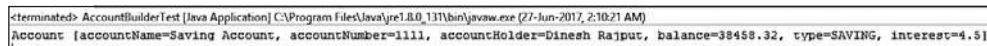


Рис. 2.7. Результат работы класса `AccountBuilderTest.java`

Резюме

После прочтения этой главы вы должны иметь хорошее представление о порождающих паттернах проектирования GoF и приемах, рекомендуемых при их использовании. Я подчеркнул проблемы, возникающие при разработке корпоративных приложений без применения паттернов проектирования, и рассказал о решении этих проблем фреймворком Spring с помощью порождающих паттернов проектирования и следования рекомендуемым практикам.

В этой главе я упомянул только одну из трех категорий паттернов проектирования GoF — порождающие паттерны. Они служат для создания экземпляров

классов, а также для применения ограничений особым образом во время создания объектов, с помощью паттернов «Фабрика», «Абстрактная фабрика», «Строитель», «Прототип» и «Одиночка».

В следующей главе мы рассмотрим другие категории паттернов проектирования GoF — структурные паттерны и поведенческие. Первые используются для проектирования структуры корпоративных приложений с помощью композиции классов или объектов в целях снижения сложности приложения и повышения его производительности и возможностей повторного использования. К этой категории относятся паттерны «Адаптер», «Мост», «Составной объект», «Декоратор», «Фасад» и «Приспособленец». Поведенческие паттерны проектирования описывают способы взаимодействия и распределения обязанностей между классами или объектами. Паттерны из этой категории специально предназначены для взаимодействия между объектами.

3

Соображения по поводу структурных и поведенческих паттернов

В главе 2 вы уже видели реализации и примеры порождающих паттернов проектирования из семейства паттернов GoF. Здесь же я приведу обзор других представителей семейства GoF, а именно структурных и поведенческих паттернов, а также рекомендуемых приемов проектирования приложений. Кроме того, мы разберем несколько часто встречающихся задач, решаемых с помощью этих паттернов.

К концу данной главы вы поймете, почему эти паттерны предоставляют оптимальное решение проблем проектирования и разработки, когда речь идет о формировании объектов и делегировании обязанностей между работающими в приложении объектами. Вы получите информацию о внутренней реализации фреймворком Spring структурных и поведенческих паттернов проектирования, нацеленной на создание оптимального корпоративного решения.

В этой главе:

- ❑ реализация структурных паттернов проектирования;
- ❑ реализация поведенческих паттернов проектирования;
- ❑ паттерны проектирования J2EE.

Базовые паттерны проектирования

Продолжим изучение базовых паттернов проектирования.

- ❑ *Структурные паттерны проектирования.* Паттерны из этой категории имеют дело с композицией классов или объектов. В корпоративных приложениях существует две основные методики повторного использования функциональности в объектно-ориентированных системах:
 - *наследование* — служит для наследования от других классов часто применяемых состояний и поведений;
 - *композиция* — используется для составления других объектов в виде переменных экземпляров классов. Определяет способы композиции объектов в целях получения новых видов функциональности.

- *Поведенческие паттерны проектирования.* Характеризуют способы взаимодействия классов или объектов и распределения обязанностей между ними. Определяют методы обмена данными между объектами в корпоративных приложениях. Из этой главы вы узнаете, как использовать поведенческие паттерны для сокращения сложных потоков данных. Более того, вы научитесь применять поведенческие паттерны для инкапсуляции алгоритмов и динамического выбора их во время выполнения.

Структурные паттерны проектирования

В предыдущей главе мы обсуждали порождающие паттерны проектирования, обеспечивающие оптимальные варианты создания объектов в соответствии с бизнес-требованиями. Порождающие паттерны решают лишь задачу создания объектов. Когда же речь заходит о том, как эти объекты объединяются между собой в приложении для конкретных бизнес-целей, оказываются нужны структурные паттерны проектирования. В текущей главе мы будем изучать структурные паттерны и их возможности по описанию отношений между объектами с помощью или наследования, или композиции в более крупные структуры данных приложения. Структурные паттерны позволяют решить множество задач структурирования связей между объектами. Они демонстрируют разработчику гибкие и расширяемые способы «склеивания» воедино различных частей системы. Благодаря структурным паттернам можно гарантировать, что при изменении одной из частей не придется менять всю структуру, подобно тому как в автомобиле можно установить шины от другого производителя, не влияя на остальные части машины. Эти паттерны также указывают способы переделки нужных, но плохо сочетающихся с остальной системой частей.

Паттерн проектирования «Адаптер»

Преобразует интерфейс класса в другой интерфейс, ожидаемый клиентом. Благодаря адаптеру становится возможным взаимодействие тех классов, которые иначе не могли бы работать друг с другом из-за несовместимых интерфейсов.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

Паттерн проектирования «Адаптер» относится к структурным. Благодаря ему могут работать друг с другом два несовместимых класса, которые иначе не могли бы делать это из-за несовместимых интерфейсов. Данный паттерн служит мостом между двумя несовместимыми интерфейсами и используется, когда два интерфейса приложения функционально несовместимы, но их нужно объединить из-за бизнес-требования.

Существует множество примеров из реальной жизни, в которых можно применить паттерн «Адаптер». Допустим, у вас есть разные типы вилок электроприборов — цилиндрические, прямоугольные и т. п. (рис. 3.1). Адаптер позволяет

использовать прямоугольную вилку для цилиндрического разъема (при условии, что вольтаж совпадает).



Рис. 3.1. Пример применения адаптера

Преимущества использования

У использования паттерна проектирования «Адаптер» в приложении есть следующие положительные стороны:

- ❑ возможность взаимодействия между двумя или более несовместимыми объектами;
- ❑ этот паттерн способствует повторному использованию уже существующих в приложении более старых элементов функциональности.

Области применения

Паттерн «Адаптер» имеет смысл применять для решения проблем проектирования в следующих случаях:

- ❑ вам нужно воспользоваться существующим классом, интерфейс которого несовместим с вашими нуждами;
- ❑ вам хотелось бы создать в своем приложении повторно используемый класс, который должен взаимодействовать с классами с несовместимыми интерфейсами;
- ❑ необходимо использовать несколько уже существующих подклассов, однако адаптировать их интерфейсы, создавая производные подклассы от каждого из них, нецелесообразно. Лучше применить адаптер объектов для подгонки интерфейса родительского класса.

Реализация паттерна во фреймворке Spring

Паттерн «Адаптер» повсеместно используется в Spring для прозрачной реализации множества различных элементов функциональности. Ниже представлены несколько основанных на этом паттерне классов из фреймворка:

- ❑ `JpaVendorAdapter`;
- ❑ `HibernateJpaVendorAdapter`;
- ❑ `HandlerInterceptorAdapter`;
- ❑ `MessageListenerAdapter`;

- ❑ `SpringContextResourceAdapter`;
- ❑ `ClassPreProcessorAgentAdapter`;
- ❑ `RequestMappingHandlerAdapter`;
- ❑ `AnnotationMethodHandlerAdapter`;
- ❑ `WebMvcConfigurerAdapter`.

UML-диаграмма паттерна проектирования «Адаптер». Разберемся в следующей UML-диаграмме, иллюстрирующей компоненты паттерна проектирования «Адаптер» (рис. 3.2).

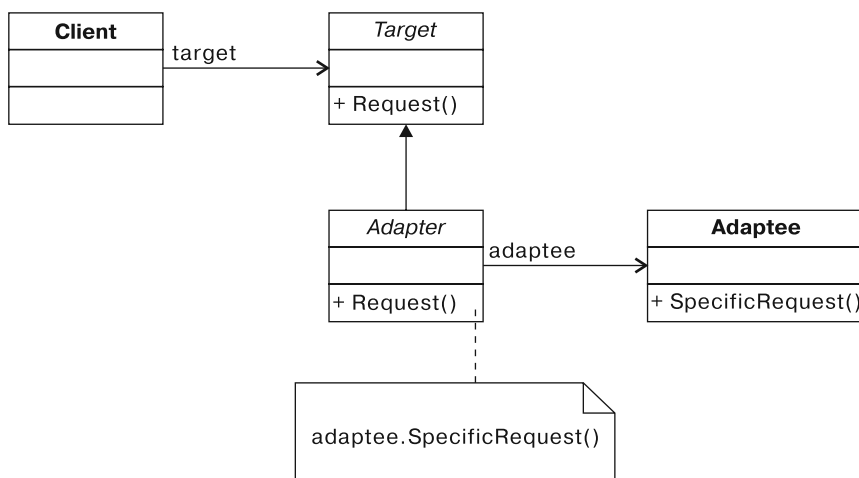


Рис. 3.2. UML-диаграмма паттерна проектирования «Адаптер»

- ❑ **Target (Цель).** Класс целевого интерфейса, который будут использовать клиенты.
- ❑ **Adapter (Адаптер).** Класс-адаптер, реализующий желаемый целевой интерфейс и модифицирующий конкретный запрос класса `Adaptee`.
- ❑ **Adaptee (Адаптируемый).** Класс, задействуемый классом `Adapter` для повторного использования существующей функциональности и модификации ее под нужный вид применения.
- ❑ **Client.** Класс, взаимодействующий с классом `Adapter`.

Пример реализации

Я собираюсь создать пример, демонстрирующий реальное использование паттерна проектирования «Адаптер». В основе паттерна лежит выполнение оплаты через платежную систему. Пусть у нас есть старая платежная система и более совершенная новая, не связанные друг с другом, и нам нужно перейти от использования старой системы к новой с изменением существующего исходного кода. Для решения

данной задачи я создам класс-адаптер. Он служит мостом между двумя различными платежными системами. Изучите следующий код.

Создадим интерфейс для старой платежной системы:

```
package com.packt.patterninspring.chapter3.adapter.pattern;
import com.packt.patterninspring.chapter3.model.Account;
public interface PaymentGateway {
    void doPayment(Account account1, Account account2);
}
```

Теперь создадим класс для реализации старой платежной системы PaymentGateway.java:

```
package com.packt.patterninspring.chapter3.adapter.pattern;
import com.packt.patterninspring.chapter3.model.Account;
public class PaymentGatewayImpl implements PaymentGateway{
    @Override
    public void doPayment(Account account1, Account account2){
        System.out.println("Do payment using Payment Gateway");
    }
}
```

В следующем интерфейсе и его реализации имеется новая, продвинутая функциональность платежной системы:

```
package com.packt.patterninspring.chapter3.adapter.pattern;
public interface AdvancedPayGateway {
    void makePayment(String mobile1, String mobile2);
}
```

Теперь создадим класс для реализации новой платежной системы:

```
package com.packt.patterninspring.chapter3.adapter.pattern;
import com.packt.patterninspring.chapter3.model.Account;
public class AdvancedPaymentGatewayAdapter implements
    AdvancedPayGateway{
    private PaymentGateway paymentGateway;
    public AdvancedPaymentGatewayAdapter(PaymentGateway
        paymentGateway) {
        this.paymentGateway = paymentGateway;
    }
    public void makePayment(String mobile1, String mobile2) {
        // получаем номер счета по номеру мобильного телефона
        Account account1 = null;
        Account account2 = null;
        paymentGateway.doPayment(account1, account2);
    }
}
```

Создадим класс для демонстрации работы паттерна:

```
package com.packt.patterninspring.chapter3.adapter.pattern;
public class AdapterPatternMain {
    public static void main(String[] args) {
        PaymentGateway paymentGateway = new PaymentGatewayImpl();
    }
}
```



```

    AdvancedPayGateway advancedPayGateway = new
        AdvancedPaymentGatewayAdapter(paymentGateway);
    String mobile1 = null;
    String mobile2 = null;
    advancedPayGateway.makePayment(mobile1, mobile2);
}
}

```

В предыдущем классе есть объект старой платежной системы — интерфейса `PaymentGateway`, но мы преобразуем эту реализацию старой системы в более продвинутый вариант с помощью класса-адаптера `AdvancedPaymentGatewayAdapter`. Запустим этот демонстрационный класс и получим в консоли следующий результат (рис. 3.3).

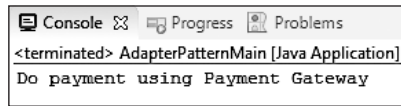


Рис. 3.3. Результат работы демонстрационного класса

Паттерн проектирования «Мост»

Расцепляет абстракцию с ее реализацией так, чтобы можно было менять их независимо друг от друга.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

Одна из наиболее популярных идей инженерии ПО — предпочтение композиции наследованию. Паттерн проектирования «Мост» поощряет использование этой популярной идеи. Подобно паттерну «Адаптер», он также относится к числу структурных паттернов проектирования из семейства паттернов GoF. Суть подхода паттерна «Мост» состоит в расцеплении используемой клиентским кодом абстракции с ее реализацией. Это значит, что он разделяет абстракцию и ее реализацию в отдельные иерархии классов. Кроме того, паттерн «Мост» предпочитает композицию наследованию, поскольку гибкость наследования невысока и оно нарушает инкапсуляцию, так что любые изменения в реализаторе повлияли бы на применяемую клиентским кодом абстракцию.

Мост предоставляет способ добиться при разработке программного обеспечения взаимодействия между двумя различными независимыми компонентами, а также способ расцепления абстрактного класса и класса-реализатора. Поэтому любые изменения в классе реализации или реализаторе (то есть интерфейсе) не повлияют ни на абстрактный класс, ни на класс расширенной абстракции. Это становится возможным благодаря композиции интерфейса и абстракции. Паттерн «Мост» использует интерфейс в качестве моста между конкретными классами абстрактного класса и классами реализации данного интерфейса. При этом можно вносить изменения в классы обоих типов без какого-либо влияния на клиентский код.

Преимущества использования

У использования паттерна проектирования «Мост» есть следующие положительные стороны:

- ❑ позволяет разделить реализацию и абстракцию;
- ❑ обеспечивает достаточную гибкость внесения изменений в классы обоих типов без какого-либо влияния на клиентский код;
- ❑ позволяет скрывать нюансы фактической реализации от клиента за счет наличия между ними абстракции.

Часто встречающиеся проблемы, решаемые с помощью паттерна «Мост»

Мост решает такие часто встречающиеся проблемы:

- ❑ устраняет стойкую привязку функциональной абстракции к ее реализации;
- ❑ позволяет вносить изменения в классы реализации без какого-либо влияния на абстракцию и клиентский код;
- ❑ дает возможность расширять абстракцию и ее реализацию с помощью подклассов.

Реализация паттерна во фреймворке Spring

Паттерн проектирования «Мост» лежит в основе такого модуля Spring, как `ViewRendererServlet`. Этот сервлет-мост предназначен в основном для поддержки фреймворка Portlet MVC.

Кроме того, паттерн «Мост» используется в процессе журналирования Spring.

Пример реализации

Допустим, вам нужно иметь возможность открывать в своей банковской системе два типа счетов: сберегательные (savings account) и текущие (current account).

Система без использования паттерна проектирования «Мост». Рассмотрим пример, в котором паттерн «Мост» не применяется. На следующей схеме показаны отношения между интерфейсами `Bank` и `Account` (рис. 3.4).

Создадим такую архитектуру. Сначала создадим интерфейс или абстрактный класс `Bank`, а затем производные от него классы: `IciciBank` и `HdfcBank`. Для открытия счета в банке нужно сначала выбрать тип класса счета — `Saving Account` или `Current Account`, которые расширяют вышеуказанные классы банков (`HdfcBank` и `IciciBank`). В этом приложении предусмотрена простая глубокая иерархия наследования. Что же с ней не так по сравнению с предыдущей схемой? Вы можете заметить: данная архитектура делится на две части — абстракцию и реализацию. Клиентский код взаимодействует с частью, относящейся к абстракции. При модификации части, относящейся к абстракции, клиентский код сможет обращаться только к новым изменениям или новым элементам функциональности части реализации, а значит, абстракция и реализация сильно сцеплены друг с другом.

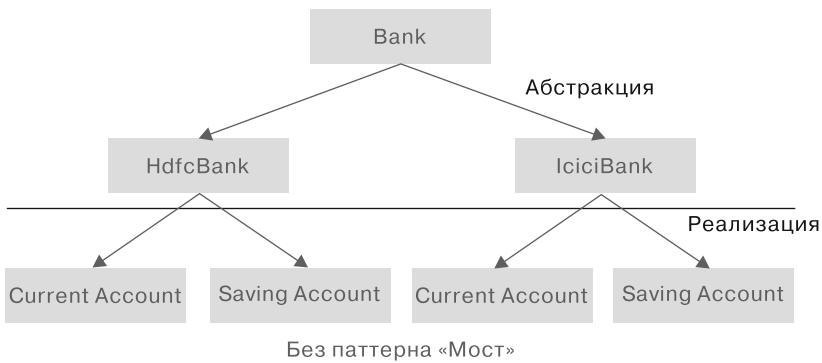


Рис. 3.4. Система без использования паттерна проектирования «Мост»

Теперь посмотрим, как можно усовершенствовать этот пример с помощью паттерна проектирования «Мост».

Система с паттерном проектирования «Мост». На следующей схеме (рис. 3.5) мы свяжем интерфейсы Bank и Account с помощью паттерна «Мост».

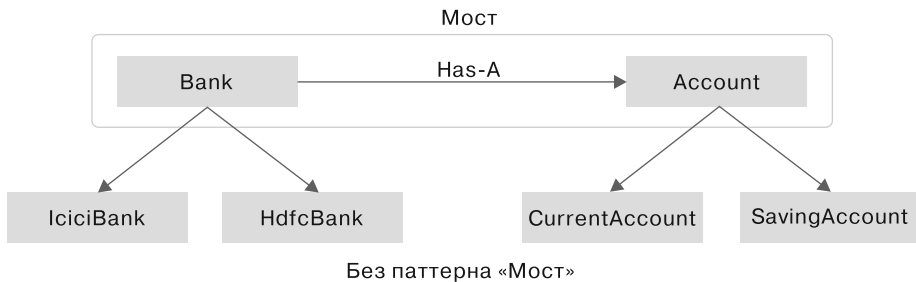


Рис. 3.5. Система с паттерном проектирования «Мост»

UML-структура для паттерна проектирования «Мост». На следующей схеме (рис. 3.6) видно, как мост решает показанные в вышеприведенном примере проблемы. Он разделяет абстракцию и реализацию на две иерархии классов.

Интерфейс Account служит реализатором моста, а конкретные классы SavingAccount и CurrentAccount реализуют интерфейс Account. Класс Bank является абстрактным, использующим объект типа Account.

Создадим интерфейс реализатора моста.

Файл Account.java:

```
package com.packt.patterninspring.chapter3.bridge.pattern;
public interface Account {
    Account openAccount();
    void accountType();
}
```

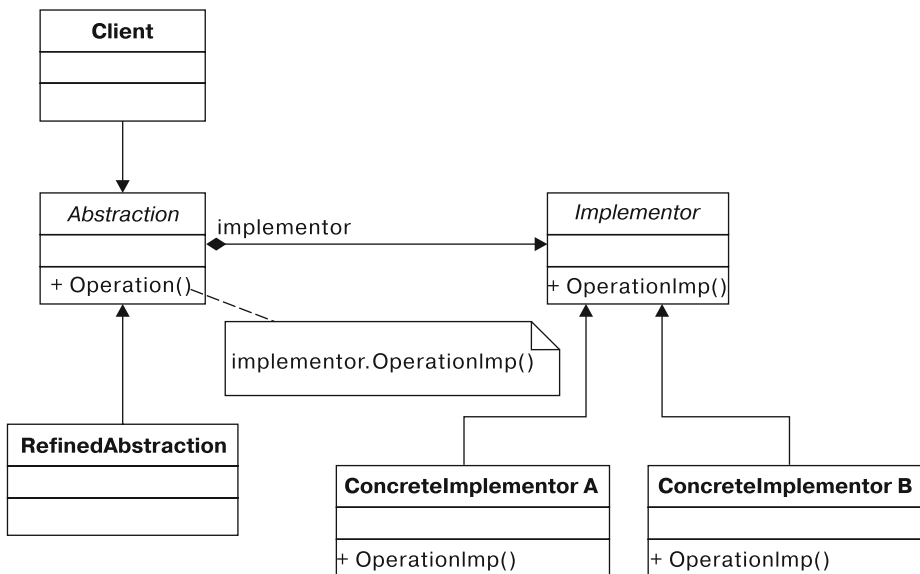


Рис. 3.6. UML-диаграмма паттерна проектирования «Мост»

Создадим конкретные классы, реализующие интерфейс Account. Сначала создадим класс SavingAccount.

Файл SavingAccount.java:

```

package com.packt.patterninspring.chapter3.bridge.pattern;
public class SavingAccount implements Account {
    @Override
    public Account openAccount() {
        System.out.println("OPENED: SAVING ACCOUNT ");
        return new SavingAccount();
    }
    @Override
    public void accountType() {
        System.out.println("##It is a SAVING Account##");
    }
}

```

Теперь создадим класс CurrentAccount, реализующий интерфейс Account.

Файл CurrentAccount.java:

```

package com.packt.patterninspring.chapter3.bridge.pattern;
public class CurrentAccount implements Account {
    @Override
    public Account openAccount() {
        System.out.println("OPENED: CURRENT ACCOUNT ");
        return new CurrentAccount();
    }
}

```

```

@Override
public void accountType() {
    System.out.println("##It is a CURRENT Account##");
}
}

```

Нам понадобится абстракция паттерна проектирования «Мост», но сначала создадим интерфейс Bank.

Файл Bank.java:

```

package com.packt.patterninspring.chapter3.bridge.pattern;
public abstract class Bank {
    // Композиция с реализатором
    protected Account account;
    public Bank(Account account){
        this.account = account;
    }
    abstract Account openAccount();
}

```

Реализуем первую абстракцию для интерфейса Bank; ниже представлен класс реализации для этого интерфейса.

Файл IciciBank.java:

```

package com.packt.patterninspring.chapter3.bridge.pattern;
public class IciciBank extends Bank {
    public IciciBank(Account account) {
        super(account);
    }
    @Override
    Account openAccount() {
        System.out.print("Open your account with ICICI Bank");
        return account;
    }
}

```

Реализуем вторую абстракцию для интерфейса Bank; ниже представлен класс реализации для этого интерфейса.

Файл HdfcBank.java:

```

package com.packt.patterninspring.chapter3.bridge.pattern;
public class HdfcBank extends Bank {
    public HdfcBank(Account account) {
        super(account);
    }
    @Override
    Account openAccount() {
        System.out.print("Open your account with HDFC Bank");
        return account;
    }
}

```

Создадим класс для демонстрации работы паттерна проектирования «Мост».

Файл BridgePatternMain.java:

```
package com.packt.patterninspring.chapter3.bridge.pattern;
public class BridgePatternMain {
    public static void main(String[] args) {
        Bank icici = new IciciBank(new CurrentAccount());
        Account current = icici.openAccount();
        current.accountType();
        Bank hdfc = new HdfcBank(new SavingAccount());
        Account saving = hdfc.openAccount();
        saving.accountType();
    }
}
```

Запустим этот демонстрационный класс и получим в консоли следующий результат (рис. 3.7).

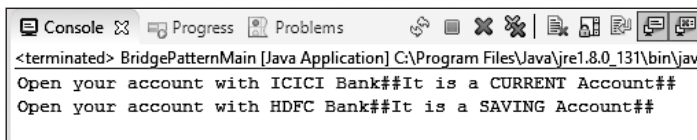


Рис. 3.7. Результат работы демонстрационного класса

Паттерн проектирования «Составной объект»

Объекты можно объединять в древовидные структуры, отражающие иерархии типа «часть — целое». Благодаря составному объекту клиенты могут работать одинаковым образом как с отдельными объектами, так и с их композициями.

GoF. Приемы объектно-ориентированного проектирования

Паттерн «Составной объект» (Composite) в инженерии разработки ПО относят к структурным паттернам проектирования. Его суть в том, что клиент взаимодействует с группой объектов одного типа как с единым объектом. Идея паттерна «Составной объект» состоит в композиции набора объектов в древовидную структуру, которая бы представляла модуль большего приложения. А клиенты рассматривают эту структуру как единое целое.

Объекты в системе группируются в древовидную структуру, представляющую собой сочетание узлов-листьев и веток. В ней узлы могут иметь листья и другие узлы, а у листьев нет никаких дочерних элементов. Листья рассматриваются как конечные точки структурированных в дерево данных.

Рассмотрим следующую схему, представляющую данные из древовидной структуры в виде узлов и листьев (рис. 3.8).

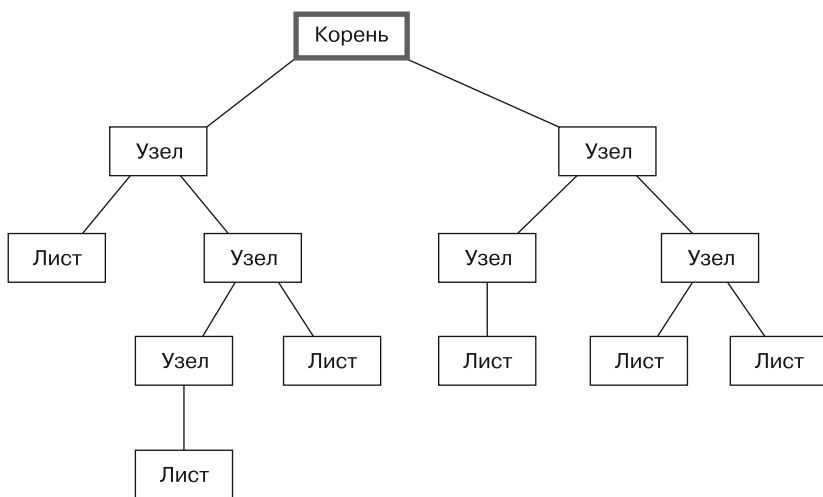


Рис. 3.8. Структурированные в виде дерева с узлами и листьями данные

Часто встречающиеся проблемы, решаемые с помощью паттерна «Составной объект»

С точки зрения разработчика, сложнее спроектировать приложение так, чтобы клиент мог обращаться к объектам в приложении одинаковым образом, независимо от того, являются они отдельными или входят в состав других объектов. Данный паттерн упрощает задачу и позволяет проектировать объекты так, чтобы можно было использовать их и как отдельные объекты, и как композицию других объектов.

Паттерн позволяет решать трудные проблемы, которые возникают при создании иерархических древовидных структур и связаны с предоставлением клиентам единообразного способа обращения к объектам в этом дереве и манипуляции ими. В подобной ситуации проще всего рассматривать простые и составные объекты как однотипные, и для этого отлично подходит паттерн «Составной объект».

UML-структура паттерна

Паттерн основан на композиции объектов схожих типов в древовидную структуру. Как вы знаете, у каждого дерева есть три основные составляющие: ветви, узлы и листья. Обсудим используемую в этом паттерне терминологию.

- ❑ **Component (Компонент).** Фактически ветвь дерева, у которой могут быть другие ветви, узлы и листья. **Component** — абстракция для всех компонентов, включая составные объекты. В паттерне «Составной объект» компонент, по сути, объявляет интерфейс для объектов.
- ❑ **Leaf (Лист).** Объект, реализующий все методы компонента.

- ❑ **Composite** (Составной объект). В древовидной структуре представляется узлом, у которого могут быть другие узлы и листья. Определяет составной компонент. Обладает методами для добавления потомков, то есть представляет собой коллекцию объектов одного типа. Есть в нем и другие методы для работы с потомками.

Рассмотрим следующую UML-схему для этого паттерна проектирования (рис. 3.9).

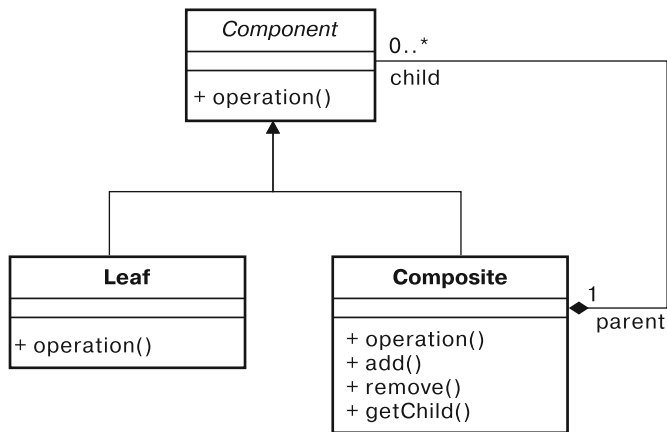


Рис. 3.9. UML-диаграмма для паттерна проектирования «Составной объект»

Преимущества использования паттерна «Составной объект»

- ❑ Облегчает добавление новых видов компонентов, избавляя от необходимости вносить в клиент изменения.
- ❑ Позволяет создавать иерархию классов, включающую как отдельные, так и составные объекты.

Пример реализации

В следующем примере я реализую интерфейс **Account**, который может как относиться к одному из типов **SavingAccount** или **CurrentAccount**, так и представлять собой композицию нескольких счетов. У нас есть класс **CompositeBankAccount**, играющий роль класса-актера паттерна «Составной объект». Рассмотрим следующий код этого примера.

Создаем интерфейс **Account**, который будет играть роль компонента:

```
public interface Account {
    void accountType();
}
```

Создаем классы **SavingAccount** и **CurrentAccount** — реализации компонента, играющие роль листа нашего дерева.

Файл SavingAccount.java:

```
public class SavingAccount implements Account{
    @Override
    public void accountType() {
        System.out.println("SAVING ACCOUNT");
    }
}
```

Файл CurrentAccount.java:

```
public class CurrentAccount implements Account {
    @Override
    public void accountType() {
        System.out.println("CURRENT ACCOUNT");
    }
}
```

Создаем реализующий интерфейс Account класс CompositeBankAccount, который будет играть роль составного объекта.

Файл CompositeBankAccount.java:

```
package com.packt.patterninspring.chapter3.composite.pattern;
import java.util.ArrayList;
import java.util.List;
import com.packt.patterninspring.chapter3.model.Account;
public class CompositeBankAccount implements Account {
    // Коллекция счетов-потомков
    private List<Account> childAccounts = new ArrayList<Account>();
    @Override
    public void accountType() {
        for (Account account : childAccounts) {
            account.accountType();
        }
    }
    // Добавляет счет в композицию
    public void add(Account account) {
        childAccounts.add(account);
    }
    // Удаляет счет из композиции
    public void remove(Account account) {
        childAccounts.remove(account);
    }
}
```

Создаем класс CompositePatternMain, который будет играть роль клиента.

Файл CompositePatternMain.java:

```
package com.packt.patterninspring.chapter3.composite.pattern;
import com.packt.patterninspring.chapter3.model.CurrentAccount;
import com.packt.patterninspring.chapter3.model.SavingAccount;
public class CompositePatternMain {
    public static void main(String[] args) {
```

```

// Сберегательные счета
SavingAccount savingAccount1 = new SavingAccount();
SavingAccount savingAccount2 = new SavingAccount();
// Текущие счета
CurrentAccount currentAccount1 = new CurrentAccount();
CurrentAccount currentAccount2 = new CurrentAccount();
// Составной банковский счет
CompositeBankAccount compositeBankAccount1 = new
CompositeBankAccount();
CompositeBankAccount compositeBankAccount2 = new
CompositeBankAccount();
CompositeBankAccount compositeBankAccount = new
CompositeBankAccount();
// Выполняем композицию банковских счетов
compositeBankAccount1.add(savingAccount1);
compositeBankAccount1.add(currentAccount1);
compositeBankAccount2.add(currentAccount2);
compositeBankAccount2.add(savingAccount2);
compositeBankAccount.add(compositeBankAccount2);
compositeBankAccount.add(compositeBankAccount1);
compositeBankAccount.accountType();
}
}

```

Запустим этот демонстрационный класс и получим в консоли следующий результат (рис. 3.10).

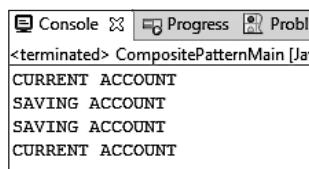


Рис. 3.10. Результат работы демонстрационного класса

Паттерн проектирования «Декоратор»

Динамически назначает объекту новые обязанности. Гибкий альтернативный вариант расширения функциональности по сравнению с созданием производных классов.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

В инженерии разработки ПО общая задача всех структурных паттернов проектирования GoF заключается в упрощении сложных взаимосвязей между объектами и классами в гибких корпоративных приложениях. Паттерн «Декоратор» (Decorator) представляет особую разновидность структурного паттерна проекти-

рования и позволяет динамически/статически добавлять и исключать поведение отдельных объектов, не изменяя имеющееся поведение других связанных объектов того же класса. При этом не нарушаются принципы единственной обязанности и SOLID¹ объектно-ориентированного программирования.

Для задания ассоциаций объектов в этом паттерне оказывается предпочтение композиции, а не наследованию. Он дает возможность разделять функциональность на различные конкретные классы с непересекающимися областями ответственности.

Паттерн известен также под названием «Обертка» (Wrapper).

Преимущества использования паттерна проектирования «Декоратор»:

- ❑ позволяет динамически и статически расширять функциональность, не меняя структуры существующих объектов;
- ❑ с помощью этого паттерна можно динамически назначить объекту новую обязанность;
- ❑ чтобы обеспечить соблюдение принципов SOLID, он использует композиции взаимосвязей объектов;
- ❑ упрощает написание кода благодаря созданию новых классов для каждого элемента функциональности вместо изменения существующего кода приложения.

Часто встречающиеся проблемы, решаемые с помощью паттерна «Декоратор»

К корпоративному приложению могут выдвигаться бизнес-требования, или для него могут быть планы на будущее по расширению поведения за счет добавления новой функциональности. Для расширения поведения объекта можно воспользоваться наследованием. Но его нужно проводить во время компиляции, да и методы оказываются доступны для других объектов того же класса. А из-за модификации кода нарушается принцип открытости/закрытости. Чтобы избежать этого нарушения одного из принципов SOLID, можно назначать объектам новые обязанности динамически. Именно для этого служит декоратор, позволяющий весьма гибко решить данную задачу. Рассмотрим образец реализации паттерна на реальном примере из практики.

Допустим, что банк предлагает вкладчикам несколько видов счетов с различными бонусами. Вкладчики делятся на три категории: пенсионеры, VIP-вкладчики и молодежь. Банк запускает акцию по сберегательным счетам для пенсионеров: при открытии сберегательного счета в этом банке им бесплатно предоставляется медицинская страховка на сумму до \$1000. Аналогично банк делает акционное предложение и VIP-вкладчикам: страхование от несчастных случаев на сумму до \$1600 и возможность овердрафта до \$84. Акции для молодежи нет.

¹ Мнемонический акроним для основных принципов объектно-ориентированного программирования и проектирования: принципа единственной ответственности, принципа открытости/закрытости, принципа подстановки Барбары Лисков, принципа разделения интерфейсов, принципа инверсии зависимостей. — *Здесь и далее примеч. пер.*

Чтобы учесть эти требования, можно создать новые подклассы класса `SavingAccount` — каждый из них в качестве декорации представляет свой тип сберегательно-го счета с дополнительными бонусами, в результате чего наша архитектура теперь выглядит следующим образом (рис. 3.11).

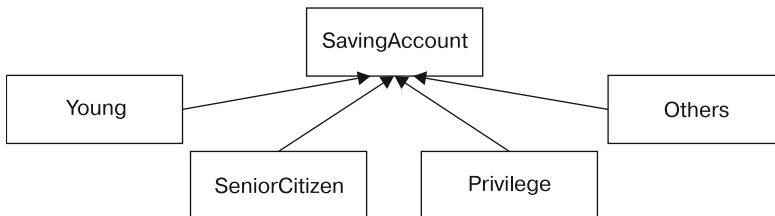


Рис. 3.11. Архитектура приложения с наследованием без паттерна проектирования «Декоратор»

Такая архитектура окажется и без того очень запутанной при добавлении новых акционных схем в `SavingAccount`, но что будет, когда банк запустит аналогичные схемы для `CurrentAccount`? Она очевидно несовершенна, однако это идеальный сценарий для использования паттерна «Декоратор». Он позволяет динамически добавлять поведение во время выполнения. В этом случае для реализации `Account` я создам абстрактный класс `AccountDecorator`. Более того, я создам классы `SeniorCitizen` и `Privilege`, расширяющие класс `AccountDecorator` (поскольку у молодежи бонусов нет, то нет и соответствующего класса, расширяющего `AccountDecorator`). Архитектура выглядит следующим образом (рис. 3.12).

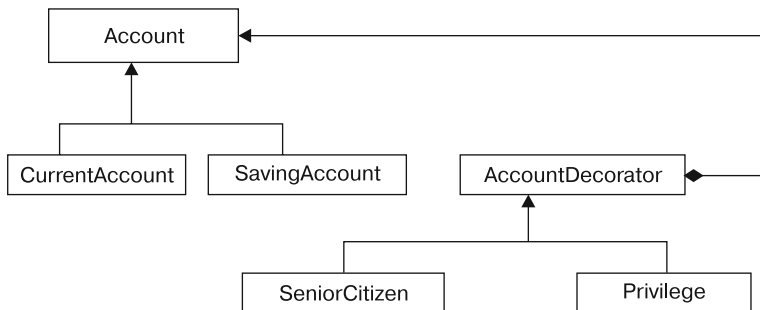


Рис. 3.12. Архитектура приложения в случае композиции с помощью паттерна проектирования «Декоратор»

Предыдущая схема следует паттерну проектирования «Декоратор», в котором `AccountDecorator` играет роль декоратора. В ней отражены важные нюансы отношения между `Account` и `AccountDecorator`. Это отношение носит следующий характер:

- ❑ отношение типа «является» (*is-a*) между `AccountDecorator` и `Account`, то есть наследование для соответствующего типа;

- ❑ отношение принадлежности (*has-a*) AccountDecorator и Account, то есть композиция в целях добавления нового поведения без изменения существующего кода.

На рис. 3.13 представлена UML-структура паттерна «Декоратор».

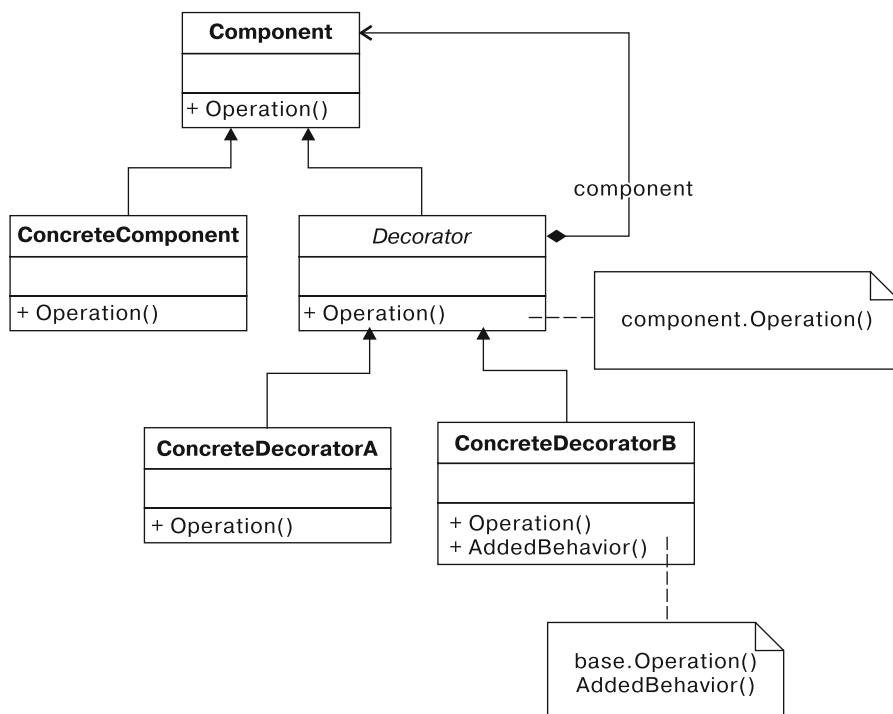


Рис. 3.13. UML-диаграмма паттерна проектирования «Декоратор»

В этом паттерне участвуют следующие классы и объекты.

- ❑ **Component** (**Account**) (**Компонент**). Интерфейс для объектов, которым могут динамически назначаться новые обязанности.
- ❑ **ConcreteComponent** (**SavingAccount**) (**Конкретный компонент**). Соответствующий интерфейсу компонента конкретный класс, задающий объект, которому могут быть динамически назначены новые обязанности.
- ❑ **Decorator** (**AccountDecorator**) (**Декоратор**). Содержит ссылку на объект **Component** и задает интерфейс, соответствующий интерфейсу компонента.
- ❑ **ConcreteDecorator** (**SeniorCitizen** и **Privilege**) (**Конкретный декоратор**). Конкретная реализация **Decorator**, которая назначает компоненту новые обязанности.

Реализация паттерна

Создадим класс компонента.

Файл `Account.java`:

```
package com.packt.patterninspring.chapter3.decorator.pattern;
public interface Account {
    String getTotalBenefits();
}
```

Создадим классы конкретных компонентов.

Файл `SavingAccount.java`:

```
package com.packt.patterninspring.chapter3.decorator.pattern;
public class SavingAccount implements Account {
    @Override
    public String getTotalBenefits() {
        return "This account has 4% interest rate with per day
        $5000 withdrawal limit";
    }
}
```

Еще один конкретный класс для компонента `Account`.

Файл `CurrentAccount.java`:

```
package com.packt.patterninspring.chapter3.decorator.pattern;
public class CurrentAccount implements Account {
    @Override
    public String getTotalBenefits() {
        return "There is no withdrawal limit for current account";
    }
}
```

А теперь создадим класс `Decorator` для компонента `Account`, который назначает классам компонента `Account` другое поведение во время выполнения.

Файл `AccountDecorator.java`:

```
package com.packt.patterninspring.chapter3.decorator.pattern;
public abstract class AccountDecorator implements Account {
    abstract String applyOtherBenefits();
}
```

Создадим класс `ConcreteDecorator`, реализующий класс `AccountDecorator`. Нижеприведенный класс `SeniorCitizen` расширяет `AccountDecorator`, чтобы иметь возможность использовать другое поведение во время выполнения, например `applyOtherBenefits()`.

Файл `SeniorCitizen.java`:

```
package com.packt.patterninspring.chapter3.decorator.pattern;
public class SeniorCitizen extends AccountDecorator {
    Account account;
    public SeniorCitizen(Account account) {
        super();
    }
}
```

```

        this.account = account;
    }
    public String getTotalBenefits() {
        return account.getTotalBenefits() + " other benefits are
            "+applyOtherBenefits();
    }
    String applyOtherBenefits() {
        return " an medical insurance of up to $1,000 for Senior
            Citizen";
    }
}

```

Создадим еще один класс `ConcreteDecorator`, реализующий класс `AccountDecorator`. Нижеприведенный класс `Privilege` расширяет `AccountDecorator`, чтобы иметь возможность использовать другое поведение во время выполнения, например `applyOtherBenefits()`.

Файл `Privilege.java`:

```

package com.packt.patterninspring.chapter3.decorator.pattern;
public class Privilege extends AccountDecorator {
    Account account;
    public Privilege(Account account) {
        this.account = account;
    }
    public String getTotalBenefits() {
        return account.getTotalBenefits() + " other benefits are
            "+applyOtherBenefits();
    }
    String applyOtherBenefits() {
        return " an accident insurance of up to $1,600 and
            an overdraft facility of $84";
    }
}

```

Теперь напишем тестовый код, чтобы посмотреть на работу паттерна «Декоратор» во время выполнения программы.

Файл `DecoratorPatternMain.java`:

```

package com.packt.patterninspring.chapter3.decorator.pattern;
public class DecoratorPatternMain {
    public static void main(String[] args) {
        /* Сберегательный счет без декорации */
        Account basicSavingAccount = new SavingAccount();
        System.out.println(basicSavingAccount.getTotalBenefits());
        /* Сберегательный счет с декорацией в виде акционной схемы
            для пенсионеров */
        Account seniorCitizenSavingAccount = new SavingAccount();
        seniorCitizenSavingAccount = new
            SeniorCitizen(seniorCitizenSavingAccount);
        System.out.println
            (seniorCitizenSavingAccount.getTotalBenefits());
        /* Сберегательный счет с декорацией в виде акционной схемы
            для VIP-вкладчиков */
    }
}

```

```

    Account privilegeCitizenSavingAccount = new SavingAccount();
    privilegeCitizenSavingAccount = new
        PrivilegeCitizenSavingAccount();
    System.out.println
        (privilegeCitizenSavingAccount.getTotalBenefits());
}
}

```

Запускаем этот демонстрационный класс и получаем в консоли следующие результаты (рис. 3.14).

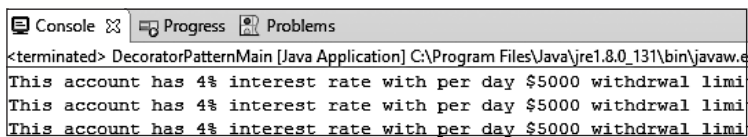


Рис. 3.14. Результат работы демонстрационного класса

Паттерн «Декоратор» во фреймворке Spring

В Spring паттерн проектирования «Декоратор» используется в качестве основы таких важных элементов функциональности, как транзакции, синхронизация кэша, а также для связанных с безопасностью задач. Рассмотрим отдельные элементы функциональности, в которых фреймворк реализует декоратор прозрачным образом.

- ❑ Вплетение совета в приложение Spring. Фреймворк при этом использует паттерн «Декоратор» с помощью CGLIB-прокси. Паттерн в этот момент работает путем генерации подкласса целевого класса во время выполнения.
- ❑ `BeanDefinitionDecorator`. Служит для декорирования описания компонента с помощью пользовательских атрибутов.
- ❑ `WebSocketHandlerDecorator`. Используется для добавления дополнительных видов поведения в экземпляр `WebSocketHandler`.

Паттерн проектирования «Фасад»

Предоставляет единый интерфейс для набора интерфейсов подсистемы. Фасад задает высокоуровневый интерфейс, облегчающий использование подсистемы.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

Паттерн проектирования «Фасад» (Facade), по сути, просто интерфейс интерфейсов, предназначенный для упрощения взаимодействия клиентского кода и классов подсистемы. Он относится к структурным паттернам проектирования.

Преимущества использования паттерна «Фасад»:

- ❑ упрощает для клиентов взаимодействие с подсистемами;
- ❑ делает более понятными все бизнес-сервисы, объединяя их в единые интерфейсы;
- ❑ снижает количество зависимостей клиентского кода от внутренних механизмов системы.

Когда использовать паттерн «Фасад»

Допустим, вы проектируете систему, состоящую из огромного множества независимых классов и набора сервисов, которые нужно реализовать. Эта система будет очень сложной, так что вам пригодится паттерн «Фасад», с помощью которого можно упростить систему и взаимодействие клиентского кода с классами подсистем большой сложной системы.

Предположим, вы хотите разработать банковское корпоративное приложение с большим количеством сервисов для выполнения определенных задач, например `AccountService` для получения объекта `Account` по `accountId`, `PaymentService` для сервисов платежных систем и `TransferService` для перевода средств с одного счета на другой. Клиентский код взаимодействует со всеми этими сервисами во время перевода средств между счетами. Именно так различные клиенты взаимодействуют в процессе перевода средств банковской системы. Как показано на следующей схеме (рис. 3.15), клиентский код непосредственно взаимодействует с классами подсистемы, и вдобавок клиент осведомлен о внутреннем устройстве классов подсистемы, что является прямым нарушением принципов проектирования SOLID, поскольку клиентский код сильно связан с классами подсистемы банковского приложения.

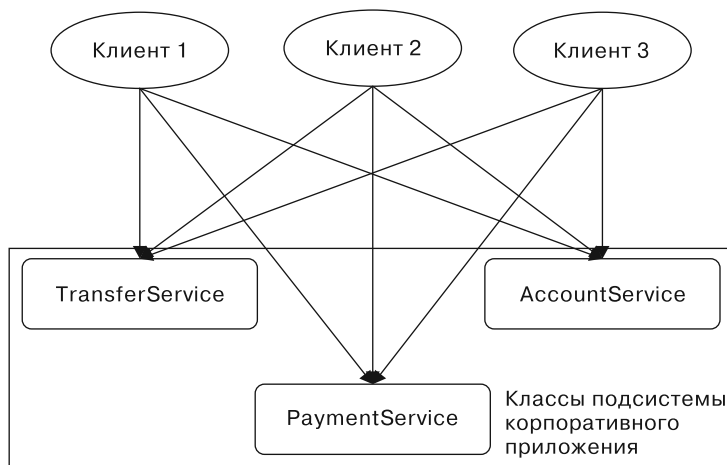


Рис. 3.15. Подсистема банковского приложения без паттерна проектирования «Фасад»

Вместо непосредственного взаимодействия клиентского кода с классами подсистемы лучше было бы ввести в архитектуру еще один интерфейс, упрощающий использование подсистем, как показано на следующей схеме (рис. 3.16). Этот интерфейс называется *фасадом* и основывается на одноименном паттерне. Он представляет собой простой способ взаимодействия с подсистемами.

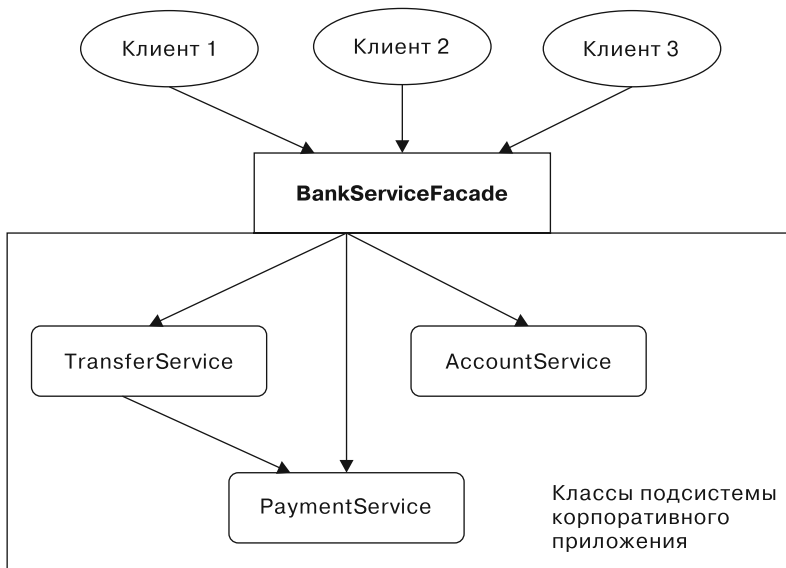


Рис. 3.16. Подсистема банковского приложения с паттерном проектирования «Фасад»

Реализация паттерна

Создадим классы сервисов для подсистемы нашего банковского приложения. Начнем с класса `PaymentService`.

Файл `PaymentService.java`:

```
package com.packt.patterninspring.chapter3.facade.pattern;
public class PaymentService {
    public static boolean doPayment(){
        return true;
    }
}
```

Создадим класс еще одного сервиса для подсистемы — `AccountService`.

Файл `AccountService.java`:

```
package com.packt.patterninspring.chapter3.facade.pattern;
import com.packt.patterninspring.chapter3.model.Account;
import com.packt.patterninspring.chapter3.model.SavingAccount;
public class AccountService {
    public static Account getAccount(String accountId) {
```

```

    return new SavingAccount();
}
}

```

Создадим еще один класс сервиса для подсистемы — `TransferService`.

Файл `TransferService.java`:

```

package com.packt.patterninspring.chapter3.facade.pattern;
import com.packt.patterninspring.chapter3.model.Account;
public class TransferService {
    public static void transfer(int amount, Account fromAccount,
        Account toAccount) {
        System.out.println("Transferring Money");
    }
}

```

Создадим класс фасадного сервиса для взаимодействия с подсистемой. Рассмотрим следующий интерфейс-фасад для подсистемы, после чего реализуем его в приложении в качестве глобального банковского сервиса.

Файл `BankingServiceFacade.java`:

```

package com.packt.patterninspring.chapter3.facade.pattern;
public interface BankingServiceFacade {
    void moneyTransfer();
}

```

Файл `BankingServiceFacadeImpl.java`:

```

package com.packt.patterninspring.chapter3.facade.pattern;
import com.packt.patterninspring.chapter3.model.Account;
public class BankingServiceFacadeImpl implements
    BankingServiceFacade{
    @Override
    public void moneyTransfer() {
        if(PaymentService.doPayment()){
            Account fromAccount = AccountService.getAccount("1");
            Account toAccount = AccountService.getAccount("2");
            TransferService.transfer(1000, fromAccount, toAccount);
        }
    }
}

```

Создаем клиент для фасада.

Файл `FacadePatternClient.java`:

```

package com.packt.patterninspring.chapter3.facade.pattern;
public class FacadePatternClient {
    public static void main(String[] args) {
        BankingServiceFacade serviceFacade = new
            BankingServiceFacadeImpl();
        serviceFacade.moneyTransfer();
    }
}

```

UML-структура паттерна

В данном паттерне участвуют следующие классы и объекты.

- ❑ **Фасад (BankingServiceFacade)**. Это интерфейс-фасад, знающий, какие классы подсистемы отвечают за обработку запроса. Данный интерфейс отвечает за делегирование запросов клиентов соответствующим объектам подсистемы.
- ❑ **Классы подсистемы (AccountService, TransferService, PaymentService)**. Эти интерфейсы фактически представляют собой элементы функциональности подсистемы приложения для банковской системы. Они отвечают за обработку запросов, распределяемых объектом-фасадом. Ни в одном из интерфейсов данной категории нет ссылки на объект-фасад, они не знают никаких подробностей реализации фасада и полностью независимы от объектов-фасадом.

На рис. 3.17 представлена UML-диаграмма для паттерна «Фасад».

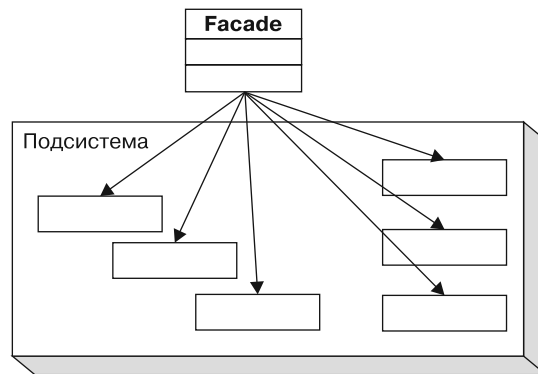


Рис. 3.17. UML-диаграмма для паттерна проектирования «Фасад»

Паттерн «Фасад» во фреймворке Spring

В корпоративном приложении при работе с приложениями Spring паттерн «Фасад» часто используется на уровне бизнес-сервисов программы для объединения всех сервисов. Можно также применять его и к объектам DAO на уровне сохранения.

Паттерн проектирования «Заместитель»

Предоставляет суррогат (заместитель) объекта, позволяющий управлять доступом к этому объекту.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

Паттерн проектирования «Заместитель» («Прокси», Proxy) позволяет получить объект класса, обладающий функциональностью другого класса. Он относится к структурным паттернам проектирования GoF. Цель заместителя — предоставить окружающему миру класс-заместитель другого класса, со всей его функциональностью.

Цели паттерна

- ❑ Скрытие реального объекта от окружающего мира.
- ❑ Повышение производительности за счет создания объекта по запросу.

UML-структура паттерна

На рис. 3.18 представлена UML-диаграмма данного паттерна.

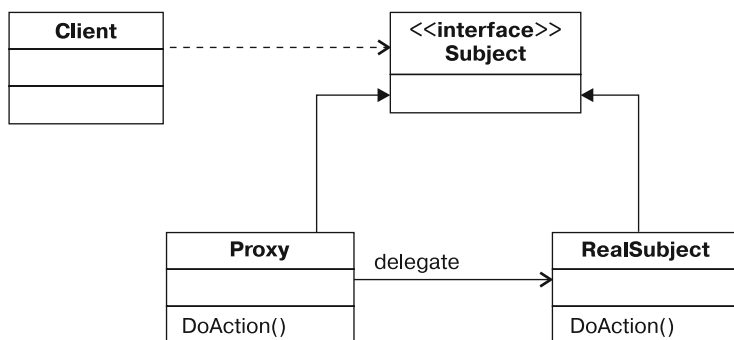


Рис. 3.18. UML-диаграмма паттерна проектирования «Заместитель»

Рассмотрим различные компоненты этой UML-диаграммы.

- ❑ **Subject** (Субъект). Общий интерфейс, реализуемый классами **Proxy** и **RealSubject**.
- ❑ **RealSubject** (Реальный субъект). Фактическая реализация **Subject** — представляемый заместителем реальный объект.
- ❑ **Proxy** (Прокси). Объект-заместитель и одновременно реализация **Subject** реального объекта. В нем хранятся ссылки на реальный объект.

Реализация паттерна

Следующий код иллюстрирует использование паттерна проектирования «Заместитель».

Создадим компонент **Subject**.

Файл **Account.java**:

```
public interface Account {
    void accountType();
}
```

Создадим следующий, реализующий **Subject**, класс **RealSubject** для паттерна проектирования «Заместитель».

Файл **SavingAccount.java**:

```
public class SavingAccount implements Account{
    public void accountType() {
        System.out.println("SAVING ACCOUNT");
    }
}
```

Создадим класс Proxy, реализующий Subject и содержащий RealSubject.

Файл ProxySavingAccount.java:

```
package com.packt.patterninspring.chapter2.proxy.pattern;
import com.packt.patterninspring.chapter2.model.Account;
import com.packt.patterninspring.chapter2.model.SavingAccount;
public class ProxySavingAccount implements Account{
    private Account savingAccount;
    public void accountType() {
        if(savingAccount == null){
            savingAccount = new SavingAccount();
        }
        savingAccount.accountType();
    }
}
```

Паттерн «Заместитель» во фреймворке Spring

Фреймворк Spring прозрачным образом использует паттерн проектирования «Заместитель» в модуле Spring AOP, как мы обсуждали в главе 1. В Spring AOP заместители объектов создаются в целях применения сквозной функциональности к срезам приложения Spring. Другие модули фреймворка также реализуют паттерн «Заместитель», в частности RMI, механизм вызова через протокол HTTP фреймворка Spring, Hessian и Burlap.

Поведенческие паттерны проектирования

Цель поведенческих паттернов проектирования состоит в организации взаимодействия и совместных действий группы объектов для выполнения задачи, которую ни один из этих объектов самостоятельно выполнить не может. Взаимодействие между объектами не должно выходить за рамки слабого сцепления. Паттерны из этой категории описывают способы взаимодействия и распределения обязанностей между классами или объектами. Далее мы рассмотрим различные виды поведенческих паттернов проектирования.

Паттерн проектирования «Цепочка обязанностей»

Позволяет избежать сцепления отправителя запроса с получателем за счет предоставления возможности обработки этого запроса более чем одному объекту. Связывает цепочкой получающие запрос объекты, передавая его по цепочке вплоть до объекта, которому поручается обработка.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

Паттерн проектирования «Цепочка обязанностей» (Chain of Responsibility) относится к поведенческим паттернам семейства GoF. В соответствии с ним отправитель и получатель запроса должны быть расцеплены. Отправитель посылает запрос

цепочке получателей, каждый из которых может оказаться его обработчиком. В этом паттерне у объекта-получателя есть ссылка на другой объект-получатель, и если активный объект не может обработать запрос, то передает тот же запрос следующему в цепочке объекту-получателю.

Например, в банковской системе деньги можно снимать в любом банкомате в любом месте. Это один из реальных примеров паттерна проектирования «Цепочка обязанностей».

Среди преимуществ использования этого паттерна:

- ❑ снижение сцепления в системе между объектом-отправителем и объектом-получателем при обработке запроса;
- ❑ гибкость распределения обязанностей по обработке запроса между объектами;
- ❑ создание цепочки объектов с помощью композиции, которая функционирует как единое целое.

Рассмотрим следующую UML-диаграмму, на которой показаны все компоненты паттерна проектирования «Цепочка обязанностей» (рис. 3.19).

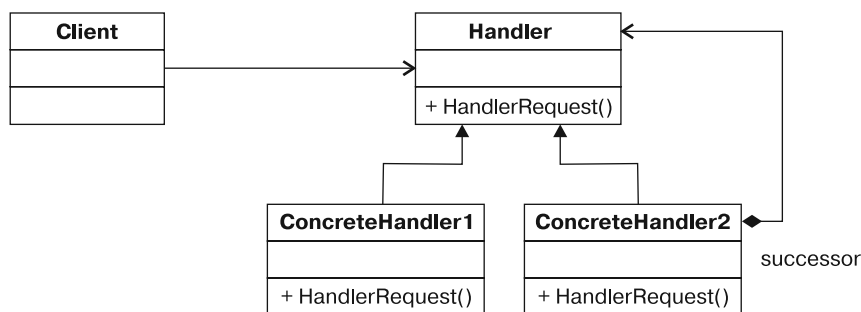


Рис. 3.19. UML-диаграмма паттерна проектирования «Цепочка обязанностей»

- ❑ **Handler** (Обработчик). Абстрактный класс или интерфейс, предназначенный для обработки запроса.
- ❑ **ConcreteHandler** (Конкретный обработчик). Конкретные классы, реализующие **Handler** для обработки запроса или передачи неизменного запроса следующему элементу в цепочке обработчиков.
- ❑ **Client** (Клиент). Основной класс приложения, инициирующий отправку запроса цепочке объектов-обработчиков.

Паттерн «Цепочка обязанностей» во фреймворке Spring. Паттерн проектирования «Цепочка обязанностей» реализован в проекте Spring Security из фреймворка Spring. Spring Security позволяет реализовывать функциональность аутентификации/авторизации с помощью цепочек фильтров безопасности. Возможности настройки этого фреймворка чрезвычайно широки: благодаря паттерну «Цепочка обязанностей» можно добавлять в эту цепочку фильтров свои пользовательские фильтры, чтобы адаптировать функциональность к своим задачам.

Паттерн проектирования «Команда»

Инкапсулирует запрос в объекте, позволяя, таким образом, задавать параметры клиентов для обработки соответствующих запросов, организовывать очереди и журналирование запросов, а также поддержку допускающих отмену операций.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

«Команда» (Command) относится к числу поведенческих паттернов из семейства GoF и представляет собой очень простой, ориентированный на работу с данными паттерн. Он позволяет инкапсулировать данные запроса в объекте с последующей передачей этого объекта в качестве команды методу-инициатору, который затем возвращает результат выполнения этой команды в виде другого объекта изначальной вызывающей стороне.

Вот некоторые преимущества использования паттерна «Команда»:

- ❑ дает возможность перемещать данные между компонентами системы (отправителем и получателем) в виде объекта;
- ❑ позволяет задавать параметры объектов в соответствии с выполняемым действием;
- ❑ дает возможность легко добавлять новые команды в систему без изменения существующих классов.

Рассмотрим следующую UML-диаграмму, на которой показаны все компоненты паттерна проектирования «Команда» (рис. 3.20).

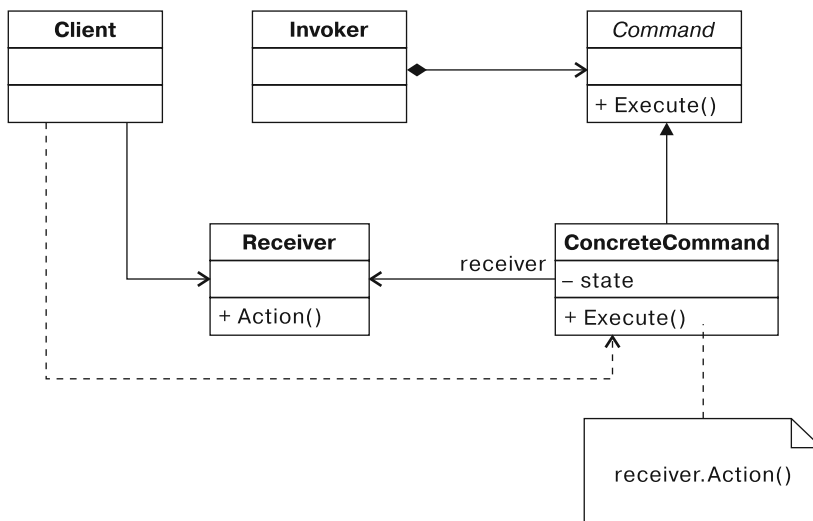


Рис. 3.20. UML-диаграмма для паттерна проектирования «Команда»

- ❑ **Command** (Команда). Интерфейс или абстрактный класс, описывающий выполняемое в системе действие.
- ❑ **ConcreteCommand** (Конкретная команда). Конкретная реализация интерфейса **Command**, определяющая выполняемое действие.
- ❑ **Client** (Клиент). Основной класс, создающий объект **ConcreteCommand** и задающий его получателя.
- ❑ **Invoker** (Инициатор). Объект, обращающийся к объекту команды в целях выполнения запроса.
- ❑ **Receiver** (Получатель). Простой метод-обработчик, выполняющий фактическую операцию с помощью объекта **ConcreteCommand**.

Паттерн «Команда» во фреймворке Spring. В Spring паттерн «Команда» реализован в модуле MVC. Вы часто будете наблюдать применение принципов паттерна «Команда» при использовании объектов **Command** в своих корпоративных приложениях Spring.

Паттерн проектирования «Интерпретатор»

Определяет представление грамматики заданного языка, а также, на основе этого представления, интерпретатор предложений этого языка.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

Паттерн проектирования «Интерпретатор» (Interpreter) позволяет анализировать элементы языка выражений для описания представления его грамматики. Относится к числу поведенческих паттернов проектирования GoF.

К числу преимуществ использования паттерна «Команда» можно отнести:

- ❑ возможность с легкостью менять и расширять грамматику;
- ❑ легкость и удобство использования языка выражений.

Рассмотрим следующую UML-диаграмму, на которой показаны все компоненты паттерна проектирования «Интерпретатор» (рис. 3.21).

- ❑ **AbstractExpression** (Абстрактное выражение). Интерфейс для выполнения задачи с помощью абстрактной операции **interpret()**.
- ❑ **TerminalExpression** (Терминальное выражение). Реализация вышеупомянутого интерфейса, в том числе операции **interpret()** для терминальных выражений.
- ❑ **NonterminalExpression** (Нетерминальное выражение). Еще одна реализация вышеупомянутого интерфейса, в том числе операции **interpret()** для нетерминальных выражений.
- ❑ **Context** (Контекст). Строковое (**String**) выражение, содержащее информацию, глобальную по отношению к интерпретатору.
- ❑ **Client** (Клиент). Основной класс, вызывающий операцию **interpret()**.

