Spring Framework 4.x Reference Documentation 中文翻译

waylau

Published with GitBook

Table of Contents

spring-framework-4-reference

Chinese translation of [Spring Framework 4.x Reference Documentation] (http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/). The current version of Spring Framework 4.x is 4.2.1.RELEASE. There is also a GitBook version of the book: http://waylau.gitbooks.io/spring-framework-4-reference. Let's READ!

《Spring Framework 4.x参考文档》中文翻译（包含了官方文档以及其他文章）。至今为止，Spring Framework 的最新版本为 4.2.1.RELEASE。

利用业余时间对此进行翻译，并在原文的基础上，插入配图，图文并茂方便用户理解。如有勘误欢迎指正，点此提问。如有兴趣，也可以参与到本翻译工作中来 :)

从目录开始阅读吧

Contact:

- Blog:www.waylau.com
- Gmail: waylau521@gmail.com
- Weibo: waylau521
- Twitter: waylau521
- Github : waylau

Part I. Overview of Spring Framework 总览

Spring Framework 是一个轻量级的解决方案，可以一站式构建企业级应用。然而，Spring 是模块化的，允许你使用的你需要的部分，而不必把其余带进来。你可以在任何框架之上去使用IOC容器，但你也可以只使用 Hibernate 集成代码 或 JDBC 抽象层。Spring Framework 支持声明式事务管理，通过 RMI 或 Web 服务远程访问你的逻辑，并支持多种选择持久化你的数据。它提供了一个全功能的 MVC 框架，使您能够将 AOP 透明地集成到您的软件。

Spring 的设计是非侵入性的，也就是说你的领域逻辑代码一般对框架本身无依赖性。在你的集成层（如数据访问层），在数据访问技术和 Spring 的库一些依赖将存在。然而，它应该很容易从你的剩余代码中分离这些依赖。

本文档是 Spring Framework 功能的参考指南。如果你有关于这个文档的任何要求，意见或问题，请发送到用户邮件列表。对 Framework 本身的问题应该到 StackOverflow 请问（见https://spring.io/questions）

译者注：对本文档的翻译有任何问题，请在https://github.com/waylau/spring-framework-4-reference/issues上面提问

开始

本参考指南提供了关于 Spring Framework 的详细信息。提供它的所有功能全面的文档，以及Spring所涵盖的一些关于底层方面的背景资料（如“Dependency Injection（依赖注入）”）。

如果你是刚刚开始 Spring，你可能要开始使用 Spring Framework 创建一个基于 Spring Boot 的应用。Spring Boot 提供了一种快速（自用）的方法来创建一个生产准备基于 Spring 的应用程序。它是基于Spring Framework，约定大于配置，为了让你尽快地运行内。

您可以使用 start.spring.io 生产一个基本项目或遵循一个类似Getting Started Building a RESTful Web Service的“入门”指南。这些指南都非常专注于任务，其中大部分是基于 Spring Boot，非常容易理解。这些还涵盖了你可能想解决一个特定问题时要考虑到的 spring 其他项目。

介绍 Spring Framework

Spring Framework 是一个提供完善的基础支持来开发 Java 应用程序的Java 平台。Spring 负责基础功能，让您可以专注于您的应用。

Spring 可以使你从“简单的Java对象”（POJO）构建应用程序，并且将企业服务非侵入性的应用到 POJO。此功能适用于 Java SE 编程模型和完全或者部分的 Java EE 。

举例，作为一个应用程序的开发者，你可以从 Spring 平台获得以下好处：

- 使 Java 方法可以执行数据库事务而不用去处理事务 API。
- 使本地 Java 方法可以执行远程过程而不用去处理远程 API。
- 使本地 Java 方法可以拥有管理操作而不用去处理 JMX API。
- 使本地 Java 方法可以执行消息处理而不用去处理 JMS API。

依赖注入和控制反转

Java 应用程序--运行在各个松散的领域,从受限的嵌入式应用程序,到 n 层架构的服务端企业级应用程序--通常由来自应用适当的对象进行组合合作。因此,对象在应用程序中是彼此依赖。

尽管 Java 平台提供了丰富的应用程序开发功能,但它缺乏来组织基本构建块成为一个完整的方法。这个任务留给了架构师和开发人员。虽然您可以使用设计模式,例如 Factory, Abstract Factory, Builder, Decorator, 和 Service Locator 来组合各种类和对象实例构成应用程序,这些模式是:给出一个最佳实践的名字,描述什么模式,哪里需要应用它,它要解决什么问题,等等。模式是形式化的最佳实践,但你必须在应用程序中自己来实现。

Spring Framework 的 *Inversion of Control* (IoC) 组件旨在通过提供正规化的方法来组合不同的组件成为一个完整的可用的应用。 Spring Framework 将规范化的设计模式作为一等的对象,您可以集成到自己的应用程序。许多组织和机构使用 Spring Framework 以这种方式来开发健壮的、可维护的应用程序。

背景

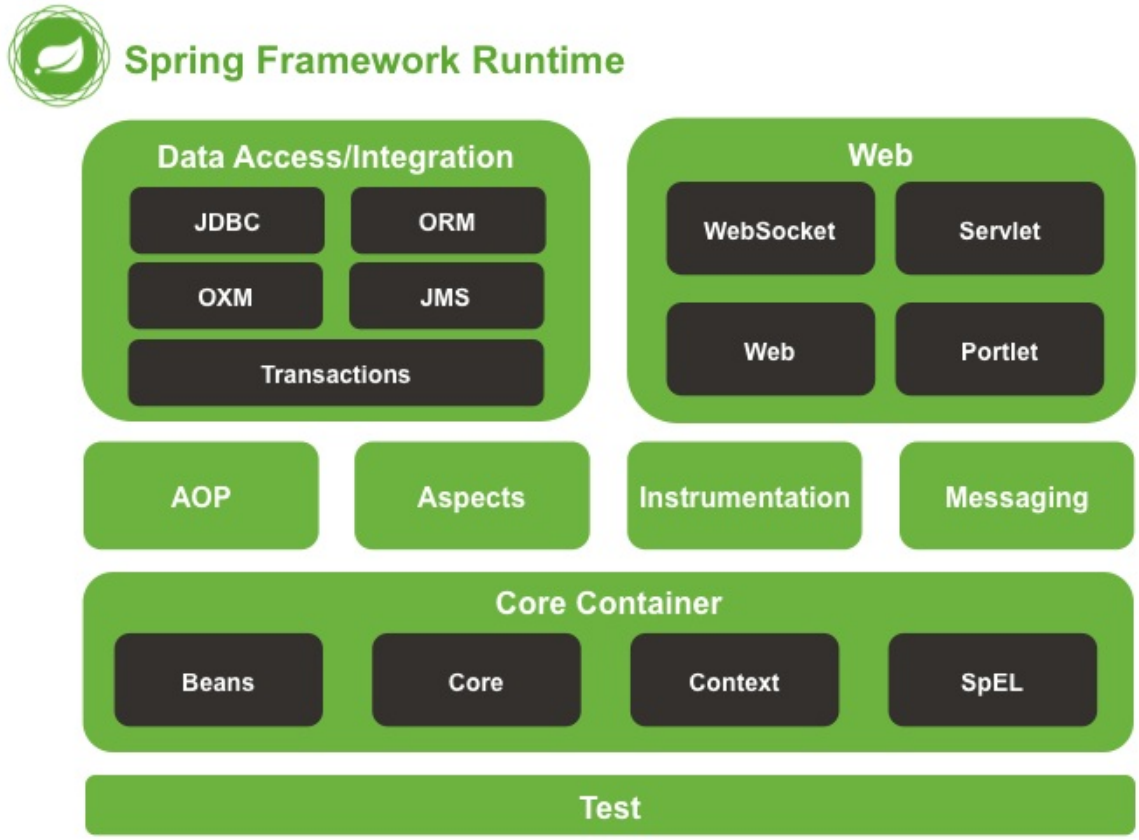
“问题是，*[他们]*哪些方面的控制被反转？”这个问题由 *Martin Fowler*在他的 *Inversion of Control (IoC)* 网站在 *2004* 年提出。*Fowler* 建议重新命名这个说法,使得他更加好理解,并且提出了 *Dependency Injection*（依赖注入）*(*这个新的说法*)*。

译者注：Dependency Injection 和 Inversion of Control 其实就是一个东西的两种不同的说法而已。本质上是一回事。

模块

Spring Framework 的功能被组织成了 20 来个模块。这些模块分成 Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, Messaging, 和 Test，如下图：

Figure 2.1. Overview of the Spring Framework



下面章节会列出可用的模块，名称与功能及他们的主题相关。组件名称与组件的 ID 相关，用于2.3.1节中的依赖管理工具。

核心容器

Core Container 由 spring-core, spring-beans, spring-context, spring-context-support, 和 spring-expression (Spring Expression Language) 模块组成

spring-core 和 spring-beans 提供框架的基础部分，包括 IoC 和 Dependency Injection 功能。BeanFactory 是一个复杂的工厂模式的实现。不需要可以编程的单例，并允许您将配置和特定的依赖从你的实际程序逻辑中解耦。

Context (spring-context) 模块建立且提供于在Core 和 Beans 模块的基础上，它是一种在框架类型下实现对象存储操作的手段，有一点像 JNDI 注册。Context 继承了 Beans 模块的特性，并且增加了对国际化的支持（例如用在资源包中）、事件广播、资源加载和创建上下文（例如 一个 Servlet 容器）。Context 模块也支持例如 EJB，JMX 和基础远程这样的 JavaEE 特性。ApplicationContext 是 Context 模块的焦点。spring-context-support 提供对常见第三方库的支持集成进 Spring 应用上下文，如缓存 (EhCache, Guava, JCache), 通信 (JavaMail), 调度 (CommonJ, Quartz) 和 模板引擎 (FreeMarker, JasperReports, Velocity).

spring-expression 模块提供了一个强大的Expression Language（表达式语言）用来在运行时查询和操作对象图。这是作为

JSP2.1 规范所指定的统一表达式语言（unified EL）的一种延续。这种语言支持对属性值、属性参数、方法调用、数组内容存储、收集器和索引、逻辑和算数操作及命名变量，并且通过名称从 Spring 的控制反转容器中取回对象。表达式语言模块也支持 List 的映射和选择，正如像常见的列表汇总一样。

AOP 及 Instrumentation

spring-aop 模块提供 AOP Alliance-compliant（联盟兼容）的面向切面编程实现，允许你自定义，比如，方法拦截器和切入点完全分离代码。使用源码级别元数据的功能，你也可以在你的代码中加入 behavioral information (行为信息)，在某种程度上类似于 .NET 属性。

单独的 spring-aspects 模块提供了集成使用 AspectJ。

spring-instrument 模块提供了类 instrumentation 的支持和在某些应用程序服务器使用类加载器实现。spring-instrument-tomcat 用于 Tomcat Instrumentation 代理。

消息

Spring Framework 4 包含了 spring-messaging 模块，从 Spring 集成项目中抽象出来，比如 Message, MessageChannel, MessageHandler 及其他用来提供基于消息的基础服务。该模块还包括一组消息映射方法的注解，类似于基于编程模型 Spring MVC 的注解。

数据访问/集成

Data Access/Integration 层由 JDBC, ORM, OXM, JMS, 和 Transaction 模块组成。

spring-jdbc 模块提供了不需要编写冗长的JDBC代码和解析数据库厂商特有的错误代码的JDBC-抽象层。

spring-tx 模块的支持可编程和声明式事务管理，用于实现了特殊的接口的和你所有的POJO类（Plain Old Java Objects）。

spring-orm 模块提供了流行的 object-relational mapping（对象-关系映射）API集成层，其包含 JPA，JDO，Hibernate。使用ORM包，你可以使用所有的 O/R 映射框架结合所有Spring 提供的特性，比如前面提到的简单声明式事务管理功能。

spring-oxm 模块提供抽象层用于支持 Object/XML mapping （对象/XML映射）的实现,如 JAXB、Castor、XMLBeans、JiBX 和 XStream 的。

spring-jms 模块（Java Messaging Service）包含生产和消费信息的功能。 从 Spring Framework 4.1 开始提供集成 spring-messaging 模块。

2.2.5 Web

Web 层由 spring-web, spring-webmvc, spring-websocket, 和 spring-webmvc-portlet 组成 。

spring-web 模块提供了基本的面向 web 开发的集成功能，例如多方文件上传、使用 Servlet listeners 和 Web 开发应用程序上下文初始化 IoC 容器。它也包含 HTTP 客户端以及 Spring 远程访问的支持的 web 相关的部分。

spring-webmvc 模块（也被称为 Web Servlet 模块）包含 Spring 的model-view-controller （模型-视图-控制器（MVC）和 REST Web Services 实现的Web应用程序。Spring 的 MVC 框架提供了domain model（领域模型）代码和 web form (网页) 之间的完全分离，并且集成了 Spring Framework 所有的其他功能。

spring-webmvc-portlet 模块（也被称为 Web-Portlet 模块）提供了MVC 模式的实现是用一个 Portlet 的环境和 spring-webmvc 模块功能的镜像。

2.2.6 Test

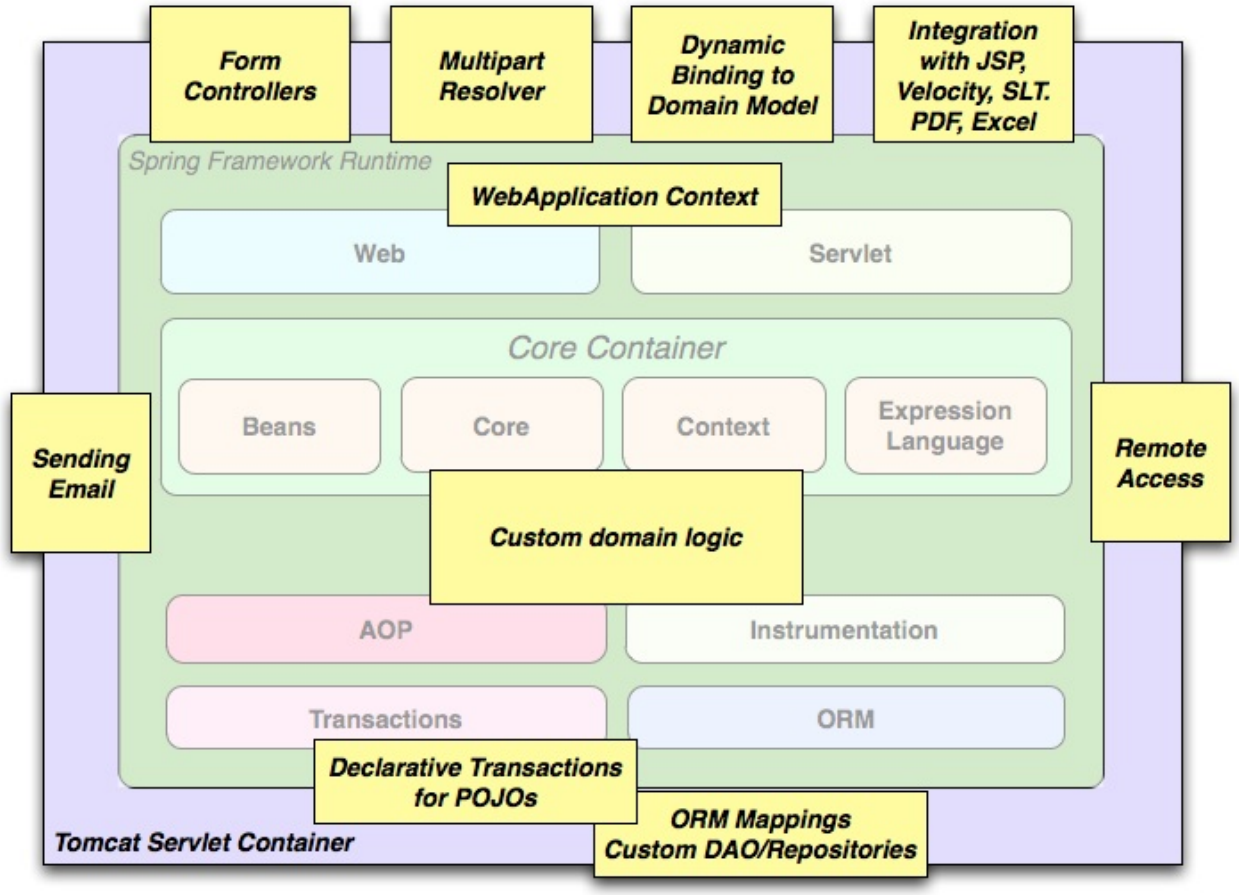
spring-test 模块支持通过组合 JUnit 或 TestNG 来进行单元测试和集成测试 。它提供了连续的加载 ApplicationContext 并且

缓存这些上下文。它还提供了 mock object（模仿对象），您可以使用隔离测试你的代码。

使用场景

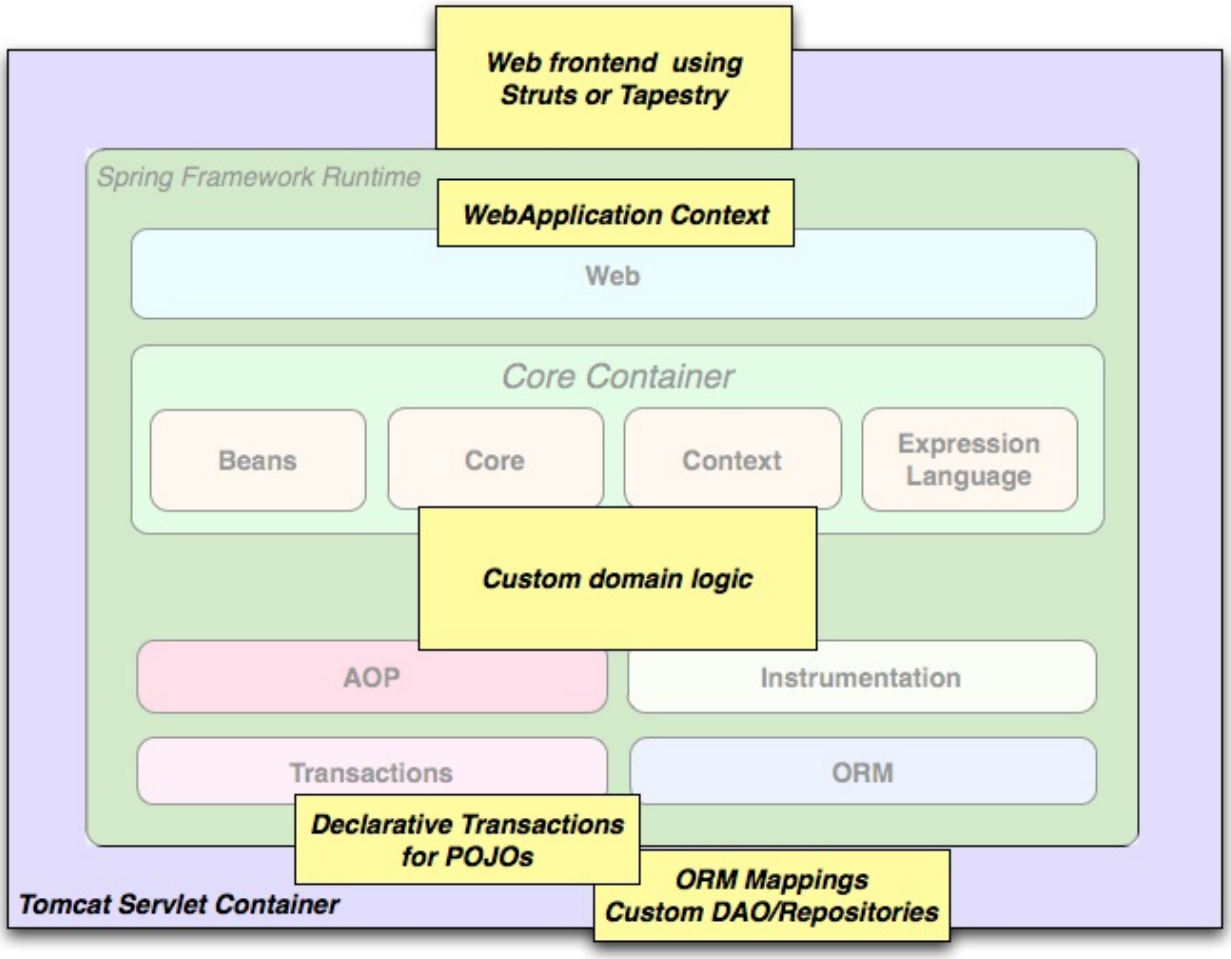
前面的构建模块描述在很多的情况下使 Spring 是一个合理的选择，从资源受限的嵌入式程序到成熟的企业应用程序都可以使用 Spring 事务管理功能和 web 框架集成。

