

2008

Spring2.5

开发简明教程

中文版

E-mail: zhangyong232@163.com

张勇
四川农业大学信息与工程技术学院
2008-7-21



前言

.....正在等待完成...

...由于最近要找工作、而且学习任务也比较紧.....

该书的更新可能将会非常慢，请见谅。

再次感谢您的支持。

等过了这段时间，我会继续更新。

如果您有任何建议或者板砖，我都非常欢迎，请发送 **mail** 到
dashoumail@163.com

版权声明

本书所有版权归作者所有，仅供个人学习、研究使用，不得用于商业目的。在保证免费、且无任何附加条件的情况下，可以在网络自由传播，但必须保证本书的完整性。未经作者书面许可，不得对本书以任何形式进行出版、修改和编辑。作者保留所有权利。

任何商业机构都不得使用本书作为培训教材，作者保留诉诸法律的权利。

如果发现有任何错误、疏漏之处或者任何建议，敬请 email 到 dashoumail@163.com。

个人博客地址：<http://www.blogjava.net/cmzy/>

版本历史

版本	时间	作者	备注
1.0	7/14/2008	张勇	完成第三章
1.0.1	8/24/2008	张勇	完成第四章
1.0.2	9/18/2008	张勇	完成数据持久化的 JDBC 部分

目录

前言.....	2
版权声明.....	3
版本历史.....	4
第一章 什么是 SPRING.....	8
1.1 SPRING 简介.....	9
1.2 SPRING 七大模块.....	10
1.2.1 Spring Core.....	10
1.2.2 Spring AOP.....	10
1.2.3 Spring DAO.....	10
1.2.4 Spring Context.....	10
1.2.5 Spring ORM.....	11
1.2.6 Spring Web.....	11
1.2.7 Spring MVC.....	11
1.3 小结.....	11
参考文献:	11
第二章 SPRING 开发环境搭建.....	12
2.1 系统需求.....	12
2.2 JDK/JRE 下载与安装 (可选)	13
2.2.1 下载.....	13
2.2.2 安装.....	13
2.2.3 配置环境变量 (可选)	14
2.3 ECLIPSE 下载与安装.....	15
2.3.1 下载.....	16
2.3.2 安装.....	17
2.3.3 运行与配置.....	17
2.3.4 开发一个 HelloWorld 程序.....	20
2.4 ECLIPSE 插件安装.....	22
2.4.1 直接复制文件 Eclipse 安装插件.....	22
2.4.2 使用 Link 文件安装 Eclipse 插件.....	22
2.5 MYECLIPSE 下载与安装.....	22
2.5.1 下载.....	23
2.5.2 安装.....	23
2.6 数据库下载与安装.....	24
2.6.1 安装 MySQL.....	25
2.6.2 安装 SQL-Server 2000.....	26
2.7 TOMCAT 的下载与安装.....	30
2.7.1 Tomcat 下载与安装.....	30
2.7.2 在 MyEclipse 中安装 Tomcat.....	32

2.8	打造 JAVA 开发绿色环境.....	33
2.9	SPRING 之 HELLOWORD.....	33
2.10	小结.....	38
第三章 IOC IN SPRING.....		39
3.1	什么是 IoC.....	40
3.2	如何配置受管 BEAN.....	42
3.2.1	使用<bean/>标签.....	43
3.2.2	为受管 Bean 注入值.....	44
1、	设值注入.....	44
2、	构造子注入.....	45
3、	Autowire 自动装配.....	47
3.2.3	使用赋值标签.....	51
1、	<ref/>标签.....	51
2、	<list/>标签.....	51
3、	<set/>标签.....	51
4、	<map/>标签.....	51
5、	<props/>标签.....	52
6、	<null/>标签.....	52
3.3	受管 BEAN 作用范围.....	56
3.4	受管 BEAN 的生命周期.....	60
3.4.1	受管 Bean 在容器中的生命周期.....	60
3.4.2	受管 Bean 的预处理和后处理.....	62
1、	使用 BeanPostProcessor 接口.....	62
2、	使用初始化回调接口 InitializingBean.....	64
3、	使用析构回调接口 DisposableBean.....	65
4、	使用<bean>标签的 init-method 和 destroy-method 属性.....	67
3.5	SPRING 中的 IOC 容器.....	68
3.5.1	BeanFactory 容器.....	68
3.5.2	ApplicationContext 容器.....	69
1、	ApplicationContext 容器的实现类.....	69
2、	Spring 事件.....	70
3.5.3	使用 BeanFactoryPostProcessor 工厂后置处理器.....	76
3.5.4	定义配置元数据.....	78
3.6	受管 BEAN 了解自己.....	80
3.6.1	获取 Bean 自身在容器中的 Id.....	80
3.6.2	获取 IoC 容器的引用.....	81
3.7	基于注解（ANNOTATION-BASED）的配置（SPRING2.5 新增）.....	82
3.7.1	@Autowired 注解.....	82
3.7.2	@Qualifier 注解.....	87
3.7.3	@Resource 注解.....	88
3.7.4	@PostConstruct 和 @PreDestroy 注解.....	89
3.7.5	@Component 注解.....	90
3.7.6	@scope 注解.....	92
3.8	小结.....	92

第四章 IOC IN SPRING.....	93
4.1 什么是 AOP?	94
4.2 SPRING AOP 概述.....	95
4.3 使用 PROXYFACTORYBEAN.....	99
4.3.1 从一个范例开始.....	99
4.3.2 使用 CGLIB 代理.....	101
4.4 SPRING 的 POINTCUT.....	102
4.4.1 静态 Pointcut.....	103
4.4.2 动态 Pointcut.....	108
4.4.3 自定义切入点.....	109
4.4.4 使用 Advisor.....	110
4.5 SPRING AOP 的支持通知类型.....	113
4.5.1 前置通知 (Before advice)	113
4.5.2 返回后通知 (After returning advice)	113
4.5.3 环绕通知 (Around Advice)	116
4.5.4 异常通知 (After throwing advice)	117
4.5.5 引入通知 (Introduction advice)	119
4.6 使用自动代理.....	123
4.6.1 使用 BeanNameAutoProxyCreator.....	123
4.6.2 使用 DefaultAdvisorAutoProxyCreator.....	126
4.7 @ASPECTJ 风格的 AOP.....	127
4.7.1 一些必要的准备工作.....	127
4.7.2 声明切面.....	128
4.7.3 使用切入点.....	130
4.7.4 使用通知.....	135
1、 前置通知 (@Before)	135
2、 后置通知 (@After)	140
3、 返回后通知 (@AfterReturning)	140
4、 异常通知 (@AfterThrowing)	141
5、 环绕通知 (@Around)	145
4.8 基于 AOP 命名空间的 AOP.....	146
4.8.1 一点准备工作和一个例子.....	146
4.8.2 声明一个切面.....	149
4.8.3 声明一个切入点.....	149
4.8.4 声明一个通知.....	150
1、 前置通知.....	150
2、 后置通知.....	150
3、 返回后通知.....	151
4、 异常通知.....	151
5、 环绕通知.....	151
6、 一个例子.....	151
4.9 小结.....	155
4.10 参考文献和推荐资料:	155
第五章 数据持久化.....	156

5.1	概述.....	157
5.2	SPRING 对 DAO 的支持	159
5.3	SPRING 对 JDBC 的封装	160
5.3.1	从一个例子开始.....	160
5.3.2	ConnectionCallback 回调接口和 StatementCallback 回调接口.....	163
5.3.3	使用静态 SQL 查询数据库.....	164
5.3.4	ResultSetExtractor 和 RowMapper 接口的实现类.....	167
5.3.5	使用预编译语句.....	168
5.3.6	插入和更新数据库.....	171
5.3.7	一个完整的 JdbcDaoSupport 的例子.....	173
5.3.8	使用 NamedParameterJdbcDaoSupport	179
5.3.9	使用 SimpleJdbcDAOSupport	180
5.4	集成 HIBERNATE	184
5.4.1	Hibernate 简介.....	184
5.4.2	一个简单的 Hibernate 例子.....	184
5.4.3	HibernateTemplate 和 HibernateDaoSupport	184
5.4.4	使用 MyEclipse 自动生成实体类和 DAO 类.....	184
5.5	小结.....	184
5.6	参考文档和推荐资料.....	184

第一章 什么是 **Spring**

1.1 Spring 简介

What is Spring? Spring 是一个开源框架，2003 年由 Rod Johnson 创建并启动，它最初形成于 Rod Johnson 在 2002 年出版的一本很有影响力的书《Expert One-on-One J2EE Design and Development》中的源代码^[1]。传统的 java 企业级应用开发很复杂，由此带来了开发难度加大、代码维护成本高、代码质量不稳定、开发进度难以控制、测试复杂等一大堆的问题。Spring 正是为了降低这种复杂度而设计。它使用依赖注入（IoC）和面向切面编程（AOP）这两种先进的技术为基础，大大的降低企业开发的难度，而且工作量也不会因此而变大。

Spring 做了很多来降低 JEE 开发的复杂度，当你使用它的时候却变的十分简单。它是一个轻量级的非侵入式框架。

轻量级：Spring 在体积和开销上都是轻量级的。首先，Spring 的完整发布包最小可以集成在一个 2.5MB 多一点的 jar 文件中。其次，Spring 使用的微内核体系，使它的处理开销也非常的小。最后，Spring 是非侵入式的，典型的，Spring 应用中的对象不依赖于 Spring 特定的类。

1.2 Spring 七大模块

Spring 使用了分层架构模式，主要由七大功能模块组成。在实际应用中，你可以需选择使用它为你提供一个或者多个功能模块，灵活方便的部署到你的应用。

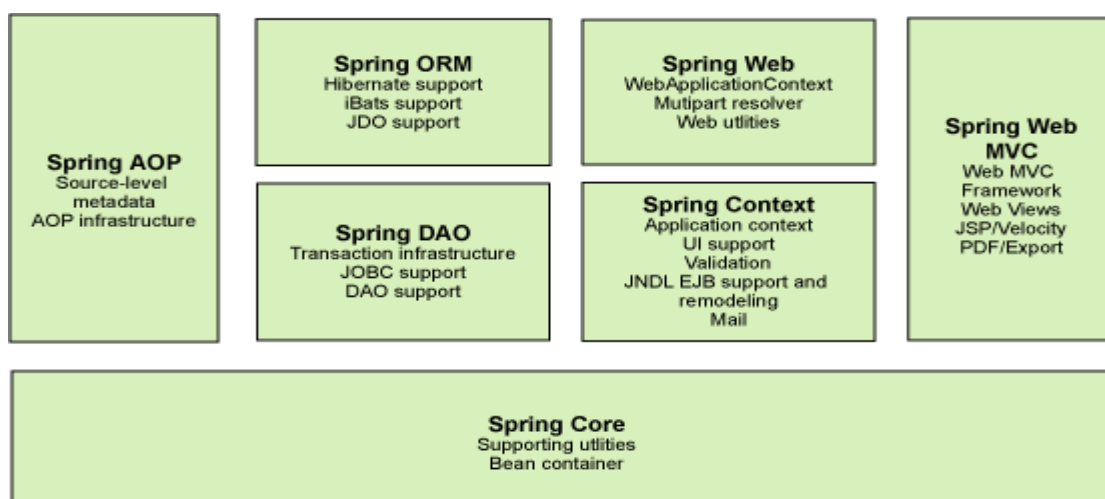


图 1.1 Spring 七大模块示意图^[2]

1.2.1 Spring Core

Spring Core 是 Spring 的核心容器，实现了 Spring 框架的基本功能 IoC（Inversion of Control，控制反转）。Spring 其它所有的功能模块都必须依靠它作为支撑。正如上所说，IoC 作为 Spring 框架的基础，因此 Spring Core 模块是其它所有模块的基石。

1.2.2 Spring AOP

通过该模块，Spring 实现了对 AOP 的支持。此模块为 Spring 实现对应用程序中事务管理的支持提供了一个基础。有了该模块，即使不使用 EJB 组件，Spring 也可以提供声明式事务管理^[2]。

1.2.3 Spring DAO

提供了 JDBC 抽象层，使 JDBC 编程更加简化。它简化了数据库的异常信息，提供了声明式事务管理和编程式事务管理两种事务管理方式^[3]。

1.2.4 Spring Context

它扩展了 BeanFactory 的功能，提供了 JNDI 访问、EJB 支持、远程调用(RMI)、java Mail、任务调度等企业级功能的支持。

1.2.5 Spring ORM

提供了“对象—关系模式映射”(Object-Relation Map)的支持。因此，它使得常见的工具如 Hibernate、iBats、JDO 等可以很好的集成到 Spring 应用中。

1.2.6 Spring Web

它为 Web 应用提供了上下文支持。还简化了 Web 应用中见参数绑定到对象的操作。

1.2.7 Spring MVC

Spring 提供的 MVC 框架。在 Spring 2.5 版本中，它包括 Spring Web MVC 和 Spring Portlet MVC 的支持。此外，它还提供了对集成其他 MVC 框架（如 Struts、JSF 等）的支持、提供了对 pdf 文档的支持等。

1.3 小结

这一章简单介绍了 Spring 框架和它的七大功能模块，如果觉得一开始就看到这些生涩难懂的术语，无法理解的话。可以先跳过此节，直接阅读第二章。

参考文献:

[1] 靳俊山.[Spring 的历史和发展趋势](http://blog.csdn.net/junnef).CSDN 博客:<http://blog.csdn.net/junnef>

[2] Naveen Balani.[Spring 系列: Spring 框架简介](#).IBM 中国

[3] 明日科技, 李钟尉, 冯东庆.Spring 应用开发完全手册.北京: 人民邮电出版社,2007.9

第二章 **Spring** 开发环境搭建

古人云：“工欲善其事，必先利其器”。这一章介绍如何搭配一个 java 开发环境。如果你对此不感兴趣，或者已经对此很熟悉，可以直接跳过。如果你并不想了解太复杂的配置方式，可以跳至“MyEclipse 下载与安装”。

2.1 系统需求

对于 JDK 来说，256MB 的内存就完全足够。可是现实是，我们要安装的是整个 java 开发环境 Eclipse 和它的插件 MyEclipse，所以这点内存是远远不够滴！要知道，在 MyEclipse 运行的时候，它会吞噬掉系统 230 多 MB 的内存，如果你还安装使用其他的 Eclipse 插件，那么内存使用还会疯狂的增长。如果你要做 Web 开发，还要运行 tomcat 服务器，这也会吞噬掉系统 70-100 MB 的内存。所以，要使 MyEclipse 能够流畅运行，至少要求 512MB 的物理内存，推荐 1G 以上。

另外，MyEclipse 运行时对 CPU 的计算能力要求也较高。我的电脑是赛扬 2.0G 的 CPU，在使用的过程中，电脑跑起来像乌龟一样，特别是在 Eclipse 中编辑 jsp 文件、切换工作区、编译、运行工程、启动 tomcat 的时候更是慢的要让人急出心脏病。所以，为了你的健康，强烈建议使用速度更快的 CPU☺。

2.2 JDK/JRE 下载与安装（可选）

2.2.1 下载

JDK 是 Java(TM) SE Development Kit 的简写，翻译为中文就是“Java 标准版开发工具包”。这是 java 程序开发和一个基本的平台。在这个包中也包含了 JRE(Java Runtime Environment, Java 运行时环境，仅对 java 程序的运行提供支持，不包含 JDK 类的源代码和 java 编译器等工具)。因此，下载了 JDK 后就不必再安装 JRE 了。但是，如果你觉得 JDK 安装包太大，也可以只下载 JRE，因为我们使用的 Eclipse 不需要 JDK 也可以编译 java 代码。但是由于它是由 java 语言开发的，所以要让它运行起来，就必须先在系统中安装 JRE。

到 sun 的官方网站下载页面 <http://java.sun.com/javase/downloads/> 下载最新的 JDK 稳定版本即可，现在最稳定的版本是 JDK6。下载页面如下：

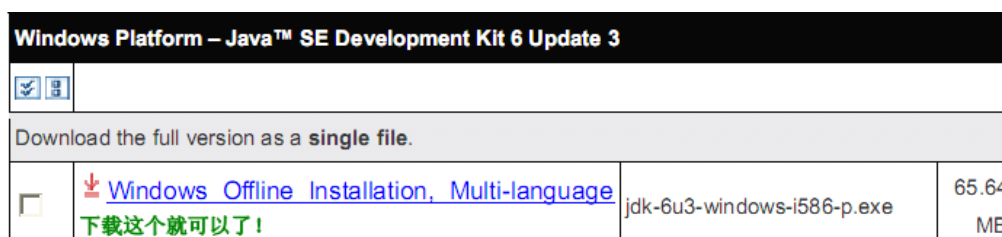


图 2.1 JDK 下载页面

注意：在下载时要看清操作系统版本。同时建议下载离线安装包。否则，如果你的网速不够快的话，在线安装会让你抓狂。

2.2.2 安装

JDK 的安装并不复杂。和其他任何软件一样，双击下载的到的 exe 文件，同意软件授权协议后，一路点击 Next 就可以了。需要注意的是，如果你的 Windows 安装在 C 盘的话，JDK 默认安装到 C:\java_program_files\Java\jdk1.6.0_01\，这不是一个理想的安装目录，原因大家都知道☺。所以我们可以把它安装到诸如：E:\java_program_files\Java\jdk1.6.0_01\之类的非系统分区的目录下。

2.2.3 配置环境变量（可选）

配置环境变量不是必须要做的。但是为了完整性，在这里对环境变量的设置方式做个介绍。步骤如下

- 1、在桌面**我的电脑**图标上点击右键，选择**属性**命令，弹出**系统属性**对话框。图 2.2

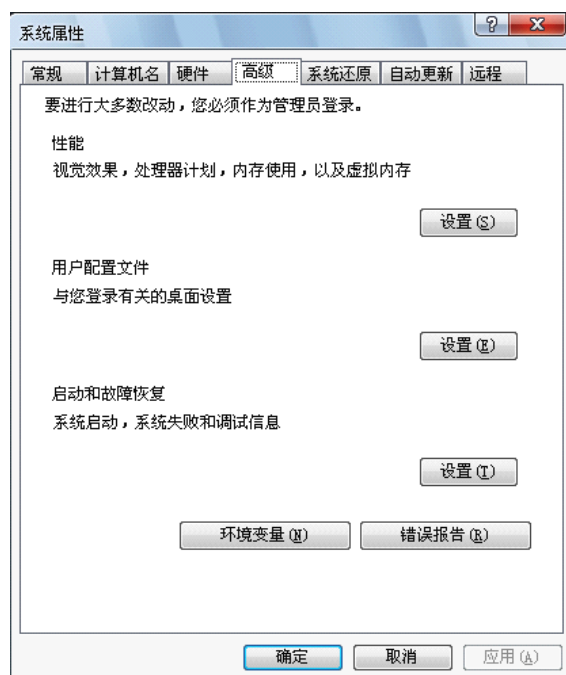


图 2.2 系统属性对话框

- 1、单击**环境变量**按钮，弹出**环境变量**对话框。
- 2、在**系统变量**标签下，单击**新建**按钮，弹出**新建系统变量**对话框，新建环境变量，名字为：**JAVA_HOME**，值为 JDK 安装路径，如“E:\java_Program_Files\Java\jdk1.6.0_01”（不包括括号）。完成后单击**确定**。
- 3、同第三步，新建环境变量：“**CLASSPATH**”，值为：“.;%java_home%\lib;%java_home%\lib\tools.jar”（不包括括号）。
- 4、在**系统变量**标签下，选中 Path 变量，单击**编辑**按钮，弹出**编辑系统变量**对话框，在变量值文本框中加入如下内容：“%java_home%\bin;%java_home%\jre\bin”（不包括括号）。
- 5、一路单击确定按钮，即可完成环境变量的设置。如果设置成功，在命令提示符下输入 javac 命令，并回车后，情况图 2.3:

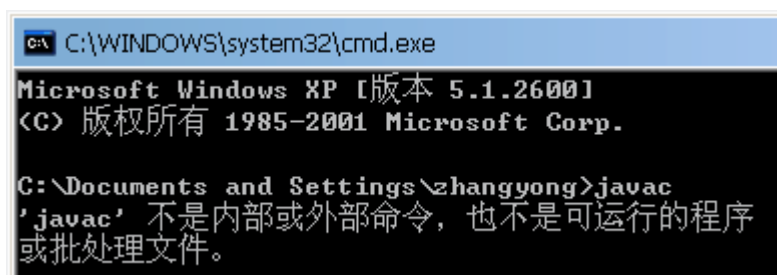


```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\zhangyong>javac
用法: javac <选项> <源文件>
其中, 可能的选项包括:
    -g               生成所有调试信息
    -g:none          不生成任何调试信息
    -g:{lines,vars,source} 只生成某些调试信息
    -nowarn          不生成任何警告
    -verbose         输出有关编译器正在执行的操
```

图 2.3 环境变量设置成功

如果显示如图 2.4 信息，则标识设置错误，可以再设置一次。



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\zhangyong>javac
'javac' 不是内部或外部命令, 也不是可运行的程序
或批处理文件。
```

图 2.4 环境变量设置失败

2.3 Eclipse 下载与安装

Eclipse 是蓝色巨人 IBM 释出的一款开源的平台，2001 年 11 月发布第一版——Eclipse1.0。开源，意味着免费，而且根据“公共/公众许可书（Common Public License, CPL）”，任何人都可以免费获得 Eclipse 和它的源代码并且可以任意修改，但是前提是其他人对修改后的软件也具有相同的权利。对现在的我们来说最实在的是免费这一点，其它就不管了☺。平台免费，但是并不意味着插件也一定免费，比如我们使用的 MyEclipse 就是一款收费插件，

还有下文提到的 JBuilder 也是收费插件，而且价格不菲。

Eclipse 是一个平台（Platform），仅为各种插件提供一组服务。它跨语言、跨平台。即是说它既可以做 java 开发的 IDE(Integrated Development Environment)，也可以做其他语言的 IDE，如 C/C++；既可以运行在 Windows 平台，也可以运行在 Linux 平台。但是，一般 Eclipse 下载回来后，就已经安装了 java 开发环境插件，具备一般的 java 开发能力。Eclipse 的一大优点是在它的之下，除了底层核心以外，所有的东西都是插件。它通过这种高扩展性的机制，很方便的为开发者提供各种功能。甚至，你也可以为了某种需要，自己打造自己的插件；或者，把它作为某个您的应用程序的界面框架，这时，它就不再是一个开发平台，而摇身一变成成为一个应用框架了。

由于这些优点，Eclipse 目前已成为最流行的 java 开发环境之一，甚至曾经一度辉煌一时的 Borland 公司的 JBuilder 也成为了它的一组插件。

在这个教程中，我选择 Eclipse 作为 Spring 的开发环境。在这里详细介绍 Eclipse 的下载与安装。

2.3.1 下载

打开网站 <http://www.eclipse.org>，点击黄色的 **Download Eclipse** 按钮后，转到 Eclipse 的下载页面，如图：

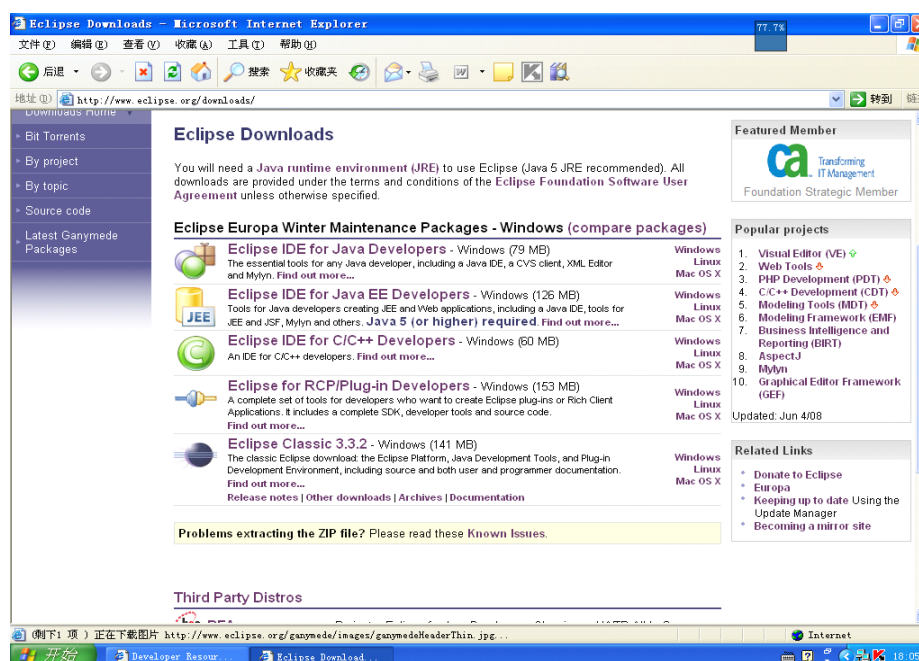


图 2.5 Eclipse 下载页面一

在这里你可以下载到最新版本的 Eclipse，目前最新的版本是 3.4。正如此页面所示，你可以从这里下载 eclipse 的各个不同的压缩包。分别介绍如下：

Eclipse IDE for Java Developers 是普通的压缩包,仅提供一般的java 开发;第二个 Eclipse IDE for Java EE Developers 提供简单的 Java EE 开发能力;第三个是 C/C++ 的开发包;第四个是为 Eclipse 插件和 RCP 开发做的开发包;第四个包含 Eclipse 平台, java 开发工具和插件开发工具。

现在由于 Eclipse3.4 刚刚释出,大多数插件都还不支持,所以我们下载上一个版本 Eclipse3.3.2,地址是 <http://download.eclipse.org/eclipse/downloads/>。单击 3.3.2 的版本连接后,选择一个镜像就可以下载了,如图 2.6。由于文件较大,建议使用迅雷等支持断点续传的工具下载。

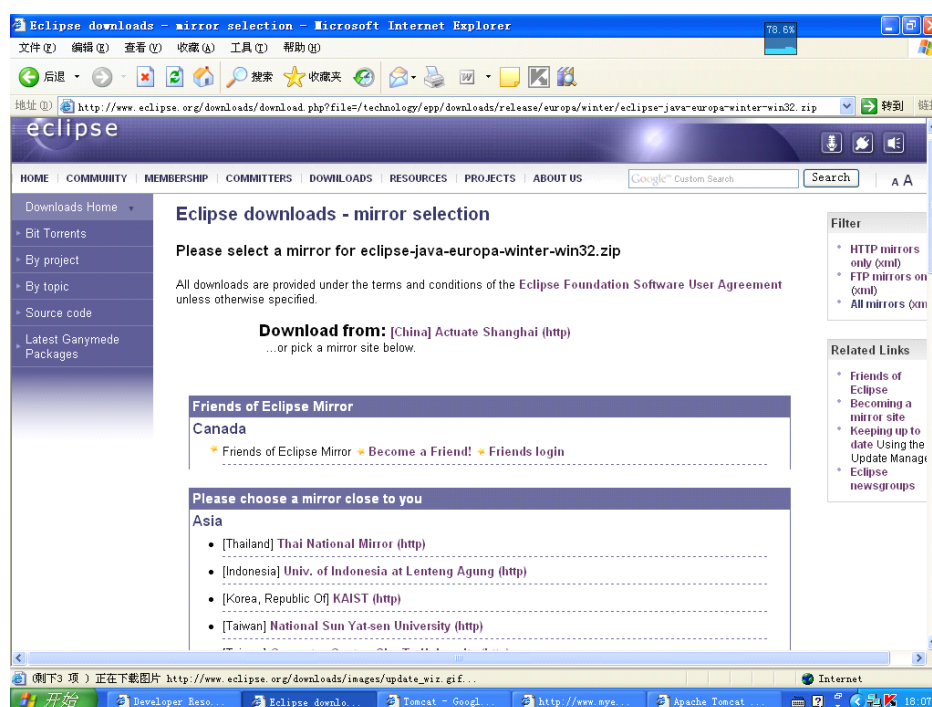


图 2.6 Eclipse 下载镜像选择

2.3.2 安装

下载完成后,你会得到一个 zip 格式的压缩包: eclipse-java-europa-winter-win32。Eclipse 是绿色软件,安装十分简单,把它用 WinRAR 等工具把它解压到任意目录就算完成了安装,这里是 E:\java_Program_Files 目录。

注意: 为了避免不必要的麻烦,路径中最好不要含有空格和中文,这对于本教程提到的其他软件也是一样。

2.3.3 运行与配置

进入 Eclipse 目录,双击 Eclipse 图标,即可启动它。在 Eclipse 启动的时候,它首先按照如下顺序在系统中搜寻安装的 JRE 中的 java.exe 文件: 1、Eclipse 启动时的-vm 参数; 2、

Eclipse 的安装目录下的文件夹，如 E:\java_Program_Files\eclipse\jre\bin\java.exe；3、系统环境变量 path；4、系统注册表。所以，如果我们的系统中也可以不安装 JDK/JRE，只需要从安装了 JDK 的电脑上把 JDK 安装目录中的 jre 文件夹拷贝到 Eclipse 安装目录下就可以了，如 E:\java_Program_Files\eclipse。如果拷到其他目录，则需要为 Eclipse.exe 加上启动参数：

Eclipse.exe -vm E:\java_Program_Files\Java\jre\bin\javaw.exe

具体操作步骤如下：在 Eclipse.exe 文件上单击右键；选择**创建快捷方式**命令为 eclipse.exe 创建运行快捷方式；在创建的快捷方式上单击右键，选择**属性**命令，出现 **eclipse.exe 属性**对话框如图 2.7：



图 2.7 快捷方式属性对话框

选择**快捷方式**页，在**目标**文本框中增加参数。增加参数后文本框内容如下：

E:\java_Program_Files\eclipse\eclipse.exe -vm E:\java_Program_Files\Java\jre\bin\javaw.exe

然后从快捷方式就可以启动 Eclipse 了，为了今后使用方便，可以把这个快捷方式复制到桌面和开始菜单中。这也是我为什么在前面说不必要安装 JDK 的原因。

Eclipse 启动的时候，要求选择一个目录作为工作区，如图：

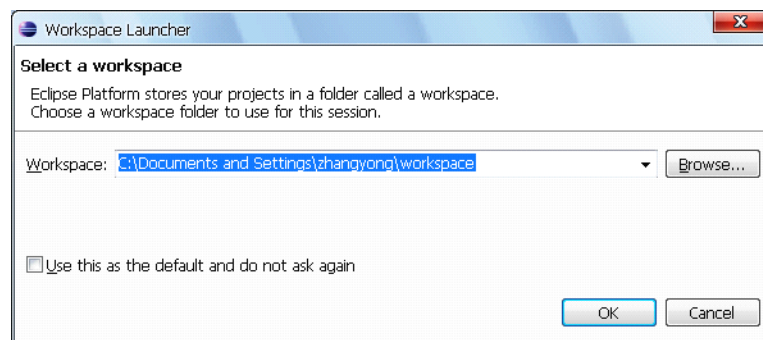



图 2.8 Eclipse 工作区选择

单击 **Browse** 按钮，选择一个目录作为工作区，即可启动 Eclipse。如果在下次启动时不想再选择工作区，可以勾选住 *Use this as the default and do not ask again* 复选框。Eclipse 启动后的欢迎界面如下：



图 2.9 Eclipse 欢迎界面

单击 close 按钮  关闭欢迎页，即可打开默认的工作区 **Java (default)** 图 2.9。这时，你就可以用它来开发 java 程序了。

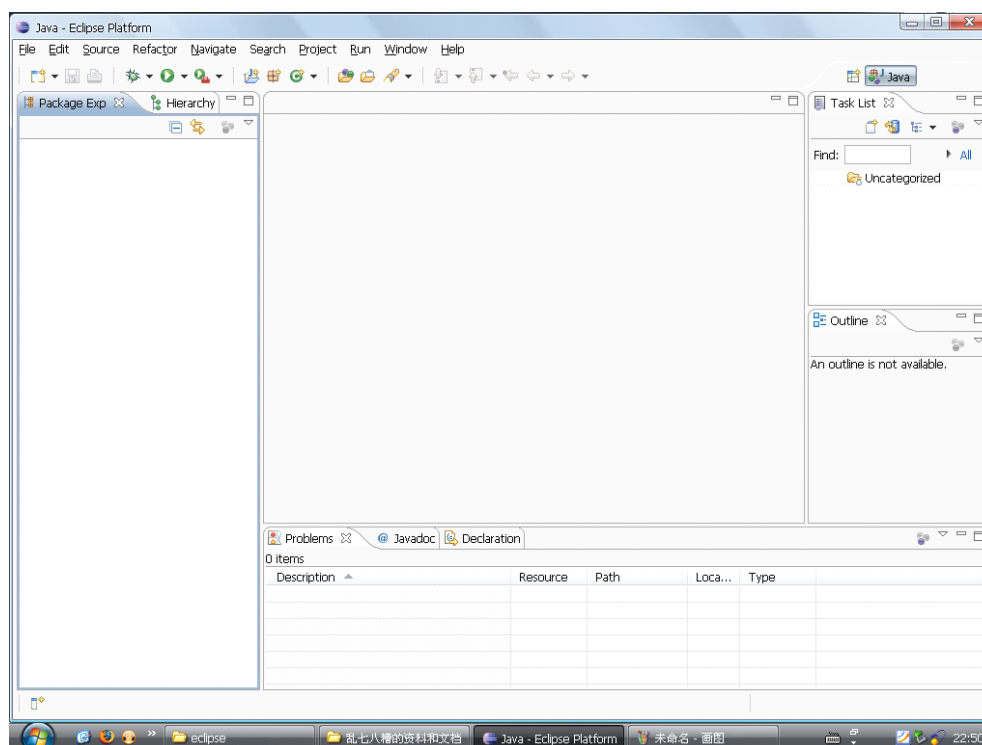


图 2.10 Java (default) 工作区

在编写 Java 源代码的时候, Eclipse 的代码提示功能可以及时的提示 JDK 类的方法和成员以及他们的说明。但是都是英文的, 对英文不好的朋友有点困难。我们可以去 sun 的官方网站下载 JDK 的中文帮助文档, 地址是 http://download.java.net/jdk/jdk-api-localizations/jdk-api-zh-cn/publish/1.6.0/html_zh_cn.zip, 下载后得到一个 zip 格式的压缩文件。把它加入到 Eclipse 中就可以提示中文注释了。步骤如下:

- 1、选择 **Windows** → **Preferences** 命令, 弹出 **Preferences** 对话框。
- 2、单击 **Java** → **Installed JREs**, 配置 Eclipse 中已经安装的 JRE。
- 3、需按则一个已经安装的 JRE, 单击 **Edit** 按钮, 弹出 **Edit JRE** 对话框。
- 4、在 **JRE system libraries** 中选择 **rt.jar**。单击 **Javadoc Location** 按钮。
- 5、在弹出的 **Java Doc** 配置对话框中选 **Javadoc in archive** 选项, 选中 zip 文件的路径, 和帮助文档在 zip 文件中的路径后一路 Ok 即可完成配置。

2.3.4 开发一个 HelloWorld 程序

现在, 我们利用 Eclipse 开发一个简单的 HelloWorld 程序, 熟悉使用 Eclipse (代码见例程 2.1)。详细步骤如下:

- 2 选择 **File** → **New** → **Project** 命令, 弹出 **New Project** 对话框, 选择 **Java** → **Java Project**。
- 3 单击 **Next** 按钮, 在 **Project name** 中填入工程名称“JavaHelloWord”后单击 **Finish**

按钮，创建工程。

4 选择 **File** → **New** → **Class** 命令, 弹出 **New Java Class** 对话框 (图 2.10), 创建一个类, 在 **Name** 中填入类名 “HelloWord”, 勾选 **public static void main(String[] args)** 和 **Generate comments** 复选框。让 Eclipse 为我们自动创建 main 方法和相关注释。

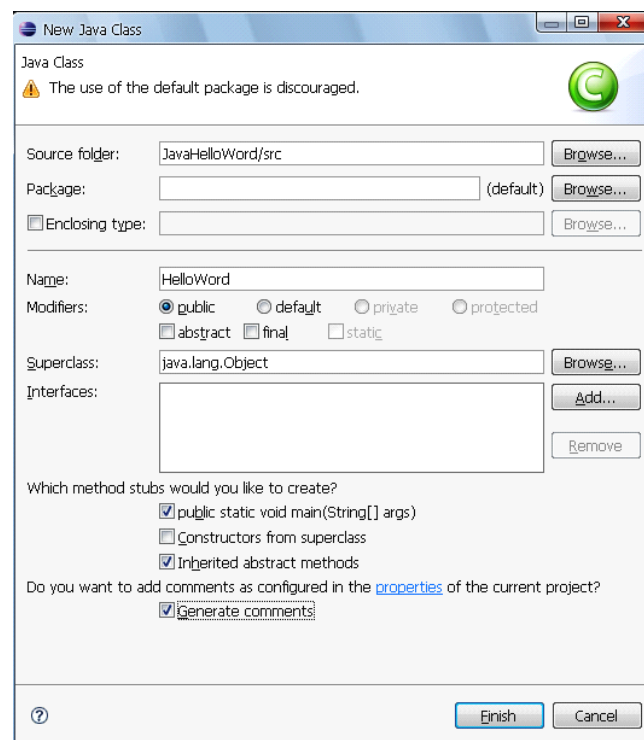


图 2.11 新建类

5 单击 **Finish** 按钮，新建一个类 HelloWord。

6 在生成的 main 方法中填入代码：

```
System.out.println("Hello,Eclipse");
```

完成后所有代码如下：

```
/**
 * @author zhangyong
 *
 */
public class HelloWorld {
    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Hello,Eclipse");
    }
}
```

7 单击运行按钮  运行程序，输出结果如下：



图 2.12 例程 2.1 运行结果

2.4 Eclipse 插件安装

2.4.1 直接复制文件 Eclipse 安装插件

这种安装方式十分简单。一般来说，Eclipse 插件下载回来后会得到一个压缩文件，解压缩文件后会得到一个名为 eclipse 的文件夹，该文件夹下有 features 和 plugins 两个文件夹，将这两个文件夹中的所有文件复制到 eclipse 安装目录下的对应文件夹即可。**注意不是复制文件夹，而是文件夹中的内容。**这种方式的优点是安装方式简单易行，但是缺点是显而易见的。插件的维护十分不方便，一旦安装成功后就很难卸载。比如你同时安装了 MyEclipse 和 SWT Designer 做 B/S 和 C/S 程序开发，做 B/S 开发的时候希望只加载 MyEclipse 插件，卸掉 SWT Designer，而做 C/S 开发的时候又希望只加载 SWT Designer 卸掉 MyEclipse，这种安装方法几乎是不可能做到这点的；另一方面，在重装 eclipse 后要安装插件，又要重新复制安装一次所有的插件，很是麻烦，所以推荐使用第二种方式。

2.4.2 使用 Link 文件安装 Eclipse 插件

这种插件的安装方式稍微比第一种方式复杂，但是维护起来却十分简单。方法是在 eclipse 安装目录下新建一个名为 links 的文件夹，在其中加入一些 link 文件（即文本文件，最好每

个插件对应一个), 让 eclipse 从这些文本文件中加载插件。由于 eclipse 启动时会自动搜索其目录下 links 文件夹中的所有文本文件, 所以新建的文件的扩展名不一定必须是.txt 或者一些文章中写的.link。但是推荐为.txt, 这样维护其内容比较方便——双击即可编辑。比如你的插件文件夹的路径是 E:\java_Program_Files\EclipsePlugins\MyEclipse 6.0。那么文本文件名字可以为 “MyEclipse_6.0.txt”, 内容如下:

```
path=E:\\java_Program_Files\\EclipsePlugins\\MyEclipse 6.0
```

注意: 路径可以为相对路径, 也可以为绝对路径, 但是分隔符必须是双斜杠。为了维护方便, 还可以在 links 文件夹下新建一个文件夹 “暂时不用的”, 将不用的插件的 link 文件放到此目录, eclipse 启动时就不会加载该插件了。

2.5 MyEclipse 下载与安装

MyEclipse 是一款商业的 java EE 开发插件, 最新版本是 6.5, 现在国内比较流行。它集成了 java 开发所需的大多数框架如 Spring、Hibernate、Struts、JSF、JPA 等。使用它我们就没必要满世界去 “跪求” 什么插件的下载地址了。但是遗憾的是 MyEclipse 直到现在最新的 6.5 版也还不支持 Struts2, 这真是让人摸不着头脑, 一个堂堂的商业开发软件, 在 Struts2 释出这么久了竟然还不支持。还有一点就是它是收费的, 而且价格不菲, 这也是很多人攻击它的原因之一, 但是我们有我们的解决办法, 呵呵, 天知地知、你知我知☺, 必要的时候为了节约一下还是去 “跪求” 一下吧, 网上很多好心人的。

2.5.1 下载

MyEclipse 的官方网站是 <http://www.MyEclipseIDE.com/>。下载地址为 <http://www.myeclipseide.com/module-htmlpages-display-pid-4.html>。如图:



图 2.13 MyEclipse 下载页面

其中,第一个 All in ONE 是 MyEclipse in ONE 版本下载连接,它集成了 JRE 和 Eclipse3.3 版。Archived Update Site 是一个 Eclipse 升级包,利用 Eclipse 的升级功能就可以安装 MyEclipse6.5 了。这里为了简便,单击 All in ONE 图标,下载 All in ONE 版本即可。

2.5.2 安装

下载完成后,我们得到一个可执行文件 MyEclipse_6.5.0GA_E3.3.2_Installer_A.exe,双击执行,开始 MyEclipse 的安装向导。如图:

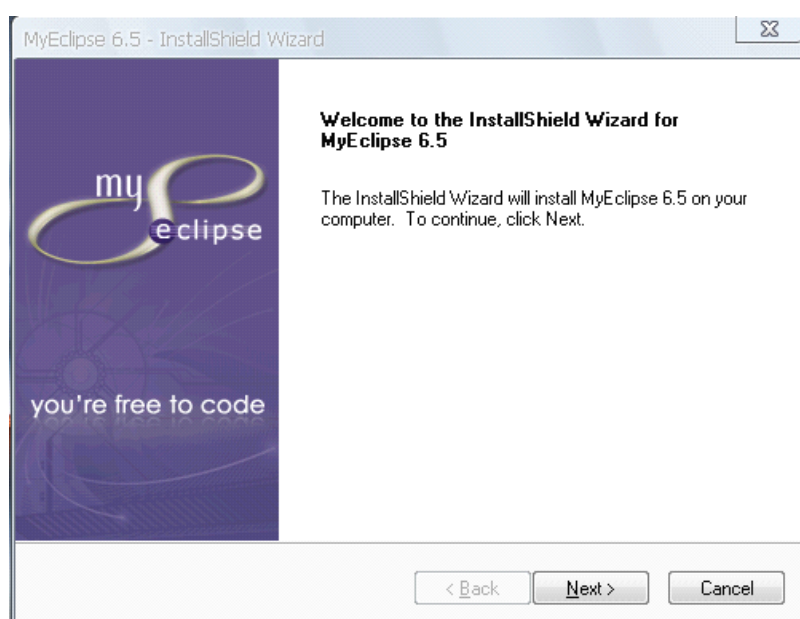


图 2.14 MyEclipse6.5 安装向导

单击 Next 后接收协议，选择安装目录就可以安装了，安装过程十分简单。事实上，安装向导只是拷贝相关文件到安装目录。我们可以看到 6.5 版的 MyEclipse 安装向导做了很大的改变，再也不像以前版本那样难看了，而且也不再需要 Executing ANT 了。安装速度有了很大的提升。

安装完成后目录结构如下：

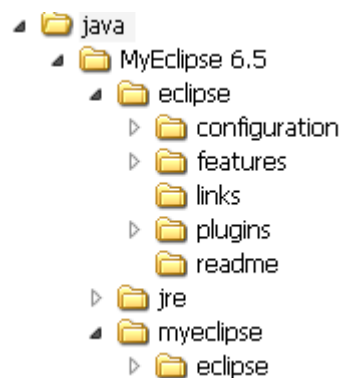


图 2.15 MyEclipse6.5 安装目录

其中 Eclipse 文件夹放的是 Eclipse 平台；jre 文件夹放的是 jre，Eclipse 启动所必须；myeclipse 文件夹中是 MyEclipse 插件目录。安装完成后，在开始菜单中找到快捷方式，就可以启动 MyEclipse6.5 了。

2.6 数据库下载与安装

做 java 开发，数据库必不可少。在 MyEclipse 中自带了一款出自 Apache 的开源数据库 Derby，它也是用 java 开发的，支持标准 SQL，对于开发来说已经足够使用了。在这里简单介绍其他两款常用的数据库管理软件的安装，一款开源 MySQL，一款商业 SQL-Server200。

2.6.1 安装 MySQL

MySQL 是一款优秀的数据库，是开源数据库的领军人物（不过好像“开源”得并不怎么彻底）。08 年 1 月，Sun 用 10 亿美元收购了 MySQL。MySQL 支持标准 SQL，支持多用户、多线程，体积小，速度快，可以说，它是 Java EE 应用的最佳搭档，但有个小小的缺憾，它不支持外键。MySQL 的官方网站是 <http://www.mysql.com/>，下载链接为：<http://dev.mysql.com/downloads/mysql/5.0.html>，界面如下：

Windows downloads (platform notes)			
Windows Essentials (x86)	5.0.51b	22.7M	Pick a mirror
MD5: c04a95d1eb8b525e6e7b4ba2532b8901 Signature			
Windows ZIP/Setup.EXE (x86)	5.0.51b	44.3M	Pick a mirror
MD5: 5170ecdeeb65aaf36415d026e7c5487e Signature			
Without installer (unzip in C:\)	5.0.51b	55.8M	Pick a mirror
MD5: 807aee1d3f2ee1097cd84074456abb79 Signature			
Windows x64 downloads (platform notes)			
Windows Essentials (AMD64 / Intel EM64T)	5.0.51b	27.1M	Pick a mirror
MD5: 718ef861305e95daa372abdb5f053966 Signature			
Windows ZIP/Setup.EXE (AMD64 / Intel EM64T)	5.0.51b	52.0M	Pick a mirror
MD5: f4205ca4fca5ef43730e627d33902a42 Signature			

图 2.16 mysql 下载页面

选择 **Pick a mirror** 即可下载。下载后得到一个 exe 文件，双击即可安装，一路 yes 或者 next 就行，安装完毕 MySQL 即可自动启动而无需额外配置。MySQL 没有图形管理工具，自带的命令行工具，黑乎乎的控制台既不好看也不好。所幸的是 MySQL-front 为它提供了一个图形管理环境，网址是 <http://www.mysqlfront.de/>，是一款小巧的绿色软件，使用很方便。界面如下：

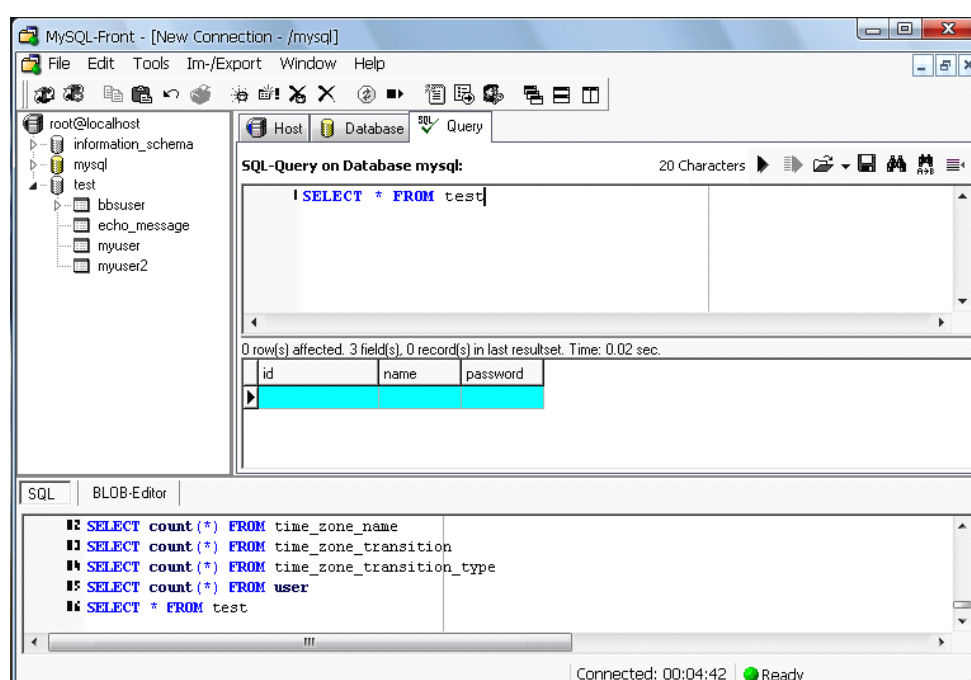


图 2.17 mysqlfront 运行界面

MySQL 的安装程序不带有 JDBC 驱动，因此要在 java 中连接 MySQL 数据库，还需要下载它的 JDBC 驱动程序，地址为 <http://dev.mysql.com/downloads/connector/j/5.0.html>。页面有两个包可供下载，一个是 tar.gz 包，用于 Linux 平台；一个是 zip 包，用于 Windows 平台。我们下载 zip 包即可。

2.6.2 安装 SQL-Server 2000

SQL-Server 是微软公司出版的一款商业数据库管理系统 (DBMS)，最新版是 2005，2008 版也即将面世，但是我们用不着那么笨重的数据库——只安装文件就有好几百兆。SQL-Server2000 对于我们来说已经足够了。由于是商业数据库，所以它的工具、文档也相对较完备齐全，使用更加简单，功能也比较丰富。美中不足的是收费，一分钱一分货嘛，倒也货真价实。本文就将使用 SQL-Server2000 作为开发数据库。下面介绍 SQL-server2000 的安装。

把 SQL-Server2000 安装光盘放入光驱，系统会自动运行 SQL-Server2000 的安装菜单。如果不能自动弹出，则打开光盘，运行 AutoRun.exe 可执行文件。

- 1、 选择 **安装 SQL-Server2000 组件** → **安装数据库服务器** 打开 SQL-Server2000 安装向导。

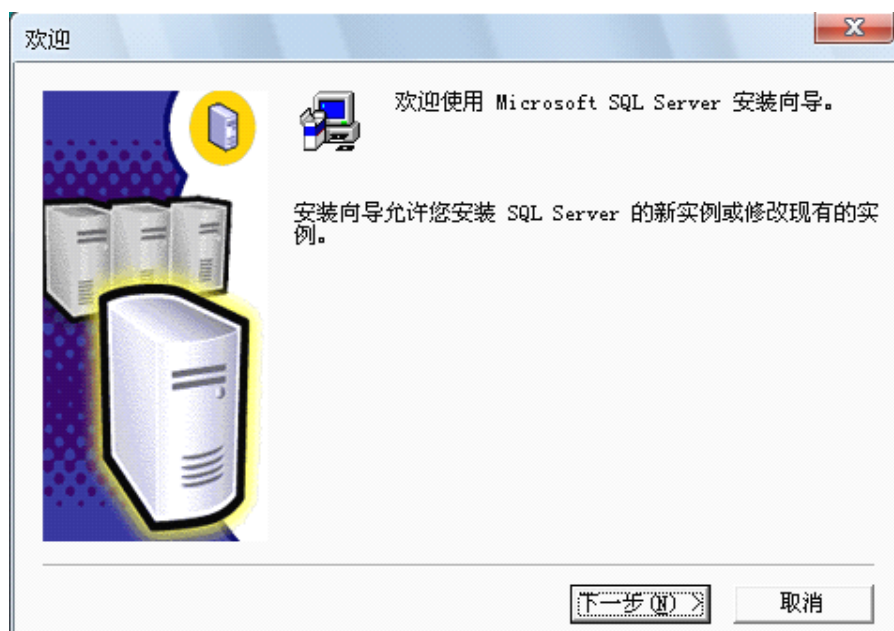


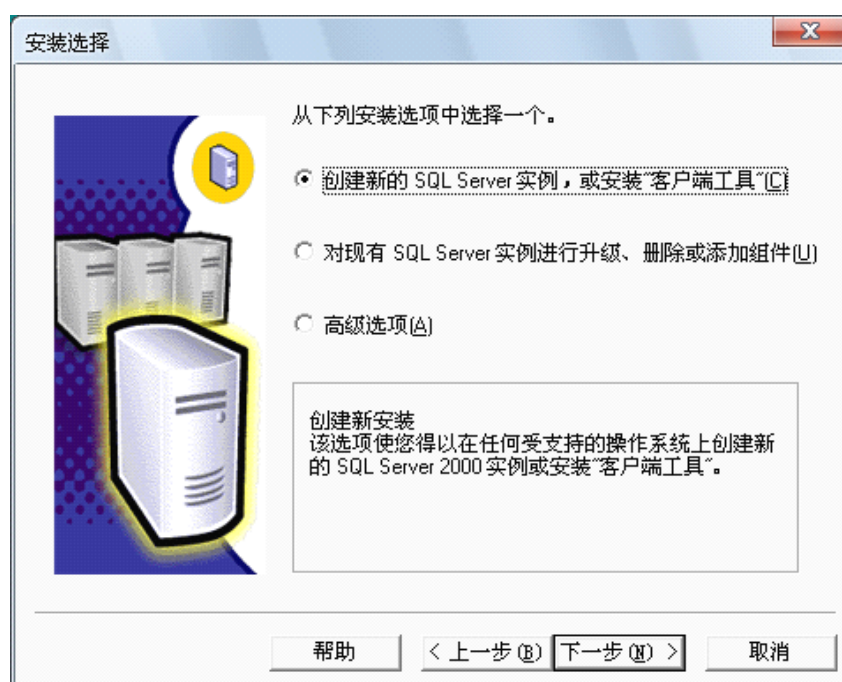
图 2.18 SQL-Server2000 安装向导

- 2、 出现欢迎界面后，单击 **下一步**。
- 3、 在 **计算机名** 对话框中可以选择要安装到哪台计算机，我们可以把 SQL-Server 安装到局域网或者域中的其他计算机上。在这里，保持默认设置，单击 **下一步**。



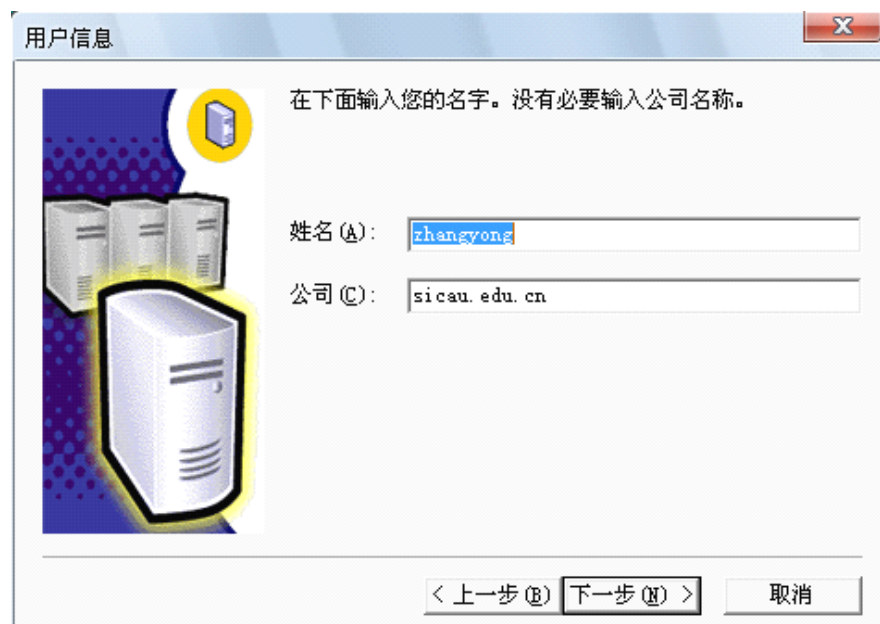
图 2.19 选择安装到的计算机

4、 进入**安装选择**对话框，在这里选择本次要执行的任务。因为我们要安装 SQL-Server 服务器，故选中**创建一个新的安装 SQL-Server 实例，或安装“客户端工具”**，单击**下一步**。



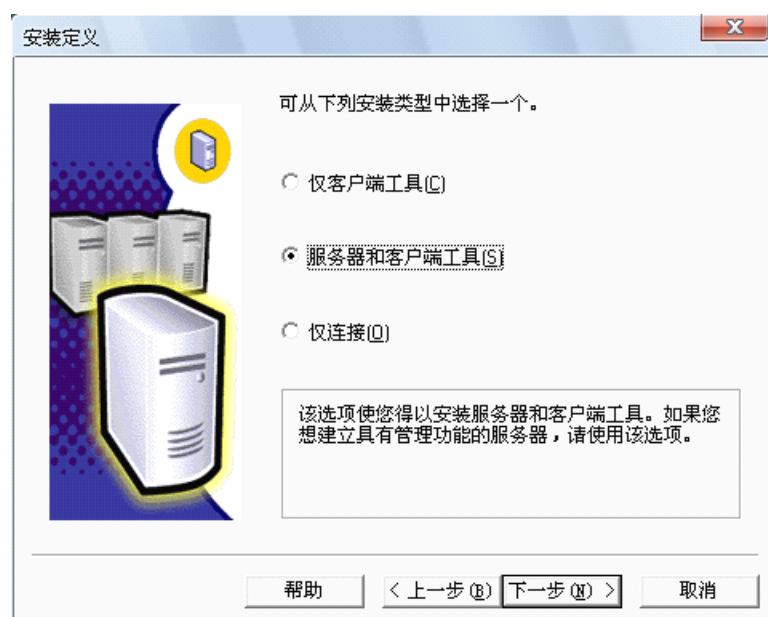
2.20 安装选择

5、 在**用户信息**中输入自己的姓名和公司名称，可以随便填。单击**下一步**。



2.21 填入用户信息

- 6、 在**软件许可协议**对话框中接受协议，单击**下一步**。
- 7、 在**安装定义**中选择服务器和客户端工具，单击**下一步**。



2.22 安装定义

8、 取消选择**默认**复选框，在**实例名**中填入实例名称，名称必须是 SQL-Server 的合法标识符，如图 2.23。SQL-Server 允许在一台计算机上运行几个不同的 SQL-Server 服务器，为了便于管理，需要给每个实例起个名字，如果选择默认，安装向导将随机分配一个字符串。如果系统中已经安装了一个 SQL-Server 的实例，则**默认**复选框这项灰色的，要求必须指定一个与已存在的实例名字不同的字符串作为新安装实例的名字。

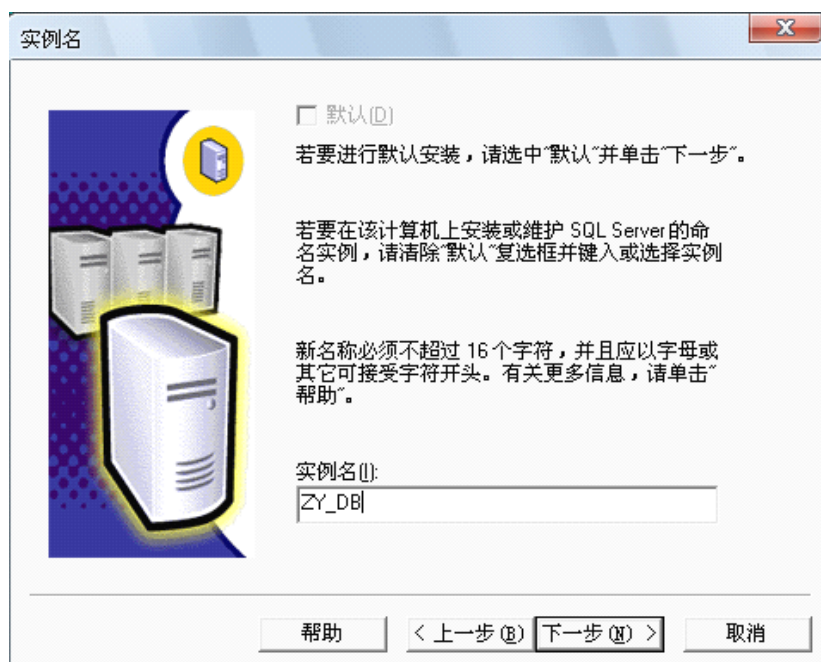


图 2.23 SQL-Server2000 安装向导输入实例名

9、在**安装类型**中，SQL-Server 为我们制定了几种安装方式。为了方便，一般选择**典型**的安装方式，单击**下一步**。

10、在**服务帐户**中选择**对每个服务使用同一帐户，自动启动 SQL-Server 服务**和**使用本地系统帐户**，让 SQL-Server 的各项服务以本地系统帐户启动。单击**下一步**。

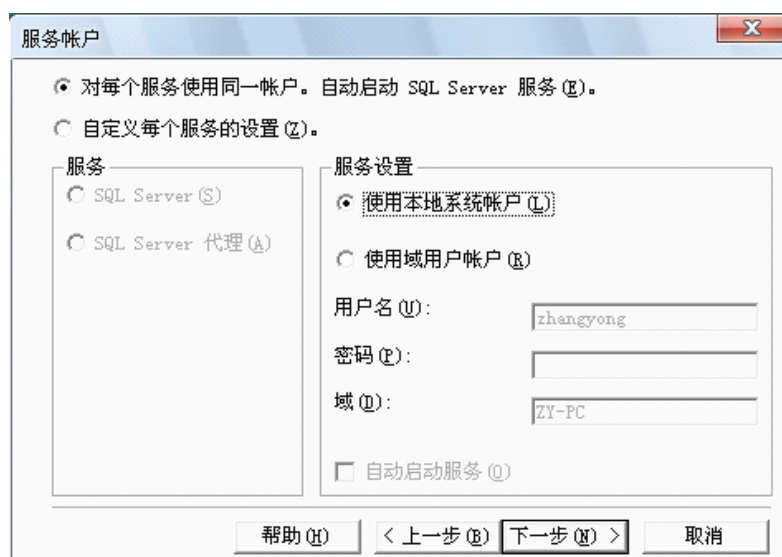


图 2.24 选择服务的启动帐户

11、弹出**身份验证模式**对话框，在这里，需要我们选择 SQL-Server 的身份验证模式。选择**混合验证**模式，填入系统默认账号 sa 的密码。关于 SQL-Server 的身份验证模式的更多介绍，请查阅其他关于 SQL-Server 的资料。单击**下一步**，安装向导会自动为我们在计算机上安装好 SQL-Server，完毕后向导会提示。

安装完成后，就可以使用它作为我们的开发数据库了。但是要能在 java 中连接 SQL-Server，就必须下载它的 JDBC 驱动程序。SQL-Server 的 JDBC 驱动实际上是 4 个 jar 文件，用的时候把他们加入到 BuildPath 或者拷贝到工程中的 lib 目录下就可以了。到 <http://www.microsoft.com/> 网站中搜索“SQL-Server JDBC”就可以找到它的下载连接了。

注意，有时候，在安装 SQL-Server 的时候，会安装失败，出现“挂起文件的错误”，解决办法也很简单。在注册表编辑器中展开到 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager，删除 PendingFileRenameOperations 键，重新安装即可。

2.7 Tomcat 的下载与安装

2.7.1 Tomcat 下载与安装

由于 MyEclipse 已经内置了一个 Tomcat 服务器，所以你也可以不用下载安装 Tomcat。

Tomcat 是一个优秀的开源 JSP 服务器，是 Apache 组织下的一个项目，使用非常广泛。目前最新的版本是 6.0，支持 JSP2.1 和 Servlet2.5。我们可以到它的官方网站去下载：<http://tomcat.apache.org/>，打开页面后，单击左侧的 Tomcat 6.X 链接来到 Tomcat6.0 下载页面，如图：

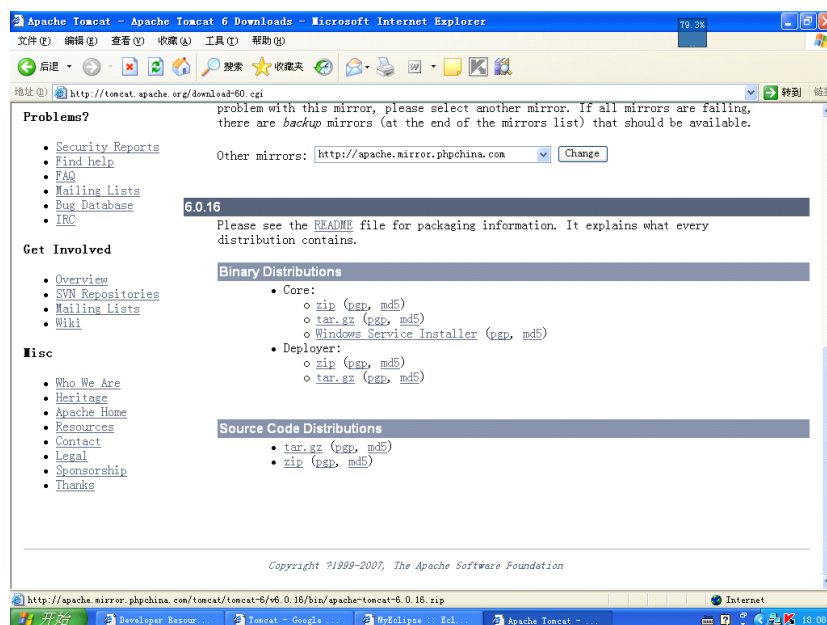


图 2.25 tomca6.0 下载页面

它提供了两种下载，Binary Distributions 是二进制文件包，不包含源代码。Source Code Distributions 是源代码包。我们下载 Binary Distributions 包就可以了。单击 Core 下的 zip 链接，即可以下载它的 Windows 下的 zip 压缩包，解压缩就可以使用。其中 tar.gz 是 linux 下

的安装包。Windows Service Installer 是 Windows 下的安装包，不必下载这个文件，安装起来太麻烦。

将下载的 zip 文件解压缩后，就可以使用，不必配置环境变量，在 MyEclipse 中也是如此。相反，如果配置了 CATALINA_HOME 环境变量，系统中则只能启动设置了环境变量的那个 Tomcat 了，其它的就永远启动不了。如果要使用多个 Tomcat，就不要设置环境变量。如果你非要设置，操作也很简单：新建一个名字为 CATALINA_HOME 的环境变量，它的值为 Tomcat 的安装路径。

进入 Tomcat 目录下的 Bin 文件夹，双击 startup.bat 文件，就可以启动服务器。在启动的过程中，Tomcat 会输出大量的日志信息，如果出现了“信息：Server startup in 66761 ms”，则表示启动成功，这时，在浏览器地址栏中输入 <http://localhost:8080/>，回车后看到如下界面，则安装成功。关闭时只需关掉日志窗口就可以了。如果双击 startup.bat 一闪而过，不出现任何信息，则可能是 JDK 没安装好，或者是 JDK 的环境变量没设置好，检查一下环境变量的设置就可以正常启动了。

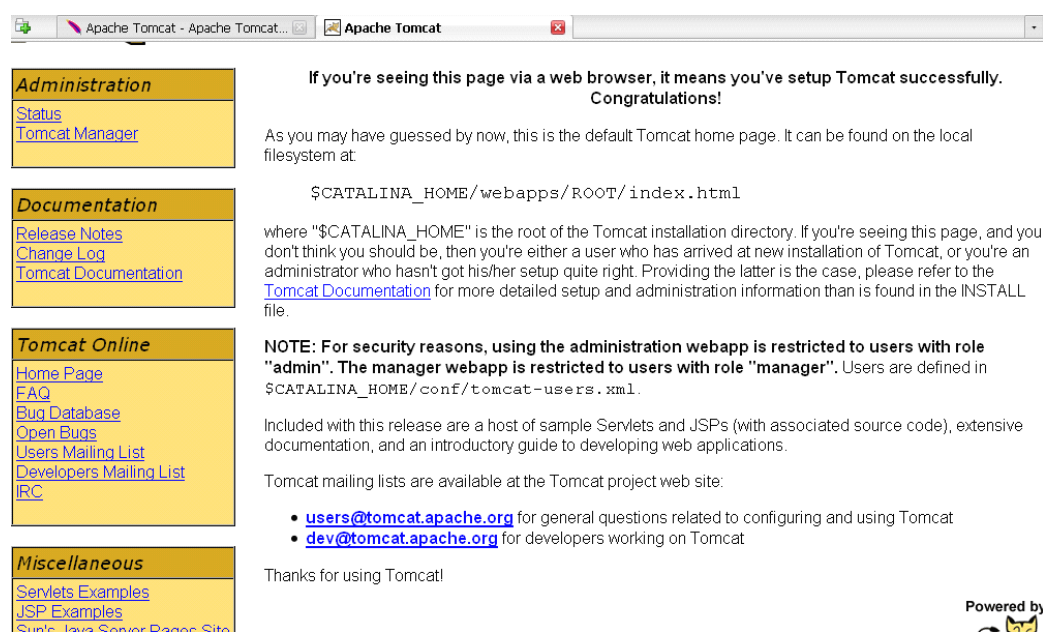



图 2.26 tomca6.0 起始页面

2.7.2 在 MyEclipse 中安装 Tomcat

要在 MyEclipse 中使用我们下载的 Tomcat，就必须先将 Tomcat 添加到 MyEclipse 的管理之中。步骤如下：

单击 Eclipse 工具栏上的服务器旁边的下拉按钮 ；单击 **Configure Server** 命令，弹出 **Preferences** 对话框；在弹出的对话框中选则 **MyEclipse Enterprise Workbench** → **Servers**

→ **Tomcat** → **Tomcat 6.x**。如图：

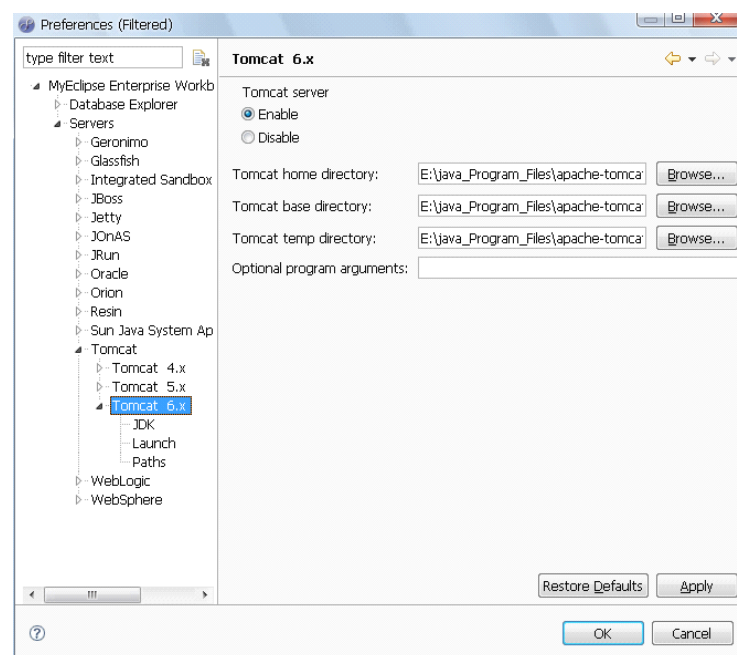



图 2.27 Tomcat6.0 配置

单击 **Browse** 按钮，选择 Tomcat 的解压目录，确定；选择 **Enable** 按钮，启用 Tomcat6.x；单击 **OK** 完成配置。这样，我们就可以单击按钮  看到我们配置的服务器了，选择 **Tomcat6.x** → **Start** 命令就可以启动 Tomcat。其日志信息会在 **Console** 视图中输出。

2.8 打造 java 开发绿色环境

虽然 MyEclipse6.5 较之以前版本，安装速度已经有很大的改观了。但是如果你想把 MyEclipse 移动到你的移动硬盘上去，做一个移动的开发环境，或者要传到其他电脑上去，或者要批量安装，那么就要试试 MyEclipse 的绿色版本了。打造一个移动开发环境非常简单，首先进入到 MyEclipse 的安装目录，把 jre 文件夹拖到 eclipse 文件夹中，这样系统中即使没有安装 jre，MyEclipse 也可以启动。打开 eclipse 文件夹下的 links 文件夹，把 link 文件中 path 后的路径由绝对路径改成相对路径就可以了，我的 link 文件原来的内容是 path=E:/JavaBook/MyEclipse6.5/myeclipse，修改为 path=../myeclipse（在 Windows 中 ../ 表示上一级目录），保存后就可以启动 MyEclipse6.5 了。

另外，有人做了 MySQL 绿色版，Google 一下就可以找到下载连接，下载后就可以直接使用。如果不想找，也可以直接使用 MyEclipse 自带的 Derby 数据库。

注意：1、这里的 jre 文件夹不一定非要 MyEclipse 安装文件夹下的 jre 不可，也可以是从 Sun 下载的 JDK/JRE 安装目录下的 jre 文件夹，但是 MyEclipse 自带的 jre 要小一些，因为

它删除一些不必要的文件,可以节约一点硬盘空间; 2、我们需要更改所有 link 文件夹的 path 路径; 3、要同时把你插件目录也复制到你的移动硬盘上。

2.9 Spring 之 HelloWorld

到这里,我们就已经完成了一个 Java 开发基本环境的搭建工作了,现在,我们来做一个 Spring 的 HelloWorld 程序(源代码见例程 2),体会一下 Spring 开发。在这个例子中,我们创建一个 Animal 类,利用 Spring 的 IoC 功能为其 name, age 等成员注入值。然后在 HelloTest 类的 main 方法从 Spring IoC 容器中取的该类的实例,并将其信息在控制台打印出来。步骤如下:

1、选择 **File** → **New** → **Java Project** 命令,弹出 **New Java Project** 对话框,在 **Project name** 中输入 HelloSpring,单击 **Finish**,完成工程创建。

2、接下来就要为工程添加 Spring 开发能力。选择 **MyEclipse** → **Project Capabilities** → **Add Spring Capabilities** 命令,弹出 **Add Spring Capabilities** 对话框:

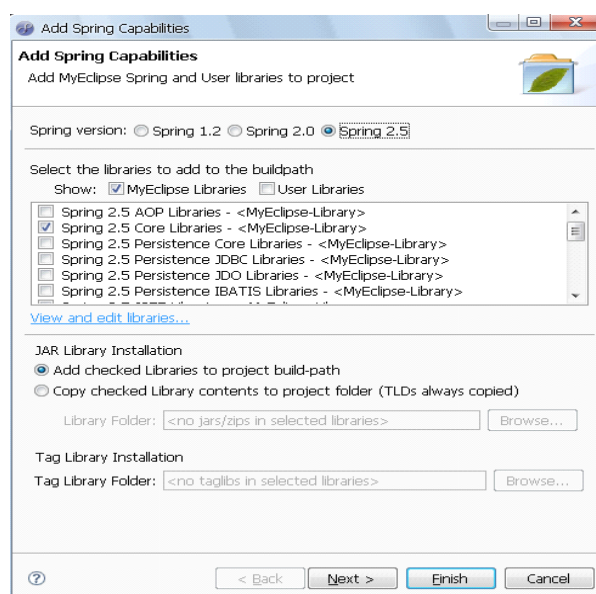


图 2.28 Add Spring Capabilities 对话框

在对话框中 **Spring Version** 可以选择 Spring 的版本,我们默认为 2.5。可以在列表框里选择 Spring 组件,我们只需要核心库,故选择 **Spring Core Libraries** 即可。**JAR Library Installation** 可以选择 jar 文件的安装方式, **Add checked Libraries to project build-path** 选项表示只是把 jar 库的路径加入到 build path 中,在我们练习时,可以选择此项,节约硬盘空间; **Copy checked Libraries contents to project folder** 则把选中 jar 文件复制到工程目录中,这样我们在打包发布工程时就不需要拷贝工程所依赖的 jar 文件了,做

项目时采用这种方式。*Tag Library Folder* 是标签文件的存放目录。

- 3、单击 **Finish**，这个 Java Project 就具有了 Spring 开发的能力了。
- 4、在 **Package** 视图的 *src* 节点上单击右键，选择 **New** → **Package** 命令，在弹出 **New Java Package** 对话框的 *Name* 中填入 Hello，新建一个名为 Hello 的包。
- 5、在 Hello 包中分别创建一个动物类和一个含有主方法的测试类，名字是 Animal 和 HelloTest。完成后，情况如下：

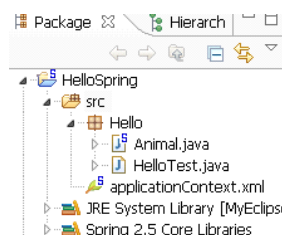


图 2.29 SpringHelloWord 完成的类视图

- 6、修改 Animal 类的代码，添加 name, age, myClass 成员和 Speak 方法。具体代码如下：

```
package Hello;

/**
 * @author zhangyong
 *
 */
public class Animal {
    private String name; //动物名字
    private int age; //动物年龄
    private String myClass; //动物类别

    /**
     * 动物介绍
     * */
    public void Speack() {
        System.out.println("我是: " + this.myClass + ",我的名字叫: " + this.name
+ ",我: "
            + this.age + "岁! ");
    }
    //Getter和Setter方法以后将省略
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getMyClass() {
        return myClass;
    }
    public void setMyClass(String myClass) {
        this.myClass = myClass;
    }
}
```

7、 修改 HelloTest 类，完成后代码如下：

```
package Hello;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 *
 * @author zhangyong
 *
 */
public class HelloTest {

    /**
     * @param args
     */
    public static void main(String[] args) {

        //加载Spring 配置文件，初始化IoC容器
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");


        //从容器中获取受管bean
        Animal animal = (Animal) ac.getBean("Animal");

        //调用bean的Speak方法，输出bean的属性信息
        animal.Speak();
    }
}
```

8、 修改 Spring 的配置文件 applicationContext.xml，添加一个 bean 定义。完成后代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="Animal" class="Hello.Animal">
        <property name="myClass">
            <value>猴子</value>
        </property>
        <property name="name">
            <value>瓜瓜</value>
        </property>
        <property name="age">
            <value>2</value>
        </property>
    </bean>
</beans>
```

9、单击运行按钮 ，运行该程序，Console 视图输出如下：

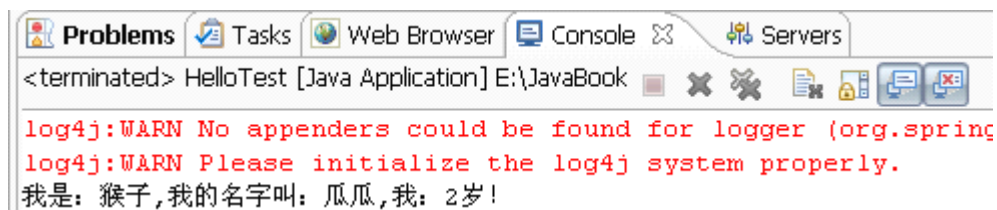


图 2.30 SpringHelloWord 运行输出

可以看到，通过Spring，我们可以在xml配置文件中为类的属性赋值。如果在下次运行时，我们希望的动物是一只1岁的叫姗姗的猪，那么我们只需要修改Spring配置文件的相应内容即可，从而不再像平时一样修改源代码，输出结果就可以很容易的变为“我是：猪,我的名字叫：姗姗,我：1岁！”。

注意:在Console视图中，我们看到输出这样警告信息：

log4j:WARN No appenders could be found for logger

这是由于 Spring 利用 log4j 来输出日志信息，而 log4j 没有找到配置文件的警告信息。我们只需要在 src 目录下新建一个名为 log4j.properties 的文件，就可以解决问题,文件内容如下：

```
log4j.rootLogger=WARN,stdout  
log4j.appender.stdout=org.apache.log4j.ConsoleAppender  
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n
```

2.10 小结

在这一章中，我们介绍了 Java 开发常见软件的安装与配置，由于有了 MyEclipse6.5 All in ONE，大多的软件安装配置并不是必须的。但是为了让读者熟悉 java 各种软件的配置，还是做了详细的介绍。本章还通过一个简单的例子让读者初步尝试 Spring IoC 开发的过程。

第三章 IoC In Spring

3.1 什么是 IoC

IoC 即 Inversion of Control，控制反转。它不是一种编程技术，而是一种设计模式。利用它，可以促进应用程序组件或类之间的松散耦合。一般的情况下，我们需要自己创建对象的实例，比如在 Computer 类中创建一个打印机的实例：

```
Printer printer = new DellPrinter(); // 创建一个戴尔打印机的实例
```

