Spring FrameWork3.0 开发教程

作者:JAVA 自由城市

http://www.elearn007.com

加入 VIP 会员咨询 QQ：1229173104

目 录

第1章 Spring Framework概述

Spring Framework 是一个轻量级的解决方案，可以让你很容易的建立自己的应用程序。
然而 Spring 是模块化的，允许你只使用你需要的部分。你可以使用 IoC 容器，而让 Struts 负责顶层的应用，但是你也可以只用整合 Hibernate 的代码或是 JDBC 的抽象层。
Spring 框架支持声明式的事务管理，远程访问你的应用逻辑，可以通过 RMI 和 web Services.以及持久化你应用数据的时候也会有多种选择。Spring 也提供了全功能的 MVC 框架，并且允许你整合 AOP 到你的应用系统。

Spring 被设计成架构无关的，意味着你可以管理你的代码，而不会与框架有强依赖关系。在整合层（例如数据访问层），有些依赖，并且需要 Spring 类库文件，但是这种依赖很容易从你的代码中消除。

该文档是使用Spring Framework框架的参考书，如果你有任何的要求，建议或是问题，可以把它们发布到用户列表，或是在http://forum.springsource.org上进行发表。

1-1节 介绍 Spring Framework

Spring Framework 是一个 Java 平台，该平台提供了系统全面的架构来支持开发 Java 应用程序。Spring 处理架构的事情，而你可以集中在业务应用实现上。

Spring 能使你从”plain old Java Object“（POJOs）来建立应用，应用企业级服务到 POJOs。 该特性可以应用到 Java SE 的开发模式中，也可以是全面的 Java EE 的企业级开发。

使用 Spring 平台的优点，可以简单的列几个：
- 可以让你的方法以数据库事务的方式来执行，并且不需要处理事务的 APIs.
- 让本地方法执行一个远程处理过程，而不需要处理远程的 APIs.
- 让本地方法执行一个流程管理的操作，而不需要处理 JMX APIs.
- 让本地方法作为一个消息处理器而不需要处理 JMX APIs.

1-1-1节 依赖注入与控制反转

背景知识：

关于是什么是 aspect of control 的反转，Martin Fowler 在 2004 年在他的网站上给出了解答，后来 Fowler 建议把 IoC 重新命名为 Dependency Injection.因为这样做，更容易理解。
更多关于 DI 的资料请访问如下的网站：
http://martinfowler.com/articles/injection.html

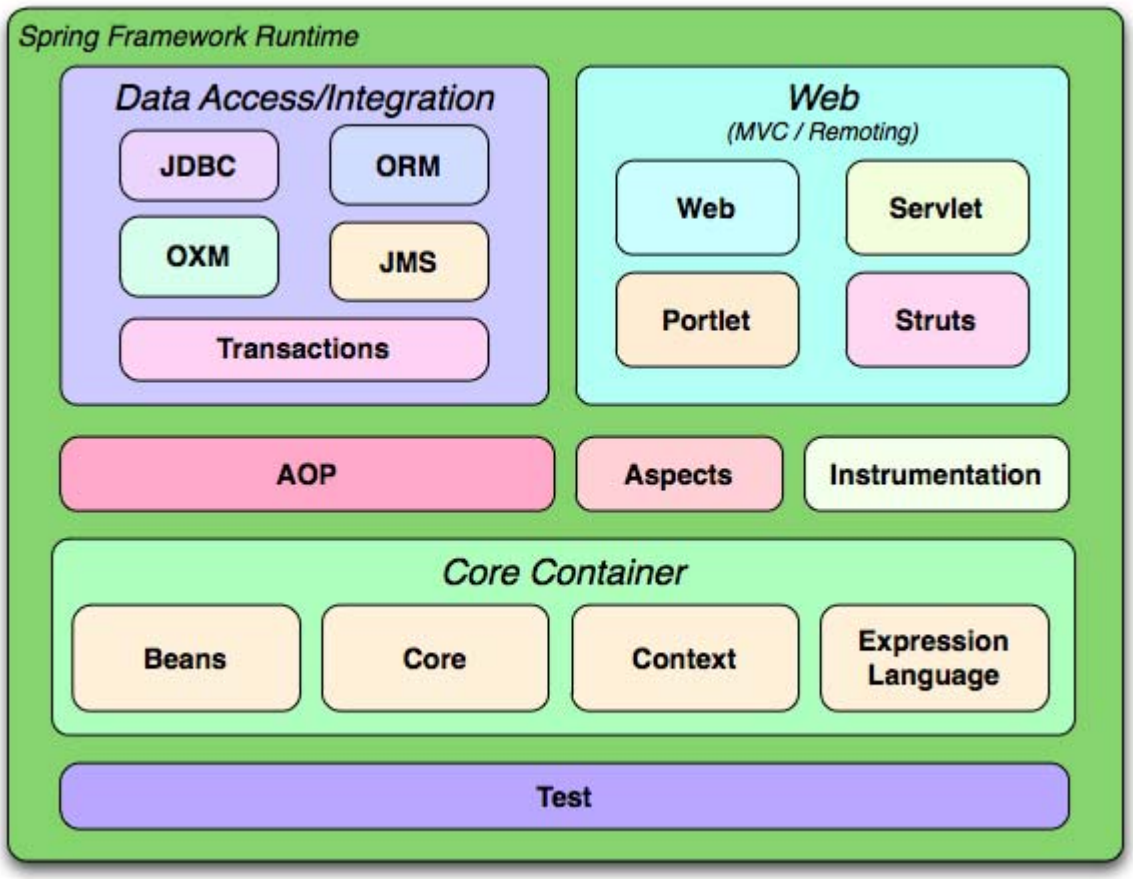
Java 应用程序—松耦合的概念从 applets 小程序到具备 N 层应用的服务器端企业级应用程序，都是存在的。---典型的是对象间的合作交互形成了一个业务应用。因此在应用系统中的对象之间具备依赖的关系。

尽管 Java 平台提供了丰富的应用开发功能，但是它缺少方法来组织基础的代码块到一个合乎逻辑的统一整体中，把这个任务留给了架构师和开发人员。事实上你可以使用诸如 Factory,Abstract Factory,Builder,Decorator,和 Service Locator 来组织不同的类同时让类实例组成一个实例。但是这些模式太简单了：靠一个名字来进行应用，使用模式的描述来达到模式的应用目的，在什么地方应用它们，它能解决的问题，等等。设计模式是比较好的组建应用，你可以在你的应用中进行自主应用。

Spring 框架的控制反转组件是这样来考虑这个问题的：通过提供一个格式良好的方法来组装不同的组件到一个完整的应用程序中。Spring Framework 提供了良好的设计模式类，你可以把它们整合到你自己的应用中。很多的组织和研究机构使用 Spring Framework 在他们的应用中，这样做可以形成良好的系统架构，并且足够灵活，方便后期的应用维护。

1-1-2节 模块组成

Spring Framework 有 20 多个模块组成，这些模块被组织到了一个核心容器中，数据访问整合，WEB,AOP,框架和测试，如下图表所展示的那样：



核心容器 Core Container

Core Container 由：Core，Beans,Context 和 Expression Language 模块组成。
Core 和 beans 模块提供了框架的基础功能部分，包括 Ioc 和 Dependency Injection 的特性。BeanFactory 是一个工厂模式的应用，它消除了程序化的 singletons 并且允许你封装配置和从实际程序中确定依赖关系。

Context 模型是建立在 Core and Beans 模型上：通过它可以访问被框架管理的对象，这类似于 JNDI 注册。Context 模块从 beans 模块中集成了不少的特性，并且添加了对国家化的支持（例如对资源打包的时候）。事件循环，资源装载，和 contexts 的生成及转换，例如一个 servlet 容器。Context 模块同样支持 Java EE 的特性，例如 EJB,JMX 和基础的远程访问。ApplicationContext 接口是 Context 模块的本地点。

表达式语言提供了一个强大的表达式语言来查询和处理一个对象，在运行的时候。它是统一表达式(unified expression language)的扩展，该规范分属 JSP 2.1 规范。该语言支持设置和访问属性数值。方法的调用，访问上下文数组，集合数组，索引数组，命名变量，通过名字获取对象从 Spring IoC 容器中。

数据访问与整合

数据访问与整合层包括：JDBC,ORM,OXM,JMS,和事务模块。

JDBC 模块提供了一个 JDBC 的抽象层，消除了对 JDBC 个性编码的需求。而且统一了数据库访问的错误代码。

ORM 模块提供了处理对象关系映射的 APIs 来进行整合，败落 JPA,JDO,Hibernate 和 iBatis. 使用 ORM 包，你可以使用这些 O/R-mapping 框架，同时也可以使用 Spring 的其他功能。例如相对简单的声明性事务管理功能。

而 OXM 模块提供了 Object/XML 映射的抽象层，为 JAXB,Castor,XMLBeans,JiBX 和 XStream 等。

JMS 模块包含了生产和消费消息的功能。

事务模块提供了程序化和声明性的事务管理。这个功能可以为所有类和 POJOs 所用。

WEB

Web 层有 Web,Web-Servlet,Web-Struts 和 Web-Portlet 模块组成。
Spring 的 Web 模块提供了基础的面向 web 的整合特性，例如多部分文件上传功能，使用 servlet 监听来初始化 IoC 容器，面向 WEB 的应用程序上下文环境。它同样包含 Spring 的远程访问相关的 Web 部分的内容。

Web-Servlet 模块包含了 Spring 的 MVC 应用。Spring 的 MVC 框架提供了一个条理的在代码模型和 WEB 表单及其他 Spring Framework 整合方面的分离。

Web-Struts 模块包含了整合传统 struts web 层的类，在一个 Spring 应用中。但是值得注意的是该部分已经在 Spring3.0 中已经不推荐使用了，可以考虑整合你的应用到 Struts2.0.或是整合到一个 Spring MVC 解决方案。或是跟 Spring 进行整合。