Figure 2.2. Typical full-fledged Spring web application



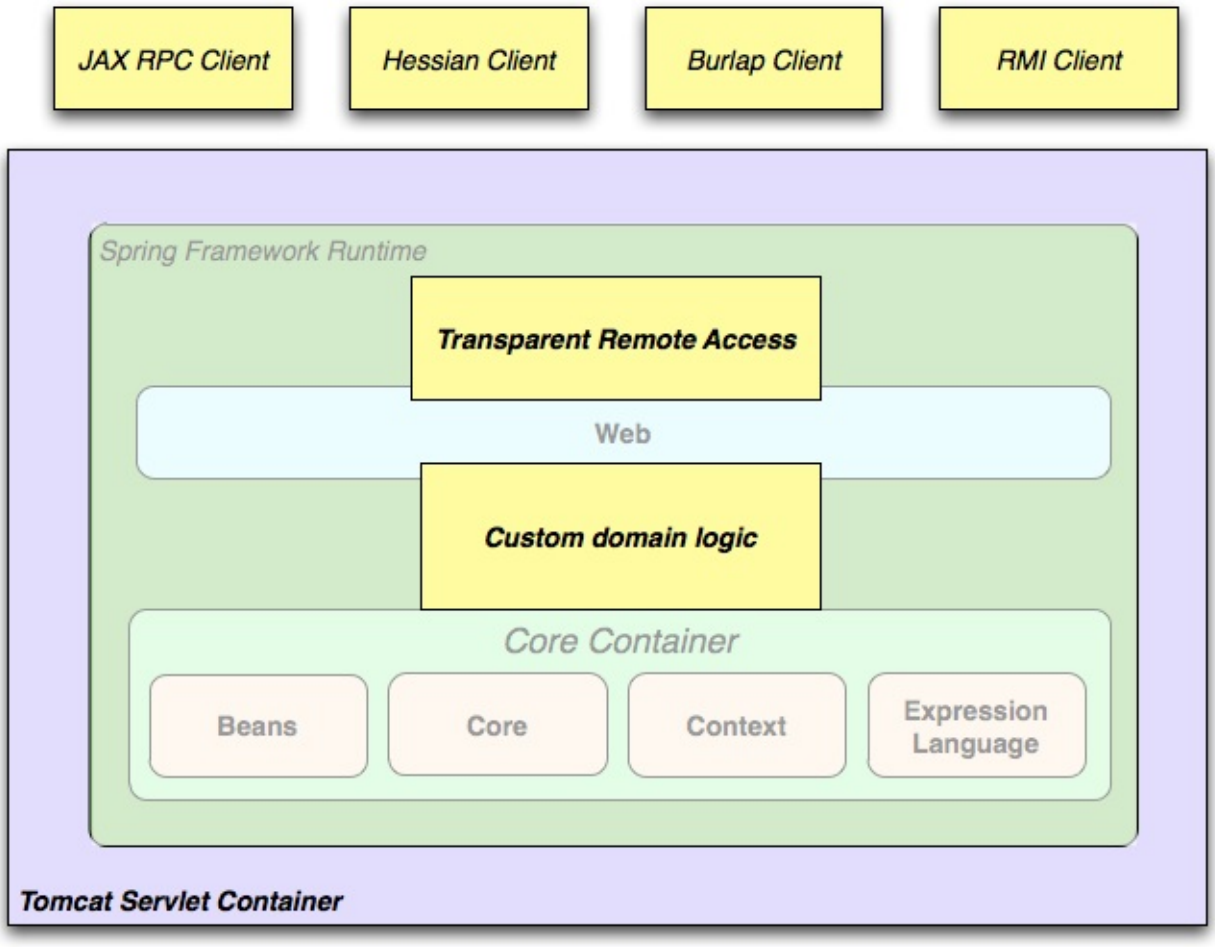
Spring 声明式事务管理特性使 Web 应用程序拥有完全的事务，就像你使用 EJB 容器管理的事务。所有的自定义业务逻辑可以用简单的 POJO 实现，用 Spring 的 IoC 容器进行管理。额外的服务包括发送电子邮件和验证，是独立的网络层的支持，它可以让你选择在何处执行验证规则。Spring 的 ORM 支持集成 JPA，Hibernate，JDO；例如，当使用 Hibernate，您可以继续使用现有的映射文件和 标准的 Hibernate 的 SessionFactory 配置。表单控制器将 Web 层和领域模型无缝集成，消除 ActionForms 或其他类用于变换 HTTP 参数成为您的域模型值的需要。

Figure 2.3. Spring middle-tier using a third-party web framework



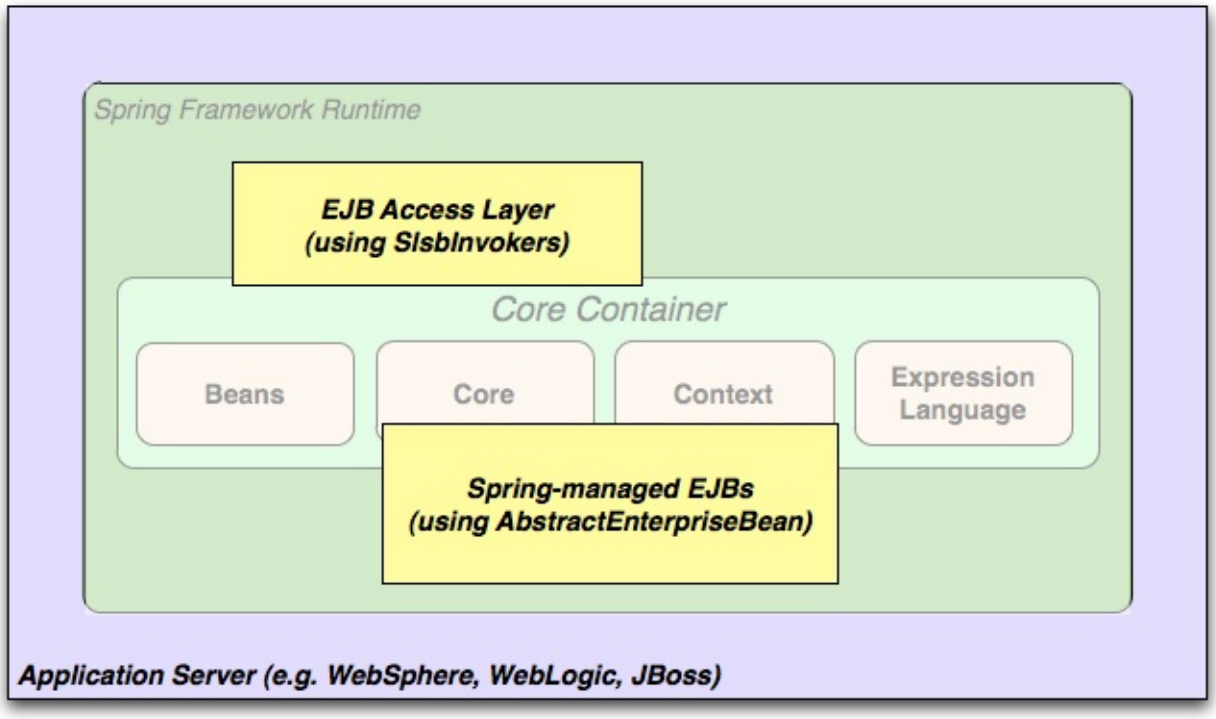
有时，不允许你完全切换到一个不同的框架。Spring Framework 不强制使用它,它不是一个全有或全无的解决方案。现有的前端 Struts, Tapestry, JSF 或其他 UI 框架，可以集成一个基于 Spring 中间件，它允许你使用 Spring 事务的功能。你只需要将业务逻辑连接使用 ApplicationContext 和使用WebApplicationContext 集成到你的 web 层。

Figure 2.4. Remoting usage scenario



当你需要通过 web 服务访问现有代码时，可以使用 Spring 的 Hessian-, Burlap-, Rmi- 或 JaxRpcProxyFactory 类。启用远程访问现有的应用程序并不难。

Figure 2.5. EJBs - Wrapping existing POJOs



Spring Framework 也提供了 Enterprise JavaBeans 访问和抽象层 使你能重用现有的 POJOs,并且在需要声明安全的时候打包他们成为无状态的 bean 用于可伸缩，安全的 web 应用里。

依赖关系管理和命名约定

依赖管理和依赖注入是不同的概念。为了让 Spring 的这些不错的功能运用到运用程序中（比如依赖注入），你需要收集所有的需要的库（JAR文件），并且在编译、运行的时候将它们放到你的类路径中。这些依赖不是虚拟组件的注入，而是物理的资源在文件系统中（通常）。依赖管理过程包括定位这些资源，存储它们并添加它们到类路径。依赖可以是直接（如我的应用程序运行时依赖于 Spring ），或者是间接（如我的应用程序依赖 commons-dbcp ，而 commons-dbcp 又依赖于 commons-pool）。间接的依赖也被称为 “transitive （传递）”，它是最难识别和管理的依赖。

如果你将使用 Spring，你需要复制哪些包含你需要的 Spring 功能的 jar包。为了使这个过程更加简单，Spring 被打包为一组模块，这些模块尽可能多的分开依赖关系。例如，如果不想写一个 web 应用程序，你就不需要引入 Spring-web 模块。为了在本指南中标记 Spring 库模块我们使用了速记命名约定 spring- 或者 *spring-*.jar ， 其中*代表模块的短名（比如 spring-core,spring-webmvc, spring-jms 等）。实际的jar 文件的名字，通常是用模块名字和版本号级联（如spring-core-4.1.4.BUILD-SNAPSHOT.jar）

每个 Spring Framework 发行版本将会放到下面的位置：

- Maven Central （Maven 中央库），这是 Maven 查询的默认库，而不需要任何特殊的配置就能使用。许多常用的 Spring 的依赖库也存在与Maven Central ， 并且 Spring 社区的很大一部分都使用 Maven 进行依赖管理，所以这是最方便他们的。jar 命名格式是 `spring-*-<version>.jar` ， Maven groupId 是 `org.springframework`
- 公共 Maven 仓库还拥有 Spring 专有的库。除了最终的 GA 版本，这个库还保存开发的快照和里程碑。JAR 文件的名字是和 Maven Central 相同的形式，所以这是让 Spring 的开发版本使用其它部署在 Maven Central 库的一个有用的地方。该库还包含一个用于发布的 zip 文件包含所有Spring jar，方便下载。

所以首先，你要决定用什么方式管理你的依赖，通常建议你使用自动系统像 Maven, Gradle 或 Ivy, 当你也可以下载 jar

下面是 Spring 中的组件列表。更多描述，详见2.2. Modules （模块）

Table 2.1. Spring Framework Artifacts

GroupId	ArtifactId	Description
org.springframework	spring-aop	Proxy-based AOP support
org.springframework	spring-aspects	AspectJ based aspects
org.springframework	spring-beans	Beans support, including Groovy
org.springframework	spring-context	Application context runtime, including scheduling and remoting abstractions
org.springframework	spring-context-support	Support classes for integrating common third-party libraries into a Spring application context
org.springframework	spring-core	Core utilities, used by many other Spring modules
org.springframework	spring-expression	Spring Expression Language (SpEL)
org.springframework	spring-instrument	Instrumentation agent for JVM bootstrapping
	spring-	

org.springframework	instrument-tomcat	Instrumentation agent for Tomcat
org.springframework	spring-jdbc	JDBC support package, including DataSource setup and JDBC access support
org.springframework	spring-jms	JMS support package, including helper classes to send and receive JMS messages
org.springframework	spring-messaging	Support for messaging architectures and protocols
org.springframework	spring-orm	Object/Relational Mapping, including JPA and Hibernate support
org.springframework	spring-oxm	Object/XML Mapping
org.springframework	spring-test	Support for unit testing and integration testing Spring components
org.springframework	spring-tx	Transaction infrastructure, including DAO support and JCA integration
org.springframework	spring-web	Web support packages, including client and web remoting
org.springframework	spring-webmvc	REST Web Services and model-view-controller implementation for web applications
org.springframework	spring-webmvc-portlet	MVC implementation to be used in a Portlet environment
org.springframework	spring-websocket	WebSocket and SockJS implementations, including STOMP support

Spring 的依赖以及基于 Spring

虽然 Spring 提供了集成在一个大范围的企业和其他外部工具的支持，它故意保持其强制性依赖关系降到最低：在简单的用例里，你无需查找并下载（甚至自动）一大批 jar 库来使用 Spring 。基本的依赖注入只有一个外部强制性的依赖，这是用来做日志的（见下面更详细地描述日志选项）。

接下来我们将一步步展示如果配置依赖 Spring 的程序，首先用 Maven 然后用 Gradle 和最后用 Ivy。在所有的情况下，如果有不清楚的地方，查看的依赖性管理系统的文档，或看一些示例代码。Spring 本身是使用 Gradle 来管理依赖的，我们的很多示例也是使用 Gradle 或 Maven。

译者注：有关 Gradle 的使用，可以参见笔者的另外一部作品《Gradle 2 用户指南》

Maven 依赖管理

如果您使用的是 Maven 的依赖管理你甚至不需要明确提供日志依赖。例如，要创建一个应用程序的上下文和使用依赖注入来配置应用程序，你的Maven 依赖将看起来像这样：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.2.1.RELEASE</version>
```

```
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

就是它。注意 scope 可声明为 runtime （运行时）如果你不需要编译 Spring 的API，这通常是基本的依赖注入使用的案例。

以上与 Maven Central 存储库工程实例。使用 Spring Maven 存储库（如里程碑或开发者快照），你需要在你的 Maven 配置指定的存储位置。如下：

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.release</id>
        <url>http://repo.spring.io/release/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

里程碑：

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.milestone</id>
        <url>http://repo.spring.io/milestone/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

以及快照：

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.snapshot</id>
        <url>http://repo.spring.io/snapshot/</url>
        <snapshots><enabled>true</enabled></snapshots>
    </repository>
</repositories>
```

Maven "Bill Of Materials" 依赖

有可能不小心使用不同版本的 Spring JAR 在使用 Maven 时。例如，你可能发现一个第三方的库，或另一 Spring 的项目，拉取了一个在传递依赖较早的发布包。如果你自己忘记了显式声明一个直接依赖，各种意想不到的问题出现。

为了克服这些问题，Maven 支持 "bill of materials" (BOM) 的依赖的概念。你可以在你的 dependencyManagement 部分引入 spring-framework-bom 来确保所有 spring依赖（包括直接和传递的）是同一版本。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-framework-bom</artifactId>
            <version>4.2.1.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

使用 BOM 后，当依赖 Spring Framework 组件后，无需指定 `<version>` 属性

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
    </dependency>
<dependencies>
```

Gradle 依赖管理

用 Gradle 来使用 Spring ,在 repositories 中填入适当的 URL :

```
repositories {
    mavenCentral()
    // and optionally...
    maven { url "http://repo.spring.io/release" }
}
```

可以适当修改 URL 从 /release 到 /milestone 或 /snapshot 。库一旦配置，就能声明 Gradle 依赖:

```
dependencies {
    compile("org.springframework:spring-context:4.2.1.RELEASE")
    testCompile("org.springframework:spring-test:4.2.1.RELEASE")
}
```

Ivy 依赖管理

如果你更喜欢用Ivy管理依赖也有类似的配置选项。

配置 Ivy， 将指向 Spring 的库 添加下面的 resolver（解析器）到你ivysettings.xml：

```
<resolvers>
    <ibiblio name="io.spring.repo.maven.release"
            m2compatible="true"
            root="http://repo.spring.io/release/"/>
</resolvers>
```

可以适当修改 root URL 从 /release 到 /milestone 或 /snapshot 。

一旦配置，就能添加依赖， 举例 (在 ivy.xml):

```
<dependency org="org.springframework"
    name="spring-core" rev="4.2.1.RELEASE" conf="compile->runtime"/>
```

分发的 zip 文件

虽然使用构建系统，支持依赖管理是推荐的方式获得了 Spring Framework,，它仍然是可下载分发的 zip 文件。

分发的 zip 文件是发布到 Spring Maven Repository （这是为了我们的便利，在下载这些文件的时候你不需要 Maven 或者其他的构建系统）。

下载一个 Zip，在web 浏览器打开 http://repo.spring.io/release/org/springframework/spring ，选择适当的文件夹的版本。下载完毕文件结尾是 -dist.zip，例如， spring-framework-{spring-version}-RELEASE-dist.zip 。分发也支持发布里程碑和快照。

日志

对于 Spring 日志是非常重要的依赖，因为：a）它是唯一的外部强制性的依赖；b）每个人都喜欢从他们使用的工具看到一些输出；c）Spring 结合很多其他工具都选择了日志依赖。应用开发者的一个目标就是往往是有统一的日志配置在一个中心位置为了整个应用程序，包括所有的外部元件。这就更加困难，因为它可能已经有太多选择的日志框架。

在 Spring 强制性的日志依赖 是 Jakarta Commons Logging API（JCL）。我们编译 JCL，我们也使得 JCL `Log` 对象对 Spring Framework 的扩展类可见。所有版本的 Spring 使用同样的日志库，这对于用户来说是很重要的：迁移就会变得容易向后兼容性，即使扩展 Spring的应用程序。我们这样做是为了是 Spring 的模块之一明确依赖 commons-logging (JCL的典型实现)，然后是的其他的所有模块在编译的时候都依赖它。如果你使用 Maven 为例，在你想拿起依赖 commons-logging ，这个是来自 Spring 的并且明确来自中心模块 spring-core。

关于 commons-logging 的好处是你不需要任何东西就能让你的应用程序程序跑起来。它运行时有一个发现算法，该算法在类路径的所有地方寻找其他的日志框架并且使用它认为适合的（或者你可以告诉它你需要的是哪一个）。如果没有其他的日志框架存在，你可以从JDK（Java.util.logging 或者JUL 的简称）获得日志。在大多数情况下，你可以在控制台查看你的Spring 应用程序工作和日志，并且这是很重要的。

不使用 **Commons Logging**

不幸的，在 commons-logging 里运行时发现算法，方便最终用户,是有问题的。如果我们能够时光倒流，现在从新开始 Spring 项目并且他使用了不同的日志依赖。第一个选择很可能是Simple Logging Facade for Java (SLF4J)，过去也曾被许多其他工具通过 Spring 使用在他们的应用程序。

基本上有两种方法可以关闭commons-logging：

1. 通过 spring-core 模块排除依赖（因为它是唯一的显示依赖于 commons-logging 的模块）。
2. 依赖特殊的 commons-logging 依赖，用空的jar（更多的细节可以在SLF4J FAQ中找到）替换掉库。

排除 commons-logging，添加以下的内容到 dependencyManagement 部分：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.2.1.RELEASE</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

现在，这个应用程序可以打破，因为在类路径上没有实现 JCL API，因此要修复它就必须提供有一个新的。在下一节我们将向你展示如何提供另一种实现 JCL，使用 SLF4J 作为例子的另一种实现。

使用 **SLF4J**

SLF4J 是一个更加简洁的依赖，在运行时相对于 commons-logging 更加的有效因为它使用编译时绑定来代替运行时发现其他日志框架的集成。这也意味着，你不得不更加明确你想在运行时发生什么，并相应的声明它或者配置它。SLF4J 提供绑定很多的常见日志框架，因此你可以选择一个你已经使用的，并且绑定到配置和管理。

