The AspectJ™ Programming Guide

AspectJ 程序设计指南

编译：屈忠慰

目　录

0. 前言

这个程序设计指南做如下三件事情：

- 介绍 AspectJ 语言
- 怎样在程序中找到方面
- AOP 的学习曲线是什么样的？
- 详细说明 AspectJ 每个部份的构造以及它的语义，并且
- 提供有关它们使用的例子

本指南的附录提供了 AspectJ 的一个参考语法、一个内容丰富的形式化的语义学描述以及一个 AspectJ 使用局限说明。

第一个部份：AspectJ 起步，提供了一个写 AspectJ 程序的较温和（比较柔和和简单）的看法，同时这部份内容也为我们显示了如何在一个现有开发项目中引入 AspectJ 并努力减小各方面关联的风险。如果你是第一次接触 AspectJ 或是你想要对 AspectJ 有一个整体的认识，那么你应该读这个部份。

第二部份：AspectJ 语言，在描述中用一些代码片断的例子对 AspectJ 的特征做了更详细的说明。所有 AspectJ 的基础部份都被包含了进来，读过这个部份后你应该能正确地使用 AspectJ 语言。

第三部份：例子，由一系列完整的程序组成，这些例子不仅显示了 AspectJ 一些重要特征的使用，而且还举例说明了 AspectJ 实践中的操作规程，这个部份读过之后你将熟悉 AspectJ 的基本原理。

最后，有两个短章，一个叙述 AspectJ 习惯用语，一个叙述 AspectJ 目前的版本所导致的令人惊奇的奇怪行为。

附件收录了一个 AspectJ 的快速参考语法、一个覆盖面较深的 AspectJ 语义以及一个全面的 AspectJ 局限性的描述。

1. AspectJ 起步

1.1. 概述

许多开发人员被面向方面的程序设计（**aspect-oriented programming (AOP)**）所吸引，但又苦于不知道如何使用这项技术。他们了解横切关注点的概念，并且他们也确实遇到了这样的问题。但如何把 AOP 技术引入开发过程中呢？一般情况下包括了如下几个问题：

- 在现有代码中能使用方面吗？
- 使用方面能得到什么样的受益呢？
- AOP 技术的学习曲线是什么样的呢？
- 用这项新技术要冒多大的风险呢？

本章在AspectJ环境下论述这些问题：AspectJ是Java语言面向方面的扩展。一系列削减

了的例子说明了AspectJ程序设计的种类以及诸如此类的工作将得到怎样的好处。读者可以在http://eclipse.org/aspectj 站点找到更多的关于AspectJ程序设计的完整例子和技术支持材料。

采用新技术确实要冒风险，因为新技术发展得很快说不定那天全变了。因此许多机构对此是很保守的。对这些问题的论述主要集中在下列三个不同的范畴中进行讨论：

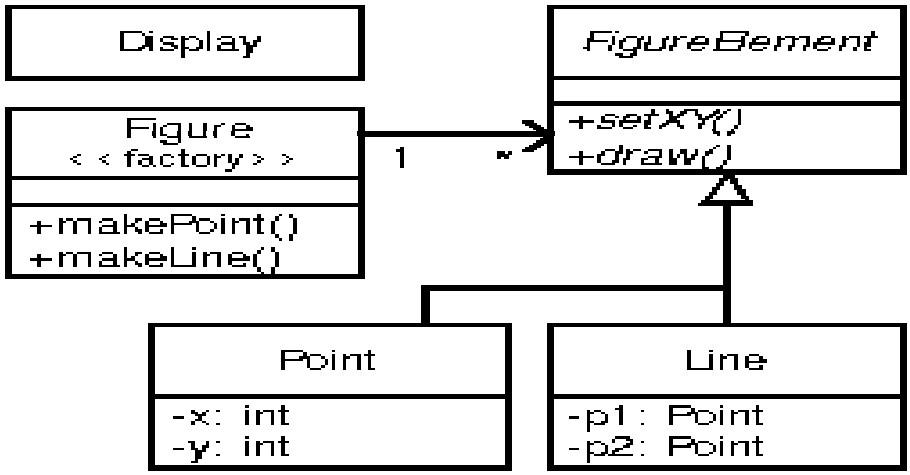
- 在本章中把 AOP 技术较早地引入现有项目的开发；
- 其次我们描述 AspectJ 的核心特征，并且在开发阶段的方面中我们描述方面如何使调试、测试和性能调整变得容易；在随后紧接着的那个部份我们描述 Java 应用中横切面的一般功能；
- 第三我们将在 AspectJ 语言部份讨论方面以及可复用的方面；

上面这些不正式的讨论方式不仅仅只是描述了仅能使用 AspectJ 的方式，一些开发者想要马上使用产品阶段的方面，但我们的经验表明当前的 AspectJ 用户强烈建议采用这种方式来快速获取 AOP 新技术的经验，这也是风险最小的做法。

1.2. AspectJ 入门

我们用随后的这个章节描述 AspectJ 特征的概貌。这些特征是 AspectJ 语言的核心，但不意味着是 AspectJ 的全部。

下面是一个简单的图形编辑系统，一个图由一些FigureElements组成，它们能编辑 Points 或 Lines，这个Figure类提供了一个工厂方法。这里也有一个Display类。本章更多的例子都是基于这个系统来论述的。



FigureEditor 的UML示例

AspectJ（也就是 AOP）的动机是发现那些使用传统的编程方法无法处理得很好的问题。考虑一个要在某些应用中实施安全策略的问题。安全性是贯穿于系统所有模块间的问题，而且每一模块都必须添加安全性才能保证整个应用的安全性，并且安全性模块自身也需要安全性，很明显这里的安全策略的实施问题就是一个横切关注点，使用传统的编程解决此问题非常的困难而且容易产生差错，这就正是 AOP 发挥作用的时候了。

传统的面向对象编程中，每个单元就是一个类，而类似于安全性这方面的问题，它们通常不能集中在一个类中处理因为它们横跨多个类，这就导致了代码无法重用，它们是不可靠和不可承继的，这样可维护性差而且产生了大量代码冗余，这是我们不愿意看到的。

面向方面编程的出现正好给处于黑暗中的我们带来了光明，它针对于这些横切关注点进

行处理，就好象面向对象编程处理一般的关注点一样。AspectJ 是一种面向方面程序设计的基于 Java 的实现。

它向 Java 中加入了连接点（***Join Point***）这个新概念，其实它也只是现存的一个 Java 概念的名称而已。它向 Java 语言中加入少许新结构：切点(***pointcut***)、通知(***Advice***)、类型间声明(***Inter-type declaration***)和方面(***Aspect***)。切点和通知动态地影响程序流程，类型间声明则是静态的影响程序的类等级结构，而方面则是对所有这些新结构的封装。

连接点是程序流中适当的一点。切点收集特定的连接点集合和在这些点中的值。一个通知是当一个连接点到达时执行的代码，这些都是 AspectJ 的动态部分。其实连接点就好比是程序中的一条一条的语句，而切点就是特定一条语句处设置的一个断点，它收集了断点处程序栈的信息，而通知就是在这个断点前后想要加入的程序代码。AspectJ 中也有许多不同种类的类型间声明，这就允许程序员修改程序的静态结构、名称、类的成员以及类之间的关系。AspectJ 中的方面是横切关注点的模块单元。它们的行为与 Java 语言中的类很象，但是方面还封装了切点、通知以及类型间声明。

下面我们首先观察连接点以及如何把它们组合进切点，然后我们看一看通知，通知的代码是在切点到达时运行。我们将看到切点和通知在方面中是如何组合的，也将看到 AspectJ 模块的可重用性、继承性。最后我们考察如何用类型间声明处理一个程序的类结构的横切关注点。

1.2.1. 动态连接点模型

任何面向方面编程的关键元素就是连接点模型。动态连接点模型提供了一个一般的框架，在该框架中定义横切关注点的动态结构。连接点是程序执行中某些具有适当定义的点。AspectJ 提供了许多不同种类的连接点，但本章只介绍它们中的一个：方法调用连接点集(***method call join points***) 。一个方法调用连接点捕捉对象的方法调用，这包括一个方法调用的所有操作——方法调用开始时所有参数的分析以及方法返回(正常地返回或抛出异常)。

每一个方法在运行时都是一个不同的连接点，即使在程序中是同一个调用表达式。许多其它的连接点集合可能在方法调用连接点执行时运行，包括方法执行时的所有连接点集合以及在方法中其它方法的调用。我们说这些连接点集合在原来调用的连接点的动态环境中执行。

1.2.2. 切点

在 AspectJ 中，切点捕捉程序流中某些的连接点集合。例如，切点

call(void Point.setX(int))

捕捉每一个签名为 void Point.setX(int)的方法调用的连接点，也就是说，Point 对象有一个整型参数的 void setX 方法。切点能与其它切点通过或(||)、与(&&)以及非(!)操作符联合。例如：call(void Point.setX(int)) || call(void Point.setY(int)) 捕捉每一个 setX 或 setY 调用的连接点。切点还能够确定不同类型的连接点集合，换句话说，它们能横切类型。例如

```
call(void FigureElement.setXY(int,int)) ||
call(void Point.setX(int))              ||
call(void Point.setY(int)               ||
call(void Line.setP1(Point))            ||        call(void Line.setP2(Point));
```

捕捉上述五个方法调用的任意一个的连接点(顺便指出，第一个方法是接口中的方法)。这个捕捉当 FigureElement 移动时的所有连接点集合。AspectJ 使程序员可以命名一个切点集合，下面的代码声明一个新的命名切点：

```
pointcut move():
    call(void FigureElement.setXY(int,int)) ||
    call(void Point.setX(int))              ||
    call(void Point.setY(int))              ||
    call(void Line.setP1(Point))            ||
    call(void Line.setP2(Point));
```

无论什么时候，这个定义都是可见的，程序员都可以使用 move()代替捕捉这些复杂的切点。

前面所说的切点都是基于显示（明确）的签名方法，我们有时称为基于名字(***name-based***)的横切，AspectJ还提供了一种不同于基于命名横切的机制，它通过方法的属性来指定一个切点，我们称为基于属性(***property-based***)的横切。这种方法直接使用通配符来确定签名方法，例如切点call(void Figure.make*(..)) 捕捉Figure对象中以make开头的参数列表任意的方法调用的连接点。而 call(public * Figure.*(..)) 则捕捉Figure对象中的任何公共方法调用的连接点。

但是通配符不是 AspectJ 支持的唯一属性，例如 cflow 根据连接点集合是否在其它连接点集合的动态环境中发生而标识连接点集合。所以 cflow(move()) 捕捉被 move()捕捉到的连接点集合的动态环境中发生的连接点(参见 1.3.3)，我们命名的切点被定义在这上面，这个切点捕捉每一个这样的连接点集合——当 move()方法被调用和返回(包括正常返回和抛出异常)。

1.2.3. 通知

切点捕捉连接点集合，但除了捕捉连接点集合以外什么事情都没有做。事实上实现横切行为我们要用通知。通知聚合了切点(捕捉到的连接点集合)和在连接点集处要执行的代码。AspectJ 有几种不同种类的通知：

• 前通知(***Before Advice***) 当到达一个连接点但是在程序进程运行之前执行。例如，前通知在方法实际调用之前运行，刚刚在方法的参数被分析之后。

```
before() : move(){ System.out.println("物体将移动了");}
```

• 后通知(***After Advice***) 当特定连接点处的程序进程执行之后运行。例如，一个方法调用的后通知在方法体运行之后，刚好在控制返回调用者之前执行。因为 Java 程序有两种退出连接点的形式，正常的和抛出异常。相对的就有三种后通知：返回后通知(***after returning***)、抛出异常后通知(***after throwing***)和清楚的后通知(***after***)，所谓清楚后通知就是指无论是正常还是异常都执行的后通知，就像 Java 中的 finally 语句。

```
after() returning : move(){
    System.out.println("物体刚刚成功的移动了");
}
```

• 在周围通知(***Around Advice***) 在连接点到达后，显示的控制程序进程是否执行（暂不讨论）

1.2.4. 暴露切点环境

切点不仅仅捕捉连接点，它还能暴露连接点处的部分执行环境。切点中暴露的值可以在通知体中声明以后使用。通知声明有一个参数列表（和方法相同）用来描述它所使用的环境的名称。例如后通知：

```
after(FigureElement fe, int x, int y) returning:
        ...SomePointcut... {
    ...SomeBody...
}
```

使用了三个暴露的环境，一个名为 fe 的 FigureElement 对象，两个整型变量 x,y。

通知体可以像使用方法的参数那样使用这些变量，例如：

```
after(FigureElement fe, int x, int y) returning:
        ...SomePointcut... {
    System.out.println(fe + " moved to (" + x + ", " + y + ")");
}
```

通知的切点发布了通知参数的值，三个原生切点（也叫做原始切点、默认切点、AspectJ 缺省定义的切点或内部初始切点）this、target 和 args 被用来发布这些值，所以上述例子的完整代码为：

```
after(FigureElement fe, int x, int y) returning:
        call(void FigureElement.setXY(int, int))
        && target(fe)
        && args(x, y) {
    System.out.println(fe + " moved to (" + x + ", " + y + ")");
}
```

目标对象是 FigureElement 所以 fe 是 after 的第一个参数，调用的方法包含两个整型参数所以 x 和 y 为 after 的第二和第三个参数。所以通知打印出方法 setXY 调用返回后对象移动到的点 x 和 y。

象通知的一部份一样，一个命名切点可以有参数，当这个命名切点被通知或其它切点使用时它象上面的 this、target 和 args 一样暴露了环境，因此也可以用另外的方法写上面的通知：

```
pointcut setXY(FigureElement fe, int x, int y):
    call(void FigureElement.setXY(int, int))
    && target(fe)
    && args(x, y);

after(FigureElement fe, int x, int y) returning: setXY(fe, x, y) {
    System.out.println(fe + " moved to (" + x + ", " + y + ").");
}
```

1.2.5. 类型间声明

在 AspectJ 中的类型间声明指的是那些跨越类和它们的等级结构的声明。它们可能是横跨多个类的成员声明或者是类之间继承关系的改变。不像通知那样动态地操作，类型间声明

是编译时的静态操作。

我们可以考虑一下，在 Java 语言中如何向一个类中加入新的方法，这需要实现一个特定接口，所有类都必须在各自内部实现接口声明的方法，这种类与类之间的影响是极坏的。而使用 AspectJ 则可以将这些工作利用类型间声明放在一个方面中。这个方面声明方法和字段（或叫做域或类的属性），然后将它们与需要的类联系。

假设我们想要一个 Screen 对象来观察 Point 对象的变化，当 Point 是一个已经存在的类。我们可以通过书写一个方面，由这个方面声明 Point 对象有一个实例字段 observers 用来保存所有观察 Point 对象的 Screen 对象的引用，从而实现这个功能。

```
aspect PointObserving {
    private Vector Point.observers = new Vector();
    ...
}
```

observers 字段是私有字段，只有 PointObserving 能使用。因此，要在 aspect 中加入方法来管理 observers。

```
aspect PointObserving {
    private Vector Point.observers = new Vector();
    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }
    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }
    ...
}
```

然后我们可以定义一个切点 changes 决定我们想要观察什么并且提供一个 after 通知定义当观察到变化时我们想要做什么：

```
aspect PointObserving {
    private Vector Point.observers = new Vector();

    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }
    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }

    pointcut changes(Point p): target(p) && call(void Point.set*(int));

    after(Point p): changes(p) {
        Iterator iter = p.observers.iterator();
        while ( iter.hasNext() ) {
            updateObserver(p, (Screen)iter.next());
        }
    }
```

```
    static void updateObserver(Point p, Screen s) {
        s.display(p);
    }
}
```

注意无论是 Screen 还是 Point 的代码都没有被修改，所有的新功能的加入都在方面中实现了。

1.2.6. 方面

方面以横切模块单元的形式包装了所有的切点、通知和类型间声明。这非常像 Java 语言的类。实际上，方面也可以定义自己的方法，字段和初始化方法。像类一样一个方面也可以用 abstract 关键字声明为抽象方面，可以被子方面继承。在 AspectJ 中方面的设计实际上使用了单例模式，缺省情况下，它不能使用 new 构造，但是可以使用一个方法实例化例如方法 aspectOf()可以获得方面的实例。所以在方面的通知中可以使用非静态的成员字段。

例如：

```
aspect Logging {
    OutputStream logStream = System.err;

    before(): move() {
        logStream.println("about to move");
    }
}
```

1.3. 用于开发的方面

下面两个部份描述 AspectJ 的使用方式。用于开发的方面可以很容易从产品中删除，用于产品的方面可以用于开发和生产过程，而且只影响少数几个类。

接下来这个部份我们用例子描述 Java 应用的开发阶段如何使用方面。这些方面有助于调试、测试和性能调整工作。方面定义的行为范围包括简单的代码跟踪、环境解释、测试应用的内在联系等等。使用 AspectJ 可以很清楚地使这些功能范畴模块化，而且在需要的时候可以很容易地关闭和打开这些功能项。

1.3.1. 代码跟踪

下面的例子为我们显示了如何增加一个程序内部工作的可视性。这个简单的方面用于代码跟踪并且在指定方法调用时输出一些信息，在我们的图形编辑系统中，无论什么时候这个方面能简单地跟踪被画出来的点：

```
aspect SimpleTracing {
    pointcut tracedCall():
        call(void FigureElement.draw(GraphicsContext));

    before(): tracedCall() {
```

```
            System.out.println("Entering: " + thisJoinPoint);
        }
}
```

这段代码用了 thisJoinPoint 变量。在所有的通知体内，这个变量将与描述当前连接点的对象绑定。所以上述代码在每次一个 FigureElement 对象接受到 draw 方法时输出如下信息：

```
Entering: call(void FigureElement.draw(GraphicsContext))
```

理解使用 AspectJ 代码跟踪方法调用的集合是有好处的。AspectJ 只需要定义切点 tracedCall 和对其进行编译，被跟踪的方法不需要重新编辑。

调试时，程序设计者通常要花很大的精力针对问题的细节找出一些适当的跟踪点(程序流程中的一点)。调试完成时又要从代码中不情愿地删除那些为完成调试而增加的代码。这种代码很难看并且为了调试这些代码所做的跟踪声明会对其它代码调试的跟踪声明产生负作用。

使用 AspectJ 我们可以很容易地克服以上的两个问题——设计一个适当的跟踪点集合并在不需要跟踪时阻止其发挥作用，用方面写一个跟踪模块而在不需要时移走这个方面。

AspectJ 的这种简明的实现和可重用的调试构造有力地证明了一个结论：当我们需要某种信息的时候，AspectJ 模块化一个切面对于代码跟踪非常适合。

1.3.2. 环境解释和日志分析

在这个部份中，我们举例说明如何做详细而且精确的环境解释。虽然有很多的环境解释工具可供使用并且这些工具是很好的搜集信息和显现结果的有用方式，但有些时候你可能需要有关环境和日志的非常详细而且精确的行为分析。在这种情况下，你可以在这项工作之上写一个简单的方面。

下面的这个方面对 Line 对象的 rotate 方法调用进行计数并且也对发生在流程控制内的 Line 对象的 rotate 方法调用（包括 rotate 本身）和 Point 对象的 set*方法调用进行计数：

```
aspect SetsInRotateCounting {
    int rotateCount = 0;
    int setCount = 0;

    before(): call(void Line.rotate(double)) {
        rotateCount++;
    }

    before(): call(void Point.set*(int))
                && cflow(call(void Line.rotate(double))) {
        setCount++;
    }
}
```

这段代码实现了下列任务(问题)：

- 那些时候 Line 对象的 rotate 方法被调用？
- 那些时候在 rotate 方法调用达到时 Point 对象中被定义为以“set”开头的方法被调用？

用传统的环境解释和日志分析工具处理这些问题是较为困难的。

1.3.3. 前提条件和后续条件

许多程序设计者用类似 Bertand Meyer 在《面向对象的软件构造》中介绍的那样按契约编程，在这种程序设计风格中要用显示的前提条件测试方法的调用是否适合并且还要用后续条件来保证方法调用工作也是适当(正常)的。

AspectJ 能实现前提条件和后续条件的模块化形式。例如方面：

```
aspect PointBoundsChecking {

    pointcut setX(int x):
        (call(void FigureElement.setXY(int, int)) && args(x, *))
        || (call(void Point.setX(int)) && args(x));

    pointcut setY(int y):
        (call(void FigureElement.setXY(int, int)) && args(*, y))
        || (call(void Point.setY(int)) && args(y));

    before(int x): setX(x) {
        if ( x < MIN_X || x > MAX_X )
            throw new IllegalArgumentException("x is out of bounds.");
    }

    before(int y): setY(y) {
        if ( y < MIN_Y || y > MAX_Y )
            throw new IllegalArgumentException("y is out of bounds.");
    }
}
```

是一个实现了的边界检测方面(点移动的前提条件检测)。

注意这里 setX 切点引用了那些能设置 Point 对象的 x 值的操作，这也包括 setX 方法和作为 setXY 方法的一部份的 setX 方法。要理解这里的 setX 切点正如看到的那样能有条例的横切——指定方法 setX 和 setXY 方法的一部分进行调用。

尽管前提条件和后续条件测试在软件测试期间是最适合的，但在一些情况下开发者希望最好能将它们包括进产品开发阶段，AspectJ 能使横切关注点达到比较纯的模块化，所以开发者还是得设计一个好的控制结构。