Web-Portlet 模块提供了一个 MVC 的应用可以使用在一个 portlet 环境下，并且可以映射实现一个 Web-Servlet 模块的功能。

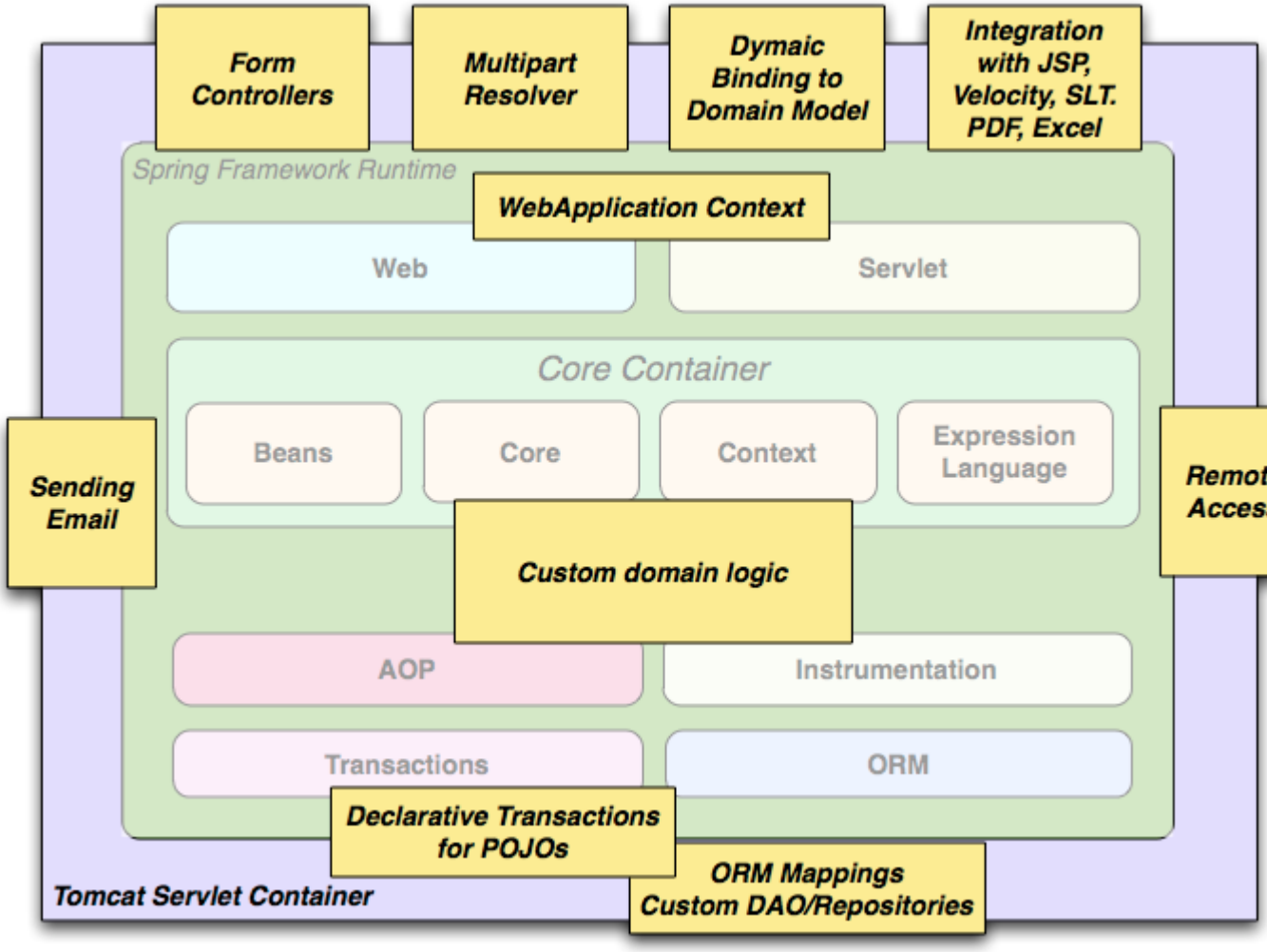
AOP 和架构

TEST
Test 模块支持测试 Spring 组件，通过使用 JUnit 和 TestNG.提供了同步装载 Spring ApplicationContexts 和缓冲这些上下文环境。它也提供了 Mock 对象，这些对象你可以用来独立测试你的代码。

