作者博客：http://jinnianshilongnian.iteye.com/ 欢迎访问

目　　录

第一章　Spring 概述

1.1　概述

1.1.1　Spring 是什么

Spring 是一个开源的轻量级 Java SE（Java 标准版本）/Java EE（Java 企业版本）开发应用框架，其目的是用于简化企业级应用程序开发。应用程序是由一组相互协作的对象组成。而在传统应用程序开发中，一个完整的应用是由一组相互协作的对象组成。所以开发一个应用除了要开发业务逻辑之外，最多的是关注如何使这些对象协作来完成所需功能，而且要低耦合、高内聚。业务逻辑开发是不可避免的，那如果有个框架出来帮我们来创建对象及管理这些对象之间的依赖关系。可能有人说了，比如“抽象工厂、工厂方法设计模式”不也可以帮我们创建对象，“生成器模式”帮我们处理对象间的依赖关系，不也能完成这些功能吗？可是这些又需要我们创建另一些工厂类、生成器类，我们又要而外管理这些类，增加了我们的负担，如果能有种通过配置方式来创建对象，管理对象之间依赖关系，我们不需要通过工厂和生成器来创建及管理对象之间的依赖关系，这样我们是不是减少了许多工作，加速了开发，能节省出很多时间来干其他事。Spring 框架刚出来时主要就是来完成这个功能。

Spring 框架除了帮我们管理对象及其依赖关系，还提供像通用日志记录、性能统计、安全控制、异常处理等面向切面的能力，还能帮我管理最头疼的数据库事务，本身提供了一套简单的 JDBC 访问实现，提供与第三方数据访问框架集成（如 Hibernate、JPA），与各种 Java EE 技术整合（如 Java Mail、任务调度等等），提供一套自己的 web 层框架 Spring MVC、而且还能非常简单的与第三方 web 框架集成。从这里我们可以认为 Spring 是一个超级粘合平台，除了自己提供功能外，还提供粘合其他技术和框架的能力，从而使我们可以更自由的选择到底使用什么技术进行开发。而且不管是 JAVA SE（C/S 架构）应用程序还是 JAVA EE(B/S 架构)应用程序都可以使用这个平台进行开发。让我们来深入看一下 Spring 到底能帮我们做些什么？

1.1.2 Spring 能帮我们做什么

Spring 除了不能帮我们写业务逻辑，其余的几乎什么都能帮助我们简化开发：
一、传统程序开发，创建对象及组装对象间依赖关系由我们在程序内部进行控制，这样会加大各个对象间的耦合，如果我们要修改对象间的依赖关系就必须修改源代码，重新编译、

部署；而如果采用 Spring，则由 Spring 根据配置文件来进行创建及组装对象间依赖关系，只需要改配置文件即可，无需重新编译。所以，**Spring 能帮我们根据配置文件创建及组装对象之间的依赖关系。**

二、当我们要进行一些日志记录、权限控制、性能统计等时，在传统应用程序当中我们可能在需要的对象或方法中进行，而且比如权限控制、性能统计大部分是重复的，这样代码中就存在大量重复代码，即使有人说我把通用部分提取出来，那必然存在调用还是存在重复，像性能统计我们可能只是在必要时才进行，在诊断完毕后要删除这些代码；还有日志记录，比如记录一些方法访问日志、数据访问日志等等，这些都会渗透到各个要访问方法中；还有权限控制，必须在方法执行开始进行审核，想想这些是多么可怕而且是多么无聊的工作。如果采用 Spring，这些日志记录、权限控制、性能统计从业务逻辑中分离出来，通过 Spring 支持的面向切面编程，在需要这些功能的地方动态添加这些功能，无需渗透到各个需要的方法或对象中；有人可能说了，我们可以使用“代理设计模式”或“包装器设计模式”，你可以使用这些，但还是需要通过编程方式来创建代理对象，还是要耦合这些代理对象，而采用 Spring 面向切面编程能提供一种更好的方式来完成上述功能，一般通过配置方式，而且不需要在现有代码中添加任何额外代码，现有代码专注业务逻辑。所以，**Spring 面向切面编程能帮助我们无耦合的实现日志记录，性能统计，安全控制。**

三、在传统应用程序当中，我们如何来完成数据库事务管理？需要一系列“获取连接，执行 SQL，提交或回滚事务，关闭连接”，而且还要保证在最后一定要关闭连接，多么可怕的事情，而且也很无聊；如果采用 Spring，我们只需获取连接，执行 SQL，其他的都交给 Spring 来管理了，简单吧。所以，**Spring 能非常简单的帮我们管理数据库事务。**

四、Spring 还提供了与第三方数据访问框架（如 Hibernate、JPA）无缝集成，而且自己也提供了一套 JDBC 访问模板，来方便数据库访问。

五、Spring 还提供与第三方 Web（如 Struts、JSF）框架无缝集成，而且自己也提供了一套 Spring MVC 框架，来方便 web 层搭建。

六、Spring 能方便的与 Java EE（如 Java Mail、任务调度）整合，与更多技术整合（比如缓存框架）。

Spring 能帮我们做这么多事情，提供这么多功能和与那么多主流技术整合，而且是帮我们做了开发中比较头疼和困难的事情，那可能有人会问，难道只有 Spring 这一个框架，没有其他选择？当然有，比如 EJB 需要依赖应用服务器、开发效率低、在开发中小型项目是宰鸡拿牛刀，虽然发展到现在 EJB 比较好用了，但还是比较笨重还需要依赖应用服务器等。那为何需要使用 Spring，而不是其他框架呢？让我们接着往下看。

1.1.3 为何需要 Spring

一 首先阐述几个概念

1、应用程序：是能完成我们所需要功能的成品，比如购物网站、OA 系统。

2、框架：是能完成一定功能的半成品，比如我们可以使用框架进行购物网站开发；框架做一部分功能，我们自己做一部分功能，这样应用程序就创建出来了。而且框架规定了你

在开发应用程序时的整体架构，提供了一些基础功能，还规定了类和对象的如何创建、如何协作等，从而简化我们开发，让我们专注于业务逻辑开发。

3、非侵入式设计：从框架角度可以这样理解，无需继承框架提供的类，这种设计就可以看作是非侵入式设计，如果继承了这些框架类，就是侵入设计，如果以后想更换框架之前写过的代码几乎无法重用，如果非侵入式设计则之前写过的代码仍然可以继续使用。

4、轻量级及重量级：轻量级是相对于重量级而言的，轻量级一般就是非入侵性的、所依赖的东西非常少、资源占用非常少、部署简单等等，其实就是比较容易使用，而重量级正好相反。

5、POJO：POJO（Plain Old Java Objects）简单的 Java 对象，它可以包含业务逻辑或持久化逻辑，但不担当任何特殊角色且不继承或不实现任何其它 Java 框架的类或接口。

6、容器：在日常生活中容器就是一种盛放东西的器具，从程序设计角度看就是装对象的的对象，因为存在放入、拿出等操作，所以容器还要管理对象的生命周期。

7、控制反转：即 Inversion of Control，缩写为 IoC，控制反转还有一个名字叫做依赖注入（Dependency Injection），就是由容器控制程序之间的关系，而非传统实现中，由程序代码直接操控。

8、Bean：一般指容器管理对象，在 Spring 中指 Spring IoC 容器管理对象。

二 为什么需要 Spring 及 Spring 的优点

- **非常轻量级的容器：**以集中的、自动化的方式进行应用程序对象创建和装配，负责对象创建和装配，管理对象生命周期，能组合成复杂的应用程序。Spring 容器是非侵入式的（不需要依赖任何 Spring 特定类），而且完全采用 POJOs 进行开发，使应用程序更容易测试、更容易管理。而且核心 JAR 包非常小，Spring 3.0.5 不到 1M，而且不需要依赖任何应用服务器，可以部署在任何环境（Java SE 或 Java EE）。
- **AOP：**AOP 是 Aspect Oriented Programming 的缩写，意思是面向切面编程，提供从另一个角度来考虑程序结构以完善面向对象编程（相对于 OOP），即可以通过在编译期间、装载期间或运行期间实现在不修改源代码的情况下给程序动态添加功能的一种技术。通俗点说就是把可重用的功能提取出来，然后将这些通用功能在合适的时候织入到应用程序中；比如安全，日记记录，这些都是通用的功能，我们可以把它们提取出来，然后在程序执行的合适地方织入这些代码并执行它们，从而完成需要的功能并复用了这些功能。
- **简单的数据库事务管理：**在使用数据库的应用程序当中，自己管理数据库事务是一项很让人头疼的事，而且很容易出现错误，Spring 支持可插入的事务管理支持，而且无需 JEE 环境支持，通过 Spring 管理事务可以把我们从事务管理中解放出来来专注业务逻辑。
- **JDBC 抽象及 ORM 框架支持：**Spring 使 JDBC 更加容易使用；提供 DAO（数据访问对象）支持，非常方便集成第三方 ORM 框架，比如 Hibernate 等；并且完全支持 Spring 事务和使用 Spring 提供的一致的异常体系。
- **灵活的 Web 层支持：**Spring 本身提供一套非常强大的 MVC 框架，而且可以非常容易的与第三方 MVC 框架集成，比如 Struts 等。

- **简化各种技术集成：**提供对 Java Mail、任务调度、JMX、JMS、JNDI、EJB、动态语言、远程访问、Web Service 等的集成。

Spring 能帮助我们简化应用程序开发，帮助我们创建和组装对象，为我们管理事务，简单的 MVC 框架，可以把 Spring 看作是一个超级粘合平台，能把很多技术整合在一起，形成一个整体，使系统结构更优良、性能更出众，从而加速我们程序开发，有如上优点，我们没有理由不考虑使用它。

1.1.4 如何学好 Spring

要学好 Spring，首先要明确 Spring 是个什么东西，能帮我们做些什么事情，知道了这些然后做个简单的例子，这样就基本知道怎么使用 Spring 了。Spring 核心是 IoC 容器，所以一定要透彻理解什么是 IoC 容器，以及如何配置及使用容器，其他所有技术都是基于容器实现的；理解好 IoC 后，接下来是面向切面编程，首先还是明确概念，基本配置，最后是实现原理，接下来就是数据库事务管理，其实 Spring 管理事务是通过面向切面编程实现的，所以基础很重要，IoC 容器和面向切面编程搞定后，其余都是基于这俩东西的实现，学起来就更加轻松了。要学好 Spring 不能急，一定要把基础打牢，基础牢固了，这就是磨刀不误砍柴工。

1.2 Spring 基础

1.2.1 Spring 架构图

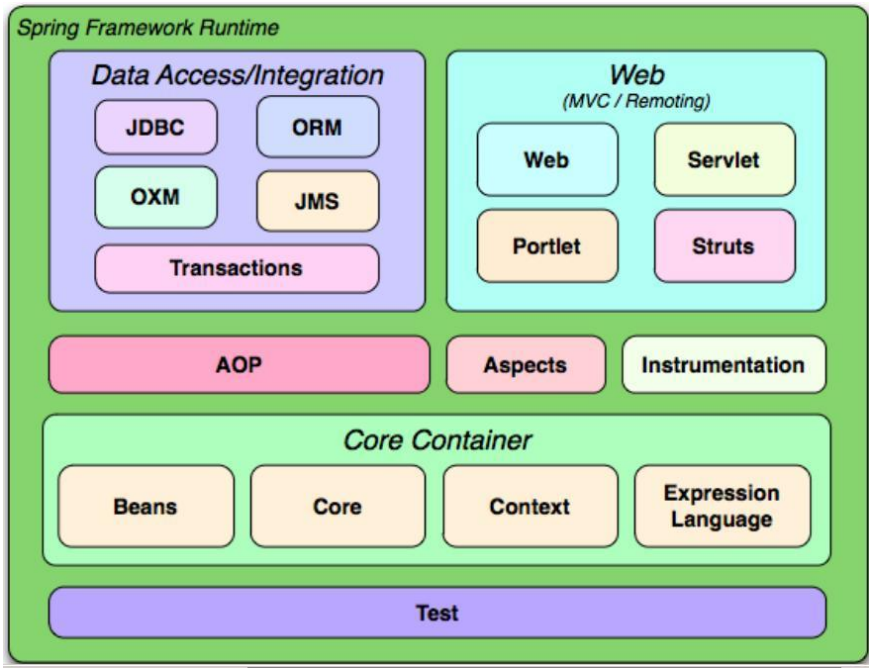


图 1-1 Spring 架构图

核心容器：包括 Core、Beans、Context、EL 模块。

- **Core 模块：**封装了框架依赖的最底层部分，包括资源访问、类型转换及一些常用工具类。
- **Beans 模块：**提供了框架的基础部分，包括反转控制和依赖注入。其中 Bean Factory 是容器核心，本质是“工厂设计模式”的实现，而且无需编程实现“单例设计模式”，单例完全由容器控制，而且提倡面向接口编程，而非面向实现编程；所有应用程序对象及对象间关系由框架管理，从而真正把你从程序逻辑中把维护对象之间的依赖关系提取出来，所有这些依赖关系都由 BeanFactory 来维护。
- **Context 模块：**以 Core 和 Beans 为基础，集成 Beans 模块功能并添加资源绑定、数据验证、国际化、Java EE 支持、容器生命周期、事件传播等；核心接口是 ApplicationContext。
- **EL 模块：**提供强大的表达式语言支持，支持访问和修改属性值，方法调用，支持访问及修改数组、容器和索引器，命名变量，支持算数和逻辑运算，支持从 Spring 容器获取 Bean，它也支持列表投影、选择和一般的列表聚合等。

AOP、Aspects 模块：

- **AOP 模块：**Spring AOP 模块提供了符合 *AOP Alliance* 规范的面向方面的编程（aspect-oriented programming）实现，提供比如日志记录、权限控制、性能统计等通用功能和业务逻辑分离的技术，并且能动态的把这些功能添加到需要的代码中；这样各专其职，降低业务逻辑和通用功能的耦合。
- **Aspects 模块：**提供了对 AspectJ 的集成，AspectJ 提供了比 Spring ASP 更强大的功能。

数据访问/集成模块：该模块包括了 JDBC、ORM、OXM、JMS 和事务管理。

- 事务模块：该模块用于 Spring 管理事务，只要是 Spring 管理对象都能得到 Spring 管理事务的好处，无需在代码中进行事务控制了，而且支持编程和声明性的事物管理。
- JDBC 模块：提供了一个 JBDC 的样例模板，使用这些模板能消除传统冗长的 JDBC 编码还有必须的事务控制，而且能享受到 Spring 管理事务的好处。
- ORM 模块：提供与流行的“对象-关系”映射框架的无缝集成，包括 Hibernate、JPA、Ibatiss 等。而且可以使用 Spring 事务管理，无需额外控制事务。
- OXM 模块：提供了一个对 Object/XML 映射实现，将 java 对象映射成 XML 数据，或者将 XML 数据映射成 java 对象，Object/XML 映射实现包括 JAXB、Castor、XMLBeans 和 XStream。
- JMS 模块：用于 JMS(Java Messaging Service)，提供一套 “消息生产者、消息消费者”模板用于更加简单的使用 JMS，JMS 用于用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

Web/Remoting 模块：Web/Remoting 模块包含了 Web、Web-Servlet、Web-Struts、Web-Porlet 模块。

- Web 模块：提供了基础的 web 功能。例如多文件上传、集成 IoC 容器、远程过程

访问（RMI、Hessian、Burlap）以及 Web Service 支持，并提供一个 RestTemplate 类来提供方便的 Restful services 访问。

- Web-Servlet 模块：提供了一个 Spring MVC Web 框架实现。Spring MVC 框架提供了基于注解的请求资源注入、更简单的数据绑定、数据验证等及一套非常易用的 JSP 标签，完全无缝与 Spring 其他技术协作。
- Web-Struts 模块：提供了与 Struts 无缝集成，Struts1.x 和 Struts2.x 都支持

Test 模块： Spring 支持 Junit 和 TestNG 测试框架，而且还额外提供了一些基于 Spring 的测试功能，比如在测试 Web 框架时，模拟 Http 请求的功能。

1.2.2 典型应用场景

Spring 可以应用到许多场景，从最简单的标准 Java SE 程序到企业级应用程序都能使用 Spring 来构建。以下介绍几个比较流行的应用场景：

- **典型 Web 应用程序应用场景：**

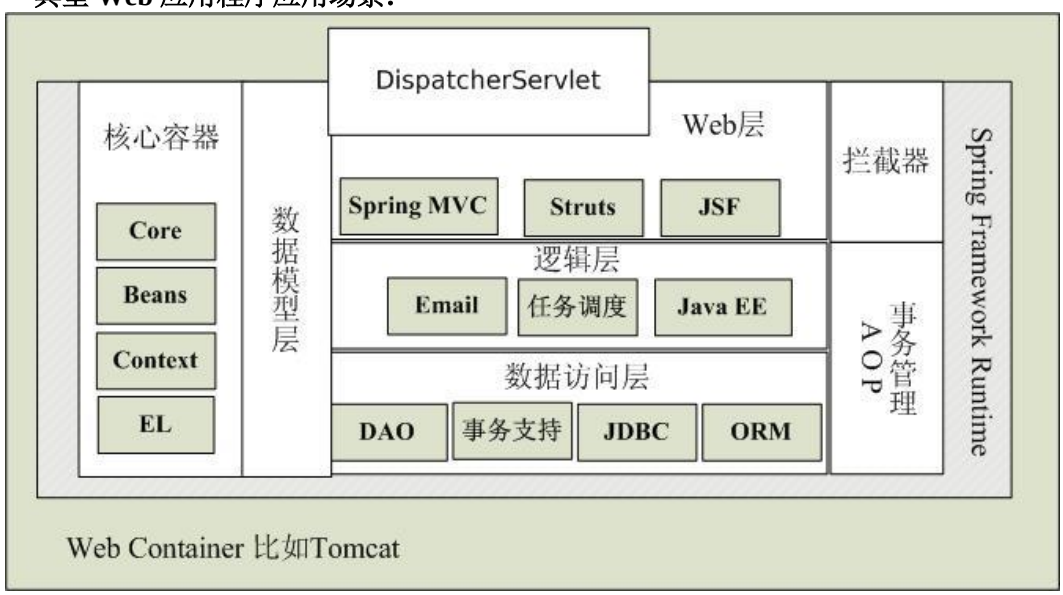


图 1-2 web 应用程序应用场景

在 Web 应用程序应用场景中，典型的三层架构：数据模型层实现域对象；数据访问层实现数据访问；逻辑层实现业务逻辑；web 层提供页面展示；所有这些层组件都由 Spring 进行管理，享受到 Spring 事务管理、AOP 等好处，而且请求唯一入口就是 DispachterServlet，它通过把请求映射为相应 web 层组件来实现相应请求功能。

- **远程访问应用场景：**

  Spring 能非常方便的提供暴露 RMI 服务，远程访问服务如 Hessian、Burlap 等，实现非常简单只需通过在 Spring 中配置相应的地址及需要暴露的服务即可轻松实现，后边会有介绍；

- **EJB 应用场景：**

  Spring 也可以与 EJB 轻松集成，后边会详细介绍。

第二章　IoC

2.1　IoC 基础

2.1.1　IoC 是什么

Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。在 Java 开发中，Ioc 意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。如何理解好 Ioc 呢？理解好 Ioc 的关键是要明确“谁控制谁，控制什么，为何是反转（有反转就应该有正转了），哪些方面反转了”，那我们来深入分析一下：

- **谁控制谁，控制什么：**传统 Java SE 程序设计，我们直接在对象内部通过 new 进行创建对象，是程序主动去创建依赖对象；而 IoC 是有专门一个容器来创建这些对象，即由 Ioc 容器来控制对象的创建；谁控制谁？当然是 IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。
- **为何是反转，哪些方面反转了：**有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。

用图例说明一下，传统程序设计如图 2-1，都是主动去创建相关对象然后再组合起来：

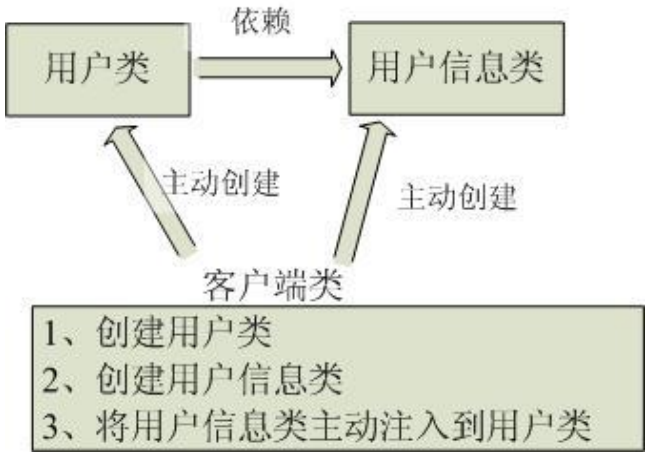


图 2-1 传统应用程序示意图

当有了 IoC/DI 的容器后，在客户端类中不再主动去创建这些对象了，如图 2-2 所示:

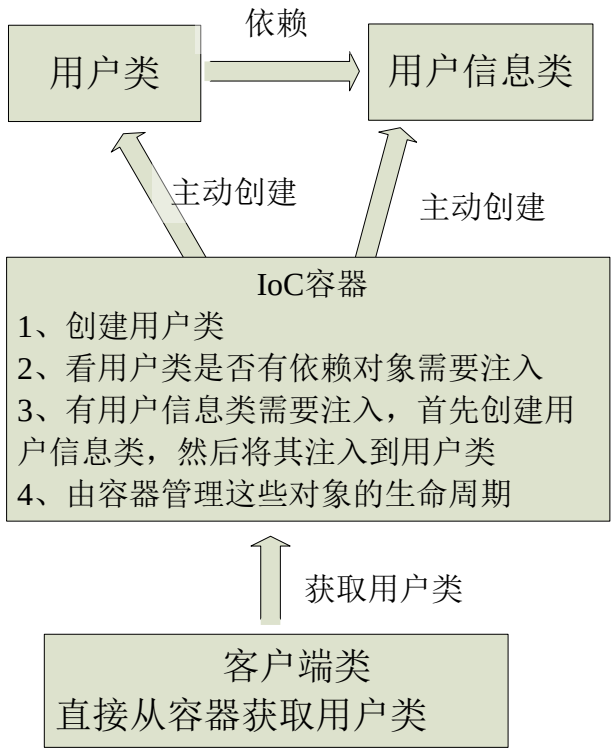


图 2-2 有 IoC/DI 容器后程序结构示意图

1.1.2 IoC 能做什么

IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了 IoC 容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

其实 IoC 对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在 IoC/DI 思想中，应用程序就变成被动的了，被动的等待 IoC 容器来创建并注入它所需要的资源了。

IoC 很好的体现了面向对象设计法则之一——好莱坞法则：“别找我们，我们找你”；即由 IoC 容器帮对象找相应的依赖对象并注入，而不是由对象主动去找。

2.1.3 IoC 和 DI

DI—Dependency Injection，即“依赖注入”：是组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标

需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

理解 DI 的关键是：“谁依赖谁，为什么需要依赖，谁注入谁，注入了什么”，那我们来深入分析一下：

- **谁依赖于谁：**当然是某个容器管理对象依赖于 IoC 容器；“被注入对象的对象”依赖于“依赖对象”；
- **为什么需要依赖：**容器管理对象需要 IoC 容器来提供对象需要的外部资源；
- **谁注入谁：**很明显是 IoC 容器注入某个对象，也就是注入“依赖对象”；
- **注入了什么：**就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）。

IoC 和 DI 由什么关系呢？其实它们是同一个概念的不同角度描述，由于控制反转概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护对象关系），所以 2004 年大师级人物 Martin Fowler 又给出了一个新的名字：“依赖注入”，相对 IoC 而言，**“依赖注入”明确描述了“被注入对象依赖 IoC 容器配置依赖对象”。**

> 注：如果想要更加深入的了解 IoC 和 DI，请参考大师级人物 Martin Fowler 的一篇经典文章《Inversion of Control Containers and the Dependency Injection pattern》，原文地址：http://www.martinfowler.com/articles/injection.html。

2.2 IoC 容器基本原理

2.2.1 IoC 容器的概念

IoC 容器就是具有依赖注入功能的容器，IoC 容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。应用程序无需直接在代码中 new 相关的对象，应用程序由 IoC 容器进行组装。在 Spring 中 BeanFactory 是 IoC 容器的实际代表者。

Spring IoC 容器如何知道哪些是它管理的对象呢？这就需要配置文件，Spring IoC 容器通过读取配置文件中的配置元数据，通过元数据对应用中的各个对象进行实例化及装配。一般使用基于 xml 配置文件进行配置元数据，而且 Spring 与配置文件完全解耦的，可以使用其他任何可能的方式进行配置元数据，比如注解、基于 java 文件的、基于属性文件的配置都可以。

那 Spring IoC 容器管理的对象叫什么呢？

2.2.2 Bean 的概念

由 IoC 容器管理的那些组成你应用程序的对象我们就叫它 Bean， Bean 就是由 Spring *容器*初始化、装配及管理的对象，除此之外，bean 就与应用程序中的其他对象没有什么区

别了。那 IoC 怎样确定如何实例化 Bean、管理 Bean 之间的依赖关系以及管理 Bean 呢？这就需要配置元数据，在 Spring 中由 BeanDefinition 代表，后边会详细介绍，配置元数据指定如何实例化 Bean、如何组装 Bean 等。概念知道的差不多了，让我们来做个简单的例子。

2.2.3 Hello World

一、配置环境：

- **JDK 安装：**安装最新的 JDK，至少需要 Java 1.5 及以上环境；
- **开发工具：**SpringSource Tool Suite，简称 STS，是个基于 Eclipse 的开发环境，用以构建 Spring 应用，其最新版开始支持 Spring 3.0 及 OSGi 开发工具，但由于其太庞大，很多功能不是我们所必需的所以我们选择 Eclipse+ SpringSource Tool 插件进行 Spring 应用开发；到 eclipse 官网下载最新的 Eclipse，注意我们使用的是 Eclipse IDE for Java EE Developers（eclipse-jee-helios-SR1）；

安装插件：启动 Eclipse，选择 Help->Install New Software，如图 2-3 所示

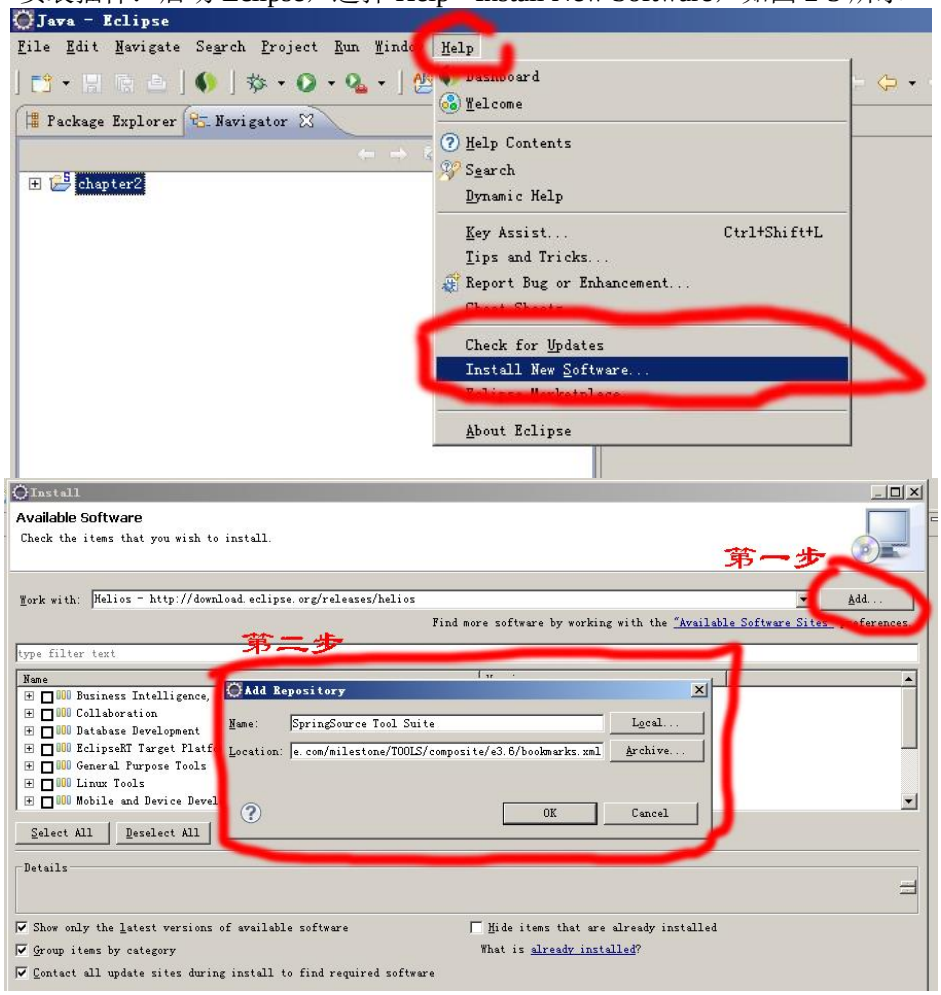


图 2-3 安装

2、首先安装 SpringSource Tool Suite 插件依赖，如图 2-4:

Name 为：SpringSource Tool Suite Dependencies

Location 为：http://dist.springsource.com/release/TOOLS/composite/e3.6

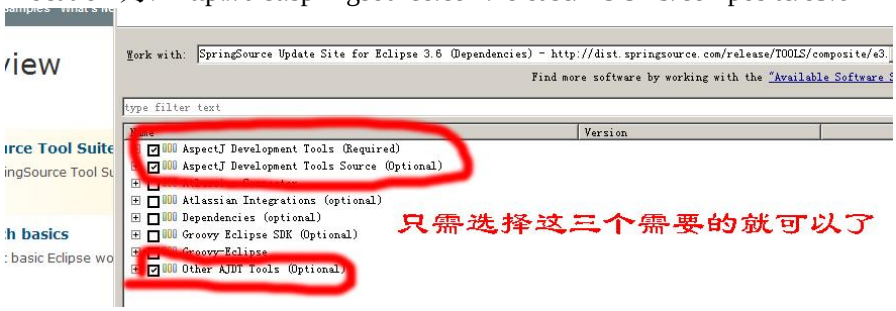


图 2-4 安装

3、安装 SpringSource Tool Suite 插件，只需安装如图 2-5 所选中的就可以：

Name 为：SpringSource Tool Suite

Location 为：http://dist.springsource.com/release/TOOLS/update/e3.6

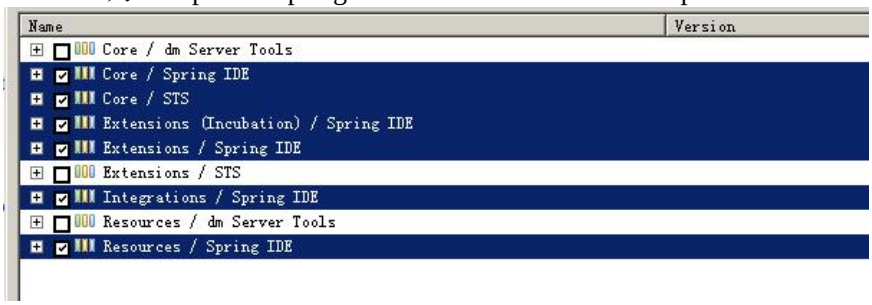


图 2-4 安装

4、安装完毕，开始项目搭建吧。

- **Spring 依赖：**本书使用 spring-framework-3.0.5.RELEASE
  - ✧ spring-framework-3.0.5.RELEASE-with-docs.zip 表示此压缩包带有文档的；
  - ✧ spring-framework-3.0.5.RELEASE-dependencies.zip 表示此压缩包中是 spring 的依赖 jar 包，所以需要什么依赖从这里找就好了；
  - ✧ 下载地址：http://www.springsource.org/download

二、开始 Spring Hello World 之旅

1、准备需要的 jar 包

核心 jar 包：从下载的 spring-framework-3.0.5.RELEASE-with-docs.zip 中 dist 目录查找如下 jar 包

org.springframework.asm-3.0.5.RELEASE.jar
org.springframework.core-3.0.5.RELEASE.jar
org.springframework.beans-3.0.5.RELEASE.jar
org.springframework.context-3.0.5.RELEASE.jar
org.springframework.expression-3.0.5.RELEASE.jar

依赖的 jar 包：从下载的 spring-framework-3.0.5.RELEASE-dependencies.zip 中查找如下依赖 jar 包

```
com.springsource.org.apache.log4j-1.2.15.jar
com.springsource.org.apache.commons.logging-1.1.1.jar
com.springsource.org.apache.commons.collections-3.2.1.jar
```

2、创建标准 Java 工程：

（1）选择“window”—> “Show View” —>“Package Explorer”，使用包结构视图；

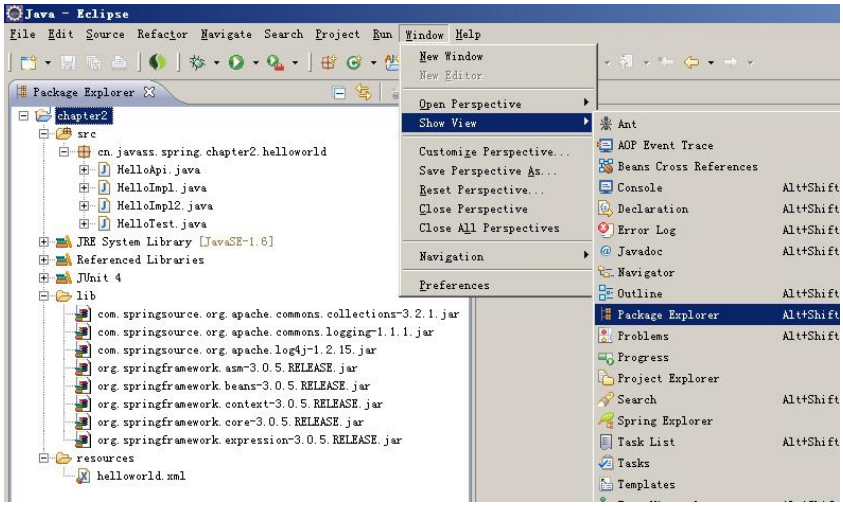


图 2-5 包结构视图

（2）创建标准 Java 项目，选择“File”—>“New”—>“Other”；然后在弹出来的对话框中选择“Java Project”创建标准 Java 项目；

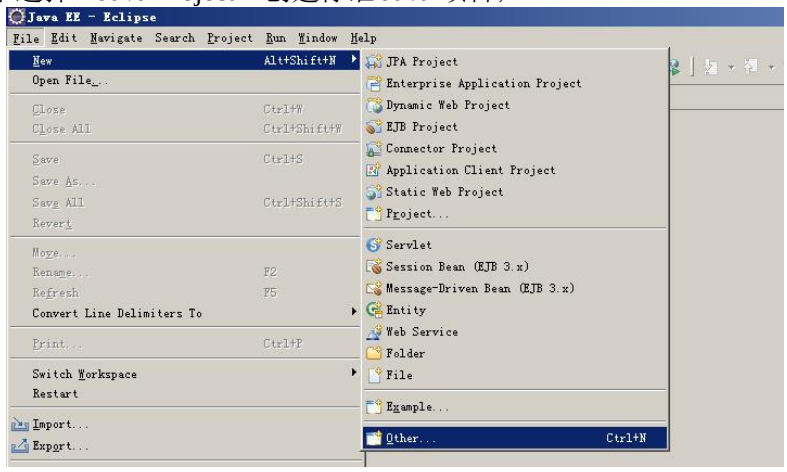


图 2-6 创建 Java 项目

图 2-7 创建 Java 项目

图 2-8 创建 Java 项目

（3）配置项目依赖库文件，右击项目选择“Properties”；然后在弹出的对话框中点击“Add JARS”在弹出的对话框中选择“lib”目录下的 jar 包；然后再点击“Add Library”，然后在弹出的对话框中选择“Junit”，选择“Junit4”；

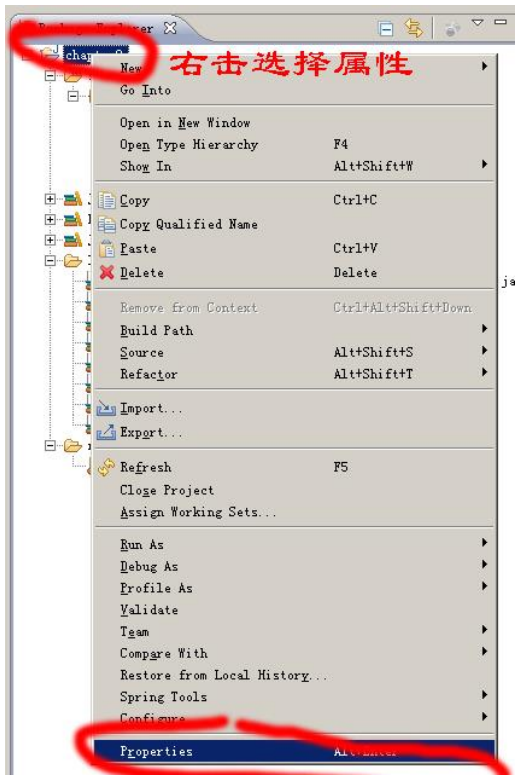


图 2-9 配置项目依赖库文件

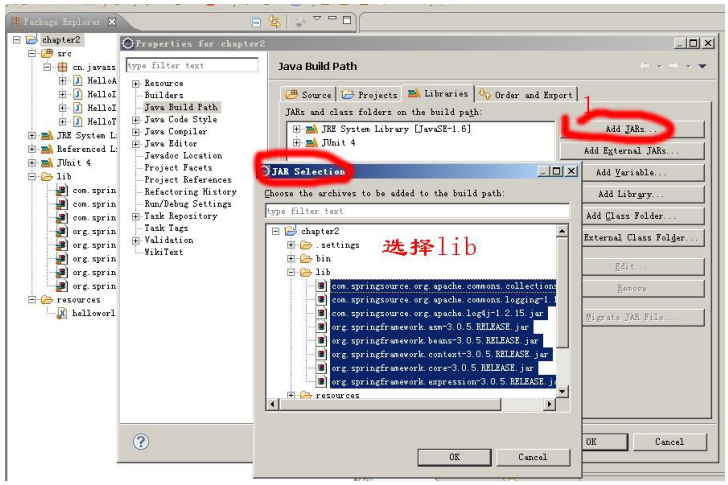


图 2-10 配置项目依赖库文件

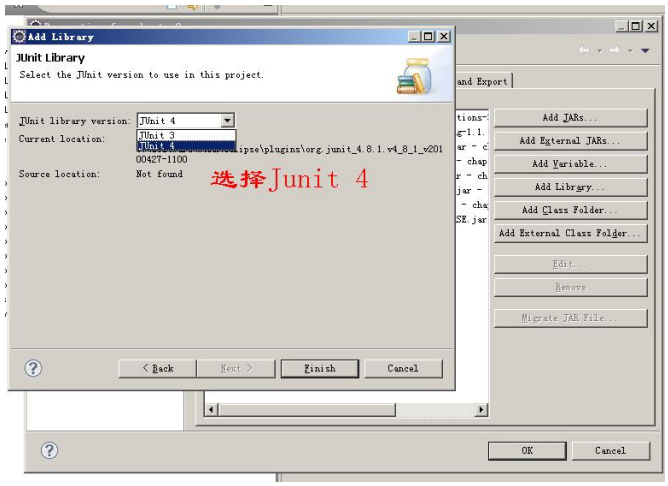


图 2-11 配置项目依赖库文件

（4）项目目录结构如下图所示，其中“src”用于存放 java 文件；“lib”用于存放 jar 文件；“resources”用于存放配置文件；

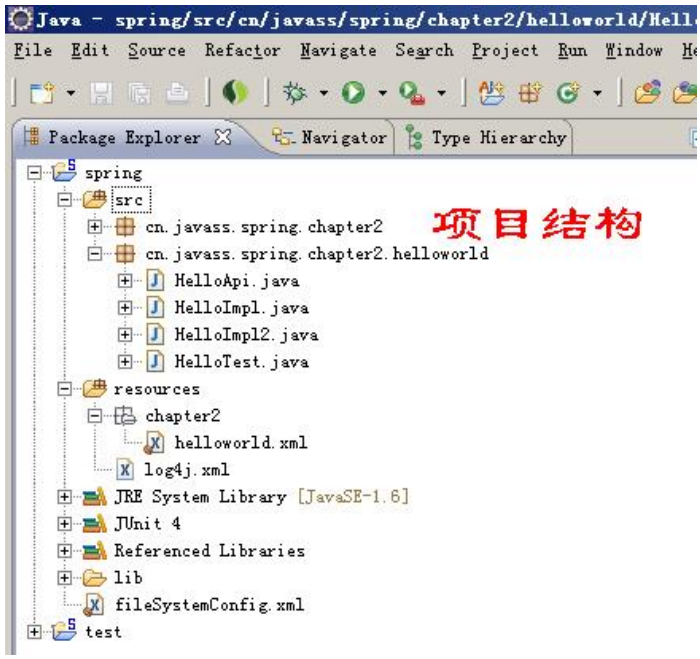


图 2-12 项目目录结构

3、项目搭建好了，让我们来开发接口，此处我们只需实现打印“Hello World!”，所以我们定义一个“sayHello”接口，代码如下：

```
package cn.javass.spring.chapter2.helloworld;
public interface HelloApi {
    public void sayHello();
}
```