SLF4J 提供了绑定很多的常见日志框架，包括 JCL，它也做了反向工作:是其他日志框架和它自己之间的桥梁。因此在 Spring 中使用 SLF4J 时，你需要使用 SLF4J-JCL 桥接替换掉 commons-logging 的依赖。一旦你这么做了，Spring 调用日志就会调用 SLF4J API，因此如果在你的应用程序中的其他库使用这个API，那么你就需要有个地方配置和管理日志。

一个常见的选择就是桥接 Spring 和 SLF4J，提供显示的绑定 SLF4J 到Log4J 上。你需要支持 4 个的依赖（排除现有的 commons-logging）：桥接，SLF4J API，绑定 Log4J 和 Log4J 实现自身。在 Maven 中你可以这样做：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.2.1.RELEASE</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
    </dependency>
</dependencies>
```

这似乎是一个很大的依赖性，其仅仅是为了得到一些日志文件。那就这样吧，但是它是可选的，它在关于类加载器的问题上应该比 commons-logging 表现的更加的好，值得注意的是，如果你在一个严格的容器中像OSGi 平台。据说也有一个性能优势应为绑定是在编译时而不是在运行时。

SLF4J 是用户中一个常见的选择，使用它可以更少的步骤和产生更少的依赖，直接绑定 Logback。这消除了多余的绑定步骤，因为 Logback 直接实现了 SLF4J，因此你只需要依赖两个库而不是4个（jcl-over-slf4j 和 logback）。如果你这样做，你可能还需要从其他外部依赖（不是 Spring）排除 slf4j-api 依赖，因为在类路径中你只需要一个版本的API。

使用 **Log4J**

许多人使用 Log4j 作为日志框架，用于配置和管理的目的。它是有效的和完善的，事实上这也是我们在运行时使用的，当我们构架和测试 Spring 时。Spring 也提供一些配置和初始化 Log4j 的工具，因此在某些模块上它有一个可选的编译时依赖在 Log4j。

为了使 Log4j 工作在默认的 JCL 依赖下（commons-logging），所有你需要做的事就是把 Log4j 放到类路径下，为它提供配置文件(log4j.properties 或者 log4j.xml 在类路径的根目录下)。因此对于Maven 用户这就是你的依赖声明：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.2.1.RELEASE</version>
    </dependency>
```

```
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
    </dependency>
</dependencies>
```

下面是一个 log4j.properties 的实例，用于将日志打印到控制台：

```
log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %t %c{2}:%L - %m%n

log4j.category.org.springframework.beans.factory=DEBUG
```

Runtime Containers with Native JCL

很多的人在提供 JCL 实现的容器下运行他们的 Spring 应用程序。IBM Websphere Application Server (WAS)为例。这样往往会导致问题，遗憾的是没有一个一劳永逸的解决方案；简单的包含 commons-logging 在大多数情况下是不够的。

要清楚这一点：问题报告通常不是 JCL 本身，或者 commons-logging：相反，他们是绑定 commons-logging 到其他的框架（通常是 Log4j ）。这可能会失败，因为commons-logging改变了路径，当他们运行时发现在一些容器中找到了老版本（1.0）并且现在大多数人使用的现代版本(1.1)。Spring 不使用 JCL API 任何不常见的模块，所以没有什么问题出现，但是很快 Spring 或者你的应用程序视图做一些日志时，你会发现绑定的 Log4j 不工作了。

在这种情况下，WAS 最容易的是转化类加载器的层次结构（IBM 称之为“parent last”），使应用程序控制 JCL 的依赖关系，而不是容器。该选项不是总是开放的，但是有很多其他的建议在公共领域的替代方法，你的里程可能取决于确切的版本和特定的容器。

在 Spring 4.0 新功能和增强

Spring 框架第一个版本发布于 2004 年，自发布以来已历经三个主要版本更新: Spring 2.0 提供了 XML 命名空间和 AspectJ 支持；Spring 2.5 增加了注释驱动（annotation-driven）的配置支持； Spring 3.0增加了对 Java 5+ 版本的支持和 `@Configuration` 模型。

Spring 4.0 是最新的主要版本，并且首次完全支持 Java 8 的特性。你仍然可以使用老版本的 Java，但是最低版本的要求已经提高到 Java SE 6。我们也借主要版本更新的机会删除了许多过时的类和方法。

你可以在Spring Framework GitHub Wiki上查看 升级 Spring 4.0 的迁移指南。

改进的入门体验

新的 spring.io 网站提供了一整个系列的 "入门指南" 帮助你学习 Spring。你可以本文档的 1. Getting Started with Spring 一节阅读更多的入门指南。新网站还提供了Spring 之下其他额外项目的一个全面的概述。

如果你是一个 Maven 用户，你可能会对 BOM 这个有用的 POM 文件感兴趣， 这个文件已经与每个 Spring 的发布版发布。

移除不推荐的包和方法

所有过时的包和许多过时的类和方法已经从Spring4中移除。如果你从之前的发布版升级Spring，你需要保证已经修复了所有使用过时的API方法。

查看完整的变化： API差异报告。

请注意，所有可选的第三方依赖都已经升级到了最低2010/2011(例如Spring4 通常只支持 2010 年的最新或者现在的最新发布版本):尤其是 Hibernate 3.6+、EhCache 2.1+、Quartz 1.8+、Groovy 1.8+、Joda-Time 2.0+。但是有一个例外，Spring4依赖最近的Hibernate Validator 4.3+，现在对Jackson的支持集中在2.0+版本 (Spring3.2支持的Jackson 1.8/1.9，现在已经过时）。

Java 8(以及6和7)

Spring4 支持 Java8 的一些特性。你可以在 Spring 的回调接口中使用 lambda 表达式 和 方法引用。支持 `java.time` (JSR-310)的值类型和一些改进过的注解，例如 `@Repeatable`。你还可以使用 Java8 的参数名称发现机制（基于 `-parameters` 编译器标志）。

Spring 仍然兼容老版本的 Java 和 JDK：Java SE 6（具体来说，支持JDK6 update 18 ）以上版本，我们建议新的基于 Spring4 的项目使用Java7或Java8。

Java EE 6 和 7

Java EE 6 或以上版本是 Spring4 的底线,与 JPA2.0 和 Servlet3.0规范有着特殊的意义。为了保持与 Google App Engine 和旧的应用程序服务器兼容,Spring4 可以部署在 Servlet2.5 运行环境。但是我们强烈的建议您在 Spring 测试和模拟测试的开发环境中使用 Servlet3.0+。

如果你是*WebSphere 7*的用户，一定要安装*JPA2.0*功能包。在*WebLogic 10.3.4*或更高版本，安装附带的*JPA2.0*补丁。这样就可以将这两种服务器变成*Spring4*兼容的部署环境。

从长远的观点来看，Spring4.0 现在支持 Java EE 7 级别的适用性规范：尤其是 JMS 2.0, JTA 1.2, JPA 2.1, Bean Validation 1.1 和JSR-236 并发工具类。像往常一样，支持的重点是独立的使用这些规范。例如在 Tomcat 或者独立环境中。但是，当把 Spring 应用部署到 Java EE 7 服务器时它同样适用。

注意，Hibernate 4.3 是 JPA 2.1 的提供者，因此它只支持 Spring4。同样适用用于作为 Bean Validation 1.1 提供者的 Hibernate Validator 5.0。这两个都不支持 Spring3.2。

Groovy Bean Definition DSL

Spring4.0 支持使用 Groovy DSL 来进行外部的 bean 定义配置。这在概念上类似于使用 XML 的 bean 定义，但是支持更简洁的语法。使用Groovy 还允许您轻松地将 bean 定义直接嵌入到引导代码中。例如：

```
def reader = new GroovyBeanDefinitionReader(myApplicationContext)
reader.beans {
    dataSource(BasicDataSource) {
        driverClassName = "org.hsqldb.jdbcDriver"
        url = "jdbc:hsqldb:mem:grailsDB"
        username = "sa"
        password = ""
        settings = [mynew:"setting"]
    }
    sessionFactory(SessionFactory) {
        dataSource = dataSource
    }
    myService(MyService) {
        nestedBean = { AnotherBean bean ->
            dataSource = dataSource
        }
    }
}
```

有关更多信息，请参阅 `GroovyBeanDefinitionReader` javadocs

核心容器改进

有几种对核心容器的常规改进：

- Spring 现在注入 Bean 的时候把 泛型类型当成一种形式的限定符。例如：如果你使用S pring Data `Repository` 你可以方便的插入特定的实现：`@Autowired Repository<Customer> customerRepository`。
- 如果你使用 Spring 的元注解支持，你现在可以开发自定义注解来公开源注解的特定属性。
- 当自动装配到lists和arrays时，Beans 现* 在可以被 排序 了。支持 `@Order` 注解和 `Ordered` 接口两种方式。 `@Lazy` 注解现在可以用在注入点以及 `@Bean` 定义上。
- 引入 `@Description` 注解,开发人员可以使用基于Java 方式的配置。
- 根据条件筛选 Beans的广义模型通过 `@Conditional` 注解加入。这和 `@Profile` 支持的类似，但是允许以编程式开发用户定义的策略。
- 基于CGLIB的代理类不在需要默认的构造方法。这个支持是由 objenesis库提供。这个库重新打包到 Spring 框架中，作为Spring框架的一部分发布。通过这个策略，针对代理实例被调用没有构造可言了。
- 框架现在支持管理时区。例如 `LocaleContext`。

常规Web改进

现在仍然可以部署到 Servlet 2.5 服务器，但是 Spring4.0 现在主要集中在 Servlet 3.0+ 环境。如果你使用 Spring MVC 测试框架，你需要将 Servlet 3.0 兼容的 JAR 包放到 测试的 *classpath* 下。

除了稍后会提到的 WebSocket 支持外，下面的常规改进已经加入到Spring 的 Web 模块：

- 你可以在Spring MVC应用中使用新的 `@RestController` 注解，不在需要给 `@RequestMapping` 的方法添加 `@ResponseBody` 注解。
- AsyncRestTemplate 类已被添加进来，当开发 REST 客户端时，允许非阻塞异步支持。
- 当开发 Spring MVC 应用时，Spring现在提供了全面的时区支持 。

WebSocket, SockJS, 和 STOMP 消息

一个新的 `spring-websocket` 模块提供了全面的基于 WebSocket 和在Web 应用的客户端和服务器之间双向通信的支持。它和 Java WebSocket API JSR-356 兼容，此外还提供了当浏览器不支持 WebSocket 协议时 (如 Internet Explorer < 10)的基于 SockJS 的备用选项(如 WebSocket emulation)。

一个新的 `spring-messaging` 模块添加了支持 STOMP 作为 WebSocket 子协议用于在应用中使用注解编程模型路由和处理从 WebSocket 客户端发送的 STOMP 消息。由于 `@Controller` 现在可以同时包含 `@RequestMapping` 和 `@MessageMapping` 方法用于处理 HTTP 请求和来自 WebSocket 连接客户端发送的消息。新的 `spring-messaging` 模块还包含了来自以前 Spring 集成项目的关键抽象，例如 `Message`、 `MessageChannel`、 `MessageHandler` 和其他作为基于消息传递的应用程序的基础。

欲知详情以及较全面的介绍，请参见Chapter 20, WebSocket 支持一节。

测试改进

除了精简 `spring-test` 模块中过时的代码外，Spring 4 还引入了几个用于单元测试和集成测试的新功能。

- 几乎 `spring-test` 模块中所有的注解（例如：`@ContextConfiguration`、`@WebAppConfiguration`、`@ContextHierarchy`、`@ActiveProfiles` 等等)现在可以用作元注解来创建自定义的 *composed annotations* 并且可以减少测试套件的配置。
- 现在可以以编程方式解决Bean定义配置文件的激活。只需要实现一个自定义的ActiveProfilesResolver，并且通过@ActiveProfiles的resolver属性注册。
- 新的 `SocketUtils` 类被引入到了 `spring-core` 模块。这个类可以使你能够扫描本地主机的空闲的 TCP 和 UDP 服务端口。这个功能不是专门用在测试的，但是可以证明在你使用 Socket 写集成测试的时候非常有用。例如测试内存中启动的SMTP服务器，FTP服务器，Servlet容器等。
- 从 Spring 4.0 开始, `org.springframework.mock.web` 包中的一套mock是基于Servlet 3.0 API。此外，一些Servlet API mocks（例如：`MockHttpServletRequest`、`MockServletContext` 等等）已经有一些小的改进更新，提高了可配置性。

JMS 改进

Spring 4.1 引入了一个更简单的基础架构，使用 `@JmsListener` 注解bean 方法来注册 JMS 监听端点。XML 命名空间已经通过增强来支持这种新的方式（ `jms:annotation-driven` ），它也可以完全通过Java配置(`@EnableJms` , `JmsListenerContainerFactory`)来配置架构。也可以使用 `JmsListenerConfigurer` 注解来注册监听端点。

Spring 4.1 还调整了 JMS 的支持，使得你可以从 `spring-messaging` 在 Spring4.0 引入的抽象获益，即：

- 消息监听端点可以有更为灵活的签名，并且可以从标准的消息注解获益，例如 `@Payload` 、 `@Header` 、 `@Headers` 和 @ `SendTo` 注解。另外，也可以使用一个标准的消息，以代替 `javax.jms.Message` 作为方法参数。
- 一个新的可用 `JmsMessageOperations` 接口和允许操作使用 `Message` 抽象的 `JmsTemplate` 。

最后，Spring 4.1提供了其他各种各样的改进：

- JmsTemplate中的同步请求-答复操作支持
- 监听器的优先权可以指定每个 `<jms:listener/>` 元素
- 消息侦听器容器恢复选项可以通过使用 `BackOff` 实现进行配置
- JMS 2.0消费者支持共享

缓存改进

Spring 4.1 支持JCache (JSR-107)注解使用Spring的现有缓存配置和基础结构的抽象；使用标准注解不需要任何更改。

Spring 4.1也大大提高了自己的缓存抽象：

- 缓存可以在运行时使用 `CacheResolver` 解决。因此使用 `value` 参数定义的缓存名称不在是强制性的。
- 更多的操作级自定义项：缓存解析器，缓存管理器，键值生成器
- 一个新的 `@CacheConfig` 类级别注解允许在类级别上共享常用配置，不需要启用任何缓存操作。
- 使用 `CacheErrorHandler` 更好的处理缓存方法的异常

Spring 4.1为了在 `CacheInterface` 添加一个新的 `putIfAbsent` 方法也做了重大的更改。

Web 改进

- 现有的基于 `ResourceHttpRequestHandler` 的资源处理已经扩展了新的抽象 `ResourceResolver`， `ResourceTransformer` 和 `ResourceUrlProvider` 。一些内置的实现提供了版本控制资源的 URL（有效的 HTTP 缓存），定位 gzip 压缩的资源，产生 HTML5 AppCache清单，以及更多的支持。参见第17.16.7，“Serving of Resources(服务资源)”。
- JDK 1.8 的 `java.util.Optional` 现在支持 `@RequestParam`， `@RequestHeader` 和 `@MatrixVariable` 控制器方法的参数。
- `ListenableFuture` 支持作为返回值替代 `DeferredResult` 所在的底层服务（或者调用 `AsyncRestTemplate` ）已经返回 `ListenableFuture` 。
- `@ModelAttribute` 方法现在依照相互依存关系的顺序调用。见SPR-6299。
- Jackson的 `@JsonView` 被直接支撑在 `@ResponseBody` 和 `ResponseEntity` 控制器方法用于序列化不同的细节对于相同的POJO（如摘要与细节页）。同时通过添加序列化视图类型作为模型属性的特殊键来支持基于视图的渲染。见Jackson Serialization View Support(Jackson序列化视图支持)
- Jackson 现在支持 JSONP， 见 Jackson JSONP Support
- 一个新的生命周期选项可用于在控制方法返回后，响应被写入之前拦截 `@ResponseBody` 和 `ResponseEntity` 方法。要充分利用声明 `@ControllerAdvice` bean 实现 `ResponseBodyAdvice` 。为 `@JsonView` 和 JSONP 的内置支持利用这一优势。参见第17.4.1，“使用HandlerInterceptor 拦截请求”。
- 有三个新的 `HttpMessageConverter` 选项：
  - GSON - 比 Jackson 更轻量级的封装;已经被使用在 Spring Android
  - Google Protocol Buffers - 高效和有效的企业内部跨业务的数据通信协议，但也可以用于浏览器的 JSON 和 XML 的扩展
  - Jackson 基于 XML 序列化，现在通过 jackson-dataformat-xml 扩展得到了支持。如果 `jackson-dataformat-xml` 在 classpath， 默认情况下使用 `@EnableWebMvc` 或 `<mvc:annotation-driven/>` ，这是，而不是JAXB2。
- 如 JSP 等视图现在可以通过名称参照控制器映射建立链接控制器。默认名称分配给每一个 `@RequestMapping` 。例如 `FooController` 的方法与 `handleFoo` 被命名为“FC＃handleFoo”。命名策略是可插拔的。另外，也可以通过其名称属性明确命名的 `@RequestMapping` 。在Spring JSP标签库的新 `mvcUrl` 功能使这个简单的JSP页面中使用。参见第17.7.2，“Building URIs to Controllers and methods from views”
- `ResponseEntity` 提供了一种 builder 风格的 API 来指导控制器向服务器端的响应的展示，例如，ResponseEntity.ok()。
- `RequestEntity` 是一种新型的，提供了一个 builder 风格的 API 来引导客户端的 REST 响应 HTTP 请求的展示。
- MVC 的 Java 配置和 XML 命名空间：
  - 视图解析器现在可以配置包括内容协商的支持，请参见17.16.6“视图解析”。
  - 视图控制器现在已经内置支持重定向和设置响应状态。应用程序可以使用它来配置重定向的URL，404 视图的响应，发送“no content”的响应，等等。有些用例这里列出。
  - 现在内置路径匹配的自定义。参见第17.16.9，“路径匹配”。
- Groovy 的标记模板支持（基于Groovy的2.3）。见 `GroovyMarkupConfigurer` 和各自的 `ViewResolver` 和“视图”的实现。

WebSocket STOMP 消息 改进

- SockJS(Java)客户端支持.查看 `SockJsClient` 和在相同包下的其他类.
- 发布新的应用上下文实践 `SessionSubscribeEvent` 和 `SessionUnsubscribeEvent`, 用于STOMP客户端的订阅和取消订阅.
- 新的"websocket"级别.查看 Section 25.4.14, “WebSocket Scope”.
- `@SendToUser` 仅仅靠一个会话就可以命中, 而不一定需要一个授权的用户.
- `@MessageMapping` 方法使用 **.** 来代替 **/** 作为目录分隔符。查看 SPR-11660.。
- STOMP/WebSocket 监听消息的收集和记录。查看25.4.16, “Runtime Monitoring”.。
- 显著优化和改善在调试模式下依然保持日志的可读性和简洁性。
- 优化消息创建，包含对临时消息可变性的支持和避免自动消息ID和时间戳的创建。查看 `MessageHeaderAccessor` 的java文档。
- 在WebSocket会话建立之后的1分钟没有任何活动,则关闭STOMP/WebSocket连接。

测试 改进

- Groovy脚本现在能使用 `ApplicationContext` 配置，在TestContext框架中整合测试。the section called “Context configuration with Groovy scripts”
- 现在通过新的 `TestTransaction` 接口，使用编程化来开始结束测试管理事务。the section called “Programmatic transaction management”
- 现在SQL脚本的执行可以通过 `Sql` 和 `SqlConfig` 注解申明在每一个类和方法中。the section called “Executing SQL scripts”
- 测试属性值可以通过配置 `@TestPropertySource` 注解来自动覆盖系统和应用的属性值。the section called “Context configuration with test property sources”。
- 默认的 `TestExecutionListeners` 现在可以自动被发现。the section called “Automatic discovery of default TestExecutionListeners” 。
- 自定义的 `TestExecutionListeners` 现在可以通过默认的监听器自动合并。the section called “Merging TestExecutionListeners”。
- 在TestContext框架中的测试事务方面的文档已经通过更多解释和其他事例得到改善。the section called “Transaction management”。
- `MockServletContext` 、 `MockHttpServletRequest` 和其他servlet接口mocks等诸多改善。
- `AssertThrows` 重构后， `Throwable` 代替 `Exception` 。
- Spring MVC测试中，使用JSONPath选项返回JSON格式可以使用JSON Assert来断言,非常像之前的XML和XMLUnit。
- `MockMvcBuilder` 现在可以在 `MockMvcConfigurer` 的帮助下创建。 `MockMvcConfigurer` 的加入使得Spring Security的设置更加简单，同时使用于任何第三方的普通封装设置或则仅仅在本项目中。
- `MockRestServiceServer` 现在支持 `AsyncRestTemplate` 作为客户端测试。

