eclipse

PEARSON Addison Wesley

Eclipse AspectJ
Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools

Eclipse AspectJ 中文版

——利用 Eclipse 和 AspectJ 进行面向方面程序设计

(美) Adrian Colyer
Andy Clement 等著
钱竹青 邹正武 译

PEARSON Education
清华大学出版社

Eclipse AspectJ

Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools

现在每个 Java 开发者都能利用 AOP 的强大功能

使用 AspectJ，Java 开发者就能利用已熟识的语言，在当前最流行的 Eclipse 开发环境中尽情体会面向方面编程(AOP)带来的优势。AOP 能够提高程序的模块性，使得编写的代码更接近于设计目标。它能减少实现普通特性和功能所花费的时间，提高质量，也能在系统和服务中整合简单的 Java 对象以及创建更简单且重用性更高的组件，并且提供了更多的附加功能。

本书是学习 AspectJ 语言的权威指南，涵盖了利用 AspectJ 和 Eclipse 进行 AOP 开发的方方面面，包括从创建新项目到扩展和文档化已完全成型的应用程序的所有内容，并提供了完整的 API 参考以及如何在项目中采用 AspectJ 的现实性指导。从而帮助您轻松掌握 AOP 的重要原理和技术，以应对最棘手的软件质量、效率和维护方面的挑战。

内容提要

- 安装并配置 Eclipse 和 AspectJ 开发工具(AJDT)
- AOP 如何对从错误检测到性能的每个方面进行模块化和优化
- 创建新的 AspectJ 应用程序以及将 AOP 应用到现有系统
- 构建、调试和文档化 AspectJ 应用程序
- 理解关键的 AOP 概念，如连接点、切入点、advice 和类型间声明
- 掌握高级技术：Aspect 库、与已编译好的.class 文件进行链接、可视化、面向方面的设计等

读者对象

所有面向方面的编程爱好者、Java 程序员以及 Eclipse 环境下的编程人员。

作者简介

Adrian Colyer IBM 的高级技术员，AspectJ Eclipse 项目的主管，AJDT 的创始人之一。现负责 IBM 的一个项目组，开发并应用面向方面的技术。

Andy Clement IBM Hursley 实验室的高级软件开发人员，AspectJ 项目的负责人，AJDT 项目的创始人之一。致力于在 J2EE 中间件中使用面向方面的编程技术。

源代码及示例下载

www.awprofessional.com/title/0321245873

www.tupwk.com.cn/downpage

ISBN 7-302-13976-8

9 787302 139768 >

定价：49.80 元

Eclipse AspectJ 中文版

—— 利用 Eclipse 和 AspectJ 进行面向方面程序设计

(美) Adrian Colyer
Andy Clement 等著

钱竹青 邹正武 译

清华大学出版社

北 京

EISBN: 0-321-24587-3

图书在版编目(CIP)数据

Eclipse AspectJ 中文版—— 利用 Eclipse 和 AspectJ 进行面向方面程序设计/(美)科尔勒(Colyer, A.)等著；钱竹青，邹正武 译. —北京：清华大学出版社，2006.12
书名原文：Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools
ISBN 7-302-13976-8
Ⅰ.E… Ⅱ. ①科… ②钱… ③邹… Ⅲ. JAVA 语言—程序设计 Ⅳ.TP312
中国版本图书馆 CIP 数据核字(2006)第 120629 号

责任编辑：王 军 王 婷
装帧设计：孔祥丰
责任校对：成凤进
责任印制：孟凡玉
出版发行：清华大学出版社 **地 址**：北京清华大学学研大厦
http://www.tup.com.cn **邮 编**：100084
c-service@tup.tsinghua.edu.cn
社 总 机：010-62770175 **邮购热线**：010-62786544
投稿咨询：010-62772015 **客户服务**：010-62776969
印 刷 者：清华大学印刷厂
装 订 者：三河市李旗庄少明装订厂
经 销：全国新华书店
开 本：185×260 **印 张**：25.25 **字 数**：524 千字
版 次：2006 年 12 月第 1 版 **印 次**：2006 年 12 月第1 次印刷
印 数：1～4000
定 价：49.80 元

