AspectJ Cookbook

AspectJ Cookbook™ 中文版

Russ Miles 著

程利剑 译

O'REILLY®

清华大学出版社

Java

O'REILLY®

AspectJ Cookbook 中文版

当面向对象编程（OO）首次出现时，它是一种启示。OO使开发人员能够创建更灵活、更健壮的软件。但随着时间的推移，应用程序变得更复杂，人们发现“传统的”OO体系结构在某些领域不能胜任。面向方面编程（AOP）通过进一步扩展OO方法，解决了这些问题。

许多开发人员都对AOP感兴趣——特别是对AspectJ感兴趣，这个Java编程语言的开源扩展明确支持AOP方法。然而，尽管AspectJ纳入在Eclipse这个日益流行的Java开源IDE之内，但是找到一种实用、非理论的方式来学习这种语言及其他AOP工具和技术已经成为一个实际的问题。

迄今为止，本书提供了实用的解决方案以及大量的代码和秘诀，便于使用AOP的独特方法来解决日常设计和编程问题。本书包括以下内容：

- 初识AOP
- 集成AspectJ与Eclipse、Ant和Java命令行工具
- 将面向方面的应用程序部署为servlet、JSP、Web服务等
- 关于方面、切入点、通知以及核心AOP构造的每个方面的秘诀
- 在Java和J2EE设计模式实现中使用方面
- 使用AOP特有的设计模式，增强现有的Java应用程序

本书展示了为什么以及如何使用AOP技术来解决常见的Java开发问题。本书利用受欢迎的问题/解决方案/讨论格式，介绍了许多现实的示例，不仅仅从概念角度演示AOP；它是一个开发过程，让用户以一种即时、可见的方式受益。如果想了解AOP如何改变软件开发的方式，如何使用AspectJ使代码更加模块化，更易于开发、维护、演进和部署，那么本书确实很适合你。

ISBN 7-302-14011-1

9 787302 140115 >

O'Reilly Media, Inc.授权清华大学出版社出版

ISBN 7-302-14011-1

定价：40.00元

AspectJ Cookbook™

中文版

Russ Miles 著

程利剑 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权清华大学出版社出版

清华大学出版社

图书在版编目（CIP）数据

AspectJ Cookbook™ 中文版 /（美）米尔斯（Miles, R.）著；程利剑译．－北京：清华大学出版社，2006.11

书名原文：AspectJ Cookbook

ISBN 7-302-14011-1

I. A… II. ①米… ②程… III. JAVA 语言－程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2006)第 121805 号

北京市版权局著作权合同登记

图字：01-2006-5903 号

Copyright ©2004 by O'Reilly Media, Inc.

Authorized Simplified Chinese translation edition, by O'Reilly Media, Inc., is published by Tsinghua University Press, 2006. Authorized translation of the original English edition, 2004 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书之英文原版由 O'Reilly Media, Inc. 于 2004 年出版。

本中文简体翻译版由 O'Reilly Media, Inc. 授权清华大学出版社于 2006 年出版。此翻译版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc. 的许可。

版权所有，未经书面许可，本书的任何部分和全部不得以任何形式复制。

版权所有，翻印必究。举报电话：010-62782989　13501256678　13801310933
本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
本书防伪标签采用特殊防伪技术，用户可通过在图案表面涂抹清水，图案消失，水干后图案复现；或将表面膜揭下，放在白纸上用彩笔涂抹，图案在白纸上再现的方法识别真伪。

书　　名 / AspectJ Cookbook™ 中文版
书　　号 / ISBN 7-302-14011-1/TP・8415
责任编辑 / 常晓波
封面设计 / Emma Colby，张健
出版发行 / 清华大学出版社（www.tup.tsinghua.edu.cn）
地　　址 / 北京清华大学学研大厦（邮政编码 100084）
经　　销 / 各地新华书店
印　　刷 / 清华大学印刷厂
装　　订 / 三河市李旗庄少明装订厂
开　　本 / 178 毫米×233 毫米　16 开本　20.75 印张　451 千字
版　　次 / 2006 年 11 月第一版　2006 年 11 月第1 次印刷
印　　数 / 1～3000 册
定　　价 / 40.00 元（册）

作者简介

Russ Miles 是一位具有 10 年编程经验的开发人员，其涉足的语言包括：从 Atari 800XL 上的 Basic 到 Mac OS X 上的 Mono。Russ 是一位虔诚的 Mac 信徒，近年来，他专注于多平台环境下复杂的分布式系统开发，包括 Windows、Linux、Unix 和 Mac OS X 上多种风格的开发。Russ 目前是 General Dynamic UK Limited 的一位高级技术专家，还是许多项目的顾问，包括从内部工具开发到国际标准出版物。只要有可能，Russ 就会回到编程领域中来，并为多个开源项目做出了贡献。他还创立了他自己的一些开源项目。

在 Russ 意外地参加英国计算机协会的高级编程专家组的一次错误的会议时，他“发现”了面向方面编程。Russ 无法记起他参加会议的实际目的，但是他很幸运，他参加了一次名为“Aspect-Oriented Programming Using AspectJ”的讲座，演讲人是 IBM 的 Adrian Colyer。Russ 对面向方面的潜力和 AspectJ 的成熟印象深刻，他关于软件体系结构的观点也发生了永久性的变化。他立即利用这种新方法，改变了他及其团队开发软件的方式。很快两年过去了，Russ 仍在开发和倡导面向方面的方法。本书是 Russ 幸运地与 AspectJ 倡导者第一次碰面的结晶。

Russ持有格林威治大学计算机科学的优等学士学位，目前是牛津大学Kellogg学院的成员，他在这里完成了他的计算机科学硕士学位。他还为 ONJava.com 和 java.net 写了许多文章，并且是 O'Reilly 的技术顾问。

封面介绍

本书封面上的动物是一只长尾猴。长尾猴是一种小毛猴（其大小不会超过硕大的老鼠），居住在南美洲的热带和亚热带森林里。它们是树栖动物，并且异常敏捷。它们强有力的后腿、灵巧得像爪子一样的手以及特别长的尾巴，使得它们可以轻而易举地从一根树枝摆动到另一根树枝。

长尾猴不太挑食。它们偏爱树液（树越粗越好），但是它们也享用蜥蜴、青蛙、蜗牛、昆虫、水果和花蜜。

O'Reilly Media, Inc. 介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc. 授权清华大学出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的 50 本书之一）到 GNN（最早的 Internet 门户和商业网站），再到 WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商 —— 每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体 —— 他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。

目录

前言 1

第 1 章 面向方面概述 7

面向方面简史 7

AspectJ 7

面向方面的定义 8

在哪里找到更多信息 10

第 2 章 初识 AspectJ 12

2.0 简介 12

2.1 安装 AspectJ 13

2.2 开发一个简单的方面 15

2.3 编译一个方面和多个 Java 文件 18

2.4 织入方面到 jar 中 19

2.5 在加载时织入方面 20

2.6 生成 Javadoc 文档 23

2.7 使用 Eclipse 编译一个 AspectJ 项目 26

2.8 选择织入 Eclipse 内的构建中的方面 30
2.9 使用 Ant 构建一个 AspectJ 项目 34

第 3 章 部署 AspectJ 应用程序 36
3.0 简介 36
3.1 部署命令行 AspectJ 应用程序 36
3.2 将 AspectJ 应用程序部署为全包含的可执行 JAR 文件 39
3.3 部署使用 AspectJ 的 Java Servlet 41
3.4 部署使用 AspectJ 的 JSP 46
3.5 部署使用 AspectJ 的 Axis Web 服务 51

第 4 章 捕获方法上的连接点 55
4.0 简介 55
4.1 捕获方法调用 55
4.2 捕获方法调用上传递的参数值 59
4.3 捕获方法调用的目标 61
4.4 当执行一个方法时捕获它 62
4.5 在执行方法时捕获 this 引用的值 64

第 5 章 捕获异常处理上的连接点 66
5.0 简介 66
5.1 捕获何时捕捉异常 66
5.2 捕获抛出的异常 68
5.3 捕获处理异常的对象 70

第 6 章 捕获通知上的连接点 72
6.0 简介 72
6.1 捕获何时执行通知 72
6.2 排除作为通知执行结果的连接点 74
6.3 在发出通知时展示原始连接点 77

第 7 章 捕获类和对象构造上的连接点 79
7.0 简介 79
7.1 捕获对构造函数的调用 79
7.2 在执行构造函数时捕获它 81
7.3 捕获何时初始化对象 83
7.4 捕获何时将要初始化一个对象 86
7.5 捕获何时初始化类 88

第 8 章 捕获属性上的连接点 90
8.0 简介 90
8.1 捕获何时访问对象的属性 90
8.2 捕获访问的字段值 93
8.3 捕获何时修改对象的字段 94
8.4 在修改字段时捕获它的值 96

第 9 章 捕获程序作用域内的连接点 98
9.0 简介 98
9.1 捕获特定类中的所有连接点 98
9.2 捕获特定包中的所有连接点 100
9.3 捕获特定方法内的所有连接点 101

第 10 章 捕获基于控制流程的连接点 104
10.0 简介 104
10.1 捕获通过初始连接点开始的程序控制流程内的所有连接点 104
10.2 捕获程序控制流程内的所有连接点，不包括初始连接点 107

第 11 章 捕获基于对象类型的连接点 110
11.0 简介 110
11.1 捕获何时 this 引用是一个特定的类型 110

11.2 捕获何时连接点的目标对象是特定的类型 113
11.3 捕获何时连接点的参数是某个数字、类型和次序 116

第 12 章 捕获基于布尔或复合表达式的连接点 120
12.0 简介 120
12.1 捕获何时连接点上的运行时条件评估为 true 120
12.2 使用逻辑 AND（&&）结合切入点 122
12.3 使用逻辑 OR（||）结合切入点 125
12.4 捕获所有未通过切入点声明指定的连接点 128
12.5 声明匿名切入点 129
12.6 重用切入点 131

第 13 章 定义通知 134
13.0 简介 134
13.1 访问类成员 134
13.2 访问连接点环境 136
13.3 在连接点之前执行通知 139
13.4 在连接点周围执行通知 140
13.5 在连接点之后无条件执行通知 143
13.6 仅在从连接点正常返回之后才执行通知 144
13.7 仅当连接点中引发了一个异常之后才执行通知 147
13.8 控制通知优先级 148
13.9 通知方面 150

第 14 章 定义方面实例化 152
14.0 简介 152
14.1 定义单件方面 152
14.2 在每个实例上定义一个方面 154
14.3 在每个控制流程上定义一个方面 158

第 15 章 定义方面的关系 161

15.0 简介 161
15.1 继承切入点定义 161
15.2 实现抽象切入点 163
15.3 把类继承进方面中 165
15.4 在类内部声明方面 167

第 16 章 增强类和编译器 170

16.0 简介 170
16.1 扩展现有的类 170
16.2 声明类之间的继承关系 171
16.3 使用方面实现接口 172
16.4 声明默认的接口实现 174
16.5 减轻异常的影响 175
16.6 扩展编译 177

第 17 章 实现创建型面向对象设计模式 179

17.0 简介 179
17.1 实现单件模式 180
17.2 实现原型模式 183
17.3 实现抽象工厂模式 185
17.4 实现工厂方法模式 187
17.5 实现生成器模式 189

第 18 章 实现结构型面向对象设计模式 191

18.0 简介 191
18.1 实现复合模式 191
18.2 实现享元模式 195
18.3 实现适配器模式 198

18.4 实现桥接模式 200
18.5 实现装饰模式 202
18.6 实现代理模式 205

第 19 章 实现行为型面向对象设计模式 210
19.0 简介 210
19.1 实现观察者模式 210
19.2 实现命令模式 214
19.3 实现迭代器模式 219
19.4 实现调停者模式 221
19.5 实现责任链模式 224
19.6 实现备忘录模式 229
19.7 实现策略模式 232
19.8 实现访问者模式 236
19.9 实现模板方法模式 240
19.10 实现状态模式 241
19.11 实现解释器模式 243

第 20 章 应用类和组件级方面 245
20.0 简介 245
20.1 验证传递给方法的参数 245
20.2 重写在构造函数调用上实例化的类 249
20.3 添加持久性到类中 252
20.4 应用模拟组件支持单元测试 256

第 21 章 应用应用程序级方面 261
21.0 简介 261
21.1 应用面向方面的跟踪 262
21.2 应用面向方面的日志记录 265

21.3 应用延迟加载 269
21.4 管理应用程序属性 274

第22章 应用企业级方面 280
22.0 简介 280
22.1 应用开发指导和规则 280
22.2 应用事务 283
22.3 应用资源池 286
22.4 使用RMI透明地远程访问类 290
22.5 应用安全策略 292

第23章 应用面向方面设计模式 294
23.0 简介 294
23.1 应用Cuckoo's Egg设计模式 295
23.2 应用Director设计模式 297
23.3 应用Border Control设计模式 300
23.4 应用策略设计模式 303

附录 AspectJ运行时API 307

前言

本书重点关注的是用AspectJ完成任务。面向方面的软件开发（Aspect-Oriented Software Development，AOSD）给纳入了面向对象方法的某些旧思想贴上了新的商标。这个相当复杂的定义实际上概括了一种灵巧的方法，用于解决一些传统的面向对象的问题。

虽然本书并没有深入研究面向方面背后的理论细节，但是提供了一个简短的概述，从而给你提供了一个有用的基础，用以支持构成本书余下部分的编码秘诀。这个编码秘诀将指导你如何为面向方面的开发建立系统，构建你的第一个小程序，并最终把面向方面应用于复杂的现实问题。

本书的目标是成为你案头的那些有用书籍之一，并定期被要求“给我展示如何执行那个任务”。你应该能够在需要时直接跳到所需的秘诀上，以与O'Reilly Cookbook系列的主旨即“取精去粗”方法保持一致。在记住这一点后，本书中介绍的主题包括：

- 简述面向方面和 AspectJ
- 在广泛的构建和目标环境中建立、认真处理和运行 AspectJ 程序
- 实际地检查针对 Java 语言的每个 AspectJ 扩展
- 将 AspectJ 应用于可从面向方面的方法中获益的现实软件开发问题

本书读者对象

虽然可以说本书足以满足有兴趣学习 AspectJ 和面向方面的任何人的需要，但是本书对于那些在面向对象设计方面经验相当丰富的人最有用，特别是当利用Java实现时则更是如此。可以用新的方法来解决一些传统的面向对象方法的问题，对新方法持开放态度的

任何人都应该会在本书中找到一些内容，这些内容鼓励他们在自己的应用程序中尝试AspectJ。

关于本书

本书涵盖了以下主题：

- 概述使用AspectJ的面向方面的各种元素、切入点（pointcut）、连接点（join point）和通知（advice）
- 建立你自己的环境，并开始AspectJ开发
- 在多种不同的环境中运行基于AspectJ的软件
- 把AspectJ应用程序部署到变化的目标环境中
- 如何使用AspectJ支持的多种形式的切入点
- 如何使用AspectJ中可用的多种形式的通知
- 控制如何创建和销毁方面
- 使用AspectJ更改软件和支持工具的静态结构
- 使用AspectJ增强现有的面向对象的设计模式实现
- 将方面应用于组件、系统和企业领域内的现实问题

很少见到从头至尾一页不漏地阅读cookbook图书，但是关于每一节重点的概述是有用的，它们便于理解每一章的地位及其意义。这本cookbook被组织成4大部分，如图P-1所示。

第1~3章提供了一组秘诀，用于让你初识支持使用AspectJ开发的工具和环境。这几章介绍了如何使用命令行工具、Eclipse，甚至ANT来开发面向方面的应用程序，并通过展示如何部署到各种各样的目标环境（如Java Servlet、Java Server Page和Web服务）来结束这几章的介绍。

第4~16章提供了AspectJ语言的深度语法分析。这些章旨在提供一个有用、详尽的参考，以理解如何使用AspectJ语言添加到Java中的不同构造。

第17~23章充分展示了AspectJ超过传统的面向对象软件实现的优点。在这几章中，使用方面来增强和改进各种软件解决方案的设计，包括从设计模式实现到企业应用程序。第23章通过介绍一些面向方面的设计模式来结束全书的内容，这些设计模式是作为工具（如AspectJ）逐渐发现的，从而把面向方面引入软件开发的主流。

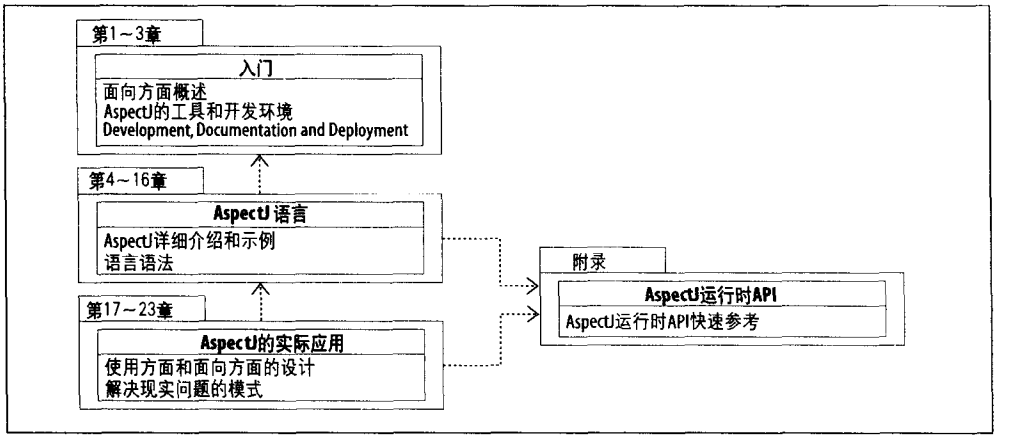

图 P-1：本书各章打算说明什么

既然你知道了本书介绍了哪些内容，现在就应该解释本书没有解释什么。如前所述，本书不是对 AspectJ 的理论研究。本书中的代码随处可见，以确保每个领域都有实际的关注。有许多优秀的在线文章，并且它们会逐渐出现在研究OA的更关注理论的印刷品中，因此，本书不会尝试论及这些方面。

本书也没有声明是对AspectJ的简要介绍。AspectJ处在一种非常稳定的状态中，从而具有早期的实现，但是甚至现在对该语言和技术的扩展仍在考虑中，因此本书中的某些领域可能比其他一些领域更成熟。为了尝试确保本书具有尽可能最长的寿命，在编写本书时，只包括了最稳定的领域。

本书所做的假定

下面假定你具有对面向方面开发的相关支持工具、语言以及概念的了解和使用它们的经验。

- 假定你具有 Java 语言的相当高级的知识
- 假定你的系统上具有现有的 Java Source Development Kit（Java 源代码开发工具包，SDK），它至少应该是 JSDK 1.2，最好是 1.4+
- 假定你对Java Server Page、Java Servlet和Web服务（确切地讲，是Apache Axis）编程有一定程度的了解，并有一些使用它们的经验
- 假定你具有一些 UML 知识（它们是有用的，但是不必理解图）

排版约定

本书使用了以下排版约定：

斜体字（Italic）

表示新出现的名词、URL、电子邮件地址、文件名、文件扩展名、路径、目录以及Unix实用程序。

等宽字体（Constant Width）

表示命令、选项、分支、变量、类型、类、命名空间、方法、模块、属性、参数、值、对象、时间、时间处理程序和XML标记。

等宽斜体（*Constant Width Italic*）

显示应被用户提供的值替换的文本。

等宽黑体（**Constant Width Bold**）

在代码示例中强调重要文本内容。

使用代码示例

本书用于帮助你完成工作。一般而言，你可能在程序和文档中使用本书中的代码。你不必联系我们以获得许可，除非你正在复制代码的重要部分。例如，编写一个使用了本书中多块代码的程序不需要许可。销售或分发O'Reilly书籍中示例的CD-ROM确实需要许可。通过引用本书和引用示例代码来解答问题不需要许可。把本书中大量的示例代码纳入到你的产品文档中确实需要许可。

我们感谢但不要求注明出处。在注明出处时，通常要包括标题、作者、出版社和ISBN。例如："*AspectJ Cookbook* by Russ Miles, Copyright 2005 O'Reilly Media, Inc., 0-596-00654-3"。

如果你感到对代码示例的使用不在公平使用或者上面给出的许可之内，可通过*permissions@oreilly.com*自由地与我们联系。

我们很想听到你的建议

我们在O'Reilly尽最大努力确保本书内的秘诀是对作者最佳能力的测试和验证。不过，AspectJ是一种快速发展的技术，其支持工具集或语言本身的某些特性可能发生了变化，或者我们自己犯了错误。如果是这样，请通过以下地址把关于本书的评论和问题发送给出版社：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

本书的网页上列出了勘误表、示例和任何额外的信息。可登录以下网址查询：

http://www.oreilly.com/catalog/aspectjckbk
http://www.oreilly.com.cn/book.php?bn=7-302-14011-1

如果想要发表关于本书的评论和技术问题，请发邮件至：

bookquestions@oreilly.com
info@mail.oreilly.com.cn

关于图书、会议、资源中心和 O'Reilly 网络的更多信息，请查看我们的站点：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

无论如何，永远都不能把一本书的成果归于如此之少的几个人。本书是一个人编写的，但是得到了太多人的支持，我可以记住在这里提到的只是其中的一部分人，因此，如果漏掉了某个人，我预先表达我的歉意；我没有忘记你们。这只是由于我设法（但是未能）避免奥斯卡式的演讲。

我在这里有机会向那些使我集中注意力并支持我编写我的第一本书的人表示感谢。首先（并且是最重要的）必然是感谢我的家庭。感谢妈妈和爸爸，你们用无尽的笑容、温暖和理解见证了我通过最艰难的时刻。感谢 Bobs、Rich、Ad，甚至还要感谢 Aimee，你们总是让我保持做人的正道。你们全都帮助我完成这个项目，并让我保持脚踏实地，因此，我无论怎么感谢你们都是不够的，但是我将很开心地尝试这样做。

感谢我的编辑 Brett McLaughlin 和 O'Reilly 的所有员工，你们见证了我通过提案和编写的完整过程，并且感谢你们如此努力地工作，以使得本书出版。衷心感谢 Rob Romano，

你在项目初期容忍了我的“只有少量图形”的乐观情绪。真的，Bob，我确实认为只有少量的图形！

感谢AspectJ的真正英雄，即AspectJ社区。特别感谢Adrian Colyer、George Harley、Matthew Webster、Mik Kersten和Wes Isberg使AspectJ成为一种如此伟大的实现。感谢参与Ron Bodkin领导的Track项目的所有人提供如此多有趣和有用的演示文稿，以及一个令人印象如此深刻的测试台，它可用于本书中的秘诀中表达的许多概念，还要感谢Ramnivas Laddad的内行、友好的建议。

感谢我的团队中专注、耐心的校对人员。感谢Laura Paterson，在极早的时候，当时我还只是确实想编写本书，那时你就给我提供了如此多的支持。没有你的帮助和专业知识，我怀疑这个项目是否会通过提案阶段。还要感谢Andy（Tiger）Winter、Rob Wilson和Grant Tarrant-Fisher，你们接受了我对面向方面的狂热，并且没有让我完全发狂。总而言之，你们是我曾经共事过的最让人愉快的一群人。

感谢牛津大学软件工程计划中的所有人，你们为每个秘诀提供了它们所需的透彻评论；要特别感谢：Ganesh Sitamplan，负责语言输入；Peet，负责与初学者打交道；Jeremy Gibbons，使整个过程运转起来。

感谢我的朋友们，在我编写和修订本书的漫长的日子里，你们全都知道何时来看我，以及何时不要打扰我，我足够幸运地遇到了其中一些最好的人。我要向Jo Westcott、Sam和Martin、Jason以及Kerry表示巨大的感谢。还要特别感谢Jason询问（在我第一次告诉他我要编写一本书时）这本书是否会被制作成电影。（我确实不希望这样！）

最后，我想感谢Kim。你在过去的一年里对我意味着很多，并且你对本书做出了比其他任何人都多的贡献。尽管我竭力想完全毁坏你的语法技能，但你一直坚持帮助我，并成为我最好的朋友。单单这样的感谢永远都赶不上你带给我的快乐，在接下来的许多年里，我会一直盼望着偿还这笔人情债。

最后但并非最不重要的，感谢在编写本书时帮助过我的所有人。我不会忘记你们的帮助，并且我知道我欠你们所有人一两瓶啤酒。

第1章

面向方面概述

本章将简单地介绍一些有关面向方面的概念。这里所讨论的内容决不是面向方面概念的权威说明，但是它应该展示这些概念及相关术语的一种特质。当你实现本书余下内容中的秘诀时，这提供了一个将会证明是有用的参考。

面向方面简史

面向方面并不是一种全新的编写软件的方法。这与以下情形非常相似：当Java被软件团体所认知并采纳后，虚拟机系统就不再是一个全新的概念了，以前的很多开发工具和环境所支持的部分功能现在被划到了面向方面的范畴中。与Java一样，面向方面正被普遍采纳，并且成为实践那些几乎可以回溯到软件开发初期的旧思想的事实上的方法。

织入代码、pragma指令的开发环境和工具，甚至包括调试器，它们全都包含一些面向方面的方法之下的行为。但是，重要的区别是：方法背后的基本原理，以及该基本原理如何驱动技术和工具。面向方面自身并不代表任何一种具体的技术，不过它能够以新的、更加模块化的方式来实现这些技术过去在各自领域所引入的优势。

如前所述，这类书籍的主题不是关于面向方面的基本原理和概念基础的。如果想了解更多有关这项技术的基本知识，最好的方法就是在Google上搜索“面向方面的软件开发”。本书关注的是理解这项技术的实用方法；它介绍了通过利用面向方面软件开发的令人印象深刻的威力来完成工作。

AspectJ

公平地说，在Xerox Palo Alto研发中心（Xerox Palo Alto Research Center，PARC）成立之后，面向方面才算是真正步入了发展的正轨，它标志着计算机业的一次革命。

Xerox 的先期投资用于开发专用的面向方面语言，而后又转为用 Java 开发通用模型。AspectJ 随之应运而生，并且成为全书中所看到秘诀的核心开发工具。

在编写本书时，AspectJ 正快速成长为一种面向方面的开发工具，并且有丰富的示例可用。在 2002 年，Xerox PARC 做出了一个重要的决定，就是把 AspectJ 的开发转移到 eclipse.org Web 站点上的一个更开放的论坛中。AspectJ 的当前下载数字显示：人们对该方法的兴趣正呈指数级增长，并且软件开发团体也把面向方面视做软件开发中的一个极其重要的演进。所以，现在正是使用本书中现实的面向方面秘诀的最佳时机，以把这种新的、强大的工具添加到你的软件开发技能集中。

面向方面的定义

在开始研究实际的秘诀之前，值得简要介绍一下全书中使用的一些概念和术语。

横切关注点

面向方面编程的基本前提就是让开发人员能够在软件中表述模块化的横切关注点(cross-cutting concern)。这意味着什么？横切关注点是跨软件特定部分使用的一种行为，通常也是一种数据。它可能是一种约束，作为软件本身的一种特征，或者只是所有类都必须执行的一种行为。

横切关注点最常见的例子就是日志记录（在第 21 章中介绍），它几乎可以看做面向方面方法的“Hello World”示例。日志记录之所以可以看做横切关注点，是因为它会影响整个软件体系中的很多领域，而且会对业务逻辑产生影响。日志记录可能会被应用于很多类，而正是因为它具有日志记录方面的水平应用的形式，所以称之为横切。

方面

方面（aspect）是横切关注点的另一种称呼。在面向方面中，方面提供了一种机制，利用该机制，可以用一种模块化的方式指定横切关注点。为了充分利用方面的威力，我们需要了解一些基本概念，以便用一般的方式指定和应用方面。我们必须能够：

- 以模块化的方式定义方面
- 动态地应用方面
- 根据一组规则应用方面
- 提供一种机制和一种环境，用于指定将为特定方面执行的代码

面向方面方法提供了一组语义和语法构造来满足这些要求，使得无论编写的是哪一类软件，都可以一般地应用方面。这些构造就是通知（advice）、连接点（join point）和切入点（pointcut）。

通知

通知就是方面被调用时所执行的代码。通知包含自身的一组规则，这组规则规定了何时调用通知，这是与被触发的连接点相关的。

注意： 第13章中直接论及了针对不同形式通知的秘诀，并展示了AspectJ内可用的一些更高级的通知特性，如多个通知之间的优先级。

连接点

连接点就是可能会或者可能不会调用某个通知的应用程序内的特定点。具体可能有哪些连接点取决于使用的工具以及开发应用程序时所用的编程语言。下面是AspectJ中支持的连接点：

- 在调用方法时连接
- 在方法执行期间连接
- 在调用构造函数时连接
- 在构造函数执行期间连接
- 在方面通知执行期间连接
- 在对象初始化以前连接
- 在对象初始化期间连接
- 在静态初始化执行期间连接
- 在引用类的字段时连接
- 在给类的字段赋值时连接
- 在执行处理程序时连接

切入点

切入点是用于声明连接点中关注的AspectJ机制，用来发起一份通知。它们封装了决策逻辑，通过计算这个逻辑来决定在遇到某个连接点时是否调用一份特定的通知。

切入点是面向方面方法中的一个至关重要的概念，因为它提供了一个抽象的机制，用来指定所选连接点中的关注，而无需依赖于哪些连接点在特定应用程序中的细节。

注意： 第 4~12 章的秘诀中说明了如何定义和使用切入点。

结束语

图 1-1 显示了了连接点、方面、切入点、通知和应用程序类之间的关系。

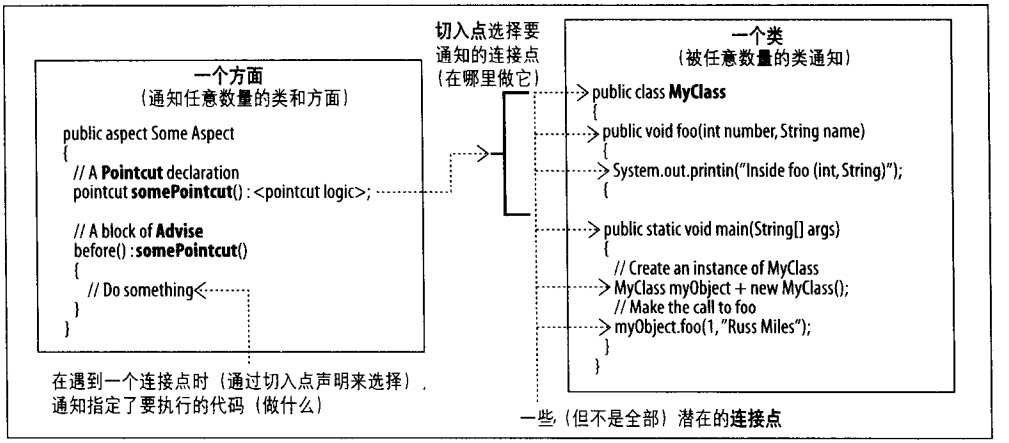

图 1-1：方面、切入点和通知之间的关系

在哪里找到更多信息

有关面向方面背后的概念和基本原理的更详细的信息，请查看以下 Web 站点：

http://www.parc.xerox.com/research/csl/projects/aspectj/default.html

Xerox PARC 发布的有关 AspectJ 项目的官方信息。

http://www.eclipse.org/aspectj

AspectJ 开发技术官方下载站点，带有指向支持工具的链接。

http://www.eclipse.org/ajdt

Eclipse 提供了一组 AspectJ 开发工具，可以使用列出的 Web 站点中内置的更新站点机制来下载它们。

http://sourceforge.net/projects/aspectj4jbuildr/

这个站点提供了 Borland JBuilder 的一个插件，允许它与 AspectJ 集成。

http://sourceforge.net/projects/aspectj4netbean/

这个 Web 站点提供了 NetBeans 的一个插件，用于支持 AspectJ 开发。

第2章

初识 AspectJ

2.0 简介

本章包含使用AspectJ开始开发自己的面向方面代码的秘诀。开始的任务是建立一个构建环境，并了解不同类型的构建和目标，本章提供了AspectJ开发基础的教程。

在理解了面向方面中的基本概念之后，本章首先介绍了使用AspectJ的开发环境。AspectJ的核心是其对织入方面到Java代码中的构建支持。这目前是使用编译器ajc来完成的，ajc可以与现有的Java编译器javac交互。在Java编译过程的约束内工作，确保面向方面的程序会产生可以被任何Java虚拟机（Java Virtual Machine，JVM）运行的标准Java类。ajc会简单地把各个方面自动映射到Java类上。

ajc工具用于从命令行或者在IDE或其他构建工具内在编译时织入方面，但是AspectJ中还支持另外一种织入方法，即加载时织入。顾名思义，加载时织入支持在Java类加载器把应用程序加载进JVM中时，把方面织入应用程序中。这是AspectJ的一个相当新的特性，本章中简要描述了它，同时指出：使用AspectJ实现加载时织入的当前方法随着这个工具的成熟在将来可能会发生变化。

一旦熟练使用ajc命令行编译器，就可以在Eclipse开发环境内安装AspectJ组件，并利用可供AspectJ开发人员使用的一些能力。这些能力包括：如何使用新项目创建向导通过AspectJ Nature建立一个Eclipse项目，以及用Eclipse来修改使用AspectJ构建配置文件实际地构建到单个项目里的方面。

最后，本章中介绍了AspectJ项目的内务处理任务，比如：从AspectJ代码生成文档以及使用Ant构建项目，它们示范了AspectJ开发工具有多丰富、多强大以及多么受支持。

2.1 安装 AspectJ

问题

想建立 AspectJ 构建环境。

解决方案

从最简单的方面讲，AspectJ 安装是一个 4 步骤的过程：

1. 从 *http://www.aspectj.org* 下载 AspectJ 工具。
2. 通过双击下载的 JAR 文件来安装它。这会在计算机上创建一个 *aspectj1.2* 目录。
3. 利用以下命令测试环境：

```
> ajc
AspectJ Compiler

       Usage: <options> <source file | @argfile>..

AspectJ-specific options:
       ... Listed compiler options


1 fail|abort
```

   不要担心这里的 `1 fail | abort` 消息；如果得到上面所示的输出，那么就已经成功地安装和定位了 AspectJ 工具，并且它们可供使用。

4. 把 *aspectj1.2/lib/aspectjrt.jar* 复制到你的 JRE 库目录中。这个目录的典型位置如下：对于 Windows，它是 *%JAVA_HOME%\jre\lib\ext*；对于 Mac OS X，它是 */Library/Java/Extensions*。

讨论

准备使用方面来开发程序的第一步是下载你的环境的合适支持工具。就AspectJ而言，这涉及下载AspectJ库和工具。可以通过访问*http://www.aspectj.org*并沿着指向*Downloads*的链接前进，来获得这些工具。

一旦下载了合适的AspectJ开发工具包，就可以通过双击下载的*.jar*文件来安装它们。在安装后，将看到一个名为 *aspectj1.2 的目录*（如果接受默认的安装选项）。

注意： AspectJ安装程序在安装过程的末尾详细说明了一些重要的指导。这些指导包括：建立你的搜索路径，使得可以从命令行使用AspectJ工具；以及把AspectJ Java库添加到Java类路径中。一定要小心遵循这些指导，以在建立AspectJ开发环境时，使事情尽可能地容易。

在*%MY_HOME_DIRECTORY%/aspectj1.2/bin*目录中，你将看到AspectJ带有3种工具：

`ajc` *(Windows上的ajc.bat)*

它是AspectJ编译器。

`ajbrowser` *(Windows上的ajbrowser.bat)*

它是可用于开发AspectJ项目的AspectJ工具测试环境。

`ajdoc` *(Windows上的ajdoc.bat)*

它是AspectJ文档工具。

注意： `ajc`和`ajdoc`是标准Java开发工具包中的`javac`和`javadoc`的AspectJ对等工具。事实上，这两个AspectJ工具都使用标准Java工具来完成它们的工作，并提供了增强功能来迎合额外的AspectJ语法和结构。

值得看一下AspectJ安装，特别是*%MY_HOME_DIRECTORY/aspectj1.2/doc/index.html*目录，其中提供了许多支持文档，包括：

%MY_HOME_DIRECTORY%/aspectj1.2/doc/progguide/

它是AspectJ编程指南，针对AspectJ的所有用户。

%MY_HOME_DIRECTORY%/aspectj1.2/doc/devguide/

它是AspectJ开发人员指南，针对想构建AspectJ并为之做贡献的任何人。

%MY_HOME_DIRECTORY%/aspectj1.2/doc/api/

它是AspectJ运行时API，附录中也总结了它。

%MY_HOME_DIRECTORY%/aspectj1.2/examples/

它是有用和有趣的AspectJ编码示例集合，可以帮助你起步。

你现在已经在自己的计算机上建立了环境，用于使用AspectJ开发面向方面的软件。

一个好主意是：在*http://www.eclipse.org/aspectj*上定期检查AspectJ Web站点，以了解有关工具集最新更新的信息。也可以在AspectJ用户邮件列表中注册自己，以获得关于工具更改的通知，以及关于如何在整个软件业中使用AspectJ工具的新闻。

注意： 使用从*http://www.aspectjcookbook.com*上得到的AspectJ，可以让你尽可能快且简单地安装和运行一组容易的安装程序。这些安装程序会安装和配置整个AspectJ构建环境，包括：AspectJ、Eclipse IDE以及用于Eclipse的AspectJ插件。虽然不能保证这些安装程序中包含的版本是最新的，可以从*www.eclipse.org*、*www.eclipse.org/aspectj*和*www.eclipse.org/ajdt*这些独立来源下载它们，但是它们确实提供了轻松安装程序包所需的一切。

参考

熟悉AspectJ的下一个阶段是：如2.2节中所示，开始编写第一个方面的代码；2.6节中详细地研究了ajdoc工具；*http://www.eclipse.org/aspectj*上的AspectJ开发环境指南提供了关于AspectJ工具支持的运行时选项和标志的更多详细信息；本章中其他各节说明了如何通过在Eclipse内编译更复杂的AspectJ项目以及使用Ant构建工具来执行进一步的任务。

2.2 开发一个简单的方面

问题

想编写一个简单的方面。

解决方案

首先编写你自己的业务逻辑类，如例2-1所示。

例2-1：简单的业务逻辑Java类

```
package com.oreilly.aspectjcookbook;

public class MyClass
{
    public void foo(int number, String name)
    {
        System.out.println("Inside foo (int, String)");
    }

    public static void main(String[] args)
    {
        // Create an instance of MyClass
        MyClass myObject = new MyClass();
        // Make the call to foo
        myObject.foo(1, "Russ Miles");
    }
}
```

定义一个将应用于这个类的方面。例2-2中的方面为AspectJ拙劣地模仿了传统的“Hello World”，其方法是：提供一个方面来捕获对MyClass类中的void foo(int, String)方法的所有调用。

例2-2：AspectJ中简单的HelloWorld方面

```
package com.oreilly.aspectjcookbook;

public aspect HelloWorld
{
   pointcut callPointcut() :
      call(void com.oreilly.aspectjcookbook.MyClass.foo(int, String));

   before() : callPointcut()
   {
      System.out.println(
         "Hello World");
      System.out.println(
         "In the advice attached to the call pointcut");
   }
}
```

将这个文件保存在与你的业务逻辑类（如*HelloWorld.aj*或*HelloWorld.java*）相同的目录中。运行ajc命令，编译这个简单的应用程序，并为方面和类产生字节码*.class*文件：

```
> ajc -classpath %MY_CLASSPATH% -d %MY_DESTINATION_DIRECTORY% com/
oreilly/aspectjcookbook/MyClass.java com/oreilly/aspectjcookbook/HelloWorld.java
```

如果得到下列消息，那么你将需要添加*aspectjrt.jar*到你的类路径中：

```
warning couldn't find aspectjrt.jar on classpath, checked:

error can't find type org.aspectj.lang.JoinPoint

1 error, 1 warning
```

为了只把*aspectjrt.jar*添加到类路径中，可输入以下命令来调用ajc编译器（使用;(而不是:）来隔开Windows上的路径中的成分）：

```
> ajc -classpath %MY_CLASSPATH%:%ASPECTJ_INSTALLATION_DIRRECTORY%/lib/aspectjrt.jar
-d %MY_DESTINATION_DIRECTORY% com/oreilly/aspectjcookbook/MyClass.java com/
oreilly/aspectjcookbook/HelloWorld.java
```

ajc编译器将产生两个*.class*文件：*MyClass.class*和*HelloWorld.class*。AspectJ 1.2会产生常规的Java字节码，它们可以运行在任何1.2 JVM及更高版本上，因此，现在可以使用正常的java命令来运行这个应用程序：

```
> java -classpath %MY_CLASSPATH% com.oreilly.aspectjcookbook.MyClass
Hello World
```

```
In the advice attached to the call point cut
Inside foo (int, String)
```

祝贺你！你现在已经使用 AspectJ 编译并运行了你的第一个面向方面的应用程序。

讨论

这个秘诀展示了你的方面的第一个示例，以及 AspectJ 如何扩展 Java 语言。首先，新语法可能看起来有点奇怪，并且本书的有用部分专用于研究可以把新的语言构造用于创建方面的方式。为了预先阐明其中一些语法，例 2-3 简要研究了这个秘诀指定的方面中的每一行。

例 2-3：新 AspectJ 语法的简单示例

```
 1 package com.oreilly.aspectjcookbook;
 2
 3 public aspect HelloWorld
 4 {
 5    pointcut callPointcut() :
 6       call(void com.oreilly.aspectjcookbook.MyClass.foo(int, String));
 7
 8    before() : callPointcut()
 9    {
10       System.out.println(
11          "Hello World");
12       System.out.println(
13          "In the advice attached to the call pointcut");
14    }
15 }
```

第 3 行声明这是一个方法。

第 5 行和第 6 行声明单一命名的切入点的逻辑。切入点逻辑指定了应用程序中的任何连接点，其中会捕获对 `void MyClass.foo(int, String)` 方法的调用。切入点被命名为 `callPointcut()`，使得可以在方面的作用域内的任意位置引用它。

第7~13行声明单一通知块。`before()` 通知只是简单地指出它将在任何被 `callPointcut()` 切入点匹配的连接点之前执行。当连接点被匹配时，通知只会简单地输出几条消息给系统，以告知你已经执行了通知。

这个秘诀提供了一种用于测试开发环境的良好机制，以确保在对开发工具执行任何定制之前，它们都会按预期的那样工作。

注意： 可以用 *.aj* 或 *.java* 扩展名来保存 AspectJ 方面。不管扩展名是什么，ajc 工具都会编译所提供的文件。不同的扩展名（*.aj* 和 *.java*）在很大程度上只关乎个人的喜好。

编译方面和Java类只会产生*.class*文件。这是AspectJ的一个非常重要的特性；方面自身会被视作对象。由于这种处理方式，所以可以把它们编码成类文件；这确保在运行应用程序时，Java运行时环境（Java Runtime Environment，JRE）不需要理解任何额外的特定于方面的文件格式。通过在JRE类路径中包含*aspectjrt.jar*支持库，可以把面向方面的软件应用程序部署到任何平台上的任何JRE中，以与Java的“编写一次，随处运行”的哲理保持一致。

参考

在使用这个秘诀之前，有必要获得AspectJ工具，并准备一个如2.1节中所介绍的简单的命令行构建环境；第4～12章中描述了切入点，并在4.1节中专门研究了`call(Signature)`切入点；9.1节中描述了`within(TypePattern)`切入点；12.4节中描述了用于切入点的NOT（!）运算符；在13.3节中可以看到通知的`before()`形式。

2.3 编译一个方面和多个Java文件

问题

想方便地编译所选的方面，它们将应用于多个Java类。

解决方案

创建一个名为*<appname>.lst*的AspectJ构建配置文件，其中包含要包括在编译中的所有类文件和方面的名称，类似于例2-4中的*.lst*示例文件。

例2-4：AspectJ构建配置.lst示例文件的内容

```
// File files.lst
com/oreilly/aspectjcookbook/MyClass.java
com/oreilly/aspectjcookbook/MyAspect.java
com/oreilly/aspectjcookbook/AnotherClass.java
com/oreilly/aspectjcookbook/AnotherAspect.java
```

使用以下命令指示`ajc`编译器把这些方面应用于类：

```
> ajc -argfile files.lst -classpath %MY_CLASSPATH% -d %MY_DESTINATION_DIRECTORY%
```

讨论

`ajc`工具完成方面和类的编译过程在极大程度上对于开发人员是透明的，并且可以被视做黑盒。你确实不应该担心AspectJ编译器内可能发生的中间步骤，除非想深入研究`ajc`自身的开发工作。

把输入看做构建过程是重要的；通过创建 *.lst* 构建配置文件来处理它。ajc 编译器不会搜索要编译的文件的源或类路径；必须告诉它编译中涉及哪些文件。这意味着将与方面一起编译的所有源都必须直接输入到 ajc 编译器中。有3种方式可用于把待编译的文件提供给 ajc 编译器（其中两种在语义上是等价的）：

-argfile *选项*

可以在 ajc 命令行上利用这个选项指定文件名，从而在一个 *.lst* 文件中提供所有的文件。

@ *选项*

这个选项等价于 -argfiles 选项。

直接列出文件

在调出 ajc 编译器时，可以在命令行上简单地指定文件。

参考

2.1 节显示了如何为使用 AspectJ 执行开发工作准备简单的命令行构建区域；可以从 *http://www.eclipse.org/aspectj* 下载 AspectJ 开发环境指南，它提供了 ajdoc 工具支持的运行时选项和标志的详细信息；*http://hugunin.net/papers/aosd-2004-cameraReady.pdf* 中提供了 AspectJ 编译器过程的完整描述；2.8 节显示了如何使用 *.lst* 文件，为特定的构建配置改变织入的方面。

2.4 织入方面到 jar 中

问题

想把方面织入到已经编译并收集进 *.jar* 文件的代码中。

解决方案

在运行 ajc 命令时，使用 -inpath 命令行选项。

讨论

ajc 命令把方面织入到 Java 字节码中，这些字节码可以驻留在 *.class* 文件中、Java *.jar* 库文件内，或者同时驻留在这两个文件中。下面的指导显示了如何获取2.2节中的代码，并在把 HelloWorld 方面织入到库中之前把 MyClass 类打包到一个 *.jar* 文件中：

1. 使用传统的javac命令编译MyClass类：

```
> javac -classpath %MY_CLASSPATH% -d %MY_DESTINATION_DIRECTORY% com/
oreilly/aspectjcookbook/MyClass.java
```

2. 把生成的*MyClass.class*文件打包到一个名为*MyApp.jar*的*.jar*文件中：

```
> jar -cvf MyApp.jar com/oreilly/aspectjcookbook/MyClass.class
```

3. 使用ajc命令编译*HelloWorld.java*方面，使用-inpath选项在命令行上指定新的*MyApp.jar*：

```
> ajc -classpath %MY_CLASSPATH% -d %MY_DESTINATION_DIRECTORY% -inpath MyApp.jar
com/oreilly/aspectjcookbook/HelloWorld.java
```

-inpath选项强制ajc编译器从提供的*.jar*文件中把Java字节码提取到-d选项所指定的目标目录中。然后，ajc编译器将把提取的字节码包含在方面织入过程中。

4. 如果在利用ajc进行编译期间没有发生错误，那么将成功地织入*MyApp.jar*文件内包含的类以及HelloWorld方面。由于ajc命令从提供给-inpath选项的*.jar*文件中提取类，所以运行应用程序不再需要它们。不过，可以在运行ajc命令时，使用-outjar选项有选择地在其自身的*.jar*文件中重新打包新的应用程序：

```
> ajc -classpath %MY_CLASSPATH% -d %MY_DESTINATION_DIRECTORY% -inpath MyApp.jar
-outjar MyAspectOrientedApp.jar com/oreilly/aspectjcookbook/HelloWorld.java
```

这会产生一个MyAspectOrientedApp.jar，它包含应用程序的方面和类，之后可以使用传统的java命令来运行它们：

```
> java -classpath MyAspectOrientedApp.jar com.oreilly.aspectjcookbook.MyClass
```

警告： 在把方面织入到第三方提供的*.jar*库中之前，确保用于保护那个库的许可允许你更改内容。如果你没有检查是否可以更改*.jar*文件的内容，那么就可能侵犯第三方的许可协议。例如，保护Java标准库的许可通常不支持把方面织入到驻留在java或javax包或者其子包的代码中。

参考

建立环境，以从2.1节介绍的命令行中编译AspectJ项目。

2.5 在加载时织入方面

问题

你想推迟到正在把特定的应用程序加载进Java虚拟机中时，才决定是否应该把某个方面应用于该应用程序。

解决方案

使用 AspectJ 1.2 新增的加载时织入特性。

讨论

使用 2.2 节中所示的代码，下列指导说明了如何在加载时把 HelloWorld 方面应用于 MyClass 类：

1. 在使用传统的 javac 命令时编译 MyClass 类：

   ```
   > javac -classpath %MY_CLASSPATH% -d %MY_DESTINATION_DIRECTORY% com/
   oreilly/aspectjcookbook/MyClass.java
   ```

   通过使用 javac 命令，完全避免了在编译时把任何方面织入到应用程序中。如果想在编译时包括一些方面，则可使用 ajc 命令，并列出在那个时刻要包括在 AspectJ 构建配置文件中的方面，而不要指定你只打算在加载时织入的任何方面。

2. 现在可以通过使用 java 命令运行应用程序来执行检查，以确保不带任何方面地编译了 MyClass 类：

   ```
   > java com.oreilly.aspectjcookbook.MyClass
   Inside foo (int, String)
   ```

3. 使用 ajc 命令把 HelloWorld 方面编译进方面库 *.jar* 文件中：

   ```
   > ajc -outjar helloworldlibrary.jar -d %MY_DESTINATION_DIRECTORY% com/
   oreilly/aspectjcookbook/HelloWorld.java
   ```

   你可能发现自己得到了以下警告，指出不存在与 HelloWorld 方面中声明的切入点逻辑的匹配：

   ```
   warning no match for this type name: com$oreilly$aspectjcookbook$MyClass
   [Xlint:invalidAbsoluteTypeName]
   call(void com.oreilly.aspectjcookbook.MyClass.foo(int, String));
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   ```

   此时，没有把方面织入到 MyClass 类中，因此，预期会出现该警告，并且在这里可以忽略它。

4. 你现在准备好运行独立编译的类和方面，并在加载时一起织入它们。AspectJ 1.2 中提供的一个脚本有助于运行加载时织入器。这个脚本目前位于 *%MY_HOME_DIRECTORY%/aspectj1.2/doc/examples/ltw* 目录中，并且被称为 *aj* 或 *aj.bat*。正如 ajc 命令是与 java 命令等价的 AspectJ 命令一样，从而可以把 aj 脚本看作与 java 命令等价的 AspectJ 命令的基础。

注意： 在AspectJ 1.2中，aj脚本位于作为AspectJ文档一部分的示例目录中。在AspectJ的将来版本中，随着加载时织入功能的成熟，aj脚本很可能在*%ASPECTJ_INSTALL_DIRECTORY%/bin*目录中加入更主流的脚本。

5. aj命令依赖于设置3个环境变量，使得它可以为应用程序选择相关的方面库和Java类。需要把以下变量设置到合适的项目和AspectJ安装位置：

   CLASSPATH

   通常把项目的类和*aspectjrt.jar*的位置安装到*%MY_HOME_DIRECTORY%/aspectj1.2/lib*中。

   ASPECTPATH

   指向包含要在加载时织入的方面的任何*.jar*文件。在这个示例中，这个变量应该指向*helloworldlibrary.jar*文件。

   ASPECTJ_HOME

   包含AspectJ安装的位置。

6. 现在可以使用aj命令来运行应用程序。*helloworldlibrary.jar*中包含的方面会在加载时织入到MyClass类中：

```
> %ASPECTJ_INSTALL_DIRECTORY%/doc/examples/ltw/aj com.oreilly.aspectjcookbook.
MyClass
Hello World
In the advice attached to the call point cut
Inside foo (int, String)
```

除了AspectJ 1.2提供的加载时织入的默认实现之外，还可以通过扩展和使用org.aspectj.weaver.tools包中提供的类来创建自己的实现。这是一个针对那些使用AspectJ工具的开发人员的高级主题，这里没有介绍它。

注意： 这个秘诀说明了如何利用AspectJ 1.2中的当前实现，来处理加载时织入。尽管方面的加载时织入是工具集的一个重要补充，但它是AspectJ中一个相当新的特性，并且随着它逐渐成熟，当前实现很可能发生变化。

参考

2.1节中介绍了建立环境，以从命令行编译AspectJ项目；2.3节中更详细地展示了AspectJ构建配置文件；Adrian Colyer在*http://www.jroller.com/comments/colyer/Weblog/load_time_weaving_with_aspectj*上提供了加载时织入的另一个示例；AspectJ 1.2中的

加载织入功能的详尽解释是作为AspectJ文档的一部分提供的，并且安置在*%ASPECTJ_INSTALL_DIRECTORY%/doc/README-12.html*中。

2.6 生成 Javadoc 文档

问题

想为自己的 AspectJ 应用程序生成 javadoc 格式的文档。

解决方案

从命令行或者 Eclipse 内使用 ajdoc 工具。

讨论

自从 Java 1.1 起，Java 开发人员就使用 javadoc 工具为他们的 Java 应用程序生成全面的文档。javadoc 工具从未被设计成处理 AspectJ 引入的新语言构造，并且不能为 AspectJ 应用程序生成文档。为了应对这个问题，AspectJ 的开发人员创建了 ajdoc，这种工具扩展了 javadoc，使得它可以正确和有用地用文档记录应用程序中面向方面的构造。

倘若如2.1节中所述，正确建立了环境，就可以通过输出以下命令从命令行访问 ajdoc 工具：

```
> ajdoc -sourcepath <The root location of you code> -d <The directory in
which you want your documentation to be placed> <The file locations of each
of the classes aspects and classes to include in the generated documentation>
```

ajdoc 工具提供了 -argfile 参数，使得你可以提供一个显式的 AspectJ 构建配置 *.lst* 文件，如2.3节中所提到的那样。*.lst* 文件包含用于应该为其产生文档的应用程序中的一组类和方面的构建配置。

使用 ajdoc 意味着现在可以如例 2-5 中所示的那样，用文档记录切入点、通知和方面。

例 2-5：将 javadoc 标签应用于方面、切入点和通知

```
package com.oreilly.aspectjcookbook;

/**
 * A simple aspect to weave in a HelloWorld message when foo(int,name)
 * is called on objects of the MyClass class.
 * @author russellmiles
```

```
 * @version 1.0
 *
 */
public aspect HelloWorld
{
   /**
    * Selects join points on calls to the MyClass.foo(int,String) method.
    */
   pointcut callPointCut() :
      call(void com.oreilly.aspectjcookbook.MyClass.foo(int, String));

   /**
    * Simple before advice that prints HelloWorld to the standard output.
    */
   before() : callPointCut()
   {
      System.out.println(
      "Hello World");
      System.out.println(
      "In the advice attached to the call point cut");
   }
}
```

例2-5是2.2节中所示的相同方面，但是突出显示了为ajdoc工具提供的文档。要为这个简单的示例从命令行产生文档，将从与源代码相同的目录中输出以下命令，以确保正确建立了AspectJ构建环境（同样如2.2节中所示），并且创建了*docs*目录，以准备接受生成的文档：

```
> ajdoc -sourcepath . -d docs com/oreilly/aspectjcookbook/MyClass.java com/
oreilly/aspectjcookbook/HelloWorld.java
```

如果ajdoc工具正确地运行，就应该看到如下所示的一段长长的内容，但它是前一个命令的成功输出：

```
> Calling ajc...
> Building signature files...
> Calling javadoc...
Loading source file /source/Chapter 2/2.5/ajdocworkingdir/com/oreilly/
aspectjcookbook/MyClass.java...
Loading source file /source/Chapter 2/2.5/ajdocworkingdir/com/oreilly/
aspectjcookbook/HelloWorld.java...
Constructing Javadoc information...
Standard Doclet version 1.4.2_03
Generating constant-values.html...
Building tree for all the packages and classes...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
```

```
Generating allclasses-noframe.html...
Generating index.html...
Generating packages.html...
Generating com/oreilly/aspectjcookbook/package-frame.html...
Generating com/oreilly/aspectjcookbook/package-summary.html...
Generating com/oreilly/aspectjcookbook/package-tree.html...
Generating com/oreilly/aspectjcookbook/HelloWorld.html...
Generating com/oreilly/aspectjcookbook/MyClass.html...
Generating package-list...
Generating help-doc.html...
Generating stylesheet.css...
> Decorating html files...
> Decorating /source/Chapter 2/2.5/com/oreilly/aspectjcookbook/MyClass.html...
> Decorating /source/Chapter 2/2.5/com/oreilly/aspectjcookbook/HelloWorld.html...
> Removing generated tags (this may take a while)...
> Finished.
```

在使用AspectJ开发工具(AspectJ Development Tools, AJDT)插件时，也可以从Eclipse IDE内部使用ajdoc工具。为了使用ajdoc工具，可以在Package Explorer面板中选择AspectJ项目，并单击Project → Generate Javadoc...。

当Eclipse显示Generate Javadoc对话框时，将需要把Javadoc Command更改成ajdoc。为了执行该任务，可单击Configure按钮，然后导航到安装ajdoc工具的位置；通常，这是*%HOME DIRECTORY%/aspectj1.2/bin*。选择ajdoc工具，并单击OK。然后，应该把Javadoc Command更改成指向ajdoc工具，以指示它已经被配置成在生成项目文档时要使用的工具。由于ajdoc工具的工作方式与javadoc工具几乎相同，所以Eclipse不需要知道更多的信息即可使用新的配置，并且你可以像以往那样为javadoc生成设置选项。

注意： 从Eclipse 3和AJDT 1.1.11起，已经简化了使用ajdoc生成文档的过程。以前，你不得不手动把javadoc生成设置到ajdoc工具；随着这些版本的发布，ajdoc文档具有它自己的菜单项，可以通过单击Project → Generate ajdoc... 来访问它。

参考

David Flanagan所著的Java in a Nutshell (O'Reilly)；Steve Holzner所著的Eclipse和Eclipse Cookbook (O'Reilly)；可从*http://www.eclipse.org/aspectj*下载的AspectJ开发环境指南提供了关于ajdoc工具支持的运行时选项和标志的更多详细信息；2.3节中讨论了AspectJ构建配置文件。

2.7 使用 Eclipse 编译一个 AspectJ 项目

问题

你想使用 Eclipse 编译 AspectJ 项目。

> **不使用 Eclipse 或 Ant 吗？**
>
> 如果在利用 AspectJ 编写面向方面的软件时，你不打算使用 Eclipse 或 Ant 工具，那么值得向前跳到下一章，开始查看部署 AspectJ 应用程序的公共方法。

解决方案

下载并安装 AspectJ 开发工具（AJDT）插件到 Eclipse 中。

讨论

AspectJ运行在与Eclipse项目相同的开源集体之下，并且为IDE提供了最高级的AspectJ插件。可以通过遵循 *http://www.eclipse.org/ajdt* 上提供的指导，来下载 AspectJ Eclipse 插件。

注意： 一个良好的经验法则是：下载AspectJ的最新版本、对应的AJDT最新版本，然后下载Eclipse的支持版本，以防 Eclipse 工具是 AspectJ 开发中所涉及的一个步骤，或者如此超前于它。如果你被这些可用的选项弄糊涂了，在 AJDT 下载页面上提供了一份兼容性表，它会把你指引到正确的方向上。

为了检查是否正确安装了插件，可单击 Eclipse 菜单栏上的 File → New → Project...。当出现 New Project 对话框时，AspectJ 项目选项应该可用，如图 2-1 所示。

注意： 默认情况下，Eclipse 3 早期的里程碑构建隐藏了所有额外的向导。为了看到 New AspectJ Project 选项，可能不得不选中 Show All Wizards 复选框。

突出显示 AspectJ Project 选项，在 New Project 对话框中单击 Next，并完成接下来的几个步骤，以建立新的 AspectJ 项目。

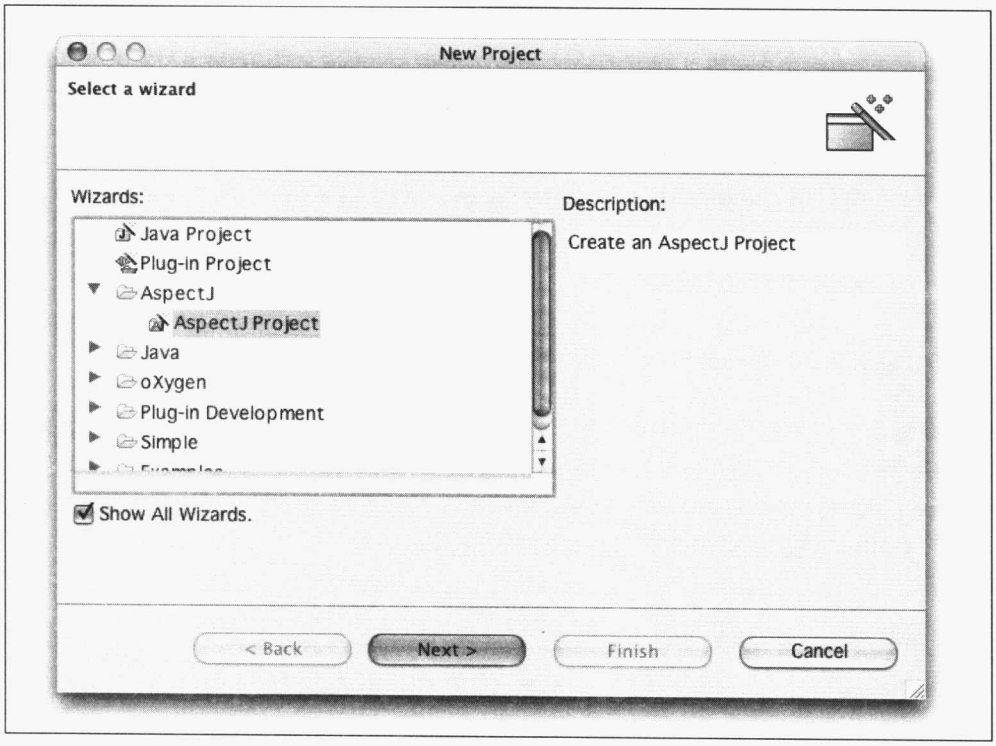

图 2-1：创建新的 AspectJ 项目

一旦完成了项目向导，可能会要求你切换到 Java 视图。如果这是你第一次使用该向导，则还可能要求你设置一些 AJDT 参数。切换到 Java 视图并为项目参数设置接受默认的值是一个好主意。一旦 Eclipse 切换到 Java 视图，项目看起来应该如图 2-2 所示。

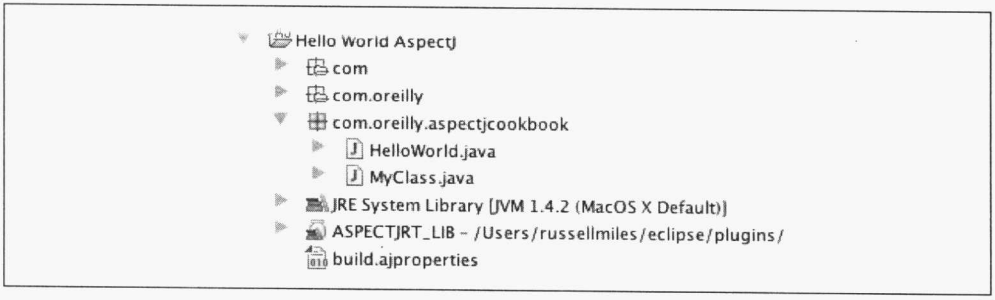

图 2-2：Eclipse Java 视图内的 AspectJ 项目源和 jars

注意： 从 AJDT 1.1.7 起，默认情况下，AspectJ 项目将增量式地编译对方面和 Java 类所做的所有更改。如果构建性能变得过于重要，则可在 Eclipse AspectJ 项目配置内关闭这个选项。

应该在创建新项目时自动编译新的 AspectJ 应用程序。如果你确实担心这不会发生，可单击图 2-3 中所示的 Build AspectJ Project 按钮。这将强制确实会重建项目。

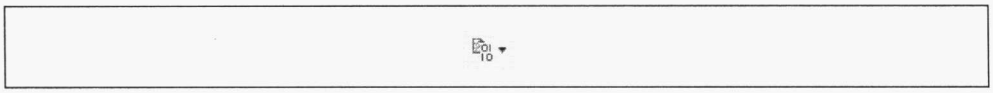

图 2-3：使用 Build AspectJ Project 按钮来检查项目已经成功地编译

现在就编译了应用程序，并且准备好运行，但是在运行它之前，注意到 AJDT 带给 Java 视图的增强功能是有趣的。也许最明显的增强是：当选择一个方面或被通知的类时，Outline 视图的内容中将包含新的区域，指示在什么地方应用了通知。图 2-4 和图 2-5 中显示了 Outline 视图中可用的额外信息的一个示例。

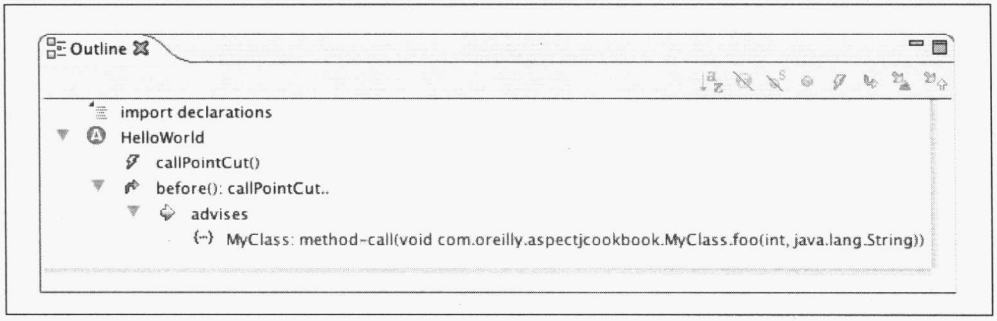

图 2-4：使用 Outline 视图来查看将特定的通知应用于哪些类或方面

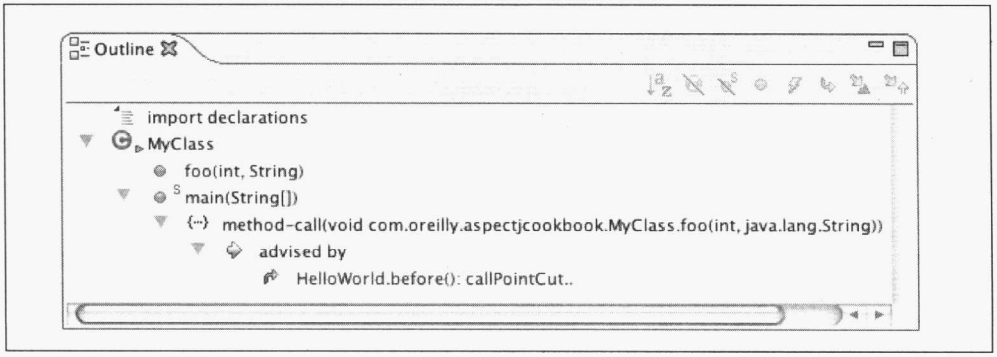

图 2-5：使用 Outline 视图来查看将什么通知应用于特定的类

AJDT还提供了项目的另一个视图，它以图形方式显示了如何将方面应用于应用程序的概要信息。可以通过选择 Window → Open Perspective → Other...，然后启用 Aspect Visualization视图，来打开Aspect Visualization视图。然后，项目的新视图将显示如何将方面应用于类的图形描述，如图 2-6 所示。

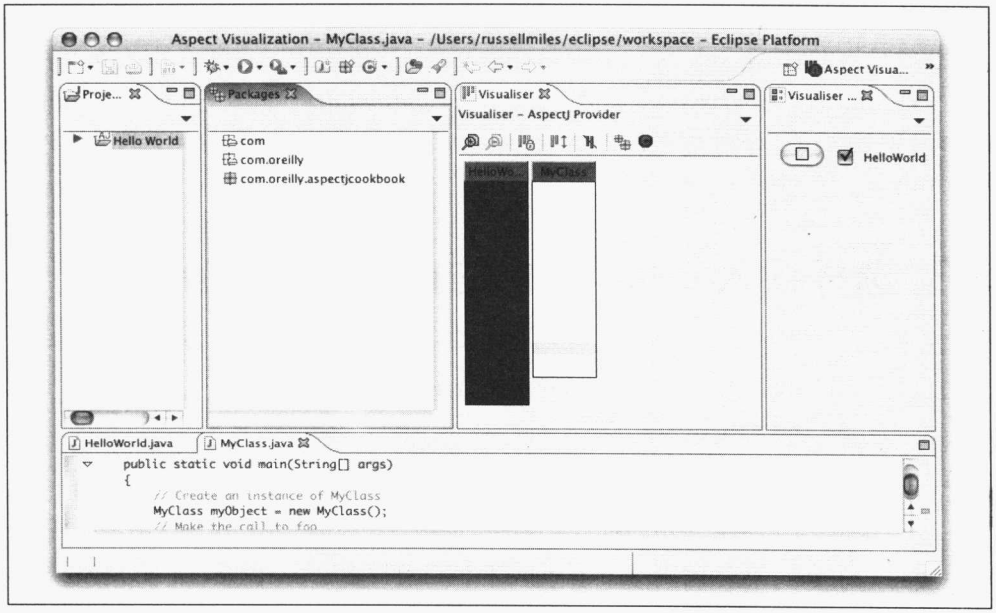

图 2-6：带有所选示例项目的 AJDT Aspect Visualization 视图

切换回Java视图，现在就可以运行应用程序。在Eclipse中，可以像运行传统的Java应用程序那样容易地运行AspectJ应用程序。在Java视图中，单击包含`public static void main(String[])`方法的类，并从主菜单中选择Run→Run As→Java Application。Eclipse随后将在所选类中搜索并运行`public static void main(String[])`方法，产生应用程序命令行输出，并将其输出到Console视图中，如图 2-7 所示。

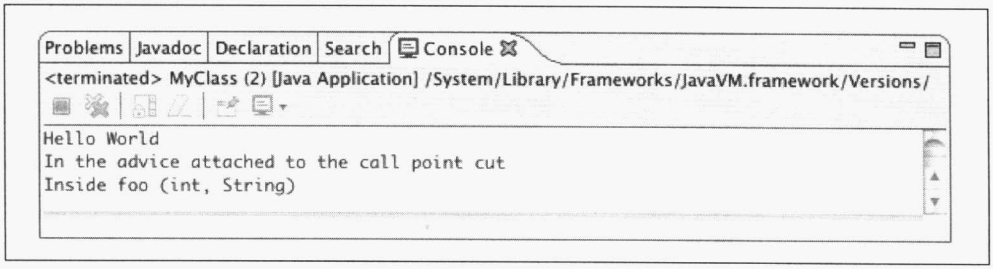

图 2-7：运行 5.1 节中创建的应用程序时 Console 视图上的输出

AJDT支持的特性和增强足以编写一章的内容。花一些时间了解可用的不同参数和特性，在你学习本书中的其他秘诀时，也许它们可以帮助你熟悉对你最有用的特性。

参考

2.9节说明了如何使用Ant工具来自动执行构建过程；可以通过访问*http://www.eclipse.org/ajdt*并沿着User Resources:Documentation链接前进，来访问AJDT插件的完整的用户指南；可以从*http://sourceforge.net/projects/aspectj4jbuildr/*下载用于Borland JBuilder的AspectJ插件；可以从*http://sourceforge.net/projects/aspectj4netbeans/*下载用于Sun's NetBeans的AspectJ插件；Steve Holzner所著的Eclipse和Eclipse Cookbook（O'Reilly）。

2.8 选择织入Eclipse内的构建中的方面

问题

在Eclipse中开发应用程序时，你想改变织入应用程序中的方面。

解决方案

为你所选的想织入应用程序中的每个不同的方面创建单独的AspectJ构建配置。然后，Eclipse允许你选择它将用于构建AspectJ项目的当前构建配置，因此，你可以轻松选择想应用于特定构建的方面集。

讨论

2.3节展示了AspectJ编译器可以使用构建配置*.lst*文件，来选择它将包括在其方面织入中的类和方面。默认情况下，Eclipse中的AspectJ项目具有单独一个*.lst*文件，自然称之为*default.lst*，它列出了项目中的所有文件。使用这个默认的文件，AspectJ编译器将把所有方面应用于通过这些方面中的切入点逻辑来指示的类。

对于许多应用程序，这种默认的行为是良好的，但是许多时候你可能想依赖于诸如部署目标或特性选择之类的事情，来改变应用于特定应用程序的方面。AspectJ提供了一种灵巧的方式，使用一个自定义的构建配置*.lst*文件（它包含或不包含你希望应用的方面），来改变要应用的方面选择。

Eclipse提供了一些手段，可以通过它们来创建自定义的AspectJ构建配置，并将其用于特定的项目。

当运行 2.7 节中创建的项目时，在 Eclipse 中的 Console 上产生的输出如下：

