《深入 Spring 2：轻量级 J2EE 开发框架原理与实践》

(作者：蔡世友　吴嘉俊　冯煜　张钰)

简介：

本书首先是一本通过通俗案例讲解 Spring 的教程；同时也是一本深入挖掘 Spring 及相关框架结构、设计原理的书；更是一本探讨 J2EE 软件开发中的艺术的书。本书还想讲述一条开源框架设计中金科玉律：思想决定一切，万变不离其宗。

本书分成四个部分，第一部分是 Spring 新手上路，主要讲解轻量级构架中的相关概念、发展过程、所涉及到的相关技术及详细使用方法等；第二部分是一个综合的案例，讲解如何使用 Spring 及相关技术来构建 J2EE 应用；第三部分是 Spring 的原理部分，主要探讨 Spring 框架的结构，设计原理，Spring 项目源码分析等，让我们深入到 Spring 的核心；本书的第四部分主要探讨开源领域的一些相关话题，让大家对开源有更加深入的认识。

为了能让大家了解Spring、学会Spring、透视Spring的内核、掌握Spring的设计原理、领略Java艺术之精妙，我们为此做了很多工作。我们在EasyJF开源交流社区上开通了一个专用于解决轻量级J2EE开发问题的栏目，并请专人负责解决大家在学习及工作过程中遇到的问题，网址：http://www.easyjf.com/bbs。另外我们还通过EasyJF把本书核心案例作为一个持续的开源项目，会长期根据 Spring 的变更而更新，SVN 地址：http://svn.easyjf.com/repository/easyjf/spring-road/。

当然，由于时间仓促及作者水平有限，本书难免带有一些不成熟的观点，不可避免存在一些问题。为此，我们将通过 SVN 及交流论坛对本书的补充内容进行不断更新，以确保广大的读者能接触最新、最实用的 Spring 技术。书中存在的问题及不足之处，还请您多给我们提建议及意见，谢谢！

关于本书的电子版本发布申明：

在出版社及本书全部作者的一致同意下，本书的一些重要章节将在互联网免费公开发行，欢迎各大网站及媒体在保留作者及版权申明的前提下转载，本书电子版本不得用于收费发布及平面发行。

另外，由于本书还处于最后组稿阶段，因此，电子版与最终出版的图书内容会存在一定的差异，我们将会通过 EasyJF 官网及相关网站上对电子版进行即时更新。

致谢：

在创作本书的过程中，EasyJF 开源的 williamRyam、天一、瞌睡虫、云淡风轻、与狼共舞、abc、netgod、navImg2 等很多成员给予了很我们很大的帮助，在此深表感谢。

作者邮箱：

蔡世友　caishiyou@sina.com.cn
吴嘉俊　stef_wu@163.com
冯　煜　fengyu8299@126.com
张　钰　zhangyu20030607@hotmail.com

《深入 **Spring 2**：轻量级 **J2EE** 开发框架原理与实践》第五章

面向切面的编程(**AOP**)及在 **Spring** 中的应用

目　录

第五章　面向方面的编程(AOP)及在Spring中的应用 1
5.1 AOP简介 1
5.1.1 AOP概念 1
5.1.2 AOP中的一些相关术语介绍 3
5.1.3 AOP与OOP关系 6
5.1.4 AOP联盟简介 6
5.1.5 AOP相关框架及工具简介 8
5.1.6 AOP在企业级应用程序中的作用 8
5.2 AspectJ简介及快速入门 9
5.2.1 AspectJ介绍 9
5.2.2 AspectJ的下载及安装 9
5.2.3 在Eclipse中开发AspectJ程序 13
5.2.4 AspectJ版的HelloWorld 15
5.2.5 AspectJ中相关语法 17
5.2.6 一个简单的回合格斗小游戏示例 23
5.3 一个简单的Spring AOP示例 27
5.3.1 定义业务组件 28
5.3.2 使用基于Schema的配置文件配置Spring AOP 29
5.3.3 使用Java5 注解配置及使用Spring AOP 31
5.3.4 基于API方式来使用Spring AOP 32
5.4 Spring中的AOP实现及应用 34
5.4.1 简介 34
5.4.2 Spring AOP中对AspectJ的支持 35
5.4.3 Spring AOP配置方法 36
5.4.4 切入点(Pointcut) 43
5.4.5 增强(Advice) 47
5.4.6 引介(Introduction) 51
5.4.7 增强器/切面封装(Advisor) 54
5.4.8 ProxyFactoryBean 57
5.5 示例：模拟Warcraft游戏 60
5.5.1 示例简介 60
5.5.2 核心关注点及系统主模块 61
5.5.3 横切关注点需求引入及实现 70
5.5.4 使用AspectJ注解支持的AOP实现 78
5.5.5 使用基于Schema的方式配置Spring AOP 83
5.6 小结 87
5.7 思考题 87

第五章　面向方面的编程(AOP)及在 Spring 中的应用

AOP 全名 Aspect-Oriented Programming，中文直译为面向切面(方面)编程，当前已经成为一种比较成熟的编程思想，可以用来很好的解决应用系统中分布于各个模块的交叉关注点问题。在轻量级的 J2EE 中应用开发中，使用 AOP 来灵活处理一些具有横切性质的系统级服务，如事务处理、安全检查、缓存、对象池管理等，已经成为一种非常适用的解决方案。

本章首先简单讲解 AOP 的相关概念以及在 Java 领域中最为出色的 AOP 实现 AspectJ 的应用，然后重点讲解 Spring2 中 AOP 的实现及应用，最后通过一个有趣、完整的模拟 Warcraft 游戏示例来演示 Spring2 中的 AOP 的各种用法。

本章的主要是针对刚刚开始接触 AOP 编程方法、AsepectJ、Spring AOP 的读者，另外也针对熟悉 Spring2.0 以前的 AOP 但不熟悉 Spring2 中 AOP 使用的读者。本章主要从应用的角度分析轻量级应用中的 AOP 编程以及 Spring2 中 AOP 的使用方法，若您对 AOP 的实现原理、Spring AOP 的底层构架原理及 AOP 高级应用技巧感兴趣，请阅读本书第三部分的《AOP 原理及实现》一章中的相关内容。

5.1 AOP 简介

AOP 全名 Aspect-Oriented Programming，中文直译为面向切面 (方面)编程，是近两三年来流行起来的一种编程思想，其弥补了面向对象编程(OOP)中存在的一些不足，让我们可以编写出更加简洁、易维护、复用性更强的程序。本节主要通过一个实例引入 AOP 中的相关概念，并简单介绍了 AOP 中的各种相关术语，然后分析了 AOP 与 OOP 的关系，介绍了 AOP 联盟及其发布的 API，并对当前一些 AOP 框架及工具作了简单介绍，最后简单分析了 AOP 在企业级应用中的作用。

5.1.1 AOP 概念

AOP 全名 Aspect-Oriented Programming，中文直译为面向切面 (方面)编程，是近两三年来流行起来的一种编程思想，用来解决 OOP 编程中无法很好解决问题。作为一种编程思想，其最初是由Gregor Kiczales在施乐的Palo Alto研究中心领导的一个研究小组于1997年提出。

■　问题引入

在传统 OOP 编程，我们通过分析、抽象出一系列具有一定属性与行为的对象，并通过这些对象之间的协作来形成一个完整的软件功能。由于对象可以继承，因此我们可以把具有相同功能或相同特性的属性抽象到一个层次分明的类结构体系中。随着软件规范的不断扩大，专业化分工越来越系列，以及 OOP 应用实践的不断增多，随之也暴露出了一些 OOP 无法很好解决的问题。

假设我们有一个业务组件 Component，里面有 3 个业务方法，如下所示：

```
public class Component {
//业务方法1
public void business1()
{
    //doSomeThing1
```

```
}
//业务方法2
public void business2()
{
    //doSomeThing2
}
//业务方法3
public void business3()
{
    //doSomeThing3
}
}
```

由于需求的变更，需要在每个方法执行前都要进行用户合法性验证，只有合法的用户才能执行业务方法里面的内容，因此，我们在三个方法中的第一行都需要加如一个用户合法性检验的代码。另外，我们还需要在运行方法中的实际业务逻辑前启动一个事务，在业务逻辑代码执行完成后结束一个事务，需要在方法中加入开始事务处理及结束事务处理的代码。最后，我们还需要在每一个方法结束后，把一些信息写入日志系统中。因此需要在每一个方法的最后添加记录日志的代码，这时业务方法变成如下的形式：

```
public void businessX()
{
    validateUser();
    beginTransaction();
    //doSomeThing
    endTransaction();
    writeLogInfo();
    }
```

假如我们的系统有成千上万个这样业务方法，都需要执行用户权限验证、事务处理、日志书写或审计等类似的工作，系统中就会充斥着非常多的重复性代码，造成代码书写及维护极其不便。例如，由于需求的变更，我们要取消一部分业务方法中的开启事务或结束事务处理的功能。此时，我们需要手工逐一删除掉这些方法中事务处理语句。

■ **问题解决**

我们能不能不用在业务方法中添加哪些重性的代码，而通过某种机制，让权限验证、事务处理、日志记录等功能自动在这些方法指定的位置自动运行呢？为了能更好地解决上面提到的问题，于是引入了 AOP（即面向切面）编程思想。

例如，使用 AspectJ 中，我们可以定义一个切面，代码如下：

```
public aspect MyAspect {
void around():call(void Component.business*(..))
{
    validateUser();
    beginTransaction();
    proceed();
    endTransaction();
    writeLogInfo();
}
```

```
}
```

这样，所有 Component 组件中返回值为 void、名称以 business 开头的方法都会自动具有了用户验证、事务处理、日志记录等功能。

假如要进一步使某一个包中的所有名称以 business 开头、返回值为 void 的方法都具有上面的功能。则只需要把上面 MyAspect 中的内容修改一下即可，如下所示：

```
void around(): call(void springroad.demo.chap5..business*(..))
{
    validateUser();
    beginTransaction();
    proceed();
    endTransaction();
    writeLogInfo();
}
```

在这里只需要知道可以使用 AOP 方式来实现前面的所提需求，要编译这个程序，需要使用到 AspectJ 编译器，关于 AspectJ，我们将会在本章下一节作介绍。

在 AOP 中，我们把前面示例中分散程序各个部分，解决同样问题的代码片段，称为问题的切面(或方面)。一个切面可以简单的理解为解决跨越多个模块的交叉关注点问题(大多数是一些系统级的或者核心关注点外围的问题)的模块。通过 AOP 可以使用一种非常简单、灵活的方式，在切面中实现了以前需要在各个核心关注点中穿插的交叉关注的功能，从而使得解决系统中交叉关注点问题的模块更加容易设计、实现及维护。

提供对 AOP 编程方法支持的平台称为 AOP 实现或框架，比如 AspectJ、JBoos AOP、Spring AOP 等。

5.1.2 AOP 中的一些相关术语介绍

在 AOP 编程中，包括很多新名词及概念，如关注点、核心关注点、方面、连接点、通知、切入点、引介等。由于 AOP 仍处于发展阶段，很多名称及术语没有统一的解释。因此，本书中关于 AOP 的一些术语均为当前主流的叫法。本章重点介绍的讲解轻量级 J2EE 中的 AOP 框架的应用，而关于面向切面的设计及编程方法等一些话题，我们只略作介绍。

- **关注点(Concern)**

关注点也就是我们要考察或解决的问题。比如在一个在一个电子商务系统中，订单的处理，用户的验证、用户日志记录等都属于关注点(Core Concerns)。核心关注点，是只一个系统中的核心功能，也就是一个系统中跟特定业务需求联系最紧密的商业逻辑。在一个电子商务系统中，订单处理、客户管理、库存及物流管理都是属于系统中的核心关注点。除了核心关注点以外，还有一种关注点，他们分散在每个各个模块中解决同一样的问题，这种跨越多个模块的关注点称为横切关注点或交叉关注点(Crosscutting Concerns)。在一个电子商业系统中，用户验证、日志管理、事务处理、数据缓存都属于交叉关注点。

在 AOP 的编程方法中，主要在于对关注点的提起及抽象。我们可以把一个复杂的系统看作是由多个关注点来有机组合来实现，一个典型的系统可能会包括几个方面的关注点，如核心业务逻辑、性能、数据存储、日志、授权、安全、线程及错误检查等，另外还有开发过程中的关注点，如易维护、易扩展等。

- **切面(Aspect)**

切面是一个抽象的概念，从软件的角度来说是指在应用程序不同模块中的某一个领域或

方面。从程序抽象的角度来说，可以对照 OOP 中的类来理解。OOP 中的类(class)是实现世界模板的一个抽象，其包括方法、属性、实现的接口、继承等。而 AOP 中的切面(aspect)是实现世界领域问题的抽象，他除了包括属性、方法以外，同时切面中还包括切入点 Pointcut、增强(advice)等，另外切面中还可以给一个现存的类添加属性、构造函数，指定一个类实现某一个接口、继承某一个类等。比如，在 Spring AOP 中可以使用下面的配置来定义一个切面：

```
<aop:aspect id="aspectDemo" ref="aspectBean">
 <aop:pointcut id="somePointcut"  expression="execution(* Component.*(..)" />
 <aop:after-returning pointcut-ref="log" method="" />
</aop:aspect>
```

- **连接点(Join point)**

连接点也就是运用程序执行过程中需要插入切面模块的某一点。连接点主要强调的是一个具体的“点”概念。这个点可以是一个方法、一个属性、构造函数、类静态初始化块，甚至一条语句。比如前面的例子中，连接点就是指具体的某一个方法。

在一般的 AOP 框架中，一般采用签名的方式来描述一个连接点，有的 AOP 框架只有很少类型的连接点，如 Spring AOP 中当前只有方法调用。

- **切入点(Pointcuts)**

切入点指一个或多个连接点，可以理解成一个点的集合。切入点的描述比较具体，而且一般会跟连接点上下文环境结合。比如，在前面的例子中，切入点“execution(* Component.*(..)”表示“在 Component 类中所有以 business 打头的方法执行过程中”，其包含了 3 个连接点(business1、business2、business3)的集合。另外，“Component 类中的所有方法调用”、“包 com.easyjf.service 里面所有类中所有方法抛出错误”、“类 UserInfo 的所有 getter 或 seeter 方法执行”，这些都可以作为切入点。另外，在大多数 AOP 框架实现中，切入点还支持集合运算，可以把多个切入点通过一定的组合，形成一个新的切入点。在 AspectJ 中，可以使用||、&&、!等操作符来组合得到一个符合特定要求的切入点。如：

pointcut setter(): target(UserInfo) && (call(void set*(String)) || call(void set*(int)));

表示所有 UserInfo 类中的所有带一个 String 或 int 型参数的 setter 方法。

pointcut transaction():target(service..)&&call(* save*(..))

表示 service 包中所有以 save 开头的方法。

- **增强或通知(Advice)**

Advice 一词不管翻译成建议、通知或者增强，都不能直接反映其内容，因此本书主要使用“增强”这一叫法。当然也可以把其仅看作是一个简单名词的来看待。增强(Advice)里面定义了切面中的实际逻辑（即实现)，比如日志的写入的实际代码，或是安全检查的实际代码。换一种说法增强(Advice)是指在定义好的切入点处，所要执行的程序代码。比如，下面的话都是用来描述增强(Advice)的例子：“当到达切入点 seeter 时，检查该方法的参数是否正确”、“在 save 方法出现错误这个切入点，执行一段错误处理及记录的操作”。一般情况下，增强(通知)主要有前增强、后增强、环绕增强三种基本类型。

前增强（before advice）－是指在连接点之前，先执行增强中的代码。

后增加（after advice）－是指在连接点执行后，再执行增强中的代码。后增强一般分为

连接点正常返回增加及连接点异常返回增强等类型。

环绕增强(around advice)－是一种功能强大的增强，可以自由改变程序的流程，连接点返回值等。在环绕增强中除了可以自由添加需要的横切功能以外，还需要负责主动调用连接点(通过 proceed)来执行激活连接点的程序。

■ **引介(Introduction)**

引介是指给一个现有类添加方法或字段属性，引介还可以在不改变现有类代码的情况下，让现有的 Java 类实现新的接口，或者为其指定一个父类从而实现多重继承。相对于增强(Advice)可以动态改变程序的功能或流程来说，引介(Introduction)则用来改变一个类的静态结构。比如我们可以让一个现有为实现 java.lang.Cloneable 接口，从而可以通过 clone()方法复制这个类的实例。

■ **织入(weaving)**

织入是指把解决横切问题的切面模板，与系统中的其它核心模块通过一定策略或规则组合到一起的过程。在 java 领域，主要包括以下三种织入方式：

1、运行时织入－即在 java 运行的过程中，使用 Java 提供代理来实现织入。根据代理产生方式的不同，运行时织入又可以进一步分为 J2SE 动态代理及动态字节码生成两种方式。由于 J2SE 动态代理只能代理接口，因此，需要借助于一些动态字节码生成器来实现对类的动态代理。大多数 AOP 实现都是采用这种运行时织入的方式。

2、类加载器织入－指通过自定义的类加载器，在虚拟机 JVM 加载字节码的时候进行织入，比如 AspectWerkz(已并入 AspecJ)及 JBoss 就使用这种方式。

3、编译器织入－使用专门的编译器来编译包括切面模块在内的整个应用程序，在编译的过程中实现织入，这种织入是功能最强大的。编译器织入的 AOP 实现一般都是基于语言扩展的方式，即通过对标准 java 语言进行一些简单的扩展，加入一些专用于处理 AOP 模块的关键字，定义一套语言规范，通过这套语言规范来开发切面模块，使用自己的编译器来生成 java 字节码。AspectJ 主要就是是使用这种织入方式。

■ **拦截器(interceptor)**

拦截器是用来实现对连接点进行拦截，从而在连接点前或后加入自定义的切面模块功能。在大多数 JAVA 的 AOP 框架实现中，都是使用拦截器来实现字段访问及方法调用的拦截(interception)。所用作用于同一个连接点的多个拦截器组成一个连接器链(interceptor chain)，链接上的每个拦截器通常会调用下一个拦截器。Spring AOP 及 JBoos AOP 实现都是采用拦截器来实现的。

■ **目标对象(Target object)**

指在基于拦截器机制实现的 AOP 框架中，位于拦截器链上最未端的对象实例。一般情况下，拦截器未端都包含一个目标对象，通常也就是实际业务对象。当然，也可以不使用目标对象，直接把多个切面模块组织到一起，形成一个完整最终应用程序，整个系统完全使用基于 AOP 编程方法实现，这种情况少见。

■ **AOP 代理(proxy)**

Aop 代理是指在基于拦截器机制实现的 AOP 框架中，实际业务对象的代理对象。这个代理对象一般被切面模块引用，AOP 的切面逻辑正是插入在代理对象中来执行的。AOP 代理的包括 J2SE 的代理以及其它字节码生成工具生成的代理两种类型。

5.1.3 AOP 与 OOP 关系

在面向对象(OOP)的编程中，我们是通过对现实世界的抽象及模型化上来分析问题，也即把一个大的应用系统分成一个一个的对象，然后把他们有机的组合在一起完成；而在面向切面(AOP)的编程中，分析问题是从关注点的角度出发，把一个软件分成不同的关注点，软件核心业务逻辑一般都比较集中、单一，这种关注点称为核心关注点，而一些关注属于分散在软件的各个部分(主要是软件核心业务逻辑)，这种关注点称为横切关注点。核心关注点可以通过传统的 OOP 方法来实现，而横切关注点则可以通过 AOP 的方法解决，即把实现相同功能、解决共性问题并分散在系统中各个部分的模块纳入一个切面中来处理。使用 AOP 编程，除了把一些具有共性的功能放到切面模块中以外，还可以在切面中给已有的类增加新的属性、实现新的接口等。也就是说，不但可以从类的外部动态改变程序的运行流程、给程序增加特定功能，还可以改变其静态结构。

因此，面向对象编程（OOP）解决问题的重点在于对具体领域模型的抽象，而面向切面编程（AOP）解决问题的关键则在于对关注点的抽象。也就是说，系统中对于一些需要分散在多个不相关的模块中解决的共同问题，则交由 AOP 来解决；AOP 能够使用一种更好的方式来解决 OOP 不能很好解决的横切关注点问题以及相关的设计难题来实现松散耦合。因此，面向方面编程 (AOP) 提供另外一种关于程序结构的思维完善了 OOP，是 OOP 的一种扩展技术，弥补补了 OOP 的不足。

OOP 编程基本流程

1、 归纳分析系统，抽象领域模型；
2、 使用 class 来封装领域模型，实现类的功能；
3、 把各个相关联的类组装到一起形成一个完整的系统。

AOP 编程基本流程

１、归纳分析系统中的关注点，分解切面；
２、按模块化的方式实现各个关注点中的功能，使用传统的编程方法如 OOP；
３、按一定的规则分解及组合切面(织入或集成)，形成一个完整的系统。

5.1.4 AOP 联盟简介

AOP 联盟(AOP Alliance)是由 Java 领域比较知名的一些专家及组织为了推进 AOP 运用研究，建立一个通用的 AOP 规范而成立起来的组织。组织中的成员都是在 AOP 编程思想及技术研究中有着比较突出贡献的专家及学者，其中有 AspectWerkz 的 Jonas Bonér、JAC 的 Laurent Martelli、Spring 的发起人 Rod Jonhson 等等。

通过 AOP 联盟的共同研究，可以避免一些重复性工作。AOP 联盟提供了一个公共的 AOP API，大多数知名的 AOP 框架或实现 (如 JBoss AOP、AspectJ、Spring 等)都直接或间接对其 AOP 进行了集成或支持。从而可以供各种 AOP 开发工具及框架能简单在各个 AOP 应用环境中应用、移植。

AOP 联盟 API 简介

AOP 联盟制订了一套用于规范 AOP 实现的底层 API，通过这些统一的底层 API，可以使得各个 AOP 实现及工具产品之间实现相互移植。这些 API 主要以标准接口的形式提供，是 AOP 编程所要解决的横切交叉关注点问题各部件的最高抽象，Spring 的 AOP 框架中也直接以这些 API 为基础所构建。下面我我们来看看当前 AOP 联盟发布的 AOP 相关接口。

AOP 联盟的 API 主要包括四个部分，第一个是 aop 包，定义了一个表示增强(Advice)

的标识接口，各种各样的增强(Advice)都继承或实现了这个接口；aop 包中还包括了一个用于描述 AOP 系统框架错误的运行时异常 AspectException。

第二个部分是 intercept 包，也就是拦截器包，这个包中规范了 AOP 核心概念中的连接点(join point)及增强(Advice)类型。

第三部及第四部分是 instrument 及 reflect 包。这两个包中的 API 主要包括 AOP 框架或产品为了实现把横切关注点的模块与核心应用模块组合集成，所需要使用的设施、技术及底层实现规范等。

“图 5-1”及“图 5-2”是两张关于介绍 AOP 联盟所发布的连接点(Joinpint)及增强(Advice)的 UML 结构图，通过这两张图，我们可以更加清晰了解一些 AOP 框架(如 Spring 中的 AOP 框架)的体系结构。