序　言

面向方面的程序设计(Aspect-oriented Programming，AOP)技术来源于 Palo Alto 研究中心(Palo Alto Research Center，PARC)在 20 世纪 80 年代到 90 年代之间的研究成果。第一篇使用这个术语的论文题目为 *Aspect-Oriented Programming*(Kiczales，1997)，发表于 1997 年 6 月。第一个 AspectJ 的公开版本出现在 7 年前，也就是 1998 年 3 月。此后，人们对 AOP 和 AspectJ 的兴趣与日俱增。

我们依然记得当第一个 AOP 程序运行时所体验到的兴奋，简直就是魔术！如今我们既要开发使用 AspectJ 的 AOP 所需的工具，又要将这些工具应用到实际问题中。在工作过程中，我们有幸把 AOP 的思想介绍给更多的开发人员、架构师和管理者。观察 AspectJ 的使用情况是理解这些新思想的重要途径。正是由于这个原因，在同人们谈论 AOP 的时候，我们要描绘出使用 Eclipse AJDT 进行 AOP 开发的场景。而 AJDT 则能帮助我们想象 AspectJ 程序的行为并在一定程度上加深我们对其的理解，这是文字叙述所无法企及的。

本书的目的是让读者全面了解 AOP，并将其成功地应用到自己的项目中。为此，我们会介绍 AOP 的概念、AspectJ 语言以及单个集成软件包中的 Eclipse AJDT。希望通过对书中示例的学习，您也能体会到 AOP 带给您的喜悦。

参考文献

Kiczales，G. et al.，面向方面的程序设计，面向方面程序设计欧洲会议(European Conference on Object-Orientd Programming，ECOOP)，1997 年 6 月。

致　　谢

首先要感谢创立 AspectJ 的 PARC 实验室，尤其是那些让我们有机会与其共同致力于 AspectJ 项目工作的人们：Gregor Kiczales、Jim Hugunin、Erik Hilsdale、Mik Kersten 和 Wes Isberg。Mik Kersten 从最初就全身心地投入到 AspectJ 开发工具(AspectJ Development Tool，AJDT)的开发工作中，在发现 AJDT 作为一个面向 aspect 的 IDE 需要一些什么功能，以及具体操作上做出了重大的贡献；Julie Waterhouse 从第一个版本开始就投入到了 AJDT 的开发中；Ron Bodkin 在 IBM Hursley 实验室协助组建了第一个面向方面程序设计(Aspect-Oriented Programming，AOP)的工作平台，这帮助我们采用 AOP 建立了最初的业务案例。

我们还要感谢所有 AspectJ 和 AJDT 的支持者和使用者，有了你们对工程的参与，才有今天 AspectJ 的地位。还有本书的编辑和书评撰写人，使得这本书能够做的更好。

Adrian 要感谢 IBM 的管理层，对他近乎疯狂的想法给予了足够的支持，使他可以组建开发小组并逐步实现其思想。特别要感谢的是他的家人——Sara、Rebecca、Elliott 和 Derwent，给了他充足的时间来完成项目。而像我已不可能有如此充足的时间了！

Andy 要感谢 Hursley 的 AOSD 项目组，帮助我们建立了一个开发工具，像 Sian Whiting、Matt Chapman、Helen Hawlins 以及 Luzius Meisser。而特别需要感谢的是 Julie，不仅支持他写书，甚至在对 aspect 毫无概念的情况下复查了某些章节。

George 要感谢 Adrian、Andy、Matthew、Sian、Helen、Matt 和 Luzius，能够在日常工作中容忍他的种种行为。也要感谢 AspectJ 小组成员，他们付出了智慧并富有幽默感，当然还有独到的创新思维。另外，还要感谢表现优秀的 Katy，Eleanor 和 Dylan，在本书的写作中表现出了足够的耐心。

