Spring в действии

Третье издание

Крейг Уоллс

ДМК издательство

MANNING

УДК 004Spring
ББК 32.973-018.2
У62

Уоллс К.
У62 Spring в действии. – М.: ДМК Пресс, 2013. – 752 с.: ил.

ISBN 978-5-94074-568-6

Фреймворк Spring Framework – необходимый инструмент для разработчиков приложений на Java.

В книге описана последняя версия Spring 3, который несет в себе новые мощные особенности, такие как язык выражений SpEL, новые аннотации для работы с контейнером IoC и поддержка архитектуры REST. Автор, Крейг Уоллс, обладает особым талантом придумывать весьма интересные примеры, сосредоточенные на особенностях и приемах использования Spring, которые действительно будут полезны читателям.

В русскоязычном переводе добавлены главы из 2-го американского издания, которые автор не включил в 3-е издание «Spring in Action».

Издание предназначено как для начинающих пользователей фреймворка, так и для опытных пользователей Spring, желающих задействовать новые возможности версии 3.0.

УДК 004Spring
ББК 32.973-018.2

ISBN 978-1-9351-8235-1 (анг.)

ISBN 978-5-94074-568-6 (рус.)

Содержание

Глава 1. Введение в Spring

В этой главе рассматриваются следующие темы:

- обзор основных модулей Spring;
- разделение прикладных объектов;
- управление сквозными задачами с помощью AOP;
- контейнер компонентов в Spring.

Все началось с компонента.

В 1996 году язык программирования Java был еще новой, перспективной платформой, вызывающей большой интерес. Многие разработчики пришли в этот язык после того, как они увидели, как с помощью апплетов можно создавать полнофункциональные и динамические веб-приложения. Однако вскоре они увидели, что этот новый и странный язык способен на большее, нежели простое управление мультяшными героями. В отличие от других существующих языков, Java позволил писать сложные приложения, состоящие из отдельных частей. Они пришли ради апплетов, а остались ради компонентов.

В декабре 1996 года компания Sun Microsystems опубликовала спецификацию JavaBeans 1.00-A, определившую модель программных компонентов Java – набор приемов программирования, позволяющих повторно использовать простые Java-объекты и легко конструировать из них более сложные приложения. Несмотря на то что компоненты JavaBeans были задуманы как универсальный механизм определения повторно используемых прикладных компонентов, они использовались преимущественно в качестве шаблона для создания элементов пользовательского интерфейса. Они казались слишком простыми для «настоящей» работы. Промышленным программистам требовалось нечто большее.

Сложные приложения часто требуют таких услуг, как поддержка транзакций, безопасность и распределенные вычисления, которые не были предусмотрены спецификацией JavaBeans. Поэтому в марте 1998 года компания Sun Microsystems опубликовала новую спецификацию – Enterprise JavaBeans (EJB) 1.0. Эта спецификация

расширила понятие серверных Java-компонентов, обеспечив столь необходимые службы, однако она не смогла поддержать той простоты, что была заложена в оригинальную спецификацию JavaBeans. Кроме похожего названия, спецификация EJB имеет мало общего со спецификацией JavaBeans.

Несмотря на появление множества успешных приложений, созданных на основе спецификации EJB, она никогда не применялась по прямому назначению: для упрощения разработки корпоративных приложений. Это правда, что декларативная модель программирования EJB упрощает многие инфраструктурные аспекты разработки, такие как транзакции и безопасность. Но она привнесла другие сложности, требуя создания дескрипторов развертывания и использования шаблонного кода (домашние и удаленные/локальные интерфейсы). С течением времени многие разработчики разочаровались в EJB, что привело к спаду популярности данной технологии и поиску более простых путей.

Сегодня разработка Java-компонентов вернулась к своим истокам. Новые технологии программирования, включая аспектно-ориентированное программирование (AOP) и внедрение зависимостей (DI), дают JavaBeans дополнительные возможности, ранее заложенные в EJB. Эти технологии оснащают обычные Java-объекты (Plain-Old Java Objects, POJO) моделью декларативного программирования, напоминающей EJB, но без всей сложности спецификации EJB. Больше не надо создавать громоздкий компонент EJB, когда достаточно простого компонента JavaBean.

Следует отметить, что даже EJB эволюционировала к модели программирования на основе POJO. Заимствуя идеи AOP и DI, последняя спецификация EJB стала существенно проще, чем предшествующие. Однако многие разработчики расценили этот шаг как слишком маленький и сделанный достаточно поздно. К моменту выхода спецификации EJB 3 другие фреймворки на основе модели POJO уже зарекомендовали себя в сообществе пользователей Java как фактические стандарты.