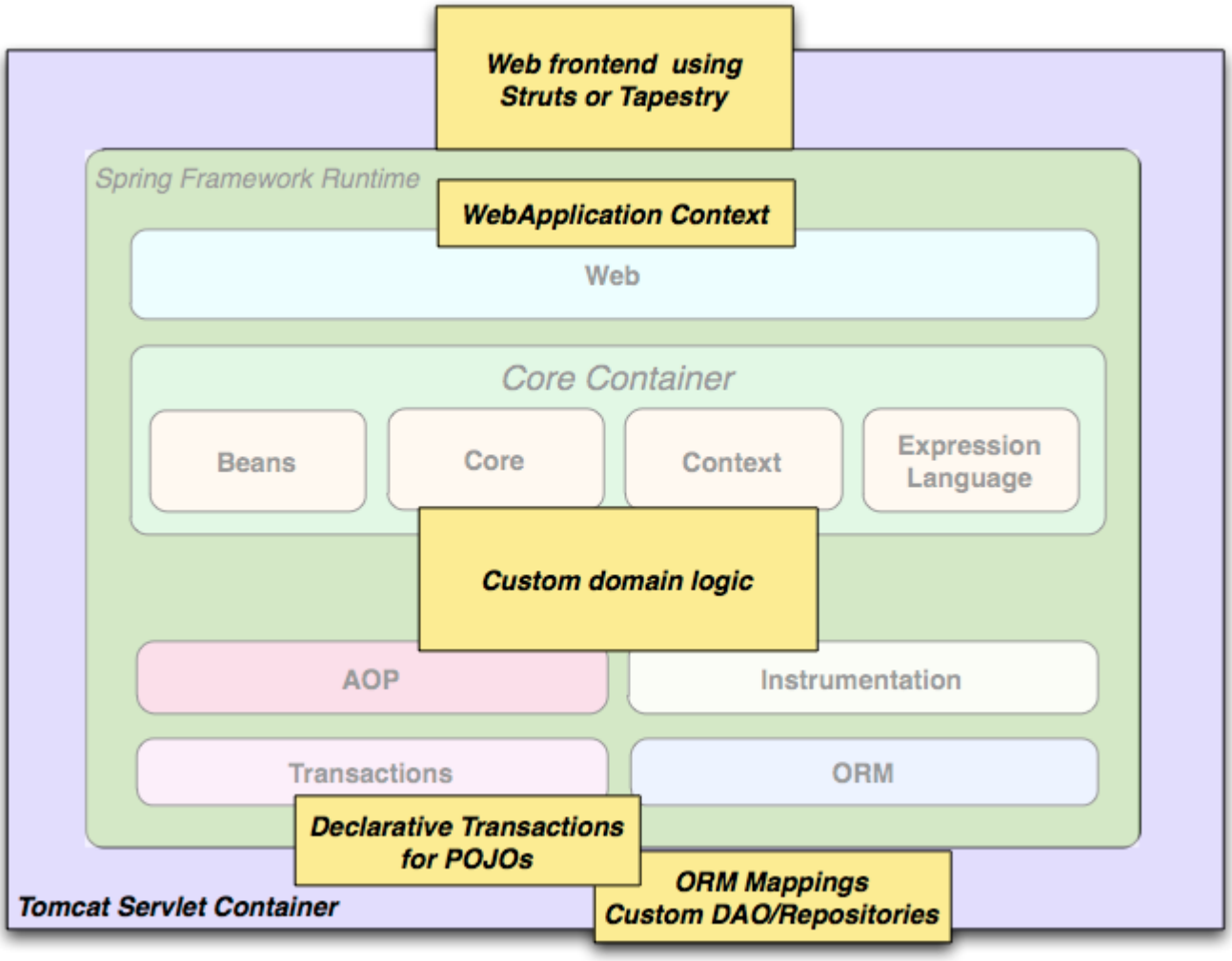
1-1-3节 Spring 的应用场景

上面描述的组件模块使 Spring 称为一个逻辑性的选择在许多场景下，从 applets 到一个全面

的企业级应用程序，该应用程序使用了 Spring 的事务管理功能和 Web 框架的集成。

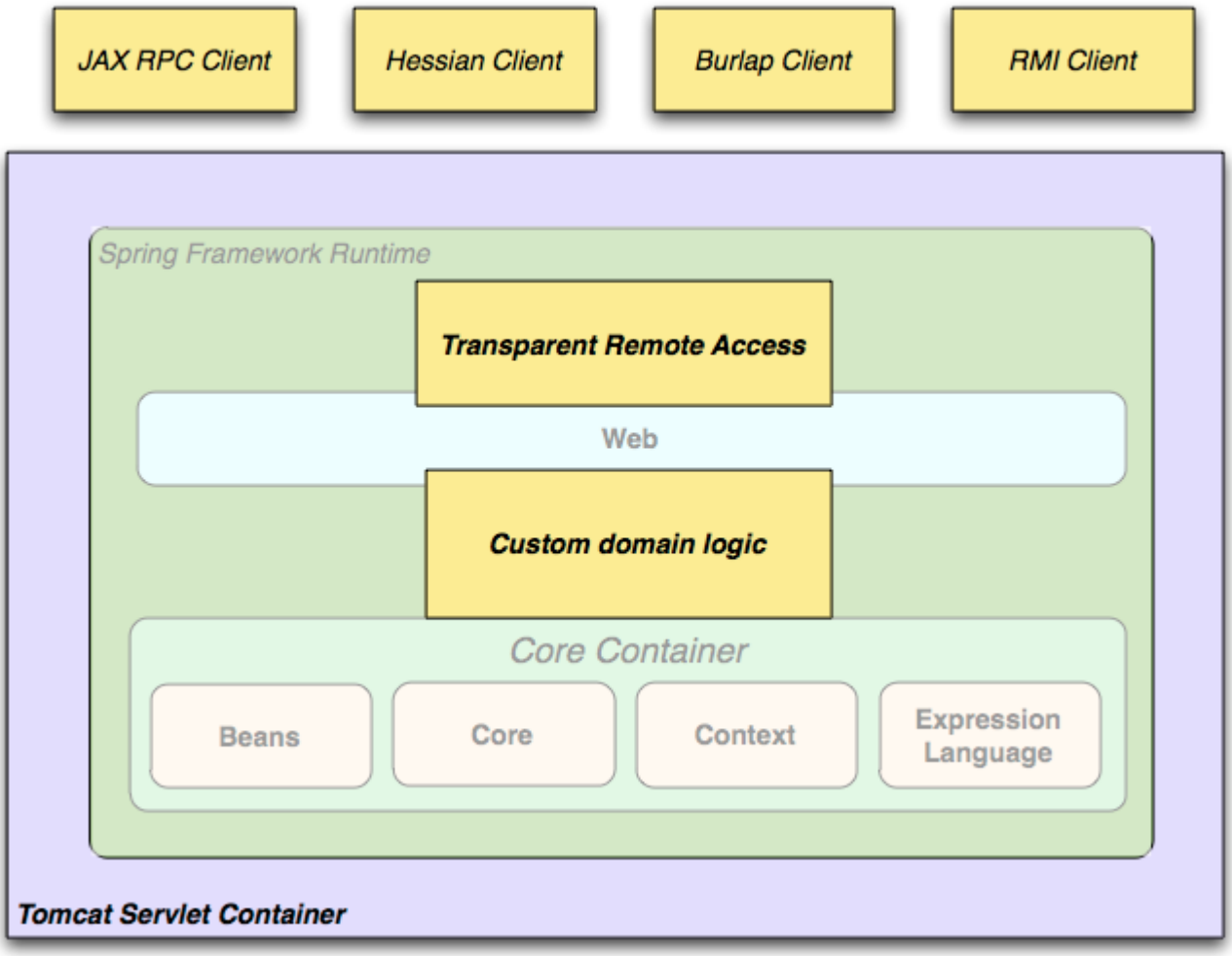


Spring 的声明性事务管理特点让 web 应用程序具备完全的事务性，类似它本来就是支持事务的特征。如果你使用了 EJB 容器管理的事务。你所有的自定义业务逻辑都可以通过 POJOs 和可管理的 Spring IoC 容器，带达到这个目的。其他的服务包括支持邮件发送，验证关联的 WEB 层。Spring 的 ORM 支持与 JPA,Hibernate,JDO 和 iBatis 进行整合。例如你使用了 Hiberante，那么你可以继续使用存在的映射文件和标准的 Hibernate SessionFactory 配置。而 Controllers 可以无缝的整合 WEB 层和主应用模型，去除了对 ActionForms 及其他类似类的需要，因为这些类的作用仅仅是传送 HTTP 参数到你的主应用模型。



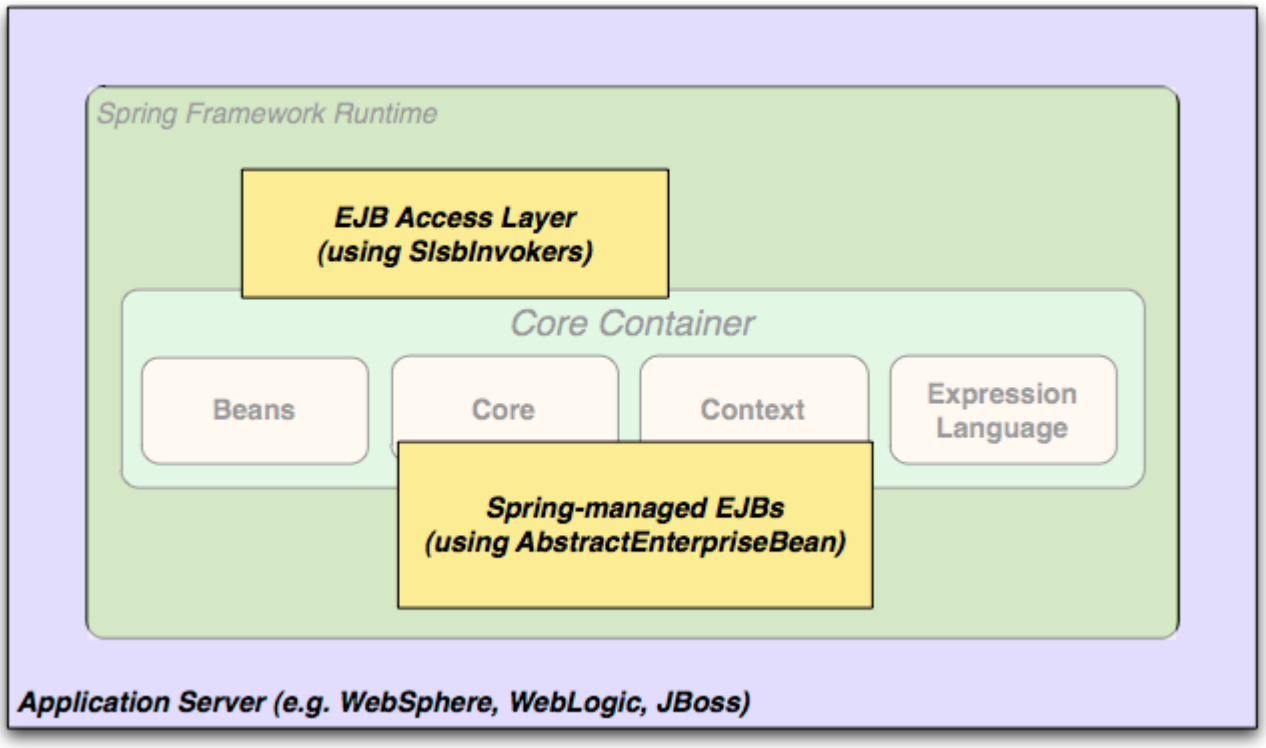
Spring middle-tier using a third-party web framework

在一些情况下不允许你完全的移植程序到不同的框架。Spring Framework 不强制你使用它内部的所有模块。它不是要么用，要么一点别用的，二选一解决方案，是足够灵活的解决方案。已经存在的前端可以使用 WebWork,Struts,Tapestry,或其他的 UI 框架，这些框架可以与 Spring 进行整合，这时候的 Spring 类似中间层。允许你使用其的事务特性。你只需要简单的使用 ApplicationContext 和 WebApplicationContext 来整合你的应用逻辑代码即可。



Remoting usage scenario 远程使用的样例

当你需要通过 web services 来访问存在的代码，你可以使用 Spring 的 Hession-.Burlap-,Rmi-或是 JaxRpcProxyFactory 类。让我们远程访问存在的业务应用系统不再困难。



EJBs - Wrapping existing POJOs 封装已经存在的 POJOs

Spring Framework 也提供一个访问层和抽象层针对 EJB，可以使你能复用已经存在的 POJOs 并且把它们封装到一个状态无关的会话 bean 中，这样更灵活，适合需要声明式部署安全性的 fail-safe 的 WEB 应用程序。

Dependency Management and Naming Conventions

依赖管理和依赖注入是不同的事情。为了更好的使用 Spring 你需要把所有需要的类库文件放到类路径上。以让运行和编译的时候能够找到。这些依赖不是虚拟的组件，仅仅是物理资源，通常情况下是文件系统。依赖管理的过程包括：找到这些资源，保存，并且把它们添加到类路径。依赖可以是 direct,也可以是间接的（例如我的应用程序是依赖 commons-dbcp 而该 commons-dbcp 又依赖 commons-pool）间接依赖是著名的 transitive 并且这样的依赖非常难发现和管理。

如果你计划使用 Spring 你需要得到组成 Spring 的所有 jar 文件包。为了做起来更容易，Spring 打包是按照模块的方式，分组打包的。所以如果你不想写 WEB 应用程序，那么你就不需要 spring-web 模块。

通常情况下，Spring 发布它的 artifacts 到四个不同的地方：

- 在交流社区的下载站点

http://www.springsource.org/downloads/community.

在这里你可以找到所有的 spring jar 包文件，通常是以 zip 文件格式存在的。

- Maven Central.

  这是 Maven 默认的组件库，Maven 会查询，不需要任何的特别设置。需要常用的库文件，都可以在这里找到。在这里 spring 的 groupid 是 org.springframework.

- The Enterprise Bundle Repository（EBR）这是被 SpringSource 管理维护的，包括所有与 Spring 整合需要的库文件。Maven 和 Ivy 的组件库在这里都可用。
- 一个公开的 Maven 库在 Amazon S3 上，是开发的快照，和各版本。从这里可以得到 spring 的不同的开发版本，来与其他的包含在 Maven Central 里的库文件公用。

所以这里的第一件事情就是你决定如何管理你的依赖：很多人使用自动化工作，例如 Maven 或是 Ivy,但是你也可以手工来完成，只需要下载所有需要的 JAR 包文件即可。

记住这里的存放位置需要单独，不能混合，这点特别重要，因为 EBR 的 artifacts 需要使用不同的命名规范，跟 Maven Central 的 artifacts 是不同的。

下表是对 EBR 和 Maven 的比较：

Feature	Maven Central	EBR
OSGi Compatible	Not explicit	Yes
Number of Artifacts	Tens of thousands; all kinds	Hundreds; those that Spring integrates with
Consistent Naming Conventions	No	Yes
Naming Convention: GroupId	Varies. Newer artifacts often use domain name, e.g. org.slf4j. Older ones often just use the artifact name, e.g. log4j.	Domain name of origin or main package root, e.g. org.springframework
Naming Convention: ArtifactId	Varies. Generally the project or module name, using a hyphen "-" separator, e.g.	Bundle Symbolic Name, derived from the main package root, e.g. org.springframework.beans. If the jar had to be patched to ensure OSGi compliance then com.springsource is appended, e.g.