Matthew 要感谢 Adrian 介绍他加入 AOSD 开发小组，并被他有远见的领导艺术所折服。过去的两年中由于整个团队成员的帮助，Matthew 学到了很多。同样要感谢 Robert Berry，在令人极度压抑的 2002 年给了他指导性的支持，正是在这一年，Matthew 开始了 AOP 和 Eclipse 的初步尝试。Matthew 特别需要感谢的是 Catherine，在项目开发过程中给予他关怀和照顾。

译 者 序

在传统的面向对象编程(OOP)中，是以类作为编程的基本单元。对于跨越多个类的功能(例如，日志记录、事务管理、设计模式及安全性)，通常不能将它们集中在一个类中处理。这就导致了代码无法重用，可维护性差且产生了大量代码冗余，这是我们不愿看到的。面向方面编程(AOP)的出现正好给处于黑暗中的我们带来了光明。作为 OOP 的扩展和补充，AOP 能很好地实现跨越多个类的模块化功能，提高了面向对象应用程序的性能。

AOP 的思想是将一些多个对象都要实现的操作，例如权限检查、日志记录、安全功能等看成一个切面(方面)，所有对这些对象的访问都要经过这个切面，从而避免了这些公用代码的重复编写，提高了编程的效率。

来自 Xerox PARC 的 AspectJ 是一种 AOP 语言，它是对 Java 的无缝扩展，向 Java 中加入了几个重要的新概念：连接点(join point)、切入点(pointcut)、advice(通知)、类型间声明(inter-type declaration)和 aspect。切入点和 advice 动态地影响程序流程，类型间声明静态地影响程序的类层次结构，而 aspect 则是对所有这些新结构的封装。

本书由 AspectJ 项目领导人 Adrian Colyer 和另外三位经验丰富的 AOP、AspectJ 方面的专家联合编写。全书不仅对 AOP、eclipse 和 AspectJ 语言进行了介绍，还以一个完整的企业应用作为示例贯穿全书，阐明了如何在 eclipse 开发环境中进行面向 aspect 的编程。这不仅使读者能学到 AspectJ 的语法，还能学到基于 AspectJ 语言的开发方法。本书内容详实，叙述清晰，对欲掌握 AOP 和 AspectJ 技术的读者来说，是一本不可多得的好书。

本书由李化组织翻译，由钱竹青、邹正武翻译完成。因为 AOP 是比较新的技术，有些术语尚无一致的翻译方法。虽然我们在翻译过程中力求通俗准确，但限于译者的水平，可能存在错误和不是十分准确之处，欢迎各位读者对本书提供反馈意见。我们希望读者能从本书中受益，也希望通过读者的反馈意见来了解自己的不足，以求在今后的译作中更多更切实际地考虑读者的需要。请将您的反馈信息发送至 fwkbook@tup.tsinghua.edu.cn，我们将不胜感激。

译者

2006 年 5 月

前　言

这是一本关于 AspectJ 语言(一种程序设计语言)和面向方面程序设计(Aspect-Oriented Programming，AOP)的书籍，其中 AOP 对程序构建提出了新的思路。

使用面向对象程序设计语言编写过程式的程序是可能的(许多的情况足以证明这一点)，但是这样做不能利用面向对象方法的全部优点。实际面向对象编程时需要改变对程序的思考方式。同样，当您开始改变思维方法时，AOP 将会展现其强大的优势，希望在我们的帮助下，您可以比较快地做到这一点。

前言的剩余部分将会解释 AOP 产生的背景，并给出一个框架来说明 AOP 能做些什么。而在本书的后续部分，我们会讨论进行面向方面编程的有效途径——AspectJ 语言，以及能够处理使用 AspectJ 所编写程序的工具——AJDT(AspectJ Development Tools)。

读者对象

我们假定本书的读者都熟悉 Java 程序设计语言，但在 AspectJ 方面没有或基本没有任何经验。掌握使用 Eclipse 有助于加快 Java 开发的学习但这不是学习本书所必需的。