如果使用 IoC，我们则不必再自己创建类的实例，而是将创建对象的任务交给 IoC 容器去做。这样，我们在使用某个对象的时候，就可以直接从容器中获取一个实例。就像一个好莱坞的原则一样“Don't call me.I'll call you!”，容器会在合适的时候创建它。我们就像不必考虑对象的销毁一样，也可以不必考虑对象的创建，这就是控制反转。

这样做有什么好处呢？在上面的例子中，我们为电脑安装了一个戴尔打印机（创建一个戴尔打印机类的实例）。可是如果有一天，我们的系统要升级了，需要使用惠普的打印机。这时，由于我们把打印机已经“固化”到电脑中了（打印机和电脑的耦合度很高），要更换打印机，就不得不先拆开电脑，从主板中卸下打印机，再安装新的打印机（修改 Computer 类的源代码，然后重新编译，重新部署）。可以看到，这是一个计算机专家才能完成的任务，对于一个对计算机不是很了解的人来说，这将会是一场噩梦。那么如何解决这个问题呢？办法之一就是我们把打印机抽象成一个接口：

```
public interface Printer {  
    public void print();  
}
```

这样各个厂商在生产打印机的时候，就只是需要实现这个接口所定义的功能就行了。就像我们在电脑上做一个专门的打印机模块，需要的时候卸掉原来的模块，再安装上新型号就的打印机可以了，前提是新的模块必须和以前的有一样的功能和调用接口。在电脑（Computer）中则变成了这样：

```
private Printer printer; // 提供一个打印机的接口  
  
// 提供一个安装打印机的方法  
public void setPrinter(Printer printer) {  
    this.printer = printer;  
}
```

这样看似很好的解决了上述问题。但是实际上电脑(Computer)和打印机(Printer)之间的耦合性仍然很高：打印机的代码写入了电脑中。我们不得不在使用之前先调 `SetPrinter` 方法为电脑“安装”打印机。但是，电脑使用者仅仅想使用打印机而已，这里却不得不做安装打印机的工作。

这种依赖关系在“分离关注”的思想下是无法忍耐的，于是产生了新的模式，即控制反转 (IoC)，也叫依赖注入 (dependency injection, DI)。使用这种方式，电脑(Computer)只是需要提供一个打印机(Printer)的“属性”和安装打印机的“渠道”。如果需要（当需要调用打印任务时），IoC容器会在合适的时候自动为电脑注入（安装）打印机。就好像我们请了个网管，我们要使用打印机的时候，可以直接告诉网管“我需要使用惠普打印机”，然后网管就自动的找到并安装好惠普打印机，我们就可以使用它了。这样电脑使用者就可以完全不必关心打印机而只需使用它就行了。代码如下：

```
public class Computer {
    private Printer printer;
    public Print() {
        printer.print();
    }
    public Printer getPrinter() {
        return printer;
    }
    public void setPrinter(Printer printer) {
        this.printer = printer;
    }
}
```

```
<bean id="Computer" class="com.Computer">
    <property name="printer" ref="Printer"/>
</bean>
```

Spring 框架为我们提供了一个优秀的 IoC 容器。我们只是需要在它的配置文件中指明我们需要那个类的实例，这个类在那里，Spring 的 IoC 容器会在合适的时候自动的为我们注入这个类的实例。从而我们可以完全不必考虑对象的创建工作，还可以实现应用程序各个组件的“即插即用”，而不需要修改源代码。

Spring 提供多种配置文件的形式，但是推荐使用 xml 配置文件来驱动应用，尽管它也支持.properties 文件和数据库的配置形式。

在 Spring 中，被容器管理的 Java 类被称之为受管 Bean、Bean，或者受管 POJO，他们

是非常普通的 java 类。在本书中，不必分辨这些名称上的异同。

3.2 如何配置受管 Bean

Spring为我们提供了多种配置驱动方式，这里仅介绍官方推荐使用的xml方式配置。在Spring1.x里，Spring支持的是DTD的配置方式，由于其表达数据的能力不够强大，到了2.x里，在业界的强烈要求下，Spring提供了xml Schema的配置驱动方式，但是为了向下兼容，Spring2.x仍然支持DTD的配置。到现在最新的2.5版，甚至支持基于Annotation注解的驱动方式，但是前提是JDK版本必须为1.5以上。

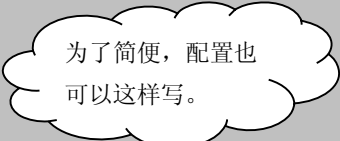
在 MyEclipse6.5 中，我们为 Project 添加 Spring 开发能力时，已经为我们创建了一个默认的 Spring 配置文件，名字为 applicationContext.xml。MyEclipse6.5 已经为我们添加了对 xml Schema 支持，即引入了 Spring 的命名空间。

在 Spring 中，我们可以使用多个配置文件，将不同的 bean 配置放在不同的文件中，这样便于管理与维护。比如数据库相关配置在 DataConfig.xml 中定义，Action 在 ActionConfig.xml 中定义，然后在 applicationContext.xml 中利用<import/>标签引入即可，Spring 再加载配置文件时会自动的加载这些文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
                           beans-2.0.xsd">

  <import resource="DataConfig.xml"></import>
  <import resource="ActionConfig.xml"/>

</beans>
```



在配置文件中，我们还可以使用<description/>标签来为各个配置文件增加注释，方便阅读和管理。<description/>是<beans/>的子标签，故只能在<beans>和</beans>之间使用。而另一种注释方式是<!-- -->，它可以在 xml 配置文件的任意位置使用。语法如下：

```
<beans>
  <description>
    这里是一些描述信息，你可以写任何的符号
  </description>
  <!--
    这里是注释
  -->
</beans>
```

3.2.1 使用<bean/>标签

<bean/>标签是<beans/>的子标签，用来定义一个受管 Bean，最简单的 Bean 必须要指定用 class 属性来指定该 Java Bean 的类（必须包含类路径）。同时如果不是内部 Bean，则必须要指定其 id 或者 name 属性之一来标识该 Bean，这样我们才能引用到它。范例如下：

```
<beans>
  <bean id="MyBean" class="Hello.Animal"></bean>
  <bean name="MyBean2" class="Hello.Animal"></bean>
</beans>
```

除此之外，<bean/>标签有很多其他属性，以后将逐步介绍。这里用表格的形式给出部分常用属性的意义和用法。

表 3.1 <bean/>标签的部分属性

属性名称	属性描述	使用方法	默认值	备注
id	指定 bean 的 id	<bean id="ID"	没有默认值	如果不是内部 bean，则必须指定 id 和 name 中的一个作为该 Bean 在容器中的标识。
		class="MyBean.Mybean">		
name	指定 bean 的名称	<bean name="myname"	没有默认值	无备注
		class="MyBean.Mybean">		
class	指定 bean 的类名 (必须包含有类路径)	<bean id="ID"	没有默认值	在 bean 定义时必须存在
		class="MyBean.Mybean">		
scope	指定 bean 的作用范围	<bean id="ID"	Singleton(单例模式)	取值为：singleton、prototype、request、session、globalSession
		class="MyBean.Mybean"> scope=" prototype">		

		<code><bean id="ID"</code>		
lazy-init	是否延迟加载	<code>class="MyBean.Mybean"</code>	lazy- init =" true">	false 取值为 true 或者 false
autowire	指定自动装配的形式	<code><bean id="ID"</code> <code>class="MyBean.Mybean"</code> <code>autowire =" byName"></code>	No(不自动装配)	取值可以为: no、byName、byType 、 constructor 、 autodetect、 default
init-method	指定初始化方法	<code><bean id="ID"</code> <code>class="MyBean.Mybean"</code> <code>init-method =" initMethod"></code>	没有默认值	无备注
destroy-method	指定销毁方法	<code><bean id="ID"</code> <code>class="MyBean.Mybean"</code> <code>destroy-method ="</code> <code>destroyMethod"></code>	没有默认值	无备注

除了这些常用的属性外，Spring 还为我们提供了其他的如 `parent`，`abstract` 等配置属性，极大的方便了配置工作、减少配置的工作量和增加了开发的灵活性。

注意：在 java 程序中，我们可以定义一个内部类，那么在写编写 Spring 配置文件的时候，我们时候是否也可以配置一个内部 Bean 呢？当然可以。配置的方法如下：

```
<bean id="ClassA" class="Hello.ClassA ">
  <property name="ClassB">
    <bean class="Hello.ClassB"/>
  </property>
</bean>
```

这是一个匿名的内部 Bean

可以看到，我们不必指定内部 bean 的 id 或者 name 属性，因为 IoC 容器会忽略它的存在，也就是说，我们不能用形如 `applicationContext.getBean("school")` 的方法来获得该内部 bean 的实例。

3.2.2 为受管 Bean 注入值

使用`<bean/>`标签中定义好一个受管 Bean 后，就可以为该 Bean 的属性注入其依赖的对象。Spring 中有两个标签为我们提供了这个功能：一个`<property/>`，通过 Setter 方法注入；一个`<constructor-arg/>`，通过构造子注入。它们都是`<bean/>`的子标签。

1、 设值注入

设值注入就是 Spring IoC 容器首先通过无参数构造子实例化受管 Bean，再调用标准的 Setter 方法为其受管 Bean 的成员注入其依赖对象。可以使用标签<value/>注入 java 基本类型的值，也可以使用<ref/>标签注入对象。范例如下：

```
<bean id="Animal" class="Hello.Animal">
  <property name="myClass">
    <value>猴子</value>
  </property>
  <property name="name">
    <value>瓜瓜</value>
  </property>
  <property name="age">
    <value>2</value>
  </property>
</bean>
```

注意：如果要使用设置注入的方式，受管 Bean 就必须要有有一个无参构造子。否则容器会抛出 `BeanCreationException` 异常，提示没有找到默认的构造方法。

2、构造子注入

构造子注入是通过向 Bean 的构造方法中传入若干参数的方法，把其依赖的对象注入到受管 Bean 中，构造子的参数都必须是该受管 Bean 所依赖的对象。修改上一章例程 2.2 的配置文件，使用<constructor-arg/>标签就可以通过构造方法为受管 Bean 注入依赖对象了。修改后的配置文件如下（工程代码见例程 3.1）：

```
<bean id="Animal" class="Hello.Animal">
  <constructor-arg>
    <value>珊珊</value>
  </constructor-arg>
  <constructor-arg value="5"/>
  <constructor-arg value="猪"/>
</bean>
```

利用构造子注入为
Animal Bean 注入
值。

再修改 Animal 类，为其添加含有参数的构造方法，完成后代码如下：

```
/**
 * @author zhangyong
 */
public class Animal {
    private String name; //动物名字
    private int age; //动物年龄
    private String myClass; //动物类别

    Animal() {}
    Animal(String name, int age, String myClass) {
        this.name = name;
        this.age = age;
        this.myClass = myClass;
    }

    /**
     * 动物介绍
     */
    public void Speack() {
        System.out.println("我是: " + this.myClass + ", 我的名字叫: " +
            this.name + ", 我: " + this.age + "岁!");
    }

    //.....省略 Getter 和 Setter
}
```

我们只是需要添加这样一个构造方法即可。在使用设置值注入时，无参构造子才是必须的。

运行 HelloTest 后将会输出:



图 3.1 例程 3.1 运行结果

可以看到，通过<constructor-arg/>标签，Spring IoC 容器按照标签的顺序依次为构造方法的参数注入依赖对象。当然，我们也可以通过<constructor-arg/>标签的 index 属性为标签指定一个顺序，或者通过其 type 属性指定注入对象的类型。代码如下：

```
<bean id="Animal1" class="Hello.Animal">
  <constructor-arg index="0">
    <value>姗姗1</value>
  </constructor-arg>
  <constructor-arg index="2" value="哈哈" />
  <constructor-arg index="1" value="2" />
</bean>

<bean id="Animal2" class="Hello.Animal">
  <constructor-arg type="java.lang.String" index="0">
    <value>姗姗2</value>
  </constructor-arg>
  <constructor-arg index="1" value="1" />
  <constructor-arg type="java.lang.String" value="猪" />
</bean>
```

在指定注入顺序时，要注意参数的类型是否匹配。

3、 Autowire 自动装配

在应用中，我们常常使用<ref/>标签为受管 Bean 注入它依赖的对象。但是对于一个大型的系统，这个操作将会耗费我们大量的资源，我们不得不花费大量的时间和精力用于创建和维护系统中大量的<ref/>标签。实际上，这种方式也在另一种形式上增加了应用程序的复杂性，那么如何解决这个问题呢？Spring 为我们提供了一个自动装配的机制，尽管这种机制不是很完善，但是在应用中结合<ref/>标签还是可以大大的减少我们的劳动强度。前面提到过，在定义 Bean 时，<bean>标签有一个 autowire 属性，我们可以通过指定它来让容器为受管 JavaBean 自动注入依赖对象。

<bean/>的 autowire 属性有如下六个取值，他们的说明如下：

- 1、No：即不启用自动装配。默认为此值。
- 2、byName：通过属性的名字的方式查找受管 Bean 依赖的对象并为其注入。比如说类 Computer 有个 Printer 类型的成员 printer，指定其 autowire 属性为 byName 后，Spring IoC 容器会在配置文件中查找 id/name 属性为 printer 的 bean，然后使用 Setter 方法为其注入。
- 3、byType：通过属性的类型查找受管 Bean 依赖的对象并为其注入。比如类 Computer 有个成员 printer，类型为 Printer。那么，指定其 autowire 属性为 byType 后，Spring IoC 容器会查找 Class 属性为 Printer 的 bean，并使用 Setter 方法为其注入。
- 4、constructor：通 byType 一样，也是通过类型查找依赖对象。与 byType 的区别在于它不是使用 Setter 方法注入，而是使用构造子注入。
- 5、autodetect：在 byType 和 constructor 之间自动的选择注入方式。

6、default: 由上级标签<beans>的 default-autowire 属性确定。

注意: 在配置 bean 时, <bean>标签中 Autowire 属性的优先级比其上级标签高, 即是说, 如果在上级标签中定义 default-autowire 属性为 byName, 而在<bean>中定义为 byType 时, Spring IoC 容器会优先使用<bean>标签的配置。

下面通过一个例子来说明如何在应用中使用自动装配(工程代码见例程 3.2)。例程使用自动装配(协作者)机制自动为“电脑”安装“主机”和“显示器”。新建一个 java 工程, 为其添加上 Spring 开发能力后, 创建一个 ioc.test 包, 再分别创建电脑类(Computer)、主机类(Host)和显示器类(Display), 为电脑类添加 Host 类型的成员 host 和 Display 类型的成员 display, 再添加一个 run 方法, 让电脑可以“运行”一起来。代码如下:

```
package ioc.test;

/**
 * @author zhangyong
 */
public class Computer {

    private Host host;
    private Display display;
    //电脑运行方法
    public void run() {
        System.out.println("你好, 我是电脑, 正在运行!");
        System.out.print("    "+host.run()+"");
        System.out.println(display.run());
    }
    //Getter 和 Setter 方法, 省略
}
```

再给主机类添加 run 方法, 让主机也可以“运行”, 代码如下:

```
package ioc.test;

/**
 * @author zhangyong
 */
public class Host {

    public String run() {
        return "我是主机, 正在运行!";
    }

}
```

同上也给 Display 类添加 run 方法，如下：

```
package ioc.test;

/**
 * @author zhangyong
 */
public class Display {
    public String run() {
        return "我是显示器，正在运行！";
    }
}
```

再修改 Spring 的配置文件，让 IoC 容器为我们的“电脑”自动装配“主机”和“显示器了”。分别配置两个 bean，host 和 display。再配置一个名 computer1 的 bean，autowire 属性设为 byName，同理配置 computer2 和 computer3，autowire 属性分别设为 byType 和 default，最后设置<beans>标签的 default-autowire 属性为 autodetect。至此，配置工作已经完成，可以看到，我们并没有显式的给 computer bean 它注入依赖对象 host 和 display。配置代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-autowire="autodetect">
    <bean id="computer1" class="ioc.test.Computer"
autowire="byName"></bean>
    <bean id="computer2" class="ioc.test.Computer"
autowire="byType"></bean>
    <bean id="computer3" class="ioc.test.Computer"
autowire="default"></bean>

    <bean id="host" class="ioc.test.Host"></bean>
```

现在可以建立一个测试类来测试一下 Spring 是否已经为我们自动装配好了我们需要的 bean。代码如下：

```
package ioc.test;

//import省略
public class TestMain {

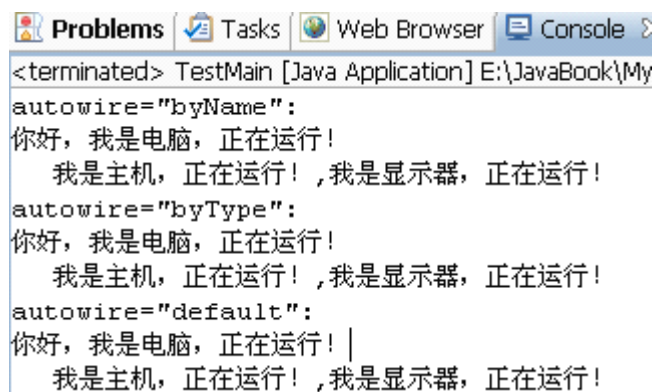
    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");

        //byName
        Computer computer1 = (Computer)ac.getBean("computer1");
        System.out.println("autowire=\"byName\"");
        computer1.run();

        //byType
        Computer computer2 = (Computer)ac.getBean("computer2");
        System.out.println("autowire=\"byType\"");
        computer2.run();

        //default
        Computer computer3 = (Computer)ac.getBean("computer3");
        System.out.println("autowire=\"default\"");
        computer3.run();
    }
}
```

运行该类，输出结果如下：



```
<terminated> TestMain [Java Application] E:\JavaBook\My
autowire="byName":
你好，我是电脑，正在运行！
    我是主机，正在运行！,我是显示器，正在运行！
autowire="byType":
你好，我是电脑，正在运行！
    我是主机，正在运行！,我是显示器，正在运行！
autowire="default":
你好，我是电脑，正在运行！ |
    我是主机，正在运行！,我是显示器，正在运行！
```

图 3.2 例程 3.2 运行结果

从运行结果可以看出，Spring IoC 容器已经为我们自动的装配好的“电脑”，从而配置文件简洁了许多。但是，自动装配并不是十全十美，我们不论是使用 byName 还是 byType 的方法，Spring 不一定就能很准确的为我们找到受管 Bean 依赖的对象。另外，如果使用自动装配，Spring 配置文件的可读性也大大降低，我们不能很容易的从配置文件中看出各个受

管 Bean 之间的依赖关系，这也在一定程度上降低了程序可维护性。因此在使用自动装配时，应当权衡利弊，合理的与 ref 的方法相结合，尽量在降低工作量的同时，保证应用的可维护度。

3.2.3 使用赋值标签

在上面的例子中我们使用<value/>标签为受管 Bean 的属性直接赋值。使用<value/>标签只是能够给基本的 java 数据库类型注入值，比如 int、String、float 等。如何给复杂的 JavaBean 属性注入值呢？下面介绍的几种标签可以做到这一点。

1、<ref/>标签

使用这个标签可以为受管 Bean 注入一个其他的受管 Bean 的实例的引用，当受管 Bean 的要引用其它类的实例时使用该标签。比如（本例子工程源代码见例程 3.3）：

```
<property name="master" ref="master" />
```

2、<list/>标签

这个标签可以为受管 Bean 注入 List 类型或者数组类型的值。当要给某个受管 Bean 的 List 或者数组类型的属性注入值时，使用该标签。用法：

```
<property name="students">
  <list>
    <ref bean="student1" />
    <ref bean="student2" />
  </list>
</property>
```

3、<set/>标签

这个标签可以为受管 Bean 注入 Set 类型的值，和 List 类似，用法如下：

```
<property name="teacher">
  <set>
    <ref bean="teacher1" />
    <ref bean="teacher2" />
  </set>
</property>
```

4、<map/>标签

Map 是 java 中一个重要的接口，有 HashMap、Properties 等很多实现类。由于它以键、值对的形式存储数据，所以要用<entry>标签来为其 key 和 value 赋值。用法如下：

```
<property name="book">
  <map>
    <entry key="name">
      <value>Spring学习</value>
    </entry>
    <entry key="price" value="50元" />
    <entry key="author" value="残梦追月" />
  </map>
</property>
```

5、<props/>标签

在 java 中 Properties 继承实现了 Map 接口，因此它也是用键值对的形式存放数据，与 Map 不同的是它的键(Key)、值(value)是 java.lang.String 类型，无法为其赋 Object 类型的值。其用法如下：

```
<property name="course">
  <props>
    <prop key="teacher">李老师</prop>
    <prop key="score">85</prop>
    <prop key="classroom">10A-302</prop>
  </props>
</property>
```

6、<null/>标签

这个标签可以为受管 Bean 的属性注入空值。语法为：

```
<null/>
```

下面通过一个例子来具体应用以上介绍的各种标签。在例程中，使用上面介绍的标签为“学校”类（School）注入校名(name)，校长(master)，学生列表(students)和教师列表(teacher)。

新建一个工程，为其添加 Spring 开发能力。然后分别创建二个类，一个 School 类，一个 Peoples 类，部分代码如下：

School 类：

```
package test

//导入包的部分省略
public class School {

    private String name; //学校的名字
    private People master; //校长
    private List<People> students; //学校里的所有学生
    private Set<People> teacher; //学校里的所有老师

    //学校的自我介绍
    public void say() {

        System.out.println("嗨！我是学校。");
        System.out.println("我的名字是：" + this.name);
        System.out.println("我的校长是：" + this.master.getName());
        System.out.println("在我校的学生有：" + this.students.size() + "个, 他们是：\n");

        //调用每个学生的say方法。
        Iterator<People> is = students.iterator();
        while (is.hasNext()) {
            is.next().say();
        }

        System.out.println("在我校的老师有：" + this.teacher.size() + "个, 他们是：");

        //调用每个老师的say方法。
        Iterator<People> it = teacher.iterator();
        while (it.hasNext()) {
            System.out.print(it.next().getName() + "、");
        }
    }

    //所有Getter和Setter方法，省略
}
```

People 类:

```
package test

//导入包的部分省略
public class People {

    private String name; //人的名字
    private Map<String, Object> course; //学生的课程
    private Properties book; //学生的书

    public void say() {
        System.out.println(" 我的名字是:" + this.name);

        //输出课程
        System.out.println(" 我有一门课程, 它是: ");
        Iterator im = course.entrySet().iterator();
        while(im.hasNext()) {

            Map.Entry entry = (Map.Entry)im.next();
            System.out.print("    " + entry.getKey() + ":" + entry.getValue());

        }

        //输出书
        System.out.println("\n 我有一本书, 它是: ");
        Iterator ip = book.entrySet().iterator();
        while(ip.hasNext()) {
            Map.Entry entry = (Map.Entry)ip.next();
            System.out.print("    " + entry.getKey() + ":" + entry.getValue());
        }
        System.out.println("\n");
    }

    //所有Getter和Setter方法, 省略
}
```

3 修改生成的 Spring 配置文件 applicationContext.xml, 分别为两个类的各成员注入值。代码如下:

```

<beans .....省略部分代码，详见例程3.2 >
  <bean name="school" class="test.School">
    <property name="name" value="四川农业大学" />
    <property name="master" ref="master" />
    <property name="students">
      <list><ref bean="student1"/><ref bean="student2"/></list>
    </property>
    <property name="teacher">
      <set><ref bean="teacher1"/><ref bean="teacher2"/></set>
    </property>
  </bean>
  <bean id="student1" class="test.People">
    <property name="name" value="张三" />
    <property name="book">
      <map>
        <entry key="name"><value>Spring学习</value></entry>
        <entry key="price" value="50元" />
        <entry key="author" value="残梦追月" />
      </map>
    </property>
    <property name="course">
      <props>
        <prop key="teacher">李老师</prop>
        <prop key="score">85</prop>
        <prop key="classroom">10A-302</prop>
      </props>
    </property>
  </bean>
  <bean id="teacher1" class="test.People">
    <property name="name" value="潘Sir" />
  </bean>
  <bean id="teacher2" class="test.People">
    <property name="name" value="浦Sir" />
  </bean>
  <bean id="master" class="test.People">
    <property name="name" value="文校长" />
  </bean>
  .....省略部分代码，详见例程3.2
</beans>

```

为学校分别注入两个“学生”和两个“老师”。

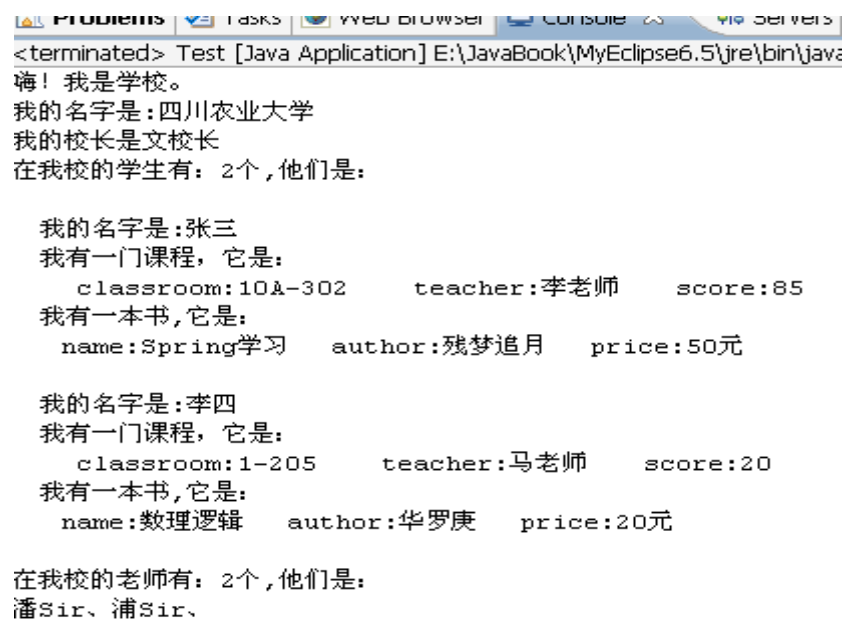
3 至此，配置已经完成，我们可以创建一个 Test 类来运行我们的程序：


```
package test;

//导入包的部分省略
/**
 * @author zhangyong
 */
public class Test {

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");
        School school = (School) ac.getBean("school");
        school.say();
    }
}
```

单击运行按钮运行程序，结果如下：



```
<terminated> Test [Java Application] E:\JavaBook\MyEclipse6.5\jre\bin\jav
嗨！我是学校。
我的名字是：四川农业大学
我的校长是文校长
在我校的学生有：2个，他们是：

    我的名字是：张三
    我有一门课程，它是：
        classroom:10A-302    teacher:李老师    score:85
    我有一本书，它是：
        name:Spring学习    author:残梦追月    price:50元

    我的名字是：李四
    我有一门课程，它是：
        classroom:1-205    teacher:马老师    score:20
    我有一本书，它是：
        name:数理逻辑    author:华罗庚    price:20元

在我校的老师有：2个，他们是：
潘Sir、浦Sir、
```

图 3.3 例程 3.3 运行结果

3.3 受管 Bean 作用范围

在 jsp 中，使用 jsp 标签<jsp:useBean>来引用 JavaBean，可以通过其 scope 属性的值来确定该标签引用 JavaBean 的作用范围。在 Spring IOC 容器中，由它管理的 Java Bean 也具有作用范围。

在 Spring1.x 版本中，<bean/>标签有一个属性 singleton，取值为布尔类型。如果将其设置为 true，那么容器之中只存在一个该 bean 的共享实例，当其他的任何 Bean 依赖该 bean

时,只要请求的 id 与该 bean 的 id 相同,容器就将该 bean 的一个引用注入到请求的 bean 中。换言之, Spring IoC 容器只是创建该 bean 的一个唯一实例,并把它存贮在容器的单例缓存中。这样的受管 bean 称之为“单例 bean”。

如果设置为 false,那么每当其他 bean 请求此 bean 时,容器则会重新实例化一个该 bean 对象,为其注入。

需要注意的是:

1、 在下面的例子中,如果把 computer1 和 computer2 两个受管 bean 都设置成单例 bean, Spring IoC 容器则分别实例化两个 bean,把它们作为两个不同的 bean 对待,尽管他们的类相同。

```
<bean id="computer1" class="ioc.test.Computer" scope="singleton"/></bean>

<bean id="computer2" class="ioc.test.Computer" scope="singleton"/></bean>
```

2、 一般来说,对于无状态的 bean 使用单例模式,对于有状态的 bean 使用 prototype 模式。

3、 Spring IoC 容器不会维护 prototype 类型的 bean 的整个生命周期,容器在实例化、配置、注入之后就把它扔给调用者,然后就不管了。

4、 如果一个单例 bean computer 引用了一个 prototype 类型的 bean host,由于单例 bean 只初始化一次,所以并不能保证每次调用 computer 时 host 都是最新的。解决办法是使用 lookup 方法注入。

到了 Spring2.0 时代, scope 属性代替了原来的 singleton 属性, scope 提供了更多的选项,从而可以更加灵活的配置 bean 的作用范围。Spring2.0 中, scope 属性有如下可能的取值,说明如下:

1、 singleton, 即单例 bean, 和 1.x 中 singleton="true" 相同。

2、 prototype, 同 Spring1.x 中的 singleton="false"。

3、 request, 这种 bean 在 web 的 request 范围内有效,即每次有 http 请求到来时都会产生一个实例。只用于 web 应用中。

4、 session, 这种 bean 在 web 的 session 范围内有效。只用于 web 程序中

5、 global session, 这种 bean 在 web 的全局 session 范围内有效。只用于 web portlet 框架中。

下面通过一个例子演示单例bean和prototype bean。在例子中，我们创建一个DateTime类，在其构造方法中获取当前的系统时间，并存储于date成员变量之中，以记录下该类实例化的时间。然后利用该类定义两个bean，一个为单例bean，一个为prototype bean。利用线程，两次调用getBean方法从IoC容器中获取这两个bean的实例的引用，并将存储于其中时间打印出来。为了便于测试，在两次调用getBean方法之间让线程暂停小段时间。这样，如果是单例bean，由于在容器中只是实例化一次，构造方法也只调用一次，那么两次显示的时间应当相同；prototype类型的Bean显示的时间则不一样。这样通过其返回时间是否一致就可以知道受管bean是否重新被实例化过。

- 1、新建一个java工程，为添加Spring开发能力后，建一个包ioc.test。
- 2、创建一个类DateTime，添加一Date类型的成员，并添加Getter方法。修改其构造方法，让其在构造方法中获取当前系统时间，并存储与date成员中。代码如下：

```
package ioc.test;

import java.util.Calendar;
import java.util.Date;

/**
 * @author zhangyong
 */
public class DateTime {
    private Date date;

    DateTime() {
        this.date = Calendar.getInstance().getTime();
    }

    public Date getDate() {
        return date;
    }
}
```

- 3、新建一Thread类的子类MyThread，重载run方法，在run方法中两次调用getBean方法从容器获取bean实例，然后分别将存储与bean实例中的时间打印出来。代码如下：

```

package ioc.test;
import org.springframework.context.ApplicationContext;
public class MyThread extends Thread {

    private ApplicationContext ac;
    private DateTime dt;
    private String bean;

    MyThread(ApplicationContext ac,String bean){
        this.ac=ac;
        this.bean=bean;
    }

    @Override
    public void run() {
        //第一次从容器取得bean
        dt = (DateTime) ac.getBean(bean);
        System.out.println("Thread Id:" + this.getId()+" 时间:
"+dt.getDate());

        //线程暂停5秒
        try {
            sleep(1000 * 5);
        } catch (InterruptedException e) {
        }

        //第二次从容器取得bean
        dt = (DateTime) ac.getBean(bean);
        System.out.println("Thread Id:" + this.getId()+" 时间:
"+dt.getDate());
    }
}

```

通过这两个参数
传入容器实例的引用
和需要测试的 Bean
的名称。

4、编写Spring配置文件，配置两个bean，一个singleton，一个prototype，如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans .....>
    <bean id="singletonDateTime" class="ioc.test.DateTime"
scope="singleton"></bean>
    <bean id="prototypeDateTime" class="ioc.test.DateTime"
scope="prototype"></bean>
</beans>

```

定义两个
Bean，分别
为单例 Bean
和 prototype
Bean

5、编写一测试类TestMain，代码如下：

```
package ioc.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

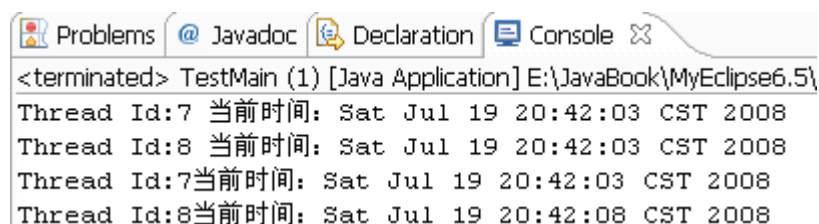
public class TestMain {

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");

        //测试单例bean
        MyThread mt1 = new MyThread(ac, "singletonDateTime");
        mt1.start();

        //测试prototype bean
        MyThread mt2 = new MyThread(ac, "prototypeDateTime");
        mt2.start();
    }
}
```

6、运行测试类，结果如下：



```
<terminated> TestMain (1) [Java Application] E:\JavaBook\MyEclipse6.5\
Thread Id:7 当前时间: Sat Jul 19 20:42:03 CST 2008
Thread Id:8 当前时间: Sat Jul 19 20:42:03 CST 2008
Thread Id:7当前时间: Sat Jul 19 20:42:03 CST 2008
Thread Id:8当前时间: Sat Jul 19 20:42:08 CST 2008
```

图 3.4 例程 3.4 运行结果

3.4 受管 Bean 的生命周期

3.4.1 受管 Bean 在容器中的生命周期

在 Spring 中，IoC 容器负责管理其中所有受管 Bean 的生命周期。从每个 Bean 的实例化、初始化、注入到销毁，都在 Spring IoC 容器的控制之下。其中某一个受管 Bean 的生命周期如下图所示：

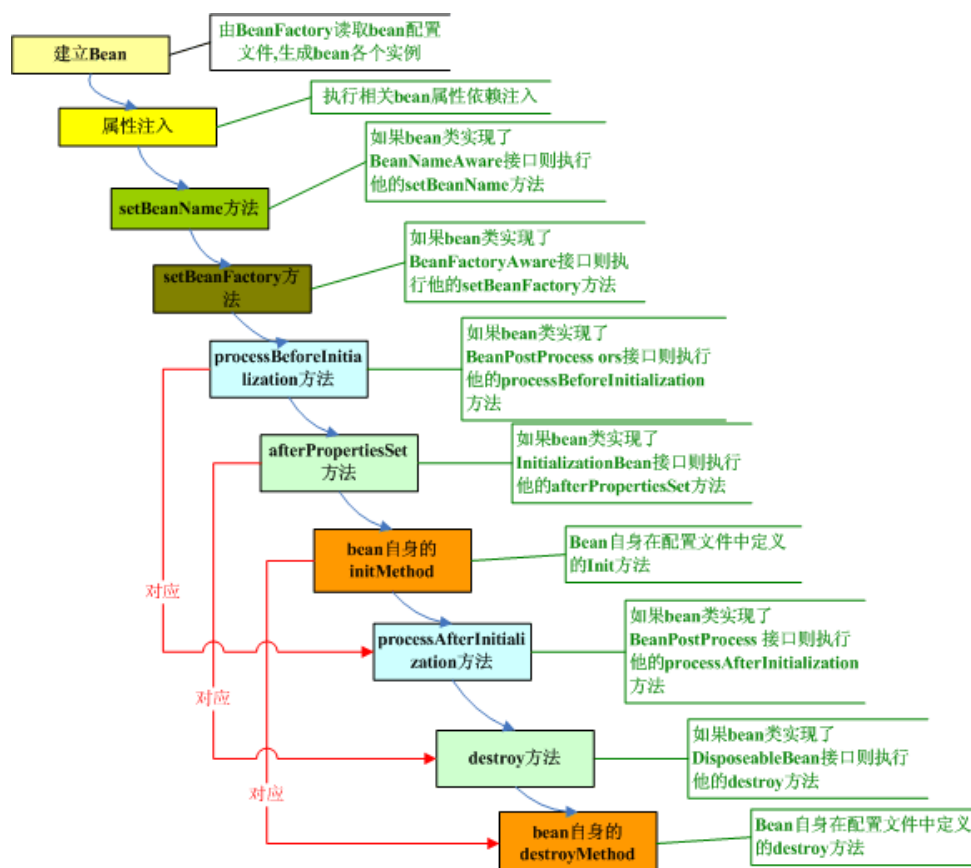


图 3.5 受管 Bean 的生命周期（来源于网络）

上图展示了一个受管 Bean 在 Spring IoC 容器中的生命过程。具体过程如下：

- 3 首先，Spring IoC 容器加载配置文件，分析其中配置的 Bean，创建 Bean 的实例。
- 3 分析各 Bean 之间的依赖关系，执行相关 Bean 的依赖注入。
- 3 检查受管 Bean 是否实现了 BeanNameAware 接口。如果是，则调用 setBeanName () 方法，将该受管 Bean 的 name 注入到中。
- 3 受管 Bean 是否实现 BeanFactoryAware 接口，是则调用 setBeanFactory () 方法，将 IoC 容器 BeanFactory 的一个引用注入到该受管 Bean 中。
- 3 在容器中是否注册有 BeanPostProcessor 接口的实现类，是则调用该注册类的 postProcessBeforeInitialization () 方法。
- 3 是否实现 InitializingBean 接口，是则调用 afterPropertiesSet () 方法。
- 3 是否指定 init-method 方法，是则调用指定的方法。
- 3 在容器中是否注册有 BeanPostProcessor 接口的实现类，是则调用该注册类的 postProcessAfterInitialization () 方法。
- 3 是否实现 DisposableBean 接口，调用 destroy () 方法。
- 3 是否指定 destroy-Method 方法，是则调用指定的方法。

3.4.2 受管 Bean 的预处理和后处理

有时候，我们希望在 Spring IoC 容器初始化受管 Bean 之前、属性设置之后对该 Bean 先做一些预处理，或者在容器销毁受管 Bean 之前自己释放资源。那么该如何实现呢？正如下图所示，Spring IoC 为我们提供了多种方法来实现受管 Bean 的预处理和后处理。

1、使用 BeanPostProcessor 接口

在 Spring 中定义了 BeanPostProcessors 接口，代码如下：

```
package org.springframework.beans.factory.config;

import org.springframework.beans.BeansException;

public interface BeanPostProcessor {

    Object postProcessBeforeInitialization(Object bean, String
    BeanName) throws BeansException;

    Object postProcessAfterInitialization(Object bean, String
    BeanName) throws BeansException;
}
```

如果这个接口的某个实现类被注册到某个容器，那么该容器的每个受管 Bean 在调用初始化方法之前，都会获得该接口实现类的一个回调。容器调用接口定义的方法时会将该受管 Bean 实例的引用和名字通过参数传入方法，经过处理后通过方法的返回值返回给容器。根据这个原理，我们就可以很轻松的自定义受管 Bean。

上面提到过，要使用 BeanPostProcessor 回调，就必须先在容器中注册实现该接口的类，那么如何注册呢？BeanFactory 和 ApplicationContext 容器的注册方式不大一样：若使用 BeanFactory，则必须要显示的调用其 addBeanPostProcessor() 方法进行注册，参数为 BeanPostProcessor 实现类实例的一个引用；如果是使用 ApplicationContext，那么容器会在配置文件在中自动寻找实现了 BeanPostProcessor 接口的 Bean，然后自动注册，我们要做的只是配置一个 BeanPostProcessor 实现类的 Bean 就可以了。

注意，假如我们使用了多个的 BeanPostProcessor 的实现类，那么如何确定处理顺序呢？其实只要实现 Ordered 接口，设置 order 属性就可以很轻松的确定不同实现类的处理顺序了。

例程 3.5 展示了如何使用 BeanPostProcessor 回调接口。创建 Java 工程，为其添加 Spring 开发能力后，创建 ioc.test 包。再创建一个名字为 Animal 的 Bean，添加 name、age 成员和 speak() 方法。代码如下：

