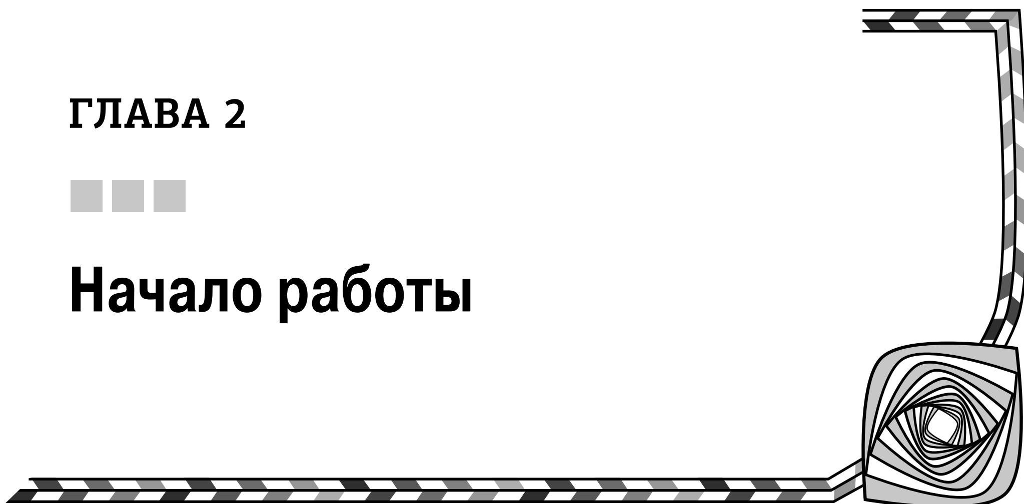

ГЛАВА 2

Начало работы

Зачастую при освоении нового инструментального средства разработки сложнее всего выяснить, с чего следует начать. Как правило, положение усугубляется, если инструментальное средство предоставляет слишком много вариантов выбора, как это делает Spring. Правда, приступить к работе с Spring не так уж и трудно, если знать, где и что искать в первую очередь. В этой главе поясняется, с чего следует начать. В частности, здесь будут рассмотрены следующие вопросы.

- **Получение Spring.** Первый логический шаг состоит в получении и сборке архивных JAR-файлов Spring. Если вы хотите сделать все быстро, воспользуйтесь фрагментами кода для управления зависимостями в своей системе сборки, руководствуясь примерами, предоставленными по адресу `http://projects.spring.io/spring-framework`. Но если вы стремитесь оказаться на переднем крае разработки средствами Spring, поищите последнюю версию исходного кода Spring в хранилище GitHub[1].
- **Варианты упаковки Spring.** Упаковка Spring является модульной; разрешается выбирать те компоненты, которые должны использоваться в приложении, а при распространении готового приложения включать в его состав только эти компоненты. Каркас Spring состоит из многих модулей, но вам понадобится только их подмножество, которое зависит от конкретных потребностей приложения. У каждого модуля имеется свой скомпилированный двоичный код в архивном JAR-файле вместе с документацией, автоматически составленной утилитой Javadoc, а также исходными архивными JAR-файлами.

[1] Хранилище GitHub исходного кода Spring находится по адресу `http://github.com/spring-projects/spring-framework`.

- **Руководства по Spring.** На веб-сайте Spring имеется раздел Guides (Руководства), доступный по адресу `https://spring.io/guides`. Под руководствами понимаются краткие практические инструкции по составлению начального примера для решения любой задачи разработки средствами Spring. В этих руководствах отражены также последние выпуски проектов и технологий Spring, а следовательно, в ваше распоряжение предоставлены наиболее актуальные примеры.
- **Тестовый комплект и документация.** К предметам особой гордости членов сообщества разработчиков Spring относится всеобъемлющий тестовый набор и комплект документации. Тестированию отводится львиная доля работы в команде разработчиков. Комплект документации, входящий в стандартный дистрибутив, также составлен превосходно.
- **Пример приложения “Hello World!” в Spring.** Как бы то ни было, мы считаем, что начинать работу с любым новым инструментальным средством для программирования лучше всего с написания какого-нибудь кода. Поэтому мы представим простой пример полноценной реализации, основанной на внедрении зависимостей, хорошо известного приложения “Hello World!”. Не отчаивайтесь, если вы не сразу поймете весь его исходный код, поскольку исчерпывающие пояснения будут приведены далее в книге.

Если вы уже знакомы с основами Spring Framework, можете переходить непосредственно к главе 3, где рассматривается реализация принципов инверсии управления и внедрения зависимостей в Spring. Но, даже зная основы Spring, вы наверняка найдете в этой главе интересные сведения, особенно касающиеся упаковки и зависимостей.

Получение Spring Framework

Прежде чем приступить к разработке средствами Spring, необходимо получить код самого каркаса. Для этого имеются две возможности: воспользоваться своей системой сборки для установки требующихся модулей или извлечь и построить код из хранилища Spring в GitHub. Применение инструментального средства для управления зависимостями (например, Maven или Gradle) зачастую оказывается самым простым подходом, поскольку для этого достаточно объявить зависимость в файле конфигурации и дать инструментальному средству возможность автоматически получить требующиеся библиотеки.

■ ***На заметку*** Если у вас имеется подключение к Интернету и вы пользуетесь инструментальным средством сборки вроде Maven или Gradle вместе с такой IDE, как Eclipse или IntelliJ IDEA, то загрузите документацию в формате Javadoc и библиотеки автоматически, чтобы иметь к ним доступ в процессе разработки. При переходе на новые версии в файлах конфигурации сборки эти библиотеки и документирующие комментарии будут также обновлены в процессе сборки проекта.