Feature	Maven Central	EBR
	spring-core, logj4.	com.springsource.org.apache.log4j
Naming Convention: Version	Varies. Many new artifacts use m.m.m or m.m.m.X (with m=digit, X=text). Older ones use m.m. Some neither. Ordering is defined but not often relied on, so not strictly reliable.	OSGi version number m.m.m.X, e.g. 3.0.0.RC3. The text qualifier imposes alphabetic ordering on versions with the same numeric values.
Publishing	Usually automatic via rsync or source control updates. Project authors can upload individual jars to JIRA.	Manual (JIRA processed by SpringSource)
Quality Assurance	By policy. Accuracy is responsibility of authors.	Extensive for OSGi manifest, Maven POM and Ivy metadata. QA performed by Spring team.
Hosting	Contegix. Funded by Sonatype with several mirrors.	S3 funded by SpringSource.
Search Utilities	Various	http://www.springsource.com/repository
Integration with SpringSource Tools	Integration through STS with Maven dependency management	Extensive integration through STS with Maven, Roo, CloudFoundry

Spring Dependencies and Depending on Spring

尽管 Spring 提供了集成，并且支持很大方位的企业级及其他的扩展工具，它还是倾向保持它自身的独立性。你不需要查找和下载很大数量的 JAR 包为了使用 Spring 做个演示。为了方便基础的依赖注入，只有一个外部的依赖，并且只是关于日志的，有关日志的选项我们后面会仔细探讨。

下面我们整理出一个大纲来讲解如何配置应用，以支持 Spring..首先讲解 Maven 然后是 Ivy.在所有的例子中，如果一些内容没理解，你需要参考本文档的依赖管理系统部分，或是查看些样例代码-Spring 使用了 Ivy 来管理依赖，当其组建的时候，我们的例子更多的使用 Maven.

Maven Dependency Management

Maven 的依赖管理

如果你正在使用 Maven 来管理，你甚至不需要显示的支持日志依赖，例如，为了生成一个应用上下文环境，并且使用依赖注入，你的 Maven 依赖配置看上去将是这样的：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.0.0.RELEASE</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

就这么简单，注意这里的 scope 可以被声明为 runtime 如果你不需要依赖 Spring APIs 进行编译，这是典型的应用场景。

我们使用 Maven Central 命名格式在上面的例子中，所以与 Maven Central 或是 SpringSource S3 Maven Respository。为了使用 S3 Maven respository（例如为了下载细版本或是开发快照），你需要确定 repository 的位置在你的 Maven 配置中，例如这里的：

```
<repositories>
    <repository>
        <id>com.springsource.repository.maven.release</id>
        <url>http://maven.springframework.org/release/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
```

```
</repositories>
```

For milestones:

```
<repositories>
   <repository>
      <id>com.springsource.repository.maven.milestone</id>
      <url>http://maven.springframework.org/milestone/</url>
      <snapshots><enabled>false</enabled></snapshots>
   </repository>
</repositories>
```

And for snapshots:

```
<repositories>
   <repository>
      <id>com.springsource.repository.maven.snapshot</id>
      <url>http://maven.springframework.org/snapshot/</url>
      <snapshots><enabled>true</enabled></snapshots>
   </repository>
</repositories>
```

为了使用 SpringSource EBR 你需要使用不同的名字格式来管理依赖关系，这里的名字通常很容易猜到，例如这里的例子：

```
<dependencies>

    <dependency>

        <groupId>org.springframework</groupId>

        <artifactId>org.springframework.context</artifactId>

        <version>3.0.0.RELEASE</version>

        <scope>runtime</scope>

    </dependency>

</dependencies>
```

你也需要声明 repository 的位置：

```
<repositories>
    <repository>
        <id>com.springsource.repository.bundles.release</id>
        <url>http://repository.springsource.com/maven/bundles/release/</url>
    </repository>
</repositories>
```

如果你手动管理依赖，那么上面声明的 URL 是不可浏览的，但是你可以访问下面的地址

http://www.springsource.com/repository

Logging

日志是一个非常重要的依赖：

a) 它是唯一的强制性的外部依赖。
b) 每个人都乐意看到他们正在使用工具的一些外部输出。
c) Spring 整合了很多其他的工具，这些工具可以决定以怎么样的方式来输出日志。

开发人员的一个全局性的任务就是必须具有一个全局的日志配置，来集中存放应用程序的日志信息，这其中包括所有的外部组件。其实这是非常困难的没，因为有很多的可选择的日志框架。

强制性的日志依赖在 spring 中是 Jakarta Commons Logging API（JCL）。我们可以依据 JCL 编译，我们也可以把 JCL 的 Log 对象让扩展 Spring 框架的所有类可见。让所有版本的 Spring 使用相同的日志库，这是非常重要的：这样让整合更加容易。

我们通过让 Spring 的其中一个模块显示的依赖 commons-logging（实现了 JCL）然后让其他的模块都依赖其中的这个模块，在编译的时候。如果你正在使用 Maven，那么你在想从那里设置这种对 commons-logging 的依赖，实际上这个设置来自 Spring，来自核心模块 spring-core.

关于 commons-logging 比较好的地方是你不需要做任何事情，来让你的应用工作。它具有个运行时发现算法，该算法可以在类路径里查找其他的日志框架，并且自动选择合适的框架。如果没有可用的，你得到的日志仅仅来自 JDK(java.util.logging 或是 JUL)你会发现你的 Spring 应用工作，并且日志被打印到控制台，这是很重要的。

Not Using Commons Logging

不幸运的是，运行时发现算法在 commons-logging,对于方便终端使用者来说是个问题。如果我们会到开头，把 Spring 当作一个新的项目，那么它会使用一个不同的日志依赖。第一个选择也许就是 The Simple Logging Facade for Java(SLF4J),slf4j 也被许多其他的工具来使用。
关闭 commons-logging 比较容易：确信它不在类路径下在运行的时候，在 Maven 的概念中，你可以剔除该依赖，因为在 Spring 里依赖是声明式的，你可以这样做：

```
<dependencies>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>3.0.0.RELEASE</version>
<scope>runtime</scope>
<exclusions>
<exclusion>
<groupId>commons-logging</groupId>
<artifactId>commons-logging</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
```

现在这里的应用程序将会中断，因为没有实现 JCL API 的类，所以如果解决这个问题，就必须提供新的实现类，下面的部分演示如何解决该问题。

Using SLF4J 使用 SLF4J

SLF4J 更清晰的依赖，并且比 commons-logging 更有效率，因为它使用编译时间绑定的方式，来替代在运行时查找框架的方法。这意味着你必须做出更明确的日志设置。并且声明它或做正确的配置。SLF4J 提供了绑定很多 common logging 框架的途径，你通常选择个你已经用的框架，并且把该框架做好配置和管理。

如果使用 SLF4J 在 Spring 中，你需要替换 commons-logging 依赖，用 SLF4J-JCL 桥。你旦你做了，那么从 Spring 内部的日志调用将被转发为对 SLF4J API 的日志调用。所以如果其他的库文件在你的应用中，你可以有个单独的地方来管理和配置日志。

一个更普遍的选择是把 Spring 和 SLF4J 关联，然后提供显示绑定，把 SLF4J 到 Log4J.
你需要提供 4 个依赖（并且剔除已经存在的 commons-logging）:桥，SLF4J API，到 Log4J 的绑定和 Log4j 应用自身。在 Maven 中你会类似这样做：

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

```
<version>3.0.0.RELEASE</version>
<scope>runtime</scope>
<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>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>slf4j-api</artifactId>
<version>1.5.8</version>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>org.slf4j</groupId>
<artifactId>slf4j-log4j12</artifactId>
<version>1.5.8</version>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.14</version>
<scope>runtime</scope>
</dependency>
</dependencies>
```

这看上去跟日志管理的依赖很多，是的的确是，但是这是可选择的。而且这样做的效果更好。如果你在使用一个限制性的容器，例如 OSGi 平台。理论上这样做，会得到性能的提升。

另一个更普及的选择是针对那些 SLF4J 的用户，可以使用更少的步骤和更少的依赖，可以通过直接跟 Logback 进行绑定。因为 Logback 实现了 SLF4J,所以你仅仅依赖两个库，而不是 4 个库(jcl-over-slf4j 和 logback).如果你那么做，你也需要剔除 slf4j-api 的依赖，从其他的外部依赖中(不是 Spring),因为你只需要 API 的一个版本在类路径中。

Using Log4j

许多人使用 Log4j 当作日志框架。他效率高，而且方便部署，而且在事实上当我们建立和测试 Spring 的时候也是在用它。Spring 也提供了一些工具来配置和使用 Log4j,所以这里有个可选择的编译时依赖在一些模块中。

让 Log4j 与默认的 JCL 依赖工作，你需要做的是把 Log4j 放到类路径中，为它提供一个配置文件(log4j.properties 或 log4j.xml 在跟类路径中)在 Maven 中你做如下的声明：

```
<dependencies>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>3.0.0.RELEASE</version>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.14</version>
<scope>runtime</scope>
</dependency>
```

而对应的配置文件如下：

```
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
```

带本地 JCL 的运行时容器

许多人运行 Spring 程序在那些实现了 JCL 的容器中，例如 WAS。这样常会出现问题，而且仍然没有好的解决办法，仅仅是移除 commons-logging 是不够的。

第2章 Spring Framework新特性与改进

如果你使用 Spring 框架一些时间了，你会知道 Spring 已经发布了 2 个大的版本：Spring2.0（2006 年发布）和 Spring2.5（2007 年发布），现在发布了 Spring3.0

目前的 Spring 框架基于 Java 5 并且全部支持 Java 6.而且 Spring 支持 J2EE1.4 和 J2EE5.