1.3.4. 合同执行

在定义复杂的合同执行的时候，基于属性的横切机制对这非常有用。一个十分强大的功能是它可以强制特定的方法调用只出现在对应的程序中，而在其它程序中不出现。例如，下面的方面实施了一个限制，使得只有在知名的工厂方法中才能注册并添加 FigureElement 对象。实施这个限制的目的是为了确保没有任何一个 FigureElement 对象被注册多次：

```
static aspect RegistrationProtection {
    pointcut register(): call(void Registry.register(FigureElement));
    pointcut canRegister(): withincode(static * FigureElement.make*(..));
```

```
    before(): register() && !canRegister() {
        throw new IllegalAccessException("Illegal call " + thisJoinPoint);
    }
}
```

这个方面使用了 withincode 初始切点，它表示在 FigureElement 对象的工厂方法(以 make 开头的方法)体内出现的所有连接点。这是一个基于属性的切点因为它识别连接点的方式并不基于签名(即方法的签名)，但它发生在明确的其它方法内部。在 before 通知中声明一个异常，该通知用于捕捉任何不在工厂方法代码内部产生的 register 方法的调用。该通知在特定连接点集处抛出一个运行时异常，AspectJ 做得更好。使用 declare error 的形式，我们可以声明一个编译时的错误，比如：

```
static aspect RegistrationProtection {

    pointcut register(): call(void Registry.register(FigureElement));
    pointcut canRegister(): withincode(static * FigureElement.make*(..));

    declare error: register() && !canRegister(): "Illegal call"
}
```

当使用这个方面后，如果代码中存在定义的这些非法调用我们将无法通过编译。这种情况只出现在我们只需要静态信息的时候，如果我们需要动态信息，像上面提到的前提条件实施时，就可以利用在通知中抛出带参数的异常来实现。

注意，withincode 切点在它们的代码之上完整地捕捉连接点集合并且使用静态的连接点集合。

1.3.5. 配置管理

AspectJ 配置管理能有把握地使用类似 make-file 技术。程序员可以简单的包括他们想要的方面进行编译。

不想要任何方面出现在产品阶段的开发者也可以通过配置他们的 make-file 使用传统的 Java 编译器编译整个应用。可以很容易地编写 make 文件，AspectJ 编译器有一个命令行接口，它和普通得 Java 编译器是一致的。

1.4. 用于产品的方面

下面这些方面的例子将被包括进一个已有应用的产品建造阶段。产品阶段的方面注重添加功能到一个应用之中远胜于添加可视性到一个程序之中。

基于名字的用于产品阶段的方面仅仅只影响少数几个方法，正是由于这个原因，你可以在你的下一个项目中采用 AspectJ 并从中受益，虽然它们影响小而且简单，但是它们对程序的开发和维护有十分重要的意义。

1.4.1. 改变监视

下面的这个例子显示了用于产品的方面如何实现一些功能性的问题，我们将进行尝试并

且明确地实现它。假设有一段显示刷新的代码，方面的角色是用于维护一位数据标志，由它说明对象从最后一次显示刷新开始是否移动过。在方面中实现这样的功能是十分直接的，testAndClear 方法被其它代码显示地调用以便找到一个图形元素是否在最近移动过。这个方法返回标志的状态并将它设置为假。切点 move 捕捉所有能够使图形移动的方法调用。After 通知截获 move 切点并设置标志位：

```
aspect MoveTracking {
    private static boolean dirty = false;

    public static boolean testAndClear() {
        boolean result = dirty;
        dirty = false;
        return result;
    }

    pointcut move():
        call(void FigureElement.setXY(int, int)) ||
        call(void Line.setP1(Point))             ||
        call(void Line.setP2(Point))             ||
        call(void Point.setX(int))               ||
        call(void Point.setY(int));

    after() returning: move() {
        dirty = true;
    }
}
```

这个简单例子同样说明了在产品代码中使用 AspectJ 的一些好处。考虑使用普通的 Java 代码实现这个功能：将有可能需要包含标志位，testAndClear 以及 setFlag 方法的辅助类。这些方法需要每个移动的图形元素包含一个对 setFlag 方法的调用。这些方法的调用就是这个例子中的横切关注点：

- 显示的捕捉了横切关注点的结构
- 进一步开发是容易的
- 功能容易拔插
- 实现更加稳定

1.4.2. 上下文传递

横切结构的上下文传递在 Java 程序中是十分复杂的一部分。考虑实现一个功能，它允许客户设置所创建的图形对象的颜色。这个需求需要从客户端传入一个颜色或颜色工厂。而要在大量的方法中加入一个参数，目的仅仅是为传递上下文信息这种不方便的情况是所有的程序员都十分熟悉的。

使用 AspectJ，这种上下文的传递可以使用模块化的方式实现。下面代码中的 after 通知仅当一个图形对象的工厂方法在客户 ColorControllingClient 的某个方法控制流程中被调用时才运行。

```
aspect ColorControl {
    pointcut CCClientCflow(ColorControllingClient client):
        cflow(call(* * (..)) && target(client));

    pointcut make(): call(FigureElement Figure.make*(..));

    after (ColorControllingClient c) returning (FigureElement fe):
            make() && CCClientCflow(c) {
        fe.setColor(c.colorFor(fe));
    }
}
```

这个方面仅仅影响一小部分的方法，但是注意该功能的非 AOP 实现可能需要编辑更多的方法。

1.4.3. 提供一致的行为

下面这个例子为我们展示了基于属性的方面如何提供有很多操作的功能一致性。这个方面确保包 com.bigboxco 的所有公共方法记录有它们抛出的任何错误。PublicMethodCall 切点捕捉包中的公共方法调用，after 通知在任何一个这种调用抛出错误后运行并且记录下这个错误。

```
aspect PublicErrorLogging {
    Log log = new Log();

    pointcut publicMethodCall():
        call(public * com.bigboxco.*.*(..));

    after() throwing (Error e): publicMethodCall() {
        log.write(e);
    }
}
```

在一些情况中，这个方面可以记录一个异常两次。这在 com.bigboxco 包内部的代码自己调用本包中的公共方法时发生。为解决这个问题，我们可以使用 cflow 初始切点将这些内部调用排除：

```
after() throwing (Error e):
        publicMethodCall() && !cflow(publicMethodCall()) {
    log.write(e);
}
```

下面的这个方面工作在 AspectJ 编译器上。这个方面 JavaParser 类的 35 个方法周围散发通知。其中每个方法处理必须被解析的不同种类的构成元素。这些方法的名字形如：parseMethodDec、parseThrows 和 parseExpr。

```
aspect ContextFilling {
    pointcut parse(JavaParser jp):
        call(* JavaParser.parse*(..))
        && target(jp)
        && !call(Stmt parseVarDec(boolean)); // var decs
                                             // are tricky

    around(JavaParser jp) returns ASTObject: parse(jp) {
        Token beginToken = jp.peekToken();
        ASTObject ret = proceed(jp);
        if (ret != null) jp.addContext(ret, beginToken);
        return ret;
     }
}
```

这个例子展现了在有大量的基于属性的连接点的许多方面中建立基于属性的切点的方法。例子中添加了基于属性的一个形如 call(* JavaParser.parse*(..))的切点，也包括了一个例外——形如!call(Stmt parseVarDec(boolean))的切点。排除 parseVarDec 是有好处的，因为在 Java 中变量声明的解析太复杂了，以至于难于和其它的 parse*方法的清晰表达相适应。也就是说 parse*方法返回 ASTObject 对象而排除了对上下文解析的影响。

1.5. 结论

AspectJ 是对 Java 语言的简单而且实际的面向方面的扩展。仅通过加入几个新结构，AspectJ 提供了对模块化实现各种横切关注点的有力支持。向已经有的 Java 开发项目中加入 AspectJ 是一个直接而且渐增的任务。一条路径就是通过从使用开发方面开始再到产品方面当拥有了 AspectJ 的经验后就使用开发可重用方面。当然可以选取其它的开发路径。例如，一些开发者将从使用产品方面马上得到好处，另外的人员可能马上编写可重用的方面。

AspectJ 可以使用基于名字和基于属性这两种横切点。使用基于名字横切点的方面仅影响少数几个类，虽然它们是小范围的，但是比起普通的 Java 实现来说它们能够减少大量的复杂度。使用基于属性横切点的方面可以有小范围或着大范围。使用 AspectJ 导致了横切关注点的清晰、模块化的实现。当编写 AspectJ 方面时，横切关注点的结构变得十分明显和易懂。方面也是高度模块化的，使得开发可拔插的横切功能变成现实。

2. AspectJ 语言

2.1. 概述

在本系列的前一章中，我们简要的说明了 AspectJ 语言的总揽。为了理解 AspectJ 的语法和语义，你应该阅读本章。这一部分包括了前述的一些材料，但是将更加完整和更多的讨论细节。文章将由一个具体方面的例子开始，这个方面包括了一个切点，一个类型间声明和两个通知，这个例子将给我们一些讨论的话题。

2.2. 剖析一个方面

2.2.1. 一个方面的例子

下面是一个用 AspectJ 定义的方面例子：

```
1 aspect FaultHandler {
 2
 3    private boolean Server.disabled = false;
 4
 5    private void reportFault() {
 6        System.out.println("Failure! Please fix it.");
 7    }
 8
 9    public static void fixServer(Server s) {
10        s.disabled = false;
11    }
12
13    pointcut services(Server s): target(s) && call(public * *(..));
14
15    before(Server s): services(s) {
16        if (s.disabled) throw new DisabledException();
17    }
18
19    after(Server s) throwing (FaultException e): services(s) {
20        s.disabled = true;
21        reportFault();
22    }
23 }
```

FaultHandler 包括一个在 Server 上的类型间字段声明（第 3 行），两个方法(5-7 行和 9-11 行)，切点定义（13 行），两个通知（15-17 行和 19-22 行）。这些覆盖了方面能够包括的基本信息。通常来说，方面包括其它程序实体、不同的变量和方法、切点定义、类型间声明和通知（可能有 before、after 或 around）。文章的余下部分将逐一讨论这些横切相关的构造。

2.2.2. 切点

AspectJ 的切点定义为切点取名。切点自己捕捉连接点集合，例如，程序执行中的感兴趣的点集合。这些连接点可以是方法或者构造子（或叫构造方法）的调用或执行，异常处理，字段的赋值和读取等等。举个例子，在 13 行的切点定义：

```
pointcut services(Server s): target(s) && call(public * *(..))
```

这个切点，被命名为 services，捕捉当 Server 对象的公共方法被调用时程序执行过程中的连接点。它同样允许使用 services 切点的任何人访问那些方法被调用的 Server 对象。FaultHandler 方面的这个切点背后的思想是指错误处理相关的行为必须由公共方法的调用来

触发。例如，server 可能因为某些错误不能处理请求。因此，对那些方法的调用是该方面感兴趣的事件，当这些事件发生时，对应错误相关的事情也将发生。

事件发生时的部分上下文由切点的参数暴露。在这个例子中，就是指 Server 类型的对象。这个参数在切点声明的右边被使用，以便指明哪个事件与切点相关。在例子中，services 切点包括两部分，它是由捕捉那些以 Server 为目标对象的操作(target(s))的连接点，与那些捕捉 call 的连接点(call(..))组合起来(&&,意为逻辑与 and)形成的切点。调用连接点(calls)通过方法签名描述，在此例中，使用了几个通配符来表达。在返回类型的位置上是(*)，在方法名的位置是(*)而在参数列表位置是(..);只指定了一个具体信息，就是 public。

切点捕捉程序中的大量连接点。但是它们仅仅捕捉几种连接点。这些种类的连接点代表了 Java 语言中的一些重要概念。这里是对于这些概念的不完整列表：方法调用、方法执行、异常处理、实例化、构造子执行以及字段的访问。每一种连接点都由特定的切点捕捉，你将在本文的其它部分学到它们。

2.2.3. 通知

一个通知由切点和通知代码组成，它定义了在切点捕捉到的连接点处运行的逻辑。例如，15-17 行的通知中的代码

```
{
   if (s.disabled) throw new DisabledException();
}
```

当 Server 的实例调用它的公共方法时执行。19-22 行定义的另一个通知在同一个切点(services)处执行。

```
{
   s.disabled = true;
   reportFault();
}
```

但是第二个通知只有在程序抛出 FaultException 时才执行。有三类的 after 通知：分别基于方法正常结束、方法异常结束和方法一任何方式结束。

2.3. 连接点和切点

2.3.1. 一些切点的例子

考虑下面的 Java 类：

```
class Point {
      private int x, y;

      Point(int x, int y) { this.x = x; this.y = y; }

      void setX(int x) { this.x = x; }
      void setY(int y) { this.y = y; }
```

```
    int getX() { return x; }
    int getY() { return y; }
}
```

为了能够理解 AspectJ 的连接点和切点概念，让我们回顾一下 Java 语言的一些基本原理。

考虑类 Point 中的方法声明

```
void setX(int x) { this.x = x; }
```

这个程序片段说明当 Point 类实例调用名为 setX 有一个整型参数的方法时，程序执行方法体{ this.x=x;}。与此类似的是，类的构造子表明如果当 Point 类使用两个整型参数实例化时，构造子体内的{this.x=x;this.y=y;}将被执行。

用一句话总结就是：当一些事发生时，就有一些东西（语句）被执行。

在面向对象程序中，有一些种类的事情“发生”是由 Java 语言本身所决定。我们把这些称为 Java 的连接点。连接点有一些像方法调用、方法执行、对象实例化、构造子执行、字段引用以及异常处理等组成。

而切点就是用来捕捉这些连接点的结构，例如，下面的切点：

```
pointcut setter(): target(Point) &&
                        (call(void setX(int)) ||
                         call(void setY(int)));
```

捕捉对于 Point 实例上 setX（int）或 setY（int）的每一个方法调用。

看看另外一个例子

```
pointcut ioHandler(): within(MyClass) && handler(IOException);
```

这个切点捕捉类 MyClass 内异常处理代码执行时的每个连接点。

切点定义包括由冒号分割的两部分。左边包括切点的名称和切点的参数（例如事件发生时的数据）。右边则包括切点本身。

下面的切点是特定方法执行时起作用

```
    execution(void Point.setX(int))
```

进行方法调用时则使用

```
    call(void Point.setX(int))
```

异常处理执行时的切点定义如下

```
    handler(ArrayOutOfBoundsException)
```

当前正在使用 SomeType 类型的对象

```
    this(SomeType)
```

SomeType 类型对象为目标对象时

```
    target(SomeType)
```

如果连接点处在 Test 的无参数 main 函数调用流程中

```
    cflow(call(void Test.main())
```

切点还可以使用或(“||”)以及与(“and”)和非(“!”)组合。

• 可以使用通配符。因此

1．execution(* *(..))

2．call(* set(..))

代表（1）不考虑参数和返回值的任何方法执行（2）对于任何参数和返回值的方法名为 set 方法的调用。

• 可以基于类型选择元素，例如

1．execution(int *())

2．call(* setY(long))
3．call(* Point.setY(int))
4．call(*.new(int,int))

代表（1）返回值是 int 型的任何无参数方法执行；（2）任何返回类型且参数为 long 类型的名为 setY 方法的调用；（3）任意 Point 对象的有一个 int 类型 setY 方法的调用，忽略返回类型；（4）对于任何类的构造子的调用，只要该构造子有两个 int 类型的参数。

• 如何组合切点，例如

1．target(Point) && call(int *())
2．call(* *(..)) && (within(Line) || within(Point))
3．within(*) && execution(*.new(int))
4．!this(Point) && call(int *(..))

代表（1）Point 实例上返回类型为 int 的任意无参数方法调用；（2）Line 或 Point 定义及其实例产生的任意方法调用；（3）任意带一个 int 类型参数的构造子调用；（4）任意返回类型为 int 类型的方法调用只要当前执行类实例不是 Point 类型。

• 选择有特定修饰的方法或构造子

1．call(public * *(..))
2．execution(!static * *(..))
3．execution(public !static * *(..))

代表（1）任意公共方法的调用；（2）任意非静态方法的执行；（3）任意公共非静态方法的执行。

• 切点还能够处理接口。例如给定接口

interface MyInterface { … }

切点 call(* MyInterface.*(..))捕捉 MyInterface 定义的方法以及超类型定义的方法调用。

2.3.2．调用和执行

当方法和构造子运行时，它们会被有趣地关联两次——一次是当它们被调用(call)、一次是当它们实际地执行(execution)。

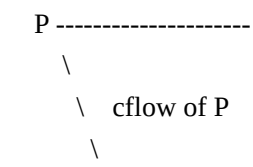
AspectJ 有时同样地暴露 call 和 execution 连接点，允许它们捕捉特定的 call 和 execution 切点。那么它们有什么不同呢？

- call 捕捉调用栈的信息，而 execution 捕捉已经执行的方法；
- call 切点不能捕捉超类的非静态方法，因为它们没有经过动态地分配；

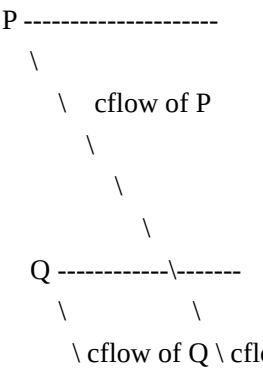
经验告诉表明，如果你要捕捉的连接点已经实际地运行(比如代码跟踪)，那么用 execution；如果你要捕捉一个特定的签名方法用 call。

2.3.3．切点的合成

切点可以使用操作符与（&&）、或（||）和非（!）。这样可以使用简单的原始切点来创建强大功能的切点。当使用原始切点 cflow 和 cflowbelow 进行组合时，可能会有些迷糊。例如，cflow（p）捕捉 p 流程内（包括 P 在内）的所有连接点，可以使用图形表示

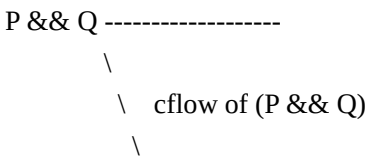


那么 cflow(P) && cflow(Q)捕捉的是什么呢？它捕捉同时处于 P 和 Q 流程中的连接点。



注意 P 和 Q 可能没有任何公共的连接点，但是它们的程序流程中可能有公共的连接点。

cflow(P && Q)又是什么意思呢？它的意思是捕捉 P 和 Q 的控制流的连接点。



如果没有捕捉到 P 和 Q 的连接点，那么在（P&&Q）的程序流程中不可能有任何连接点。

下面代码表明上述意思

```
public class Test {
	public static void main(String[] args) {
		foo();
	}
	static void foo() {
		goo();
	}
	static void goo() {
		System.out.println("hi");
	}
}

aspect A	{
	pointcut fooPC(): execution(void Test.foo());
	pointcut gooPC(): execution(void Test.goo());
	pointcut printPC(): call(void java.io.PrintStream.println(String));

	before(): cflow(fooPC()) && cflow(gooPC()) && printPC() && !within(A) {
		System.out.println("should occur");
	}
```

```
    before(): cflow(fooPC() && gooPC()) && printPC() && !within(A) {
        System.out.println("should not occur");
    }
}
```

2.3.4. 切点的参数

考虑下述代码：

```
pointcut setter(): target(Point) &&
                   (call(void setX(int)) ||
                   call(void setY(int)));
```

这个切点捕捉以 Point 实例为目标的每个 setX(int)或 setY(int)方法的调用。切点名为 setter 并且在左边没有任何参数。空的参数列意味着切点不会暴露任何连接点的上下文信息。但是考虑另一版本的切点定义：

```
pointcut setter(Point p): target(p) &&
                          (call(void setX(int)) ||
                          call(void setY(int)));
```

它的功能与前述的切点相同，但是这里的切点包括一个参数类型为 Point。这就意味着任何使用这个切点的通知都可以访问被切点捕捉连接点中的 Point 实例。

现在看看又一版本的 setter 切点

```
pointcut setter(Point p, int newval): target(p) && args(newval)
&& (call(void setX(int)) || call(void setY(int)));
```

这里切点暴露了一个 Point 对象和一个 int 值。在切点定义的右边，我们发现 Point 对象是目标对象，而且 int 值是被调用方法的参数。

切点参数的使用有很大的伸缩性。最重要的规则是所有的切点参数必须被绑定在每个切点捕捉的连接点上。因此，下面的例子就会参数编译错误。

```
pointcut badPointcut(Point p1, Point p2):
    (target(p1) && call(void setX(int))) ||
    (target(p2) && call(void setY(int)));
```

因为 p1 仅当 setX 调用时被绑定，而 p2 只在 setY 调用时被绑定，但是切点却捕捉所有的这些连接点并且试图同时绑定 p1 和 p2

2.3.5. 例子：HandleLiveness

这个例子包括两个对象类，一个异常类和一个方面。Handle 对象仅仅是代理 Partner 对象的非静态公共方法。方面 HandleLiveness 确保在代理之前 Partner 存在并且可用，否则抛出一个异常。

```
class Handle {
    Partner partner = new Partner();
    public void foo() { partner.foo(); }
    public void bar(int x) { partner.bar(x); }
    public static void main(String[] args) {
        Handle h1 = new Handle();
```

```
        h1.foo();
        h1.bar(2);
    }
}
class Partner {
    boolean isAlive() { return true; }
    void foo() { System.out.println("foo"); }
    void bar(int x) { System.out.println("bar " + x); }
}
aspect HandleLiveness {
    before(Handle handle): target(handle) && call(public * *(..)) {
        if ( handle.partner == null    || !handle.partner.isAlive() ) {
            throw new DeadPartnerException();
        }
    }
}
  class DeadPartnerException extends RuntimeException {}
```

2.4. 通知

通知定义几个方面的实现，以便于在特定的程序运行段执行。这些特定段可以使用命名的切点给出也可以使用匿名切点。下面是一个使用命名切点的通知的例子：

```
pointcut setter(Point p1, int newval): target(p1) && args(newval)
                                                (call(void setX(int) ||
                                                 call(void setY(int)));


before(Point p1, int newval): setter(p1, newval) {
        System.out.println("About to set something in " + p1 +
                                    " to the new value " + newval);
}
```