- 发布新的应用上下文实践 `SessionSubscribeEvent` 和 `SessionUnsubscribeEvent` ,用于STOMP客户端的订阅和取消订阅.
- 新的"websocket"级别.查看[Section 21.4.13, “WebSocket Scope”.](http://docs.spring.io/spring/docs/current/spring-framework-

核心容器改进

- 如 @bean 注释,就如同得到发现和处理 Java 8 默认方法一样,可以允许组合配置类与默认@bean 接口方法。
- 配置类现在可以声明 @import 作为常规组件类,允许引入的配置类和组件类进行混合。
- 配置类可以声明一个 @Order 值,用来得到相应的处理顺序(例如重写 bean 的名字)，即使通过类路径扫描检测。
- @Resource 注入点支持 @Lazy 声明,类似于 @autowired, 用于接收用于请求目标 bean 的懒初始化代理。
- 现在的应用程序事件基础架构提供了一个基于注解的模型以及发布任意事件的能力。
  - 任何受管 bean 的公共方法使用 @EventListener 注解来消费事件。
  - @TransactionalEventListener 提供事务绑定事件支持。
- Spring Framework 4.2引入了一流的支持声明和查找注释属性的别名。新 @AliasFor 注解可用于声明一双别名属性在一个注释中或从一个属性在一个声明一个别名定义注解在元注释一个属性组成。
  - 下面的注解已加了 @AliasFor 为了支持提供更有意义的 value 属性的别名: @Cacheable, @CacheEvict, @CachePut, @ComponentScan, @ComponentScan.Filter, @ImportResource, @Scope, @ManagedResource, @Header, @Payload, @SendToUser, @ActiveProfiles, @ContextConfiguration, @Sql, @TestExecutionListeners, @TestPropertySource, @Transactional, @ControllerAdvice, @CookieValue, @CrossOrigin, @MatrixVariable, @RequestHeader, @RequestMapping, @RequestParam, @RequestPart, @ResponseStatus, @SessionAttributes, @ActionMapping, @RenderMapping, @EventListener, @TransactionalEventListener
  - 例如，spring-test 的 @ContextConfiguration 现在声明如下：

```
public @interface ContextConfiguration {

    @AliasFor("locations")
    String[] value() default {};

    @AliasFor("value")
    String[] locations() default {};

    // ...
}
```

```
* 同样，组合注解（composed annotations）从元注解覆盖的属性,现在可以使用 @AliasFor 进行细粒度控制哪些属性是覆盖在一个注释的层次结构。事实
* 例如，开发一个组合注解用于一个自定义的属性的覆盖
```

```
@ContextConfiguration
public @interface MyTestConfig {

    @AliasFor(annotation = ContextConfiguration.class, attribute = "value")
    String[] xmlFiles();

    // ...
}
```

```
* 见 [Spring Annotation Programming Model](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsin
```

- 许多改进Spring的搜索算法用于寻找元注解。例如,局部声明组合注解现在喜欢继承注解。
- 从元注解覆盖属性的组合注解，可以被发现在接口和 abstract, bridge, & interface 方法就像在类，标准方法，构造函数，和字段。
- Map 表示的注解属性(和 AnnotationAttributes 实例)可以 synthesized (合成，即转换)成一个注解。
- 基于字段的数据绑定的特点(DirectFieldAccessor)与当前的基于属性的数据绑定关联(BeanWrapper)。特别是,基于字段

的绑定现在支持集合,数组和 Map 的导航。

- DefaultConversionService 现在提供开箱即用的转化器给 Stream, Charset, Currency, 和 TimeZone. 这些转换器可以独立的添加到任何 ConversionService
- DefaultFormattingConversionService 提供开箱即用的支持 JSR-354 的 Money & Currency 类型 (前提是 'javax.money' API 出现在 classpath): 这些被命名为 MonetaryAmount 和 CurrencyUnit。支持使用 @NumberFormat
- @NumberFormat 现在作为元注解使用
- JavaMailSenderImpl 中新的 testConnection() 方法用于检查与服务器的连接
- ScheduledTaskRegistrar 用于暴露调度的任务
- Apache commons-pool2 现在支持用于 AOP CommonsPool2TargetSource 的池化
- 引入 StandardScriptFactory 作为脚本化 bean 的 JSR-223 的基本机制，通过 XML 中的 `lang:std` 元素暴露。支持如 JavaScript 和 JRuby。（注意：JRubyScriptFactory 和 `lang:jruby` 现在不推荐使用了，推荐用 JSR-223）

数据访问改进

- javax.transaction.Transactional 现在可以通过 AspectJ 支持
- SimpleJdbcCallOperations 现在支持命名绑定
- 完全支持 Hibernate ORM 5.0: 作为 JPA 供应商 (自动适配)和原生的 API 一样 (在新的 org.springframework.orm.hibernate5 包中涵盖了该内容)
- 嵌入式数据库可以自动关联唯一名字，并且 `<jdbc:embedded-database>` 支持新的 database-name 属性。见下面“测试改进”内容

JMS 改进

- autoStartup 属性可以通过 JmsListenerContainerFactory 进行控制
- 应答类型 Destination 可以配置在每个监听器容器
- @SendTo 的值可以用 SpEL 表达式
- 响应目的地可以通过 JmsResponse 在运行时计算
- @JmsListener 是可以可以重复的注解用于声明多个 JMS 容器在相同的方法上 (若你还没有用上 Java8 请使用新引入的 @JmsListeners)。

Web 改进

- 支持 HTTP Streaming 和 Server-Sent Events , se见 “HTTP Streaming”
- 内建支持 CORS， 包括全局 (MVC Java 配置和 XML 命名空间) 和本地 (如 @CrossOrigin) 配置。 见 26 章, CORS 支持
- HTTP 缓存升级
  - 新的 CacheControl 构建器; 插入 ResponseEntity, WebContentGenerator, ResourceHttpRequestHandler
  - 改进的 ETag/Last-Modified 在 WebRequest 中支持
- 自定义映射注解使用 @RequestMapping 作为 元数据注解
- AbstractHandlerMethodMapping 中的 public 方法用于运行时注册和注销请求映射
- AbstractDispatcherServletInitializer 中的 Protected createDispatcherServlet 方法用来进一步自定义 DispatcherServlet 实例
- HandlerMethod 作为 @ExceptionHandler方法的方法参数,特别是方便 @ControllerAdvice 组件
- java.util.concurrent.CompletableFuture 作为 @Controller 方法返回值类型
- 字节范围（Byte-range）的请求支持在 HttpHeaders，用于静态资源
- @ResponseStatus 发现嵌套异常。
- 在 RestTemplate 中的 UriTemplateHandler 扩展端点
  - DefaultUriTemplateHandler 暴露 baseUrl 属性和路径段的编码选项
  - 扩展端点可以使用插入任何 URI 模板库
- OkHTTP 与 RestTemplate 集成
- 自定义 baseUrl 在 MvcUriComponentsBuilder 选择方法。
- 序列化/反序列化异常消息现在记录为 WARN 级别
- 默认的 JSON 前缀改变了从 `{}&&` 改为更安全的 `)]}',`
- 新的 RequestBodyAdvice 扩展点和内置的实现支持 Jackson 的 在 @RequestBody 的 @JsonView
- 当使用 GSON 或 Jackson 2.6 +,处理程序方法的返回类型是用于提高参数化类型的序列化，比如 `List<Foo>`
- 引入的 ScriptTemplateView 作为 JSR-223 的脚本化 web 视图机制为基础,关注 JavaScript 视图模板 Nashorn (JDK 8)。

WebSocket 消息改进

- 暴露展示信息关于用户的连接和订阅:
  - 新 SimpUserRegistry 公开为一个名为“userRegistry”的bean。
  - 共享在服务器集群的展示信息(见代理中继配置选项)
- 解决用户目的地在集群的服务器(见代理中继配置选项)。
- StompSubProtocolErrorHandler 扩展端点用来自定义和控制 STOMP ERROR 帧给用户
- 全局 @MessageExceptionHandler 方法通过 @ControllerAdvice 组件
- 心跳和 SpEL 表达式'selector'头用 SimpleBrokerMessageHandler 订阅
- STOMP 客户端使用TCP 和 WebSocket; 见 25.4.13, “STOMP 客户端”
- @SendTo 和 @SendToUser 可以包含目标变量的占位符。 Jackson 的 @JsonView 支持 @MessageMapping 和 @SubscribeMapping 方法返回值
- ListenableFuture 和 CompletableFuture 是从 @MessageMapping 和 @SubscribeMapping 方法返回类型值
- MarshallingMessageConverter 用于 XML 有效载荷

测试改进

- 基于 JUnit 集成测试现在可以执行 JUnit 规则而不是SpringJUnit4ClassRunner。这允许基于 spring 的集成测试与运行 JUnit 的 Parameterized 或第三方 运行器 MockitoJUnitRunner 等。详见 Spring JUnit 规则
- Spring MVC Test 框架，现在支持第一类 HtmlUnit，包括集成 Selenium's WebDriver,允许基于页面的 Web 应用测试而无需部署到 Servlet 容器。详见 14.6.2, "HtmlUnit 集成"
- AopTestUtils 是一个新的测试工具，允许开发者获得潜在的目标对象的引用隐藏在一个或多个 Spring 代理。详见 13.2.1, "常见测试工具"
- ReflectionTestUtils 现在支持 setting 和 getting static 字段,包括常量
- bean 定义归档文件的原始顺序，通过 @ActiveProfiles 声明，现在保留为了支持用例,如 Spring 的 ConfigFileApplicationListener 引导加载配置文件基于活动归档文件的名称。
- @DirtiesContext 支持新 BEFORE_METHOD BEFORE_CLASS,BEFORE_EACH_TEST_METHOD 模式，用于测试之前关闭ApplicationContext——例如,如果一些烦人的(即，有待确定)测试在一个大型测试套件的 ApplicationContext 的原始配置已经损坏。
- @Commit 是新的注解直接可以用来代替 @Rollback(false)
- @Rollback 用来配置类级别的默认回滚语义
  - 因此,现在 @TransactionConfiguration 弃用,在后续版本将被删除。
- @Sql 现在支持内联 SQL 语句的执行通过一个新的 statements 属性
- ContextCache 用于缓存测试之间的 ApplicationContext，而现在这是一个公开的 API，默认的实现可以替代自定义的缓存需求
- DefaultTestContext, DefaultBootstrapContext, 和 DefaultCacheAwareContextLoaderDelegate 现在是公开的类，支持子包，允许自定义扩展
- TestContextBootstrapper 现在负责构建 TestContext
- 在 Spring MVC Test 框架，MvcResult 详情可以被日志记录在 DEBUG 级别或者写入自定义的 OutputStream 或 Writer。详见 log(), print(OutputStream), 和 MockMvcResultHandlers 的 print(Writer) 方法
- JDBC XML名称空间支持一个新的 `<jdbc:embedded-database>` 的 database-name 属性,允许开发人员为嵌入式数据库设置独特的名字——例如,通过一个 SpEL 表达式或 前活动bean定义配置文件所影响的占位符属性
- 嵌入式数据库现在可以自动分配一个唯一的名称,允许常用的测试数据库配置在不同的 ApplicationContext 的测试套件中。 参见18.8.6"给嵌入式数据库生成惟一名称"的细节。

Part III. 核心技术

该部分的参考文档涵盖了Spring Framework中所有绝对不可或缺的技术。

这其中最重要的部分就是Spring Framework中的控制反转（IoC)容器。Spring Framework中IoC容器是紧随着Spring中面向切面编程（AOP)技术的全面应用的来完整实现的。Spring Framework有它自己的一套AOP框架，这套框架从概念上很容易理解，而且成功解决了Java企业级应用中AOP需求80%的核心要素。

同样Spring与AspectJ的集成（目前从功能上来说是最丰富，而且也无疑是Java企业领域最成熟的AOP实现）也涵盖在内。

- Chapter 6, IoC容器
- Chapter 7, 资源
- Chapter 8, 验证，数据绑定和类型转换
- Chapter 9, Spring表达式语言（SpEL)
- Chapter 10, Spring中的面向切面编程
- Chapter 11, Spring AOP API

介绍 Spring IoC 容器和 bean

本章涵盖了 Spring Framework实现控制翻转 (IoC) 的原则。 IoC 有时也被称为依赖注入 (DI)。这是一个对象定义他们依赖的过程，其中对象之间的相关性，也就是说，它们一起工作，只能通过构造函数参数，参数工厂方法或设置在其构造后的对象实例或者是从一个工厂方法返回的对象实例的属性上。容器在创建的 bean 注入这些依赖。这个过程是根本的反转，因此称为控制反转（IoC），bean 本身通过直接构造类，或作为 Service Locator(服务定位器)模式的机制，来控制其依赖的实例或依赖的位置。

org.springframework.beans 和 org.springframework.context 是Sprig 框架的 IoC 容器的基础包的。 BeanFactory 提供能够管理任何类型的对象的高级配置机制。 ApplicationContext 是 BeanFactory 的一个子接口。它增加了与 Spring AOP 功能的整合更容易;消息资源处理（用于国际化），事件发布;和应用层的上下文，如WebApplicationContext 中的 Web 应用程序使用。

简而言之，BeanFactory 提供了配置框架和基本功能，而ApplicationContext 则增加了更多支持企业特定的功能。一个 ApplicationContext 是 BeanFactory 的一个完整的超集，本章将专对 门 Spring 的 IoC 容器进行描述。有关使用 BeanFactory 来代替ApplicationContext 的更多信息，请参见第6.16节，“BeanFactory”。

在 Spring 中，形成了应用程序的骨干，该对象由 Spring IoC 容器管理被称为 bean。一个 bean 是由 Spring IoC 容器实例化，组装，并以其他方式管理的对象。否则，一个 bean 只是简单的应用程序中的许多对象之一。bean，以及它们之间的相关性，反映在所使用的容器的配置元数据中（configuration metadata ）。

容器总览

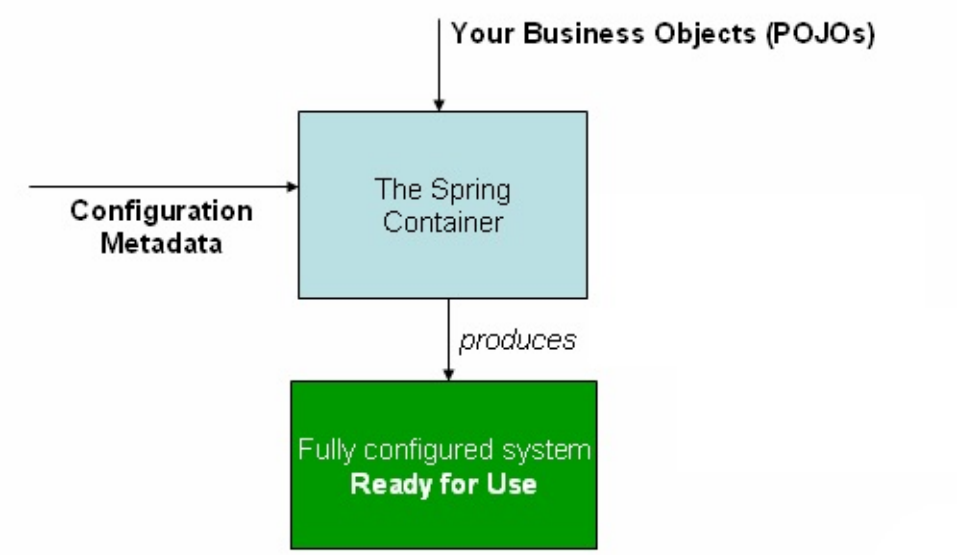
org.springframework.context.ApplicationContext 代表 Spring IoC 容器，并负责实例化，配置和组装上述 bean 的接口。容器是通过对象实例化，配置，和读取配置元数据汇编得到对象的构建。配置元数据可以是用 XML，Java 注解，或 Java 代码来展示。它可以让你描述组成应用程序的对象和对象间丰富的相互依赖。

Spring ApplicationContext 接口提供了几种即装即用的实现方式。在独立应用中，通常以创建 ClassPathXmlApplicationContext 或FileSystemXmlApplicationContext 的实例。虽然 XML 一直是传统的格式来定义配置元数据，但也可以指示容器使用 Java 注解或代码作为元数据格式，并通过提供少量的XML配置以声明方式启用这些额外的元数据格式的支持。

在大多数应用场合，不需要明确的用户代码来实例化一个 Spring IoC 容器的一个或多个实例。例如，在 web 应用程序中，在应用程序的 web.xml文件中一个简单的样板网站的 XML 描述符通常就足够了（见第6.15.4，“便捷的 ApplicationContext 实例化 Web 应用程序”）。如果您使用的是基于Eclipse的 Spring Tool Suite开发环境，该样板配置可以很容易地用点击几下鼠标或键盘创建。

下面的图展示了 Spring 是如何工作的高级视图。您的应用程序的类是通过配置元数据来结合的，以便 ApplicationContext 需要创建和初始化后，你有一个完全配置和可执行的系统或应用程序。

Figure 6.1. The Spring IoC container



配置元数据

如上述图所示，Spring IoC 容器使用 配置元数据（ configuration metadata）;这个配置元数据代表了应用程序开发人员告诉 Spring 容器在应用程序中如何来实例化，配置和组装对象。

传统的配置元数据是一个简单而直观的 XML 格式，这是大多数本章用来传达关键概念和 Spring IoC 容器的功能。

基于 *XML* 的元数据并不是配置元数据的唯一允许的形式。 *Spring IoC*容器本身是完全从此配置元数据实际写入格式脱钩。现在，许多开发商选择适合自己的 *Spring* 应用程序的基于 *Java* 的配置。

更多其他格式的元数据见：

- 基于注解的配置：Spring 2.5 的推出了基于注解的配置元数据支持。
- 基于Java的配置：Spring3.0 开始，由 Spring JavaConfig 项目提供了很多功能成为核心 Spring 框架的一部分。因此，你可以通过使用Java，而不是 XML 文件中定义外部 bean 到你的应用程序类。要使用这些新功能，请参阅

@Configuration，@Bean，@Import 和 @DependsOn 注解。

Spring 配置至少一个，通常不止一个 bean 来由容器来管理。基于 XML 的配置元数据显示这些 bean 配置的 `<bean/>` 包含于顶层元素 `<beans/>` 元素。 Java 配置通常使用 @Configuration 类中 @Bean 注解的方法。

这些 bean 定义对应于构成应用程序的实际对象。通常，您定义服务层对象，数据访问对象（DAO），展示对象，如 Struts Action 的情况下，基础设施的对象，如 Hibernate 的 SessionFactories，JMS Queues，等等。通常一个不配置细粒度域对象在容器中，因为它通常负责 DAO 和业务逻辑来创建和负载域对象。但是，你可以使用 Spring 和 AspectJ的集成配置在 IoC 容器的控制之外创建的对象。请参阅使用 AspectJ 在 Spring 中进行依赖关系注入域对象。

以下示例显示基于 XML 的配置元数据的基本结构：

```
<?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="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions go here -->

</beans>
```

id 属性是一个字符串，唯一识别一个独立的 bean 定义。class 属性定义了 bean 的类型，并使用完整的类名。 id 属性的值是指协作对象。将XML 用于参照协作对象未在本例中示出;请参阅 依赖以获取更多信息。

