# 深入理解Android之AOP

格式更加精美的PDF版请到：http://vdisk.weibo.com/s/z68f8l0xTgCLK 下载

## 一、闲谈AOP

大家都知道OOP，即ObjectOriented Programming，面向对象编程。而本文要介绍的是AOP。AOP是Aspect Oriented Programming的缩写，中译文为面向切向编程。OOP和AOP是什么关系呢？首先：

* l OOP和AOP都是方法论。我记得在刚学习C++的时候，最难学的并不是C++的语法，而是C++所代表的那种看问题的方法，即OOP。同样，今天在AOP中，我发现其难度并不在利用AOP干活，而是从AOP的角度来看待问题，设计解决方法。这就是为什么我特意强调AOP是一种方法论的原因！
* l 在OOP的世界中，问题或者功能都被划分到一个一个的模块里边。每个模块专心干自己的事情，模块之间通过设计好的接口交互。从图示来看，OOP世界中，最常见的表示比如：

|  |
| --- |
| 图1  [**Android**](http://lib.csdn.net/base/android) Framework中的模块 |

图1中所示为AndroidFramework中的模块。OOP世界中，大家画的模块图基本上是这样的，每个功能都放在一个模块里。非常好理解，而且确实简化了我们所处理问题的难度。

OOP的精髓是把功能或问题模块化，每个模块处理自己的家务事。但在现实世界中，并不是所有问题都能完美得划分到模块中。举个最简单而又常见的例子：现在想为每个模块加上日志功能，要求模块运行时候能输出日志。在不知道AOP的情况下，一般的处理都是：先设计一个日志输出模块，这个模块提供日志输出API，比如Android中的Log类。然后，其他模块需要输出日志的时候调用Log类的几个函数，比如e(TAG,...)，w(TAG,...)，d(TAG,...)，i(TAG,...)等。

在没有接触AOP之前，包括我在内，想到的解决方案就是上面这样的。但是，从OOP角度看，除了日志模块本身，其他模块的家务事绝大部分情况下应该都不会包含日志输出功能。什么意思？以ActivityManagerService为例，你能说它的家务事里包含日志输出吗？显然，ActivityManagerService的功能点中不包含输出日志这一项。但实际上，软件中的众多模块确实又需要打印日志。这个日志输出功能，从整体来看，都是一个面上的。而这个面的范围，就不局限在单个模块里了，而是横跨多个模块。

* l 在没有AOP之前，各个模块要打印日志，就是自己处理。反正日志模块的那几个API都已经写好了，你在其他模块的任何地方，任何时候都可以调用。功能是得到了满足，但是好像没有Oriented的感觉了。是的，随意加日志输出功能，使得其他模块的代码和日志模块耦合非常紧密。而且，将来要是日志模块修改了API，则使用它们的地方都得改。这种搞法，一点也不酷。

AOP的目标就是解决上面提到的不cool的问题。在AOP中：

* l 第一，我们要认识到OOP世界中，有些功能是横跨并嵌入众多模块里的，比如打印日志，比如统计某个模块中某些函数的执行时间等。这些功能在各个模块里分散得很厉害，可能到处都能见到。
* l 第二，AOP的目标是把这些功能集中起来，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。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。

讲了这么多，还是先来看个例子。在这个例子中，我们要：

* l Activity的生命周期的几个函数运行时，要输出日志。
* l 几个重要函数调用的时候，要检查有没有权限。

注意，本文的例子代码在https://code.csdn[**.NET**](http://lib.csdn.net/base/dotnet)/Innost/androidaopdemo上。

## 二、没有AOP的例子

先来看没有AOP的情况下，代码怎么写。主要代码都在AopDemoActivity中

[-->AopDemoActivity.[**Java**](http://lib.csdn.net/base/javase)]

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. **public** **class** AopDemoActivity **extends** Activity {
2. **private** **static** **final** String TAG = "AopDemoActivity";
3.   onCreate，onStart,onRestart,onPause,onResume,onStop,onDestory返回前，都输出一行日志
4. **protected** **void** onCreate(Bundle savedInstanceState) {
5. **super**.onCreate(savedInstanceState);
6. setContentView(R.layout.layout\_main);
7. Log.e(TAG,"onCreate");
8. }
9. **protected** **void** onStart() {
10. **super**.onStart();
11. Log.e(TAG, "onStart");
12. }
13. **protected** **void** onRestart() {
14. **super**.onRestart();
15. Log.e(TAG, "onRestart");
16. }
17. protectedvoid onResume() {
18. **super**.onResume();
19. Log.e(TAG, "onResume");
20.    checkPhoneState会检查app是否申明了android.permission.READ\_PHONE\_STATE权限
21. checkPhoneState();
22. }
23. **protected** **void** onPause() {
24. **super**.onPause();
25. Log.e(TAG, "onPause");
26. }
27. **protected** **void** onStop() {
28. **super**.onStop();
29. Log.e(TAG, "onStop");
30. }
31. **protected** **void** onDestroy() {
32. **super**.onDestroy();
33. Log.e(TAG, "onDestroy");
34. }
35. **private** **void** checkPhoneState(){
36. **if**(checkPermission("android.permission.READ\_PHONE\_STATE")== **false**){
37. Log.e(TAG,"have no permission to read phone state");
38. **return**;
39. }
40. Log.e(TAG,"Read Phone State succeed");
41. **return**;
42. }
43. **private** **boolean** checkPermission(String permissionName){
44. **try**{
45. PackageManager pm = getPackageManager();
46. //调用PackageMangaer的checkPermission函数，检查自己是否申明使用某权限
47. **int** nret = pm.checkPermission(permissionName,getPackageName());
48. **return** nret == PackageManager.PERMISSION\_GRANTED;
49. }......
50. }
51. }

代码很简单。但是从这个小例子中，你也会发现要是这个程序比较复杂的话，到处都加Log，或者在某些特殊函数加权限检查的代码，真的是一件挺繁琐的事情。

## 三、AspectJ介绍

### 3.1  AspectJ极简介

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：

l 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。

l 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。

AspectJ现在托管于Eclipse项目中，官方网站是：

* l http://www.eclipse.org/aspectj/   <=AspectJ官方网站
* l http://www.eclipse.org/aspectj/doc/released/runtime-api/index.html  <=AspectJ类库参考文档，内容非常少
* l http://www.eclipse.org/aspectj/doc/released/aspectj5rt-api/index.html  <=@AspectJ文档，以后我们用Annotation的方式最多。

### 3.2  AspectJ语法

题外话：AspectJ东西比较多，但是AOP做为方法论，它的学习和体会是需要一步一步，并且一定要结合实际来的。如果一下子讲太多，反而会疲倦。更可怕的是，有些胆肥的同学要是一股脑把所有高级玩法全弄上去，反而得不偿失。这就是是方法论学习和其他知识学习不一样的地方。请大家切记。

#### 3.2.1  Join Points介绍

Join Points（以后简称JPoints）是AspectJ中最关键的一个概念。什么是JPoints呢？JPoints就是程序运行时的一些执行点。那么，一个程序中，哪些执行点是JPoints呢？比如：

* l 一个函数的调用可以是一个JPoint。比如Log.e()这个函数。e的执行可以是一个JPoint，而调用e的函数也可以认为是一个JPoint。
* l 设置一个变量，或者读取一个变量，也可以是一个JPoint。比如Demo类中有一个debug的boolean变量。设置它的地方或者读取它的地方都可以看做是JPoints。
* l for循环可以看做是JPoint。

理论上说，一个程序中很多地方都可以被看做是JPoint，但是AspectJ中，只有如表1所示的几种执行点被认为是JPoints：

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 |  |

表1列出了AspectJ所认可的JoinPoints的类型。下面我们来看个例子以直观体会一把。

|  |
| --- |
| 图2  示例代码 |

图2是一个Java示例代码，下面我们将打印出其中所有的join points。图3所示为打印出来的join points：

|  |
| --- |
| 图3  所有的join points |

图3中的输出为从左到右，我们来解释红框中的内容。先来看左图的第一个红框：

* l staticinitialization(test.Test.<clinit>)：表示当前是哪种类型的JPoint，括号中代表目标对象是谁（此处是指Test class的类初始化）。由于Test类没有指定static block，所以后面的at:Test.java:0 表示代码在第0行（其实就是没有找到源代码的意思）。

initialization(test.Test.TestBase.new(int))