Быстрое начало

Посетите веб-страницу проекта Spring Framework[2], чтобы получить фрагмент кода управления зависимостями для системы сборки, который позволит включить в ваш проект последнюю версию RELEASE каркаса Spring. Можете также воспользоваться промежуточными моментальными снимками предстоящих выпусков или предыдущих версий данного каркаса.

Если применяется модуль Spring Boot, то указывать требующуюся версию Spring не нужно, поскольку этот модуль предоставляет файлы "стартовой" объектной модели проекта (POM) с целью упростить конфигурацию Maven и выбираемую по умолчанию стартовую конфигурацию Gradle. Следует лишь иметь в виду, что в версиях Spring Boot, предшествующих версии 2.0.0.RELEASE, используются версии Spring 4.*x*.

Извлечение Spring из хранилища GitHub

Если вы хотите иметь доступ к новым функциональным средствам Spring еще до того, как они будут отражены в моментальных снимках, можете извлечь исходный код непосредственно из хранилища GitHub в Pivotal. Для получения последней версии исходного кода Spring установите сначала систему контроля версий Git, которую можно загрузить по адресу `http://git-scm.com/`, а затем откройте окно командной строки или терминальной оболочки и введите следующую команду:

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

В корневой папке проекта находится файл `README.md` с подробным описанием требований и процесса сборки каркаса Spring из исходного кода.

Выбор подходящего комплекта JDK

Каркас Spring Framework построен на Java, а это означает, что для его применения необходимо иметь возможность выполнять приложения Java на своем компьютере. Для этого необходимо установить Java. Когда речь заходит о разработке приложений на Java, разработчики обычно употребляют следующие термины и их сокращения.

- **Виртуальная машина Java (JVM)** — это абстрактная машина. По ее спецификации предоставляется среда выполнения, в которой исполняется байт-код Java.
- **Среда выполнения Java (Java Runtime Environment — JRE).** Предоставляет окружение для исполнения байт-кода Java и является физически существующей реализацией виртуальной машины JVM. Состоит из ряда библиотек и прочих файлов, применяемых в виртуальной машине JVM во время выполнения. Корпорация Oracle приобрела компанию Sun Microsystems в 2010 году и с тех пор активно предоставляет новые версии и обновления Java. Другие компании,

[2] См. `http://projects.spring.io/spring-framework`.

в том числе IBM, предоставляют собственные реализации виртуальной машины JVM.

- **Комплект разработки прикладных программ на Java (Java Development Kit — JDK).** Содержит среду JRE, документацию, а также инструментальные средства Java. Именно этот комплект устанавливают разработчики на своих машинах. В интегрированной среде разработки (IDE) вроде IntelliJ IDEA или Eclipse требуется указывать место установки комплекта JDK, чтобы загружать из него классы и документацию в процессе разработки.

Если вы пользуетесь инструментальным средством сборки вроде Maven или Gradle (исходный код примеров из этой книги организован в многомодульный проект Gradle), для него потребуется также виртуальная машина JVM. Оба инструментальных средства сборки Maven и Gradle сами являются проектами, построенными на Java.

На момент написания данной книги последней устойчивой версией Java считалась версия Java 8, хотя в конце сентября 2017 года была выпущена Java 9. Комплект JDK можно загрузить по адресу `https://www.oracle.com/`. По умолчанию он будет установлен в каком-нибудь стандартном месте на вашем компьютере, хотя это зависит от конкретной операционной системы. Если вы желаете пользоваться Maven или Gradle в режиме командной строки, вам придется определить переменные окружения для комплекта JDK и инструментального средства сборки Maven или Gradle, указав путь к их исполняемым файлам в системе. Инструкции, поясняющие, как это сделать, находятся на официальном веб-сайте каждого продукта, а также приведены в приложении к данной книге.

В главе 1 был приведен перечень версий Spring и требующиеся версии JDK. В этой книге рассматривается версия Spring 5.0.*x*. Исходный код примеров, представленных в книге, написан с использованием синтаксиса Java 8, поэтому вам понадобится версия JDK 8, чтобы скомпилировать и выполнить исходный код этих примеров.

Упаковка Spring

Модули Spring просто являются архивными JAR-файлами, в которых упакован код, требующийся для конкретного модуля. Уяснив назначение каждого модуля, вы сможете выбрать модули для вашего проекта и включить их в свой код.

В версии 5.0.0.RELEASE каркас Spring состоит из 21 модуля, упакованного в 21 архивный JAR-файл. Эти JAR-файлы и соответствующие им модули перечислены в табл. 2.1. В действительности имена архивных JAR-файлов представлены в форме, подобной следующей: `springaop-5.0.0.RELEASE.jar`, но ради простоты в табл. 2.1 приводится только та их часть, которая характерна для данного модуля (например, `aop`).