实例化容器

实例化 Spring IoC 容器是直截了当的。提供给 ApplicationContext构造器的路径就是实际的资源字符串，使容器装入从各种外部资源的配置元数据，如本地文件系统， Java CLASSPATH，等等。

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});
```

当你了解 Spring 的 IoC 容器，你可能想知道更多关于 Spring 的Resource 抽象，如第7章， 资源，它提供了一种方便的从一个 URI 语法定义的位置读取一个InputStream 描述。具体地，资源路被用作在第7.7节，“应用环境和资源的路径”中所述构建的应用程序的上下文。

下面的例子显示了服务层对象（services.xml中）配置文件：

```
<?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">

    <!-- services -->

    <bean id="petStore" class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>
```

```
    <!-- more bean definitions for services go here -->

</beans>
```

下面的例子显示了数据访问对象 daos.xml 文件：

```
<?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="accountDao"
        class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <bean id="itemDao" class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions for data access objects go here -->

</beans>
```

在上面的例子中，服务层由类 PetStoreServiceImpl，以及类型的两个数据访问对象 JpaAccountDao 和 JpaItemDao（基于 JPA对象/关系映射标准）组成。property name 元素是指 JavaBean 属性的名称，而 ref 元素引用另一个 bean 定义的名称。id 和 ref 元素之间的这种联系表达了合作对象之间的依赖关系。对于配置对象的依赖关系的详细信息，请参阅依赖。

撰写基于**XML**的配置元数据

它可以让 bean 定义跨越多个 XML 文件,这样做非常有用。通常，每个单独的 XML 配置文件代表你的架构一个逻辑层或模块。

您可以使用应用程序上下文构造从所有这些 XML 片段加载 bean 定义。这个构造函数的多个 Resource 位置，作为上一节中被证明。另外，使用 `<import/>` 元素的一个或多个出现，从另一个或多个文件加载 bean 定义。 例如：

```
<beans>
    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>

    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>
</beans>
```

services.xml，messageSource.xml 及 themeSource.xml 在前面的例子中，外部 bean 定义是由三个文件加载而来。所有位置路径是相对于导入文件的，因此 services.xml 是必须和导入文件在同一目录或类路径中的位置，而 messageSource.xml 和 themeSource.xml 来必须在导入文件的 resources 以下位置。正如你所看到的，前面的斜线被忽略，但考虑到这些路径是相对的，它更好的形式是不使用斜线。该文件的内容被导入，包括顶级 `<beans />` 元素，必须根据 Spring Schema 是有效的 XML bean 定义。

这是可能的，但不推荐，引用在使用相对*"../"*的路径的父目录中的文件。这样将创建一个文件，该文件是当前应用程序之外的依赖。特别是，该引用不推荐*"classpath：" URL*（例如，*"classpath:../services.xml"*），在运行时解决过程中选择了*"就近"*的类路径的根，然后查找到它的父目录。类路径配置的变化可能导致不同的，不正确的目录的选择。 您可以随时使用完全合格的资源位置，而不是相对路径：例如，*file:C:/config/services.xml"* 或 *"classpath:/config/services.xml"*。但是，要知道，你这是是在耦合应用程序的配置到特定的绝对位置。通常优选间接的方式应对这种绝对路径，例如，通过*"${…}"*在运行时解决了对*JVM*系统属性占位符。

使用容器

ApplicationContext 是能够保持 bean 定义以及相互依赖关系的高级工厂接口。使用方法 T getBean(String name, Class requiredType)就可以取得 bean 的实例。

ApplicationContext 中可以读取 bean 定义并访问它们，如下所示：

```
// create and configure beans
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});

// retrieve configured instance
PetStoreService service = context.getBean("petStore", PetStoreService.class);

// use configured instance
List<String> userList = service.getUsernameList();
```

您可以使用 getBean() 方法来获取 bean 的实例。 ApplicationContext 接口有一些其他的方法来获取 bean，但理想的应用程序代码不应该使用它们。事实上，你的应用程序代码不应该调用的getBean() 方法，因此在 Spring 的 API 没有依赖性的。例如，Spring如何与Web框架的集成提供了依赖注入各种Web框架类，如控制器和 JSF 管理的bean。

Bean 总览

Spring IoC 容易管理一个或者多个 bean。 bean 由应用到到容器的配置元数据创建,例如,在 XML 中定义 `<bean/>` 的形式。

容器内部,这些 bean 定义表示为 BeanDefinition 对象,其中包含(其他信息)以下元数据:

- 限定包类名称:典型的实际实现是定义 bean 的类。
- bean 行为配置元素,定义了容器中的Bean应该如何行为(范围、生命周期回调,等等)。
- bean 需要引用其他 bean 来完成工作,这些引用也称为合作者或依赖关系。
- 其他配置设置来设置新创建的对象,例如,连接使用 bean 的数量管理连接池,或者池的大小限制。

以下是每个 bean 定义的属性。

Table 6.1. The bean definition

属性	解释
class	Section 6.3.2, “Instantiating beans”
name	Section 6.3.1, “Naming beans”
scope	Section 6.5, “Bean scopes”
constructor arguments	Section 6.4.1, “Dependency Injection”
properties	Section 6.4.1, “Dependency Injection”
autowiring mode	Section 6.4.5, “Autowiring collaborators”
lazy-initialization mode	Section 6.4.4, “Lazy-initialized beans”
initialization method	the section called “Initialization callbacks”
destruction method	the section called “Destruction callbacks”

除了包含的信息里面 bean 定义的如何创建一个特定的bean, ApplicationContext 的实现还允许由用户注册现有创建在容器之外的现有对象。这是通过访问 ApplicationContext 的 BeanFactory 的 getBeanFactory() 方法 返回 BeanFactory 的实现 DefaultListableBeanFactory 。DefaultListableBeanFactory 支持这种通过 registerSingleton(..) 和registerBeanDefinition(..) 方法来注册。然而,典型的应用程序只能通过元数据定义的 bean 来定义。

需要尽早注册 Bean 元数据和手动使用单例的实例,这是为了使容器正确推断它们在自动装配和其他内省的步骤。虽然覆盖现有的元数据和现有的单例实例在某种程度上是支持的,新 bean 在运行时(同时动态访问工厂)注册不是官方支持,可能会导致并发访问 bean 容器中的异常和/或不一致的状态。

命名bean

每个 bean 都有一个或多个标识符。这些标识符在容器托管 bean 必须是唯一的。bean 通常只有一个标识符,但如果它需要不止一个,可以考虑额外的别名。

在基于 xml 的配置元数据,您可以使用 id 和 / 或名称属性指定 bean 标识符(。id 属性允许您指定一个 id。通常这些名字字母数字(“myBean”、“fooService”,等等),但可以包含特殊字符。如果你想介绍其他别名 bean,您还可以指定属性名称,由逗号分隔(,),分号(;),或白色空格。作为一个历史因素的要注意,在 Spring 3.1 版本之前,id 属性被定义为 xsd:ID类型,它限制可能的字符。3.1,它被定义为一个 xsd:string 类型。注意,bean id 独特性仍由容器执行,虽然不再由 XML 解析器。

你不需要提供一个 bean 的名称或id。如果没有显式地提供名称或id, 容器生成一个唯一的名称给 bean 。然而,如果你想引用 bean 的名字,通过使用 ref 元素或使用 Service Locator（服务定位器）风格查找,你必须提供一个名称。不使用名称的原因是， 内部 bean 和自动装配的合作者。

bean 名约定

约定是使用标准 *Java* 实例字段名称命名 *bean* 时的约定。也就是说,*bean* 名称开始以小写字母开头,后面采用“骆峰式”。例如*“accountManager”*、*“accountService’,‘userDao’,‘loginController’*,等等。

一致的*beans*命名可以让您的配置容易阅读和理解，如果你正在使用*Spring AOP*，当你通过 *bean* 名称应用到 *advice* 时，这会对你帮助很大。

bean 的别名

在对 bean 定义时，除了使用 id 属性指定一个唯一的名称外，为了提供多个名称，需要通过 name 属性加以指定，所有这个名称都指向同一个bean，在某些情况下提供别名非常有用，比如为了让应用每一个组件都能更容易的对公共组件进行引用。然而，在定义 bean 时就指定所有的别名并不总是很恰当。有时我们期望能够在当前位置为那些在别处定义的bean引入别名。在XML配置文件中，可以通过 `<alias/>` 元素来完成 bean 别名的定义，例如：

```
<alias name="fromName" alias="toName"/>
```

在这种情况下，如果容易中存在名为 fromName 的 bean 定义，在增加别名定义后，也可以用 toName 来引用。

例如，在子系统 A 中通过名字 subsystemA-dataSource 配置的数据源。在子系统B中可能通过名字 subsystemB-dataSource 来引用。当两个子系统构成主应用的时候，主应用可能通过名字 myApp-dataSource 引用数据源，将全部三个名字引用同一个对象，你可以将下面的别名定义添加到应用配置中：

```
<alias name="subsystemA-dataSource" alias="subsystemB-dataSource"/>
<alias name="subsystemA-dataSource" alias="myApp-dataSource" />
```

现在每个子系统和主应用都可以通过唯一的名称来引用相同的数据源，并且可以保证他们的定义不与任何其他的定义冲突。

基于 *Java* 的配置

如果你想使用基于 *Java* 的配置，*@Bean* 注解可以用来提供别名，详细信息请看 *Section 6.12.3, “Using the @Bean annotation”*

实例化bean

bean 定义基本上就是用来创建一个或多个对象的配置，当需要一个 bean 的时候，容器查看配置并且根据 bean 定义封装的配置元数据创建（或获取）一个实际的对象。

如果你使用基于 XML 的配置，你可以在 `<bean/>` 元素通过 class 属性来指定对象的类型。这个 class 属性，实际上是 BeanDefinition 实例中的一个 Class 属性。这个 class 属性通常是必须的（例外情况，查看 “使用实例工厂方法实例化” 章节 和 Section 6.7, “Bean定义的继承”），使用 Class 属性的两种方式：

- 通常情况下，直接通过反射调用构造方法来创建 bean，和在 Java 代码中使用 new 有点像。
- 通过静态工厂方法创建，类中包含静态方法。通过调用静态方法返回对象的类型可能和 Class 一样，也可能完全不一样。

内部类名。如果你想配置使用静态的内部类，你必须用内部类的二进制名称。例如，在 *com.example* 包下有个 *Foo* 类，这里类里面有个静态的内部类*Bar*，这种情况下*bean*定义的*class*属性应该*…com.example.Foo$Bar* 注意，使用*$*字符来分割外部类和内部类的名称。

通过构造函数实例化

当你使用构造方法来创建 bean 的时候，Spring 对类来说并没有什么特殊。也就是说，正在开发的类不需要实现任何特定的

接口或者以特定的方式进行编码。但是，根据你使用那种类型的 IoC 来指定 bean，你可能需要一个默认（无参）的构造方法。

Spring IoC 容器可以管理几乎所有你想让它管理的类，它不限于管理POJO。大多数 Spring 用户更喜欢使用 POJO（一个默认无参的构造方法和setter,getter方法）。但在容器中使用非 bean 形式(non-bean style)的类也是可以的。比如遗留系统中的连接池，很显然它与 JavaBean规范不符，但 Spring 也能管理它。

当使用基于XML的元数据配置文件，可以这样来指定 bean 类：

```
<bean id="exampleBean" class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

给构造方法指定参数以及为bean实例化设置属性将在后面的依赖注入中详细说明。

使用静态工厂方法实例化

当采用静态工厂方法创建 bean 时，除了需要指定 class 属性外，还需要通过 factory-method 属性来指定创建 bean 实例的工厂方法。Spring将调用此方法(其可选参数接下来介绍)返回实例对象，就此而言，跟通过普通构造器创建类实例没什么两样。

下面的 bean 定义展示了如何通过工厂方法来创建bean实例。注意，此定义并未指定返回对象的类型，仅指定该类包含的工厂方法。在此例中，createInstance() 必须是一个 static 方法。

```
<bean id="clientService"
    class="examples.ClientService"
    factory-method="createInstance"/>
```

```
public class ClientService {
    private static ClientService clientService = new ClientService();
    private ClientService() {}

    public static ClientService createInstance() {
        return clientService;
    }
}
```

给工厂方法指定参数以及为bean实例设置属性的详细内容请查阅依赖和配置详解。

使用实例工厂方法实例化

与通过 静态工厂方法 实例化类似，通过调用工厂实例的非静态方法进行实例化。 使用这种方式时，class属性必须为空，而factory-bean属性必须指定为当前(或其祖先)容器中包含工厂方法的bean的名称，而该工厂bean的工厂方法本身必须通过factory-method属性来设定。

```
<!-- 工厂bean，包含createInstance()方法 -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- 其他需要注入的依赖项 -->
</bean>

<!-- 通过工厂bean创建的ben -->
<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>
```

```
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();
    private DefaultServiceLocator() {}

    public ClientService createClientServiceInstance() {
        return clientService;
    }
}
```

一个工厂类也可以有多个工厂方法，如下代码所示：

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- 其他需要注入的依赖项 -->
</bean>

<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>

<bean id="accountService"
    factory-bean="serviceLocator"
    factory-method="createAccountServiceInstance"/>
```

```
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();
    private static AccountService accountService = new AccountServiceImpl();

    private DefaultServiceLocator() {}

    public ClientService createClientServiceInstance() {
        return clientService;
    }

    public AccountService createAccountServiceInstance() {
        return accountService;
    }

