OpenDoc Series'

Spring Framework 概述

-Introduction to the Spring Framework

V1.0

原著：Rod Johnson
译者：Digital Sonic

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

文档说明

参与人员：

作者	联络
Digital Sonic	dingx(at)citiz.net

(at) 为 email @ 符号

发布记录

版本	日期	作者	说明
1.0	2006.01.20	Digital Sonic	翻译
1.0	2006.01.20	夏昕	文档格式编排

OpenDoc 版权说明

本文档版权归原作者所有。
在免费、且无任何附加条件的前提下，可在网络媒体中自由传播。

如需部分或者全文引用，请事先征求作者意见。

如果本文对您有些许帮助，表达谢意的最好方式，是将您发现的问题和文档改进意见及时反馈给作者。当然，倘若有时间和能力，能为技术群体无偿贡献自己的所学为最好的回馈。

Open Doc Series 目前包括以下几份文档：

- **Spring 开发指南**
- **Hibernate 开发指南**
- **ibatis2 开发指南**
- **Webwork2 开发指南**
- **持续集成实践之 CruiseControl**
- **Shift to Dynamic Smalltalk**

以上文档可从 http://www.redsaga.com 获取最新更新信息

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

Spring Framework 概述

自从这篇文章的第一版在 2003 年 10 月发表以来，Spring 框架正在逐步普及。经历了 1.0 最终版到现在的 1.2 版，而且被运用于很多行业和项目中。在这篇文章中，我会解释 Spring 为什么会获得成功，并告诉你我十分肯定它能帮助你开发 J2EE 应用程序。

又是一个框架？

你可能正在想“不过是另一个的框架”。如今有这么多 J2EE 框架，并且你可以建立你自己的框架，为什么你应该读这篇文章或者下载 Spring 框架(或者你还没有下载)？社区中持续的高关注度暗示了 Spring 一定有它的价值；这也有很多技术原因。

以下的几个原因让我相信 Spring 是独一无二的：

- 它关注于很多其它框架没有关注的地方。Spring 着重于提供一种管理你业务对象的方法。
- Spring 是全面的、模块化的。Spring 采用分层架构，这意味着你可以仅选择其中任何一个独立的部分，而它的架构是内部一致的。因此你能从学习中获得最大的价值。例如，你可以仅用 Spring 来简化你的 JDBC 使用，或者你可以选择使用 Spring 来管理你的业务对象。把 Spring 增量地引入现有的项目中是十分容易的。
- Spring 从设计之初就是要帮助你写出易于测试的代码。Spring 是测试驱动项目的一个理想框架。
- Spring 是一个日益重要的集成技术，它的角色已得到一些大厂商的认可。

Spring 不需要你的项目再依赖于另一个框架。Spring 也许能称得上是一个“一站式”商店，提供了大多数传统应用所需要的基础结构。它还提供了别的框架没有涉及到的东西。

作为一个从 2003 年 2 月开始的开源项目，Spring 有深厚的历史背景。这个开源项目源自我在 2002 年底出版的《Expert One-on-One J2EE Design and Development》中的基础代码。书中展现了 Spring 背后的基础性架构思考。然而，这个架构概念可以追溯到 2000 年早期，并反映了我在一系列成功的商业项目的基础结构的开发中所获得的经验。

从 2003 年 1 月起，Spring 落户于 SourceForge。现在有 20 位开发者，一些主要人员把所有的时间都花在了 Spring 的开发和支持上。繁荣的开源社区帮助它茁壮成长，这远非任何个人所及。

Spring 架构上的好处

在继续深入前，让我们来看看 Spring 带给一个项目的好处：

- Spring 可以有效组织你的中间层对象，无论你是否选择使用 EJB。Spring 关心那些当你只选择 Struts 或其他为 J2EE API 量身定做的框架时被留给你解决的问题。Spring 的配置管理服务可以被运用于任何运行环境的各种架构性分层中，这也许是中间层中最有价值的。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

- Spring 可以消除在很多项目中所常见的单例的过度使用。在我看来，它的主要问题是降低了可测试性和面向对象的程序。
- Spring 通过一种在应用程序和项目之间一致的方法来处理配置，这消除了需要自定义配置文件格式的烦恼。还记为了知道某个类要找哪个神奇的属性项或系统属性而不得不去读 Javadoc，甚至读源代码吗？有了 Spring 你只要简单地看看类的 JavaBean 属性或构造参数。控制反转和依赖注入(将在下文讨论)的使用帮助实现了这一简化。
- Spring 通过把针对接口而非类编码的代价降低到几乎未零来帮助养成好的编码习惯。
- Spring 被设计为让构建在它之上的应用程序尽可能少地依赖于它的 API。大多数 Spring 应用程序中的业务对象不依赖于 Spring。
- 构建于 Spring 之上的应用程序很容易进行单元测试。
- Spring 使得是否使用 EJB 成为实现时的选择，而非架构上的决定。你能在不改变调用代码的情况下选择用 POJO 或 EJB 来实现业务接口。
- Spring 帮助你在不用 EJB 的情况下解决很多问题。Spring 能提供一种适用于很多应用程序的 EJB 的替代品。例如，Spring 可以无需 EJB 容器，用 AOP 来提供声明性事务管理；如果你仅与一个数据库打交道，甚至可以没有 JTA 实现。
- Spring 为数据访问提供了一个一致的框架，无论使用 JDBC 还是像 TopLink、Hibernate 或者 JDO 实现这样的实体关系映射产品。
- Spring 为很多方面提供了一种一致的简单的编程模型，这使得它成为了一种理想的架构“胶”。你可以从 Spring 访问 JDBC、JMS、JavaMail、JNDI 和很多其他重要 API 的途径中发现这种一致性。

Spring 是一种帮助你使用 POJO 来构建应用程序的基础技术。要达到这个目标需要一个能将复杂性隐藏起来的成熟的框架。

因此 Spring 真的可以帮助你实现针对你的问题的最简单可行的解决方案。这是十分有价值的。

Spring 做了些什么？

Spring 提供许多功能，所以我将依次地快速浏览每个主要方面。

任务描述

首先，让我们明确一下 Spring 的范围。尽管 Spring 囊括了很多东西，但我们应该清楚的知道它该涉及什么，不该涉及什么。

Spring 的主要目的是使 J2EE 更易于使用，培养好的编程习惯。它通过使用一种能适用于很多环境下的基于 POJO 的编程模型来实现这一目的。

Spring 不重新发明轮子。因此你会发现 Spring 中没有日志，没有连接池，没有分布式事务调度。所有这些东西都由开源项目(例如提供我们所有的日志输出的 Commons Logging，或者是 Commons DBCP)或你的应用服务器提供。同样的道理，我们不提供实体/关系映射层。因为有像 TopLink、Hibernate 和 JDO 这样的优秀的解决方案。

Spring 致力于使现有技术更加易用。例如，尽管我们没有底层业务的事务调度，但我们提供了一个凌驾于 JTA 或其他事物策略的抽象层。

Spring 不直接与其他开源项目竞争，除非我们觉得我们能提供些新的东西。比如说，像

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

其他开发者一样，我们从未就 Struts 感到满意，我们觉得 MVC Web 框架还有改进的余地。(随着 Spring 应用地快速推广，很多人也同意了我们的观点。)在很多领域，例如它的轻量级 IoC 容器和 AOP 框架，Spring 有直接的竞争，但 Spring 确实是这些领域的先锋。

Spring 得益于内部一致性。所有开发者正唱着同一首赞歌，基础思想依然忠于《Expert One-on-One J2EE Design and Development》中提出的思想。我们已经能够在多个领域中使用些核心概念，例如控制反转。

Spring 可用于各种应用服务期。当然保证可移植性一直是一个挑战，但我们避免了开发者眼中的各种平台特有的或非标准的东西，支持 WebLogic、Tomcat、Resin、Jboss、Jetty、Geronimo、WebSphere 和其他应用服务器。Spring 的非侵入性、POJO 方法是我们可以利用环境特有特性而不用放弃可移植性，就像 Spring 1.2 中在掩护下使用 BEA 特有 API 从而开启增强 WebLogic 事务管理功能。

控制反转(Inversion of Control, IoC)容器

Spring 的核心是为与 JavaBeans 一起工作而设计的 org.springframework.beans 包。这个包一般不直接被用户调用，而是作为 Spring 功能的基础。

下一个更高的抽象层是 bean 工厂。一个 Spring 的 bean 工厂是一个普通的工厂，它能通过名称获得对象，并管理对象的关系。

Bean 工厂支持两种模式的对象：

- 单例：这种情况下，存在一个有特定名称，在查找时能被获取的共享对象实例。这是默认的，也是最常用的模式。是无状态服务对象的理想选择。
- 原型或非单例：在这种情况下，每次获取操作都会创建一个独立的对象作为结果。例如，这能被用来使每个调用者都有自己的独立的对象引用。