使用匿名切点也可以实现同样的通知，如下

```
before(Point p1, int newval): target(p1) && args(newval)
                                            (call(void setX(int)) ||
                                             call(void setY(int))) {
        System.out.println("About to set something in " + p1 +
                                    " to the new value " + newval);
}
```

下面是一些不同的通知。

这个 before 通知在匿名切点捕捉到连接点之前运行：

```
before(Point p, int x): target(p) && args(x) && call(void setX(int)) {
        if (!p.assertX(x)) return;
}
```

这个 after 通知在匿名切点捕捉到各个连接点之后运行，无论程序是正常返回还是抛出异常：

```
    after(Point p, int x): target(p) && args(x) && call(void setX(int)) {
        if (!p.assertX(x)) throw new PostConditionViolation();
    }
```

下面的 after 通知在匿名切点捕捉到连接点后且仅仅在程序正常返回的情况下在运行：

```
    after(Point p) returning(int x): target(p) && call(int getX()) {
        System.out.println("Returning int value " + x + " for p = " + p);
    }
```

下面的 after 通知在匿名切点捕捉到连接点后且程序抛出异常的情况下运行：

```
    after() throwing(Exception e): target(Point) && call(void setX(int)) {
        System.out.println(e);
    }
```

around 通知可以取代被捕捉的连接点而执行它自己定义的逻辑，程序原有的逻辑可以通过调用一个特定的方法 proceed 执行：

```
void around(Point p, int x): target(p)
                                && args(x)
                                && call(void setX(int)) {
    if (p.assertX(x)) proceed(p, x);
    p.releaseResources();
}
```

2.5. 类型间声明

方面能够声明属于其它类型的成员（字段、方法和构造子）。这些成员被称为类型间成员。方面也能够声明实现新接口或扩展一个新类的其它类型。这里有一些这样的声明的例子。

下述声明每个 Server 对象有一个名为 disabled 的 boolean 类型字段，其初始值为 false：

```
    private boolean Server.disabled=false;
```

这里声明了一个私有的字段，只有这个方面能够访问这个字段。就算是 Server 对象本身有另一个私有字段 disabled（在 Server 内或其它方面里定义）也不会产生名字冲突，因为对于 disabled 引用没有二义性。

下面代码段声明每个 Point 对象有一个名为 getX 的方法，它返回各自 x 变量的值：

```
    public int Point.getX(){ return this.x;}
```

在方法内部，this 就是当前执行的 Point 对象。因为方法声明为公共方法，所以任何代码都能调用它，但是如果有另一个 Point.getX()声明，那么就会产生编译期的冲突。

下面的公有声明定义 Point 对象的一个构造函数，它有两个整型参数：

```
    public Point.new(int x,int y){ this.x=x;this.y=y;   }
```

下面是一个公有字段声明：

```
    public int Point.x=0;
```

它为 Point 对象声明了一个 x 公有字段并初始化为 0，因为字段是公有的，在任何地方都可以访问它，所以如果还有另一个 x 字段，则会产生冲突。

下面声明了 Point 类实现的接口 Comparable

```
declare parents : Point implement Comparable;
```

当然，除非 Point 类实现了接口的方法，否则会有错误。

下面则为 Point 类声明了其扩展的类 GeometricObject

```
declare parents : Point extends GeometricObject;
```

一个方面可以有多个类型间声明。例如下面的声明。

```
Public String Point.name;
Public void Point.setName(String name){    this.name=name;    }
```

类型间成员仅仅能够有一个目标类型，但是通常你可能想要在多个类型上声明相同的成员。这可以通过联合使用类型间成员和一个私有接口实现。

```
aspect A {
   private interface HasName {}
   declare parents: (Point || Line || Square) implements HasName;

   private String HasName.name;
   public   String HasName.getName()   { return name; }
}
```

这里声明了一个 HasName 接口，并声明 Point、Line 或 Square 都实现这个接口。而且为接口声明了私有字段 name 和公有方法 getName。

2.5.1. 类型间变量的作用域

AspectJ 允许私有、包保护（缺省）以及公有的类型间声明。私有意味着与 aspect 有私有关系，而与目标对象无关（即目标对象不知道变量或方法的存在）。因此，如果一个方面作出一个字段的私有类型间声明

```
private int Foo.x;
```

那么方面中的代码可以访问 Foo 的 x 字段，其它类或方面都不行。类似地，如果一个方面作出一个包保护类型间声明

```
int Foo.x;
```

那么在包中的任何代码都可以访问它，包外的代码无权访问。

2.5.2. 例子：PointAssertions

这个例子包括一个类和一个方面。方面为 Point 声明了私有的 assertion 方法 assertX 和 assertY。利用这两个断言方法方面对 setX 和 setY 方法的调用提供了保护。断言方法声明为私有是因为没有其它地方需要用到它们，只有方面内部可以使用这些方法。

```
class Point   {
    int x, y;
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
```

```
    public static void main(String[] args) {
        Point p = new Point();
        p.setX(3);
        p.setY(333);//非法的 Y 值
    }
}

aspect PointAssertions {
    //如果 X 或 Y 的值不在 0 到 100 之间，则视为非法。
    private boolean Point.assertX(int x) {
        return (x <= 100 && x >= 0);
    }
    private boolean Point.assertY(int y) {
        return (y <= 100 && y >= 0);
    }
    before(Point p, int x): target(p) && args(x) && call(void setX(int)) {
        if (!p.assertX(x)) {     //若非法输入 X，则输出提示信息
            System.out.println("Illegal value for x"); return;
        }
    }
    before(Point p, int y): target(p) && args(y) && call(void setY(int)) {
        if (!p.assertY(y)) {     //若非法输入 Y，则输出提示信息
            System.out.println("Illegal value for y"); return;
        }
    }
}
```

2.6. thisJoinPoint 变量

AspectJ 提供了一个特别的引用变量，thisJoinPoint，它包含了当前连接点处的相关信息并可以被通知使用。ThisJoinPoint 变量仅仅可以在通知环境中被使用，就象 this 仅能用于非静态方法和构造函数环境中一样。在通知中，thisJoinPoint 是 org.aspectj.lang.JoinPoint 类型的变量。使用它的一个简单作用是直接输出它。和其它 Java 对象一样，thisJoinPoint 有一个 toString()方法简化了格式化的输出：

```
class TraceNonStaticMethods {
    before(Point p): target(p) && call(* *(..)) {
        System.out.println("Entering " + thisJoinPoint + " in " + p);
    }
}
```

thisJoinPoint 可以被用来访问静态和动态信息，比如说参数等：

```
thisJoinPoint.getArgs();
```

另外，它持有一个包括所有静态信息的对象，可以通过以下方法得到该对象的引用：

```
thisJoinPoint.getStaticPart();
```

如果你仅需要关于连接点处的静态信息，你可能访问连接点的静态部分直接使用变量

thisJoinPointStaticPart。使用它将避免运行时直接使用 thisJoinPoint 创建连接点对象。

通常情况下

```
thisJoinPointStaticPart == thisJoinPoint.getStaticPart()
 thisJoinPoint.getKind() == thisJoinPointStaticPart.getKind()
thisJoinPoint.getSignature() == thisJoinPointStaticPart.getSignature()
thisJoinPoint.getSourceLocation() == thisJoinPointStaticPart.getSourceLocation()
```

还有一个相关变量：thisEnclosingJoinPointStaticPart。它与 thisJoinPointStaticPart 类似，使用它可以打印调用者的位置，例如

```
before() : execution (* *(..)) {
   System.err.println(thisEnclosingJoinPointStaticPart.getSourceLocation())
}
```

3. 例子

3.1. 概述

这一章由一些使用 AspectJ 的例子组成。它们的内容组成如下：

- 技巧：怎样使用 AspectJ 更多的特性
- 开发：怎样把 AspectJ 用于一个项目的开发阶段
- 产品：怎样用 AspectJ 为一个已经存在的应用添加功能
- 复用：复用方面和切点

3.2. 获得、编译以及运行例子

这些例子的源代码是AspectJ发行包的一部分，你可以在 http://eclipse.org/aspectj 处下载到。

大部分的例子可以直接编译，在 AspectJ 安装路径下面有一个 examples 目录，并且在每一个例子的子目录中都有一个.lst 文件。用带有选项 arglist 的 ajc 命令编译这些例子即可。比如下面的例子：

```
ajc -argfile telecom/billing.lst
```

为了运行这些例子，你的 classpath 设置必须包括 AspectJ 的运行时 aspectjrt.jar 库。你可以在 classpath 环境变量或者在 Java 解释器的-classpath 选项中包括这个库。比如：

```
(In Unix use a : in the CLASSPATH)
java -classpath ".:InstallDir/lib/aspectjrt.jar" telecom.billingSimulation
(In Windows use a ; in the CLASSPATH)
java -classpath ".;InstallDir/lib/aspectjrt.jar" telecom.billingSimulation
```

3.3. 基本技巧

这个部份描述两个使用 AspectJ 的基本技巧，每一个都从两种基本的捕捉横切关注点的

方式进行论述：一个是动态连接点和通知，一个是静态引用。通知改变一个应用的行为，静态引用同时改变一个应用的行为和它的结构。

在第一个例子(Join Points and thisjoinPoint)中，我们聚集和利用横切关注点的相关信息去触发一些通知。在第二个部份(Roles and Views)中，我们用横切关注点去观察一个已有类的行为。

3.3.1. 连接点集和 thisJoinPoint 变量

这些代码在 AspectJ 安装路径下的 examples/tjp 目录。

连接点是程序执行过程中的一些点——当这些点到达时的执行环境。连接点集被切点捕捉。当一个程序到达(执行)一个连接点时，通知将加进(或代替)这个连接点自身。

当用切点捕捉被命名的单一种类的连接点集时，通知会正确地知道连接点的联合使用。切点可以恰当地暴露连接点的环境。因为切点捕捉连接点处确定的方法调用，所以我们能直接地取得目标值和方法调用的参数值。比如：

```
before(Point p, int x): target(p)
                        && args(x)
                        && call(void setX(int)) {
    if (!p.assertX(x)) {
        System.out.println("Illegal value for x"); return;
    }
}
```

但，有些时候连接点的构造不是十分完整的，比如，假设有一个复杂的应用正在调试，而且我们想要跟踪一些类执行时的任何方法调用，可以使用下面的切点：

```
pointcut execsInProblemClass(): within(ProblemClass)
                                && execution(* *(..));
```

这个切点捕捉在类 ProblemClass 中被定义的每个方法的连接点的执行。由于通知在每个连接点被切点捕捉到时执行，所以我们在连接点到达时作些适当的处理。

通过特殊的 thisJoinPoint 变量我们匹配一些可利用的信息到通知之中，thisJoinPoint 变量是 org.aspectj.lang.JoinPoint 类型，通过这个对象我们可以存取如下的一些的信息：

- 匹配连接点的种类
- 用连接点关联特定的源代码
- 当前连接点的字符串表示法
- 用连接点关联签名成员(比如签名方法)
- 当前执行对象
- 目标对象
- 封装连接点的静态信息，当然这也可以通过变量 thisJoinPointStaticPart 来达到同样的目的

类 tjp.Demo 在 tjp/Demo.java 位置，这个类定义了有不同参数和不同返回类型的两个方法 foo、bar。go 方法从 main 方法中调用：

```
public class Demo {
    static Demo d;

    public static void main(String[] args){
        new Demo().go();
```

```
    }

    void go(){
        d = new Demo();
        d.foo(1,d);
        System.out.println(d.bar(new Integer(3)));
    }

    void foo(int i, Object o){
        System.out.println("Demo.foo(" + i + ", " + o + ")\n");
    }

    String bar (Integer j){
        System.out.println("Demo.bar(" + j + ")\n");
        return "Demo.bar(" + j    + ")";
    }
}
```

方面 GetInfo 和 Demo 类配套。这个方面用 around 型通知截取 Demo 类中 foo 和 bar 方法的执行，并且在控制台上打印存储在 thisJoinPoint 变量里面的信息：

```
aspect GetInfo {

    static final void println(String s){ System.out.println(s); }

    pointcut goCut(): cflow(this(Demo) && execution(void go()));

    pointcut demoExecs(): within(Demo) && execution(* *(..));

    Object around(): demoExecs() && !execution(* go()) && goCut() {
        println("Intercepted message: " +
            thisJoinPointStaticPart.getSignature().getName());
        println("in class: " +
            thisJoinPointStaticPart.getSignature().getDeclaringType().getName());
        printParameters(thisJoinPoint);
        println("Running original method: \n" );
        Object result = proceed();
        println("    result: " + result );
        return result;
    }

    static private void printParameters(JoinPoint jp) {
        println("Arguments: " );
        Object[] args = jp.getArgs();
        String[] names = ((CodeSignature)jp.getSignature()).getParameterNames();
        Class[] types = ((CodeSignature)jp.getSignature()).getParameterTypes();
```

```
        for (int i = 0; i < args.length; i++) {
            println("    "    + i + ". " + names[i] +
                    " : " +                  types[i].getName() +
                    " = " +                   args[i]);
        }
    }
}
```

在这个方面中，goCut 切点有如下定义：

```
cflow(this(Demo)) && execution(void go())
```

所以，Demo.go 的执行控制流程被截取。由于这个控制流程包括了 go 方法自身，所以在 around 通知中用!execution(* go())排除了 go 方法执行的考虑。

同时我们也看到了方法的名字和方法的定义类的变量是 thisJoinPoint 或 thisJoinPointStaticPart 对象调用 getSignature()时返回的 org.aspectj.lang.Signature 类的一部分；参数的类型和名字可以通过 org.aspectj.lang.reflect.CodeSignature 类用连接点进行关联。

3.3.2. 任务和观察

这些代码在 AspectJ 安装路径下的 examples/introduction 目录。

通知、类型间声明，声明的是一个方面的成员。这些成员产生了对其它类的影响就好像它们在其它类一样。不像通知，类型间声明不仅影响一个应用的行为，而且也影响一个应用的类与类之间的关系。

至关重要的一点是，类型间声明可以修改一个应用的可用组件，不仅如此它还可以影响一个应用的类结构。

在 AspectJ 中，可以有如下的类型间声明：

- 域
- 方法
- 构造子

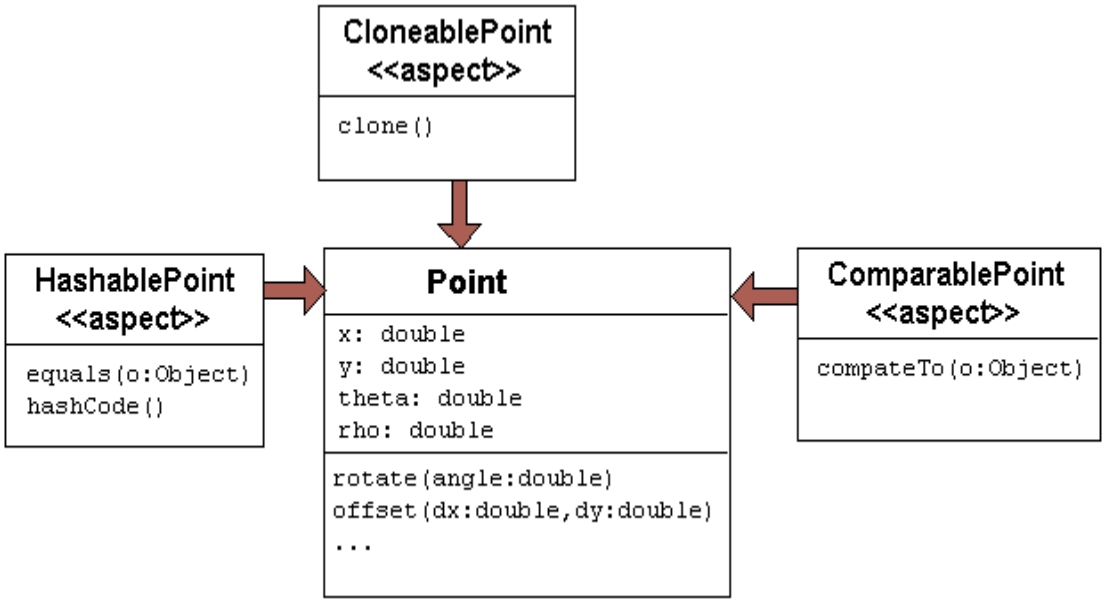
也可以声明如下的目标对象类型：

- 实现新的接口
- 扩展新的类

下面的例子提供了用类型间声明去封装一个类的任务和观察的三个方面。我们的方面将处理 Point 对象——Point 对象实现直角和极坐标系。我们的类型间声明将使 Point 对象有旋转、克隆、散列和比较功能。在不修改代码的情况下这些便利是 AspectJ 为我们提供的功能。

Point 类定义了在直角和极坐标系中的几何点，并加上了一些简单的操作来重新部署这些几何点。一些操作使得极坐标系中的点从直角坐标修改而来，同时也包括了相反的一些操作。在这个类中，实现了两个坐标系之间最小数的转换，所有的属性并不都存储在 Point 对象中，对这些点的存储、比较、克隆和制作这些点的散列值要给出一个标准表示。所以这些方面，虽然简单，但不全面。

下面是这些方面交互作用于 Point 对象的一个 UML 图：



下面的 CloneablePoint 方面主要是为了实现 Cloneable 接口。为此它用语法形式：

declare parents:

声明了：

Point implements Cloneable;

并且还声明了 Point 对象中必须实现的 clone()方法。在 Java 中所有对象度都从 Object 对象那里继承 clone()方法，但一个对象本身还不可能实现克隆除非它同时实现 Cloneable 接口。另外，实现 Cloneable 接口的类还需要重置 clone()方法。这里我们想要达到的目的是在修改 Point 对象的坐标系统之前实际地克隆 Point 对象。所以重新写了一个新方法来重置 Object.clone()方法从而达到我们的目的。

在这个方面中我们也写了一个 main()方法以有助于测试：

```
public aspect CloneablePoint {

    declare parents: Point implements Cloneable;

    public Object Point.clone() throws CloneNotSupportedException {
        // we choose to bring all fields up to date before cloning.
        makeRectangular();
        makePolar();
        return super.clone();
    }

    public static void main(String[] args){
        Point p1 = new Point();
        Point p2 = null;

        p1.setPolar(Math.PI, 1.0);
        try {
            p2 = (Point)p1.clone();
        } catch (CloneNotSupportedException e) {}
```

```
        System.out.println("p1 =" + p1 );
        System.out.println("p2 =" + p2 );

        p1.rotate(Math.PI / -2);
        System.out.println("p1 =" + p1 );
        System.out.println("p2 =" + p2 );
    }
}
```

ComparablePoint 方面是为了 Point 对象实现 Comparable 接口。接口定义了一个compareTo()方法，该方法要求定义一个类的对象间的自然顺序关系并且实现它。

同样采用语法形式：

declare parents:

声明了：

Point implements Comparable;

并且也声明了一个 public 方法 Point.compareTo()，这个方法的意思是：如果 p1、p2 都是 Point 对象，看那一个更靠近坐标系的原点。

同样在这个方面中我们也写了一个 main()方法以有助于测试：

```
public aspect ComparablePoint {

    declare parents: Point implements Comparable;

    public int Point.compareTo(Object o) {
        return (int) (this.getRho() - ((Point)o).getRho());
    }

    public static void main(String[] args){
        Point p1 = new Point();
        Point p2 = new Point();

        System.out.println("p1 =?= p2 :" + p1.compareTo(p2));

        p1.setRectangular(2,5);
        p2.setRectangular(2,5);
        System.out.println("p1 =?= p2 :" + p1.compareTo(p2));

        p2.setRectangular(3,6);
        System.out.println("p1 =?= p2 :" + p1.compareTo(p2));

        p1.setPolar(Math.PI, 4);
        p2.setPolar(Math.PI, 4);
        System.out.println("p1 =?= p2 :" + p1.compareTo(p2));

        p1.rotate(Math.PI / 4.0);
```

```
        System.out.println("p1 =?= p2 :" + p1.compareTo(p2));

        p1.offset(1,1);
        System.out.println("p1 =?= p2 :" + p1.compareTo(p2));
    }
}
```

方面HashablePoint主要为了制作Point对象的散列值而重置对象的equals()和hashCode()方法。

Java 中一个对象的 hashCode()方法返回唯一的整数，它适合于作 HashTable 的键(key)值。不同的实现允许返回不同的整数值。但为了独特的对象必须返回独特的整数值。相同的整数值说明对象在实施 equals()操作后相等。由于缺省定义的 Object.equals()方法返回 true 时说明两个对象时同一个对象，因此为了类型 Point 的 equals()和 hashCode()方法能正常工作需要重新定义它们。比如：当两个对象有相同的 x 和 y 值，或者有相同的 rho 和 thrta 值时，我们说它们相等。为此我们重置 Point 类的 equals()和 hashCode()方法。

HashablePoint 方面声明了 Point 对象的 equals()和 hashCode()方法，用 Point 对象的直角坐标产生一个散列值并测试对象的等同性。x 和 y 坐标用适当的 get 方法得到，同时保证了直角坐标在被修改以前返回其值。

同样在这个方面中我们也写了一个 main()方法以有助于测试：

```
public aspect HashablePoint {

    public int Point.hashCode() {
        return (int) (getX() + getY() % Integer.MAX_VALUE);
    }

    public boolean Point.equals(Object o) {
        if (o == this) { return true; }
        if (!(o instanceof Point)) { return false; }
        Point other = (Point)o;
        return (getX() == other.getX()) && (getY() == other.getY());
    }

    public static void main(String[] args) {
        Hashtable h = new Hashtable();
        Point p1 = new Point();

        p1.setRectangular(10, 10);
        Point p2 = new Point();

        p2.setRectangular(10, 10);

        System.out.println("p1 = " + p1);
        System.out.println("p2 = " + p2);
        System.out.println("p1.hashCode() = " + p1.hashCode());
```

```
        System.out.println("p2.hashCode() = " + p2.hashCode());

        h.put(p1, "P1");
        System.out.println("Got: " + h.get(p2));
    }
}
```