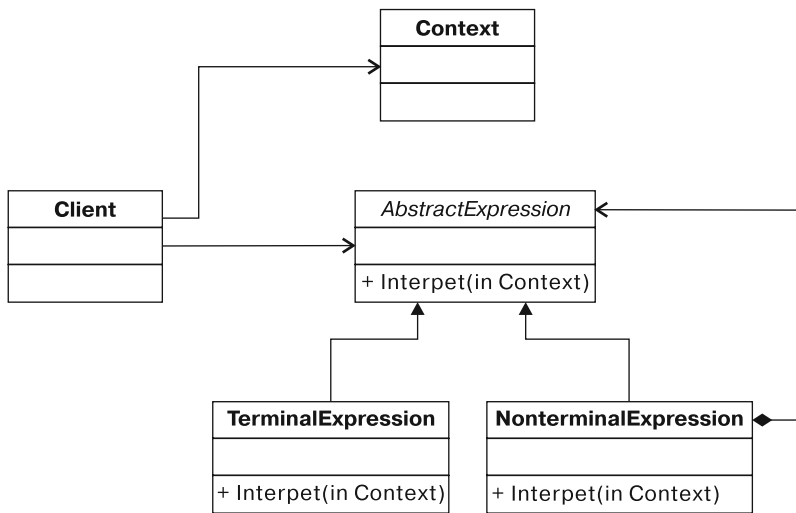


Рис. 3.21. UML-диаграмма для паттерна проектирования «Интерпретатор»

Паттерн «Интерпретатор» во фреймворке Spring. В фреймворке Spring данный паттерн применяется вместе с языком выражений Spring (SpEL). Эта новая возможность была добавлена в Spring 3.0, вы можете использовать ее в своих корпоративных приложениях Spring.

Паттерн проектирования «Итератор»

Обеспечивает способ последовательного обращения к элементам составного объекта без раскрытия его внутреннего представления.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

«Итератор» относится к числу поведенческих паттернов семейства GoF и очень часто используется в таких языках программирования, как Java. С его помощью становится возможным последовательное обращение к элементам составного объекта без информации о его внутреннем представлении.

Среди преимуществ использования паттерна «Итератор»:

- ❑ удобный доступ к элементам коллекции;
- ❑ доступ к элементу коллекции возможен несколькими способами, поскольку паттерн поддерживает множество вариантов обхода;
- ❑ единообразный интерфейс обхода различных структур в коллекции.

Рассмотрим следующую UML-диаграмму, на которой показаны все компоненты паттерна проектирования «Итератор» (рис. 3.22).

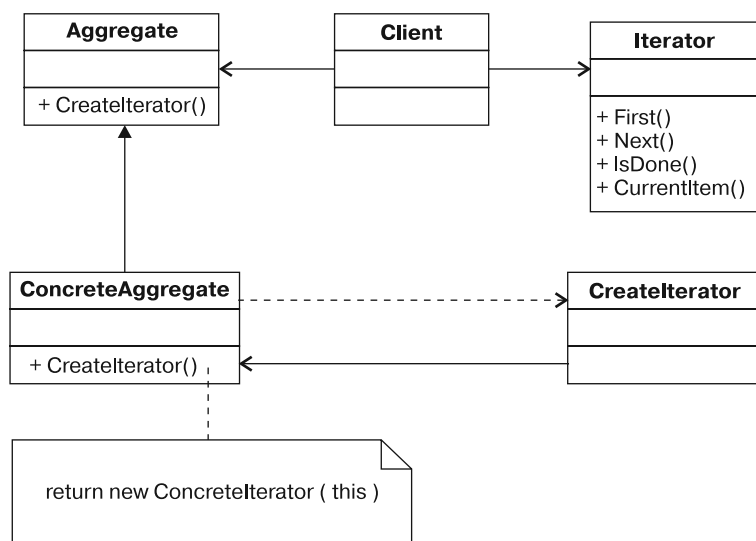


Рис. 3.22. UML-диаграмма для паттерна проектирования «Итератор»

- ❑ **Iterator (Итератор).** Интерфейс или абстрактный класс для доступа к элементам коллекции и их обхода.
- ❑ **ConcreteIterator (Конкретный итератор).** Реализация интерфейса **Iterator**.
- ❑ **Aggregate (Агрегат).** Интерфейс для создания объекта **Iterator**.
- ❑ **ConcreteAggregate (Конкретный агрегат).** Реализация интерфейса **Aggregate**. С помощью реализации интерфейса создания **Iterator** возвращает экземпляры соответствующего **ConcreteIterator**.

Паттерн «Итератор» во фреймворке Spring. Фреймворк Spring реализует данный паттерн в классе **CompositeIterator**. Итератор в основном используется во фреймворке **Collections** языка **Java** для последовательного прохода по элементам коллекции в цикле.

Паттерн проектирования «Наблюдатель»

Задаёт зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного из объектов все зависящие от него объекты оповещаются и модифицируются автоматически.

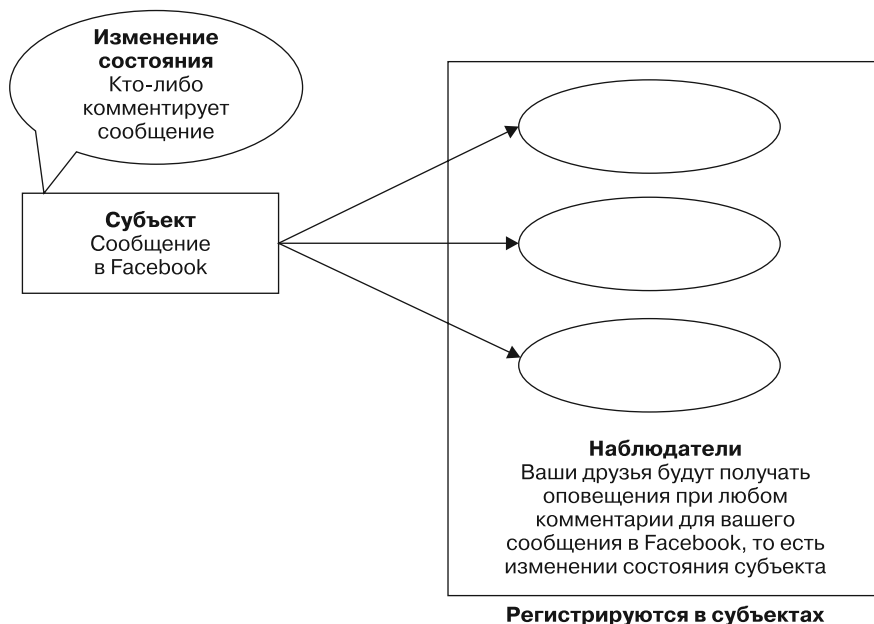
GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

«Наблюдатель» (**Observer**) — один из очень распространенных паттернов. Он относится к семейству поведенческих паттернов проектирования из числа паттернов **GoF**, связанных с обязанностями объектов в приложении и обменом данными

между ними во время выполнения. Согласно этому паттерну между объектами приложения формируется такая зависимость типа «один ко многим», что при модификации одного из объектов все остальные зависимые объекты оповещаются автоматически.

Комментарии к сообщениям в Facebook — один из примеров применения паттерна «Наблюдатель». Если вы комментируете в сообщении одного из ваших друзей, то Facebook всегда будет оповещать вас о других комментариях к этому же сообщению.

Паттерн «Наблюдатель» обеспечивает обмен данными между расцепленными объектами и сводит связи между ними преимущественно к связям типа «один ко многим». В этом паттерне один из объектов называется *субъектом* (subject). При любом изменении в состоянии последнего оповещаются все зависимые от него объекты. Они называются *наблюдателями* (observers). Следующая схема иллюстрирует паттерн проектирования «Наблюдатель» (рис. 3.23).



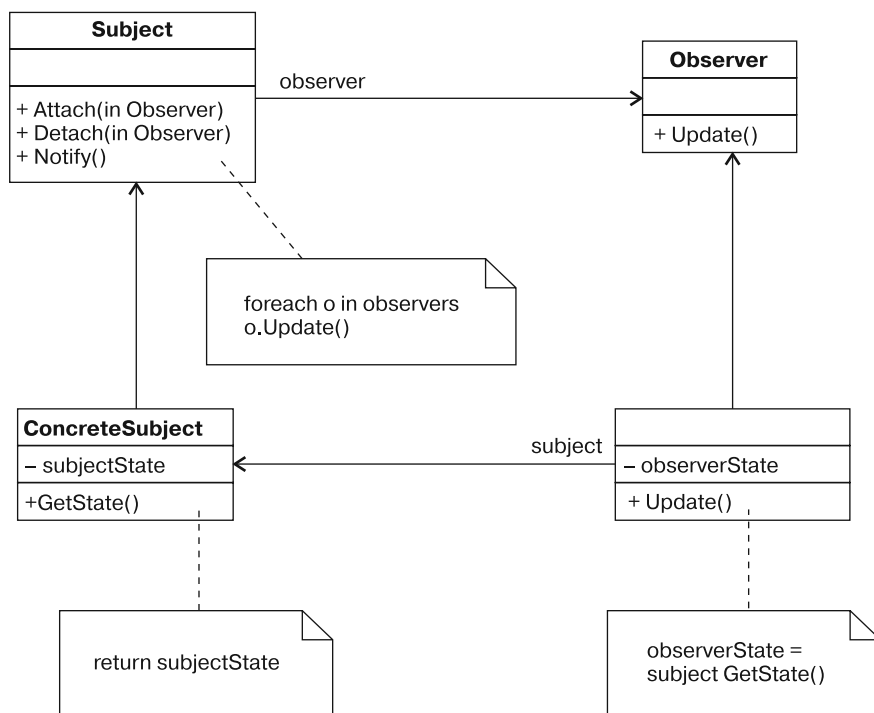


Рис. 3.24. UML-диаграмма для паттерна проектирования «Наблюдатель»

- ❑ **Subject** (Субъект). Интерфейс, содержащий информацию о своих наблюдателях.
- ❑ **ConcreteSubject** (Конкретный субъект). Конкретная реализация интерфейса **Subject**, содержит информацию обо всех наблюдателях, которых нужно оповестить при изменении состояния.
- ❑ **Observer** (Наблюдатель). Интерфейс для объектов, которые должны оповещаться при изменении состояния субъекта.
- ❑ **ConcreteObserver** (Конкретный наблюдатель). Конкретная реализация интерфейса **Observer**, поддерживает согласованное с субъектом состояние.

Паттерн «Наблюдатель» во фреймворке Spring. Во фреймворке Spring этот паттерн используется для реализации функций обработки событий объекта **ApplicationContext**. Для обработки событий в **ApplicationContext** Spring предоставляет класс **ApplicationEvent** и интерфейс **ApplicationListener**. Все компоненты, реализующие интерфейс **ApplicationListener**, будут получать объект **ApplicationEvent** при каждой публикации события издателем. Издатель событий играет здесь роль субъекта, а реализующий интерфейс **ApplicationListener** компонент — наблюдателя.

Паттерн проектирования «Шаблонный метод»

Задаёт каркас алгоритма в операции, делегируя выполнение части этапов алгоритма подклассам. Шаблонный метод позволяет подклассам переопределять некоторые этапы алгоритма без изменения его (алгоритма) структуры.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

В паттерне проектирования «Шаблонный метод» абстрактный класс обертывает некоторые заранее выбранные части алгоритма в метод, позволяющий переопределять его части без переписывания. Для выполнения подобных операций в своих приложениях вы можете использовать конкретный класс. Этот паттерн относится к поведенческим паттернам проектирования GoF.

Среди преимуществ использования паттерна «Шаблонный метод»:

- ❑ уменьшение объема стереотипного кода в приложении за счет переиспользования кода;
- ❑ этот паттерн создает шаблон — способ повторного использования нескольких схожих алгоритмов в целях удовлетворения каких-либо бизнес-требований.

Рассмотрим следующую UML-диаграмму, на которой показаны все компоненты паттерна «Шаблонный метод» (рис. 3.25).

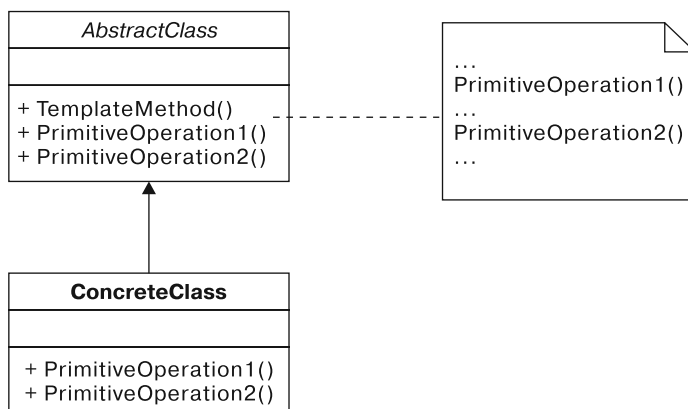


Рис. 3.25. UML-диаграмма для паттерна проектирования «Шаблонный метод»

- ❑ **AbstractClass** (Абстрактный класс). Абстрактный класс, включающий шаблонный метод с описанием каркаса алгоритма.
- ❑ **ConcreteClass** (Конкретный класс). Конкретный подкласс **AbstractClass**, реализующий операции выполнения простейших шагов, относящихся к конкретному алгоритму.

Следующий раздел мы посвятим использованию паттернов проектирования J2EE в корпоративных распределенных приложениях.

Паттерны проектирования J2EE

Это еще одна из основных категорий паттернов проектирования. Паттерны J2EE позволяют значительно упростить архитектуру приложения. Описаны в документации Java Blueprints компании Sun. Представляют собой руководство по проверенным временем готовым решениям и рекомендуемым практикам по интеграции объектов на различных уровнях приложений Java EE. Паттерны J2EE особенно тесно связаны со следующими уровнями приложения.

□ Паттерны проектирования на уровне визуализации.

- *Вспомогательный компонент представления* (View Helper). Разделяет представление с бизнес-логикой корпоративного приложения J2EE.
- *Единая точка входа* (Front Controller). Предоставляет единое место обработки всех входящих запросов к веб-приложению J2EE, перенаправляет запрос конкретному контроллеру приложения для обращения к модели и представлению за ресурсами уровня визуализации¹.
- *Контроллер приложения* (Application Controller). Запрос фактически обрабатывает именно контроллер приложения, играющий роль вспомогательного компонента для единой точки входа. Он отвечает за согласование с компонентами бизнес-модели и представления.
- *Диспетчер представления* (Dispatcher View). Имеет отношение только к представлению, подготавливает ответ для следующего представления, не выполняя никакой бизнес-логики.
- *Перехватывающий фильтр* (Intercepting Filter). В приложениях J2EE есть возможность настройки нескольких перехватчиков для предварительной и постобработки пользовательских запросов, например их отслеживания и аудита.

□ Паттерны проектирования на бизнес-уровне.

- *Бизнес-делегат* (Business Delegate). Служит мостом между контроллерами приложения и бизнес-логикой.
- *Сервис приложения* (Application Service). Предоставляет бизнес-логику реализации модели в виде простых Java-объектов для уровня визуализации.

□ Паттерны проектирования на уровне интеграции.

¹ Эта фраза в корне противоречит описанию работы единой точки входа в главе 10 (а равно и описанию в литературе). См., например, рис. 10.2, где явно показано, что единая точка входа перенаправляет запрос конкретному контроллеру, который создает/обновляет модель, после чего возвращает ее единой точке входа для визуализации на основе представления.

- *Объект доступа к данным* (Data Access Object). Предназначен для доступа к бизнес-данным, отделяет логику доступа к данным от бизнес-логики в корпоративном приложении.
- *Брокер веб-сервиса* (Web Service Broker). Инкапсулирует логику обращения к ресурсам внешнего приложения, доступен в виде веб-сервисов.

Резюме

После прочтения этой главы вы должны иметь четкое представление о паттернах проектирования GoF и рекомендуемых приемах их использования. Я подчеркнул проблемы, возникающие из-за пренебрежения этими паттернами в корпоративных приложениях, и рассказал, как Spring решает их за счет применения множества паттернов проектирования и рекомендуемых практик при создании приложений.

В предыдущей главе я также упоминал три основные категории паттернов GoF. Это порождающие паттерны, которые полезны при создании объектов, особенно в случае применения каких-либо ограничений: «Фабрика», «Абстрактная фабрика», «Строитель», «Прототип» и «Одиночка». Вторая основная категория — структурные паттерны, которые используются для проектирования структуры корпоративных приложений на основе композиции классов или объектов, благодаря чему приложение упрощается и повышаются его повторная используемость и производительность. В их числе паттерны проектирования «Адаптер», «Мост», «Составной объект», «Декоратор» и «Фасад». Наконец, последняя основная категория — поведенческие паттерны, характеризующие способы взаимодействия классов или объектов и распределения между ними обязанностей. Относящиеся к этой категории паттерны связаны прежде всего с обменом данными между объектами.

4

Связывание компонентов с помощью паттерна внедрения зависимостей

В предыдущей главе вы познакомились с *паттернами проектирования GoF*, с примерами и сценариями использования каждого из них. Теперь мы поговорим подробнее о внедрении компонентов и настройках зависимостей в приложениях Spring. Вы увидите различные способы настройки зависимостей, включая конфигурации на основе XML, аннотаций, Java и Mix.

Все любят фильмы, правда? Ну, если не фильмы, то спектакли, балеты, оперу. Когда-нибудь думали о том, что может случиться, если члены коллектива перестанут разговаривать друг с другом? Под коллективом я понимаю не актеров, а художников-декораторов, гримеров, ответственных за спецэффекты, звукооператоров и т. д. Нет нужды упоминать, что каждый из работников вносит важный вклад в конечный продукт, так что координация между командами должна быть очень четкой.

Кассовый фильм обычно является результатом совместной работы сотен людей. Аналогично хорошее ПО представляет собой приложение, в котором для достижения определенной бизнес-цели множество объектов действуют заодно. Все объекты должны быть осведомлены о функционировании друг друга и обмениваться информацией, чтобы достичь общей цели.

В банковской системе сервис перевода денег должен быть осведомлен о функционировании сервиса счетов, а сервис счетов — о репозитории счетов и т. д. Функционирование банковской системы обеспечивается совместной работой всех этих компонентов. В главе 1 вы видели пример банковской системы, написанный на основе традиционного подхода, то есть создания объектов путем их конструирования и непосредственного создания экземпляров. Этот традиционный подход ведет к излишне сложному коду, его повторное использование и модульное тестирование представляют собой непростую задачу. Кроме того, при этом объекты сильно сцеплены друг с другом.

Но в Spring выполнение объектами своих обязанностей не требует поиска и создания других необходимых им зависимых объектов. Контейнер Spring берет на себя работу по поиску или созданию других зависимых объектов, а также по организации взаимодействия с их зависимостями. В предыдущем примере банковской системы

сервис перевода денег зависел от сервиса счетов, но создавать сервис счетов ему не нужно, так как зависимость создается контейнером и передается зависимым объектам.

В этой главе мы обсудим внутренние механизмы и работу приложений Spring применительно к паттерну внедрения зависимостей (dependency injection, DI). К концу этой главы вы узнаете, как создаются зависимости объектов приложений Spring и как Spring связывает их, чтобы достичь желаемого результата. Вы также познакомитесь со множеством способов связывания компонентов в Spring.

Эта глава охватывает следующие темы.

- ❑ Паттерн внедрения зависимостей.
- ❑ Виды внедрения зависимостей.
- ❑ Разрешение зависимости с помощью паттерна «Абстрактная фабрика».
- ❑ Внедрение зависимостей на основе поиска.
- ❑ Задание настроек компонентов с помощью паттерна «Фабрика».
- ❑ Конфигурации зависимостей.
- ❑ Распространенные рекомендуемые практики по настройке зависимостей приложения.

Паттерн внедрения зависимостей

В любом корпоративном приложении для достижения бизнес-цели чрезвычайно важно согласование действий объектов. Отношения между объектами приложения отражают зависимости объектов, так что все объекты могут выполнять свои задачи согласованно с зависимыми объектами. Подобные жесткие зависимости между объектами обычно приводят к усложнению и сильному сцеплению. Spring позволяет решить проблему сильного сцепления благодаря использованию паттерна внедрения зависимостей. Этот паттерн проектирования поощряет использование в приложении слабо сцепленных классов. Это значит, что одни классы в системе зависят от поведения других классов, а не от создания их экземпляров. Паттерн внедрения зависимостей также поощряет работу с интерфейсами вместо реализаций. Объекты должны зависеть от интерфейсов, а не конкретных классов, поскольку слабо сцепленные структуры повышают возможности переиспользования, удобство сопровождения и тестирования.

Решение проблем с помощью паттерна внедрения зависимостей

В любом корпоративном приложении может возникнуть проблема настройки и связывания различных элементов для достижения бизнес-цели — например, привязки контроллеров на веб-уровне к сервисам и интерфейсам репозитория,

написанных различными участниками команды разработчиков без предоставления какой-либо информации о контроллерах веб-уровня. Существует несколько фреймворков, решающих эту проблему за счет использования легковесных контейнеров для компоновки компонентов с различных уровней. К их числу относятся, например, PicoContainer и Spring.

Контейнеры фреймворков PicoContainer и Spring применяют несколько паттернов проектирования для решения проблемы компоновки различных компонентов с различных уровней. Мы обсудим один из этих паттернов — паттерн внедрения зависимостей. При использовании внедрения зависимостей система получается расцепленной или слабо сцепленной. Кроме того, оно гарантирует создание зависимого объекта. В следующем примере мы покажем, как паттерн внедрения зависимостей решает распространенные проблемы взаимодействия между различными компонентами с разных уровней.

Без внедрения зависимостей

В следующем примере на языке Java мы прежде всего определим характер зависимости между двумя классами. Рассмотрим схему классов (рис. 4.1).

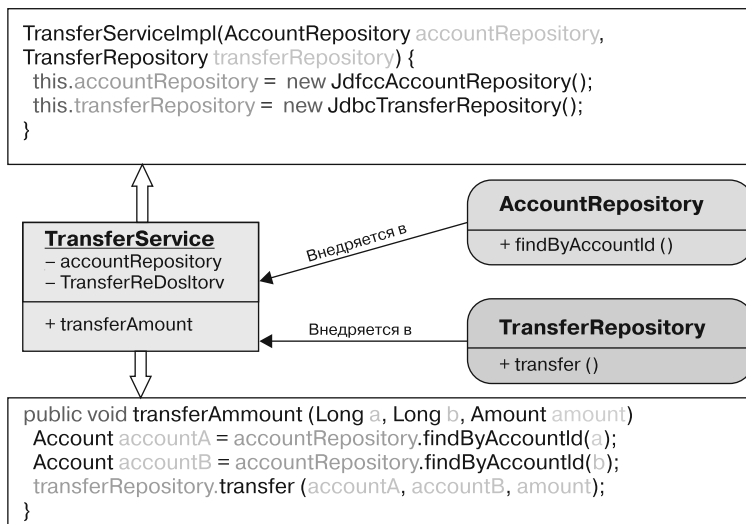


Рис. 4.1. Класс **TransferService** зависит от классов **AccountRepository** и **TransferRepository** вследствие непосредственного создания их экземпляров в методе `transferAmount()`

Как можно видеть на схеме, класс **TransferService** включает две переменные экземпляра: **AccountRepository** и **TransferRepository**. Начальные их значения задаются в конструкторе класса **TransferService**. Он определяет, какие реализации репозитория используются, а также контролирует их конструирование. В подобном

случае говорится, что у `TransferService` имеется жестко запрограммированная зависимость.

Файл `TransferServiceImpl.java`:

```
public class TransferServiceImpl implements TransferService {
    AccountRepository accountRepository;
    TransferRepository transferRepository;
    public TransferServiceImpl(AccountRepository accountRepository,
        TransferRepository transferRepository) {
        super();
        // Задает конкретную реализацию в конструкторе
        // вместо использования внедрения зависимости
        this.accountRepository = new JdbcAccountRepository();
        this.transferRepository = new JdbcTransferRepository();
    }
    // Используя accountRepository и transferRepository метод сервиса
    @Override
    public void transferAmount(Long a, Long b, Amount amount) {
        Account accountA = accountRepository.findById(a);
        Account accountB = accountRepository.findById(b);
        transferRepository.transfer(accountA, accountB, amount);
    }
}
```

В предыдущем примере у класса `TransferServiceImpl` есть зависимости от двух классов, а именно `AccountRepository` и `TransferRepository`. В классе `TransferServiceImpl` есть две переменные экземпляра, относящиеся к зависимым классам, которые инициализируются через конструктор с помощью JDBC-реализаций репозиториев, — `JdbcAccountRepository` и `JdbcTransferRepository`. Класс `TransferServiceImpl` сильно сцеплен с JDBC-реализациями репозиториев. При замене JDBC-реализации на JPA-реализацию придется изменить и класс `TransferServiceImpl`.

Согласно принципам SOLID у класса в приложении должна быть единственная обязанность, но в предыдущем примере класс `TransferServiceImpl` отвечает также за конструирование объектов классов `JdbcAccountRepository` и `JdbcTransferRepository`. Нельзя непосредственно создавать экземпляры объектов в классе.

В нашей первой попытке избежать задействования логики прямого создания экземпляров в классе `TransferServiceImpl` мы воспользуемся классом-фабрикой для создания экземпляров `TransferServiceImpl`. Согласно этой идее минимизируется зависимость класса `TransferServiceImpl` от классов `AccountRepository` и `TransferRepository`. Ранее мы использовали сильно сцепленную реализацию этих репозиториев, а теперь ссылаемся только на интерфейс, как показано на следующей схеме (рис. 4.2).

Но класс `TransferServiceImpl` опять же сильно сцеплен с реализацией класса `RepositoryFactory`. Более того, этот вариант не подходит для случаев большего числа зависимостей, из-за чего растет или количество классов-фабрик, или

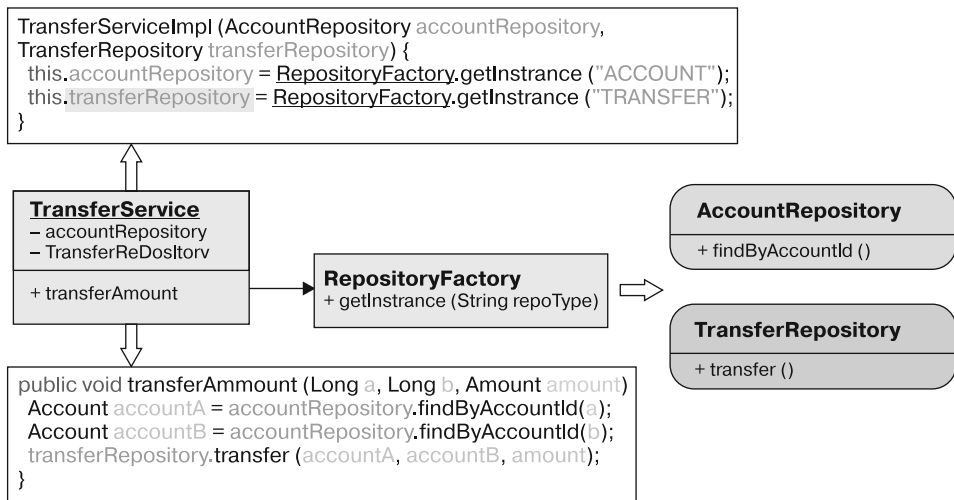


Рис. 4.2. Класс TransferService зависит от классов AccountRepository и TransferRepository вследствие создания их экземпляров для метода transferAmount() с помощью фабрики классов репозиторий

сложность класса-фабрики. У классов-репозиторий могут быть и другие зависимости.

В следующем фрагменте кода класс-фабрика используется для получения экземпляров классов AccountRepository и TransferRepository.

Файл TransferServiceImpl.java:

```

package com.packt.patterninspring.chapter4.bankapp.service;
public class TransferServiceImpl implements TransferService {
    AccountRepository accountRepository;
    TransferRepository transferRepository;
    public TransferServiceImpl(AccountRepository accountRepository,
    TransferRepository transferRepository) {
        this.accountRepository = RepositoryFactory.getInstance();
        this.transferRepository = RepositoryFactory.getInstance();
    }
    @Override
    public void transferAmount(Long a, Long b, Amount amount) {
        Account accountA = accountRepository.findById(a);
        Account accountB = accountRepository.findById(b);
        transferRepository.transfer(accountA, accountB, amount);
    }
}

```

В предыдущем примере кода нам удалось минимизировать сильное сцепление и убрать из класса TransferServiceImpl непосредственное создание экземпляров, но это не оптимальное решение.

При использовании внедрения зависимостей

Благодаря идее с фабрикой удастся избежать непосредственного создания объекта класса. Кроме того, нам нужно создать еще один модуль, в чьи обязанности входило бы связывание зависимостей между классами. Этот модуль называется *механизмом внедрения* (injector) и основывается на паттерне «Инверсия управления» (Inversion of the Control, IoC). Согласно паттерну IoC контейнер отвечает за создание объекта, а также за разрешение зависимостей между классами приложения. У этого модуля есть собственный жизненный цикл между созданием/уничтожением объектов, описанных в пределах его области видимости.

В следующей схеме внедрение зависимостей используется для разрешения зависимостей класса `TransferServiceImpl` (рис. 4.3).

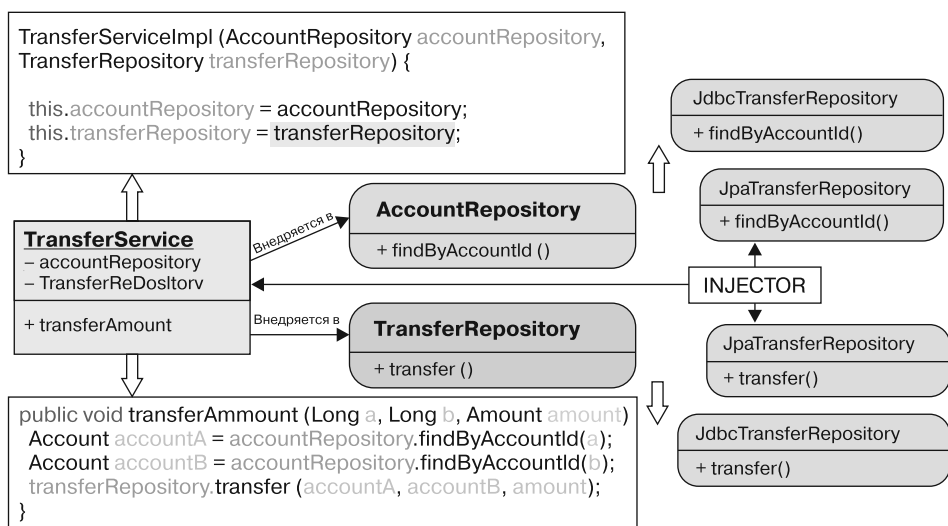


Рис. 4.3. Использование паттерна внедрения зависимостей для разрешения зависимостей класса `TransferService`

В следующем примере мы воспользовались интерфейсом для разрешения зависимостей.

Файл `TransferServiceImpl.java`:

```

package com.packt.patterninspring.chapter4.bankapp.service;
public class TransferServiceImpl implements TransferService {
    AccountRepository accountRepository;
    TransferRepository transferRepository;
    public TransferServiceImpl(AccountRepository accountRepository,
    TransferRepository transferRepository) {
        this.accountRepository = accountRepository;
        this.transferRepository = transferRepository;
    }
    @Override
  
```

```

public void transferAmount(Long a, Long b, Amount amount) {
    Account accountA = accountRepository.findById(a);
    Account accountB = accountRepository.findById(b);
    transferRepository.transfer(accountA, accountB, amount);
}
}

```

В классе `TransferServiceImpl` мы передали конструктору ссылки на интерфейсы `AccountRepository` и `TransferRepository`. Теперь он слабо сцеплен с классом реализации репозитория (можете использовать любые: хоть JDBC-, хоть JPA-реализации интерфейсов репозитория), а фреймворк отвечает за связывание зависимостей с соответствующим зависимым классом. Слабое сцепление означает более широкие возможности переиспользования, большее удобство сопровождения и тестирования.

Фреймворк Spring реализует паттерн внедрения зависимостей для разрешения зависимостей между классами в приложениях Spring. Spring DI основывается на понятии IoC, то есть во фреймворке Spring предусмотрен контейнер, в котором происходит создание/уничтожение объектов и управление ими. Он называется контейнером IoC Spring. Находящиеся в нем объекты известны как *компоненты Spring*. Существует множество способов связывания компонентов в приложении Spring. Мы разберем три наиболее часто используемых подхода настройки контейнера Spring.

В следующем разделе мы рассмотрим виды внедрения зависимостей. Задавать настройки зависимостей можно с помощью любого из них.

Виды внедрения зависимостей

Существует два вида внедрения зависимостей, которые можно использовать в приложениях:

- ❑ через конструктор;
- ❑ через сеттер.

Внедрение зависимостей через конструктор

«Внедрение зависимостей» — паттерн проектирования, предназначенный для разрешения зависимостей зависимых классов, а зависимости — это, по сути, просто атрибуты объектов. Механизм внедрения для зависимых объектов нужно реализовывать одним из двух способов: через конструктор или через сеттер. Внедрение зависимости через конструктор — один из вариантов заполнения этих атрибутов объектов в момент создания объекта. У объекта при этом есть общедоступный конструктор, принимающий в качестве аргументов зависимые классы для внедрения зависимостей. В зависимом классе можно объявить и несколько конструкторов. Ранее только фреймворк `PicoContainer` поддерживал внедрение зависимости через конструктор для разрешения зависимостей. В настоящее время Spring также поддерживает эту возможность.

Преимущества

У внедрения зависимостей через конструктор в приложениях Spring есть следующие преимущества:

- ❑ оно лучше подходит для жестких зависимостей, обеспечивая прочный контракт зависимости;
- ❑ структура кода получается более компактной;
- ❑ обеспечивает возможности тестирования благодаря передаче зависимостей в зависимый класс в виде аргументов конструктора;
- ❑ благоприятствует использованию неизменяемых объектов и не нарушает принципа сокрытия данных.

Недостатки

Единственный недостаток этого паттерна состоит в том, что он может вызывать циклические зависимости (циклическая зависимость означает, что зависимый класс и класс зависимости зависят друг от друга, например, класс А зависит от класса Б, а класс Б — от класса А).

Пример

Рассмотрим пример внедрения зависимости через конструктор. В следующем фрагменте кода описан класс `TransferServiceImpl`, конструктор которого принимает на входе два аргумента:

```
public class TransferServiceImpl implements TransferService {
    AccountRepository accountRepository;
    TransferRepository transferRepository;
    public TransferServiceImpl(AccountRepository accountRepository,
        TransferRepository transferRepository) {
        this.accountRepository = accountRepository;
        this.transferRepository = transferRepository;
    }
    // ...
}
```

Управление репозиториями также осуществляется контейнером Spring, который внедряет в них объект `datasource` для конфигурации базы данных, как показано ниже.

Файл `JdbcAccountRepository.java`:

```
public class JdbcAccountRepository implements AccountRepository{
    JdbcTemplate jdbcTemplate;
    public JdbcAccountRepository(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
    // ...
}
```


Файл `JdbcTransferRepository.java`:

```
public class JdbcTransferRepository implements TransferRepository{
    JdbcTemplate jdbcTemplate;
    public JdbcTransferRepository(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
    // ...
}
```

В вышеприведенном коде можно видеть JDBC-реализацию репозитория `AccountRepository` и `TransferRepository`. У этих классов есть также конструктор с одним аргументом для внедрения зависимости от класса `DataSource`.

Рассмотрим теперь другой способ реализации внедрения зависимостей в корпоративных приложениях, а именно внедрение зависимости через сеттер.

Внедрение зависимости через сеттер

У механизма внедрения контейнера есть еще один способ связывания зависимостей. Внедрение зависимости через сеттер осуществляется путем создания метода-сеттера в зависимом классе. Для внедрения зависимостей используются общедоступные сеттеры, принимающие в качестве аргументов зависимые классы. При внедрении зависимостей на основе сеттеров конструктор зависимого класса не требуется. При изменении зависимостей зависимого класса не нужно выполнять никаких изменений кода. Фреймворк `Spring`, как и фреймворк `PicoContainer`, поддерживает внедрение через сеттер с целью разрешения зависимостей.

Преимущества

У внедрения зависимостей через сеттер в приложениях `Spring` есть следующие преимущества:

- ❑ код оказывается более удобочитаемым;
- ❑ решается проблема циклических зависимостей в приложении;
- ❑ такое внедрение позволяет как можно дольше откладывать создание дорогостоящих ресурсов или сервисов и создавать их только при необходимости;
- ❑ не требуется изменения конструктора, зависимости передаются через предоставляемые общедоступные свойства.

Недостатки

Недостатки паттерна внедрения зависимостей через сеттер:

- ❑ безопасность ниже, поскольку существует возможность переопределения сеттера;

- ❑ структура кода не так компактна, как при внедрении зависимостей через конструктор;
- ❑ зависимости при внедрении через сеттер оказываются необязательными, не забывайте об этом.

Пример

Рассмотрим пример внедрения зависимостей через сеттер. В следующем фрагменте кода описан класс `TransferServiceImpl`, сеттеры которого принимают на входе один аргумент типа репозитория.

Файл `TransferServiceImpl.java`:

```
public class TransferServiceImpl implements TransferService {
    AccountRepository accountRepository;
    TransferRepository transferRepository;
    public void setAccountRepository(AccountRepository
accountRepository) {
        this.accountRepository = accountRepository;
    }
    public void setTransferRepository(TransferRepository
transferRepository) {
        this.transferRepository = transferRepository;
    }
    // ...
}
```

Аналогично описан сеттер для реализаций репозитория, как показано ниже.

Файл `JdbcAccountRepository.java`:

```
public class JdbcAccountRepository implements AccountRepository{
    JdbcTemplate jdbcTemplate;
    public setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
    // ...
}
```

Файл `JdbcTransferRepository.java`:

```
public class JdbcTransferRepository implements TransferRepository{
    JdbcTemplate jdbcTemplate;
    public setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
    // ...
}
```

В вышеприведенном коде можно видеть JDBC-реализацию репозитория `AccountRepository` и `TransferRepository`. У этих классов есть сеттер с одним аргументом для внедрения зависимости от класса `DataSource`.

Сравнение внедрений через конструктор и сеттер, а также рекомендуемые практики

Фреймворк Spring поддерживает оба типа внедрения зависимостей. И тот и другой связывают элементы системы. Выбор между внедрением через конструктор и внедрением через сеттер делают, исходя из требований приложения и решаемой задачи.

Рассмотрим следующую таблицу, в которой перечислены различия между обоими типами внедрения, а также рекомендуемые практики по выбору более подходящего варианта для конкретного приложения.

Внедрение через конструктор	Внедрение через сеттер
Класс с конструктором, принимающим аргументы; код иногда очень компактен, и понятно, что происходит	Объект конструируется, но при этом непонятно, заданы ли начальные значения атрибутов
Оптимальный выбор в случае жесткой зависимости	Подходит в случае нежестких зависимостей
Позволяет скрывать неизменяемые атрибуты объектов в силу отсутствия сеттеров для этих атрибутов. Чтобы обеспечить неизменяемость объектов, следует использовать паттерн внедрения через конструктор, а не через сеттер	Не гарантирует неизменяемости объекта
Создает циклические зависимости в приложении	Решает проблему циклических зависимостей в приложениях. В подобном случае внедрение через сеттер подходит лучше, чем внедрение через конструктор
Не подходит для зависимостей со скалярными значениями	Если зависимости носят характер простых параметров, например строк или целочисленных значений, то лучше использовать внедрение через сеттер, поскольку название сеттера указывает, для чего нужно данное значение

В следующем разделе вы научитесь настраивать механизм внедрения для поиска компонентов и связывания их воедино, а также узнаете, как механизм внедрения управляет компонентами. Для внедрения зависимостей мы воспользуемся конфигурацией Spring.

Описание конфигурации паттерна внедрения зависимостей с помощью Spring

В этом разделе я расскажу о процессе настройки зависимостей в приложении. Основные существующие на рынке механизмы внедрения — Google Guice, Spring и Weld. В этой главе я буду использовать фреймворк Spring, так что мы рассмотрим

конфигурацию Spring. На следующей схеме приведено высокоуровневое представление функционирования Spring (рис. 4.4).

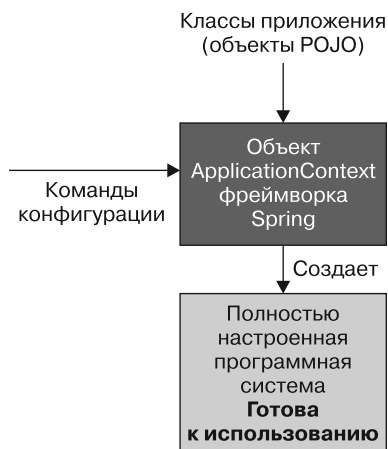


Рис. 4.4. Работа Spring при использовании паттерна внедрения зависимостей

На рис. 4.4 *команды конфигурации* представляют собой метаконфигурацию приложения. Зависимости задаются в *классах приложения (объектах POJO)*, после чего происходит инициализация контейнера Spring для разрешения зависимостей посредством сочетания объектов POJO и команд конфигурации. И наконец, вы получаете полностью настроенную и работоспособную систему или приложение.

Как видите, контейнер Spring создает в приложении компоненты и компоует их по связям между объектами с помощью паттерна DI. Контейнер Spring делает это в соответствии с переданной фреймворку конфигурацией, так что в ваши обязанности входит передача Spring информации о том, какие компоненты создавать и как их связывать между собой.

Возможности настройки зависимостей компонентов Spring очень широки. Приведу три возможных способа задания настроек метаданных приложения.

- ❑ *Использование паттерна внедрения зависимостей с Java-конфигурацией* — явное задание настроек на языке Java.
- ❑ *Использование паттерна DI с конфигурацией на основе аннотаций* — неявное обнаружение компонентов и их автоматическое связывание.
- ❑ *Использование паттерна внедрения зависимостей с XML-конфигурацией* — явное задание настроек в формате XML.

Фреймворк Spring предлагает все три способа связывания компонентов. Вы должны выбрать один из них, но помните, что нет того, который идеально подойдет для любого приложения. Все зависит от самого приложения. Кроме того, можно сочетать и комбинировать эти способы в одном приложении.

Использование паттерна внедрения зависимостей с Java-конфигурацией

Spring версии 3.0 позволяет использовать для связывания компонентов Spring Java-конфигурацию. Рассмотрим следующий класс Java-конфигурации (`AppConfig.java`) с описанием компонентов Spring и их зависимостей. Благодаря своим возможностям и типобезопасности Java-конфигурация — оптимальный выбор для внедрения зависимостей.

Создание класса Java-конфигурации: `AppConfig.java`

Создадим для нашего примера класс Java-конфигурации `AppConfig.java`:

```
package com.packt.patterninspring.chapter4.bankapp.config;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
    // ...
}
```

Вышеприведенный класс `AppConfig` снабжен аннотацией `@Configuration`, указывающей, что это класс конфигурации приложения, который содержит описания компонентов. Контекст приложения Spring загрузит этот файл, чтобы создать компоненты для приложения.

Посмотрим теперь на объявления компонентов `TransferService`, `AccountRepository` и `TransferRepository` из класса `AppConfig`.

Объявления компонентов Spring в классе конфигурации

Для объявления компонента в Java-конфигурации необходимо написать в классе конфигурации метод для создания нужного типа объекта и снабдить этот метод аннотацией `@Bean`. Взглянем на следующие изменения, которые нужно внести в класс `AppConfig` для объявления компонентов:

```
package com.packt.patterninspring.chapter4.bankapp.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService(){
        return new TransferServiceImpl();
    }
    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository();
    }
}
```

```
@Bean
public TransferRepository transferRepository() {
    return new JdbcTransferRepository();
}
}
```

В предыдущем файле конфигурации я объявил три метода для создания экземпляров компонентов: `TransferService`, `AccountRepository` и `TransferRepository`. Они снабжены аннотацией `@Bean`, указывающей, что методы отвечают за создание экземпляров, настройку и инициализацию новых объектов под управлением контейнера `IoC Spring`. У каждого компонента в контейнере имеется уникальный идентификатор. По умолчанию идентификатор совпадает с именем аннотированного `@Bean` компонента. Для предыдущего примера компоненты получают названия `transferService`, `accountRepository` и `transferRepository`. Можно также переопределить это поведение по умолчанию с помощью атрибута `name` аннотации `@Bean` вот так:

```
@Bean(name="service")
public TransferService transferService(){
    return new TransferServiceImpl();
}
}
```

Теперь этот компонент `TransferService` носит имя `service`.

Далее посмотрим на внедрение зависимостей для компонентов `TransferService`, `AccountRepository` и `TransferRepository` в `AppConfig`.

Внедрение компонентов Spring

В предыдущем коде я объявил компоненты без зависимостей — `TransferService`, `AccountRepository` и `TransferRepository`. Но на самом деле `TransferService` зависит от компонентов `AccountRepository` и `TransferRepository`. Внесем в объявления компонентов в классе `AppConfig` следующие изменения:

```
package com.packt.patterninspring.chapter4.bankapp.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService(){
        return new TransferServiceImpl(accountRepository(),
            transferRepository());
    }
    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository();
    }
    @Bean
    public TransferRepository transferRepository() {
        return new JdbcTransferRepository();
    }
}
```

В предыдущем примере простейший способ связывания компонентов в Java-конфигурации — сослаться на метод соответствующего компонента. Метод `transferService()` конструирует экземпляр класса `TransferServiceImpl`, обращаясь к конструктору, принимающему аргументы типов `AccountRepository` и `TransferRepository`. У вас могло сложиться впечатление, что конструктор класса `TransferServiceImpl` вызывает методы `accountRepository()` и `transferRepository()` для создания экземпляров классов `AccountRepository` и `TransferRepository` соответственно, но это не совсем вызов. Контейнер Spring создает экземпляры `AccountRepository` и `TransferRepository`, поскольку методы `accountRepository()` и `transferRepository()` снабжены аннотациями `@Bean`. Все вызовы методов `@Bean` другими такими методами перехватываются Spring, чтобы гарантировать, что у компонентов Spring по умолчанию будет одиночная область видимости (мы обсудим это в главе 5). Это делается путем возврата из метода вместо его дальнейшего вызова.

Оптимальный подход к настройке паттерна внедрения зависимостей с помощью Java

В предыдущем примере конфигурации я объявил метод `transferService()` с аннотацией `@Bean`, предназначенный для конструирования экземпляра класса `TransferServiceImpl`, с помощью конструктора с аргументами. В качестве аргументов конструктора передаются снабженные аннотацией `@Bean` методы `accountRepository()` и `transferRepository()`. Но в корпоративных приложениях многие файлы конфигурации относятся к определенным уровням архитектуры приложения. Допустим, что у уровня сервисов и уровня инфраструктуры есть свои файлы конфигурации. Это значит, что методы `accountRepository()` и `transferRepository()` могут находиться в одних файлах конфигурации, а метод `transferService()` — в другом. При настройке паттерна внедрения зависимостей с помощью Java не рекомендуется передавать методы компонента в конструктор. Рассмотрим другой, более оптимальный подход к настройке внедрения зависимостей:

```
package com.packt.patterninspring.chapter4.bankapp.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService(AccountRepository
accountRepository, TransferRepository transferRepository){
        return new TransferServiceImpl(accountRepository,
transferRepository);
    }
    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository();
    }
}
```

```

@Bean
public TransferRepository transferRepository() {
    return new JdbcTransferRepository();
}
}

```

В предыдущем коде метод `transferService()` получает объекты классов `AccountRepository` и `TransferRepository` как параметры. Фреймворк Spring, вызывая метод `transferService()` для создания компонента `TransferService`, автоматически связывает `AccountRepository` и `TransferRepository` с методом конфигурации. При таком подходе метод `transferService()` по-прежнему может внедрить `AccountRepository` и `TransferRepository` в конструктор класса `TransferServiceImpl`, без того, чтобы ссылаться явным образом на снабженные аннотацией `@Bean` методы `accountRepository()` и `transferRepository()`.

Рассмотрим теперь внедрение зависимостей с помощью XML-конфигурации.

Использование паттерна внедрения зависимостей с XML-конфигурацией

Возможность внедрения зависимостей с XML-конфигурацией была предусмотрена в Spring с самого начала. Это исходный способ настройки приложений Spring. По моему мнению, все разработчики должны знать, как использовать XML при работе с приложениями Spring. В этом подразделе я собираюсь продемонстрировать тот же пример, что и ранее, но теперь уже с XML-конфигурацией вместо Java-конфигурации.

Создание файла XML-конфигурации

В подразделе, посвященном Java-конфигурации, мы создали класс `AppConfig`, снабженный аннотацией `@Configuration`. Аналогично в XML-конфигурации мы создадим файл `applicationContext.xml` с корневым элементом `<beans>`. Следующий простейший из возможных пример демонстрирует базовую структуру метаданных XML-конфигурации.

Файл `applicationContext.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">
    <!-- Здесь располагается конфигурация для описаний компонентов -->
</beans>

```


Вышеприведенный XML-файл представляет собой файл конфигурации приложения, содержащий подробные описания компонентов. Этот файл также загружается XML-ориентированной реализацией объекта `ApplicationContext` с целью создания компонентов для приложения. Взглянем, как объявить в предыдущем файле компоненты `TransferService`, `AccountRepository` и `TransferRepository`.

Объявление компонентов Spring в XML-файле

Как и в случае с Java, нам нужно объявить класс в качестве компонента Spring в XML-конфигурации Spring с помощью элемента `<bean>` схемы компонентов Spring. Элемент `<bean>` — аналог аннотации `@Bean` Java-конфигурации. Добавим в файл XML-конфигурации следующий код:

```
<bean id="transferService"
      class="com.packt.patterninspring.chapter4.
      bankapp.service.TransferServiceImpl"/>
<bean id="accountRepository"
      class="com.packt.patterninspring.chapter4.
      bankapp.repository.jdbc.JdbcAccountRepository"/>
<bean id="transferService"
      class="com.packt.patterninspring.chapter4.
      bankapp.repository.jdbc.JdbcTransferRepository"/>
```

В предыдущем коде я создал очень простое описание компонента. В этой конфигурации у элемента `<bean>` есть атрибут `id` для идентификации описания конкретного компонента. Атрибут `class` содержит полное имя класса для создания компонента. Значение атрибута `id` ссылается на взаимодействующие объекты. Посмотрим, как настроить взаимодействующие компоненты для разрешения зависимостей в приложении.

Внедрение компонентов Spring

Spring предоставляет два способа описания внедрения зависимостей в приложении:

- ☐ внедрение через конструктор;
- ☐ внедрение через сеттер.

Внедрение через конструктор

Spring предоставляет две базовые возможности для внедрения зависимостей через конструктор, а именно: элемент `<constructor-arg>` и появившееся в Spring 3.0 пространство имен конструктора (`c-namespaces`). Пространство имен конструктора отличается лишь чуть большей лаконичностью кода — это единственное различие между ними, так что можете использовать любой из этих вариантов.

Внедрим взаимодействующие компоненты через конструктор:

```
<bean id="transferService"
  class="com.packt.patterninspring.chapter4.
    bankapp.service.TransferServiceImpl">
  <constructor-arg ref="accountRepository"/>
  <constructor-arg ref="transferRepository"/>
</bean>
<bean id="accountRepository"
  class="com.packt.patterninspring.chapter4.
    bankapp.repository.jdbc.JdbcAccountRepository"/>
<bean id="transferRepository"
  class="com.packt.patterninspring.chapter4.
    bankapp.repository.jdbc.JdbcTransferRepository"/>
```

В предыдущей конфигурации у элемента `<bean>` класса `TransferService` есть два подэлемента `<constructor-arg>`. Это значит, что нужно передавать в конструктор класса `TransferServiceImpl` ссылку на компоненты с идентификаторами `accountRepository` и `transferRepository`.

Начиная с версии 3.0, пространство имен конструктора позволяет более лаконично выразить аргументы конструктора на XML. Для использования этого пространства имен необходимо добавить его схему в XML-файл:

```
<?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="transferService"
  class="com.packt.patterninspring.chapter4.
    bankapp.service.TransferServiceImpl"
  c:accountRepository-ref="accountRepository"
  c:transferRepository-ref="transferRepository"/>
<bean id="accountRepository"
  class="com.packt.patterninspring.chapter4.
    bankapp.repository.jdbc.JdbcAccountRepository"/>
<bean id="transferRepository"
  class="com.packt.patterninspring.chapter4.
    bankapp.repository.jdbc.JdbcTransferRepository"/>
  <!-- Здесь могут находиться дополнительные описания компонентов -->
</beans>
```

Посмотрим теперь, как задать эти зависимости с помощью внедрения через сеттер.

Внедрение через сеттер

При этом виде внедрения зависимостей Spring также предоставляет две базовые возможности, а именно: элемент `<property>` и появившееся в Spring 3.0 пространство имен свойств (`p-namespaces`). Использование пространства имен свойств также

лишь повышает лаконичность кода, и это единственное различие между ними, поэтому можно задействовать любой из этих вариантов. Внедрим взаимодействующие компоненты через сеттер:

```
<bean id="transferService"
    class="com.packt.patterninspring.chapter4.
    bankapp.service.TransferServiceImpl">
    <property name="accountRepository" ref="accountRepository"/>
    <property name="transferRepository" ref="transferRepository"/>
</bean>
<bean id="accountRepository"
    class="com.packt.patterninspring.chapter4.
    bankapp.repository.jdbc.JdbcAccountRepository"/>
<bean id="transferRepository"
    class="com.packt.patterninspring.chapter4.
    bankapp.repository.jdbc.JdbcTransferRepository"/>
```

В вышеприведенной конфигурации у элемента `<bean>` класса `TransferService` есть два подэлемента `<property>`, указывающие Spring передавать сеттерам класса `TransferServiceImpl` ссылки на компоненты `accountRepository` и `transferRepository`:

```
package com.packt.patterninspring.chapter4.bankapp.service;

import com.packt.patterninspring.chapter4.bankapp.model.Account;
import com.packt.patterninspring.chapter4.bankapp.model.Amount;
import com.packt.patterninspring.chapter4.bankapp.repository.AccountRepository;
import com.packt.patterninspring.chapter4.bankapp.repository.
TransferRepository;

public class TransferServiceImpl implements TransferService {
    AccountRepository accountRepository;
    TransferRepository transferRepository;
    public void setAccountRepository(AccountRepository
        accountRepository) {
        this.accountRepository = accountRepository;
    }
    public void setTransferRepository(TransferRepository
        transferRepository) {
        this.transferRepository = transferRepository;
    }
    @Override
    public void transferAmmount(Long a, Long b, Amount amount) {
        Account accountA = accountRepository.findById(a);
        Account accountB = accountRepository.findById(b);
        transferRepository.transfer(accountA, accountB, amount);
    }
}
```

Если использовать в вышеприведенном файле компонент без сеттеров, то начальные значения свойств `accountRepository` и `transferRepository` будут пустыми, а зависимость внедрена не будет.

Начиная с Spring 3.0, пространство имен свойств позволяет выразить свойства в XML более лаконично. Для использования этого пространства имен необходимо добавить его схему в XML-файл:

```
<?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="transferService"
  class="com.packt.patterninspring.chapter4.bankapp.
    service.TransferServiceImpl"
  p:accountRepository-ref="accountRepository"
  p:transferRepository-ref="transferRepository"/>
<bean id="accountRepository"
  class="com.packt.patterninspring.chapter4.
    bankapp.repository.jdbc.JdbcAccountRepository"/>
<bean id="transferRepository"
  class="com.packt.patterninspring.chapter4.
    bankapp.repository.jdbc.JdbcTransferRepository"/>
  <!-- Здесь могут находиться дополнительные описания компонентов -->
</beans>
```

Рассмотрим теперь внедрение зависимостей с помощью конфигурации на основе аннотаций.

Использование паттерна внедрения зависимостей с конфигурацией на основе аннотаций

Как обсуждалось в двух предыдущих разделах, при описании паттерна DI с помощью Java- или XML-конфигурации зависимости задаются явным образом. Компоненты Spring создаются или с помощью метода с аннотацией `@Bean` в Java-файле `AppConfig`, или с применением тега `<bean>` в файле XML-конфигурации. Благодаря этим методам можно также создавать компоненты для классов, находящихся вне приложения, то есть классов из сторонних библиотек. Обсудим теперь еще один способ создания компонентов Spring и описания зависимостей между ними с помощью неявной конфигурации: с использованием стереотипных аннотаций.

Что такое стереотипные аннотации

Фреймворк Spring позволяет использовать особые виды аннотаций. Они применяются для автоматического создания компонентов Spring в контексте приложения. Основная стереотипная аннотация — `@Component`. В Spring для нее существуют стереотипные метааннотации, например `@Service`, служащая для создания компонентов Spring на уровне сервисов, `@Repository` — для создания компонентов Spring для

репозиториях на уровне DAO и `@Controller` — для создания компонентов Spring на уровне контроллеров. Это проиллюстрировано на следующей схеме (рис. 4.5).

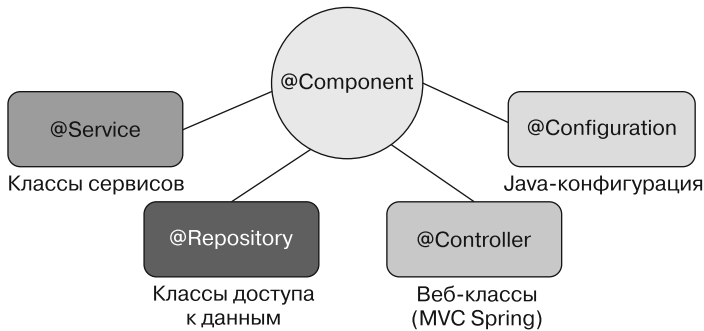


Рис. 4.5. Стереотипные метааннотации

С помощью этих аннотаций Spring выполняет автоматическое связывание следующими двумя способами.

- ❑ *Просмотр компонентов.* Spring автоматически ищет создаваемые компоненты в контейнере Spring IoC.
- ❑ *Автоматическое связывание.* Spring автоматически ищет зависимости компонентов в контейнере Spring IoC.

Неявная конфигурация паттерна DI повышает лаконичность кода приложения и минимизирует размер явной конфигурации. Продемонстрируем просмотр компонентов и автоматическое связывание на том же примере, что обсуждался выше. Spring при этом будет создавать компоненты для `TransferService`, `TransferRepository` и `AccountRepository`, обнаруживая их и внедряя друг в друга в соответствии с описанными зависимостями.

Создание допускающих автоматический поиск компонентов с помощью стереотипных аннотаций

Рассмотрим следующий интерфейс `TransferService`. Его реализация снабжена аннотацией `@Component`.