因为 Spring 容器管理对象间的关系，它能在以下情况添加值，在诸如受管理的 POJO 的透明池、支持热交换之类的服务需要的地方，为在运行时交换目标引用但不影响调用者和线程安全性而由容器引入的一个间接层中。依赖注入的众多优点之一(简单讨论一下)就是这所有的一切几乎是透明的，没有 API 介入。

org.springframework.beans.factory.BeanFactory 是一个简单的接口，它能通过多种途径被实现。BeanDefinitionReader 接口将元数据格式从 BeanFactory 各自的实现中分离出来，所以 Spring 提供的普通 BeanFactory 实现能和不同类别的元数据一起使用。尽管很少有人发现有这个必要，你还是可以简单地实现你自己的 BeanFactory 或者 BeanDefinitionReader。最常用的 BeanFactory 定义是：

- XmlBeanFactory：它可解析简单直观的定义类和命名对象属性的 XML 结构。我们提供了一个 DTD 帮助简化编写。
- DefaultListableBeanFactory：它提供了解析属性文件中的 bean 定义的能力，可通过编程创建 BeanFactory。

每个 bean 定义能被当作一个 POJO(用类名和 JavaBean 初始属性或构造方法参数来定义)，或被当作一个 FactoryBean。FactoryBean 接口添加了一个间接层。一般，这用来创建用 AOP 或其他方法的代理对象：例如，添加声明性事物管理的代理。这在概念上和 EJB 的拦截机制相近，但实践起来更方便，更有效。

BeanFactory 能选择性地参与于一个层次结构中，“继承”先辈的定义。这使得像控制器 servlet 这样的个体资源能拥有自己的独立对象集的同时，在整个应用程序中能共享公共配置。

如此使用 JavaBeans 的动机在《Expert One-on-One J2EE Design and Development》的第

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

四章中已经描述过了，你同样也可以在 theServerSide 站点上以免费 PDF 的形式获得(/articles/article.tss?l=RodJohnsonInterview)。

通过 bean 工厂的概念，Spring 成为了一个控制反转容器。(我不太喜欢容器这个词，因为它令人想起了类似 EJB 容器的重量级容器。一个 Spring 的 BeanFactory 是一个能用一行代码创建，无需特别部署的容器。)Spring 用了名为依赖注入(Dependency Injection, DI)的控制反转，依赖注入是由 Martin Fowler、Rod Johnson 和 PicoContainer 团队在 2003 年底命名的。

控制反转背后的原则常被称为好莱坞原则："不要打电话找我，我会打给你的。"IoC 将创建的职责从应用程序代码中搬到了框架中。但在你的代码调用一个传统类库时，IoC 框架调用你的代码。在很多 API 中的生命周期回调证明了这点，比如会话 EJB 的 setSessionContext()方法。

依赖注入是 IoC 的一种形式，它消除了对容器 API 的显式依赖；普通的 Java 方法被用来将诸如协作对象或配置值之类的依赖注入应用程序对象实例。涉及到配置的地方，这就意味着在像 EJB 这样的传统容器架构中，一个组件可以调用容器并说"我需要的对象 X 在什么地方"，有了依赖注入容器指出组件需要对象 X，并在运行时将它提供给组件。容器通过方法签名(一般是 JavaBean 的属性或构造方法)和可能的诸如 XML 的配置数据来实现这一功能。

两种主要的依赖注入是 Setter 注入(通过 JavaBean 的 setter 注入)；和构造方法注入(通过构造方法参数注入)。Spring 对两者都提供了很好的支持，你甚至可以在配置一个对象时将两者结合起来。

在支持各种形式的依赖注入的同时，Spring 也提供一系列回调事件，和一个针对某些需要传统查找的 API。但是，我们基本上是建议使用纯依赖注入途径的。

依赖注入有几个重要的好处。例如：

- 因为组件不需要在运行时查找协作者，所以它们更易开发和维护。在 Spring 版本的 IoC 中，组件通过暴露 JavaBean 的 setter 方法或构造方法参数来表示它们对其他组件的依赖。这相当于 EJB 的 JNDI 查找，EJB 的 JNDI 查找需要编写代码、设置环境参数。
- 由于同样的原因，应用程序代码更易测试。例如，JavaBean 属性是简单的，纯 Java 的并且容易测试：简单地编写一个创建对象并设置相关属性的独立 Junit 测试方法。
- 一个好的 IoC 实现保留了强类型。如果你需要用一个通用工厂来查找协作者，你可以把结果转换为需要的类型。这并不是主要问题，但这不优雅。使用 IoC，你在代码中表达了强类型依赖后，框架会负责类型转换。这意味着类型不匹配会在框架配置应用程序时被当作错误抛出；你不必在你的代码中担心类转换异常。
- 依赖是显式的。例如，如果一个应用程序类尝试加载一个属性文件或通过实例连接一个数据库，不读代码可能弄不清楚环境参数(有复杂的测试并降低了部署的灵活性)。使用了依赖注入的手段，依赖是显式的，能通过构造方法或 JavaBean 属性得知。
- 大多数业务对象不依赖 IoC 容器 API。这使得使用现有代码变得十分容易，并且能方便地使用 IoC 容器内和容器外的对象。比如说，Spring 的用户经常把 Jakarta Commons DBCP 数据源配置为一个 Spring bean：没有必要写代码来实现这一步。我们说一个 IoC 容器不是侵入性的：使用它并不用让你的代码依赖于它的 API。几乎任何 POJO 能作为一个 Spring bean 工厂中的组件。现有的 JavaBean 或有多参数构造方法的对象都能很好地工作，但 Spring 也需要对从静态工厂方法实例化的对象或者 IoC 容器管理的其他对象的方法提供特别支持。

这最后一点需要强调一下。依赖注入不像传统容器架构(比如 EJB)那样应用程序代码存在最小限度的对容器的依赖。这意味着你的业务对象可以潜在地被运行于其他依赖注入框架，或不使用任何框架，而不需要改变代码。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

以我和 Spring 用户的经验来说，就算再怎么强调 IoC(特别是依赖注入)带给应用程序代码的好处都不为过。

尽管依赖注入刚在 J2EE 社区中达到它的黄金时间，但它并不是一个新概念。还有其他可选的 DI 容器：notably、PicoContainer 和 HiveMind。PicoContainer 是轻量级的且强调通过构造方法而不是 JavaBean 属性来表现依赖。它不在 Java 代码外使用元数据，与 Spring 相比这限制了它的功能。HiveMind 在概念上更接近 Spring(它也关注 IoC 以外的东西)，但它缺乏 Spring 项目这样广泛的领域且没有相同规模的用户社群。EJB 3.0 也会提供基本的 DI 能力。

Spring 的 BeanFactory 是非常轻量级的。用户们曾成功地在 applet 和独立 Swing 应用程序中使用过它们。(它们在 EJB 容器中也有很好表现。)它们没有特殊的部署步骤，也没有与容器本身相关的可察觉的启动时间(尽管容器配置这类对象需要花时间去初始化)。这种在应用程序的任何一层中都能立即实例化一个容器的能力是十分有价值的。

Spring 的 BeanFactory 的概念贯穿于整个 Spring 中，这也是 Spring 如此内部一致的一个关键原因。在 IoC 容器中，Spring 也是唯一一个将 IoC 作为基本概念贯穿在一个功能丰富的框架中的。

对于一个应用程序开发者来说，一个或多个 BeanFactory 提供了一个定义明确的业务对象层。这类似于本地会话 bean 层，但更简单且更强大。不像 EJB，这层中的对象可能是相关的，它们的关系被拥有它们的工厂管理着。有一个定义明确的业务对象层对于一个成功的架构来说是十分重要的。

一个 Spring 的 ApplicationContext 是 BeanFactory 的一个子接口，它提供了如下支持：

- 消息查找，支持国际化
- 一种事件机制，允许发布应用程序对象或者可选的注册以接收事件
- 自动识别特殊的应用程序细节或通用 bean 定义来自定义容器行为
- 可移植文件和访问资源

XmlBeanFactory 例子

Spring 用户一般用 XML“bean 定义”文件来配置他们的应用程序。一个 Spring XML bean 定义文件的根是<beans>元素。<beans>元素包含一个或多个<bean>定义。我们一般会指定每个 bean 定义的类和属性。我们还必须指定 id，这是我们在自己的代码里使用这个 bean 时的名字。