```
MyAspect before() advice
In the advice attached to the call point cut
Inside foo (int, String)
```

下面的步骤将创建一个新的构建配置，它将从该构建排除项目中的 MyAspect 方面，从而更改应用程序的行为，如 Console 上修正的输出所示。

从 Eclipse 3.0 起，通过安装 AJDT 版本 1.1.11 或更高版本，在创建新的 AspectJ 构建配置文件时会更改以下步骤。Eclipse AJDT插件现在还使用了一种新格式的构建配置文件，即 *.ajproperties* 文件。

可以通过单击 Project → Active Build Configuration → Save as...，为构建配置输入一个新名称，然后遵循第 8 步以后的步骤，在 Eclipse 3.0 中创建一个新的构建配置。这将创建一个新的 *.ajproperties* 文件，它与更传统的 *.lst* 文件相对。

也可以通过右击 *.lst* 文件并选择 Save as *.ajproperties* file，把现有的 *.lst* 文件转换成更新的 *.ajproperties* 格式。如果需要转换回 *.lst* 格式，可以右击 *.ajproperties* 文件，并选择 Save as *.lst* file。

1. 确保在 Package Explorer 视图中选择 AspectJ 项目。
2. 单击 File → New → Other...。
3. 在 New 对话框中，选择 AspectJ Build Configuration File 并单击 Next，如图 2-8 所示。
4. 出现 New AspectJ Build Configuration File 对话框，如图 2-9 所示。
5. 检查选择了正确的项目，并为新的构建配置文件输入一个名称，对于这个示例，可使用 *excludeAspects.lst* 这个名称。
6. 取消选中 Include all source files from project 复选框，并单击 Finish。
7. 将把新的 *excludeAspects.lst* 文件添加到项目中，并在文件编辑器中打开它。
8. 选择你想包括在这个构建配置中的文件，如图 2-10 所示。在这个示例中，你只想构建类并把所有的方面都排除在外，因此只需选择 *MyClass.java* 文件。
9. 将所做的更改保存到构建配置文件中。
10. 需要告诉 AspectJ 编译器切换到用于这个项目的配置。找到 Eclipse 工具栏上的 Build AspectJ Project/Select Configuration 按钮，如图 2-11 所示。

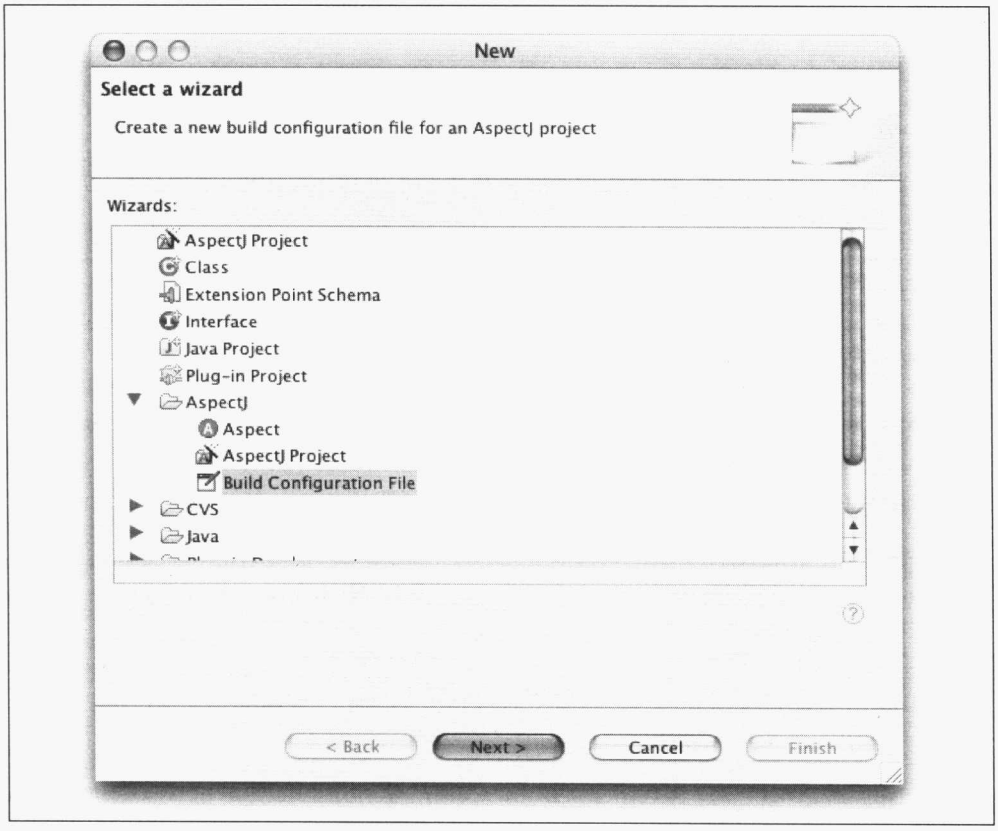

图 2-8：选择新的 AspectJ 构建配置文件向导

11. 单击该按钮右边较小的下三角箭头，弹出可用于这个项目的构建配置的列表，如图 2-12 所示。

12. 选择新的构建配置，AspectJ 编译器将依据新的设置自动重建项目。

注意： 在带有 AJDT 1.1.11 或更高版本的 Eclipse 3.0 中，可以从当前活动的配置中快速排除或包含单个源文件；然后，可以右击该源文件，并单击 Exclude from "%CURRENT_CONFIGURATION%"，或者如果文件已经被排除，则可单击 Include from "%CURRENT_CONFIGURATION%"。

在新的 `excludeAspects.lst` 配置之下构建项目后，当再次运行应用程序时，Console 输出将显示以下内容：

```
Inside foo (int, String)
```

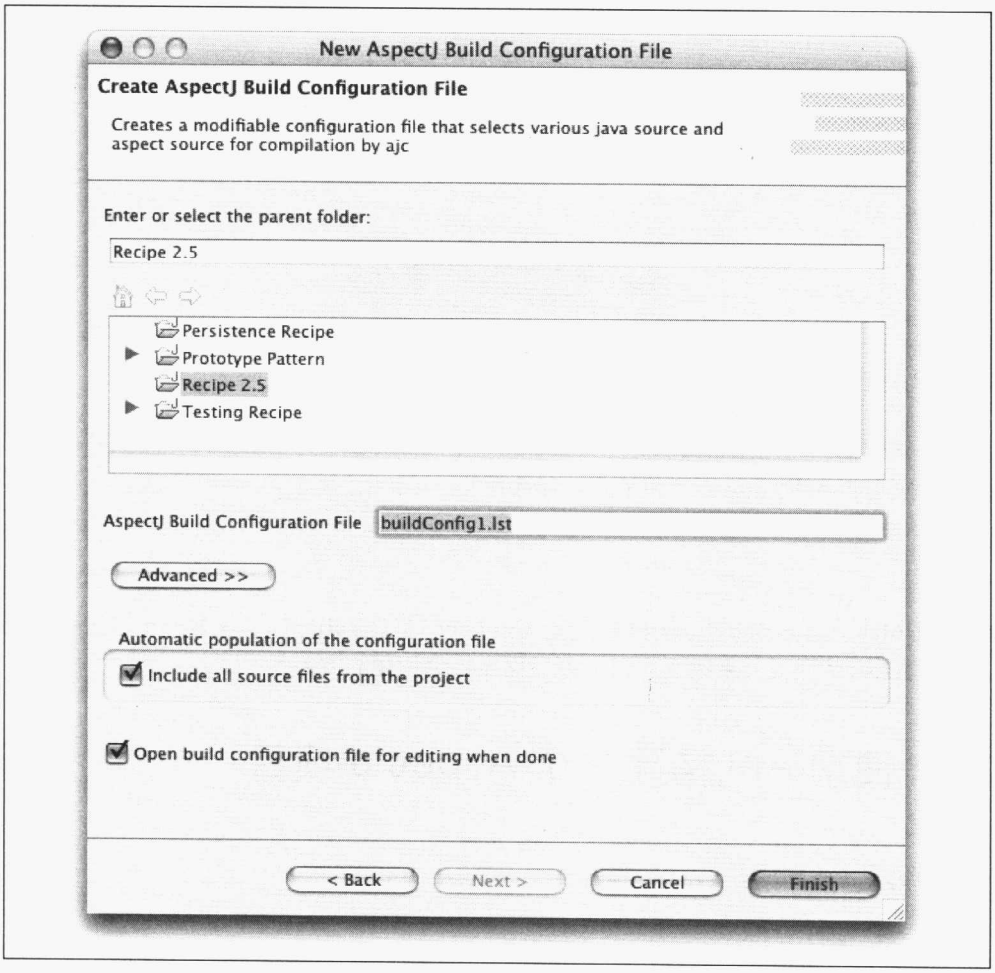

图 2-9：输入新的 AspectJ 构建配置文件的详细信息

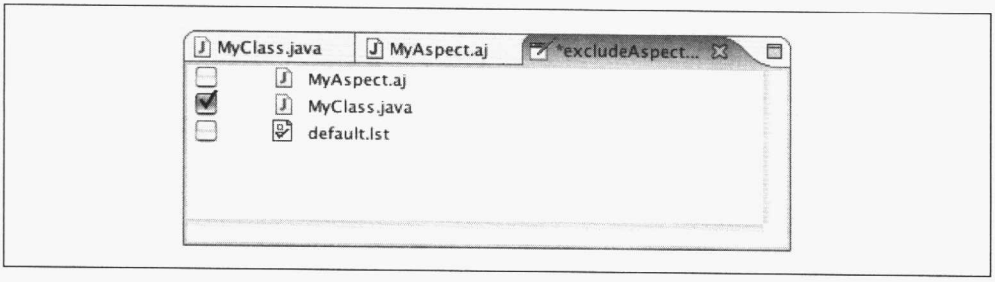

图 2-10：在 Eclipse 中编辑 AspectJ 构建配置文件

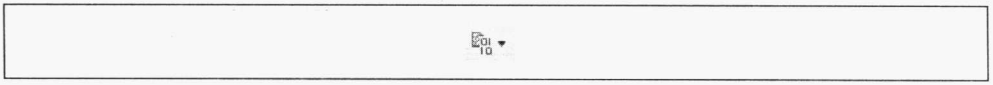

图 2-11：Eclipse 中的 Build AspectJ Project/Select Configuration 按钮

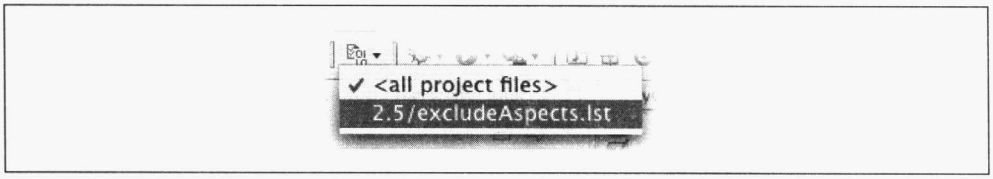

图 2-12：选择新的 excludeAspects.lst AspectJ 构建配置

当利用这个配置构建项目时，就会从应用程序中删除方面。为了切换回完全默认的构建配置，可单击 Selecting 下三角箭头，并选择 `<all project files>` 选项。

参考

Steve Holzner 所著的 Eclipse 和 Eclipse Cookbook（O'Reilly）；2.9 节中显示了如何使用 Ant AspectJ 任务，来改变应用于应用程序特定构建的方面。

2.9 使用 Ant 构建一个 AspectJ 项目

问题

你想使用 Ant 编译一个 AspectJ 项目。

解决方案

利用 AspectJ 工具包中包含的任务，使用 Ant 构建项目。

讨论

例2-6中的*Ant build.xml*配置文件显示了一个示例，其中示范了如何调用额外的AspectJ Ant 任务。

例 2-6：使用 AspectJ 任务的 Ant 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="compile" name="test">
    <property name="src" value="src"/>
    <property name="build" value="build"/>
```

```
    <taskdef resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.properties">
        <classpath>
            <pathelement location="%ASPECTJ_INSTALLATION%/lib/aspectjtools.jar"/>
        </classpath>
    </taskdef>
    <target name="compile">
        <mkdir dir="${build}"/>
        <iajc destdir="${build}" sourceroots="${src}">
            <classpath>
                <pathelement location="%ASPECTJ_INSTALLATION%/lib/aspectjrt.jar"/>
            </classpath>
        </iajc>
    </target>
</project>
```

下面是例 2-6 所做的工作：

1. 使用 AspectJ 任务属性定义一个新任务
2. 指定 *aspectjtools.jar* 的位置
3. 声明一个构建目标，它使用 iajc 任务来编译项目，这个任务反过来又依赖于 *aspectjrt.jar* 来执行

参见

Jesse Tilly 和 Eric M. Burke 所著的 Ant: The Definitive Guide (O'Reilly)；*http://jakarta.apache.org/ant/manual/index.html* 上的 Jakarta Ant 在线手册。

第3章

部署AspectJ应用程序

3.0 简介

无论用AspectJ开发什么应用程序，通常都希望把应用程序部署到目标环境，从而使得它可供用户使用。Java应用程序部署所涵盖的范围可以复杂到提供特定于环境的运行时包装器和脚本，以运行应用程序；也可以简单到提供可双击的*.jar*或可执行文件。因此，部署通常可能是真正让开发人员头痛的事。

为了添加这种混合的功能，AspectJ添加了一些额外的要求到应用程序部署中。本章中的秘诀详细描述了AspectJ应用程序强加给传统Java应用程序的那些额外的要求，以及在部署过程中可任由你支配的一些工具支持。

AspectJ绝不仅限于普通的Java应用程序。可以把AspectJ应用程序部署到许多不同的目标运行时环境中，包括Tomcat和Axis Web服务内的Java Servlet和Java Server Page（Java服务器页面）。这些更复杂的部署环境通常提供了为部署同一个应用程序提供了多种不同的方式。其中一些选项与另外一些选项相比更好地适合于AspectJ，因此本章重点描述把应用程序部署到这些目标环境的最容易的路径。

在学完本章中的秘诀之后，值得记住：可以使用Apache Ant（甚至命令行脚本和批处理文件）来自动执行许多手动步骤。不过，这里的秘诀会慎重地引导你通过部署到多种不同的目标环境中所需的所有手动步骤，以使你随时与将要发生的事情保持联系。

3.1 部署命令行AspectJ应用程序

问题

你想部署一个要部署的简单的AspectJ应用程序，然后从命令行运行它。

解决方案

在可以运行 Java 应用程序之前，直观的 Java 应用程序通常只需要目标计算机上的 Java 运行时环境（Java Runtime Environment），以及需要把应用程序中的类添加到 Java 类路径中。

AspectJ 需要 Java 运行时环境版本 1.1 或更高版本，并且它需要把额外的 *aspectjrt.jar* 库添加到类路径中，以支持 AspectJ 应用程序的面向方面的特性。

讨论

使用 2.2 节中开发的简单应用程序，下面的步骤会创建一个目录，其中包含运行 AspectJ 应用程序所需的所有必要的部署文件：

1. 创建一个包含运行时部署的新目录，将其与源目录分隔开。对于这个示例，将该目录命名为 *deployment*。
2. 创建顶级 deployment 目录的一个子目录，并将其命名为 *classes*。
3. 把应用程序的所有编译过的 *.class* 文件放在 *classes* 目录中，小心维持程序包目录结构。对于这个示例，将使用 2.2 节中的代码，因此，目录结构应该如图 3-1 所示。

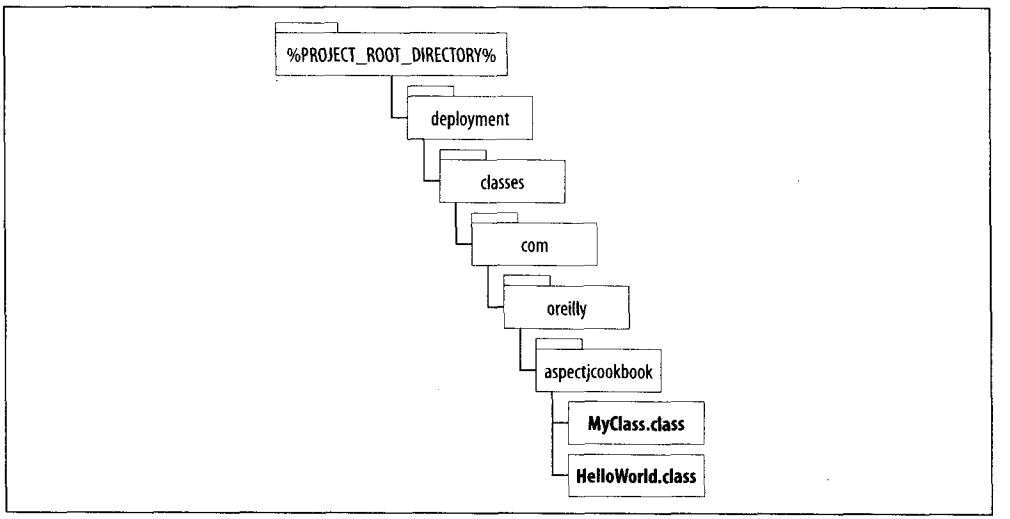

图 3-1：把 .class 文件放在部署区域之后的目录结构

可以通过手动复制 *.class* 文件的每个包，或者通过用 `ajc` 工具构建应用程序，通过目标标志 `-d` 设置到 *classes* 目录，来执行该任务：

```
> ajc -classpath %MY_CLASSPATH% -d %PROJECT_ROOT_DIRECTORY%/deployment/classes
com/oreilly/aspectjcookbook/MyClass.java com/oreilly/aspectjcookbook/HelloWorld.
java
```

如果得到一个错误，指出找不到*aspectjrt.jar*，则可检查2.2节中关于如何使用`ajc`命令的指导，其中解释了如何克服这个问题。

4. 在顶级*deployment*目录下创建一个名为lib的子目录。
5. 把*%ASPECTJ_INSTALLATION_DIRECTORY%/lib/aspectjrt.jar*文件复制到新的*lib*目录中。
6. 最终的部署设置应该如图3-2所示。

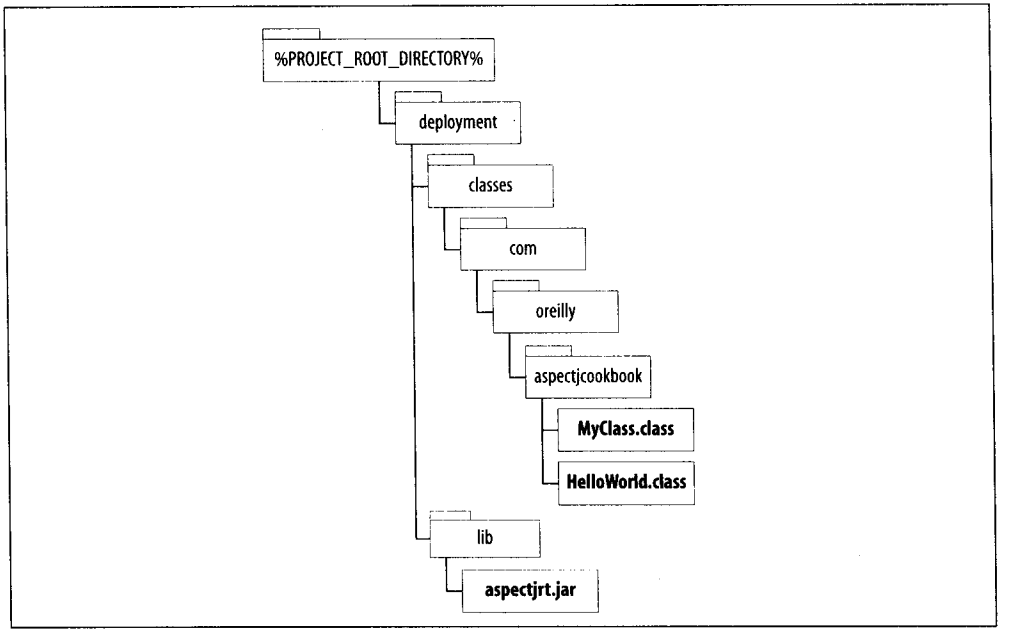

图3-2：添加了运行时AspectJ库之后的目录结构

7. 可以把*deployment*目录复制到正确安装了兼容的Java运行时环境的任何目标计算机上，并且应该能够使用`java`命令来运行应用程序：

```
java -classpath %MY_CLASSPATH% com.oreilly.aspectjcookbook.MyClass
```

目标计算机上的类路径必须包含*deployment/lib*目录内的*aspectjrt.jar*，并指向*classes*目录。例如，如果正从*deployment*目录内运行`java`命令，那么类路径的内容将类似于`classes/.:lib/aspectjrt.jar`，记住：在Windows上用；代替：。

参考

2.2节介绍了如何使用`ajc`命令来编译AspectJ应用程序的基础知识；David Flanagan所著的Java in a Nutshell（O'Reilly）提供了关于完整的Java命令行工具集（包括`java`）的详细信息。

3.2 将 AspectJ 应用程序部署为全包含的可执行 JAR 文件

问题

你想部署一个 AspectJ 应用程序，使之作为一个可执行的 JAR 文件运行。

解决方案

取出*aspectjrt.jar*文件的内容，然后把AspectJ类与你自己应用程序的类一起打包进单独一个JAR文件中。为了使单一的JAR文件可执行，应该包含一个合适的清单文件，其中列出了包含标准的`public static void main(String[])`方法（它是运行Java应用程序的入口点）的类。

讨论

创建一个可执行的JAR文件是部署传统的Java应用程序的流行方式。默认情况下，JAR文件格式包含用于软件组件的所有必要的类，并且它可能被配置成一个打包的Java应用程序，只需在大多数流行的操作系统内双击该文件，即可运行该应用程序。

以下步骤为2.2节中的应用程序手动创建了一个可执行的JAR文件：

1. 获取2.1节中创建的*deployment*目录的一个副本。
2. 从命令行使用`jar`工具，取出存储在*deployment/lib*目录中的*aspectjrt.jar*的内容：
   ```
   jar -xvf aspectjrt.jar
   ```
3. 完成jar取出之后，将发现从*deployment/lib*目录内取出了两个文件夹：*META-INF*和*org*。*META-INF*目录包含一个*MANIFEST.MF*文件，其中包含*aspectjrt.jar*的清单信息。*org*目录包含应用程序内面向方面的机制所需的*.class*文件，在部署应用程序时将需要运行它。
4. 把*deployment/lib/org*目录及其所有内容复制到*deployment/classes*目录中。

5. 使用文本编辑器在新的 *deployment* 目录中创建一个名为 *manifest* 的清单文件，其中包含的信息与例 3-1 中展示的信息相似。

   例 3-1：可执行 .jar 文件清单的内容示例

```
Manifest-Version: 1.0

Name: com/oreilly/aspectjcookbook/
Specification-Title: My simple AspectJ Application
Specification-Version: 1.0
Specification-Vendor: Russ Miles
Implementation-Title: com.oreilly.aspectjcookbook
Implementation-Version: 1.0
Implementation-Vendor: Russ Miles
Main-Class: com.oreilly.aspectjcookbook.MyClass
```

   重要的是：让清单文件中的 `Main-Class` 信息正确地指向包含启动应用程序的 `public void main(String[])` 方法的类。

6. 一旦创建了清单文件，并将其保存在 *deployment* 目录中，*deployment* 的目录结构以及 *deployment/classes* 目录看起来应该如图 3-3 所示。

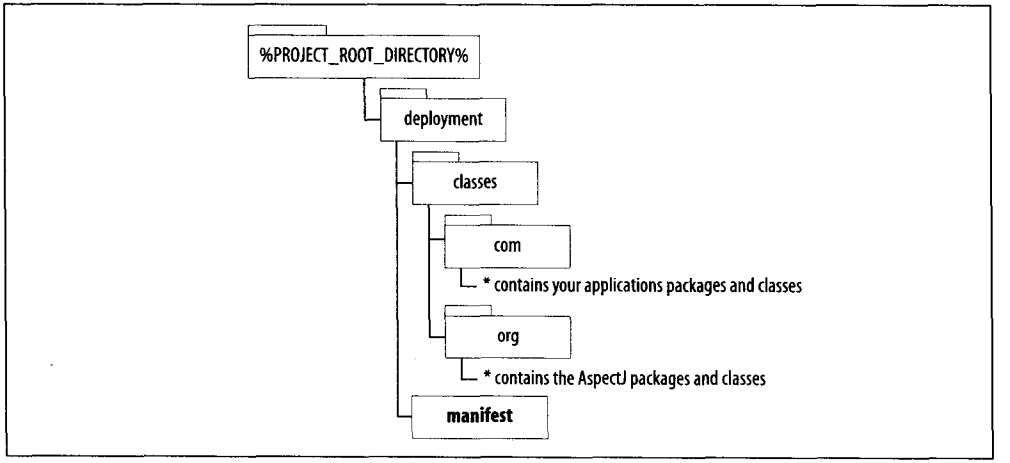

图 3-3：复制了 org 目录（它包含 AspectJ 类并创建一个新的清单文件）之后，deployment 和 classes 目录的内容

7. 创建 *deployment* 目录的一个名为 *build* 的子目录。

8. 从 *deployment* 目录内运行以下 `jar` 命令，在 *deployment/build* 目录中创建可执行的 *myapplication.jar* 文件：

```
jar -cfm build/myapplication.jar manifest -C classes/
```

9. 为了运行可执行的*.jar*文件，记住：这个示例应用程序是一个控制台应用程序，因此没有GUI，可以如下所示使用`-jar`选项从构建目录运行`java`命令，或者双击*myapplication.jar*文件。

```
java -jar myapplication.jar
```

10. 如果使用的是Mac OS X，并且决定双击*myapplication.jar*，那么可以先运行控制台应用程序（它位于*/Applications/utilities*目录内），来查看应用程序运行的输出，如图3-4所示。

图3-4：Mac OS X中的控制台应用程序显示了双击可执行的myapplication.jar后的输出

参考

David Flanagan所著的Java in a Nutshell（O'Reilly）提供了关于完整的Java命令行工具集（包括`java`和`jar`）的详细信息。

3.3 部署使用AspectJ的Java Servlet

问题

你想把使用AspectJ开发的servlet部署到Apache Tomcat中。

解决方案

使用`ajc`命令从命令行或者从Eclipse AspectJ项目内编译Java Servlet。在Apache Tomcat内的*webapps*目录下，建立一个新的Web应用程序目录和*WEB-INF*子目录。对Tomcat配置中的*server.xml*文件执行合适的修改，以启用Web应用程序。

把编译过的Java Servlet *.class*文件和相应的方面*.class*文件复制进*webapps/%YOUR_APPLICATION_DIRECTORY%/WEB-INF/classes*目录中。把*aspectjrt.jar*文件复制进

webapps/%YOUR_APPLICATION_DIRECTORY%/WEB-INF/lib 目录中，使得软件的面向方面的特性可以找到它们所需的支持组件。

修改 Web 应用程序的 *webapps/%YOUR_APPLICATION_DIRECTORY/WEB-INF/web.xml* 文件，以支持访问新的 Java Servlet。最后，重新启动 Tomcat，以激活 Web 应用程序。

讨论

下面的步骤显示了如何创建、编译和部署使用 AspectJ 的简单的 Java Servlet：

1. 创建类似于例 3-2 和例 3-3 中的 Java Servlet 和相应的方面。

例 3-2：简单的 HelloWorld Java Servlet

```
package com.oreilly.aspectjcookbook;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class AOHelloWorldServlet extends HttpServlet
{
   public void doGet (HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
   {
      ServletOutputStream out = response.getOutputStream();
      out.println("<h1>Hello World from an aspect-oriented Servlet!</h1>");
   }

   public String getServletInfo()
   {
      return "Create a page that says <i>Hello World</i> and send it back";
   }
}
```

例 3-3：通知 AOHelloWorldServlet 类上的 doGet(..)方法的方面

```
package com.oreilly.aspectjcookbook;

import java.io.IOException;

import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public aspect AddHTMLHeaderAndFooter
{
```

```
    public pointcut captureHttpRequest(HttpServletRequest request,
                                       HttpServletResponse response) :
      execution(public void AOHelloWorldServlet.doGet(HttpServletRequest,
                                                      HttpServletResponse)) &&
      args(request, response);

    before(HttpServletRequest request, HttpServletResponse response)
      throws IOException :
      captureHttpRequest(request, response)
    {
      response.setContentType("text/html");
      ServletOutputStream out = response.getOutputStream();
      out.println("<html>");
      out.println("<head><title>Adding a title using AspectJ!</title></head>");
      out.println("<body>");
    }

    after(HttpServletRequest request, HttpServletResponse response)
      throws IOException :
      captureHttpRequest(request, response)
    {
      ServletOutputStream out = response.getOutputStream();
      out.println("</body>");
      out.println("</html>");
    }
}
```

2. 使用 ajc 命令行工具或者作为 Eclipse 中的 AspectJ 项目的一部分，按正常的那样编译 Java Servlet 和方面。

3. 通过创建*%TOMCAT_INSTALL_DIRECTORY%/webapps*的一个名为*mywebapplication*的子目录，在 Tomcat 中创建一个新的 Web 应用程序目录。

4. 创建 *mywebapplication* 的一个名为 *WEB-INF* 的子目录，其中包含 *classes* 和 *lib* 目录。

5. 在*mywebapplication/WEB-INF*内创建一个名为*web.xml*的新文件，它有以下内容：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

    <display-name>MyApplication</display-name>
    <description>
      A simple web application that demonstrates the deployment of Java Servlets
      that have been built using AspectJ.
    </description>
```

```
    <servlet>
      <servlet-name>AOHelloWorld</servlet-name>
      <servlet-class>com.oreilly.aspectjcookbook.AOHelloWorldServlet</servlet-class>
    </servlet>

    <servlet-mapping>
       <servlet-name>AOHelloWorld</servlet-name>
       <url-pattern>/AOHelloWorld</url-pattern>
    </servlet-mapping>

</web-app>
```

6. 把编译Java Servlet和方面所生成的*.class*文件复制到*mywebapplication/WEB-INF/classes*目录中，记住：保持程序包目录结构。

7. 把*aspectjrt.jar*从*%ASPECTJ_INSTALL_DIRECTORY%/lib*复制到*mywebapplication/WEB-INF/lib*中。

8. 为方便起见，在*mywebapplication*内创建一个名为*index.html*的新文件，它具有以下内容：

```
<html>
<title>Homepage for MyApplication</title>
<body>
<h1>
Testing hompage for aspect-oriented Java Servlets</h1>
<p>
The purpose of this page is to simply provide a set of links with which
to conveniently access your aspect-oriented Java Servlets.
</p>
<p>
<ul>
  <li>
  <a href="AOHelloWorld">
    The simple HelloWorld aspect-oriented Java Servlet
  </a>
  </li>
</ul>
</p>
</body>
</html>
```

这是一个可选的步骤，它提供了一个HTML页面，可用于从浏览器方便地访问Java Servlet，而不必手动输入其特定的URL。

9. 部署的Web应用程序的目录结构看起来应该如图3-5所示。

10. 编辑*%TOMCAT_INSTALL_DIRECTORY%/conf/server.xml*，并添加下面几行代码，以在Tomcat内启用Web应用程序：

```
<Context path="/mywebapplication" docBase="mywebapplication" debug="0"
               reloadable="true" crossContext="true">
</Context>
```

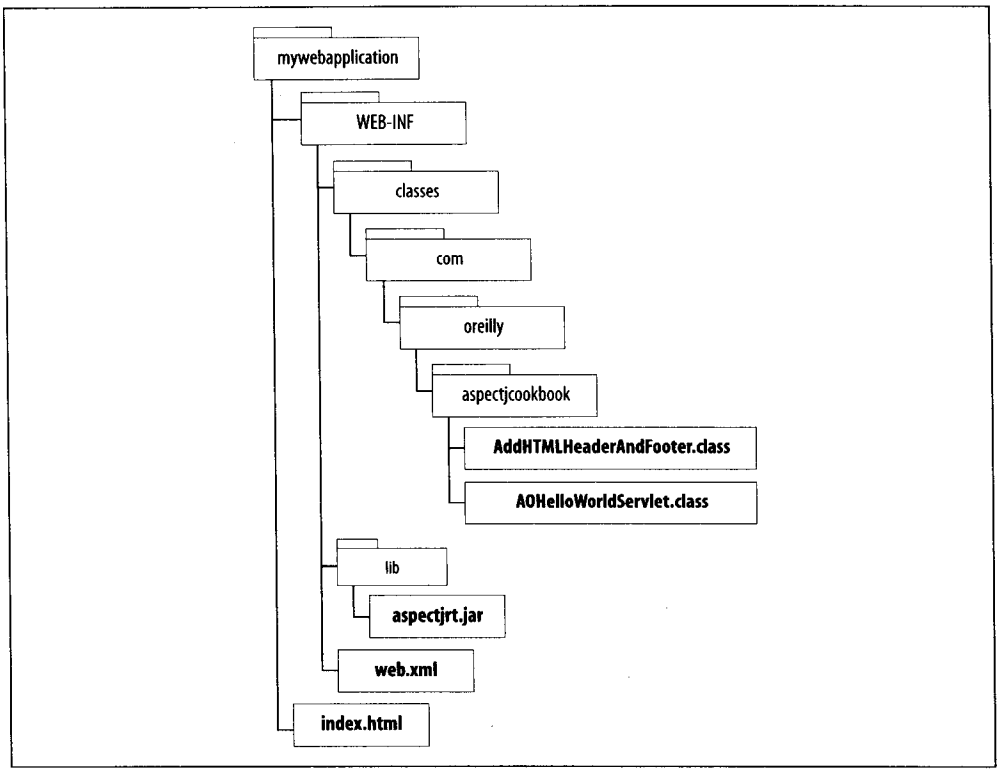

图 3-5：部署的 Web 应用程序目录和文件结构

11. 最后，重新启动 Tomcat，并且应该能够通过使用浏览器输入类似于 *http://<HOSTNAME>:<TOMCAT_PORT>/mywebapplication/index.html* 的 URL，并输入安装 Tomcat 的主机名和 Tomcat 端口号，来访问 Web 应用程序及其 Java Servlet。图 3-6 显示了 Web 应用程序的主页。
12. 通过单击 Web 应用程序主页上的“The simple HelloWorld aspect-oriented Java Servlet”链接，浏览器应该显示 Java Servlet 的输出，如图 3-7 所示。

参考

2.2 节中解释了使用 `ajc` 命令行工具编译应用程序；2.7 节中描述了在 Eclipse 中创建和编译 AspectJ 项目；4.4 节中解释了 `execution(Signature)` 切入点；11.3 节中解释了 `args([TypePatterns | Identifiers])` 切入点；13.3 节中讨论了 `before()` 形式的通知；13.5 节中说明了 `after()` 形式的通知；Bruce W. Perry 所著的 Java Servlet & JSP Cookbook（O'Reilly）包含如何在 Tomcat 及其他容器中配置和部署 Java Servlet 的示例；Jason Brittain 和 Ian Darwin 所著的 Tomcat: The Definitive Guide（O'Reilly）。

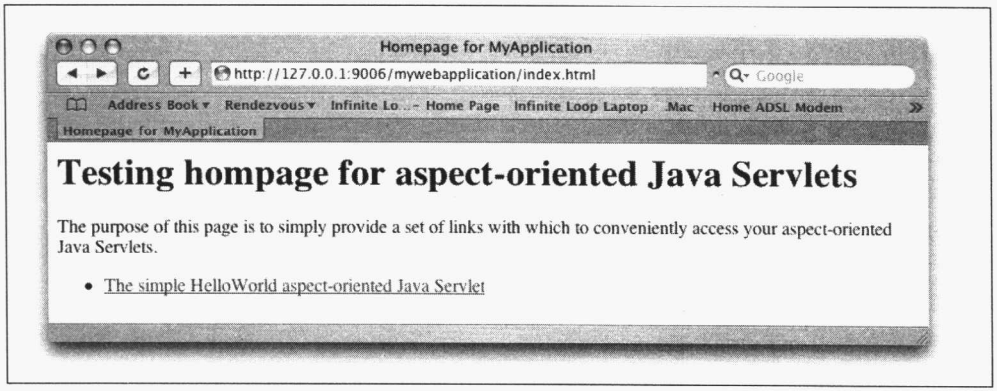

图 3-6：Web 应用程序的主页

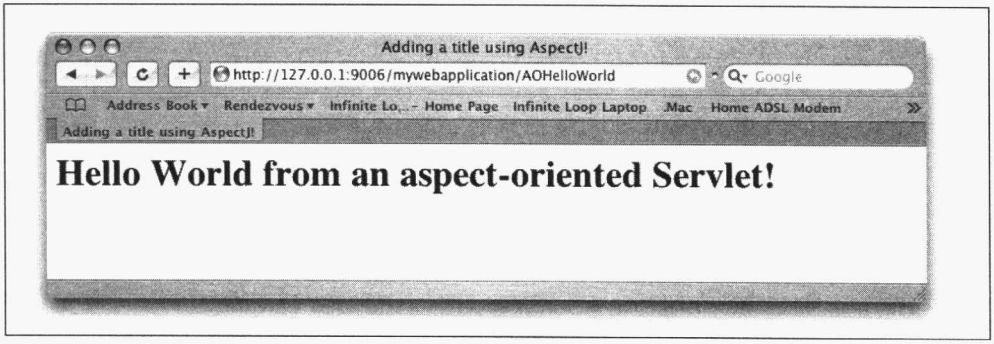

图 3-7：面向方面的 Java Servlet 的输出

3.4 部署使用 AspectJ 的 JSP

问题

你想把使用 AspectJ 的 JSP 构建和部署到 Apache Tomcat 中。

解决方案

使用Tomcat提供的工具，从JSP创建Java Servlet。使用`ajc`命令从命令行或者在Eclipse AspectJ 项目内编译 Java Servlet。

建立一个目录结构，从中部署完整的 JSP 应用程序，包括通过 `ajc` 命令行工具编译的 *aspectjrt.jar* 支持库和 *.class* 文件。

把部署目录完整地复制到 *%TOMCAT_INSTALL_DIRECTORY%/webapps* 目录中。在 Tomcat 配置中对 *server.xml* 文件执行适当的修改，以启用 Web 应用程序。最后，重新启动 Tomcat 以激活 Web 应用程序。

讨论

在使用自定义的编译器（如 AspectJ）时，Java Server Page 比 Java Servlet 更难以处理，因为JSP传统上通过像Apache Tomcat这样的Servlet容器透明地编译进Java Servlet中。

下面的步骤描述了如何构建使用 AspectJ 的简单 JSP，并将其部署进 Apache Tomcat 中：

1. 在项目区中创建一个名为 *jsp* 的目录。
2. 在 *jsp* 目录中创建一个名为 *simple.jsp* 的文件，它具有以下内容：

```
<html>
<body bgcolor="white">
<h1> Request Information </h1>
<font size="4"> JSP Request Method: <%= request.getMethod() %>
<br> Request URI: <%= request.getRequestURI() %>
<br> Request Protocol: <%= request.getProtocol() %>
<br> Servlet path: <%= request.getServletPath() %>
<br>
</font>
</body>
</html>
```

3. 在项目区中创建一个名为 *source* 的目录，用于保存方面和预处理的 JSP 源代码。
4. 在项目区中创建一个名为 *myjspapplication* 的目录和一个名为 *WEB-INF* 的子目录，用于在 Web 应用程序准备好部署到 Tomcat 时存放完成的 Web 应用程序。此外，把 *classes* 和 *lib* 子目录添加到 *WEB-INF* 目录中。一旦完成了这些更改，目录结构看起来应该如图 3-8 所示。
5. 从项目区中运行 Tomcat 自带的 `jspc` 命令行工具。这将生成源代码 Java Servlet，它是预编译 JSP 时的第一步：

```
jspc -d source/ -webxml myjspapplication/WEB-INF/web.xml -webapp jsp/
```

6. 在运行 `jspc` 命令行工具之后，将在 *myjspapplication/WEB-INF* 目录中找到一个 *web.xml* 文件，以及在 *source* 目录中找到一个 *simple_jsp.java* 文件。
7. 在 *source* 目录中创建一个名为 *AddCopyrightTomcatJSP.java* 的文件，其代码如例 3-4 所示。

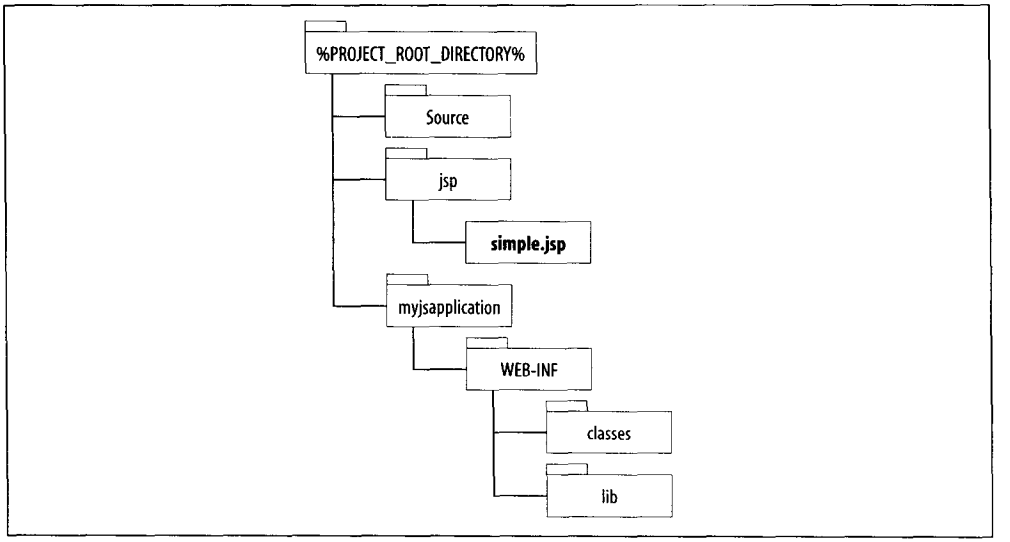

图 3-8：创建 myjspapplication 部署目录、source 目录以及 jsp 目录中的一个 .jsp 文件之后的目录结构

例 3-4：使用方面来影响 JSP 的输出

```
import javax.servlet.jsp.JspWriter;

public aspect AddCopyrightTomcatJSP
{
   public pointcut captureOutput(String message, JspWriter writer) :
      call(public void JspWriter.write(String) ) &&
      within(simple_jsp) &&
      args(message) &&
      target(writer) &&
      if (message.indexof("</body>")!=-1;

   before(String message, JspWriter writer) : captureOutput(message, writer)
   {
      try
      {
        writer.write("<p>Copyleft Russ Miles 2004</p>\n");
      }
      catch (Exception e)
      {
        e.printStackTrace();
      }
   }
}
```

8. 从项目根目录中使用ajc命令行工具编译*source*目录中的两个文件：*simple_jsp.java*和*AddCopyrightTomcatJSP.java*：

```
ajc -classpath %TOMCAT_INSTALL_DIRECTORY%/common/lib/servlet.jar:%TOMCAT_INSTALL_
DIRECTORY%/common/lib/jasper-runtime.jar -d myjspapplication/WEB-INF/classes
source/simple_jsp.java source/AddCopyrightTomcatJSP.java
```

9. 运行这个 ajc 命令的结果是：将在 *myjspapplication/META-INF/classes* 目录中生成两个 *.class* 文件。

10. 从 *%ASPECTJ_INSTALL_DIRECTORY%/lib* 把 *aspectjrt.jar* 复制到 *myweb-application/WEB-INF/lib* 中。

11. 为方便起见，在 *mywebapplication* 内创建一个名为 *index.html* 的新文件，它具有以下内容：

```
<html>
<title>Homepage for MyJSPApplication</title>
<body>
<h1>
Testing hompage for aspect-oriented Java Server Pages</h1>
<p>
The purpose of this page is to simply provide a set of links with which
to conveniently access your aspect-oriented Java Server Pages.
</p>
<p>
<ul>
  <li>
  <a href="simple.jsp">
    The simple aspect-oriented Java Server Page example
    </a>
  </li>
</ul>
</p>
</body>
</html>
```

这一步是可选的，它提供了一个HTML页面，可用于从浏览器方便地访问JSP，而不必手动输入其特定的URL。

12. 可部署的 *myjspapplication* 目录的最终内容如图 3-9 所示。

13. 把整个 *myjspapplication* 目录及其所有内容复制到 *%TOMCAT_INSTALL_DIRECTORY%/webapps* 中。

14. 编辑 *%TOMCAT_INSTALL_DIRECTORY%/conf/server.xml* 并添加下面几行代码，以在 Tomcat 内启用 Web 应用程序：

```
<Context path="/myjspapplication" docBase="myjspapplication" debug="0"
                reloadable="true" crossContext="true">
</Context>
```

15. 最后，重新启动 Tomcat，并且应该能够通过使用浏览器输入类似于 *http://<HOSTNAME>:<TOMCAT_PORT>/myjspapplication/index.html* 的URL，并输入安装 Tomcat 的主机名和 Tomcat 端口号，来访问 Web 应用程序及其 Java Servlet。图 3-10 显示了新的 JSP Web 应用程序的主页。

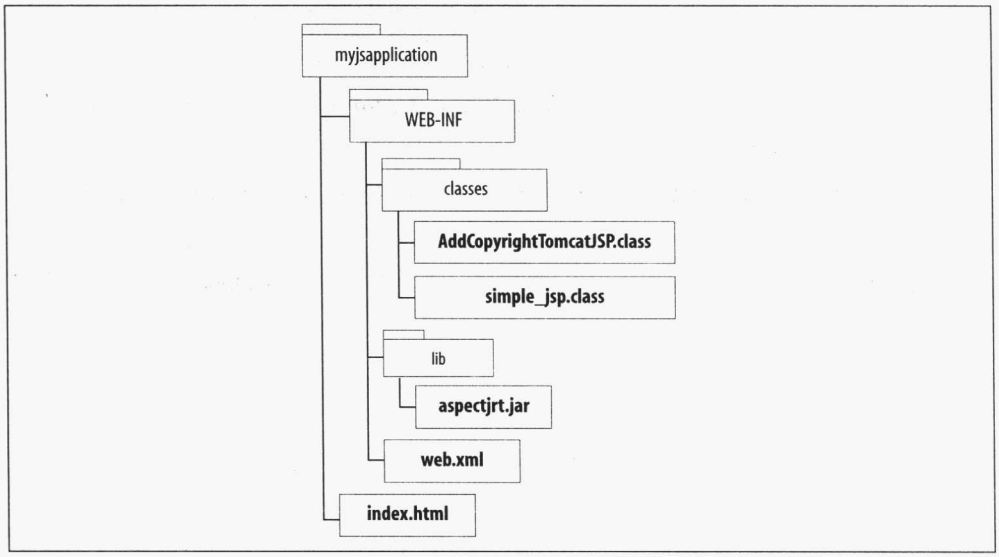

图 3-9：在准备好部署到 Tomcat 中时，myjspapplication 目录的内容

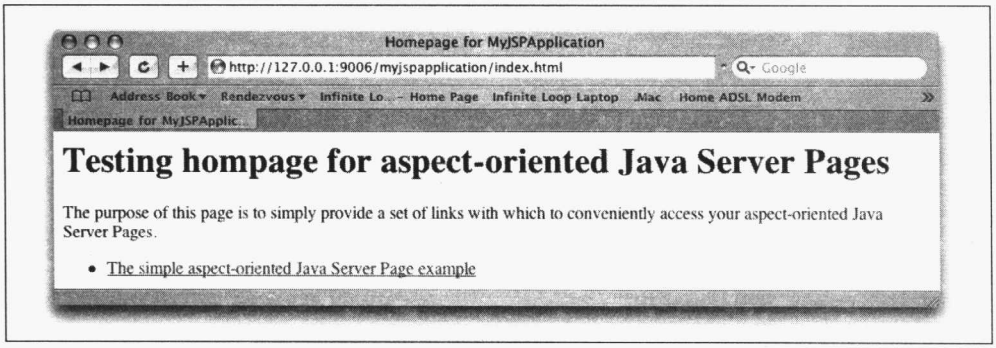

图 3-10：JSP Web 应用程序的主页

16. 通过在 Web 应用程序主页上单击“The simple aspect-oriented Java Server Page example”链接，浏览器应该显示 JSP 的输出，如图 3-11 所示。

    将使用 `AddCopyrightJSPTomcat` 方面把“Copyleft Russ Miles 2004”消息织入到 JSP 的输出中。

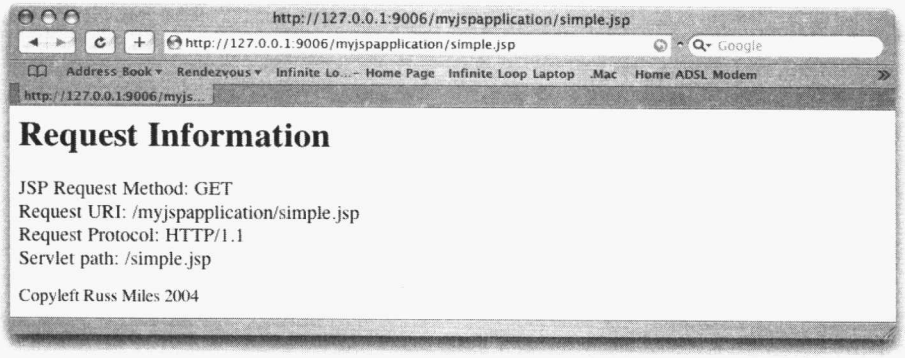

图 3-11：面向方面的 Java Server Page 的输出

参考

2.2 节中解释了使用 ajc 命令行工具编译应用程序；2.7 节中描述了在 Eclipse 中创建和编译 AspectJ 项目；4.1 节中解释了 `call(Signature)` 切入点；9.1 节中讨论了 `within(TypePattern)` 切入点；11.2 节中解释了 `target(TypePattern | Identifier)` 切入点；11.3 节中解释了 `args([TypePatterns | Identifiers])` 切入点；12.1 节中描述了 `if(Expression)` 语句；13.4 节中讨论了 `around()` 形式的通知；Bruce W. Perry 所著的 *Java Servlet & JSP Cookbook*（O'Reilly）包含如何在 Tomcat 及其他容器中配置和部署 *Java Servlet* 的示例；Jason Brittain 和 Ian Darwin 所著的 *Tomcat: The Definitive Guide*（O'Reilly）。

3.5 部署使用 AspectJ 的 Axis Web 服务

问题

你想部署使用 AspectJ 创建的面向方面的 Apache Axis Web 服务。

解决方案

使用 `ajc` 命令行工具或者在 Eclipse 中的 AspectJ 项目中利用方面创建和编译 Web 服务 Java 类。将应用程序的 *.class* 文件复制到 *%AXIS_INSTALLATION_IN_TOMCAT%/WEB-INF/classes* 目录中，并将 *aspectjrt.jar* 复制到 *%AXIS_INSTALLATION_IN_TOMCAT%/WEB-INF/lib* 中。

创建一个Apache Axis部署配置文件，并使用org.apache.axis.client.AdminClient命令行工具利用Apache Axis注册新的Web服务。

讨论

Apache Axis正迅速成为针对Java开发人员的最流行的Web服务实现之一。它不仅是开源的，而且由于其非常活跃的支持开发人员基础，它在与使用其他Web服务框架（如.NET）的互操作性方面也取得了高度的成功。

一旦在计算机上的Tomcat内运行Apache Axis，就可以使用以下步骤来编译一个简单的面向方面的Web服务，并将其部署在Apache Axis内：

1. 在项目内创建一个名为*source*的目录，并在这个新目录内创建一个名为*MyWebService.java*的文件，它包含以下内容：

```
package com.oreilly.aspectjcookbook;

public class MyWebService
{
  public String echo(String message)
  {
      return message;
  }
}
```

2. 在*source*目录中创建另一个名为*AddMessageHeaderAspect.java*的文件，它包含以下内容：

```
package com.oreilly.aspectjcookbook;

public aspect AddMessageHeaderAspect
{
   public pointcut captureEcho(String message) :
      execution(public void MyWebService.echo(String)) &&
      args(message);

   Object around(String message) : captureEcho(message)
   {
      return "Your original message was: " + message;
   }
}
```

3. 在项目区内创建一个*classes*目录。

4. 从项目根目录中使用ajc命令行工具编译*source*目录中的两个文件：*MyWebService.java*和*AddMessageHeaderAspect.java*：

```
ajc -classpath %AXIS_INSTALL_DIRECTORY%/lib/axis.jar:%AXIS_INSTALL_DIRECTORY%/
lib/axis.jar -d classes/ source/*.java
```

5. 一旦成功地运行 ajc 命令，就会在 *classes* 目录下生成编译过的 *.class* 文件和程序包结构。

6. 在 *classes* 目录内创建一个名为 *deploy.wsdd* 的文件，它将用于自动把 Web 应用程序部署到 Axis 中。*deploy.wsdd* 应该包含以下信息：

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

 <service name="MyWebService" provider="java:RPC">
   <parameter name="className" value="com.oreilly.aspectjcookbook.MyWebService"/>
   <parameter name="allowedMethods" value="*"/>
 </service>

</deployment>
```

7. 一个良好的惯例是：在与 *deploy.wsdd* 相同的类目录中创建一个 *undeploy.wsdd* 文件，以能够从 Axis 中删除 Web 应用程序。*undeploy.wsdd* 应该包含以下信息：

```
<undeployment xmlns="http://xml.apache.org/axis/wsdd/">
 <service name="MyWebService"/>
</undeployment>
```

8. 把 *%ASPECTJ_INSTALL_DIRECTORY%/lib/aspectjrt.jar* 复制到 *%AXIS_INSTALLATION_IN_TOMCAT%/WEB-INF/lib* 中。

9. 在保持程序包目录结构的情况下，把 *classes* 目录中的 *.class* 文件复制到 *%AXIS_INSTALLATION_IN_TOMCAT%/WEB-INF/classes* 中。

10. 如果正把项目部署到运行在 Tomcat 内的 Apache Axis 中，则要确保 Tomcat 正在运行，从 *classes* 目录中运行以下 java 命令，以运行 Apache Axis 部署程序，它将用 Axis 注册新的 Web 服务：

```
java -classpath %AXIS_INSTALL_DIRECTORY%/lib/axis.jar:
%AXIS_INSTALL_DIRECTORY%/
lib/jaxrpc.jar:%AXIS_INSTALL_DIRECTORY%/lib/log4j-
1.2.4.jar:%AXIS_INSTALL_DIRECTORY%/lib/commons-
logging.jar:%AXIS_INSTALL_DIRECTORY%/lib/commons-
discovery.jar:%AXIS_INSTALL_DIRECTORY%/lib/saaj.jar
org.apache.axis.client.AdminClient -h%HOST% -p%PORT% deploy.wsdd
```

11. 重新启动 Tomcat，确保它刷新了所管理的所有库和应用程序，包括 Axis。

12. 要测试已经正确部署了Web服务，可以在浏览器中转到 *http://<hostname>:<post>/axis/services/MyWebService*，并且应该看到图 3-12 中所示的输出。

13. Web服务客户现在可以使用新的面向方面的Web服务，例如，使用 MyWebService 的简单 AppleScript 的输出如图 3-13 所示。

图 3-12：面向方面的 MyWebService 服务，它进行了初始化，并准备好在 Apache Axis 内使用

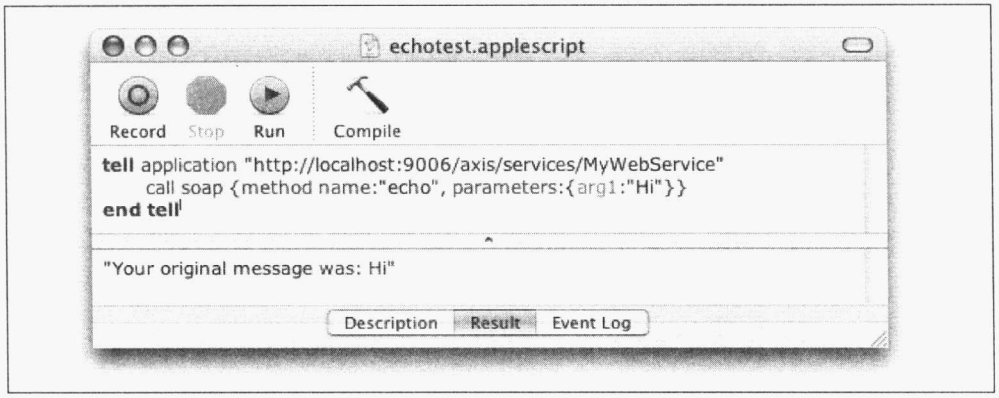

图 3-13：在 MyWebService 服务上使用 echo 方法的 AppleScript Web 服务客户

参考

2.2 节中解释了使用 `ajc` 命令行工具编译应用程序；2.7 节中描述了在 Eclipse 中创建和编译 AspectJ 项目；4.4 节中解释了 `execution(Signature)` 切入点；11.3 节中解释了 `args([TypePatterns | Identifiers])` 切入点；Christopher Haddad、Kevin Bedell 和 Paul Brown 所著的 *Programming Apache Axis*（O'Reilly）；Bruce W. Perry 所著的 *AppleScript in a Nutshell*（O'Reilly）。

第4章

捕获方法上的连接点

4.0 简介

连接点是一个特定的点，在这个点上可以把通知织入应用程序的代码中。切入点为选择将调用一份通知的连接点提供了逻辑定义。

下面9章对应于Java语言构造的类型，它们包含在AspectJ中可以使用切入点声明捕获的连接点。本章重点介绍可以捕获所选连接点的切入点，这些连接点可供Java方法使用。

`before()`形式的通知用于构成下面8章的大多数基于切入点的秘诀，它们通过使用不同类型的通知来避免混淆。在不可避免的地方，可能不得不使用其他形式的通知，因此，参考第9章以了解带给所提供解决方案的不同形式通知的隐含意义可能是有帮助的。

注意： 一旦掌握了AspectJ在第4~12章中提供的不同类型的切入点，即可通过到达*http://www.eclipse.org/aspectj*，然后单击Documentation → standard pointcut idioms，来查看Mik Kersten的标准切入点习惯用语。这些可重用的切入点定义提供了一些优秀的工具，可以利用它们来构造自己的切入点逻辑。

4.1 捕获方法调用

问题

在调用匹配特定签名的方法时，你想捕获它。

解决方案

使用call(Signature)切入点。call(Signature)切入点的语法是：

```
pointcut <pointcut name>(<any values to be picked up>) :
   call(<optional modifier> <return type> <class>.<method>(<paramater types>));
```

警告：与Java方法参数相比，Signature参数中的空白是重要的，因为它用于隔开不同的成分。

讨论

call(Signature)切入点具有两个关键特征：

1. 在方法调用上触发通知；其环境是调用类。
2. Signature可以包含通配符，用于选择不同类和方法上的一系列连接点。

表4-1显示了在把方法Signature提供给切入点声明时，可以使用的通配符选项的一些示例。

表4-1：在方法Signature内使用通配符的示例

具有通配符的签名	描述
* void MyClass.foo(int, float) void MyClass.foo(int, float)	无论修饰符是什么，都会捕获方法上的连接点。也可以通过完全忽略可见性来做到这一点
* * MyClass.foo(int, float) * MyClass.foo(int, float)	无论修饰符或返回类型是什么，都会捕获方法上的连接点
* * *.foo(int,float) * * foo(int,float)	无论修饰符、返回类型或类是什么，都会捕获方法上的连接点
* * *.*(int,float)	无论修饰符、返回类型、类或方法是什么，都会捕获方法上的连接点
* * *.*(*,float)	无论修饰符、返回类型、类，或者其中的参数包含任何内容并且后接一个浮点数的方法是什么，都会捕获方法上的连接点
* * *.*(*,..)	无论修饰符、返回类型、类，或者其中的参数包含至少一个单值并且后接任意数量参数的方法是什么，都会捕获方法上的连接点

表 4-1：在方法 Signature 内使用通配符的示例（续）

具有通配符的签名	描述
* * *.*(..) * *(..)	无论修饰符、返回类型、类，或者其中有任意数量参数的方法是什么，都会捕获方法上的连接点
* mypackage..*.*(..)	捕获mypackage包和子包内的任何方法上的连接点
* MyClass+.*(..)	捕获MyClass和任何子类中的任何方法上的连接点

例4-1显示了用于在所有与签名MyClass.foo(int,String)匹配的方法中声明一个关注的call(Signature)切入点。

例 4-1：使用 call(Signature)切入点来捕获对特定方法签名的调用

```
public aspect CallRecipe
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets called:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: void
   Method Parameters: an int followed by a String
   */
   pointcut callPointCut() : call(void MyClass.foo(int, String));

   // Advice declaration
   before() : callPointCut()
   {

      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice attached to the call point cut");
      System.out.println(
         "Actually executing before the point cut call ...");
      System.out.println("But that's a recipe for Chapter 6!");
      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");

      }
}
```

图 4-1 显示了如何把 call(Signature)切入点应用于简单的类。

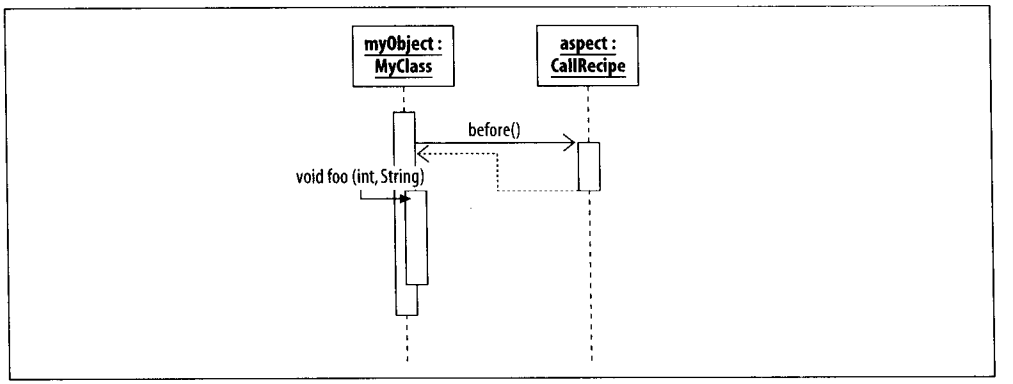

图4-1：如何应用call(Signature)切入点

要知道：当在对象的继承和/或重写方法上捕获连接点时，call(Signature)和execution(Signature)切入点定义在某些情形下可能导致奇怪的行为，这依赖于对象的动态和静态类型。

在AspectJ中，call(Signature)切入点和execution(Pointcuts)（参见4.4节）可能具有奇怪的行为，这依赖于方法的目标的静态和动态类型。考虑下面的代码：

```
A someObject = new E();
someObject.foo();
```

在这个简单的示例中，E是A的子类；依据Java中的动态类型化规则，someObject的静态类型是A，而其动态类型是E。然后可以声明一个call(Signature)切入点来捕获对someObject.foo()的调用：

```
call(public void A.foo())
```

如果foo()方法是在A中声明的，并且被E继承，那么切入点将捕获对方法的调用。不过，如果foo()方法在E中被重写，那么call(Signature)切入点仍会捕获方法调用连接点。这最初看起来似乎有些奇怪，但是，如果把call(Signature)切入点看作检查someObject的静态类型（它仍然是A），它就是有意义的。

现在，事情变得有点奇怪。如果把someObject的静态类型更改为E，保持foo()方法在E中被重写，并把使用方法的代码更改如下，则会发生什么：

```
E someObject = new E();
someObject.foo();
```

对象的静态类型与其动态类型相同，它们都是E，这仍然是A的一个子类。foo()在E中被重写，这样，A中的任何代码都不会被调用，也不会是A引用的静态类型。按以前

那样使用相同的切入点定义，就不会期待对someObject.foo()的调用将被捕获，但是，事实上它会被捕获。在这种情况下，你可能期待强制使用+继承指定符来捕获对foo()的调用，例如：

```
call(public void A+.foo()) // Captures calls to foo() on A and all subclasses
```

由于AspectJ中实现call(Signature)切入点的方式，无需+指定符来捕获对子类中重写方法的调用。看起来似乎即使把someObject的静态和动态类型声明为E，因为foo()是存在于A（它仍然是E的超类）中的一个方法，这样原来的call(Signature)切入点定义仍然会捕获方法调用。当认为原来的切入点定义甚至没有提及E，并且它没有使用+继承指定符来指示A的子类中的关注时，这看起来甚至更奇怪。

这只是call(Signature)切入点与继承和/或重写的方法（依赖于对象的静态和动态类型以及Signature内声明的类型）协同工作的微妙并且有时令人混淆的方式的一个例子。execution(Signature)切入点定义具有类似但并不完全相同的问题，因为它更多地强调对象的动态类型，它也许是在捕获方法内的连接点（与方法调用上的连接点相对）时所期待的。

彻底研究这些微妙之处将需要一份详尽的报告，在*www.cs.iastate.edu/~leavens/FOAL/papers-2004/barzilay-etal.pdf*上有一份这样的报告。日常使用call(Signature)和execution(Signature)可能不会导致遇到这些问题；不过，至少记住它们并知道它们存在（以防万一）是有帮助的。

参考

在*www.cs.iastate.edu/~leavens/FOAL/papers-2004/barzilay-etal.pdf*上可以得到一份报告，其中更详细地解释了捕获继承或重写方法中的连接点时call(Signature)和execution(Signature)切入点的微妙特征；4.2节中说明了如何捕获连接点上的参数，特别是方法调用上的连接点的参数；第13章描述了AspectJ中可用的不同类型的通知。

4.2 捕获方法调用上传递的参数值

问题

你想捕获并使用方法调用上传递的参数。

解决方案

创建一个切入点，用于指定你想在其签名中作为标识符捕获的参数。使用call(Signature)和args([TypePatterns | Identifiers])切入点来捕获对方法的调用，然后把需要的标识符绑定到方法的参数值上。

讨论

例4-2显示了用于在所有与签名MyClass.foo(int,String)匹配的方法中声明一个关注的call(Signature)切入点。captureCallParameters(int,String)切入点需要通过值和名称标识符指定一个int和一个String。然后通过args([Types | Identifiers])切入点把这些标识符绑定到方法参数上。

例4-2：捕获在对MyClass.foo(..)方法的调用上传递的int和String值

```
public aspect CaptureCallParametersRecipe
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets called:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: void
   Method Parameters: an int followed by a String
   */
   pointcut captureCallParameters(int value, String name) :
      call(void MyClass.foo(int, String)) &&
      args(value, name);

   // Advice declaration
   before(int value, String name) : captureCallParameters(value, name)
   {

      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice attached to the call point cut");
      System.out.println("Captured int parameter on method: " + value);
      System.out.println("Captured String parameter on method: " + name);
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

before()通知可以访问在captureCallParameters(int,String)上声明的标识符，其方式是：在其签名中包括值和名称标识符，然后把这些标识符绑定到captureCallParameters(int,String)切入点。

参考

4.1 节中描述了 call(Signature)切入点；4.1 节中还说明了一些可以用在 Signature 中的通配符变体；11.3 节中讨论了 args([Types | Identifiers])切入点；12.2 节中说明了使用逻辑 AND (&&) 结合切入点；13.3 节中说明了 before()形式的通知；第 13 章中介绍了可用于通知的调用环境。

4.3 捕获方法调用的目标

问题

你想捕获作为方法调用的对象。

解决方案

创建一个切入点，它指定了单一参数，这个参数的类型与希望捕获的方法调用目标的类型相同。使用 call(Signature)和 target(Type | Identifier)切入点来捕获方法的调用，然后把单一标识符绑定到正在调用方法的对象上。

讨论

例4-3说明了用于在所有与签名MyClass.foo(int,String)匹配的方法中声明一个关注的 call(Signature)切入点。captureCallTarget(MyClass)切入点需要一个 MyClass 对象，它由 myObject 标识符指定。然后把 myObject 标识符绑定到由 MyClass.foo(int,String)方法通过 target(Type | Identifier)切入点调用的对象上。

例 4-3：捕获正在其上调用 MyClass.foo(..)方法的对象

```
public aspect CaptureCallTargetRecipe
{
    /*
    Specifies calling advice whenever a method
    matching the following rules gets called:

    Class Name: MyClass
    Method Name: foo
    Method Return Type: void
    Method Parameters: an int followed by a String
    */
    pointcut captureCallTarget(MyClass myObject) :
        call(void MyClass.foo(int, String)) &&
        target(myObject);
```

```
    // Advice declaration
    before(MyClass myObject) : captureCallTarget(myObject)
    {

        System.out.println(
            "------------------- Aspect Advice Logic ----------- --------");
        System.out.println(
            "In the advice attached to the call point cut");
        System.out.println("Captured target object for the method call: " + myObject);
        System.out.println(
            "------------------------------------------------------------");
    }
}
```

before()通知可以访问在captureCallTar-get(MyClass)切入点上声明的单一标识符，其方式是：在其签名中包括myObject标识符，然后把那个标识符绑定到captureCallTarget(MyClass)切入点。

参考

4.1节中描述了call(Signature)切入点；4.1节中还说明了一些可以用在Signature中的通配符变体；11.2节中讨论了target(Type | Identifier)切入点；12.2节中说明了使用逻辑AND（&&）结合切入点；13.3节中说明了before()形式的通知；第13章中介绍了可用于通知的调用环境。

4.4 当执行一个方法时捕获它

问题

你想在执行与特定签名匹配的方法时捕获它。

解决方案

使用execution(Signature)切入点。execution(Signature)切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
    execution((<optional modifier> <return type> <class>.<method>(<paramater types>);
```

讨论

execution(Signature)切入点具有两个关键特征：

1. 触发连接点的环境在目标类方法中。
2. Signature可以包含通配符，以选择不同类和方法上的一系列连接点。

例4-4显示了execution(Signature)切入点，它用于在与签名MyClass.foo(int,String)匹配的任何方法上的方法执行连接点中声明一个关注。

例 4-4：使用 execution(Signature)方法来捕获方法执行中的连接点

```
public aspect ExecutionRecipe
{
   /*
       Specifies calling advice whenever a method
       matching the following rules enters execution:

       Class Name: MyClass
       Method Name: foo
       Method Return Type: int
       Method Parameters: an int followed by a String
   */
   pointcut executionPointcut() : execution(void MyClass.foo(int, String));

   // Advice declaration
   before() : executionPointcut() && !within(ExecutionRecipe +)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println("In the advice picked by ExecutionRecipe");
      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

图 4-2 显示了如何将 execution(Signature)切入点应用于简单的类。

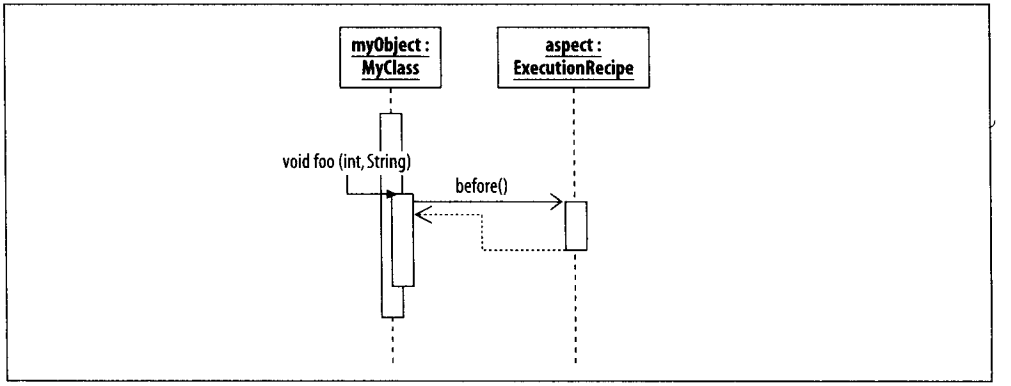

图 4-2：如何应用 execution(Signature)切入点

首先，与上一个秘诀中描述的call(Signature)切入点相比，execution(Signature)切入点看起来似乎没有提供更多的内容。对于这个秘诀要记住的重要事情是：在什么地方调用通知，以及它的环境是什么。

就call(Signature)切入点来说，在调用方法的地方调用通知。调用通知的环境是调用类。一旦进入方法，因而调用环境是正在执行的方法时，就会调用execution(Signature)切入点。

最后，如果还没有阅读4.1节，那么值得翻回几页来阅读而知：在捕获继承和/或重写（依赖于对象的静态和动态类型）的对象方法上的连接点时，call(Signature)和execution(Signature)切入点可能具有哪些奇怪的行为。

参考

在*www.cs.iastate.edu/~leavens/FOAL/papers-2004/barzilay-etal.pdf*上可以得到一份报告，其中更详细地解释了捕获继承或重写方法中的连接点时，call(Signature)和execution(Signature)切入点的微妙特征；4.1节中说明了一些可用在Signature中的通配符变体；4.2节中说明了如何捕获连接点上的参数，特别是方法调用上的连接点的参数；第13章介绍了可用于通知的调用环境。

4.5 在执行方法时捕获this引用的值

问题

在执行期间捕获方法时，你想显示Java的this引用所指向的对象，使之可以被通知使用。

解决方案

创建一个切入点，它指定了单一参数，这个参数的类型与想捕获的this引用的类型相同。使用execution(Signature)和this(Type | Identifier)切入点来捕获方法的执行，然后把单一标识符绑定到方法执行期间this引用所指向的对象上。

讨论

例4-5说明了execution(Signature)切入点，它用于在所有与签名MyClass.foo(int,String)匹配的方法中声明一个关注。captureThisDuringExecution(MyClass)切入点需要一个MyClass对象，它由myObject标识符指定。然后把myObject标识符

绑定到this引用通过this(Type | Identifier)切入点所引用的方法上。

例4-5：在执行MyClass.foo(..)方法期间捕获this引用