```
package ioc.test;

public class Animal {
    private String name;
    private int age;

    public String speak(){
        return "我的名字是: "+this.name+", 年龄是: "+this.age+"!\n";
    }

    //Getter 和 Setter 省略
}
```

创建一个 BeanPost 类，实现 BeanPostProcessor 接口。在其 postProcessAfterInitialization() 方法中修改通过参数传入的受管 Bean，然后将其返回给容器。由于它处理容器中的每一个 Bean，因此在修改前，应判断参数传入的 Bean 是否为我们处理的 Bean。我们可以通过传入 Bean 的类型判定，也可以通过传入 Bean 的名字判定。代码如下：

```
package ioc.test;

//Import 省略

public class BeanPost implements BeanPostProcessor {

    public Object postProcessAfterInitialization(Object bean, String
beanName) throws BeansException {
        System.out.println("BeanPostProcessor.post"+
            "ProcessAfterInitialization 正在预处理!");
        if ((bean instanceof Animal)) {
            Animal animal = (Animal) bean;
            animal.setAge(50);
            animal.setName("猴子");
            return bean;
        }
        return bean;
    }

    public Object postProcessBeforeInitialization(Object bean, String
beanName) throws BeansException {
        System.out.println("BeanPostProcessor.post"+
            "ProcessBeforeInitialization 正在预处理!");
        return bean;
    }
}
```

定义好 Animal Bean，为其属性随便注入一个值，再定义一个 Bean，class 为 BeanPost 类。

代码如下：

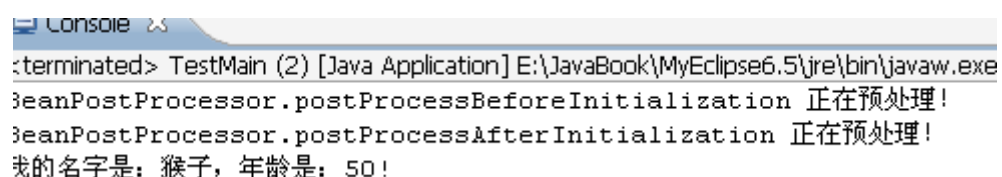
```
<?xml version="1.0" encoding="UTF-8"?>
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
  <bean id="animal" class="ioc.test.Animal">
    <property name="age" value="5"></property>
    <property name="name" value="猪"></property>
  </bean>

  <bean id="beanPost" class="ioc.test.BeanPost"></bean>
</beans>
```

最后创建 TestMain 类输出结果，代码如下：

```
package ioc.test;
//Import 省略
public class TestMain {
    public static void main(String[] args) {
        ApplicationContext ac = new
        ClassPathXmlApplicationContext("applicationContext.xml");
        Animal animal = (Animal)ac.getBean("animal");
        System.out.println(animal.speak());
    }
}
```

运行后输出结果如下：



```
terminated> TestMain (2) [Java Application] E:\JavaBook\MyEclipse6.5\jre\bin\javaw.exe
BeanPostProcessor.postProcessBeforeInitialization 正在预处理!
BeanPostProcessor.postProcessAfterInitialization 正在预处理!
我的名字是：猴子，年龄是：50!
```

图 3.6 例程 3.5 运行结果

可以看到，输出结果并不是我们在配置文件中注入的值，这说明 BeanPost 已经按照我们的意愿成功的修改了目标 Bean。

2、使用初始化回调接口 InitializingBean

要对某个受管 Bean 进行预处理里时，可以实现 Spring 定义的初始化回调接口 InitializingBean，它定义了一个方法如下：

```
void afterPropertiesSet() throws Exception
```

开发者可以在受管 Bean 中实现该接口，再在此方法中对受管 Bean 进行预处理。

注意：应该尽量避免使用这种方法，因为这种方法会将代码与 Spring 耦合起来。推荐使用下面的第四种方法。

3、使用析构回调接口 DisposableBean

同上面一样，要对受管 Bean 进行后处理，该 Bean 可以实现析构回调接口 DisposableBean，它定义的方法如下：

```
void destroy() throws Exception
```

为了降低与 Spring 的耦合度，这种方法也不推荐。

为了让读者熟悉这种方式，下面的例子（例程 3.6）简要的展示如何使用这两个接口。

新建 Java 工程，名字为 IoC_Test3.6，为其添加 Spring 开发能力后，新建一 ioc.test 包，添加一个 Animal 类，该类实现 InitializingBean 接口和 DisposableBean 接口，再为其添加 name 和 age 成员，Getter、Setter 方法和 speak 方法。完成后代码如下：

```
package ioc.test;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class Animal implements InitializingBean, DisposableBean {

    private String name;
    private int age;

    public String speak() {
        return "我的名字: "+this.name+"我的年龄: "+this.age;
    }

    public void afterPropertiesSet() throws Exception {
        System.out.println("初始化接口afterPropertiesSet()方法正在运行!");
    }

    public void destroy() throws Exception {
        System.out.println("析构接口destroy()方法正在运行!");
    }

    //Getter 和 Setter 省略
}
```

在配置文件中配置受管 Bean，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    <bean id="animal" class="ioc.test.Animal">
        <property name="age" value="5"></property>
        <property name="name" value="猪"></property>
    </bean>
</beans>
```

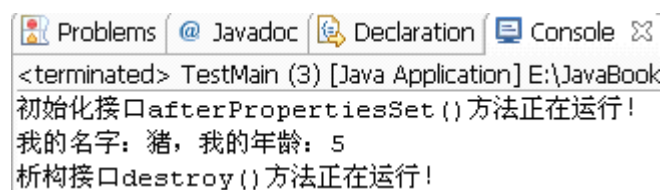
新建含有主方法的 TestMain 类，在主方法中添加测试代码，如下：

```
package ioc.test;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class TestMain {

    public static void main(String[] args) {
        AbstractApplicationContext ac = new
ClassPathXmlApplicationContext("applicationContext.xml");
        //注册容器关闭钩子，才能关掉容器，调用析构方法
        ac.registerShutdownHook();
        Animal animal = (Animal)ac.getBean("animal");
        System.out.println(animal.speak());
    }
}
```

除了注册关闭钩子外，还可以显式的调用 close 方法，但是推荐采用注册钩子的方法关闭容器。

运行工程，结果如下：



```
Problems @ Javadoc Declaration Console
<terminated> TestMain (3) [Java Application] E:\JavaBook
初始化接口afterPropertiesSet()方法正在运行!
我的名字: 猪, 我的年龄: 5
析构接口destroy()方法正在运行!
```

图 3.7 例程 3.6 运行结果

可以看到，Spring IoC 容器自动的调用了两个接口的相关方法对 bean 进行了预处理和后处理。

注意：由于后处理是在 Bean 被销毁之前调用，所有要在 MyEclipse 中看到后处理方法的输出，必须先注册容器的关闭钩子，在主方法退出后关掉容器，这样其管理的 Bean 就会被销毁。

4、使用<bean>标签的 init-method 和 destroy-method 属性

前面提到过，<bean> 标签中，有 init-method 和 destroy-method 属性，通过设置这两个属性的值，可以很方便的指定该受管 Bean 的缺省的初始化方法和析构方法。

要给应用中每个 Bean 都指定 init-method 和 destroy-method 属性，那将是一个麻烦的工作，要简化配置，可以通过<beans>标签的 default-init-method 和 default-destroy-method 属性来为其范围内的所有受管 Bean 制定相同的初始化方法和析构方法。

下面的范例展示如何使用<bean>标签的init-method和destroy-method属性(工程代码见例程3.7)。在范例中通过这两个属性指定受管Bean Animal的初始化和析构方法。

创建java工程，添加Spring开发能力，创建ioc.test包。创建Animal类，为其添加name、age成员、Getter和Setter方法、speak方法后，再添加一个初始化方法和一个析构方法，名字可以任意，这里为start和end。代码如下：

```
package ioc.test;

public class Animal{

    private String name;
    private int age;

    public String speak(){
        return "我的名字: "+this.name+", 我的年龄: "+this.age;
    }

    public void start() throws Exception {
        System.out.println("初始化方法start() 正在运行! ");
    }

    public void end() throws Exception {
        System.out.println("析构方法end() 正在运行! ");
    }

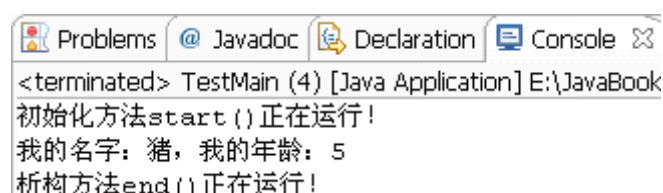
    //Getter 和 Setter 省略
}
```

在配置文件中配置好bean,并为其指定响应的预处理方法和析构方法：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
  <bean id="animal" class="ioc.test.Animal" init-method="start" destroy-
method="end">
    <property name="age" value="5"></property>
    <property name="name" value="猪"></property>
  </bean>
</beans>
```

指定初始化和析构方法

创建含有主方法的测试类，代码和例程3.6完全一样，这里就不赘述了。运行结果如下：



```
<terminated> TestMain (4) [Java Application] E:\JavaBook
初始化方法start () 正在运行!
我的名字: 猪, 我的年龄: 5
析构方法end () 正在运行!
```

图 3.8 例程 3.7 运行结果

需要注意的是：要看到析构方法的输出，也必须注册关闭钩子。

有趣的试验：可以将以上的四种方式应用于同一个受管Bean，看看四种方式的执行顺序如何？

3.5 Spring 中的 Ioc 容器

Spring 提供了两种 IoC 容器， BeanFactory 和 ApplicationContext。Spring IoC 容器为我们提供 IoC 功能的实现，它负责读取配置、装配 Bean、管理 Bean 的生命周期和维护各 Bean 之间的关系，同时为用户获取、管理的 Bean 提供接口。BeanFactory 是 Spring IoC 容器的基本实现，ApplicationContext 继承于 BeanFactory，提供了比 BeanFactory 更加丰富的功能。

3.5.1 BeanFactory 容器

BeanFactory 是整个 Spring IoC 容器的核心。它是一个接口，定义了 IoC 容器的基本功能。它的实现类实现了从配置文件中读取配置、装配 Bean、管理 Bean 的生命周期、Bean 之间的关系维护等 IoC 容器的基本功能。这些实现类通过 Java 的反射机制和工厂模式来实现装配、管理 Bean 的过程，从而使被管对象（受管 Bean）无需知道 Spring 的存在，无需实现任何 Spring 接口、使用任何 Spring 提供的 API，这就降低了 Spring 与受管 Bean 之间的耦合度。

BeanFactory 容器占用资源相对 ApplicationContext 较小，所以一般在对资源要求比较严格的环境使用，比如 applet。

正如上所说，BeanFactory 是一个接口，被放在包 org.springframework.beans.factory 中。Spring 提供了它的多种实现类，其中 XmlBeanFactory 是最简单、最常用的实现类之一。使用方法如下：

```
//New一个资源，这个资源可以是ClassPathResource,也可以是FileSystemResource
ClassPathResource cr = new ClassPathResource("applicationContext.xml");
BeanFactory bf = new XmlBeanFactory(cr);

//使用BeanFactory
bf.getBean("beanName");
.....
```

3.5.2 ApplicationContext 容器

ApplicationContext 容器继承于 BeanFactory 容器，除了提供 BeanFactory 的所有功能外，还提供了比 BeanFactory 更加丰富的功能。因此 Spring 的团队推荐使用 ApplicationContext 容器。在 Spring 官方的文档中，提供了他们推荐使用 ApplicationContext 而不是 BeanFactory 的原因，摘录如下：

表 3.2 BeanFactory 和 ApplicationContext 特性比较（来自 Spring 官方帮助文档）

特性	BeanFactory	ApplicationContext
Bean 的实例化、装配	Yes	Yes
自动注册 BeanPostProcessor	No	Yes
自动注册 BeanFactoryPostProcessor	No	Yes
便捷的 MessageSource 访问	No	Yes
ApplicationEvent 发送	No	Yes

1、ApplicationContext 容器的实现类

为了适用于不同的使用场合，Spring 为我们提供了很多 **ApplicationContext** 的实现类，部分说明如下：

AbstractApplicationContext: 是 ApplicationContext 的抽象实现类，不能被直接实例化。

AbstractXmlApplicationContext: 继承于 AbstractApplicationContext，提供了对 XML 配置文件的支持，就是说它可以从 XML 配置文件加载配置。

ClassPathXmlApplicationContext: 继承于 AbstractXmlApplicationContext。它从加载的 CLASSPATH 路径搜索并装载 Xml 配置文件、创建 ApplicationContext 的实例。范例代码如下:

```
ApplicationContext ac = new
    ClassPathXmlApplicationContext("applicationContext.xml");
//使用ApplicationContext
ac.getBean("beanName");
.....
```

FileSystemXmlApplicationContext: 继承于 AbstractXmlApplicationContext, 与 ClassPathXmlApplicationContext 类似, 不同的是它是从参数指定的路径来装载配置文件。参数可以是文件的绝对路径, 也可以是相对路径。范例代码如下:

```
ApplicationContext ac = new
    FileSystemXmlApplicationContext("applicationContext.xml");
//使用ApplicationContext
ac.getBean("beanName");
.....
```

XmlWebApplicationContext: 用于 Web 应用中的 ApplicationContext 的实现。

2、Spring 事件

在 Windows 编程中, 我们常常需要处理各类事件, 比如鼠标单击事件、双击事件。在 Spring 中, ApplicationContext 也有发布和监听事件的能力。我们知道, 在 windows 开发中, 如果要响应某个事件, 我们只需要编写相应 windows 消息的响应函数就可以了。比如鼠标单击事件, 相应的消息就是 WM_LBUTTONDOWN。在 Spring 中也是一样, Spring 中 ApplicationEvent 类及其子类就相当于 Windows 中的消息, 事件监听器 ApplicationListener 的实现类就相当于 Windows 编程中的消息处理函数。

要使用 Spring 事件处理机制, 就必须先定义一个事件 (定义一个 Windows 消息), 发布出去后, 再定义一个事件监听器 (编写消息处理函数), 发布到容器中。这样, 当该事件发生后, 我们就可以在事件监听器的 onApplicationEvent() 方法中处理我们的事件了。

ApplicationEvent 类是抽象类, 不能被实例化, 故 Spring 中的事件类都是其子类。我们要自定义一个事件就必须扩展该类, 该类的代码如下:

```
public abstract class ApplicationEvent extends EventObject {  
    private final long timestamp;  
    public ApplicationEvent(Object source) {  
        super(source);  
        this.timestamp = System.currentTimeMillis();  
    }  
    public long getTimestamp() {  
        return timestamp;  
    }  
}
```

如上所示，抽象类 `ApplicationEvent` 中 `timestamp` 字段存储事件发生的时间，在该事件发生时（事件类被实例化），`ApplicationEvent` 会自动的把当前的时间存储到该字段中，在应用中可以通过 `getTimestamp()` 方法获取该时间。如果应用要传递一个对象给监听器，那么可以把要传递的对象作为构造函数的参数 `source` 传递给 `ApplicationEvent` 类。

为了监听事件的发生并处理事件，Spring 提供了一个接口 `ApplicationListener`，代码如下：

```
package org.springframework.context;  
import java.util.EventListener;  
public interface ApplicationListener extends EventListener {  
    void onApplicationEvent(ApplicationEvent event);  
}
```

当有 `ApplicationEvent` 类的子类被发布到 `ApplicationContext` 时，如果已经在上下文中部署有实现了 `ApplicationListener` 接口的 Bean，那么这个 Bean 会得到一个通知，方法 `onApplicationEvent()` 会被调用，并将该事件的实例作为参数传入。与 Windows 消息处理函数不同的是，这个方法会响应所有的事件。因此在编写 `onApplicationEvent()` 方法时，应当要先判断当前的事件是否是我们要响应的事件。

使用 Spring 内部定义的事件：

在 Spring 中已经定义了五个标准事件，分别介绍如下：

- 1、 `ContextRefreshedEvent`：当 `ApplicationContext` 初始化或者刷新时触发该事件。
- 2、 `ContextClosedEvent`：当 `ApplicationContext` 被关闭时触发该事件。容器被关闭时，其管理的所有单例 Bean 都被销毁。
- 3、 `RequestHandleEvent`：在 Web 应用中，当一个 http 请求（request）结束触发该事件。
- 4、 `ContextStartedEvent`：Spring 2.5 新增的事件，当容器调用 `ConfigurableApplicationContext`

的 Start()方法开始/重新开始容器时触发该事件。

- 5、 ContestStopedEvent: Spring2.5 新增的事件，当容器调用 ConfigurableApplicationContext 的 Stop()方法停止容器时触发该事件。

下面通过一个例子展示如何处理 Spring 内定的事件（例程 3.8）。创建一个 Java 工程，添加 Spring 开发能力后，新建 ioc.test 包。在包中新建 ApplicationEventListener 类，实现 ApplicationListener 接口，在 onApplicationEvent()方法中添加事件处理代码，如下：

```
package ioc.test;

//Import省略
public class ApplicationEventListener implements ApplicationListener {

    public void onApplicationEvent(ApplicationEvent event) {
        //如果是容器刷新事件
        if(event instanceof ContextClosedEvent ){
            System.out.println(event.getClass().getSimpleName()+" 事件已发生!");
        }else if(event instanceof ContextRefreshedEvent ){//如果是容器关闭事件
            System.out.println(event.getClass().getSimpleName()+" 事件已发生!");
        }else if(event instanceof ContextStartedEvent ){
            System.out.println(event.getClass().getSimpleName()+" 事件已发生!");
        }else if(event instanceof ContextStoppedEvent){
            System.out.println(event.getClass().getSimpleName()+" 事件已发生!");
        }else{
            System.out.println("有其它事件发生:"+event.getClass().getName());
        }
    }
}
```

在 Spring 配置文件中定义一个 Bean，类为 ApplicationEventListener，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ..... >
<bean id="ApplicationEventListener"
class="ioc.test.ApplicationEventListener" />
</beans>
```

添加含有主方法的 TesMain 类，在主方法中，调用容器的相应方法，触发 Spring 内

定事件，代码如下：

```
package ioc.test;

//import省略

public class TesMain {

    public static void main(String[] args) {


        AbstractApplicationContext ac = new
            ClassPathXmlApplicationContext("applicationContext.xml");

        // ac.refresh();//触发ContextRefreshedEvent事件
        ac.start(); //触发ContextStartedEvent事件
        ac.stop(); //触发ContextStoppedEvent事件
        ac.close(); //关闭容器，触发ContextClosedEvent事件

    }

}
```

运行主类，控制台输出如下：



```
<terminated> TesMain [Java Application] E:\JavaBook\MyE
ContextRefreshedEvent 事件已发生!
ContextStartedEvent 事件已发生!
ContextStoppedEvent 事件已发生!
ContextClosedEvent 事件已发生!
```

图 3.9 例程 3.8 运行结果

从例子中可以知道，要注册事件监听器，我们只需要把它配置成一个 Bean 即可，ApplicationContext 容器会自动将其注册。

开发者发布监听自定义事件

如果仅仅使用 Spring 的内定事件，那显然是远远不够的，幸好，Spring 为我们提供了自定义发布事件的能力。下面通过例程 3.9 来展示如何发布并监听自定义的事件。

在工程中，我们定义一个 Animal 类，为受管 Bean，它具有一个 Speak 方法，我们要做的就是监视该方法，当用户调用该方法时触发 AnimalSpeakEvent 事件。具体操作如下：

新建名字为 IoC_Test3.9 的 java 工程，添加 Spring 开发能力后，建立 ioc.test 包。新建一个事件类 AnimalSpeakEvent，它继承自 ApplicationEvent，重载其默认的构造方法。再添加一个 String 成员 animalName 和它的 Getter 方法。添加一个构造方法，此方法有两个参数，一个为 source，一个为 animalName。代码如下：

```
package ioc.test;
import org.springframework.context.ApplicationEvent;
public class AnimalSpeakEvent extends ApplicationEvent {
    private static final long serialVersionUID = 1L;
    private String animalName;
    public AnimalSpeakEvent(Object source) {
        super(source);
    }
    public AnimalSpeakEvent(Object source,String animalName) {
        super(source);
        this.animalName = animalName;
    }
    public String getAnimalName() {
        return animalName;
    }
}
```

创建好该事件类后,就应该把它再合适的时候发布出去。既然它是一个“动物讲话事件”,那么就应该再动物“讲话”的时候发布,如何发布呢?我们知道要发布一个事件,就必须调用 `ApplicationContext` 的 `publishEvent` 方法。要在 `Animal` 类中获得 `ApplicationContext` 的实例,就要实现 `ApplicationContextAware` 接口(关于该接口,3.6.3 会详细介绍),代码如下:

```
package ioc.test;
//import省略
public class Animal implements ApplicationContextAware {
    private ApplicationContext ac;
    private String name;
    private int age;

    public String speak(){
        ac.publishEvent(new AnimalSpeakEvent(this,this.name));
        return " 我的名字是:"+this.name+",我的年龄是:"+this.age;
    }

    public void setApplicationContext(ApplicationContext arg0)
        throws BeansException {
        this.ac = arg0;
    }
    //Getet和Setter省略
}
```

到目前为之,我们已经在 `Animal` 类中把事件发布出去了,现在要监听该事件的发生,至于监听方法,前面已经演示过,直接上代码:

```
package ioc.test;
import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;
public class AnimalEventListener implements ApplicationListener {

    public void onApplicationEvent(ApplicationEvent event) {
        if (event instanceof AnimalSpeakEvent) {
            AnimalSpeakEvent a = (AnimalSpeakEvent) event;
            System.out.println("事件监听器" + this.getClass().getSimpleName()
                + ":有一个动物在讲话!它的名字是:" + a.getAnimalName());
        }
    }
}
```

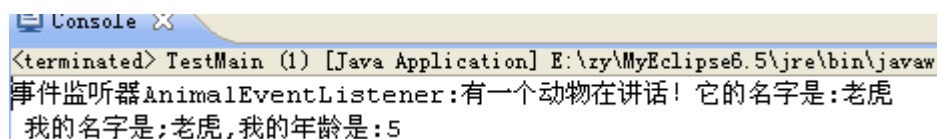
配置好 Bean，为 AnimalBean 注入值，配置文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans .....>
    <bean id="Listener" class="ioc.test.AnimalEventListener" />
    <bean id="Animal" class="ioc.test.Animal">
        <property name="name" value="老虎" />
        <property name="age" value="5" />
    </bean>
</beans>
```

现在应该来测试一下了，看看我们的自定义的事件是不是会再动物“讲话”的时候发生，测试类 TestMain 代码如下：

```
package ioc.test;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class TestMain {
    public static void main(String[] args) {
        AbstractApplicationContext ac = new
        ClassPathXmlApplicationContext(
            "applicationContext.xml");
        //从容器获取动物实例
        Animal animal = (Animal)ac.getBean("Animal");
        //让动物讲话
        System.out.println(animal.speak());
    }
}
```

运行该类中的主方法，输出结果如下：



```
<terminated> TestMain (1) [Java Application] E:\zy\MyEclipse6.5\jre\bin\javaw
事件监听器AnimalEventListener:有一个动物在讲话! 它的名字是:老虎
我的名字是:老虎,我的年龄是:5
```

图 3.10 例程 3.9 运行结果

可以看到，在动物“讲话”之前，我们的事件监听器监听到“动物讲话”，并输出了讲话动物的名字。

3.5.3 使用 **BeanFactoryPostProcessor** 工厂后置处理器

前面我们介绍过如何对受管 Bean 进行预处理。那么可以对 IoC 容器进行预处理吗？答案当然是肯定的。Spring 为我们提供了一个相应的接口 **BeanFactoryPostProcessor**，我们只需要编写一个该接口的实现类，就可以很方便的对 Spring IoC 容器进行预处理了。

Spring 会在 IoC 容器在加载完配置文件后、初始化受管 Bean 之前调用该接口实现类的 `postProcessBeanFactory()` 方法。通过这个方法，我们可以对 Spring IoC 容器进行任意的扩展或预处理。

和 **BeanPostProcessor** 类似，把该接口的实现类做为一个 Bean 配置到 Spring 配置文件中后，**ApplicationContext** 也会自动在配置文件中寻找它并加载。同时，如果使用 **BeanFactory** 容器，同样必须要先手动注册处理器。

下面的范例展示了如何使用该接口对 Spring IoC 容器进行预处理。我们将在该工程中创建一个 **ApplicationContext** 的后处理器，再处理器类中打印出容器所管理的所有的 Bean 的名字以及属性信息。

新建一个名字为 `ioc.test3.10` 的 java 工程，添加 Spring 开发能力后，创建一个名为 `ioc.test` 的包。创建 **MyBeanFactoryPostProcessor** 类，它实现了 **BeanFactoryPostProcessor** 接口，再其中的 `postProcessBeanFactory()` 方法中添加代码，以打印出容器管理的所有 bean 的名字和属性。代码如下：

```
package ioc.test;

//import省略

public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor
{
    public void postProcessBeanFactory(ConfigurableListableBeanFactory arg0)
        throws BeansException {
        // 打印出容器中定义的所有Bean的名字
        String[] beans = arg0.getBeanDefinitionNames();
        for (int i = 0; i < beans.length; i++) {
            System.out.println("定义的Bean" + i + ":" + beans[i]);
            // 打印出AnimalBean的配置信息
            System.out.println(arg0.getBeanDefinition(beans[i]));
        }
    }
}
```

创建一个 Animal 类，添加上 age, name 成员和相关的 Getter 和 Setter 方法，为了方便的打印，再添加一个 Speak 方法，放回 Animal 的相关信息，代码如下：

```
package ioc.test;

public class Animal {
    private String name;
    private int age;
    public String speak() {
        return "我的名字是: " + this.name + ", 我的年龄是: " + this.age;
    }
    //Getter 和 Setter 省略
}
```

再配置文件中配置好 Bean，再创建一个测试类 TestMain 就可以运行了，配置文件代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans .....>
    <bean id="animal" class="ioc.test.Animal">
        <property name="name" value="老虎" />
        <property name="age" value="3" />
    </bean>
    <bean id="MyBeanFactoryPostProcessor"
        class="ioc.test.MyBeanFactoryPostProcessor">
    </bean>
```

TestMain 类代码如下：

```
package ioc.test;

//import省略

public class TestMain {

    public static void main(String[] args) {

        ApplicationContext ac = new ClassPathXmlApplicationContext(

            "applicationContext.xml");

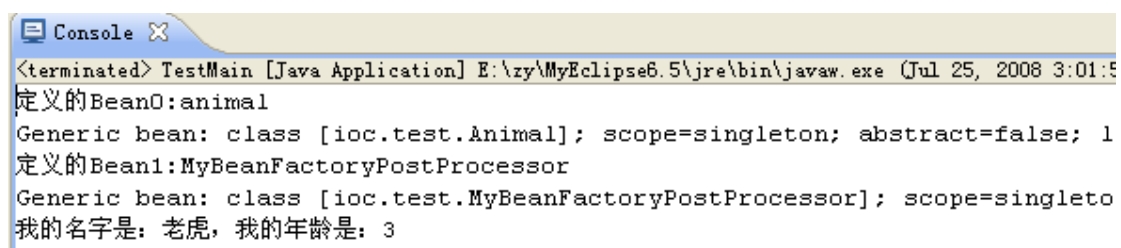
        Animal animal = (Animal) ac.getBean("animal");

        System.out.println(animal.speak());

    }

}
```

运行后得到输出结果:



```
<terminated> TestMain [Java Application] E:\zy\MyEclipse6.5\jre\bin\javaw.exe (Jul 25, 2008 3:01:5
定义的Bean0:animal
Generic bean: class [ioc.test.Animal]; scope=singleton; abstract=false; l
定义的Bean1:MyBeanFactoryPostProcessor
Generic bean: class [ioc.test.MyBeanFactoryPostProcessor]; scope=singleto
我的名字是: 老虎, 我的年龄是: 3
```

图 3.11 例程 3.10 运行结果

可以看到, `ApplicationContext` 已经自动的再配置文件中找到我们定义的容器前处理器并加载运行。

实际上, 在 Spring 中, 还为我们内置了一些 `ApplicationContext` 的前处理器来完成不同的功能。当我们需要某些功能的时候, 只需把这些内置的处理器类定义的配置文件中即可。内置的前处理器主要有 `PropertyPlaceholderConfigurer`、`PropertyOverrideConfigurer`、`CustomEditorConfigurer` 等。

3.5.4 定义配置元数据

比如说, 我们在一个工程中, 需要连接 MS-SQL 数据库, 那么在配置数据源的时候就要求配置数据库服务器的地址、用户名、密码等信息。但是如果将它发布出去后, 客户不同, 这些信息也就不同, 这时你总不能要求客户去修改你的 Spring 配置文件吧? 要解决这个问题就需要元数据了。

在这一节, 会用到上面的两个类 `PropertyPlaceholderConfigurer` 和 `PropertyOverrideConfigurer`, 用这两个类可以给我们的受管 Bean 配置元数据。由于这两个类实现了 `BeanFactoryPostProcessor` 接口, 因此它们实际上也是一种 Spring 内定的容器前处理器。我们只要在 Spring 配置文件中配置好它的相关属性而不需要在容器中注册。

下面通过一个例子来说明如何用这两个类来配置我们的元数据 (例程 3.11)。工程就不

用新建了，直接修改例程 3.10 就可以。先说步骤：

由于不再需要 `MyBeanFactoryPostProcessor` 类，直接从例程 3.10 中删除。然后在工程 `src` 目录里新建一个名字为 `AnimalInfo.properties` 的 `properties` 文件，名字可以随便取，但是扩展名不要变。利用 `MyEclipse6.5` 的属性编辑器，编辑它的内容，完成如下：

```
animal.name=\u5F20\u9F99
animal.age=90
```

其中 “`\u5F20\u9F99`” 是 “张龙” 的 Unicode 码。再从配置文件中删除 `MyBeanFactoryPostProcessorBean` 的定义，重新添加一个新的 Bean，`class` 属性为 `org.springframework.beans.factory.config.PropertyOverrideConfigurer`，`id` 不必配置，但是必须要利用它的 `location` 属性指明刚刚我们创建的属性文件（当然也可以使用数组类型的 `locations` 属性指明多个配置文件）。同时修改 `animal` 的配置，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans .....>
    <bean id="animal" class="ioc.test.Animal">
        <property name="name" value="${animal.name}" />
        <property name="age" value="${animal.age}" />
    </bean>
    <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="location" value="AnimalInfo.properties" />
    </bean>
</beans>
```

现在运行一下我们的主类，结果如下：

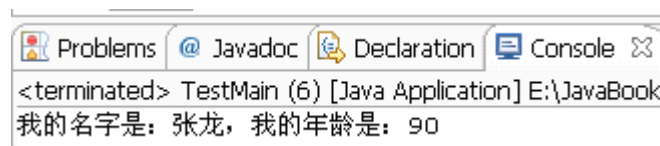


图 3.12 例程 3.11 运行结果 1

可以看到，`PropertyOverrideConfigurer` 类已经自动为我们 `animal` 注入了属性文件 `AnimalInfo.properties` 中定义的信息。其实 `PropertyOverrideConfigurer` 除了可以从属性文件中查找信息外，还会去 JVM 属性(`System.getProperty`)、系统环境变量(`System.getenv`)中来查找 `${**}` 信息。你可以分别通过它的 `systemPropertiesMode` 和 `searchSystemEnvironment` 属性来决定是否使用这两个功能。

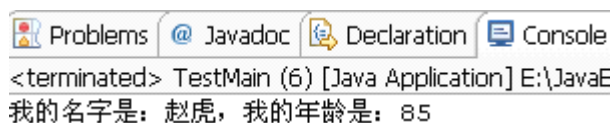
还有一个类 `PropertyOverrideConfigurer` 是 `PropertyPlaceholderConfigurer` 的孪生姐妹, 与 `PropertyPlaceholderConfigurer` 不同的是, 它会利用指定的配置文件中找到的信息强制性的替代受管 Bean 的定义。在例程 3.11 中新加一个属性文件 `AnimalInfo2.properties`, 内容为:

```
animal.name=\u8D75\u864E
animal.age=85
```

再将 `PropertyOverrideConfigurer` 增加到配置文件中, 代码如下:

```
<bean
  class="org.springframework.beans.factory.config.PropertyOverride
Configurer">
  <property name="location" value="AnimalInfo2.properties"/>
  <property name="ignoreResourceNotFound" value="true"/>
</bean>
```

其中 `location` 属性指定资源文件, `ignoreResourceNotFound` 属性表示没找到资源文件时是否抛出异常。再次运行主类, 输出如下:



```
<terminated> TestMain (6) [Java Application] E:\JavaE
我的名字是: 赵虎, 我的年龄是: 85
```

图 3.13 例程 3.11 运行结果 2

可以看到, `PropertyOverrideConfigurer` 已经强制覆盖了 `animal` 原来的配置。

3.6 受管 Bean 了解自己

有时候, 我们需要在 Bean 中获取它自身的配置信息, 比如 `id/name`、管理它的 IoC 容器的引用等。Spring 为我们提供了相应的接口, 在受管 Bean 中, 我们只是需要实现相关回调接口就可以轻松实现。

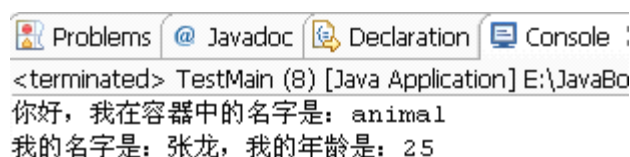
3.6.1 获取 Bean 自身在容器中的 Id

受管 Bean 要获得它自身在容器中的 Id 名称, 需要实现 `BeanNameAware` 回调接口。该接口定义了一个方法 `setBeanName(String name)`, 当受管 Bean 实现了该接口后, Spring IoC 容器会在该 Bean 的所有属性被设置后、初始化回调之前调用该方法为受管 Bean 注入它在容器中的 `id/name`。下面通过一个例程来说明如何使用该接口。

创建名字为 `IoC_Test3.11` 的工程, 添加 Spring 开发能力。创建一个 `ioc.test` 的包, 新建一个 `Animal` 类, 实现 `BeanNameAware` 接口, 代码如下:

```
package ioc.test;
import org.springframework.beans.factory.BeanNameAware;
public class Animal implements BeanNameAware{
    private String name;
    private int age;
    public String speak() {
        return "我的名字是: " + this.name + ", 我的年龄是: " + this.age;
    }
    public void setBeanName(String name) {
        System.out.println("你好, 我在容器中的名字是: "+name);
    }
    //Getter 和 Setter 省略
}
```

将它配置为 Bean，再创建含有主方法的测试类，代码和例程 3.10 完全一样。然后运行，输出结果如下：



```
<terminated> TestMain (8) [Java Application] E:\JavaBo
你好, 我在容器中的名字是: animal
我的名字是: 张龙, 我的年龄是: 25
```

图 3.14 例程 3.12 运行结果

可以看到, Spring IoC 容器已经调用 setBeanName 方法为我们的受管 Bean 注入了该 Bean 的 id。

3.6.2 获取 IoC 容器的引用

有时候，我们需要在受管 Bean 中发布事件、或者添加一个容器的后置处理器、又或者从容器中获取该 Bean 的依赖对象。那么这时候，我们就需要在受管 Bean 中获得指向 IoC 容器的引用。

要在受管 Bean 中获取一个指向管理它的 IoC 容器的引用，则需要该 Bean 实现 Spring 定义的 BeanFactoryAware 接口或者 ApplicationContextAware 接口，它们分别对应于不同的 IoC 容器。当该 Bean 被容器创建后，它就会被注入一个指向管理它的 BeanFactory 的引用。这样，该 Bean 可以通过该引用来操纵 IoC 容器了。

由于 ApplicationContextAware 接口的使用已经只在以前的 Spring 自定义事件的例子中使用，所以下面的例子展示如何使用 BeanFactoryAware 接口（工程代码见历程 3.13）。由于 BeanFactory IoC 容器只是提供有限的功能，因此除非有特别的原因，并不推荐使用该接口。

修改工程 3.12 的 Animal 类，让其实现 BeanFactoryAware 接口。在 setBeanFactory 方法中添加代码打印出容器中名字为 animal 的 bean，完成后代码如下：

```

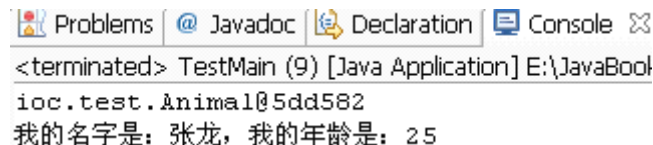
package ioc.test;

//import 省略

public class Animal implements BeanFactoryAware{
    private String name;
    private int age;
    public String speak() {
        return "我的名字是: " + this.name + ", 我的年龄是: " + this.age;
    }
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        //打印出容器中的名字为animal的bean
        System.out.println(beanFactory.getBean("animal"));
    }
    //Getter 和 Setter 省略
}

```

工程 3.12 的其它文件代码不变，直接运行主类输出结果如下：



```

<terminated> TestMain (9) [Java Application] E:\JavaBoo
ioc.test.Animal@5dd582
我的名字是: 张龙, 我的年龄是: 25

```

图 3.15 例程 3.13 运行结果

3.7 基于注解（Annotation-based）的配置（Spring2.5 新增）

在 Spring2.5 版本中，新增了一些 Annotation 注解，从而使得在 Spring2.5 中可以使用 Annotation 注解来驱动 Spring 依赖注入。Spring2.5 新增的注解是@Autowired，同时支持 JSR-250 规范的一些注解，如@Resource、@Resource 和@PreDestory 等。当然，要使用 Spring 的 Annotation 注解驱动应用，你必须使用 java 5 或者以上版本。

3.7.1 @Autowired 注解

@Autowired 注解提供了前面提到过的自动装配的功能，而且比起上面提到的方法更加细致和灵活。不同的是，@Autowired 标注采用 byType 的方法自动装配。

它可以标注成员变量：

```

@Autowired
private Host host;

```

注意：使用这种标注方法时，我们甚至可以省略掉 Getter 和 Setter 方法。

也可以标注传统的 Setter 方法：

```
@Autowired
public void setDisplay(Display display) {
    this.display = display;
}
```

还可以标注构造方法:

```
@Autowired
public Computer(Host host, Display display) {
    this.host = host;
    this.display = display;
}
```

还可以标注其它一个/多个参数的方法:

```
@Autowired
public void Setter(Host host, Display display) {
    this.host = host;
    this.display = display;
}
```

下面有一个例子可以展示如何使用该注解。在例子中, 有三个 Computer 类 ComputerA、ComputerB 和 ComputerC, 为了给他们的 Host 成员和 Display 成员注入值, 分别按照上面的方法使用了@Autowired 标注。

创建名字为 IoC_Test3.14 的工程, 添加 Spring 开发能力。创建 ioc.test 包, 添加 Host 类和 Display 类:

```
package ioc.test;
public class Display {
    public String run(){
        return "我是显示器, 正在运行! ";
    }
}
```

```
package ioc.test;
public class Host {
    public String run(){
        return "我是主机, 正在运行! ";
    }
}
```

再分别创建三个类 ComputerA、ComputerB 和 ComputerC, 它们分别使用前面提到的方法为其成员注入值:

ComputerA（标注成员变量和 Setter 方法）:

```
package ioc.test;
import org.springframework.beans.factory.annotation.Autowired;
public class ComputerA{
    @Autowired
    private Host host;
    private Display display;
    //电脑运行方法
    public void run() {
        System.out.println("你好, 我是电脑, 正在运行!");
        System.out.print("  "+host.run()+",");
        System.out.println(display.run());
    }
    @Autowired
    public void setDisplay(Display display) {
        this.display = display;
    }
}
```

ComputerB（标注构造方法）:

```
package ioc.test;
import org.springframework.beans.factory.annotation.Autowired;
public class ComputerB {
    private Host host;
    private Display display;
    @Autowired
    public ComputerB(Host host, Display display) {
        this.host = host;
        this.display = display;
    }
    //电脑运行方法
    public void run() {
        System.out.println("你好, 我是电脑, 正在运行!");
        System.out.print("  "+host.run()+",");
        System.out.println(display.run());
    }
}
```

ComputerC（标注多个参数的 Set 方法）:

```

package ioc.test;
import org.springframework.beans.factory.annotation.Autowired;
public class ComputerC {
    private Host host;
    private Display display;
    @Autowired
    public void Setter(Host host, Display display) {
        this.host = host;
        this.display = display;
    }
    //电脑运行方法
    public void run() {
        System.out.println("你好, 我是电脑, 正在运行!");
        System.out.print("    "+host.run()+"");
        System.out.println(display.run());
    }
}

```

要让标记能够工作, 就要在配置文件中配置 `AutowiredAnnotationBeanPostProcessor` 类, 它继承于 `BeanFactoryPostProcessor` 类, 是一个容器后置处理器。当容器启动后, 它就会在容器中扫描所有的 Bean, 发现 Bean 使用有 `@Autowired` 注释后就立即在容器中搜索其依赖对象, 并注入。该例程的配置代码如下:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans .....>
    <bean
        class="org.springframework.beans.factory.annotation.AutowiredAnnotation
        BeanPostProcessor">
    </bean>
    <bean id="computerA" class="ioc.test.ComputerA"></bean>
    <bean id="computerB" class="ioc.test.ComputerB"></bean>
    <bean id="computerC" class="ioc.test.ComputerC"></bean>
    <bean id="host" class="ioc.test.Host"></bean>
    <bean id="display" class="ioc.test.Display"></bean>
</beans>

```

这里配置解析 `@Autowired`
注解的容器后置处理器

最后, 创建含有主方法的测试类:

```
package ioc.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestMain {

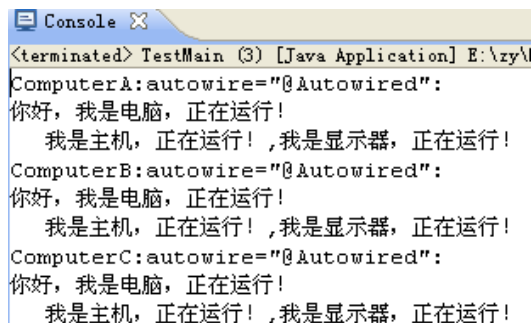
    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");

        //ComputerA
        ComputerA computerA = (ComputerA)ac.getBean("computerA");
        System.out.println("ComputerA:autowire=@"@Autowired");
        computerA.run();

        //ComputerB
        ComputerB computerB = (ComputerB)ac.getBean("computerB");
        System.out.println("ComputerB:autowire=@"@Autowired");
        computerB.run();

        //ComputerC
        ComputerC computerC = (ComputerC)ac.getBean("computerC");
        System.out.println("ComputerC:autowire=@"@Autowired");
        computerC.run();
    }
}
```

运行的结果如下：



```
<terminated> TestMain (3) [Java Application] E:\zy\l
ComputerA:autowire=@"@Autowired":
你好，我是电脑，正在运行！
    我是主机，正在运行！,我是显示器，正在运行！
ComputerB:autowire=@"@Autowired":
你好，我是电脑，正在运行！
    我是主机，正在运行！,我是显示器，正在运行！
ComputerC:autowire=@"@Autowired":
你好，我是电脑，正在运行！
    我是主机，正在运行！,我是显示器，正在运行！
```

图 3.16 例程 3.14 运行结果

当然，@Autowired 标签还可以用与来自容器的特殊 Beans，也可以作用于集合类型。

注意：有时候在开发期（尽管这种情况很少），使用了自动装配而又不允许装配的时候，可以使用 @Autowired(required = false)来显示的指定被它注解的字段不在自动装配的范围内。

在上面配置 @Autowired 标注解析器的时候，还可以使用另一种方式为我们自动配置：

```
<?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-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd">

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



这样 `<context:annotation-config/>` 就会为我们自动的隐式配置 `AutowiredAnnotationBeanPostProcessor` 、 `CommonAnnotationBeanPostProcessor` 、 `PersistenceAnnotationBeanPostProcessor` 和 `requiredAnnotationBeanPostProcessor` 4 个 `BeanPostProcessor`。

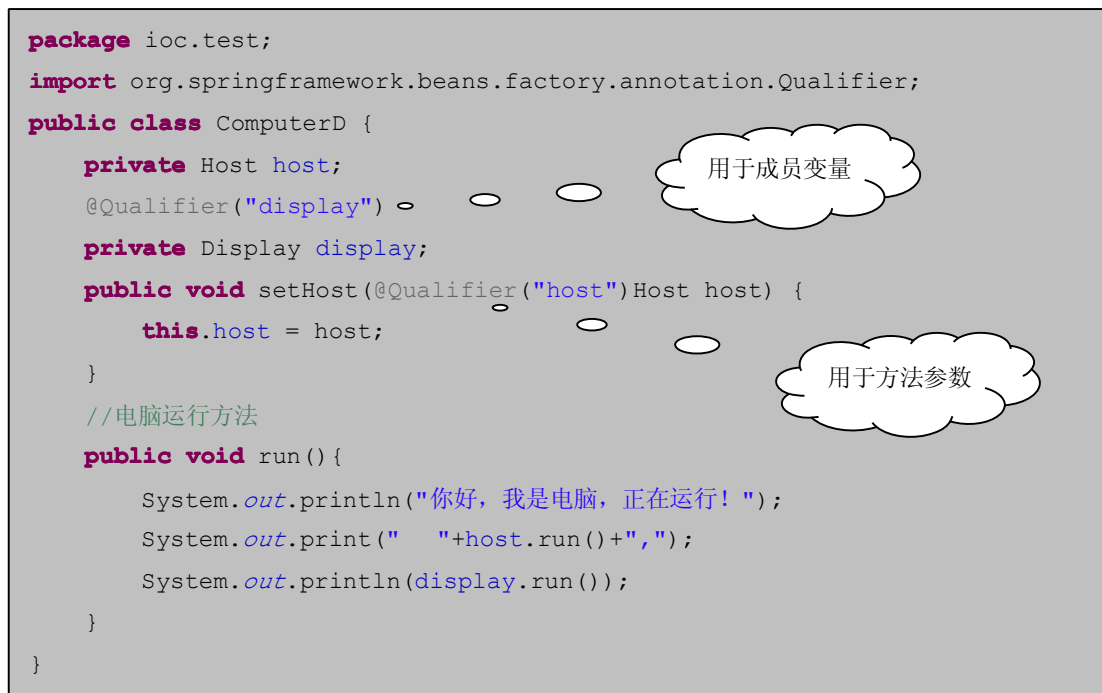
3.7.2 @Qualifier 注解

使用 `@Qualifier` 注解可以指定注入 Bean 的名称，用法如下：

```
@Qualifier("Bean 的名称")
```

`@Autowired` 注解可以应用在类的成员变量、成员方法和构造子，而 `@Qualifier` 注解则只能用于类的成员变量、方法的参数和构造子的参数。如果它与 `@Autowired` 联合使用，那么自动装配的策略就变为 `byName` 了。

在上面的例程 3.14 中新加一个类 `ComputerD`，在该类中我们练习使用 `@Qualifier` 注解（例程 3.15）。



3.7.3 @Resource 注解

@Resource 注解是 JSR-250 规范中的注解，需要 common-annotations.jar 包的支持，但是在 MyEclipse 自带的 Spring 中，该 jar 文件被精简了。因此，要在 MyEclipse 中使用该注解，需要从 Spring 依赖版的发布包中 libj2ee 目录下拷贝该文件到工程 classpath 下。为了方便以后的开发，也可以直接把它加入到 MyEclipse 中的 Spring Core 库中，如下图：

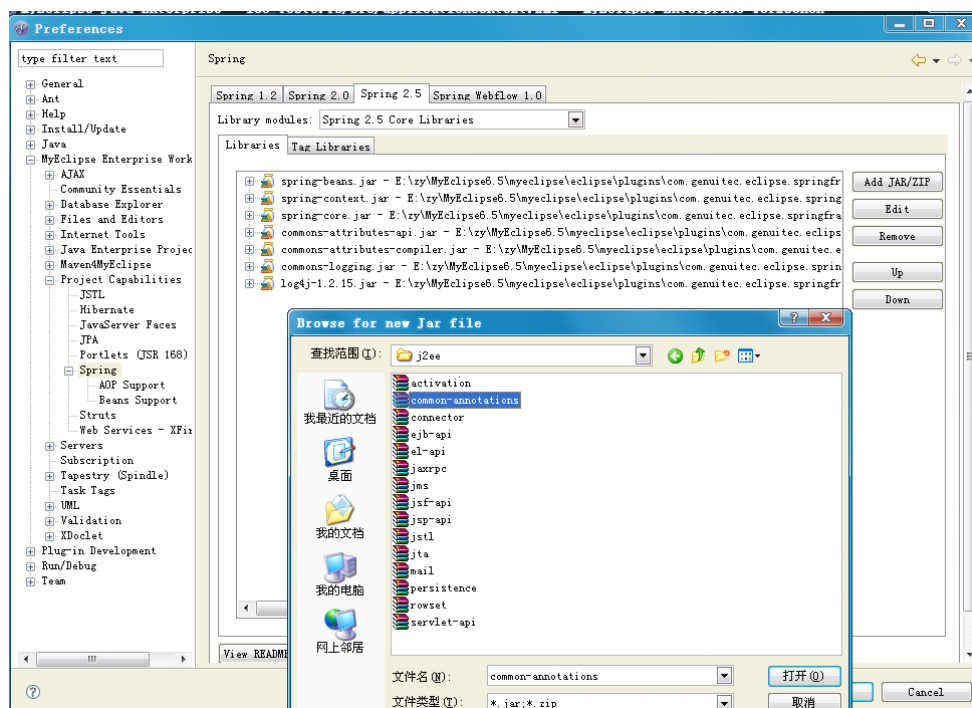


图 3.17 Spring 库中添加新 jar 文件

`@Resource` 注解有一个 `name` 属性和 `type` 属性, `name` 属性用于指定注入的 Bean 的名称, 与 `byName` 类型的 `Autowire` 类似, 如果没有显式的给出 `name` 属性, 那么它会从注释的字段名或者 `Setter` 方法名中获取; `type` 属性用于指定注入 Bean 的类型, 与 `byType` 类型的 `Autowire` 类似。如下例 (例程 3.16 ComputerA):

```
package ioc.test;
import javax.annotation.Resource;
public class ComputerA{
    @Resource(name="host")
    private Host host;
    private Display display;
    //电脑运行方法
    public void run() {
        System.out.println("你好, 我是电脑, 正在运行! ");
        System.out.print("    "+host.run()+"");
        System.out.println(display.run());
    }
    @Resource(type=Display.class)
    public void setDisplay(Display display) {
        this.display = display;
    }
}
```

注释成员变量, 指明 name 属性

注释方法, 指明 type 属性

3.7.4 @PostConstruct 和 @PreDestroy 注解

这两个注解也是 JSP-250 规范中的, 用来指定受管 Bean 的初始化方法和析构方法, 和前面提到的 `init-method` 和 `destroy-method` 属性类似。使用方法如下:

```

package ioc.test;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.beans.factory.annotation.Autowired;
public class ComputerB {
    private Host host;
    private Display display;
    @Autowired
    public ComputerB(Host host, Display display) {
        this.host = host;
        this.display = display;
    }
    //电脑运行方法
    public void run() {
        System.out.println("你好, 我是电脑, 正在运行! ");
        System.out.print("  "+host.run()+",");
        System.out.println(display.run());
    }
    @PostConstruct
    public void start() {
        System.out.println(this.getClass().getSimpleName()+" Bean预处理方法
ing.....");
    }
    @PreDestroy
    public void end() {
        System.out.println(this.getClass().getSimpleName()+" Bean析构方法
ing.....");
    }
}

```

@PostConstruct 注释指定与处理方法

@PreDestroy 注释指定析构方法

3.7.5 @Component 注解

虽然上面的方法大大的简化了我们的配置文件,但是我们还是不得不在配置文件中配置我们的受管 Bean,这也很麻烦,那么有办法可以让我们不用在 xml 中配置 Bean 呢? Spring 提供的@Component 注解可以很方便、优雅的完成这个任务。

把 Host 类配置为一个受管 Bean (见例程 3.16):

```
package ioc.test;
import org.springframework.stereotype.Component;
@Component
public class Host {
    public String run() {
        return "我是主机, 正在运行!";
    }
}
```

把 Host 类配置为一个受管 Bean

把 Display 类配置为一个名字为 display 的受管 Bean (见例程 3.16):

```
package ioc.test;
import org.springframework.stereotype.Component;
@Component("display")
public class Display {
    public String run() {
        return "我是显示器, 正在运行!";
    }
}
```

把 Display 类配置为一个名字为 display 的受管 Bean

要让它工作起来, 我们还需要在 Spring 的 xml 配置文件中加上这么一行:

```
<context:component-scan base-package="ioc.test"/>
```

`<context:component-scan />` 标签用 `base-package` 属性来指定要扫描的包的范围, Spring 会递归的扫描该属性指定的包和它子包中的所有类并处理, 同时使用该标签后 `AutowiredAnnotationBeanPostProcessor` 等容器后置处理器会被隐式的注册, 因此 `<context:annotation-config/>` 配置就不再必要了。

有时候, 我们还需要指定一个过滤器来包含或者排除某些特定的包和类, 从而自定义扫描。`<context:component-scan />` 有两个子元素用来定义过滤器: `<context:include-filter/>` 和 `<context:exclude-filter/>`, 它们都要求有一个 `type` 属性指定过滤器类型和 `expression` 属性指定过滤器的表达式。

Spring 为我们提供了如下几种过滤器:

1、 `annotation` 通过 Annotation 注解来定义被过滤的类。如下, 被 `@Repository` 注释的类就将不被扫描。

```
<context:component-scan base-package="ioc.test">
    <context:exclude-filter type="annotation"
        expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```

2、regex 通过正则表达式定义过滤的类。如下，ioc.test 包下的所有类将被扫描。

```
<context:component-scan base-package="ioc.test">
    <context:include-filter type="regex" expression="ioc.test.*"
/>
```

3、aspectj 通过 aspectj 表达式定义过滤的类。

```
<context:component-scan base-package="ioc.test">
    <context:include-filter type="aspectj" expression="
ioc.test..*DAO "/>
</context:component-scan>
```

3.7.6 @scope 注解

通常来说，受管 Bean 默认的作用范围是 singleton，在前面讲解过如何配置和使用它。在使用 Annotation 注解驱动的配置中，也可以指定容器中受管 Bean 的作用范围。如果我们需要别的改变一个受管 Bean 的作用范围的时候，就需要使用@scope 标签了。具体用法如下（工程代码见例程 3.17）：

```
package ioc.test;
//import省略

@Component("DateTimePrototype")
@Scope("prototype")
public class DateTimePrototype {
    private Date date;
    DateTimePrototype() {
        this.date = Calendar.getInstance().getTime();
    }
    public Date getDate() {
        return date;
    }
}
```

配置一个 Bean，并指定作用范围为 prototype

3.8 小结

在这一章，我们讨论了 Spring 的两个核心技术之一：控制反转（IoC）/依赖注入(DI)。介绍了如何在工程中使用 Spring 这项技术来简化我们的工作。本章以 Spring 中受管 Bean 的配置为主线，介绍了受管 Bean 配置、生命周期、Spring 的两个主要 IoC 容器以及如何在受管 Bean 中了解 Bean 自己，最后介绍了 Spring2.5 新增的 Annotation 注解驱动的配置方式。

第四章 IoC In Spring

4.1 什么是 AOP?

What is AOP? AOP 即 Aspect-Oriented Programming 的缩写, 中文意思是面向切面编程, 也有译作“面向方面编程”, 因为 Aspect 有“方面、见地”的意思。AOP 实际上是一种编程思想, 由 Gregor Kiczales 在 Palo Alto 研究中心领导的一个研究小组于 1997 年提出^[1]。在传统的面向对象 (Object-Oriented Programming, OOP) 编程中, 对垂直切面关注度很高, 横切面关注却很少, 也很难关注。也就是说, 我们利用 OOP 思想可以很好的处理业务流程, 却不能把业务流程中的某些特定的重复性行为封装在某个模块中。比如在很多的业务中都需要记录操作日志, 为此我们不得不在业务流程中嵌入大量的日志记录代码。无论是对业务代码还是对日志记录代码来说, 今后的维护都是非常复杂的。由于系统中嵌入了这种大量的与业务无关的其它重复性代码, 系统的复杂性、代码的重复性增加了, 从而使 bug 的发生的可能性也大大的增加。

在面向对象编程 (OOP) 中, 我们总是按照某种特定的执行顺序来实现业务流程, 各个执行步骤之间是相互衔接、相互耦合的。比如某个对象抛出了异常, 我们就必须对异常进行处理后才能进行下一步的操作; 又比如我们要把一个学生记录插入到教务管理系统的学生表中去, 那么我们就必须按照注册驱动程序、连接数据库、创建一个 statement、生成并执行 SQL 语句、处理结果、关闭 JDBC 对象等步骤按部就班的编写我们的代码。可以看到上面的步骤中除了执行 SQL 和处理结果是我们业务流程所必须的外, 其它的都是重复的准备和殿后工作, 与业务流程毫无关系。初次之外还必须要考虑程序执行过程中的异常。事实上, 我们只是需要向一张表中插入数据而已, 可是却不得不和这些大量的重复代码纠缠在一起, 我们不得不把大量的精力用在这些代码上而无法专心的设计业务流程。

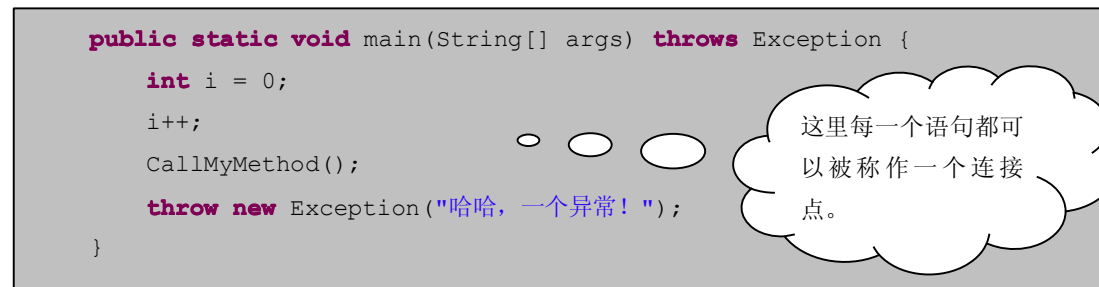
那么什么可以解决这个问题呢? 这时候, 我们需要 AOP, 关注系统的“截面”, 在适当的时候“拦截”程序的执行流程, 把程序的预处理和后处理交给某个拦截器来完成。比如在操作数据库时要记录日志, 如果使用 AOP 的编程思想, 那么我们在处理业务流程时不必再考虑日志记录, 而是把它交给一个特定的日志记录模块去完成, 即把所有日志记录代码写在一起, 然后在某个特定的时候动态的把日志记录代码“嵌入”到业务流程中去。这样, 业务流程就完全的从其它无关的代码中解放出来, 各模块之间的分工更加明确, 程序维护也变得容易多了。

AOP 不是一种技术, 而是编程思想。凡是符合 AOP 思想的技术, 都可以看成是 AOP

的实现。目前的 AOP 实现有 AspectJ、JBoss4.0、nanning、spring 等项目。其中 Spring 对 AOP 进行了很好的实现，同时 Spring AOP 也是 Spring 的两大核心之一。

下面介绍一些关于 AOP 的一些概念，如果觉得有些晦涩，也可以先不必仔细关注。

连接点 (join point): 它是指应用中执行的某个点，即程序执行流程中的某个点。如执行某个语句或者语句块、执行某个方法、装载某个类、抛出某个异常……，如下：



连接点是有“强弱”的，Spring AOP 支持到方法级的连接点粒度。

切入点 (pointcut): 切入点是连接点的集合。它通常和“通知”联系在一起，是切面和程序流程的交叉点。比如说，定义了这样一个 pointcut，它将“抛出异常 ClassNotFoundException”和某个“通知”联系起来，那么在程序执行过程中，如果抛出了该异常，那么相应的通知就会被触发执行。

通知 (advice): 也可以叫做“装备”，指切面在程序流程运行到某个连接点所触发的动作，在这个动作中我们可以定义自己的处理逻辑，比如日志记录。通知只有执行到某个包含在切入点的连接点时才会被触发。目前 AOP 定义了五种通知：前置通知 (Before advice)、返回后通知 (After returning advice)、环绕通知 (Around Advice)、异常通知 (After throwing advice)、引入通知 (Introduction advice)。这些通知以后会逐一介绍。

目标对象 (target object): 被一个或者多个切面通知的对象。所以它有时候也被称为 Advised Object。

引入 (introduction): 声明额外的成员字段或者成员方法。它可以给一个确定的目标对象新增某些字段或者方法。

织入 (weaving): 将切面和目标对象联系在一起的过程。这个过程可以在编译期完成，也可以在类加载时和运行时完成。Spring AOP 是在运行期完成织入的。

切面 (aspect): 一个关注点的模块化。它实际上是一段被织入到程序流程中的代码。

4.2 Spring AOP 概述

Spring AOP 实现了 AOP 联盟 (Alliance) 的制定的接口规范，它基于 java 的代理机制实现。JDK 从 1.3 开始就支持代理，所以目前 Spring AOP 基本上可以运行于任何地方——

没有人现在使用的 JDK 版本还在 1.3 以下吧？另一方面，由于 JDK 代理性能不佳，出现了 CGLIB 代理，Spring 中也可以使用。

AOP 作为 Spring 的核心技术之一，如何使用 Spring AOP 来为我们工作呢？这里用一个简单的范例来展示 Spring AOP 的使用，让大家对它有一个感性的认识。

我们仍然使用 Computer 来说明这个例子的工作过程。我们的 Computer 类有一些成员方法：post()、start()、run()等，他们用来表示电脑从加电自检到关机的一次运行过程。我们要做的就是利用 Spring AOP 来为这个过程做一个日志记录——在调用 Computer 对象的这些方法之前在控制台输出日志信息。

首先，我们创建一个名字为 AOP_Test4.1 的 java 工程(例程 4.1)。添加 Spring 开发能力，注意的是，在选择 Spring 库时应选中 Spring AOP 模块，如下图：

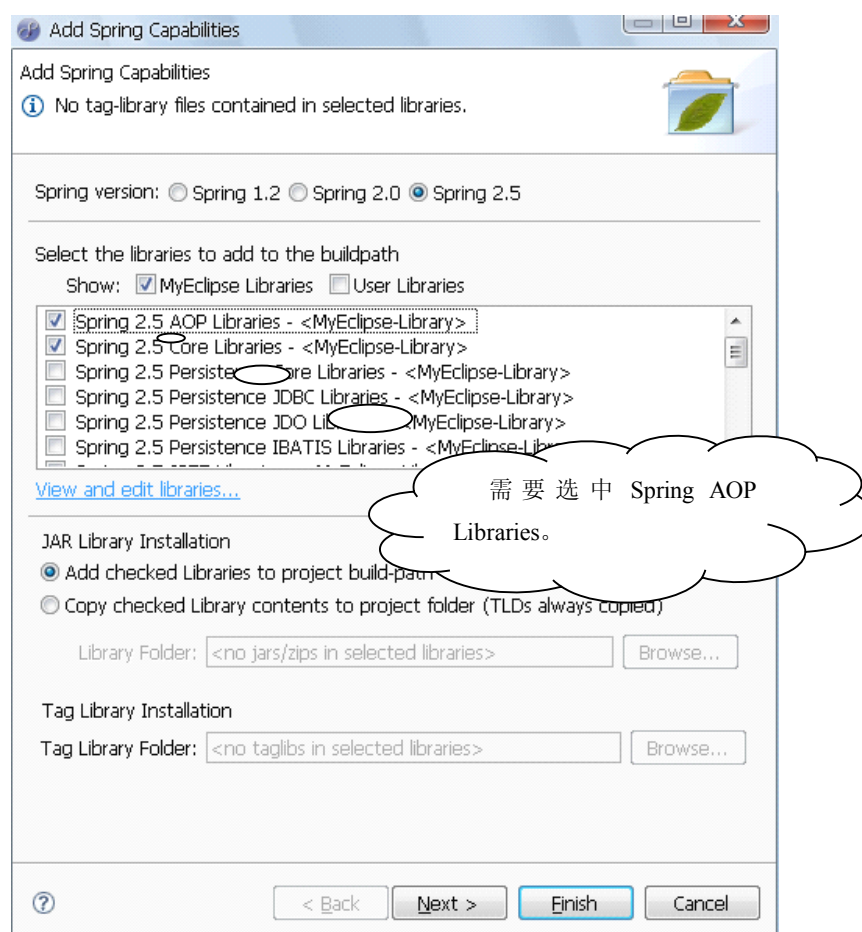


图 4.1 选择 Spring 模块

完成后创建一个 aop.test 包，在包中创建 Computer 类，并给 Computer 添加 java.lang.String 类型的 host、display、printer 成员字段。再为其添加上 post()、start()、run()等方法，完成后的代码如下：

```
package aop.test;

public class Computer {

    private String host="华硕主机";
    private String display="华硕显示器";
    private String priter="戴尔打印机";

    // 电脑加电自检
    public void post() {
        System.out.println("自检结果: \n 主机: " + this.host + ",显示器:"
            + this.display+ ",打印机: " + this.priter);
    }

    // 电脑启动
    public void start() {
        System.out.println("启动.....");
    }

    // 电脑运行
    public void run() {
        System.out.println("运行.....");
    }

    // 电脑关闭
    public void shutdown() {
        System.out.println("正在关闭.....");
    }

    // Getter 和 Setter 省略
}
```

在例子中，Computer 类的实例可以被称为“目标对象”，而对“目标对象”的方法调用则是“连接点”。

创建 Before advice，类名为 LoggerComputer，它实现了 MethodBeforeAdvice 接口。在 before()方法添加种输出日志信息的代码，提示某个方法将要被执行，该方法将会在 Computer 对象的各个方法执行之前被调用。代码如下：

```
package aop.test;

import java.lang.reflect.Method;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.aop.MethodBeforeAdvice;

public class LoggerComputer implements MethodBeforeAdvice {

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

    public void before(Method method, Object[] args, Object target)
        throws Throwable {
        //这里什么也不做，只是利用Log4J输出日志信息
        log.info("方法: " + method.getName() + "被执行!");
    }
}
```

由于在该例中，我们不使用 Spring IoC，而是使用编程式 AOP，直接创建代理对象 ProxyFactory，为其添加目标对象和通知（LoggerComputer）对象，所以到这里我们可以直接创建含有 main 方法的类 TestMain 了，而不必配置 Spring 的配置文件。代码如下：

```
package aop.test;

import org.springframework.aop.framework.ProxyFactory;

public class TestMain {

    public static void main(String[] args) {
        //创建代理工厂
        // ProxyFactory pf = new ProxyFactory(new Computer());
        ProxyFactory pf = new ProxyFactory();
        pf.addAdvice(new LoggerComputer()); //设置通知
        pf.setTarget(new Computer()); //设置目标对象

        Computer c = (Computer)pf.getProxy(); //获取目标对象
        c.post(); //运行方法，作为连接点
        c.start();
        c.run();
        c.shutdown();
    }
}
```

构造方法中设置目标对象。

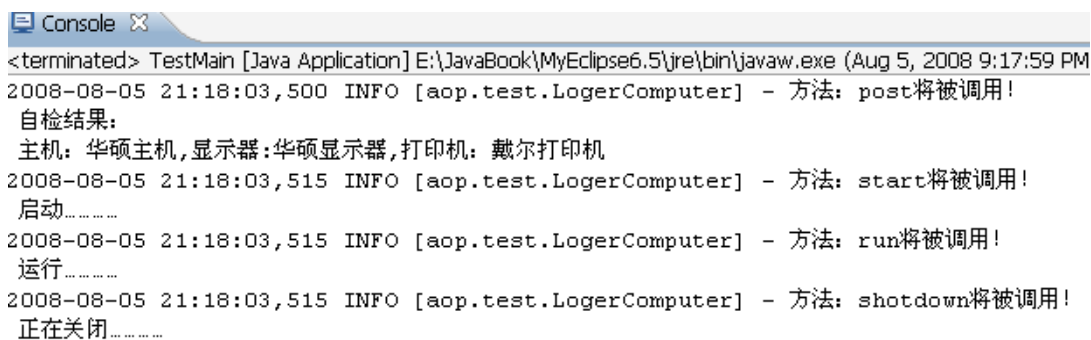
在 main 方法中我们创建一个代理对象 pf，并分别调用 addAdvice()和 setTarget()方法为它增加通知和设置目标对象，然后利用 getProxy()方法获取被代理的目标对象。当然，也可以在 ProxyFactory 的构造方法中设置目标对象，如上。

这时 Spring AOP 的部分已经算是完成了，但是我们仍然看不到结果，这是因为我们使用 Log4J 来输出信息，而我们还没有为工程添加 Log4J 的配置文件。在工程 src 目录下新建名字为 log4j.properties 的.properties 文件，内容如下：

```
log4j.rootLogger=INFO,stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] -
```

这里输出级别必须是 INFO，与第三章不同。

运行主类，控制台输出如下：



```
Console X
<terminated> TestMain [Java Application] E:\JavaBook\MyEclipse6.5\jre\bin\javaw.exe (Aug 5, 2008 9:17:59 PM)
2008-08-05 21:18:03,500 INFO [aop.test.LoggerComputer] - 方法: post将被调用!
自检结果:
主机: 华硕主机,显示器:华硕显示器,打印机: 戴尔打印机
2008-08-05 21:18:03,515 INFO [aop.test.LoggerComputer] - 方法: start将被调用!
启动.....
2008-08-05 21:18:03,515 INFO [aop.test.LoggerComputer] - 方法: run将被调用!
运行.....
2008-08-05 21:18:03,515 INFO [aop.test.LoggerComputer] - 方法: shutdown将被调用!
正在关闭.....
```

图 4.2 例程 4.1 运行结果

可以看到，在我们调用 Computer 对象的每个方法之前，我们定义的前置通知被触发，输出了相关信息。实际上，在 Before advice 中，我们可以做的远远不只于输出调试信息这么简单，以后将会详细介绍。

4.3 使用 ProxyFactoryBean

例程 4.1 使用编程式创建 AOP 代理来完成了监视 Computer 动作的功能，但是这种方式使通知、目标对象等 AOP 配置以硬编码的方式出现，不利于修改和维护，因此推荐使用这一节介绍的方式。

4.3.1 从一个范例开始

ProxyFactoryBean 类在 org.springframework.aop.framework 包中，和 ProxyFactory 一样，它继承自 ProxyCreatorSupport，提供了对 Advice 的良好支持。

下面用一个例子来介绍如何使用 ProxyFactoryBean（见例程 4.2）。在上面的例程 4.1 中，我们使用编程的方法配置 Spring AOP，这个例子里将和 Spring IoC 结合起来，在 xml 配置文件里配置 AOP，这也是从 Spring1.x 里就支持的最常见的配置方式。

修改例程 4.1 的 xml 配置文件, 分别创建目标对象 Computer Bean 和通知 LoggerComputer Bean。再利用 ProxyFactoryBean 将他们“联系”起来, 代码如下:



事实上 ProxyFactoryBean 类还暴露了其它的属性: proxyInterfaces 属性是一个 Class[] 类型的数组成员, 借助它可以指定代理接口的名字的集合。如果没有指定, 将会为目标类指定 CGLIB 代理; targetName 可以指定目标对象的名字, 从而可以不用使用 target 属性来指定引用的目标对象; singleton 用于设定工厂是否返回同一个被代理的对象。

现在修改主类, 从 Spring IoC 容器中获取 Computer 对象。代码如下:

```

package aop.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestMain {
    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");

        //通过ProxyFactoryBean获取Computer的实例
        Computer c = (Computer) ac.getBean("ProxyFactoryBean");
        c.post();
        c.start();
        c.run();
        c.shutdown();
    }
}

```



运行主类，可以得到和图 4.2 一致的结果。

```

2008-08-06 15:32:38,921 INFO [aop.test.LoggerComputer] - 方法: post被执行!
自检结果:
主机: 联想主机,显示器:宏碁显示器,打印机: 惠普打印机
2008-08-06 15:32:38,937 INFO [aop.test.LoggerComputer] - 方法: start被执行!
启动.....
2008-08-06 15:32:38,937 INFO [aop.test.LoggerComputer] - 方法: run被执行!
运行.....
2008-08-06 15:32:38,937 INFO [aop.test.LoggerComputer] - 方法: shutdown被执行!
正在关闭.....

```

图 4.3 例程 4.2 运行结果

4.3.2 使用 CGLIB 代理

前面提到过，由于 JDK 代理机制的性能不佳，以至出现了 CGLIB。与 JDK 代理相比，它不需要使用接口，性能也比 JDK 代理要好。在一些遗留的工程中，我们不会也不可能去给目标对象重构生成接口，这时 CGLIB 就显得特别有用。但是，使用 CGLIB 也需要考虑如下一些问题^[2]：


- 1、无法代理 final 方法，原因是 final 方法不能被覆写。
- 2、代理对象的构造方法会被调用两次。

3、需要 CGLIB 库的支持。要使用 CGLIB，则需要把 CGLIB 的二进制包放到工程的 classpath 下。在我们为工程添加 Spring 库时，MyEclipse 已经为我们添加上了 CGLIB 的相关包。

ProxyFactoryBean 类暴露了一个布尔型的 proxyTargetClass 属性，默认值为 false，这时 ProxyFactoryBean 会根据目标对象是否实现接口而自动的选择使用 JDK 代理还是 CGLIB 代理；设置为 true 则会直接代理目标类本身而不是目标类的接口，也就是说，设置为 true 时会强制的使用 CGLIB 代理。

使用方法如下：

```
<bean id="ProxyFactoryBean"
      class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target">
    <ref bean="Computer" />
  </property>
  <property name="interceptorNames">
    <list>
      <value>LoggerComputer</value>
    </list>
  </property>
  <property name="proxyTargetClass" value="true"></property>
</bean>
```



如果是使用@AspectJ 风格的 AOP，则使用如下方式：

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

注意：从 Spring2.0 开始，proxyTargetClass 属性没有显示的被指定为 true 时，ProxyFactoryBean 会自动选择代理模式，但是在 Spring1.2 中如果目标类没有实现任何接口，则必须显式的指定 ProxyFactoryBean 对象的 proxyTargetClass 属性为 true。与它相比，Spring2.0 的配置更简单、更智能化。

4.4 Spring 的 Pointcut

Pointcut 即切入点，用于配置切面的切入位置。由于 Spring 中连接点的粒度是方法级，因此在 Spring AOP 中 Pointcut 的作用是配置哪些类中的哪些方法在我们定义的切入点之内、哪些方法应该被过滤排除。Spring 的 Pointcut 分为静态 Pointcut、动态 Pointcut 和用户自定义 Pointcut 三种，其中静态 Pointcut 只是需要考虑类名、方法名等连接点的静态信息；动态 Pointcut 除此之外，还要考虑方法的参数，以便在运行时可以动态的确定切入点的位置；而用户也可以根据自己的需要扩展 Spring 定义的切入点类来自定义切入点。

在前面的例子中，我们只是确定了目标对象和通知而没有指定切入点，AOP 却仍然能够工作，这是为什么呢？ProxyFactoryBean 类继承了类 ProxyCreatorSupport 的 addAdvice

方法，当我们调用该方法添加一个通知到其通知列表中时，ProxyFactoryBean 对象会调用 addAdvisor() 方法来处理，然后自动的创建一个默认 Advisor——DefaultPointcutAdvisor 的实例，并把该 Advice 加入其中，默认的切入点会处理目标类的所有的方法。

4.4.1 静态 Pointcut

方法和类的名称在应用中往往是不变的，因此我们可以根据类和方法的签名来判定那些类的哪些方法在我们定义的切入点之内、哪些应该被过滤排除。静态切入点即使这样的一类切入点。Spring 中所有的切入点类都实现了 org.springframework.aop.Pointcut 接口，而其中的静态切入点则继承自 StaticMethodMatcherPointcut 抽象类。

Spring 提供了如下几个静态 Pointcut 的实现类，介绍如下：

1、StaticMethodMatcherPointcut：一个抽象的静态 Pointcut，它不能被实例化。开发者可以自己扩展该类来实现自定义的切入点。

2、NameMatchMethodPointcut：继承自 StaticMethodMatcherPointcut，只能对方法名进行判别的静态 Pointcut 实现类。它的 List 类型的 mappedNames 属性用于指定包含在切入点中的方法名，支持通配符。

使用范例如下：

```
<bean id="NameMatchMethodPointcut"
      class="org.springframework.aop.support.NameMatchMethodPointcut">
  <property name="mappedNames">
    <list>
      <value>pos*</value>
      <value>start</value>
    </list>
  </property>
</bean>
```

指定切入点中被包含的方法名。post*表示包含所有以 pos 开始的方法。(大小写敏感)

此外，NameMatchMethodPointcut 还暴露了 ClassFilter 类型的 classFilter 属性，可以用于指定 ClassFilter 接口的实现类来设置类过滤器。ClassFilter 接口的定义如下：

```
package org.springframework.aop;

public interface ClassFilter {

    boolean matches(Class clazz);

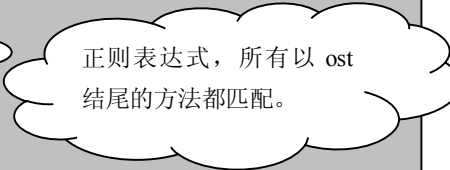
    ClassFilter TRUE = TrueClassFilter.INSTANCE;

}
```


其中 `matches()` 方法用于类的匹配, 参数 `clazz` 是需要匹配的目标类, 匹配成功则返回 `true`。

3、`JdkRegexpMethodPointcut`: 继承自 `StaticMethodMatcherPointcut`, 使用 JDK 正则表达式对方法名进行匹配的静态 `Pointcut`。范例如下:

```
<bean id="JdkRegexpMethodPointcut"
      class="org.springframework.aop.support.JdkRegexpMethodPointcut">
  <property name="patterns">
    <list>
      <value>.*ost</value>
      <value>.*tart</value>
    </list>
  </property>
  <property name="excludedPatterns">
    <list>
      <value>.*tart</value>
    </list>
  </property>
</bean>
```



在该例中, `patterns` 属性是一个 `java.lang.String[]` 类型的数组成员, 该数组中所有的正则表达式匹配的方法都包含在该切入点内。 `excludedPatterns` 则相反, 表示被排除于切入点范围的方法。同时和 `NameMatchMethodPointcut` 一样, 它也可以用 `ClassFilter` 类型的 `classFilter` 属性来定义类过滤器。

下面以 `JdkRegexpMethodPointcut` 为例, 通过一个完整的范例展示如何使用静态切入点 (完整工程代码见例程 4.3)。在工程中我们定义一个 `People` 类和一个切面, 并将他们在 Spring xml 配置文件中联系起来。在主类中, 当 `People` 对象执行我们切入点中定义的方法时, 前置通知 `LoggerPeople` 将会给出相应的提示信息。

新建一个工程 `AOP_Test4.3`, 添加 Spring 开发库后, 新建 `aop.test` 包。

创建目标类 `People`, 该类有 `speak()`、`Running()`、`Loving()`、`died()` 四个成员方法。代码如下:

```
package aop.test;

public class People{
    // 讲话
    public void speak() {
        System.out.println("Hello, 我是People! ");
    }
    // 跑步
    public void Running() {
        System.out.println("我在跑.....跑.....逃.....");
    }
    // 恋爱
    public void Loving() {
        System.out.println("我在和MM恋爱.....别来打搅我!");
    }
    // 死亡
    public void died() {
        System.out.println("完了, 我死了");
    }
}
```

创建一个类名为 `LoggerPeople` 的前置通知, 它实现 `MethodBeforeAdvice` 接口, 在 `before()` 方法中利用 `Log4J` 输出相关信息, 代码如下:

```
package aop.test;

import java.lang.reflect.Method;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.aop.MethodBeforeAdvice;

public class LoggerPeople implements MethodBeforeAdvice {

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

    public void before(Method method, Object[] args, Object target)
        throws Throwable {
        // 这里什么也不做, 只是利用Log4J输出日志信息
        log.info(target.getClass().getSimpleName() + "正在" +
            method.getName() + "!!");
    }
}
```