```
package com.packt.patterninspring.chapter4.bankapp.service;
public interface TransferService {
    void transferAmount(Long a, Long b, Amount amount);
}
```

Этот интерфейс неважен для данного подхода к заданию настроек — я взял его только для слабого сцепления в приложении. Рассмотрим его следующую реализацию:

```
package com.packt.patterninspring.chapter1.bankapp.service;
import org.springframework.stereotype.Component;
```

```

@Component
public class TransferServiceImpl implements TransferService {
    @Override
    public void transferAmount(Long a, Long b, Amount amount) {
        // Код бизнес-логики
    }
}

```

Как можно видеть, класс `TransferServiceImpl` снабжен аннотацией `@Component`. С ее помощью отмечается, что класс относится к компоненту, то есть подходит для обнаружения путем поиска и создания компонента данного класса. Теперь нет необходимости явным образом указывать, что этот класс является компонентом, с помощью Java- или XML-конфигурации, — за создание компонента класса `TransferServiceImpl` теперь отвечает Spring, поскольку класс снабжен аннотацией `@Component`.

Как уже упоминалось, в Spring есть метааннотации для аннотации `@Component` — `@Service`, `@Repository` и `@Controller`. В их основе лежат конкретные обязанности на различных уровнях приложения. В нашем случае `TransferService` является классом уровня сервисов, так что *оптимальным вариантом конфигурации Spring* будет аннотирование этого класса для создания его компонента конкретной аннотацией `@Service` вместо общей `@Component`. Вот код того же класса, снабженного аннотацией `@Service`:

```

package com.packt.patterninspring.chapter1.bankapp.service;
import org.springframework.stereotype.Service;
@Service
public class TransferServiceImpl implements TransferService {
    @Override
    public void transferAmount(Long a, Long b, Amount amount) {
        // Код бизнес-логики
    }
}

```

Посмотрим на остальные классы приложения — классы *реализации* интерфейсов `AccountRepository` и `TransferRepository`, представляющих собой репозитории, работающие на уровне DAO приложения. *Рекомендуется* снабжать эти классы аннотацией `@Repository` вместо общей аннотации `@Component`, как показано далее.

Класс `JdbcAccountRepository.java` реализует интерфейс `AccountRepository`:

```

package com.packt.patterninspring.chapter4.bankapp.repository.jdbc;
import org.springframework.stereotype.Repository;
import com.packt.patterninspring.chapter4.bankapp.model.Account;
import com.packt.patterninspring.chapter4.bankapp.model.Amount;
import com.packt.patterninspring.chapter4.bankapp.repository.AccountRepository;
@Repository
public class JdbcAccountRepository implements AccountRepository {
    @Override
    public Account findById(Long accountId) {
        return new Account(accountId, "Arnav Rajput", new
            Amount(3000.0));
    }
}

```

А класс `JdbcTransferRepository.java` реализует интерфейс `TransferRepository`:

```
package com.packt.patterninspring.chapter4.bankapp.repository.jdbc;
import org.springframework.stereotype.Repository;
import com.packt.patterninspring.chapter4.bankapp.model.Account;
import com.packt.patterninspring.chapter4.bankapp.model.Amount;
import com.packt.patterninspring.chapter4.bankapp.repository.TransferRepository;
@Repository
public class JdbcTransferRepository implements TransferRepository {
    @Override
    public void transfer(Account accountA, Account accountB, Amount
        amount) {
        System.out.println("Transferring amount from account A to B via
            JDBC implementation");
    }
}
```

В Spring необходимо отдельно включать возможность просмотра компонентов, поскольку по умолчанию она отключена. Для этого нужно создать файл класса Java-конфигурации и снабдить этот класс аннотациями `@Configuration` и `@ComponentScan`. Указанный класс используется для поиска классов, аннотированных `@Component`, и создания на их основе компонентов.

Посмотрим, как Spring просматривает классы, снабженные какими-либо из стереотипных аннотаций.

Поиск компонентов с помощью просмотра компонентов

Для поиска компонентов в приложениях Spring с помощью просмотра компонентов требуется как минимум следующая конфигурация:

```
package com.packt.patterninspring.chapter4.bankapp.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan
public class AppConfig {
}
```

Класс `AppConfig` задает класс конфигурации связывания Spring, подобно Java-конфигурации в предыдущем разделе. Стоит отметить, что в файле `AppConfig` есть еще и аннотация `@ComponentScan`, тогда как ранее там была только `@Configuration`. Файл конфигурации `AppConfig` снабжен аннотацией `@ComponentScan` для того, чтобы включить возможность просмотра компонентов в Spring. Аннотация `@ComponentScan` по умолчанию просматривает снабженные аннотацией `@Component` классы, находящиеся в том же пакете, что и класс конфигурации. Поскольку класс `AppConfig` располагается в пакете `com.packt.patterninspring.chapter4.bankapp.config`, то Spring будет просматривать только этот пакет и его подпакеты. Но классы компонентов

нашего приложения находятся в пакетах `com.packt.patterninspring.chapter1.bankapp.service` и `com.packt.patterninspring.chapter4.bankapp.repository.jdbc` и не являются подпакетами пакета `com.packt.patterninspring.chapter4.bankapp.config`. В этом случае Spring позволяет переопределять задаваемый аннотацией `@ComponentScan` по умолчанию способ просмотра пакетов и указывать начальный пакет для просмотра компонентов. Зададим другой начальный пакет. Достаточно просто указать его в атрибуте `value` аннотации `@ComponentScan`, как показано в следующем фрагменте кода:

```
@Configuration
@ComponentScan("com.packt.patterninspring.chapter4.bankapp")
public class AppConfig {
}
```

Можно также описать начальные пакеты с помощью атрибута `basePackages`:

```
@Configuration
@ComponentScan(basePackages="com.packt.patterninspring.chapter4.bankapp")
public class AppConfig {
}
```

Атрибут `basePackages` аннотации `@ComponentScan` может принимать на входе массив строк, то есть с его помощью можно описывать несколько начальных пакетов для просмотра классов компонентов приложения. В предыдущем файле конфигурации Spring просматривает все классы пакета `com.packt.patterninspring.chapter4.bankapp`, а также все его подпакеты. *Рекомендуется* всегда описывать конкретные начальные пакеты с классами компонентов. Например, в следующем коде я описал начальные пакеты для компонентов сервиса и репозитория:

```
package com.packt.patterninspring.chapter4.bankapp.config;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
@Configuration
@ComponentScan(basePackages=
{"com.packt.patterninspring.chapter4.bankapp.repository.jdbc", "com.packt.
patterninspring.chapter4.bankapp.service"})
public class AppConfig {
}
```

Теперь Spring просмотрит только пакеты `com.packt.patterninspring.chapter4.bankapp.repository.jdbc` и `com.packt.patterninspring.chapter4.bankapp.service`, а также их подпакеты, если таковые существуют, вместо того чтобы просматривать более широкий диапазон, как в предыдущих примерах.

Вместо того чтобы указывать пакеты в виде простых строковых значений атрибута `basePackages` аннотации `@ComponentScan`, Spring позволяет задавать их через классы или интерфейсы:

```
package com.packt.patterninspring.chapter4.bankapp.config;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import com.packt.patterninspring.chapter4.bankapp.repository.AccountRepository;
import com.packt.patterninspring.chapter4.bankapp.service.TransferService;
```



```
@Configuration
@ComponentScan(basePackageClasses=
{TransferService.class,AccountRepository.class})
public class AppConfig {

}
```

Как можно видеть в этом коде, мы заменили атрибут `basePackages` на `basePackageClasses`. Теперь Spring будет идентифицировать классы компонентов в соответствующих пакетах, используя `basePackageClasses` как исходный пакет для просмотра компонентов.

Фреймворк должен найти классы `TransferServiceImpl`, `JdbcAccountRepository` и `JdbcTransferRepository` и автоматически создать для них компоненты в контейнере Spring. Нет необходимости описывать явным образом методы компонентов для этих классов с целью создания компонентов Spring. Включим возможность просмотра компонентов с помощью XML-конфигурации, после чего вы сможете воспользоваться элементом `<context:component-scan>` из пространства имен `context` фреймворка Spring. Вот минимальная XML-конфигурация, достаточная для включения возможности просмотра компонентов:

```
<?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.packt.patterninspring.chapter4.bankapp" />
</beans>
```

В этом XML-файле элемент `<context:component-scan>` играет ту же роль, что и аннотация `@ComponentScan` в Java-конфигурации.

Аннотирование компонентов для автоматического связывания

Spring поддерживает автоматическое связывание компонентов. Это значит, что фреймворк автоматически разрешает зависимости, необходимые зависимому компоненту, при обнаружении других взаимодействующих с ним компонентов в контексте приложения. Автоматическое связывание компонентов — еще один способ задания настроек паттерна DI. Оно повышает лаконичность кода приложения, но настройки оказываются разбросанными по всему приложению. Для автоматического связывания компонентов используется аннотация `@Autowired`, которая указывает Spring на необходимость выполнить такое связывание для конкретного компонента.

В нашем примере есть класс `TransferService`, зависящий от классов `AccountRepository` и `TransferRepository`. Его конструктор снабжен аннотацией `@Autowired`, указывающей, что Spring, создавая компонент `TransferService`, должен создать его

экземпляр с помощью аннотированного конструктора, передав в него два других компонента: `AccountRepository` и `TransferRepository`, представляющие собой зависимости компонента `TransferService` (см. следующий код):

```
package com.packt.patterninspring.chapter4.bankapp.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.packt.patterninspring.chapter4.bankapp.model.Account;
import com.packt.patterninspring.chapter4.bankapp.model.Amount;
import com.packt.patterninspring.chapter4.bankapp.repository.AccountRepository;
import com.packt.patterninspring.chapter4.bankapp.repository.TransferRepository;
@Service
public class TransferServiceImpl implements TransferService {
    AccountRepository accountRepository;
    TransferRepository transferRepository;
    @Autowired
    public TransferServiceImpl(AccountRepository accountRepository,
        TransferRepository transferRepository) {
        super();
        this.accountRepository = accountRepository;
        this.transferRepository = transferRepository;
    }
    @Override
    public void transferAmount(Long a, Long b, Amount amount) {
        Account accountA = accountRepository.findById(a);
        Account accountB = accountRepository.findById(b);
        transferRepository.transfer(accountA, accountB, amount);
    }
}
```



В версии Spring 4.3 аннотация `@Autowired` более не требуется при описании в этом классе лишь одного конструктора с аргументами. Если же в классе несколько конструкторов с аргументами, то нужно указать для одного из них аннотацию `@Autowired`.

Использование аннотации `@Autowired` не ограничивается конструкторами, ее можно применять и к сеттерам, а также непосредственно к полям, то есть свойствам класса.

Применение аннотации `@Autowired` к методам-сеттерам

Мы можем снабдить сеттеры `setAccountRepository` и `setTransferRepository` аннотацией `@Autowired`. Ее можно использовать для любого метода, а не только для сеттеров. См. следующий код:

```
public class TransferServiceImpl implements TransferService {
    // ...
    @Autowired
    public void setAccountRepository(AccountRepository
        accountRepository) {
        this.accountRepository = accountRepository;
    }
}
```

```

@Autowired
public void setTransferRepository(TransferRepository
transferRepository) {
    this.transferRepository = transferRepository;
}
// ...
}

```

Применение аннотации @Autowired к полям класса

Мы можем снабдить аннотацией @Autowired те свойства класса, которые необходимы для достижения классом его бизнес-целей (см. следующий код):

```

public class TransferServiceImpl implements TransferService {
    @Autowired
    AccountRepository accountRepository;
    @Autowired
    TransferRepository transferRepository;
    // ...
}

```

В предыдущем коде аннотация @Autowired разрешает зависимости по типу, а затем по имени, если имя свойства совпадает с именем компонента в контейнере Spring. По умолчанию зависимости с аннотацией @Autowired обязательные. Если зависимость не разрешена, то генерируется исключение — неважно, применяется эта аннотация к конструктору или сеттеру. Переопределить принятое по умолчанию поведение аннотации @Autowired можно с помощью ее атрибута required. Необходимо задать для него булево значение false:

```

@Autowired(required = false)
public void setAccountRepository(AccountRepository
accountRepository) {
    this.accountRepository = accountRepository;
}

```

В предыдущем коде мы установили булево значение атрибута required равным false. В этом случае Spring предпримет попытку выполнить автосвязывание, но, если подходящих компонентов нет, он оставит компонент несвязанным. Впрочем, лучше избегать установки значения false этого атрибута и делать это разве что при крайней необходимости.

Автосвязывание зависимостей и неоднозначности

Аннотация @Autowired повышает лаконичность кода, но может вызывать проблемы в случае, когда в контейнере существует два компонента одного типа. Рассмотрим, что произойдет в подобной ситуации, на следующем примере:

```

@Service
public class TransferServiceImpl implements TransferService {
    @Autowired
    public TransferServiceImpl(AccountRepository accountRepository) {
        ...
    }
}

```

Из этого фрагмента кода можно видеть, что у класса `TransferServiceImpl` есть зависимость от компонента типа `AccountRepository`, но в контейнере Spring есть два компонента этого типа, а именно:

```
@Repository
public class JdbcAccountRepository implements AccountRepository
{..}
@Repository
public class JpaAccountRepository implements AccountRepository {..}
```

Как видно из предыдущего кода, существует две реализации интерфейса `AccountRepository`: `JdbcAccountRepository` и `JpaAccountRepository`. В подобном случае контейнер Spring в момент запуска приложения сгенерирует следующее исключение:

```
At startup: NoSuchBeanDefinitionException, no unique bean of type
[AccountRepository] is defined: expected single bean but found 2...
```

Разрешение неоднозначностей при автосвязывании зависимостей

Во фреймворке Spring есть еще одна аннотация, `@Qualifier`, предназначенная для решения проблемы неоднозначности при автосвязывании зависимостей. Рассмотрим следующий фрагмент кода с аннотацией `@Qualifier`:

```
@Service
public class TransferServiceImpl implements TransferService {
    @Autowired
    public TransferServiceImpl( @Qualifier("jdbcAccountRepository")
    AccountRepository accountRepository) { ... }
```

Здесь я связал зависимость по имени, а не по типу, с помощью аннотации `@Qualifier`. При этом Spring будет искать зависимость компонента с именем `jdbcAccountRepository` для класса `TransferServiceImpl`. Я указал также имена компонентов:

```
@Repository("jdbcAccountRepository")
public class JdbcAccountRepository implements AccountRepository
{..}
@Repository("jpaAccountRepository")
public class JpaAccountRepository implements AccountRepository {..}
```

Аннотация `@Qualifier`, которую можно применять и для имен компонентов при внедрении методов и полей, не должна использовать конкретику реализации, за исключением случая двух реализаций одного интерфейса.

Разрешение зависимостей с помощью паттерна «Абстрактная фабрика». Если вы используете Java-конфигурацию, то можете при необходимости ввести условные настройки типа `if...else` для компонента, а также добавить пользовательскую логику. Но добавить условия типа `if...then...else` в случае XML-конфигурации невозможно. Фреймворк Spring позволяет решить проблемы

с условными настройками в XML-конфигурации за счет применения паттерна «Абстрактная фабрика». Вы можете воспользоваться фабрикой для создания нужных компонентов и включить в ее внутреннюю логику сколь угодно сложный код на языке Java.

Реализация паттерна «Абстрактная фабрика» в Spring (интерфейс `FactoryBean`)

Фреймворк Spring реализует паттерн проектирования «Абстрактная фабрика» в виде интерфейса `FactoryBean`, который инкапсулирует логику конструирования объектов в классе. Интерфейс позволяет адаптировать к любым задачам логику создания объектов контейнера Spring IoC. Вы можете реализовывать этот интерфейс для объектов, которые сами являются фабриками. Реализующие `FactoryBean` компоненты обнаруживаются автоматически.

Вот описание этого интерфейса:

```
public interface FactoryBean<T> {
    T getObject() throws Exception;
    Class<T> getObjectType();
    boolean isSingleton();
}
```

Согласно этому описанию внедрение зависимостей с помощью `FactoryBean` приводит к прозрачному вызову метода `getObject()`. Метод `isSingleton()` возвращает `true` для объекта-одиночки и `false` в противном случае. Метод `getObjectType()` возвращает тип объекта, возвращаемого методом `getObject()`.

Реализации интерфейса `FactoryBean` в Spring

Интерфейс `FactoryBean` широко используется в Spring. В частности, он реализован в следующих классах:

- ☐ `EmbeddedDatabaseFactoryBean`;
- ☐ `JndiObjectFactoryBean`;
- ☐ `LocalContainerEntityManagerFactoryBean`;
- ☐ `DateTimeFormatterFactoryBean`;
- ☐ `ProxyFactoryBean`;
- ☐ `TransactionProxyFactoryBean`;
- ☐ `MethodInvokingFactoryBean`.

Пример реализации интерфейса `FactoryBean`

Допустим, у вас есть класс `TransferService` следующего вида:

```
package com.packt.patterninspring.chapter4.bankapp.service;
import com.packt.patterninspring.chapter4.bankapp.
    repository.IAccountRepository;
```

```

public class TransferService {
    IAccountRepository accountRepository;
    public TransferService(IAccountRepository accountRepository){
        this.accountRepository = accountRepository;
    }
    public void transfer(String accountA, String accountB, Double amount){
        System.out.println("Amount has been tranferred");
    }
}

```

А еще у вас есть следующий интерфейс `FactoryBean`:

```

package com.packt.patterninspring.chapter4.bankapp.repository;
import org.springframework.beans.factory.FactoryBean;
public class AccountRepositoryFactoryBean implements
FactoryBean<IAccountRepository> {
    @Override
    public IAccountRepository getObject() throws Exception {
        return new AccountRepository();
    }
    @Override
    public Class<?> getObjectType() {
        return IAccountRepository.class;
    }
    @Override
    public boolean isSingleton() {
        return false;
    }
}

```

Вы можете связать экземпляр `AccountRepository` с помощью гипотетического класса `AccountRepositoryFactoryBean` следующим образом:

```

<?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="transferService" class="com.packt.patterninspring.
chapter4.bankapp.service.TransferService">
        <constructor-arg ref="accountRepository"/>
    </bean>
    <bean id="accountRepository"
        class="com.packt.patterninspring.chapter4.bankapp.
repository.AccountRepositoryFactoryBean"/>
</beans>

```

В предыдущем примере класс `TransferService` зависит от компонента `AccountRepository`, но в XML-файле мы описали `AccountRepositoryFactoryBean` как компонент типа `AccountRepository`. Класс `AccountRepositoryFactoryBean` реализует интерфейс `FactoryBean` фреймворка Spring, поэтому будет переда-

ваться результат вызова метода `getObject` интерфейса `FactoryBean`, а не сам объект типа `FactoryBean`. Spring внедряет этот объект, возвращаемый методом `getObject()` интерфейса `FactoryBean`, и тип объекта, возвращаемый методом `getObjectType()` интерфейса `FactoryBean`. Область видимости данного компонента определяется значением, возвращаемым методом `isSingleton()` интерфейса `FactoryBean`.

Далее приведена та же конфигурация для интерфейса `FactoryBean` на языке Java:

```
package com.packt.patterninspring.chapter4.bankapp.config;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.packt.patterninspring.chapter4.bankapp.
    repository.AccountRepositoryFactoryBean;
import com.packt.patterninspring.chapter4.bankapp.service.TransferService;
@Configuration
public class AppConfig {
    public TransferService transferService() throws Exception{
        return new TransferService(accountRepository().getObject());
    }
    @Bean
    public AccountRepositoryFactoryBean accountRepository(){
        return new AccountRepositoryFactoryBean();
    }
}
```

У `FactoryBean` есть и другие характеристики обычного компонента контейнера Spring, в том числе возможность задействовать точки подключения жизненного цикла и все остальные сервисы, которыми пользуются компоненты в контейнере Spring.

Рекомендуемые практики для конфигураций паттерна DI

Вот некоторые рекомендации для задания настроек паттерна DI.

- ❑ Файлы конфигураций следует разбивать по категориям. Компоненты приложения нужно отделять от инфраструктурных компонентов (рис. 4.6). В настоящее время следовать этой рекомендации непросто.
- ❑ Необходимо всегда указывать имя компонента и никогда не полагаться на сгенерированные контейнером имена.
- ❑ Рекомендуется указывать имя и описание того, что делает паттерн, где его следует применять и какие задачи он решает.
- ❑ Несколько особенностей просмотра компонентов:
 - компоненты просматриваются при запуске приложения, причем просматриваются также и JAR-зависимости;

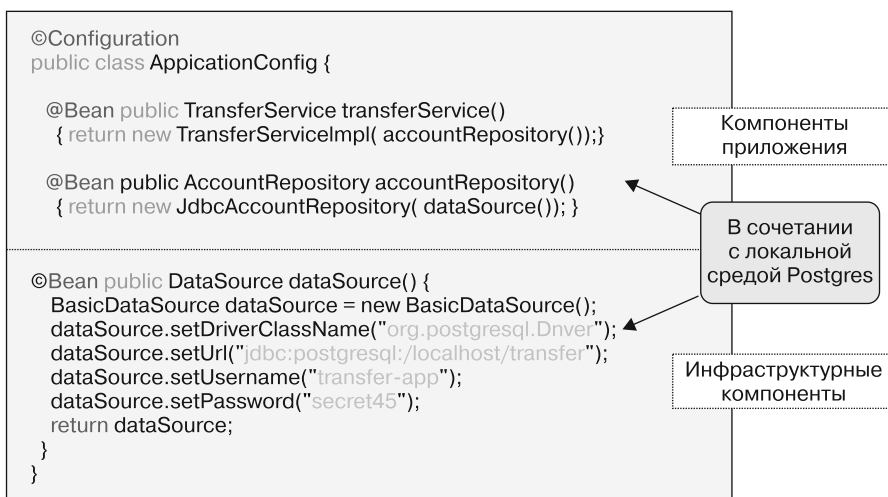


Рис. 4.6. Разбиение по категориям

- *не рекомендуется*: избегайте следующего способа просмотра компонентов, поскольку при этом просматриваются все пакеты типов `com` и `org`, что увеличивает время запуска приложения:

```
@ComponentScan (( {{ "org", "com" }} ))
```

- *оптимизированный вариант*: просматриваются только указанные пакеты:

```
@ComponentScan ( {
    "com.packt.patterninspring.chapter4.bankapp.repository",
    "com.packt.patterninspring.chapter4.bankapp.service"
})
```

❑ Рекомендации для выбора неявной конфигурации:

- используйте конфигурацию на основе аннотаций для часто изменяющихся компонентов;
- неявная конфигурация дает возможность очень быстрой разработки;
- при неявной конфигурации вносить изменения нужно только в одном месте.

❑ Особенности выбора явной Java-конфигурации:

- она сосредоточена в одном месте;
- компилятор обеспечивает строгую проверку типов;
- может использоваться для всех классов.

❑ Рекомендуемые практики Spring XML: XML существует уже долгое время, в XML-конфигурациях можно использовать множество сокращенных форм записи и удобных приемов, которые перечислены ниже:

- атрибуты `factory-method` и `factory-bean`;
- наследование описаний компонентов;

- внутренние компоненты;
- пространства имен свойств и конструкторов;
- использование коллекций в качестве компонентов Spring.

Резюме

После прочтения этой главы вы должны уже неплохо разбираться в паттернах проектирования DI, а также усвоить правила их применения. Spring делает за вас всю грязную работу, так что вы можете сосредоточить свои усилия на решении задач предметной области с помощью паттерна внедрения зависимостей. Паттерн DI освобождает объект от бремени разрешения зависимостей. У объекта есть все, что ему нужно для работы. Паттерн упрощает код, расширяет возможности его переиспользования и тестирования. Он поощряет программирование интерфейсов и скрывает детали реализации зависимостей. Паттерн DI предоставляет возможности централизованного контроля жизненного цикла объекта.

Внедрение зависимостей можно настраивать двумя способами: с использованием явной или неявной конфигурации. Явную конфигурацию — централизованную — можно задать с помощью XML- или Java-конфигурации. В основе же неявной конфигурации лежат аннотации. Spring предоставляет для этой цели стереотипные аннотации. Такая конфигурация делает код приложения более лаконичным, но разбросанным по файлам приложения.

В следующей главе мы поговорим о жизненном цикле компонентов Spring в контейнере.

5

Жизненный цикл компонентов и используемые паттерны

В предыдущей главе мы рассмотрели создание фреймворком Spring компонентов в контейнере, а также поговорили о настройке паттерна внедрения зависимостей с помощью XML, Java и аннотаций. В этой главе мы рассмотрим темы, выходящие за пределы внедрения компонентов и конфигурации зависимостей в приложении Spring. Мы разберемся с жизненным циклом и областями видимости компонентов в контейнере, а также выясним, как работает контейнер Spring при конфигурации компонента Spring, заданной с помощью XML или Java либо основанной на аннотациях. Spring позволяет управлять как различными конфигурациями паттерна внедрения зависимостей и их значениями, внедряемыми в объект, созданный на основе конкретного описания компонента, так и жизненным циклом компонентов, созданных на базе конкретного описания компонента.

Когда я писал эту главу, мой трехлетний сын Арнав пришел ко мне и начал играть в игру на моем смартфоне. На нем в этот момент была футболка с интересным высказыванием, описывавшим весь его день: *«Мой идеальный день — проснуться, сыграть в компьютерную игру, поест, сыграть в компьютерную игру, поест, сыграть в компьютерную игру и поспать»*.

И действительно, это высказывание идеально отражало его ежедневный жизненный цикл, ведь он просыпался, играл, ел, снова играл и, наконец, укладывался спать. Этим примером я хотел продемонстрировать, что жизненный цикл есть у всего. Мы могли бы говорить о жизненном цикле бабочки, звезды, лягушки или растения. Но поговорим о более интересном понятии — жизненном цикле компонента!

У каждого компонента в контейнере Spring есть свой жизненный цикл и область видимости. Контейнер Spring управляет жизненным циклом компонентов в приложении Spring. Для адаптации к конкретной задаче можно воспользоваться совместимыми со Spring интерфейсами. В этой главе мы поговорим о жизненном цикле компонентов в контейнере и об управлении компонентами в различных фазах их жизненного цикла с помощью паттернов проектирования. К концу этой главы у вас появится представление о жизненном цикле компонента в контейнере и различных

его фазах. Вы также узнаете о различных областях видимости компонентов в Spring. Данная глава охватывает следующие вопросы.

❑ Жизненный цикл компонента Spring и его фазы:

- инициализации;
- использования;
- уничтожения.

❑ Обратные вызовы в Spring.

❑ Области видимости компонента:

- паттерн «Одиночка»;
- паттерн «Прототип»;
- пользовательские области видимости;
- другие области видимости компонентов.

Теперь же уделим немного времени вопросу о том, как Spring управляет жизненным циклом компонента от его создания до уничтожения в приложении Spring.

Жизненный цикл компонента Spring и его фазы

В приложении Spring понятие «жизненный цикл» применимо к любому виду приложения — автономному Java-приложению, приложениям на основе Spring Boot или комплексным/системным тестам. Понятие «жизненный цикл» также применимо ко всем трем стилям внедрения зависимостей — конфигурациям на основе XML, Java и аннотаций. Конфигурации для компонентов обычно описываются в соответствии с бизнес-целями. Но создает эти компоненты и управляет их жизненным циклом Spring. Spring загружает конфигурации компонентов в Java или XML через интерфейс `ApplicationContext`. После загрузки этих компонентов контейнер Spring создает их экземпляры в соответствии с конфигурацией. Разобьем жизненный цикл приложения Spring на три следующие фазы:

- ❑ инициализации;
- ❑ использования;
- ❑ уничтожения.

Взгляните на рис. 5.1.

Как можно видеть из этой схемы, компонент Spring проходит три фазы своего существования. В каждой фазе для каждого компонента должен быть выполнен определенный набор операций (в зависимости от конфигурации). Spring отлично подходит для управления жизненным циклом приложений. Он играет важную роль во всех трех фазах.

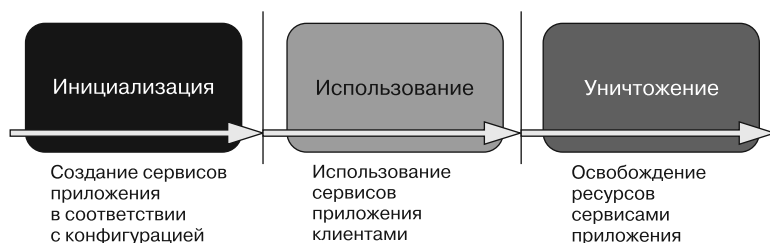


Рис. 5.1. Жизненный цикл приложения Spring

Теперь рассмотрим, как Spring работает в первой фазе — фазе инициализации.

Фаза инициализации

В этой фазе Spring прежде всего загружает все файлы конфигураций во всех стилях — XML, Java и на основе аннотаций. В этой же фазе происходит подготовка компонентов к использованию — фактически создаются сервисы приложения и компоненту выделяются системные ресурсы. Spring предоставляет интерфейс `ApplicationContext` для загрузки конфигураций компонентов; с созданием контекста приложения фаза инициализации завершается. Теперь взглянем, как Spring загружает файлы конфигураций в Java и XML.

Создание контекста приложения по конфигурации

Spring предоставляет множество реализаций интерфейса `ApplicationContext` для загрузки различных стилей файлов конфигурации.

- ❑ Для Java-конфигураций используется следующая реализация:

```
ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
```

- ❑ Для XML-конфигураций реализация имеет такой вид:

```
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
```

В предыдущих фрагментах кода Spring загружает файлы Java-конфигурации с помощью класса `AnnotationConfigApplicationContext`, а файлы XML-конфигурации — с помощью класса `ClassPathXmlApplicationContext` для контейнера Spring. Spring поступает аналогичным образом для всех типов конфигураций. Не имеет значения, какие стили конфигураций используются в вашем приложении. Следующая схема наглядно демонстрирует происходящее в этой фазе (рис. 5.2).

Как вы можете видеть на рис. 5.2, фаза инициализации делится на два шага.

1. Загрузка описаний компонентов.
2. Инициализация экземпляров компонентов.

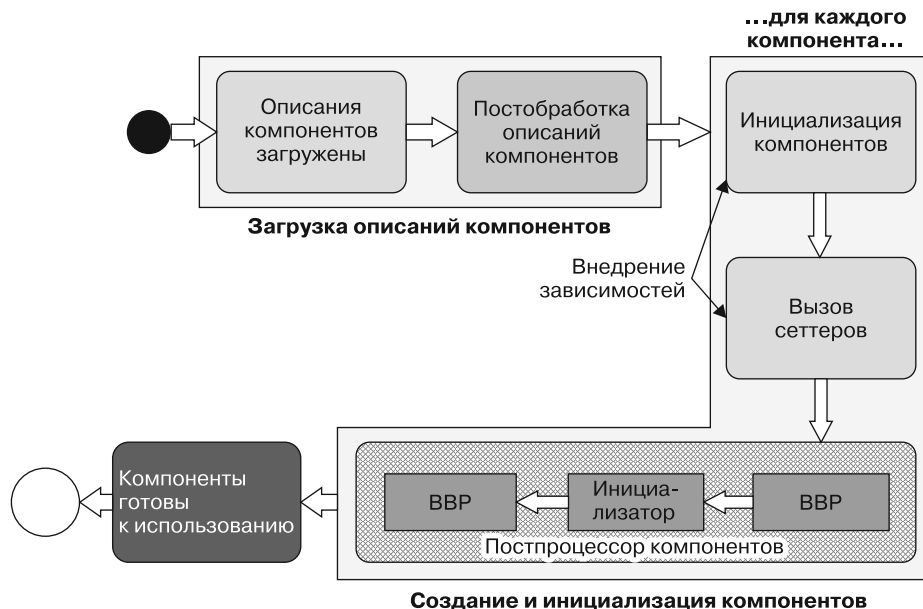


Рис. 5.2. Фаза инициализации

Загрузка описаний компонентов

На этом шаге обрабатываются все файлы конфигураций — классы `@Configuration` или XML-файлы. Для основанных на аннотациях конфигураций просматриваются все файлы, аннотированные `@Configuration`, с целью загрузки описаний компонентов. Выполняется синтаксический разбор всех XML-файлов, и все описания компонентов добавляются в объект `BeanFactory`. Все компоненты индексируются по `id`. В Spring имеется множество компонентов — реализаций интерфейса `BeanFactoryPostProcessor`. Так, он вызывается для разрешения зависимостей этапа выполнения, например для чтения значений из внешних файлов свойств. В приложении Spring интерфейс `BeanFactoryPostProcessor` может модифицировать описание любого из компонентов. Следующая схема описывает данный шаг (рис. 5.3).



Рис. 5.3. Описания компонентов

Как показано на схеме, Spring сначала загружает описания компонентов, а затем вызывает для части компонентов интерфейс `BeanFactoryProcessor`, чтобы модифицировать их описания соответствующим образом. Посмотрим на этот процесс на примере. Возьмем два файла конфигураций — `AppConfig.java` и `InfraConfig.java` — следующего вида:

❑ файл `AppConfig.java`:

```
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService(){ ... }
    @Bean
    public AccountRepository accountRepository(DataSource
        dataSource){ ... }
}
```

❑ файл `InfraConfig.java`:

```
@Configuration
public class InfraConfig {
    @Bean
    public DataSource dataSource () { ... }
}
```

Интерфейс `ApplicationContext` загружает эти файлы Java-конфигураций в контейнер и индексирует их по `id`, как показано на рис. 5.4.

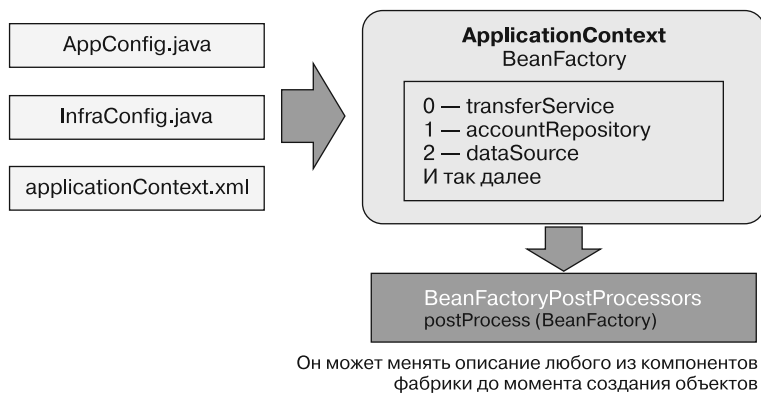


Рис. 5.4. Работа интерфейса `ApplicationContext`

На этой схеме компоненты Spring индексируются в соответствии с их идентификаторами в объекте `BeanFactory`, после чего объект `BeanFactory` передается в качестве аргумента методу `postProcess()` интерфейса `BeanFactoryPostProcessor`. Интерфейс `BeanFactoryPostProcessor` может менять описания некоторых компонентов в зависимости от заданных разработчиком конфигураций. Посмотрим, как он работает и как переопределить его в нашем приложении.

1. `BeanFactoryPostProcessor` работает с описаниями компонентов или метаданными конфигураций компонентов еще до собственно создания этих компонентов.
2. Spring предоставляет несколько удобных реализаций `BeanFactoryPostProcessor`, например для чтения свойств или регистрации пользовательской области видимости.
3. Вы можете создавать свои собственные реализации интерфейса.
4. Если вы определяете `BeanFactoryPostProcessor` в одном контейнере, то и применяться он будет только к описаниям компонентов в этом контейнере.

Вот фрагмент кода для интерфейса `BeanFactoryPostProcessor`:

```
public interface BeanFactoryPostProcessor {
    public void postProcessBeanFactory
        (ConfigurableListableBeanFactory
         beanFactory);
}
```

Далее мы рассмотрим несколько примеров точек расширения интерфейса `BeanFactoryPostProcessor`.

Чтение внешних файлов свойств (database.properties)

В этом примере мы воспользуемся компонентом `DataSource`, задав в конфигурации такие значения базы данных, как `username`, `password`, `db url` и `driver`, следующим образом:

```
jdbc.driver=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsq://production:9002
jdbc.username=doj
jdbc.password=doj@123
```

А вот описание компонента `DataSource` в файле конфигурации:

```
@Configuration
@PropertySource ( "classpath:/config/database.properties" )
public class InfraConfig {
    @Bean
    public DataSource dataSource(
        @Value("${jdbc.driver}") String driver,
        @Value("${jdbc.url}") String url,
        @Value("${jdbc.user}") String user,
        @Value("${jdbc.password}") String pwd ) {
        DataSource ds = new BasicDataSource();
        ds.setDriverClassName( driver);
        ds.setUrl( url);
        ds.setUser( user);
        ds.setPassword( pwd );
        return ds;
    }
}
```

Как же в вышеприведенном коде мы можем разрешить переменные `@Value` и `${..}`? Для их вычисления нам понадобится класс `PropertySourcesPlaceholderConfigurer` — одна из реализаций интерфейса `BeanFactoryPostProcessor`. Если вы используете XML-конфигурацию, то для создания `PropertySourcesPlaceholderConfigurer` достаточно указать пространство имен `<context:property-placeholder/>`.

Загрузка описания компонента представляет собой однократный процесс, выполняемый в момент загрузки файла конфигурации, но фаза инициализации экземпляров компонентов выполняется для каждого компонента контейнера. Посмотрим, как происходит инициализация экземпляров компонентов в приложении.

Инициализация экземпляров компонентов

После загрузки описаний компонентов в `BeanFactory` контейнер IoC Spring создает экземпляры компонентов для приложения. На рис. 5.5 приведена последовательность операций.

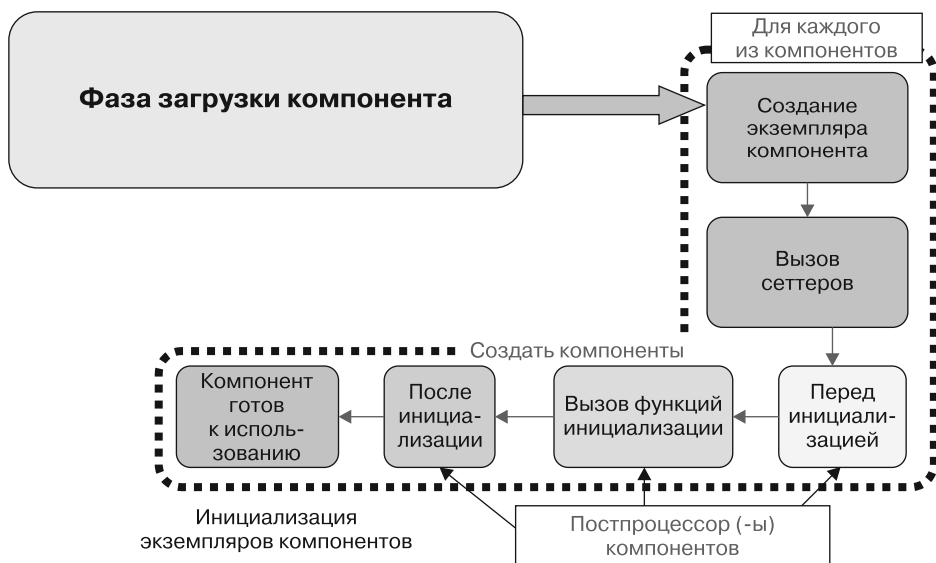


Рис. 5.5. Инициализация экземпляров компонентов

Как вы можете видеть, шаг инициализации выполняется для каждого компонента в контейнере. Можно кратко описать процесс создания компонента следующим образом.

- ❑ По умолчанию экземпляры всех компонентов создаются немедленно, причем в правильном порядке, с внедрением зависимостей, за исключением помеченных для отложенного выполнения.
- ❑ Spring предоставляет множество реализаций интерфейса `BeanPostProcessor`, так что каждый из компонентов проходит фазу постобработки, например, с по-

мощью интерфейса `BeanFactoryPostProcessor`, способного менять описание компонента. Однако интерфейс `BeanPostProcessor` может модифицировать экземпляр компонента.

- ❑ После выполнения этой фазы компонент полностью инициализирован и готов к использованию. Его можно отслеживать по `id` вплоть до момента уничтожения контекста, за исключением прототипных компонентов.

В следующем разделе мы обсудим адаптацию контейнера Spring к конкретной задаче с помощью интерфейса `BeanPostProcessor`.

Адаптация компонентов к конкретным задачам с помощью интерфейса `BeanPostProcessor`

Интерфейс `BeanPostProcessor` — важная точка расширения в Spring. Он способен модифицировать экземпляры компонентов практически как угодно. Благодаря ему становится доступна такая важная возможность, как AOP-прокси. Вы можете создать пользовательский постпроцессор, написав собственную реализацию интерфейса `BeanPostProcessor` в своем приложении. Spring предоставляет несколько готовых реализаций `BeanPostProcessor`. В Spring у интерфейса `BeanPostProcessor` имеется два метода обратного вызова:

```
public interface BeanPostProcessor {
    Object postProcessBeforeInitialization(Object bean, String
        beanName) throws BeansException;
    Object postProcessAfterInitialization(Object bean, String
        beanName) throws BeansException;
}
```

Вы можете реализовать в этих двух методах интерфейса `BeanPostProcessor` собственную логику создания экземпляров компонентов, разрешения зависимостей и т. д. Существует также возможность настройки одной из нескольких реализаций интерфейса `BeanPostProcessor` для добавления пользовательской логики в контейнер Spring. В то же время можно управлять порядком выполнения этих реализаций с помощью задания соответствующего свойства. Интерфейс `BeanPostProcessor` работает с экземплярами компонентов после их создания контейнером Spring. Область видимости `BeanPostProcessor` находится в пределах контейнера Spring, а значит, определенные в одном контейнере компоненты недоступны для постобработки интерфейсом, заданным в другом.

Каждый класс приложения Spring регистрируется в контейнере как постпроцессор; он создается для любого экземпляра компонента контейнером Spring. А контейнер Spring вызывает `postProcessBeforeInitialization()` до методов инициализации контейнера (метода `afterPropertiesSet()` интерфейса `InitializingBean` и метода `init` компонента). Он также вызывает метод `postProcessAfterInitialization()` после всех обратных вызовов инициализации компонента. AOP Spring использует постпроцессоры для формирования прокси-адаптера (паттерн проектирования «Заместитель»), хотя с помощью постпроцессора можно выполнить любое действие.

Интерфейс `ApplicationContext` фреймворка Spring автоматически обнаруживает компоненты, реализующие интерфейс `BeanPostProcessor`, и регистрирует их

как постпроцессоры. Эти компоненты вызываются при создании всех остальных компонентов. Рассмотрим пример реализации `BeanPostProcessor`.

Создадим следующий пользовательский постпроцессор для компонентов:

```
package com.packt.patterninspring.chapter5.bankapp.bpp;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.stereotype.Component;
@Component
public class MyBeanPostProcessor implements
BeanPostProcessor {
    @Override
    public Object postProcessBeforeInitialization
(Object bean, String beanName) throws BeansException {
        System.out.println("In After bean Initialization
        method. Bean name is "+beanName);
        return bean;
    }
    public Object postProcessAfterInitialization(Object bean, String
beanName) throws BeansException {
        System.out.println("In Before bean Initialization method. Bean
name is "+beanName);
        return bean;
    }
}
```

Это простейший пример для иллюстрации использования постпроцессора, в нем постпроцессор выводит в системную консоль строку текста для каждого зарегистрированного в контейнере компонента. Класс `MyBeanPostProcessor` аннотирован `@Component`, а значит, подобен любому другому классу компонента в контексте приложения. Теперь выполним следующий демонстрационный класс. Взгляните на фрагмент кода:

```
public class BeanLifecycleDemo {
    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext = new
        AnnotationConfigApplicationContext(AppConfig.class);
        applicationContext.close();
    }
}
```

На рис. 5.6 показаны результаты вывода в консоль.

Как вы можете видеть, строка для обоих методов обратного вызова выводится для метода каждого компонента в контейнере Spring. Spring предоставляет множество заранее реализованных вариантов интерфейса `BeanPostProcessor` для конкретных возможностей, например:

- ☐ `RequiredAnnotationBeanPostProcessor`;
- ☐ `AutowiredAnnotationBeanPostProcessor`;
- ☐ `CommonAnnotationBeanPostProcessor`;
- ☐ `PersistenceAnnotationBeanPostProcessor`.

```

<terminated> BeanLifeCycleDemo [Java Application] C:\Program Files\Java\jre1.8.0_131\bin\javaw.exe (04-Jul-2017, 11:27:21 PM)
Jul 04, 2017 11:27:23 PM org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@6e2c634b: startup date
In After bean Initialization method. Bean name is org.springframework.context.event.internalEventListenerProcessor
In Before bean Initialization method. Bean name is org.springframework.context.event.internalEventListenerProcessor
In After bean Initialization method. Bean name is org.springframework.context.event.internalEventListenerFactory
In Before bean Initialization method. Bean name is org.springframework.context.event.internalEventListenerFactory
In After bean Initialization method. Bean name is appConfig
In Before bean Initialization method. Bean name is appConfig
In After bean Initialization method. Bean name is transferService
In Before bean Initialization method. Bean name is transferService
Jul 04, 2017 11:27:23 PM org.springframework.context.annotation.AnnotationConfigApplicationContext doClose
INFO: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@6e2c634b: startup date [Tu

```

Рис. 5.6. Результаты работы класса BeanLifeCycleDemo

Пространство имен `<context:annotation-config/>` из XML-конфигурации дает возможность использовать несколько постпроцессоров в одном контексте приложения (там, где оно определено).

Перейдем к следующему разделу и посмотрим, как с помощью `BeanPostProcessor` реализовать точку расширения для инициализации.

Точка расширения — инициализатор

Это особый вариант компонента-постпроцессора, осуществляющего вызов методов `init` (`@PostConstruct`). На внутреннем уровне Spring использует для обеспечения инициализации несколько интерфейсов `BeanPostProcessor` (BPP) типа `CommonAnnotationBeanPostProcessor`. Следующая схема иллюстрирует взаимодействие между инициализатором и BPP (рис. 5.7).

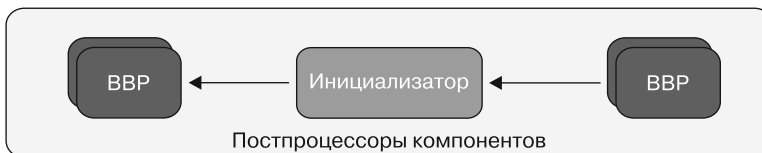


Рис. 5.7. Взаимодействие между инициализатором и BPP

Рассмотрим теперь следующий пример точки расширения — инициализатора — на XML.

Пространство имен `<context:annotation-config/>` явным образом разрешает использование нескольких постпроцессоров, вот соответствующий файл конфигурации на 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"
  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-4.3.xsd">
<context:annotation-config/>
<bean id="transferService"
class="com.packt.patterninspring.chapter5.
bankapp.service.TransferService"/>
<bean id="accountRepository"
class="com.packt.patterninspring.chapter5.
bankapp.repository.JdbcAccountRepository"
init-method="populateCache"/>
</beans>
```

Из этого кода конфигурации видно, что я описал несколько компонентов, у тега компонента одного из которых — репозитория `accountRepository` — есть метод `init`, а у этого атрибута есть значение `populateCache`. Это не что иное, как метод-инициализатор компонента `accountRepository`. Контейнер вызывает его в момент инициализации компонента, если использование постпроцессоров возможно явным образом благодаря пространству имен `<context:annotation-config/>`. Рассмотрим следующий класс `JdbcAccountRepository`:

```
package com.packt.patterninspring.chapter5.bankapp.repository;
import com.packt.patterninspring.chapter5.bankapp.model.Account;
import com.packt.patterninspring.chapter5.bankapp.model.Amount;
import com.packt.patterninspring.chapter5.
    bankapp.repository.AccountRepository;
public class JdbcAccountRepository implements AccountRepository {
    @Override
    public Account findById(Long accountId) {
        return new Account(accountId, "Arnav Rajput", new
            Amount(3000.0));
    }
    void populateCache(){
        System.out.println("Called populateCache() method");
    }
}
```

В Java-конфигурации можно воспользоваться атрибутом `initMethod` аннотации `@Bean`:

```
@Bean(initMethod = "populateCache")
public AccountRepository accountRepository(){
    return new JdbcAccountRepository();
}
```

В конфигурации же на основе аннотаций можно задействовать аннотацию из JSR-250¹ — `@PostConstruct`:

```
@PostConstruct
void populateCache(){
    System.out.println("Called populateCache() method");
}
```

¹ Java Specification Request (запрос на спецификацию Java).

Мы рассмотрели первую фазу жизненного цикла компонента, в которой Spring загружает описания компонентов с помощью XML-, Java-конфигураций или конфигураций на основе аннотаций, после чего контейнер Spring инициализирует все компоненты в правильном порядке в приложении Spring. На рис. 5.8 приведен обзор первой фазы жизненного цикла конфигурации.

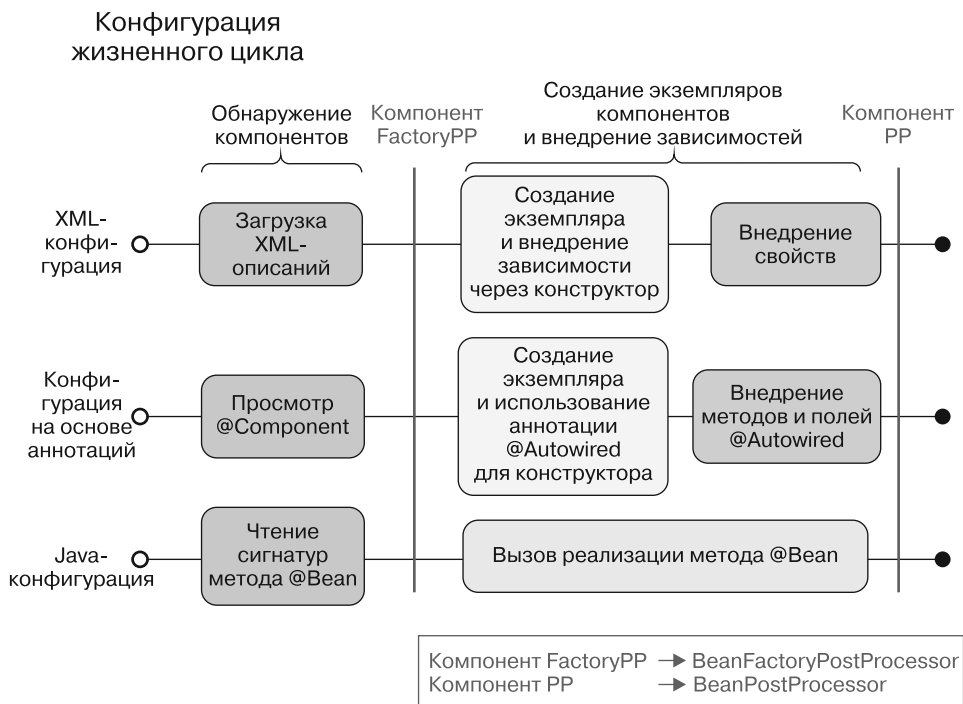


Рис. 5.8. Первая фаза жизненного цикла конфигурации

Эта схема демонстрирует метаданные компонента Spring в каждом из стилей — XML, Java и на основе аннотаций, — загружаемые соответствующей реализацией интерфейса `ApplicationContext`. Все XML-файлы подвергаются синтаксическому разбору и загружаются вместе с описаниями компонентов. В конфигурации на основе аннотаций Spring просматривает все компоненты и загружает описания компонентов. В Java-конфигурации Spring читает все методы `@Bean` с целью загрузки описаний компонентов. После загрузки описаний компонентов из конфигураций любого стиля наступает черед модификации описаний некоторых компонентов с помощью `BeanFactoryPostProcessor`, после чего контейнер создает экземпляры компонентов. Наконец, компоненты обрабатывает интерфейс `BeanPostProcessor`, способный модифицировать и менять их объекты.

Рассмотрим теперь следующую фазу жизненного цикла компонента — фазу использования.

Фаза использования компонентов

В приложении Spring 99,99 % времени компоненты находятся именно в этой фазе. В нее компонент Spring переходит в случае успешного завершения фазы инициализации. В ней компоненты используются клиентами в качестве сервисов приложения. Эти компоненты обрабатывают запросы клиентов и осуществляют возложенные на приложение обязанности. Посмотрим, как в фазе использования выполнить вызов полученного из контекста компонента *в приложении, в котором он используется*. Посмотрите на следующий фрагмент кода:

```
// Получение или создание контекста приложения
ApplicationContext applicationContext = new
    AnnotationConfigApplicationContext(AppConfig.class);

// Поиск точки входа в приложение
TransferService transferService =
    context.getBean(TransferService.class);
// и ее использование
transferService.transfer("A", "B", 3000.1);
```

Если сервис `return` возвращает первичный объект, то он просто вызывался напрямую — ничего особенного в этом нет. Но если у компонента есть адаптер-прокси, то ситуация становится интереснее. Взглянем на рис. 5.9, чтобы лучше разобраться в этом.

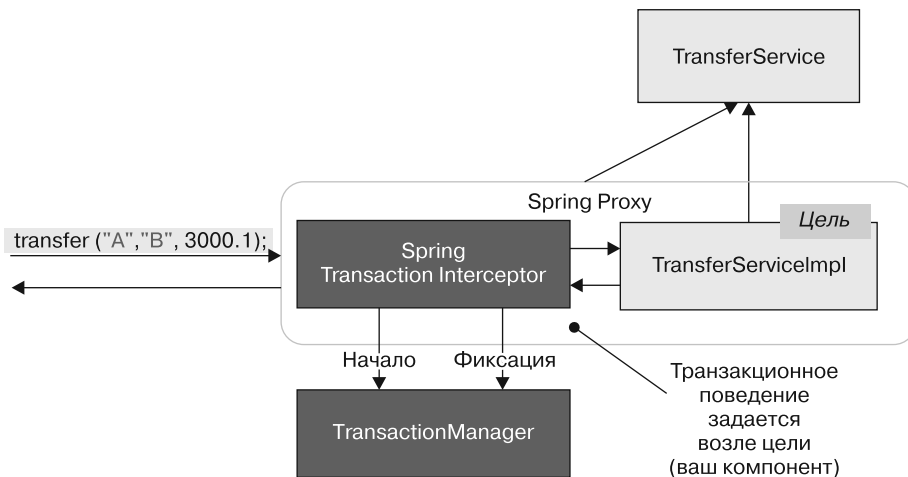


Рис. 5.9. Вызов компонента в приложении, в котором он используется

На схеме можно видеть вызов метода `service` через класс `Proxy`, создаваемый в фазе инициализации специализированным интерфейсом `BeanPostProcessor`. Он прозрачным образом добавляет в компоненты новые возможности, оберывая

их в адаптеры — динамические прокси для компонентов. При этом реализуются паттерны проектирования «Декоратор» и «Заместитель».

Реализация паттернов проектирования «Декоратор» и «Заместитель» в Spring с помощью прокси. Spring использует следующие два типа прокси в своих приложениях.

- ❑ *JDK-прокси*, известный также как динамический прокси. API этого прокси встроен в JDK. Для такого типа прокси необходим интерфейс Java.
- ❑ *CGLib-прокси*, не встроенный в JDK. Однако он включен в число JAR Spring и используется при недоступности интерфейса Java. Его нельзя применять для терминальных классов или методов.

Взглянем на возможности обоих прокси, приведенные на рис. 5.10.

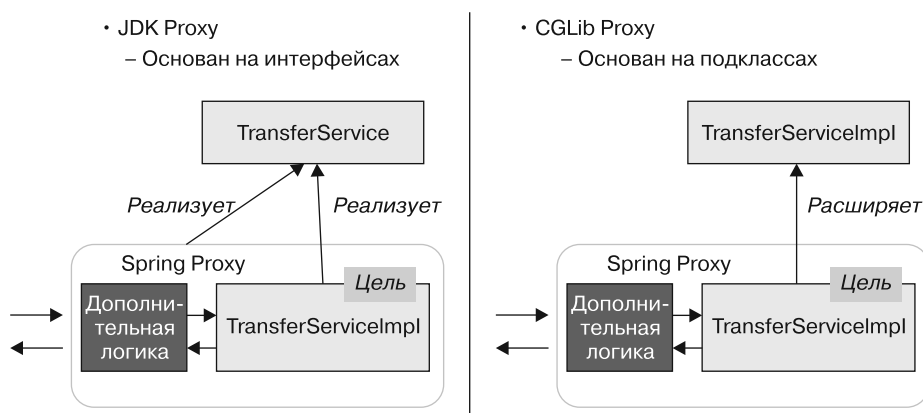


Рис. 5.10. Возможности прокси JDK и CGLib

Это вся информация о фазе использования жизненного цикла компонента Spring. Перейдем теперь к следующей фазе — уничтожения.

Фаза уничтожения компонента

В этой фазе Spring освобождает все системные ресурсы, полученные сервисами приложения, подходящие для сборки мусора. Фаза уничтожения завершается закрытием контекста приложения. Рассмотрим следующий фрагмент кода этой фазы:

```
// Некая реализация контекста приложения
```

```
ConfigurableApplicationContext applicationContext = new
AnnotationConfigApplicationContext(AppConfig.class);
```

```
// Уничтожение приложения путем закрытия его контекста
applicationContext.close();
```

Как вы думаете, что происходит в предыдущем фрагменте кода при вызове метода `applicationContext.close()`? Выполняется следующий процесс.

- ❑ Всякий реализующий интерфейс `org.springframework.beans.factory.DisposableBean` получает обратный вызов от контейнера при уничтожении. В интерфейсе `DisposableBean` содержится один-единственный метод:
`void destroy() throws Exception;`
- ❑ Если вызвать метод `destroy()` экземпляра компонента, то экземпляр будет уничтожен. У любого компонента должен быть определен метод `destroy`, то есть метод без аргументов, возвращающий тип `void`.
- ❑ Далее контекст уничтожает себя, и дальнейшее его использование невозможно.
- ❑ Только сборщик мусора на самом деле уничтожает объекты и хранит информацию об этом, он вызывается при штатном завершении работы интерфейса `ApplicationContext` или JVM и не вызывается для прототипных компонентов.

Рассмотрим теперь его реализацию с помощью XML-конфигурации:

```
<?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:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">
  <context:annotation-config/>
  <bean id="transferService"
    class="com.packt.patterninspring.chapter5.
    bankapp.service.TransferService"/>
  <bean id="accountRepository"
    class="com.packt.patterninspring.chapter5.
    bankapp.repository.JdbcAccountRepository"
    destroy-method="clearCache"/>
</beans>
```

В конфигурации у компонента `accountRepository` имеется метод типа `destroy` с названием `clearCache`:

```
package com.packt.patterninspring.chapter5.bankapp.repository;
import com.packt.patterninspring.chapter5.bankapp.model.Account;
import com.packt.patterninspring.chapter5.bankapp.model.Amount;
import com.packt.patterninspring.chapter5.bankapp.
  repository.AccountRepository;
public class JdbcAccountRepository implements AccountRepository {
  @Override
  public Account findById(Long accountId) {
    return new Account(accountId, "Arnav Rajput", new
```



```

        Amount(3000.0));
    }
    void clearCache(){
        System.out.println("Called clearCache() method");
    }
}

```

Взглянем на ту же конфигурацию в стиле Java. В Java-конфигурации можно воспользоваться атрибутом `destroyMethod` аннотации `@Bean`, как показано ниже:

```

@Bean (destroyMethod="clearCache")
public AccountRepository accountRepository() {
    return new JdbcAccountRepository();
}

```

Можно сделать то же самое с помощью конфигурации на основе аннотаций. Этот вариант требует указания пространства имен `annotation-config` или включения просмотрщика компонентов с помощью `<context:component-scan ... />`, как показано в следующем фрагменте кода:

```

public class JdbcAccountRepository {
    @PreDestroy
    void clearCache() {
        // Закрываем файлы, сетевые подключения...
        // Удаляем внешние ресурсы...
    }
}

```

Теперь вы познакомились со всеми фазами жизненного цикла компонентов Spring. В фазе инициализации применяется интерфейс `BeanPostProcessor`. В фазе использования компонентам Spring помогают чудеса прокси. Наконец, в фазе уничтожения Spring обеспечивает возможность аккуратного завершения работы приложения.

Теперь, после знакомства с жизненным циклом компонента, пришло время разобраться с областями видимости компонентов, а также с созданием пользовательских областей видимости в контейнере Spring.

Области видимости компонентов

В Spring у каждого компонента в контейнере есть своя область видимости. Существует возможность управлять не только метаданными компонента и его жизненным циклом, но и областью видимости. Можно создавать пользовательские области видимости для компонентов и регистрировать их в контейнере. Настраивать области видимости компонентов можно с помощью описаний компонентов в XML-, Java- и Annotation-конфигурациях.

Контекст приложения Spring *создает* все компоненты с одиночной областью видимости (singleton scope). Это значит, что всегда используется один и тот же компонент, вне зависимости от того, сколько раз он внедряется в другой компонент или вызывается другими сервисами. Благодаря такому поведению одиночная

область видимости снижает затраты на создание экземпляров. Это оптимальный вариант для не сохраняющих состояние объектов в приложении.

В приложении Spring иногда бывает нужно сохранять состояние части объектов, переиспользовать которые небезопасно. При таком требовании задействовать одиночную область видимости компонента также опасно в силу возможных неожиданных проблем при дальнейшем переиспользовании. В Spring имеется другая область видимости для подобных случаев, известная как *прототипная область видимости* (prototype scope).

Компонент в Spring может быть создан с одной из нескольких областей видимости, которые мы и рассмотрим далее.

Одиночная область видимости

В Spring у любого компонента с одиночной областью видимости существует только один экземпляр, созданный для контекста приложения, в котором он определяется для приложения в целом. Таково поведение контейнера Spring по умолчанию. Но оно не соответствует определению паттерна «Одиночка», данному в основополагающем руководстве по паттернам, написанном «бандой четырех». В Java одиночка означает один экземпляр конкретного класса в расчете на Classloader в JVM. А в Spring этот паттерн подразумевает экземпляр компонента в расчете на описание компонента для контейнера Spring IoC. Это продемонстрировано на следующей схеме (рис. 5.11).

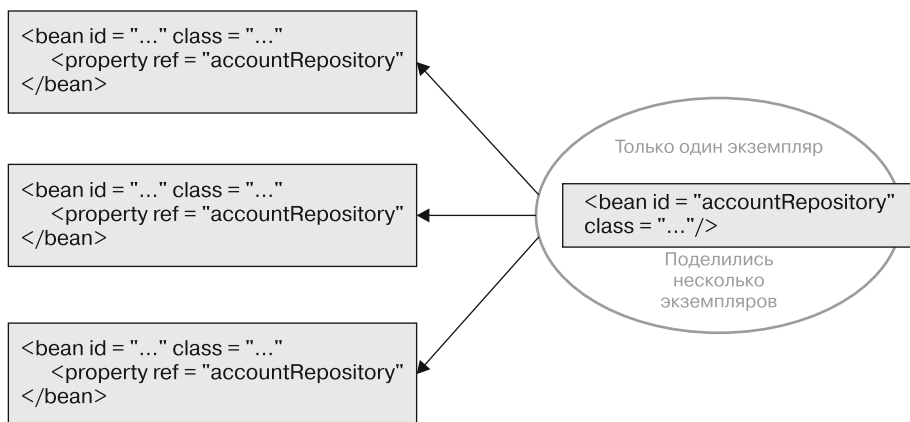


Рис. 5.11. Принцип работы одиночной области видимости

Как можно видеть, один и тот же задаваемый в описании компонента экземпляр `accountRepository` внедряется в другие компоненты одного контейнера IoC. Spring хранит все экземпляры компонентов с одиночной областью видимости в кэше, а любой из компонентов контейнера может получить из кэша ссылку на соответствующий объект.

Прототипная область видимости компонента

В Spring при каждом внедрении компонента с прототипной областью видимости в другие компоненты одного контейнера создается его экземпляр. Эта область видимости показана на рис. 5.12.

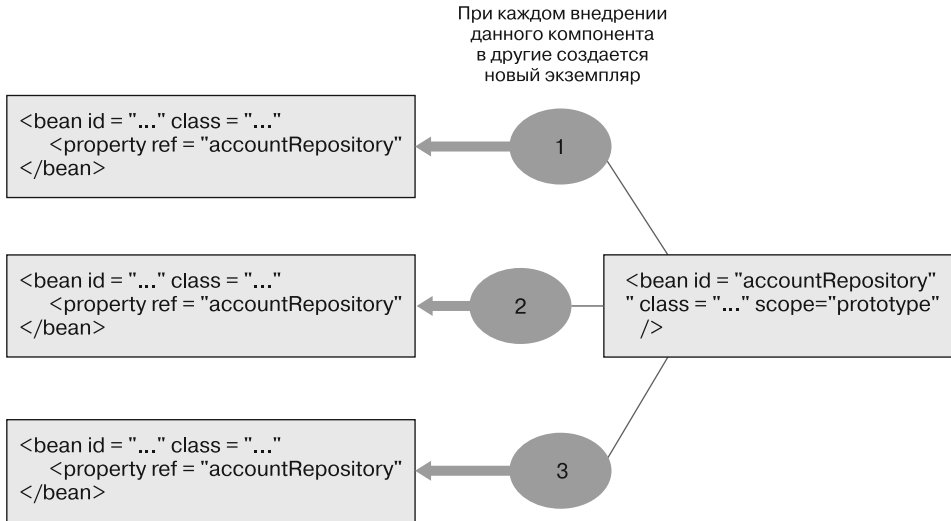


Рис. 5.12. Принцип работы прототипной области видимости

Как можно видеть на схеме, класс `accountRepository` представляет собой прототипный компонент и контейнер создает совершенно новый экземпляр при каждом его внедрении в другие компоненты.

Сеансовая область видимости компонента

Новый экземпляр создается для каждого нового пользовательского сеанса в веб-среде.

XML-конфигурация для описания компонента может при этом выглядеть следующим образом:

```
<bean id="..." class="..." scope="session"/>
```

Запросная область видимости компонента

Новый экземпляр создается для каждого нового запроса в веб-среде.

XML-конфигурация для описания компонента может при этом выглядеть таким образом:

```
<bean id="..." class="..." scope="request"/>
```

Другие области видимости в Spring

В Spring есть и другие, более узкоспециализированные, области видимости:

- ❑ область видимости веб-сокетов;
- ❑ область видимости «по обновлению»;
- ❑ потоковая область видимости (определена, но не включена по умолчанию).

Пользовательские области видимости. Фреймворк Spring также позволяет создать пользовательскую область видимости любого компонента и зарегистрировать ее в контексте приложения. Рассмотрим создание пользовательской области видимости компонента на примере.

Spring предоставляет интерфейс `org.springframework.beans.factory.config.Scope` для создания пользовательских областей видимости в контейнере Spring IoC. Для создания своей собственной области видимости необходимо реализовать этот интерфейс. Возьмем в качестве пользовательской области видимости в контейнере Spring IoC следующий класс `MyThreadScope`:

```
package com.packt.patterninspring.chapter5.bankapp.scope;
import java.util.HashMap;
import java.util.Map;
import org.springframework.beans.factory.ObjectFactory;
import org.springframework.beans.factory.config.Scope;

public class MyThreadScope implements Scope {
    private final ThreadLocal<Object> myThreadScope = new
        ThreadLocal<Object>() {
        protected Map<String, Object> initialValue() {
            System.out.println("initialize ThreadLocal");
            return new HashMap<String, Object>();
        }
    };
    @Override
    public Object get(String name, ObjectFactory<?> objectFactory) {
        Map<String, Object> scope = (Map<String, Object>)
            myThreadScope.get();
        System.out.println("getting object from scope.");
        Object object = scope.get(name);
        if(object == null) {
            object = objectFactory.getObject();
            scope.put(name, object);
        }
        return object;
    }
    @Override
    public String getConversationId() {
        return null;
    }
    @Override
    public void registerDestructionCallback(String name, Runnable callback) {
    }
}
```

```

@Override
public Object remove(String name) {
    System.out.println("removing object from scope.");
    @SuppressWarnings("unchecked")
    Map<String, Object> scope = (Map<String, Object>)
        myThreadScope.get();
    return scope.remove(name);
}
@Override
public Object resolveContextualObject(String name) {
    return null;
}
}

```

В предыдущем фрагменте кода мы переопределили несколько методов интерфейса `Scope`.

- ❑ `Object get(String name, ObjectFactory objectFactory)` — возвращает объект из нижележащей области видимости.
- ❑ `Object remove(String name)` — удаляет объект из нижележащей области видимости.
- ❑ `void registerDestructionCallback(String name, Runnable destructionCallback)` — регистрирует функции обратного вызова для уничтожения и выполняется при уничтожении заданного объекта с этой пользовательской областью видимости.

Теперь выясним, как зарегистрировать эту пользовательскую область видимости в контейнере Spring IoC и задействовать ее в приложении Spring.

Зарегистрировать пользовательскую область видимости компонента в контейнере Spring IoC можно, объявив ее с помощью класса `CustomScopeConfigurer`:

```

<?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 class="org.springframework.beans.factory.
        config.CustomScopeConfigurer">
        <property name="scopes">
            <map>
                <entry key="myThreadScope">
                    <bean class="com.packt.patterninspring.chapter5.bankapp.
                        scope.MyThreadScope"/>
                </entry>
            </map>
        </property>
    </bean>
    <bean id="myBean" class="com.packt.patterninspring.chapter5.
        bankapp.bean.MyBean" scope="myThreadScope">
        <property name="name" value="Dinesh"></property>
    </bean>
</beans>

```

Как видно из приведенного файла конфигурации, я зарегистрировал свою пользовательскую область видимости компонента с именем `MyThreadScope` в контексте приложения с помощью класса `CustomScopeConfigurer`. Данная пользовательская область видимости применяется аналогично одиночной или прототипной областям с добавлением атрибута `scope` тега `<bean>` XML-конфигурации.

Резюме

В результате чтения данной главы у вас должно было сформироваться хорошее представление о жизненном цикле компонентов Spring в контейнере, а также нескольких типах областей видимости компонентов. Теперь вы уже знаете, что существует три фазы жизненного цикла компонентов Spring в контейнере. Первая — фаза инициализации. В этой фазе Spring загружает описания компонентов из XML-, Java- или основанных на аннотациях конфигураций. После загрузки этих описаний контейнер формирует компоненты и применяет к ним логику постобработки.

Далее идет фаза использования, в которой компоненты Spring уже готовы к применению, а Spring демонстрирует чудеса паттерна «Заместитель».

Последняя фаза — фаза уничтожения. В ней при вызове приложением метода `close()` интерфейса `ApplicationContext` контейнер вызывает методы очистки всех компонентов для освобождения ресурсов.

В Spring можно управлять не только жизненным циклом компонентов, но и областью видимости компонента в контейнере. По умолчанию в контейнере Spring IoC используется одиночная область видимости, но эту настройку можно перекрыть, задав другую область видимости с помощью атрибута `scope` тега компонента в XML или аннотации `@Scope` в Java. Можно также создавать свои собственные пользовательские области видимости и регистрировать их в контейнере.

А теперь мы обратимся к самой удивительной главе этой книги, рассказывающей об *аспектно-ориентированном программировании в Spring (AOP)*. Подобно тому как внедрение зависимостей помогает расцеплять одни компоненты с другими, с которыми они взаимодействуют, AOP помогает расцеплять компоненты приложения с заданиями, охватывающими несколько компонентов приложения. Следующая глава охватывает аспектно-ориентированное программирование в Spring с помощью паттернов проектирования «Заместитель» и «Декоратор».

6

Аспектно-ориентированное программирование в Spring с помощью паттернов «Заместитель» и «Декоратор»

Прежде чем вы приступите к чтению данной главы, я хотел бы кое-чем с вами поделиться: когда я писал эту главу, моя жена Анамика делала селфи и загружала их на сайты соцсетей. Она следила за количеством полученных лайков, однако чем больше загружаешь фотографий, тем больше расходуется мобильного трафика, а мобильный трафик стоит денег. Я редко захожу в соцсети, поскольку не хочу платить лишнего интернет-провайдеру. Каждый месяц интернет-провайдер знает, какой счет нам выставить. А теперь представьте себе, что было бы, если бы мы сами тщательно планировали использование Интернета и длительность звонков и выставляли счет? Я допускаю, что некоторые одержимые интернет-пользователи так и делают, хотя понятия не имею, как им это удастся.

Подсчет расходов за использование Интернета и звонков — важная функция, но вовсе не основная для большинства пользователей. Многие пользователи, как моя жена, в основном делают селфи, загружают фотографии в социальные сети и просматривают видео на YouTube. Контроль и расчет счетов за Интернет для большинства из них не является приоритетной задачей.

Точно так же некоторые модули промышленных приложений подобны калькулятору для подсчета начислений за использование Интернета. Существуют модули с важной функциональностью, которая должна размещаться в нескольких местах приложения. Но никто не ждет, что эти функциональные возможности будут вызываться явным образом в каждой точке. Такие функции, как журналирование, безопасность и управление транзакциями, важны для приложения, но бизнес-объекты не принимают в них активного участия, поскольку должны в основном фокусироваться на тех задачах предметной области, для которых они создавались, делегируя отдельные элементы функциональности кому-то еще.

При разработке программного обеспечения приходится учитывать определенные задачи, которые необходимо выполнять в конкретных местах приложения. Эти задачи, или функции, называются *сквозной функциональностью* (cross-cutting concerns). Вся сквозная функциональность приложения должна быть отделена от

его бизнес-логики. Для этой цели Spring предоставляет модуль *аспектно-ориентированного программирования* (Aspect-Oriented Programming, AOP).

Из главы 4 вы узнали про внедрение зависимостей с целью настройки и разрешения в приложении зависимостей, реализующих общую задачу объектов. И в то время как DI способствует расцеплению объектов приложения друг с другом, модуль AOP с помощью фреймворка Spring позволяет расцеплять бизнес-логику приложения и его сквозную функциональность.

В нашем примере с банковским приложением перевод денег с одного счета на другой представляет собой бизнес-логику, а журналирование этого действия и обеспечение безопасности транзакции — сквозную функциональность нашего банковского приложения. Это значит, что журналирование, безопасность и транзакции — типичные области применения аспектов.

В этой главе мы обсудим поддержку аспектов в Spring. Она охватывает следующие вопросы.

- ❑ Паттерн «Заместитель» в Spring.
- ❑ Использование паттерна «Адаптер» при вплетении аспектов во время загрузки.
- ❑ Паттерн проектирования «Декоратор».
- ❑ Аспектно-ориентированное программирование.
- ❑ Задачи, решаемые с помощью аспектно-ориентированного программирования.
- ❑ Базовые понятия аспектно-ориентированного программирования.
- ❑ Описание срезов.
- ❑ Реализация советов.
- ❑ Работа с AOP-прокси.

Прежде чем углубиться в обсуждение модуля AOP фреймворка Spring, разберемся с паттернами, на которых он основан, и способами их применения.

Паттерн «Заместитель» в Spring

С помощью паттерна проектирования «Заместитель» можно получить объект класса, обладающий функциональностью другого класса. Этот паттерн в GoF относится к числу структурных паттернов проектирования. Его описание в GoF гласит: *«Обеспечивает заместителя для другого объекта с целью контроля доступа к нему»*. Цель этого паттерна проектирования — предоставить класс, который бы обеспечивал окружающему миру доступ к функциональности другого класса.

Проксирование классов в Spring с помощью паттерна «Декоратор». Согласно книге GoF, *благодаря динамическому добавлению объектам новых обязанностей декораторы служат гибкой альтернативой созданию производных классов с целью расширения функциональности*. Этот паттерн дает возможность добавлять и удалять функциональные возможности отдельных объектов динамически, во время

выполнения, или статистически, без изменения существующего поведения других связанных объектов из того же класса.

В модуле AOP фреймворка Spring библиотека CGLIB используется для создания в приложении прокси. Проксирование с помощью CGLIB осуществляется путем генерации (во время выполнения) класса, производного от целевого. Spring настраивает этот сгенерированный производный класс так, чтобы вызов метода передавался исходному целевому методу — производный класс применяется для реализации паттерна «Декоратор» (Decorator) влечением совета.

В Spring имеется два способа создания в приложении прокси:

- ❑ CGLIB-прокси;
- ❑ JDK-прокси (динамический прокси).

