Spring Framework 5.0 之核心容器及 AOP 内部资料

http://www.open-v.com 作品

2017.03.01

Spring Framework 5.0 系列作品写作背景

Spring Framework 已经走过了十多年的发展历程，而如今它已经进入到 5.0 时代。不同于其他框架，尽管发展多年，Spring Framework 及基于它的生态系统仍然占据着企业级 Java 的主流位置，可见其生命力多强。

为了系统性跟进 Spring Framework 5.0 的发展，我们特别整理了如下几方面的资料。随着写作工作的推进，内部资料涉及的广度和深度都会逐渐增加。

《Spring Framework 5.0 之核心容器及 AOP 内部资料》

《Spring Framework 5.0 之持久化技术及事务管理内部资料》

《Spring Framework 5.0 之 Web MVC 内部资料》

《Spring Framework 5.0 之测试内部资料》

《Spring Framework 5.0 之其他专题内部资料》

本系列作品的代码存放在 https://github.com/openvcube/springframework50 位置。

另外，如有任何问题咨询，非常欢迎加入我们的 QQ 群（106813165），加入密码是“www.open-v.com”。

修订记录

版本日期	备注
2014-5-12	规划目录，并着手针对 Spring Framework 4.0 版本写作。
2014-5-21	Spring Framework 升级到 4.0.5。
2014-7-7	第一版基本完成。
2014-7-9	Spring Framework 升级到 4.0.6。
2014-9-20	Spring Framework 升级到 4.1.0。
2014-10-8	若干细节问题修正。
2016-1-31	Spring Framework 升级到 4.2.4。
2017-03-01	Spring Framework 准备升级到 5.0 M4。

目录

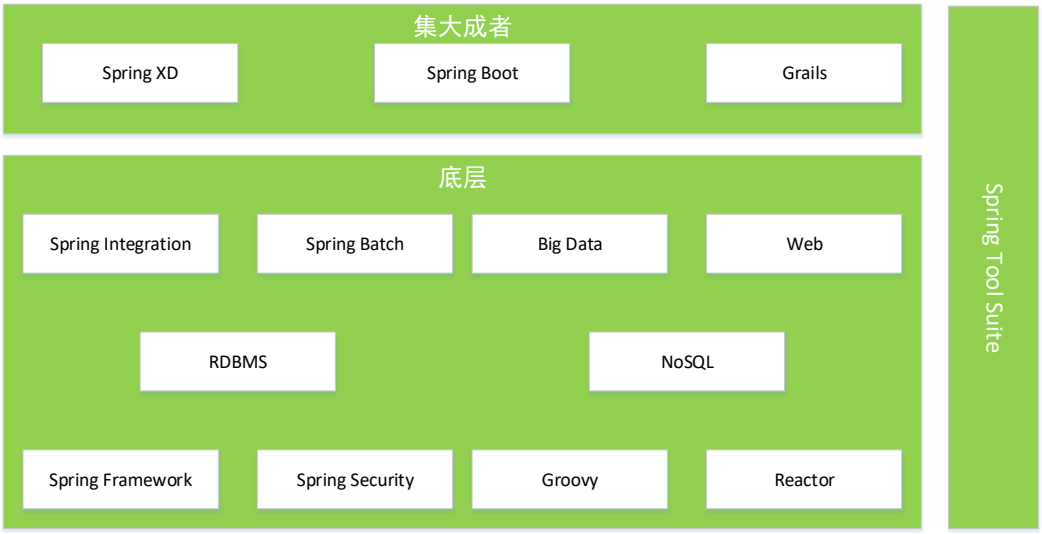
1 Spring Framework 5.0 介绍

Spring Framework 在 Java EE 领域占据着不可或缺的位置。Java EE 7 同 Spring Framework 5.0 并肩战斗，并将企业级 Java 推向另一个高度。

1.1 Spring Framework 5.0 概述

1.1.1 Spring 生态系统概述

经过十几年的发展，不只是 Spring Framework 框架本身，整个社区已经形成了以 Spring Framework 为基础的 Spring 生态系统，这一生态系统非常庞大。下图展示了主要要素。



图表 1 Spring 生态系统

从图中可以看出，Spring 生态系统针对企业应用研发、商业智能实施（包括 Hadoop、Big Data 等）提供了一站式解决方案。比如，Spring Security 用于解决安全问题，它已经成为了事实上的安全框架标准；Spring Web Flow 页面流框架；Spring Data 针对大数据提供一流集成支持（包括 Hadoop、Hive、HBase 等）；Spring Batch 用于 ETL 领域；Spring Boot 倡导微服务架构；等。

因为 Spring 生态系统很好地贯彻了开源的思想和精髓，比如各项目文档丰富、

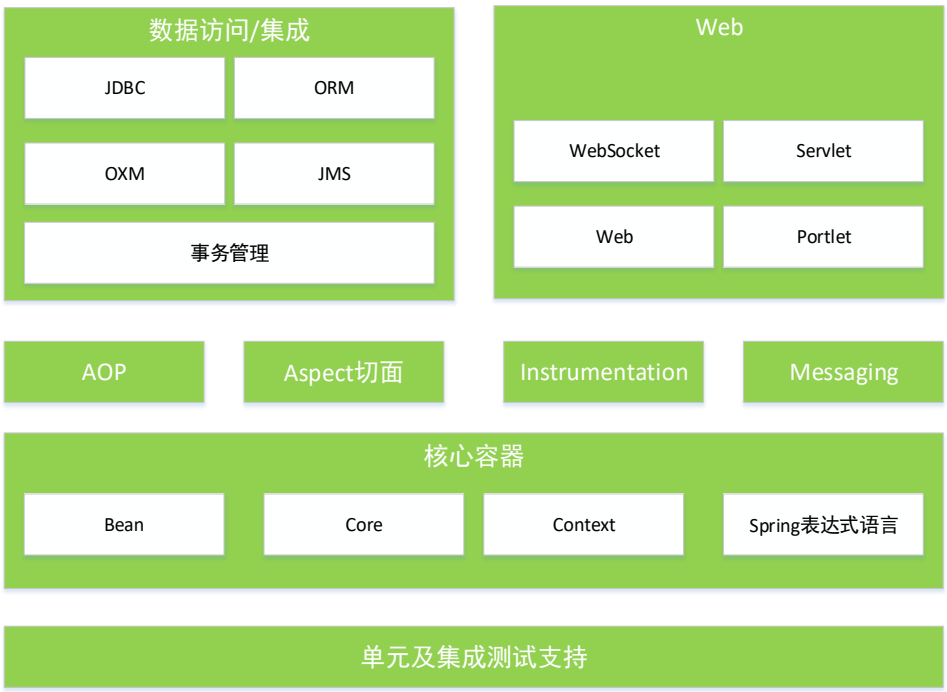
社区活跃、架构设计一致、模块化设计等，使得它们得到了广大 IT 研发者的爱戴。

1.1.2 Spring Framework 成为了企业级 Java 应用研发的事实标准

Spring Framework 作为 Spring 生态系统的基石，其发展时间最长、也最成熟。如今，它已经成为了企业级 Java 应用研发的事实标准。

针对企业级 Java 应用研发，Spring Framework 提供了一站式解决方案。有些方案是 Spring Framework 原创的，有些是 Spring Framework 集成第三方的。无论是哪种解决方案，它都从本着提升开发体验、确保应用可移植性、遵循一致的架构设计等角度着眼。

下图给出了 Spring Framework 主要内容构成。



图表 2 Spring Framework 主要内容构成

大体而言，Spring Framework 主要包括依赖注入容器（IoC）、面向切面编程（AOP）实现、各种企业级服务（Java EE）集成支持等构成。

Spring Framework 倡导敏捷开发，所以它针对单元及集成测试提供了一流的支持，这种支持融入到 Spring Framework 内部的各个角落中。Spring Framework

不重新发明轮子，比如它同 AspectJ 8 进行深度整合、抽象及简化各种 Java EE 容器服务（而不是发明）等。为确保应用的可移植性，Spring Framework 采用动态代理及 Java 反射机制实现 Spring AOP。

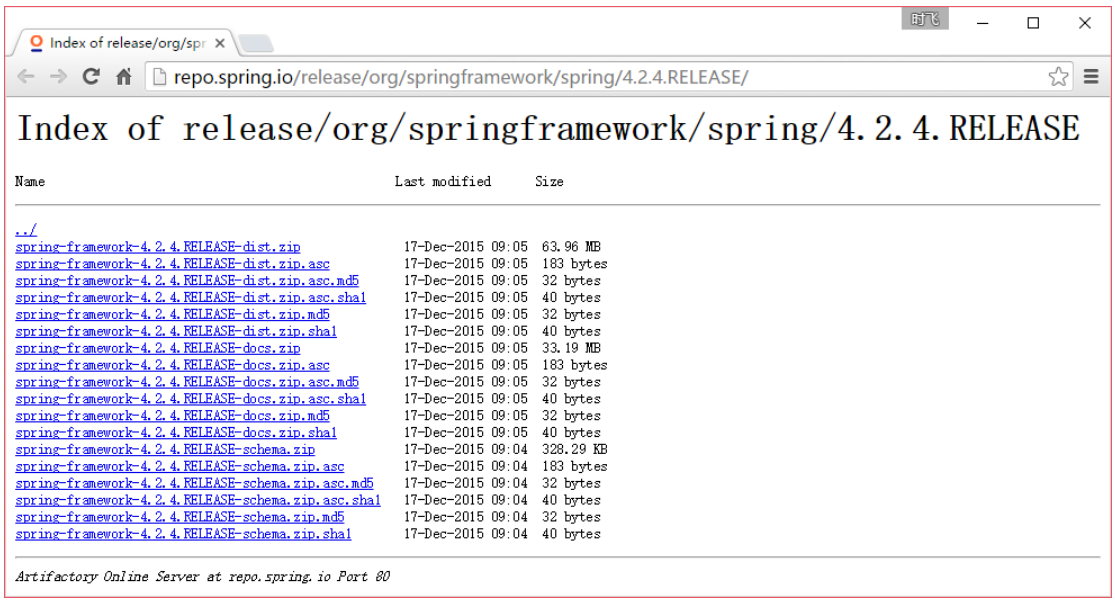
上述各个方面说明，正因为 Spring Framework 遵循了良好的设计理念，使得它成为了企业级 Java 应用研发的事实标准。与此同时，每个 Spring Framework 版本在跟进最新 Java EE 标准方面也是非常出色的，包括即将介绍的 Spring Framework 5.0，它全面跟进 Java EE 7 标准及 Java SE 8。

1.2 下载 Spring Framework 5.0.0.M5

开发者通过如下网址能够下载到对应版本的 Spring Framework 正式发布版，比如 4.2.4.RELEASE。

```
http://repo.spring.io/release/org/springframework/spring/4.2.4.RELEASE/
```

下图展示了这一界面。

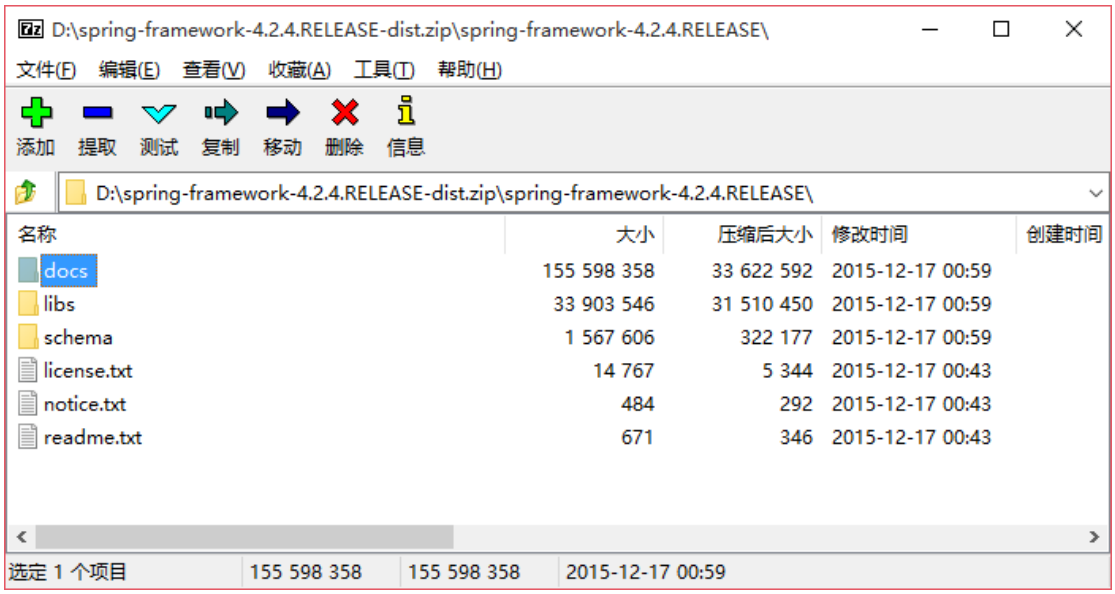


图表 3 Spring Framework 正式发布版界面截图

通常，建议直接下载 spring-framework-4.2.4.RELEASE-dist.zip 正式发布版即可，它的内容最完整，比如包括了各种 JAR 包、HTML 及 PDF 文档、Javadoc、源码等。

1.2.1 正式发布版的内部构成

下图展示了其内部构成。其中，docs 下含有 Javadoc、不同格式的《Spring Framework Reference Documentation》；libs 中含有 Spring Framework 各个模块对应的 Javadoc、编译好的 JAR 包、源码；schema 含有不同版本 Spring Framework 中内置的 XML Schema 集合。



图表 4 Spring Framework 正式发布内部构成（1）

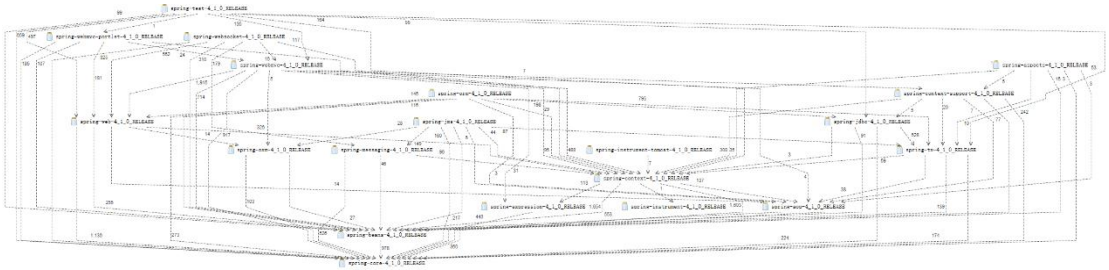
在 libs 目录中主要存在如下一些正式发布 JAR 包。

序号	模块名称	说明
1	spring-aop-4.2.4.RELEASE.jar	AOP 支持
2	spring-aspects-4.2.4.RELEASE.jar	Spring 内置的相关 AOP 切面
3	spring-beans-4.2.4.RELEASE.jar	Spring Beans
4	spring-context-4.2.4.RELEASE.jar	Spring Context
5	spring-context-support-4.2.4.RELEASE.jar	Spring Context Support
6	spring-core-4.2.4.RELEASE.jar	Spring 核心
7	spring-expression-4.2.4.RELEASE.jar	Spring 表达式语言（SpEL）
8	spring-instrument-4.2.4.RELEASE.jar	Spring Instrument
9	spring-instrument-tomcat-4.2.4.RELEASE.jar	Spring Instrument Tomcat
10	spring-jdbc-4.2.4.RELEASE.jar	JDBC 集成支持

11	spring-jms-4.2.4.RELEASE.jar	JMS 集成支持
12	spring-messaging-4.2.4.RELEASE.jar	Spring Messaging
13	spring-orm-4.2.4.RELEASE.jar	Spring O/R Mapping 集成支持
14	spring-oxm-4.2.4.RELEASE.jar	Spring Object/XML Mapping 集成支持
15	spring-test-4.2.4.RELEASE.jar	单元及集成测试支持
16	spring-tx-4.2.4.RELEASE.jar	事务管理支持
17	spring-web-4.2.4.RELEASE.jar	Web 集成支持
18	spring-webmvc-4.2.4.RELEASE.jar	Spring Web MVC
19	spring-webmvc-portlet-4.2.4.RELEASE.jar	Spring Portlet MVC
20	spring-websocket-4.2.4.RELEASE.jar	WebSocket 支持

图表 5 Spring Framework 正式发布内部构成（2）

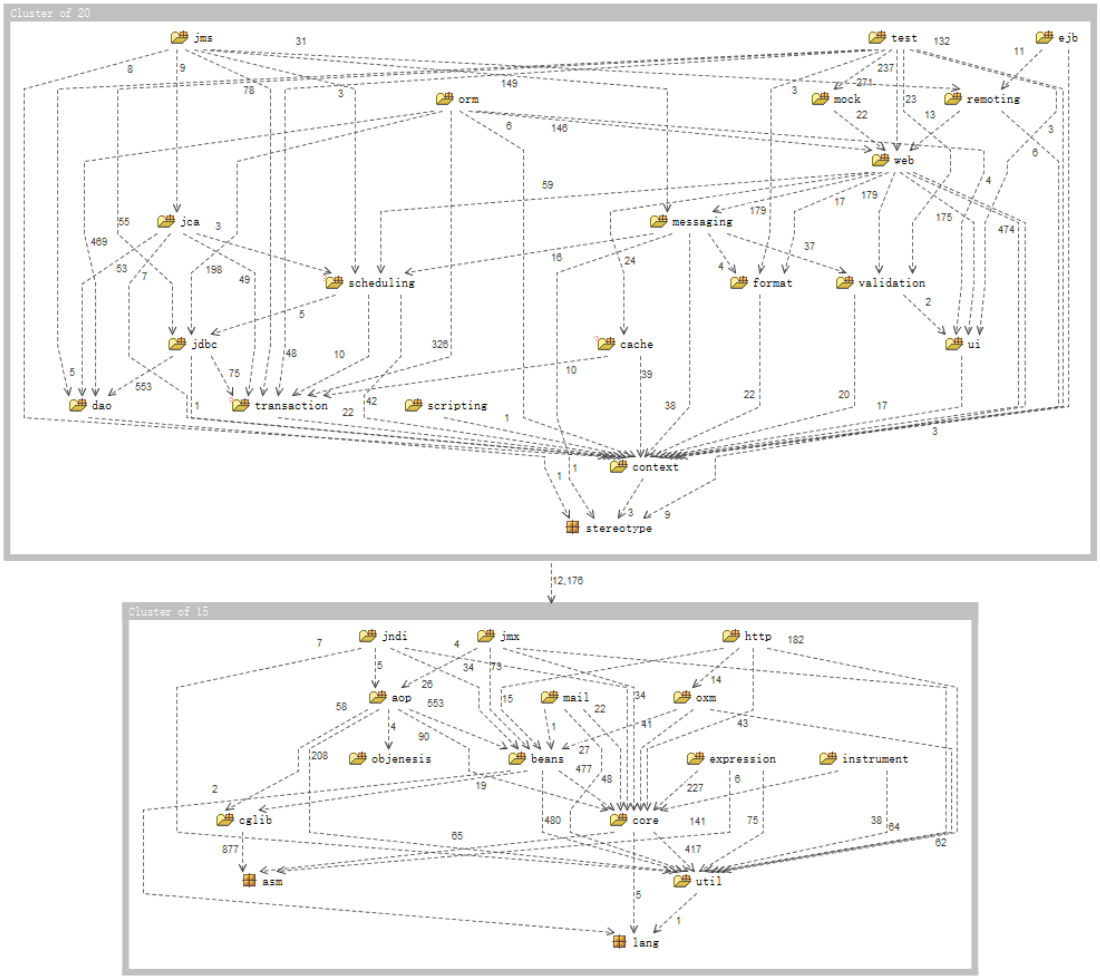
Spring Framework 5.0 系列作品会围绕这二十个模块进行系统性阐述，这些模块间的层次结构[1]如下。



图表 6 Spring Framework 5.0 中各模块间层次结构（1）

下面从 Java package 角度给出了整个 Spring Framework 5.0 的内部层次结构。

[1] 这些层次结构图是通过 http://structure101.com/的 Structure101 Studio for Java 产品生成的，这是一款非常不错的、用于分析软件架构的工具产品，值得严重推荐。



图表 7 Spring Framework 5.0 中各模块间层次结构（2）

从上述两张各模块间层次结构图能够看出，Spring Framework 5.0 的设计及落地工作质量非常高。无论是对于研究 Spring Framework 5.0 本身的内部实现，还是扩展 Spring Framework 5.0，一流的模块化能力显得非常重要，而 Spring Framework 做到了。

1.3 手工构建 Spring Framework 5.0.0.M5 源码

如果需要手工构建 Spring Framework 工程，则需要开发者去 GitHub 下载到完整的项目源码。尽管 Spring Framework 正式发布版包括了各模块的 Java 源码，但为构建这些源码所需要的其他信息没有提供。

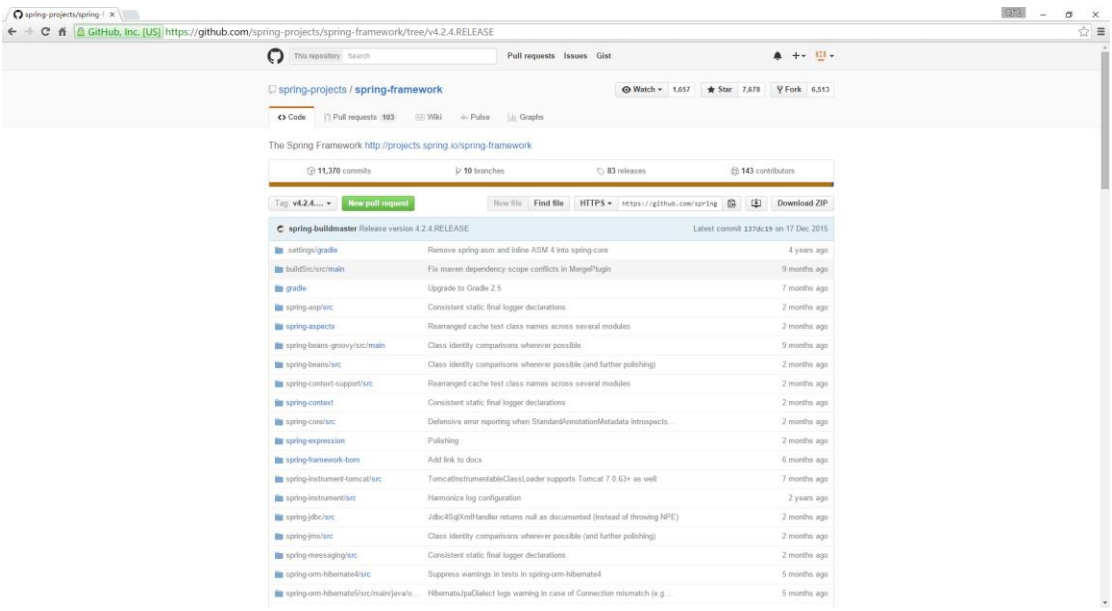
1.3.1 基于 GitHub 库构建 Spring Framework 源码

https://github.com/spring-projects/spring-framework，这一位置存放了 Spring

Framework 项目源码，其中 Spring Framework 5.0.0.M5 的源码可以通过如下 URL 下载到。

```
https://github.com/spring-projects/spring-framework/tree/v5.0.0.M5
```

通过单击这一页面中的“Download ZIP”按钮，能够下载到完整的项目源码，比如 spring-framework-4.2.4.RELEASE.zip。具体见下图。



图表 8 下载 Spring Framework 5.0.4.RELEASE 项目源码

下载到 spring-framework-4.2.4.RELEASE.zip（大约 16.8M）后，将它解压到某一位置，比如 D:\。在 D:\spring-framework-4.2.4.RELEASE 目录中执行如下命令能够详细看到各种构建任务。

```
D:\spring-framework-4.2.4.RELEASE>gradlew tasks
```

运行 build 任务后，D:\spring-framework-4.2.4.RELEASE\build\distributions 位置将生成 spring-framework-4.2.4.RELEASE-dist.zip 发布包，具体如下。

```
D:\spring-framework-4.2.4.RELEASE>gradlew build
```

如果运行 build 任务期间抛出 JVM 内存不够相关异常，则建议调整 gradlew.bat 中的如下相关内容。

```
-XX:MaxMetaspaceSize=2048m -Xmx2048m -XX:MaxHeapSize=512m
```

注意，由于 Spring Framework 5.0.4 的编译需要基于 JDK 1.8 进行，因此开发者务必保证正确安装了 JDK 1.8，比如如下类似命令行能够成功运行。

```
D:\spring-framework-4.2.4.RELEASE>java -verbose:class -version
[Opened C:\jdk1.8.0_72\jre\lib\rt.jar]
[Loaded java.lang.Object from C:\jdk1.8.0_72\jre\lib\rt.jar]

.....

java version "1.8.0_72"
Java(TM) SE Runtime Environment (build 1.8.0_72-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.72-b15, mixed mode)
[Loaded java.lang.Shutdown from C:\jdk1.8.0_72\jre\lib\rt.jar]
[Loaded java.lang.Shutdown$Lock from C:\jdk1.8.0_72\jre\lib\rt.jar]
```

从上述命令输出能够看到，JDK 1.8 被正确安装在 C:\jdk1.8.0_72 位置。

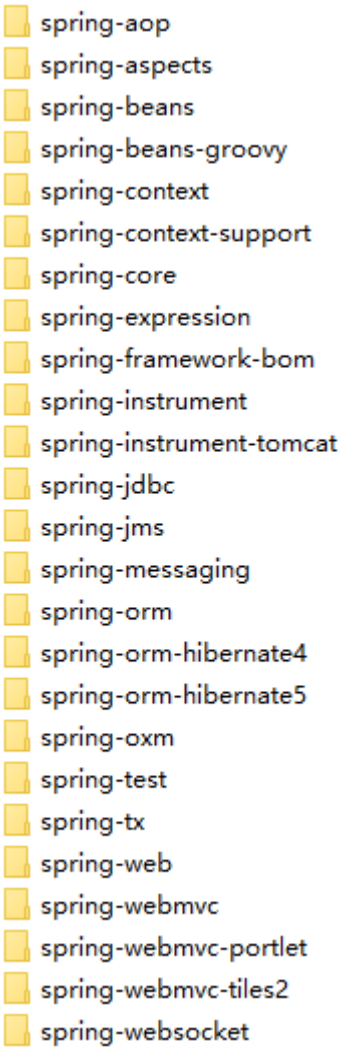
1.3.2 将 Spring Framework 源码导入到 Eclipse 中

由于 Spring Framework 5.0.4 本身是依托 JDK 1.8 研发的，因此需要依托 Eclipse 4.4+版本的 Eclipse。这里采用 4.5 版本的 Eclipse，即 Mars 版本。

在将 Spring Framework 源码导入到 Eclipse Mars 之前，需要先借助如下批处理脚本生成各个模块的 Eclipse 项目文件。

```
D:\spring-framework-4.2.4.RELEASE>import-into-eclipse.bat
```

开发者依据 import-into-eclipse.bat 的提示信息进行操作即可。一旦成功运行后，我们可以在 Eclipse 中导入如下 Spring Framework 项目集合。



图表 9 Spring Framework 项目集合

有了上述 Spring Framework 项目集合后，开发者可以方便地研究其源码实现、内核，包括相关调试工作。

值得注意的是，我们严重推荐 Spring Tool Suite 开发工具，它在 Eclipse 基础上整合了各种常见插件，尤其是在支持 Spring 生态系统方面是一流的。有关 Spring Tool Suite 的相关介绍，请参考本书最后一章内容。

下章开始将逐步深入到 Spring Framework 5.0 的各个组成部分中。

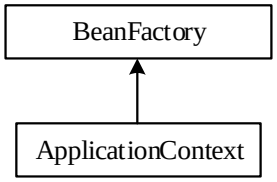
2 Spring IoC 容器

DI 容器（或称之控制反转容器，Inversion of Control，IoC），负责管理宿主其中的 Spring 受管 Bean（POJO），比如生命周期管理、协作者、事件分发、资源查找等。与此同时，Spring 内置了一流的 AOP 技术实现，并同 AspectJ 进行了无缝集成。

Spring 提供的 IoC 容器和 AOP 技术实现构成了 Spring 的核心内容，它们是 Spring 元框架，其中 Spring 内置的 Java EE 服务抽象和集成便是架构在这一元框架基础上的。

2.1 DI 及 Spring DI 概述

类似于 EJB 容器管理 EJB 组件一样，Spring DI 容器负责管理宿主在其中的受管 Bean，或者称之为受管 POJO、Spring Bean 等。就目前来看，Spring 内置了两种基础 DI 容器，即 BeanFactory 和 ApplicationContext，它们间的关系见下图。

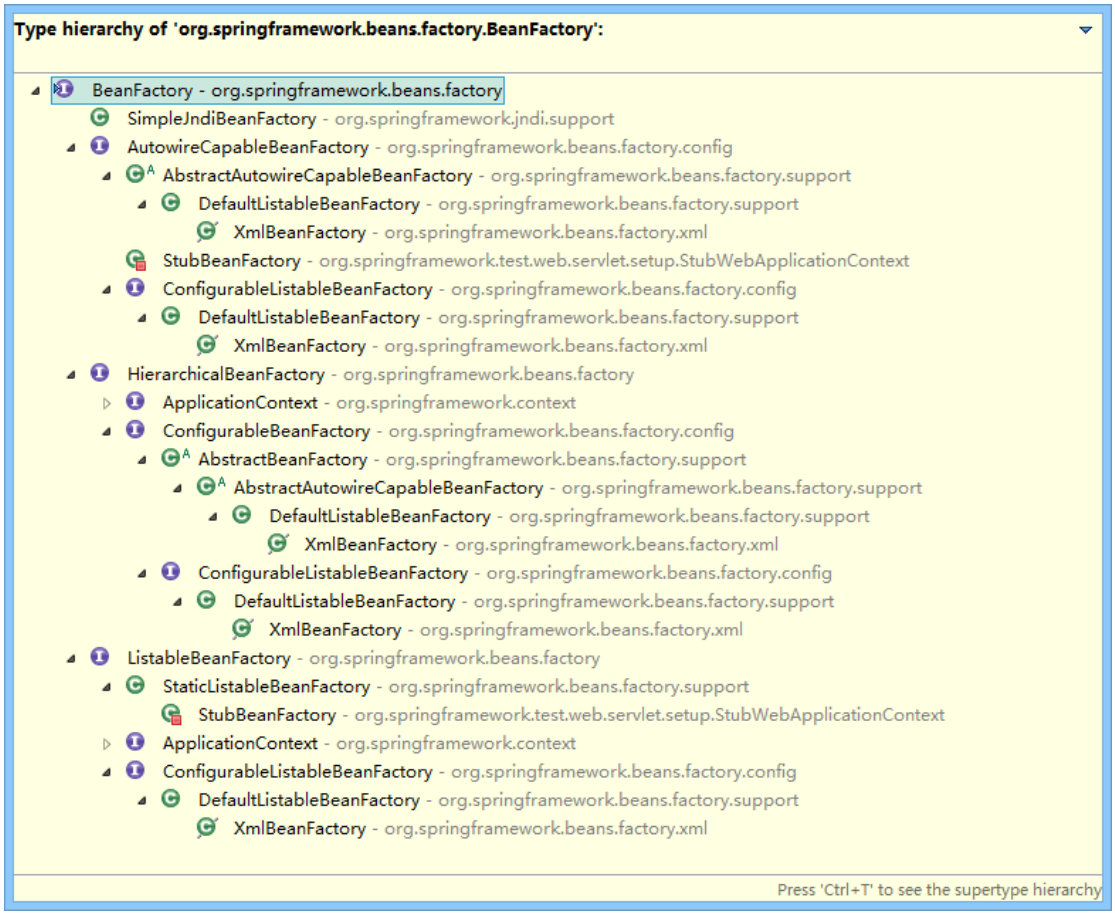


图表 10 两种基础 DI 容器

下面来分别研究这两种 DI 容器。

2.1.1 面向 Java ME/Java SE 的 BeanFactory

BeanFactory 主要用在内存、CPU 资源受限场合，比如 Applet、手持智能设备等。它内置了最基础的 DI 功能，比如配置框架、基础功能。下图完整展示了 Spring Framework 内置的 BeanFactory 接口及其子类集合，开发者会经常使用到其中列举出的各种实现，比如 XmlBeanFactory。



图表 11 BeanFactory 接口及其子类集合

在企业级计算环境，开发者往往要使用 ApplicationContext，而 BeanFactory 是不能够胜任的。

2.1.2 面向 Java EE 的 ApplicationContext

在 BeanFactory 基础上，ApplicationContext 提供了大量面向企业计算所需的特性集合，比如消息资源的国际化（i18n）处理、简化同 Spring AOP 的集成、内置事件支持、针对 Web 应用提供了诸多便利、资源操控等。下图展示了 ApplicationContext 接口及其子类集合，开发者会经常在各种场合使用到其中列举出的 DI 实现。比如，面向 Web 应用的 XmlWebApplicationContext 容器、基于注解存储 DI 元数据的 AnnotationConfigApplicationContext 容器、适合于各种场景的 ClassPathXmlApplicationContext 和 FileSystemXmlApplicationContext 容器、面向 Portal 应用的 XmlPortletApplicationContext 容器、面向 JCA 资源适配器环境的 ResourceAdapterApplicationContext 容器等。

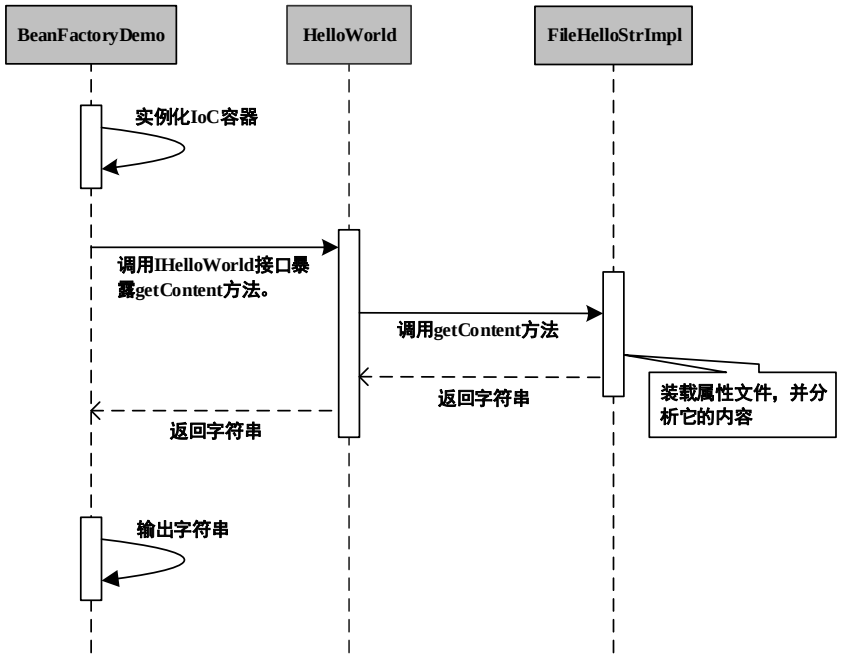
图表 12 ApplicationContext 接口及其子类集合

值得注意的是，ApplicationContext 继承了 BeanFactory 的所有特性。无论如何，本书的所有场合将围绕这两种类型的 DI 容器展开阐述。

2.1.3 “helloworld”示例

本节内容将完整展示一 Spring “helloworld”示例。

位于 Eclipse iocdemo 项目的 BeanFactoryDemo 示例应用是本小节的主角，它同 IoC 容器中受管 Bean 的关系见下图。



图表 13 BeanFactoryDemo 同受管 Bean 的交互

FileHelloStrImpl 实现了 IHelloStr 接口，并从 helloworld.properties 属性文件读取到键值，对应的代码片断如下（摘自 test.FileHelloStrImpl.java 代码）。

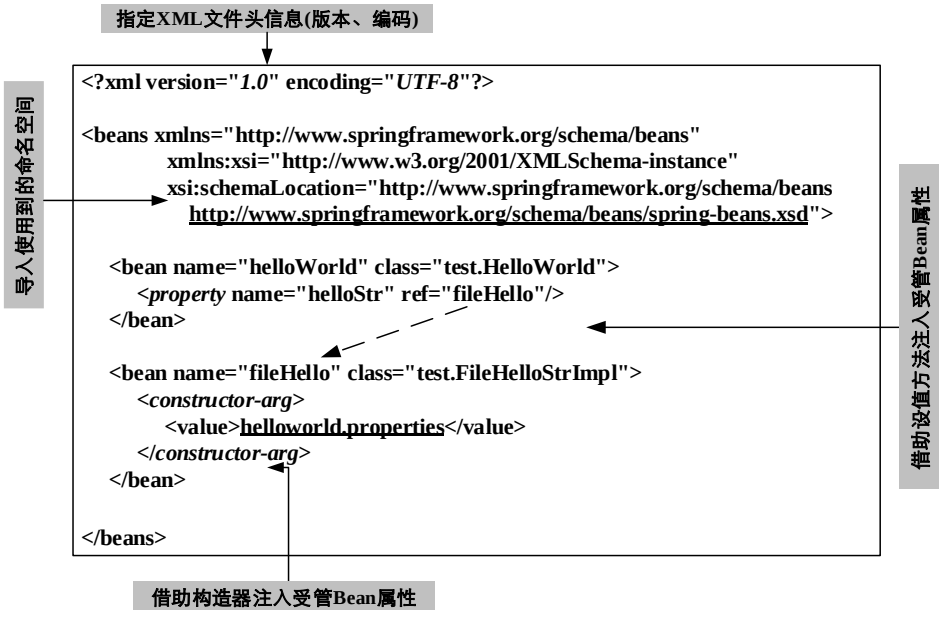
```
public String getContent() {
        String helloworld = "";

        try {
            Properties properties = new Properties();
            //读入输入流
            InputStream is = getClass().getClassLoader().getResourceAsStream(
                    propfilename);
            properties.load(is);
            is.close();
            //获得 helloworld 键对应的取值
            helloworld = properties.getProperty("helloworld");
        } catch (Exception ex) {
            log.error("", ex);
        }
```

```
        return helloworld;
}
```

HelloWorld 实现类会通过 IHelloStr 接口调用到上述 getContent 方法，示例代码如下（摘自 test.HelloWorld.java 代码）。

```
private IHelloStr helloStr;

public void setHelloStr(IHelloStr str) {
        this.helloStr = str;
}

public String getContent() {
        return helloStr.getContent();
}
```

与此同时，开发者需要将上述两个 POJO 配置在基于 XML Schema 的 Spring XML 文件中，具体见下图。通过<bean/>元素能够声明各个受管 Bean，而通过设值方法（setter）和构建器能够完成其依赖对象（比如，fileHello 受管 Bean、helloworld.properties 文件名）的注入。在定义<bean/>元素时，开发者需要提供 name、id、class 等属性。



图表 14 beanfactory.xml 配置详解

最后，开发者还需要提供 test.BeanFactoryDemo 类，它负责装载上述 XML 文件，并分析这些文件。之后，它会借助于 XmlBeanFactory 构造 IoC 容器，并获得 helloWorld 受管 Bean，从而调用到 getContent()方法。具体见如下代码示例

（摘自 test.BeanFactoryDemo）。

```
public static void main(String[] args) {
        //从 classpath 路径上装载 XML 配置信息
        Resource resource = new ClassPathResource("beanfactory.xml");
        //实例化 IoC 容器，此时，容器并未实例化 beanfactory.xml 所定义的各个受管 Bean
        BeanFactory factory = new XmlBeanFactory(resource);
        //获得受管 Bean
        IHelloWorld hw = (IHelloWorld) factory.getBean("helloWorld");
        //返回字符串
        log.info(hw.getContent());
}
```

下面给出了运行结果。

```
[INFO] 2014-06-26 01:15:52,138 org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path resource [beanfactory.xml]
[INFO] 2014-06-26 01:15:52,286 test.BeanFactoryDemo - Hello World!
```

☺，我们并没有手工构造任何对象，HelloWorld 和 FileHelloStrImpl 实例都是由 BeanFactory 构造的，而且它们的依赖关系设置也是由这一 DI 容器完成的。我们所做的事情，只是告诉了 BeanFactory 这些对象间的协作关系而已。这正是 DI 的优势所在，“Don’t call me, I’ll call you!”。

Spring Framework 内置了多个 BeanFactory 实现类，而 XmlBeanFactory 的使用最为频繁。在构造完 factory 对象后，IoC 容器的构建工作也宣告完成。默认时，配置的 POJO 都是单例，即整个 IoC 容器（而不是整个 JVM 实例）中仅有一个对应的 POJO 实例。在初始化 BeanFactory 实例后，IoC 容器并不会预先实例化配置文件中已声明的各个 POJO 实例。只有应用在使用到对应的 POJO 时，Spring Framework 才会实例化使用到的受管 Bean（比如，上述 getBean()方法会触发 IoC 容器实例化 fileHello 和 helloWorld 对象）。

来研究 ClassPathXmlApplicationContext 和 FileSystemXmlApplicationContext 吧，它们是 ApplicationContext 类型的 DI 容器。同 XmlBeanFactory 相比，它们的使用更为简单。下面给出了代码示例。

```
ApplicationContext cac = new ClassPathXmlApplicationContext("beanfactory.xml");

ApplicationContext fac =
        new FileSystemXmlApplicationContext("D:/sts-bundle/workspace/springframework42/iocdemo/src/beanfactory.xml");
```

可以看出，ClassPathXmlApplicationContext 会从类路径上查找到 XML 配置文件，而 FileSystemXmlApplicationContext 容器从文件系统获得这类信息。一旦

构造完 ApplicationContext 对象，IoC 容器便会预先实例化配置文件中已声明的各个 POJO 实例。这是同 BeanFactory 的区别之处。

多重称呼

在开发 EJB 3.x 组件过程中，EJB 3.x 组件存在不少称呼，比如企业 Bean、SLSB、SFSB、MDB。起初，Spring Framework 正是为替代和简化 EJB 组件的开发而出现的。那时候，开发者习惯将<bean/>称为，受管 Bean，这个 Bean 同 EJB 有千丝万缕的关系。再后来，POJO 被开发者宠爱，于是<bean/>有了新的称呼，受管 POJO。POJO 是非常普通的 Java 类，它不同于 JavaBean。JavaBean 的定义和命名规则是受 JCP 规范限定的，开发者习惯将它们画等号。因此，<bean/>还有另一个称呼，受管 JavaBean。

开发者在阅读本书过程中，可以将<bean/>的不同称呼同等对待。

开发者如果抱怨 beanfactory.xml 难于编写和维护，则可借助 Spring Tool Suite[2] 内置的 Spring IDE 支持。或者，AnnotationConfigApplicationContext 容器可干掉您的烦恼，因为它可完全借助注解承载 DI 元数据。

2.2 多种依赖注入方式

开发者所处的团队往往存在多人，他们需要相互交流、协作。同样地，同处于同一 DI 容器中的多个受管 Bean 实例也需要相互协作。协作者间的依赖性管理便成为了 DI 容器的基础工作。Spring DI 容器支持多种不同的依赖注入类型，比如设值注入、构建器注入、属性注入、方法注入等。接下来，我们将围绕 Eclipse iocdemo 项目阐述它们。