}
```

这种做法表明工厂bean本身也可以通过依赖注入（DI）进行管理配置。查看依赖和配置详解。

在*Spring*文档中，*factory bean*是指在*Spring*容器中配置的工厂类通过 实例 或 静态 工厂方法来创建对象。相比而言，*FactoryBean (*注意大小写*)* 代表了*Spring*中特定的 *FactoryBean*

9.1 Introduction 介绍

Aspect-Oriented Programming (面相切面编程 AOP) 用另外的一种编程架构的思考来补充 Object-Oriented Programming (面相对象编程OOP)。OOP 主要的模块单元是 class (类)，而 AOP 是 aspect（切面）。切面使得诸如事务管理等跨越多个类型和对象的关注点模块化。（这样的关注点在 AOP 的字眼里往往被称为 crosscutting （横切关注点））

AOP 是 Spring 里面的主要的组件。虽然 Spring IoC 容器没有依赖 AOP，意味着你不想用的时候也无需使用 AOP，但 AOP 提供一个非常有用的中间件解决方案来作为 Spring IoC 的补充。

Spring 2.0 AOP

Spring 2.0 引入了一种更加简单并且更强大的方式来自定义切面，用户可以选择使用 *schema-based*（基于模式）的方式或者使用 *@AspectJ* 注解样式。这两种风格都完全支持 *Advice*（通知）类型和 *AspectJ* 的切入点语言，虽然实际上仍然使用 *Spring AOP* 进行 *weaving (*织入）。

本章主要讨论 *Spring 2.0* 对基于模式和基于 *@AspectJ* 的 *AOP* 支持。 *Spring 2.0* 完全保留了对 *Spring 1.2* 的向下兼容性，下一章 *10. Spring AOP APIs* 将讨论 *Spring 1.2 API* 所提供的底层的 *AOP* 支持。

Spring中所使用的AOP：

- 提供声明式企业服务，特别是为了替代 EJB 声明式服务。最重要的服务是 declarative transaction management（声明性事务管理）， 这个服务建立在Spring的抽象事务管理（transaction abstraction）之上。
- 允许用户实现自定义的切面，用 AOP 来完善 OOP 的使用。

如果你只打算使用通用的声明式服务或者预先打包的声明式中间件服务，例如 *pooling*（缓冲池），你可以不直接使用 *AOP*，可以忽略本章大部分内容

9.1.1 AOP concepts 概念

首先让我们从定义一些重要的 AOP 概念开始。这些术语不是 Spring 特有的。 不幸的是，Spring 术语并不是特别的直观；如果 Spring 使用自己的术语，将会变得更加令人困惑。

- Aspect（切面）： 一个关注点的模块化，这个关注点可能会横切多个对象。事务管理是 企业级 Java 应用中一个关于横切关注点的很好的例子。 在 Spring AOP 中，切面可以使用通用类（ schema-based基于模式的风格） 或者在普通类中以 @Aspect 注解（@AspectJ 注解样式）来实现。
- Join point (连接点）： 在程序执行过程中某个特定的点，比如某方法调用的时候或者处理异常的时候。 在 Spring AOP 中，一个连接点 总是 代表一个方法的执行。
- Advice（通知）： 在切面的某个特定的 join point（连接点）上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。 通知的类型将在后面部分进行讨论。许多 AOP 框架，包括 Spring，都是以拦截器做通知模型，并维护一个以连接点为中心的拦截器链。
- Pointcut（切入点）： 匹配 join point（连接点）的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行（例如，当执行某个特定名称的方法时）。 切入点表达式如何和连接点匹配是 AOP 的核心：Spring 默认使用 AspectJ 切入点语法。
- Introduction（引入）： 声明额外的方法或者某个类型的字段。 Spring 允许引入新的接口（以及一个对应的实现）到任何被 advise（通知）的对象。 例如，你可以使用一个引入来使 bean 实现 IsModified 接口，以便简化缓存机制。（在 AspectJ 社区， Introduction也被称为 inter-type declaration（内部类型声明））
- Target object（目标对象）： 被一个或者多个 aspect（切面）所advise（通知）的对象。也有人把它叫做 advised （被通知） 对象。 既然 Spring AOP 是通过运行时代理实现的，这个对象永远是一个 proxied（被代理） 对象。
- AOP proxy（AOP代理）： AOP 框架创建的对象，用来实现 aspect contract（切面契约）（包括通知方法执行等功能）。 在 Spring 中，AOP 代理可以是 JDK 动态代理或者 CGLIB 代理。
- Weaving（织入）： 把 aspect（切面）连接到其它的应用程序类型或者对象上，并创建一个被 advised （被通知）的对象。 这些可以在编译时（例如使用 AspectJ 编译器），类加载时和运行时完成。 Spring 和其他纯 Java AOP 框架一

样，在运行时完成织入。

通知的类型：

- Before advice（前置通知）：在某 join point（连接点）之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。
- After returning advice（返回后通知）：在某 join point（连接点）正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。
- After throwing advice（抛出异常后通知）：在方法抛出异常退出时执行的通知。
- After (finally) advice（最后通知）：当某join point（连接点）退出的时候执行的通知（不论是正常返回还是异常退出）。
- 环绕通知（Around Advice）：包围一个join point（连接点）的通知，如方法调用。这是最强大的一种通知类型。 环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。

环绕通知是最常用的一种通知类型。跟 AspectJ 一样，Spring 提供所有类型的通知，我们推荐你使用尽量简单的通知类型来实现需要的功能。 例如，如果你只是需要用一个方法的返回值来更新缓存，虽然使用环绕通知也能完成同样的事情， 但是你最好使用 After returning 通知而不是环绕通知。 用最合适的通知类型可以使得编程模型变得简单，并且能够避免很多潜在的错误。 比如，你不需要调用 `JoinPoint`（用于Around Advice）的 `proceed()` 方法，就不会有调用的问题。

在Spring 2.0中，所有的通知参数都是静态类型，因此你可以使用合适的类型（例如一个方法执行后的返回值类型）作为通知的参数而不是使用一个对象数组。

pointcut（切入点）和 join point（连接点）匹配的概念是 AOP 的关键，这使得 AOP 不同于其它仅仅提供拦截功能的旧技术。 切入点使得 advice （通知）可独立于 OO 层次。 例如，一个提供声明式事务管理的around 通知可以被应用到一组横跨多个对象中的方法上（例如服务层的所有业务操作）。

9.1.2 Spring AOP capabilities and goals 功能和目标

Spring AOP 用纯 Java 实现。它不需要专门的编译过程。Spring AOP 不需要控制类装载器层次，因此它适用于 Servlet 容器或应用服务器。

Spring 目前仅支持使用方法调用作为join point（连接点）（在 Spring bean 上通知方法的执行）。 虽然可以在不影响到 Spring AOP核心 API 的情况下加入对成员变量拦截器支持，但 Spring 并没有实现成员变量拦截器。 如果你需要通知对成员变量的访问和更新连接点，可以考虑其它语言，例如 AspectJ。

Spring 实现 AOP 的方法跟其他的框架不同。Spring 并不是要尝试提供最完整的 AOP 实现（尽管 Spring AOP 有这个能力）， 相反的，它其实侧重于提供一种 AOP 实现和 Spring IoC 容器的整合，用于帮助解决在企业级开发中的常见问题。

因此，Spring AOP 通常都和 Spring IoC 容器一起使用。 Aspect 使用普通的bean 定义语法（尽管 Spring 提供了强大的“autoproxying（自动代理）”功能）：与其他 AOP 实现相比这是一个显著的区别。有些事使用 Spring AOP 是无法轻松或者高效的完成的，比如说通知一个细粒度的对象。 这种时候，使用 AspectJ 是最好的选择。不过经验告诉我们： 于大多数在企业级 Java 应用中遇到的问题，只要适合 AOP 来解决的，Spring AOP 都没有问题，Spring AOP 提供了一个非常好的解决方案。

Spring AOP 从来没有打算通过提供一种全面的 AOP 解决方案来取代AspectJ。 我们相信无论是 proxy-based（基于代理 ）的框架比如说Spring AOP 亦或是 full-blown 的框架比如说是 AspectJ 都是很有价值的，他们之间的关系应该是互补而不是竞争的关系。 Spring 可以无缝的整合 Spring AOP，IoC 和 AspectJ，使得所有的 AOP 应用完全融入基于 Spring 的应用体系。 这样的集成不会影响 Spring AOP API 或者AOP Alliance API；Spring AOP保留了向下兼容性。下一章 10. Spring AOP APIs会详细讨论 Spring AOP API。

一个 *Spring Framework* 的核心原则是 *non-invasiveness(*非侵袭性*)*；这意味着你不应该在您的业务/域模型被迫引入框架特定的类和接口。然而，在一些地方，*Spring Framework* 可以让你选择引入 *Spring Framework* 特定的依赖关系到你的代码，给你这样选择的理由是因为在某些情况下它可能是更容易读或编写一些特定功能。*Spring Framework* （几乎）总是给你的选择：你可以自由的做出明智的决定，选择最适合您的特定用例或场景。

这样的选择与本章有关的是 *AOP* 框架（和 *AOP* 类型）选择。你可以选择*AspectJ* 和/或 *Spring AOP*，你也可以选择 *@AspectJ* 注解式的方法或*Spring* 的 *XML* 配置方式。事实上，本章以介绍 *@AspectJ* 风格为先不应该被视为 *Spring* 团队倾向于 *@AspectJ* 的方法胜过在 *Spring* 的*XML* 配置方式。

见*9.4. Choosing which AOP declaration style to use*里面有更完整的每种风格的使用原因探讨。

9.1.3 AOP Proxies 代理

Spring缺省使用标准 JDK dynamic proxies（动态代理）来作为AOP的代理。这样任何接口（或者接口的 set）都可以被代理。

Spring 也支持使用 CGLIB 代理. 对于需要代理类而不是代理接口的时候CGLIB 代理是很有必要的。 如果一个业务对象并没有实现一个接口，默认就会使用 CGLIB。 此外，面向接口编程 也是一个最佳实践，业务对象通常都会实现一个或多个接口。此外，还可以强制的使用 CGLIB，在那些（希望是罕见的）在你需要通知一个未在接口中声明的方法的情况下，或者当你需要传递一个代理对象作为一种具体类型到方法的情况下。

一个重要的事实是，Spring AOP 是 proxy-based (基于代理)。见第9.6.1，“理解 AOP 代理”理解这个含义

15.1 Introduction to ORM with Spring 介绍 Spring 中的 ORM

Spring Framework 支持集成 Hibernate, Java Persistence API (JPA) 和 Java Data Objects (JDO) 用于资源管理、数据访问对象(DAO)的实现,和事务策略。例如,对于 Hibernate 有一流的支持，使用方便的 IoC 特性,解决许多典型的 Hibernate 集成问题。您可以通过依赖注入配置的所有 O/R (对象关系) 映射工具的特性。他们可以参与 Spring 的资源和事务管理,他们符合 Spring 的通用事务和 DAO 异常层次结构。建议集成风格是在 Hibernate, JPA, 和 JDO APIs 中使用 DAO 。旧的 Spring DAO 模板不再推荐使用;然而,关于风格的论述可以见32.1节,“经典的 ORM 使用”。

当你创建数据访问应用时，Spring 能对你选择的 ORM 层进行显着的增强。你根据你的需求进行尽可能多的集成支持，同时需要比这种整合在建一个类似的基础设施时的内部风险。通过库，你可以使用很多的 ORM 的支持，不用管技术，因为一切都是设计成一组可重用的 JavaBean。ORM 在 Spring IoC 容器便于配置和部署。因此，本文中的大多数示例配置显示在 Spring 容器里。

使用 Spring Framework 来创建您的 ORM DAO 好处包括:

- 更容易测试。Spring 的 IoC 方法便于交换实现和配置 Hibernate `SessionFactory` 实例,JDBC `DataSource` (数据源)实例,事务管理,以及映射对象实现(如果需要)。这反过来使其更容易隔离测试每一块持续相关代码。
- 常见的数据访问异常。Spring 可以从你的 ORM 工具包装异常,将他们从专有的(可能检查)异常转为共同运行时 DataAccessException 层次。这个特性允许您处理大多数持久化的异常不可恢复的，而且只出现在适当的层,没有恼人的捕捉、抛出、和异常声明。当然根据需要你仍然可以捕获和处理异常。记住,JDBC 异常(包括数据库特殊的方言)也转换为相同的层次结构,这意味着您可以在使用 JDBC 执行一些操作时拥有一致的编程模型。
- 通用资源管理。Spring 应用程序上下文可以处理的定位和配置Hibernate `SessionFactory` 实例,JPA `EntityManagerFactory` 实例,JDBC `DataSource` (数据源)实例,和其他相关资源。这使得这些值易于管理和改变。Spring 提供了高效、简单和安全处理持久性的资源。例如,关联代码来使用 Hibernate ，通常需要使用相同的 Hibernate `Session` ,以确保高效和适当的事务处理。Spring 很容易创建和透明地将 `Session` 绑定到当前线程,通过使用 Hibernate `SessionFactory` 来暴露出当前 `Session` 。因此，针对任何本地或 JTA 事务环境，Spring 解决许多在典型的 Hibernate 使用中的惯性问题。
- 集成事务管理。可以通过声明式的,面向方面的编程(AOP)风格方法拦截器包装你的 ORM 代码通过吗，可以采用 `@Transactional` 注释或通过显式配置事务 AOP advice 在一个 XML 配置文件中。在这两种情况下,都是可以为您处理事务语义和异常处理(回滚,等等)。如下面所讨论的资源和事务管理,你也可以交换不同的事务管理器,而不影响 ORM 相关的代码。例如,您可以在相同的全面服务(如声明性事务)的两个场景，实现本地事务和 JTA 之间交换。另外,JDBC 相关的代码可以完全集成事务到你使用的 ORM 代码中。这是对于数据访问有用,但不适合 ORM 中的批处理和 BLOB 流等,这仍然需要在ORM操作 交换共同的事务。

为更全面了解的 *ORM* 的支持,包括支持替代数据库技术,如 *MongoDB* 数据库,您可能希望查看 *Spring Data* 配套的项目。如果你是一个 *JPA* 用户,*Getting Started Accessing Data with JPA*提供了一个很好的介绍。

15.2 General ORM integration considerations常见的 ORM 集成方面的注意事项

本节强调的 注意事项应用与所有的 ORM 技术。15.3. Hibernate这节提供了更多细节，同时展示了这些特性和具体上下文的配置。

Spring 的 ORM 集成的主要目标是明确的应用程序分层,包括在任何数据访问、事务技术和松耦合的应用程序对象。没有更多的业务服务依赖于数据访问或事务策略,不再资源查找硬编码,不再强制替换单例,没有更多的自定义服务注册。一个简单的和一致的方法来连接应用程序对象,让他们尽可能的对容器依赖是可以重用并且是自由。所有个人数据访问特性可用的,但可以与 Spring 应用程序上下文的概念集成,提供基于 xml 的配置和交叉引用的普通 JavaBean 实例而不需要 Spring-aware（Spring 的意识）。在一个典型的 Spring 应用程序中,许多重要的对象是 JavaBean:数据访问模板,数据访问对象,事务管理器,使用数据访问对象和事务管理器的业务服务, web 视图解析器,使用业务服务的 web 控制器,等等。

15.2.1 Resource and transaction management 资源和事务管理

典型的商业应用是用重复的资源管理代码杂乱的堆积起来的。很多项目试图创造自己的解决方案，有时为了编程方便来牺牲故障的处理。Spring 提倡简单的处理适当资源的方案，即在JDBC的案例 IoC 通过模板和在 ORM 技术应用 AOP 拦截器。

基础设施提供适当的资源处理，并且能将特定的 API 异常适当的转换为一个未检查的基础的异常层次结构。Spring 引入了 DAO 异常层次结构，适用于任何的数据访问策略。对于直接的 JDBC，在前一节提到的 `JdbcTemplate` 类提供了连接处理和将 `SQLException` 适当的转换为 `DataAccessException` 层次结构，其中包括将 具体数据库 SQL 错误代码转为有意义的异常类。对于 ORM 技术，见下一节，如何得到相同异常转换的好处。

当涉及到事务管理， `JdbcTemplate` 类与 Spring 的事务支持挂钩，并且通过各自的 Spring 事务管理器支持 JTA 和 JDBC 事务。为支持 ORM 技术 Spring 提供了通过与 JTA 支持类似的 Hibernate，JPA，和 JDO 事务管理器来实现对 Hibernate，JPA 和 JDO 支持。更多事务的支持，详细信息，参见第12章，事务管理。

15.2.2 Exception translation 异常转化

当你在 DAO 中使用 Hibernate、JPA 或 JDO 时,你必须决定如何处理持久化技术的原生异常类。运用不同的技术，DAO 会抛出 `HibernateException`、 `PersistenceException` 或 `JDOException` 的子类。这些异常都是运行时的异常,不需要声明或捕获。你可能必须处理 `IllegalArgumentException` 和 `IllegalStateException`。这意味着调用者只能将异常处理成为一个通常为致命的问题,除非他们想要依赖于持久化技术自身的异常结构。捕捉乐观锁定失败等具体原因是不可能的除非把调用者与实现策略相联系。取消这交换是可接受的对于应用程序是基于 ORM 和/或 不需要任何特殊的异常处理。然而,Spring 通过 `@Repository` 注解 来使异常透明的转化:

```
@Repository
public class ProductDaoImpl implements ProductDao {

    // class body here...

}
```

```
<beans>

    <!-- Exception translation bean post processor -->
    <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

postprocessor 会自动寻找所有异常转换器(实现 `PersistenceExceptionTranslator` 接口),建议所有 bean 标有 `@Repository` 注解,以便发现的转换器可以在抛出异常时拦截和应用适当的转换。

总之:您可以实现 DAO 基于纯持久化技术的API和注解,同时仍然受益于Spring 管理事务,依赖注入、和透明将异常转换(如果需要)为 Spring 的自定义的异常层次结构。

15.3 Hibernate

现在要开始谈下 Spring 环境中的 Hibernate 3，用它来展示 Spring 对集成 O/R 映射的方法。本节将详细讨论许多问题,并显示不同的 DAO 实现和事务界定。大多数这些模式可以直接从所有其他支持 ORM 的工具中转换。以下部分在本章将覆盖其他的 ORM 技术,并显示简短的例子。

对于*Spring 4.0, Spring* 需要 *Hibernate 3.6* 或者更高版本

15.3.1 SessionFactory setup in a Spring container 在 Spring 容器中设置 SessionFactory

为了避免应用程序对象与硬编码的资源查找想绑定，您可以定义资源如 JDBC `DataSource` 或者 Hibernate `SessionFactory` 为 Spring 容器的 bean。应用对象需要通过对 bean 的引用来访问资源接受引用，就如在下一节中说明的 DAO 的定义。

以下摘录自 XML 应用程序上下文定义，展示了如何设置 JDBC `DataSource` 或者 Hibernate `SessionFactory`：

```
<beans>

    <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
        <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
        <property name="username" value="sa"/>
        <property name="password" value=""/>
    </bean>

    <bean id="mySessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="myDataSource"/>
        <property name="mappingResources">
            <list>
                <value>product.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <value>
                hibernate.dialect=org.hibernate.dialect.HSQLDialect
            </value>
        </property>
    </bean>

</beans>
```

从 Jakarta Commons DBCP `BasicDataSource` 转为 JNDI-located `DataSource`, 主要的配置为：

```
<beans>
    <jee:jndi-lookup id="myDataSource" jndi-name="java:comp/env/jdbc/myds"/>
</beans>
```

您还可以访问 JNDI-located `SessionFactory`,使用 Spring 的 `JndiObjectFactoryBean` / `<jee:jndi-lookup>` 来检索和暴露他。然而,通常在 EJB 环境外不常用。

15.3.2 Implementing DAOs based on plain Hibernate 3 API

基于平常的 Hibernate 3 API 来实现 DAO

Hibernate 3 有一个特性称为上下文会话,Hibernate 本身在每个事务管理一个当前 `Session` 。这是大致相当于 Spring 的 每个事务一个当前 Hibernate `Session` 的同步。相应的 DAO 实现像下面的例子,基于普通Hibernate API:

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public Collection loadProductsByCategory(String category) {
        return this.sessionFactory.getCurrentSession()
                .createQuery("from test.Product product where product.category=?")
                .setParameter(0, category)
                .list();
    }
}
```

这种风格类似于 Hibernate 参考文档和例子,除了在一个实例变量中保存 `SessionFactory`。我们强烈建议基于实例的设置,替换老派的Hibernate 的 CaveatEmptor 示例应用程序中的 `static HibernateUtil` 类。(一般来说,不保留任何资源 `static` 变量,除非绝对必要)

上面的 DAO 是依赖注入模式:这正好符合 Spring IoC 容器,就像对Spring 的 `HibernateTemplate` 编码。当然,这种 DAO 也可以在普通的 Java 设置(例如,在单元测试)。简单的实例化,并用所需的工厂引用调用 `setSessionFactory(..)`。Spring bean 定义 DAO 就像下面一样:

```
<beans>

    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="sessionFactory" ref="mySessionFactory"/>
    </bean>

</beans>
```

DAO 风格的主要优点是，它只依赖于 Hibernate API，而没有引进任何Spring 必需的类。从非侵入性的视角看这当然是有吸引力的，对Hibernate 开发者来说无疑会感觉更自然。

然而，DAO 将平常的 `HibernateException`（这是未检查的，所以不需要声明或者捕获），这意味着调用者当异常为一个普通的致命问题——除非他们想要依赖于 Hibernate 自身的异常结构。捕捉乐观锁定失败等具体原因是不可能除非把调用者与实现策略相联系。取消这交换是可接受的对于应用程序是基于 Hibernate 和/或 不需要任何特殊的异常处理。

幸运的是,Spring 的 `LocalSessionFactoryBean` 支持 Hibernate `SessionFactory.getCurrentSession()` 方法用于任何 Spring 事务策略,返回当前 Spring 管理的事务 `Session` 即使是 `HibernateTransactionManager.`。当然,这种方法的标准行为返回仍然是当前 `Session` 与持续的JTA事务有关。这种行为适用于不管您使用是Spring 的 `JtaTransactionManager`,EJB容器管理的事务(CMT),或 JTA。

总之:你可以基于平常的 Hibernate 3 API 来实现 DAO,同时仍然能够参与 Spring 管理事务。

15.3.3 Declarative transaction demarcation 声明式事务划分

建议你使用 Spring 声明式事务的支持,这使您能够代替显式事务划分 API调用 AOP 事务拦截器中的 Java 代码。这个事务拦截器可以配置 Spring容器通过使用 Java 注释或 XML。这个声明式事务能力允许您保持业务服务中的重复性事务划分代码码更自由，并且让你只关注添加业务逻辑,而这是您的应用程序的真正价值。

在继续之前，强烈建议你读*12.5节，“事务管理”*如果你还没有这样做。

此外，事务语义比如传播行为和隔离水平可以在配置文件的改变，不影响业务服务的实现。

下面的示例说明如何使用 XML 配置 AOP 事务拦截器，一个简单的服务类：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- SessionFactory, DataSource, etc. omitted -->

    <bean id="transactionManager"
            class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <aop:config>
        <aop:pointcut id="productServiceMethods"
                expression="execution(* product.ProductService.*(..))"/>
        <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
    </aop:config>

    <tx:advice id="txAdvice" transaction-manager="myTxManager">
        <tx:attributes>
            <tx:method name="increasePrice*" propagation="REQUIRED"/>
            <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
            <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
        </tx:attributes>
    </tx:advice>

    <bean id="myProductService" class="product.SimpleProductService">
        <property name="productDao" ref="myProductDao"/>
    </bean>

</beans>
```

下面是要处理的服务类

```
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    // notice the absence of transaction demarcation code in this method
    // Spring's declarative transaction infrastructure will be demarcating
    // transactions on your behalf
    public void increasePriceOfAllProductsInCategory(final String category) {
        List productsToChange = this.productDao.loadProductsByCategory(category);
        // ...
    }
}
```

我们还展示了一个基于配置属性的支持，在下面的例子中。你通过 `@Transactional` 注释的服务层,并引导 Spring 容器找到这些注释，这些注释的方法提供事务性语义。

```
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }
```

```
    @Transactional
    public void increasePriceOfAllProductsInCategory(final String category) {
        List productsToChange = this.productDao.loadProductsByCategory(category);
        // ...
    }

    @Transactional(readOnly = true)
    public List<Product> findAllProducts() {
        return this.productDao.findAllProducts();
    }

}
```

正如你可以看到下面的配置实例，配置更加简化，与上述 XML 实例，同时还提供了在服务层的代码注释驱动相同的功能。所有您需要提供的是TransactionManager 的实现和"" 实体

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- SessionFactory, DataSource, etc. omitted -->

    <bean id="transactionManager"
            class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <tx:annotation-driven/>

    <bean id="myProductService" class="product.SimpleProductService">
        <property name="productDao" ref="myProductDao"/>
    </bean>

</beans>
```

15.3.4 Programmatic transaction demarcation 编程式事务划分

可以在高级的应用中划分事务，在这样的底层数据访问服务生成任意数量的操作。限制也不存在于周边业务服务的实现;它只需要一个 Spring `PlatformTransactionManager`。再次,后者可以来自任何地方,但最好是通过 `setTransactionManager(..)` 方法来对 bean 引用,正如由 `setProductDao(..)` 方法来设置 `productDAO`。下面的代码片段显示了在Spring 应用程序上下文中定义一个事务管理器和业务服务,以及一个业务方法实现:

```
    <beans>

        <bean id="myTxManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
            <property name="sessionFactory" ref="mySessionFactory"/>
        </bean>

        <bean id="myProductService" class="product.ProductServiceImpl">
            <property name="transactionManager" ref="myTxManager"/>
            <property name="productDao" ref="myProductDao"/>
        </bean>