让我们来看一个简单的例子，这个例子配置三个有 J2EE 应用程序常见关系的对象：

- 一个 J2EE 数据源
- 一个使用数据源的 DAO
- 一个在工作中使用 DAO 的业务对象

下面的例子中，我们使用 Jakarta Commons DBCP 项目中的一个 BasicDataSource。(C3PO 项目中的 ComboPooledDataSource 也是一个不错的选择。) BasicDataSource，和许多别的现有类一样，能轻松地使用于 Spring bean 工厂中，因为它提供了 JavaBean 样式的配置。在关闭时需要调用的 close 方法能通过 Spring 的“destroy-method”属性来注册，这样 BasicDataSource 就不需要实现 Spring 的接口了。

```
<beans>


    <bean id="myDataSource"
            class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/mydb" />
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        <property name="username" value="someone" />
    </bean>
```

所有我们感兴趣的 BasicDataSource 属性都是字符类型的，所以我们用“value”属性来指定它们的值。(这个快捷方式是 Spring 1.2 引进的。这个对<value>子元素的一个方便的替代，它甚至在有问题的 XML 属性值中也能使用。)Spring 使用标准 JavaBean PropertyEditor 机制在需要时将字符表达式转换为其他类型。

现在我们定义 DAO，其中有一个对数据源的 bean 引用。Bean 之间的关系用“ref”属性或<ref>元素来指定。

```
    <bean id="exampleDataAccessObject" class="example.ExampleDataAccessObject">
        <property name="dataSource" ref="myDataSource" />
    </bean>
```

业务对象有一个 DAO 的引用和一个 int 属性(exampleParam)。在这个例子中，我用了与 1.2 版之前相似的子元素语法。

```
    <bean id="exampleBusinessObject" class="example.ExampleBusinessObject">
        <property name="dataAccessObject"><ref
bean="exampleDataAccessObject"/></property>
        <property name="exampleParam"><value>10</value></property>
    </bean>


</beans>
```

对象间的关系一般在配置中显式设置，就像例子中那样。我们认为在大多数情况下这是一件好事。但是，Spring 也提供了对我们称为“自动装配 (autowire)”的支持，它指出了 bean 之间的依赖。这种做法的限制(和 PicoContainer 一样)是如果存在一个特定类型的多个 bean，那么没办法判断该解析那个类型的哪个实例依赖。从积极的一面看，无法满足的依赖能在工厂初始化时被捕捉。(Spring 还提供了对显式配置的一个可选依赖检查来实现这个目标。)

如果你不想显式编写这些关系，我们可以像下面这样在上面的例子中使用自动装配这个特性。

```
    <bean id="exampleBusinessObject" class="example.ExampleBusinessObject"
            autowire="byType">

        <property name="exampleParam" value="10" />
    </bean>
```

使用这种用法，Spring 会判断出 exampleBusinessObject 的 dataSource 属性应该设置到当前 BeanFactory 中的 DataSource 实现。如果在当前 BeanFactory 中没有或有多个满足要求的类型，会产生一个错误。我们仍然需要设置 exampleParam 属性，因为它不是一个引用。

自动装配有降低配置量的好处。它也意味着容器能使用反射来学习应用程序结构，所以如果你添加一个附加的 JavaBean 属性构造方法参数，它可以成功地移植而不用改变配置。这是自动装配需要精心计算的一个折衷。

将关系从 Java 代码中提取出来比起硬编码来说有很大好处，因为它可以改变 XML 文件而不用改变一行 Java 代码。例如，我们可以简单地通过改变 myDataSource bean 定义来引用一个不同的 bean 类来使用另一个连接池或者测试数据源。我们能像下面这样用 Spring 的 JNDI 位置 FactoryBean 来从一个 XML 片断里的应用服务器中获得数据源。这对 Java 代码和任何其他 bean 定义没有影响。

```
    <bean id="myDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
        <property name="jndiName" value="jdbc/myDataSource" />
    </bean>
```

现在让我们看看范例业务对象的 Java 代码。注意下面的代码中没有对 Spring 的依赖。与 EJB 容器不同，一个 Spring 的 BeanFactory 不是侵入式的：你通常不需要有意将它编入你的对象中。

```
public class ExampleBusinessObject implements MyBusinessObject {

    private ExampleDataAccessObject dao;
    private int exampleParam;

    public void setDataAccessObject(ExampleDataAccessObject dao) {
        this.dao = dao;
    }

    public void setExampleParam(int exampleParam) {
        this.exampleParam = exampleParam;
    }

    public void myBusinessMethod() {
        // 使用 dao
    }
}
```

注意属性的 setter，它们符合 bean 定义中的 XML 引用。这些在对象被使用前由 Spring 调用。

这类应用程序 bean 不需要依赖 Spring：他们不需要实现任何 Spring 接口或扩展 Spring 类，它们仅需要遵循 JavaBeans 命名规范。在 Spring 应用程序上下文外重用是很方便的，比如在一个测试环境中。用它的默认构造方法初始化它，并通过 setDataSource()和 setExampleParam()手动设置其属性。只要你有一个无参数的构造方法，你可以自由定义其他获取多个属性的构造方法如果你想支持一行代码中的可编程构造。

注意业务接口调用者中没有声明将要使用的 JavaBean 属性。它们是实现细节。我们能轻松插入拥有不同 bean 属性的实现类而不用影响关联的对象或调用代码。

当然 Spring 的 XML bean 工厂还有许多别的功能，但让你对基本的方法一个了解。和简单属性、JavaBeans PropertyEditor 的属性一样，Spring 能处理列表、映射和 java.util.Properties。其他高级容器功能包括：

- 内部 bean，一个属性元素包含一个上层看不到的匿名 bean 定义。
- Post 处理器，特别的自定义容器行为的 bean 定义。
- 方法注入，IoC 的一种形式，容器实现一个抽象方法或覆盖一个具体方法来诸如一个依赖。这比依赖注入的 Setter 或构造方法注入更少用。但是，在为每次调用查找新的对象实例或允许配置改变额外时间时避免显式容器依赖方面这很有用，例如，在一个有 SQL 查询支持的方法实现和一个 fil 系统读入另一个。

Bean 工厂和应用程序上下文经常在 J2EE 服务器或 web 容器定义的范围内存在关联，比如：

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

- Servlet 上下文：在 Spring MVC 框架中，一个为每个 web 程序定义的应用程序上下文包含公共对象。Spring 提供了通过一个监听器或不依赖于 Spring MVC 框架的 servlet 来初始化这类上下文的能力，这也可以被用在 Struts、WebWork 或其他 web 框架中。
- 一个 Servlet：Spring MVC 框架中的每个控制器 servlet 有它自己的源自根(应用程序级)应用程序上下文的上下文。同样也能在 Struts 或其他 web 框架中实现这个。
- EJB：Spring 提供了简化 EJB 认证的 EJB 超类，还提供了一个从 EJB Jar 文件中加载的 BeanFactory。

这些 J2EE 规范提供的钩避免了使用单例来自展一个 bean 工厂。

然而，如果我们愿意的话，可以通过编码初始化一个 BeanFactory，尽管这是没有什么价值的。例如，我们能像下面两行代码那样建立一个 bean 工厂并获得一个定义好的业务对象的引用：

```
XmlBeanFactory bf = new XmlBeanFactory(new ClassPathResource("myFile.xml", getClass()));
MyBusinessObject mbo = (MyBusinessObject) bf.getBean("exampleBusinessObject");
```

这个代码能工作在应用服务器外：它不依赖于 J2EE，因为 Spring 的 IoC 容器是纯 Java 的。Spring Rich 项目(一个用 Spring 简化 Swing 应用程序开发的框架)演示了如何在一个 J2EE 环境外使用 Spring，还有文章下面要讨论的 Spring 的集成测试特性。依赖注入和相关功能太通用、太有价值以至无法被局限在一个 J2EE 或服务端环境中。