3.4. 用于开发的方面

3.4.1. 用方面进行代码跟踪

这部分的源代码位于 AspectJ 安装路径下的 examples/tracing 目录内。

编写一个类来跟踪功能性问题比较容易，比如：功能的结合、跟踪的闭合标志、输出流的选择、格式化输出——这些元素在类中都是已知的。但如果需要用代码跟踪一个程序的执行，问题就变得很复杂了。

有经验的开发者会发现在开发阶段的代码跟踪经常要在方法体执行的前后插入一些重写的临时性的脚本程序，而这些临时性的脚本程序注定要使整个系统变得越来越慢，而且对于一个大的系统而言，这些脚本常常相互作用致使一些无法忍受的副作用发生。

使用 AspectJ 能够以较少的临时性的脚本程序的方式跟踪代码，用关注点去横切一个完整的系统并在一个方面中对它们进行封装。另外这些方面可以独立于系统的情况下进行这些工作。因此这些跟踪代码的方面可以在系统中进行“插/拔”而不影响系统的基本功能。

下面的这个应用的例子由四个简单的类组成，这个应用是关于图形计算的。

```
public abstract class TwoDShape {
    protected double x, y;
    protected TwoDShape(double x, double y) {
        this.x = x; this.y = y;
    }
    public double getX() { return x; }
    public double getY() { return y; }
    public double distance(TwoDShape s) {
        double dx = Math.abs(s.getX() - x);
        double dy = Math.abs(s.getY() - y);
        return Math.sqrt(dx*dx + dy*dy);
    }
    public abstract double perimeter();
    public abstract double area();
    public String toString() {
        return (" @ (" + String.valueOf(x) + ", " + String.valueOf(y) + ") ");
    }
}
```

TwoDShape 类有两个子类：Square 和 Circle

```
public class Circle extends TwoDShape {
    protected double r;
```

```
    public Circle(double x, double y, double r) {
        super(x, y); this.r = r;
    }
    public Circle(double x, double y) { this(    x,      y, 1.0); }
    public Circle(double r)                    { this(0.0, 0.0,      r); }
    public Circle()                              { this(0.0, 0.0, 1.0); }
    public double perimeter() {
        return 2 * Math.PI * r;
    }
    public double area() {
        return Math.PI * r*r;
    }
    public String toString() {
        return ("Circle radius = " + String.valueOf(r) + super.toString());
    }
}

public class Square extends TwoDShape {
    protected double s;       // side
    public Square(double x, double y, double s) {
        super(x, y); this.s = s;
    }
    public Square(double x, double y) { this(    x,      y, 1.0); }
    public Square(double s)                    { this(0.0, 0.0,      s); }
    public Square()                              { this(0.0, 0.0, 1.0); }
    public double perimeter() {
        return 4 * s;
    }
    public double area() {
        return s*s;
    }
    public String toString() {
        return ("Square side = " + String.valueOf(s) + super.toString());
    }
}
```

除了标准的 Java 编译器你也可以用 AspectJ 编译器。如果你已经装载了 AspectJ，那么改变目录到 examples，敲入：

```
ajc -argfile tracing/notrace.lst
```

以下面的方式运行程序：

```
java tracing.ExampleMain
```

我们不需要设置任何的 classpath 环境变量，因为上面的代码是纯 Java 的代码。程序有如下的输出：

```
c1.perimeter() = 12.566370614359172
c1.area() = 12.566370614359172
```

```
s1.perimeter() = 4.0
s1.area() = 1.0
c2.distance(c1) = 4.242640687119285
s1.distance(c1) = 2.23606797749979
s1.toString(): Square side = 1.0 @ (1.0, 2.0)
```

版本 1

为了在一个应用之中插入代码跟踪，我们不用方面而是写一个 Trace 类：

```
public class Trace {
    public static int TRACELEVEL = 0;
    public static void initStream(PrintStream s) {...}
    public static void traceEntry(String str) {...}
    public static void traceExit(String str) {...}
}
```

如果不用 AspectJ，想要达到我们的目的，我们要在所有的方法和构造子中插入 traceEntry 和 traceExit 调用并且初始化 TRACELEVEL 和流 PrintStream，这将有 40 个左右的方法调用，并且你要保证没有遗漏。为了改善这种局面我们可以可靠地考虑如下的一个方面：

```
aspect TraceMyClasses {
    pointcut myClass(): within(TwoDShape) || within(Circle) || within(Square);
    pointcut myConstructor(): myClass() && execution(new(..));
    pointcut myMethod(): myClass() && execution(* *(..));

    before (): myConstructor() {
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myConstructor() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
    }

    before (): myMethod() {
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myMethod() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
    }
}
```

这个方面可以在适当的时候对调用进行跟踪。这个方面跟踪图形层次中每个人方法和构造子在进入调用前以及退出后的静态环境。它跟踪每个以签名方法执行作为连接点的每个 before 和 after 通知。因为签名信息是静态信息，所以我们能通过 thisJoinPointStaticPart 变量获得它。

用以下命令行编译我们的方面：

```
ajc -argfile tracing/tracev1.lst
```

运行 tracing.version1.TraceMyClasses 类，将有如下的输出：

```
--> tracing.TwoDShape(double, double)
   <-- tracing.TwoDShape(double, double)
   --> tracing.Circle(double, double, double)
   <-- tracing.Circle(double, double, double)
   --> tracing.TwoDShape(double, double)
   <-- tracing.TwoDShape(double, double)
   --> tracing.Circle(double, double, double)
   <-- tracing.Circle(double, double, double)
   --> tracing.Circle(double)
   <-- tracing.Circle(double)
   --> tracing.TwoDShape(double, double)
   <-- tracing.TwoDShape(double, double)
   --> tracing.Square(double, double, double)
   <-- tracing.Square(double, double, double)
   --> tracing.Square(double, double)
   <-- tracing.Square(double, double)
   --> double tracing.Circle.perimeter()
   <-- double tracing.Circle.perimeter()
c1.perimeter() = 12.566370614359172
   --> double tracing.Circle.area()
   <-- double tracing.Circle.area()
c1.area() = 12.566370614359172
   --> double tracing.Square.perimeter()
   <-- double tracing.Square.perimeter()
s1.perimeter() = 4.0
   --> double tracing.Square.area()
   <-- double tracing.Square.area()
s1.area() = 1.0
   --> double tracing.TwoDShape.distance(TwoDShape)
      --> double tracing.TwoDShape.getX()
      <-- double tracing.TwoDShape.getX()
      --> double tracing.TwoDShape.getY()
      <-- double tracing.TwoDShape.getY()
   <-- double tracing.TwoDShape.distance(TwoDShape)
c2.distance(c1) = 4.242640687119285
   --> double tracing.TwoDShape.distance(TwoDShape)
      --> double tracing.TwoDShape.getX()
      <-- double tracing.TwoDShape.getX()
      --> double tracing.TwoDShape.getY()
      <-- double tracing.TwoDShape.getY()
   <-- double tracing.TwoDShape.distance(TwoDShape)
s1.distance(c1) = 2.23606797749979
   --> String tracing.Square.toString()
      --> String tracing.TwoDShape.toString()
```

```
    <-- String tracing.TwoDShape.toString()
  <-- String tracing.Square.toString()
s1.toString(): Square side = 1.0 @ (1.0, 2.0)
```

当不把 TraceMyClasses 类提供给 ajc 编译器时，这个方面不会对系统造成任何影响即使你把跟踪代码拿掉后也是这样。

版本 2

可以通过其它方式来完成同样的事情——写一个可以重用的方面，它不仅能够应用于这些类，也可以应用于任何类。其中一种方法就是用TraceMyClasses—version1的横切面支持合并Trace—version1 的跟踪功能。

你可以在 version2/Trace.java 里找到如下的公共接口：

```
abstract aspect Trace {
    public static int TRACELEVEL = 2;
    public static void initStream(PrintStream s) {...}
    protected static void traceEntry(String str) {...}
    protected static void traceExit(String str) {...}
    abstract pointcut myClass();
}
```

为此我们需要定义我们自己的子类 version2/TraceMyClasses.java：

```
public aspect TraceMyClasses extends Trace {
    pointcut myClass(): within(TwoDShape) || within(Circle) || within(Square);
    public static void main(String[] args) {
        Trace.TRACELEVEL = 2;
        Trace.initStream(System.err);
        ExampleMain.main(args);
    }
}
```

注意这里制作了切点——它是父方面抽象切点的具体化。为了运行这个跟踪方面的的版本，到 examples 目录并敲入：

```
ajc -argfile tracing/tracev2.lst
```

这个 tracev2.lst 文件包括 Trace.java 和 TraceMyClasses.java。

运行 tracing.version2.TraceMyClasses 的 main 方法可以得到和 version 1 一样的结果。

完整的跟踪类是：

```
abstract aspect Trace {

    // implementation part

    public static int TRACELEVEL = 2;
    protected static PrintStream stream = System.err;
    protected static int callDepth = 0;

    public static void initStream(PrintStream s) {
        stream = s;
    }
```

```
    protected static void traceEntry(String str) {
        if (TRACELEVEL == 0) return;
        if (TRACELEVEL == 2) callDepth++;
        printEntering(str);
    }
    protected static void traceExit(String str) {
        if (TRACELEVEL == 0) return;
        printExiting(str);
        if (TRACELEVEL == 2) callDepth--;
    }
    private static void printEntering(String str) {
        printIndent();
        stream.println("--> " + str);
    }
    private static void printExiting(String str) {
        printIndent();
        stream.println("<-- " + str);
    }
    private static void printIndent() {
        for (int i = 0; i < callDepth; i++)
            stream.print("    ");
    }

    // protocol part

    abstract pointcut myClass();

    pointcut myConstructor(): myClass() && execution(new(..));
    pointcut myMethod(): myClass() && execution(* *(..));

    before(): myConstructor() {
        traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myConstructor() {
        traceExit("" + thisJoinPointStaticPart.getSignature());
    }

    before(): myMethod() {
        traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myMethod() {
        traceExit("" + thisJoinPointStaticPart.getSignature());
    }
}
```

这个版本不同于版本 1 的在几个微妙的地方。首先要注意的是 Trace 类用跟踪调用的横切面合并跟踪的功能部分。在版本 1 中跟踪支持和它的横切面用法是相分离的，在这个版本中这两个部分是合并的。这就是为什么可以将这个类描述为“跟踪信息是在构造方法和普通方法之前和之后被打印”，这也是我们想要的。调用的定位在这个版本中是被方面自己确定的，这也局部排除了放错地方的机会。

在这个类中不需要为 traceEntry 和 traceExit 提供 public 修饰，正如你看到的它们是 protected 的——它们是通知的内在实现细节。

这个方面的关键块是抽象的切点类——它充当构造方法和普通方法切点的基础。即使是抽象类而没有具体的类被提及，我们也能在它上面放通知，基于它的切点也一样，这话的意思是“我们不知道正确的切点集是什么，但当我们做时我们应当知道做什么”。在一些情况下，抽象切点类似于抽象方法。抽象方法不提供其实现，但你知道在其子类中将实现它，所以能调用这些方法。

3.5. 用于产品的方面

3.5.1. Bean 方面

这部分的源代码位于 AspectJ 安装路径下的 InstallDir/examples/bean 目录内。

这个例子使用属性绑定进入到 Java Bean 内部制作 Point 对象。

Java Bean是可复用的软件组件——它能真实地被用于开发工具之中。一个Bean要求具备地条件是很少的。Bean必须定义一个无参数的构造方法和必须是Serializable 或Externalizable其中之一。任何一个对象的属性已经被处理为一个Bean属性——属性用set和get方法。getproperty 和setproperty方法中property是Bean类中地一个域（属性）。有一些Bean属性，比如属性绑定——值的改变将会引起已经注册地监听器得到这些改变的通知。保持属性绑定于一个监听器列表中并创建和在对象中用方法分配事件来改变这些属性的值，比如setproperty方法。

假如 Point 类是一个简单的直角坐标系的点集的表示。Point 并不知道有关 bean 的任何事：为了 x 和 y 属性有 set 方法，但它们不是事件驱动的并且这个类也不是序列化的。Bound 是一个方面——它制作 Point 的序列化类并且制作 get 和 set 方法以支持属性绑定协议。

下面是一个没有多大用处的采用 getters 和 setters 制作的一个简单类，它有一个简单的向量偏移方法：

```
class Point {

  protected int x = 0;
  protected int y = 0;

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }
```

```
    public void setRectangular(int newX, int newY) {
        setX(newX);
        setY(newY);
    }

    public void setX(int newX) {
        x = newX;
    }

    public void setY(int newY) {
        y = newY;
    }

    public void offset(int deltaX, int deltaY) {
        setRectangular(x + deltaX, y + deltaY);
    }

    public String toString() {
        return "(" + getX() + ", " + getY() + ")" ;
    }
}
```

下面为了 Point 类进行方面的改造（BoundPoint 方面）。首先要做的事情是私有化地定义每个 Point 类有一个 support 属性并且在 PropertyChangeSupport 对象的实例中保持其引用：

```
private PropertyChangeSupport Point.support = new PropertyChangeSupport(this);
```

这里必须提供如下的支持：属性的改变必须用对 bean 的引用来构造对象，所以它的初始化通过 Point 的实例 this 进行。既然 support 属性在方面中是私有化的，那么只有在这个方面中能提到它。

为了属性改变的事件这个方面也定义一些 Point 的注册和管理监听方法。它们被委托给属性改变的支持对象。

```
    public void Point.addPropertyChangeListener(PropertyChangeListener listener){
        support.addPropertyChangeListener(listener);
    }
    public void Point.addPropertyChangeListener(String propertyName,
                                                PropertyChangeListener listener){

        support.addPropertyChangeListener(propertyName, listener);
    }
    public void Point.removePropertyChangeListener(String propertyName,
                                                   PropertyChangeListener listener) {
        support.removePropertyChangeListener(propertyName, listener);
    }
    public void Point.removePropertyChangeListener(PropertyChangeListener listener) {
        support.removePropertyChangeListener(listener);
```

```
    }
    public void Point.hasListeners(String propertyName) {
        support.hasListeners(propertyName);
    }
```

Point 实现接口 Serializable。

```
pointcut setter(Point p): call(void Point.set*(*)) && target(p);

    void around(Point p): setter(p) {
            String propertyName =
        thisJoinPointStaticPart.getSignature().getName().substring("set".length());
            int oldX = p.getX();
            int oldY = p.getY();
            proceed(p);
            if (propertyName.equals("X")){
        firePropertyChange(p, propertyName, oldX, p.getX());
            } else {
        firePropertyChange(p, propertyName, oldY, p.getY());
            }
    }

    void firePropertyChange(Point p,
                                    String property,
                                    double oldval,
                                    double newval) {
            p.support.firePropertyChange(property,
                                        new Double(oldval),
                                        new Double(newval));
    }
}
```

接下来我们测试这个程序。测试程序象一个属性值改变监听器一样登记它自己——监听对象的建立并因而执行那个 point 调用 set 和 offset 方法的简单操作。然后它序列化 point 对象并写入文件，在稍后读它。这个存储和恢复的过程导致一个新的 point 对象被建立：

```
    class Demo implements PropertyChangeListener {

        static final String fileName = "test.tmp";

        public void propertyChange(PropertyChangeEvent e){
            System.out.println("Property " + e.getPropertyName() + " changed from " +
                e.getOldValue() + " to " + e.getNewValue() );
        }

        public static void main(String[] args){
            Point p1 = new Point();
            p1.addPropertyChangeListener(new Demo());
```

```
        System.out.println("p1 =" + p1);
        p1.setRectangular(5,2);
        System.out.println("p1 =" + p1);
        p1.setX( 6 );
        p1.setY( 3 );
        System.out.println("p1 =" + p1);
        p1.offset(6,4);
        System.out.println("p1 =" + p1);
        save(p1, fileName);
        Point p2 = (Point) restore(fileName);
        System.out.println("Had: " + p1);
        System.out.println("Got: " + p2);
        }
    ...
}
```

编译和运行这个例子时，在 examples 目录下敲入：

```
ajc -argfile bean/files.lst
java bean.Demo
```

3.5.2. Subject-Observer 协议

这部分的源代码位于 AspectJ 安装路径下的 InstallDir/examples/ observer 目录内。

这个演示例子说明如何用方面进行 Subject/Observer 方式设计。

这个演示由下列内容组成：一个可以着色的标签对象——它有一种颜色（通过一个颜色集的循环），一个通过循环的数字记录，一个按钮——当点击它时对它们进行记录。

有两个种类的对象，我们能增强 Subject/Observer 的关系，点击按钮观察着色标签，也就是说着色标签是 Observers 而按钮是 Subjects。

这个例子有计划地实现 Subject/Observer 设计模式。例子的其它部分解释说明一些类和例子的方面，并且告诉你如何运行它。

这个协议的通用部分是接口：Subject、Observer 和抽象的方面 SubjectObserverProtocol。Subject 接口包括了添加、移除 Observer 对象的方法，并且也有一个获得其（Observer）状态的方法：

```
interface Subject {
        void addObserver(Observer obs);
        void removeObserver(Observer obs);
        Vector getObservers();
        Object getData();
}
```

Observer 接口也很简单，有 Subject 对象的 set 和 get 方法，并有一个当 Subject 对象获得修改时被调用的 update 方法。

```
interface Observer {
        void setSubject(Subject s);
```

```
        Subject getSubject();
        void update();
}
```

SubjectObserverProtocol 方面在其中容纳了这个协议的通用部分——当 Subject 对象的一些状态改变时，如何为 Observer 对象的 update 方法点火。

```
abstract aspect SubjectObserverProtocol {

        abstract pointcut stateChanges(Subject s);

        after(Subject s): stateChanges(s) {
            for (int i = 0; i < s.getObservers().size(); i++) {
                ((Observer)s.getObservers().elementAt(i)).update();
            }
        }

        private Vector Subject.observers = new Vector();
        public void     Subject.addObserver(Observer obs) {
            observers.addElement(obs);
            obs.setSubject(this);
        }
        public void     Subject.removeObserver(Observer obs) {
            observers.removeElement(obs);
            obs.setSubject(null);
        }
        public Vector Subject.getObservers() { return observers; }

        private Subject Observer.subject = null;
        public void         Observer.setSubject(Subject s) { subject = s; }
        public Subject    Observer.getSubject() { return subject; }

  }
```

注意这个方面做了三件事情：定义了一个抽象的切点——扩展的方面可以重置它；定义了一个后通知，在切点的连接点后运行；定义了一个类型间属性和一个类型间方法，因此每个 Observer 对象在其上拥有 Subject 对象。

Button 对象扩展 java.awt.Button 并且它正是了当按钮被按下时 click()方法被调用。

```
  class Button extends java.awt.Button {

        static final Color    defaultBackgroundColor = Color.gray;
        static final Color    defaultForegroundColor = Color.black;
        static final String defaultText = "cycle color";

        Button(Display display) {
            super();
            setLabel(defaultText);
```

```
            setBackground(defaultBackgroundColor);
            setForeground(defaultForegroundColor);
            addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                        Button.this.click();
                    }
                });
            display.addToFrame(this);
        }

        public void click() {}

    }
```

注意这个对象只知道 Subject 对象的存在。

ColorLabel 对象支持 colorCycle()方法，再者它们只知道 Observer 对象的存在。

```
    class ColorLabel extends Label {

        ColorLabel(Display display) {
            super();
            display.addToFrame(this);
        }

        final static Color[] colors = {Color.red, Color.blue,
                                       Color.green, Color.magenta};
        private int colorIndex = 0;
        private int cycleCount = 0;
        void colorCycle() {
            cycleCount++;
            colorIndex = (colorIndex + 1) % colors.length;
            setBackground(colors[colorIndex]);
            setText("" + cycleCount);
        }
    }
```

最后 SubjectObserverProtocolImpl 实现 subject/observer 协议，用 Button 对象当作 Subject 对象，ColoeLabel 对象当作 Observer 对象：

```
package observer;

import java.util.Vector;

aspect SubjectObserverProtocolImpl extends SubjectObserverProtocol {

    declare parents: Button implements Subject;
    public Object Button.getData() { return this; }
```

```
    declare parents: ColorLabel implements Observer;
    public void       ColorLabel.update() {
        colorCycle();
    }

    pointcut stateChanges(Subject s):
        target(s) &&
        call(void Button.click());

}
```

Button 和 ColorLabel 实现了适当的接口，实现了接口定义必须实现的方法，并且也提供了抽象切点 stateChanges 的具体定义。现在每次 Button 按钮按下，ColorLabel 会观察到 button 的颜色循环。

Demo 时演示例子开始的顶级类，它示例了两个按钮、三个观察者和连同它们一起的 subjects 和 observers。

敲入下面命令行运行改程序：

```
ajc -argfile observer/files.lst
java observer.Demo
```

3.5.3. 一个简单的电信仿真

这部分的源代码位于 AspectJ 安装路径下的 InstallDir/examples/ telecom 目录内。

这个例子阐明了一些用方面依靠连接点来编码的方式。它由一个简单的电话模型组成：电话连接的计时和收费（开单）的功能时用方面来添加的。这里开单的功能是建立在计时功能基础之上的。