    </beans>
```

```
    public class ProductServiceImpl implements ProductService {

        private TransactionTemplate transactionTemplate;
        private ProductDao productDao;

        public void setTransactionManager(PlatformTransactionManager transactionManager) {
            this.transactionTemplate = new TransactionTemplate(transactionManager);
        }

        public void setProductDao(ProductDao productDao) {
            this.productDao = productDao;
        }

        public void increasePriceOfAllProductsInCategory(final String category) {
            this.transactionTemplate.execute(new TransactionCallbackWithoutResult() {
                public void doInTransactionWithoutResult(TransactionStatus status) {
                    List productsToChange = this.productDao.loadProductsByCategory(category);
                    // do the price increase...
                }
            });
        }
    }
```

Spring `TransactionInterceptor` 允许任何检查的应用异常可以跟着回调代码抛出，而 `TransactionTemplate` 是限制于未检查的回调的异常。当遇到一个未检查的应用异常或者是事务被应用（通过 `TransactionStatus`）标记为 rollback-only（只回滚）, `TransactionTemplate` 触发回滚事务。默认时 `TransactionInterceptor` 表现是一样的，但是允许在每个方法中配置回滚策略。

15.3.5 Transaction management strategies 事务管理策略

`TransactionTemplate` 和 `TransactionInterceptor` 代表了对 `PlatformTransactionManager` 实例的实际事务处理，在 Hibernate 的应用中它们可以是一个 `HibernateTransactionManager`（一个Hibernate 的 `SessionFactory`，在引擎下使用的是 `ThreadLocal Session`）或 `JtaTransactionManager`（委派到容器的 JTA 子系统）。你甚至可以使用一个自定义的 `PlatformTransactionManager` 实现。从原生 Hibernate 事务管理 转到 JTA，如你的某些应用程序部署具有分布式事务处理的要求，那么这仅仅是一个配置的问题，只要将 Hibernate 事务管理简单的替换为 Spring 的 JTA 即可。两个的事务划分和数据访问代码的不用改变，因为他们只是使用了通用的事务管理 API。

对于分布式事务跨多个 Hibernate 会话工厂，只要简单地把 `JtaTransactionManager` 与具有多个定义的 `LocalSessionFactoryBean` 组合成为一个事务策略。每个 DAO 得到一个特定的 `SessionFactory` 的引用传递到其相应的 bean 属性。如果所有底层的 JDBC 数据源的事务容器，业务服务可以划分事务到任意数量的DAO 和任何数量的会话工厂，而这无需没有特殊的处理，只要是使用 `JtaTransactionManager` 策略。

```
<beans>

    <jee:jndi-lookup id="dataSource1" jndi-name="java:comp/env/jdbc/myds1"/>

    <jee:jndi-lookup id="dataSource2" jndi-name="java:comp/env/jdbc/myds2"/>

    <bean id="mySessionFactory1"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="myDataSource1"/>
        <property name="mappingResources">
            <list>
                <value>product.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <value>
                hibernate.dialect=org.hibernate.dialect.MySQLDialect
                hibernate.show_sql=true
            </value>
        </property>
    </bean>

    <bean id="mySessionFactory2"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="myDataSource2"/>
```

```
        <property name="mappingResources">
            <list>
                <value>inventory.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <value>
                hibernate.dialect=org.hibernate.dialect.OracleDialect
            </value>
        </property>
    </bean>

    <bean id="myTxManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>

    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="sessionFactory" ref="mySessionFactory1"/>
    </bean>

    <bean id="myInventoryDao" class="product.InventoryDaoImpl">
        <property name="sessionFactory" ref="mySessionFactory2"/>
    </bean>

    <bean id="myProductService" class="product.ProductServiceImpl">
        <property name="productDao" ref="myProductDao"/>
        <property name="inventoryDao" ref="myInventoryDao"/>
    </bean>

    <aop:config>
        <aop:pointcut id="productServiceMethods"
                expression="execution(* product.ProductService.*(..))"/>
        <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
    </aop:config>

    <tx:advice id="txAdvice" transaction-manager="myTxManager">
        <tx:attributes>
            <tx:method name="increasePrice*" propagation="REQUIRED"/>
            <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
            <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
        </tx:attributes>
    </tx:advice>

</beans>
```

`HibernateTransactionManager` 和 `JtaTransactionManager` 允许适当的 JVM 级别的 Hibernate 缓存处理,而不需要容器特定的事务管理查找或 JCA 连接器(如果你不使用 EJB 启动事务)。

`HibernateTransactionManager` 可以为一个特定的 `DataSource` 导出 Hibernate JDBC `Connection` 到普通 JDBC 访问代码。此功能允许高级的混合 Hibernate 和 JDBC 数据访问的完全没有 JTA 的事务划分,如果你只访问一个数据库。如果你有设置通过 `LocalSessionFactoryBean` 类的 `dataSource` 属性传入 `SessionFactory` 的 `DataSource`， `HibernateTransactionManager` 自动暴露 Hibernate 事务作为一个 JDBC 事务。或者,您可以明确指定 `DataSource` 中哪些事务是需要支持通过 HibernateTransactionManager 类的 `dataSource` 属性暴露的。

15.3.6 Comparing container-managed and locally defined resources 比较容器管理和本地定义的资源

你可以在一个容器管理的 JNDI `SessionFactory` 和本地定义的互相切换,而无需更改应用程序的代码。是否保持资源定义在容器或本地应用程序中，主要取决于使用的事务策略。对比 Spring 定义的本地的 `SessionFactory`,手动注册 JNDI `SessionFactory` 没有任何好处。部署一个通过 Hibernate JCA 连接器的 `SessionFactory` 来提供 Java EE 服务器的管理基础设施的附加值,但在这之前不增加任何实际的值。

Spring 的事务支持不是绑定到一个容器中。在配置除了 JTA 以外的任何策略后，事务支持同样也能在一个独立的或测试的环境中工作。特别是在单独的数据库事务的典型应用中。Spring 是一个轻量级的单资源本地事务支持和强大的 JTA 的替代品。当你使用本地 EJB 无状态会话 bean 来驱动事务,你都必须要依赖 EJB 容器和 JTA,即使你只访问一个数据库,并且只使用无状态会话 bean 通过容器管理的事务来提供声明式事务。另外,直接使用 JTA 编程需要一个 Java EE 环境。JTA 并不只涉及容器依赖性的JTA 本身和 JNDI `DataSource` 实例。对于 非 Spring,JTA 驱动的 Hibernate 事务交易,您必须使用 Hibernate JCA 连接器,或额外的Hibernate 事务代码 `TransactionManagerLookup` 为适当的 JVM 级别配置缓存。

Spring 驱动事务可以与本地定义的 Hibernate `SessionFactory` 和本地的JDBC `DataSource` 很好的工作，如果他们访问一个数据库。因此你只需要使用 Spring 的 JTA 事务策略，在当你有分布式事务的需求的时候。JCA 连接器需要特定容器部署步骤，显然首先需要的是 JCA 的支持。这个配置比部署一个简单的 使用本地资源定义和 Spring 驱动事务 web 应用程序需要更多的工作。同样,你经常需要你的容器是使用的是企业版,例如,WebLogic Express, 并且是不提供 JCA。Spring 的应用程序具有本地资源和事务跨越数据库的能力，可以在任何 Java EE web容器(没有 JTA、JCA 或 EJB)如Tomcat、Resin、甚至普通的 Jetty 中工作。此外,您可以很容易地重用这样的一个中间层在桌面应用程序或测试套件中。

从全面考虑,如果你不使用 EJB,请坚持使用本地 `SessionFactory` 设置和 Spring 的 `HibernateTransactionManager` 或 `JtaTransactionManager` 。你得到所有的好处,包括适当的事务 JVM 级别缓存和分布式事务,没有容器部署的不便。JNDI 通过 JCA 连接器注册 Hibernate `SessionFactory` ,在与 EJB 一起使用时只是增加了值。

15.3.7 Spurious application server warnings with Hibernate 在 Hibernate 中的虚假应用服务器告警

在一些 JTA 的非常严格的 `XADataSource` 实现的环境中 — 目前只在一些 WebLogic Server 和 WebSphere 版本中 — 当 Hibernate 配置时没有留意环境中的 JTA 的 `PlatformTransactionManager` 对象，这可能会导致 虚假告警或者异常显示在应用服务器的日志中。这些告警或者异常显示连接访问不再有效,或 JDBC 访问不再有效,这可能是因为事务已经不再活动了。举个例子,这是一个真实的 WebLogic 异常:

```
java.sql.SQLException: The transaction is no longer active - status: Committed. No
further JDBC access is allowed within this transaction.
```

要解决此警告，只需要使 Hibernate 知道 JTA `PlatformTransactionManager` 实例，它将同步（连同 Spring）。实现这个有两个选项：

- 如果在你的应用程序上下文中你已经直接获取 JTA `PlatformTransactionManager` 对象（大概是从 JNDI 通过 `JndiObjectFactoryBean` 或 `<jee:jndi-lookup>` ）将它提供给，例如，Spring 的 `JtaTransactionManager` ，那么最简单的方法是通过引用定义了这个 JTA `PlatformTransactionManager` 实例的 bean给 `LocalSessionFactoryBean` 指定一个 `jtaTransactionManager` 的属性值。那么 Spring 就会使对象在 Hibernate 中可用。
- 你很有可能没有 JTA `PlatformTransactionManager` 实例，因为Spring 的 `JtaTransactionManager` 本身可以找到它。因此，你需要配置 Hibernate 直接查找 JTA `PlatformTransactionManager` 。你可以在 Hibernate 配置中通过配置应用程序服务器特定的 `ransactionManagerLookup` 类实现这个，正如 Hibernate 手册所描述的那样。

本节的其余部分描述了事件发生的顺序和 Hibernate 对 JTA `PlatformTransactionManager` 的认知。

当 Hibernate 没有配置任何 JTA `PlatformTransactionManager` 的认知时,当一个 JTA 事务提交时以下事件发生:

- JTA 事务提交。
- Spring 的 `JtaTransactionManager` 与 JTA 事务同步时，通过 JTA 事务管理执行一个 afterCompletion 的回调。
- 在其他活动中，从 Spring 到 Hibernate 的同步可以触发回调，通过Hibernate 的 `afterTransactionCompletion` 回调（用于清除 Hibernate 缓存），随后的是一个显式 close() 调用在 Hibernate Session,,导致 Hibernate 试图 close() JDBC 连接。
- 在某些环境中,这个 `Connection.close()` 调用然后触发警告或错误,因为应用程序服务器不再认为 `Connection` 是可用的,因为事务已经提交了。

当Hibernate 配置了 JTA `PlatformTransactionManager` 的认知,当一个JTA事务提交，以下事件发生：

- JTA 事务准备提交。
- Spring 的 `JtaTransactionManager` 跟 JTA 事务是同步的,所以通过 JTA 事务管理器的 `beforeCompletion` 回调来执行事务的回调。
- Spring 感知到 Hibernate 本身与 JTA 事务是同步的,并且行为不同于在前面的场景。假设需要 Hibernate `Session` 关闭,那么 Spring将会关闭它。
- JTA 事务提交。

- Hibernate 与 JTA 事务是同步的,所以通过 JTA 事务管理器的 `beforeCompletion` 回调来执行事务的回调，并能正确清楚其缓存。

15.4 JDO

Spring 支持标准的 JDO 2.0 和 2.1 API 的数据访问策略，按照与 Hibernate 同样的支持方式。相应的集成类驻留在 `org.springframework.orm.jdo` 包。

15.4.1 PersistenceManagerFactory setup 设置

Spring提供 `LocalPersistenceManagerFactoryBean` 类允许您在一个 Spring 应用上下文定义了一个局部的 JDO 的 `PersistenceManagerFactory` ：

```
<beans>

    <bean id="myPmf" class="org.springframework.orm.jdo.LocalPersistenceManagerFactoryBean">
        <property name="configLocation" value="classpath:kodo.properties"/>
    </bean>

</beans>
```

另外，你可以通过一个 `PersistenceManagerFactory` 实现类的的实例化来设置 `PersistenceManagerFactory` 。一个 JDO 的 `PersistenceManagerFactory` 实现类遵循 JavaBean 模式，就像一个JDBC `DataSource` 的实现类，这是在 Spring 里配置使用是非常合适的。这种设置方式通常支持一个 Spring 定义的JDBC `DataSource` ，传递给 `connectionFactory` 。例如，对于开源的 JDO 实现DataNucleus（原名 JPOX ）（http://www.datanucleus.org/），下面是 `PersistenceManagerFactory` 实现的 XML 配置：

```
<beans>

 <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
 </bean>

 <bean id="myPmf" class="org.datanucleus.jdo.JDOPersistenceManagerFactory" destroy-method="close">
    <property name="connectionFactory" ref="dataSource"/>
    <property name="nontransactionalRead" value="true"/>
 </bean>

</beans>
```

也可以在 Java EE 应用服务的 JNDI 环境中 设置 JDO `PersistenceManagerFactory` ，通常是通过 JCA 连接器提供包含 JDO 的实现。Spring 的标准中 `JndiObjectFactoryBean` 或 `<jee:jndi-lookup>` 可以用来检索和暴露比如 `PersistenceManagerFactory` 。然而，在 EJB 上下文 之外,没有真正的存在于在 JNDI 中保持 `PersistenceManagerFactory` :只选择这样的一个设置是一个很好的理由。请参见15.3.6“比较容器管理和本地定义的资源”讨论;那里的论点适用于JDO。

15.4.2 Implementing DAOs based on the plain JDO API 基于平常 JDO API 的 DAO 的实现

利用注入的 `PersistenceManagerFactory` ，也可以直接利用平常的JDO API来写 DAO，而无需 Spring 的依赖。以下是相应的 DAO 实现的一个例子：

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
```

```
        this.persistenceManagerFactory = pmf;
    }

    public Collection loadProductsByCategory(String category) {
        PersistenceManager pm = this.persistenceManagerFactory.getPersistenceManager();
        try {
            Query query = pm.newQuery(Product.class, "category = pCategory");
            query.declareParameters("String pCategory");
            return query.execute(category);
        }
        finally {
            pm.close();
        }
    }
}
```

因为上面的 DAO 依赖注入模式，它适合在 Spring 容器中，就像在Spring 的 `JdoTemplate` 中编码：

```
<beans>

    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="persistenceManagerFactory" ref="myPmf"/>
    </bean>

</beans>
```

这样的 DAO 主要的问题是，他们总是从工厂获得一个新的 `PersistenceManager`。为了访问 Spring 管理的事务 `PersistenceManager`，需要定义一个 `TransactionAwarePersistenceManagerFactoryProxy`（包含在Spring 中）在你的目标 `PersistenceManagerFactory` 面前，然后传递一个那个代理的引用到你的 DAO，如下面的示例：

```
<beans>

    <bean id="myPmfProxy"
            class="org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy">
        <property name="targetPersistenceManagerFactory" ref="myPmf"/>
    </bean>

    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="persistenceManagerFactory" ref="myPmfProxy"/>
    </bean>

</beans>
```

你的数据访问代码将收到一个来自 `PersistenceManagerFactory.getPersistenceManager()` 调用的事务性的 `PersistenceManager`（如果有）的方法。后者的方法的调用会通过代理，在从从工厂获得一个新的之前它首先检查当前事务性的 `PersistenceManager`。由于 事务性的 `PersistenceManager`，任何 close() 的调用将会被忽略。

如果你的数据访问代码总是运行在一个活跃的事务中（或至少与活跃的事务同步），它会安全的忽略 `PersistenceManager.close()` 的调用。这样整个 `finally` 的块，可以让你的 DAO 实现更加简洁：

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.persistenceManagerFactory = pmf;
    }

    public Collection loadProductsByCategory(String category) {
        PersistenceManager pm = this.persistenceManagerFactory.getPersistenceManager();
        Query query = pm.newQuery(Product.class, "category = pCategory");
        query.declareParameters("String pCategory");
        return query.execute(category);
    }
```

```
}
```

由于这样 DAO 依来活动的事务，所有建议您通过关闭 `TransactionAwarePersistenceManagerFactoryProxy` 的 `allowCreate` 标签来强制激活事务：

```
<beans>

    <bean id="myPmfProxy"
            class="org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy">
        <property name="targetPersistenceManagerFactory" ref="myPmf"/>
        <property name="allowCreate" value="false"/>
    </bean>

    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="persistenceManagerFactory" ref="myPmfProxy"/>
    </bean>

</beans>
```

这种 DAO 风格的主要优势是,它只依赖于 JDO API;不需要引进任何的Spring 类。从非侵入性的角度来说更吸引人，并且对于 JDO 开发人员来说可能会觉得更自然。

然而,DAO 抛出平常的 `JDOException` (未检查的,因此不需要声明或捕获),这意味着调用者只能将异常当做是致命的,除非你想依靠 JDO 的异常结构。捕捉乐观锁失败等特殊原因是不可能，除非把调用者与实现策略相关联。取消这交易可能会更容易受应用程序接受，因为基于 JDO 和/或 不需要任何特殊的异常处理。

总之,你可以根据平常的 JDO API 生产 DAO ,他们仍然可以参与 Spring管理事务。这策略会可能会吸引你如果你已经熟悉了 JDO。然而,这样的DAO 抛出平常的 `JDOException` ,您必须显式地转换为 Spring 的 `DataAccessException` (如果需要)。

15.4.3 Transaction management 事务管理

如果你还没有看过 *12.5. Declarative transaction management* 声明式事务管理强烈建议你看下，获取更多*Spring* 声明式事务的支持

执行服务的事务操作，使用 Spring 常见的声明式事务功能，举例：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean id="myTxManager" class="org.springframework.orm.jdo.JdoTransactionManager">
        <property name="persistenceManagerFactory" ref="myPmf"/>
    </bean>

    <bean id="myProductService" class="product.ProductServiceImpl">
        <property name="productDao" ref="myProductDao"/>
    </bean>

    <tx:advice id="txAdvice" transaction-manager="txManager">
        <tx:attributes>
            <tx:method name="increasePrice*" propagation="REQUIRED"/>
            <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
            <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
        </tx:attributes>
    </tx:advice>
```

```
    <aop:config>
        <aop:pointcut id="productServiceMethods"
                expression="execution(* product.ProductService.*(..))"/>
        <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
    </aop:config>

</beans>
```

JDO 需要一个活动的事务来修改持久化的对象。非事务性的刷新概念并不存在于 JDO，相对于 Hibernate。为此，你需要为特定的环境设置选择的JDO 的实现。具体来说，你需要设置明确的 JTA 同步，来检测一个活跃的JTA 事务本身。这对于 Spring 的 `JdoTransactionManager` 执行的本地事务来说是没有必要的，但有必要参与 JTA 事务，不管是由 Spring `JtaTransactionManager` 驱动 还是 EJB CMT 和普通的 JTA。

`JdoTransactionManager` 能够使 JDO 事务 JDBC 访问代码f访问同一个JDBC `DataSource`，提供注册的 `JdoDialect` 支持底层的 JDBC `Connection` 检索。这是默认情况下基于JDBC 的 JDO 2.0实现

15.4.4 JdoDialect

作为一个高级功能, `JdoTemplate` 和 `JdoTransactionManager` 支持自定义 `JdoDialect` 可以传递到 `JdoDialect` 的 bean 属性。在这个场景中,DAO 不接受 `PersistenceManagerFactory` 的引用,而是一个完整的 `JdoTemplate` 实例(例如,传递到 `JdoDaoSupport` 的属性 `JdoTemplate` 中)。使用 `JdoDialect` 实现,您可以启用 Spring 的高级特性支持,通常特定于供应商的方式:

- 应用于特定的事务语义,如自定义隔离级别或事务超时
- 检索事务性的 JDBC `Connection`,用来暴露基于 JDBC 的 DAO
- 应用查询超时,自动从 Spring 管理事务超时进行计算
- 及时刷新 `PersistenceManager`,使事务变化对于基于 JDBC 的数据访问代码可见
- 从 `JDOExceptions` 向 Spring `DataAccessExceptions` 的高级转换

查看 `JdoDialect` 的 javadocs 获取更多如果使用 Spring JDO 的细节

15.5 JPA

Spring JPA,存在与 `org.springframework.orm.jpa` 包，提供方便的对于 Java Persistence API 的类似于 Hibernate 或者 JDO 的支持，为了解底层的实现，提供额外的功能。

15.5.1 Three options for JPA setup in a Spring environment 三种设置选项

Spring JPA 提供三种方式来设置 JPA `EntityManagerFactory` 用于应用程序实现实体的管理。

LocalEntityManagerFactoryBean

只在简单部署环境中，比如独立的应用程序和集成测试才使用该选项

`LocalEntityManagerFactoryBean` 创建了一个仅使用 JPA 访问数据适合部署在简单环境下的应用程序的 `EntityManagerFactory`。工厂 bean 使用 JPA `PersistenceProvider` 自动检测机制（根据 JPA 的Java SE 引导），在大多数情况下，需要指定唯一持久单元名称：

```
<beans>
    <bean id="myEmf" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="myPersistenceUnit"/>
    </bean>
</beans>
```