* l Test类初始化完后，就该执行main函数了。所以，下一个JPoint就是execution(voidtest.Test.main(String[]))。括号中表示此JPoint对应的是test.Test.main函数。at:Test.java:30表示这个JPoint在源代码的第30行。大家可以对比图2的源码，很准确！
* l main函数里首先是执行System.out.println。而这一行代码实际包括两个JPoint。一个是get(PrintStream java.lang.System.out)，get表示Field get，它表示从System中获取out对象。另外一个是call(void java.io.PrintStream.println(String))，这是一个call类型的JPoint，表示执行out.println函数。

再来看左图第二个红框，它表示TestBase的类的初始化，由于源码中为TestBase定义了static块，所以这个JPoint清晰指出了源码的位置是at:Test.java:5

接着看左图第三个红框，它和对象的初始化有关。在源码中，我们只是构造了一个TestDerived对象。它会先触发TestDerived Preinitialization JPoint，然后触发基类TestBase的PreInitialization JPoint。注意红框中的before和after 。在TestDerived和TestBase所对应的PreInitialization before和after中都没有包含其他JPoint。所以，Pre-Initialization应该是构造函数中一个比较基础的Phase。这个阶段不包括类中成员变量定义时就赋值的操作，也不包括构造函数中对某些成员变量进行的赋值操作。

而成员变量的初始化（包括成员变量定义时就赋值的操作，比如源码中的int base = 0，以及在构造函数中所做的赋值操作，比如源码中的this.derived = 1000）都被囊括到initialization阶段。请读者对应图三第二个红框到第三个红框（包括第3个红框的内容）看看是不是这样的。

最后来看第5个红框。它包括三个JPoint：

* l testMethod的call类型JPoint
* l testMethod的execution类型JPonint
* l 以及对异常捕获的Handler类型JPoint

好了。JPoint的介绍就先到此。现在大家对JoinPoint应该有了一个很直观的体会，简单直白粗暴点说，JoinPoint就是一个程序中的关键函数（包括构造函数）和代码段（staticblock）。

为什么AspectJ首先要定义好JoinPoint呢？大家仔细想想就能明白，以打印log的AopDemo为例，log在哪里打印？自然是在一些关键点去打印。而谁是关键点？AspectJ定义的这些类型的JPoint就能满足我们绝大部分需求。

注意，要是想在一个for循环中打印一些日志，而AspectJ没有这样的JPoint，所以这个需求我们是无法利用AspectJ来实现了。另外，不同的软件框架对表1中的JPoint类型支持也不同。比如[**spring**](http://lib.csdn.net/base/javaee)中，不是所有AspectJ支持的JPoint都有。

##### （1） 自己动手试

图2的示例代码我也放到了https://code.csdn[**.Net**](http://lib.csdn.net/base/dotnet)/Innost/androidaopdemo上。请小伙伴们自己下载玩耍。具体的操作方法是：

* l 下载得到androidaopdemo中，有一个aspectj-test目录。
* l aspectj-test目录中有一个libs目录，里边有一个文件aspectj-1.8.7.jar文件。执行这个文件（java -jar aspectj-1.8.7.jar，安装aspectj）。安装完后，按照图示要求将aspectj的安装路径加到PATH中，然后export。这样，就可以在命令行中执行aspectj的命令了。比如编译工具ajc。
* l 另外，libs下还有一个aspectjrt.jar，这个是aspectjt运行时的依赖库。使用AspectJ的程序都要包含该jar包。
* l 执行create-jar.sh。它会编译几个文件，然后生成test.jar。
* l 执行test.jar（java -jar test.jar），就会打印出图3的log。

我已经编译并提交了一个test.jar到[**Git**](http://lib.csdn.net/base/git)上，小伙伴们可以执行一把玩玩！

#### 3.2.2  Pointcuts介绍

pointcuts这个单词不好翻译，此处直接用英文好了。那么，Pointcuts是什么呢？前面介绍的内容可知，一个程序会有很多的JPoints，即使是同一个函数（比如testMethod这个函数），还分为call类型和execution类型的JPoint。显然，不是所有的JPoint，也不是所有类型的JPoint都是我们关注的。再次以AopDemo为例，我们只要求在Activity的几个生命周期函数中打印日志，只有这几个生命周期函数才是我们业务需要的JPoint，而其他的什么JPoint我不需要关注。

怎么从一堆一堆的JPoints中选择自己想要的JPoints呢？恩，这就是Pointcuts的功能。一句话，Pointcuts的目标是提供一种方法使得开发者能够选择自己感兴趣的JoinPoints。

在图2的例子中，怎么把Test.java中所有的Joinpoint选择出来呢？用到的pointcut格式为：

pointcuttestAll():within(Test)。

AspectJ中，pointcut有一套标准语法，涉及的东西很多，还有一些比较高级的玩法。我自己琢磨了半天，需不需要把这些内容一股脑都搬到此文呢？回想我自己学习AOP的经历，好像看了几本书，记得比较清楚的都是简单的case，而那些复杂的case则是到实践中，确实有需求了，才回过头来，重新查阅文档来实施的。恩，那就一步一步来吧。

##### （1） 一个Pointcuts例子

直接来看一个例子，现在我想把图2中的示例代码中，那些调用println的地方找到，该怎么弄？代码该这么写：

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

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

注意，aspectj的语法和Java一样，只不过多了一些关键词

我们来看看上述代码

  第一个public：表示这个pointcut是public访问。这主要和aspect的继承关系有关，属于AspectJ的高级玩法，本文不考虑。

  pointcut：关键词，表示这里定义的是一个pointcut。pointcut定义有点像函数定义。总之，在AspectJ中，你得定义一个pointcut。

  testAll()：pointcut的名字。在AspectJ中，定义Pointcut可分为有名和匿名两种办法。个人建议使用named方法。因为在后面，我们要使用一个pointcut的话，就可以直接使用它的名字就好。

  testAll后面有一个冒号，这是pointcut定义名字后，必须加上。冒号后面是这个pointcut怎么选择Joinpoint的条件。

  本例中，call(public  \*  \*.println(..))是一种选择条件。call表示我们选择的Joinpoint类型为call类型。

  public  \*\*.println(..)：这小行代码使用了通配符。由于我们这里选择的JoinPoint类型为call类型，它对应的目标JPoint一定是某个函数。所以我们要找到这个/些函数。public  表示目标JPoint的访问类型（public/private/protect）。第一个\*表示返回值的类型是任意类型。第二个\*用来指明包名。此处不限定包名。紧接其后的println是函数名。这表明我们选择的函数是任何包中定义的名字叫println的函数。当然，唯一确定一个函数除了包名外，还有它的参数。在(..)中，就指明了目标函数的参数应该是什么样子的。比如这里使用了通配符..，代表任意个数的参数，任意类型的参数。

  再来看call后面的&&：AspectJ可以把几个条件组合起来，目前支持 &&，||，以及！这三个条件。这三个条件的意思不用我说了吧？和Java中的是一样的。

  来看最后一个!within(TestAspectJ)：前面的!表示不满足某个条件。within是另外一种类型选择方法，特别注意，这种类型和前面讲到的joinpoint的那几种类型不同。within的类型是数据类型，而joinpoint的类型更像是动态的，执行时的类型。