Таблица 2.1. Модули Spring

Модуль	Описание
aop	Содержит все классы, требующиеся для применения в приложении средств АОП из Spring. Этот архивный JAR-файл должен быть также включен в приложение, если планируется пользоваться другими средствами Spring, в которых применяется АОП (например, декларативным управлением транзакциями). Кроме того, в этот модуль упакованы классы, поддерживающие интеграцию с библиотекой AspectJ
aspects	Содержит все классы, предназначенные для расширенной интеграции с библиотекой AspectJ для АОП. Он понадобится, например, в том случае, если вы применяете классы Java для своей конфигурации Spring и нуждаетесь в управлении транзакциями с помощью аннотаций в стиле AspectJ
beans	Содержит все классы, поддерживающие манипулирование компонентами Spring Beans. Большинство классов из этого модуля поддерживают реализацию фабрики компонентов Spring Beans. Так, в этот модуль упакованы классы, требующиеся для обработки XML-файла конфигурации Spring и аннотаций Java
beans-groovy	Содержит классы Groovy, поддерживающие манипулирование компонентами Spring Beans
context	Содержит классы, которые предоставляют многие расширения для ядра Spring. Все классы должны использовать интерфейс **ApplicationContext** из Spring, описанный в главе 5, а также классы для интеграции с EJB, Java Naming and Directory Interface (JNDI) и Java Management Extensions (JMX). В этом модуле содержатся также классы Spring для удаленного взаимодействия, классы для интеграции с языками динамических сценариев (например, JRuby, Groovy, BeanShell), классы из прикладного интерфейса API по спецификации JSR-303 (Beans Validation), классы для планирования и выполнения заданий и т.д.
context-indexer	Содержит реализацию индексатора, предоставляющую доступ к подходящим компонентам, определенным в файле **META-INF/spring.components**. Базовый класс **CandidateComponentsIndex** не предназначен для внешнего применения
context-support	Содержит дополнительные расширения для модуля **spring-context**. На стороне пользовательского интерфейса имеются классы для поддержки электронной почты и интеграции с такими шаблонизаторами, как Velocity, FreeMarker и JasperReports. В этом модуле также упакованы классы для интеграции с различными библиотеками выполнения и планирования заданий, в том числе CommonJ и Quartz

Продолжение табл. 2.1

Модуль	Описание
`core`	Основной модуль, требующийся для каждого приложения Spring. В его архивном JAR-файле находятся классы, общие для всех остальных модулей Spring (например, классы для доступа к файлам конфигурации). Здесь можно также найти ряд исключительно полезных служебных классов, которые применяются во всей кодовой базе Spring и которые можно употреблять в своих приложениях
`expression`	Содержит все классы для поддержки SpEL (Spring Expression Language — язык выражений Spring)
`instrument`	В этот модуль входит агент инструментального оснащения Spring для начальной загрузки виртуальной машины JVM. Его архивный JAR-файл непременно потребуется в приложении Spring для привязывания во время загрузки с помощью библиотеки AspectJ
`jdbc`	В этот модуль входят все классы, предназначенные для поддержки JDBC. Он необходим для всех приложений, которым требуется доступ к базам данных. В этот модуль упакованы классы для поддержки источников данных, типов данных JDBC, шаблонов JDBC, платформенно-ориентированных подключений JDBC и т.д.
`jms`	В этот модуль входят все классы, предназначенные для поддержки системы JMS
`messaging`	Содержит ключевые абстракции, заимствованные из проекта Spring Integration и служащие основанием для приложений, ориентированных обмен сообщениями. Он внедряет поддержку сообщений по протоколу STOMP
`orm`	Расширяет стандартный набор функциональных средств JDBC в Spring, поддерживая распространенные инструментальные средства ORM, в том числе Hibernate, JDO, JPA, а также преобразователь данных iBATIS. Многие классы из архивного JAR-файла этого модуля зависят от классов, содержащихся в архивном JAR-файле модуля **`spring-jdbc`**, поэтому его следует включать в свои приложения
`oxm`	Обеспечивает поддержку OXM (Object/XML Mapping — взаимное преобразование объектов и данных формата XML). В этот модуль упакованы классы, предназначенные для абстрагирования маршализации и демаршализации данных формата XML, а также для поддержки таких распространенных инструментальных средств, как Castor, JAXB, XMLBeans и XStream
`test`	Как упоминалось ранее, в Spring предоставляется ряд имитирующих классов, оказывающих помощь в тестировании приложений. Многие из этих классов используются в тестовом наборе Spring, а следовательно, они хорошо проверены и значительно упрощают тестирование разрабатываемых приложений. С одной стороны, в модульных

Окончание табл. 2.1

Модуль	Описание
	тестах веб-приложений интенсивно применяются имитирующие классы **HttpServletRequest** и **HttpServletResponse**. А с другой стороны, Spring обеспечивает тесную интеграцию со средой модульного тестирования JUnit, и в этом модуле предоставляются многие классы, поддерживающие разработку тестовых сценариев JUnit; например, класс **SpringJUnit4ClassRunner** предоставляет простой способ начальной загрузки контекста типа **ApplicationContext** в среду модульного тестирования
tx	Предоставляет все классы, предназначенные для поддержки инфраструктуры транзакций в Spring. Здесь можно найти классы из уровня абстракции транзакций, поддерживающие прикладной интерфейс Java Transaction API (JTA), а также интеграцию с серверами приложений от ведущих производителей
web	Содержит основные классы для применения Spring в веб-приложениях, в том числе классы для автоматической загрузки контекста типа **ApplicationContext**, классы для поддержки выгрузки файлов и ряд полезных классов для выполнения таких повторяющихся заданий, как извлечение целочисленных значений из строки запроса
web-reactive	Содержит базовые интерфейсы и классы для модели реактивного веб-программирования в Spring
web-mvc	Содержит все классы для собственного каркаса по проектному шаблону MVC в Spring. А если применяется отдельный каркас по шаблону MVC, то классы из архивного JAR-файла этого модуля не требуются. Более подробно модуль Spring MVC рассматривается в главе 16
websocket	Обеспечивает поддержку прикладного интерфейса Java API для протокола WebSocket (JSR-356)

Выбор модулей для приложения

Без инструментального средства управления зависимостями, подобного Maven или Gradle, выбор модулей для применения в разрабатываемом приложении может оказаться затруднительным. Так, если требуется лишь фабрика компонентов Spring Beans и поддержка внедрения зависимостей, то все равно потребуются такие модули, как spring-core, spring-beans, spring-context и spring-aop. Если же требуется поддержка веб-приложений Spring, то придется добавить модуль spring-web и т.д. Благодаря таким функциональным возможностям инструментальных средств сборки, как поддержка транзитивных зависимостей в Maven, все обязательные сторонние библиотеки будут включены в разрабатываемое приложение автоматически.