4、接口开发好了，让我们来通过实现接口来完成打印“Hello World!”功能：

```
package cn.javass.spring.chapter2.helloworld;
public class HelloImpl implements HelloApi {
        @Override
        public void sayHello() {
            System.out.println("Hello World!");
        }
    }
```

5、接口和实现都开发好了，那如何使用 Spring IoC 容器来管理它们呢？这就需要配置文件，让 IoC 容器知道要管理哪些对象。让我们来看下配置文件 chapter2/helloworld.xml（放到 resources 目录下）：

```
<?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:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
   <!-- id 表示你这个组件的名字，class表示组件类 -->
   <bean id="hello" class="cn.javass.spring.chapter2.helloworld.HelloImpl"></bean>
</beans>
```

6、现在万一具备，那如何获取 IoC 容器并完成我们需要的功能呢？首先应该实例化一个 IoC 容器，然后从容器中获取需要的对象，然后调用接口完成我们需要的功能，代码示例如下：

```
package cn.javass.spring.chapter2.helloworld;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class HelloTest {
    @Test
    public void testHelloWorld() {
        //1、读取配置文件实例化一个IoC容器
```

```
        ApplicationContext context =
                                new ClassPathXmlApplicationContext("helloworld.xml");
        //2、从容器中获取Bean，注意此处完全“面向接口编程，而不是面向实现”
        HelloApi helloApi = context.getBean("hello", HelloApi.class);
        //3、执行业务逻辑
        helloApi.sayHello();
    }
`
```

7、自此一个完整的 Spring Hello World 已完成，是不是很简单，让我们深入理解下容器和 Bean 吧。

2.2.4 详解 IoC 容器

在 Spring Ioc 容器的代表就是 org.springframework.beans 包中的 BeanFactory 接口，BeanFactory 接口提供了 IoC 容器最基本功能；而 org.springframework.context 包下的 ApplicationContext 接口扩展了 BeanFactory，还提供了与 Spring AOP 集成、国际化处理、事件传播及提供不同层次的 context 实现 (如针对 web 应用的 WebApplicationContext)。简单说， BeanFactory 提供了 IoC 容器最基本功能，而 ApplicationContext 则增加了更多支持企业级功能支持。ApplicationContext 完全继承 BeanFactory，因而 BeanFactory 所具有的语义也适用于 ApplicationContext。

容器实现一览：

- **XmlBeanFactory**：BeanFactory 实现，提供基本的 IoC 容器功能，可以从 classpath 或文件系统等获取资源；

（1）
```
File file = new File("fileSystemConfig.xml");
Resource resource = new FileSystemResource(file);
BeanFactory beanFactory = new XmlBeanFactory(resource);
```
（2）
```
Resource resource = new ClassPathResource("classpath.xml");
BeanFactory beanFactory = new XmlBeanFactory(resource);
```

- **ClassPathXmlApplicationContext**：ApplicationContext 实现，从 classpath 获取配置文件；

```
BeanFactory beanFactory =
        new ClassPathXmlApplicationContext("classpath.xml");
```

- **FileSystemXmlApplicationContext**：ApplicationContext 实现，从文件系统获取配置文件。

```
BeanFactory beanFactory =
        new FileSystemXmlApplicationContext("fileSystemConfig.xml");
```

具体代码请参考 cn.javass.spring.chapter2.InstantiatingContainerTest.java。

ApplicationContext 接口获取 Bean 方法简介：

• Object getBean(String name) 根据名称返回一个 Bean，客户端需要自己进行类型转换；

• T getBean(String name, Class<T> requiredType) 根据名称和指定的类型返回一个 Bean，客户端无需自己进行类型转换，如果类型转换失败，容器抛出异常；

• T getBean(Class<T> requiredType) 根据指定的类型返回一个 Bean，客户端无需自己进行类型转换，如果没有或有多于一个 Bean 存在容器将抛出异常；

• Map<String, T> getBeansOfType(Class<T> type) 根据指定的类型返回一个键值为名字和值为 Bean 对象的 Map，如果没有 Bean 对象存在则返回空的 Map。

让我们来看下 IoC 容器到底是如何工作。在此我们以 xml 配置方式来分析一下：

一、准备配置文件：就像前边 Hello World 配置文件一样，在配置文件中声明 Bean 定义也就是为 Bean 配置元数据。

二、由 IoC 容器进行解析元数据： IoC 容器的 Bean Reader 读取并解析配置文件，根据定义生成 BeanDefinition 配置元数据对象，IoC 容器根据 BeanDefinition 进行实例化、配置及组装 Bean。

三、实例化 IoC 容器：由客户端实例化容器，获取需要的 Bean。

整个过程是不是很简单，执行过程如图 2-5，其实 IoC 容器很容易使用，主要是如何进行 Bean 定义。下一章我们详细介绍定义 Bean。

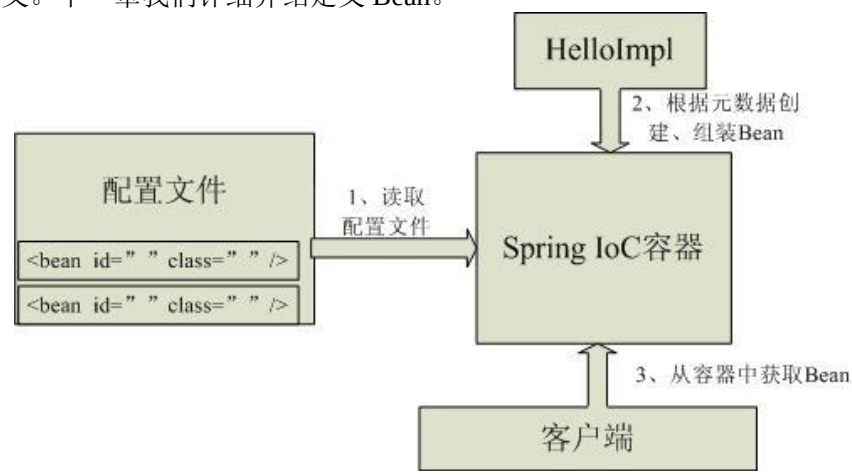


图 2-5 Spring Ioc 容器

2.2.5 小结

除了测试程序的代码外，也就是程序入口，所有代码都没有出现 Spring 任何组件，而且所有我们写的代码没有实现框架拥有的接口，因而能非常容易的替换掉 Spring，是不是非入侵。

客户端代码完全面向接口编程，无需知道实现类，可以通过修改配置文件来更换接口实现，客户端代码不需要任何修改。是不是低耦合。

如果在开发初期没有真正的实现，我们可以模拟一个实现来测试，不耦合代码，是不是很方便测试。

Bean 之间几乎没有依赖关系，是不是很容易重用。

2.3 IoC 的配置使用

2.3.1 XML 配置的结构

一般配置文件结构如下：

```
<beans>
    <import resource=”resource1.xml”/>
    <bean id=”bean1” class=””></bean>
    <bean id=”bean2” class=””></bean>
    <bean name=”bean2” class=””></bean>
    <alias alias="bean3" name="bean2"/>
    <import resource=”resource2.xml”/>
</beans>
```

1、<bean>标签主要用来进行 Bean 定义；

2、alias 用于定义 Bean 别名的；

3、import 用于导入其他配置文件的 Bean 定义，这是为了加载多个配置文件，当然也可以把这些配置文件构造为一个数组（new String[] {“config1.xml”, config2.xml}）传给 ApplicationContext 实现进行加载多个配置文件，那一个更适合由用户决定；这两种方式都是通过调用 Bean Definition Reader 读取 Bean 定义，内部实现没有任何区别。<import>标签可以放在<beans>下的任何位置，没有顺序关系。

2.3.2 Bean 的配置

Spring IoC 容器目的就是管理 Bean，这些 Bean 将根据配置文件中的 Bean 定义进行创建，而 Bean 定义在容器内部由 BeanDefinition 对象表示，该定义主要包含以下信息：

- 全限定类名（FQN）：用于定义 Bean 的实现类;
- Bean 行为定义：这些定义了 Bean 在容器中的行为；包括作用域（单例、原型创建）、是否惰性初始化及生命周期等；
- Bean 创建方式定义：说明是通过构造器还是工厂方法创建 Bean；
- Bean 之间关系定义：即对其他 bean 的引用，也就是依赖关系定义，这些引用 bean 也可以称之为同事 bean 或依赖 bean，也就是依赖注入。

Bean 定义只有“全限定类名”在当使用构造器或静态工厂方法进行实例化 bean 时是必须的，其他都是可选的定义。难道 Spring 只能通过配置方式来创建 Bean 吗？回答当然不是，某些 SingletonBeanRegistry 接口实现类实现也允许将那些非 BeanFactory 创建的、已有的用户对象注册到容器中，这些对象必须是共享的，比如使用 DefaultListableBeanFactory

的 registerSingleton() 方法。不过建议采用元数据定义。

2.3.3 Bean 的命名

每个 Bean 可以有一个或多个 id（或称之为标识符或名字），在这里我们把**第一个 id 称为“标识符”，其余 id 叫做“别名”**；这些 id 在 IoC 容器中必须唯一。如何为 Bean 指定 id 呢，有以下几种方式；

一、 不指定 id，只配置必须的全限定类名，由 IoC 容器为其生成一个标识，客户端必须通过接口“T getBean(Class<T> requiredType)”获取 Bean；

```
<bean class=” cn.javass.spring.chapter2.helloworld.HelloImpl”/>          （1）
```

测试代码片段如下：

```
@Test
public void test1() {
    BeanFactory beanFactory =
        new ClassPathXmlApplicationContext("chapter2/namingbean1.xml");
    //根据类型获取 bean
    HelloApi helloApi = beanFactory.getBean(HelloApi.class);
    helloApi.sayHello();
}
```

二、指定 id，必须在 Ioc 容器中唯一；

```
<bean id=” bean” class=” cn.javass.spring.chapter2.helloworld.HelloImpl”/>      （2）
```

测试代码片段如下：

```
@Test
public void test2() {
    BeanFactory beanFactory =
        new ClassPathXmlApplicationContext("chapter2/namingbean2.xml");
    //根据id获取bean
    HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
    bean.sayHello();
}
```

三、指定 name，这样 name 就是“标识符”，必须在 Ioc 容器中唯一；

```
<bean name=” bean” class=” cn.javass.spring.chapter2.helloworld.HelloImpl”/> （3）
```

测试代码片段如下：

```
@Test
public void test3() {
      BeanFactory beanFactory =
            new ClassPathXmlApplicationContext("chapter2/namingbean3.xml");
      //根据 name 获取 bean
      HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
      bean.sayHello();
`
```

四、指定 id 和 name，id 就是标识符，而 name 就是别名，必须在 Ioc 容器中唯一；

```
<bean id=”bean1” name=”alias1”
                  class=” cn.javass.spring.chapter2.helloworld.HelloImpl”/>
<!-- 如果id和name一样，IoC容器能检测到，并消除冲突 -->
<bean id="bean3" name="bean3"
       class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>                    （4）
```

测试代码片段如下：

```
@Test
public void test4() {
      BeanFactory beanFactory =
            new ClassPathXmlApplicationContext("chapter2/namingbean4.xml");
      //根据id获取bean
      HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
      bean1.sayHello();
      //根据别名获取bean
      HelloApi bean2 = beanFactory.getBean("alias1", HelloApi.class);
      bean2.sayHello();
      //根据id获取bean
      HelloApi bean3 = beanFactory.getBean("bean3", HelloApi.class);
      bean3.sayHello();
      String[] bean3Alias = beanFactory.getAliases("bean3");
      //因此别名不能和id一样，如果一样则由IoC容器负责消除冲突
      Assert.assertEquals(0, bean3Alias.length);
}
```

五、指定多个 name，多个 name 用“，”、“；”、“ ”分割，第一个被用作标识符，其他的（alias1、alias2、alias3）是别名，所有标识符也必须在 Ioc 容器中唯一；

```
<bean name=" bean1;alias11,alias12;alias13 alias14"
        class=" cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<!-- 当指定id时，name指定的标识符全部为别名 -->
<bean id="bean2" name="alias21;alias22"
      class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>                （5）
```

测试代码片段如下：

```
@Test
public void test5() {
    BeanFactory beanFactory =
        new ClassPathXmlApplicationContext("chapter2/namingbean5.xml");
    //1根据id获取bean
    HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
    bean1.sayHello();
    //2根据别名获取bean
    HelloApi alias11 = beanFactory.getBean("alias11", HelloApi.class);
    alias11.sayHello();
    //3验证确实是四个别名
    String[] bean1Alias = beanFactory.getAliases("bean1");
    System.out.println("=======namingbean5.xml bean1  别名========");
    for(String alias : bean1Alias) {
        System.out.println(alias);
    }
    Assert.assertEquals(4, bean1Alias.length);
    //根据id获取bean
    HelloApi bean2 = beanFactory.getBean("bean2", HelloApi.class);
    bean2.sayHello();
    //2根据别名获取bean
    HelloApi alias21 = beanFactory.getBean("alias21", HelloApi.class);
    alias21.sayHello();
    //验证确实是两个别名
    String[] bean2Alias = beanFactory.getAliases("bean2");
    System.out.println("=======namingbean5.xml bean2  别名========");
    for(String alias : bean2Alias) {
        System.out.println(alias);
    }
    Assert.assertEquals(2, bean2Alias.length);
}
```

六、使用<alias>标签指定别名，别名也必须在 IoC 容器中唯一

```
<bean name="bean" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<alias alias="alias1" name="bean"/>
<alias alias="alias2" name="bean"/>                                    （6）
```

测试代码片段如下：

```
@Test
public void test6() {
      BeanFactory beanFactory =
            new ClassPathXmlApplicationContext("chapter2/namingbean6.xml");
      //根据id获取bean
      HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
      bean.sayHello();
      //根据别名获取bean
      HelloApi alias1 = beanFactory.getBean("alias1", HelloApi.class);
      alias1.sayHello();
      HelloApi alias2 = beanFactory.getBean("alias2", HelloApi.class);
      alias2.sayHello();
      String[] beanAlias = beanFactory.getAliases("bean");
      System.out.println("=======namingbean6.xml bean  别名========");
      for(String alias : beanAlias) {
            System.out.println(alias);
     }
     System.out.println("=======namingbean6.xml bean  别名========");
      Assert.assertEquals(2, beanAlias.length);
  }
```

以上测试代码在 cn.javass.spring.chapter2.NamingBeanTest.java 文件中。

从定义来看，name 或 id 如果指定它们中的一个时都作为“标识符”，那为什么还要有 id 和 name 同时存在呢？这是因为当使用基于 XML 的配置元数据时，在 XML 中 id 是一个真正的 XML id 属性，因此当其他的定义来引用这个 id 时就体现出 id 的好处了，可以利用 XML 解析器来验证引用的这个 id 是否存在，从而更早的发现是否引用了一个不存在的 bean，而使用 name，则可能要在真正使用 bean 时才能发现引用一个不存在的 bean。

- **Bean 命名约定：**Bean 的命名遵循 XML 命名规范，但最好符合 Java 命名规范，由“字母、数字、下划线组成“，而且应该养成一个良好的命名习惯， 比如采用“驼峰式”，即第一个单词首字母开始，从第二个单词开始首字母大写开始，这样可以增加可读性。

2.3.4 实例化 Bean

Spring IoC 容器如何实例化 Bean 呢？传统应用程序可以通过 new 和反射方式进行实例化 Bean。而 Spring IoC 容器则需要根据 Bean 定义里的配置元数据使用反射机制来创建 Bean。在 Spring IoC 容器中根据 Bean 定义创建 Bean 主要有以下几种方式：

一、使用构造器实例化 Bean：这是最简单的方式，Spring IoC 容器即能使用默认空构造器也能使用有参数构造器两种方式创建 Bean，如以下方式指定要创建的 Bean 类型：

- 使用空构造器进行定义，使用此种方式，class属性指定的类必须有空构造器

```
<bean name="bean1" class="cn.javass.spring.chapter2.HelloImpl2"/>
```

- 使用有参数构造器进行定义，使用此中方式，可以使用< constructor-arg >标签指定构造器参数值，其中index表示位置，value表示常量值，也可以指定引用，指定引用使用ref来引用另一个Bean定义，后边会详细介绍：

```
<bean name="bean2" class="cn.javass.spring.chapter2.HelloImpl2">
    <!-- 指定构造器参数 -->
    <constructor-arg index="0" value="Hello Spring!"/>
</bean>
```

- 知道如何配置了，让我们做个例子的例子来实践一下吧：

（1）准备 Bean class(HelloImpl2.java)，该类有一个空构造器和一个有参构造器：

```
package cn.javass.spring.chapter2;
public class HelloImpl2 implements HelloApi {
        private String message;
        public HelloImpl2() {
            this.message = "Hello World!";
        }
        Public HelloImpl2(String message) {
            this.message = message;
        }
        @Override
        public void sayHello() {
            System.out.println(message);
        }
    }
```

（2）在配置文件(resources/chapter2/instantiatingBean.xml)配置 Bean 定义，如下所示：

```
<!--使用默认构造参数-->
<bean name="bean1" class="cn.javass.spring.chapter2.HelloImpl2"/>
    <!--使用有参数构造参数-->

<bean name="bean2" class="cn.javass.spring.chapter2.HelloImpl2">
    <!-- 指定构造器参数 -->
    <constructor-arg index="0" value="Hello Spring!"/>
</bean>
```

（3）配置完了，让我们写段测试代码（InstantiatingContainerTest）来看下是否工作吧：

```
@Test
public void testInstantiatingBeanByConstructor() {
    //使用构造器
    BeanFactory beanFactory =
        new ClassPathXmlApplicationContext("chapter2/instantiatingBean.xml");
    HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
    bean1.sayHello();
    HelloApi bean2 = beanFactory.getBean("bean2", HelloApi.class);
    bean2.sayHello();
}
```

二、使用静态工厂方式实例化 Bean，使用这种方式除了指定必须的 class 属性，还要指定 factory-method 属性来指定实例化 Bean 的方法，而且使用静态工厂方法也允许指定方法参数，spring IoC 容器将调用此属性指定的方法来获取 Bean，配置如下所示：

（1）先来看看静态工厂类代码吧 HelloApiStaticFactory：

```
public class HelloApiStaticFactory {
    //工厂方法
    public static HelloApi newInstance(String message) {
        //返回需要的Bean实例
        return new HelloImpl2(message);
}
```

（2）静态工厂写完了，让我们在配置文件(resources/chapter2/instantiatingBean.xml)配置 Bean 定义：

```
<!-- 使用静态工厂方法 -->
<bean id="bean3" class="cn.javass.spring.chapter2.HelloApiStaticFactory"
      factory-method="newInstance">
      <constructor-arg index="0" value="Hello Spring!"/>
</bean>
```

（3）配置完了，写段测试代码来测试一下吧，InstantiatingBeanTest：

```
@Test
public void testInstantiatingBeanByStaticFactory() {
    //使用静态工厂方法
    BeanFactory beanFactory =
        new ClassPathXmlApplicationContext("chaper2/instantiatingBean.xml");
    HelloApi bean3 = beanFactory.getBean("bean3", HelloApi.class);
    bean3.sayHello();
}
```

三、使用实例工厂方法实例化 Bean，使用这种方式不能指定 class 属性，此时必须使用 factory-bean 属性来指定工厂 Bean，factory-method 属性指定实例化 Bean 的方法，而且使用实例工厂方法允许指定方法参数，方式和使用构造器方式一样，配置如下：

（1）实例工厂类代码（HelloApiInstanceFactory.java）如下：

```
package cn.javass.spring.chapter2;
public class HelloApiInstanceFactory {
    public HelloApi newInstance(String message) {
        return new HelloImpl2(message);
    }
}
```

（2）让我们在配置文件(resources/chapter2/instantiatingBean.xml)配置 Bean 定义：

```
<!—1、定义实例工厂 Bean -->
<bean id="beanInstanceFactory"
    class="cn.javass.spring.chapter2.HelloApiInstanceFactory"/>
<!—2、使用实例工厂 Bean 创建 Bean -->
<bean id="bean4"
      factory-bean="beanInstanceFactory"
      factory-method="newInstance">
     <constructor-arg index="0" value="Hello Spring!"></constructor-arg>
</bean>
```

（3）测试代码 InstantiatingBeanTest：

```
@Test
public void testInstantiatingBeanByInstanceFactory() {
    //使用实例工厂方法
    BeanFactory beanFactory =
            new ClassPathXmlApplicationContext("chapter2/instantiatingBean.xml");
    HelloApi bean4 = beanFactory.getBean("bean4", HelloApi.class);
    bean4.sayHello();
}
```

通过以上例子我们已经基本掌握了如何实例化 Bean 了，大家是否注意到？这三种方式只是配置不一样，从获取方式看完全一样，没有任何不同。这也是 Spring IoC 的魅力，Spring IoC 帮你创建 Bean，我们只管使用就可以了，是不是很简单。

2.3.5 小结

到此我们已经讲完了 Spring IoC 基础部分，包括 IoC 容器概念，如何实例化容器，Bean 配置、命名及实例化，Bean 获取等等。不知大家是否注意到到目前为止，我们只能通过简单的实例化 Bean，没有涉及 Bean 之间关系。接下来一章让我们进入配置 Bean 之间关系章节，也就是依赖注入。

第三章　DI

3.1　DI 的配置使用

3.1.1　依赖和依赖注入

传统应用程序设计中所说的依赖一般指“类之间的关系”，那先让我们复习一下类之间的关系：

- **泛化：**表示类与类之间的继承关系、接口与接口之间的继承关系；
- **实现：**表示类对接口的实现；
- **依赖：**当类与类之间有使用关系时就属于依赖关系，不同于关联关系，依赖不具有“拥有关系”，而是一种“相识关系”，只在某个特定地方（比如某个方法体内）才有关系。
- **关联：**表示类与类或类与接口之间的依赖关系，表现为“拥有关系”；具体到代码可以用实例变量来表示；
- **聚合：**属于是关联的特殊情况，体现部分-整体关系，是一种弱拥有关系；整体和部分可以有不一样的生命周期；是一种弱关联；
- **组合：**属于是关联的特殊情况，也体现了体现部分-整体关系，是一种强“拥有关系”；整体与部分有相同的生命周期，是一种强关联；

Spring IoC 容器的依赖有两层含义：**Bean 依赖容器**和**容器注入 Bean 的依赖资源**：

- **Bean 依赖容器：**也就是说 Bean 要依赖于容器，这里的依赖是指容器负责创建 Bean 并管理 Bean 的生命周期，正是由于由容器来控制创建 Bean 并注入依赖，也就是控制权被反转了，这也正是 IoC 名字的由来，**此处的有依赖是指 Bean 和容器之间的依赖关系**。
- **容器注入 Bean 的依赖资源：**容器负责注入 Bean 的依赖资源，依赖资源可以是 Bean、外部文件、常量数据等，在 Java 中都反映为对象，并且由容器负责组装 Bean 之间的依赖关系，**此处的依赖是指 Bean 之间的依赖关系，可以认为是传统类与类之间的“关联”、“聚合”、“组合”关系**。

为什么要应用依赖注入，应用依赖注入能给我们带来哪些好处呢？

- **动态替换 Bean 依赖对象，程序更灵活：**替换 Bean 依赖对象，无需修改源文件：应用依赖注入后，由于可以采用配置文件方式实现，从而能随时动态的替换 Bean 的依赖对象，无需修改 java 源文件；
- **更好实践面向接口编程，代码更清晰：**在 Bean 中只需指定依赖对象的接口，接口定义依赖对象完成的功能，通过容器注入依赖实现；
- **更好实践优先使用对象组合，而不是类继承：**因为 IoC 容器采用注入依赖，也就

是组合对象，从而更好的实践对象组合。

- 采用对象组合，Bean 的功能可能由几个依赖 Bean 的功能组合而成，其 Bean 本身可能只提供少许功能或根本无任何功能，全部委托给依赖 Bean，对象组合具有动态性，能更方便的替换掉依赖 Bean，从而改变 Bean 功能；
- 而如果采用类继承，Bean 没有依赖 Bean，而是采用继承方式添加新功能，，而且功能是在编译时就确定了，不具有动态性，而且采用类继承导致 Bean 与子 Bean 之间高度耦合，难以复用。

- **增加 Bean 可复用性：**依赖于对象组合，Bean 更可复用且复用更简单；
- **降低 Bean 之间耦合：**由于我们完全采用面向接口编程，在代码中没有直接引用 Bean 依赖实现，全部引用接口，而且不会出现显示的创建依赖对象代码，而且这些依赖是由容器来注入，很容易替换依赖实现类，从而降低 Bean 与依赖之间耦合；
- **代码结构更清晰：**要应用依赖注入，代码结构要按照规约方式进行书写，从而更好的应用一些最佳实践，因此代码结构更清晰。

从以上我们可以看出，其实依赖注入只是一种装配对象的手段，设计的类结构才是基础，如果设计的类结构不支持依赖注入，Spring IoC 容器也注入不了任何东西，从而从根本上说**“如何设计好类结构才是关键，依赖注入只是一种装配对象手段”。**

前边 IoC 一章我们已经了解了 Bean 依赖容器，那容器如何注入 Bean 的依赖资源，Spring IoC 容器注入依赖资源主要有以下两种基本实现方式：

- **构造器注入：**就是容器实例化 Bean 时注入那些依赖，通过在在 Bean 定义中指定构造器参数进行注入依赖，包括实例工厂方法参数注入依赖，但静态工厂方法参数不允许注入依赖；
- **setter 注入：**通过 setter 方法进行注入依赖；
- **方法注入：**能通过配置方式替换掉 Bean 方法，也就是通过配置改变 Bean 方法功能。

我们已经知道注入实现方式了，接下来让我们来看看具体配置吧。

3.1.2 构造器注入

使用构造器注入通过配置构造器参数实现，构造器参数就是依赖。除了构造器方式，还有静态工厂、实例工厂方法可以进行构造器注入。如图 3-1 所示：

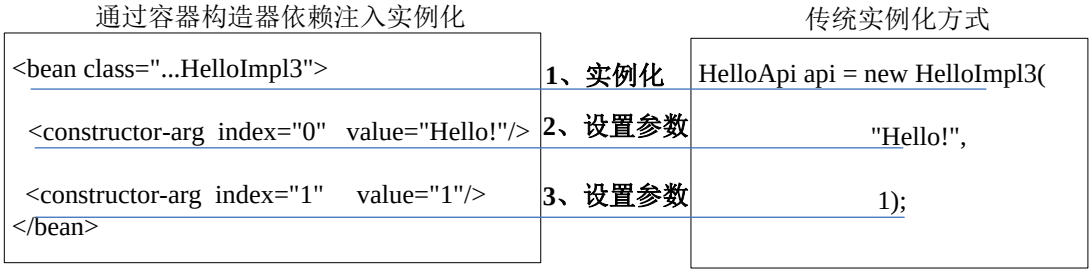


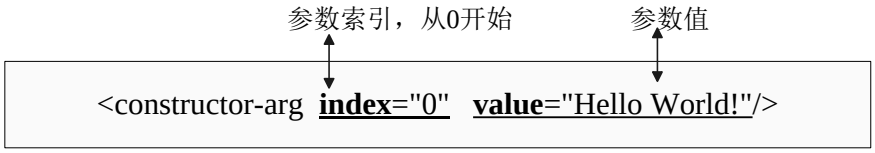
图 3-1 实例化

构造器注入可以根据参数索引注入、参数类型注入或 Spring3 支持的参数名注入，但参数名注入是有限制的，需要使用在编译程序时打开调试模式（即在编译时使用“javac –g:vars”在 class 文件中生成变量调试信息，默认是不包含变量调试信息的，从而能获取参数名字，否则获取不到参数名字）或在构造器上使用 @ConstructorProperties（java.beans.ConstructorProperties）注解来指定参数名。

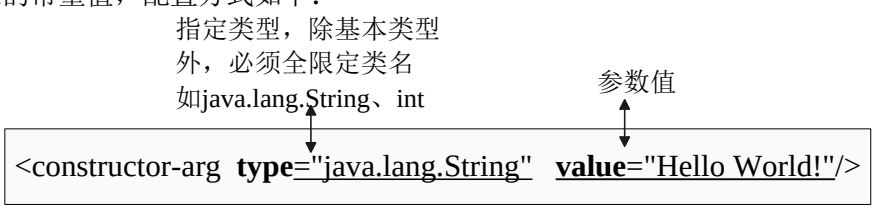
首先让我们准备测试构造器类 HelloImpl3.java，该类只有一个包含两个参数的构造器：

```
package cn.javass.spring.chapter3.helloworld;
public class HelloImpl3 implements HelloApi {
    private String message;
    private int index;
    //@java.beans.ConstructorProperties({"message", "index"})
    public HelloImpl3(String message, int index) {
        this.message = message;
        this.index = index;
    }
    @Override
    public void sayHello() {
        System.out.println(index + ":" + message);
    }
```

一、根据参数索引注入，使用标签“<constructor-arg index="1" value="1"/>”来指定注入的依赖，其中“index”表示索引，从 0 开始，即第一个参数索引为 0，“value”来指定注入的常量值，配置方式如下：



二、根据参数类型进行注入，使用标签“<constructor-arg type="java.lang.String" value="Hello World!"/>”来指定注入的依赖，其中“type”表示需要匹配的参数类型，可以是基本类型也可以是其他类型，如“int”、“java.lang.String”，“value”来指定注入的常量值，配置方式如下：



三、根据参数名进行注入，使用标签“<constructor-arg name="message" value="Hello World!"/>”来指定注入的依赖，其中“name”表示需要匹配的参数名字，“value”来指定注入的常量值，配置方式如下：

四、让我们来用具体的例子来看一下构造器注入怎么使用吧。

（1）首先准备 Bean 类，在此我们就使用“HelloImpl3”这个类。

（2）有了 Bean 类，接下来要进行 Bean 定义配置，我们需要配置三个 Bean 来完成上述三种依赖注入测试，其中 Bean ”byIndex”是通过索引注入依赖；Bean ”byType”是根据类型进行注入依赖；Bean ”byName”是根据参数名字进行注入依赖，具体配置文件（resources/chapter3/ constructorDependencyInject.xml）如下：

```
<!-- 通过构造器参数索引方式依赖注入 -->
<bean id="byIndex" class="cn.javass.spring.chapter3.HelloImpl3">
      <constructor-arg index="0" value="Hello World!"/>
      <constructor-arg index="1" value="1"/>
</bean>
<!-- 通过构造器参数类型方式依赖注入 -->
<bean id="byType" class="cn.javass.spring.chapter3.HelloImpl3">
    <constructor-arg type="java.lang.String" value="Hello World!"/>
    <constructor-arg type="int" value="2"/>
</bean>
<!-- 通过构造器参数名称方式依赖注入 -->
<bean id="byName" class="cn.javass.spring.chapter3.HelloImpl3">
    <constructor-arg name="message" value="Hello World!"/>
    <constructor-arg name="index" value="3"/>
</bean>
```

3）配置完毕后，在测试之前，因为我们使用了通过构造器参数名字注入方式，请确保编译时 class 文件包含“变量信息”，具体查看编译时是否包含“变量调试信息”请右击项目，在弹出的对话框选择属性；然后在弹出的对话框选择“Java Compiler”条目，在“Class 文件 生成”框中选择“添加变量信息到 Class 文件（调试器使用）”，具体如图 3-1：

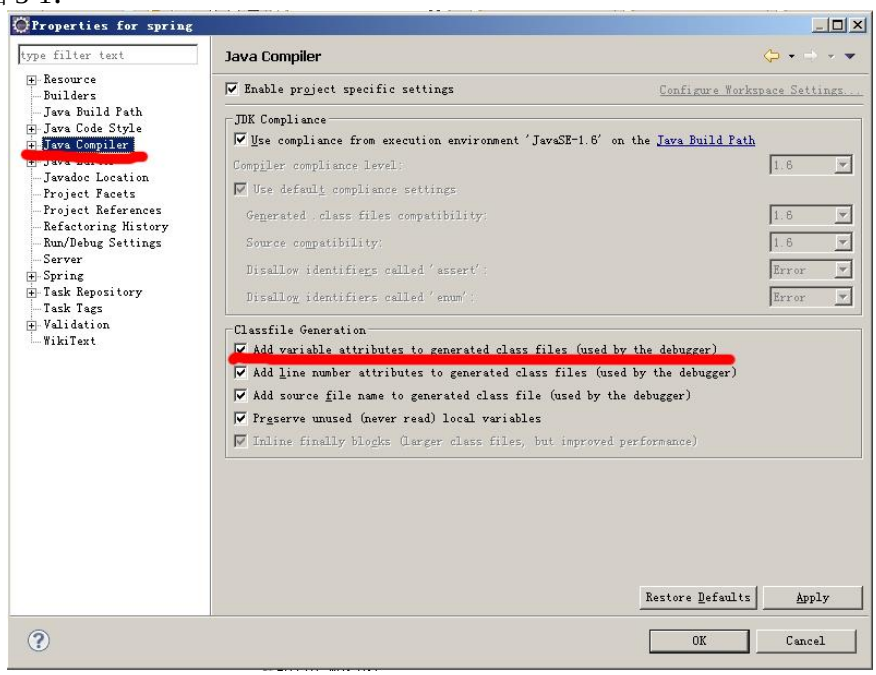


图 3-2 编译时打开“添加变量信息选项”

（4）接下来让我们测试一下配置是否工作，具体测试代码（cn.javass.spring.chapter3.DependencyInjectTest）如下：

```
@Test
public void testConstructorDependencyInjectTest() {
    BeanFactory beanFactory =  new
    ClassPathXmlApplicationContext("chapter3/constructorDependencyInject.xml");
    //获取根据参数索引依赖注入的Bean
    HelloApi byIndex = beanFactory.getBean("byIndex", HelloApi.class);
    byIndex.sayHello();
    //获取根据参数类型依赖注入的Bean
    HelloApi byType = beanFactory.getBean("byType", HelloApi.class);
    byType.sayHello();
    //获取根据参数名字依赖注入的Bean
    HelloApi byName = beanFactory.getBean("byName", HelloApi.class);
    byName.sayHello();
}
```

通过以上测试我们已经会基本的构造器注入配置了，在测试通过参数名字注入时，除了可以使用以上方式，还可以通过在构造器上添加@java.beans.ConstructorProperties({"message", "index"})注解来指定参数名字，在 HelloImpl3 构造器上把注释掉的“ConstructorProperties”打开就可以了，这个就留给大家做练习，自己配置然后测试一下。

五、大家已经会了构造器注入，那让我们再看一下静态工厂方法注入和实例工厂注入吧，其实它们注入配置是完全一样，在此我们只示范一下静态工厂注入方式和实例工厂方式配置，测试就留给大家自己练习：

（1）静态工厂类

```
//静态工厂类
package cn.javass.spring.chapter3;
import cn.javass.spring.chapter2.helloworld.HelloApi;
public class DependencyInjectByStaticFactory {
    public static HelloApi newInstance(String message, int index) {
        return new HelloImpl3(message, index);
    }
}
```

静态工厂类 Bean 定义配置文件（chapter3/staticFactoryDependencyInject.xml）

```
<bean id="byIndex"
        class="cn.javass.spring.chapter3.DependencyInjectByStaticFactory"
        factory-method="newInstance">
      <constructor-arg index="0" value="Hello World!"/>
      <constructor-arg index="1" value="1"/>
</bean>
<bean id="byType"
        class="cn.javass.spring.chapter3.DependencyInjectByStaticFactory"
        factory-method="newInstance">
      <constructor-arg type="java.lang.String" value="Hello World!"/>
            <constructor-arg type="int" value="2"/>
</bean>
<bean id="byName"
      class="cn.javass.spring.chapter3.DependencyInjectByStaticFactory"
factory-method="newInstance">
      <constructor-arg name="message" value="Hello World!"/>
            <constructor-arg name="index" value="3"/>
      </bean>
```

（2）实例工厂类

```
//实例工厂类
package cn.javass.spring.chapter3;
import cn.javass.spring.chapter2.helloworld.HelloApi;
public class DependencyInjectByInstanceFactory {
      public HelloApi newInstance(String message, int index) {
            return new HelloImpl3(message, index);
      }
}
```

实例工厂类 Bean 定义配置文件（chapter3/instanceFactoryDependencyInject.xml）

```
<bean id="instanceFactory"
        class="cn.javass.spring.chapter3.DependencyInjectByInstanceFactory"/>

<bean id="byIndex"
      factory-bean="instanceFactory"    factory-method="newInstance">
      <constructor-arg index="0" value="Hello World!"/>
      <constructor-arg index="1" value="1"/>
</bean>
<bean id="byType"
        factory-bean="instanceFactory" factory-method="newInstance">
      <constructor-arg type="java.lang.String" value="Hello World!"/>
      <constructor-arg type="int" value="2"/>
</bean>
<bean id="byName"
        factory-bean="instanceFactory" factory-method="newInstance">
      <constructor-arg name="message" value="Hello World!"/>
      <constructor-arg name="index" value="3"/>
</bean>
```

（3）测试代码和构造器方式完全一样，只是配置文件不一样，大家只需把测试文件改一下就可以了。还有一点需要大家注意就是静态工厂方式和实例工厂方式根据参数名字注入的方式只支持通过在 class 文件中添加“变量调试信息”方式才能运行，ConstructorProperties 注解方式不能工作，它只对构造器方式起作用，**不建议使用根据参数名进行构造器注入**。

3.1.3 setter 注入

setter 注入，是通过在通过构造器、静态工厂或实例工厂实例好 Bean 后，通过调用 Bean 类的 setter 方法进行注入依赖，如图 3-3 所示：

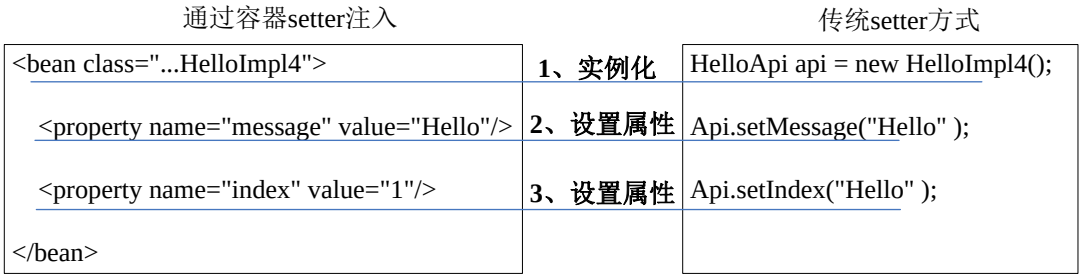


图 3-3 setter 注入方式

setter 注入方式只有一种根据 setter 名字进行注入：

知道配置方式了，接下来先让我们来做个简单例子吧。

（1）准备测试类 HelloImpl4，需要两个 setter 方法“setMessage”和“setIndex”：

```
package cn.javass.spring.chapter3;
import cn.javass.spring.chapter2.helloworld.HelloApi;
public class HelloImpl4 implements HelloApi {
    private String message;
    private int index;
    //setter方法
    public void setMessage(String message) {
        this.message = message;
    }
    public void setIndex(int index) {
        this.index = index;
    }
    @Override
    public void sayHello() {
        System.out.println(index + ":" + message);
    }
}
```

（2）配置 Bean 定义，具体配置文件（resources/chapter3/setterDependencyInject.xml）片段如下：

```
<!-- 通过setter方式进行依赖注入 -->
    <bean id="bean" class="cn.javass.spring.chapter3.HelloImpl4">
        <property name="message" value="Hello World!"/>
        <property name="index">
            <value>1</value>
        </property>
    </bean>
```

（3）该写测试进行测试一下是否满足能工作了，其实测试代码一点没变，变的是配置：

```
@Test
public void testSetterDependencyInject() {
    BeanFactory beanFactory =
    new ClassPathXmlApplicationContext("chapter3/setterDependencyInject.xml");
    HelloApi bean = beanFactory.getBean("bean", HelloApi.class);
    bean.sayHello();
}
```

知道如何配置了，但 Spring 如何知道 setter 方法？如何将值注入进去的呢？其实方法名是要遵守约定的，setter 注入的方法名要遵循“JavaBean getter/setter 方法命名约定”：