JDBC 抽象和数据存取异常层次

数据存取是 Spring 的另一个闪光点。

JDBC 提供了底层数据库的很好的抽象，但使用它的 API 十分痛苦。这些问题包括：

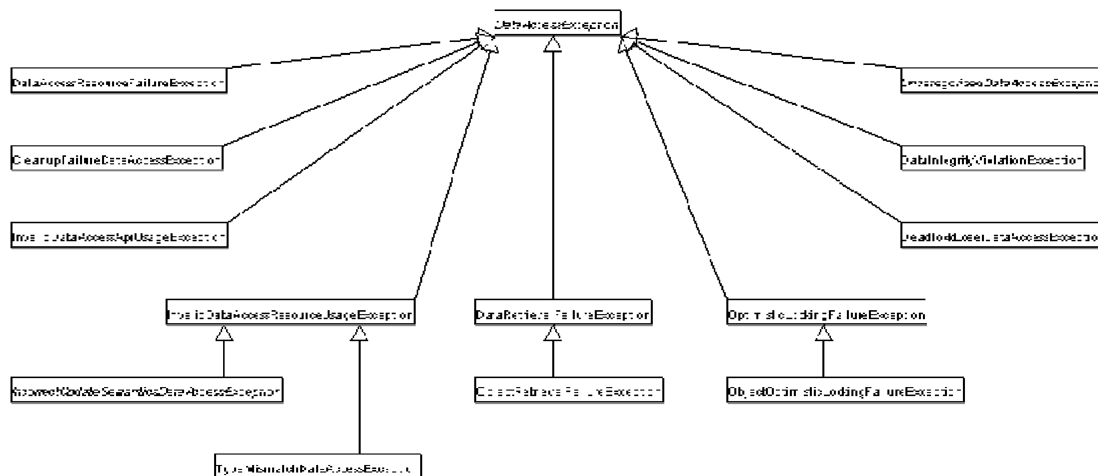
- 需要详细的异常处理来保证 ResultSet、Statement 和(最重要的) Connection 在使用后被关闭。这意味着要正确使用 JDBC 需要很多代码。这也是一个常见的错误的源头。连接泄漏可以让应用程序很快崩溃。
- SQLException 相对来说不能说明任何问题。JDBC 不提供一个异常的层次，而是用抛出 SQLException 来响应所有的错误。要找到是什么东西错了(例如，是死锁还是非法的 SQL)需要检查 SQLState 的值和错误码。这些值随着数据库的不同而不同。

Spring 通过两种方法解决这些问题：

- 提供 API 将冗长的易出错的异常处理从应用程序代码中移到框架中。框架会负责所有的异常处理；应用程序代码可以集中精力在写恰当的 SQL 和提取结果上。
- 为你的应用程序提供一个有意义的异常层次来代替 SQLException。当 Spring 第一次从数据源获得一个连接时，它会检查元数据来决定是什么数据库产品。再用这信息将 SQLExceptions 映射到它自己的从 org.springframework.dao.DataAccessException 继承下来的层次中的正确异常。这样一来你的代码可以和有意义的异常打交道，不用担心私有的 SQLState 和错误码。Spring 的数据存取异常不是 JDBC 特有的，所以你的 DAO 不用因为它们可能会抛出的异常而绑死在 JDBC 上。

下面的 UML 类图阐明了一部分数据存取异常的层次，展现了它的完善度。注意这里异常没有一个是 JDBC 特有的。有些 JDBC 特有的异常是这些异常的子类，但调用的代码一般是完全抽象于对 JDBC 的依赖的：如果你想用完全独立于 API 的 DAO 接口来隐藏你的持久策略，这是最基本的。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn



Spring 提供了两层 JDBC 抽象 API。第一层，在 org.springframework.jdbc.core 包中，用回调将控制权(并将相关的错误处理和连接获取与释放)从应用程序代码移到框架中。这是一种不同类型的控制反转，但与配置管理用的有一样的价值。

Spring 用相似的回调方法来处理其他几个包含获得和清除资源的特殊步骤的 API，例如 JDO(获得和释放一个 PersistenceManager)，事务管理(使用 JTA)和 JNDI。执行这些回调的 Spring 类叫模板(template)。

例如，Spring 的 JdbcTemplate 对象可以通过如下方法用来执行 SQL 查询和保存列表中的结果：

```
JdbcTemplate template = new JdbcTemplate(dataSource);
List names = template.query("SELECT USER.NAME FROM USER",
  new RowMapper() {
    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
      return rs.getString(1);
    }
  });
```

mapRow 回调方法会被 ResultSet 中的每一行调用。

注意回调方法内的应用程序代码可以自由抛出 SQLException：Spring 会捕获任何异常再在它自己的层次内重新抛出。应用程序开发者能选择捕获和处理哪个异常，如果有的话。

JdbcTemplate 提供了很多方法来支持不同的场景，包括已准备的语句和批处理更新。像运行 SQL 函数这样的简单任务可以像下面这样不用回调来完成。这个例子也示范了绑定变量的使用：

```
int youngUserCount = template.queryForInt("SELECT COUNT(0) FROM USER WHERE
USER.AGE < ?", new Object[] { new Integer(25) });
```

Spring 的 JDBC 抽象相对标准 JDBC 而言性能损失非常小，甚至在处理庞大结果集的时候也是如此。(在 2004 年的一个项目中，我们记录到一个金融项目每个事务执行 1200000 条插入操作。Spring JDBC 的开销是最小的，Spring 的使用方便了调整批处理大小和其他参数。)

更高层的 JDBC 抽象包含在 org.springframework.jdbc.object 包中。这是建立在核心 JDBC 回调功能上的，但是它提供了一个 API，其中的对 RDBMS 的操作(无论是查询、更新或者是存储过程)被做成了一个 Java 对象。这个 API 的灵感部分来自于 JDO 的查询 API，我发现它很直观很好用。

一个返回 User 对象的查询对象大概是这样的：

```
class UserQuery extends MappingSqlQuery {
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    public UserQuery(DataSource datasource) {
        super(datasource, "SELECT * FROM PUB_USER_ADDRESS WHERE USER_ID = ?");
        declareParameter(new SqlParameter(Types.NUMERIC));
        compile();
    }

    // 将一个记录集映射到一个 Java 对象
    protected Object mapRow(ResultSet rs, int rownum) throws SQLException {
        User user = new User();
        user.setId(rs.getLong("USER_ID"));
        user.setForename(rs.getString("FORENAME"));
        return user;
    }

    public User findUser(long id) {
        // 用超类的方法并进行强制类型转换
        return (User) findObject(id);
    }
}
```

这个类可以这样被使用：

```
User user = userQuery.findUser(25);
```

这样的对象经常是 DAO 的内部类。他们是线程安全的，除非子类做了些不寻常的事。

另一个 org.springframework.jdbc.object 包中的重要类是 StoredProcedure 类。Spring 通过一个带业务方法的 Java 类来代理一个存储过程。如果你喜欢，你能定义一个存储过程实现的接口，这意味你可以彻底把你的应用程序带从一个存储过程的依赖中解放出来。

Spring 的数据存储异常层次是基于未经检查(运行时)异常。在几个项目中使用了 Spring 之后我越来越确信这是正确的决定。

数据存取异常一般是不可恢复的。例如，如果我们不能连接到数据库，一个特定的业务对象就不能解决要处理的问题。一个潜在的异常是乐观锁冲突，但不是所有的应用程序使用乐观锁。强制编写代码去捕获不能很好处理的致命异常一般是不太好的。把它们抛给类似 Servlet 或 EJB 容器这样的高层去处理更合适。所有的 Spring 数据存取异常是 DataAccessException 的子类，所以如果我们想要选择去捕获所有的 Spring 数据存取异常，我们能轻松办到。

注意，如果我们想要从一个未经检查的数据存取异常中恢复过来，我们仍然可以做到。我们能便写只处理可恢复情况的代码。例如，如果我们认为只有一个乐观锁冲突是可恢复的，我们可以在一个 Spring 的 DAO 中编写如下代码：

```
try {
    // 工作
}
catch (OptimisticLockingFailureException ex) {
    // 我对这很感兴趣
}
```