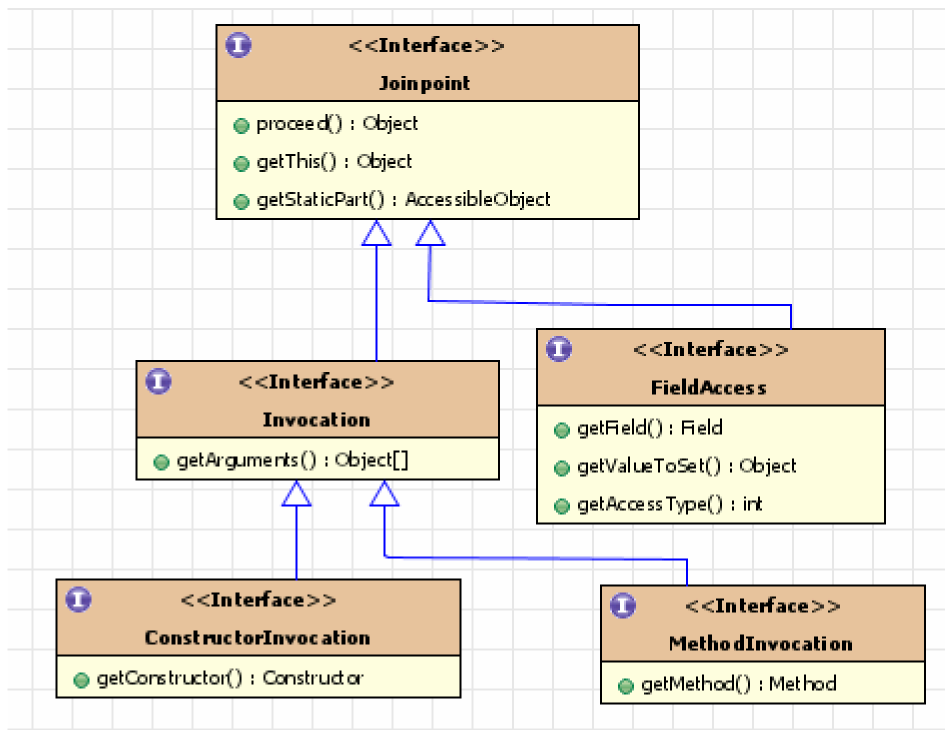

图 5-1　AOP 联盟定义的连接点(join point)API

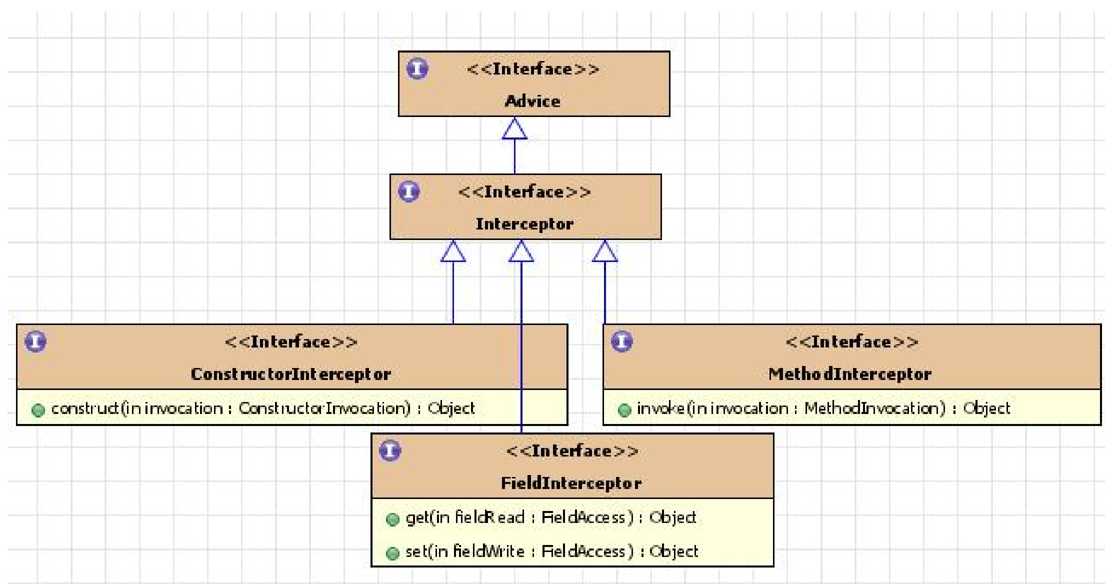

图 5-2 AOP 联盟定义的增强(Advice) API

5.1.5 AOP 相关框架及工具简介

一个 AOP 框架或实现主要有两部分功能，第一部分是通过定义一种机制来实现连接点、切入点及切面定义(描述)及封装实现，可以是一套语言规范或编程规范；另外一个部分就是提供把切面模块的逻辑通过织入(weaving)，与系统的其它部分组合到一起，形成一个完整的系统。要使用 AOP 技术，不需要从最底层开始逐一实现，可以使用一些现存的 AOP 框架或辅助工具来引入 AOP 编程方法的支持，下面我们简单介绍 Java 中的一些 AOP 框架及工具。

主要的 AOP 实现及框架

AspectJ: 对 java 进行了扩展，形成一个功能强大、灵活、实用的 AOP 语言。AspectJ 在 java 的基础上，加入一些 AOP 相关的关键字、语法结构形成一门 AOP 语言，其编译出来的程序是普通的 Java 字节码，因此，可以运行于任何 Java 平台，AspectJ 被誉为 AOP 领域的急先锋。

AspectWerkz:一个动态、轻量级、性能表现良好的 AOP 框架。可能通过使用配置文件、配合其提供的类加载器实现动态织入。该框架的当前已经与 AspectJ 合并，AspectJ5 就 合并后的成果。

JBoss-AOP：JBoos 公司开发的基于方法拦截及源码级数据的 AOP 实现框架，最开始属于 JBoos 服务器的一部分，可以脱离 JBoos 单独作为一个 AOP 框架使用。

Spring-AOP：Spring 框架中也提供了一个 AOP 实现，使用基于代理及拦截器的机制，与 Spring IOC 容器融入一体的 AOP 框架。Spring AOP 采用运行时织入方式，使得可以在基于 Spring 框架的应用程序中使用各种声明式系统级服务。

AOP 相关的框架或工具

除了上面介绍的几个 AOP 实现及框架以外，在 Java 领域，也有很多成熟的 AOP 相关技术，提供动态代理、拦截器、动态字节码生成及转换工具。下面简单列举一些用得比较多的：

ASM：一个轻量级的字节码生成及转换器。

BCEL：一个实用的字节码转换工具，在 JAC 中就是通过 BCEL 来实现方法拦截机制。

CGLIB：一个功能强大的动态代理代码工具，可以根据指定的类动态生成一个子类，并提供了方法拦截的相关机制，并且在大量的流行开源框架(如 Hibernate、Spring 等)中得到使用。

Javassist: JBoss 提供的 java 字节码转换器，在 JBoos 的很多项目中使用，包括 JBoss AOP。

5.1.6 AOP 在企业级应用程序中的作用

在企业级的应用程序中，有很多系统级服务都是横切性质的，比如事务处理、安全、对象池及缓存管理等。在以 EJB 为代表的重量级 J2EE 企业级应用中，这些系统级服务由 EJB 容器(即 J2EE 应用服务器)提供，Bean 的使用者可以通过 EJB 描述文件及服务器提供商的特定配置文件，声明式的使用这些系统服务。

而对于轻量级的应用中，由于业务组件都是多数是普通的 POJO，要使用这种声明式的

系统服务，则可以借助于 AOP 编程方法，借助 AOP 框架，通过切面模块来封装各种系统级服务模块，结合一些轻量级的容器应用，从而使得普通 POJO 业务组件也能享受声明式的系统服务。相对于 J2EE 应用服务器提供的几种固定的系统级服务来说，使用 AOP 方法可以自由定义、实现自己的系统级服务，因此变得更加灵活。比如，Spring 中的声明式事务支持、JBoss 中的缓存等都是使用 AOP 来实现的。

另外，如同前面分析的，在我们的系统中，除了一些系统级服务属于横切关注点问题以外，一些核心关注点外围的需求也会具有横切性质，因此还可以通过在程序中使用 AOP 来解决些具有横切性质的需求，使得系统设计更加容易、程序代码更加简洁、更加易于维护及扩展。

5.2 AspectJ 简介及快速入门

AspectJ 是一个基于 Java 语言扩展的 AOP 实现，被业界誉为 AOP 的急先锋，其提供了强大的 AOP 功能，其他很多 AOP 实现都借鉴或采纳其中的一些思想。学习使用 AspectJ 不但让我们可以直接在项目中使用他来解决横切关注点的问题，还可以通过他加深对 AOP 编程方法的认识及理解，由于 Spring2 中的 AOP 与 AspectJ 进行了很好的集成，因此也为我们学习使用 Spring2 中的 AOP 打下基础。

5.2.1 AspectJ 介绍

AspectJ 是 Java 语言的一个 AOP 实现，其主要包括两个部分，第一个部分定义了一套如何表达、定义面向切面(AOP)编程中的相关概念（如连接点、切入点、增强、切面等）的语法规范。通过这套语言规范，我们可以方便地用 AOP 来解决 java 语言中存在的交叉关注点问题。AspectJ 的另外一个部分是工具部分，包括编译器、调试程序的工具以及为了更方便开发基于 AOP 应用而设计的开发集成工具等。

AspectJ 是最早、功能比较强大的 AOP 实现之一，是为数不多的比较完整的 AOP 的实现，在 AOP 领域基本上充当着行业领头羊的角色，被誉为 AOP 领域的急先锋。AspectJ 的功能比较强大，从连接点、切入点、通知、引介到切面定义都有一套比较完整的机制，很多其它语言的 AOP 实现，也借鉴或采纳了 AspectJ 中很多设计。在 Java 领域，AspectJ 中的很多语法结构基本上成了 AOP 领域的标准，比如：在 Spring2.0 中，AOP 部分就作了比较大的调整，不但引入了对 AspectJ 的支持，其自己的 AOP 核心部分的很多使用方法、定义乃至表示等都力求保持与 AspectJ 一致。因此，要学习使用 AOP，我们有必要从 AspectJ 开始，因为他本身就是 java 语言 AOP 解决方案，就算不用 Spring，也可以独立地用于我们的 Java 应用程序中。

AspectJ 是 Eclipse 下面的一个开源项目，当前发布的版本是 AspcetJ5。

5.2.2 AspectJ 的下载及安装

AspectJ 的官方网址是：http://www.eclipse.org/aspectj/

要使用 AspectJ，首先需要下载并安装 AspectJ。直接进入其官网站，点击【downloads】栏目，在下载页面中选择 AspectJ 的一个版本，一般选择【Latest Stable Release】，然后点击后面 aspectj-xxx.jar 连接，即可进入下载页面，如“图 5-3”所示。

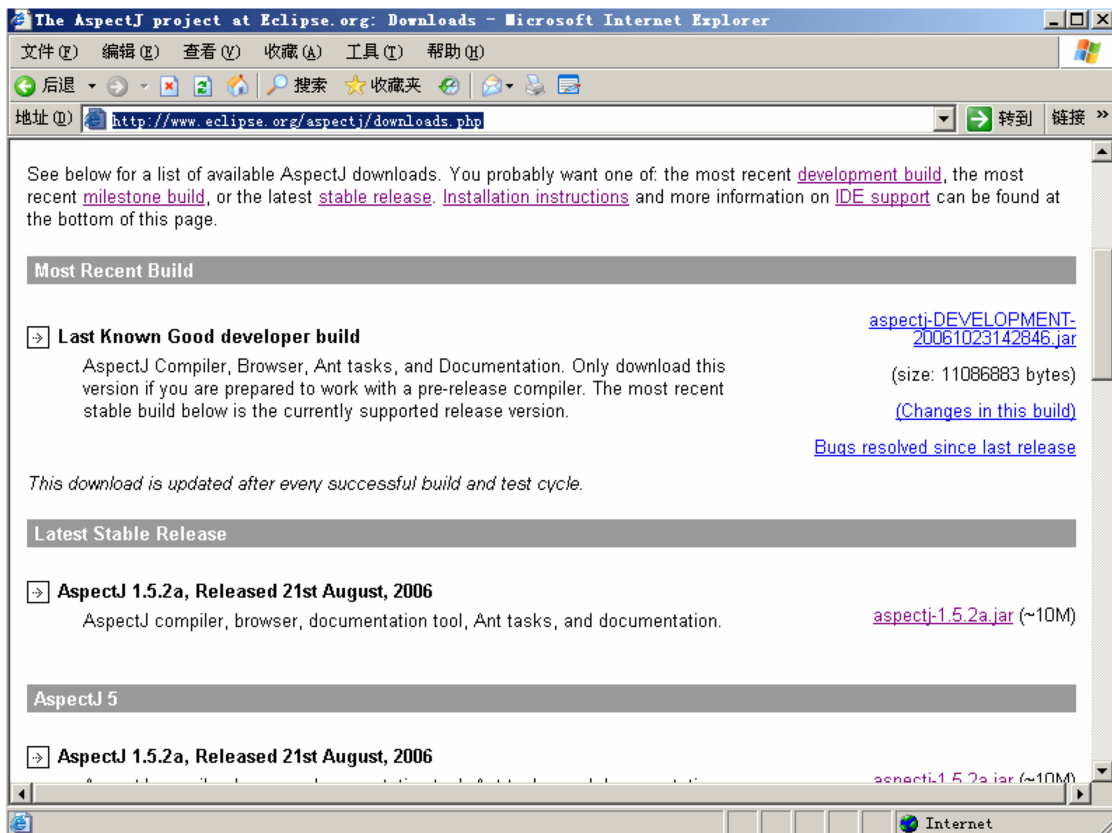

图 5-3 AspectJ 下载页面

下载得到的是一个形如 aspectj-xxx.jar 的文件，比如我们以当前比如新的 aspectj1.5 为例，我们得到一个 aspectj-1.5.2a.jar 文件。然后进入命令行，输入类似 java -jar D:\test\aspectj-1.5.2a.jar 的命令即可启动 AspectJ 安装程序，如“图 5-4”所示。

图 5-4 启动 AspectJ 安装程序

然后按照界面的提示，点击相应的按钮，开始按装。安装完成后，会出现类似“图 5-5”的界面：

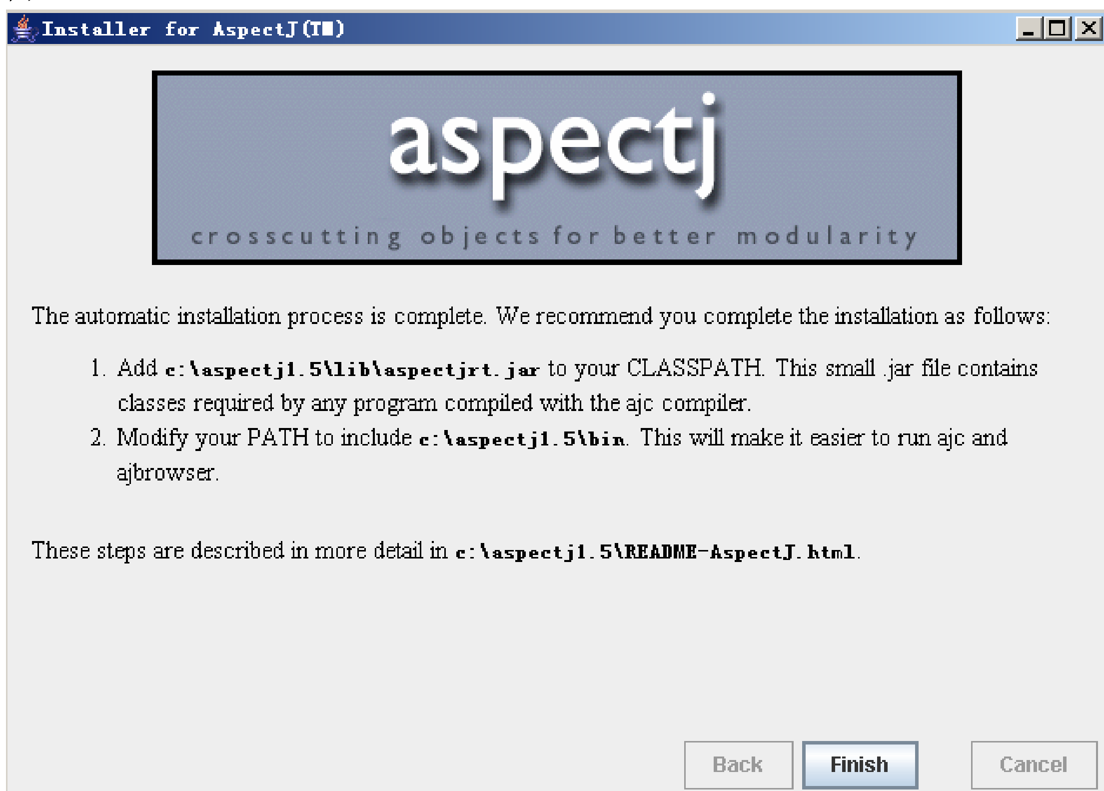

图 5-5 AspectJ 安装成功提示界面

“图 5-5”表示已经成功把 aspectj 安装到了指定目录，并建议我们把 aspectjrt.jar 文件添加到我们的 classpath 中，并把 AspectJ 的 bin 目录添加到操作系统的 path 中，这样以便于我们在任何目录使用 AspectJ 提供的工具及库文件，点击【finish】按钮完成安装！

安装完成后，切换到 aspectj 安装目录，可以看到 bin、lib、doc 三个目录，其中 bin 目录包含了 AspectJ 的编译器及相关调试工具等，lib 目录是编译 AspectJ 的程序时所要用到库文件，doc 目录是 AspectJ 的帮助、入门指南等文档及 AspectJ 应用示例代码目录。通过 doc 目录的文档及示例代码，我们可以快速学习及掌握 AspectJ 的用法。

当然，要在命令行很好的使用 AspectJ 的相关工具，需要设置一些环境变量。首先是把 lib 里面的 aspectjrt.jar 加到系统的 classpath 中，另外还要把 aspectj 主目录下的 bin 目录加到系统环境变量 path 中。分别如“图 5-6”及“图 5-7”所示：

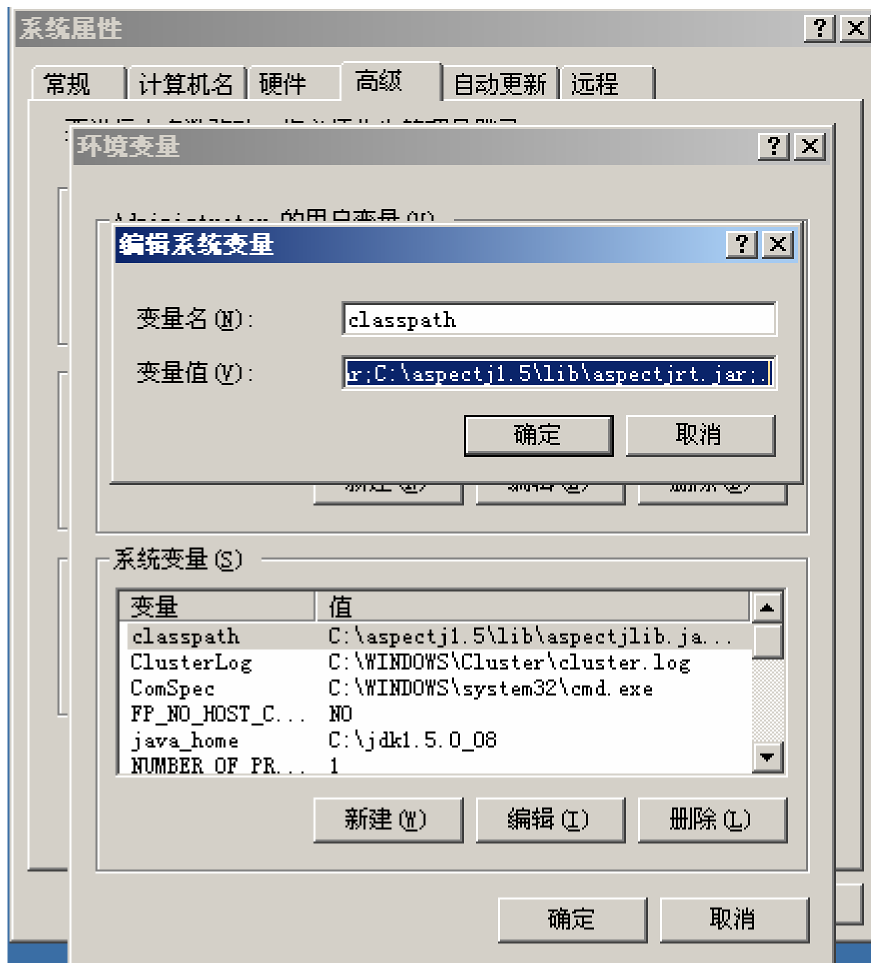

图 5-6 把 AspectJ 的相关 lib 添加到 classpath

图 5-7 把 AspectJ 主目录下的 bin 目录添加到系统 path 中

这样即完成了在 Windows 操作系统下 AspectJ 的手工安装。这时重新进入命令窗口，即可使用 AspectJ 的编译工具 ajc 命令来代替 javac 命令编译 java 源文件了。

5.2.3 在 Eclipse 中开发 AspectJ 程序

当然，在实际开发中，我们很少使用命令行来编译或调试程序了。一般情况下都是使用功能比较强大的专业 Java 开发工具及平台。AspectJ 除一套完整的语法规范以外，还提供在各种常用 java 开发工具开发 AspectJ 程序的插件，包括 Eclipse、JBuild、NetBeans、JDeveloper 等。在这里，我们简单讲解 AspectJ 与 Eclipse 集成应用。

首先需要下载并安装 AspectJ 的 Eclipse 插件 AJDT(AspectJ Development Tools)。跟安装其它 Eclipse 插件一样，有两种方法安装 AJDT，下面简要介绍。

第一种方法是直接到 AJDT 的官方网站 http://www.eclipse.org/ajdt/上面，根据自己的 Eclipse 版本，选择下载相应的版本的插件。下载下来的插件是一个形如 ajdt_1.4_for_eclipse_3.2.zip 的压缩文件，其中包含 features 及 plugins 两个目录，把这个压缩文件解压到 Eclipse 的主目录即可，然后重新进入 Eclipse，在 Eclipse 的【Preferences】面板中，即会看到一个【AspectJ Compiler】的选项，即表示 AJDT 已经正确安装。

第二种方法是直接使用 Eclipse 的插件自动更新功能来安装。直接点击 Eclipse 的【help】->【Software Updates】->【Find and Install... 】，即可进入插件自动更新/安装界面，点击界面上的【New Remote Site... 】按钮，然后在弹出的对话框中输入插件的名称，即 AJDT，在 URL 一栏输入自动更新 URL 地址，比如：http://download.eclipse.org/tools/ajdt/32/update，点"OK"按钮，开始插件安装，安装过程中会出现一些对话框，根据情况作相应的选择即可。如“图 5-8”所示：

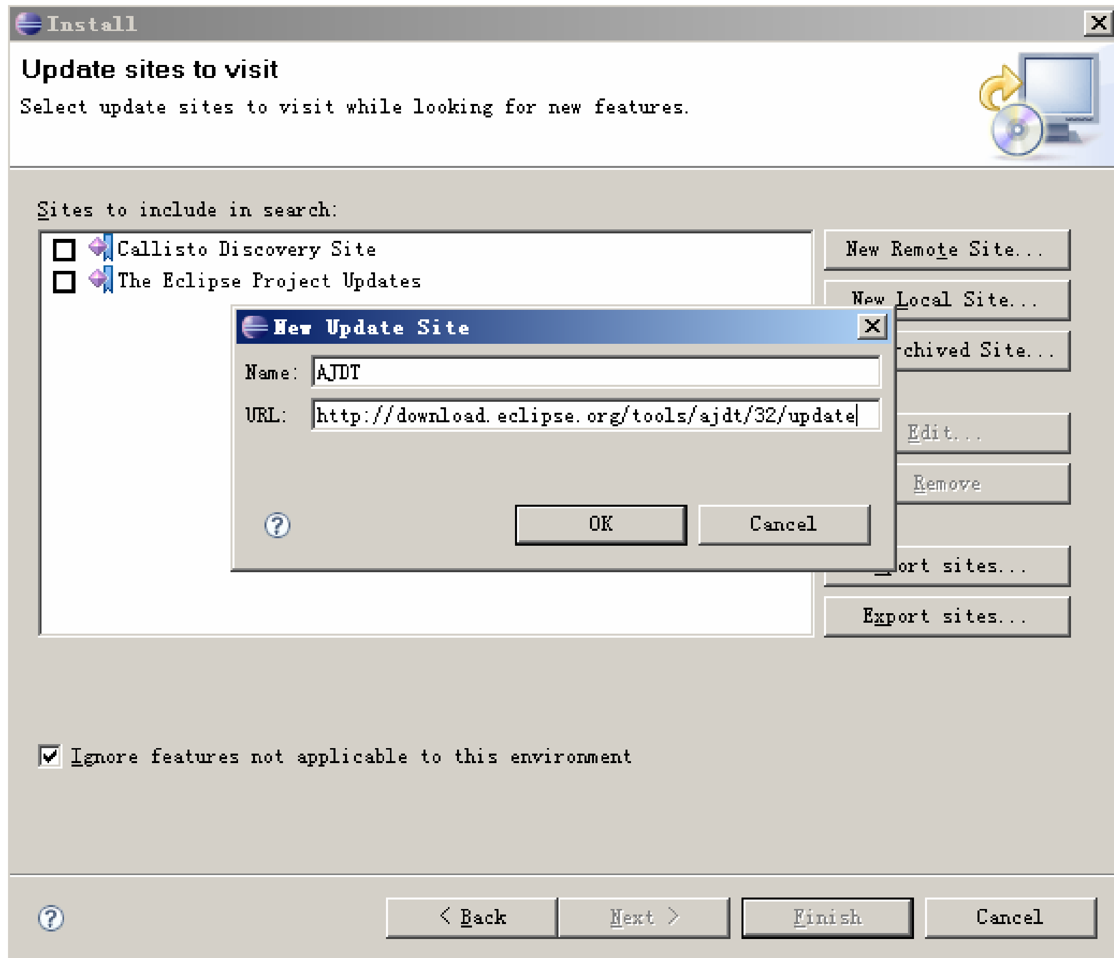

图 5-8 使用 Eclipse 自动更新功能来添加 AJDT

插件安装完成后，会要求重启动，启动后即会在 Eclipse 的【Preferences】面板中，即会看到一个【AspectJ Compiler】的选项。

插件安装完后，即可以直接在 Eclipse 新建建立 AspectJ 项目，如“图 5-9”，或者把一个已有的项目转为 AspectJ 项目，使得项目中可以支持 AspectJ 语法，并具具有可视化的切面信息标识，帮助我们更好的使用 AspectJ 进行 AOP 编程。