再编写 Spring 配置文件，完成后如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans .....>
    <bean id="Computer" class="aop.test.People"></bean>
    <bean id="LoggerComputer" class="aop.test.LoggerPeople" />

    <bean id="ProxyFactoryBean"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target">
            <ref bean="Computer" />
        </property>
        <property name="interceptorNames">
            <list>
                <value>DefaultAdvisor</value>
            </list>
        </property>
    </bean>

    <bean id="DefaultAdvisor"
        class="org.springframework.aop.support.DefaultPointcutAdvisor">
        <property name="pointcut" ref="JdkRegexpPointcut" />
        <property name="advice" ref="LoggerComputer" />
    </bean>

    <bean id="JdkRegexpPointcut"
        class="org.springframework.aop.support.JdkRegexpMethodPointcut">
        <property name="patterns">
            <list>
                <value>.*spea.*</value>
                <value>.*ing</value>
                <value>.*di.*</value>
            </list>
        </property>
        <property name="excludedPattern" value=".*Run.*" />
    </bean>
</beans>
```

指定要监视的方法和排除的方法。

为了让 ProxyFactoryBean 使用我们定义的 JdkRegexpMethodPointcut 而不是默认的 Pointcut，我们需要配置一个切入点配置器 PointcutAdvisor，其 advice 属性指定通知，Pointcut 属性指定切入点。然后再将该切入点配置器注入给 ProxyFactoryBean。各个 Bean 的依赖关系如下：

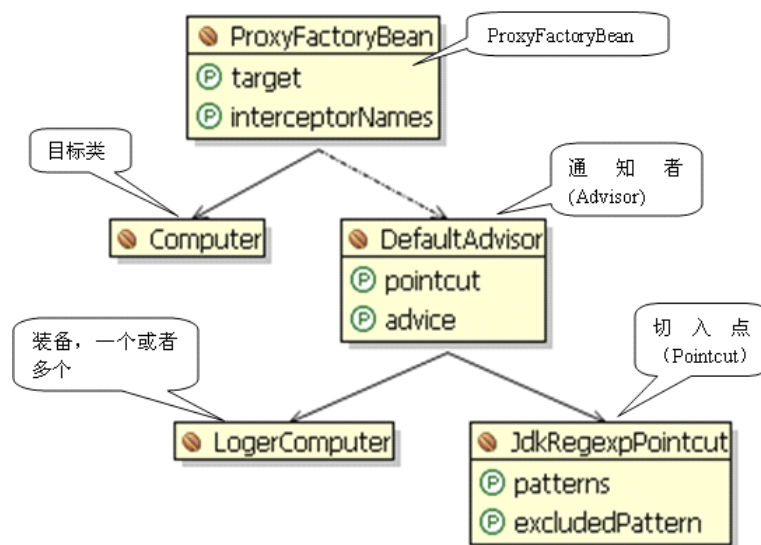


图 4.4 ProxyFactoryBean 代理生成

在 JdkRegexpMethodPointcut 中，我们使用了它两个属性 patterns 和 excludedPattern：patterns 利用正则表达式指定了该切入点中包含的方法；excludedPattern 指定了我们要排除的方法。

注意：

- 1、 “.*spea.*” 表示所有名字以 spea 开头的方法，例程中是指 speak 方法；
- 2、 “.*ing” 表示所有名字以 ing 结束的方法，例程中是指 Running 和 Loving 方法；
- 3、 “.*di.*” 表示所有名字以 di 开头的方法，例程中是指 died 方法；
- 4、 “.*Run.*” 表示所有名字以 Run 开头的方法，例程中是指 Running 方法；
- 5、 关于正则表达式的详细说明请参见其它相关文档。

创建包含主方法的测试类 TestMain，在 main() 方法中从 ProxyFactoryBean 中获得 People 对象，并逐个调用该对象的方法，代码如下：

```
package aop.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestMain {

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");

        //通过ProxyFactoryBean获取IComputer接口实现类的实例
        People c = (People) ac.getBean("ProxyFactoryBean");
        c.speak();
        c.Running();
        c.Loving();
        c.died();
    }
}
```

该类运行结果如下:

```
2008-08-07 15:28:53,828 INFO [aop.test.LoggerPeople] - People正在speak!
Hello, 我是People!
我在跑.....跑.....逃.....
2008-08-07 15:28:53,828 INFO [aop.test.LoggerPeople] - People正在Loving!
我在和MM恋爱.....别来打搅我!
2008-08-07 15:28:53,828 INFO [aop.test.LoggerPeople] - People正在died!
完了, 我死了
```

图 4.5 例程 4.3 运行结果

可以看到 People 类中的 speak()、Loving()、died()方法已经被拦截。但是 Running()方法却没有拦截,这是因为我们在 JdkRegexpMethodPointcut 中指定其 excludedPattern 属性把它排除在切入点之外的缘故。

4.4.2 动态 Pointcut

由于动态切入点除了要考虑方法的名称等静态信息外,还要考虑方法的参数。由于它是动态的,在执行时既要计算方法的静态信息,还要计算其参数,结果也不能被缓存。因此,动态切入点要消耗更多的系统资源。

Spring 中提供了如下几种动态切入点的实现,说明如下:

1、**ControlFlowPointcut**: 控制流程切入点。比如只有在某个特定的类或方法中调用某个连接点时, 通知才会被触发, 这时就可以使用 **ControlFlowPointcut**。但是它的系统开销很大, 在追求高效的应用中, 不推荐使用。

2、**DynamicMethodMatcherPointcut**: 动态方法匹配器。是抽象类, 扩展该类可以实现自己的动态 **Pointcut**。

4.4.3 自定义切入点

如前所述, **Spring** 定义了很多的切入点实现类。通过扩展这些类我们可以很轻松的自定义自己的切入点。继承这些类后, 重载 **matches()** 方法, 可以自定义方法匹配逻辑。继承 **StaticMethodMatcherPointcut** 或者 **DynamicMethodMatcherPointcut** 类, 可以分别实现基本的静态切入点和动态切入点。当然你也可以直接实现 **Pointcut** 接口实现 **getClassFilter()** 和 **getMethodMatche()** 方法自定义类过滤器和方法匹配器。

下面我们通过继承扩展 **StaticMethodMatcherPointcut** 类来实现一个自定义的静态切入点 (例程 4.4)。这个例程基于例程 4.3, 我们只是需要自定义一个扩展自 **StaticMethodMatcherPointcut** 的 **Pointcut**, 再将它配置给 **DefaultAdvisor Bean** 就可以了。

在例程 4.3 中新加一个类, 继承自 **StaticMethodMatcherPointcut** 类, 并在 **matches** 方法中自定义匹配规则, 代码如下:

```
package aop.test;

import java.lang.reflect.Method;
import org.springframework.aop.support.StaticMethodMatcherPointcut;

public class MyPointcut extends StaticMethodMatcherPointcut {

    public boolean matches(Method method, Class arg1) {
        if (method.getName().equals("speak") //匹配speak方法
            || method.getName().equals("Running") //匹配Running方法
            || method.getName().equals("Loving") //匹配Loving方法
            || method.getName().equals("died")) { //匹配died方法
            return true;
        }
        return false;
    }
}
```

修改例程 4.3 中的 **applicationContext.xml** 配置文件, 使 **DefaultAdvisor** 引用我们自定义的 **Pointcut**, 代码如下:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans ..... >
    <bean id="People" class="aop.test.People"></bean>
    <bean id="LoggerPeople" class="aop.test.LoggerPeople" />
    <bean id="ProxyFactoryBean"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target">
            <ref bean="People" />
        </property>
        <property name="interceptorNames">
            <list>
                <value>DefaultAdvisor</value>
            </list>
        </property>
    </bean>
    <bean id="DefaultAdvisor"
        class="org.springframework.aop.support.DefaultPointcutAdvisor">
        <property name="pointcut">
            <bean class="aop.test.MyPointcut"/>
        </property>
        <property name="advice" ref="LoggerPeople"/>
    </bean>
</beans>

```



使用自定义的 Pointcut

运行主类，输出结果如下：

```

2008-08-08 14:05:06,578 INFO [aop.test.LoggerPeople] - People正在speak!
Hello, 我是People!
2008-08-08 14:05:06,578 INFO [aop.test.LoggerPeople] - People正在Running!
我在跑.....跑.....逃.....
2008-08-08 14:05:06,578 INFO [aop.test.LoggerPeople] - People正在Loving!
我在和mm恋爱.....别来打搅我!
2008-08-08 14:05:06,578 INFO [aop.test.LoggerPeople] - People正在died!
完了，我死了

```

图 4.6 例程 4.4 运行结果

4.4.4 使用 Advisor

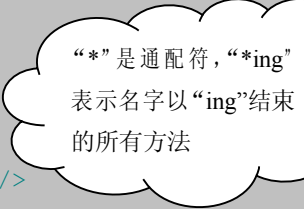
前面我们使用过一个通知者 DefaultPointcutAdvisor。实际上，Advisor 是切入点配置器，它可以将通知织入到切入点定义的程序流程中。

Spring 内置了几种继承自 AbstractGenericPointcutAdvisor 类的 Advisor。分别介绍如下：

- 1、 DefaultPointcutAdvisor：默认的通知者。也是 Spring 中最强大的 Advisor，它支持所有的 Pointcut，默认的切入点会包括类中的所有方法。
- 2、 NameMatchMethodPointcutAdvisor：内置了 NameMatchMethodPointcut 切入点的


Advisor，使用它可以不必再单独的配置 NameMatchMethodPointcut 切入点。它暴露的 mappedNames 和 mappedName 属性可以通过方法名来定义切入点中的方法。Order 属性确定其加载的顺序。使用方法如下：

```
<bean id="NameMatchMethodPointcutAdvisor"
class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
  <property name="mappedNames">
    <list>
      <value>speak</value>
      <value>*ing</value>
      <value>die*</value>
    </list>
  </property>
  <property name="advice" ref="LoggerPeople" />
</bean>
```



3、 RegexpMethodPointcutAdvisor：内置了 JDK 正则表达式切入点 JdkRegexpMethodPointcut 的 Advisor，通过它可以不必再配置 JdkRegexpMethodPointcut。直接给 patterns 和 pattern 属性注入正则表达式即可实现方法过滤。使用方法如下：

```
<bean id="RegexpMethodPointcutAdvisor"
class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
  1、 <property name="patterns">
    <list>
      <value>.*spea.*</value>
      <value>.*ing</value>
      <value>.*di.*</value>
    </list>
  </property>
  <property name="advice" ref="LoggerPeople" />
</bean>
```



下面通过一个完整的范例展示如何使用 Advisor（例程 4.5），该范例基于例程 4.3。我们不必更改例程 4.3 中的任何 Java 源代码，只是修改其 Spring 配置文件即可完成，利用新配置的 RegexpMethodPointcutAdvisor 替代原来的 DefaultPointcutAdvisor，完成后配置文件的代码如下：


```
<?xml version="1.0" encoding="UTF-8"?>
<beans .....>
  <bean id="People" class="aop.test.People"></bean>
  <bean id="LoggerPeople" class="aop.test.LoggerPeople" />
  <bean id="ProxyFactoryBean"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
      <ref bean="People" />
    </property>
    <property name="interceptorNames">
      <list>
        <value>RegexMethodPointcutAdvisor</value>
      </list>
    </property>
  </bean>
  <bean id="RegexMethodPointcutAdvisor"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="patterns">
      <list>
        <value>.*spea.*</value>
        <value>.*ing</value>
        <value>.*di.*</value>
      </list>
    </property>
    <property name="advice" ref="LoggerPeople"/>
  </bean>
</beans>
```

各 Bean 之间的依赖关系如图：

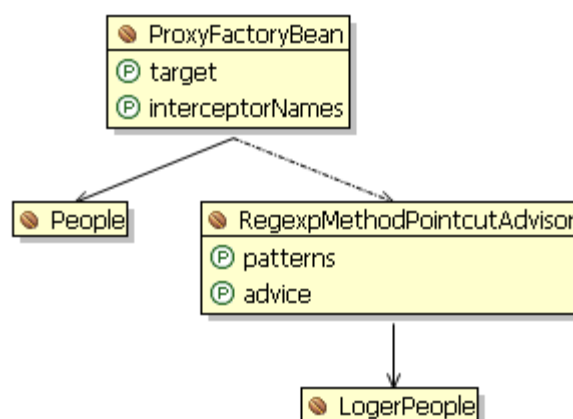


图 4.7 例程 4.5 运行结果

可见，使用这种方式的配置代码简洁许多。

运行的结果和例程 4.4 完全相同。

注意：与单独定义 `JdkRegexpMethodPointcut` 不同的是，`RegexpMethodPointcutAdvisor` 没有其内置 `JdkRegexpMethodPointcut` 的 `excludedPattern` 属性，因此它不支持定义要排除在切入点之外的方法过滤器，但是强大的正则表达式可以弥补这个缺点。

4.5 Spring AOP 的支持通知类型

Spring 支持 AOP 联盟定义的五种通知类型：前置通知（Before advice）、返回后通知（After returning advice）、环绕通知（Around Advice）、异常通知（After throwing advice）、引入通知（Introduction advice）。这一节将逐一介绍这些通知的使用方法。

4.5.1 前置通知（Before advice）

前置通知 Before advice，顾名思义，它会在连接点（目标对象的方法）执行之前触发执行。无论何种情况，连接点方法总是会得到执行，因此它适合监视程序流程、记录日志等场合——前置通知中抛出一个未经处理的异常的时例外，因为“throws Throwable”语句会把该前置通知中抛出的异常返回给调用者，使程序执行流程中断。前面关于 AOP 的例子中，由于我们只是在连接点执行前输出提示信息，因此都是使用前置通知，这样可以降低因为我们忘记调用 `proceed()` 方法而使整个程序崩溃的风险。要定义一个前置通知，需要实现 `org.springframework.aop.MethodBeforeAdvice` 接口，并在其定义的 `before()` 方法中编写处理逻辑，Spring 中该接口的源代码如下：

```
package org.springframework.aop;

import java.lang.reflect.Method;

public interface MethodBeforeAdvice extends BeforeAdvice {

    void before(Method method, Object[] args, Object target) throws
    Throwable;
```

在 `before()` 方法中，`method` 参数是即将执行的目标对象的方法，`args` 是方法的参数，`target` 是目标对象。

4.5.2 返回后通知（After returning advice）

返回后通知会在连接点被执行完毕成功返回后被调用。在实际的程序执行流程中，连接点先于本通知之前执行，因此如果在连接点中存在未处理的异常，该通知则不会被触发。要定义该通知，可以实现 `org.springframework.aop.AfterReturningAdvice` 接口，该接口定义如下：

```
package org.springframework.aop;

import java.lang.reflect.Method;

public interface AfterReturningAdvice extends AfterAdvice {

    void afterReturning(Object returnValue, Method method, Object[] args,
        Object target) throws Throwable;

}
```

在AfterReturningAdvice接口中，定义了afterReturning()方法，我们可以在该方法中编写自己的处理逻辑。该方法定义了四个参数：returnValue是目标方法的返回值；method是目标方法；arg目标方法的参数，是一个Object的数组；target是目标对象。

下面通过一个范例（例程 4.6）来展示如何使用 After returning advice，以及在连接点中抛出异常时有什么情况发生。该例程仍然修改例程 4.5，相对例程 4.5 来说，我们只是需要修改其中的 LoggerPeople 类，让其实现 AfterReturningAdvice 接口，从而摇身一变为返回后通知。修改完成后代码如下：

```
package aop.test;

import java.lang.reflect.Method;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.aop.AfterReturningAdvice;

public class LoggerPeople implements AfterReturningAdvice {

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

    public void afterReturning(Object returnValue, Method method, Object[]
        args, Object target) throws Throwable {

        // 这里什么也不做，只是利用Log4J输出日志信息
        log.info(target.getClass().getSimpleName() + "已经" +
            method.getName() + "!!");

    }

}
```

运行主类，控制台输出结果如下：

```
Hello, 我是People!
2008-08-12 10:47:53,046 INFO [aop.test.LoggerPeople] - People已经speak!
我在跑.....跑.....逃.....
2008-08-12 10:47:53,046 INFO [aop.test.LoggerPeople] - People已经Running
我在和MM恋爱.....别来打搅我!
2008-08-12 10:47:53,046 INFO [aop.test.LoggerPeople] - People已经Loving!
完了，我死了
2008-08-12 10:47:53,046 INFO [aop.test.LoggerPeople] - People已经died!
```

图 4.8 例程 4.6 运行结果（1）

可以看到，在 `People` 对象的方法执行返回后，我们的定义的通知已经被调用。但是如果目标对象的方法中抛出异常，那么结果如何呢？下面将做个试验：

修改 `People` 目标类的 `died()` 方法，让它抛出一个异常，方法代码如下：

```
// 死亡
public void died() throws Exception {
    System.out.println("完了，我死了");
    throw new Exception("有个异常，没死成！");
}
```

同时，测试类的代码也需要修改，用 `try.....catch` 包围 `c.died()` 语句，主方法代码如下：

```
public static void main(String[] args){
    ApplicationContext ac = new ClassPathXmlApplicationContext(
        "applicationContext.xml");
    //通过ProxyFactoryBean获取IComputer接口实现类的实例
    People c = (People) ac.getBean("ProxyFactoryBean");
    c.speak();
    c.Running();
    c.Loving();
    try {
        c.died();
    } catch (Exception e) {
        System.out.println(e);
        //e.printStackTrace();
    }
}
```

再次运行主类，输出结果如下：

```

Hello, 我是People!
2008-08-12 13:05:10,562 INFO [aop.test.LoggerPeople] - People已经speak!
我在跑.....跑.....逃.....
2008-08-12 13:05:10,562 INFO [aop.test.LoggerPeople] - People已经Running!
我在和MM恋爱.....别来打搅我!
2008-08-12 13:05:10,562 INFO [aop.test.LoggerPeople] - People已经Loving!
完了, 我死了
java.lang.Exception: 有个异常, 没死成!

```

图 4.9 例程 4.6 运行结果 (2)

可以看到, 返回后通知中的日志记录语句并没有被执行。实际上在调用目标方法时, 程序的执行流程是: 连接点→返回后通知→返回给调用者方法的执行结果, 所以在连接点抛出异常而未被显示的处理时, 下面的代码自然也就被中断了。

另外需要注意的一点是, 在 **AfterReturningAdvice** 通知的 **afterReturning()** 方法中无法修改目标方法的参数和返回值, 这里所说的“无法修改”是指不能有效的影响传入目标方法的参数和它的返回结果。

4.5.3 环绕通知 (Around Advice)

有过 win32 的编程经验的朋友一定知道 API Hook 技术, 在 Windows 编程中, 我们可以 Hook 某些 windows 的 API 函数, 通过改变这些函数调用时传入的参数和返回值, 从而改变目标 API 函数的行为, 很多病毒就是通过这种技术来达到隐藏自身文件和进程的。在 Spring AOP 中, 环绕通知的特性类似于 API Hook, 可以完全控制连接点的处理逻辑。我们甚至可以决定连接点是否被调用、调用的参数以及调用后的返回值, 以达到完全 “欺骗” 调用者的目的。这是 Spring AOP 中功能最强大的通知。但是在使用时, 开发者往往容易忘记调用 **proceed()** 方法来执行连接点, 从而通知链中断。因此**如非必要, 不推荐使用该通知**。

要使用该通知, 可以实现 **org.aopalliance.intercept.MethodInterceptor.MethodInterceptor** 接口, 接口定义如下:

```

package org.aopalliance.intercept;

public interface MethodInterceptor extends Interceptor {

    Object invoke(MethodInvocation invocation) throws Throwable;

}

```

MethodInterceptor 接口定义了 **invoke()** 方法, 其 **invocation** 参数描述了目标对象的方法。返回值将会作为连接点的返回值返还给调用者。

下面通过例程来展示使用该通知(例程 4.7), 该例程仍然基于例程 4.5。让其 **LoggerPeople**

类实现 `MethodInterceptor` 接口，并实现 `invoke()` 方法，在方法中修改目标方法的返回值，代码如下：

```
package aop.test;

//import省略
public class LoggerPeople implements MethodInterceptor{

    public Object invoke(MethodInvocation arg0) throws Throwable {
        //这里做个判断，只是修改Loving方法。
        if(arg0.getMethod().getName().equals("Loving")){
            return "哼！我不爱"+arg0.getArguments()[0]+"!";
        }
        return arg0.proceed();
    }
}
```

运行主类中的 `main()` 方法，可以看到原来输出的“啊！I Love :MM”已经变成“哼！我不爱 MM!”了，可见环绕通知的功能相当强大。

4.5.4 异常通知（After throwing advice）

异常通知和前面的几个不同，它在只是在连接点抛出异常时才触发。要使用该通知，需要实现 `org.springframework.aop.ThrowsAdvice` 接口，该接口只没有提供任何方法，它只是一个标志。和返回后通知（`AfterReturningAdvice`）一样，`ThrowsAdvice` 接口继承自 `AfterAdvice` 接口，也就是说，它在连接点执行完毕返回后才被触发。在该接口的实现类中，必须要有一个的异常处理方法，该方法格式如下：

```
void afterThrowing([Method method, Object[] args, Object target],
                   Throwable ThrowableSubclass);
```

方法的名字必须为 `afterThrowing`，其中 `ThrowableSubclass` 参数是必须的，其他如 `method` 等中括号内的参数可有可无。

`afterThrowing()` 方法可以被重载，Spring AOP 会自动的选择一个合适的方法调用。同时 `ThrowableSubclass` 参数的类型也可以决定该方法处理的异常的类型，例如在一个 `ThrowsAdvice` 子类中定义如下三个方法：

```
void afterThrowing(Throwable throwable)
```

```
void afterThrowing(RuntimeException throwable)
```

```
void afterThrowing(RemoteException throwable)
```

在应用执行中，如果连接点抛出了 `RuntimeException` 异常，第二个方法会优先被调用，尽管 `RuntimeException` 是 `Throwable` 的子类；如果连接点抛出 `RemoteException` 异常，第三个方法会被优先调用，如果是其他的异常类型，第一个方法会被调用。

当通知的 `afterThrowing()` 方法也抛出异常时，它将把该异常返回给调用者，而连接点抛出的异常将会被它替换掉。一个异常通知的例子如下（完整的工程代码见例程 4.8）：

LoggerPeople 异常通知类：

```
package aop.test;

//import省略
public class LoggerPeople implements ThrowsAdvice {

    public void afterThrowing(Method method, Object[] args, Object target,
        Throwable throwable) {

        System.out.println("连接点发生异常, " + method.getName() +
            ",异常通知1正在处理: "+ throwable);

        // throw new Exception("异常通知发生了异常!");
    }

    public void afterThrowing(Throwable throwable) {
        System.out.println("Throwable 异常通知: " + throwable);
        // throw new Exception("异常通知发生了异常!");
    }

    public void afterThrowing(RuntimeException throwable) {
        System.out.println("RuntimeException 异常通知: " + throwable);
        // throw new Exception("异常通知发生了异常!");
    }

    public void afterThrowing(RemoteException throwable) {
        System.out.println("RemoteException 异常通知: " + throwable);
        // throw new Exception("异常通知发生了异常!");
    }

}
```

People 目标类：

```
package aop.test;

import java.rmi.RemoteException;

public class People{

    //讲话
    public void speak() {
        System.out.println("Hello, 我是People! ");
    }

    //跑步
    public void Running() {
        System.out.println("我在跑.....跑.....逃.....");
    }

    // 恋爱
    public void Loving() throws RemoteException {
        System.out.println("我在和MM恋爱.....别来打搅我!");
        throw new RemoteException("恋爱程序发生异常! 这个失恋啦!");
    }

    //死亡
    public void died() throws RuntimeException{
        System.out.println("完了, 我死了");
        throw new RuntimeException("死亡程序发生异常! 这个人死不得!");
    }

}
```



该例的 xml 配置文件、测试类和其它的例子大同小异，具体的代码可见例程 4.8，这里就不再赘述了。

4.5.5 引入通知 (Introduction advice)

引入通知是一种特殊的通知，它能够将新的成员变量、成员方法引入到目标类中。与其他的通知不同，它是类级别的。使用引入通知，如非必要，Spring 推荐扩展 `org.springframework.aop.support.DelegatingIntroductionInterceptor` 类来定义自己的引入通知。

`DelegatingIntroductionInterceptor` 类实现了 `IntroductionInterceptor` 接口和 `invoke()` 方法。引入通知被触发时，目标对象的任何方法调用都会托管给该方法执行。如果是目标对象原有的方法，那么可以使用 `proceed()` 方法来处理；如果是从混入接口新引入的方法，则必须完成该新方法的调用。`DelegatingIntroductionInterceptor` 类已经实现了这个功能，因此如果需要

重写该方法来获取目标对象的方法的参数或者返回值等信息，使用 `return super.invoke()` 语句就可以轻松的实现该功能。

引入通知只能和它专有的通知者 `IntroductionAdvisor` 一起使用，而不兼容其它任何通知者，因为它应用在类级别而其它的通知者则是在方法级别。`IntroductionAdvisor` 是一个接口，Spring 提供了一个它的实现类 `org.springframework.aop.support.DefaultIntroductionAdvisor`。

我们可以通过一个例子来展示如何使用引入通知为目标类引入新的方法(完整代码见例程 4.9)。在例程中，有一个 `Bill` 账单类，它记录某人本月的收支记录，但是没有提供计算收支总和的方法，这里我们利用引入通知为它添加一个 `sum` 方法计算的收支总和。

新建名字为 `AOP_Test4.9` 的工程，添加 Spring 开发库后，新建一个 `aop.test` 包。再创建一账单类 `Bill`，代码如下：

```
package aop.test;

public class Bill {

    private float computer; // 电脑支出（收入）
    private float pay; // 工资
    private float lottery; // 彩票
    private float other; // 其他

    public void setComputer(float computer) {
        this.computer = computer;
    }

    public void setPay(float pay) {
        this.pay = pay;
    }

    public void setLottery(float lottery) {
        this.lottery = lottery;
    }

    public void setOther(float other) {
        this.other = other;
    }
}
```

声明一个混入接口 `ICompute`，它定义了我们给 `Bill` 目标类引入的方法 `sum`，代码如下：

```
package aop.test;

public interface ICompute {
    // 求和
    public float sum();
}
```

创建一个引入通知类 `Compute`，它扩展自 `DelegatingIntroductionInterceptor` 类，同时也实现了 `ICompute` 接口。在 `Compute` 类中实现 `ICompute` 接口定义的 `sum` 方法，并重写 `DelegatingIntroductionInterceptor` 类的 `invoke` 方法，以获取各收入支出项目的金额。代码如下：

```
package aop.test;

import org.aopalliance.intercept.MethodInvocation;
import org.springframework.aop.support.DelegatingIntroductionInterceptor;

public class Compute extends DelegatingIntroductionInterceptor implements
    ICompute {

    private static final long serialVersionUID = 1L;
    // 记录收支的总和
    private static float sum;

    // 实现ICompute接口定义的方法
    public float sum() {
        return Compute.sum;
    }

    @Override
    public Object invoke(MethodInvocation mi) throws Throwable {
        if (mi.getArguments().length > 0) {
            // 判断调用的方法是否是目标对象的set方法。
            if (mi.getMethod().getName().indexOf("set") == 0
                && mi.getArguments()[0] instanceof Float) { // 判断方法参数类
                型

                Float f = (Float) mi.getArguments()[0]; // 强制类型转换
                this.sum += f.floatValue(); // 把该项收入加入总和
            }
        }
        return super.invoke(mi);
    }
}
```

在 Spring 配置文件中配置引入通知，配置方法和前面其它例子大致相同。完成后各类的依赖关系如下：

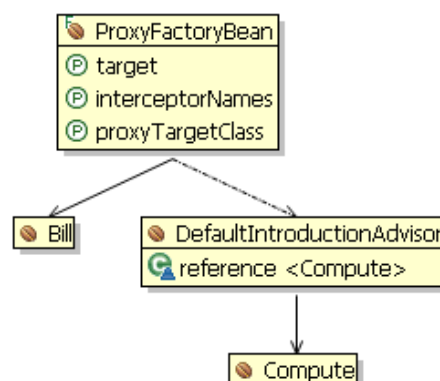


图 4.10 例程 4.9 Bean 依赖关系

配置代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans .....>

    <bean id="Bill" class="aop.test.Bill"/>
    <bean id="Compute" class="aop.test.Compute" />

    <bean id="ProxyFactoryBean"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target">
            <ref bean="Bill" />
        </property>
        <property name="interceptorNames">
            <list>
                <value>DefaultIntroductionAdvisor</value>
            </list>
        </property>
        <property name="proxyTargetClass" value="true"/>
    </bean>

    <bean id="DefaultIntroductionAdvisor"
        class="org.springframework.aop.support.DefaultIntroductionAdvisor">
        <constructor-arg ref="Compute" />
    </bean>
</beans>

```

编写含有 main() 方法的主类 TestMain，在 main() 方法中我们从代理获取账单 Bill 对象并计算账单这中各项收支的总和，代码如下：

```

package aop.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestMain {
    public static void main(String[] args) {

        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");

        // 通过ProxyFactoryBean获取IComputer接口实现类的实例
        Bill b = (Bill) ac.getBean("ProxyFactoryBean");
        b.setComputer(-3520f); //买电脑花费
        b.setPay(80000f); //工资收入
        b.setLottery(-56.2f); //买彩票花费
        b.setOther(-600f); //其他花费
        ICompute cb = (ICompute)b;
        System.out.println("收支结余: "+cb.sum()); //输出结余
    }
}

```

运行主类，控制台将会输出计算结果，如下：

```

2008-08-14 20:29:30,33
2008-08-14 20:29:40,13
2008-08-14 20:29:40,30
收支结余: 75823.8

```

图 4.11 例程 4.9 输出结果

4.6 使用自动代理

在前面的例子中，我们一直使用 ProxyFactoryBean 来显式的创建 AOP 代理。但是在很多场合，这种方式将会使编写配置文件的工作量大大增加；由于要从 ProxyFactoryBean 获得代理对象，也会使应用和 Spring AOP 之间的耦合度增加。这一节将介绍使用自动代理来解决这类问题。

4.6.1 使用 BeanNameAutoProxyCreator

Spring 提供的 BeanNameAutoProxyCreator 类允许我们通过 Bean 的 name 属性来指定代理的 Bean。它暴露了 java.lang.String[] 类型的 beanNames 和 interceptorNames 属性。beanNames 可以指定被代理的 Bean 名字列表，支持 “*” 通配符，例如 “*DAO” 表示所有名字以 “DAO” 结尾的 Bean。interceptorNames 指定通知 (Advice) 列表，或者通知者 (Advisor) 列表。

下面通过一个例程来演示如何使用 BeanNameAutoProxyCreator。在例子中，有两个 Bean: TestBeanA 和 BeanB，并在 TestMain 类中的 main 方法中调用其 MyMethod()方法。自动代理将会在方法调用前自动的执行配置的前置通知，输出提示信息。

新建一个名字为 AOP_Test4.10 的工程，添加 Spring 的 IoC 和 AOP 库后，新建 aop.test 包，再分别创建两个类 TestBeanA 和 BeanB，添加 MyMethod()方法，代码如下：

```
package aop.test;

public class TestBeanA {

    public void MyMethod() {
        System.out.println(this.getClass().getName()
            + ".MyMethod() is run!");
    }
}
```

```
package aop.test;

public class BeanB {

    public void MyMethod() {
        System.out.println(this.getClass().getName()
            + ".MyMethod() is run!");
    }
}
```

再创建前置通知类 BeforeAdvice:

```
package aop.test;

import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class BeforeAdvice implements MethodBeforeAdvice {

    public void before(Method method, Object[] args, Object target)
        throws Throwable {
        System.out.println(method.getName() + "(), 将要运行!");
    }
}
```

最后创建含有 main 方法的测试类 TestMain:

```
package aop.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestMain {

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");

        TestBeanA beanA = (TestBeanA) ac.getBean("TestBeanA");
        beanA.MyMethod();

        BeanB beanB = (BeanB) ac.getBean("BeanB");
        beanB.MyMethod();
    }
}
```

在配置文件中配置 Bean 和自动代理 Bean，完成后代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans .....>

    <bean id="TestBeanA" class="aop.test.TestBeanA"/>
    <bean id="BeanB" class="aop.test.BeanB"/>

    <bean id="BeforeAdvice" class="aop.test.BeforeAdvice"></bean>

    <bean class="org.springframework.aop.framework.autoproxy.
        BeanNameAutoProxyCreator">
        <property name="beanNames">
            <list>
                <value>Test*</value>
            </list>
        </property>
        <property name="interceptorNames">
            <list>
                <value>BeforeAdvice</value>
            </list>
        </property>
    </bean>
</beans>
```

通配符，“Test*”表示
id/name 属性以 Test 开头的
所有 Bean。

运行主类，输出结果如下：

```
MyMethod(), 将要运行!
aop.test.TestBeanA.MyMethod() is run!
aop.test.BeanB.MyMethod() is run!
```

图 4.12 例程 4.10 输出结果

可以看到，在主类 TestMain 中，我们是直接从 Spring IoC 容器中获取收管 Bean 而不是像以前那样从 ProxyFactoryBean 中获取代理 Bean，但是我们的前置通知 BeforeAdvice 仍然在 TestBeanA 对象的 MyMethod() 方法执行前被触发，这说明了我们的自动代理正在工作。

4.6.2 使用 DefaultAdvisorAutoProxyCreator

DefaultAdvisorAutoProxyCreator 允许我们只需定义相应的 Advisor 通知者，就可以完成自动代理。配置好 DefaultAdvisorAutoProxyCreator 受管 Bean 后，它会自动查找配置文件中定义的 Advisor，并将它们作用于所有的 Bean。

修改例程 4.10 的配置文件，使用 DefaultAdvisorAutoProxyCreator 来完成自动代理。完成后配置文件代码如下（本例完整工程代码见例程 4.11）：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans .....>
    <bean id="TestBeanA" class="aop.test.TestBeanA" />
    <bean id="BeanB" class="aop.test.BeanB" />
    <bean id="BeforeAdvice" class="aop.test.BeforeAdvice"/>

    <bean class="org.springframework.aop.framework.autoproxy.
DefaultAdvisorAutoProxyCreator">
    </bean>

    <bean class="org.springframework.aop.support.NameMatchMethod
PointcutAdvisor">
        <property name="advice" ref="BeforeAdvice" />
        <property name="mappedNames">
            <list>
                <value>*Method*</value>
            </list>
        </property>
    </bean>

</beans>
```

只需配置一个
Bean 即可。

运行主类输出结果如下：

```
MyMethod(), 将要运行!  
aop.test.TestBeanA.MyMethod() is run!  
MyMethod(), 将要运行!  
aop.test.BeanB.MyMethod() is run!
```

图 4.13 例程 4.11 输出结果

4.7 @AspectJ 风格的 AOP

前面提到过, AspectJ 也是一种十分优秀的 AOP 实现, 它在 AOP 上的功能比 Spring AOP 强大, 但是更复杂, 学习难度也更大。从 AspectJ 5 开始, 它支持基于 Annotation 注解的配置。Spring 从 2.0 开始同样支持注解驱动的 AOP, 而且采用 AspectJ 的切入点解析器来完成切入点的分析和匹配, 因此它切入点表达式语言和 AspectJ 基本相同。在这一节将介绍如何使用 @Aspect 风格的基于 Annotation 驱动的 Spring AOP。

4.7.1 一些必要的准备工作

AspectJ 使用了 java 5 的注解, 而 Spring AOP 使用了和 AspectJ 一样的注解, 因此它要求你的 JDK 版本必须在 1.5 或者以上。

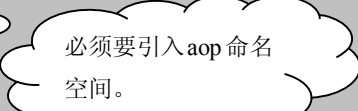
要在 Spring 配置中使用 @AspectJ 风格的切面声明, 我们需要先在 Spring 配置文件中启用该配置, Spring 为此提供了两种方式:

如果使用的是 DTD 的配置方式, 可以在 Spring 中配置一个 Bean, 该 Bean 的 class 属性为 org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator, 它和其它任何的普通 Bean 一样。代码如下:

```
<bean class="org.springframework.aop.aspectj.annotation.  
AnnotationAwareAspectJAutoProxyCreator"
```

也可以使用 <aop:aspectj-autoproxy /> 标签来启用该功能, 但是在使用之前需要加入 aop 的命名空间。代码如下:


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
  <aop:aspectj-autoproxy />
  .....其他Bean配置
</beans>
```



同时该标签还有 `proxy-target-class` 属性，用于指定是否是要强制使用 CGLIB 代理。如：

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

4.7.2 声明切面

在 `@Aspect` 风格的 Spring AOP 中可以把切面写成一个普通的 java 类，用 `@Aspect` 注解该类，然后把它配置成一个普通的受管 Bean，这就完成了切面的声明。

这里，为了让大家有一个感性的认识，我们用一个简单的例子来说明，例子中涉及的切入点配置等方面的内容将在后面逐一介绍。在例子中我们使用一个前置通知拦截 `TestBean` 类的 `MyMethod()` 方法，在该方法被调用前输出提示信息。完整的例程代码见例程 4.12。

首先，我们创建一个 `TestBean` 类，把它配置为一个普通的受管 Bean，然后在测试类从 Spring IoC 容器中获取它的实例，调用 `MyMethod()` 方法，步骤如下：

建立名字为 `AOP_Test4.12` 的工程，添加 Spring core 和 aop 开发库，创建 `aop.test` 包，再在包中创建 `TestBean` 类，代码如下：

```
package aop.test;

public class TestBean {
    public void MyMethod() {
        System.out.println(this.getClass().getName() +
                           ".MyMethod() is run!");
    }
}
```

创建测试类 `TestMain`，代码如下：

```

package aop.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestMain {

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");

        TestBean beanA = (TestBean)ac.getBean("TestBean");
        beanA.MyMethod();
    }
}

```

在 xml 配置文件中配置 Bean，代码如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans .....>
    <bean id="TestBeanA" class="aop.test.TestBean" />
</beans>

```

运行主类，将会直接在屏幕上打印出 “aop.test.TestBean.MyMethod() is run!”。

现在我们要给该工程加一个前置通知，在 MyMethod() 方法执行前给出提示信息。

添加一个类 MyAspect，用 @Aspect 注解声明为一个切面，MyAspect 类的代码如下：

```

package aop.test;

//import省略
@Aspect
public class MyAspect{

    @Pointcut("execution(* TestBean.*(..))")
    public void Before() {}

    @Before("Before()")
    public void BeforeTestBeanA(JoinPoint point) {
        TestBean bean = (TestBean) point.getTarget();
        System.out.println(bean.getClass().getName() + "."
            + point.getSignature().getName() + "(), 将要运行!");
    }
}

```

并把该类注册为一个受管 Bean，再修改 xml 配置文件，启用 Aspect，完成后完整的代

码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
  <aop:aspectj-autoproxy proxy-target-class="true"/>
  <bean id="TestBean" class="aop.test.TestBean" />
  <bean class="aop.test.MyAspect" />
</beans>
```

运行主类, 输出如下:

```
aop.test.TestBean.MyMethod(), 将要运行!
aop.test.TestBean.MyMethod() is run!
```

图 4.14 例程 4.12 输出结果

可以看到, 在连接点执行之前有提示信息输出, 这说明我们配置的切面已经被成功的织入到目标对象中。

在这个例子中, 切面实际上就是一个普通的被`@Aspect`注解的受管Bean。对于IoC容器来说, 它和其他的受管Bean没有多大区别, 如果必要, 我们甚至可以给它注入依赖对象, 或者使用上一章提到的Annotation注解来配置Bean。

4.7.3 使用切入点

在例程 4.12 中我们使用了`@Pointcut` 注解来声明了一个切面, 然后在 Before 前置通知引用了这个切面。在这一节, 我们将详细讨论`@Pointcut` 注解及其使用的切入点表达式。

1、 声明及引用一个切入点

在 Spring1.x 里, 我们通过 Spring 提供的 Pointcut 接口的实现类来定义我们自己的切入点, 比如 `NameMatchMethodPointcut`、`JdkRegexpMethodPointcut` 等。而利用 Spring2.x 提供的注解方式, 声明一个切入点将变的更加的方便和灵活。