Фреймворк Spring Framework, который будет изучаться на протяжении всей этой книги, является передовым средством разработки приложений на основе POJO. В этой главе Spring Framework будет рассматриваться с общей точки зрения, чтобы дать представление о том, что такое Spring. Данная глава позволит получить хорошее понимание круга задач, решаемых фреймворком Spring, и подготовит почву для остальной части книги. Для начала выясним, что такое Spring.

1.1. Упрощение разработки на языке Java

Spring – это свободно распространяемый фреймворк, созданный Родом Джонсоном (Rod Johnson) и описанный в его книге «Expert One-on-One: J2EE Design and Development». Он был создан с целью устранить сложности разработки корпоративных приложений и сделать возможным использование простых компонентов JavaBean для достижения всего того, что ранее было возможным только с использованием EJB. Однако область применения Spring не ограничивается разработкой программных компонентов, выполняющихся на стороне сервера. Любое Java-приложение может использовать преимущества фреймворка в плане простоты, тестируемости и слабой связанности.

> **Термины, обозначающие компоненты.** Несмотря на то, что при обсуждении фреймворка Spring для обозначения программных компонентов слова «bean» и «JavaBean» используются как синонимы, это не означает, что компоненты в Spring должны точно соответствовать спецификации JavaBeans. Компоненты в Spring могут быть любыми простыми объектами модели POJO. В этой книге термин «JavaBeans» будет использоваться как синоним термина «POJO» (обычный Java-объект).

Как будет не раз показано на протяжении этой книги, фреймворк Spring обладает весьма широкими возможностями. Но в основе практически всех его особенностей лежат несколько фундаментальных идей, направленных на достижение главной цели – *упрощение разработки приложений на языке Java*.

Весьма смелое заявление! Разработчики многих фреймворков утверждают, что их продукты упрощают те или иные аспекты разработки. Но целью фреймворка Spring является упрощение разработки приложений Java вообще. Это требует некоторых пояснений. Так как же фреймворк Spring упрощает разработку на языке Java?

В своем устремлении на сложности, связанные с разработкой на языке Java, фреймворк Spring использует четыре ключевые стратегии:

- легковесность и ненасильственность благодаря применению простых Java-объектов (POJO);
- слабое связывание посредством внедрения зависимостей и ориентированности на интерфейсы;
- декларативное программирование через аспекты и общепринятые соглашения;
- уменьшение объема типового кода через аспекты и шаблоны.

Практически все возможности фреймворка Spring уходят корнями в эти стратегии. В остальной части главы каждая из этих идей будет рассматриваться более подробно на конкретных примерах, позволяющих увидеть, как фреймворк Spring действительно упрощает разработку приложений на языке Java. Для начала посмотрим, как он обеспечивает ненасильственность, поощряя использование простых объектов.

1.1.1. Свобода использования POJO

Те, кто имеет опыт достаточно продолжительной разработки на языке Java, вероятно, видели (и даже могли использовать) фреймворки, вынуждающие расширять свои классы или предусматривать реализацию своих интерфейсов. Классическим примером являются сеансовые компоненты эры EJB 2. Как показано в простейшем примере `HelloWorldBean`, спецификация EJB 2 предъявляет достаточно сложные требования:

Листинг 1.1. Спецификация EJB 2.1 вынуждает реализовывать ненужные методы

```
package com.habuma.ejb.session;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class HelloWorldBean implements SessionBean { // Зачем все эти методы
     public void ejbActivate() {
     }

     public void ejbPassivate() {
     }

     public void ejbRemove() {
     }

     public void setSessionContext(SessionContext ctx) {
     }

     public String sayHello() {    // Основная логика компонента EJB
          return "Hello World";
     }

     public void ejbCreate() {
     }
}
```

Интерфейс SessionBean обеспечивает возможность управления жизненным циклом компонента EJB за счет реализации различных методов обратного вызова (имена этих методов начинаются с последовательности символов ejb). Или, говоря другими словами, интерфейс SessionBean вынуждает вторгаться в жизненный цикл компонента EJB, даже если в этом нет необходимости. Масса программного кода в примере HelloWorldBean нужна только ради удовлетворения нужд фреймворка. В результате возникает вопрос: кто для кого работает?

Однако спецификация EJB 2 не единственная в своей насильственности. Другие популярные фреймворки, такие как ранние версии Struts, WebWork и Tapestry, накладывали свой отпечаток на иначе простые Java-классы. Эти тяжеловесные фреймворки вынуждали разработчиков создавать классы, захламленные ненужным программным кодом, часто сложные в тестировании.