```
public aspect CaptureThisReferenceRecipe
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets executed:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: void
   Method Parameters: an int followed by a String
   */
   pointcut captureThisDuringExecution(MyClass myObject) :
      execution(void MyClass.foo(int, String)) &&
      this(myObject);

   // Advice declaration
   before(MyClass myObject) : captureThisDuringExecution(myObject)
   {

      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice attached to the execution point cut");
      System.out.println("Captured this reference: " + myObject);
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

before()通知可以访问引用对象的标识符，在方法执行期间，通过this来指向这个对象，其方式如下：在其签名中包含myObject标识符，然后把它绑定到captureThisDuringExecution(MyClass)切入点上。

参考

4.1节中说明了一些可以用在Signature中的通配符变体；4.4节中说明了execution(Signature)切入点；4.2节中说明了如何捕获连接点上的参数，特别是方法调用上的连接点的参数；11.1节中讨论了this(Type | Identifier)切入点；12.2节中说明了使用逻辑AND（&&）结合切入点；13.3节中说明了before()形式的通知。

第5章

捕获异常处理上的连接点

5.0 简介

本章说明了如何包含异常处理，作为应用于使用方面的应用程序的横切关注点的一部分。

在Java中抛出异常时，会将其向上传递给调用链，直到它被作为try/catch块一部分的catch语句处理，或者它到达Java运行库并在控制台上引发一条混乱的消息。如果捕获到Java异常，就会把该异常作为一个对象传递给相应的catch块，其中可能会发生合适的动作，以处理问题。

应用于应用程序的方面可能发现知道何时处理异常是有用的。一种极有可能发生的情况是：除了catch块的正常行为之外，使用方面实现的横切行为的一部分还需要做某些事情，或者用于代替catch块的正常行为。

AspectJ提供了handler(TypePattern)切入点，在希望捕获何时用特定类型的异常调用catch块时就会用到它。本章说明了在方面中使用handler(TypePattern)的一些方式，使得它们可以捕获目标应用程序可能引发的异常，并与之交互。

5.1 捕获何时捕捉异常

问题

你想捕获何时捕捉特定类型的异常。

解决方案

使用handler(TypePattern)切入点。handler(TypePattern)切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) : handler(<class>);
```

讨论

handler(TypePattern)切入点具有5个关键特征：

1. handler(TypePattern)在捕捉异常的作用域内选择连接点。
2. handler(TypePattern)切入点的通知仅用于类型模式指定Throwable或其子类的地方。
3. TypePattern声明无论何时捕捉到异常或其子类的匹配类型，都会应用相应的通知。
4. handler(TypePattern)切入点只支持before()形式的通知。这意味着不能使用像around()这样的通知来重写catch块的正常行为。
5. TypePattern可以包含通配符，用于选择不同类上的一系列连接点。

表5-1显示了在把TypePattern用于切入点声明时，可以使用的通配符选项的一些示例。

表5-1：在TypePattern内使用通配符的示例

带有通配符的TypePattern	描述
mypackage..*	捕获mypackage包及其子包中的连接点类
MyClass+	捕获MyClass类及其任何子类中的连接点

例5-1显示了handler(TypePattern)切入点，用于捕获被捕捉的MyException异常。

例5-1：在捕捉特定类型的异常时，使用handler(TypePattern)切入点来捕获连接点

```
public aspect HandlerRecipe
{
   /*
       Specifies calling advice when any exception object is caught
       that matches the following rules for its type pattern:

       Type: MyException
   */
   pointcut myExceptionHandlerPointcut() : handler(MyException);

   // Advice declaration
   before() : myExceptionHandlerPointcut()
   {
      System.out.println(
         "-------------- Aspect Advice Logic ---------------");
```

```
        System.out.println(
            "In the advice picked by " + "myExceptionHandlerPointcut()");
        System.out.println(
            "Signature: "
                + thisJoinPoint.getStaticPart().getSignature());
        System.out.println(
            "Source Line: "
                + thisJoinPoint.getStaticPart().getSourceLocation());
        System.out.println(
            "-------------------------------------------------");
    }
}
```

注意： handler(TypePattern)切入点将在捕捉异常（而不是引发异常）的地方捕获连接点。

图 5-1 显示了如何把 handler(TypePattern)切入点应用于简单的类层次结构。

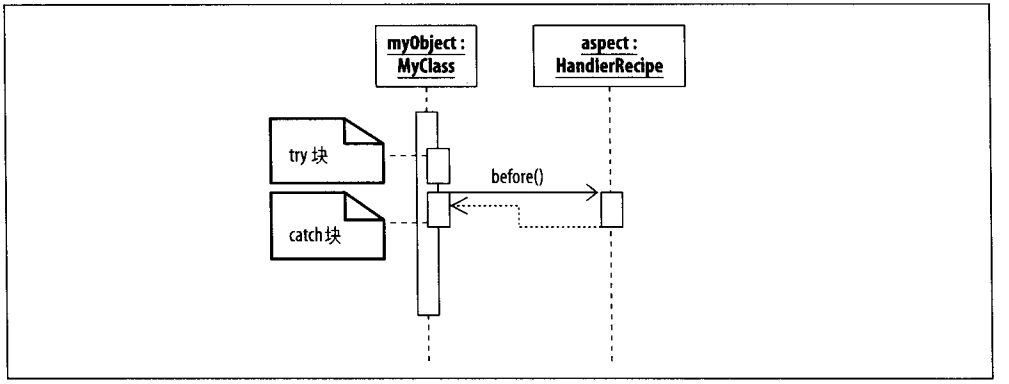

图 5-1：如果应用 handler(TypePattern)切入点

参考

第 13 章描述了 AspectJ 中可用的不同类型的通知。

5.2 捕获抛出的异常

问题

在由使用 handler(TypePattern)切入点捕捉的连接点触发的通知内，你想访问被相应通知内的 catch 块捕捉的异常。

解决方案

结合使用 args([Types | Identifiers])切入点与 handler(TypePattern)切入点，将捕捉的异常展示为切入点上的标识符，可将其传递给相应的通知。

讨论

例 5-2 显示了如何把 MyException 异常传递给 before()通知，作为 myExceptionHandlerPointcut 切入点上的异常标识符。

例 5-2：访问捕捉的 MyException 异常

```
public aspect AccessThrownException
{
   pointcut myExceptionHandlerPointcut(MyException exception) :
      handler(MyException) &&
      args(exception);

   // Advice declaration
   before(MyException exception) : myExceptionHandlerPointcut(exception)
   {
      System.out.println(
         "-------------- Aspect Advice Logic ---------------");
      System.out.println(
         "In the advice picked by " + "myExceptionHandlerPointcut()");
      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());

      System.out.println("Exception caught:");
      exception.printStackTrace();

      System.out.println(
         "--------------------------------------------------");
   }
}
```

参考

5.1 节中说明了 handler(TypePattern)切入点的语法；5.1 节中还说明了可以用在 TypePattern中的一些通配符变体；11.3节中解释了args([Types | Identifiers])切入点声明；第 13 章描述了 AspectJ 中可用的不同类型的通知。

5.3 捕获处理异常的对象

问题

在由使用handler(TypePattern)切入点捕捉的连接点触发的通知内，你想访问捕捉异常的对象，并将其用在相应的通知内。

解决方案

结合使用this([Type | Identifier])切入点与handler(TypePattern)切入点，将异常处理对象展示为切入点上的标识符，然后可将其传递给相应的通知。

讨论

例5-3显示了如何把异常处理对象MyClass传递给before()通知，作为myExceptionHandlerPointcut切入点上的myObject标识符。

例5-3：访问包含catch块用于处理MyException异常的对象

```
public aspect AccessHandlingObject
{
   pointcut myExceptionHandlerPointcut(MyClass myObject) :
      handler(MyException) &&
      this(myObject);

   // Advice declaration
   before(MyClass myObject) : myExceptionHandlerPointcut(myObject)
   {
      System.out.println(
         "-------------- Aspect Advice Logic ---------------");
      System.out.println(
         "In the advice picked by " + "myExceptionHandlerPointcut()");
      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());

      System.out.println("Exception caught by:" + myObject);

      System.out.println(
      "-------------------------------------------------");
   }
}
```

参考

5.1 节中说明了 `handler(TypePattern)` 切入点的语法；5.1 节中还说明了可以用在 `TypePattern` 中的一些通配符变体；11.1 节中解释了 `this([Type | Identifier])` 切入点声明，它使用 12.2 节中说明的逻辑 AND（`&&`）来结合切入点逻辑；第 13 章描述了 AspectJ 中可用的不同类型的通知。

第6章

捕获通知上的连接点

6.0 简介

在AspectJ中，方面是类似于类的一流的语言构造。类及其业务逻辑可以被方面通知，因此方面本身也可以被通知。尽管方面可以包含方法及其他候选连接点，仍然有一种特定于方面的构造，它需要自身的机制来捕获其连接点。这种构造就是通知。

本章论及使用切入点定义，它只牵涉到出现在通知内的捕获连接点。AspectJ为此提供了单独的`adviceexecution()`切入点。

本章首先介绍了`adviceexecution()`切入点的语法及其应用的一个简单示例，然后说明`adviceexecution()`切入点如何能够提供一种特别有效的方法用于处理如下情形：其中的通知及其调用的行为需要排除在被应用程序中的其他方面通知之外。

本章最后说明了如何使用AspectJ实现通知的一些有趣特征，来访问触发通知的原始连接点，反过来又使用`adviceexecution()`切入点来通知这些连接点。

`adviceexecution()`切入点是AspectJ开发人员工具箱中增加的一种相当新的工具，本章说明了如何协同使用它和方面。

6.1 捕获何时执行通知

问题

你想捕获何时执行任何一份通知。

解决方案

使用 adviceexecution()切入点。adviceexecution()切入点的语法如下：

```
pointcut <pointcut name>() : adviceexecution();
```

讨论

adviceexecution()切入点用于捕获在应用程序内执行任何通知的连接点。例6-1使用adviceexecution()切入点在其他通知开始执行的同时应用通知。

例 6-1：使用 adviceexecution()切入点应用通知

```
public aspect AdviceExecutionRecipe
{
   /*
       Specifies calling advice whenever advice is executed
   */
   pointcut adviceExecutionPointcut() : adviceexecution();

   // Advice declaration
   before() : adviceExecutionPointcut()
      && !within(AdviceExecutionRecipe +)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println("In the advice picked by ExecutionRecipe");
      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

图 6-1 显示了如何把 adviceexecution()切入点应用于简单的类。

参考

6.3节中介绍了一种方法，它可以防止结合使用adviceexecution()切入点和NOT（!）运算符来调用方面中的代码；第9章讨论了基于切入点的作用域；12.2节说明了使用逻辑AND（&&）来结合切入点逻辑；12.4节说明了一元NOT（!）运算符；第13章描述了AspectJ中可用的不同类型的通知。

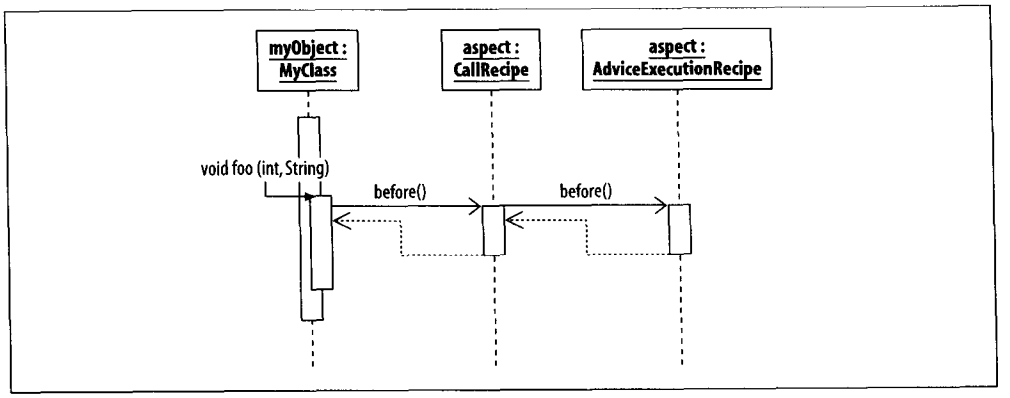

图 6-1：如何应用 adviceexecution()切入点

6.2 排除作为通知执行结果的连接点

问题

你想忽略在通知块内直接触发或者作为通知执行结果的所有连接点。当你想忽略通知块触发的任何事情，并且只想关注常规应用程序的代码业务类逻辑的控制流程内的连接点时，这是有用的。

解决方案

把 `adviceexecution()` 切入点声明提供为 `cflow(Pointcut)` 切入点的参数。使用一元 NOT（`!`）运算符排除被 `cflow(Pointcut)` 切入点捕获的连接点。

讨论

忽略直接出现在通知执行内的连接点以及那些可能被间接触发的连接点有时是有用的。如果只用一元运算符 NOT（`!`）使用 `within()` 切入点，就可以把直接出现在特定方面内的所有连接点排除在外。不过，由那些方面内的通知间接触发的任何连接点仍会被捕获。

可以通过修改 4.1 节中的 `CallRecipe` 方面，来演示这个问题的一个例子，如例 6-2 所示。`before()` 通知已经被增强成调用 `AnotherClass` 对象上的 `foo()` 方法。

例 6-2：从某个通知内调用 AnotherClass 类上的 foo()方法

```
public aspect CallRecipe
{
```

```
    /*
    Specifies calling advice whenever a method
    matching the following rules gets called:

    Class Name: MyClass
    Method Name: foo
    Method Return Type: * (any return type)
    Method Parameters: an int followed by a String
    */
    pointcut callPointCut() : call(void MyClass.foo(int, String));

    // Advice declaration
    before() : callPointCut()
    {

        System.out.println(
            "------------------- Aspect Advice Logic --------------------");
        System.out.println("In the advice attached to the call point cut");

        System.out.println(
                  "Signature: " + thisJoinPoint.getStaticPart().getSignature());
        System.out.println(
              "Source Line: "
              + thisJoinPoint.getStaticPart().getSourceLocation());

        AnotherClass anotherClass = new AnotherClass();
        anotherClass.foo();

        System.out.println(
              "------------------------------------------------------------");

    }
}
```

AnotherClass 类上的 foo() 方法反过来又会调用同一个类上的 bar() 方法：

```
    public class AnotherClass
    {
       public void foo()
       {
          System.out.println("Inside method AnotherClass.foo()");
          this.bar();
       }

       public void bar()
       {
          System.out.println("Inside method AnotherClass.bar()");
       }
    }
```

使用例6-3中所示的 !within(CallRecipe+) 切入点，将正确地排除对 foo() 的调用，但是对 bar() 的间接调用仍会被 tracedCalls() 切入点捕获。

例6-3：错误地使用within(TypePattern)切入点和布尔型NOT（!）运算符防止方面跟踪

```
public aspect TraceCalls
{
   /*
    Specifies calling advice when not within the TraceCalls and CallRecipe aspects
    or any of their subaspects.
   */
   pointcut tracedCalls() : call(* *.*(..)) &&
      !within(TraceCalls+) &&
      !within(CallRecipe+);

   // Advice declaration
   before() : tracedCalls()
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println("In the advice picked by TraceCalls");

      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

AnotherClass类中bar()方法上的调用仍会被before()通知捕获，这是因为它不会直接出现在CallRecipe方面中。不过，它确实会出现在before()通知块的控制流程内，这是由于foo()方法会被通知调用，它反过来又会调用bar()方法。

可以通过添加与adviceexecution()切入点相结合的cflow(Pointcut)切入点，来更改traceCalls()切入点，因此，甚至可以把那些在通知块的控制流程内间接出现的连接点排除在外。

```
pointcut tracedCalls() : call(* *.*(..)) &&
   !within(TraceCalls+) &&
   !within(CallRecipe+) &&
   !cflow(adviceexecution());
```

参考

4.1节中介绍了call(Signature)切入点；10.1节中解释了cflow(Pointcut)切入点；12.2节中说明了使用逻辑AND（&&）来结合切入点逻辑；12.4节中说明了一元NOT（!）运算符；第13章描述了AspectJ中可用的不同类型的通知。

6.3 在发出通知时展示原始连接点

问题

在使用adviceexecution()切入点发出一份通知的地方，你想访问触发通知块（它正被发出）的原始连接点。

解决方案

添加JoinPoint标识符到切入点定义中。把JoinPoint标识符作为单一参数绑定到args([Types | Identifiers])切入点声明上。把JoinPoint标识符添加到希望向其展示原始连接点的相应通知中。

讨论

在例6-4中，使用originalJoinPoint标识符从AdviceExecution方面的before()通知访问那个触发通知（它正被发出）的原始连接点。

例6-4：在发出通知时访问原始触发连接点

```
import org.aspectj.lang.JoinPoint;

public aspect AdviceExecutionRecipe
{
    /*
     Specifies calling advice whenever advice is executed
    */
    pointcut adviceExecutionPointcut(JoinPoint originalJoinPoint) :
       adviceexecution() &&
       args(originalJoinPoint) &&
       !within(AdviceExecutionRecipe);

    // Advice declaration
    before(JoinPoint originalJoinPoint) : adviceExecutionPointcut(originalJoinPoint)
    {
       System.out.println(
          "------------------- Aspect Advice Logic --------------------");
       System.out.println("In the advice picked by AdviceExecutionRecipe");
       System.out.println(
          "Signature: "
             + thisJoinPoint.getStaticPart().getSignature());
       System.out.println(
          "Source Line: "
             + thisJoinPoint.getStaticPart().getSourceLocation());

       System.out.println(
             "Advised Advice's Join Point Signature: "
                + originalJoinPoint.getSignature());
```

```
        System.out.println(
            "------------------------------------------------------------");
        }
}
```

注意： 在AspectJ中，有一个隐式的JoinPoint对象，它会传递给每个被调用的通知块。单一的JoinPoint参数通常会透明地传递给通知，并且变成thisJoinPoint引用的值。为了填充thisJoinPoint引用，每个通知块（当它被AspectJ实现时）必须包含单一的JoinPoint参数作为其签名的一部分。

参考

6.1节中说明了adviceexecution()切入点的语法；9.1节中描述了within(TypePattern)切入点；11.3节中解释了args([Types | Identifiers])切入点声明；12.2节中说明了使用逻辑AND（&&）来结合切入点逻辑；12.4节中说明了一元NOT（!）运算符；第13章描述了AspectJ中可用的不同类型的通知。

第7章

捕获类和对象构造上的连接点

7.0 简介

在可以在应用程序内使用Java类和对象之前，它们要经过4个初始化和构造阶段，忽略通过类加载器把类实际地加载进Java虚拟机（JVM）中。

这个过程的第一步是使用new关键字调用构造函数。如果类以前没有被应用程序使用过，这个调用就会触发正在构造对象的类的静态类初始化。静态类初始化是对类本身进行初始化，使得会正确地构造它的所有静态变量和方法。

在静态地初始化了类之后，并且在可以执行构造函数方法之前，必须初始化对象本身。对象初始化首先通过执行对象的所有超类构造函数，来构造对象的继承层次结构。最后，一旦超类构造函数成功地返回，就会把初始化的对象全都设置成完成其构造函数方法的执行。

AspectJ为类和对象的构造和初始化中涉及的每个阶段提供了一个特定的切入点，本章将逐步讲解每个切入点的语法和关键特征。

7.1 捕获对构造函数的调用

问题

你想捕获何时调用与特定签名匹配的构造函数。

解决方案

使用call(Signature)切入点，它带有额外的new关键字作为签名的一部分。使用与构造函数有关的call(Signature)切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
            call(<optional modifier> <class>.new(<parameter types>));
```

讨论

在用于捕获对构造函数的调用时，call(Signature)切入点具有3个关键特征：

1. 在把一个类实例化成一个对象时，具有new关键字的call(Signature)切入点会捕获连接点。
2. 通过使用around()形式的通知，call(Signature)切入点可以在Java的正常继承规则的限制下重写返回对象的类型。
3. 编译器不会检查指定的Signature是否对应于实际的构造函数。

例7-1说明了如何使用call(Signature)切入点来捕获对MyClass类上的构造函数的调用，该构造函数带有一个int和String作为参数。

例7-1：在调用构造函数时使用call(Signature)切入点来捕获连接点

```
public aspect CallNewRecipe
{
   /*
       Specifies calling advice when any constructor is called
       that meets the following signature rules:

       Class Name: MyClass
       Method Name: new (This is a keyword indicating the constructor call)
       Method Parameters: int, String
   */
   pointcut myClassConstructorWithIntAndStringPointcut() :
               call(MyClass.new (int, String));

   // Advice declaration
   before() : myClassConstructorWithIntAndStringPointcut()
   {
      System.out.println(
         "-------------- Aspect Advice Logic ---------------");
      System.out.println(
         "In the advice picked by "
            + "myClassConstructorWithIntAndOthersPointcut()");
      System.out.println(
         "The current type of object under construction is: ");
      System.out.println(thisJoinPoint.getThis());
```

```
        System.out.println(
            "Signature: "
                + thisJoinPoint.getSignature());
        System.out.println(
            "Source Line: "
                + thisJoinPoint.getSourceLocation());
        System.out.println(
            "-----------------------------------------------");
    }
}
```

图 7-1 说明了如何应用 call(Signature)切入点。

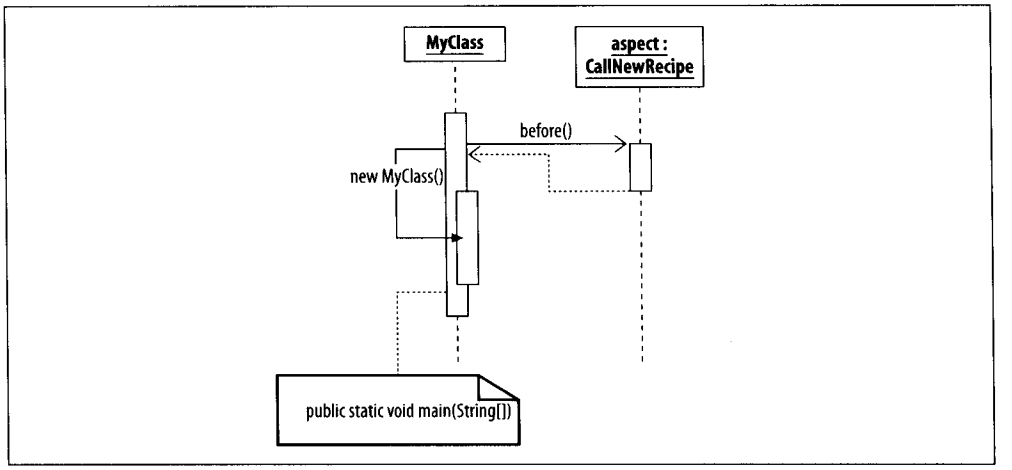

图 7-1：call(Signature)切入点如何应用于构造函数

参考

4.1 节和 4.4 节中的 execution(Signature)切入点和 call(Signature)切入点分别用于把通知附加到方法而不是构造函数上；4.1 节说明了可用在 Signature 中的一些通配符变体；第 13 章论及在被不同的切入点选择时，可供通知使用的环境；20.2 节更详细地描述了 around()通知的使用，特别论及了重写构造函数来返回不同类型的对象。

7.2 在执行构造函数时捕获它

问题

你想在执行构造函数时，捕获与特定签名匹配的构造函数。

解决方案

使用execution(Signature)切入点，它带有额外的new关键字作为签名的一部分。使用与构造函数有关的execution(Signature)切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
            execution(<optional modifier> <class>.new(<parameter types>));
```

讨论

在用于捕获构造函数的执行时，execution(Signature)切入点具有3个关键特征：

1. 在执行类的构造函数时，具有new关键字的execution(Signature)切入点会触发连接点。
2. 不能在调用类的构造函数之前那一刻触发连接点。这会阻止重写返回的对象。
3. 可以使用around()通知来重写构造函数方法的实现，但不能重写正在构造的对象的类型。

例7-2说明了execution(Signature)切入点，它用于捕获MyClass构造函数的执行，这个构造函数带有一个int和一个String作为参数。

例7-2：使用execution(Signature)切入点来捕获特定构造函数内的连接点

```
public aspect ExecutionNewRecipe
{
   /*
       Specifies calling advice when any constructor executes
       that meets the following signature rules:

       Class Name: MyClass
       Method Name: new (This is a keyword indicating the constructor call)
       Method Parameters: int, String
   */
   pointcut myClassConstructorWithIntAndStringPointcut() :
            execution(MyClass.new (int, String));

   // Advice declaration
   before() : myClassConstructorWithIntAndStringPointcut()
   {
      System.out.println(
         "---------------- Aspect Advice Logic -----------------");
      System.out.println(
         "In the advice picked by "
            + "myClassConstructorWithIntAndOthersPointcut()");
      System.out.println(
         "The current type of object under construction is: ");
      System.out.println(thisJoinPoint.getThis().getClass());
```

```
        System.out.println(
            "Signature: " + thisJoinPoint.getSignature());
        System.out.println(
            "Source Line: " + thisJoinPoint.getSourceLocation());
        System.out.println(
            "-----------------------------------------------------");
    }
}
```

图 7-2 说明了如何把 execution(Signature)切入点应用于构造函数。

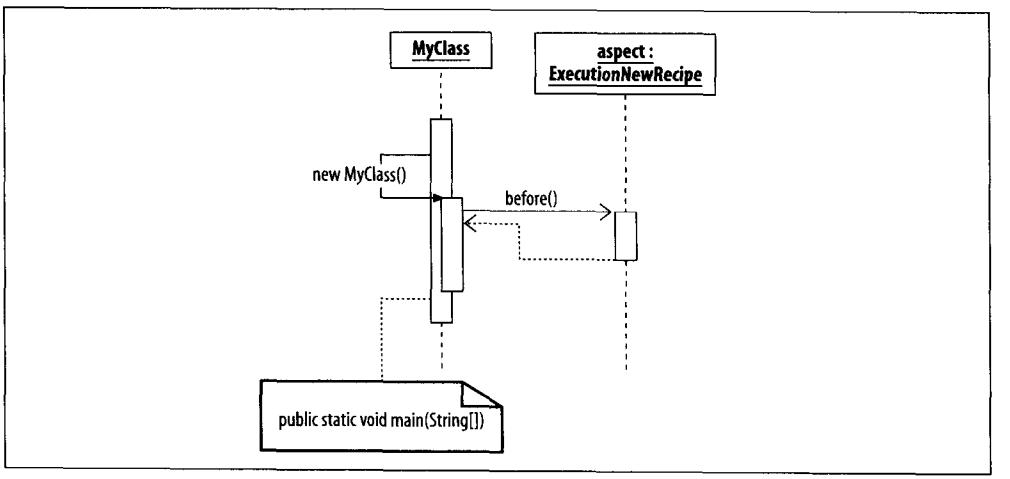

图 7-2：如何把 execution(Signature)切入点应用于构造函数

参考

4.1节说明了可用在Signature中的一些通配符变体，描述了如何把call(Signature)切入点应用于构造函数，它类似于本节中介绍的切入点，只不过 call(Signature)切入点可以重写正在构造的对象；4.1 节和4.4 节分别说明了如何定义 call(Signature)切入点和 execution(Signature)切入点从常规方法捕获连接点；第 13 章描述了 AspectJ 中可用的不同类型的通知。

7.3 捕获何时初始化对象

问题

你想捕获何时初始化对象，初始化对象是通过调用与特定签名匹配的构造函数来进行的。

解决方案

使用initialization(Signature)切入点。initialization(Signature)切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
            initialization(<optional modifier> <class>.new(<parameter types>));
```

讨论

initialization(Signature)切入点具有5个关键特征：

1. initialization(Signature)切入点必须包含new关键字。
2. initialization(Signature)切入点捕获连接点发生在任何超类的初始化之后，以及从构造函数方法返回之前。
3. Signature必须解析成特定类的构造函数，而不是一个简单的方法。
4. initialization(Signature)切入点提供了编译时检查，用于检查构造函数是否正在被引用。
5. 由于AspectJ编译器中的编译器限制，当与around()通知关联时，不能使用initialization(Signature)切入点。

例7-3说明了当构造函数具有签名MyClass.new(int,*)时，捕获MyClass对象初始化的initialization(Signature)切入点。

例7-3：在执行构造函数时，使用initialization(Signature)切入点来捕获连接点

```
public aspect InitializationRecipe
{
   /*
       Specifies calling advice when any object
       initializes using a constructor
       that meets the following signature rules:

       Class Name: MyClass
       Method Name: new (This is a keyword indicating the
       constructor call)
       Method Parameters: int and any others
   */
   pointcut myClassObjectInitializationWithIntAndOthersPointcut() :
                initialization(MyClass.new (int, *));
   // Advice declaration
   before() : myClassObjectInitializationWithIntAndOthersPointcut()
   {
      System.out.println(
         "-------------- Aspect Advice Logic ---------------");
```

```
        System.out.println(
            "In the advice picked by "
                + "myClassObjectInitializationWithIntAndOthersPointcut()");
        System.out.println(
            "The current type of object under construction is: ");
        System.out.println(thisJoinPoint.getThis().getClass());
        System.out.println(
            "Signature: "
                + thisJoinPoint.getStaticPart().getSignature());
        System.out.println(
            "Source Line: "
                + thisJoinPoint.getStaticPart().getSourceLocation());
        System.out.println(
            "-------------------------------------------------");
    }
}
```

图7-3说明了如何把initialization(Signature)切入点应用于选择构造函数上的连接点。

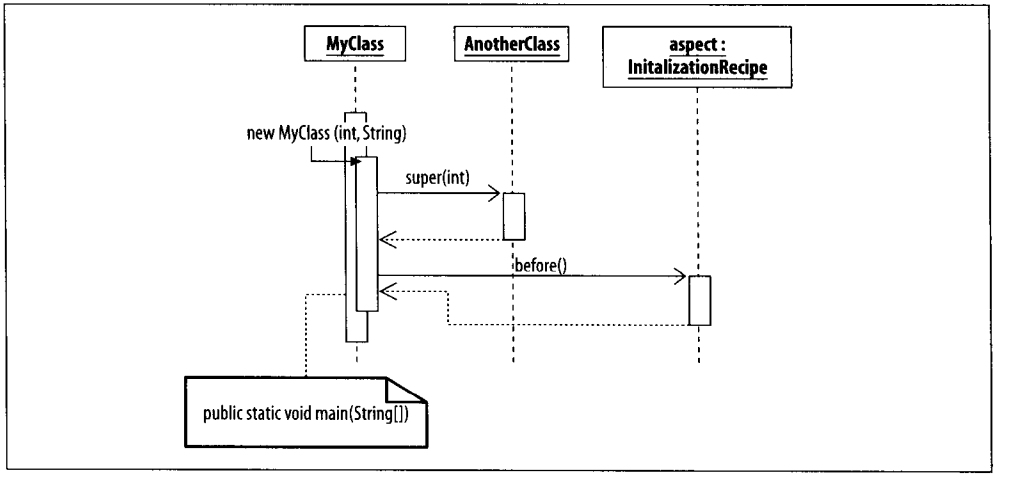

图7-3：如何应用initialization(Signature)切入点

与execution(Signature)切入点相比，使用initialization(Signature)切入点的最大优点是：它提供了编译时检查，以确保签名实际上指定了一个构造函数。

参考

4.1节说明了可用在Signature中的一些通配符变体；7.2节说明了execution(Signature)切入点，可将其与本节中描述的类似切入点作比较；第13章描述了AspectJ中可用的不同类型的通知。

7.4 捕获何时将要初始化一个对象

问题

你想捕获何时将要使用与特定签名匹配的构造函数初始化一个对象。

解决方案

使用preinitialization(Signature)切入点。preinitialization(Signature)切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
            preinitialization(<optional modifier> <class>.new(<parameter types>));
```

讨论

preinitialization(Signature)切入点具有5个关键特征：

1. preinitialization(Signature)切入点必须包含new关键字。
2. preinitialization(Signature)切入点捕获连接点发生在进入捕捉构造函数之后，以及调用任何超类构造函数之前。
3. Signature必须解析成一个构造函数。
4. preinitialization(Signature)切入点提供了编译时检查，用于检查构造函数是否正在被引用。
5. 由于AspectJ编译器中的编译器限制，当与around()通知关联时，不能使用preinitialization(Signature)切入点。

例7-4说明了在使用具有int和String作为参数的MyClass构造函数初始化对象之前，捕获连接点的preinitialization(Signature)切入点。

例7-4：在执行特定的构造函数之前，使用preinitialization(Signature)切入点捕获连接点

```
public aspect PreInitializationRecipe
{
   /*
       Specifies calling advice just before an object initializes
       using a constructor that meets the following signature rules:

       Class Name: MyClass
       Method Name: new (This is a keyword indicating the constructor call)
       Method Parameters: an int followed by a String
   */
```

```
    pointcut myClassIntStringObjectPreInitializationPointcut() :
              preinitialization(MyClass.new (int, String));

    // Advice declaration
    before(
       int number,
       String name) : myClassIntStringObjectPreInitializationPointcut()
       && args(number, name)
    {
       System.out.println(
          "-------------- Aspect Advice Logic ---------------");
       System.out.println(
          "In the advice picked by "
             + "anyMyClassObjectInitializationPointcut()");
       System.out.println(
          "The current type of object under construction is: ");
       System.out.println(thisJoinPoint.getThis());
       System.out.println(
          "The values passed in were: " + number + ", " + name);
       System.out.println(
          "Signature: "
             + thisJoinPoint.getStaticPart().getSignature());
       System.out.println(
          "Source Line: "
             + thisJoinPoint.getStaticPart().getSourceLocation());
       System.out.println(
       "--------------------------------------------------");
    }
}
```

图 7-4 说明了在初始化对象之前，如何把 preinitialization(Signature) 切入点应用于选择连接点。

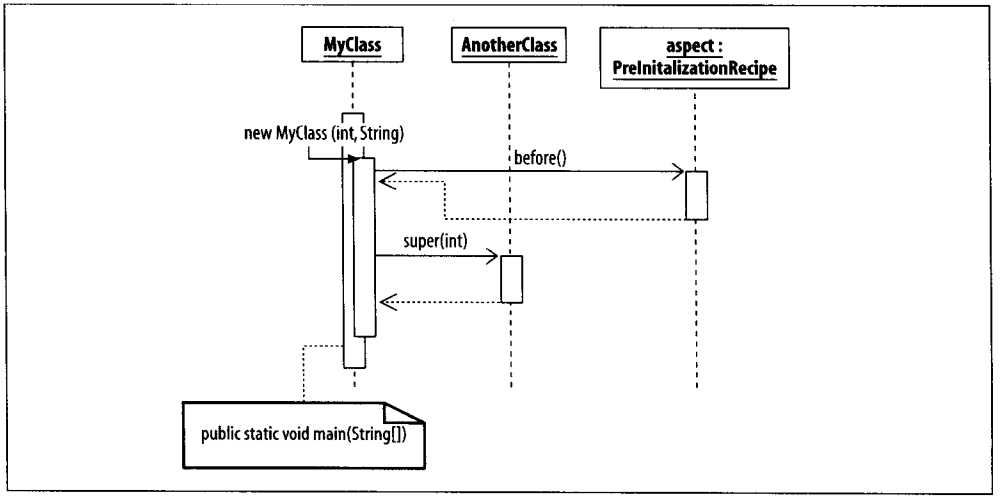

图 7-4：如何应用 preinitialization(Signature)切入点

参考

4.1 节说明了可在Signature 中使用的一些通配符变体；6.1 节中介绍了adviceexecution()切入点；第 13 章描述了 AspectJ 中可用的不同类型的通知。

7.5 捕获何时初始化类

问题

你想捕获何时初始化类。

解决方案

使用staticinitialization(TypePattern)切入点。staticinitialization(TypePattern)切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
            staticinitialization(<class>);
```

讨论

staticinitialization(TypePattern)切入点具有两个关键特征：

1. 对可供staticinitialization(TypePattern)切入点所选通知使用的环境有一些限制。没有父对象触发静态初始化；因此，没有this引用。也不涉及实例对象，因此，没有目标引用。
2. TypePattern可以包含通配符，用于选择一系列不同的类。

例 7-5 说明了 staticinitialization(TypePattern)切入点，它用于捕获静态初始化 MyClass 类中的连接点。

例 7-5：使用 staticinitialization(TypePattern)切入点，来捕获静态初始化特定类上的连接点

```
public aspect StaticInitializationRecipe
{
   /*
       Specifies calling advice when a class is initialized
       that meets the following type pattern rules:

       Class Name: MyClass
   */
   pointcut myClassStaticInitializationPointcut() : staticinitialization(MyClass);
```

```
    // Advice declaration
    before() : myClassStaticInitializationPointcut()
    {
        System.out.println(
            "-------------- Aspect Advice Logic ---------------");
        System.out.println(
            "In the advice picked by "
                + "myClassStaticInitializationPointcut()");
        System.out.println(
            "Join Point Kind: "
                + thisJoinPoint.getStaticPart().getKind());
        System.out.println(
            "Signature: "
                + thisJoinPoint.getStaticPart().getSignature());
        System.out.println(
            "Source Line: "
                + thisJoinPoint.getStaticPart().getSourceLocation());
        System.out.println(
            "-------------------------------------------------");
    }
}
```

图 7-5 说明了如何应用 staticinitialization(TypePattern) 切入点。

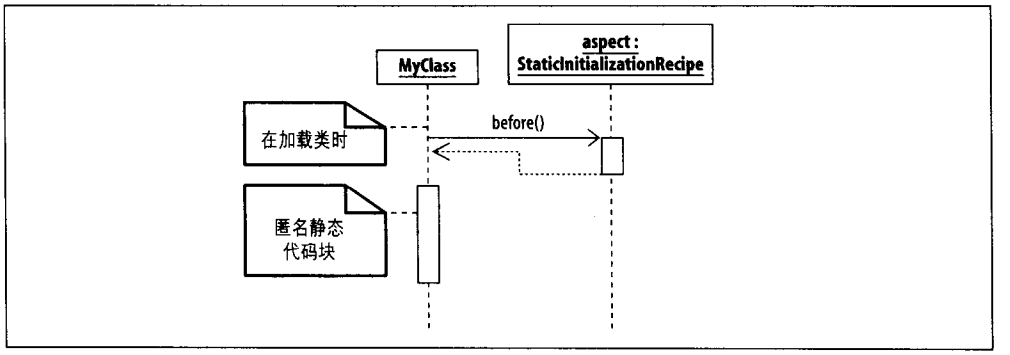

图 7-5：如何应用 staticinitialization(TypePattern)切入点

参考

5.1 节说明了可用在 TypePattern 中的一些通配符变体；第 13 章描述了 AspectJ 中可用的不同类型的通知，包括它们展示的不同形式的关联环境。

第8章

捕获属性上的连接点

8.0 简介

AspectJ开发人员工具箱的一个稍有争议的特性是：能够监视可能发生在类属性上的任何访问或修改。AspectJ以get(Signature)和set(Signature)切入点的形式提供了这种能力，它们是本章的重点。

实践过传统面向对象技术的任何人在听到这一点时都会有点担心，因为它会有效地破坏封装性，特别是当把被监视的属性声明为protected或private时。一个良好的建议是：在决定方面需要直接访问类的内部细节之前要仔细考虑一下，以避免方面与类之间不必要的紧密耦合。

不过，有时横切关注点需要与类之间的这种级别的紧密性，以有效地应用方面。明智地使用本章中介绍的get(Signature)和set(Signature)切入点，可以提供一种强大的手段来通知类，但是必须小心地使用它们，以确保不会使类和方面不必要地脆弱。

8.1 捕获何时访问对象的属性

问题

你想捕获何时访问对象的属性。

解决方案

使用get(Signature)切入点。get(Signature)切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
            get(<optional modifier> <type> <class>.<field>);
```

讨论

get(Signature)具有4个关键特征：

1. get(Signature)切入点会触发直接在其中（而不仅仅是在访问器方法的调用上）访问属性的通知。
2. get(Signature)切入点不能捕获对静态属性的访问，尽管从AspectJ的语法角度讲以这种方式定义切入点是完全合法的。
3. Signature必须解析成特定类的属性。
4. Signature可以包含通配符，用于选择不同属性上的一系列连接点。

例8-1说明了get(Signature)切入点，它用于捕获无论何时访问String MyClass.name属性时所遇到的连接点。

例8-1：使用get(Signature)切入点来捕获对属性的访问

```
public aspect GetRecipe
{
   /*
    Specifies calling advice whenever an attribute
    matching the following rules is accessed:

    Type: String
    Class Name: MyClass
    Attribute Name: name
   */
   pointcut getNamePointcut() : get(String MyClass.name);

   // Advice declaration
   before() : getNamePointcut()
   {
      System.out.println(
         "-------------- Aspect Advice Logic ---------------");
      System.out.println(
         "In the advice picked by " + "getNamePointcut()");
      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "-------------------------------------------------");
   }
}
```

图8-1显示了如何把get(Signature)切入点应用于简单的类。

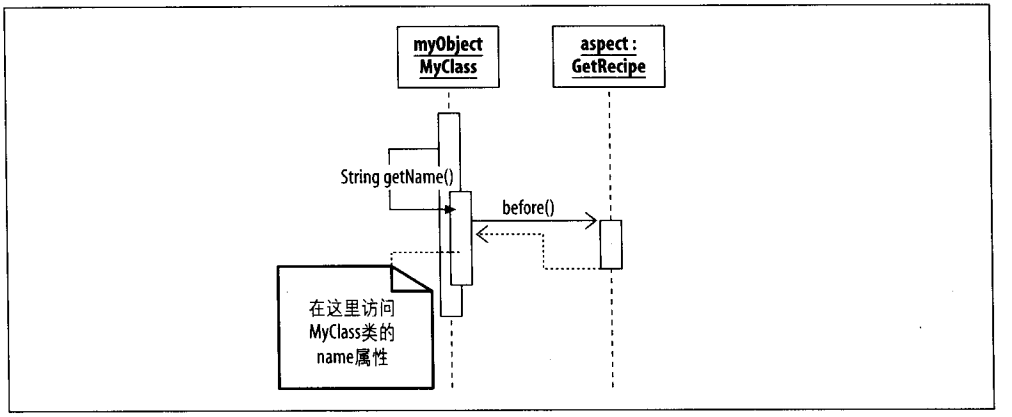

图 8-1：如何应用 get(Signature)切入点

你可能期待类使用 static 和 final 关键字定义了一个常量属性，这样，在访问这个常量时你就能使用get(Signature)切入点来捕获。例8-2中的getConstrantPointcut()切入点仅为此目的在访问 MyClass.CONSTANT 属性时而试图进行捕获。

例 8-2：在访问常量时使用 get(Signature)切入点捕获

```
public aspect GetConstant
{
   /*
    Specifies calling advice whenever an attribute
    matching the following rules is accessed.

    Type: String
    Class Name: MyClass
    Attribute Name: CONSTANT
   */
   pointcut getConstantPointcut() : get(public static final String MyClass.CONSTANT);

   // Advice declaration
   before() : getConstantPointcut()
   {
      System.out.println(
         "-------------- Aspect Advice Logic ---------------");
      System.out.println(
         "In the advice picked by " + "getConstantPointcut()");
      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
```

```
        System.out.println(
            "-----------------------------------------------");
    }
}
```

事实上，getConstantPointcut()切入点不会捕捉这种形式的常量访问，即使这种AspectJ语法是可接受的。尽管这是有效的get(Signature)切入点声明，但是与常规的变量属性不同，Java编译器会“内联”static final属性；因此，它们存在的形式不合适于切入点访问。

警告： 在使用get(Signature)切入点时要谨慎小心，因为它会破坏可能被声明为private的属性的封装性。破坏封装性可能导致脆弱的软件。由于你基于变量名称指定切入点，所以必须记住：这可能给设计增加有害的更紧密的耦合。

参考

4.1节说明了可用在Signature中的一些通配符变体；第13章描述了AspectJ中可用的不同类型的通知；9.1节中描述了within(TypePattern)切入点；12.4节中描述了NOT(!)运算符。

8.2 捕获访问的字段值

问题

你想捕获访问的字段值，使得可以在相应的通知中使用它。

解决方案

使用after() returning(<ReturnValue>)形式的通知，它在声明的returning()部分中带有一个标识符，用于包含访问过的值。

讨论

例8-3说明了在访问MyClass.name时，如何把MyClass.name属性的值传递给触发的after() returning(<ReturnValue>)通知。

例8-3：在访问字段时，访问其返回的值

```
public aspect CaptureAccessedFieldValue
{
    pointcut getNamePointcut() : get(String MyClass.name);
```

```
    // Advice declaration
    after() returning(String value) : getNamePointcut()
    {
        System.out.println(
            "-------------- Aspect Advice Logic ---------------");
        System.out.println(
            "In the advice picked by " + "getNamePointcut()");
        System.out.println(
            "Signature: "
                + thisJoinPoint.getStaticPart().getSignature());
        System.out.println(
            "Source Line: "
                + thisJoinPoint.getStaticPart().getSourceLocation());

        System.out.println("Value being accessed is " + value);

        System.out.println(
            "-------------------------------------------------");
    }
}
```

参考

4.1节说明了可用在Signature中的一些通配符变体；8.1节研究了get(Signature)切入点的语法；13.6节说明了通知的after() returning形式。

8.3 捕获何时修改对象的字段

问题

你想捕获何时修改对象的字段。

解决方案

使用set(Signature)切入点。set(Signature)切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
            set(<optional modifier> <type> <class>.<field>);
```

讨论

set(Signature)切入点具有4个关键特征：

1. set(Signature)切入点在修改字段时触发。
2. set(Signature)切入点不能捕获静态字段的修改，尽管从AspectJ的语法角度讲以这种方式定义切入点是完全合法的。

3. Signature 必须解析成特定类的属性。

4. Signature 可以包含通配符，以选择不同属性上的一系列连接点。

例8-4说明了set(Signature)切入点，它用于捕获无论何时修改String MyClass.name属性时所遇到的连接点。

例 8-4：在修改属性时，使用 set(Signature)切入点来捕获连接点

```
public aspect SetRecipe
{
   /*
       Specifies calling advice whenever an attribute
       matching the following rules is modified:

       Type: String
       Class Name: MyClass
       Attribute Name: name
   */
   pointcut setNamePointcut() : set(String MyClass.name);

   // Advice declaration
   before() : setNamePointcut() && !within(SetRecipe +)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice picked by " + "setNamePointcut()");
      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

图 8-2 说明了如何把 set(Signature)切入点应用于简单的类。

参考

4.1 节说明了可用在 Signature 中的一些通配符变体；第 13 章描述了 AspectJ 中可用的不同类型的通知；9.1 节中描述了 within(TypePattern)切入点；12.4 节中描述了 NOT（!）运算符。

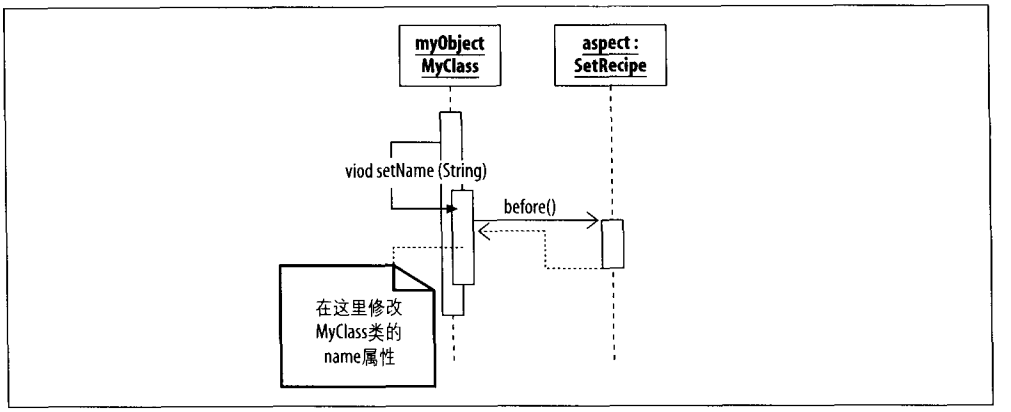

图 8-2：如何应用 set(Signature)切入点

8.4 在修改字段时捕获它的值

问题

你想在修改字段之后捕获它的值，使得可将其用在相应的通知中。

解决方案

结合使用 `args([Types | Identifiers])` 切入点和 `set(Signature)` 切入点，展示字段的新值，该字段被设置成切入点上的标识符，可将其传递给相应的通知。

讨论

例 8-5 说明了如何把为 `MyClass.name` 属性设置的新值传递给 `before()` 通知。

例 8-5：访问被设置的字段的新值

```
public aspect CaptureModifiedFieldValue
{
   pointcut setNamePointcut(String newValue) : set(String MyClass.name)
                                && args(newValue);
   // Advice declaration
   before(String newValue) : setNamePointcut(newValue)
   {
      System.out.println(
         "----   ------------ Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice picked by " + "setNamePointcut()");
```

```
        System.out.println(
            "Signature: "
                + thisJoinPoint.getStaticPart().getSignature());
        System.out.println(
            "Source Line: "
                + thisJoinPoint.getStaticPart().getSourceLocation());

        System.out.println("Field Value set to: " + newValue);

        System.out.println(
            "------------------------------------------------------------");
    }
}
```

参考

4.1节说明了可用在Signature中的一些通配符变体；8.4节研究了set(Signature)切入点的语法；11.3节解释了args([Types | Identifiers])切入点声明；使用12.2节中说明的逻辑AND（&&）结合切入点逻辑；第13章描述了AspectJ中可用的不同类型的通知。

第 9 章

捕获程序作用域内的连接点

9.0 简介

切入点定义设计中的常用方法是：基于关注的程序作用域，限制捕获连接点的范围。这可以让你即时控制在进一步的切入点定义中将会涉及哪些连接点。

本章描述了切入点集合，它们允许显式控制基于其程序作用域而捕获的连接点。这些切入点允许指定一个直接通往包作用域的方法。

本章中的切入点相当容易理解，并且他们是 AspectJ 中一些最常用的元素。例如，广泛使用的 `within(TypePattern)` 切入点（在 5.1 节讨论过）将以 `!within(%THIS_ASPECT%)` 形式使用它。这个 AspectJ 术语限制了当前方面之外的每个连接点的作用域，从而可以防止通知触发对相同通知块的递归调用，并导致一个无限循环。本章中的概念是 AspectJ 编程的基础，并且构成了本章后面详述的许多操作的骨架。

9.1 捕获特定类中的所有连接点

问题

你想捕获特定类中的所有连接点。

解决方案

使用 `within(TypePattern)` 切入点，利用 `TypePattern` 指定特定类类型模式。`within(TypePattern)` 切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) : within(<class>);
```

讨论

within(TypePattern)切入点具有3个关键特征：

1. within(TypePattern)切入点捕获指定类作用域中的所有连接点。
2. within(TypePattern)切入点极少孤立使用。相反，它通常与其他切入点结合使用，用于减少将触发附带通知的连接点。
3. TypePattern可以包含通配符，用于选择不同类上的一系列连接点。

例9-1说明了within(TypePattern)切入点，它用于捕获MyClass类中的所有连接点。

例9-1：使用within(TypePattern)切入点捕获特定类中的连接点

```
public aspect WithinClassRecipe
{

   /*
       Specifies calling advice on any join point encountered within
       the defined scope:

       Scope: MyClass
       */
   pointcut withinMyClass() : within(MyClass);

   // Advice declaration
   before() : withinMyClass() && !within(WithinClassRecipe +)
   {
      System.out.println(
         "-------------- Aspect Advice Logic ---------------");
      System.out.println(
         "In the advice picked by " + "withinMyClass()");
      System.out.println(
         "Join Point Kind: " + thisJoinPoint.getKind());
      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "--------------------------------------------------");
   }
}
```

图9-1说明了如何把within(TypePattern)切入点应用于单个类。

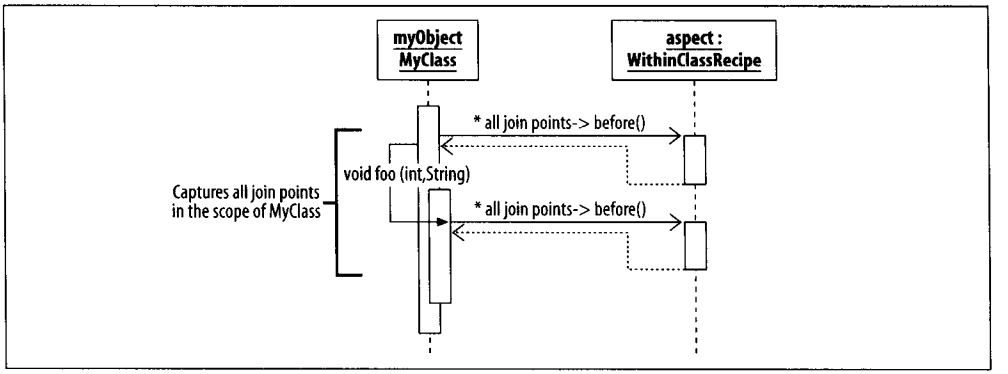

图 9-1：如何应用 within(TypePattern)切入点

参考

5.1 节说明了可用在 TypePattern 中的一些通配符变体；参见第 12 章中的秘诀，了解结合使用切入点的机制示例；通常使用一元 NOT（！）运算符（如 12.4 节所述）对 within(TypePattern) 切入点执行求反运算，以从织入的作用域中删除一个方面；第 13 章包含一些秘诀，描述了 AspectJ 中可用的不同类型的通知。

9.2 捕获特定包中的所有连接点

问题

你想捕获类中的所有连接点，这些类反过来又在特定的包作用域内。

解决方案

使用 within(TypePattern) 切入点，它利用 TypePattern 指定一个包。

讨论

within(TypePattern) 切入点提供了一种有用的手段，它使用通配符在出现在每个类中的连接点中指定一个关注。可以在 TypePattern 中使用合适的通配符，从切入点逻辑的余下部分中包含或排除连接点的整个包。

例 9-2 说明了 within(TypePattern) 切入点，它用于捕获 packageA 包内的所有连接点。

例 9-2：使用 within(TypePattern)切入点来捕获特定包内的连接点

```
public aspect WithinPackageRecipe
{
   /*
       Specifies calling advice on any join point is encountered within
       the defined scope:

       Scope: packageA
   */
   pointcut withinPackageA() : within(packageA.*);

   // Advice declaration
   before() : withinPackageA() && !within(WithinPackageRecipe +)
   {
      System.out.println("-------------- Aspect Advice Logic ---------------");
      System.out.println("In the advice picked by " + "withinPackageA()");
      System.out.println("Join Point Kind: " + thisJoinPoint.getKind());
      System.out.println("Signature: " +
          thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
          "Source Line: " + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println("-------------------------------------------------");
   }
}
```

6.1 节中列出了 within(TypePattern)切入点的关键特征。

参考

5.1 节说明了可用在 TypePattern 中的一些通配符变体；9.1 节说明了更传统地使用 within(TypePattern)切入点定义，来捕获特定类中的连接点；参见第12章中的秘诀，了解结合使用切入点的机制示例；通常使用一元NOT（!）运算符（如12.4节所述）对 within(TypePattern)切入点执行求反运算，以从织入的作用域中删除一个方面。

9.3 捕获特定方法内的所有连接点

问题

你想捕获特定方法内的所有连接点。

解决方案

使用withincode(Signature)切入点。withincode(Signature)切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
            withincode(<modifier> <class>.<method>(<parameter types>));
```

讨论

withincode(Signature)切入点有3个关键特征：

1. withincode(Signature)切入点指定了特定方法的本地作用域内的所有连接点。
2. withincode(Signature)切入点极少孤立使用。相反，它通常与其他切入点结合使用，用于减少将触发附带通知的连接点。
3. Signature可以包含通配符，用于选择跨不同类的不同方法上的一系列连接点。

例9-3说明了withincode(Signature)切入点，它用于捕获* MyClass.foo(int,String)方法的作用域内的所有连接点。

例9-3：使用withincode(Signature)切入点来捕获方法作用域内的所有连接点

```
public aspect WithinMethodRecipe
{
   /*
       Specifies calling advice whenever a method
       matching the following rules gets called:

       Class Name: MyClass
       Method Name: foo
       Method Return Type: * (any return type)
       Method Parameters: an int followed by a String
   */
   pointcut withinFooIntStringAnyReturnPointcut() :
        withincode(* MyClass.foo(int, String));

   // Advice declaration
   before() : withinFooIntStringAnyReturnPointcut()
      && !within(WithinMethodRecipe +)
   {
      System.out.println(
         "-------------- Aspect Advice Logic ---------------");
      System.out.println(
         "In the advice picked by withinFooIntStringAnyReturnPointcut");
      System.out.println(
         "Join Point Kind: "
            + thisJoinPoint.getStaticPart().getKind());
      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "-------------------------------------------------");
   }
}
```

图 9-2 说明了如何应用 `withincode(Signature)` 切入点。

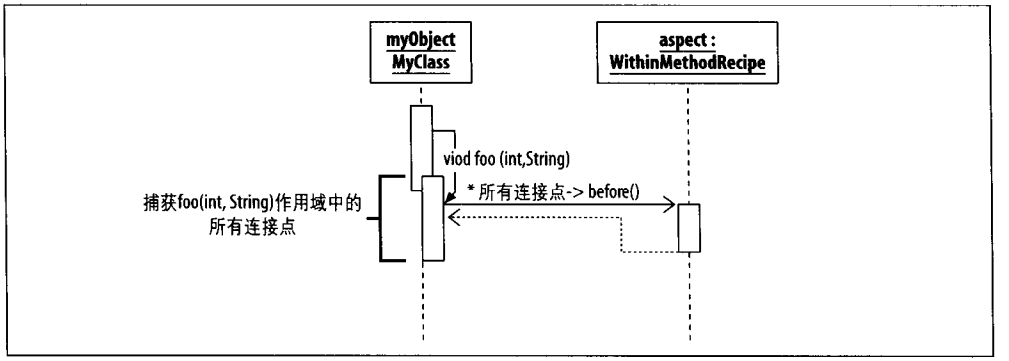

图 9-2：如何应用 withincode(Signature)切入点

参考

4.1 节说明了可用在 `Signature` 中的一些通配符变体；可以使用一元 NOT（!）运算符（如 12.4 节所述）对 `withincode(Signature)` 切入点执行求反运算，以从织入的作用域中删除一个方面；第 13 章描述了 AspectJ 中可用的不同类型的通知。

第 10 章

捕获基于控制流程的连接点

10.0 简介

本章中描述的切入点支持捕获另一个初始连接点作用域或环境内的所有连接点。每个连接点在程序的控制流程中都有一个具体位置，这为通过这里描述的切入点声明捕获的连接点提供了环境。

这最初看起来似乎有些令人糊涂。如果把连接点想象成代码中的一个特定点，它提供了可以定义作用域的标记，那么这些切入点定义就会选择那个作用域。为了理解这些切入点，必须深入研究本章中的秘诀，并尝试执行一些操作。特别值得研究与这些秘诀配套的图，因为它们用于说明在应用切入点的逻辑时，实际上将发生什么。

对于 AspectJ 初学者，这些切入点可能最难以掌握，但是值得坚持学习它们，因为它们提供了一些独特而强大的机制，用于捕获一些连接点，一旦把这些连接点添加到库中，它们通常就是有用的。

10.1 捕获通过初始连接点开始的程序控制流程内的所有连接点

问题

你想捕获在程序控制流程内遇到的所有连接点，这些连接点都在通过单独的切入点选择的开始连接点之后，并且包括该连接点。

解决方案

使用cflow(Pointcut)切入点。cflow(Pointcut)切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) : cflow(<pointcut>)
```

讨论

cflow(Pointcut)切入点具有3个关键特征：

1. cflow(Pointcut)切入点捕获在初始特定的连接点环境内遇到的所有连接点，这个初始连接点是通过另一个切入点选择的。
2. 捕捉的连接点包括初始连接点。
3. 作用域是cflow(pointcut)切入点中重要的鉴别器。这个切入点将捕获通过切入点参数捕获的连接点的控制流程内的所有连接点。

例10-1说明了cflow(Pointcut)如何用于捕获程序控制流程中的所有连接点，这些连接点都在通过callInitialPointcut()切入点捕获的初始连接点之后，并且包括该连接点。

例10-1：使用cflow(Pointcut)切入点捕获特定切入点之后的所有连接点，并且包括该切入点

```
public aspect CFlowRecipe
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets called:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: void
   Method Parameters: an int followed by a String
   */
   pointcut callInitialPointcut() : call(
      void MyClass.foo(int, String));

   /*
    Specifies calling advice whenever a join point is encountered
    including and after the initial join point that triggers the pointcut
    that is specified in the parameter:

    Pointcut Name: callInitialPointcut
   */
   pointcut cflowPointcut() : cflow(callInitialPointcut());
```

```
    // Advice declaration
    before() : cflowPointcut() && !within(CFlowRecipe +)
    {
        System.out.println(
            "------------------- Aspect Advice Logic --------------------");
        System.out.println(
            "In the advice attached to the cflowPointcut point cut");
        System.out.println(
            "Join Point Kind: "
              + thisJoinPoint.getStaticPart().getKind());
        System.out.println(
            "Signature: "
              + thisJoinPoint.getStaticPart().getSignature());
        System.out.println(
            "Source Line: "
              + thisJoinPoint.getStaticPart().getSourceLocation());
        System.out.println(
            "------------------------------------------------------------");
    }
}
```

图 10-1 给出了如何应用 cflow(Pointcut) 切入点的一个更实用的例子。

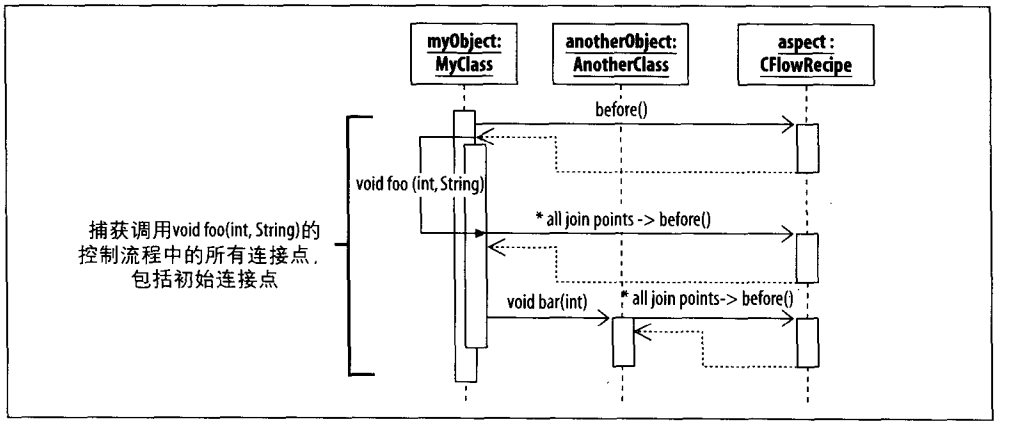

图 10-1：如何应用 cflow(Pointcut)切入点

值得更详细一点地研究 cflow(Pointcut) 做什么。这个特殊的切入点引入了连接点环境的概念。它是指每个连接点具有一个作用域，在这个用途域内，它被看成是执行程序的控制流程的一部分。

在这个控制流程内，任何遇到的连接点都会触发 cflow(Pointcut) 切入点，并调用任何关联的通知。当初始连接点触发指定的切入点参数时，cflow(Pointcut) 切入点会起作用，并触发其关联的通知。然后，将为在初始连接点环境内的控制流程中遇到的每

个连接点调用与cflow(Pointcut)关联的通知。最后，捕捉的连接点集合包括初始连接点本身，这就是这个切入点与cflowbelow(Pointcut)切入点之间的主要区别。

在cflow(Pointcut)的当前实现中，在使用它时，其实现方式会引入大量的系统开销。在可能的地方，并且连接点重用不受影响时，可以考虑使用优先使用withincode(Signature)切入点，而不是cflow(Pointcut)切入点。

参考

Ganesh Sittampalam 等人所著的论文“测量 AspectJ 程序的动态行为（Measuring the Dynamic Behaviour of AspectJ Programs）”（可以在 *http://www.sable.mcgill.ca/publications/papers/#oopsla2004* 上找到它）讨论了一些关于AspectJ实现可能优于手工编码解决方案的性能问题；9.3 节中描述了 withincode(Signature)切入点；10.2 节中研究了 cflowbelow(Pointcut)切入点；第 13 章中提供了关于连接点环境的更多信息；第 12 章中的秘诀描述了用于结合切入点定义的技术。

10.2 捕获程序控制流程内的所有连接点，不包括初始连接点

问题

你想捕获在程序控制流程内遇到的所有连接点，这些连接点都在通过单独的切入点选择的开始连接点之后。

解决方案

使用 cflowbelow(Pointcut)切入点。cflowbelow(Pointcut)切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) : cflowbelow(<pointcut>);
```

讨论

例10-2说明了cflowbelow(Pointcut)切入点，它用于捕获在被callInitialPointcut()切入点捕获的初始连接点之后出现的所有连接点。

例 10-2：使用cflowbelow(Pointcut)切入点捕获位于初始捕获的连接点之后的所有连接点

```
public aspect CFlowBelowRecipe
{
```

```
    /*
    Specifies calling advice whenever a method
    matching the following rules gets called:

    Class Name: MyClass
    Method Name: foo
    Method Return Type: void
    Method Parameters: an int followed by a String
    */
    pointcut callInitialPointcut() : call(
       void MyClass.foo(int, String));

    /*
     Specifies calling advice whenever a join point is encountered
     after but excluding the initial join point that triggers the pointcut
     that is specified in the parameter:

     Pointcut Name: callInitialPointcut
    */
    pointcut cflowPointcut() : cflowbelow(callInitialPointcut());

    // Advice declaration
    before() : cflowPointcut() && !within(CFlowBelowRecipe +)
    {
       System.out.println(
          "------------------- Aspect Advice Logic --------------------");
       System.out.println(
          "In the advice attached to the cflowbelowPointcut point cut");
       System.out.println(
          "Join Point Kind: " + thisJoinPoint.getKind());
       System.out.println(
          "Signature: "
             + thisJoinPoint.getStaticPart().getSignature());
       System.out.println(
          "Source Line: "
             + thisJoinPoint.getStaticPart().getSourceLocation());
       System.out.println(
          "------------------------------------------------------------");
    }
}
```

图 10-2 说明了如何应用 cflowbelow(Pointcut)切入点。

这个秘诀与 10.1 节中介绍的秘诀稍有区别；其区别是实际捕获的连接点数量。cflow(Pointcut)切入点会触发在初始连接点环境内遇到的所有连接点（包括初始连接点）上的通知，而 cflowbelow(Pointcut)切入点则不包括那个初始连接点。

参考

10.1 节描述了 cflow(Pointcut)切入点；第 12 章描述了用于结合切入点定义的技术；第 13 章提供了关于连接点环境的更多信息。

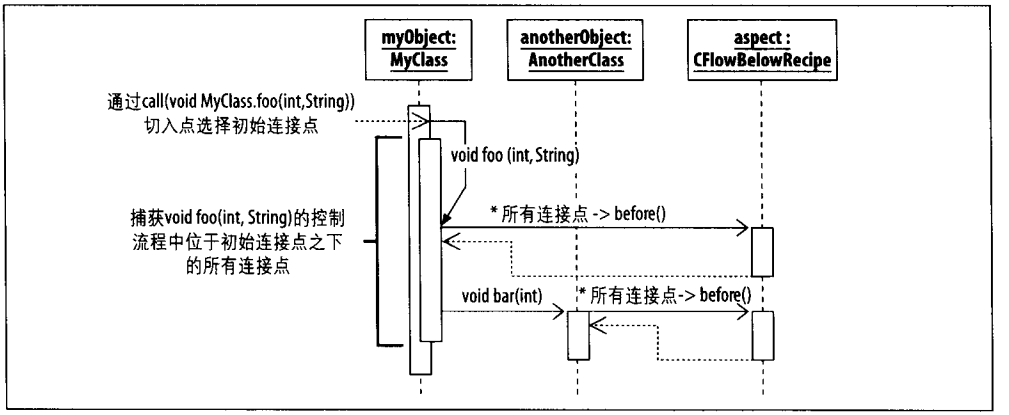

图 10-2：如何应用 cflowbelow(Pointcut)切入点

第 11 章

捕获基于对象类型的连接点

11.0 简介

本章中介绍的切入点允许研究特定连接点处的this引用、可选目标引用或参数的类型。

this([Type | Identifier])研究的是在捕捉的连接点处由 this 引用的对象的类型。连接点的目标（通过target([Type | Identifier])切入点指定）依赖于连接点类型而有所不同，但它常常是在其上调用方法的对象，或者是以某种方式访问的在其中遇到连接点的属性。连接点的*args*（通过 args([Types | Identifiers])切入点指定）就是在捕捉的连接点处可用的参数（如果有的话）。

可以用两种方式之一研究this、target和args引用的类型：使用静态Type指示，或者使用一个 Identifier，它引用一个实际的运行时对象。

Type指示对连接点处的对象类型提供了一个静态编译时评估，并且采用完全限定类名的形式（例如，com.oreilly.Foo是可接受的，但是Foo或Foo.class则不可接受）。

Identifier提供了一种方法，可以通过它来评估连接点处的运行时对象的实际类型，而不仅仅是静态类型。Identifier在运行时动态地赋予合适的对象。

11.1 捕获何时 this 引用是一个特定的类型

问题

你想捕获所有连接点，这些连接点处的 this 引用是一个特定的类型。

解决方案

使用this([Type | Identifier])切入点。this([Type | Identifier])切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
            this(<type> or <identifier> or *);
```

讨论

this([Type | Identifier])切入点检查捕捉的连接点处的this引用，以决定是否调用关联的通知，它具有5个关键特征：

1. 当执行对象是指定的类型时，this([Type | Identifier])切入点会捕获所有连接点。
2. Type定义参数必须解析成有效的类。它不同于TypePattern，TypePattern中可能使用通配符。
3. Identifier用于检查this引用的运行时对象的类型，以及根据需要把那个对象展示给通知。
4. 使用*通配符允许指定一个有效的对象，连接点处的this引用必须指向它，但是你对它是什么类型不感兴趣。
5. 出现在异常处理上的连接点不具有使用handler(TypePattern)切入点的值，当它们使用任何静态代码块（包括使用staticinitialization(TypePattern)切入点指定的静态类初始化）内的handler(TypePattern)，并且有趣地使用preinitialization(Signature)切入点预初始化对象时，它们将不具有使用this([Type | Identifier])切入点展示this引用的值。

例11-1显示了this([Type | Identifier])切入点的两个示例，它们用于：

- 当this引用指向MyClass类型的对象时，使用Identifier捕获连接点
- 当this引用的类型是AnotherClass时，使用Type指示捕获

例11-1：使用this(Type | Identifier)切入点来捕获基于this引用类型的连接点

```
public aspect ThisRecipe
{
   /*
       Specifies calling advice whenever the executing
       object is of a type that matches the following rules:
```

```
        Identifier/s: MyClass object
    */
    pointcut thisIdentifierMyClassPointcut(MyClass object) : this(object);

    /*
        Specifies calling advice whenever the executing
        object is of a type that matches the following rules:

        Type Pattern: AnotherClass
        */
    pointcut thisTypePatternAnotherClassPointcut() : this(AnotherClass);

    // Advice declaration
    before(MyClass object) : thisIdentifierMyClassPointcut(object)
       && !within(ThisRecipe +)
    {
       System.out.println(
          "------------------- Aspect Advice Logic --------------------");
       System.out.println(
          "In the advice picked by thisIdentifierMyClassPointcut");
       System.out.println("Join Point Kind: " + thisJoinPoint.getKind());
       System.out.println(
          "this reference as passed by Identifier " + object);
       System.out.println(
          "Object referenced by this: " + thisJoinPoint.getThis());
       System.out.println(
          "Signature: " + thisJoinPoint.getStaticPart().getSignature());
       System.out.println(
          "Source Line: " + thisJoinPoint.getStaticPart().getSourceLocation());
       System.out.println(
          "------------------------------------------------------------");
       }

    // Advice declaration
    before() : thisTypePatternAnotherClassPointcut() && !within(ThisRecipe +)
    {
       System.out.println(
          "------------------- Aspect Advice Logic --------------------");
       System.out.println(
          "In the advice picked by thisTypePatternAnotherClassPointcut");
       System.out.println("Join Point Kind: " + thisJoinPoint.getKind());
       System.out.println(
          "Type of executing object: "
             + thisJoinPoint.getThis().getClass().getName());
       System.out.println(
          "Object referenced by this: " + thisJoinPoint.getThis());
       System.out.println(
          "Signature: " + thisJoinPoint.getStaticPart().getSignature());
       System.out.println(
          "Source Line: " + thisJoinPoint.getStaticPart().getSourceLocation());
       System.out.println(
          "------------------------------------------------------------");
       }
}
```

