深入理解 Android 之 AOP

一、闲谈 AOP

大家都知道 OOP，即 Object Oriented 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：

- **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！
- 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：

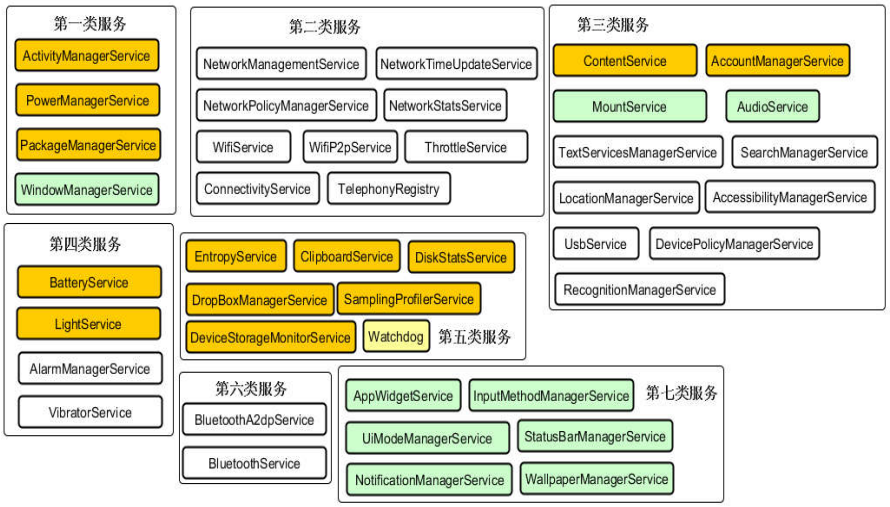

图 1　Android Framework 中的模块

图 1 中所示为 Android Framework 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。

- 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：

- 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。
- 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：

- Activity 的生命周期的几个函数运行时，要输出日志。
- 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/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]

```
public class AopDemoActivity extends Activity {

    private static final String TAG = "AopDemoActivity";

❶  onCreate，onStart,onRestart,onPause,onResume,onStop,onDestory 返回前，都输出一行日志

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.layout_main);
        Log.e(TAG,"onCreate");
    }
    protected void onStart() {
        super.onStart();
        Log.e(TAG, "onStart");
    }
    protected void onRestart() {
        super.onRestart();
        Log.e(TAG, "onRestart");
    }
    protected void onResume() {
        super.onResume();
        Log.e(TAG, "onResume");

❷   checkPhoneState 会检查 app 是否申明了 android.permission.READ_PHONE_STATE 权限

        checkPhoneState();
    }
    protected void onPause() {
        super.onPause();
        Log.e(TAG, "onPause");
    }
```

```
    protected void onStop() {
        super.onStop();
        Log.e(TAG, "onStop");
    }
    protected void onDestroy() {
        super.onDestroy();
        Log.e(TAG, "onDestroy");
    }
    private void checkPhoneState(){
        if(checkPermission("android.permission.READ_PHONE_STATE") == false){
            Log.e(TAG,"have no permission to read phone state");
            return;
        }
        Log.e(TAG,"Read Phone State succeed");
        return;
    }
    private boolean checkPermission(String permissionName){
        try{
            PackageManager pm = getPackageManager();
           //调用 PackageMangaer 的 checkPermission 函数，检查自己是否申明使用某权限
            int nret = pm.checkPermission(permissionName,getPackageName());
            return nret == PackageManager.PERMISSION_GRANTED;
        }.....
    }
}
```

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：

- 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。

AspectJ 现在托管于 Eclipse 项目中，官方网站是：

- http://www.eclipse.org/aspectj/ **<=AspectJ 官方网站**
- http://www.eclipse.org/aspectj/doc/released/runtime-api/index.html **<=AspectJ 类库参考文档，内容非常少**
- 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。

[1] 关于 Groovy 更多的故事，请阅读我的博客《深入理解 Android 之 Gradle》http://blog.csdn.net/innost/article/details/48228651