上例中的pointcut合起来就是：

* l 选择那些调用println（而且不考虑println函数的参数是什么）的Joinpoint。
* l 另外，调用者的类型不要是TestAspect的。

图4展示了执行结果：

|  |
| --- |
| 图4  新pointcut执行结果 |

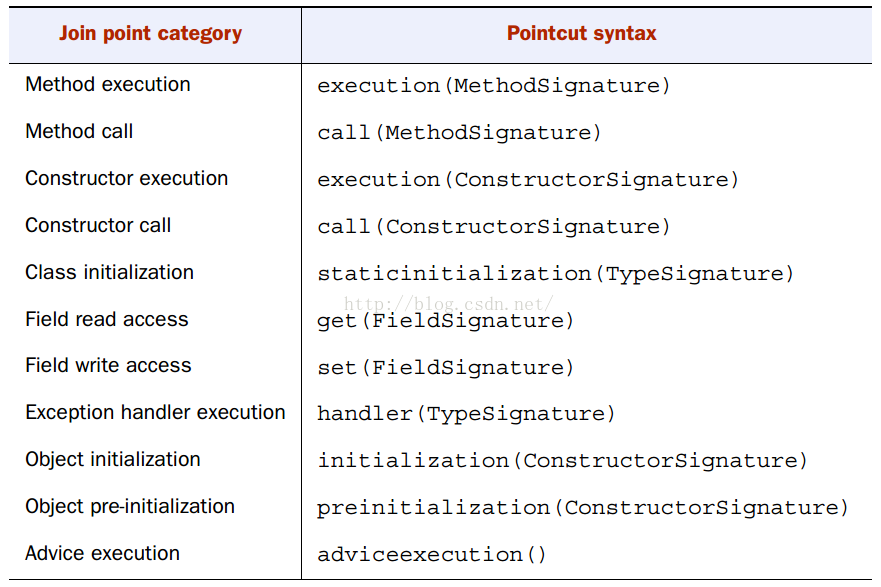
我在图2所示的源码中，为Test类定义了一个public static void println()函数，所以图4的执行结果就把这个println给匹配上了。

看完例子，我们来讲点理论。

##### （2） 直接针对JoinPoint的选择

pointcuts中最常用的选择条件和Joinpoint的类型密切相关，比如图5：

|  |
| --- |
| 图5  不同类型的JPoint对应的pointcuts查询方法 |



以图5为例，如果我们想选择类型为methodexecution的JPoint，那么pointcuts的写法就得包括execution(XXX)来限定。

除了指定JPoint类型外，我们还要更进一步选择目标函数。选择的根据就是图5中列出的什么MethodSignature，ConstructorSignature，TypeSinature，FieldSignature等。名字听起来陌生得很，其实就是指定JPoint对应的函数（包括构造函数），Static block的信息。比如图4中的那个println例子，首先它的JPoint类型是call，所以它的查询条件是根据MethodSignature来表达。一个Method Signature的完整表达式为：

**[plain]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. @注解 访问权限 返回值的类型 包名.函数名(参数)
2.   @注解和访问权限（public/private/protect，以及static/final）属于可选项。如果不设置它们，则默认都会选择。以访问权限为例，如果没有设置访问权限作为条件，那么public，private，protect及static、final的函数都会进行搜索。
3.   返回值类型就是普通的函数的返回值类型。如果不限定类型的话，就用\*通配符表示
4.   包名.函数名用于查找匹配的函数。可以使用通配符，包括\*和..以及+号。其中\*号用于匹配除.号之外的任意字符，而..则表示任意子package，+号表示子类。
5. 比如：
6. java.\*.Date：可以表示java.sql.Date，也可以表示java.util.Date
7. Test\*：可以表示TestBase，也可以表示TestDervied
8. java..\*：表示java任意子类
9. java..\*Model+：表示Java任意package中名字以Model结尾的子类，比如TabelModel，TreeModel
10. 等
11.   最后来看函数的参数。参数匹配比较简单，主要是参数类型，比如：
12. (int, char)：表示参数只有两个，并且第一个参数类型是int，第二个参数类型是char
13. (String, ..)：表示至少有一个参数。并且第一个参数类型是String，后面参数类型不限。在参数匹配中，
14. ..代表任意参数个数和类型
15. (Object ...)：表示不定个数的参数，且类型都是Object，这里的...不是通配符，而是Java中代表不定参数的意思

是不是很简单呢？

**[plain]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. Constructorsignature和Method Signature类似，只不过构造函数没有返回值，而且函数名必须叫new。比如：
2. public \*..TestDerived.new(..)：
3.   public：选择public访问权限
4.   \*..代表任意包名
5.   TestDerived.new：代表TestDerived的构造函数
6.   (..)：代表参数个数和类型都是任意
7. 再来看Field Signature和Type Signature，用它们的地方见图5。下面直接上几个例子：
8. Field Signature标准格式：
9. @注解 访问权限 类型 类名.成员变量名
10.   其中，@注解和访问权限是可选的
11.   类型：成员变量类型，\*代表任意类型
12.   类名.成员变量名：成员变量名可以是\*，代表任意成员变量
13. 比如，
14. set(inttest..TestBase.base)：表示设置TestBase.base变量时的JPoint
15. Type Signature：直接上例子
16. staticinitialization(test..TestBase)：表示TestBase类的static block
17. handler(NullPointerException)：表示catch到NullPointerException的JPoint。注意，图2的源码第23行截获的其实是Exception，其真实类型是NullPointerException。但是由于JPointer的查询匹配是静态的，即编译过程中进行的匹配，所以handler(NullPointerException)在运行时并不能真正被截获。只有改成handler(Exception)，或者把源码第23行改成NullPointerException才行。

以上例子，读者都可以在aspectj-test例子中自己都试试。

##### （3） 间接针对JPoint的选择

除了根据前面提到的Signature信息来匹配JPoint外，AspectJ还提供其他一些选择方法来选择JPoint。比如某个类中的所有JPoint，每一个函数执行流程中所包含的JPoint。

特别强调，不论什么选择方法，最终都是为了找到目标的JPoint。

表2列出了一些常用的非JPoint选择方法：

表2  其它常用选择方法

|  |  |  |
| --- | --- | --- |
| 关键词 | 说明 | 示例 |
| within(TypePattern) | TypePattern标示package或者类。TypePatter可以使用通配符 | 表示某个Package或者类中的所有JPoint。比如  within(Test)：Test类中（包括内部类）所有JPoint。图2所示的例子就是用这个方法。 |
| withincode(Constructor Signature|Method Signature) | 表示某个构造函数或其他函数执行过程中涉及到的JPoint | 比如  withinCode(\* TestDerived.testMethod(..))  表示testMethod涉及的JPoint  withinCode( \*.Test.new(..))  表示Test构造函数涉及的JPoint |
| cflow(pointcuts) | cflow是call flow的意思  cflow的条件是一个pointcut | 比如  cflow(call TestDerived.testMethod)：表示调用TestDerived.testMethod函数时所包含的JPoint，包括testMethod的call这个JPoint本身 |
| cflowbelow(pointcuts) | cflow是call flow的意思。 | 比如  cflowblow(call TestDerived.testMethod)：表示调用TestDerived.testMethod函数时所包含的JPoint，不包括testMethod的call这个JPoint本身 |
| this(Type) | JPoint的this对象是Type类型。  （其实就是判断Type是不是某种类型，即是否满足instanceof Type的条件） | JPoint是代码段（不论是函数，异常处理，static block），从语法上说，它都属于一个类。如果这个类的类型是Type标示的类型，则和它相关的JPoint将全部被选中。  图2示例的testMethod是TestDerived类。所以  this(TestDerived)将会选中这个testMethod JPoint |
| target(Type) | JPoint的target对象是Type类型 | 和this相对的是target。不过target一般用在call的情况。call一个函数，这个函数可能定义在其他类。比如testMethod是TestDerived类定义的。那么  target(TestDerived)就会搜索到调用testMethod的地方。但是不包括testMethod的execution JPoint |
| args(TypeSignature) | 用来对JPoint的参数进行条件搜索的 | 比如args(int,..)，表示第一个参数是int，后面参数个数和类型不限的JPoint。 |

上面这些东西，建议读者：

* l 进入androidaopdemo/aspectj-test目录。
* l 修改test/TestAspect.aj文件。主要是其中的pointcuts:testAll()这一行。按照图2中的解释说明，随便改改试试。
* l 执行./create-jar.sh，得到一个test.jar包，然后java -jar test.jar得到执行结果