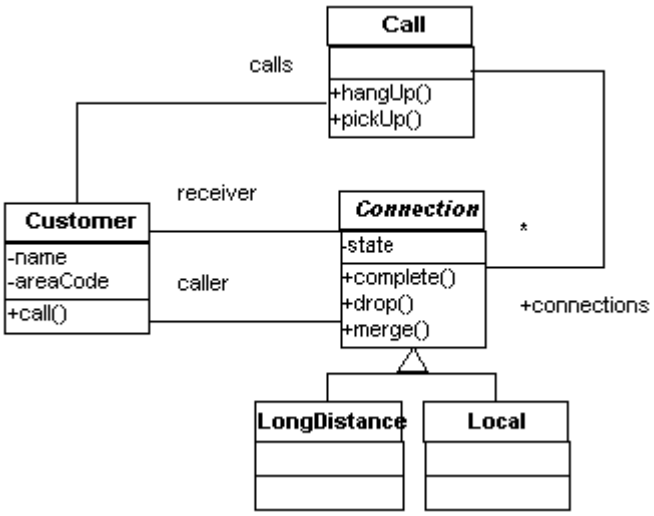
这些例子应用于一种客户使用并能接受的能把本地和长途电话合并和挂断的简单模拟的电话系统。 这个应用体系有三个层次：

- 基本对象为模仿消费者提供基本功能——呼叫和连接（正常的呼叫有一个连接，会议的呼叫有多个连接）
- 记时功能是有关定时的连接并保持每个消费者的连接时间合计，方面为每个客户添加一个定时器，并且管理每个消费者的时间合计
- 开单是有关收费的，方面用于每个收费客户的连接和终止的计算，添加适当的收费发票。builling 方面建立在 timing 方面的基础之上，它有一个切点定义在 Timing 中，并且用这个定时器进行连接

系统有三个配置：basic、timing 和 billing。程序为了这三个配置而使用类：BasicSimulation, TimingSimulation 和 BillingSimulation。它们共享一个父类 AbstractSimulation 它定义了仿真它自己的方法和模仿共（用去的）用时间的方法。

Basic Object

电话仿真系统包括了下列类：Customer、Call 和 Connection。Connection 有两个具体的子类：Local 和 LongDistance。Customer 有 name 和 areaCode 属性，并且有一个 call 方法，它在呼叫者和接受者之间进行简单的调用，Connection 对象用于连接它们。会议呼叫可以多个客户包括多个连接。在一个时刻内一个客户可以有许多的呼叫。



Customer 对象有方法：call、pickup，hangup 和 merge，其中 merge 管理 Call 对象：

```
public class Customer {

    private String name;
    private int areacode;
    private Vector calls = new Vector();

    protected void removeCall(Call c){
        calls.removeElement(c);
    }

    protected void addCall(Call c){
        calls.addElement(c);
    }

    public Customer(String name, int areacode) {
        this.name = name;
        this.areacode = areacode;
    }

    public String toString() {
        return name + "(" + areacode + ")";
    }

    public int getAreacode(){
        return areacode;
    }

    public boolean localTo(Customer other){
        return areacode == other.areacode;
    }
```

```
        public Call call(Customer receiver) {
            Call call = new Call(this, receiver);
            addCall(call);
            return call;
        }

        public void pickup(Call call) {
            call.pickup();
            addCall(call);
        }

        public void hangup(Call call) {
            call.hangup(this);
            removeCall(call);
        }

        public void merge(Call call1, Call call2){
            call1.merge(call2);
            removeCall(call2);
        }
    }
```

Call 对象用来建立呼叫者和接收者，如果呼叫者和接收者有相同的区域代码，那么 call 能确定用本地连接 Local，相反会用 LongDistance。开始时仅仅只在呼叫者和连接者之间进行连接，但另外的连接也是能被添加的——如果 call 被合并倒电话会议之中去的话。

Connection 模仿客户之间建立连接的物理细节，它有一个简单的状态机（由 PENDING、COMPLETED 和 DROPPED 描述）。信息被打印到控制台，所以你能观察它们。

```
abstract class Connection {

        public static final int PENDING = 0;
        public static final int COMPLETE = 1;
        public static final int DROPPED = 2;

        Customer caller, receiver;
        private int state = PENDING;

        Connection(Customer a, Customer b) {
            this.caller = a;
            this.receiver = b;
        }

        public int getState(){
            return state;
        }
```

```
        public Customer getCaller() { return caller; }

        public Customer getReceiver() { return receiver; }

        void complete() {
            state = COMPLETE;
            System.out.println("connection completed");
        }

        void drop() {
            state = DROPPED;
            System.out.println("connection dropped");
        }

        public boolean connects(Customer c){
            return (caller == c || receiver == c);
        }

    }
```

仿真支持两个种类的连接，它们是：

```
class Local extends Connection {
        Local(Customer a, Customer b) {
            super(a, b);
            System.out.println("[new local connection from " +
                a + " to " + b + "]");
        }
    }
    class LongDistance extends Connection {
        LongDistance(Customer a, Customer b) {
            super(a, b);
            System.out.println("[new long distance connection from " +
                a + " to " + b + "]");
        }
    }
```

用下面的命令编译和运行这个基本的仿真系统：

```
ajc -argfile telecom/basic.lst
java telecom.BasicSimulation
```

Timing 方面保持着对每个客户的总连接时间的跟踪——启动和连接一个定时器的过程被关联到每个连接。它有一些辅助的类。

一个 Timer 对象简单地记录启动和停止的当前时间，当需要它们的共用时间时返回它们的差。在下面的方面 TimeLog 要用到启动和停止的时间并将它输出到标准输出中去。

```
    class Timer {
```

```
        long startTime, stopTime;

        public void start() {
            startTime = System.currentTimeMillis();
            stopTime = startTime;
        }

        public void stop() {
            stopTime = System.currentTimeMillis();
        }

        public long getTime() {
            return stopTime - startTime;
        }
    }
```

Timing 方面定义了一个 Customer 对象的类型间属性（域）totalConnectTime 来保存每个客户积累的连接时间。另外在每个连接对象 Connection 当中也定义了一个 Timer：

```
    public long Customer.totalConnectTime = 0;
    private Timer Connection.timer = new Timer();
```

两个后通知确保 timer 被启动——一一个是在连接完成时，，另外一个当电话挂断时，在这个时刻也进行记分（计算时间）。切点 endTiming 被定义了所以它能够被 Billing 方面使用。

```
public aspect Timing {

    public long Customer.totalConnectTime = 0;

    public long getTotalConnectTime(Customer cust) {
        return cust.totalConnectTime;
    }
    private Timer Connection.timer = new Timer();
    public Timer getTimer(Connection conn) { return conn.timer; }

    after (Connection c): target(c) && call(void Connection.complete()) {
        getTimer(c).start();
    }

    pointcut endTiming(Connection c): target(c) &&
        call(void Connection.drop());

    after(Connection c): endTiming(c) {
        getTimer(c).stop();
        c.getCaller().totalConnectTime += getTimer(c).getTime();
        c.getReceiver().totalConnectTime += getTimer(c).getTime();
    }
```

```
}
```

Billing 系统在后添加了开票功能到 Timing 这个电信应用的顶部。

Billing 方面定义了每个 Connection 有一个名为 payer 的类型间属性，这个属性指明了谁呼叫谁该付帐。它也定义了 Connection 的一个类型间方法 callRate，所以本地和长途电话将被指明为不同的——Timer 停止之后它必须被合理地计算出来，Timing.endTiming 做这事，并且在同样的连接点上 Billing 定义了比 Timing 更多的运行于 Timing 的通知之后的通知。最后，它为 Customer 定义了类型间属性和方面以处理 totalCharge：

```
public aspect Billing {
    // precedence required to get advice on endtiming in the right order
    declare precedence: Billing, Timing;

    public static final long LOCAL_RATE = 3;
    public static final long LONG_DISTANCE_RATE = 10;

    public Customer Connection.payer;
    public Customer getPayer(Connection conn) { return conn.payer; }

    after(Customer cust) returning (Connection conn):
        args(cust, ..) && call(Connection+.new(..)) {
        conn.payer = cust;
    }

    public abstract long Connection.callRate();

    public long LongDistance.callRate() { return LONG_DISTANCE_RATE; }
    public long Local.callRate() { return LOCAL_RATE; }

    after(Connection conn): Timing.endTiming(conn) {
        long time = Timing.aspectOf().getTimer(conn).getTime();
        long rate = conn.callRate();
        long cost = rate * time;
        getPayer(conn).addCharge(cost);
    }

    public long Customer.totalCharge = 0;
    public long getTotalCharge(Customer cust) { return cust.totalCharge; }

    public void Customer.addCharge(long charge){
        totalCharge += charge;
    }
}
```

Timing 和 Billing 两个方面包括使系统暂时停止的操作，譬如，用其中一个或两个运行这个仿真系统，我们想找出每个用户的电话机用了多长时间和是否他的钞票已经用完。这个信息存储在类中，但它通过方面的静态方法进行存取，因为这些状态被声明为 private 的。

看 TimingSimulation.java 这个文件，这个类嘴重要的方法是 report(Customer)，它是超类 AbstractSimulation 的实现，这个方法打印出 Customer 的状态，当然这关系到了 Timing 的状态特征。

```
protected void report(Customer c){
    Timing t = Timing.aspectOf();
    System.out.println(c + " spent " + t.getTotalConnectTime(c));
}
```

文件 timing.lst 和 billing.lst 包含了 timing 和 billing 配置的列表，仅用 timing 的特真创建和运行合格应用可以运行如下的命令：

```
ajc -argfile telecom/timing.lst
java telecom.TimingSimulation
```

用 timng 和 billing 的共同特真建立和运行这个应用，可以用下面的命令：

```
ajc -argfile telecom/billing.lst
java telecom.BillingSimulation
```

讨论：

有一些明显的功能依赖于 Timing 和 Billing：

- Billing 比 Timing 更先定义，所以在同样的连接点上 Billing 的后通知运行在 Timing 的之后。
- Billing 使用了切点 Timing.endTiming。
- Billing 需要存取与一个连接想 关联的 timer。

3.6. 可复用的方面

3.6.1. 再论用方面进行代码跟踪

这些例子位于 InstallDir/examples/tracing。

版本 3

不暴露方法 traceEntry 和 tracExit 益处是容易改变它们的接口而没有依赖于其它多余的代码。

这里我们考虑一下没有 AspectJ 的程序设计。假设程序中的一些执行点需要跟踪其变化，方法跟踪的状态信息总是包括在对象的字符串表示中。为了达到这个目的至少有两种方法，一种是保持方法 traceEntry 和 traceExit 的接口如下：

```
public static void traceEntry(String str);
public static void traceExit(String str);
```

这种情况下，调用者必须确保对象的字符串表示作为字符串参数来使用，所以必须象下面这样：

```
Trace.traceEntry("Square.distance in " + toString());
```

另外的一种方式是在跟踪操作中提供第二个参数：

```
public static void traceEntry(String str, Object obj);
public static void traceExit(String str, Object obj);
```

在这种情况下，调用者必须确保发送了正确的对象，至少对象要被传递过来。调用会象这样：

```
Trace.traceEntry("Square.distance", this);
```

在这两种情况中的任何一种，改变跟踪的需求都将会引入多余的代码——每一次调用都将使 traceEntry 和 traceExit 方法有所改变。这不是我们追求的编程效率。

使用方面可以改变这种情况，我们已经看到代码跟踪部分的版本 2 中的 traceEntry 和 traceExit 方法不是公开暴露的，所以改变它们的接口仅仅只影响类 Trace 的很少部分。下面使一个 Trace 方面实现的局部视图：

```
abstract aspect Trace {

    public static int TRACELEVEL = 0;
    protected static PrintStream stream = null;
    protected static int callDepth = 0;

    public static void initStream(PrintStream s) {
        stream = s;
    }

    protected static void traceEntry(String str, Object o) {
        if (TRACELEVEL == 0) return;
        if (TRACELEVEL == 2) callDepth++;
        printEntering(str + ": " + o.toString());
    }

    protected static void traceExit(String str, Object o) {
        if (TRACELEVEL == 0) return;
        printExiting(str + ": " + o.toString());
        if (TRACELEVEL == 2) callDepth--;
    }

    private static void printEntering(String str) {
        printIndent();
        stream.println("Entering " + str);
    }

    private static void printExiting(String str) {
        printIndent();
        stream.println("Exiting " + str);
    }

    private static void printIndent() {
        for (int i = 0; i < callDepth; i++)
            stream.print("    ");
    }

    abstract pointcut myClass(Object obj);
```

```
    pointcut myConstructor(Object obj): myClass(obj) && execution(new(..));
    pointcut myMethod(Object obj): myClass(obj) &&
        execution(* *(..)) && !execution(String toString());

    before(Object obj): myConstructor(obj) {
        traceEntry("" + thisJoinPointStaticPart.getSignature(), obj);
    }
    after(Object obj): myConstructor(obj) {
        traceExit("" + thisJoinPointStaticPart.getSignature(), obj);
    }

    before(Object obj): myMethod(obj) {
        traceEntry("" + thisJoinPointStaticPart.getSignature(), obj);
    }
    after(Object obj): myMethod(obj) {
        traceExit("" + thisJoinPointStaticPart.getSignature(), obj);
    }
}
```

正象你看到的，我们首先设计 traceEntry 和 traceExit 的接口，但这还不是最重要的——我们的目的是在第二个设计上面（代码在目录 examples/tracing/version3），它指明了在 Trace 方面中跟踪需求对代码的改变的影响是有限的。

这个实现值得注意的地方是切点的详细说明，它们现在暴露了对象。为了维护版本 2 的完全一致的行为，我们在方面中包括了静态方法，为了静态方法定义了切点并通知它。而且我们从切点 methods 中排斥了 toString 方法连接点的执行。这里有一个问题是 toString 方法在通知里面被调用，所以如果跟踪它，我们“死”于无限的递归调用之中。在这个细微之出你必须明白——当你写通知，如果通知调用“向后”的对象，总是有递归的可能，所以必须保持清醒的头脑。

实际上拒绝执行连接点可能还不够，如果在它里面调用了其它跟踪方法那么应该有如下限制：

```
&& !cflow(execution(String toString()))
```

即拒绝执行 toString 方法和其所有连接点之下（避免递归）的调用。最后用下面的命令来运行这个示例：

```
ajc -argfile tracing/tracev3.lst
java tracing.version3.TraceMyClasses
```

程序将输出：

```
  --> tracing.TwoDShape(double, double)
  <-- tracing.TwoDShape(double, double)
  --> tracing.Circle(double, double, double)
  <-- tracing.Circle(double, double, double)
  --> tracing.TwoDShape(double, double)
  <-- tracing.TwoDShape(double, double)
  --> tracing.Circle(double, double, double)
  <-- tracing.Circle(double, double, double)
```

```
   --> tracing.Circle(double)
   <-- tracing.Circle(double)
   --> tracing.TwoDShape(double, double)
   <-- tracing.TwoDShape(double, double)
   --> tracing.Square(double, double, double)
   <-- tracing.Square(double, double, double)
   --> tracing.Square(double, double)
   <-- tracing.Square(double, double)
   --> double tracing.Circle.perimeter()
   <-- double tracing.Circle.perimeter()
c1.perimeter() = 12.566370614359172
   --> double tracing.Circle.area()
   <-- double tracing.Circle.area()
c1.area() = 12.566370614359172
   --> double tracing.Square.perimeter()
   <-- double tracing.Square.perimeter()
s1.perimeter() = 4.0
   --> double tracing.Square.area()
   <-- double tracing.Square.area()
s1.area() = 1.0
   --> double tracing.TwoDShape.distance(TwoDShape)
      --> double tracing.TwoDShape.getX()
      <-- double tracing.TwoDShape.getX()
      --> double tracing.TwoDShape.getY()
      <-- double tracing.TwoDShape.getY()
   <-- double tracing.TwoDShape.distance(TwoDShape)
c2.distance(c1) = 4.242640687119285
   --> double tracing.TwoDShape.distance(TwoDShape)
      --> double tracing.TwoDShape.getX()
      <-- double tracing.TwoDShape.getX()
      --> double tracing.TwoDShape.getY()
      <-- double tracing.TwoDShape.getY()
   <-- double tracing.TwoDShape.distance(TwoDShape)
s1.distance(c1) = 2.23606797749979
   --> String tracing.Square.toString()
      --> String tracing.TwoDShape.toString()
      <-- String tracing.TwoDShape.toString()
   <-- String tracing.Square.toString()
s1.toString(): Square side = 1.0 @ (1.0, 2.0)
```

4. 习惯用语

4.1. 介绍

这一章由一些非常小的 AspectJ 代码组成——典型的切点。

```
/* Any call to methods or constructors in java.sql */
```

任何方法和构造方法的调用

```
pointcut restrictedCall():
    call(* java.sql.*.*(..)) || call(java.sql.*.new(..));

/* Any code in my system not in the sqlAccess package */
```

在我的系统包中而不在 sqlAcess 包中

```
pointcut illegalSource():
    within(com.foo..*) && !within(com.foo.sqlAccess.*);
```

声明错误：

```
declare error: restrictedCall() && illegalSource():
    "java.sql package can only be accessed from com.foo.sqlAccess";
```

任何不等于 AbstractFacade 的子类型的实例：

```
pointcut nonAbstract(AbstractFacade af):
    call(* *(..))
    && target(af)
    && !if(af.getClass() == AbstractFacade.class);
```

如果 AbstractFacade 是抽象类或接口，那么必须是子类型的实例并且你可以用下面的方法进行替换：

```
pointcut nonAbstract(AbstractFacade af):
    call(* *(..))
    && target(af);
```

调用 AbstractFacade 子类型定义的的任何方法，但它没有在 AbstractFacade 中被定义：

```
pointcut callToUndefinedMethod():
     call(* AbstractFacade+.*(..))
     && !call(* AbstractFacade.*(..));
```

定义在 AbstractFacade 的子类型中的方法的执行，但不包括 AbstractFacade 的方法的执行：

```
pointcut executionOfUndefinedMethod():
    execution(* *(..))
    && within(AbstractFacade+)
    && !within(AbstractFacade)
```

5. 缺陷

5.1. 介绍

这章包括几个由 AspectJ 程序引起的令人惊奇的行为以及说明怎样来了解它们。

5.2. 无限循环

这里由一个 Java 程序的奇特行为：

```
public class Main {
    public static void main(String[] args) {
        foo();
        System.out.println("done with call to foo");
    }

    static void foo() {
        try {
            foo();
        } finally {
            foo();
        }
    }
}
```

这个程序从不达到 println 方法调用，当它异常中断时也可能没有堆栈跟踪。

这种 Java 程序静默的原因是由于多重 StackOverflowExceptions 的抛出。首先一个无限循环在方法体中产生，finally 子句企图处理，但 finally 子句也产生了无限循环，这时 JVM 不能处理最主要的中断异常。

下面的方面也会产生这种行为：

```
aspect A {
    before(): call(* *(..)) { System.out.println("before"); }
    after():    call(* *(..)) { System.out.println("after"); }
}
```

这是因为调用 println 时也同样调用了匹配切点的 call(* *(..))方法。我们没有获得输出是因为使用了太简单的 after 通知。如果将方面改为：

```
aspect A {
    before(): call(* *(..)) { System.out.println("before"); }
    after() returning:    call(* *(..)) { System.out.println("after"); }
}
```

然后 StackOverflowException 异常的堆栈跟踪信息将被看到。

可以通过使用 within 来进行限制：

```
aspect A {
```

```
    before(): call(* *(..)) && !within(A) { System.out.println("before"); }
    after() returning:    call(* *(..)) && !within(A) { System.out.println("after"); }
}
```

其它的解决方案由更多的切点限制：

```
aspect A {
    before(): call(* MyObject.*(..))    { System.out.println("before"); }
    after() returning:    call(* MyObject.*(..))    { System.out.println("after"); }
}
```

附录 A　AspectJ 快速参考

1. 切点

方法和构造方法：

call(*Signature*)

调用任何方法和构造方法匹配 *Signature*

execution(*Signature*)

执行任何方法和构造方法匹配 *Signature*

属性：

get(Signature)

引用任何属性匹配 Signature

set(Signature)

任何属性的赋值匹配 Signature 赋值变量能用 args 切点来使其暴露

异常句柄：

handler(*TypePattern*)

任何用 *TypePattern* 指明的 Throwable 异常，赋值变量能用 args 切点来使其暴露

通知：

adviceexecution()

任何通知块的执行

初始化：

staticinitialization(*TypePattern*)

用 *TypePattern* 指明的静态初始化的执行

initialization(*Signature*)

当对象用 *TypePattern* 指明的构造方法开始被调用时的初始化，这包括超类的构造方法开始被调用和返回

preinitialization(*Signature*)

当对象用 *TypePattern* 指明的构造方法开始被调用时的初始化前，这包括超类的构造方法开始被调用前

词汇：

within(*TypePattern*)

用 *TypePattern* 指明的连接点

withincode(*Signature*)

用 *TypePattern* 指明的方法和构造方法连接点

Instanceof检测和环境暴露：

this(*Type* or *Id*)

当前执行对象是 *Type* or *Id* 的实例的连接点

target(*Type* or *Id*)

目标执行对象是 *Type* or *Id* 的实例的连接点

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

当参数是 *Type* or *Id* 的实例的连接点

控制流：

cflow(*Pointcut*)

捕捉切点 Pointcut 的连接点 P 的控制流——也就是流中的连接点。包括 P 自身。

cflowbelow(*Pointcut*)

捕捉切点 Pointcut 的连接点 P 之下的控制流，不包括 P 本身。

条件：

if(*Expression*)