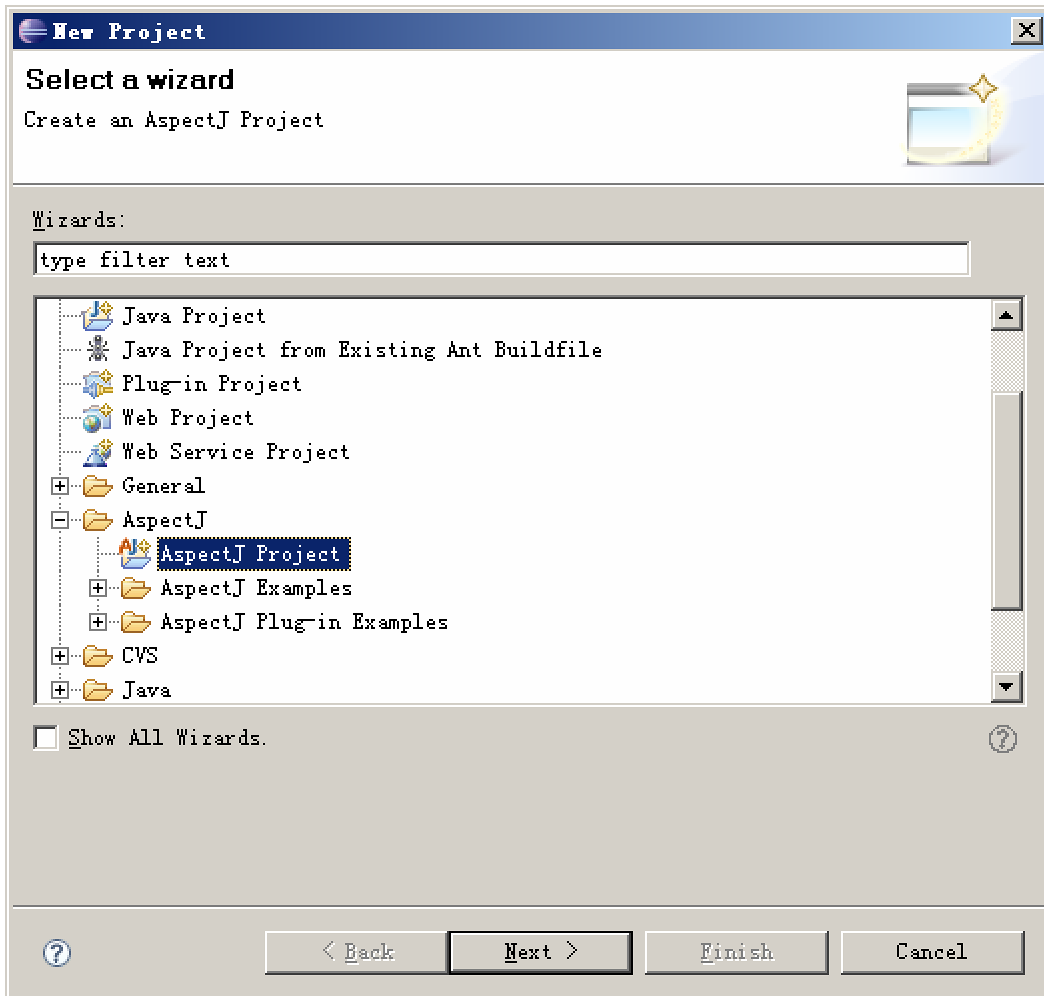

图 5-9 AJDT 安装成功后可用 Eclipse 来建立 AspectJ Project

5.2.4 AspectJ 版的 HelloWorld

下面演示在 Eclipse 中建立 AspectJ 版本 HelloWorld，首先需要安装 AspectJ 的 Eclipse 插件 AJDT。然后新建一个 AspectJ 工程,如“图 5-9”，然后新建一个 demo.Hello 类，内容如下：

```
package demo;
public class Hello {
public void sayHello()
{
    System.out.println("Hello AspectJ!");
}
public static void main(String[] args) {
    Hello h=new Hello();
    h.sayHello();
}
}
```

然后使用使用 Eclipse 新建一个名为 HelloAspect 的 Aspect 切面，如“图 5-10”所示。

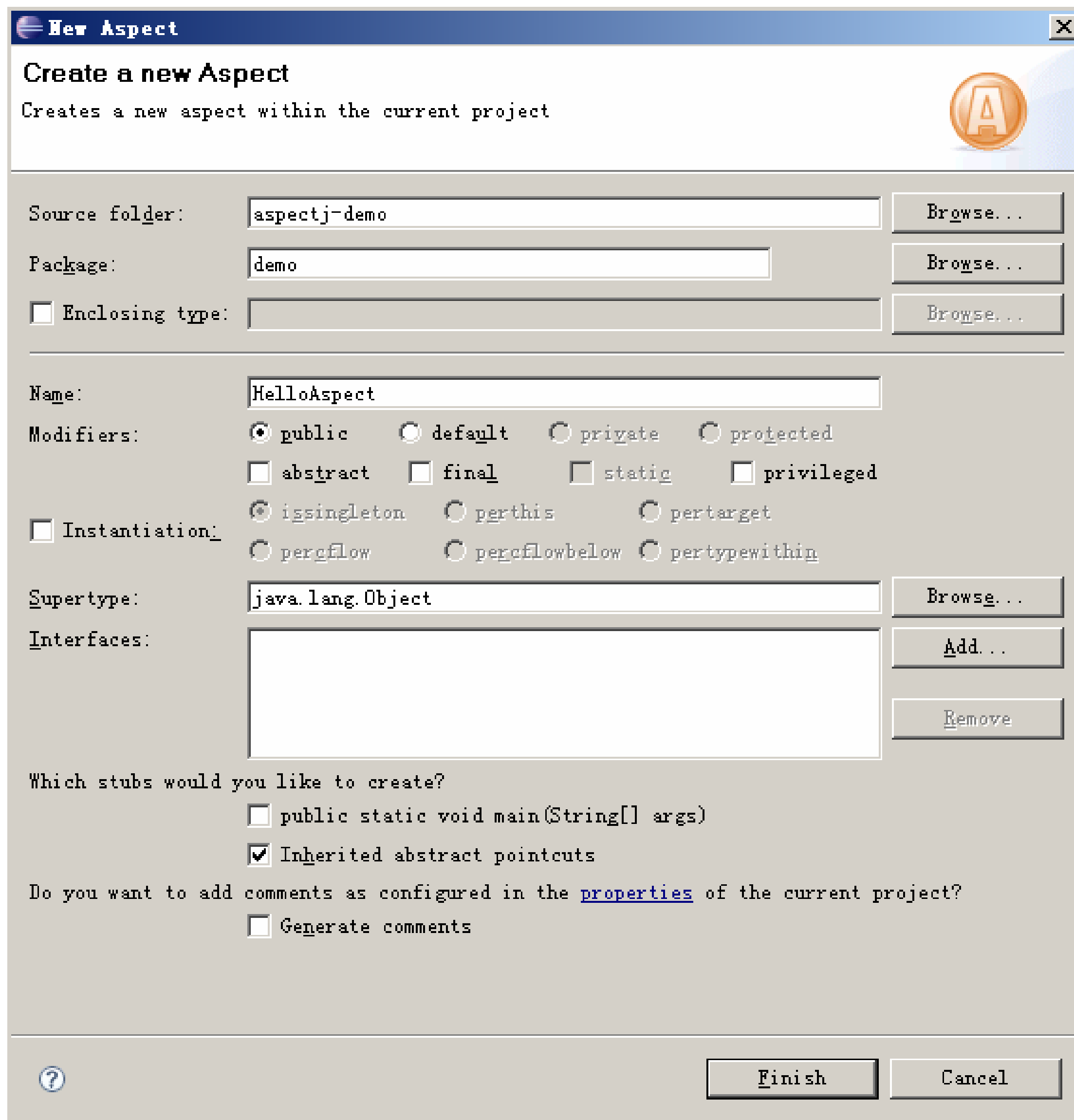

图 5-10 在 Eclipse 中新建 AspectJ 切面

在 HelloAspect 中定义了一个名为 somePointcut()的切入点，定义了两个增强，其中一个是在连接点前执行，一个是在连接点后执行。HelloAspect 的全部代码如下所示：

```
package demo;
public aspect HelloAspect {
    pointcut somePointcut():execution(* Hello.sayHello());
    before():somePointcut(){
        System.out.println("要准备说Hello了...");
    }
    after():somePointcut(){
        System.out.println("Hello已经说完！");
    }
    }
```

在 Hello 上点右键，使用【Run As】->【Java Application】运行 Hello，即会看到程序输出结果，如“图 5-11”所示。

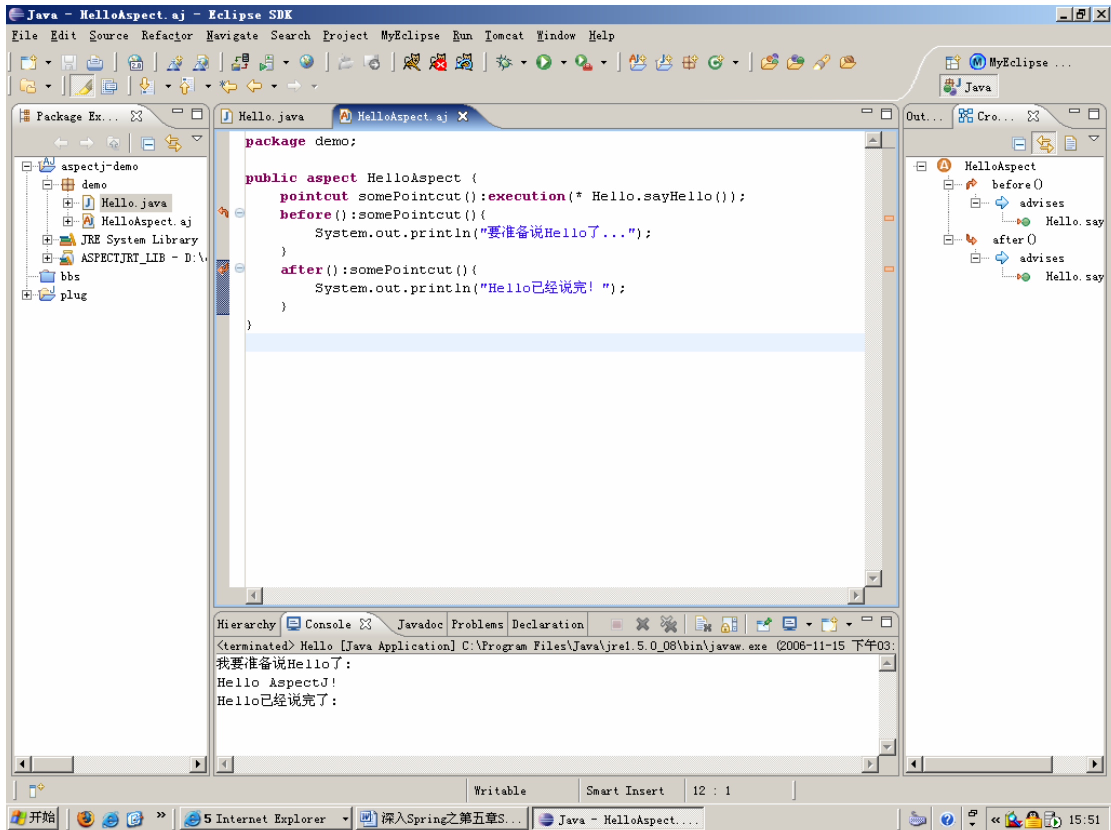

图 5-11 AspectJ 版 Hello 的项目总体图

5.2.5 AspectJ 中相关语法

前面说了，AOP 主要用来解决软件中交叉关注点的问题，AOP 实现要把交叉关注点的切面模块与系统的其他关注点进行组合，即把处理交叉关注点的功能按照一定规则或策略织入到核心关注点的模块中。这里“一定规则”是指什么，怎么来描述？这正是 AOP 实现的关键所在。在 AspectJ 中，通过一套完整的语言定义规范，来灵活、清晰地定义、描述这个织入过程中的“一定规则”。而我们程序员使用 AspectJ，也就是只需要掌握 AspectJ 的语言规范，然后按照规则写出适合我们的实际应用程序需求的“织入规则”，最后交给 AspectJ 的编译器负责按照这些织入规则及策略来把位于切面中处理交叉关注点的模块与其他关注点的模块组合到一起，即可实现灵活、复杂的软件功能。

我们首先来看 AspectJ 中的关于 AOP 一些概念的定义及表示方法：

■ **切面(Aspect)**

在 AspectJ 中，切面封装了切入点、通知及通知的实现代码，切面中还可以声明改变一个类的继承关系、给一个类添加属性、方法、构造函数，指定一个现有类实现一个接口等等。切面是一个独立的模块单元，跟普通的 java 类一样，切面中还可以定义自己属性、定义方法。一个切面一般写在一个以 aj 为扩展名的文件中，切面的定义根据 AspectJ 切面初始化的方式及生命周期的不同，有如下几种形式：

[modifier] aspect aspectName{ ... }

[modifier] aspect aspectName issingleton() { ... }

[modifier] aspect aspectName perthis(Pointcut) { ... }

[modifier] aspect aspectName pertarget(Pointcut) { ... }

[modifier] aspect aspectName percflow(Pointcut) { ... }

[modifier] aspect aspectName percflowbelow(Pointcut) { ... }

[modifier] privileged aspect aspectName { ... }

在上面的格式中，方括号"[]"中的内容是可省的，一般情况下都不需要，[modifier]可以是 abstract、public 及 final。aspect 是表示切面的关键字，aspectName 表示切面的名称，aspectName 后面的关键字如 issingleton 等用来标识不同的切面初始化方式及生命周期。

下面是一个 AspectJ 切面源文件内容：

```
public aspect AspectDemo {
//切面里面的属性
private int times=0;
//定义一个切入点
pointcut somePointcut():call(* Component.*(..));
//给切入点somePointcut定义一个通知
after():somePointcut(){
    this.times++;
    System.out.println("执行了内中
的："+thisJoinPoint.getSignature().getName());
    this.print();
}
//切面中定义方法
private void print()
{
    System.out.println(this.times);
}
}
```

对于 OOP 编程来说，我们主要是针对类 class 来编程，把一个类相关的属性、方法、构造子等都封装到了类中。而对于 AOP 编程来说，主要就是对切面 Aspect 编程，也就是把切面相关连接点、切入点、通知以及实现、引介等都封装到切面中。通过前面的 Helo 及上面的示例，我们对 AspectJ 有了一个初步的印象，接下来我们将对 AspectJ 中如何实现连接点、切入点、通知、引介等分别作介绍。

■ **连接点(Join point)**

连接接点是指程序中的某一个点。在 AspectJ 中，连接点分得非常细致，如一个方法、一个属性、一条语句、对象加载、构造函数等都可以作为连接点。AspecJ 中的连接点主要有下面的几种形式：

方法调用(Method Call)－方法被调用的时；

方法执行(Method execution)－方法体的内容执行的时；

构造函数调用(Constructor call)－构造函数被调用时；

构造函数执行(Constructor execution)－构造函数体的内容执行时；

静态初始化部分执行(Static initializer execution)－类中的静态部分内容初始化时；

对象预初始化(Object pre-initialization)，主要是指执行构造函数中的 this()及 super()时；

对象初始化(Object initialization)－在初始化一个类的时候；

属性引用(Field reference)－引用属性值时；

属性设值(Field set)－设置属性值时；

异常执行(Handler execution)－异常执行时；

通知执行(Advice execution)－当一个 AOP 通知(增强)执行时。

在 AspectJ 中，连接点的表示使用系统提供的关键字来表达，比如，call 来表示方法调用连接点，使用 execution 来表示方法执行连接点。连接点不会单独存在，需要与一定的上下文结合，而是在原始切入点中包含连接点的表述。

■ **切入点(Pointcut)**

切入点是用来表示在连接点的何处插入切面模块，也可以称为一组连接点在一定上下文环境中的集合。AspectJ 中的切入点主要有两种，一种是最基本的原始切入点，另外一种是由基本切入点组合而成的切入点。原始切入点是对连接点在特定上下文的表述，通过连接点的关键字以及一定的格式来声明。下面简单介绍一些 AsepctJ 中的原始切入点：

(1)、方法相关的切入点

call(MethodPattern)

execution(MethodPattern)

(2)、属性相关的切入点

get(FieldPattern)

set(FieldPattern)

(3)、对象创建相关的切入点

call(ConstructorPattern)

execution(ConstructorPattern)

initialization(ConstructorPattern)

preinitialization(ConstructorPattern)

(4)、类初始化相关的切入点

staticinitialization(TypePattern)

(5)、异常处理相关的切入点

handler(TypePattern)

(6)、通知(增强)相关的切入点

adviceexecution()

(7)、基于状态的切入点

this(Type or Id)

target(Type or Id)

args(Type or Id or "..", ...)

(8)、控制流程相关的切入点

cflow(Pointcut)

cflowbelow(Pointcut)

(9)、程序内容构相关的切入点

within(TypePattern)

withincode(MethodPattern)

withincode(ConstructorPattern)

(10)、语句相关的切入点

if(BooleanExpression)

在 AspectJ 中，切入点一般在源代码中通过一条代有签名性质的语句来声明，如下面的例子：

pointcut anyCall() : call(* *.*(..));

pintcut 是切入点的声明的关键字，anyCall 是我们自己定义的切入点名称，“:”号后面

是原始切入点或多个原始切入点的组合。跟其它 java 语句一样，切入点声明以“;”结束。

在 Java5 及以及的版本中，也可以在代码中使用注解来标识切入点。如下面的例子：

```
@Pointcut("call(* *.*(..))")
        void anyCall() {}
```

@Pointcut 是切入点的注解标签，参数中的内容为原始切入点或切入点表达式。下面定义的方法 anyCall 表示切入点的名称，需要用一对大括号“{}”把其括进来，也即一个空的方法体。

切入点的签名及表述遵循固定的语法格式及规范，下面是 AspectJ 中一些常用切入点签名语法格式：

(1)、与方法相关切入点签名语法

call/execution/withincode(MethodPart)－方法调用/方法执行/在方法体内，MethodPart 代表方法签名，格式如下：

[Modifier] Type [ClassType.] methodName(ArgumentType1...N...) [throws ExceptionType]

[]中的内容为可选择的内容，Modifier 表示修饰符，如 private、public 等，type 表示返回值类型，ClassType 表示类名称，methodName 表示方法名称，ArgumentType 表示参数类型及顺序，ExceptionType 表示异常类型！如下面切入点表示调用 UserService 类中的所有返回值为 void 的公开方法：

call("public void UserService.*(..)");

(2)、与构造子相关的签名语法

call/execute/initialization/preinitialization/withincode(ConstructorPart)－表示构造子调用/构造子执行/对象初始化/对象预初始化/在构造子体内。ConstructorPart 代表构造子部分，其格式如下：

[Modifier] [ClassType.]new(ArgumentType1...N...) [throws Throwabe]

(3)、与属性相关的签名语法

get/set(FieldPart)－属性引用/属性设值。FieldPart 部分的格式如下：

[Modifier] Type [ClassType.] fieldName

(4)、与上下文件相关的签名语法

this(Type|var)－传递当前切入点对象；

target(Type|var)－传递连接点所属的目标对象；

args(Type|var)－传递上下文参数；

另外还有与异常相关的 handler(TypePart)，与包范围的 within(somePackage)，与表达式相关的 if(Expression)，与切入点控制流程相关的 cflow/cflowbelow(Poincut)，与注解相关的 @annotation(Type|Var)等等。

■ 组合切入点

在 AOP 程序中，我们可以直接使用单个原始切入点，有时候需要把几个原始切入点通过一定的组合，形成一个更加适合特定条件的切入点。AspectJ 中可以使用集合运算，把原始切入点有机的组合到一起。切入点组合运算主要包括&&“and(与)”、||“or(或)”、!“not(非)”三种。如下所示：

```
pointcut someCall():(call(void buss*(..))||call(void save*(..)))
&& within(springroad.demo.service.*);
```

把三个原始切入点通过&&与||操作符组合起来，形成一个表示在 springroad.demo.service 包内，名称与 buss 或 save 开头，返回值为 void 的用有方法调用切入点。再看下面的组合切

入点：

```
pointcut  supperRole(Soldier s): target(s)&&execution(boolean Soldier.canTreat());
```

使用&&把两个切入点连接起来，得到一个带有目标对象作为上下文件传送参数的组合切入点。

■ **增强(Advice)**

增强，也称为通知，是指在切入点里执行的具体程序逻辑，也即切入点中执行什么操作，交叉关注点中需要实现的程序功能。在 AspectJ 中，也有很多专用于定义通知的关键字，通知的定义如下：

[strictfp] AdviceSpec [throws TypeList] : Pointcut { Body }

AdviceSpec 表示具体的通知类型，具体的切面逻辑写在{}中。AspectJ 主要有以下几种通知：

before(Formals)－前置通知，在切入点前执行；

after(Formals) returning [(Formal)] －后置通知，在切入点正常返回后执行；

after(Formals) throwing [(Formal)] －异常后通知，在切入点出现异常时执行；

after(Formals) －final 后通知，在切入点执后执行；

Type around(Formals)－环绕通知，在切入点任意位置执行，需要手动调用连接点。

我们来看一个例子：

```
after() returning(boolean value) :doRecord()
 {
//执行相关功能
System.out.println("执行切面逻辑!");
}
```

表示在切入点 doRecord 所描述的连接点正常执行并返回后，还要执行 after 通知中的代码，本例输出一个“执行切面逻辑!”。

通知“:”后面的切入点可以是使用 pointcut 关键字定义的切入点，也可以直接写原始切入点或组合切入点，Formals 可以用来代表一些参数定义，看下面的例子：

```
after(Soldier s)returning(boolean value):target(s)&&call(boolean Soldier.canTreat())
 {
  if(value)System.out.println(s.getName()+"得到治疗！ ");
  }
```

定义了一个作用于 Soldier 类的 canTreat 方法的通知，并且可以直接在通知程序代码中通过 s 这一参数，得到连接点所属目标对象。

■ **使用 thisJoinPoint**

在一个 Java 类中，我们可以在方法中使用 this 关键字来引用当前对象。同样在 AspectJ 的通知实现中，使用 thisJoinPoint 可以得到当前连接点的相关信息，比如对于方法连接点来说，可以得到方法名，方法参数等等。如下面的例子，可以输出当前连接点的相关信息：

```
after():someCall()
{
    System.out.println(thisJoinPoint);
    }
```

■ 引介(Introduction)

引介是指不改变现有类的情况下，给现有类增加属性、方法，或让现有类现实指定接口、或继承某个类等，引介改变了类的静态结构。AspectJ 中的引介功能比较强大，这里简单介绍其中一些常用功能。

(1)、增加内部成员

通过引介可以很容易在不修改已有类的代码，就给一个现有类增加属性、方法、构造函数等内部成员。

增加方法：

[Modifiers] Type OnType . Id(Formals) [ThrowsClause] { Body }

abstract [Modifiers] Type OnType . Id(Formals) [ThrowsClause] ;

如下面的例子给 Soldier 增加了一个名为 exit、返回值为 void 的公共方法:

```
public void Soldier.exit(){
    System.out.println("退出战场!");
    }
```

增加属性：

[Modifiers] Type OnType . Id = Expression;

[Modifiers] Type OnType . Id;

如下面的例子给 Soldier 增加了一个名为 nickName 的属性：

```
private String Soldier.nickName;
```

也可以在定义的时候初始化属性，如：

```
private String Soldier.nickName="游客";
```

增加构造子：

[Modifiers] OnType . new (Formals) [ThrowsClause] { Body }

如下面的例子给 Soldier 增加一个带有参数构造子：

```
public Soldier.new(String userName)
  {
    //初始化
    }
```

(2)、实现接口

AspectJ 可以在切面中使用 declare parents 关键字让一个现有的类实现某一个接口，语法如下：

declare parents: TypePattern implements TypeList;

其中 TypePattern 是指现有的类；当声明了实现接口以后，需要在切面中定义接口的实现逻辑。如下面的例子，我们让 Soldier 实现一个 Comparable 接口：

```
declare parents:Soldier implements java.lang.Comparable;
public int Soldier.compareTo(Object o) {...}
```

(3)、指定继承

AspectJ可以在切面中用declare parents关键字，给一个现有的类指定一个父类，实现多重继承。格式如下：

declare parents: TypePattern extends Type;

如下面的例子，给Soldier指定了一个父类UserInfo，这样Soldier就有了UserInfo类的特性及功能：

```
declare parents:Soldier extends springroad.demo.UserInfo;
```

另外AspectJ中还有其它一些引入功能,请参考最新的AspectJ文档。

■ 织入(weaving)

在写好一个切面模块后，需要把切面模块与系统中的其它模块进行组合，形成一个完整的程序，这一过程称为织入。在 AspectJ 中，支持两种织入方式，即编译器织入及类加载器织入。编译器织入是指直接使用 AspectJ 提供的编译器取代普通的 javac 命令来编译整个应用程序。AspectJ 中使用前面安装过程中介绍的 ajc 命令，ajc 的命令使用跟 javac 命令差不多，只是有一些参数略有差别。ajc 可以对所有源代码一起编译(Compile-time weaving），也可以把切面源代码与已经编译好的 class 文件或 jar 包一起编译，进行织入(Post-compile weaving)。当然，若我们在开发工作中使用 AspectJ 提供的插件，插件中就自带 AspectJ 编译器，不需要使用命令符。

由于 AspectWerkz 合并入了 AspectJ，因此合并后的 AspectJ5 还支持类加载器，类加载器织入是指使用 AspectJ 提供的类加载器，取代普通的 java 类加载器，由类加载在加载 class 到系统虚拟机中的时候进行织入操作。AspectJ5 提供两个命令脚本 aj 及 aj5，可以用来取代普通的 java 命令，运行需要进行织入的程序，具体的织入参数一般配置在一个名为 aop.xml 文件中。AspectJ5 中负责处理类加载器织入的包是 aspectjweaver.jar，在 Spring AOP 中，也是通过类加载器织入的方式，来达到与 AspectJ 的完全集成及支持。