注意：this()和target()匹配的时候不能使用通配符。

图6给出了修改示例和输出：

|  |
| --- |
| 图6  示例代码和输出结果 |

注意，不是所有的AOP实现都支持本节所说的查询条件。比如Spring就不支持withincode查询条件。

#### 3.2.3  advice和aspect介绍

恭喜，看到这个地方来，AspectJ的核心部分就掌握一大部分了。现在，我们知道如何通过pointcuts来选择合适的JPoint。那么，下一步工作就很明确了，选择这些JPoint后，我们肯定是需要干一些事情的。比如前面例子中的输出都有before，after之类的。这其实JPoint在执行前，执行后，都执行了一些我们设置的代码。在AspectJ中，这段代码叫advice。简单点说，advice就是一种Hook。

ASpectJ中有好几个Hook，主要是根据JPoint执行时机的不同而不同，比如下面的：

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. before():testAll(){
2. System.out.println("before calling: " + thisJoinPoint);//打印这个JPoint的信息
3. System.out.println("      at:" + thisJoinPoint.getSourceLocation());//打印这个JPoint对应的源代码位置
4. }

testAll()是前面定义的pointcuts，而before()定义了在这个pointcuts选中的JPoint执行前我们要干的事情。

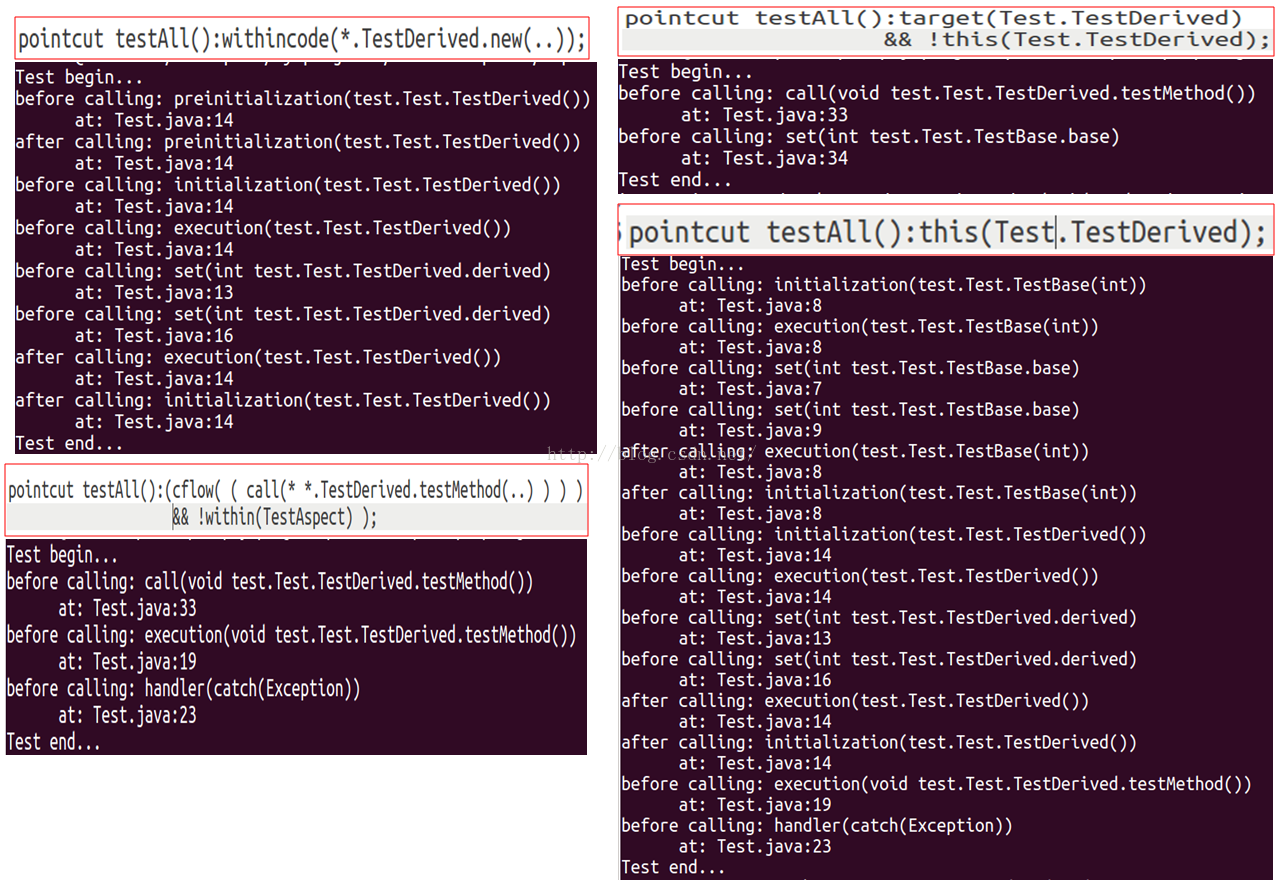
表3列出了AspectJ所支持的Advice的类型：

表3  advice的类型

|  |  |  |
| --- | --- | --- |
| 关键词 | 说明 | 示例 |
| before() | before advice | 表示在JPoint执行之前，需要干的事情 |
| after() | after advice | 表示JPoint自己执行完了后，需要干的事情。 |
| after():returning(返回值类型)  after():throwing(异常类型) | returning和throwing后面都可以指定具体的类型，如果不指定的话则匹配的时候不限定类型 | 假设JPoint是一个函数调用的话，那么函数调用执行完有两种方式退出，一个是正常的return，另外一个是抛异常。  注意，after()默认包括returning和throwing两种情况 |
| 返回值类型 around() | before和after是指JPoint执行前或执行后备触发，而around就替代了原JPoint | around是替代了原JPoint，如果要执行原JPoint的话，需要调用proceed |

注意，after和before没有返回值，但是around的目标是替代原JPoint的，所以它一般会有返回值，而且返回值的类型需要匹配被选中的JPoint。我们来看个例子，见图7。

|  |
| --- |
| 图7  advice示例和结果 |

图7中：

* l 第一个红框是修改后的testMethod，在这个testMethod中，肯定会抛出一个空指针异常。
* l 第二个红框是我们配置的advice，除了before以外，还加了一个around。**我们重点来看around，它的返回值是Object**。虽然匹配的JPoint是testMethod，其定义的返回值是void。但是AspectJ考虑的很周到**。在around里，可以设置返回值类型为Object来表示返回任意类型的返回值**。AspectJ在真正返回参数的时候，会自动进行转换。比如，假设inttestMethod定义了int作为返回值类型，我们在around里可以返回一个Integer，AspectJ会自动转换成int作为返回值。
* l 再看around中的//proceed()这句话。这代表调用真正的JPoint函数，即testMethod。由于这里我们屏蔽了proceed，所以testMethod真正的内容并未执行，故运行的时候空指针异常就不会抛出来。也就是说，我们完全截获了testMethod的运行，甚至可以任意修改它，让它执行别的函数都没有问题。。

注意：从技术上说，around是完全可以替代before和after的。图7中第二个红框还把after给注释掉了。如果不注释掉，编译时候报错，[error]circular advice precedence: can't determine precedence between two or morepieces of advice that apply to the same join point: method-execution(voidtest.Test$TestDerived.testMethod())（大家可以自己试试）。我猜测其中的原因是around和after冲突了。around本质上代表了目标JPoint，比如此处的testMethod。而after是testMethod之后执行。那么这个testMethod到底是around还是原testMethod呢？真是傻傻分不清楚！

（我觉得再加一些限制条件给after是可以避免这个问题的，但是没搞成功...）

advice讲完了。现在回顾下3.2节从开始到现在我们学到了哪些内容：

* l AspectJ中各种类型的JoinPoint，JPoint是一个程序的关键执行点，也是我们关注的重点。
* l pointcuts：提供了一种方法来选择目标JPoint。程序有很多JPoint，但是需要一种方法来让我们选择我们关注的JPoint。这个方法就是利用pointcuts来完成的。
* l 通过pointcuts选择了目标JPoint后，我们总得干点什么吧？这就用上了advice。advice包括好几种类型，一般情况下都够我们用了。