当 *Expression* 为真的连接点集合

结合：

! *Pointcut*

不捕捉切点 *Pointcut* 的连接点集合

Pointcut0* && *Pointcut1

捕捉切点 *Pointcut0* 和 *Pointcut1* 的连接点集合

Pointcut0* || *Pointcut1

捕捉切点 *Pointcut0* 或 *Pointcut1* 的连接点集合

(*Pointcut*)

捕捉切点 *Pointcut* 的连接点集合

2. 类型样式

TypeNamePattern

所有的 Name 类型样式。

SubtypePattern

所有的子类型样式，样式用一个“＋”号表示。

ArrayTypePattern

所有的数组类型样式，样式用一个或多个“[]”表示。

!TypePattern

所有的 not 类型样式。

TypePattern0 && TypePattern1

所有的 *TypePattern0* 和 *TypePattern1* 样式

TypePattern0 || TypePattern1

所有的 *TypePattern0* 或 *TypePattern1* 样式

(TypePattern)

所有的 *TypePattern* 样式

这里 *TypeNamePattern* 可以是任意的普通类型名、“*”通配符或标识符加“*”和“..”通配符。

一个含有“*”的标识符匹配任何的字符系列，但不匹配包和类型间分隔符号“.”。

一个含有“..”的标识符匹配任何的字符系列，但包括包和类型间分隔符号“.”。

3. 通知

每一个通知块有下列形式：

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

AdviceSpec 是下列内容其中之一：

before(***Formals*** **)**

在连接点之前运行。

after(***Formals*** **) returning [(** ***Formal*** **)]**

在连接点正常返回之后运行。可选的 *Formal* 给出返回值。

after(***Formals*** **) throwing [(** ***Formal*** **)]**

在连接点抛出一个 Throwable 后运行。如果给出了选项 *Formal*，那么仅在连接点抛出类型为 *Formal* 的异常后运行。并且 *Formal* 给出了 Throwable 的异常的值。

after(***Formals*** **)**

在连接点处不管是否是正常返回还是抛出了异常后都将运行。

Type **around(** ***Formals*** **)**

在连接点处运行。连接点持续地被调用，那么将获得相同数量和参数类型的 around 通知。

在通知体内有三个特殊的变量可以用到：

thisJoinPoint

类型org.aspectj.lang.JoinPoint 的一个对象描述了一个连接点——在通知执行的时候。

thisJoinPointStaticPart

和 thisJoinPoint.getStaticPart()的意义相同，但可以较少地占用系统运行时间。

thisEnclosingJoinPointStaticPart

动态地封装连接点的静态部分。

4. 类型间成员声明

类型间成员是下列其中之一：

Modifiers ReturnType OnType . Id (Formals) [throws TypeList] { Body }

OnType 上的方法

abstract *Modifiers ReturnType OnType . Id (Formals) [throws TypeList] ;*

OnType 上的抽象方法

***Modifiers OnType* . new (*Formals*) [throws *TypeList*] { *Body* }**

OnType 上的构造方法

***Modifiers Type OnType . Id* [= *Expression*] ;**

OnType 上的属性（域）

4. 其它的声明

declare parents : *TypePattern* extends *Type* ;

类型 *TypePattern* 扩展 *Type* 。

declare parents : *TypePattern* implements *TypeList* ;

类型 *TypePattern* 实现 *TypeList* 。

declare warning : *Pointcut* : *String* ;

让编译器通知（用 *String*）在程序中可能存在的切点 *Pointcut* 的任何连接点集合。

declare error : *Pointcut* : *String* ;

让编译器通知错误（用 *String*）在程序中可能存在的切点 *Pointcut* 的任何连接点集合。

declare soft : *Type* : *Pointcut* ;

捕捉切点*Pointcut*的异常抛出，它被封装包**org.aspectj.lang.SoftException**中。

declare precedence : *TypePatternList* ;

根据列表 *TypePatternList* 决定通知的次序

4. 方面

方面有如下的形式：

[privileged] *Modifiers* aspect *Id* [extends *Type*] [implements *TypeList*] [*PerClause*] { *Body* }

这里 *PerClause* 是实例或联合（issingleton 是缺省定义）。

参见下面的表格：

PerClause	描述	算子
[issingleton]	方面的实例	**所有连接点集的 aspectOf()**
perthis(*Pointcut*)	实例是一个联合（当前对象切点 *Pointcut* 处的连接点集合）	**所有连接点集的 aspectOf(Object)**
pertarget(*Pointcut*)	实例是一个联合（目标对象是 Object 上的切点 *Pointcut* 处的连接点集合）	**所有连接点集的 aspectOf(Object)**
percflow(*Pointcut*)	定义了切点 *Pointcut* 的连接点集合的控制流的入口	**被 cflow(*Pointcut*)限制的所有连接点的 aspectOf()**
percflowbelow(*Pointcut*)	定义了切点 *Pointcut* 的连接点集合的控制流之下的入口	**被 cflowbelow(*Pointcut*)限制**的所有连接点的 aspectOf()

附录 B　AspectJ 语义

1. 介绍

AspectJ 在连接点概念的基础上扩展了 Java 语言语义，并添加了几个新的程序设计元素。连接点的一个较好的定义是程序执行的点——包括方法、构造方法的调用、域的存取和其它一些程序执行时的描述。切点捕捉连接点集合，切点能暴露连接点集在执行上下文中的值。有几个原始的切点已经被设计好，另外的切点用 pointcut 声明。一个通知是切点的连接点处代码的执行。通知存取被切点暴露的变量值。通知用 before、after 和 around 加以声明。方面的类型间声明形式有静态横切的特征。也就是说，代码可以改变一个程序的类型结构——用新的属性、构造方法和方法添加和扩充接口和类。一些类型间声明是通过平常的属性、方法和构造方法完成的，还有另外一些类型间声明是通过关键字 declare 完成的。一个方面是封装切点、通知和静态横切功能的横切类型。

关于类型，我们的 Java 概念是：代码的模块有良好的定义，它在编译时刻确定。方面用关键字 aspect 声明。

2. 连接点

AspectJ 不允许完全任意的横切。更正确的说法是：方面的定义类型是横切程序中执行的点，这些点被叫做连接点。

连接点是程序的执行点，在 AspectJ 中连接点被定义为：

Method call

当一个方法被调用时，但不包括超类非静态方法的调用。

Method execution

一个实际方法执行的代码体。

Constructor call

当对象被创建并且对象的初始化构造方法被调用（注意并不是“super”和“this”构造方法调用）。对象被构造被返回是构造方法调用的连接点。返回类型是对象的类型，并且这

个对象它自己可能用 after returning 通知进行存取。

Constructor execution

当 this 和 super 构造方法调用后一个实际的构造方法执行代码体。这个对象被创建为当前执行的对象，因而可能要用 this 切点进行存取。这里的构造方法的连接点可以是超类的构造方法被调用，同样也包括非静态封装类的初始化。从一个构造方法执行的连接点没有值返回，所以返回类型可以考虑为“void”。

Static initializer execution

当一个类执行静态初始化时。没有值从一个静态初始化执行中返回，所以返回类型可以考虑为“void”。

Object pre-initialization

对象初始化之前代码运行时。这包括了自身的构造方法和超类的构造方法第一次被调用的那段时间。因此这些连接点的执行包括了 this()和 super()构造方法调用的参数赋值的连接点。没有值返回，所以返回类型可以考虑为“void”。

Object initialization

当一个实际的类初始化代码运行时。这包括了超类构造方法的返回和它自己第一次构造方法调用的返回。它包括所有的动态初始化和用构造方法建立对象的连接点。这个对象被构造为当前的执行对象，所以它有可能存取 this 切点。没有值返回，所以返回类型可以考虑为“void”。

Field reference

当一个非常量域（属性）被引用时。

【注意】常量引用(static final 属性被绑定到常量字符串对象和原始值)不是一个连接点，因为需要它们内嵌于 Java 之中。

Field set

当域被赋值时。没有值返回，所以返回类型可以考虑为“void”。

【注意】常量域的初始化(static final 属性被绑定到常量字符串对象和原始值)不是一个连接点，因为需要它们内嵌于 Java 之中。

Handler execution

当一个异常句柄执行时。没有值返回，所以返回类型可以考虑为“void”。

Advice execution

当通知体的代码块执行时。

3. 切点

切点是一个程序设计元素——它捕捉连接点和在连接点的执行环境中暴露数据。切点首先被通知使用。切点能通过布尔操作符加进其它的切点。AspectJ 语言提供的原始切点和相关的配合是下面的这些内容：

call(*MethodPattern*)

捕捉签名方法 *MethodPattern* 调用的连接点。

execution(*MethodPattern*)

捕捉签名方法 *MethodPattern* 执行的连接点。

get(*FieldPattern*)

捕捉签名为 *FieldPattern* 的属性引用的连接点。

set(*FieldPattern*)

捕捉签名为 *FieldPattern* 的属性设置的连接点。

call(*ConstructorPattern*)

捕捉签名为 *ConstructorPattern* 的构造方法调用的连接点。

execution(*ConstructorPattern*)

捕捉签名为 *ConstructorPattern* 的构造方法执行的连接点。

initialization(*ConstructorPattern*)

捕捉有签名为 *ConstructorPattern* 的构造方法的对象初始化的连接点。

preinitialization(*ConstructorPattern*)

捕捉有签名为 *ConstructorPattern* 的构造方法的对象初始化前的连接点。

staticinitialization(*TypePattern*)

捕捉有签名为 *TypePattern* 的静态初始化执行的连接点。

handler(*TypePattern*)

捕捉有签名为 *TypePattern* 的异常句柄的连接点。

adviceexecution()

捕捉有所有通知执行的连接点。

within(*TypePattern*)

捕捉被类型 *TypePattern* 匹配定义的代码执行的连接点。

withincode(*MethodPattern*)

捕捉被方法 *MethodPattern* 匹配定义的代码执行的连接点。

withincode(*ConstructorPattern*)

捕捉被构造方法 *ConstructorPattern* 匹配定义的代码执行的连接点。

cflow(*Pointcut*)

捕捉被切点 pointcut 捕捉到的连接点 p 的控制流的连接点，包括 p 自身。

cflowbelow(*Pointcut*)

捕捉被切点 pointcut 捕捉到的连接点 p 的控制流的连接点，不包括 p 自身。

this(*Type* or *Id*)

捕捉当前对象（被绑定到 this）实例执行的连接点——实例由 *Type* 和 *Id* 描述。

target(*Type* or *Id*)

捕捉目标对象（被应用于对象上的调用和属性操作）实例的连接点——实例由 *Type* 和 *Id* 描述（必须绑定和封装后放入通知或者切点定义）。它不匹配任何静态的调用、引用和设置成员。

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

捕捉具有适当类型样式的实例的连接点。

***PointcutId*(*TypePattern* or *Id*, ...)**

捕捉被命名为 *PointcutId* 的切点的连接点。

if(*BooleanExpression*)

捕捉布尔表达式 *BooleanExpression* 为真时的连接点。这个布尔表达式仅能够存取静态成员、切点和通知暴露的变量和 thisJoinPoint 数据。特别的，它不能在方面中调用非静态方法。

! *Pointcut*

不捕捉切点 *Pointcut* 的连接点。

Pointcut0* && *Pointcut1

捕捉切点 *Pointcut0* 和 *Pointcut1* 的连接点集合。

Pointcut0* || *Pointcut1

捕捉切点 *Pointcut0* 或 *Pointcut1* 的连接点集合。

(*Pointcut*)

捕捉切点 *Pointcut* 的连接点集合。

3.1.切点定义

切点用 pointcut 声明：

```
pointcut publicIntCall(int i):
    call(public * *(int)) && args(i);
```

一个命名切点可以定义在类和方面之中。在类和方面中就象是一个成员一样。作为成员它可以由 public 和 private 修饰符：

```
class C {
    pointcut publicCall(int i):
    call(public * *(int)) && args(i);
}

class D {
    pointcut myPublicCall(int i):
    C.publicCall(i) && within(SomeType);
}
```

切点不是 final 的就可以定义为 abstract 的——没有体。抽象的切点仅能在抽象方面中声明：

```
abstract aspect A {
    abstract pointcut publicCall(int i);
}
```

在这种情况下，扩展抽象的方面必须重置抽象切点：

```
aspect B extends A {
    pointcut publicCall(int i): call(public Foo.m(int)) && args(i);
}
```

切点一旦用 final 声明就终止了其在扩展方面中的扩展。

切点的声明有点象方法的声明，它可以在子方面中被重置（override），但不能重载（overload）。在相同的类和方面中声明两个相同名字的切点会产生错误。

切点的范围是封装类的范围。在这一点上它不同于其它成员，其它成员的范围是封装类的体。所以下列代码是和语法的：

```
aspect B percflow(publicCall()) {
    pointcut publicCall(): call(public Foo.m(int));
}
```

3.2.暴露切点环境

切点有接口用于暴露它捕捉到的连接点的执行环境的一些部分。譬如：PublicIntCall 暴露方法接收的参数。环境（一个 Java 方法形式参数）向切点和通知暴露。这些形式参数被名字匹配绑定。

在切点和通知的声明中，有些切点允许放入 Java 的基本类型和集合类型的标识符，这

些切点是 this、target 和 args。在这些情况下用标识符远好于用“类型”，用类型必须做两件事情，首先选择基于类型的形式参数的连接点，所以切点 pointcut intArg(int i): args(i);通过一个 int 参数捕捉连接点，其次在切点和通知中封装参数变量。

这样变量能在切点中暴露，所以：

```
pointcut publicCall(int x): call(public *.*(int)) && intArg(x);
pointcut intArg(int i): args(i);
```

是合语法的——它捕捉所有公共方法的调用，而这些公共方法接收 int 参数并使它暴露。

有一种特殊的暴露方式，暴露对象类型的参数——这和匹配原始数据类型一样，所以：

```
pointcut publicCall(): call(public *.*(..)) && args(Object);
```

仅仅捕捉那些参数类型为 Object 的子类型的方法，但

```
pointcut publicCall(Object o): call(public *.*(..)) && args(o);
```

捕捉那些有任何参数的方法，并且如果参数是 int 型时，那么传递给通知的将是：java.lang.Interger 类型。

3.3.原始切点

与方法有关系的切点

AspectJ 提供了两个原始的切点来捕获方法调用和执行的连接点集合。

call(*MethodPattern*)

execution(*MethodPattern*)

与域(属性)有关系的切点

AspectJ 提供了两个原始的切点来捕获域的引用和设置的连接点集合。

get(*FieldPattern*)

set(*FieldPattern*)

所有设置的连接点已经被处理为有一个参数。这个值被设置给域。所以在这个设置的连接点处这个值可以用 args 切点进行存取，所以下面这个方面监视类型 T 中 int 型 x 变量的改变情况：

```
aspect GuardedX {
    static final int MAX_CHANGE = 100;
    before(int newval): set(int T.x) && args(newval) {
    if (Math.abs(newval - T.x) > MAX_CHANGE)
        throw new RuntimeException();
    }
}
```

与对象建立有关系的切点

AspectJ 提供了下列用于捕获对象初始化执行的连接点：

call(*ConstructorPattern*)

execution(*ConstructorPattern*)

initialization(*ConstructorPattern*)

preinitialization(*ConstructorPattern*)

与类初始化有关系的连接点

AspectJ 提供了一个个原始的切点来捕获静态初始化执行的连接点集合。

staticinitialization(*TypePattern*)

与异常句柄有关系的切点

AspectJ 提供了一个原始的切点来捕获异常句柄的执行。

handler(*TypePattern*)

所有异常句柄的连接点已经被处理为有一个参数。这个值被设置给域。所以在这个设置的连接点处这个值可以用 args 切点进行存取，所以下面这个方面放置 FooException 对象进入异常句柄处理前：

```
aspect NormalizeFooException {
    before(FooException e): handler(FooException) && args(e) {
    e.normalize();
    }
}
```

与通知有关系的切点

AspectJ 提供了一个原始的切点来捕获通知的执行。

adviceexecution()

下面这个方面从控制流中过滤出通知：

```
aspect TraceStuff {
    pointcut myAdvice(): adviceexecution() && within(TraceStuff);

    before(): call(* *(..)) && !cflow(myAdvice) {
    // do something
    }
}
```

基于状态的通知

当一个有实际类型的对象执行时有许多的横切关注点动态地“产生”。AspectJ 提供了原始地切点在这些时刻来捕获连接点。这些切点有它们动态地对象类型。它们也可能被暴露——用于对象的区分。

this(*Type* or *Id*)

target(*Type* or *Id*)

this 切点捕捉当前执行对象（对象绑定到 this）的一个特殊类型的实例的连接点。target 捕捉目标对象（在对象上的方法调用和域的存取）的一个特殊类型的连接点。注意 target 并不知道对象的当前连接点正在传递控制。这个意思时说在当前方法执行的连接点处目标对象和当前对象是同一个对象。

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

args 捕捉到的连接点是一些参数，这些参数是一些类型的实例。每个元素由逗号分隔，所分隔的内容可能由四种组成：

1. 如果它是一个类型名，那么参数所在的位置必须是那个类型的实例；
2. 如果是一个标识符，那么标识符必须被绑定到通知和切点声明中，所以参数的位置必须

是一个用标识符指明的类型的实例（）；

3. 如果是“*”通配符，那么将匹配任何参数；
4. 如果是“..”通配符，那么将匹配任何数量的参数；

你应该正确地使用签名的样式，所以切点 args(int, .., String)捕捉那些第一个参数是 int 类型而最后一个参数是 String 类型的连接点。

基于控制流的切点

程序的控制流的横切关注点，用下面两个原始切点捕捉：

cflow(*Pointcut*)

cflowbelow(*Pointcut*)

cflow 捕捉那些被切点 *Pointcut* 捕捉到的连接点 P 的出口和入口，包括 P 本身。因此它捕捉被切点 *Pointcut* 捕捉到的连接点的控制流。

cflowbelow 捕捉那些被切点 *Pointcut* 捕捉到的连接点 P 的出口和入口，不包括 P 本身。因此它捕捉被切点 *Pointcut* 捕捉到的连接点的控制流。

基于程序代码的切点

有时为了描述许多关注点——程序运行时刻的结构，必须安排一些词汇，AspectJ 允许捕捉程序代码中进行联合的连接点：

within(*TypePattern*)

withincode(*MethodPattern*)

withincode(*ConstructorPattern*)

within 捕捉由类型样式 *TypePattern* 声明的连接点，这包括类和对象的初始化，方法以及构造方法执行的连接点。另外任何连接点的联合有声明和有有关类型的表达式。它也包括用代码联合的嵌套类型的连接点，并且也包括类型的默认构造方法。

withincode 切点捕捉那些方法和构造方法代码执行的连接点，这包括任何方法和构造方法，并且也包括联合那些方法和构造方法声明和表达式的连接点并且也包括匿名类型。

基于表达式的切点

if(*BooleanExpression*)

如果切点捕捉基于一个动态的属性的连接点，那么你应该使用布尔表达式，在表达式的内部可以使用 thisJoinPoint 对象，所以一个极端的低效率的方式是使用切点：

```
if(thisJoinPoint.getKind().equals("call"))
```

3.4.签名

切点的一个非常重要的属性是它的签名。它被许多的方面用来选择连接点。

方法

连接点联合方法的典型是签名方法，方法的签名由方法名、参数类型、返回类型以及声明的异常（检测）类型组成。

一个方法调用的连接点，签名是方法签名。有资格的静态类型可以存取方法，这话的意思是从((Integer)i).toString()建立的连接点不同于从((Object)i).toString()建立的连接点，即使是相同的变量。

在方法执行的连接点处，有资格的签名类型是方法的声明类型。

属性

连接点联合属性有一个典型是域签名，域的签名由域的名字和类型组成，域的引用连接点有一个这样的签名，但没有参数。域的设置连接点也有这样一个参数，但有一个单一的参数，这个参数的类型和域的类型相同。

构造方法

构造方法的签名由参数类型、异常声（检测）明类型和声明类型组成。

在构造方法调用的连接点处，由被调用的构造方法签名，在构造方法执行的连接点处由当前执行的构造方法签名。

在对象初始化和初始化前的连接点处，由开始初始化的那个构造方法签名：在这个对象类型初始化的构造方法第一次进入期间。

其它

在异常执行的连接点处，由异常类型（异常句柄）签名。

在通知执行的连接点处，签名由方面的类型、通知的参数的类型、返回类型（void 尽管由 around 通知）和异常的类型完成（组成）。

3.5.匹配

withincode, call, execution, get, 和 set 原始切点指明了描述连接点的签名样式。一个签名样式是一个抽象的一个或多个连接点的签名描述。签名样式有意匹配相同种类的事务。

在 AspectJ 中方法和构造方法的声明除了和 Java 一样外，可以使用通配符：

譬如：

call(public final void C.foo() throws ArrayOutOfBoundsException)

call(public final void *.*() throws ArrayOutOfBoundsException)

下面我们重点讨论**基于匹配的抛出异常子句**。

类型样式可以用于基于抛出异常子句的方法和构造方法的连接点。这允许用下面两种极端的用法来符描述切点：

```
pointcut throwsMathlike():
    // each call to a method with a throws clause containing at least
    // one exception exception with "Math" in its name.
    call(* *(..) throws *..*Math*);

pointcut doesNotThrowMathlike():
    // each call to a method with a throws clause containing no
    // exceptions with "Math" in its name.
    call(* *(..) throws !*..*Math*);
```

一个 **ThrowsClausePattern** 是一个用逗号分隔的列表 **ThrowsClausePatternItems**。

ThrowsClausePatternItem :

[!] *TypeNamePattern*

ThrowsClausePattern 匹配任何代码成员的签名，*ThrowsClausePatternItem* 必须匹配成员的抛出异常子句。如果条目（item）不做匹配，那么全部的样式都不作匹配。