5.2.6 一个简单的回合格斗小游戏示例

下面，我们使用一个简单的回合格斗的小游戏，来演示 AspectJ 的应用。这个示例主要设计了一个战士 Soldier 类，这个类包括发动攻击、治疗、躲避、移动等功能。另外有一个充当客户端的主程序 MainTest，里面的功能就是让两个战士回合制互相攻击，直到一个被倒下。

核心类 Soldier 的源码如下：

```
public class Soldier {
private String name;
private int health=100;
private int damage=10;
private int x=10;
private int y=10;
//攻击其它角色
public boolean attack(Soldier target){
    boolean ret=false;
    if(!target.dodge())//目标是否躲闪成功
    {
        target.setHealth(target.getHealth()-this.damage);
        ret=true;
    }
    move();    //移动一下
    treat();//冶我疗伤
    return ret;
}
public void move()
```

```
{
    this.x+=getRandom(5);
    this.y+=getRandom(5);
}
//躲避x及y随机变动，成功率为50%
public boolean dodge()
{
    return getRandom(10)%2==0;
}
//治疗，具有一定成功的机会，可以提高生命值0-20点
public void treat()
{
    if(canTreat())
        this.health+=getRandom(20);
}
public boolean canTreat()
{
    return getRandom(10)/2==0;
}

private int getRandom(int seed)
{
    return RandomUtil.getRandomValue(seed);
}

//getter及setter方法
public int getHealth() {
    return health;
}
public void setHealth(int health) {
    this.health = health;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public int getX() {
    return x;
}
public void setX(int x) {
    this.x = x;
}
```

```
public int getY() {
    return y;
}
public void setY(int y) {
    this.y = y;
}
public int getDamage() {
    return damage;
}
public void setDamage(int damage) {
    this.damage = damage;
}
}
```

Soldier 引用了一个随机数生成工具类 RandomUtil，用于模拟一定的发生概率，代码如下：

```
public class RandomUtil {
private static java.util.Random random=new java.util.Random();
public static int getRandomValue(int seed)
{
    return random.nextInt(seed);
}
    }
```

然后就是使用 Soldier 的客户端程序 MainTest，这里是一个简单的控制台程序，代码如下：

```
public class MainTest {
    public static void main(String[] args) {
    Soldier p1=new Soldier();
    p1.setName("角色 1 ");
    Soldier p2=new Soldier();
    p2.setName("角色 2 ");
    int i=0;
    while(p1.getHealth()>0 && p2.getHealth()>0)
    {
        p2.attack(p1);
        p1.attack(p2);
        i+=2;
    }
    System.out.println("战斗次数: "+i);
    if(p1.getHealth()>0)System.out.println("角色 1 战胜!");
    else System.out.println("角色2战胜!");
    }
}
```

这三个类组成了一个完成的应用程序，执行 MainTest，你会发现经过一会儿的战斗以后，在控制台会输出战斗的结果。

现在由于我们需要观察两个角色的详细战斗情况，也就是 attach 的方法执行情况，包括何时，对谁发动攻击，攻击结果等，另外还想给 Soldier 加入一个 Boss 级角色，Boos 角色的疗伤 treat 的成功率为 100%。

由于各种原因，我们不能直接更改 Soldier 的源代码（毕竟，在其相关的方法中直接添加输出语句，就好比让战士每发动一次攻击都需要自己记录一次战斗情况，这在激烈的即时战斗中肯定是不科学的。）。为此，我们想到 AOP，通过在 AOP 切面模块中实现观察战斗情况的功能。想从什么角度观察，观察哪些内容，都是由切面模块来定义，对 Soldier 的核心功能不影响。

除了观察详细战斗情况以外，我们还会对 Soldier 的一些方法进行进行切入，引入 Boss 级角色。

设计一个 AspectJ 的切面 RecordGame，来处理战斗详情输出及引入 Boss 角色的功能，RecordGame.aj 的全部内容如下：

```
public  aspect RecordGame {
   private static java.text.SimpleDateFormat df=new
java.text.SimpleDateFormat("yyyy-MM-dd H:m:s");
    pointcut doRecord():execution(boolean
Soldier.attack(Soldier));
    pointcut  supperRole(Soldier s): target(s)&&execution(boolean
Soldier.canTreat());
    after() returning(boolean value) :doRecord()
    {
     Soldier s=(Soldier)thisJoinPoint.getTarget();
     Soldier t=(Soldier)thisJoinPoint.getArgs()[0];
     System.out.println(df.format(new
java.util.Date())+":"+s.getName()+"  向  "+t.getName()+"  发动了一次
攻击！——结果："+(value?"成功":"失败"));
     System.out.println(s.getName()+"生命值：
"+s.getHealth()+";"+t.getName()+"生命值："+t.getHealth());
    }
    after(Soldier s)returning(boolean
value):target(s)&&call(boolean Soldier.canTreat())
    {
     if(value)System.out.println(s.getName()+"得到治疗！");
    }
    boolean around(Soldier s): supperRole(s)
    {
      if("super".equals(s.getName())) return true;
      else return proceed(s);
    }
   }
```

在上面的代码中，定义了两个切入点 doRecord()及 supperRole(Soldier s)，doRecord 用来切入 Soldier 的 attach 方法，supperRole 用来切入 Soldier 的 canTreat 方法。

第一个后置增强实现用来输出战斗情况，使用 AspectJ 的 thisJointPoint 关键字，得到连

接点上的目标对象以及方法参数，根据返回值 value 来判断攻击是否成功，最后输出交战双方的生命值，实现了对战斗情况的详细观察。

第二个后置返回增强用于根据 canTreat 方法的返回情况，输出是否得到成功治疗的信息。

第三个增强是环绕增强，用于给系统加入 Boss 角色的判断功能，这里只是通过角色的名称进行判断，若角色名称为 super，则将跳过 canTreat 的其它部分，直接返回 true，使得治疗成功率为 100%，否则正常执行 canTreat 方法。

可以使用 AspectJ 提供的编译器编译 4 个文件，假如我们的示例存放在包 springroad.demo.chap5 中，可以命令行执行下面的命令：

ajc springroad\demo\chap5*.*j*

编译成功后，使用普通 java 命令运行程序，这时就可以看到程序详细的战斗情况记录了，如“图 5-12”所示。

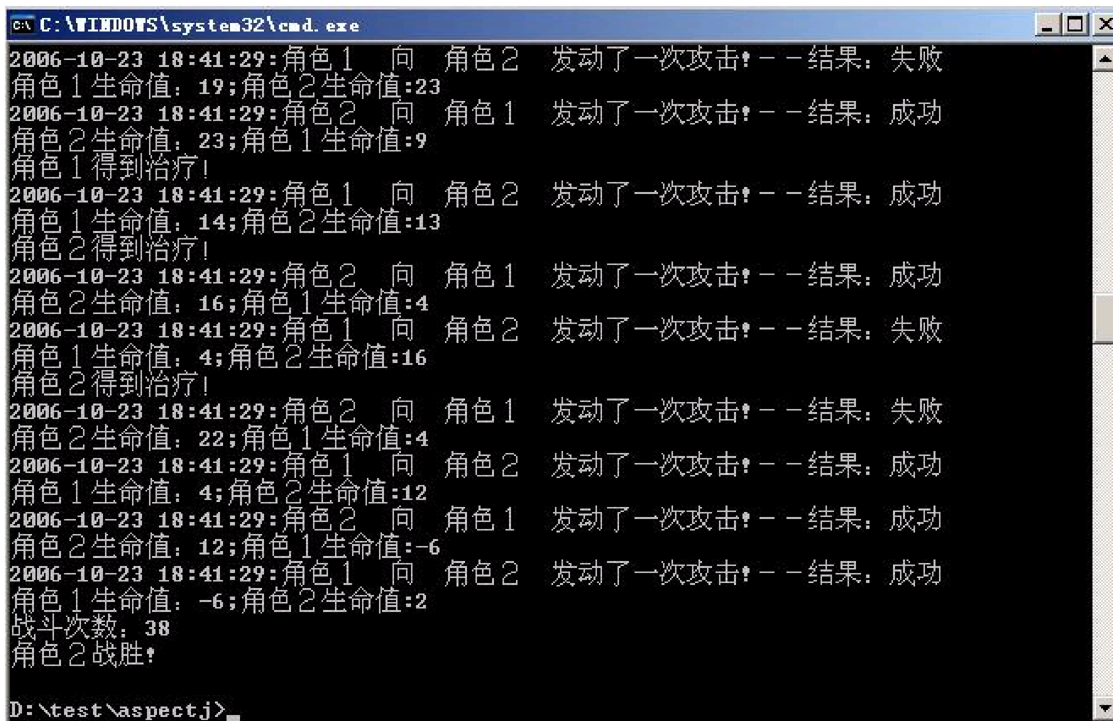

图 5-12 回合格斗小游戏运行结果截图

我们还可以把客户端程序 MainTest 中的某一个角色的名称设置为"super"，这样连续运行多次 MainTest，会发现 super 的胜率要大得多。

5.3 一个简单的 Spring AOP 示例

本节是 Spring AOP 的入门示例教程，主要是为了演示使用 Spring AOP 编程的基本步骤及使用方法，建议新手按照相关的步骤进行练习，先对 Spring 的 AOP 有一个感性认识。本节主要提供一个简单例子，演示 Spring2.0 中 AOP 的配置及使用方法，并与 AspectJ 中的的使用进行简单的对比及分析。我们使用了本章开篇提出来的示例，并按一般 J2EE 应用的开发流程来演示，这里主要是讲解使用方法，因此省略一些与 Spring AOP 不相关的讲述。

5.3.1 定义业务组件

设计系统的核心业务组件。基于针对接口编程的原则，一个好习惯是先使用接口来定义业务组件的功能，下面使用 Component 来代表业务组件接口。

Component.java 代码如下：

```
package springroad.demo.chap5.exampleB;

public interface Component {
     void business1();//商业逻辑方法1
     void business2();//商业逻辑方法2
     void business3();//商业逻辑方法3
    }
```

写一个 Component 的实现 ComponentImpl 类，ComponentImpl.java 代码如下：

```
package springroad.demo.chap5.exampleB;
public class ComponentImpl implements Component {
    public void business1() {
        System.out.println("执行业务处理方法1");
    }
    public void business2() {
        System.out.println("执行业务处理方法2");
    }
    public void business3() {
        System.out.println("执行业务处理方法3");
    }
}
```

写一个 Spring 的配置文件，配置业务 Bean。aspect-spring.xml 的内容如下：

```
<?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/bean
s
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd"
>
    <bean id="component"
        class="springroad.demo.chap5.exampleB.ComponentImpl">
    </bean>
    </beans>
```

然后写一个用于在客户端使用业务 Bean 的 ComponentClient 类,代码如下：

```
package springroad.demo.chap5.exampleB;
import org.springframework.context.ApplicationContext;
public class ComponentClient {
    public static void main(String[] args) {
```

```java
    ApplicationContext context=new
org.springframework.context.support.ClassPathXmlApplicationContex
t("springroad/demo/chap5/exampleB/aspect-spring.xml");
    Component component=(Component)context.getBean("component");
    component.business1();
    System.out.println("-----------");
    component.business2();
    }
    }
```

运行程序，我们可以看到结果输出为：

```
执行业务处理方法1
-----------
执行业务处理方法2
```

这个业务 Bean 只是简单的执行业务方法中代码，现在由于企业级应用的需要，我们需要把业务 Bean 中的所有 business 打头所有方法中的业务逻辑前，都要作一次用户检测、启动事务操作，另外在业务逻辑执行完后需要执行结束事务、写入日志的操作。直接修改每一个方法中的代码，添加上面的逻辑，前面已经说过存在不可维护等诸多问题，是不可取的。

由于安全检测、事务处理、日志记录等功能需要穿插分散在各个方法中，具有横切关注点的特性，因此我们想到使用 Spring 的 AOP 来实现。

5.3.2 使用基于 Schema 的配置文件配置 Spring AOP

定义一个用于处理横切交叉关注点问题的切面模块，Spring AOP 使用纯 Java 的方式来实现 AOP 的，因此我们使用一个名为 AspectBean 的类来处理上面所说的问题。

作为示例，AspectBean.java 中的内容如下：

```java
package springroad.demo.chap5.exampleB;
public class AspectBean {
    public void validateUser()
    {
        System.out.println("执行用户验证!");
    }
    public void writeLogInfo()
    {
        System.out.println("书写日志信息");
    }
    public void beginTransaction()
    {
        System.out.println("开始事务");
    }
    public void endTransaction()
    {
        System.out.println("结束事务");
    }
```

}

(在实现应用中，用户验证、日志记录、事务处理都应该是在上面的方法中调用专门的模块来完成。另外，还要考虑很多问题，比如与连接点上下文相关的目标对象、参数值等。)

有了处理横切交叉问题的切面模块 Bean，下面我们就可以在 Spring 的配置文件中进行 Spring AOP 相关的配置了。把 aspect-spring.xml 改成如下内容：

```
<?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/bean
s http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
    <aop:config>
        <aop:aspect id="aspectDemo" ref="aspectBean">
            <aop:pointcut id="somePointcut"
                expression="execution(*
springroad.demo.chap5.exampleB.Component.business*(..))" />
            <aop:before pointcut-ref="somePointcut"
                method="validateUser" />
            <aop:before pointcut-ref="somePointcut"
                method="beginTransaction" />
            <aop:after-returning pointcut-ref="somePointcut"
                method="endTransaction" />
            <aop:after-returning pointcut-ref="somePointcut"
                method="writeLogInfo" />
        </aop:aspect>
    </aop:config>
    <bean id="aspectBean"
        class="springroad.demo.chap5.exampleB.AspectBean">
    </bean>
    <bean id="component"
        class="springroad.demo.chap5.exampleB.ComponentImpl">
    </bean>
    </beans>
```

上面配置文件中的黑体部分是增加的内容，原来与业务 Bean 相关的配置不变。

为了能正确运行客户端代码，需要把 Spring 项目 lib 目录下 aspectj 目录中的 aspectjweaver.jar 文件添加到 classpath 中。

不需要重新编译客户端代码，直接运行示例程序 ComponentClient，会看到如下的内容输出：

执行用户验证！

```
开始事务
执行业务处理方法1
结束事务
书写日志信息
-----------
执行用户验证！
开始事务
执行业务处理方法2
结束事务
书写日志信息
```

由此可见，在客户调用业务 Bean Component 中的 business1 及 business2 的方法时，都自动执行了用户验证、事务处理及日志记录等相关操作，满足了我们应用的需求。

5.3.3 使用 Java5 注解配置及使用 Spring AOP

Spring2 中的 AOP 提供了使用 AspectJ 中定义的 Java 注解在 Bean 中配置切入点及通知的方法。这里演示演示这种使用方法，我们写一个包含使用了 Java 注解来标识切面相关信息的 Bean，方法名称及内容跟上面 AspectBean 的完全一样，AspectAnnotationBean.java 中的内容如下所示：

```
package springroad.demo.chap5.exampleB;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.After;
@Aspect
public class AspectAnnotationBean {
@Pointcut("execution(* springroad.demo.chap5.exampleB.Component.business*(..))")
    public void somePointcut()
    {
    }
    @Before("somePointcut()")
    public void validateUser()
    {
        System.out.println("执行用户验证!");
    }
    @After("somePointcut()")
    public void writeLogInfo()
    {
        System.out.println("书写日志信息");
    }
    @Before("somePointcut()")
    public void beginTransaction()
    {
```

```
        System.out.println("开始事务");
    }
    @After("somePointcut()")
    public void endTransaction()
    {
        System.out.println("结束事务");
    }
}
```

其中黑体部分的内容是相对于前面示例中 AspectBean 增加的。现在我们来修改 Spring 的配置文件，修改后 aspect-spring.xml 的内容如下：

```
<?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/bean
s http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
    <aop:aspectj-autoproxy />
    <bean id="aspectBean"

    class="springroad.demo.chap5.exampleB.AspectAnnotationBean">
    </bean>
    <bean id="component"
        class="springroad.demo.chap5.exampleB.ComponentImpl">
    </bean>
    </beans>
```

对比上一个配置文件，我们看到这个文件比前面一个简单多了，关于 aop 的配置只有一行，即**<aop:aspectj-autoproxy/>**。这时运行客户端示例代码，我们会发现，其输出的内容跟前面示例的内容完全一样（当然，这个例子要求你必须是在 Jdk1.5 及以上才能运行，因为 java 注解是 Jdk1.5 才引入的功能!)。比较仔细一点的读者会发现，AspectAnnotationBean 中的那些注解标签跟前面配置文件中的大致差不多。

5.3.4 基于 API 方式来使用 Spring AOP

当然，在上面的两个示例中，都需要使用到 AspectJ 的 aspectjweaver.jar 来帮助解析切入点相关表达式。假如我们不想使用 AspectJ，仅仅依靠 Spring AOP 方面的 API，也可以实现类似的功能。(假如你还是使用 Spring2.0 以前的版本，比如 Spring1.2，那么必须使用 Spring AOP 的 API 来处理横切交叉关注点的问题)。

写一个用来具体处理连接点的通知 Bean，这个 Bean 实现了 MethodBeforeAdvice 及 AfterReturningAdvice 接口。AdviceBean 的代码如下：

```
package springroad.demo.chap5.exampleB;
```

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

public class AdviceBean implements MethodBeforeAdvice, AfterReturningAdvice {
    //MethodBeforeAdvice 接口要求实现的方法，将在方法内的代码执行之前运行
    public void before(Method method, Object[] args, Object target) throws Throwable {
        validateUser();
        beginTransaction();
    }
    //AfterReturningAdvice 接口要求实现的方法，将在访问执行完后运行
    public void afterReturning(Object returnValue, Method method, Object[] args, Object
target) throws Throwable {
        endTransaction();
        writeLogInfo();
    }
    public void validateUser()
    {
        System.out.println("执行用户验证!");
    }
    public void writeLogInfo()
    {
        System.out.println("书写日志信息");
    }
    public void beginTransaction()
    {
        System.out.println("开始事务");
    }
    public void endTransaction()
    {
        System.out.println("结束事务");
    }
}
```

同前面解释的一样，作为演示，我们把切面模块处理的代码直放在了 AdviceBean 中，实际应用中将会调用具体的处理模块来实现。

有了通知处理的类 AdviceBean，还要进一步定义切入点以及切面模块，这里我们直接使用 Spring 自带的切入点处理实现 NameMatchMethodPointcut 及默认切面封装（Spring 中称为增强器）DefaultPointcutAdvisor 类来实现。因此，不需要在书写自己的切入点及切面类，直接使用上面的两个类在配置文件中进行配置即可。

接下就是在 Spring 配置文件中进行配置，分别配置一个代表直接业务组件的 targetBean，一个用来代表通知(Advice)具体实现的 adviceBean，一个代表切入点描述的 pointcutBean，一个代表切面模块的 aspectBean，最后是使用代理 Bean 来定义的业务组件。配置文件 aspect-spring.xml 的内容如下：

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
```

```
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="targetBean"
    class="springroad.demo.chap5.exampleB.ComponentImpl">
    </bean>
    <bean id="adviceBean"
    class="springroad.demo.chap5.exampleB.AdviceBean">
    </bean>
    <bean id="pointcutBean"
    class="org.springframework.aop.support.NameMatchMethodPointcut">
    <property name="mappedName" value="business*"></property>
    </bean>
    <bean id="aspectBean"
    class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="advice" ref="adviceBean"></property>
    <property name="pointcut" ref="pointcutBean"></property>
    </bean>
    <bean id="component"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="targetBean"></property>
    <property name="interceptorNames">
    <list>
    <value>aspectBean</value>
    </list>
    </property>
    </bean>
    </beans>
```

运行客户端程序，你会发现跟前面示例输出的一样。表示我们确实在调用业务组件的业务方法之前及之后都确保调用了安全检查、事务处理、日志记录等模块。

细心的读取会发现，此时客户端程序中用到的 component 这个 Bean 被定义为一个 ProxyFactoryBean 类型，其实，也正是这个代理工厂 Bean 的作用，才使得客户端用到的业务组件集成了处理横切问题的功能。

5.4 Spring 中的 AOP 实现及应用

Spring 是一个实现了 AOP 的框架，我们可以在应用 Spring 的系统中直接进行 AOP 编程，解决应用程序中横切关注点问题；另外 Spring 框架的其它一些主要模块（如数据访问层、事务处理等）也是建立在 Spring AOP 的框架的基础上，学会使用 Spring AOP，可以为我们进一步学会使用 Spring 的其它实用功能打下基础。

5.4.1 简介

Spring 框架中包含一个 AOP 实现，是 Spring 框架的重要组成部分，实现了 AOP 联盟

约定的接口。Spring 中的 AOP 主要使用基于拦截器的方式，实现了对方法调用连接点相关的拦截。，

Spring 框架使用纯 Java 的方式来实现 AOP，也就是不需要像 AspectJ 那样需要自己的专门的编译器或类加载器来实现织入。Spring AOP 的织入过程是在运行时由 Spring 使用 Java 的代理机制来完成。Spring AOP 依赖于 Spring 的核心 IOC 容器，并与容器融为一体，因此可以在配置文件中声明应用 AOP 功能，提供在 Spring 框架中像 EJB 中的声明式系统服务一样的功能。

Spring 没有像 AspectJ 那样强大的功能，只支持与方法调用有关的连接点。不支持属性连接点拦截，也不支持构造函数连接点拦截。用 Spring 主创人员的话说："这是有意为之，不希望将 Spring 用于增强所有对象"。当然，在 J2EE 应用中，AOP 拦截到方法级的操作已经足够。

Spring2.0 中参考了 AspectJ 中的很多设计及用法，提供易于理解的 AOP 配置方式来声明模块的切面、切入点及通知等，同时还提供了 Java5 注解标签来标识 AOP 相关信息。因此，我们可以在 Spring 非常简单的使用 AOP 功能，来解决软件项目中具有横切性质的问题。当然 Spring2.0 完全保留了对以前版本的 AOP 框架兼容，因此，如果你愿意，仍然按以前的方式来使用 Spring 的 AOP 功能。

通过使用 Spring 的 AOP 功能，我们可以在企业级 J2EE 应用程序中使用以下功能：

使用声明式系统服务，比如事务服务、安全性服务。

根据实际业务需求的自定义切面，使用 AOP 的方式解决核心关注点外围的问题。

5.4.2 Spring AOP 中对 AspectJ 的支持

Spring 是一个非常灵活的应用框架，因此非常容易与其它框架进行集成，部分或者完全引入其它开源应用框架的功能。Spring2 中的 AOP 部分引入了 AspectJ 的切入点语法，使得可以使用基于 AspectJ 的切入点表达式来描述切入点。

Spring2.0 当前支持的 AspectJ 原始切入点表达关键字有，execution(方法执行的连接点，这是 Spring 中最主要的切入点指定者)，within(限定匹配特定类型的连接点)，this(连接点本身)，target(连接点目标对象)，arg(连接点参数)等。另外还有@target、@args、@within 、@annotation 等。当然，Spring 可能还会在后面的版本中提供更多 AspectJ 原始切入点的支持。

可以使用 AspectJ 的切入点表达式来描述切入点，通过的格式如下：

execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)

有"?"号的部分表示可省略的，modifers-pattern 表示修饰符如 public、protected 等，ret-type-pattern 表示方法返回类型，declaring-type-pattern 代表特定的类，name-pattern 代表方法名称，param-pattern 表示参数，throws-pattern 表示抛出的异常。在切入点表达式中，可以使用*来代表任意字符，用..来表示任意个参数。

比如前面的示例中，execution(* springroad.demo.chap5.exampleB.Component.business*(..)) 就是一个基于 AspectJ 的切入点表达式。匹配 springroad.demo.chap5.exampleB.Component 类中以 business 开头，返回值为任意类型的方法。

各个原始切入点表达式的详细使用方法请参考前面的 AspectJ 一节及其它 AspectJ 相关资料。

另外，Spring 还支持通过使用 AspectJ 的注解方式来在源码中标识切入点、增强、切面等，就像在 AspectJ 中使用注解来开发切面一样。Spring2 主要支持以下的 AspectJ 标签：

@Aspect－标识一个切面；
@Pointcut－标识一个切入点；
@Before－标识一个前置增强；
@AfterReturning－标识返回后增强；
@AfterThrowing－标识异常抛出增强；
@After－标识后增强；
@Around－标识环绕增强；
@DeclareParents－标识引介；

如前面的例子中，在普通的 POJO 中，使用下面的代码标识了一个后置增强，validateUser 方法增强的实现代码。

```
@Before("somePointcut()")
public void validateUser()
{
    System.out.println("执行用户验证!");
    }
```

当然，由于 Spring 的 AOP 功能比较简单，只实现了针对方法的拦截。假如需要在应用程序中引入更多 AOP 功能，可以直接在 Spring 项目中使用 AspectJ。由于 AspectJ5 提供了类加载器织入机制，因此我们可以通过配置类加载器，在基于 Spring 的应用中使用全部的 AspectJ 功能。

关于如何配置及使用 AspectJ 的类加载器织入，请参考本章关于 AspectJ 一节的介绍及其它 AspectJ 相关文档。要注意的一点是，尽量在类加载织入的配置参数文件 aop.xml 中定义自己需要织入的包及类，不要把 Spring 框架的相关包包含其中。