2.2.1 设值注入

设值（setter）注入指在通过调用无参构建器（或无参静态工厂方法，或工厂 Bean 的非静态工厂方法）实例化受管 Bean 后调用 setter 方法，从而建立起对象之间的依赖关系。比如，通过 test.HelloWorld 提供的 setHelloStr()方法可以完成 fileHello 对象的设值注入，XML 示例配置如下（摘自 iocdemo 项目的 beanfactory.xml）。

[2] 本书最后一章将详细介绍 Spring Tool Suite 的各方面内容。

```
<bean name="helloWorld" class="test.HelloWorld">
   <property name="helloStr" ref="fileHello"/>
</bean>
```

我们可通过 ref 属性指定依赖的对象或 value 属性直接给出目标对象。在调用 HelloWorld 的 setHelloStr()方法后，helloStr 属性便能够引用到 fileHello 对象。

如果希望通过调用无参静态工厂方法获得受管 Bean，则需要往对应的 POJO 类中添加静态方法，这里以修改 HelloWorld 实现类为例，具体详情请参考 FactoryMethodDemo.java 类。比如，HelloWorld 新增了如下 getHelloWorld 静态方法，并返回了 HelloWorld 对象。

```
public static HelloWorld getHelloWorld(){
        return new HelloWorld();
}
```

新的 helloWorld 受管 Bean 定义如下（摘自 factorymethod.xml）。注意，工厂方法可以返回任何对象，这里返回的对象类型刚好是 class 属性指定的类型。

```
<bean id="helloWorld" factory-method="getHelloWorld" class="test.HelloWorld">
          <property name="helloStr" ref="fileHello"/>
</bean>
```

如果希望通过调用工厂 Bean 的非静态工厂方法获得受管 Bean，则需要往对应的 POJO 类中添加非静态方法，具体详情请参 FactoryBeanDemo.java 类。比如，HelloWorld 新增了如下 makeHelloWorld 非静态方法，它返回了 HelloWorld 对象。

```
public HelloWorld makeHelloWorld(){
        return new HelloWorld();
}
```

新的 helloWorld 受管 Bean 定义如下（摘自 factorybean.xml）。开发者需要同时提供 factory-bean（指定创建 POJO 的工厂）和 factory-method（指定创建 POJO 的方法）属性。本书新配置的 helloWorldFactory 受管 Bean 承担了工厂 Bean 的角色。

```
<bean id="helloWorldFactory" class="test.HelloWorld"/>

<bean id="helloWorld" factory-bean="helloWorldFactory" factory-method="makeHelloWorld">
          <property name="helloStr" ref="fileHello"/>
</bean>
```

这里的工厂 Bean 和工厂方法在集成现有遗留企业应用时非常有帮助。

2.2.2 构建器注入

构建器注入，指往构建器传入若干参数而完成的依赖注入，传入的各个参数都是受管 Bean 依赖的对象，这些对象之间构成了协作关系。BeanFactoryDemo 应用中就使用过构建器注入（见如下代码片断，摘自 beanfactory.xml）。当然，这里的注入比较特殊，因为它直接注入了一字符串对象，而不是其他受管 Bean。使用<constructor-arg/>元素能够往构建器传入参数。

```
<bean name="fileHello" class="test.FileHelloStrImpl">
  <constructor-arg>
        <value>helloworld.properties</value>
  </constructor-arg>
</bean>
```

ConstructorDemo.java 示例演示了传入多个参数的情形，下面摘录了部分 XML 片断（位于 constructor.xml 中）。可以看出，在不提供 index（指定参数在构建器中的位置，从 0 开始计算）、type（指定构建器中对应的参数类型）属性时，Spring Framework 会依据给定参数（类型）的顺序来调用相应的构建器。在提供 index、type 属性时，Spring Framework 能够更容易选择相应的构建器。在定义同一受管 Bean 时，Spring Framework 允许混合使用 index 和 type 属性。

```
<bean name="fileHello1" class="test.FileHelloStrImpl">
  <constructor-arg value="helloworld.properties"/>
  <constructor-arg ref="testBean"/>
</bean>

<bean name="fileHello2" class="test.FileHelloStrImpl">
  <constructor-arg ref="testBean"/>
  <constructor-arg value="helloworld.properties"/>
</bean>

<bean name="fileHello3" class="test.FileHelloStrImpl">
  <constructor-arg index="0" value="helloworld.properties"/>
  <constructor-arg index="1" ref="testBean"/>
</bean>

<bean name="fileHello4" class="test.FileHelloStrImpl">
  <constructor-arg index="0" ref="testBean"/>
  <constructor-arg index="1" value="helloworld.properties"/>
</bean>

<bean name="fileHello5" class="test.FileHelloStrImpl">
  <constructor-arg type="test.ITestBean" ref="testBean"/>
  <constructor-arg type="java.lang.String" value="helloworld.properties"/>
</bean>
```

```
<bean name="fileHello6" class="test.FileHelloStrImpl">
   <constructor-arg index="0" ref="testBean"/>
   <constructor-arg type="java.lang.String" value="helloworld.properties"/>
</bean>
```

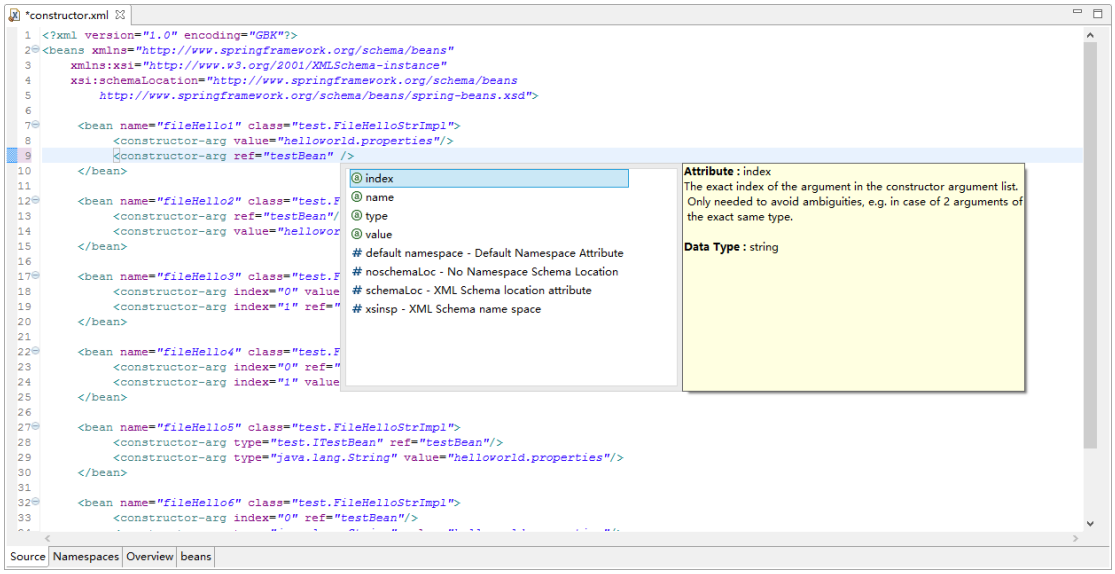
上节内容阐述 factory-bean 和 factory-method 的用法时，开发者并没有为相应的工厂方法提供参数。比如，下面给出的两个工厂方法（摘自 test.HelloWorld.java）就要求为它们提供 IHelloStr 对象。

```
public static HelloWorld getHelloWorld(IHelloStr helloStr){
        HelloWorld hw = new HelloWorld();
        hw.setHelloStr(helloStr);
        return hw;
}

public HelloWorld makeHelloWorld(IHelloStr helloStr){
        HelloWorld hw = new HelloWorld();
        hw.setHelloStr(helloStr);
        return hw;
}
```

开发者借助于<constructor-arg/>元素能够为工厂方法传入所需的对象类型，示例如下（摘自 factoryarg.xml），具体规则同构建器注入。

```
<bean id="helloWorld1" factory-method="getHelloWorld" class="test.HelloWorld">
   <constructor-arg ref="fileHello"/>
</bean>

<bean id="helloWorld2" factory-bean="helloWorldFactory"
        factory-method="makeHelloWorld">
   <constructor-arg ref="fileHello"/>
</bean>
```

借助 Spring Tool Suite 内置的智能支持，开发者能够快速编写和调整 XML 配置文件，下图给出了操作示例。



图表 15 Spring Tool Suite 内置的智能支持

开发者能够决定一切，根据自身需要灵活使用构建器注入吧！

2.2.3 属性注入

在没有提供构建器和设值方法的前提下，借助属性注入，我们同样可以将协作者注入进来。比如，借助@Autowired、@Resource、@EJB 等注解。这里介绍@Autowired 的使用，比如，下面摘录了 iocdemo 项目中的 AutowiredHelloWorld 类，它的 helloStr 属性被应用了@Autowired 注解。

```
public class AutowiredHelloWorld implements IHelloWorld {

    @Autowired
    private IHelloStr helloStr;

    public String getContent() {
        return helloStr.getContent();
    }

}
```

为激活这一注解，要在 DI 容器中配置 AutowiredAnnotationBeanPostProcessor 对象，具体示例如下。此时，我们没有在 XML 配置信息中给出对象间的协作关系。

```
<bean id="autowiredHelloWorld" class="test.AutowiredHelloWorld"/>

<bean name="fileHello" class="test.FileHelloStrImpl">
  <constructor-arg>
      <value>helloworld.properties</value>
```

```
  </constructor-arg>
</bean>

<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>
```

test.AutowiredDemo 示例客户应用加载了这一 Spring DI 容器，具体如下。BeanPostProcessor 类型对象需宿主在 ApplicationContext 容器中，因此它启用了 ClassPathXmlApplicationContext。

```
// 从 classpath 路径上装载 XML 配置信息
ApplicationContext aac = new ClassPathXmlApplicationContext("autowired.xml");

// 获得受管 Bean
IHelloWorld helloWorld = (IHelloWorld) aac.getBean("autowiredHelloWorld");
log.info(helloWorld.getContent());
```

2.2.4 方法注入

Spring Framework 提供了查找（Lookup）方法注入和方法替换（Replacement）注入。Spring Framework 借助于 CGLIB 库提供的 bytecode 生成技术实现了方法注入。考虑到应用的可移植性，Spring Framework 内置了 CGLIB 库，其包名进行了重新编排，具体参考 org.springframework.cglib 包。

查找方法注入，指重载受管 Bean 的（抽象）方法。它会将查找到的其他受管 Bean（通常为原型）替换现有方法的返回结果，从而起到方法注入的效果。在实施查找方法注入时，开发者需要启用<bean/>元素中的<lookup-method/>子元素，附录 A 完整地给出了相关介绍。

位于 Eclipse iocdemo 项目中的 test.LookupDemo 客户应用演示了查找方法注入。比如，test.LookupBeanImpl 分别定义了 createAbstractLookedBean()抽象方法和 createLookedBean()方法，代码示例如下。被查找方法注入机制实施的目标类（比如，LookupBeanImpl）可以是抽象类。运行期，Spring Framework 会自动实现 createAbstractLookedBean()方法，从而返回开发者在 XML 中指定的对象（见后续内容）。相比之下，createLookedBean()会被 Spring 重载。

```
public abstract class LookupBeanImpl {
    private static final Log log = LogFactory.getLog(LookupBeanImpl.class);

    private String name;

    public LookupBeanImpl(String nam){
        this.name = nam;
```

```
    }

    private int count;

    public abstract LookedBean createAbstractLookedBean();

    public LookedBean createLookedBean(){
        log.info("进入 LookupBeanImpl 实现的 createLookedBean()方法");
        return new LookedBean();
    }

    public int getCount() {
        return count;
    }

    public void setCount(int count) {
        this.count = count;
    }

    public String getName() {
        return name;
    }
}
```

随后，开发者需要在 XML 中配置相关信息，具体如下（摘自 lookup.xml）。上述两个方法分别被指定在<lookup-method/>元素的 name 属性中，运行期，lookedBean 受管 Bean 会作为方法的调用结果返回给调用者。由于 lookedBean 的 scope 属性置为 prototype，因此每次调用 createAbstractLookedBean() 和 createLookedBean()方法时，DI 容器始终会生成新的 LookedBean 实例。如果不指定 scope 属性，则总是返回同一 LookedBean 实例。有关 scope 属性的介绍，请参考“Spring 受管 Bean 的作用范围”节内容。

```
<bean id="lookupBean" class="test.LookupBeanImpl">
    <lookup-method name="createAbstractLookedBean" bean="lookedBean"/>
    <lookup-method name="createLookedBean" bean="lookedBean"/>
    <property name="count" value="10"/>
    <constructor-arg>
        <value>lookupBean</value>
    </constructor-arg>
</bean>

<bean id="lookedBean" class="test.LookedBean" scope="prototype">
    <property name="time" value="2000"/>
</bean>
```

LookupDemo 客户代码摘录如下。注意，每次调用 createAbstractLookedBean() 和 createLookedBean()方法时，DI 容器都会生成新的 LookedBean 实例，开发者

可以通过运行控制台得以证实。

```
//实例化 IoC 容器
ApplicationContext aac = new ClassPathXmlApplicationContext("lookup.xml");

//获得受管 Bean
LookupBeanImpl lb = aac.getBean("lookupBean");

log.info(lb.createAbstractLookedBean());
log.info(lb.createAbstractLookedBean());

log.info(lb.createLookedBean());
log.info(lb.createLookedBean());

log.info(lb.getCount());
log.info(lb.getName());
```

方法替换注入，指使用其他受管 Bean 实现的方法替换掉目标 POJO 中的现有方法。在实施方法替换注入时，开发者需要启用<bean/>的<replaced-method/>子元素。

位于 Eclipse iocdemo 项目中的 test.ReplacementDemo 客户应用展示了方法替换注入。ReplacedBeanImpl 实现了如下两个 computeResult()方法。被方法替换注入机制实施的目标类（比如，ReplacedBeanImpl）不能是抽象类，但它可以实现特定接口。

```
public class ReplacedBeanImpl implements IReplacedBean{
	private static final Log log = LogFactory.getLog(ReplacedBeanImpl.class);

	private String name;

	private int count;

	public ReplacedBeanImpl(String nam) {
		this.name = nam;
	}

	public String computeResult(String key) {
		log.info("进入 ReplacedBeanImpl 实现的 computeResult(String key)方法");
		return this.name + "-" + key + "-" + count;
	}

	public String computeResult(int key) {
		log.info("进入 ReplacedBeanImpl 实现的 computeResult(int key)方法");
		return this.name + "-" + key + "-" + count;
	}

	public String getName() {
		return name;
```

```
    }

    public int getCount() {
        return count;
    }

    public void setCount(int count) {
        this.count = count;
    }
}
```

为了在运行期替换 computeResult()方法，开发者定义的 POJO（ReplacedBean）需要实现 Spring 指定的 MethodReplacer 接口，其定义如下。其中，obj 指操作的目标对象、method 指操作的目标方法（比如 computeResult()）、args 指传入目标方法的参数。

```
public interface MethodReplacer {

    //重新实现目标方法
    Object reimplement(Object obj, Method method, Object[] args) throws Throwable;

}
```

ReplacedBean 定义如下，它简单地处理了传入 computeResult 方法的参数。

```
public class ReplacedBean implements MethodReplacer{

    //实现 MethodReplacer 定义的方法
    public Object reimplement(Object obj, Method method, Object[] args) throws Throwable {
        Assert.notEmpty(args);
        return args[0] + ", haha!";
    }

}
```

最后，我们需要将相关信息集成到 replace.xml 中，具体如下。使用 replacer 属性指定的受管 Bean 能够在运行时替代 computeResult()的原生实现。此时，开发者也能够同时使用多种依赖注入机制。如果目标类过载了目标方法，则还需要借助<arg-type/>元素标识目标方法。<arg-type/>的 match 属性取值可以是参数类型对应的全限定名（FQN）的子集，比如 java.lang.String、lang.String 和 String。

```
<bean id="replacementBean" class="test.ReplacedBeanImpl">
    <replaced-method name="computeResult" replacer="replacedBean">
        <arg-type match="int"/>
    </replaced-method>
    <replaced-method name="computeResult" replacer="replacedBean">
        <arg-type match="java.lang.Str"/>
    </replaced-method>
```

```
        <property name="count" value="10"/>
        <constructor-arg>
                <value>replacementBean</value>
        </constructor-arg>
</bean>

<bean id="replacedBean" class="test.ReplacedBean"/>
```

test.ReplacementDemo 示例客户应用如下，很显然，ReplacedBeanImpl 的两个 computeResult()业务方法会被 ReplacedBean 替换掉。运行 ReplacementDemo 期间，开发者不能从控制台浏览到 computeResult ()方法中“log.info”输出的日志信息。

```
//实例化 IoC 容器
ApplicationContext aac = new ClassPathXmlApplicationContext("replace.xml");

//获得受管 Bean
IReplacedBean rb = (IReplacedBean) aac.getBean("replacementBean");

log.info(rb.computeResult("password"));
log.info(rb.computeResult(2007));
```

2.3 借助 Autowiring 策略智能注入协作者

在很多场合，开发人员都会拿 DRY（讨厌重复劳动!，Don't Repeat Yourself!）和 KISS（保持简单、傻瓜!，Keep It Simple, Stupid!）说笑。事实上，这正是优秀开发者必须具备的基本素质，即遵循这两条效率和敏捷原则。为简化 DI 容器中协作者的管理，Spring Framework 引入了 Autowiring 特性，在某种程度上，这正体现了 DRY 和 KISS 的思想。

何谓 Autowiring，即受管 Bean 间的协作关系不用显式指定，启用相关的 Autowiring 策略后，相应的协作关系便会自动得以体现。最终，降低了维护 XML 配置文件的工作强度。下面一一阐述 Spring Framework 支持的各种 Autowiring 策略。

2.3.1 <bean/>元素的 autowire 属性

开发者借助<ref/>元素或 ref 属性能够显式指定当前受管 Bean 的协作者，比如下面给出了配置示例。

```
<bean id="helloWorld" class="test.HelloWorld">
```

```
  <property name="helloStr" ref="fileHello"/>
</bean>
```

试想，如果待配置的协作者数量惊人，而且 Spring XML 配置文件非常多，则维护它们是非常痛苦的事情。通过启用<bean/>元素内置的 autowire 属性，DI 容器能够自动完成协作者的依赖注入。下面一一列举出了 autowire 属性的取值集合。

autowire 属性取值	详细解释
byName	通过属性名完成 autowire 操作。如果某受管 Bean a 含有 b 属性，则 IoC 容器会去所有受管 Bean 中寻找名字（id）为 b 的受管 Bean，并注入到受管 Bean a 的 b 属性中。如果未找到，则不会设置 b 属性的取值
byType	通过判断属性类型完成 autowire 操作。如果某受管 Bean a 中 b 属性的类型（type）为 IBankSecurityDao，则 IoC 容器会去所有受管 Bean 中寻找类型为 IBankSecurityDao 的受管 Bean，并注入到受管 Bean a 的 b 属性中。如果找到多个，则会抛出异常。如果未找到，则不会设置 b 属性的取值
constructor	同 byType，但作用于构建器
no	不启用 autowire 特性，这是 Spring 受管 Bean 的默认行为
default	同<beans/>级别设定的 autowiring 策略

图表 16 autowire 属性取值结合

Eclipse autowiringdemo 项目定义了 TestBean1 类，其主要内容如下（testBean2 和 testBean3 属性对应的 setter 和 getter 方法没有给出）。

```
public class TestBean1 {
        private TestBean2 testBean2;

        private TestBean3 testBean3;

        public TestBean1(){
        }

        public TestBean1(TestBean2 tb2, TestBean3 tb3){
                this.testBean2 = tb2;
                this.testBean3 = tb3;
        }

        ……
}
```

下面（摘自 AutowiringDemo 应用的 autowire.xml）给出了 autowire 属性的若干种用法，它们达到的效果是完全一样的，最终 testBean2 和 testBean3 受管 Bean 被注入到对应的属性中。

```
<bean id="tbByName" class="test.TestBean1" autowire="byName"/>
```

```
<bean id="tbByType" class="test.TestBean1" autowire="byType"/>

<bean id="tbConstructor" class="test.TestBean1" autowire="constructor"/>

<bean id="testBean2" class="test.TestBean2"/>

<bean id="testBean3" class="test.TestBean3"/>
```

在实施 Autowiring 策略期间，如果目标受管 Bean 找不到，则对应的属性值为 null；但如果找到多个匹配对象，则会有异常抛出。如果需要避免异常的抛出，则开发者可以启用<bean/>内置的 primary 属性。下面给出了配置示例，此时，testBean2、testBean2primary 受管 Bean 都满足条件，但由于 testBean2primary 的 primary 属性取值为 true，因此优先使用它。

```
<bean id="tbByType" class="test.TestBean1" autowire="byType"/>

<bean id="testBean2" class="test.TestBean2"/>

<bean id="testBean2primary" class="test.TestBean2" primary="true"/>
```

与此同时，如果目标受管 Bean 同时启用了 autowire 属性和显式地给出了所依赖的协作者集合，比如通过<property/>、ref 属性、<constructor-arg/>等方式显示指定，则 DI 容器会优先认可这些显式方式。一旦存在冲突时，autowire 属性注入的具体取值会被覆盖掉。如果某受管 Bean 不希望成为其他受管 Bean 的协作者，则开发者可以使用<bean/>元素内置的 autowire-candidate 属性。在下面给出的 XML 配置（摘自 AutowiringCandidateDemo 使用的 autowire-candidate.xml）中，TestBean4 中定义了 testBean5 属性、TestBean6 中定义了 testBean4 属性。一旦将 testBean4 的 autowire-candidate 属性置为 false 后，则它只能够 autowire 其他受管 Bean，而不能去充当协作者。因此，运行过程中，testBean6 中的 testBean4 属性取值始终为 null。

```
<bean id="testBean4" class="test.TestBean4" autowire-candidate="false" autowire="byType"/>

<bean id="testBean5" class="test.TestBean5"/>

<bean id="testBean6" class="test.TestBean6" autowire="byType"/>
```

2.3.2 基于@Required 注解加强协作者管理

开发者是否注意到，启用 autowire 属性期间，我们并不能够确定协作者是否找到。通常，协作者没有找到时，NPE 异常可能会抛出，但这可能已经是运行

期行为。我们是否能够在启动应用期间就将这类问题找出来，并告知开发者，从而尽可能避免生产问题的发生。借助@Required 注解，我们能够加强协作者管理。不过，启用这一注解时，需要单独配置如下受管 Bean 定义，并启用 ApplicationContext 容器。

```
<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"/>
```

或者，激活如下配置。

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

    <context:annotation-config/>

</beans>
```

下面展示了@Required 注解的使用（摘自 autowiringdemo 项目中的 TestBean 类）。这一注解只能够作用在方法上。的部分内容摘录如下，这些设值方法都应用了@Required 注解。

```
@Required
public void setHw1(HelloWorld hw1) {
    this.hw1 = hw1;
}

@Required
public void setHw2(HelloWorld hw2) {
    this.hw2 = hw2;
}

@Required
public void setStr(String str) {
    this.str = str;
}
```

事实上，@Required 注解只能够确保其作用的方法是否被调用到，它也不能确保协作者是否成功注入。即使将<null/>取值提供给其作用的方法，则也是没问题的，即骗过@Required 注解。下面给出了配置示例，它将<null/>，即 null 值给了 hw2 属性。

```
<bean id="testBean" class="test.TestBean">
```

```
        <property name="hw1" ref="helloWorld"/>
        <property name="hw2">
                <null/>
        </property>
        <property name="str">
                <value/>
        </property>
</bean>

<bean id="helloWorld" class="test.HelloWorld"/>
```

正常情况下，当应用启动时，如果@Required 注解作用的方法被调用到，则说明协作者确实找到了；如果@Required 注解作用的方法没被调用到，则应用是不能够正常启动的，即会有如下类似异常的抛出。

```
Caused by: org.springframework.beans.factory.BeanInitializationException: Property 'hw1' is required for bean 'testBean'
        at
org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor.postProcessPropertyValues(RequiredAn
notationBeanPostProcessor.java:156)
        at
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.populateBean(AbstractAutowireCapableBea
nFactory.java:1185)
        at
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBe
anFactory.java:537)
```

如果开发者不打算使用@Required，而准备启用自己提供的其他注解，则这也是可以的，RequiredAnnotationBeanPostProcessor 暴露了 requiredAnnotationType 属性，默认时，它使用@Required 注解。比如，ForYouDemo 应用使用了@ForYou 注解，为启用它，开发者只需要在配置 RequiredAnnotationBeanPostProcessor 时指定它即可，示例如下（摘自 foryou.xml）。开发者顺便还要将代码中使用的@Required 注解替换成@ForYou 注解。注意，这些注解只是一种标识，具体使用哪个并没有太多的区别。

```
<bean id="requiredAnnotationBeanPostProcessor"
        class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor">
        <property name="requiredAnnotationType" value="test.ForYou"/>
</bean>
```

2.3.3 基于@Autowired 或@Inject 注解的另一 Autowiring 策略

之前介绍属性注入时，我们已经了解到@Autowired 注解的使用了。为使用@Autowired 或@Inject 注解，开发者需要在 DI 容器中单独配置如下受管 Bean。

```
<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>
```

或者，激活如下配置。

```
<context:annotation-config/>
```

@Inject 注解是 JSR330（javax.inject）引入的，它同@Autowired 注解一样，都可以作用到构建器、属性、方法上。下面展示了这一注解作用到属性上的代码示例。

```
@Inject
private TestBean2 testBean2;
```

默认时，一旦@Autowired 或@Inject 注解作用到构建器，或属性，或方法上，则协作者必须存在，否则应用不能够正常启动。这有点类似于@Required 注解的功效。即，@Autowired 和@Inject 注解内置了@Required 注解的能力，不过，@Autowired 注解允许开发者不启用这一能力。下面给出了代码示例（摘自 autowired.xml）。默认时，required 属性取值为 true，一旦设置为 false 后，即使没有为 testBean3 属性提供协作者，应用也能够“正常”启动。

```
@Autowired(required=false)
private TestBean3 testBean3;
```

下面给出了应用于设值方法的@Autowired 注解。

```
@Autowired(required=false)
public void setTestBean2(TestBean2 testBean2) {
        this.testBean2 = testBean2;
}

@Autowired
public void setTestBean3(TestBean3 testBean3) {
        this.testBean3 = testBean3;
}
```

下面给出了应用于构建器的@Autowired 注解。

```
@Autowired(required=false)
public TestBean1(TestBean2 tb2, TestBean3 tb3) {
        this.testBean2 = tb2;
        this.testBean3 = tb3;
}
```

下面给出了应用于普通方法的@Autowired 注解。

```
@Autowired
public void preparedBean(TestBean2 tb2, TestBean3 tb3){
        this.testBean2 = tb2;
        this.testBean3 = tb3;
}
```

除了能够自动注入普通受管 Bean 外，@Autowired 注解还能够注入一些特殊对象，比如受管 Bean 所在 DI 容器中的各种元数据对象，示例如下。@Autowired 将当前 DI 容器注入进来。

```
@Autowired
private ApplicationContext ac;
```

在@Autowired 的 required 属性与@Required 注解间进行取舍

@Required 是专门用来依赖性检查的注解，而@Autowired 注解的 required 属性也能够用来承担依赖性检查工作。一旦@Required 应用到目标设值方法后，则 DI 容器必须成功调用这一设值方法，@Required 注解才认为依赖性条件得到满足，否则异常将抛出。相比之下，当我们将@Autowired 注解的 required 属性置为 false 取值时，即使未找到合适的协作者，异常始终都不会抛出。

2.3.4 借助@Qualifier 注解细粒度控制 Autowiring 策略

当多个协作者满足 autowire 注入条件时，我们可以启用<bean/>元素的 primary 属性，从而避免异常的抛出。但是，primary 属性不能够满足企业级应用的复杂需求，比如存在多个同一类型的不同协作者需要注入到同一受管 Bean 时。

下面给出了某 XML 配置示例。

```
<bean id="testBean2a" class="test.TestBean2"/>

<bean id="testBean2b" class="test.TestBean2"/>

<bean id="testBean2c" class="test.TestBean2"/>
```

现在希望将它们分别注入到下面给出的属性中。显然，启用 primary 属性是不行的，因为一旦启用它，testBean2a、testBean2b、testBean2c 将引用到同一受管 Bean。

```
@Autowired
private TestBean2 testBean2a;

@Autowired
private TestBean2 testBean2b;

@Autowired
private TestBean2 testBean2c;
```

为此，我们需要启用<qualifier/>元素及@Qualifier 注解。调整后的 XML 配置示例如下。

```
<bean id="testBean2a" class="test.TestBean2">
        <qualifier value="2a"/>
</bean>

<bean id="testBean2b" class="test.TestBean2">
        <qualifier value="2b"/>
</bean>

<bean id="testBean2c" class="test.TestBean2">
        <qualifier value="2c"/>
</bean>
```

相应地，属性的定义也需要作相应调整，下面给出了代码示例。各个@Qualifier 注解分别指定了待注入的受管 Bean，比如“2a”表示要注入<qualifier/>的 value 属性取值为“2a”的目标受管 Bean。

```
@Autowired
@Qualifier("2a")
private TestBean2 testBean2a;

@Autowired
@Qualifier("2b")
private TestBean2 testBean2b;

@Autowired
@Qualifier("2c")
private TestBean2 testBean2c;
```

@Qualifier 注解能够作用于属性、参数、类、其他注解等地方，比如下面给出了参数级的使用示例。

```
@Autowired
private void preparedBean(@Qualifier("2a") TestBean2 testBean2a,
                @Qualifier("2b") TestBean2 testBean2b,
                @Qualifier("2c") TestBean2 testBean2c) {
        this.testBean2a = testBean2a;
        this.testBean2b = testBean2b;
        this.testBean2c = testBean2c;
}
```

开发者还可以基于@Qualifier 注解构建更复杂的 Autowiring 策略，下面给出了一注解示例。

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Qualifier
public @interface FineQualifier {

        String keyFine() default "";
```

```
    String valueFine() default "";

}
```

同使用@Qualifier 注解一样，下面给出了@FineQualifier 应用示例。

```
@Autowired
@FineQualifier(keyFine="key2A", valueFine="value2A")
private TestBean2 testBean2a;

@Autowired
@FineQualifier(keyFine="key2B", valueFine="value2B")
private TestBean2 testBean2b;

@Autowired
@FineQualifier(keyFine="key2C", valueFine="value2C")
private TestBean2 testBean2c;
```

为配合@FineQualifier 注解的使用，XML 配置文件也需要作相应调整，下面给出了调整后的配置示例。

```
<bean id="testBean2a" class="test.TestBean2">
    <qualifier type="test.FineQualifier">
        <attribute key="keyFine" value="key2A"/>
        <attribute key="valueFine" value="value2A"/>
    </qualifier>
</bean>

<bean id="testBean2b" class="test.TestBean2">
    <qualifier type="test.FineQualifier">
        <attribute key="keyFine" value="key2B"/>
        <attribute key="valueFine" value="value2B"/>
    </qualifier>
</bean>

<bean id="testBean2c" class="test.TestBean2">
    <qualifier type="test.FineQualifier">
        <attribute key="keyFine" value="key2C"/>
        <attribute key="valueFine" value="value2C"/>
    </qualifier>
</bean>
```

或者，开发者也可以启用<meta/>元素，配置示例如下。当在同一受管 Bean 中同时指定<qualifier/>和<meta/>元素时，DI 容器会优先使用<qualifier/>。

```
<bean id="testBean2a" class="test.TestBean2">
    <meta key="keyFine" value="key2A"/>
    <meta key="valueFine" value="value2A"/>
</bean>

<bean id="testBean2b" class="test.TestBean2">
    <meta key="keyFine" value="key2B"/>
```

```
        <meta key="valueFine" value="value2B"/>
</bean>

<bean id="testBean2c" class="test.TestBean2">
        <meta key="keyFine" value="key2C"/>
        <meta key="valueFine" value="value2C"/>
</bean>
```

当@FineQualifier 注解没有应用类一级的@Qualifier 注解时，开发者必须借助如下对象注册它。

```
<bean id="customAutowireConfigurer"
                class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">
        <property name="customQualifierTypes">
                <set>
                        <value>test.FineQualifier</value>
                </set>
        </property>
</bean>
```

总之，开发者需要依据自身不同情况，而灵活启用不同的 Autowiring 策略。

2.4 Spring 受管 Bean 的作用范围

应用程序存在全局和局部变量。定义在方法中的变量是局部变量，在离开这一方法时就离开了局部变量的管辖范围。定义在类一级的变量是全局变量，它的作用范围很大，各个方法都能使用它。借助 Spring Framework 开发应用期间，开发者可以定义有不同作用范围（scope）的受管 Bean。下表给出了 DI 容器为受管 Bean 内置的五种作用范围。

作用范围（scope）	具 体 含 义
singleton	默认<bean/>定义。各个 IoC 容器仅仅会实例化对应它的单个受管 Bean 实例。IoC 容器会预先创建好处于这种作用范围的<bean/>定义。即使开发者不指定这一作用范围，默认<bean/>定义正是使用了这一作用范围
prototype	各个 IoC 容器会根据应用的需要实例化对应它的若干个受管 Bean 实例。IoC 容器不会主动实例化处于这种作用范围的<bean/>定义
request	相应的受管 Bean 实例仅持续到单个 HTTP 请求生命周期的结束。这一作用范围仅适用于 Web 领域的 ApplicationContext 实现（比如，XmlWebApplicationContext）
session	相应的受管 Bean 实例会持续到单个 HttpSession 生命周期的结束。这一作用范围仅适用于 Web 领域的 ApplicationContext 实现
globalSession	相应的受管 Bean 实例会持续到全局 HttpSession 生命周期的结束。这一作用范围仅适用于 Web 领域的 ApplicationContext 实现。它是针对 Portal 应用开发而推出的

图表 17 受管 Bean 的五种作用范围

2.4.1 单例和原型

在 Spring Framework 1.x 使能应用中定义<bean/>时，开发者可以借助<bean/>元素暴露的 singleton 属性声明受管 Bean 的作用范围，比如将 singleton 设为 true 时，则意味着整个 IoC 容器仅仅会实例化当前<bean/>定义的单个实例。我们将这种 Bean 称为单例（singleton）。从 Spring Framework 2.x 开始，singleton 属性便被 scope 属性取代了。

下面给出的配置（摘自 scopedemo 项目的 singleton.xml）定义了两个同样类型的 ResInfo 对象。默认时，如果不指定 scope 属性，定义的<bean/>都将是单例。因此，这两个<bean/>定义是等效的。DI 容器会针对它们分别实例化一个单例，即 DI 容器中将存在两个 ResInfo 对象。

```
<bean name="resInfo1" class="test.ResInfo" scope="singleton"/>

<bean name="resInfo2" class="test.ResInfo"/>
```

如果将 singleton（scope）属性设为 false（prototype），则情况将变得不一样。比如，PrototypeDemo 应用使用的 prototype.xml 中配置了如下 UserInfo 和 ResInfo 信息。由于 resInfo 的 scope 属性取值是 prototype，因此应用每次调用它时，DI 容器都会实例化新的实例。

```
<bean name="resInfo" class="test.ResInfo" scope="prototype"/>

<bean name="userInfo" class="test.UserInfo">
        <property name="ri1" ref="resInfo"/>
        <property name="ri2">
                <bean class="test.ResInfo"/>
        </property>
</bean>
```

在借助 ClassPathXmlApplicationContext 完成容器的构建后，IoC 容器会生成两个 ResInfo 实例。由于 userInfo 中的 ri1 引用到 resInfo 对象，因此它会触发新 ResInfo 实例的创建，这是第一个实例；在为 ri2 指定 ResInfo 时使用了内部 Bean，这时，DI 容器会创建新的实例，这是第二个实例。应用使用 getBean("resInfo") 或 resInfo 被其他受管 Bean 引用时，新的受管 Bean 实例就会被 IoC 容器创建。本书将指定 singleton="false"（scope="prototype"）的<bean/>称为原型（prototype）。

注意，DI 容器不会主动实例化原型受管 Bean。

内部 Bean

我们将定义在<property/>或<constructor-arg/>元素中的<bean/>称为内部 Bean。开发者不用为此类 Bean 指定 id 或 name 属性，因为 DI 容器会忽略它们的存在。另外，scope 属性也不用开发者指定，因为容器也会忽视它。其实，内部 Bean 是一种特殊原型，除了定义它的<bean/>能够使用它外，别的<bean/>不可能看到它的存在。

如果 PrototypeDemo 应用希望通过 getBean("resInfo")方式获得 ResInfo 实例，则 IoC 容器会创建新的实例给它。但如果希望通过 getBean("userInfo")间接创建 ResInfo 实例，则 IoC 容器不会理会它，因为在构建好 ApplicationContext 容器后，userInfo 实例已经被构建好并缓存在 Spring IoC 容器中。为了让 Spring IoC 容器理会 getBean("userInfo")操作，则只有一个办法，即启用延迟初始化特性，通过往<bean/>定义中添加 lazy-init="true"即可启用这一特性，示例如下。默认时，<bean/>并没有启用这一特性，即 lazy-init 取值为 false。

```
<bean name="userInfo" class="test.UserInfo" lazy-init="true">
    <property name="ri1" ref="resInfo"/>
    <property name="ri2">
        <bean class="test.ResInfo"/>
    </property>
</bean>
```

DI 容器在初始化各个<bean/>定义时，如果碰到那些定义了“lazy-init="true"”内容的<bean/>，则不会主动去实例化它们。只有应用显式调用 getBean("userInfo")或 userInfo 被协作者引用时，IoC 容器才会去实例化 userInfo 对象。开发者要注意到 lazy-init="true"与 singleton="false"的区别。无论如何，启用了延迟初始化的<bean/>最多会被实例化一次，随后的引用或 getBean()操作只会在 IoC 容器缓存中获得，但是原型 Bean 却不一样，因为容器不会去缓存这类实例。

打破延迟初始化神话

在<bean/>定义中添加 lazy-init="true"后，是否就意味着在构造 IoC 容器时这一<bean/>不会被实例化呢？不要简单地回答这个问题。设定 lazy-init="true"后，并不意味着容器就会启用延迟初始化。开发者试想，如果受管 Bean A 需要引用受管 Bean B，A 未启用延迟初始化，而 B 却启用了。当容器在完成 A 的实例化工作时，容器发现它引用了 B，因此容器会马上实例化 B。无论 B 是否设定

lazy-init="true"，只要协作者需要它时，IoC 会毫不犹豫地去完成 B 的实例化。

总之，开发者在使用这一特性时一定要仔细考虑好受管 Bean 之间的协作关系。

2.4.2 仅仅适合于 Web 环境的三种作用范围

Spring Framework 1.x 仅仅支持单例和原型作用范围。单例非常适合 DAO 对象，因为 DAO 对象是不存储特定客户信息的，原型适合于特定客户，而且在原型实例交付给应用后 IoC 容器不再负责它的生命周期管理。

开发 Web 应用过程中，开发者可以在 JSP 或其他地方定义 JavaBean 的作用范围，比如 page、request、session、application，示例代码如下。

```
<jsp:useBean id="sessionProcess" scope="session" class="test.SessionProcess"/>
```

由于受到 JSP 中 JavaBean 的启发，Spring Framework 2.x 针对 Web 和 Portal 应用引入了另外三种作用范围：request、session 和 globalSession。它们对 Web 应用使用的具体框架不作任何限定，比如 Tapestry、Spring Web MVC、Spring Portlet MVC、Struts、JSF 都能够支持。其中，request 指 Web 容器服务每个 HTTP 请求时，受 scope 控制的单个受管 Bean 实例在整个 request 范围内有效。一旦 Web 容器处理完客户请求，这一实例也将被丢弃。而 session 的作用范围更广，在整个 session 持续有效的时间里，受 scope 控制的单个受管 Bean 实例都可供特定用户使用。从表面上看，这里的 session 作用范围同 useBean 中定义的 scope="session"是等效的。在 JSP 中，sessionProcess 对象由 Web 容器创建；而在 Spring Framework 中，受 scope 控制的受管 Bean 实例由 Spring IoC 容器创建，而且 Spring Framework 负责将创建的实例存储到 HttpSession 中。另外，globalSession 仅适合于 Portal 环境。

这里将结合 scopewebdemo 项目展开对 request 和 session 作用范围的论述。这一 Web 应用的主页见下图。受到 BEA WebLogic 服务器中测试集群的一简单 Web 应用的启发，我们开发了这一示例，通过位于界面上方的新增会话属性 Tab 页，能够新增会话属性到 HttpSession 中。通过位于界面下方的打印会话属性 Tab 页，能够浏览到 HttpSession 中存储的各个属性。每次访问到这个应用的首页时，Web 容器便会自动创建 session。当然，在实际应用开发中这是不好的开发实践。如果希望将当前会话失效，则可以通过单击下方的“失效当前 session”按钮达

到这一目的。随后还可以返回到首页，此时新的 session 被创建了，而原有的 session 信息等待 Web 容器去处理。

图表 18 scopewebdemo 应用主页

为启用针对 Web 应用的这三种作用范围，开发者需要在 web.xml 中定义对应的辅助对象，比如监听器、过滤器。或者，如果开发者直接使用 Spring Web MVC 或 Spring Portlet MVC，则不用单独定义它们，因为 DispatcherServlet 和 DispatcherPortlet 前端控制器已经内置了相应的能力。

Spring Framework 针对不同 Web 容器提供了两套方案，一套是基于 ServletRequestListener 实现的，它会对 ServletRequestEvent 事件进行监控，具体配置如下。由于 ServletRequestListener 是 Servlet 2.4 才引入的新特性，因此如果开发者使用的 Web 容器未实现这一特性，则只能使用 Filter。监听器的配置非常简单。

```
<listener>
    <listener-class>
        org.springframework.web.context.request.RequestContextListener
    </listener-class>
</listener>
```

另一套是基于 Servlet 过滤器实现的，具体配置如下。Java EE 容器在处理 HTTP 请求之前，RequestContextFilter 会将 HTTP Request 对象保存起来。

```
<filter>
```

```
        <filter-name>requestContextFilter</filter-name>
        <filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>
</filter>

<filter-mapping>
        <filter-name>requestContextFilter</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

配置 DI 容器相关信息时，开发者需要注意 scope 控制的<bean/>的定义。比如，scopewebdemo 的 applicationContext.xml 中配置了如下一些受管 Bean。开发者可以针对目标受管 Bean 的不同性质来合理使用不同的作用范围。注意，nameList 和 valueList 使用了 request，而 sessionContent 使用了 singleton 作用范围。这里的 sessionContent 同定义普通<bean/>无任何差别，而且它还引用到 nameList 和 valueList 对象。因此，DI 容器只会创建单个 sessionContent 实例。

```
<bean id="nameList" class="test.NameList" scope="request">
        <aop:scoped-proxy proxy-target-class="true"/>
</bean>

<bean id="valueList" class="test.ValueList" scope="request">
        <aop:scoped-proxy proxy-target-class="true"/>
</bean>