2-1节 Java 5

Spring 整个框架被修正成完全支持 Java 5 的新特性，例如:generics,varargs,及其他的语言提升方面。而且 Spring 尽力做到代码的前后兼容，现在可以使用 generic Collections 和 Maps,同时可以使用 generic FactoryBeans。同时提供了桥方法在 Spring AOP API.Generic ApplicationListeners 自动接收指定的事件类型。所有的回调接口例如 TransactionCallback 和 HibernateCallback 声明了一个 generic 结果数现在。同时 Spring 代码现在做了更新，为 Java 5 做了一些优化。

Spring 的 TaskExecutor abstraction 做了更新，为了与 Java 5 的 java.util.concurrent 工具进行整合。现在的框架更兼容 JSR-236.

2-2节 经过完善的文档

Spring 参考文档也做了全面的修订，以反映 Spring 3.0 的各新特性。

2-3节 新的文章和样例

有很多优秀的文档和指导例子演示了如何使用 Spring3 的特性，可以从 Spring Documentation 页面找到。
一些修订后的例子也从源代码中抽离到一个独立的 SVN 库地址如下：
https://anonsvn.springframework.org/svn/spring-samples/
这些例子需要单独下载才可以。

2-4节 新的模块组织与项目组件

框架模块进行修订，现在模块进行独立管理：
• org.springframework.aop
• org.springframework.beans

• org.springframework.context
• org.springframework.context.support
• org.springframework.expression
• org.springframework.instrument
• org.springframework.jdbc
• org.springframework.jms
• org.springframework.orm
• org.springframework.oxm
• org.springframework.test
• org.springframework.transaction
• org.springframework.web
• org.springframework.web.portlet
• org.springframework.web.servlet
• org.springframework.web.struts

Spring.jar 包括了所有的框架内容，不再提供！

我们现在使用了一个新的 Spring 的组建系统，Spring Web Flow2.0 它给我们了：
• Ivy-based "Spring Build" system
• consistent deployment procedure
• consistent dependency management
• consistent generation of OSGi manifests

2-5节 新特性

下面的列表，显示了 Spring3.0 的新特性，我们接下来会仔细探讨这些特性：
• Spring Expression Language
• IoC enhancements/Java based bean metadata
• General-purpose type conversion system and field formatting system
• Object to XML mapping functionality (OXM) moved from Spring Web Services project
• Comprehensive REST support
• @MVC additions
• Declarative model validation
• Early support for Java EE 6
• Embedded database support

2-5-1节 为 Java 5 更新的 API

BeanFactory 接口返回的 bean 的实例类型如下：
• T getBean(Class<T> requiredType)

• T getBean(String name, Class<T> requiredType)
• Map<String, T> getBeansOfType(Class<T> type)

Spring 的 TaskExecutor 接口现在继承自：
Java.util.concurrent.Executor:

扩展 AsyncTaskExecutor 支持标准的定期调用

2-5-2节 Spring 表达式

2-5-3节 控制反转容器

基于Java bean的元数据

在组件内定义Java bean的元数据

2-5-4节 通用的类型转换和数据格式化系统

2-5-5节 数据层

2-5-6节 Web 层

2-5-7节 声明校验模型

2-5-8节 对 J2EE 6 的支持

2-5-9节 对嵌入式数据库的支持

第3章 核心技术

本章介绍的参考资料涵盖了 Spring Framework 的核心技术，这些技术组成了完整了 Spring Framework 框架。

首先在这些技术中包括 Spring Framework 的 IoC 容器。这个 IoC 容器又紧紧被 AOP 的编程技术所围绕。Spring Framework 具有自己的 AOP 框架，该框架比较容易理解，而且可以满足 AOP 开发的 80%的需要。

同时涵盖了 Spring 使用 AspectJ 进行整合的技术，也被提供了。

最后 TDD 开发方法被 Spring 团队所提倡。

3-1节 IoC 容器

3-1-1节 关于 spring IoC 容器与组件

该章涵盖了 Spring Framework 实现 IoC 的基础理论。IoC 同时也被称为 dependency injection（DI）。这是一个定义对象依赖关系的方法。另外的对象只依靠构造行数参数，或是工厂方法参数，或是属性来跟具有依赖关系的对象合作。容器生成一个 bean 就会注入这些依赖。这个过程类似反转，所以这个过程的名字也叫做 Inversion of control（IoC）.bean 通过直接调用类的构造函数来控制实例化和其的相关依赖。类似 Service Locator 模式。

Org.springframeowork.beans 和 org.springframework.context 包是 IoC 容器的基础。BeanFactory 接口提供了一个高级的配置机制来管理任意类的对象。
ApplicationContext 是 BeanFactory 的子接口。它添加了一些方便与 Spring AOP 进行整合的特性；消息资源处理，事件发布和特别的应用层环境，例如 WebApplicationContext 该上下文环境是在 WEB 应用程序中使用的。

简单说，BeanFactory 提供了配置的框架和基础的功能。并且 ApplicationContext 添加了支持企业级应用的功能。ApplicationContext 是一个完整的 BeanFactory 的超级类。专门用来描述 IoC 容器。更多的信息可以参考 3.14 节。

在 Spring 中对象组成了你的整个应用程序，并且被 Spring IoC 容器管理着，这些对象被我们称为 beans.一个 bean 就是一个对象，这个对象可以被实例化，被装配，或者被容器管理。另一方面 bean 是一个简单的对象。它们之间的依赖关系在配置元数据中被反映。

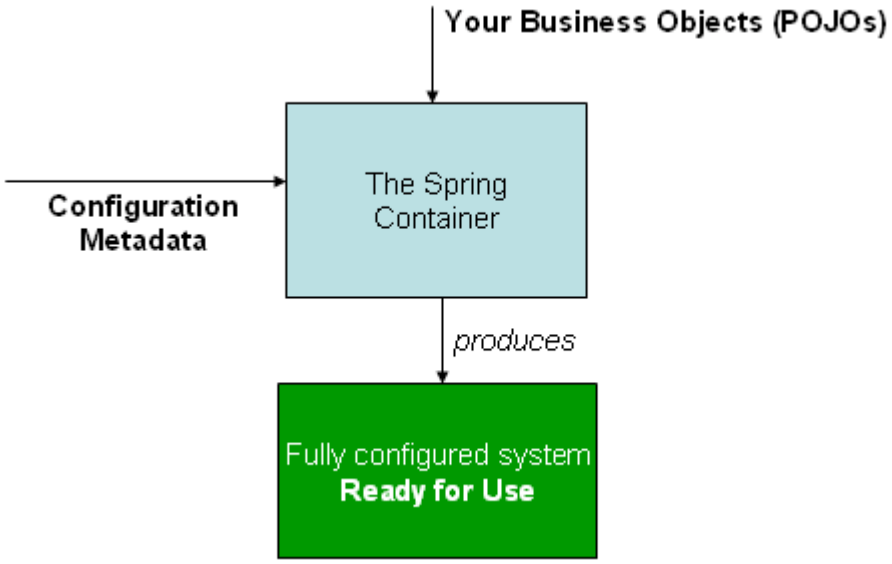
3-1-2节 容器概述

接口 org.springframework.context.ApplicationContext 就代表了 Spring 的 IoC 容器。对实例化，配置，装配 beans 负责。容器得到整个的架构，这个架构主要说明什么对象需要被实例化，配置和装配，而这个过程是通过阅读配置元数据来完成的。
配置元数据展示为 XML 格式，JAVA 注释，或是 JAVA 代码等。它允许你描述一个组成你应用的对象和对象之间的富依赖关系。

有几个被 Spring 框架自动提供的 ApplicationContext 接口实现类。在独立的应用中，它通常生成一个 ClassPathXmlApplicationContext 或是 FileSystemXmlApplicationContext.
XML 已经称为定义配置元数据的标准格式，你可以指定你的容器是使用那种配置策略，例如是通过 JAVA 注释或是代码等。

在大多数的场景下，外在的用户代码不需要实例化一个或多个 Spring IoC 容器实例。例如，在 WEB 应用程序场景下，一个简单的 8 行左右的 J2EE WEB XML 描述文件已经足够了。如果你在使用 SpringSource Tool Suite 或是 Spring Rool 这里的配置内容可以非常简单的靠鼠标的单击即可完成。

下面的图标描述了 Spring 工作的高级视图。你应用程序的类和配置元数据关联，以让 ApplicationContext 生成后被初始化，你有一个完整的配置过的可执行系统或是应用程序。



The Spring IoC container

Configuration metadata 配置元数据

正像前面图表所表示，Spring 的 IoC 容器会使用到配置元数据的一个表格；该配置文件显示了 Spring 容器如何实例化，配置，和装配在你业务应用程序中的那些类。

配置元数据传统上是 XML 文档格式，该格式在本章的大多数地方被采用来说明 Spring 容器的关键的概念和特性。

注意：

XML 的元数据不仅用在配置元数据上。Spring 的 IoC 容器自身也是从这个格式来进行解包，这个格式的配置文件用在多数场合。

更多其他类型的容器元数据，可以看下面的内容：

- Annotation-based configuration:Spring 2.5 介绍了支持的基于注释的配置元数据。
- Java-based configuration: 开始于 Spring 3.0 许多特性都被 Spring JavaConfig project 所提供。并且称为了 Spring Framework 的核心部分。因此你可以使用 Java 来在外部定义 beans.