如果抛出子句 ThrowsClausePatternItem 的开始有“!”，那么被命名的异常类型决不被抛出，抛出子句匹配 *ThrowsClausePattern*。

“!”的匹配规则有一个令人吃惊的性质，看下面两个切点：

- call(* *(..) throws !IOException)
- call(* *(..) throws (!IOException))

将匹配不同的调用：

void m() throws RuntimeException, IOException {}

[1] 将不匹配方法 m，因为 m 断言抛出 IOException 异常；

[2] 将匹配方法 m，因为 m 断言要抛出的异常将不做匹配。

3.6.类型样式

首先所有的类型名也是类型样式，所以 Object、java.lang.util.HashMap、Map.Entry、int 是类型样式。特殊的类型名、“*”是类型样式，“*”捕捉所有的类型，包括原始类型，所以，

call(void foo(*))

捕捉返回值为 void 的 foo 方法调用的连接点。

“*”和“..”都是类型样式，“*”匹配除“.”字符外的零个或多个字符，所以有下面的表示：handler(java.util.*Map)

捕捉 java.util 包的异常句柄，但不包括内部类 java.util.Map.Entry。

“..”匹配任何的字符系列，所以你能够捕捉任何类型的任何子包，譬如：

within(com.xerox..*)

捕捉代码中类型为 com.xerox 的任何类型的连接点。

子类型样式

你可能需要捕捉一个类型的所有子类型（或者类型的一个集合），这时用“＋”通配符。“＋”通配符允许直接匹配一个类型名的样式。

call(Foo.new())

捕捉 Foo 的所有构造子调用的连接点。而

call(Foo+.new())

捕捉 Foo 的任何子类的实例——构造子调用的连接点（包括 Foo 本身）。下面的切点是靠不住的：

call(*Handler+.new())

数组类型样式

一个类型名称或子类型样式名后可以跟一个或多个方括号系列以构成数组类型样式。譬如：Object[]、com.Xerox..*[][]和 Object+[]。

类型样式之间可以用&&、||和!进行操作。

staticinitialization(Foo || Bar)

捕捉 Foo 或 Bar 的静态初始化的连接点。

call((Foo+ && ! Foo).new(..))
捕捉 Foo 的子类型的构造子调用的连接点，但不包括它自己。

4. 通知

每一个通知形如：
[strictfp] ***AdviceSpec*** **[throws** ***TypeList*** **] : *Pointcut* { *Body* }**
AdviceSpec 是些劣内容的其中之一：
before(*Formals*)
after(*Formals*) returning [(*Formal*)]
after(*Formals*) throwing [(*Formal*)]
after(*Formals*)
***Type* around(*Formals*)**

通知定义了横切的行为。它用切点进行定义。通知运行在切点捕捉到的连接点处。应该正确地使用通知地种类。

AspectJ 有三种通知种类。通知地种类决定了在连接点处代码如何相互影响。AspectJ 的通知可以运行于连接点运行之前、之后和连接点运行的周围。

前通知的实现没有什么问题。而后通知有三种，它们分别是连接点正常执行之后通知、抛出异常后通知和包括前两种功能的后通知。

```
aspect A {
    pointcut publicCall(): call(public Object *(..));
    after() returning (Object o): publicCall() {
    System.out.println("Returned normally with " + o);
    }
    after() throwing (Exception e): publicCall() {
    System.out.println("Threw an exception: " + e);
    }
    after(): publicCall(){
    System.out.println("Returned or threw an Exception");
    }
}
```

After returning 通知可以不用当心返回的对象，这种情况下可以这样书写代码：

```
after() returning: call(public Object *(..)) {
    System.out.println("Returned normally");
}
```

如果 After returning 通知暴露了返回对象，那么在通知中要考虑用 instanceof 来约束：确保返回值是适当的类型。

对于适当的类型 Java 遵循这样的判断：byte 值可以赋值给 short 参数，反之不然，int 之可以赋值给 float 参数，boolean 值只可以赋值给 boolean 参数，引用类型用 instanceof 帮助确认。

有两种特殊的情况：如果暴露值是一个 Object，那么通知不强迫其类型的一致性：实际的返回值是为了通知体而修改对象类型：int 值描述为 java.lang.Integer 对象等等，并且无值被描述为 null。

其次，null 值可以赋值给参数 T——如果连接点能返回类型 T 的（一些）对象。

Around 通知运行于连接点操作之上的地方，不在连接点执行的之前和之后，因为它可以返回一个值，所以它必须象方法一样要声明返回值。它可以声明返回值是 void 的，这样它就没有返回值，它必须返回值除非 around 通知抛出异常。

下面的 around 通知返回一个常量：

```
aspect A {
    int around(): call(int C.foo()) {
    return 3;
    }
}
```

在 around 通知体中，原始连接点的计算有一个专用的语法：

proceed(...)

proceed 接收被通知的切点暴露的环境。由于 around 通知必须有返回，所以下面的通知是正确的，某些值被平分：

```
aspect A {
    int around(int i): call(int C.foo(Object, int)) && args(i) {
    int newi = proceed(i*2)
    return newi/2;
    }
}
```

如果 around 通知的返回值是 Object，那么处理的结果是这个对象的表示被修改了——即使是一个原始值，所以可以用另外的方式书写上面那个方面 A：

```
aspect A {
    Object around(int i): call(int C.foo(Object, int)) && args(i) {
    Integer newi = (Integer) proceed(i*2)
    return new Integer(newi.intValue() / 2);
    }
}
```

在所有的通知种类中，通知参数的行为象方法的参数，赋值仅仅影响参数的值，譬如：

```
aspect A {
    after() returning (int i): call(int C.foo()) {
    i = i * 2;
    }
}
```

通知的变量将不加倍返回，它是局部参数（就像局部变量一样），改变这个参数的值要用 around 通知。

4.1. 通知修饰符

strictfp 仅用于修饰通知，并且它影响所有的 float 表达式，使计算更精确。

4.2. 通知和检测异常

一个通知声明了 throws 子句，那么通知将监听通知体可能抛出的异常。为了通知的每个目标连接点异常检测的列表必须是一致的，否则错误会被编译器通知：

```
import java.io.FileNotFoundException;

class C {
    int i;

    int getI() { return i; }
}

aspect A {
    before(): get(int C.i) {
    throw new FileNotFoundException();
    }
    before() throws FileNotFoundException: get(int C.i) {
    throw new FileNotFoundException();
    }
}
```

这两个通知块是不合规定的，首先，通知体抛出异常没有声明，其次引用的连接点并没有抛出 FileNotFoundException 异常。

在 AspectJ 中可以抛出的连接点的异常种类是：

method call and execution

被目标方法的 throws 子句声明的异常检测

constructor call and execution

被目标构造子的 throws 子句声明的异常检测

field get and set

这些连接点没有异常检测

exception handler execution

被目标构异常句柄抛出的异常检测

static initializer execution

这些连接点没有异常检测

pre-initialization and initialization

初始化类的所有构造子的 throws 子句声明的异常检测

advice execution

通知的 throws 子句声明的任何异常

4.3. 通知的优先序

多个通知可以作用于同样的连接点，那么怎样决定通知的优先序呢？

如何决定优先序

如果有两个通知块被定义在不同的方面中，那么有如下三种情况：

[1] 如果方面 A 早于方面 B 使用 declare precedence 形式，那么在相同的连接点处，方面 A 的所有通知将先于方面 B 的所有通知；
[2] 另外方面 A 是方面 B 的子方面，那么 A 的通知先于 B 的通知，所以除非另有说明，否则有 declare precedence 声明，子类的通知将优先于超类的通知；
[3] 两个通知定义在两个不同的通知中，优先序不确定。

如果两个通知定义在同一个方面中，那么有两种情况：
[1] 如果是 after 通知，那么出现较晚的那个先于出现较早的那个；
[2] 另外的通知中，出现较早的那个先于出现较晚的那个；

这些规则会导致形成一个环，例如：

```
aspect A {
    before(): execution(void main(String[] args)) {}
    after():  execution(void main(String[] args)) {}
    before(): execution(void main(String[] args)) {}
}
```

这样的环将出现编译错误。

优先序的影响

在一个实际的连接点处，通知的优先序是已经被安排好了的：
around 通知有最高优先级，调用 proceed 的 around 通知低一个级次；
before（低一个级次）通知能防止低优先级的通知在运行中抛出异常。如果它正常地返回，那么下一个优先级的通知将运行
after returning（低一个级次）通知
after throwing（低一个级次）通知
after（低一个级次）通知

——如果没有其它更深级次的通知的话。

4.4. 连接点的反射存取

有三个特殊的变量可以访问通知的代码体：thisJoinPoint, thisJoinPointStaticPart, 和 thisEnclosingJoinPointStaticPart。它们每一个都将绑定到一个对象——封装当前通知和连接点的那个对象。这些变量存在的理由是：一些切点可能捕捉到非常大的连接点的集合，如：

```
pointcut publicCall(): call(public * *(..));
```

捕捉许多方法调用，但是在通知体上切点可以存取一个连接点的方法和参数。ThisJoinPoint 被绑定到连接点对象。ThisJoinPointStaticPart 被绑定到连接点对象的一部分上面，它包括的信息很少，但为了解决内存分配问题要用到它，它等价的形式是 thisJoinPoint.getStaticPart()。ThisEnclosingJoinPointStaticPart 被绑定到封装当前连接点的连接点的一部分上面，仅只是封装连接点的静态部分——通过一定的装置。

标准的 Java 反射机制用 java.lang.reflect 包的层次对象实施。同样地，AspectJ 也有一个对象的层次机制。对象的类型绑定到 ThisJoinPoint 对象即 org.aspectj.lang.ThisJoinPoint，thisStaticJoinPoint 被绑定到 org.aspectj.lang.JoinPoint.StaticPart 接口对象类型。

5. 静态横切

通知声明改变横切类的行为，但不改变它们的静态类型结构。为了横切关注点能对类型层次的静态结构产生作用，AspectJ 提供了类型间成员声明和其它的 declare 形式。

5.1. 类型间成员声明

AspectJ 允许在方面中声明成员并用于联合其它类型。

一个类型间成员声明有如下形式：

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

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

如此的一个声明所造成地影响是 ***OnType*** 支持新的方法。即使 ***OnType*** 是一个接口（即使方法既不是 public 的也不是 abstract 的）。所以下面的代码合乎 AspectJ 语言的语法：

```
interface Iface {}

aspect A {
    private void Iface.m() {
    System.err.println("I'm a private method on an interface");
    }
    void worksOnI(Iface iface) {
    // calling a private method on an interface
    iface.m();
    }
}
```

一个类型间构造子声明地语法是：

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

如此的一个声明所造成地影响是 ***OnType*** 支持新的构造方法。但在接口中是错误地。

类型间声明的构造子不能给一个在 ***OnType*** 中定义地 final 变量赋值。这个限制意味着了解和编译类 ***OnType*** 和方面的声明相分离。

注意在 Java 中类如果没有定义构造子，那么它隐含定义了一个无参数的构造子调用 supper()，如果企图定义一个类型间无参数的构造子可能会引起冲突，即使看上去没有构造子被定义。

类型间于的声明如下：

[*Modifiers*] *Type OnType* . *Id* = *Expression*;

[*Modifiers*] *Type OnType* . *Id*;

如此地一个声明所造成地影响是 ***OnType*** 支持新的域。即使 ***OnType*** 是一个接口（即使域既不是 public 的，也不是 static 的和不是 final 的）。

类型间声明的域的初始化运行于目标类的本地类初始化之前。

This 可以用在类型间构造子和方法声明地体中，也可以用在类型间域声明的初始化中，也可以在方面类型中引用 ***OnType*** 对象。和 Java 的传统语法一样在 static 的类型间成员声明中使用 this 是错误的。

5.2. 存取修饰符

类型间成员声明可以显示地用 public、private 和用于包保护的缺省的 protected 修饰。但 AspectJ 不提供 protected 修饰符。

存取修饰符的应用关系到方面，但不关系到目标类型。所以一个 private 的类型间成员仅在被定义的方面中被断言。一个有缺省定义的类型间成员仅在方面的包中被断言——即保护由方面构成的包而不是类。

注意一个断言为 private 的类型间方法不同于在其它类的 private 方法。Private 的类型间方法仅允许在方面中存取，而类中的 private 方法在目标类中存取。Java 的序列化，如：void writeObject(ObjectOutputStream) 实现 java.io.Serializable 接口。private 方法的类型间声明不履行这样的任务，因为它是一个私有的方面不是一个私有目标类型。

5.3. 冲突

在本地类的成员和类型间成员在类型间声明时可能会发生冲突。例如，下面 otherPakage 不包含方面 A，代码：

```
aspect A {
    private Registry otherPackage.*.r;
    public void otherPackage.*.register(Registry r) {
    r.register(this);
    this.r = r;
    }
}
```

在每个 otherPackage 类型有一个 r，它仍然是在方面 A 中进行存取的，然而方面也定义了每个 otherPackage 类型有一个“register”方法，但这个方法可以在任何地方存取。如果类型 otherPackage 已经存在一个 private 或 package-protected 的域 r，那么就产生了冲突。方面不知道有此域并且代码也看不到有关 otherPackage 的域类型间声明 r。在类型 otherPackage 中定义的 public 域 r 是有冲突的，表达式 this.r = r 是错误的，因为类型间声明的 private 的 r 和类型声明的 public 的 r 是关系暧昧的（不明确的）。

如果在类型 otherPackage 中定义任何 register(Register)方法，也会引起冲突，它们和类型间声明的 register 方法同样是关系暧昧的。

这个冲突规则也是 Java 的冲突规则：

A subclass can inherit multiple *fields* from its superclasses, all with the same name and type. However, it is an error to have an ambiguous *reference* to a field.

一个子类能继承一个超类的多个域，以同名同类型的方式，但不管怎样引用一个关系暧昧的域是错误的。

A subclass can only inherit multiple *methods* with the same name and argument types from its superclasses if only zero or one of them is concrete (i.e., all but one is abstract, or all are abstract).

子类仅仅以同名同参数类型的方式从超类继承多个方法，仅仅只有零个或一个是具体方法（或者一个抽象方法，或所有抽象方法等等）。

下面给出一个在不同的方面中类型间声明可能的潜在的冲突，如果一个方面有优先级超

过其它的声明，编译器不会有任何通知，这确实是真的——当父方面优先级显示地用 declare precedence 声明，子方面也用它声明，但子方面隐含地优先级超过了它的父方面。

5.4. 扩充和实现

任何方面可以改变一个系统的继承层次——改变一个类型的超类和添加一个父接口到这个类型之上。使用 declare parents 关键字可以达到这样的目的：

```
declare parents: TypePattern extends Type;
declare parents: TypePattern implements TypeList;
```

例如：

```
aspect A {
    declare parents: SomeClass implements Runnable;
    public void SomeClass.run() { ... }
}
```

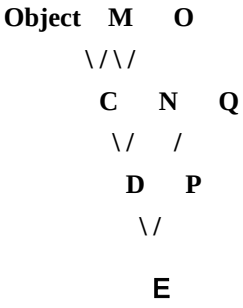
如果一个方面象要实现一个具体地 Runnable 类，它可以定义一个适当地类型间 void run()方法，但它也必须履行实现 Runnable 接口的义务。为了实现 Runnable 接口的方法，类型间 run 方法必须是 public 的。

5.5. 使用接口成员

通过类型间成员声明，接口的（非 public、static 和 fianl）域和（非 public 的 abstract）方法可以被继承。现在冲突可能发生在一个超类和多个子类的继承之中。

因为接口支持非 static 的初始化，每个接口的行为就象有一个零个参数的构造子的初始化。父接口的初始化顺序是非常重要的。我们用一个模型固定下面的顺序：父类型先于子类型初始化，初始化的代码仅执行一次，它的父接口的初始化先于它自己类型的父类。看下面的模型：

{Object, C, D, E}是类，{M, N, O, P, Q}是接口



当一个新的 E 被初始化，那么初始化采用这个顺序：

Object M C O N D Q P E

5.6. 警告和错误

AspectJ 可以指明公开下列内容：

declare error: *Pointcut*: *String*;

declare warning: *Pointcut*: *String*;

用 ***String*** 指明错误或警告信息。

5.7. Softtened 异常

可以指明一个实际的异常种类——如果在连接点处抛出异常的话，应该设置旁路于 Java 通常的 static 异常检测系统盘边并且抛出一个 org.aspectj.lang.SoftException 异常，它是 RuntimeException 的子类型并且 RuntimeException 不需要声明，例如：

```
aspect A {
    declare soft: Exception: execution(void main(String[] args));
}
```

如果愿意，在连接点的执行处，可以在捕获到任何异常时抛出一个 org.aspectj.lang.SoftException 异常，譬如下面方面中的通知：

```
aspect A {
    void around() execution(void main(String[] args)) {
    try { proceed(); }
    catch (Exception e) {
        throw new org.aspectj.lang.SoftException(e);
    }
    }
}
```

除了影响包装它的异常处理块之外，它也影响 Java 的静态异常检测机制。

declare soft 声明形式不影响一个抽象的方面——不延伸进一个还没有分裂的方面（**具体**方面）。所以下面的代码不会编译除非扩张一个具体方面：

```
abstract aspect A {
    abstract pointcut softeningPC();

    before() : softeningPC() {
        Class.forName("FooClass"); // error:    uncaught ClassNotFoundException
    }

    declare soft : ClassNotFoundException : call(* Class.*(..));
}
```

5.8. 通知的优先序

在具体方面中可以声明方面之间的优先序，用如下形式：

declare precedence : *TypePatternList* ;

下面这一点很重要，如果连接点的通知从两个具体方面出发匹配样式 ***TypePatternList***，那么通知的优先序将遵从这个列表。

在 ***TypePatternList*** 中，通配符至多使用一次。意思是：在列表中任何类型不被其它任何样式匹配。例如：

```
declare precedence: *..*Security*, Logging+, *;
```

这个约束有两层意思：

(1) 在那些用 Security 作为它们的名字的一部分的方面中，通知的优先序超过其它的方面；

(2) 在那些用 Loggin 作为它们的名字的一部分的方面(或扩展他的方面)中，其优先序超过了 non-securuty 的方面。

再看下面的例子：

```
aspect Ordering {
    declare precedence: CountEntry, DisallowNulls;
}
aspect DisallowNulls {
    pointcut allTypeMethods(Type obj): call(* *(..)) && args(obj, ..);
    before(Type obj):    allTypeMethods(obj) {
    if (obj == null) throw new RuntimeException();
    }
}
aspect CountEntry {
    pointcut allTypeMethods(Type obj): call(* *(..)) && args(obj, ..);
    static int count = 0;
    before():    allTypeMethods(Type) {
    count++;
    }
}
```

方面 CountEntry 可能想要计算当前包中一个对象的进入次数（args 的第一个参数），即使方面 DisallowNulls 的通知体抛出异常，它也应该计算所有的对象进入次数。这样可以在一个另外的通知 Ordering 中声明 CountEntry 的通知先于 DisallowNulls 的同名通知运行。

我们在这里讨论以下各种各样的优先序声明的循环：

```
declare precedence:    A, B, A ;    // error
```

象这样声明的优先序是错误的。

无论如何多个声明构成一个环状种类声明是合语法的。譬如：

```
declare precedence: B, A;
declare precedence: A, B;
```

在一个系统中容纳两个这样的声明是合法的，方面 A 和 B 不共享一个连接点。在这样的习惯用法中 A 和 B 要很好地保持中立。

考虑下面的一个方面库：

```
abstract aspect Logging {
```

```
        abstract pointcut logged();

        before(): logged() {
            System.err.println("thisJoinPoint: " + thisJoinPoint);
        }
    }

    abstract aspect MyProfiling {
        abstract pointcut profiled();

        Object around(): profiled() {
            long beforeTime = System.currentTimeMillis();
            try {
                return proceed();
            } finally {
                long afterTime = System.currentTimeMillis();
                addToProfile(thisJoinPointStaticPart,
                                afterTime - beforeTime);
            }
        }
        abstract void addToProfile(
            org.aspectj.JoinPoint.StaticPart jp,
            long elapsed);
    }
```

为了用库中任一的方面，必须用一个具体方面来扩展，由于在具体方面中声明：
declare precedence: Logging, Profiling;所以不会对 Logging 和 Profiling;造成影响。但

```
    declare precedence: MyLogging, MyProfiling;
    declare precedence: Logging+, Profiling+;
```

确是值得认真思考的。

5.9. 静态可决定的切点

用 declare 公开的切点有几点限制，它们是静态可决定的。因此它们不包括（区别）直接地或间接地基于动态环境（运行时）的切点。所以这些切点不能用下列条目：

cflow
cflowbelow
this
target
args
if

所有这些条目能区别运行时信息。

6. 方面

方面是被 aspect 声明的横切类型。aspect 声明类似于 class 声明。它们的不同之处在于 aspect 可以横切其它类型（包括其它被声明为方面的类型）。它的实例并不是直接由 new 表达式、克隆或序列化对象产生。方面也可以定义一个构造子，但如果这样的话这个构造子必须是一个没有参数和异常检测的构造子。

方面可以定义任意的包级别上，或者当作一个静态的嵌套方面——一个类、接口和方面的静态成员。如果方面不在包水平上定义，那么方面必须用关键字 static 定义，局部和匿名的方面是不允许的。