Взгляните на следующую таблицу и на рис. 6.1.

JDK-прокси	CGLIB-прокси
Также носит название динамического прокси	Не встроен в JDK
API встроен в JDK	Включен в JAR-файлы Spring
Требования: Java-интерфейс	Используется при недоступности Java-интерфейса
Все интерфейсы проксируются	Нельзя использовать для терминальных классов или методов

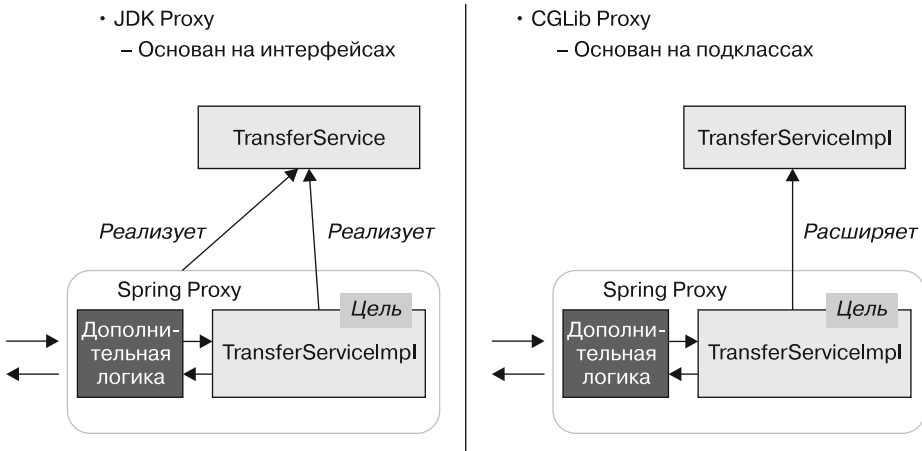


Рис. 6.1. Два способа создания в приложении прокси



Желательно помнить про один нюанс проксирования с помощью CGLIB: терминальные методы не могут быть советами, ведь их переопределение невозможно.

Что такое сквозная функциональность

В любом приложении есть какая-то универсальная функциональность, используемая во многих местах. Но эта функциональность обычно не имеет отношения к бизнес-логике приложения. Например, перед выполнением каждого бизнес-метода приложения проверяются полномочия на основе ролей. Это сквозная функциональность. Она необходима для любого приложения, но не обязательна с точки зрения бизнес-логики; это просто универсальная функциональность, которую нужно реализовать в нескольких местах приложения. Вот примеры сквозной функциональности для корпоративного приложения:

- ☐ журналирование и отладка;
- ☐ управление транзакциями;
- ☐ безопасность;
- ☐ кэширование;
- ☐ обработка ошибок;
- ☐ мониторинг производительности;
- ☐ пользовательские бизнес-правила.

Теперь попробуем реализовать эту сквозную функциональность в нашем приложении с помощью аспектов модуля AOP фреймворка Spring.

Что такое аспектно-ориентированное программирование

Как уже упоминалось выше, *аспектно-ориентированное программирование (АОП)* обеспечивает возможность модульной организации сквозной функциональности. Оно дополняет другую парадигму программирования — *объектно-ориентированное программирование (ООП)*. Ключевыми элементами ООП являются классы и объекты, а АОП — аспекты. Аспекты дают возможность модульной организации элементов функциональности в различных местах по всему приложению. Подобная функциональность носит название *сквозной* (cross-cutting concerns). Одним из примеров подобной функциональности может служить безопасность приложения, поскольку подобные возможности необходимо обеспечивать во множестве методов. Аналогично к сквозной функциональности для приложения относятся журналирование и управление транзакциями, а также многое другое. Рассмотрим на рис. 6.2 применение этой функциональности к бизнес-модулям.

Как можно видеть, в приложении есть три основных бизнес-модуля: `TransferService`, `AccountService` и `BankService`. Для каждого требуется определенная общая функциональность, такая как *безопасность*, *управление транзакциями* и *журналирование*. Взглянем, с какими проблемами мы столкнемся в приложении, если не будем использовать модуль AOP фреймворка Spring.

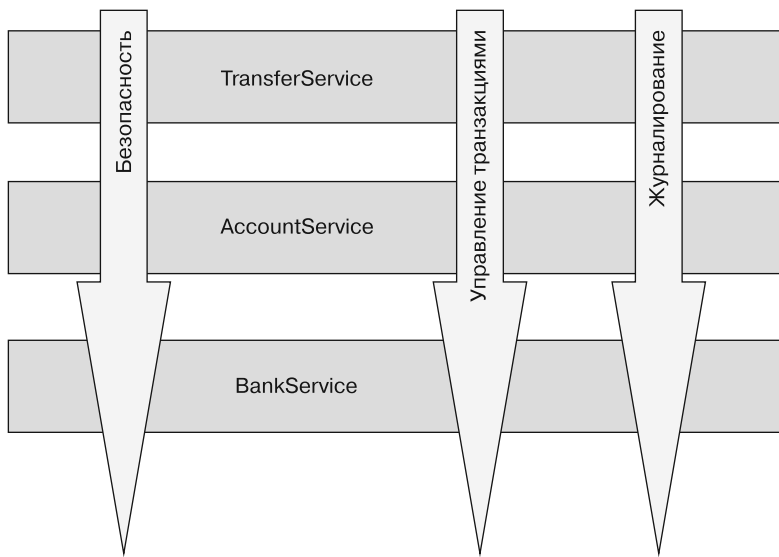


Рис. 6.2. Применение сквозной функциональности к бизнес-модулям

Проблемы, решаемые с помощью АОР

Как говорилось выше, аспекты делают возможной модульную организацию сквозной функциональности. Так что, если не задействовать аспекты, модульная организация сквозной функциональности будет невозможна. Это зачастую ведет к смешению сквозной функциональности с бизнес-модулями. В случае обычной объектно-ориентированной парадигмы для переиспользования общих элементов функциональности, например для обеспечения безопасности, журналирования и управления транзакциями, *приходится* применять наследование или композицию. Но использование в этом случае наследования может привести к нарушению одного из принципов SOLID — принципа единственной ответственности, а также может повысить сложность иерархии объектов. Кроме того, композиция может оказаться непростой задачей в масштабах приложения. Это значит, что невозможность модульной организации сквозной функциональности ведет к следующим двум основным проблемам:

- ❑ к спутыванию кода (code tangling);
- ❑ разбрасыванию кода (code scattering).

Спутывание кода

Эта проблема представляет собой сцепление функциональности в приложении. Спутывание кода означает смешивание сквозной функциональности с бизнес-логикой приложения. Оно ведет к сильному сцеплению между модулями сквозной

функциональности и бизнес-модулями. Чтобы лучше понять, что такое спутывание кода, рассмотрим следующий код:

```
public class TransferServiceImpl implements TransferService {
    public void transfer(Account a, Account b, Double amount) {
        // Начало связанной с безопасностью функциональности
        if (!hasPermission(SecurityContext.getPrincipal())) {
            throw new AccessDeniedException();
        }
        // Конец связанной с безопасностью функциональности
        // Начало бизнес-логики
        Account aAct = accountRepository.findByAccountId(a);
        Account bAct = accountRepository.findByAccountId(b);
        accountRepository.transferAmount(aAct, bAct, amount);
        ...
    }
}
```

Как видно из этого фрагмента, код обеспечения безопасности (отмечен комментариями) смешивается с кодом бизнес-логики. Такая ситуация — типичный пример спутывания кода. В этот пример мы включили только код для обеспечения безопасности, но в реальном корпоративном приложении необходимо реализовывать множество элементов сквозной функциональности, например журналирование, управление транзакциями и т. п. В подобных случаях организация кода и внесение в него изменений еще более усложняются, что может стать причиной появления критических ошибок в коде (рис. 6.3).

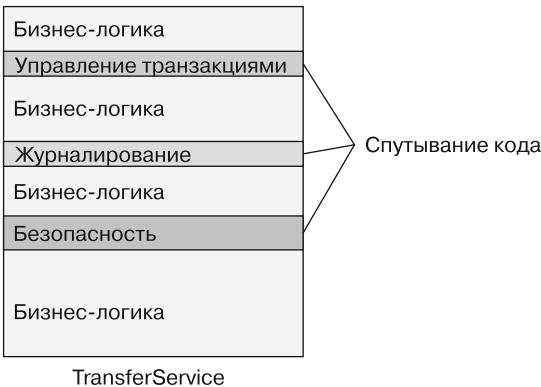


Рис. 6.3. Три элемента сквозной функциональности

На рисунке можно видеть три элемента сквозной функциональности, разбросанные по бизнес-классу `TransferService`, и смешанную с бизнес-логикой класса `AccountService` логику сквозной функциональности. Подобное сцепление между сквозной функциональностью и логикой приложения называется *спутыванием кода*. Рассмотрим еще одну серьезную проблему, возникающую в том случае, если мы не используем аспекты для сквозной функциональности.

Разбрасывание кода

Разбрасывание кода означает, что один и тот же элемент функциональности разбросан по множеству модулей приложения. Это способствует дублированию кода таких элементов в модулях приложения. Рассмотрим следующий фрагмент кода, чтобы лучше понять, что такое разбрасывание кода:

```
public class TransferServiceImpl implements TransferService {
    public void transfer(Account a, Account b, Double amount) {
        // Начало связанной с безопасностью функциональности
        if (!hasPermission(SecurityContext.getPrincipal())) {
            throw new AccessDeniedException();
        }
        // Здесь связанная с безопасностью функциональность заканчивается
        // Здесь начинается бизнес-логика
        ...
    }
}

public class AccountServiceImpl implements AccountService {
    public void withdrawl(Account a, Double amount) {
        // Здесь начинается связанная с безопасностью функциональность
        if (!hasPermission(SecurityContext.getPrincipal())) {
            throw new AccessDeniedException();
        }
        // Конец связанной с безопасностью функциональности
        // Начало бизнес-логики
        ...
    }
}
```

Как вы можете видеть из предыдущего кода, в приложении есть два модуля: `TransferService` и `AccountService`. В обоих присутствует один и тот же код сквозной функциональности, обеспечивающей безопасность. Выделенный жирным шрифтом код в обоих модулях одинаков, а значит, мы имеем дело с дублированием кода. Рисунок 6.4 иллюстрирует разбрасывание кода.

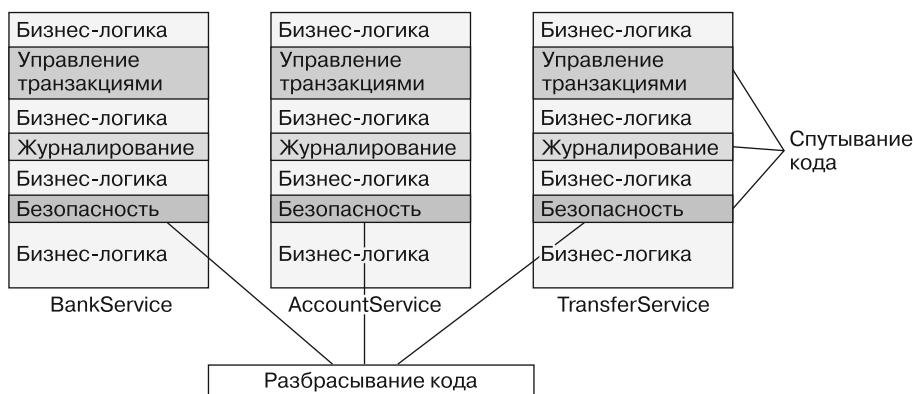


Рис. 6.4. Разбрасывание кода

На предыдущей схеме показаны три бизнес-модуля: `TransferService`, `AccountService` и `BankService`. Каждый из них содержит сквозную функциональность: *безопасность*, *журналирование* и *управление транзакциями*. Код этих элементов сквозной функциональности во всех модулях совпадает. Фактически это дублирование кода функциональности по приложению.

Модуль AOP фреймворка Spring предлагает решение проблем спутывания и разбрасывания кода в приложении. Аспекты открывают возможность модульной организации сквозной функциональности, позволяя избежать спутывания и исключить разбрасывание кода. В следующих разделах мы изучим, как именно AOP помогает в этом.

Как AOP решает проблемы

Модуль AOP Spring дает возможность разделять сквозную функциональность с основной логикой приложения. Это значит, что вы можете реализовывать основную логику приложения, сосредотачивая свое внимание на его главных задачах. Кроме того, вы можете писать аспекты для реализации сквозной функциональности. Spring предлагает множество готовых аспектов. После создания аспектов можно их (а по факту — логику сквозной функциональности) вставить в нужные места приложения. Рассмотрим рис. 6.5.

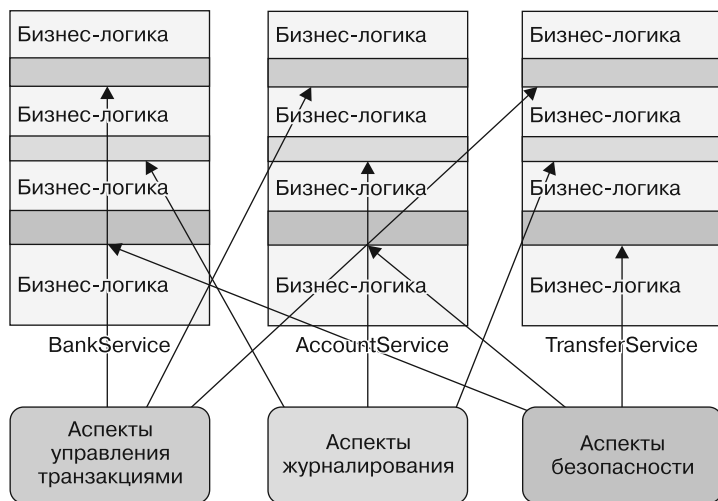


Рис. 6.5. Функциональность AOP

Как вы можете видеть, аспекты безопасности, журналирования и управления транзакциями реализуются в приложении отдельно. Мы добавили их в нужные места приложения. Теперь логика нашего приложения отделена от сквозной функциональности. В следующем разделе вы познакомитесь с основными понятиями и терминологией AOP применительно к фреймворку Spring.

Основные понятия и терминология AOP

Как и у других технологий, у AOP есть своя терминология. Начнем с изучения нескольких базовых понятий и терминов AOP. В Spring парадигма аспектно-ориентированного программирования используется для модуля Spring AOP. Но, к сожалению, в Spring AOP применяется специфическая для Spring терминология. Используемые для описания модулей и возможностей AOP термины нельзя назвать интуитивно понятными. Несмотря на это, они необходимы для изучения AOP. Без понимания идиомы AOP вы не сможете разобраться с функциональностью модуля AOP. По сути, AOP описывается в категориях советов (advices), срезов и точек соединения. Рассмотрим рис. 6.6.

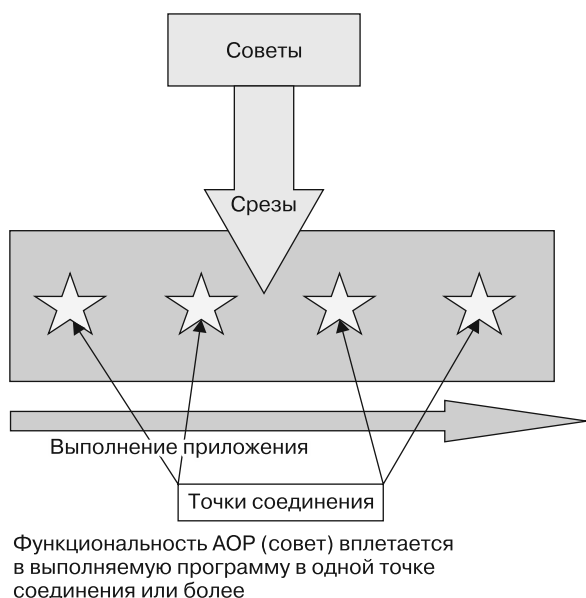


Рис. 6.6. Основные понятия AOP и их взаимосвязь во фреймворке

Итак, к функциональности AOP относятся *советы* (advices), реализованные в нескольких точках. Эти точки, называемые *точками соединения* (join points), описываются с помощью выражений — *срезов* (pointcuts). Разберемся с этими понятиями подробнее на примере.

Совет

Тарифный план используется интернет-провайдером для подсчета начислений в соответствии с потребленным трафиком в мегабайтах или гигабайтах. У интернет-провайдера имеется список пользователей, и компания выставляет им счета за Интернет. Таким образом, для провайдера расчет начислений — базовая функция,

а для пользователей — нет. Аналогично у каждого аспекта есть основное задание, в выполнении которого и состоит его назначение. Это задание аспекта называется в АОР *советом*.

Как вы уже знаете, совет представляет собой конкретное выполняемое задание, так что возникают вопросы: когда это задание выполнять и что оно в себя включает? Выполнять ли это задание перед вызовом бизнес-метода? Или после вызова бизнес-метода? Или и до, и после? Или его нужно выполнять, когда бизнес-метод сгенерирует исключение? Иногда такой бизнес-метод называют также *советуемым методом* (advised method). Рассмотрим следующие пять видов советов, используемых аспектами Spring.

❑ *До* (Before) — задание совета выполняется до вызова советуемого метода.



В случае, если совет генерирует исключение, целевой метод не вызывается — это вполне допустимый вариант использования совета типа *до*.

❑ *После* (After) — задание совета выполняется после завершения советуемого метода, независимо от того, сгенерировал ли целевой метод исключение.

❑ *После возврата* (After-returning) — задание совета выполняется после успешного завершения советуемого метода. Например, при возврате из бизнес-метода без генерации исключения.

❑ *После исключения* (After-throwing) — задание совета выполняется при выходе из советуемого метода посредством генерации исключения.

❑ *Везде* (Around) — один из обладающих наиболее широкими возможностями советов Spring АОР. Он охватывает советуемый метод с обеих сторон, выполняя части задания совета до и после вызова советуемого метода.

Если коротко, то код задания совета должен выполняться в каждой из выбранных точек, называемых точками соединения, — еще один термин АОР, с которым мы познакомимся в следующем подразделе.

Точка соединения

Интернет-провайдер предоставляет возможность подключения к Сети множеству пользователей. У каждого из них есть тарифный план, используемый для расчета начислений. На основе тарифных планов интернет-провайдер может выставить счета за Интернет для всех своих пользователей. Аналогично в приложении может быть множество точек, в которых можно применить советы. Эти места в приложении называются *точками соединения* (join points). Точка соединения — это место в потоке выполнения программы, например вызов метода или генерация исключения. В этих точках аспекты Spring внедряют сквозную функциональность в приложение. Разберемся, откуда АОР знает о точках соединения, и поговорим еще об одном понятии АОР.

Срез

Интернет-провайдер создает несколько тарифных планов в соответствии с объемами использования Интернета (таким пользователям, как моя жена, необходимо больше трафика), поскольку никакая компания не смогла бы создать единый план, подходящий для всех пользователей, или уникальный тарифный план для каждого — вместо этого определенный тарифный план закрепляется за некоторым подмножеством пользователей. Аналогичным образом совет не обязательно применяется ко всем точкам соединения в приложении. Существует возможность задавать выражения для выбора одной или нескольких точек соединения. Эти выражения носят название *срезов* (pointcut). Срезы помогают уменьшить количество точек соединения, куда аспект позволяет внедрить совет. И мы плавно переходим к следующему понятию AOP — аспекту.

Аспект

Интернет-провайдеру известно, какой тарифный план у какого пользователя. На основе этой информации интернет-провайдер формирует счет за Интернет и отправляет его пользователю. В этом примере провайдер является аспектом, тарифные планы — срезами, пользователи — точками соединения, а сформированные провайдером счета за Интернет — советами. Аналогично этому аспект в приложении — модуль, инкапсулирующий срезы и совет. Аспект «знает», что, где и когда он делает в приложении. Посмотрим, как AOP применяет аспекты к бизнес-методам.

Вплетение

Вплетение (weaving) — метод, с помощью которого аспекты сочетаются с бизнес-кодом. Это процесс применения аспектов к целевому объекту с использованием нового проксирующего объекта. Вплетение может производиться во время компиляции или загрузки классов, а также во время выполнения. Вплетение во время выполнения в Spring AOP осуществляется с помощью паттерна проектирования «Заместитель».

Вы уже познакомились со многими понятиями AOP. Все они пригодятся вам при изучении любого фреймворка AOP, вне зависимости от того, идет речь об AspectJ или Spring AOP. В Spring использовался фреймворк AspectJ при реализации фреймворка Spring AOP. Последний поддерживает ограниченный набор возможностей AspectJ. Он также предоставляет вариант AOP на основе прокси. При этом Spring поддерживает точки соединения только для методов.

Теперь, когда у вас есть представление о Spring AOP и том, как оно работает, мы можем перейти к следующим вопросам, в частности к описанию срезов в декларативной модели AOP фреймворка Spring.

Задание срезов

Как уже упоминалось, срезы используются для задания точек применения советов, поэтому срез — один из важнейших элементов аспекта. Разберемся с описанием срезов. В Spring AOP для описания срезов можно использовать язык выражений. Модуль AOP позволяет описывать срезы AspectJ для выбора мест применения советов с помощью специального языка выражений. Spring AOP поддерживает лишь некоторое подмножество имеющихся в AspectJ дескрипторов срезов, поскольку в основе Spring AOP лежит использование прокси, а некоторые дескрипторы не поддерживают AOP на основе прокси. В следующей таблице приведены поддерживаемые Spring AOP дескрипторы.

Поддерживаемые модулем AOP фреймворка Spring дескрипторы	Описание
execution	Основной дескриптор срезов Spring AOP. Подбирает точки соединения, соответствующие вызовам методов
within	Выбирает только те точки соединения, где исполняемый код описан в одном из указанных типов
this	Ограничивает выбор только теми точками соединения, где ссылка на компонент относится к указанному типу
target	Ограничивает выбор только точками соединения с заданным типом целевого объекта
args	Ограничивает выбор только точками соединения с указанными типами аргументов
@target	Ограничивает выбор только точками соединения, в которых целевой объект аннотирован указанным типом
@args	Ограничивает выбор только точками соединения, в которых у фактического типа передаваемых во время выполнения аргументов имеются аннотации указанного типа
@within	Ограничивает выбор только точками соединения, в которых у объявленного типа целевого объекта есть аннотация заданного типа
@annotation	Ограничивает выбор только точками соединения, помеченными указанной аннотацией

Как уже упоминалось, основным среди поддерживаемых Spring дескрипторов срезов является дескриптор выполнения (execution), поэтому здесь я продемонстрирую описание срезов только с помощью таких дескрипторов.

Написание кода срезов. С помощью дескриптора выполнения код срезов можно написать следующим образом.

- ❑ `execution(<шаблон_метода>)` — метод должен соответствовать приведенному шаблону.

- ❑ Возможно соединение цепочкой для создания составных срезов с помощью следующих операторов: && (and), || (or), ! (not).
- ❑ Шаблон метода:
 - [Модификаторы] ВозвращаемыйТип [ТипКласса];
 - ИмяМетода ([Аргументы]) [throws ТипИсключения].

В вышеприведенном шаблоне метода значения, заключенные в квадратные скобки, — модификаторы, тип класса, аргументы и исключения — необязательны. Задавать их для всех срезов с помощью дескриптора выполнения не нужно. Значения без квадратных скобок, например возвращаемый тип и имя метода, обязательны.

Опишем интерфейс `TransferService`:

```
package com.packt.patterninspring.chapter6.bankapp.service;
public interface TransferService {
    void transfer(String accountA, String accountB, Long amount);
}
```

Интерфейс `TransferService` представляет собой сервис для перевода денежных сумм с одного счета на другой. Предположим, вам нужно написать аспект для журналирования, который бы вызывал метод `transfer()` интерфейса `TransferService`. Следующая схема иллюстрирует выражение для среза, подходящее для применения совета при каждом выполнении метода `transfer()` (рис. 6.7).

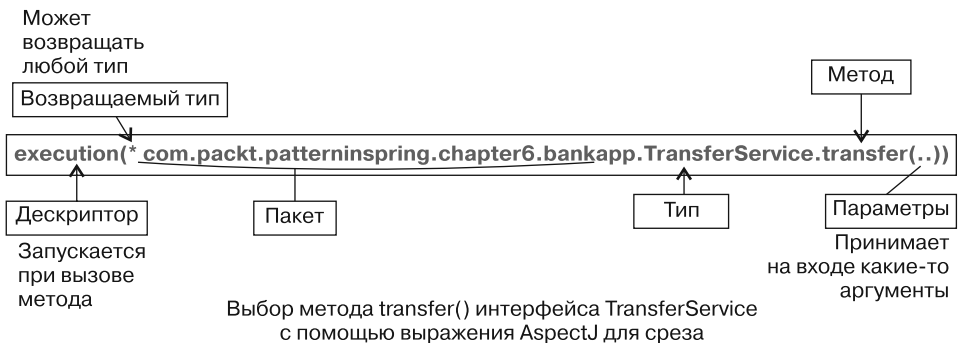


Рис. 6.7. Выражение для среза

Как вы можете видеть, я воспользовался дескриптором `execution()` для выбора точки соединения, соответствующей методу `transfer()` интерфейса `TransferService`. Я указал символ `*` в начале выражения. Это значит, что метод может возвращать любой тип. А после символа `*` я привел полное имя класса и имя метода `transfer()`. В качестве аргументов метода я указал две точки, то есть срез может выбрать метод с именем `transfer` без параметров или с произвольным количеством параметров.

Рассмотрим еще несколько выражений срезов для выбора точек соединения.

- ❑ Произвольный класс или пакет:
 - `execution(void transfer*(String))` — произвольный метод, начинающийся с `transfer`, с одним параметром типа `String` и возвращаемым типом `void`;
 - `execution(* transfer(*))` — произвольный метод с названием `transfer`, принимающий на входе один параметр;
 - `execution(* transfer(int, ..))` — произвольный метод с названием `transfer`, первый параметр которого имеет тип `int` (здесь `..` означает, что за ним может следовать несколько других параметров или ни одного).
- ❑ Ограничение по классу: `execution(void com.packt.patterninspring.chapter6.bankapp.service.TransferServiceImpl.*(..))` — произвольный метод из класса `TransferServiceImpl`, возвращающий `void`, включая любой подкласс, но не в случае использования другой реализации.
- ❑ Ограничение по интерфейсу: `execution(void com.packt.patterninspring.chapter6.bankapp.service.TransferService.transfer(*))` — произвольный метод с названием `transfer()`, принимающий один аргумент и возвращающий `void` из любого реализующего интерфейс `TransferService` объекта. Это вариант, обеспечивающий большую гибкость — он работает даже при изменении реализации.
- ❑ Использование аннотаций: `execution(@javax.annotation.security.RolesAllowed void transfer*(..))` — произвольный возвращающий `void` метод, название которого начинается с `transfer`. Аннотирован `@RolesAllowed`.
- ❑ Работа с пакетами:
 - `execution(* com..bankapp.*.*(..))`: `com` и `bankapp` может отделять один каталог;
 - `execution(* com.*.bankapp.*.*(..))`: `com` и `bankapp` может разделять несколько каталогов;
 - `execution(* *.bankapp.*.*(..))`: произвольный подпакет с названием `bankapp`.

Теперь, когда вы научились писать простые срезы, предлагаю перейти к написанию кода советов, а также объявлению аспектов, использующих эти срезы.

Создание аспектов

Аспекты — одно из важнейших понятий АОР. Они комбинируют в приложении срезы и советы. Разберемся, как описать аспект в приложении.

Вы уже описывали интерфейс `TransferService` в качестве объекта срезов вашего аспекта. Теперь воспользуемся аннотациями `AspectJ` для создания аспекта.

Описание аспектов с помощью аннотаций. Предположим, вы хотели бы генерировать в своем банковском приложении журнал переводов денег для целей аудита и отслеживания действий, чтобы лучше понимать поведение клиентов. Любое коммерческое предприятие обречено на провал, если оно не понимает своих

клиентов. Если взглянуть на этот сервис с точки зрения бизнеса, аудит необходим, но не является ключевым для функционирования бизнеса, это отдельная функциональность. Следовательно, имеет смысл задать аудит в виде аспекта, применяемого к сервису перевода денег. Рассмотрим код, в котором приведен класс `Auditing`, описывающий аспекты для данной функциональности:

```
package com.packt.patterninspring.chapter6.bankapp.aspect;

import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class Auditing {

    // До перевода средств
    @Before("execution(* com.packt.patterninspring.chapter6.bankapp.
        service.TransferService.transfer(..))")
    public void validate(){
        System.out.println("банк проверяет ваши учетные данные
            до перевода средств");
    }

    // До перевода средств
    @Before("execution(* com.packt.patterninspring.chapter6.bankapp.
        service.TransferService.transfer(..))")
    public void transferInstantiate(){
        System.out.println("банк создает экземпляр сервиса перевода средств");
    }

    // После перевода средств
    @AfterReturning("execution(* com.packt.patterninspring.chapter6.
        bankapp.service.TransferService.transfer(..))")
    public void success(){
        System.out.println("банк успешно выполнил перевод средств");
    }

    // После неудачного перевода средств
    @AfterThrowing("execution(* com.packt.patterninspring.chapter6.
        bankapp.service.TransferService.transfer(..))")
    public void rollback() {
        System.out.println("банк откатил операцию перевода средств");
    }
}
```

Как вы можете видеть, класс `Auditing` снабжен аннотацией `@Aspect`. А значит, этот класс — не просто компонент Spring, а аспект приложения. Методы класса `Auditing` являются советами, заключающими в себе определенную логику. Как известно, до перевода денег с одного счета на другой банк проверяет (`validate()`) используемые учетные данные, после чего создает экземпляр соответствующего

сервиса (`transferInstantiate()`). После успешной проверки учетных данных (`success()`) банк переводит нужную сумму и выполняет аудит. Но в случае любого сбоя при переводе банк должен откатить (`rollback()`) операцию.

Как вы можете видеть, все методы класса `Auditing` снабжены аннотациями советов, указывающими, когда их следует вызывать. Spring AOP предоставляет пять типов аннотаций для описания советов. Они перечислены в следующей таблице.

Аннотация	Совет
@Before	Используется для совета типа «до», метод совета выполняется до вызова советуемого метода
@After	Применяется для совета типа «после», метод совета выполняется после (неважно, успешного или завершившегося нештатно) вызова советуемого метода
@AfterReturning	Используется для совета типа «после возврата», метод совета выполняется после успешного выполнения советуемого метода
@AfterThrowing	Применяется для совета типа «после исключения», метод совета выполняется после того, как выполнение советуемого метода завершилось аварийно (при генерации исключения)
@Around	Используется для совета типа «везде», метод совета выполняется и до, и после вызова советуемого метода

Теперь взглянем на реализации этих советов и их функционирование в приложении.

Реализация советов

Как вы уже знаете, Spring дает возможность использования пяти типов советов. Рассмотрим их по очереди.

Тип совета: до

Рассмотрим следующую схему для совета типа «до» (рис. 6.8). Этот совет выполняется до целевого метода.

Как вы видите, сначала выполняется совет типа «до», а потом целевой метод. Как известно, Spring AOP основывается на использовании прокси. Поэтому на базе паттернов проектирования «Заместитель» и «Декоратор» создается объект-прокси целевого класса.

Пример. Посмотрим на использование аннотации `@Before`:

```
// До перевода средств
@Before("execution(* com.packt.patterninspring.chapter6.
    bankapp.service.TransferService.transfer(..)")
public void validate(){
```

```

System.out.println("банк проверяет ваши учетные данные
до перевода средств");
}

// До перевода средств
@Before("execution(* com.packt.patterninspring.chapter6.
    bankapp.service.TransferService.transfer(..)")
public void transferInstantiate(){
    System.out.println("банк создает экземпляр сервиса перевода средств");
}

```

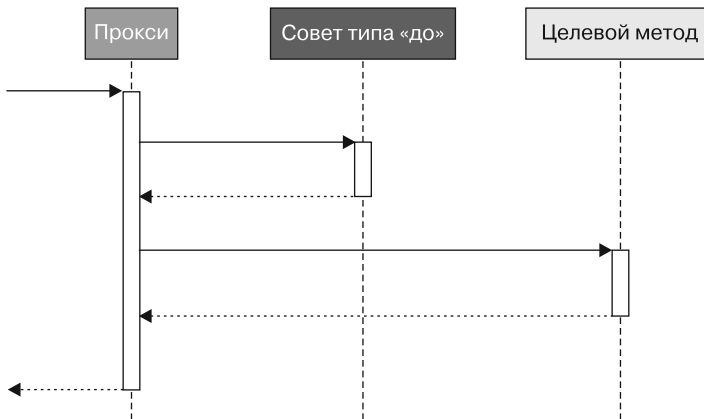


Рис. 6.8. Тип совета: до



Если совет генерирует исключение, целевой метод не вызывается — это вполне допустимый вариант использования совета типа «до».

Тип совета: после возврата

Рассмотрим следующую схему для совета типа «*после возврата*» (рис. 6.9). Этот совет выполняется после успешного выполнения целевого метода.

Как вы можете видеть, совет типа «*после возврата*» выполняется после успешного возврата из целевого метода. Он никогда не выполняется при генерации целевым методом какого-либо исключения.

Пример. Посмотрим на использование аннотации `@AfterReturning`:

```

// После успешного перевода денег
@AfterReturning("execution(* com.packt.patterninspring.chapter6.
    bankapp.service.TransferService.transfer(..)")
public void success(){
    System.out.println("банк успешно выполнил перевод средств");
}

```

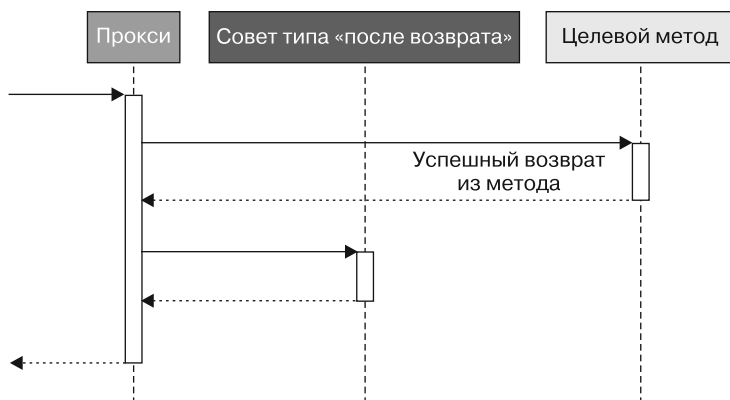


Рис. 6.9. Тип совета: после возврата

Тип совета: после исключения

Рассмотрим рис. 6.10, иллюстрирующий совет типа «*после исключения*». Этот совет выполняется после нештатного завершения целевого метода. Иначе говоря, он выполняется при генерации целевым методом какого-либо исключения.

Совет типа «*после исключения*» никогда не будет выполняться, если целевой метод не сгенерировал никакого исключения.

Пример. Посмотрим на использование аннотации `@AfterThrowing`:

```
// После неудачного перевода средств
@AfterThrowing("execution(* com.packt.patterninspring.chapter6.
    bankapp.service.TransferService.transfer(..)")
public void rollback() {
    System.out.println("банк откатил операцию перевода средств ");
}
```

Можно также использовать аннотацию `@AfterThrowing` с атрибутом, задающим тип исключения, при этом совет будет вызываться только при генерации соответствующего типа исключения:

```
// После неудачного перевода средств
@AfterThrowing(value = "execution(*com.packt.patterninspring.chapter6.
    bankapp.service.TransferService.transfer(..)", throwing="e")
public void rollback(DataAccessException e) {
    System.out.println("банк откатил операцию перевода средств ");
}
```

Выполняется каждый раз, когда `TransferService` генерирует исключение типа `DataAccessException`.



Совет типа `@AfterThrowing` не блокирует дальнейшее распространение исключения. Однако он может инициировать генерацию исключения другого типа.

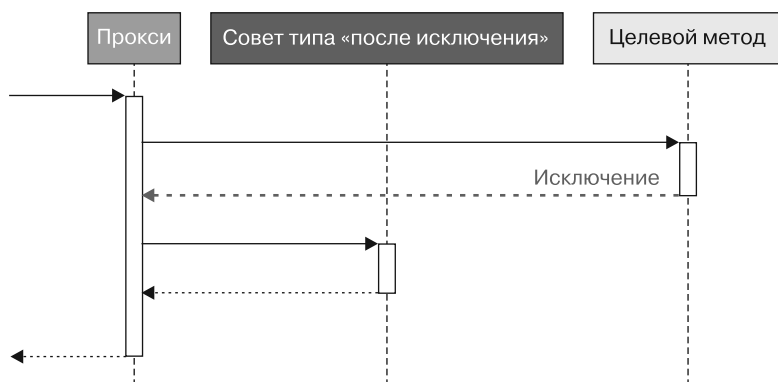


Рис. 6.10. Тип совета: после исключения

Тип совета: после

Рассмотрим следующую схему для совета типа «после». Этот совет выполняется после завершения — независимо, штатного или нештатного — работы целевого метода. Не имеет значения, генерирует целевой метод какое-либо исключение или выполняется без исключений (рис. 6.11).

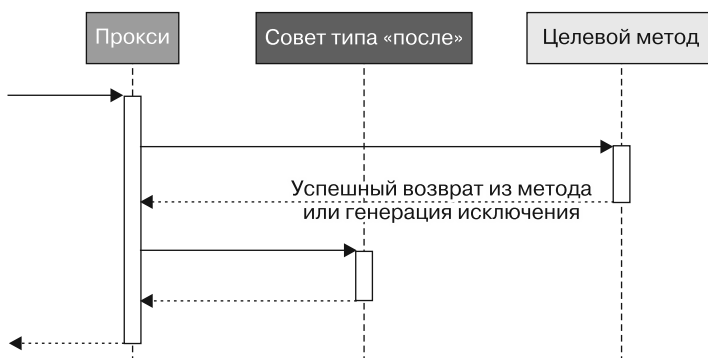


Рис. 6.11. Тип совета: после

Пример. Посмотрим на использование аннотации `@After`:

```
// После попытки перевода средств
@After ("execution(* com.packt.patterninspring.chapter6.
    bankapp.service.TransferService.transfer(..)")
public void trackTransactionAttempt(){
    System.out.println("банк попытался провести транзакцию");
}
```

Метод, аннотированный `@After`, вызывается независимо от того, сгенерировал целевой метод исключение или нет.

Тип совета: везде

Рассмотрим следующую схему для совета типа «везде» (рис. 6.12). Он выполняется как до, так и после вызова целевого метода. Этот совет Spring AOP обладает очень широкими возможностями. Множество функций фреймворка Spring реализовано с помощью этого совета. Это единственный совет в Spring, который может останавливать или возобновлять выполнение целевого метода.

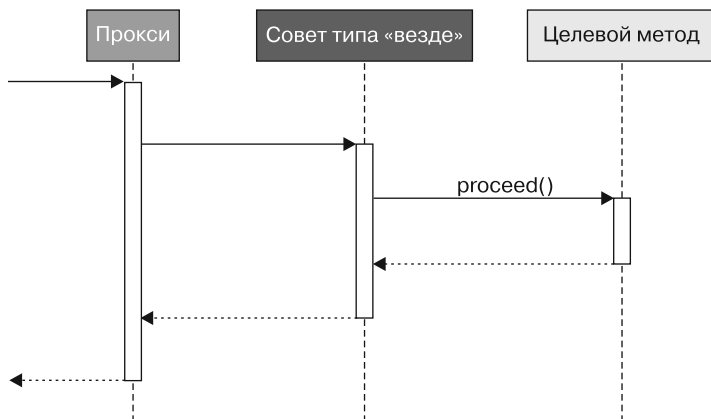


Рис. 6.12. Тип совета: везде

Как вы можете видеть на рис. 6.12, совет типа «везде» выполняется дважды, первый раз перед советуемым методом, а второй — после его вызова. Этот совет также вызывает¹ метод `proceed()` для выполнения советуемого метода.

Пример. Посмотрим на использование аннотации `@Around`:

```

@Around(execution(* com.packt.patterninspring.chapter6.
    bankapp.service.TransferService.createCache(..)))
public Object cache(ProceedingJoinPoint point){
    Object value = cacheStore.get(CacheUtils.toKey(point));
    if (value == null) {
        value = point.proceed();
        cacheStore.put(CacheUtils.toKey(point), value);
    }
    return value;
}

```

В этом фрагменте кода я воспользовался аннотацией `@Around` и интерфейсом `ProceedingJoinPoint`, который наследует интерфейс `JoinPoint`, добавляя метод `proceed()`. Как вы можете видеть, данный тип совета возобновляет выполнение целевого метода только в том случае, если значение пока не закэшировано.

¹ На самом деле не обязательно. С помощью данного типа совета можно вообще исключить вызов целевого метода.

Итак, ранее вы уже видели, как реализовывать советы в приложении, создавать аспекты и задавать срезы с помощью аннотаций. В этом примере в качестве класса аспекта выступал `Auditing`, снабженный аннотацией `@Aspect`, однако она не станет работать, если не активизировать возможности AOP-проксирования в Spring.

Рассмотрим следующий файл Java-конфигурации `AppConfig.java`: автоматическое проксирование можно включить, применив аннотацию `@EnableAspectJAutoProxy` на уровне класса.

```
package com.packt.patterninspring.chapter6.bankapp.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

import com.packt.patterninspring.chapter6.bankapp.aspect.Auditing;

@Configuration
@EnableAspectJAutoProxy
@ComponentScan

public class AppConfig {
    @Bean
    public Auditing auditing() {
        return new Auditing();
    }
}
```

При использовании XML-конфигурации для налаживания взаимодействия компонентов в Spring и активизации возможностей Spring AOP можно задействовать элемент `<aop:aspectj-autoproxy>` из пространства имен AOP 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:aop="http://www.springframework.org/schema/aop"
    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">
    <context:component-scan base-
        package="com.packt.patterninspring.chapter6.bankapp" />
    <aop:aspectj-autoproxy />
    <bean class="com.packt.patterninspring.chapter6.
        bankapp.aspect.Auditing" />
</beans>
```

Перейдем теперь к объявлению аспектов в файле XML-конфигурации фреймворка Spring.

Описание аспектов с помощью XML-конфигурации

Как вы уже знаете, задавать настройки компонентов можно в XML-конфигурации аналогично объявлению в ней аспектов. Spring предоставляет для объявления аспектов в XML еще одно пространство имен AOP и множество элементов, приведенных в следующей таблице.

Аннотация	Соответствующий XML-элемент	Назначение XML-элемента
@Before	<aop:before>	Описывает совет типа «до»
@After	<aop:after>	Описывает совет типа «после»
@AfterReturning	<aop:after-returning>	Описывает совет типа «после возврата»
@AfterThrowing	<aop:after-throwing>	Описывает совет типа «после исключения»
@Around	<aop:around>	Описывает совет типа «везде»
@Aspect	<aop:aspect>	Описывает аспект
@EnableAspectJAutoProxy	<aop:aspectj-autoproxy>	Делает возможным использование основанных на аннотациях аспектов с помощью @AspectJ
@Pointcut	<aop:pointcut>	Описывает срез
—	<aop:advisor>	Описывает объект класса Advisor
—	<aop:config>	Верхний уровень элементов AOP

Для многих элементов пространства имен AOP в Java-конфигурациях доступна соответствующая аннотация. Рассмотрим приведенный выше пример для XML-конфигурации, но сначала посмотрим на класс аспекта `Auditing`. Удалим все `AspectJ`-аннотации, как показано в следующем фрагменте кода:

```
package com.packt.patterninspring.chapter6.bankapp.aspect;
```

```
public class Auditing {
    public void validate(){
        System.out.println("банк проверяет ваши учетные данные
        до перевода средств");
    }

    public void transferInstantiate(){
        System.out.println("банк создает экземпляр сервиса перевода средств");
    }

    public void success(){
        System.out.println("банк успешно выполнил перевод средств");
    }

    public void rollback() {
        System.out.println("банк откатил операцию перевода средств");
    }
}
```

Теперь наш класс аспекта ничем не выдает, что это класс аспекта. Это простой класс для объекта Java в старом стиле (POJO) с несколькими методами. Взглянем теперь, как объявить советы в XML-конфигурации:

```
<aop:config>
  <aop:aspect ref="auditing">
    <aop:before pointcut="execution(*
      com.packt.patterninspring.chapter6.bankapp.
      service.TransferService.transfer(..))"
      method="validate"/>
    <aop:before pointcut="execution(*
      com.packt.patterninspring.chapter6.bankapp.
      service.TransferService.transfer(..))"
      method="transferInstantiate"/>
    <aop:after-returning pointcut="execution(*
      com.packt.patterninspring.chapter6.
      bankapp.service.TransferService.transfer(..))"
      method="success"/>
    <aop:after-throwing pointcut="execution(*
      com.packt.patterninspring.chapter6.bankapp.
      service.TransferService.transfer(..))"
      method="rollback"/>
  </aop:aspect>
</aop:config>
```

Как видите, `<aop:config>` относится к верхнему уровню. В `<aop:config>` объявляются другие элементы, например `<aop:aspect>` с атрибутом `ref`, ссылающийся на POJO-компонент `Auditing`. Он указывает, что `Auditing` в нашем приложении представляет собой класс аспекта. В элемент `<aop:aspect>` теперь включены элементы-советы и элементы-срезы. Логика при этом остается точно такой же, как описывалось в Java-конфигурации.

В следующем разделе мы изучим создание АОР-прокси фреймворком Spring.

АОР-прокси

Поскольку в основе Spring АОР лежат прокси, Spring создает прокси для вплетения аспектов между бизнес-логикой, расположенной в целевом объекте. АОР основано на паттернах проектирования «Заместитель» и «Декоратор». Рассмотрим класс `TransferServiceImpl`, реализующий интерфейс `TransferService`:

```
package com.packt.patterninspring.chapter6.bankapp.service;
import org.springframework.stereotype.Service;
public class TransferServiceImpl implements TransferService {
    @Override
    public void transfer(String accountA, String accountB, Long
        amount) {
        System.out.println(amount+" Денежная сумма была переведена
            с "+accountA+" на "+accountB);
    }
}
```

Вызывающая сторона обращается к этому сервису (метод `transfer()`) напрямую по ссылке на объект. Подробности приведены на рис. 6.13.

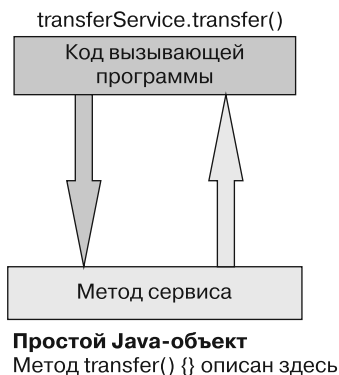


Рис. 6.13. Реализация интерфейса `TransferService`

Как видите, вызывающая программа может непосредственно обращаться к сервису и выполнять требуемую задачу.

Однако `TransferService` объявлен у нас как целевой для аспекта. Вследствие этого все немного меняется: у класса появляется адаптер-прокси и клиентский код не вызывает сервис напрямую — вызовы маршрутизирует прокси. Рассмотрим рис. 6.14.

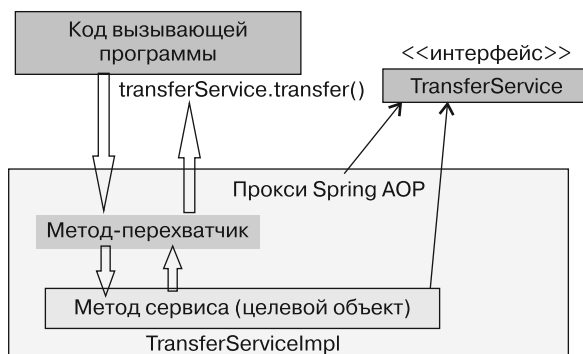


Рис. 6.14. Применение AOP-прокси к объекту фреймворком Spring

Рассмотрим последовательность применения AOP-прокси к объекту фреймворком Spring.

1. Spring создает прокси, сплетающий аспект и целевой объект.
2. Прокси также реализует интерфейс целевого объекта, то есть интерфейс `TransferService`.

3. Все вызовы метода `transfer()` сервиса перевода средств маршрутизируются через перехватчик прокси.
4. Выполняется соответствующий совет.
5. Выполняется целевой метод.

Эта последовательность действий выполняется при вызове метода с созданным Spring прокси.

Резюме

В этой главе мы рассмотрели фреймворк Spring AOP и паттерны проектирования, на которых он основан. Аспектно-ориентированное программирование (AOP) — парадигма, обладающая исключительно широкими возможностями, дополняющая парадигму объектно-ориентированного программирования. AOP дает возможность модульной организации таких элементов сквозной функциональности, как журналирование, безопасность и управление транзакциями. Аспект — это Java-класс, снабженный аннотацией `@Aspect`, который представляет собой модуль, содержащий логику сквозной функциональности. Этот модуль разделен с бизнес-логикой приложения. Его можно использовать повторно для других бизнес-модулей приложения без каких-либо изменений.

В Spring AOP поведение реализуется в виде методов-советов. В Spring есть пять типов советов: *«до»*, *«после исключения»*, *«после возврата»*, *«после»* и *«везде»*. Особенно широкими возможностями обладает тип совета *«везде»* — с его помощью реализовано множество интересных функций. Вы уже научились вплетать эти советы, используя методику вплетения во время загрузки.

Вы научились также объявлять в приложении Spring срезы и использовать их для выбора точек соединения, в которых применяются советы.

Теперь мы можем перейти к важнейшей части нашей книги и изучить работу фреймворка Spring в прикладной части, при соединении с базой данных и чтении данных из приложения. Начиная со следующей главы, вы будете работать над созданием приложений с помощью JDBC-шаблонов в Spring.

7 Доступ к базе данных с помощью фреймворка Spring и JDBC-реализаций паттерна «Шаблонный метод»

В предыдущих главах вы познакомились с базовыми модулями фреймворка Spring, такими как контейнер IoC, паттерн внедрения зависимостей, жизненный цикл контейнера, а также с используемыми в Spring паттернами проектирования. Кроме того, вы увидели своими глазами чудеса, творимые Spring с помощью AOP. Теперь пришло время перейти на поле боя настоящих приложений Spring и заняться сохранением данных. Помните приложение, в котором вы впервые столкнулись с доступом к базе данных? Тогда вам, вероятно, пришлось писать скучный стереотипный код для загрузки драйверов базы данных, инициализации фреймворка доступа к данным, открытия соединений, обработки различных исключений и закрытия соединений. Вам пришлось писать этот код очень осторожно. При любой ошибке вам не удалось бы подключиться к базе данных, несмотря на потраченное на этот скучный код время, и это не считая самого SQL-кода и кода бизнес-логики.

А поскольку всегда хочется все улучшить и упростить, то приходится избавляться от утомительной работы по организации доступа к данным. Spring предлагает способ решения данной проблемы и берет доступ к базе данных на себя. В Spring имеются фреймворки доступа к данным, обеспечивающие интеграцию со множеством технологий доступа. Spring позволяет использовать для сохранения данных или непосредственно JDBC, или любой фреймворк *объектно-реляционного отображения* (object-relational mapping, ORM), например Hibernate. Spring берет на себя всю низкоуровневую работу по организации доступа к данным, вам остается только написать SQL, логику приложения и работать со своими данными вместо того, чтобы тратить время на код открытия/закрытия соединений с базой данных и т. п.

Вы можете выбрать любую технологию для сохранения данных приложения, например JDBC, Hibernate, API сохранения Java (Java Persistence API, JPA) и др. Что бы вы ни предпочли, Spring поддерживает эти технологии для вашего прило-

жения. В этой главе мы изучим поддержку фреймворком Spring технологии JDBC, в частности следующие темы.

- ❑ Оптимальный подход к схеме доступа к данным.
- ❑ Реализация паттерна проектирования «Шаблонный метод».
- ❑ Проблемы традиционного JDBC.
- ❑ Решение этих проблем с помощью класса `JdbcTemplate` фреймворка Spring.
- ❑ Настройка источников данных.
- ❑ Использование паттерна проектирования «Пул объектов» для работы с соединениями базы данных.
- ❑ Абстрагирование доступа к базе данных с помощью паттерна DAO.
- ❑ Работа с классом `JdbcTemplate`.
- ❑ Интерфейсы обратного вызова JDBC.
- ❑ Рекомендуемые практики настройки класса `JdbcTemplate` в приложении.

Прежде чем углубиться в изучение JDBC и паттерна проектирования «Шаблонный метод», рассмотрим оптимальный подход к созданию уровня доступа к данным в многоуровневой архитектуре.

Оптимальный подход к проектированию доступа к данным

В предыдущих главах вы видели, что одна из целей фреймворка Spring состоит в обеспечении возможности разработки приложений путем следования одному из принципов ООП — программированию интерфейсов. Любому коммерческому приложению приходится читать данные из базы какого-либо вида и записывать в нее, а чтобы удовлетворить это требование, необходимо написать логику сохранения. Spring дает возможность избежать разбрасывания логики сохранения по всем модулям приложения. С этой целью можно создать отдельный компонент для доступа к данным и логики сохранения, который известен под названием «*объект доступа к данным*» (data access object, DAO). Взглянем на оптимальный способ создания модулей в многоуровневых приложениях (рис. 7.1).

Как можно видеть на схеме, многие корпоративные приложения ради оптимальности подхода состоят из следующих трех уровней.

- ❑ *Уровень сервисов* (service layer), также называемый прикладным уровнем (application layer), открывает доступ к высокоуровневым функциям приложения, например к реализации сценариев использования и бизнес-логики. Именно на этом уровне описаны все сервисы приложения.
- ❑ *Уровень доступа к данным* (data access layer) определяет интерфейс к хранилищу данных приложения (например, реляционной или NoSQL-базе данных). В нем содержатся классы и интерфейсы, обеспечивающие в приложении логику сохранения и обращения к данным.

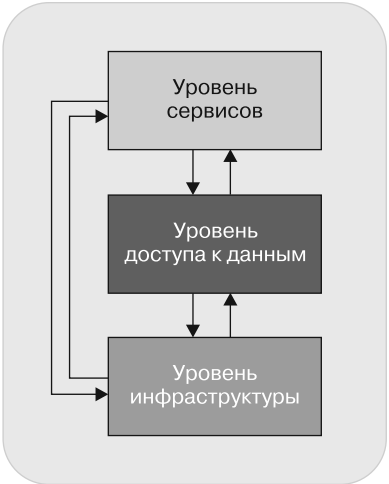


Рис. 7.1. Создание модулей в многоуровневых приложениях

- ❑ *Уровень инфраструктуры* (infrastructure layer) предоставляет другим уровням низкоуровневые сервисы, например задание URL базы данных, учетных данных пользователя в источнике данных и т. п.

На рис. 7.1 можно видеть взаимодействие *уровня сервисов* с *уровнем доступа к данным*. Чтобы избежать сцепления логики приложения с логикой доступа к данным, следует обеспечить доступ к их функциональности через интерфейсы, поскольку интерфейсы способствуют расцеплению взаимодействующих между собой компонентов. Если реализовать интерфейсы для логики доступа к данным, то можно будет настраивать любую нужную стратегию доступа к данным в приложении без изменения логики приложения на *уровне сервисов*. Рисунок 7.2 демонстрирует правильный подход к проектированию уровня доступа к данным.

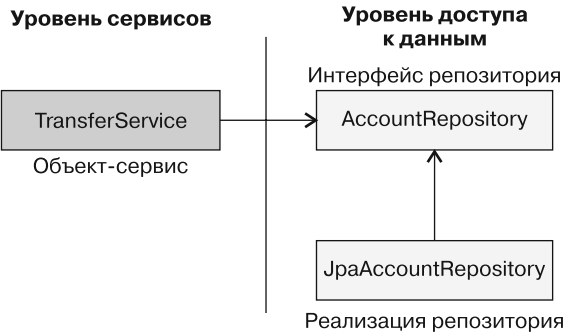


Рис. 7.2. Проектирование уровня доступа к данным

Объект-сервис вашего приложения, а именно `TransferService`, не может сам выполнять доступ к данным. Вместо этого он делегирует задачу доступа репозиторию. Интерфейс репозитория, а именно `AccountRepository`, обеспечивает слабое сцепление между ними. Это дает возможность использовать любой вариант реализации `AccountRepository` — и Jpa-реализацию (`JpaAccountRepository`), и Jdbc-реализацию (`JdbcAccountRepository`).

Spring не только обеспечивает слабое сцепление между работающими на разных уровнях компонентами приложения в многоуровневой архитектуре, но и помогает управлять ресурсами в корпоративном приложении с многоуровневой архитектурой. Посмотрим, как Spring управляет ресурсами и какие паттерны проектирования использует для решения этой задачи.

Задача управления ресурсами

Выясним, в чем состоит задача управления ресурсами, на реальном примере. Вы наверняка хоть раз заказывали пиццу через Интернет. Сколько же этапов включает в себя этот процесс, начиная с заказа пиццы и заканчивая ее доставкой? Довольно много. Сначала мы переходим на сайт пиццерии, выбираем размер и начинку пиццы. Затем мы размещаем и оплачиваем заказ. Заказ берет на выполнение ближайшее отделение пиццерии: там готовят пиццу, добавляют начинку, упаковывают, затем разносчик приносит ее вам домой и вручает вам, после чего, наконец, вы наслаждаетесь пиццей вместе с друзьями. Хотя шагов в этом процессе много, вы активно участвуете только в нескольких из них. Пиццерия отвечает за приготовление пиццы и ее доставку. Вы вовлекаетесь в процесс только при необходимости, за все прочие шаги отвечает пиццерия.

Как вы видите из этого примера, управление разбивается на множество шагов, на каждый из которых нужно выделять соответствующие ресурсы, чтобы избежать прерывания технологического процесса. Это идеальный сценарий для такого мощного паттерна проектирования, как «Шаблонный метод» (Template Method). Фреймворк Spring реализует этот паттерн проектирования специально для подобных сценариев на DAO-уровне приложения. Взглянем, с какими проблемами мы столкнулись бы без Spring в случае традиционного приложения.

В традиционном приложении для доступа к данным в базе используется API JDBC. Для простого приложения с получением и сохранением данных с помощью API JDBC необходимо выполнить следующие шаги.

1. Описать параметры соединения.
2. Обратиться к источнику данных и установить соединение.
3. Начать транзакцию.
4. Задать SQL-оператор.
5. Объявить параметры и задать их значения.

6. Подготовить и выполнить оператор.
7. Создать цикл для прохода по результатам.
8. Прodelать необходимые действия для каждой итерации цикла — выполнить бизнес-логику.
9. Обработать все исключения.
10. Зафиксировать или откатить транзакцию.
11. Закрыть соединение, оператор и результирующий набор.

При использовании для того же приложения фреймворка Spring достаточно будет написать код лишь для нескольких шагов из предыдущего списка. Spring возьмет на себя выполнение всех шагов, предполагающих такие низкоуровневые процессы, как установление соединения, открытие транзакции, обработка исключений на уровне доступа к данным и закрытие соединения. Spring выполняет эти шаги с помощью паттерна «Шаблонный метод», который мы рассмотрим в следующем разделе.

Реализация паттерна проектирования «Шаблонный метод»

Определяет общую структуру алгоритма, делегируя часть его шагов производным классам. Благодаря паттерну «Шаблонный метод» производные классы могут переопределять определенные шаги алгоритма без изменения его общей структуры.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

Мы обсуждали этот паттерн проектирования в главе 3. Он очень популярен и отнесен в GoF к семейству поведенческих паттернов проектирования. «Шаблонный метод» определяет общую структуру алгоритма, оставляя нюансы конкретной реализации на более позднее время и скрывая таким образом значительные объемы стереотипного кода. Spring предоставляет нам множество шаблонных классов, например `JdbcTemplate`, `JmsTemplate`, `RestTemplate` и `WebServiceTemplate`.

В основном этот паттерн служит для скрытия низкоуровневого управления ресурсами, как обсуждалось выше в примере с пиццей. Технологический процесс, которому следует пиццерия в отношении каждого покупателя, включает несколько фиксированных шагов: оформление заказа, приготовление пиццы, добавление начинок в соответствии с пожеланиями покупателя и доставку пиццы по нужному адресу. Мы можем добавить эти шаги в конкретный алгоритм, а система реализует его соответствующим образом.

Spring реализует этот паттерн для получения данных из базы. В работе базы данных или любой другой технологии всегда предусмотрено несколько общих

шагов, например установление соединения с базой данных, обработка транзакций, обработка исключений и определенные действия по очистке, необходимые для любого процесса доступа к данным. Но есть и несколько нефиксированных шагов, зависящих от потребностей конкретного приложения. За описание этих шагов отвечает разработчик, но Spring дает возможность разделять фиксированную и динамическую часть процесса доступа к данным на шаблонные методы и методы обратного вызова. Все фиксированные шаги относятся к шаблонным методам, а динамические пользовательские шаги — к обратным вызовам. На рис. 7.3 все это подробно проиллюстрировано.

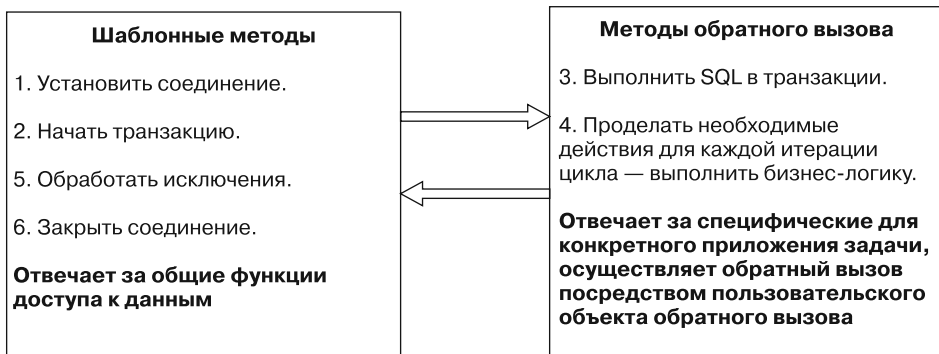


Рис. 7.3. Разделение фиксированной и динамической частей процесса

Как можно видеть на предыдущей схеме, все фиксированные части процесса доступа к данным обертываются в шаблонные классы фреймворка Spring, в частности операции открытия и закрытия соединения, открытия и закрытия операторов, обработки исключений и управления ресурсами. Но прочие шаги, например выполнение SQL, объявление параметров соединения и т. п., представляют собой элементы обратных вызовов, а за обратные вызовы отвечает разработчик.

Фреймворк Spring предлагает несколько реализаций паттерна проектирования «Шаблонный метод», например `JdbcTemplate`, `JmsTemplate`, `RestTemplate` и `WebServiceTemplate`, но в этой главе мы будем рассматривать лишь его реализацию для API JDBC — `JdbcTemplate`. Существует еще один вариант `JdbcTemplate` — `NamedParameterJdbcTemplate`, который представляет собой адаптер `JdbcTemplate`, позволяющий задавать поименованные параметры вместо обычных «заполнителей» JDBC — «?».

Проблемы традиционного JDBC

При работе с традиционным JDBC могут возникнуть следующие проблемы.

- ❑ *Избыточность результатов в случае потенциально подверженного ошибкам кода.*
Традиционный API JDBC требует написания большого объема громоздкого кода

для работы с уровнем доступа к данным. Рассмотрим следующий код подключения к базе данных и выполнения нужного запроса:

```
public List<Account> findByAccountNumber(Long accountNumber) {
    List<Account> accountList = new ArrayList<Account>();
    Connection conn = null;
    String sql = "select account_name,
account_balance from ACCOUNT where account_number=?";
    try {
        DataSource dataSource = DataSourceUtils.getDataSource();
        conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setLong(1, accountNumber);
        ResultSet rs = ps.executeQuery();
        while (rs.next()) {
            accountList.add(new Account(rs.getString(
"account_name"), ...));
        }
    } catch (SQLException e) { /* Что именно тут нужно обрабатывать? */ }
    finally {
        try {
            conn.close();
        } catch (SQLException e) { /* Что именно тут нужно обрабатывать? */ }
    }
    return accountList;
}
```

Большая часть здесь — стереотипный код. Кроме того, этот код неэффективно обрабатывает `SQLException`, поскольку разработчику неизвестно, что нужно обрабатывать в этом месте.

- ❑ *Неэффективная обработка исключений.* В предыдущем фрагменте кода возникающие в приложении исключения обрабатываются крайне неудачным образом. Разработчики не знают, какие исключения им нужно обрабатывать в этом месте. Тип `SQLException` представляет собой проверяемое исключение. Это значит, что оно требует от создателя обработки, а при ее невозможности — его объявления. Это крайне неудачный способ обработки исключений, при котором промежуточным методам необходимо объявлять исключение (-я) из всех методов кода, что представляет собой одну из форм сильного сцепления.

Решение проблем с помощью класса `JdbcTemplate` фреймворка Spring

Класс `JdbcTemplate` фреймворка Spring решает обе перечисленные в предыдущем разделе проблемы. `JdbcTemplate` существенно упрощает использование API JDBC и устраняет повторяющийся стереотипный код. Он нивелирует типичные причины возникновения ошибок и должным образом обрабатывает исключения типа

SQLException, не жертвуя при этом функциональными возможностями. Он предоставляет полный доступ к стандартным структурным компонентам JDBC.

Взглянем на решение указанных проблем в приведенном выше коде с помощью класса JdbcTemplate фреймворка Spring.

- ❑ *Удаление из приложения избыточного кода с помощью JdbcTemplate.* Допустим, нам требуется подсчитать количество счетов в банке. При использовании класса JdbcTemplate для этого понадобится следующий код:

```
int count = jdbcTemplate.queryForObject("SELECT COUNT(*)  
FROM ACCOUNT", Integer.class);
```

Для получения списка счетов для конкретного идентификатора пользователя:

```
List<Account> results = jdbcTemplate.query(someSql,  
new RowMapper<Account>() {  
    public Account mapRow(ResultSet rs, int row) throws  
        SQLException {  
        // Привязывает текущую строку к объекту Account  
    }  
});
```

Как можно видеть, больше нет необходимости писать код для открытия и закрытия соединения с базой данных, подготовки SQL-оператора перед выполнением запроса и т. д.

- ❑ *Исключения при доступе к данным.* Spring предоставляет согласованную иерархию исключений с DataAccessException в качестве корневого класса. С помощью этой иерархии можно обрабатывать такие относящиеся к конкретной технологии исключения, как SQLException, в рамках его собственной иерархии классов исключений. Фреймворк Spring обеспечивает адаптеры для этих исходных исключений в виде различных непроверяемых (unchecked) исключений. Благодаря этому разработчикам не требуется обрабатывать эти исключения во время написания кода. Предоставляемая Spring иерархия DataAccessException дает возможность скрытия используемой технологии: JPA, Hibernate, JDBC или аналогичной. Фактически это иерархия производных исключений, а не просто одно исключение на все случаи жизни (рис. 7.4).

Как можно видеть на рисунке, класс DataAccessException фреймворка Spring расширяет класс RuntimeException, а значит, это непроверяемое исключение. В корпоративном приложении можно передавать непроверяемые исключения вверх по иерархии вызовов вплоть до нахождения оптимального места для их обработки. А самое главное: промежуточные методы в приложении могут ничего не знать о них.

Далее мы обсудим вопрос настройки Spring для подключения источника данных к базе, а затем перейдем к объявлению шаблонных методов и репозитория в приложениях Spring.

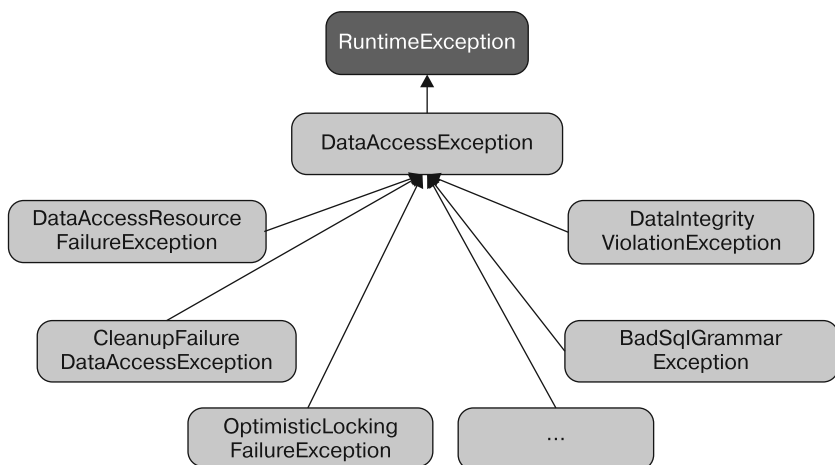


Рис. 7.4. Иерархия исключений доступа к данным фреймворка Spring

Настройка источника данных и паттерн «Пул объектов»

Во фреймворке Spring интерфейс `DataSource`, обеспечивающий соединение с базой данных, представляет собой часть API JDBC. Он скрывает большую часть стереотипного кода для создания пула соединений, обработки исключений и управления транзакциями; администраторы приложения отвечают за настройку управляемых контейнером источников данных в промышленной эксплуатации. Вам нужно просто написать и протестировать код бизнес-логики.

В корпоративном приложении существует несколько вариантов получения объекта `DataSource`. Для этого можно воспользоваться JDBC-драйвером, но это не лучший способ создания `DataSource` в среде промышленной эксплуатации. Поскольку производительность — один из ключевых вопросов при разработке приложения, Spring реализует паттерн «Пул объектов» (Object Pool) для предоставления приложению источника данных наиболее эффективным способом. Суть паттерна определяется так: *создание объектов требует больших затрат, чем переиспользование*.

Spring позволяет реализовать паттерн «Пул объектов» для переиспользования объекта `DataSource` в приложении. Можно или воспользоваться сервером приложения и управляемым контейнером пулом (JNDI), или создать контейнер с помощью сторонних библиотек, например DBCP, c3p0 и т. п. Эти пулы помогают оптимизировать управление доступными источниками данных.

В приложении Spring есть несколько возможных вариантов задать настройки для компонентов источников данных.

- ❑ Конфигурирование источника данных с помощью JDBC-драйвера.
- ❑ Реализация паттерна проектирования «Пул объектов» для получения объектов источников данных.

- ❑ Конфигурирование источника данных с помощью JNDI.
- ❑ Конфигурирование источника данных с помощью соединений пула.
- ❑ Реализация паттерна «Строитель» с целью создания встроенного источника данных.

Рассмотрим задание настроек компонента в приложении Spring.

Задание настроек источника данных с помощью JDBC-драйвера

Использование JDBC-драйвера для конфигурирования компонента источника данных — простейший вариант из всех. Для этой цели в Spring предусмотрено три класса источников данных:

- ❑ `DriverManagerDataSource` — всегда создает новое соединение для каждого запроса соединения;
- ❑ `SimpleDriverDataSource` — аналогичен `DriverManagerDataSource`, за исключением того, что работает непосредственно с драйвером JDBC;
- ❑ `SingleConnectionDataSource` — возвращает одно и то же соединение для каждого запроса соединения, но это источник данных без пула соединений.

Рассмотрим следующий фрагмент кода, предназначенный для конфигурирования компонента источника данных и использующий класс `DriverManagerDataSource` фреймворка Spring.

В конфигурации на основе Java код имеет такой вид:

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("org.h2.Driver");
dataSource.setUrl("jdbc:h2:tcp://localhost/bankDB");
dataSource.setUsername("root");
dataSource.setPassword("root");
```

В конфигурации на основе XML код будет таким:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.
      DriverManagerDataSource">
  <property name="driverClassName" value="org.h2.Driver"/>
  <property name="url" value="jdbc:h2:tcp://localhost/bankDB"/>
  <property name="username" value="root"/>
  <property name="password" value="root"/>
</bean>
```

Описанный в предыдущем фрагменте кода источник данных очень прост, он подходит для использования в среде разработки. Для промышленной эксплуатации он не годится. Я предпочитаю конфигурировать источник данных для производственной среды с помощью JNDI. Посмотрим, как это делается.

Реализуем паттерн проектирования «Пул объектов» так, чтобы он возвращал объекты источников данных *посредством* конфигурирования источника данных с помощью JNDI.

В приложении Spring можно настраивать источник данных с помощью JNDI-поиска. Spring предоставляет элемент `<jee:jndi-lookup>` из пространства имен JEE фреймворка Spring. Рассмотрим необходимый для выполнения этих настроек код.

В XML-конфигурации код будет иметь следующий вид:

```
<jee:jndi-lookup id="dataSource"
jndi-name="java:comp/env/jdbc/datasource" />
```

В Java-конфигурации код будет иметь такой вид:

```
@Bean
public JndiObjectFactoryBean dataSource() {
    JndiObjectFactoryBean jndiObject = new JndiObjectFactoryBean();
    jndiObject.setJndiName("jdbc/datasource");
    jndiObject.setResourceRef(true);
    jndiObject.setProxyInterface(javax.sql.DataSource.class);
    return jndiObject;
}
```

Серверы приложений, такие как WebSphere и JBoss, дают возможность устанавливать настройки источников данных с помощью JNDI. Даже веб-контейнеры, например Tomcat, предоставляют такую возможность. Эти серверы служат для управления источниками данных в приложении. Это приводит к повышению производительности источников данных, поскольку серверы приложений часто организованы в виде пула. Управление источниками данных полностью осуществляется извне приложения. Это один из лучших способов настройки источника данных для извлечения его объекта через JNDI. Если же возможности воспользоваться поиском по JNDI в промышленной эксплуатации нет, то есть другой, еще лучший способ, который мы обсудим далее.

Конфигурирование источника данных с помощью пула соединений

Следующие технологии с открытым исходным кодом предоставляют источники данных с пулом соединений:

- ❑ компонент DBCP из набора библиотек Apache Commons;
- ❑ библиотека c3p0;
- ❑ библиотека BoneCP.

Рассмотрим код, который позволяет настроить объект `BasicDataSource` компонента DBCP.

Соответствующая XML-конфигурация для DBCP выглядит следующим образом:

```
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
    <property name="driverClassName" value="org.h2.Driver"/>
    <property name="url" value="jdbc:h2:tcp://localhost/bankDB"/>
```

```
<property name="username" value="root"/>
<property name="password" value="root"/>
<property name="initialSize" value="5"/>
<property name="maxActive" value="10"/>
</bean>
```

А Java-конфигурация для DBCP выглядит вот так:

```
@Bean
public BasicDataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName("org.h2.Driver");
    dataSource.setUrl("jdbc:h2:tcp://localhost/bankDB");
    dataSource.setUsername("root");
    dataSource.setPassword("root");
    dataSource.setInitialSize(5);
    dataSource.setMaxActive(10);
    return dataSource;
}
```

Как можно видеть из предыдущего кода, для провайдера источников данных с пулом соединений существует множество различных свойств. Ниже перечислены¹ свойства класса `BasicDataSource` в Spring:

- ❑ `initialSize` — количество соединений, создаваемое при инициализации пула;
- ❑ `maxActive` — максимальное количество соединений, выделяемое из пула при его инициализации. Если это свойство равно 0, то ограничений нет²;
- ❑ `maxIdle` — максимальное количество соединений, которые могут простаивать без освобождения дополнительных соединений. Если присвоить данному свойству значение 0, то ограничений не будет³;
- ❑ `maxOpenPreparedStatements` — максимальное количество подготовленных соединений, которые могут быть выделены из пула операторов при его инициализации. Если присвоить данному свойству значение 0, то ограничений не будет⁴;
- ❑ `maxWait` — максимальное время ожидания возврата соединения в пул. В случае превышения генерируется исключение. Значение -1 означает «ждать неограниченно долго»;
- ❑ `minEvictableIdleTimeMillis` — максимальная длительность простоя соединения перед тем, как оно будет сочтено удовлетворяющим критериям для вытеснения;
- ❑ `minIdle` — максимальное количество соединений, которые могут простаивать без создания дополнительных соединений.

¹ Со времени выхода оригинального издания данной книги появились новые версии DBCP, в которых список свойств класса `BasicDataSource` существенно изменился. Рекомендую заглянуть в актуальную документацию компонента DBCP.

² Или меньше 0.

³ Согласно официальной документации, для отсутствия ограничений необходимо задать отрицательное значение данного свойства.

⁴ Или меньше 0.

Реализация паттерна «Строитель» для создания встроенного источника данных

При разработке приложений встроенные базы данных очень удобны, поскольку не требуют подключения приложения к отдельному серверу баз данных. Spring предоставляет дополнительный источник данных для встроенных баз. Для среды промышленной эксплуатации его возможностей недостаточно, однако его можно использовать для разработки и тестирования. Рассмотрим настройку встроенной базы данных H2 с помощью пространства имен `jdbc`.

В XML-конфигурации H2 настраивается следующим образом:

```
<jdbc:embedded-database id="dataSource" type="H2">
  <jdbc:script location="schema.sql"/>
  <jdbc:script location="data.sql"/>
</jdbc:embedded-database>
```

В Java-конфигурации H2 настраивается таким образом:

```
@Bean
public DataSource dataSource(){
    EmbeddedDatabaseBuilder builder =
        new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2);
    builder.addScript("schema.sql");
    builder.addScript("data.sql");
    return builder.build();
}
```

Как можно видеть из предыдущего фрагмента кода, Spring предоставляет ет класс `EmbeddedDatabaseBuilder`. Фактически при создании объекта класса `EmbeddedDatabaseBuilder` реализуется паттерн проектирования «Строитель».

Рассмотрим еще один паттерн проектирования.

Абстрагирование доступа к базе данных с помощью паттерна DAO

Уровень доступа к данным играет роль аспекта между бизнес-уровнем и базой данных. Доступ к данным зависит от обращения к бизнес-уровню и меняется в зависимости от первоисточника данных для учебной базы, неструктурированных файлов, XML и т. п. Таким образом, весь доступ к базе данных можно абстрагировать путем создания интерфейса. Эта методика известна под названием паттерна «Объект доступа к данным» (Data Access Object). С точки зрения приложения разницы между обращением к реляционной базе данных и синтаксическим разбором XML-файлов при использовании DAO нет.

В предыдущей версии корпоративной модели Java-компонентов (EJB) присутствовали управляемые контейнером компоненты-сущности — распределенные, безопасные, транзакционные. Для клиента они были прозрачны, то есть обеспечивали автоматическое сохранение на уровне сервисов приложения, независимо от

нижележащей базы данных. Но большая часть предлагаемых этими компонентами-сущностями возможностей не нужна для вашего приложения, ведь вам требуется только сохранять данные в базе. Поэтому растут некоторые нежелательные свойства компонентов-сущностей, например сетевой трафик, что влияет на производительность приложения. И теперь компоненты-сущности необходимо запускать внутри EJB-контейнеров, что сильно затрудняет их тестирование.

Одним словом, при работе с традиционными API JDBC или предыдущими версиями EJB вы столкнетесь со следующими проблемами в своем приложении.

- ❑ В традиционном JDBC-приложении логика бизнес-уровня сливается с логикой сохранения.
- ❑ Для уровня сервисов и бизнес-уровня не достигается согласованность использования уровня сохранения или уровня DAO. Но в корпоративном приложении DAO должен предоставлять единообразный интерфейс для уровня сервисов.
- ❑ В традиционном JDBC-приложении приходится иметь дело с большим количеством стереотипного кода, например кода открытия и закрытия соединений, подготовки операторов, обработки исключений и т. д., что снижает возможности переиспользования и повышает длительность разработки.
- ❑ При использовании EJB создаваемый компонент-сущность требовал дополнительных ресурсов от приложения, а его тестирование представляло собой непростую задачу.

Посмотрим, как фреймворк Spring решает эти проблемы.

Реализация паттерна DAO с помощью фреймворка Spring

Для проектирования и разработки основанных на JDBC объектов DAO Spring предоставляет нам всеобъемлющий модуль JDBC. Эти объекты DAO в приложении берут на себя весь стереотипный код API JDBC и участвуют в создании единообразного API для доступа к данным. В Spring JDBC DAO — обобщенный объект для доступа к данным с бизнес-уровня, предоставляющий согласованный интерфейс для сервисов бизнес-уровня. Основная задача классов DAO состоит в абстрагировании нижележащей логики доступа к данным от сервисов на бизнес-уровне.

Предыдущий пример с пиццерией должен был помочь вам лучше понять задачу управления ресурсами, а теперь вернемся к примеру банковского приложения. Рассмотрим следующий пример реализации DAO в приложении. Допустим, в банковском приложении нам нужно получить общее количество банковских счетов в городском подразделении банка. Чтобы сделать это, мы прежде всего создадим интерфейс для DAO. Паттерн DAO поощряет программирование посредством интерфейсов, как мы уже обсуждали ранее. Это одна из рекомендуемых практик принципов проектирования. Далее мы внедряем вышеупомянутый интерфейс DAO в сервисы на бизнес-уровне и можем создать на его основе несколько конкретных классов в соответствии с используемыми в приложении базами данных. Это значит,

что наш уровень DAO будет иметь согласованный вид для всего бизнес-уровня. Создадим интерфейс DAO с помощью следующего кода:

```
package com.packt.patterninspring.chapter7.bankapp.dao;
public interface AccountDao {
    Integer totalAccountsByBranch(String branchName);
}
```

Создадим конкретную реализацию интерфейса DAO с помощью класса `JdbcDaoSupport` фреймворка Spring:

```
package com.packt.patterninspring.chapter7.bankapp.dao;

import org.springframework.jdbc.core.support.JdbcDaoSupport;
public class AccountDaoImpl extends JdbcDaoSupport implements
AccountDao {
    @Override
    public Integer totalAccountsByBranch(String branchName) {
        String sql = "SELECT count(*) FROM Account WHERE branchName =
        "+branchName;
        return this.getJdbcTemplate().queryForObject(sql,
        Integer.class);
    }
}
```

Как вы можете видеть из этого кода, класс `AccountDaoImpl` реализует интерфейс `AccountDao` и расширяет класс `JdbcDaoSupport` для упрощения разработки на основе JDBC. Класс `JdbcDaoSupport` предоставляет своим производным классам возможность получения объекта `JdbcTemplate` с помощью метода `getJdbcTemplate()`. Класс `JdbcDaoSupport` связывается с источником данных и предоставляет объект `JdbcTemplate` для использования в DAO.

Работа с JdbcTemplate

Итак, класс `JdbcTemplate` фреймворка Spring устраняет две основные проблемы приложения: проблему избыточного кода и проблему низкоэффективной обработки исключений, генерируемых кодом доступа к данным нашего приложения. В отсутствие `JdbcTemplate` лишь 20 % кода приложения фактически применяется для выполнения запроса, а 80 % представляет собой стереотипный код для обработки исключений и управления ресурсами. При использовании же `JdbcTemplate` не требуется волноваться об этих 80 % стереотипного кода. Если вкратце, то класс `JdbcTemplate` фреймворка Spring отвечает за следующее:

- ☐ установление соединения;
- ☐ участие в транзакции;
- ☐ выполнение оператора;
- ☐ обработку результирующего набора;
- ☐ обработку исключений;
- ☐ освобождение соединения.

Разберемся, когда в приложении нужно использовать объект `JdbcTemplate` и как его создать.

Когда использовать `JdbcTemplate`

Объект `JdbcTemplate` пригодится в автономных приложениях и везде, где нужен JDBC. Он хорошо подходит для того, чтобы привести в порядок запутанные унаследованные фрагменты кода во вспомогательном или тестовом коде. С его помощью можно также реализовать репозиторий или объект доступа к данным в любом многоуровневом приложении. Посмотрим, как можно создать его в нашем приложении.

Создание в приложении объекта `JdbcTemplate`

Если вы хотите создать объект класса `JdbcTemplate` для доступа к данным в приложении Spring, не забывайте: для создания подключения к базе данных необходим объект `DataSource`. Нужно создать шаблонный метод однократно и дальше переиспользовать его. Не стоит создавать его для каждого потока выполнения, класс `JdbcTemplate` гарантирует потокобезопасность:

```
JdbcTemplate template = new JdbcTemplate(dataSource);
```

Зададим конфигурацию компонента `JdbcTemplate` в Spring с помощью следующего метода `@Bean`:

```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}
```

Здесь мы воспользовались в приложении Spring внедрением через конструктор, чтобы внедрить объект `DataSource` в компонент `JdbcTemplate`. Компонент `dataSource`, на который мы ссылаемся, может быть любой из реализаций интерфейса `javax.sql.DataSource`.

Рассмотрим пример использования компонента `JdbcTemplate` в основанном на JDBC-репозитории для доступа к базе данных в приложении.

Реализация репозитория на основе JDBC

Класс `JdbcTemplate` фреймворка Spring можно задействовать для реализации репозитория в приложениях Spring. Посмотрим, как реализовать класс репозитория на основе `JdbcTemplate`:

```
package com.packt.patterninspring.chapter7.bankapp.repository;

import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;
```

```

import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;
import com.packt.patterninspring.chapter7.bankapp.model.Account;
@Repository
public class JdbcAccountRepository implements AccountRepository{
    JdbcTemplate jdbcTemplate;
    public JdbcAccountRepository(DataSource dataSource) {
        super();
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
    @Override
    public Account findAccountById(Long id){
        String sql = "SELECT * FROM Account WHERE id = "+id;
        return jdbcTemplate.queryForObject(sql,
            new RowMapper<Account>(){
                @Override
                public Account mapRow(ResultSet rs, int arg1) throws
                    SQLException {
                    Account account = new Account(id);
                    account.setName(rs.getString("name"));
                    account.setBalance(new Long(rs.getInt("balance")));
                    return account;
                }
            });
    }
}

```

В предыдущем коде мы внедряем компонент `DataSource` в класс `JdbcAccountRepository` через конструктор. С помощью этого компонента мы создали объект `JdbcTemplate` для доступа к данным. Класс `JdbcTemplate` предоставляет следующие методы для получения данных из базы:

- ❑ `queryForObject(..)` — запрашивает простые типы данных Java (`int`, `long`, `String`, `Date`) и пользовательские объекты предметной области;
- ❑ `queryForMap(..)` — используется, когда ожидается, что результат запроса будет состоять из одной строки. `JdbcTemplate` возвращает каждую строку набора результатов в виде ассоциативного массива;
- ❑ `queryForList(..)` — применяется, когда ожидается, что результат запроса будет состоять из нескольких строк.



Надо отметить, что методы `queryForInt` и `queryForLong` считаются устаревшими, начиная с версии Spring 3.2. Вместо них можно использовать `queryForObject` (API был усовершенствован в Spring 3).

Зачастую бывает удобно задавать соответствие реляционных данных объектам предметной области, например `ResultSet` — `Account` в предыдущем коде. Класс `JdbcTemplate` фреймворка Spring поддерживает эту возможность благодаря обратным вызовам.

Интерфейсы обратного вызова JDBC

Фреймворк Spring предоставляет три следующих интерфейса обратного вызова для JDBC.

На основе реализации интерфейса `RowMapper`. В Spring предусмотрен интерфейс `RowMapper` для отображения отдельной строки `ResultSet` в объект. Его можно использовать как для однострочных, так и для многострочных запросов. По состоянию на версию Spring 3.0 он параметризован следующим образом:

```
public interface RowMapper<T> {
    T mapRow(ResultSet rs, int rowNum)
        throws SQLException;
}
```

Рассмотрим пример.

Создание класса на основе `RowMapper`

В следующем примере класс `AccountRowMapper` реализует интерфейс `RowMapper` модуля JDBC фреймворка Spring:

```
package com.packt.patterninspring.chapter7.bankapp.rowmapper;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
import com.packt.patterninspring.chapter7.bankapp.model.Account;

public class AccountRowMapper implements RowMapper<Account>{
    @Override
    public Account mapRow(ResultSet rs, int id) throws SQLException {
        Account account = new Account();
        account.setId(new Long(rs.getInt("id")));
        account.setName(rs.getString("name"));
        account.setBalance(new Long(rs.getInt("balance")));
        return account;
    }
}
```

Здесь класс `AccountRowMapper` отображает строку результирующего набора в объект предметной области. Этот класс отображения строк реализует интерфейс обратного вызова `RowMapper` модуля JDBC фреймворка Spring.

Выполнение однострочных запросов с помощью `JdbcTemplate`. Взглянем на то, как класс отображения строк устанавливает соответствие отдельной строки объекту предметной области:

```
public Account findAccountById(Long id){
    String sql = "SELECT * FROM Account WHERE id = "+id;
    return jdbcTemplate.queryForObject(sql, new AccountRowMapper());
}
```

Здесь нет необходимости выполнять приведение типов для объекта `Account`. Класс `AccountRowMapper` отображает строки в объекты типа `Account`.

Многострочные запросы. Следующий код демонстрирует задание соответствия нескольких строк списку объектов предметной области:

```
public List<Account> findAccountById(Long id){
    String sql = "SELECT * FROM Account ";
    return jdbcTemplate.queryForList(sql, new AccountRowMapper());
}
```

Интерфейс `RowMapper` — оптимальный выбор для случая, когда каждая строка `ResultSet` соответствует объекту предметной области.

Реализация интерфейса `RowCallbackHandler`

Фреймворк Spring предоставляет более простой интерфейс `RowCallbackHandler` на случай, когда никакой объект не возвращается. Он используется для потоковой отправки строк в файл, преобразования строк в XML и фильтрации их перед добавлением в коллекцию. Но для больших запросов фильтрации в SQL этот интерфейс намного эффективнее и выполняется быстрее, чем его JPA-эквивалент. Рассмотрим пример:

```
public interface RowCallbackHandler {
    void processRow(ResultSet rs) throws SQLException;
}
```

Пример использования. Следующий фрагмент кода представляет собой пример использования интерфейса `RowCallbackHandler` в приложении:

```
package com.packt.patterninspring.chapter7.bankapp.callbacks;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowCallbackHandler;
public class AccountReportWriter implements RowCallbackHandler {
    public void processRow(ResultSet resultSet) throws SQLException {
        // Синтаксический разбор объекта ResultSet и потоковая запись
        // неструктурированного файла или XML
    }
}
```

В предыдущем коде мы создали реализацию интерфейса `RowCallbackHandler`. Класс `AccountReportWriter` реализует этот интерфейс и обрабатывает полученный из базы данных результирующий набор. Рассмотрим пример использования обратного вызова `AccountReportWriter`:

```
@Override
public void generateReport(Writer out, String branchName) {
    String sql = "SELECT * FROM Account WHERE branchName = '"+
        branchName;
    jdbcTemplate.query(sql, new AccountReportWriter());
}
```

Интерфейс `RowCallbackHandler` — оптимальный вариант для случая, когда метод обратного вызова не возвращает значения каждой строки, особенно при больших запросах.

Реализация интерфейса ResultSetExtractor

Для обработки сразу всего объекта `ResultSet` Spring предоставляет интерфейс `ResultSetExtractor`. В этом случае за обработку в цикле `ResultSet` (например, отображение всего `ResultSet` в один объект) отвечаете вы. Рассмотрим пример:

```
public interface ResultSetExtractor<T> {
    T extractData(ResultSet rs) throws SQLException,
        DataAccessException;
}
```

Пример использования. Следующий фрагмент кода реализует интерфейс `ResultSetExtractor`:

```
package com.packt.patterninspring.chapter7.bankapp.callbacks;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.ResultSetExtractor;
import com.packt.patterninspring.chapter7.bankapp.model.Account;

public class AccountExtractor implements
    ResultSetExtractor<List<Account>> {
    @Override
    public List<Account> extractData(ResultSet resultSet) throws
        SQLException, DataAccessException {
        List<Account> extractedAccounts = null;
        Account account = null;
        while (resultSet.next()) {
            if (extractedAccounts == null) {
                extractedAccounts = new ArrayList<>();
                account = new Account(resultSet.getLong("ID"),
                    resultSet.getString("NAME"), ...);
            }
            extractedAccounts.add(account);
        }
        return extractedAccounts;
    }
}
```

Вышеприведенный класс `AccountExtractor` реализует интерфейс `ResultSetExtractor` и используется для создания объекта, содержащего все элементы возвращаемого из базы данных результирующего набора. Рассмотрим пример использования этого класса в приложении:

```
public List<Account> extractAccounts() {
    String sql = "SELECT * FROM Account";
    return jdbcTemplate.query(sql, new AccountExtractor());
}
```

Предыдущий фрагмент кода отвечает за получение всех банковских счетов и подготовку их списка с помощью класса `AccountExtractor`. Этот класс реализует интерфейс обратного вызова `ResultSetExtractor` из модуля JDBC фреймворка Spring.

Интерфейс `ResultSetExtractor` — оптимальный вариант для отображения нескольких строк результирующего набора в один объект.

Рекомендуемые практики JDBC и настройки JdbcTemplate

Экземпляры класса `JdbcTemplate` достаточно настроить один раз, чтобы можно было их безопасно использовать многопоточным образом. При конфигурации `JdbcTemplate` в приложениях Spring рекомендуется внедрять компонент источника данных в классах DAO через конструктор или сеттер, передавая этот компонент источника данных в качестве аргумента конструктора класса `JdbcTemplate`. При этом объекты DAO выглядят (частично) следующим образом:

```
@Repository
public class JdbcAccountRepository implements AccountRepository{
    JdbcTemplate jdbcTemplate;
    public JdbcAccountRepository(DataSource dataSource) {
        super();
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
    //...
}
```

Рассмотрим несколько рекомендуемых практик задания настроек базы данных и написания кода для уровня DAO.

- ❑ При необходимости задания настроек встроенной базы данных во время разработки приложения рекомендуется, чтобы ей всегда присваивалось уникальное имя. Дело в том, что в контейнере Spring доступ к базе данных осуществляется путем настройки компонента типа `javax.sql.DataSource`, который внедряется в объекты доступа к данным.
- ❑ Всегда используйте пулы объектов. Существует два варианта:
 - *пулы соединений* — диспетчер пула может сохранять закрытые соединения в пуле;
 - *пулы операторов* — драйвер может переиспользовать объекты подготовленных операторов.
- ❑ Обдуманно выбирайте режим фиксации.
- ❑ Подумайте: возможно, имеет смысл отказаться в приложении от режима автофиксации и применять ручную фиксацию, чтобы лучше контролировать логику приложения:


```
Connection.setAutoCommit(false);
```

Резюме

Приложение без данных подобно автомобилю без горючего. Данные — движитель приложения. Некоторые приложения могут существовать и без данных, но обычно они представляют собой что-то вроде статических блогов. Данные — важная часть любого приложения, и разработка кода доступа к ним для приложения играет важнейшую роль. Этот код должен быть простым, надежным и легко адаптироваться к любой задаче.

В традиционных Java-приложениях для доступа к данным можно задействовать JDBC. Это очень простой способ, однако может быть весьма хлопотно описывать спецификации, обрабатывать исключения JDBC, выполнять соединения с базой данных, загружать драйверы и т. п. Фреймворк Spring упрощает эти действия, устраняя стереотипный код и облегчая обработку исключений JDBC. Разработчику остается лишь написать код SQL, предназначенный для выполнения в данном приложении, все остальное берет на себя фреймворк Spring.

В этой главе было показано, как Spring помогает прикладной части в доступе к данным и их сохранении. JDBC тоже подходит для этого, но непосредственное использование API JDBC — утомительная и чреватая ошибками работа. Интерфейс `JdbcTemplate` упрощает доступ к данным и обеспечивает согласованность. Доступ к данным с помощью Spring основан на принципах многоуровневой архитектуры — верхние уровни не должны ничего знать о нюансах управления данными. Spring изолирует SQL-исключения посредством класса `DataAccessException` и создает целую иерархию классов для упрощения их обработки.

В следующей главе мы продолжим обсуждение доступа к данным и их сохранения с помощью фреймворков ORM, например Hibernate или JPA.

8

Доступ к базе данных с помощью паттернов ORM и транзакций

В главе 7 вы узнали, как обращаться к базе данных с помощью JDBC и как Spring позволяет разработчику избавиться от стереотипного кода на своей стороне, передав его реализацию фреймворку с помощью паттерна «Шаблонный метод» и обратных вызовов. В этой главе вы изучите продвинутый метод доступа к базе данных с помощью фреймворка *объектно-реляционного отображения* (object-relational mapping, ORM) и управления транзакциями приложения.

Когда моему сыну Арнаву было полтора года, он часто играл с игрушечным мобильным телефоном. Но скоро он перерос игрушечные телефоны, и мне пришлось купить ему смартфон.

Точно так же, когда бизнес-уровню приложения требуется лишь небольшой набор данных, JDBC работает прекрасно, но по мере роста и усложнения приложения становится все сложнее отображать таблицы на объекты приложения. JDBC — маленький игрушечный телефон в мире доступа к данным. Но в случае сложных приложений необходимы решения класса ORM, которые могли бы отобразить свойства объектов в столбцы базы данных. Требуются более совершенные платформы для уровня доступа к данным нашего приложения, которые могли бы создавать запросы и операторы независимо от технологий базы данных, причем с возможностью декларативного или программного описания.

Многие фреймворки ORM способны предоставлять сервисы на уровне доступа к данным. Среди примеров подобных сервисов объектно-реляционное отображение, отложенная загрузка данных, немедленная загрузка данных, каскадирование и т. д. Эти ORM-сервисы избавляют разработчика от написания тонн кода обработки ошибок и управления ресурсами приложения. Фреймворки ORM снижают длительность разработки и помогают писать не содержащий ошибок код, позволяя сосредоточить внимание на удовлетворении бизнес-требований. Фреймворк Spring не реализует собственное ORM-решение, но поддерживает множество фреймворков для сохранения данных, например *Hibernate*, *API сохранения Java* (Java Persistence API, JPA), *iBATIS* и *объекты данных Java* (Java Data Objects, JDO). Spring также предоставляет точки интеграции для фреймворков ORM, чтобы можно было удобно интегрировать их в приложение.

Spring обеспечивает поддержку в приложении всех этих технологий. В этой главе мы займемся исследованием поддержки Spring ORM-решений и охватим следующие темы.

- ❑ Фреймворки ORM и используемые в них паттерны.
- ❑ Паттерн «Объект доступа к данным».
- ❑ Создание объектов DAO в Spring с помощью паттерна проектирования «Фабрика».
- ❑ Паттерн «Отображение данных».
- ❑ Паттерн «Модель предметной области».
- ❑ Прокси для паттерна «Отложенная загрузка».
- ❑ Паттерн «Шаблонный метод» в Hibernate.
- ❑ Интеграция Hibernate с фреймворком Spring.
- ❑ Настройка объекта `SessionFactory` фреймворка Hibernate в контейнере Spring.
- ❑ Реализация DAO на основе простого API Hibernate.
- ❑ Стратегии управления транзакциями в Spring.
- ❑ Декларативная реализация и задание границ транзакций.
- ❑ Программная реализация и задание границ транзакций.
- ❑ Рекомендуемые практики для ORM и модулей транзакций.

Прежде чем мы перейдем к более подробному обсуждению фреймворков ORM, рассмотрим некоторые паттерны проектирования, используемые на *уровне доступа к данным* (data access layer, DAL) приложения.

Фреймворки ORM и используемые в них паттерны

Spring поддерживает несколько фреймворков ORM, в частности Hibernate, API сохранения Java (Java Persistence API, JPA), iBATIS и объекты данных Java (Java Data Objects, JDO). Благодаря использованию в приложении любого из этих ORM-решений можно легко сохранять данные в реляционных базах данных и получать доступ к ним в виде объектов POJO. Модуль ORM фреймворка Spring представляет собой расширение обсуждавшегося ранее модуля JDBC DAO фреймворка Spring. Вот список фреймворков ORM и возможностей интеграции, поддерживаемых фреймворком Spring:

- ❑ Hibernate;
- ❑ API сохранения Java;
- ❑ объекты данных Java;
- ❑ iBATIS;
- ❑ реализации объектов доступа к данным;
- ❑ стратегии управления транзакциями.

Для настройки ORM-решений в приложении можно использовать возможности внедрения зависимостей фреймворка Spring. Spring также предлагает серьезные расширения ORM-уровня приложений, выполняющих доступ к данным. Применение фреймворка Spring для создания объектов DAO ORM несет следующие преимущества.

- ❑ *Упрощение разработки и тестирования.* Контейнер IoC фреймворка Spring управляет компонентами объектов DAO ORM. При необходимости можно легко воспользоваться другой реализацией интерфейса DAO с помощью внедрения зависимостей Spring. Упрощается также изоляционное тестирование кода сохранения данных.
- ❑ *Иерархия исключений доступа к данным.* Spring предоставляет согласованную иерархию исключений, связанных с доступом к данным. Она позволяет обрабатывать исключения на уровне сохранения. Spring оборачивает все проверяемые исключения, генерируемые ORM, и транслирует их в непроверяемые общие исключения, зависящие лишь от базы данных, а не от конкретного ORM-решения.
- ❑ *Общее управление ресурсами.* Контейнер IoC фреймворка Spring управляет такими ресурсами, как источники данных, соединения с базой данных, объект `SessionFactory` фреймворка Hibernate, объект `EntityManagerFactory` JPA и др. Spring также управляет транзакциями — локальными или глобальными — с помощью JTA.
- ❑ *Централизованное управление транзакциями.* Spring предоставляет возможность декларативного или программного управления транзакциями в приложении. Для декларативного управления транзакциями можно использовать аннотацию `@Transactional`.

Основной подход к интеграции Spring с ORM-решениями — слабое сцепление между уровнями приложения, то есть бизнес-уровнем и уровнем доступа к данным. При таком подходе мы получаем чистое разбиение приложения на уровни, не зависящее от конкретной базы данных и технологии выполнения транзакций. Бизнес-сервисы приложения больше не зависят от доступа к данным и конкретной стратегии управления транзакциями. А поскольку фреймворк Spring управляет ресурсами, используемыми на уровне интеграции, то нет необходимости находить ресурсы для конкретных технологий доступа к данным. Spring предоставляет шаблонные методы для ORM-решений, позволяющие избавиться от стереотипного кода, и обеспечивает единообразный подход при использовании любого из ORM-решений.

В главе 7 вы видели, как Spring решает в приложении две основные проблемы уровня интеграции. Первая из них — *избыточный код управления ресурсами из приложения*, а вторая — *обработка проверяемых исключений* в приложении во время разработки. Модуль ORM фреймворка Spring также решает эти две проблемы, как вы увидите в следующих разделах.

Управление ресурсами и транзакциями

В модуле JDBC фреймворка Spring управление такими ресурсами, как соединения и операторы, а также обработку исключений берет на себя объект `JdbcTemplate`. Он же транслирует коды ошибок SQL конкретной базы данных в осмысленные классы непроверяемых исключений. То же самое справедливо и для модуля ORM фреймворка Spring — Spring управляет как локальными, так и глобальными транзакциями корпоративных приложений с помощью соответствующих диспетчеров транзакций Spring. Spring предоставляет диспетчеры транзакций для всех поддерживаемых технологий ORM. Например, Spring предоставляет диспетчер транзакций для Hibernate, JPA и поддержку JTA для глобальных или распределенных транзакций.

Единообразная обработка и трансляция исключений

В модуле JDBC фреймворка Spring имеется класс `DataAccessException`, предназначенный для обработки всех кодов ошибок SQL, относящихся к конкретной базе данных, и генерации на их основе осмысленных классов исключений. В модуле ORM, как вы уже знаете, Spring поддерживает интеграцию нескольких ORM-решений, в частности Hibernate, JPA, JDO в DAO, и у каждой из этих технологий сохранения имеются свои классы исключений, например `HibernateException`, `PersistenceException` или `JDOException`, в зависимости от технологии. Эти нативные исключения фреймворков ORM представляют собой непроверяемые исключения, так что обрабатывать их в приложении не требуется. Сторона, вызывающая сервисы DAO, не может выполнять нужную ей обработку, разве что приложение в значительной степени основано на ORM или не требует особой обработки исключений. Spring предлагает единообразный подход для всех фреймворков ORM, разработчику не нужно писать в приложении Spring отдельный код для какого-либо из них. С помощью аннотации `@Repository` становится возможной трансляция исключений. Любой аннотированный `@Repository` класс приложения Spring подходит для трансляции исключений в иерархию `DataAccessException` фреймворка Spring. Рассмотрим, например, следующий код класса `AccountDaoImpl`:

```
@Repository
public class AccountDaoImpl implements AccountDao {
    // Здесь располагается тело класса...
}

<beans>
    <!-- Компонент-постобработчик для трансляции исключений -->
    <bean class="org.springframework.dao.annotation.
        PersistenceExceptionTranslationPostProcessor"/>
    <bean id="accountDao" class="com.packt.patterninspring.chapter8.
        bankapp.dao.AccountDaoImpl"/>
</beans>
```

Как можно видеть из этого кода, класс `PersistenceExceptionTranslationPostProcessor` представляет собой постпроцессор компонентов, автоматически выполняющий поиск трансляторов исключений. Он снабжает советом все зарегистрированные компоненты в контейнере, аннотированные `@Repository`. Найденные им трансляторы исключений применяются к этим аннотированным компонентам, после чего могут перехватывать сгенерированные исключения и транслировать их.

Рассмотрим еще несколько паттернов проектирования, которые были реализованы в модуле ORM фреймворка Spring, чтобы обеспечить оптимальное корпоративное решение для уровня интеграции корпоративного приложения.

Паттерн «Объект доступа к данным»

«Объект доступа к данным» (Data Access Object, DAO) — чрезвычайно популярный паттерн проектирования для уровня сохранения в приложениях J2EE. Он отделяет уровень бизнес-логики от уровня сохранения. Паттерн DAO основан на объектно-ориентированных принципах инкапсуляции и абстракции. Контекст использования паттерна DAO — доступ к данным и их сохранение в зависимости от конкретной реализации и типа хранилища, например объектно-ориентированной базы данных, неструктурированных файлов, реляционных баз данных и т. д. На основе паттерна DAO можно создать интерфейс DAO и реализовать его, чтобы абстрагировать и инкапсулировать все обращения к источнику данных. Подобная реализация DAO управляет такими ресурсами базы данных, как соединения с источником данных.

Интерфейсы DAO очень легко адаптируются ко всем нижележащим механизмам источников данных, их не нужно заменять при изменениях в технологиях сохранения на более низких уровнях. Этот паттерн позволяет внедрять различные технологии доступа к данным, никак не затрагивая бизнес-логику корпоративного приложения. Рассмотрим рис. 8.1, чтобы лучше понять принципы паттерна DAO.

Как можно видеть на схеме, в паттерне участвуют следующие объекты.

- ❑ **BusinessObject** — объект, работающий на бизнес-уровне, — клиент для уровня доступа к данным. Ему нужны данные, чтобы моделировать бизнес-процессы, а также подготавливать Java-объекты для вспомогательных функций или контроллеров приложения.
- ❑ **DataAccessObject** — основной объект паттерна DAO. Скрывает от **BusinessObject** всю низкоуровневую реализацию работы нижележащей базы данных.
- ❑ **DataSource** — тоже объект, содержащий всю низкоуровневую информацию о том, что именно представляет собой нижележащая база данных: RDBMS, неструктурированные файлы или XML.
- ❑ **TransferObject** — объект, используемый как носитель информации. Применяется объектом **DataAccessObject** для возврата данных объекту **BusinessObject**.

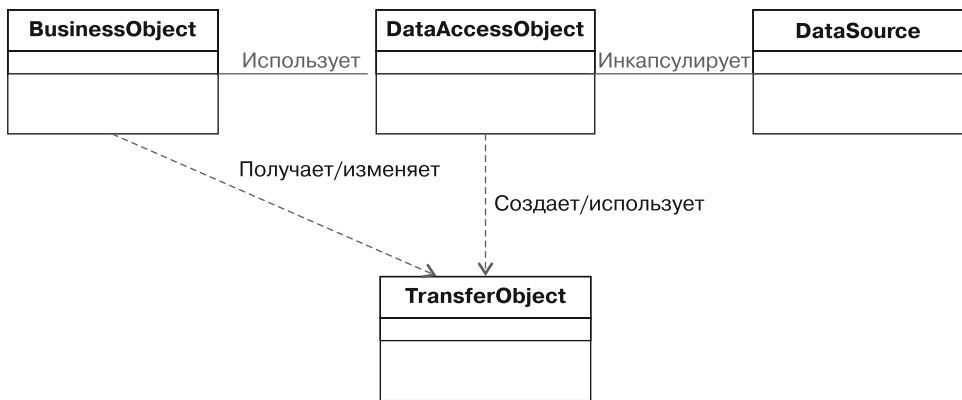


Рис. 8.1. Принципы работы паттерна DAO

Рассмотрим следующий пример паттерна DAO, в котором `AccountDao` представляет собой интерфейс объекта `DataAccessObject`, а `AccountDaoImpl` — класс, реализующий интерфейс `AccountDao`:

```

public interface AccountDao {
    Integer totalAccountsByBranch(String branchName);
}

public class AccountDaoImpl extends JdbcDaoSupport implements
AccountDao {
    @Override
    public Integer totalAccountsByBranch(String branchName) {
        String sql = "SELECT count(*) FROM Account WHERE branchName =
        "+branchName;
        return this.getJdbcTemplate().queryForObject(sql,
        Integer.class);
    }
}

```

Создание объектов DAO в Spring с помощью паттерна проектирования «Фабрика»

Как вы знаете, во фреймворке Spring задействуется множество паттернов проектирования. Паттерн «Фабрика» представляет собой порождающий паттерн проектирования и используется для создания объекта без раскрытия клиенту нижележащей логики, а также назначения вызывающей стороне нового объекта с помощью общего интерфейса или абстрактного класса. Благодаря паттернам проектирования «Фабричный метод» и «Абстрактная фабрика» можно достичь очень высокой гибкости паттерна DAO.

Выясним, где в нашем примере мы реализуем стратегию, в которой фабрика производит объекты DAO для реализации общей базы данных (рис. 8.2).

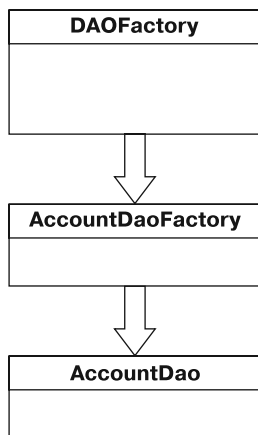


Рис. 8.2. Фабрика производит объекты DAO

Вы можете увидеть на предыдущей схеме, что `AccountDaoFactory` производит объект `AccountDao`, то есть является для него фабрикой. Нижележащую базу данных можно заменить в любой момент, при этом бизнес-код менять не нужно — фабрика берет эту работу на себя. Spring поддерживает хранение всех DAO в фабрике компонентов, а также в фабрике DAO.

Паттерн «Отображение данных»

Уровень отображения данных переносит данные между объектами и базой данных, сохраняя их независимость друг от друга и самих подпрограмм отображения.

Мартин Фаулер. Шаблоны корпоративных приложений¹

Фреймворк ORM обеспечивает отображение между объектами и реляционными базами данных, ведь объекты и таблицы реляционных баз данных по-разному хранят данные приложения. Кроме того, в объектах и таблицах есть различные механизмы структурирования данных. При использовании в приложении Spring любого ORM-решения, например Hibernate, JPA или JDO, нет нужды волноваться о механизме отображения между объектами и реляционными базами данных.

Рассмотрим рис. 8.3, чтобы лучше разобраться с паттерном «Отображение данных» (Data Mapper).

¹ Фаулер М. Шаблоны корпоративных приложений. — М.: Вильямс, 2018.

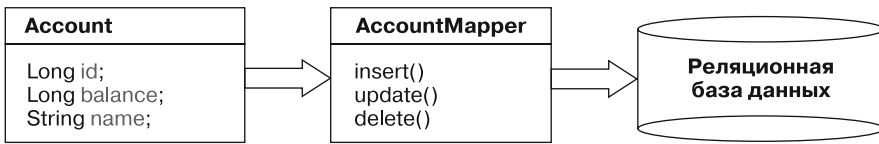


Рис. 8.3. Принципы работы паттерна «Отображение данных»

На схеме происходит отображение объекта `Account` в реляционную базу данных посредством интерфейса `AccountMapper`. `AccountMapper` играет в приложении роль посредника между Java-объектом и нижележащей базой данных. Рассмотрим еще один паттерн, используемый на уровне доступа к данным.

Паттерн «Модель предметной области»

Представляет собой модель предметной области, включающую как поведение, так и данные.

Мартин Фаулер. Шаблоны корпоративных приложений

Модель предметной области — объект, у которого есть и данные, и поведение, где поведение определяет бизнес-логику корпоративного приложения, а данные представляют собой информацию о результатах бизнеса. Модель предметной области объединяет данные и технологический процесс. В корпоративном приложении модель данных располагается ниже бизнес-уровня и определяет бизнес-логику, возвращая результаты бизнес-поведения. Рассмотрим следующую схему для большей ясности (рис. 8.4).

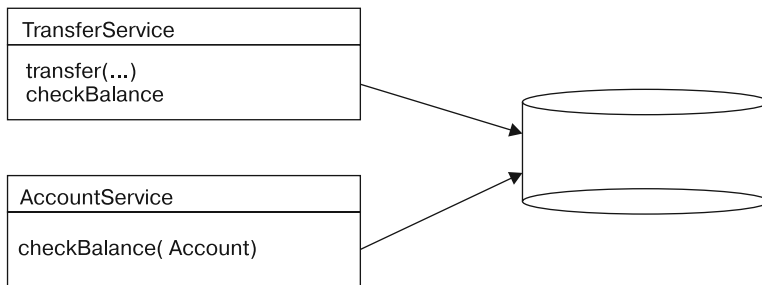


Рис. 8.4. Приложение с двумя моделями предметной области

Как можно видеть на схеме, в приложении заданы две модели предметной области в соответствии с бизнес-требованиями. Бизнес-алгоритм перевода денег с одного счета на другой описан в классе `TransferService`. Классы `TransferService` и `AccountService` относятся к паттерну «Модель предметной области» в корпоративном приложении.

Прокси для паттерна «Отложенная загрузка»

«Отложенная загрузка» — паттерн проектирования, используемый некоторыми из ORM-решений, например Hibernate, в корпоративных приложениях, чтобы отложить инициализацию объекта до момента обращения к нему другого объекта, то есть момента, когда он понадобится. Цель этого паттерна проектирования состоит в оптимизации памяти в приложении. Паттерн проектирования «Отложенная загрузка» в Hibernate реализуется с помощью виртуального объекта-прокси. При демонстрации отложенной загрузки мы используем прокси, но он не относится к паттерну «Заместитель».

Паттерн «Шаблонный метод» для поддержки Hibernate в Spring

Фреймворк Spring предоставляет вспомогательный класс для доступа к данным на уровне DAO, основанный на паттерне проектирования «Шаблонный метод» GoF. Класс `HibernateTemplate` фреймворка Spring поддерживает такие операции баз данных, как `save`, `create`, `delete` и `update`. Этот класс гарантирует, что для каждой транзакции используется только один сеанс Hibernate.

Интеграция Hibernate со Spring

Hibernate — ORM-фреймворк сохранения с открытым исходным кодом, который не только обеспечивает отображение простых объектных взаимосвязей между Java-объектами и таблицами базы данных, но и предлагает множество продвинутых возможностей для повышения производительности приложения, а также помогает улучшить использование ресурсов, например с помощью кэширования, отложенной загрузки, немедленного извлечения данных и распределенного кэширования.

Spring обеспечивает полную поддержку интеграции с фреймворком Hibernate, у него имеется несколько встроенных библиотек, позволяющих задействовать Hibernate на все 100 %. Для задания настроек Hibernate в приложении можно воспользоваться паттерном внедрения зависимостей и контейнером IoC фреймворка Spring.

В следующем разделе мы выясним, как правильно настроить Hibernate в контейнере IoC фреймворка Spring.

Задание настроек объекта SessionFactory фреймворка Hibernate в контейнере Spring

Оптимальный подход к настройке Hibernate и других технологий сохранения в любом корпоративном приложении состоит в разделении бизнес-объектов с жестко «защитыми» справочниками ресурсов, такими как `DataSource` в JDBC или `SessionFactory` в Hibernate. Эти ресурсы можно описать в виде компонентов

в контейнере Spring. Но для доступа к ним бизнес-объектам необходимы ссылки на них. Рассмотрим следующий класс DAO, использующий объект `SessionFactory` для получения данных для приложения:

```
public class AccountDaoImpl implements AccountDao {
    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
    // ...
}
```

Как можно видеть, класс DAO, `AccountDaoImpl`, следует паттерну внедрения зависимостей. Для доступа к данным в него внедряется объект `SessionFactory` фреймворка Hibernate, и он прекрасно чувствует себя в контейнере IoC Spring. Объект `SessionFactory` фреймворка Hibernate представляет собой объект-одиночку, он генерирует основной объект интерфейса Hibernate `org.hibernate.Session`. Объект `SessionFactory` управляет объектом `Session`, а также отвечает за его открытие и закрытие. Интерфейс `Session` содержит реальную функциональность доступа к данным — сохранения (`save`), обновления (`update`), удаления (`delete`) и загрузки (`load`) объектов из базы данных. В приложении объект класса `AccountDaoImpl` или любой другой репозиторий выполняет с помощью этого объекта `Session` все необходимые операции хранения данных.

Фреймворк Spring предоставляет встроенные модули Hibernate, так что вы можете использовать в своих приложениях компоненты `SessionFactory` Hibernate.

Компонент `org.springframework.orm.hibernate5.LocalSessionFactoryBean` представляет собой реализацию интерфейса `FactoryBean` фреймворка Spring. `LocalSessionFactoryBean` основан на паттерне «Абстрактная фабрика», он генерирует в приложении объект `SessionFactory`. Этот объект можно сконфигурировать в виде компонента в контексте Spring приложения следующим образом:

```
@Bean
public LocalSessionFactoryBean sessionFactory(DataSource
dataSource) {
    LocalSessionFactoryBean sfb = new LocalSessionFactoryBean();
    sfb.setDataSource(dataSource);
    sfb.setPackagesToScan(new String[] {
        "com.packt.patterninspring.chapter8.bankapp.model" });
    Properties props = new Properties();
    props.setProperty("dialect",
        "org.hibernate.dialect.H2Dialect");
    sfb.setHibernateProperties(props);
    return sfb;
}
```

В этом коде мы сконфигурировали объект `SessionFactory` в виде компонента с помощью класса `LocalSessionFactoryBean` фреймворка Spring. Метод этого компонента принимает на входе объект типа `DataSource` в качестве аргумента, который

определяет место и способ соединения с базой данных. Мы также указали, какой пакет просматривать, задав значение `com.packt.patterninspring.chapter8.bankapp.model` для свойства `setPackagesToScan` компонента `LocalSessionFactoryBean`, и задали свойство `dialect` компонента `SessionFactory` с помощью метода `setHibernateProperties` для указания того, с каким типом базы данных мы имеем дело в приложении.

Теперь, после настройки компонента `SessionFactory` Hibernate, в контексте приложения Spring посмотрим, как можно реализовать объекты доступа к данным для уровня сохранения нашего приложения.

Реализация объектов DAO на основе простого API Hibernate

Создадим такой DAO-класс, реализующий интерфейс:

```
package com.packt.patterninspring.chapter8.bankapp.dao;

import org.hibernate.SessionFactory;
import org.springframework.stereotype.Repository;
import org.springframework.beans.factory.annotation.Autowired;
@Repository
public class AccountDaoImpl implements AccountDao {
    @Autowired
    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
    @Override
    public Integer totalAccountsByBranch(String branchName) {
        String sql = "SELECT count(*) FROM Account WHERE branchName = "
            + branchName;
        return this.sessionFactory.getCurrentSession().createQuery(sql,
            Integer.class).getSingleResult();
    }
    @Override
    public Account findOne(long accountId) {
        return (Account)
            this.sessionFactory.getCurrentSession().get(Account.class, accountId);
    }
    @Override
    public Account findByName(String name) {
        return (Account)
            this.sessionFactory.getCurrentSession().createCriteria(Account.class)
                .add(Restrictions.eq("name", name))
                .list().get(0);
    }
}
```



```

@Override
public List<Account> findAllAccountInBranch(String branchName) {
    return (List<Account>) this.sessionFactory.currentSession()
        .createCriteria(Account.class).add(Restrictions.eq("branchName",
            branchName)).list();
}
}

```

Здесь `AccountDaoImpl` представляет собой класс реализации паттерна DAO, в который с помощью аннотации `@Autowired` фреймворка Hibernate внедряется компонент `Hibernate SessionFactory`. Описанные ранее реализации DAO генерируют непроверяемые исключения Hibernate типа `PersistenceExceptions`, которым нежелательно разрешать распространяться вверх до уровня сервисов или других пользователей объектов DAO. Но модуль AOP фреймворка Spring дает возможность трансляции их в обширную и независимую от поставщика иерархию исключений `DataAccessException`, благодаря чему скрывается используемая технология доступа к данным. Это стандартная возможность фреймворка Spring, доступная при снабжении класса реализации интерфейса DAO аннотацией `@Repository`, — вам остается только описать предоставляемый Spring постпроцессор компонентов, а именно `PersistenceExceptionTranslationPostProcessor`.

Добавим в класс реализации DAO для Hibernate трансляцию исключений. Для этого достаточно добавить в контекст приложения Spring компонент `PersistenceExceptionTranslationPostProcessor`:

```

@Bean
public BeanPostProcessor persistenceTranslation() {
    return new PersistenceExceptionTranslationPostProcessor();
}

```

Вышеприведенный зарегистрированный компонент `PersistenceExceptionTranslationPostProcessor` отвечает за добавление объекта класса `Advisor` в компоненты, снабженные аннотацией `@Repository`, и за повторную генерацию любых перехваченных платфомерноориентированных исключений в виде непроверяемых Spring-ориентированных исключений доступа к данным.

В следующем разделе мы рассмотрим, как фреймворк Spring управляет транзакциями на бизнес-уровне и уровне сохранения приложения Spring.

Стратегии управления транзакциями в Spring

Spring предоставляет всестороннюю поддержку управления транзакциями в приложениях Spring. Это одна из наиболее интересных возможностей фреймворка. В основном именно она вынуждает предприятия-разработчики ПО создавать корпоративные приложения с помощью фреймворка Spring. Фреймворк обеспечивает

единообразное управление транзакциями в пределах всего приложения при использовании любой технологии сохранения — API транзакций Java, JDBC, Hibernate, API сохранения Java (JPA) и объектов данных Java (JDO). Spring поддерживает как декларативное, так и программное управление транзакциями.

В Java есть два типа транзакций.

- ❑ *Локальные транзакции.* Позволяют работать с одним ресурсом. Такие транзакции ориентированы на работу с конкретным нижележащим ресурсом и управляются им. Рассмотрим это на следующей схеме (рис. 8.5).



Рис. 8.5. Локальные транзакции

Транзакции, выполняемые между приложением и базой данных, гарантируют, что каждый фрагмент решаемой задачи удовлетворяет набору требований ACID базы данных.

- ❑ *Глобальные транзакции.* Позволяют работать с несколькими ресурсами. Глобальные транзакции, которыми управляют отдельные, специализированные диспетчеры транзакций, позволяют работать с несколькими транзакционными ресурсами. Рассмотрим следующую схему, проливающую свет на функционирование глобальных и распределенных транзакций (рис. 8.6).

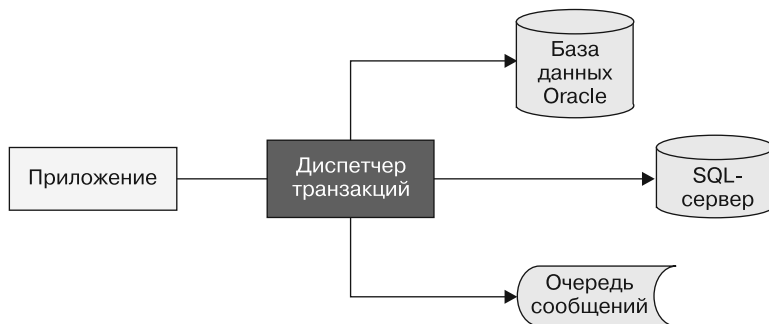


Рис. 8.6. Глобальные транзакции

Диспетчер транзакций работает сразу с несколькими технологиями баз данных, используемыми в приложении. Глобальная транзакция не зависит от ориентированных на конкретную платформу технологий сохранения.

Spring обеспечивает один и тот же API для обоих типов транзакций в приложениях Java. Фреймворк Spring гарантирует единообразную модель программи-

рования в любых средах за счет или декларативного, или программного конфигурирования транзакций.

Перейдем к следующим разделам, в которых поговорим о том, как конфигурировать транзакции в приложениях Spring.

Декларативное задание границ и реализация транзакций

Фреймворк Spring поддерживает декларативное управление транзакциями, причем разделяет задание границ транзакций с их реализацией. Границы транзакций устанавливаются декларативно, посредством AOP Spring. Рекомендуется декларативно задавать границы транзакций и реализовывать их в приложениях Spring, поскольку декларативная модель программирования дает возможность заменять из своего кода внешний API задания границ и конфигурировать их с помощью перехватчиков транзакций AOP Spring. Транзакции, по сути, являются сквозной функциональностью, декларативная модель транзакций позволяет разделять бизнес-логику приложения с повторяющимся кодом установления границ транзакций.

Как уже упоминалось, Spring обеспечивает единообразную модель обработки транзакций в приложениях Spring и предоставляет интерфейс `PlatformTransactionManager` для скрытия деталей реализации. Во фреймворке Spring имеется несколько реализаций этого интерфейса, часть из которых перечислена ниже:

- ❑ `DataSourceTransactionManager`;
- ❑ `HibernateTransactionManager`;
- ❑ `JpaTransactionManager`;
- ❑ `JtaTransactionManager`;
- ❑ `WebLogicJtaTransactionManager`;
- ❑ `WebSphereUowTransactionManager`.

Вот ключевой интерфейс:

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(
        TransactionDefinition definition) throws TransactionException;
    void commit(TransactionStatus status) throws
        TransactionException;
    void rollback(TransactionStatus status) throws
        TransactionException;
}
```

В предыдущем коде метод `getTransaction()` возвращает объект типа `TransactionStatus`. Этот объект содержит информацию о состоянии транзакции: новая ли она, или речь идет об участии в уже существующей транзакции из текущего стека вызовов.

Важную роль в нем играет параметр `TransactionDefinition`. Как и в JDBC- и ORM-модулях, Spring также обеспечивает единообразный способ обработки исключений, сгенерированных каким-либо из диспетчеров транзакций. Метод `getTransaction()` может генерировать непроверяемое исключение `TransactionException`.

В приложениях Spring один и тот же API используется как для глобальных, так и для локальных транзакций. Чтобы перейти от локальной транзакции к глобальной, понадобятся очень небольшие изменения — лишь поменять диспетчер транзакций.

Развертывание диспетчера транзакций

Развертывание диспетчера транзакций в приложении Spring осуществляется в два этапа. На первом этапе необходимо реализовать в приложении класс диспетчера транзакций Spring или настроить уже реализованный. Второй этап — объявление границ транзакций, то есть того, что транзакции Spring должны в себя включать.

Этап 1: реализация диспетчера транзакций

Создайте компонент для требуемой реализации аналогично созданию любого другого компонента Spring. Диспетчер транзакций можно настроить для применения (при необходимости) любой технологии сохранения, например JDBC, JMS, JTA, Hibernate, JPA и т. д. В следующем примере приведен использующий JDBC диспетчер транзакций для `DataSource`.

В Java-конфигурации описание компонента `transactionManager` в приложении выглядит следующим образом:

```
@Bean
public PlatformTransactionManager transactionManager(DataSource
dataSource) {
    return new DataSourceTransactionManager(dataSource);
}
```

В XML-конфигурации данный компонент можно создать так:

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.
DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

В предыдущем коде мы воспользовались компонентом `dataSource`, который должен быть описан где-то в другом месте. Идентификатор компонента, `transactionManager`, представляет собой название по умолчанию. Его можно поменять, но в этом случае необходимо везде прописать другое название, что может оказаться непростой задачей!

Этап 2: объявление границ транзакций

Оптимальным вариантом будет объявить границы транзакций на уровне сервисов нашего приложения:

```
@Service
public class TransferServiceImpl implements TransferService{
    // ...
    @Transactional
    public void transfer(Long amount, Long a, Long b){
        // Атомарная операция
    }
    // ...
}
```

Как можно видеть в предыдущем коде, `TransferServiceImpl` представляет собой класс нашего сервиса на уровне сервисов приложения. Этот сервис — идеальное место задания границ транзакций для операций. Spring предоставляет для установления границ аннотацию `@Transactional`, которую можно использовать в приложении на уровне как классов, так и методов классов сервисов. Посмотрим на использование аннотации `@Transactional` на уровне класса:

```
@Service
@Transactional
public class TransferServiceImpl implements TransferService{
    // ...
    public void transfer(Long amount, Account a, Account b){
        // Атомарная операция
    }
    public Long withdraw(Long amount, Account a){
        // Атомарная операция
    }
    // ...
}
```

Если объявить аннотацию `@Transactional` на уровне класса, то все бизнес-методы данного сервиса окажутся транзакционными.



При использовании аннотации `@Transactional` область видимости метода должна быть `public`. При использовании этой аннотации с не общедоступным методом, например `protected`, `private` или `package-visible`, никакой ошибки не произойдет и исключение сгенерировано не будет, но такой аннотированный метод транзакционного поведения демонстрировать не станет.

Но одного использования аннотации `@Transactional` в приложении Spring недостаточно. Необходимо активизировать возможность управления транзакциями фреймворка Spring посредством аннотации `@EnableTransactionManagement`

в Java-конфигурации или пространства имен `<tx:annotation-driven/>` в файле конфигурации в стиле XML. Взгляните, например, на следующий фрагмент кода:

```
@Configuration
@EnableTransactionManagement
public class InfrastructureConfig {
    // Определения других инфраструктурных компонентов
    @Bean
    public PlatformTransactionManager transactionManager(){
        return new DataSourceTransactionManager(dataSource());
    }
}
```

Как можно видеть, класс `InfrastructureConfig` описан в файле Java-конфигурации приложения Spring, и в нем описываются относящиеся к инфраструктуре компоненты. Здесь же описан и один из компонентов `transactionManager`. Этот класс конфигурации снабжен еще и аннотацией `@EnableTransactionManagement`, описывающей в приложении постпроцессор компонентов, который служит прокси для компонентов с аннотацией `@Transactional`. Взглянем теперь на рис. 8.7.

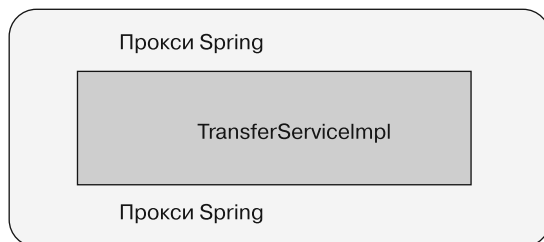


Рис. 8.7. Класс `TransferServiceImpl` внутри прокси

Как можно видеть, класс `TransferServiceImpl` обернут в прокси.

Но что же конкретно происходит со снабженными аннотацией `@Transactional` компонентами в приложении? Рассмотрим следующие шаги.

1. Объект `target` обернут в прокси; он использует совет типа «*везде*», обсуждавшийся в главе 6.
2. Прокси реализует следующее поведение:
 - 1) начать транзакцию перед входом в бизнес-метод;
 - 2) зафиксировать транзакцию по окончании выполнения бизнес-метода;
 - 3) откатить транзакцию, если бизнес-метод сгенерирует исключение `RuntimeException`, — это поведение по умолчанию для транзакций Spring, но его можно переопределить для проверяемых и пользовательских исключений.
3. Теперь контекст транзакции ограничен текущим потоком выполнения.
4. Все шаги контролируются конфигурациями на основе XML, Java или аннотаций.

Рассмотрим теперь схему локальной конфигурации JDBC с включенным в нее диспетчером транзакций (рис. 8.8).

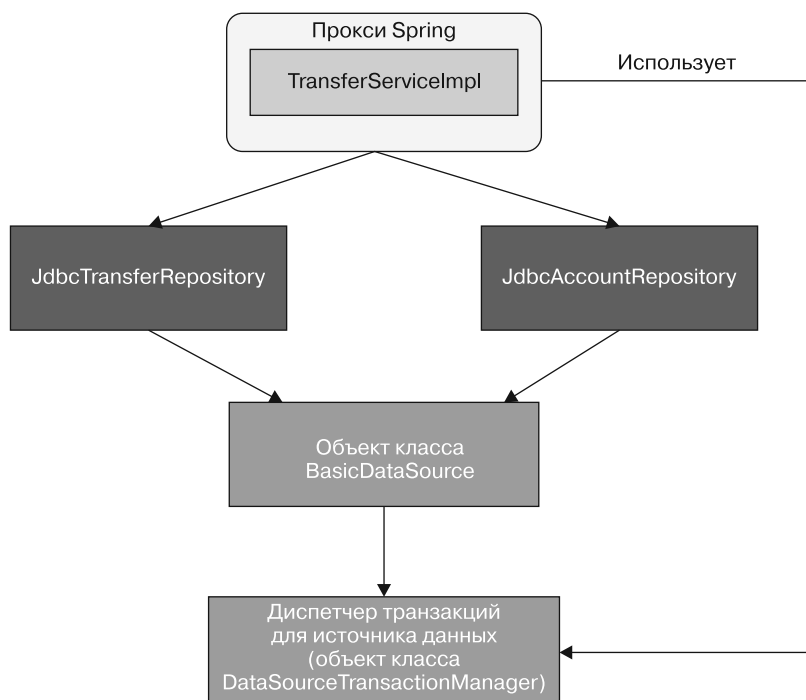


Рис. 8.8. Локальная конфигурация JDBC

На этой схеме описан локальный источник данных с помощью JDBC и диспетчера транзакций для источника данных (объекта класса `DataSourceTransactionManager`).

В следующем разделе мы обсудим реализацию и задание границ транзакций в приложении программным способом.

Программное задание границ и реализация транзакций

Spring предоставляет возможность реализации и задания границ транзакций в приложении программным способом, с помощью непосредственного использования объекта класса `TransactionTemplate` и реализации `PlatformTransactionManager`. Но я настоятельно рекомендую использовать все-таки декларативное управление транзакциями, поскольку в этом случае код оказывается чище, а конфигурация — более гибкой.