上面这些东西都有点像函数定义，在Java中，这些东西都是要放到一个class里的。在AspectJ中，也有类似的[**数据结构**](http://lib.csdn.net/base/datastructure)，叫aspect。

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. **public** aspect 名字 {//aspect关键字和class的功能一样，文件名以.aj结尾
2. pointcuts定义...
3. advice定义...
4. }

你看，通过这种方式，定义一个aspect类，就把相关的JPoint和advice包含起来，是不是形成了一个“关注面”？比如：

* l 我们定义一个LogAspect，在LogAspect中，我们在关键JPoint上设置advice，这些advice就是打印日志
* l 再定义一个SecurityCheckAspect，在这个Aspect中，我们在关键JPoint上设置advice，这些advice将检查调用app是否有权限。

通过这种方式，我们在原来的JPoint中，就不需要写log打印的代码，也不需要写权限检查的代码了。所有这些关注点都挪到对应的Aspectj文件中来控制。恩，这就是AOP的精髓。

注意，读者在把玩代码时候，一定会碰到AspectJ语法不熟悉的问题。所以请读者记得随时参考官网的文档。这里有一个官方的语法大全：

http://www.eclipse.org/aspectj/doc/released/quick5.pdf 或者官方的另外一个文档也可以：

http://www.eclipse.org/aspectj/doc/released/progguide/semantics.html

#### 3.2.4  参数传递和JPoint信息

##### （1） 参数传递

到此，AspectJ最基本的东西其实讲差不多了，但是在实际使用AspectJ的时候，你会发现前面的内容还欠缺一点，尤其是advice的地方：

l 前面介绍的advice都是没有参数信息的，而JPoint肯定是或多或少有参数的。而且advice既然是对JPoint的截获或者hook也好，肯定需要利用传入给JPoint的参数干点什么事情。比方所around advice，我可以对传入的参数进行检查，如果参数不合法，我就直接返回，根本就不需要调用proceed做处理。

往advice传参数比较简单，就是利用前面提到的this(),target(),args()等方法。另外，整个pointcuts和advice编写的语法也有一些区别。具体方法如下：

  先在pointcuts定义时候指定参数类型和名字

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. pointcut testAll(Test.TestDerived derived,**int** x):call(\*Test.TestDerived.testMethod(..))
2. && target(derived)&& args(x)

  注意上述pointcuts的写法，首先在testAll中定义参数类型和参数名。这一点和定义一个函数完全一样

  接着看target和args。此处的target和args括号中用得是参数名。而参数名则是在前面pointcuts中定义好的。这属于target和args的另外一种用法。

  注意，增加参数并不会影响pointcuts对JPoint的匹配，上面的pointcuts选择和

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. pointcut testAll():call(\*Test.TestDerived.testMethod(..)) && target(Test.TestDerived) &&args(**int**)是一样的

只不过我们需要把参数传入advice，才需要改造

接下来是修改advice：

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. Object around(Test.TestDerived derived,**int** x):testAll(derived,x){
2. System.out.println("     arg1=" + derived);
3. System.out.println("     arg2=" + x);
4. **return** proceed(derived,x); //注意，proceed就必须把所有参数传进去。
5. }

  advice的定义现在也和函数定义一样，把参数类型和参数名传进来。

  接着把参数名传给pointcuts，此处是testAll。注意，advice必须和使用的pointcuts在参数类型和名字上保持一致。

  然后在advice的代码中，你就可以引用参数了，比如derived和x，都可以打印出来。

总结，参数传递其实并不复杂，关键是得记住语法：

* l pointcuts修改：像定义函数一样定义pointcuts，然后在this,target或args中绑定参数名（注意，不再是参数类型，而是参数名）。
* l advice修改：也像定义函数一样定义advice，然后在冒号后面的pointcuts中绑定参数名（注意是参数名）
* l 在advice的代码中使用参数名。

##### （2） JoinPoint信息收集

我们前面示例中都打印出了JPoint的信息，比如当前调用的是哪个函数，JPoint位于哪一行代码。这些都属于JPoint的信息。AspectJ为我们提供如下信息：

* l thisJoinpoint对象：在advice代码中可直接使用。代表JPoint每次被触发时的一些动态信息，比如参数啊之类的、
* l thisJoinpointStatic对象：在advice代码中可直接使用，代表JPoint中那些不变的东西。比如这个JPoint的类型，JPoint所处的代码位置等。
* l thisEnclosingJoinPointStaticPart对象：在advice代码中可直接使用。也代表JPoint中不可变的部分，但是它包含的东西和JPoint的类型有关，比如对一个call类型JPoint而言，thisEnclosingJoinPointStaticPart代表包含调用这个JPoint的函数的信息。对一个handler类型的JPoint而言，它代表包含这个try/catch的函数的信息。

关于thisJoinpoint，建议大家直接查看API文档，非常简单。其地址位于http://www.eclipse.org/aspectj/doc/released/runtime-api/index.html。

## 四、使用AOP的例子

现在正式回到我们的AndroidAopDemo这个例子来。我们的目标是为AopDemoActivity的几个Activity生命周期函数加上log，另外为checkPhoneState加上权限检查。一切都用AOP来集中控制。

前面提到说AspectJ需要编写aj文件，然后把AOP代码放到aj文件中。但是在Android开发中，我建议不要使用aj文件。因为aj文件只有AspectJ编译器才认识，而Android编译器不认识这种文件。所以当更新了aj文件后，编译器认为源码没有发生变化，所以不会编译它。

当然，这种问题在其他不认识aj文件的java编译环境中也存在。所以，AspectJ提供了一种基于注解的方法来把AOP实现到一个普通的Java文件中。这样我们就把AOP当做一个普通的Java文件来编写、编译就好。

### 4.1  打印Log

马上来看AopDemoActivity对应的DemoAspect.java文件吧。先看输出日志第一版本：

[-->第一版本]

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. **package** com.androidaop.demo;
2. **import** android.util.Log;
3. **import** org.aspectj.lang.annotation.Aspect;
4. **import** org.aspectj.lang.annotation.Before;
5. **import** org.aspectj.lang.annotation.Pointcut;
6. **import** org.aspectj.lang.JoinPoint;
8. @Aspect   //必须使用@AspectJ标注，这样class DemoAspect就等同于 aspect DemoAspect了
9. **public** **class** DemoAspect {
10. staticfinal String TAG = "DemoAspect";
11. /\*
12. @Pointcut：pointcut也变成了一个注解，这个注解是针对一个函数的，比如此处的logForActivity()
13. 其实它代表了这个pointcut的名字。如果是带参数的pointcut，则把参数类型和名字放到
14. 代表pointcut名字的logForActivity中，然后在@Pointcut注解中使用参数名。
15. 基本和以前一样，只是写起来比较奇特一点。后面我们会介绍带参数的例子
16. \*/
17. @Pointcut("execution(\* com.androidaop.demo.AopDemoActivity.onCreate(..)) ||"
18. +"execution(\* com.androidaop.demo.AopDemoActivity.onStart(..))")
19. **public** **void** logForActivity(){};  //注意，这个函数必须要有实现，否则Java编译器会报错
21. /\*
22. @Before：这就是Before的advice，对于after，after -returning，和after-throwing。对于的注解格式为
23. @After，@AfterReturning，@AfterThrowing。Before后面跟的是pointcut名字，然后其代码块由一个函数来实现。比如此处的log。
24. \*/
25. @Before("logForActivity()")
26. **public** **void** log(JoinPoint joinPoint){
27. //对于使用Annotation的AspectJ而言，JoinPoint就不能直接在代码里得到多了，而需要通过
28. //参数传递进来。
29. Log.e(TAG, joinPoint.toShortString());
30. }
31. }