Фреймворк Spring не вынуждает (насколько это возможно) захламлять приложения программным кодом для поддержки своего API. Он практически никогда не заставляет обеспечивать реализацию своих интерфейсов или наследовать свои классы. Напротив, в приложениях, основанных на фреймворке Spring, классы часто вообще не имеют никаких признаков, по которым можно было бы судить, что они используются фреймворком. В худшем случае класс может быть аннотирован одной из аннотаций Spring, но во всех остальных отношениях он будет обычным объектом POJO.

Для иллюстрации класс HelloWorldBean, представленный в листинге 1.1, можно преобразовать в компонент, управляемый фреймворком Spring, как показано в листинге 1.2.

Листинг 1.2. Фреймворк Spring не выдвигает необоснованных требований к классу HelloWorldBean

```
package com.habuma.spring;

public class HelloWorldBean {
     public String sayHello() {                // Это все, что необходимо
          return "Hello World";
     }
}
```

Так лучше? Исчезли все ненужные методы управления жизненным циклом. Класс HelloWorldBean не реализует, не наследует и не

импортирует ничего из Spring API. Класс `HelloWorldBean` мал, краток и во всех смыслах является простым Java-объектом.

Несмотря на свою простоту, простые Java-объекты могут обладать большими возможностями. Одним из механизмов Spring, увеличивающих мощь простых объектов, является их объединение с помощью внедрения зависимостей. Рассмотрим, как внедрение зависимостей помогает сохранить прикладные объекты независимыми друг от друга.

1.1.2. Внедрение зависимостей

Для кого-то фраза «внедрение зависимостей» может звучать устрашающе, вызывая в воображении сложные приемы программирования или шаблоны проектирования. Однако на самом деле DI не настолько сложно, как кажется. На самом деле применение DI в проектах позволяет существенно упростить программный код, облегчит его понимание и тестирование.

Любое нетривиальное приложение (более сложное, чем простое приложение Hello World) состоит из двух или более классов, которые взаимодействуют друг с другом, реализуя некоторую логику. Обычно каждый объект ответствен за получение собственных ссылок на объекты, с которыми он взаимодействует (его зависимости). Это может привести к сильной связности и сложностям при тестировании.

Например, взгляните на класс **рыцаря** в листинге 1.3 ниже.

Листинг 1.3. Класс DamselRescuingKnight может принимать только экземпляр класса RescueDamselQuests

```
package com.springinaction.knights;

public class DamselRescuingKnight implements Knight {
     private RescueDamselQuest quest;

     public DamselRescuingKnight() {
          quest = new RescueDamselQuest();  // Тесная связь с классом
     }                                      // RescueDamselQuest

     public void embarkOnQuest() throws QuestException {
          quest.embark();
     }
}
```

Как показано в листинге 1.3, экземпляр `DamselRescuingKnight` создает собственный экземпляр `RescueDamselQuest` в конструкторе. Это обусловливает тесную связь между `DamselRescuingKnight` и `RescueDamselQuest` и существенно ограничивает возможности рыцаря. Если потребуется спасти даму, он готов будет совершить этот подвиг. Но если потребуется убить дракона или, например, стать рыцарем Круглого стола, он не сможет сделать этого[1].

Но, что хуже всего, для класса `DamselRescuingKnight` очень сложно будет написать модульный тест. В процессе тестирования желательно было бы убедиться, что при вызове метода `embarkOnQuest()` класса `DamselRescuingKnight` вызывается метод `embark()` класса `RescueDamselQuest`. Но в данном случае нет очевидного способа реализовать такую проверку. К сожалению, класс `DamselRescuingKnight` останется непротестированным.

Связь классов – это «двуглавый зверь». С одной стороны, сильно связанный код сложен в тестировании, его трудно использовать повторно, он сложен в понимании, и, как правило, такой код оказывается очень хрупким при исправлении ошибок (исправление одной ошибки может привести к нескольким новым). С другой стороны, совершенно не связанный код ничего не делает. Чтобы делать что-то полезное, классы должны знать друг о друге. Связанность необходима, но должна тщательно контролироваться.

С другой стороны, благодаря DI объекты получают свои зависимости во время создания от некоторой третьей стороны, координирующей работу каждого объекта в системе. Объекты не создают и не получают свои зависимости самостоятельно – зависимости внедряются в объекты.

Для иллюстрации рассмотрим класс `BraveKnight`, представленный в листинге 1.4, реализующий рыцаря, который не только храбр, но и способен совершать любые подвиги.