5.4.3 Spring AOP 配置方法

在 Spring2 提供了比较灵活的 AOP 配置方式。可以使用基于 Schema 的方式使用<aop:config>在配置文件配置切入点、增强及切面；也可以直接启用@AspectJ 支持功能，直接在 POJO 中使用 Java 注解来标识切面相关配置信息；当然，还可以使用 Spring2 以前的配置方式，即通过在配置文件中分别配置增强(Advice)、切入点(Pointcut)、切面封装/增强器(Advisor)、然后使用代理工厂 Bean 等配置业务对象的代理，按部就班的进行 Spring AOP 配置；另外还提供了一些便捷的配置方式，如自动代理。

在 Spring2 中，我们重点强调使用与 AspectJ 切入点语言相结合的方式来使用 Spring 的 AOP 功能。因此，配置文件将从@AspectJ 支持自动代理、基于 Shcema 的 AOP 配置标签及基于普通 Bean 方式三个方面来介绍 Spring AOP 中的配置。

(1)、使用@AspectJ 标签

在 AspectJ5 中，增加对了 Java5 注解的完全支持，可以使用 Java 注解来取代专门的 AOP 语法，把普通的 Java 类(POJO)声明为切面模块。Spring2 提供了对 AspectJ 的支持，引用了 AspectJ 中的一个库来做切入点表达式语言的解析。这里，我们可以像在 AspectJ5 中进行切面编程一样，直接在 Java 类中用相关的注解来标识切面模块中的各部分。最后直接在 Spring 的配置文件中加上一个开启@AspectJ 注解支持的标签即可。

要开启使用@AspectJ 标签功能，需要在 Spring 配置文件中使用<aop:aspectj-autoproxy/>来开启在 POJO 中通过注解来标识切面模块的识别功能。这时要使用一个切面，只需把带有

@AspectJ 标签的 Java 类配置成一个普通 Bean 即可。如下面的例子：

```
<bean id="someAspect" bean="springroad.demo.LogAspect"/>
```

对应的 java 类 LogAspect.java 源码为：

```java
package springroad.demo;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class LogAspect {
}
```

进一步完善 LogAspect 类中功能。首先加入切入点的定义，使用@Pointcut 标签来定义，如下面在 LogAspect 中加入了一个匹配 springroad.demo 包及下面所有包中，名称为 business 开头的方法的切入点：

```java
@Pointcut("execution(* business*(..))&&within(springroad.demo..*)")
private void businessMethod(){}
```

当然，这个切入点可以直接在配置文件中引用。我们可以在切面中进一步定义增强(Advice)实现，同样使用标签。AspectJ 支持很多增强类型，但当前 Spring 只支持其中部分标签，包括@Before、@AfterReturning、@AfterThrowing、@After、@Around 等几种。比如，在切面 POJO 中要定义一个@After 增强及实现，可以使用下面的方式：

```java
@After("businessMethod()")
public void writeLog()
{
    System.out.println("书写日志信息!");
}
```

@After 标签中的参数也就是前面我们定义的切入点；然后下面的方法即是增强的实现代码！

最后就是在 Spring 配置文件中使用上面的切面，如下面是使用 LogAspect 的一个完整的 Spring 配置文件内容：

```xml
<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-2.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
    <aop:aspectj-autoproxy />
    <bean id="aspectBean2"
class="springroad.demo.LogAspect"></bean>
    <bean id="component"
        class="springroad.demo.chap5.exampleB.ComponentImpl">
    </bean>
</beans>。
```

通过这个配置，就会在指定的连接点执行完成后，插入书写日志的代码。注意在

<beans>中要加入“xmlns:aop”命名声明，并在“xsi:schemaLocation”中指定 aop 配置的 schema 地址。

当然，我们也可以在一个切面类中只包含切点定义信息，在另外一个切面类中包含通知实现信息。比如，下面的代码在 MyAppPointcuts 类中，我们一共定义了三个切入点。分别为：onSaveSomething()、onDelSomething()、onUpdateSomething()。

```
package springroad.demo;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class MyAppPointcuts {
@Pointcut("execution(public * save*(..))")
public void onSaveSomething(){}

@Pointcut("execution(public * del*(..))")
public void onDelSomething(){}

@Pointcut("execution(public * update*(..))")
public void onUpdateSomething(){}
}
```

接下来，我们可以使用另外一个或多个类来存放处理增强(Advice)的实现逻辑。看下面用来进行对象删除处理的切面封装代码：

```
package springroad.demo;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;
@Aspect
public class DelSomethingAdvice {
@Around("MyAppPointcuts.onDelSomething()")
public Object doDelSomething(ProceedingJoinPoint joinPoint) throws
Throwable {
    System.out.println("准备执行删除:"+joinPoint.getArgs()[0]);
    Object retVal = joinPoint.proceed();
    System.out.println("完成了删除操作! ");
    return retVal;
  }
}
```

注意其中@Around标签中参数值是"MyAppPointcuts.onDelSomething()",这表是我们引用的是MyAppPointcuts中的onDelSomething()这个切入点。

(2)、基于 schema(模式）配置 Spring AOP

假如我们的 JDK 底于 1.5，或者我们不想在 Java 类中使用注解来标识切入点、通知实现等相关信息。这时我们可以选择使用完全基于 Schema 的方式来配置 Spring AOP。同样是用普通的 Java Bean（POJO）来封装切面模块，只是需要在 Spring 配置文件中通过 AspecJ 切入点语言表达式来定义切入点，并配置相关的增强(Advice)实现方法等。

在配置文件中，把所有关于 AOP 配置的信息统一放在<aop:config>标签内。通过以 aop 开头的 xml 命令空间来行进行 AOP 配置。一个典型的的包含 AOP 配置信息的 Spring 配置文件结构如下所示：

```
<aop:config>
<aop:pointcut id="somePointcut" .../>
<aop:advisor id="someAdvisor" .../>
<aop:aspect id="someAspect" ref="someBean">
<aop:adviceType id="someAdvice" .../>
</aop:aspect>
</aop:config>
```

<aop:config>中，可以依次定义切入点(Pointcut)、增强器(Advisor)、及切面(Aspect)。在 Advisor 及 Aspect 中又进一步指定增强(Advice)的实现。

来看下面的配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/bean
s http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
    <aop:config>
    <aop:pointcut id="myPointcut" expression="execution(public *
del*(..))" />
    <aop:aspect id="myAspect" ref="aspectBean">
    <aop:around pointcut-ref="myPointcut" method="doDelSomething" />
    </aop:aspect>
    </aop:config>
    <bean id="aspectBean"
class="springroad.demo.DelSomethingAdvice"></bean>
    <bean id="userService"
        class="springroad.demo.UserServiceImpl">
    </bean>
</beans>
```

假如使用增强器(advisor)来封装切面模块，这时增强(advice)必须实现 AOP 联盟 Advice 接口。下面定义一个环绕通知 MethodInterceptor 的实现 RealDelAdvice，代码如下：

```
package springroad.demo;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
public class RealDelAdvice implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws
Throwable {
        System.out.println("准备执行删
```

```
除:"+invocation.getArguments()[0]);
        Object retVal = invocation.proceed();
        System.out.println("完成了删除操作! ");
        return retVal;
    }
}
```

然后修改 Spring 配置文件内容，AOP 配置部分如下：

```
    <aop:config>
    <aop:pointcut id="myPointcut" expression="execution(public * del*(..))" />
    <aop:advisor pointcut-ref="myPointcut" advice-ref="delAdvice" />

    </aop:config>
    <bean id="delAdvice"
class="springroad.demo.RealDelAdvice"></bean>
    <bean id="userService"
        class="springroad.demo.UserServiceImpl">
    </bean>
```

关于基于 Schema 配置的详细信息，请参考 Spring AOP 的 Schema 文件中详细定义，地址：http://www.springframework.org/schema/beans/spring-beans-2.0.xsd

(3)、基于 Spring API 的配置文件

如果不想使用 AspectJ 的切入点描述语言，也不想使用 Spring2 提供的基于 Schema 的 AOP 配置方式，对于 Spring2 以前的版本来说，还可以直接在 Spring 配置文件像配置普通 Bean 一样来进行 Spring AOP 配置。在默认情况下，配置文件中的内容大致包含如下内容：

1、0 个或多个切入点定义 Bean，必须实现 Pointcut 接口；

2、1 个或多个通知实现 Bean，必须实现 Advice 接口；

3、0 个或多个引介 Bean，实现 IntroductionInfo 接口；

4、一个或多个切面封装 Bean，必须实现 Advisor 接口；

5、一个或多个真实业务 Bean；

6、一个或多个代理 Bean。

下面来看一个例子：

```
    <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--定义切入点 -->
    <bean id="pointcutBean"

    class="org.springframework.aop.support.NameMatchMethodPointcut
">
        <property name="mappedName" value="del*"></property>
    </bean>
```

```
    <!-- 定义通知实现-->
    <bean id="adviceBean"
class="springroad.demo.RealDelAdvice"></bean>
    <!--定义一个切面封装-->
    <bean id="aspectBean"

    class="org.springframework.aop.support.DefaultPointcutAdvisor"
>
        <property name="advice" ref="adviceBean"></property>
        <property name="pointcut" ref="pointcutBean"></property>
    </bean>
    <!--定义一个实际业务Bean-->
    <bean id="targetBean" class="springroad.demo.UserServiceImpl">
    </bean>
    <!--定义一个代理业务Bean-->
    <bean id="userService"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="targetBean"></property>
        <property name="interceptorNames">
            <list>
                <value>aspectBean</value>
            </list>
        </property>
    </bean>
</beans>
```

当然，上面的配置并非每一步都不可少，适当选择 Spring 的一些切面封装实现，可以减少切入点配置环节，另外，也可以使用自动代理或代理模板等，从而达到简化 Spring AOP 配置的目的。比如，通过使用 NameMatchMethodPointcutAdvisor 代替 DefaultPointcutAdvisor，然后在代理 Bean 中使用内部 Bean，上面的配置文件可以简化成如下的形式：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 定义通知实现-->
    <bean id="adviceBean"
class="springroad.demo.RealDelAdvice"></bean>
    <!--定义一个切面封装-->
    <bean id="aspectBean"

    class="org.springframework.aop.support.NameMatchMethodPointcut
Advisor">
        <property name="advice" ref="adviceBean"></property>
        <property name="mappedName" value="del*"></property>
    </bean>
```

```
    <!--定义一个代理业务Bean-->
    <bean id="userService"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target"><bean
class="springroad.demo.UserServiceImpl">
    </bean></property>
        <property name="interceptorNames">
            <list>
                <value>aspectBean</value>
            </list>
        </property>
    </bean>
</beans>
```

如果使用自动代理，还可以进一步把上面的配置简化成如下的形式：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 定义通知实现-->
    <bean id="adviceBean"
class="springroad.demo.RealDelAdvice"></bean>
    <!-- 定义一个自动代理 -->
    <bean

    class="org.springframework.aop.framework.autoproxy.BeanNameAut
oProxyCreator">
        <property name="beanNames">
            <value>*Service</value>
        </property>
        <property name="interceptorNames">
            <list>
                <value>adviceBean</value>
            </list>
        </property>
    </bean>
    <!--定义一个实际业务Bean-->
    <bean id="userService" class="springroad.demo.UserServiceImpl">
    </bean>
</beans>
```

注意在自动代理中我们是直接定义业务 Bean，而代理则是由 Spring 在创建 Bean 的时候自动加上去的，不需要专门定义，这种方式对于我们的 Bean 名称一致，需要到处使用插入横切模块的时候使用，比如业务层的事务处理及日志记录等。

尽管我们通过 Spring 提供的一些 PointcutAdvisor 实现，或者是使用自动代理，可以使得配置文件得到一定的减化，相比于 Spring2 中提供的配置方式，基于 Spring AOP API 的配置还是复杂得多。因此，在 Spring2 中，我们推荐尽量选择使用前面的两种配置方式，因为一方面是 AspectJ 的切入点语言功能比较强大，是比较成熟的 Java 语言扩展，另一方面所写的切面封装模块可以完成不依赖于 Spring 或 AOP 联盟的 API，而是针对 POJO，因此会使得我们的切面模块更加独立、更加灵活，编写及调试都将更加方便。

5.4.4 切入点(Pointcut)

在任何一个 AOP 框架中，一个必要的部分就是如何定义及描述连接点与切入点。在 Spring 的 AOP 实现中，由于只实现了针对方法调用的拦截及增强，在此我们只关心方法连接点。Spring 中的跟其它 AOP 框架一样，对于方法连接点主要是观察方法执行前、方法执行正常返回后、方法执行抛出异常时以及方法执行过程中这几个方面。针对这几个方面，连接点的描述也比较简单，在 Spring AOP 框架内部已经预定义了这些连接点的描述，这一描述是通过与增强(Advice)相关的接口来固定建立关系的，比如 MethodBeforeAdvice 这个增强就与方法执行前这个连接点关联。因此，我们现在要关心的问题是切入点的描述，由于 Spring 只关心方法调用连接点，因此切入点的描述重点也就是方法的描述上。方法的描述涉及到几个方面，一个是方法名称的描述（比如要考虑通配符)、方法所带参数、方法所处的类、方法执行的流程等。基于以上的分析，在 Spring AOP 中，引入了一个 Pointcut 接口来表示切入点，Poincut 接口的内容如下：

```
public interface Pointcut {
    ClassFilter getClassFilter();
    MethodMatcher getMethodMatcher();
    Pointcut TRUE = TruePointcut.INSTANCE;
    }
```

其中 ClassFilter 用来处理切入点中的类集合，而 `MethodMatcher` 是用来处理匹配的方法。`ClassFilter` 完整的定义如下：

```
public interface ClassFilter {
    boolean matches(Class clazz);
}
```

其中 matches 方法用来判定某一个类是否在切入点的类集合中。

MethodMatcher 完整的定义如下：

```
public interface MethodMatcher {
    boolean matches(Method m, Class targetClass);
    boolean isRuntime();
    boolean matches(Method m, Class targetClass, Object[] args);
}
```

matches(Method m, Class targetClass)方法用来判定某一个类中的某一个方法是否匹配。isRuntime 来来指定这个切点是否为运行时动态切点。若 matches 为 true 且是运行时动态切点，则进一步使用带方法参数的 matches 来判定方法匹配。

在 Spring AOP 的核心处理引擎中，就是通过这几个接口来包装描述一个切入点的。当然，这里只是接口，实现应用中我们需要使用特定的实现。好在 Spring 已经为我们定实现了大多数实现应用中的切入点类型。在我们的应用程序中，直接使用这些 Spring 预定义的

切入点类，设置一定的参数，即可得到一个我们所希望的切入点。

■ **NameMatchMethodPointcut**

全路径：org.springframework.aop.support.NameMatchMethodPointcut

这个是一个通过直接定义方法名称来判断匹配的切入点(Pointcut)实现，这个类使用很简单，可以通过下面几个方法添加匹配的方法。

void setMappedName(String mappedName) —直接指定一个匹配的方法名称。

void setMappedNames(String[] mappedNames) —指定一组(多个)匹配的方法名称。

参数中可以使用“*”来代表 0 个或多个字符。因此，要在 Spring 配置文件中配置一个匹配 set、get 及 save 开头的方法的切入点，可以像下面这样设置：

```
<bean id="pointcutBean"

class="org.springframework.aop.support.NameMatchMethodPointcut
">
    <property name="mappedNames">
        <list>
            <value>set*</value>
            <value>get*</value>
            <value>save*</value>
        </list>
    </property>
    </bean>
```

也可以直接通过 mappedName 来设置，如下所示：

```
<bean id="pointcut1"
    class="org.springframework.aop.support.NameMatchMethodPointcut
">
    <property name="mappedName" value="business*"></property>
    </bean>
```

还可以在程序中，直接使用 addMethodName(String name)方法来往对象中手动添加匹配值。如：

```
NameMatchMethodPointcut pc=new NameMatchMethodPointcut();
pc.addMethodName("set*").addMethodName("get*");
```

NameMatchMethodPointcut 只能解决像什么或是什么该当方法描述与匹配，比如所有像 set*的方法。如果没有设置任何值匹配值，默认将匹配所有方法！

■ 正则表达式切入点

在前面介绍的 NameMatchMethodPointcut 中，只能使用通配符“*”来代表 0 个或多个字符。但在实现应用中，我们的切入点方法名称可能不只这么简单，比如我们需要匹配所有不包含 s 字符的方法，或者是匹配第一个字符为任意字符、后面一个是 e 或 a、后面是 0 或多个字符的方法，这时 NameMatchMethodPointcut 就难以满足这种要求了。不用急，Spring 的设计已经考虑到这种情况，因此设计了通过正则表达式来匹配方法名称的切入点实现。跟使用其它的切入点一样，我们直接把相应的切入点处理类拿来使用即可。

正则表达式切入点有两个，一个是 JdkRegexpMethodPointcut，这是在直接使用 JDK 1.4 或更高版本里提供的正则表达式支持功能来进行切入点匹配的；若是 JDK 比较低的版本，

可以使用 Jakarta ORO 项目(http://jakarta.apache.org/oro/)提供的 Perl5 兼容的正则表达式处理功能来进行切入点匹配处理，即使用 Perl5RegexpMethodPointcut 来处理。

正则表达式的功能很多，比如我们可以使用“.”代表任意字符，使用“?”来表示出现 0 或 1 次，使用“*”来表示出现 0 或 n 次，可以用“[]”来表示一个集合等等，关于正则表达式的使用方法请查阅有关正则表达的资料。总之，使用正则表达式基本上可以满足 90%以上的方法名称匹配需求。

切入点：JdkRegexpMethodPointcut

全路径：org.springframework.aop.support.JdkRegexpMethodPointcut

需要在 JDK1.4 及以上的环境运行，不需要额外的库。

切入点：Perl5RegexpMethodPointcut

全路径：org.springframework.aop.support.JdkRegexpMethodPointcut

需要把 jakarta-oro-xx.jar 文件放到 classpath 上，比如 jakarta-oro-2.0.8.jar。

正则表达式切入点关键方法：

void setPattern(String pattern) －设置匹配的正则表达式字符串；

void setPatterns(String[] patterns) －设置一组匹配的正则表达式字符串；

void setExcludedPattern(String excludedPattern) －设置需要排除指定的字符串；

void setExcludedPatterns(String[] excludedPatterns) －设置需要排除的一组字符串。

下面是使用正则表达式切入点的使用示例：

```
<bean id="pointcutBean"

class="org.springframework.aop.support.Perl5RegexpMethodPointc
ut">
      <property name="patterns">
          <list>
              <value>.*business.*</value>
              <value>.*save.*</value>
          </list>
      </property>
      </bean>
```

上面的设置将匹配所有包含 business 及 save 字符串的方法。假如我们要排除名为 business2 的方法，则需要设置 execludePattern 属性值，配置文件如下所示：

```
<bean id="pointcutBean"

class="org.springframework.aop.support.Perl5RegexpMethodPointc
ut">
      <property name="patterns">
          <list>
              <value>.*business.*</value>
              <value>save*</value>
          </list>
      </property>
      <property name="excludedPattern"
value=".*business2"></property>
```

```
    </bean>
```

上面的配置以直接换成使用 JdkRegexpMethodPointcut 来定义，只需要把 class 中的 Perl5RegexpMethodPointcut 改成 JdkRegexpMethodPointcut 即可，其它的都不用变，如下所示：

```
<bean id="pointcutBean"

    class="org.springframework.aop.support.JdkRegexpMethodPointcut
">
        <property name="patterns">
            <list>
                <value>.*business.*</value>
                <value>save*</value>
            </list>
        </property>
        <property name="excludedPattern"
value=".*business2"></property>
        </bean>
```

■ **描述式切入点 ExpressionPointcut**

在 Spring2 中，在 Pointcut 的基础上，引入了一个 ExpressionPointcut 接口用来通过类似 AspectJ 中的切入点表达语言来描述切入点（其实当前发布的版本也只提供了一个调用 AspectJ 的表达式描述语言解析工具来实现表达式描述切入点）。有了 ExpressionPointcut，我们可以使用下面更加简单的方式来描述切入点，如 execution(* Component.business*(..))表示执行所有 Component 的业务方法（此处为 business 打头的方法）。

ExpressionPointcut 定义了一个返回描述切入点表达式字符串的方法，getExpression()，完全代码如下

```
public interface ExpressionPointcut extends Pointcut {
    String getExpression();
}
```

Spring2 提供了一个 `ExpressionPointcut` 的实现，即 AspectJExpressionPointcut，提供与 AspectJ 中切入点描述语法一样表述方式来描述切入点。当然，由于 Spring AOP 核心引擎仍然是代理实现对方法的拦截，因此，这里只有与方法调用连接点描述相关的切入点表达式才能处理。要使用 AspectJExpressionPointcut，需要用到 AspectJ 中的语法解析处理器，因此需要引入 AspectJ 的 aspectjweaver.jar 包。当然 Spring AOP 在这里只用了这个包的一小部分功能，而且最终仍然使用 Spring 纯 java 的方式通过代理在运行时完成织入的，并不依赖于 AspectJ 的编译器及运行时。

关于 AspectJExpressionPointcut 的使用很简单，只需要直接使用 setExpression 方法设置表达式字符串即可！下面的切入点配置表示匹配 Component 下以 business 开头的所有方法，不管参数个数！

```
<bean id="pointcutBean"

    class="org.springframework.aop.aspectj.AspectJExpressionPointc
ut">
```

```
        <property name="expression"
            value="execution(void
springroad.demo.chap5.exampleB.Component.business*(..))">
        </property>
        </bean>
```

■ 切入点集合运算

在 Spring AOP 中，还支持切入点集合运算，通过集合运算来组合出更多符合实际需要的切入点。下面我来看一个简单的示例：

```
NameMatchMethodPointcut p1=new NameMatchMethodPointcut();
    p1.setMappedName("set*");
    JdkRegexpMethodPointcut p2=new JdkRegexpMethodPointcut();
    p2.setPattern(".*business.*");
    Pointcut p=Pointcuts.union(p1,p2);
```

在示例中，我们使用 Spring 提供的 Pointcuts 类的静态方法 union，来把 p1 及 p2 两个切入点组合到一起，得到切入点可以用来匹配以 set 开头或包含 business 字符串的方法。

5.4.5 增强(Advice)

前面我们介绍了切入点，我们知道可以通过多种方式来匹配 Spring AOP 所要切入的方法。但是具体是在方法的什么位置切入，是方法调用前还是方法调用完成后？切入后做些什么？是改变程序的执行流程还是改变返回值？关于这些问题，就由增强或通知 (Advice)负责处理。

在 Spring AOP，增强的类型比较少，都是围绕方法调用连接点展开的，一共有方法调用前增强(MethodBeforeAdvice)、方法调用后增强(AfterReturningAdvice)、环绕增强(MethodInterceptor)、方法调用异常增强(ThrowsAdvice)等几种类型。下面将分别介绍：

■ **MethodBeforeAdvice**

MethodBeforeAdvice 的代码如下：

```
public interface MethodBeforeAdvice extends BeforeAdvice {
    void before(Method m, Object[] args, Object target) throws Throwable;
}
```

其中 BeforeAdvice 是 AOP 联盟提供的一个标识接口，代码如下：

```
import org.aopalliance.aop.Advice;
public interface BeforeAdvice extends Advice {
}
```

就像其名称所指的一样，这个增强的位置是在方法之前，因此也称为前置增强。也就是在执行到连接点的时候，首先执行增强中的自定义模块。由于这个增强没有返回值，因此，执行完后下一步将继续执行连接点的方法调用。来看下面的代码：

```
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;
public class MethodInvokCount implements MethodBeforeAdvice {
    private int num=0;
    public void before(Method method, Object[] args, Object target)
            throws Throwable {
```

```
        num++;
    }
    public int getNum() {
        return num;
    }
}
```

MethodInvokCount 类用来记录连接点方法的调用次数，每当要执行连接点方法时，都会首先调用一次 MethodInvokCount 中的 before 方法。

在 before 方法中，method 表示具体连接点，args 表示方法所代的参数，target 返回该方法所属的目标对象。

如果再执行 MethodBeforeAdvice 的过程中抛出异常，这将中止拦截器链的进一步执行，异常将沿着拦截器链向上传播。如果异常是非强制检查的（unchecked）或者已经被包含在被调用方法的签名中（译者：即出现在方法声明的 throws 子句中），它将被直接返回给客户端； 否则它将以一个运行时非强制检查(不需要客户端使用 try-catch)异常中返回。

- **AfterReturningAdvice，返回后增强**

方法调用正常返回后增强，也称为后置增强。这是在连接点的方法执行并通过一个语句返回后，执行增强中的自定义模块。**AfterReturningAdvice** 的定义如下：

```
import java.lang.reflect.Method;
import org.aopalliance.aop.Advice;
public interface AfterReturningAdvice extends Advice {
void afterReturning(Object returnValue, Method method, Object[] args,
Object target) throws Throwable;
}
```

在 afterReturning 方法的几个参数中，returnValue 表示连接点的返回值，method 为当前方法，args 代表方法的参数值，target 表示方法所属的实际目标对象。来看下面的代码：