<bean id="sessionContent" class="test.SessionContent" scope="singleton">
        <property name="nameList" ref="nameList"/>
        <property name="valueList" ref="valueList"/>
</bean>
```

开发者是否还注意到<aop:scoped-proxy/>元素。通常，普通受管 Bean 之间在进行协作时，如果它们未启用延迟初始化，则一旦受管 Bean（A）已经对其他受管 Bean（B）实施了依赖注入，则 A 持有的总是 B 实例。由于 nameList 和 valueList 的作用范围是 request，这意味着 IoC 容器中将存在大量的 NameList 和 ValueList 实例。

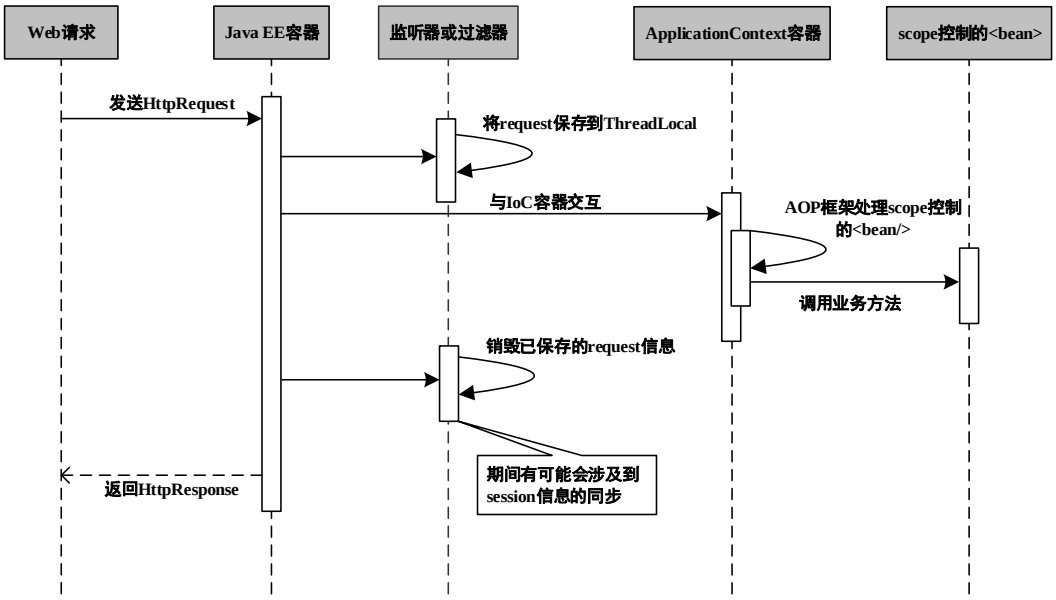
配置<aop:scoped-proxy/>元素后，sessionContent 引用到的不再是 NameList 和 ValueList 实例本身，而是它们的代理实现，这些代理控制了具体实例的一切行为。现在，开发者可以配置 Web 容器，并启动和部署 scopewebdemo Web 应用了。每次打开浏览器访问首页或通过“新增”按钮提交 session 信息时，新的 nameList 和 valueList 实例便会被创建，通过浏览控制台日志可以确认这一点。这些实例都是针对各个 HTTP 请求的，因此应用（用户）可以放心地修改它们的

内部状态，这些状态的修改并不波及到其他用户。

为了证实 session 作用范围的使用，开发者只需要将 request 修改为 session。只要用户不关闭浏览器，不去单击“失效当前 session”按钮，不管用户完成“新增”操作次数多少，新的 nameList 和 valueList 实例只会被创建一次。

在 Web 应用中，开发者经常要使用到处于 request 和 session 作用范围的受管 Bean。这类对象的生命周期不用开发者管理，而且比在 JSP 中声明的 JavaBean 更具魅力，因为它们可以使用到 Spring IoC 容器。

在本节最后，同开发者一起结合下图来大致研究这三种作用范围的实现机理。



图表 19 三种 scope 支持的内部实现机理

当 Java EE 容器接收到客户的 Web 请求时，注册到 Web 应用中的 ServletRequestListener 监听器或 Filter 过滤器会将 HTTP 请求中的相关信息记录到同线程绑定在一起的 ThreadLocal 属性中。随后，Java EE 容器需要同 ApplicationContext 容器交互，比如使用其中的受管 Bean。由于某些受管 Bean 的作用范围为 request 和 session，因此 Spring Framework 提供的 AOP 框架需要介入进来。AOP 框架将控制这类<bean/>实例的创建工作，包括将业务操作请求委派给具体受管 Bean 的业务方法。Spring AOP 响应 HTTP 请求（含有后端方法调

用）期间，可能会涉及到 request 或 HttpSession 信息的修改，而所发生的修改内容也会保存到同线程绑定的 ThreadLocal 属性中。当 Java EE 容器处理完客户的 HTTP 请求后，监听器或过滤器要负责 ThreadLocal 信息的清理工作，比如解除对 Request 对象的引用、同步 ThreadLocal 信息到 session 中。经过上述过程，HttpResponse 响应结果便分发给最终客户了。这一过程都是建立在标准 Java EE（Java）平台技术（比如 ThreadLocal、Filter、Servlet 2.4 引入 ServletRequestListener）之上的，因此对 Web 层使用的展示技术没有任何限制。

2.5 将 DI 容器宿主到 Web 容器中

从企业应用的形式看，大部分都是 Web 应用，而不是单独桌面应用，也不是 Java Applet。在单独桌面应用中，开发者需要手工管理并创建 DI 容器。相比之下，在 Web 应用中，Spring Framework 能够自动管理 DI 容器。本节将围绕 Web 应用是如何使用 DI 容器而展开论述，最后还将介绍消息资源的国际化和本地化支持。

2.5.1 往 Web 应用中加载 DI 容器

为成功管理 BeanFactory 和 ApplicationContext 容器，并将它们宿主在 Web 容器中，开发者需要完成如下一些编程工作。

首先，负责实例化 IoC 容器，并存储到 Web 应用中的某个位置。位置的选择对于 IoC 容器的后续使用非常关键。

其次，应用需要使用 IoC 容器时，开发者需要提供某种策略，使得 DI 容器能够同应用进行交互。

最后，不需要使用 ApplicationContext 时，开发者需要负责销毁它。而且，在销毁 IoC 容器前，务必要将一些受管 Bean 持有的敏感资源释放掉，比如 RDBMS 数据库连接、JMX 服务器连接、等等。

不幸的是，这些工作无疑给开发者加重了工作负担，而且不利于分离关注（Separation of Concerns，SoC）。开发者应该只负责完成自身的开发任务，而其他工作丢给 Spring Framework 自身去完成。所幸，Spring Framework 内置的

ContextLoaderListene 辅助类成功解决了 DI 容器的实例化和销毁工作，而 WebApplicationContextUtils 实用类使得应用能够访问到宿主在 Web 应用中的 DI 容器。

如果目标 Web 容器支持 ServletContextListener，则只需要将如下类似配置放置在/WEB-INF/web.xml 中。如果没有指定 contextConfigLocation 上下文参数，则默认时 ContextLoaderListener 会试图装载/WEB-INF/applicationContext.xml 配置文件。

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/classes/applicationContext.xml</param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

Web 容器加载 Spring Framework 使能 Web 应用时，ContextLoaderListener 会读取 contextConfigLocation 指定的 XML 配置信息，并试图加载这些配置信息构成的 DI 容器。如果开发者未指定，则 Spring Framework 会在默认位置（/WEB-INF/）查找 applicationContext.xml 文件。当 Web 容器停止或卸载 Spring 使能 Web 应用时，ContextLoaderListener 会优雅地完成 DI 容器的销毁工作。

2.5.2 复合多个配置文件

在大型项目中，开发者需要管理成千上万个受管 Bean。其中的一些对象有依赖关系，而另一些又几乎没有关系。如今，分层架构是主流设计技术，比如将应用分成展示层、服务层和 DAO 层。如果将成千上万个受管 Bean 进行分门别类，从而达到分层的目的，则受管 Bean 的管理将会变得更轻松。Spring Framework 允许开发者将不同受管 Bean 放置在不同的 XML 配置文件中。

上述 scopewebdemo 项目仅定义单个 applicationContext.xml 文件。为了检验 Spring Framework 对多个配置文件的支持能力，可以将 applicationContext.xml 中定义的受管 Bean 分别配置在多个不同的 XML 文件中，比如 applicationContext.xml 配置文件中持有 nameList 和 valueList 受管 Bean 定义，而 sessioncontent.xml 持有 sessionContent 受管 Bean 定义。为了使用这些 XML 文件

集合，下面来给出两种可行的解决方案。

其一，调整 web.xml 中的 contextConfigLocation 上下文参数取值，从而引用到这些 XML 文件集合，示例配置如下。文件间可以用空格、逗号、分号等隔开，并支持回车键的使用。Spring Framework 会将这些文件集合集结到一起，比如 applicationContext.xml 和 sessioncontent.xml，并形成单个 DI 容器，最终达到与原先单个 applicationContext.xml 相同的效果。

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/classes/applicationContext.xml,/WEB-INF/classes/sessioncontent.xml
    </param-value>
</context-param>
```

其二，新的 applicationContext.xml 中使用<import/>元素，从而引用到 sessioncontent.xml 文件，示例配置如下。然后，web.xml 中的 contextConfigLocation 上下文参数取值只需包括 applicationContext.xml 文件。开发者可以在任何类型的 Spring Framework 使能应用中启用<import/>元素。

```
<import resource="sessioncontent.xml"/>
```

开发实际项目期间，为避免在运行期出现人为错误，开发者最好在开发阶段使用 Spring IDE（或 STS）来保证不同 XML 文件中受管 Bean 配置的正确性和一致性。

2.5.3 于 Web 应用中操控 DI 容器

Spring Framework 内置的 WebApplicationContextUtils 实用类能够定位到 ContextLoaderListener 存储在 Web 应用中的 IoC 容器，进而访问到宿主在其中的受管 Bean。WebApplicationContextUtils 定义如下，WebApplicationContext 继承于 ApplicationContext。

```
public abstract class WebApplicationContextUtils {

    public static WebApplicationContext getRequiredWebApplicationContext(
            ServletContext sc) throws IllegalStateException {
    }

    public static WebApplicationContext getWebApplicationContext(
            ServletContext sc) {
    }
```

```
    public static WebApplicationContext getWebApplicationContext(
            ServletContext sc, String attrName){
    }

}
```

WebApplicationContextUtils 实用类中的 getRequiredWebApplicationContext()静态方法会去 ServletContext 实例中查找 IoC 容器是否被注册过，如果被注册，则返回给应用；如果没有注册，则会抛出异常。如果不希望出现异常，则还可以使用 getWebApplicationContext()静态方法。在 scopewebdemo 示例应用中，SessionProcess 的 processSessionContent()方法便使用到这一实用类，相关代码摘录如下。

```
public void processSessionContent(HttpSession hs) {
    if (hs == null)
        return;
    ServletContext sc = hs.getServletContext();
    ApplicationContext ac = WebApplicationContextUtils.getRequiredWebApplicationContext(sc);

    ISessionContent sessionContent = (ISessionContent) ac.getBean("sessionContent");

    ......

    sessionContent.processSessionContent(map);
}
```

注意，WebApplicationContextUtils 暴露了两个 getWebApplicationContext()方法，getWebApplicationContext(ServletContext sc)方法会去默认位置查找 IoC 容器，相比之下，getWebApplicationContext(ServletContext sc, String attrName)方法能够查找到注册在 ServletContext 中的任意 IoC 容器。

无论 Web 层采用了何种 Web 视图（框架）技术，它们都需要去 ServletContext 中查找 IoC 容器，而 WebApplicationContextUtils 为开发者提供了一种较好的选择。

2.5.4 国际化和本地化消息资源

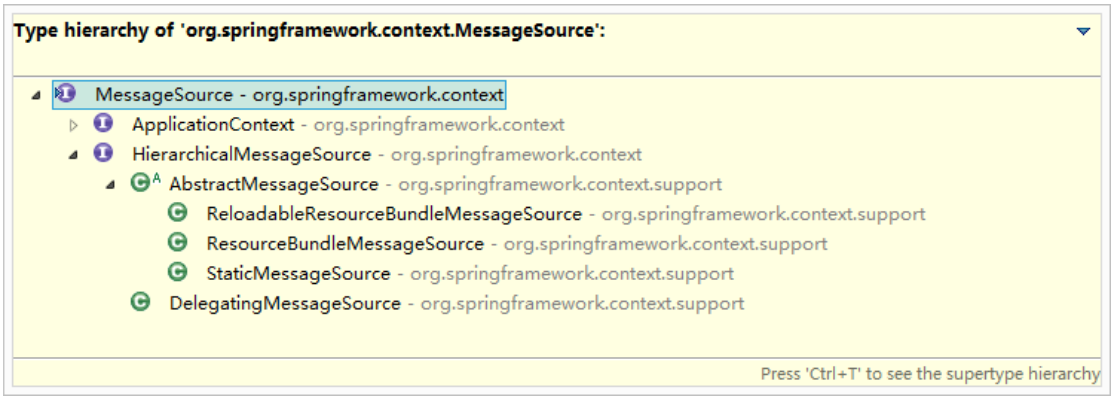
开发 Tapesty 使能 Web 应用期间，开发者只需要在存储模板文件（即.html 和.page）的同一目录放置 Xxx.properties 及其他本地化版本（比如，Xxx_zh_CN.properties）的属性文件（消息资源），Tapestry 4.x/5.x 便能够在运行

时智能地从相应的属性文件中获得本地化消息。类似地，Spring Framework 框架也提供了一流的国际化和本地化消息资源集成支持。它们都是基于 java.util.Locale 和 ResourceBundle 实现的，Java 本身对国际化消息提供了很好的基础支持。

ApplicationContext 继承的 org.springframework.context.MessageSource 策略接口是整个国际化消息支持的基础。如下给出了 MessageSource 接口定义的方法。通过传入消息键、参数、默认消息、区域，getMessage()则会返回消息键对应的取值给调用者。

```
public interface MessageSource {

    String getMessage(String code, Object[] args, String defaultMessage, Locale locale);

    String getMessage(String code, Object[] args, Locale locale) throws NoSuchMessageException;

    String getMessage(MessageSourceResolvable resolvable, Locale locale) throws NoSuchMessageException;

}
```

下图展示了 Spring Framework 提供的 MessageSource 继承链。其中，开发者可以在应用中操作 StaticMessageSource 实现，并动态构建消息内容，企业应用很少使用到它，它主要用于测试目的。ResourceBundleMessageSource 是使用最频繁的实现之一。在运行期，如果需要动态维护消息内容，则开发者还可以使用 ReloadableResourceBundleMessageSource。很明显，上述三种实现都继承于 HierarchicalMessageSource 抽象类，因此它们都支持消息的分层管理。



图表 20 MessageSource 接口及其实现

位于 iocdemo 项目中的 MessageSourceDemo 应用展示了相关知识点的使用。

比如，为了借助 ResourceBundleMessageSource 访问底层消息，下面给出了示例配置（摘自 messagesource.xml）。在初始化 IoC 容器期间，ApplicationContext 会在已注册的受管 Bean 集合中查找名字为“messageSource”的 MessageSource，并完成相关的初始化工作。

```
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename">
                <value>test/message</value>
        </property>
        <property name="parentMessageSource" ref="parentMessageSource"/>
</bean>
```

注意，basename 用于指定.properties 属性文件所在的 classpath。如果存在若干属性文件，则可以使用 basenames 属性（String[]）；parentMessageSource 用于指定父 MessageSource 实现。在 messageSource 定义的属性文件中找不到所需要的属性键时，Spring Framework 会去定义的父实现中查找。父 MessageSource 实现的具体配置见如下片断（摘自 messagesource.xml）。

```
<bean id="parentMessageSource"
        class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
        <property name="basename">
                <value>parenthelloworld</value>
        </property>
        <property name="cacheSeconds" value="10"/>
</bean>
```

由于 parentMessageSource 允许动态修改消息内容，所以它还提供了 cacheSeconds 属性（单位秒）。在 10 秒过后，它会失效缓存中的属性取值，而重新去属性文件中读取相应的内容。当应用在 MessageSource 指定的属性文件中查找不到需要的属性值对时，Spring Framework 会抛出异常。

下面给出了 MessageSource 的各种用法（摘自 MessageSourceDemo.java）。

```
//访问 message_en.properties 中的消息
log.info(aac.getMessage("helloworld", null, Locale.ENGLISH));
//访问 message_zh_CN.properties 中的消息
log.info(aac.getMessage("helloworld", null, Locale.CHINA));
//访问 message_en.properties 中的消息，并传入参数
log.info(aac.getMessage("welcome", new Object[]{"worldheart"} , Locale.ENGLISH));
//访问 message_zh_CN.properties 中的消息，并传入参数
log.info(aac.getMessage("welcome", new Object[]{"访客"}, Locale.CHINA));

//访问 parenthelloworld_en.properties 中的消息，使用父 MessageSource
log.info(aac.getMessage("parenthelloworld", null, Locale.ENGLISH));
```

```
//访问 message_zh_CN.properties 中的消息。由于 hw 键不存在，因此抛出异常
log.info(aac.getMessage("hw", null, Locale.CHINA));
```

消息资源的使用面非常广，比如 Java 桌面应用和 Web 应用都可以使用它生成界面的提示信息。许多著名开源项目使用到 MessageSource。比如 Spring Security 安全性框架使用它来管理认证和授权提示信息。我们在 JA-SIG CAS 4.x 的源代码中也能够浏览到相关应用场景。借助 JDK 5.0+提供的 native2ascii 实用工具（位于%JAVA_HOME%\bin 目录），开发者可以生成 Xxx_zh_CN.properties 文件。

2.6 外在化配置应用参数

在开发企业应用期间，或者在将企业应用部署到生产环境时，应用依赖的很多参数信息往往需要经常调整，比如 LDAP 连接、RDBMS JDBC 连接信息。此时，我们将这类信息进行外在化管理显得格外重要。Spring Framework 内置了 PropertyPlaceholderConfigurer 和 PropertyOverrideConfigurer 对象，它们正是担负外在化配置应用参数的重任。

本节内容将结合 beanfactorypostprocessordemo 项目展开对它们的讨论。

2.6.1 <context:property-placeholder/>元素

PropertyPlaceholderConfigurer 实现了 BeanFactoryPostProcessor 接口，它能够将<bean/>中的属性值进行外在化管理。开发者可以提供单独的属性文件来管理相关属性。比如，存在如下属性文件（摘自 userinfo.properties）。

```
db.username=scott
db.password=tiger
```

如下内容摘自 propertyplaceholderconfigurer.xml。正常情况下，在 userInfo 的定义中不会出现${db.username}、${db.password}等类似信息，这里采用 PropertyPlaceholderConfigurer 管理 username 和 password 属性的取值。DI 容器实例化 userInfo 前，PropertyPlaceholderConfigurer 会修改 userInfo 的元数据信息（<bean/>定义），它会用 userinfo.properties 中 db.username 对应的 scott 值替换${db.username}、db.password 对应的 tiger 值替换${db.password}。最终，DI 容器在实例化 userInfo 时，UserInfo 便会得到新的属性值了，而不是${db.username}、

${db.password}等类似信息。

```
<bean id="propertyPlaceholderConfigurer"
        class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
                <list>
                        <value>userinfo.properties</value>
                </list>
        </property>
</bean>

<bean name="userInfo" class="test.UserInfo">
   <property name="username" value="${db.username}"/>
   <property name="password" value="${db.password}"/>
</bean>
```

通过运行并分析 PropertyPlaceholderConfigurerDemo 示例应用，开发者能够深入理解 PropertyPlaceholderConfigurer。为简化 PropertyPlaceholderConfigurer 的使用，Spring 提供了<context:property-placeholder/>元素。下面给出了配置示例，当启用它后，开发者便不用配置 PropertyPlaceholderConfigurer 对象了。

```
<context:property-placeholder location="userinfo.properties"/>
```

PropertyPlaceholderConfigurer 对象内置的功能非常丰富，如果它未找到${xxx}中定义的 xxx 键，它还会去 JVM 系统属性（System.getProperty()）和环境变量（System.getenv()）中寻找。开发者通过控制 systemPropertiesMode 和 searchSystemEnvironment 属性取值能够定制 PropertyPlaceholderConfigurer 相关行为。

2.6.2 <context:property-override/>元素

PropertyOverrideConfigurer 会从属性文件读取属性值，以替代受管 Bean 的定义。比如，PropertyOverrideConfigurerDemo 客户应用使用到如下属性文件（摘自 ui.properties）。其中，userInfo 是 propertyoverrideconfigurer.xml 中受管 Bean 的 name（惟一标识符）；username 和 password 分别是 UserInfo 受管 Bean 的属性名。

```
userInfo.username=scott
userInfo.password=tiger
```

下面给出了 propertyoverrideconfigurer.xml 配置示例。DI 容器实例化 userInfo 受管 Bean 前，PropertyOverrideConfigurer 会去 ui.properties 中检查是否存在属性值需要覆盖。如果有，则它会实施覆盖操作；如果未找到，则它会保留 userInfo

受管 Bean 中对应属性指定的属性值。其中，ignoreResourceNotFound 属性含义为，在找不到 ui.properties 属性文件时不会有异常抛出。

```
<bean id="propertyOverrideConfigurer"
        class="org.springframework.beans.factory.config.PropertyOverrideConfigurer">
    <property name="location">
        <value>ui.properties</value>
    </property>
    <property name="ignoreResourceNotFound" value="true"/>
</bean>

<bean name="userInfo" class="test.UserInfo">
      <property name="username" value="un"/>
      <property name="password" value="pw"/>
</bean>
```

为简化 PropertyOverrideConfigurer 对象的使用，Spring Framework 提供了<context:property-override/>元素。下面给出了配置示例，当启用它后，开发者便不用配置 PropertyOverrideConfigurer 对象了。

```
<context:property-override location="ui.properties"/>
```

有关 IoC 容器的分层管理

事实上，IoC 容器支持分层管理，其具体含义是，在同一 JVM 中存在多个 IoC 容器，这些容器之间存在父子关系。在很多场合，企业应用需要使用这一 IoC 特性。比如，同时使用 EJB 3.x 和 Spring 4.2 开发 EJB 3.x 应用、开发复杂 Web 和 Java EE 应用。

PropertyPlaceholderConfigurer 和 PropertyOverrideConfigurer 属于 BeanFactoryPostProcessor 类型的对象。BeanFactoryPostProcessor（包括 BeanPostProcessor）的作用域是单个 IoC 容器，因此不同 IoC 容器只能够管理自身的 BeanFactoryPostProcessor（包括 BeanPostProcessor）。处于层级中的 IoC 容器也要遵循这一规则，即在 BeanFactoryPostProcessor（包括 BeanPostProcessor）看来各层定义的 IoC 容器是独立的。

另外，开发者不要在定义 BeanFactoryPostProcessor（包括 BeanPostProcessor）时设置“lazy-init="true"”（lazy-init 属性的具体介绍见本章后续章节）。否则，这些 BeanFactoryPostProcessor（包括 BeanPostProcessor）将不会被实例化，从而不会被激活。

2.7 值得重视的若干 DI 特性

本节内容将针对 Spring 内置的一些 DI 特性进行阐述。

2.7.1 depends-on 属性

开发者通过<ref/>元素、ref 属性等方式能够完成协作者的依赖注入，此时，这些受管 Bean 间的关系非常清晰。然而，在一些场合，受管 Bean 之间的关系并不能够通过这些方式进行指定，但应用又要求 DI 容器在实例化它们时能根据开发者指定的顺序进行。比如，Eclipse dependsondemo 项目中 DependsOnDemo 应用使用了如下 dependson1.xml 配置文件。由于它们之间没有引用关系，因此容器通常会根据定义的顺序来依次实例化它们。如果 testBean1 要求容器先实例化 testBean2、testBean3，则开发者需要使用<bean/>元素内置的 depends-on 属性。如果存在多个依赖对象，则中间需要用逗号、分号或空格隔开。

```
<bean id="testBean1" class="test.TestBean1" depends-on="testBean2,testBean3"/>

<bean id="testBean2" class="test.TestBean2"/>

<bean id="testBean3" class="test.TestBean3"/>
```

通过<ref/>元素或 ref 属性等方式指定受管 Bean 间的协作关系时，开发者是否就不需要启用 depends-on 属性呢？不一定，因为不同应用的设计策略和实现都不一样。比如 dependson2.xml 配置文件定义了如下内容。由于 testBean4 定义在 testBean5 之前，DI 容器会先试图完成 testBean4 的实例化工作。在实施设值注入过程中，由于 testBean4 引用到 testBean5，此时才会去完成 testBean5 的实例化工作。因此，<ref/>元素只是指定了受管 Bean 之间的依赖关系，而它并没有告知 DI 容器是依赖对象（testBean4）先试图完成实例化，还是被依赖对象（协作者，testBean5）先试图完成实例化。但是有一点我们可以确定，即协作者肯定比依赖对象先完成实例化工作（包括依赖注入的完成）。

```
<bean id="testBean4" class="test.TestBean4">
    <property name="testBean5" ref="testBean5"/>
</bean>

<bean id="testBean5" class="test.TestBean5"/>
```

由于 TestBean4 和 TestBean5 间的特殊关系，要求 DI 容器先试图完成 testBean5 的初始化工作，如下所示。因此，上述配置肯定会触发 IllegalArgumentException 异常的出现。

```
//借助 System.setProperty 设置了属性
public TestBean5(){
```

```
        System.setProperty("dependson","testBean5");
}

//借助 System.getProperty 读取属性
public TestBean4(){
        String proper = System.getProperty("dependson");
        Assert.hasText(proper);
}
```

为了解决这一问题，开发者有两种简单解决方案可供选用。

其一，将 testBean5 定义在 testBean4 之前，示例如下。此时，容器会先试图完成 testBean5 的实例化工作。使用这种办法倒是解决问题了，但日后在开发过程中 testBean5 被换了个位置，则后果不堪设想，毕竟这是隐式表达它们之间的依赖关系，而且应用也不应该依赖这种不成文的特性。

```
<bean id="testBean5" class="test.TestBean5"/>

<bean id="testBean4" class="test.TestBean4">
        <property name="testBean5" ref="testBean5"/>
</bean>
```

其二，启用 depends-on。这将从根本上保证 DI 容器会先试图完成 testBean5 的实例化工作，如下所示。

```
<bean id="testBean4" class="test.TestBean4" depends-on="testBean5">
        <property name="testBean5" ref="testBean5"/>
</bean>

<bean id="testBean5" class="test.TestBean5"/>
```

2.7.2 别名（Alias）

如同叫别人小名一样，我们也可以对受管 Bean 取“小名”。默认时，我们需要在定义<bean/>元素时指定 id 或 name 属性。由于 id 属性是 XML 标准行为，因此一般建议开发者通过给定 id 属性，以唯一标识受管 Bean。通常，如果 id 和 name 属性都没有指定，Spring 会为当前受管 Bean 生成一默认 id 值，即类的全限定名（FQN）。

某些时候，我们需要借助<alias/>元素，给特定受管 Bean 取别名，比如为了达到同具体<bean/>解耦的目的。下面给出了配置示例（摘自 iocdemo 项目中的 alias.xml），以后无论是通过“testBean”，还是通过“tb”别名查找到的受管 Bean 始终是同一个。

```
<bean class="test.TestBean"/>

<bean id="testBean" class="test.TestBean"/>

<!-- name 指定源 POJO，alias 指定别名 -->
<alias name="testBean" alias="tb"/>

<bean name="tB" class="test.TestBean"/>
```

在<alias/>元素中，name 属性用于指定被取别名的目标受管 Bean 的 id 或 name，而 alias 属性用于指定别名。

AliasDemo 示例应用展示了如何通过名字获得上述受管 Bean 的过程，具体如下。

```
ITestBean tb1 = (ITestBean) factory.getBean("test.TestBean");
ITestBean testBean = (ITestBean) factory.getBean("testBean");
ITestBean tB = (ITestBean) factory.getBean("tB");
ITestBean tb = (ITestBean) factory.getBean("tb");
```

2.7.3 <p/>命名空间

开发者只需往 Spring DI 配置文件中新增如下粗体内容，<p/>命名空间便可启用了。

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

比如，某 PropertyPlaceholderConfigurer 对象的配置（pnamespace.xml）如下，摘自 beanfactorypostprocessordemo 项目。此时，locations 属性引用到另一<util:list/>对象，而 order 属性只是一简单 int 属性。

```
<bean id="propertyPlaceholderConfigurer" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations" ref="locations"/>
    <property name="order" value="1"/>
</bean>

<util:list id="locations">
    <value>userinfo.properties</value>
</util:list>
```

启用<p/>命名空间后，我们给出了如下等效配置。“p:*-ref”用于引用其他受管 Bean，这里的*表示属性名，比如 locations。对于简单属性，则直接启用属性

名即可，即“p:*”，比如 order。

```
<bean id="propertyPlaceholderConfigurer" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
      p:locations-ref="locations" p:order="1"/>

<util:list id="locations">
      <value>userinfo.properties</value>
</util:list>
```

开发者可以去运行 PNamespaceDemo 示例应用，感受一下<p/>命名空间的魅力。<p/>命名空间的使用使得配置文件变得简洁。而且，这一命名较特殊，即它不存在对应的 XSD（XML Schema）。

2.7.4 <c/>命名空间

开发者只需往 Spring DI 配置文件中新增如下粗体内容，<c/>命名空间便可启用了。

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

某 FileHelloStrImpl 对象的配置（cnamespace.xml）如下，摘自 iocdemo 项目。此时，通过构建器注入完成了 FileHelloStrImpl 对象的初始化工作。其中一参数是字符串，而另一 testBean 参数引用到 testBean 对象。

```
<bean name="fileHello1" class="test.FileHelloStrImpl">
      <constructor-arg value="helloworld.properties"/>
      <constructor-arg ref="testBean"/>
</bean>

<bean id="testBean" class="test.TestBean"/>
```

启用<c/>命名空间后，我们给出了如下等效配置。“c:*-ref”用于引用其他受管 Bean，这里的*表示属性名，比如 testBean。对于简单属性，则直接启用属性名即可，即“c:*”，比如 propfilename。可以看出，<c/>类似于<p/>命名空间，前者实施构建器注入，而后者实施设值注入。

```
<bean name="fileHello2" class="test.FileHelloStrImpl"
      c:propfilename="helloworld.properties" c:testBean-ref="testBean"/>
```

或者，也可以以如下形式存在。

```
<bean name="fileHello3" class="test.FileHelloStrImpl"
      c:_0="helloworld.properties" c:_1-ref="testBean"/>
```

开发者可以去运行 CNamespaceDemo 示例应用，感受一下<c/>命名空间的魅力。<c/>命名空间的使用使得配置文件变得简洁。而且，这一命名较特殊，即它不存在对应的 XSD（XML Schema）。

2.7.5 抽象和子 Bean

在面向对象领域，对象之间存在继承关系。在大型 Spring Framework 使能应用中，会存在大量受管 Bean，比如不同子系统都会定义完成类似功能的受管 Bean，它们分别用于事务、安全等目的。如果仔细分析这些受管 Bean，则会发现它们之间会存在很大的相似性。开发者能否在 Spring Framework 使能应用中采用 OO 的继承机制来管理受管 Bean 呢？

通过定义抽象 Bean 和子 Bean，开发者能够简化对<bean/>元素的管理。我们可以将公共内容放置在抽象 Bean 中。此时，抽象 Bean 仅仅是担负模板功能，DI 容器并不会去实例化它。子 Bean 继承于相应的抽象 Bean（即父 Bean），开发者可以通过<bean/>中提供的 parent 属性完成继承关系的映射。子 Bean 能够重用抽象 Bean 中定义的内容，而且还能够“重载”它们，甚至重载的次数没有限制。

Spring Framework 允许开发者以显式和隐式方式声明抽象 Bean。显式，指在定义抽象 Bean 时给定 class 属性；而隐式并没有指定 class 属性。Eclipse iocdemo 项目中的 TestAbstractBeanDemo 应用演示了抽象 Bean 的若干用法。下面给出的示例配置中（摘自 testabstractbean.xml），childBean1 继承了 abstractBean1 定义的 username 属性。通过给定 abstract 属性可以将<bean/>元素指定的 XML 配置声明为抽象 Bean（置 abstract="true"）。子 Bean 通过指定 parent 属性便可以引用到抽象 Bean。

```
<bean id="abstractBean1" class="test.TestAbstractBean" abstract="true">
      <property name="username" value="un"/>
</bean>

<bean id="childBean1" class="test.TestAbstractBean" parent="abstractBean1">
      <property name="password" value="pw"/>
</bean>
```

如果采用隐式方式，则可以给出等效于上述配置的如下配置片断。

```
<bean id="abstractBean4" abstract="true">
        <property name="username" value="un"/>
</bean>

<bean id="childBean3" class="test.TestAbstractBean" parent="abstractBean4">
        <property name="password" value="pw"/>
</bean>
```

下面的配置片断展示了多重“继承”的使用，abstractBean2 和 abstractBean3 同为抽象 Bean，而且 abstractBean3 继承于 abstractBean2。abstractBean3 覆盖了 abstractBean2 指定的 username 属性。

```
<bean id="abstractBean2" class="test.TestAbstractBean" abstract="true">
        <property name="username" value="un"/>
        <property name="sex" value="male"/>
</bean>

<bean id="abstractBean3" class="test.TestAbstractBean" parent="abstractBean2" abstract="true">
        <property name="username" value="username"/>
</bean>

<bean id="childBean2" class="test.TestAbstractBean" parent="abstractBean3">
        <property name="password" value="pw"/>
</bean>
```

如果采用隐式方式，则可以给出等效于上述配置的如下配置片断。

```
<bean id="abstractBean5" abstract="true">
        <property name="username" value="un"/>
        <property name="sex" value="male"/>
</bean>

<bean id="abstractBean6" parent="abstractBean5" abstract="true">
        <property name="username" value="username"/>
</bean>

<bean id="childBean4" class="test.TestAbstractBean" parent="abstractBean6">
        <property name="password" value="pw"/>
</bean>
```

2.8 <util/>命名空间

事情的发展总是存在一曲折、前进的过程。当 Spring Framework 刚出现时，开发者可以使用<list/>、<map/>、<set/>等元素定义集合，然而这些集合不能够在不同受管 Bean 间进行复用。尽管开发者可以采用抽象 Bean 机制实现复用，但实在不怎么优雅。与此同时，开发者借助 ListFactoryBean、MapFactoryBean 和 SetFactoryBean 等对象能够定义出可供复用的集合。然而，这也不是很友好的

做法。再后来，<util/>命名空间被 Spring Framework 引入，这才使得集合的定义变得简单。本节将结合 utilnsdemo 项目，并围绕这一命名空间内置的各元素展开。

2.8.1 <util:constant/>元素

通常，Java 类或对象中总是存在静态和非静态属性，比如某 HelloWorld 类存在如下属性定义。

```
public static final String hwStatic = "hw static";

public String hw = "hw";
```

一些场合希望将这些属性取值作为受管 Bean，FieldRetrievingFactoryBean 能帮助开发者达成这一目的。下面展示了相应的配置示例（摘自 utilnsdemo 项目中的 constant.xml）。

```
<bean id="helloWorld" class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
    <property name="staticField" value="test.HelloWorld.hwStatic"/>
</bean>
```

此时，helloWorld 受管 Bean 将是一 java.lang.String 类型的对象，而且取值是“hw static”。开发者也可将 test.HelloWorld 的 hwStatic 属性指定为<bean/>的 id，下面给出了等效配置，但其他受管 Bean 一旦要引用它，则要使用冗长的 id。

```
<bean id="test.HelloWorld.hwStatic"  class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
```

为简化 FieldRetrievingFactoryBean 的使用，开发者可以启用<util:constant/>元素，下面给出了配置示例。

```
<util:constant id="hwConstant" static-field="test.HelloWorld.hwStatic"/>
```

另外，FieldRetrievingFactoryBean 也可以引用非静态属性，下面展示了相应的用法。

```
<bean id="hw" class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
    <property name="targetObject">
        <bean class="test.HelloWorld"/>
    </property>
    <property name="targetField" value="hw"/>
</bean>
```

关于 MethodInvokingFactoryBean

类似于 FieldRetrievingFactoryBean，它能够将静态或非静态方法的返回值作为受管 Bean。有兴

趣的开发者可以去了解它的使用。

2.8.2 <util:property-path/>元素

PropertyPathFactoryBean 能够依据属性路径评估出其他受管 Bean 的属性值，并将这一属性值作为当前的受管 Bean。下面给出了其使用示例，摘自 propertypath.xml。

```
<bean id="propertyPath" class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
        <property name="targetBeanName">
                <idref bean="helloWorld"/>
        </property>
        <property name="propertyPath" value="hello"/>
</bean>

<bean id="helloWorld" class="test.HelloWorld">
        <property name="hello" value="hi"/>
</bean>
```

此时，targetBeanName 属性用于指定受管 Bean 的惟一标识符，而 propertyPath 用于指定属性名。为简化 PropertyPathFactoryBean 的使用，开发者可以启用<util:property-path/>元素，示例如下。其中，path 属性由受管 Bean 的惟一标识符、属性名构成。

```
<util:property-path id="property-path" path="helloWorld.hello"/>

<bean id="helloWorld" class="test.HelloWorld">
        <property name="hello" value="hi"/>
</bean>
```

下面给出了 PropertyPathFactoryBean 的另一用法，这时，它直接引用到目标对象，即内部 Bean。

```
<bean id="propertyPath" class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
        <property name="targetObject">
                <bean class="test.HelloWorld">
                        <property name="hello" value="hi"/>
                </bean>
        </property>
        <property name="propertyPath" value="hello"/>
</bean>
```

2.8.3 <util:properties/>元素

借助<props/>元素，开发者能够定义 java.util.Properties 集合。下面摘录了 properties.xml 中的配置信息。

```
<bean id="abstractCollectionBean" abstract="true">
    <property name="properties">
        <props>
            <prop key="propKey1">propValue1</prop>
            <prop key="propKey2">propValue2</prop>
            <prop key="propKey3">propValue3</prop>
        </props>
    </property>
</bean>

<bean id="collectionBean" class="test.CollectionBean" parent="abstractCollectionBean">
    <property name="properties">
        <props merge="true">
            <prop key="propKey1">propValue1Override</prop>
            <prop key="propKey4">propValue4</prop>
        </props>
    </property>
</bean>
```

通过给定 merge 属性，可以合并抽象 Bean 和子 Bean 中定义的取值。如果它们之间存在重复内容，则以子 Bean 中定义的为准。如果抽象 Bean 定义的取值在子 Bean 中找不到，则它会合并到子 Bean 的 Properties 属性取值集合中（比如，键 propKey2、propKey3 对应的值对将合并到 collectionBean 定义的 properties 取值中）。

不幸的是，在某种程度上，<props/>元素只是一种内部 Bean，同一<props/>元素不能够起到复用目的。相比之下，借用 PropertiesFactoryBean 对象，开发者能够构建出可供复用的 Properties 对象。下面展示了配置示例。

```
<bean id="properties"
    class="org.springframework.beans.factory.config.PropertiesFactoryBean">
    <property name="properties">
        <props>
            <prop key="propKey1">propValue1</prop>
            <prop key="propKey2">propValue2</prop>
            <prop key="propKey3">propValue3</prop>
        </props>
    </property>
</bean>
```

另外，PropertiesFactoryBean 还能够基于属性文件构造出 Properties 对象，下面给出了配置示例，location 属性用于指定属性文件的位置信息。“classpath:”表明，将从类路径上查找并装载 commons-logging.properties 属性文件。

```
<bean id="commonsLogging" class="org.springframework.beans.factory.config.PropertiesFactoryBean">
    <property name="location" value="classpath:commons-logging.properties"/>
</bean>
```

为简化 PropertiesFactoryBean 的使用，开发者可以启用<util:properties/>元素，它能够基于属性文件构造出 Properties 对象，示例配置如下。

```
<util:properties id="commonsLoggingProperties" location="classpath:commons-logging.properties"/>
```

2.8.4 <util:list/>元素

借助<list/>元素，开发者能够定义 java.util.List 集合。下面摘录了 list.xml 中的配置信息。

```
<bean id="abstractCollectionBean" abstract="true">
    <property name="list">
        <list>
            <value>list1</value>
            <value>list2</value>
        </list>
    </property>
</bean>

<bean id="collectionBean" class="test.CollectionBean" parent="abstractCollectionBean">
    <property name="list">
        <list merge="true" value-type="java.lang.String">
            <value>list1</value>
            <idref local="collectionBean"/>
            <null></null>
        </list>
    </property>
</bean>
```

其中，value-type 属性指定存入 list 的默认 Java 类型。

不幸的是，在某种程度上，<list/>元素只是一种内部 Bean，同一<list/>元素不能够起到复用目的。相比之下，借用 ListFactoryBean 对象，开发者能够构建出可供复用的 List 对象。下面展示了配置示例，此时，targetListClass 属性用于指定 List 具体类型，而 sourceList 属性用于指定 List 中的具体内容。

```
<bean id="list" class="org.springframework.beans.factory.config.ListFactoryBean">
    <property name="targetListClass" value="java.util.ArrayList"/>
    <property name="sourceList">
        <list>
            <value>first</value>
            <value>second</value>
            <value>three</value>
            <value>four</value>
        </list>
    </property>
</bean>
```

为简化 ListFactoryBean 的使用，开发者可以启用<util:list/>元素，示例配置

如下。

```
<util:list id="listUtil" list-class="java.util.ArrayList">
        <value>first</value>
        <value>second</value>
        <value>three</value>
        <value>four</value>
</util:list>
```

2.8.5 <util:map/>元素

借助<map/>元素，开发者能够定义 java.util.Map 集合。下面摘录了 map.xml 中的配置信息。

```
<bean id="abstractCollectionBean" abstract="true">
        <property name="map">
                <map>
                        <entry key="mapKey1" value="mapValue1"/>
                        <entry key="mapKey2" value="mapValue2"/>
                </map>
        </property>
</bean>

<bean id="collectionBean" class="test.CollectionBean" parent="abstractCollectionBean">
        <property name="map">
                <map merge="true" key-type="java.lang.String" value-type="java.lang.String">
                        <entry key="mapKey1" value="mapValue1Override"/>
                        <entry>
                                <key><value>mapKey2</value></key>
                                <value>mapValue2</value>
                        </entry>
                        <entry key="testBean" value-ref="testBean"/>
                </map>
        </property>
</bean>

<bean id="testBean" class="test.TestBean"/>
```

其中，key-type 属性指定键对应的默认 Java 类型、value-type 属性指定值对应的默认 Java 类型。

自从 JDK 5.0 开始，泛型便得到广泛支持。在定义 Map 对象时，开发者可以指定键值分别对应的 Java 类型。一旦开发者往 Map 中存放类型不匹配的 Java 类型时，编译器便会抛出异常。Spring Framework 支持强类型化（typed）Map 定义，它会根据 Java 代码中声明的泛型使用情况，来动态完成类型化转换，下面给出了一配置示例。由于 CollectionBean 定义的 mapTyped 属性为 Map<String, Double> 类型，因此运行期，mapKey1 和 mapKey2 会自动转换成 java.lang.String，而 10.1

和 20 会被转换成相应的 Double 类型。

```
<bean id="collectionBean1" class="test.CollectionBean">
    <property name="mapTyped">
        <map>
            <entry key="mapKey1" value="10.1"/>
            <entry>
                <key><value>mapKey2</value></key>
                <value>20</value>
            </entry>
        </map>
    </property>
</bean>
```