  请参考：

@Configuration, @Bean, @Import and @DependsOn

Spring 配置的组成必须包括一个或多个 bean 的定义，该 bean 必须被容器管理。在 XML 格式下通过使用<bean/>元素，而该元素的父元素为<beans/>

这些 bean 定义可以代表组成你业务应用程序的实际类。典型情况下，你定义服务层的对象，数据访问对象，展示层对象例如 struts 的 Action 实例。构建对象例如 Hibernate SessionFactories,JMS Queues,等等。同茶馆内情况下我们无法定义个完美的域对象在容器里，因为它通常为 DAOs 和上落逻辑的生成及装载域对象负责。然而你可以使用 AspectJ 来配置对象，这些对象通常是在容器外生成的。

下面的片段演示了一个基于 XML 的基础的配置文件：

```
<?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-3.0.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>
```

这里的 id 属性是个字符串标识单个 bean 定义，而 class 属性定义了 bean 的类型，而且需要使用全限定类名。Id 的数值被当作对协作对象的参考。XML 所指的协作对象在这个例子中没有被展现。

Instantiating a container 实例化容器

实例化一个 Spring IoC 容器是非常简单的。路径地址或是路径列表传送到一个 ApplicationContext 构造函数。实际的资源字符串允许容器从各种外部资源装载配置元数据例如从 JAVA 类路径装载等：

```
ApplicationContext context =

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

当你学了 Spring 的 IoC 容器后，你一定想知道些 Spring 资源抽象的情况。资源提供了一种简便的机制来从特定位置读一个输入流，例如一个 URI 的格式的资源地址上。通常情况下资源地址被用来构造应用程序的上下文环境。

下面的例子演示了服务层的对象(services.xml)的配置文件：

```
<?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-3.0.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 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-3.0.xsd">

<bean id="accountDao"

class="org.springframework.samples.jpetstore.dao.ibatis.SqlMapAccountDao"
>

<!-- additional collaborators and configuration for this bean go here -->

</bean>

<bean                                                    id="itemDao"
class="org.springframework.samples.jpetstore.dao.ibatis.SqlMapItemDao">

<!-- additional collaborators and configuration for this bean go here -->

</bean>

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

</beans>
```

前面的例子，服务层有类 PetStoreServiceImpl 和两个数据访问类对象 SqlMapAccountDal 及 SqlMapItemDao，这两个对象都是基于 iBatis 对象与关系映射框架。Propertiy 属性名代表 JavaBean 的属性，而 ref 元素则代表另外的 bean 名字，这个 id 和 ref 两者之间的联系，就代表了相互学做的对象。

Composing XML-based configuration metadata

组合的基于 XML 的配置元数据

对于整合多个 XML 文件这个非常有用，通常每个单独的 XML 配置文件表示一个逻辑层或是架构中的某个模块。

你可以使用 application context 构造函数来从这些所有的 XML 文件中装载所有 bean 的定义。那么构造函数就需要具备多个 resource 路径。这样做有不方便的地方，作为替代我们可以使用<import/>元素来从其他文件中装载 bean 的定义，**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>
```

在上面的例子中，外部的 bean 定义从三个文件中进行装载，services.xml,messageSource.xml,themeSource.xml。这些路径都是相对主配置文件的相对路径。

注意：

使用"../"路径是可行的，但是不推荐使用。在实际中，该用法也不应该在类路径中出现。(class:../services.xml)你可以使用资源的全限定路径来替代相对路径，例如："file:c:/config/services.xml 或 class:/config/services.xml" 而且需要注意封装路径地址的时候使用${…} 这部分的解释会根据不同的 JVM 系统运行属性不同而不同。

Using the container 使用容器

ApplicationContext 是一个更高级的 factory 实现，来维护不同 BEAN 的注册和它们之间的依赖关系。使用方法 T getBean(Stringname,Class<T> requiredType) 你可以得到你需要的 beans 的实例。

ApplicationContext 使你能够读 bean 定义和按照下面的方式访问它们：

```
// create and configure beans

ApplicationContext context =

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

// retrieve configured instance

PetStoreServiceImpl service = context.getBean("petStore",
```

```
PetStoreServiceImpl.class);

// use configured instance

List userList service.getUsernameList();
```

你使用 getBean()得到了你的 beans 实例。ApplicationContext 接口含有很多其他的获取 beans 实例的方法。但是你的应用代码不需要直接使用它们。实时上，你的应用代码不需要调用 getBean()方法。因此在 Spring APIs 上没有任何依赖。

例如，Spring 为整合 web 框架提供了一些 WEB 跨框架类，例如 controllers 和 JSF-managed beans.

3-1-3节 Bean 概述

一个 Spring IoC 容器管理一个或多个 beans.这些 beans 按照配置元数据进行生成。这些配置元数据例如会保存在 XML 格式的配置文件中<bean/>定义。

在容器内部，这些 bean 定义被表现为 BeanDefinition 对象，通过下面的元数据：

- 一个打包的全限定类名：典型的是实现这个 bean 的具体类。
- 关于 Bean 处理方式的元素定义，例如(scope,lifecycle call backs,等等)
- 指定的其他 bean，其他 bean 与当前 bean 配合来完成具体工作，这样的关系也被称为协作和相互依赖。
- 其他的配置项，这些配置项目会影响生成对象的过程，例如连接数，或连接池的大小。

下面的表格 3.1 说明了 bean 定义的具体信息

属性	使用说明
class	该部分叫做：实例化 beans.
name	为 beans 命名
scope	Bean 的范围
Constructor arguments	构造函数参数，该部分被叫做依赖注入
properties	该部分叫做依赖注入
Autowiring mode	
Lazy-initialization mode	
Initialization method	该部分被称为初始化回调
Destruction method	该部分被称为销毁回调

除了依靠 bean 的定义信息来生成 bean,ApplicationContext 也可以允许注册已经存在的对象，这些对象是在用户外边生成的。这是通过访问 ApplicationContext 的 BeanFactory 通过调用 getBeanFactory(),该方法返回一个 BeanFactory 的一个实现类:
DefaultListtableBeanFactory. 该实现类中有两个方法支持注册容器外面对象的功能：registerSingleton(..)和 registerBeanDefinition(..).而典型的应用程序是通过配置元数据来生成 beans 的。

Naming beans 命名 beans

每一个 bean 具有一个或多个标记。该标记必须在容器内是唯一的，这样才能唯一定位一个 bean.一个 bean 通常只有一个标记，但是如果需要多个，那么另外多出的可以被称为别名。

在基于 XML 的配置元数据中，你使用 id 或 name 属性来指定 bean 的标识。Id 属性允许你指定准确的 id,因为它是一个真实的 XML 元素的 ID 属性，XML 解析器可以做额外的验证，当其他的元素指向它的时候。它比较适合来指定一个 bean 的标识。然而 XML 规范限制了在 id 傻姑娘可以使用的字符数目和可以使用的合法字符，这通常不是个问题，但是如果你需要使用特殊的 XML 字符或是介绍其他的别名给这个 bean,你也可以在 name 属性中来指定，你可以选择使用逗号，分号或是空格来做分割符号。

你不需要必须指定一个 name 或是 id 给一个 bean.如果没有显示的出现 name 或是 id 属性。容器则生成一个唯一的 bean 的名字。然而如果你需要通过名字来指定该 bean 通过 ref 元素，或是 service location 风格的查找，那么你必须提供一个名字。如果使用 inner beans 或是 autowiring collaborators 则可以不提供 bean 的名字。

Bean 命名的约定

命名规范遵循 Java 的标准规范，bean 的名字使用小写字母开头，并且使用逗号等包围起来，例如这样的名字(没有引号) ‘accountManager’,’accountService’,’userDao’,’loginController’等等。

命名规范可以让你的配置文件容易阅读和理解，如果你使用 Spring AOP 它可以为你提供更多的帮助，当你应用 advice 到一组相关的 beans 上的时候。

定义 bean 的别名

在 bean 定义的地方，你可以提供多余一个的 bean 名字，通过使用一组组合到一个 id 属性，并且是任意数目的另外的名字都可以出现在 name 属性。这里的名字列表可以等同一个 bean 的多个别名。这在一些应用中比较有用，例如在应用中的每个组件都可以指定一个通常的依赖，通过使用一个 bean 名字，而这个名字就是组件的名字自身。

指定所有的别名在定义的时候不是总是充分的，然而在一些时候，指定一个 bean 的别名，在其他地方也是有需要，一个通常的案例是，在大型的系统中，配置项目被分割成了一些子的配置项目，每个子的配置项目都有一组基于 XML 的配置元数据，你可以使用<alias/>元素来实现它们。

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

在这个例子中，一个 bean 在相同的容器里，名字被定义为 fromName,在后面的别名定义中，指向 toName.