从需求到代码

假定读者正在编写代码，对于程序要做什么有一些想法(我们这样希望)，虽不一定在纸上，但至少在头脑中对于程序要支持的主要特征和功能的概念，以及如何将其实现到代码中，有了某种“设计”思路。这样，在设计层思想和源代码实现之间的理想映射中，问题域中唯一的概念、需求和实体都会同具体实现结构有一个清晰的一对一关系。

例如，如果程序需要处理货币数量，将“货币”这个概念同类 Money 建立一对一的关系会是一个比较好的想法。类 Money 封装了程序需要对货币进行描述的所有方面。如果程序需要面向顾客，可以建立“顾客”这一概念和类 Customer 之间的一对一映射。如果顾客有不同的类别，可能就需要建立类 Customer 层次的映射。在上面这两个例子中，很显然，实现的部分与设计层面上货币和顾客的概念相对应。

这种清晰明了的一对一映射对很多方面都有帮助。首先，程序变得更加易于理解。如果想要知道“程序如何处理 X”，只需寻找 X 模块就能得到答案。其次，程序变得更加易于维护。在程序的生命周期中，设计层次上的概念和需求同要改变的单元有着紧密的联系。例如，如果要实现增加货币数量的功能，只需在类 Money 中添加一个 multiply 方法；如果要实现对一种新类型顾客的支持，只需在 customer 层

次上添加一个新类；如果不再需要了解顾客头发的颜色，只需在类 Customer 中删除 hairColor 属性。

然而，对于一些设计层次上的需求，当使用面向对象语言时很难获得与实现结构之间清晰的一对一映射。比如有这样一个简单的需求：当顾客对象所展现的状态更新时需要通知观测结果，这显然不能由一个精确封装的模块来实现，但可以使用贯穿整个顾客层次的代码段来实现。观察类 Customer 的内部，会发现类似于程序清单 0-1 中的代码。同“观测通知”需求相关联的代码部分用粗体表示。

程序清单 0-1　顾客更改通知

```
public class Customer {
  private Address address;
  private String lastName;
  private String firstName;
  private CustomerID id;
  private List listeners;

  public Customer(...) {...} // details omitted
  public void addListener(CustomerListener listener) {
    listeners.add(listener);
  }

  public void removeListener(CustomerListener listener) {
    listeners.remove(listener);
  }

  public Address getAddress() { return this.address; }

  public String getLastName() { return this.lastName; }

  public String setLastName(String name) {
    this.lastName = name;
    notifyListeners(this);
  }
  // etc
}
```

Customer 类中含有添加和删除监听者(listener)的方法，和在每个状态改变操作后对 notifyListeners 方法的调用。顾客层次中其他的类也要确保在每次执行状态改变操作后均调用了 notifyListeners 方法。

此例没有采用精确的一对一映射，而是采用了一对多映射。一个设计层次上的需求变成了多段代码。在 AOP 团队中这种变化被描述为“离散化(scattering)”。实际的程序中有不少一对多映射的例子。设想一个在数据访问层上处理 SQL 异常的方法。当抛出异常 SQLException 时有一种方式可以描述下一步需要做些什么，但是其

执行却要贯穿于整个代码中的 try-catch-finally 块。或者假设需要对一个资源集进行并发访问的控制，允许多个读操作和一个写操作。尽管这只代表设计层次上的需求，但可以证明在一整串的锁和对 acquire 以及 release 的调用上，这种控制会贯穿于所有的读操作和写操作。

无论何时，只要设计层概念和需求与具体实现结构之间存在一对多的映射关系，就会发现许多问题[①]：

- **难以理解需求的实现，并且难以进行相关的推导**——这需要在源代码的多处进行寻找才能获取完整的描述。
- **难以将需求的具体实现添加到代码库中**——对细节的关注(人类似乎并不擅长)要求在每个需要的位置注意增加逻辑关系。因此，在每一个这样的位置都要正确地完成需求的实现。针对具体实现编写良好的测试程序也变得异常困难。
- **难以维护实现**——要改变计算货币数量的方式，只需访问 Money 类即可。而要改变数据访问层上处理异常 SQLException 的方式，就需确保找到所有处理 SQLException 的位置，然后进行统一且正确地更改。这种流程既费时又容易出错。
- **当不需要某个需求时，难以删除代码库中相应的实现**——这个问题等同于实现维护中存在的问题。
- **难以将实现任务交给任何一个小组成员**——让某人去写一个 Money 类显然很容易，然而，让他去写“观测通知”就要困难不少(因为这涉及到许多其他程序员所编写的类)。
- **难以在其他系统上重用该实现**——在其他系统上重用设计和设计所倚靠的基础服务是可能的，但许多具体的实现细节并不是以这些系统能接受的方式来模块化的。