Посмотрим на код программной реализации транзакций в приложении:

```
package com.packt.patterninspring.chapter8.bankapp.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionTemplate;

import com.packt.patterninspring.chapter8.bankapp.model.Account;
import com.packt.patterninspring.chapter8.bankapp.
    repository.AccountRepository;

@Service
public class AccountServiceImpl implements AccountService {
    // Все методы этого экземпляра используют один объект
    // класса TransactionTemplate
    private final TransactionTemplate transactionTemplate;
    @Autowired
    AccountRepository accountRepository;
    // Используем внедрение зависимости через конструктор
    // для передачи PlatformTransactionManager
    public AccountServiceImpl(PlatformTransactionManager
        transactionManager) {
        this.transactionTemplate = new
            TransactionTemplate(transactionManager);
    }
    @Override
    public Double checkAccountBalance(Account account) {
        return transactionTemplate.execute(new
            TransactionCallback<Double>() {
                // Код в этом методе выполняется в транзакционном контексте
                public Double doInTransaction(TransactionStatus status) {
                    return accountRepository.checkAccountBalance(account);
                }
            });
    }
}
```

В предыдущем фрагменте кода мы использовали объект `TransactionTemplate` явным образом для выполнения логики приложения в транзакционном контексте. Класс `TransactionTemplate` также основан на паттерне проектирования «Шаблонный метод» и использует тот же подход, что и другие шаблоны фреймворка Spring, такие как класс `JdbcTemplate`. Подобно `JdbcTemplate`, класс `TransactionTemplate` использует подход с обратным вызовом и освобождает код приложения от стереотипного кода управления ресурсами транзакций. Мы сформировали объект класса `TransactionTemplate` в процессе формирования класса сервиса и передали объект класса `PlatformTransactionManager` в качестве аргумента конструктора класса `TransactionTemplate`. Мы также написали реализацию интерфейса

TransactionCallback, которая содержит код бизнес-логики приложения и демонстрирует сильное сцепление между логикой приложения и транзакционным кодом.

В этой главе вы увидели, насколько эффективно Spring управляет транзакциями в корпоративных приложениях. Рассмотрим теперь несколько рекомендуемых практик, которыми не помешает владеть при работе над любым корпоративным приложением.

Рекомендуемые практики для ORM Spring и модуля транзакций приложения

Избегайте использования вспомогательного класса `HibernateTemplate` в реализациях DAO, применяйте в своих приложениях классы `SessionFactory` и `EntityManager`. Воспользуйтесь наличием контекстных сеансов в Hibernate и задействуйте в своих DAO непосредственно `SessionFactory`. Кроме того, используйте метод `getCurrentSession()` для доступа к текущему транзакционному сеансу для выполнения операций сохранения в приложении (см. следующий код):

```
@Repository
public class HibernateAccountRepository implements
AccountRepository {
    SessionFactory sessionFactory;
    public HibernateAccountRepository(SessionFactory
sessionFactory) {
        super();
        this.sessionFactory = sessionFactory;
    }
    // ...
}
```

Всегда используйте в своем приложении аннотацию `@Repository` для объектов доступа к данным или для репозитория — она обеспечивает трансляцию исключений (см. следующий код):

```
@Repository
public class HibernateAccountRepository{//...}
```

Уровень сервисов должен быть отделен, несмотря даже на то, что бизнес-методы сервисов лишь передают свои обязанности на выполнение соответствующим методам объектов DAO.

Всегда реализуйте транзакции на уровне сервисов приложения, а не на уровне DAO — это оптимальное место для транзакций (см. следующий код):

```
@Service
@Transactional
public class AccountServiceImpl implements AccountService {//...}
```

Декларативное управление транзакциями дает больше возможностей и лучше подходит для задания настроек в приложении, а потому в приложениях Spring

настоятельно рекомендуется использовать именно этот подход. Оно разделяет сквозную функциональность с бизнес-логикой.

Всегда генерируйте исключения времени выполнения вместо проверяемых исключений с уровня сервисов.

Осторожно обращайтесь с флагом `readOnly` (только для чтения) аннотации `@Transactional`. Помечайте транзакции как `readOnly=true` лишь тогда, когда методы сервиса содержат только запросы данных.

Резюме

В главе 7 вы узнали, что фреймворк Spring предоставляет класс `JdbcTemplate`, в основе которого лежит паттерн проектирования GoF «Шаблонный метод». Этот класс обеспечивает весь необходимый стереотипный код, лежащий в основе традиционного API JDBC. Но при работе с модулем JDBC фреймворка Spring отображение таблиц в объекты становится очень трудоемким. В этой главе вы увидели решение, позволяющее отображать объекты в таблицы реляционной базы данных, — с помощью ORM можно добиться гораздо большего от реляционной базы данных в сложном приложении. Spring поддерживает интеграцию с несколькими ORM-решениями, такими как Hibernate, JPA и др. Эти фреймворки ORM дают возможность использовать декларативную модель программирования для сохранения данных вместо модели программирования JDBC.

Мы рассмотрели несколько паттернов проектирования, реализованных на уровне доступа к данным и уровне интеграции. Эти паттерны реализованы в виде возможностей фреймворка Spring, например паттерн «Заместитель» для отложенной загрузки, паттерн «Фасад» для интеграции с бизнес-уровнем, паттерны DAO для доступа к данным и т. д.

В следующей главе мы рассмотрим возможности улучшения производительности приложения в промышленной эксплуатации за счет поддержки фреймворком Spring паттернов кэширования.

9

Улучшение производительности приложения с помощью паттернов кэширования

Из предыдущих глав вы узнали, как фреймворк Spring получает данные для приложения в прикладной части. Вы также познакомились со вспомогательным классом `JdbcTemplate`, предоставляемым модулем JDBC фреймворка Spring для доступа к данным. Spring поддерживает интеграцию с ORM-решениями, такими как Hibernate, JPA, JDO и др., и управляет транзакциями во всем приложении. В этой главе мы исследуем поддержку фреймворком Spring кэширования для улучшения производительности.

Бывало ли так, что, поздно вернувшись домой из офиса, вы сталкивались с градом вопросов от жены? Да, я знаю, необходимость отвечать на такое количество вопросов, будучи уставшим и обессиленным, очень раздражает. Еще больше раздражает, когда вам задают один и тот же вопрос снова и снова.

На одни вопросы можно ответить просто «Да» или «Нет», а на другие придется давать подробные пояснения. Представьте себе, что вам снова задают один из таких требующих подробного ответа вопросов спустя некоторое время! Аналогично этому приложение может быть спроектировано таким образом, что некоторые компоненты задают одни и те же вопросы снова и снова, чтобы выполнить каждую задачу отдельно. Как и для вопросов, задаваемых вашей женой, для некоторых вопросов в системе требуется потратить немало времени на извлечение соответствующих данных — в этом может участвовать какая-либо сложная логика, данные могут извлекаться из базы данных либо может требоваться обращение к удаленному сервису.

Если заранее известно, что ответ на вопрос вряд ли будет часто меняться, то можно запомнить его на будущее, чтобы вернуть, когда соответствующий вопрос будет снова задан той же системой. Нет смысла проходить тот же путь для извлечения данных в ущерб производительности приложения — это будет напрасной тратой ресурсов. В корпоративных приложениях кэширование представляет собой способ сохранения таких часто требующихся ответов, чтобы извлекать их из кэша вместо прохождения обычного пути для получения одного и того же ответа снова и снова. В этой главе мы обсудим абстракцию кэша — возможность фреймворка

Spring и поддержку им декларативной реализации кэширования. Мы охватим следующие вопросы.

- ☐ Что такое кэш.
- ☐ В каких случаях применяется кэширование.
- ☐ Абстракция кэша.
- ☐ Включение возможности кэширования посредством паттерна «Заместитель».
- ☐ Декларативное кэширование с помощью аннотаций.
- ☐ Декларативное кэширование с помощью XML.
- ☐ Настройка хранилища кэша.
- ☐ Реализация пользовательских аннотаций кэширования.
- ☐ Рекомендуемые практики кэширования.

Что такое кэш

Если говорить простым языком, *кэш* — блок памяти, где хранится предварительно обработанная информация для приложения. В этом смысле хранилище пар «ключ — значение», например ассоциативный массив, может служить в приложении в качестве кэша. В Spring кэш — это интерфейс, абстрагирующий и символизирующий кэширование. Интерфейс кэширования предоставляет методы для помещения объектов в хранилище кэша, извлечения объектов оттуда по заданному ключу, обновления объектов в хранилище кэша по заданному ключу и удалению объектов оттуда по заданному ключу. Этот интерфейс кэширования предлагает множество функций для работы с кэшем.

В каких случаях применяется кэширование? Кэширование применяется в тех случаях, когда метод, вызванный с одним (-и) аргументом (-ами), всегда возвращает один и тот же результат. Делать такой метод может все что угодно, например выполнять вычисления над данными на лету, выполнять запросы базы данных, запрашивать данные через RMI, JMS или веб-сервис и т. д. На основе аргументов должен генерироваться уникальный ключ, являющийся кэш-ключом.

Абстракция кэша

По сути, кэширование в Java-приложениях используется для Java-методов, чтобы снизить количество их выполнений для получения одной и той же информации, имеющейся в кэше. Это значит, что при каждом вызове этих Java-методов абстракция кэша реализует для них алгоритм работы кэша на основе заданных аргументов. Если для заданного аргумента информация уже имеется в кэше, она возвращается без выполнения целевого метода. Если же в кэше нужной информации нет, вызывается целевой метод, а результат кэшируется и возвращается вызывающей стороне. Абстракция кэша предоставляет и другие относящиеся к кэшу операции, например обновление или удаление содержимого кэша. Эти операции иногда бывают нужны при изменениях данных приложения.

Фреймворк Spring предоставляет абстракцию кэша для приложений Spring посредством использования интерфейсов `org.springframework.cache.Cache` и `org.springframework.cache.CacheManager`. Для хранения данных при кэшировании требуется фактическое хранилище. Но абстракция кэша обеспечивает лишь логику кэширования, но не предоставляет никакого физического хранилища для закэшированных данных. Поэтому разработчику необходимо реализовать в приложении фактическое хранилище для закэшированных данных. Если речь идет о распределенном приложении, то поставщик кэширования должен быть настроен соответствующим образом: в зависимости от сценариев использования приложения можно или сделать копии одних и тех же данных во всех узлах распределенного приложения, или создать централизованный кэш.

На рынке существует несколько поставщиков функциональности кэширования, которые вы можете использовать в зависимости от требований вашего приложения. Вот некоторые из них:

- ☐ Redis;
- ☐ OrmLiteCacheClient;
- ☐ Memcached;
- ☐ In Memory Cache;
- ☐ Aws DynamoDB Cache Client;
- ☐ Azure Cache Client.

Для реализации абстракции кэша в приложении необходимо позаботиться о решении следующих задач.

- ☐ *Объявление кэширования.* Нужно отметить требующие кэширования методы приложения и либо аннотировать эти методы с помощью аннотаций кэширования, либо воспользоваться XML-конфигурацией с помощью модуля AOP Spring.
- ☐ *Конфигурация кэша.* Следует настроить фактическое хранилище для закэшированных данных — хранилище, где будут находиться данные и откуда они будут читаться.

Рассмотрим теперь, как активизировать абстракцию кэша Spring в приложении Spring.

Включение возможности кэширования посредством паттерна «Заместитель»

Активизировать абстракцию кэша Spring можно следующими двумя способами:

- ☐ с помощью аннотаций;
- ☐ с помощью пространства имен XML.

Фреймворк Spring прозрачным образом применяет кэширование к методам компонентов Spring с помощью модуля AOP. Spring добавляет к компонентам Spring прокси, в котором можно объявить кэшируемые методы. Этот прокси добавляет

динамическое поведение кэширования к компонентам Spring. Рисунок 9.1 иллюстрирует это поведение.

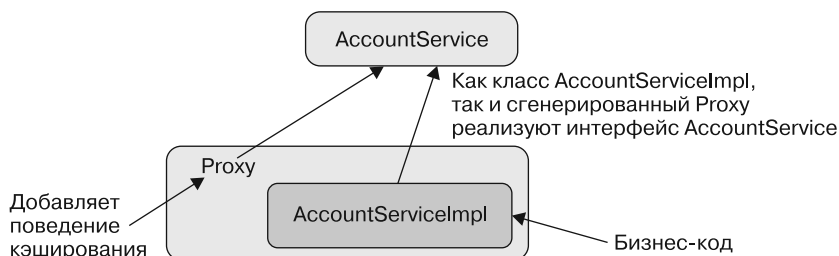


Рис. 9.1. Кэширование в Spring

На схеме можно видеть, что фреймворк применяет адаптер «Заместитель» к `AccountServiceImpl` для добавления поведения кэширования. Spring использует паттерн «Заместитель» из GoF для реализации кэширования в приложении.

Взглянем теперь, как включить эту возможность в приложении Spring.

Включение прокси для кэширования с помощью аннотаций

Как вы уже знаете, у Spring имеется множество возможностей, но большая часть их отключена. Их необходимо включить, прежде чем использовать. Если вы хотели бы использовать абстракцию кэша фреймворка Spring в своем приложении, то вам нужно включить эту возможность. При использовании Java-конфигурации можно включить абстракцию кэша Spring путем добавления аннотации `@EnableCaching` к одному из классов конфигурации. Следующий класс конфигурации демонстрирует использование аннотации `@EnableCaching`:

```
package com.packt.patterninspring.chapter9.bankapp.config;

import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.concurrent.ConcurrentMapCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages=
{"com.packt.patterninspring.chapter9.bankapp"})
@EnableCaching // Включаем кэширование
public class AppConfig {
    @Bean
    public AccountService accountService() { ... }

    // Объявляем диспетчер кэширования
```

```
@Bean
public CacheManager cacheManager() {
    CacheManager cacheManager = new ConcurrentMapCacheManager();
    return cacheManager;
}
```

В вышеприведенном файле Java-конфигурации мы добавили к классу конфигурации `AppConfig.java` аннотацию `@EnableCaching`; эта аннотация указывает фреймворку Spring на необходимость включить поведение кэширования для данного приложения.

Посмотрим теперь, как включить абстракцию кэша Spring с помощью XML-конфигурации.

Включение прокси для кэширования с помощью пространства имен XML

При задании настроек приложения с помощью XML-конфигурации включить ориентированное на работу с аннотациями кэширование можно с помощью элемента `<cache:annotation-driven>` пространства имен `cache` фреймворка Spring, 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:context="http://www.springframework.org/schema/context"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:cache="http://www.springframework.org/schema/cache"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
        http://www.springframework.org/schema/jdbc/spring-jdbc-4.3.xsd
        http://www.springframework.org/schema/cache
        http://www.springframework.org/schema/cache/spring-cache-4.3.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/aop
        http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-4.3.xsd">
    <!-- Включаем возможность кэширования -->
    <cache:annotation-driven />
    <context:component-scan base-
        package="com.packt.patterninspring.chapter9.bankapp"/>
    <!-- Объявляем диспетчер кэширования -->
    <bean id="cacheManager"
        class="org.springframework.cache.concurrent.ConcurrentMapCacheManager" />
</beans>
```

Как видно из этих файлов конфигурации, независимо от того, используете ли вы Java- или XML-конфигурацию, с помощью аннотации `@EnableCaching` или пространства имен `<cache:annotation-driven>` можно активизировать абстракцию кэша Spring путем создания аспекта со срезами, которые задействуются аннотациями кэширования Spring.

Посмотрим теперь, как можно воспользоваться аннотациями кэширования фреймворка Spring для задания границ кэша.

Декларативное кэширование с помощью аннотаций

В приложениях Spring абстракция Spring предоставляет следующие аннотации для объявления кэширования.

- ❑ `@Cacheable` — позволяет перед выполнением метода посмотреть, нет ли в кэше возвращаемого этим методом значения. Если значение есть — вернуть его, если нет, то вызвать метод и поместить возвращенное им значение в кэш.
- ❑ `@CachePut` — дает возможность обновить кэш без проверки того, есть там значение или нет. Всегда приводит к вызову метода.
- ❑ `@CacheEvict` — отвечает за запуск процесса вытеснения данных из кэша.
- ❑ `@Caching` — используется для группировки нескольких аннотаций с целью одно-временного применения к методу.
- ❑ `@CacheConfig` — позволяет задавать некоторые часто используемые настройки, связанные с кэшированием, на уровне класса, с тем чтобы фреймворк Spring дальше распространял их на методы в качестве значений по умолчанию.

Рассмотрим каждую из этих аннотаций подробнее.

Аннотация `@Cacheable`

Аннотация `@Cacheable` отмечает метод как кэшируемый. Результат его выполнения сохраняется в кэше. При всех последующих вызовах этого метода с теми же аргументами данные будут извлекаться из кэша по ключу, а сам метод выполняться не будет. Вот некоторые из атрибутов аннотации `@Cacheable`:

- ❑ `value` — название используемого кэша;
- ❑ `key` — ключ для каждого кэшируемого элемента данных;
- ❑ `condition` — выражение языка SpEL¹, результатом вычисления которого является истина или ложь; в последнем случае результат кэширования не применяется к данному вызову метода;
- ❑ `unless` — тоже выражение языка SpEL; при истинном значении возвращаемое значение не будет помещено в кэш.

¹ Язык выражений Spring (Spring Expression Language).

Можно использовать в этой аннотации SpEL вместе с аргументом (-ами) метода. В следующем коде приведено простейшее объявление аннотации `@Cacheable`. Оно требует указания соответствующего методу названия кэша:

```
@Cacheable("accountCache ")
public Account findAccount(Long accountId) {...}
```

В предыдущем коде метод `findAccount` снабжен аннотацией `@Cacheable`. Это значит, что данный метод связан с кэшем. Название этого кэша — `accountCache`. При каждом вызове этого метода для конкретного `accountId` происходит проверка кэша на наличие в нем возвращаемого значения данного метода для этого `accountId`. Можно также присваивать кэшу несколько названий, как показано далее:

```
@Cacheable({"accountCache ", "saving-accounts"})
public Account findAccount(Long accountId) {...}
```

Аннотация `@CachePut`

Как упоминалось ранее, задача у аннотаций `@Cacheable` и `@CachePut` одна — заполнение кэша. Но работают они немного по-разному. `@CachePut` помечает метод как кэшируемый, и результат сохраняется в кэше. При каждом вызове такого метода с одними и теми же аргументами метод всегда выполняется без проверки, есть ли возвращаемое им значение в кэше или нет. Вот некоторые из атрибутов аннотации `@CachePut`:

- ❑ `value` — название используемого кэша;
- ❑ `key` — ключ для каждого кэшируемого элемента данных;
- ❑ `condition` — выражение языка SpEL, результатом вычисления которого является истина или ложь; в последнем случае результат кэширования не применяется к данному вызову метода;
- ❑ `unless` — тоже выражение языка SpEL; в случае значения «истина» возвращаемое значение не будет помещено в кэш.

Можно также использовать для этой аннотации SpEL и аргумент (-ы) метода. В следующем коде приведено простейшее объявление аннотации `@CachePut`:

```
@CachePut("accountCache ")
public Account save(Account account) {...}
```

В предыдущем коде при вызове метода `save()` происходит сохранение объекта `account`. Затем объект `account` помещается в кэш `accountCache`.

Как упоминалось ранее, кэш заполняется на основе аргумента метода. Фактически это кэш-ключ по умолчанию. В случае аннотации `@Cacheable` аргументом метода `findAccount(Long accountId)` является `accountId`, который и используется в качестве кэш-ключа для данного метода. Но в случае аннотации `@CachePut` единственным параметром метода `save()` является объект типа `Account`. Он и используется в качестве кэш-ключа. Использовать `account` в качестве кэш-ключа не лучший вариант. В данном случае идентификатором свежесохраненного `account` должен быть кэш-ключ, а не сам `account`. Поэтому нам нужно адаптировать к нашему случаю поведение генератора ключей. Посмотрим, как это можно сделать.

Адаптация кэш-ключа к конкретному сценарию

Адаптировать кэш-ключ к конкретному сценарию можно путем использования атрибута `key` аннотаций `@Cacheable` и `@CachePut`. Кэш-ключ получается из SpEL-выражения с использованием таких свойств объекта, как выделенный жирным шрифтом атрибут `key` в следующем фрагменте кода. Рассмотрим примеры:

```
@Cacheable(cacheNames=" accountCache ", key="#accountId")
public Account findAccount(Long accountId)

@Cacheable(cacheNames=" accountCache ", key="#account.accountId")
public Account findAccount(Account account)

@CachePut(value=" accountCache ", key="#account.accountId")
Account save(Account account);
```

В предыдущих фрагментах кода показано, как мы создали кэш-ключ с помощью атрибута `key` аннотации `@Cacheable`.

Рассмотрим еще один атрибут этих аннотаций.

Условное кэширование

Аннотация кэширования фреймворка Spring предоставляет возможность отключать кэширование в отдельных случаях с помощью атрибута `condition` аннотаций `@Cacheable` и `@CachePut`. В нем указывается SpEL-выражение для вычисления условного значения. Если значение условного выражения истинно, то метод кэшируется. Если же ложно, то метод не кэшируется, а выполняется всякий раз без каких-либо операций кэширования, вне зависимости от того, какие значения содержатся в кэше или какие аргументы метода используются. Следующий метод будет кэшироваться только в том случае, если значение переданного в него аргумента больше или равно 2000:

```
@Cacheable(cacheNames="accountCache", condition="#accountId >= 2000")
public Account findAccount(Long accountId);
```

У аннотаций `@Cacheable` и `@CachePut` есть и еще один атрибут — `unless`. В нем также задается SpEL-выражение. Этот атрибут может показаться аналогичным атрибуту `condition`, но между ними есть и различия. В отличие от `condition`, выражения атрибута `unless` вычисляются после вызова метода и предотвращают отправку значения в кэш. Рассмотрим следующий пример, в котором мы хотели бы кэшировать результат только в том случае, если название банка не содержит HDFC:

```
@Cacheable(cacheNames="accountCache", condition="#accountId >=
2000", unless="#result.bankName.contains('HDFC')")
public Account findAccount(Long accountId);
```

Как вы можете видеть из предыдущего фрагмента кода, мы воспользовались обоими атрибутами: `condition` и `unless`. Но в атрибуте `unless` содержалось SpEL-выражение следующего вида: `#result.bankName.contains('HDFC')`. Результат этого

выражения представляет собой расширение SpEL или SpEL метаданные кэша. Вот список метаданных кэширования, доступных в языке SpEL.

Выражение	Описание
#root.methodName	Имя кэшируемого метода
#root.method	Кэшируемый метод, то есть вызываемый метод
#root.target	Экземпляр целевого объекта, содержащий вызываемый метод
#root.targetClass	Класс целевого объекта, содержащий вызываемый метод
#root.caches	Массив кэшей, соответствующих текущему выполняемому методу
#root.args	Массив аргументов, передаваемых в кэшируемый метод
#result	Возвращаемое из кэшируемого метода значение, доступно только в выражениях атрибута unless для аннотации @CachePut



Не следует никогда использовать аннотации @Cacheable и @CachePut одновременно для одного метода в силу различия их поведения. Аннотация @CachePut приводит к принудительному выполнению метода с целью обновления кэшей. А аннотация @Cacheable выполняет кэшируемый метод только в том случае, если возвращаемое значение этого метода отсутствует в кэше.

Вы увидели, как добавлять информацию в кэш с помощью аннотаций @Cacheable и @CachePut фреймворка Spring. Но как удалить информацию оттуда? Абстракция кэша фреймворка Spring предоставляет еще одну аннотацию для удаления закэшированных данных из кэша — @CacheEvict. Посмотрим, как удалить закэшированные данные из кэша с помощью аннотации @CacheEvict.

Аннотация @CacheEvict

Абстракция кэша фреймворка Spring позволяет не только заполнять кэш, но и удалять из кэша закэшированные данные. На определенном этапе работы приложения необходимо удалить устаревшие или неиспользуемые данные из кэша. В этом случае можно воспользоваться аннотацией @CacheEvict, поскольку она ничего не добавляет в кэш, в отличие от аннотации @Cacheable. Аннотация @CacheEvict используется только для вытеснения данных из кэша. Рассмотрим следующий пример:

```
@CacheEvict("accountCache")
void remove(Long accountId);
```

Как вы можете видеть, значение, соответствующее аргументу accountId, удаляется из кэша accountCache при вызове метода remove(). Вот некоторые из атрибутов аннотации @CacheEvict.

- ❑ value — массив названий используемого кэша.
- ❑ key — SpEL-выражение для вычисления используемого кэш-ключа.

- ❑ `condition` — выражение языка SpEL, результатом вычисления которого является истина или ложь; в последнем случае результат кэширования не применяется к данному вызову метода.
- ❑ `allEntries` — если значение этого атрибута истинно, то все записи будут удалены из кэшей.
- ❑ `beforeInvocation` — если значение этого атрибута истинно, то записи будут удалены из кэша до вызова метода, в противном же случае (по умолчанию) записи удаляются после успешного вызова метода.



Аннотацию `@CacheEvict` можно использовать для любого метода, даже возвращающего `void`, поскольку она лишь удаляет значение из кэша. Но в случае аннотаций `@Cacheable` и `@CachePut` использовать возвращающие `void` методы нельзя, поскольку эти аннотации требуют кэширования результата.

Аннотация `@Caching`

Абстракция кэша фреймворка Spring позволяет использовать несколько аннотаций одного типа для кэширования метода посредством применения в приложении Spring аннотации `@Caching`. Аннотация `@Caching` группирует другие аннотации, такие как `@Cacheable`, `@CachePut` и `@CacheEvict`, для одного метода. Например:

```
@Caching(evict = {
    @CacheEvict("accountCache "),
    @CacheEvict(value="account-list", key="#account.accountId") })
public List<Account> findAllAccount(){
    return (List<Account>) accountRepository.findAll();
}
```

Аннотация `@CacheConfig`

Абстракция кэша фреймворка Spring предоставляет аннотацию `@CacheConfig` уровня класса, позволяющую избежать повторения настроек для каждого метода. В некоторых случаях задание настроек кэширования каждого из методов может оказаться чрезвычайно трудоемкой задачей. При этом может оказаться полезной аннотация `@CacheConfig`, применяемая ко всем операциям класса. Например:

```
@CacheConfig("accountCache ")
public class AccountServiceImpl implements AccountService {

    @Cacheable
    public Account findAccount(Long accountId) {
        return (Account) accountRepository.findOne(accountId);
    }
}
```

Из предыдущего фрагмента кода видно, что аннотация `@CacheConfig` используется на уровне класса и дает возможность настройки использования кэша `accountCache` всеми кэшируемыми методами класса.



Поскольку модуль абстракции кэша фреймворка Spring использует прокси, применять аннотации кэширования следует лишь для методов с областью видимости `public`. При использовании этих аннотаций с любым необщедоступным методом никаких исключений сгенерировано не будет, но аннотированные таким образом методы кэширующего поведения не продемонстрируют.

Мы уже сталкивались с тем, что Spring предоставляет возможность использования пространств имен XML для настройки и реализации кэша в приложениях Spring. В следующем разделе мы увидим, как это происходит.

Декларативное кэширование с помощью XML

Гораздо более изящным, чем аннотации, решением, позволяющим разделить код настройки кэширования с бизнес-кодом и сохранить слабое сцепление между аннотациями Spring и вашим исходным кодом, будет конфигурация кэширования на основе XML. Так, для настройки кэша Spring с помощью XML воспользуемся пространством имен `cache` вместе с пространством имен `aop`, поскольку кэширование представляет собой операцию AOP и за кулисами декларативного кэширования используется паттерн «Заместитель».

```
<?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:aop="http://www.springframework.org/schema/aop"
  xmlns:cache="http://www.springframework.org/schema/cache"
  xsi:schemaLocation="http://www.springframework.org/schema/cache
    http://www.springframework.org/schema/cache/spring-cache-4.3.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/aop
    http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
  <!-- Включаем возможность кэширования -->
  <cache:annotation-driven />
  <!-- Объявляем диспетчер кэша -->
  <bean id="cacheManager" class="org.springframework.cache.
    concurrent.ConcurrentMapCacheManager" />
</beans>
```

Из предыдущего XML-файла видно, что мы включили пространства имен `cache` и `aop`. В таблице приведены элементы, с помощью которых пространство имен `cache` задает настройки кэширования.

XML-элемент	Описание кэширования
<code><cache:annotation-driven></code>	Эквивалентно аннотации <code>@EnableCaching</code> в Java-конфигурации, используется для включения кэширующего поведения Spring
<code><cache:advice></code>	Описывает кэширующий совет
<code><cache:caching></code>	Эквивалентно аннотации <code>@Caching</code> и используется для группировки набора правил кэширования в рамках кэширующего совета
<code><cache:cacheable></code>	Эквивалентно аннотации <code>@Cacheable</code> ; делает метод кэшируемым
<code><cache:cache-put></code>	Эквивалентно аннотации <code>@CachePut</code> и используется для заполнения кэша
<code><cache:cache-evict></code>	Эквивалентно аннотации <code>@CacheEvict</code> и используется для вытеснения данных из кэша

Рассмотрим пример, основанный на XML-конфигурации. Создайте файл конфигурации `spring.xml` со следующим содержимым:

```
<?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:aop="http://www.springframework.org/schema/aop"
  xmlns:cache="http://www.springframework.org/schema/cache"
  xsi:schemaLocation="http://www.springframework.org/schema/cache
    http://www.springframework.org/schema/cache/spring-cache-4.3.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/aop
    http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
  <context:component-scan base-
    package="com.packt.patterninspring.chapter9.bankapp.service,
    com.packt.patterninspring.chapter9.bankapp.repository"/>
  <aop:config>
    <aop:advisor advice-ref="cacheAccount" pointcut="execution(*
    com.packt.patterninspring.chapter9.bankapp.service.*(..)"/>
  </aop:config>
  <cache:advice id="cacheAccount">
    <cache:caching>
      <cache:cacheable cache="accountCache" method="findOne" />
      <cache:cache-put cache="accountCache" method="save"
        key="#result.id" />
      <cache:cache-evict cache="accountCache" method="remove" />
    </cache:caching>
  </cache:advice>

  <!-- Объявляем диспетчер кэша -->
```

```
<bean id="cacheManager" class="org.springframework.cache.concurrent.
ConcurrentMapCacheManager" />
</beans>
```

В вышеприведенном файле XML-конфигурации код представляет собой настройки кэширования Spring. Первое, что мы видим в этих настройках кэширования, — объявление `<aop:config>`, затем `<aop:advisor>` со ссылками на совет с идентификатором `cacheAccount` и выражение среза для выбора совета. Совет объявлен с помощью элемента `<cache:advice>`. Этот элемент может включать множество элементов `<cache:caching>`. Но в нашем примере только один элемент `<cache:caching>`, содержащий элементы `<cache:cacheable>`, `<cache:cache-put>`, и один элемент `<cache:cache-evict>`; каждый из них объявляет какой-либо метод из среза кэшируемым.

Взглянем теперь на класс `Service` нашего приложения с аннотациями кэширования:

```
package com.packt.patterninspring.chapter9.bankapp.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

import com.packt.patterninspring.chapter9.bankapp.model.Account;
import com.packt.patterninspring.chapter9.
bankapp.repository.AccountRepository;

@Service
public class AccountServiceImpl implements AccountService{
    @Autowired
    AccountRepository accountRepository;

    @Override
    @Cacheable("accountCache")
    public Account findOne(Long id) {
        System.out.println("findOne called");
        return accountRepository.findAccountById(id);
    }

    @Override
    @CachePut("accountCache")
    public Long save(Account account) {
        return accountRepository.save(account);
    }

    @Override
    @CacheEvict("accountCache")
    public void remove(Long id) {
        accountRepository.findAccountById(id);
    }
}
```

В вышеприведенном файле с описанием мы воспользовались аннотациями кэширования для создания кэша. Посмотрим теперь, как настроить хранилище кэша для нашего приложения.

Настройка хранилища кэша

Абстракция кэша фреймворка Spring обеспечивает интеграцию множества разных хранилищ. Для каждого хранилища данных в памяти предоставляется диспетчер кэша (интерфейс `CacheManager`). Необходимо только настроить диспетчер кэша для вашего приложения. А далее он возьмет на себя контроль и управление кэшем. Посмотрим, как настроить диспетчер кэша в приложении.

Настройка диспетчера кэша. Вы должны указать в приложении диспетчер кэша для хранилища, а также передать поставщик функциональности кэширования диспетчеру кэша или же написать свой собственный диспетчер кэша. В пакете `org.springframework.cache` фреймворка Spring имеется несколько диспетчеров кэша, например `ConcurrentMapCacheManager`, создающий по объекту `ConcurrentHashMap` для каждого элемента хранилища:

```
@Bean
public CacheManager cacheManager() {
    CacheManager cacheManager = new ConcurrentMapCacheManager();
    return cacheManager;
}
```

Объекты `SimpleCacheManager`, `ConcurrentMapCacheManager` и др. — диспетчеры кэша абстракции кэша фреймворка Spring. Но Spring обеспечивает интеграцию и со сторонними диспетчерами кэша, как вы увидите в следующем разделе.

Сторонние диспетчеры кэша

Объект `SimpleCacheManager` фреймворка Spring вполне подходит для тестирования, но в нем отсутствуют возможности контроля кэша (переполнения, вытеснения). Поэтому приходится использовать сторонние альтернативные варианты, например:

- ❑ EhCache компании Terracotta;
- ❑ Guava и Caffeine компании Google;
- ❑ Gemfire компании Pivotal.

Посмотрим поближе на одну из сторонних реализаций диспетчеров кэша.

EhCache

Библиотека EhCache — один из наиболее популярных поставщиков функциональности кэширования. Spring предоставляет возможность интеграции с EhCache посредством настройки в приложении объекта `EhCacheCacheManager`. Рассмотрим следующую Java-конфигурацию:


```

@Bean
public CacheManager cacheManager(CacheManager ehCache) {
    EhCacheCacheManager cmgr = new EhCacheCacheManager();
    cmgr.setCacheManager(ehCache);
    return cmgr;
}

@Bean
public EhCacheManagerFactoryBean ehCacheManagerFactoryBean() {
    EhCacheManagerFactoryBean eh = new EhCacheManagerFactoryBean();
    eh.setConfigLocation(new
        ClassPathResource("resources/ehcache.xml"));
    return eh;
}

```

В предыдущем коде метод `cacheManager()` компонента создает объект класса `EhCacheCacheManager` и задает в качестве целевого диспетчера кэша объект `ehCache` типа `CacheManager`. Здесь объект типа `CacheManager` библиотеки `EhCache` внедряется в класс `EhCacheCacheManager` фреймворка `Spring`. Второй метод компонента, `ehCacheManagerFactoryBean()`, создает и возвращает объект класса `EhCacheManagerFactoryBean`. А поскольку это компонент-фабрика, то он вернет экземпляр класса `CacheManager`. Конфигурация библиотеки `EhCache` содержится в XML-файле `ehcache.xml`. Вот ее код:

```

<ehcache>
  <cache name="accountCache" maxBytesLocalHeap="50m"
    timeToLiveSeconds="100">
  </cache>
</ehcache>

```

Содержимое файла `ehcache.xml` в разных приложениях будет различным, но в нем должен быть объявлен по крайней мере минимальный кэш. Например, следующая конфигурация `EhCache` объявляет кэш `accountCache` с хранилищем в куче с максимальным размером 50 Мбайт и временем жизни 100 секунд.

XML-конфигурация

Создадим конфигурацию в стиле XML для `EhCache`, в которой задаются настройки `EhCacheCacheManager` (см. следующий код):

```

<bean id="cacheManager"
class="org.springframework.cache.ehcache.EhCacheCacheManager"
p:cache-manager-ref="ehcache"/>

<!-- Настройки библиотеки EhCache -->
<bean id="ehcache"
class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"
p:config-location="resources/ehcache.xml"/>

```

Аналогично в случае XML-конфигурации необходимо настроить диспетчер кэша для `EhCache`, настроить класс `EhCacheManagerFactoryBean` и задать расположение файла `ehcache.xml` с конфигурацией `EhCache`, показанного в предыдущем разделе.

Существует множество других сторонних хранилищ кэша, интеграцию которых поддерживает фреймворк Spring. В этой главе мы обсудили только диспетчер EhCache.

В следующем разделе мы обсудим предоставляемые Spring возможности создания своих собственных аннотаций для кэширования.

Создание пользовательских аннотаций кэширования

Абстракция кэша фреймворка Spring предоставляет возможность создания пользовательских аннотаций кэширования для приложений. Аннотации `@Cacheable` и `@CacheEvict` фреймворка Spring используется при создании пользовательских аннотаций кэширования в качестве метааннотаций. Рассмотрим код создания пользовательской аннотации для приложения:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Cacheable(value="accountCache", key="#account.id")
public @interface SlowService {
}
```

В этом фрагменте кода мы описали пользовательскую аннотацию с названием `SlowService`, снабженную аннотацией `@Cacheable` фреймворка Spring. Если мы используем аннотацию `@Cacheable` в приложении, то должны настроить ее так, как показано в следующем коде:

```
@Cacheable(value="accountCache", key="#account.id")
public Account findAccount(Long accountId)
```

Заменим вышеприведенную конфигурацию нашей пользовательской аннотацией с помощью следующего кода:

```
@SlowService
public Account findAccount(Long accountId)
```

Как вы можете видеть, достаточно одной аннотации `@SlowService`, чтобы метод в приложении стал кэшируемым.

Перейдем теперь к следующему разделу, где обсудим рекомендуемые практики, связанные с организацией кэширования в приложении.

Лучшие рекомендуемые практики для веб-приложений

Должное использование кэширования в корпоративном веб-приложении обеспечивает очень быструю визуализацию, минимизирует обращения к базе данных и снижает потребление ресурсов сервера — памяти, сети и т. д. Кэширование — очень действенный метод повышения производительности приложения за счет

хранения часто используемых данных в кэш-памяти. Перечислим рекомендуемые практики, которых имеет смысл придерживаться при проектировании и разработке веб-приложений.

- ❑ В веб-приложениях Spring следует использовать аннотации кэширования, например `@Cacheable`, `@CachePut` и `@CacheEvict`, для конкретных классов, а не интерфейсов приложения. Однако можно снабжать аннотациями и методы интерфейсов с помощью основанных на интерфейсах прокси. Помните, что Java-аннотации не наследуются от интерфейсов, так что при использовании прокси на основе классов (с помощью атрибута `proxy-target-class="true"`) аннотации кэширования Spring не распознаются прокси.
- ❑ Чтобы получить хоть какую-то пользу от кэширования, никогда не следует вызывать аннотированный `@Cacheable`, `@CachePut` или `@CacheEvict` метод непосредственно из другого метода того же класса. Дело в том, что при прямом вызове кэшируемого метода применения прокси AOP Spring никогда не происходит.
- ❑ В корпоративных приложениях никогда не следует использовать в качестве кэша ассоциативные массивы Java и другие коллекции пар «ключ — значение». Коллекция пар «ключ — значение» не может служить кэшем. Иногда разработчики используют ассоциативные массивы языка Java в качестве пользовательского кэширующего решения, но они на самом деле не предназначены для кэширования, поскольку кэш должен обеспечивать не только хранение пар «ключ — значение», а еще и, например:
 - стратегии вытеснения;
 - возможность задать максимальный размер кэша;
 - постоянное хранилище;
 - ключи для слабых ссылок;
 - статистику.
- ❑ Фреймворк Spring позволяет использовать декларативный подход — идеальный для реализации и настройки в приложении кэширующего решения. Поэтому всегда используйте уровень абстракции кэша — он обеспечивает гибкость. Как мы знаем, благодаря аннотации `@Cacheable` становится возможным отделить код бизнес-логики от сквозной функциональности кэширования.
- ❑ Осторожнее используйте кэш в своих приложениях. Всегда используйте кэш только там, где он действительно нужен, например в веб-сервисах или при дорогостоящих обращениях к базе данных, поскольку всякий API кэширования приносит дополнительные накладные расходы.
- ❑ При реализации кэша в приложении следует обеспечить синхронизацию данных в кэше с хранилищем данных. Для должной реализации стратегии кэширования при хорошей производительности можно воспользоваться такими распределенными диспетчерами кэша, как `Memcached`.
- ❑ Кэш следует использовать лишь в качестве запасного варианта, в том случае, если извлечение данных из базы данных представляет большие сложности вследствие медленного выполнения запросов. Дело в том, что при всяком использовании

кэширования в приложении сначала проверяется наличие значения в кэше и только при его отсутствии выполняется сам метод. Подобное поведение приводит к дополнительным накладным расходам и в некоторых случаях нежелательно.

- ❑ В этой главе мы уже видели, как кэширование повышает производительность приложения. Кэширование в основном выполняется на уровне сервисов приложения. Методы в приложении возвращают данные, которые можно закэшировать, если код приложения вызывает эти методы снова и снова с одними и теми же параметрами. Кэширование — отличный способ избежать повторного выполнения метода приложения с теми же параметрами. Возвращаемое значение метода для конкретного параметра сохраняется в кэше при первом вызове. При последующих же вызовах того же метода с теми же параметрами значение извлекается из кэша. Кэширование повышает производительность приложения, исключая выполнение некоторых операций, требующих расхода времени и ресурсов для получения одних и тех же ответов, например запросов к базе данных.

Резюме

Фреймворк Spring предоставляет интерфейс `CacheManager` для управления кэшем в приложениях Spring. В этой главе вы увидели, как описать диспетчер кэша для одной из технологий кэширования. Spring предоставляет несколько аннотаций для кэширования, в частности `@Cacheable`, `@CachePut` и `@CacheEvict`, которыми вы можете воспользоваться в своих приложениях Spring. Можно также задать настройки кэширования в приложении Spring с помощью XML-конфигурации. Фреймворк Spring предоставляет для этой цели специальное пространство имен. Вместо соответствующих аннотаций применяются элементы `<cache:cacheable>`, `<cache:cache-put>` и `<cache:cache-evict>`.

Spring дает возможность управления кэшированием в приложении с помощью аспектно-ориентированного программирования. Для фреймворка Spring кэширование представляет собой сквозную функциональность. Это значит, что кэширование — аспект в приложениях Spring. Spring реализует кэширование посредством совета типа «*везде*» модуля AOP Spring.

В следующей главе рассказывается, как использовать Spring на веб-уровне и с паттерном MVC.

10 Реализация паттерна MVC в веб-приложениях с помощью фреймворка Spring

В нескольких предыдущих главах книги в основе всех примеров лежали автономные приложения Spring. Вы наблюдали важные возможности, предоставляемые Spring, такие как паттерн внедрения зависимостей, управление жизненным циклом компонентов, AOP, управление кэшем, а также работу Spring в прикладной части с помощью модулей JDBC и ORM. В этой главе вы увидите, как Spring работает в веб-среде, решая некоторые часто встречающиеся в любом веб-приложении задачи: организацию технологического процесса, проверку корректности данных и управление состоянием.

В фреймворке Spring наряду с другими модулями есть и свой веб-фреймворк, известный под названием Spring Web MVC. В его основе лежит паттерн «*Модель — Представление — Контроллер*» (Model — View — Controller, MVC). Модуль Web MVC фреймворка Spring поддерживает уровень визуализации данных (presentation tier) и помогает в создании гибкого и слабо сцепленного веб-приложения. Модуль MVC фреймворка Spring решает задачу тестирования веб-компонентов в корпоративном приложении. С его помощью можно писать тестовые сценарии без использования в приложении объектов запроса/ответа. Мы поговорим о нем подробнее в данной главе.

В этой главе мы не только обсудим внутреннее устройство модуля MVC Spring, но и поговорим о различных уровнях веб-приложения. Мы посмотрим на реализацию паттерна MVC, в том числе обсудим, что это такое и почему следует его использовать. Мы разберем в этой главе следующие темы, касающиеся веб-фреймворка MVC Spring.

- ❑ Реализация паттерна MVC в веб-приложении.
- ❑ Реализация паттернов контроллеров.
- ❑ Настройка класса `DispatcherServlet` как реализации паттерна «Единая точка входа».
- ❑ Активизация возможностей модуля MVC Spring и использование прокси.
- ❑ Обработка форм веб-страницы.

- ❑ Реализация представления из паттерна MVC.
- ❑ Создание JSP-представлений в веб-приложениях.
- ❑ Паттерн «Вспомогательный компонент представления».
- ❑ Паттерн «Составное представление» и компонент ViewResolver фреймворка Apache Tiles.

Реализация паттерна MVC в веб-приложении

«Модель — Представление — Контроллер» (MVC) представляет собой паттерн проектирования J2EE. Впервые его использовал в своем проекте Трюгве Реенскауг для разделения различных компонентов традиционного, не веб-, приложения. Основной подход данного паттерна заключается в продвижении в среде разработчиков ПО принципа разделения ответственности. Паттерн MVC разделяет систему на три вида компонентов, причем у каждого компонента системы есть свои конкретные обязанности. Вот эти три вида компонентов.

- ❑ *Модель.* Модель в паттерне MVC отвечает за данные для представления, с целью дальнейшей их визуализации в одном из шаблонов представления. Если вкратце, можно сказать, что модель представляет собой объект данных, подобный объекту `SavingAccount` в нашем примере системы для банка, содержащий список счетов отделения какого-либо банка.
- ❑ *Представление.* Представление в паттерне MVC отвечает за визуализацию модели в веб-приложении в виде веб-страницы. Оно представляет данные модели в удобочитаемом для пользователя формате. Для этой цели существует несколько технологий, например JSP, страницы JSF, PDF, XML и др.
- ❑ *Контроллер.* Это тот компонент паттерна MVC, который фактически производит все действия. Код контроллера управляет взаимодействием между представлением и моделью. Такие взаимодействия, как отправка формы или щелчок на ссылке, являются в корпоративном приложении частью контроллера. Кроме того, контроллер отвечает за создание и обновление модели, а также перенаправление модели представлению для визуализации.

Рассмотрим следующую схему, чтобы лучше разобраться в паттерне MVC (рис. 10.1).

В приложении есть три компонента, у каждого из которых — свои обязанности. Как уже упоминалось, паттерн MVC нацелен на разделение ответственности. В программной системе разделение ответственности играет очень важную роль, обеспечивая гибкость и легкость тестирования компонентов, а также чистую структуру кода. В паттерне MVC пользователь взаимодействует с компонентом «контроллер» посредством компонента «представление», а компонент «контроллер» запускает фактические действия по подготовке компонента «модель». Компонент «модель» распространяет изменения на «представление», и, наконец, компонент «представление» визуализирует модель на экране пользователя. Такова общая идея

реализации паттерна MVC. Паттерн MVC отлично подходит для большинства приложений, особенно традиционных, не веб-. Паттерн MVC известен также как архитектура «Модель 1».

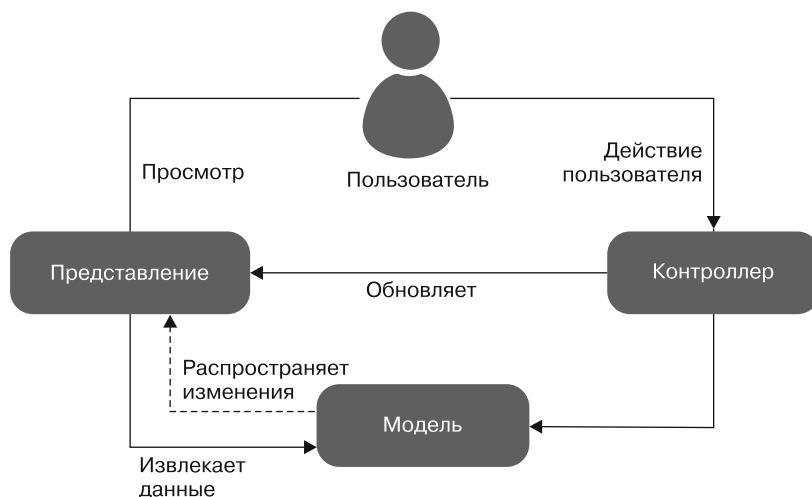


Рис. 10.1. Принцип работы паттерна MVC

Впрочем, корпоративные веб-приложения несколько отличаются от традиционных приложений, ведь в силу того, что протокол HTTP не сохраняет состояние, сохранять информацию о модели в течение всего жизненного цикла запроса довольно трудно. В следующем разделе мы рассмотрим усовершенствованную версию паттерна MVC и ее применение во фреймворке Spring для создания корпоративного веб-приложения.

Архитектура «Модель 2» паттерна MVC в Spring

Архитектура «Модель 1» для веб-приложения представляется не очень простой. В «Модели 1» есть также децентрализованное управление навигацией по сайту, ведь в этой архитектуре у каждого пользователя есть отдельный контроллер и отдельная логика определения следующей страницы. Основные технологии для разработки веб-приложений с архитектурой «Модель 1» — сервлеты и JSP.

Для веб-приложений паттерн MVC реализуется в виде архитектуры «Модель 2». Эта архитектура обеспечивает централизованную логику управления навигацией, что позволяет легко тестировать и сопровождать веб-приложение, и разделение обязанностей в ней для веб-приложений организовано лучше, чем в архитектуре «Модель 1». Усовершенствованный паттерн MVC на основе архитектуры «Модель 2» отличается от паттерна MVC на основе архитектуры «Модель 1» наличием контроллера — единой точки входа, распределяющего все входящие

запросы по другим контроллерам. Эти контроллеры обрабатывают входящие запросы, возвращая модель, и выбирают представление. Рассмотрим следующую схему, чтобы лучше понять принципы функционирования паттерна MVC архитектуры «Модель 2» (рис. 10.2).

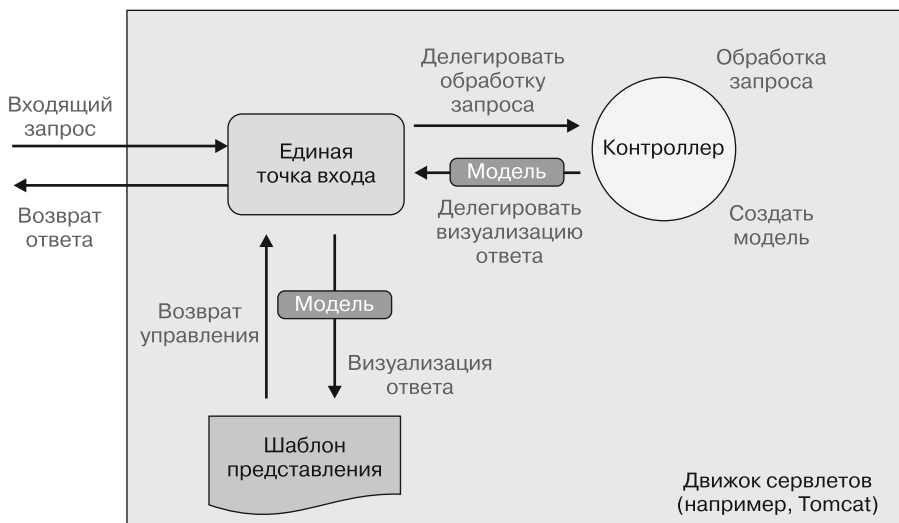


Рис. 10.2. Функционирование паттерна MVC архитектуры «Модель 2»

Как можно видеть на схеме, в паттерне MVC появляется новый компонент, а именно единая точка входа (front controller). Он реализуется в виде сервлета типа `javax.servlet.Servlet`, например `ActionServlet` во фреймворке Apache Struts, `FacesServlet` в JSF или `DispatcherServlet` в MVC Spring. Он получает входящие запросы, передавая их одному из контроллеров приложения, который, в свою очередь, создает и обновляет модель, передавая ее обратно единой точке входа для визуализации. Наконец, единая точка входа определяет конкретное представление и визуализирует данные модели.

Паттерн проектирования «Единая точка входа»

Паттерн проектирования «Единая точка входа» — паттерн J2EE, решающий следующие проблемы проектирования приложений.

- ❑ В основанных на архитектуре «Модель 1» веб-приложениях для обработки большого количества запросов необходимо слишком много контроллеров. Их сопровождение и переиспользование представляет собой непростую задачу.
- ❑ У каждого запроса своя точка входа в приложение. Желательно, чтобы точка входа была одна для всех запросов.

- ❑ JSP и сервлеты — основные компоненты паттерна MVC с архитектурой «Модель 1», так что эти компоненты отвечают как за фактические действия, так и за визуализацию, нарушая тем самым принцип *единственной обязанности*.

Единая точка входа предлагает решение вышеописанных проектных проблем веб-приложения. В веб-приложении она играет роль основного компонента, маршрутизирующего все запросы. Это значит, что в отдельный контроллер (единую точку входа) попадает слишком много запросов, так что их обработка затем делегируется конкретным контроллерам. Единая точка входа обеспечивает централизованное управление и расширяет возможности переиспользования и удобство управления, поскольку обычно в веб-контейнере регистрируется только ресурс. Эта точка входа не только распределяет/обрабатывает излишки запросов, но также отвечает за:

- ❑ инициализацию обслуживающего запросы фреймворка;
- ❑ загрузку ассоциативного массива всех URL и компонентов, отвечающих за обработку запроса;
- ❑ подготавливает ассоциативный массив для представлений.

Рассмотрим следующую схему единой точки входа (рис. 10.3).



Рис. 10.3. Единая точка входа

Как можно видеть на предыдущей схеме, все запросы приложения попадают в единую точку входа, которая делегирует эти запросы предварительно настроенным контроллерам приложения.

Фреймворк Spring содержит модуль, основанный на паттерне MVC, — реализацию архитектуры «Модель 2». Модуль MVC фреймворка Spring предоставляет готовую реализацию паттерна «Единая точка входа» в виде класса `org.springframework.web.servlet.DispatcherServlet`. Этот простой класс сервлета составляет основу модуля MVC фреймворка Spring. И он интегрирован с контейнером IoC фреймворка Spring, что позволяет воспользоваться возможностями паттерна внедрения зависимостей Spring. Веб-фреймворк Spring использует сам Spring для настройки, а все контроллеры представляют собой компоненты Spring, причем с возможностями тестирования.

Подробнее изучим в этой главе внутреннее устройство модуля MVC Spring и взглянем поближе на то, как класс `org.springframework.web.servlet.DispatcherServlet` фреймворка MVC Spring обрабатывает все входящие запросы к веб-приложению.

Технологический процесс жизненного цикла запроса

Случалось ли вам играть в *настольную головоломку-лабиринт со стальными шарикоподшипниками*? Это совершенно безумная игра (рис. 10.4). Ее цель — загнать все стальные шарикоподшипники в центр деревянной доски-лабиринта по взаимосвязанным изогнутым дорожкам с прорезями, ведущими к следующей дорожке, ближе к центру. Необходимо провести все шарики в центр деревянного лабиринта через эти прорези между изогнутыми дорожками. Если нам удалось довести один из шариков до центра, нужно быть осторожнее, чтобы он не выскользнул оттуда, пока мы пытаемся провести следующий.



Рис. 10.4. Деревянный лабиринт

На первый взгляд, модуль MVC Spring похож на эту настольную игру. Вместо перемещения стальных шарикоподшипников по изогнутым дорожкам и прорезям модуль MVC Spring перемещает запросы веб-приложения через различные компоненты, например единую точку входа, то есть сервлет-диспетчер, отображения обработчиков, контроллеры и арбитры представлений.

Изучим технологический процесс обработки запросов в модуле MVC фреймворка Spring для веб-приложений. Технологический процесс обработки запросов класса `DispatcherServlet` веб-фреймворка MVC Spring показан на рис. 10.5.

Как вы уже знаете, единая точка входа играет очень важную роль в паттерне MVC архитектуры «Модель 2», поскольку отвечает за все входящие запросы к веб-приложению и подготовку ответа браузеру. Во фреймворке MVC Spring объект `org.springframework.web.servlet.DispatcherServlet` играет роль единой точки входа паттерна MVC архитектуры «Модель 2». Как вы можете видеть на рис. 10.5, этот объект `DispatcherServlet` для выполнения своих задач использует множество других компонентов. Рассмотрим поэтапно обработку запроса во фреймворке MVC Spring.

1. Пользователь нажимает на ссылку в браузере или отправляет веб-форму в приложении. Запрос, включающий или какую-то дополнительную, или просто основную информацию, покидает браузер и оказывается в объекте `DispatcherServlet` фреймворка Spring, представляющем собой просто класс

Жизненный цикл обработки запроса

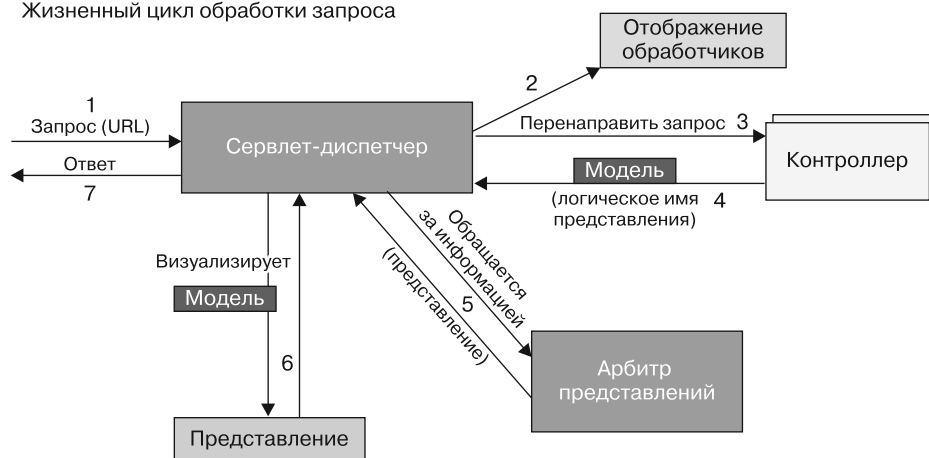


Рис. 10.5. Технологический процесс обработки запросов класса `DispatcherServlet` фреймворка MVC

сервлета, как и другие веб-приложения на основе Java. Он представляет собой единую точку входа фреймворка MVC Spring, через которую пропускаются все входящие запросы. Благодаря использованию этой единой точки входа фреймворк MVC Spring централизует управление всем потоком запросов.

- После поступления запроса в объект `DispatcherServlet` фреймворка Spring последний передает этот запрос в контроллер MVC Spring, то есть контроллер приложения. Хотя в веб-приложениях Spring может быть несколько контроллеров, но каждый запрос должен попасть к одному из них. Для этого объект `DispatcherServlet` пользуется отображениями обработчиков, заданными в настройках веб-приложения. Отображение обработчиков определяет конкретный контроллер по URL и параметрам запроса.
- После выбора нужного контроллера приложения объектом `DispatcherServlet` с помощью настроек отображения обработчиков `DispatcherServlet` направляет запрос этому контроллеру. Именно этот контроллер на самом деле отвечает за обработку информации в соответствии с запросом пользователя и его параметрами.
- Контроллер фреймворка MVC Spring выполняет бизнес-логику, используя бизнес-сервисы приложения, и создает модель, в которую обертывается отправляемая обратно пользователю и отображаемая в браузере информация. Эта модель несет соответствующую запросу пользователя информацию, но она не форматирована, так что можно использовать любую технологию шаблонов представлений для визуализации в браузере содержащейся в ней информации. Именно поэтому контроллер MVC Spring возвращает, помимо модели, название логического представления. А делает он это потому, что контроллер MVC Spring не привязан ни к какой конкретной технологии представления — JSP, JSF, Thymeleaf и т. д.

5. И снова объект `DispatcherServlet` модуля MVC фреймворка Spring пользуется помощью арбитра представлений, который настроен в веб-приложении таким образом, чтобы разрешать представления. В соответствии с настройками объекта `ViewResolver` он выдает реальное имя представления вместо логического имени представления. Теперь у объекта `DispatcherServlet` есть необходимое для визуализации информации модели представление.
6. Объект `DispatcherServlet` модуля MVC фреймворка Spring визуализирует модель в представление и генерирует информацию модели в удобочитаемом для пользователя формате.
7. Наконец, на основе этой информации `DispatcherServlet` создает ответ и возвращает его браузеру пользователя.

Как видите, процесс обработки запроса приложения состоит из нескольких этапов и в него вовлечено несколько компонентов. Большинство из этих компонентов относятся к фреймворку MVC Spring, и у каждого из них есть свои конкретные обязанности.

Пока вы узнали только, что `DispatcherServlet` — ключевой компонент в обработке запросов с помощью MVC Spring, самое сердце веб-модуля MVC Spring. Именно единая точка входа координирует все действия по обработке запросов аналогично объекту `ActionServlet` фреймворка Struts и `FacesServlet` в JSF. Она делегирует обработку запросов инфраструктурным веб-компонентам и обращается к пользовательским веб-компонентам. Она чрезвычайно гибка в настройке и полностью адаптируется к конкретным задачам. Гибкость ей обеспечивает тот факт, что все используемые этим сервлетом компоненты являются интерфейсами для инфраструктурных компонентов. В следующей таблице перечислены некоторые из подобных интерфейсов, предоставляемых фреймворком MVC Spring.

Компонент модуля MVC Spring	Роль в обработке запросов
<code>org.springframework.web.multipart.MultipartResolver</code>	Обрабатывает составные запросы, например загрузку файла
<code>org.springframework.web.servlet.LocaleResolver</code>	Осуществляет локальное разрешение и изменение представлений
<code>org.springframework.web.servlet.ThemeResolver</code>	Осуществляет разрешение и изменение тем оформления
<code>org.springframework.web.servlet.HandlerMapping</code>	Отображает все входящие запросы на объекты-обработчики
<code>org.springframework.web.servlet.HandlerAdapter</code>	Этот интерфейс основан на паттерне «Адаптер» (Adapter), должен реализовываться для каждого типа объекта-обработчика; используется для вызова обработчика
<code>org.springframework.web.servlet.HandlerExceptionResolver</code>	Обрабатывает исключения, сгенерированные во время выполнения обработчика
<code>org.springframework.web.servlet.ViewResolver</code>	Транслирует логическое представление в фактическую реализацию представления

Перечисленные в предыдущей таблице компоненты модуля MVC фреймворка Spring относятся к жизненному циклу обработки запроса в веб-приложениях. В следующем разделе мы узнаем о настройке основного компонента модуля MVC Spring — `DispatcherServlet`. Мы также подробнее рассмотрим различные способы его реализации и настройки на основе Java- и XML-конфигураций.

Настраиваем `DispatcherServlet` в качестве единой точки входа

Все сервлеты в основанных на Java веб-приложениях описываются в файле `web.xml`. Он загружается в веб-контейнер в момент инициализации приложения и устанавливает соответствие каждого из сервлетов конкретному шаблону URL. Аналогично интерфейс `org.springframework.web.servlet.DispatcherServlet` — основной элемент модуля MVC Spring, необходимо задать его настройки в том же файле `web.xml`, и загружается он также при инициализации веб-приложения. При инициализации веб-приложения `DispatcherServlet` вызывается для создания веб-контекста `org.springframework.web.context.WebApplicationContext` с помощью загрузки настроек компонентов из конфигураций в стиле Java, XML или основанных на аннотациях. Сервлет пытается извлечь все необходимые компоненты из этого контекста веб-приложения. Его обязанность — маршрутизировать запрос по остальным компонентам.



Как обсуждалось в предыдущих главах нашей книги, `WebApplicationContext` представляет собой веб-версию класса `ApplicationContext`. Он обладает некоторыми дополнительными, по сравнению с `ApplicationContext`, возможностями, необходимыми для веб-приложений, например: сервлет-ориентированными областями видимости уровня запроса (`request`), сеанса (`session`) и т. д. Объект `WebApplicationContext` привязан к объекту `ServletContext`, можно также обратиться к нему с помощью статического метода класса `RequestContextUtils` (см. следующий фрагмент кода):

```
ApplicationContext webApplicationContext = RequestContextUtils.  
findWebApplicationContext(request);
```

Описание с помощью XML-конфигурации

Как вы знаете, `web.xml` — корневой файл любого веб-приложения, размещаемый в каталоге `WEB-INF`. В нем находится спецификация сервлета, а также все загружаемые настройки сервлетов. Вот такой код необходим в конфигурации объекта `DispatcherServlet` в веб-приложении:

```
<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"  
  metadata-complete="true">  
  <servlet>
```

```

<servlet-name>bankapp</servlet-name>
<servlet-
  class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>bankapp</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
</web-app>

```

Предыдущий фрагмент представляет собой минимальный возможный код настройки объекта `DispatcherServlet` в веб-приложении Spring с помощью конфигурации в стиле XML.



Ничего особенного в файле `web.xml` нет; обычно в нем описаны настройки только одного сервлета, очень похожего на традиционные веб-приложения Java. Но `DispatcherServlet` загружает файл с конфигурацией компонентов Spring приложения. По умолчанию он загружает файл с именем `[servletname]-servlet.xml` из каталога `WEB-INF`. В нашем случае это должен быть файл `bankapp-servlet.xml` из каталога `WEB-INF`.

Описание с помощью Java-конфигурации

В этой главе вместо XML-конфигурации мы воспользуемся возможностями Java для задания настроек объекта `DispatcherServlet` в контейнере сервлетов нашего приложения. Servlet 3.0 и более поздние версии поддерживают начальную загрузку на основе Java, так что можно не использовать файл `web.xml`. Вместо него мы можем создать Java-класс, реализующий интерфейс `javax.servlet.ServletContainerInitializer`. Для того чтобы обеспечить загрузку и инициализацию конфигурации Spring в любом контейнере Servlet 3, модуль MVC Spring предоставляет интерфейс `WebApplicationInitializer`. Но фреймворк MVC Spring упрощает эту задачу еще больше благодаря реализации интерфейса `WebApplicationInitializer` в виде абстрактного класса. С помощью этого абстрактного класса можно просто выполнить привязку сервлета к URL и указать классы конфигурации — корневой и для MVC. Я предпочитаю использовать именно такой способ задания настроек в своих приложениях. Вот код класса конфигурации:

```

package com.packt.patterninspring.chapter10.bankapp.web;

import org.springframework.web.servlet.support.
    AbstractAnnotationConfigDispatcherServletInitializer;

import com.packt.patterninspring.chapter10.bankapp.config.AppConfig;
import com.packt.patterninspring.chapter10.bankapp.web.mvc.SpringMvcConfig;

public class SpringApplicationInitilizer extends
    AbstractAnnotationConfigDispatcherServletInitializer {

```

```
// Указываем фреймворку Spring, что использовать в качестве корневого
// (root) контекста: ApplicationContext – «корневая» конфигурация
@Override
protected Class<?>[] getRootConfigClasses() {
    return new Class <?>[]{AppConfig.class};
}
// Указываем фреймворку Spring, что использовать в качестве
// контекста DispatcherServlet: WebApplicationContext – MVC
// configuration
@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class <?>[]{SpringMvcConfig.class};
}
// Отображение DispatcherServlet, данный метод отвечает за шаблон URL
// аналогично элементу <url-pattern>/</url-pattern> в файле web.xml
@Override
protected String[] getServletMappings() {
    return new String[]{"/*"};
}
}
```

Как видно из этого кода, класс `SpringApplicationInitilizer` расширяет класс `AbstractAnnotationConfigDispatcherServletInitializer`. Он требует от разработчика только действительно необходимую информацию и выполняет все относящиеся к `DispatcherServlet` настройки с помощью интерфейсов контейнера сервлетов. Взглянем на следующую схему, чтобы разобраться в том, как класс `AbstractAnnotationConfigDispatcherServletInitializer` и его реализация задают настройки `DispatcherServlet` в приложении (рис. 10.6).