提示：如果开发者已经切到AndroidStudio的话，AspectJ注解是可以被识别并能自动补齐。

上面的例子仅仅是列出了onCreate和onStart两个函数的日志，如果想在所有的onXXX这样的函数里加上log，该怎么改呢？

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. @Pointcut("execution(\* \*..AopDemoActivity.on\*(..))")
2. **public** **void** logForActivity(){};

图8给出这个例子的执行结果：

|  |
| --- |
| 图8  AopDemoActivity执行结果 |

### 4.2  检查权限

#### 4.2.1  使用注解

检查权限这个功能的实现也可以采用刚才打印log那样，但是这样就没有太多意思了。我们玩点高级的。不过这个高级的玩法也是来源于现实需求：

* l 权限检查一般是针对API的，比如调用者是否有权限调用某个函数。
* l API往往是通过SDK发布的。一般而言，我们会在这个函数的注释里说明需要调用者声明哪些权限。
* l 然后我们在API检查调用者是不是申明了文档中列出的权限。

如果我有10个API，10个不同的权限，那么在10个函数的注释里都要写，太麻烦了。怎么办？这个时候我想到了注解。注解的本质是源代码的描述。权限声明，从语义上来说，其实是属于API定义的一部分，二者是一个统一体，而不是分离的。

Java提供了一些默认的注解，不过此处我们要使用自己定义的注解：

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. **package** com.androidaop.demo;
2. **import** java.lang.annotation.ElementType;
3. **import** java.lang.annotation.Retention;
4. **import** java.lang.annotation.RetentionPolicy;
5. **import** java.lang.annotation.Target;
7. //第一个@Target表示这个注解只能给函数使用
8. //第二个@Retention表示注解内容需要包含的Class字节码里，属于运行时需要的。
9. @Target(ElementType.METHOD)
10. @Retention(RetentionPolicy.RUNTIME)
11. **public** **@interface** SecurityCheckAnnotation {//@interface用于定义一个注解。
12. publicString declaredPermission();  //declarePermssion是一个函数，其实代表了注解里的参数
13. }
14. 怎么使用注解呢？接着看代码：
15. //为checkPhoneState使用SecurityCheckAnnotation注解，并指明调用该函数的人需要声明的权限
16. @SecurityCheckAnnotation(declaredPermission="android.permission.READ\_PHONE\_STATE")
17. **private** **void** checkPhoneState(){
18. //如果不使用AOP，就得自己来检查权限
19. **if**(checkPermission("android.permission.READ\_PHONE\_STATE") ==**false**){
20. Log.e(TAG,"have no permission to read phone state");
21. **return**;
22. }
23. Log.e(TAG,"Read Phone State succeed");
24. **return**;
25. }

#### 4.2.2  检查权限

下面，我们来看看如何在AspectJ中，充分利用这注解信息来帮助我们检查权限。

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. /\*
2. 来看这个Pointcut，首先，它在选择Jpoint的时候，把@SecurityCheckAnnotation使用上了，这表明所有那些public的，并且携带有这个注解的API都是目标JPoint
3. 接着，由于我们希望在函数中获取注解的信息，所有这里的poincut函数有一个参数，参数类型是
4. SecurityCheckAnnotation，参数名为ann
5. 这个参数我们需要在后面的advice里用上，所以pointcut还使用了@annotation(ann)这种方法来告诉
6. AspectJ，这个ann是一个注解
7. \*/
8. @Pointcut("execution(@SecurityCheckAnnotation public \* \*..\*.\*(..)) && @annotation(ann)")
9. publicvoid checkPermssion(SecurityCheckAnnotation ann){};
11. /\*
12. 接下来是advice，advice的真正功能由check函数来实现，这个check函数第二个参数就是我们想要
13. 的注解。在实际运行过程中，AspectJ会把这个信息从JPoint中提出出来并传递给check函数。
14. \*/
15. @Before("checkPermssion(securityCheckAnnotation)")
16. publicvoid check(JoinPoint joinPoint,SecurityCheckAnnotationsecurityCheckAnnotation){
17. //从注解信息中获取声明的权限。
18. String neededPermission = securityCheckAnnotation.declaredPermission();
19. Log.e(TAG, joinPoint.toShortString());
20. Log.e(TAG, "\tneeded permission is " + neededPermission);
21. **return**;
22. }

如此这般，我们在API源码中使用的注解信息，现在就可以在AspectJ中使用了。这样，我们在源码中定义注释，然后利用AspectJ来检查。图9展示了执行的结果

|  |
| --- |
| 图9  权限检查的例子 |

#### 4.2.3  和其他模块交互

事情这样就完了？很明显没有。为什么？刚才权限检查只是简单得打出了日志，但是并没有真正去做权限检查。如何处理？这就涉及到AOP如何与一个程序中其他模块交互的问题了。初看起来容易，其实有难度。

比如，DemoAspect虽然是一个类，但是没有构造函数。而且，我们也没有在代码中主动去构造它。根据AsepctJ的说明，DemoAspect不需要我们自己去构造，AspectJ在编译的时候会把构造函数给你自动加上。具体在程序什么位置加上，其实是有规律的，但是我们并不知道，也不要去知道。

这样的话，DemoAspect岂不是除了打打log就没什么作用了？非也！以此例的权限检查为例，我们需要：

* l 把真正进行权限检查的地方封装到一个模块里，比如SecurityCheck中。
* l SecurityCheck往往在一个程序中只会有一个实例。所以可以为它提供一个函数，比如getInstance以获取SecurityCheck实例对象。
* l 我们就可以在DemoAspect中获取这个对象，然后调用它的check函数，把最终的工作由SecurityCheck来检查了。

恩，这其实是Aspect的真正作用，它负责收集Jpoint，设置advice。一些简单的功能可在Aspect中来完成，而一些复杂的功能，则只是有Aspect来统一收集信息，并交给专业模块来处理。

最终代码：

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. @Before("checkPermssion(securityCheckAnnotation)")
2. publicvoid check(JoinPoint joinPoint,SecurityCheckAnnotation securityCheckAnnotation){
3. String neededPermission = securityCheckAnnotation.declaredPermission();
4. Log.e(TAG, "\tneeded permission is " + neededPermission);
5. SecurityCheckManager manager =SecurityCheckManager.getInstanc();
6. **if**(manager.checkPermission(neededPermission) == **false**){
7. **throw** **new** SecurityException("Need to declare permission:" + neededPermission);
8. }
9. **return**;
10. }

图10所示为最终的执行结果。

|  |
| --- |
| 图10  执行真正的权限检查 |

注意，

* 1 所有代码都放到https://code.csdn.net/Innost/androidaopdemo上了....
* 2  编译：请在ubuntu下使用gradle assemble。编译结果放在out/apps/目录下。关于gradle的使用，请大家参考我的另外一篇重磅文章http://blog.csdn.net/innost/article/details/48228651

## 五、其他、总结和参考文献

最后我们来讲讲其他一些内容。首先是AspectJ的编译。

### 5.1  AspectJ编译

* l AspectJ比较强大，除了支持对source文件（即aj文件、或@AspectJ注解的Java文件，或普通java文件）直接进行编译外，
* l 还能对Java字节码（即对class文件）进行处理。有感兴趣的同学可以对aspectj-test小例子的class文件进行反编译，你会发现AspectJ无非是在被选中的JPoint的地方加一些hook函数。当然Before就是在调用JPoint之前加，After就是在JPoint返回之前加。
* l 更高级的做法是当class文件被加载到虚拟机后，由虚拟机根据AOP的规则进行hook。