```
import java.lang.reflect.Method;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.aop.AfterReturningAdvice;
public class WriteLogAdvice implements AfterReturningAdvice {
    private final Log log=LogFactory.getLog(WriteLogAdvice.class);
    public void afterReturning(Object returnValue, Method method,
            Object[] args, Object target) throws Throwable {
        log.info("成功执行了类"+target.getClass()+"的方法
"+method.getName()+",返回值为"+returnValue);
    }
}
```

在后置增强的实现 afterReturning 方法中，调用 log 的 info 方法把方法调用的相关情况写入了日志中。在实际应用中，有一些特别重要的业务组件，需要详细记录其每一个方法的执行情况，以便于后期审计，这时可以使用类似的后置增强来实现。

在后置增强的 afterReturning 方法运行过程中，若中间出现异常，异常将沿着拦截器链返回（或抛出），而不会给客户端继续返回其结果。

■ **异常增强 ThrowsAdvice(After throwing advice)**

Spring 中异常增强 TrowsAdvice 的定义如下：

```
package org.springframework.aop;
import org.aopalliance.aop.Advice;
public interface ThrowsAdvice extends Advice {
}
```

由定义我们可以看到，ThrowAdvice 接口中没有定义任何方法，因此他只是一个标识接口。但是在实际应用中，具体的异常增强实现方法将实现下面的模板方法：

```
afterThrowing([Method], [args], [target], Throwable subclass)
```

在上面的模板方法 afterThrowing 的几个参数中，中括号的参数是可选的。[Method]用来代表一个方法，[args]用来表示方法调用参数，[target]表示方法所属的实际目标对象，最后一个必选的是异常类型 Throwable。

因为一个方法在执行的过程中，可能会抛出各式各样的异常，我们可以在异常增强中分别针对各个异常进行相应的处理。下面例子中设计了一个 ErrorLogAdvice 异常处理增强，可以用来作为我们系统中对所有异常进行日志记录的增强，可以在这个增强中使用统一的方式集中记录或处理系统中出现的各式各样的异常。

```
import java.lang.reflect.Method;
import org.springframework.aop.ThrowsAdvice;
import springroad.demo.chap5.exampleB.InValidateUserException;
public class ErrorLogAdvice implements ThrowsAdvice {
    //当出现Servlet异常时
    public void afterThrowing(javax.servlet.ServletException ex)
            throws java.lang.Throwable {
        log("出现了Servlet异常!");
    }
    //当用户不合法的时候
    public void afterThrowing(Method method, Object[] args,
            Object target, InValidateUserException ex)
            throws java.lang.Throwable {
        log("用户权限不够!" + method.getName());
    }
    //…
    private void log(String s) {
        // 把日志信息s写入日志中
    }
}
```

■ **环绕增强 MethodInterceptor(Around advice)**

环绕增强也称为拦截器，是功能比较强大也非常灵活的增强类型，因为他即可改变连接点的程序流程，又可以改变连接点方法的返回值。Spring 环绕增强中的相关定义全部使用的 AOP 联盟规范的接口，我们先来看看 MethodInterceptor 的完全代码：

```
public interface MethodInterceptor extends Interceptor {
    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

拦截器Interceptor只是一个标识接口，定义如下：

```java
public interface Interceptor extends Advice {
}
```

在上面的定义中，环绕增强需要实现 nvoke(MethodInvocation invocation)这个方法，这个方法的参数 MethodInvocation 中封装了方法调用的相关对象，包括方法名称、方法参数、方法所属的实际目标对象等。另外 MethodInvocation 中还有一个 proceed()方法，用于继续执行连接点本身的调用。

下面来看一个示例：

```java
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
public class MethodInterceptorBean implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws
Throwable {
        validateUser();
        beginTransaction();
        Object ret=invocation.proceed();
        endTransaction();
        writeLogInfo();
        return ret;
    }
    public void validateUser()
    {
        System.out.println("执行用户验证!");
    }
    public void writeLogInfo()
    {
        System.out.println("书写日志信息");
    }
    public void beginTransaction()
    {
        System.out.println("开始事务");
    }
    public void endTransaction()
    {
        System.out.println("结束事务");
    }
}
```

在上面的代码中，我们使用的环绕通知，取代了前面 AdviceBean 中的 MethodBeforeAdvicee 及 AfterReturningAdvice 增强组合所完成的功能。即在连接点方法执行前，先执行用户验证、开启事务操作，然后再通过 invocation 的 proceed()方法继续执行连接点方法的业务逻辑，执行完成后我们进一步执行结束事务及日志记录操作。

上面的示例代码只是演示了在环绕增强中插入自定义的功能，那么，如何通过环绕增强来改变程序流程呢？是不是可以不执行连接点方法直接返回呢？答案是肯定的，你可以通过本章最后的综合示例程序中看到我们使用环绕增强来改变了 Hero 对象 getArmor 方法返回

值。

当然，在 Spring AOP 中，一个通知增强模块是否就需要实现上面的接口，从底层原理的角度来说，这是肯定的。然而，Spring 2 引入了基于 Schema 的 AOP 配置定义，又引入了 AspectJ 中切面模块相关标签，支持从常规的 java 类中定义增强实现逻辑。也就是说可以通过配置指定某一个普通类(POJO)的某一个方法用来作为增强的实现逻辑。这样我们就可以更加集中精力编写切面单元模块中的相关业务逻辑，并且可以单独作单元测试，最后再使用配置把其与系统中的其它模块组合到一起来。

5.4.6 引介(Introduction)

引介(Introduction)是指在不更改源代码的情况，给一个现有类增加属性、方法，以及让现有类实现其它接口或指定其它父类，从而改变类的静态结构。Spring AOP 通过采代理与拦截器的方式来实现的，可以通过拦截器机制使一个实有类实现指定的接口，由于是使用拦截器的机制，因此 Spring AOP 中引介的底层仍然是增强(Advice)及拦截(Interceptor)。引介不能和任何切入点一起使用，因为它是应用在类级别而不是方法级别。下面是 Spring AOP 中引介拦截器(IntroductionInterceptor)的定义：

```
import org.aopalliance.intercept.MethodInterceptor;
public interface IntroductionInterceptor extends MethodInterceptor,
DynamicIntroductionAdvice {
    }
```

MethodInterceptor 是前面讲的环绕增强(Advice)，即方法拦截器，DynamicIntroductionAdvice 的定义如下：

```
import org.aopalliance.aop.Advice;
public interface DynamicIntroductionAdvice extends Advice {
    boolean implementsInterface(Class intf);
    }
```

implementsInterface 用来指定实现某个接口。

要实现一个引介拦截器，只需要实现 IntroductionInterceptor 接口即可，在 Spring 中，为我们提供了两个 IntroductionInterceptor 的实现，其中比较常用 的是 DelegatingIntroductionInterceptor，另外一个是 DelegatePerTargetObjectDelegatingIntroductionInterceptor。DelegatingIntroductionInterceptor 需要一个准备引入的接口实现的实例作为参数，而 DelegatePerTargetObjectDelegatingIntroductionInterceptor 需要指定一个接口及实现类作为参数。

如下面的代码可以构造一个引介拦截，使用这个引介拦截的代理对象将会自动拥有 UserServiceImpl 类所实现的接口及功能。

```
    DelegatingIntroductionInterceptor                    introduction=new
DelegatingIntroductionInterceptor(new UserServiceImpl());
```

另外 Spring 还定义了一个用于描述引介信息的 IntroductionInfo，内容如下：

```
public interface IntroductionInfo {
    Class[] getInterfaces();
    }
```

还有一个封装 IntroductionInfo 及 Advisor 的引介增强器(切面)，内容如下：

```java
public interface IntroductionAdvisor extends Advisor,
IntroductionInfo {
    ClassFilter getClassFilter();
    void validateInterfaces() throws IllegalArgumentException;
}
```

Spring 为提供了 IntroductionAdvisor 的一个实现 DefaultIntroductionAdvisor，在实际应用中可以通过继承该类并实现指定的接口来实现一个引介增强器(切面)。注意：在配置文件中不能在在没有 IntroductionAdvisor 的情况下使用 IntroductionInterceptor。

因为是引介是改变类的静态结构，因此不需要定义切入点 Pointcut。在 Spring AOP 中，引介的使用是通过把一个指定的接口实现添加到代理工厂中，从而使得代理工厂返回的代理对象也实现引介指定的接口。

下面我们举例子说明，首先定义一个简单的业务接口 TestService 及接口实现 TestServiceImpl，也可以直接是类：

```java
public interface TestService {
    void doSomeThing();
    }
public class TestServiceImpl implements TestService {
    public void doSomeThing() {
        System.out.println("执行某样操作!");
    }
    }
```

另外有一个 UserService 接口及其实现 UserServiceImpl。我们可以在程序中通过代理使用 TestService，如下所示：

```java
ProxyFactory proxy=new ProxyFactory(new TestServiceImpl());
TestService s=(TestService)proxy.getProxy();
s.doSomeThing();
```

下面，我们在不修改 TestServiceImpl 的情况下，要让代理工厂返回的代理对象实现 UserService 接口，这时需要使用引介(Introduction)。

于是创建一个引介拦截，并把拦截添加到代理工厂中，代码如下所示：

```java
DelegatingIntroductionInterceptor introduction=new
DelegatingIntroductionInterceptor(new UserServiceImpl());
proxy.addAdvice(introduction);
```

通过在代理工厂中加入引介拦截，使得代理对象实现 UserService 接口。于是下面的代码变得合法了：

```java
UserService userService=(UserService)proxy.getProxy();
doDelUser(userService);
```

下面的语句，输出结果都将为 true：

```java
System.out.println(proxy.getProxy() instanceof TestService);
System.out.println(proxy.getProxy() instanceof UserService);
```

当然，在 Spring 中，都是通过使用配置文件来配置。于是我们把上面的 ProxyFactory 换成最常用的 ProxyFactoryBean。配置文件 api-aop-introduction.xml 的内容如下：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
```

```
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--定义引介切面-->
    <bean id="introductionAdvisor"

    class="org.springframework.aop.support.DefaultIntroductionAdvi
sor">
        <constructor-arg>
            <bean

    class="org.springframework.aop.support.DelegatingIntroductionI
nterceptor">
                <constructor-arg>
                    <bean class="springroad.demo.UserServiceImpl" />
                </constructor-arg>
            </bean>
        </constructor-arg>
    </bean>
    <bean id="service"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target">
            <bean class="springroad.demo.TestServiceImpl" />
        </property>
        <property name="proxyTargetClass" value="true" />
        <property name="interceptorNames">
            <list>
                <value>introductionAdvisor</value>
            </list>
        </property>
    </bean>
</beans>
```

客户端应用示例代码如下：

```
ApplicationContext context=new
org.springframework.context.support.ClassPathXmlApplicationContex
t("springroad/demo/api-aop-introduction.xml");
    TestService s=(TestService)context.getBean("service");
    s.doSomeThing();
    UserService
userService=(UserService)context.getBean("service");
    doDelUser(userService);
```

当然，也可以继承 DefaultIntroductionAdvisor 来实现一个引介切面，为了定义一个实现 UserService 接口的引介，可以设计一个 UserServiceIntroductionAdvisor，代码如下：

```java
import org.springframework.aop.support.DefaultIntroductionAdvisor;
import
org.springframework.aop.support.DelegatingIntroductionInterceptor
;
public class UserServiceIntroductionAdvisor extends
DefaultIntroductionAdvisor {
    public UserServiceIntroductionAdvisor()
    {
        super(new DelegatingIntroductionInterceptor(new
UserServiceImpl()),UserService.class);
    }
}
```

这时要在配置文件中使用引介，配置文件可以简化成如下的形式：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--定义引介Bean-->
    <bean id="introductionAdvisor"
        class="springroad.demo.UserServiceIntroductionAdvisor" />
    <!-- 配置代理业务Bean -->
    <bean id="service"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target">
            <bean class="springroad.demo.TestServiceImpl" />
        </property>
        <property name="proxyTargetClass" value="true" />
        <property name="interceptorNames">
            <list>
                <value>introductionAdvisor</value>
            </list>
        </property>
    </bean>
</beans>
```

5.4.7 增强器/切面封装(Advisor)

增强器(advisor)可以看作是一个切面的模块化封装，因此也称为切面封装，简称为切面，相当于 AspectJ 中的 Aspect。一个切面封装模块中一般包括一个切入点、一个通知实现。在 Spring AOP 中，切面封装主要有大类，一类是用来封装一个切入点与及通知的实现，这种切面封装主要功能是用来改变程序的流程，这类切面里面包含了切入点；而另外一种切面封装是用来让一个现有的类实现一个接口、甚至指定类继承某一个类、给一个类增加方法等，这种类型的切面封装主要功能是改变了类的静态结构，这类切面封装了引入 Introduction。

Spring AOP 最底层的切面封装是 Advisor 接口，在其下面有两个接口 IntroductionAdvisor

及 PointcutAdvisor。就像他们的名称所定义的一样，IntroductionAdvisor 是用来定义引介的切面封装，而 PointcutAdvisor 是用来定义改变程序流程的切面封装。 Advisor 包含了增强(Advice)的定义，其代码如下：

```
import org.aopalliance.aop.Advice;
public interface Advisor {
    boolean isPerInstance();
    Advice getAdvice();
}
```

PointcutAdvisor 是我们使用得最多的切面封装类型，前面的示例中使用到的 AOP 功能都是这种改变程序流程、在程序的方法调用中加入自定义流程的切面封装。PointcutAdvisor 把切入点与增强封装到了一起，其完全内容如下：

```
public interface PointcutAdvisor extends Advisor {
    Pointcut getPointcut();
}
```

在 Spring 中，给我们提供预定义的 PointcutAdvisor 实现，我们直接在应用程序中使用他即可，下面分别看看这些切入点封装实现。

- **DefaultPointcutAdvisor－默认的切面封装**

全路径：org.springframework.aop.support.DefaultPointcutAdvisor

DefaultPointcutAdvisor 类的使用很简单，他有一个 advice 及 pointcut 属性，我们可以通过构造子或设值注入方式来配置这个 Bean。看下面的构造子注入方式：

```
    <bean id="aspectBean"

    class="org.springframework.aop.support.DefaultPointcutAdvisor"
>
        <constructor-arg ref="adviceBean"/>
        <constructor-arg ref="poingcutBean"/>
    </bean>
```

或者使用设值方法注入，如下所示：

```
<bean id="aspectBean"

    class="org.springframework.aop.support.DefaultPointcutAdvisor"
>
        <property name="advice" ref="adviceBean"></property>
        <property name="pointcut" ref="pointcutBean"></property>
        </bean>
```

- **NameMatchMethodPointcutAdviso 方法匹配切入点切面封装**

全路径：org.springframework.aop.support.NameMatchMethodPointcutAdvisor

在前面的 DefaultPointcutAdvisor 中，需要分别设置 Advice 实现及切入点，才能形成一个完整的切面封装。而 NameMatchMethodPointcutAdvisor 不需要专门定义一个切入点，直接通过设置方法名称匹配字符串，再指定一个增强实现即可。NameMatchMethodPointcutAdvisor 的主要设值方法有：

void setAdvice(org.aopalliance.aop.Advice advice)－设置通知实现；

setClassFilter(ClassFilter classFilter) －设置切入点匹配的类；
void setMappedName(String mappedName) －设置切入点匹配的方法名称，可以使用*通配符；
void setMappedNames(String[] mappedNames) －设置切入点匹配的一组方法名称，可以使用*通配符；

使用 NameMatchMethodPointcutAdvisor，前面示例的配置文件可以写成如下的形式：

```
<bean id="aspectBean"

    class="org.springframework.aop.support.NameMatchMethodPointcut
Advisor">
        <property name="advice" ref="adviceBean"></property>
        <property name="mappedName" value="business*"></property>
    </bean>
```

■ **RegexpMethodPointcutAdvisor－正则表达式切入点切面封装**

全路径：org.springframework.aop.support.RegexpMethodPointcutAdvisor

跟 NameMatchMethodPointcutAdvisor 功能一样，RegexpMethodPointcutAdvisor 也是不需要指定专门的切入点，而是通过直接使用正则表达式字符串匹配来指定切入点。RegexpMethodPointcutAdvisor 提供以下设值方法：

void setAdvice(org.aopalliance.aop.Advice advice)－设置增强实现；
void setPattern(String pattern) －设置一个正则字符串匹配切入点；
void setPatterns(String[] patterns) －设置一组正则字符串匹配切入点；
void setPerl5(boolean perl5) －强制使用 Perl5 来处理正则字符串。

下面的配置文件配置了一个 RegexpMethodPointcutAdvisor：

```
    <bean id="aspectBean"

    class="org.springframework.aop.support.RegexpMethodPointcutAdv
isor">
        <property name="advice" ref="adviceBean"></property>
        <property name="pattern"
value=".*Component.business.*"></property>
    </bean>
```

■ **AspectJExpressionPointcutAdvisor－AspectJ 表达式切入点切面封装**

这个切面封装同样不需要专门设置切入点，而是直接通过使用 AspectJ 的切入点表达式来生成切入点，我们只需要给他设置一个切入点表达式，然后设置一个通知实现，即可工作。AspectJExpressionPointcutAdvisor 的主要设值方法如下：

void setAdvice(org.aopalliance.aop.Advice advice)－设置增强实现；
void setExpression(String expression) －设置 AspectJ 切入点描述表达式
void setLocation(String location) －设置位置参数
void setParameterNames(String[] names) －设置表达式中的参数名称
void setParameterTypes(Class[] types)－设置参数类型，与名称对应

下面的配置文件定义了一个 AspectJExpressionPointcutAdviso：

```
<bean id="aspectBean"
```

```
    class="org.springframework.aop.aspectj.AspectJExpressionPointc
utAdvisor">
        <property name="advice" ref="adviceBean"></property>
        <property name="expression"
            value="execution(void
springroad.demo.chap5.exampleB.Component.business*(..))">
        </property>
    </bean>
```

5.4.8 ProxyFactoryBean

Spring 的 AOP 横切面切入过程是在运行时进行的，是一个基于拦截机制的 AOP 框架，因此，其核心为代理的实现。因为只有在代理中才能很好的引入拦截。关于使用代理及拦截机制的 AOP 具体实现原理，我们将在后面 Spring AOP 原理一章详细讲解，这里只作简单介绍。

代理的工作机制很简单，就是当我们需要使用某一个类的时候，由 Spring 通过一定的代理机制，创建一个我们所需类代理对象，代理对象跟实际对象实现相同的接口或者是实际对象类的一个子类。当执行代理对象上的某一个方法时，其交由一个回调对象来处理，而我们可以自定义这个回调对象，从而加入自定义的程序逻辑，即 AOP 中的增强。因此，在系统中就需要有一个负责根据一定的策略，创建代理对象的代理工厂角色，在 Spring 的 AOP 实现中，ProxyFactoryBean 正是扮演了这个角色。

ProxyFactoryBean 是一个工厂 Bean，回顾前面第四章中《Spring IOC》一节关于工厂 Bean(即 FactoryBean)的介绍，FacotryBean 返回的并非是 class 属性或构造方法创建出来的对象，返回的对象是其通过其 getObject()得到。因此，可以使用 ProxyFactoryBean 来充当任何需要代理的对象的代理工厂。如下面的配置文件，就将返回由 target 所指定的(在这里是 UserServiceImpl)一个代理对象：

```
    <bean id="userService"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target"><bean
class="springroad.demo.UserServiceImpl">
    </bean></property>
        <property name="interceptorNames">
            <list>
                <value>aspectBean</value>
            </list>
        </property>
    </bean>
```

在 Spring AOP 中，既可以对目标接口进行代理，也可以对目标类创建代理。当要代理一个目标类的时候，Spring 自动使用 CGLIB(因为 JDK 的代理机制只能对接口代理)来创建代理，而代理目标为一个或多个接口时，默认将使用 JDK 的代理来创建代理，当然也可通过设置使用 CGLIB 来对接口代理。

下面，我们来重点看看 ProxyFactoryBean 的设值方法：

void setTargetName(String targetName)－设置代理目标对象的 bean 名称；

void setTarget(Object target)－设备代理目标对象 Bean；这个用来设置的是目标对象。在使用代理的时候，我们一般都不会再需要直接使用目标对象，因此，我们可以把目标对象配置成一个内部 Bean，这是比较推荐的使用方式。
void setInterceptorNames(String[] interceptorNames)－设置拦截器的名称，在 Spring AOP 中，这里的拦截器可以是 Advice 或 Advisor。若是 Advice，则所有方法都将拦截，若是 Advisor，则将根据其中具体的切入点及通知现实来处理；拦截器的名称可以使用*来代表 0 或多个任意字符；
void setProxyInterfaces(Class[] proxyInterfaces)－设置代理的目标类接口；可以指定代理一个或多个接口；
void setInterfaces(Class[] interfaces)－设置代理接口，跟 proxyInterfaces 属性作用一样；
void setProxyTargetClass(boolean proxyTargetClass) －设置是否代理目标类。若为 true，则在代理的时候直接使用 CGLIB 来创建一个目标类的子类；
void setAutodetectInterfaces(boolean autodetectInterfaces) －设置是否自动检测目标类实现的接口，默认为 true。因此，在没有设置 proxyInterfaces 而且没把 proxyTargetClass 设为 true 时，Spring 将自动检测目标类实现的所有接口，若实现了 1 个或多个接口，将创建这些这些接口的代理，否则将使用 CGLIB 来创建代理。
void setBeanClassLoader(ClassLoader classLoader) －设置类加载器；
void setFrozen(boolean frozen)－设置是否固定增强，即在代理工厂被配置之后，是否还允许修改通知；默认值为 false，即在代理配置被加载之后，不允许再修改代理的配置。
void setAopProxyFactory(AopProxyFactory apf) －设置是 JDK 用动态代理、CGLIB 还是其它代理策略，缺省实现将根据情况自动选择动态代理(有接口)或者 CGLIB。
void setSingleton(boolean singleton) －设置在每次调用代理工厂的 getObject 时，是否应该返回同一个对象。工厂是否应该返回同一个对象，不论方法 getObject()被调用的多频繁。多个 FactoryBean 实现都提供了这个方法。缺省值是 true。如果你希望使用有状态的增强，可以把单例属性的值设置为 false 来使用原型通知。
public void setOptimize(boolean optimize)－是否使用 CGLIB 代理优化策略。仅用于 CGLIB 代理；对于 JDK 动态代理（缺省代理）无效。除非完全了解 AOP 代理如何处理优化，否则不推荐用户使用这个设置。
void setExposeProxy(boolean exposeProxy) －决定当前代理是否被保存在一个 ThreadLocal 中以便被目标对象访问。

在 ProxyFactoryBean 的众多 JavaBean 属性中，target 或(targetName)是必须指定的 interceptorNames 也是必须指定，否则就没必要使用代理了！其它的属性可以根据实际情况选择使用。

使用抽象 Bean 配置来简化代理设置

如果使用 Spring API 的方式来开发 AOP 应用，则要求每一个需要插入横切模块的业务 Bean 都需要使用 ProxyFactoryBean 来进行配置，也就是要配置多个 ProxyFactoryBean，使得配置文件中的内容过于烦琐。此时我们可以借助于 Spring 提供的抽象 Bean 配置，来配置一些模板 Bean。这样其它的业务 Bean 的配置直接继承模板 Bean 配置即可，大致可以分成两个步骤来实现。

第一步、配置一个或多个抽象 Bean，用于充当配置模板，如下所示：

```
<bean id="baseProxyParent"
      class="org.springframework.aop.framework.ProxyFactoryBean"
      abstract="true">
```

```
        </bean>
<bean id="businessProxyParent" parent="baseProxyParent"
        abstract="true">
        <property name="interceptorNames">
            <list>
                <value>aspectBean</value>
            </list>
        </property>
        </bean>
```

第二步、通过继承抽象 Bean 的配置，使用简化的方式配置业务 Bean，如下所示，

```
<bean id="userService" parent="businessProxyParent">
        <property name="target">
            <bean class="springroad.demo.UserServiceImpl"></bean>
        </property>
        </bean>
```

下面是一个使用了抽象（模板)代理配置的完整示例：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- 基本ProxyFactoryBean配置-->
    <bean id="baseProxyParent"
        class="org.springframework.aop.framework.ProxyFactoryBean"
        abstract="true">
    </bean>
    <!-- 业务层代理ProxyFactoryBean配置-->
    <bean id="businessProxyParent" parent="baseProxyParent"
        abstract="true">
        <property name="interceptorNames">
            <list>
                <value>aspectBean</value>
            </list>
        </property>
    </bean>
    <!-- 数据访问层代理ProxyFactoryBean配置-->
    <bean id="daoProxyParent" parent="baseProxyParent"
        abstract="true">
        <property name="interceptorNames">
            <list>
                <value>daoAspectBean</value>
            </list>
        </property>
    </bean>
    <!-- 定义通知实现-->
```