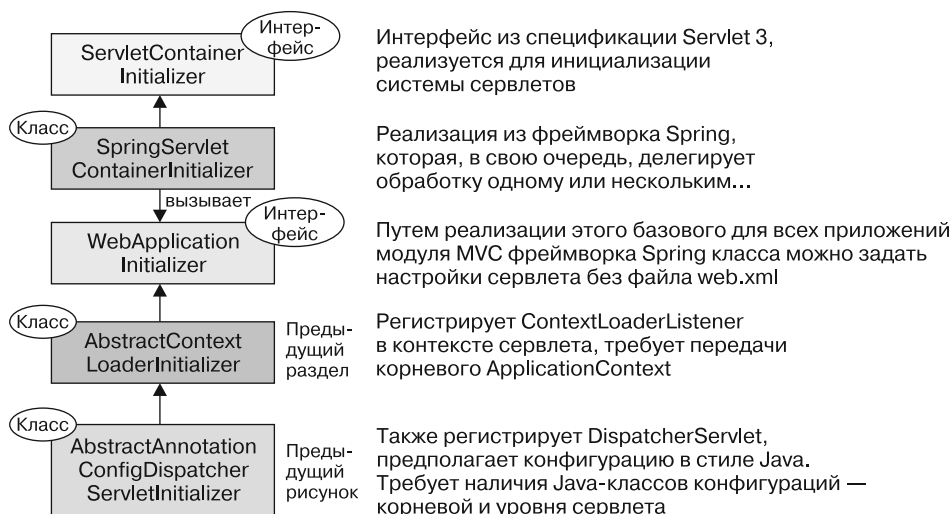


Рис. 10.6. Схема того, как класс `AbstractAnnotationConfigDispatcherServletInitializer` и его реализация задают настройки `DispatcherServlet` в приложении

Вы видели, что класс `SpringApplicationInitializer` переопределяет три метода класса `AbstractAnnotationConfigDispatcherServletInitializer`, а именно: `getServletMappings()`, `getServletConfigClasses()` и `getRootConfigClasses()`. Метод `getServletMappings()` описывает привязку сервлета к URL — в нашем приложении он привязан к URL `"/"`. Метод `getServletConfigClasses()` просит `DispatcherServlet` загрузить контекст приложения вместе с компонентами, описанными в классе конфигурации `SpringMvcConfig`. В этом файле конфигурации содержатся определения таких веб-компонентов, как контроллеры, арбитры представлений и отображения обработчиков. У веб-приложений Spring есть еще один контекст приложения, создаваемый классом `ContextLoaderListener`. Поэтому третий метод, `getRootConfigClasses()`, выполняет загрузку остальных компонентов, например сервисов, репозиториях, источников данных и прочих компонентов приложения, описанных в классе конфигурации `AppConfig` и используемых в промежуточном уровне и уровне данных приложения.



Фреймворк Spring предоставляет класс прослушателя — `ContextLoaderListener`, отвечающий за начальную загрузку контекста прикладной части приложения.

Взглянем на следующую схему, чтобы лучше понять архитектуру веб-приложений Spring после запуска контейнера сервлетов (рис. 10.7).

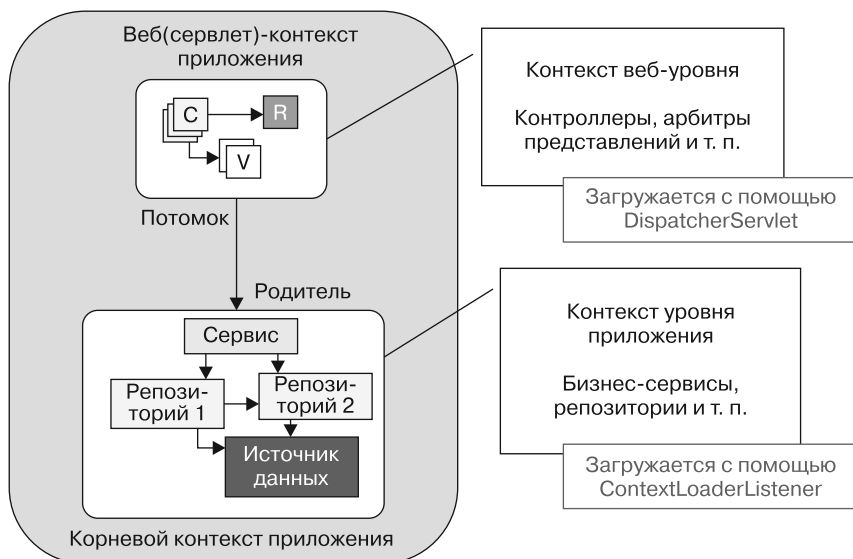


Рис. 10.7. Архитектура веб-приложений Spring

Возвращаемые методом `getServletConfigClasses()` классы конфигурации с описаниями веб-компонентов загружаются с помощью `DispatcherServlet`, а клас-

сы конфигурации с описаниями остальных компонентов, возвращаемые методом `getRootConfigClasses()`, загружаются с помощью `ContextLoaderListener`.



Веб-конфигурация на основе Java работает только при развертывании на сервере, поддерживающем Servlet 3.0, например Apache Tomcat 7 или более свежих версий.

Посмотрим в следующем разделе, как включить дополнительные возможности фреймворка MVC Spring.

Включение возможностей MVC Spring

Существует множество способов настройки `DispatcherServlet` и других веб-компонентов. Во фреймворке MVC Spring есть множество возможностей, недоступных по умолчанию, например `HttpMessageConverter`, поддержка проверки корректности входных данных контроллеров с помощью аннотации `@Valid` и т. д. Включить эти возможности можно с помощью как Java-, так и XML-конфигурации.

Для активизации MVC в Java-конфигурации достаточно добавить аннотацию `@EnableWebMvc` в один¹ из классов, снабженных аннотацией `@Configuration`:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
@Configuration
@EnableWebMvc
public class SpringMvcConfig {
}
```

В XML-конфигурации можно воспользоваться пространством имен MVC, в котором есть элемент `<mvc:annotation-driven>`, позволяющий активизировать возможности ориентированного на работу с аннотациями модуля MVC фреймворка Spring:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  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/mvc
    http://www.springframework.org/schema/mvc/spring-mvc.xsd">
  <mvc:annotation-driven/>

</beans>
```

¹ И только один!

Дополнительные возможности модуля MVC фреймворка Spring можно сделать доступными в веб-приложении или с помощью аннотации `@EnableWebMvc`, или посредством пространства имен XML `<mvc:annotation-driven/>`. Фреймворк MVC Spring позволяет также адаптировать используемую по умолчанию конфигурацию на Java к конкретной задаче посредством расширения класса `WebMvcConfigurerAdapter` или реализации интерфейса `WebMvcConfigurer`. Взглянем на модифицированный файл конфигурации, в который мы добавили дополнительные настройки:

```
package com.packt.patterninspring.chapter10.bankapp.web.mvc;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.
    DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.
    WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@ComponentScan(basePackages = {"
    com.packt.patterninspring.chapter10.bankapp.web.controller"})
@EnableWebMvc
public class SpringMvcConfig extends WebMvcConfigurerAdapter{
    @Bean
    public ViewResolver viewResolver(){
        InternalResourceViewResolver viewResolver = new
            InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/view/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
    @Override
    public void configureDefaultServletHandling(
        DefaultServletHandlerConfigurer configurer){
        configurer.enable();
    }
}
```

Как видно из предыдущего кода, класс конфигурации `SpringMvcConfig` снабжен аннотациями `@Configuration`, `@ComponentScan` и `@EnableWebMvc`. Здесь также видно, что производится поиск компонентов в пакете `com.packt.patterninspring.chapter10.bankapp.web.controller`. Этот класс расширяет класс `WebMvcConfigurerAdapter` и переопределяет метод `configureDefaultServletHandling()`. Мы также задали настройки компонента `ViewResolver`.

Пока вы узнали только, что такое паттерн MVC, какова его архитектура, как настроить `DispatcherServlet` и сделать доступными для веб-приложений важнейшие компоненты модуля MVC фреймворка Spring. В следующем разделе мы обсудим вопросы реализации контроллеров в приложениях Spring, а также обработки этими контроллерами веб-запросов.

Реализация контроллеров

Контроллеры являются одним из критически важных компонентов паттерна MVC. Они отвечают за фактическое выполнение запроса, подготовку модели и отправку этой модели единой точке входа вместе с логическим именем представления. В веб-приложениях контроллеры располагаются между веб-уровнем и уровнем ядра приложения. В фреймворке MVC Spring контроллеры скорее напоминают классы POJO с методами, эти методы называют обработчиками, поскольку они снабжены аннотациями `@RequestMapping`. Взглянем на описание классов контроллеров в веб-приложениях Spring.

Описание контроллера с помощью аннотации `@Controller`. Создадим класс контроллера для нашего банковского приложения. `HomeController` представляет собой класс контроллера, который обрабатывает запросы к корневому URL / и визуализирует домашнюю страницу банковского приложения:

```
package com.packt.patterninspring.chapter10.bankapp.web.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class HomeController {
    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String home () {
        return "home";
    }
}
```

Как вы можете видеть из предыдущего кода, класс `HomeController` включает метод `home()`. Это метод-обработчик, поскольку он снабжен аннотацией `@RequestMapping`. Она указывает, что данный метод обрабатывает все запросы, относящиеся к URL /. Стоит отметить также, что наш класс контроллера, `HomeController`, снабжен аннотацией `@Controller`. Как вы знаете, `@Controller` — одна из стереотипных аннотаций, используемая также для создания компонентов в контейнере IoC фреймворка Spring, подобно другим метааннотациям аннотации `@Component`, таким как `@Service` и `@Repository`. Да, эта аннотация объявляет любой класс контроллером и добавляет в этот класс дополнительные возможности модуля MVC Spring. Для создания компонентов Spring в веб-приложении можно также использовать аннотацию `@Component` вместо `@Controller`, но в этом случае у подобного

компонента будут отсутствовать такие возможности фреймворка MVC Spring, как обработка исключений на веб-уровне, отображение обработчиков и т. п.

Взглянем поближе на аннотацию `@RequestMapping`, а также на ее составные варианты.

Отображение запросов с помощью аннотации `@RequestMapping`

Ранее описанный класс `HomeController` включает только один метод-обработчик, снабженный аннотацией `@RequestMapping`. Там использовались два атрибута этой аннотации: атрибут `value` — для отображения HTTP-запроса на шаблон URL / и атрибут `method` — HTTP-метод (в данном случае GET), к которому осуществляется привязка. Можно описать несколько отображений URL в одном методе-обработчике, как показано в следующем фрагменте кода:

```
@Controller
public class HomeController {
    @RequestMapping(value = {"/", "/index"}, method = RequestMethod.GET)
    public String home () {
        return "home";
    }
}
```

В предыдущем коде у атрибута `value` аннотации `@RequestMapping` имеется массив строковых значений. Теперь этот метод-обработчик связан с двумя шаблонами URL, а именно: / и /index. Аннотация `@RequestMapping` модуля MVC фреймворка Spring поддерживает несколько HTTP-методов, таких как GET, POST, PUT, DELETE и т. д. По состоянию на версию 4.3 в Spring есть несколько составных вариантов аннотации `@RequestMapping` — простых методов для отображения основных методов HTTP, как показано в следующих выражениях:

```
@RequestMapping + HTTP GET = @GetMapping
@RequestMapping + HTTP POST = @PostMapping
@RequestMapping + HTTP PUT = @PutMapping
@RequestMapping + HTTP DELETE = @DeleteMapping
```

Вот модифицированная версия класса `HomeController` с составными аннотациями отображения:

```
@Controller
public class HomeController {
    @GetMapping(value = {"/", "/index"})
    public String home () {
        return "home";
    }
}
```

Аннотация `@RequestMapping` может использоваться на уровне как класса, так и метода. Рассмотрим несколько примеров.

@RequestMapping на уровне метода

MVC Spring позволяет отметить метод как обработчик путем использования аннотации `@RequestMapping` в веб-приложениях Spring на уровне метода. Проиллюстрируем ее использование в следующем классе:

```
package com.packt.patterninspring.chapter10.bankapp.web.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.packt.patterninspring.chapter10.bankapp.model.User;

@Controller
public class HomeController {
    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String home (){
        return "home";
    }
    @RequestMapping(value = "/create", method = RequestMethod.GET)
    public String create (){
        return "addUser";
    }
    @RequestMapping(value = "/create", method = RequestMethod.POST)
    public String saveUser (User user, ModelMap model){
        model.put("user", user);
        return "addUser";
    }
}
```

Как можно видеть в вышеприведенном коде, мы воспользовались аннотацией `@RequestMapping` для трех методов — `home()`, `create()` и `saveUser()`. Мы также воспользовались здесь атрибутами `value` и `method` данной аннотации. В атрибуте `value` задается отображение запроса на URL, а атрибут `method` применяется для задания метода HTTP-запроса, например `GET` или `POST`. Правила отображения обычно основываются на URL, в них могут использоваться джокерные символы, как показано ниже:

```
- /create
- /create/account
- /edit/account
- /listAccounts.htm — По умолчанию расширение игнорируется.
- /accounts/*
```

В предыдущем примере у методов-обработчиков есть аргументы, так что можно передать любое количество аргументов произвольного типа. MVC Spring работает с этими аргументами как с параметрами запроса. Рассмотрим сначала использование аннотации `@RequestMapping` на уровне класса, а потом обсудим параметры запроса.

@RequestMapping на уровне класса

MVC Spring позволяет использовать аннотацию `@RequestMapping` на уровне класса. Это значит, что можно снабдить класс контроллера аннотацией `@RequestMapping`, как показано в следующем фрагменте кода:

```
package com.packt.patterninspring.chapter10.bankapp.web.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/")

public class HomeController {
    @RequestMapping(method=GET)
    public String home() {
        return "home";
    }
}
```

Как видно в этом коде, класс `HomeController` снабжен аннотациями `@RequestMapping` и `@Controller`. А HTTP-метод по-прежнему указан над методами-обработчиками. Отображение на уровне класса применяется ко всем методам-обработчикам, описанным в этом контроллере.

После конфигурации MVC Spring мы создали класс контроллера с методами-обработчиками. Протестируем этот контроллер, прежде чем углубиться в дальнейшие подробности. В этой книге я не использую никаких тестовых сценариев библиотеки JUnit, так что здесь я просто запущу это веб-приложение в контейнере Tomcat. Вы должны увидеть в браузере следующие результаты его работы (рис. 10.8).



В версиях Spring до 3.1 модуль MVC Spring связывает запросы с обработчиками в два этапа. Во-первых, `DefaultAnnotationHandlerMapping` выбирает контроллер, а затем `AnnotationMethodHandlerAdapter` связывает с входящими запросами настоящий метод. Но, начиная с версии 3.1, модуль MVC Spring связывает запросы за один этап непосредственно с методами-обработчиками с помощью класса `RequestMappingHandlerMapping`.

В следующем разделе вы научитесь описывать методы-обработчики и задавать тип возвращаемого значения и допустимые для методов-обработчиков параметры в MVC Spring.

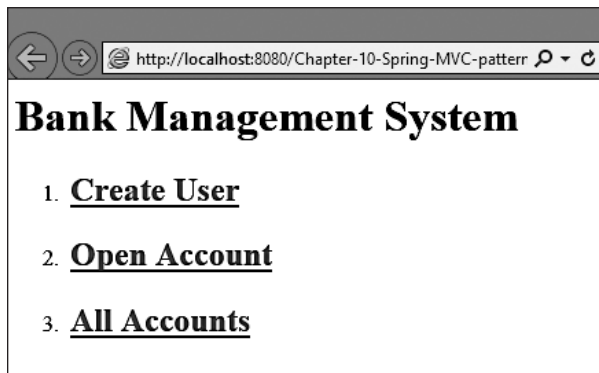


Рис. 10.8. Домашняя страница веб-приложения Bank Management System
(Система управления банком)

Описание методов-обработчиков с аннотацией @RequestMapping

Во фреймворке MVC Spring сигнатуры методов-обработчиков с аннотацией `@RequestMapping` очень гибки. Можно передавать произвольное количество аргументов в любом порядке. Эти методы поддерживают большинство типов аргументов и очень гибки в смысле возвращаемого значения. Типов возвращаемого значения может быть несколько, часть из них перечислена далее.

❑ Поддерживаемые типы аргументов метода:

- объект запроса или ответа (Servlet API);
- объект-сеанс (Servlet API);
- `java.util.Locale`;
- `java.util.TimeZone`;
- `java.io.InputStream/java.io.Reader`;
- `java.io.OutputStream/java.io.Writer`;
- `java.security.Principal`;
- `@PathVariable`;
- `@RequestParam`;
- `@RequestBody`;
- `@RequestPart`;
- `java.util.Map/org.springframework.ui.Model/org.springframework.ui.ModelMap`;
- `org.springframework.validation.Errors/org.springframework.validation.BindingResult`.

□ Поддерживаемые типы возвращаемого значения метода:

- ModelAndView;
- Model;
- Map;
- View;
- String;
- void;
- HttpEntity<?> или ResponseEntity<?>;
- HttpHeaders;
- Callable<?>;
- DeferredResult<?>.

Я перечислил часть поддерживаемых типов возвращаемого значения и типов аргументов метода. В отличие от других фреймворков MVC, MVC Spring представляется весьма гибким и легко адаптируемым под конкретную задачу инструментом в смысле описания методов обработчиков запросов.



Хотя во фреймворке MVC Spring порядок аргументов методов-обработчиков может быть любым, ошибки и параметры `BindingResult` необходимо помещать в начале списка с последующим указанием объекта модели для немедленного связывания, поскольку у метода-обработчика может быть произвольное число объектов модели и MVC Spring создает отдельные экземпляры объектов ошибок и объектов `BindingResult` для каждого из них. Например:

Неправильный порядок

```
@PostMapping
public String saveUser(@ModelAttribute ("user") User user,
    ModelState model, BindingResult result){...}
```

Правильный порядок

```
@PostMapping
public String saveUser(@ModelAttribute ("user") User user,
    BindingResult result, ModelState model){...}
```

Посмотрим теперь, как передавать данные модели на уровень представления.

Передача данных модели представлению

Пока мы реализовали и протестировали лишь очень простой класс `HomeController`. Но в веб-приложении мы также передаем данные модели на уровень представления. Эти данные передаются в модели (попросту говоря, представляют собой ассоциативный массив — объект типа `Map`), которую контроллер возвращает вместе с логическим именем представления. Как вы уже знаете, MVC Spring поддерживает

несколько типов возвращаемых значений для методов-обработчиков. Рассмотрим следующий пример:

```
package com.packt.patterninspring.chapter10.bankapp.web.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import com.packt.patterninspring.chapter10.bankapp.model.Account;
import com.packt.patterninspring.chapter10.bankapp.service.AccountService;

@Controller
public class AccountController {
    @Autowired
    AccountService accountService;
    @GetMapping(value = "/open-account")
    public String openAccountForm (){
        return "account";
    }
    @PostMapping(value = "/open-account")
    public String save (Account account, ModelMap model){
        account = accountService.open(account);
        model.put("account", account);
        return "accountDetails";
    }
    @GetMapping(value = "/all-accounts")
    public String all (ModelMap model){
        List<Account> accounts = accountService.findAllAccounts();
        model.put("accounts", accounts);
        return "accounts";
    }
}
```

Как видно из предыдущего примера, у класса `AccountController` есть три метода-обработчика. Два метода-обработчика возвращают данные модели вместе с логическим именем представления. Но в этом примере я использую класс `ModelMap` модуля MVC Spring, так что нет необходимости специально возвращать данные с именем логического представления — оно автоматически связывается с ответом.

А теперь разберемся, как принимать параметры запроса.

Принятие параметров запроса

Иногда в веб-приложениях Spring выполняется лишь чтение данных со стороны сервера, как в нашем примере. Чтение данных всех лицевых счетов требует лишь простого вызова функции чтения, для которого не требуется никаких параметров HTTP-запроса. Но в случае, если нужно извлечь данные из базы по конкретному лицевому счету, необходимо передать идентификатор счета через параметры

HTTP-запроса. Аналогично для создания нового счета в банке необходимо передать в качестве параметра соответствующий объект. В MVC Spring можно принимать параметры HTTP-запроса следующими способами:

- ❑ принятие параметров SQL-запроса;
- ❑ принятие параметров HTTP-запроса через переменные пути;
- ❑ принятие параметров форм.

Рассмотрим каждый из этих способов по очереди.

Принятие параметров SQL-запроса

В веб-приложении можно извлекать параметры запроса из него — в нашем примере это идентификатор лицевого счета, — если требуется получить доступ к детальной информации по конкретному счету. Извлечем идентификатор лицевого счета из параметра HTTP-запроса с помощью следующего кода:

```
@Controller
public class AccountController {
    @GetMapping(value = "/account")
    public String getAccountDetails (ModelMap model, HttpServletRequest
    request){
        String accountId = request.getParameter("accountId");
        Account account = accountService.findOne(Long.valueOf(accountId));
        model.put("account", account);
        return "accountDetails";
    }
}
```

В предыдущем фрагменте кода мы воспользовались традиционным способом обращения к параметрам HTTP-запроса. Для доступа к параметрам HTTP-запроса во фреймворке MVC Spring есть аннотация `@RequestParam`. Воспользуемся ею для привязки параметров запроса к параметру `method` нашего контроллера. Следующий фрагмент кода демонстрирует использование аннотации `@RequestParam`. Он извлекает параметр из HTTP-запроса и выполняет преобразование типов:

```
@Controller
public class AccountController {
    @GetMapping(value = "/account")
    public String getAccountDetails (ModelMap model,
    @RequestParam("accountId") long accountId){
        Account account = accountService.findOne(accountId);
        model.put("account", account);
        return "accountDetails ";
    }
}
```

В предыдущем коде мы получаем доступ к параметру HTTP-запроса с помощью аннотации `@RequestParam`. Можно также заметить, что мы не преобразовывали тип из `String` в `Long`, аннотация производит это автоматически. Еще стоит отметить, что по умолчанию при использовании этой аннотации параметры становятся обя-

зательными, но Spring позволяет изменить такое поведение с помощью атрибута `required` аннотации `@RequestParam`.

```
@Controller
public class AccountController {
    @GetMapping(value = "/account")
    public String getAccountDetails (ModelMap model,
    @RequestParam(name = "accountId") long accountId,
    @RequestParam(name = "name", required=false) String name){
        Account account = accountService.findOne(accountId);
        model.put("account", account);
        return " accountDetails ";
    }
}
```

Посмотрим теперь, как воспользоваться переменными пути для принятия входных данных в виде части пути HTTP-запроса.

Принятие параметров HTTP-запроса через переменные пути

Модуль MVC фреймворка Spring позволяет передавать параметры в URI вместо передачи их через параметры HTTP-запроса. Передаваемые значения можно извлечь из URL запроса. Этот метод основан на шаблонах URI. Данная концепция используется не только в Spring, но и во множестве других фреймворков с помощью заполнителей {...} и аннотации `@PathVariable`. Благодаря этому URL становятся аккуратнее и избавляются от громоздких параметров HTTP-запросов. Вот пример:

```
@Controller
public class AccountController {
    @GetMapping("/accounts/{accountId}")
    public String show(@PathVariable("accountId") long accountId, Model
    model) {
        Account account = accountService.findOne(accountId);
        model.put("account", account);
        return "accountDetails";
    }
    ...
}
```

В вышеприведенном обработчике метод может работать с запросами следующего вида (рис. 10.9):

<http://localhost:8080/Chapter-10-Spring-MVC-pattern/account?accountId=1000>

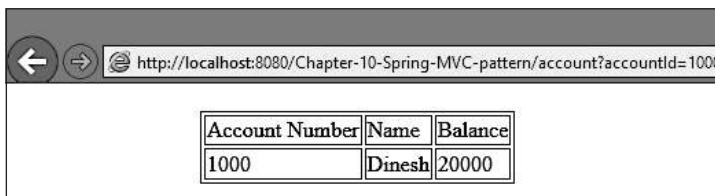


Рис. 10.9. Результат обработки запроса 1

Но в предыдущем примере метод-обработчик может обрабатывать запросы вот такого вида (рис. 10.10):

<http://localhost:8080/Chapter-10-Spring-MVC-pattern/accounts/2000>

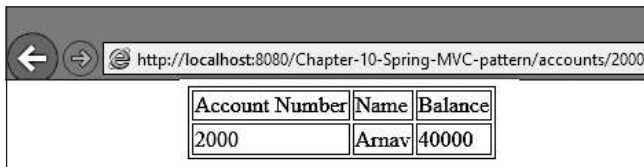


Рис. 10.10. Результат обработки запроса 2

В приведенных фрагментах кода и на рисунках показано, как передавать значения или через параметры HTTP-запроса, или с помощью параметров пути. Оба способа отлично подходят для передачи в запросе небольших объемов данных. Но в некоторых случаях приходится передавать на сервер большие объемы данных, например при отправке форм. Взглянем, как написать метод контроллера для обработки отправляемых форм.

Обработка форм веб-страницы

Как вы знаете, в любом веб-приложении можно отправлять данные на сервер и получать данные с него. Отправка данных в веб-приложениях происходит путем заполнения форм с последующей передачей их на сервер. MVC Spring также поддерживает обработку форм на стороне клиента посредством вывода этой формы, проверки введенных в нее данных и их отправки.

Фактически сначала MVC Spring выполняет показ формы и ее обработку. В банковском приложении нам нужно будет создать нового пользователя и открыть в банке новый лицевой счет, так что напомним класс контроллера, `AccountController`, содержащий один метод обработки запросов, предназначенный для отображения формы открытия лицевого счета:

```
package com.packt.patterninspring.chapter10.bankapp.web.controller;
```

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
```

```
@Controller
public class AccountController {
    @GetMapping(value = "/open-account")
    public String openAccountForm () {
        return "accountForm";
    }
}
```

Аннотация `@GetMapping` метода `openAccountForm()` объявляет, что он будет обрабатывать HTTP-запросы типа GET для URL `/open-account`. Это простой метод,

не требующий входных данных и возвращающий лишь логическое представление `accountForm`. Мы задали настройки класса `InternalResourceViewResolver`, в результате чего для визуализации формы открытия лицевого счета будет вызываться JSP по адресу `/WEB-INF/views/accountForm.jsp`.

Вот код JSP, который мы будем пока использовать:

```
<%@ taglib prefix = "c" uri = "http://java.sun.com/jsp/jstl/core" %>
<html>
  <head>
    <title>Bank Management System</title>
    <link rel="stylesheet" type="text/css" href="<c:url
      value="/resources/style.css" />" >
  </head>
  <body>
    <h1>Open Account Form</h1>
    <form method="post">
      Account Number:<br>
      <input type="text" name="id"><br>
      Account Name:<br>
      <input type="text" name="name"><br>
      Initial Balance:<br>
      <input type="text" name="balance"><br>
      <br>
      <input type="submit" value="Open Account">
    </form>
  </body>
</html>
```

Как можно видеть из предыдущего кода, у нас есть форма открытия лицевого счета с такими полями, как `AccountId` (идентификатор лицевого счета), `Account Name` (название лицевого счета) и `Initial Balance` (начальный баланс). В коде этой JSP-страницы есть тег `<form>` для формы, у которого нет никаких параметров действия. Это значит, что при отправке этой формы данные будут переданы на тот же URI `/open-account` с помощью вызова HTTP-метода `POST`. Форма создания счета показана на рис. 10.11.

The screenshot shows a web browser window with the address bar set to `http://localhost:8080/`. The page title is `Bank Management System`. The main content area displays the **Open Account Form**. The form includes three text input fields: **Account Number:** with the value `10000`, **Account Name:** with the value `Dinesh Rejput`, and **Initial Balance:** with the value `20000`. At the bottom of the form is a button labeled `Open Account`.

Рис. 10.11. Форма создания счета

Добавим теперь еще один метод для обработки вызова HTTP-метода POST с тем же URI /open-account.

Реализация контроллера обработки форм

Добавим в тот же класс `AccountController` еще один метод-обработчик для HTTP-запроса типа POST к URI /open-account в веб-приложении:

```
package com.packt.patterninspring.chapter10.bankapp.web.controller;

import java.util.List;

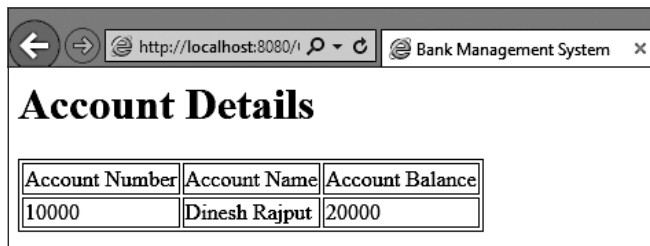
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;

import com.packt.patterninspring.chapter10.bankapp.model.Account;
import com.packt.patterninspring.chapter10.bankapp.service.AccountService;

@Controller
public class AccountController {
    @Autowired
    AccountService accountService;
    @GetMapping(value = "/open-account")
    public String openAccountForm (){
        return "accountForm";
    }
    @PostMapping(value = "/open-account")
    public String save (Account account){
        accountService.open(account);
        return "redirect:/accounts/"+account.getId();
    }
    @GetMapping(value = "/accounts/{accountId}")
    public String getAccountDetails (ModelMap model, @PathVariable Long
    accountId){
        Account account = accountService.findOne(accountId);
        model.put("account", account);
        return "accountDetails";
    }
}
```

Мы добавили еще два метода-обработчика в класс `AccountController`, а также внедрили в этот контроллер сервис `AccountService` для сохранения подробных данных о счете в базе данных. При каждой обработке HTTP-запроса типа POST из формы открытия счета контроллер принимает данные формы и сохраняет их в базе данных с помощью внедренного сервиса. Он принимает данные формы открытия

счета в виде объекта класса `Account`. Вы могли также заметить, что после обработки данных формы с помощью HTTP-запроса типа `POST` метод-обработчик перенаправляет пользователя на страницу подробных данных о счете. Перенаправление после отправки данных посредством `POST` предотвращает случайную отправку данных формы повторно и рекомендуется к использованию. После отправки запроса вы увидите следующую таблицу (рис. 10.12).



The screenshot shows a web browser window with the address bar displaying 'http://localhost:8080/' and the page title 'Bank Management System'. The main content area has the heading 'Account Details' and a table with the following data:

Account Number	Account Name	Account Balance
10000	Dinesh Rajput	20000

Рис. 10.12. Экран после отправки запроса

Данная страница визуализируется после отправки формы открытия счета. Добавленный нами метод обрабатывает запрос и визуализирует еще одну веб-страницу — с подробной информацией о счете. В качестве представления выше-приведенного вывода визуализируется следующая JSP-страница:

```
<%@ taglib prefix = "c" uri = "http://java.sun.com/jsp/jstl/core" %>
<html>
  <head>
    <title>Bank Management System</title>
    <link rel="stylesheet" type="text/css" href="<c:url
      value="/resources/style.css" />" />
  </head>
  <body>
    <h1>${message} Account Details</h1>
    <c:if test="${not empty account }">
      <table border="1">
        <tr>
          <td>Account Number</td>
          <td>Account Name</td>
          <td>Account Balance</td>
        </tr>
        <tr>
          <td>${account.id }</td>
          <td>${account.name }</td>
          <td>${account.balance }</td>
        </tr>
      </table>
    </c:if>
  </body>
</html>
```

В этом коде метод-обработчик отправляет объект `Account` модели, а также возвращает логическое имя представления. Эта JSP-страница визуализирует полученный из ответа объект `Account`.



Стоит отметить, что у объекта `Account` есть свойства: идентификатор, имя и баланс, заполняемые из одноименных (в смысле имени поля в форме открытия счета) параметров HTTP-запроса. Если имя любого свойства объекта совпадает с именем поля в HTML-форме, то это свойство будет инициализировано значением `NULL`.

Привязка данных с помощью паттерна проектирования «Команда»

Инкапсулирует запрос в объекте, позволяя таким образом задавать параметры клиентов для обработки соответствующих запросов, организовывать очереди и журналирование запросов, а также поддержку допускающих отмену операций.

GoF. Приемы объектно-ориентированного проектирования: паттерны проектирования

Паттерн проектирования «Команда» описывается в главе 3. Он входит в семейство поведенческих паттернов GoF и является очень простым, ориентированным на работу с данными паттерном, который позволяет инкапсулировать данные HTTP-запроса в объекте и передавать этот объект в виде команды методу-инициатору, который затем возвращает результат выполнения этой команды в виде другого объекта изначальной вызывающей стороне.

MVC Spring реализует паттерн проектирования «Команда» с целью привязки данных запроса из веб-формы к объекту с последующей передачей этого объекта методу-обработчику запроса из класса контроллера. В этом разделе мы изучим использование этого паттерна для привязки данных запроса к объекту, а также обсудим выгоды и возможности использования привязки данных. В следующем классе Java-компонент `Account` представляет собой простой объект с тремя свойствами: `id`, `name` и `balance`:

```
package com.packt.patterninspring.chapter10.bankapp.model;
```

```
public class Account{
    Long id;
    Long balance;
    String name;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
}
```



```
public Long getBalance() {
    return balance;
}
public void setBalance(Long balance) {
    this.balance = balance;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
@Override
public String toString() {
    return "Account [id=" + id + ", balance=" + balance + ", name=" +
        name + "]";
}
}
```

Вариантов здесь два: или мы отправляем веб-форму с именами входных текстовых полей, совпадающими с названиями свойств объекта, или получаем запрос в виде `http://localhost:8080/Chapter-10-Spring-MVC-pattern/account?id=10000`. В обоих случаях Spring «за кулисами» вызывает метод-сеттер класса `Account` для привязки данных запроса или данных веб-формы к объекту. Spring также позволяет осуществлять привязку индексированных коллекций — списков, ассоциативных массивов и т. п.

Привязки данных можно адаптировать под конкретную задачу. В Spring это можно сделать двумя способами.

- ❑ *Глобальная адаптация* — модифицирует привязки данных по всему веб-приложению для конкретного объекта типа `Command`.
- ❑ *Адаптация под конкретный контроллер* — модифицирует привязки данных индивидуально для каждого класса контроллера в отношении конкретного объекта типа `Command`.

Мы обсудим только адаптацию под конкретный контроллер. Рассмотрим следующий фрагмент кода для индивидуальной адаптации привязок данных для объекта `Account`:

```
package com.packt.patterninspring.chapter10.bankapp.web.controller;
```

```
....
....
@Controller
public class AccountController {
    @Autowired
    AccountService accountService;
    ....
    ....
    @InitBinder
```

```

public void initBinder(WebDataBinder binder) {
    binder.initDirectFieldAccess();
    binder.setDisallowedFields("id");
    binder.setRequiredFields("name", "balance");
}
...
...
}

```

У класса `AccountController` есть метод `initBinder(WebDataBinder binder)`, снабженный аннотацией `@InitBinder`. Тип возвращаемого значения у него должен быть `void`, а в качестве аргумента метода должен быть объект типа `org.springframework.web.bind.WebDataBinder`. У объекта `WebDataBinder` есть несколько методов, мы воспользовались некоторыми из них в вышеприведенном коде. `WebDataBinder` используется для адаптации привязок данных под конкретную задачу.

Использование аннотации `@ModelAttribute` для адаптации привязок данных под конкретную задачу. В MVC Spring есть еще одна аннотация, `@ModelAttribute`, для привязки данных к объекту типа `Command`. Это еще один способ привязки данных и адаптации привязки данных под конкретную задачу. Эта аннотация дает возможность управлять созданием объекта типа `Command`. В приложении MVC Spring можно применить эту аннотацию к методу или аргументам метода. Рассмотрим следующие примеры.

- ❑ Применение аннотации `@ModelAttribute` к методам.

Аннотацию `@ModelAttribute` можно применять к методам для создания используемого в форме объекта, например:

```

package com.packt.patterninspring.chapter10.bankapp.web.controller;
....
....
@Controller
public class AccountController {
    ....
    @ModelAttribute
    public Account account () {
        return new Account();
    }
    ....
}

```

- ❑ Применение аннотации `@ModelAttribute` к аргументам методов.

Можно использовать эту аннотацию также и для аргументов методов. В этом случае аргументы метода-обработчика извлекаются из объекта модели. Если же в модели их нет, то они создаются с помощью конструктора по умолчанию:

```

package com.packt.patterninspring.chapter10.bankapp.web.controller;
....
....
@Controller
public class AccountController {
    ...

```

```

@PostMapping(value = "/open-account")
public String save (@ModelAttribute("account") Account account){
    accountService.open(account);
    return "redirect:/accounts/"+account.getId();
}
....
}

```

Как можно видеть в этом фрагменте кода, аннотация `@ModelAttribute` применяется к аргументу метода. Это значит, что объект `Account` извлекает данные из объекта модели. При отсутствии объекта модели он будет создан с помощью конструктора по умолчанию.



Метод, к которому применяется аннотация `@ModelAttribute`, будет вызван до вызова метода-обработчика запроса.

До сих пор мы изучали, как MVC Spring обрабатывает запросы и параметры запроса или традиционным способом, или с помощью аннотаций `@RequestParam` и `@PathVariable`. Мы также изучили обработку веб-страниц с формами и обработку запросов типа POST с привязкой данных формы к объекту на уровне контроллера. Перейдем теперь к проверке корректности/некорректности (с точки зрения предметной области) отправляемых данных формы.

Проверка корректности входных параметров форм

В веб-приложениях проверка корректности данных веб-форм чрезвычайно важна, ведь конечные пользователи могут ввести туда что угодно. Допустим, что в приложении пользователь отправляет форму открытия счета, в которой заполняет название счета, после чего в банке создается новый счет с соответствующим именем владельца. Разумеется, перед созданием новой записи в базе данных необходимо проверить корректность данных формы. Логика проверки не обязана находиться в методе-обработчике. Spring поддерживает API JSR-303. Начиная с версии Spring 3.0, модуль MVC Spring поддерживает это API валидации Java. Настроек для его использования в веб-приложениях Spring требуется немного — нужно просто добавить одну из его реализаций (например, `Hibernate Validator`) в путь классов приложения.

В API валидации Java есть несколько аннотаций для проверки свойств объекта `Command`. Они позволяют, в частности, налагать ограничения на значение свойств объекта `Command`. В этой главе мы не станем изучать все эти аннотации, а лишь рассмотрим следующие примеры с некоторыми из них:

```

package com.packt.patterninspring.chapter10.bankapp.model;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

```

```

public class Account{
    // Непустое значение
    @NotNull
    Long id;
    // Непустое значение
    @NotNull
    Long balance;
    // Непустое значение, от 2 до 30 символов
    @NotNull
    @Size(min=2, max=30)
    String name;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public Long getBalance() {
        return balance;
    }
    public void setBalance(Long balance) {
        this.balance = balance;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
        return "Account [id=" + id + ", balance=" + balance + ", name=" +
            name + "]";
    }
}

```

Как видно из предыдущего кода, свойства класса `Account` снабжены аннотациями `@NotNull`, гарантирующими, что значения не будут пустыми, а некоторые — и аннотацией `@Size`, гарантирующей, что количество символов значения будет находиться в заданных пределах.

Снабжения аннотациями одних только свойств объекта `Account` недостаточно. Необходимо также аннотировать аргумент метода `save()` класса `AccountController`:

```

package com.packt.patterninspring.chapter10.bankapp.web.controller;
....
....
@Controller
public class AccountController {
    ....
    @PostMapping(value = "/open-account")
    public String save (@Valid @ModelAttribute("account") Account account,
        Errors errors){

```

```

    if (errors.hasErrors()) {
        return "accountForm";
    }
    accountService.open(account);
    return "redirect:/accounts/"+account.getId();
}
....
}

```

Как можно видеть в предыдущем коде, параметр `account` теперь аннотирован `@Valid`, что указывает фреймворку Spring на наличие у объекта команды проверочных ограничений, соблюдение которых необходимо обеспечить. Взглянем на результаты отправки веб-формы создания счета при внесении некорректных данных (рис. 10.13).

The screenshot shows a web browser window with the address bar displaying `http://localhost:8080/` and the page title "Bank Management System". The main content area is titled "Open Account Form". It contains three input fields with associated validation error messages:

- Account Number:** The input field is empty, and the error message "may not be null" is displayed below it.
- Account Name:** The input field is empty, and the error message "size must be between 2 and 30" is displayed below it.
- Initial Balance:** The input field is empty, and the error message "may not be null" is displayed below it.

At the bottom of the form, there is a button labeled "Open Account".

Рис. 10.13. Результаты отправки веб-формы

Попытавшись отправить эту форму вообще без данных, я оказался перенаправлен на ту же страницу с ошибками проверки. Spring также предоставляет возможности задания пользовательских сообщений о результатах проверки путем указания их в файле свойств.

Пока что в этой главе вы узнали о компоненте «Контроллер» паттерна MVC. Вы также научились создавать и настраивать его в веб-приложении. Взглянем на еще один компонент паттерна MVC — «Представление» — в следующем разделе.

Реализация компонента «Представление» в паттерне MVC

Представление — важнейший компонент паттерна MVC. Контроллер возвращает модель единой точке входа вместе с логическим именем представления. Единая точка входа разрешает это имя в реальное представление с помощью настроенного заранее арбитра представлений. MVC Spring поддерживает множество различных

технологий представлений — JSP, Velocity, FreeMarker, JSF, Tiles, Thymeleaf и др., посредством предоставления различных арбитров представлений. При этом необходимо задать настройки арбитра представлений в соответствии с используемой в приложении технологией представлений. Взглянем на следующую схему, чтобы лучше разобраться с паттерном представлений в MVC Spring (рис. 10.14).

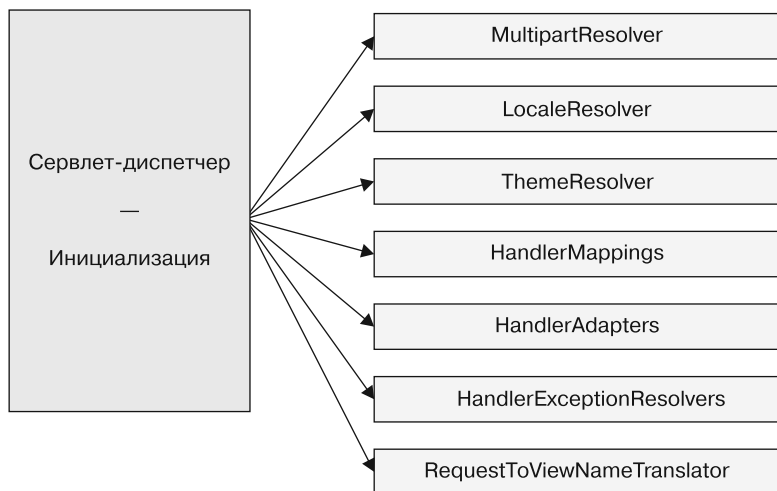


Рис. 10.14. Схема, поясняющая работу паттерна представлений

Как можно видеть на этой схеме, у единой точки входа фреймворка MVC Spring есть несколько арбитров представлений для разных технологий представлений. Но в этой главе мы будем использовать в качестве технологии представлений только JSP, а значит, будем говорить только про относящийся к JSP арбитр представлений — `InternalResourceViewResolver`.

Представление визуализирует выводимые на веб-страницу данные. Существует множество встроенных представлений для JSP, XSLT, шаблонизаторов (Velocity, FreeMarker) и др. В MVC Spring есть также вспомогательные классы представлений для создания PDF, электронных таблиц Excel и т. д.

Контроллеры в MVC Spring обычно возвращают *логическое имя представления*, а арбитры представлений Spring подбирают представление по имени. Посмотрим, как задать настройки арбитра представлений в приложении MVC Spring.

Описание арбитра представлений в MVC Spring

В MVC Spring сервлет-диспетчер делегирует обработку арбитра представлений, чтобы получить соответствующую имени представления реализацию представления. Арбитр представлений по умолчанию трактует имя представления как относительный путь файла веб-приложения, то есть для JSP это будет так: `/WEB-INF/views/account.jsp`. Эту настройку, задаваемую по умолчанию, можно

перекрыть, зарегистрировав компонент `ViewResolver` в `DispatcherServlet`. В нашем веб-приложении мы использовали `InternalResourceViewResolver`, поскольку речь идет о JSP-представлении, но в Spring MVC есть несколько других возможностей, как упоминалось в предыдущем разделе.

Реализация представления

Следующий код визуализирует представление в паттерне MVC:

`accountDetails.jsp` :

```
<%@ taglib prefix = "c" uri = "http://java.sun.com/jsp/jstl/core" %>
<html>
  <head>
    <title>Bank Management System</title>
    <link rel="stylesheet" type="text/css" href="<c:url
      value="/resources/style.css" />" >
  </head>
  <body>
    <h1>${message} Account Details</h1>
    <c:if test="${not empty account }">
      <table border="1">
        <tr>
          <td>Account Number</td>
          <td>Account Name</td>
          <td>Account Balance</td>
        </tr>
        <tr>
          <td>${account.id }</td>
          <td>${account.name }</td>
          <td>${account.balance }</td>
        </tr>
      </table>
    </c:if>
  </body>
</html>
```

Здесь MVC Spring визуализирует данное представление, когда контроллер возвращает `accountDetails` в качестве логического имени представления. Но как это MVC Spring разрешает это имя? Посмотрим на настройки `ViewResolver` в файле конфигурации Spring.

Регистрация ViewResolver в MVC Spring

Зарегистрируем предназначенный для JSP `ViewResolver`, то есть зададим настройки `InternalResourceViewResolver` в веб-приложении Spring:

```
package com.packt.patterninspring.chapter10.bankapp.web.mvc;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
```

```

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.
    WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@ComponentScan(basePackages =
{"com.packt.patterninspring.chapter10.bankapp.web.controller"})
@EnableWebMvc
public class SpringMvcConfig extends WebMvcConfigurerAdapter{
    ....
    @Bean
    public ViewResolver viewResolver(){
        InternalResourceViewResolver viewResolver = new
        InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
    ....
}

```

Допустим, в предыдущем коде контроллер возвращает логическое имя представления, `accountDetails`. Все JSP-файлы представлений размещаются в каталоге `/WEB-INF/views/` веб-приложения. `accountDetails.jsp` представляет собой файл представления для подробной информации о счете. Согласно вышеприведенному файлу конфигурации фактическое имя представления получается из логического путем добавления `/WEB-INF/views/` до и `.jsp` — после логического имени представления, возвращаемого контроллером приложения. Если контроллер приложения вернул `accountDetails` в качестве логического имени представления, то `ViewResolver` поменяет его на физическое, добавив указанные части к логическому; в результате мы получим в нашем приложении имя `/WEB-INF/views/accountDetails.jsp`. Следующая схема иллюстрирует, как единая точка входа MVC Spring разрешает имена представлений в веб-приложении Spring (рис. 10.15).

Предыдущая схема иллюстрирует всю картину движения запроса MVC Spring, со всеми компонентами (моделью, представлением и контроллером) паттерна MVC, а также паттерн «Единая точка входа». Любой HTTP-запрос, GET или POST, сначала оказывается в единой точке входа, а именно в объекте `DispatcherServlet` MVC Spring. Контроллеры в веб-приложениях Spring отвечают за генерацию и обновление модели, где модель — еще один компонент паттерна MVC. Наконец, контроллер возвращает модель вместе с логическим именем представления объекту `DispatcherServlet`. С помощью обращения к настроенному арбитру представлений он разрешает физический путь представления. Представление — еще один компонент паттерна MVC.

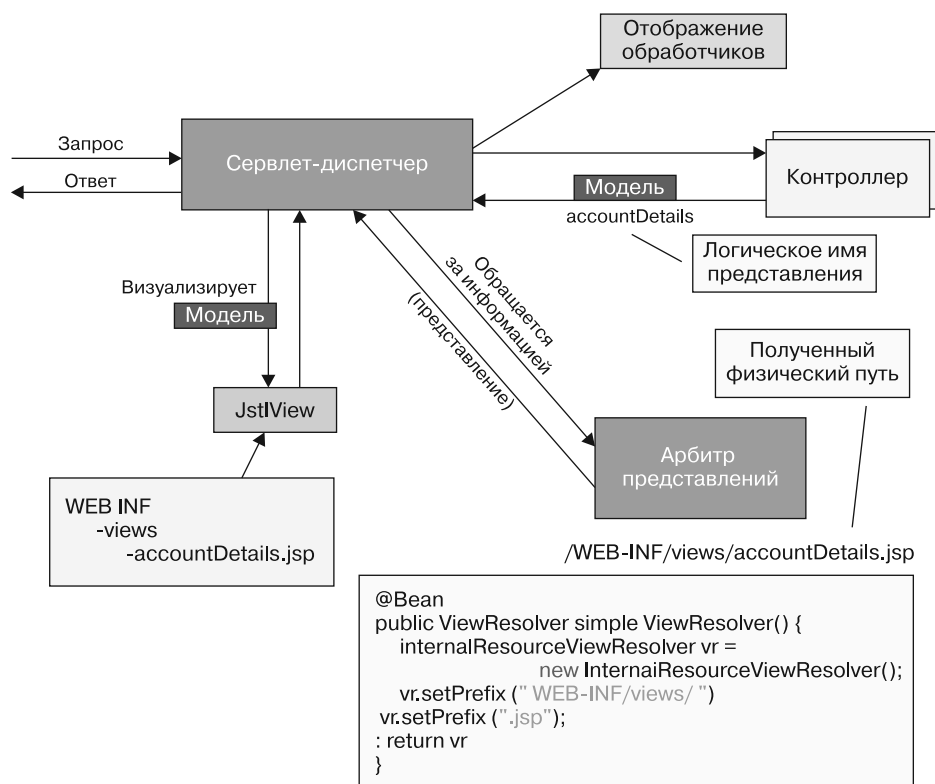


Рис. 10.15. Жизненный цикл обработки запроса

В следующем разделе мы подробно поговорим о паттерне «Вспомогательный компонент представления» (View Helper), а также о поддержке фреймворком Spring этого паттерна в веб-приложениях Spring.

Паттерн «Вспомогательный компонент представления»

Паттерн «Вспомогательный компонент представления» разделяет статическое представление, например JSP, с обработкой данных бизнес-модели. Паттерн «Вспомогательный компонент представления» используется на уровне визуализации, модифицируя данные модели и компоненты представления. Вспомогательный компонент представления может придавать данным модели соответствующую бизнес-требованиям форму, но не может генерировать их. Следующая схема иллюстрирует паттерн «Вспомогательный компонент представления» (рис. 10.16).

Представление — это статический форматированный компонент паттерна MVC, но иногда на уровне визуализации тоже требуется выполнение определенной бизнес-логики. При использовании JSP-страниц можно задействовать скриптлет для

подобной обработки данных на уровне представлений, но этого делать не рекомендуется из-за возникающего при этом сильного сцепления между представлением и бизнес-логикой. Впрочем, некоторые из вспомогательных классов представлений, основанные на рассматриваемом паттерне, берут на себя обязанности по выполнению бизнес-логики на уровне визуализации. Вот некоторые из основанных на этом паттерне технологий:

- ❑ вспомогательный Java-компонент представления;
- ❑ вспомогательный компонент представления на основе библиотеки тегов;
- ❑ с использованием тегов JSTL;
- ❑ с использованием тегов Spring;
- ❑ с использованием сторонней библиотеки тегов.

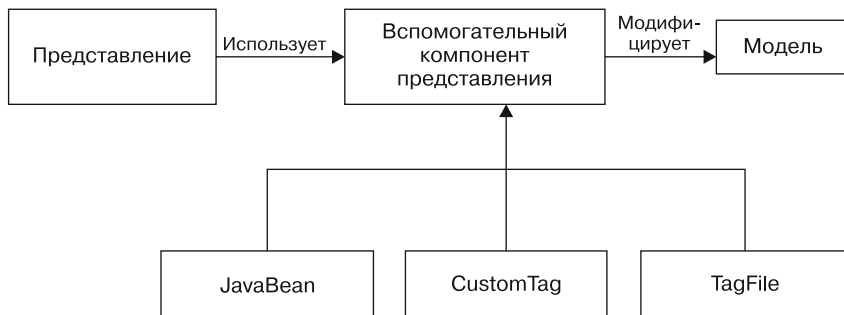


Рис. 10.16. Принцип работы паттерна «Вспомогательный компонент представления»

В нашем веб-приложении в этой главе используются следующие библиотеки тегов:

```

<%@ taglib prefix = "c" uri = "http://java.sun.com/jsp/jstl/core" %>
<c:if test="${not empty account }">
    ....
    ....
</c:if>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<form:form method="post" commandName="account">
    ....
    ...
</form:form>
  
```

Здесь я использовал библиотеку тегов JSTL для проверки того, не пусто ли поле `account` в модели, и библиотеку тегов Spring для создания формы открытия счета в веб-приложении.

В следующем разделе вы узнаете о паттерне «Составное представление», а также о том, какую поддержку его реализации в веб-приложениях предоставляет MVC Spring.

Паттерн «Составное представление» и использование арбитра представлений фреймворка Apache Tiles

В веб-приложении представление — один из самых важных компонентов. Разработка этого компонента не так проста, как может показаться, а его сопровождение — чрезвычайно сложная задача. При создании представления для веб-приложения всегда нужно уделять особое внимание переиспользуемости компонентов представления. Согласно паттерну проектирования «Составной объект» (Composite) из числа паттернов GoF компоненты субпредставлений объединяются в нужный компонент представления. Паттерн «Составное представление» способствует переиспользованию представлений, его сопровождение не представляет сложности благодаря использованию нескольких простых субпредставлений вместо создания большого и сложного представления. Следующая схема иллюстрирует паттерн «Составное представление» (рис. 10.17).



Рис. 10.17. Объединение подкомпонентов в нужный компонент представления

Как можно видеть на предыдущей схеме, представление в веб-приложении можно создавать из нескольких субпредставлений, переиспользуемых в разных местах веб-приложения.

MVC Spring поддерживает реализацию паттерна «Составное представление» с помощью таких фреймворков, как SiteMesh и Apache Tiles. Мы рассмотрим использование Apache Tiles для приложений MVC Spring. Итак, взглянем, как настроить объект `ViewResolver` фреймворка Apache Tiles.

Задание настроек объекта `ViewResolver` фреймворка Apache Tiles. Определим настройки для `ViewResolver` фреймворка Apache Tiles в приложении MVC Spring. Для этого понадобится задать настройки двух компонентов в файле конфигурации Spring:

```
package com.packt.patterninspring.chapter10.bankapp.web.mvc;
.....
@Configuration
@ComponentScan(basePackages =
{"com.packt.patterninspring.chapter10.bankapp.web.controller"})
```

```

@EnableWebMvc
public class SpringMvcConfig extends WebMvcConfigurerAdapter{
    ....
    @Bean
    public TilesConfigurer tilesConfigurer() {
        TilesConfigurer tiles = new TilesConfigurer();
        tiles.setDefinitions(new String[] {
            "/WEB-INF/layout/tiles.xml"
        });
        tiles.setCheckRefresh(true);
        return tiles;
    }
    @Bean
    public ViewResolver viewResolver() {
        return new TilesViewResolver();
    }
    ...
}

```

В предыдущем файле конфигурации мы задали настройки двух компонентов, `TilesConfigurer` и `TilesViewResolver`. Первый из них, `TilesConfigurer`, отвечает за обнаружение и загрузку описаний шаблонов Tiles и общую координацию работы Apache Tiles. К области ответственности второго компонента, `TilesViewResolver`, относится разрешение логических имен представлений в описания шаблонов Tiles. Описания этих шаблонов находятся в приложении в XML-файле `tiles.xml`. Взглянем на следующий код файла конфигурации шаблонов Tiles:

```

<tiles-definitions>
  <definition name="base.definition" template="/WEB-INF/views/mainTemplate.jsp">
    <put-attribute name="title" value=""/>
    <put-attribute name="header" value="/WEB-INF/views/header.jsp"/>
    <put-attribute name="menu" value="/WEB-INF/views/menu.jsp"/>
    <put-attribute name="body" value=""/>
    <put-attribute name="footer" value="/WEB-INF/views/footer.jsp"/>
  </definition>
  <definition extends="base.definition" name="openAccountForm">
    <put-attribute name="title" value="Account Open Form"/>
    <put-attribute name="body" value="/WEB-INF/views/accountForm.jsp"/>
  </definition>
  <definition extends="base.definition" name="accountsList">
    <put-attribute name="title" value="Employees List"/>
    <put-attribute name="body" value="/WEB-INF/views/accounts.jsp"/>
  </definition>
  ...
  ...
</tiles-definitions>

```

В предыдущем коде элемент `<tiles-definitions>` включает несколько элементов `<definition>`. Каждый элемент `<definition>` описывает шаблон Tiles, а каждый шаблон Tiles ссылается на шаблон JSP. Некоторые элементы `<definition>` расширяют описание базового шаблона, поскольку описание базового шаблона содержит общий для всех представлений веб-приложения макет веб-страницы.

Взглянем на описание базового шаблона, а именно `mainTemplate.jsp`:

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="t" %>
<%@ page session="false" %>
<html>
  <head>
    <title>
      <tiles:insertAttribute name="title" ignore="true"/>
    </title>
  </head>
  <body>
    <table border="1" cellpadding="2" cellspacing="2" align="left">
      <tr>
        <td colspan="2" align="center">
          <tiles:insertAttribute name="header"/>
        </td>
      </tr>
      <tr>
        <td>
          <tiles:insertAttribute name="menu"/>
        </td>
        <td>
          <tiles:insertAttribute name="body"/>
        </td>
      </tr>
      <tr>
        <td colspan="2" align="center">
          <tiles:insertAttribute name="footer"/>
        </td>
      </tr>
    </table>
  </body>
</html>
```

В этом JSP-файле я воспользовался JSP-тегом `<tiles:insertAttribute>` из библиотеки тегов `tiles` для вставки других шаблонов.

Рекомендуемые практики проектирования веб-приложений

Приведу несколько рекомендуемых практик, которые пригодятся вам при проектировании и разработке веб-приложений.

- ❑ MVC Spring — оптимальный выбор для проектирования и разработки веб-приложения благодаря паттерну DI Spring и чрезвычайно гибкому паттерну MVC. Класс `DispatcherServlet` фреймворка Spring тоже очень гибок и легко адаптируется к любой задаче.
- ❑ Единая точка входа в любом веб-приложении, которое использует паттерн MVC, должна быть универсальной и как можно более облегченной.

- ❑ Важно соблюдать четкое разделение обязанностей между уровнями веб-приложения. Разделение уровней повышает чистоту программного проекта приложения.
- ❑ Если число зависимостей одного из уровней приложения от других слишком велико, то оптимально будет ввести еще один уровень для снижения количества зависимостей.
- ❑ Никогда не внедряйте объекты DAO в контроллеры в веб-приложении; всегда внедряйте объекты сервисов в контроллер. Объекты DAO должны внедряться в уровень сервисов, так чтобы уровень сервисов взаимодействовал с уровнем доступа к данным, а уровень визуализации взаимодействовал с уровнем сервисов.
- ❑ Такие уровни приложения, как уровень сервисов, DAO и уровень визуализации, должны быть подключаемыми, и реализация не должна их ни в чем ограничивать: использование интерфейсов ослабляет сцепление по сравнению с конкретными реализациями, а слабо сцепленные многоуровневые приложения легче тестировать и сопровождать.
- ❑ Настоятельно рекомендуется размещать JSP-файлы в каталоге WEB-INF, поскольку к нему не обращается ни один клиент.
- ❑ Всегда указывайте имя объекта команды в JSP-файле.
- ❑ В JSP-файлах не должно содержаться никакой бизнес-логики. Для выполнения этого требования настоятельно советую вам использовать вспомогательные классы представлений — классы, библиотеки, JSTL и т. п.
- ❑ Уберите логику программы из основанных на шаблонах представлений, например JSP.
- ❑ Создавайте переиспользуемые компоненты, которые можно было бы применять для комбинирования данных модели в представлениях.
- ❑ Каждый компонент паттерна MVC должен вести себя так, как предполагает MVC. Это значит, что контроллер должен следовать принципу единственной обязанности. Контроллеры отвечают только за делегирование вызовов бизнес-логики и выбор представлений.
- ❑ Наконец, называйте файлы конфигурации согласованным образом. Например, веб-компоненты — контроллеры, перехватчики и арбитры представлений — должны описываться в отдельных файлах конфигурации. Другие компоненты приложения — сервисы, репозитории и т. п. — должны описываться в отдельном файле. Аналогично следует поступать с элементами безопасности.

Резюме

В этой главе вы наблюдали, как фреймворк Spring предоставляет возможность создания гибкого и слабо сцепленного веб-приложения. Spring применяет аннотации в веб-приложениях для приближенной к POJO модели разработки. Вы узнали, что с помощью MVC Spring можно создавать веб-приложения, используя обрабатывающие запросы контроллеры, которые очень легко тестировать. В этой главе мы

рассмотрели паттерн MVC, включая его сущность и решаемые им задачи. Фреймворк Spring реализует паттерн MVC, а значит, в любом веб-приложении есть три компонента: модель, представление, контроллер.

MVC Spring реализует паттерны проектирования «Контроллер приложения» и «Единая точка входа». Сервлет-диспетчер фреймворка Spring (класс `org.springframework.web.servlet.DispatcherServlet`) служит единой точкой входа веб-приложения. Этот диспетчер (единая точка входа) маршрутизирует все запросы к контроллеру приложения с помощью отображения обработчиков. В MVC Spring методы-обработчики запросов у классов контроллеров чрезвычайно гибки. И эти методы обрабатывают все запросы веб-приложения. Как я объяснял в этой главе, существует несколько способов обработки параметров запросов. Один из них — использование аннотации `@RequestParam`, обеспечивающей удобство тестирования без применения в тестовых сценариях объектов HTTP-запросов.

В этой главе мы исследовали технологический процесс обработки запроса и обсудили все участвующие в нем компоненты. Основным компонентом MVC Spring можно считать `DispatcherServlet`, играющий в MVC Spring роль единой точки входа. Еще один важный компонент — арбитр представлений, отвечающий за визуализацию данных модели в каком-либо шаблоне представления, например JSP, Thymeleaf, FreeMarker, Velocity, PDF, XML и т. д., в зависимости от настроек арбитра представлений в веб-приложении. MVC Spring поддерживает несколько технологий представлений, но в этой главе мы вкратце рассмотрели лишь написание представлений для контроллеров с помощью JSP. Существует также возможность добавления согласованных макетов представлений с помощью фреймворка Apache Tiles.

И наконец, мы охватили вопрос архитектуры веб-приложения, а также обсудили различные уровни веб-приложения, такие как предметная область, интерфейс пользователя, веб-интерфейс, сервисы и доступ к данным. Мы создали небольшое веб-приложение для управления банком и развернули его на сервере Tomcat.

11

Реализация реактивных паттернов проектирования

В этой главе мы изучим одну из важнейших возможностей фреймворка Spring 5 – программирование на основе реактивных паттернов. Во фреймворке Spring 5 эта новая возможность появилась вместе с реактивным веб-модулем, который мы обсудим в данной главе. Но сначала взглянем на реактивные паттерны. Что такое реактивный паттерн и почему они сейчас становятся все более популярными? Я начну наше обсуждение реактивных паттернов со следующего заявления Сатья Наделлы, генерального исполнительного директора корпорации Microsoft:

Каждое ныне существующее предприятие — это в какой-то мере компания-разработчик ПО, цифровая компания.

Мы рассмотрим в этой главе следующие вопросы.

- ❑ Почему именно реактивные паттерны.
- ❑ Принципы работы реактивных паттернов.
- ❑ Блокирующие вызовы.
- ❑ Неблокирующие вызовы.
- ❑ Контроль обратного потока данных.
- ❑ Реализация паттерна «Реактивность» с помощью фреймворка Spring.
- ❑ Реактивный веб-модуль фреймворка Spring.

Изменение требований к приложениям с течением времени

Еще 10–15 лет назад пользователей Интернета было очень мало и интернет-порталов для конечных пользователей было намного меньше по сравнению с нынешним днем. Сегодня невозможно представить себе жизнь без компьютера или онлайн-систем. Если коротко, мы стали очень сильно зависимы от компьютеров и использования онлайн-вычислений как для личных, так и для коммерческих

целей. Все бизнес-модели сегодня смещаются в сторону использования вычислительных машин. Премьер-министр Индии, г-н Нарендра Дамодардас Модии, запустил кампанию «Цифровая Индия», чтобы обеспечить доступность для населения коммунальных услуг через Интернет посредством усовершенствования онлайн-инфраструктуры, повышения распространенности доступа в Интернет и расширения возможностей страны в сфере цифровых технологий.

Все это означает взрывной рост количества интернет-пользователей. В отчете компании Ericsson относительно мобильной связи говорится:

Мы ожидаем, что Интернет вещей (IoT) станет в 2018 году самой массовой категорией подключенных к Интернету устройств, обогнав мобильные телефоны.

Количество пользователей мобильного Интернета растет с невероятной скоростью, и нет никаких признаков замедления этого роста в ближайшее время. В этой отрасли по определению серверная сторона ПО должна быть способна обслуживать миллионы подключенных устройств одновременно. В следующей таблице сравниваются требования к инфраструктуре и ПО сегодня и десять лет назад.

Требование	Сегодня	Десять лет назад
Серверные узлы	Необходимо более 1000 узлов	Десяти узлов было вполне достаточно
Время отклика	Обслуживание запросов и отправка ответа должна происходить в течение миллисекунд	Ответ приходил в течение нескольких секунд
Время простоя по причине технического обслуживания	В настоящее время ожидается нулевое время простоя по причине технического обслуживания	Аппаратное обеспечение простаивало часами из-за технического обслуживания
Объем данных	Объемы данных для современных приложений уже выросли с терабайтов до петабайтов	Объемы данных измерялись в гигабайтах

В таблице вы можете видеть различия в требуемых ресурсах. Требования возросли, поскольку теперь пользователи ждут немедленного ответа в течение долей секунды. В то же время сложность задач, которые ставятся компьютерам, тоже возросла. Эти задачи не просто вычисления в математическом смысле слова, но требуют также извлечения данных для ответов из огромных массивов данных. Так что теперь приходится делать упор на производительность систем, проектируя отдельные компьютеры с многоядерными CPU, возможно устанавливаемые в серверы с несколькими разъемами. Прежде всего нам хотелось бы сделать систему отзывчивой. Это первый из отличительных признаков реактивности. Эта глава научит вас создавать отзывчивые системы при любой переменной нагрузке,

частичных перебоях в обслуживании, сбоях программ и т. д. В настоящее время системы обычно для эффективной выдачи запросов распределены по различным узлам.