这种形式的 JPA 的部署是最简单和最有限的。你不能引用现有的 JDBC `DataSource` 的 bean 的定义，并且不支持全局事务的存在。此外，织入（字节码转换）持久化类是提供者特定的，往往需要一个特定的 JVM 代理在启动时指定。此选项仅适用于为 JPA 规范设计的独立的应用程序和测试环境。

Obtaining an EntityManagerFactory from JNDI 从 JNDI 中获得 EntityManagerFactory

当部署在 *Java EE 5* 服务器中使用该选项，查看你的服务器的文档来获知如何部署自定义的 *JPA* 提供者 在你的服务器中，允许不同于服务器默认的提供者。

从 JNDI 中获得 `EntityManagerFactory` (举例 在 Java EE 5 环境中)，只需简单配置 XML:

```
<beans>
    <jee:jndi-lookup id="myEmf" jndi-name="persistence/myPersistenceUnit"/>
</beans>
```

这个动作指定标准的 Java EE 5 的引导： Java EE服务器自动检测持久单元（实际上， `META-INF/persistence.xml` 文件在应用的 jar 中）和在Java EE部署描述符中的 `persistence-unit-ref` 的实体（例如，web.xml）并为这些定义环境命名上下文的位置。

在这种情况下，整个持久化单元的部署，包括织入（字节码转换）持久化类，到 Java EE 服务器。JDBC `DataSource` 是通过 JNDI 位置定义在META-INF/persistence.xml文件中。EntityManager 事务集成在服务器的 JTA 子系统中。Spring 只是使用获得的 `EntityManagerFactory`，通过依赖注入传递给应用程序对象，并且为持久单元管理事务，通常是通过 `JtaTransactionManager`。

如果多个持久单元中使用相同的应用程序， JNDI检索的持久单元的 bean 名称应与持久单元的名称匹配，应用程序引用它们，比如，在 `@PersistenceUnit` 和 `@PersistenceContext` 注解。

LocalContainerEntityManagerFactoryBean

在基于 *Spring* 的使用 *JPA* 全功能的应用环境中，使用该选项。这个包含了 *web* 容器你比如 *Tomcat* 作为具有复杂的持续性要求的单独的应用和集成测试

`LocalContainerEntityManagerFactoryBean` 给 `EntityManagerFactory` 完全控制配置和按需定制细粒度的适合的环境。 `LocalContainerEntityManagerFactoryBean` 创建基于 `persistence.xml` 文件 的 `PersistenceUnitInfo` 的实例,提供 `dataSourceLookup` 的策略,指定 `loadTimeWeaver` 。因此可以在 JNDI 外部使用自定义数据源和控制编织过程。下面的示例显示了一个典型的定义 `LocalContainerEntityManagerFactoryBean` 的 bean:

```
<beans>
    <bean id="myEmf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="someDataSource"/>
        <property name="loadTimeWeaver">
            <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
        </property>
    </bean>
</beans>
```

下面展示常见的 persistence.xml ：

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
    <persistence-unit name="myUnit" transaction-type="RESOURCE_LOCAL">
        <mapping-file>META-INF/orm.xml</mapping-file>
        <exclude-unlisted-classes/>
    </persistence-unit>
</persistence>
```

`<exclude-unlisted-classes/>` 快捷表明应该出现未扫描的注解的实体类。一个明确的 *true* 值指定 *`<exclude-unlisted-classes>true</exclude-unlisted-classes/>`* 也意味着没有扫描。 *`<exclude-unlisted-classes>false</exclude-unlisted-classes/>`* d 触发扫描；然而，它是建议干脆省略 *`exclude-unlisted-classes`* 元素如果你想扫描产生的实体类。

使用 `LocalContainerEntityManagerFactoryBean` 是最强大的 JPA 设置选项，允许丰富的在应用中本地配置。它支持连接到现有的 JDBC `DataSource` ,支持 包括 本地和全局的事务，等等。然而,它还对运行时环境的有特殊的需求,比如需要 weaving-capable（可织入的）的类载入器，当持久性提供者要求字节码转换时。

此选项可能 Java EE 5 服务器中内置 JPA 功能冲突。在一个完整的 Java EE 5 的环境下,考虑从 JNDI 获取你的 `EntityManagerFactory` 。另外, `LocalContainerEntityManagerFactoryBean` 定义中指定一个自定义 `persistenceXmlLocation` ,例如, META-INF/my-persistence.xml ,并且只包含一个描述符,这个名字在你的应用程序 jar 文件。因为 Java EE 5 服务器只查找默认 `META-INF/persistence.xml` 文件,它忽略了这些自定义持久性单元,从而避免与 Spring 驱动的 JPA 预先设置冲突。(例如, 这适用于 Resin 3.1)。

什么时候需要载入时织入?

不是所有的 *JPA* 提供者需要 *JVM* 代理;*Hibernate* 就是这样的一个例子。如果你的提供者不需要一个代理或你有其他选择,如应用增强在构建时通过一个自定义的编译器或一个 *ant* 任务,此时不应使用载入时织入。

`LoadTimeWeaver` 接口是一个 Spring 类,允许将 JPA `ClassTransformer` 实例插入一个特定的方式,这取决于环境是 web 容器或应用程序服务器。通过一个代理来挂钩 `ClassTransformers` 通常是无效的。代理工作在整个虚拟机并检查每一个加载类,通常是在生产服务器环境中不受欢迎的。

Spring 提供了许多 `LoadTimeWeaver` 各种环境的实现,允许 `ClassTransformer` 实例仅适用于每个类装入器,而不是每个 VM。

参考 AOP 章节“Spring配置”了解关于 `LoadTimeWeaver` 实现及其设置,包括泛型或定制各种平台(如 Tomcat、WebLogic、GlassFish、Resin 和JBoss)。

如上述所述部分,您可以配置一个context-wide（宽泛上下文的） `LoadTimeWeaver` 使用 `context:load-time-weaver` 元素中的 `@EnableLoadTimeWeaving` 注释。这样一个全球织入是所有 JPA `LocalContainerEntityManagerFactoryBeans` 自动捕捉到。这是设置加载时织入的首选方法,能自动识别出平台(WebLogic, GlassFish, Tomcat, Resin, JBoss 或者 VM 代理)和自动传播的织入到所有可织入的 bean 中:

```
<context:load-time-weaver/>
<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    ...
</bean>
```

不过,如果需要,可以手动通过 `loadTimeWeaver属性` 指定一个专门的织入:

```
<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="loadTimeWeaver">
        <bean class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>
    </property>
</bean>
```

无论 LTW 如何配置,使用这种技术,JPA 应用程序依赖于器可以运行在目标平台(例:Tomcat)而不需要代理的基础设施。这是非常重要的,尤其是当托管的应用程序依赖于不同的 JPA 实现，因为 JPA 转换器只在类装入器级别,因此彼此是隔离。

Dealing with multiple persistence units 处理多个持久单元

对于依赖于多个持久单元的位置的应用程序，在类路径中，存储在不同的JAR 中，例如，Spring 提供 `PersistenceUnitManager` 作为中央存储库，以避免持久单元的发现过程，它可以是昂贵的。默认的实现允许多个位置被指定，稍后被通过持久单元名称检索。（默认情况下，路径搜索的的是META-INF/persistence.xml 文件。）

```
<bean id="pum" class="org.springframework.orm.jpa.persistenceunit.DefaultPersistenceUnitManager">
    <property name="persistenceXmlLocations">
        <list>
            <value>org/springframework/orm/jpa/domain/persistence-multi.xml</value>
            <value>classpath:/my/package/**/custom-persistence.xml</value>
            <value>classpath*:META-INF/persistence.xml</value>
        </list>
    </property>
    <property name="dataSources">
        <map>
            <entry key="localDataSource" value-ref="local-db"/>
            <entry key="remoteDataSource" value-ref="remote-db"/>
        </map>
    </property>
    <!-- if no datasource is specified, use this one -->
    <property name="defaultDataSource" ref="remoteDataSource"/>
</bean>

<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceUnitManager" ref="pum"/>
    <property name="persistenceUnitName" value="myCustomUnit"/>
</bean>
```

默认的实现允许的自定义 `PersistenceUnitInfo` 实例，在他们传入 JPA 提供者之前，声明通过它的属性，影响所有的单元，或以编程方式，通过 `PersistenceUnitPostProcessor`，允许持久单元的选择。如果没有指定一个 `PersistenceUnitManager`，由 `LocalContainerEntityManagerFactoryBean` 内部创建和使用。

15.5.2 Implementing DAOs based on plain JPA 基于平常 JPA的 DAO 的实现

虽然 *`EntityManagerFactory`* 实例是线程安全的，但 *`EntityManager`* 不是。注入的 *JPA* *`EntityManager`* 的行为像一个 从应用服务器的 *JNDI* 环境中通过 *JPA* 规范定义的 *`EntityManager`* 。它代表所有调用当前事务 *`EntityManager`*，如果是的话；否则，它在每次操作时返回新创建的 *`EntityManager`*，使其线程安全。

通过注入 `EntityManagerFactory` 或 `EntityManager`，对于编写平常 JPA 代码对 Spring 没有任何依赖。 Spring 可以理解 `@PersistenceUnit` 和 `@PersistenceContext` 注解在字段和方法层面，如果启动 `PersistenceAnnotationBeanPostProcessor` 的话。普通的JPA DAO实现使用 `@PersistenceUnit` 注解可能看起来像这样:

```
public class ProductDaoImpl implements ProductDao {

    private EntityManagerFactory emf;

    @PersistenceUnit
    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.emf = emf;
    }

    public Collection loadProductsByCategory(String category) {
        EntityManager em = this.emf.createEntityManager();
        try {
            Query query = em.createQuery("from Product as p where p.category = ?1");
            query.setParameter(1, category);
            return query.getResultList();
        }
        finally {
            if (em != null) {
                em.close();
            }
        }
    }
}
```

上面的 DAO 没有依赖 Spring ,但 任然非常符合 Spring 应用的上下文。此外，该 DAO 充分利用 `EntityManagerFactory` 默认注解：

```
<beans>

    <!-- bean post-processor for JPA annotations -->
    <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

作为一种明确替代定义 `PersistenceAnnotationBeanPostProcessor`，考虑使用 Spring `context:annotation-config` 在你的应用程序环境配置。这样做自动注册所有的 Spring 基于注释的配置标准处理器，包括 `CommonAnnotationBeanPostProcessor` 等等。

```
<beans>

    <!-- post-processors for all standard config annotations -->
    <context:annotation-config/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

这样的 DAO 的主要的问题是，它总是通过工厂创建一个新的 `EntityManager` 。你可以请求一个事务 `EntityManager` （也被称为“共享 EntityManager ”因为它是一个共享的，线程安全的代理在实际事务 EntityManager 中）被注入而不是工厂来避免这种情况：

```
public class ProductDaoImpl implements ProductDao {

    @PersistenceContext
    private EntityManager em;

    public Collection loadProductsByCategory(String category) {
        Query query = em.createQuery("from Product as p where p.category = :category");
```

```
        query.setParameter("category", category);
        return query.getResultList();
    }
}
```

`@PersistenceContext` 注解有个可选属性 `type`，默认值是 `PersistenceContextType.TRANSACTION` 。默认的是你需要接收共享的 `EntityManager` 代理。 `PersistenceContextType.EXTENDED` ， 是一个完全不同的事情,这个结果对 `EntityManager` 的扩展，它不是线程安全的，因此不能用于并发访问的组件如 Spring 管理单例 bean。扩展的 EntityManager 只能用在有状态的组件，例如，驻留在一个会话上，这样EntityManager 的生命周期不依赖于当前事务，而是完全取决于应用程序。

方法和字段级别的注入

注释表明依赖注入（如 `@PersistenceUnit` 和 `@PersistenceContext` ）可应用于类中的字段或方法，因此表现方法级别的注入和字段级别的注入。字段级别的注入是简洁和容易使用而方法级别允许进一步处理注入的依赖。在这两种情况下的成员可见性（公共，保护，私人）不要紧。

那类级别的注入呢？

在 Java EE 5 平台，它们是用来声明依赖而不是资源注入

注入 `EntityManager` 是 Spring 管理的（意识到正在进行的事务）。需要注意的是，尽管新的 DAO 实现使用一个 `EntityManager` 方法注入而不是一个 `EntityManagerFactory` ，在应用程序上下文的 XML 注释的用法无需改变。

这种 DAO 风格的主要优点是，它不仅取决于 Java Persistence API;（Java 持久性API），而无需引进任何 Spring 的类。此外，作为 JPA 注释更容易理解，注解可以被 Spring 容器自动应用。这是从非侵袭性的的角度看很具有吸引力，对于 JPA 的开发人员来说可能感觉更自然。

15.5.3 Transaction Management 事务管理

如果你还没有看过 *12.5. Declarative transaction management* 声明式事务管理强烈建议你看下，获取更多*Spring* 声明式事务的支持

执行服务的事务操作，使用 Spring 常见的声明式事务功能，举例：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean id="myTxManager" class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="myEmf"/>
    </bean>

    <bean id="myProductService" class="product.ProductServiceImpl">
        <property name="productDao" ref="myProductDao"/>
    </bean>

    <aop:config>
        <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
        <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
    </aop:config>

    <tx:advice id="txAdvice" transaction-manager="myTxManager">
        <tx:attributes>
            <tx:method name="increasePrice*" propagation="REQUIRED"/>
```

```
            <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
            <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
        </tx:attributes>
    </tx:advice>

</beans>
```

Spring 的 JPA 允许配置 `JpaTransactionManager` 来暴露 JPA 事务给 JDBC 访问代码从而能够访问同一个 JDBC `DataSource`，提供注册 `JpaDialect` 支持底层的 JDBC `Connection` 检索。开箱即用，Spring 提供了 TopLink，Hibernate 和 OpenJPA 的 JPA 实现的方言。请参阅下一节 `JpaDialect` 机制。

15.5.4 JpaDialect

作为一个高级功能 `JpaTemplate`，`JpaTransactionManager` 和 `AbstractEntityManagerFactoryBean` 子类支持自定义 JpaDialect,传递到 `JpaDialect` bean 属性。在这种情况下,DAO 未得到 `EntityManagerFactory` 的引用,而是一个完整的 `JpaTemplate` 实例(例如,传递到 `JpaDaoSupport` 的 `JpaTemplate` 属性)。`JpaDialect` 实现可以使一些 Spring 支持的高级功能,通常取决于特定供应商的方式:

- 应用特定的事务语义,如自定义隔离级别或事务超时)
- 检索事务暴露于基于 JDBC 的 DAO 的 JDBC `Connection`)
- `PersistenceExceptions` 到 Spring `DataAccessExceptions` 的高级转换

这对于特殊事务语义和高级的异常转换来说是非常有价值的。默认实现使用(`DefaultJpaDialect`)不提供任何特殊功能,如果需要上面的功能,你必须指定适当的方言。

查看 `JpaDialect` 的 javadocs 获取更多如果使用 Spring JPA 的细节

28.6 Using the Quartz Scheduler使用Quartz任务调度

Quartz使用`Trigger`， `Job`和`JobDetail`来实现各种调度任务。Quartz的基础概念可以查看http://quartz-scheduler.org。为了方便，Spring提供了在Spring基础应用下简单使用Quartz的数个类。

28.6.1 Using the JobDetailFactoryBean

Quartz `JobDetail`对象包含了执行一个工作任务的所有必要信息。Spring提供`JobDetailFactoryBean`以XML格式配置Bean样式的属性。让我们来看一个例子：

```
<bean name="exampleJob" class="org.springframework.sheduling.quartz.JobDetailFactoryBean">
    <property name="jobClass" value="example.ExampleJob" />
    <property name="jobDataAsMap">
        <map>
            <entry key="timeout" value="5" />
        </map>
    </property>
</bean>
```

工作任务细节配置包含了执行的所有需要的信息（`ExampleJob`）。timeout是在工作任务数据以键值形式被指定。工作任务数据键值可以通过`JobExecutionContext`工作执行上下文获取（在执行期间传递），但是`JobDetail`也是通过工作实例被数据映射的属性来得到它的属性值。所以在这个例子中，如果`ExampleJob`包含了一个名称为`timeout`的bean属性，`JobDetail`将会自动获取到:

```
package example;

public class ExampleJob extends QuarztJobBean {
    private int timeout;

    /**
       * Setter called after the ExampleJob is instantiated
     * with the value from the JobDetailFactoryBean(5)
     */
    public void setTimeout(int timeout) {
        this.timeout = timeout;
    }

    protected void executeInternal(JobExecutionContext ctx) throws JobExecutionException {
        // do the actual work
    }
}
```

通过工作数据键值配置的所有附件属性也都是可用的。

> 使用`name`和`group`属性，可以分别改变工作的名称和组。工作的名称默认是匹配`JobDetailFactoryBean`的名称(在上面的例子中，就是`exampleJob`)

28.6.2 Using the JobDetailFactoryBean

经常需要调用特定对象的方法。使用`MethodInvokingJobDetailFactoryBean`可以实现这一点：

```
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="exampleBusinessObject" />
    <property name="targetMethod" value="doIt" />
</bean>
```

上面的例子将会调用 `exampleBusinessObject` 对象的 `doIt` 方法：

```
public class ExampleBusinessObject {

    // properties and collaborators

    public void doIt() {
        // do the actual work
    }
}
```

```
<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject" />
```

使用 `MethodInvokingJobDetailFactoryBean`，不需要去创建一个仅仅实现调用方法的主工作任务，只需创建真正业务逻辑对象和关注对象实现的细节。

Quartz工作任务默认是无状态的，导致的结果工作任务之间会相互干扰。如果给同一个 `JobDetail` 指定两个触发器，可能会出现在第一个任务完成之前，第二个任务就要开始了。如果 `JobDetail` 类实现了 `Stateful` 接口，这种情况就不会发生。在第一个任务完成之前第二个任务将不会开始。为了使任务通过 `MethodInvokingJobDetailFactoryBean` 不并发，设置 `concurrent` 标志为 `false`。

```
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="exampleBusinessObject" />
    <property name="targetMethod" value="doIt" />
    <property name="concurrent" value="false" />
</bean>
```

> 默认的工作任务是在并发形式下执行的

28.6.3 Wiring up jobs using triggers and the SchedulerFactoryBean

我们已经创建了工作任务和任务的实现。也有方便的bean来调用指定的对象方法。我们仍然需要这些工作任务自己进行调度。使用触发器和 `SchedulerFactoryBean` 来完成。Quartz实现了几个触发器，Spring提供两个方便的Quartz `FactoryBean` 实现： `CronTriggerFactoryBean` 和 `SimpleTriggerFactoryBean`。

触发器是需要被安排调用的。Spring提供SchedulerFactoryBean使触发器做可以设置的属性暴露出来。 `SchedulerFactoryBean` 使用触发器来安排调用工作任务。

下面是两个例子：

```
<bean id="simpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerFactoryBean">
    <!-- see the example of method invoking job above -->
    <property name="jobDetail ref="jobDetail" />
    <!-- 10 seconds -->
    <property name="startDelay" value="10000" />
    <!-- repeat every 50 seconds -->
    <property name="repeatInterval" value="50000" />
</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">
    <property name="jobDetail" ref="exampleJob" />
    <!-- run every morning at 6 AM -->
    <property name="cronExpression" value="0 0 6 * * ?" />
</bean>
```

现在我们已经设置了两个触发器，一个每隔50秒延迟10秒启动一个次和一个每天在上6点执行。最后我们需要设置 `SchedulerFactoryBean`：

```
<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="cronTrigger" />
            <ref bean="simpleTrigger" />
        </list>
    </property>
</bean>
```

`SchedulerFactoryBean` 更多的可设置属性，例如工作任务实现使用的calendars，Quartz自定义的属性等等。查看 `SchedulerFactoryBean` 的javadocs了解更多信息。