Доступ к модулям Spring в хранилище Maven

Проект Maven[3], основанный организацией Apache Software Foundation, стал одним из самых распространенных инструментальных средств управления зависимостями для приложений на Java, которые охватывают как среды с открытым кодом, так и корпоративные среды. Это весьма эффективное средство для сборки, упаковки и управления зависимостями приложений, поддерживающее полный цикл сборки приложения, начиная с обработки ресурсов и компиляции и кончая тестированием и упаковкой. Кроме того, существует большое разнообразие модулей, подключаемых к Maven для решения разных задач, включая обновление баз данных и развертывание упакованного приложения на конкретном сервере (например, Tomcat, Jboss или WebSphere). На момент написания этой книги текущей была версия Maven 3.3.9.

Практически во всех проектах с открытым кодом поддерживается распространение их библиотек через хранилище Maven. Наиболее распространено хранилище Maven Central, размещаемое на сервере Apache Software Foundation. А на веб-сайте Maven Central[4] можно осуществлять поиск артефактов и получать сведения о них. После загрузки и установки Maven в среде разработки хранилище Maven Central становится доступным автоматически. Ряд других сообществ разработчиков открытого кода (например, JBoss и Spring от компании Pivotal) также предоставляют своим пользователям доступ к своим хранилищам Maven. Но для доступа к таким хранилищам их придется добавить в файл настроек Maven или файл POM (Project Object Model — объектная модель проекта) своего проекта.

Подробное обсуждение Maven выходит за рамки данной книги, но вы можете всегда обратиться к оперативно доступной документации или соответствующей литературе, чтобы получить дополнительную информацию. Тем не менее здесь ст*о*ит хотя бы вкратце упомянуть структуру упаковки разрабатываемого проекта в хранилище Maven по причине столь широкого распространения Maven.

С каждым артефактом Maven связаны идентификатор группы, идентификатор артефакта, тип упаковки и версия. Например, для артефакта log4j идентификатором группы является `log4j`, идентификатором артефакта — `log4j`, а типом упаковки — `jar`. Далее следует номер версии. Так, для версии 1.2.17 файл артефакта будет иметь имя `log4j-1.2.17.jar` и располагаться в папке для конкретного идентификатора группы, идентификатора артефакта и версии. Файлы конфигурации Maven составлены в формате XML и должны соблюдать стандартный синтаксис, определенный в схеме, доступной по адресу `http://maven.apache.org/xsd/maven-4.0.0.xsd`. По умолчанию файлу конфигурации Maven для разрабатываемого проекта присваивается имя `om.xml`, а ниже приведено его примерное содержимое.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```

[3] См. `http://maven.apache.org`.

[4] См. `http://search.maven.org`.

```
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.apress.prospring5.ch02</groupId>
  <artifactId>hello-world</artifactId>
  <packaging>jar</packaging>
  <version>5.0-SNAPSHOT</version>
  <name>hello-world</name>
  <properties>
    <project.build.sourceEncoding>UTF-8
    </project.build.sourceEncoding>
    <spring.version>5.0.0.RELEASE</spring.version>
  </properties>
  <dependencies>
    <!-- https://mvnrepository.com/artifact/log4j/log4j -->
    <dependency>
       <groupId>log4j</groupId>
       <artifactId>log4j</artifactId>
       <version>1.2.17</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
      ...
      </plugin>
    </plugins>
  </build>
</project>
```

Кроме того, в Maven определяется стандартная структура типичного проекта, как показано на рис. 2.1.

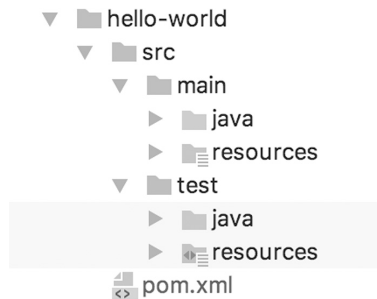

Рис. 2.1. Структура типичного проекта в Maven

В каталоге `main/java` находятся основные классы, а в каталоге `main/resources` — файлы конфигурации приложения. В каталоге `test/java` находятся тестовые классы, а в каталоге `test/resources` — файлы конфигурации для тестирования конкретного приложения из каталога `main`.

Доступ к модулям Spring из Gradle

Стандартная структура проекта в Maven, а также разделение и организация артефактов по категориям очень важны, поскольку в Gradle соблюдаются те же самые правила и даже используется центральное хранилище Maven для извлечения артефактов. Gradle является весьма эффективным инструментальным средством сборки, для конфигурации которого ради простоты и гибкости вместо громоздкой XML-разметки применяется синтаксис языка Groovy. На момент написания данной книги текущей была версия Gradle 4.0[5]. Начиная с версии Spring 4.*x*, разработчики перешли на Gradle для конфигурирования каждого продукта Spring. Именно поэтому исходный код примеров из этой книги может быть построен и выполнен и средствами Gradle. По умолчанию файлу конфигурации Gradle для разрабатываемого проекта присваивается имя `pom.xml`, а ниже приведено его примерное содержимое.

```
group 'com.apress.prospring5.ch02'
version '5.0-SNAPSHOT'

apply plugin: 'java'

repositories {
  mavenCentral()
}

ext{
  springVersion = '5.0.0.RELEASE'
}

tasks.withType(JavaCompile) {
  options.encoding = "UTF-8"
}

dependencies {
  compile group: 'log4j', name: 'log4j', version: '1.2.17'
  ...
}
```

Как видите, содержимое такого файла конфигурации оказывается более удобочитаемым. Артефакты определяются в нем с помощью идентификаторов группы, артефакта и номера версии, как это было принято ранее в Maven, хотя имена свойств отличаются. Но поскольку рассмотрение Gradle выходит за рамки данной книги, на этом придется его завершить.