在Android里边，我们用得是第二种方法，即对class文件进行处理。来看看代码：

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. //AndroidAopDemo.build.gradle
2. //此处是编译一个App，所以使用的applicationVariants变量，否则使用libraryVariants变量
3. //这是由Android插件引入的。所以，需要import com.android.build.gradle.AppPlugin;
4. android.applicationVariants.all { variant ->
5. /\*
6. 这段代码之意是：
7. 当app编译个每个variant之后，在javaCompile任务的最后添加一个action。此action
8. 调用ajc函数，对上一步生成的class文件进行aspectj处理。
9. \*/
10. AppPluginplugin = project.plugins.getPlugin(AppPlugin)
11. JavaCompile javaCompile = variant.javaCompile
12. javaCompile.doLast{
13. String bootclasspath =plugin.project.android.bootClasspath.join(File.pathSeparator)
14. //ajc是一个函数，位于utils.gradle中
15. ajc(bootclasspath,javaCompile)
16. }
17. }

ajc函数其实和我们手动试玩aspectj-test目标一样，只是我们没有直接调用ajc命令，而是利用AspectJ提供的API做了和ajc命令一样的事情。

**[java]** [view plain](http://blog.csdn.net/innost/article/details/49387395) [copy](http://blog.csdn.net/innost/article/details/49387395)

1. **import** org.aspectj.bridge.IMessage
2. **import** org.aspectj.bridge.MessageHandler
3. **import** org.aspectj.tools.ajc.Main
5. def ajc(String androidbootClassFiles,JavaCompile javaCompile){
6. String[] args = ["-showWeaveInfo",
7. "-1.8", //1.8是为了兼容java 8。请根据自己java的版本合理设置它
8. "-inpath",javaCompile.destinationDir.toString(),
9. "-aspectpath",javaCompile.classpath.asPath,
10. "-d",javaCompile.destinationDir.toString(),
11. "-classpath",javaCompile.classpath.asPath,
12. "-bootclasspath", androidbootClassFiles]
13. MessageHandlerhandler = **new** MessageHandler(**true**);
14. **new** Main().run(args,handler)
16. deflog = project.logger
17. **for**(IMessage message : handler.getMessages(**null**, **true**)) {
18. **switch** (message.getKind()) {
19. **case** IMessage.ABORT:
20. **case** IMessage.ERROR:
21. **case** IMessage.FAIL:
22. log.error message.message, message.thrown
23. **throw** message.thrown
24. **break**;
25. **case** IMessage.WARNING:
26. **case** IMessage.INFO:
27. log.info message.message, message.thrown
28. **break**;
29. **case** IMessage.DEBUG:
30. log.debug message.message, message.thrown
31. **break**;
32. }
33. }
34. }

主要利用了https://eclipse.org/aspectj/doc/released/devguide/ajc-ref.html中TheAspectJ compiler API一节的内容。由于代码已经在csdn git上，大家下载过来直接用即可。

### 5.2  总结

除了hook之外，AspectJ还可以为目标类添加变量。另外，AspectJ也有抽象，继承等各种更高级的玩法。根据本文前面的介绍，这些高级玩法一定要靠需求来驱动。AspectJ肯定对原程序是有影响的，如若贸然使用高级用法，则可能带来一些未知的后果。关于这些内容，读者根据情况自行阅读文后所列的参考文献。

最后再来看一个图。

|  |
| --- |
| 图11 未使用AOP的情况 |

图11中，左边是一个程序的三个基于OOP而划分的模块（也就是concern）。安全、业务逻辑、交易管理。这三个模块在设计图上一定是互相独立，互不干扰的。

但是在右图实现的时候，这三个模块就搅在一起了。这和我们在AndroidAopDemo中检查权限的例子中完全一样。在业务逻辑的时候，需要显示调用安全检查模块。

自从有了AOP，我们就可以去掉业务逻辑中显示调用安全检查的内容，使得代码归于干净，各个模块又能各司其职。而这之中千丝万缕的联系，都由AOP来连接和管理，岂不美哉？！

### 5.3  参考文献

[1]  Manning.AspectJ.in.Action第二版

看书还是要挑简单易懂的，AOP概念并不复杂，而AspectJ也有很多书，但是真正写得通俗易懂的就是这本，虽然它本意是介绍Spring中的AOP，但对AspectJ的解释真得是非常到位，而且还有对@AspectJ注解的介绍。本文除第一个图外，其他参考用图全是来自于此书。

[2]  http://fernandocejas.com/2014/08/03/aspect-oriented-programming-in-android/

Android中如何使用AspectJ，最重要的是它教会我们怎么使用aspectj编译工具API。

**关于this 跟 target在pointcut里面的区别**

首先我们要明确，aspectj实现的方式在于在编译的时候将advice插入joinpoint所处的位置

比如有两个类

Class BeinvokeClass{

public void testmethod(){

println(“方法执行”);

}

}

Class InvokeClass{

void invoke(){

BeinvokeClass obj=new BeinvokeClass();

obj.testmethod();

}

}

那么对于

pointcut point1:call(\* \*.testmethod(..))

before():point1(){

println(“方法调用前插入”)

}

after():point1(){

println(“方法调用后插入”)

}

aspectj实现这两个advice织入的方式无非是在所有 testmethod调用的前后加入两个advice代码

Class InvokeClass{

void invoke(){

BeinvokeClass obj=new BeinvokeClass();

println(“方法调用前插入”);obj.testmethod();println(“方法调用后插入”);

}

}

此时，对于before 和after两个点，this是 InvokeClass的对象 而 testmethod所处的对象就是target 也就是 BeinvokeClass的对象

而对于

pointcut point2:execution(\* \*.testmethod(..))

before():point2(){

println(“方法执行前插入”)

}

after():point2(){

println(“方法执行后插入”)

}

aspectj实现这两个advice织入的方式无非是在 testmethod执行的前后加入两个advice代码

Class BeinvokeClass{

public void testmethod(){println(“方法执行前插入”);

println(“方法执行”);

println(“方法执行后插入”)}

}

此时this 和target都是 BeinvokeClass

**关于通配符\* .. +的使用**

1 \* 匹配任意数量字符

2 .. 匹配任意数量字符的重复，如在类型模式中匹配任何数量子包；而在方法参数模式中匹配任何数量参数。

例如：

pointcut point2:execution(\* \*.testmethod(..))

这里..匹配任意数量参数

pointcut point3:execution(\* com..testmethod(..))

第一个..匹配任意数量的子包

**3 +** 匹配指定类型的子类型；仅能作为后缀放在类型模式后边。

例如 "execution(\* \*..BaseActivity+.onCreate(..))"

这里的+匹配任意 BaseActivity的子类型

五、args：使用“args(参数类型列表)”匹配当前执行的方法传入的参数为指定类型的执行方法；注意是匹配传入的参数类型，不是匹配方法签名的参数类型；参数类型列表中的参数必须是类型全限定名，通配符不支持；args属于动态切入点，这种切入点开销非常大，非特殊情况最好不要使用；

|  |  |
| --- | --- |
| 模式 | 描述 |
| args (java.io.Serializable,..) | 任何一个以接受“传入参数类型为 java.io.Serializable” 开头，且其后可跟任意个任意类型的参数的方法执行，args指定的参数类型是在运行时动态匹配的 |

六、@within：使用“@within(注解类型)”匹配所以持有指定注解类型内的方法；注解类型也必须是全限定类型名；

|  |  |
| --- | --- |
| 模式 | 描述 |
| @within cn.javass.spring.chapter6.Secure) | 任何目标对象对应的类型持有Secure注解的类方法；  必须是在目标对象上声明这个注解，在接口上声明的对它不起作用 |

七、@target：使用“@target(注解类型)”匹配当前目标对象类型的执行方法，其中目标对象持有指定的注解；注解类型也必须是全限定类型名；

|  |  |
| --- | --- |
| 模式 | 描述 |
| @target (cn.javass.spring.chapter6.Secure) | 任何目标对象持有Secure注解的类方法；  必须是在目标对象上声明这个注解，在接口上声明的对它不起作用 |

八、@args：使用“@args(注解列表)”匹配当前执行的方法传入的参数持有指定注解的执行；注解类型也必须是全限定类型名；

|  |  |
| --- | --- |
| 模式 | 描述 |
| @args (cn.javass.spring.chapter6.Secure) | 任何一个只接受一个参数的方法，且方法运行时传入的参数持有注解 cn.javass.spring.chapter6.Secure；动态切入点，类似于arg指示符； |