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：

- 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。
- 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。
- 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。

```
1  package test;
2  public class Test{
3     static public class TestBase{
4          static{
5            int x = 0;
6          }
7          int base = 0;
8          public TestBase(int index){
9              base = index;
10         }
11    }
12      static  public class TestDerived extends TestBase{
13         public int derived = 0;
14         public TestDerived(){
15             super(0);
16             this.derived = 1000;
17
18         }
19         public void testMethod() {
20             try{
21                 byte[] test = null;
22                 test[1] = 0x33;
23             }catch(Exception ex){
24             }
25         }
26         static int getFixedIndex(){
27            return 1000;
28         }
29     }
30      public static void main(String args[]){
31         System.out.println("Test begin...");
32         TestDerived derived = new TestDerived();
33         derived.testMethod();
34         derived.base = 1;
35         System.out.println("Test end...");
36     }
37 }
```

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：

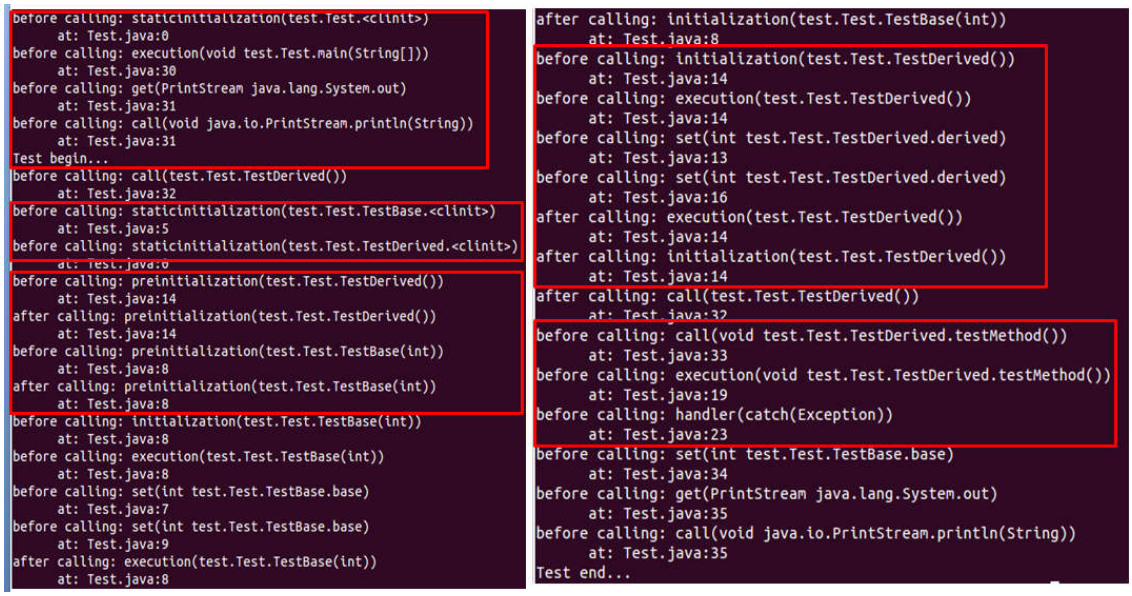

图 3　所有的 join points

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：

- **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）。
- Test 类初始化完后，就该执行 main 函数了。所以，下一个 JPoint 就是 **execution(void test.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！
- 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：

- testMethod 的 call 类型 JPoint
- testMethod 的 execution 类型 JPonint
- 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（static block）。

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 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/Innost/androidaopdemo** 上。请小伙伴们自己下载玩耍。具体的操作方法是：

- 下载得到 androidaopdemo 中，有一个 aspectj-test 目录。
- 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**。
- 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。
- 执行 create-jar.sh。它会编译几个文件，然后生成 test.jar。
- 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 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：**pointcut testAll():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：

```
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

1. 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。
2. 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。
3. 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。

④ 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。

⑤ 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。

⑥ 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。

⑦ 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。

⑧ 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：

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

图 4 展示了执行结果：