在某种程度上，<map/>元素只是一种内部 Bean，同一<map/>元素不能够起到复用目的。相比之下，借用 MapFactoryBean 对象，开发者能够构建出可供复用的 Map 对象。下面展示了配置示例，此时，targeMapClass 属性用于指定 Map 具体类型，而 sourceMap 属性用于指定 Map 中的具体内容。

```
<bean id="map" class="org.springframework.beans.factory.config.MapFactoryBean">
    <property name="targetMapClass" value="java.util.HashMap"/>
    <property name="sourceMap">
        <map>
            <entry key="1" value="first"/>
            <entry key="2" value="second"/>
            <entry key="3" value="three"/>
            <entry key="4" value="four"/>
        </map>
    </property>
</bean>
```

为简化 MapFactoryBean 的使用，开发者可以启用<util:map/>元素，示例配置如下。

```
<util:map id="mapUtil" map-class="java.util.HashMap">
    <entry key="1" value="first"/>
    <entry key="2" value="second"/>
    <entry key="3" value="three"/>
    <entry key="4" value="four"/>
</util:map>
```

2.8.6 <util:set/>元素

借助<set/>元素，开发者能够定义 java.util.Set 集合。下面摘录了 set.xml 中的配置信息。

```
<bean id="abstractCollectionBean" abstract="true" class="test.CollectionBean">
    <property name="set">
        <set>
```

```
                <value>set1</value>
                <value>set2</value>
            </set>
        </property>
</bean>

<bean id="collectionBean" class="test.CollectionBean" parent="abstractCollectionBean">
        <property name="set">
            <set merge="true" value-type="java.lang.String">
                <value>set1</value>
                <idref bean="abstractCollectionBean"/>
                <null/>
            </set>
        </property>
</bean>
```

其中，value-type 属性指定存入 set 的默认 Java 类型。

不幸的是，在某种程度上，<set/>元素只是一种内部 Bean，同一<set/>元素不能够起到复用目的。相比之下，借用 SetFactoryBean 对象，开发者能够构建出可供复用的 Set 对象。下面展示了配置示例，此时，targetSetClass 属性用于指定 Set 具体类型，而 sourceSet 属性用于指定 Set 中的具体内容。

```
<bean id="set" class="org.springframework.beans.factory.config.SetFactoryBean">
        <property name="targetSetClass" value="java.util.HashSet"/>
        <property name="sourceSet">
            <set>
                <value>first</value>
                <value>second</value>
                <value>three</value>
                <value>four</value>
            </set>
        </property>
</bean>
```

为简化 SetFactoryBean 的使用，开发者可以启用<util:set/>元素，示例配置如下。

```
<util:set id="setUtil" set-class="java.util.HashSet">
        <value>first</value>
        <value>second</value>
        <value>three</value>
        <value>four</value>
</util:set>
```

2.9 回调接口集合及其触发顺序

Spring Framework 内置了用于不同目的的大量回调接口，很多场合都会使用

到它们。使用借助这些回调接口后，往往能够达到事半功倍的效果。一旦目标受管 Bean 实现了回调接口，则当 DI 容器实例化受管 Bean 时，DI 容器会自动调用这些回调接口所定义的方法，进而将相关对象注入进来。最终，受管 Bean 便可使用它们了。本节将结合 iocdemo 项目中的 ApplicationContextCallbackDemo 示例应用（包括其使用的 applicationcontext-callback.xml 配置文件）展开对常用回调接口集合的介绍。大致上，回调接口的触发时机同这里介绍的顺序相吻合。注意，这里并没有和盘托出所有的回调接口，而是重点介绍同开发者密切相关的接口。

2.9.1 BeanNameAware 回调接口

如果受管 Bean 实现了 BeanNameAware 回调接口，则受管 Bean 本身的 id（name）会被 BeanFactory 注入到受管 Bean 中。BeanNameAware 接口的定义如下。

```
public interface BeanNameAware {

	void setBeanName(String name);

}
```

比如，TestBean 实现了 BeanNameAware 回调接口，示例代码如下。

```
public class TestBean implements ITestBean, BeanNameAware {

	private String beanName;

	public void setBeanName(String name) {
		this.beanName = name;
	}

}
```

2.9.2 BeanClassLoaderAware 回调接口

BeanClassLoaderAware 回调接口可让受管 Bean 本身知道，它是由哪一类装载器负责装载的。这一回调接口的定义如下。

```
public interface BeanClassLoaderAware {

	void setBeanClassLoader(ClassLoader classLoader);

}
```

2.9.3 BeanFactoryAware 回调接口

如果当前受管 Bean 实现了 BeanFactoryAware 接口，则运行时它可以使用 BeanFactory 显式查找其依赖的对象。显然，这同 Spring Framework 耦合在一起。BeanFactoryAware 接口定义如下。

```
public interface BeanFactoryAware {

	void setBeanFactory(BeanFactory beanFactory) throws BeansException;

}
```

TestBean 实现了这一接口，示例代码如下。开发者负责实现 setBeanFactory() 方法，而 IoC 容器负责调用它。

```
public class TestBean implements ITestBean, BeanFactoryAware {

	private BeanFactory bf;

	public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
		this.bf = beanFactory;
	}

}
```

作者推荐开发者尽量采用依赖注入完成协作者的注入，而不要采用上述接口。但在一些场合，确实需要在运行时动态查找那些定义为原型的受管 Bean。

2.9.4 ResourceLoaderAware 回调接口

ResourceLoaderAware 回调接口负责将 ResourceLoader 对象注入到当前受管 Bean 实例中，其定义如下。当受管 Bean 获得 ResourceLoader 对象后，它便能够通过它获得各种资源。

```
public interface ResourceLoaderAware {

	void setResourceLoader(ResourceLoader resourceLoader);

}
```

2.9.5 ApplicationEventPublisherAware 回调接口

ApplicationEventPublisherAware 回调接口负责将 ApplicationEventPublisher 对象注入到当前受管 Bean 实例中，以供分发事件使用，其定义如下。

```
public interface ApplicationEventPublisherAware {

    void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher);

}
```

2.9.6 MessageSourceAware 回调接口

MessageSourceAware 回调接口负责将 MessageSource 对象注入到当前受管 Bean 实例中，其定义如下。通过使用 MessageSource 对象，受管 Bean 能够获得国际化和本地化消息支持。

```
public interface MessageSourceAware {

    void setMessageSource(MessageSource messageSource);

}
```

2.9.7 ApplicationContextAware 回调接口

类似于 BeanFactoryAware 回调接口，ApplicationContextAware 使得受管 Bean 能够感知到 IoC 容器的存在，其定义如下。注意，ApplicationContextAware 仅仅适合于 ApplicationContext 容器。

```
public interface ApplicationContextAware {

    void setApplicationContext(ApplicationContext applicationContext) throws BeansException;

}
```

值得注意的是，Spring Framework 还提供了 ApplicationObjectSupport 抽象类来简化 ApplicationContextAware 的使用，受管 Bean 只需要直接继承这一抽象类，而不用再去实现 ApplicationContextAware 回调接口，因为它已实现 ApplicationContextAware，而且还为访问底层 MessageSource 提供了便利。

2.9.8 @PostConstruct 注解

@PostConstruct 是 Java EE 5 引入的注解，Spring Framework 允许开发者在受管 Bean 中使用它。当 DI 容器实例化当前受管 Bean 时，@PostConstruct 注解的方法会被自动触发，从而完成一些初始化工作，示例代码如下。

```
@PostConstruct
public void postConstruct(){
    log.info("调用 postConstruct");
}
```

2.9.9 InitializingBean 回调接口

如果目标受管 Bean 实现了 InitializingBean 回调接口，则当 DI 容器实例化当前受管 Bean 期间，其定义的回调方法会被自动触发，从而完成一些初始化工作，其接口定义如下。

```
public interface InitializingBean {

    void afterPropertiesSet() throws Exception;

}
```

2.9.10<bean/>元素的 init-method 属性

除了启用@PostConstruct 注解、InitializingBean 回调接口，开发者还可以使用<bean/>元素的 init-method 属性，以完成一些初始化工作，示例配置如下。

```
<bean name="testBean" init-method="start" class="test.TestBean"/>
```

此时，init-method 属性用于指定自定义初始化方法，即 start()，示例代码如下。

```
public void start(){
    log.info("调用<bean/>定义的 init-method 方法 start");
}
```

2.9.11@PreDestroy 注解

@PreDestroy 是 Java EE 5 引入的注解，Spring Framework 允许开发者在受管 Bean 中使用它。当 DI 容器销毁当前受管 Bean 实例时，@PreDestroy 注解的方法会被自动触发，从而完成一些资源清理工作，示例代码如下。

```
@PreDestroy
public void preDestroy(){
    log.info("调用 preDestroy");
}
```

如果当前受管 Bean 不是单例，则@PreDestroy 注解应用的方法没有机会执行。

2.9.12DisposableBean 回调接口

如果目标受管 Bean 实现了 DisposableBean 回调接口，则当 DI 容器销毁当前受管 Bean 实例时，其定义的回调方法会被自动触发，从而完成一些资源清理工作，其接口定义如下。

```
public interface DisposableBean {

    void destroy() throws Exception;

}
```

如果当前受管 Bean 不是单例，则 destroy()方法没有机会执行。

2.9.13<bean/>元素的 destroy-method 属性

除了启用@PreDestroy 注解、DisposableBean 回调接口，开发者还可以使用<bean/>元素的 destroy-method 属性，以完成一些资源清理工作，示例配置如下。

```
<bean name="testBean" destroy-method="finish" class="test.TestBean"/>
```

此时，destroy-method 属性用于指定自定义销毁方法，即 finish()，示例代码如下。

```
public void finish(){
    log.info("调用<bean/>定义的 destroy-method 方法 finish");
}
```

如果当前受管 Bean 不是单例，则 destroy-method 属性指定的销毁方法没有机会执行。

注意@Autowired 注解带来的便利

@Autowired 注解不仅能够完成普通受管 Bean 的注入工作，DI 容器中的大量配置元数据对象也同样适合，比如 ApplicationContext、MessageSource 等。当采用@Autowired 注解注入这类对象时，本节介绍的许多回调接口便可以宣布退休了。

2.10 基于 Java 及注解配置 DI 容器

自 JDK 5.0 引入注解特性后，开发社区便积极跟进并挖掘它的潜能。这不，Spring Framework 从 3.0 开始便试图依托 Java 及注解承载 DI 元数据，其目标是彻底抛弃时下最流行的 XML 配置方式，而全面采用 Java 及注解替代它。

下面给出了 Java 版本的 JavaConfigDemo（摘自 javaconfigdemo 项目）。这里不再出现 XML 方面的痕迹，取而代之的是 Annotation。比如，@Bean 同<bean/>元素达到相同的效果，而 AnnotationConfigApplicationContext 同 XmlBeanFactory（或者 ClassPathXmlApplicationContext）的作用也非常类似。

```
@Configuration
public class JavaConfigDemo {

	private static final Log log = LogFactory.getLog(JavaConfigDemo.class);

	@Bean
	public IHelloWorld helloWorld() {
		HelloWorld hw = new HelloWorld();
		hw.setHelloStr(helloStr());
		return hw;
	}

	@Bean
	public IHelloStr helloStr() {
		return new FileHelloStrImpl("helloworld.properties");
	}

	public static void main(String[] args) {
		// 从注解中装载 DI 容器配置元数据
		ApplicationContext context = new AnnotationConfigApplicationContext(JavaConfigDemo.class);

		// 获得受管 Bean
		IHelloWorld hw = context.getBean(IHelloWorld.class);

		// 返回字符串
		log.info(hw.getContent());
	}

}
```

从上述内容看出，依托纯 Java 承载 Spring DI 元数据是没有问题的，从而可以抛弃繁琐的 XML 配置文件。在上述示例中，@Bean 和@Configuration 注解位于 org.springframework.context.annotation 包中。@Configuration 是只能够作用于类一级的注解，下面给出了其定义。作用了这一注解的类承载了 Spring DI 元数据，其同单个 Spring DI 配置文件（XML）等效。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {

	//作用了@Configuration 注解的受管 Bean 名称
	String value() default "";

}
```

在上述示例代码中，JavaConfigDemo 类也将被注册成一受管 Bean，其名字是 javaConfigDemo。@Bean 用于定义单个受管 Bean，并作用于方法一级，其定

义如下。从这一定义可以看出，@Bean 实现了<bean/>元素相关语义。

```
@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Bean {

    //受管 Bean 的名字
    String[] name() default {};

    Autowire autowire() default Autowire.NO;

    String initMethod() default "";

    String destroyMethod() default AbstractBeanDefinition.INFER_METHOD;

}
```

在上述 JavaConfigDemo 类中，我们定义了两个受管 Bean，它们的名字分别是 helloStr 和 helloWorld，而 helloWorld 受管 Bean 依赖 helloStr 受管 Bean。

开发者清楚，Spring Framework 提供了大量的 XML 元素，以实现 Spring DI 配置文件的定义、管理。类似地，除@Configuration 和@Bean 外，Spring Framework 于 org.springframework.context.annotation 包中提供了其他大量注解，以承载 Spring DI 元数据，比如@DependsOn、@Description、@Import、@ImportResource、@Lazy、@Primary、@Scope 等。

事实上，@Configuration 和 XML 配置文件可以混合使用，即部分受管 Bean 采用<bean/>元素表达，而部分受管 Bean 用@Bean 等注解表达。我们会在 Spring Framework 5.0 系列作品的其他内部资料中结合具体实例深入阐述这方面的问题。

Spring XML 配置文件不可替代

基于 Java 及注解配置 DI 容器，这种做法避免了手工维护 XML 配置文件这一繁琐的工作。但是，开发者必须记住，Java 及注解替代不了 XML 配置文件，因为在一些复杂场合必须借助 XML 配置文件才能够完成。

3 Spring 表达式语言（SpEL）

Spring 表达式语言，简称 SpEL，它是功能强大的表达式语言，用于运行期操控 Spring DI 容器相关内容。SpEL 是 Spring 生态系统中用于评估表达式的基础语言，开发者可以在非 Spring Framework 应用中单独启用 SpEL。本章内容将结合 speldemo 项目，并围绕 SpEL 的方方面面展开论述。

3.1 使用 SpEL

开发者可以借助注解或在 XML 配置文件中使用 SpEL。下面示例代码借助@Value 注解给出了 Spring 表达式，它会将 file.encoding 系统属性取值赋给 userName 属性，比如 UTF-8。

```
public class TestBean1 {

    @Value("#{systemProperties['file.encoding']}")
    private String userName;

    private boolean validFlag;
```

为启用@Value 注解，开发者需要启用<context:annotation-config/>元素。下面给出了示例配置，摘自 spel.xml。其中，validFlag 属性取值是基于 Spring 表达式给出的，即进行布尔计算得出。

```
<context:annotation-config/>

<bean id="testBean1" class="test.TestBean1">
    <property name="validFlag" value="#{true and true}"/>
</bean>
```

受管 Bean 也可以通过 Spring 表达式引用其他受管 Bean 的属性取值，下面给出了示例配置。此时，testBean2 的 userName1 属性取值来自 testBean1 的 userName 属性取值。

```
<bean id="testBean2" class="test.TestBean2">
    <property name="userName1" value="#{testBean1.userName}"/>
    <property name="userName2" value="#{systemProperties['file.encoding']}"/>
</bean>
```

有兴趣的读者可以去运行 SpelDemo 示例应用。

3.2 基于 API 方式使用 SpEL

通过 API 方式使用 SpEL，能够发挥 SpEL 的最大价值，这也是最灵活的使用方式，尤其是在非 Spring Framework 应用场合中意义非凡。接下来将围绕示例应用 SpelApiDemo 展开介绍。

下面给出了评估字符串（文字表达式）的应用场景。可以看出，开发者通过构造 SpelExpressionParser 对象可以获得 Spring 表达式分析器，调用其暴露的 parseExpression()方法对 Spring 表达式进行评估。

```
ExpressionParser parser = new SpelExpressionParser();
//评估字符串
Expression exp = parser.parseExpression("'worldheart'");
String valueStr = (String)exp.getValue();
log.info(valueStr);
```

借助 Spring 表达式也可以完成数学运算。下面展示了绝对值计算。

```
//数学运算
valueStr = parser.parseExpression("-100 对应的绝对值是， #{T(java.lang.Math).abs(-100)}",
                    new TemplateParserContext()).getValue(String.class);
log.info(valueStr);
```

或者，借助 Spring 表达式也可以完成字符串长度评估，具体如下。

```
//评估字符串长度
exp = parser.parseExpression("'worldheart'.bytes.length");
int valueLength = (Integer) exp.getValue();
log.info(valueLength);
```

待下节介绍完 SpEL 语言指引后，开发者能够看到它的强大爆发力。

3.3 SpEL 语言指引

这里将围绕 SpEL 语言的各种语法进行系统性阐述，具体过程将结合示例应用 SpelRefDemo 进行。

文字表达式的相关内容之前已经看过了，下面再次给出了相关示例。

```
ExpressionParser parser = new SpelExpressionParser();

double doubleValue = (Double) parser.parseExpression("3.1415E+2").getValue();
log.info(doubleValue);

int intValue = (Integer) parser.parseExpression("0x6FFF").getValue();
log.info(intValue);

boolean booleanValue = (Boolean) parser.parseExpression("false").getValue();
log.info(booleanValue);
```

```
Object nullValue = parser.parseExpression("null").getValue();
log.info(nullValue);
```

或者，可以借助 Spring 表达式评估集合对象，正如下面给出的各种 List 对象。

```
List strList = (List) parser.parseExpression("{'a','b','c','d'}").getValue();
log.info(strList);

List intList = (List) parser.parseExpression("{1,2,3,4}").getValue();
log.info(intList);

List listList = (List) parser.parseExpression("{{'a','b'},{'c','d'}}").getValue();
log.info(listList);
```

或者，也可以借助 Spring 表达式进行相应的方法调用操作，具体如下。

```
log.info(parser.parseExpression("'Hi,FIFA 2014!'.substring(3,12)").getValue(String.class));
```

Spring 表达式也非常擅长进行各种关系运算，比如关系运算、逻辑运算、数学运算。下面给出了示例代码。

```
boolean trueBoolean = parser.parseExpression("6 != 8").getValue(Boolean.class);
log.info(trueBoolean);

trueBoolean = parser.parseExpression("6 < 8").getValue(Boolean.class);
log.info(trueBoolean);

trueBoolean = parser.parseExpression("'hello' < 'hi'").getValue(Boolean.class);
log.info(trueBoolean);

trueBoolean = parser.parseExpression("!('hello' instanceof T(char))").getValue(Boolean.class);
log.info(trueBoolean);

trueBoolean = parser.parseExpression("true or false").getValue(Boolean.class);
log.info(trueBoolean);

double dou = parser.parseExpression("6 + 8 + 6*8 - 1e2 + 8/6").getValue(Double.class);
log.info(dou);
```

借助 Spring 表达式也可以动态构建对象，示例如下。

```
parser.parseExpression("new java.lang.String('hi,FIFA 2014!')").getValue(String.class);
```

或者，Spring 表达式也可以进行 If-Then-Else 运算，示例如下。

```
parser.parseExpression("'hello' == 'hi' ? true : false").getValue(Boolean.class);
```

有关 SpEL 语言的其他知识，请开发者参考 Spring Framework Reference Documentation。

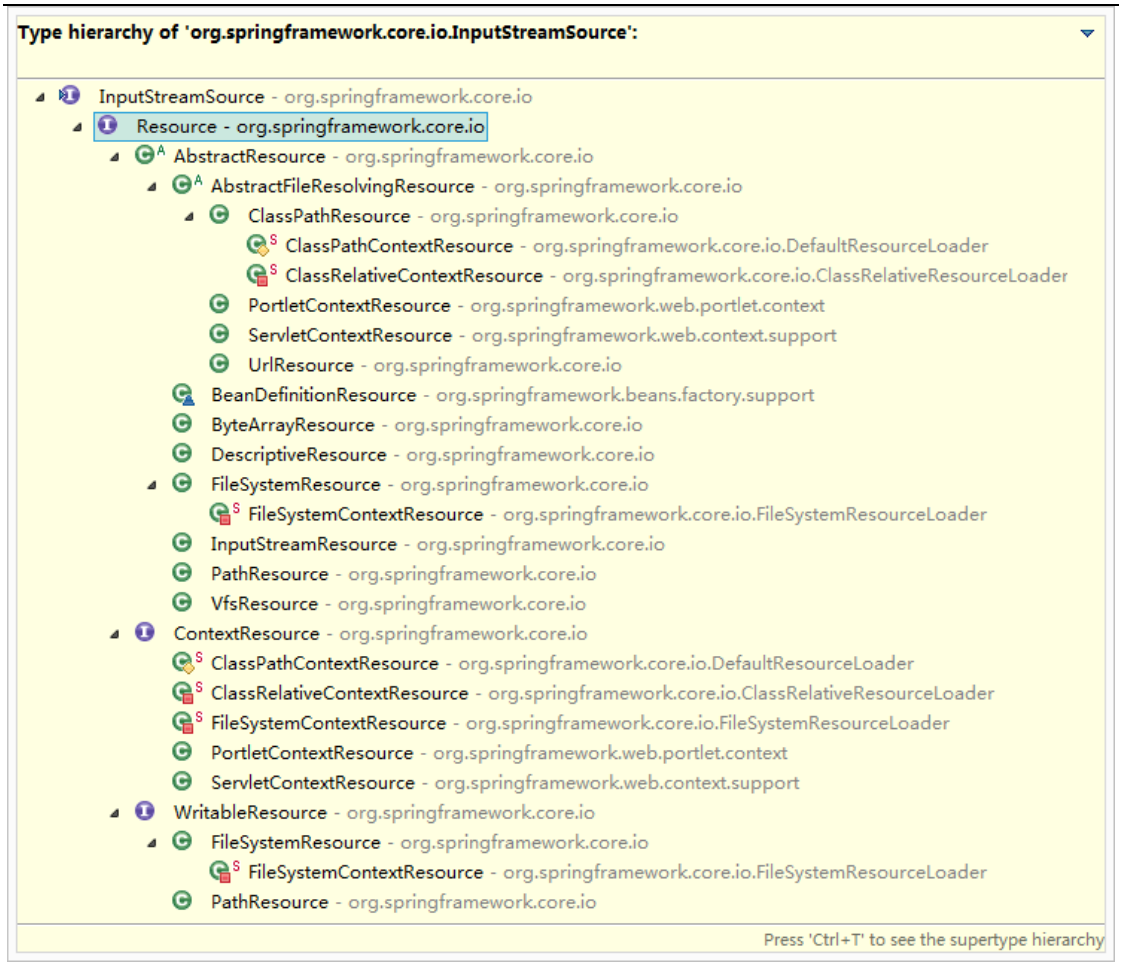
4 Spring Framework 资源抽象

大千世界中，万事万物都传递着信息。这些信息经过简单或复杂加工，便能够成为宝贵的数据。在计算机领域，RDBMS 会将数据以记录的形式存储，而文件系统会以文件的形式存储数据，LDAP 系统会以目录形式存储数据。在这些不同领域，访问并操纵数据资源的方式和策略都不同。在 Spring DI 容器中，Resource 接口统一了访问各种底层资源的具体策略，比如属性文件、.xml、WEB-INF/下的各类资源、物理文件、输入流、等等。

本节将结合 resourcedemo 项目，并围绕 Resource 接口展开论述。

4.1 内置的 Resource 继承链

下图展示了 org.springframework.core.io.Resource 接口及其重要实现，它们分别针对不同资源进行了抽象和封装。开发者从它们各自的名字就能够很容易判断出各个实现的用途。



图表 21 Resource 接口及其子类

细心的开发者应该已经注意到，之前的 iocdemo 项目使用到 Resource 接口、ClassPathResource 实现。比如，new ClassPathResource("beanfactory.xml")中的 beanfactory.xml 传递了这样一种信息：beanfactory.xml 位于应用执行的 classpath 中，而且位于根路径下。Spring Framework 将各种底层资源抽象成 Resource。无论是 Spring IoC 容器本身，还是应用程序（包括那些非 Spring Framework 架构的应用），它们都可以使用这一抽象。

下面展示了 ResourceDemo1 应用的部分内容。ClassPathResource 支持字符串输入，它还能够依据字节码的相对位置查找资源，比如在 TestBean 类所在目录定位 resource2.xml。

```
//在 classpath 根路径查找 resource1.xml
Resource res = new ClassPathResource("resource1.xml");

//在 test 目录定位 resource2.xml
res = new ClassPathResource("test/resource2.xml");
```

```
//在 TestBean 类所在目录定位 resource2.xml
res = new ClassPathResource("resource2.xml", TestBean.class);
```

FileSystemResource 支持字符串输入或 File Handler 输入，示例如下。

```
//直接传入文件路径
res = new FileSystemResource("D:/sts-bundle/workspace/springframework42/src/test/resource2.xml");

//将构建的 File Handler 传入到 FileSystemResource 中
File file = new File("D:/sts-bundle/workspace/springframework42/resourcedemo/src/resource1.xml");
res = new FileSystemResource(file);
```

UrlResource 支持标准的 URL 格式，示例如下。Resource 全面兼容于 URL，并将 URL 作为 Resource 抽象的重要底层实现技术，因此它并没有替代 URL 的功能。

```
try{
        //支持 file:、http:、ftp:，等前缀，这些都是标准的 URL 格式
        res = new UrlResource("file:D:/sts-bundle/workspace/springframework42/resourcedemo/src/resource1.xml");
} catch(MalformedURLException mue){
        log.error("", mue);
}
```

在完成 Resource 的定位和构建工作后，可以将它们传入到 XmlBeanFactory 中，从而完成 DI 容器的构建。这些资源便是构成 DI 容器的元数据配置信息。相比之下，在构建 ApplicationContext 时，开发者并不需要直接将 Resource 传入到 ApplicationContext 实现类中，比如 FileSystemXmlApplicationContext。

ApplicationContext 主要是面向企业级应用的开发，因此开发者只需将资源的位置信息告知给它即可，至于底层 Resource 的具体构建工作，则完全可以交给 DI 容器智能完成。ApplicationContextConstructDemo1 演示了各种构建方式，具体如下。

```
//在 classpath 根路径查找 resource1.xml
ApplicationContext ac = new ClassPathXmlApplicationContext("resource1.xml");

//在 classpath test 路径查找 resource2.xml
ac = new ClassPathXmlApplicationContext("classpath:test/resource2.xml");

//在 TestBean 类所在 classpath 定位 resource2.xml
ac = new ClassPathXmlApplicationContext("resource2.xml", TestBean.class);

//通过 URL 定位 resource1.xml
ac= new ClassPathXmlApplicationContext(
        "file:D:/  sts-bundle/workspace/springframework42/resourcedemo/src/resource1.xml");

//直接传入文件路径
```

```
ac = new FileSystemXmlApplicationContext(
        "D:/sts-bundle/workspace/springframework42/resourcedemo/src/test/resource2.xml");

//通过 classpath 定位 resource2.xml
ac = new FileSystemXmlApplicationContext("classpath:test/resource2.xml");
```

构建 IoC 容器时，不同资源类型可以提供给不同 ApplicationContext 实现者。比如，classpath 类型的资源可以提供给 FileSystemXmlApplicationContext，而 url 类型的资源可以提供给 ClassPathXmlApplicationContext。

4.2 借助 DI 容器访问各种资源

一旦 DI 容器构建成功，开发者便可借助它访问到各种资源，比如通过 ApplicationContext 暴露的 getResource()方法。ResourceDemo2 应用展示了这一方法的各种用法，示例如下。

```
ApplicationContext ac = new ClassPathXmlApplicationContext("classpath:test/resource2.xml");

//获得 ClassPathResource
Resource res = ac.getResource("test/resource2.xml");

//获得 ClassPathResource
res = ac.getResource("classpath:resource1.xml");

//获得 UrlResource
res = ac.getResource("file:D:/sts-bundle/workspace/springframework42/resourcedemo/src/resource1.xml");

//获得 ClassPathResource
res = ac.getResource("D:/sts-bundle/workspace/springframework42/resourcedemo/src/resource1.xml");

ac = new FileSystemXmlApplicationContext(
        "file:D:/sts-bundle/workspace/springframework42/resourcedemo/src/resource1.xml");

//获得 UrlResource
res = ac.getResource("file:D:/sts-bundle/workspace/springframework42/resourcedemo/src/resource1.xml");

//获得 ClassPathResource
res = ac.getResource("classpath:resource1.xml");

//获得 FileSystemResource
res = ac.getResource("D:/sts-bundle/workspace/springframework42/resourcedemo/src/resource1.xml");
```

默认时，ClassPathXmlApplicationContext 容器会返回 ClassPathResource 类型的资源，而 FileSystemXmlApplicationContext 会返回 FileSystemResource。如果开发者将 URL、classpath 相关信息添加到 getResource()方法中，则 IoC 容器会返回期望的 Resource 类型。

4.3 妙用 classpath*前缀

为强制 DI 容器返回 ClassPathResource 类型的资源，开发者可以为传入 geResource()方法的参数中添加“classpath:”前缀，下面给出了示例。

```
//获得 ClassPathResource
res = ac.getResource("classpath:resource1.xml");
```

与此同时，在构建 DI 容器时，还可以使用通配符，示例如下。此时，查找到的所有 XML 文件将构成单个 DI 容器，比如/resource1.xml、/test/resource2.xml 集结在一起，以组建单个 DI 容器。

```
//在 classpath 路径上查找以 resource 开头的所有 XML 配置文件（相对当前应用）
ApplicationContext ac1 = new ClassPathXmlApplicationContext("classpath:**/resource*.xml");
```

下面展示了 classpath*前缀的使用，摘自 ApplicationContextConstructDemo2。此时，位于 classpath 路径上的所有 resource1.xml 配置文件将集结在一起，以组建单个 DI 容器。

```
//在 classpath 路径上查找所有 resource1.xml 配置文件（相对当前应用）
ApplicationContext ac2 = new ClassPathXmlApplicationContext("classpath*:**/resource1.xml");
```

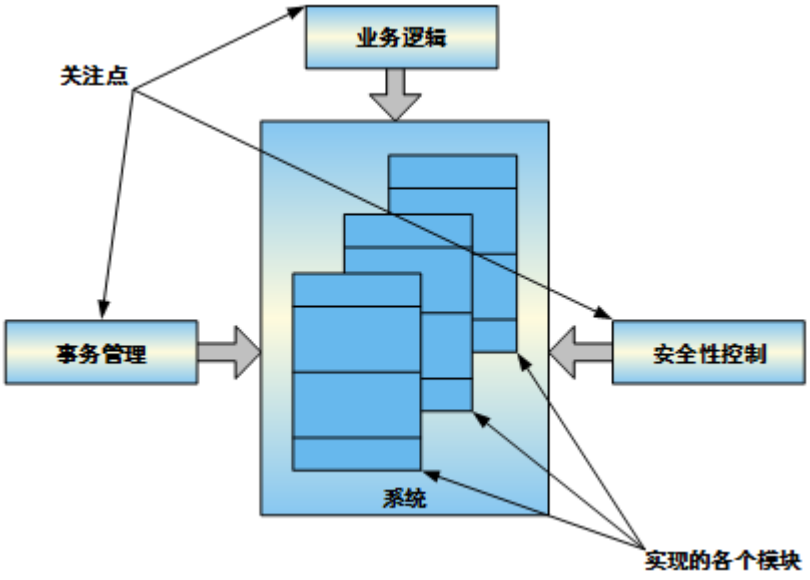
或者，当 DI 容器由多个 XML 配置文件组合而成时，开发者可通过如下方式构建 DI 容器。

```
//指定多个 XML 配置文件
ApplicationContext ac3 = new ClassPathXmlApplicationContext(new String[]{"resource1.xml", "test/resource2.xml"});
```

5 Spring AOP

5.1 AOP 背景知识

通常，为满足整个企业应用某方面的需求，开发者（架构师）需要整理出系统的关注点，下图形象地描述了关注点（http://ramnivas.com/ajia.html），比如事务管理、安全性控制、应用的业务逻辑通常被认为是应用需要重点解决的问题。因此，它们通常被作为关注点（concern）看待。从整个系统角度考虑，它往往是由大量的关注点构成的，比如在收银系统（POS）中，业务架构师往往需要考虑如下几方面的关注点：收银管理、用户管理、销售管理、促销管理、外设管理、报表查询等。除了这些业务功能之外，还需要涉及到系统的可移植性、可维护性、安全性。



图表 22 系统涉及到的各个关注点

大体而言，所有的关注点可划分为两类：核心关注点和横切关注点。其中，核心关注点主要关注系统的业务逻辑；横切关注点主要关注系统级的服务，以供业务逻辑使用。在整个业务逻辑中，到处都将涉及到横切关注点。因此，对于各个已实现的模块（业务逻辑）而言，都将有大量的横切关注点（即系统级服务，比如日志、事务、安全性）实现为它服务。

使用 OO 技术能够很好地实现业务逻辑关注点。但是，由于业务逻辑的差异

性，使得开发者不能够提供统一的框架来满足各种各样的业务需求。相比之下，AOP 技术可用来实现横切关注点。因为横切关注点关注的是系统级服务，这类服务对于大部分应用而言都是最常见的，很容易将它们抽象出来，并加以实现。本章内容正是专注于横切关注点的。

面向对象技术将应用中的核心关注点分解成由层次（继承）结构组成的领域对象集合，而 AOP 能够将应用中的横切关注点分解成由切面（Aspect）组成的生态子系统。OOP 和 AOP 看待问题的角度不同，但它们互相补充，这使得系统的开发变得简单、可维护性得到增强。为实现单个横切关注点，开发者可能需要采用若干切面满足它。比如，为实现安全性控制关注点，开发者需要提供用户认证切面、用户授权切面。

各种 AOP 技术和框架实现切面的具体策略和方法都不一样，但是 AOP 涉及到的基本概念适用于所有的 AOP 技术和框架。下面罗列了相关术语及具体解释。

连接点（join point）。它指应用（目标对象）执行的某个点。比如，执行到某个方法、访问到某个成员变量、某异常的抛出、装载某个类。通常，人们会用连接点模型的强弱来衡量目标 AOP 实现的强弱。AspectJ 8 支持的连接点类型非常多，相比之下，Spring AOP 这方面能力就很一般。

pointcut。它能够声明、集结连接点。通常，装备要同 pointcut 关联在一起，一旦某连接点被触发，则会立即执行相应的装备。开发者借助于 pointcut 表达语言能够完成 pointcut 的指定工作。

装备（advice）。装备指切面在特定连接点所采取的动作，这些特定连接点是由 pointcut 选定的。目前，主要存在五种装备类型，即 Before、AfterReturning、AfterThrowing、After、Around。Before 装备能够在特定连接点被触发前执行，比如某方法执行前，Before 装备会先执行。其他装备类型类似，我们会在后续内容详细介绍各种类型装备的使用。

引入（introduction），也称为 inter-type 声明。引入能够往 Java 类、接口或切面新增新的接口、方法、成员变量。某些场合，引入也是一种装备。

目标对象（target object）。指由若干切面装备的对象。人们常常将目标对象也称为被装备对象（advised object）。

织入（weaving）。将切面、目标对象连接在一起的过程称为织入，也就是

将切面应用到目标对象中。通常，织入时机有编译期（Compile Time，CT）、装载期（Load Time，LT）、运行期（Runtime）。不同 AOP 技术支持的织入时机都不同，比如 Spring AOP 支持运行期织入、AspectJ 8 支持编译期和装载期织入。

切面（aspect）。同 OOP 中的对象（object）概念一样，切面在 AOP 中占据了重要的位置，它将 pointcut、装备、引入、目标对象等信息集结在一起，从而定义出相应的织入规则，这样一个整体称为切面。比如，AspectJ 8 和 Spring AOP 支持@AspectJ 风格的切面定义，另外 Spring AOP 还支持基于<aop:config/>元素的 POJO 切面定义和其他类型的切面定义。

我们即将进入到 AspectJ 8 的介绍之中，还是用实践说话吧！

5.2 AspectJ 8 介绍

AspectJ 针对 Java 编程语言进行了扩展，从而引入 AOP 技术到 Java 平台中，其历史非常悠久。如今，AspectJ 项目已经交由 Eclipse 组织维护和开发（http://www.eclipse.org/aspectj）。AspectJ 易于使用、兼容于 Java 平台，因此许多企业级 Java 应用广泛使用了这一 AOP 技术和框架。AspectJ 开发的 pointcut 表达语言简单，但功能强大。AspectJ 8 是目前的最新版本。

开发者能够借助注解定义切面，即@AspectJ 切面。另外，AspectJ 8 支持 LTW（装载期织入）机制，即在类装载期动态织入 AOP 切面，这大大提升了开发者的开发体验。

回到 PASCAL 年代

在计算机编程领域，PASCAL 曾一度被作为面向对象技术的授课语言。如今，Java 取代了 PASCAL 的地位，它成为了课堂上讲授 OO 技术的目标语言。

AOP 时代已经到来，面向切面编程领域会出现类似的轨迹吗？是的，AspectJ 就是非常好的 AOP 语言和框架。如果需要讲授 AOP 技术，则 AspectJ 当之无愧。AspectJ 是非常严谨的一门 AOP 语言和技术框架，它的 pointcut 表达语言堪称完美，而且其切面支持的织入时机灵活多样（比如，编译期、Load-Time）。

注意，AspectJ 可不是简单的、仅仅适合于教学的 AOP 语言和技术，无数的企业级 Java 应用中已经看到了它的伟大身躯。

因此，为了便于开发者更好地理解和掌握 AOP 基础知识，本书结合 AspectJ 8 来阐述。另外，

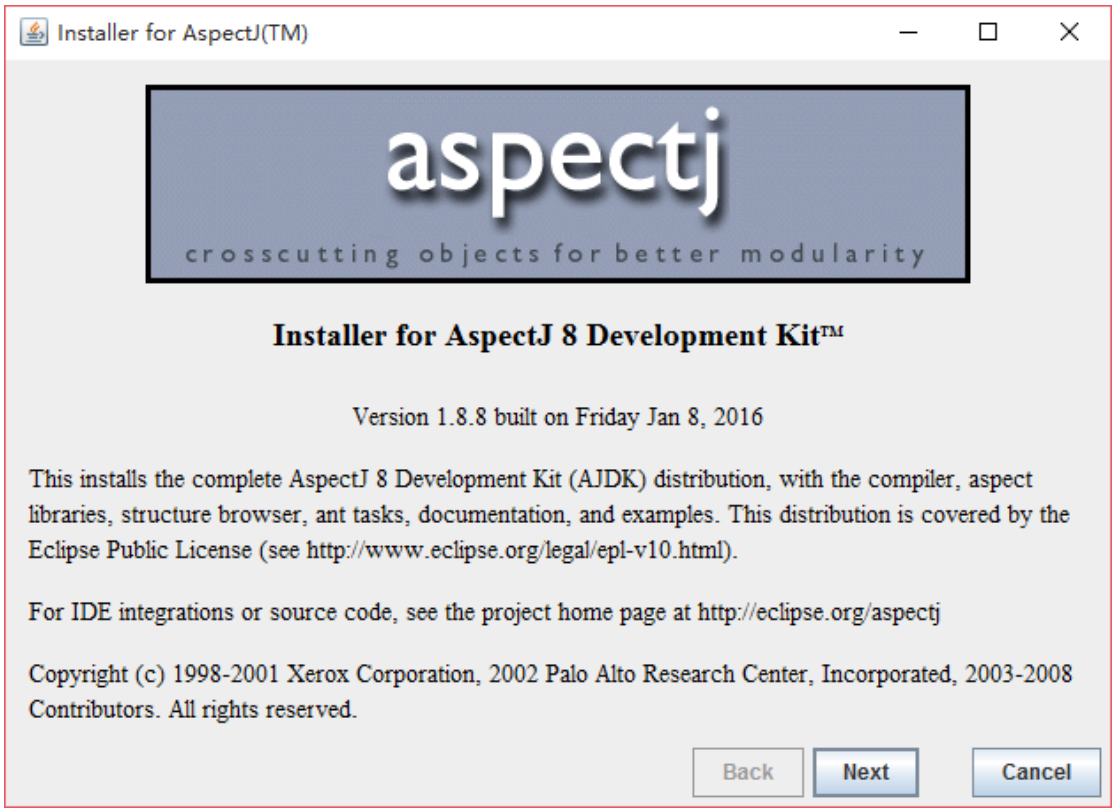
Spring Framework 5.0 中的 AOP 实现同 AspectJ 8 进行了一流的集成，比如它集成了 AspectJ 8 的 pointcut 表达语言。

无论如何，开发者都要重视本节内容。

下面我们以 Eclipse aspectj8demo 项目为例展开对 AspectJ 8 的介绍。

5.2.1 AspectJ 的安装及使用

首先，开发者能够从 http://www.eclipse.org/aspectj 官方网站下载到最新的 AspectJ 8 发布包(.jar 存档)，比如 aspectj-1.8.8.jar。借助“java –jar aspectj-1.8.8.jar”命令行便可启动 AspectJ 8 的安装过程。随后，开发者需要选定目标 JDK 和安装目录，从而完成 AspectJ 8 的安装，具体见下图。



图表 23 AspectJ 1.8 安装

按照要求安装完 AspectJ 8 后，开发者需要将 lib 目录中的 aspectjrt.jar 追加到 Java classpath 中。同时，开发者还可以将 D:\aspectj1.8\bin 目录追加到 path 路径中，从而能够在任何地方执行到 AspectJ 8 提供的命令行工具。

我们在 aspectj8demo 项目中提供了 test.DisPersonInfo.java 类，具体如下。它暴露了两个 compute()方法。

```
public class DisPersonInfo {
```

```
    private static final Log log = LogFactory.getLog(DisPersonInfo.class);

    public static void compute(String person) {
        log.info(person);
    }

    public static void compute(String person, int age) {
        log.info(person + "已经" + age + "岁了！ ");
    }

}
```

然后，MainTest 示例应用简单地调用了上述两个方法，具体如下。

```
public class MainTest {

    private static final Log log = LogFactory.getLog(MainTest.class);

    public static void main(String[] args) {
        log.info("MainTest main().....");

        DisPersonInfo.compute("李三");
        DisPersonInfo.compute("王二", 30);
    }

}
```

现在，我们希望 MainTest 在调用 DisPersonInfo 的 compute 方法之前，能够智能地输出一些提示信息，比如某某应用要调用 compute()方法了。基于这一需求，我们提供了如下 test.AspectJAop.aj 类，这是一个 AspectJ 8 切面实现。它通过 pointcut 关键字定义了名字为 xx()的 pointcut，并通过 before 关键字引用到这一 pointcut。借助于 before 关键字，开发者能够定义出相应的装备。注意，xx() pointcut 集结了相应的连接点，即执行 DisPersonInfo 的所有 compute()方法，execution 是 AspectJ 8 中 pointcut 表达语言的一个关键字。我们会在后续内容系统性阐述 pointcut 表达语言。

其中，execution 后面的*表示若干字符（排除“,”在外），后面的“..”表示若干字符（包括“,”在内）。因此，所有的 compute()方法执行都将被拦截，无论各自的返回类型、可见性（private/public/…）如何。