[5] Подробные сведения о загрузке, установке и конфигурировании Gradle для целей разработки можно найти на официальном веб-сайте данного проекта по адресу `https://gradle.org/install`.

Пользование документацией на Spring

Одна из отличительных черт Spring, которая делает этот каркас столь удобным для разработчиков, строящих реальные приложения, состоит в обилии грамотно и аккуратно написанной документации. В каждом выпуске Spring Framework команда, отвечающая за составление документации, старается довести всю документацию до состояния полной готовности, после чего она уточняется командой разработки. Это означает, что каждое функциональное средство Spring не только полностью документировано с помощью утилиты Javadoc, но и описано в справочном руководстве, включаемом в каждый выпуск. Если вы еще не знакомы с документацией в формате Javadoc и справочным руководством по Spring, то теперь самое время это сделать. Ведь эта книга не в состоянии заменить любой из упомянутых выше ресурсов и служит лишь дополняющим справочным пособием, где демонстрируются особенности построения приложений Spring с самого начала.

Внедрение Spring в приложение “Hello World!”

Надеемся, что к этому моменту вы уже осознали, что Spring является серьезным, хорошо поддерживаемым проектом, обладающим всем необходимым для того, чтобы стать отличным инструментальным средством для разработки приложений. Но мы пока еще не привели ни одного примера исходного кода. И вы, вероятно, с нетерпением ждете того момента, когда каркас Spring будет продемонстрирован в действии, а поскольку это невозможно сделать, не написав исходный код, то займемся этим вплотную. Не отчаивайтесь, если вы не сразу поймете приведенный далее исходный код; по ходу изложения материала будут предоставлены дополнительные пояснения.

Построение примера приложения “Hello World!”

Вы определенно должны быть знакомы с традиционным для программирования примером “Hello World!” (Здравствуй, мир), но на тот случай, если вы не в курсе дела, ниже приведен его исходный код на Java во всей своей красе.

```
package com.apress.prospring5.ch2;

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

Этот пример очень прост: он делает свое дело, но совсем не пригоден для расширения. Что, если требуется изменить сообщение, выводимое на консоль? А что, если сообщение требуется выводить разными способами, например, в стандартный поток вывода ошибок вместо стандартного потока вывода данных или заключить его в дескрипторы HTML-разметки, а не представлять простым текстом?

Итак, переопределим требования к данному примеру приложения, указав, что в нем должен поддерживаться простой и гибкий механизм изменения выводимого сообщения, а также возможность легко изменить режим воспроизведения. В первоначальном примере приложения "Hello World!" оба изменения можно сделать быстро и легко, внеся соответствующие поправки в исходный код. Но более крупное приложение потребует больше времени на перекомпиляцию и повторное тестирование. Поэтому более удачное решение предусматривает вынесение содержимого сообщения наружу и его чтение во время выполнения — возможно, из аргументов командной строки, как демонстрируется в следующем фрагменте кода:

```
package com.apress.prospring5.ch2;

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

В данном примере мы добились то, чего хотели: теперь можно изменять сообщение, не меняя исходный код. Но в рассматриваемом здесь приложении по-прежнему остается следующее затруднение: компонент, отвечающий за воспроизведение сообщения, отвечает также и за его получение. Изменение порядка получения выводимого сообщения, по существу, означает изменение кода воспроизведения. К этому следует добавить еще и тот факт, что мы по-прежнему не можем так просто сменить средство воспроизведения, поскольку для этого придется внести коррективы в класс, запускающий данное приложение на выполнение.

В порядке дальнейшего совершенствования рассматриваемого здесь приложения следует отметить, что лучшее решение предполагает реорганизацию кода с целью вынести логику воспроизведения и получения сообщений в отдельные компоненты. А для того чтобы сделать данное приложение действительно гибким, в этих компонентах должны быть реализованы интерфейсы, с помощью которых определяются взаимозависимости между компонентами и классом, запускающим данное приложение на выполнение. Реорганизовав код, реализующий логику получения сообщений, можно определить простой интерфейс `MessageProvider` с единственным методом `getMessage()`:

```
package com.apress.prospring5.ch2.decoupled;

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

Интерфейс MessageRenderer реализуется во всех компонентах, способных воспроизводить сообщения. И один из таких компонентов приведен в следующем фрагменте кода:

```
package com.apress.prospring5.ch2.decoupled;

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

Как видите, в интерфейсе MessageRenderer определен метод render(), а также метод setMessageProvider() в стиле компонентов JavaBeans. Любые реализации интерфейса MessageRenderer отделены от получения сообщений и поручают эту обязанность интерфейсу MessageProvider, с которым они поставляются. Здесь интерфейс MessageProvider представляет собой зависимость от интерфейса MessageRenderer. Создать простые реализации этих интерфейсов совсем не трудно, как показано в следующем фрагменте кода:

```
package com.apress.prospring5.ch2.decoupled;

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

Как видите, мы создали простую реализацию интерфейса MessageProvider, всегда возвращающую в качестве сообщения символьную строку "Hello World!". Создать класс StandardOutMessageRenderer для воспроизведения этой строки так же просто, как показано ниже.