JavaBean：是本质就是一个 POJO 类，但具有一下限制：

- 该类必须要**有公共的无参构造器**，如 public HelloImpl4() {}；
- **属性为 private 访问级别**，不建议 public，如 private String message;
- **属性必要时通过一组 setter（修改器）和 getter（访问器）方法来访问**；
- **setter 方法，以“set” 开头，后跟首字母大写的属性名**，如“setMesssage”，简单属性一般只有一个方法参数，方法返回值通常为“void”；
- **getter 方法，一般属性以“get”开头，对于 boolean 类型一般以“is”开头，后跟首字母大写的属性名**，如“getMesssage”，“isOk”；
- 还有一些其他特殊情况，**比如属性有连续两个大写字母开头，如“URL”，则 setter/getter 方法为：“setURL”和“getURL”**，其他一些特殊情况请参看“Java Bean”命名规范。

3.1.4 注入常量

注入常量是依赖注入中最简单的。配置方式如下所示：

```
<property name="message" value="Hello World!"/>
或
<property name="index"><value>1</value></property>
```

以上两种方式都可以，从配置来看第一种更简洁。注意此处“value”中指定的全是字符串，由 Spring 容器将此字符串转换成属性所需要的类型，如果转换出错，将抛出相应的异常。

Spring 容器目前能对各种基本类型把配置的 String 参数转换为需要的类型。

注：Spring 类型转换系统对于 boolean 类型进行了容错处理，除了可以使用“true/false”标准的 Java 值进行注入，还能使用“yes/no”、“on/off”、“1/0”来代表“真/假”，所以大家在学习或工作中遇到这种类似问题不要觉得是人家配置错了，而是 Spring 容错做的非常好。

测试类

```
public class BooleanTestBean {
    private boolean success;
    public void setSuccess(boolean success) {
        this.success = success;
    }
    public boolean isSuccess() {
        return success;
    }
}
```

配置文件（chapter3/booleanInject.xml）片段：

```
<!-- boolean参数值可以用on/off -->
<bean id="bean2" class="cn.javass.spring.chapter3.bean.BooleanTestBean">
    <property name="success" value="on"/>
</bean>
<!-- boolean参数值可以用yes/no -->
<bean id="bean3" class="cn.javass.spring.chapter3.bean.BooleanTestBean">
    <property name="success" value="yes"/>
</bean>
<!-- boolean参数值可以用1/0 -->
<bean id="bean4" class="cn.javass.spring.chapter3.bean.BooleanTestBean">
    <property name="success" value="1"/>
</bean>
```

3.1.5 注入 Bean ID

用于注入 Bean 的 ID，ID 是一个常量不是引用，且类似于注入常量，但提供错误验证功能，配置方式如下所示：

```
<property name="id"><idref bean="bean1"/></property>
```

```
<property name="id"><idref local="bean2"/></property>
```

两种方式都可以，上述配置本质上在运行时等于如下方式

```
<bean id="bean1" class="……"/>
<bean id="idrefBean1" class="……">
    <property name="id" value ="bean1"/>
</bean>
```

第一种方式可以在容器初始化时校验被引用的 Bean 是否存在，如果不存在将抛出异常，而第二种方式只有在 Bean 实际使用时才能发现传入的 Bean 的 ID 是否正确，可能发生不可预料的错误。因此如果想注入 Bean 的 ID，推荐使用第一种方式。

接下来学习一下如何使用吧：

首先定义测试 Bean：

```
package cn.javass.spring.chapter3.bean
public class IdRefTestBean {
    private String id;
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
}
```

其次定义配置文件（chapter3/idRefInject.xml）：

```
<bean id="bean1" class="java.lang.String">
    <constructor-arg index="0" value="test"/>
</bean>
<bean id="bean2" class="java.lang.String">
    <constructor-arg index="0" value="test"/>
</bean>
```

```
<bean id="idrefBean1" class="cn.javass.spring.chapter3.bean.IdRefTestBean">
        <property name="id"><idref bean="bean1"/></property>
</bean>
<bean id="idrefBean2" class="cn.javass.spring.chapter3.bean.IdRefTestBean">
    <property name="id"><idref local="bean2"/></property>
</bean>
```

从配置中可以看出，注入的 Bean 的 ID 是一个 java.lang.String 类型，即字符串类型，因此注入的同样是常量，只是具有校验功能。

<idref bean="……"/>将在容器初始化时校验注入的 ID 对于的 Bean 是否存在，如果不存在将抛出异常。

<idref local="……"/>将在 XML 解析时校验注入的 ID 对于的 Bean 在当前配置文件中是否存在，如果不存在将抛出异常，它不同于<idref bean="……"/>是校验发生在 XML 解析式而非容器初始化时，且只检查当前配置文件中是否存在相应的 Bean。

3.1.6 注入集合、数组和字典

Spring 不仅能注入简单类型数据，还能注入集合（Collection、无序集合 Set、有序集合 List）类型、数组(Array)类型、字典(Map)类型数据、Properties 类型数据，接下来就让我们一个个看看如何注入这些数据类型的数据。

一、注入集合类型：包括 Collection 类型、Set 类型、List 类型数据：

（1）List类型：需要使用<list>标签来配置注入，其具体配置如下：

1、可选的“value-type”属性，表示列表中条目的数据的类型，比如value-type=”java.lang.String”表示列表需要条目为String数据类型；

2、也可以采用泛型，Spring能根据泛型数据自动检测到List里条目的数据类型，比如java.util.List<String>,Spring能自动识别列表需要条目为String数据类型；

3、如果既没有指定“value-type”属性List也不是泛型的则默认就是String类型；

```
<bean ……>
  <property name="values">
    <list value-type="java.lang.String"  merge="default">
      <value>1</value>
      <value>2</value>
      <value>3</value>
    </list>
  </property>
</bean>
```

指定列表条目值

可选的merge属性，后边再介绍，用于父子Bean情况是否合并list条目，

让我们来写个测试来练习一下吧：

准备测试类：

```
package cn.javass.spring.chapter3.bean;
import java.util.List;
public class ListTestBean {
	private List<String> values;
	public List<String> getValues() {
		return values;
	}
	public void setValues(List<String> values) {
		this.values = values;
	}
}
```

进行 Bean 定义，在配置文件（resources/chapter3/listInject.xml）中配置 list 注入：

```
<bean id="listBean" class="cn.javass.spring.chapter3.bean.ListTestBean">
	<property name="values">
		<list>
			<value>1</value>
			<value>2</value>
			<value>3</value>
		</list>
	</property>
</bean>
```

测试代码：

```
@Test
public void testListInject() {
	 BeanFactory beanFactory =
	    new ClassPathXmlApplicationContext("chapter3/listInject.xml");
	ListTestBean listBean = beanFactory.getBean("listBean", ListTestBean.class);
	System.out.println(listBean.getValues().size());
	Assert.assertEquals(3, listBean.getValues().size());
}
```

（2）Set类型：需要使用<set>标签来配置注入，其配置参数及含义和<lsit>标签完全一样，在此就不阐述了：

准备测试类：

```
package cn.javass.spring.chapter3.bean;
import java.util.Collection;
public class CollectionTestBean {
    private Collection<String> values;
    public void setValues(Collection<String> values) {
        this.values = values;
    }
    public Collection<String> getValues() {
        return values;
    }
}
```

进行 Bean 定义，在配置文件（resources/chapter3/listInject.xml）中配置 list 注入：

```
<bean id="setBean" class="cn.javass.spring.chapter3.bean.SetTestBean">
    <property name="values">
        <set>
            <value>1</value>
            <value>2</value>
            <value>3</value>
        </set>
    </property>
</bean>
```

具体测试代码就不写了，和 listBean 测试代码完全一样。

（2）Collection 类型：因为 Collection 类型是 Set 和 List 类型的基类型，所以使用<set>或<list>标签都可以进行注入，配置方式完全和以上配置方式一样，只是将测试类属性改成“Collection”类型，如果配置有问题，可参考 cn.javass.spring.chapter3.DependencyInjectTest 测试类中的 testCollectionInject 测试方法中的代码。

二、**注入数组类型**：需要使用<array>标签来配置注入，其中标签属性“value-type”和“merge”和<list>标签含义完全一样，具体配置如下：

```
public class ArrayTestBean {
    private String[] array;
    private String[][] array2;
    public void setArray(String[] array) {
        this.array = array;
    }
    public void setArray2(String[][] array2) {
        this.array2 = array2;
    }
}
```

一维数组注入

二维数组注入

```
<bean id="arrayBean"
      class="cn.javass.spring.chapter3.bean.ArrayTestBean">
  <property name="array">
    <array value-type="java.lang.String" merge="default">
      <value>1</value>
      <value>2</value>
    </array>
  </property>
  <property name="array2">
    <array>
      <array>
        <value>1</value>
      </array>
      <array>
        <value>4</value>
      </array>
    </array>
  </property>
</bean>
```

如果练习时遇到配置问题，可以参考“cn.javass.spring.chapter3.DependencyInjectTest”测试类中的 testArrayInject 测试方法中的代码。

三、注入字典（Map）类型：字典类型是包含键值对数据的数据结构，需要使用<map>标签来配置注入，其属性“key-type”和“value-type”分别指定“键”和“值”的数据类型，其含义和<list>标签的“value-type”含义一样，在此就不罗嗦了，并使用<key>子标签来指定键数据，<value>子标签来指定键对应的值数据，具体配置如下：

```
//对应的类文件代码
public class MapTestBean {
    private Map<String, String> values;
    //对应的setter/getter方法
    public void setValues(Map<String, String> values) {
        this.values = values;
    }
}
```

需要注入的Map数据

配置

```
   <bean id="mapBean"
class="cn.javass.spring.chapter3.bean.MapTestBean">
        <property name="values">
            <map  key-type="java.lang.String"
                  value-type="java.lang.String">
                <entry>
                    <key><value>1</value></key>
                    <value>11</value>
                </entry>

                <entry key="2" value="22"/>
            </map>
        </property>
    </bean>
```

<map>表示Map注入

<entry>表示键值对

<key>表示键数据

<value>表示键对应的值数据

更简单的配置方式

如果练习时遇到配置问题，可以参考“cn.javass.spring.chapter3.DependencyInjectTest”测试类中的 testMapInject 测试方法中的代码。

四、**Properties 注入**：Spring 能注入 java.util.Properties 类型数据，需要使用<props>标签来配置注入，键和值类型必须是 String，不能变，子标签<prop key=”键”>值</prop>来指定键值对，具体配置如下：

```
public class PropertiesTestBean {
    private Properties values;
    public void setValues(Properties values) {
        this.values = values;
    }
}
```

配置

```
 <bean id="propertiesBean"
  class="cn.javass.spring.chapter3.bean.PropertiesTestBean">
  <property name="values">
    <props value-type="int" merge="default">
      <prop key="1">1sss</prop>
      <prop key="2">2</prop>
    </props>
  </property>
</bean>
```

虽然指定了value-type，但其实该属性不起作用，Properties键和值全是String类型

```
public class PropertiesTestBean {
    private Properties values;
    public void setValues(Properties values) {
        this.values = values;
    }
}
```

```
 <bean id="propertiesBean"
  class="cn.javass.spring.chapter3.bean.PropertiesTestBean">
  <property name="values">
    <value>
      1=11
      2=22;
      3=33,
      4=44
    </value>
  </property>
</bean>
```

配置

分隔符可以是“换行”、“;”、“,”
不建议使用该配置方式，应该优先选择第一种配置方式

如果练习时遇到配置问题，可以参考 cn.javass.spring.chapter3.DependencyInjectTest 测试类中的 testPropertiesInject 测试方法中的代码。

到此我们已经把简单类型及集合类型介绍完了，大家可能会问怎么没见注入“Bean 之间关系”的例子呢？接下来就让我们来讲解配置 Bean 之间依赖关系，也就是注入依赖 Bean。

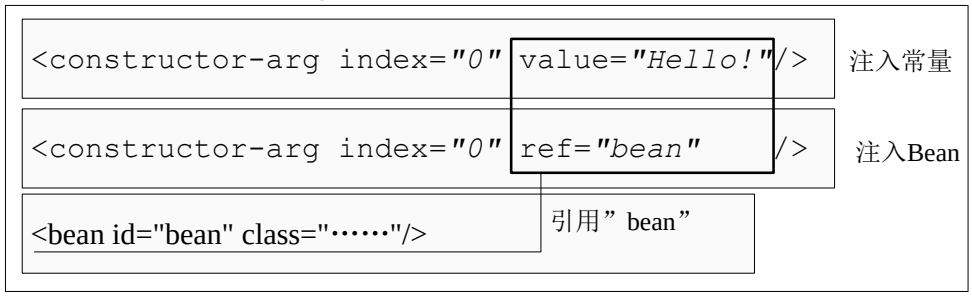
3.1.7 引用其它 Bean

上边章节已经介绍了注入常量、集合等基本数据类型和集合数据类型，本小节将介绍注入依赖 Bean 及注入内部 Bean。

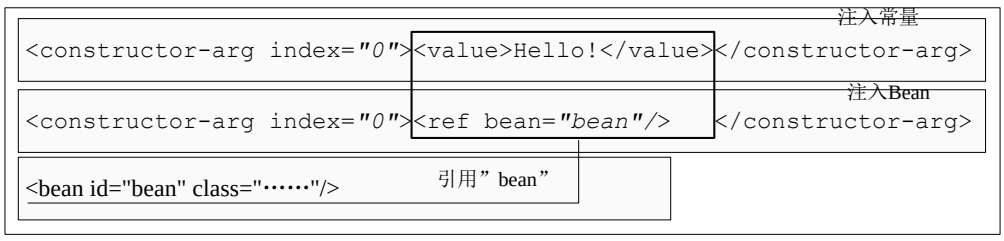
引用其他 Bean 的步骤与注入常量的步骤一样，可以通过构造器注入及 setter 注入引用其他 Bean，只是引用其他 Bean 的注入配置稍微变化了一下：可以将“<constructor-arg index="0" value="Hello World!"/>”和“<property name="message" value="Hello World!"/>”中的 value 属性替换成 bean 属性，其中 bean 属性指定配置文件中的其他 Bean 的 id 或别名。另一种是把<value>标签替换为<.ref bean=”beanName”>，bean 属性也是指定配置文件中的其他 Bean 的 id 或别名。那让我们看一下具体配置吧：

一、构造器注入方式：

（1）通过” <constructor-arg>”标签的 ref 属性来引用其他 Bean，这是最简化的配置：

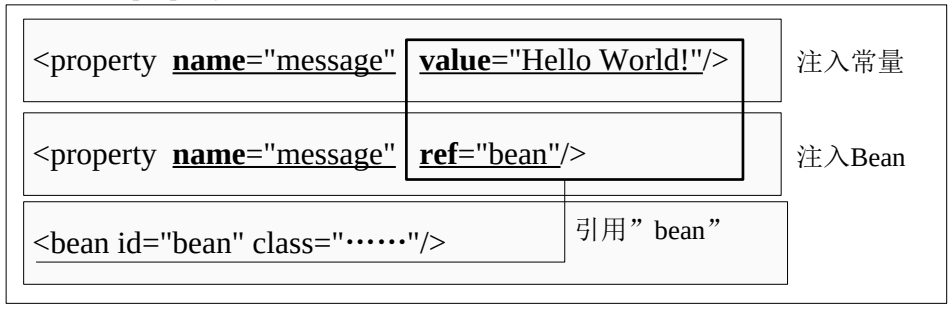


（2）通过” <constructor-arg>”标签的子<ref>标签来引用其他 Bean，使用 bean 属性来指定引用的 Bean：

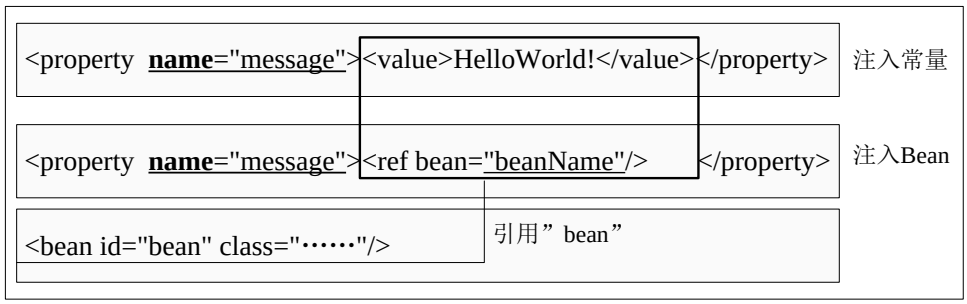


二、setter 注入方式：

（1）通过” <property>”标签的 ref 属性来引用其他 Bean，这是最简化的配置：



（2）通过” <property>”标签的子<ref>标签来引用其他 Bean，使用 bean 属性来指定引用的 Bean：



三、接下来让我们用个具体例子来讲解一下具体使用吧：

（1）首先让我们定义测试引用 Bean 的类，在此我们可以使用原有的 HelloApi 实现，然后再定义一个装饰器来引用其他 Bean，具体装饰类如下：

```
package cn.javass.spring.chapter3.bean;
import cn.javass.spring.chapter2.helloworld.HelloApi;
public class HelloApiDecorator implements HelloApi {
    private HelloApi helloApi;
    //空参构造器
    public HelloApiDecorator() {
    }
    //有参构造器
    public HelloApiDecorator(HelloApi helloApi) {
        this.helloApi = helloApi;
    }
    public void setHelloApi(HelloApi helloApi) {
        this.helloApi = helloApi;
    }
    @Override
    public void sayHello() {
        System.out.println("==========装饰一下==========");
        helloApi.sayHello();
        System.out.println("==========装饰一下==========");
    }
}
```

（2）定义好了测试引用 Bean 接下来该在配置文件(resources/chapter3/beanInject.xml)进行配置 Bean 定义了，在此将演示通过构造器及 setter 方法方式注入依赖 Bean：

```
<!-- 定义依赖Bean -->
<bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<!-- 通过构造器注入 -->
<bean id="bean1" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
    <constructor-arg index="0" ref="helloApi"/>
</bean>
<!-- 通过构造器注入 -->
<bean id="bean2" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
    <property name="helloApi"><ref bean=" helloApi"/></property>
</bean>
```

（3）测试一下吧，测试代码(cn.javass.spring.chapter3.DependencyInjectTest)片段如下：

```
@Test
public void testBeanInject() {
    BeanFactory beanFactory =
        new ClassPathXmlApplicationContext("chapter3/beanInject.xml");
    //通过构造器方式注入
    HelloApi bean1 = beanFactory.getBean("bean1", HelloApi.class);
    bean1.sayHello();
    //通过setter方式注入
    HelloApi bean2 = beanFactory.getBean("bean2", HelloApi.class);
    bean2.sayHello();
}
```

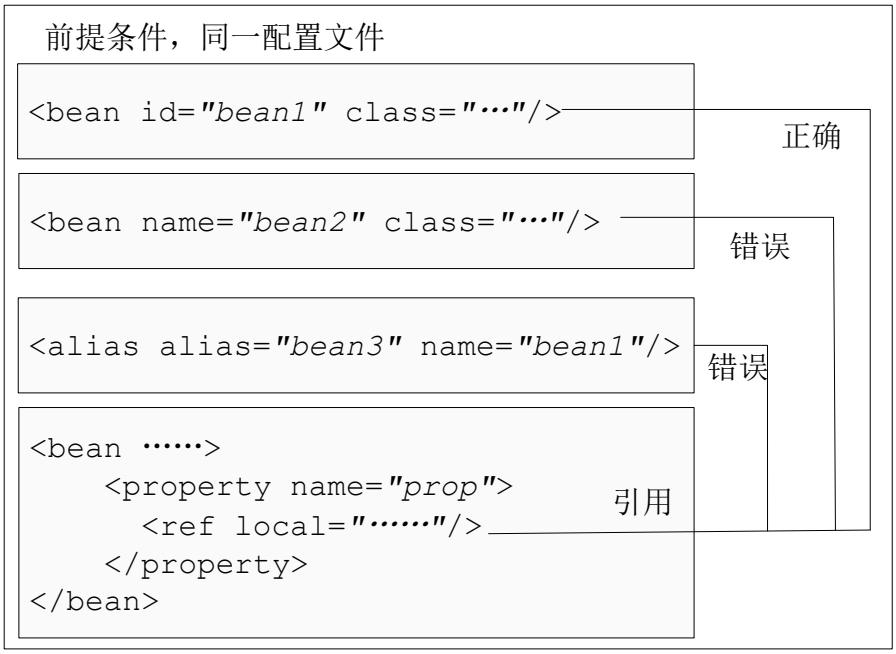
四、其他引用方式：除了最基本配置方式以外，Spring 还提供了另外两种更高级的配置方式，<ref local=””/>和<ref parent=””/>：

（1）<ref local=””/>配置方式：用于引用通过<bean id=”beanName”>方式中通过 id 属性指定的 Bean，它能利用 XML 解析器的验证功能在读取配置文件时来验证引用的 Bean 是否存在。因此如果在当前配置文件中有相互引用的 Bean 可以采用<ref local>方式从而如果配置错误能在开发调试时就发现错误。

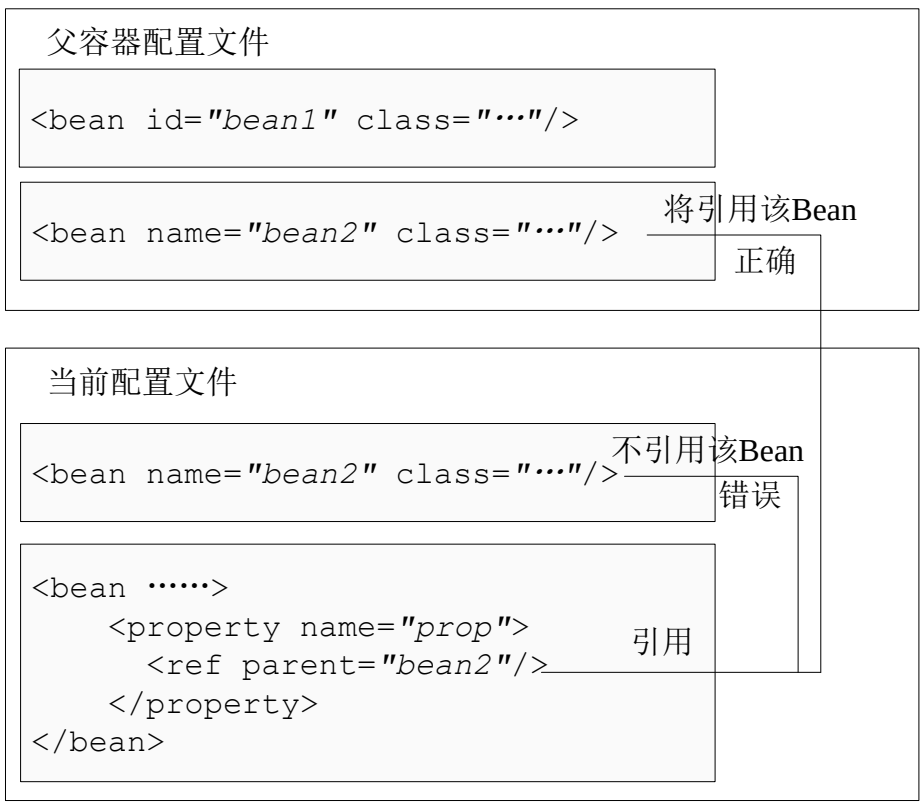
如果引用一个在当前配置文件中不存在的 Bean 将抛出如下异常：

org.springframework.beans.factory.xml.XmlBeanDefinitionStoreException: Line 21 in XML document from class path resource [chapter3/beanInject2.xml] is invalid; nested exception is org.xml.sax.SAXParseException: cvc-id.1: There is no ID/IDREF binding for IDREF 'helloApi'.

<ref local>具体配置方式如下：



（2）<ref parent=””/>配置方式：用于引用父容器中的 Bean，不会引用当前容器中的 Bean，当然父容器中的 Bean 和当前容器的 Bean 是可以重名的，获取顺序是先查找当前容器中的 Bean，如果找不到再从父容器找。具体配置方式如下：



接下来让我们用个例子演示一下<ref local>和<ref parent>的配置过程：

首先还是准备测试类,在此我们就使用以前写好的 HelloApiDecorator 和 HelloImpl4 类；其次进行 Bean 定义，其中当前容器 bean1 引用本地的”helloApi”，而”bean2”将引用父容器的”helloApi”，配置如下：

```
<!-- sources/chapter3/parentBeanInject.xml表示父容器配置-->
<!--注意此处可能子容器也定义一个该Bean-->
<bean id="helloApi" class="cn.javass.spring.chapter3.HelloImpl4">
    <property name="index" value="1"/>
    <property name="message" value="Hello Parent!"/>
</bean>
```

```
<!-- sources/chapter3/localBeanInject.xml表示当前容器配置-->
<!-- 注意父容器中也定义了id 为 helloApi的Bean -->
<bean id="helloApi" class="cn.javass.spring.chapter3.HelloImpl4">
    <property name="index" value="1"/>
    <property name="message" value="Hello Local!"/>
</bean>
<!-- 通过local注入 -->
<bean id="bean1" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
    <constructor-arg index="0"><ref local="helloApi"/></constructor-arg>
</bean>
<!-- 通过parent注入 -->
<bean id="bean2" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
    <property name="helloApi"><ref parent="helloApi"/></property>
</bean>
```

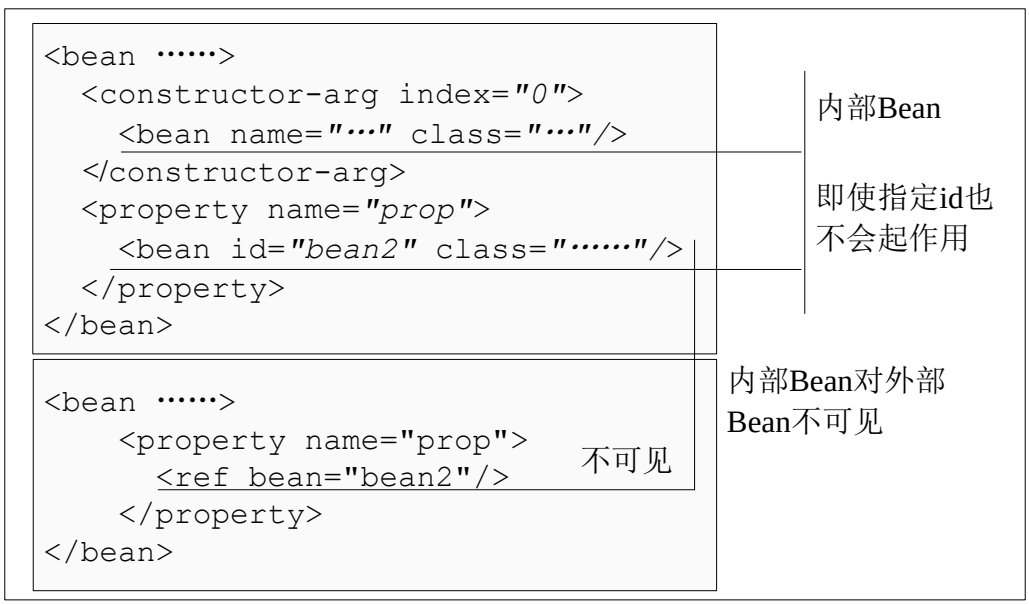
（3）写测试类测试一下吧，具体代码片段如下：

```
@Test
public void testLocalAndparentBeanInject() {
    //初始化父容器
    ApplicationContext parentBeanContext =
        new ClassPathXmlApplicationContext("chapter3/parentBeanInject.xml");
    //初始化当前容器
    ApplicationContext beanContext = new ClassPathXmlApplicationContext(
        new String[] {"chapter3/localBeanInject.xml"}, parentBeanContext);
    HelloApi bean1 = beanContext.getBean("bean1", HelloApi.class);
    bean1.sayHello();//该Bean引用local bean
    HelloApi bean2 = beanContext.getBean("bean2", HelloApi.class);
    bean2.sayHello();//该Bean引用parent bean
}
```

“bean1”将输出“Hello Local!”表示引用当前容器的 Bean，”bean2”将输出“Hello Paren!”，表示引用父容器的 Bean，如配置有问题请参考 cn.javass.spring.chapter3.DependencyInjectTest 中的 testLocalAndparentBeanInject 测试方法。

3.1.8 内部 Bean 定义

内部 Bean 就是在<property>或<constructor-arg>内通过<bean>标签定义的 Bean，该 Bean 不管是否指定 id 或 name，该 Bean 都会有唯一的匿名标识符，而且不能指定别名，该内部 Bean 对其他外部 Bean 不可见，具体配置如下：




```
<bean ……>
  <constructor-arg index="0">
    <bean name="…" class="…"/>
  </constructor-arg>
  <property name="prop">
    <bean id="bean2" class="……"/>
  </property>
</bean>
```

```
<bean ……>
    <property name="prop">
      <ref bean="bean2"/>
    </property>
</bean>
```

（1）让我们写个例子测试一下吧，具体配置文件如下：

```
<bean id="bean" class="cn.javass.spring.chapter3.bean.HelloApiDecorator">
  <property name="helloApi">
    <bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
  </property>
</bean>
```

（2）测试代码（cn.javass.spring.chapter3.DependencyInjectTest.testInnerBeanInject）：

```
@Test
public void testInnerBeanInject() {
    ApplicationContext context =
        new ClassPathXmlApplicationContext("chapter3/innerBeanInject.xml");
    HelloApi bean = context.getBean("bean", HelloApi.class);
    bean.sayHello();
}
```

3.1.9 处理 null 值

Spring 通过<value>标签或 value 属性注入常量值，所有注入的数据都是字符串，那如何注入 null 值呢？通过“null”值吗？当然不是因为如果注入“null”则认为是字符串。Spring 通过<null/>标签注入 null 值。即可以采用如下配置方式：

```
<bean class="...HelloImpl4">
   <property name="message"><null/></property>
   <property name="index" value="1"/>
</bean>
```

3.1.10 对象图导航注入支持

所谓对象图导航是指类似 a.b.c 这种点缀访问形式的访问或修改值。Spring 支持对象图导航方式依赖注入。对象图导航依赖注入有一个限制就是比如 a.b.c 对象导航图注入中 a 和 b 必须为非 null 值才能注入 c，否则将抛出空指针异常。

Spring 不仅支持对象的导航，还支持数组、列表、字典、Properties 数据类型的导航，对 Set 数据类型无法支持，因为无法导航。

数组和列表数据类型可以用 array[0]、list[1]导航，注意”[]”里的必须是数字，因为是按照索引进行导航，对于数组类型注意不要数组越界错误。

字典 Map 数据类型可以使用 map[1]、map[str]进行导航，其中“[]”里的是基本类型，无法放置引用类型。

让我们来练习一下吧。首先准备测试类，在此我们需要三个测试类，以便实现对象图导航功能演示：

NavigationC 类用于打印测试代码，从而观察配置是否正确；具体类如下所示：

```
package cn.javass.spring.chapter3.bean;
public class NavigationC {
      public void sayNavigation() {
            System.out.println("===navigation c");
      }
}
```

NavigationB 类，包含对象和列表、Properties、数组字典数据类型导航，而且这些复合数据类型保存的条目都是对象，正好练习一下如何往复合数据类型中注入对象依赖。具体类如下所示：

```
package cn.javass.spring.chapter3.bean;
import java.util.List;
import java.util.Map;
import java.util.Properties;
public class NavigationB {
    private NavigationC navigationC;
    private List<NavigationC> list;
    private Properties properties;
    private NavigationC[] array = new NavigationC[1];
    private Map<String, NavigationC> map;
    //由于setter和getter方法占用太多空间，故省略，大家自己实现吧
}
```

NavigationA 类是我们的前端类，通过对它的导航进行注入值，具体代码如下：

```
package cn.javass.spring.chapter3.bean;
public class NavigationA {
    private NavigationB navigationB;
    public void setNavigationB(NavigationB navigationB) {
        this.navigationB = navigationB;
    }
    public NavigationB getNavigationB() {
        return navigationB;
    }
}
```

接下来该进行 Bean 定义配置（resources/chapter3/navigationBeanInject.xml）了，首先让我们配置一下需要被导航的数据，NavigationC 和 NavigationB 类，其中配置 NavigationB 时注意要确保比如 array 字段不为空值，这就需要或者在代码中赋值如“NavigationC[] array = new NavigationC[1];”，或者通过配置文件注入如“<list></list>”注入一个不包含条目的列表。具体配置如下：

```
<bean id="c" class="cn.javass.spring.chapter3.bean.NavigationC"/>
<bean id="b" class="cn.javass.spring.chapter3.bean.NavigationB">
    <property name="list"><list></list></property>
    <property name="map"><map></map></property>
    <property name="properties"><props></props></property>
</bean>
```

配置完需要被导航的 Bean 定义了，该来配置 NavigationA 导航 Bean 了，在此需要注意，由于“navigationB”属性为空值，在此需要首先注入“navigationB”值；还有对于数组导航不能越界否则报错；具体配置如下：

```
<bean id="a" class="cn.javass.spring.chapter3.bean.NavigationA">
    <!-- 首先注入navigatiionB 确保它非空 -->
    <property name="navigationB" ref="b"/>
    <!-- 对象图导航注入 -->
    <property name="navigationB.navigationC" ref="c"/>
    <!-- 注入列表数据类型数据 -->
    <property name="navigationB.list[0]" ref="c"/>
    <!-- 注入map类型数据 -->
    <property name="navigationB.map[key]" ref="c"/>
    <!-- 注入properties类型数据 -->
    <property name="navigationB.properties[0]" ref="c"/>
    <!-- 注入properties类型数据 -->
    <property name="navigationB.properties[1]" ref="c"/>
    <!-- 注入数组类型数据 ，注意不要越界-->
    <property name="navigationB.array[0]" ref="c"/>
</bean>
```

配置完毕，具体测试代码在 cn.javass.spring.chapter3. DependencyInjectTest，让我们看下测试代码吧：

```
//对象图导航
@Test
public void testNavigationBeanInject() {
    ApplicationContext context =
    new ClassPathXmlApplicationContext("chapter3/navigationBeanInject.xml");
    NavigationA navigationA = context.getBean("a", NavigationA.class);
    navigationA.getNavigationB().getNavigationC().sayNavigation();
    navigationA.getNavigationB().getList().get(0).sayNavigation();
    navigationA.getNavigationB().getMap().get("key").sayNavigation();
    navigationA.getNavigationB().getArray()[0].sayNavigation();
    ((NavigationC)navigationA.getNavigationB().getProperties().get("1"))
    .sayNavigation();
}
```

测试完毕，应该输出 5 个“===navigation c”，是不是很简单，注意这种方式是不推荐使用的，了解一下就够了，最好使用 3.1.5 一节使用的配置方式。

3.1.11 配置简写

让我们来总结一下依赖注入配置及简写形式，其实我们已经在以上部分穿插着进行简化配置了：

一、构造器注入：

1）常量值

简写：<constructor-arg index="0" value="常量"/>

全写：<constructor-arg index="0"><value>常量</value></constructor-arg>

2）引用

简写：<constructor-arg index="0" ref="引用"/>

全写：<constructor-arg index="0"><ref bean="引用"/></constructor-arg>

二、setter 注入：

1）常量值

简写：<property name="message" value="常量"/>

全写：<property name="message"><value>常量</value></ property>

2）引用

简写：<property name="message" ref="引用"/>

全写：<property name="message"><ref bean="引用"/></ property>

3）数组：<array>没有简写形式

4）列表：<list>没有简写形式

5）集合：<set>没有简写形式

6）字典

简写：<map>

<entry key="键常量" value="值常量"/>

<entry key-ref="键引用" value-ref="值引用"/>

</map>

全写：<map>

<entry><key><value>键常量</value></key><value>值常量</value></entry>

<entry><key><ref bean="键引用"/></key><ref bean="值引用"/></entry>

</map>

7）Properties：没有简写形式

三、使用 p 命名空间简化 setter 注入：

使用 p 命名空间来简化 setter 注入，具体使用如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans    xmlns="http://www.springframework.org/schema/beans"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:p="http://www.springframework.org/schema/p"
          xsi:schemaLocation="
              http://www.springframework.org/schema/beans
              http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean id="bean1" class="java.lang.String">
        <constructor-arg index="0" value="test"/>
    </bean>
    <bean id="idrefBean1" class="cn.javass.spring.chapter3.bean.IdRefTestBean"
        p:id="value"/>
    <bean id="idrefBean2" class="cn.javass.spring.chapter3.bean.IdRefTestBean"
        p:id-ref="bean1"/>
</beans>
```

- **xmlns:p="http://www.springframework.org/schema/p" ：首先指定 p 命名空间；**
- **<bean id="……" class="……" p:id="value"/> ：**常量 setter 注入方式，其等价于<property name="id" value="value"/>；
- **<bean id="……" class="……" p:id-ref="bean1"/> ：**引用 setter 注入方式，其等价于<property name="id" ref="bean1"/>。

3.2 循环依赖

3.2.1 什么是循环依赖

循环依赖就是循环引用，就是两个或多个 Bean 相互之间的持有对方，比如 CircleA 引用 CircleB，CircleB 引用 CircleC，CircleC 引用 CircleA，则它们最终反映为一个环。此处不是循环调用，循环调用是方法之间的环调用。如图 3-5 所示：

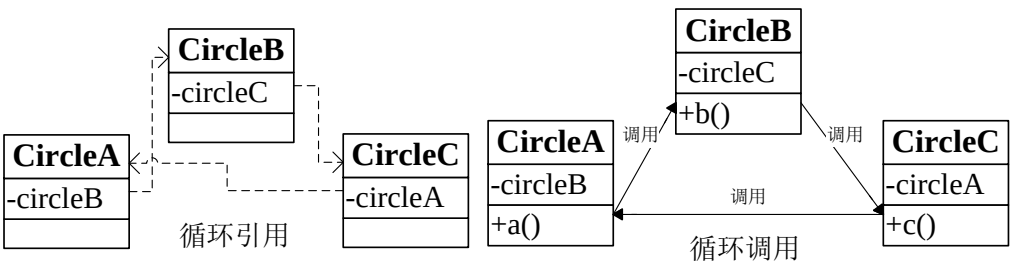


图 3-5 循环引用

循环调用是无法解决的，除非有终结条件，否则就是死循环，最终导致内存溢出错误。

Spring 容器循环依赖包括构造器循环依赖和 setter 循环依赖，那 Spring 容器如何解决循环依赖呢？首先让我们来定义循环引用类：

```
package cn.javass.spring.chapter3.bean;
public class CircleA {
    private CircleB circleB;
    public CircleA() {
    }
    public CircleA(CircleB circleB) {
        this.circleB = circleB;
    }
    public void setCircleB(CircleB circleB)
    {
        this.circleB = circleB;
    }
    public void a() {
        circleB.b();
    }
}
```

```
package cn.javass.spring.chapter3.bean;
public class CircleB {
    private CircleC circleC;
    public CircleB() {
    }
    public CircleB(CircleC circleC) {
        this.circleC = circleC;
    }
    public void setCircleC(CircleC circleC)
    {
        this.circleC = circleC;
    }
    public void b() {
        circleC.c();
    }
}
```

```
package cn.javass.spring.chapter3.bean;
public class CircleC {
    private CircleC circleC;
    public CircleB() {
    }
    public CircleB(CircleC circleC) {
        this.circleC = circleC;
    }
    public void setCircleC(CircleC circleC)
    {
        this.circleC = circleC;
    }
    public void b() {
        circleC.c();
    }
}
```

3.2.2 Spring 如何解决循环依赖

一、构造器循环依赖：表示通过构造器注入构成的循环依赖，此依赖是无法解决的，只能抛出 BeanCurrentlyInCreationException 异常表示循环依赖。

如在创建 CircleA 类时，构造器需要 CircleB 类，那将去创建 CircleB，在创建 CircleB 类时又发现需要 CircleC 类，则又去创建 CircleC，最终在创建 CircleC 时发现又需要 CircleA；从而形成一个环，没办法创建。

Spring 容器将每一个正在创建的 Bean 标识符放在一个“当前创建 Bean 池”中，Bean 标识符在创建过程中将一直保持在这个池中，因此如果在创建 Bean 过程中发现自己已经在“当前创建 Bean 池”里时将抛出 BeanCurrentlyInCreationException 异常表示循环依赖；而对于创建完毕的 Bean 将从“当前创建 Bean 池”中清除掉。

1）首先让我们看一下配置文件（chapter3/circleInjectByConstructor.xml）：

```
<bean id="circleA" class="cn.javass.spring.chapter3.bean.CircleA">
        <constructor-arg index="0" ref="circleB"/>
</bean>
<bean id="circleB" class="cn.javass.spring.chapter3.bean.CircleB">
      <constructor-arg index="0" ref="circleC"/>
</bean>
<bean id="circleC" class="cn.javass.spring.chapter3.bean.CircleC">
      <constructor-arg index="0" ref="circleA"/>
</bean>
```

2）写段测试代码（cn.javass.spring.chapter3.CircleTest）测试一下吧：