```
before calling: call(void java.io.PrintStream.println(String))
        at: Test.java:33
Test begin...
before calling: call(void test.Test.println())
      at: Test.java:37
before calling: call(void java.io.PrintStream.println(String))
        at: Test.java:38
Test end...
```

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：

Join point category	Pointcut syntax
Method execution	`execution(MethodSignature)`
Method call	`call(MethodSignature)`
Constructor execution	`execution(ConstructorSignature)`
Constructor call	`call(ConstructorSignature)`
Class initialization	`staticinitialization(TypeSignature)`
Field read access	`get(FieldSignature)`
Field write access	`set(FieldSignature)`
Exception handler execution	`handler(TypeSignature)`
Object initialization	`initialization(ConstructorSignature)`
Object pre-initialization	`preinitialization(ConstructorSignature)`
Advice execution	`adviceexecution()`

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 method execution 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：

@注解 访问权限 返回值的类型 包名.函数名(参数)

❶ @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。

❷ 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

❸ 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。

比如：

java.*.Date：可以表示 java.sql.Date，也可以表示 java.util.Date

Test*：可以表示 TestBase，也可以表示 TestDervied

java..*：表示 java 任意子类

java..*Model+：表示 Java 任意 package 中名字以 Model 结尾的子类，比如 TabelModel，TreeModel 等

④ 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：

(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

(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，..代表任意参数个数和类型

(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？

Constructor_signature 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：

public *..TestDerived.new(..)：

① public：选择 public 访问权限

② *..代表任意包名

③ TestDerived.new：代表 TestDerived 的构造函数

④ (..)：代表参数个数和类型都是任意

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：

Field Signature 标准格式：

@注解 访问权限 类型 类名.成员变量名

① 其中，@注解和访问权限是可选的

② 类型：成员变量类型，*代表任意类型

③ 类名.成员变量名：成员变量名可以是*，代表任意成员变量

比如，

set(int test..TestBase.base)：表示设置 TestBase.base 变量时的 JPoint

Type Signature：直接上例子

staticinitialization(test..TestBase)：表示 TestBase 类的 static block

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(ConstructorSignature|MethodSignature)	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：

- 进入 **androidaopdemo/aspectj-test** 目录。
- 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。
- 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：

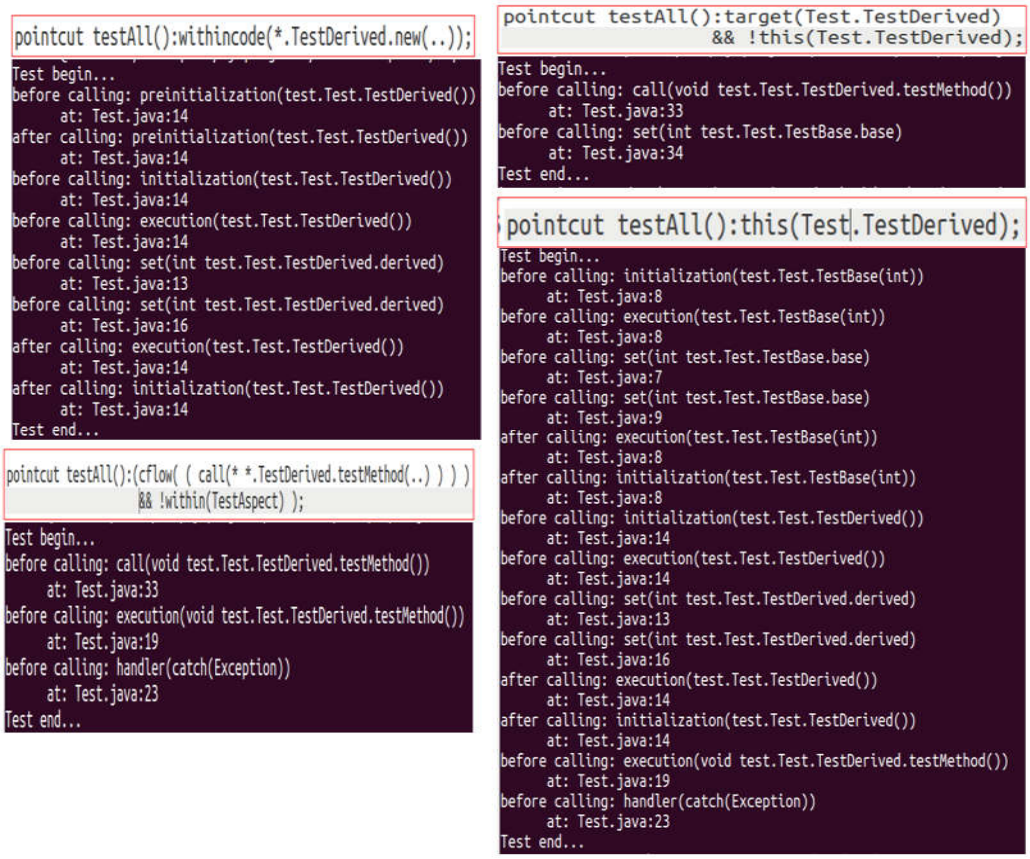

图 6　示例代码和输出结果

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：

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

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 around 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。