例如一个子系统的配置元数据 A 会通过一个名字'subsystemA-dataSource'指向一个数据源。而 subsystem B 则会使用一个名字'subsystemB-dataSource'指向一个数据源。当组合成一个主应用的时候，这两个子系统都会通过一个名字'myApp-dataSource'来指向数据源。

使用三个名字来指向同一个对象，你需要在 MyApp 配置元数据中添加如下的别名配置：

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

那么现在就可以解决子系统都可以使用主系统的数据源，而且不会发生混乱。

Instantiating beans 初始化 beans

Bean 的定义是为了生成一个或多个对象集合。容器通过查看 bean 的定义信息，并且使用封装好的配置元数据来生成或是请求一个实际的对象。

如果你使用基于 XML 的配置元数据，你可以制定 一个对象的类型或是类，这个类出现在<bean/>的 class 属性上。该 class 属性，在一个 BeanDefinition 实例内部就对应 Class 属性。你使用 Class 属性可以通过两个方法：

- 典型情况下，指定一个 bean 类来在特定情况下让容器直接进行构造，通过调用该类的构造函数，有点类似在 Java 代码中使用 new 运算符。
- 可以指定一个实际的类包含一个 static 工厂方法，当生成对象的时候该方法被调用。在一些特殊场合下，容器激活一个 static 工厂方法来生成一个 bean.工厂方法返回的对象类型可以是相同的类或是完全不同的类。

内部类的名字

如果你想为一个嵌套的静态类定义一个 bean,你必须使用内部类的二进制名字。
例如如果你在 com.example 包中有一个类 Foo,并且在 Foo 类中有一个静态的内部类叫:Bar,那么'Class'的属性数值在该 bean 定义上将会是 com.example.Foo$Bar

注意这里的$符号是为了分隔外部类与内部类名字的。

使用构造函数来完成初始化

当你通过构造函数的方法来生成 bean，所有常见的类都是可用的，而且会和 Spring 兼容。这就是说，类可以不经过实现任何接口或是编码就可以被开发出来。简单的制定 bean 的类是有效的。然而依据不同类型的 IoC,你也许需要一个默认的(空的)构造函数。

Spring 可以管理任意的虚拟类，并且它不限制管理真实的 JavaBeans.很多的 Spring 用户倾向与在实际的 JavaBeans 上只使用一个无参的构造函数。

和配套的 getters 和 setters 方法。你同样也可以使用更多的非 bean 风格的类在你的容器内，例如你会使用一个连接池，那不是 JavaBean 规范的，Spring 同样可以管理类似的类。
基于 XML 的配置，你会这样来做设置：

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

Instantiation with a static factory method 使用静态方法来初始化

当你定义一个包含静态工厂方法的 bean 的时候，你使用 class 属性来指定类，而使用 factory-method 属性来制定那个工厂方法的名字。你可以调用该方法，并且返回一个实时对象，使用上类似通过构造函数生成的。一个用户也可以通过代码的方式来调用静态工厂方法。

下面的 bean 定义就演示了如何调用一个工厂方法的 bean 定义。该定义没有指定返回的对象类型，只是类里包含了工厂方法，在这个例子中 createInstance()方法一定是个静态方法。

```
<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;
}
}
```

Instantiation using an instance factory method
使用工厂实例来初始化

类似 static factory method，通过一个工厂方法的实例来初始化，来调用一个存在 bean

的非静态方法，来生成一个新的 bean.为了使用该机制，保持 class 属性为空，在 factory-bean 属性上，制定一个 bean 的名字，该 bean 包含了一个实例方法。在 factory-method 属性上设置工厂方法。

```
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
<!-- inject any dependencies required by this locator bean -->
</bean>
<!-- the bean to be created via the factory bean -->
<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;
}
}
```

一个工厂类可以具有多个工厂方法：

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
<!-- inject any dependencies required by this locator bean -->
</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;
}
}
```

注意：

factory bean 代表通过工厂方法来实例化的 bean.

FactoryBean 就是代表 Spring 的 FactoryBean。

3-1-4节 依赖性

Dependency injection(DI)是一个定义对象之间依赖性的过程，也就是说，相关的工作对象只通过构造函数的参数，工厂方法的参数或是相关的设置到对象的属性来确定的。容器然后注入这些依赖，当容器生成 bean 的时候。这个过程是倒置的，因此这个过程的名字也被称为 Inversion of Control（IoC）.bean 自己控制实例化或它的依赖关系的查找路径，通过使用直接的类的构造函数或是 Service Locator 查找模式。

使用 DI 理论，代码会更清晰，而且解析依赖关系的时候会更有效。对象不会查找它自己的依赖性，不知道位置信息或是类的相关依赖性。正因为这样，类变的容易被测试，特别是依赖在接口或是抽象的基类上的时候，这样就允许进行更方便的单元测试。

DI 存在两个主要的课题，基于构造函数的注入依赖，和基于 Setter-based 的注入依赖。

Constructor-based dependency injection 基于构造函数的注入依赖

基于构造函数的依赖被通过调用一个带有一定数量的参数的构造函数来完成，每个构造函数代表一种依赖关系。调用静态的工厂方法也是类似的方法。下面的例子演示了一个类，该类只能通过构造函数来完成依赖。这里需要注意，该来没有特别的地方，它只是个 POJO 并且该类在容器上没有特定的依赖，例如对接口的依赖，基类的依赖，或是有注释性的配置。

```
public class SimpleMovieLister {
// the SimpleMovieLister has a dependency on a MovieFinder
private MovieFinder movieFinder;
// a constructor so that the Spring container can 'inject' a MovieFinder
public SimpleMovieLister(MovieFinder movieFinder) {
this.movieFinder = movieFinder;
}
// business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

构造行数的参数比对通过使用参数的类型，如果没有潜在的混乱在构造函数的参数的 bean 定义的时候，那么构造函数的类定义的时候，参数的顺序与定义 bean 的时候，构造函数的参数顺序相同。可以考虑下面的例子：

```
package x.y;
```

```
public class Foo {
public Foo(Bar bar, Baz baz) {
// ...
}
}
```

没有潜在混乱的存在，假设 Bar,Baz 通过继承的方式没有任何联系。因此下面的配置会工作的很好。你不需要指定构造函数的参数索引或是显示的在<constructor-arg/>标签内指定参数的类型：

```
<beans>
<bean id="foo" class="x.y.Foo">
<constructor-arg ref="bar"/>
<constructor-arg ref="baz"/>
</bean>
<bean id="bar" class="x.y.Bar"/>
<bean id="baz" class="x.y.Baz"/>
</beans>
```

当另外的 bean 被引用的时候，类型是知道的，那么比对就发生了（事实上前面的例子也是如此）当一个简单类型被使用的时候，例如<value>true</value>,Spring 不会直接确定该数值的数据类型，并且不会通过类型来进行比对，考虑下面的例子：

```
package examples;
public class ExampleBean {
// No. of years to the calculate the Ultimate Answer
private int years;
// The Answer to Life, the Universe, and Everything
private String ultimateAnswer;
public ExampleBean(int years, String ultimateAnswer) {
this.years = years;
this.ultimateAnswer = ultimateAnswer;
}
}
```

Constructor argument type matching 构造函数的参数类型比对

在前面的例子中，容器可以使用简单类型进行类型比对，如果你显示的指定了参数的数据类型，类似如下：

```
<bean id="exampleBean" class="examples.ExampleBean">
<constructor-arg type="int" value="7500000"/>
<constructor-arg type="java.lang.String" value="42"/>
</bean>
```

Constructor argument index 构造函数参数索引

使用 index 属性来显示的指定构造函数的参数索引，类似下面的片段：

```
<bean id="exampleBean" class="examples.ExampleBean">
<constructor-arg index="0" value="7500000"/>
<constructor-arg index="1" value="42"/>
</bean>
```

为了解决多个简单类型的混乱，制定索引来避免混乱，尤其是构造函数的参数数据类型都相同的时候，这里注意，索引是从 0 开始的。

Constructor argument name 构造函数的名字

在 Spring3.0 你也可以使用构造函数的参数名字来避免前面提到的混乱：

```
<bean id="exampleBean" class="examples.ExampleBean">
<constructor-arg name="years" value="7500000"/>
<constructor-arg name="ultimateanswer" value="42"/>
</bean>
```

这里需要注意如果上述的配置工作，你的代码必须在 debug 的模式下被编译，让 Spring 可以查找参数名字，从构造函数中。如果你不这样做，你可以使用@ConstructorProperties 的 JDK 注释来显示的声明你的参数名字，下面的例子就演示了这个过程：

```
package examples;
public class ExampleBean {
// Fields omitted
@ConstructorProperties({"years", "ultimateAnswer"})
public ExampleBean(int years, String ultimateAnswer) {
this.years = years;
this.ultimateAnswer = ultimateAnswer;
}
}
```

Setter-based dependency injection 基于 setter-based 的依赖注入

Setter-based 的 Di 是通过容器调用 setter 方法，在你的 beans 被激活后来实现的。
下面的例子演示了一个类只使用 setter 的方式进行依赖注入。该类是传统的 JAVA 类。这是个没有依赖关系的 POJO.

```
public class SimpleMovieLister {
// the SimpleMovieLister has a dependency on the MovieFinder
private MovieFinder movieFinder;
// a setter method so that the Spring container can 'inject' a MovieFinder
public void setMovieFinder(MovieFinder movieFinder) {
this.movieFinder = movieFinder;
}
// business logic that actually 'uses' the injected MovieFinder is omitted...
```

```
}
```

ApplicationContext 支持构造函数和 setter-based 的 DI，针对一个容器管理下的 beans.
它也支持 setter-based 的注入，当一些依赖已经通过构造函数的方式进行了注入。
你可以在一个 BeanDefinition 下配置依赖注入，这里的 BeanDefinition 可以让你使用 PropertyEditor 实例来转船一个属性从一种格式到另一种格式。然而，绝大部分的用户不会直接使用这里的类，宁肯使用基于 XML 的配置元数据文件，那么这里的转换就是容器做内部转换了。

Constructor-based or setter-based DI? 是选用构造函数还是 setter 方法？
既然你可以混合使用构造函数和 setter 方法的 DI，那么你可以使用构造函数来部署强制的依赖注入，而使用 setters 进行可选择的依赖注入。这里注意，使用@Required 注释可以让 setters 需要一个依赖。

Spring 团队通常实现 setter 注入，因为大量的构造函数参数不是太简洁，显得笨拙。尤其是属性是可选择的情况下。Setter 方法可以让类对象可以在配置和再后面重复注入。如果使用 JMX 的 Mbeans 来管理是一个比较典型的应用。

一些构造函数注入的支持者，提供所有的对象进行注入依赖，那么这意味着对象总是返回给客户代码，在整个初始化状态，这里的缺点是对象变得的不容易在配置和重新注入。

使用依赖注入可以解决大多数的类的需求，但是当你处理第三方类的时候，而且你没有代码，这个选择为你而开始，你个合法的类不暴露任何的 setter 方法，那么只能通过构造函数注入了。

Dependency resolution process 依赖关系的确定过程
通常按照如下的步骤：

1， ApplicationContext 生成，并且结合配置元数据进行初始化。
2， 针对每一个 bean，它的依赖按照属性的形式，构造函数参数，和静态工厂方法的参数的形式来确认。当 bean 被生成的时候，这些依赖关系就与 bean 进行绑定了。
3， 每个属性，或构造函数参数就是一个实际定义的数据属性集合，或是对容器内另外 bean 的引用。
4， 每个属性或是构造函数的参数数值都开始进行类型转换。默认情况下，转换的 string 格式可以到内建的所有类型，例如 int,long,String ,boolean 等等。

Spring 容器验证每个 bean 的配置信息当容器生成的时候，其中包括：验证引用的 bean 是否是有效的 bean 等。然后 bean 的属性直到 bean 被实际生成的时候才会被赋值。如果 bean 的范围是 singleton-scoped 并且设置为 pre-instantiated（默认设置），那么当容器生成的时候即 bean 生成。否则 bean 只有在需要的时候才被生成。生成一个 bean 引起连锁的 beans 的生成。其中包括具备依赖关系的 bean 或是依赖关系的依赖关系的相关 bean 都被生成。

闭合依赖

如果你主要使用构造函数依赖注入，这也许会生成一个无法解析的闭路依赖的案例。
例如：类 A 需要一个类 B 的实例，这是通过构造函数进行如入的。并且类 B 也需要一个类 A 的实例，同样是通过构造函数注入来完成。如果你为类 A 和类 B 进行这样的相互配置。Spring IoC 容器就会检测到一个闭路引用在运行的时候，而且会抛出一个异常：BeanCurrentlyInCreationException.