如果 Spring 数据存取异常是经过检查的，我们需要如下代码。注意，随便什么情况我们都能这么写。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
try {
    // 工作
}
catch (OptimisticLockingFailureException ex) {
    // 我对这很感兴趣
}
catch (DataAccessException ex) {
    // 致命；仅再次抛出它
}
```

第一个例子的一个潜在缺陷(编译器不会强制处理潜在可恢复异常)在第二个例子中仍然存在。因为我们被强制去捕获基本异常(DataAccessException)，编译器不会强制对一个子类(OptimisticLockingFailureException)的检查。因此编译器可能会强制我们去写代码处理一个不可恢复的问题，但不会对强制我们处理的可恢复问题提供帮助。

Spring 对未经检查的数据存取异常的使用和许多(也许是大多数)成功的持久化框架是一致的。(确实，这部分受到了 JDO 的影响。)JDBC 是少数几个使用经检查的异常的数据存取 API。例如，TopLink 和 JDO 只使用未经检查的异常。Hibernate 在版本 3 中可以在经检查的和未经检查的异常间切换。

Spring JDBC 可以通过以下途径来帮助你：

- 你不再需要写 finally 块来使用 JDBC。
- 连接泄漏会成为过去。
- 总的来说你写的代码少了，并且那些代码清楚地集中在需要的 SQL 上。
- 你不再需要翻你的 RDBMS 文档来找那因错误的字段名而返回的晦涩的错误码。你的应用程序不依赖于特定的 RDBMS 错误处理代码。
- 无论是用何种持久化技术，你会发现可以很方便地实现 DAO 模式而不用让业务逻辑依赖于任何特定的数据存取 API。
- 你能在例如 BLOB 处理和调用存储过程这样的高级应用中获得更好的可移植性(与纯 JDBC 相比)，从而获益。

在实践中我们发现，所有这些都有助提高生产力和减少错误。我曾经讨厌写 JDBC 代码；现在我发现我可以关注我想要执行的 SQL，而不是随之而来的 JDBC 资源管理。

如果需要，Spring 的 JDBC 抽象能被单独使用——你可以不使用 Spring 的其他部分。

实体关系映射(O/R mapping, ORM)集成

当然你经常需要使用实体关系映射，而不是使用关系型数据访问。你的整体应用程序框架也必须支持这个。因而 Spring 继承了 Hibernate(版本 2 和 3)、JDO(版本 1 和 2)、TopLink 和其他 ORM 产品。他的数据访问架构允许与任何底层数据访问技术集成。Spring 和 Hibernate 是一个相当流行的组合。

为什么你要用一个 ORM 产品加上 Spring，而不是直接使用它呢？Spring 在以下方面增加了重要价值：

- 会话管理。Spring 提供了对诸如 Hibernate 或 Toplink 会话的有效、简单而且安全的处理。相关的单独使用 ORM 工具的代码通常需要使用同一个“Session”对象来达到有效且合适的事务处理。Spring 能使用一个声明性的 AOP 方法拦截器或使用显式的 Java 代码级的“模板”封装类来透明创建一个会话并将其绑定到当前线程。因而 Spring 解决了许多影响 ORM 技术用户的使用问题。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

- 资源管理。Spring 应用程序上下文能处理 Hibernate SessionFactory、JDBC 数据源和其他相关资源的定位和配置。这使得这些值易于管理和更改。
- 完整的事务管理。Spring 允许你将你的 ORM 代码封装为一个声明性的 AOP 方法拦截器或者一个显式的 Java 代码级的“模板”封装类。不管哪种情况，事务语义都为你处理了，并且在异常发生时也作了恰当的事务处理(回滚等)。正如我们后面讨论的，你也可以使用并切换不同事务管理器而不影响你的 ORM 相关代码，并从中获益。一个额外的好处是为了支持大多数 ORM 工具 JDBC 相关代码能完全事务性地与 ORM 代码集成。这对于处理 ORM 没有实现的功能时很有用。
- 如上所述的异常封装。Spring 能封装来自 ORM 层的异常，将它们从私有(可能是经过检查的)异常转换为一组抽象的运行时异常。这允许你可以不用烦人的样板化的 catch/throw 和异常声明仍能在恰当的层中处理大多数不可恢复的持久化异常。你仍能在任何需要的地方捕获和处理异常。请记住 JDBC 异常(包括数据库特有方言)也被转换为相同层次，这意味着你能在一致的编程模型内用 JDBC 执行一些操作。
- 避免和厂商绑定。ORM 解决方案有不同的性能和特性，没有能满足所有情况的完美解决方案。作为选择，你会发现某一个功能正好不匹配你用的 ORM 工具的一个实现。这让你意识到你的数据访问对象接口的特定工具实现对你的架构有影响。一旦你因为功能、性能或其他原因需要切换到另一个实现，现在使用了 Spring 让最终的切换变得更容易了。Spring 对你 ORM 工具的事务和异常的抽象，和它的 IoC 方法一起让你在映射器/DAO 对象实现数据访问功能间轻松切换，简单地实现将 ORM 特有代码控制在你应用程序的一个范围里而不牺牲你 ORM 工具的能力。和 Spring 一起发布的宠物医院范例程序演示了 Spring 通过提供使用 JDBC、Hibernate、TopLink 和 Apache OJB 实现的不同持久层而带来的可移植性方面的价值。
- 简化测试。Spring 的控制反转方法使得切换实现和诸如 Hibernate 会话工厂、数据源、事务管理器和映射器对象实现(如果需要)之类的资源位置变得简单了。这简化了隔离和每部分持久化相关代码的单独测试。

综上所述，Spring 让混用数据存取过程更简单了。尽管 ORM 在很多案例里赢得了颇有价值的生产力，无论一些 ORM 厂商怎么宣称，ORM 不是所有问题的解决方案。即使你混用各种持久化方法，即使你不用 JTA，Spring 仍能提供一致的架构和事务策略。

在 ORM 并不理想的地方，Spring 的简化 JDBC 并不是唯一选择：iBATIS SQL Maps 提供的“映射语句”也值得一看。它在保持自动从查询结果中创建映射对象的同时提供了对 SQL 的高层控制。Spring 集成了 SQL Maps。Spring 的宠物店范例程序演示了 iBATIS 集成。

事务管理

仅抽象一个数据访问 API 是不够的；我们还需要考虑事务管理。JTA 是当然的解决方案，但直接使用它的 API 是很笨重的，因而许多 J2EE 开发者曾觉得 EJB CMT 是事务管理的唯一明智的选择。Spring 改变了这一点。

Spring 提供了它自己的针对事务管理的抽象。Spring 使用它来提供：

- 通过一个类似 JdbcTemplate 的回调模板来实现可编程的事务管理，比起直接使用 JTA 容易许多。
- 类似 EJB CMT 的声明性事务管理，但不需要 EJB 容器。实际上，正如我们所见，Spring 的声明性事务管理能力是 EJB CMT 语义上的超集，有些独特的重要的好处。

Spring 的事务抽象的独特之处在于它不绑定于 JTA 或其他任何事务管理技术。Spring 使用事务策略概念，这减弱了底层事务基础部分(例如 JDBC)对应用程序代码的影响。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

为什么你需要关心这个？JTA 不是对所有事务管理问题的最好答案吗？如果你正在写一个只与一个数据库打交道的应用程序，你不需要 JTA 的复杂性。你不关心 XA 事务或两阶段提交。你甚至可以不需要一个提供这些东西的高端应用服务器。但另一方面，你不会愿意在和多个数据源打交道时重写你的代码。

假定你决定直接使用 JDBC 或 Hibernate 的事务来避免 JTA 的开销。一旦你需要与多个数据源打交道，你会不得不找出所有事务管理代码并用 JTA 事务来替换它们。这难道不具吸引力？这导致包括我在内的大多数 J2EE 开发者推荐只使用全局 JTA 事务，并用像 Tomcat 这样的简单 Web 容器来有效管理事务程序。不管怎样，使用 Spring 事务抽象，你只要重新配置 Spring 来使用 JTA，而不是 JDBC 或 Hibernate 的事务策略就可以了。这是一个配置的修改，不是代码变更。因此，Spring 让你的应用程序能随意缩放。

AOP

从 2003 年起在企业关注问题上(例如一般使用 EJB 来做的事务管理)使用 AOP 解决方案大家有了很大的兴趣。

Spring 的 AOP 支持的首要目标是为 POJO 提供 J2EE 支持。Spring AOP 能够在应用服务器之间移植，所以没有厂商绑定的风险。它可以工作在 web 容器或者是 EJB 容器中，而且已被成功应用于 WebLogic、Tomcat、JBoss、Resin、Jetty、Orion 和其他很多应用服务器及 web 容器上。

Spring AOP 支持方法拦截。所支持的关键 AOP 概念包括：

- 拦截(Interception)：自定义的行为能插入到任何接口和类的方法调用的前面或后面。这和 AspectJ 术语中的“around advice”相近。
- 引入(Introduction)：一个执行逻辑(advice)会导致一个对象实现额外的接口。这会引起继承混合。
- 静态和动态切入点：在程序执行过程中指定发生拦截的地方。静态切入点关注方法签名；动态关注点还能考虑被计算处的方法参数。切入点与拦截器分开定义，这使得一个标准拦截器可以被用在不同应用程序和代码上下文中。

Spring 支持有状态的(每个执行逻辑对象一个实例)和无状态的(所有执行逻辑用一个实例)拦截器。

Spring 不支持字段拦截。这是一个经过深思熟虑的设计决定。我总觉得字段拦截不符合封装原则。我更愿意认为 AOP 是 OOP 的一个补充而不是与它冲突。在 5 到 10 年后，我们在 AOP 的学习曲线上走得更远了很自然地会觉得应该在应用程序设计的桌面上给 AOP 留个位置。(那时像 AspectJ 这样的基于语言的解决方案会比今天更具吸引力。)

Spring 用动态代理(需要存在一个接口)或运行时 CGLIB 字节码生成(这使得类的代理成为可能)来实现 AOP。这两种方法能在任何应用服务器中运行，也能工作在独立环境中。

Spring 是第一个实现了 AOP 联盟接口(www.sourceforge.net/projects/aopalliance)的 AOP 框架。这些代表了不同 AOP 框架的拦截器也许可以协同工作。

Spring 集成了 AspectJ，提供了将 AspectJ 的方面无逢集成到 Spring 应用程序中的能力。从 Spring 1.1 开始已经可以用 Spring 的 IoC 容器依赖注入 AspectJ 的方面，就像任何 Java 类一样。因此 AspectJ 的方面可以依赖于任何 Spring 管理的对象。对即将到来的 AspectJ 版本 5 的集成更令人激动，基于注释驱动切入点，AspectJ 开始提供用 Spring 依赖注入任何 POJO 的能力。

因为 Spring 拦截的是实例级对象而不是类加载器级的，所以可以在同一个类的不同实例上使用不同的执行逻辑，或把未拦截的对象和已拦截的对象一起使用。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

也许 Spring AOP 的常见用途是用于声明性事务管理。这构建于前面讨论过的事务抽象之上，并且可以用任何 POJO 进行声明性事务管理。依靠事务策略，底层机制可以是 JTA、JDBC、Hibernate 或是其他任何提供事务管理的 API。

以下是与 EJB CMT 的主要不同：

- 事务管理可以应用于任何 POJO。我们建议业务对象实现接口，这只是一个好的编程习惯，不是由框架所强制的。
- 通过使用 Spring 的事务 API 在一个事务性 POJO 中实现可编程的回滚。我们为此提供了静态方法，使用 ThreadLocal 变量，所以你不必传播一个类似 EJBContext 这样的上下文对象来保证回滚。
- 你能定义声明式的回滚规则。应用程序开发者常想要事务能够在遇到任何异常时回滚，但 EJB 在遇到未捕获的应用程序异常时不会自动回滚(仅仅在未检查的异常和其他 Throwable 异常还有“系统”异常时才回滚)。Spring 的事务管理允许你声明式地指定哪个异常和子类是应该引起自动回滚的。默认的行为和 EJB 一样，但你可以指定经检查异常和未经检查异常一样回滚。这在最小化可编程回滚需要上有很大好处，而这可编程回滚也建立了对 Spring 事务 API 的依赖(就像 EJB 可编程回滚建立对 EJBContext 的依赖一样)。
- 底层 Spring 事务抽象支持保存点(如果底层事务基本结构支持的话)，所以 Spring 的声明性事务管理能支持嵌套事务，和 EJB CMT 特有的传播模式(Spring 支持和 EJB 一样的语义)。因而，举例来说，如果你在 Oracle 上执行 JDBC 操作，你能通过 Spring 使用声明性嵌套事务。
- 事务管理没有绑定 JTA。正如前面解释的，Spring 事务管理能和不同事务策略合作。

你还可以使用 Spring AOP 实现应用程序特有方面。是否要这样做取决于你对 AOP 概念的理解程度，而不是 Spring 的能力，但这会是很有用的。我们见到的成功例子包括：

- 在安全检查要求的复杂度超过了标准 J2EE 安全基础结构的地方自定义安全拦截器。(当然，在开始你自己的安全结构前，你应该看看 Acegi Security for Spring，一个强大、灵活的用 AOP 和 Spring 整合的安全框架，这也反映了 Spring 架构上的方法。)
- 在开发中使用调式和记录方面。
- 实现在一个地方使用一致的异常处理策略的方面。
- 发送邮件给管理员或用户，警告不正常情况的拦截器。

应用程序特有方面能成为消除许多方法对样板代码需要的有效途径。

Spring AOP 透明地与 Spring BeanFactory 概念集成。代码从一个 Spring 的 BeanFactory 中获得一个对象的时候不用知道它是不是被拦截。和任何对象一样，契约是在对象实现的接口上定义的。

下面的 XML 片断描述了如何定义一个 AOP 代理：