比如在上面的例子中, 我们使用`@Pointcut` 注解来声明一个切入点:

```
@Pointcut("execution(* TestBean.*(..))")
public void Before() {}
```

切入点表达式

在这里我们定义了一个名字为 “Before()” 的切入点, 传给该注解`@Pointcut` 的值

“`execution(* TestBean.*(..))`”是切入点表达式，它将匹配 `TestBean` 类中的所有方法。我们可以在通知中这样引用它：

```
@Before("Before()")
public void BeforeTestBeanA(JoinPoint point) {
    .....代码
}
```

当然，有时候为了方便，也只需要直接在声明通知时定义一个切入点，但是它不能被其它的通知引用：

```
@Before("execution(* TestBean.*(..))")
public void BeforeTestBeanA(JoinPoint point) {
    .....代码
}
```

2、 execution 关键词

前面我们使用了形如 “`execution(* TestBean.*(..))`” 的切入点表达式，其中 `execution` 是关键词，括号中的内容是子表达式，格式如下（方括号内为可选项）：

```
execution([可见性] 返回类型 [声明类型].方法名(参数) [异常])
```

可见性：可选项。一般java类的方法、字段成员都有可见性，如 `public`、`private`、`protected`。Pointcut表达式的可见性仍然遵循java语义，“`public`”将匹配所有的`public`方法，开发者可以使用 “`*`” 通配符匹配所有的可见性。

返回类型：必选项。可以用它匹配连接点方法的返回类型，如 `void`、`String`、`int` 等，用 “`*`” 通配符可以匹配所有的返回类型。

声明类型：可选项。用于匹配 java 包。如 `aop.test` 等，可用 “`*`”、“`+`” 等通配符。

方法名：必选项。用于匹配连接点方法名，可以使用 “`*`” 通配符。实际上，声明类型和方法名组合成了理解点的全路径。

参数：必选项。指定连接点的参数类型及个数，“`..`” 通配符匹配任何可能的情况。

异常：可选项。用于匹配连接点抛出异常的类型。

下面有一个使用 `execution` 的例子：

```
@Pointcut("execution(public void aop.test.TestBean.MyMethod2(..) throws
Exception)")
public void Pointcut1() {
```

3、 切入点表达式通配符

在切入点表达式中，我们可以使用通配符，其中使用最多的包括 “`*`”、“`..`” 和 “`+`”。

其中“*”匹配所有的字符，但是“.”除外，一般匹配单个包，单个方法，单个参数等；“..”也匹配所有的字符，但是包括“.”，一般用于匹配多个包，多个参数。“+”表示类及其子类。

“Test+”匹配 Test 类及其子类。下面有一些例子：

```
@Pointcut("execution(* *.MyMethod1(..))")
public void Pointcut1() { }

@Pointcut("execution(* aop..MyMethod2(..))")
public void Pointcut2() { }

@Pointcut("execution(* Test+.*(..))")
public void Pointcut3() { }
```

其中“execution(* *.MyMethod1(..))”匹配所有类中的名字为 MyMethod1 的方法，参数任意；“execution(* aop..MyMethod2(..))”匹配 aop 包及其子包中的所有类中名字为 MyMethod2 的方法，参数任意；“execution(* Test+.*(..))”匹配所有 Test 类及其子类中的所有方法，参数任意。

组合使用切入点

切入点表达式支持逻辑运算符“&”、“||”和“!”，他们分别对应 java 的逻辑运算符与、或和非，仍然遵循 java 语义。有时候，我们还可以使用括号()来确定他们各个组合之间的优先级。例如：

```
@Before("Before1() || Before2() & Before3()")
public void BeforeAdvice1() {
    //处理逻辑
}

@Before("(Before1() || Before2()) & Before3()")
public void BeforeAdvice2() {
    //处理逻辑
}
```

注意：这些逻辑运算符的优先顺序也遵循 java 语义。

4、 切入点表达式关键词

前面一节，我们已经介绍了 execution 切入点表达式的关键词。下面将继续介绍它支持的其它关键词的用法。

➤ **within** 用于匹配连接点所在的 java 类或者包。也就是说，在它定义的某个包及其子包中的所有类的方法都会被匹配。如：

```
@Pointcut("within(aop.test.TestBean)")
public void Pointcut1() {}

@Pointcut("within(aop..*)")
public void Pointcut2() {}
```

Pointcut1()切入点匹配 aop.test.TestBean 类中的所有方法；Pointcut2()切入点匹配 aop 包中所有类的所有方法。

- **this** 利用该关键词可以向通知方法中传入代理对象的引用。用法范例如下：

```
@Before("Pointcut1() && this(proxy)")
public void BeforeTestBean(Object proxy) {
    //.....处理逻辑
}
```

- **target** 利用该关键词可以向通知方法中传入目标对象的引用。用法范例如下：

```
@Before("Pointcut1() && target(target)")
public void BeforeTestBean(Object target) {
    //.....处理逻辑
}
```

➤ **args** 在 Spring AOP 中，连接点实际上就是方法的调用，在调用某个方法时可能会给其传入参数，利用该关键词则可以把这些参数传入到通知中。用法范例如下：

```
@Before("Pointcut1() && target(target)")
public void BeforeTestBean(Object target) {
    //.....处理逻辑
}
```

➤ **@within** 它用于匹配在类一级使用了参数确定的注解的类，其所有方法都将被匹配。例如：

```
@Before("@within(aop.test.MyPointcut)")
public void BeforeTestBean() {
    //.....处理逻辑
}
```

在这个例子中，假设 aop.test.MyPointcut 是一个已经被声明的合法 Annotation 注解类，那么所有被@MyPointcut 标注的类都将匹配，其中的方法将被织入该 Before 通知。

➤ **@target** 和@within 的功能类似，但是必需要指定注解接口的保留策略为 RUNTIME，也就是说，它指定的 Annotation 注解接口必须用

@Retention(RetentionPolicy.RUNTIME)注解标注。例如：

```
package aop.test;

//import省略
@Retention(RetentionPolicy.RUNTIME)
public @interface MyPointcut {}
```

```
@Before("@target(aop.test.MyPointcut)")
public void BeforeTestBean2() {
    //.....处理逻辑
}
```

➤ **@args** 传入连接点的对象对应的java类型必须被@args指定的Annotation注解标注。比如：

```
@Before("@args(aop.test.MyPointcut)")
public void BeforeTestBean2() {
    //.....处理逻辑
}
```

目标对象对应的类中有如下的方法：

```
public return_type name(MyClass object) {
    //.....处理逻辑
}
```

那么当且仅当 MyClass 类必须被@MyPointcut 注解时该方法该通知才会被触发。

➤ **@annotation** 它由于匹配连接点被它参数指定的 Annotation 注解的方法。如下，被@MyPointcut 注解的方法调用时，该装备就会被触发：

```
@Before("@annotation(aop.test.MyPointcut)")
public void BeforeTestBean2() {
    //.....处理逻辑
}
```

➤ **bean** Spring2.5 新增的一个关键词，它可以通过受管 Bean 的名字来限定连接点所在的 Bean。用法如下：

```
@Pointcut("bean(*DAO)")
public void pointcut() {
}
```

其中“*”是通配符，它表示所有名称以 DAO 结尾的 Bean 都将被匹配。同其它的关键

词一样，它也可以通过“&&”、“||”等逻辑连接符和其它的表达式组合使用。

注意：该关键词是 Spring2.5 新增的，基于 AspectJ 扩展而来，而实际的 AspectJ 5 框架并不支持该关键词。

4.7.4 使用通知

在传统的 Spring AOP 中，我们通过实现特定的接口来定义一个通知，而采用 AspectJ 风格的 AOP，我们可以使用 Annotation 注解来定义一个通知。Spring 为各种通知都提供了相应的注解：前置通知@Before、后置通知@After、返回后通知@AfterReturning、异常通知@AfterThrowing 和环绕通知@Around。我们可以在一个切面中定义一个通知，也可以定义多个通知，不论类型是否相同，在合适的时机它都会被触发。


1、前置通知（@Before）

前置通知在连接点方法被调用之前触发，它可以用@Before 注解声明。就像前面介绍的一样，前置通知可以引用一个已经声明的切入点，也可以使用一个 in-place 的切入点表达式。例如：

```
@Pointcut("within(aop..*)")
public void Pointcut1() {}

@Before("Pointcut1()")
public void BeforeAdvice1() {
    //处理逻辑
}

@Before("within(aop..*)")
public void BeforeAdvice2() {
    //处理逻辑
}
```



在实际应用中，我们有时候还需要在前置通知中得到目标对象、连接点的一些信息，这时候，我们可以通过给通知方法中传入一些参数来完成这些功能。

下的例子讨论了如何在 Before 通知中访问连接点（例程 4.13）。为了一致性，该例程将使用第三章介绍的 Annotation 注解来驱动 Spring IoC。

新建 AOP_Test4.13 工程，添加 Spring 开发库，新建 aop.test 包。

编写 xml 配置文件，在其中启用 Annotation 驱动的 Spring IoC 和@AspectJ 风格的 AOP，代码如下：


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <aop:aspectj-autoproxy proxy-target-class="true" />
    <context:component-scan base-package="aop.test" />

</beans>
```

创建目标对象对应的类 TestBean，添加有参数的 MyMethod1() 方法和无参数的 MyMethod2() 方法，代码如下：

```
package aop.test;

import org.springframework.stereotype.Component;

/**
 * 该类将作为目标对象对应的类。
 * "@Component("TestBean")" 标注将该类注册为一个受管Bean，名字为"TestBean"。
 * @author zhangyong
 * */
@Component("TestBean")
public class TestBean{

    public void MyMethod1(String str) {
        System.out.println(this.getClass().getName()
            + ".MyMethod1() is run! the arg is:"+str);
    }

    public void MyMethod2() {
        System.out.println(this.getClass().getName()
            + ".MyMethod2() is run!");
    }

}
```

创建一个MyAspect类，分别用@Aspect和@Component注解把他声明为一个切面和受管Bean，再声明一个切入点三个前置通知。在前置通知BeforeAdvice1()中，使用args绑定连接点参数；在前置通知BeforeAdvice2()中利用this和target分别绑定了连接点代理实例和目标对象；在前置通知BeforeAdvice3()中通过JoinPoint参数访问连接点。具体代码如下：

```
package aop.test;

//import省略
/**
 *"@Aspect"声明一个切面, @Component注册受管Bean
 */
@Aspect// 声明一个切面
@Component// 将该类注册为一个受管Bean
public class MyAspect{

    /**
     * "@Pointcut"注解声明了一个切入点, 它匹配了aop.test.TestBean 类中的所有方法。
     */
    @Pointcut("execution(* aop.test.TestBean.*(..))")
    public void pointcut() {
    }

    /**
     * 该通知利用args绑定了连接点的参数
     */
    @Before("pointcut() && args(str)")
    public void BeforeAdvice1(String str) {
        System.out.println("Before通知1: 参数:" + str);
    }

    /**
     * 该通知利用this和target分别绑定了连接点代理实例和目标对象
     */
    @Before("pointcut() && this(p) && target(target)")
    public void BeforeAdvice2(Object p, Object target) {
        System.out.println("Before通知2: " + "代理对象:" + p + ", 目标对象: " + target);
    }

    /**
     * 通过JoinPoint参数访问连接点
     */
    @Before("pointcut()")
    public void BeforeAdvice3(JoinPoint point) {
        System.out.print("Before通知3: ");
        System.out.println("详细信息: " + point.toLongString());
    }
}
```

创建含有 `main()` 方法的主类 `TestMain`，代码如下：

```
package aop.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestMain {

    public static void main(String[] args) {

        // 实例化Spring IoC容器
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");
        // 获取受管Bean的实例
        TestBean beanA = (TestBean) ac.getBean("TestBean");
        beanA.MyMethod1("传入的参数值");
        beanA.MyMethod2();
    }
}
```

运行主类，输出结果如下：

```
Before通知1: 参数:传入的参数值
Before通知2: 代理对象:aop.test.TestBean@ee3aa7, 目标对象: aop.test.TestBean@ee3aa7
Before通知3: 详细信息: org.springframework.aop.aspectj.MethodInvocationProceedingJoinPoint: execution: [ReflectiveMethod
aop.test.TestBean.MyMethod1() is run! the arg is:传入的参数值
Before通知2: 代理对象:aop.test.TestBean@ee3aa7, 目标对象: aop.test.TestBean@ee3aa7
Before通知3: 详细信息: org.springframework.aop.aspectj.MethodInvocationProceedingJoinPoint: execution: [ReflectiveMethod
aop.test.TestBean.MyMethod2() is run!
```

图 4.15 例程 4.13 输出结果

可见，使用 `Annotation` 注解驱动的 `Spring` 配置，使我们配置文件简洁了许多，也在一定程度上的降低了我们的工作量。

值得注意的是，我们在前置通知 `BeforeAdvice3()` 中通过 `org.aspectj.lang.JoinPoint` 类型的参数访问连接点信息。`JoinPoint` 是 `AspectJ` 中定义的一个接口，它可以用于除了环绕通知外的其他任何通知，但是必须保证该类型的参数必须是通知方法的第一个参数。环绕通知使用继承于 `JoinPoint` 的 `org.aspectj.lang.ProceedingJoinPoint` 接口。

`JoinPoint` 描述了 `@AspectJ` 连接点，我们可以通过它访问到连接点的相关信息。它提供了很多有用的方法，介绍如下：

表 4.1 `JoinPoint` 接口的方法说明

方法声明	方法说明
<code>String toString();</code>	返回连接点的相关信息
<code>String toShortString();</code>	返回连接点的简短信息
<code>String toLongString();</code>	返回连接点的详细信息
<code>Object getThis();</code>	返回代理对象
<code>Object getTarget();</code>	返回目标对象
<code>Object[] getArgs();</code>	返回连接点方法的参数
<code>Signature getSignature();</code>	返回正在被通知的方法的相关信息

2、 后置通知（@After）

后置通知在一个连接点返回后被触发。与返回后通知不同的是，无论连接点以何种方式返回，无论连接点是否抛出了异常，后置通知都会被触发。例如：

```
@After("execution(* *.MyMethod(..)")  
public void AfterAdvice() {  
    //处理逻辑  
}
```

当然，我们也可以通过 `JoinPoint` 类型的参数访问连接点：

```
@After("pointcut()")  
public void AfterAdvice(JoinPoint point) {  
    //处理逻辑  
}
```

3、 返回后通知（@AfterReturning）

返回后通知和后置通知一样，都是在连接点返回后触发，与后置通知不同的是如果连接点非正常返回，如抛出异常，那么该通知将不被触发。它用 `@AfterReturning` 注解来声明：

```
@AfterReturning("execution(* *.MyMethod(..)")  
public void AfterReturningAdvice() {  
    //处理逻辑  
}
```

有时候，需要在返回后通知方法中使用连接点方法的返回值，我们可以使用 `@AfterReturning` 注解的 `returning` 属性来绑定连接点方法的返回值。示例如下：

```
@AfterReturning(pointcut = "pointcut()", returning = "re")
public void AfterReturning(Object re) {
    // 处理逻辑
}
```

利用 `returning` 绑定后，通知方法中必须要有一个名字和绑定的名字相同的参数，当连接点方法执行返回后，返回值会被该参数传入到通知方法中。和前面的 Spring1.x 中 Spring AOP 一样，虽然该通知在连接点方法返回给调用者之前触发，但是它仍然不能影响连接点方法的返回值。

4、 异常通知（@AfterThrowing）

异常通知将在连接点抛出异常时被触发。使用 `@AfterThrowing` 注解可以声明一个异常通知：

```
@AfterThrowing("pointcut()")
public void afterThrowsAdvice() {
    // 处理逻辑
}
```

有时候，我们只是需要对特定的异常进行处理、在通知体内得到被抛出的异常，那么你可以使用 `@AfterThrowing` 注解的 `throwing` 属性：

```
@AfterThrowing(pointcut="pointcut()", throwing="throwing")
public void afterThrowsAdvice(Throwable throwing) {
    // 处理逻辑
}
```

和返回后注解 `@AfterReturning` 的 `returning` 属性类似，我们还必须要在通知方法中定义一个名字和绑定异常名字相同的参数，连接点中抛出异常后，该异常将作为参数传入通知方法。当然，只有抛出的异常和异常参数类型一致时，该通知才会被触发。比如有如下通知：

```
@AfterThrowing(pointcut="pointcut()",throwing="throwing")
public void afterThrowsAdvice1(Throwable throwing) {
    // 处理逻辑
}

@AfterThrowing(pointcut="pointcut()",throwing="throwing")
public void afterThrowsAdvice2(RuntimeException throwing) {
    // 处理逻辑
}

@AfterThrowing(pointcut="pointcut()",throwing="throwing")
public void afterThrowsAdvice3(SQLException throwing) {
    // 处理逻辑
}
```

当连接点抛出 `RuntimeException` 异常时，`afterThrowsAdvice2` 和 `afterThrowsAdvice1` 通知会被触发；当抛出 `SQLException` 类型的异常时 `afterThrowsAdvice3` 和 `afterThrowsAdvice1` 通知会被触发。与 Spring1.x 中 AOP 不同的是在连接点中抛出任何异常时，`afterThrowsAdvice1` 通知都会被触发，因为其它异常都是 `Throwable` 的子类。

另外，如果异常通知也抛出一个异常，那么连接点原来抛出的异常将被覆盖。

下面用一个例程来讨论 `@AspectJ` 异常通知的使用。

新建 java 工程 AOP_Test4.14，添加 Spring 开发库，创建 `aop.test` 包。

修改 `applicationContext.xml` 配置文件，启用 `@AspectJ` 支持和 Spring IoC 的 Annotation 注解支持。代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">
    <aop:aspectj-autoproxy proxy-target-class="true" />
    <context:component-scan base-package="aop.test" />
</beans>
```

创建目标对象对应的类 `TestBean`，并利用 `@Component` 注解把它注册为一个 Bean，全

部代码如下：

```
package aop.test;

import java.sql.SQLException;

import org.springframework.stereotype.Component;

/**
 *
 * 该类将作为目标对象对应的类。
 * "@Component("TestBean")"标注将该类注册为一个受管Bean，名字为"TestBean"。
 * @author zhangyong
 * */
@Component("TestBean")
public class TestBean{

    public void MyMethod1(String str) throws SQLException{
        System.out.println(this.getClass().getName()
            + ".MyMethod1() is run! the arg is:"+str);
        throw new SQLException("RemoteException!");
    }

    public void MyMethod2() throws Exception {
        System.out.println(this.getClass().getName()
            + ".MyMethod2() is run!");
        throw new Exception("一个Exception异常!");
    }
}
```

创建切面类 MyAspect，声明一个切入点和三个通知：


```
package aop.test;

import java.sql.SQLException;

import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

/**
 * “@Aspect”声明一个切面
 */
@Aspect
// 声明一个切面
@Component
// 将该类注册为一个受管Bean
public class MyAspect{

    /**
     * “@Pointcut”注解声明了一个切入点，它匹配了aop.test.TestBean 类中的所有方法。
     */
    @Pointcut("execution(* aop.test.TestBean.*(..))")
    public void pointcut() {
    }

    /**
     * 拦截Throwable类型的异常
     */
    @AfterThrowing(pointcut = "pointcut()", throwing = "throwing")
    public void afterThrowsAdvice1(Throwable throwing) {
        System.out.println("afterThrowsAdvice1(Throwable)通知:"
            + throwing.toString());
    }

    /**
     * 截SQLException类型的异常
     */
    @AfterThrowing(pointcut = "pointcut()", throwing = "throwing")
    public void afterThrowsAdvice2(SQLException throwing) {
        System.out.println("afterThrowsAdvice2(SQLException)通知:"
            + throwing.toString());
    }
}
```

创建含有 main 方法的测试类，代码如下：

```
package aop.test;

// import省略
public class TestMain {

    public static void main(String[] args) {

        // 实例化Spring IoC容器
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");
        // 获取受管Bean的实例
        TestBean beanA = (TestBean) ac.getBean("TestBean");

        try {
            beanA.MyMethod1("传入的参数值");
        } catch (SQLException e) {
            System.out.println("调用者:" + e);
        }
        System.out.print("\n");

        try {
            beanA.MyMethod2();
        } catch (Exception e) {
            System.out.println("调用者:" + e);
        }
    }
}
```

运行主类，控制台视图输出如下：

```
aop.test.TestBean.MyMethod1() is run! the arg is:传入的参数值
afterThrowsAdvice1(Throwables)通知:java.sql.SQLException: RemoteException!
afterThrowsAdvice2(SQLException)通知:java.sql.SQLException: RemoteException!
调用者:java.sql.SQLException: RemoteException!

aop.test.TestBean.MyMethod2() is run!
afterThrowsAdvice1(Throwables)通知:java.lang.Exception: 一个Exception异常!
调用者:java.lang.Exception: 一个Exception异常!
```

图 4.16 例程 4.14 输出结果

5、 环绕通知 (@Around)

环绕通知用@Around 注解声明。通知方法的第一个参数必须是 ProceedingJoinPoint 类型。和 Spring1.x 中的环绕通知类似，如果为了保证连接点的执行，则需要在通知体内调用

ProceedingJoinPoint 参数的 proceed() 方法。如果不对连接点的返回值进行处理，则要把 proceed() 方法的返回值返回给调用者。如：

```
@Around("pointcut()")
public Object AroundAdvice(ProceedingJoinPoint point) throws Throwable {
    // 处理逻辑
    Object o = point.proceed(args); //调用连接点方法
    // 处理逻辑
    return o; // 把连接点的返回值返回给调用者
}
```

ProceedingJoinPoint 接口的其它成员方法的用法和 JoinPoint 接口一样，见表 4.1。

4.8 基于 aop 命名空间的 AOP

在某些时候，我们工程中使用的 JDK 不一定是 1.5 以上，也就是说可能不支持 Annotation 注解，这时自然也就不能使用 @AspectJ 注解驱动的 AOP 了，那么如果我们仍然想使用 AspectJ 灵活的切入点表达式，那么该如何呢？Spring 为我们提供了基于 xml schematic 的 aop 命名空间，它的使用方式和 @AspectJ 注解类似，不同的是配置信息从注解中转移到了 Spring 配置文件中。在这一节，我们将详细介绍如何使用 Spring 提供的 <aop:config/> 标签来配置 Spring AOP。

4.8.1 一点准备工作和一个例子

要使用 <aop:config/> 标签，需要给 Spring 配置文件中引入基于 xml schema 的 Spring AOP 命名空间。完成后的 Spring 配置文件如下（在该节，所有例程的配置文件中添加了 Spring AOP 命名空间，除非特殊情况外，为了节约空间，这部分将在给出的代码中省略），粗体内容即为我们需要添加的内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-2.5.xsd >
    ..... Spring 配置信息
</beans>
```

关于 aop 命名空间的标签，我们前面使用过的有 <aop:aspectj-autoproxy/>，在这一节，

我们将以<aop:config/>标签作为重点。事实上，我们在这一节介绍的所有标签都是该标签的子标签。

下面有一个例程来直观的展示如何使用<aop:config/>标签来配置 Spring AOP（完整代码见例程 4.15）。在例子中，我们使用<aop:config/>配置一个切面并拦截目标对象 Peoples 的 SayHello()方法，在它执行前输出提示信息。

首先创建工程 AOP_Test4.15，添加 Spring IoC 和 Spring AOP 库后，创建 aop.test 包，新建目标类 People，代码如下：

```
package aop.test;

/**
 * 该类将作为目标对象对应的类。
 * @author zhangyong
 * */
public class People{

    public String SayHello(String str){
        System.out.println(this.getClass().getName() + "说: " + str);
        return str;
    }
}
```

修改 Spring xml 配置文件，将该类注册为一个受管 Bean：

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

创建含有 main()方法的测试类 TestMain，从 Spring IoC 容器中获取 Peoples 对象，并调用其 SayHello()方法，代码如下：

```
package aop.test;

// import省略
public class TestMain {
    public static void main(String[] args) {
        // 实例化Spring IoC容器
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");
        // 获取受管Bean的实例
        People p = (People) ac.getBean("TestBean");
        p.SayHello("传入的参数值");
    }
}
```

创建 MyAspect 类，添加一个 beforeAdvice()方法作为前置通知方法，代码如下：

```
package aop.test;

import org.aspectj.lang.JoinPoint;

public class MyAspect {

    public void beforeAdvice(JoinPoint point) {
        System.out.println("前置通知被触发: " +
            point.getTarget().getClass().getName() +
            "将要" + point.getSignature().getName());
    }
}
```

修改 xml 配置文件，为其添加 aop 命名空间，并把 MyAspect 注册为一个受管 Bean，作为我们下面定义切面的 backing bean。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <bean id="MyAspect" class="aop.test.MyAspect" />
    <bean id="TestBean" class="aop.test.People" />

    <aop:config proxy-target-class="true">
        <aop:aspect ref="MyAspect" order="0" id="Test">
            <aop:pointcut id="testPointcut"
                expression="execution(* aop..*(..))" />
            <aop:before pointcut-ref="testPointcut"
                method="beforeAdvice" />
        </aop:aspect>
    </aop:config>
</beans>
```

声明一个切面。

声明一个切入点。

声明一个通知。

运行主类，输出如下：

```
前置通知被触发: aop.test.People将要SayHello
aop.test.People说: 传入的参数值
```

图 4.17 例程 4.15 输出结果

4.8.2 声明一个切面

在基于 AOP 命名空间的 Spring AOP 中，要声明一个切面，需要使用<aop:config/>的子标签<aop:aspect>。<aop:aspect>标签有一个 ref 属性必须被赋值，它用于指定和该切面关联的受管 Bean（backing bean，以后我们都将使用 Backing Bean 来称呼这样的 Bean）。正如下例所示，该 Bean 对应的 java 类是一个普通的 java 类，在该类中定义了切面的通知方法。此外，<aop:aspect>标签还有两个可选的 order 属性和 id 属性，order 属性用于指定该切面的加载顺序，id 属性用于标识该切面。范例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans .....>
  <bean id="MyAspect" class="aop.test.MyAspect"/>
  <aop:config proxy-target-class="true">
    <aop:aspect ref="MyAspect" order="1" id="TestAspectName">
      .....切面其他配置
    </aop:aspect>
  </aop:config>
  .....其他配置
</beans>
```

配置切面所需要的 Bean

声明切面，并指定加载顺序。

4.8.3 声明一个切入点

要声明一个切入点，可以使用<aop:aspect>的子标签<aop:pointcut>，在 Spring2.5 中它有两个属性 id 和 expression，分别用于标示该切入点和设定该切入点表达式。例如：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans .....>
  <bean id="MyAspect" class="aop.test.MyAspect"/>
  <aop:config proxy-target-class="true">
    <aop:aspect ref="MyAspect" order="1" id="TestAspectName">
      <aop:pointcut id="test"
        expression="execution(* aop.test.TestBean.*(..))" />
      <aop:before pointcut="aop.test.MyAspect.Pointcut1()"
        method="beforeAdvice" />
    </aop:aspect>
  </aop:config>
  .....其他配置
</beans>
```

声明切入点

<aop:pointcut>标签的 expression 属性使用前面介绍的切入点表达式语言，也就是说支持 AspectJ 切入点表达式。但是由于 xml 对“&&”、“||”、“!”等逻辑运算符不友好，@AspectJ

切入点表达式语言中使用的这些逻辑运算符在 xml 配置中需要分别用 “and”、“or” 和 “not” 来代替。

有时候，我们也需要在 xml 中使用@Pointcut 注解声明的切入点，那么该如何呢？大家可能记得，我们可以在切入点表达式中可以引用另一个切入点。对了，就在这里，我们使用该特性可以完成这个任务，如下：

```
<aop:pointcut id="test" expression="aop.test.MyAspect.Pointcut1()" />
```

注意：这里我们必须使用全路径来标示引用的切入点。

4.8.4 声明一个通知

和@AspectJ 一样，基于 AOP 命名空间的配置也可以定义五种通知类型，并且使用方式和特性类似。与@AspectJ 不同的是，配置信息从 Annotation 中转移到了 xml 配置文件。

1、前置通知

声明一个前置通知可以使用<aop:aspect>的子标签<aop:before/>。该标签的属性说明如下表：

表 4.2 <aop:before/>标签的属性说明

属性	说明
pointcut	指定该通知的内置切入点
pointcut-ref	通过 id 引用已定义的切入点
method	指定通知对应的方法，该方法必须已在切面的 backing bean 中被声明
arg-names	通过方法的参数名字来匹配切入点参数

对于一个通知来说，切入点和对应的通知方法是必须的。也就是说，在这些属性中，method 属性是必须的，我们必须要给通知指定一个对应的方法；pointcut 属性和 pointcut-ref 必须有一个被指定，以此确定该通知的切入点。范例如下：

```
<aop:aspect ref="MyAspect" order="0" id="Test">
  <aop:pointcut id="testPointcut"
    expression="execution(* aop.test.TestBean.*(..))" />
  <aop:before pointcut-ref="testPointcut" method="beforeAdvice" />
</aop:aspect>
```

2、后置通知

声明一个后置通知使用<aop:after/>标签，它的属性等和<aop:before/>标签类似，下面是范例：

```
<aop:aspect ref="MyAspect" order="0" id="Test">
    <aop:pointcut id="testPointcut"
        expression="execution(* aop.test.TestBean.*(..))" />
    <aop:after pointcut-ref="testPointcut" method="AfterAdvice"/>
</aop:aspect>
```

3、 返回后通知

<aop:after-returning/>标签可以声明一个返回后通知，该标签的属性和<aop:before/>相比它多了一个 `returning` 属性。该属性的意义类似于 `@AfterReturning` 注解的 `returning` 属性，用于将链接点的返回值传给通知方法。用法如下：

```
<aop:aspect ref="MyAspect" order="0" id="Test">
    <aop:pointcut id="testPointcut"
        expression="execution(* aop.test.TestBean.*(..))" />
    <aop:after-returning pointcut-ref="testPointcut"
        method="AfterReturnAdvice" returning="reVlue" />
</aop:aspect>
```

4、 异常通知

声明一个异常通知使用<aop:after-throwing />标签，它有一个类似于 `throwing` 属性又来指定该通知匹配的异常类型。用法如下：

```
<aop:aspect ref="MyAspect" order="0" id="Test">
    <aop:pointcut id="testPointcut"
        expression="execution(* aop.test.TestBean.*(..))" />
    <aop:after-throwing pointcut-ref="testPointcut"
        method="afterThrowingAdvice" throwing="throwable" />
</aop:aspect>
```

5、 环绕通知

环绕通知是所有通知中功能最强大的通知，用<aop:around/>标签来声明。用法如下：

```
<aop:aspect ref="MyAspect" order="0" id="Test">
    <aop:pointcut id="testPointcut"
        expression="execution(* aop.test.TestBean.*(..))" />
    <aop:around pointcut-ref="testPointcut"
        method="aroundAdvice"/>
```

6、 一个例子

下面有一个例子详细的展示 AOP 命名空间的 AOP 的使用（例程 4.16）。

目标对象对应的类 `People`：


```
package aop.test;

/**
 * 该类将作为目标对象对应的类。
 * @author zhangyong
 * */
public class People{

    public String SayHello(String str){
        System.out.println(this.getClass().getName()
            + "说: "+str);
        return str;
    }

    public String SayBye(String str){
        System.out.println(this.getClass().getName()
            + "说: "+str);
        return str;
    }

    public void Run() throws Exception {
        System.out.println(this.getClass().getName()
            + "is Run()!");
        throw new Exception("在Run时发生了一个异常!");
    }
}
```

切面 MyAspect 类:

```
package aop.test;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;

/**
 * 该类将作为切面的backing Bean
 * @author zhangyong
 * */
public class MyAspect {

    /**
     * 前置通知，使用point参数获取连接点信息
     * */
    public void beforeAdvice(JoinPoint point) {
        System.out.println("前置通知被触发: " +
            point.getTarget().getClass().getName() + "将要" +
            point.getSignature().getName());
    }

    /**
     * 返回后通知，使用reValue参数获取连接点返回值
     * */
    public void afterReturningAdvice(Object reValue) {
        System.out.println("返回后通知被触发: " + reValue);
    }

    /**
     * 环绕通知
     * */
    public Object aroundAdvice(ProceedingJoinPoint point) throws Throwable
    {
        System.out.println("环绕通知通知被触发: " + point);
        return point.proceed(); //执行连接点
    }

    /**
     * 异常通知，使用throwing获取连接点异常
     * */
    public void afterThrowingAdvice(Throwable throwing) throws Throwable {
        System.out.println("异常通知通知被触发: " + throwing);
    }
}
```

Xml 配置文件:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans .....>
    <bean id="MyAspect" class="aop.test.MyAspect" />
    <bean id="TestBean" class="aop.test.People" />
    <aop:config proxy-target-class="true">
        <aop:aspect ref="MyAspect" order="0" id="Test">
            <aop:pointcut id="testPointcut"
                expression="execution(* aop..*(..))" />
            <aop:before pointcut-ref="testPointcut"
                method="beforeAdvice" />
            <aop:after-returning method="afterReturningAdvice"
                pointcut-ref="testPointcut" returning="reValue" />
            <aop:around method="aroundAdvice"
                pointcut-ref="testPointcut" />
            <aop:after-throwing method="afterThrowingAdvice"
                pointcut-ref="testPointcut" throwing="throwing" />
        </aop:aspect>
    </aop:config>
</beans>
```

创建主类:

```
package aop.test;

// import省略
public class TestMain {
    public static void main(String[] args) {
        // 实例化Spring IoC容器
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");
        // 获取受管Bean的实例
        People p = (People) ac.getBean("TestBean");
        p.SayHello("hello!");
        System.out.print("\n");
        p.SayBye("Byebye! ");
        System.out.print("\n");
        try {
            p.Run();
        } catch (Exception e) {
            // TODO Auto-generated catch block
        }
    }
}
```

运行结果:

```

前置通知被触发: aop.test.People将要SayHello
环绕通知通知被触发: org.springframework.aop.aspectj.MethodInvocationProceedingJoinPoint: execution(SayHello)
aop.test.People说: hello!
返回后通知被触发: hello!

前置通知被触发: aop.test.People将要SayBye
环绕通知通知被触发: org.springframework.aop.aspectj.MethodInvocationProceedingJoinPoint: execution(SayBye)
aop.test.People说: Byebye!
返回后通知被触发: Byebye!