```
public aspect AspectJAop {

    private static final Log log = LogFactory.getLog(AspectJAop.class);

    pointcut xx()
        : execution(* DisPersonInfo.compute(..));
```

```
    before(): xx() {
        log.info("AspectJAop @Before");
    }

}
```

定义 pointcut 表达式时，开发者可以以 private pointcut、pointcut、public pointcut 形式定义出相应的 pointcut。比如，如果在 pointcut 前加上 private，则这一 pointcut 定义只能够在本切面中可见；如果未在 pointcut 前加上任何内容，则这一 pointcut 定义在包一级可见。其他依次类推，这同 Java 的语义一致。现在，开发者可以借助 AspectJ 8 安装目录中 bin 下的 ajc 批处理命令来静态织入（即编译期织入）上述切面，并运行 MainTest 应用。具体操作如下，请开发者仔细研究这一具体过程。

```
D:\sts-bundle\workspace\springframework42\aspectj8demo\src>set classpath
classpath=.;D:\aspectj1.8\lib\aspectjrt.jar;

D:\sts-bundle\workspace\springframework42\aspectj8demo\src>set
classpath=%classpath%;..\..\springframework-libs\commons-logging-1.1.3.jar;..\..\springframework-libs\log4j-1.2.17.jar;

D:\sts-bundle\workspace\springframework42\aspectj8demo\src>set classpath
classpath=.;D:\aspectj1.8\lib\aspectjrt.jar;;..\..\springframework-libs\commons-logging-1.1.3.jar;..\..\springframework-libs\log4j-1
.2.17.jar;

D:\sts-bundle\workspace\springframework42\aspectj8demo\src>ajc test\AspectJAop.aj test\DisPersonInfo.java
    test\MainTest.java

D:\sts-bundle\workspace\springframework42\aspectj8demo\src>java test.MainTest
[INFO] 2014-06-25 12:11:02,419 test.MainTest - MainTest main().....
[INFO] 2014-06-25 12:11:02,419 test.AspectJAop - AspectJAop @Before
[INFO] 2014-06-25 12:11:02,419 test.DisPersonInfo - 李三
[INFO] 2014-06-25 12:11:02,419 test.AspectJAop - AspectJAop @Before
[INFO] 2014-06-25 12:11:02,419 test.DisPersonInfo - 王二已经 30 岁了！

D:\sts-bundle\workspace\springframework42\aspectj8demo\src>
```

借助上述 java 命令行执行 test.MainTest 时，classpath 路径上仅仅包含了 AspectJ 8 的 aspectjrt.jar。这一 jar 包大小仅仅有一百多千字节（KB），太不可思议了。我们可以推测到：ajc 在静态织入 AspectJAop 切面时动了手脚。此时，如果开发者去 aspectj8demo\src\test 目录查看，则会发现一共存在 3 个.class 文件，切面本身也被编译成 Java 字节码。注意，这三个.class 文件都是标准的 Java .class 文件，它们能够在任何 Java 虚拟机中运行，但前提是将 aspectjrt.jar 包追加到 classpath 执行路径中。如果借助反编译工具[3]对上述三个.class 进行反编译，开发

[3] 本书推荐 http://www.neshkov.com/位置的 DJ Java Decompiler。

者则会发现 DisPersonInfo 存在另一番风景，具体如下。开发者不难猜测，“AspectJAop @Before”是由 AspectJAop.aspectOf().ajc$before$test_AspectJAop$1$385401()输出的。至于 AspectJAop .class 字节码，留给开发者自行研究。

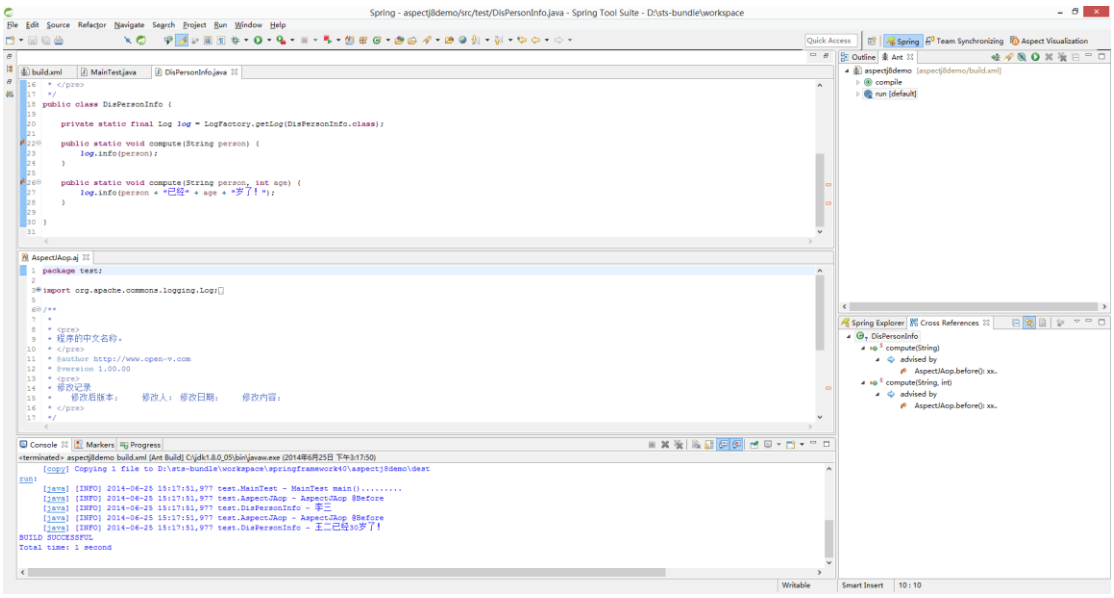
```
public static void compute(String person)
{
    AspectJAop.aspectOf().ajc$before$test_AspectJAop$1$385401();
    log.info(person);
}

public static void compute(String person, int age)
{
    AspectJAop.aspectOf().ajc$before$test_AspectJAop$1$385401();
    log.info(person + "\u5DF2\u7ECF" + age + "\u5C81\u4E86\uFF01");
}
```

AspectJ 8 除了提供 ajc 批处理命令工具静态织入切面外，它还针对 Ant 进行了集成。具体如下（摘自 aspectj8demo 中的 Ant build.xml），这同运行 ajc 批处理命令工具毫无差别，但更方便，没有太多的开发者愿意手工敲 DOS 命令来编译和运行 Java 应用。

```
<?xml version="1.0"?>
<project name="aspectj8demo" default="run">

    <target name="run" depends="compile">

        <java classname="test.MainTest">
            <classpath>
                <pathelement location="../springframework-libs/commons-logging-1.1.3.jar"/>
                <pathelement location="../springframework-libs/log4j-1.2.17.jar"/>
                <pathelement location="dest"/>
                <pathelement location="D:/aspectj1.8/lib/aspectjrt.jar"/>
            </classpath>
        </java>

    </target>

    <target name="compile">

        <taskdef resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.properties">
            <classpath>
              <pathelement location="D:/aspectj1.8/lib/aspectjtools.jar"/>
            </classpath>
        </taskdef>

        <delete dir="dest"/>
        <mkdir dir="dest"/>

        <iajc destdir="dest" source="1.8">
            <sourceroots>
```

```
                    <pathelement location="src"/>
                </sourceroots>
                <classpath>
                    <pathelement location="../springframework-libs/commons-logging-1.1.3.jar"/>
                    <pathelement location="../springframework-libs/log4j-1.2.17.jar"/>
                    <pathelement location="D:/aspectj1.8/lib/aspectjrt.jar"/>
                </classpath>
            </iajc>

            <copy todir="dest" file="src/log4j.properties"/>
            <copy todir="dest" file="src/commons-logging.properties"/>

        </target>

</project>
```

AspectJ 8 还存在 AspectJ 开发工具，通过 http://www.eclipse.org/ajdt 能够下载到它，开发者借助 ajdt 能够开发 AspectJ 项目。注意，aspectj8demo 本身就是 AspectJ 项目（借助于 ajdt 提供的 wizard 生成），具体见下图。



图表 24 AspectJ 开发工具（ajdt）

AspectJ 开发工具会直观地将各种装备、连接点显式地标识清楚。如果项目中定义的切面、连接点很多，则可以借助 AspectJ 开发工具提供的 Aspect Visualization Perspective 更直观地统计、展现它们。由于 aspectj8demo 是 AspectJ 项目，因此运行 test.MainTest 同运行普通 Java 应用没有任何差别，因为 AspectJ 项目会智能地调用 ajc 编译器完成切面的静态织入工作，这些都说明“工欲善其事，必先利其器”。

当然，如果开发者没有 ajdt，则通过 AspectJ 8 中 bin 目录下的 ajbrowser 命

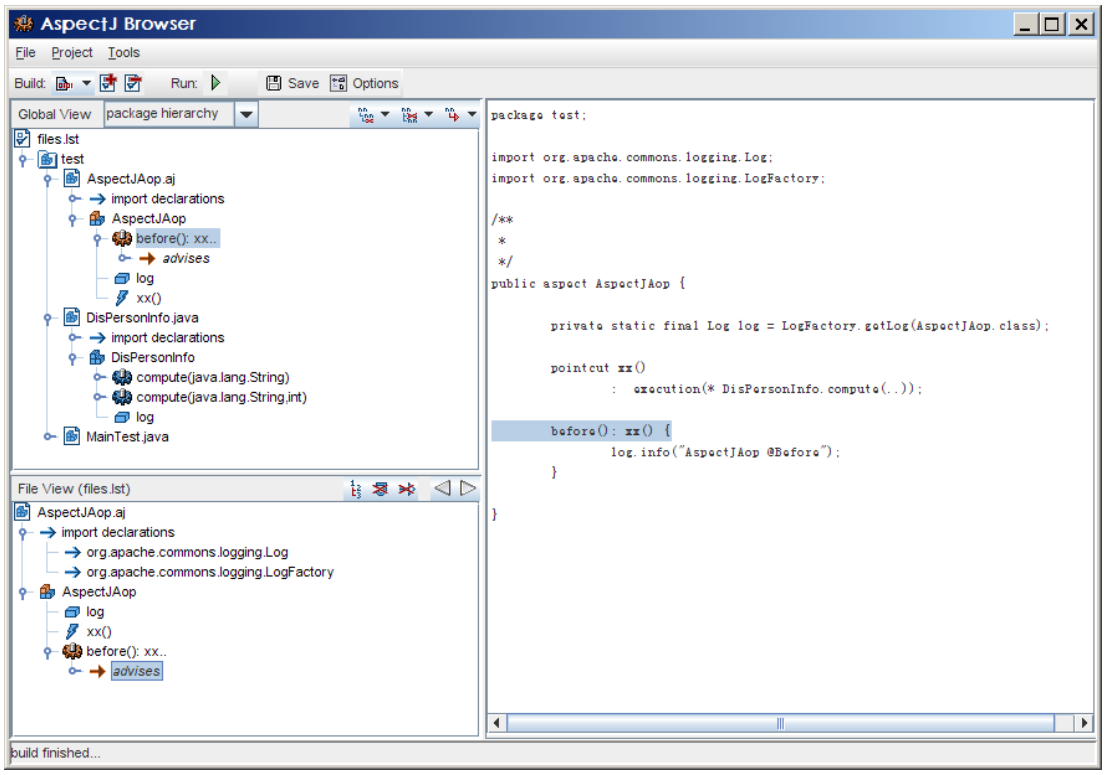
令行工具也能够浏览到相应的切面、连接点等信息，并运行相应的客户应用。比如，我们在 aspectj8demo 的 src 目录下提供了 files.lst 文件，其具体内容如下，它指定了供 ajbrowser 操控的相关 Java 类和切面信息。

```
test\AspectJAop.aj

test\DisPersonInfo.java

test\MainTest.java
```

随后，我们通过如下命令便能够启动这一 GUI 工具。

```
D:\sts-bundle\workspace\springframework42\aspectj8demo\src>ajbrowser files.lst
```

具体见下图，开发者可以在其中构建和运行 AspectJ 8 使能应用。无论是功能，还是易用性，ajbrowser 都没有比 Eclipse ajdt 更占优势。



图表 25 AspectJ 8 提供的 ajbrowser 命令行工具

开发者是否注意到，上述介绍的所有织入时机都是发生在编译期。实际开发场景中，开发者更希望 AspectJ 8 能够动态织入切面到 POJO 类中。幸好，AspectJ 8 确实也支持装载期（Load-Time）织入。启用 AspectJ 8 的这一织入时机时，开发者开发的 POJO 类的字节码不会遭到 AspectJ 8 的修改，至少从物理.class 上觉察不到，AspectJ 8 会透明处理字节码。AspectJ 8 提供了 aj5 命令行工具来运行 POJO 类，具体见如下运行序列。

```
D:\sts-bundle\workspace\springframework42\aspectj8demo\src>set
classpath=.;D:\aspectj1.8\lib\aspectjrt.jar;D:\sts-bundle\workspace\springframework42\springframework-libs\commons-logging-1.1.3.jar;D:\sts-bundle\workspace\springframework42\springframework-libs\log4j-1.2.17.jar

D:\sts-bundle\workspace\springframework42\aspectj8demo\src>ajc -outxml test\AspectJAop.aj test\DisPersonInfo.java

D:\sts-bundle\workspace\springframework42\aspectj8demo\src>javac -encoding UTF-8 test\*.java

D:\sts-bundle\workspace\springframework42\aspectj8demo\src>aj5 test.MainTest
[INFO] 2014-06-25 15:35:29,213 test.MainTest - MainTest main().....
[INFO] 2014-06-25 15:35:29,291 test.AspectJAop - AspectJAop @Before
[INFO] 2014-06-25 15:35:29,291 test.DisPersonInfo - 李三
[INFO] 2014-06-25 15:35:29,291 test.AspectJAop - AspectJAop @Before
[INFO] 2014-06-25 15:35:29,291 test.DisPersonInfo - 王二已经 30 岁了!

D:\sts-bundle\workspace\springframework42\aspectj8demo\src>
```

首先，ajc 编译 AspectJAop.aj 切面和 DisPersonInfo POJO 类，由于在 AspectJAop.aj 切面的 pointcut 表达式中使用到这一 POJO 类，因此也需要将它一同传给 ajc。在编译完后，ajc 会在 src 目录下生成 META-INF 目录，并将 aop-ajc.xml（具体如下）存放在那里。aop-ajc.xml 指定了待使用的 AspectJ 8 切面，aj5 会从 META-INF/aop-ajc.xml 获得这一信息，并动态完成切面的织入工作。然后，我们使用了 javac 来编译 test 下的所有 Java 类，期间不会涉及到 AspectJ 8 的任何内容。注意，javac 运行后生成的 DisPersonInfo.class 会覆盖 ajc 生成的 DisPersonInfo.class，因此 DisPersonInfo.class 是纯 POJO 字节码。最后，启用 aj5 命令行工具来完成 test.MainTest 的执行。

```
<aspectj>
    <aspects>
        <aspect name="test.AspectJAop"/>
    </aspects>
</aspectj>
```

有关 AspectJ 8 发布包（.jar）的相关内容，我们就介绍到这里，相信这些知识对于理解本章及本书的后续内容是绰绰有余的。但如果开发者打算对 AspectJ 8 进一步研究，则可以登录其官方网站。本书介绍 AspectJ 8 的目的有如下三点：

- 帮助开发者理解 AOP 的基本概念。
- 由于 Spring 引入了 AspectJ 8 的 pointcut 表达语言（后续内容会深入涉及到），因此有必要对 AspectJ 8 进行较为深入的介绍。
- AspectJ 8 是非常优秀的 AOP 框架，希望本书介绍的内容能够激发开发者的兴趣，AOP 是非常重要的使能技术。

如果开发者只使用到 Spring AOP 技术，则本书介绍的 AspectJ 8 内容是足够

的。下面开始介绍各种装备类型。

5.2.2 Before 装备

Before 装备，即在执行目标操作（比如，调用某业务方法、读取或设置属性取值）之前执行的装备，正如我们之前介绍的 AspectJAop.aj 中的“before(): xx()”一样，通过 before 关键字能够声明 Before 装备。

下面的代码示例展示了如何借助 AspectJ 8 中的@Aspect 注解定义切面。

```
@Aspect
public class AopBeforeAspect {

    private static final Log log = LogFactory.getLog(AopBeforeAspect.class);

    //定义匿名 pointcut，并将传入 setIj()的 i 参数暴露给这一装备
    @Before("set(int *InfoBefore.ij) && args(i)")
    public void xx(int i, JoinPoint thisJoinPoint){
        log.info(i + "");
        log.info(thisJoinPoint);
        log.info("AopBeforeAspect xx @Before");
    }

    //这里使用了 call 关键字，注意它同 execution 的区别
    @Pointcut("call(* IDisPersonInfo.compute(..))")
    public void yy(){}

    @Before("yy()")
    public void zz(){
        log.info("AopBeforeAspect zz @Before");
    }

}
```

我们在使用@AspectJ 风格的 AOP 切面定义时，切面本身是普通 POJO 类，只不过它将一些注解应用到这一 POJO 类中。前面在定义 AspectJAop.aj 切面时，我们使用了 aspect 来标识切面，而这里使用@Aspect 注解。@Aspect 是 AspectJ 5 新引入的风格，开发者习惯将这种风格称为@AspectJ 风格。开发者对这种风格都比较满意，因为其使得 AspectJ AOP 不再陌生，这是基于标准 Java 语法的切面定义。上述切面有如下一些知识点值得开发者注意。

其一，我们定义了一个匿名 pointcut，即“"set(int *InfoBefore.ij) && args(i)"”，其他装备是无法引用到它的。注意，set(int *InfoBefore.ij)指定了连接点，即在对 ij 进行设值时，args(i)将传入 setIj()的 i 暴露给这一装备，并打印出来。装备实现中的 thisJoinPoint 封装了连接点信息。&&后面的“args(i)”表明目标设值方法能

够接收 int 参数，而&&进行的是与运算。AspectJ 8 还支持其他运算规则，比如“||”指定或运算、“!”指定非运算。处于“&&”、“||”、“!”两边的表达式都是 pointcut 表达式。

其二，IDisPersonInfo.compute(..)指拦截 IDisPersonInfo 类中的任何 compute()方法。之前我们在阐述 AspectJAop.aj 切面时，使用了 execution pointcut 关键字来集结连接点，而这里使用了 call 关键字，它们的含义类似，但存在细微差别，execution 指执行目标操作体本身，而 call 指调用者调用 compute()方法。开发者通过反编译 DisPersonInfoBefore.class 和 MainTestAspectBefore.class 能够获得若干重要信息，MainTestAspectBefore 应用调用了 DisPersonInfoBefore，DisPersonInfoBefore.class 中的 setIj()方法被 AspectJ 8 动了手脚，而它的其他方法并没有被修改过字节码，示例如下。

```
public void setIj(int ij)
{
        int i = ij;
        DisPersonInfoBefore dispersoninfobefore = this;
        org.aspectj.lang.JoinPoint joinpoint =
                Factory.makeJP(ajc$tjp_0, this, dispersoninfobefore, Conversions.intObject(i));
        AopBeforeAspect.aspectOf().xx(i, joinpoint);
        dispersoninfobefore.ij = i;
}
```

通过反编译并分析 MainTestAspectBefore.class，开发者能够发现其 main()方法被 AspectJ 8 处理过，具体如下。很显然，execution 和 call 的区别就在此，前者（execution）是修改目标类的字节码达到装备目的；后者（call）是通过修改调用者的字节码达到装备目的。

```
public static void main(String args[])
{
        log.info("MainTestAspectBefore main().....");
        IDisPersonInfo dpi = new DisPersonInfoBefore();
        dpi.setVar(10);
        AopBeforeAspect.aspectOf().zz();
        dpi.compute("\u674E\u4E09");
        AopBeforeAspect.aspectOf().zz();
        dpi.compute("\u738B\u4E8C", 30);
}
```

其三，AspectJ 8 支持的连接点模型非常强健，我们已经在这里体会到了。只要开发者通过调用 setIj()方法修改 ij 成员变量，AspectJ 8 就能够捕捉到这一事件，并采用相应的动作。基于动态代理的 Spring AOP 并没有提供成员变量一级的 pointcut 支持。

最后，如果开发者运行 MainTestAspectBefore 应用，则能够浏览到如下输入信息，这体现了@Before 装备的语义。

```
[INFO] 2014-06-29 22:53:30,161 test.MainTestAspectBefore - MainTestAspectBefore main().....
[INFO] 2014-06-29 22:53:30,241 test.AopBeforeAspect - 10
[INFO] 2014-06-29 22:53:30,241 test.AopBeforeAspect - set(int test.DisPersonInfoBefore.ij)
[INFO] 2014-06-29 22:53:30,243 test.AopBeforeAspect - AopBeforeAspect xx @Before
[INFO] 2014-06-29 22:53:30,243 test.AopBeforeAspect - AopBeforeAspect zz @Before
[INFO] 2014-06-29 22:53:30,243 test.DisPersonInfoBefore - 李三
[INFO] 2014-06-29 22:53:30,243 test.AopBeforeAspect - AopBeforeAspect zz @Before
[INFO] 2014-06-29 22:53:30,243 test.DisPersonInfoBefore - 王二已经 40 岁了!
```

5.2.3 AfterReturning 装备

开发者已经知道，@Before 装备在目标操作执行之前肯定会被触发，那我们在执行目标操作之后是否也可以提供相应的装备呢？答案是肯定的。当然，由于目标操作执行过程中可能会正常执行，也可能会抛出异常，从而非正常退出。鉴于此，AspectJ 8 提供了三种不同的@After 装备类型，具体如下。

- @AfterReturning 装备：在目标操作（即连接点）正常执行后，会触发这类装备的执行。
- @AfterThrowing 装备：在执行目标操作（即连接点）期间，如果抛出了异常，则会触发这类装备的执行。
- @After 装备：在执行目标操作（即连接点）期间，无论是正常退出，还是抛出了异常，则都会触发这类装备的执行。

下面给出了@AfterReturning 装备示例。

```
@Aspect
public class AopAfterReturningAspect {

    private static final Log log = LogFactory.getLog(AopAfterReturningAspect.class);

    @Pointcut("execution(* test.*InfoAfterReturning.compute(..))")
    public void xx(){}

    @AfterReturning("xx()")
    public void yy(){
        log.info("AopAfterReturningAspect yy @AfterReturning");
    }

    @AfterReturning(pointcut="xx()",returning="str")
    public void zz(String str){
        log.info(str);
        log.info("AopAfterReturningAspect zz @AfterReturning");
    }
```

```
}
```

AopAfterReturningAspect 定义了单个 pointcut，并提供了两个不同的@AfterReturning 装备。在同一 AspectJ 切面中，同一装备类型的触发顺序同它们声明的顺序一致，即 yy()会在 zz()之前被触发。如果装备对目标方法的返回值感兴趣，则可以通过 returning 成员指定，并在 zz 中使用到它，注意 returning 成员取值必须同 zz 中的形参一致。至于 str 的类型，开发者能够依据目标方法的返回类型确定。

一旦运行 MainTestAspectAfterReturning 应用，则能够输出如下结果。很显然，在调用 DisPersonInfoAfterReturning 的 compute(String person)方法时，zz()装备没有被触发，因为这一方法没有返回值。

```
[INFO] 2014-06-30 00:09:38,019 test.MainTestAspectAfterReturning - MainTestAspectAfterReturning main().....
[INFO] 2014-06-30 00:09:38,020 test.DisPersonInfoAfterReturning - 李三
[INFO] 2014-06-30 00:09:38,063 test.AopAfterReturningAspect - AopAfterReturningAspect yy @AfterReturning
[INFO] 2014-06-30 00:09:38,063 test.AopAfterReturningAspect - AopAfterReturningAspect yy @AfterReturning
[INFO] 2014-06-30 00:09:38,063 test.AopAfterReturningAspect - 王二已经 30 岁了!
[INFO] 2014-06-30 00:09:38,063 test.AopAfterReturningAspect - AopAfterReturningAspect zz @AfterReturning
```

DisPersonInfoAfterReturning 的业务方法定义如下。

```
public void compute(String person) {
        log.info(person);
}

public String compute(String person, int age) {
        StringBuilder sb = new StringBuilder(person);
        sb.append("已经" + age + "岁了！ ");
        return sb.toString();
}
```

5.2.4 AfterThrowing 装备

为演示@AfterThrowing 装备的使用，DisPersonInfoAfterThrowing 定义了如下方法，它抛出了非受查异常。

```
public void compute(String person) {
        log.info(person);
        throw new RuntimeException("runtime exception");
}
```

下面定义了相应的切面，它定义了两个@AfterThrowing 装备。

```
@Aspect
public class AopAfterThrowingAspect {

        private static final Log log = LogFactory.getLog(AopAfterThrowingAspect.class);

        @Pointcut("execution(* test.*InfoAfterThrowing.compute(..))")
        public void xx(){}
```

```
    @AfterThrowing("xx()")
    public void yy(){
        log.info("AopAfterThrowingAspect yy @AfterThrowing");
    }

    @AfterThrowing(pointcut="xx()",throwing="thr")
    public void zz(RuntimeException thr){
        log.info(thr.getMessage());
        log.info("AopAfterThrowingAspect zz @AfterThrowing");
    }

}
```

如果@AfterThrowing 装备对特定类型的异常感兴趣，则可以借助于 throwing 成员来指定，并在装备方法中提供相应的异常类型，正如我们在 zz()中定义的一样。此后，凡是连接点抛出 RuntimeException 类型的异常（包括其子类），都会被这一@AfterThrowing 装备捕捉到。MainTestAspectAfterThrowing 运行示例如下。

```
[INFO] 2014-06-30 00:20:01,596 test.MainTestAspectAfterThrowing - MainTestAspectAfterThrowing main().....
[INFO] 2014-06-30 00:20:01,598 test.DisPersonInfoAfterThrowing - 王二
[INFO] 2014-06-30 00:20:01,644 test.AopAfterThrowingAspect - AopAfterThrowingAspect yy @AfterThrowing
[INFO] 2014-06-30 00:20:01,644 test.AopAfterThrowingAspect - runtime exception
[INFO] 2014-06-30 00:20:01,644 test.AopAfterThrowingAspect - AopAfterThrowingAspect zz @AfterThrowing
Exception in thread "main" java.lang.RuntimeException: runtime exception
    at test.DisPersonInfoAfterThrowing.compute(DisPersonInfoAfterThrowing.java:27)
    at test.MainTestAspectAfterThrowing.main(MainTestAspectAfterThrowing.java:30)
```

如果开发者对 DisPersonInfoAfterThrowing.class 进行反编译，则会发现其 compute()方法已经被 AspectJ 8 修改过，具体如下。

```
public void compute(String person)
{
    try
    {
        try
        {
            log.info(person);
            throw new RuntimeException("runtime exception");
        }
        //对应 yy()装备
        catch(Throwable throwable)
        {
            AopAfterThrowingAspect.aspectOf().yy();
            throw throwable;
        }
    }
    //对应 zz()装备
    catch(RuntimeException runtimeexception)
    {
```

```
            AopAfterThrowingAspect.aspectOf().zz(runtimeexception);
            throw runtimeexception;
        }
}
```

5.2.5 After 装备

无论目标操作是正常执行（适合于@AfterReturning 装备），还是抛出异常（适合于@AfterThrowing 装备），@After 装备始终都会执行到。很显然，@After 装备非常强势。AopAfterAspect 切面定义示例如下。

```
@Aspect
public class AopAfterAspect {

        private static final Log log = LogFactory.getLog(AopAfterAspect.class);

        @Pointcut("execution(* test.*InfoAfter.compute(..))")
        public void xx(){}

        @After("xx()")
        public void yy(){
                log.info("AopAfterAspect yy @After");
        }

        @After("xx()")
        public void zz(){
                log.info("AopAfterAspect zz @After");
        }

}
```

MainTestAspectAfter 示例运行结果如下。

```
[INFO] 2014-06-30 00:30:24,906 test.MainTestAspectAfter - MainTestAspectAfter main().....
[INFO] 2014-06-30 00:30:24,907 test.DisPersonInfoAfter - 王二
[INFO] 2014-06-30 00:30:24,951 test.AopAfterAspect - AopAfterAspect yy @After
[INFO] 2014-06-30 00:30:24,951 test.AopAfterAspect - AopAfterAspect zz @After
Exception in thread "main" java.lang.RuntimeException: runtime exception
        at test.DisPersonInfoAfter.compute(DisPersonInfoAfter.java:27)
        at test.MainTestAspectAfter.main(MainTestAspectAfter.java:30)
```

5.2.6 Around 装备

Around 装备，指在目标操作（连接点）调用前后执行的装备。@Around 装备的功能最强大，因此它能够在目标操作执行前后实现特定的行为。而且，@Around 装备的使用最为灵活，它甚至可以控制目标操作的执行与否、改变传入到目标操作的参数值。我们可以看出，@Around 装备是上述四种装备类型的全集，这四种装备类型只是它的一个子集。下面给出了 AopAroundAspect 切面定义示例。

```
@Aspect
public class AopAroundAspect {

	private static final Log log = LogFactory.getLog(AopAroundAspect.class);

	@Pointcut("execution(* test.*InfoAround.compute(..)) && args(String)")
	public void xx() {
	}

	@Around("xx() && args(persion)")
	public void yy(ProceedingJoinPoint pjp, String persion) {
		log.info("AopAroundAspect yy @Around");
		try {
			log.info(persion);
			//替换原先的 persion
			persion = "李三";
			pjp.proceed(new Object[]{persion});
		} catch (Throwable thro) {
			log.error("", thro);
		}
	}

	@Around("xx()")
	public void zz(ProceedingJoinPoint pjp) throws Throwable {
		log.info("AopAroundAspect zz @Around");
		log.info(pjp);
		pjp.proceed();
	}

}
```

这一切面定义了两个@Around 装备。其中，yy()装备传入了 persion 参数，这是目标方法使用的参数。开发者通过 pjp.proceed(new Object[]{persion})能够替换传入的 persion 参数，从而动态控制应用的行为。如果开发者不需要控制目标操作，则直接调用 proceed()方法即可，zz()装备正是这样操作目标方法的。另外，ProceedingJoinPoint 对象封装了连接点信息、调用信息。MainTestAspectAround 运行示例如下。

```
[INFO] 2014-06-30 00:33:56,043 test.MainTestAspectAround - MainTestAspectAround main().....
[INFO] 2014-06-30 00:33:56,109 test.AopAroundAspect - AopAroundAspect yy @Around
[INFO] 2014-06-30 00:33:56,109 test.AopAroundAspect - 王二
[INFO] 2014-06-30 00:33:56,109 test.AopAroundAspect - AopAroundAspect zz @Around
[INFO] 2014-06-30 00:33:56,109 test.AopAroundAspect - execution(void test.DisPersonInfoAround.compute(String))
[INFO] 2014-06-30 00:33:56,110 test.DisPersonInfoAround - 李三
```

注意，由于@Around 装备的连接点没有返回值，因此 yy()和 zz()方法的返回类型也是 void。如果连接点存在返回值，则开发者在定义@Around 装备时也必须注意到这一差异性。我们将在“基于@AspectJ 的 Spring AOP”一节内容对

@Around 装备进行更深入的阐述。

5.2.7 引入（Introduction）

Introduction（引入）允许开发者将新的成员变量、方法动态引入到目标类中。AspectJ 8 将引入称为 inter-type 声明。比如，现存在如下 Java 接口，即 IDisPersonInfoIntroduction。

```
public interface IDisPersonInfoIntroduction {

	public void compute(String person);

	public void compute(String person, int age);

}
```

与此同时，我们还提供了上述接口的 DefaultDisPersonInfoIntroduction 实现。注意，compute(String person)方法体为空。

```
public class DefaultDisPersonInfoIntroduction implements IDisPersonInfoIntroduction {

	private static final Log log = LogFactory.getLog(DefaultDisPersonInfoIntroduction.class);

	public void compute(String person) {
	}

	public void compute(String person, int age) {
		log.info(person + "已经" + age + "岁了！");
	}

}
```

下面给出了 DisPersonInfoIntroduction 实现类，它是我们开发的一个 POJO 类，它并未实现任何接口，比如 IDisPersonInfoIntroduction 接口，但它的 compute(String person)方法同 IDisPersonInfoIntroduction 中的方法类似。

```
public class DisPersonInfoIntroduction {

	private static final Log log = LogFactory.getLog(DisPersonInfoIntroduction.class);

	public void compute(String person) {
		log.info(person);
	}

}
```

为了让 DisPersonInfoIntroduction 实现 IDisPersonInfoIntroduction 接口，我们引入了如下切面。这一@AspectJ 风格的切面借助于@DeclareParents 注解将 IDisPersonInfoIntroduction 接口引入到 test.DisPersonInfoIntroduction 类中，即

DisPersonInfoIntroduction 实现类将实现 IDisPersonInfoIntroduction 接口。开发者是否注意到，这一接口定义了两个 compute()方法，而 DisPersonInfoIntroduction 类仅仅提供了一个方法。一旦 AspectJ 8 将该接口引入到 DisPersonInfoIntroduction 类后，其方法实现由谁提供呢？答案是由 defaultImpl 成员指定的类和 DisPersonInfoIntroduction 类共同完成。如果 DisPersonInfoIntroduction 类实现了这一接口中的任意方法，则这些方法会直接作为这一接口中对应方法的实现；如果接口定义的方法在 DisPersonInfoIntroduction 类中找不到，则直接使用 defaultImpl 成员指定的类所提供的方法实现。

```
@Aspect
public class AopIntroduction {

	private static final Log log = LogFactory.getLog(AopIntroduction.class);

	@DeclareParents(value="test.DisPersonInfoIntroduction", defaultImpl=test.DefaultDisPersonInfoIntroduction.class)
	public IDisPersonInfoIntroduction dpi;

	@Pointcut("execution(* test.*InfoIntroduction.compute(..))")
	public void xx() {
	}

	@Around("xx() && this(dp)")
	public void yy(ProceedingJoinPoint pjp, IDisPersonInfoIntroduction dp)
			throws Throwable {
		log.info("AopIntroduction yy @Around");
		log.info(dp.getClass());
		pjp.proceed();
	}

}
```

上述切面使用了 this 关键字，它将引用到当前连接点的作用对象，从而能够在 yy()方法中打印出作用对象的类型。比如，我们提供了如下 MainTestAspectIntroduction 类，它定义的 idpi 对 dpi 进行了造型。

```
public class MainTestAspectIntroduction {

	private static final Log log = LogFactory.getLog(MainTestAspectIntroduction.class);

	public static void main(String[] args) {
		log.info("MainTestAspectIntroduction main().....");

		DisPersonInfoIntroduction dpi = new DisPersonInfoIntroduction();

		IDisPersonInfoIntroduction idpi = (IDisPersonInfoIntroduction)dpi;

		idpi.compute("王二");
```

```
            idpi.compute("李三", 40);
        }

}
```

一旦运行 MainTestAspectIntroduction 应用，能够浏览到如下输出。很显然，在调用 compute(String persion)方法时，DisPersonInfoIntroduction 响应了这一请求；而在调用 compute(String persion, int age)方法时，DefaultDisPersonInfoIntroduction 响应了这一请求。太完美了!

```
[INFO] 2014-06-30 00:43:58,931 test.MainTestAspectIntroduction - MainTestAspectIntroduction main().....
[INFO] 2014-06-30 00:44:18,604 test.AopIntroduction - AopIntroduction yy @Around
[INFO] 2014-06-30 00:44:18,604 test.AopIntroduction - class test.DisPersonInfoIntroduction
[INFO] 2014-06-30 00:44:18,605 test.DisPersonInfoIntroduction - 王二
[INFO] 2014-06-30 00:44:18,609 test.AopIntroduction - AopIntroduction yy @Around
[INFO] 2014-06-30 00:44:18,609 test.AopIntroduction - class test.DefaultDisPersonInfoIntroduction
[INFO] 2014-06-30 00:44:18,610 test.DefaultDisPersonInfoIntroduction - 李三已经 40 岁了!
```

如果开发者对 DisPersonInfoIntroduction.class 进行反编译，则能够看到 compute(String persion, int age)方法的内容被 AspectJ 8 处理过了，具体如下。当调用 compute(String persion, int age)方法时，DefaultDisPersonInfoIntroduction 将响应这一请求。

```
public void compute(String s, int i)
{
    if(ajc$instance$test_AopIntroduction$test_IDisPersonInfoIntroduction == null)
        ajc$instance$test_AopIntroduction$test_IDisPersonInfoIntroduction = new DefaultDisPersonInfoIntroduction();
    ajc$instance$test_AopIntroduction$test_IDisPersonInfoIntroduction.compute(s, i);
}
```

至于成员变量的动态引入，留给开发者自行研究。

至此，我们已经对 AspectJ 8 的基础知识进行了全方位的介绍，更多内容，开发者可以参考 AspectJ 8 官方网站。我们在介绍 Spring AOP 过程中需要使用到上述知识，甚至会介绍到 AspectJ 8 其他方面的知识。开发者应该认识到，AspectJ 8 通过修改 Java 字节码，而达到实施 AOP 的目的。相比之下，Spring AOP 并不会去修改字节码，而是通过动态代理来实现 AOP。

5.3 Spring AOP 基本概念

Spring Framework 刚出现时，它便采用 AOP 技术实现事务管理。后来，Acegi（Spring Security）也采用 Spring AOP 技术实现安全性控制。如今，开发者可以使用 Spring AOP 声明各种企业级服务，甚至能够实现新的切面。

无论是 Spring 1.x 风格的传统 Spring AOP（简称老式 Spring AOP），还是 Spring

2.x 引入的基于<aop:config/>元素的 POJO 切面和@AspectJ 风格的切面，Spring Framework 始终使用了动态代理实现 AOP 技术。基于<aop:config/>元素的 POJO 切面和@AspectJ 风格的切面仅仅借用到 AspectJ 8 的 pointcut 表达语言。因此，AspectJ 8 和 Spring AOP 进行了无缝集成，它们各自的优点都体现在 Spring AOP 技术当中。

IoC 和 AOP 的关系

如果不启用 Spring AOP 技术，开发者同样可以通过程序手工实现事务管理。但如果启用 Spring AOP 技术，事务管理将变得简单、一致，因为 AOP 技术避免了大量的重复代码，从而降低了 Bug 出现的几率。

如果不启用 Spring IoC 技术,开发者同样可以通过程序手工使用到 AOP 技术。但如果启用 Spring IoC 技术，AOP 技术的使用将变得简单、一致，因为 IoC 提供的 XML 配置管理为 AOP 技术的实施创造了基础条件。

因此，AOP 简化了企业级服务的使用，改善了代码的可复用能力；IoC 通过 XML 配置文件，较好地实施依赖注入，从而为 AOP 创造了基础条件。IoC 和 AOP 的强强联手使得开发企业级应用变得简单、高效，从而大大加深了开发体验，最终交付出客户满意的软件产品。

何谓动态代理？动态代理指在运行时能够动态实现指定接口的机制，从而生成新的类。动态代理是 JDK 1.3 引入的。开发者需要借助 java.lang.reflect.Proxy 类启用动态代理。下面的代码展示了 org.springframework.jdbc.core.JdbcTemplate 类中代理的使用。

```
/**
 * 创建 JDBC 连接的代理
 *
 * @param con  目标 JDBC 连接，供代理使用
 */
protected Connection createConnectionProxy(Connection con) {
        return (Connection) Proxy.newProxyInstance(
                        ConnectionProxy.class.getClassLoader(),
                        new Class[] {ConnectionProxy.class},
                        new CloseSuppressingInvocationHandler(con));
}

/**
 * 调用处理器，它代理了客户对目标对象的一切调用请求
 */
private class CloseSuppressingInvocationHandler implements InvocationHandler {

        private final Connection target;
```

```
    public CloseSuppressingInvocationHandler(Connection target) {
        this.target = target;
    }

    public Object invoke(Object proxy, Method method, Object[] args)throws Throwable{
        //省略了一些无关内容

        //调用目标连接（target）的方法
        try {
            Object retVal = method.invoke(this.target, args);

            //将 Statement 的相关设置应用到 retVal 上(fetch size, max rows, transaction timeout)
            if (retVal instanceof Statement) {
                applyStatementSettings(((Statement) retVal));
            }

            return retVal;
        }
        catch (InvocationTargetException ex) {
            throw ex.getTargetException();
        }
    }