九、@annotation：使用“@annotation(注解类型)”匹配当前执行方法持有指定注解的方法；注解类型也必须是全限定类型名；

|  |  |
| --- | --- |
| 模式 | 描述 |
| @annotation(cn.javass.spring.chapter6.Secure ) | 当前执行方法上持有注解 cn.javass.spring.chapter6.Secure将被匹配 |

## 通知参数

## 6.6  通知参数

       前边章节已经介绍了声明通知，但如果想获取被被通知方法参数并传递给通知方法，该如何实现呢？接下来我们将介绍两种获取通知参数的方式。

* 使用JoinPoint获取：Spring AOP提供使用org.aspectj.lang.JoinPoint类型获取连接点数据，任何通知方法的第一个参数都可以是JoinPoint(环绕通知是ProceedingJoinPoint，JoinPoint子类)，当然第一个参数位置也可以是JoinPoint.StaticPart类型，这个只返回连接点的静态部分。

1) JoinPoint：提供访问当前被通知方法的目标对象、代理对象、方法参数等数据：

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/281.html)

1. **package** org.aspectj.lang;
2. **import** org.aspectj.lang.reflect.SourceLocation;
3. **public** **interface** JoinPoint {
4. String toString();         //连接点所在位置的相关信息
5. String toShortString();     //连接点所在位置的简短相关信息
6. String toLongString();     //连接点所在位置的全部相关信息
7. Object getThis();         //返回AOP代理对象
8. Object getTarget();       //返回目标对象
9. Object[] getArgs();       //返回被通知方法参数列表
10. Signature getSignature();  //返回当前连接点签名
11. SourceLocation getSourceLocation();//返回连接点方法所在类文件中的位置
12. String getKind();        //连接点类型
13. StaticPart getStaticPart(); //返回连接点静态部分
14. }

2）ProceedingJoinPoint：用于环绕通知，使用proceed()方法来执行目标方法：

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/281.html)

1. **public** **interface** ProceedingJoinPoint **extends** JoinPoint {
2. **public** Object proceed() **throws** Throwable;
3. **public** Object proceed(Object[] args) **throws** Throwable;
4. }

3) JoinPoint.StaticPart：提供访问连接点的静态部分，如被通知方法签名、连接点类型等：

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/281.html)

1. **public** **interface** StaticPart {
2. Signature getSignature();    //返回当前连接点签名
3. String getKind();          //连接点类型
4. **int** getId();               //唯一标识
5. String toString();         //连接点所在位置的相关信息
6. String toShortString();     //连接点所在位置的简短相关信息
7. String toLongString();     //连接点所在位置的全部相关信息
8. }

使用如下方式在通知方法上声明，必须是在第一个参数，然后**使用jp.getArgs()就能获取到被通知方法参数：**

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/281.html)

1. @Before(value="execution(\* sayBefore(\*))")
2. **public** **void** before(JoinPoint jp) {}
4. @Before(value="execution(\* sayBefore(\*))")
5. **public** **void** before(JoinPoint.StaticPart jp) {}

* 自动获取：通过切入点表达式可以将相应的参数自动传递给通知方法，例如前边章节讲过的返回值和异常是如何传递给通知方法的。

在Spring AOP中，除了execution和bean指示符不能传递参数给通知方法，其他指示符都可以将匹配的相应参数或对象自动传递给通知方法。

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/281.html)

1. @Before(value="execution(\* test(\*)) && args(param)", argNames="param")
2. **public** **void** before1(String param) {
3. System.out.println("===param:" + param);
4. }

       切入点表达式execution(\* test(\*)) && args(param) ：

**1）首先execution(\* test(\*))匹配任何方法名为test，且有一个任何类型的参数；**

**2）args(param)将首先查找通知方法上同名的参数，并在方法执行时（运行时）匹配传入的参数是使用该同名参数类型，即java.lang.String；如果匹配,将把该被通知方法的参数传递给通知方法上同名参数。**

其他指示符（除了execution和bean指示符）都可以使用这种方式进行参数绑定。

在此有一个问题，即前边提到的类似于【3.1.2构造器注入】中的参数名注入限制：在class文件中没生成变量调试信息是获取不到方法参数名字的。

所以我们可以使用策略来确定参数名：

1. 如果我们通过“argNames”属性指定了参数名，那么就是要我们指定的；

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/281.html)

1. @Before(value=" args(param)", argNames="param") //明确指定了
2. **public** **void** before1(String param) {
3. System.out.println("===param:" + param);
4. }

1. 如果第一个参数类型是JoinPoint、ProceedingJoinPoint或JoinPoint.StaticPart类型，应该从“argNames”属性省略掉该参数名（可选，写上也对），这些类型对象会自动传入的，但必须作为第一个参数；

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/281.html)

1. @Before(value=" args(param)", argNames="param") //明确指定了
2. **public** **void** before1(JoinPoint jp, String param) {
3. System.out.println("===param:" + param);
4. }

1. 如果“class文件中含有变量调试信息”将使用这些方法签名中的参数名来确定参数名；

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/281.html)

1. @Before(value=" args(param)") //不需要argNames了
2. **public** **void** before1(JoinPoint jp, String param) {
3. System.out.println("===param:" + param);
4. }

1. 如果没有“class文件中含有变量调试信息”，将尝试自己的参数匹配算法，如果发现参数绑定有二义性将抛出AmbiguousBindingException异常；对于只有一个绑定变量的切入点表达式，而通知方法只接受一个参数，说明绑定参数是明确的，从而能配对成功。

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/281.html)

1. @Before(value=" args(param)")
2. **public** **void** before1(JoinPoint jp, String param) {
3. System.out.println("===param:" + param);
4. }

1. 以上策略失败将抛出IllegalArgumentException。

接下来让我们示例一下组合情况吧：

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/281.html)

1. @Before(args(param) && target(bean) && @annotation(secure)",
2. argNames="jp,param,bean,secure")
3. **public** **void** before5(JoinPoint jp, String param,
4. IPointcutService pointcutService, Secure secure) {
5. ……
6. }

              该示例的执行步骤如图6-5所示。

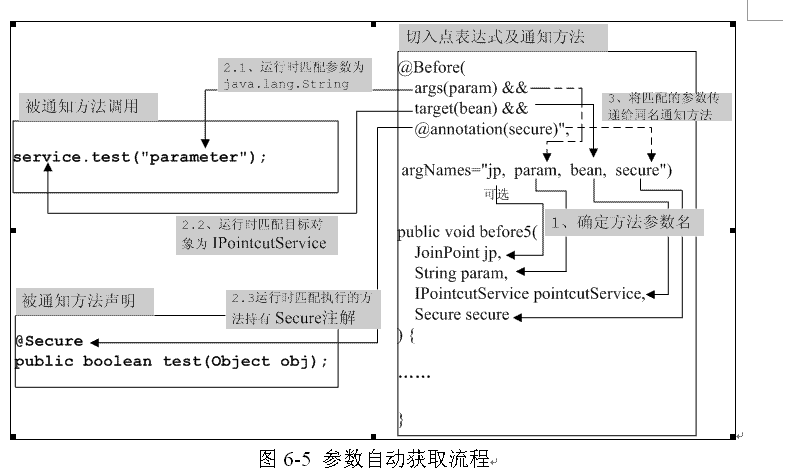


图6-5 参数自动获取流程

除了上边介绍的普通方式，也可以对使用命名切入点自动获取参数：

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/281.html)

1. @Pointcut(value="args(param)", argNames="param")
2. **private** **void** pointcut1(String param){}
3. @Pointcut(value="@annotation(secure)", argNames="secure")
4. **private** **void** pointcut2(Secure secure){}
6. @Before(value = "pointcut1(param) && pointcut2(secure)",
7. argNames="param, secure")
8. **public** **void** before6(JoinPoint jp, String param, Secure secure) {
9. ……
10. }

       自此给通知传递参数已经介绍完了，示例代码在cn.javass.spring.chapter6.ParameterTest文件中。