```
<bean id="myTest"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <value>org.springframework.beans.ITestBean</value>
    </property>
    <property name="interceptorNames">
        <list>
            <value>txInterceptor</value>
            <value>target</value>
        </list>
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    </property>
</bean>
```

注意，尽管引用的或 BeanFactory 的 getBean()方法返回的 bean 类型依赖于代理接口，bean 定义里的类总是 AOP 框架的 ProxyFactoryBean。(支持多代理方法。) ProxyFactoryBean 的“interceptorNames”属性接受一个 String 列表。(必须用 bean 名称而不是 bean 引用，因为如果代理是一个原型那么新的有状态拦截器需要被创建。)列表里的名字可以是拦截器或者切入点(拦截器和何时该被应用的信息)。上面列表里的“target”值自动建立一个“调用拦截器”封装目标对象。它是工厂里的一个实现了代理接口的 bean 的名称。例子里的 myTest bean 能像工厂里的其他 bean 那样被使用。例如其他对象可以通过<ref>元素引用它，并且这些引用可以用 Spring IoC 设置。

有很多方法能更简单地建立代理，如果你不需要 AOP 框架的全部功能(例如不用 XML 而用 Java 5.0 的注释来驱动事务性代理，或者用一段简单的 XML 实现对一个 Spring 工厂里的许多 bean 应用一致的代理策略)。

还可以不用 BeanFactory 而用编程方法来构建 AOP 代理，虽然这种方法很少用：

```
TestBean target = new TestBean();
DebugInterceptor di = new DebugInterceptor();
MyInterceptor mi = new MyInterceptor();
ProxyFactory factory = new ProxyFactory(target);
factory.addInterceptor(0, di);
factory.addInterceptor(1, mi);
// 一个用来封装目标的“调用拦截器”被自动添加
ITestBean tb = (ITestBean) factory.getProxy();
```

我们相信最好把应用程序的装配从 Java 代码里拿出来，AOP 也不例外。

使用 AOP 作为 EJB(版本 2 或以上版本)的替代物来进行企业服务是的重要性正在加大。Spring 很成功地展现了这个主张的价值。

MVC web 框架

Spring 包括一个强大且高度可配置的 MVC web 框架。

Spring 的 MVC 模型尽管不是源自 Struts，但和 Struts 的很相似。一个 Spring 的 Controller 和 Struts 的 Action 很像，它们都是多线程服务对象，只有一个实例代表所有客户端执行。但是，我们相信 Spring MVC 比起 Struts 有一些显著的优点。例如：

- Spring 在控制器、JavaBean 模型和视图间提供清晰的划分。
- Spring 的 MVC 非常灵活。不像 Struts 那样强迫你的 Action 和 Form 对象有具体的继承(你只能用 Java 的继承)，Spring 的 MVC 完全基于接口。此外，几乎 Spring MVC 框架的每个部分都能通过插入你自己的接口来配置。当然，我们也提供了简单的类作为一个可选的实现。
- Spring，像 WebWork 一样，提供拦截器和控制器，这使得提取处理多个请求的公共行为变得容易了。
- Spring MVC 是真正视图无关的。如果你不愿意你不会被强制使用 JSP; 你能用 Velocity、XLST 或其他视图技术。如果你想用自定义的视图机制(比如你自己的模板语言)，你可以轻松实现 Spring 的 View 接口来整合它。
- Spring 的 Controller 和其他对象一样是通过 IoC 来配置的。这让它们易于测试，和其他 Spring 管理的对象漂亮地集成在一起。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

- 因为没有强迫使用具体的继承和显式地依赖于调度器 Servlet 的控制器，Spring MVC 的 web 层比起 Struts 的 web 层更易于测试。
- Web 层变成了业务对象层上的薄薄一层。这鼓励了好的习惯。Struts 和其他专门的 web 框架让你自己实现你的业务对象；Spring 为你的应用程序提供了一个完整的框架。

和 Struts 1.1 及更高版本一样，你能根据你的需要在 Spring MVC 应用程序中拥有多个调度器 Servlet。

下面的范例演示了一个简单的 Spring Controller 如何访问同一个应用程序上下文中定一个业务对象。这个控制器在它的 handleRequest()方法中执行了一个 Google 查询：

```
public class GoogleSearchController implements Controller {
    private IGoogleSearchPort google;
    private String googleKey;

    public void setGoogle(IGoogleSearchPort google) {
        this.google = google;
    }

    public void setGoogleKey(String googleKey) {
        this.googleKey = googleKey;
    }

    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        String query = request.getParameter("query");
        GoogleSearchResult result =
        // Google 属性定义忽略