图 11-1 说明了如何应用 this([Type | Identifier])切入点。

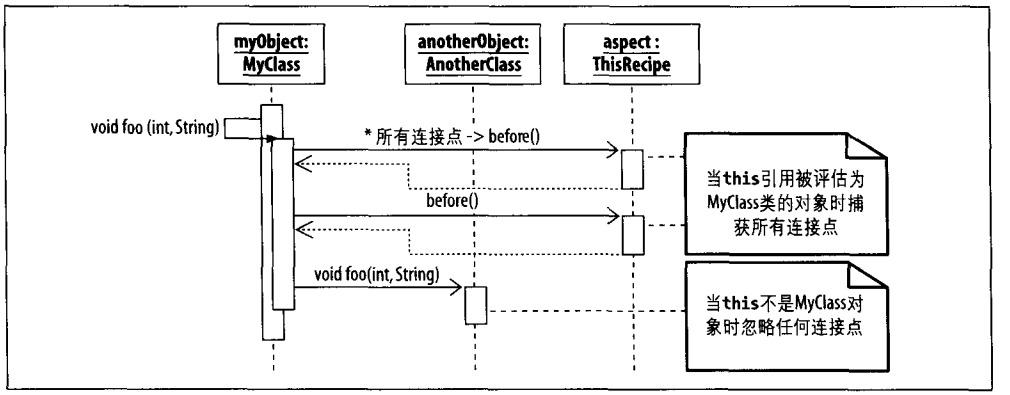

图 11-1：如何应用 this([Type | Identifier])切入点

参考

第 13 章介绍了可用于通知的调用环境；5.1 节描述了 handler(TypePattern)切入点；7.4 节描述了 preinitialization(Signature)切入点；7.5 节描述了 staticinitialization(TypePattern)切入点；9.1 节描述了 within(TypePattern)切入点；12.4 节描述了 NOT（!）运算符。

11.2 捕获何时连接点的目标对象是特定的类型

问题

你想在连接点的目标对象是特定的类型时捕获所有的连接点。

解决方案

使用 target([Type | Identifier])切入点。target([Type or Identifier])切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
        target(<type> or <identifier> or *);
```

讨论

target([Type | Identifier])切入点具有 5 个关键特征：

1. 当目标对象是特定的类型时，target([Type | Identifier])切入点将选择所有的连接点。
2. Type定义参数必须解析成有效的类，以选择相关的连接点。它有别于TypePattern，TypePattern中可能使用通配符。
3. Identifier用于检查在捕捉的连接点处被引用为目标的运行时对象的类型，并根据需要把那个对象展示给通知。
4. 使用 * 通配符允许指定连接点必须有一个目标，但是你对它的类型是什么不感兴趣。
5 在使用handler(TypePattern)切入点的异常处理上，使用staticinitialization(TypePattern)的静态类初始化上，以及有趣地使用preinitialization(Signature)的对象初始化上，在这些地方出现的连接点不具有使用target([Type | Identifier])切入点进行展示的任何目标环境。

例 11-2 显示了 target([Type | Identifier])切入点的两个示例，它们用于：

- 当目标是 Identifier 指定的 MyClass 对象时，捕获连接点
- 当方法调用目标的类型是 Type 指定的 AnotherClass 时，捕获连接点

例 11-2：使用 target([Type | Identifier])切入点，来捕获基于方法目标类型的连接点

```
public aspect TargetRecipe
{
   /*
       Specifies calling advice whenever the target of a methods
       is of a type that matches the following rules:

       Identifier/s: MyClass object
   */
   pointcut targetIdentifierMyClassPointcut(MyClass object) : target(object);

   /*
       Specifies calling advice whenever the target of a methods
       is of a type that matches the following rules:

       Type Pattern: AnotherClass
   */
   pointcut targetTypePatternAnotherClassPointcut() : target(AnotherClass);

   // Advice declaration
   before(MyClass object) : targetIdentifierMyClassPointcut(object)
      && !within(TargetRecipe +)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice picked by targetIdentifierMyClassPointcut");
```

```
        System.out.println("Join Point Kind: " + thisJoinPoint.getKind());
        System.out.println("Object referenced by Target passed by Identifier: " + object);
        System.out.println(
            "Signature: " + thisJoinPoint.getStaticPart().getSignature());
        System.out.println(
            "Source Line: " + thisJoinPoint.getStaticPart().getSourceLocation());
        System.out.println(
            "------------------------------------------------------------");
    }

    // Advice declaration
    before() : targetTypePatternAnotherClassPointcut() && !within(TargetRecipe +)
    {
        System.out.println(
            "------------------- Aspect Advice Logic --------------------");
        System.out.println(
            "In the advice picked by targetTypePatternAnotherClassPointcut");
        System.out.println("Join Point Kind: " + thisJoinPoint.getKind());
        System.out.println(
            "Object referenced by Target: " + thisJoinPoint.getTarget());
        System.out.println(
            "Signature: " + thisJoinPoint.getStaticPart().getSignature());
        System.out.println(
            "Source Line: " + thisJoinPoint.getStaticPart().getSourceLocation());
        System.out.println(
            "------------------------------------------------------------");
    }
}
```

图 11-2 说明了如何应用 target([Type | Identifier])切入点。

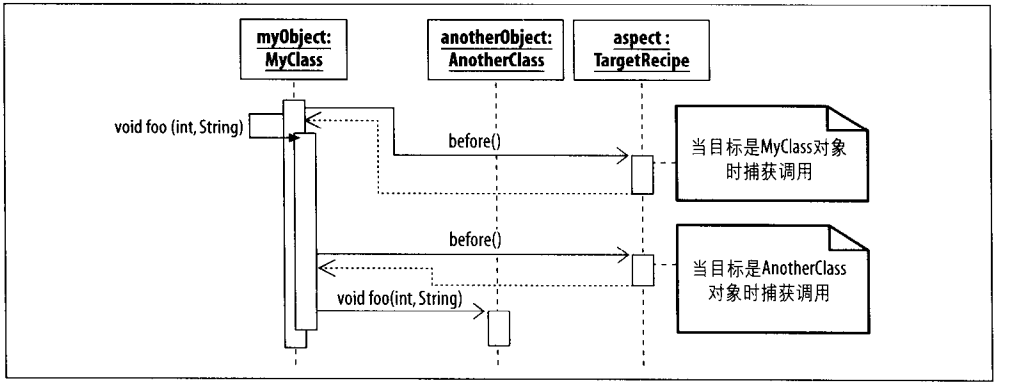

图 11-2：如何应用 target(Type | Identifier)切入点

参考

第 13 章介绍了可供通知使用的调用环境；5.1 节描述了 handler(TypePattern)切入

点；7.4 节描述了preinitialization(Signature)切入点；7.5 节描述了staticinitialization(TypePattern)切入点；9.1节描述了within(TypePattern)切入点；12.4 节描述了 NOT（!）运算符。

11.3 捕获何时连接点的参数是某个数字、类型和次序

问题

你想在连接点的参数（如果有的话）是特定的类型、数字和次序时，捕获遇到的所有连接点。

解决方案

使用 args([Types | Identifiers])切入点。这个切入点的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
            args(<types> or <identifiers> | .. | *, <repeat>);
```

讨论

args([Types | Identifiers])切入点具有 8 个关键特征：

1. 可以使用Types和Identifiers的任意组合来收缩到适当的连接点上，并把适当的环境展示给通知。
2. Identifier用于检查运行时对象的类型，它们是捕捉的连接点处的参数，以及根据需要把这些对象展示给通知。
3. 使用 . . 允许在一定程度上灵活地选择特定连接点处的参数数量，它们必须与 args([Types | Identifiers])声明匹配。
4. 如果不使用 . . , args([Types | Identifiers])切入点就会选择与指定的类型类型精确匹配的所有连接点。这就是说：切入点只与在运行时其参数具有相同次序、数量和类型的方法上的连接点匹配。

   如果使用 . . ，切入点就会使用“最佳适配”策略。这意味着语句 args(MyClass,..,float)将导致任何匹配的连接点都具有两个参数，它开始于一个 MyClass 对象，其后接着任意数量的参数，它们中间包括一个浮点数。
5. 在单个 args([Types | Identifiers])声明中只能使用一个 . . 。

6. 使用*通配符允许在参数的类型方面表现出一定的灵活性，但是连接点的参数数量必须匹配。

   例如，如果在args([Types | Identifiers])切入点声明中指定单个*——例如，args(*)——那么这个切入点将与具有一个任意类型参数的任何连接点匹配，并且会忽略没有参数的任何连接点。类似地，语句args(*,*)将匹配包含两个任意类型参数的任何连接点。

7. 出现在字段访问中的连接点（通过get(TypePattern)切入点或set(TypePattern)切入点捕获）或者出现在静态类初始化中的连接点（通过staticinitialization(TypePattern)切入点捕获）没有要使用args([Types | Identifiers])切入点展示的任何参数。

8. 如果把Object用作要被args([Types | Identifiers])切入点选择的标识符的类型，那么这将成功地匹配基本类型以及特定的Object实例。基本类型会自动“装箱”进其对应的对象类型中，例如，在把float基本类型传递给通知之前，会将其“装箱”进Float中。这种行为提供了一种方法，可以通过它把基本参数值带到通知中。

例11-3说明了args([Types | Identifiers])切入点的两个示例，它们用于：

- 当方法的参数是Identifier指定的MyClass类的对象时，捕获所有的连接点
- 当参数的类型是Type指定的AnotherClass类的类型时，捕获连接点

例11-3：使用args([Types | Identifiers])切入点，捕获基于方法参数类型的连接点

```
public aspect ArgsRecipe
{
   /*
       Specifies calling advice whenever the type of a methods argument
       is of a type that matches the following rules:

       Identifier/s: MyClass object
   */
   pointcut argIdentifierMyClassPointcut(MyClass object) : args(object);

   /*
       Specifies calling advice whenever the type of a methods argument
       is of a type that matches the following rules:

       Type Pattern: AnotherClass
   */
   pointcut argTypePatternAnotherClassPointcut() : args(AnotherClass);

   // Advice declaration
   before(MyClass object) : argIdentifierMyClassPointcut(object)
      && !within(ArgsRecipe +)
```

```
    {
        System.out.println(
            "------------------- Aspect Advice Logic --------------------");
        System.out.println(
            "In the advice picked by argIdentifierMyClassPointcut");
        System.out.println("Join Point Kind: " + thisJoinPoint.getKind());
        System.out.println(
            "Signature: " + thisJoinPoint.getStaticPart().getSignature());
        System.out.println(
            "Source Line: " + thisJoinPoint.getStaticPart().getSourceLocation());
        System.out.println(
            "Arguments picked up using Identifiers: " + object);
        System.out.println(
            "------------------------------------------------------------");
    }

    // Advice declaration
    before() : argTypePatternAnotherClassPointcut() && !within(ArgsRecipe +)
    {
        System.out.println(
            "------------------- Aspect Advice Logic --------------------");
        System.out.println(
            "In the advice picked by argTypePatternAnotherClassPointcut");
        System.out.println("Join Point Kind: " + thisJoinPoint.getKind());
        System.out.println(
            "Signature: " + thisJoinPoint.getStaticPart().getSignature());
        System.out.println(
            "Source Line: " + thisJoinPoint.getStaticPart().getSourceLocation());
        System.out.println(
            "------------------------------------------------------------");
    }
}
```

图 11-3 说明了如何应用 args([Types | Identifiers])切入点。

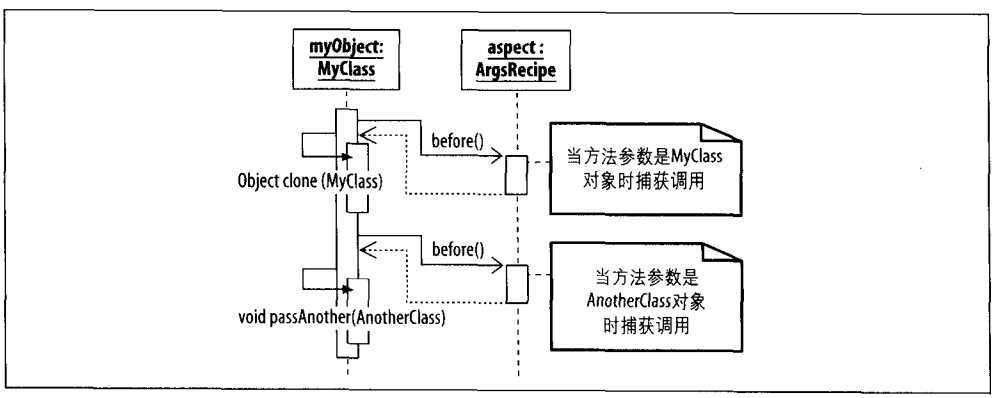

图 11-3：如何应用 args([Types | Identifiers])切入点

警告： 如果通过创建比原始方法具有更少参数的新方法来进行重载，方法重载就可能发生问题。`args([Types | Identifiers])`切入点会检查连接点处的参数数量、类型和次序，因此，提供更少参数的重载方法可能不会触发通过原始方法触发的通知。

参考

11.1 节和 11.2 节分别描述了`this([Type | Identifier])`切入点和`target([Type | Identifier])`切入点；8.1 节和 8.3 节分别描述了`get(TypePattern)`切入点和`set(TypePattern)`切入点；第 13 章介绍了可供通知使用的调用环境；9.1 节描述了`within(TypePattern)`切入点；12.4 节描述了 NOT（!）运算符。

第 12 章

捕获基于布尔或复合表达式的连接点

12.0 简介

本章研究了如何结合切入点，并将其表示为布尔表达式。评估切入点逻辑，以确定是否捕获了特定的连接点。切入点逻辑的布尔性质允许使用传统的布尔表达式（如逻辑 AND、逻辑 OR 或逻辑 NOT）来结合切入点声明。

在切入点声明中也可以使用 `if` 语句来表达条件逻辑。`if` 语句用于比较包含非布尔值的表达式。在运行时传递 `if` 语句评估的值，得到一个 `true` 或 `false` 结果。

匿名切入点是更复杂地命名切入点声明的构件（building block）。匿名切入点可以声明为复合命名切入点的一部分，或者声明为通知声明的一部分。这样，命名切入点反过来又是切入点逻辑重用的构件。切入点重用允许高效管理复杂的逻辑，可以使用 AspectJ 内可用的切入点来定义它。本章最后研究了如何声明匿名切入点，并把这些切入点结合进可重用的命名切入点中。

12.1 捕获何时连接点上的运行时条件评估为 true

问题

在比较一些可在捕捉的连接点处评估的运行时值时，你想触发基于 `true` 结果的通知。

解决方案

使用 `if(Expression)` 语句来评估包含要比较的运行时变量的布尔表达式。`if(Expression)` 语句的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
            if(<Boolean expression>);
```

讨论

if(Expression)语句具有两个关键特征：

1. if(Expression)切入点评估在运行时提供的变量，得到关于连接点是否应该触发相应通知的true或false结果。
2. Expression可以由多个逻辑元素组成，包括展示的连接点环境、静态变量，以及其他切入点声明。

例12-1显示了if(Expression)语句的用法。如果构成表达式的变量的运行时值导致结果为true，则只会执行after通知。

例12-1：用于评估目标应用程序变量的if(Expression)语句

```
public aspect IfRecipe
{
   // Define some variables for comparison
   private static final long realisticSalary = 300001;

   /*
    * Specifies calling advice if this is referencing an object of class MyClass
    * and the object has a realistic salary:
   */
   pointcut ifJoinPointThisHasRealisticSalaryPointcut() : if (
      (thisJoinPoint.getThis() instanceof MyClass)
         && ((MyClass) thisJoinPoint.getThis()).getSalary()
            < realisticSalary
         && ((MyClass) thisJoinPoint.getThis()).getSalary() > 0)
      && !withincode(* MyClass.get*())
      && !execution(* MyClass.get*());

   // Advice declaration
   //This advice will be executed before the pointcut that picks it
   after() : ifJoinPointThisHasRealisticSalaryPointcut()
      && !within(IfRecipe +)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice picked by ifJoinPointThisHasRealisticSalaryPointcut");
      System.out.println(
         "Join Point Kind: " + thisJoinPoint.getKind());
      System.out.println(
         "Executing object: " + thisJoinPoint.getThis());
      System.out.println(
         "MyClass instance: "
            + ((MyClass) thisJoinPoint.getThis()).getName()
```

```
            + " : "
          + ((MyClass) thisJoinPoint.getThis()).getSalary());
      System.out.println(
          "Signature: " + thisJoinPoint.getSignature());
      System.out.println(
          "Source Line: " + thisJoinPoint.getSourceLocation());
      System.out.println(
          "------------------------------------------------------------");
   }
}
```

if(Expression)语句允许定义关于是否应该把一份通知应用于特定连接点的条件逻辑。这个条件逻辑在运行时执行，并且在到达连接点那一刻必须工作在有效的类型上。

例 12-1 中的 ifJoinPointThisHasRealisticSalaryPointcut()切入点中包含的条件逻辑指定在发生以下事件时，应该触发相应的 after 通知：

- 执行对象是 MyClass 类型。
- 对象的 salary 属性小于 realisticSalary 常量。
- 对象的 salary 属性大于 0。
- 当前连接点不在 getSalary()方法内。通过使用通配符，连接点绝对不能出现在开始到达的 MyClass 类的任何方法中。

可以使用 AND（&&）运算符来逻辑地结合每个条件。前 3 个条件可以相当容易地理解成切入点布尔逻辑的一部分，但是最后一个条件更有趣一点。必须包含 !withincode(* MyClass.get*())条件，以防止从通知内调用 getSalary()，这反过来又会触发对通知的递归调用，并导致一个无限循环。

参考

12.2 节和 12.3 节分别描述了 AND（&&）运算符和 OR（||）运算符；12.4 节描述了 NOT（!）运算符；第 13 章描述了 AspectJ 支持的不同形式的通知。

12.2 使用逻辑 AND（&&）结合切入点

问题

你想结合一些切入点声明，使得只要切入点声明内的所有条件都评估为 true，就会在连接点上执行通知。

解决方案

使用&&运算符。&& operator运算符的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
                    <pointcut declaration> && <pointcut declaration>
```

讨论

例12-2说明了一个使用&&运算符的例子，它把两个切入点的逻辑结合进单独一个切入点声明中。

例12-2：使用&&运算符结合两个切入点

```
public aspect LogicalAndRecipe
{
   /*
    Specifies calling advice whenever a method
    matching the following rules gets called:

    Class Name: MyClass
    Method Name: Any Method
    Method Return Type: Any Return Type
    Method Parameters: Any Parameters
   */
   pointcut callAnyMethodOnMyClass() : call(* MyClass.* (..));

   /*
    Specifies calling advice whenever a method
    matching the following rules gets called:

    Class Name: MyClass
    Method Name: bar
    Method Return Type: void
    Method Parameters: None
   */
   pointcut callBarPointcut() : call(void MyClass.bar());

   /*
    Specifies calling advice whenever a join points is
    encountered that would be picked by both pointcuts
    specified:

    Pointcut name: callAnyMethodOnMyClass
    Pointcut name: callBarPointcut
    Method Return Type: void
    Method Parameters: None
   */
   pointcut callIntersectionAnyAndBar() : callAnyMethodOnMyClass()
                        && callBarPointcut();
```

```
    // Advice declaration
    before() : callAnyMethodOnMyClass() && !within(LogicalAndRecipe +)
    {
       System.out.println(
          "------------------- Aspect Advice Logic --------------------");
       System.out.println(
          "In the advice picked by callAnyMethodOnMyClass");
       System.out.println(
          "Signature: " + thisJoinPoint.getSignature());
       System.out.println(
          "Source Line: " + thisJoinPoint.getSourceLocation());
       System.out.println(
          "------------------------------------------------------------");
    }

    // Advice declaration
    before() : callBarPointcut() && !within(LogicalAndRecipe +)
    {
       System.out.println(
          "------------------- Aspect Advice Logic --------------------");
       System.out.println("In the advice picked by callBarPointcut");
       System.out.println(
          "Signature: " + thisJoinPoint.getSignature());
       System.out.println(
          "Source Line: " + thisJoinPoint.getSourceLocation());
       System.out.println(
          "------------------------------------------------------------");
    }

    // Advice declaration
    before() : callIntersectionAnyAndBar()
       && !within(LogicalAndRecipe +)
    {
       System.out.println(
          "------------------- Aspect Advice Logic --------------------");
       System.out.println(
          "In the advice picked by callIntersectionAnyAndBar");
       System.out.println(
          "Signature: " + thisJoinPoint.getSignature());
       System.out.println(
          "Source Line: " + thisJoinPoint.getSourceLocation());
       System.out.println(
          "------------------------------------------------------------");
    }
}
```

大多数开发人员都熟悉 && 运算符的行为。不过，对于初学者，其数学解释是：当使用逻辑 AND 把两组事情结合到一起时，这就提供了两个集合的交集。把这些数学术语放到 AspectJ 环境中，当用 && 运算符把两个或多个简单的切入点结合进一个复合切入点时，被两个独立的切入点选择的连接点将触发复合切入点的关联通知。更简单地讲，如果任一个简单切入点选择了任何一个连接点，那么整个复合切入点就不会选择这个连接点。

使用&&运算符结合切入点的次序还会影响复合切入点的解释方式。&&运算符的运行时分析是从左到右执行的。这意味着在检查候选连接点时，指示它不包含连接点的第一个切入点就是停止比较的地方。这对于Java中的&&运算符是成立的，并且当复合中的比较之一必须通过前一个条件加以保护时特别有用，如例12-3所示(从9.1节中复制而来)。

例 12-3：使用 && 运算符次序保护后面的比较

```
pointcut ifJoinPointThisHasUnRealisticSalaryPointcut() : if (
        thisJoinPoint.getThis() instanceof MyClass
        && ((MyClass) thisJoinPoint.getThis()).getSalary() >= realisticSalary)
        && !withincode(long MyClass.getSalary());
```

在这个示例中，第一个条件指出：this引用必须是MyClass的一个实例，以继续对语句的余下部分进行评估。这会保护语句下一部分中的调用，它会把this引用强制转换成MyClass。如果第一个条件失败，那么永远不会到达第二个条件；由于这种行为，&&运算符有时也称为短路运算符。

参考

12.1 节描述了if(Expression)切入点；9.1 节描述了within(TypePattern)切入点；12.4 节描述了 NOT（!）运算符；第 13 章描述了 AspectJ 中可用的不同类型的通知。

12.3 使用逻辑 OR（||）结合切入点

问题

你想结合一些切入点声明，使得只要切入点声明内的条件之一评估为true，就会在连接点上执行通知。

解决方案

使用||运算符。||运算符的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
            <pointcut declaration> || <pointcut declaration>
```

讨论

例12-4说明了||运算符，它把callFooIntStringPointcut()和callBarPointcut()结合进单独一个名为callIntersectionFooOrBar()的复合切入点中。

例 12-4：使用||运算符结合两个切入点声明

```
public aspect LogicalOrRecipe
{
   /*
    Specifies calling advice whenever a method
    matching the following rules gets called:

    Class Name: MyClass
    Method Name: foo
    Method Return Type: void
    Method Parameters: int and a String
   */
   pointcut callFooIntStringPointcut() : call(
      void MyClass.foo(int, String));

   /*
    Specifies calling advice whenever a method
    matching the following rules gets called:

    Class Name: MyClass
    Method Name: bar
    Method Return Type: void
    Method Parameters: None
   */
   pointcut callBarPointcut() : call(void MyClass.bar());

   /*
    Specifies calling advice whenever a join points is
    encountered that would be picked by both pointcuts
    specified:

    Pointcut name: callFooIntStringPointcut
    Pointcut name: callBarPointcut
    Method Return Type: void
    Method Parameters: None
   */
   pointcut callIntersectionFooOrBar() : callFooIntStringPointcut()
                                || callBarPointcut();

   // Advice declaration
   before() : callFooIntStringPointcut()
      && !within(LogicalOrRecipe +)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice picked by callFooIntStringPointcut");
      System.out.println(
         "Signature: " + thisJoinPoint.getSignature());
      System.out.println(
         "Source Line: " + thisJoinPoint.getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
```

```
   // Advice declaration
   before() : callBarPointcut() && !within(LogicalOrRecipe +)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println("In the advice picked by callBarPointcut");
      System.out.println(
         "Signature: " + thisJoinPoint.getSignature());
      System.out.println(
         "Source Line: " + thisJoinPoint.getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }

   // Advice declaration
   before() : callIntersectionFooOrBar()
      && !within(LogicalOrRecipe +)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice picked by callIntersectionFooOrBar");
      System.out.println(
         "Signature: " + thisJoinPoint.getSignature());
      System.out.println(
         "Source Line: " + thisJoinPoint.getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

除了使用逻辑 AND 结合切入点（如 12.2 节所示）之外，还可以使用逻辑 OR 结合切入点及其他逻辑表达式。那些熟悉逻辑运算性质的人将知道：如果任何一个组成部分满足比较条件，就会把逻辑 OR 比较的结果看作为正。在 AspectJ 语言中，如果连接点在使用逻辑OR结合进复合切入点的任何成分切入点上启动通知，连接点就会触发那个通知。

AspectJ 中的 || 运算符还显示了短路行为，这与 && 运算符相似。不过，其短路规则与 && 运算符相反。利用 || 运算符，在某个条件评估为正的那一刻，可以把复合语句评估为正并且它是完整的。

参考

12.2节介绍了AND(&&)运算符及其短路评估特性；9.1节描述了within(TypePattern)切入点；12.4 节描述了 NOT（!）运算符；第 13 章描述了 AspectJ 中可用的不同类型的通知。

12.4 捕获所有未通过切入点声明指定的连接点

问题

你想捕获所有未被特定切入点捕捉的连接点。

解决方案

使用一元！运算符，指定忽略那些通常由特定切入点捕获的连接点。!运算符的语法如下：

```
pointcut <pointcut name>(<any values to be picked up>) :
            !<pointcut declaration>
```

讨论

例 12-5 显示了！运算符的一个例子，它用于捕获所有未被 call(void MyClass.foo(int,String)切入点捕获的连接点。

例 12-5：使用!运算符，来捕获未被 call(Signature)切入点捕捉的连接点

```
public aspect LogicalNotRecipe
{
   /*
    Specifies calling advice whenever a method
    does NOT match the following rules
    gets called:

    Class Name: MyClass
    Method Name: foo
    Method Return Type: void
    Method Parameters: an int followed by a String
   */
   pointcut notCallPointCutFooIntString() : !call(void MyClass.foo(int, String));

   // Advice declaration
   before() : notCallPointCutFooIntString()
      && !within(LogicalNotRecipe +)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice picked by notCallPointCutFooIntStringAnyReturn()");
      System.out.println(
         "Signature: " + thisJoinPoint.getSignature());
      System.out.println(
         "Source Line: " + thisJoinPoint.getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
```

```
    }
}
```

参考

12.2 节和 12.3 节介绍了用于结合切入点声明的技术；9.1 节中描述了 within(TypePattern) 切入点；第 13 章描述了 AspectJ 中可用的不同类型的通知。

12.5 声明匿名切入点

问题

你想在命名的切入点声明中匿名地声明一个简单的切入点，或者把它直接附加到一个通知上。

解决方案

匿名切入点是切入点声明的构件。在所有基于切入点的章节中都会用到它们，但是本节更细致地关注了匿名切入点。

例12-6说明了匿名切入点的一个例子，它可以用作更复杂地命名的切入点的基础，以及直接用在通知声明上。

例 12-6：使用匿名切入点

```
public aspect AnonymousPointcutRecipe
{
    /*
     A pointcut declaration that is built up from one
     anonymous pointcut:

     Anonymous Pointcuts: call(void MyClass.foo(int,String)
    */
    pointcut singleAnonymousPointcut() : call(void MyClass.foo(int, String));

    /*
     A pointcut declaration that is built up from two
     anonymous pointcuts:

     Anonymous Pointcuts: call(void MyClass.foo(int,String)
                          call(void MyClass.foo(int,String)
                          !within(AnonymousPointcutRecipe +)

    */
    pointcut multipleAnonymousPointcut() : (
       call(void MyClass.bar())
```

```
      || call(void MyClass.foo(int, String))
      && !within(AnonymosPointcutRecipe +));

   /*
    A pointcut declaration attached to the advice it will invoke,
    built up from anonymous pointcuts:

    Anonymous Pointcuts: within(LogicalOrRecipe +)
   */
   before() : singleAnonymousPointcut()
      && !within(AnonymousPointcutRecipe +)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice picked by singleAnonymousPointcut and");
      System.out.println("!within(AnonymousPointcutRecipe +");
      System.out.println(
         "Signature: " + thisJoinPoint.getSignature());
      System.out.println(
         "Source Line: " + thisJoinPoint.getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }

   /*
    A pointcut declaration attached to the advice it will invoke,
    built up from anonymous pointcuts:

    Anonymous Pointcuts: None
   */
   before() : multipleAnonymousPointcut()
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice picked by multipleAnonymousPointcut()");
      System.out.println(
         "Signature: " + thisJoinPoint.getSignature());
      System.out.println(
         "Source Line: " + thisJoinPoint.getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

讨论

这个秘诀不同于本书中其他基于切入点的秘诀，因为它涉及两个特殊的机制之一：用于声明切入点，而不是检查特定的切入点类型。可以在命名切入点声明内匿名地声明切入点，或者通过把切入点直接附加到它们将调用的通知上来进行声明。

参考

12.2节和12.3节介绍了用于结合切入点声明的技术；9.1节描述了within(TypePattern)切入点；12.4节描述了NOT（!）运算符；第13章描述了AspectJ中可用的不同类型的通知。

12.6 重用切入点

问题

你想重用切入点表达式。

解决方案

声明一个切入点，可以在要重用它的地方通过名称来引用它。

讨论

例12-7显示了一个例子，其中在声明reuseNamedPointcut()时重用了命名为foundationNamedPointcut()的切入点。

例12-7：重用命名的切入点

```
public aspect PointcutReuseRecipe
{
   /*
    A pointcut declaration that is to be used and reused:

    Anonymous Pointcuts: call(void MyClass.foo(int,String)

   */
   pointcut foundationNamedPointcut() : call(
      void MyClass.foo(int, String));

   /*
    A pointcut declaration that is built up from two
    pointcuts:

    Anonymous Pointcuts: !within(AnonymousPointcutRecipe +)
    Named Pointcuts: foundationNamedPointcut()

   */
   pointcut reuseNamedPointcut() : foundationNamedPointcut()
      && !within(PointcutReuseRecipe +);

   /*
    A pointcut declaration attached to the advice it will invoke,
    built up from simple named and anonymous pointcuts:
```

```
    Anonymous Pointcuts: !within(LogicalOrRecipe +)
    Named Pointcuts: foundationNamedPointcut();
   */
   before() : foundationNamedPointcut()
      && !within(PointcutReuseRecipe +)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice picked by foundationNamedPointcut() and");
      System.out.println("!within(AnonymousPointcutRecipe() +");
      System.out.println(
         "Signature: " + thisJoinPoint.getSignature());
      System.out.println(
         "Source Line: " + thisJoinPoint.getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }

   /*
    A pointcut declaration attached to the advice it will invoke,
    built up from complex pointcuts built reusing other pointcut
    declarations:

    Named Pointcuts: reuseNamedPointcut
   */
   before() : reuseNamedPointcut()
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice picked by reuseNamedPointcut()");
      System.out.println(
         "Signature: " + thisJoinPoint.getSignature());
      System.out.println(
         "Source Line: " + thisJoinPoint.getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

命名切入点类似于一个方法。它具有一个签名，并且可以在特定方法内的切入点声明的所有余下部分中引用它，或者甚至在其他方面中依据命名切入点的访问修饰符来引用它。在考虑方面之间的继承关系时，其中可以把切入点声明为抽象的并且可以通过继承方面来实现实际的切入点逻辑，这一点会变得愈加重要。

当声明切入点以便重用时，考虑将在哪里重用切入点声明是重要的。可以使用与切入点声明关联的访问修饰符，以便控制切入点声明在应用程序内的什么位置是可见的。

在 AspectJ 中，切入点声明具有与普通的 Java 方法相同的访问修饰符：public，切入点声明在整个应用程序方面中都是可见的；默认（没有指定修饰符），切入点声明对于同

一个包中的所有其他方面是可见的；protected，切入点声明仅对于子方面是可见的；private，切入点声明仅在声明它的方面内是可见的。

参考

第15章讨论了方面之间的继承关系以及切入点声明的隐含意义；12.2节和12.3节介绍了用于结合切入点声明的技术；9.1节描述了`within(TypePattern)`切入点；12.6节描述了NOT（!）运算符；第13章描述了AspectJ中可用的不同类型的通知。

第 13 章

定义通知

13.0 简介

本章研究了在 AspectJ 中可用于指定通知的不同方式。切入点定义了你对哪些连接点感兴趣，通知则定义了当遇到这些连接点时做什么。

通知块包含直观的 Java 代码。通知看起来非常像 Java 方法，只不过不能直接从应用程序中调用它。执行通知的是 AspectJ 运行时系统，它依据与通知关联的切入点逻辑来执行通知。

本章首先研究了通知如何同应用程序中的类交互，包括用于访问触发连接点环境的机制。然后讨论了不同类型的通知，最后介绍了通知的优先级，以及在把多份通知应用于同一个连接点时，如何控制调用通知的次序。

13.1 访问类成员

问题

你想从通知内访问特定对象的属性和方法。

解决方案

把合适的对象作为切入点声明上的参数传递给通知。

讨论

例13-1说明了如何访问类的公共和私有成员，在遇到调用连接点时，这个类的对象是可用的。

例13-1：把对象传递给通知，用于访问其方法和属性

```
public privileged aspect MemberAccessRecipe
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets executed:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: void
   Method Parameters: an int followed by a String
   */
   pointcut executionOfFooPointCut() : execution(
      void MyClass.foo(int, String));

   // Advice declaration
   after(MyClass myClass) : executionOfFooPointCut() && this(myClass)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "Accessing the set(float) member of the MyClass object");
      System.out.println(
         "Privileged access not required for this method call as it is public");
      myClass.setF(2.0f);
      System.out.println(
         "Using the privileged aspect access to the private f member variable");
      System.out.print("The current value of f is: ");
      System.out.println(myClass.f);
      System.out.println(
         "Signature: " + thisJoinPoint.getSignature());
      System.out.println(
         "Source Line: " + thisJoinPoint.getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

通过使用this(Identifier)切入点定义，使得MyClass类的对象可供通知使用。this(Identifier)切入点定义有效地把通知展示给被触发连接点处的this引用指向的对象。从通知内调用setF(float)方法，并显示对MyClass对象的公共方法的访问。为了获得对MyClass.f私有属性的访问，方面不得不对其结构执行一些额外的更改。方面尝试通过直接访问私有成员来破坏封装性，因此，必须把方面声明为privileged，因为它正在提交潜在的侵入动作。

AspectJ提供了privileged关键字，它用在方面需要完全不受限制地访问类的地方，包括那些未在类的公共接口上声明的成员变量和方法。方面的privileged状态应该充当一个警告，在对方面或者它所应用的类执行任何更改时，必须当心这个警告，因为这些更改可能潜在地在整个应用程序中引发其他问题。

参考

4.1 节中描述了 call(Signature)切入点；9.1 节中描述了 within(TypePattern)切入点；11.1 节中描述了 this(TypePattern or Identifier)切入点；附录中包含关于 JoinPoint 类及其子类的一个快速参考。

13.2 访问连接点环境

问题

你想从通知内访问连接点环境。

解决方案

使用 thisJoinPoint 和 thisJoinPointStaticPart 变量。

讨论

Java 中的类具有一个 this 变量，它允许类的对象引用和处理它们自身。方面可通过 AspectJ 编译器转换成类；因此，方面也具有 this 引用。

不过，可以从触发通知的连接点把额外的连接点环境展示给通知。AspectJ 提供了 thisJoinPoint 变量，用于展示 this 连接点环境。除了 thisJoinPoint 之外，如果可以静态地访问正在访问的环境，那么 thisJoinPointStaticPart 变量也是有用的。

例 13-2 显示了一些可从 thisJoinPoint 泛型变量得到的信息。

例 13-2：使用 thisJoinPoint 变量

```
public aspect ThisJoinPointRecipe
{
    /*
    Specifies calling advice whenever a method
    matching the following rules gets called:

    Class Name: MyClass
    Method Name: foo
```

```
Method Return Type: void
Method Parameters: an int followed by a String
*/
pointcut callPointCut() : call(void MyClass.foo(int, String));

// Advice declaration
before() : callPointCut() && !within(ThisJoinPointRecipe +)
{

   System.out.println(
      "------------------- Aspect Advice Logic --------------------");
   System.out.println(
      "Exercising the static parts of AspectJ 1.1.1 thisJoinPoint");
   System.out.println(
      "Source Line: "
         + thisJoinPointStaticPart.getSourceLocation());
   System.out.println(
      "Join Point Kind: "
         + thisJoinPointStaticPart.getKind());
   System.out.println(
      "Simple toString: "
         + thisJoinPointStaticPart.toString());
   System.out.println(
      "Simple toShortString: "
         + thisJoinPointStaticPart.toShortString());
   System.out.println(
      "Simple toLongString: "
         + thisJoinPointStaticPart.toLongString());
   System.out.println(
      "Exercising the join point generic signature of AspectJ 1.1.1 thisJoinPoint");
   System.out.println(
      "Signature: "
         + thisJoinPointStaticPart.getSignature());
   System.out.println(
      "Signature name: "
         + thisJoinPointStaticPart.getSignature().getName());
   System.out.println(
      "Signature declaring type: "
         + thisJoinPointStaticPart.getSignature().getDeclaringType());
   System.out.println(
      "------------------------------------------------------------");
}

// Advice declaration
before() : callPointCut() && !within(ThisJoinPointRecipe +)
{

   System.out.println(
      "------------------- Aspect Advice Logic --------------------");
   System.out.println(
      "Exercising the dynamic parts of AspectJ 1.1.1 thisJoinPoint");
   System.out.println(
      "Get the this reference: " + thisJoinPoint.getThis());
   System.out.println(
```

```
            "Getting the Target: " + thisJoinPoint.getTarget());
        System.out.println("Join Point Arguments: ");
        Object[] args = thisJoinPoint.getArgs();
        for (int count = 0; count < args.length; count++)
        {
            System.out.println(args[count]);
        }
        System.out.println(
            "------------------------------------------------------------");
    }
}
```

例 13-2 中的代码产生如下输出：

```
------------------- Aspect Advice Logic --------------------
Exercising the static parts of AspectJ 1.1.1 thisJoinPoint
Source Line: MyClass.java:14
Join Point Kind: method-call
Simple toString: call(void MyClass.foo(int, String))
Simple toShortString: call(MyClass.foo(..))
Simple toLongString: call(public void MyClass.foo(int, java.lang.String))
Exercising the join point generic signature of AspectJ 1.1.1 thisJoinPoint
Signature: void MyClass.foo(int, String)
Signature name: foo
Signature declaring type: class MyClass
------------------------------------------------------------
------------------- Aspect Advice Logic --------------------
Exercising the dynamic parts of AspectJ 1.1.1 thisJoinPoint
Get the this reference: null
Getting the Target: MyClass@d19bc8
Join Point Arguments:
1
Russ Miles
------------------------------------------------------------
```

thisJoinPoint变量包含关于触发连接点的静态和动态环境信息。静态连接点环境信息包含可以在编译和织入时决定的任何信息，如第 1 章中所解释的。动态连接点环境信息则只能在运行时填充，因为它依赖于连接点环境实际的运行时状态。

为了保持事情尽可能高效，可以通过使用thisJoinPoint.getStaticPart()，或者如例 13-2 所示通过访问 thisJoinPointStaticPart 变量，来访问静态连接点信息。如果特定的通知只使用了 getStaticPart()方法或 thisJoinPointStaticPart，那么 AspectJ 编译器就可以执行编译优化，以减少与访问连接点环境关联的系统开销。

注意： thisJoinPoint变量是AspectJ运行时库中声明的JoinPoint类的一个对象。JoinPoint泛型类的不同子类可以实例化成thisJoinPoint变量，这依赖于触发连接点的类型。附录中提供了 JoinPoint 类及其子类的一个快速参考。

参考

4.1 节中描述了 call(Signature)切入点；9.1 节中描述了 within(TypePattern)切入点；附录中包含关于 JoinPoint 类及其子类的一个快速参考。

13.3 在连接点之前执行通知

问题

你想在触发它的连接点之前执行通知。

解决方案

使用 before()类型的通知。

讨论

例 13-3 说明了在调用 void MyClass.foo(int,String)方法之前，如何执行通知。

例 13-3：在方法调用之前执行通知

```
public aspect BeforeAdviceRecipe
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets called:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: void
   Method Parameters: an int followed by a String
   */
   pointcut callPointCut() : call(void MyClass.foo(int, String));

   // Advice declaration
   before() : callPointCut() && !within(BeforeAdviceRecipe +)
   {

      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "Source Location: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

参考

4.1 节中描述了 call(Signature)切入点；9.1 节中描述了 within(TypePattern)切入点；13.2 节中研究了 thisJoinPoint 变量。

13.4 在连接点周围执行通知

问题

你想在触发它的连接点周围执行通知。

解决方案

使用 around()类型的通知。

讨论

例 13-4 显示了如何在 void MyClass.foo(int, String)方法周围执行通知。

例 13-4：在方法调用周围执行通知

```
public aspect AroundAdviceRecipe
{
   /*
    Specifies calling advice whenever a method
    matching the following rules gets called:

    Class Name: MyClass
    Method Name: bar
    Method Return Type: int
    Method Parameters:
   */
   pointcut callFooPointCut() : call(int MyClass.foo());

   /*
    Specifies calling advice whenever a method
    matching the following rules gets called:

    Class Name: MyClass
    Method Name: bar2
    Method Return Type: int
    Method Parameters: int
    */
   pointcut callBarPointCut(int value) : call(int MyClass.bar(int))
      && args(value);

   /*
    Specifies calling advice whenever a method
    matching the following rules gets called:
```

```
 Class Name: MyClass
 Method Name: baz
 Method Return Type: int
 Method Parameters:
 */
pointcut callBazPointCut() : call(int MyClass.baz());

 // Advice declaration
 // This advice will be executed before the pointcut that picks it
int around() : callFooPointCut() && !within(AroundAdviceRecipe +)
{
   System.out.println(
      "------------------- Aspect Advice Logic --------------------");
   System.out.println(
      "Signature: " + thisJoinPoint.getSignature());
   System.out.println(
      "Source Location: "
         + thisJoinPoint.getStaticPart().getSourceLocation());
   System.out.println(
      "------------------------------------------------------------");
   return proceed();
}

 // Advice declaration
 // This advice will be executed before the pointcut that picks it
int around(int value) : callBarPointCut(value)
   && !within(AroundAdviceRecipe +)
{
   System.out.println(
      "------------------- Aspect Advice Logic --------------------");
   System.out.println(
      "Signature: " + thisJoinPoint.getSignature());
   System.out.println(
      "Source Location: "
         + thisJoinPoint.getStaticPart().getSourceLocation());
   System.out.println(
      "------------------------------------------------------------");
   return proceed(value);
}

 // Advice declaration
 // This advice will be executed before the pointcut that picks it
int around() : callBazPointCut() && !within(AroundAdviceRecipe +)
{
   System.out.println(
      "------------------- Aspect Advice Logic --------------------");
   System.out.println(
      "Signature: " + thisJoinPoint.getSignature());
   System.out.println(
      "Source Location: "
         + thisJoinPoint.getStaticPart().getSourceLocation());
   System.out.println(
      "------------------------------------------------------------");
   return 200;
```

```
    }
}
```

around()通知是一种强大的构造，它指示AspectJ应该运行通知，而不是指示连接点触发它。这允许重写应用程序中存在的原始逻辑。这个秘诀说明了around()通知，可以依据是否从around()通知块内发起proceed()调用，来被动或主动地应用这个通知。图 13-1 显示了例 13-4 中的行为，其中应用了被动和主动的around()通知。

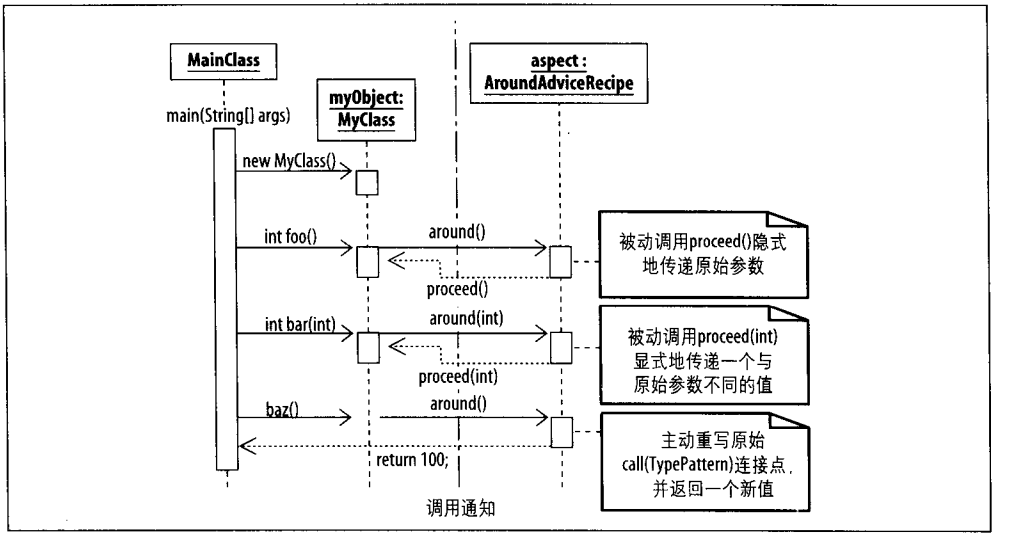

图 13-1：在 call(TypePattern)切入点上被动和主动地使用 around()通知

proceed()调用指示around()通知应该继续执行原始连接点逻辑，并传递原来可用的任何值。在例 13-4 的第一份通知中，没有参数要传递；在第二份通知中，在调用连接点上有一个 int 可用，由于这个原因，必须把参数传递到proceed()调用上。该参数可以具有与原始值完全不同的值，但是不能具有不同的类型或类；否则，这将引发编译错误。

对比例 13-4 中前两份通知中所示的around()通知更被动的形式可知，around()通知可用于重写触发它的连接点。由于proceed()调用，MyClass.baz()方法会被第三个around()通知重写。为了完成原始连接点的重写，around()通知将返回一个值，它不同于原来预期的值。

可以用相同的重写方法来重写构造函数。当方面需要控制实例化的实际对象时，这种机制是有用的。从构造函数返回的对象是最近实例化的对象，从而可以由通知决定。通知能够返回一个与目标应用程序所预期的不同类的对象，只要它是合适的类即可。

around()通知必须具有指定的返回值，但是如果不需要值，那么它可以是void。可以依据proceed()调用的返回值或者适合于所捕捉连接点的类的全新对象，从原始逻辑获得返回值。

如果把通知应用于具有不同返回类型的连接点，那么通知可能声明它的返回类型是Object类。如果那些返回类型之一是基本类型，如float或int，AspectJ将自动在运行时“取消装箱”基本的返回值。

例如，当返回一个Object的around()通知块通知某个方法（它返回float基本类型）上的方法调用连接点时，通知必须把返回值“装箱”成Float类的一个实例，使得它可以依据通知声明，作为一个Object返回。在连接点返回的地方，依据方法的原始签名，会自动并且透明地把返回值取消装箱回基本的float。

在使用around()通知时，性能是需要考虑的事项。在AspectJ中使用around()通知时会有性能损失，这意味着只要有可能，就要结合使用before()和after()返回形式的通知。

参考

Ganesh Sittampalam等人所著的论文“测量AspectJ程序的动态行为（Measuring the Dynamic Behaviour of AspectJ Programs）”（可以在*http://www.sable.mcgill.ca/publications/papers/#oopsla2004*上找到它）讨论了一些关于AspectJ实现可能优于手工编码解决方案的性能问题；13.6节讨论了after()返回形式的通知；4.1节描述了call(Signature)切入点；9.1节描述了within(TypePattern)切入点；13.2节研究了thisJoinPoint变量及其封装的环境，讨论了一些常见的切入点术语，其中之一是能够忽略异常处理和对象初始化连接点，因为它们不支持around()通知。

13.5 在连接点之后无条件执行通知

问题

你想在特定的连接点之后执行通知，而不管连接点如何返回。

解决方案

使用after()类型的通知。

讨论

例 13-5 说明了如何在 void MyClass.foo(int,String)方法之后执行通知，而不管该方法如何返回。

例 13-5：在方法调用之后执行通知

```
public aspect AfterAdviceRecipe
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets called:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: void
   Method Parameters: an int followed by a String
   */
   pointcut callPointCut() : call(void MyClass.foo(int, String));

   // Advice declaration
   after() : callPointCut() && !within(AfterAdviceRecipe +)
   {

      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "Source Location: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

参考

4.1 节中描述了 call(Signature)切入点；9.1 节中描述了 within(TypePattern)切入点；13.2 节中研究了 thisJoinPoint 变量及其封装的环境，讨论了一些常见的切入点术语，其中之一是能够忽略异常处理和对象初始化连接点，因为它们不支持after()通知。

13.6 仅在从连接点正常返回之后才执行通知

问题

仅当特定的连接点正常返回时，你才想在该连接点之后执行通知。

解决方案

使用after() returning或after() returning(<ReturnType> <Identifier>)类型的通知。

讨论

例13-6说明了如何指定：仅当void MyClass.foo(int)方法调用正常返回时，才应该执行该方法调用之后的通知。

例13-6：仅当方法调用正常返回时，才执行该调用之后的通知

```
public aspect AfterReturningAdviceRecipe
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets called:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: void
   Method Parameters: an int followed by a String
   */
   pointcut callPointCut() : call(void MyClass.foo(int));

   // Advice declaration
   after() returning : callPointCut() && !within(AfterReturningAdviceRecipe +)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "Source Location: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

例13-6中的after() returning通知不会访问连接点的返回值，因为void MyClass.foo(int)方法不会返回任何值。如果连接点确实返回了一个值，那么AspectJ就会提供after() returning(ReturnType Identifier)变体，它允许把返回值赋予identifier()，并访问相应通知块中的值，如例13-7所示。

例13-7：如果方法正常返回，则访问该方法的返回值

```
public aspect AfterReturningValueAdviceRecipe
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets called:
```

```
Class Name: MyClass
Method Name: foo
Method Return Type: int
Method Parameters: an int followed by a String
*/
pointcut callPointCut() : call(int MyClass.foo(int));

// Advice declaration
after() returning(Object value) : callPointCut()
   && !within(AfterReturningAdviceRecipe +)
{
   System.out.println(
      "------------------- Aspect Advice Logic --------------------");
   System.out.println(
      "Source Location: "
         + thisJoinPoint.getStaticPart().getSourceLocation());
   System.out.println(
      "Value being returned: "
         + value);
   System.out.println(
      "------------------------------------------------------------");
}
```

使用after() returning(<Type> <Identifier>)通知访问基本类型（如例13-7所示）的一个有趣的副作用是：必须把基本的int值装箱进Integer类的一个实例中，以传递给通知。当通知期待的返回类型是Object类型时，并且如果返回值是基本类型，AspectJ将自动并且透明地把基本值装箱进其对应的Java类中。around()形式的通知也可以使用这种自动和透明的装箱行为，其中会期待Object类型的值，并且将从通知传递基本值或将基本值传递给通知。

与正常的after()类型的通知（13.5节中介绍过）相比，after() returning形式的通知提供了为触发通知的连接点提供了更精细的过滤器。如果遇到的连接点没有引发异常地返回，则会执行after() returning通知块内的代码，这提供了一种机制，用于避免在发生问题的方法上捕获连接点。

参考

4.1节中描述了call(Signature)切入点；9.1节中描述了within(TypePattern)切入点；13.2节中研究了thisJoinPoint变量及其封装的环境；13.4节中描述了around()通知；13.5节中说明了after()通知。

13.7 仅当连接点中引发了一个异常之后才执行通知

问题

仅当特定的连接点引发了一个异常时，你才想在该连接点之后执行通知。

解决方案

使用after() throwing或after() throwing(<ExceptionType> <Identifier>)类型的通知。

讨论

例 13-8 说明了如何指定：应该在 void MyClass.foo(int) 方法调用带有异常返回时才执行通知。

例 13-8：如果方法调用引发一个异常，则在该调用之后执行通知。

```
public aspect AfterThrowingAdviceRecipe
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets called:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: void
   Method Parameters: an int followed by a String
   */
   pointcut callPointCut() : call(void MyClass.foo(int));

   // Advice declaration
   after() throwing : callPointCut()
      && !within(AfterThrowingAdviceRecipe +)
   {
      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "Source Location: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

after() throwing通知类型完成了有关在连接点返回之后应用要执行的通知的讨论。在这种情况下，应该在发生异常时执行通知。这使开发人员能够处理执行连接点之后的异常情况。

参考

4.1节中描述了call(Signature)切入点；5.1节中更详细地说明了handler(TypePattern)切入点；9.1节中描述了within(TypePattern)切入点；13.2节中研究了thisJoinPoint变量及其封装的环境；13.5节和13.6节分别描述了after()和after() returning通知类型。

13.8 控制通知优先级

问题

你想在把多个通知块应用于同一个连接点时，控制它们的优先级。

解决方案

如果把位于不同方面中的相同类型的通知应用于同一个连接点，则可以使用declare precedence语句。declare precedence语句的语法如下：

```
declare precedence : TypePattern, TypePattern, ..;
```

如果相同类型的通知位于同一个方面中，那么在把通知应用于共享的连接点时，可以使用通知声明的位置来指示其优先级。

讨论

例13-9说明了使用declare precedence语句，来指定AspectA比AspectB具有更高的优先级。

例 13-9：使用 declare precedence 语句控制方面优先级

```
public aspect AspectA
{
   // Declare precedence rules
   declare precedence : AspectA, AspectB;

   /*
   Specifies calling advice whenever a method
   matching the following rules gets called:
```

```
    Class Name: MyClass
    Method Name: foo
    Method Return Type: * (any return type)
    Method Parameters: an int followed by a String
    */
    pointcut callPointCut() : call(void MyClass.foo(int, String));

    // Advice declaration
    before() : callPointCut() && !within(AspectA +)
    {

        System.out.println(
            "------------------- Aspect Advice Logic --------------------");
        System.out.println("In the advice of AspectA");
        System.out.println("Target: " + thisJoinPoint.getTarget());
        System.out.println("This: " + thisJoinPoint.getThis());
        System.out.println("Aspect Instance: " + AspectA.aspectOf());
        System.out.println(
            "------------------------------------------------------------");
    }
}
```

在某些应用程序中，方面可以相对孤立地工作，但是，随着面向方面体系结构变得更复杂，让两个方面通知同一个连接点是合理的。在这些情况下，在把方面的通知应用于任何共享的连接点时，必须控制方面的优先级。

在declare precedence语句中使用TypePatterns来指定不同的方面，以及它们的显式次序。可以使用通配符来指定TypePatterns，以根据需要为特定的方面集合或方面的整个包指示优先级。

当把同一个方面中的两个相同类型的通知块应用于同一个连接点时，就会发生同样的问题。在这种情况下，使用declare precedence语句没有意义，因为通知都在同一个方面中。为了处理这种情况，AspectJ基于方面声明内通知的类型和位置，来应用隐式的优先级次序。

当把同一个方面内相同类型的通知应用于同一个连接点时，存在以下隐式的优先级规则：

- 按在方面中声明before()和around()通知类型的次序，来应用它们的隐式优先级规则。如果把同一个方面中的两个before()通知块应用于同一个连接点，那么声明的第一个块将具有最高的优先级，而最后一个块将具有最低的优先级。
- after()、after() returning和around() throwing通知类型的隐式优先级的应用次序与before()和around()通知类型相反。这意味着如果把同一个方面中的两个或多个after()类型的通知块应用于同一个连接点，那么最后声明的通知块将具有最高的优先级，而第一个块将具有最低的优先级。

无论通知是在同一个方面中声明的，还是在单独的方面中声明的，通知优先级都意味着不同的事情，这依赖于通知类型。图 13-2 说明了与连接点相关的不同优先级次序，以及它们对于不同形式通知的隐含意义。

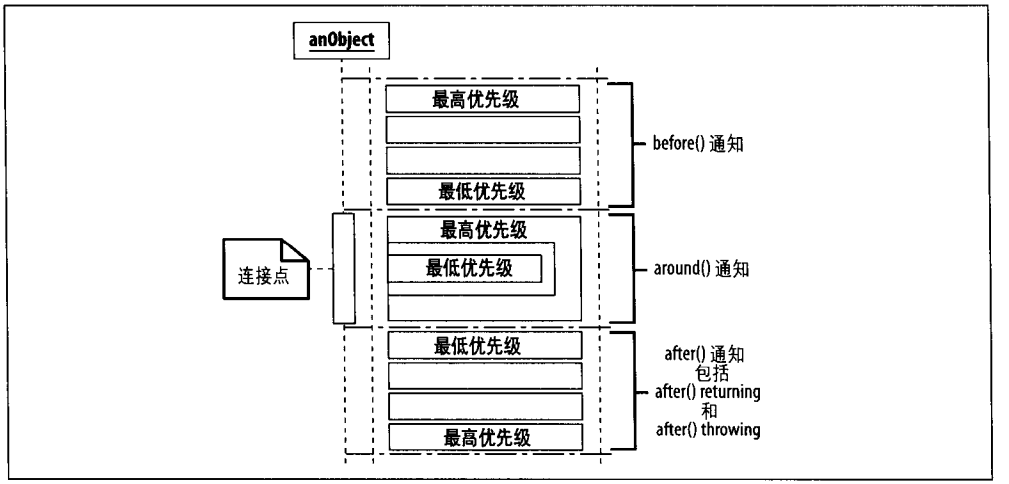

图 13-2：与连接点相关的通知优先级的次序

参考

4.1 节中介绍了 `call(Signature)` 切入点；9.1 节中描述了 `within(TypePattern)` 切入点；12.4 节中描述了 NOT（!）运算符；13.3 节中描述了 `before()` 类型的通知；13.4 节中介绍了 `around()` 类型的通知；13.5 节、13.6 节和 13.7 节分别描述了 `after()` 类型的通知及其变体。

13.9 通知方面

问题

你想把通知应用于其他方面。

解决方案

使用 AspectJ 中可用的切入点，来指定要通知的方面。

讨论

AspectJ包含可通知事情的作用域内的方面，因此，可以使用AspectJ中可用的多个切入点来指定通知与应用程序中的类相关的其他方面。例13-10说明了如何声明通知，以应用于另一个方面内的通知。

例13-10：把通知应用于其他通知

```
public aspect AdviseAspectRecipe
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets called:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: void
   Method Parameters: an int followed by a String
   */
   pointcut callPointCut() : call(void MyClass.foo(int, String));

   // Advice declaration
   before() : callPointCut() && within(AdvisedAspect +)
   {

      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice attached to the call point cut");
      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

参考

4.1节中描述了call(Signature)切入点；9.1节中描述了within(TypePattern)切入点；13.2节中研究了thisJoinPoint变量及其封装的环境。

第 14 章

定义方面实例化

14.0 简介

在第 9 章中，关于通知的秘诀暗示了 AspectJ 实现方面的一个有趣特征。在设计 AspectJ 时，决定尽可能地重用现有的Java语言。这个决定意味着即使面向方面的本质和意图是 AspectJ 上的前沿性工作，也可以通过保持在 Java 的约束内来协调这个目标，从而使得不需要专门的运行时环境。

为了实现这个目标，设计人员决定把方面实现为后处理的面向方面应用程序内的类。这种聪明的方法意味着世界上 100% 的虚拟机都可以不加更改地运行 AspectJ 程序。由于另一个原因而使得这种方法是有趣的：因为方面在织入过程中都会转换成普通类（参见第 1 章，以了解详细信息），这样，它们必须具有自己的普通对象生命周期。进一步讲，因为方面有它们自己的生命周期，它们必须有关联的实例化策略，编译器可以通过它来决定何时创建一个新的方面对象。在本章中，你将看到如何声明和控制特定的实例化策略，它们用于在应用程序流程内控制何时创建方面对象。

14.1 定义单件方面

问题

你想声明把方面实例化成一个单件（singleton）。

解决方案

使用`issingleton()`显式方面实例化策略声明，或者依靠默认的隐式方面实例化策略。

讨论

AspectJ 默认把单件行为赋予方面。你迄今为止见过的每个方面都是一个单件，因为没有显式声明实例化策略。

可以通过把 issingleton()语句显式添加到方面声明中，来显式建立单件方面实例化策略，如例 14-1 所示。

例 14-1：显式声明单件方面

```
public aspect Singleton issingleton()
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets called:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: void
   Method Parameters: an int followed by a String
   */
   pointcut callPointCut() : call(void MyClass.foo(int, String));

   // Advice declaration
   before() : callPointCut() && !within(Singleton +)
   {

      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice attached to the call point cut");
      System.out.println(
         "Target: " + thisJoinPoint.getTarget());
      System.out.println(
         "This: " + thisJoinPoint.getThis());
      System.out.println("Aspect Instance: " + Singleton.aspectOf());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

单件模式的核心目标是：声明一个类，它在应用程序的生存期内把自身约束到一个对象实例上。图 14-1 显示了如何通过应用程序中相同的方面实例来通知每个不同的对象。

单件是在整个应用程序中应用一种公共组件的优秀方式。把方面声明为单件，允许跨应用程序中应用方面的所有区域共享实例化的方面。方面实例化占据的所有内存空间跨所有对象和线程共享，如果小心使用，它还可以提供一种共享数据的有用方式。

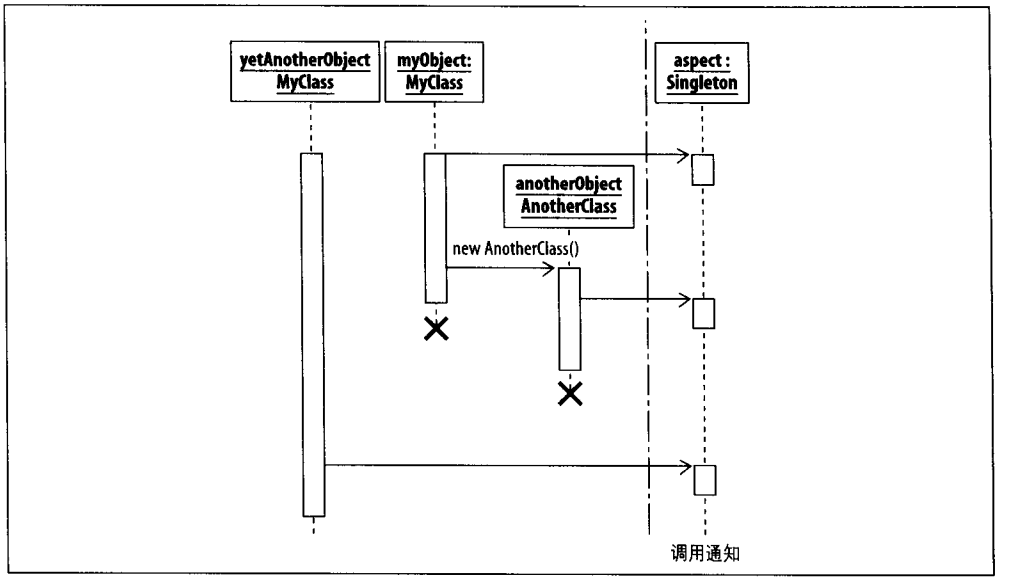

图 14-1：用同一个单件方面接收来自所有对象的调用

使用传统的面向对象单件的主要缺点是：使用单件的每个类都与单件的公共接口紧密耦合。如果单件的公共接口发生变化，那么使用单件的所有类都必须作相应的更改。这会在整个应用程序中产生复杂的更改连锁反应。

面向方面的单件方面没有这种缺点。通过把应用单件的方式与单件提供的行为模块化在一个位置，如果行为发生变化，则只需要修改方面。

参考

17.1 节更详细地讨论了单件模式；4.1 节介绍了 `call(Signature)` 切入点；9.1 节描述了 `within(TypePattern)` 切入点；12.4 节描述了 NOT（!）运算符；第 15 章更详细地描述了方面之间的继承关系。

14.2 在每个实例上定义一个方面

问题

你想在每个对象实例上声明一个要实例化的方面。

解决方案

使用perthis(Pointcut)或pertarget(Pointcut)方面实例化策略声明。

讨论

AspectJ提供了perthis(Pointcut)和pertarget(Pointcut)方面实例化策略，它们依据Pointcut定义选择的类，来声明应该为每个新的对象实例所实例化的方面。

perthis(Pointcut)声明和pertarget(Pointcut)声明之间的区别必须涉及：在到达通知的连接点时，将会检查什么对象。perthis(Pointcut)声明指定：将为通知触发连接点处的this所引用的每个新对象而实例化一个新的方面。pertarget(Pointcut)实例化策略指定：如果新对象是通知触发连接点的目标，则为每个这样的新对象实例化一个新的方面。尽管有这些细微的差别，这两种声明都会显式地把单个方面实例关联到单个对象实例上。

例14-2说明了如何把perthis(Pointcut)声明指派给特定的方面，图14-2说明了如何在应用程序的流程内实现方面实例化策略。

例14-2：使用perthis(Pointcut)声明为每个执行对象实例化一个方面

```
public aspect PerThis perthis(callPointCut())
{
   /*
   Specifies calling advice whenever a method
   matching the following rules is executed:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: void
   Method Parameters: an int followed by a String
   */
   pointcut callPointCut() : call(
      void MyClass.foo(int, String));

   // Advice declaration
   before() : callPointCut() && !within(PerThis +)
   {

      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice attached to the call point cut");
      System.out.println("Target: " + thisJoinPoint.getTarget());
      System.out.println("This: " + thisJoinPoint.getThis());
      System.out.println(
         "Aspect Instance: "
            + PerThis.aspectOf(thisJoinPoint.getThis()));
```

```
        System.out.println(
            "-----------------------------------------------------------");
    }
}
```

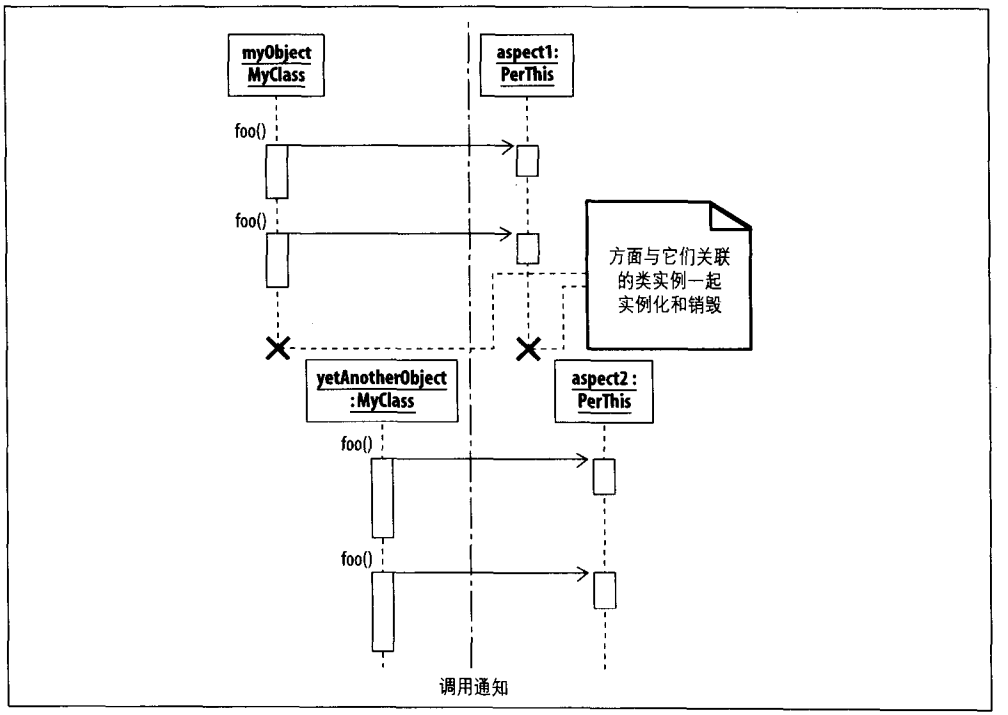

图 14-2：在每个通知的对象上创建一个方面实例

在考虑可以通过单个方面通知多个类时，perthis(Pointcut)和pertarget(Pointcut)方面实例化策略声明上的Pointcut参数提出了一些有趣的问题。例14-3说明了如何声明一个方面来通知多个类，其中perthis(Pointcut)方面实例化策略只与executeMyClassFoo()切入点指定的MyClass类的对象相关。

例 14-3：试图通知多个类，但是指定的方面实例化策略只对一个类感兴趣

```
public aspect AdviseMultipleClasses perthis(executeMyClassFoo())
{
    public pointcut executeMyClassFoo() : execution(void MyClass.foo());

    public pointcut executeAnotherClassFoo() : execution(void AnotherClass.foo());

    before() : executeMyClassFoo()
    {
        System.out.println("Advising foo");
```

```
        System.out.println("Aspect is: " + this);
    }

    before() : executeAnotherClassFoo()
    {
        System.out.println("Advising foo");
        System.out.println("Aspect is: " + this);
    }
}
```

声明只为executeMyClassFoo()切入点指定的每个新对象实例化AdviseMultiple-Classes方面，这隐式排除了其他类的对象。即使声明了executeAnotherClassFoo()切入点，并且它具有相应的通知，也不会导致把任何方面应用于除了它与executeMyClassFoo()共享的那些类之外的任何类。

在例14-3中，两个切入点之间没有共享任何公共类，因此，executeMyClassFoo()切入点及关联的通知会被忽略，因为这个切入点参与了perthis(Pointcut)实例化策略的定义，如图14-3所示。

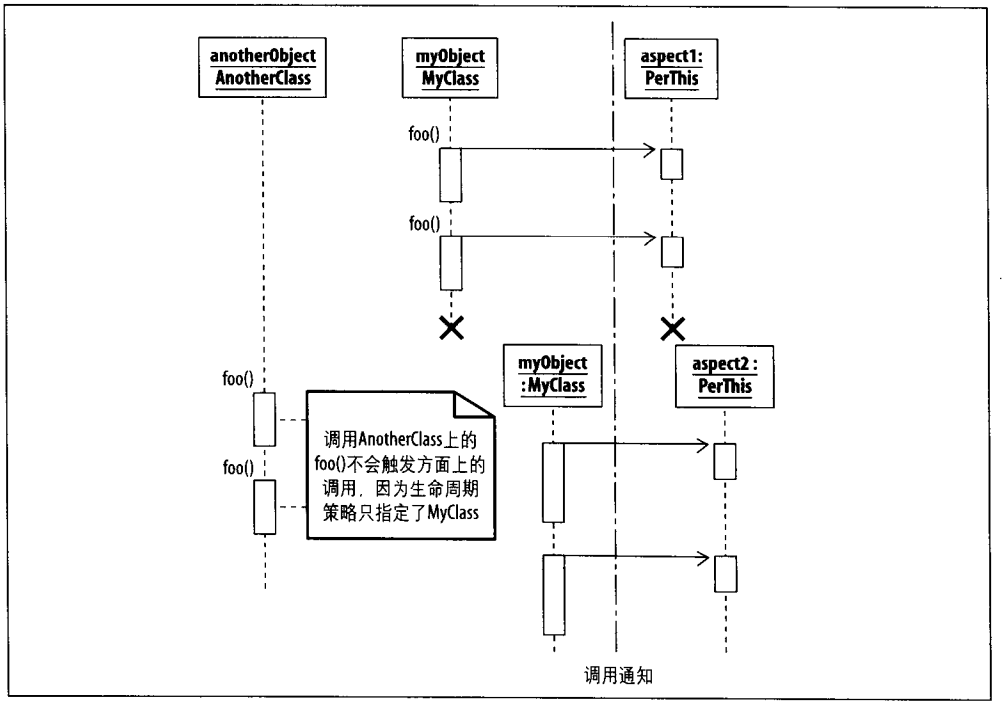

图14-3：依据方面实例化策略，从AdviseMultipleAspects方面中隐式排除AnotherClass对象

一个方面不能具有针对两类不同对象的两种实例化策略。为了确保方面的实例化策略与它通知的类的所有对象相关，一种有用的技术是：纯粹为了使用方面的实例化策略，声明一个切入点来结合方面中所有其他的切入点声明，如例 14-4 所示。

例 14-4：捕获方面通知的所有类，用以包含进方面实例化策略定义中