一个程序只精确地完成一个任务的情况是很少见的。当要实现多个设计概念和需求且其中存在一对多的映射关系时，不可避免要采用抛硬币的方式决定。可以使用包含逻辑关系的程序模块(面向对象语言中的类)来处理多个概念和需求，也就是说，在设计和实现之间存在一种多对一的映射关系。程序清单 0-1 中的 Customer 类就展示了二对一的映射关系：单个模块(Customer 类)同时实现了顾客的核心概念和“观测通知”需求。在 AOP 团队里这种实现被形象地描述为“杂乱的(tangling)”：不同的实现成分相互交错地整合在一个模块中。程序清单 0-2 的代码段就是基于这样的逻辑关系，在 IBM 的一个应用服务器内部实现了部分一对一设计概念——实体 bean 的钝化。

---

① 该问题列表摘自 AspectJ 小组编写的 AspectJ 指南。

程序清单 0-2　实体 bean 的钝化

```
try {
  if (!removed) entityBean.ejbPassivate();
  setState(POOLED);
} catch (RemoteException ex) {
  destroy();
  throw ex;
} finally {
  removed = false;
  beanPool.put(this);
}
```

显然，很容易就可以理解其中的逻辑关系和具体的实现步骤。程序清单 0-3 给出了同样代码段的另一个版本，这段代码更接近于真实实现。其中举例说明了某些杂乱的问题(多对一映射)，还有要处理的钝化过程中的核心逻辑关系：将异常交给分析引擎进行诊断的需求；输出统计钝化 bean 次数信息的需求；以及记录出入钝化方法轨迹的需求。

程序清单 0-3　杂乱的实体 bean 钝化

```
try {
  if (!removed) entityBean.ejbPassivate();
  setState(POOLED);
} catch (RemoteException ex) {
  AnalysisEngine.processException(
     ex,"EntityBeanO.ejbPassivate",
     "2078",this);
  destroy();
  throw ex;
} finally {
  if (!removed && (statisticsCollector != null)) {
    statisticsCollector.recordPassivation();
  }
  removed = false;
  beanPool.put(this);
  if (Logger.isEntryExitEnabled) {
    Logger.exit(tc, "passivate");
  }
}
```

当在设计层概念上，以及需求与实现结构之间出现多对一的映射关系时，将会出现很多问题：

- **代码难以理解**——观察程序清单 0-3，其中实体 bean 钝化的逻辑关系显然要比程序清单 0-2 复杂。

- **实现难以维护**—— 要维护源模块中任何一个概念或需求具体化后的实现，需涉及到所有的实现。这就意味着如果开发者要维护诸如 Customer 类，不仅要了解顾客需求，还需知道 “观测通知” 这个需求。因为设计层的概念和需求倾向于整个单元的更改，而每一个又有其自身的变化方式，所以可以得出结论，含多对一映射关系的模块要比含一对一映射关系的模块需要更频繁地维护。任何想说“我只是想改变注释而已”的人都将会明白其中的原因。
- **模块难以测试**—— 例如，想要对 Account 类进行测试，但由于在其具体实现中和安全检测有杂乱的逻辑关系，安全管理系统就必不可少了。因此，测试实体 bean 钝化的逻辑关系需要分析引擎、跟踪器和集合统计模块。
- **难以重用任何一个设计层的概念或需求的具体实现，因为它很大程度上依赖于当前与之相互关联的系统**—— 例如，在一个使用不同安全解决方案(或根本不存在)的系统上重用 Account 类是很难的。