```
package com.apress.prospring5.ch2.decoupled;

public class StandardOutMessageRenderer
       implements MessageRenderer {
  private MessageProvider messageProvider;

  @Override
  public void render() {
    if (messageProvider == null) {
      throw new RuntimeException("You must set the "
            + "property messageProvider of class:"
            + StandardOutMessageRenderer.class.getName());
            // Установите свойство messageProvider
            // в данном классе
```

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

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

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

Теперь осталось лишь переписать метод `main()` в главном классе данного приложения:

```
package com.apress.prospring5.ch2.decoupled;

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

Абстрактная схема построенного до сих пор приложения приведена на рис. 2.2.

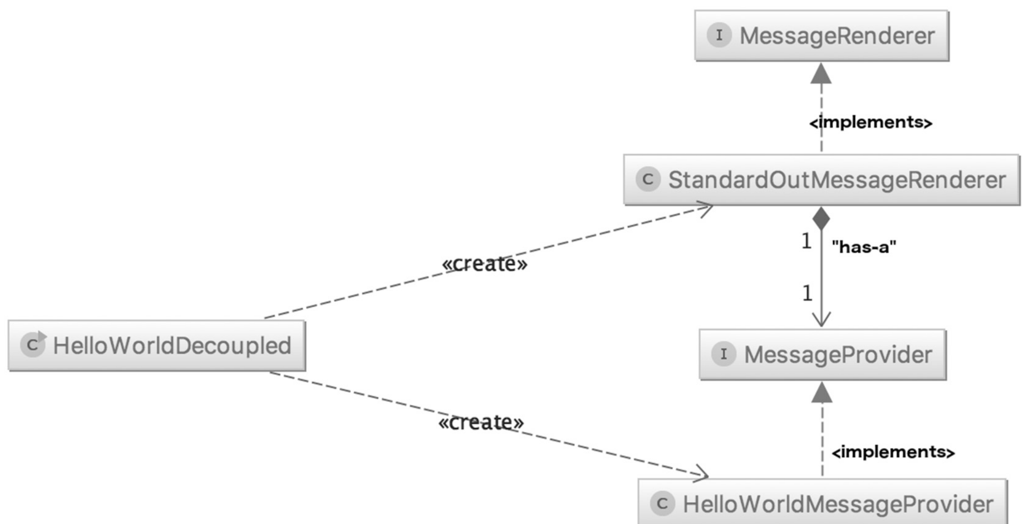

Рис. 2.2. Несколько более развязанное приложение “Hello World!”

Приведенный выше код довольно прост. Сначала в нем получаются экземпляры типа HelloWorldMessageProvider и StandardOutMessageRenderer, хотя объявленными оказываются типы MessageProvider и MessageRenderer соответственно. Дело в том, что взаимодействовать в этом коде требуется только с методами, предоставляемыми этими интерфейсами, а в классах HelloWorldMessageProvider и StandardOutMessageRenderer эти интерфейсы уже реализованы. Затем объект класса, реализующего интерфейс MessageProvider, передается экземпляру типа MessageRenderer и далее вызывается метод MessageRenderer.render(). Скомпилировав и запустив данное приложение на выполнение, мы получим вполне предсказуемый результат: вывод символьной строки "Hello World!" на консоль.

Теперь рассматриваемый здесь пример приложения больше похож на то, к чему мы стремимся, но осталось одно небольшое затруднение. Изменение реализации любого из интерфейсов MessageRenderer или MessageProvider означает изменение в исходном коде. В качестве выхода из этого затруднительного положения можно создать простой фабричный класс, в котором имена классов реализации читаются из файла свойств, а их экземпляры получаются от имени данного приложения:

```
package com.apress.prospring5.ch2.decoupled;
import java.util.Properties;

public class MessageSupportFactory {
  private static MessageSupportFactory instance;

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

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

    try {
      props.load(this.getClass().getResourceAsStream(
                             "/msf.properties"));
      String rendererClass = props.getProperty(
                             "renderer.class");
      String providerClass = props.getProperty(
                             "provider.class");

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

```
  static {
    instance = new MessageSupportFactory();
  }

  public static MessageSupportFactory getInstance() {
    return instance;
  }

  public MessageRenderer getMessageRenderer() {
    return renderer;
  }

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

Приведенная выше реализация элементарна и несколько наивна, обработка ошибок упрощена, а имя файла конфигурации жестко закодировано, но мы уже имеем значительный объем кода. Файл конфигурации этого фабричного класса очень прост:

```
renderer.class= com.apress.prospring5.ch2.decoupled
                   .StandardOutMessageRenderer
provider.class= com.apress.prospring5.ch2.decoupled
                   .HelloWorldMessageProvider
```

Чтобы воспользоваться приведенной выше реализацией, необходимо внести снова коррективы в метод `main()`, как показано ниже.

```
package com.apress.prospring5.ch2.decoupled;

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

Прежде чем перейти к внедрению Spring в данное приложение, напомним, что было сделано раньше. Начав с простого приложения “Hello World!”, мы определили два дополнительных требования, которым должно удовлетворять данное приложение. Первое требование: изменение сообщения должно осуществляться просто, а второе требование: изменение механизма воспроизведения должно быть столь же простым. Чтобы удовлетворить этим требованиям, мы определили два интерфейса: `MessageProvider` и `MessageRenderer`. А для того чтобы получить сообщение для воспро-

изведения, интерфейс `MessageRenderer` полагается на реализацию интерфейса `MessageProvider`. И, наконец, мы определили простой фабричный класс для извлечения имен классов реализации и получения их экземпляров по мере необходимости.

Реорганизация кода средствами Spring

Продемонстрированный выше окончательный пример соответствует целям, намеченным для рассматриваемого здесь приложения, но и он не лишен недостатков. Первый состоит в том, что приходится писать немало связующего кода для соединения всех частей в единое приложение, в то же время сохраняя компоненты слабо связанными. Второй недостаток заключается в том, что мы все еще должны вручную предоставлять реализацию интерфейса `MessageRenderer` вместе с экземпляром реализации интерфейса `MessageProvider`. Оба эти недостатка можно устранить, применяя Spring.

Чтобы устранить недостаток, связанный со слишком большим объемом связующего кода, достаточно полностью удалить фабричный класс `MessageSupportFactory` из данного приложения и заменить его интерфейсом `ApplicationContext` из Spring, как показано ниже. В отношении этого интерфейса пока что достаточно знать, что он применяется в Spring для сохранения всей информации о среде, относящейся к приложению, которым управляет каркас Spring. Этот интерфейс расширяет другой интерфейс `ListableBeanFactory`, действующий в качестве поставщика для любого экземпляра компонентов Spring Beans.