```
public aspect AdviseMultipleClasses perthis(applyLifecyclePolicy())
{
   public pointcut executeMyClassFoo() : execution(void MyClass.foo());

   public pointcut executeAnotherClassFoo() : execution(void AnotherClass.foo());

   public pointcut applyLifecyclePolicy() : executeMyClassFoo() ||
                                   executeAnotherClassFoo();

   before() : executeMyClassFoo()
   {
      System.out.println("Advising foo");
      System.out.println("Aspect is: " + this);
   }

   before() : executeAnotherClassFoo()
   {
      System.out.println("Advising foo");
      System.out.println("Aspect is: " + this);
   }
}
```

参考

4.4 节介绍了 execution(Signature) 切入点；7.1 节描述了 within(TypePattern) 切入点；12.3 节描述了 OR（||）运算符；12.4 节描述了 NOT（!）运算符；14.1 节描述了单件方面实例化策略。

14.3 在每个控制流程上定义一个方面

问题

你想在每个控制流程上声明一个要实例化的方面。

解决方案

使用 percflow(Pointcut) 方面实例化声明。

讨论

percflow(Pointcut)语句指示编译器，它应该为从Pointcut参数指定的连接点集合内进入的每个新控制流程创建方面的一个新实例。

例14-5说明了percflow(Pointcut)实例化声明，它用于为每个新的程序控制流程（其中会遇到通过callPointCut()指定的连接点）指定PerControlFlow方面的一个新实例。

例 14-5：使用 percflow(Pointcut)，为进入的每个新的程序控制流程创建方面的一个新实例

```
public aspect PerControlFlow percflow(callPointCut())
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets called:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: * (any return type)
   Method Parameters: an int followed by a String
   */
   pointcut callPointCut() :
      call(void MyClass.foo(int, String));

   // Advice declaration
   before() : callPointCut() && !within(PerControlFlow +)
   {

      System.out.println(
         "-------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice attached to the call point cut");
      System.out.println("Target: " + thisJoinPoint.getTarget());
      System.out.println("This: " + thisJoinPoint.getThis());
      System.out.println(
         "Aspect Instance: " + PerControlFlow.aspectOf());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

图 14-4 说明了这种关系对于 percflow(Pointcut)声明的工作方式。

percflow(Pointcut)语句代表最细粒度的方面实例化策略，并且会为特定的代码段创建最大数量的不同方面实例。利用这种方面实例化声明，方面的内存需求将变得更为重要。

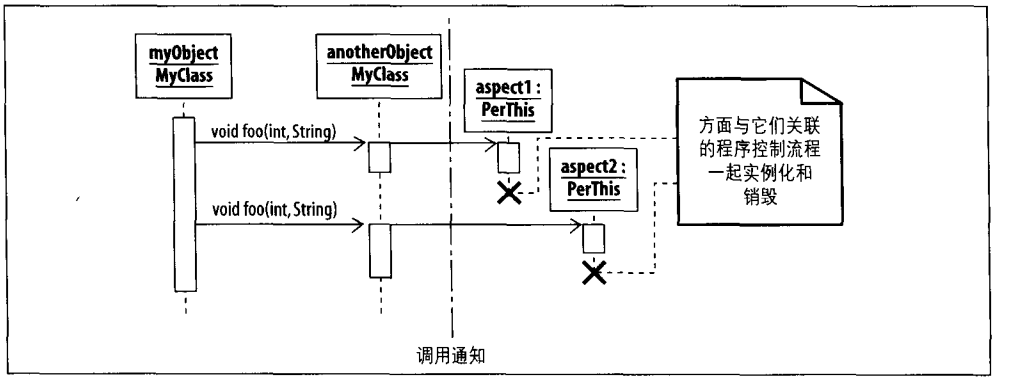

图 14-4：在每个程序控制流程上创建一个新的方面实例

参考

4.1 节介绍了 `call(Signature)` 切入点；9.1 节描述了 `within(TypePattern)` 切入点；12.4 节描述了 NOT（!）运算符。

第15章

定义方面的关系

15.0 简介

AspectJ中的方面本质上都是对象，它们可以从传统的面向对象（OO）机制受益，这种机制使得面向对象成为软件开发的一种如此优秀的方法。

在使用关联和复合这些OO概念时，针对方面的规则将相当简单：一个方面无需使用任何独特的语法，即可包含或使用其他的类。如果忽略一些不规则的方法签名，类就可以在使用方面时，把它看作就好像是另一个类一样。

本章论及了方面和类之间的继承关系的细节，以及在定义这些关系时要考虑的一些特殊事情。

15.1 继承切入点定义

问题

你想在方面中创建切入点定义，然后可以通过继承来重用它们。

解决方案

创建一个抽象类。使用合适的`public`、`protected`或`default`访问修饰符在抽象方面内定义可重用的切入点逻辑。最后，把抽象方面继承进子方面中，以重用声明的切入点。

讨论

在使用继承时，切入点会受到与正常方法相同的一组规则的支配。如果把切入点声明为public、protected或default，那么就可以继承其逻辑，并在其他方面中重用它。

例 15-1 显示了一个抽象方面，它为继承其定义的切入点提供了一个基础。

例 15-1：使用抽象方面定义可重用的切入点逻辑

```
public abstract aspect BasePointcutDefinitionsAspect
{
   public pointcut callPointcut() : call(void MyClass.foo(int, String));
}
```

注意： 在例15-1中，并没有明确要求BasePointcutDefinitionsAspect是抽象的。由于AspectJ语言指示只有抽象方面才可以被子方面扩展，所以把方面声明为抽象的。

例15-2显示了ReusePointcutsRecipe方面，它从例15-1的方面中继承了callPointcut()声明。

例 15-2：重用抽象基础方面中的切入点逻辑

```
public aspect ReusePointcutsRecipe extends BasePointcutDefinitionsAspect
{
   // Advice declaration
   before() : callPointcut() && !within(ReusePointcutsRecipe +)
   {

      System.out.println(
         "------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice attached to the call point cut");
      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

图 15-1 显示了例 15-1 和例 15-2 中定义的继承关系的静态结构。

定义只封装切入点逻辑的抽象方面的能力提供了创建切入点库的可能性。在大型企业环境中，这个强大的概念是有用的。可以为整个应用程序中的织入模式定义特定的切入点，并重用它们，而不必一遍又一遍地反复重写通用的切入点逻辑。

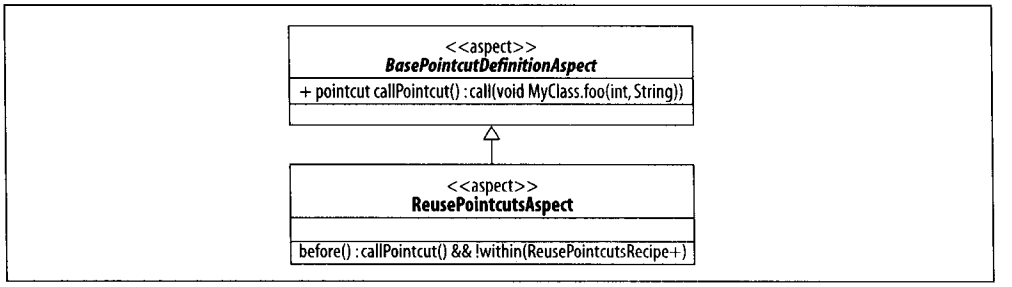

图 15-1：方面之间的继承关系

参考

4.1 节说明了 call(Signature) 切入点的定义；9.1 节描述了 within(TypePattern) 切入点；12.4 节描述了 NOT（!）运算符。

15.2 实现抽象切入点

问题

你想声明并引用一个切入点，但是把切入点逻辑的定义留给特殊的子方面实现。

解决方案

在声明切入点和周围的方面时，使用abstract关键字，并且不要提供任何切入点逻辑。

讨论

例 15-3 说明了在整个方面中声明和使用的抽象切入点。

例 15-3：声明抽象切入点，通过派生方面来实现它们

```
public abstract aspect BaseAbstractAspect
{
   /*
   Specifies an abstract pointcut placeholder
   for derived aspects to specify
   */
   public abstract pointcut abstractBasePointcut();

   /*
    Specifies calling advice whenever a jhin point
    picked by the abstractBasePointcut (specified
```

```
    by specialized aspects) is encountered, and not within
    this aspect or any inheriting aspects.
    */
   pointcut runAdvicePointcut() : abstractBasePointcut()
      && !within(BaseAbstractAspect +);
}
```

例 15-4 说明了如何在子方面中充分定义例 15-3 中声明的抽象切入点。

例 15-4：实现继承的抽象基础切入点的完整切入点逻辑

```
public aspect AbstractImplementationAspect extends BaseAbstractAspect
{
   /*
   Specifies calling advice whenever a method
   matching the following rules gets called:

   Class Name: MyClass
   Method Name: foo
   Method Return Type: void
   Method Parameters: an int followed by a String
   */
   public pointcut abstractBasePointcut() : call(void MyClass.foo(int, String));

   // Advice declaration
   before() : runAdvicePointcut()
   {

      System.out.println(
         "-------------------- Aspect Advice Logic --------------------");
      System.out.println(
         "In the advice attached to the call point cut");
      System.out.println(
         "Signature: "
            + thisJoinPoint.getStaticPart().getSignature());
      System.out.println(
         "Source Line: "
            + thisJoinPoint.getStaticPart().getSourceLocation());
      System.out.println(
         "------------------------------------------------------------");
   }
}
```

图 15-2 显示了例 15-3 和例 15-4 中的方面之间的继承关系。

如果想推迟定义特定的切入点逻辑，同时又希望仍然声明更通用和潜在可重用的切入点，那么抽象切入点声明就是有用的。使用这种方法，可以创建通用的抽象方面，可以为特定的应用程序特别地扩展它们。这可能导致可重用的方面库。

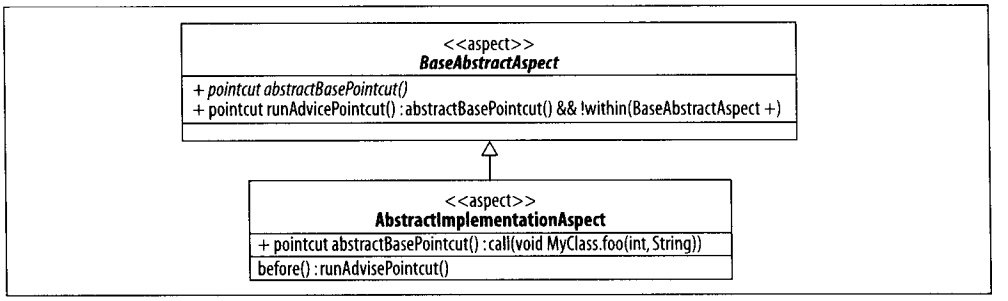

图 15-2：实现抽象切入点

参考

4.1 节说明了 call(Signature) 切入点的定义；9.1 节描述了 within(TypePattern) 切入点；12.4 节描述了 NOT（!）运算符；第 17~19 章在实现面向对象设计模式时，使用了本示例中展示的技术来定义可重用的方面。

15.3 把类继承进方面中

问题

你想通过继承现有类的行为，把现有类重构成一个方面。

解决方案

使用 extends 关键字来声明方面扩展类。

讨论

例 15-5 显示了一个伪日志记录类，它代表一种现有的日志记录机制。其目标是：重构对应用程序中日志记录类的所有现有的调用，并把日志记录模块化进一个方面中，它可以更灵活地织入进应用程序中。

例15-5：伪传统的日志记录类代表在应用程序内为方面进行重构之前的一个现有的日志记录类

```
public class OOLogging
{
   public void logEntry(String entry)
   {
```

```
        System.out.println("Entry logged: " + entry);
    }
}
```

一旦从应用程序中重构了对类的现有调用，就可以创建一个方面，把日志记录重新应用于应用程序，并通过继承重用原始日志记录类，如例 15-6 所示。

例 15-6：使用传统类的继承机制，来重用被重构到一个方面中的行为

```
public aspect AOLogging extends OOLogging
{
    /*
    Specifies calling advice whenever a method
    matching the following rules gets called:

    Class Name: MyClass
    Method Name: foo
    Method Return Type: void
    Method Parameters: an int followed by a String
    */
    pointcut callPointcut() : call(void MyClass.foo(int, String));

    // Advice declaration
    before() : callPointcut()
        && !within(AOLogging +)
        && !within(OOLogging)
    {
        this.logEntry(thisJoinPoint.toShortString());
    }
}
```

注意： 重构是一个术语，用于以某种方式更改现有的设计，从而改进它。重构不一定会在解决方案中纳入任何新特性；相反，它会重新构造软件，以提供一种可以更轻松地维护并且更优雅的设计，或者它会纳入其他设计改进。Martin Fowler 所著的 Refactoring: Improving the Design of Existing Code（Addison-Wesley）是迄今为止关于重构的权威著作。

图 15-3 显示了在声明例 15-6 中所示的方面之后的静态结构。

尽管方面可以使用继承来重用类行为，但这有时可能是一种脆弱的方法。一种更优雅的解决方案是：通过聚合纳入日志记录类作为日志记录方面的一个属性，来执行重用。然后可以针对目标应用程序织入日志记录方面，并有效地把日志记录行为委托给日志记录类的内部实例。

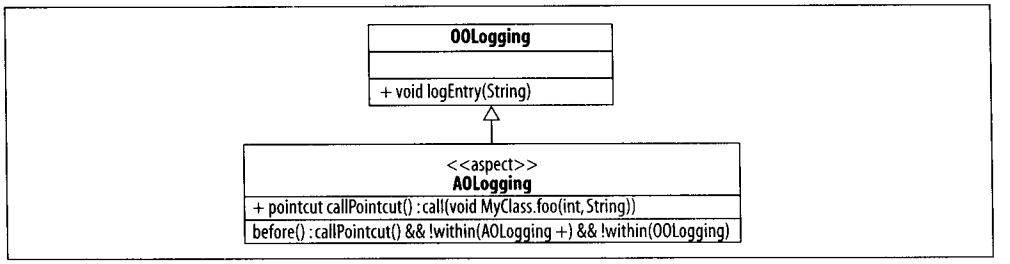

图 15-3：使用方面和类之间的继承关系来应用重用

参考

4.1 节说明了 call(Signature) 切入点的定义；9.1 节描述了 within(TypePattern) 切入点；12.4 节描述了 NOT（!）运算符；经常可以使用内部方面来实现重构，这将在 15.4 节中讨论。

15.4 在类内部声明方面

问题

你想声明一个内部类级的横切关注点。

解决方案

使用内部方面声明。

讨论

例 15-7 说明了如何定义一个内部方面，以应用于它的包含类。call(TypePattern) 切入点用于捕获 MyClass 类的所有调用上的连接点。

例 15-7：声明一个内部方面

```
public class MyClass
{
   /*
      Specifies calling advice whenever a method
      matching the following rules gets called:

      Class Name: MyClass
      Method Name: *
      Method Return Type: * (any return type)
```

```
      Method Parameters: .. (any parameters0
      */
   private static aspect CallMethods
   {
      pointcut callPointCut() : call(* MyClass.* (..));

      // Advice declaration
      before() : callPointCut() && !within(CallMethods +)
      {

         System.out.println(
            "--------------- Aspect Advice Logic ----------------");
         System.out.println(
            "In the advice attached to the call point cut");
         System.out.println(
            "Signature: "
               + thisJoinPoint.getStaticPart().getSignature());
         System.out.println(
            "Source Line: "
               + thisJoinPoint.getStaticPart().getSourceLocation());
         System.out.println(
            "----------------------------------------------------");
      }
   }

   public void foo(int number, String name)
   {
      System.out.println("Inside foo (int, String)");
   }

   public void bar(String name)
   {
      System.out.println("Inside bar (String)");
      this.privateMethod();
   }

   private void privateMethod()
   {
      System.out.println("In privateMethod ()");
   }

   public static void main(String[] args)
   {
      // Create an instance of MyClass
      MyClass myObject = new MyClass();
      // Make the call to foo
      myObject.foo(1, "Russ");
      // Make the call to bar
      myObject.bar("Kim");
   }
}
```

横切行为通常被看作具有系统级的作用域，但是类可能在内部展示横切关注点。跨类中一组方法的任何公共行为都可以作为横切关注点的候选，特别是当它不是业务逻辑中类目标的核心时则更是如此。

内部方面具有3个关键特征：

1. 必须显式地把内部方面声明为静态的，因为它们的实例化受到方面实例化策略控制，而不受周围类构造的控制。
2. 内部方面可能具有应用于它们的所有规则的访问修饰符，从而允许在合适时重用封装保护和静态继承机制。
3. 内部方面并不限制于针对周围的类进行织入。通过声明一个内部方面，将暗示该方面与周围的类密切相关，但是并不会像那样约束方面。

在重构现有的代码时，内部方面是有用的。第一步是确定本质上是横切的行为，即它会影响类中的多个区域，并且自然不会影响类的基本业务目标。

然后可以声明一个或多个内部方面，用于模块化这种横切行为。在应用程序中的内部方面之间可能存在一些共性，因此下一阶段可能是把内部方面重构进更传统的系统级方面中。

参考

4.1节说明了`call(Signature)`切入点的定义；9.1节描述了`within(TypePattern)`切入点；12.4节描述了NOT（!）运算符；13.3节解释了`before()`形式的通知；第14章介绍了方面实例化策略；15.3节说明了更多的重构考虑事项和方法；Ramnivas Laddad的一系列文章（可在*http://www.theserverside.com/articles/article.tss?l= AspectOrientedRefactoringPart1/1*上找到）提供了关于面向方面重构的更多信息。

第 16 章

增强类和编译器

16.0 简介

本章说明了如何使用静态横切技术，以静态方式使用 AspectJ 中的方面把行为和接口引入现有的类中。使用这些技术，可以扩展类来实现接口，从新的父类扩展类，引入新的方法和属性，减轻所发生异常的影响，以及继承多个基类。

静态横切技术非常强大，必须小心使用。可以通过语言约束（如多重继承复杂性）谨慎地避免问题，并且在使用语言约束之前必须考虑到这些问题。人们在这个问题上分成两派。有些人坚持开发人员应该决定他们是否想使用更复杂的技术，从而接受潜在的问题。另外一些人（如 Java 设计者们）则试图使用语言尽可能地制约这些决定。

最终，个人的观点或组织的风格确定了静态横切技术是否是可接受的。不过，知道方面开发人员工具箱中存在这些技术是有用的。

16.1 扩展现有的类

问题

你想扩展现有的类。

解决方案

声明要添加到方面内现有类中的新方法和属性。

讨论

例 16-1 说明了如何把属性和方法引入到 MyClass 类中。

例 16-1：把属性和方法添加到现有的类中

```
public aspect ExtendClassRecipe
{

   private int MyClass.newVariable = 20;

   public int MyClass.bar(String name)
   {
      System.out.println("In bar(String name), name:" + name);
      return this.newVariable;
   }
}
```

图 16-1 显示了在应用例 16-1 中的方面之前和之后 MyClass 类的类结构。

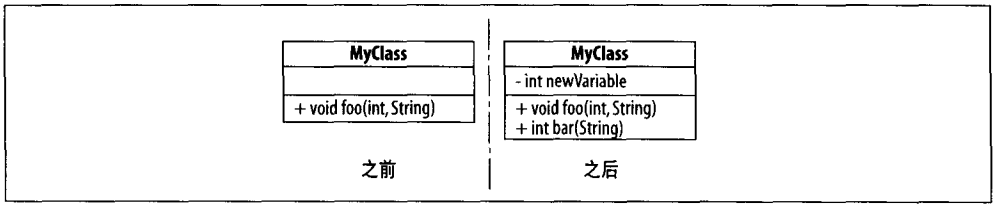

图 16-1：在方面引入新的属性和方法之前和之后的 MyClass 类

参考

16.2 节说明了如何指定类之间新的继承关系；16.3 节说明了如何使用方面把新的接口应用于现有的类。

16.2 声明类之间的继承关系

问题

你想引入两个类之间新的继承关系。

解决方案

使用 declare parents 语句，指定特定的类是从另一个类扩展而来。

讨论

例 16-2 说明了如何为 MyClass 类指定新的继承关系。

例 16-2：在类之间添加新的继承关系

```
public aspect IntroduceInheritanceRecipe
{
   declare  parents : MyClass extends AnotherClass;
}
```

图 16-2 显示了在应用例 16-2 中的方面之前和之后 MyClass 类的类结构。

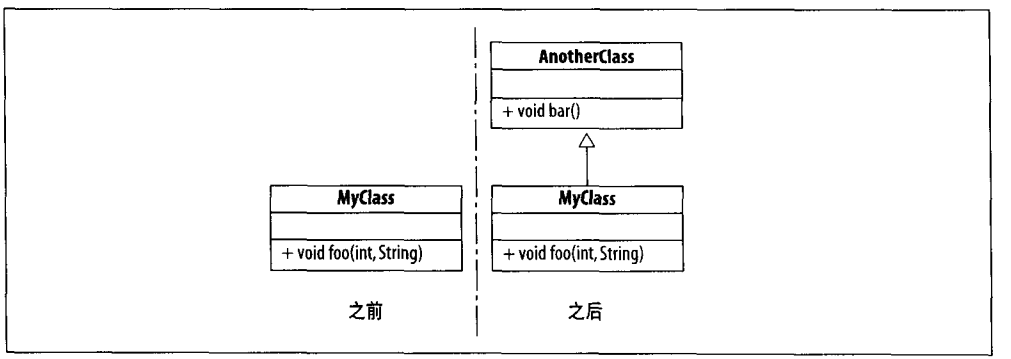

图 16-2：在方面引入新的继承关系之前和之后的 MyClass 类

参考

16.3 节说明了如何使用方面把新的接口应用于现有的类。

16.3 使用方面实现接口

问题

你想把接口添加给类。

解决方案

使用 declare parents 语句，指定特定的类实现特定的接口。

讨论

例 16-3 说明了 AspectJ 中的 declare parents 语句如何把一个接口应用于一个类，在应用方面之前，这个类可能对该接口一无所知。

例 16-3：使用方面在现有类上声明一个新接口

```
public aspect ImplementInterfaceRecipe
{
   declare parents : MyClass implements MyInterface;
}
```

图 16-3 显示了通过应用例 16-3 中的方面，将会如何影响静态类体系结构。

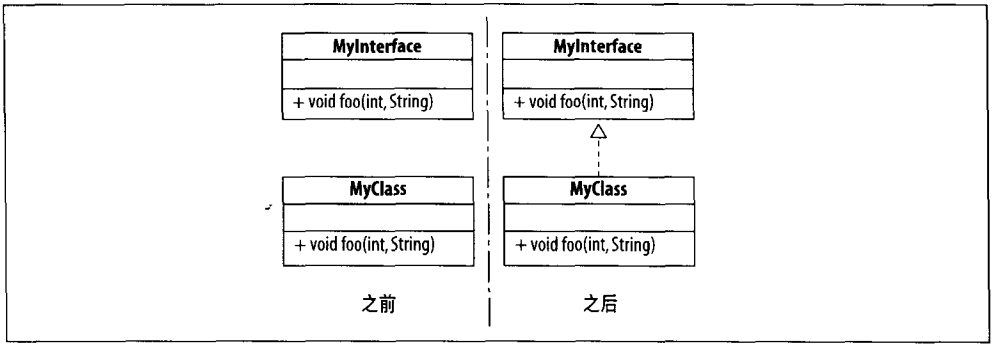

图 16-3：在方面引入新接口之前和之后的静态类结构

把接口应用于现有类的能力允许通过接口类型来引用那个类的对象，如例 16-4 所示。

例 16-4：依据声明接口的方面来引用对象

```
// Create an instance of MyClass
MyInterface myObject = new MyClass();

// ...

// Work with the interface reference
myObject.foo(1, "Russ");
```

通过把新接口应用于类，就可以用不同于原来设计的方式使用这个类。当赋予类新角色时，这特别有用。

类必须实现接口声明的任何方法。在传统的面向对象技术中，Adapter 设计模式是一种良好的解决方案。不过，AspectJ 确实提供了 Adapter 模式的一种替代实现。使用 16.1 节说明的机制，方面无需使用单独的 Adapter，即可把接口方法的实现添加到类中。

参考

16.1 节说明了如何用新方法扩展现有的类；第 17~19 章给出了在实现 OO 设计模式时，如何把角色赋予类的多个示例；18.3 节讨论了用于实现 Adapter 设计模式的面向方面的方法；23.2 节说明了用于面向方面角色声明的通用设计模式。

16.4 声明默认的接口实现

问题

你想为接口提供一种默认的实现。

解决方案

在方面内为接口的方法声明默认的实现。

讨论

例 16-5 说明了如何通过实现 `void bar(String)` 方法，为 `MyInterface` 接口提供一种默认的实现。

例 16-5：使用方面提供接口方法的默认实现

```
public aspect DefaultInterfaceImplementationRecipe
{
   declare parents : MyClass implements MyInterface;

   // Declare the default implementation of the bar method
   public void MyInterface.bar(String name)
   {
      System.out.println("bar(String) called on " + this);
   }
}
```

图 16-4 显示了通过应用例 16-5 中的方面如何影响静态类结构。

图 16-4 有点误导性，它是应用例 16-5 之后所得到的结果。在织入和编译方面时，将不会看到 `DefaultImplementation` 类。不过，出于形象表示体系结构的目的，这幅图展示了类似的结果，只不过 `void bar(int,String)` 是透明地添加到 `MyClass` 类中的。

最终得到的结果与多重继承有些相似。Java 中的接口提供了一种方式，类可以通过它拥有一个显式声明的公共接口，或者它可以用于之的角色。这允许通过单个类实现多个接口，因此没有要关心的方法实现。传统上，Java 类只可以显式继承自另外一个类，这阻止了多重继承。对于 AspectJ 中的方面则不是这样。

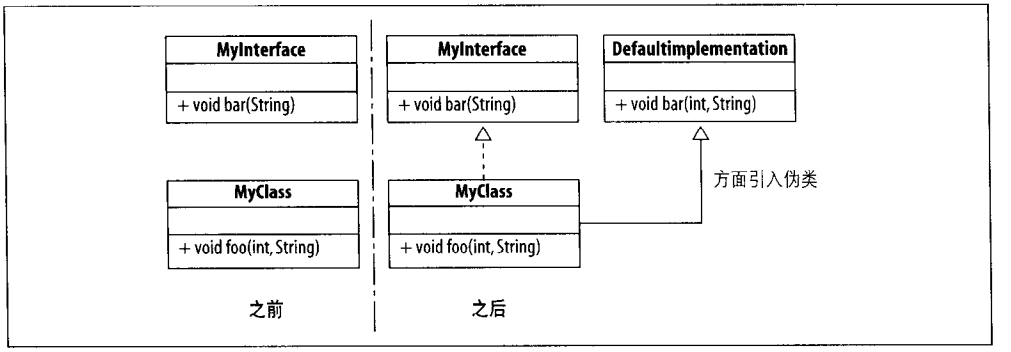

图 16-4：在应用方面以引入默认的接口实现之前和之后的静态类结构

Java接口可以具有通过方面声明其方法的实现，并且仍被声明为一个接口，从而允许多重继承形式。良好的候选情况出现在应用传统的OO设计模式时，在这种情况下，它用于提供接口的默认实现。设计模式可以是应用程序设计的一种侵略性的元素。当类的业务逻辑与应用的模式逻辑之间的界限变得模糊不清时，这可能并不总是一个好的结果。只要有可能，可能总是应该作为方面应用模式，根据需要纳入合适角色的默认实现，以及使类中的业务逻辑保持清晰并且是可管理的。

许多传统的设计模式使用抽象类作为它们使用的不同角色的基础。在Java中，只允许有一个基类，这意味着清楚表达类之间的业务关系的愿望与应用良好实践设计模式的愿望之间会发生冲突，这可能会使得体系结构变得复杂。AspectJ 通过提供为接口指定默认实现的方式，消除了把抽象类作为许多设计模式基础的需要。

参考

16.1 节解释了如何通过引入新方法和属性的方面扩展类；第 17~19 章研究了 OO 设计模式，以及这些模式通过使用面向方面的方法可以获得的好处。

16.5 减轻异常的影响

问题

你想指定一组应该减轻其影响的异常 —— 也就是说，在通过特定连接点选择的连接点上引发这些异常时，将其转换成未捕捉异常（uncaught exception）。

解决方案

使用 declare soft 语句。

讨论

例 16-6 说明了减轻在 void foo() 方法上引发的 ExceptionA 异常的影响，使得该方法的用户不必关心如何处理那个异常。

例 16-6：减轻方法调用上的异常的影响

```
public aspect SoftenExceptionRecipe
{
   pointcut callPointCut() : call(void MyClass.foo());

   declare soft : ExceptionA : callPointCut();
}
```

Java 支持两类异常：检查异常和未检查（运行时）异常。检查异常从 java.lang.Exception 扩展而来，并且由于以下原因而得此名：如果它们被声明为由特定的方法引发，那么调用方法必须显式处理这些异常，否则编译器将会发出抱怨。运行时异常从 java.lang.RuntimeException 或 java.lang.Error 扩展而来，可以在应用程序生存期内的任意时间引发，但是不需要显式捕捉。

减轻异常的影响涉及：把在特定连接点上引发的检查异常包装在 org.aspectj.lang.SoftException 运行时异常类的一个实例中。在减轻 ExceptionA 异常的影响之前，不得不捕捉或重新抛出它，如例 16-7 所示。

例 16-7：如果没有减轻异常的影响，则需要异常处理

```
...

try
{
   myObject.foo();
}
catch (ExceptionA ea)
{
   ea.printStackTrace();
}

...
```

通过把检查异常转换成运行时异常，无需异常处理即可使用类，如例 16-8 所示。

例 16-8：当减轻异常的影响时，不需要异常处理

```
...
myObject.foo();
...
```

声明应该以这种方式减轻异常的影响，使得周围的方法不必关心捕捉这些异常。尽管减轻了异常的影响，它仍然可以被引发，并且最有可能发生的情况是：直至它到达产生类似于例 16-9 所示输出的周围系统时，才会捕捉它。

例 16-9：被减轻影响的异常的输出，在这个秘诀的示例应用程序中将会抛出这个异常，但不会捕捉它

```
Exception in thread "main" org.aspectj.lang.SoftException
    at MyClass.main(MyClass.java:14)
Caused by: ExceptionA
    at MyClass.foo(MyClass.java:7)
    ... 1 more
```

参考

附录中更详细地介绍了 AspectJ 语言 API，包括 `SoftException` 类；5.1 节中介绍了当捕捉了异常的类型时，使用 `handler(TypePattern)` 切入点来捕获连接点。

16.6 扩展编译

问题

你想扩展编译器的能力，以强制执行应用程序特定的规则。

解决方案

分别使用 `declare error` 或 `declare warning` 语句，指定应该引发编译器错误或警告的条件。

讨论

AspectJ 支持通知大部分 Java 系统，但是，也许最有趣、最古怪的构造是 Java 编译器。例 16-10 说明了如何声明一个新的错误和警告，如果在正在编译的应用程序内发现指定的条件，编译器就会引发该错误和警告。

例 16-10：使用方面来声明可以通过编译器引发的新警告和错误

```
public aspect CompilationAdviceRecipe
{
   declare error : call(void ProtectedAccessClass.setValue(int))
      && !this(MyClass)
      : "Must only set the ProtectedAccessClass.value from a MyClass object";

   declare warning : call(int ProtectedAccessClass.getValue())
      && !this(MyClass)
      : "Should only be reading ProtectedAccessClass.value from a MyClass object";
}
```

无论允许的 Java 构造是什么以及应该如何使用它们，AspectJ 都能够强制执行新规则，当潜在地创建一个其需求可能发生变化的应用程序(依赖于其目标环境以及允许它展示什么功能）时，AspectJ 的这种能力是有用的。这允许定义系统策略警告，以及指导开发人员的实现。

警告：错误和警告消息必须是简单的、声明的字符串值。不能在这些消息内引用软件体系结构的任何其他部分，甚至不能引用静态结构部分。

参考

17.5 节说明了如何使用这种形式的警告，来指示开发人员破坏了 Builder 设计模式的规则；有关使用 declare error 和 declare warning 语句来强制执行设计约束的现实示例和一般性的开发人员指南，可以查看aTrack错误跟踪项目，可在*https://atrack.dev.java.net*上找到它。

第 17 章

实现创建型面向对象设计模式

17.0 简介

随着 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 所著的开创性书籍 *Design Patterns: Elements of Reusable Object-Oriented Software*（Addison-Wesley）的出版，现在著名的 Gang of Four（GoF）设计模式正式被公认为面向对象（OO）软件开发中有用的准则。

GoF 图书把设计模式分成 3 类：创建型（Creational）、结构型（Structural）和行为型（Behavioral）。原始的GoF设计模式被设计成使用大多数OO语言中可用的机制来实现。当使用AspectJ 实现面向方面时，它会添加一些新的机制，可以通过这些机制应用模式：

- 可以模块化涉及设计模式机制的代码，将其从业务逻辑的其余部分中分离出来，从而降低侵入性。
- 在 Java 中，必须小心使用类之间的继承关系，因为 Java 只允许两个类之间的单一继承关系。方面可以提供一些机制，通过这些机制，可以独立于类之间的任何核心业务关系来应用更通用的面向模式的关系。因此，方面可以消除对抽象基类的需要，使得类可以尽可能自由地为它们的业务逻辑定义适当的关系，而不会有设计模式的妨碍。
- 上述两种优点的一个副作用是：由于把模式模块化进方面中，从而从业务逻辑类中删除了以模式为中心的关系和逻辑，这使得代码更清晰，并且更易于理解。

GoF 书籍中的全部 3 类设计模式都可以从面向方面的实现中获益，本章重点介绍创建型模式。

注意： 下面3章中描述的模式实现基于Jan Hannemann和Gregor Kiczales为关于Object-Oriented ProProgramming, Systems, Languages, and Applications (OOPSLE)的第 17 届年度 ACM 大会所撰写的作品。可以到 *http://www.cs.ubc.ca/~jan/AODPs/* 上查看原始的研究成果。

17.1 实现单件模式

问题

你想使用 AspectJ 应用单件模式（singleton pattern）。

解决方案

单件模式允许把类定义成具有应用程序内的一个运行时实例。通常，通过不给特定类提供默认构造函数（它通常被重写为受保护的构造函数）来实现单件，因此，应用程序不能直接实例化类的对象。通常，通过创建一个返回类的单个实例的静态方法，来实现对单件对象的访问。

例 17-1 显示了一个抽象方面，它使用 Director（导向者）面向方面的设计模式（参见第 23 章），为应用单件模式提供一个通用基础。

例 17-1：使用方面来定义单件模式

```
public abstract aspect SingletonPattern issingleton()
{
   private Hashtable singletons = new Hashtable();

   public interface Singleton
   {
   }

   public interface NonSingleton
   {
   }

   // Pointcut to define specify an interest in all creations
   // of all Classes that extend Singleton
   pointcut selectSingletons() : call((Singleton +).new (..));

   // Pointcut to ensure that any classes in the Singleton inheritance tree
   // that are marked as Non Singletons are not included in the Singleton
   // logic.
   pointcut excludeNonSingletons() : !call((NonSingleton +).new (..));

   Object around() : selectSingletons() && excludeNonSingletons()
   {
      Class singleton = thisJoinPoint.getSignature().getDeclaringType();
```

```
        synchronized(singletons)
        {
            if (singletons.get(singleton) == null)
            {
                singletons.put(singleton, proceed());
            }
        }

        return (Object) singletons.get(singleton);
    }
}
```

讨论

SingletonPattern抽象方面定义了两个角色：Singleton和NonSingleton。这些角色被实现为接口，使得抽象方面可以处理单件，而无需关注实现细节。

图17-1显示了SingletonPattern抽象方面的结构，其中带有它定义的接口和行为，用于支持单件设计模式。

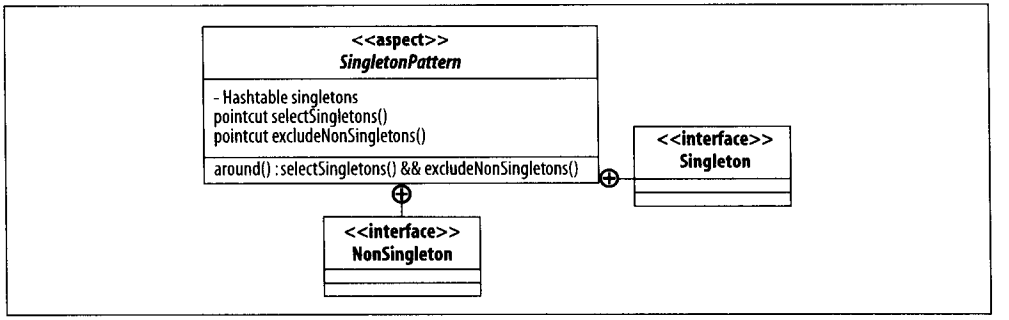

图17-1：SingletonPattern 抽象方面的结构

Singleton接口由SingletonPattern抽象方面的子方面应用于目标应用程序内将被视作单件的任何类。类似地，NonSingleton接口被应用于可能通过继承从其父类获得单件行为的类。如果决定子类不是一个单件，那么就可以使用NonSingleton接口，使得重写父类的单件特征。

声明两个切入点来捕获何时实例化具有Singleton接口的类。selectSingletons()切入点定义用于选择对扩展Singleton接口的类上构造函数的调用。为了支持单件子类的关闭单件行为的要求，声明了excludeNonSingletons()切入点。当需要阻止超类的单件行为影响子类时，可以通过单定的方面重写这个切入点。

around()通知用于捕获对应用了Singleton接口的类上构造函数的调用。around()通知会重写构造函数，以检查尚未创建正在实例化的对象的类型。

使用thisJoinPoint变量提供的类信息，在单件散列表上查询正在创建的对象的类。如果类的类型在散列表中不存在，那么就添加它的类，并通过调用proceed()来构造那个类的一个对象，这会执行原始的构造函数逻辑。proceed()调用会返回构造的对象，并把这个对象与类对象一起添加到散列表中。

如果散列表中存在类的类型，那么就不需要创建新对象。依据它的类从散列映射检索单件对象，并作为构造函数调用的结果从around()通知返回这个对象。

默认情况下，AspectJ 中的方面是一个单件。SingletonPattern 方面显式的issingleton()实例化策略，来突出显示这个方面的重要的行为特征。

注意： 这个秘诀中单件模式的实现是线程安全的，因为在around()通知执行期间，为简单起见将使用同步块锁定散列表。在第一次访问和创建单件时，这会导致性能损失，但是，这意味着之后可以在多线程应用程序中应用模式。更高效的机制可能是：把ThreadLocal类的一个实例用作方面内的一个变量，以确保只会为单个线程执行一次锁定检查，如 Brian Goetz 所著的文章中所讨论的那样，可以在 *http://www.javaworld.com/javaworld/jw-11-2001/jw-1116-dcl.html* 上找到该文章。

例 17-2 说明了如何把 SingletonPattern 抽象方面应用于特定的应用程序。

例 17-2：把 SingletonPattern 抽象方面应用于目标应用程序类

```
public aspect PrinterSingleton extends SingletonPattern
{
   declare parents: Printer implements Singleton;

   declare parents: SpecializedPrinter implements NonSingleton;
}
```

图 17-2 显示了 PrinterSingleton 方面如何影响应用程序类。

图 17-3 显示了 Printer 类的新的单件行为在示例应用程序中的表现方式。

参考

13.4节提供了关于proceed()调用及其在around()通知内的应用的更多详细信息；7.1节描述了在捕获对类构造函数的调用时的call(Signature)切入点；第14章中提供了关于方面实例化策略的更多信息；23.3 节研究了 Director 面向方面的设计模式。

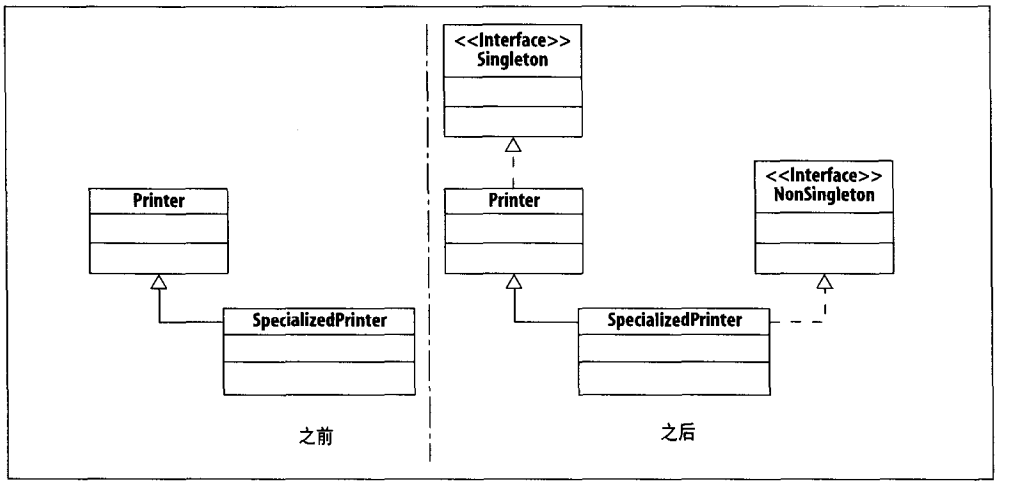

图 17-2：PrinterSingleton 方面如何影响静态应用程序结构

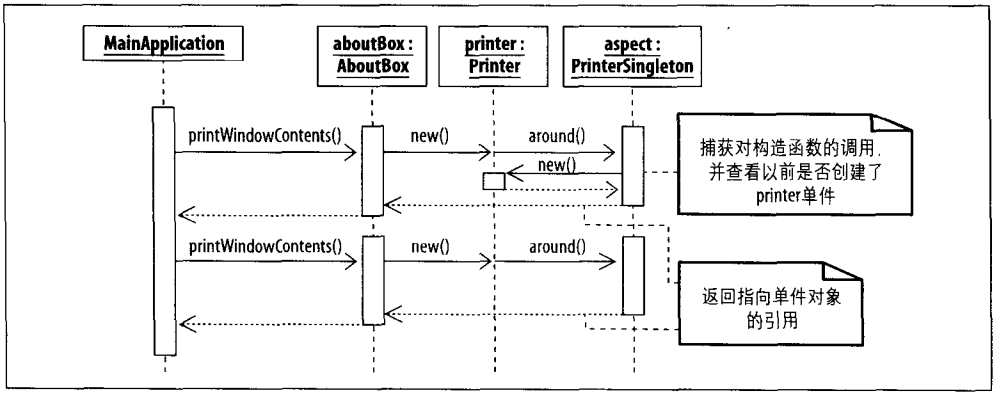

图 17-3：使用 Printer 类的新的单件行为

17.2 实现原型模式

问题

你想使用 AspectJ 应用原型模式（prototype pattern）。

解决方案

原型模式支持基于原始对象（即原型）创建复制的对象。

例 17-3 使用Director面向方面的设计模式（参见第23章），来定义应用程序模式所需的通用行为。

例 17-3：使用方面定义原型模式

```
public abstract aspect PrototypePattern
{
   protected interface Prototype
   {
   }

   public Object Prototype.clone() throws CloneNotSupportedException
   {
      return super.clone();
   }

   public Object cloneObject(Prototype object)
   {
      try
      {
         return object.clone();
      }
      catch (CloneNotSupportedException ex)
      {
         return createCloneFor(object);
      }
   }

   protected Object createCloneFor(Prototype object)
   {
      return null;
   }
}
```

讨论

PrototypePattern抽象方面定义了Prototype接口，它可以应用于目标应用程序中将作为原型的任何类。可以利用 clone()方法来扩展这些类，以支持原型复制。

图17-4显示了PrototypePattern抽象方面的结构，其中具有它定义的接口和行为，用于支持原型设计模式。

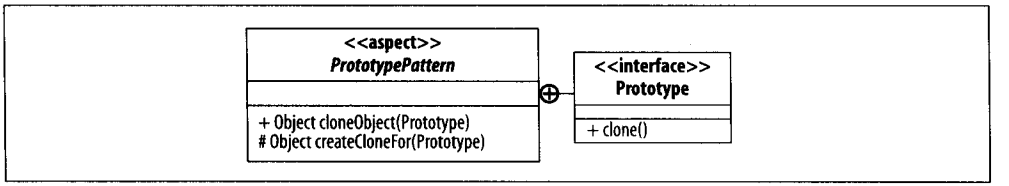

图 17-4：PrototypePattern 抽象方面的结构

clone()方法是用于实现对象深复制的Java机制。有些基类可能不支持被复制；PrototypePattern方面提供了createCloneFor(Prototype)方法，使得它可以被子方面重写，以执行通用方面不知道的特定复制操作。

例17-4说明了如何把PrototypePattern抽象方面应用于特定的应用程序。

例17-4：把PrototypePattern抽象方面应用于目标应用程序

```
public aspect GraphicPrototypes extends PrototypePattern
{
   declare parents : Graphic implements Prototype;

   declare parents : MusicalNote implements Prototype;

   declare parents : Staff implements Prototype;

   protected Object createCloneFor(Prototype object)
   {
      if (object instanceof MusicalNote)
      {
         return new MusicalNote(
            ((MusicalNote) object).getX(),
            ((MusicalNote) object).getY());
      }
      else if (object instanceof Staff)
      {
         return new Staff(((Staff) object).getX(), ((Staff) object).getY());
      }
      else
      {
         return null;
      }
   }
}
```

图17-5显示了GraphicsPrototype方面对应用程序的类所产生影响的示例。

图17-6显示了新的原型模式行为在示例应用程序中如何交互作用。

参考

16.2节中更详细地解释了使用declare关键字来影响类的静态层次结构；23.3节中解释了Director面向方面的设计模式。

17.3 实现抽象工厂模式

问题

你想使用AspectJ应用抽象工厂模式（abstract factory pattern）。

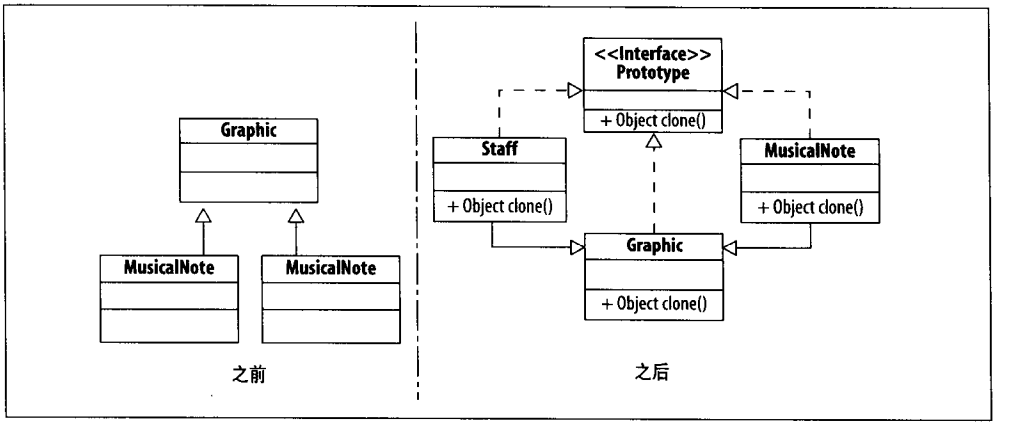

图 17-5：GraphicsPrototype 方面如何影响静态应用程序的结构

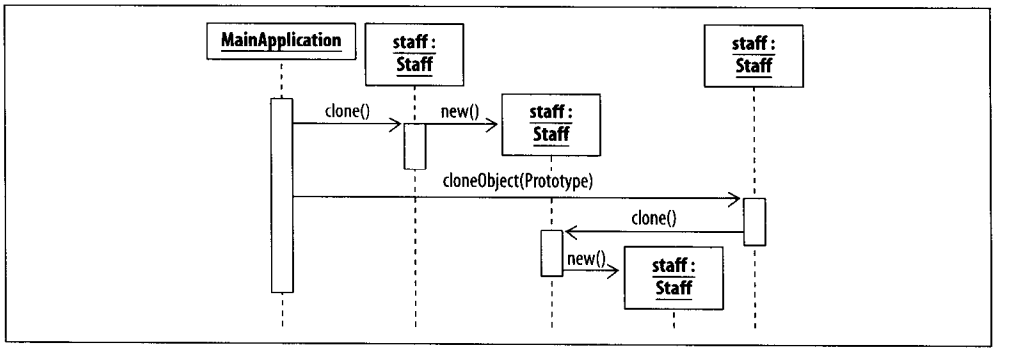

图 17-6：使用 Staff 类的新原型行为

解决方案

抽象工厂模式支持对一组相关类进行实例化，同时把工厂模式的客户与准确的实现隔离开。

为了使用 AspectJ 实现抽象工厂模式，可创建一个方面，它可以使用静态横切技术消除抽象工厂模式对抽象基类的依赖，从而提供工厂方法的默认实现。

讨论

使用抽象方法创建一个通用工厂没有任何意义，因为工厂包含特定于可创建对象的方法。AspectJ 实现提供给这种设计模式的唯一优点是：能够消除抽象工厂对抽象基类的依赖，

并用简单的接口代替它。这意味着专用工厂可以继承自其他更合适的类，而不是使用它们的一个允许的继承关系来支持该设计模式。

例 17-5 说明了如何通过提供 Java 接口中声明的方法的默认实现，将抽象工厂模式应用于需要工厂的简单应用程序。

例 17-5：使用方面应用抽象工厂模式

```
public interface ComputerFactory
{
   public Computer createPentiumProcessorComputer();

   public Computer createComputerWithHardDisk(HardDisk hardDisk);
}

public aspect DefaultComputerFactoryImplementation
{

   public Computer ComputerFactory.createPentiumProcessorComputer()
   {
      Processor processor = new Processor("Pentium 4 : 9089085043");
      Motherboard motherboard = new Motherboard("019283", processor);
      HardDisk hardDisk = new HardDisk("738947");
      FloppyDisk floppyDisk = new FloppyDisk("93746");
      Computer computer = new Computer("12345", motherboard, hardDisk, floppyDisk);
      return computer;
   }

   public Computer ComputerFactory.createComputerWithHardDisk(HardDisk hardDisk)
   {
      Processor processor = new Processor("Pentium Standard : 123478");
      Motherboard motherboard = new Motherboard("434244", processor);
      FloppyDisk floppyDisk = new FloppyDisk("432434");
      Computer computer = new Computer("56789", motherboard, hardDisk, floppyDisk);
      return computer;
   }
}
```

参考

第16章包含一些秘诀，它们说明了如何使用静态横切技术来扩展类，并提供接口元素的默认实现。

17.4 实现工厂方法模式

问题

你想使用 AspectJ 应用工厂方法模式（factory method pattern）。

解决方案

工厂方法模式类似于抽象工厂模式，这是由于它提供了一些机制，通过这些机制，可以把对象的准确实现与工厂的客户分离开。不过，工厂方法提供了单一方法，用于实例化单一接口的不同实现。

包含抽象工厂方法的抽象类是通过专用类实现的，这些专用类显式重写了工厂方法，以提供一些机制，用于实例化所期待对象的不同实现。

为了使用 AspectJ 实现工厂方法模式，可使用与抽象工厂模式相同的机制来创建一个方面，它可以使用静态横切技术删除工厂方法模式对抽象基类的依赖，从而提供了工厂方法的默认实现。

讨论

例 17-6 显示了一个专用方面，它提供了工厂方法的默认实现。

例 17-6：使用方面应用工厂方法模式

```
public interface ComputerCreator
{
   public Computer createComputer(String serial);
}

public aspect DefaultComputerCreatorImplementation
{
   public void ComputerCreator.createComputerAndPrintInventory(String serial)
   {
      System.out.println("Inventory of computerparts:");
      System.out.println(this.createComputer(serial).toString());
   }
}
```

传统上，这个解决方案中的 ComputerCreator 接口将是一个抽象类。不过，在应用工厂方法设计模式时，静态横切技术可以通过消除对抽象基类的需要，而提供更多的自由。

参考

第16章包含一些秘诀，它们说明了静态横切技术，用于扩展类和提供接口元素的默认实现；20.3 节说明了如何分离关于仅仅用方面实例化接口的哪些实现类的决定。

17.5 实现生成器模式

问题

你想使用 AspectJ 应用生成器模式（builder pattern）。

解决方案

生成器模式用于捕获在创建特定对象时可能需要的复杂步骤。这些步骤被实现为生成器类上的方法；在完成每个必须的步骤之后，就可以调用生成器来创建所得到的生成对象。

为了使用 AspectJ 实现生成器模式，可以创建一个方面，它给顶级生成器类添加一个字段和一个方法，前者用于存储生成结果，后者则使用静态横切技术来访问那个结果。这使生成器能够作为一个接口，而不是一个抽象类。

讨论

可以使用方面来实现生成器模式，如例 17-7 所示。

例 17-7：使用方面应用生成器模式

```
public interface TextPhraseBuilder
{
   public void buildHeader(String title);

   public void buildBody(String content);

   public void buildFooter(String closingContent);

   public String getResult();
}

public aspect TextPhraseBuilderDefaultImplementation
{
   public StringBuffer TextPhraseBuilder.result = new StringBuffer();

   public String TextPhraseBuilder.getResult()
   {
      return result.toString();
   }

   /**
    * Declares a compiler error that gets reported if other classes
    * (except Builders or this aspect) try to access the result variable.
    */
   declare error : (
      set(public StringBuffer TextPhraseBuilder +.result)
         || get(public StringBuffer TextPhraseBuilder +.result))
```

```
        && !(within(TextPhraseBuilder +)
        || within(TextPhraseBuilderDefaultImplementation)) :
        "variable result is aspect protected. use getResult() to access it";
}
```

TextPhraseDefaultImplementationBuilder方面提供了getResult()方法的默认实现。这利用一个抽象基类使专用生成器不会用尽它们的单一继承关系。getResult()方法提供了对result字段的访问权限，这个字段也会添加到接口及其实现类中，从而提供一个位置用于存储生成器的结果。

理想情况下，result字段将被声明为受保护的，因为它只由TextPhraseBuilder及其子类在内部使用。AspectJ编译器不允许在接口中引入protected字段，public是唯一的选择。

这导致第二个问题。如何约束模式的使用，从而向开发人员指明：直接访问公共result字段来获得字段值是一种错误的机制？其解决方案是由AspectJ的另一个强大特性提供的，即其定义新的编译时检查和错误通知的能力。

直到出现面向方面时，编译器才成为一种相当刚性的软件，而不会被开发人员篡改。不过，在与可能影响其代码验证的编译器交互时，会发生少数几种情况，它们可能是有用的，AspectJ提供了这种特性。

TextPhraseBuilderDefaultImplementation方面定义了一个由编译器触发的错误，以防代码试图直接访问最近添加的结果属性。这将对模式正在遵守的规则提供最终的检查。尽管要预防的问题是设计模式的面向方面实现的一个缺点，但是通过纳入编译时检查可以减少这个问题。

参考

第16章提供了特定的秘诀，它们说明了用于扩展类、提供默认实现以及扩展编译器能力的静态横切技术。

第18章

实现结构型面向对象设计模式

18.0 简介

沿着前一章中的创建型设计模式继续前进，你将使用AspectJ来利用实现结构型设计模式中的优点。

尽管应用方面的好处因特定的模式而异，与创建型面向对象（OO）设计模型相比，结构型面向对象设计模型可以从方面得到更多的好处。这主要是由于结构型设计模式的横切和通用性，这使其自身极佳地适合于面向方面的实现。

不幸的是，在设计或实现外观（façade）模式时，面向方面并没有提供任何真正的好处。外观模式的目标是：给复杂组件的对象的更大集合提供一个更简单的接口。可以在一个系统中实现这个目标，在这个系统中，已知源代码使用标准类和方法调用，并且不需要AspectJ的额外机制。

18.1 实现复合模式

问题

你想使用AspectJ应用复合模式（composite pattern）。

解决方案

复合模式能够把集合中的对象组合在一起，并且以某种方式与整个组交互，这种交互方式类似于同组中的单个成员进行交互。

例 18-1 使用 Director（导向者）面向方面设计模式（参见第 23 章），利用 AspectJ 提供复合模式的通用实现。

例 18-1：使用方面定义复合模式

```
public abstract aspect CompositePattern
{
   public interface Component
   {
   }

   protected interface Composite extends Component
   {
   }

   protected interface Leaf extends Component
   {
   }

     private WeakHashMap perComponentChildren = new WeakHashMap();

   private Vector getChildren(Component s)
   {
      Vector children = (Vector) perComponentChildren.get(s);
      if (children == null)
      {
         children = new Vector();
         perComponentChildren.put(s, children);
      }
      return children;
   }

   public void addChild(Composite composite, Component component)
   {
      getChildren(composite).add(component);
   }

   public void removeChild(Composite composite, Component component)
   {
      getChildren(composite).remove(component);
   }

    public Enumeration getAllChildren(Component c) {
        return getChildren(c).elements();
    }

    public interface Visitor {

        public void doOperation(Component c);
    }

    public void recurseOperation(Component c, Visitor v) {
        for (Enumeration enum = getAllChildren(c); enum.hasMoreElements();) {
            Component child = (Component) enum.nextElement();
```

```
            v.doOperation(child);
        }
    }

  public interface FunctionVisitor
  {
     public Object doFunction(Component c);
  }

  public Enumeration recurseFunction(Component c, FunctionVisitor fv)
  {
     Vector results = new Vector();
     for (Enumeration enum = getAllChildren(c); enum.hasMoreElements();) {
     Component child = (Component) enum.nextElement();
        results.add(fv.doFunction(child));
     }
     return results.elements();
  }
}
```

讨论

CompositePattern方面定义了Composite和Leaf接口，以应用于应用程序内扮演这些角色的类。这个方面使用访问者模式（visitor pattern）进行递归访问，并处理复合模式中的每个组件。

图 18-1 显示了 CompositePattern抽象方面的结构，以及它定义用于支持复合设计模式的接口和行为。

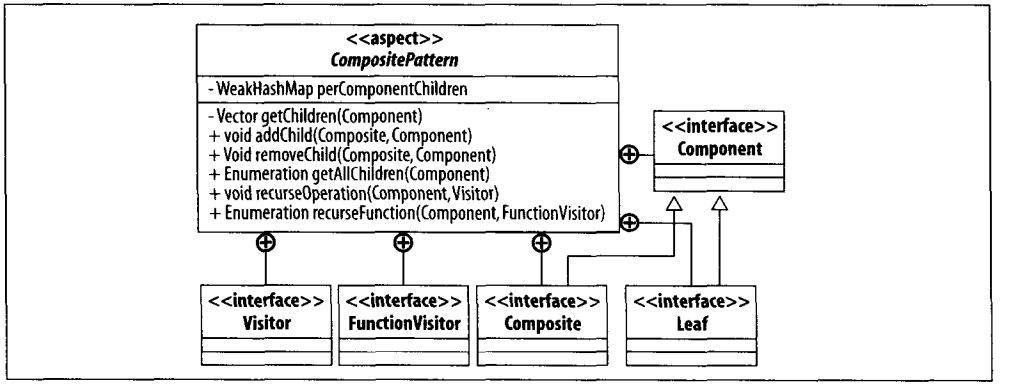

图 18-1：CompositePattern 方面以及它为设计模式的角色定义的接口

这个 CompositePattern 抽象方面被扩展成一些专用的子方面，它们指定了扮演Composite和Leaf角色的类。例18-2说明了如何将CompositePattern抽象方面应用于特定的应用程序。

例 18-2：将 CompositePattern 方面应用于目标应用程序

```
public aspect GraphicsComposite extends CompositePattern
{
   declare parents : Window implements Composite;
   declare parents : Line implements Leaf;
   declare parents : Rectangle implements Leaf;

   public void Component.draw(PrintStream s)
   {
      s.println("Drawing: " + this);
   }

   public void Composite.draw(final PrintStream s)
   {
      s.println("Composite: " + this);
      GraphicsComposite.aspectOf().recurseOperation(this, new Visitor()
         {
            public void doOperation(Component c)
            {
               c.draw(s);
            }
         });
   }

   public void Leaf.draw(PrintStream s)
   {
      s.println("Drawing Leaf: " + this);
   }
}
```

图 18-2 显示了在应用 GraphicsComposite 方面之前的一组示例应用程序。

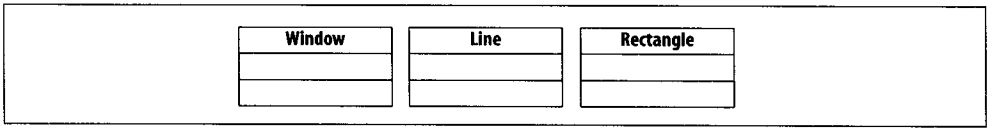

图 18-2：Window、Line 和 Rectangle 业务逻辑类

图18-3显示了将例18-2中所示的GraphicsComposite方面应用于一组应用程序类的效果。

图 18-4 显示了如何在示例应用程序中与 Window 类新的复合行为交互。

参考

第16章提供了关于使用方面扩展现有类的更多信息；23.3节解释了Director面向方面设计模式。

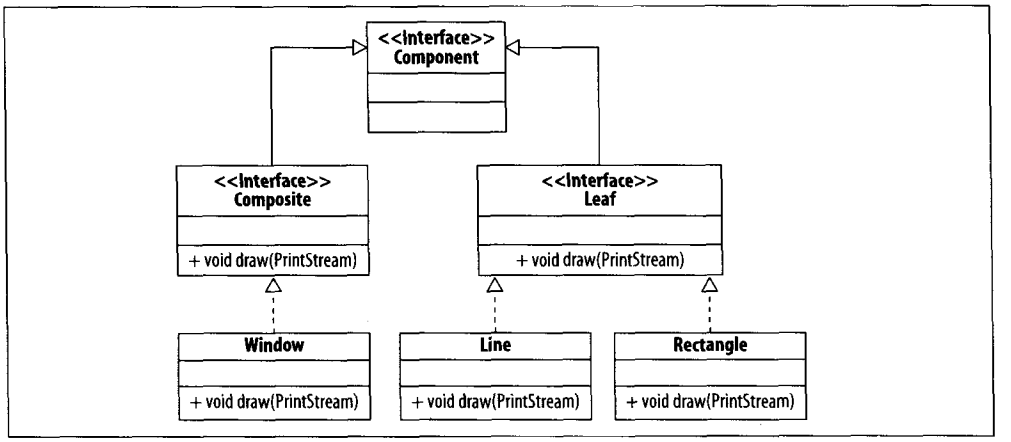

图 18-3：在将复合模式应用于 Window、Line 和 Rectangle 类之后的静态结构

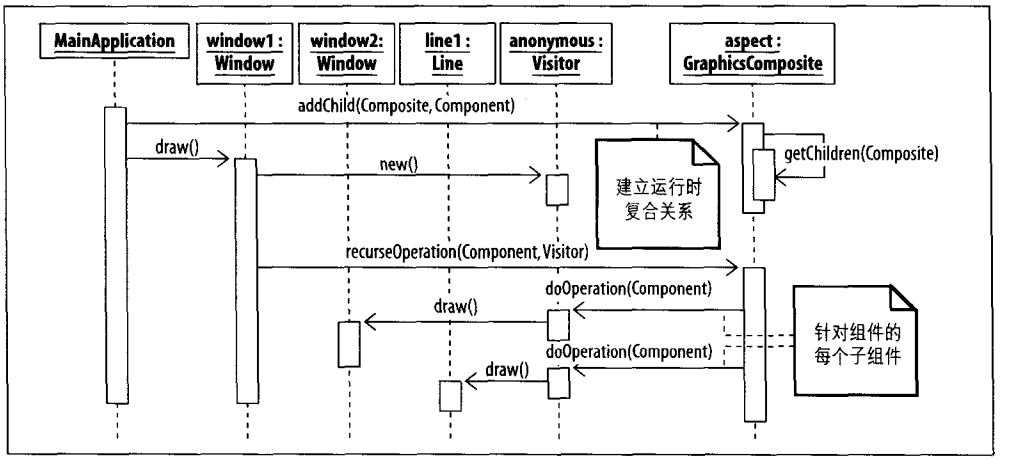

图 18-4：使用 Window 类新的复合行为

18.2 实现享元模式

问题

你想使用 AspectJ 应用享元模式（flyweight pattern）。

解决方案

享元模式提供了一些机制，利用这些机制通过在合适的地方共享对象，可以把细粒度的

对象纳入到OO设计中，而不会带来资源和性能损失。重型对象（heavyweight object）潜在地通过许多享元对象来封装要引用的实际数据。

例18-3中的抽象方面使用Director面向方面设计模式（参见第23章）提供一个模板，应用程序特定的方面可以通过它来应用享元模式。

例 18-3：使用抽象方面定义享元模式

```
public abstract aspect FlyweightPattern
{
   private Hashtable flyweightPool = new Hashtable();

   public interface Flyweight
   {
   };

   protected abstract pointcut flyweightCreation(Object key);

   Object around(Object key) : flyweightCreation(key) &&
      !within(com.oreilly.aspectjcookbook.oopatterns.FlyweightPattern+)
   {
      return this.checkFlyweight(key);
   }

   public synchronized Flyweight checkFlyweight(Object key)
   {
      if (flyweightPool.containsKey(key))
      {
         return (Flyweight) flyweightPool.get(key);
      }
      else
      {
         Flyweight flyweight = createNewFlyweight(key);
         flyweightPool.put(key, flyweight);
         return flyweight;
      }
   }

   protected abstract Flyweight createNewFlyweight(Object key);
}
```

讨论

图 18-5 显示了 FlyweightPattern 抽象方面的结构，及其定义的用于支持享元设计模式的接口和行为。

例 18-3 中的 FlyweightPattern 抽象方面包含一个享元池集合。这个集合（实现为一个散列表）会记住已经创建的重型对象，因此，在可用时，可以把享元设置成现有的重型对象。

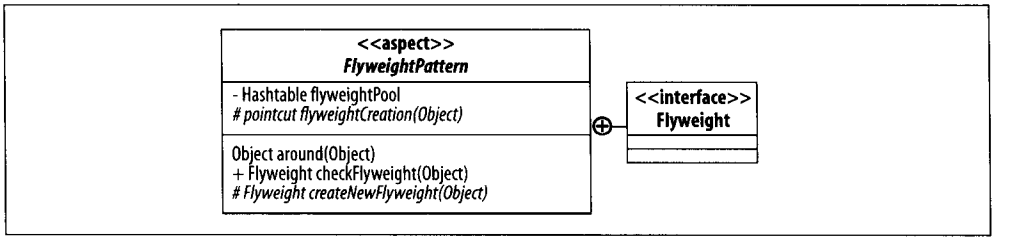

图 18-5：FlyweightPattern 方面及其为设计模式的角色定义的接口

由于抽象方面不知道如何创建不同的具体享元对象，所以它定义了一个抽象方法 createNewFlyweight(...)。这个抽象方法允许子方面实现它们将如何创建特定的享元对象。

FlyweightPattern方面还包含flyweightCreation(Object)抽象切入点，它用于捕获何时创建被指定为享元的对象。这个切入点使用关联的around()通知，允许FlyweightPattern重写享元对象的创建，使用checkFlyweight(..)方法通过检查是否可以利用现有的重型对象来应用设计模式的规则，从而根据需要创建新的重型对象。

例 18-4 中的方面说明了如何把 FlyweightPattern 抽象方面应用于特定的应用程序。

例 18-4：将 FlyweightPattern 方面应用于应用程序的类

```
public aspect PrintableCharacterFlyweight extends FlyweightPattern
{
   declare parents : PrintableCharacter implements Flyweight;

   protected pointcut flyweightCreation(Object key) :
      call(public com.oreilly.aspectjcookbook.PrintableCharacter.new(Character))
      && args(key);

   protected Flyweight createNewFlyweight(Object key)
   {
      return new PrintableCharacter((Character) key);
   }
}
```

PrintableCharacterFlyweight 指定将 PrintableCharacter 类作为享元管理。flyweightCreation(..)切入点被实现成捕获何时创建 PrintableCharacter对象。createNewFlyweight(Object)方法被实现成根据需要创建新的PrintableCharacter对象。

图 18-6 显示了在把 PrintableCharacterFlyweight 方面应用于应用程序之前和之后的效果。

图 18-7 显示了 PrintableCharacter 类的新享元行为在示例应用程序中的作用方式。

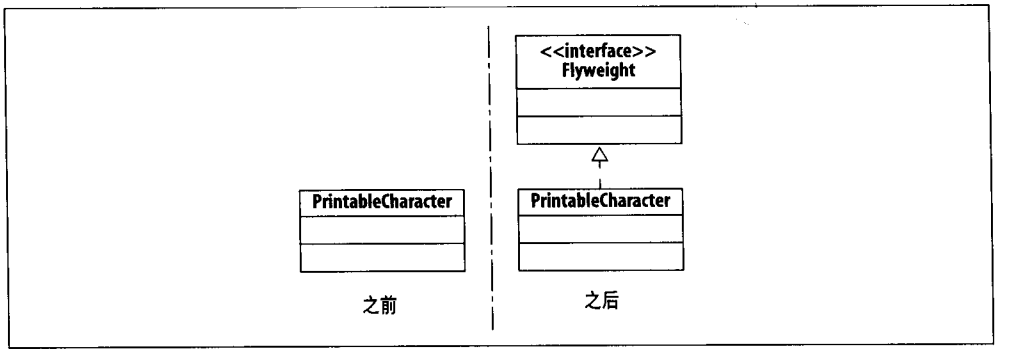

图 18-6：将享元模式应用于 PrintableCharacter 类的效果

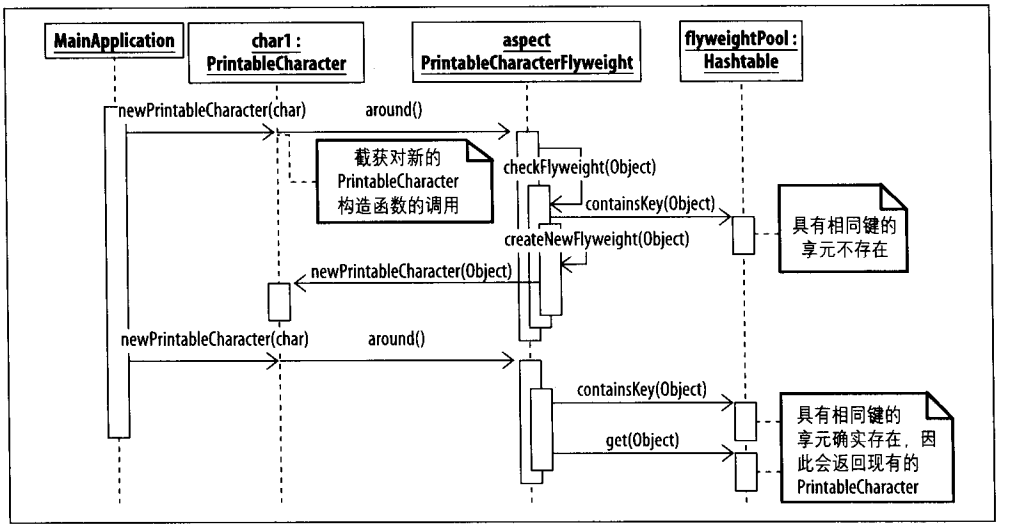

图 18-7：使用 PrintableCharacter 类的新享元行为

参考

第 15 章中的秘诀介绍了关于定义抽象方面和特殊化它们的更多信息；23.3 节解释了 Director 面向方面设计模式。

18.3 实现适配器模式

问题

你想使用 AspectJ 应用适配器模式（adapter pattern）。

解决方案

适配器模式提供了一种方式，把从一个类发送的消息更改成真正的目标类所期待的消息，使消息适合于把两个粘连在一起。

例 18-5 说明了如何定义应用程序特定的适配器模式方面。

例 18-5：使用方面适配现有的类

```
public aspect PrinterScreenAdapter
{
   declare parents : Screen implements Printer;

   public void Screen.print(String s)
   {
      outputToScreen(s);
   }
}
```

讨论

特定的PrinterScreenAdapter方面通过使用AspectJ的能力来扩展带有新方法的现有类以及潜在的新父类，来扩展类的行为，使之适合于它所需要的新能力，在任何时候都要小心不要妨碍这个类原来的能力。

图 18-8 显示了在把PrinterScreenAdapter方面应用于Screen类（它被修改成支持打印行为）之前和之后的效果。

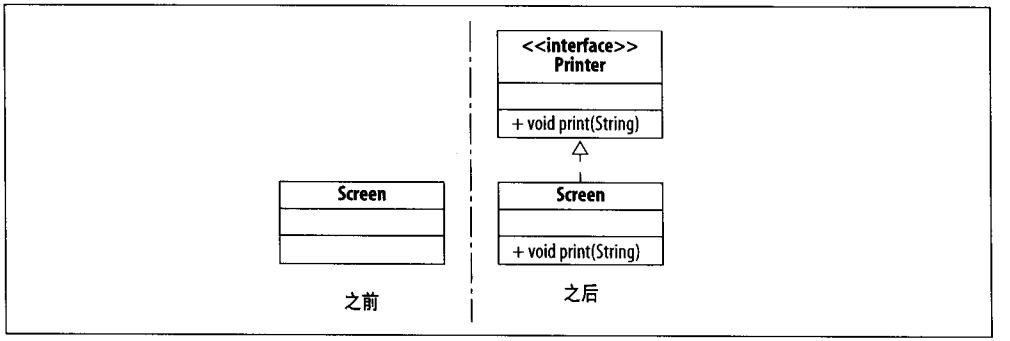

图 18-8：将适配器模式应用于 Screen 类的效果

参考

第 16 章中的秘诀提供了关于在定义方面时，如何扩展现有的类以及纳入接口的更多信息。

18.4 实现桥接模式

问题

你想使用 AspectJ 应用桥接模式（bridge pattern）。

解决方案

桥接模式把类与一种特定实现的底层特征分离开，使得可以应用不同的实现，而不会影响类的客户。

例 18-6 说明了如何定义应用程序特定的桥接模式方面。

例 18-6：使用特定的窗口系统来桥接实现无关的窗口与实现它的方式