6.1. 方面的扩充

方面支持横切关注点的抽象和复合。方面能象类一样地进行扩充。方面通过一些新添加的规则进行扩充。

方面可以扩充类和实现接口

一个抽象或具体的方面可以扩展一个类和实现一个接口的集合。扩展一个类但不提供用 new 表达式创建一个实例的能力。没有构造子，方面依然可以得到定义。

类不能扩充方面

类实现或扩充一个方面是错误的。

方面可以扩充方面

方面可以扩充另外的方面，在这种情况下，除切点以外的域和方法都能被继承。无论如何，方面可以扩充抽象方面，但一个具体的方面扩充一个具体的方面是错误的。

6.2. 方面的实例化

不象类，方面的实例不用 new 表达式创建，方面的实例在系统的横切程序中被自动地创建。

由于通知仅运行于方面的上下文环境，所以方面的实例直接控制通知的运行。

如果方面没有一个父方面，那么该方面是一个 singleton（单例）方面。

aspect *Id* { ... }

aspect *Id* issingleton { ... }

默认定义的方面（包括用 issingleton 修饰定义）是哪个完整横切程序的一个实例。这个实例是一个变量，在程序运行的任何时刻都可以用定义在方面之上的 aspectOf()得到它。因此，A.asoectOf()返回 A 的实例。方面的实例在方面“类文件”（classfile）载入时实例化。

因为方面的实例中有程序运行过程中的所有连接点（一旦类文件载入），所有连接点的通知都将会偶然地发生。

Per-object 方面

aspect *Id* perthis(*Pointcut*) { ... }
aspect *Id* pertarget(*Pointcut*) { ... }

如果方面 A 被定义了 perthis(*Pointcut*)，那么类型 A 的一个对象是为了横切被 ***Pointcut*** 捕捉到的每个连接点处的执行对象而被建立的。定义在 A 的通知可以（因而）运行在当前连接点的执行对象上——这些对象用方面 A 来联合。

类似地，如果方面 A 被定义了 pertarget(*Pointcut*)，那么类型 A 的一个对象是为了横切被 ***Pointcut*** 捕捉到的每个连接点处的目标对象而被建立的。定义在 A 的通知可以（因而）运行在当前连接点的目标对象上——这些对象用方面 A 来联合。

在这种情况下，调用方法 A.aspectOf(Object)能获得被 Object 登记的方面 A 的实例。每个方面的实例都尽可能快地被建立，但在达到被 ***Pointcu*** 捕捉到的连接点前类型 A 的方面不被联合。

perthis 和 pertarget 可能受到 AspectJ 编译器代码的影响，这一点在下面有关“实现的注意事项”中加以讨论。

Per-control-flow 方面

aspect *Id* percflow(*Pointcut*) { ... }
aspect *Id* percflowbelow(*Pointcut*) { ... }

如果一个方面 A 被定义了 percflow(*Pointcut*) 或 percflowbelow(*Pointcut*)，那么类型 A 的一个对象是为了横切被 ***Pointcut*** 捕捉到的每个连接点控制的流而建立的，它们每一个分别作为控制流的进入或作为在控制流之下的连接点而被建立。定义在 A 中的通知可以运行于连接点处或其控制流之下。在这样的控制流执行期间，A.aspectOf()将返回方面 A 的一个对象。方面的实例被建立在每一个这样的控制流之上。

方面的实例化和通知

同志运行于一个方面的上下文，但可以用切点写一个通知的片段——切点捕捉方面初始化之前的连接点，譬如：

```
public class Client
{
    public static void main(String[] args) {
        Client c = new Client();
    }
}

aspect Watchcall {
    pointcut myConstructor(): execution(new(..));

    before(): myConstructor() {
        System.err.println("Entering Constructor");
    }
```

```
}
```

这个前通知运行于系统所有构造子的执行之前。由于它运行于方面 Watchcall 的上下文环境，所以仅有一种方式来获得 Watchcall 方面的实例——Watchcal 的默认构造子执行，但其执行之前需要执行前通知。

6.3. 方面的特权

privileged aspect *Id* { ... }

当使用类和方面的成员时，在方面中写代码服从和 Java 代码一样的存取控制规则：方面不允许存取成员，那么使用默认的书写方法（package-protected）除非方面被定义在同样的包中。

在许多方面中这些限制是适当的——为了方面中可以有通知和类型间成员需要存取其它类型的 private 和 protected 资源。为了这个目的可以用 privileged 声明方面：方面可以存取所有成员，即使是一个 private 的成员。

```
class C {
    private int i = 0;
    void incI(int x) { i = i+x; }
}
privileged aspect A {
    static final int MAX = 1000;
    before(int x, C c): call(void C.incI(int)) && target(c) && args(x) {
    if (c.i+x > MAX) throw new RuntimeException();
    }
}
```

在这种情况下，如果方面 A 不声明为 privileged 的，那么域 c.i 的引用将会发生编译器错误。一个 privileged 的方面有能力存取一个特殊成员的多个版本。正如看到的那样：并不是 privileged 获得优先权。譬如：

```
class C {
    private int i = 0;
    void foo() { }
}
privileged aspect A {
    private int C.i = 999;
    before(C c): call(void C.foo()) target(c) {
    System.out.println(c.i);
    }
}
```

最初 private 的类型间声明被邦定到 999，而后通知的体引用 C 的 private 域，因为 A 存取它自己的类型间声明域，即使它不是 privileged 的。

一个 privileged 的方面可以存取其它方面的 private 的类型间成员。

附录 C　AspectJ 实现的注意事项

1. 有关编译器的注意事项

AspectJ 的最初实现为完全基于编译器的。确定 AspectJ 的语义学元素是非常困难的——如果不进行 JVM 的修改，那么这样一个基于编译器的实现将不可能。一种容易实现的方法是安排有关这个问题的详细清单。但，我们选择了一个稍微不同的方法，指定了一个理想的语言语义，也清晰地定义了允许偏离这个语义的方式。这些开发一直保持到今天都是一致的，而且明天也将做得更好。

象下面的通知声明：

```
before(): get(int Point.x) { System.out.println("got x"); }
```

允许通知所有的通知从类型 Point（及其子类型）实例存取类型为 int 的 x 的域，在源代码履行存取这个变量的义务的时候不管方面是否容纳通知而被编译，或者在晚些时候进行改变这些内容。

但 AspectJ 允许偏离这个这个有良好定义的方式——可以实现为通知仅在代码执行时进行存取。每个执行自由地邦定到提供的控制代码的自身定义上。

在当前的 AspectJ 编译器 ajc 中，为了任何方面的代码在编译期间影响变量，ajc 有意地控制字节码——这个意思是说：一些类如果用了表达试 new Point().x（连接点在运行时获得一个域），当前的 AspectJ 编译器将编译错误除非 Java 的类文件已经被编译好（比如 Client.java 编译为 Client.class）。因此用连接点联合有 native 方法的代码是不能考虑的。

不同的连接点有不同需求，方法或构造子能够被通知仅仅是 ajc 控制字节码调用的结果。域引用和赋值的连接点仅仅是 ajc 控制字节码作用于调用处（caller）的结果——代码实际地引用和赋值（处）。连接点的初始化仅仅是由于 ajc 控制类型初始化的字节码的结果。连接点的执行能够被通知是 ajc 控制方法和构造子的体的结果。最后 Handler 异常句柄在字节码中的修改理由或材料不是很充分，因此在 ajc 中不实现 Handler 连接点的 after 和 around 通知，类似地 ajc 不实现初始化和初始化前连接点的 around 通知。在这些情况下，编译一个便宜器不实现的通知，是错误的。

AspectJ 提供的 perthis 和 pertarget 也有基于代码控制的限制。在这种情况中，在连接点处当前执行对象的字节码不是可用的，所以方面定义的连接点 perthis 将不被联合，所以方面为每个对象 Object 定义的 perthis(Object)将不建立方面实例除非编译的是 Object 对象是可编译的部分。类似地 pertarget 也有同样的限制。

象 declare parents 的类型间声明也有基于代码控制的限制，如果一个类型间声明的目标字节码不是可用的，那么在目标类型上的类型间声明将不能制作。所以

```
declare parents : String implements MyInterface
```

将不为 java.lang.String 工作除非 java.lang.String 是可编译的部分。

当成员定义在接口上时，必须控制接口或接口的顶级实现（实现类的接口但不实现超类的接口）。这种编织有利于模块分离，但你必须明白如果你运行顶级类那么你将获得一个异常——接口没有被相同 ajc 实现当作成熟的产品而加以延伸。最后你必须明白不能在接口中定义 static 的域或方法。

重要的事情是记住 AspectJ 的核心概念，连接点是不能改变的。在你开发期间你必须明白 ajc 编译器的局限。这些局限对于你的方面的设计应该不是阻力。

2. 有字节码的注意事项

2.1. 类的表达式和 String+表达式

Java 语言的类 Foo.class 用字节码实现，调用 Class.forName 方法是在被异常句柄 ClassNotFoundException 监视其异常状况中进行的。

Java 中的“+”运算符，当用 String 做参数时，在字节码中被 StringBuffer.append 调用。

在这两种情况中，当前的 AspectJ 编译器在字节码上实施操作来实现这些语言特征。

当前的 AspectJ 编译器考虑到了想下面的一些连接点：

```
class Test {
    void main(String[] args) {
        System.out.println(Test.class);              // calls Class.forName
        System.out.println(args[0] + args[1]); // calls StringBuffer.append
    }
}
```

2.2. Handler 异常句柄连接点

Handler 异常句柄不能在字节码中被可靠地找到。当前的 AspectJ 编译器限制 Handler 连接点于下面两点，而不是以移走的方式代替它：

After and around advice cannot apply to handler join points.

The control flow of a handler join point cannot be detected.

（

After 和 around 通知不用于 handler 连接点

handler 连接点的控制流不能被发觉

）

after 通知（returning、throwing 和 finally）可以正常地应用于 hangdler 连接点，它将不被当前的 AspectJ 编译器输出。编译器将产生一个警告。Before 通知是允许的。

Handler 控制流连接点不会被捕捉到：

```
cflow(call(void foo()) || handler(java.io.IOException))
```

它的等价表达式是 cflow(call(void foo()))。一般地，**cflow(handler(*Type*))**不捕捉连接点。

不限制在包中发 before 通知和在其它的 handler 上捕捉控制流连接点：

```
before(): handler(java.io.IOException) && cflow(void parse()) {
    System.out.println("about to handle an exception while parsing");
}
```

AspectJ 的代码实现（比如 AspectJ 1.0.6）能发觉 handler 连接点的终点并且上面的那些限制是一直存在的。

2.3. 初始化和类型间构造子

Java 代码的初始化，如域的赋值，象在下面的类中：

```
class C {
    double d = Math.sqrt(2);
}
```

AspectJ 获得字节码的抓取时构造子被重点考虑到了。

类型间构造子不需要运行一个目标类型的初始化代码。特别地，如果类型间构造子调用一个父类的构造子，当类型间构造子被调用时，目标类型的初始化代码不会运行：

```
aspect A {
    C.new(Object o) {} // implicitly calls super()

    public static void main(String[] args) {
        System.out.println((new C()      ).d);      // prints 1.414...
        System.out.println((new C(null)).d);      // prints 0.0
}
```

类型间构造子的工作需要做所有的初始化工作，或者在需要的请况下委托给 this 构造子。

结论

AspectJ 是对 Java 语言的简单而且实际的面向方面的扩展。仅通过加入几个新结构，AspectJ 提供了对模块化实现各种横切关注点的有力支持。向已经有的 Java 开发项目中加入 AspectJ 是一个直接而且渐增的任务。一条路径就是通过从使用开发方面开始再到产品方面当拥有了 AspectJ 的经验后就使用开发可重用方面。当然可以选取其它的开发路径。例如，一些开发者将从使用产品方面马上得到好处，另外的人员可能马上编写可重用的方面。

AspectJ 可以使用基于名字和基于属性这两种横切点。使用基于名字横切点的方面仅影响少数几个类，虽然它们是小范围的，但是比起普通的 Java 实现来说它们能够减少大量的复杂度。使用基于属性横切点的方面可以有小范围或着大范围。使用 AspectJ 导致了横切关注点的干净、模块化的实现。当编写 AspectJ 方面时，横切关注点的结构变得十分明显和易懂。方面也是高度模块化的，使得开发可拔插的横切功能变成现实。

AspectJ的安装和设置

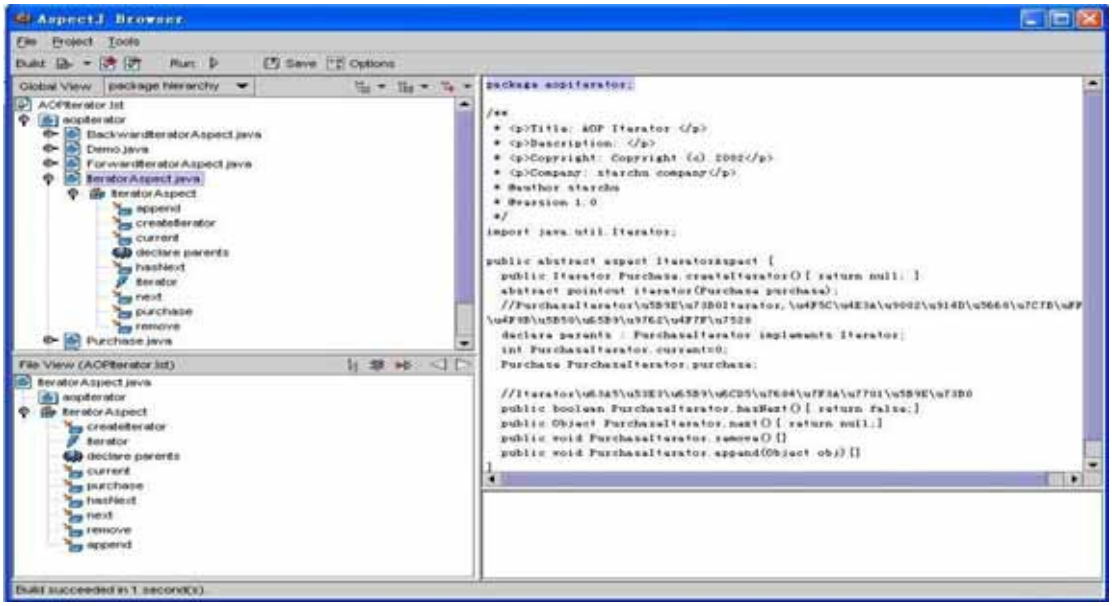
从AspectJ下载页下载AspectJ的最新版本，下载完成后可将其解压缩到指定目录下。然后执行下述步骤：

拷贝<aspectJ install dir>\lib\aspectjrt.jar文件到<java_home>\jre\lib\ext目录下或者将其加入到你的CLASSPATH环境变量中。

创建目录<aspectJ install dir>\bin并将其加入环境变量PATH中，在bin目录下新建两个.bat文件ajc.bat和ajcbrowser.bat。

将<JAVA_HOME>\bin\java.exe -classpath <aspectJ install dir>\lib\aspectjtools.jar - Xmx64 org.aspectj.tools.ajc.Main %*语句拷贝到 ajc.bat 文件中保存，你就可以使用 ajc.bat 为你的 aspectJ 应用编译代码了，例如 ajc -argfile examples.lst。

将<JAVA_HOME>\bin\java.exe -jar <aspectJ install dir>\lib\aspectjtools.jar %*拷贝到 ajcbrowser.bat 文件中保存,则你可以使用 aspectJ 的图形浏览方式编译代码了，例如在命令行敲入 ajcbrowser examples.lst。



AspectJ 的图形编辑编译窗口

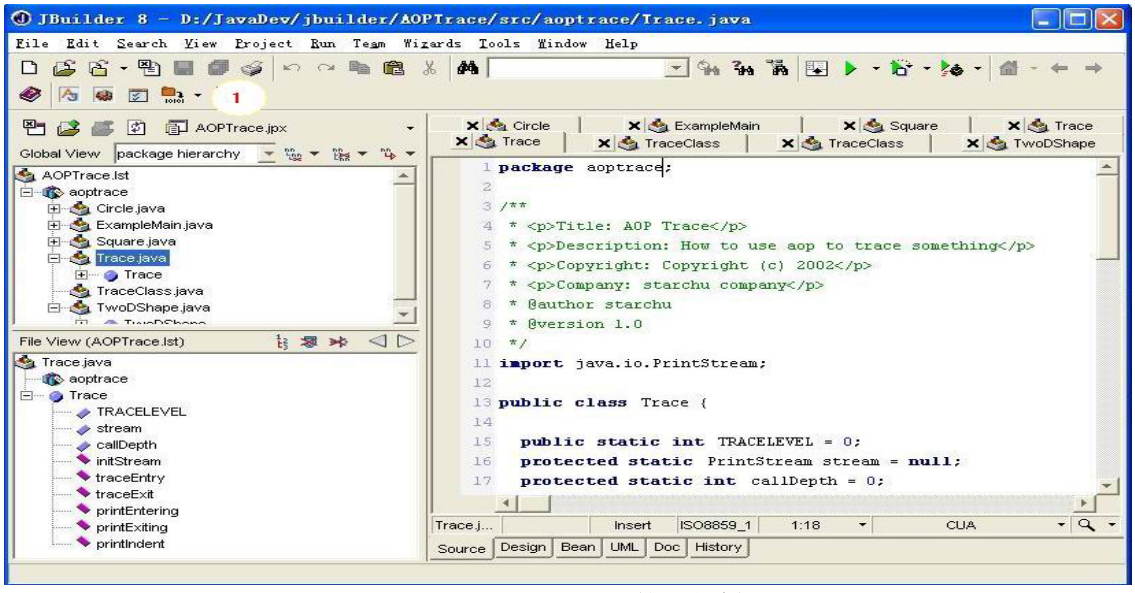
注：<JAVA_HOME>为你的 jdk 的安装目录，<aspectJ install dir>为你的 aspectJ 的安装目录文件扩展名为 “.lst”的文件是包含了所有 aspectJ 应用的文件路径信息的文本文件（必须为绝对路径名，两个路径名中间没有空格或换行符）。

AspectJ For Jbuilder 开发工具的安装和设置

AspectJ For Jbuilder 是支持 Jbuilder 中使用 AspectJ 的开放工具，你可以在 aspectj4jbuilder下载页下载它的最新版本并解压缩到指定目录，然后执行下面的步骤：

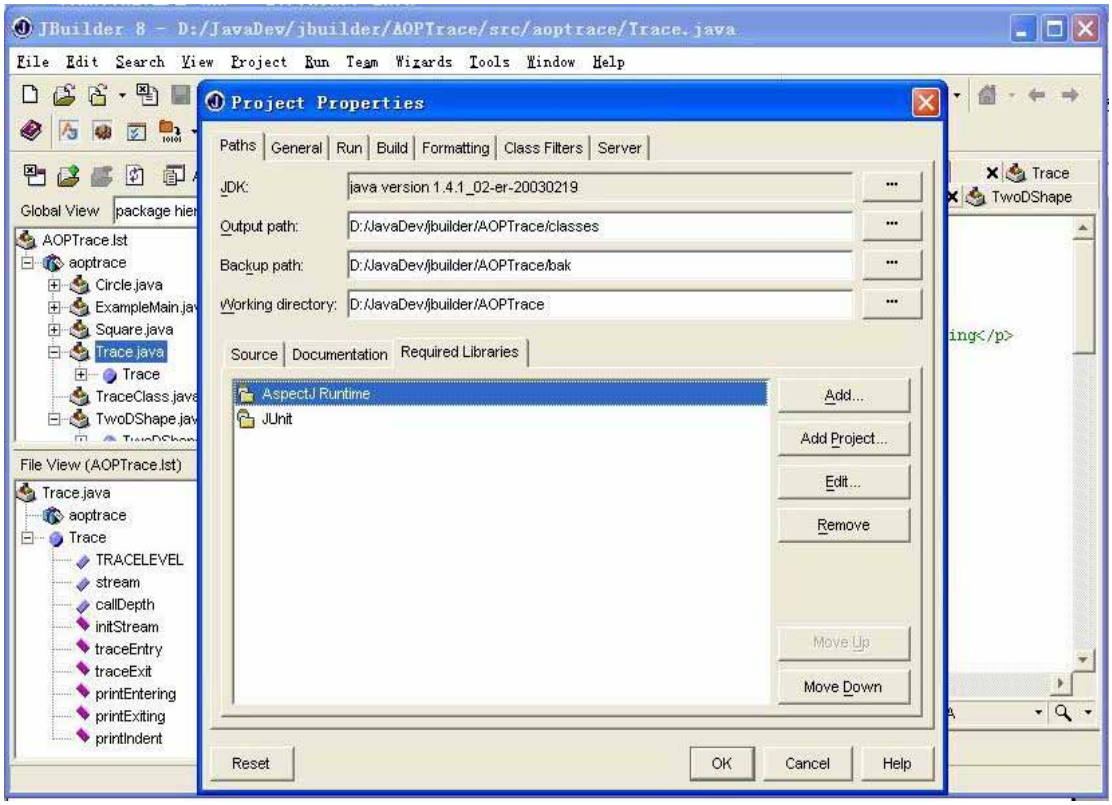
将<aspectJ for jbuilder dir>\lib\ext 目录中下的所有.jar 文件拷贝到 Jbuilder 的 lib\ext 目录下。如果要删除它则将 Jbuilder 的 lib\ext 中的对应的三个文件删除。

然后启动 Jbuider 即可发现在工具栏中多出几个选项



AspectJ for jbuilder 的工具栏目

图中标为”1”的地方有四个图标，按下第一个随即启动 aspectJ 的浏览器，第二个按下后将会提供导航和类浏览的功能，第三个栏目为编译选项，最后一个按下后将编译所有在”.lst”文件中的类。



工程所需的库文件设定

注：编译之前必须在工程的 library 中加入 Aspect Rutime 库，如图三展示。另外由于 aspectJ 的 ajc 编译器不会自动搜索工程路径，所以你需要将所有必须的”.lst”文件加入到当前工程中来，这样 ajc 编译器才会编译所有的文件，否则它只编译已知”.lst”文件中的文件。