}
```

其中，createConnectionProxy()方法创建了 Connection 对象，即 con 的代理。随后，客户对这一方法返回的“Connection”所进行的任何操作都将经过代理的处理，也就是说代理拦截了客户的所有调用请求。为创建动态代理，需要借助 Proxy 提供的 newProxyInstance()方法，并将定义代理类的类装载器、代理实现的接口集合、InvocationHandler 传入到这一静态方法中。返回的代理实现了 ConnnectionProxy 接口，它继承于 Connection。CloseSuppressingInvocationHandler 对象负责处理客户对目标 con 对象的调用操作。开发者必须为动态代理准备好相应的 InvocationHandler 实现，它内置了代理所要完成的具体逻辑。

对于上述 con 对象而言，我们称为目标对象（Target Object）。由于 Spring Framework 使用运行期代理实现 AOP，因此目标对象就是被代理对象（Proxied Object），或者称之为被装备对象（Advised Object）。

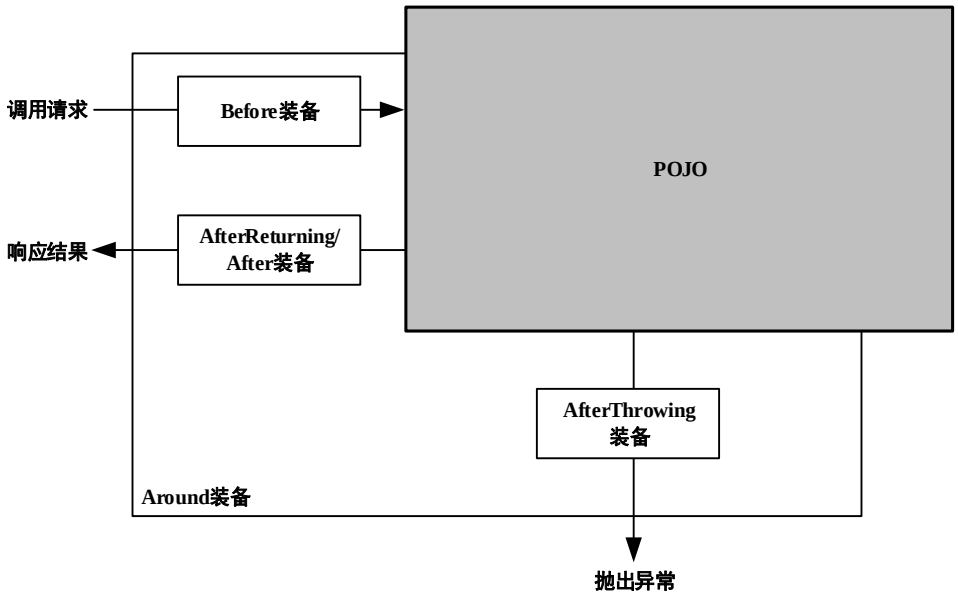
动态代理是 Java 平台本身的重要组成部分，无论是在哪种平台、运行环境、JVM 下，Proxy 的表现都是一致的。Spring AOP 正是看到这种秉承 Java“一次编写、到处运行”的优势，进而基于动态代理实现 Spring AOP。基于前面 AspectJ 8 的研究过程，我们看到开发者必须在编译期借助于静态织入（ajc）完成字节码

的修改操作，或者借助于它的特定工具来实现 Load-Time 期的动态织入（aj5 其实也是对字节码进行了修改）。无论是在调试上，还是开发体验，开发者都更乐意使用 Spring AOP，因为基于动态代理的 AOP 使能应用的开发过程同开发普通的 Java 应用没有任何区别。

与此同时，如果 con 对象（目标对象）并未实现任何接口，则动态代理显然不适合这种场景，因为它要求其操控的目标对象至少实现了单个 Java 接口。幸运的是，除了支持 JDK 风格的动态代理外，Spring AOP 还支持 CGLIB（http://cglib.sourceforge.net/）风格的代理，CGLIB 风格的代理能够代理未实现任何接口的类。值得注意的是，具体的代理类型对于应用而言都是透明的，因为开发者能够随意切换 DI 容器配置元数据而达到启用不同风格代理的目的。

5.3.1 各种装备间的关系

可以看出，动态代理是 Spring AOP 的基础。基于动态代理的 Spring AOP 支持各种类型的装备，Introduction 引入也是其支持的范围。各种装备之间的关系如下图所示，开发者可以通过它深入认识到各种装备的具体含义，而且结合前面 AspectJ 8 的研究过程效果更佳。



图表 26 各种装备间的关系

通常，开发者可以根据自身的需要来合理创建不同类型的装备。基于动态代理实现的 Spring AOP 适合于所有的 Java EE 应用服务器（包括 Web 容器）。它

对目标环境没有任何限制、要求，它不会去控制类装载体系结构、默认的类委派模型。更可喜的是，开发者不用采取特殊的步骤来编译 Spring Framework 使能应用。

Spring Framework 同 AspectJ 8 进行了无缝集成，尤其是 Spring Framework 集成了 AspectJ 8 的 pointcut 表达语言。我们已经知道，AspectJ 8 的连接点模型非常强大，前面只是介绍了同 Spring AOP 相关的连接点内容，其实 AspectJ 8 支持的连接点非常丰富，比如方法、成员变量、调用流程。相应地，Spring Framework 是针对简化企业级 Java 应用的开发而出现的，而 AspectJ 8 并不是源于这一背景。自然地，AspectJ 8 提供了完整的 AOP 实现，而 Spring AOP 更多地是为了解决企业应用中的常见问题，因此 Spring AOP 一直将方法执行连接点作为其主要支持的 JoinPoint。当 Spring AOP 不能够满足应用需求时，建议开发者使用 AspectJ 8 提供的、更强有力的 AOP 支持。

借助 Spring IoC 容器，开发者不仅能够更轻松使用 Spring AOP 技术，甚至可以在 IoC 容器中配置、集成 AspectJ 8 提供的 AOP 支持。

最后，有一点需要开发者重视，Spring AOP 本身是非常强大的，它的可扩展性也非常强。比如，如果需要基于 Spring AOP 实现成员变量连接点支持，开发者借助于 Spring AOP 暴露的相关 API 也能够做到。但是，在实际企业应用中，这种需求到底存在多少呢？Spring Framework 是非常注重平衡的 Java EE 框架，它提供了大量的实用功能。在某种程度上，这其实也是一种敏捷的体现，因为实用使得开发者不用花费精力去选择技术，而是将重点放在如何使用各种实用技术上。

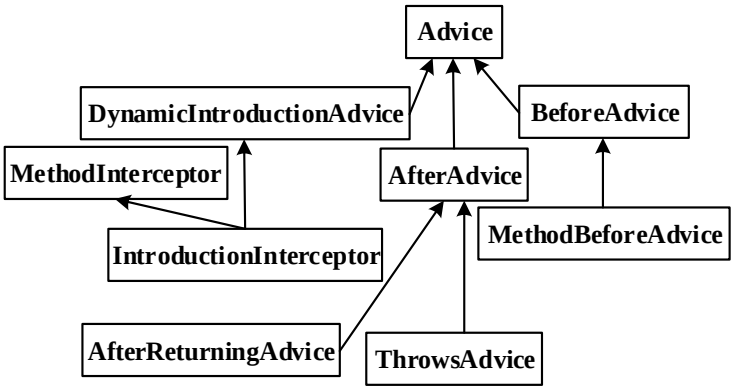
接下来正式进入到 Spring AOP 的研究中，先从老式 Spring AOP 技术入手吧！

5.4 老式 Spring AOP

在 AspectJ 8 中，开发者可以通过切面（aspect 关键字或@Aspect）把装备、pointcut 表达式等信息集结在一起，因此@AspectJ 风格的切面使用起来非常方便。装备和 pointcut 表达式始终是 AOP 领域的两个非常重要的概念，不同 AOP 技术会采用不同机制实施它们。

自 Spring 1.x 开始，开发者可以根据其内置的各种 Advice API 来定义装备。

Spring AOP 实现了 AOP 联盟（http://aopalliance.sourceforge.net/）定义的各个 AOP API，具体见下图。



图表 27 Advice 及相关接口的关系

开发实际应用期间，各自的业务需求存在很大的差异性，因此开发者需要灵活选用 Spring AOP 装备。从上图可以看出：

- MethodBeforeAdvice 用于实现 Before 装备（@Before）。
- AfterReturningAdvice 用于实现 AfterReturning 装备（@AfterReturning）。
- ThrowsAdvice 用于实现 AfterThrowing 装备（@AfterThrowing）。
- MethodInterceptor 可供实现 Around 装备（@Around）使用。
- IntroductionInterceptor 用于实现引入。Spring AOP 将引入（Introduction）也看待成一种装备。

与此同时，为定义和集结连接点，我们需借助 Spring Framework 内置的 NameMatchMethodPointcut、JdkRegexpMethodPointcut 等辅助类定义 pointcut。

最后，开发者通过 Advisor 对象将装备、Pointcut 对象集成在一起，进而为 ProxyFactoryBean 服务。

本节将结合 springaopclassicdemo 项目展开对各种装备、pointcut 表达式等内容的研究。

5.4.1 Before 装备

Before 装备需要实现 MethodBeforeAdvice 接口，其定义如下。开发者是否注意到，before()方法的返回类型为 void，即 Before 装备不能够改变返回值。

```
public interface MethodBeforeAdvice extends BeforeAdvice {
```

```
    /**
     * 回调方法
     *
     * @param method 待调用的目标方法
     * @param args 传入目标方法的参数
     * @param target 目标对象，可能为 null
     *
     * @throws Throwable 如果希望放弃对目标方法的调用，则可以抛出异常
     */
    void before(Method method, Object[] args, Object target) throws Throwable;

}
```

LoggingBeforeAdvice 装备的示例如下。

```
public class LoggingBeforeAdvice implements MethodBeforeAdvice {

    private static final Log log = LogFactory.getLog(LoggingBeforeAdvice.class);

    public void before(Method method, Object[] args, Object target)
            throws Throwable {
        log.info(method + "," + args[0] + "," + target);
        log.info("before: The Invocation of getContent()");

    }

}
```

很显然，Before 装备需要实现 before()方法，该方法将在调用目标操作前被触发。这很适于那些有安全性要求的场合，即在调用目标操作前检查客户的身份，或者用于记录客户调用日志。同时，开发者还能够修改传入到目标操作的参数，即修改 args 参数。为将上述 LoggingBeforeAdvice 装备配置到 DI 容器中，开发者还需要提供 before.xml 文件，具体如下。

```
<bean id="loggingBeforeAdvice" class="test.LoggingBeforeAdvice"/>

<bean id="pointcut" class="org.springframework.aop.support.NameMatchMethodPointcut"
    p:mappedName="getContent"/>

<bean id="loggingBeforeAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor"
    p:advice-ref="loggingBeforeAdvice" p:pointcut-ref="pointcut"/>

<bean id="helloworldbeanTarget" class="test.HelloWorld"/>

<bean id="helloworldbean" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:proxyInterfaces="test.IHelloWorld" p:target-ref="helloworldbeanTarget">
    <property name="interceptorNames">
        <list>
            <idref bean="loggingBeforeAdvisor"/>
        </list>
    </property>
```

```
</bean>
```

关于这一 XML 配置文件，如下两点细节值得开发者注意。

首先，开发者声明装备同普通受管 Bean 的过程是相同的，比如上述定义了 loggingBeforeAdvice 装备。随后，借助 NameMatchMethodPointcut 辅助类定义了 pointcut，它暴露了 mappedName 和 mappedNames 变量，它们用于指定匹配的目标方法集合。此时，mappedName 或 mappedNames 中给出的取值要同目标方法进行匹配，比如 getContent。如果将 mappedName 取值定为*getContent，则所有以 getContent 结尾的方法将成为连接点；如果将 mappedName 取值定为 getContent*，则所有以 getContent 开始的方法将成为连接点；如果将 mappedName 取值定为*getContent*，则所有含有 getContent 的方法将成为连接点。

其次，为了将 pointcut 和 Before 装备集结在一起，我们使用了 DefaultPointcutAdvisor，这是 Spring Framework 提供的辅助对象。同@AspectJ 风格相比，开发者可以在@Before 注解中给定 pointcut 定义或引用，因此 AspectJ 8 在集结 pointcut 和装备时更具优势。Advisor 将单个装备和 pointcut 表达式集结在一起，从而构成了一特殊的切面。通过 DefaultPointcutAdvisor 暴露的 advice 和 pointcut 属性，开发者能够分别引用装备和 pointcut 定义。可以看出，借助 Spring IoC 实施 AOP 是非常方便的，因为开发者能够动态切换 AOP 中的不同组成部分，期间不用修改 Java 源代码。

现在，我们可以一同比较@AspectJ 风格的切面定义和老式 Spring AOP 切面定义。在 AspectJ 8 中，开发者可以将所有相关内容定义在单个应用了@Aspect 注解的 Java 类中，而老式 Spring AOP 切面的定义却比较烦琐。但老式 Spring AOP 使能应用的开发过程同普通 Java 应用的开发并无区别，这正是它的优势所在。

下面给出了基于 before.xml 的 test.HelloClientBefore 应用。

```
public static void main(String[] args) {

    //开发者可以动态切换到不同的 XML 配置文件
    ListableBeanFactory factory = new ClassPathXmlApplicationContext("before.xml");

    IHelloWorld hw = (IHelloWorld)factory.getBean("helloworldbean");

    log.info(hw.getContent("worldheart"));

}
```

一旦运行上述应用，开发者能够浏览到如下类似输出信息。它将连接点、传

入参数、目标对象信息打印出来，并触发了对装备的调用操作。

```
[INFO] 2014-06-30 10:12:29,909 test.LoggingBeforeAdvice –
        public abstract java.lang.String test.IHelloWorld.getContent(java.lang.String),worldheart,test.HelloWorld@4abdb505
[INFO] 2014-06-30 10:12:29,909 test.LoggingBeforeAdvice - before: The Invocation of getContent()
[INFO] 2014-06-30 10:12:29,909 test.HelloWorld - worldheart
[INFO] 2014-06-30 10:12:29,909 test.HelloClientBefore - worldheart
```

5.4.2 基于 ProxyFactoryBean 的手工代理

继续介绍 Spring AOP 支持的其他装备前，有必要仔细研究 ProxyFactoryBean。借助它，开发者能够实现手工代理。

开发 AspectJ 8 使能应用时，开发者需要借助 ajc 命令完成切面的织入工作，从而修改掉原有的 Java 字节码。同 AspectJ 8 相比，Spring AOP 是基于动态代理实现 AOP 技术的，因此借助 ProxyFactoryBean 能够完成切面的织入工作。

通过指定 ProxyFactoryBean 的 proxyInterfaces 属性，开发者能够控制代理实现的接口集合。如果目标类实现了多个接口，则借助 proxyInterfaces 能够控制代理暴露的具体接口，但 proxyInterfaces 属性取值的编辑却很容易出错，因为它是 Class[]数组类型。相比之下，ProxyFactoryBean 暴露的 autodetectInterfaces 属性能够自动探测出目标类实现的接口集合，进而不用开发者显式给定接口集合，示例如下。注意，autodetectInterfaces 属性默认取值是 true，因此这一属性可以不用显式给出。

```
<bean id="helloworldbean" class="org.springframework.aop.framework.ProxyFactoryBean"
        p:target-ref="helloworldbeanTarget">
        <property name="interceptorNames">
                <list>
                        <idref bean="loggingBeforeAdvisor"/>
                </list>
        </property>
        <property name="autodetectInterfaces" value="true"/>
</bean>
```

如果目标类没有实现任何接口，则 Spring AOP 会启用 CGLIB 代理。CGLIB 会采取继承的方式来完成代理的生成。

借助 ProxyFactoryBean 暴露的 targetName 属性能够指定目标受管 Bean 的名字，从而不用使用 target 属性直接引用到目标受管 Bean 本身，示例如下。target 或 targetName 都是用于指定被装备对象的重要属性，开发者可自由选用它们。

```
<bean id="helloworldbean" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="targetName">
                <idref bean="helloworldbeanTarget"/>
        </property>
```

```
        <property name="interceptorNames">
            <list>
                <idref bean="loggingBeforeAdvisor"/>
            </list>
        </property>
</bean>
```

如果开发者未指定 targetName 或 target 属性，则需要将目标对象（即受管 Bean）的名字声明在 interceptorNames 取值中。注意，一定要放到最后一项内容中，示例如下。

```
<bean id="helloworldbean" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>test.IHelloWorld</value>
        </property>
        <property name="interceptorNames">
            <list>
                <idref bean="loggingBeforeAdvisor"/>
                <idref bean="helloworldbeanTarget"/>
            </list>
        </property>
</bean>
```

另外，我们在 before.xml 中将 pointcut 单独定义在一个对象中。老式 Spring AOP 的 pointcut 表达能力其实很一般，将 pointcut 单独定义出来并没有获得太多的价值。因此，是否存在其他简单可用方案供我们选择呢？

NameMatchMethodPointcutAdvisor 和 RegexpMethodPointcutAdvisor，这两个辅助对象会自动处理 Pointcut 的实例化工作，这是不同于 DefaultPointcutAdvisor 的重要方面。下面给出了 NameMatchMethodPointcutAdvisor 使用示例。此时，开发者只需要配置单个对象，便完成了 pointcut 表达式和 Advisor 的声明。NameMatchMethodPointcutAdvisor 底层会采纳 NameMatchMethodPointcut 表示 pointcut。

```
<bean id="loggingBeforeAdvisor" class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor"
        p:advice-ref="loggingBeforeAdvice" p:mappedName="getContent"/>
```

RegexpMethodPointcutAdvisor 是比 NameMatchMethodPointcutAdvisor 更强大的辅助对象，其示例如下。因为开发者能够启用正则表达式（java.util.regex）表示 pointcut。RegexpMethodPointcutAdvisor 底层会采纳 JdkRegexpMethodPointcut 表示 pointcut。

```
<bean id="loggingBeforeAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
        p:advice-ref="loggingBeforeAdvice" p:pattern=".*"/>
```

5.4.3 AfterReturning 装备

AfterReturning 装备同 Before 装备很类似。当正常执行完目标操作后，AfterReturning 装备中的 afterReturning()方法会被触发，而 Before 装备在调用目标操作前执行装备中的 before()方法。为实现 AfterReturning 装备，开发者需要实现 AfterReturningAdvice 接口，其定义摘录如下。

```
/**
 * 只有正常执行方法时，才会触发@AfterReturning 装备的执行
 *
 * 这一装备能够看到返回值，但没有机会修改它
 */
public interface AfterReturningAdvice extends AfterAdvice {

        /**
         * 回调方法
         *
         * @param returnValue 返回的执行结果
         * @param method 被调用方法
         * @param args 传入目标方法的参数
         * @param target 目标方法所在的目标对象，可能为 null
         * @throws Throwable 抛出的异常
         */
        void afterReturning(Object returnValue, Method method, Object[] args,
                        Object target) throws Throwable;

}
```

LoggingAfterReturningAdvice 示例实现如下。

```
public class LoggingAfterReturningAdvice implements AfterReturningAdvice {

        private static final Log log =
                LogFactory.getLog(LoggingAfterReturningAdvice.class);

        public void afterReturning(Object returnValue, Method method,
                        Object[] args, Object target) throws Throwable {
                log.info(returnValue);
                log.info(method + "," + args[0] + "," + target);
                log.info("after: The Invocation of getContent()");
        }

}
```

下面摘录了 afterreturning.xml 的内容，它使用到 LoggingAfterReturningAdvice 装备。

```
<bean id="loggingAfterReturningAdvice" class="test.LoggingAfterReturningAdvice"/>

<bean id="loggingAfterReturningAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
        p:advice-ref="loggingAfterReturningAdvice" p:pattern=".*"/>
```

```
<bean id="helloworldbeanTarget" class="test.HelloWorld"/>

<bean id="helloworldbean" class="org.springframework.aop.framework.ProxyFactoryBean"
        p:proxyInterfaces="test.IHelloWorld" p:target-ref="helloworldbeanTarget">
        <property name="interceptorNames">
                <list>
                        <idref bean="loggingAfterReturningAdvisor"/>
                </list>
        </property>
</bean>
```

如果运行基于 afterreturning.xml 的 HelloClientAfterReturning 应用示例，开发者能够浏览到如下类似日志。

```
[INFO] 2014-06-30 11:02:32,087 test.HelloWorld - worldheart
[INFO] 2014-06-30 11:02:32,087 test.LoggingAfterReturningAdvice - worldheart
[INFO] 2014-06-30 11:02:32,087 test.LoggingAfterReturningAdvice –
        public abstract java.lang.String test.IHelloWorld.getContent(java.lang.String),worldheart,test.HelloWorld@23d2a7e8
[INFO] 2014-06-30 11:02:32,087 test.LoggingAfterReturningAdvice - after: The Invocation of getContent()
[INFO] 2014-06-30 11:02:32,087 test.HelloClientAfterReturning - worldheart
```

5.4.4 AfterThrowing 装备

AfterThrowing 装备对于处理异常，或者特定的业务需求很有帮助，比如某应用抛出了 OptimisticLockingFailureException 乐观锁异常，此时目标应用会考虑重做目标操作。为实现 AfterThrowing 装备，开发者需要实现 ThrowsAdvice 接口，其定义摘录如下，subclass 参数用于限定装备感兴趣的异常类型。

```
/**
 * 标记目标类，将它标识为@AfterThrowing 装备
 *
 * 这一接口并未定义任何方法，
 * 装备实现者必须以如下形式实现 afterThrowing 方法
 *
 * afterThrowing([Method], [args], [target], Throwable subclass)
 *
 * 注意，前 3 个参数都是可选的
 */
public interface ThrowsAdvice extends AfterAdvice {

}
```

针对 AfterThrowing 装备的特殊需要，我们开发了 IHelloWorld 的另一版本，示例如下。

```
public class HelloWorldThrowing implements IHelloWorld {

    private static final Log log = LogFactory.getLog(HelloWorldThrowing.class);

    public String getContent(String helloworld) {
        log.info(helloworld);
        if(true)
```

```
            throw new RuntimeException("getContent");
        return helloworld;
    }

}
```

LoggingAfterThrowingAdvice 示例如下，如果需要针对 subclass 处理相关异常，则可以给出相应的代码逻辑。开发者还可以根据需要将 Throwable subclass 中的 Throwable 修改成其他的具体异常类型。甚至，开发者还可以在同一 ThrowsAdvice 实现类中提供若干不同的 afterThrowing()方法。

```
public class LoggingAfterThrowingAdvice implements ThrowsAdvice {

    private static final Log log = LogFactory.getLog(LoggingAfterThrowingAdvice.class);

    public void afterThrowing(Method method, Object[] args, Object target,
            Throwable subclass) {
        log.info(method + "," + args[0] + "," + target);
        log.info(subclass);
    }

}
```

下面摘录了 afterthrowing.xml 的内容，它使用到 LoggingAfterThrowingAdvice 装备。

```
<bean id="loggingAfterThrowingAdvice" class="test.LoggingAfterThrowingAdvice"/>

<bean id="loggingAfterThrowingAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
    p:advice-ref="loggingAfterThrowingAdvice" p:pattern=".*"/>

<bean id="helloworldbeanTarget" class="test.HelloWorldThrowing"/>

<bean id="helloworldbean" class="org.springframework.aop.framework.ProxyFactoryBean"
    p:proxyInterfaces="test.IHelloWorld" p:target-ref="helloworldbeanTarget">
    <property name="interceptorNames">
        <list>
            <idref bean="loggingAfterThrowingAdvisor"/>
        </list>
    </property>
</bean>
```

如果运行基于 afterthrowing.xml 的 HelloClientAfterThrowing 应用示例，开发者能够浏览到如下类似日志。

```
[INFO] 2014-06-30 11:03:59,336 test.HelloWorldThrowing - worldheart
[INFO] 2014-06-30 11:03:59,336 test.LoggingAfterThrowingAdvice - public abstract java.lang.String test.IHelloWorld.getContent(java.lang.String),worldheart,test.HelloWorldThrowing@1500955a
[INFO] 2014-06-30 11:03:59,336 test.LoggingAfterThrowingAdvice - java.lang.RuntimeException: getContent
Exception in thread "main" java.lang.RuntimeException: getContent
    at test.HelloWorldThrowing.getContent(HelloWorldThrowing.java:26)
```

5.4.5 Around 装备

Around 装备功能很强大，灵活性也最好，它能够在调用目标操作前后执行，因此这对于需要完成资源初始化、释放操作的应用而言特别有用。为实现@Around 装备，开发者需要实现 org.aopalliance.intercept.MethodInterceptor 接口，其定义如下。期间，invocation 对象封装了方法调用信息。

```
public interface MethodInterceptor extends Interceptor {

    /**
     * 在调用目标方法前后进行其他操作
     *
     * @param invocation  方法调用连接点
     *
     * @return Object  调用结果
     *
     * @throws Throwable  拦截器或目标对象抛出的异常
     */
    Object invoke(MethodInvocation invocation) throws Throwable;

}
```

下面给出 LogggingAroundAdvice 示例实现。此时，开发者不仅能控制传入到目标方法的参数取值，还能控制目标方法的执行结果。

```
public class LoggingAroundAdvice implements MethodInterceptor {

    private static final Log log = LogFactory.getLog(LoggingAroundAdvice.class);

    public Object invoke(MethodInvocation invocation) throws Throwable {
        log.info("before: The Invocation of getContent()");
        log.info(invocation);

        invocation.getArguments()[0] = "world heart";
        Object returnValue = invocation.proceed();

        log.info("after: The Invocation of getContent()");

        return returnValue;
    }

}
```

开发者是否注意到@Around 装备同其他装备的区别，我们必须调用 invocation 对象的 proceed()方法，否则目标方法的调用操作不会被触发。另外，invoke()方法的返回值将作为目标方法的执行结果返回给调用者。开发者能够在 invoke()方法中完成任何操作，这正体现了这一装备的灵活性。

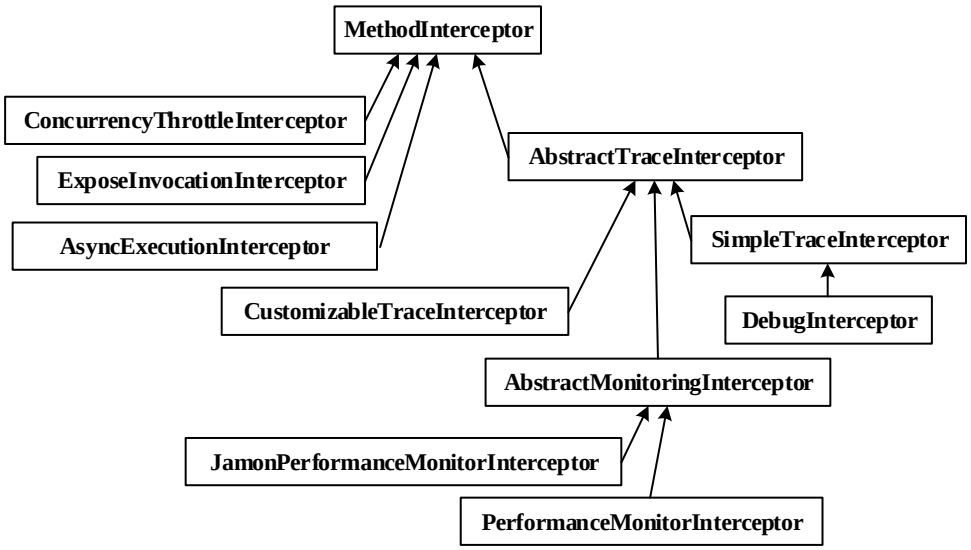
下面摘录了 around.xml 的内容，它使用到 LoggingAroundAdvice 装备。

```
<bean id="loggingAroundAdvice" class="test.LoggingAroundAdvice"/>

<bean id="loggingAroundAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
        p:advice-ref="loggingAroundAdvice" p:pattern=".*"/>

<bean id="helloworldbeanTarget" class="test.HelloWorld"/>

<bean id="helloworldbean" class="org.springframework.aop.framework.ProxyFactoryBean"
        p:proxyInterfaces="test.IHelloWorld" p:target-ref="helloworldbeanTarget">
        <property name="interceptorNames">
                <list>
                        <idref bean="loggingAroundAdvisor"/>
                </list>
        </property>
</bean>
```

如果运行基于 around.xml 的 HelloClientAround 应用示例，开发者能够获得如下类似输出结果。

```
[INFO] 2014-06-30 11:07:28,555 test.LoggingAroundAdvice - before: The Invocation of getContent()
[INFO] 2014-06-30 11:07:28,555 test.LoggingAroundAdvice - ReflectiveMethodInvocation: public abstract java.lang.String test.IHelloWorld.getContent(java.lang.String); target is of class [test.HelloWorld]
[INFO] 2014-06-30 11:07:28,555 test.HelloWorld - world heart
[INFO] 2014-06-30 11:07:28,555 test.LoggingAroundAdvice - after: The Invocation of getContent()
[INFO] 2014-06-30 11:07:28,555 test.HelloClientAround - world heart
```

Spring Framework 的 org.springframework.aop.interceptor 包提供了大量的 Around 装备实现，如下图所示，它们在调试、监控应用性能方面能起到一定作用。本书会介绍到其中的一些装备。在 Spring AOP 实现中，我们会将装备作为一种拦截器（Interceptor）看待，尤其是@Around 装备，或者说，Spring AOP 通过拦截器的形式实现了 AOP 技术。因此，本书会在很多场合使用到拦截器这一术语。开发者可以将@Around 装备和拦截器划等号。



图表 28 aop.interceptor 包提供的 Around 装备实现

5.4.6 Introduction 引入

下面给出了 IntroductionInterceptor 接口的定义。

```
public interface IntroductionInterceptor extends MethodInterceptor, DynamicIntroductionAdvice {

}
```

同其他装备相比，老式 Spring AOP 中引入的开发较麻烦。考虑到这一点后，Spring 提供了 DelegatingIntroductionInterceptor 辅助类，基于它能够快速开发出 Introduction 装备。

实际企业应用中，引入的开发、使用并不常见。当然，还是有不少场合会使用到引入的，比如在借助 JMX 管理运行环境中的各种受管 Bean 实例时，就可以借助引入往现有对象中新增接口，从而达到统一管理的目的。

与此同时，如果应用确实需要使用 Introduction 装备，则可以考虑使用@AspectJ 风格和基于<aop:config/>元素风格的引入装备，因为基于它们的实现更简单、引入 Bug 的概率更小。

5.4.7 使用自动代理特性

前面的内容中，我们直接使用底层的 ProxyFactoryBean 来显式地（手工）创建 AOP 代理。Spring Framework 允许 AOP 开发者隐式地创建 AOP 代理。比如，借助于 BeanNameAutoProxyCreator 对象、DefaultAdvisorAutoProxyCreator 对象，

开发者能够隐式（自动）创建 AOP 代理。甚至，开发者还可以根据 Annotation 元数据来隐式创建 AOP 代理，比如支持@AspectJ 风格的 Spring AOP。

BeanNameAutoProxyCreator 能够根据装备、Advisor 对象自动创建 AOP 代理。比如，autoproxy.xml 配置了如下部分内容。如果开发者在 interceptorNames 中指定了装备（比如 loggingAroundAdvice），则目标对象的所有业务方法都将成为这一装备的连接点。它会为 beanNames 属性指定的<bean/>集合自动创建 AOP 代理，这一属性取值可以使用“*”通配符，多个取值间采用逗号隔开，比如“helloworldbean1,helloworldbean2”。

```
<bean id="simpleTraceAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <bean class="test.SimpleTraceInterceptorOrdered"/>
        </property>
        <property name="pattern" value=".*"/>
</bean>

<bean id="loggingAroundAdvice" class="test.LoggingAroundAdviceOrdered"/>

<bean id="helloworldbean1" class="test.HelloWorld"/>

<bean id="helloworldbean2" class="test.HelloWorld"/>

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator"
         p:beanNames="helloworldbean*">
        <property name="interceptorNames">
                <list>
                        <idref local="loggingAroundAdvice"/>
                        <idref local="simpleTraceAdvisor"/>
                </list>
        </property>
</bean>
```

此后，HelloClientAutoProxy 通过 getBean()访问到的 helloworldbean1 和 helloworldbean2 不再是目标对象本身了，而是 AOP 代理。在一些场合，如果应用不希望开发者通过 getBean()访问到目标对象，则可以启用 autoproxy 特性。

从上述 BeanNameAutoProxyCreator 的使用来看，开发者要显式给出被代理<bean/>的名字，甚至要显式指定 Advisor 对象或装备的名字。为了克服这几个局限性，开发者可以使用 DefaultAdvisorAutoProxyCreator。

DefaultAdvisorAutoProxyCreator 能够智能地将 Advisor 对象作用到各个<bean/>中，它克服了 BeanNameAutoProxyCreator 的这一局限性。当然，局限性在某些场合也是一种优势。下面给出了 autoproxy.xml 中的相关内容。

DefaultAdvisorAutoProxyCreator 会将查找到的两个 Advisor 对象自动作用到 helloworldbean1 和 helloworldbean2 上，从而生成 AOP 代理。

```
<bean id="simpleTraceInterceptor" class="org.springframework.aop.interceptor.SimpleTraceInterceptor"/>

<bean id="loggingAroundAdvice" class="test.LoggingAroundAdvice"/>

<bean id="simpleTraceAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice" ref="simpleTraceInterceptor"/>
        <property name="pattern" value=".*"/>
        <property name="order" value="1"/>
</bean>

<bean id="loggingAroundAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice" ref="loggingAroundAdvice"/>
        <property name="pattern" value=".*"/>
        <property name="order" value="2"/>
</bean>

<bean id="helloworldbean1" class="test.HelloWorld"/>

<bean id="helloworldbean2" class="test.HelloWorld"/>

<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>
```

从研发经验看，DefaultAdvisorAutoProxyCreator 的应用非常广泛。

5.4.8 切换代理机制

开发者是否注意到，我们使用 ProxyFactoryBean、BeanNameAutoProxyCreator、DefaultAdvisorAutoProxyCreator 等期间，各种版本的 IHelloWorld 接口实现都遵循了“针对接口编程”这一最佳实践。在实现场景中，一些遗留代码，或者一些设计并未遵循这一原则，这使得 JDK 动态代理不能够派上用场，因为这些类并未实现相应的接口。所幸，这些对象同时支持那些未实现任何接口的实现类的代理工作。

是否注意到，这些对象都暴露了 proxyTargetClass 属性。如果开发者将 proxyTargetClass 取值置为 true，则目标类及其实现的所有接口都将被代理，此时，应用将拿到 CGLIB 代理。开发者可以将 CGLIB 代理造型成目标类和目标类实现的任意接口。相比之下，在使用 JDK 动态代理时，开发者只能够将代理造型成相应的接口。注意，如果被代理对象未实现任何接口，而开发者又没有显式将 proxyTargetClass 的取值置为 true，则 Spring AOP 还是会自动启用 CGLIB 代理。无论是 ProxyFactoryBean，还是其他支持 autoproxy 特性的辅助类都是如此。

下面的示例配置摘自 cglib.xml，它将 proxyTargetClass 属性取值置为 true。

```
<bean id="helloworldbean" class="org.springframework.aop.framework.ProxyFactoryBean"
      p:target-ref="helloworldbeanTarget" p:proxyTargetClass="true">
      <property name="interceptorNames">
            <list>
                  <idref bean="loggingAfterReturningAdvisor"/>
            </list>
      </property>
</bean>
```

HelloClientCGLIB 使用到 cglib.xml 配置文件，其内置了如下代码示例。此时，同一受管 Bean 能够造型成接口和类。

```
IHelloWorld hw1 = gbfa.getBean("helloworldbean");
log.info(hw1.getContent("worldheart"));
HelloWorld hw2 = gbfa.getBean("helloworldbean");
log.info(hw2.getContent("worldheart"));
```

无论是使用 JDK 动态代理，还是 CGLIB 代理，这对于开发者而言是透明的。通过 DI 容器能够灵活切换代理机制，这正是 Spring DI 的优势。注意，启用 CGLIB 代理后，proxyInterfaces 属性会被忽略。

5.4.9 基于 ProxyFactory 的编程代理

开发者也可以借助 ProxyFactory 实现编程式代理。因此，这将不会依赖于 Spring IoC 容器，而只是 Spring AOP 本身，示例如下。可以看出，缺乏 DI 的支持后，AOP 的使用还是很生涩的。

```
public class HelloClient {

    private static final Log log = LogFactory.getLog(HelloClient.class);

    public static void main(String[] args) {
        //创建 ProxyFactory，从而不需要借助 Spring IoC 容器提供控制反转功能
        ProxyFactory factory = new ProxyFactory(new HelloWorld());

        //创建 LoggingAroundAdvice 装备
        Advice advice = new LoggingAroundAdvice();

        //创建 Advisor 对象
        RegexpMethodPointcutAdvisor advisor = new RegexpMethodPointcutAdvisor();
        advisor.setAdvice(advice);
        advisor.setPatterns(new String[]{".*"});

        factory.addAdvisor(advisor);

        //调用业务操作
        IHelloWorld hw = (IHelloWorld) factory.getProxy();
        log.info(hw.getContent("worldheart"));
    }
```

```
}
```

实践证明，大部分企业应用的开发很少使用到编程式代理，因为开发者有更好的选择。

5.5 基于@AspectJ 的 Spring AOP

到目前为止，我们已经讨论完 AspectJ 8 和 Spring AOP 的老式用法。研究和比较它们期间，发现它们各有利弊。如果能够将 AspectJ 8 的 pointcut 表达语言集成到 Spring AOP 中，而仍然采用基于动态代理的 AOP 实现，这势必将打造一流的 AOP 框架。幸运的是，基于@AspectJ 的 Spring AOP 和基于<aop:config/>元素的 Spring AOP 都已经实现了这一设想，本节讨论基于@AspectJ 的 Spring AOP，下节讨论基于<aop:config/>元素的 Spring AOP。

介绍 AspectJ 8 支持的@AspectJ 风格时，我们使用了 ajc 编译器或 Load-Time 机制完成切面的织入工作。自 Spring Framework 2.x 开始，它也支持@AspectJ 风格的切面声明，我们已经从前面的内容领悟到这一风格的优势。此时，Spring Framework 会使用 AspectJ 8 提供的 Jar 库完成 pointcut 的分析和匹配工作，进而完成基于动态代理的切面的织入工作。一旦织入切面后，AOP 运行时不再需要 AspectJ 8 相关 Jar 库，因为 Spring Framework 仍然是基于动态代理的 AOP 实现。

进入 Spring Framework 之前，开发者务必注意到，AspectJ 8 定义了完整的 pointcut 表达语言，这是到目前为止功能最丰富的一种 AOP 语言。Spring Framework 会逐步集成这一 pointcut 表达语言到其 AOP 实现中，因此目前的 Spring Framework 并没有支持 pointcut 表达语言的全部内容。

本节将结合 springaopaspectjdemo 项目展开论述。

5.5.1 声明切面、pointcut 和装备

在 Spring Framework 中，定义@AspectJ 风格的切面同 AspectJ 8 中是一致的，我们已经在前面内容阐述过。比如，本书定义了如下 LoggingBeforeAspect 切面。

```
@Aspect
public class LoggingBeforeAspect {

    private static final Log log = LogFactory.getLog(LoggingBeforeAspect.class);

    //将连接点作用范围圈定在 service 包中
```

```
    @Pointcut("within(*test.service.*)")
    public void with(){}

    //连接点对应的方法的返回值必须是 void 或 java.lang.String
    @Pointcut("execution(public void || String test.service..*(..))")
    public void service(){}

    //进行与运算
    @Before("service() && with() && args(str) && target(obj) && this(proxy)")
    public void beforeAspect(JoinPoint jp, String str, Object obj, Object proxy){
        log.info(jp);
        log.info(str);
        log.info(obj);
        log.info(proxy);
    }

}
```

上述@AspectJ 切面定义了两个 pointcut 表达式，@Before 装备引用到它们，并进行了与运算（&&）。另外，我们还使用了 args、this、target 关键字：args 指传入连接点的参数、this 指代理本身、target 指代理目标。

定义 LoggingBeforeAspect 切面后，开发者需要将它配置成受管 Bean，这是同 AspectJ 8 的区别之处。在 AspectJ 8 中，开发者需要手工借助于 ajc 编译器或 Load-Time 机制完成切面的织入工作。在 Spring Framework 中，开发者只需要将切面定义成相应的受管 Bean，并启用自动代理特性（autoproxy），即可完成切面的织入工作。比如，before.xml 配置示例如下。

```
<aop:aspectj-autoproxy/>

<bean id="helloworldDao" class="test.dao.HelloWorldDaoImpl"
    p:helloworld="hello world"/>

<bean id="helloworldService" class="test.service.HelloWorldServiceImpl"
    p:helloWorldDao-ref="helloworldDao"/>

<bean id="loggingBeforeAspect" class="test.aspect.LoggingBeforeAspect"/>
```

从上述信息可以看出，配置@AspectJ 风格的切面同普通受管 Bean 没有任何区别，而且如果这一切面存在协作者，则它还能够享受到 DI。为启用@AspectJ 风格切面的支持，开发者还需要配置<aop:aspectj-autoproxy/>元素。此后，Spring Framework 会将各@AspectJ 切面织入到具体的受管 Bean 中，比如“helloworldService”受管 Bean 将被动态代理控制，一旦开发者通过 getBean(“helloworldService”)访问受管 Bean，调用者将拿到代理，而不是目标对象。

为了通过<aop:aspectj-autoproxy/>启用自动代理特性，开发者还需要将 aop 命名空间导入到 XML 文件中，详情见 before.xml 的头信息。如果开发者还在使用 DTD，或者打算直接使用<bean/>启用自动代理特性，则可以配置如下受管 Bean 到 before.xml 中。注意，这一<bean/>不能够同<aop:aspectj-autoproxy/>同时出现。无论开发者使用哪种方式启用自动代理特性，Spring Framework 底层会始终使用 AnnotationAwareAspectJAutoProxyCreator。

```
<bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator"/>
```

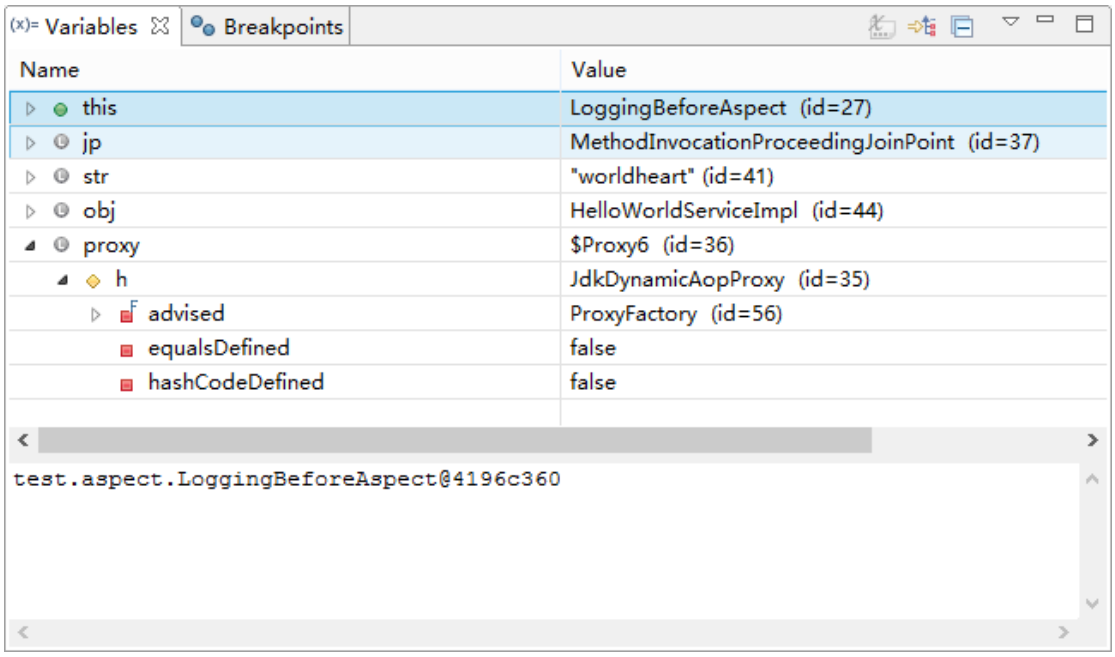
我们可以提供 HelloClientBefore.java 来集成上述各方面的内容，摘录如下。

```
ListableBeanFactory factory = new ClassPathXmlApplicationContext("before.xml");

IHelloWorldService hw = (IHelloWorldService)factory.getBean("helloworldService");

log.info(hw.getContent("worldheart"));
```

运行 test.HelloClientBefore 后，能够从控制台浏览到如下一些日志。JoinPoint 将连接点信息封装在自身中。

```
[INFO] 2014-06-30 12:00:14,183 test.aspect.LoggingBeforeAspect –
        execution(String test.service.IHelloWorldService.getContent(String))
[INFO] 2014-06-30 12:00:14,183 test.aspect.LoggingBeforeAspect - worldheart
[INFO] 2014-06-30 12:00:14,183 test.aspect.LoggingBeforeAspect - test.service.HelloWorldServiceImpl@20d28811
[INFO] 2014-06-30 12:00:14,183 test.aspect.LoggingBeforeAspect - test.service.HelloWorldServiceImpl@20d28811
[INFO] 2014-06-30 12:00:14,183 test.service.HelloWorldServiceImpl - worldheart
[INFO] 2014-06-30 12:00:14,183 test.dao.HelloWorldDaoImpl - worldheart
[INFO] 2014-06-30 12:00:14,183 test.HelloClientBefore - worldheart
```

运行 HelloClientBefore 期间，如果 LoggingBeforeAspect 切面中“log.info(jp);”所在行打一断点，则通过调试能够发现 args、this、target 的真正含义，具体见下图。注意，target 引用的 obj 只是代理目标，而 this 引用的 proxy 却是代理本身。Spring AOP 是基于动态代理的 AOP 实现，图中展示的内容同这一点也是相吻合的。



图表 29 args、this 和 target 的含义

通过上述几个简单步骤，我们完成了一简单的、基于@Before 切面的 Spring AOP 使能应用的开发。同 AspectJ 8 相比，这些步骤显得更为轻松、自然。同开发普通 Spring Framework 使能应用一样，开发@AspectJ 风格的 Spring AOP 应用并无太多的差异性。此时，开发者可能会问，如果不借助 Spring DI 启用 autoproxy 特性，能否通过编程手工实现业务对象的代理呢？答案是可以的，我们提供的 HelloClientBeforeProgram 客户应用正是手工完成了 AOP 创建的相关工作，示例代码如下。可以看出，手工完成代理的创建工作也并非难事。