```
public class Window
{
   public void drawText(String text)
   {

   }

   public void drawRect()
   {


   }
}

public aspect XWindowBridge perthis(captureAllBridgedCalls())
{
   private XWindow imp = new XWindow();

   public pointcut captureDrawText(String text) :
      execution(public void Window.drawText(String))
      && args(text);

   public pointcut captureDrawRect() : execution(public void Window.drawRect());

   public pointcut captureAllBridgedCalls() :
      captureDrawText(String)
      || captureDrawRect();

   void around(String text) : captureDrawText(text)
   {
      imp.drawText(text);
   }

   void around() : captureDrawRect()
   {
      imp.drawLine();
      imp.drawLine();
```

```
        imp.drawLine();
        imp.drawLine();
    }
}
```

讨论

图18-9显示了XWindowBridge方面的结构，以及它定义用于支持桥接设计模式的行为。

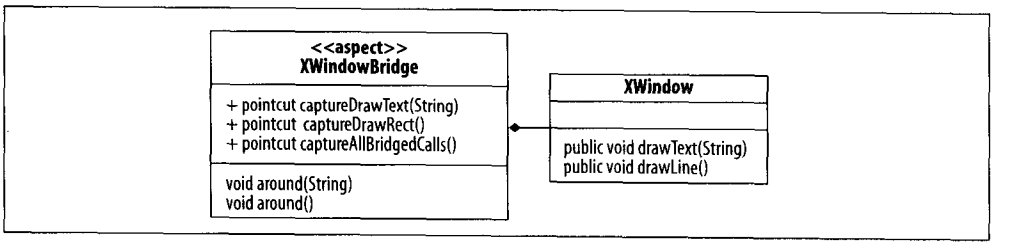

图 18-9：XWindowBridge 方面的结构

XWindowBridge方面捕获需要导向XWindow类上特定调用的Window类上的所有方法。例如，Window类上的drawRect()方法会在XWindow类上产生4个调用，但是通过使用桥接模式，Window类不必将这些调用硬编码进它的行为中。在例18-6中，Window类并不确切知道将如何在不同的系统上实现它，但是XWindowBridge方面会照料好这一切。

通过使用perthis(..)方面实例化策略，将会为通过captureAllBridgedCalls()切入点指定的每个新的桥接Window对象创建一个新方面。这意味着每个Window对象都有它自己的XWindow实现对象的副本。如果想在多个Window对象之间共享XWindow对象，那么所需做的全部工作是删除perthis(..)策略，并删除冗余的captureAllBridgedCalls()切入点定义。

如果使用另一种方式来绘制窗口，那么就可以用不同的方式创建用于桥接Window类上调用的单独方面，并将其织入到应用程序中，而不是XWindowBridge方面中。

图18-10说明了Window类新的桥接行为在示例应用程序中的作用方式。

参考

4.4节解释了execution(Signature)切入点；11.3节解释了args([Types | Identifiers])；13.4节说明了around()形式的通知；14.2节解释了perthis(Pointcut)方面实例化策略；2.6节介绍了在编译时指定构建到应用程序中的方面。

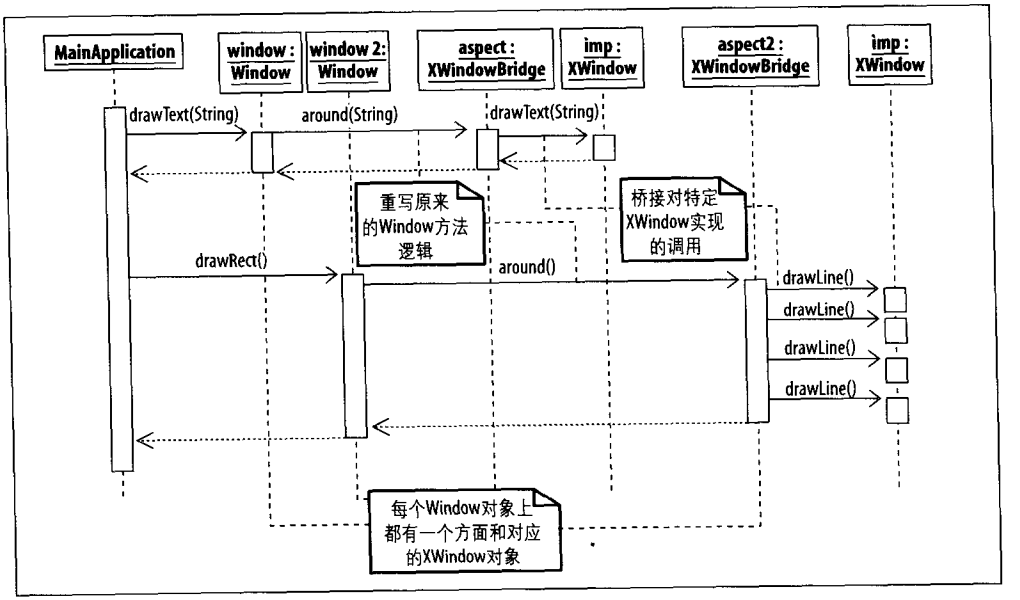

图 18-10：使用 Window 类的桥接行为

18.5 实现装饰模式

问题

你想使用 AspectJ 应用装饰模式（decorator pattern）。

解决方案

装饰模式扩展了类方法的行为，同时维持其现有的公共接口，而无需类知道或关心扩展。

例18-7说明了一个抽象方面，它为装饰模式打好基础，之后可以使用目标应用程序中更特定的子方面来应用装饰模式。

例 18-7：使用方面定义装饰模式

```
public abstract aspect DecoratorPattern
{
   public interface DecoratedComponent
   {
   };

   private boolean DecoratedComponent.decorated = false;
```

```
   public void DecoratedComponent.setDecorated(boolean decorated)
   {
      this.decorated = decorated;
   }

   public void DecoratedComponent.isDecorated(boolean decorated)
   {
      return this.decorated ;
   }
}
```

讨论

图18-11显示了DecoratorPattern抽象方面的结构，及其定义的用于支持装饰设计模式的接口和行为。

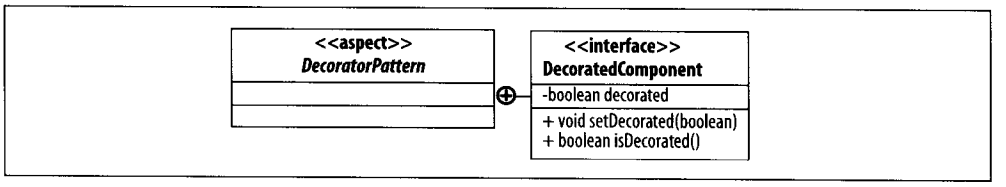

图18-11：DecoratorPattern方面及其为设计模式的角色定义的接口和行为

DecoratorPattern抽象方面定义了DecoratedComponent接口，用于为将要被装饰实现的所有类提供一个公共基接口。在这些类中引入了装饰字段，以及一对访问器和修改器方法。

例18-8说明了如何将DecoratorPattern抽象方面应用于特定的应用程序。TextDisplayDecorator指定可以装饰TextDisplay类，并且如果装饰特定的对象，则会在调用display(..)方法之前和之后装饰它。

例18-8：将DecoratorPattern应用于应用程序的类

```
public aspect TextDisplayDecorator extends DecoratorPattern
{
   declare parents : TextDisplay implements DecoratedComponent;

   public pointcut selectDecorators(Object object) :
      call(public void TextDisplay.display(String))
      && target(object);

   before(Object object) : selectDecorators(object) &&
      if(((DecoratedComponent)object).getDecorated)
   {
      System.out.print("<Decoration>");
   }
```

```
    after(Object object) : selectDecorators(object) &&
        if(((DecoratedComponent)object).getDecorated)
    {
        System.out.print("</Decoration>");
    }
}
```

图18-12显示了在把例18-8中所示的TextDisplayDecorator方面应用于TextDisplay类时的效果。

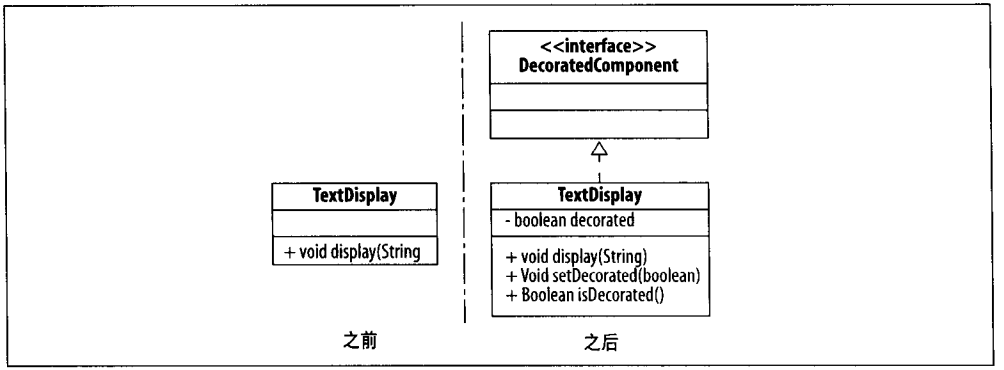

图 18-12：将装饰模式应用于 TextDisplay 类的效果

图 18-13 显示了 TextDisplay 类新的装饰行为在示例应用程序中的作用方式。

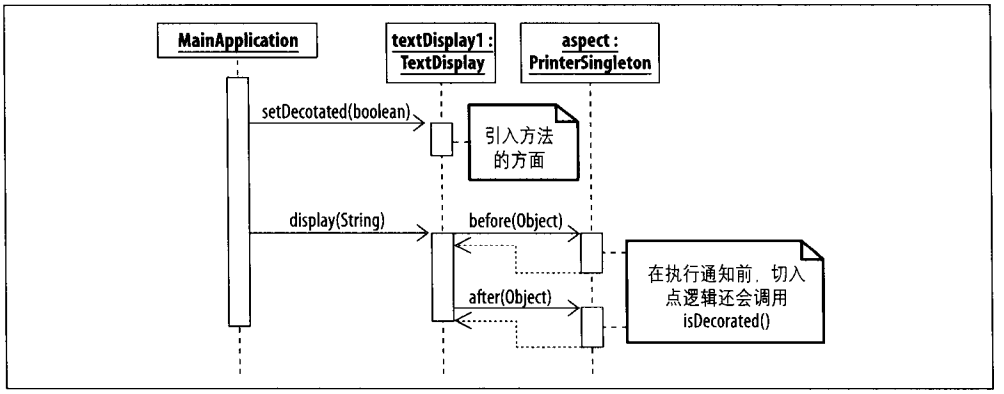

图 18-13：使用 TextDisplay 类的装饰行为

参考

第 16 章中的秘诀包含关于一些机制的更多详细信息，利用这些机制，可以使用方面和

declare关键字来扩展现有的类；4.1节介绍了call(Signature)切入点；13.2节研究了展示连接点环境；11.3节描述了args(Type | Identifier)切入点。

18.6 实现代理模式

问题

你想使用AspectJ应用代理模式（proxy pattern）。

解决方案

代理模式允许开发人员提供代理对象来代替实际的对象，以防需要委托或控制对真实对象的访问。

例18-9使用Director面向方面设计模式（参见第23章）来提供代理模式的抽象实现。

例18-9：使用抽象方面定义代理模式

```
public abstract aspect ProxyPattern
{
   protected interface Subject
   {
   }

   protected abstract pointcut requestTriggered();

   private pointcut accessByCaller(Object caller) : requestTriggered() && this(caller);

   private pointcut accessByUnknown() : requestTriggered() && !accessByCaller(Object);

   Object around(Object caller, Subject subject) : accessByCaller(caller)
      && target(subject)
   {
      if (reject(caller, subject, thisJoinPoint))
      {
         return rejectRequest(caller, subject, thisJoinPoint);
      }
      else if (delegate(caller, subject, thisJoinPoint))
      {
         return delegateRequest(caller, subject, thisJoinPoint);
      }
      return proceed(caller, subject);
   }

   Object around(Subject subject) : accessByUnknown()
      && target(subject)
   {
      // Without a caller then reject does not really make sense
      // as there is no way of deciding to reject or not
```

```
        if (delegate(null, subject, thisJoinPoint))
        {
            return delegateRequest(null, subject, thisJoinPoint);
        }
        return proceed(subject);
    }

    protected abstract boolean reject(
        Object caller,
        Subject subject,
        JoinPoint joinPoint);

    protected abstract boolean delegate(
        Object caller,
        Subject subject,
        JoinPoint joinPoint);

    protected abstract Object rejectRequest(
        Object caller,
        Subject subject,
        JoinPoint joinPoint);

    protected abstract Object delegateRequest(
        Object caller,
        Subject subject,
        JoinPoint joinPoint);
}
```

讨论

图 18-14 显示了 ProxyPattern 抽象方面的结构，以及它定义的用于支持代理设计模式的接口和行为。

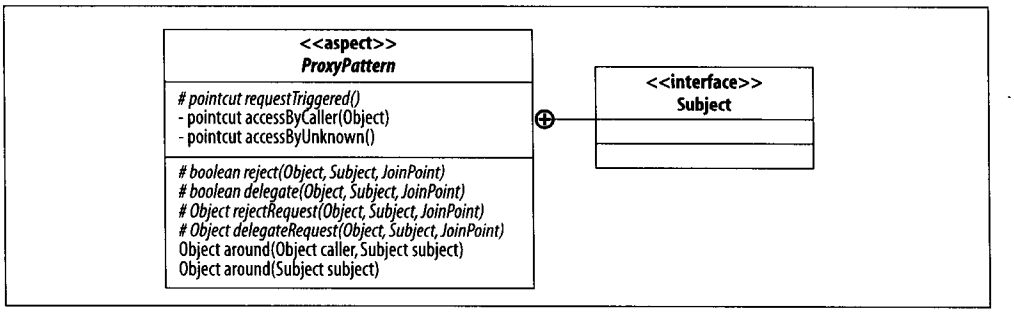

图 18-14：ProxyPattern 方面及其为设计模式的角色定义的接口和行为

例 18-9 中的代理模式的抽象方面定义封装了 Subject（主题）的角色，它应用于需要定义代理逻辑的对象。对于可能应用代理的两种情况（委托和保护）中的任何一种情况，

都存在一条定义的路径，通过它来检查和委托或者拒绝对主题的调用，这取决于继承子方面中包含的逻辑。

代理模式的这个方面实现的最重要的优点是：目标应用程序的原始类不必知道将在代理情况下涉及它们。这是绝对关键的，因为这可以保证应用程序中的对象集合的安全，并且其他代理敏感的考虑事项不会影响原始类的设计目标。

例 18-10 说明了如何将 ProxyPattern 抽象方面应用于特定的应用程序。DelegationProxy 方面定义了一个代理，它会截取并委托对主题对象的调用。

例 18-10：使用 ProxyPattern 方面应用代理

```
public aspect DelegationProxy extends ProxyPattern
{
   declare parents : RealSubject implements Subject;

   protected pointcut requestTriggered() : call(* RealSubject.write(..));

   protected boolean reject(
      Object caller,
      Subject subject,
      JoinPoint joinPoint)
   {
      return false;
   }

   protected boolean delegate(
      Object caller,
      Subject subject,
      JoinPoint joinPoint)
   {
      return true;
   }

   protected Object rejectRequest(
      Object caller,
      Subject subject,
      JoinPoint joinPoint)
   {
      return null;
   }

   protected Object delegateRequest(
      Object caller,
      Subject subject,
      JoinPoint joinPoint)
   {
      Object[] args = joinPoint.getArgs();
      if (args != null)
      {
         AnotherRealSubject.write((String) args[0]);
```

```
        }
        else
        {
            AnotherRealSubject.write("");
        }
        return null;
    }
}
```

图 18-15 显示了将 DelegationProxy 方面应用于 RealSubject 类时的效果。

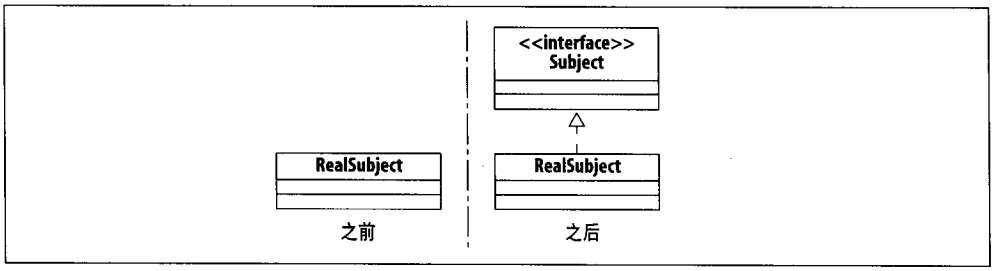

图 18-15：将代理模式应用于 RealSubject 类的效果

图 18-16 显示了与面向方面的代理模式特性交互的示例。

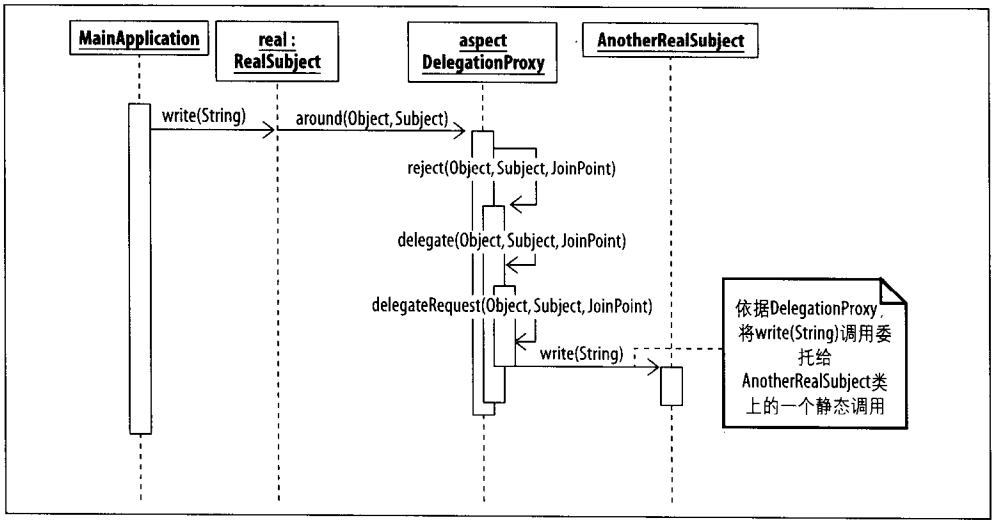

图 18-16：委托对 RealSubject 类上的 write(String)方法的调用

参考

有关定义抽象方面和特殊化它们的更多信息，参见第16章中的秘诀；4.1节介绍了call(Signature)切入点；13.2节研究了展示连接点环境；11.3节描述了args(Type | Identifier)切入点；11.1节更详细地描述了this(Type | Identifier)切入点；11.2节研究了target([Type | Identifier])切入点；13.4节提供了关于proceed()调用及其在around()通知内的应用的更多详细信息；23.3节解释了Director面向方面设计模式。

第 19 章

实现行为型面向对象设计模式

19.0 简介

在介绍三种设计模式的章节中，本章将介绍最后一种设计模式，它研究了行为型模式如何从面向方面的实现中获益。

行为型模式是在所有参与类中构造交互和定位行为的通用指导。本章中的秘诀说明了如何能够通过使用面向方面来获得相当多的体系结构优点。

与上一章中介绍的结构型模式一样，有些GoF模式看起来简直不能从方面显著地受益。就行为型方面来说，这意味着本章中没有介绍解释器（Interpreter）设计模式，尽管Jan Hanneman完成的研究（在*http://www.cs.ubc.ca/~jan/AODPs/*上可以找到）可能揭示一些方法，它们进一步细化了本章中介绍的机制，并且也许还会导致一些改进，可以使用面向方面对解释器模式执行这些改进。

19.1 实现观察者模式

问题

你想使用AspectJ应用观察者模式（observer pattern）。

解决方案

观察者模式允许设计者创建对象之间的依赖关系，使得如果一个对象的状态发生变化，则会通知另一个对象，并且它可能会产生相应的行动。

例 19-1 使用 Director（导向者）面向方面设计模式（参见第 23 章），来提供观察者模式的通用方面实现。

例 19-1：使用抽象方面定义观察者模式

```
public abstract aspect ObserverPattern
{
   protected interface Subject
   {
      public void addObserver(Observer observer);
      public void removeObserver(Observer observer);
   }

   protected interface Observer
   {
      public void notifyOfChange(Subject subject);
   }

   private List Subject.observers = new LinkedList();

   public void Subject.addObserver(Observer observer)
   {
      this.observers.add(observer);
   }

   public void Subject.removeObserver(Observer observer)
   {
      this.observers.remove(observer);
   }

   private synchronized void Subject.notifyObservers()
   {
      Iterator iter = this.observers.iterator();
      while (iter.hasNext())
      {
         ((Observer)iter.next()).notifyOfChange(this);
      }
   }

   protected abstract pointcut subjectChange(Subject s);

   after(Subject subject) : subjectChange(subject)
   {
      subject.notifyObservers();
   }
}
```

讨论

ObserverPattern方面把Subject（主题）和Observer（观察者）角色定义为标识符，可以通过继承子方面把它们应用于特定的类。通过定义这些角色，ObserverPattern方面就会实现一种机制，当主题改变时，通过该机制来通知观察者。

图 19-1 显示了 ObserverPattern 抽象方面的结构，以及它定义用于支持观察者设计模式的接口和行为。

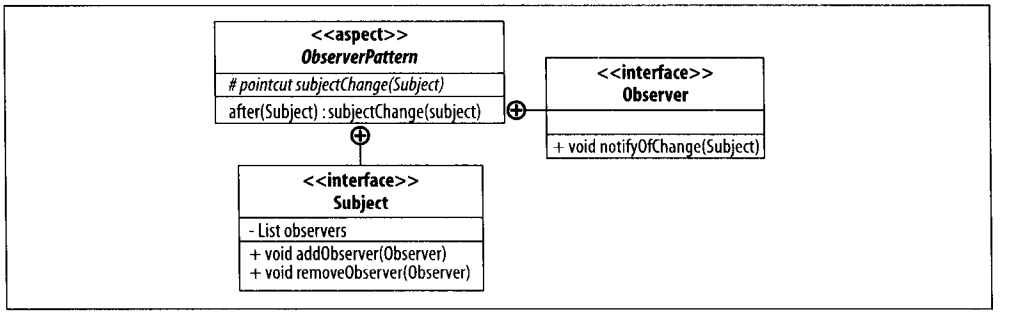

图 19-1：ObserverPattern 方面以及它为设计模式的角色定义的接口

例 19-2 说明了如何将 ObserverPattern 方面应用于特定的应用程序。

例 19-2：将 ObserverPattern 应用于应用程序的类

```
public aspect ConcreteClassAObserver extends ObserverPattern
{
   declare parents : ConcreteClassB implements Subject;

   declare parents : ConcreteClassA implements Observer;

   protected pointcut subjectChange(Subject s) :
      call(* ConcreteClassB.set*(..))
      && target(s);

   public void ConcreteClassA.notifyOfChange(Subject subject)
   {
      this.doSomething(
         "ConcreteClassA was notified of a change on " + subject);
   }
}
```

ConcreteClassAObserver 把 Observer 接口应用于 ConcreteClassA 类，以及把 Subject 接口应用于 ConcreteClassB 类。这意味着在 ConcreteClassB 对象上调用修改器时，将会通知被注册为观察者的 ConcreteClassA 对象。在 notifyOfChange(Subject) 方法中声明了在发出通知时，将会调用哪些方法的细节，该方法被添加到 ConcreteClassA 类中。

图 19-2 显示了在应用 ConcreteClassAObserver 方面之前的一组示例应用程序类。

图 19-3 显示了在把 ConcreteClassAObserver 方面应用于 ConcreteClassA 类和 ConcreteClassB 类之后的效果。

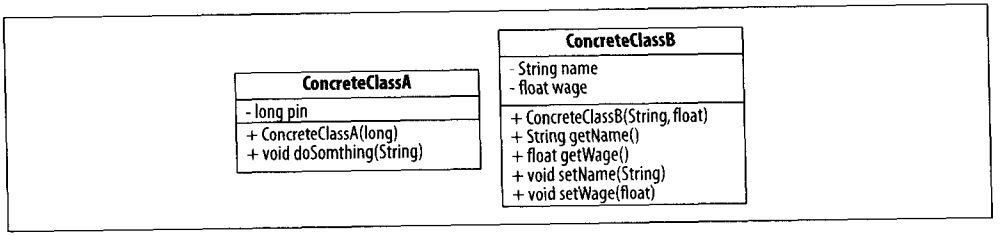

图 19-2：ConcreteClassA 类和 ConcreteClassB 类

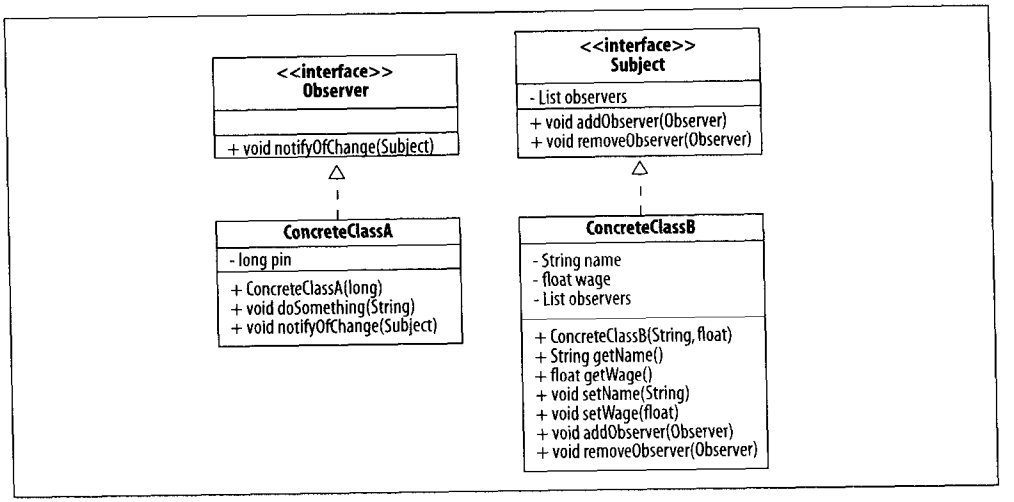

图 19-3：在把观察者模式应用于 ConcreteClassA 类和 ConcreteClassB 类之后的静态结构

参考

第 16 章中的秘诀包含关于一些机制的更多详细信息，利用这些机制，可以使用方面和 `declare` 关键字来扩展现有的类；4.1 节介绍了 `call(Signature)` 切入点；4.4 节描述了 `execution(Signature)` 切入点；13.2 节研究了展示连接点环境；11.2 节研究了 `target([Type | Identifier])` 切入点；23.3 节解释了 Director 面向方面设计模式。

图 19-4 显示了 `ConcreteClassA` 类的新的观察者行为和 `ConcreteClassB` 类的主题行为如何在示例应用程序内交互。

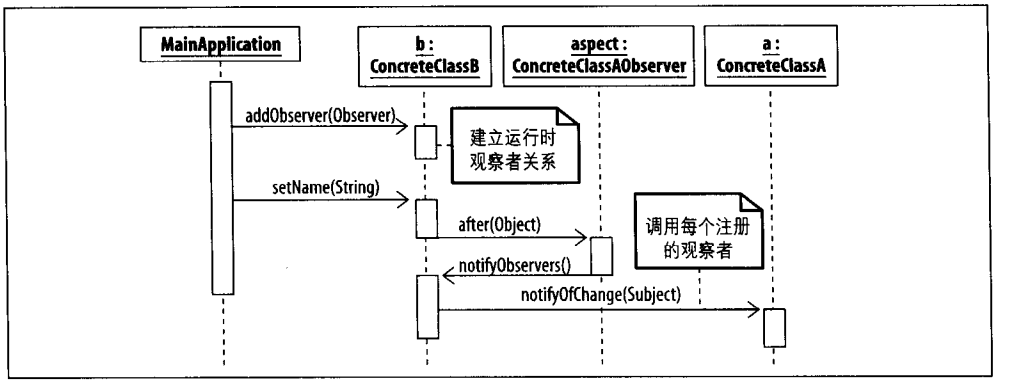

图 19-4：使用 ConcreteClassA 类和 ConcreteClassB 类的观察者和主题行为

19.2 实现命令模式

问题

你想使用 AspectJ 应用命令模式（command pattern）。

解决方案

命令设计模式本身支持把请求封装成对象。可以依赖于其目的，把单个或多个操作组合进一个请求。一旦构造了请求对象，就可以从发起对象把它作为单独的实体进行管理。

例 19-3 使用 Director 面向方面设计模式（参见第 23 章）来提供一个抽象方面，可以使用它来应用命令模式。

例 19-3：使用方面定义命令模式

```
public abstract aspect CommandPattern
{
   public interface Command
   {
      public void executeCommand(CommandReceiver receiver);

      public boolean isExecutable();
   }

   public interface CommandInvoker
   {
   }
```

```
public interface CommandReceiver
{
}

private WeakHashMap mappingInvokerToCommand = new WeakHashMap();

public Object setCommand(CommandInvoker invoker, Command command)
{
   return mappingInvokerToCommand.put(invoker, command);
}

public Object removeCommand(CommandInvoker invoker)
{
   return setCommand(invoker, null);
}

public Command getCommand(CommandInvoker invoker)
{
   return (Command) mappingInvokerToCommand.get(invoker);
}

private WeakHashMap mappingCommandToReceiver = new WeakHashMap();

public Object setReceiver(Command command, CommandReceiver receiver)
{
   return mappingCommandToReceiver.put(command, receiver);
}

public CommandReceiver getReceiver(Command command)
{
   return (CommandReceiver) mappingCommandToReceiver.get(command);
}

protected abstract pointcut commandTrigger(CommandInvoker invoker);

after(CommandInvoker invoker) : commandTrigger(invoker)
{
   Command command = getCommand(invoker);
   if (command != null)
   {
      CommandReceiver receiver = getReceiver(command);
      command.executeCommand(receiver);
   } else
   {
      // Do nothing: This Invoker has no associated command
   }
}

protected pointcut setCommandTrigger(CommandInvoker invoker, Command command);

after(CommandInvoker invoker, Command command) : setCommandTrigger(invoker, command)
{
   if (invoker != null)
      setCommand(invoker, command);
}
```

```
    protected pointcut removeCommandTrigger(CommandInvoker invoker);

    after(CommandInvoker invoker) : removeCommandTrigger(invoker)
    {
        if (invoker != null)
            removeCommand(invoker);
    }

    public boolean Command.isExecutable()
    {
        return true;
    }
}
```

讨论

CommandPattern 抽象方面定义了一些机制，可以通过这些机制来建立 Command-Invoker、Command 和 CommandReceiver 的角色，并且让它们彼此交互。通过特殊化的子方面把这些抽象角色赋予应用程序特定的类。

图19-5说明了CommandPattern抽象方面的结构，以及它定义的用于支持命令设计模式的接口和行为。

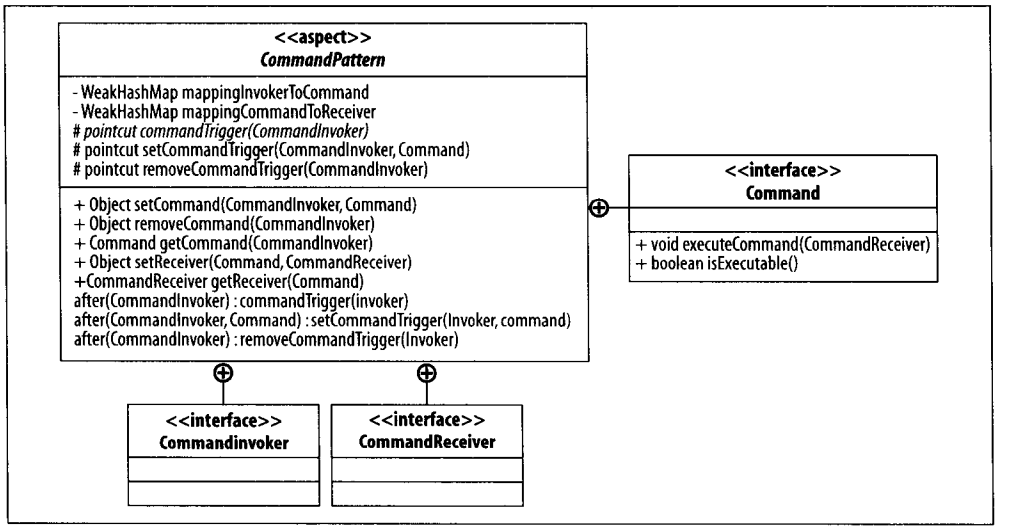

图 19-5：CommandPattern 方面以及它为设计模式的角色定义的接口

例 19-4 说明了如何将 CommandPattern 抽象方面应用于特定的应用程序。

例 19-4：将 CommandPattern 方面应用于应用程序的类

```
public aspect ConcreteCommand extends CommandPattern
{
   declare parents : TimedEvent implements CommandInvoker;
   declare parents : Printer implements CommandReceiver;
   declare parents : VCardPrinter implements CommandReceiver;
   declare parents : BusinessCard implements Command;

   public void BusinessCard.executeCommand(CommandReceiver receiver)
   {
      if (receiver instanceof Printer)
      {
         ((Printer) receiver).println(this.toString());
      } else
      {
         ((VCardPrinter) receiver).printVCard(this);
      }
   }

   public void executeCommand(CommandReceiver receiver)
   {
      ((Printer) receiver).println("Command triggered on printer receiver");
   }

   protected pointcut commandTrigger(CommandInvoker invoker) :
      call(void TimedEvent.timedOut())
      && target(invoker);
}
```

图 19-6 显示了在应用 ConcreteCommand 方面之前的一组示例应用程序类。从 Java 标准库中使用 TimerTask 类，该类支持以定时的时间间隔执行 TimedEvent 类。

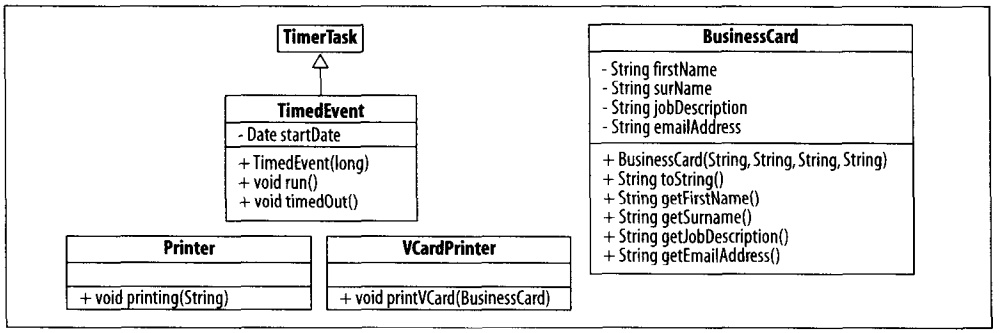

图 19-6：Printer 类、VCardPrinter 类、TimedEvent 类和 BusinessCard 类

图 19-7 显示了将 ConcreteCommand 方面应用于 TimedEvent 类、Printer 类、VCardPrinter 类和 BusinessCard 类的效果。

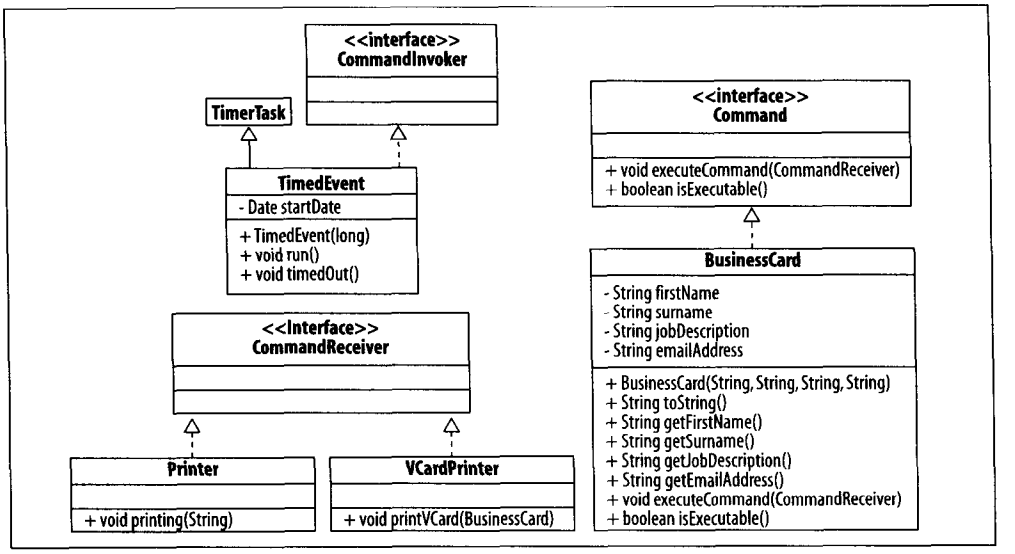

图 19-7：将命令模式应用于 TimedEvent 类、Printer 类、VCardPrinter 类和 BusinessCard 类之后的静态结构

图 19-8 显示了应用程序的类的新命令模式行为在示例应用程序中如何交互。

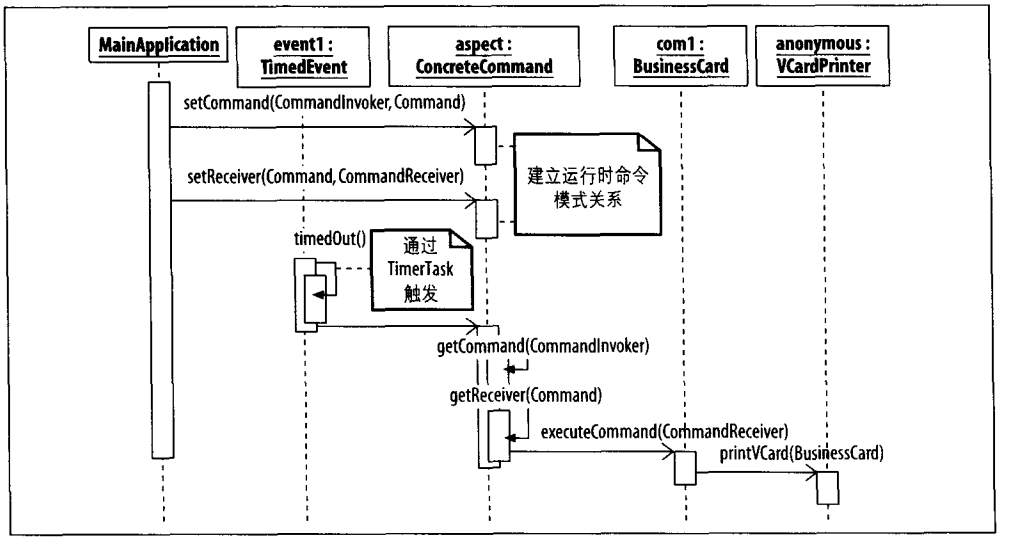

图 19-8：使用 TimedEvent 类、BusinessCard 类和 VCardPrinter 类的命令模式特征

参考

第16章中的秘诀包含关于一些机制的更多详细信息，利用这些机制，可以使用方面和declare关键字来扩展现有的类；15.1节介绍了方面和继承；13.5节研究了after()形式的通知；23.3节解释了Director面向方面设计模式。

19.3 实现迭代器模式

问题

你想使用AspectJ应用迭代器模式（iterator pattern）。

解决方案

迭代器模式提供了一种机制，通过它将对象集合或聚合（在有时调用它时）的实现与顺序访问集合的机制隔离开。迭代器（或游标）移动对象的聚合移动，从而提供每个对象，同时对聚合的用户隐藏次序和访问实现的细节。

例19-5使用Director面向方面设计模式（参见第23章）来提供一个抽象方面，可以使用它来应用迭代器模式。

例19-5：使用方面定义迭代器模式

```
public abstract aspect IteratorPattern
{
   public interface Aggregate
   {
      public Iterator createIterator();

      public Iterator createReverseIterator();
   }
}
```

讨论

IteratorPattern抽象方面通过定义一个聚合必须实现的接口(这意味着可以迭代任何对象集合)，来声明Aggregate（聚合）角色。这个接口包括两个操作：创建正常的迭代器或倒排迭代器。Java标准库在这里做了一些简化，因为通用方面可以使用java.util包中的Iterator接口来实现模式的迭代器角色。

图19-9显示了IteratorPattern抽象方面的结构，以及它定义用于支持迭代器设计模式的接口和行为。

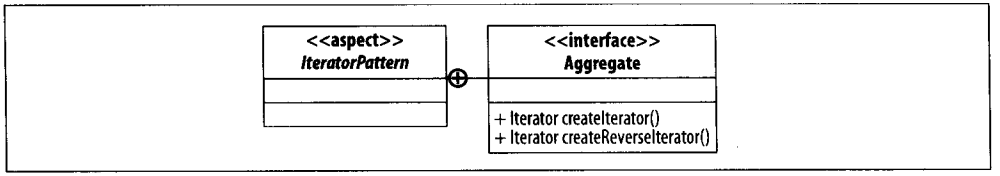

图 19-9：IteratorPattern 方面以及它为设计模式的角色定义的接口

例 19-6 说明了如何将 IteratorPattern 抽象方面应用于特定的应用程序。

例 19-6：为一组特定的类特殊化 IteratorPattern 抽象方面

```
public aspect EmployeeIteration extends IteratorPattern
{
   declare parents : EmployeeCollection implements Aggregate;

   public Iterator EmployeeCollection.createIterator()
   {
      return new EmployeeIterator(this, true);
   }

   public Iterator EmployeeCollection.createReverseIterator()
   {
      return new EmployeeIterator(this, false);
   }
}
```

EmployeeIteration方面与支持的EmployeeIterator一起声明EmployeeCollection是一个聚合，然后为该集合实现迭代器创建方法。

图 19-10 显示了在应用 EmployeeIteration 方面之前和之后的 EmployeeCollection 类。

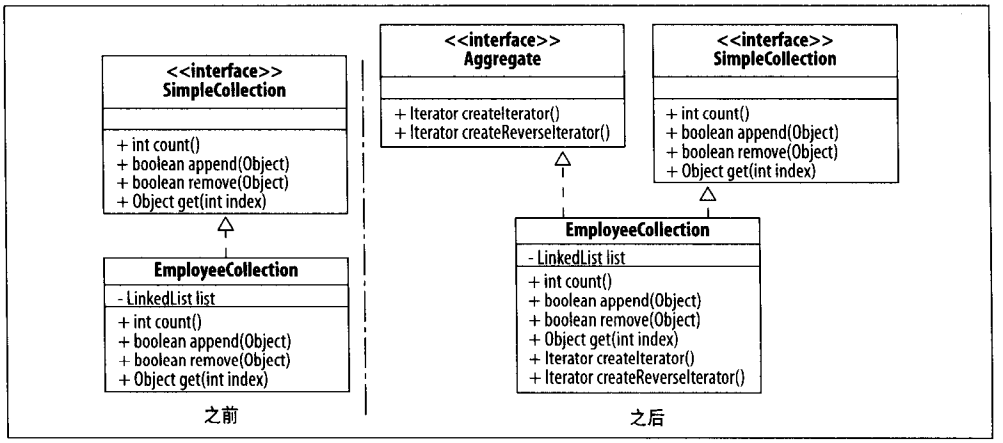

图 19-10：将迭代器模式应用于 EmployeeCollection 类之后的静态结构

图19-11显示了EmployeeCollection类的新迭代器模式行为在示例应用程序内如何交互。

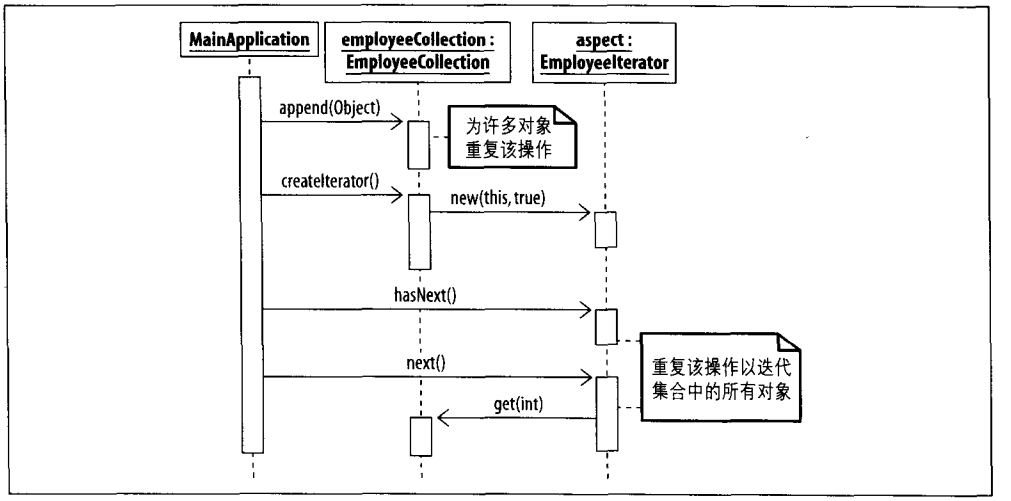

图 19-11：使用 EmployeeCollection 类新的迭代器模式特征

参考

有关定义抽象方面和特殊化它们的更多信息，参见第 15 章；第 16 章包含关于一些机制的更多详细信息，利用这些机制，可以使用方面和 declare 关键字来扩展现有的类；23.3 节解释了 Director 面向方面设计模式。

19.4 实现调停者模式

问题

你想使用 AspectJ 应用调停者模式（mediator pattern）。

解决方案

调停者模式通过在调停者角色中提供单点依赖性，将适合同事角色的潜在大量类彼此隔开。扮演调停者角色的类通过提供一个公共点来控制同事启动的不同事件，从而把同事类之间的依赖性减至最小。调停者自身会接受事件，然后封装一个逻辑，用于通知原始事件的合适同事。

例19-7中的抽象方面使用Director面向方面设计模式（参见第23章）提供一些机制，可以利用这些机制将调停者模式应用于应用程序。

例 19-7：使用方面定义调停者模式

```
public abstract aspect MediatorPattern
{
   protected interface Colleague
   {
   }

   protected interface Mediator
   {
   }

   private WeakHashMap mappingColleagueToMediator = new WeakHashMap();

   private Mediator getMediator(Colleague colleague)
   {
      Mediator mediator =
         (Mediator) mappingColleagueToMediator.get(colleague);
      return mediator;
   }

   public void setMediator(Colleague c, Mediator m)
   {
      mappingColleagueToMediator.put(c, m);
   }

   protected abstract pointcut change(Colleague c);

   after(Colleague c) : change(c)
   {
      notifyMediator(c, getMediator(c));
   }

   protected abstract void notifyMediator(Colleague c, Mediator m);
}
```

讨论

MediatorPattern抽象方面把Colleague（同事）和Mediator（调停者）角色定义为接口，可以通过继承子方面将其用于应用程序特定的类。这个方面定义了mappingColleagueToMediator查寻，可以使用setMediator(Colleague,Mediator)方法操纵它，将同事对象赋予调停者对象。

MediatorPattern方面提供了由子方面实现的change(..)抽象切入点，在同事发生变化时，这个切入点通过调用notifyMediator(Colleague,Mediator)方法（它也由子方面实现）来触发调停者上的通知。

图 19-12 显示了 MediatorPattern 抽象方面的结构，以及它定义用于支持调停者设计模式的接口和行为。

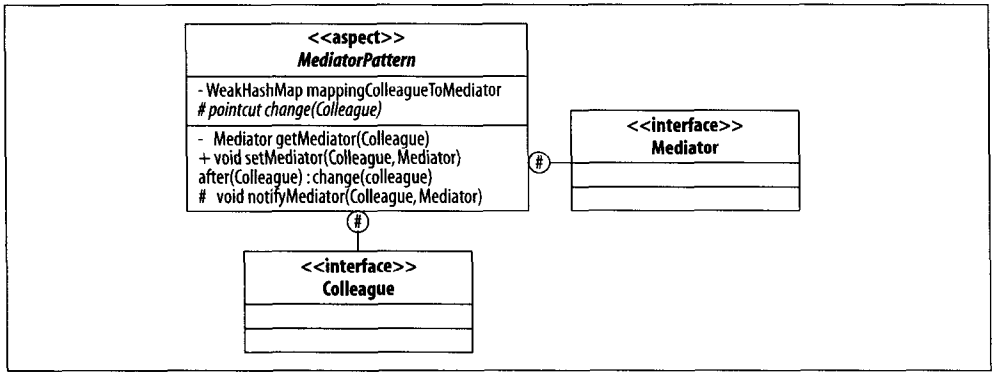

图 19-12：MediatorPattern 方面以及它为设计模式的角色定义的接口

例 19-8 说明了如何把 MediatorPattern 抽象方面应用于特定的应用程序。

例 19-8：将 MediatorPattern 方面应用于应用程序的类

```
public aspect DialogMediator extends MediatorPattern
{
   declare parents : ListBox implements Colleague;
   declare parents : EntryField implements Mediator;

   protected pointcut change(Colleague c) : (
         execution(void ListBox.setSelection(..)) && this(c));

   protected void notifyMediator(Colleague c, Mediator m)
   {
      ListBox listBox = (ListBox) c;
      EntryField entryField = (EntryField) m;
      entryField.setText(listBox.getSelection());
   }
}
```

图 19-13 显示了在应用 DialogMediator 方面之前和之后的 ListBox 类和 EntryField 类。

图 19-14 显示了 ListBox 类和 EntryField 类的调停者模式特征在示例应用程序内如何一起交互。

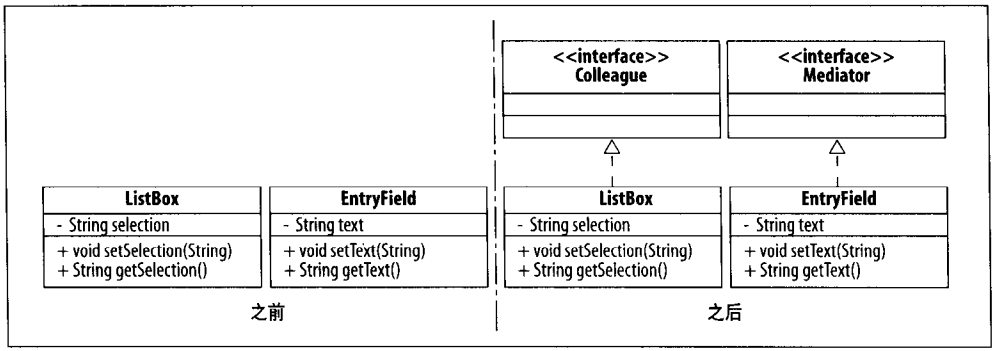

图 19-13：将调停者模式应用于 ListBox 类和 EntryField 类之后的静态结构

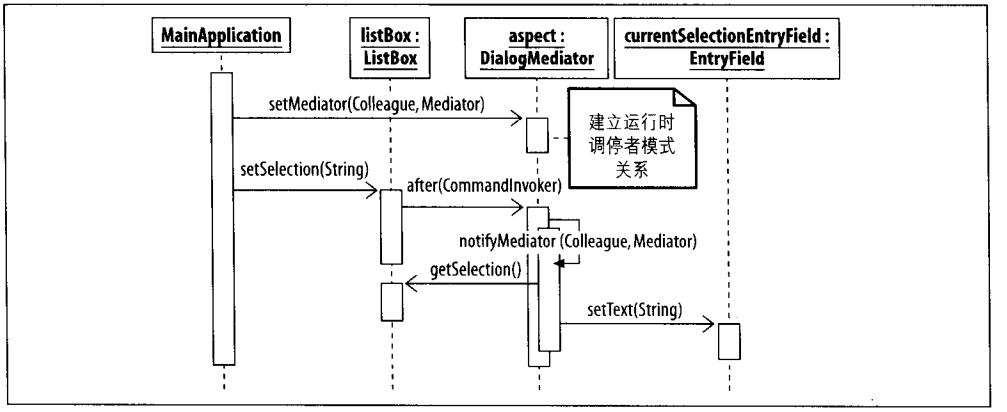

图 19-14：使用 ListBox 类和 EntryField 类新的调停者模式行为

参考

观察者模式的工作方式比调停者模式更具层次性，但是，在依据它们对于特定设计问题的合适度来比较这二者时，其实现可能是有用的。17.1 节说明了使用方面实现观察者模式；16.1 节提供了关于使用方面来扩展现有类的更多信息；23.2 节解释了 Director 面向方面设计模式。

19.5 实现责任链模式

问题

你想使用 AspectJ 应用责任链模式（chain of responsibility pattern）。

解决方案

责任链模式允许把请求的来源与决定应该使用哪个处理程序来处理请求隔离开，这个处理程序是从潜在的大量针对请求的处理程序中选择的。代表链角色的类把请求从其来源，沿着处理程序列表，一直导向一个处理程序，它会接受请求并处理它。

例 19-9 中的抽象方面使用 Director 面向方面设计模式（参见第 23 章）提供一些通用机制，通过这些机制可以将责任链模式应用于应用程序。关于如果链中没有处理程序可用于接受请求则会发生什么情况的决定是一个特定于应用程序的决定；尽管默认的行为是在抽象方面中实现的，但可以通过扩展实现来重写它。

例 19-9：使用方面定义责任链模式

```
public abstract aspect ChainOfResponsibilityPattern
{
   protected interface Handler
   {
   }

   public WeakHashMap successors = new WeakHashMap();

   protected void receiveRequest(Handler handler, Object request)
   {
      if (handler.acceptRequest(request))
      {
         handler.handleRequest(request);
      }
      else
      {
         // The handler will not accept the request
         Handler successor = getSuccessor(handler);
         if (successor == null)
         {
            // Last handler in the chain so must deal with the request
            // This is a rudimentary implementation and more complex
            // logic could be applied here or perhaps in the concrete
            // aspects that extend this abstract one
            handler.handleRequest(request);
         }
         else
         {

            // Hand the request on to the next successor in the chain
            receiveRequest(successor, request);
         }
      }
   }

   public boolean Handler.acceptRequest(Object request)
   {
```

```
        // The default as defined here is to reject the request
        // This is implemented by the application specific
        // concrete aspects
        return false;
    }

    public void Handler.handleRequest(Object request)
    {
        // A default empty implementation that is overridden
        // if required by the application specific concrete aspects
    }

    protected abstract pointcut eventTrigger(Handler handler, Object request);

    after(Handler handler, Object request) : eventTrigger(handler, request)
    {
        receiveRequest(handler, request);
    }

    public void setSuccessor(Handler handler, Handler successor)
    {
        successors.put(handler, successor);
    }

    public Handler getSuccessor(Handler handler)
    {
        return ((Handler) successors.get(handler));
    }
}
```

讨论

ChainOfResponsibilityPattern 抽象方面定义了 Handler 接口，然后可通过特殊化的子方面将其应用于参与链中的特定应用程序内的所有类。方面维护链，并询问将如何通过特定子方面处理请求的细节。

图 19-15 显示了 ChainOfResponsibilityPattern 抽象方面的结构，以及它定义用于支持责任链设计模式的接口和行为。

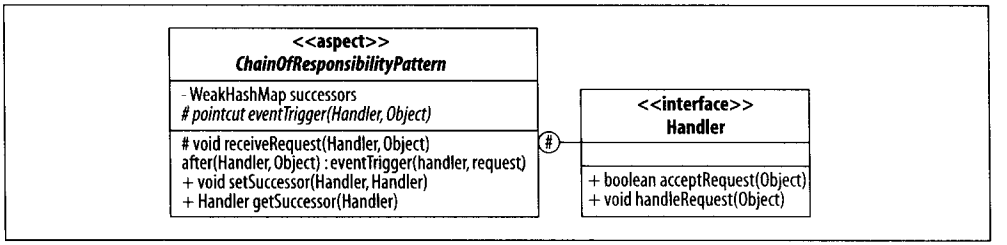

图 19-15：ChainOfResponsibilityPattern 方面以及它为设计模式的角色定义的接口

例 19-10 说明了如何将 ChainOfResponsibility 抽象方面应用于特定的应用程序。

例 19-10：将 ChainOfResponsibility 方面应用于应用程序的类

```
public aspect HelpChain extends ChainOfResponsibilityPattern
{
   declare parents : PrintButton implements Handler;
   declare parents : PrintDialog implements Handler;
   declare parents : Manager implements Handler;

   protected pointcut eventTrigger(Handler handler, Object event) :
         call(void PrintButton.doClick(..))
         && target(handler)
         && args(event);

   private boolean Handler.alreadyHandledRequest = false;

   public boolean Handler.acceptRequest(Object event)
   {
      return !this.alreadyHandledRequest;
   }

   public void PrintButton.handleRequest(Object event)
   {
      if (!this.acceptRequest(event))
      {
         System.out.println(
            "PrintButton Forced to handle Request" +
            "due to being last in the chain (Implementation Decision)");
      }
      System.out.println("PrintButton handling request: " + event);
      this.alreadyHandledRequest = true;
   }

   public void PrintDialog.handleRequest(Object event)
   {
      if (!this.acceptRequest(event))
      {
         System.out.println(
            "PrintDialog Forced to handle Request" +
            "due to being last in the chain (Implementation Decision)");
      }
      System.out.println("PrintDialog handling request: " + event);
      this.alreadyHandledRequest = true;
   }

   public void Manager.handleRequest(Object event)
   {
      if (!this.acceptRequest(event))
      {
         System.out.println(
            "Manager Forced to handle Request due to being" +
            "last in the chain (Implementation Decision)");
      }
```

```
        System.out.println("Manager handling request: " + event);
        this.alreadyHandledRequest = true;
    }
}
```

顾名思义，例19-10中的HelpChain方面实现了一条帮助链，其中针对帮助信息的请求将被传递给作为注册处理程序的类，直到一个类接受请求，或者它是链中的最后一个类为止。

图 19-16 显示了在应用 HelpChain 方面之前的 PrintButton 类、PrintDialog 类和 Manager 类。

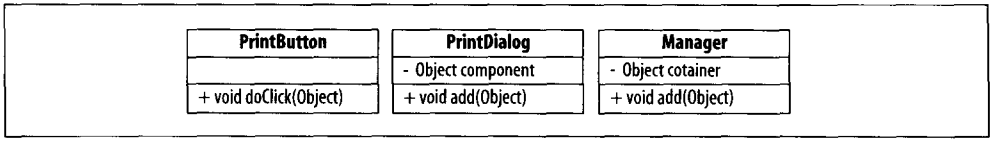

图 19-16：PrintButton 类、PrintDialog 类和 Manager 类

图 19-17 显示了将 HelpChain 方面应用于 PrintButton 类、PrintDialog 类和 Manager 类的效果。

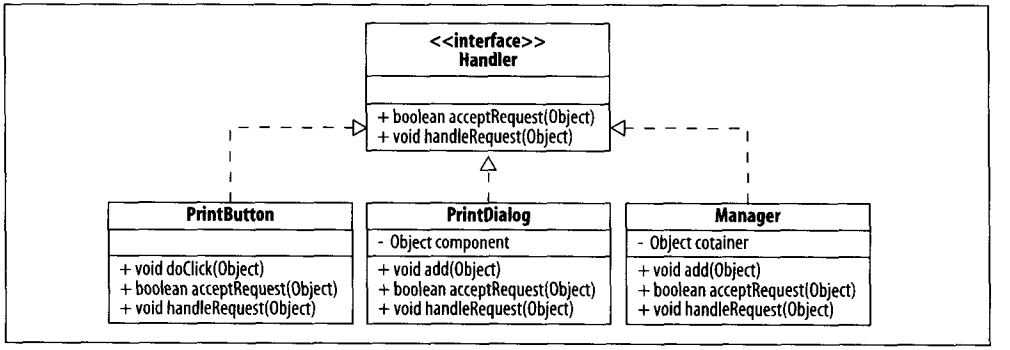

图 19-17：将责任链模式应用于 PrintButton 类、PrintDialog 类和 Manager 类之后的静态结构

图 19-18 显示了一个与图 19-17 中的类交互的示例，它使用了责任链模式特性引入的方面。

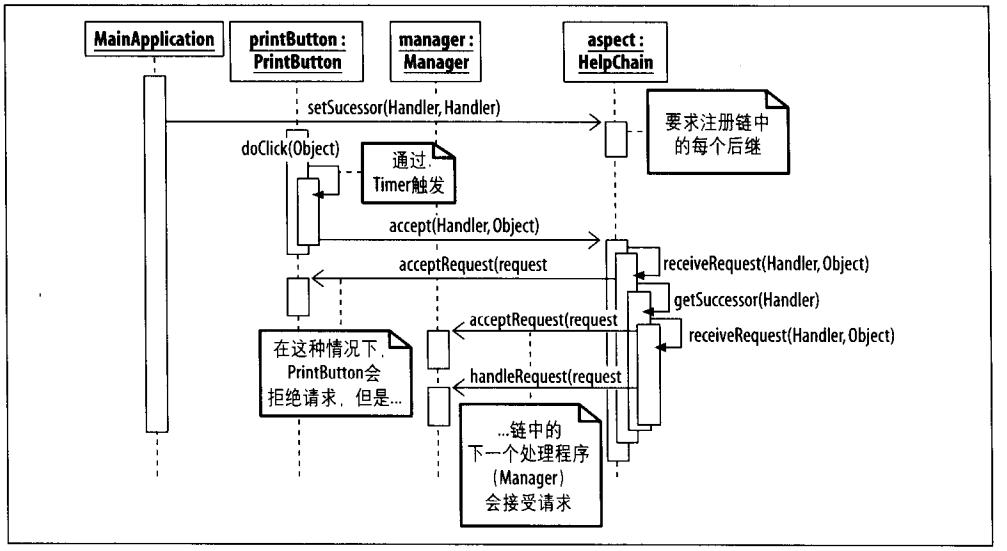

图 19-18：使用责任链模式来处理 PrintButton 类和 Manager 类之间的请求

参考

调停者模式（如19.4节所示）通常与责任链模式结合使用，其中需要一个公共调停者来处理应用程序的许多对象，它使用责任链在整个应用程序中传递请求；16.1节提供了关于使用方面来扩展现有类的更多信息；23.3节解释了 Director 面向方面设计模式。

19.6 实现备忘录模式

问题

你想使用 AspectJ 应用备忘录模式（memento pattern）。

解决方案

备忘录模式提供了一种机制，通过这种机制，可以在以后的某个时间恢复对象的原始状态，而无需把准确的机制与要回滚状态的对象耦合起来。备忘录封装了在以后某个时间恢复对象的先前内部状态所需的全部信息。这种能力可用于提供某种形式的撤销特性，以恢复特定应用程序内对象的状态。

例19-11中的抽象方面和支持类使用Director面向方面设计模式（参见第23章）来指定一些机制，可以使用 AspectJ 通过这些机制将备忘录模式应用于应用程序。

例 19-11：使用方面定义备忘录模式

```
public abstract aspect MementoPattern
{
   public interface Memento
   {
      public void setState(Originator originator);
      public Object getState();
   }

   public interface Originator
   {
      public void setMemento(Memento memento);
      public Memento createMemento();
      public Object getState();
   }
}
```

讨论

MementoPattern方面依据设计模式定义了备忘录和发起者对象的角色和行为。可以通过特殊化的子方面将这些角色应用于应用程序特定的类。

图 19-19 显示了 MementoPattern 抽象方面的结构，以及它定义用于支持备忘录设计模式的接口和行为。

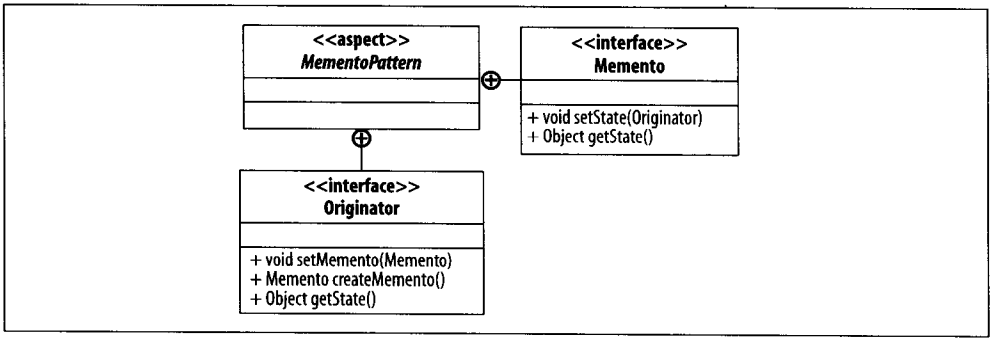

图 19-19：MementoPattern 方面以及它为设计模式的角色定义的接口

例 19-12 说明了如何将 MementoPattern 抽象方面应用于特定的应用程序。

例 19-12：将 MementoPattern 方面应用于应用程序的类

```
public aspect EmployeeMemento extends MementoPattern
{
   declare parents : Employee implements Originator;

   public void Employee.setMemento(Memento memento)
   {
```

```
        Object object = memento.getState();
        Employee stateToRestore = (Employee) object;
        this.setName(stateToRestore.getName());
        this.setSalary(stateToRestore.getSalary());
    }

    public Memento Employee.createMemento()
    {
        Memento memento = new DefaultMemento();
        memento.setState(this);
        return memento;
    }

    public Object Employee.getState() throws MementoException
    {
        Employee employee = new Employee(this.getName(), this.getSalary());
        return employee;
    }
}
```

EmployeeMementor指定Employee类是一个发起者，从而支持创建其状态的备忘录。为了完全支持这个新角色，利用createMemento()、setMemento(Memento)和getState()这些方法扩展Employee类。

createMemento()方法允许客户获得Employee对象的备忘录，setMemento(Memento)方法将把Employee对象恢复到备忘录中存储的状态，当创建备忘录以访问和恢复Employee发起者对象时将使用getState()方法。

图19-20显示了在将EmployeeMemento方面应用于Employee类之前和之后的效果。

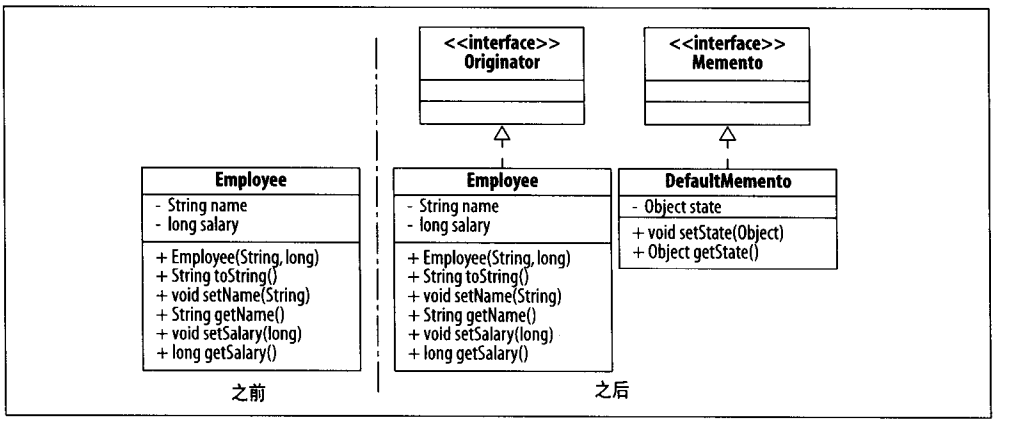

图19-20：在将备忘录模式应用于Employee类之前和之后的静态结构

图19-21显示了使用备忘录模式特性引入的方面与Employee类交互的示例。

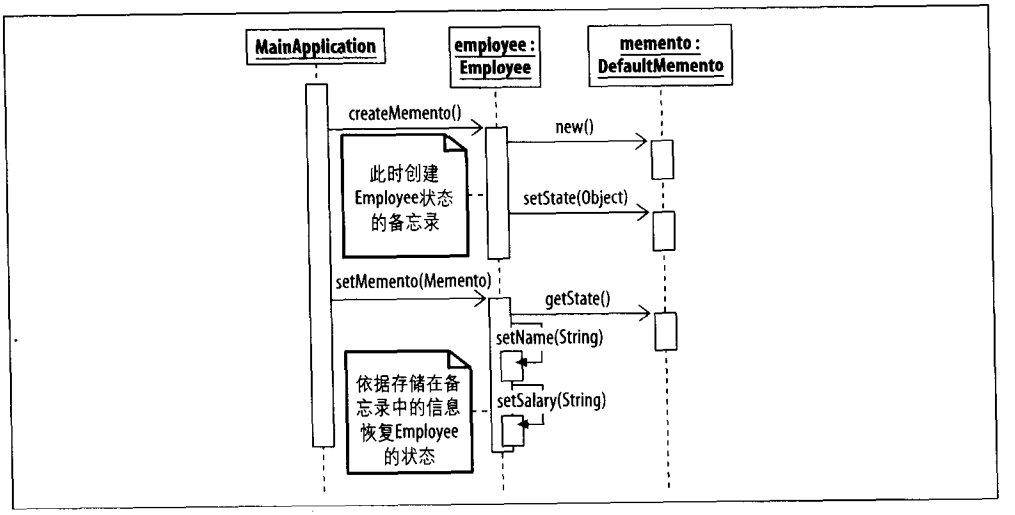

图 19-21：使用 Employee 类的新备忘录模式特征

参考

第 16 章中的秘诀包含关于一些机制的更多详细信息，利用这些机制，可以使用方面和 `declare` 关键字来扩展现有的类；23.3 节解释了 Director 面向方面设计模式。

19.7 实现策略模式

问题

你想使用 AspectJ 应用策略模式（strategy pattern）。

解决方案

策略模式提供了一种机制，用于把客户类与特定算法或策略的实际实现细节隔离开。传统上，实现策略的所有单独的类都会实现一个不同的接口，允许把客户与不同的实现分开。

例 19-13 中的抽象方面使用 Director 面向方面设计模式（参见第 23 章）来提供策略模式的一种通用实现，可以把它应用于应用程序。

例 19-13：使用方面定义策略模式

```
public abstract aspect StrategyPattern
{
   Hashtable strategyPerContext = new Hashtable();

   protected interface Strategy
   {
   }

   protected interface Context
   {
   }

   public void setConcreteStrategy(Context c, Strategy s)
   {
      strategyPerContext.put(c, s);
   }

   public Strategy getConcreteStrategy(Context c)
   {
      return (Strategy) strategyPerContext.get(c);
   }
}
```

讨论

StrategyPattern抽象方面将Strategy（策略）和Context（环境）角色定义为接口。使用一个散列表来查寻要使用的特定的具体策略。

图 19-22 显示了 StrategyPattern 抽象方面的结构，以及它定义用于支持策略设计模式的接口和行为。

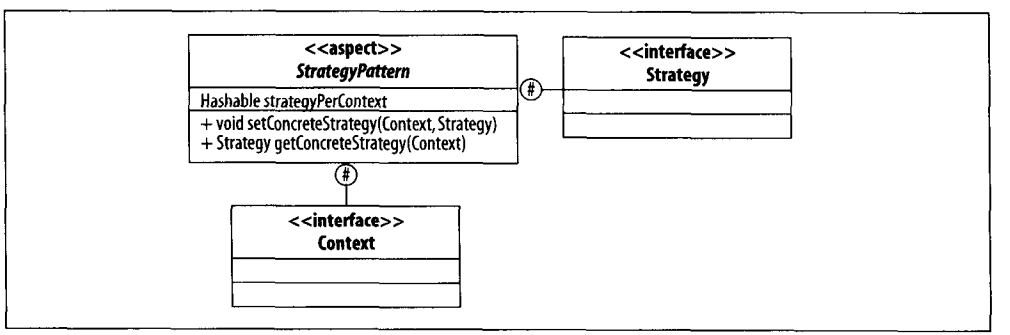

图 19-22：StrategyPattern 方面以及它为设计模式的角色定义的接口

例19-14说明了如何将StrategyPattern抽象方面应用于特定的应用程序。Sorting-Strategy方面定义了将策略角色映射到两种排序器实现，以及把总体环境映射到一个Sorter类。应用通知来重写Sorter类上的排序方法，以应用合适的排序策略。

例 19-14：将 StrategyPattern 方面应用于应用程序