前置通知被触发: aop.test.People将要Run
环绕通知通知被触发: org.springframework.aop.aspectj.MethodInvocationProceedingJoinPoint: execution(Run)
aop.test.Peopleis Run()!
异常通知通知被触发: java.lang.Exception: 在Run时发生了一个异常!

```

图 4.18 例程 4.16 输出结果

4.9 小结

从 Spring2.5 开始, Spring 全面拥抱了 Java EE 5, 支持 Annotation 注解驱动的配置。无论是上一张提到的 Spring IoC、这一章讨论的 AOP, 还是以后介绍的 Spring 内置的 MVC 框架, 都支持 Annotation 注解驱动的配置方式。Spring 从 2.0 开始就支持@AspectJ 风格的 AOP 配置方式, 它完美的集成了 AspectJ 5 的很多方面, 借鉴了 AspectJ 5 的很多优点, 同时又根据自身的特点加入了些许改进。特别是 pointcut 表达式, Spring 提供了近乎完全的支持, 使得在 Spring AOP 中也可以用上 AspectJ 灵活的切入点表达式语言。不仅如此, Spring 还提供了 AOP 命名空间和@AspectJ 互相兼容的可能, 即是说, 使用@Pointcut 注解声明的切入点和<aop:pointcut/>标签声明的切入点可以相互引用。

本章详细的介绍了 Spring AOP 的使用, 从 Spring1.2 的 xml 配置方式, 到 Spring2.X 支持的 Annotation 注解驱动的 AOP 和 AOP 命名空间的配置方式, 我们都进行了详细的讨论, 到这里, 相信大家已经对 Spring AOP 有了大致的了解。

4.10 参考文献和推荐资料:

- [1] 周玲, 文红民.AOP 简介及其 Spring 中的实现.科技广场: 2005-12, P70
- [2] Spring 团队.Spring reference
- [3] aspectj 官方网: <http://www.eclipse.org/aspectj/> 提供了 AspectJ 的完整资料。
- [4] Sing Li.AOP 介绍. [IBM 中国网站](#):
<https://www6.software.ibm.com/developerworks/cn/education/java/j-aopintro/tutorial/index.html>
- [5] Adrian Colyer . 介绍 AspectJ 5.IBM [中国网站](#):
<http://www.ibm.com/developerworks/cn/java/j-aopwork8/>
- [6] Mik Kersten. AOP 工具的比较. IBM [中国网站](#),
<http://www.ibm.com/developerworks/cn/java/j-aopwork1/>
<http://www.ibm.com/developerworks/cn/java/j-aopwork2/>

第五章 数据持久化

5.1 概述

如今，数据库技术已经深入到各个领域。在大多数企业应用中，数据库系统已经成为其重要的组成部分，甚至是核心内容。

关系数据库系统是当前数据库中的主流产品。为了访问关系数据库，J2SE 为我们提供了相关 API，这就是 JDBC(Java Database Connectivity)规范。它作为一个桥梁，将关系数据库管理系统(Relational Database Management System, RDBMS)和应用程序连接起来，使我们可以在 java 应用中通过结构化查询语言 (SQL) 访问到 RDBMS 管理的数据，和数据库交互。在 java 领域，JDBC 成了访问 RDBMS 的标准，现在市场上流行的 O/R Mapping 工具，都是以 JDBC 为基础；同时在很多场合，开发者也有直接使用 JDBC API 和 RDBMS 交互的必要。所以了解并熟悉 JDBC 的使用就变得非常必要。但是 JDBC 的复杂性是众所周知的，使用 JDBC API，我们不得不和太多的与业务无关的代码打交道，编写大量的 JDBC 代码，这样引入 Bug 的几率也大大增加。

通常来说，我们使用类似于下面代码来和 RDBMS 交互（完整工程代码见例程 5.1）：

```
public static void main(String[] args) throws SQLException {
    String driver = "com.microsoft.jdbc.sqlserver.SQLServerDriver";
    String dbUrl =
        "jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=DB_Demo";
    String username = "sa";
    String password = "123456";
    Connection conn = null;
    Statement stmt = null;
    Resulted rs = null;
    try {
        Class.forName(driver); //加载驱动程序
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    try { //创建数据库连接
        conn = DriverManager.getConnection(dbUrl, username, password);
        stmt = conn.createStatement(); //创建Statement
        rs = stmt.executeQuery("Select * from admin"); //执行SQL语句
        while (rs.next()) { //输出结果
            System.out.println("id=" + rs.getString("id") + ",username="
                + rs.getString("username") + ",password="
                + rs.getString("password"));
        }
    } catch (SQLException e) {
        System.out.println("错误代码: "
            + e.getErrorCode() + "; 消息: " + e.getMessage());
    } finally {
        rs.close(); //依次释放资源
        stmt.close();
        conn.close();
    }
}
```

在这里我们已经使用了将近 30 行代码来完成了执行 “Select * from admin” 语句并把其结果打印出来这个看似简单的任务。我们还没有使用事务管理、存储过程等高级功能，否则，将会花费我们更多的精力来处理这些内容。实际上，JDBC API 访问数据库非常繁杂。

为了使开发者可以统一的、方便的访问数据库，Spring 提供了一整套的解决方案。无论是对 JDBC，还是对 hibernate、iBatis 等 O/R Mapping 工具，Spring 都提供了良好的集成支持。使用 Spring，开发者可以不必再去考虑 JDBC、hibernate 等的底层细节，Spring 提供的一系列的模板类和 DaoSupport 类帮助我们屏蔽了这些细节问题。使用 Spring，我们完全可以不必考虑数据访问技术的底层信息，从而使我们把大量的精力转移到业务数据的处理中

其中，`org.springframework.dao.support.DaoSupport.DaoSupport` 是一个抽象类，它是一个通用的基类，Spring 提供的所有的 `DaoSupport` 类都继承自该类，它提供了模板方法和 DAO 初始化方法。

`DaoSupport.DaoSupport` 类的扩展类提供了目前市场上流行的大多数数据访问技术的封装。我们可以利用这些类很方便的把这些技术和 Spring 集成，从而免去大量的重复劳动和 Bug 出现的几率。在本章，我们将详细讨论 `JdbcDaoSupport` 和 `HibernateDaoSupport` 等的使用。

5.3 Spring 对 JDBC 的封装

上面一节提到过，Spring 利用 `JdbcDaoSupport` 提供了对 JDBC 数据访问的封装，它给其子类提供了 `JdbcTemplate` 类的实例，这样，我们就可以在其子类中使用 `JdbcTemplate` 方便的利用 JDBC API 完成和 RDBMS 的交互、对数据库进行 CRUD 操作，而不必考虑 JDBC API 底层的技术细节和异常处理。

在 Spring 中提供了 JDBC 模板类 `org.springframework.jdbc.core.JdbcTemplate` 来实现了对 JDBC API 的封装。它提供了大量的方法来对 JDBC 进行简化和封装。

5.3.1 从一个例子开始

在本章的例子中，我们将重复的使用一个数据库 `DB_Demo`，其中有两张表，ER 图如下：

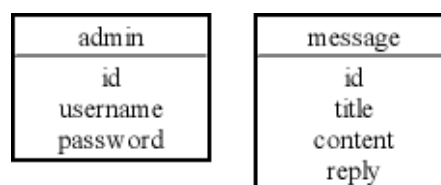


图 5.3 `DB_Demo` 数据库结构

这是一个留言板系统的数据库，其中只有两张表，结构非常简单，这样也正可以方便我们讨论。

在 SQL-Server2000 的查询分析器中执行下面的 SQL 脚本，创建数据库和表，并为表添加必要的测试数据。如下：

```
/*创建一个数据库*/
CREATE DATABASE DB_Demo
ON(
    NAME='DB_Demo',
    FILENAME='E:\JavaBook\Database\DB_Demo.mdf',
    SIZE = 1MB,
    FILEGROWTH=1%
)
LOG ON(
    NAME='DB_Demo_LOG',
    FILENAME='E:\JavaBook\Database\DB_Demo_LOG.ldf',
    SIZE = 1MB,
    FILEGROWTH=1%
)
GO

USE DB_Demo
GO

/*管理员表*/
CREATE TABLE admin(
    id int identity(1,1) primary key,
    username varchar(20) not null,
    password varchar(32) not null,
)
GO
INSERT INTO admin(username,password) VALUES('admin','admin888')
INSERT INTO admin(username,password) VALUES('cmzy','cmzy888')
INSERT INTO admin(username,password) VALUES('test','test888')

/*留言表*/
CREATE TABLE message (
    id int IDENTITY (1, 1) NOT NULL PRIMARY KEY ,
    title varchar (40) NOT NULL ,
    content varchar(600) NULL ,
    reply varchar(600) NULL
)
INSERT INTO message(title,content,reply) VALUES('留言1','留言1内容','留言1回复')
INSERT INTO message(title,content,reply) VALUES('留言2','留言2内容','留言2回复')
INSERT INTO message(title,content,reply) VALUES('留言3','留言3内容','留言3回复')
INSERT INTO message(title,content,reply) VALUES('留言4','留言4内容','留言4回复')
INSERT INTO message(title,content,reply) VALUES('留言5','留言5内容','留言5回复')
```

新建一个名字为 DB_Demo5.2 的工程，添加 SQL-Server2000 的 JDBC 驱动库和 Spring 开发库到工程的 Classpath 中，新一个 db.demo 包，创建 TestMain 类，在 main() 方法中使用 JdbcTemplate 模板来查询 admin 表并打印结果，代码如下：

```
package db.demo;

import java.sql.SQLException;
import java.util.List;

import org.springframework.jdbc.core.ColumnMapRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

public class TestMain {

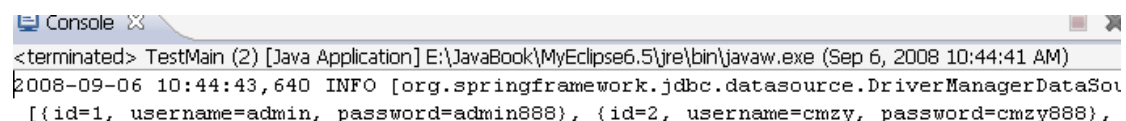
    public static void main(String[] args) throws SQLException {

        DriverManagerDataSource ds = new DriverManagerDataSource(
            "com.microsoft.jdbc.sqlserver.SQLServerDriver",
            "jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=DB_Demo",
            "sa", "123456");

        JdbcTemplate jt = new JdbcTemplate();
        jt.setDataSource(ds);

        List list = jt.query("select * from admin", new
        ColumnMapRowMapper());
        System.out.println(list);
    }
}
```

运行该类输出结果如下：



```
<terminated> TestMain (2) [Java Application] E:\JavaBook\MyEclipse6.5\jre\bin\javaw.exe (Sep 6, 2008 10:44:41 AM)
2008-09-06 10:44:43,640 INFO [org.springframework.jdbc.datasource.DriverManagerDataSource] [{id=1, username=admin, password=admin888}, {id=2, username=cmzy, password=cmzy888},
```

图 5.4 例程 5.2 运行结果

虽然这段代码还有不少问题，但是和例程 5.1 相比代码已经简洁了不少。我们只是使用 5 行代码即完成了和例程 5.1 同样的功能，而且也没用考虑处理异常等其它因素。

注意：在为工程添加 Spring 开发能力时，需要将 JDBC 支持勾选，如下：

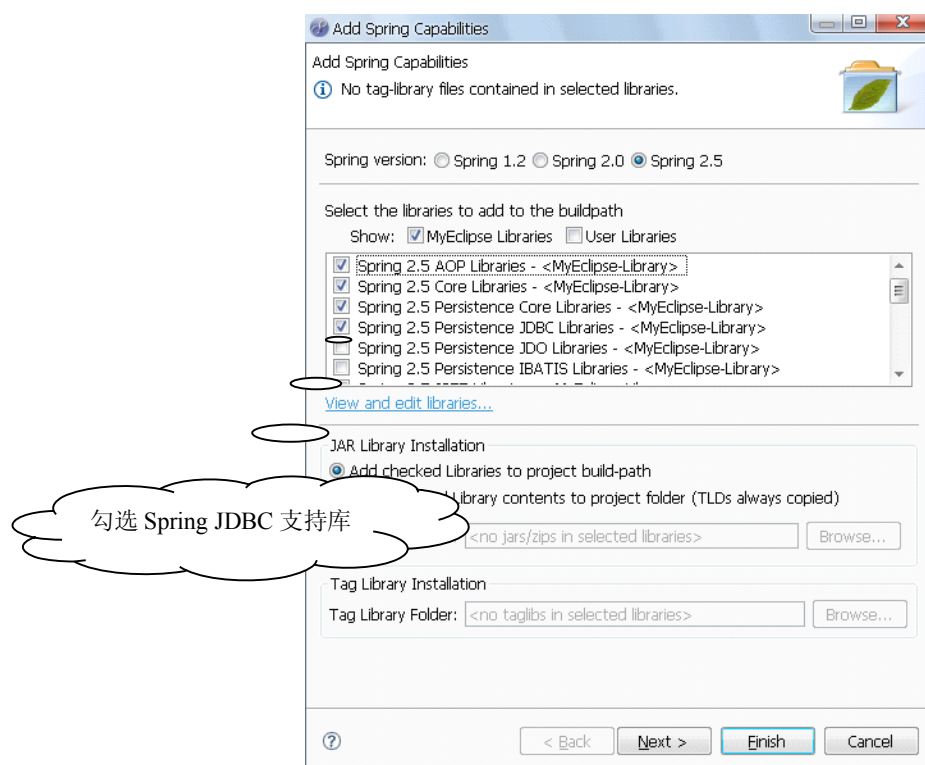


图 5.5 添加 Spring 库

上面代码的缺点显而易见，直接在业务代码中使用 `JdbcTemplate` 模板并不是一个好主意。可以看到在上面的代码中，我们还是把数据库的配置写在了程序代码中，而且每次操作数据库时都要 `new` 一个 `JdbcTemplate`。

Spring 为我们提供的 `DaoSupport` 类的子类 `JdbcDaoSupport`，通过扩展 `JdbcDaoSupport` 类，我们可以把数据库操作封装在其子类中，使用 Spring IoC 为其注入 `DataSource` 实例然后在需要的时候将该 DAO 类的实例注入依赖它的对象中。在下面的例子中，我们都将使用这种模式。

5.3.2 ConnectionCallback 回调接口和 StatementCallback 回调接口

有时候，`JdbcTemplate` 模板类提供的数据库操作方法不能满足我们的需要，那么这个时候，我们则可以使用 Spring 提供的如下两个方法：

public	Object	execute (ConnectionCallback	action)	throws
	DataSourceException			

通过它们，我们可以直接使用 `JdbcTemplate` 提供的 `Connection` 和 `Statement` 而不需要手工关闭和处理异常。

使用方法如下：

```
getJdbcTemplate().execute(new ConnectionCallback() {  
    public Object doInConnection(Connection con) throws SQLException,  
        DataAccessException {  
        //这里可以直接使用参数传入的con  
        return null;  
    }  
});
```

我们使用内部类声明了一个 ConnectionCallback 接口的实现类。在 doInConnection()回调方法中，我们可以实现自己的处理逻辑。所需要的 Connection 对象由参数传入，开发者不必理会 Connection 对象的关闭操作，JdbcTemplate 会自动的关闭 Connection 对象和处理异常，同时 doInConnection()回调方法的返回值也将作为 execute()方法的返回值返回给调用者。

```
getJdbcTemplate().execute(new StatementCallback() {  
    public Object doInStatement(Statement stmt) throws SQLException,  
        DataAccessException {  
        ResultSet rs = null; //准备ResultSet对象  
        try {  
            rs = stmt.executeQuery(sql);  
            //处理rs  
        } finally {  
            rs.close();  
        }  
        return null;  
    }  
});
```

使用 StatementCallback 回调接口，Spring 将会自动的把 Statement 对象传入到 doInStatement()回调方法中，在该方法中我们只需处理自己的逻辑而不必关注 Statement 对象的关闭操作和异常处理。与 ConnectionCallback 回调接口一样，doInStatement()回调方法的返回值也将作为 execute()方法的返回值返回给调用者。此外，在 Spring 的 JdbcTemplate 模板类内部也大量的使用该回调方法。

需要注意的是：在 doInStatement()方法中，我们需要手工关闭 ResultSet 对象。

5.3.3 使用静态 SQL 查询数据库

Spring 提供的 JdbcTemplate 模板类暴露了很多方法，方便我们执行 SQL 查询。JdbcTemplate 提供了如下方法用于执行静态 SQL：

```

public Object execute(StatementCallback action) throws
    DataAccessException

public void execute(final String sql) throws DataAccessException;

public Object query(final String sql, final ResultSetExtractor rse)
    throws DataAccessException;

public void query(String sql, RowCallbackHandler rch) throws
    DataAccessException;

public List query(String sql, RowMapper rowMapper) throws
    DataAccessException

public Map queryForMap(String sql) throws DataAccessException

public Object queryForObject(String sql, RowMapper rowMapper) throws
    DataAccessException

public Object queryForObject(String sql, Class requiredType) throws
    DataAccessException

public long queryForLong(String sql) throws DataAccessException

public int queryForInt(String sql) throws DataAccessException

public List queryForList(String sql, Class elementType) throws
    DataAccessException

public List queryForList(String sql) throws DataAccessException

public SqlRowSet queryForRowSet(String sql) throws DataAccessException

public int update(final String sql) throws DataAccessException

public int[] batchUpdate(final String[] sql) throws
    DataAccessException

```

如果使用 `StatementCallback` 回调接口，我们将必须手工关闭 `ResultSet`，和处理执行查询时可能抛出的异常。为了开发者避免忘记关闭 `ResultSet` 而带来的问题，我们可以使用 `ResultSetExtractor` 回调接口，在该接口提供的 `extractData()` 方法将会通过参数把准备好的 `ResultSet` 对象提供给我们，使用方式如下：

```
public List queryAll() {
    final String sql = "Select * from " + TABLE_NAME;

    return (List) getJdbcTemplate().query(sql, new ResultSetExtractor() {
        public Object extractData(final ResultSet rs) throws SQLException,
            DataAccessException {
            final List list = new ArrayList();
            while (rs.next()) {
                final Admin admin = new Admin();
                admin.setId(rs.getInt(ID));
                admin.setUsername(rs.getString(USERNAME));
                admin.setPassword(rs.getString(PASSWORD));
                list.add(admin);
            }
            return list;
        }
    });
}
```

注意：TABLE_NAME、ID、USERNAME、PASSWORD 为常量，它们的值分别为：admin、id、username、password。

在 extractData()方法中，我们对 ResultSet 对象进行迭代，并取出数据映射到 Admin 实体。很显然，在大量的方法中进行 while 迭代也是个繁杂重复性的工作，这里我们可以使用 RowCallbackHandler 回调接口，使用范例如下：

```
public List<Admin> findAll() {

    final List<Admin> adminList = new ArrayList<Admin>();
    String sql = "select * from " + TABLE_NAME;

    getJdbcTemplate().query(sql, new RowCallbackHandler() {
        // 使用RowCallbackHandler回调接口进行简单的OR映射
        public void processRow(ResultSet rs) throws SQLException {
            Admin admin = new Admin();
            admin.setId(rs.getInt(ID));
            admin.setUsername(rs.getString(USERNAME));
            admin.setPassword(rs.getString(PASSWORD));
            adminList.add(admin);
        }
    });
    return adminList;
}
```

此外，我们还可以使用 `RowMapper`，直接通过 `query()`方法返回 `List` 类型的查询结果。范例如下：

```
public List queryAll() {
    final String sql = "Select * from " + TABLE_NAME;

    return getJdbcTemplate().query(sql, new RowMapper() {
        public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
            Admin admin = new Admin();
            admin.setId(rs.getInt(ID));
            admin.setUsername(rs.getString(USERNAME));
            admin.setPassword(rs.getString(PASSWORD));
            return admin;
        }
    });
}
```

此外，`queryForInt()`也是一个重要的方法，它执行一个 `sql` 语句后将返回一个 `int` 类型的值。使用方法如下：

```
getJdbcTemplate().queryForInt("Select count(*) from " + TABLE_NAME);
```

这个例子将返回 `admin` 表中的行数。

5.3.4 ResultSetExtractor 和 RowMapper 接口的实现类

前面我们使用过 `ResultSetExtractor` 和 `RowMapper` 回调接口来实现简单的 O-R Mapping。实际上，`Spring` 内部为我们提供了很多它们的实现类，可以方便我们的操作。

`Spring` 提供了一个 `ResultSetExtractor` 接口的实现类 `RowMapperResultSetExtractor`，它可以将处理结果的任务交给 `RowMapper` 去完成。比如：

```
(List<Admin>) getJdbcTemplate().query(
    "SELECT * FROM " + TABLE_NAME,
    new RowMapperResultSetExtractor(new BeanPropertyRowMapper(
        Admin.class), 2));
```

`Spring` 提供了一些 `RowMapper` 接口的实现类：

1、 `BeanPropertyRowMapper`：它可以把 `ResultSet` 和实体类的字段进行实现自动映射。使用如下：


```
public List<Admin> queryAll() {  
    final String sql = "SELECT * FROM " + TABLE_NAME;  
    return getJdbcTemplate().query(sql,  
        new BeanPropertyRowMapper(Admin.class));  
}
```

2、 ColumnMapRowMapper: 当查询产生的 ResultSet 中有多列数据时, 可以使用该实现类。在例程 DB_Demo5.2 中我们使用这个实现类。

3、 ParameterizedBeanPropertyRowMapper: 继承自 ColumnMapRowMapper 类, 使用了 Java 5 泛型。使用例子如下:

```
public List<Admin> queryAll() {  
    final String sql = "SELECT * FROM " + TABLE_NAME;  
    return getJdbcTemplate().query(sql,  
        ParameterizedBeanPropertyRowMapper.newInstance(Admin.class));  
}
```

4、 SingleColumnRowMapper: 当查询结果为单列数据时, 它将直接返回查询结果。如:

```
public List queryAll() {  
    final String sql = "SELECT username FROM " + TABLE_NAME;  
    return getJdbcTemplate().query(sql,  
        new SingleColumnRowMapper());  
}
```

5、 ParameterizedSingleColumnRowMapper: 是 SingleColumnRowMapper 的子类, 使用了 Java 5 的泛型定义。

5.3.5 使用预编译语句

在使用 Spring 提供的 JdbcTemplate 类时, 我们还可以使用预编译 SQL 语句。数据查询非常频繁的情况下, 在 RDBMS 中使用预编译语句会显著的提高其查询性能。JDBC 提供 PreparedStatement 接口。通过与编译 PreparedStatement 对象, 我们可以使用预编译语句来提高查询效率, Spring 对 PreparedStatement 做了封装。

在 JdbcTemplate 中封装了一系列方法用于执行预编译的 SQL 语句, 它们都使用了 PreparedStatement 对象。下面列出一些常用的方法进行介绍:

```
public Object execute(String sql, PreparedStatementCallback action)
    throws DataAccessException

public List query(String sql, PreparedStatementSetter pss,
    RowMapper rowMapper) throws DataAccessException

public Object query(String sql, Object[] args, ResultSetExtractor rse)
throws
    DataAccessException

public void query(String sql, Object[] args, RowCallbackHandler rch)
throws
    DataAccessException

public List query(String sql, Object[] args, RowMapper rowMapper) throws
    DataAccessException

public Object queryForObject(String sql, Object[] args, RowMapper
rowMapper)
    throws DataAccessException
```

PreparedStatementCallback 回调接口可以控制预编译语句的执行，例如：

```
public List queryByUsername(final String username) {  
    final String sql = "SELECT * FROM " + TABLE_NAME  
        + " where username = ?";  
  
    final List list = new ArrayList<Admin>();  
  
    return (List) getJdbcTemplate().execute(sql,  
        new PreparedStatementCallback() {  
            public Object doInPreparedStatement(PreparedStatement ps)  
                throws SQLException, DataAccessException {  
                ps.setString(1, username);  
                ResultSet rs = ps.executeQuery();  
                while (rs.next()) {  
                    Admin admin = new Admin();  
                    admin.setId(rs.getInt(ID));  
                    admin.setUsername(rs.getString(USERNAME));  
                    admin.setPassword(PASSWORD);  
                    list.add(admin);  
                }  
                rs.close();  
                return list;  
            }  
        });  
}
```

可以看到，在 SQL 语句中，我们多了一个“？”。为什么要这样呢？使用传统的 Statement 时，JDBC 将会直接把 SQL 语句发送给 DBMS，DBMS 会先分析该 SQL 的语法正误、逻辑错误，再选择最优的查询方案、编译 SQL 语句并查询，最后返回结果。

这样看起来没什么问题，但是想象一下这样的情况：我们很频繁的执行一个类似的 SQL 语句：SELECT * FROM table WHERE property=value，于是 DBMS 也不得不频繁的去分析同一个 SQL 语句的语法错误并优化它。这看起来是多么荒唐，我们为什么不把这个语句存贮在 DBMS 中，让它只是执行一次这样的操作呢？这样将会节约大量的系统资源。

在上面的例子中“？”实际上是一个参数，我们通过 PreparedStatement 预先“告诉”DBMS：你把这句 SQL 语句分析优化后存贮起来，我将要重复使用，“？”是一个未知变量，它代表的值我在使用时再告诉你。这样，在大量的重复执行这个语句时，效率的提高将是非常可观的。

像上上面的例子那样使用 PreparedStatementCallback 并不是个好主意，因为使用起来将会非常的麻烦，实际上我们可以这样：

```
public List queryByUsername(final Admin admin) {
    final String sql = "SELECT * FROM " + TABLE_NAME
        + " where username = ? AND password = ?";

    return getJdbcTemplate().query(sql,
        new PreparedStatementSetter() {

            public void setValues(PreparedStatement ps)
                throws SQLException {
                ps.setString(1, admin.getUsername());
                ps.setString(2, admin.getPassword());
            }

        }, new BeanPropertyRowMapper(Admin.class));
}
```

这样看起来是要比 PreparedStatementCallback 回调接口的使用简洁的许多，但是我们还不满意，因为这样我们还是需要使用内部类的方式来执行 SQL 语句中的参数值。Spring 为我们提供了使用数组来传递参数的方法，使用范例如下：

```
public List queryByInstance(final Admin admin) {
    final String sql = "SELECT * FROM " + TABLE_NAME
        + " where username = ? AND password = ?";

    return getJdbcTemplate().query(sql,
        new Object[] { admin.getUsername(), admin.getPassword() },
        new BeanPropertyRowMapper(Admin.class));
}
```

实际上，Spring 为我们提供大量的这样的方法，在开发中，这将极大的减少了我们的代码量。这些方法的使用和这里类似，你可以参看 Spring API Doc。

5.3.6 插入和更新数据库

前面我们一直在讨论如何使用 SELECT 语句从数据库中获取信息，却没有提及如何更新数据库中的数据。这里我们将介绍 JdbcTemplate 模板中提供的数据库更新方法 update()，在 JdbcTemplate 中 INSERT、DELETE、UPDATE 这些数据库更新语句都将通过这个方法被执行：

```

public int update(final String sql) throws DataAccessException

public int[] batchUpdate(final String[] sql) throws DataAccessException

public int update(PreparedStatementCreator psc) throws
DataAccessException

public int update(final PreparedStatementCreator psc, final KeyHolder
generatedKeyHolder) throws DataAccessException

public int update(String sql, PreparedStatementSetter pss) throws
DataAccessException

public int update(String sql, Object[] args, int[] argTypes) throws
DataAccessException

public int update(String sql, Object[] args) throws DataAccessException

public int[] batchUpdate(String sql, final BatchPreparedStatementSetter
---

```

其中第一个和第二个方法将被用来执行静态的 SQL 语句：第一个执行单条 SQL 语句，第二个方法执行多条 SQL 语句。

其它的方法用来执行预编译 SQL 语句。下面有一些例子可供参考：

插入数据：

```

public int Insert(final Admin admin) {
    final String insertSql = "INSERT INTO " + TABLE_NAME
        + "(username,password) VALUES (?,?)";
    return getJdbcTemplate().update(insertSql,
        new PreparedStatementSetter() {
            public void setValues(PreparedStatement ps)
                throws SQLException {
                ps.setString(1, admin.getUsername());
                ps.setString(2, admin.getPassword());
            }
        });
}

```

或者：

```
public int Insert(final Admin admin) {  
    final String insertSql = "INSERT INTO " + TABLE_NAME  
        + "(username,password) VALUES (?,?)";  
    return getJdbcTemplate().update(insertSql,  
        new Object[] { admin.getUsername(), admin.getPassword() });  
}
```

要删除表中的数据，可以这样：

```
public int deleteById(int id) {  
    final String insertSql = "DELETE FROM " + TABLE_NAME + " WHERE id=?";  
    return getJdbcTemplate().update(insertSql, new Object[] { id });  
}
```

更新数据：

```
public int updateByInstance(final Admin admin) {  
    final String sql = "UPDATE " + TABLE_NAME  
        + " SET username = ? ,password = ? WHERE id=?";  
    return getJdbcTemplate().update(  
        sql,  
        new Object[] { admin.getUsername(), admin.getPassword(),  
            admin.getId() });  
}
```

当然，我们还可以使用 PreparedStatementCreator 回调接口来控制 PreparedStatement 的生成和 SQL 语句的执行。

5.3.7 一个完整的 JdbcDaoSupport 的例子

前面的代码都是一些代码片段，为了给大家一个完整的印象，我们这里用一个例子来展示如何使用 JdbcDaoSupport 类来和数据库交互。

例子代码详见 DB_Demo5.3。其工程在 MyEclipse6.5 中的结构如下：

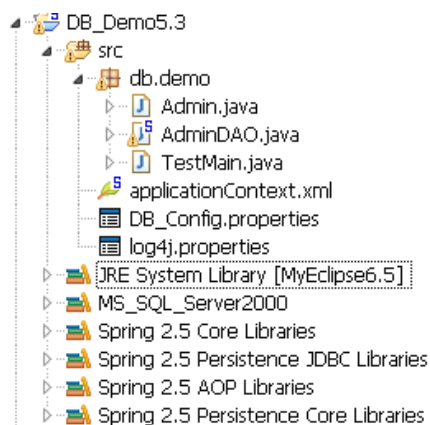


图 5.6 DB_Demo5.3 的工程结构

需要注意的是，我们必须添加 SQL-Server2000 的驱动程序 jar 文件到工程的 classpath 中。

这里我们仍然使用前面的 Admin 表，为此，我们需要一个 Admin 实体类，它是一个标准的 Java Bean，拥有和 Admin 属性一样的成员变量，代码如下：

```
package db.demo;

public class Admin {
    private int id;
    private String username;
    private String password;
    //Getter 和 Setter 省略
}
```

在工程中，和数据库交互的核心类是 AdminDAO 类，它继承于 JdbcDaoSupport 类，因此我们可以在其中直接使用 getJdbcTemplate()方法来获取 JdbcTemplate 类的实例，另外，我们也需要为其注入 dataSource。该类的详细代码如下：

```
package db.demo;

//import省略
public class AdminDAO extends JdbcDaoSupport {

    private final String ID = "id";
    private final String USERNAME = "username";
    private final String PASSWORD = "password";
    private final String TABLE_NAME = "admin";

    // 查找所有的Admin表中的实体<br/> 使用RowCallbackHandler
    public List<Admin> findAll() {

        final List<Admin> adminList = new ArrayList<Admin>();
        String sql = "select * from " + TABLE_NAME;

        getJdbcTemplate().query(sql, new RowCallbackHandler() {
            // 使用RowCallbackHandler回调接口进行简单的OR映射
            public void processRow(ResultSet rs) throws SQLException {
                Admin admin = new Admin();
                admin.setId(rs.getInt(ID));
                admin.setUsername(rs.getString(USERNAME));
                admin.setPassword(rs.getString(PASSWORD));
                adminList.add(admin);
            }
        });
        return adminList;
    }

    // 通过Id值查找所有的Admin表中的实体<br/> 使用RowMapper
    public Admin findById(int id) {

        String sql = "Select * from " + TABLE_NAME + " where " + ID + " = "
            + id;
        return (Admin) getJdbcTemplate().queryForObject(sql, new RowMapper() {
            // 使用RowMapper回调接口进行简单的OR映射
            public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
                Admin admin = new Admin();
                admin.setId(rs.getInt(ID));
                admin.setUsername(rs.getString(USERNAME));
                admin.setPassword(rs.getString(PASSWORD));
                return admin;
            }
        });
    }
}
```



```
//通过password字段查找所有的Admin表中的实体
public List findByPassword(final String password) {
    String sql = "Select * from " + TABLE_NAME + " where " + PASSWORD
        + " = ?";

    return getJdbcTemplate().query(sql, new PreparedStatementSetter() {

        public void setValues(PreparedStatement ps) throws SQLException {
            ps.setString(1, password);
        }

    }, new BeanPropertyRowMapper(Admin.class));
}

//通过username字段查找所有的Admin表中的实体<br/> 使用RowCallbackHandler 参数查询
public List<Admin> findByUsername(String username) {
    final List<Admin> adminList = new ArrayList<Admin>();
    String sql = "Select * from " + TABLE_NAME + " where " + USERNAME
        + " = ?";
    getJdbcTemplate().query(sql, new String[] { username },
        new BeanPropertyRowMapper(Admin.class));
    return adminList;
}

public List queryByInstance(final Admin admin) {
    final String sql = "SELECT * FROM " + TABLE_NAME
        + " where username = ? AND password = ?";
    return getJdbcTemplate().query(sql,
        new Object[] { admin.getUsername(), admin.getPassword() },
        new BeanPropertyRowMapper(Admin.class));
}

//查询记录总数<br/>
public int findColumnNumber() {
    String sql = "Select count(*) from " + TABLE_NAME;
    return getJdbcTemplate().queryForInt(sql);
}

public int Insert(final Admin admin) {
    final String insertSql = "INSERT INTO " + TABLE_NAME
        + "(username,password) VALUES (?,?)";
    return getJdbcTemplate().update(insertSql,
        new Object[] { admin.getUsername(), admin.getPassword() });
}
```

```
public int deleteById(int id) {  
    final String insertSql = "DELETE FROM " + TABLE_NAME + " WHERE  
id=?";  
    return getJdbcTemplate().update(insertSql, new Object[] { id });  
}  
  
public int updateByInstance(final Admin admin) {  
    final String sql = "UPDATE " + TABLE_NAME  
        + " SET username = ? ,password = ? WHERE id=?";  
    return getJdbcTemplate().update(  
        sql,  
        new Object[] { admin.getUsername(), admin.getPassword(),  
            admin.getId() });  
}
```

着看起来是有点冗长，但是我们通过它把对 Admin 表的常用操作都封装了起来。在以后要对 Admin 表进行相关操作的时候，我们就可以直接对 Admin 对象进行操作而不需要再编写 SQL 代码，使其看起来和其它的 ORM 工具没有区别。这全靠 Spring 为我们做了很多的前期工作。

该例子的 Spring 配置文件非常简单，如下：

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">  
    <bean id="AdminDAO" class="db.demo.AdminDAO">  
        <property name="dataSource" ref="dataSource" />  
    </bean>  
    <bean id="dataSource"  
class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
        <property name="driverClassName" value="${db.ClassName}" />  
        <property name="url" value="${db.url}" />  
        <property name="username" value="${db.username}" />  
        <property name="password" value="${db.password}" />  
    </bean>  
    <bean class="org.springframework.beans.factory.config.PropertyPlaceholderC  
onfigurer">  
        <property name="location" value="DB_Config.properties" />  
    </bean>  
</beans>
```

当然，我们还需要一个数据库配置文件 DB_Config.properties:

```
db.ClassName=com.microsoft.jdbc.sqlserver.SQLServerDriver
db.url=jdbc\:microsoft\:sqlserver\://localhost\:1433;DatabaseName\=DB_Demo
db.username=sa
db.password=123456
```

这样，我们就可以创建一个测试类来试试我们的 AdminDAO 类是不是能够正常工作了。

测试类的代码如下：

```
package db.demo;

import java.sql.SQLException;
import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestMain {

    public static void main(String[] args) throws SQLException {

        // 从数据库源获得连接 ,这里可以用Spring IoC注入
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "applicationContext.xml");
        AdminDAO adminDAO = (AdminDAO)ac.getBean("AdminDAO");

        Admin admin = new Admin();
        admin.setUsername("username2");
        admin.setPassword("password2");
        admin.setId(6);

        adminDAO.Insert(admin); // 增
        adminDAO.deleteById(5); // 删除
        adminDAO.updateByInstance(admin); // 修改
        List<Admin> adminList=adminDAO.findAll(); // 查询
        for (Admin admin1:adminList) {
            System.out.println(admin1.getId() + ": username:"
                + admin1.getUsername() + ",password:" +
                admin1.getPassword());
        }
    }
}
```

运行主类，输出结果如下：

```

terminated: resultant (J) Java Application L:\java\book\spring\demo5.3
1: username:admin,password:admin888
2: username:cmzy,password:cmzy888
3: username:test,password:test888
4: username:TestUsername,password:TestPassword
6: username:username2,password:password2
7: username:username2,password:password2

```

图 5.7 DB_Demo5.3 运行结果

5.3.8 使用 NamedParameterJdbcDaoSupport

从 2.0 开始，Spring 通过 `org.springframework.jdbc.core.namedparam` 包提供了对命名参数的支持。前面我们使用 “?” 符来指定 SQL 语句中的参数占位符，而通过 `NamedParameterJdbcTemplate` 模板类，我们可以使用命名参数。

Spring 提供了命名参数支持的 DAO 类 `NamedParameterJdbcDaoSupport`，通过扩展该类，可以使用 `getNamedParameterJdbcTemplate()` 方法获取 `NamedParameterJdbcTemplate` 模板类的实例，`NamedParameterJdbcTemplate` 模板类的使用和 `JdbcTemplate` 大致相同，不同的是它提供了对命名参数的支持。同 `JdbcTemplate` 一样，`NamedParameterJdbcTemplate` 同样需要 `DataSource`，我们可以使用 Spring IoC 容器为它注入。一个简单的例子如下（代码详见 DB_Demo5.4 AdminDAO 类）：

```

public Admin findById(int id) {

    final String sql = "Select * from " + TABLE_NAME + " where "
        + "ID+\":\"+id";

    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("id", Integer.valueOf(id));

    return (Admin) getNamedParameterJdbcTemplate().queryForObject(sql,
        new MapSqlParameterSource(map),
        new BeanPropertyRowMapper(Admin.class));
}

```

在这里，我们使用 `ParameterJdbcTemplate` 提供的 `queryForObject()` 方法来执行查询。在 SQL 语句中使用了 “id” 字符串作为参数名称。因此，我们需要在 `queryForObject()` 方法中传入一个 `SqlParameterSource` 的实例来指定 “id” 参数所代表的值。`SqlParameterSource` 是 Spring 提供的一个接口，它的实现类继承关系如下：

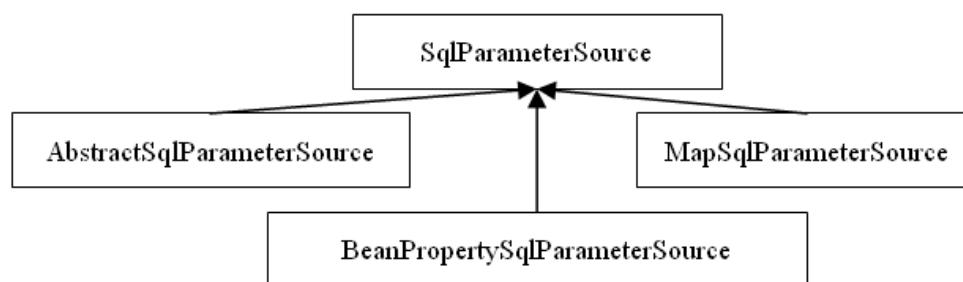


图 5.8 SqlParameterSource 的继承图

MapSqlParameterSource: 通过 Map 类指定命名参数的值。其中 Map 中的 Key 指定命名参数名称，Value 指定命名参数的值。甚至，我们也可以直接将 Map 对象传入到 queryForObject()方法中，比如（代码详见 DB_Demo5.4 AdminDAO 类）：

```

public List findByPassword(final String password) {
    String sql = "Select * from " + TABLE_NAME + " where " + PASSWORD
        + " = :password";

    Map<String, String> map = new HashMap<String, String>();
    map.put("password", password);

    return getNamedParameterJdbcTemplate().query(sql, map,
        new BeanPropertyRowMapper (Admin.class));
}

```

BeanPropertySqlParameterSource: 通过给定 Bean 的属性指定命名参数的值。比如（代码详见 DB_Demo5.4 AdminDAO 类）：

```

public List findByInstance(final Admin admin) {
    final String sql = "Select * from " + TABLE_NAME
        + " where "+USERNAME+" = :username OR "+PASSWORD+" =
:password";

    return getNamedParameterJdbcTemplate().query(sql,
        new BeanPropertySqlParameterSource(admin),
        new BeanPropertyRowMapper (Admin.class));
}

```

上面的例子在工程 DB_Demo5.4 中可以看到完整的源代码。

5.3.9 使用 SimpleJdbcDAOSupport

为了使用 Java 5 的新特性，Spring 提供了 org.springframework.jdbc.core.simple 包。其中最重要的是 SimpleJdbcTemplate 和 SimpleJdbcDaoSupport 类，通过扩展 SimpleJdbcDaoSupport 类，我们可以轻松的完成自己的 DAO 类。

下面有一些使用 SimpleJdbcTemplate 的例子，详见工程 DB_Demo5.5。

一个简单的查询:

```
public List<Admin> findAll() {  
    final String sql = "select * from " + TABLE_NAME;  
    // 准备RowMapper  
    ParameterizedBeanPropertyRowMapper<Admin> pm = new  
        ParameterizedBeanPropertyRowMapper<Admin>();  
    pm.setMappedClass(Admin.class);  
  
    return getSimpleJdbcTemplate().query(sql, pm);  
}
```

通过含命名参数的预编译语句查询:

```
public List<Admin> findByPassword(final String password) {  
    final String sql = "Select * from " + TABLE_NAME + " where " +  
PASSWORD  
        + " = :password";  
    // 准备参数  
    Map<String, String> map = new HashMap<String, String>();  
    map.put("password", password);  
    // 准备RowMapper  
    ParameterizedBeanPropertyRowMapper<Admin> pm = new  
        ParameterizedBeanPropertyRowMapper<Admin>();  
    pm.setMappedClass(Admin.class);  
  
    return getSimpleJdbcTemplate().query(sql, pm, map);  
}
```

通过预编译语句查询, SQL 语句的参数可以直接通过 query()方法的参数传入:

```
public List<Admin> findByUsername(String username) {  
  
    String sql = "Select * from " + TABLE_NAME + " where " + USERNAME  
        + " = ?";  
    // 准备RowMapper  
    ParameterizedBeanPropertyRowMapper<Admin> pm = new  
        ParameterizedBeanPropertyRowMapper<Admin>();  
    pm.setMappedClass(Admin.class);  
  
    return getSimpleJdbcTemplate().query(sql, pm, username);  
}
```

插入数据:

```
public int insert(final Admin admin) {
    final String sql = "insert into " + TABLE_NAME + "(" + USERNAME +
        ","
        + PASSWORD + ") values (:username,:password)";

    return getSimpleJdbcTemplate().update(sql,
        new BeanPropertySqlParameterSource(admin));
}
```

从 Spring2.5 开始, Spring 还扩展 org.springframework.jdbc.core.simple 包, 增加了如下类:

- 1、 **ParameterizedBeanPropertyRowMapper<T>**: BeanPropertyRowMapper 的子类。用 Java 5 的泛型封装了 BeanPropertyRowMapper。
- 2、 **ParameterizedSingleColumnRowMapper<T>**: SingleColumnRowMapper 的子类, 功能和 SingleColumnRowMapper 一致, 不同的是通过它可以享受到 Java 5 泛型带来的好处。
- 3、 **SimpleJdbcInsert**: Spring2.5 提供的一个数据库插入操作的辅助类。使用例子如下:

```
public int insert(String username, String password) {

    SimpleJdbcInsert si = new SimpleJdbcInsert(getDataSource())
        .withTableName(TABLE_NAME);
    si.usingColumns(USERNAME, PASSWORD);

    Map<String, Object> map = new HashMap<String, Object>();
    map.put(USERNAME, username);
    map.put(PASSWORD, password);

    return si.execute(map);
}
```

其中 withTableName()方法用于指定将要被插入的数据表名, usingColumns()方法指定被插入的列名, 必须要和 map 中的 key 值一样。

注意: 使用 SimpleJdbcInsert 时, 可能会抛出 MetadataAccessException 异常, 提示驱动程序未实现 DatabaseMetaData, 因此如果仅仅是为了测试, 可以使用 JDBC-ODBC 桥接驱动程序连接 SQL-Server 数据库。代码如下:

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
        value="sun.jdbc.odbc.JdbcOdbcDriver" />
    <property name="url" value="jdbc:odbc:SQLServer" />
    <property name="username" value="${db.username}" />
    <property name="password" value="${db.password}" />
</bean>
```

4、 SimpleJdbcCall: Spring2.5 提供的一个调用数据库存储过程的辅助类。

在我们提供的 DB_Demo5.5 中详细的演示了如何使用 SimpleJdbcTemplate 类来提高的我们的工作效率，如果你的工程是运行在 Java 5 以上，那么 Spring 推荐你使用 SimpleJdbcTemplate 类而不是 JdbcTemplate 和 NamedParameterJdbcTemplate，因为前者除了已经包含了后两种类的所有功能外，还提供了额外的便捷的调用它们的方式。

.....正在等待完成...

...由于最近要找工作、而且学习任务也比较紧.....

该书的更新可能将会非常慢，请见谅。

再次感谢您的支持。

等过了这段时间，我会继续更新。

如果您有任何建议或者板砖，我都非常欢迎，请发送 **mail** 到 **dashoumail@163.com**

5.4 集成 Hibernate

5.4.1 Hibernate 简介

5.4.2 一个简单的 Hibernate 例子

5.4.3 HibernateTemplate 和 HibernateDaoSupport

5.4.4 使用 MyEclipse 自动生成实体类和 DAO 类

5.5 小结

5.6 参考文档和推荐资料

[1] Spring 团队.Spring reference