```
    <bean id="adviceBean"
class="springroad.demo.RealDelAdvice"></bean>
    <!--定义一个切面封装-->
    <bean id="aspectBean"

    class="org.springframework.aop.support.NameMatchMethodPointcut
Advisor">
        <property name="advice" ref="adviceBean"></property>
        <property name="mappedName" value="del*"></property>
    </bean>
    <!--定义业务层的Bean，继承了businessProxyParent的配置属性-->
    <bean id="userService" parent="businessProxyParent">
        <property name="target">
            <bean class="springroad.demo.UserServiceImpl"></bean>
        </property>
    </bean>
    <bean id="roleService" parent="businessProxyParent">
        <property name="target">
            <bean class="springroad.demo.RoleServiceImpl"></bean>
        </property>
    </bean>
    <!--定义数据访问层的Bean-->
    <bean id="userDao" parent="daoProxyParent">
        <property name="target">
            <bean class="springroad.demo.UserDaoImpl"></bean>
        </property>
    </bean>
    <!-- ... -->
</beans>
```

当然，还可以使用前面介绍的通过自动代理来简化 Spring AOP 配置。

5.5 示例：模拟 Warcraft 游戏

通过前面的几节，我们应该对 AOP 的概念及 AOP 的一些使用方法有了一定的认识，现在通过一个综合、完整的模拟 warcraft 游戏示例，来进一步演示 Spring 中 AOP 的应用。本示例从系统核心关注点（主模块）的设计开始，到引入横切关注点问题，并通过 Spring2 提供的几种方式来实现横切关注点的需求，每一个部分都作了比较详细的讲述，可以让我们灵活掌握 Spring2 中的 AOP 应用。

5.5.1 示例简介

相信大家都玩过 warcraft3 的吧，这里就借 Warcraft 中的一些角色，做一个模拟的小游

戏。程序中将通过 AOP 编程方法，来解决一些具有横切性质的问题。示例程序主要是实现一个战斗系统，有战场 WarField、战场中的角色(游戏主角，也称为英雄)Hero、及角色所持有的物品或技能(Props)三个部分，道具又进一步分成增加攻击性质的道具系列(AttackProp)以及增加防御(护甲)的道具系列(ArmorProp)。

5.5.2 核心关注点及系统主模块

首先来主角英雄 Hero 进行抽象。一个英雄大致有名称(name)、生命值(health)、攻击力(damage)、防御力(armor)、物品或技能（AuraAndSkill）等属性；另外还有一个用于攻击其它角色的 attack 方法，该方法返回值表示攻击是否成功；当然，英雄还应该是可复制的，因此还有一个 clone 方法，Hero 接口的内容如下所示：

```
package springroad.demo.chap5.wow;
import java.util.List;
//英雄的抽象
public interface Hero extends java.lang.Cloneable {
    boolean attack(Hero target);//攻击别人
    int getHealth();//英雄当前生命值
    void setHealth(int health);
    int getDamage();//伤害值
    void setDamage(int damage);
    List getAuraAndSkill();//持有物品或技能
    void setAuraAndSkill(List auraAndSkill);
    int getArmor();//防御力
    void setArmor(int armor);
    String getName();//名称
    void setName(String name);
    Hero clone();//克隆一个英雄
}
```

在本例中，英雄的实现 HeroImpl 比较简单，代码如下所示：

```
package springroad.demo.chap5.wow;
import java.util.List;
import java.util.Random;
public class HeroImpl implements Hero {
    private String name;// 名称
    private int health;// 生命值
    private int damage;// 被攻击
    private List auraAndSkill=new java.util.ArrayList();// 拥有物品
    private int armor;// 护甲
    private Random random = new Random();
    public HeroImpl() {
    }
    public HeroImpl(int damage) {
        this.damage = damage;
    }
```

```java
    public int getArmor() {
        return armor;
    }
    public void setArmor(int armor) {
        this.armor = armor;
    }
    public boolean attack(Hero target) {
        if (this.health < 0)
            return false;
        return random.nextInt(10) <= 6;
    }
    public int getDamage() {
        return this.damage;
    }
    public int getHealth() {
        return this.health;
    }
    public List getAuraAndSkill() {
        return auraAndSkill;
    }
    public void setAuraAndSkill(List auraAndSkill) {
        this.auraAndSkill = auraAndSkill;
    }
    public void setDamage(int damage) {
        this.damage = damage;
    }
    public void setHealth(int health) {
        this.health = health;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String toString() {
        return "[英雄名称:" + this.name + ",生命值:" + this.health + ",基
本防御:"
                + this.armor + "]";
    }
    //实现简单的浅clone
    public  Hero clone() {
        try{
        return (Hero)super.clone();
```

```
        }
        catch(Exception e)
        {
            return null;
        }
    }
    }
```

我们来注意看 attack 方法中的内容，其中“this.health < 0”用于判断英雄是否已经死亡，而返回语句“return random.nextInt(10) <= 6”表示有 60%以上的攻击成功率。这里没有攻击动作的具体实现，因为作战的规则及策略将由具体的战场来定，英雄只有放到战场中才能开始有意义的战斗。当然，也可以直接在 attack 方法中实现攻击的动作，并计算命中目标后的伤害值等，可参考本章前面 AspectJ 一节所用的回合格斗小游戏示例。

英雄持有的物品或技能主要有两种：增加攻击的物品或技能，增加防御的物品或技能。我们分别使用AttackProp及ArmorProp两个接口来表示，AttackProp有一个获得攻击力的方法getAttack，ArmorProp有一个获得防御力的方法getArmor。两个接口的内容如下面的代码所示：

```
package springroad.demo.chap5.wow;
//攻击道具或技能
public interface AttackProp {
int getAttack(Hero hero);
}

package springroad.demo.chap5.wow;
//防御道具或技能
public interface ArmorProp {
    int getArmor(Hero hero);
}
```

凡是实现了AttackProp或ArmorProp接口的物品或技能，都可以让主角持有。在本例中，我们提供了以下几个AttackProp的实现，分别是重击技能(Bash)、强击光环(TrueshotAura)、野兽卷轴(ScrollOftheBeast)；提供了两个两个ArmorProp的实现，分别是防御光环(DevotionAura)、防御卷轴(ScrollOfProtection)，你也可以自己往系统中增加各种各样的物品及技能。

技能重击Bash的源码如下所示：

```
package springroad.demo.chap5.wow;
import java.util.Random;
//重击技能，具有20%的概率，可以增加英雄的攻击力20点
public class Bash implements AttackProp {
    private boolean haveHit;
    private Random random = new Random();
    public int getAttack(Hero hero) {
        if (bigHit()) {
            haveHit = true;
        } else {
            haveHit = false;
```

```java
        }
        return (haveHit ? 20 : 0);
    }
    public boolean bigHit() {
        int rat = random.nextInt(10);
        return (rat <= 2);
    }
    public String toString() {
        return "重击技能，" + (haveHit ? "伤害提高了20点" : "未使出来");
    }
}
```

物品野兽卷轴ScrollOftheBeast的源码如下所示：

```java
package springroad.demo.chap5.wow;
//野兽卷轴，用于增加伤害百分比
public class ScrollOftheBeast implements AttackProp {
    public int getAttack(Hero hero) {
        return (int)(hero.getDamage()*0.25);
    }
    public String toString() {
        return "物品野兽卷轴，攻击增加了25%";
    }
}
```

技能强击光环TrueshotAura的源码如下所示：

```java
package springroad.demo.chap5.wow;
//强击光环，增加英雄攻击伤害百分比
public class TrueshotAura implements AttackProp {
    public int getAttack(Hero hero) {
        return (int)(hero.getDamage()*0.1);
    }
    public String toString()
    {
    return "强击光环技能，攻击力增加了10%";
    }
    }
```

技能防御光环DevotionAura的源码如下所示：

```java
package springroad.demo.chap5.wow;
//光环增加，相当于英雄的护甲，也即增加防御力
public class DevotionAura implements ArmorProp {
    public int getArmor(Hero hero) {
        return 1;
    }
    public String toString()
    {
```

```
    return  "专注光环技能，护甲增加了1点！";
    }
}
```

物品防御卷轴ScrollOfProtection的源码如下所示：

```
package springroad.demo.chap5.wow;

//卷轴增强，增加英雄的防御能力
public class ScrollOfProtection implements ArmorProp {
    public int getArmor(Hero hero) {
        return 2;
    }
    public String toString() {
        return "物品保护卷轴，护甲增加了2点";
    }
    }
```

战场中的战斗一般有对战与混战两种。对战是指有明确敌对的双方，如攻方对守方、侵略者对反侵略者、绿军对蓝军等都属于对战，参与对战的所有角色都将首先归到战斗的某一方（或称为友军），战斗过程中不能主动杀本方的角色(误杀除外)。而混战则没有这些限制，属于群雄并起，能者称霸、强者生存的性质，战场中的角色见到谁都可以主动发动攻击，谁能战斗到最后谁就是胜利者。除了基本的对战与混战以外，战场根据具体的地理位置、战争目的及参与者不同，也会有不同的实现。因此，我们对战场也作了简单抽象，使用 WarField 来表示战场，该接口的内容如下：

```
package springroad.demo.chap5.wow;
import java.util.Collection;
//战斗战场
public interface WarField {
    void join(Hero hero);//角色加入战场
    void begin();//战场战斗开始
    void attack(Hero source, Hero target);//战场中的source角色攻击target
角色
    void setHeros(Collection heros);//设置战场中的角色
    Collection getHeros();//得到战场中的角色
    }
```

这个例子中，我们只实现了一个基于混战模式的战场 WarFieldScuffle。WarFieldScuffle 中主要包含一个集合类型的 heros 属性，用于存放战场中的角色，join 方法用来往战场中增加角色，attack 是角色战斗的实际逻辑（即攻击动作的处理），begin 用来启动战场中的战斗，另外还有一个私有 RandomAttack 方法用来随机挑选攻击目标。WarFieldScuffle 的代码如下所示：

```
package springroad.demo.chap5.wow;

import java.util.Collection;
import java.util.HashSet;
import java.util.Random;
```

```
public class WarFieldScuffle implements WarField{
    private Collection heros = new HashSet();
    private Random random = new Random();
    public void join(Hero hero) {
        heros.add(hero);
    }
    public void begin() {
        while (heros.size() > 1) {
            Hero[] hs = new Hero[heros.size()];
            java.util.Iterator it = heros.iterator();
            heros.toArray(hs);
            // 开始一轮战斗，每人发一次招，攻击的对象随机产生
            for (int i = 0; i < hs.length; i++) {
                attack(hs[i], hs[RandomAttack(i, hs.length)]);
                System.out.println();
            }
            // 把已挂的角色清理出战场
            for (int i = 0; i < hs.length; i++) {
                if (hs[i].getHealth() < 0)
                    heros.remove(hs[i]);
            }
        }
    }

    public void attack(Hero source, Hero target) {
        if (source.getHealth() < 0 || (!source.attack(target)))
            return;
        int attack = source.getDamage();
        java.util.Iterator it = source.getAuraAndSkill().iterator();
        StringBuffer s = new StringBuffer("基本伤害:" + attack + ";英雄持
有以下攻击道具:");
        while (it.hasNext()) {
            Object p = it.next();
            if (p instanceof AttackProp) {
                attack += ((AttackProp) p).getAttack(source);
                s.append(p + ";");
            }
        }
        int damage = (attack - target.getArmor());
        target.setHealth(target.getHealth() - damage);
        s.append(s).append("[实际攻攻击力:").append(attack).append("]");
        s.append("——实际伤害点数:").append(damage);
        System.out.println(s.toString());
        if (target.getHealth() < 0)
```

```java
            System.out.println("英雄 " + target.getName() + " 挂了");
        }
    // 计随机选择战场中的 一个英雄来攻击
    private int RandomAttack(int current, int max) {
        int rat = random.nextInt(max);
        if (rat == current)
            return RandomAttack(current, max);
        else
            return rat;
    }
    public void setHeros(Collection heros) {
        this.heros = heros;
    }
    public Collection getHeros() {
        return heros;
    }
}
```

以上便是我们整个示例程序的核心部分(也即核心关注点或业务逻辑)，把各个部件组合到一起，我们可以得到一个如“图 5-13”所示的 UML 类图：

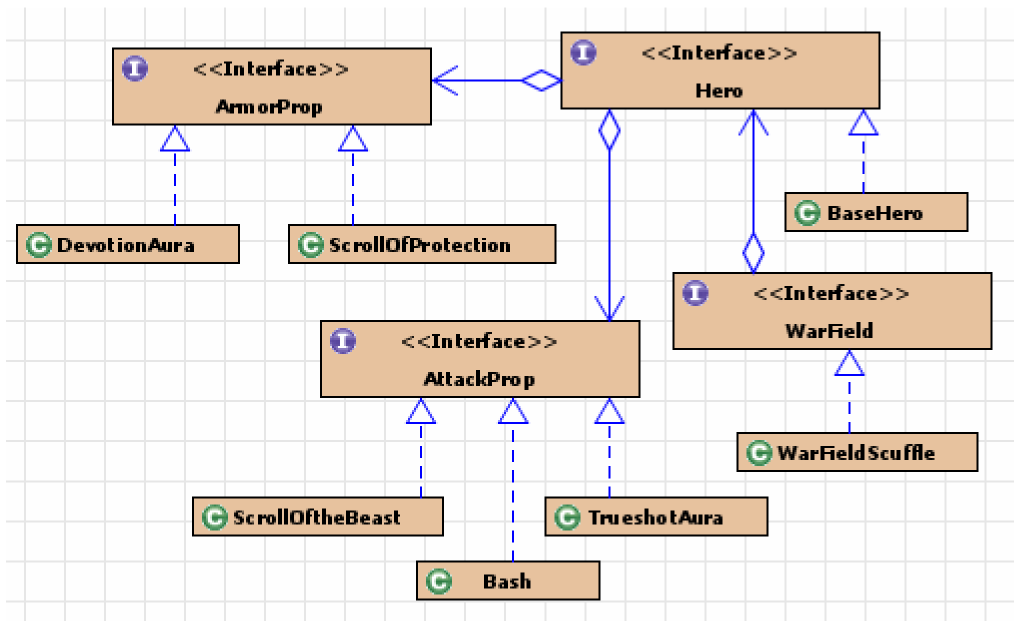

图 5-13 模拟 Warcraft 游戏主模块的 UML 类图

下面我们可以写一个简单的客户端程序来启动这个小游戏，代码如下所示：

```java
package springroad.demo.chap5.wow;
public class MainTest {
    public static void main(String[] args) {
        WarField war=new WarFieldScuffle();
        //定义英雄1—山丘之王
        Hero h1=new HeroImpl();
        h1.setName("10级山丘之王");
```

```
        h1.setHealth(500);
        h1.setArmor(1);
        h1.setDamage(60);
        h1.getAuraAndSkill().add(new Bash());
        h1.getAuraAndSkill().add(new DevotionAura());
        h1.getAuraAndSkill().add(new ScrollOftheBeast());
        h1.getAuraAndSkill().add(new TrueshotAura());
        //定义英雄2—Priestess of the Moon
        Hero h2=new HeroImpl();
        h2.setName("Priestess of the Moon");
        h2.setHealth(500);
        h2.setArmor(1);
        h2.setDamage(60);
        h2.getAuraAndSkill().add(new DevotionAura());
        h2.getAuraAndSkill().add(new ScrollOftheBeast());
        h2.getAuraAndSkill().add(new TrueshotAura());
        //英雄进入战场并启动战斗
        war.join(h1);
        war.join(h2);
        war.begin();
    }
    }
```

当然我们也可以使用 Spring 来组织装配我们的这个应用程序，实现在配置文件中灵活的角色属性及战场配置。下面是一个配置了三个角色的 Spring 配置示例 aopdemo-1.xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--定义英雄MK-->
    <bean id="MK" class="springroad.demo.chap5.wow.HeroImpl">
        <constructor-arg value="60" />
        <property name="armor" value="2" />
        <property name="health" value="500" />
        <property name="name" value="Mountain King" />
        <property name="auraAndSkill">
            <list>
                <bean class="springroad.demo.chap5.wow.Bash" />
                <bean
class="springroad.demo.chap5.wow.ScrollOfProtection" />
            </list>
        </property>
    </bean>
    <!--定义英雄POM-->
```

```xml
    <bean id="POM" class="springroad.demo.chap5.wow.HeroImpl">
        <constructor-arg value="60" />
        <property name="armor" value="1" />
        <property name="health" value="500" />
        <property name="name" value="Priestess of the Moon" />
        <property name="auraAndSkill">
            <list>
                <bean class="springroad.demo.chap5.wow.DevotionAura" />
                <bean
class="springroad.demo.chap5.wow.ScrollOfProtection" />
                <bean class="springroad.demo.chap5.wow.TrueshotAura" />
            </list>
        </property>
    </bean>
    <!--定义英雄10级山丘之王-->
    <bean id="superHero" class="springroad.demo.chap5.wow.HeroImpl">
        <constructor-arg value="60" />
        <property name="armor" value="1" />
        <property name="health" value="500" />
        <property name="name" value="10级山丘之王" />
        <property name="auraAndSkill">
            <list>
                <bean class="springroad.demo.chap5.wow.Bash" />
                <bean class="springroad.demo.chap5.wow.DevotionAura" />
                <bean
class="springroad.demo.chap5.wow.ScrollOfProtection" />
                <bean
class="springroad.demo.chap5.wow.ScrollOftheBeast" />
                <bean class="springroad.demo.chap5.wow.TrueshotAura" />
            </list>
        </property>
    </bean>
    <bean id="WarField"
class="springroad.demo.chap5.wow.WarFieldScuffle">
        <property name="heros">
            <set>
                <ref bean="MK" />
                <ref bean="POM" />
                <ref bean="superHero" />
            </set>
        </property>
    </bean>
</beans>
```

此时，使用这个程序的客户端代码就变得简单多了，如下所示：

```
public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext(
            "/springroad/demo/chap5/wow/aopdemo-1.xml");
    WarField war = (WarField) context.getBean("WarField");
    war.begin();
}
```

仔细观察这个系统，你会发现，游戏的防御系统功能没有启用，也即不管英雄持有多少个防御物品或技能，都不会影响其防御值。

5.5.3 横切关注点需求引入及实现

假如上面的程序主模块已经固定，现在我们需要对系统中加入以下几个功能。

1、 记录所有战场战斗情况，开始时间，最终胜利者等；

2、 记录各个英雄作战详细情况，包括交战发生的时间、交战双方、回合、攻击结果等；

3、另外开通防御系统功能，也即在每一次访问 Armor 属性的时候，还要加上角色身上的防御物品及技能的值。

4、我们要开通一个超级角色功能，也即满级的英雄都比较厉害。当超级角色使用一些具有一定成功概率的技能(比如 Bash)的时候，具有 100%的成功率。

针对第一个功能需要，我们发现战场的开始及结束只需要在调用 WarField 的 begin 方法前后分别插入战斗开始及战斗结果的代码即可。本例中只有一个战场实现，而实际中的战场是很多的，假如有几十种不同的战场实现，就需要在每一个 WarField 实现的 begin 前后插入相同的代码，这将是一个不小的工作量。因此，对每个战场战斗情况的关注可以归为横切性质的关注点，可以使用 AOP 编程方法来解决这一个问题，即引入一个专用于记录战场战斗情况的横切模块来解决记录并汇报战场战斗情况的信息。由于涉及到战斗开始及最终结束，也即需要在 begin 方法的前后进行处理，因此，可以使用环绕(around)增强来解决这个问题。

WarAdvice 的代码如下：

```java
package springroad.demo.chap5.wow;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class WarAdvice implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws Throwable
{
        WarField war=(WarField)invocation.getThis();
        java.util.Collection heros=war.getHeros();
        if (heros == null || heros.size() < 1)
        {
            System.out.println("战场中没有战士！");
            return null;
        }
        java.util.Iterator ih = heros.iterator();
```

```
        while (ih.hasNext())
            System.out.println(ih.next());
        System.out.println("战斗开始......");
        Object ret=invocation.proceed();//调用连接点
        java.util.Iterator it = heros.iterator();
        Hero winner = (Hero) it.next();
        System.out.println("\n最终胜利者:" + winner.getName());
        return ret;
    }
    }
```

在 WarAdvice 这个增强中，invocation.proceed()表示调用连接点上，这之前用于输出站斗开始信息，而之后输出了最终的胜利者，使用 invocation.getThis()来得实际 WarField 对象。

同需求 1 一样，对于需求 2，记录各个英雄作战的详细情况，可以通过在 Hero 的 attack 方法调用后加入用于记录战斗情况的代码来实现，也即用后置(after)增强来实现。用于 Hero 的 attack 方法后置增强的全部代码如下所示：

```
package springroad.demo.chap5.wow;
import java.lang.reflect.Method;
//记录英雄之间攻击情况
public class GameRecordAdvice implements
org.springframework.aop.AfterReturningAdvice {
    private int attackCount=0;
public void afterReturning(Object returnValue, Method method, Object[]
args, Object target) throws Throwable {
    Hero s=(Hero)target;
    Hero t=(Hero)args[0];
    attackCount++;
    boolean ret=(Boolean)returnValue;
    System.out.println("第"+attackCount+"回合:玩家["+s.getName()+"]向
["+t.getName()+"]发动攻击![命中: "+(ret?"成功":"失败")+"]");
    if(ret)System.out.println("生命值
--["+s.getName()+":"+s.getHealth()+"]"+"-["+t.getName()+":"+t.getHeal
th()+"]");
    }
}
```

在上面的代码中，使用 target 可以得到发出攻击的对象，而方法参数数组中的第 1 个参数(args[0])得到攻击目标，然后再根据连接点(本例中主要针对 Hero 的 attack 方法)的返回值，输出相应的战斗记录信息。

对于需求 3，我们不能直接改变 Hero 的 armor 属性值，因为英雄所持有的物品都是随机的，而且是多变的，armor 是一个基本属性值。然而我们可以在其它对象调用 Hero 的 getArmor 方法的时候，计算 Hero 身上防御性质的物品或技能的防御值总合，并与 armor 属性值相加，然后结果返回给调用者。虽然本例中只有不多的几个地方调用 Hero 的 getArmor 方法，但实际中调用 Hero 的 getArmor 方法的地方肯定会很多。因此，我们同样可以使用 AOP 编程方法来把这个需求封装到一个横切模块中。由于需要改变 getArmor 方法的返回值，因此可以使用一个环绕(around)增强来解决这个问题。getArmor 增强的全部代码如下所示：

```java
package springroad.demo.chap5.wow;

import org.aopalliance.intercept.MethodInvocation;
public class ArmorAdvice implements
org.aopalliance.intercept.MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws Throwable
{
        Hero hero=(Hero)invocation.getThis();
        //System.out.println("执行拦截!");
        java.util.Iterator it=hero.getAuraAndSkill().iterator();
        int armorValue=(Integer)invocation.proceed();
        String s=hero.getName()+ "--基本护甲:"+armorValue+";";
        while(it.hasNext())
        {
        Object p=it.next();
        if(p instanceof ArmorProp)
        {
        armorValue+=((ArmorProp)p).getArmor(hero);
        s+=p+";";
        }
        }
        System.out.println(s+"[实际防御:"+armorValue+"]");
        return armorValue;
    }
}
```

在上面的代码中，先通过调用 invocation.proceed()得到连接点（本例是 Hero 的 getArmor 方法）的返回值，然后再通过 target(Hero)的防御型物品或技能的属性，把防御值加到返回结果中返回。

对于需求 4，我们可以在系统中增加一个用于表示超级角色的超级接口，新增加的道具或技能实现能识别这个接口，只要发现使用当前道具的角色属于超级角色，则把出招成功率改为 100%即可。超级接口的内容如下：

```java
package springroad.demo.chap5.wow;
public interface SuperHero {
}
```

修改重击技能Bash，使得其可以识别超级接口，并调整出招成功率，修改后的getAttack方法如下：

```java
public int getAttack(Hero hero) {
        if (bigHit() || (hero instanceof SuperHero)) {
            haveHit = true;
        } else {
            haveHit = false;
        }
        return (haveHit ? 20 : 0);
    }
```

在 if 语句中可以看到，只要 hero 属于 SuperHero，则 haveHit 为 true，将增加 20 点攻击。

现在的问题是，我们如何在现有系统中，不改组件的源代码得到一个实现了 SuperHero 接口的 Hero 对象？HeroImpl 不能直接实现 SuperHero 接口，否则所有的角色都属于超级角色了。我们想到，在 AOP 编程方法中，可以通过引介来改变一个类的静态结构，让一个已有的 Java 类实现另外的接口。因此，这里我们直接使用 Spring 中的引介，再通过在配置文件把引介配置给相应的超级角色即可。这个示例中，引介的使用很简单，直接通过 DefaultIntroductionAdvisor 扩展实现，代码如下：

```
package springroad.demo.chap5.wow;

import org.springframework.aop.support.DefaultIntroductionAdvisor;
import
org.springframework.aop.support.DelegatingIntroductionInterceptor;
public class SuperHeroIntroductionAdvisor extends
DefaultIntroductionAdvisor {
    public SuperHeroIntroductionAdvisor() {
        super(new DelegatingIntroductionInterceptor(new SuperHero(){}),
SuperHero.class);
    }
}
```