```
public aspect SortingStrategy extends StrategyPattern
{
   declare parents : Sorter implements Context;
   declare parents : LinearSorter implements Strategy;
   declare parents : BubbleSorter implements Strategy;

   int[] around(Sorter s, int[] numbers) : call(int[] Sorter.sort(int[]))
      && target(s)
      && args(numbers)
   {
      Strategy strategy = getConcreteStrategy(s);
      if (strategy instanceof BubbleSorter)
         ((BubbleSorter) strategy).sort(numbers);
      else if (strategy instanceof LinearSorter)
         ((LinearSorter) strategy).sort(numbers);
      return numbers;
   }
}
```

SortingStrategy方面为待排序的特定环境声明了两种不同的排序策略。将执行的实际策略依赖于在运行时为特定环境声明的策略类型。

图 19-23 显示了应用 SortingStrategy 方面之前的 Sorter 类、LinearSorter 类和 BubbleSorter 类。

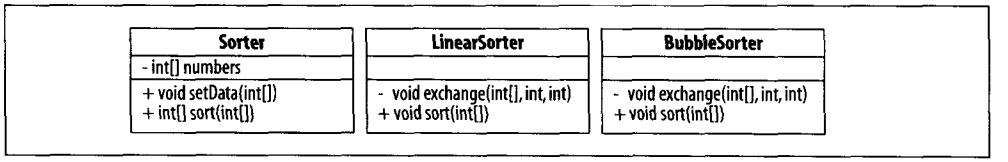

图 19-23：Sorter 类、LinearSorter 类和 BubbleSorter 类

图 19-24 显示了将 SortingStrategy 方面应用于图 19-23 中的类的效果。

图 19-25 显示了使用策略模式特性引入的方面与 Sorter 类、LinearSorter 类和 BubbleSorter 类交互的示例。

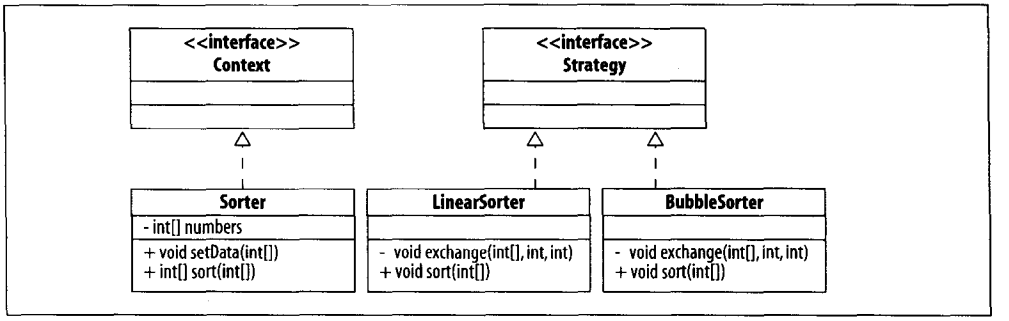

图 19-24：将策略模式应用于 Sorter 类、LinearSorter 类和 BubbleSorter 类之后的静态结构

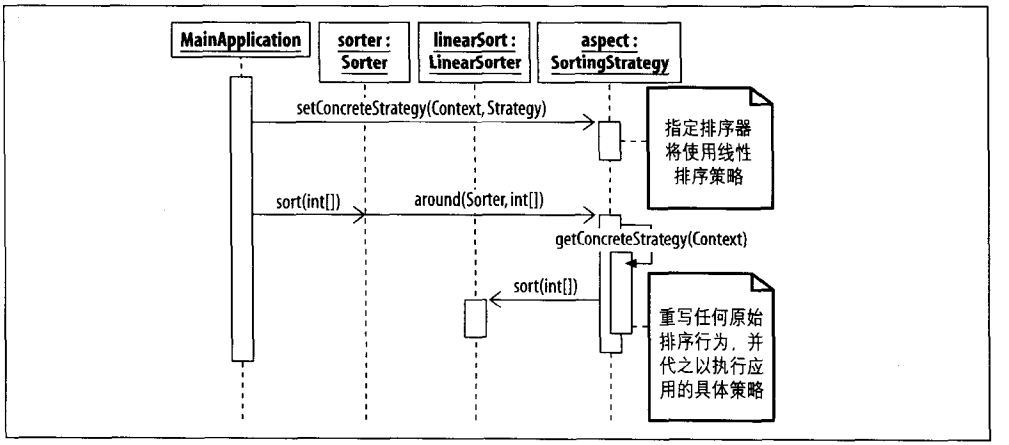

图 19-25：使用 Sorter 类和 LinearSorter 类的新策略模式特征

参考

第 16 章中的秘诀包含关于一些机制的更多详细信息，利用这些机制，可以使用方面和 `declare` 关键字来扩展现有的类；第 15 章研究了声明和使用抽象方面；11.3 节描述了 `args(Type | Identifier)` 切入点；11.2 节研究了 `target(Type | Identifier)` 切入点；4.1 节介绍了 `call(Signature)` 切入点；13.2 节研究了展示连接点环境；23.3 节解释了 Director 面向方面设计模式。

19.8 实现访问者模式

问题

你想使用 AspectJ 应用访问者模式（visitor pattern）。

解决方案

访问者模式封装了一个请求，当在整个结构中传递它时，可以通过对象的层次结构来执行它。例 19-15 中的抽象方面使用 Director 面向方面设计模式（参见第 23 章）来定义参与访问者模式的角色。

例 19-15：使用方面定义访问者模式

```
public abstract aspect VisitorPattern
{
   public interface Element
   {
      public void accept(Visitor visitor);
   }

   public interface CompositeElement extends Element
   {
      public Element[] getElements();
   }

   public interface Result
   {
   }

   public interface Visitor
   {
      public void visitElement(Element element);
      public void visitComposite(CompositeElement element);
      public Result getResult();
   }

   public void CompositeElement.accept(Visitor visitor)
   {
      visitor.visitComposite(this);
   }

   public void Element.accept(Visitor visitor)
   {
      visitor.visitElement(this);
   }
}
```

讨论

VisitorPattern抽象方面把CompositeElement（复合元素）和Element（元素）角色定义为待访问的对象结构的一部分。Visitor（访问者）角色描述了如何通知Visitor它正在访问哪种元素。这个角色应用于可能传递到结构不同部分的对象，它们是复合元素或简单元素。然后扩展CompositeElement和Element角色，以提供用于传递Visitor的方法。

图19-26显示了VisitorPattern抽象方面的结构，以及它定义用于支持访问者设计模式的接口和行为。

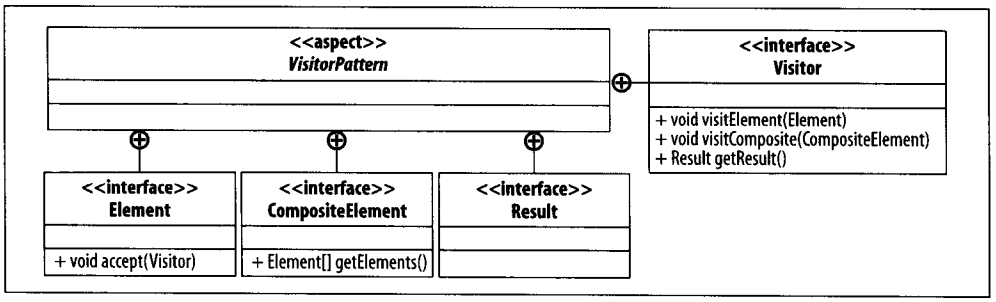

图19-26：VisitorPattern方面以及它为设计模式的角色定义的接口

例19-16说明了如何将VisitorPattern抽象方面应用于特定的应用程序。

例19-16：将VisitorPattern方面应用于应用程序

```
public aspect InventoryVisitor extends VisitorPattern
{
   declare parents : FloppyDisk implements Element;
   declare parents : HardDisk implements Element;
   declare parents : Processor implements Element;
   declare parents : Computer implements CompositeElement;
   declare parents : Motherboard implements CompositeElement;
   declare parents : InventoryReport implements Result;

   public Element[] Computer.getElements()
   {
      Element[] elements = new Element[3];
      elements[0] = this.getMotherboard();
      elements[1] = this.getHardDisk();
      elements[2] = this.getFloppyDisk();
      return elements;
   }

   public Element[] Motherboard.getElements()
   {
      Element[] elements = new Element[1];
```

```
        elements[0] = this.getProcessor();
        return elements;
    }
}
```

图 19-27 显示了在应用 InventoryVisitor 方面之前的一组示例应用程序类。

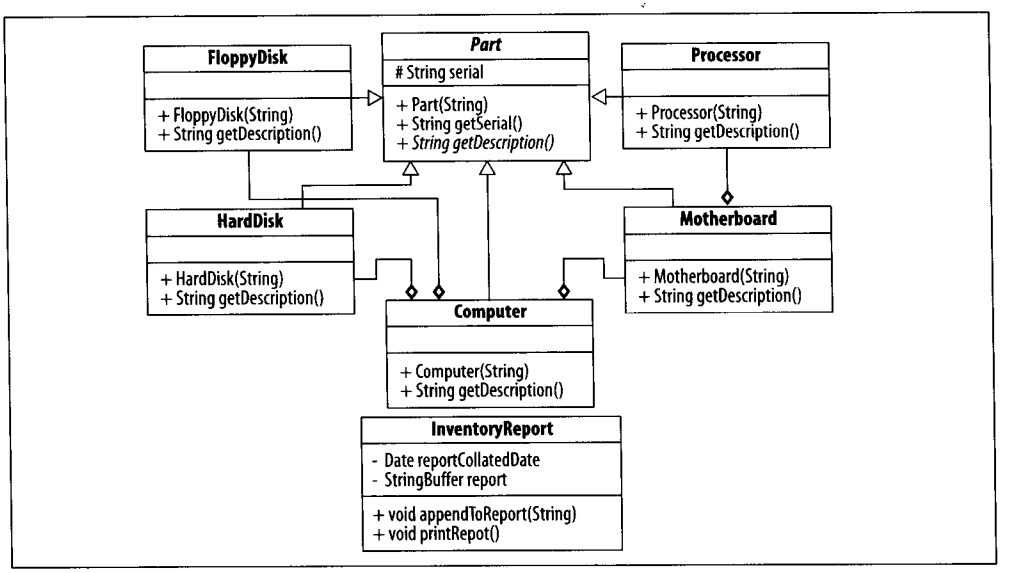

图 19-27：代表计算机组件的类之间的关系

图 19-28 显示了将 InventoryVisitor 方面应用于图 19-27 所示的应用程序类的效果。

图19-29显示了使用访问者模式特性引入的方面与图19-28中的一些类交互的部分示例。这些交互会在所有复合元素和单个元素中递归地继续，这会在构成顶级 Computer 对象的每个元素上调用 accept(Visitor)，然后调用 getDescription()。

参考

访问者模式通常与18.1节中说明的复合模式一起使用；有关定义抽象方面和特殊化它们的更多信息，请参阅第 15 章中的秘诀；第 16 章中的秘诀包含关于一些机制的更多详细信息，利用这些机制，可以使用方面和 declare 关键字来扩展现有的类；23.3 节解释了 Director 面向方面设计模式。

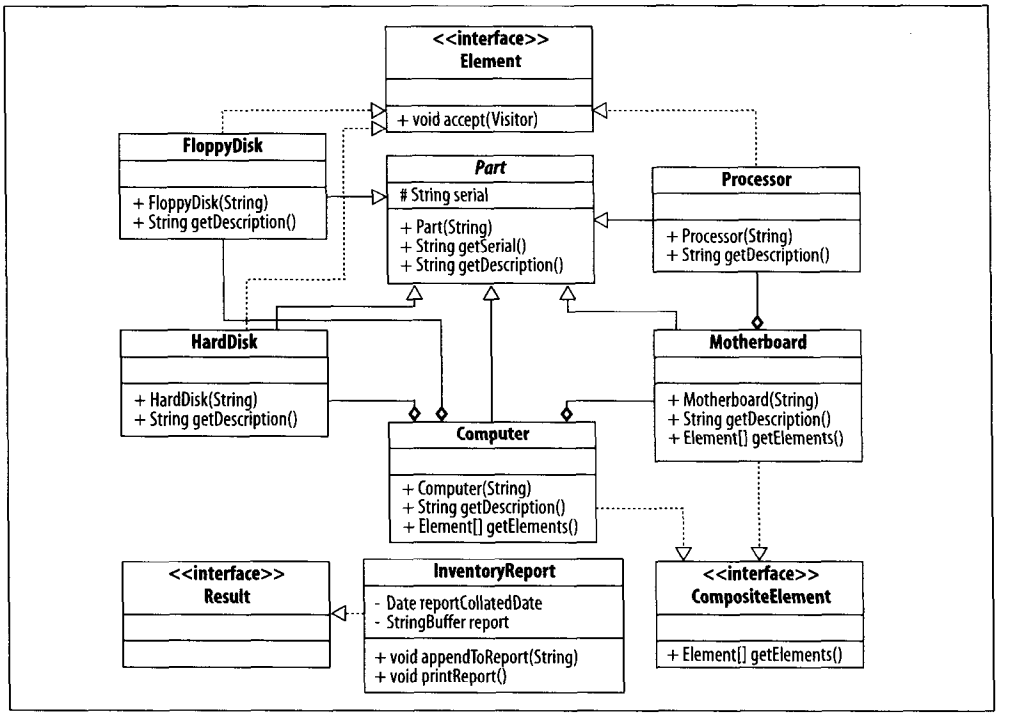

图 19-28：应用了新的访问者模式接口的计算机部件类

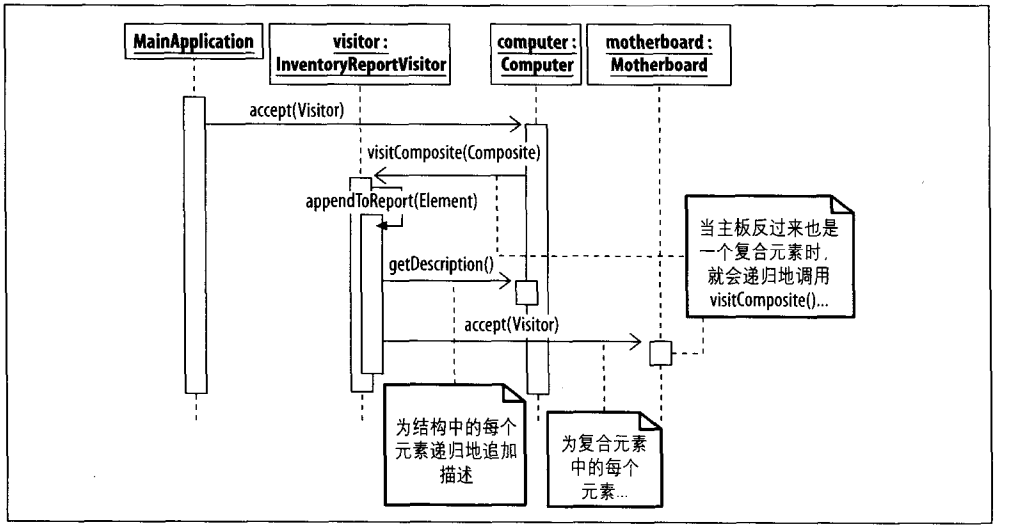

图 19-29：与 Computer 类、Motherboard 类和 InventoryReportVisitor 类的访问者模式特征的部分交互

19.9 实现模板方法模式

问题

你想使用 AspectJ 应用模板模式（template pattern）。

解决方案

模板方法模式（template method pattern）提供了一种机制，可以利用它来声明通用工作流内的抽象方法。这些抽象步骤或方法是通过特殊化的类实现的。

例 19-17 说明了如何使用方面应用模板方法模式。

例 19-17：使用方面定义模板方法模式

```
public interface AlgorithmDefinition
{
   public String runAlgorithm();

   public StringBuffer stepOne();

   public void stepTwo(StringBuffer data);

   public void stepThree(StringBuffer data);

   public String stepFour(StringBuffer data);
}

public aspect DefaultAlgorithmImplementation
{
   public String AlgorithmDefinition.runAlgorithm()
   {
      StringBuffer dataInProcess = this.stepOne();
      this.stepTwo(dataInProcess);
      this.stepThree(dataInProcess);
      return this.stepFour(dataInProcess);
   }

   public StringBuffer AlgorithmDefinition.stepOne()
   {
      return new StringBuffer();
   }

   public String AlgorithmDefinition.stepFour(StringBuffer data)
   {
      return data.toString();
   }
}
```

讨论

DefaultAlgorithmImplementation方面指定了算法执行步骤的顺序，以及几个步骤的默认实现。通过使用面向方面技术，可以在接口而不是抽象类中声明算法模板。然后，依靠静态横切技术，可以在具体的方面中指定合适的通用步骤的默认实现，包括以正确顺序调用步骤的方法。可以通过实现算法接口的类来完成更特定的步骤，从而在合适的地方自动获得默认的行为。

这种模式的面向方面实现的优点是：它通过把部分抽象实现移到方面中并使用静态横切方法来提供默认在接口上的部分实现，消除了模式中的顶级类必须在抽象类中的限制。这使得设计更灵活，这是由于具体类现在实现了接口，而不是耗尽它们的一个继承关系（如Java中所允许的那样）来纳入模式。

参考

第16章中的秘诀包含关于一些机制的更多详细信息，利用这些机制，可以使用方面来扩展现有的类。

19.10 实现状态模式

问题

你想使用AspectJ应用状态模式（state pattern）。

解决方案

状态模式提供了一种机制，通过这种机制，可以基于对象的状态来改变它的行为。状态被封装在它自己的对象中，这个对象然后又被调用对象所包含。调用对象把受其状态影响的所有方法请求传递给状态对象，状态对象则基于此刻它的类来改变响应。

例19-18说明了如何使用方面来声明伪TCPConnection类的状态，包括在调用方法时如何在这些状态之间转换。

例19-18：使用方面为特定应用程序的类实现状态模式

```
public aspect TCPConnectionState
{
   protected TCPState listening = new TCPListen();

   protected TCPState acknowledged = new TCPAcknowledged();

   protected TCPState closed = new TCPClosed();
```

```
    after(TCPConnection connection) : initialization(new ())
        && target(connection)
    {
        listening.setConnection(new SocketConnection());
        connection.setState(listening);
    }

    after(TCPConnection connection, TCPState state) : call(
        void TCPState +.acknowledge())
        && target(state)
        && this(connection)
    {
        if (connection.getState() == listening)
        {
            acknowledged.setConnection(listening.getConnection());
            connection.setState(acknowledged);
        }
    }

    after(TCPConnection connection, TCPState state) : call(
        void TCPState +.close())
        && target(state)
        && this(connection)
    {
        if ((connection.getState() == listening)
            || (connection.getState() == acknowledged))
        {
            connection.setState(closed);
        }
    }
}
```

讨论

TCPConnectionState方面指定：TCPConnection类状态将在创建时被侦听，在调用acknowledge()方法时被确认，以及在调用close()调用时被关闭。它还指定：如果依赖于这些方法的顺序调用它们，则会发生什么。无需影响TCPConnection类，即可完成所有这些操作。

这个解决方案中的方面用于模块化从一种状态变成另一种状态的规则。通过模块化这些规则，分析状态改变的规则就会更容易，因为它们都在一个位置，并且能够在不影响原始类的情况下，更灵活地执行将来的状态改变，如纳入新的或变化的状态。这提供了超过更传统的面向对象方法的独特优点，在面向对象方法中，用于决定状态变化的逻辑必须嵌入在方法中，因而可能会混淆业务逻辑。

图 19-30 显示了在TCPConnectionState方面管理的类上调用的方法序列如何影响TCPConnection的状态。

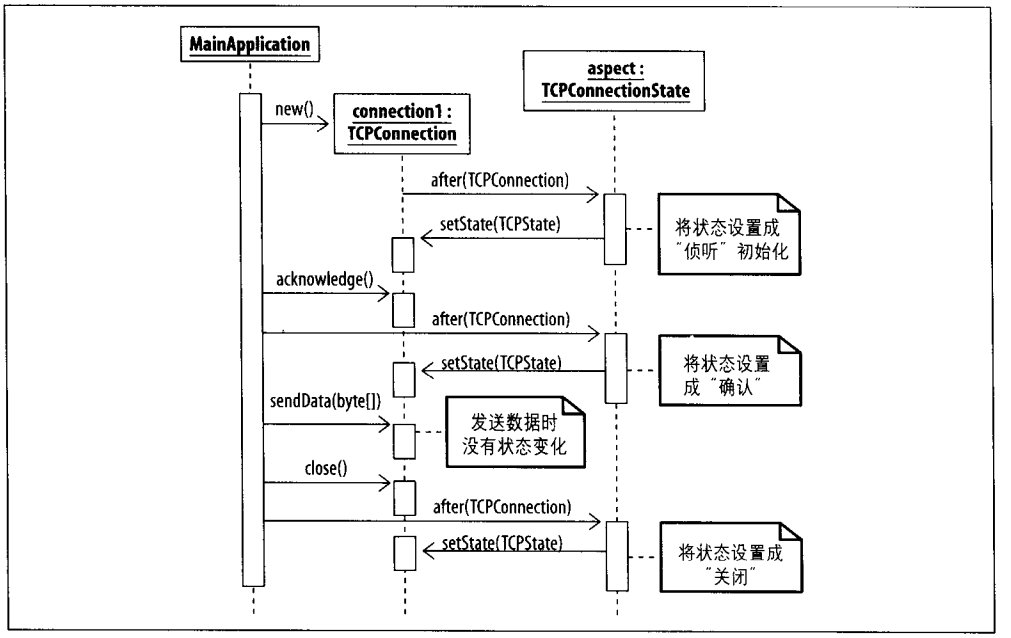

图 19-30：使用 TCPConnectionState 方面管理 TCPConnection 对象的状态

参考

4.1 节介绍了 `call(Signature)` 切入点；13.5 节讨论了 `after()` 形式的通知；11.3 节描述了 `this(Type | Identifier)` 切入点；11.2节研究了 `target(Type | Identifier)` 切入点；13.2 节研究了展示连接点环境；7.3 节研究了 `initialization(Signature)` 切入点。

19.11 实现解释器模式

问题

你想利用使用AspectJ实现外观模式(façade pattern)和解释器模式(interpreter pattern)的好处。

解决方案

不幸的是，此时，根据定义解释器模式看起来不会从利用 AspectJ 使用面向方面方法来

获得任何设计或实现上的好处。解释器模式的实现显式地绑定到正被解释的结构的类上；因此，尽管概念是通用的，但是实现是特定的，并且看起来不会从面向方面方法获益。

讨论

指出解释器模式和外观模式不会从 AspectJ 获益回避了如下问题的实质，即为什么还要包括它们呢？首先，包含所有其他的GoF设计模式而仅仅把这两个模式排除在外看起来似乎比较古怪。其次，特别是面向方面的外观模式证明可能是有用的。这些情况处于一个灰色区，它们直接从 AspectJ 织入库（*.jar* 文件）和编译代码（*.class* 文件）的能力获益。

AspectJ 可以并且确实能够织入 Java 源代码和经过编译的字节码。当希望决定应用程序是否为后来的合法构建过程包含特定的方面时，这是有用的。不过，额外的特性是：可以潜在地织入遗留的和第三方的库，其中的源代码是不可用的。由于更改第三方专有库的法律和技术因素，这尤其是一个灰色区。AspectJ 确实提供了一种方式，如果克服了所有其他的技术性和非技术性障碍，则可通过这种方式来做到这一点，尽管在选择这种做法时要加倍小心。

参考

2.4 节说明了使用 `ajc` 命令来编译方面，并将其织入到 *.jar* 库中。

第 20 章

应用类和组件级方面

20.0 简介

似乎可以很容易地把方面归类为系统级或企业级关注点，因为这些环境可以从面向方面的方法中获得大多数显而易见的好处。不过，任何横切行为都是使用 AspectJ 的实现的良好候选，而不管该行为是跨企业、系统，还是跨单个类或组件。

后 3 章将研究可以在如下 3 种不同的软件开发环境内应用的不同类型的方面：企业、系统和组件。本章重点介绍微观方面，它们显然是应用在类级或组件级横切关注点中。

你最好原谅这种想法：即只有很少的事务可以描述为单个类或组件内的横切。通过更近地观察，你将发现可以许多行为，它们都会影响作为方面的强大候选的类的特征，特别是当这些特征不是类的业务逻辑的核心时则更是如此。

本章中研究的这些微观横切关注点的示例包括从验证传递给方法的参数，到控制类的实例化。这些领域都会从模块化以及与类的核心业务关注点隔离开而获益，并且说明了使用 AspectJ 可以改进设计和实现，甚至在类级和组件级也是如此。

20.1 验证传递给方法的参数

问题

你想验证传递给方法的参数。

解决方案

创建一个模块化参数检查逻辑的方面。声明一个切入点，用于捕获其中将检查参数的方法的执行。切入点应该把参数展示给相应的通知，使得它可以执行检查。

依赖于参数检查的结果，通知将继续执行方法，或者每当参数不合适时都重写方法。

讨论

每当开发人员无法控制其客户使用组件的方式时，都不得不关心验证方法参数的琐事。甚至当开发人员正在编写将使用特定组件的客户时，一个有用的惯例是：在使用传递给方法的值之前，验证这些值是可接受的。

这种检查通过引入初始检查代码，它与方法将完成的工作具有少许关系，通常会混淆方法内的逻辑。通过使用 AspectJ，可以独立于方法的核心业务逻辑检查方法的参数，从而使得解决方案更清晰。

Java 应用程序中的 main 方法是一个公共位置，在应用程序可以继续执行之前，在这个位置中提供的参数不受控制，并且必须对它们执行检查。完成这些检查的正常方式是：在 main 方法中使用一条 if 语句来编码测试逻辑，如果在命令行上提供的参数不满足应用程序的要求，就产生一条错误消息。

例20-1显示了一个典型的 main 方法，其中突出显示了传统的参数验证和检查支持代码。

例 20-1：把传统的命令行参数检查与 main 方法的业务逻辑交织在一起

```
package com.oreilly.aspectjcookbook;

import java.net.URL;
import java.net.MalformedURLException;

public class TraditionalMainApplication
{
   private static final String COMMAND_LINE_USAGE =
      "TraditionalMainAppliction usage :\n\n" +
      "\tjava TraditionalMainApplication <url>";

   public static void main(String[] args)
   {
      if (args.length == 1)
      {
         try
         {
            // Assuming that the first argument supplied is a url
            // Concentrating on the business logic, not validation which
            // is handled by the aspect.
            URL url = new URL(args[0]);
```

```
            System.out.println("Application Started, doing stuff with " + url);
        }
        catch (MalformedURLException mue)
        {
            System.err.println(COMMAND_LINE_USAGE);
            System.err.println("Please enter a valid URL for <url>");
        }
    }
    else
    {
        System.err.println(COMMAND_LINE_USAGE);
    }
  }
}
```

首先，TraditionalMainApplication类声明了COMMAND_LINE_USAGE常量，它包含一条消息，如果在把参数传递给main方法时发生问题，就会把这条消息显示给用户。然后，main方法内的一条if语句会检查是否提供了正确数量的参数；如果不是，就会显示错误消息。最后，其中一个参数是URL，因为这可能是无效的，所以会通过try/catch块检查它。这一切都发生在main方法可以做任何有用的事情（如启动应用程序）之前。

将检查逻辑重构到一个单独的方法中就把问题转移到同一个类的另一部分中，并且不会提供令人满意的解决方案。AspectJ可以通过创建一个如例20-2中所示的方面，删除应用程序开始的一些杂乱细节。

例20-2：使用方面检查参数

```
package com.oreilly.aspectjcookbook;

import java.net.URL;
import java.net.MalformedURLException;

public aspect VerifyMethodArgsAspect
{
   private static final String COMMAND_LINE_USAGE =
      "MyAppliction usage :\n\n" +
      "\tjava MainApplication <url>";

   public pointcut captureMain(String[] arguments) :
      execution(void MainApplication.main(String[])) &&
      args(arguments);

   public pointcut createURLCalledinMainMethod() :
      call(java.net.URL.new(..)) &&
      withincode(public void MainApplication.main(String[]));

   void around(String[] arguments) : captureMain(arguments)
   {
      if (arguments.length == 1)
      {
```

```
      // Test that the host and port are valid
         try
         {
            URL url = new URL(arguments[0]);
            proceed(arguments);
         }
         catch(MalformedURLException mfe)
         {
            System.err.println(COMMAND_LINE_USAGE);
            System.err.println("Please enter a valid URL for <url>");
         }
      }
      else
      {
         System.err.println(COMMAND_LINE_USAGE);
      }
   }

   // If necessary soften the exception that would normally have been raised
   // if the url parameter was badly formed, but only in the validated main method
   declare soft : MalformedURLException : createURLCalledinMainMethod();

}
```

VerifyMethodArgsAspect 方面声明 captureMain(String[])切入点，它会中断 main 方法的执行，并把参数提供给相应的通知。around()通知然后会检查参数的数量，如果所有的一切都检验合格，则在用原始参数调用proceed()之前，around()通知会检查URL参数是否有效。如果参数发生任何问题，则以与以前相同的方式显示错误消息。

使用方面来模块化参数检查逻辑，可以使得main方法看起来要整洁得多，并且不会丢失任何重要的检查行为，如例 20-3 所示。

例 20-3：通过一个方面提供参数检查逻辑，可以让 main 方法专注于它自己的工作

```
package com.oreilly.aspectjcookbook;

import java.net.URL;

public class MainApplication
{

   public static void main(String[] args)
   {
      // Assuming that the first argument supplied is a url
      // Concentrating on the business logic, not validation which
      // is handled by the aspect.
      URL url = new URL(args[0]);

      System.out.println("Application Started, doing stuff with " + url);
   }
}
```

main 方法逻辑仍会像以前那样使用 URL 参数，但是，由于方面提供了必要的检查以确保不会发生问题，所以可以相当安全地使用 declare soft 语句，使得 main 方法不必显式地处理 MalformedURLException。

createURLCalledinMainMethod()切入点会捕获通常会在哪里引发异常 —— 也就是说，在创建 java.net.URL 实例时并且在 main 方法内 —— 并且这是可以在其中减轻 MalformedURLException 影响的唯一连接点。如果不减轻异常的影响，main 方法将不得不处理抛出 MalformedURLException 的可能性，甚至使用基于方面的检查也是如此。

参考

4.4 节描述了 execution(Signature) 切入点；11.3 节介绍了 args([TypePatterns || Identifiers]) 切入点；13.4 节说明了 around() 形式的通知；16.5 节讨论了减轻异常的影响。

20.2 重写在构造函数调用上实例化的类

问题

你想能够控制在构造函数调用上实例化的实际类。

解决方案

使用 Cuckoo's Egg 面向方面设计模式创建一个方面，用于模块化构造函数调用的重写，从而能够改变在运行时返回的实现。使用 call(Signature) 切入点捕获对类构造函数的调用，然后使用 around() 通知返回一个不同的对象。

讨论

在不影响使用接口或基类的代码的情况下，能够迁移接口或基类特殊化的不同实现可能是有用的。传统上，在对象构造时决定所选的类，如例 20-4 所示。

例 20-4：*硬编码所选的 MyInterface 的 MyClass 实现*

```
public static void traditionalObjectOrientedImplementationSelection()
{
   MyInterface myObject = new MyClass(); // Specifies the MyClass implementation
                                         // of the MyInterface interface

   System.out.println(myObject);
   myObject.foo(); // Calls the MyClass implementation of the foo() method
}
```

这段代码会产生如下输出：

```
com.oreilly.aspectjcookbook.MyClass@cf8583
foo() called on MyClass
```

为了更改MyInterface的实现，需要把代码更改为：

```
public static void traditionalObjectOrientedImplementationSelection()
{
   MyInterface myObject = new AnotherClass(); // Specifies the AnotherClass
                                              // implementation  of the
                                              // MyInterface interface

   System.out.println(myObject);
   myObject.foo(); // Calls the AnotherClass implementation of the foo() method
}
```

这将产生如下输出：

```
com.oreilly.aspectjcookbook.AnotherClass@dbe178
foo() called on AnotherClass
```

在调用foo()方法时，客户代码不会关心MyInterface接口的实现是什么，因为在Java中，方法是基于对象类型在运行时动态调用的。这种方法的优点是：在实例化实现类(MyClass或AnotherClass)时，会公开展示它。无论在什么地方使用这个接口，至少可以出现一行特定于实现的代码，如果想更改实现，则在迁移至新类时，将会导致易出错的更改连锁反应。

解决这个问题的一种常见方式是使用一个工厂方法或者抽象工厂设计模式。面向方面的技术为这两种方法都提供了优点，分别如13.3节和13.4节所示。不过，通过使用AspectJ的call(Signature)切入点和around()通知，可以使得事情更简单、更整洁，如例20-5所示。

例20-5：使用一个方面来重写类的实例化

```
package com.oreilly.aspectjcookbook;

public aspect ControlClassSelectionAspect
{
   public pointcut myClassConstructor() : call(MyClass.new());

   Object around() : myClassConstructor()
   {
      return new AnotherClass();
   }
}
```

ControlClassSelectionAspect 声明了 myClassConstructor()切入点，它会截获调用，以实例化MyClass对象。相应的around()通知然后会返回重写AnotherClass类的一个新实例。

警告： 从 around()通知实例化并返回的类必须是重写构造函数所期待类型的子类型。例如，在例20-5中，AnotherClass必须把MyClass扩展为所指定的正被重写的构造函数调用，它正在创建一个新的MyClass()。如果around()通知返回一个不是MyClass子类型的对象，则无论何时应用 around()通知，都会在运行时得到 ClassCastException 错误。

使用这种技术，根本不需要修改例 20-4 中的代码，即可使用 MyInterface 接口的 AnotherClass 实现。如果把 ControlClassSelectionAspect 织入到应用程序中，AnotherClass 实现就会自动重写 MyClass 实现。

可以进一步细化myClassConstructor()切入点，使得可以在某些环境中重写MyClass构造函数。例如，可以添加within(TypePattern)或withincode(Signature)切入点定义，以声明重写只应该出现在指定的作用域内。

可以检查用户定义的运行时参数，以改变在 around()通知内实例化的实现。

```
Object around() : myClassConstructor() &&
                  if (System.getProperty("select_class").equals("AnotherClass"))
{
   return new AnotherClass();
}
```

这种变体通过更改应用程序的select_class运行时参数的值，允许更改在运行时实例化的类，而不会强制重新编译应用程序。

注意： 预先说明一下，23.1 节显示了 Cuckoo's Egg 面向方面设计模式的典型应用。这种模式为重写构造函数调用的问题提供了一个公共解决方案，用于返回非预期的不同对象。

参考

7.1 节描述了用于捕获构造函数上的调用时的 call(Signature)切入点；13.4 节讨论了 around()形式的通知；9.1 节和 9.3 节分别描述了 within(TypePattern)切入点和 withincode(Signature)切入点；23.1 节讨论了 Cuckoo's Egg 面向方面设计模式。

20.3 添加持久性到类中

问题

你想添加持久性到类中，使得它可以保存和恢复。

解决方案

应用 Director（导向者）面向方面设计模式，并创建一个抽象方面，它定义了管理对象持久性所需的角色和行为，如例 20-6 所示。

可以把这个抽象方面扩展成一些特殊化的子方面，它们将为应用程序内的每个持久对象集合实现一个合适的持久性机制。

例 20-6：在抽象方面中定义持久性的通用角色和行为

```
public abstract aspect PersistenceAspect
{
   public interface ObjectStore
   {
      public void persist();
      public void restore();
   }

   protected abstract pointcut restoreStorage(ObjectStore store);

   after(ObjectStore store) : restoreStorage(store)
   {
      store.restore();
   }

   protected abstract pointcut persistStorage(ObjectStore store);

   after(ObjectStore store) : persistStorage(store)
   {
      store.persist();
   }
}
```

讨论

PersistenceAspect 抽象方面把 ObjectStore 角色定义为接口，可以将其应用于任何类，来管理对象集合的持久性。restoreStorage(ObjectStore) 和 persistStorage(ObjectStore) 抽象切入点通过特殊化的子方面来实现，用于触发对应的 after() 通知块，它将恢复或保持指定的 ObjectStore。

ObjectStore接口中指定的restore()和persist()方法是依据特定的持久性策略实现的，该策略用于应用程序内的对象集合，使得可以基于每个ObjectStore来改变持久性策略。

例20-7说明了如何实现抽象的PersistenceAspect，给EmployeeCollection类中作为集合管理的Employee类提供持久性。

例 20-7：将 Persistence 抽象方面应用于 Employee 类和 EmployeeCollection 类

```
public aspect EmployeePersistenceAspect extends PersistenceAspect
{
   declare parents : EmployeeCollection implements ObjectStore;

   protected pointcut restoreStorage(ObjectStore store) :
      execution(EmployeeCollection.new(..)) &&
      target(store);

   protected pointcut persistStorage(ObjectStore store) :
      call(* java.util.List.add(..)) &&
      target(EmployeeCollection) &&
      target(store);

   declare parents : Employee extends Serializable;

   private File EmployeeCollection.employeesFile = new File("employees.ser");

   public void EmployeeCollection.persist()
   {
      try
      {
         ObjectOutput out = new ObjectOutputStream(
            new FileOutputStream(this.employeesFile));

         Object[] objectsToStore = this.toArray();
         out.writeObject(objectsToStore);
         out.flush();
         out.close();
      }
      catch (Exception e)
      {
         System.err.println("Couldn't store employees to " + this.employeesFile);
      }
   }

   public void EmployeeCollection.restore()
   {
      if (this.employeesFile.exists() && this.employeesFile.canRead())
      {
         try
         {
            ObjectInput input = new ObjectInputStream(
               new FileInputStream(this.employeesFile));
```

```
            Object[] objectsToRestore = (Object[]) input.readObject();
            for(int x = 0; x < objectsToRestore.length; x++)
            {
               this.add(objectsToRestore[x]);
            }

            input.close();
         }
         catch (Exception e)
         {
            System.err.println(
               "Couldn't restore employees due to a corrupt " +
                this.employeesFile +
               " file");
            e.printStackTrace();
         }
      }
   }
}
```

EmployeePersistenceAspect将ObjectStore接口应用于EmployeeCollection类。然后实现restoreStorage(ObjectStore)方面，以捕获何时构造EmployeeCollection，并使用target(TypePattern || Identifier)切入点将EmployeeCollection展示为要存储的ObjectStore。

实现persistStorage(ObjectStore)切入点，用以捕获无论何时更改EmployeeCollection，并在此时保持ObjectStore的内容。EmployeeCollection是java.util.ArrayList的特殊化；为了避免直接织入到Java标准库中，使用call(Signature)切入点来捕获何时在java.util.List上调用add(..)方法。

不过，call(Signature)切入点定义本质上过于普通，因此必须将通过persistStorage(ObjectStore)切入点捕获的连接点限制于只捕获何时在EmployeeCollection上调用add(..)。为了应用这种限制，第一个target(TypePattern || Identifier)切入点使用TypePattern来指定你只对其目标是EmployeeCollection类的连接点感兴趣。第二个target(TypePattern || Identifier)切入点使用一个标识符将当前ObjectStore传递给通知块，作为persistStorage(ObjectStore)切入点的单一参数。

最后，将直观的对象串行化持久性策略应用于EmployeeCollection。EmployeeCollection中的每个对象都是Employee类的一个实例，并且扩展这个类以实现Serializable接口，使得可对它应用标准的Java对象串行化技术。串行化的Employee对象将存储在一个文件中；因此，将以employeesFile属性的形式把这个文件信息添加到EmployeeCollection类中。

为了完成图形，将把persist()和restore()方法实现添加到EmployeeCollection类中，使得它可以满足ObjectStore接口所需的行为。这些方法会执行Employee对象的串行化和恢复，这是通过employeesFile属性指定的文件来进行的。

在例20-7中，每次更改EmployeeCollection时都会保持它的状态。这对于应用程序可能有些矫枉过正，但是可以更改通过persistStorage(ObjectStore)切入点捕获的连接点，将ObjectStore保持为更适合于自己的应用程序。

例20-8显示了抽象的PersistenceAspect替代实现的一部分，它只会在关闭应用程序时保持其对应的ObjectStore。

例20-8：声明在关闭应用程序时，应该保持一个ObjectStore

```
public privileged aspect AccountPersistenceAspect extends PersistenceAspect
{
   declare parents : MainApplication implements ObjectStore, Runnable;

   protected pointcut restoreStorage(ObjectStore store) :
      execution(MainApplication.new(..)) &&
      target(store);

   protected pointcut persistStorage(ObjectStore store) :
      execution(public void MainApplication.run()) &&
      this(store);

   declare parents : Account extends Serializable;

   private File MainApplication.accountsFile = new File("accounts.ser");

   public void MainApplication.persist()
   {
      // ... Code to persist the Accounts ...
   }

   public void MainApplication.restore()
   {
      // ... Code to restore the Accounts ...
   }

   after(MainApplication mainApplication) :
      restoreStorage(ObjectStore) &&
      target(mainApplication)
   {
      // Register a shutdown hook
      Thread shutdownThread = new Thread(mainApplication);
      Runtime.getRuntime().addShutdownHook(shutdownThread);
   }

   public void MainApplication.run()
   {
      // Do nothing, merely provides the trigger that the shutdown hook has been
      // executed so as to persist the store on shutdown.
```

```
    }
}
```

AccountPersistenceAspect声明MainApplication类将实现ObjectStore接口，以及添加Runnable接口。Runnable接口（来自于java.lang）在Java中用作所有对象的接口，可以在这些对象自己的线程中执行它。Runnable接口需要public void run()方法，为Java虚拟机（JVM）提供入口点，以执行它自己线程中的类。

在这个示例中，将persistStorage(ObjectStore)切入点修改成捕获Runnable的执行，它在MyApplication类上强制执行public void run()方法。然后，AccountPersistenceAspect把必要的run()方法实现添加到MainApplication类中，以满足Runnable接口的需要，但是，这只会提供一个标记，而不需要任何实现。

增加Runnable接口和MainApplication类上的run()存根方法意味着：可以利用JVM把MainApplication注册为关闭挂钩（shutdown hook），使得在整个应用程序正常完成时将调用MainApplicatin.run()方法。在恢复ObjectStore（它是一个MainApplication类）时，通过执行around(MainApplication)通知块，来完成把MainApplication类注册为关闭挂钩的任务。

通过把MainApplication用作关闭挂钩，当触发关闭挂钩时，persist-Storage(ObjectStore)切入点将触发MainApplication对象的保持操作。因此，一旦干净利索地关闭应用程序，就会保持MainApplication中存储的Account类的集合。

参考

4.1节讨论了call(Signature)切入点；4.4节解释了execution(Signature)切入点；9.1 节解释了this(TypePattern | Identifier)切入点；9.2 节描述了target(TypePattern | Identifier)切入点；13.5节描述了after()形式的通知；15.2 节详细讨论了特殊化抽象方面；16.2 节解释了使用declare parents语句扩展现有的类；23.2 节解释了 Director 面向方面设计模式。

20.4 应用模拟组件支持单元测试

问题

你想通过提供组件使用的测试装置模块（test harness）实现，孤立地对自己的组件进行单元测试。

解决方案

创建组件所依赖的外部组件的模拟实现。创建一个方面，应用模拟组件实现来代替真实的组件。

单元测试完成时，使用单独的AspectJ构建配置文件转换出测试方面，使得可以再次使用真实的实现。

讨论

在创建软件组件时，孤立地对那些组件进行单元测试，以确保组件如所预期的那样工作。不过，当你的组件依赖于其他组件时，它可能难以处理，并且需要手动解除与外部组件的所有依赖性挂钩以进行单元测试，以及在单元测试完成时恢复这些挂钩，而这些操作容易出错。

例20-9显示了一种典型情况，其中MyComponent是要测试的组件，并且它具有与ThirdPartyComponentInterface的外部实现的依赖性。ThirdPartyComponentInterface的真实实现是通过调用工厂方法ThirdPartyFactory.getThirdPartyComponent()来获得的。

例20-9：创建和使用外部组件

```
package com.ourcompany;

import com.thirdparty.ThirdPartyComponentFactory;
import com.thirdparty.ThirdPartyComponentInterface;

public class MyComponent implements MyComponentInterface
{
   private ThirdPartyComponentInterface thirdPartyComponent;

   public MyComponent()
   {
      this.thirdPartyComponent = ThirdPartyComponentFactory.getThirdPartyComponent();
      System.out.println("Component found " + thirdPartyComponent);
   }

   public void foo()
   {
      System.out.println("Inside MyComponent.foo()");
      this.thirdPartyComponent.bar();
   }
}
```

注意： 例20-9说明了工厂类的使用，但是这不是这个秘诀解决方案的必不可少的特性。使用20.2节中所示的相同技术直接重写第三方组件的实例化是可能的，这无需用到工厂。

为了孤立地对MyComponent运行单元测试，将需要通过在测试中包括真实的外部组件，重写ThirdPartyComponent实现，从而不会混淆测试结果。一种策略是：手动应用重写真实组件实现的模拟组件，如例20-10所示。

例20-10：创建模拟类，然后手动修改组件以使用新的模拟对象

```
package test.com.ourcompany;

import com.thirdparty.ThirdPartyComponentInterface;

public class MockThirdPartyComponent implements ThirdPartyComponentInterface
{
   public void bar()
   {
      System.out.println("Inside MockThirdPartyComponent.bar()");
   }
}

package com.ourcompany;

import com.thirdparty.ThirdPartyComponentFactory;
import com.thirdparty.ThirdPartyComponentInterface;
import test.com.ourcompany.MockThirdPartyComponent;

public class MyComponent implements MyComponentInterface
{
   private ThirdPartyComponentInterface thirdPartyComponent;

   public MyComponent()
   {
      // Manually commented out for unit testing purposes
      // this.thirdPartyComponent =
      //    ThirdPartyComponentFactory.getThirdPartyComponent();

      // Replaced with mock object
      this.thirdPartyComponent = new MockThirdPartyComponent();

      System.out.println("Component found " + thirdPartyComponent);
   }

   public void foo()
   {
      System.out.println("Inside MyComponent.foo()");

      this.thirdPartyComponent.bar();
   }
}
```

与ThirdPartyComponentInterface的真实实现不同的是，MockThirdPartyComponent在你的控制之下，因此可以定制组件，以正确地测试MyComponent。MyComponent不再依赖于ThirdPartyComponentInterface的真实实现，因此可以安全地得出如下结论：在测试时发生的任何问题都是你自己的问题。

在完成单元测试并对MyComponent正确地工作感到满意之后，可以删除模拟组件，并取消对原始代码的注释。使用这种方法，切换回ThirdPartyComponentInterface的真实实现是另一种手动并且潜在地易出错的任务。

尽管例20-10中的解决方案工作得很好，但是，如果组件使用了较多的接口，它可能就是一种难以管理的方法。使用面向方面的替代方法，可以创建一个方面，用于截获ThirdPartyComponent的创建，并用模拟对象实现重写返回的对象，如例20-11所示。

例20-11：使用一个方面将模拟对象应用于孤立的单元测试

```
package test.com.ourcompany;

import com.thirdparty.*;

public aspect MockThirdPartyComponentAspect
{
   public pointcut catchThirdPartyConstructor() :
      call(ThirdPartyComponentInterface ThirdPartyComponentFactory.
               getThirdPartyComponent());

   Object around() : catchThirdPartyConstructor()
   {
      return new MockThirdPartyComponent();
   }
```

catchThirdPartyConstructor()切入点捕获对ThirdPartyFactory.getThirdPartyComponent()方法的调用，并使用关联的around()通知返回MockThirdPartyComponent对象，而不是ThirdPartyComponentInterface的真实实现。通过使用方面而不是手动更改代码，在把方面织入到应用程序中之后，就会自动把组件准备好进行孤立的单元测试。

注意： 与6.3节类似，这个秘诀显示了Cuckoo's Egg面向方面设计模式的另一种典型应用，将在23.1节中讨论它。

可以通过创建两个不同的AspectJ构建配置文件，在编译时切换使用外部组件的真实实现和模拟实现。一个构建配置文件包括测试方面的模拟对象，这将导致一个构建，它为组件的孤立单元测试准备好一切；另一个构建配置文件把模拟对象测试方面排除在外，这将导致一个部署构建，它使用外部组件的真实实现。

AspectJ使得将模拟对象用于测试目的的任务对代码具有较少的侵入性，并使之更加可伸缩，并且大大地减少了在测试和部署构建之间切换时的麻烦。

参考

2.5 节解释了使用不同的构建配置来改变应用于应用程序的方面；4.1 节描述了 `call(Signature)` 切入点；13.4 节介绍了 `around()` 形式的通知；17.3 节和 17.4 节分别说明了如何使用 AspectJ 实现抽象工厂和工厂方法设计模式；23.1 节讨论了 Cuckoo's Egg 面向方面设计模式。

第21章

应用应用程序级方面

21.0 简介

应用程序级方面会影响软件相当多的区域，它们通常是软件的特征，将会影响应用程序中的许多（甚至所有）类。本章重点介绍这些系统级的横切关注点集合，其中可以使用AspectJ更好地实现它们的特征。本章中的秘诀可以分成两类：被动方面和主动方面。

被动方面（passive aspect）是截获器或观察器，或者是应用程序的逻辑，并且不会以明显的方式影响或反馈进这个逻辑中。被动方面的关键特征之一是：它通常只包含`before()`和`after()`通知，如果使用`around()`通知，它将总是调用`proceed()`方法。

主动方面（active aspect）会以各种方式影响它们所应用的应用程序，如更改通向软件的逻辑路径。尽管在编译和织入这两个方面时，它们都会影响应用程序的代码，但是主动方面会更改应用程序的行为。主动方面通常包含`around()`通知，它不能调用`proceed()`方法，但是可以重写作为原始业务逻辑一部分的触发连接点。

本章首先介绍了两个实现跟踪和日志记录的被动应用程序级方面。跟踪和日志记录几乎是AspectJ的“Hello World”用例，这部分是由于它们是灵活的、模块化的应用程序级横切行为的良好例子。尽管跟踪和日志记录不是最令人感兴趣的方面，但是它们是AspectJ中被动方面实现的最强大的例子。

在讨论了更多的主动方面之后，本章最后说明了如何通过使用方面，来模块化应用程序属性的存储方式以及应用和更新它们的方式，从而更好地管理它们。属性管理方面直接与使用那些属性的任何类交互，并且可以在更新属性时监视要注册的那些类的内部细节。

21.1 应用面向方面的跟踪

问题

你想使用 AspectJ 对应用程序的一些区域应用跟踪。

解决方案

创建一个抽象基础方面，如例21-1所示，用以封装通用跟踪逻辑。使用特殊化的子方面扩展通用跟踪方面，以将应用程序的区域包括在跟踪内，或将其排除在外。

例21-1：抽象跟踪方面为应用程序定义要特殊化的通用跟踪逻辑

```
import org.aspectj.lang.JoinPoint;

public abstract aspect TracingAspect
{
   public abstract pointcut pointsToBeTraced();

   public abstract pointcut pointsToBeExcluded();

   public pointcut filteredPointsToBeTraced(Object caller) :
      pointsToBeTraced() &&
      !pointsToBeExcluded() &&
      !within(com.oreilly.aspectjcookbook.tracing.TracingAspect+) &&
      this(caller);

   public pointcut catchStaticCallers() :
      pointsToBeTraced() &&
      !pointsToBeExcluded() &&
      !within(com.oreilly.aspectjcookbook.tracing.TracingAspect+) &&
      !filteredPointsToBeTraced(Object);

   before(Object caller) : filteredPointsToBeTraced(caller)
   {
      traceBefore(thisJoinPoint, caller);
   }

   before() : catchStaticCallers()
   {
      traceStaticBefore(thisJoinPoint);
   }

   after(Object caller) : filteredPointsToBeTraced(caller)
   {
      traceAfter(thisJoinPoint, caller);
   }

   after() : catchStaticCallers()
   {
      traceStaticAfter(thisJoinPoint);
   }
```

```
protected void traceBefore(JoinPoint joinPoint, Object caller)
{
   System.out.println(caller + " calling " +
         joinPoint.getSignature() + " @ " +
         joinPoint.getSourceLocation());
}

protected void traceStaticBefore(JoinPoint joinPoint)
{
   System.out.println("Static code calling " +
         joinPoint.getSignature() + " @ " +
         joinPoint.getSourceLocation());
}

protected void traceAfter(JoinPoint joinPoint, Object caller)
{
   System.out.println("Returning from call to" +
         joinPoint.getSignature() + " @ " +
         joinPoint.getSourceLocation());
}

protected void traceStaticAfter(JoinPoint joinPoint)
{
   System.out.println("Returning from static call to " +
         joinPoint.getSignature() + " @ " +
         joinPoint.getSourceLocation());
}

private static aspect FormatCallDepthAspect
{
   private static int callDepth;

   private pointcut captureTraceBefore() :
      call(protected void TracingAspect.trace*Before(..));

   private pointcut captureTraceAfter() :
      call(protected void TracingAspect.trace*After(..));

   after() : captureTraceBefore()
   {
      callDepth++;
   }

   before() : captureTraceAfter()
   {
      callDepth--;
   }

   private pointcut captureMessageOutput(String message) :
      call(* *.println(String)) &&
      args(message) &&
      within(TracingAspect) &&
      !within(FormatCallDepthAspect);
```

```
    Object around(String originalMessage) : captureMessageOutput(originalMessage)
    {
        StringBuffer buffer = new StringBuffer();
        for (int x = 0; x < callDepth; x++)
        {
            buffer.append("  ");
        }
        buffer.append(originalMessage);

        return proceed(buffer.toString());
    }
  }
}
```

讨论

TracingAspect 具有两个抽象切入点（pointsToBeTraced()和pointsToBeExcluded()），它们允许特殊化的子方面指定要跟踪的目标应用程序区域。

然后，filteredPointsToBeTraced(Object)切入点把pointsToBeTraced()和pointsToBeExcluded()切入点与逻辑组合起来，用于排除TracingAspect本身，以及展示触发跟踪的调用对象。组合的切入点会捕获具有调用对象的目标应用程序内要跟踪的所有连接点。不幸的是，由于使用了this(TypePattern | Identifier)切入点，将从静态代码块中出现的那些连接点中排除filteredPointsToBeTraced(Object)切入点。

catchStaticCallers()切入点通过捕获要包括在跟踪中但不会被filteredPointsToBeTraced(Object)切入点捕获的所有连接点，来解决这个问题。

两组before()和after()通知用于把跟踪消息输出到System.out。其中一组用于在调用对象可用时执行消息输出；另外一组则用于在调用对象不可用时做同样的事情。

如果遗漏了TracingAspect，跟踪消息将稍微有点难以阅读。通过依据当前调用深度缩进每条消息，适当地格式化跟踪消息将是有用的。这被证明是一个方面级横切关注点，因为需要用合适的格式化逻辑来影响方面中的所有通知块。

FormatCallDepth内部方面可以满足跨TracingAspect方面的所有跟踪消息的格式化要求。FormatCallDepth方面通过监视TracingAspect在跟踪前后的执行时间，在当前调用深度上保持一个内部计数器。当TracingAspect对System.out发起一个跟踪调用时，FormatCallDepth方面就会依据当前调用深度来修改跟踪消息。

例21-2说明了如何将TracingAspect方面应用于示例应用程序。

例 21-2：将 TracingAspect 应用于特定的示例应用程序

```
public aspect ApplicationSpecificTracingAspect extends TracingAspect
{
   public pointcut pointsToBeTraced() : call(* *.*(..));

   public pointcut pointsToBeExcluded() : call(void java.io.PrintStream.*(..));
}
```

参考

4.1 节描述了 call(Signature) 切入点；9.1 节描述了 within(TypePattern) 切入点；7.1 节解释了 this(TypePattern. | Identifier) 切入点；7.3 节介绍了 args([TypePatterns || Identifiers]) 切入点；12.2 节和 12.3 节分别描述了 AND (&&) 运算符和 OR (||) 运算符；13.3 节解释了 before() 形式的通知；13.4 节讨论了 around() 形式的通知，包括使用 proceed() 方法；13.5 节解释了 after() 形式的通知；15.2 节解释了使用继承来实现抽象方面；15.4 节解释了定义和使用内部方面。

21.2 应用面向方面的日志记录

问题

你想以模块化以及与应用程序无关的方式应用日志记录。

解决方案

创建一个抽象的 Logging 方面，扩展 21.1 节中的通用 TracingAspect。通过用于异常日志记录的抽象切入点定义和通知扩展 TracingAspect，如例 21-3 所示。新的异常日志记录抽象切入点定义允许特殊化的子方面捕获目标应用程序的区域，其中需要发生异常日志记录。

例 21-3：抽象的日志记录方面定义了要为应用程序进行特殊化的通用日志记录逻辑

```
public abstract aspect LoggingAspect extends TracingAspect
{
   protected abstract pointcut exceptionsToBeLogged();

   private pointcut filteredExceptionCapture() :
      exceptionsToBeLogged() &&
      !pointsToBeExcluded();

   before() : filteredExceptionCapture()
   {
      logException(thisJoinPoint);
   }
```

```
   public abstract void logException(JoinPoint joinPoint);
}
```

讨论

LoggingAspect方面继承21.1节中所示的TracingAspect的所有行为，但是添加了一些特定于日志记录的新能力。提供了exceptionsToBeLogged()抽象切入点，使得特殊化的子方面可以指定要在其中记录异常信息的连接点。logException(JoinPoint)抽象方法允许子方面实现在记录异常时将发生的准确行为。

在实现exceptionsToBeLogged()抽象切入点时，handler(TypePattern)切入点是要在特殊化的子方面中使用的最合适的切入点声明。不过，无法把exceptionsToBeLogged()切入点限制于只用于handler(TypePattern)切入点定义。

然后，filteredExceptionCapture()切入点可以把exceptionsToBeLogged()切入点与继承自TracingAspect的pointcutsToBeExcluded()切入点组合起来，使得从日志记录中继续排除被声明为要排除的任何连接点。

例21-4说明了如何将例21-3中的抽象LoggingAspect应用于示例应用程序。

例21-4：为示例应用程序实现日志记录

```
public aspect ApplicationLoggingAspect extends LoggingAspect
{
   public pointcut pointsToBeTraced() : call(* *.*(..));

   public pointcut pointsToBeExcluded() :  call(* java.io.*.*(..));

   public pointcut exceptionsToBeLogged() :
      handler(com.oreilly.aspectjcookbook.PackageA.BusinessException);

   protected void traceBefore(JoinPoint joinPoint, Object caller)
   {
      System.out.println("Log Message: Called " + joinPoint.getSignature());
   }

   protected void traceStaticBefore(JoinPoint joinPoint)
   {
      System.out.println("Log Message: Statically Called " + joinPoint.getSignature());
   }

   protected void traceAfter(JoinPoint joinPoint, Object object)
   {
      System.out.println("Log Message: Returned from " + joinPoint.getSignature());
   }

   protected void traceStaticAfter(JoinPoint joinPoint)
   {
```

```
        System.out.println("Log Message: Returned from static call to " +
            joinPoint.getSignature());
    }

    protected void logException(JoinPoint joinPoint)
    {
        System.out.println("Log Message: " + joinPoint.getArgs()[0] + " exception thrown");
    }

    private static aspect FormatCallDepthAspect
    {
        private static int callDepth;

        private pointcut captureTraceBefore() :
            call(protected void TracingAspect.trace*Before(..));

        private pointcut captureTraceAfter() :
            call(protected void TracingAspect.trace*After(..));

        after() : captureTraceBefore()
        {
            callDepth++;
        }

        before() : captureTraceAfter()
        {
            callDepth--;
        }

        private pointcut captureMessageOutput(String message) :
            call(* *.println(String)) &&
            args(message) &&
            within(ApplicationLoggingAspect) &&
            !within(FormatCallDepthAspect);

    Object around(String originalMessage) : captureMessageOutput(originalMessage)
    {
            StringBuffer buffer = new StringBuffer();
            for (int x = 0; x < callDepth; x++)
            {
                buffer.append("  ");
            }
            buffer.append(originalMessage);

            return proceed(buffer.toString());
        }
    }
}
```

ApplicationLoggingAspect 方面提供了 pointsToBeTraced()和pointsToBeExcluded()切入点的实现，用于指定要记录的目标应用程序的区域，以及要从记录中排除的区域。新的异常日志记录切入点 exceptionsToBeLogged()被实现用于满足 LoggingAspect 的要求。

出于方便性考虑，在ApplicationLoggingAspect中包括了FormatCallDepth内部方面，使得在通过System.out输出日志记录消息时，它们易于阅读。如果正在使用完整的日志记录解决方案，最有可能不需要这个内部方面，因为日志记录API可能提供它自己的格式化选项。

注意： 可以通过在LoggingAspect方面的特殊化子方面中放置用于初始化和调用日志记录API的代码，插入你想使用的任何日志记录API。

LoggingAspect抽象方面支持多个子方面，它们同时以不同方式潜在地记录应用程序的各个部分。例21-5中的代码示例说明了如何把额外的子方面添加到对特定包内目标的日志调用中，这个包把XML格式用于消息输出。

例21-5：把多个日志记录解决方案应用于同一个应用程序

```
public aspect PackageSpecificLoggingAspect extends LoggingAspect
{
   declare precedence : ApplicationLoggingAspect, PackageSpecificLoggingAspect;

   public pointcut pointsToBeTraced() :
      call(* com.oreilly.aspectjcookbook.PackageA.*.*(..));

   public pointcut pointsToBeExcluded() : call(void java.io.PrintStream.*(..));

   public pointcut exceptionsToBeLogged() : handler(PackageA.*);

   protected void traceBefore(JoinPoint joinPoint, Object object)
   {
      System.out.println("<before>" + joinPoint.getSignature() + "</before>");
   }

   protected void traceStaticBefore(JoinPoint joinPoint)
   {
      System.out.println("<before type=\"static\">" +
         joinPoint.getSignature() + "</before>");
   }

   protected void traceAfter(JoinPoint joinPoint, Object object)
   {
      System.out.println("<after>" + joinPoint.getSignature() + "</after>");
   }

   protected void traceStaticAfter(JoinPoint joinPoint)
   {
      System.out.println("<after type=\"static\">" +
         joinPoint.getSignature() + "</after>");
   }

   protected void logException(JoinPoint joinPoint)
   {
```

```
    System.out.println("<exception>" + joinPoint.getSignature() + "</exception>");
}
```

参考

4.1节描述了call(Signature)切入点；5.1节说明了handler(TypePattern)切入点；12.2节和12.3节分别描述了AND（&&）运算符和OR（||）运算符；13.3节解释了before()形式的通知；13.4节讨论了around()形式的通知，包括使用proceed()方法；13.5节解释了after()形式的通知；15.2节解释了使用继承来实现抽象方面；15.4节解释了定义和使用内部方面。

21.3 应用延迟加载

问题

你想把延迟加载技术应用于应用程序的组件。

解决方案

使用Director面向方面设计模式，将延迟加载中涉及的角色建立为抽象方面内的接口，如例21-6所示。

例21-6：使用Director面向方面设计模式来定义参与延迟加载的角色

```
public abstract aspect LazyLoading extends DelegatingProxyPattern
{
   public interface RealComponent extends Subject
   {
   }

   public interface LazyProxy extends RealComponent
   {
      public RealComponent getRealComponent() throws LazyLoadingException;
   }

   public abstract LazyProxy initializeComponent(Object configuration);
}
```

LazyLoading方面反过来继承自代理模式的实现，如14.6节所示，它重点关注的是代理模式的委托特征，如例21-7所示。

例21-7：特殊化ProxyPattern行为以关注委托

```
public abstract aspect DelegatingProxyPattern extends ProxyPattern
{
```

```
    protected boolean reject(
        Object caller,
        Subject subject,
        JoinPoint joinPoint)
    {
        return false;
    }

    protected boolean delegate(
        Object caller,
        Subject subject,
        JoinPoint joinPoint)
    {
        return true;
    }

    protected Object rejectRequest(
        Object caller,
        Subject subject,
        JoinPoint joinPoint)
    {
        return null;
    }
}
```

最后，创建LazyLoading方面的特殊化子方面，它们将为目标应用程序的特定组件实现延迟加载行为。

讨论

延迟加载涉及将类的加载和实例化延迟到刚好使用实例之前的那一刻。延迟加载的目标是：通过在需要某个对象的那一刻加载和实例化它，根据需要使用内存资源。

例21-6中的LazyLoading方面构建成例21-7中的DelegatingProxyPattern的一部分，用于截获对根据需要延迟加载的类的调用。DelegationProxyPattern反过来又特殊化了14.6节中的ProxyPattern方面，用以限制代理的委托形式的作用域。

RealComponent接口是在LazyLoading方面中声明的，通过针对要延迟加载的目标应用程序中的任何类的特殊化子方面来应用它。需要一个代理对象来存储信息，在需要时，可以利用这种信息实例化真实的类，以及通过LazyProxy接口提供这种角色。

LazyProxy接口提供了足够的功能，用于处理真实的组件，而不必加载它们。LazyProxy定义了单个方法，在需要时可以调用它来加载真实的组件。

最后，LazyLoading抽象方面定义了initializeComponent(Object)抽象方法，通过子方面来实现它，用于实例化延迟代理来代替真实的组件。

例21-8说明了如何将LazyLoading抽象方面应用于示例应用程序。

例21-8：将LazyLoading抽象方面应用于应用程序的类

```
package com.oreilly.aspectjcookbook.feature_management;

import org.aspectj.lang.JoinPoint;
import com.oreilly.aspectjcookbook.lazyloading.LazyLoading;
import com.oreilly.aspectjcookbook.lazyloading.LazyLoadingException;
import com.oreilly.aspectjcookbook.features.Feature;

public aspect LazyFeatureLoading extends LazyLoading
{
   declare parents : Feature implements RealComponent;

   declare parents : LazyProxy implements Feature;

   public LazyProxy initializeComponent(Object object)
   {
      LazyProxy proxy =
         new LazyFeatureProxy((String) object);
      return proxy;
   }

   protected pointcut requestTriggered() :
      call(* com.oreilly.aspectjcookbook.features.Feature.* (..)) &&
      !within(com.oreilly.aspectjcookbook.oopatterns.ProxyPattern+);

   protected Object delegateRequest(
      Object caller,
      Subject subject,
      JoinPoint joinPoint)
   {
      if (subject instanceof LazyFeatureProxy)
      {
         LazyFeatureProxy feature =
            (LazyFeatureProxy) subject;

         try
         {
            Feature implementedFeature =
               (Feature) feature.getRealComponent();

            implementedFeature.doSomething(
               (String) joinPoint.getArgs()[0]);

         }
         catch (LazyLoadingException lle)
         {
            lle.printStackTrace();
            lle.getOriginalException().printStackTrace();
            System.out.println(
                  "Exception when attempting to "
                  + "lazy load"
                  + " a particular class,"
```

```
                + " aborting the call");
        }
    }
    else
    {
        ((Feature) subject).doSomething(
              (String) joinPoint.getArgs()[0]);
    }
    return null;
}

private class LazyFeatureProxy implements Feature, LazyProxy
{
    private Object configuration;
    private Feature delegate;

    public LazyFeatureProxy(Object configuration)
    {
        this.configuration = configuration;
    }

    public synchronized RealComponent getRealComponent()
        throws LazyLoadingException
    {
        if (this.configuration instanceof String)
        {
            try
            {
                if (this.delegate == null)
                {
                    return this.delegate =
                        (Feature) Class
                        .forName((String) this.configuration)
                        .newInstance();
                }
                else
                {
                    return this.delegate;
                }
            }
            catch (Exception e)
            {
                throw new LazyLoadingException(
                    "Exception raised when loading real component", e);
            }
        }
        else
        {
            throw new LazyLoadingException("Error in configuration");
        }
    }

    public void doSomething(String message)
    {
```

```
        }
    }
}
```

LazyFeatureLoading方面封装了如何延迟加载实现Feature接口的类类型。两个declare parent语句指定：实现Feature接口的任何类都满足RealComponent角色，并且LazyProxy可以代替Feature来履行其为任何Feature对象提供一个代理的角色。

initializeComponent(Object)方法返回一个LazyProxy实例，它是用必要的配置建立的，用以加载真实的组件。无论何时目标应用程序决定延迟加载特定的实例，都会调用initializeComponent(Object)方法。

抽象的DelegatingProxyPattern需要requestTriggered(..)切入点和delegateRequest(..)方法。requestTriggered(..)切入点用于捕获对实现Feature接口的类上方法的所有调用。当方面调用在LazyProxy上调用的方法时，将提供额外的保护来阻止LazyFeatureLoading方面通知它自身。

delegateRequest(..)方法会检查调用的主题，以查看它是否是LazyfeatureProxy类的一个实例。如果该主题是LazyFeatureProxy的一个实例，就会调用getRealComponent()方法来加载真实的组件。然后，将把方法调用转发给真实的组件。

最后，LazyFeatureLoading方面定义了LazyFeatureProxy类。这个类包含调用一个方法来加载相应的真实Feature实现所需的所有信息。通过LazyProxy 接口所需的getRealComponent()方法来执行真实Feature实现的加载。

例21-9说明了示例应用程序如何使用LazyFeatureLoading方面来支持FeatureA类和FeatureB类的延迟加载，这两个类都实现了Feature接口。

例21-9：在示例应用程序中使用LazyFeatureLoading方面

```
public class MainApplication
{
   private Feature[] features;

   public MainApplication()
   {
      features = new Feature[2];

      features[0] =
         LazyFeatureLoading
            .aspectOf()
            .initializeComponent(
            "com.oreilly.aspectjcookbook.features.FeatureA");
```

```
        features[1] =
           LazyFeatureLoading
              .aspectOf()
              .initializeComponent(
              "com.oreilly.aspectjcookbook.features.FeatureB");

        features[0].doSomething("Hello there");
        features[0].doSomething("Hello again");

        features[1].doSomething("Hi to you too");
        features[1].doSomething("Hi again");
    }

    public static void main(String[] args)
    {
        MainApplication mainApplication =
           new MainApplication();
    }
}
```

参考

4.1节描述了call(Signature)切入点；9.1节描述了within(TypePattern)切入点；12.2节和12.3节分别描述了AND（&&）运算符和OR（||）运算符；15.2节解释了使用继承来实现抽象方面；18.6节说明了使用方面来创建委托代理模式；23.2节解释了Director面向方面设计模式。

21.4 管理应用程序属性

问题

你想对应用程序类透明地管理应用程序配置属性。

解决方案

创建一个方面，用于加载、供应、监视和存储应用程序属性。

讨论

Java应用程序属性传统上是从单件类加载的，并由其管理。它的一个示例是System.getProperty()方法，它从命令行利用 *-D* 选项返回提供给应用程序的属性。

不幸的是，单件倾向于是一种脆弱的解决方案，它导致应用程序的许多区域依赖于对单件的接口。如果单件的接口发生变化，则应用程序的许多相关区域很可能不得不做出改变，以纳入新的接口。这是方面可以解决的一类横切关注点。

在针对属性管理的传统方法中，单件属性管理器是应用程序中的被动参与者，并且对来自应用程序多个部分的属性信息请求做出响应。属性管理器对将利用它所提供的属性信息做什么或者它会到达哪里一无所知，并且在更新任何属性时都依赖于通知。

利用针对系统属性的面向方面的方法，就变换了看问题的视角。AspectJ 提供的机制允许设计属性管理器，使得它可以主动把属性应用于那些需要它们的应用程序区域。关于将属性部署在何处的所有信息都包含在方面内，因此，如果需要新的属性，那么只有方面会发生变化。

属性管理方面将应用程序的余下部分与关于如何加载、存储和提供属性的任何考虑事项隔离开。不再需要把过分简单化的名/值形式的接口绑定到属性上，因为接口不再存在。方面会加载属性，在需要的地方应用它们，以及在应用程序关闭时把它们存储起来。

使用方面管理属性的最后一个优点是：由于属性管理方面很可能是一个特权方面，可以设置应用程序中对应于它所管理属性的变量，所以如果需要，它可以监视这些变量的任何变化，以将这些变化反映回它所管理的属性中。这意味着方面可以加载、供应和存储属性，并且可以监视属性的变化，因此在属性发生变化时，根本不需要应用程序通知属性管理器。

例 21-10 显示了用于简单示例程序的属性管理器方面。

例 21-10：使用方面加载、应用、监视和存储应用程序的属性

```
package com.oreilly.aspectjcookbook;

import java.util.Properties;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;

public privileged aspect MyApplicationProperties
{
   // Property names

   private static final String MYCLASS_PROPERTY_NAME =
      "com.oreilly.aspectjcookbook.MyClass.property";

   private static final String MAINAPPLICATION_PROPERTY_NAME =
      "com.oreilly.aspectjcookbook.MainApplication.property";

   // Default property values

   private static final int DEFAULT_MAINAPPLICATION_PROPERTY = 1;

   private static final String DEFAULT_MYCLASS_PROPERTY = "Property Initialized:";

   // Property Storage
```

```
private static final String PROPERTY_FILE_SYSTEM_PROPERTY = "props";

private static final String DEFAULT_PROPERTIES_FILENAME =
   "myapplication.properties";

Properties applicationProperties = new Properties();

File propertiesFile;