Листинг 1.4. Класс BraveKnight, достаточно гибкий, чтобы совершить любой подвиг

```
package com.springinaction.knights;

public class BraveKnight implements Knight {
```

[1] Здесь необходимо уточнить, что класс `DamselRescuingKnight` реализует «рыцаря, спасающего даму», а класс `RescueDamselQuest` реализует «сценарий спасения дамы». – *Прим. перев.*

```
    private Quest quest;

    public BraveKnight(Quest quest) {
        this.quest = quest;                    // Внедрение сценария подвига
    }

    public void embarkOnQuest() throws QuestException {
        quest.embark();
    }
}
```

Как видно из листинга, в отличие от класса `DamselRescuingKnight`, класс `BraveKnight` не создает собственного сценария подвига, а получает его извне, в виде аргумента конструктора. Такой способ внедрения зависимостей называется *внедрением через конструктор*.

Более того, сценарий подвига имеет тип интерфейса `Quest`, который реализуют все такие сценарии. Поэтому `BraveKnight` (храбрый рыцарь) сможет совершать любые подвиги, такие как `RescueDamselQuest` (спасти даму), `SlayDragonQuest` (убить дракона), `MakeRoundTableRounderQuest` (стать рыцарем Круглого стола) или любой другой, реализующий интерфейс `Quest`.

Фактически класс `BraveKnight` никак не связан с конкретной реализацией `Quest`. Для него не важно, какой подвиг будет поручен, при условии что он реализует интерфейс `Quest`. В этом состоит основное преимущество DI – слабая связанность. Если объект взаимодействует со своими зависимостями только через их интерфейсы (ничего не зная о конкретных реализациях или особенностях их создания), зависимости можно будет замещать любыми другими реализациями, без необходимости учитывать эти различия в самом объекте.

Прием замены зависимостей очень часто используется при тестировании, когда выполняется подстановка фиктивной реализации. Класс `DamselRescuingKnight` невозможно было протестировать в полной мере из-за тесной связи, но класс `BraveKnight` легко поддается тестированию за счет подстановки фиктивной реализации интерфейса `Quest`, как показано в листинге 1.5.

Листинг 1.5. Протестировать класс BraveKnight можно с помощью фиктивной реализации интерфейса Quest

```
package com.springinaction.knights;

import static org.mockito.Mockito.*;
```

```
import org.junit.Test;

public class BraveKnightTest {
     @Test
     public void knightShouldEmbarkOnQuest() throws QuestException {
          Quest mockQuest = mock(Quest.class);   // Создание фиктивного
                                                 // объекта Quest
          BraveKnight knight = new BraveKnight(mockQuest); // Внедрение
          knight.embarkOnQuest();

          verify(mockQuest, times(1)).embark();
     }
}
```

В данном примере фиктивная реализация интерфейса `Quest` создана с помощью фреймворка Mockito. После получения фиктивного объекта создается новый экземпляр `BraveKnight`, в который через конструктор внедряется фиктивный объект `Quest`. После вызова метода `embarkOnQuest()` выполняется обращение к фреймворку Mockito с целью убедиться, что метод `embark()` интерфейса `Quest` фиктивного объекта был вызван только один раз.

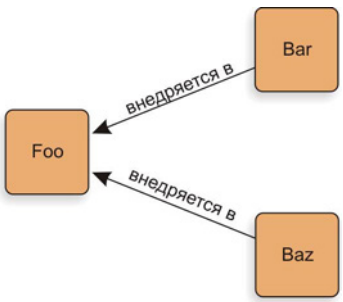

Рис. 1.1. Механизм внедрения зависимостей основан на предоставлении объекту его зависимостей извне, а не на приобретении этих зависимостей самим объектом

Передача сценария подвига рыцарю

Теперь, когда класс `BraveKnight` может принимать любые задания, как определить, какой именно объект `Quest` был ему передан? Процесс создания связей между прикладными компонентами называется *связыванием* (wiring). Фреймворк Spring поддерживает множест-

во способов связывания компонентов, но наиболее общим из них является способ на основе XML. В листинге 1.6 показано содержимое простого конфигурационного файла Spring, `knights.xml`, который передает объекту `BraveKnight` задание `SlayDragonQuest`.