```
@Test(expected = BeanCurrentlyInCreationException.class)
public void testCircleByConstructor() throws Throwable {
      try {
         new ClassPathXmlApplicationContext("chapter3/circleInjectByConstructor.xml");
      }
      catch (Exception e) {
         //因为要在创建circle3时抛出；
         Throwable e1 = e.getCause().getCause().getCause();
         throw e1;
      }
}
```

让我们分析一下吧：

1、Spring 容器创建“circleA” Bean，首先去“当前创建 Bean 池”查找是否当前 Bean 正在创建，如果没发现，则继续准备其需要的构造器参数“circleB”，并将“circleA” 标

识符放到“当前创建 Bean 池”；

2、Spring 容器创建“circleB” Bean，首先去“当前创建 Bean 池”查找是否当前 Bean 正在创建，如果没发现，则继续准备其需要的构造器参数“circleC”，并将“circleB” 标识符放到“当前创建 Bean 池”；

3、Spring 容器创建“circleC” Bean，首先去“当前创建 Bean 池”查找是否当前 Bean 正在创建，如果没发现，则继续准备其需要的构造器参数“circleA”，并将“circleC” 标识符放到“当前创建 Bean 池”；

4、到此为止 Spring 容器要去创建“circleA”Bean，发现该 Bean 标识符在“当前创建 Bean 池”中，因为表示循环依赖，抛出 BeanCurrentlyInCreationException。

二、setter 循环依赖：表示通过 setter 注入方式构成的循环依赖。

对于 setter 注入造成的依赖是通过 Spring 容器提前暴露刚完成构造器注入但未完成其他步骤（如 setter 注入）的 Bean 来完成的，而且只能解决单例作用域的 Bean 循环依赖。

如下代码所示，通过提前暴露一个单例工厂方法，从而使其他 Bean 能引用到该 Bean。

```
addSingletonFactory(beanName, new ObjectFactory() {
        public Object getObject() throws BeansException {
            return getEarlyBeanReference(beanName, mbd, bean);
        }
});
```

具体步骤如下：

1、Spring 容器创建单例“circleA” Bean，首先根据无参构造器创建 Bean，并暴露一个“ObjectFactory ”用于返回一个提前暴露一个创建中的 Bean，并将“circleA” 标识符放到“当前创建 Bean 池”；然后进行 setter 注入“circleB”；

2、Spring 容器创建单例“circleB” Bean，首先根据无参构造器创建 Bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的 Bean，并将“circleB” 标识符放到“当前创建 Bean 池”，然后进行 setter 注入“circleC”；

3、Spring 容器创建单例“circleC” Bean，首先根据无参构造器创建 Bean，并暴露一个“ObjectFactory ”用于返回一个提前暴露一个创建中的 Bean，并将“circleC” 标识符放到“当前创建 Bean 池”，然后进行 setter 注入“circleA”；进行注入“circleA”时由于提前暴露了“ObjectFactory”工厂从而使用它返回提前暴露一个创建中的 Bean；

4、最后在依赖注入“circleB”和“circleA”，完成 setter 注入。

对于“prototype”作用域 Bean，Spring 容器无法完成依赖注入，因为“prototype”作用域的 Bean，Spring 容器不进行缓存，因此无法提前暴露一个创建中的 Bean。

```
<!-- 定义 Bean 配置文件，注意 scope 都是“prototype”-->
<bean id="circleA" class="cn.javass.spring.chapter3.bean.CircleA" scope="prototype">
        <property name="circleB" ref="circleB"/>
    </bean>
    <bean id="circleB" class="cn.javass.spring.chapter3.bean.CircleB" scope="prototype">
        <property name="circleC" ref="circleC"/>
    </bean>
    <bean id="circleC" class="cn.javass.spring.chapter3.bean.CircleC" scope="prototype">
```

```
//测试代码cn.javass.spring.chapter3.CircleTest
@Test(expected = BeanCurrentlyInCreationException.class)
public void testCircleBySetterAndPrototype () throws Throwable {
    try {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(
            "chapter3/circleInjectBySetterAndPrototype.xml");
        System.out.println(ctx.getBean("circleA"));
    }
    catch (Exception e) {
        Throwable e1 = e.getCause().getCause().getCause();
        throw e1;
    }
}
```

对于“singleton”作用域 Bean，可以通过“setAllowCircularReferences(false);”来禁用循环引用：

```
@Test(expected = BeanCurrentlyInCreationException.class)
public void testCircleBySetterAndSingleton2() throws Throwable {
    try {
        ClassPathXmlApplicationContext ctx =
            new ClassPathXmlApplicationContext();
        ctx.setConfigLocation("chapter3/circleInjectBySetterAndSingleton.xml");
        ctx.refresh();
    }
    catch (Exception e) {
        Throwable e1 = e.getCause().getCause().getCause();
        throw e1;
    }
}
```

3.3 更多 DI 的知识

3.3.1 延迟初始化 Bean

延迟初始化也叫做惰性初始化，指不提前初始化 Bean，而是只有在真正使用时才创建及初始化 Bean。

配置方式很简单只需在<bean>标签上指定 “lazy-init” 属性值为“true”即可延迟初始化 Bean。

Spring 容器会在创建容器时提前初始化“singleton”作用域的 Bean，“singleton”就是单例的意思即整个容器每个 Bean 只有一个实例，后边会详细介绍。Spring 容器预先初始化 Bean 通常能帮助我们提前发现配置错误，所以如果没有什么情况建议开启，除非有某个 Bean 可能需要加载很大资源，而且很可能在整个应用程序生命周期中很可能使用不到，可以设置为延迟初始化。

延迟初始化的 Bean 通常会在第一次使用时被初始化；或者在被非延迟初始化 Bean 作为依赖对象注入时在会随着初始化该 Bean 时被初始化，因为在这时使用了延迟初始化 Bean。

容器管理初始化 Bean 消除了编程实现延迟初始化，完全由容器控制，只需在需要延迟初始化的 Bean 定义上配置即可，比编程方式更简单，而且是无侵入代码的。

具体配置如下：

```
<bean id="helloApi"
        class="cn.javass.spring.chapter2.helloworld.HelloImpl"
        lazy-init="true"/>
```

3.3.2 使用 depends-on

depends-on 是指指定 Bean 初始化及销毁时的顺序，使用 depends-on 属性指定的 Bean 要先初始化完毕后才初始化当前 Bean，由于只有“singleton”Bean 能被 Spring 管理销毁，所以当指定的 Bean 都是“singleton”时，使用 depends-on 属性指定的 Bean 要在指定的 Bean 之后销毁。

> 注：文档中说销毁 Bean 的顺序：Dependent beans that define a depends-on relationship with a given bean aredestroyed first, prior to the given bean itself being destroyed.
> 意思是：在 depends-on 属性中定义的“依赖 Bean”要在定义该属性的 Bean 之前销毁。
>
> 但实际是错误的，定义“depends-on”属性的 Bean 会首先被销毁，然后才是“depends-on”指定的 Bean 被销毁。大家可以试验一下。

配置方式如下：

```
<bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<bean id="decorator"
      class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
      depends-on="helloApi">
  <property name="helloApi"><ref bean="helloApi"/></property>
</bean>
```

“decorator”指定了“depends-on”属性为“helloApi”，所以在“decorator”Bean 初始化之前要先初始化“helloApi”，而在销毁“helloApi”之前先要销毁“decorator”，大家注意一下销毁顺序，与文档上的不符。

“depends-on”属性可以指定多个 Bean，若指定多个 Bean 可以用“;”、“,”、空格分割。

那“depends-on”有什么好处呢？主要是给出明确的初始化及销毁顺序，比如要初始化“decorator”时要确保“helloApi”Bean 的资源准备好了，否则使用“decorator”时会看不到准备的资源；而在销毁时要先在“decorator”Bean 的把对“helloApi”资源的引用释放掉才能销毁“helloApi”，否则可能销毁 “helloApi”时而“decorator”还保持着资源访问，造成资源不能释放或释放错误。

让我们看个例子吧，在平常开发中我们可能需要访问文件系统，而文件打开、关闭是必须配对的，不能打开后不关闭，从而造成其他程序不能访问该文件。让我们来看具体配置吧：

1）准备测试类：

ResourceBean 从配置文件中配置文件位置，然后定义初始化方法 init 中打开指定的文件，然后获取文件流；最后定义销毁方法 destroy 用于在应用程序关闭时调用该方法关闭掉文件流。

DependentBean 中会注入 ResourceBean，并从 ResourceBean 中获取文件流写入内容；定义初始化方法 init 用来定义一些初始化操作并向文件中输出文件头信息；最后定义销毁方法用于在关闭应用程序时想文件中输出文件尾信息。

具体代码如下：

```
package cn.javass.spring.chapter3.bean;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
public class ResourceBean {
    private FileOutputStream fos;
    private File file;
    //初始化方法
    public void init() {
        System.out.println("ResourceBean:========初始化");
        //加载资源,在此只是演示
        System.out.println("ResourceBean:========加载资源，执行一些预操作");
        try {
            this.fos = new FileOutputStream(file);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
    //销毁资源方法
    public void destroy() {
        System.out.println("ResourceBean:========销毁");
        //释放资源
        System.out.println("ResourceBean:========释放资源，执行一些清理操作");
        try {
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public FileOutputStream getFos() {
        return fos;
    }
    public void setFile(File file) {
        this.file = file;
    }
}
```

```
package cn.javass.spring.chapter3.bean;
import java.io.IOException;
public class DependentBean {
    ResourceBean resourceBean;
    public void write(String ss) throws IOException {
        System.out.println("DependentBean:=======写资源");
        resourceBean.getFos().write(ss.getBytes());
    }
    //初始化方法
    public void init() throws IOException {
        System.out.println("DependentBean:=======初始化");
        resourceBean.getFos().write("DependentBean:=======初始化=====".getBytes());
    }
    //销毁方法
    public void destroy() throws IOException {
        System.out.println("DependentBean:=======销毁");
        //在销毁之前需要往文件中写销毁内容
        resourceBean.getFos().write("DependentBean:=======销毁=====".getBytes());
    }

    public void setResourceBean(ResourceBean resourceBean) {
        this.resourceBean = resourceBean;
    }
}
```

2）类定义好了，让我们来进行 Bean 定义吧，具体配置文件如下：

```
<bean id="resourceBean"
        class="cn.javass.spring.chapter3.bean.ResourceBean"
        init-method="init" destroy-method="destroy">
  <property name="file" value="D:/test.txt"/>
</bean>
<bean id="dependentBean"
      class="cn.javass.spring.chapter3.bean.DependentBean"
      init-method="init" destroy-method="destroy" depends-on="resourceBean">
  <property name="resourceBean" ref="resourceBean"/>
</bean>
```

<property name="file" value="D:/test.txt"/>配置：Spring 容器能自动把字符串转换为 java.io.File。

init-method="init" ：指定初始化方法，在构造器注入和 setter 注入完毕后执行。

destroy-method="destroy"：指定销毁方法，只有“singleton”作用域能销毁，“prototype”作用域的一定不能，其他作用域不一定能；后边再介绍。

在此配置中，dependentBean 初始化在 resourceBean 之前被初始化，resourceBean 销毁会在 dependentBean 销毁之后执行。

3）配置完毕，测试一下吧：

```
package cn.javass.spring.chapter3;
import java.io.IOException;
import org.junit.Test;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import cn.javass.spring.chapter3.bean.DependentBean;
public class MoreDependencyInjectTest {
    @Test
    public void testDependOn() throws IOException {
        ClassPathXmlApplicationContext context =
            new ClassPathXmlApplicationContext("chapter3/depends-on.xml");
        //一点要注册销毁回调，否则我们定义的销毁方法不执行
        context.registerShutdownHook();
        DependentBean dependentBean =
                context.getBean("dependentBean", DependentBean.class);
        dependentBean.write("aaa");
    }
}
```

测试跟其他测试完全一样，只是在此我们一定要注册销毁方法回调，否则销毁方法不会执行。

如果配置没问题会有如下输出：

```
ResourceBean:========初始化
ResourceBean:========加载资源，执行一些预操作
DependentBean:=========初始化
DependentBean:=========写资源
DependentBean:=========销毁
ResourceBean:========销毁
ResourceBean:========释放资源，执行一些清理操作
```

3.3.3 自动装配

自动装配就是指由 Spring 来自动地注入依赖对象，无需人工参与。

目前 Spring3.0 支持“no”、“byName ”、“byType”、“constructor”四种自动装配，默认是“no”指不支持自动装配的，其中 Spring3.0 已不推荐使用之前版本的“autodetect”自动装配，推荐使用 Java 5+支持的（@Autowired）注解方式代替；如果想支持“autodetect”自动装配，请将 schema 改为“spring-beans-2.5.xsd”或去掉。

自动装配的好处是减少构造器注入和 setter 注入配置，减少配置文件的长度。自动装配通过配置<bean>标签的“autowire”属性来改变自动装配方式。接下来让我们挨着看下配置的含义。

一、**default**：表示使用默认的自动装配，默认的自动装配需要在<beans>标签中使用 default-autowire 属性指定，其支持“no”、“byName ”、“byType”、“constructor”四种自动装配，如果需要覆盖默认自动装配，请继续往下看；

二、**no**：意思是不支持自动装配，必须明确指定依赖。

三、**byName**：通过设置 Bean 定义属性 autowire="byName"，意思是根据名字进行自动装配，只能用于 setter 注入。比如我们有方法“setHelloApi”，则“byName”方式 Spring 容器将查找名字为 helloApi 的 Bean 并注入，如果找不到指定的 Bean，将什么也不注入。

例如如下 Bean 定义配置：

```
<bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<bean id="bean" class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
      autowire="byName"/>
```

测试代码如下：

```
package cn.javass.spring.chapter3;
import java.io.IOException;
import org.junit.Test;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import cn.javass.spring.chapter2.helloworld.HelloApi;
public class AutowireBeanTest {
    @Test
    public void testAutowireByName() throws IOException {
        ClassPathXmlApplicationContext context =
            new ClassPathXmlApplicationContext("chapter3/autowire-byName.xml");
        HelloApi helloApi = context.getBean("bean", HelloApi.class);
        helloApi.sayHello();
    }
}
```

是不是不要配置<property>了，如果一个 bean 有很多 setter 注入，通过“byName”方式是不是能减少很多<property>配置。此处注意了，**在根据名字注入时，将把当前 Bean 自己排除在外**：比如“hello”Bean 类定义了“setHello”方法，则 hello 是不能注入到“setHello”的。

四、“byType”：通过设置 Bean 定义属性 autowire="byType"，意思是指根据类型注入，用于 setter 注入，比如如果指定自动装配方式为“byType”，而“setHelloApi”方法需要注入 HelloApi 类型数据，则 Spring 容器将查找 HelloApi 类型数据，如果找到一个则注入该 Bean，如果找不到将什么也不注入，如果找到多个 Bean 将优先注入<bean>标签“primary”属性为 true 的 Bean，否则抛出异常来表明有个多个 Bean 发现但不知道使用哪个。让我们用例子来讲解一下这几种情况吧。

1）根据类型只找到一个 Bean，此处注意了，**在根据类型注入时，将把当前 Bean 自己排除在外**，即如下配置中 helloApi 和 bean 都是 HelloApi 接口的实现，而“bean”通过类型进行注入“HelloApi”类型数据时自己是排除在外的，配置如下（具体测试请参考 AutowireBeanTest.testAutowireByType1 方法）：

```
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<bean id="bean" class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
      autowire="byType"/>
```

2）根据类型找到多个 Bean 时，对于集合类型（如 List、Set）将注入所有匹配的候选者，而对于其他类型遇到这种情况可能需要使用“autowire-candidate”属性为 false 来让指定的 Bean 放弃作为自动装配的候选者，或使用“primary”属性为 true 来指定某个 Bean 为首选 Bean：

2.1）通过设置 Bean 定义的“autowire-candidate”属性为 false 来把指定 Bean 后自动装配候选者中移除：

```
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<!-- 从自动装配候选者中去除 -->
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"
      autowire-candidate="false"/>
<bean id="bean1" class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
      autowire="byType"/>
```

2.2）通过设置 Bean 定义的“primary”属性为 false 来把指定自动装配时候选者中首选 Bean：

```
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<!-- 自动装配候选者中的首选Bean-->
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl" primary="true"/>
<bean id="bean" class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
      autowire="byType"/>
```

具体测试请参考 AutowireBeanTest 类的 testAutowireByType***方法。

五、“constructor”：通过设置 Bean 定义属性 autowire="constructor"，功能和“byType”功能一样，根据类型注入构造器参数，只是用于构造器注入方式，直接看例子吧：

```
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<!-- 自动装配候选者中的首选Bean-->
<bean class="cn.javass.spring.chapter2.helloworld.HelloImpl" primary="true"/>
<bean id="bean"
      class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
      autowire="constructor"/>
```

测试代码如下：

```
@Test
public void testAutowireByConstructor() throws IOException {
    ClassPathXmlApplicationContext context =
        new ClassPathXmlApplicationContext("chapter3/autowire-byConstructor.xml");
    HelloApi helloApi = context.getBean("bean", HelloApi.class);
    helloApi.sayHello();
}
```

六、autodetect：自动检测是使用“constructor”还是“byType”自动装配方式，已不推荐使用。如果 Bean 有空构造器那么将采用“byType”自动装配方式，否则使用“constructor”自动装配方式。此处要把 3.0 的 xsd 替换为 2.5 的 xsd，否则会报错。

```
<?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:context="http://www.springframework.org/schema/context"
          xsi:schemaLocation="
              http://www.springframework.org/schema/beans
              http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
              http://www.springframework.org/schema/context
              http://www.springframework.org/schema/context/spring-context-2.5.xsd">
  <bean class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
  <!-- 自动装配候选者中的首选Bean-->
  <bean class="cn.javass.spring.chapter2.helloworld.HelloImpl" primary="true"/>
  <bean id="bean"
          class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
          autowire="autodetect"/>
</beans>
```

可以采用在“<beans>”标签中通过“default-autowire”属性指定全局的自动装配方式，即如果 default-autowire=”byName”，将对所有 Bean 进行根据名字进行自动装配。

不是所有类型都能自动装配：

- 不能自动装配的数据类型：Object、基本数据类型（Date、CharSequence、Number、URI、URL、Class、int）等；
- 通过“<beans>”标签 default-autowire-candidates 属性指定的匹配模式，不匹配的将不能作为自动装配的候选者，例如指定“*Service，*Dao”，将只把匹配这些模式的 Bean 作为候选者，而不匹配的不会作为候选者；
- 通过将“<bean>”标签的 autowire-candidate 属性可被设为 false，从而该 Bean 将不会作为依赖注入的候选者。

数组、集合、字典类型的根据类型自动装配和普通类型的自动装配是有区别的：

- **数组类型、集合（Set、Collection、List）接口类型**：将根据泛型获取匹配的所有候选者并注入到数组或集合中，如“List<HelloApi> list”将选择所有的 HelloApi 类型 Bean 并注入到 list 中，而对于集合的具体类型将只选择一个候选者，“如 ArrayList<HelloApi> list”将选择一个类型为 ArrayList 的 Bean 注入，而不是选择所有的 HelloApi 类型 Bean 进行注入；
- **字典（Map）接口类型：**同样根据泛型信息注入，键必须为 String 类型的 Bean 名字，值根据泛型信息获取，如“Map<String, HelloApi> map” 将选择所有的 HelloApi 类型 Bean 并注入到 map 中，而对于具体字典类型如“HashMap<String, HelloApi> map”将只选择类型为 HashMap 的 Bean 注入，而不是选择所有的 HelloApi 类型 Bean 进行注入。

自动装配我们已经介绍完了，自动装配能带给我们什么好处呢？首先，自动装配确实减少了配置文件的量；其次， “byType”自动装配能在相应的 Bean 更改了字段类型时自动更新，即修改 Bean 类不需要修改配置，确实简单了。

自动装配也是有缺点的，最重要的缺点就是没有了配置，在查找注入错误时非常麻烦，还有比如基本类型没法完成自动装配，所以可能经常发生一些莫名其妙的错误，在此我推荐大家不要使用该方式，最好是指定明确的注入方式，或者采用最新的 Java5+注解注入方式。所以大家在使用自动装配时应该考虑自己负责项目的复杂度来进行衡量是否选择自动装配方式。

自动装配注入方式能和配置注入方式一同工作吗？当然可以，大家只需记住**配置注入的数据会覆盖自动装配注入的数据。**

大家是否注意到对于采用自动装配方式时如果没找到合适的的 Bean 时什么也不做，这样在程序中总会莫名其妙的发生一些空指针异常，而且是在程序运行期间才能发现，有没有办法能在提前发现这些错误呢？接下来就让我来看下依赖检查吧。

3.3.4 依赖检查

上一节介绍的自动装配，很可能发生没有匹配的 Bean 进行自动装配，如果此种情况发生，只有在程序运行过程中发生了空指针异常才能发现错误，如果能提前发现该多好啊，这就是依赖检查的作用。

依赖检查：用于检查 Bean 定义的属性都注入数据了，不管是自动装配的还是配置方式注入的都能检查，如果没有注入数据将报错，从而提前发现注入错误，只检查具有 setter 方法的属性。

Spring3+也不推荐配置方式依赖检查了，建议采用 Java5+ @Required 注解方式，测试时请将 XML schema 降低为 2.5 版本的，和自动装配中“autodetect”配置方式的 xsd 一样。

```
<?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-2.5.xsd
</beans>
```

依赖检查有 none、simple、object、all 四种方式，接下来让我们详细介绍一下：

一、none：默认方式，表示不检查；

二、objects：检查除基本类型外的依赖对象，配置方式为：dependency-check="objects"，此处我们为 HelloApiDecorator 添加一个 String 类型属性“message”，来测试如果有简单数据类型的属性为 null，也不报错；

```
<bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<!-- 注意我们没有注入helloApi，所以测试时会报错 -->
<bean id="bean"
      class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
      dependency-check="objects">
  <property name="message" value="Haha"/>
</bean>
```

注意由于我们没有注入 bean 需要的依赖“helloApi”，所以应该抛出异常 UnsatisfiedDependencyException，表示没有发现满足的依赖：

```
package cn.javass.spring.chapter3;
import java.io.IOException;
import org.junit.Test;
import org.springframework.beans.factory.UnsatisfiedDependencyException;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class DependencyCheckTest {
    @Test(expected = UnsatisfiedDependencyException.class)
    public void testDependencyCheckByObject() throws IOException {
        //将抛出异常
        new ClassPathXmlApplicationContext("chapter3/dependency-check-object.xml");
```

三、**simple**：对基本类型进行依赖检查，包括数组类型，其他依赖不报错；配置方式为：dependency-check="simple"，以下配置中没有注入 message 属性，所以会抛出异常：

```
<bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<!-- 注意我们没有注入message属性，所以测试时会报错 -->
<bean id="bean"
        class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
        dependency-check="simple">
    <property name="helloApi" ref="helloApi"/>
</bean>
```

四、**all**：对所以类型进行依赖检查，配置方式为：dependency-check="all"，如下配置方式中如果两个属性其中一个没配置将报错。

```
<bean id="helloApi" class="cn.javass.spring.chapter2.helloworld.HelloImpl"/>
<bean id="bean"
        class="cn.javass.spring.chapter3.bean.HelloApiDecorator"
        dependency-check="all">
    <property name="helloApi" ref="helloApi"/>
    <property name="message" value="Haha"/>
</bean>
```

依赖检查也可以通过“<beans>”标签中 default-dependency-check 属性来指定全局依赖检查配置。

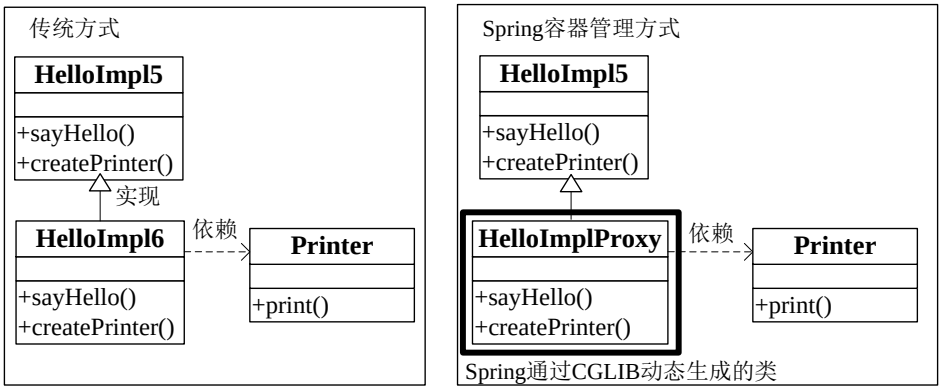
3.3.5 方法注入

所谓方法注入其实就是通过配置方式覆盖或拦截指定的方法，通常通过代理模式实现。Spring 提供两种方法注入：查找方法注入和方法替换注入。

因为 Spring 是通过 CGLIB 动态代理方式实现方法注入，也就是通过动态修改类的字节码来实现的，本质就是生成需方法注入的类的子类方式实现。

在进行测试之前，我们需要确保将“com.springsource.cn.sf.cglib-2.2.0.jar”放到 lib 里并添加到“Java Build Path”中的 Libararies 中。否则报错，异常中包含**“nested exception is**

java.lang.NoClassDefFoundError: cn/sf/cglib/proxy/CallbackFilter”。



传统方式和 Spring 容器管理方式唯一不同的是不需要我们手动生成子类，而是通过配置方式来实现；其中如果要替换 createPrinter()方法的返回值就使用查找方法注入；如果想完全替换 sayHello()方法体就使用方法替换注入。 接下来让我们看看具体实现吧。

一、查找方法注入：又称为 Lookup 方法注入，用于注入方法返回结果，也就是说能通过配置方式替换方法返回结果。使用<lookup-method name="方法名" bean="bean 名字"/>配置；其中 name 属性指定方法名，bean 属性指定方法需返回的 Bean。

方法定义格式：访问级别必须是 public 或 protected，保证能被子类重载，可以是抽象方法，必须有返回值，必须是无参数方法，查找方法的类和被重载的方法必须为非 final：

<public|protected> [abstract] <return-type> theMethodName(no-arguments);

因为“singleton”Bean 在容器中只有一个实例，而“prototype”Bean 是每次获取容器都返回一个全新的实例，所以如果“singleton”Bean 在使用“prototype” Bean 情况时，那么“prototype”Bean 由于是“singleton”Bean 的一个字段属性，所以获取的这个“prototype”Bean 就和它所在的“singleton”Bean 具有同样的生命周期，所以不是我们所期待的结果。因此查找方法注入就是用于解决这个问题。

1） 首先定义我们需要的类，Printer 类是一个有状态的类，counter 字段记录访问次数：

```
package cn.javass.spring.chapter3.bean;
public class Printer {
    private int counter = 0;
    public void print(String type) {
        System.out.println(type + " printer: " + counter++);
    }
}
```

HelloImpl5 类用于打印欢迎信息，其中包括 setter 注入和方法注入，此处特别需要注意的是该类是抽象的，充分说明了需要容器对其进行子类化处理，还定义了一个抽象方法 createPrototypePrinter 用于创建“prototype”Bean，createSingletonPrinter 方法用于创建“singleton”Bean，此处注意方法会被 Spring 拦截，不会执行方法体代码：

```
package cn.javass.spring.chapter3;
import cn.javass.spring.chapter2.helloworld.HelloApi;
import cn.javass.spring.chapter3.bean.Printer;
public abstract class HelloImpl5 implements HelloApi {
    private Printer printer;
    public void sayHello() {
        printer.print("setter");
        createPrototypePrinter().print("prototype");
    }
    public abstract Printer createPrototypePrinter();
    public Printer createSingletonPrinter() {
        System.out.println("该方法不会被执行，如果输出就错了");
        return new Printer();
    }
    public void setPrinter(Printer printer) {
        this.printer = printer;
    }
}
```

2）开始配置了，配置文件在（resources/chapter3/lookupMethodInject.xml），其中“prototypePrinter”是“prototype”Printer，“singletonPrinter”是“singleton”Printer，“helloApi1”是“singleton”Bean，而“helloApi2”注入了“prototype”Bean：

```
<bean id="prototypePrinter"
    class="cn.javass.spring.chapter3.bean.Printer" scope="prototype"/>
<bean id="singletonPrinter"
    class="cn.javass.spring.chapter3.bean.Printer" scope="singleton"/>
<bean id="helloApi1" class="cn.javass.spring.chapter3.HelloImpl5" scope="singleton">
    <property name="printer" ref="prototypePrinter"/>
    <lookup-method name="createPrototypePrinter" bean="prototypePrinter"/>
    <lookup-method name="createSingletonPrinter" bean="singletonPrinter"/>
</bean>
<bean id="helloApi2" class="cn.javass.spring.chapter3.HelloImpl5" scope="prototype">
    <property name="printer" ref="prototypePrinter"/>
    <lookup-method name="createPrototypePrinter" bean="prototypePrinter"/>
    <lookup-method name="createSingletonPrinter" bean="singletonPrinter"/>
</bean>
```

3）测试代码如下：

```
package cn.javass.spring.chapter3;
import org.junit.Test;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import cn.javass.spring.chapter2.helloworld.HelloApi;
public class MethodInjectTest {
    @Test
    public void testLookup() {
        ClassPathXmlApplicationContext context =
          new ClassPathXmlApplicationContext("chapter3/lookupMethodInject.xml");
        System.out.println("=======singleton sayHello======");
        HelloApi helloApi1 = context.getBean("helloApi1", HelloApi.class);
        helloApi1.sayHello();
        helloApi1 = context.getBean("helloApi1", HelloApi.class);
        helloApi1.sayHello();
        System.out.println("=======prototype sayHello======");
        HelloApi helloApi2 = context.getBean("helloApi2", HelloApi.class);
        helloApi2.sayHello();
        helloApi2 = context.getBean("helloApi2", HelloApi.class);
        helloApi2.sayHello();
    }}
```

其中“helloApi1”测试中，其输出结果如下：

```
=======singleton sayHello======
setter printer: 0
prototype printer: 0
singleton printer: 0
setter printer: 1
prototype printer: 0
singleton printer: 1
```

首先“helloApi1”是“singleton”，通过 setter 注入的“printer”是“prototypePrinter”，所以它应该输出“setter printer: 0”和“setter printer: 1”；而“createPrototypePrinter”方法注入了“prototypePrinter”，所以应该输出两次“prototype printer: 0”；而“createSingletonPrinter”注入了“singletonPrinter”，所以应该输出“singleton printer: 0”和“singleton printer: 1”。

而“helloApi2”测试中，其输出结果如下：

```
=======prototype sayHello======
setter printer: 0
prototype printer: 0
singleton printer: 2
setter printer: 0
prototype printer: 0
singleton printer: 3
```

首先“helloApi2”是“prototype”，通过 setter 注入的“printer”是“prototypePrinter”，所以它应该输出两次“setter printer: 0”；而“createPrototypePrinter”方法注入了“prototypePrinter”，所以应该输出两次“prototype printer: 0”；而“createSingletonPrinter”注入了“singletonPrinter”，所以应该输出“singleton printer: 2”和“singleton printer: 3”。

大家是否注意到“createSingletonPrinter”方法应该输出“该方法不会被执行，如果输出就错了”，而实际是没输出的，这说明 Spring 拦截了该方法并使用注入的 Bean 替换了返回结果。

方法注入主要用于处理“singleton”作用域的 Bean 需要其他作用域的 Bean 时，采用 Spring 查找方法注入方式无需修改任何代码即能获取需要的其他作用域的 Bean。

二、替换方法注入：也叫“MethodReplacer”注入，和查找注入方法不一样的是，他主要用来替换方法体。通过首先定义一个MethodReplacer接口实现，然后如下配置来实现：

```
<replaced-method name="方法名" replacer="MethodReplacer实现">
   <arg-type>参数类型</arg-type>
</replaced-method>”
```

1）首先定义MethodReplacer实现，完全替换掉被替换方法的方法体及返回值，其中reimplement方法重定义方法功能，参数obj为被替换方法的对象，method为被替换方法，args为方法参数；最需要注意的是不能再 通过“method.invoke(obj, new String[]{"hehe"});” 反射形式再去调用原来方法，这样会产生循环调用；如果返回值类型为Void，请在实现中返回null：

```
package cn.javass.spring.chapter3.bean;
import java.lang.reflect.Method;
import org.springframework.beans.factory.support.MethodReplacer;
public class PrinterReplacer implements MethodReplacer {
    @Override
    public Object reimplement(Object obj, Method method, Object[] args)
    throws Throwable {
        System.out.println("Print Replacer");
        //注意此处不能再通过反射调用了,否则会产生循环调用，知道内存溢出
        //method.invoke(obj, new String[]{"hehe"});
        return null;
    }
```

2）配置如下，首先定义 MethodReplacer 实现，使用< replaced-method >标签来指定要进行替换方法，属性 name 指定替换的方法名字，replacer 指定该方法的重新实现者，子标签< arg-type >用来指定原来方法参数的类型，必须指定否则找不到原方法：

```
<bean id="replacer" class="cn.javass.spring.chapter3.bean.PrinterReplacer"/>
<bean id="printer" class="cn.javass.spring.chapter3.bean.Printer">
    <replaced-method name="print" replacer="replacer">
        <arg-type>java.lang.String</arg-type>
    </replaced-method>
</bean>
```

3）测试代码将输出“`Print Replacer `”，说明方法体确实被替换了：

```
@Test
public void testMethodReplacer() {
    ClassPathXmlApplicationContext context =
    new ClassPathXmlApplicationContext("chapter3/methodReplacerInject.xml");
    Printer printer = context.getBean("printer", Printer.class);
    printer.print("我将被替换");
}
```

3.4 Bean 的作用域

什么是作用域呢？即“scope”，在面向对象程序设计中一般指对象或变量之间的可见范围。而在 Spring 容器中是指其创建的 Bean 对象相对于其他 Bean 对象的请求可见范围。

Spring 提供“singleton”和“prototype”两种基本作用域，另外提供“request”、“session”、“global session”三种 web 作用域；Spring 还允许用户定制自己的作用域。

3.4.1 基本的作用域

一、singleton：指“singleton”作用域的 Bean 只会在每个 Spring IoC 容器中存在一个实例，而且其完整生命周期完全由 Spring 容器管理。对于所有获取该 Bean 的操作 Spring 容器将只返回同一个 Bean。

> GoF 单例设计模式指“**保证一个类仅有一个实例，并提供一个访问它的全局访问点**”，介绍了两种实现：通过在类上定义静态属性保持该实例和通过注册表方式。

1）通过在类上定义静态属性保持该实例：一般指一个 Java 虚拟机 ClassLoader 装载的类只有一个实例，一般通过类静态属性保持该实例，这样就造成需要单例的类都需要按照单例设计模式进行编码；Spring 没采用这种方式，因为该方式属于侵入式设计；代码样例如下：

```
package cn.javass.spring.chapter3.bean;
public class Singleton {
    //1.私有化构造器
    private Singleton() {}
    //2.单例缓存者，惰性初始化，第一次使用时初始化
    private static class InstanceHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    //3.提供全局访问点
    public static Singleton getInstance() {
        return InstanceHolder.INSTANCE;
    }
    //4.提供一个计数器来验证一个ClassLoader一个实例
    private int counter=0;
}
```

以上定义个了个单例类，首先要私有化类构造器；其次使用 InstanceHolder 静态内部类持有单例对象，这样可以得到惰性初始化好处；最后提供全局访问点 getInstance，使得需要该单例实例的对象能获取到；我们在此还提供了一个 counter 计数器来验证一个 ClassLoader 一个实例。具体一个 ClassLoader 有一个单例实例测试请参考代码“cn.javass.spring.chapter3. SingletonTest”中的“testSingleton”测试方法，里边详细演示了一个 ClassLoader 有一个单例实例。

3）**通过注册表方式：** 首先将需要单例的实例通过唯一键注册到注册表，然后通过键来获取单例，让我们直接看实现吧，注意本注册表实现了 Spring 接口“SingletonBeanRegistry”，该接口定义了操作共享的单例对象，Spring 容器实现将实现此接口；所以共享单例对象通过“registerSingleton”方法注册，通过“getSingleton”方法获取，消除了编程方式单例，注意在实现中不考虑并发：

```
package cn.javass.spring.chapter3;
import java.util.HashMap;
import java.util.Map;
import org.springframework.beans.factory.config.SingletonBeanRegistry;
public class SingletonBeanRegister implements SingletonBeanRegistry {
    //单例 Bean 缓存池，此处不考虑并发
    private final Map<String, Object> BEANS = new HashMap<String, Object>();
    public boolean containsSingleton(String beanName) {
        return BEANS.containsKey(beanName);
    }
    public Object getSingleton(String beanName) {
        return BEANS.get(beanName);
    }
    @Override
    public int getSingletonCount() {
        return BEANS.size();
    }
    @Override
    public String[] getSingletonNames() {
        return BEANS.keySet().toArray(new String[0]);
    }
    @Override
    public void registerSingleton(String beanName, Object bean) {
        if(BEANS.containsKey(beanName)) {
            throw new RuntimeException("[" + beanName + "] 已存在");
        }
        BEANS.put(beanName, bean);
    }
    }
```

Spring 是注册表单例设计模式的实现，消除了编程式单例，而且对代码是非入侵式。

接下来让我们看看在 Spring 中如何配置单例 Bean 吧，在 Spring 容器中如果没指定作用域默认就是“singleton”，配置方式通过 scope 属性配置，具体配置如下：

```
<bean    class="cn.javass.spring.chapter3.bean.Printer" scope="singleton"/>
```

Spring 管理单例对象在 Spring 容器中存储如图 3-5 所示，Spring 不仅会缓存单例对象，Bean 定义也是会缓存的，对于惰性初始化的对象是在首次使用时根据 Bean 定义创建并存

放于单例缓存池。

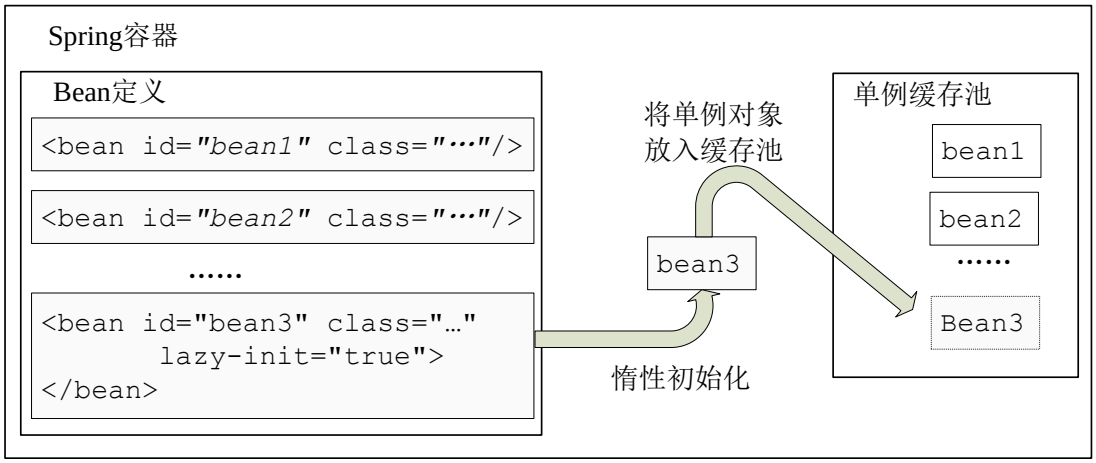


图 3-5 单例处理

二、prototype：即原型，指每次向 Spring 容器请求获取 Bean 都返回一个全新的 Bean，相对于“singleton”来说就是不缓存 Bean，每次都是一个根据 Bean 定义创建的全新 Bean。

> GoF 原型设计模式，指用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

Spring 中的原型和 GoF 中介绍的原型含义是不一样的：

- GoF 通过用原型实例指定创建对象的种类，而 Spring 容器用 Bean 定义指定创建对象的种类；
- GoF 通过拷贝这些原型创建新的对象，而 Spring 容器根据 Bean 定义创建新对象。

其相同地方都是根据某些东西创建新东西，而且 GoF 原型必须显示实现克隆操作，属于侵入式，而 Spring 容器只需配置即可，属于非侵入式。

接下来让我们看看 Spring 如何实现原型呢？

1）首先让我们来定义 Bean“原型”：Bean 定义，所有对象将根据 Bean 定义创建；在此我们只是简单示例一下，不会涉及依赖注入等复杂实现：BeanDefinition 类定义属性“class”表示原型类，“id”表示唯一标识，“scope”表示作用域，具体如下：

```
package cn.javass.spring.chapter3;
public class BeanDefinition {
    //单例
    public static final int SCOPE_SINGLETON = 0;
    //原型
    public static final int SCOPE_PROTOTYPE = 1;
    //唯一标识
    private String id;
    //class全限定名
    private String clazz;
    //作用域
    private int scope = SCOPE_SINGLETON;
```

2）接下来让我们看看 Bean 定义注册表，类似于单例注册表：