```
ListableBeanFactory factory = new ClassPathXmlApplicationContext("beforeprogram.xml");

AspectJProxyFactory ajpFactory = new AspectJProxyFactory(factory.getBean("helloworldService"));

ajpFactory.addAspect(LoggingBeforeAspect.class);

IHelloWorldService hw = ajpFactory.getProxy();

log.info(hw.getContent("worldheart"));
```

使用@AspectJ 风格的 Spring AOP 期间，开发者需要将 aspectjweaver.jar 放置在 classpath 路径上。

5.5.2 各种装备的使用

Spring Framework 支持各种类型的装备。下面给出了@AfterReturning 装备的开发示例，它引用到 LoggingBeforeAspect 定义的 pointcut。通过 returning 成员能

够将目标操作的返回结果传入到@AfterReturning 装备中。开发者可以运行相应的 HelloClientAfterReturning 应用示例。

```
@Aspect
public class LoggingAfterReturningAspect {

    private static final Log log = LogFactory.getLog(LoggingAfterReturningAspect.class);

    //引用到 LoggingBeforeAspect 定义的 pointcut
    @AfterReturning(pointcut="LoggingBeforeAspect.service() && LoggingBeforeAspect.
        with()" + " && args(str) && target(obj) && this(proxy) ",returning="retur")
    public void afterReturningAspect(JoinPoint jp, String str, Object obj, Object proxy,Object retur){
        log.info(jp);
        log.info(str);
        log.info(obj);
        log.info(proxy);
        log.info(retur);
    }

}
```

下面给出了@AfterThrowing、@After 装备的开发示例，通过 throwing 成员能够将异常信息传给@AfterThrowing 装备，并采取相应的动作。

```
@Aspect
public class LoggingAfterThrowingAspect {

    private static final Log log =LogFactory.getLog(LoggingAfterThrowingAspect.class);

    //引用到 LoggingBeforeAspect 定义的 pointcut
    @AfterThrowing(pointcut="LoggingBeforeAspect.service() && LoggingBeforeAspect.with()" +
            " && args(str) && target(obj) && this(proxy)",throwing="thro")
    public void afterThrowingAspect(JoinPoint jp, String str, Object obj, Object proxy,
        RuntimeException thro){
        log.info("afterThrowingAspect.....");
        log.info(jp);
        log.info(str);
        log.info(obj);
        log.info(proxy);
        log.info(thro);
    }

    //引用到 LoggingBeforeAspect 定义的 pointcut
    @After("LoggingBeforeAspect.service() && LoggingBeforeAspect.with()" +
            " && args(str) && target(obj) && this(proxy)")
    public void afterAspect(JoinPoint jp, String str, Object obj, Object proxy){
        log.info("afterAspect.....");
        log.info(jp);
        log.info(str);
        log.info(obj);
        log.info(proxy);
    }
```

```
}
```

下面给出了@Around 装备的开发示例。@Around 装备同@Before、@AfterReturning、@AfterThrowing、@After 装备存在很大差别。@Around 是这四种装备的全集，能够在连接点之前、之后执行相应的动作，从而完全控制目标操作的返回值。由于 LoggingAroundAspect 匹配的连接点存在返回值，因此 aroundAspect 方法返回了 Object，开发者可以让它直接返回 String 字符串，因为目标操作会返回 String。

```
@Aspect
public class LoggingAroundAspect {

	private static final Log log = LogFactory.getLog(LoggingAroundAspect.class);

	//传入 str 参数到@Around 装备中
	@Pointcut("execution(public * test.service..*(..)) && args(str)")
	public void service(String str){}

	@Around("service(str)" + " && target(obj) && this(proxy)")
	public Object aroundAspect(ProceedingJoinPoint pjp, String str, Object obj,Object proxy){
		log.info(pjp);
		log.info(str);
		log.info(obj);
		log.info(proxy);
		try {
			str = "worldheart changed!";
			return pjp.proceed(new Object[]{str});
		} catch (Throwable e) {
			log.error("", e);
		}
		return null;
	}

}
```

为了让整个 AOP 拦截器链递进下去，开发者需要调用 pjp 的 proceed()方法。前面在定义四种装备时，我们要使用 JointPoint 接口，而这里使用了它的子接口（ProceedingJoinPoint）。如果开发者不需要修改传入目标操作的参数，则直接调用 pjp.proceed()无参方法即可；如果需要修改传入目标操作的参数，则调用 pjp.proceed(java.lang.Object[])方法。

HelloClientAround 应用示例演示了这一切面的使用情况，运行日志示例如下。可以看出，传入的参数在中途被@Around 装备修改了！

```
[INFO] 2014-06-30 14:23:08,567 test.aspect.LoggingAroundAspect -
    execution(String test.service.IHelloWorldService.getContent(String))
[INFO] 2014-06-30 14:23:08,568 test.aspect.LoggingAroundAspect - worldheart
```

```
[INFO] 2014-06-30 14:23:08,568 test.aspect.LoggingAroundAspect -
    test.service.HelloWorldServiceImpl@28eaa59a
[INFO] 2014-06-30 14:23:08,568 test.aspect.LoggingAroundAspect -
    test.service.HelloWorldServiceImpl@28eaa59a
[INFO] 2014-06-30 14:23:08,568 test.service.HelloWorldServiceImpl -
    worldheart changed!
[INFO] 2014-06-30 14:23:08,568 test.dao.HelloWorldDaoImpl - worldheart changed!
[INFO] 2014-06-30 14:23:08,568 test.HelloClientAround - worldheart changed!
```

Spring AOP 中，Introduction（引入）也是一种装备。借助@DeclareParents 注解能够声明引入，它往往伴随其他装备存在，比如前面讨论的几种装备，下面给出了开发示例。此时，service 包中接口的实现者都将实现 IIntroductionInfo 接口，开发者在通过 getBean(“helloworldService”)获得代理后，可以将这一代理造型成上述接口，从而访问到它定义的各个方法。有兴趣的开发者，可以去研究我们提供的 HelloClientIntroduction 示例。我们已经在介绍 AspectJ 8 过程中详细介绍过引入。

```
@Aspect
public class LoggingIntroductionAspect {

    @DeclareParents(value="test.service.*", defaultImpl=test.IntroductionInfoImpl.class)
    public IIntroductionInfo iInfo;

    ……
}
```

5.5.3 切换代理机制

如果开发者希望启用 CGLIB 代理，则需要使用到<aop:aspectj-autoproxy/>元素的如下属性，并将其置为 true。

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

此时，应用将拿到 CGLIB 代理，而不是动态代理。开发者可以将 CGLIB 代理造型成目标类和目标类实现的任意接口。

应用直接采纳 AnnotationAwareAspectJAutoProxyCreator 对象时，则开发者需要启用它暴露的 proxyTargetClass 属性，并将其置为 true。示例配置如下。

```
<bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator">
        <property name="proxyTargetClass" value="true"/>
</bean>
```

5.5.4 控制各装备的触发顺序

Ordered 接口和@Order 注解，这是两个非常重要的 Spring Framework 对象，前者位于 Spring Framework 框架的 org.springframework.core 包中，而后者位于 org.springframework.core.annotation 包中。依据它们，开发者能够细粒度控制切

面的触发顺序，比如同一连接点匹配上多个装备时。下面给出了@Order 注解的使用。LoggingBeforeAspect1 作用的@Order 注解取值更小，因此它比 LoggingBeforeAspect2 先被触发。

```
@Aspect
@Order(2)
public class LoggingBeforeAspect1 {
	.....
}

@Aspect
@Order(8)
public class LoggingBeforeAspect2 {
	.....
}
```

类似地，下面给出了 Ordered 接口的使用。LoggingBeforeAspect1 实现的 Ordered 接口的 getOrder()方法返回的取值更小，因此它比 LoggingBeforeAspect2 先被触发。

```
@Aspect
public class LoggingBeforeAspect1 implements Ordered {

	public int getOrder() {
		return 2;
	}

}

@Aspect
public class LoggingBeforeAspect2 implements Ordered {

	public int getOrder() {
		return 8;
	}

}
```

Ordered 接口和@Order 注解的使用面非常广，各种风格的 Spring AOP 技术都会直接或间接借用到它们。

5.5.5 pointcut 表达语言

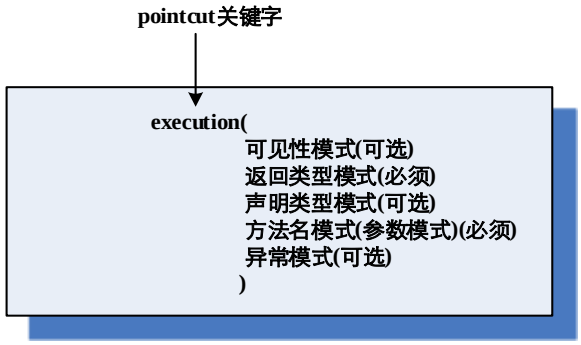
AspectJ 8 提供的 pointcut 表达语言功能非常丰富，而且容易使用。在 AspectJ 5 之前，开发者必须使用 aspect 关键字标识切面，并将 pointcut 定义在它所在的文件中，比如 Xxx.aj。自从 JDK 5.0 推出标准的 Annotation 注解技术以来，企业级 Java 社区开始全面拥抱这一技术，比如 AspectJ 5 就全面拥抱注解了，因此，

它提供了@AspectJ 风格的切面定义。Spring Framework 不仅支持@AspectJ 风格的切面，还扩展了它，直接在 XML 文件中即可声明和引用到 pointcut 表达式。我们将在“基于<aop:config/>元素的 Spring AOP”一节领略到，Spring Framework 是如何通过 XML 声明 pointcut 表达式的。

无论是哪种形式，pointcut 本身的语义始终没有改变。人们常说，光线是摄影的声音。某种程度上，pointcut 表达语言正是 AOP 技术中的声音。也正是意识到 pointcut 的重要性，Spring Framework 集成并扩展了 AspectJ 8 的 pointcut 表达语言。

事实上，我们对 pointcut 表达语言的很多方面已经给出了大量的示例和介绍。在此，我们再来简单研究（温习或总结）同 Spring Framework 相关的 pointcut。

目前，Spring Framework 将精力放在方法级的连接点上，而成员变量级及其他类型的连接点并没有刻意去提供。方法级的连接点能够满足大部分企业级 Java 应用，而这也正是 Spring Framework 所关注的。既然关注点是方法，则 Spring Framework 必然会关注方法的执行操作。我们知道，AspectJ 8 同时提供了 execution 和 call 关键字来集结方法执行（调用），它们的区别我们已经分析过。目前，Spring Framework 仅支持 execution 关键字，它的使用见下图。因此，execution 关键字将是 Spring Framework 开发者使用最频繁的 pointcut 关键字。



图表 30 execution 表达式的语法

某 execution 表达式示例如下。这里的 public 指可见性模式，它指定了目标方法的可见性；第一个*指定了返回类型模式，即各种返回类型（包括 void）；test.service..指定声明类型模式，即 service 包及其子包中的所有 Java 类型；第二个*指定了方法名模式，即所有方法；第二个*后面的..指定了参数模式，即含有

若干参数（包括无参情形），异常模式并没有在这里体现出来。这些模式的具体使用同 Java 语法、语义有直接关系。

```
execution(public * test.service..*(..))
```

上图中各模式的解释如下。

可见性模式。我们知道，Java 类、方法和成员变量都存在可见性，并由 public、protected 和 private 等关键字来标识它们。在 pointcut 表达式中，这些关键字的语义并未改变，它们仍然遵循 Java 的语义。甚至，开发者可以使用“*”，即任意可见性。

返回类型模式。所谓返回类型，即 Java 类型，比如 String、int 等。也可以无返回值，即 void。

声明类型模式。所谓声明类型，即 Java 包，比如 java.lang、test.service 等。

方法名模式。它指定目标操作的方法名，比如 doSomething 和*。

参数模式。它指定目标方法的参数模式，即方法接收的参数数量和类型。

异常模式。它指定目标方法的方法签名中是否存在异常类型的指定。

上述各种模式非常容易理解和掌握，我们可以通过调试、分析各种 pointcut 表达式的具体应用。我们能够在 pointcut 子表达式（比如 execution 表达式）中使用*、..、+等通配符。其中，*表示若干字符（排除.在外），比如单个包、单个参数等；..表示若干字符（包括.在内），比如单个包、多个包、单个参数、多个参数等；+表示子类，比如 Info+表示 Info 类及其子类。

由于 execution 表达式只是 pointcut 表达式的组成部分，因此不同的 pointcut 子表达式之间需要进行运算处理，AspectJ 8 提供了与运算（&&）、或运算（||）、非运算（!）支持。为了定义各种子表达式，开发者需要使用到 pointcut 表达语言暴露的各个 pointcut 关键字。下表对 Spring Framework 涉及到的关键字进行了总结，Spring AOP 支持这些关键字的使用。

pointcut 关键字	具 体 含 义
execution	用于匹配方法执行连接点。它不同于 call 关键字，目前 Spring AOP 不支持 call。开发者经常要同 execution 打交道
within	用于圈定连接点所在的 Java 类型。在 Spring AOP 中，连接点即目标方法，而目标方法所在的 Java 类可以由 within 圈定，比如限制在某个 Java 包中（test.service）。另外，AspectJ 8 还提供了 withincode 关键字，不过目前 Spring AOP 不支持它
@within	用于匹配在类一级应用了特定 Annotation 注解（比如@ForYou）的那些类中的各连接点（方法）
this	在 Spring AOP 中，开发者可以认为 this 引用了 AOP 代理。开发

	者通过 this 关键字能够将 AOP 代理传入到各种装备中
target	在 Spring AOP 中，开发者可以认为 target 引用了被代理对象。开发者通过 target 关键字能够将被代理对象（即受管 Bean）传入到各种装备中
@target	用于指定被代理对象对应的 Java 类型必须在类一级应用了特定的 Annotation，比如@ForYou
args	用于指定传入到连接点（比如，执行方法）的参数，从而供各种装备使用
@args	用于指定传入到连接点的参数对应的 Java 类型必须在类一级应用了特定的 Annotation
@annotation	用于指定连接点（即执行方法）必须应用了方法一级的 Annotation
bean	它扩展了 AspectJ pointcut 语言，用于限定具体的受管 Bean，允许使用“*”通配符。注意，开发者不能够在 AspectJ 使能应用中使用它

图表 31 Spring AOP 的关键字

为了展示上述各种 pointcut 关键字的使用，我们特意提供了 ClassForPointInfo 类，其内容如下，它位于 test.pointcut 包中。

```
@ForYou
public class ClassForPointInfo {

	protected static final Log log = LogFactory.getLog(ClassForPointInfo.class);

	public String getInfo1(){
		return "getInfo1()";
	}

	public String getInfo1(String str){
		return "getInfo1(String str)";
	}

	public String getInfo1(String str, int i) throws RuntimeException{
		return "getInfo1(String str, int i) throws RuntimeException";
	}

	public void setInfo2(AnnotationParam ap){
		log.info(ap);
	}

	@ForYou
	public void setInfo3(){
		;
	}

}
```

下面给出了 ClassForPointInfoClient 客户应用，其大量使用了 pointcut。

```
//within 将连接点作用范围圈定在 test.pointcut 包中
@Pointcut("within(test.pointcut.*)")
public void with(){}
```

```
//连接点对应的方法的返回值必须是 void 或 java.lang.String
@Pointcut("execution(public void || String test.pointcut..*(..))")
public void service1(){}

//连接点对应的方法的返回值必须是 java.lang.String
@Pointcut("execution(public String test.pointcut..*(..) throws RuntimeException)")
public void service2(){}

//service1()进行与运算
@Before("service1() && with() && args(str) && target(obj) && this(proxy)")
public void before1Aspect(JoinPoint jp, String str, Object obj, Object proxy){
        ......
}

//service2()进行与运算
@Before("service2() && with() && target(obj) && this(proxy)")
public void before2Aspect(JoinPoint jp, Object obj, Object proxy){
        ......
}

//@args 表明，传入连接点的参数对应的类必须应用@ForYou 注解（类一级）
@Before("service1() && with() && args(ap) && target(obj) && this(proxy) && @args(test.pointcut.ForYou)")
public void before3Aspect(JoinPoint jp, AnnotationParam ap, Object obj, Object proxy){
        ......
}

//@within 表明，只有那些在类一级应用了@ForYou 注解的类中的各方法才能够成为连接点候选
@Before("service1() && with() && @within(test.pointcut.ForYou) && target(obj) && this(proxy)")
public void before4Aspect(JoinPoint jp, Object obj, Object proxy){
        ......
}

//@annotation 表明，被代理对象中的连接点（即目标方法）必须应用了@ForYou 注解（方法一级）
@Before("service1() && with() && @annotation(test.pointcut.ForYou) && target(obj) && this(proxy)")
public void before5Aspect(JoinPoint jp, Object obj, Object proxy){
        ......
}

//@target 表明，被代理对象必须在类一级应用了@ForYou 注解
@Before("service1() && with() && @target(test.pointcut.ForYou) && target(obj) && this(proxy)")
public void before6Aspect(JoinPoint jp, Object obj, Object proxy){
        ......
}

//PointInfo+表明，所有以 PointInfo 结尾的类及子类
@Before("within(test.pointcut.*) &&  execution(* test..*PointInfo+.*(..) throws RuntimeException)")
public void before7Aspect(JoinPoint jp){
        ......
}

//service2()进行与运算，另外限定受管 Bean 的名字
@Before("bean(classForPointInfo*) && service2()")
```

```
public void before8Aspect(JoinPoint jp){
        ......
}
```

有关 pointcut 表达语言的完整内容，请开发者参考 AspectJ 8 相关文档，比如《AspectJ Programming Guide》。AspectJ 8 官方文档对 pointcut 表达语言进行了全方位的介绍，希望 Spring Framework 开发者都能够仔细阅读《AspectJ Programming Guide》和《The AspectJ 5 Development Kit Developer's Notebook》。

5.6 基于<aop:config/>元素的 Spring AOP

使用@AspectJ 风格的切面时，由于使用了 Annotation 技术，因此 JDK 5.0+是必需的。如今，实际的企业开发环境中还可能在使用 JDK 1.4，甚至更低版本。AspectJ 8 提供的 pointcut 表达语言是非常好用的，功能又很强大。我们是否可以在不定义@AspectJ 风格切面的前提下，同样使用到 pointcut 表达语言呢？是的，基于 XML Schema 的 Spring Framework 配置文件允许开发者这么做。Spring Framework 引入了 aop 命名空间，我们在之前使用了 aop 命名空间中的一些元素，比如<aop:aspectj-autoproxy/>。这里将使用到另一<aop:config/>元素，基于这一元素，开发者能够定义 pointcut、advisor、切面、更改代理机制等。

相关内容基于 springaopschemademo 项目展开，它同 springaopaspectjdemo 项目非常类似。不过，这里的 springaopschemademo 项目是通过 XML 文件定义 pointcut 表达式的，而 springaopaspectjdemo 项目是将 pointcut 表达式直接定义在@AspectJ 切面中。

5.6.1 声明切面、pointcut 和装备

不难想象，<aop:config/>元素只是@AspectJ 的另一种形式。比如，我们在 before.xml 配置文件中定义了如下内容。

```
<bean id="helloworldDao" class="test.dao.HelloWorldDaoImpl"
        p:helloworld="hello world"/>

<bean id="helloworldService" class="test.service.HelloWorldServiceImpl"
        p:helloWorldDao-ref="helloworldDao"/>

<bean id="loggingBeforeAspect" class="test.aspect.LoggingBeforeAspect"/>

<aop:config>
        <aop:pointcut id="before"
                expression="execution(public void or String test.service..*(..))
```

```
                and args(str) and target(obj) and this(proxy)"/>
    <aop:aspect ref="loggingBeforeAspect">
        <aop:before pointcut-ref="before" method="beforeAspect"/>
    </aop:aspect>
</aop:config>
```

这里的 LoggingBeforeAspect 只是一普通的 POJO 对象，代码摘录如下，其中并未应用@Aspect 注解，但它内置的方法集合仍然可被作为装备看待。<aop:pointcut/>元素用于指定 pointcut 表达式，以供其他元素引用，比如<aop:aspect/>元素。借助<aop:before/>元素，我们能够定义出@Before 类型的装备，比如 LoggingBeforeAspect 内置的 beforeAspect 方法。

```
public class LoggingBeforeAspect {

    private static final Log log = LogFactory.getLog(LoggingBeforeAspect.class);

    public void beforeAspect(JoinPoint jp, String str,
        Object obj, Object proxy){
        log.info(jp);
        log.info(str);
        log.info(obj);
        log.info(proxy);
    }

}
```

注意，pointcut 表达式可以直接通过<aop:pointcut/>元素进行指定，从而实现装备与 pointcut 表达式的剥离。通过<aop:pointcut/>暴露的 id 属性能够指定 pointcut 表达式的唯一名称，而 expression 属性取值同@AspectJ 风格的 pointcut 表达式是一样的，只不过&&和||被替换成 and（与运算）和 or（或运算）罢了。注意，XML 文件对“&&”感冒，因此在 XML 中定义 pointcut 表达式时，开发者要注意这一区别。如果需要在 pointcut 表达式中进行非运算（!），则可以用 not 代替。

上述<aop:config/>定义中，我们借助<aop:pointcut/>元素直接定义了 pointcut 表达式。试想，如果开发者在使用 JDK 5.0+，而且希望上述 XML 文件使用到定义在@AspectJ 切面中的 pointcut 表达式，Spring AOP 是否满足这一场景呢？是的，确实可以做到。比如，我们定义了如下切面，它仅仅定义了一个 Pointcut。

```
@Aspect
public class LoggingBeforeAspectAspectJ {

    @Pointcut("execution(public void || String test.service..*(..))")
    public void before(){};
```

```
}
```

下面一段配置引用到上述 pointcut 定义，它最终达到了同直接在 expression 属性中定义 pointcut 表达式一样的效果。尽管开发者可以在<aop:config/>中定义若干<aop:pointcut/>，但这种重用仅仅局限于单个<aop:config/>中，如果存在多个<aop:config/>元素，怎么办呢？因此，将通用的 pointcut 定义在@AspectJ 切面中还是存在不少优势的。

```
<aop:config>
    <aop:pointcut id="before"
        expression="test.aspect.LoggingBeforeAspectAspectJ.before() and args(str) and target(obj) and this(proxy)"/>
    <aop:aspect ref="loggingBeforeAspect">
        <aop:before pointcut-ref="before" method="beforeAspect"/>
    </aop:aspect>
</aop:config>
```

5.6.2 各种装备的使用

Spring Framework 中的<aop:config/>元素支持各种类型的装备，比如@Before、@AfterReturning、@AfterThrowing、@After、@Around、引入（Introduction）。对应的 XML 元素有<aop:before/>、<aop:after-returning/>、<aop:after-throwing/>、<aop:after/>、<aop:around/>、<aop:declare-parents/>。@Before 装备已经研究过，接下来研究其他装备。

@AfterReturning 装备对应于<aop:after-returning/>元素，它在<aop:before/>元素基础上引入了 returning 属性，这一属性用于指定返回结果的取值。

比如，LoggingAfterReturningAspect POJO 定义示例如下。

```
public class LoggingAfterReturningAspect {

    private static final Log log = LogFactory.getLog(LoggingAfterReturningAspect.class);

    public void afterReturningAspect(JoinPoint jp, Object retur){
        log.info(jp);
        log.info(retur);
    }

}
```

下面展示了 LoggingAfterReturningAspect 及<aop:after-returning/>元素的使用，相关内容摘自 afterreturning.xml。

```
<bean id="loggingAfterReturningAspect" class="test.aspect.LoggingAfterReturningAspect"/>

<aop:config>
    <aop:pointcut id="afterRe"
        expression="execution(public void or String test.service..*(String))"/>
```

```
    <aop:aspect ref="loggingAfterReturningAspect">
        <aop:after-returning pointcut-ref="afterRe" method="afterReturningAspect" returning="retur"/>
    </aop:aspect>
</aop:config>
```

@AfterThrowing 装备对应于<aop:after-throwing/>元素，它在<aop:before/>元素基础上引入了 throwing 属性，这一属性用于指定抛出的异常。

比如，LoggingAfterThrowingAspect POJO 装备定义示例如下。

```
public class LoggingAfterThrowingAspect {

    private static final Log log = LogFactory.getLog(LoggingAfterThrowingAspect.class);

    public void afterThrowingAspect(JoinPoint jp, RuntimeException thro){
        log.info(jp);
        log.info(thro);
    }

}
```

下面展示了 LoggingAfterThrowingAspect 及<aop:after-throwing/>元素的使用，相关内容摘自 afterthrowing.xml。

```
<bean id="helloworldDao" class="test.dao.HelloWorldDaoThrowingImpl">
    <property name="helloworld" value="hello world"/>
</bean>

<bean id="helloworldService" class="test.service.HelloWorldServiceImpl">
    <property name="helloWorldDao" ref="helloworldDao"/>
</bean>

<bean id="loggingAfterThrowingAspect" class="test.aspect.LoggingAfterThrowingAspect"/>

<aop:config>
    <aop:pointcut id="afterThro"
        expression="execution(public void or String test.service..*(String))"/>
    <aop:aspect ref="loggingAfterThrowingAspect">
        <aop:after-throwing pointcut-ref="afterThro" method="afterThrowingAspect" throwing="thro"/>
    </aop:aspect>
</aop:config>
```

@After 装备对应于<aop:after/>元素，它同<aop:before/>元素的定义完全一致。

比如，LoggingAfterAspect POJO 装备定义示例如下。

```
public class LoggingAfterAspect {

    private static final Log log = LogFactory.getLog(LoggingAfterAspect.class);

    public void afterAspect(JoinPoint jp, String str){
        log.info(jp);
        log.info(str);
    }
```

```
}
```

下面展示了 LoggingAfterAspect 及<aop:after/>元素的使用，相关内容摘自 after.xml。

```
<bean id="loggingAfterAspect" class="test.aspect.LoggingAfterAspect"/>

<aop:config>
	<aop:pointcut id="after"
		expression="execution(public void or String test.service..*(String)) and args(str)"/>
	<aop:aspect ref="loggingAfterAspect">
		<aop:after pointcut-ref="after" method="afterAspect"/>
	</aop:aspect>
</aop:config>
```

@Around 装备对应于<aop:around/>元素，它同<aop:before/>、<aop:after/>元素的定义完全一致。

比如，LoggingAroundAspect POJO 装备定义示例如下。

```
public class LoggingAroundAspect {

	private static final Log log = LogFactory.getLog(LoggingAroundAspect.class);

	public Object aroundAspect(ProceedingJoinPoint pjp, String str)
			throws Throwable {
		log.info(pjp);
		log.info(str);
		str = "worldheart changed!";
		return pjp.proceed(new Object[] { str });
	}

}
```

下面展示了 LoggingAroundAspect 及<aop:around/>元素的使用，相关内容摘自 around.xml。

```
<bean id="loggingAroundAspect" class="test.aspect.LoggingAroundAspect"/>

<aop:config>
	<aop:pointcut id="around"
		expression="execution(public void or String test.service..*(String)) and args(str)"/>
	<aop:aspect ref="loggingAroundAspect">
		<aop:around pointcut-ref="around" method="aroundAspect"/>
	</aop:aspect>
</aop:config>
```

Introduction（引入，inter-type）装备对应于<aop:declare-parents/>元素。下面展示了这一元素的使用，相关内容摘自 introduction.xml。

```
<bean id="loggingIntroductionAspect"
	class="test.aspect.LoggingIntroductionAspect"/>

<aop:config>
	<aop:pointcut id="introduction"
```

```
            expression="execution(public void or String test.service..*(String)) and args(str)"/>
        <aop:aspect ref="loggingIntroductionAspect">
            <aop:declare-parents types-matching="test.service.*"
                implement-interface="test.IIntroductionInfo"
                default-impl="test.IntroductionInfoImpl"/>
            <aop:around pointcut-ref="introduction" method="aroundAspect"/>
        </aop:aspect>
</aop:config>
```

本书为上述各个装备提供了相应的客户应用，考虑到篇幅所限，并没有将它们一一列举出来。开发者能够从 test 包很容易找到它们。

5.6.3 <aop:advisor/>元素

Advisor 对象是一种特殊的切面，它仅仅含有单个装备。开发者通过<aop:advisor/>元素能够隐式定义出 Advisor 对象。

下面给出了<aop:advisor/>元素的使用示例，摘自 advisor.xml 配置文件。

```
<bean id="simpleTraceInterceptor" class="org.springframework.aop.interceptor.SimpleTraceInterceptor"/>

<aop:config>
    <aop:pointcut id="all"
        expression="execution(public void or String test.service..*(..))"/>
    <aop:advisor advice-ref="simpleTraceInterceptor" pointcut-ref="all"/>
</aop:config>
```

5.6.4 切换代理机制

为了切换到 CGLIB 代理机制，开发者需要启用<aop:config/>元素的 proxy-target-class 属性，并将其取值置为 true。默认时，proxy-target-class 取值为 false。示例配置如下。

```
<aop:config proxy-target-class="true">
    ......
</aop:config>
```

此时，应用将拿到 CGLIB 代理，而不是动态代理。开发者可以将 CGLIB 代理造型成目标类和目标类实现的任意接口。

如何从各种 AOP 使用风格中作出选择

本章介绍了使用 Spring AOP 的三种不同方式。这些方式的具体机制都是基于代理实现的，但其使用过程还是存在许多差异性的，不同方式的功能也会有不同程度的区别。

如果开发者使用 JDK 5.0+开发 Spring Framework 使能应用，则建议使用@AspectJ 风格的 AOP。这是 Spring Framework 的发展趋势和方向。如果开发者不能够使用到 JDK 5.0+，则可以考虑基于<aop:config/>元素的 Spring AOP，或者使用老式的 Spring AOP。面对老式 Spring AOP 和基于

<aop:config/>元素的 Spring AOP 时，建议优先选用基于<aop:config/>元素的 Spring AOP。

当然，选择没有对错之分，开发者应该对自身的情况、周边环境有清醒的认识，最终选择出各方都满意的 Spring AOP 使用方式。

5.7 在 AspectJ 8 应用中启用@Configurable 注解

不难发现，spring-aspects-4.2.4.RELEASE.jar 包内置了不少 AspectJ 切面，比如 AnnotationBeanConfigurerAspect（同@Configurable 注解配合使用能实现领域对象的 DI 操作）、AnnotationTransactionAspect（同@Transactional 注解配合使用能智能完成事务操作）等，具体细节可参考这一 JAR 包内置的 META-INF/aop.xml 文件。本节将通过启用 AspectJ 装载期织入（LTW）阐述 AnnotationBeanConfigurerAspect 切面与@Configurable 注解的使用，其它切面将在本系列作品相关章节介绍。

借助 AnnotationBeanConfigurerAspect 切面，并配合@Configurable 注解的使用，我们能够实现领域对象的依赖注入操作。领域对象并不是由 Spring Framework 容器创建的，它们可能由“new”关键字构造而成，也可能由 O/R Mapping 工具创建，或通过其它途径获得。

本节内容将围绕 Eclipse configurabledemo 项目进行。

5.7.1 显式使用 AnnotationBeanConfigurerAspect 切面

AnnotationBeanConfigurerAspect 是一@AspectJ 切面，借助它能够进行领域对象的 DI 操作，本节将围绕它展开论述。

下面给出的 ConfiguredBySpringDI 类定义了 infoBean 协作者，这是封装了 userName 和 password 的 POJO 类。如果将 ConfiguredBySpringDI 类作为领域对象看待，则 infoBean 就是协作者。同时，@Configurable 注解被应用到这一类中。

```
@Configurable
public class ConfiguredBySpringDI {

    private InfoBean infoBean;

    public InfoBean getInfoBean() {
        return infoBean;
    }

    public void setInfoBean(InfoBean infoBean) {
```

```
            this.infoBean = infoBean;
        }

}
```

接下来，我们将 AnnotationTransactionAspect 切面配置在 config.xml 中，相关内容摘录如下。它将 ConfiguredBySpringDI 配置成受管 Bean，并将 InfoBean 配置成内部 Bean。注意到没有，我们将这一受管 Bean 的作用范围置为 prototype。与此同时，AnnotationBeanConfigurerAspect @AspectJ 切面也出现在这一配置文件中。

```
<bean class="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect" factory-method="aspectOf"/>

<bean scope="prototype" class="test.ConfiguredBySpringDI">
        <property name="infoBean">
            <bean class="test.InfoBean">
                <property name="userName" value="userName"/>
                <property name="password" value="password"/>
            </bean>
        </property>
</bean>
```

ConfigurableDemo 客户应用摘录如下，它使用了上述 ConfiguredBySpringDI 领域对象。

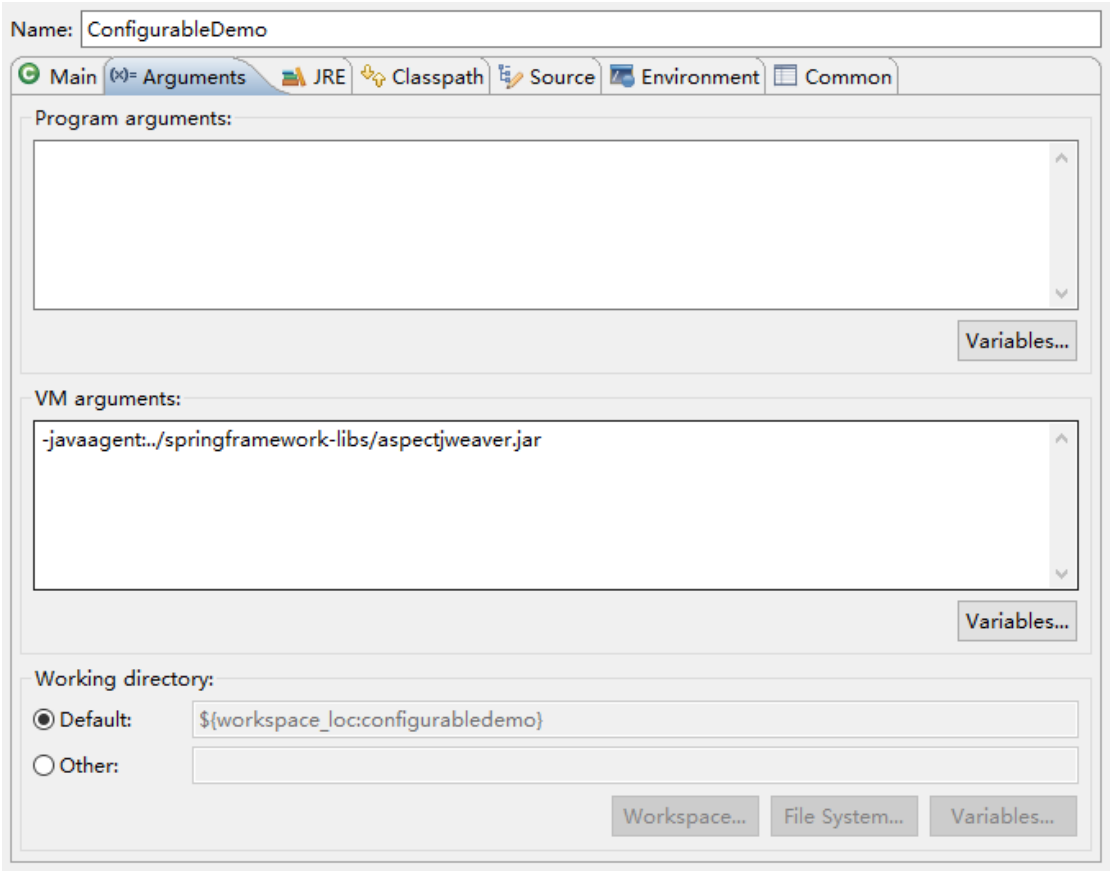
```
public static void main(String[] args) {

        //启用 Spring DI，并完成 AspectJ 8 切面的配置工作，比如将 IoC 容器暴露给切面
        new ClassPathXmlApplicationContext("config.xml");

        log.info("即将构建领域对象");

        //很多时候，Hibernate/JPA/应用代码会负责创建领域对象
        ConfiguredBySpringDI springDI = new ConfiguredBySpringDI();

        log.info("AspectJ 8 已经完成了领域对象的配置工作");

        //我们并没有显式设置 springDI 的协作者，但 infoBean 确实不再是 null
        log.info(springDI.getInfoBean().getUserName());
        log.info(springDI.getInfoBean().getPassword());

}
```

很显然，上述代码手工创建了 ConfiguredBySpringDI 对象实例，并借助 springDI 访问到 infoBean 协作者，此时应用代码并没有显式设置 infoBean 的取值，这一细节值得开发者深思。

运行上述应用前，开发者需要设置如下 JVM 参数，从而激活 AspectJ 8 的 LTW 行为。

```
-javaagent:../springframework-libs/aspectjweaver.jar
```

下图展示了 Eclipse 中是如何设置这一 JVM 参数的。



图表 32 激活 AspectJ 的 LTW 行为

应用被执行后，控制台将输出如下类似日志。我们可以推测出，AspectJ 从 config.xml 容器中找到 ConfiguredBySpringDI 对象，并用它动态替换掉代码中手工创建的 ConfiguredBySpringDI 对象，从而避免了 NPE 异常的抛出。

```
[INFO] 2014-09-20 20:42:45,559 AspectJ Weaver - [AspectJ] AspectJ Weaver Version 1.8.2 built on Thursday Aug 14, 2014 at 21:45:02 GMT
......
[INFO] 2014-09-20 20:42:45,684 AspectJ Weaver - [AspectJ] register aspect org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect
......
[DEBUG] 2014-09-20 20:42:45,996 AspectJ Weaver - [AspectJ] weaving 'test.ConfiguredBySpringDI'
......
[INFO] 2014-09-20 20:42:46,106 test.ConfigurableDemo - ¼´½«¹¹½¨ÁìÓò¶ÔÏó
......
[INFO] 2014-09-20 20:42:46,137 test.ConfigurableDemo - AspectJ 8ÒÑ¾-Íê³ÉÁËÁìÓò¶ÔÏóµÄÅäÖÃ¹¤×÷
[INFO] 2014-09-20 20:42:46,137 test.ConfigurableDemo - userName
[INFO] 2014-09-20 20:42:46,137 test.ConfigurableDemo - password
```

如果开发普通 Java 应用，则 springDI.getInfoBean()肯定会抛出空指针异常。从我们了解到的内容来看，@Configurable 注解肯定在这期间担任了重要的角色，

事实证明确实如此。

Spring Framework 提供的 AnnotationBeanConfigurerAspect 切面用来完成领域对象的依赖注入工作，这是 AspectJ 8 切面，而不是 Spring AOP 拦截器。它会对类一级应用了@Configurable 注解的类实施@AspectJ 切面织入，而且是 LTW 行为（即装载期织入），从而管理这些类的实例化、配置工作。开发者需要在 DI 容器中配置这一切面，并将领域对象也配置在 DI 容器中，比如 config.xml 中配置了 ConfiguredBySpringDI 对象。

一旦 ConfigurableDemo 示例客户应用中出现手工创建 ConfiguredBySpringDI 对象时，由于它在类一级应用了@Configurable 注解，因此已启用的 AnnotationBeanConfigurerAspect 切面便会介入进来，它会从当前 DI 容器中找到已配置的、ConfiguredBySpringDI 类型的受管 Bean。随后，相应的受管 Bean 会从 DI 容器返回，并替换手工创建的 ConfiguredBySpringDI 对象。最终，springDI.getInfoBean()成功返回了 InfoBean 对象，这正是配置在 DI 容器中的那个内部 Bean。

通常情况下，由于同一应用存在大量领域对象，因此 ConfiguredBySpringDI 受管 Bean 的作用范围是 prototype。细心的开发者或许还有这样一个疑问，AnnotationBeanConfigurerAspect 切面是如何从 Spring DI 容器查找到所需的 ConfiguredBySpringDI 受管 Bean 呢？这正是我们接下来要重点阐述的知识点。

5.7.2 阐述@Configurable 注解

Spring Framwork 引入的@Configurable 注解只能够作用于类，通常是领域对象本身，而不能将它运用到方法或其它地方。被应用了这一注解的类将参与到 AspectJ LTW 行为中，并完成协作者的智能注入工作。下面展示了@Configurable 注解暴露的所有成员，通过它们能够自定义若干行为，比如领域对象对应受管 Bean 的查找策略、协作者的注入策略等。

```
/**
 * 标识类，由 Spring DI 驱动
 *
 * 通常，它同 AnnotationBeanConfigurerAspect 一起使用
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Inherited
```

```
public @interface Configurable {

    //受管 Bean 的名字，对应到 DI 容器
    String value() default "";

    //是否启用 autowiring 来注入协作者
    Autowire autowire() default Autowire.NO;

    //是否启用依赖性检查
    boolean dependencyCheck() default false;

    //注入协作者出现在构建领域对象前
    boolean preConstruction() default false;

}
```

其中，value 属性用来指定领域对象对应受管 Bean 的名字，下面给出了配置示例。默认时，AnnotationBeanConfigurerAspect AspectJ 切面会根据领域对象的全限定名（FQN）查找到对应的受管 Bean 的名字，比如下面示例配置示例中给出的“test.ConfiguredBySpringDI”取值。如果受管 Bean 的定义中未出现 id 或 name 属性，则类的 FQN 便被当作领域对象对应受管 Bean 的名字。

```
@Configurable(value=" test.ConfiguredBySpringDI")
```

或者，开发者可以提供如下简写形式。

```
@Configurable("test.ConfiguredBySpringDI")
```

另外，autowire 属性用来指定领域对象注入协作者的 Autowiring 策略。在展示这一属性的使用前，得先调整 config.xml 配置文件，将 ConfiguredBySpringDI 对 InfoBean 的静态引用去掉，configautowiring.xml 持有调整后的配置，相关内容摘录如下。

```
<bean scope="prototype" class="test.ConfiguredBySpringDI"/>

<bean class="test.InfoBean" scope="prototype"
    p:userName="userName" p:password="password"/>
```

如果不启用 Autowiring 策略，则 ConfiguredBySpringDI 的 infoBean 属性将永远为空，下面给出了配置示例。

```
@Configurable(autowire=Autowire.BY_TYPE)
```

或者，可在定义 ConfiguredBySpringDI 时启用<bean/>元素暴露的 autowire 属性，以达到同样的效果，配置示例如下。

```
<bean scope="prototype" class="test.ConfiguredBySpringDI" autowire="byType"/>
```

@Configurable 注解的 dependencyCheck 属性能够检查受管 Bean 的属性是否符合条件，它主要配合 autowiring 属性使用，下面给出了配置示例。由于上述 InfoBean 受管 Bean 的默认惟一标识符为 test.InfoBean，加上 dependencyCheck 属

性取值为 true，因此当运行客户应用时，DI 容器会抛出异常。