        // 使用 Google 业务对象
        google.doGoogleSearch(this.googleKey, query,start, maxResults, filter, restrict,
            safeSearch, lr, ie, oe);

        return new ModelAndView("googleResults", "result", result);
    }
}
```

这个代码的原型 IGoogleSearchPort 是一个 GLUE web 服务代理，由一个 Spring FactoryBean 返回。然而，Spring IoC 将这个控制器从底层 web 服务库中隔离出来。也可以用一个简单的 Java 对象、测试桩、模拟对象或者用像下面要讨论的 EJB 代理来实现这个接口。这个控制器没有包含资源查找；除了支持它的 web 交互的必要代码外没有别的东西了。

Spring 也提供了对数据绑定、表单、向导和更复杂的工作流的支持。马上要发表的这个系列文章中的一篇文章会详细讨论 Spring MVC。

如果你的需求真的很复杂，你应该考虑 Spring Web Flow，一个提供比传统 web MVC 框架更高层次 web 工作流抽象的强大的框架，它的架构师 Keith Donald 会在最近的 TSS 文章上讨论它的。

一个不错的 Spring MVC 框架的入门材料是 Thomas Risberg 的 Spring MVC 指南 (http://www.springframework.org/docs/MVC-step-by-step/Spring-MVC-step-by-step.html)。也可

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

以看《Web MVC with the Spring Framework》
(http://www.springframework.org/docs/web_mvc.html)。

如果对你喜欢的 MVC 框架情有独钟，Spring 的分层结构允许你使用 MVC 层以外的 Spring 的其他部分。我们有把 Spring 作为中间层管理和数据访问但在 web 层使用 Struts、WebWork、Tapestry 或 JSF 的用户。

实现 EJB

如果你选择使用 EJB，Spring 可以令你从 EJB 实现和客户端对 EJB 的访问中获益。

将业务逻辑重构进 EJB 外观后的 POJO 中是一个被广泛认同的最佳实践。(和其它实践相比，这使得对业务逻辑的单元测试更加容易，因为 EJB 极其依赖容器且很难独立测试。) Spring 为会话 bean 和消息驱动 bean 提供了好用的超类，通过自动加载包含在 EJB Jar 文件中的基于一个 XML 文档的 BeanFactory 来方便实现这点。

这意味着一个无状态的会话 EJB 可以这样来获得并使用一个协作者：

```
import org.springframework.ejb.support.AbstractStatelessSessionBean;

public class MyEJB extends AbstractStatelessSessionBean implements MyBusinessInterface {
	private MyPOJO myPOJO;

	protected void onEjbCreate() {
		this.myPOJO = getBeanFactory().getBean("myPOJO");
	}

	public void myBusinessMethod() {
		this.myPOJO.invokeMethod();
	}
}
```

假定 MyPOJO 是一个接口，实现类(和任何它要求的诸如原始属性和进一步的协作者之类的配置)隐藏在 XML bean 工厂的定义中。

我们通过一个在标准 ejb-jar.xml 部署描述符中的名为 ejb/BeanFactoryPath 的环境变量定义告诉 Spring 到什么地方去加载 XML 文档，就像这样：

```
<session>
	<ejb-name>myComponent</ejb-name>
	<local-home>com.test.ejb.myEjbBeanLocalHome</local-home>
	<local>com.mycom.MyComponentLocal</local>
	<ejb-class>com.mycom.MyComponentEJB</ejb-class>
	<session-type>Stateless</session-type>
	<transaction-type>Container</transaction-type>

	<env-entry>
		<env-entry-name>ejb/BeanFactoryPath</env-entry-name>
		<env-entry-type>java.lang.String</env-entry-type>
		<env-entry-value>/myComponent-ejb-beans.xml</env-entry-value>
	</env-entry>
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
</session>
```

myComponent-ejb-beans.xml 文件会被从 classpath 中加载：在本例中，是在 EJB Jar 文件的根部。每个 EJB 能指定它自己的 XML 文件，所以这个机制能在每个 EJB Jar 文件中多次使用。

Spring 超类实现了 EJB 的 setSessionContext()和 ejbCreate()一类的生命周期方法，让应用程序开发者只要选择性地实现 Spring 的 onEjbCreate()方法就可以了。

当 EJB 3.0 蓝图公布后，我们会对使用 Spring IoC 容器在那个环境中提供更丰富的依赖注入语义提供支持。我们也同样会将 JSR-220 实体/关系映射 API 作为一个被支持的数据访问 API 整合进来。

使用 EJB

Spring 除了让实现 EJB 更容易外，还让它的使用变得简单了。许多 EJB 应用程序使用服务定位器和业务代理模式。这比在客户代码中遍布 JNDI 查找好得多，但它们的一般实现存在明显的缺陷。例如：

- 依赖于服务定位器或业务代理单例的典型 EJB 调用代码很难测试。
- 在没有一个业务代理的服务定位器模式中，应用程序代码仍然要调用一个 EJB home 的 create()方法，并处理可能的异常。因而它仍然绑定了 EJB API 和 EJB 编程模型的复杂性。
- 实现业务代理模式通常会导致明显的代码重复，我们不得不写很多方法来简单地调用同一个 EJB 方法。

为这些和其他一些原因，传统的 EJB 访问(就像 Sun Adventure Builder 和 OTN J2EE 虚拟大卖场里演示的)会降低生产力并带来显著的复杂度。

Spring 通过引入少量代码业务代理在这方面先行一步。有了 Spring 你不再需要在手工编写的业务代理中写另外的服务定位器、JNDI 查找或重复方法除非你加入了真正的价值。

举例来说，想象我们有一个使用本地 EJB 的 web 控制器。我们会遵循最佳实践并使用 EJB 业务方法接口模式，所以 EJB 本地接口要扩展一个非 EJB 特有的业务方法接口。(这样做的一个主要原因是确保本地接口和 bean 实现类的自动同步。)我们称这个业务方法接口为 MyComponent。当然我们也需要实现本地 home 接口并提供一个实现了 SessionBean 和 MyComponent 业务方法接口的 bean 实现类。

有了 Spring EJB 访问，我们为挂接我们的 web 层控制器和 EJB 实现所需的唯一的 Java 编码就是暴露控制器上的一个 MyComponent 的 setter 方法。这会像一个实例变量那样保存引用：

```
private MyComponent myComponent;

public void setMyComponent(MyComponent myComponent) {
    this.myComponent = myComponent;
}
```

随后我们就能在任何业务方法中使用这个实例变量了。

Spring 通过 XML bean 定义项来自动完成剩下的工作。LocalStatelessSessionProxyFactoryBean 是一个能被任何 EJB 使用的通用工厂 bean。它创建的对象能自动被 Spring 转换为 MyComponent 类型。

```
<bean id="myComponent"
        class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
```


PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

```
    <property name="jndiName" value="myComponent" />
    <property name="businessInterface" value="com.mycom.MyComponent" />
</bean>