```
package cn.javass.spring.chapter3;
import java.util.HashMap;
import java.util.Map;
public class BeanDifinitionRegister {
    //bean 定义缓存，此处不考虑并发问题
    private final Map<String, BeanDefinition> DEFINITIONS =
                                        new HashMap<String, BeanDefinition>();
    public void registerBeanDefinition(String beanName, BeanDefinition bd) {
        //1.本实现不允许覆盖 Bean 定义
        if(DEFINITIONS.containsKey(bd.getId())) {
            throw new RuntimeException("已存在 Bean 定义，此实现不允许覆盖");
        }
        //2.将 Bean 定义放入 Bean 定义缓存池
        DEFINITIONS.put(bd.getId(), bd);
    }
    public BeanDefinition getBeanDefinition(String beanName) {
        return DEFINITIONS.get(beanName);
    }
    public boolean containsBeanDefinition(String beanName) {
        return DEFINITIONS.containsKey(beanName);
    }
}
```

3）接下来应该来定义 BeanFactory 了：

```
package cn.javass.spring.chapter3;
import org.springframework.beans.factory.config.SingletonBeanRegistry;
public class DefaultBeanFactory {
    //Bean定义注册表
    private BeanDifinitionRegister DEFINITIONS = new BeanDifinitionRegister();

    //单例注册表
    private final SingletonBeanRegistry SINGLETONS = new
SingletonBeanRegister();

    public Object getBean(String beanName) {
```

```
public void registerBeanDefinition(BeanDefinition bd) {
        DEFINITIONS.registerBeanDefinition(bd.getId(), bd);
 }

 private Object createBean(BeanDefinition bd) {
        //根据Bean定义创建Bean
        try {
                Class clazz = Class.forName(bd.getClazz());
                //通过反射使用无参数构造器创建Bean
                return clazz.getConstructor().newInstance();
        } catch (ClassNotFoundException e) {
                throw new RuntimeException("没有找到Bean[" + bd.getId() + "]类");
```

其中方法 getBean 用于获取根据 beanName 对于的 Bean 定义创建的对象，有单例和原型两类 Bean；registerBeanDefinition 方法用于注册 Bean 定义，私有方法 createBean 用于根据 Bean 定义中的类型信息创建 Bean。

3）测试一下吧，在此我们只测试原型作用域 Bean，对于每次从 Bean 工厂中获取的 Bean 都是一个全新的对象，代码片段（BeanFatoryTest）如下：

```
@Test
public void testPrototype () throws Exception {
    //1.创建Bean工厂
    DefaultBeanFactory bf = new DefaultBeanFactory();
    //2.创建原型 Bean定义
    BeanDefinition bd = new BeanDefinition();
    bd.setId("bean");
    bd.setScope(BeanDefinition.SCOPE_PROTOTYPE);
    bd.setClazz(HelloImpl2.class.getName());
    bf.registerBeanDefinition(bd);
    //对于原型Bean每次应该返回一个全新的Bean
    System.out.println(bf.getBean("bean") != bf.getBean("bean"));
}
```

最后让我们看看如何在 Spring 中进行配置吧，只需指定<bean>标签属性“scope”属性为“prototype”即可：

```
<bean class="cn.javass.spring.chapter3.bean.Printer" />
```

Spring 管理原型对象在 Spring 容器中存储如图 3-6 所示，Spring 不会缓存原型对象，而是根据 Bean 定义每次请求返回一个全新的 Bean：

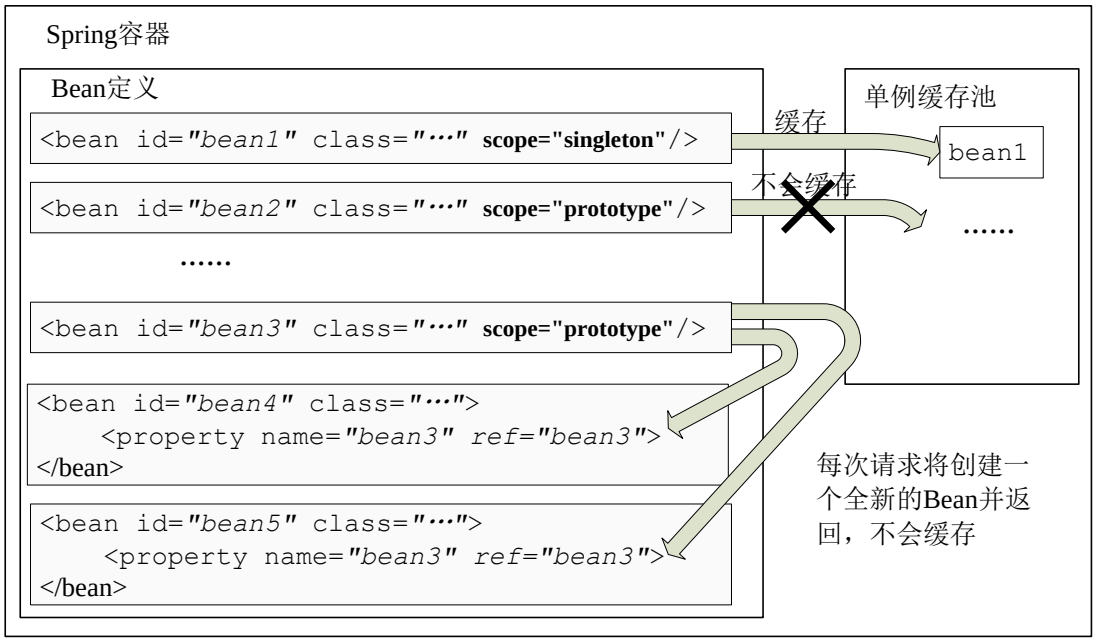


图 3-6 原型处理

单例和原型作用域我们已经讲完，接下来让我们学习一些在Web应用中有哪些作用域：

3.4.2 Web 应用中的作用域

在 Web 应用中，我们可能需要将数据存储到 request、session、global session。因此 Spring 提供了三种 Web 作用域：request、session、globalSession。

一、request 作用域：表示每个请求需要容器创建一个全新 Bean。比如提交表单的数据必须是对每次请求新建一个 Bean 来保持这些表单数据，请求结束释放这些数据。

二、session 作用域：表示每个会话需要容器创建一个全新 Bean。比如对于每个用户一般会有一个会话，该用户的用户信息需要存储到会话中，此时可以将该 Bean 配置为 web 作用域。

三、globalSession：类似于 session 作用域，只是其用于 portlet 环境的 web 应用。如果在非 portlet 环境将视为 session 作用域。

配置方式和基本的作用域相同，只是必须要有 web 环境支持，并配置相应的容器监听器或拦截器从而能应用这些作用域，我们会在集成 web 时讲解具体使用，大家只需要知道有这些作用域就可以了。

3.4.4 自定义作用域

在日常程序开发中，几乎用不到自定义作用域，除非又必要才进行自定义作用域。

首先让我们看下 Scope 接口吧：

```
package org.springframework.beans.factory.config;
import org.springframework.beans.factory.ObjectFactory;
public interface Scope {
    Object get(String name, ObjectFactory<?> objectFactory);
    Object remove(String name);
```

1）**Object get(String name, ObjectFactory<?> objectFactory)：**用于从作用域中获取 Bean，其中参数 objectFactory 是当在当前作用域没找到合适 Bean 时使用它创建一个新的 Bean；

2）**void registerDestructionCallback(String name, Runnable callback)：**用于注册销毁回调，如果想要销毁相应的对象则由 Spring 容器注册相应的销毁回调，而由自定义作用域选择是不是要销毁相应的对象；

3）**Object resolveContextualObject(String key)：**用于解析相应的上下文数据，比如 request 作用域将返回 request 中的属性。

4）**String getConversationId()：**作用域的会话标识，比如 session 作用域将是 sessionId。

让我们来实现个简单的 thread 作用域，该作用域内创建的对象将绑定到 ThreadLocal 内。

```
package cn.javass.spring.chapter3;
import java.util.HashMap;
import java.util.Map;
import org.springframework.beans.factory.ObjectFactory;
import org.springframework.beans.factory.config.Scope;
public class ThreadScope implements Scope {
    private final ThreadLocal<Map<String, Object>> THREAD_SCOPE =
                                new ThreadLocal<Map<String, Object>>() {
        protected Map<String, Object> initialValue() {
            //用于存放线程相关Bean
            return new HashMap<String, Object>();
        }
    };
```

```
    @Override
    public Object get(String name, ObjectFactory<?> objectFactory) {
        //如果当前线程已经绑定了相应Bean，直接返回
        if(THREAD_SCOPE.get().containsKey(name)) {
            return THREAD_SCOPE.get().get(name);
        }
        //使用objectFactory创建Bean并绑定到当前线程上
        THREAD_SCOPE.get().put(name, objectFactory.getObject());
        return THREAD_SCOPE.get().get(name);
    }
```

Scope 已经实现了，让我们将其注册到 Spring 容器，使其发挥作用：

```
<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
        <property name="scopes">
            <map><entry>
<!-- 指定scope关键字 --><key><value>thread</value></key>
<!-- scope实现 -->        <bean class="cn.javass.spring.chapter3.ThreadScope"/>
            </entry></map>
        </property>
    </bean>
```

通过 CustomScopeConfigurer 的 scopes 属性注册自定义作用域实现，在此需要指定使用作用域的关键字“thread”，并指定自定义作用域实现。来让我们来定义一个“thread”作用域的 Bean，配置（chapter3/threadScope.xml）如下：

```
<bean id="helloApi"
      class="cn.javass.spring.chapter2.helloworld.HelloImpl"
      scope="thread"/>
```

最后测试（cn.javass.spring.chapter3.ThreadScopeTest）一下吧，首先在一个线程中测试，在同一线程中获取的 Bean 应该是一样的；再让我们开启两个线程，然后应该这两个线程创建的 Bean 是不一样：

```
@Test
public void testSingleThread() {
    BeanFactory beanFactory =
        new ClassPathXmlApplicationContext("chapter3/threadScope.xml");
    HelloApi bean1 = beanFactory.getBean("helloApi", HelloApi.class);
    HelloApi bean2 = beanFactory.getBean("helloApi", HelloApi.class);
    //在同一线程中两次获取的Bean应该是相等的
    Assert.assertEquals(bean1, bean2);
}
```

```
@Test
public void testTwoThread() throws InterruptedException {
    final BeanFactory beanFactory =
        new ClassPathXmlApplicationContext("chapter3/threadScope.xml");
    final HelloApi[] beans = new HelloApi[2];
    Thread thread1 = new Thread() {
        public void run() {
            beans[0] = beanFactory.getBean("helloApi", HelloApi.class);
        }};
    Thread thread2 = new Thread() {
        public void run() {
            beans[1] = beanFactory.getBean("helloApi", HelloApi.class);
        }};
    thread1.start();thread1.sleep(1000);
    thread2.start();thread2.sleep(1000);
    //在两个线程中两次获取的Bean应该是相等的
    Assert.assertEquals(beans[0], beans[1]);
}
```

自定义作用域实现其实是非常简单的，其实复杂的是如果需要销毁 Bean，自定义作用域如何正确的销毁 Bean。

第四章 资源

4.1 基础知识

4.1.1 概述

在日常程序开发中，处理外部资源是很繁琐的事情，我们可能需要处理 URL 资源、File 资源资源、ClassPath 相关资源、服务器相关资源（JBoss AS 5.x 上的 VFS 资源）等等很多资源。因此处理这些资源需要使用不同的接口，这就增加了我们系统的复杂性；而且处理这些资源步骤都是类似的（打开资源、读取资源、关闭资源），因此如果能抽象出一个统一的接口来对这些底层资源进行统一访问，是不是很方便，而且使我们系统更加简洁，都是对不同的底层资源使用同一个接口进行访问。

Spring 提供一个 Resource 接口来统一这些底层资源一致的访问，而且提供了一些便利的接口，从而能提供我们的生产力。

4.1.2 Resource 接口

Spring 的 Resource 接口代表底层外部资源，提供了对底层外部资源的一致性访问接口。

```
public interface InputStreamSource {
    InputStream getInputStream() throws IOException;
}
```

```
public interface Resource extends InputStreamSource {
    boolean exists();
    boolean isReadable();
    boolean isOpen();
    URL getURL() throws IOException;
    URI getURI() throws IOException;
    File getFile() throws IOException;
    long contentLength() throws IOException;
    long lastModified() throws IOException;
    Resource createRelative(String relativePath) throws IOException;
    String getFilename();
    String getDescription();
}
```

1）InputStreamSource 接口解析：

- **getInputStream**：每次调用都将返回一个新鲜的资源对应的 java.io.InputStream 字节流，调用者在使用完毕后必须关闭该资源。

2）Resource 接口继承 InputStreamSource 接口，并提供一些便利方法：

- **exists**：返回当前 Resource 代表的底层资源是否存在，true 表示存在。
- **isReadable**：返回当前 Resource 代表的底层资源是否可读，true 表示可读。
- **isOpen**：返回当前 Resource 代表的底层资源是否已经打开，如果返回 true，则只能被读取一次然后关闭以避免内存泄漏；常见的 Resource 实现一般返回 false。
- **getURL**：如果当前 Resource 代表的底层资源能由 java.util.URL 代表，则返回该 URL，否则抛出 IOException。
- **getURI**：如果当前 Resource 代表的底层资源能由 java.util.URI 代表，则返回该 URI，否则抛出 IOException。
- **getFile**：如果当前 Resource 代表的底层资源能由 java.io.File 代表，则返回该 File，否则抛出 IOException。
- **contentLength**：返回当前 Resource 代表的底层文件资源的长度，一般是值代表的文件资源的长度。
- **lastModified**：返回当前 Resource 代表的底层资源的最后修改时间。
- **createRelative**：用于创建相对于当前 Resource 代表的底层资源的资源，比如当前 Resource 代表文件资源“d:/test/”则 createRelative（“test.txt”）将返回表文件资源“d:/test/test.txt”Resource 资源。
- **getFilename**：返回当前 Resource 代表的底层文件资源的文件路径，比如 File 资源“file://d:/test.txt”将返回“d:/test.txt”，而 URL 资源 http://www.javass.cn 将返回“”，因为只返回文件路径。
- **getDescription**：返回当前 Resource 代表的底层资源的描述符，通常就是资源的全路径（实际文件名或实际 URL 地址）。

Resource 接口提供了足够的抽象，足够满足我们日常使用。而且提供了很多内置 Resource 实现：ByteArrayResource、InputStreamResource、FileSystemResource、UrlResource、ClassPathResource、ServletContextResource、VfsResource 等。

4.2 内置 Resource 实现

4.2.1 ByteArrayResource

ByteArrayResource 代表 byte[]数组资源，对于“getInputStream”操作将返回一个 ByteArrayInputStream。

首先让我们看下使用 ByteArrayResource 如何处理 byte 数组资源：

```
package cn.javass.spring.chapter4;
import java.io.IOException;
import java.io.InputStream;
import org.junit.Test;
import org.springframework.core.io.ByteArrayResource;
import org.springframework.core.io.Resource;
public class ResourceTest {
@Test
public void testByteArrayResource() {
    Resource resource = new ByteArrayResource("Hello World!".getBytes());
        if(resource.exists()) {
            dumpStream(resource);
        }
    }
}
```

是不是很简单，让我们看下“dumpStream”实现：

```
private void dumpStream(Resource resource) {
        InputStream is = null;
        try {
            //1.获取文件资源
            is = resource.getInputStream();
            //2.读取资源
            byte[] descBytes = new byte[is.available()];
            is.read(descBytes);
            System.out.println(new String(descBytes));
        } catch (IOException e) {
            e.printStackTrace();
        }
        finally {
            try {
                //3.关闭资源
                is.close();
            } catch (IOException e) {
            }
        }
    }
```

让我们来仔细看一下代码，dumpStream 方法很抽象定义了访问流的三部曲：打开资源、读取资源、关闭资源，所以 dunpStrean 可以再进行抽象从而能在自己项目中使用；byteArrayResourceTest 测试方法，也定义了基本步骤：定义资源、验证资源存在、访问资源。

ByteArrayResource 可多次读取数组资源，即 isOpen ()永远返回 false。

1.2.2 InputStreamResource

InputStreamResource 代表 java.io.InputStream 字节流，对于“getInputStream ”操作将直接返回该字节流，因此只能读取一次该字节流，即“isOpen”永远返回 true。

让我们看下测试代码吧：

```
@Test
public void testInputStreamResource() {
    ByteArrayInputStream bis = new ByteArrayInputStream("Hello World!".getBytes());
    Resource resource = new InputStreamResource(bis);
     if(resource.exists()) {
         dumpStream(resource);
     }
     Assert.assertEquals(true, resource.isOpen());
}
```

测试代码几乎和 ByteArrayResource 测试完全一样，注意“isOpen”此处用于返回 true。

4.2.3 FileSystemResource

FileSystemResource 代表 java.io.File 资源，对于“getInputStream ”操作将返回底层文件的字节流，“isOpen”将永远返回 false，从而表示可多次读取底层文件的字节流。

让我们看下测试代码吧：

```
@Test
public void testFileResource() {
     File file = new File("d:/test.txt");
     Resource resource = new FileSystemResource(file);
     if(resource.exists()) {
         dumpStream(resource);
     }
     Assert.assertEquals(false, resource.isOpen());
}
```

注意由于“isOpen”将永远返回false，所以可以多次调用dumpStream(resource)。

4.2.4 ClassPathResource

ClassPathResource 代表 classpath 路径的资源，将使用 ClassLoader 进行加载资源。classpath 资源存在于类路径中的文件系统中或 jar 包里，且“isOpen”永远返回 false，表示可多次读取资源。

ClassPathResource 加载资源替代了 Class 类和 ClassLoader 类的“getResource(String name)”和“getResourceAsStream(String name)”两个加载类路径资源方法，提供一致的访问方式。

ClassPathResource 提供了三个构造器：

- **public ClassPathResource(String path)**：使用默认的 ClassLoader 加载“path”类路径资源；
- **public ClassPathResource(String path, ClassLoader classLoader)**：使用指定的 ClassLoader 加载“path”类路径资源；

  比如当前类路径是“cn.javass.spring.chapter4.ResourceTest”，而需要加载的资源路径是“cn/javass/spring/chapter4/test1.properties”，则将加载的资源在“cn/javass/spring/chapter4/test1.properties”；
- **public ClassPathResource(String path, Class<?> clazz)**：使用指定的类加载“path”类路径资源，将加载相对于当前类的路径的资源；

  比如当前类路径是“cn.javass.spring.chapter4.ResourceTest”，而需要加载的资源路径是“cn/javass/spring/chapter4/test1.properties”，则将加载的资源在“cn/javass/spring/chapter4/cn/javass/spring/chapter4/test1.properties”；

  而如果需要加载的资源路径为“test1.properties”，将加载的资源为“cn/javass/spring/chapter4/test1.properties”。

让我们直接看测试代码吧：

1）使用默认的加载器加载资源，将加载当前 ClassLoader 类路径上相对于根路径的资源：

```
@Test
public void testClasspathResourceByDefaultClassLoader() throws IOException {
    Resource resource =
            new ClassPathResource("cn/javass/spring/chapter4/test1.properties");
     if(resource.exists()) {
            dumpStream(resource);
     }
     System.out.println("path:" + resource.getFile().getAbsolutePath());
     Assert.assertEquals(false, resource.isOpen());
}
```

2）使用指定的 ClassLoader 进行加载资源，将加载指定的 ClassLoader 类路径上相对于

根路径的资源：

```
@Test
public void testClasspathResourceByClassLoader() throws IOException {
    ClassLoader cl = this.getClass().getClassLoader();
    Resource resource =
        new ClassPathResource("cn/javass/spring/chapter4/test1.properties" , cl);
    if(resource.exists()) {
        dumpStream(resource);
    }
    System.out.println("path:" + resource.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource.isOpen());
    }
```

3）使用指定的类进行加载资源，将尝试加载相对于当前类的路径的资源：

```
@Test
public void testClasspathResourceByClass() throws IOException {
    Class clazz = this.getClass();
    Resource resource1 =
        new ClassPathResource("cn/javass/spring/chapter4/test1.properties" , clazz);
    if(resource1.exists()) {
        dumpStream(resource1);
    }
    System.out.println("path:" + resource1.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource1.isOpen());

    Resource resource2 = new ClassPathResource("test1.properties" , this.getClass());
    if(resource2.exists()) {
        dumpStream(resource2);
    }
    System.out.println("path:" + resource2.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource2.isOpen());
}
```

“resource1”将加载 cn/javass/spring/chapter4/cn/javass/spring/chapter4/test1.properties 资源；“resource2”将加载“cn/javass/spring/chapter4/test1.properties”；

4）加载 jar 包里的资源，首先在当前类路径下找不到，最后才到 Jar 包里找，而且在第一个 Jar 包里找到的将被返回：

```
@Test
public void classpathResourceTestFromJar() throws IOException {
    Resource resource = new ClassPathResource("overview.html");
    if(resource.exists()) {
        dumpStream(resource);
    }
    System.out.println("path:" + resource.getURL().getPath());
    Assert.assertEquals(false, resource.isOpen());
}
```

如果当前类路径包含“overview.html”，在项目的“resources”目录下，将加载该资源，否则将加载 Jar 包里的“overview.html”，而且不能使用“resource.getFile()”，应该使用“resource.getURL()”，因为资源不存在于文件系统而是存在于 jar 包里，URL 类似于“`file:/C:/.../***.jar!/overview.html`”。

类路径一般都是相对路径，即相对于类路径或相对于当前类的路径，因此如果使用“/test1.properties”带前缀“/”的路径，将自动删除“/”得到“test1.properties”。

4.2.5 UrlResource

UrlResource 代表 URL 资源，用于简化 URL 资源访问。“isOpen”永远返回 false，表示可多次读取资源。

UrlResource 一般支持如下资源访问：

- **http**：通过标准的 http 协议访问 web 资源，如 new UrlResource(“http://地址”)；
- **ftp**：通过 ftp 协议访问资源，如 new UrlResource(“ftp://地址”)；
- **file**：通过 file 协议访问本地文件系统资源，如 new UrlResource(“file:d:/test.txt”)；

具体使用方法在此就不演示了，可以参考 cn.javass.spring.chapter4.ResourceTest 中 urlResourceTest 测试方法。

4.2.6 ServletContextResource

ServletContextResource 代表 web 应用资源，用于简化 servlet 容器的 ServletContext 接口的 getResource 操作和 getResourceAsStream 操作；在此就不具体演示了。

4.2.7 VfsResource

VfsResource 代表 Jboss 虚拟文件系统资源。

Jboss VFS(Virtual File System)框架是一个文件系统资源访问的抽象层，它能一致的访问物理文件系统、jar 资源、zip 资源、war 资源等，VFS 能把这些资源一致的映射到一个目录上，访问它们就像访问物理文件资源一样，而其实这些资源不存在于物理文件系统。

在示例之前需要准备一些 jar 包，在此我们使用的是 Jboss VFS3 版本，可以下载最新的 Jboss AS 6x，拷贝 lib 目录下的“jboss-logging.jar”和“jboss-vfs.jar”两个 jar 包拷贝到我们项目的 lib 目录中并添加到“Java Build Path”中的“Libaries”中。

让我们看下示例（cn.javass.spring.chapter4.ResourceTest）：

```
@Test
public void testVfsResourceForRealFileSystem() throws IOException {
    //1.创建一个虚拟的文件目录
    VirtualFile home = VFS.getChild("/home");
    //2.将虚拟目录映射到物理的目录
    VFS.mount(home, new RealFileSystem(new File("d:")));
    //3.通过虚拟目录获取文件资源
    VirtualFile testFile = home.getChild("test.txt");
    //4.通过一致的接口访问
    Resource resource = new VfsResource(testFile);
    if(resource.exists()) {
        dumpStream(resource);
    }
    System.out.println("path:" + resource.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource.isOpen());
}
@Test
public void testVfsResourceForJar() throws IOException {
    //1.首先获取jar包路径
    File realFile = new File("lib/org.springframework.beans-3.0.5.RELEASE.jar");
    //2.创建一个虚拟的文件目录
    VirtualFile home = VFS.getChild("/home2");
    //3.将虚拟目录映射到物理的目录
    VFS.mountZipExpanded(realFile, home,
        TempFileProvider.create("tmp", Executors.newScheduledThreadPool(1)));
    //4.通过虚拟目录获取文件资源
    VirtualFile testFile = home.getChild("META-INF/spring.handlers");
    Resource resource = new VfsResource(testFile);
    if(resource.exists()) {
            dumpStream(resource);
    }
    System.out.println("path:" + resource.getFile().getAbsolutePath());
    Assert.assertEquals(false, resource.isOpen());
}
```

通过 VFS，对于 jar 里的资源和物理文件系统访问都具有一致性，此处只是简单示例，如果需要请到 Jboss 官网深入学习。

4.3 访问 Resource

4.3.1 ResourceLoader 接口

ResourceLoader 接口用于返回 Resource 对象；其实现可以看作是一个生产 Resource 的工厂类。

```
public interface ResourceLoader {
    Resource getResource(String location);
    ClassLoader getClassLoader();
}
```

getResource 接口用于根据提供的 location 参数返回相应的 Resource 对象；而 getClassLoader 则返回加载这些 Resource 的 ClassLoader。

Spring 提供了一个适用于所有环境的 DefaultResourceLoader 实现，可以返回 ClassPathResource、UrlResource；还提供一个用于 web 环境的 ServletContextResourceLoader，它继承了 DefaultResourceLoader 的所有功能，又额外提供了获取 ServletContextResource 的支持。

ResourceLoader 在进行加载资源时需要使用前缀来指定需要加载：“classpath:path”表示返回 ClasspathResource，“http://path”和“file:path”表示返回 UrlResource 资源，如果不加前缀则需要根据当前上下文来决定，DefaultResourceLoader 默认实现可以加载 classpath 资源，如代码所示（cn.javass.spring.chapter4.ResourceLoaderTest）：

```
@Test
public void testResourceLoad() {
    ResourceLoader loader = new DefaultResourceLoader();
    Resource resource = loader.getResource("classpath:cn/javass/spring/chapter4/test1.txt");
    //验证返回的是ClassPathResource
    Assert.assertEquals(ClassPathResource.class, resource.getClass());
    Resource resource2 = loader.getResource("file:cn/javass/spring/chapter4/test1.txt");
    //验证返回的是ClassPathResource
    Assert.assertEquals(UrlResource.class, resource2.getClass());
    Resource resource3 = loader.getResource("cn/javass/spring/chapter4/test1.txt");
    //验证返默认可以加载ClasspathResource
    Assert.assertTrue(resource3 instanceof ClassPathResource);
}
```

对于目前所有 ApplicationContext 都实现了 ResourceLoader，因此可以使用其来加载资源。

- **ClassPathXmlApplicationContext**：不指定前缀将返回默认的 ClassPathResource 资源，否则将根据前缀来加载资源；
- **FileSystemXmlApplicationContext**：不指定前缀将返回 FileSystemResource，否则将根据前缀来加载资源；
- **WebApplicationContext**：不指定前缀将返回 ServletContextResource，否则将根据前缀来加载资源；
- **其他**：不指定前缀根据当前上下文返回 Resource 实现，否则将根据前缀来加载资源。

4.3.2 ResourceLoaderAware 接口

ResourceLoaderAware 是一个标记接口，用于通过 ApplicationContext 上下文注入 ResourceLoader。

```
public interface ResourceLoaderAware {
    void setResourceLoader(ResourceLoader resourceLoader);
}
```

让我们看下测试代码吧：

1） 首先准备测试 Bean，我们的测试 Bean 还简单只需实现 ResourceLoaderAware 接口，然后通过回调将 ResourceLoader 保存下来就可以了：

```
package cn.javass.spring.chapter4.bean;
import org.springframework.context.ResourceLoaderAware;
import org.springframework.core.io.ResourceLoader;
public class ResourceBean implements ResourceLoaderAware {
    private ResourceLoader resourceLoader;
    @Override
    public void setResourceLoader(ResourceLoader resourceLoader) {
        this.resourceLoader = resourceLoader;
    }
    public ResourceLoader getResourceLoader() {
        return resourceLoader;
    }
}
```

2） 配置 Bean 定义（chapter4/resourceLoaderAware.xml）：

```
<bean class="cn.javass.spring.chapter4.bean.ResourceBean"/>
```

3）测试(cn.javass.spring.chapter4.ResoureLoaderAwareTest)：

```
@Test
public void test() {
      ApplicationContext ctx =
        new ClassPathXmlApplicationContext("chapter4/resourceLoaderAware.xml");
      ResourceBean resourceBean = ctx.getBean(ResourceBean.class);
      ResourceLoader loader = resourceBean.getResourceLoader();
      Assert.assertTrue(loader instanceof ApplicationContext);
 }
```

注意此处“loader instanceof ApplicationContext”，说明了 ApplicationContext 就是个 ResoureLoader。

由于上述实现回调接口注入 ResourceLoader 的方式属于侵入式，所以不推荐上述方法，可以采用更好的自动注入方式，如“byType”和“constructor”，此处就不演示了。

4.3.3 注入 Resource

通过回调或注入方式注入“ResourceLoader”，然后再通过“ResourceLoader”再来加载需要的资源对于只需要加载某个固定的资源是不是很麻烦，有没有更好的方法类似于前边实例中注入“java.io.File”类似方式呢？

Spring 提供了一个 PropertyEditor “ResourceEditor”用于在注入的字符串和 Resource 之间进行转换。因此可以使用注入方式注入 Resource。

ResourceEditor 完全使用 ApplicationContext 根据注入的路径字符串获取相应的 Resource，说白了还是自己做还是容器帮你做的问题。

接下让我们看下示例：

1）准备 Bean：

```
package cn.javass.spring.chapter4.bean;
import org.springframework.core.io.Resource;
public class ResourceBean3 {
      private Resource resource;
      public Resource getResource() {
            return resource;
      }
      public void setResource(Resource resource) {
            this.resource = resource;
      }
}
```

2）准备配置文件（chapter4/ resourceInject.xml）：

```
<bean id="resourceBean1" class="cn.javass.spring.chapter4.bean.ResourceBean3">
    <property name="resource" value="cn/javass/spring/chapter4/test1.properties"/>
</bean>
<bean id="resourceBean2" class="cn.javass.spring.chapter4.bean.ResourceBean3">
  <property name="resource"
            value="classpath:cn/javass/spring/chapter4/test1.properties"/>
</bean>
```

注意此处“resourceBean1”注入的路径没有前缀表示根据使用的 ApplicationContext 实现进行选择 Resource 实现。

3）让我们来看下测试代码（cn.javass.spring.chapter4.ResourceInjectTest）吧：

```
@Test
public void test() {
  ApplicationContext ctx =
      new ClassPathXmlApplicationContext("chapter4/resourceInject.xml");
  ResourceBean3 resourceBean1 = ctx.getBean("resourceBean1", ResourceBean3.class);
  ResourceBean3 resourceBean2 = ctx.getBean("resourceBean2", ResourceBean3.class);
  Assert.assertTrue(resourceBean1.getResource() instanceof ClassPathResource);
  Assert.assertTrue(resourceBean2.getResource() instanceof ClassPathResource);
}
```

接下来一节让我们深入 ApplicationContext 对各种 Resource 的支持，及如何使用更便利的资源加载方式。

4.4 Resource 通配符路径

4.4.1 使用路径通配符加载 Resource

前面介绍的资源路径都是非常简单的一个路径匹配一个资源，Spring 还提供了一种更强大的 Ant 模式通配符匹配，从能一个路径匹配一批资源。

Ant 路径通配符支持“？”、“*”、“**”，注意通配符匹配不包括目录分隔符“/”：

- **“?”：匹配一个字符**，如“config?.xml”将匹配“config1.xml”；
- **“*”：匹配零个或多个字符串**，如“cn/*/config.xml”将匹配“cn/javass/config.xml”，但不匹配匹配“cn/config.xml”；而“cn/config-*.xml”将匹配“cn/config-dao.xml”；
- **“**”：匹配路径中的零个或多个目录**，如“cn/**/config.xml”将匹配“cn/config.xml”，也匹配“cn/javass/spring/config.xml”；而“cn/javass/config-**.xml”将匹配“cn/javass/config-dao.xml”，即把“**”当做两个“*”处理。

Spring 提供 AntPathMatcher 来进行 Ant 风格的路径匹配。具体测试请参考 cn.javass.spring.chapter4. AntPathMatcherTest。

Spring 在加载类路径资源时除了提供前缀“classpath:”的来支持加载一个 Resource，还提供一个前缀“classpath*:”来支持加载所有匹配的类路径 Resource。

Spring 提供 ResourcePatternResolver 接口来加载多个 Resource，该接口继承了 ResourceLoader 并添加了“Resource[] getResources(String locationPattern)”用来加载多个 Resource：

```
public interface ResourcePatternResolver extends ResourceLoader {
    String CLASSPATH_ALL_URL_PREFIX = "classpath*:";
    Resource[] getResources(String locationPattern) throws IOException;
}
```

Spring 提供了一个 ResourcePatternResolver 实现 PathMatchingResourcePatternResolver，它是基于模式匹配的，默认使用 AntPathMatcher 进行路径匹配，它除了支持 ResourceLoader 支持的前缀外，还额外支持“classpath*:”用于加载所有匹配的类路径 Resource，ResourceLoader 不支持前缀“classpath*:”：

首先做下准备工作，在项目的“resources”创建“META-INF”目录，然后在其下创建一个“INDEX.LIST”文件。同时在“org.springframework.beans-3.0.5.RELEASE.jar”和“org.springframework.context-3.0.5.RELEASE.jar”两个 jar 包里也存在相同目录和文件。然后创建一个“LICENSE”文件，该文件存在于“com.springsource.cn.sf.cglib-2.2.0.jar”里。

一、“classpath”： 用于加载类路径（包括 jar 包）中的一个且仅一个资源；对于多个匹配的也只返回一个，所以如果需要多个匹配的请考虑“classpath*:”前缀；

```
@Test
public void testClasspathPrefix() throws IOException {
   ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
   //只加载一个绝对匹配Resource，且通过ResourceLoader.getResource进行加载
   Resource[] resources=resolver.getResources("classpath:META-INF/INDEX.LIST");
   Assert.assertEquals(1, resources.length);
   //只加载一个匹配的Resource，且通过ResourceLoader.getResource进行加载
   resources = resolver.getResources("classpath:META-INF/*.LIST");
   Assert.assertTrue(resources.length == 1);
}
```

二、“classpath*”： 用于加载类路径（包括 jar 包）中的所有匹配的资源。带通配符的 classpath 使用“ClassLoader”的“Enumeration<URL> getResources(String name)”方

法来查找通配符之前的资源，然后通过模式匹配来获取匹配的资源。如“classpath:META-INF/*.LIST”将首先加载通配符之前的目录“META-INF”，然后再遍历路径进行子路径匹配从而获取匹配的资源。

```
@Test
public void testClasspathAsteriskPrefix () throws IOException {
  ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
  //将加载多个绝对匹配的所有Resource
  //将首先通过ClassLoader.getResources("META-INF")加载非模式路径部分
  //然后进行遍历模式匹配
  Resource[] resources=resolver.getResources("classpath*:META-INF/INDEX.LIST");
  Assert.assertTrue(resources.length > 1);
  //将加载多个模式匹配的Resource
  resources = resolver.getResources("classpath*:META-INF/*.LIST");
    Assert.assertTrue(resources.length > 1);
}
```

注意“resources.length > 1”说明返回多个 Resource。不管模式匹配还是非模式匹配只要匹配的都将返回。

在“com.springsource.cn.sf.cglib-2.2.0.jar”里包含“asm-license.txt”文件，对于使用“classpath*: asm-*.txt”进行通配符方式加载资源将什么也加载不了“asm-license.txt”文件，注意一定是模式路径匹配才会遇到这种问题。这是由于“ClassLoader”的“getResources(String name)”方法的限制，对于 name 为“”的情况将只返回文件系统的类路径，不会包换 jar 包根路径。

```
@Test
public void testClasspathAsteriskPrefixLimit() throws IOException {
    ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
    //将首先通过ClassLoader.getResources("")加载目录，
    //将只返回文件系统的类路径不返回jar的跟路径
    //然后进行遍历模式匹配
    Resource[] resources = resolver.getResources("classpath*:asm-*.txt");
    Assert.assertTrue(resources.length == 0);
    //将通过ClassLoader.getResources("asm-license.txt")加载
    //asm-license.txt存在于com.springsource.net.sf.cglib-2.2.0.jar
    resources = resolver.getResources("classpath*:asm-license.txt");
    Assert.assertTrue(resources.length > 0);
    //将只加载文件系统类路径匹配的Resource
    resources = resolver.getResources("classpath*:LICENS*");
    Assert.assertTrue(resources.length == 1);
}
```

对于“resolver.getResources("classpath*:asm-*.txt");”，由于在项目“resources”目录下没有所以应该返回 0 个资源；“resolver.getResources("classpath*:asm-license.txt");”将返回 jar 包里的 Resource；“resolver.getResources("classpath*:LICENS*");”，因为将只返回文件系统类路径资源，所以返回 1 个资源。

因此在通过前缀“classpath*”加载通配符路径时，必须包含一个根目录才能保证加载的资源是所有的，而不是部分。

三、“file”：加载一个或多个文件系统中的 Resource。如“file:D:/*.txt”将返回 D 盘下的所有 txt 文件；

四、无前缀：通过 ResourceLoader 实现加载一个资源。

AppliacationContext 提供的 getResources 方法将获取资源委托给 ResourcePatternResolver 实现，默认使用 PathMatchingResourcePatternResolver。所有在此就无需介绍其使用方法了。

4.4.2 注入 Resource 数组

Spring 还支持注入 Resource 数组，直接看配置如下：

```
<bean id="resourceBean1" class="cn.javass.spring.chapter4.bean.ResourceBean4">
    <property name="resources">
        <array>
            <value>cn/javass/spring/chapter4/test1.properties</value>
            <value>log4j.xml</value>
        </array>
    </property>
</bean>
<bean id="resourceBean2" class="cn.javass.spring.chapter4.bean.ResourceBean4">
    <property name="resources" value="classpath*:META-INF/INDEX.LIST"/>
</bean>
<bean id="resourceBean3" class="cn.javass.spring.chapter4.bean.ResourceBean4">
    <property name="resources">
        <array>
            <value>cn/javass/spring/chapter4/test1.properties</value>
            <value>classpath*:META-INF/INDEX.LIST</value>
        </array>
    </property>
</bean>
```

“resourceBean1”就不用多介绍了，传统实现方式；对于“resourceBean2”则使用前

缀“classpath*”，看到这大家应该懂的，加载匹配多个资源；“resourceBean3”是混合使用的；测试代码在“cn.javass.spring.chapter4.ResourceInjectTest.testResourceArrayInject”。

Spring 通过 ResourceArrayPropertyEditor 来进行类型转换的，而它又默认使用“PathMatchingResourcePatternResolver”来进行把路径解析为 Resource 对象。所有大家只要会使用“PathMatchingResourcePatternResolver”，其它一些实现都是委托给它的，比如 AppliacationContext 的“getResources”方法等。

4.4.3 AppliacationContext 实现对各种 Resource 的支持

一、ClassPathXmlApplicationContext：默认将通过 classpath 进行加载返回 ClassPathResource，提供两类构造器方法：

```
public class ClassPathXmlApplicationContext {
    //1）通过 ResourcePatternResolver 实现根据 configLocation 获取资源
    public ClassPathXmlApplicationContext(String configLocation);
    public ClassPathXmlApplicationContext(String... configLocations);
    public ClassPathXmlApplicationContext(String[] configLocations, ……);

    //2）通过直接根据 path 直接返回 ClasspathResource
    public ClassPathXmlApplicationContext(String path, Class clazz);
    public ClassPathXmlApplicationContext(String[] paths, Class clazz);
    public ClassPathXmlApplicationContext(String[] paths, Class clazz, ……);
}
```

第一类构造器是根据提供的配置文件路径使用“ResourcePatternResolver ”的“getResources()”接口通过匹配获取资源；即如“classpath:config.xml”

第二类构造器则是根据提供的路径和 clazz 来构造 ClassResource 资源。即采用“public ClassPathResource(String path, Class<?> clazz)”构造器获取资源。

二、FileSystemXmlApplicationContext：将加载相对于当前工作目录的“configLocation”位置的资源，注意在 linux 系统上不管“configLocation”是否带“/”，都作为相对路径；而在 window 系统上如“D:/resourceInject.xml”是绝对路径。因此在除非很必要的情况下，不建议使用该 ApplicationContext。

```
public class FileSystemXmlApplicationContext{
    public FileSystemXmlApplicationContext(String configLocation);
    public FileSystemXmlApplicationContext(String... configLocations,……);
}
```

```
//linux 系统，以下全是相对于当前 vm 路径进行加载
new FileSystemXmlApplicationContext("chapter4/config.xml");
new FileSystemXmlApplicationContext("/chapter4/confg.xml");
//windows 系统，第一个将相对于当前 vm 路径进行加载；
//第二个则是绝对路径方式加载
new FileSystemXmlApplicationContext("chapter4/config.xml");
```

此处还需要注意：在 linux 系统上，构造器使用的是相对路径，而 ctx.getResource()方法如果以“/”开头则表示获取绝对路径资源，而不带前导“/”将返回相对路径资源。如下：