Паттерн «Реактивность»

Сегодня современные приложения должны быть устойчивее к ошибкам, отказоустойчивее, гибче и лучше приспособлены к требованиям организаций, поскольку за последние несколько лет требования к приложениям изменились радикально. Как мы видели в вышеприведенной таблице, у большого приложения 10–15 лет назад было десять серверных узлов, время отклика (выдачи результатов запроса) измерялось в секундах, для техобслуживания и развертывания требовалось несколько часов простоя оборудования, а объемы данных измерялись в гигабайтах. Но сегодня для приложения требуются тысячи серверных узлов, поскольку к нему обращаются через различные информационные каналы, например с помощью мобильных устройств. Отклик сервера ожидается в течение миллисекунд, а время простоя для техобслуживания и развертывания близко к нулю. Объемы данных выросли до терабайтов, а затем и петабайтов.

Системы десятилетней давности не удовлетворяют требованиям современных приложений; нам нужна система, которая могла бы удовлетворить всем требованиям пользователей или на уровне приложения, или на уровне системы, а значит, нам нужна отзывчивая система. Отзывчивость — одно из свойств паттерна «Реактивность». Итак, нам нужна отзывчивая, отказоустойчивая, адаптивная и ориентированная на работу с сообщениями система. Такие системы известны под названием *реактивных*. Они гибки, слабо сцеплены и легко масштабируются.

Система должна корректно реагировать на сбои, оставаясь доступной, то есть должна быть отказоустойчивой, а также корректно реагировать на различные изменения нагрузки, не перегружаясь при этом чрезмерно. Система должна реагировать на события — быть ориентированной на обработку событий или обмен сообщениями. Если все эти свойства присущи системе, то ее можно считать отзывчивой, то есть если система реагирует на действия пользователей — она отзывчивая. Для создания отзывчивой системы необходимо сосредоточить внимание на уровне системы и уровне приложения. Рассмотрим сначала все характерные особенности реактивности.

Отличительные признаки паттерна «Реактивность»

Основные принципы паттерна «Реактивность»:

- *отзывчивость (responsive)* — цель любого современного приложения;
- *отказоустойчивость (resilient)* — требуется для отзывчивости приложения;
- *масштабируемость (scalable)* — также требуется для отзывчивости приложения; без отказоустойчивости и масштабируемости невозможно достичь отзывчивости;

- *ориентированность на обмен сообщениями (message-driven)* — ориентированная на обработку сообщений архитектура лежит в основе всякого масштабируемого и отказоустойчивого приложения и в конечном счете делает систему отзывчивой. Ориентированность на обмен сообщениями основывается или на ориентированности на обработку событий, или на акторной модели программирования.

Вышеперечисленные пункты — основные принципы паттерна «Реактивность». Рассмотрим каждый из них подробнее и разберемся, почему все они должны использоваться одновременно при создании отзывчивой системы для современного приложения с качественным ПО, способного обрабатывать миллионы параллельных запросов за миллисекунды без каких-либо сбоев. Взглянем сначала на следующую схему (рис. 11.1).

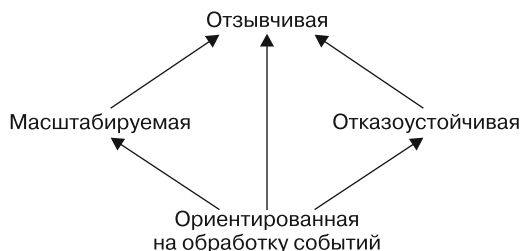


Рис. 11.1. Взаимосвязанность качеств системы

Как вы можете видеть на схеме, для того чтобы сделать систему отзывчивой, необходимы масштабируемость и отказоустойчивость. А чтобы сделать систему масштабируемой и отказоустойчивой, нужна ориентированная на обработку событий или обмен сообщениями архитектура приложения. В конечном итоге эти принципы — масштабируемость, отказоустойчивость и ориентированная на обработку событий архитектура — делают систему отзывчивой относительно клиента. Рассмотрим эти свойства подробнее.

ОТЗЫВЧИВОСТЬ

Если о системе или приложении говорят, что она (оно) отзывчивая (-ое) (responsive), значит это приложение или система быстро отвечают на запросы всех пользователей при любых условиях, как хороших, так и плохих. А это гарантирует, что пользователь всегда будет доволен работой с системой.

Отзывчивость необходима системе для удобства использования и сопровождения. Отзывчивость системы означает, что сбои системы — произошедшие по причине отказа внешней системы или всплеска трафика — оперативно обнаруживаются и эффективно устраняются за короткое время, так что пользователи этого даже не замечают. У конечных пользователей должна быть возможность взаимодействовать с системой, неизменно получая отклик достаточно быстро. Пользователь не должен страдать от каких-либо отказов во время взаимодействия

с системой, которая должна демонстрировать ему постоянный уровень качества обслуживания. Такое единообразное поведение решает проблему отказов и внушает конечному пользователю доверие к системе. Отзывчивой систему делают скорость работы и положительный опыт взаимодействия пользователя с ней при различных условиях. Это зависит еще от двух отличительных признаков реактивного приложения или системы, а именно от отказоустойчивости и масштабируемости. Еще один отличительный признак — ориентированная на обработку событий или сообщений архитектура — формирует основной фундамент для отзывчивой системы (рис. 11.2).



Рис. 11.2. Отзывчивая система

Как вы можете видеть, в основе отзывчивости системы лежат ее отказоустойчивость и масштабируемость, а в их основе — ориентированная на обработку событий архитектура. Рассмотрим остальные отличительные признаки реактивного приложения.

Отказоустойчивость

При проектировании и разработке приложения рассматривают все условия: как благоприятные, так и нет. Если учитывать возможность только благоприятных условий, то существует большая вероятность создать систему, которая проработает лишь несколько дней. Серьезный сбой приложения приведет к простоя и потере данных, а значит, нанесет ущерб репутации приложения на рынке.

Поэтому следует учитывать все возможные условия, чтобы гарантировать отзывчивость приложения при любом сценарии. Подобные системы/приложения известны как отказоустойчивые.

Система должна быть отказоустойчивой, чтобы быть отзывчивой. Если система не отказоустойчива, в случае сбоя она перестанет быть отзывчивой. А значит,

система должна быть отзывчивой и в случае сбоя. Сбой может произойти в любом компоненте приложения/системы. Поэтому компоненты системы должны быть изолированы друг от друга, чтобы при сбое компонента восстановление не влияло на систему в целом. Восстановление после сбоя отдельных компонентов достигается за счет репликации. Отказоустойчивая система должна обладать возможностями репликации, обособленности, изоляции и делегирования (рис. 11.3).

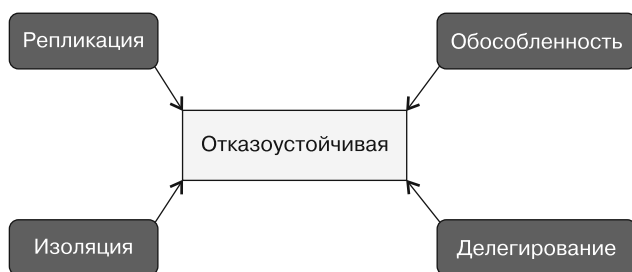


Рис. 11.3. Отличительные признаки отказоустойчивого приложения/системы

Как вы видите на схеме, отзывчивость достигается при сочетании репликации, обособленности, изоляции и делегирования. Рассмотрим каждое из этих свойств по отдельности.

- ❑ *Репликация.* Обеспечивает при необходимости высокую доступность в момент отказа какого-либо из компонентов.
- ❑ *Изоляция.* Смысл этого свойства в том, что сбой любого компонента должен изолироваться путем расщепления компонентов в как можно большей степени. Изоляция необходима для самовосстановления системы. Если в системе реализована изоляция, можно легко оценить производительность каждого из компонентов и выяснить, сколько он использует памяти и ресурсов CPU. Более того, отказ одного компонента не повлияет на отзывчивость системы или приложения в целом.
- ❑ *Обособленность.* В результате расщепления сбой оказывается обособлен от остальной системы. Это помогает избежать отказа системы в целом.
- ❑ *Делегирование.* Восстановление каждого компонента после сбоя делегируется другому компоненту. Это возможно только в том случае, если система пригодна для компоновки.

Современные приложения не только зависят от внутренней инфраструктуры, но и тесно интегрированы с другими веб-сервисами через сетевые протоколы. Поэтому наше приложение должно быть отзывчивым по своей сути, чтобы оставаться таковым при множестве разнообразных сценариев реального мира, в зачастую диаметрально противоположных условиях. Приложения должны быть отказоустойчивыми не только на уровне приложения, но и на уровне системы.

Масштабируемость

Отказоустойчивость и масштабируемость вместе делают систему согласованно отзывчивой. Возможности масштабируемой (адаптивной) системы можно легко наращивать в случае переменной нагрузки. Реактивная система масштабируется по мере необходимости путем увеличения/уменьшения ресурсов, выделяемых для обслуживания входных данных. Такая система поддерживает несколько алгоритмов масштабирования за счет предоставления соответствующих показателей производительности в режиме реального времени. Достичь масштабируемости можно также посредством использования экономичного программного обеспечения и дешевого серийного аппаратного обеспечения (например, с помощью облачных решений).

Приложение является масштабируемым, если оно допускает следующие виды расширения возможностей в соответствии с его загрузкой:

- ❑ *вертикальное масштабирование*, использующее возможности параллелизма многоядерных систем;
- ❑ *горизонтальное масштабирование*, использующее возможности мультисерверных узлов. При этом важны отказоустойчивость и прозрачность расположения.

Для масштабирования очень важно минимизировать разделяемое изменяемое состояние.



Адаптивность и масштабируемость — одно и то же! Масштабируемость относится к эффективному использованию уже имеющихся ресурсов, а адаптивность — к добавлению в приложение новых ресурсов по мере необходимости, при изменении потребностей системы. Так что в конечном итоге систему можно сделать отзывчивой любым из этих способов — или с помощью существующих ресурсов системы, или путем добавления в нее дополнительных ресурсов.

Рассмотрим последний из элементов фундамента паттерна «Реактивность», а именно ориентированную на обмен сообщениями архитектуру.

Ориентированная на обмен сообщениями архитектура

Ориентированная на обмен сообщениями архитектура — основа отзывчивого приложения. Ориентированное на обмен сообщениями приложение может быть ориентированным на обработку событий или основываться на акторной модели, а также представлять собой сочетание обеих архитектур: событийной и акторной.

В ориентированной на обработку событий архитектуре главную роль играют события и наблюдатели событий. События происходят, но не направляются на конкретный адрес; прослушиватели слушают на предмет этих событий и предпринимают на основе полученной информации какие-либо действия. В ориентированной же на обмен сообщениями архитектуре сообщения имеют конкретную точку назначения (рис. 11.4).

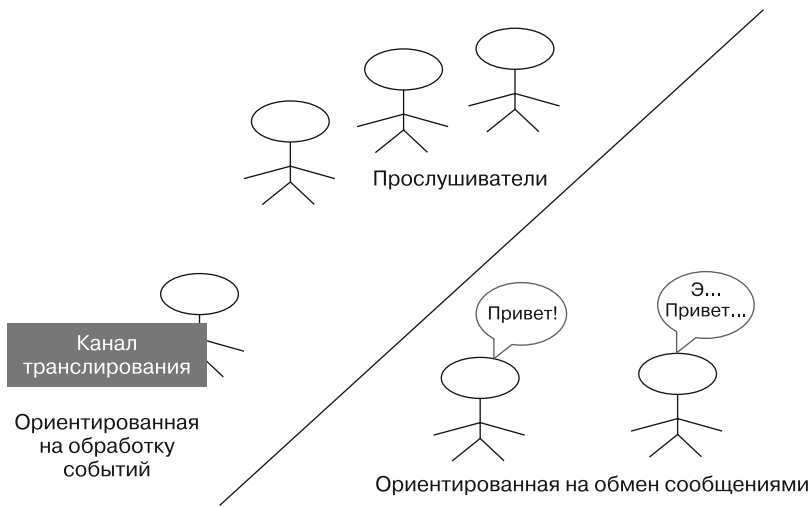


Рис. 11.4. Схема, иллюстрирующая обе архитектуры

Как видите, в ориентированной на обработку событий архитектуре прослушиватели слушают на предмет возникновения события. Но в ориентированной на сообщения архитектуре у каждого сгенерированного сообщения есть имеющий адрес получатель и конкретная цель.

Асинхронная ориентированная на обмен сообщениями архитектура играет роль фундамента реактивной системы, устанавливая ограничения на компоненты. Она обеспечивает слабое сцепление, изоляцию и прозрачность расположения. Изоляция компонентов полностью зависит от слабого сцепления между ними. А изоляция вкупе со слабым сцеплением формирует фундамент отказоустойчивости и адаптивности.

В большой системе обычно много компонентов. Эти компоненты или представляют собой меньшие приложения, или могут обладать свойствами реактивности. А значит, чтобы система стала пригодной для компоновки, принципы реактивного проектирования должны использоваться на всех ее уровнях.

Традиционно большие системы состояли из нескольких потоков выполнения, взаимодействующих через разделяемое синхронизируемое состояние. Они склонны к сильному сцеплению компонентов и малопригодны для компоновки, а также склонны к блокировкам ресурсов. Но в настоящее время все большие системы состоят из слабо сцепленных обработчиков событий. А события можно обрабатывать асинхронно без блокирования.

Изучим блокирующие и неблокирующие модели программирования.

В самом упрощенном виде понятие «реактивное программирование» относится к неблокирующим асинхронным приложениям, ориентированным на обработку событий и требующим лишь небольшого количества потоков выполнения, чтобы масштабироваться вертикально, а не горизонтально.

Блокирующие вызовы

В системе вызов может удерживать ресурсы, которые в то же время требуются другим вызовам. Эти ресурсы освобождаются тогда, когда первый из вышеуказанных вызовов прекращает их использовать.

Опишу это же на техническом языке: фактически блокировка вызова означает, что часть операций приложения или системы будет выполняться дольше. Например, это могут быть операции файлового ввода/вывода или обращения к базе данных посредством блокирующего драйвера.

На рис. 11.5 вы можете видеть блокирующие операции, показанные серым цветом, — вызов пользователем сервлета для извлечения данных с последующим обращением к JDBC и соединением с сервером базы данных. Текущий поток выполнения ждет поступления результирующего набора данных от сервера БД. При задержке поступления ответа от сервера БД это время ожидания растет. Это значит, что выполнение этого потока зависит от задержки сервера БД.

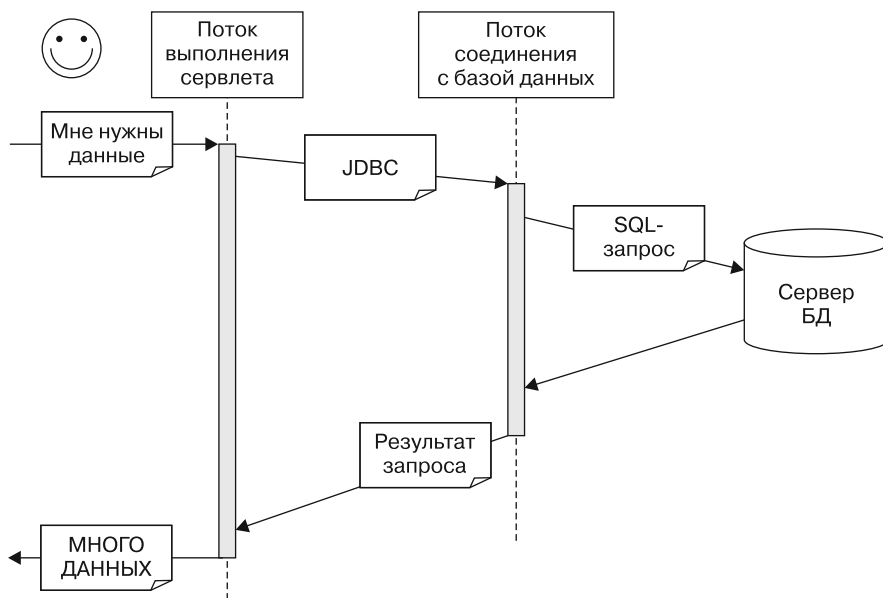


Рис. 11.5. Схема блокирующих вызовов для JDBC-операции в системе

Посмотрим теперь, как сделать эту операцию неблокирующей.

Неблокирующие вызовы

Блокирующее выполнение программы означает, что поток выполнения конкурирует за ресурс, а не ждет его освобождения. Неплокирующий API для ресурсов предоставляет возможность обращения к ресурсам без ожидания завершения

блокирующего вызова, например обращения к базе данных или сетевого вызова. Если ресурсы недоступны на момент вызова, выполняются другие задания, вместо того чтобы ждать освобождения заблокированных ресурсов. Система получает уведомление при освобождении заблокированных ресурсов.

Как можно видеть на рис. 11.6, для выполнения потока не требуется дожидаться возвращения результата запроса от сервера БД. Поток выполняет и соединение с базой данных, и SQL-оператор для сервера БД. При задержке на стороне сервера БД поток переходит к следующему заданию, а не блокирует выполнение до момента освобождения ресурса.

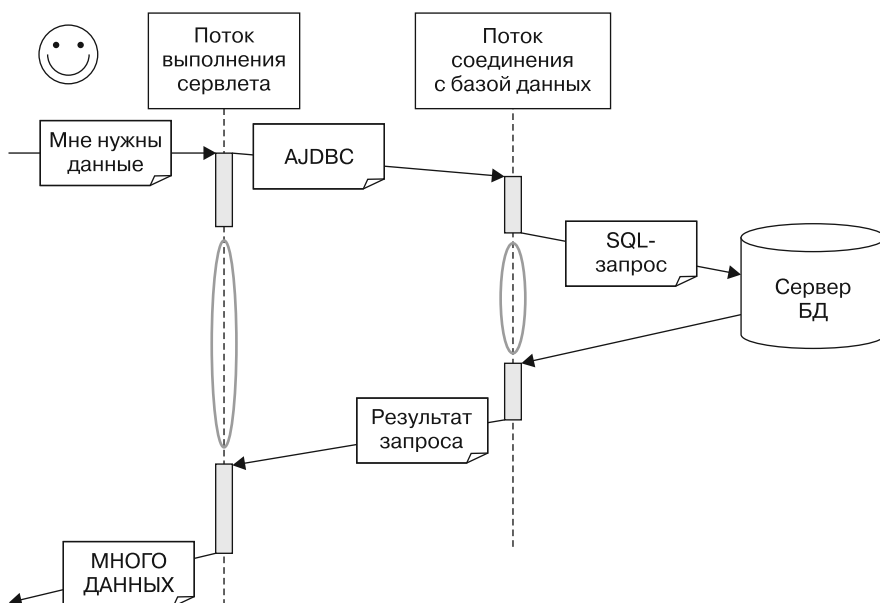


Рис. 11.6. JDBC-соединение с целью обращения к данным без блокировки выполнения потока

Контроль обратного потока данных

Реактивное приложение никогда не сдастся в условиях чрезмерной нагрузки. Контроль обратного потока данных (back-pressure) — ключевая особенность реактивного приложения. С помощью этого механизма гарантируется, что реактивное приложение не завалит потребителей слишком большим объемом данных. Он проверяет аспекты реактивного приложения, а также время отклика системы под произвольной нагрузкой.

Механизм контроля обратного потока данных гарантирует отказоустойчивость системы под нагрузкой. В условиях контроля обратного потока система обеспечивает масштабируемость, подключая дополнительные ресурсы для перераспределения нагрузки.

До сих пор мы рассматривали принципы паттерна «Реактивность», обязательные для обеспечения отзывчивости системы в нормальных или затрудненных условиях. В следующем разделе мы посмотрим на реализацию парадигмы реактивного программирования фреймворком Spring 5.0.

Реализация реактивности с помощью фреймворка Spring 5.0

Наиболее яркая возможность последней версии фреймворка Spring — новый веб-фреймворк с реактивным стеком технологий. Реактивность — та новая возможность, которая открывает перед нами будущее. Популярность этой сферы технологий растет с каждым днем, именно поэтому во фреймворк Spring 5.0 была включена возможность реактивного программирования. Благодаря этому дополнению последняя версия фреймворка Spring подходит для обработки циклов с ожиданием событий, в результате чего масштабирование возможно при небольшом количестве потоков выполнения.

Фреймворк Spring 5.0 реализует паттерн реактивного программирования с помощью проекта Reactor. Этот проект представляет собой реализацию расширения базовых возможностей Reactive Streams. С помощью Reactive Streams Twitter был реализован в виде реактивной системы.

Reactive Streams. Стандарт Reactive Streams представляет собой протокол (набор правил) асинхронной потоковой обработки с неблокирующим контролем обратного потока данных. Этот стандарт включен также в Java 9 в виде класса `java.util.concurrent.Flow`. Reactive Streams состоит из четырех простых Java-интерфейсов: `Publisher`, `Subscriber`, `Subscription` и `Processor`. Но основная задача Reactive Streams заключается в контроле обратного потока данных. Как уже обсуждалось ранее, контроль обратного потока данных представляет собой процесс, при котором у получателя есть возможность запрашивать у источника объем отправляемых данных.

Для добавления Reactive Streams в приложение можно использовать следующую Maven-зависимость:

```
<dependency>
  <groupId>org.reactivestreams</groupId>
  <artifactId>reactive-streams</artifactId>
  <version>1.0.1</version>
</dependency>
<dependency>
  <groupId>org.reactivestreams</groupId>
  <artifactId>reactive-streams-tck</artifactId>
  <version>1.0.1</version>
</dependency>
```

Предыдущий код Maven-зависимости добавляет необходимые для Reactive Streams библиотеки в приложение. В следующем разделе мы увидим, как фреймворк Spring реализует Reactive Streams в веб-модуле Spring и фреймворке MVC Spring.

Реактивный веб-модуль Spring

Во фреймворке Spring 5.0 появился новый модуль для реактивного программирования — `spring-web-reactive`. В его основе лежит Reactive Streams. По сути, он использует модуль MVC Spring в сочетании с реактивным программированием, так что вы можете использовать модуль MVC Spring для своего веб-приложения — отдельно или вместе с модулем `spring-web-reactive`.

Этот новый модуль фреймворка Spring 5.0 содержит поддержку реактивно-функциональной модели веб-программирования. Он же поддерживает модель программирования на основе аннотаций. Модуль `spring-web-reactive` поддерживает реактивные HTTP- и WebSocket-клиенты для вызова реактивных серверных приложений. Он же обеспечивает возможность реактивного HTTP-соединения реактивного веб-клиента с реактивным веб-приложением.

Как можно видеть на рис. 11.7, параллельно существует два модуля: один — для обычного фреймворка MVC Spring, а второй — для реактивных веб-модулей Spring. В левой части схемы находятся относящиеся к MVC Spring компоненты, такие как контроллеры `@MVC`, модуль `spring-web-mvc`, модуль *API сервлетов*, а также *контейнеры сервлетов*. В правой части — компоненты, относящиеся к модулю `spring-web-reactive`, например *маршрутизирующие функции*, сам модуль `spring-web-reactive`, *HTTP/Reactive Streams*, реактивная версия Tomcat и т. д.

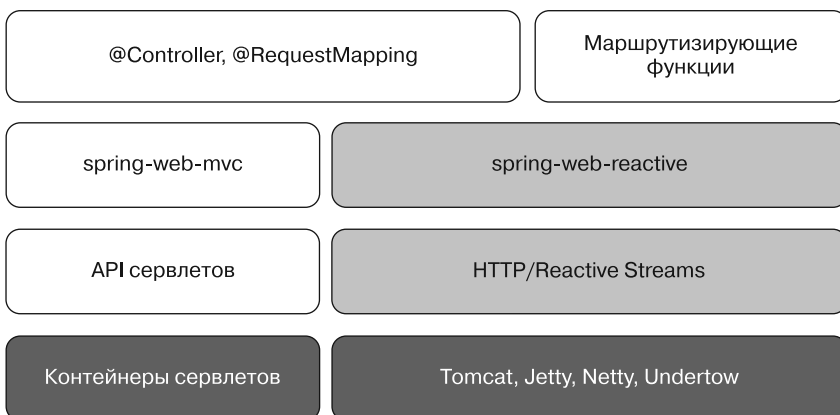


Рис. 11.7. Модуль `spring-web-reactive` и его компоненты, обеспечивающие реактивное поведение веб-приложений Spring

На предыдущей схеме обратите внимание на размещение модулей. У каждого традиционного модуля MVC Spring на конкретном уровне есть аналог для `spring-web-reactive`. Сравним эти модули.

- ❑ В реактивных веб-модулях Spring маршрутизирующие функции подобны контроллерам `@MVC` в модулях Spring MVC, например с аннотациями `@Controller`, `@RestController` и `@RequestMapping`.
- ❑ Модуль `spring-web-reactive` аналогичен модулю `spring-web-mvc`.
- ❑ В традиционном фреймворке Spring MVC для объектов `HttpServletRequest` и `HttpServletResponse` в контейнере сервлетов используется API сервлетов. Но во фреймворке `spring-web-reactive` используются HTTP/Reactive Streams, создающие объекты `HttpServletRequest` и `HttpServletResponse` при реактивной поддержке сервера Tomcat.
- ❑ Контейнер сервлетов можно использовать и для традиционного фреймворка MVC Spring, но для приложений на основе `spring-web-reactive` требуется веб-сервер с поддержкой реактивности. Spring поддерживает веб-серверы Tomcat, Jetty, Netty и Undertow.

Из главы 10 вы узнали, как можно реализовать веб-приложение с помощью модуля MVC Spring. Теперь взглянем, как реализовать реактивное веб-приложение с помощью модуля `spring-web-reactive`.

Реализация реактивного веб-приложения на стороне сервера

Реактивные веб-модули Spring поддерживают обе модели программирования: как основанную на аннотациях, так и функциональную. Посмотрим, как эти модели работают на стороне сервера.

- ❑ *Модель программирования на основе аннотаций.* В ее основе лежат аннотации MVC, такие как `@Controller`, `@RestController`, `@RequestMapping` и многие другие. Фреймворк MVC Spring поддерживает использование аннотаций для программирования веб-приложений на стороне сервера.
- ❑ *Функциональная модель программирования.* Это новая парадигма программирования, поддерживаемая фреймворком Spring 5. Она основывается на маршрутизации и обработке с помощью лямбда-выражений языка Java 8. Язык Scala также поддерживает парадигму функционального программирования.

Для создания реактивного веб-приложения на основе Spring Boot понадобится добавить следующие Maven-зависимости:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.M3</version>
  <relativePath/>
```

```

<!-- lookup parent from repository -->
</parent>

<properties>
  <project.build.sourceEncoding>UTF-8
</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8
</project.reporting.outputEncoding>
  <java.version>1.8</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

```

Как вы можете видеть из вышеприведенного файла конфигурации Maven для зависимостей, мы добавили в приложение зависимости `spring-boot-starter-webflux` и `reactor-test`.

Теперь создадим реактивное веб-приложение на основе модели программирования с использованием аннотаций.

Модель программирования на основе аннотаций

Для нее можно воспользоваться теми же аннотациями, которые мы использовали в главе 10. Такие аннотации, как `@Controller` и `@RestController`, поддерживаются и в реактивных приложениях. До этого момента нет разницы между традиционными приложениями MVC Spring и веб-приложениями Spring на основе реактивного модуля. Различия начнутся после того, как вы объявите конфигурации аннотации `@Controller`, то есть при переходе к работе с внутренними механизмами MVC Spring, начиная с компонентов `HandlerMapping` и `HandlerAdapter`.

Основное различие между традиционным модулем MVC Spring и модулем `spring-web-reactive` состоит в механизме обработки запросов. Нереактивный MVC Spring обрабатывает запросы с помощью блокирующих интерфейсов API сервлетов `HttpServletRequest` и `HttpServletResponse`, а реактивный веб-фреймворк не является блокирующим и работает с реактивными интерфейсами `ServerHttpRequest` и `ServerHttpResponse` вместо `HttpServletRequest` и `HttpServletResponse`.

Рассмотрим следующий пример с реактивным контроллером:

```
package com.packt.patterninspring.chapter11.reactivewebapp.controller;

import org.reactivestreams.Publisher;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.packt.patterninspring.chapter11.reactivewebapp.model.Account;
import com.packt.patterninspring.chapter11.reactivewebapp.repository.AccountRepository;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@RestController
public class AccountController {
    @Autowired
    private AccountRepository repository;
    @GetMapping(value = "/account")
    public Flux<Account> findAll() {
        return repository.findAll().map(a -> new
            Account(a.getId(), a.getName(),
                a.getBalance(), a.getBranch()));
    }
    @GetMapping(value = "/account/{id}")
    public Mono<Account> findById(@PathVariable("id") Long id) {
        return repository.findById(id)
            .map(a -> new Account(a.getId(), a.getName(), a.getBalance(),
                a.getBranch()));
    }
    @PostMapping("/account")
    public Mono<Account> create(@RequestBody
        Publisher<Account> accountStream) {
        return repository
            .save(Mono.from(accountStream)
                .map(a -> new Account(a.getId(), a.getName(), a.getBalance(),
                    a.getBranch())))
            .map(a -> new Account(a.getId(), a.getName(), a.getBalance(),
                a.getBranch()));
    }
}
```

Как можно видеть из этого кода контроллера из файла `AccountController.java`, я использовал те же аннотации MVC Spring, например `@RestController`, для объявления класса контроллера, а `@GetMapping` и `@PostMapping` — для создания методов-обработчиков запросов GET и POST соответственно.

Обсудим подробнее возвращаемые типы данных методов-обработчиков. Эти методы возвращают значения в виде объектов типов *Mono* и *Flux* — реактивных по-

токов данных из фреймворка Reactor. Кроме того, метод-обработчик принимает на входе тело запроса с помощью объекта типа `Publisher`.

Reactor — Java-фреймворк, созданный командой разработчиков программного обеспечения с открытым исходным кодом Pivotal. Он основан непосредственно на Reactive Streams, так что необходимости в промежуточном нет. Проект Reactor IO предоставляет адаптеры для таких низкоуровневых сетевых средств, как Netty или Aeron. Reactor относится к библиотекам четвертого поколения в соответствии с классификацией поколений реактивности Дэвида Кэрнока.

Взглянем на тот же класс контроллера при использовании для обработки запросов функциональной модели программирования.

Функциональная модель программирования

Функциональная модель программирования использует API с такими функциональными интерфейсами, как `RouterFunction` и `HandlerFunction`. В ней применяется программирование в лямбда-стиле языка Java 8 с маршрутизацией и обработкой запросов вместо аннотаций MVC Spring. Это простые, но обладающие большими возможностями стандартные блоки, из которых создаются веб-приложения.

Вот пример функциональной обработки запроса:

```
package com.packt.patterninspring.chapter11.web.reactive.function;

import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.BodyInserters.fromObject;

import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;

import com.packt.patterninspring.chapter11.web.reactive.model.Account;
import com.packt.patterninspring.chapter11.web.reactive.repository.AccountRepository;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public class AccountHandler {

    private final AccountRepository repository;

    public AccountHandler(AccountRepository repository) {
        this.repository = repository;
    }

    public Mono<ServerResponse> findById(ServerRequest request) {
        Long accountId = Long.valueOf(request.pathVariable("id"));
    }
}
```

```

Mono<ServerResponse> notFound =
    ServerResponse.notFound().build();
Mono<Account> accountMono =
    this.repository.findById(accountId);
return accountMono
    .flatMap(account -> ServerResponse.ok().contentType(
        APPLICATION_JSON).body(
            fromObject(account)))
    .switchIfEmpty(notFound);
}
public Mono<ServerResponse> findAll(ServerRequest request) {
    Flux<Account> accounts = this.repository.findAll();
    return ServerResponse.ok().contentType(
        APPLICATION_JSON).body(accounts, Account.class);
}
public Mono<ServerResponse> create(ServerRequest request) {
    Mono<Account> account = request.bodyToMono(Account.class);
    return ServerResponse.ok().build(this.
        repository.save(account));
}
}

```

В предыдущем коде файл класса `AccountHandler.java` основан на функциональной реактивной модели программирования. В нем я воспользовался фреймворком `Reactor` для обработки запроса. Для обработки запросов и генерации ответов на них используются два функциональных интерфейса, `ServerRequest` и `ServerResponse`.

Взглянем на классы репозитория для этого приложения. Нижеприведенные классы `AccountRepository` и `AccountRepositoryImpl` одинаковы для обеих моделей программирования — как основанной на аннотациях, так функциональной.

Создадим класс интерфейса `AccountRepository.java` в следующем виде:

```

package com.packt.patterninspring.chapter11.
    reactivewebapp.repository;
import com.packt.patterninspring.chapter11.
    reactivewebapp.model.Account;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public interface AccountRepository {
    Mono<Account> findById(Long id);
    Flux<Account> findAll();
    Mono<Void> save(Mono<Account> account);
}

```

Предыдущий код представляет собой интерфейс, реализуем его в виде класса `AccountRepositoryImpl`:

```

package com.packt.patterninspring.chapter11.
    web.reactive.repository;

import java.util.Map;

```



```

import java.util.concurrent.ConcurrentHashMap;

import org.springframework.stereotype.Repository;

import com.packt.patterninspring.chapter11.web.
    reactive.model.Account;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@Repository
public class AccountRepositoryImpl implements AccountRepository {
    private final Map<Long, Account> accountMap = new
        ConcurrentHashMap<>();
    public AccountRepositoryImpl() {
        this.accountMap.put(10001, new Account(10001,
            "Dinesh Rajput", 500001,
            "Sector-1"));
        this.accountMap.put(20001, new Account(20001,
            "Anamika Rajput", 600001,
            "Sector-2"));
        this.accountMap.put(30001, new Account(30001,
            "Arnav Rajput", 700001,
            "Sector-3"));
        this.accountMap.put(40001, new Account(40001,
            "Adesh Rajput", 800001,
            "Sector-4"));
    }

    @Override
    public Mono<Account> findById(Long id) {
        return Mono.justOrEmpty(this.accountMap.get(id));
    }

    @Override
    public Flux<Account> findAll() {
        return Flux.fromIterable(this.accountMap.values());
    }

    @Override
    public Mono<Void> save(Mono<Account> account) {
        return account.doOnNext(a -> {
            accountMap.put(a.getId(), a);
            System.out.format("Saved %s with id %d\n", a, a.getId());
        }).thenEmpty(Mono.empty());
        // Возвращаем accountMono;
    }
}

```

Как видно из предыдущего кода, мы создали класс `AccountRepository`. У него только три метода: `findById()`, `findAll()` и `save()`. Мы реализовали их в соответствии

с бизнес-требованиями. В этом классе репозитория я специально использовал реактивные типы Flux и Mono, чтобы сделать приложение реактивным.

Создадим сервер для функциональной модели программирования. При программировании на основе аннотаций мы использовали для развертывания веб-приложения простой контейнер Tomcat. Но в случае функциональной модели программирования нам придется создать класс Server для запуска сервера Tomcat или Reactor:

```
package com.packt.patterninspring.chapter11.web.reactive.function;

// Здесь располагаются импорты

public class Server {

    public static final String HOST = "localhost";
    public static final int TOMCAT_PORT = 8080;
    public static final int REACTOR_PORT = 8181;
    // Здесь располагается метод main, код которого можно скачать с GITHUB
    public RouterFunction<ServerResponse> routingFunction() {
        AccountRepository repository = new AccountRepositoryImpl();
        AccountHandler handler = new AccountHandler(repository);

        return nest(path("/account"), nest(accept(APPLICATION_JSON),
            route(GET("/{id}"), handler::findById)
                .andRoute(method(HttpMethod.GET), handler::findAll)
                .andRoute(POST("/").and(contentType
                    (APPLICATION_JSON)), handler::create)));
    }

    public void startReactorServer() throws InterruptedException {
        RouterFunction<ServerResponse> route = routingFunction();
        HttpHandler httpHandler = toHttpHandler(route);
        ReactorHttpHandlerAdapter adapter = new
            ReactorHttpHandlerAdapter(httpHandler);
        HttpServer server = HttpServer.create(HOST, REACTOR_PORT);
        server.newHandler(adapter).block();
    }

    public void startTomcatServer() throws LifecycleException {
        RouterFunction<?> route = routingFunction();
        HttpHandler httpHandler = toHttpHandler(route);

        Tomcat tomcatServer = new Tomcat();
        tomcatServer.setHostname(HOST);
        tomcatServer.setPort(TOMCAT_PORT);
        Context rootContext = tomcatServer.addContext("",
            System.getProperty("java.io.tmpdir"));
        ServletHttpHandlerAdapter servlet = new
            ServletHttpHandlerAdapter(httpHandler);
```

```

    Tomcat.addServlet(rootContext, "httpHandlerServlet", servlet);
    rootContext.addServletMapping("/", "httpHandlerServlet");
    tomcatServer.start();
}
}

```

Как вы можете видеть в предыдущем файле класса `Server.java`, я добавил в него оба сервера: и Tomcat, и Reactor. Сервер Tomcat использует порт 8080, а сервер Reactor — 8181.

В классе `Server.java` есть три метода. Первый из них, `routingFunction()`, отвечает за обработку запросов клиентов с помощью класса `AccountHandler`. Он зависит от класса `AccountRepository`. Второй метод, `startReactorServer()`, отвечает за запуск сервера Reactor с помощью класса `ReactorHttpHandlerAdapter` сервера. Чтобы создать отображение обработчика запроса, этот класс принимает объект класса `HttpHandler` на входе конструктора в качестве аргумента. Аналогично третий метод, `startTomcatServer()`, отвечает за запуск сервера Tomcat. Для этой цели он тоже использует объект класса `HttpHandler` через класс-адаптер `ReactorServletHttpHandlerAdapter`.

Этот класс `Server` можно запустить как Java-приложение и получить результаты в браузере, набрав URL `http://localhost:8080/account/` (рис. 11.8).

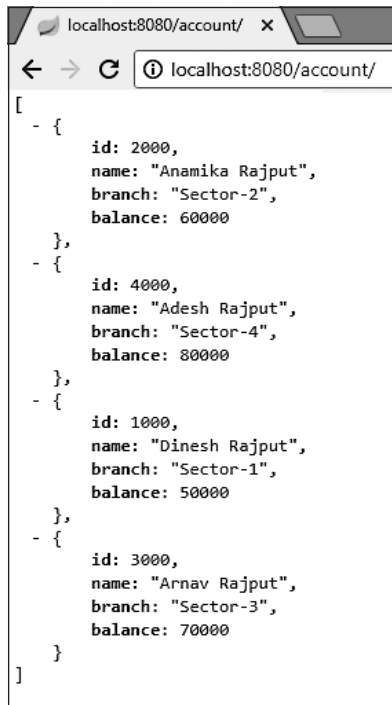


Рис. 11.8. Класс `Server` запущен как Java-приложение

Можно также ввести тот же URL с портом 8181 для использования сервера Reactor, как показано ниже, и получить те же результаты:

```
http://localhost:8181/account/
```

В этом подразделе вы научились создавать реактивные веб-приложения с помощью модуля `spring-web-reactive`. Мы создали веб-приложение с помощью каждой из парадигм программирования: на основе аннотаций и функциональной.

Далее мы обсудим код на стороне клиента, а также обращение клиента к реактивным веб-приложениям.

Реализация реактивного приложения на стороне клиента

Во фреймворке Spring 5 появился функциональный реактивный веб-клиент. Это совершенно не блокирующий реактивный веб-клиент, альтернатива классу `RestTemplate`. Он осуществляет ввод/вывод по сети в форме реактивных объектов `ClientHttpRequest` и `ClientHttpResponse`, создавая тело запроса и ответа в виде `Flux<DataBuffer>` вместо `InputStream` и `OutputStream`.

Посмотрим на код веб-клиента из класса `Client.java`:

```
package com.packt.patterninspring.chapter11.web.reactive.function;

// Здесь располагаются импорты

public class Client {

    private ExchangeFunction exchange = ExchangeFunctions.create(new
    ReactorClientHttpConnector());

    public void findAllAccounts() {
        URI uri = URI.create(String.format("http://%s:%d/account",
        Server.HOST,
        Server.TOMCAT_PORT));
        ClientRequest request = ClientRequest.method(HttpMethod.GET,
        uri).build();

        Flux<Account> account = exchange.exchange(request)
        .flatMapMany(response -> response.bodyToFlux(Account.class));

        Mono<List<Account>> accountList = account.collectList();
        System.out.println(accountList.block());
    }

    public void createAccount() {
        URI uri = URI.create(String.format("http://%s:%d/account",
        Server.HOST,
```

```

Server.TOMCAT_PORT));
Account jack = new Account(50001, "Arnav Rajput", 5000001,
"Sector-5");

ClientRequest request = ClientRequest.method(HttpMethod.POST, uri)
.body(BodyInserters.fromObject(jack)).build();

Mono<ClientResponse> response = exchange.exchange(request);

System.out.println(response.block().statusCode());
}
}

```

Предыдущий класс, `Client.java`, представляет собой класс веб-клиента для сервера `Server.java`. В нем содержится два метода. Первый — `findAllAccounts()`. Он извлекает все счета из репозитория счетов, используя интерфейс `ClientRequest` из пакета `org.springframework.web.reactive.function.client` для создания запроса к URI `http://localhost:8080/account/` с методом запроса `GET`. С помощью интерфейса `ExchangeFunction` из пакета `org.springframework.web.reactive.function.client` он обращается к серверу и извлекает результат в формате `JSON`. Аналогично второй метод, `createAccount()`, создает новый лицевой счет на сервере посредством метода `POST`, используя URI `http://localhost:8080/account/`.

Запустим класс `Client` как Java-приложение и посмотрим на выводимые в консоль результаты (рис. 11.9).

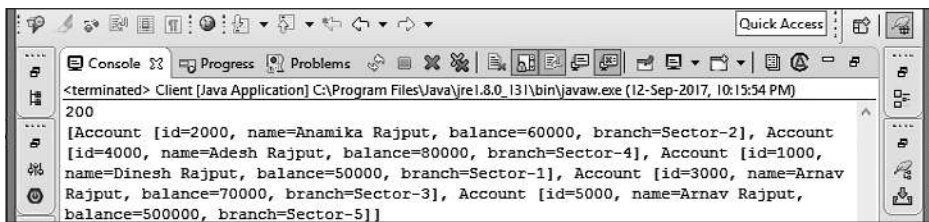


Рис. 11.9. Класс `Client` запущен как Java-приложение

Он создает новую запись и извлекает все пять записей в формате `JSON`-списка.



Класс `AsyncRestTemplate` также поддерживает неблокирующие взаимодействия. Основное отличие состоит в отсутствии в нем поддержки неблокирующей потоковой обработки, например, как в `Twitter`, поскольку по своей сути он все равно основывается на `InputStream` и `OutputStream` и зависит от них.

В следующем разделе мы поговорим о параметрах тел запроса и ответа в реактивных веб-приложениях.

Преобразование типов тела запроса и ответа

В главе 10 мы обсуждали преобразование типов сообщений для тела запроса/ответа либо из Java в JSON, либо из формата JSON в Java-объект и др. Аналогично преобразование типа необходимо и для реактивных веб-приложений. Базовый модуль Spring предоставляет реактивные интерфейсы `Encoder` и `Decoder` для сериализации Flux байтов в типизированные объекты и наоборот.

Рассмотрим следующий пример преобразования типа тела запроса. Разработчикам не нужно принудительно выполнять преобразование типа – фреймворк Spring автоматически преобразует тип при обоих подходах: программировании на основе аннотаций и функциональном программировании.

- ❑ *Счет в виде объекта Account* — объект счета десериализуется перед вызовом контроллера без блокировки.
- ❑ *Счет в виде Mono<Account>* — `AccountController` может использовать тип `Mono` для объявления логики. Объект счета сначала десериализуется, а затем выполняется логика.
- ❑ *Счета в виде Flux<Account>* — `AccountController` может использовать `Flux` в случае сценария с потоковыми входными данными.
- ❑ *Счет в виде Single<Account>* — очень схоже с `Mono`, но тут контроллер использует библиотеку RxJava.
- ❑ *Счет в виде Observable<Account>* — очень похоже на `Flux`, но контроллер осуществляет потоковый ввод данных с помощью библиотеки RxJava.

В предыдущем списке фреймворк Spring использовался для преобразования типов в реактивной модели программирования. Взглянем на следующие возвращаемые типы данных в примере тела ответа.

- ❑ `Account` — сериализует без блокировки заданного объекта `Account`. Предполагает, что метод контроллера синхронный и неблокирующий.
- ❑ `void` — предназначен для модели программирования на основе аннотаций. Обработка запроса завершается с возвратом из метода. Предполагает, что метод контроллера синхронный и неблокирующий.
- ❑ `Mono<Account>` — сериализует без блокировки заданного объекта `Account` при завершении выполнения логики `Mono`.
- ❑ `Mono<Void>` — предполагает, что обработка запроса завершается при завершении выполнения логики `Mono`.
- ❑ `Flux<Account>` — используется при потоковой обработке, возможно, посылаемые сервером события (SSE) зависят от используемого типа контента.
- ❑ `Flux<ServerSentEvent>` — делает возможной потоковую обработку посылаемых сервером событий.
- ❑ `Single<Account>` — то же самое, но с использованием библиотеки RxJava.

- ❑ `Observable<Account>` — то же самое, но с применением типа `Observable` библиотеки RxJava.
- ❑ `Flowable<Account>` — то же самое, но с использованием типа `Flowable` библиотеки RxJava 2.

В этом списке приведены возвращаемые типы данных методов-обработчиков. Преобразования типов в реактивной модели программирования осуществляет фреймворк Spring.

Резюме

Из этой главы вы узнали про паттерн «Реактивность» и принципы его работы. Это не новинка в программировании, но паттерн прекрасно удовлетворяет требованиям современных приложений.

Реактивное программирование основывается на четырех принципах: отзывчивости, отказоустойчивости, адаптивности и ориентированной на обмене сообщениями архитектуре. Отзывчивость означает, что система должна быстро реагировать на действия пользователя при любых условиях, как необычных, так и обычных.

Фреймворк Spring 5 поддерживает модель реактивного программирования посредством использования фреймворка Reactor и реактивных потоков данных. В Spring появился новый реактивный веб-модуль — модуль `spring-web-reactive`. Он воплощает реактивный подход к программированию веб-приложений посредством или аннотаций MVC Spring, таких как `@Controller`, `@RestController` и `@RequestMapping`, или функционального программирования, с помощью лямбда-выражений Java 8.

В этой главе мы создали веб-приложение с помощью модуля `spring-web-reactive`. Код этого приложения вы можете найти на GitHub.

12

Реализация конкурентных паттернов

В главе 11 мы обсуждали реактивные паттерны проектирования и удовлетворение с их помощью требований современных приложений. Во фреймворке Spring 5 появились модули реактивного программирования для веб-приложений. В этой главе мы рассмотрим несколько конкурентных паттернов проектирования и решение с их помощью частых проблем многопоточных приложений. Реактивные модули фреймворка Spring 5 также решают подобные проблемы.

Если вы разработчик программного обеспечения или готовитесь стать таковым, то вам наверняка знаком термин «конкурентность» (concurrency). В геометрии параллельными кругами или фигурами называют круги или фигуры с общим центром. Их размеры могут различаться, но центры или срединные точки совпадают.

В разработке ПО также есть аналогичные термины. Термин «конкурентное программирование» в сфере техники или программирования означает способность программы производить несколько вычислений параллельно, а равно и выполнять несколько различных действий в один интервал времени.

Используя термины инженерии разработки ПО и программирования, конкурентные паттерны — это паттерны проектирования, предназначенные для работы с многопоточными моделями программирования. Далее мы рассмотрим несколько конкурентных паттернов, а также следующие темы.

- ❑ Конкурентные вычисления с помощью конкурентных паттернов.
- ❑ Паттерн «Активный объект».
- ❑ Паттерн «Монитор».
- ❑ Паттерны «Полусинхронность» и «Полуасинхронность».
- ❑ Паттерн «Ведущий/ведомые».
- ❑ Локальная память потока выполнения.
- ❑ Паттерн «Реактор».
- ❑ Рекомендуемые практики для использующих конкурентность модулей.

Рассмотрим теперь подробнее каждый из указанных пяти конкурентных паттернов проектирования.

Паттерн «Активный объект»

Конкурентный паттерн проектирования «Активный объект» разграничивает/различает выполнение метода и вызов метода. Задача его состоит в расширении возможностей конкурентности наряду с упрощением синхронизированного доступа к объектам, расположенным в отдельных потоках управления. Он используется для обработки нескольких поступающих одновременно запросов клиента, а также для повышения уровня качества обслуживания. Рассмотрим следующую схему, иллюстрирующую использование паттерна проектирования «Активный объект» в конкурентном и многопоточном приложении (рис. 12.1).

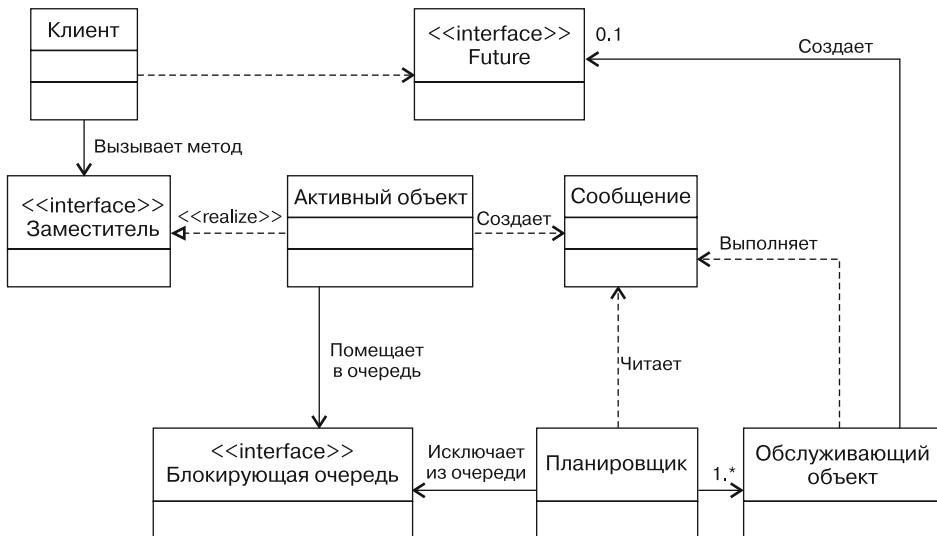


Рис. 12.1. Использование паттерна проектирования «Активный объект»

На этой схеме можно видеть следующие компоненты этого конкурентного паттерна проектирования.

- ❑ *Заместитель (proxy)*. Видимый клиенту активный объект. Заместитель оповещает заинтересованные стороны о своем интерфейсе.
- ❑ *Обслуживающий объект (servant)*. Предоставляет реализацию описанного в интерфейсе заместителя метода.
- ❑ *Список активизации (activation list)*. Сериализованный список с объектами запросов методов, получаемыми от заместителя. Благодаря этому списку у обслуживающего объекта появляется возможность работы в конкурентном режиме.

Как же работает этот паттерн проектирования? Каждый конкурентный объект располагается в отдельном потоке управления, отделенном от потока управления

клиента. Вызывается один из его методов, причем выполнение и вызов метода происходят в отдельных потоках управления. Однако клиент рассматривает этот процесс как обычный метод. Чтобы заместитель мог передавать запросы от клиента обслуживаемому объекту во время выполнения, они оба должны работать в отдельных потоках выполнения.

В этом паттерне проектирования заместитель после получения запроса формирует объект запроса метода и помещает его в список активизации. Этот метод выполняет две задачи: хранит объекты запросов методов и отслеживает, какие запросы методов он может выполнить. В объектах запросов методов содержатся также параметры запросов и любая другая информация, нужная для выполнения в дальнейшем нужного метода. Этот список активизации, в свою очередь, помогает конкурентному выполнению заместителя и обслуживаемого объекта.

Рассмотрим в следующем разделе еще один конкурентный паттерн проектирования — «Монитор».

Паттерн проектирования «Монитор»

Паттерн «Монитор» — еще один конкурентный паттерн проектирования, предназначенный для упрощения многопоточного программирования. Его реализация позволяет гарантировать, что в отдельный интервал времени выполняется только один метод отдельного объекта. С этой целью он синхронизирует выполнение конкурентных методов.

В отличие от паттерна проектирования «Активный объект», в паттерне «Монитор» нет отдельных потоков управления. Все получаемые запросы выполняются в потоке управления самого клиента с блокировкой доступа вплоть до возврата управления из метода. В отдельный интервал времени в одном мониторе может выполняться только один синхронизированный метод.

Паттерн проектирования «Монитор» предусматривает следующие программные требования к реализации.

- ❑ Интерфейс объекта задает границы синхронизации, гарантируя, что в отдельном объекте активным является только один метод.
- ❑ Необходимо гарантировать, что все объекты контролируют все нуждающиеся в синхронизации методы, сериализуя их прозрачным образом, без какого-либо уведомления клиента. Операции, с другой стороны, являются взаимоисключающими, но вызываются так же, как и обычные методы. Простейшие операции — сигналы и ожидания — используются для осуществления условной синхронизации.
- ❑ Для предотвращения взаимных блокировок и использования доступных механизмов конкурентности нужно обеспечить доступ других клиентов к объекту во время блокировки при выполнении метода объекта.
- ❑ Необходимо обеспечить соблюдение инвариантов в случае спонтанного прерывания методом потока управления.

На рис. 12.2 объект-клиент вызывает объект-монитор, насчитывающий несколько синхронизированных методов, связанный с условиями монитора и блокировками монитора.

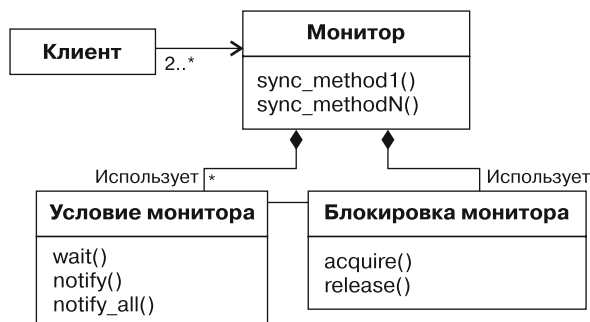


Рис. 12.2. Использование паттерна проектирования «Монитор» в конкурентных приложениях

Рассмотрим все компоненты этого конкурентного паттерна.

- ❑ *Объект-монитор (monitor object)*. Компонент, который предоставляет методы, синхронизируемые с клиентами.
- ❑ *Синхронизируемые методы (synchronized methods)*. Эти методы реализуют типобезопасные функции, экспортируемые интерфейсом объекта.
- ❑ *Условия монитора (monitor conditions)*. Этот компонент вместе с блокировками монитора определяет, продолжить выполнение синхронизируемого метода или перевести его в состояние ожидания.

Паттерны «Активный объект» и «Монитор» относятся к конкурентным паттернам проектирования.

А теперь мы обратимся к другой разновидности паттернов — конкурентным архитектурным паттернам.

Паттерны «Полусинхронность» и «Полуасинхронность»

Задача паттернов «Полусинхронность» и «Полуасинхронность» заключается в разграничении двух видов обработки — синхронного и асинхронного — с целью упрощения программы без негативного влияния на ее производительность.

Для целей обработки создаются два взаимодействующих уровня — синхронных и асинхронных сервисов — с уровнем организации очередей между ними.

В любой конкурентной системе есть как синхронные, так и асинхронные сервисы. Чтобы эти сервисы могли взаимодействовать друг с другом, паттерны «Полусинхронность» и «Полуасинхронность» расчленяют сервисы системы на уровни.

С помощью уровня организации очередей эти два уровня могут передавать сообщения друг другу.

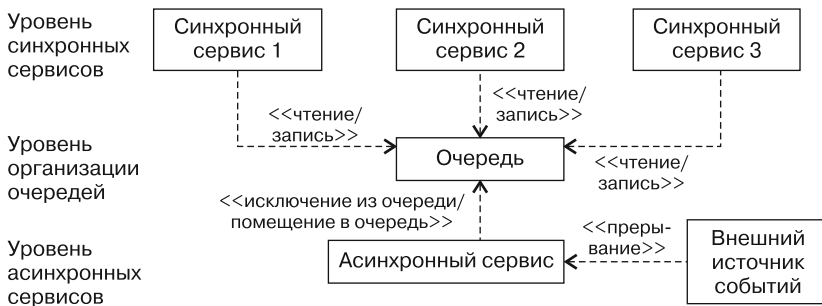


Рис. 12.3. Применение паттернов «Полусинхронность» и «Полуасинхронность»

Как можно видеть на рис. 12.3, в системе есть три уровня: *синхронных сервисов*, *организации очередей* и *асинхронных сервисов*. Синхронный уровень включает сервисы, синхронно работающие с очередями на уровне организации очередей, которые, в свою очередь, работают асинхронно с использованием асинхронных сервисов на уровне асинхронных сервисов. А эти асинхронные сервисы на данном уровне используют внешние источники событий.

Итак, существует три уровня задач.

- ❑ *Уровень синхронных задач.* Задачи на этом уровне представляют собой активные объекты и выполняют высокоуровневые операции ввода/вывода, синхронно перемещающие данные на уровень организации очередей.
- ❑ *Уровень организации очередей.* Этот уровень обеспечивает необходимые синхронизацию и буферизацию между уровнями синхронных и асинхронных задач.
- ❑ *Уровень асинхронных задач.* Задачи этого уровня выполняют обработку поступающих из внешних источников событий. Отдельного потока управления в них нет.

Мы обсудили конкурентные паттерны проектирования «Полусинхронность» и «Полуасинхронность». Перейдем к следующему конкурентному паттерну проектирования под названием «Ведущий/ведомые».

Паттерн «Ведущий/ведомые»

В конкурентной модели эффективно выполняется обнаружение, демультиплексирование, диспетчеризация и обработка запросов сервисов в источниках событий — множество потоков по очереди обрабатывают события, поступающие из набора источников событий. Альтернатива паттернам «Полусинхронность» и «Полу-

асинхронность» — паттерн «Ведущий/ведомые». Его можно применять вместо паттернов «Полусинхронность» и «Полуасинхронность» и «Активный объект» для повышения производительности. Использовать его можно при условии отсутствия ограничений на упорядоченность и синхронизацию при обработке нескольких потоков событий (рис. 12.4).

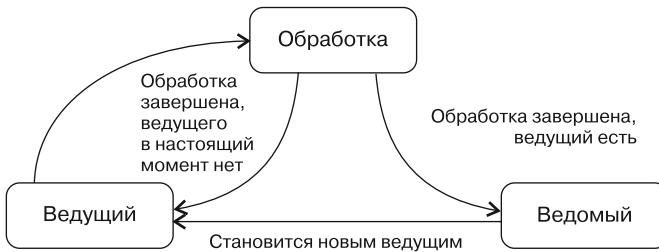


Рис. 12.4. Применение паттерна «Ведущий/ведомые»

Основная задача этого паттерна — конкурентная или одновременная обработка нескольких событий. Вследствие дополнительных затрат, связанных с организацией конкурентности, может оказаться недостаточно ресурсов для связывания отдельного потока выполнения с каждым дескриптором сокета. Важная особенность данной архитектуры: с помощью этого паттерна становится возможным демультиплексирование связей между потоками выполнения и источниками событий. При поступлении событий в источники данный паттерн формирует пул потоков выполнения с целью эффективного распределения набора источников событий. Эти источники событий демультиплексировывают поступающие события по очереди. Кроме того, события синхронно направляются сервисам приложения для обработки. За счет упорядоченного паттерном «Ведущий/ведомые» пула событий наступления события ждет только один поток выполнения, остальные потоки ожидают в очереди. При обнаружении потоком события ведомый переводится в ведущие, после чего обрабатывает поток и перенаправляет событие обработчику приложения.

В этом паттерне потоки-обработчики могут выполняться конкурентно, но только один поток ожидает поступления новых событий.

Паттерн «Реактор»

Паттерн «Реактор» используется для обработки запросов сервисов, поступающих в обработчик в конкурентном порядке от одного или нескольких входных источников. Получаемые запросы сервисов демультиплексировываются обработчиком сервиса и перенаправляются соответствующим обработчикам запросов. Все реакторные системы часто бывают однопоточными, но говорят, что они встречаются и в многопоточных средах.

Основное преимущество этого паттерна — возможность разделения компонентов системы на несколько модулей или переиспользуемых частей (рис. 12.5). Более того, благодаря ему можно без труда организовать крупномодульную конкурентность без усложнения системы за счет многопоточности.

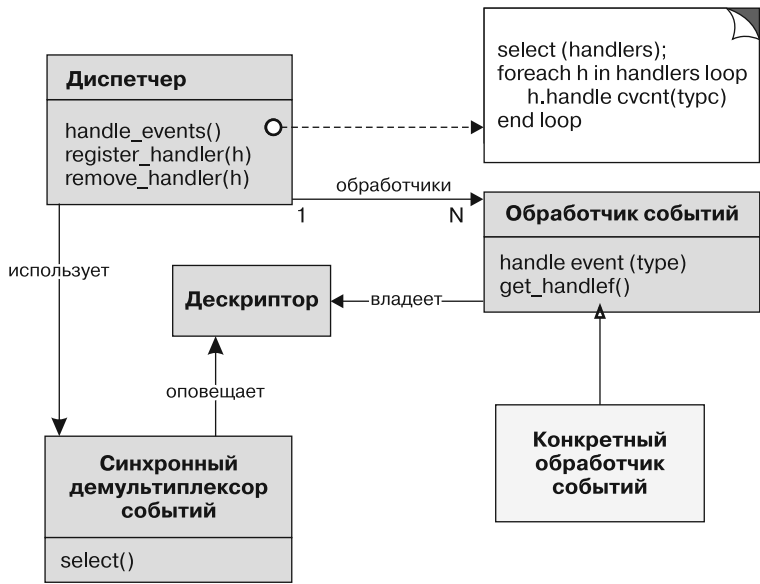


Рис. 12.5. Схема паттерна проектирования «Реактор»

Как можно видеть на предыдущей схеме, диспетчер с помощью демультиплексора оповещает обработчик, а уже он делает все, что нужно для обработки события ввода/вывода. Реактор откликается на события ввода/вывода, перенаправляя их соответствующему обработчику. На показанной выше схеме можно найти следующие компоненты данного паттерна проектирования.

- ❑ **Ресурсы.** Ресурсы, через которые поступают входные или потребляются выходные данные.
- ❑ **Синхронный демультиплексор событий.** Блокирует все ресурсы в цикле ожидания событий. При появлении возможности запуска синхронной операции ресурс отправляется диспетчеру через демультиплексор без блокирования.
- ❑ **Диспетчер.** Этот компонент отвечает за регистрацию и отмену регистрации обработчиков запросов. Через него происходит направление данных от ресурсов соответствующим обработчикам запросов.
- ❑ **Обработчик запросов.** Обрабатывает направленный диспетчером запрос.

А теперь мы перейдем к нашему следующему и последнему конкурентному паттерну «Локальная память потока выполнения».

Паттерн «Локальная память потока выполнения»

Для извлечения объекта, локального по отношению к потоку выполнения, может использоваться единая глобальная логическая точка доступа. Данный конкурентный паттерн проектирования позволяет реализовывать эту функцию несколькими потоками. При этом не требуются никакие дополнительные расходы на блокировку при каждом доступе к объекту. Порой этот конкретный паттерн рассматривается как противоположность всем остальным конкурентным паттернам проектирования, поскольку он решает несколько проблем за счет предотвращения разделения доступных ресурсов между потоками.

Все выглядит так, как будто поток выполнения приложения вызывает метод для обычного объекта. На самом же деле метод вызывается для объекта, относящегося к определенному потоку. Несколько потоков выполнения приложения могут использовать единый объект-заместитель для доступа к связанным с каждым из них уникальным объектам. Чтобы различать инкапсулируемые им объекты, относящиеся к определенным потокам, заместитель использует идентификаторы потоков приложения.

Рекомендуемые практики для использующих конкурентность модулей. Рассмотрим следующие практики, которые разработчикам рекомендуется применять при работе с модулями конкурентных приложений.

□ *Получение объектов Executor.* Фреймворк Executor языка Java предоставляет вспомогательные классы для получения объектов Executor. Различные типы объектов Executor предлагают различные стратегии выполнения потоков. Приведу три примера.

- `ExecutorService newCachedThreadPool()`. Создает пул потоков, использующий ранее созданные потоки, если таковые доступны¹. Применение подобного пула потоков обычно повышает производительность программ, выполняющих асинхронные задачи с коротким временем существования.
- `ExecutorService newSingleThreadExecutor()`. Создает объект Executor с одним потоком-исполнителем, который работает с неограниченной очередью. Задачи добавляются в очередь и выполняются строго последовательно. В случае сбоя этого потока-исполнителя во время выполнения на замену сбойному создается новый поток для выполнения последующих задач.
- `ExecutorService newFixedThreadPool(int nThreads)`. Производится переиспользование в рамках пула заданного количества `nThreads` потоков в рамках разделяемой неограниченной очереди. В каждый момент в активной обработке задач участвует не более `nThreads` потоков. Если все потоки пула заняты (активны) в момент поступления новой задачи, эта задача добавляется в очередь и ожидает там освобождения какого-либо из потоков. В случае сбоя потока до завершения работы объекта Executor создается новый поток для выполнения соответствующей задачи². Обратите внимание, что потоки в пуле существуют вплоть до вызова явным образом метода `shutdown`.

¹ Или при необходимости создающий новые.

² И последующих задач.

- ❑ *Использование синхронизированных конструкций с невытесняющей моделью конкурентности.* Рекомендуется использовать при возможности синхронизированные конструкции с невытесняющей моделью конкурентности.
- ❑ *Отказ от слишком продолжительных задач и превышение лимита.* Слишком продолжительные задачи могут приводить к взаимным блокировкам, нехватке ресурсов и даже мешать нормальному функционированию других задач. Для повышения производительности лучше разбивать большие задачи на более маленькие подзадачи. Один из способов избежать взаимных блокировок, нехватки ресурсов и т. п. — превышение лимита (oversubscription). С помощью этой методики можно создать больше потоков, чем доступно физически. Она демонстрирует отличные результаты в случае продолжительных задач с высоким значением задержки.
- ❑ *Использование конкурентных функций управления памятью.* При необходимости крайне рекомендуется использовать следующие конкурентные функции управления памятью. Они удобны при выделении памяти для большого количества маленьких объектов с коротким временем жизни. Для выделения и освобождения памяти служат такие функции, как `Alloc` и `Free`, не использующие блокировки и барьеры памяти.
- ❑ *Использование RAII для управления жизненным циклом конкурентных объектов.* RAII расшифровывается как «Получение ресурса есть инициализация» (Resource Acquisition Is Initialization). Эта программная идиома представляет собой эффективный способ управления жизненным циклом конкурентного объекта.

Вот и все, что я хотел вам рассказать о конкурентности и паттернах проектирования, предназначенных для ее реализации и управления ею в приложении.

Резюме

В этой главе вы изучили несколько конкурентных паттернов проектирования и познакомились со сценариями их использования. Я охватил лишь основы конкурентных паттернов проектирования, а именно паттерны «Активный объект», «Монитор», «Полусинхронность» и «Полуасинхронность», «Ведущий/ведомые», «Локальная память потока» и «Реактор». Все они относятся к паттернам проектирования конкурентных приложений в многопоточной среде. Мы также обсудили некоторые рекомендуемые практики создания конкурентных модулей. Надеюсь, это было полезно и помогло вам разобраться в работе конкурентных паттернов!