**一、AOP**

**目标:**是把某些功能集中起来，放到一个统一的地方来控制和管理。

如果说，OOP如果是把问题划分到单个模块的话，那么AOP就是把涉及到众多模块的某一类问题进行统一管理。

**例子：** 比如我们可以设计两个Aspects，一个是管理某个软件中所有模块的日

志输出的功能，另外一个是管理该软件中一些特殊函数调用的权限检查。

(1)、**AspectJ介绍**

* **代表框架： Hugo(Jake Wharton)**
* **AspectJ支持编译期和加载时代码注入**

AOP虽然是方法论，但就好像OOP中的Java一样，一些先行者也开发了一套语言来支持AOP。目前用得比较火的就是AspectJ了，它是一种几乎和Java完全一样的语言，而且完全兼容Java（AspectJ应该就是一种扩展Java，但它不是像Groovy[1]那样的拓展。）。当然，除了使用AspectJ特殊的语言外，AspectJ还支持原生的Java，只要加上对应的AspectJ注解就好。所以，使用AspectJ有两种方法：完全使用AspectJ的语言。这语言一点也不难，和Java几乎一样，也能在AspectJ中调用Java的任何类库。AspectJ只是多了一些关键词罢了。

或者使用纯Java语言开发，然后使用AspectJ注解，简称@AspectJ。

Anyway，不论哪种方法，最后都需要AspectJ的编译工具ajc来编译。由于AspectJ实际上脱胎于Java，所以ajc工具也能编译java源码。

* **Advice（通知）**: 典型的 Advice 类型有 before、after 和 around，分别表示在目标方法执行之前、执行后和完全替代目标方法执行的代码。
* **Joint point（连接点）**: 程序中可能作为代码注入目标的特定的点和入口.（JPoints就是程序运行时的一些执行点）

<表1  AspectJ中的Join Point>

* **Pointcut（切入点）**: 告诉代码注入工具，在何处注入一段特定代码的表达

式。

例子：

public pointcut testAll(): call(public \* \*.println(..)) && !within(TestAspect) ;

* **Aspect（切面）**: Pointcut 和 Advice 的组合看做切面。例如，在本例中通过定义一个 pointcut 和给定恰当的advice，添加一个了内存缓存的切面。
* **Weaving（织入）**: 注入代码（advices）到目标位置（joint points）的过程。

表1  AspectJ中的Join Point

|  |  |  |
| --- | --- | --- |
| **Join Points** | **说明** | **示例** |
| **method call** | 函数调用 | 比如调用Log.e()，这是一处JPoint |
| **method execution** | 函数执行 | 比如Log.e()的执行内部，是一处JPoint。注意它和method call的区别。method call是调用某个函数的地方。而execution是某个函数执行的内部。 |
| **constructor call** | 构造函数调用 | 和method call类似 |
| **constructor execution** | 构造函数执行 | 和method execution类似 |
| **field get** | 获取某个变量 | 比如读取DemoActivity.debug成员 |
| **field set** | 设置某个变量 | 比如设置DemoActivity.debug成员 |
| **pre-initialization** | Object在构造函数中做得一些工作。 | 很少使用，详情见下面的例子 |
| **initialization** | Object在构造函数中做得工作 | 详情见下面的例子 |
| **static initialization** | 类初始化 | 比如类的static{} |
| **handler** | 异常处理 | 比如try catch(xxx)中，对应catch内的执行 |
| **advice execution** | 这个是AspectJ的内容，稍后再说 |  |