```
//linux 系统，第一个将相对于当前 vm 路径进行加载；
//第二个则是绝对路径方式加载
ctx.getResource ("chapter4/config.xml");
ctx.getResource ("/root/confg.xml");
//windows 系统，第一个将相对于当前 vm 路径进行加载；
//第二个则是绝对路径方式加载
ctx.getResource ("chapter4/config.xml");
ctx.getResource ("d:/chapter4/confg.xml");
```

因此如果需要加载绝对路径资源最好选择前缀“file”方式，将全部根据绝对路径加载。如在 linux 系统“ctx.getResource ("file:/root/confg.xml");”

第五章　Spring 表达式语言

5.1　概述

5.1.1　概述

Spring 表达式语言全称为“Spring Expression Language”，缩写为“SpEL”，类似于 Struts2x 中使用的 OGNL 表达式语言，能在运行时构建复杂表达式、存取对象图属性、对象方法调用等等，并且能与 Spring 功能完美整合，如能用来配置 Bean 定义。

表达式语言给静态 Java 语言增加了动态功能。

SpEL 是单独模块，只依赖于 core 模块，不依赖于其他模块，可以单独使用。

5.1.2　能干什么

表达式语言一般是用最简单的形式完成最主要的工作，减少我们的工作量。

SpEL 支持如下表达式：

一、基本表达式：字面量表达式、关系，逻辑与算数运算表达式、字符串连接及截取表达式、三目运算及 Elivis 表达式、正则表达式、括号优先级表达式；

二、类相关表达式：类类型表达式、类实例化、instanceof 表达式、变量定义及引用、赋值表达式、自定义函数、对象属性存取及安全导航表达式、对象方法调用、Bean 引用；

三、集合相关表达式：内联 List、内联数组、集合，字典访问、列表，字典，数组修改、集合投影、集合选择；不支持多维内联数组初始化；不支持内联字典定义；

四、其他表达式：模板表达式。

注：SpEL 表达式中的关键字是不区分大小写的。

5.2　SpEL 基础

5.2.1　HelloWorld

首先准备支持 SpEL 的 Jar 包：“org.springframework.expression-3.0.5.RELEASE.jar”将其添加到类路径中。

SpEL 在求表达式值时一般分为四步，其中第三步可选：首先构造一个解析器，其次解析器解析字符串表达式，在此构造上下文，最后根据上下文得到表达式运算后的值。

让我们看下代码片段吧：

```
package cn.javass.spring.chapter5;
import junit.framework.Assert;
import org.junit.Test;
import org.springframework.expression.EvaluationContext;
import org.springframework.expression.Expression;
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.expression.spel.support.StandardEvaluationContext;
public class SpELTest {
    @Test
    public void helloWorld() {
        ExpressionParser parser = new SpelExpressionParser();
        Expression expression =
            parser.parseExpression("('Hello' + ' World').concat(#end)");
        EvaluationContext context = new StandardEvaluationContext();
        context.setVariable("end", "!");
        Assert.assertEquals("Hello World!", expression.getValue(context));
    }
}
```

接下来让我们分析下代码：

1）创建解析器：SpEL 使用 ExpressionParser 接口表示解析器，提供 SpelExpressionParser 默认实现；

2）解析表达式：使用 ExpressionParser 的 parseExpression 来解析相应的表达式为 Expression 对象。

3）构造上下文：准备比如变量定义等等表达式需要的上下文数据。

4）求值：通过 Expression 接口的 getValue 方法根据上下文获得表达式值。

是不是很简单，接下来让我们看下其具体实现及原理吧。

5.2.3 SpEL 原理及接口

SpEL 提供简单的接口从而简化用户使用，在介绍原理前让我们学习下几个概念：

一、表达式：表达式是表达式语言的核心，所以表达式语言都是围绕表达式进行的，从我们角度来看是“干什么”；

二、解析器：用于将字符串表达式解析为表达式对象，从我们角度来看是“谁来干”；

三、上下文：表达式对象执行的环境，该环境可能定义变量、定义自定义函数、提供类型转换等等，从我们角度看是“在哪干”；

四、根对象及活动上下文对象：根对象是默认的活动上下文对象，活动上下文对象表示了当前表达式操作的对象，从我们角度看是“对谁干”。

理解了这些概念后，让我们看下 SpEL 如何工作的呢，如图 5-1 所示：

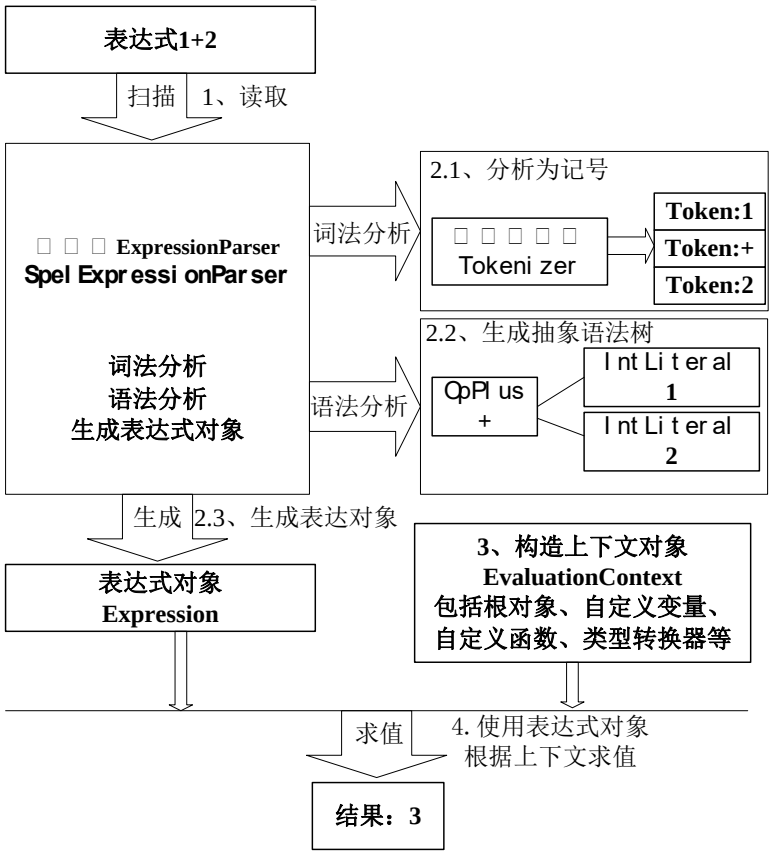


图 5-1 工作原理

1）首先定义表达式：“1+2”；

2）定义解析器 ExpressionParser 实现，SpEL 提供默认实现 SpelExpressionParser；

2.1）SpelExpressionParser 解析器内部使用 Tokenizer 类进行词法分析，即把字符串流分析为记号流，记号在 SpEL 使用 Token 类来表示；

2.2）有了记号流后，解析器便可根据记号流生成内部抽象语法树；在 SpEL 中语法树节点由 SpelNode 接口实现代表：如 OpPlus 表示加操作节点、IntLiteral 表示 int 型字面量节点；使用 SpelNodel 实现组成了抽象语法树；

2.3）对外提供 Expression 接口来简化表示抽象语法树，从而隐藏内部实现细节，并提供 getValue 简单方法用于获取表达式值；SpEL 提供默认实现为 SpelExpression；

3）定义表达式上下文对象（可选），SpEL 使用 EvaluationContext 接口表示上下文对象，用于设置根对象、自定义变量、自定义函数、类型转换器等，SpEL 提供默认实现 StandardEvaluationContext；

4）使用表达式对象根据上下文对象（可选）求值（调用表达式对象的 getValue 方法）获得结果。

接下来让我们看下 SpEL 的主要接口吧：

1 ） ExpressionParser 接 口 ： 表 示 解 析 器 ， 默 认 实 现 是 org.springframework.expression.spel.standard 包 中 的 SpelExpressionParser 类 ， 使 用 parseExpression 方法将字符串表达式转换为 Expression 对象，对于 ParserContext 接口用于定义字符串表达式是不是模板，及模板开始与结束字符：

```
public interface ExpressionParser {
    Expression parseExpression(String expressionString);
    Expression parseExpression(String expressionString, ParserContext context);
}
```

来看下示例：

```
@Test
public void testParserContext() {
    ExpressionParser parser = new SpelExpressionParser();
    ParserContext parserContext = new ParserContext() {
        @Override
         public boolean isTemplate() {
            return true;
        }
        @Override
        public String getExpressionPrefix() {
            return "#{";
        }
        @Override
        public String getExpressionSuffix() {
            return "}";
        }
    };
    String template = "#{'Hello '}#{'World!'}";
    Expression expression = parser.parseExpression(template, parserContext);
    Assert.assertEquals("Hello World!", expression.getValue());
}
```

在此我们演示的是使用 ParserContext 的情况，此处定义了 ParserContext 实现：定义表达式是模块，表达式前缀为“#{”，后缀为“}”；使用 parseExpression 解析时传入的模板必须以“#{”开头，以“}”结尾，如"#{'Hello '}#{'World!'}"。

默认传入的字符串表达式不是模板形式，如之前演示的 Hello World。

2 ） EvaluationContext 接 口 ： 表 示 上 下 文 环 境 ， 默 认 实 现 是

org.springframework.expression.spel.support 包中的 StandardEvaluationContext 类，使用 setRootObject 方法来设置根对象，使用 setVariable 方法来注册自定义变量，使用 registerFunction 来注册自定义函数等等。

3）Expression 接口：表示表达式对象，默认实现是 org.springframework.expression.spel.standard 包中的 SpelExpression，提供 getValue 方法用于获取表达式值，提供 setValue 方法用于设置对象值。

了解了 SpEL 原理及接口，接下来的事情就是 SpEL 语法了。

5.3 SpEL 语法

5.3.1 基本表达式

一、字面量表达式： SpEL 支持的字面量包括：字符串、数字类型（int、long、float、double）、布尔类型、null 类型。

类型	示例
字符串	String str1 = parser.parseExpression("'Hello World!'").getValue(String.class); String str2 = parser.parseExpression("\"Hello World!\"").getValue(String.class);

数字类型	int int1 = parser.parseExpression("1").getValue(Integer.class);
	long long1 = parser.parseExpression("-1L").getValue(long.class);
	float float1 = parser.parseExpression("1.1").getValue(Float.class);
	double double1 = parser.parseExpression("1.1E+2").getValue(double.class);
	int hex1 = parser.parseExpression("0xa").getValue(Integer.class);
	long hex2 = parser.parseExpression("0xaL").getValue(long.class);

布尔类型	boolean true1 = parser.parseExpression("true").getValue(boolean.class); boolean false1 = parser.parseExpression("false").getValue(boolean.class);
null 类型	Object null1 = parser.parseExpression("null").getValue(Object.class);

二、算数运算表达式： SpEL 支持加(+)、减(-)、乘(*)、除(/)、求余（%）、幂（^）运算。

类型	示例
加减乘除	int result1 = parser.parseExpression("1+2-3*4/2").getValue(Integer.class);//-3
求余	int result2 = parser.parseExpression("4%3").getValue(Integer.class);//1
幂运算	int result3 = parser.parseExpression("2^3").getValue(Integer.class);//8

SpEL 还提供求余（MOD）和除（DIV）而外两个运算符，与“%”和“/”等价，不区分大小写。

三、关系表达式：等于（==）、不等于(!=)、大于(>)、大于等于(>=)、小于(<)、小于等于(<=)，区间（between）运算，如“parser.parseExpression("1>2").getValue(boolean.class);”将返回 false；而“parser.parseExpression("1 between {1, 2}").getValue(boolean.class);”将返回 true。

between 运算符右边操作数必须是列表类型，且只能包含 2 个元素。第一个元素为开始，第二个元素为结束，区间运算是包含边界值的，即 xxx>=list.get(0) && xxx<=list.get(1)。

SpEL 同样提供了等价的“EQ”、“NE”、“GT”、“GE”、“LT”、“LE”来表示等于、不等于、大于、大于等于、小于、小于等于，不区分大小写。

四、逻辑表达式：且（and）、或(or)、非(!或 NOT)。

```
String expression1 = "2>1 and (!true or !false)";
boolean result1 = parser.parseExpression(expression1).getValue(boolean.class);
Assert.assertEquals(true, result1);

String expression2 = "2>1 and (NOT true or NOT false)";
boolean result2 = parser.parseExpression(expression2).getValue(boolean.class);
Assert.assertEquals(true, result2);
```

注：**逻辑运算符不支持 Java 中的 && 和 || 。**

五、字符串连接及截取表达式：使用“+”进行字符串连接，使用“'String'[0] [index]”来截取一个字符，目前只支持截取一个，如“'Hello ' + 'World!'”得到“Hello World!”；而“'Hello World!'[0]”将返回“H”。

六、三目运算及 Elivis 运算表达式：

三目运算符 **“表达式 1?表达式 2:表达式 3”**用于构造三目运算表达式，如“2>1?true:false”将返回 true；

Elivis 运算符**“表达式 1?:表达式 2”**从 Groovy 语言引入用于简化三目运算符的，当表达式 1 为非 null 时则返回表达式 1，当表达式 1 为 null 时则返回表达式 2，简化了三目运算符方式“表达式 1? 表达式 1:表达式 2”，如“null?:false”将返回 false，而“true?:false”将返回 true；

七、正则表达式：使用“str matches regex，如“'123' matches '\\d{3}'”将返回 true；

八、括号优先级表达式：使用“(表达式)”构造，括号里的具有高优先级。

5.3.3 类相关表达式

一、类类型表达式：使用“T(Type)”来表示 java.lang.Class 实例，“Type”必须是类全限定名，“java.lang”包除外，即该包下的类可以不指定包名；使用类类型表达式还可以进行访问类静态方法及类静态字段。

具体使用方法如下：

```
@Test
public void testClassTypeExpression() {
    ExpressionParser parser = new SpelExpressionParser();
    //java.lang包类访问
    Class<String> result1 = parser.parseExpression("T(String)").getValue(Class.class);
    Assert.assertEquals(String.class, result1);
    //其他包类访问
    String expression2 = "T(cn.javass.spring.chapter5.SpELTest)";
    Class<String> result2 = parser.parseExpression(expression2).getValue(Class.class);
    Assert.assertEquals(SpELTest.class, result2);
    //类静态字段访问
    int result3=parser.parseExpression("T(Integer).MAX_VALUE").getValue(int.class);
    Assert.assertEquals(Integer.MAX_VALUE, result3);
    //类静态方法调用
    int result4 = parser.parseExpression("T(Integer).parseInt('1')").getValue(int.class);
    Assert.assertEquals(1, result4);
}
```

对于 java.lang 包里的可以直接使用“T(String)”访问；其他包必须是类全限定名；可以进行静态字段访问如“T(Integer).MAX_VALUE”；也可以进行静态方法访问如“T(Integer).parseInt('1')”。

二、类实例化：类实例化同样使用 java 关键字“new”，类名必须是全限定名，但 java.lang 包内的类型除外，如 String、Integer。

```
@Test
public void testConstructorExpression() {
    ExpressionParser parser = new SpelExpressionParser();
    String result1 = parser.parseExpression("new String('haha')").getValue(String.class);
    Assert.assertEquals("haha", result1);
    Date result2 = parser.parseExpression("new java.util.Date()").getValue(Date.class);
    Assert.assertNotNull(result2);
}
```

实例化完全跟 Java 内方式一样。

三、instanceof 表达式：SpEL 支持 instanceof 运算符，跟 Java 内使用同义；如“'haha' instanceof T(String)”将返回 true。

四、变量定义及引用：变量定义通过 EvaluationContext 接口的 setVariable(variableName, value)方法定义；在表达式中使用“#variableName”引用；除了引用自定义变量，SpE 还允许引用根对象及当前上下文对象，使用“#root”引用根对象，使用“#this”引用当前上下文对象；

```
@Test
public void testVariableExpression() {
    ExpressionParser parser = new SpelExpressionParser();
    EvaluationContext context = new StandardEvaluationContext();
    context.setVariable("variable", "haha");
    context.setVariable("variable", "haha");
    String result1 = parser.parseExpression("#variable").getValue(context, String.class);
    Assert.assertEquals("haha", result1);

    context = new StandardEvaluationContext("haha");
    String result2 = parser.parseExpression("#root").getValue(context, String.class);
    Assert.assertEquals("haha", result2);
    String result3 = parser.parseExpression("#this").getValue(context, String.class);
    Assert.assertEquals("haha", result3);
}
```

使用“#variable”来引用在 EvaluationContext 定义的变量；除了可以引用自定义变量，还可以使用“#root”引用根对象，“#this”引用当前上下文对象，此处“#this”即根对象。

五、自定义函数：目前只支持类静态方法注册为自定义函数；SpEL 使用 StandardEvaluationContext 的 registerFunction 方法进行注册自定义函数，其实完全可以使用 setVariable 代替，两者其实本质是一样的；

```
@Test
public void testFunctionExpression() throws SecurityException,
    NoSuchMethodException {
    ExpressionParser parser = new SpelExpressionParser();
    StandardEvaluationContext context = new StandardEvaluationContext();
    Method parseInt = Integer.class.getDeclaredMethod("parseInt", String.class);
    context.registerFunction("parseInt", parseInt);
    context.setVariable("parseInt2", parseInt);
    String expression1 = "#parseInt('3') == #parseInt2('3')";
    boolean result1 =
            parser.parseExpression(expression1).getValue(context, boolean.class);
    Assert.assertEquals(true, result1);
}
```

此处可以看出“registerFunction”和“setVariable”都可以注册自定义函数，但是两个方法的含义不一样，推荐使用“registerFunction”方法注册自定义函数。

六、赋值表达式：SpEL 即允许给自定义变量赋值，也允许给跟对象赋值，直接使用

“#variableName=value”即可赋值：

```
@Test
public void testAssignExpression() {
    ExpressionParser parser = new SpelExpressionParser();
    //1.给root对象赋值
    EvaluationContext context = new StandardEvaluationContext("aaaa");
    String result1 =
        parser.parseExpression("#root='aaaaa'").getValue(context, String.class);
    Assert.assertEquals("aaaaa", result1);
    String result2 =
        parser.parseExpression("#this='aaaa'").getValue(context, String.class);
    Assert.assertEquals("aaaa", result2);

    //2.给自定义变量赋值
    context.setVariable("#variable", "variable");
    String result3 =
        parser.parseExpression("#variable=#root").getValue(context, String.class);
    Assert.assertEquals("aaaa", result3);
}
```

使用“#root='aaaaa'”给根对象赋值，使用“'#this='aaaa'”给当前上下文对象赋值，使用“#variable=#root”给自定义变量赋值，很简单。

七、对象属性存取及安全导航表达式：对象属性获取非常简单，即使用如“a.property.property”这种点缀式获取，SpEL 对于属性名首字母是不区分大小写的；SpEL 还引入了 Groovy 语言中的安全导航运算符“(对象|属性)?.属性”，用来避免但“?.”前边的表达式为 null 时抛出空指针异常，而是返回 null；修改对象属性值则可以通过赋值表达式或 Expression 接口的 setValue 方法修改。

```
ExpressionParser parser = new SpelExpressionParser();
//1.访问root对象属性
Date date = new Date();
StandardEvaluationContext context = new StandardEvaluationContext(date);
int result1 = parser.parseExpression("Year").getValue(context, int.class);
Assert.assertEquals(date.getYear(), result1);
int result2 = parser.parseExpression("year").getValue(context, int.class);
Assert.assertEquals(date.getYear(), result2);
```

对于**当前上下文对象属性及方法访问，可以直接使用属性或方法名访问**，比如此处根对象 date 属性“year”，注意此处属性名首字母不区分大小写。

```
//2.安全访问
context.setRootObject(null);
Object result3 = parser.parseExpression("#root?.year").getValue(context, Object.class);
Assert.assertEquals(null, result3);
```

SpEL 引入了 Groovy 的安全导航运算符，比如此处根对象为 null，所以如果访问其属性时肯定抛出空指针异常，而采用“?.”安全访问导航运算符将不抛空指针异常，而是简单的返回 null。

```
//3.给root对象属性赋值
context.setRootObject(date);
int result4 = parser.parseExpression("Year = 4").getValue(context, int.class);
Assert.assertEquals(4, result4);
parser.parseExpression("Year").setValue(context, 5);
int result5 = parser.parseExpression("Year").getValue(context, int.class);
Assert.assertEquals(5, result5);
```

给对象属性赋值可以采用赋值表达式或 Expression 接口的 setValue 方法赋值，而且也可以采用点缀方式赋值。

八、对象方法调用：对象方法调用更简单，跟 Java 语法一样；如“'haha'.substring(2,4)”将返回“ha”；而对于根对象可以直接调用方法；

```
Date date = new Date();
StandardEvaluationContext context = new StandardEvaluationContext(date);
int result2 = parser.parseExpression("getYear()").getValue(context, int.class);
Assert.assertEquals(date.getYear(), result2);
```

比如根对象 date 方法“getYear”可以直接调用。

九、Bean 引用：SpEL 支持使用“@”符号来引用 Bean，在引用 Bean 时需要使用 BeanResolver 接口实现来查找 Bean，Spring 提供 BeanFactoryResolver 实现；

```
@Test
ublic void testBeanExpression() {
    ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext();
    ctx.refresh();
    ExpressionParser parser = new SpelExpressionParser();
    StandardEvaluationContext context = new StandardEvaluationContext();
    context.setBeanResolver(new BeanFactoryResolver(ctx));
    Properties result1 =
      parser.parseExpression("@systemProperties").getValue(context, Properties.class);
    Assert.assertEquals(System.getProperties(), result1);
}
```

在示例中我们首先初始化了一个 IoC 容器，ClassPathXmlApplicationContext 实现默认会把“System.getProperties()”注册为“systemProperties”Bean，因此我们使用“@systemProperties”来引用该 Bean。

5.3.3 集合相关表达式

一、内联 List：从 Spring3.0.4 开始支持内联 List，使用{表达式，……}定义内联 List，如“{1,2,3}”将返回一个整型的 ArrayList，而“{}”将返回空的 List，对于字面量表达式列表，SpEL 会使用 java.util.Collections.unmodifiableList 方法将列表设置为不可修改。

```
//将返回不可修改的空 List
List<Integer> result2 = parser.parseExpression("{}").getValue(List.class);
```

```
//对于字面量列表也将返回不可修改的List
List<Integer> result1 = parser.parseExpression("{1,2,3}").getValue(List.class);
Assert.assertEquals(new Integer(1), result1.get(0));
try {
    result1.set(0, 2);
    //不可能执行到这，对于字面量列表不可修改
    Assert.fail();
} catch (Exception e) {
}
```

```
//对于列表中只要有一个不是字面量表达式，将只返回原始List，
//不会进行不可修改处理
String expression3 = "{{1+2,2+4},{3,4+4}}";
List<List<Integer>> result3 = parser.parseExpression(expression3).getValue(List.class);
result3.get(0).set(0, 1);
Assert.assertEquals(2, result3.size());
```

二、内联数组：和 Java 数组定义类似，只是在定义时进行多维数组初始化。

```
//定义一维数组并初始化
int[] result1 = parser.parseExpression("new int[1]").getValue(int[].class);
```

```
//声明二维数组并初始化
int[] result2 = parser.parseExpression("new int[2]{1,2}").getValue(int[].class);
```

```
//定义多维数组但不初始化
int[][][] result3 = parser.parseExpression(expression3).getValue(int[][][].class);
```

```
//错误的定义多维数组，多维数组不能初始化
String expression4 = "new int[1][2][3]{{1}{2}{3}}";
try {
    int[][][] result4 = parser.parseExpression(expression4).getValue(int[][][].class);
    Assert.fail();
} catch (Exception e) {
}
```

三、集合，字典元素访问：SpEL 目前支持所有集合类型和字典类型的元素访问，使用“集合[索引]”访问集合元素，使用“map[key]”访问字典元素；

```
//SpEL内联List访问
int result1 = parser.parseExpression("{1,2,3}[0]").getValue(int.class);
//即list.get(0)
Assert.assertEquals(1, result1);
```

```
//SpEL目前支持所有集合类型的访问
Collection<Integer> collection = new HashSet<Integer>();
collection.add(1);
collection.add(2);
EvaluationContext context2 = new StandardEvaluationContext();
context2.setVariable("collection", collection);
int result2 = parser.parseExpression("#collection[1]").getValue(context2, int.class);
//对于任何集合类型通过Iterator来定位元素
Assert.assertEquals(2, result2);
```

```
//SpEL对Map字典元素访问的支持
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 1);
EvaluationContext context3 = new StandardEvaluationContext();
context3.setVariable("map", map);
int result3 = parser.parseExpression("#map['a']").getValue(context3, int.class);
Assert.assertEquals(1, result3);
```

注：**集合元素访问是通过 Iterator 遍历来定位元素位置的**。

四、列表，字典，数组元素修改：可以使用赋值表达式或 Expression 接口的 setValue 方法修改；

```
//1.修改数组元素值
int[] array = new int[] {1, 2};
EvaluationContext context1 = new StandardEvaluationContext();
context1.setVariable("array", array);
int result1 = parser.parseExpression("#array[1] = 3").getValue(context1, int.class);
Assert.assertEquals(3, result1);
```

```
//2.修改集合值
Collection<Integer> collection = new ArrayList<Integer>();
collection.add(1);
collection.add(2);
EvaluationContext context2 = new StandardEvaluationContext();
context2.setVariable("collection", collection);
int result2 = parser.parseExpression("#collection[1] = 3").getValue(context2, int.class);
Assert.assertEquals(3, result2);
parser.parseExpression("#collection[1]").setValue(context2, 4);
result2 = parser.parseExpression("#collection[1]").getValue(context2, int.class);
Assert.assertEquals(4, result2);
```

```
//3.修改map元素值
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 1);
EvaluationContext context3 = new StandardEvaluationContext();
context3.setVariable("map", map);
int result3 = parser.parseExpression("#map['a'] = 2").getValue(context3, int.class);
Assert.assertEquals(2, result3);
```

对数组修改直接对“#array[index]”赋值即可修改元素值，同理适用于集合和字典类型。

五、集合投影：在 SQL 中投影指从表中选择出列，而在 SpEL 指根据集合中的元素中通过选择来构造另一个集合，该集合和原集合具有相同数量的元素；SpEL 使用“（list|map）.![投影表达式]”来进行投影运算：

```
//1.首先准备测试数据
Collection<Integer> collection = new ArrayList<Integer>();
collection.add(4);      collection.add(5);
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 1);        map.put("b", 2);
```

```
//2.测试集合或数组
EvaluationContext context1 = new StandardEvaluationContext();
context1.setVariable("collection", collection);
Collection<Integer> result1 =
    parser.parseExpression("#collection.![#this+1]").getValue(context1, Collection.class);
Assert.assertEquals(2, result1.size());
Assert.assertEquals(new Integer(5), result1.iterator().next());
```

对于集合或数组使用如上表达式进行投影运算，其中投影表达式中“#this”代表每个集合或数组元素，可以使用比如“#this.property”来获取集合元素的属性，其中“#this”可以省略。

```
//3.测试字典
EvaluationContext context2 = new StandardEvaluationContext();
context2.setVariable("map", map);
List<Integer> result2 =
      parser.parseExpression("#map.![ value+1]").getValue(context2, List.class);
Assert.assertEquals(2, result2.size());
```

SpEL 投影运算还支持 Map 投影，但 Map 投影最终只能得到 List 结果，如上所示，对于投影表达式中的“#this”将是 Map.Entry，所以可以使用“value”来获取值，使用“key”来获取键。

六、集合选择：在 SQL 中指使用 select 进行选择行数据，而在 SpEL 指根据原集合通过条件表达式选择出满足条件的元素并构造为新的集合，SpEL 使用“(list|map).?[选择表达式]”，其中选择表达式结果必须是 boolean 类型，如果 true 则选择的元素将添加到新集合中，false 将不添加到新集合中。

```
//1.首先准备测试数据
Collection<Integer> collection = new ArrayList<Integer>();
collection.add(4);      collection.add(5);
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("a", 1);        map.put("b", 2);
```

```
//2.集合或数组测试
EvaluationContext context1 = new StandardEvaluationContext();
context1.setVariable("collection", collection);
Collection<Integer> result1 =
   parser.parseExpression("#collection.?[#this>4]").getValue(context1, Collection.class);
Assert.assertEquals(1, result1.size());
Assert.assertEquals(new Integer(5), result1.iterator().next());
```

对于集合或数组选择，如“#collection.?[#this>4]”将选择出集合元素值大于 4 的所有元素。选择表达式必须返回布尔类型，使用“#this”表示当前元素。

```
//3.字典测试
EvaluationContext context2 = new StandardEvaluationContext();
context2.setVariable("map", map);
Map<String, Integer> result2 =
      parser.parseExpression("#map.?[#this.key != 'a']").getValue(context2, Map.class);
Assert.assertEquals(1, result2.size());

List<Integer> result3 =
   parser.parseExpression("#map.?[key != 'a'].![value+1]").getValue(context2, List.class);
Assert.assertEquals(new Integer(3), result3.iterator().next());
```

对于字典选择，如“#map.?[#this.key != 'a']”将选择键值不等于”a”的，其中选择表达式中“#this”是 Map.Entry 类型，而最终结果还是 Map，这点和投影不同；集合选择和投影可以一起使用，如“#map.?[key != 'a'].![value+1]”将首先选择键值不等于”a”的，然后在选出的 Map 中再进行“value+1”的投影。

5.3.4 表达式模板

模板表达式就是由字面量与一个或多个表达式块组成。每个表达式块由“前缀+表达式+后缀”形式组成，如“${1+2}”即表达式块。在前边我们已经介绍了使用 ParserContext 接口实现来定义表达式是否是模板及前缀和后缀定义。在此就不多介绍了，如“Error ${#v0} ${#v1}”表达式表示由字面量“Error ”、模板表达式“#v0”、模板表达式“#v1”组成，其中 v0 和 v1 表示自定义变量，需要在上下文定义。

5.4 在 Bean 定义中使用 EL

5.4.1 xml 风格的配置

SpEL 支持在 Bean 定义时注入，默认使用“#{SpEL 表达式}”表示，其中“#root”根对象默认可以认为是 ApplicationContext，只有 ApplicationContext 实现默认支持 SpEL，获取根对象属性其实是获取容器中的 Bean。

首先看下配置方式（chapter5/el1.xml）吧：

```
<bean id="world" class="java.lang.String">
    <constructor-arg value="#{' World!'}"/>
</bean>
<bean id="hello1" class="java.lang.String">
    <constructor-arg value="#{'Hello'}#{world}"/>
</bean>
<bean id="hello2" class="java.lang.String">
    <constructor-arg value="#{'Hello' + world}"/>
    <!-- 不支持嵌套的 -->
    <!--<constructor-arg value="#{'Hello'#{world}}"/>-->
</bean>
<bean id="hello3" class="java.lang.String">
    <constructor-arg value="#{'Hello' + @world}"/>
</bean>
```

模板默认以前缀“#{”开头，以后缀“}”结尾，且不允许嵌套，如“#{'Hello'#{world}}”错误，如“#{'Hello' + world}”中“world”默认解析为 Bean。当然可以使用“@bean”引用了。

接下来测试一下吧：

```
@Test
public void testXmlExpression() {
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext("chapter5/el1.xml");
    String hello1 = ctx.getBean("hello1", String.class);
    String hello2 = ctx.getBean("hello2", String.class);
    String hello3 = ctx.getBean("hello3", String.class);
    Assert.assertEquals("Hello World!", hello1);
    Assert.assertEquals("Hello World!", hello2);
    Assert.assertEquals("Hello World!", hello3);
}
```

是不是很简单，除了 XML 配置方式，Spring 还提供一种注解方式@Value，接着往下看吧。

5.4.2 注解风格的配置

基于注解风格的 SpEL 配置也非常简单，使用@Value 注解来指定 SpEL 表达式，该注解可以放到字段、方法及方法参数上。

测试 Bean 类如下，使用@Value 来指定 SpEL 表达式：

```
package cn.javass.spring.chapter5;
import org.springframework.beans.factory.annotation.Value;
public class SpELBean {
    @Value("#{'Hello' + world}")
    private String value;
    //setter和getter由于篇幅省略，自己写上
}
```

首先看下配置文件(chapter5/el2.xml)：

```
<?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:context="http://www.springframework.org/schema/context"
          xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <context:annotation-config/>
    <bean id="world" class="java.lang.String">
        <constructor-arg value="#{' World!'}"/>
    </bean>
    <bean id="helloBean1" class="cn.javass.spring.chapter5.SpELBean"/>
    <bean id="helloBean2" class="cn.javass.spring.chapter5.SpELBean">
        <property name="value" value="haha"/>
    </bean>
</beans>
```

配置时必须使用“**<context:annotation-config/>**”来开启对注解的支持。

有了配置文件那开始测试吧：

```
@Test
public void testAnnotationExpression() {
      ApplicationContext ctx =
            new ClassPathXmlApplicationContext("chapter5/el2.xml");
      SpELBean helloBean1 = ctx.getBean("helloBean1", SpELBean.class);
      Assert.assertEquals("Hello World!", helloBean1.getValue());
      SpELBean helloBean2 = ctx.getBean("helloBean2", SpELBean.class);
      Assert.assertEquals("haha", helloBean2.getValue());
}
```

其中“helloBean1 ”值是 SpEL 表达式的值，而“helloBean2”是通过 setter 注入的值，这说明 setter 注入将覆盖@Value 的值。

5.4.3 在 Bean 定义中 SpEL 的问题

如果有同学问“#{我不是 SpEL 表达式}”不是 SpEL 表达式，而是公司内部的模板，想换个前缀和后缀该如何实现呢？

那我们来看下 Spring 如何在 IoC 容器内使用 BeanExpressionResolver 接口实现来求值 SpEL 表达式，那如果我们通过某种方式获取该接口实现，然后把前缀后缀修改了不就可以了。

此处我们使用 BeanFactoryPostProcessor 接口提供 postProcessBeanFactory 回调方法，它是在 IoC 容器创建好但还未进行任何 Bean 初始化时被 ApplicationContext 实现调用，因此在这个阶段把 SpEL 前缀及后缀修改掉是安全的，具体代码如下：

```
package cn.javass.spring.chapter5;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.context.expression.StandardBeanExpressionResolver;
public class SpELBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
          throws BeansException {
          StandardBeanExpressionResolver resolver =
              (StandardBeanExpressionResolver) beanFactory.getBeanExpressionResolver();
          resolver.setExpressionPrefix("%{");
          resolver.setExpressionSuffix("}");
    }
}
```

首先通过 ConfigurableListableBeanFactory 的 getBeanExpressionResolver 方法获取 BeanExpressionResolver 实现，其次强制类型转换为 StandardBeanExpressionResolver，其为 Spring 默认实现，然后改掉前缀及后缀。

开始测试吧，首先准备配置文件(chapter5/el3.xml)：

```
<?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:context="http://www.springframework.org/schema/context"
          xsi:schemaLocation="
              http://www.springframework.org/schema/beans
              http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
              http://www.springframework.org/schema/context
              http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <context:annotation-config/>
    <bean class="cn.javass.spring.chapter5.SpELBeanFactoryPostProcessor"/>
    <bean id="world" class="java.lang.String">
        <constructor-arg value="%{' World!'}"/>
    </bean>
    <bean id="helloBean1" class="cn.javass.spring.chapter5.SpELBean"/>
    <bean id="helloBean2" class="cn.javass.spring.chapter5.SpELBean">
        <property name="value" value="%{'Hello' + world}"/>
    </bean>
</beans>
```

配置文件和注解风格的几乎一样，只有 SpEL 表达式前缀变为“%{”了，并且注册了“cn.javass.spring.chapter5.SpELBeanFactoryPostProcessor”Bean，用于修改前缀和后缀的。

写测试代码测试一下吧：