一个可选择的解决方案可以编辑一些类的源代码，可以通过配置成 setters 来进行注入依赖。另外就是阻止构造函数注入，只使用 setteer 注入。从另一方面，这是不提倡的，你可以使用 setter 注入来应对闭路依赖。

与常见的依赖不同，一个闭路依赖在 bean A 和 bean B 之间，必须强制其中的一个 bean 注入到另一个 bean 中，在另外的 bean 完全初始化它自身之前。（ 这类似先有鸡蛋还是先有鸡的问题）

通常你会信任 Spring 会做正确的事情。它检测配置信息的问题，例如引用了个不存在的 bean. 出现了闭路依赖，在容器装载的时候。Spring 设置属性和处理依赖会尽可能的推迟，当 bean 生成的时候。这意味着 Spring 容器装载是正确的时候，但是后面你请求一个对象的时候，反而会出现问题。这样的延迟，就是 ApplicationContext 的应用为什么默认是提前初始化的 beans.处于资源和内存的消耗情况，你需要发现配置的一些问题，当 ApplicationContext 被生成的时候，而不是以后再检测。你仍然是可以改变默认的行为，让 singleton 的 beans 进行进行延迟初始化。

如果没有闭路依赖，那么一个或多协作的 beans 被注入到一个独立 bean 的时候，那么每个协作的 bean 必须提前配置成可以被注入到中心依赖 bean.这意味着，bean A 在 bean B 具有一个依赖。那么 Spring 的 IoC 容器会完全提前配置好 bean B 来调用在 bean A 上的 setter 方法。该 bean 被初始化后，那么它的相关依赖关系就被设置。并且其他相关的关系生命周期的方法也会被调用(configured_init_method 或是 IntializingBean_callback_mehod).

Examples of dependency injection 依赖注入的例子

下面的例子使用了 XML 的配置元数据来进行 setter 的 DI 配置，其中一小部分的代码片段定义如下：

```
<bean id="exampleBean" class="examples.ExampleBean">
<!-- setter injection using the nested <ref/> element -->
<property name="beanOne"><ref bean="anotherExampleBean"/></property>
<!-- setter injection using the neater 'ref' attribute -->
<property name="beanTwo" ref="yetAnotherBean"/>
<property name="integerProperty" value="1"/>
</bean>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {
```

```
private AnotherBean beanOne;
private YetAnotherBean beanTwo;
private int i;
public void setBeanOne(AnotherBean beanOne) {
this.beanOne = beanOne;
}
public void setBeanTwo(YetAnotherBean beanTwo) {
this.beanTwo = beanTwo;
}
public void setIntegerProperty(int i) {
this.i = i;
}
}
```

前面的例子，setter 的声明是匹配在 XML 元数据中的声明的，下面的是基于构造函数的 DI 配置：

```
<bean id="exampleBean" class="examples.ExampleBean">
<!-- constructor injection using the nested <ref/> element -->
<constructor-arg>
<ref bean="anotherExampleBean"/>
</constructor-arg>
<!-- constructor injection using the neater 'ref' attribute -->
<constructor-arg ref="yetAnotherBean"/>
<constructor-arg type="int" value="1"/>
</bean>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {
private AnotherBean beanOne;
private YetAnotherBean beanTwo;
private int i;
public ExampleBean(
AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
this.beanOne = anotherBean;
this.beanTwo = yetAnotherBean;
this.i = i;
}
}
```

构造函数的参数出现在配置文件中的，也将在 ExampleBean 的代码中出现。
现在考虑这个例子的更多方面，Spring 开始使用调用一个静态工厂方法来返回一个对象的实

例：

```
<bean id="exampleBean" class="examples.ExampleBean"
factory-method="createInstance">
<constructor-arg ref="anotherExampleBean"/>
<constructor-arg ref="yetAnotherBean"/>
<constructor-arg value="1"/>
</bean>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {
// a private constructor
private ExampleBean(...) {
...
}
// a static factory method; the arguments to this method can be
// considered the dependencies of the bean that is returned,
// regardless of how those arguments are actually used.
public static ExampleBean createInstance (
AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
ExampleBean eb = new ExampleBean (...);
// some other operations...
return eb;
}
}
```

提供给静态工厂方法的参数通过<constructor-arg/>来提供的。完全跟实际的构造函数需要的完全相同。返回的类类型，不需要必须与包含工厂方法的类相同。

而一个实例工厂方法的使用，跟这里的静态工厂方法使用的过程几乎相同，这里不再另外举例子说明了。

3-1-5节 Bean 范围

3-1-6节 自定义 bean 的状态

生命周期回调

初始化回调

退出回调

默认的初始化和删除回调

生命周期机制

开始启动和退出回调

在非web应用程序中彻底推出IoC容器

ApplicationContextAware and BeanNameAware

其他的Aware interfaces

3-1-7节 Bean 定义依赖性

3-1-8节 容器的扩展点

使用 BeanPostProcessor 定义 bean

使用 BeanFactoryPostProcessor 定义配置元数据

使用 FactoryBean 来定义实例化过程

3-1-9节 基于注释的元数据配置

@Required

@Autowired 和 @Inject

@qualifiers

CustomAutowireConfigurer

@Resource

@PostConstruct and @PreDestroy

3-1-10节 类路径查找和可管理 bean

@Component和更多注释语句

自动检测类和注册bean定义

使用过滤器和自定义扫描

使用组件定义bean的元数据

名字自动侦测bean

为自动侦测组件提供一个范围

使用注释提供qualifier元数据

3-1-11节 基于 java 的容器配置

@Configuration和@Bean概念

使用AnnotationConfigApplicationContext实例化容器

例子介绍

使用**register(Class<?>...)**建立可编程的容器

scan(String...)启用组件扫描

AnnotationConfigWebApplicationContext支持**WEB**应用

组合基于java的配置

使用**@Import**

注入依赖

为了方便检索引入全限定beans导入

整合XML和java配置

@Configuration基于XML核心的配置

声明@Configuration的类当作bean声明

<context:component-scan/> 检测类

@ImportResource使用XML配置

使用@Bean注释

声明一个bean

注入依赖

接收声明周期回调

指定bean范围

使用@Scope注释

@Scope and scoped-proxy

检索方法注入

自定义bean名字

Bean别名

更多JAVA配置的内部工作原理

3-1-12节 注册 LoadTimeWeaver

3-1-13节 ApplicationContext 更多功能

使用MessageSource进行内部初始化

标准与自定义组件

访问底层资源

WEB应用中更方便的ApplicationContext初始化

作为RAR文件来部署ApplicationContext

3-1-14节 BeanFactory

BeanFactory 或 ApplicationContext

黏性代码和不好的框架

3-2节 资源

3-2-1节 介绍

3-2-2节 Resource 接口

3-2-3节 内置的 Resource 应用

UrlResource

ClassPathResource

FileSystemResource

ServletContextResource

InputStreamResource

ByteArrayResource

3-2-4节 ResourceLoader

3-2-5节 ResourceLoaderAware 接口

3-2-6节 Resources 依赖

3-2-7节 Application contexts 和资源路径

构建application contexts

ClassPathXmlApplicationContext

匹配符的应用

Ant 风格

classpath*:前缀

其他相关说明

FileSystemResource警告

3-3节 校验，数据绑定和类型转换

3-3-1节 介绍

3-3-2节 使用 Validator 接口

3-3-3节 分析代码

3-3-4节 BeanWrapper 和 bean 处理

3-3-5节 Setting and getting

3-3-6节 PropertyEditor

注册自定义的属性编辑器

3-4节 Spring 表达式（SpEL）

3-4-1节 介绍

3-4-2节 特性说明

3-4-3节 使用 Expression 接口

EvaluationContext

3-4-4节 定义 bean 的表达式支持

基于 XML

基于注释

3-4-5节 语言参考

Literal表达式

Properties, Arrays, Lists, Maps, Indexers

Inline lists

Array

Methods

Operators

Relational operators

Logical operators

Mathematical operators

Assignment

Types

Constructors

Variables

Functions

Bean references

Ternary Operator (If-Then-Else)

Elvis Operator

Safe Navigation operator

Collection Selection

Collection Projection

Expression templating

3-4-6节 相关例子

3-5节 AOP 编程

3-6节 Spring AOP APIs

3-7节 测试

第4章 数据访问

4-1节 事务管理

4-2节 DAO 支持

4-3节 使用 JDBC 进行数据访问

4-4节 ORM 数据访问

4-5节 O/X Mappers

第5章 WEB