上面所定义的 3 个增强(Advice)及 1 个引介器的总体 UML 结构图如“图 5-14”所示，其中接口均为 Spring 框架或 AOP 联盟所定义的。

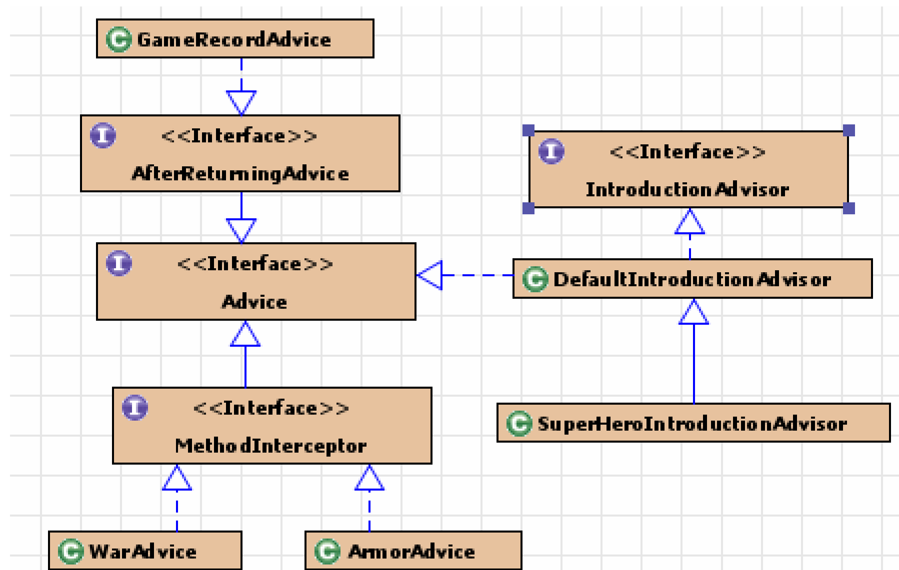

图 5-14 模拟 Warcraft 游戏用到的 AOP API 类的 UML 结构图

最后，我们需要在 Spring 配置文件中组装切入点及切面，配置完整的 AOP，并使用 ProxyFactoryBean 来定义需要拦截的英雄及战场。修改后的完整配置文件如下所示：

```
<?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">
    <!-- 技能及道具 -->
    <bean id="bash" class="springroad.demo.chap5.wow.Bash" />
    <bean id="devotionAura"
        class="springroad.demo.chap5.wow.DevotionAura" />
    <bean id="scrollOfProtection"
        class="springroad.demo.chap5.wow.ScrollOfProtection" />
    <bean id="scrollOftheBeast"
        class="springroad.demo.chap5.wow.ScrollOftheBeast" />
    <bean id="trueshotAura"
        class="springroad.demo.chap5.wow.TrueshotAura" />
    <!--定义AOP切面(包括引介器及增强器) -->
    <bean id="superHeroIntroduction"
        class="springroad.demo.chap5.wow.SuperHeroIntroductionAdvisor"
/>
    <bean id="armorAspect"

    class="org.springframework.aop.support.NameMatchMethodPointcutAdv
isor">
        <property name="mappedName" value="getArmor" />
        <property name="advice">
            <bean class="springroad.demo.chap5.wow.ArmorAdvice"></bean>
        </property>
    </bean>
    <bean id="warAspect"

    class="org.springframework.aop.support.NameMatchMethodPointcutAdv
isor">
        <property name="mappedName" value="begin" />
        <property name="advice">
            <bean class="springroad.demo.chap5.wow.WarAdvice"></bean>
        </property>
    </bean>
    <bean id="gameRecordAspect"

    class="org.springframework.aop.support.NameMatchMethodPointcutAdv
isor">
        <property name="mappedName" value="attack" />
        <property name="advice">
            <bean
class="springroad.demo.chap5.wow.GameRecordAdvice"></bean>
        </property>
    </bean>
    <!--代理配置模板-->
```

```xml
<bean id="baseHeroProxy"
    class="org.springframework.aop.framework.ProxyFactoryBean"
    abstract="true">
    <property name="proxyInterfaces"
        value="springroad.demo.chap5.wow.Hero" />
    <property name="interceptorNames">
        <list>
            <value>gameRecordAspect</value>
            <value>armorAspect</value>
        </list>
    </property>
</bean>

<!--定义英雄MK-->
<bean id="MK" parent="baseHeroProxy">
    <property name="target">
        <bean class="springroad.demo.chap5.wow.HeroImpl">
            <constructor-arg value="60" />
            <property name="armor" value="2" />
            <property name="health" value="500" />
            <property name="name" value="Mountain King" />
            <property name="auraAndSkill">
                <list>
                    <ref bean="bash" />
                    <ref bean="scrollOfProtection" />
                </list>
            </property>
        </bean>
    </property>
</bean>
<!--定义英雄POM-->
<bean id="POM" parent="baseHeroProxy">
    <property name="target">
        <bean class="springroad.demo.chap5.wow.HeroImpl">
            <constructor-arg value="60" />
            <property name="armor" value="1" />
            <property name="health" value="500" />
            <property name="name" value="Priestess of the Moon" />
            <property name="auraAndSkill">
                <list>
                    <ref bean="devotionAura" />
                    <ref bean="scrollOftheBeast" />
                    <ref bean="trueshotAura" />
                </list>
```

```xml
            </property>
        </bean>
    </property>
</bean>
<!--定义英雄superHero-->
<bean id="superHero" parent="baseHeroProxy">
    <property name="target">
        <bean class="springroad.demo.chap5.wow.HeroImpl">
            <constructor-arg value="60" />
            <property name="armor" value="1" />
            <property name="health" value="500" />
            <property name="name" value="10级山丘之王" />
            <property name="auraAndSkill">
                <list>
                    <ref bean="bash" />
                    <ref bean="devotionAura" />
                    <ref bean="scrollOftheBeast" />
                    <ref bean="trueshotAura" />
                </list>
            </property>
        </bean>
    </property>
    <property name="interceptorNames">
        <list merge="true">
            <value>superHeroIntroduction</value>
        </list>
    </property>
</bean>
<bean id="WarField"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <bean class="springroad.demo.chap5.wow.WarFieldScuffle">
            <property name="heros">
                <set>
                    <ref bean="MK" />
                    <ref bean="POM" />
                    <ref bean="superHero" />
                </set>
            </property>
        </bean>
    </property>
    <property name="interceptorNames">
        <list>
            <value>warAspect</value>
```

```
            </list>
        </property>
    </bean>
</beans>
```

客户端代码：

```
package springroad.demo.chap5.wow;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
    public class MainTest {
public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
                "/springroad/demo/chap5/wow/aopdemo.xml");
        WarField war = (WarField) context.getBean("WarField");
        war.begin();
    }
    }
```

运行这个程序，得到与“图5-15”相似的结果。

图5-15 模拟Warcraft游戏使用Spring AOP后运行结果截图

从运行结果中可以看到，战斗中有了比较详尽的战斗情况(日志)显示；防御系统也启动了，并在每次对战中显示被攻击方的防御情况；另外属于超级角色的“10级山丘之王”的重击技能(Bash)技能具有100%的出招成功率，因此其攻击是稳定的；而同样具有重击技能(Bash)技能的英雄“Priestess of the Moon”的攻击显得就不那么稳定了。

由此可见，我通过引入AOP编程方法，实现了上面所说的4个属于横切关注点的需求。当然，由于使用的是Spring老式的API配置方式，尽管我们通过模板(抽象)Bean配置来简化了一部分配置文件，然而配置文件仍然显得非常繁琐。

5.5.4 使用 AspectJ 注解支持的 AOP 实现

前面说过，在 Spring2 中，引入了 AspectJ 的切入点表达式解析引擎，并提供了对 AspectJ 的集成机制，配合使用 JDK5 的注解功能，可以大简化 AOP 应用程序的开发。下面我们将使用 Aspect 提供的注解标签，来实现上面提出的横切关注点中的 1-3 的需求，即实现角色及战场战斗情况的详细记录，并开启防御系统功能。

由于这些横切关注点的问题都比较简单，而且都是针对模拟战斗组件的增强，因此，我们可以把解决三个横切需求的功能封装到一个切面 AspectModule 中，AspectModule 是一个普通的 POJO，代码如下所示：

```
package springroad.demo.chap5.wow;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class AspectModule {
    private int attackCount=0;
    //用于针对Hero的attak方法进行拦截，记录角色相互攻击情况
    @AfterReturning(pointcut="execution(*
Hero.attack(Hero))",returning="ret")
    public void afterHeroAttack(JoinPoint thisJoinPoint,boolean ret)
    {
    Hero s=(Hero)thisJoinPoint.getTarget();
    Hero t=(Hero)thisJoinPoint.getArgs()[0];
    attackCount++;
    System.out.println("第"+attackCount+"回合:玩家["+s.getName()+"]向
["+t.getName()+"]发动攻击![命中: "+(ret?"成功":"失败")+"]");
    if(ret)System.out.println("生命值
--["+s.getName()+":"+s.getHealth()+"]"+"-["+t.getName()+":"+t.getHeal
th()+"]");
    }
    //用于针对WarField的begin方法进行拦截，
    @Around("execution(void WarField.begin())")
    public Object aroundWarFieldBegin(ProceedingJoinPoint
theJoinPoint)throws Throwable
    {
        WarField war=(WarField)theJoinPoint.getThis();
        java.util.Collection heros=war.getHeros();
        if (heros == null || heros.size() < 1)
```

```
        {
            System.out.println("战场中没有战士！");
            return null;
        }
        java.util.Iterator ih = heros.iterator();
        while (ih.hasNext())
            System.out.println(ih.next());
        System.out.println("战斗开始......");
        Object ret=theJoinPoint.proceed();//调用连接点
        java.util.Iterator it = heros.iterator();
        Hero winner = (Hero) it.next();
        System.out.println("\n最终胜利者:" + winner.getName());
        return ret;
    }
    //用于针对WarField的getArmor方法进行拦截，
    @Around("execution(public int Hero.getArmor())")
    public Object aroundHeroGetDamage(ProceedingJoinPoint
theJoinPoint)throws Throwable
    {
    Hero hero=(Hero)theJoinPoint.getThis();
    java.util.Iterator it=hero.getAuraAndSkill().iterator();
    int armorValue=(Integer)theJoinPoint.proceed();
    String s=hero.getName()+ "--基本护甲:"+armorValue+";";
    while(it.hasNext())
    {
    Object p=it.next();
    if(p instanceof ArmorProp)
    {
    armorValue+=((ArmorProp)p).getArmor(hero);
    s+=p+";";
    }
    }
    System.out.println(s+"[实际防御:"+armorValue+"]");
    return armorValue;
    }
}
```

通过上面的代码可以看出，AspectModule 这个切面模块不用实现任何框架指定的接口，这个 POJO 中将引入 aspectj 的注解标签库，在类名前面使用@Aspect 标签表示这个类是一个切面。在切面中，我们使用@AfterReturning 及@Around 标识横切关注点中的逻辑，即增强，标签中的参数 pointcut 的值即为切入点表示式语言。比如：

```
  @AfterReturning(pointcut="execution(*
Hero.attack(Hero))",returning="ret")
    public void afterHeroAttack(JoinPoint thisJoinPoint,boolean ret)
     {
```

…

}

这段代码定义了一个切入点，以及一个方法后置增加，具体的切入点含义由 pointcut 参数的值“execution(* Hero.attack(Hero))”决定，这里表示 Hero 对象的 attack(Hero)方法调用；returning="ret"用来标识返回参数，字符串 ret 与下面的方法定义中的 ret 参数对应。

在 afterHeroAttack 方法中，第一个参数代表当前的连接点，类型为 org.aspectj.lang.JoinPoint，通过这个参数我们可以得到当前连接点的相关信息，如目标对象、方法的参数等，第二个参数ret表示方法调用的返回值。在afterHeroAttack中，通过连接点的getTarget()及getArgs()方法得到当前连接点的目标对象，以及方法的参数，由于参数为数组类型，而且我们也知道Hero的attack方法只带有一个参数，因此可以直接使用(Hero)thisJoinPoint.getArgs()[0]得到attack方法中的参数。如下所示：

```
Hero s=(Hero)thisJoinPoint.getTarget();
Hero t=(Hero)thisJoinPoint.getArgs()[0];
```

我们再来看针对 getArmor 的环绕增强的实现，代码如下：

```
@Around("execution(public int Hero.getArmor())")
public Object aroundHeroGetDamage(ProceedingJoinPoint
theJoinPoint)throws Throwable
{
…
}
```

通过@Around标签来标识一个环绕增强，标签中的参数即为切入点，"execution(public int Hero.getArmor())"表示Hero的getArmor方法调用。在下面的增强的实现方法中，包含一个类型为org.aspectj.lang.ProceedingJoinPoint的参数，表示环绕增强的连接点，我们可以通过该连接点得到调用的相关信息，如目标对象、参数值、方法名等，更重要的是通过 ProceedingJoinPoint的proceed()来启动连接点的执行。在这个示例中，通过语句int armorValue=(Integer)theJoinPoint.proceed()得到Hero对象的armor原始属性值。然后再根据当前Hero所持有防御类物品或技能进一步计算当前对象的附加防御值，最后把原始属性值与附加防御值加在一起返回给调用者。

针对WarField.begin()的拦截比较简单，在切面中通过如下方式定义：

```
//用于针对WarField的begin方法进行拦截，
@Around("execution(void WarField.begin())")
public Object aroundWarFieldBegin(ProceedingJoinPoint
theJoinPoint)throws Throwable
{
…
}
```

在完成了切面模块的开发以后，我们可以直接在配置文件中使用 <aop:aspectj-autoproxy/>标签来开启自动代理，然后在配置文件中声明一个普通的 AspectModule Bean即可。配置文件如下：

```
<?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-2.0.xsd
        http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
<!-- 使用@AspectJ自动代理-->
<aop:aspectj-autoproxy/>
<!--使用普通Bean方式定义一个切面模块-->
<bean id="aspectModuel"
class="springroad.demo.chap5.wow.AspectModule"></bean>
<!--定义英雄MK-->
<bean id="MK" class="springroad.demo.chap5.wow.HeroImpl">
        <constructor-arg value="60" />
        <property name="armor" value="2" />
        <property name="health" value="500" />
        <property name="name" value="Mountain King" />
        <property name="auraAndSkill">
            <list>
                <bean class="springroad.demo.chap5.wow.Bash" />
                <bean
class="springroad.demo.chap5.wow.ScrollOfProtection" />
            </list>
        </property>
    </bean>
    <!--定义英雄POM-->
    <bean id="POM" class="springroad.demo.chap5.wow.HeroImpl">
        <constructor-arg value="60" />
        <property name="armor" value="1" />
        <property name="health" value="500" />
        <property name="name" value="Priestess of the Moon" />
        <property name="auraAndSkill">
            <list>
                <bean class="springroad.demo.chap5.wow.DevotionAura" />
                <bean
class="springroad.demo.chap5.wow.ScrollOfProtection" />
                <bean class="springroad.demo.chap5.wow.TrueshotAura" />
            </list>
        </property>
    </bean>
    <!--定义英雄10级山丘之王-->
    <bean id="superHero" class="springroad.demo.chap5.wow.HeroImpl">
        <constructor-arg value="60" />
        <property name="armor" value="1" />
        <property name="health" value="500" />
```

```
        <property name="name" value="10级山丘之王" />
        <property name="auraAndSkill">
            <list>
                <bean class="springroad.demo.chap5.wow.Bash" />
                <bean class="springroad.demo.chap5.wow.DevotionAura" />
                <bean
class="springroad.demo.chap5.wow.ScrollOfProtection" />
                <bean
class="springroad.demo.chap5.wow.ScrollOftheBeast" />
                <bean class="springroad.demo.chap5.wow.TrueshotAura" />
            </list>
        </property>
    </bean>
    <bean id="WarField"
class="springroad.demo.chap5.wow.WarFieldScuffle">
        <property name="heros">
            <set>
                <ref bean="MK" />
                <ref bean="POM" />
                <ref bean="superHero" />
            </set>
        </property>
    </bean>
</beans>
```

这个配置文件比aopdemo.xml这个文件内容少了很多，基本上全部是普通的Bean配置，而且切面封装模块是一个简单的POJO，也不用依赖于Spring框架，使得整个系统的结构更加清晰及简洁。当然，由于需要在AspectJ的切面模块通过用AspectJ的注解标签来定义切面相关信息，所以对AspectJ仍然有一定依赖。

注意：由于 Spring 底层使用运行时字节码生成机制，运行上面的程序需要在 ClassPath 上添加 asm 相关包，如 asm-2.2.2.jar、asm-commons-2.2.2.jar 等，当然还要添加 AspectJ 的 aspectjweaver.jar。

运行程序可以发现，我们实现了前面所列出的横切关注点 1、2、3 的需求。也即通过 AOP 切面达到了对战斗系统的记录，并通过 AOP 切面启动了防御系统功能。

这里没有实现需求 4，即引入超级角色(SuperHero)的功能，主要是因为引介是改变类的静态结构，而本例中需要的引介有点特殊，只是给类的某一个或部分实例添加 SuperHero 接口，而并不是要让 HeroImpl 实现 SuperHero 接口，无法通过直接在横切模块中定义引介实现。

当然，若要在不修改的 HeroImpl 代码的情况下让 HeroImpl 类实现 SuperHero 接口，则可以在切面模块 AspectModule 中添加如下的代码即可：

```
    @DeclareParents(value="springroad.demo.chap5.wow.HeroImpl",def
aultImpl=SuperHeroImpl.class)
    private SuperHero superHero;
```

其中第一行@DeclareParents 标签是 aspectJ 中的标签，可以用来指定一个现有类实现某个接口，或者为现有类指定一个父类。@DeclareParents 的 value 属性值为匹配类的切入点表达式，defaultImpl 属性值指定一个目标接口的实现类，这里 SuperHeroImpl.class 是 SuperHero 接口的一个空实现，没有任何内容；标签后面的属性是声明是 AspectJ 注解的一种语法形式使得整个系统的结构更加清晰及简洁。

假如你在上面的横切模块 AspectModule 中加入了上面的代码，再运行客户端示例程序，你会发现战场中的所有英雄都变成了超级角色，“Mountain King”以及“10 级山丘之王”的重击技能的出招成功率都变成了 100%，也即都实现了 SuperHero 接口，这跟我们的需求有一定的差异。

5.5.5 使用基于 Schema 的方式配置 Spring AOP

当然，如果不使用@AspectJ 注解方式来封装 AOP 切面模块，可以直接使用基于 Schema 的配置文件方式，通过 Spring 提供的 spring-aop-2.0.xsd，使用命名空间<aop:xxx>在配置文件中直接配置切面。

同样是基于 POJO 的方式来封装切面模块，书写横切关注点的业务逻辑。本例中我们通过 AspectJModule2 这个类来封装模拟游戏中的切面逻辑，直接把上面 AspectJModule 中的实现代码拷过来，然后删除掉与@AspectJ 注解相关的各种信息，得到一个传统的 POJO。AspectJModule2 的代码如下所示：

```
package springroad.demo.chap5.wow;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
public class AspectModule2 {
    private int attackCount=0;
    //用于针对Hero的attak方法进行拦截，记录角色相互攻击情况
    public void afterHeroAttack(JoinPoint thisJoinPoint,boolean ret)
    {
    Hero s=(Hero)thisJoinPoint.getTarget();
    Hero t=(Hero)thisJoinPoint.getArgs()[0];
    attackCount++;
    System.out.println("第"+attackCount+"回合:玩家["+s.getName()+"]向
["+t.getName()+"]发动攻击![命中: "+(ret?"成功":"失败")+"]");
    if(ret)System.out.println("生命值
--["+s.getName()+":"+s.getHealth()+"]"+"-["+t.getName()+":"+t.getHeal
th()+"]");
    }
    //用于针对WarField的begin方法进行拦截
    public Object aroundWarFieldBegin(ProceedingJoinPoint
theJoinPoint)throws Throwable
    {
        WarField war=(WarField)theJoinPoint.getThis();
        java.util.Collection heros=war.getHeros();
        if (heros == null || heros.size() < 1)
```

```
        {
            System.out.println("战场中没有战士!");
            return null;
        }
        java.util.Iterator ih = heros.iterator();
        while (ih.hasNext())
            System.out.println(ih.next());
        System.out.println("战斗开始......");
        Object ret=theJoinPoint.proceed();//调用连接点
        java.util.Iterator it = heros.iterator();
        Hero winner = (Hero) it.next();
        System.out.println("\n最终胜利者:" + winner.getName());
        return ret;
    }
    //用于针对WarField的getArmor方法进行拦截
    public Object aroundHeroGetDamage(ProceedingJoinPoint
theJoinPoint)throws Throwable
    {
    Hero hero=(Hero)theJoinPoint.getThis();
    java.util.Iterator it=hero.getAuraAndSkill().iterator();
    int armorValue=(Integer)theJoinPoint.proceed();
    String s=hero.getName()+ "--基本护甲:"+armorValue+";";
    while(it.hasNext())
    {
    Object p=it.next();
    if(p instanceof ArmorProp)
    {
    armorValue+=((ArmorProp)p).getArmor(hero);
    s+=p+";";
    }
    }
    System.out.println(s+"[实际防御:"+armorValue+"]");
    return armorValue;
    }
}
```

然后使用下面信息来直接在配置文件中定义切面封装，AOP 配置信息位于<aop:config>标签中。如下所示：

```
<aop:aspect ref="aspectModuel">
<aop:after-returning pointcut="execution(boolean
springroad.demo.chap5.wow.Hero.attack(..))" returning="ret"
method="afterHeroAttack"/>
<aop:around pointcut="execution(void
springroad.demo.chap5.wow.WarField.begin())"
method="aroundWarFieldBegin"/>
```

```
<aop:around pointcut="execution(public int
springroad.demo.chap5.wow.Hero.getArmor())"
method="aroundHeroGetDamage"/>
    </aop:aspect>
```

其中<aop:aspect>标签用来定义一个切面，ref 属性指向一个普通的 Spring Bean；<aop:after-returning> 及 <aop:around> 标签用来定义各个增强，即对 Hero.attack(Hero)、Hero.getArmor()及 WarField.begin 方法增强，标签的 pointcut 属性是基于 AspectJ 切入点描述表达定义的切入点信息，method 属性是增强所对应的方法。

aopdemo-3.xml 是完整的配置文件内容，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-2.0.xsd
       http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
<aop:config>
<aop:aspect ref="aspectModuel">
<aop:after-returning pointcut="execution(boolean
springroad.demo.chap5.wow.Hero.attack(..))" returning="ret"
method="afterHeroAttack"/>
<aop:around pointcut="execution(void
springroad.demo.chap5.wow.WarField.begin())"
method="aroundWarFieldBegin"/>
<aop:around pointcut="execution(public int
springroad.demo.chap5.wow.Hero.getArmor())"
method="aroundHeroGetDamage"/>
</aop:aspect>
</aop:config>
<!--使用普通Bean方式定义一个切面模块-->
<bean id="aspectModuel"
class="springroad.demo.chap5.wow.AspectModule2"></bean>
<!--定义英雄MK-->
<bean id="MK" class="springroad.demo.chap5.wow.HeroImpl">
        <constructor-arg value="60" />
        <property name="armor" value="2" />
        <property name="health" value="500" />
        <property name="name" value="Mountain King" />
        <property name="auraAndSkill">
            <list>
                <bean class="springroad.demo.chap5.wow.Bash" />
                <bean
```

```
class="springroad.demo.chap5.wow.ScrollOfProtection" />
            </list>
        </property>
    </bean>
    <!--定义英雄POM-->
    <bean id="POM" class="springroad.demo.chap5.wow.HeroImpl">
        <constructor-arg value="60" />
        <property name="armor" value="1" />
        <property name="health" value="500" />
        <property name="name" value="Priestess of the Moon" />
        <property name="auraAndSkill">
            <list>
                <bean class="springroad.demo.chap5.wow.DevotionAura" />
                <bean
class="springroad.demo.chap5.wow.ScrollOfProtection" />
                <bean class="springroad.demo.chap5.wow.TrueshotAura" />
            </list>
        </property>
    </bean>
    <!--定义英雄10级山丘之王-->
    <bean id="superHero" class="springroad.demo.chap5.wow.HeroImpl">
        <constructor-arg value="60" />
        <property name="armor" value="1" />
        <property name="health" value="500" />
        <property name="name" value="10级山丘之王" />
        <property name="auraAndSkill">
            <list>
                <bean class="springroad.demo.chap5.wow.Bash" />
                <bean class="springroad.demo.chap5.wow.DevotionAura" />
                <bean
class="springroad.demo.chap5.wow.ScrollOfProtection" />
                <bean
class="springroad.demo.chap5.wow.ScrollOftheBeast" />
                <bean class="springroad.demo.chap5.wow.TrueshotAura" />
            </list>
        </property>
    </bean>
    <bean id="WarField"
class="springroad.demo.chap5.wow.WarFieldScuffle">
        <property name="heros">
            <set>
                <ref bean="MK" />
                <ref bean="POM" />
                <ref bean="superHero" />
```

```
            </set>
        </property>
    </bean>
</beans>
```

跟基于 AspectJ 注解实现的配置一样，配置文件减少了很多内容，由于可以在配置文件中灵活定义切入点，定义用于切面的普通 Spring Bean，以及指定处理横切关注点实现方法。由此可见，相对于 Spring 以前的 AOP 实现来说，借助于 AspectJ，Spring2 的 AOP 功能更加灵活、使用更加简单了。

5.6 小结

本章主要讲解了 AOP 的相关概念，Spring AOP 的各种使用方法、实现原理及一些简单技巧等，另外还介绍了 Java 领域功能强大的 AOP 实现 AspectJ 的应用，以便我们更好的掌握和学习使用 Spring AOP。

5.7 思考题

对 Spring2 提供三种 AOP 使用方式进行对比分析，各自的优缺点是什么？
Java 代理及拦截器机制的是如何来实现 AOP 功能的？
在一个新闻发布系统应用程序中，有哪功能需求属于横切交叉关注点需求？