```
@Test
public void testPrefixExpression() {
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext("chapter5/el3.xml");
    SpELBean helloBean1 = ctx.getBean("helloBean1", SpELBean.class);
    Assert.assertEquals("#{'Hello' + world}", helloBean1.getValue());
    SpELBean helloBean2 = ctx.getBean("helloBean2", SpELBean.class);
    Assert.assertEquals("Hello World!", helloBean2.getValue());
}
```

此处 helloBean1 中通过@Value 注入的“#{'Hello' + world}”结果还是“#{'Hello' +

world}”说明不对其进行 SpEL 表达式求值了，而 helloBean2 使用“%{'Hello' + world}”注入，得到正确的“"Hello World!”。

第六章　Spring 的 AOP

6.1　AOP 基础

6.1.1　AOP 是什么

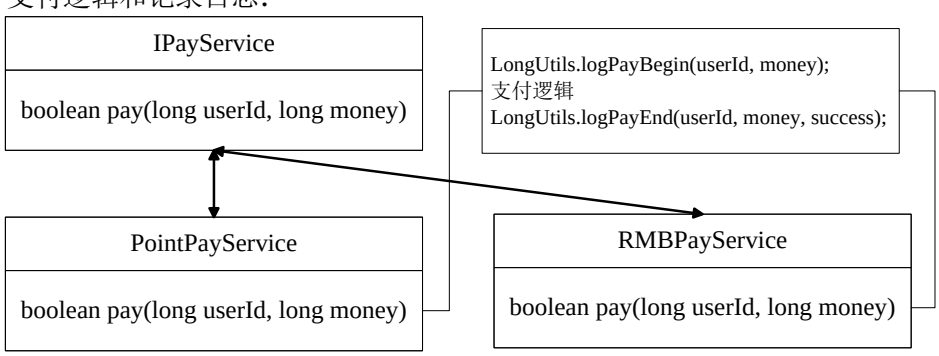
- 考虑这样一个问题：需要对系统中的某些业务做日志记录，比如支付系统中的支付业务需要记录支付相关日志，对于支付系统可能相当复杂，比如可能有自己的支付系统，也可能引入第三方支付平台，面对这样的支付系统该如何解决呢？

- **传统解决方案：**

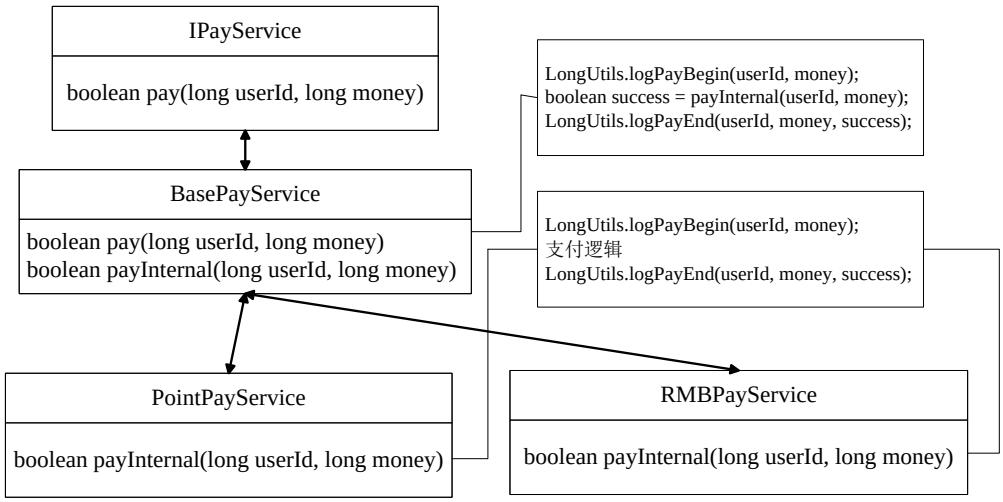
  1）日志部分提前公共类 LogUtils，定义“longPayBegin”方法用于记录支付开始日志，“logPayEnd”用于记录支付结果：

LogUtils
logPayBegin(long userId, long money) logPayEnd(long userId, long money, boolean success)

2）支付部分，定义 IPayService 接口并定义支付方法“pay”，并定义了两个实现：“PointPayService”表示积分支付，“RMBPayService”表示人民币支付；并且在每个支付实现中支付逻辑和记录日志：



3）支付实现很明显有重复代码，这个重复很明显可以使用模板设计模式消除重复：

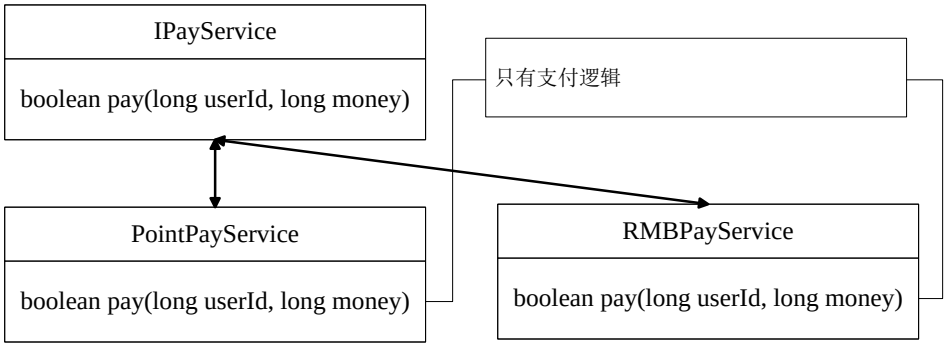


4）到此我们设计了一个可以复用的接口；但大家觉得这样记录日志会很好吗，有没有更好的解决方案？

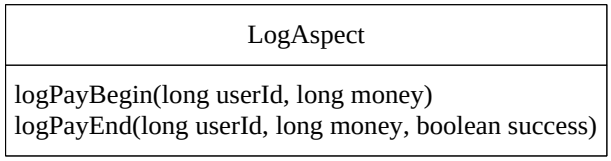
如果对积分支付方式添加统计功能，比如在支付时记录下用户总积分数、当前消费的积分数，那我们该如何做呢？直接修改源代码添加日志记录，这完全违背了面向对象最重要的原则之一：开闭原则（对扩展开放，对修改关闭）？

- **更好的解决方案**：在我们的支付组件中由于使用了日志组件，即日志模块横切于支付组件，在传统程序设计中很难将日志组件分离出来，即不耦合我们的支付组件；因此面向方面编程 AOP 就诞生了，它能分离我们的组件，使组件完全不耦合:

  1）采用面向方面编程后，我们的支付组件看起来如下所示，代码中不再有日志组件的任何东西；



  2）所以日志相关的提取到一个切面中，AOP 实现者会在合适的时候将日志功能织入到我们的支付组件中去，从而完全解耦支付组件和日志组件。



  看到这大家可能不是很理解，没关系，先往下看。

面向方面编程(AOP)：也可称为面向切面编程，是一种编程范式，提供从另一个角度来考虑程序结构从而完善面向对象编程(OOP)。

在进行 OOP 开发时，都是基于对组件（比如类）进行开发，然后对组件进行组合，OOP 最大问题就是无法解耦组件进行开发，比如我们上边举例，而 AOP 就是为了克服这个问题而出现的，它来进行这种耦合的分离。

AOP 为开发者提供一种进行横切关注点（比如日志关注点横切了支付关注点）分离并织入的机制，把横切关注点分离，然后通过某种技术织入到系统中，从而无耦合的完成了我们的功能。

6.1.2 能干什么

AOP 主要用于横切关注点分离和织入，因此需要理解横切关注点和织入：

- **关注点**：可以认为是所关注的任何东西，比如上边的支付组件；
- **关注点分离**：将问题细化从而单独部分，即可以理解为不可再分割的组件，如上边的日志组件和支付组件；
- **横切关注点**：一个组件无法完成需要的功能，需要其他组件协作完成，如日志组件横切于支付组件；
- **织入**：横切关注点分离后，需要通过某种技术将横切关注点融合到系统中从而完成需要的功能，因此需要织入，织入可能在编译期、加载期、运行期等进行。

横切关注点可能包含很多，比如非业务的：日志、事务处理、缓存、性能统计、权限控制等等这些非业务的基础功能；还可能是业务的：如某个业务组件横切于多个模块。如图 6-1

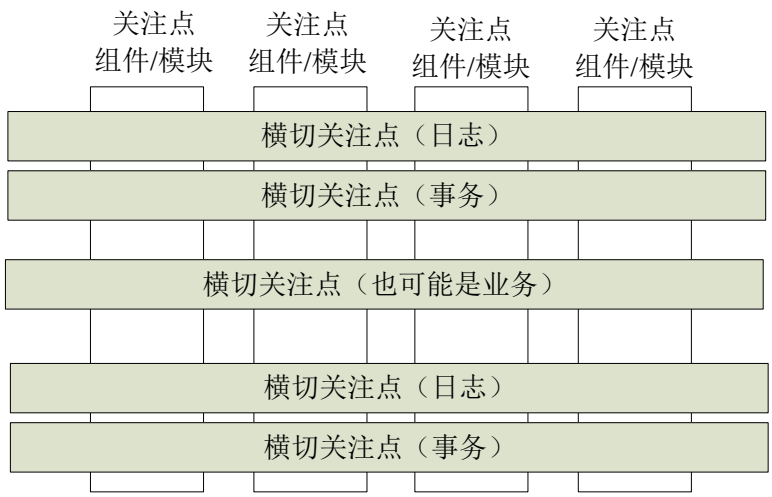
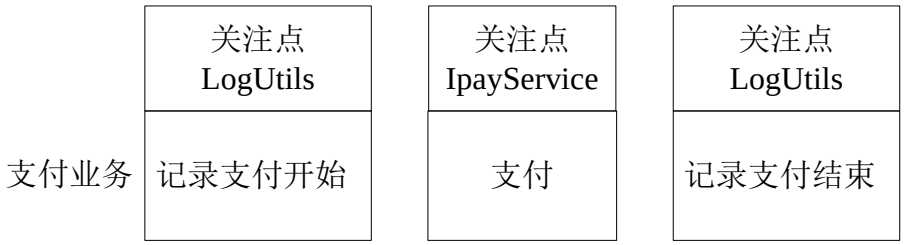
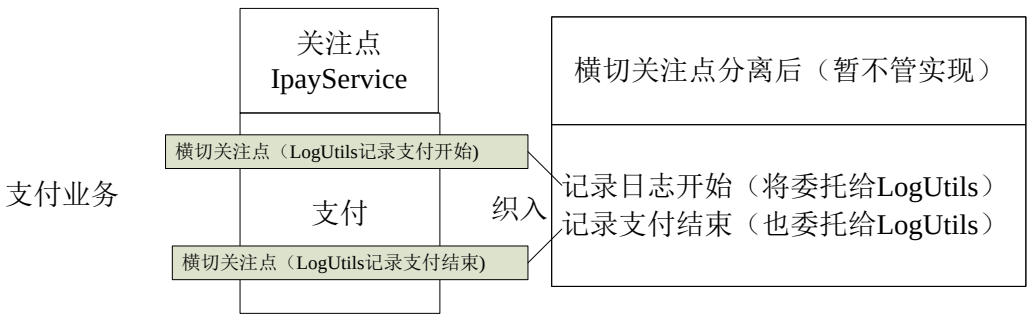


图 6-1 关注点与横切关注点

传统支付形式，流水方式：



面向切面方式，先将横切关注点分离，再将横切关注点织入到支付系统中：



AOP 能干什么：

- 用于横切关注点的分离和织入横切关注点到系统；比如上边提到的日志等等；
- 完善 OOP；
- 降低组件和模块之间的耦合性；
- 使系统容易扩展；
- 而且由于关注点分离从而可以获得组件的更好复用。

6.1.3 AOP 的基本概念

在进行 AOP 开发前，先熟悉几个概念：

- **连接点（Jointpoint）：**表示需要在程序中插入横切关注点的扩展点，连接点可能是类初始化、方法执行、方法调用、字段调用或处理异常等等，Spring 只支持方法执行连接点，**在 AOP 中表示为“在哪里干”**；
- **切入点（Pointcut）：**选择一组相关连接点的模式，即可以认为连接点的集合，Spring 支持 perl5 正则表达式和 AspectJ 切入点模式，Spring 默认使用 AspectJ 语法，**在 AOP 中表示为“在哪里干的集合”**；
- **通知（Advice）：**在连接点上执行的行为，通知提供了在 AOP 中需要在切入点所选择的连接点处进行扩展现有行为的手段；包括前置通知（before advice）、后置通知(after advice)、环绕通知（around advice），在 Spring 中通过代理模式实现 AOP，并通过拦截器模式以环绕连接点的拦截器链织入通知；**在 AOP 中表示为“干什么”；**
- **方面/切面（Aspect）：**横切关注点的模块化，比如上边提到的日志组件。可以认

为是通知、引入和切入点的组合；在 Spring 中可以使用 Schema 和@AspectJ 方式进行组织实现；**在 AOP 中表示为“在哪干和干什么集合”；**

- **引入（inter-type declaration）：**也称为内部类型声明，为已有的类添加额外新的字段或方法，Spring 允许引入新的接口（必须对应一个实现）到所有被代理对象（目标对象），**在 AOP 中表示为“干什么（引入什么）”；**
- **目标对象（Target Object）：**需要被织入横切关注点的对象，即该对象是切入点选择的对象，需要被通知的对象，从而也可称为“被通知对象”；由于 Spring AOP 通过代理模式实现，从而这个对象永远是被代理对象，**在 AOP 中表示为“对谁干”；**
- **AOP 代理（AOP Proxy）：**AOP 框架使用代理模式创建的对象，从而实现在连接点处插入通知（即应用切面），就是**通过代理来对目标对象应用切面**。在 Spring 中，AOP 代理可以用 JDK 动态代理或 CGLIB 代理实现，而通过拦截器模型应用切面。
- **织入（Weaving）：**织入是一个过程，是将切面应用到目标对象从而创建出 AOP 代理对象的过程，织入可以在编译期、类装载期、运行期进行。

在 AOP 中，通过切入点选择目标对象的连接点，然后在目标对象的相应连接点处织入通知，而切入点和通知就是切面（横切关注点），而在目标对象连接点处应用切面的实现方式是通过 AOP 代理对象，如图 6-2 所示。

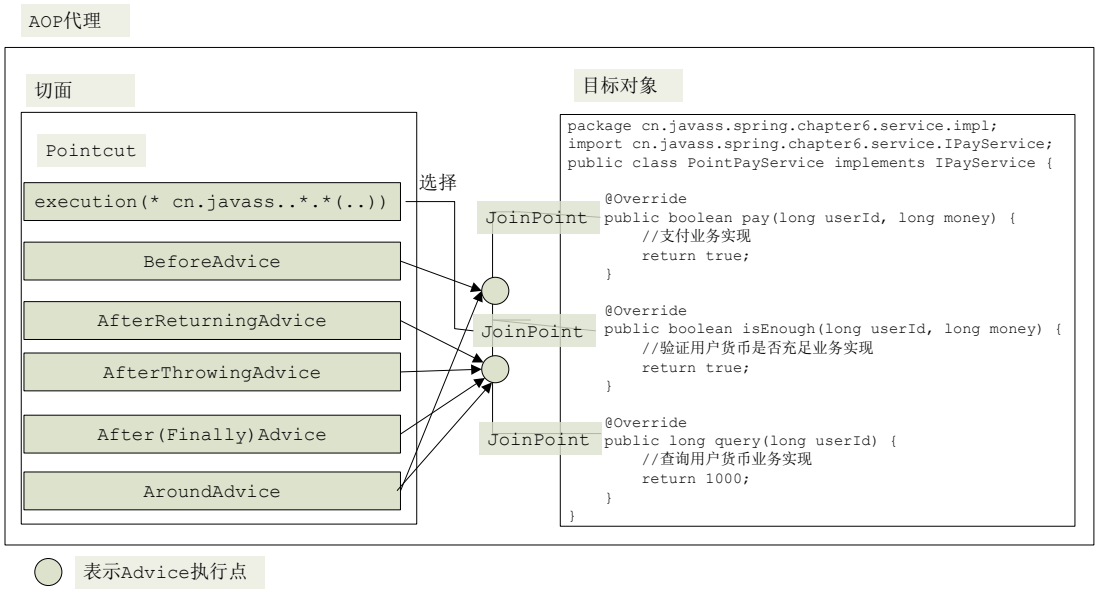


图 6-2 概念关系

接下来再让我们具体看看 Spring 有哪些通知类型：

- **前置通知（Before Advice）:**在切入点选择的连接点处的方法之前执行的通知，该通知不影响正常程序执行流程（除非该通知抛出异常，该异常将中断当前方法链的执行而返回）。
- **后置通知（After Advice）：**在切入点选择的连接点处的方法之后执行的通知，包括如下类型的后置通知：

- **后置返回通知（After returning Advice）:**在切入点选择的连接点处的方法正常执行完毕时执行的通知，必须是连接点处的方法没抛出任何异常正常返回时才调用后置通知。
- **后置异常通知（After throwing Advice）:** 在切入点选择的连接点处的方法抛出异常返回时执行的通知，必须是连接点处的方法抛出任何异常返回时才调用异常通知。
- **后置最终通知（After finally Advice）:** 在切入点选择的连接点处的方法返回时执行的通知，不管抛没抛出异常都执行，类似于 Java 中的 finally 块。

➢ **环绕通知（Around Advices）:** 环绕着在切入点选择的连接点处的方法所执行的通知，环绕通知可以在方法调用之前和之后自定义任何行为，并且可以决定是否执行连接点处的方法、替换返回值、抛出异常等等。

各种通知类型在 UML 序列图中的位置如图 6-3 所示：

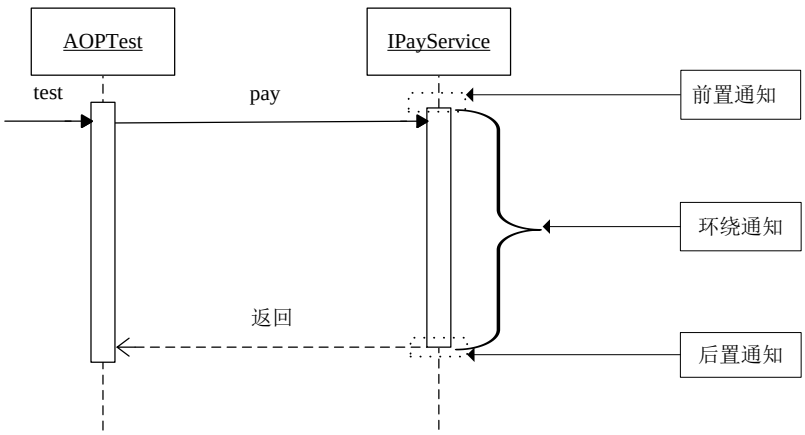


图 6-3 通知类型

6.1.4 AOP 代理

AOP 代理就是 AOP 框架通过代理模式创建的对象，Spring 使用 JDK 动态代理或 CGLIB 代理来实现，Spring 缺省使用 JDK 动态代理来实现，从而任何接口都可别代理，如果被代理的对象实现不是接口将默认使用 CGLIB 代理，不过 CGLIB 代理当然也可应用到接口。

AOP 代理的目的就是将切面织入到目标对象。

概念都将完了，接下来让我们看一下 AOP 的 HelloWorld!吧。

6.2 AOP 的 HelloWorld

6.2.1 准备环境

首先准备开发需要的 jar 包，请到 spring-framework-3.0.5.RELEASE-dependencies.zip 和 spring-framework-3.0.5.RELEASE-with-docs 中查找如下 jar 包：

```
org.springframework.aop-3.0.5.RELEASE.jar
com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar
com.springsource.org.aopalliance-1.0.0.jar
com.springsource.net.sf.cglib-2.2.0.jar
```

将这些 jar 包添加到“Build Path”下。

6.2.2 定义目标类

1）定义目标接口：

```
package cn.javass.spring.chapter6.service;
public interface IHelloWorldService {
    public void sayHello();
}
```

2）定义目标接口实现：

```
package cn.javass.spring.chapter6.service.impl;
import cn.javass.spring.chapter6.service.IHelloWorldService;
public class HelloWorldService implements IHelloWorldService {
    @Override
    public void sayHello() {
        System.out.println("============Hello World!");
    }
}
```

注：在日常开发中最后将业务逻辑定义在一个专门的 service 包下，而实现定义在 service 包下的 impl 包中，服务接口以 IXXXService 形式，而服务实现就是 XXXService，这就是规约设计，见名知义。当然可以使用公司内部更好的形式，只要大家都好理解就可以了。

6.2.2 定义切面支持类

有了目标类，该定义切面了，切面就是通知和切入点的组合，而切面是通过配置方式定义的，因此这定义切面前，我们需要定义切面支持类，切面支持类提供了通知实现：

```
package cn.javass.spring.chapter6.aop;
public class HelloWorldAspect {
    //前置通知
    public void beforeAdvice() {
        System.out.println("===========before advice");
    }
    //后置最终通知
    public void afterFinallyAdvice() {
        System.out.println("===========after finally advice");
    }
}
```

此处 HelloWorldAspect 类不是真正的切面实现，只是定义了通知实现的类，在此我们可以把它看作就是缺少了切入点的切面。

注：对于 AOP 相关类最后专门放到一个包下，如“aop”包，因为 AOP 是动态织入的，所以如果某个目标类被 AOP 拦截了并应用了通知，可能很难发现这个通知实现在哪个包里，因此推荐使用规约命名，方便以后维护人员查找相应的 AOP 实现。

6.2.3 在 XML 中进行配置

有了通知实现，那就让我们来配置切面吧：

1）首先配置 AOP 需要 aop 命名空间，配置头如下：

```
<?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"
          xsi:schemaLocation="
              http://www.springframework.org/schema/beans
              http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
              http://www.springframework.org/schema/aop
              http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
</beans>
```

2）配置目标类：

```
<bean id="helloWorldService"
      class="cn.javass.spring.chapter6.service.impl.HelloWorldService"/>
```

3）配置切面：

```
<bean id="aspect" class="cn.javass.spring.chapter6.aop.HelloWorldAspect"/>
<aop:config>
    <aop:pointcut id="pointcut" expression="execution(* cn.javass..*.*(..))"/>
    <aop:aspect ref="aspect">
        <aop:before pointcut-ref="pointcut"
                    method="beforeAdvice"/>
        <aop:after pointcut="execution(* cn.javass..*.*(..))"
                   method="afterFinallyAdvice"/>
    </aop:aspect>
</aop:config>
```

切入点使用<aop:config>标签下的<aop:pointcut>配置，expression属性用于定义切入点模式，默认是AspectJ语法，“execution(* cn.javass..*.*(..))”表示匹配cn.javass包及子包下的任何方法执行。

切面使用<aop:config>标签下的<aop:aspect>标签配置，其中“ref”用来引用切面支持类的方法。

前置通知使用<aop:aspect>标签下的<aop:before>标签来定义，pointcut-ref属性用于引用切入点Bean，而method用来引用切面通知实现类中的方法，该方法就是通知实现，即在目标类方法执行之前调用的方法。

最终通知使用<aop:aspect>标签下的<aop:after >标签来定义，切入点除了使用pointcut-ref属性来引用已经存在的切入点，也可以使用pointcut属性来定义，如pointcut="execution(* cn.javass..*.*(..))"，method属性同样是指定通知实现，即在目标类方法执行之后调用的方法。

6.2.4 运行测试

测试类非常简单，调用被代理 Bean 跟调用普通 Bean 完全一样，Spring AOP 将为目标对象创建 AOP 代理，具体测试代码如下：

```
package cn.javass.spring.chapter6;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import cn.javass.spring.chapter6.service.IHelloWorldService;
import cn.javass.spring.chapter6.service.IPayService;
public class AopTest {
    @Test
    public void testHelloworld() {
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext("chapter6/helloworld.xml");
        IHelloWorldService helloworldService =
            ctx.getBean("helloWorldService", IHelloWorldService.class);
        helloworldService.sayHello();
    }
}
```

该测试将输出如下如下内容：

```
===========before advice
============Hello World!
===========after finally advice
```

从输出我们可以看出：前置通知在切入点选择的连接点（方法）之前允许，而后置通知将在连接点（方法）之后执行，具体生成 AOP 代理及执行过程如图 6-4 所示。

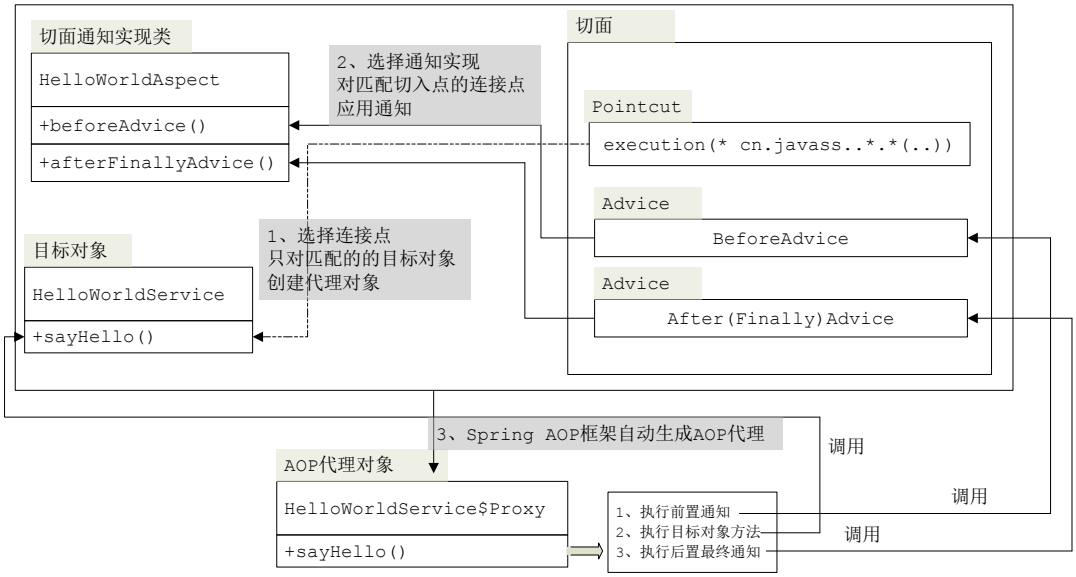


图 6-4 Spring AOP 框架生成 AOP 代理过程

6.3 基于 Schema 的 AOP

基于 Schema 的 AOP 从 Spring2.0 之后通过“aop”命名空间来定义切面、切入点及声明通知。

在 Spring 配置文件中，所以 AOP 相关定义必须放在<aop:config>标签下，该标签下可以有<aop:pointcut>、<aop:advisor>、<aop:aspect>标签，配置顺序不可变。

- <aop:pointcut>：用来定义切入点，该切入点可以重用；
- <aop:advisor>：用来定义只有一个通知和一个切入点的切面；
- <aop:aspect>：用来定义切面，该切面可以包含多个切入点和通知，而且标签内部的通知和切入点定义是无序的；和 advisor 的区别就在此，advisor 只包含一个通知和一个切入点。

<aop:config>	AOP定义开始（有序）
<aop:pointcut/>	切入点定义（零个或多个）
<aop:advisor/>	Advisor定义（零个或多个）
<aop:aspect>	切面定义开始（零个或多个，无序）
<aop:pointcut/>	切入点定义（零个或多个）
<aop:before"/>	前置通知（零个或多个）
<aop:after-returning/>	后置返回通知（零个或多个）
<aop:after-throwing/>	后置异常通知（零个或多个）
<aop:after/>	后置最终通知（零个或多个）
<aop:around/>	环绕通知（零个或多个）
<aop:declare-parents/>	引入定义（零个或多个）
</aop:aspect>	切面定义开始（零个或多个）
</aop:config>	AOP定义结束

6.3.1 声明切面

切面就是包含切入点和通知的对象，在 Spring 容器中将被定义为一个 Bean，Schema 方式的切面需要一个切面支持 Bean，该支持 Bean 的字段和方法提供了切面的状态和行为信息，并通过配置方式来指定切入点和通知实现。

切面使用<aop:aspect>标签指定，ref 属性用来引用切面支持 Bean。

```
<bean id="aspectSupportBean" class="……"/>
<aop:config>
    <aop:aspect id="aspectId" ref="aspectSupportBean">
      ……
    </aop:aspect>
</aop:config>
```

切面支持 Bean“aspectSupportBean”跟普通 Bean 完全一样使用，切面使用“ref”属性引用它。

6.3.2 声明切入点

切入点在 Spring 中也是一个 Bean，Bean 定义方式可以有很三种方式：

1）在<aop:config>标签下使用<aop:pointcut>声明一个切入点 Bean，该切入点可以被多个切面使用，对于需要共享使用的切入点最好使用该方式，该切入点使用 id 属性指定 Bean 名字，在通知定义时使用 pointcut-ref 属性通过该 id 引用切入点，expression 属性指定切入点表达式：

```
<aop:config>
    <aop:pointcut id="pointcut" expression="execution(* cn.javass..*.*(..))"/>
    <aop:aspect ref="aspectSupportBean">
        <aop:before pointcut-ref="pointcut" method="before"/>
    </aop:aspect>
</aop:config>
```

2）在<aop:aspect>标签下使用<aop:pointcut>声明一个切入点 Bean，该切入点可以被多个切面使用，但一般该切入点只被该切面使用，当然也可以被其他切面使用，但最好不要那样使用，该切入点使用 id 属性指定 Bean 名字，在通知定义时使用 pointcut-ref 属性通过该 id 引用切入点，expression 属性指定切入点表达式：

```
<aop:config>
    <aop:aspect ref="aspectSupportBean">
        <aop:pointcut id=" pointcut" expression="execution(* cn.javass..*.*(..))"/>
         <aop:before pointcut-ref="pointcut" method="before"/>
    </aop:aspect>
</aop:config>
```

3）匿名切入点 Bean，可以在声明通知时通过 pointcut 属性指定切入点表达式，该切入点是匿名切入点，只被该通知使用：

```
<aop:config>
    <aop:aspect ref="aspectSupportBean">
        <aop:after pointcut="execution(* cn.javass..*.*(..))"
                   method="afterFinallyAdvice"/>
    </aop:aspect>
</aop:config>
```

6.3.3 声明通知

基于 Schema 方式支持前边介绍的 5 中通知类型：

一、前置通知：在切入点选择的方法之前执行，通过<aop:aspect>标签下的<aop:before>标签声明：

```
<aop:before pointcut="切入点表达式"  pointcut-ref="切入点Bean引用"
            method="前置通知实现方法名"
            arg-names="前置通知实现方法参数列表参数名字"/>
```

- **pointcut 和 pointcut-ref：**二者选一，指定切入点；
- **method：**指定前置通知实现方法名，如果是多态需要加上参数类型，多个用“，”隔开，如 beforeAdvice(java.lang.String)；
- **arg-names：**指定通知实现方法的参数名字，多个用“，”分隔，可选，类似于【3.1.2 构造器注入】中的参数名注入限制：**在 class 文件中没生成变量调试信息是获取不到方法参数名字的，因此只有在类没生成变量调试信息时才需要使用 arg-names 属性来指定参数名，如** arg-names="param"表示通知实现方法的参数列表的第一个参数名字为“param”。

首先在 cn.javass.spring.chapter6.service.IhelloWorldService 定义一个测试方法：

```
public void sayBefore(String param);
```

其次在 cn.javass.spring.chapter6.service.impl. HelloWorldService 定义实现

```
@Override
public void sayBefore(String param) {
    System.out.println("============say " + param);
}
```

第三在 cn.javass.spring.chapter6.aop. HelloWorldAspect 定义通知实现：

```
public void beforeAdvice(String param) {
    System.out.println("===========before advice param:" + param);
}
```

最后在 chapter6/advice.xml 配置文件中进行如下配置：

```
<bean id="helloWorldService"
        class="cn.javass.spring.chapter6.service.impl.HelloWorldService"/>
<bean id="aspect" class="cn.javass.spring.chapter6.aop.HelloWorldAspect"/>
<aop:config>
    <aop:aspect ref="aspect">
        <aop:before pointcut="execution(* cn.javass..*.sayBefore(..))
                              and args(param)"
                    method="beforeAdvice(java.lang.String)"
                    arg-names="param"/>
    </aop:aspect>
</aop:config>
```

测试代码 cn.javass.spring.chapter6.AopTest:

```
@Test
public void testSchemaBeforeAdvice(){
    System.out.println("====================================");
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext("chapter6/advice.xml");
    IHelloWorldService helloworldService =
            ctx.getBean("helloWorldService", IHelloWorldService.class);
    helloworldService.sayBefore("before");
    System.out.println("====================================");
}
```

将输入：

```
==========================================
===========before advice param:before
============say before
==========================================
```

分析一下吧：

1）切入点匹配：在配置中使用“execution(* cn.javass..*.sayBefore(..)) ”匹配目标方法 sayBefore，且使用“args(param)”匹配目标方法只有一个参数且传入的参数类型为通知实现方法中同名的参数类型；

2）目标方法定义：使用 method=" beforeAdvice(java.lang.String) "指定前置通知实现方法，且该通知有一个参数类型为 java.lang.String 参数；

3）目标方法参数命名：其中使用 arg-names=" param "指定通知实现方法参数名为

“param”，切入点中使用“args(param)”匹配的目标方法参数将自动传递给通知实现方法同名参数。

二、后置返回通知：在切入点选择的方法正常返回时执行，通过<aop:aspect>标签下的<aop:after-returning>标签声明：

```
<aop:after-returning pointcut="切入点表达式"   pointcut-ref="切入点Bean引用"
                      method="后置返回通知实现方法名"
                     arg-names="后置返回通知实现方法参数列表参数名字"
                     returning="返回值对应的后置返回通知实现方法参数名"
                     />
```

- **pointcut 和 pointcut-ref：**同前置通知同义；
- **method：**同前置通知同义；
- **arg-names：**同前置通知同义；
- **returning：**定义一个名字，该名字用于匹配通知实现方法的一个参数名，当目标方法执行正常返回后，将把目标方法返回值传给通知方法；returning 限定了只有目标方法返回值匹配与通知方法相应参数类型时才能执行后置返回通知，否则不执行，对于 returning 对应的通知方法参数为 Object 类型将匹配任何目标返回值。

首先在 cn.javass.spring.chapter6.service.IhelloWorldService 定义一个测试方法：

```
public boolean sayAfterReturning();
```

其次在 cn.javass.spring.chapter6.service.impl. HelloWorldService 定义实现

```
@Override
public boolean sayAfterReturning() {
    System.out.println("============after returning");
    return true;
}
```

第三在 cn.javass.spring.chapter6.aop. HelloWorldAspect 定义通知实现：

```
public void afterReturningAdvice(Object retVal) {
    System.out.println("===========after returning advice retVal:" + retVal);
}
```

最后在 chapter6/advice.xml 配置文件中接着前置通知配置的例子添加如下配置：

```
<aop:after-returning pointcut="execution(* cn.javass..*.sayAfterReturning(..))"
                     method="afterReturningAdvice"
                     arg-names="retVal"
                     returning="retVal"/>
```

测试代码 cn.javass.spring.chapter6.AopTest:

```
@Test
public void testSchemaAfterReturningAdvice() {
    System.out.println("====================================");
    ApplicationContext ctx =
          new ClassPathXmlApplicationContext("chapter6/advice.xml");
    IHelloWorldService helloworldService =
        ctx.getBean("helloWorldService", IHelloWorldService.class);
    helloworldService.sayAfterReturning();
    System.out.println("====================================");
}
```

将输入：

```
====================================
============after returning
===========after returning advice retVal:true
====================================
```

分析一下吧：

1）切入点匹配：在配置中使用“execution(* cn.javass..*.sayAfterReturning(..)) ”匹配目标方法 sayAfterReturning，该方法返回 true；

2）目标方法定义：使用 method="afterReturningAdvice"指定后置返回通知实现方法；

3）目标方法参数命名：其中使用 arg-names="retVal"指定通知实现方法参数名为“retVal”；

4）返回值命名：returning="retVal"用于将目标返回值赋值给通知实现方法参数名为“retVal”的参数上。

三、后置异常通知：在切入点选择的方法抛出异常时执行，通过<aop:aspect>标签下的<aop:after-throwing>标签声明：

```
<aop:after-throwing pointcut="切入点表达式"  pointcut-ref="切入点Bean引用"
                    method="后置异常通知实现方法名"
                    arg-names="后置异常通知实现方法参数列表参数名字"
                    throwing="将抛出的异常赋值给的通知实现方法参数名"/>
```

- **pointcut 和 pointcut-ref**：同前置通知同义；
- **method**：同前置通知同义；
- **arg-names**：同前置通知同义；
- **throwing**：定义一个名字，该名字用于匹配通知实现方法的一个参数名，当目标方法抛出异常返回后，将把目标方法抛出的异常传给通知方法；throwing 限定了只有目标方法抛出的异常匹配与通知方法相应参数异常类型时才能执行后置异常通知，否则不执行，对于 throwing 对应的通知方法参数为 Throwable 类型将匹配任何异常。

首先在 cn.javass.spring.chapter6.service.IhelloWorldService 定义一个测试方法：

```
public void sayAfterThrowing();
```

其次在 cn.javass.spring.chapter6.service.impl. HelloWorldService 定义实现

```
@Override
public void sayAfterThrowing() {
    System.out.println("============before throwing");
    throw new RuntimeException();
}
```

第三在 cn.javass.spring.chapter6.aop. HelloWorldAspect 定义通知实现：

```
public void afterThrowingAdvice(Exception exception) {
  System.out.println("===========after throwing advice exception:" + exception);
}
```

最后在 chapter6/advice.xml 配置文件中接着前置通知配置的例子添加如下配置：

```
<aop:after-throwing pointcut="execution(* cn.javass..*.sayAfterThrowing(..))"
                    method="afterThrowingAdvice"
                    arg-names="exception"
                    throwing="exception"/>
```

测试代码 cn.javass.spring.chapter6.AopTest:

```
@Test(expected = RuntimeException.class)
public void testSchemaAfterThrowingAdvice() {
    System.out.println("====================================");
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext("chapter6/advice.xml");
    IHelloWorldService helloworldService =
        ctx.getBean("helloWorldService", IHelloWorldService.class);
    helloworldService.sayAfterThrowing();
    System.out.println("====================================");
}
```

将输入：

```
====================================
============before throwing
===========after throwing advice exception:java.lang.RuntimeException
====================================
```

分析一下吧：

1）切入点匹配：在配置中使用“execution(* cn.javass..*.sayAfterThrowing(..))”匹配目标方法 sayAfterThrowing，该方法将抛出 RuntimeException 异常；

2）目标方法定义：使用 method="afterThrowingAdvice"指定后置异常通知实现方法；

3）目标方法参数命名：其中使用 arg-names="exception"指定通知实现方法参数名为“exception”；

4）异常命名：returning="exception"用于将目标方法抛出的异常赋值给通知实现方法参数名为“exception”的参数上。

四、后置最终通知：在切入点选择的方法返回时执行，不管是正常返回还是抛出异常都执行，通过<aop:aspect>标签下的<aop:after >标签声明：

```
<aop:after pointcut="切入点表达式"   pointcut-ref="切入点Bean引用"
                    method="后置最终通知实现方法名"
                    arg-names="后置最终通知实现方法参数列表参数名字"/>
```

- **pointcut 和 pointcut-ref：**同前置通知同义；
- **method：**同前置通知同义；
- **arg-names：**同前置通知同义；

首先在 cn.javass.spring.chapter6.service.IhelloWorldService 定义一个测试方法：

```
public boolean sayAfterFinally();
```

其次在 cn.javass.spring.chapter6.service.impl. HelloWorldService 定义实现

```
@Override
    public boolean sayAfterFinally() {
        System.out.println("============before finally");
        throw new RuntimeException();
}
```

第三在 cn.javass.spring.chapter6.aop. HelloWorldAspect 定义通知实现：

```
public void afterFinallyAdvice() {
        System.out.println("===========after finally advice");
}
```

最后在 chapter6/advice.xml 配置文件中接着前置通知配置的例子添加如下配置：

```
<aop:after pointcut="execution(* cn.javass..*.sayAfterFinally(..))"
          method="afterFinallyAdvice"/>
```

测试代码 cn.javass.spring.chapter6.AopTest:

```
@Test(expected = RuntimeException.class)
public void testSchemaAfterFinallyAdvice() {
    System.out.println("====================================");
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext("chapter6/advice.xml");
    IHelloWorldService helloworldService =
        ctx.getBean("helloWorldService", IHelloWorldService.class);
    helloworldService.sayAfterFinally();
    System.out.println("====================================");
}
```

将输入：

```
====================================
============before finally
===========after finally advice
====================================
```

分析一下吧：

1）切入点匹配：在配置中使用“execution(* cn.javass..*.sayAfterFinally(..))”匹配目标方法 sayAfterFinally，该方法将抛出 RuntimeException 异常；

2）目标方法定义：使用 method=" afterFinallyAdvice "指定后置最终通知实现方法。

五、环绕通知：环绕着在切入点选择的连接点处的方法所执行的通知，环绕通知非常强大，

可以决定目标方法是否执行，什么时候执行，执行时是否需要替换方法参数，执行完毕是否需要替换返回值，可通过<aop:aspect>标签下的<aop:around >标签声明：

```
<aop:around pointcut="切入点表达式"   pointcut-ref="切入点Bean引用"
                        method="后置最终通知实现方法名"
                        arg-names="后置最终通知实现方法参数列表参数名字"/>
```

- **pointcut 和 pointcut-ref**：同前置通知同义；
- **method**：同前置通知同义；
- **arg-names**：同前置通知同义；

环绕通知第一个参数必须是 org.aspectj.lang.ProceedingJoinPoint 类型，在通知实现方法内部使用 ProceedingJoinPoint 的 proceed()方法使目标方法执行，proceed 方法可以传入可选的 Object[]数组，该数组的值将被作为目标方法执行时的参数。

首先在 cn.javass.spring.chapter6.service.IhelloWorldService 定义一个测试方法：

```
public void sayAround(String param);
```

其次在 cn.javass.spring.chapter6.service.impl. HelloWorldService 定义实现

```
@Override
public void sayAround(String param) {
      System.out.println("============around param:" + param);
}
```

第三在 cn.javass.spring.chapter6.aop. HelloWorldAspect 定义通知实现：

```
public Object aroundAdvice(ProceedingJoinPoint pjp) throws Throwable {
      System.out.println("===========around before advice");
      Object retVal = pjp.proceed(new Object[] {"replace"});
      System.out.println("===========around after advice");
      return retVal;
}
```

最后在 chapter6/advice.xml 配置文件中接着前置通知配置的例子添加如下配置：

```
<aop:around pointcut="execution(* cn.javass..*.sayAround(..))"
                method="aroundAdvice"/>
```

测试代码 cn.javass.spring.chapter6.AopTest:

```
@Test
public void testSchemaAroundAdvice() {
    System.out.println("====================================");
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext("chapter6/advice.xml");
    IHelloWorldService helloworldService =
        ctx.getBean("helloWorldService", IHelloWorldService.class);
    helloworldService.sayAround("haha");
    System.out.println("====================================");
}
```

将输入：

```
====================================
==========around before advice
==========around param:replace
==========around after advice
====================================
```

分析一下吧：

1）切入点匹配：在配置中使用“execution(* cn.javass..*.sayAround(..))”匹配目标方法sayAround；

2）目标方法定义：使用 method="aroundAdvice"指定环绕通知实现方法，在该实现中，第一个方法参数为 pjp，类型为 ProceedingJoinPoint，其中“Object retVal = pjp.proceed(new Object[] {"replace"});”，用于执行目标方法，且目标方法参数被“new Object[] {"replace"}”替换，最后返回“retVal ”返回值。

3）测试：我们使用“helloworldService.sayAround("haha");”传入参数为“haha”，但最终输出为“replace”，说明参数被替换了。

6.3.4 引入

Spring 引入允许为目标对象引入新的接口，通过在< aop:aspect>标签内使用< aop:declare-parents>标签进行引入，定义方式如下：

```
<aop:declare-parents
    types-matching="AspectJ语法类型表达式"
    implement-interface=引入的接口"
    default-impl="引入接口的默认实现"
    delegate-ref="引入接口的默认实现Bean引用"/>
```

- **types-matching**：匹配需要引入接口的目标对象的 AspectJ 语法类型表达式；
- **implement-interface**：定义需要引入的接口；
- **default-impl 和 delegate-ref**：定义引入接口的默认实现，二者选一，default-impl 是接口的默认实现类全限定名，而 delegate-ref 是默认的实现的委托 Bean 名；

接下来让我们练习一下吧：

首先定义引入的接口及默认实现：

```
package cn.javass.spring.chapter6.service;
public interface IIntroductionService {
    public void induct();
}
```

```
package cn.javass.spring.chapter6.service.impl;
import cn.javass.spring.chapter6.service.IIntroductionService;
public class IntroductiondService implements IIntroductionService {
    @Override
    public void induct() {
        System.out.println("==========introduction");
    }
}
```

其次在 chapter6/advice.xml 配置文件中接着前置通知配置的例子添加如下配置：

```
<aop:declare-parents
    types-matching="cn.javass..*.IHelloWorldService+"
    implement-interface="cn.javass.spring.chapter6.service.IIntroductionService"
    default-impl="cn.javass.spring.chapter6.service.impl.IntroductiondService"/>
```

最后测试一下吧，测试代码 cn.javass.spring.chapter6.AopTest：

```
@Test
public void testSchemaIntroduction() {
    System.out.println("====================================");
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext("chapter6/advice.xml");
    IIntroductionService introductionService =
        ctx.getBean("helloWorldService", IIntroductionService.class);
    introductionService.induct();
    System.out.println("====================================");
}
```

将输入：