```
package com.apress.prospring5.ch2;

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

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

Как следует из приведенного выше фрагмента кода, в теле метода `main()` сначала получается экземпляр класса `ClassPathXmlApplicationContext` (сведения о конфигурации данного приложения загружаются из файла `spring/app-context.xml`, находящегося по пути к классам текущего проекта), типизированный как `ApplicationContext`, а затем из этого экземпляра получаются экземпляры реализации интерфейса `MessageRenderer` с помощью метода `ApplicationContext.`

getBean(). Не обращайте пока что особого внимания на метод getBean(); достаточно знать, что он читает конфигурацию приложения (в данном случае — из XML-файла app-context.xml), инициализирует среду интерфейса ApplicationContext (по существу, контекст приложения Spring), а затем возвращает экземпляр сконфигурированного компонента Spring Bean[6]. Этот XML-файл служит тем же целям, что и аналогичный файл для фабричного класса MessageSupportFactory, 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:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation=
        "http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans
               /spring-beans.xsd">

  <bean id="provider"
      class="com.apress.prospring5.ch2.decoupled
                .HelloWorldMessageProvider"/>
  <bean id="renderer"
      class="com.apress.prospring5.ch2.decoupled
                .StandardOutMessageRenderer"
      p:messageProvider-ref="provider"/>
</beans>
```

Выше приведена типичная конфигурация контекста типа ApplicationContext в Spring. Сначала в ней объявляется пространство имен Spring, а им является стандартное пространство имен beans, которое служит для объявления управляемых компонентов Spring Beans, а также их требований к зависимостям. В данном примере свойство messageProvider компонента, реализующего средство воспроизведения (renderer), ссылается на компонент, реализующий поставщика (provider). Все эти зависимости должны быть разрешены и внедрены средствами Spring.

Далее объявляется компонент Spring Bean с идентификатором provider и соответствующий класс реализации. Обнаружив такое определение компонента Spring Bean во время инициализации интерфейса ApplicationContext, каркас Spring получает экземпляр заданного класса и сохраняет его с указанным идентификатором.

После этого объявляется компонент Spring Bean с идентификатором renderer и соответствующим классом реализации. Напомним, что при получении сообщения для воспроизведения этот компонент полагается на интерфейс MessageProvider. Чтобы известить Spring о таком требовании к внедрению зависимостей, в данном случае используется атрибут p разметки пространства имен. В частности, атрибут дескриптора p:messageProvider-ref="provider" извещает Spring, что свойство messageProvider одного компонента Spring Bean должно быть внедрено с по-

[6] См. http://search.maven.org.

мощью другого компонента. Внедряемый в это свойство компонент должен иметь идентификатор `provider`. Обнаружив это определение компонента Spring Bean, каркас Spring получает экземпляр заданного класса, находит в данном компоненте свойство `messageProvider` и внедряет его, используя экземпляр компонента с идентификатором `provider`.

Теперь, как видите, после инициализации контекста типа `ApplicationContext` в методе `main()` просто получается компонент Spring Bean типа `MessageRenderer`. С этой целью сначала вызывается типизированный метод `getBean()`, которому передается идентификатор и ожидаемый возвращаемый тип (в данном случае — интерфейса `MessageRenderer`), а затем метод `render()`. А каркас Spring создает реализацию интерфейса `MessageProvider` и внедряет ее в реализацию интерфейса `MessageRenderer`. Обратите внимание на то, что в данном случае никаких изменений не было внесено в классы, связанные вместе с помощью Spring. На самом деле эти классы не имеют никакого отношения к каркасу Spring и находятся в полном неведении относительно его существования. Хотя это не всегда так. Ваши классы могут реализовывать интерфейсы Spring, чтобы взаимодействовать с контейнером внедрения зависимостей самыми разными способами.

А теперь необходимо выяснить, каким образом действуют новая конфигурация Spring и модифицированный метод `main()`. С этой целью воспользуйтесь Gradle и введите приведенную ниже команду из командной строки, чтобы собрать проект и корневой каталог исходного кода.