// Load Properties

public MyApplicationProperties()
{
   try
   {
      String propertyFilename = System.getProperty(PROPERTY_FILE_SYSTEM_PROPERTY);
      if (propertyFilename != null)
      {
         propertiesFile = new File(propertyFilename);
      }
      else
      {
         propertiesFile = new File(DEFAULT_PROPERTIES_FILENAME);
      }
      FileInputStream inputStream = new FileInputStream(propertiesFile);
      applicationProperties.load(inputStream);
      inputStream.close();
   }
   catch (Exception e)
   {
      // Just using default properties instead.
      System.err.println("Unable to load properties file, reverting to default
      values");
   }

}

// Supply Properties

public pointcut mainApplicationInitialization() :
   staticinitialization(MainApplication);

after() : mainApplicationInitialization()
{
   try
   {
      int mainApplicationProperty =
         new Integer(
            applicationProperties.getProperty(
               MAINAPPLICATION_PROPERTY_NAME)).intValue();

      MainApplication.property = mainApplicationProperty;
   }
```

```
    catch (Exception e)
    {
        MainApplication.property = DEFAULT_MAINAPPLICATION_PROPERTY;

        applicationProperties.setProperty(
            MAINAPPLICATION_PROPERTY_NAME,
            new Integer(DEFAULT_MAINAPPLICATION_PROPERTY).toString());
    }
}

public pointcut myClassObjectCreation(MyClass myObject) :
    execution(public MyClass.new(..)) &&
    this(myObject);

before(MyClass myObject) : myClassObjectCreation(myObject)
{
    String myClassProperty = applicationProperties.
        getProperty(MYCLASS_PROPERTY_NAME);
    if (myClassProperty != null)
    {
        myObject.property = myClassProperty;
    }
    else
    {
        myObject.property = DEFAULT_MYCLASS_PROPERTY;
        applicationProperties.setProperty(
            MYCLASS_PROPERTY_NAME, DEFAULT_MYCLASS_PROPERTY);
    }
}

// Monitoring properties

public pointcut monitorMainApplicationProperty(int newValue) :
    set(int MainApplication.property) &&
    args(newValue);

after(int newValue) : monitorMainApplicationProperty(newValue) &&
    !within(MyApplicationProperties)
{
    applicationProperties.setProperty(
        MAINAPPLICATION_PROPERTY_NAME,
        new Integer(newValue).toString());
}

public pointcut monitorMyClassProperty(String newValue) :
    set(String MyClass.property) &&
    args(newValue);

after(String newValue) : monitorMyClassProperty(newValue) &&
    !within(MyApplicationProperties)
{
    applicationProperties.setProperty(MYCLASS_PROPERTY_NAME, newValue);
}

// Store properties on application close
```

```
    class ShutdownMonitor implements Runnable
    {
        public ShutdownMonitor()
        {
            Thread shutdownThread = new Thread(this);
            Runtime.getRuntime().addShutdownHook(shutdownThread);
        }

        public void run()
        {
            try
            {
                FileOutputStream outputStream = new FileOutputStream(propertiesFile);

                applicationProperties.store(
                outputStream);
                outputStream.close();
            }
            catch (Exception e)
            {
                System.err.println(
                    "Unable to save properties file, will use default on next run");
            }
        }
    }

    private ShutdownMonitor shutdownMonitor = new ShutdownMonitor();
}
```

首先，需要把MyApplicationProperties方面声明为特权方面，因为它将影响应用程序类的内部私有区域，以对需要它们的区域应用和监视属性值。

然后，把两个应用程序属性的名称声明为常量。在这个示例中，将要管理两个属性，一个是com.oreilly.aspectjcookbook.MainApplication类中的int，另一个是com.oreilly.aspectjcookbook.MyClass类中的String。将定义这两个属性的一些默认值，以防在加载属性时发生任何问题。

从文件加载属性，并将其存储为方面内的java.util.Properties，就像java.util.File属性、propertiesFile和applicationProperties属性所指定的那样。AspectJ中的方面在包含Java应用程序的public void main(String[])方法入口点的类之前初始化，因此MyApplicationProperties构造函数可以在应用程序开始运行之前，从指定的文件加载应用程序属性，然后应用这些属性。

mainApplicationInitialization()和myClassObjectCreation()切入点会捕获何时初始化需要属性的类，从而在类的实例访问这些属性之前应用它们。monitorMainApplicationProperty()和monitorMyClassProperty()切入点然后观察需要它们的类中的属性，以检测何时任何类会改变属性的值。如果发生改变，就会

触发 after()通知，并更新属性的主值，因为这不是属性的传统工作方式。如果一个对象改变了属性值，将不会就此通知另一个对象。通过使用19.1节介绍的观察者模式，当属性值发生变化时，可以实现所有相关对象的通知。

最后，为了处理在关闭应用程序时如何存储应用程序属性的问题，创建了ShutdownMonitor类，并将其实例化成方面的一个属性。ShutdownMonitor类在其构造函数中把自身注册为一个关闭挂钩，随着应用程序关闭而调用其run()方法时，ShutdownMonitor会把应用程序属性存储回指定的文件。

参考

4.4节描述了execution(Signature)切入点；8.3节研究了set(Signature)切入点；7.5节描述了staticinitialization(TypePattern)切入点；9.1节描述了within(TypePattern)切入点；11.1节解释了this(TypePattern | Identifier)切入点；11.3节介绍了args([TypePatterns || Identifiers])切入点；12.2节和12.3节分别描述了AND（&&）运算符和OR（||）运算符；13.3节解释了before()形式的通知；13.5节解释了after()形式的通知；19.1节说明了如何使用方面来实现观察者面向对象设计模式。

第 22 章

应用企业级方面

22.0 简介

企业级应用程序通常不得不考虑更传统的桌面软件之上和之外的一些因素。这些考虑事项是面向方面方法的良好候选，因为它们极少与软件的核心业务逻辑紧密耦合。

诸如事务行为、分布式通信以及安全意义之类的考虑事项是任何严肃的企业系统的支柱。甚至像提供和强制执行开发指导之类的任务（它们通常是大型系统开发的顽疾）也表明可以从 AspectJ 为编译器提供建议的能力中获益。

由于企业系统中如此多的部分并不是应用程序的业务逻辑的核心，所以把这些系统看作是可以发挥面向方面所长的领域之一。企业系统是应用面向方面的最快速增长的领域，本章中的方面代表那些系统中的一小部分考虑事项，它们都可以从面向方面的方法中获益。

注意： 领导性的J2EE社区内的工作证明了面向方面技术在企业系统中的快速采用。特别是，开源 Spring Framework（*http://www.springframework.org* 和 JBoss（*http://www.jboss.org*）社区纳入了面向方面的相当成熟的实现，一些商业公司（比如著名的IBM）坚定地对这种方法提供支持，面向方面的工具可能不久就会变成企业开发人员工具包中常备的、重要的一部分。

22.1 应用开发指导和规则

问题

你想通过提供一种在编译时强制执行的策略，来控制在应用程序中允许哪些程序构造。

解决方案

使用Border Controller（边界控制器）面向方面设计模式声明代码内的一组区域。当依据策略模式在方面中为项目声明任何顶级规则时，重用这些区域。可以选择扩展项目的顶级策略，为应用程序的特定区域特殊化它们。

讨论

这个解决方案允许窥视第23章中将介绍的两个面向方面设计模式。Border Controller设计模式为这个秘诀的解决方案提供了基础，它允许捕获应用程序的体系结构，作为一组声明了代码内重要区域的可重用的切入点。可以在应用程序中的其余所有方面中引用这些区域。

例22-1中所示的BorderControllerAspect声明了示例应用程序内的4个区域：withinTestingRegion()区域纳入了存放测试代码的包，withinMyApp()指定了组成应用程序的包和子包，withinThirdParty()指定了可以使用第三方源代码的任何区域，withinMyAppMainMethod()则方便地声明了应用程序的main(..)方法的位置。

例22-1：通过声明代码内的重要区域，为项目和应用程序的策略提供基础

```
package com.oreilly.aspectjcookbook;

public aspect BorderControllerAspect
{
   /**
    * Specifies the testing region.
    */
   public pointcut withinTestingRegion() :
      within(com.oreilly.aspectjcookbook.testing.+);

   /**
    * Specifies My Applications region.
    */
   public pointcut withinMyApp() : within(com.oreilly.aspectjcookbook.myapp.+);

   /**
    * Specifies a third party source code region.
    */
   public pointcut withinThirdParty() :
      within(com.oreilly.aspectjcookbook.thirdpartylibrary.+);

   /**
    * Specifies the applications main method.
    */
   public pointcut withinMyAppMainMethod() :
      withincode(public void com.oreilly.aspectjcookbook.myapp.MyClass.main(..));
}
```

例22-1只显示了可以在应用程序中定义的其中一些区域。其他良好的候选有：将发生特殊日志记录的区域；受延迟加载逻辑支配的区域；以及你发现体系结构的正式定界部分是有用的任何区域，它们使得更多的切入点定义可以重用，并在那些边界内安全地工作。其思想是：如果这些边界发生变化，则只需更改Border Controller，使得应用程序其余的切入点逻辑将立即获得对其作用域的任何相关的改变。

Border Controller提供了一个有用的可重用切入点定义的库，它被纳入到策略面向方面设计模式中。这个模式通过为项目的区域声明不同的策略，可用于解决这个秘诀的问题，如例22-2所示。

例22-2：定义项目级策略

```
package com.oreilly.aspectjcookbook;

public abstract aspect ProjectPolicyAspect
{
   protected abstract pointcut allowedSystemOuts();

   declare warning :
      call(* *.println(..)) &&
      !allowedSystemOuts() &&
      !BorderControllerAspect.withinTestingRegion()
   : "System.out usage detected. Suggest using logging?";
}
```

例22-2中的ProjectPolicyAspect定义了项目级规则，它指示在编译时发出将输出到System.out流的消息。这个方面留下了一个抽象切入点，使得特殊化的子方面可以定义一些区域，在这些区域中，使用System.out是可接受的。例22-3中特殊化的MyAppPolicyAspect扩展了抽象的ProjectPolicyAspect，声明允许测试和thirdpartylibrary源目录使用System.out。

例22-3：为应用程序区域的细节特殊化项目级策略

```
package com.oreilly.aspectjcookbook.myapp;

import com.oreilly.aspectjcookbook.ProjectPolicyAspect;
import com.oreilly.aspectjcookbook.BorderControllerAspect;

public aspect MyAppPolicyAspect extends ProjectPolicyAspect
{
   /**
    * Specifies regions within the application where messages
    * to System.out are allowed.
    */
   protected pointcut allowedSystemOuts() :
      BorderControllerAspect.withinMyAppMainMethod() ||
      BorderControllerAspect.withinThirdParty() ||
      BorderControllerAspect.withinTestingRegion();
}
```

使用Policy和Border Controller面向方面模式，可以正式化体系结构的结构，并应用一致的项目、应用程序、包、类和方法作用域策略。

参考

4.1节描述了call(Signature)切入点；9.1节描述了within(TypePattern)切入点；9.3节描述了withincode(Signature)切入点；12.2节和12.3节分别描述了AND（&&）运算符和OR（||）运算符；12.4节说明了一元NOT（!）运算符；12.6节介绍了定义切入点定义的可重用库；16.6节说明了用新的警告和错误扩展编译器；23.3节解释了Border Controller面向方面设计模式；23.4节描述了策略面向方面设计模式。

22.2 应用事务

问题

你想将事务行为引入到应用程序中的方法中。

解决方案

使用Director（导向者）面向方面设计模式声明一个抽象方面，用于捕获控制事务的通用行为，如例22-4所示。扩展抽象的事务方面，为应用程序内的每个事务声明特殊化的子方面。

例22-4：在可重用的抽象方面中声明事务的通用行为

```
public abstract aspect TransactionAspect
{
   protected abstract pointcut transactionalCall();

   protected pointcut transactionBoundary() :
      transactionalCall() && !cflowbelow(transactionalCall());

   protected interface Transaction
   {
      public void commit();
      public void rollback();
   }

   protected Transaction transaction;

   before() : transactionBoundary()
   {
      setupTransaction(thisJoinPoint.getArgs());
   }
```

```
    after() returning: transactionBoundary()
    {
        transaction.commit();
    }

    after() throwing: transactionBoundary()
    {
        transaction.rollback();
    }

    protected abstract void setupTransaction(Object[] args);
}
```

讨论

事务用于把所选的操作组合进一个内聚单元中，这个单元要么全部完成，要么如果其执行过程中的某一步失败，则它会回复到其原始状态。

在例22-4中，TransactionAspect首先指定transactionCall()抽象切入点。特殊化的子方面使用这个切入点来指定将把目标应用程序内的方法视作事务性的。

transactionBoundary()切入点然后依靠transactionCall()切入点来指定事务开始和结束的位置。cflowbelow()切入点用于忽略可能出现在事务寿命内的任何连接点。

TransactionAspect一般需要存储事务并与之交互，因此定义了Transaction接口。Transaction接口为子方面提供了一个基础，用以实现它们自己的事务类。然后使用单个事务属性来指定正被方面管理的当前事务。

最后，在事务生命周期内的不同时刻，将有三份通知处理事务属性。before()通知调用setupTransaction(Object[])抽象方法，以便用合适的Transaction实现正确地初始化事务属性。如果transactionCall()切入点所选的连接点未引发异常地返回，将会执行after() returning通知；这是提交事务的良好时间。after() throwing通知用于连接点带异常地返回的情况，因此需要回滚事务。

例22-5说明了如何为示例应用程序特殊化TransactionAspect，在这个示例应用程序中，transactionalCall()切入点的实现会选择transfer(..)方法，用以标记事务的开始。

例22-5：实现一种事务性行为，用于在应用程序内执行两个银行账户间的转账

```
import com.oreilly.aspectjcookbook.Account;
import com.oreilly.aspectjcookbook.InsufficientFundsException;

public aspect TransferTransactionAspect extends TransactionAspect
{
```

```
  protected pointcut transactionalCall() :
     call(public void com.oreilly.aspectjcookbook.Bank.transfer(..));

  private class TransferTransaction extends ThreadLocal implements Transaction
  {
     private Account from;
     private Account to;
     private float value;

     public TransferTransaction(Account from, Account to, float value)
     {
        this.from = from;
        this.to = to;
        this.value = value;
     }

     public void commit()
     {
        System.out.println("Committing");
        // Nothing to actually commit here, all the changes have been accepted ok
     }

     public void rollback()
     {
        System.out.println("Rolling back");
        try
        {
           to.debit(value);
        }
        catch(InsufficientFundsException ife)
        {
           System.err.println("Could not complete rollback!");
           ife.printStackTrace();
        }
     }
  }

  protected void setupTransaction(Object[] args)
  {
     this.transaction =
        new TransferTransaction(
              (Account) args[0],
              (Account) args[1],
              ((Float)args[2]).floatValue());
  }
}
```

参考

本示例是从 Ron Bodkin 提供的演示文稿 New Aspects of Security 改编而成的，可以从 *http://newaspects.com/presentations/*上找到该演示文稿；William Crawford、Jim Farley 和 David Flanagan 所著的 *Java Enterprise in a Nutshell*（O'Reilly）介绍了企业级 Java

应用程序中的事务；4.1节介绍了call(Signature)切入点；10.2节描述了cflowbelow(Pointcut)切入点；13.3节介绍了before()形式的通知；13.6节介绍了after() returning形式的通知；13.7节介绍了after() throwing形式的通知；15.4节解释了定义抽象方面和切入点；23.3节讨论了Director面向方面设计模式。

22.3 应用资源池

问题

你想在不影响应用程序现有操作的情况下，通过创建可重用的常用资源池，来优化对那些资源的访问。

解决方案

使用Director面向方面设计模式，创建一个抽象方面，把Resource（资源）和Resource Pool（资源池）的角色定义为接口，以及定义资源池可重用的通用行为，如例22-6所示。然后可以为应用程序内要存入池中的每种资源特殊化抽象资源池方面。

例22-6：在抽象方面中定义通用资源池行为

```
public abstract aspect ResourcePoolingAspect
{
   public interface Resource
   {

   }

   public interface ResourcePool
   {
      public void add(Resource resource);
      public Resource remove();
   }

   protected class ResourcePoolsCollection
   {
      WeakHashMap pools = new WeakHashMap();

      public void putResourcePool(ResourcePool pool, Class resourceClass)
      {
         pools.put(resourceClass, pool);
      }

      public ResourcePool getResourcePool(Class resourceClass)
      {
         return (ResourcePool) pools.get(resourceClass);
      }
   }
```

```
    protected ResourcePoolsCollection resourcePools = new ResourcePoolsCollection();

    public ResourcePoolingAspect()
    {
       initializeSpecificPool();
    }

    protected abstract void initializeSpecificPool();

    private pointcut excludeAspects() : !within(ResourcePoolingAspect+);

    public abstract pointcut catchResourceConstruction();

    public abstract pointcut catchResourceDestruction(Resource resource);

    Object around() : catchResourceConstruction() && excludeAspects()
    {
       ResourcePool resources =
          resourcePools.getResourcePool(
             thisJoinPoint.getSignature().getDeclaringType());
       return resources.remove();
    }

    Object around(Resource resource) :
       catchResourceDestruction(resource) && excludeAspects()
    {
       ResourcePool resources =
          resourcePools.getResourcePool(
             thisJoinPoint.getSignature().getDeclaringType());
       Object returnValue = resourceReturnedToPool(resource);
       System.out.println("Resource added back into pool: " + resource);
       resources.add(resource);
       return returnValue;
    }

    protected abstract Object resourceReturnedToPool(Resource resource);

    // A resource must use and resort to a simple default constructor for initialization
    // As protected by the warning declared below
    declare warning : call(public Resource+.new(*,..))
     : "Use a default constructor when using classes declared as pooled resources";
}
```

讨论

例22-6说明了如何定义一个可重用的抽象方面，它将在最初不支持资源池的应用程序内提供一个资源池。声明Resource和ResourcePool接口，以便可以对这些接口定义通用的资源池行为，并把它们与这些接口可能的实现方式隔离开。

可以跨所有声明为子方面的资源池共享在抽象的ResourcePoolingAspect内声明的行为。ResourcePoolCollection类为整个应用程序中的所有资源池提供了一个公共库，使得通用代码可以依据特定资源池所包含资源的类，来查寻该资源池。

在初始化ResourcePoolingAspect时，将会发出一个对initializeSpecificPool()抽象方法的调用。这个方法由ResourcePoolingAspect的子方面实现用于初始化它们自己的池实现，并把它们添加到ResourcePoolCollection中。

该应用程序提供了catchResourceConstruction()和catchResourceDestruction(Resource)抽象切入点，使得子方面可以指定第一次访问和释放某种资源的连接点。然后把这些切入点应用于两组around()通知，它们会在创建或销毁资源时重写，使得可以代之以使用相应资源池中存在的资源。在释放某种资源并将其放回到相应池中时，将会调用resourceReturnedToPool()抽象方法，从而给子方面提供一个机会，来执行任何适当的后处理。

例22-7说明了如何将抽象的ResourcePoolingAspect应用于示例应用程序。一旦应用了这个方面，就会得到BusinessResource类的对象，并将其释放回资源池中，这个操作对应用程序的其余部分是透明的。

例22-7：将资源池应用于简单示例应用程序内的资源

```
import java.util.List;
import java.util.ArrayList;

import com.oreilly.aspectjcookbook.BusinessResource;

public aspect BusinessResourcePoolingAspect extends ResourcePoolingAspect
{
   declare parents : BusinessResource implements Resource;

   public pointcut catchResourceConstruction() : call(public BusinessResource.new());

   public pointcut catchResourceDestruction(Resource resource) :
      call(public void BusinessResource.close()) && target(resource);

   private class BusinessResourcePool implements ResourcePool
   {
      private static final int RESOURCE_POOL_SIZE = 10;

      List resources = new ArrayList();

      public BusinessResourcePool()
      {
         for (int x = 0; x < RESOURCE_POOL_SIZE; x++)
         {
            this.add(new BusinessResource());
         }
      }

      public synchronized void add(Resource resource)
      {
         resources.add(resource);
      }
```

```
    public synchronized Resource remove()
    {
        if (resources.size() == 0)
        {
            resources.add(new BusinessResource());
        }
        return (Resource) resources.remove(resources.size() - 1);
    }
}

protected void initializeSpecificPool()
{
    try
    {
        this.resourcePools.putResourcePool(new BusinessResourcePool(),
                Class.forName("com.oreilly.aspectjcookbook.BusinessResource"));
    }
    catch (ClassNotFoundException cnfe)
    {
        System.err.println("Couldn't find resource class to pool");
    }
}

protected Object resourceReturnedToPool(Resource resource)
{
    // Do any resource specific tudying up if necessary
    // None to do in this example
    return null;
}
}
```

BusinessResourcePoolingAspect将Resource角色应用于示例应用程序的BusinessResource类。catchResourceConstruction()和catchResourceDestruction(Resource)切入点被实现用于指定何时构造BusinessResource，以及何时通过调用其close()方法来释放它。

ResourcePool角色由BusinessResourcePool类实现，它包含一个BusinessResource对象列表。如果应用程序需要它，就可以依赖于情况和需要，以更高效的方式实现这个资源池。

最后，initializeSpecificPool()方法被实现用于构造BusinessResourcePool的单个实例。这个BusinessResourcePool将被添加到ResourcePoolCollection中，它由父ResourcePoolingAspect管理。在把BusinessResource对象释放回池中时，不需要做任何特殊的事情，因此，在这个示例中，resourceReturnedToPool(Resource)方法不会做任何额外的事情。

参考

4.1节描述了call(Signature)切入点；7.1节和20.1节研究了使用call(Signature)切入点来捕获和重写对构造函数的调用；4.3节描述了使用target()切入点；13.4节介绍了around()形式的通知；15.4节解释了定义抽象方面和切入点；23.3节解释了Director面向方面设计模式。

22.4 使用RMI透明地远程访问类

问题

你想使用Java远程方法调用（Remote Method Invocation，RMI）为另一台计算机上的一个实例远程调用一个本地对象，而不会更改原始本地对象的使用方式。

解决方案

创建一个RMI服务器应用程序，它包含类的一个实例，如例22-8所示。

例22-8：声明BusinessClass的一个实例，它包含在一个RMI服务器实现中

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ThisOrThatServerImpl extends UnicastRemoteObject
   implements ThisOrThatServer
{
   BusinessClass businessClass = new BusinessClass();

   public ThisOrThatServerImpl() throws RemoteException
   {

   }

   public void foo() throws RemoteException
   {
      this.businessClass.foo();
   }
}
```

在客户应用程序内创建一个方面，它用于截获对远程类的特定实例的调用，并把这些调用路由到相应的RMI服务器，如例22-9所示。

例22-9：截获对本地实例的调用，并把这些调用导向远程管理的实例

```
import java.rmi.*;

public aspect RemoteBusinessClassAspect
{
```

```
    public pointcut callBusinessClassFooInMain() :
        call(public void BusinessClass.foo()) &&
        withincode(public void MainApplication.main(String[]));

    void around() : callBusinessClassFooInMain()
    {
        try
        {
            ThisOrThatServer rmtServer =
                (ThisOrThatServer) Naming.lookup("rmi://localhost/TTServer");
            rmtServer.foo();
        }
        catch (Exception e)
        {
            System.err.println("Problems occured when attempting " +
            "to use remote object, default to local");
            proceed();
        }
    }
}
```

讨论

使用Java RMI，可以依据必须纳入的新接口、新异常和新代码，来干预应用程序的类。传统上，面向对象的外观设计模式可用于隐藏这些复杂性，但是通过使用方面，将能够使用RMI对现有代码透明地远程访问应用程序的区域，并且无需增加复杂的外观。

例22-9中的RemoteBusinessClass方面捕获在执行应用程序的main(..)方法的过程中何时在BusinessClass实例上调用foo()方法。相应的around()通知会重写那个调用，并尝试把消息传递给BusinessClass的远程实例，它被RMI服务器管理，并在这个类中被标识为TTServer。如果在尝试使用远程对象时发生任何异常，那么这个方面就会通过调用proceed()，从而恢复使用原来的本地实例。

注意： 需要知道RMI的一些特性，以使用完成这个秘诀的示例代码，可以在*http://www.aspectjcookbook.com*上找到它。与示例代码一起提供的*readme.txt*文件包含关于如何使RMI和示例正确工作的有用信息。

参考

William Crawford、Jim Farley和David Flanagan所著的*Java Enterprise in a Nutshell*（O'Reilly）提供了这个示例的基础；4.1节描述了call(Signature)切入点；9.3节描述了withincode(Signature)切入点；13.4节介绍了around()形式的通知。

22.5 应用安全策略

问题

你想在现有的应用程序区域中添加一些安全考虑事项。

解决方案

定义一个方面，在需要接受额外的安全策略支配的应用程序内指定一些连接点。在遇到这些连接点时，存储和验证这个方面内特定的安全实现，如例 22-10 所示。

例 22-10：把验证应用于特定类上的调用

```
public aspect SecureClassAAspect
{
   private boolean authenticated;

   public pointcut secureClassAMethods() :
      call(* com.oreilly.aspectjcookbook.ClassA.*(..));

   Object around() : secureClassAMethods()
   {
      if (authenticated)
      {
         return proceed();
      }
      else
      {
         LoginScreen loginScreen = new LoginScreen();
         loginScreen.setVisible(true);

         // Use the authentication procedure of your choice here
         // In this simple example we are just going to check that
         // it is the one person we know of
         if ((loginScreen.getUsername().equals("Kim")) &&
               (new String(loginScreen.getPassword()).equals("password")))
         {
            authenticated = true;
            loginScreen.dispose();
            return proceed();
         }
         loginScreen.dispose();
         return null;
      }
   }
}
```

讨论

安全是横切关注点的一个理想示例。安全特征极少涉及应用程序的简单业务逻辑。它们通常是插入的，并且不幸的是，它们经常是对软件所做的最后一件事。

在例22-10中，方面捕获何时在ClassA类上调用任何方法，并且检查当前用户是否经过了验证以运行这些方法。在这个示例中，验证的形式是：显示一个登录对话框，将用户名和密码与内部存储的一些常量作比较。一旦识别了用户，就会设置方面的验证属性，以便记住：当应用程序调用ClassA的实例上的其他方法时，不必执行将来的检查。

注意：例22-10使用简单的验证机制，使这个秘诀专注于应用安全策略的面向方面的特征，而不是把注意力放在特定验证技术的细节上。在实际中，可以考虑使用更正式的验证过程，如JAAS。

利用方面来应用安全策略，可以把安全代码模块化在一个位置，在极大程度上透明地应用安全策略，以及把安全策略应用于初始设计未考虑它的应用程序。

参考

Scott Oaks所著的*Java Security*(O'Reilly)更详细地研究了在应用程序中如何使用JAAS以及Java的其他安全特性；4.1节描述了call(Signature)切入点；13.4节介绍了around()形式的通知。

第23章

应用面向方面设计模式

23.0 简介

在第17~19章中已经看到了AspectJ如何增强现有的面向对象设计模式。本书最后一章通过重点介绍一些新的面向方面设计模式以及如何使用AspectJ实现这些模式，概括了AspectJ软件开发中的一些新兴的最佳实践。

面向方面设计模式是公共的、最佳实践的面向方面解决方案，它们针对的是常见的面向方面设计问题。AspectJ术语是常用的AspectJ语言语法和工具用法。尽管面向方面是一门相当新的学科，并且AspectJ甚至是这门学科的更年轻的实现，活跃的AspectJ社区仍然发现了一些设计模式和术语。

随着采纳面向方面方法的人数继续飞速上涨，新的最佳实践将会被发现，并被捕捉为设计模式。这些发现可能及时地导致一个丰富、有用的模式集合，就像面向对象开发社区所享受的那些模式一样。

注意： 有关AspectJ语言和工具使用术语的更多信息，请查看Ron Bodkin在aTrack项目上的工作，它具有丰富的可重用方面和切入点库，可以在http*://atrack.dev.java.net*上找到它。查看Mik Kersten的标准切入点术语，可以通过到达*http://www.eclipse.org/aspectj*并单击Documentation → standard pointcut idioms来找到它。这两位专家都提供了关于AspectJ语言和工具术语的一些先驱性的工作。

23.1 应用 Cuckoo's Egg 设计模式

问题

你想重写在构造函数调用上实例化的对象类型，以对原始业务逻辑透明地返回一个不同类的对象。

解决方案

应用 Cuckoo's Egg 面向方面设计模式。图 23-1 显示了 Cuckoo's Egg 模式的关键组件。

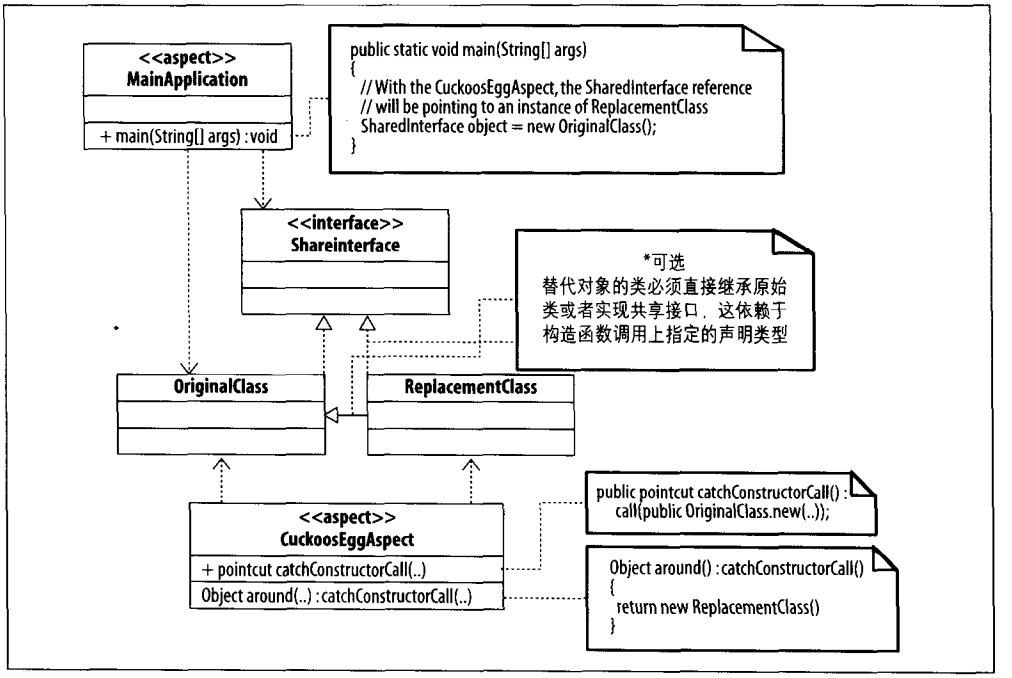

图 23-1：Cuckoo's Egg 模式的结构

图 23-1 中所示的 Cuckoo's Egg 模式中的关键角色如下：

`CuckoosEggAspect`

位于设计模式中心的方面，它会截获 `OriginalClass` 类的创建，并代之以返回 `ReplacementClass` 的一个实例。

`SharedInterface`

该模式的一个可选组件。依据AspectJ针对重写around()的规则，`ReplacementClass`

必须继承自构造函数调用上所期待的类型。在这个示例中，这个类型是 SharedInstance 接口。

OriginalClass

当 CuckoosEggAspect 重写构造函数调用时，最初会构造这个类。

ReplacementClass

当重写对 OriginalClass 的构造函数调用时，会实例化这个类。

MainApplication

表示应用程序内的一个示例区域，其中会重写对OriginalClass的构造函数调用。

讨论

与 Cuckoo's Egg 模式一样简单，它是面向方面在 AspectJ 中实现的更强大的能力之一。一种相当常见的情况是：在应用方面时，希望控制和改变在构造函数调用或工厂方法上实例化的对象。Cuckoo's Egg 设计模式正式化了这种常见的 AspectJ 用例。

例 23-1 是修改 20.2 节中的示例而得的，它说明了如何在 AspectJ 中实现 Cuckoo's Egg 模式，以便在调用 MyClass 构造函数时返回 AnotherClass 的一个实例。

例 23-1：Cuckoo's Egg 设计模式的一个示例

```
public aspect ControlClassSelectionAspect // Cuckoo's Egg Aspect
{
   public pointcut myClassConstructor() : call(MyClass.new());

   Object around() : myClassConstructor()
   {
      return new AnotherClass();
   }
}
```

Cuckoo's Egg 模式的关键特征如下：

- 它提供了合适的切入点声明，用于捕获要重写的类的构造。
- 如果合适的话，可以通过使用 args([Types | Identifiers])切入点把标识符传递给相应的通知，来获得原始构造函数调用上提供的任何参数。

在执行以下任务时，Cuckoo's Egg 模式是有用的：

- 实现模拟对象以进行测试
- 提供代理对象来代替类的真实实例

可以协同使用Cuckoo's Egg模式与Border Controller设计模式，用于限制对被Cuckoo's Egg 模式重写的特定类进行构造函数调用。

参考

http://www.onjava.com/pub/a/onjava/2004/10/20springaop2.html 上的文章说明了如何使用 Spring Framework 实现 Cuckoo's Egg 模式；7.1 节和 20.1 节描述了使用 `call(Signature)`切入点来捕获和重写对构造函数的调用；13.4节介绍了`around()`形式的通知；20.2节更详细地介绍了关于如何应用重写的`around()`通知的AspectJ规则。

23.2 应用 Director 设计模式

问题

你想定义一组角色，它们将由一组未知的应用程序类实现，使得一般可以通过一个抽象方面来与它们交互。

解决方案

应用 Director 面向方面设计模式。图 23-2 显示了 Director 模式的关键组件。

图 23-2 中所示的 Director 模式中的关键角色如下：

`DirectorAspect`

位于设计模式中心的抽象方面，它使用 Java 接口指定要指引的角色，并且可以选择指定在这些角色之间可以发生的交互。

`Role1`和`Role2`

定义为抽象 `DirectorAspect` 内的嵌套接口的一对示例角色。

`BusinessClassA`和`BusinessClassB`

目标应用程序内的一对业务类，它们是通过`DirectorAspect`定义的角色的候选。

`SpecializedAspect`

将角色应用于目标应用程序内的特定类。在这个示例中，将`Role1`应用于`BusinessClassA`，将`Role2`应用于`BusinessClassB`。当`BusinessClassB`没有实现`Role2`接口所需的`bar()`方法时，可以通过`SpecializedAspect`使用静态横切技术来应用它。

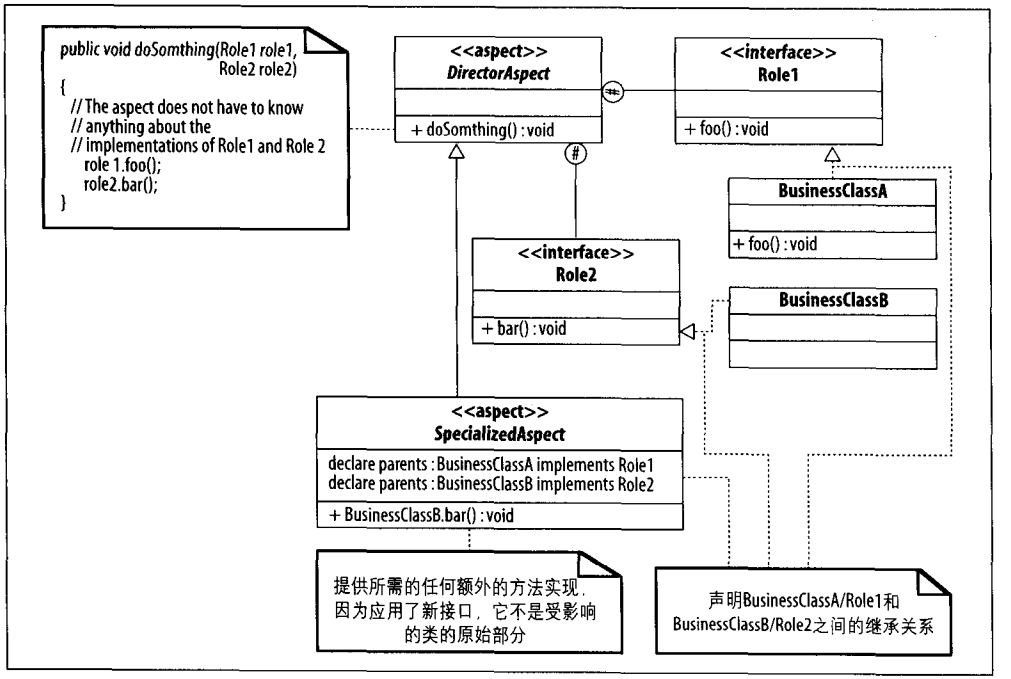

图 23-2：Director 模式的结构

讨论

Director 模式把通用的、可重用的方面行为与特定应用程序的实现类分离开，从而允许方面的逻辑指引抽象的角色，而不是实现。

例23-2是从19.1节中的示例修改而成的，它说明了如何在AspectJ中实现Director模式，以指定 Subject（主题）和Observer（观察者）角色，以及 Subject 角色的一些默认的实现细节。

例 23-2：Director 模式的示例

```
public abstract aspect ObserverPattern
{
   protected interface Subject
   {
      public void addObserver(Observer observer);
      public void removeObserver(Observer observer);
   }
```

```
   protected interface Observer
   {
      public void notifyOfChange(Subject subject);
   }

   private List Subject.observers = new LinkedList();

   public void Subject.addObserver(Observer observer)
   {
      this.observers.add(observer);
   }

   public void Subject.removeObserver(Observer observer)
   {
      this.observers.remove(observer);
   }

   private synchronized void Subject.notifyObservers()
   {
      Iterator iter = this.observers.iterator();
      while (iter.hasNext())
      {
         ((Observer)iter.next()).notifyOfChange(this);
      }
   }

   protected abstract pointcut subjectChange(Subject s);

   after(Subject subject) : subjectChange(subject)
   {
      subject.notifyObservers();
   }
}
```

例23-3（从19.1节中的示例修改而成）说明了ObserverPattern抽象方面如何使用Director模式，将Observer和Subject角色应用于ConcreteClassA和ConcreteClassB目标应用程序类。

例23-3：将Director模式角色应用于目标应用程序的类

```
public aspect ConcreteClassAObserver extends ObserverPattern
{
   declare parents : ConcreteClassB implements Subject;

   declare parents : ConcreteClassA implements Observer;

   protected pointcut subjectChange(Subject s) :
      call(* ConcreteClassB.set*(..))
      && target(s);

   public void ConcreteClassA.notifyOfChange(Subject subject)
   {
      this.doSomething(
         "ConcreteClassA was notified of a change on " + subject);
```

```
    }
}
```

下面给出了 Director 模式的关键特征：

- 提供了一种机制，用于声明可重用的方面逻辑（包括通知），以及它如何与一组抽象角色交互，而不必知道哪些目标实现类将与这些角色相耦合。
- 支持把角色以及在这些角色上工作的逻辑封装在一个位置：即 Director 方面。
- 允许声明可应用于目标应用程序类的整个家族的逻辑。

在以下情况下，Director 模式是有用的：

- 在不知道将要把方面的逻辑应用于哪些目标应用程序类的情况下，定义一个抽象方面。
- 在应用程序中的抽象实体之间实现一组关系，如面向对象设计模式的实现。

注意： 由于AspectJ 1.1的发布，依据Director模式将抽象角色应用为接口是引入跨多个类的行为的唯一方式，在 AspectJ 文档中解释了这一点，可以在 *http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectjhome/doc/README-11.html#SINGLE_INTERCLASS_TARGET* 上找到该文档。

参考

第 15 章介绍了使用抽象方面和切入点，以及声明方面之间的继承关系；在第 17~19 章中，当使用面向方面的技术实现传统的面向对象设计模式时，主要使用的是Director模式。

23.3 应用 Border Control 设计模式

问题

你想正式定义应用程序内的重要区域，使得方面可以重用这些定义，以确保只在正确的区域应用它们。

解决方案

应用Border Control面向方面设计模式。图23-3显示了Border Control模式的关键组件。

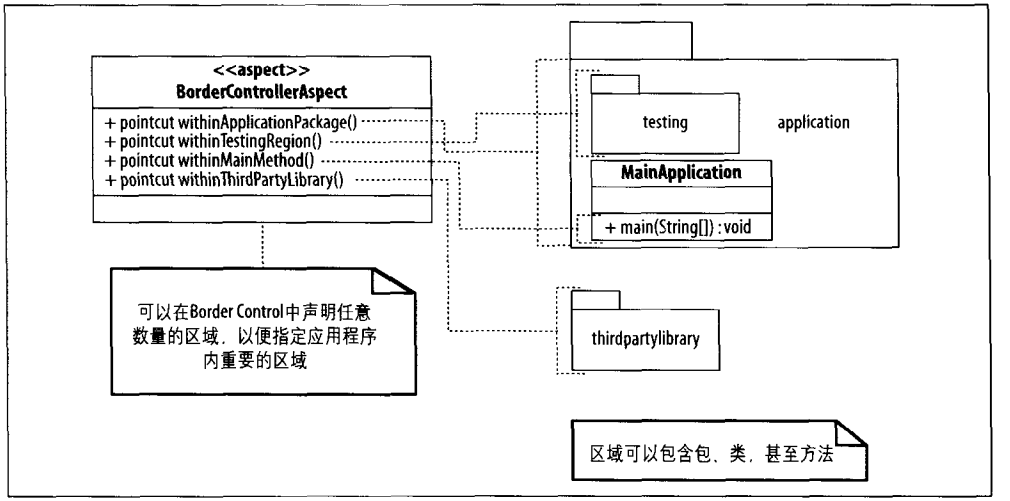

图 23-3：Border Control 模式的结构

图 23-3 中所示的 Border Control 模式中的关键角色如下：

`BorderControllerAspect`

位于设计模式中心的方面，指定了应用程序中任何重要的区域，其他方面可以使用这个方面来限制它们对合适区域的影响。

`application`、`application.testing` 和 `thirdpartylibrary` 包

它们是示例目标应用程序的重要包，通过 `BorderControllerAspect` 的可重用切入点声明来选择它们。

`MainApplication`

这个类的 `main(String[])` 方法提供了类中的重要区域的一个示例。

讨论

Border Controller 声明了代码的重要特征区域，它们可以是作用域中的方法、类和包。例 23-4 是从 22.1 节中的示例修改而成的，它说明了如何在 AspectJ 中实现 Border Controller 模式，以指定一组包含示例应用程序中的包和方法的区域。

例 23-4：Border Controller 模式的示例

```
public aspect BorderControllerAspect
{
    /**
     * Specifies the testing region.
     */
```

```
    public pointcut withinTestingRegion() :
       within(com.oreilly.aspectjcookbook.testing.+);

    /**
     * Specifies My Applications region.
     */
    public pointcut withinMyApp() : within(com.oreilly.aspectjcookbook.myapp.+);

    /**
     * Specifies a third party source code region.
     */
    public pointcut withinThirdParty() :
       within(com.oreilly.aspectjcookbook.thirdpartylibrary.+);

    /**
     * Specifies the applications main method.
     */
    public pointcut withinMyAppMainMethod() :
       withincode(public void com.oreilly.aspectjcookbook.myapp.MyClass.main(..));
}
```

在例 23-4 中，之后可以在目标应用程序的方面中重用在 Border Controller 方面中声明的区域：

```
pointcut regionsOfInterest() :
     BorderControllerAspect.withinMyAppMainMethod() ||
     BorderControllerAspect.withinThirdParty() ||
     BorderControllerAspect.withinTestingRegion();
```

下面给出了 Border Controller 模式的关键特征：

- 它提供了一种机制，用于声明可重用的切入点逻辑，以正式化应用程序的体系结构。
- 每当某个方面只工作在一个或多个特定的应用程序区域内时，都可以在整个应用程序的方面中重用 Border Controller 方面内声明的区域。
- 如果应用程序的结构发生变化，并且应用了 Border Controller 模式，那么很可能只需要更新 Border Controller 方面，以在应用程序中的所有方面中反映这些变化。
- Border Controller 通常最好作为单件应用，这是由于一个 Border Controller 通常足以满足单个应用程序。

在执行以下任务时，Border Controller 模式是有用的：

- 假定有一个相当复杂的应用程序，在这个应用程序中定义了一些内部区域，并且必须把方面限制于这些区域。
- 防止应用程序的方面在将来更改应用程序的总体结构。为了避免在应用程序的结构

发生变化时更新所有方面的麻烦，Border Controller模式提供了可以执行那些更改的单一点，并且自动在应用程序的所有方面中反映所做的更改。

在使用大多数设计模式和应用程序（包括Cuckoo's Egg和Policy（策略）面向方面设计模式）时，都可以把Border Controller设计模式用作定义应用程序结构的基础。

参考

第9章研究了基于切入点的程序作用域；22.1节使用了Border Controller方面；23.1节描述了Cuckoo's Egg面向方面设计模式；23.4节描述了策略面向方面设计模式。

23.4 应用策略设计模式

问题

你想在策略内定义一组开发规则，可将其应用于应用程序的结构。

解决方案

应用策略面向方面设计模式。图23-4显示了策略模式的关键组件。

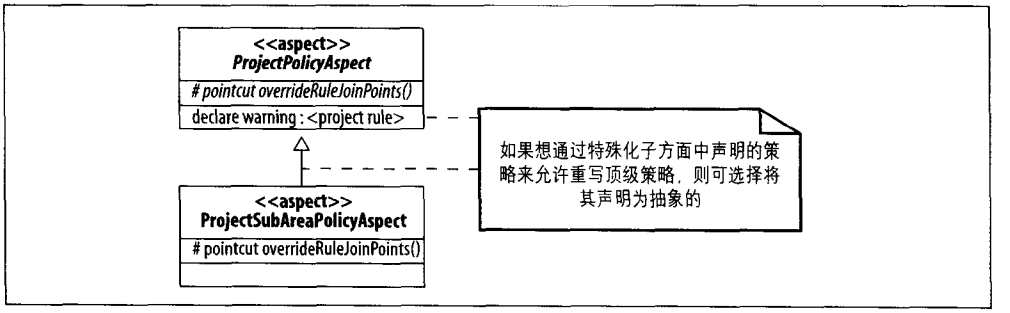

图23-4：策略模式的结构

图23-4所示的策略模式中的关键角色如下：

`ProjectPolicyAspect`

为应用程序指定项目级或顶级策略的方面。如果要通过特殊化的子方面扩展它，则可选择将其声明为抽象方面。

ProjectSubAreaPolicyAspect

在这个示例中，将ProjectPolicyAspect声明为抽象的，从而留下一个抽象切入点，它使这个子方面能够依据特定项目子区域的策略的细节来重写顶级规则。

讨论

策略模式可以为应用程序内的任何区域声明一组规则。这些规则涵盖广泛的范围，从编译器错误和警告到重写某些类和库的使用。

例23-5是从22.1节中的示例修改而成的，它说明了如何在AspectJ中实现策略模式，以为示例应用程序指定一种顶级策略，它包含一条规则，指出如果调用Java的System.out.println(..)方法，则在编译时发出一个警告。

例23-5：把顶级规则应用于应用程序的策略模式的示例

```
public abstract aspect ProjectPolicyAspect
{
   protected abstract pointcut allowedSystemOuts();

   declare warning :
      call(* *.println(..)) &&
      !allowedSystemOuts() &&
      !BorderControllerAspect.withinTestingRegion()
   : "System.out usage detected. Suggest using logging?";
}
```

在这个示例中，由于顶级策略方面是抽象的，并且声明了一个可用于重写System.out.println(..)规则的抽象切入点，这样就可以为目标应用程序内的不同区域特殊化它，如例23-6所示。

例23-6：扩展抽象的顶级策略

```
public aspect MyAppPolicyAspect extends ProjectPolicyAspect
{
   /**
    * Specifies regions within the application where messages
    * to System.out are allowed.
    */
   protected pointcut allowedSystemOuts() :
      BorderControllerAspect.withinMyAppMainMethod() ||
      BorderControllerAspect.withinThirdParty() ||
      BorderControllerAspect.withinTestingRegion();
}
```

下面给出了策略模式的关键特征：

- 为可以在编译时和运行时应用的规则提供一种机制。
- 可用于为应用程序的不同区域声明复杂规则的层次结构。

在下列情况下，策略模式是有用的：

- 在开发有许多开发人员参与的应用程序时（比如在一个开源项目中），并且希望更主动地传达一些开发规则和指导，而不仅仅是提供文档。
- 可以更改策略，以便于把应用程序从一组库和API迁移到另一组库和API。首先，可以在策略中对使用一组库发出警告；当绝对禁止使用库时，可将其上升为一个错误。最后，如果由于任何原因仍在使用库，则可应用一个代理，把代码从被禁止的库中移到策略首选的库中。

可以协同使用策略模式与以下设计模式：

- 它可以使用Cuckoo's Egg设计模式，重写特定类或库的使用。
- 它可以使用Proxy（代理）向面对象设计模式，截获对特定类或库的调用，并且依据应用程序策略，拒绝这些调用或者将其委托给适当的库或类。

你想为基于Border Controller设计模式的应用程序的不同区域定义特殊化的策略。

参考

16.6节说明了扩展编译器以包括新的警告和错误；18.6节说明了如何使用面向方面的技术来实现代理面向对象模式；23.1节解释了Cuckoo's Egg面向方面设计模式；23.3节解释了Border Controller面向方面设计模式。

附录

AspectJ 运行时 API

访问关于在运行时触发方面中的通知的连接点信息有时是有用的，AspectJ 在 org.aspectj.* 包中提供了运行时 API，它正好用于这个目的。图 A-1 显示了 AspectJ 运行时 API 的包细目分类。

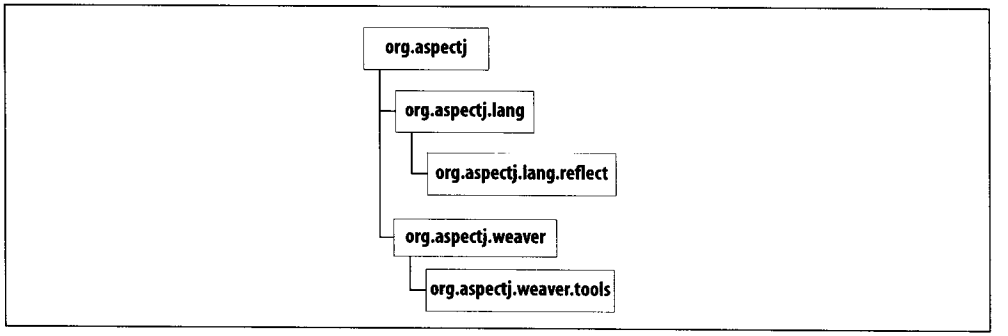

图 A-1：AspectJ 运行时 API 的包结构

AspectJ 运行时 API 的主要包包括：

org.aspectj.lang

提供了接口和支持类，用于发现关于连接点的运行时信息。

org.aspectj.lang.reflect

包含一些接口，它们提供了关于连接点签名的更特殊化的信息。

org.aspectj.weaver.tools

提供了一些接口，用于支持 JVM 类加载器执行加载时织入。

本附录重点介绍org.aspectj.lang和org.aspectj.lang.reflect的内容，并且给出了这些包中每个主要组件的简短概述。如果想深入研究 API，可以在 AspectJ 安装目录内的 *%ASPECTJ_INSTALLATION_DIR%/doc/api/index.html* 上找到详尽的文档。

org.aspectj.lang

与熟悉的 java.lang 包类似，它包含 Java 语言的最基本的构造；org.aspectj.lang 包提供了一组类，用于在最基本、最通用的级别上与连接点交互。

JoinPoint 和 JoinPoint.StaticPart

图 A-2 显示了 JoinPoint 和 JoinPoint.StaticPart 接口的结构。

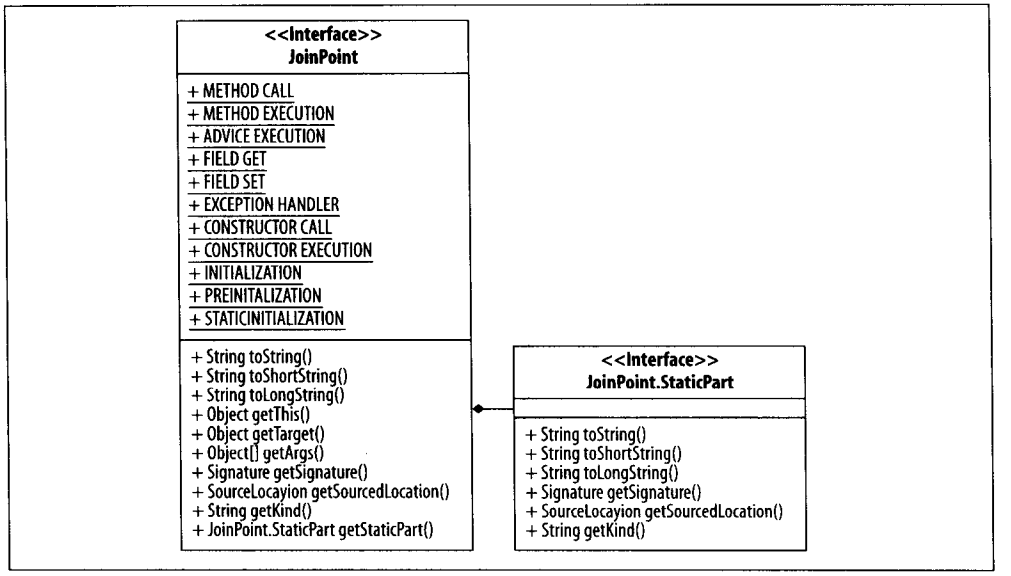

图 A-2：JoinPoint 和 JoinPoint.StaticPart 接口

JoinPoint 接口

JoinPoint 接口最常由 thisJoinPoint 引用使用，它可供任何通知块使用。该接口允许访问关于特定连接点的动态和静态信息。

public String toString()

重写对象上的toString()默认方法，返回连接点的字符串表示。从方法调用连接点上的toString()方法返回字符串的一个示例如下：

```
call(void com.oreilly.aspectjcookbook.MyClass.foo(int, String))
```

public String toShortString()

提供toString()方法的一个替代方法，返回连接点的字符串表示的一个简化版本。从方法调用连接点上的toShortString()方法返回字符串的一个示例如下：

```
call(MyClass.foo(..))
```

public String toLongString()

提供toString()方法的一个替代方法，返回连接点的字符串表示的一个扩展版本。从方法调用连接点上的toLongString()方法返回字符串的一个示例如下：

```
call(public void com.oreilly.aspectjcookbook.MyClass.foo(int, java.lang.String))
```

public Object getThis()

返回连接点上正在执行的对象（如果有的话）——即this引用的值。如果相应的连接点上没有引用，则返回null，与静态代码块的情况一样。

public Object getTarget()

返回作为连接点目标的对象（如果有的话）。例如，方法调用连接点的目标是正在调用的对象。如果相应的连接点上没有目标对象，则返回null。

public Object[] getArgs()

返回包含连接点上可用参数（如果有的话）的数组。例如，getArgs()方法返回在方法调用连接点上传递的参数。

public Signature getSignature()

返回连接点上的签名。

注意：JoinPoint.StaticPart.getSignature()方法返回与这个方法相同的值。

public SourceLocation getSourceLocation()

返回包含关于连接点源位置的所有可用信息（如果有的话）的对象。如果没有可用的源位置信息，则返回null。

注意：JoinPoint.StaticPart.getSourceLocation()方法返回与这个方法相同的值。

public String getKind()

返回一个指示连接点类型的字符串枚举。JoinPoint接口声明了以下常量，可将其与从该方法返回的字符串做比较，以确定连接点的类型：

METHOD_CALL

指定在方法调用上出现的连接点。参见4.1节，了解专用于捕获这类连接点的切入点声明。

METHOD_EXECUTION

指定在方法执行期间出现的连接点。参见4.2节，了解专用于捕获这类连接点的切入点声明。

ADVICE_EXECUTION

指定在通知执行期间出现的连接点。参见4.3节，了解专用于捕获这类连接点的切入点声明。

FIELD_GET

指定在访问属性时出现的连接点。参见4.4节，了解专用于捕获这类连接点的切入点声明。

FIELD_SET

指定在修改属性时出现的连接点。参见4.5节，了解专用于捕获这类连接点的切入点声明。

EXCEPTION_HANDLER

指定在处理异常时出现的连接点。参见4.6节，了解专用于捕获这类连接点的切入点声明。

CONSTRUCTOR_CALL

指定在构造函数调用上出现的连接点。参见4.7节，了解专用于捕获这类连接点的切入点声明。

CONSTRUCTOR_EXECUTION

指定在构造函数执行期间出现的连接点。参见4.8节，了解专用于捕获这类连接点的切入点声明。

INITIALIZATION

指定在对象初始化期间出现的连接点。参见4.9节，了解专用于捕获这类连接点的切入点声明。

PREINITIALIZATION

指定在对象初始化之前出现的连接点。参见4.10节，了解专用于捕获这类连接点的切入点声明。

STATICINITIALIZATION

指定在类初始化期间出现的连接点。参见4.11节，了解专用于捕获这类连接点的切入点声明。

注意：JoinPoint.StaticPart.getKind()方法返回与这个方法相同的值。

public JoinPoint.StaticPart getStaticPart()

返回包含关于可以静态确定的连接点信息的对象。用于访问静态信息的一种替代方法是使用可供任何通知块使用的 thisJoinPointStaticPart 引用变量。

JoinPoint.StaticPart 接口

最常使用可供所有通知块使用的 thisJoinPointStaticPart 引用变量来访问 JoinPoint.StaticPart 接口。一种替代方法是使用 JoinPoint 接口上的 getStaticPart()方法，可以使用 thisJoinPoint 引用访问它。

public String toString()

重写对象上的 toString()默认方法，返回连接点的字符串表示。

注意：JoinPoint.toString()方法返回与这个方法相同的值。

public String toShortString()

提供 toString()方法的一个替代方法，返回连接点的字符串表示的一个简化版本。

注意：JoinPoint.toShortString()方法返回与这个方法相同的值。

public String toLongString()

提供 toString()方法的一个替代方法，返回连接点的字符串表示的一个扩展版本。

注意：JoinPoint.toLongString()方法返回与这个方法相同的值。

public Signature getSignature()

返回连接点上的签名。

注意：JoinPoint.getSignature()方法返回与这个方法相同的值。

public SourceLocation getSourceLocation()

返回包含关于连接点源位置的所有可用信息（如果有的话）的对象。如果没有可用的源位置信息，则返回 null。

注意：JoinPoint.getSourceLocation()方法返回与这个方法相同的值。

public String getKind()

返回一个指示连接点类型的字符串枚举。

注意：JoinPoint.getKind()方法的描述表明 AspectJ 支持不同的枚举。

Signature

Signature 接口用于确定与连接点关联的代码签名。Signature 接口允许访问任何连接点上可用的信息，而其子接口（在下一节中介绍）则提供了更多的信息，这依赖于连接点的类型（参见图 A-3）。

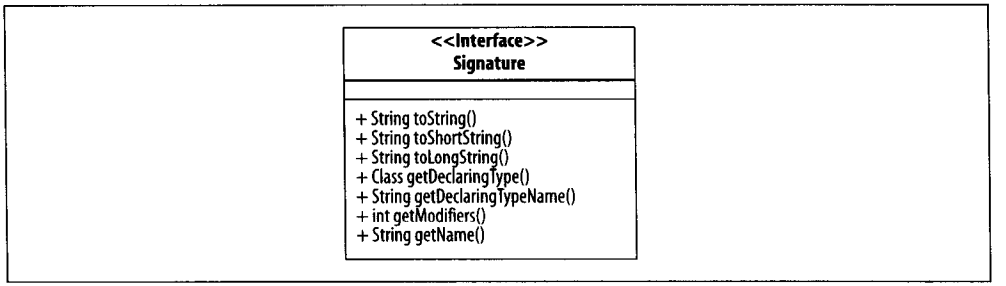

图 A-3：Signature 接口

Signature 接口

`public String toString()`

重写对象上的 `toString()` 默认方法，返回签名的字符串表示。从方法调用连接点上的 `toString()` 方法返回字符串的一个示例如下：

```
void com.oreilly.aspectjcookbook.MyClass.foo(int, String)
```

`public String toShortString()`

提供 `toString()` 方法的一个替代方法，返回签名的字符串表示的一个简化版本。从方法调用连接点上的 `toShortString()` 方法返回字符串的一个示例如下：

```
MyClass.foo(..)
```

`public String toLongString()`

提供 `toString()` 方法的一个替代方法，返回签名的字符串表示的一个扩展版本。从方法调用连接点上的 `toLongString()` 方法返回字符串的一个示例如下：

```
public void com.oreilly.aspectjcookbook.MyClass.foo(int, java.lang.String)
```

`public Class getDeclaringType()`

返回一个类，它声明了与相应签名关联的成员。

`public String getDeclaringTypeName()`

返回与相应签名关联的成员的完全限定名称。从方法调用连接点上的 `getDeclaringTypeName()` 方法返回字符串的一个示例如下：

```
com.oreilly.aspectjcookbook.MyClass
```

`public int getModifiers()`

返回设置在相应签名上的修饰符。

注意：可以使用`java.lang.reflect.Modifier`类来操作表示修饰符的`int`值。

`public String getName()`

只返回签名的标识符部分。从方法调用连接点上的`getName()`方法返回字符串的一个示例如下：

```
Foo
```

org.aspectj.lang.reflect

`org.aspectj.lang.reflect`包中包含所选的更特殊化的类，它们表示连接点的特定类型；还包含它们能够在运行时提供的额外信息。

Signature 特殊化的子接口

依赖于连接点，可以使用多个特殊化的子接口来访问不同连接点上的签名的细节。图 A-4 显示了可以与连接点关联的不同接口之间的关系。

AdviceSignature 接口

表示通知块上的签名：

`public Class getReturnType()`

返回特定通知块返回的对象（如果有的话）的类型。

CatchClauseSignature 接口

表示 catch 代码块的签名：

`public String getParameterName()`

返回作为参数传递给特定 catch 块的对象名称。

`public Class getParameterType()`

返回作为参数传递给特定 catch 块的对象类型。

CodeSignature 接口

表示通用代码块的签名：

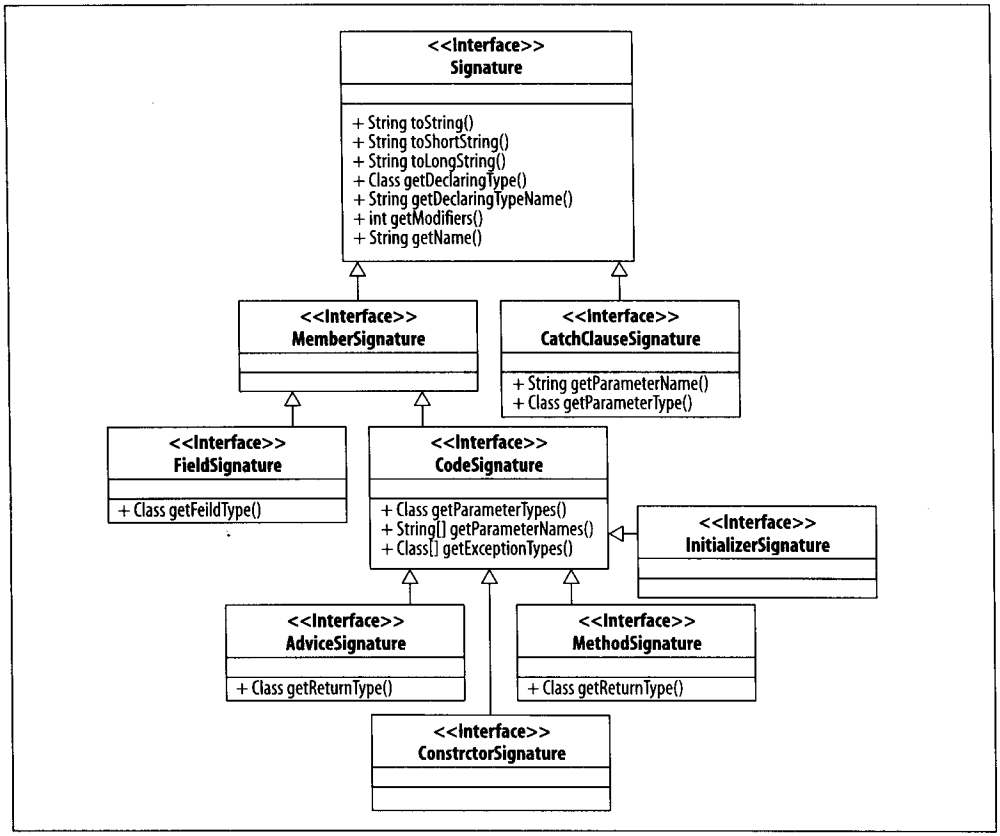

图 A-4：Signature 接口及其特殊化的子接口之间的关系

`public Class[] getParameterTypes()`

返回签名上任何可用参数的类型。

`public String[] getParameterNames()`

返回签名上任何可用参数的名称。

`public Class[] getExceptionTypes()`

返回可以在签名上引发的任何可用异常的类型。

ConstructorSignature 接口

表示构造函数的签名。

FieldSignature 接口

表示字段访问或修改的签名：

`public Class getFieldType()`

依据签名以及关联的连接点，返回访问或修改的字段类型。

`InitializerSignature` 接口

表示对象初始化的签名。

MemberSignature 接口

表示涉及类成员的连接点的签名。

MethodSignature 接口

表示方法上的签名：

`public Class getReturnType()`

返回特定方法返回的对象（如果有的话）的类型。

SourceLocation

`SourceLocation`接口定义了一些潜在地可以访问的信息，这些信息是关于在源位置中遇到特定连接点的位置（参见图 A-5）。

`public String getFileName()`

返回关联连接点所处源文件的文件名。

`public int getLine()`

返回关联连接点所处源文件中的行。

`public Class getWithinType()`

返回连接点所处位置的类型。

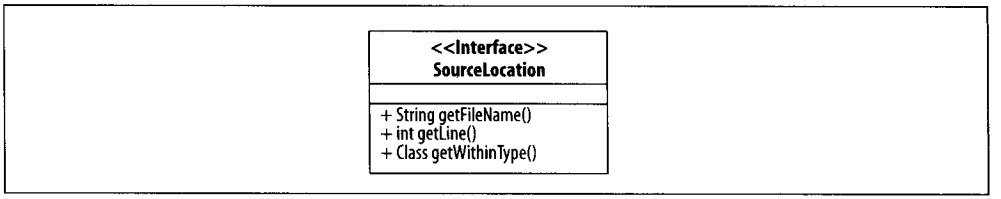

图 A-5：SourceLocation 接口

SoftException 类

当把在连接点上抛出的异常声明为将其软化成运行时异常时，就会使用SoftException类。

12.5 节说明了如何使用异常软化技术。

NoAspectBoundException 类

当调用aspectOf(..)方法以获得对特定方面的访问，并且没有指定类型的方面可用时，就会抛出一个异常，此时潜在地会使用 NoAspectBoundException 类。

[General Information]
书名=AspectJ Cookbook中文版
作者=（美）米尔斯（Miles, R.）著；程利剑译
页数=316
SS号=11752456
出版日期=2006
出版社=北京：清华大学出版社
ISBN号=7-302-14011-1
中图法分类号=TP312
原书定价=36.00
主题词=JAVA语言-程序设计
参考文献格式=（美）米尔斯（Miles, R.）著；程利剑译.AspectJ Cookbook中文版.北京：清华大学出版社,2006.
内容提要=本书利用受欢迎的问题，解决方案，讨论的形式，介绍面向方面的编程。

封面
书名
版权
前言
目录
前言
第1章　面向方面概述
　面向方面简史
　AspectJ
　面向方面的定义
　在哪里找到更多信息
第2章　初识AspectJ
　2.0　简介
　2.1　安装AspectJ
　2.2　开发一个简单的方面
　2.3　编译一个方面和多个Java文件
　2.4　织入方面到jar中
　2.5　在加载时织入方面
　2.6　生成Javadoc文档
　2.7　使用Eclipse编译一个AsPectJ项目
2.8　选择织入Eclipse内的构建中的方面
2.9　使用Ant构建一个AspectJ项目
第3章　部署AspectJ应用程序
　3.0　简介
　3.1　部署命令行AspectJ应用程序
　3.2　将AspectJ应用程序部署为全包含的可执行JAR文件
　3.3　部署使用AspectJ的Java Servlet
　3.4　部署使用AspectJ的JSP
　3.5　部署使用AspectJ的Axis Web服务
第4章　捕获方法上的连接点
　4.0　简介
　4.1　捕获方法调用
　4.2　捕获方法调用上传递的参数值
　4.3　捕获方法调用的目标
　4.4　当执行一个方法时捕获它
　4.5　在执行方法时捕获this引用的值

第5章　捕获异常处理上的连接点
5.0　简介
5.1　捕获何时捕捉异常
5.2　捕获抛出的异常
5.3　捕获处理异常的对象
第6章　捕获通知上的连接点
6.0　简介
6.1　捕获何时执行通知
6.2　排除作为通知执行结果的连接点
6.3　在发出通知时展示原始连接点
第7章　捕获类和对象构造上的连接点
7.0　简介
7.1　捕获对构造函数的调用
7.2　在执行构造函数时捕获它
7.3　捕获何时初始化对象
7.4　捕获何时将要初始化一个对象
7.5　捕获何时初始化类
第8章　捕获属性上的连接点
8.0　简介
8.1　捕获何时访问对象的属性
8.2　捕获访问的字段值
8.3　捕获何时修改对象的字段
8.4　在修改字段时捕获它的值
第9章　捕获程序作用域内的连接点
9.0　简介
9.1　捕获特定类中的所有连接点
9.2　捕获特定包中的所有连接点
9.3　捕获特定方法内的所有连接点
第10章　捕获基于控制流程的连接点
10.0　简介
10.1　捕获通过初始连接点开始的程序控制流程内的所有连接点
10.2　捕获程序控制流程内的所有连接点，不包括初始连接点
第11章　捕获基于对象类型的连接点
11.0　简介
11.1　捕获何时this引用是一个特定的类型
11.2　捕获何时连接点的目标对象是特定的类型

11.3　捕获何时连接点的参数是某个数字、类型和次序
第12章　捕获基于布尔或复合表达式的连接点
12.0　简介
12.1　捕获何时连接点上的运行时条件评估为true
12.2　使用逻辑AND（&&）结合切入点
12.3　使用逻辑OR（||）结合切入点
12.4　捕获所有未通过切入点声明指定的连接点
12.5　声明匿名切入点
12.6　重用切入点
第13章　定义通知
13.0　简介
13.1　访问类成员
13.2　访问连接点环境
13.3　在连接点之前执行通知
13.4　在连接点周围执行通知
13.5　在连接点之后无条件执行通知
13.6　仅在从连接点正常返回之后才执行通知
13.7　仅当连接点中引发了一个异常之后才执行通知
13.8　控制通知优先级
13.9　通知方面
第14章　定义方面实例化
14.0　简介
14.1　定义单件方面
14.2　在每个实例上定义一个方面
14.3　在每个控制流程上定义一个方面
第15章　定义方面的关系
15.0　简介
15.1　继承切入点定义
15.2　实现抽象切入点
15.3　把类继承进方面中
15.4　在类内部声明方面
第16章　增强类和编译器
16.0　简介
16.1　扩展现有的类
16.2　声明类之间的继承关系
16.3　使用方面实现接口

16.4　声明默认的接口实现
16.5　减轻异常的影响
16.6　扩展编译
第17章　实现创建型面向对象设计模式
17.0　简介
17.1　实现单件模式
17.2　实现原型模式
17.3　实现抽象工厂模式
17.4　实现工厂方法模式
17.5　实现生成器模式
第18章　实现结构型面向对象设计模式
18.0　简介
18.1　实现复合模式
18.2　实现享元模式
18.3　实现适配器模式
18.4　实现桥接模式
18.5　实现装饰模式
18.6　实现代理模式
第19章　实现行为型面向对象设计模式
19.0　简介
19.1　实现观察者模式
19.2　实现命令模式
19.3　实现迭代器模式
19.4　实现调停者模式
19.5　实现责任链模式
19.6　实现备忘录模式
19.7　实现策略模式
19.8　实现访问者模式
19.9　实现模板方法模式
19.10　实现状态模式
19.11　实现解释器模式
第20章　应用类和组件级方面
20.0　简介
20.1　验证传递给方法的参数
20.2　重写在构造函数调用上实例化的类
20.3　添加持久性到类中

20.4　应用模拟组件支持单元测试
第21章　应用应用程序级方面
21.0　简介
21.1　应用面向方面的跟踪
21.2　应用面向方面的日志记录
21.3　应用延迟加载
21.4　管理应用程序属性
第22章　应用企业级方面
22.0　简介
22.1　应用开发指导和规则
22.2　应用事务
22.3　应用资源池
22.4　使用RMI透明地远程访问类
22.5　应用安全策略
第23章　应用面向方面设计模式
23.0　简介
23.1　应用Cuckoo's Egg设计模式
23.2　应用Director设计模式
23.3　应用Border Control设计模式
23.4　应用策略设计模式
附录　AspectJ运行时API