<bean id="myController" class = "com.mycom.myController">
    <property name="myComponent" ref="myComponent" />
</bean>
```

幕后发生了很多魔法般的事情，出于 Spring AOP 框架的谦虚，你没有被强迫使用 AOP 概念来享受这结果。“myComponent” bean 定义建立了一个实现了业务方法接口的 EJB 代理。EJB 本地 home 在启动时被缓存，所以一般只需要一次 JNDI 查找。(也有对失败时重试的支持，所以一次 EJB 重部署不会导致客户端失败。)EJB 每次被调用时，代理调用本地 EJB 的 create()方法并调用 EJB 的相应业务方法。

myController bean 定义将控制类的 myController 属性设置到这个代理。

这个 EJB 访问机制对应用程序代码进行了大量简化：

- Web 层代码没有了对使用 EJB 的依赖。如果我们想要用一个 POJO、模拟对象或其他测试桩来代替这个 EJB 引用，我们可以简单地改变 myComponent bean 定义而不用修改一行 Java 代码。
- 我们没有必要写一行 JNDI 查找代码或其他 EJB 组装代码来作为我们应用程序的一部分。

我们还能通过相似的 org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean 工厂 bean 将相同的方法运用于远程 EJB。但是，要把一个远程 EJB 业务方法接口的 RemoteExceptions 隐藏起来却是很棘手的。(如果你希望提供一个匹配 EJB 远程接口但方法签名中没有“throws RemoteException”语句的客户端服务接口，Spring 确实能让你办到这点。)

测试

正如你所注意到的，我和其他 Spring 的开发者都是全面单元测试的忠实拥护者。我们相信框架经过彻底的单元测试是很重要的，而且框架设计的一个主要目的应该是使构建于框架之上的应用程序应该是易于进行单元测试的。

Spring 自己有一个出色的单元测试包。我们发现这个项目的测试优先带来的好处是实实在在的。例如，它使得一个国际化分布式开发团队工作极富效率，而且用户们评论 CVS 快照往往很稳定，用起来很安全。

我们相信构建在 Spring 上的应用程序测试很方便，有以下原因：

- IoC 易于进行单元测试。
- 应用程序不包含那些一般很难测试的直接使用例如 JNDI 之类的 J2EE 服务的代码。
- Spring 的 bean 工厂或上下文能在容器外建立。

在容器外建立一个 Spring 的 bean 工厂为开发过程提供了有趣的选择权。在几个使用 Spring 的 web 应用程序项目中，工作始于定义业务接口和在一个 web 容器外进行它们的实现的集成测试。只有在业务功能彻底完成后，再在上面加薄薄一层提供 web 接口。

从 Spring 1.1 开始，Spring 提供了对在部署环紧外进行集成测试的强大且独特的支持。这并不是有意要作为单元测试或针对部署环境的测试的代替品。然而，这可以显著提高生产力。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

org.springframework.test 包提供了使用一个 Spring 容器而不用依赖于一个应用服务器或其他部署环境进行集成测试的很有价值的超类。这样的测试可以在 Junit 里运行(甚至是在一个 IDE 里)而不用特殊的部署步骤。他们运行起来会比单元测试稍慢，但比 Cactus 测试或靠部署到一个应用服务器的测试要快得多。通常很可能在几秒钟而不是几分钟或几小时内要运行好几百个针对一个开发数据库的测试(一般不是一个嵌入式数据库，而是生产中用的数据库产品)。这样的测试能快速检验你 Spring 上下文和使用 JDBC 或 ORM 工具的数据访问的正确装配，比如 SQL 语句的正确性。举例来说，你可以测试你的 DAO 实现类。

org.springframework.test 包中实现的功能包括：

- 通过依赖注入实现 JUnit 测试用例移植的能力。这使测试时重用 Spring XML 配置和消除针对测试的自定义设置代码成为可能。
- 在测试用例之间缓存容器配置的能力，这在用到那些比如 JDBC 连接池或 Hibernate 的 SessionFactory 之类的初始化很慢的资源的地方大大提高了的性能。
- 默认情况下为每个测试方法建立事务并在测试结束时回滚的基础结构。这将允许测试进行各种各样的数据存取而不用担心影响其它测试的环境。从我在几个复杂项目中使用这一功能的的经验看来，用这样一个基于回滚的方法带来的生产力和速度的提升是很明显的。

谁在使用 Spring？

有很多应用程序产品使用 Spring。用户包括投资和零售的银行组织、知名互联网公司、全球性顾问公司、学院机构、政府机关、防卫设备承包商、几家航空公司和科研组织(包括 CERN)。

很多用户使用全部的 Spring，但有些使用单独的组件。例如，一些用户从使用我们的 JDBC 或其它数据存取功能开始。

发展历程

自从这篇文章的第一版在 2003 年 10 月发表以来，Spring 经历了 1.0 最终发布版(2004 年 3 月)到 1.1 版(2004 年 9 月)到 1.2 最终版(2005 年 5 月)的过程。我们相信这样一个哲学"早发布，勤发布"，所以维护性发布和小的改进一般 4 到 6 周发布一次。

从那时起的改进包括：

- 引入一个支持包括 RMI 和多种 web 服务协议在内的多协议远程框架。
- 支持方法注入和其他的 IoC 容器增强，例如管理从静态或实例工厂方法的调用中获得的对象的能力。
- 集成更多数据访问技术，在最近的 1.2 版中包括 TopLink、 Hibernate 版本 3。
- 支持用 Java 5.0 注释配置的声明性事务管理，不需要用 XML 元数据来识别事务性方法(1.2)。
- 支持 Spring 所管理对象的 JMX 管理(1.2)。
- 集成了 Jasper 报告、Quartz 计划和 AspectJ。
- 将 JSF 作为一种 web 层技术集成进来。

我们打算继续快速的革新和增强。下一个主要版本会是 1.3。计划中的增强包括：

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

- XML 配置增强(计划在 1.3 版内)，这将允许用自定义 XML 标签通过定义一个或多个单一的、合法的标签来扩展基本的 Spring 配置格式。这不仅可能显著简化典型配置并减少配置错误，还会成为基于 Spring 的第三方产品的开发者的理想选择。
- 把 Spring Web Flow 集成到 Spring 的核心中(计划在 1.3 版内)。
- 支持运行时应用程序动态重配置。
- 支持用 Java 外的语言编写的应用程序对象，比如 Groovy、Jython 或其他运行于 Java 平台上的脚本语言。这些对象能得益于 Spring IoC 容器的完整服务和在脚本改变时动态重新装载而不影响由 IoC 容器引用到它们的对象。

作为一个敏捷项目，Spring 主要由用户需求驱动。所以我们不会开发没有任用的功能，我们也会仔细倾听来自我们用户社群的声音。

Spring Modules 是一个由 Interface 21 的 Rob Harrop 领导的相关项目，它将 Spring 平台扩展到那些 Spring 核心没有必要完全支持的，但仍然对很多用户有价值的领域。这个项目也作为一个孵化器，所以其中的一些功能会最终集成到 Spring 核心中。Spring Modules 目前包括如下领域：与 Lucene 搜索引擎和 OSWorkflow 工作流引擎的集成、一个基于 AOP 的声明性缓存解决方案、与 Commons Validator 框架的集成。

有趣的是，尽管这篇文章的第一版发表在 Spring 1.0 最终版发布的六个月前，几乎所有的示范代码和配置仍然能不经改动地在今天的 1.2 版上运行。我们为自己在向下兼容方面的骄人记录感到自豪。这也展现了依赖注入和 AOP 实现非侵入性 API 的能力，还表现了我们尽心尽力为社区提供一个能够运行重大应用程序的稳定框架的严谨。

总结

Spring 是一个解决了许多常见 J2EE 问题的强大框架。许多 Spring 的功能也可以被运用于很多超越经典 J2EE 的 Java 环境中。

Spring 提供了一种以一致方法管理业务对象的方法，并鼓励好的编程习惯，例如针对接口而不是类编程。Spring 的架构基础是一个使用 JavaBean 属性的控制反转容器。可是，这只是 Spring 全貌的一部分：Spring 在将 IoC 容器作为所有架构层的完整解决方案的基本构建块方面是独一无二的。

Spring 提供了一个独特的数据访问抽象，包括一个大大改善生产力并降低错误可能的简单而有效的 JDBC 框架。Spring 的数据访问架构还集成了 TopLink、Hibernate、JDO 和其他实体/关系映射解决方案。

Spring 提供了唯一的事务管理抽象，这能够在类似 JTA 或 JDBC 这样的不同底层事务技术上使用一致的编程模型。

Spring 提供了一个用标准 Java 写的 AOP 框架，它提供了声明性事务管理和其他用于 POJO 的企业服务或者(如果你希望)也能用于实现你自己的自定义方面。这个框架强大到足以使很多应用程序在享受传统的 EJB 相关的关键服务的同时放弃 EJB 的复杂性。

Spring 还提供了一个可整合到整个 IoC 容器中的强大且灵活的 MVC web 框架。

更多信息

需要更多的关于 Spring 的信息请参阅：

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn

- Interface21 提供的一个核心 Spring 培训课程——http://www.springframework.com/training。
- 《Expert One-on-One J2EE Design and Development》(Rod Johnson, Wrox, 2002)。尽管 Spring 在此书出版后有了很明显的改进，但它仍然是理解 Spring 动机的好地方。
- 《J2EE without EJB》(Rod Johnson 与 Juergen Hoeller 合著 Wrox, 2004)。《Expert One-on-One J2EE Design and Development》的后续作品，讨论 Spring 和它的轻量级容器架构的基本原理。
- 《Spring 参考手册》。Spring 1.2 的打印版本又超过 240 页。Spring 还带了几个展示最佳实践并可用作你自己的应用程序模板的范例。
- 《Pro Spring》：由核心开发者 Rob Harrop 深入讨论 Spring。
- 《Spring: A Developer's Notebook》：由 Bruce Tate 和 Justin Gehtland 所著的入门。
- Spring 框架主页：http://www.springframework.org/，这里包括 Javadoc 和几个教程。
- Sourceforge 上的论坛和下载。
- Spring 开发者邮件列表。

我们为对待论坛和邮件列表中问题的认真态度和出色的回复率感到自豪。我们欢迎您早日加入我们的社区。

关于作者

Rod Johnson 拥有差不多十年作为 Java 开发者和架构师的经验，并从 J2EE 平台出现后就在其上进行开发。他是畅销书《Expert One-on-One J2EE Design and Development》(Wrox, 2002)和《J2EE without EJB》(Wrox, 2004 与 Juergen Hoeller 合著)的作者，也参与过其它 J2EE 著作的编写。Rod 参与了两个 Java 标准委员会并经常在大会发言。他是 Interface21 的 CEO，这是一家国际化咨询公司，领导 Spring 框架开发，提供 Spring 框架和 J2EE 方面的专业服务。

PDF 文件使用 "pdfFactory Pro" 试用版本创建 www.fineprint.cn