正如不会仅实现单个概念或需求一样，任何规格的程序也不会仅包含一个模块(如果真这么做了，在接触并学习 AOP 之前会遇到更大的问题)。让我们止步不前的并不是设计与需求同具体实现之间一对多或多对一的映射关系，而是多对多的映射关系。为了达到目标，我们从简单的一对一映射开始，却被复杂交错的杂乱阻断。这并不是读者的错[②]。面向对象的方法不提供这种工具，能清晰地将所有的概念和需求映射到模块实现结构中。而这个问题，正是 AOP 能够解决的。AOP 尽可能的把问题接近一对一映射，也就是说，AOP 具有模块性。

AOP 如何工作

AOP 提供了一种新型模块，称为 aspect，在获得设计概念和需求的一对多实现并将其转化为一对一的过程中发挥了重要的作用。使用单个 aspect，通过移除顾客层次内处理“观测通知”的代码并置入其自己的模块中，就可以实现“观测通知”的需求。还可以使用一个稳定的方法来处理整个数据访问层的 SQLException 异常。使用另一个 aspect，可以实现多读操作单写操作对资源进行同步的逻辑关系。另外，可以对程序清单 0-3 中的错误分析、统计和跟踪实现模块化，从而使得实体 bean 钝化的模块变回到程序清单 0-2 的样子。另外，使用 aspect 实现银行应用的安全需求，不会像 Account 类的实现一样那么复杂。

前面所列出的可以用 aspect 进行模块化的特征，在 AOP 团队内部称为横切关系(crosscutting concern)。一个 aspect 模块会对程序执行过程中的多个时刻产生影响(如每次处理异常 SQLException 的时候)。第 4 章将对横切关系进行深入完整的定义。

---

② 或者说不全是您的错。使用好的工具仍然可能写出模块性差的软件，但是使用不恰当的工具一定很难写出模块性好的软件。

丰富程序员的词汇

在面向对象的分析和设计中，我们学会了分析需求状态和查找名词及动词。其中名词变成了候选的类，而动词变成了这些类中候选的方法。AOP 丰富了我们的词汇，它将模块化形容词和副词的实现变成了可能，如安全的业务事务，持久的记录，线程安全的类，账单式的事件。形容词和副词之所以存在，是因为它们定义了独立于名词和动词的概念。换句话说，形容词和副词描绘了可以应用到许多不同实体的概念，因而是横切关系的一种形式。当出现形容词或副词时，总能找到一个候选的 aspect。我们不是建议把 aspect 起名为“安全的(Secured)”，但是对于安全的资源这样的需求就意味着“安全(Security)” aspect 的出现。同样，需要获得缓存的结果时，意味着要使用“缓存(Caching)” aspect。正如所有的设计一样，这并不是简单的黑与白映射关系，每个案例都有它自身的优点，其他形容词和副词的实现技术就有子类和接口。AOP 并不取代 OOP，只是作了适当的补充。

小结

AOP 讨论了如何提高程序的模块化，使设计概念和需求同具体实现结构之间的映射关系接近理想的一对一关系。为了达到这个目的，AOP 提供了一种新型的模块，称为 aspect，可以用来对宽泛的横切关系实现模块化。AOP 代替不了 OOP，但是，它建立在 OOP 之上并作了适当的补充。

本书的结构

本书分为 3 个部分。第 I 部分将介绍 AspectJ 和 AJDT，为使用 AspectJ 进行面向方面的程序设计提供一些初步的知识。第 II 部分在第 I 部分的基础上，将详细讲述 AspectJ 语言的语法和语义，并对从第 I 部分过渡到第 II 部分的过程中出现的难以理解的概念进行阐述。第III部分将讨论如何将 AspectJ 引入项目和组织的策略，指导性地阐述了如何正确地使用 AspectJ，并列举了许多例子。附录中有 AspectJ 的快速参考，一系列继续学习 AspectJ 和了解 AOP 的有用资源，还有关于使用 Eclipse、AJDT 和 AspectJ 进行开发的信息。

目　录