Листинг 1.6. Внедрение сценария SlayDragonQuest в объект BraveKnight средствами 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-3.0.xsd">

     <bean id="knight" class="com.springinaction.knights.BraveKnight">
          <constructor-arg ref="quest" />   <!-- Внедрение компонента quest -->
     </bean>
     <!-- Создание SlayDragonQuest -->
     <bean id="quest"
          class="com.springinaction.knights.SlayDragonQuest" />
</beans>
```

Это простой способ связывания компонентов. Пока не стоит слишком беспокоиться о деталях. Подробнее о конфигурировании Spring и о том, что происходит на данном этапе, будет рассказываться в главе 2, где также будут показаны другие способы связывания компонентов в Spring.

Теперь, объявив отношения между `BraveKnight` и `Quest`, необходимо загрузить XML-файл и запустить приложение.

Рассмотрим этот механизм в действии

В приложении, созданном на основе Spring, *контекст приложения* загружает определения компонентов и связывает их вместе. За создание объектов, составляющих приложение, и их связывание полностью отвечает контекст приложения. В составе фреймворка Spring имеется несколько реализаций контекста приложения.

Поскольку *компоненты* приложения объявлены в XML-файле `knights.xml`, в качестве контекста приложения может использоваться класс `ClassPathXmlApplicationContext`. Реализация контекста в Spring загружает контекст из одного или более XML-файлов, находящихся в библиотеке классов (classpath). Метод `main()` в листинге 1.7 использует `ClassPathXmlApplicationContext`, чтобы загрузить `knight.xml` и получить ссылку на объект `Knight`.

Листинг 1.7. KnightMain.java загружает контекст Spring, содержащий объект Knight

```
package com.springinaction.knights;

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

public class KnightMain {
     public static void main(String[] args) {
          // Загрузка контекста Spring
          ApplicationContext context =
               new ClassPathXmlApplicationContext("knights.xml");

          // Получение компонента knight
          Knight knight = (Knight) context.getBean("knight");
          // Использование компонента knight
          knight.embarkOnQuest();
     }
}
```

Здесь метод `main()` создает контекст приложения Spring на основе файла `knights.xml`. Затем использует контекст как фабрику для извлечения компонента с идентификатором «knight». Получив ссылку на объект `Knight`, он вызывает метод `embarkOnQuest()`, чтобы отправить рыцаря выполнять задание. Обратите внимание, что этот класс ничего не знает о задании `Quest`, переданном рыцарю. Собственно говоря, он даже не знает, что имеет дело с классом `BraveKnight`. Только файл `knights.xml` имеет полную информацию о реализациях, участвующих в работе.

На этом мы закончим краткое знакомство с приемом внедрения зависимостей. Дополнительные примеры применения DI будут встречаться на протяжении всей книги. Желающим поближе познакомиться с этим приемом я рекомендую прочитать книгу Дханжи Прасанна (Dhanji R. Prasanna) «Dependency Injection».

А теперь познакомимся с другими стратегиями упрощения программирования на языке Java – декларативным программированием посредством *аспектов*.

1.1.3. Применение аспектно-ориентированного программирования

Хотя DI делает возможным ослабить связь между компонентами приложения, *аспектно-ориентированное программирование* позво-

ляет оформлять функциональность, используемую в приложении, в виде многократно используемых компонентов.

Аспектно-ориентированное программирование часто определяют как прием программирования, поддерживающий разделение задач в пределах программной системы. Системы конструируются из нескольких компонентов, каждый из которых отвечает за определенную часть функциональности. Зачастую эти компоненты также несут дополнительную ответственность сверх своей основной функциональности. Системные службы, такие как журналирование, управление транзакциями и безопасность, часто находят свое отражение в компонентах, основная задача которых заключается в чем-то другом. Такие системные службы обычно называют *сквозными задачами*, потому что в их работу может вовлекаться несколько компонентов системы.

Распространение этих задач на несколько компонентов влечет за собой увеличение сложности программного кода:

- программный код, реализующий решение общесистемных задач, дублируется в разных компонентах. Это означает, что в случае необходимости что-то изменить в этой реализации придется просмотреть множество компонентов. Даже если вынести решение задачи в отдельный модуль, чтобы компонентам оставалось только вызвать единственный метод, все равно вызов этого метода будет дублироваться в разных местах;
- компоненты захламляются программным кодом, не имеющим прямого отношения к их основной функциональности. Метод добавления записи в адресную книгу должен заботиться только о том, как добавить адрес, а не о безопасности или о необходимости использования транзакции.

Эта сложность иллюстрируется на рис. 1.2. Бизнес-объекты слева слишком тесно связаны с системными службами. Мало того, что каждый объект знает, что его операции должны фиксироваться в журнале, соответствовать требованиям безопасности и выполняться в рамках транзакции, они еще несут ответственность за использование этих служб.

AOP делает возможным отделение этих служб и декларативное их применение к необходимым компонентам. Благодаря этому компоненты могут сосредоточиться на решении собственных задач, полностью игнорируя системные службы, которые могут быть вовлечены в общий процесс. Проще говоря, аспекты позволяют сохранить простоту POJO.