```
@Configurable(autowire=Autowire.BY_NAME, dependencyCheck=true)
```

最后，看看 preConstruction 属性。它能够控制调用领域对象的构建器前是否注入协作者，比如下面展示了这一属性的使用。preConstruction 属性取值为 false，即 ConfiguredBySpringDI ()构建器会被先触发，随后才是 setInfoBean ()方法，因此这段代码的执行不会成功。

```
@Configurable(autowire=Autowire.BY_TYPE, preConstruction=true)
public class ConfiguredBySpringDI {

    private InfoBean infoBean;

    public ConfiguredBySpringDI() {
        Assert.notNull(infoBean);
    }

    ......
}
```

通常，对构建器调用会在协作者被注入前被触发，比如 ConfiguredBySpringDI 所依赖的 InfoBean 协作者。不幸的是，在某些场合，我们需要在构建器中使用到协作者，比如上述 ConfiguredBySpringDI 构建器便使用到 infoBean 协作者。此时，preConstruction 属性便可派上用场，它能够确保构建器调用发生在协作者注入之后被触发。

5.7.3 通过 META-INF/aop.xml（或 aop-ajc.xml）控制启用的切面集合

当前，存储 AnnotationBeanConfigurerAspect 切面的 JAR 包中还持有其它 AspectJ 切面。为控制启用的切面集合，开发者需要使用 META-INF/aop.xml 或 aop-ajc.xml 配置文件。下面给出了一配置示例。

```
<!DOCTYPE aspectj PUBLIC
  "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>
    <weaver options="-showWeaveInfo -XmessageHandlerClass:org.springframework
                                                    .aop.aspectj.AspectJWeaverMessageHandler">
            <include within="test..*"/>
    </weaver>
    <aspects>
            <include within="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect"/>
    </aspects>
</aspectj>
```

AspectJWeaverMessageHandler 监听了 AspectJ 产生的各种织入事件，并输出到控制台上。<aspects/>元素中的<include/>子元素指定了待启用的切面。通过

aspectj.dtd，开发者能够获得这一 DTD 的完整介绍。

5.7.4 <aop:include/>元素

通过上述内容，我们完成了领域对象的织入操作，这是 Spring AOP 同 AspectJ 8 集成成果的展示。但到目前为止，我们也只是单独使用 AspectJ 8 切面和 Spring AOP，同时使用 AspectJ 8 切面和 Spring AOP 技术将是更复杂的问题。比如，在 Spring IoC 容器中，部分@AspectJ 切面由 AspectJ 8 完成织入工作，而部分@AspecJ 切面参与到 Spring AOP 中。

设计、架构 Spring 使能应用过程中，我们不可避免地要同时使用到 AspectJ 8 切面和 Spring AOP 技术，比如在对领域对象实施 DI 时，可能会涉及到 RDBMS 操作。为了完成 RDBMS 操作，就需要采用 Spring AOP 完成事务管理。为了控制 AspectJ 8 中@AspectJ 风格的切面的织入工作，开发者通过维护 aop.xml 文件能够达到这一目的；为了控制 Spring AOP 中@AspectJ 风格的切面的织入工作，开发者能够通过维护<aop:aspectj-autoproxy/>内置的<aop:include/>子元素达到这一目的。

下面展示了<aop:include/>的使用。此时，name 属性用于指定待匹配的那些@AspectJ“受管切面”。

```
<aop:aspectj-autoproxy>
        <aop:include name="loggingBeforeAspect"/>
</aop:aspectj-autoproxy>

<bean id="helloworldDao" class="test.dao.HelloWorldDaoImpl">
        <property name="helloworld" value="hello world"/>
</bean>

<bean id="helloworldService" class="test.service.HelloWorldServiceImpl">
        <property name="helloWorldDao" ref="helloworldDao"/>
</bean>

<bean id="loggingBeforeAspect" class="test.aspect.LoggingBeforeAspect"/>
```

默认时，Spring DI 中配置的所有@AspectJ 风格的切面都将参与到 autoproxy 自动代理处理过程中。如果开发者显式地通过<aop:include/>元素指定了@AspectJ 风格的切面集合，则只有这些切面才会参与到 autoproxy 自动代理处理过程中。

最终，同一应用中，通过有效控制 aop.xml 和<aop:include/>元素，使得不同@AspectJ 切面的用途、宿主环境不一样，比如某些切面宿主在 DI 容器中，而有

些宿主在 AspectJ 运行时中。

5.7.5 <context:spring-configured/>元素

<context:spring-configured/>能起到隐式使用 AnnotationBeanConfigurerAspect 切面的目的，下面给出了配置示例（springconfigured.xml）。

```
<context:spring-configured/>

<bean scope="prototype" autowire="byType" class="test.ConfiguredBySpringDI"/>

<bean class="test.InfoBean" scope="prototype">
        <property name="userName" value="userName"/>
        <property name="password" value="password"/>
</bean>
```

同样地，运行 ConfigurableDemo 时需要指定上述一致的-javaagent JVM 参数。

5.7.6 初探<context:load-time-weaver/>元素

除借助<context:spring-configured/>隐式使用 AnnotationBeanConfigurerAspect 切面外，采用<context:load-time-weaver/>元素也能做到这一点。比如，ltw.xml 中配置了如下内容。

```
<context:load-time-weaver/>

<bean scope="prototype" autowire="byType"
        class="test.ConfiguredBySpringDI"/>

<bean class="test.InfoBean" scope="prototype"
        p:userName="userName" p:password="password"/>
```

此时，开发者需要指定如下 JVM 参数，否则 AspectJ LTW 行为无法激活。

```
-javaagent:../springframework-libs/spring-instrument-4.2.4.RELEASE.jar
```

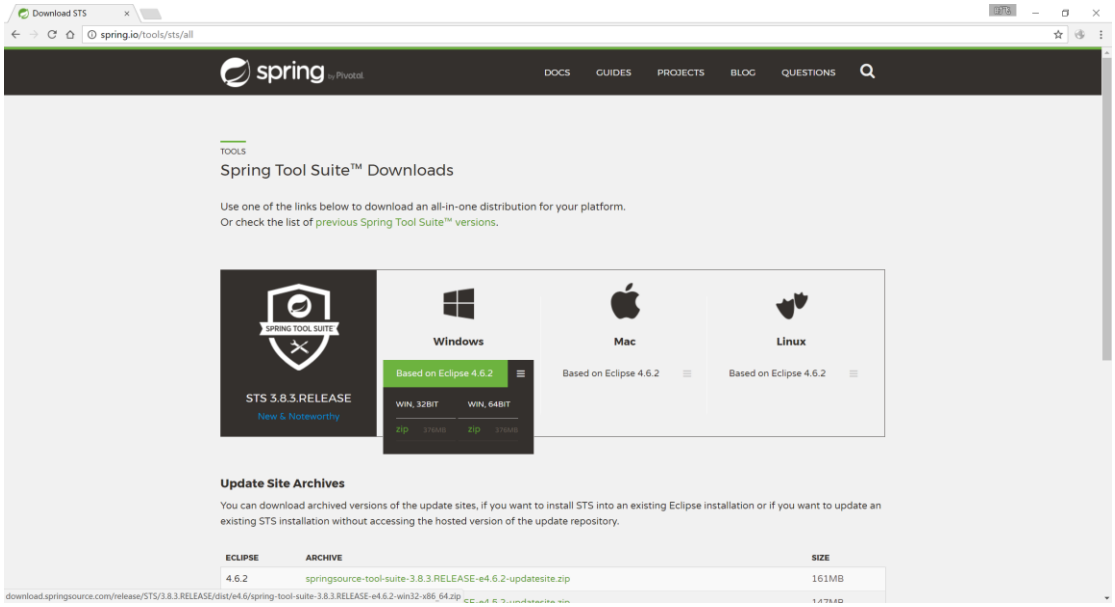
事实上，<context:load-time-weaver/>只是兼容<context:spring-configured/>，它的功能非常强大，我们将在相关地方深入阐述<context:load-time-weaver/>的机理及使用。

6 Spring Tool Suite 介绍

Spring Tool Suite（简称 STS）是 Pivotal Software 公司针对 Spring 开发者研发的一款重量级、开源的开发工具，它针对 Spring 生态系统提供了一流的研发支持。本章内容将围绕 STS 的安装及使用展开论述。

6.1 获得 Spring Tool Suite

开发者通过 http://spring.io/tools/sts/all 网址能够找到 STS 的下载入口。下图给出了对应的操作界面。



图表 33 下载 STS 的操作界面

无论是 Windows、Linux，还是 Apple Mac OS X 操作系统用户，他们都能够使用到 STS。这里以下载 spring-tool-suite-3.8.3.RELEASE-e4.6.2-win32-x86_64.zip 为例。

6.2 安装 Spring Tool Suite

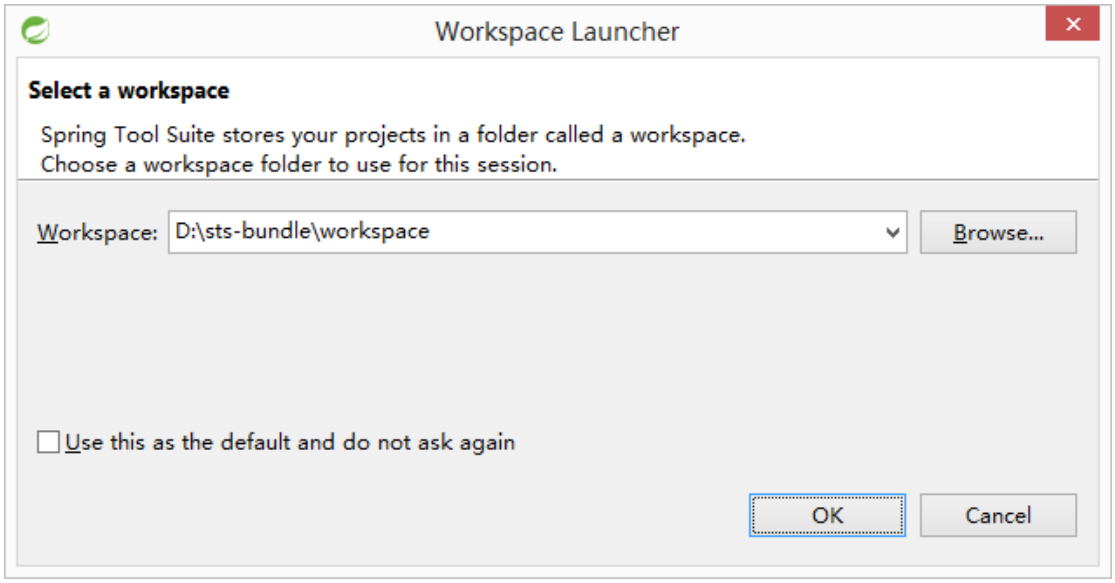
安装 STS 的步骤非常简单，但在此之前开发者要准备好 JDK，比如在“C:\jdk1.8.0_121”目录位置安装好 Java SE 8。并设置好若干环境变量，比如将 JAVA_HOME 环境变量设置成“C:\jdk1.8.0_121”、classpath 设置成“.;”、将

“%JAVA_HOME%\bin”追加到 Path 环境变量中。

当通过 DOS 窗口成功运行“java -version”后，开发者便可开始进行 STS 的安装工作了，下面给出了相关环境变量的确认操作。

```
D:\>set JAVA_HOME
JAVA_HOME=C:\jdk1.8.0_121

D:\>java -version
java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)

D:\>
```

直接将 spring-tool-suite-3.8.3.RELEASE-e4.6.2-win32-x86_64.zip 解压到 D:\ 位置即可，解压完成后整个 STS 会安装在 D:\sts-bundle 位置。安装完后，开发者即可启动 STS，在选定 Eclipse 工作空间（见下图）后，便可开始享受 STS 带来的快乐了。本书假定使用 D:\ sts-bundle\workspace 目录作为工作空间。



图表 34 选择工作空间

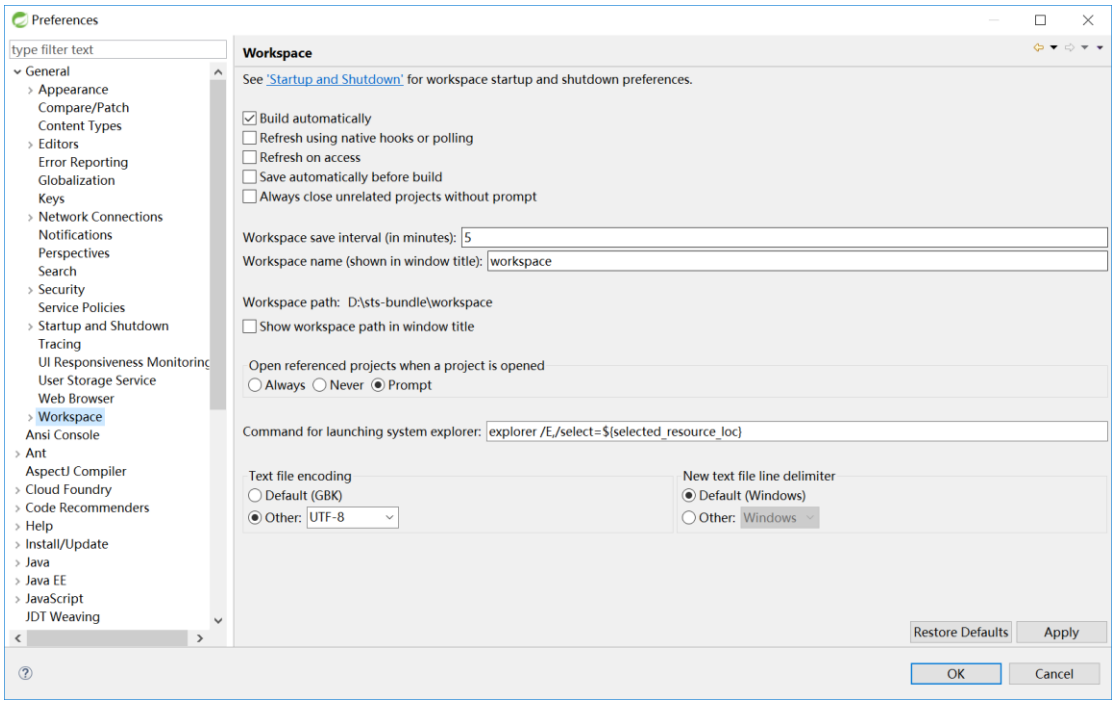
6.2.1 将本书配套代码导入到 STS 中

从 https://github.com/openvcube/springframework50 位置能够下载到本书配套代码，比如单击这一页面中的“Download ZIP”按钮。将下载到的 ZIP 文件解压到 D:\sts-bundle\workspace 目录中，并将 springframework50-master 子目录重命名成 springframework50。

随后，借助 STS 内置的“Import…”菜单能够完成所有工程的导入操作。

另外，这些工程的代码默认都以 UTF-8 形式存储，因此开发者还需要手工将

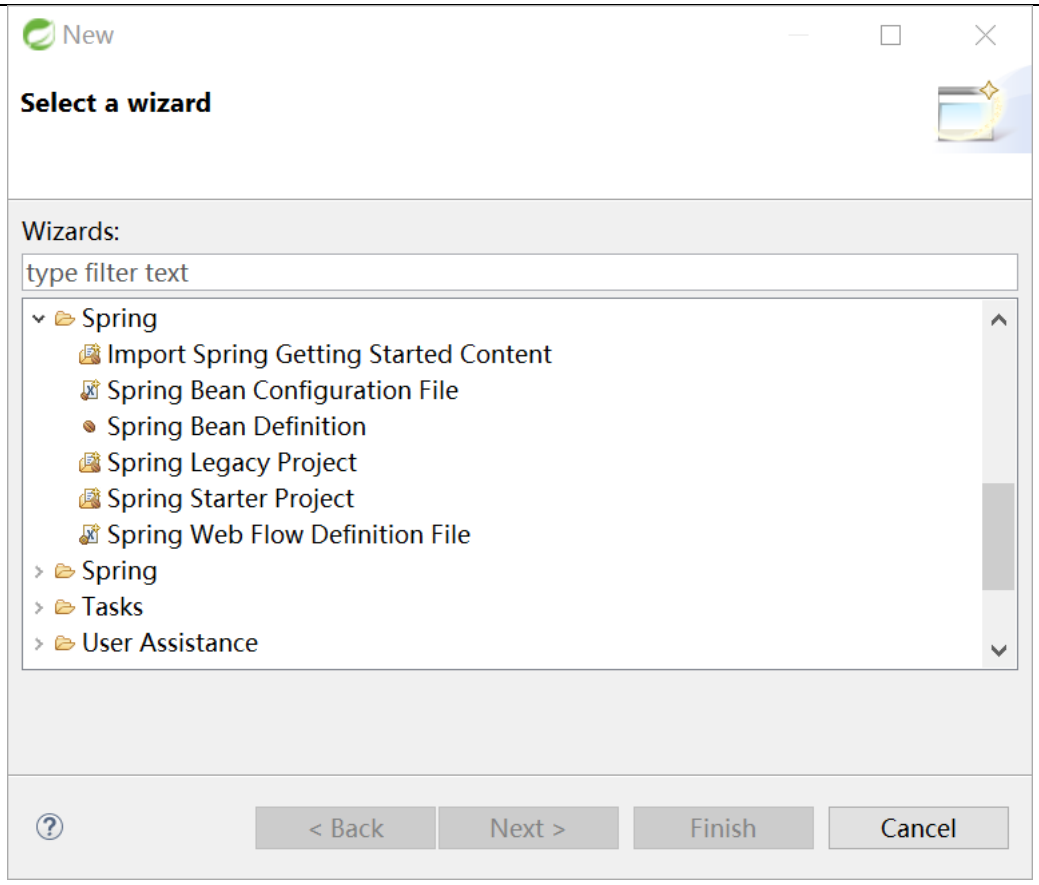
STS 工作空间的“Text file encoding”取值调整成 UTF-8，具体见下图。



图表 35 将“Text file encoding”取值置成 UTF-8

6.2.2 STS 提供的向导

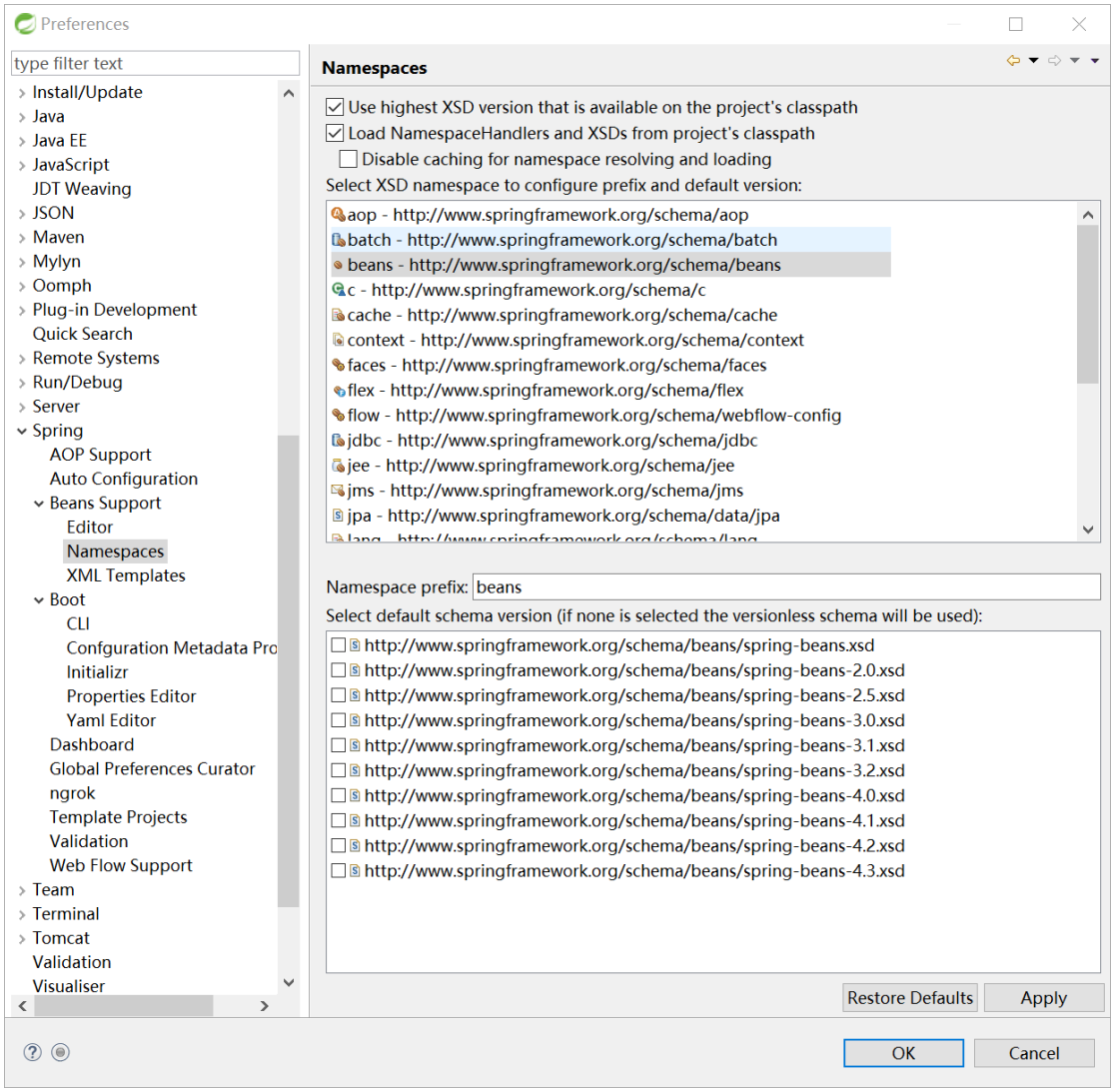
STS 提供了下图展示的向导，供开发者快速完成相关工作。比如，新建 Spring 工程、创建 Spring DI 配置文件、创建 Spring Web Flow 定义文件等。



图表 36 STS 提供的向导

6.2.3 Spring 选项

透过 STS 的 Preferences 对话框，能够看到如下 Spring 选项。



图表 37 Spring 选项

这一 Spring 选项针对 Spring 生态系统中的不同项目提供了大量配置项，比如命名空间的版本、Spring Web Flow 支持等。

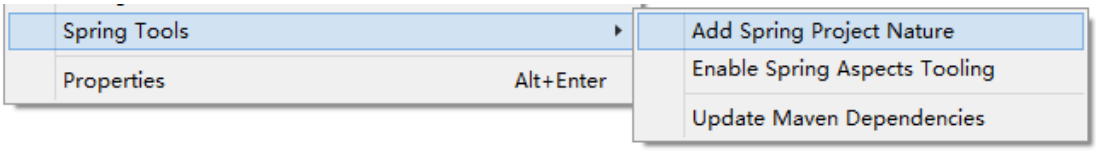
6.3 STS（Spring IDE）针对 Spring Framework 提供的相关支持

STS 内置的 Spring IDE 针对 Spring Framework 提供了强大的支持能力，相关介绍如下。

6.3.1 激活和删除项目的“Spring Project Nature”

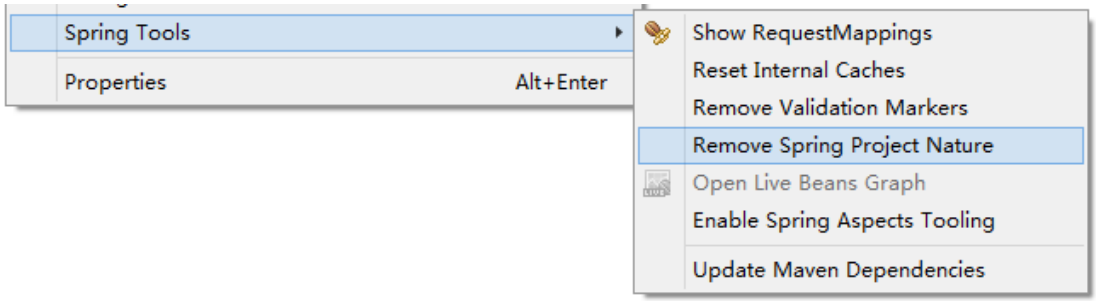
目标 Java SE/Java EE 项目为使用到 STS 内置的 Spring IDE 支持，则必须启

用“Spring Project Nature”。下图给出了操作界面，右键单击项目能够看到它。



图表 38 启用 Spring Project Nature 支持

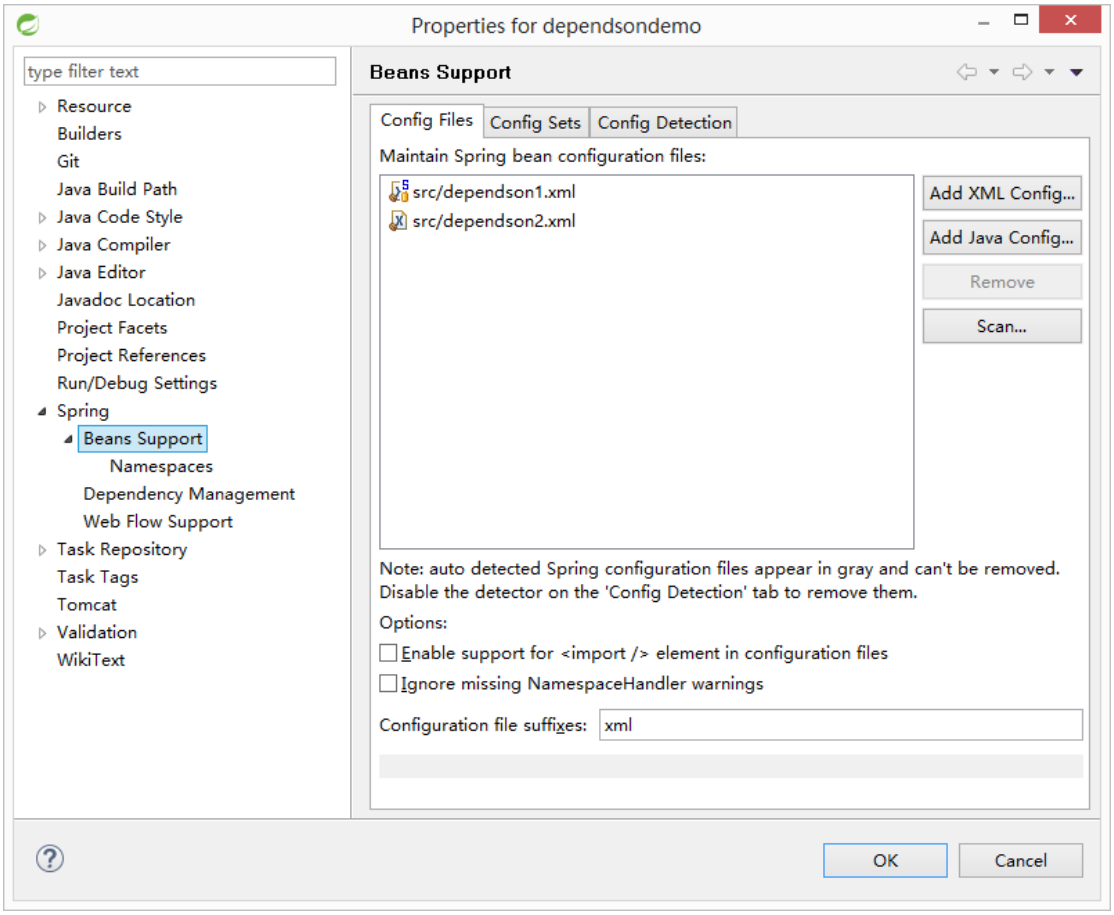
类似地，如果要删除项目的“Spring Project Nature”，则要根据如下操作界面操作。



图表 39 删除 Spring Project Nature 支持

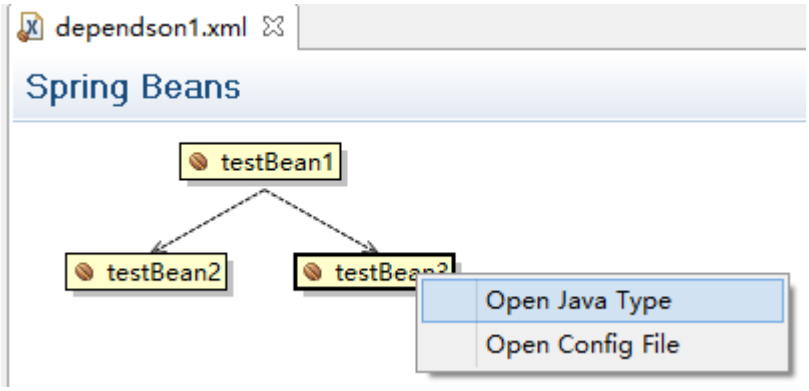
6.3.2 使用 Spring IDE

随后，开发者需要打开“Spring Explorer”视图，并激活下图界面。通过它，能够完成多项设置，比如 Spring DI 配置文件的设定、配置集合、命名空间的启用等。



图表 40 配置项目中同 Spring Framework 相关的内容

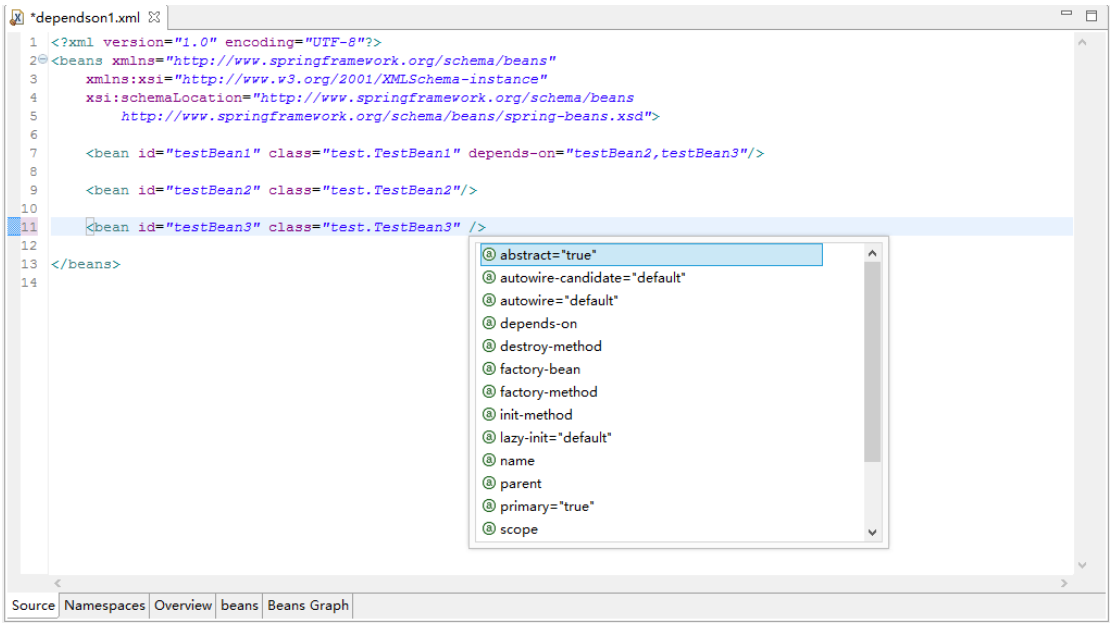
当我们将若干 Spring DI 配置文件放置到“Config Files”Tab 页后，便能够通过“Spring Explorer”视图浏览到它们，并以可视化方式展现出 DI 容器的内部结构，比如各协作者间的层次关系。下图展示了相关界面。



图表 41 可视化 DI 容器的内部结构

上图展示的内容非常重要。实践证明，开发者之间经常需要探讨应用的设计、架构师需要评审开发者的工作，而可视化 DI 容器无疑为这些工作提供了最好的帮助。而且，开发者可以同这一可视化 DI 容器进行交互。

除了针对 DI 容器进行可视化操作外，开发者还可以借助 Spring IDE 内置的 XML 辅助编写能力，以加快 Spring DI 配置文件的编写速度及质量的保证，因为一旦有问题，Spring IDE 会及时指出。为启用 XML 辅助编写，我们需要用“Spring Config Editor”打开 Spring DI 配置文件。下图展示了这一特性的使用，通常用 Alt+“/”键组合能够激活代码辅助特性。

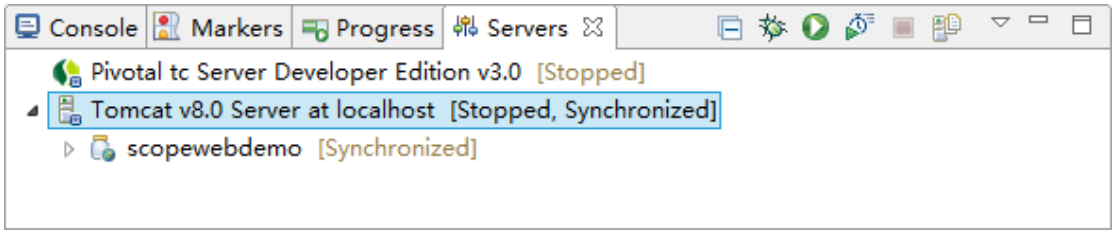


图表 42 Spring IDE 内置的 XML 辅助编写能力

更多细节，还需要开发者在日常开发工作期间体会、总结。

6.4 Apache Tomcat 8.0 集成支持

在 STS 中部署及调试 Web 应用方法很多，比如借助 STS 内置的 Servers 视图添加 Java EE 容器或 Web 容器，如下图所示。

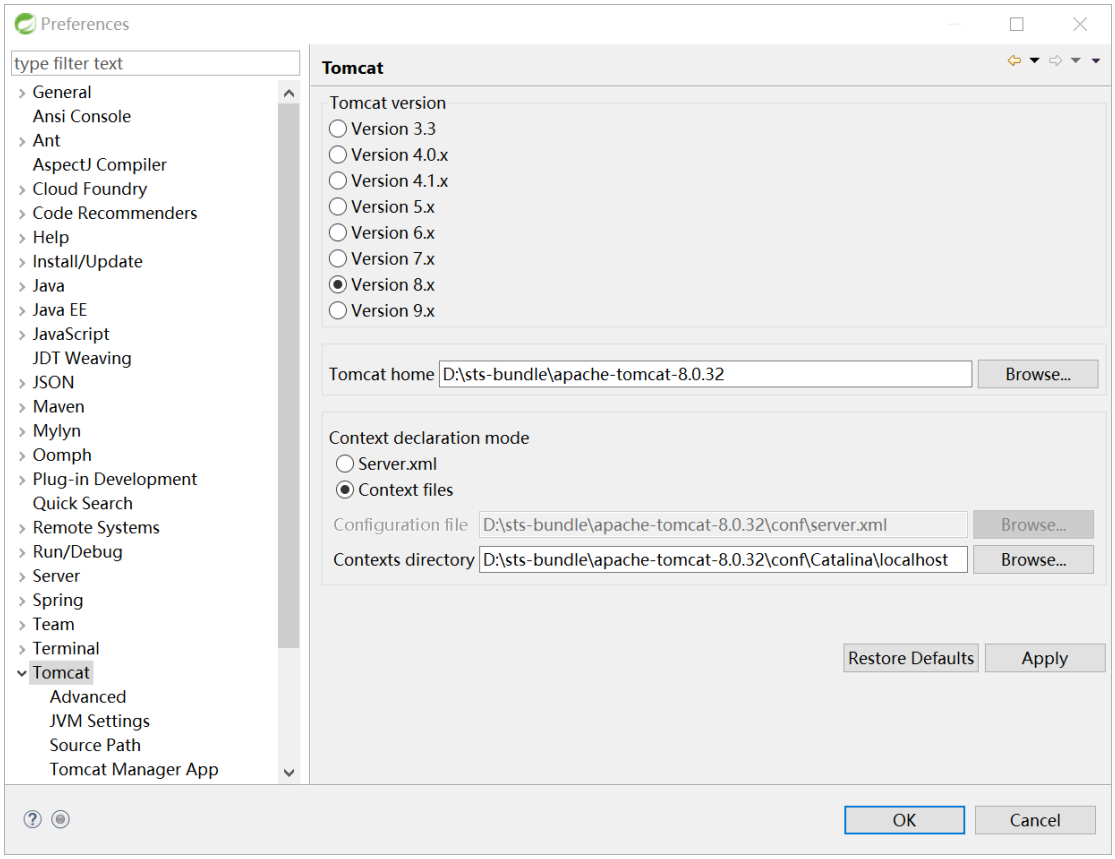


图表 43 STS 内置的 Servers 视图

Sysdeo Eclipse Tomcat Launcher 插件为我们提供了另外一种选择，它也是我们严重推荐的插件。使用 Server 视图方式时，应用本身需要物理拷贝到某个地方，而 Sysdeo Eclipse Tomcat Launcher 插件不需要这么做，这在开发及调试效率

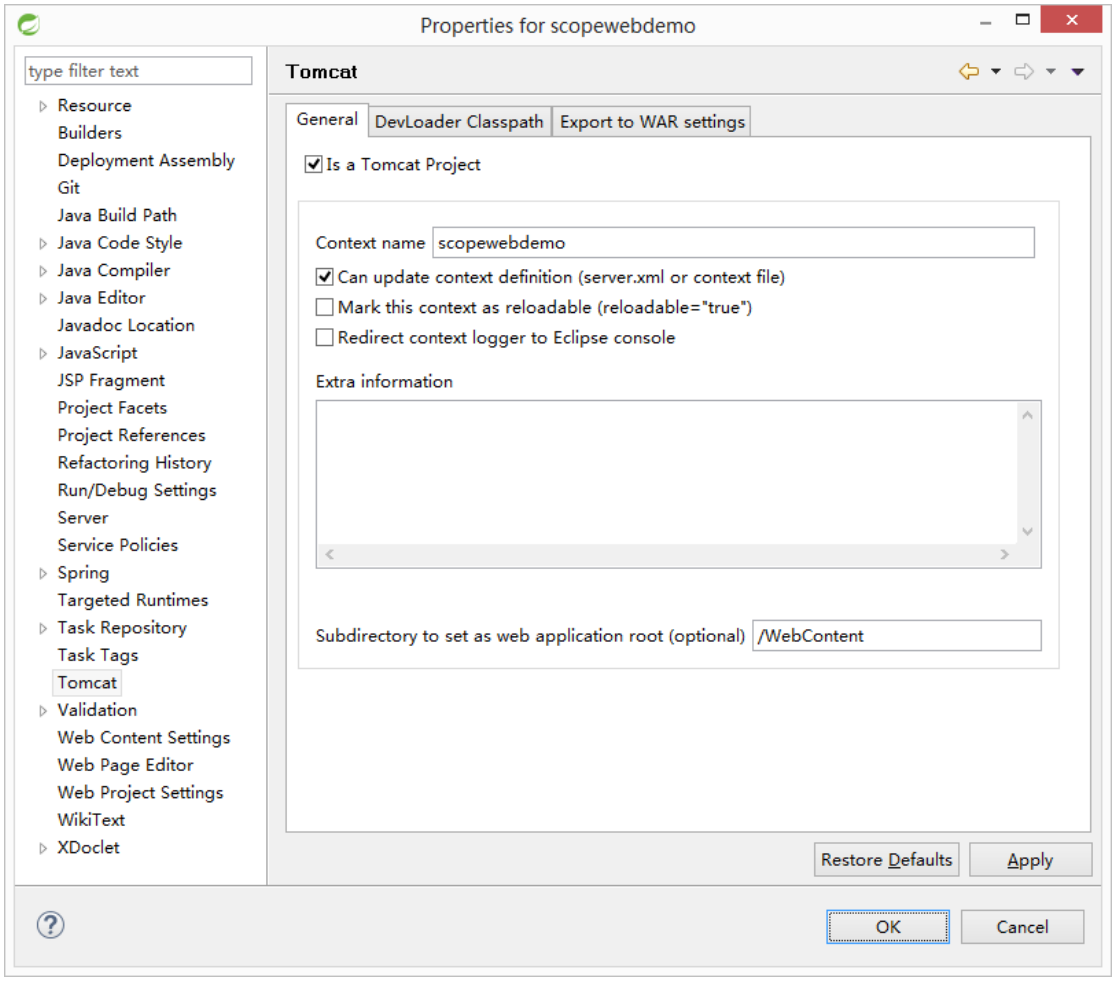
上得到保证。

从 https://sourceforge.net/projects/tomcatplugin/files/能够下载到这一插件，将下载的插件解压到 STS 中，比如 D:\sts-bundle\sts-3.8.3.RELEASE\plugins 位置。重启 STS 后，透过 STS 的 Preferences 对话框，能够看到如下 Tomcat 选项。



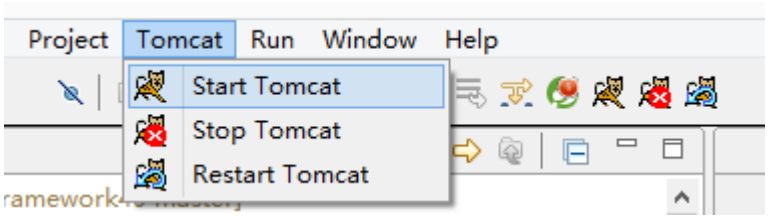
图表 44 全局 Tomcat 选项

配置好相应的 Apache Tomcat 位置及相关参数后，开发者便可以将具体的 Web 应用部署到 Tomcat 中。比如，下面给出了将 scopewebdemo 项目部署到其中的示例配置。



图表 45 项目级 Tomcat 选项

在这一示例配置中，scopewebdemo 项目的上下文配置成 scopewebdemo，而 Web 应用的位置在/WebContent 目录中。最后，通过下图给出的操作内容可以完成 Tomcat 的启动及停止等操作。



图表 46 操控 Apache Tomcat

关于 Spring Tool Suite 的未尽事宜

Spring Tool Suite 对 Spring 生态系统提供了一流的端到端支持，本书仅仅介绍了同 Spring Framework 相关的 STS 特性。在介绍 Spring 生态系统时，相关内部资料会有针对性的介绍，比如

Spring Web Flow、Spring Boot。

参考资料

网站

- Spring Framework Reference Documentation
  - http://docs.spring.io/spring/docs/4.2.4.RELEASE/spring-framework-reference/htmlsingle/
- Spring 官方网站：http://spring.io/
- Eclipse Tomcat 插件：
  - https://sourceforge.net/projects/tomcatplugin/files/

图书

- 《精通 Spring-深入 Java EE 开发核心技术》（第三版），作者罗时飞。电子工业出版社，2008.10
- 《Expert One-on-One J2EE Development without EJB》，作者 Rod Johnson、Juergen Hoeller。WILEY，2004.6
- 《Professional Java Development with the Spring Framework》，作者，Rod Johnson、Juergen Hoeller 等。WILEY，2005.7
- 《AspectJ in Action: Enterprise AOP with Spring Applications》（第二版），作者，Ramnivas Laddad。Manning，2009.9

期待您的打赏

原创不易。如果这一内部资料对您有所帮助，欢迎通过如下支付宝和微信打赏，您的支持是我们前行的动力。

如有任何问题咨询，非常欢迎加入我们的 QQ 群（106813165），加入密码是“www.open-v.com”。

或者，欢迎关注我们的微信订阅号（openvcube）。

任何建议、合作，我们欢迎您的来信，openvcube@gmail.com。