```
gradle clean build copyDependencies
```

Единственным обязательным модулем Spring, который должен быть объявлен в файле конфигурации, является модуль `spring-context`. Gradle автоматически внедрит любые транзитивные зависимости, требующиеся для данного модуля. Транзитивные зависимости модуля `spring-context` приведены на рис. 2.3.

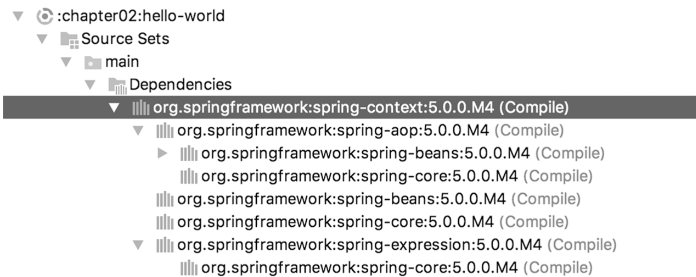

Рис. 2.3. Транзитивные зависимости модуля **`spring-context`**, отображаемые в IDE IntelliJ IDEA

По приведенной выше команде проект собирается заново. При этом удаляются сформированные ранее файлы, а все требующиеся зависимости копируются в то же место, где находится результирующий артефакт, т.е. по пути `build/libs`. Этот же

путь присоединяется в качестве префикса к именам библиотечных файлов, введенных в файл манифеста MANIFEST.MF при построении архивного JAR-файла. Если вы незнакомы с конфигурированием и процессом построения архивного JAR-файла средствами Gradle, обращайтесь за справкой к файлу hello-world/build.gradle свойств Gradle, находящемуся в папке chapter02 исходного кода примеров, доступного для загрузки на веб-сайте издательства Apress по адресу, указанному в конце введения.

И, наконец, чтобы запустить пример реализации внедрения зависимостей в Spring, введите команды

```
cd build/libs; java -jar hello-world-5.0-SNAPSHOT.jar
```

В итоге вы должны увидеть несколько протокольных сообщений, формируемых в процессе начального запуска контейнера Spring, а после них — ожидаемый вывод сообщения "Hello World!".

Конфигурирование Spring с помощью аннотаций

Начиная с версии Spring 3.0, XML-файлы конфигурации больше не требуются для разработки приложений в Spring. Их можно заменить аннотациями и конфигурационными классами. Последние являются классами Java, снабженными аннотацией @Configuration и содержащими определения компонентов Spring Beans, где методы снабжены аннотацией @Bean. Они могут быть сами сконфигурированы для обозначения определений компонентов Spring Beans в приложении с помощью аннотации @ComponentScanning. Ниже приведена конфигурация, равнозначная содержимому XML-файла конфигурации app-context.xml, представленному ранее в этой главе.

```
package com.apress.prospring5.ch2.annotated;

import com.apress.prospring5.ch2.decoupled
         .HelloWorldMessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageProvider;
import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import com.apress.prospring5.ch2.decoupled
         .StandardOutMessageRenderer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class HelloWorldConfiguration {

  // равнозначно разметке <bean id="provider" class=".."/>
  @Bean
  public MessageProvider provider() {
    return new HelloWorldMessageProvider();
  }
```

```
  // равнозначно разметке <bean id="renderer" class=".."/>
  @Bean
  public MessageRenderer renderer() {
    MessageRenderer renderer = new
                    StandardOutMessageRenderer();
    renderer.setMessageProvider(provider());
    return renderer;
  }
}
```

В таком случае в метод `main()` необходимо внести коррективы, заменив класс `ClassPathXmlApplicationContext` другим классом, реализующим интерфейс `ApplicationContext` и способным читать определения компонентов Spring Beans из конфигурационных классов. И таким заменителем является класс `AnnotationConfigApplicationContext`:

```
package com.apress.prospring5.ch2.annotated;

import com.apress.prospring5.ch2.decoupled.MessageRenderer;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation
         .AnnotationConfigApplicationContext;

public class HelloWorldSpringAnnotated {
  public static void main(String... args) {
    ApplicationContext ctx = new
              AnnotationConfigApplicationContext
             (HelloWorldConfiguration.class);
    MessageRenderer mr = ctx.getBean(
                    "renderer", MessageRenderer.class);
    mr.render();
  }
}
```

Это лишь один из вариантов конфигурирования с помощью аннотаций и конфигурационных классов. В отсутствие XML-разметки конфигурирование Spring становится довольно гибким процессом. Подробнее об этом речь пойдет далее, где основное внимание будет уделено конфигурированию на языке Java и с помощью аннотаций.

■ ***На заметку*** Некоторые интерфейсы и классы, определенные в примере приложения “Hello World”, могут использоваться в последующих главах. И хотя в данном примере исходный код был приведен полностью, в других главах могут быть показаны менее полные версии исходного кода, чтобы соблюсти краткость особенно в тех случаях, когда в код вносятся постепенные коррективы. Исходный код примеров из этой книги был сделан более организованным. В частности, все классы, применяемые в последующих примерах, размещены в пакетах **`com.apress.prospring5.ch2.decoupled`** и **`com.apress.prospring5.ch2.annotated`**. Однако исходный код реального приложения должен быть организован соответствующим образом.

Резюме

В этой главе были представлены основные сведения, необходимые для подготовки и приведения в действие каркаса Spring. В ней было показано, как приступить к работе с каркасом Spring, используя системы управления зависимостями, и получить текущую разрабатываемую версию непосредственно из хранилища GitHub. Затем было описано, каким образом упакован каркас Spring, а также перечислены зависимости, требующиеся для его функциональных средств. Располагая этими сведениями, можно принимать обоснованные решения относительно того, какие архивные JAR-файлы Spring требуются для приложения и какие зависимости должны распространяться вместе с ним. Документация на Spring, руководства и тестовый набор служат пользователям Spring идеальным основанием, чтобы приступить к разработке приложений, поэтому часть этой главы была посвящена исследованию тех возможностей, которые становятся доступными благодаря Spring. И, наконец, в этой главе был рассмотрен пример внедрения зависимостей в Spring, где традиционное приложение "Hello World!" было превращено в слабо связанное и расширяемое приложение для воспроизведения сообщений.

Важно понимать, что в данной главе мы лишь слегка коснулись особенностей внедрения зависимостей в частности и каркаса Spring в целом. А в следующей главе мы подробно рассмотрим принципы инверсии управления и внедрения зависимостей в Spring.