```
=====================================
=========introduction
=====================================
```

分析一下吧：

1）目标对象类型匹配：使用 types-matching="cn.javass..*.IHelloWorldService+"匹配 IHelloWorldService 接口的子类型，如 HelloWorldService 实现；

2）引入接口定义：通过 implement-interface 属性表示引入的接口，如“cn.javass.spring.chapter6.service.IIntroductionService”。

3）引入接口的实现：通过 default-impl 属性指定，如“cn.javass.spring.chapter6.service.impl.IntroductiondService”，也可以使用“delegate-ref”来指定实现的 Bean。

4）获取引入接口：如使用“ctx.getBean("helloWorldService", IIntroductionService.class);”可直接获取到引入的接口。

6.3.5 Advisor

Advisor表示只有一个通知和一个切入点的切面，由于Spring AOP都是基于AOP联盟的拦截器模型的环绕通知的，所以引入Advisor来支持各种通知类型（如前置通知等5种），Advisor概念来自于Spring1.2对AOP的支持，在AspectJ中没有相应的概念对应。

Advisor可以使用<aop:config>标签下的<aop:advisor>标签定义：

```
<aop:advisor pointcut="切入点表达式" pointcut-ref="切入点Bean引用"
                advice-ref="通知 API 实现引用"/>
```

- **pointcut 和 pointcut-ref：**二者选一，指定切入点表达式；
- **advice-ref：**引用通知 API 实现 Bean，如前置通知接口为 MethodBeforeAdvice；

接下来让我们看一下示例吧：

首先在 cn.javass.spring.chapter6.service.IhelloWorldService 定义一个测试方法：

```
public void sayAdvisorBefore(String param);
```

其次在 cn.javass.spring.chapter6.service.impl. HelloWorldService 定义实现

```
@Override
public void sayAdvisorBefore(String param) {
    System.out.println("============say " + param);
}
```

第三定义前置通知 API 实现：

```
package cn.javass.spring.chapter6.aop;
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;
public class BeforeAdviceImpl implements MethodBeforeAdvice {
    @Override
    public void before(Method method, Object[] args, Object target)
        throws Throwable {
        System.out.println("===========before advice");
    }
}
```

在 chapter6/advice.xml 配置文件中先添加通知实现 Bean 定义：

```
<bean id="beforeAdvice" class="cn.javass.spring.chapter6.aop.BeforeAdviceImpl"/>
```

然后在<aop:config>标签下，添加 Advisor 定义，添加时注意顺序：

```
<aop:advisor pointcut="execution(* cn.javass..*.sayAdvisorBefore(..))"
            advice-ref="beforeAdvice"/>
```

测试代码 cn.javass.spring.chapter6.AopTest:

```
@Test
public void testSchemaAdvisor() {
    System.out.println("====================================");
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext("chapter6/advice.xml");
    IHelloWorldService helloworldService =
            ctx.getBean("helloWorldService", IHelloWorldService.class);
    helloworldService.sayAdvisorBefore("haha");
    System.out.println("====================================");
}
```

将输入：

```
====================================
===========before advice
============say haha
====================================
```

在此我们只介绍了前置通知 API，其他类型的在后边章节介绍。

不推荐使用 Advisor，除了在进行事务控制的情况下，其他情况一般不推荐使用该方式，该方式属于侵入式设计，必须实现通知 API。

6.4 基于@AspectJ 的 AOP

Spring 除了支持 Schema 方式配置 AOP，还支持注解方式：使用@AspectJ 风格的切面声明。

6.4.1 启用对@AspectJ 的支持

Spring 默认不支持@AspectJ 风格的切面声明，为了支持需要使用如下配置：

```
<aop:aspectj-autoproxy/>
```

这样 Spring 就能发现@AspectJ 风格的切面并且将切面应用到目标对象。

6.4.2 声明切面

@AspectJ 风格的声明切面非常简单，使用@Aspect 注解进行声明：

```
@Aspect()
Public class Aspect{
……
}
```

然后将该切面在配置文件中声明为 Bean 后，Spring 就能自动识别并进行 AOP 方面的配置：

```
<bean id="aspect" class="……Aspect"/>
```

该切面就是一个 POJO，可以在该切面中进行切入点及通知定义，接着往下看吧。

6.4.3 声明切入点

@AspectJ 风格的命名切入点使用 org.aspectj.lang.annotation 包下的@Pointcut+方法（方法必须是返回 void 类型）实现。

```
@Pointcut(value="切入点表达式", argNames = "参数名列表")
public void pointcutName(……) {}
```

- **value**：指定切入点表达式；
- **argNames**：指定命名切入点方法参数列表参数名字，可以有多个用“，”分隔，这些参数将传递给通知方法同名的参数，同时比如切入点表达式“args(param)”将匹配参数类型为命名切入点方法同名参数指定的参数类型。
- **pointcutName**：切入点名字，可以使用该名字进行引用该切入点表达式。

```
@Pointcut(value="execution(* cn.javass..*.sayAdvisorBefore(..)) && args(param)",
            argNames = "param")
public void beforePointcut(String param) {}
```

定义了一个切入点，名字为“beforePointcut”，该切入点将匹配目标方法的第一个参数类型为通知方法实现中参数名为“param”的参数类型。

6.4.4 声明通知

@AspectJ 风格的声明通知也支持 5 种通知类型：

一、**前置通知**：使用 org.aspectj.lang.annotation 包下的@Before 注解声明；

```
@Before(value = "切入点表达式或命名切入点", argNames = "参数列表参数名")
```

- **value**：指定切入点表达式或命名切入点；
- **argNames**：与 Schema 方式配置中的同义。

接下来示例一下吧：

1、定义接口和实现，在此我们就使用 Schema 风格时的定义；

2、定义切面：

```
package cn.javass.spring.chapter6.aop;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class HelloWorldAspect2 {

}
```

3、定义切入点：

```
@Pointcut(value="execution(* cn.javass..*.sayAdvisorBefore(..)) && args(param)",
            argNames = "param")
public void beforePointcut(String param) {}
```

4、定义通知：

```
@Before(value = "beforePointcut(param)", argNames = "param")
public void beforeAdvice(String param) {
    System.out.println("==========before advice param:" + param);
}
```

5、在 chapter6/advice2.xml 配置文件中进行如下配置：

```
<?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"
          xsi:schemaLocation="
              http://www.springframework.org/schema/beans
              http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
              http://www.springframework.org/schema/aop
              http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

   <aop:aspectj-autoproxy/>
   <bean id="helloWorldService"
          class="cn.javass.spring.chapter6.service.impl.HelloWorldService"/>

   <bean id="aspect"
          class="cn.javass.spring.chapter6.aop.HelloWorldAspect2"/>

</beans>
```

6、测试代码 cn.javass.spring.chapter6.AopTest:

```
@Test
public void testAnnotationBeforeAdvice() {
    System.out.println("====================================");
    ApplicationContext ctx =
    new ClassPathXmlApplicationContext("chapter6/advice2.xml");
    IHelloWorldService helloworldService =
    ctx.getBean("helloWorldService", IHelloWorldService.class);
    helloworldService.sayBefore("before");
    System.out.println("====================================");
}
```

将输出：

```
==========================================
===========before advice param:before
============say before
==========================================
```

切面、切入点、通知全部使用注解完成：

1）使用@Aspect 将 POJO 声明为切面；

2）使用@Pointcut 进行命名切入点声明，同时指定目标方法第一个参数类型必须是 java.lang.String，对于其他匹配的方法但参数类型不一致的将也是不匹配的，通过 argNames = "param"指定了将把该匹配的目标方法参数传递给通知同名的参数上；

3）使用@Before 进行前置通知声明，其中 value 用于定义切入点表达式或引用命名切入点；

4）配置文件需要使用<aop:aspectj-autoproxy/>来开启注解风格的@AspectJ 支持；

5）需要将切面注册为Bean，如“aspect”Bean；

6）测试代码完全一样。

二、后置返回通知：使用 org.aspectj.lang.annotation 包下的@AfterReturning 注解声明；

```
@AfterReturning(
    value="切入点表达式或命名切入点",
    pointcut="切入点表达式或命名切入点",
    argNames="参数列表参数名",
    returning="返回值对应参数名")
```

- **value：**指定切入点表达式或命名切入点；
- **pointcut：**同样是指定切入点表达式或命名切入点，如果指定了将覆盖 value 属性指定的，pointcut 具有高优先级；
- **argNames：**与 Schema 方式配置中的同义；
- **returning：**与 Schema 方式配置中的同义。

```
@AfterReturning(
    value="execution(* cn.javass..*.sayBefore(..))",
    pointcut="execution(* cn.javass..*.sayAfterReturning(..))",
    argNames="retVal", returning="retVal")
public void afterReturningAdvice(Object retVal) {
    System.out.println("===========after returning advice retVal:" + retVal);
}
```

其中测试代码与 Schema 方式几乎一样，在此就不演示了，如果需要请参考 AopTest.java 中的 testAnnotationAfterReturningAdvice 测试方法。

三、后置异常通知：使用 org.aspectj.lang.annotation 包下的@AfterThrowing 注解声明；

```
@AfterThrowing (
    value="切入点表达式或命名切入点",
    pointcut="切入点表达式或命名切入点",
    argNames="参数列表参数名",
    throwing="异常对应参数名")
```

- **value：**指定切入点表达式或命名切入点；
- **pointcut：**同样是指定切入点表达式或命名切入点，如果指定了将覆盖 value 属性指定的，pointcut 具有高优先级；
- **argNames：**与 Schema 方式配置中的同义；
- **throwing：**与 Schema 方式配置中的同义。

```
@AfterThrowing(
    value="execution(* cn.javass..*.sayAfterThrowing(..))",
    argNames="exception", throwing="exception")
public void afterThrowingAdvice(Exception exception) {
   System.out.println("============after throwing advice exception:" + exception);
}
```

其中测试代码与 Schema 方式几乎一样，在此就不演示了，如果需要请参考 AopTest.java 中的 testAnnotationAfterThrowingAdvice 测试方法。

四、后置最终通知：使用 org.aspectj.lang.annotation 包下的@After 注解声明；

```
@After (
    value="切入点表达式或命名切入点",
    argNames="参数列表参数名")
```

- **value：**指定切入点表达式或命名切入点；
- **argNames：**与 Schema 方式配置中的同义；

```
@After(value="execution(* cn.javass..*.sayAfterFinally(..))")
public void afterFinallyAdvice() {
    System.out.println("============after finally advice");
}
```

其中测试代码与 Schema 方式几乎一样，在此就不演示了，如果需要请参考 AopTest.java 中的 testAnnotationAfterFinallyAdvice 测试方法。

五、环绕通知：使用 org.aspectj.lang.annotation 包下的@Around 注解声明；

```
@Around (
    value="切入点表达式或命名切入点",
    argNames="参数列表参数名")
```

- **value：**指定切入点表达式或命名切入点；
- **argNames：**与 Schema 方式配置中的同义；

```
@Around(value="execution(* cn.javass..*.sayAround(..))")
public Object aroundAdvice(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("============around before advice");
    Object retVal = pjp.proceed(new Object[] {"replace"});
    System.out.println("============around after advice");
    return retVal;
}
```

其中测试代码与 Schema 方式几乎一样，在此就不演示了，如果需要请参考 AopTest.java 中的 annotationAroundAdviceTest 测试方法。

6.4.5 引入

@AspectJ 风格的引入声明在切面中使用 org.aspectj.lang.annotation 包下的@DeclareParents 声明：

```
@DeclareParents(
    value=" AspectJ 语法类型表达式",
    defaultImpl=引入接口的默认实现类)
private Interface interface;
```

- **value：**匹配需要引入接口的目标对象的 AspectJ 语法类型表达式；与 Schema 方式中的 types-matching 属性同义；
- private Interface interface：指定需要引入的接口；
- defaultImpl：指定引入接口的默认实现类，没有与 Schema 方式中的 delegate-ref 属性同义的定义方式；

```
@DeclareParents(
    value="cn.javass..*.IHelloWorldService+",
    defaultImpl=cn.javass.spring.chapter6.service.impl.IntroductiondService.class)
private IIntroductionService introductionService;
```

其中测试代码与 Schema 方式几乎一样，在此就不演示了，如果需要请参考 AopTest.java 中的 testAnnotationIntroduction 测试方法。

6.5 AspectJ 切入点语法详解

6.5.1 Spring AOP 支持的 AspectJ 切入点指示符

切入点指示符用来指示切入点表达式目的，，在 Spring AOP 中目前只有执行方法这一个连接点，Spring AOP 支持的 AspectJ 切入点指示符如下：

- **execution：**用于匹配方法执行的连接点；
- **within：**用于匹配指定类型内的方法执行；
- **this：**用于匹配当前 AOP 代理对象类型的执行方法；注意是 AOP 代理对象的类型匹配，这样就可能包括引入接口也类型匹配；
- **target：**用于匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；
- **args：**用于匹配当前执行的方法传入的参数为指定类型的执行方法；
- **@within：**用于匹配所以持有指定注解类型内的方法；
- **@target：**用于匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；
- **@args：**用于匹配当前执行的方法传入的参数持有指定注解的执行；
- **@annotation：**用于匹配当前执行方法持有指定注解的方法；
- **bean：**Spring AOP 扩展的，AspectJ 没有对于指示符，用于匹配特定名称的 Bean 对象的执行方法；
- **reference pointcut：**表示引用其他命名切入点，只有@ApectJ 风格支持，Schema 风格不支持。

AspectJ 切入点支持的切入点指示符还有：call、get、set、preinitialization、staticinitialization、initialization、handler、adviceexecution、withincode、cflow、cflowbelow、if、@this、@withincode；但 Spring AOP

目前不支持这些指示符，使用这些指示符将抛出 IllegalArgumentException 异常。这些指示符 Spring AOP 可能会在以后进行扩展。

6.5.1 命名及匿名切入点

命名切入点可以被其他切入点引用，而匿名切入点是不可以的。

只有@AspectJ 支持命名切入点，而 Schema 风格不支持命名切入点。

如下所示，@AspectJ 使用如下方式引用命名切入点：

```
@Pointcut(
   value="execution(* cn.javass..*.sayBefore(java.lang.String)) && args(param)",
   argNames = "param")
public void beforePointcut(String param) {}
                          ↓ 引用命名切入点
@Before(value = "beforePointcut(param)", argNames = "param")
public void beforeAdvice(String param) {
      System.out.println("===========before advice param:" + param);
}
```

6.5.2 ；类型匹配语法

首先让我们来了解下 AspectJ 类型匹配的通配符：

- *：匹配任何数量字符；
- ..：匹配任何数量字符的重复，如在类型模式中匹配任何数量子包；而在方法参数模式中匹配任何数量参数。
- +：匹配指定类型的子类型；仅能作为后缀放在类型模式后边。

java.lang.String	匹配 String 类型；
java.*.String	匹配 java 包下的任何“一级子包”下的 String 类型； 如匹配 java.lang.String，但不匹配 java.lang.ss.String
java..*	匹配 java 包及任何子包下的任何类型; 如匹配 java.lang.String、java.lang.annotation.Annotation
java.lang.*ing	匹配任何 java.lang 包下的以 ing 结尾的类型；
java.lang.Number+	匹配 java.lang 包下的任何 Number 的自类型； 如匹配 java.lang.Integer，也匹配 java.math.BigInteger

接下来再看一下具体的匹配表达式类型吧：

- **匹配类型**：使用如下方式匹配

```
注解？ 类的全限定名字
```

- **注解：**可选，类型上持有的注解，如@Deprecated；
- **类的全限定名：**必填，可以是任何类全限定名。

■ **匹配方法执行：**使用如下方式匹配：

注解？ 修饰符? 返回值类型 类型声明?方法名(参数列表) 异常列表？

- **注解：**可选，方法上持有的注解，如@Deprecated；
- **修饰符：**可选，如 public、protected；
- **返回值类型：**必填，可以是任何类型模式；“*”表示所有类型；
- **类型声明：**可选，可以是任何类型模式；
- **方法名：**必填，可以使用“*”进行模式匹配；
- **参数列表：**“()”表示方法没有任何参数；“(..)”表示匹配接受任意个参数的方法，“(..,java.lang.String)”表示匹配接受 java.lang.String 类型的参数结束，且其前边可以接受有任意个参数的方法；“(java.lang.String,..)” 表示匹配接受 java.lang.String 类型的参数开始，且其后边可以接受任意个参数的方法；“(*,java.lang.String)” 表示匹配接受 java.lang.String 类型的参数结束，且其前边接受有一个任意类型参数的方法；
- **异常列表：**可选，以“throws 异常全限定名列表”声明，异常全限定名列表如有多个以“，”分割，如 throws java.lang.IllegalArgumentException, java.lang.ArrayIndexOutOfBoundsException。

■ **匹配 Bean 名称：**可以使用 Bean 的 id 或 name 进行匹配，并且可使用通配符“*”；

6.5.3 组合切入点表达式

AspectJ 使用 且（&&）、或（||）、非（！）来组合切入点表达式。

在 Schema 风格下，由于在 XML 中使用“&&”需要使用转义字符“&&”来代替之，所以很不方便，因此 Spring ASP 提供了 and、or、not 来代替&&、||、！。

6.5.3 切入点使用示例

一、**execution：**使用“execution(方法表达式)”匹配方法执行；

模式	描述
public * *(..)	任何公共方法的执行
* cn.javass..IPointcutService.*()	cn.javass 包及所有子包下 IPointcutService 接口中的任何无参方法
* cn.javass..*.*(..)	cn.javass 包及所有子包下任何类的任何方法

* cn.javass..IPointcutService.*(*)	cn.javass 包及所有子包下 IPointcutService 接口的任何只有一个参数方法
* (!cn.javass..IPointcutService+).*(..)	非“cn.javass 包及所有子包下 IPointcutService 接口及子类型”的任何方法
* cn.javass..IPointcutService+.*()	cn.javass 包及所有子包下 IPointcutService 接口及子类型的的任何无参方法
* cn.javass..IPointcut*.test*(java.util.Date)	cn.javass 包及所有子包下 IPointcut 前缀类型的的以 test 开头的只有一个参数类型为 java.util.Date 的方法，注意该匹配是根据方法签名的参数类型进行匹配的，而不是根据执行时传入的参数类型决定的 如定义方法：public void test(Object obj);即使执行时传入 java.util.Date，也不会匹配的；
* cn.javass..IPointcut*.test*(..) throws IllegalArgumentException, ArrayIndexOutOfBoundsException	cn.javass 包及所有子包下 IPointcut 前缀类型的的任何方法，且抛出 IllegalArgumentException 和 ArrayIndexOutOfBoundsException 异常
* (cn.javass..IPointcutService+ && java.io.Serializable+).*(..)	任何实现了 cn.javass 包及所有子包下 IPointcutService 接口和 java.io.Serializable 接口的类型的任何方法
@java.lang.Deprecated * *(..)	任何持有@java.lang.Deprecated 注解的方法
@java.lang.Deprecated @cn.javass..Secure * *(..)	任何持有@java.lang.Deprecated 和@cn.javass..Secure 注解的方法
@(java.lang.Deprecated || cn.javass..Secure) * *(..)	任何持有@java.lang.Deprecated 或@ cn.javass..Secure 注解的方法
(@cn.javass..Secure *) *(..)	任何返回值类型持有@cn.javass..Secure 的方法
* (@cn.javass..Secure *).*(..)	任何定义方法的类型持有@cn.javass..Secure 的方法
* *(@cn.javass..Secure (*) , @cn.javass..Secure (*))	任何签名带有两个参数的方法，且这个两个参数都被@ Secure 标记了， 如 public void test(@Secure String str1, @Secure String str1);
* *((@ cn.javass..Secure *))或 * *(@ cn.javass..Secure *)	任何带有一个参数的方法，且该参数类型持有@ cn.javass..Secure； 如 public void test(Model model);且 Model 类上持有@Secure 注解
* *(@cn.javass..Secure (@cn.javass..Secure *) , @ cn.javass..Secure (@cn.javass..Secure *))	任何带有两个参数的方法，且这两个参数都被@ cn.javass..Secure 标记了；且这两个参数的类型上都持有@ cn.javass..Secure；

* *(java.util.Map<cn.javass..Model, cn.javass..Model> , ..)	任何带有一个 java.util.Map 参数的方法，且该参数类型是以< cn.javass..Model, cn.javass..Model >为泛型参数；注意只匹配第一个参数为 java.util.Map,不包括子类型； 如 public void test(HashMap<Model, Model> map, String str);将不匹配，必须使用“* *(java.util.HashMap<cn.javass..Model,cn.javass..Model> , ..)”进行匹配； 而 public void test(Map map, int i);也将不匹配，因为泛型参数不匹配
* *(java.util.Collection<@cn.javass..Secure *>)	任何带有一个参数（类型为 java.util.Collection）的方法，且该参数类型是有一个泛型参数，该泛型参数类型上持有@cn.javass..Secure 注解； 如 public void test(Collection<Model> collection);Model 类型上持有@cn.javass..Secure
* *(java.util.Set<? extends HashMap>)	任何带有一个参数的方法，且传入的参数类型是有一个泛型参数，该泛型参数类型继承与 HashMap； **Spring AOP 目前测试不能正常工作**
* *(java.util.List<? super HashMap>)	任何带有一个参数的方法，且传入的参数类型是有一个泛型参数，该泛型参数类型是 HashMap 的基类型； 如 public voi test(Map map)； **Spring AOP 目前测试不能正常工作**
* *(*<@cn.javass..Secure *>)	任何带有一个参数的方法，且该参数类型是有一个泛型参数，该泛型参数类型上持有@cn.javass..Secure 注解； **Spring AOP 目前测试不能正常工作**

二、**within：使用“within(类型表达式)”**匹配指定类型内的方法执行；

模式	描述
within(cn.javass..*)	cn.javass 包及子包下的任何方法执行
within(cn.javass..IPointcutService+)	cn.javass 包或所有子包下 IPointcutService 类型及子类型的任何方法
within(@cn.javass..Secure *)	持有 cn.javass..Secure 注解的任何类型的任何方法 必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

三、**this：**使用“this(类型全限定名)”匹配当前 AOP 代理对象类型的执行方法；注意是 AOP 代理对象的类型匹配，这样就可能包括引入接口方法也可以匹配；注意 this 中使用的表达式必须是类型全限定名，不支持通配符；

模式	描述
this(cn.javass.spring.chapter6.service.IPointcutService)	当前 AOP 对象实现了 IPointcutService 接口的任何方法
this(cn.javass.spring.chapter6.service.IIntroductionService)	当前 AOP 对象实现了 IIntroductionService 接口的任何方法 也可能是引入接口

四、**target：**使用“target(类型全限定名)”匹配当前目标对象类型的执行方法；注意是目标对象的类型匹配，这样就不包括引入接口也类型匹配；注意 target 中使用的表达式必须是类型全限定名，不支持通配符；

模式	描述
target(cn.javass.spring.chapter6.service.IPointcutService)	当前目标对象（非 AOP 对象）实现了 IPointcutService 接口的任何方法
target(cn.javass.spring.chapter6.service.IIntroductionService)	当前目标对象（非 AOP 对象） 实现了 IIntroductionService 接口的任何方法 不可能是引入接口

五、**args：**使用“args(参数类型列表)”匹配当前执行的方法传入的参数为指定类型的执行方法；注意是匹配传入的参数类型，不是匹配方法签名的参数类型；参数类型列表中的参数必须是类型全限定名，通配符不支持；args 属于动态切入点，这种切入点开销非常大，非特殊情况最好不要使用；

模式	描述
args (java.io.Serializable,..)	任何一个以接受“传入参数类型为 java.io.Serializable” 开头，且其后可跟任意个任意类型的参数的方法执行，args 指定的参数类型是在运行时动态匹配的

六、@within：使用“@within(注解类型)”匹配所以持有指定注解类型内的方法；注解类型也必须是全限定类型名；

模式	描述
@within cn.javass.spring.chapter6.Secure)	任何目标对象对应的类型持有 Secure 注解的类方法； 必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

七、@target：使用“@target(注解类型)”匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；注解类型也必须是全限定类型名；

模式	描述
@target (cn.javass.spring.chapter6.Secure)	任何目标对象持有 Secure 注解的类方法； 必须是在目标对象上声明这个注解，在接口上声明的对它不起作用

八、@args：使用“@args(注解列表)”匹配当前执行的方法传入的参数持有指定注解的执行；注解类型也必须是全限定类型名；

模式	描述
@args (cn.javass.spring.chapter6.Secure)	任何一个只接受一个参数的方法，且方法运行时传入的参数持有注解 cn.javass.spring.chapter6.Secure；动态切入点，类似于 arg 指示符；

九、@annotation：使用“@annotation(注解类型)”匹配当前执行方法持有指定注解的方法；注解类型也必须是全限定类型名；

模式	描述
@annotation(cn.javass.spring.chapter6.Secure)	当前执行方法上持有注解 cn.javass.spring.chapter6.Secure 将被匹配

十、bean：使用“bean(Bean id 或名字通配符)”匹配特定名称的 Bean 对象的执行方法；Spring ASP 扩展的，在 AspectJ 中无相应概念；

模式	描述
bean(*Service)	匹配所有以 Service 命名（id 或 name）结尾的 Bean

十一、reference pointcut：表示引用其他命名切入点，只有@ApectJ 风格支持，Schema 风格不支持，如下所示：

```
@Pointcut(value="bean(*Service)")
private void pointcut1(){}                                    //命名切入点1
@Pointcut(value="@args(cn.javass.spring.chapter6.Secure)")    //命名切入点2
private void pointcut2(){}

@Before(value = "pointcut1() && pointcut2()")                 //引用命名切入点
public void referencePointcutTest1(JoinPoint jp) {
    dump("pointcut1() && pointcut2()", jp);
}
```

比如我们定义如下切面：

```
package cn.javass.spring.chapter6.aop;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class ReferencePointcutAspect {
    @Pointcut(value="execution(* *())")
    public void pointcut() {}
}
```

可以通过如下方式引用：

```
@Before(value = "cn.javass.spring.chapter6.aop.ReferencePointcutAspect.pointcut()")
public void referencePointcutTest2(JoinPoint jp) {}
```

除了可以在@AspectJ 风格的切面内引用外，也可以在 Schema 风格的切面定义内引用，引用方式与@AspectJ 完全一样。

到此我们切入点表达式语法示例就介绍完了，我们这些示例几乎包含了日常开发中的所有情况，但当然还有更复杂的语法等等，如果以上介绍的不能满足您的需要，请参考 AspectJ 文档。

由于测试代码相当长，所以为了节约篇幅本示例代码在 cn.javass.spring.chapter6.PointcutTest 文件中，需要时请参考该文件。

6.6 通知参数

前边章节已经介绍了声明通知，但如果想获取被被通知方法参数并传递给通知方法，该如何实现呢？接下来我们将介绍两种获取通知参数的方式。

- **使用 JoinPoint 获取**：Spring AOP 提供使用 org.aspectj.lang.JoinPoint 类型获取连接点数据，任何通知方法的第一个参数都可以是 JoinPoint(环绕通知是 ProceedingJoinPoint，JoinPoint 子类)，当然第一个参数位置也可以是 JoinPoint.StaticPart 类型，这个只返回连接点的静态部分。

  1) JoinPoint：提供访问当前被通知方法的目标对象、代理对象、方法参数等数据：

```
package org.aspectj.lang;
import org.aspectj.lang.reflect.SourceLocation;
public interface JoinPoint {
    String toString();          //连接点所在位置的相关信息
    String toShortString();     //连接点所在位置的简短相关信息
    String toLongString();      //连接点所在位置的全部相关信息
    Object getThis();           //返回 AOP 代理对象
    Object getTarget();         //返回目标对象
    Object[] getArgs();         //返回被通知方法参数列表
    Signature getSignature();   //返回当前连接点签名
    SourceLocation getSourceLocation();//返回连接点方法所在类文件中的位置
    String getKind();           //连接点类型
    StaticPart getStaticPart(); //返回连接点静态部分
}
```

2）ProceedingJoinPoint：用于环绕通知，使用 proceed()方法来执行目标方法：

```
public interface ProceedingJoinPoint extends JoinPoint {
    public Object proceed() throws Throwable;
    public Object proceed(Object[] args) throws Throwable;
}
```

3) JoinPoint.StaticPart：提供访问连接点的静态部分，如被通知方法签名、连接点类型等：

```
public interface StaticPart {
    Signature getSignature();     //返回当前连接点签名
    String getKind();             //连接点类型
    int getId();                  //唯一标识
    String toString();            //连接点所在位置的相关信息
    String toShortString();       //连接点所在位置的简短相关信息
    String toLongString();        //连接点所在位置的全部相关信息
}
```

使用如下方式在通知方法上声明，必须是在第一个参数，然后使用 jp.getArgs()就能获取到被通知方法参数：

```
@Before(value="execution(* sayBefore(*))")
public void before(JoinPoint jp) {}

@Before(value="execution(* sayBefore(*))")
public void before(JoinPoint.StaticPart jp) {}
```

- **自动获取：**通过切入点表达式可以将相应的参数自动传递给通知方法，例如前边章节讲过的返回值和异常是如何传递给通知方法的。

在 Spring AOP 中，除了 execution 和 bean 指示符不能传递参数给通知方法，其他指示符都可以将匹配的相应参数或对象自动传递给通知方法。

```
@Before(value="execution(* test(*)) && args(param)", argNames="param")
public void before1(String param) {
    System.out.println("===param:" + param);
}
```

切入点表达式 execution(* test(*)) && args(param) ：

1）首先 execution(* test(*))匹配任何方法名为 test，且有一个任何类型的参数；

2）args(param)将首先查找通知方法上同名的参数，并在方法执行时（运行时）匹配传入的参数是使用该同名参数类型，即 java.lang.String；如果匹配将把该被通知参数传递给通知方法上同名参数。

其他指示符（除了 execution 和 bean 指示符）都可以使用这种方式进行参数绑定。

在此有一个问题，即前边提到的类似于【3.1.2 构造器注入】中的参数名注入限制：**在 class 文件中没生成变量调试信息是获取不到方法参数名字的。**

所以我们可以使用策略来确定参数名：

1. 如果我们通过“argNames”属性指定了参数名，那么就是要我们指定的；

```
@Before(value=" args(param)", argNames="param") //明确指定了
public void before1(String param) {
    System.out.println("===param:" + param);
}
```

2. 如果第一个参数类型是 JoinPoint、ProceedingJoinPoint 或 JoinPoint.StaticPart 类型，应该从“argNames”属性省略掉该参数名（可选，写上也对），这些类型对象会自动传入的，但必须作为第一个参数；

```
@Before(value=" args(param)", argNames="param") //明确指定了
public void before1(JoinPoint jp, String param) {
    System.out.println("===param:" + param);
}
```

3. 如果“**class 文件中含有变量调试信息**”将使用这些方法签名中的参数名来确定参数名；

```
@Before(value=" args(param)") //不需要argNames了
public void before1(JoinPoint jp, String param) {
    System.out.println("===param:" + param);
}
```

4. 如果没有“**class 文件中含有变量调试信息**”，将尝试自己的参数匹配算法，如果发现参数绑定有二义性将抛出 AmbiguousBindingException 异常；对于只有一个绑定变量的切入点表达式，而通知方法只接受一个参数，说明绑定参数是明确的，从而能配对成功。

```
@Before(value=" args(param)")
public void before1(JoinPoint jp, String param) {
      System.out.println("===param:" + param);
}
```

5. 以上策略失败将抛出 IllegalArgumentException。

接下来让我们示例一下组合情况吧：

```
@Before(args(param) && target(bean) && @annotation(secure)",
          argNames="jp,param,bean,secure")
public void before5(JoinPoint jp, String param,
  IPointcutService pointcutService, Secure secure) {
……
}
```

该示例的执行步骤如图 6-5 所示。

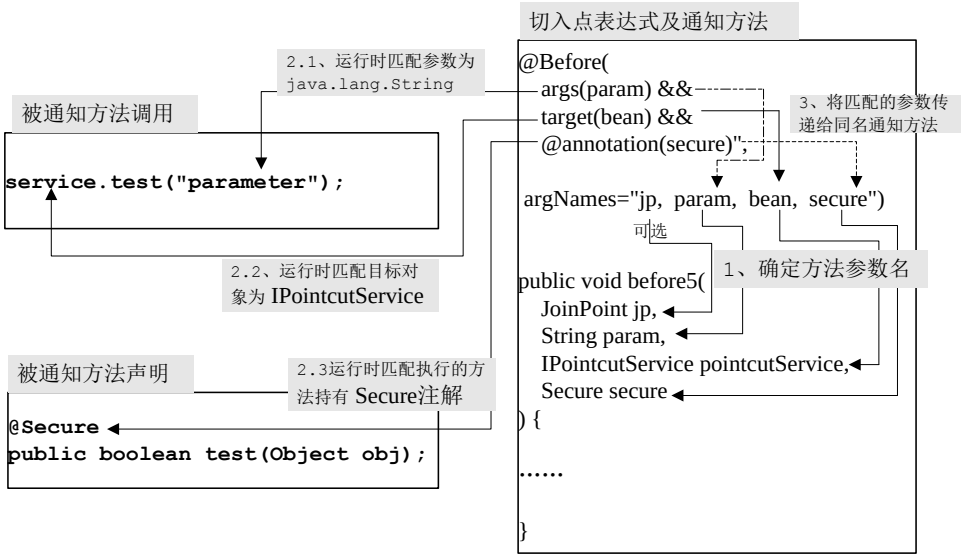


图 6-5 参数自动获取流程

除了上边介绍的普通方式，也可以对使用命名切入点自动获取参数：

```
@Pointcut(value="args(param)", argNames="param")
private void pointcut1(String param){}
@Pointcut(value="@annotation(secure)", argNames="secure")
private void pointcut2(Secure secure){}

@Before(value = "pointcut1(param) && pointcut2(secure)",
        argNames="param, secure")
public void before6(JoinPoint jp, String param, Secure secure) {
……
}
```

自此给通知传递参数已经介绍完了，示例代码在 cn.javass.spring.chapter6.ParameterTest 文件中。

6.7 通知顺序

如果我们有多个通知想要在同一连接点执行，那执行顺序如何确定呢？Spring AOP 使用 AspectJ 的优先级规则来确定通知执行顺序。总共有两种情况：同一切面中通知执行顺序、不同切面中的通知执行顺序。

首先让我们看下

1） 同一切面中通知执行顺序：如图 6-6 所示。

```
1、前置通知/环绕通知（proceed方法执行之前）---执行顺序不确定
2、被通知方法
3、后置通知/环绕通知（proceed之后）          ---执行顺序不确定
```

图 6-6 同一切面中的通知执行顺序

而如果在同一切面中定义两个相同类型通知（如同是前置通知或环绕通知（proceed 之前））并在同一连接点执行时，其执行顺序是未知的，如果确实需要指定执行顺序需要将通知重构到两个切面，然后定义切面的执行顺序。

错误“Advice precedence circularity error”：说明 AspectJ 无法决定通知的执行顺序，只要将通知方法分类并按照顺序排列即可解决。

2）**不同切面中的通知执行顺序：**当定义在不同切面的相同类型的通知需要在同一个连接点执行，如果没指定切面的执行顺序，这两个通知的执行顺序将是未知的。

如果需要他们顺序执行，可以通过指定切面的优先级来控制通知的执行顺序。

Spring 中可以通过在切面实现类上实现 org.springframework.core.Ordered 接口或使用 Order 注解来指定切面优先级。在多个切面中，Ordered.getValue()方法返回值（或者注解值）较小值的那个切面拥有较高优先级，如图 6-7 所示。

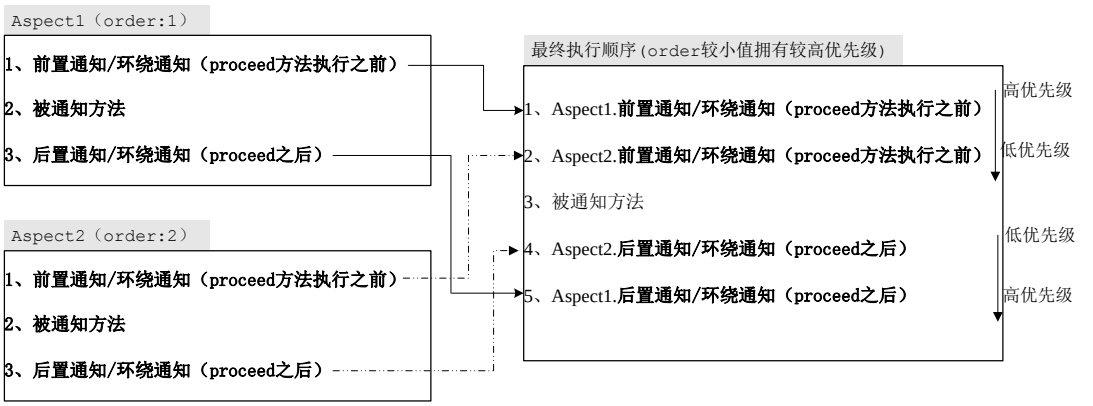


图 6-7 两个切面指定了优先级

对于@AspectJ 风格和注解风格可分别用以下形式指定优先级：

```
@Aspect
@Order(2)
public class OrderAspect2 {
}
```

```
<aop:aspect ref="aspect1" order="1">
……
</aop:aspect>
```

在此我们不推荐使用实现 Ordered 接口方法，所以没介绍，示例代码在 cn.javass.spring.chapter6. OrderAopTest 文件中。

6.8 切面实例化模型

所谓切面实例化模型指何时实例化切面。

Spring AOP 支持 AspectJ 的 singleton、perthis、pertarget 实例化模型（目前不支持 percflow、percflowbelow 和 pertypewithin）。

- **singleton：**即切面只会有一个实例；
- **perthis：**每个切入点表达式匹配的连接点对应的 AOP 对象都会创建一个新切面实例；
- **pertarget：**每个切入点表达式匹配的连接点对应的目标对象都会创建一个新的切

面实例；

默认是 singleton 实例化模型，Schema 风格只支持 singleton 实例化模型，而@AspectJ 风格支持这三种实例化模型。

- **singleton：**使用@Aspect()指定，即默认就是单例实例化模式，在此就不演示示例了。
- **perthis：**每个切入点表达式匹配的连接点对应的 AOP 对象都会创建一个新的切面实例，使用@Aspect("perthis(切入点表达式)")指定切入点表达式；

  如 @Aspect("perthis(this(cn.javass.spring.chapter6.service.IIntroductionService))") 将对每个匹配“this(cn.javass.spring.chapter6.service.IIntroductionService)”切入点表达式的 AOP 代理对象创建一个切面实例，注意“IIntroductionService”可能是引入接口。
- **pertarget：**每个切入点表达式匹配的连接点对应的目标对象都会创建一个新的切面实例，使用@Aspect("pertarget(切入点表达式)")指定切入点表达式；

  如@Aspect("pertarget(target(cn.javass.spring.chapter6. service.IPointcutService))")将对每个匹配“target(cn.javass.spring.chapter6.service. IPointcutService)”切入点表达式的目标对象创建一个切面，注意“IPointcutService”不可能是引入接口。

在进行切面定义时必须将切面 scope 定义为“prototype”，如“<bean class="……Aspect" scope="prototype"/>”，否则不能为每个匹配的连接点的目标对象或 AOP 代理对象创建一个切面。

示例请参考 cn.javass.spring.chapter6. InstanceModelTest。

6.9 代理机制

Spring AOP 通过代理模式实现，目前支持两种代理：JDK 动态代理、CGLIB 代理来创建 AOP 代理，Spring 建议优先使用 JDK 动态代理。

- **JDK 动态代理：**使用 java.lang.reflect.Proxy 动态代理实现，即提取目标对象的接口，然后对接口创建 AOP 代理。
- **CGLIB 代理：**CGLIB 代理不仅能进行接口代理，也能进行类代理，CGLIB 代理需要注意以下问题：
  - 不能通知 final 方法，因为 final 方法不能被覆盖（CGLIB 通过生成子类来创建代理）。
  - 会产生两次构造器调用，第一次是目标类的构造器调用，第二次是 CGLIB 生成的代理类的构造器调用。如果需要 CGLIB 代理方法，请确保两次构造器调用不影响应用。

Spring AOP 默认首先使用 JDK 动态代理来代理目标对象，如果目标对象没有实现任何接口将使用 CGLIB 代理，如果需要强制使用 CGLIB 代理，请使用如下方式指定：

对于 Schema 风格配置切面使用如下方式来指定使用 CGLIB 代理：

```
<aop:config proxy-target-class="true">
</aop:config>
```

而如果使用@AspectJ 风格使用如下方式来指定使用 CGLIB 代理：

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

第七章　对 JDBC 的支持

第八章　对 ORM 的支持

第九章　Spring 的事务

第十章　集成其它 Web 框架

第十一章　SSH 集成开发

第十二章　零配置

第十三章　测试

第十四章　IoC 扩展

第十五章　Spring 的 AOP 的接口

第十六章　深入理解 Spring 的 AOP

第十七章　校验和数据绑定

第十八章　Spring 的 MVC

第十九章　综合示例

第二十章　集成视图技术

第二十一章　远程访问

第二十二章　集成 EJB

第二十三章　集成 Web 服务

第二十四章　集成 JMS

第二十五章　集成 JavaMail

第二十六章　集成 JMX

第二十七章　定时调度和线程池

第二十八章　集成动态语言