```
public void testMethod() {
    byte[] test = null;
    test[1] = 0x33;
    return;
}
```

```
pointcut testAll():execution(* *.TestDerived.testMethod(..));

before():testAll(){
        System.out.println("before calling: " + thisJoinPoint);
        System.out.println("      at: " + thisJoinPoint.getSourceLocation());
}
/*after():testAll() {
        Signature sig = thisJoinPoint.getSignature();
        if(sig instanceof ConstructorSignature){
           System.out.println("after calling: " + thisJoinPoint);
           System.out.println("      at: " + thisJoinPoint.getSourceLocation());
        }

}*/
Object around():testAll(){
     System.out.println("Around:" + thisJoinPoint);
     System.out.println("      at: " + thisJoinPoint.getSourceLocation());
     //proceed();
     return null;
  }
```

```
Test begin...
before calling: execution(void test.Test.TestDerived.testMethod())
      at: Test.java:19
Around:execution(void test.Test.TestDerived.testMethod())
      at: Test.java:19
Test end...
```

图 7　advice 示例和结果

图 7 中：

- 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。
- 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 **int testMethod** 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。
- 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 more pieces of advice that apply to the same join point: method-execution(void test.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：

- 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。
- 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。
- 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，叫 aspect。

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

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：

- 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
- 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：

- 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：

❶ 先在 pointcuts 定义时候指定参数类型和名字

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

❷ 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

❸ 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。

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

```
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：

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

❺ 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。

❻ 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。

❼ 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：

- 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）。
- 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）
- 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：

- 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、
- 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。
- 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]

```
package com.androidaop.demo;

import android.util.Log;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.Pointcut;

import org.aspectj.lang.JoinPoint;


@Aspect     //必须使用@AspectJ 标注，这样 class DemoAspect 就等同于  aspect DemoAspect 了

public class DemoAspect {

      static final String TAG = "DemoAspect";

/*

@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()

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

代表 pointcut 名字的 logForActivity 中，然后在@Pointcut 注解中使用参数名。

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

*/

@Pointcut("execution(* com.androidaop.demo.AopDemoActivity.onCreate(..)) ||"

            + "execution(* com.androidaop.demo.AopDemoActivity.onStart(..))")

public void logForActivity(){};   //注意，这个函数必须要有实现，否则 Java 编译器会报错


/*

@Before：这就是 Before 的 advice，对于 after，after -returning，和 after-throwing。对于的注解格式为

@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。

*/

      @Before("logForActivity()")
```

```
    public void log(JoinPoint joinPoint){
        //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
        //参数传递进来。
            Log.e(TAG, joinPoint.toShortString());
    }
}
```

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？

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

图 8 给出这个例子的执行结果：

```
10-24 09:46:19.919   579  1194 I ActivityManager: START u0 {act=android.intent.action.MAIN cat=[a
ty (has extras)} from uid 10015 on display 0
10-24 09:46:19.969   579   798 I ActivityManager: Start proc 3654:com.androidaop.demo/u0a6 for ac
10-24 09:46:20.043  3654  3654 E DemoAspect: execution(AopDemoActivity.onCreate(..))
10-24 09:46:20.054  3654  3654 E AopDemoActivity: onCreate
10-24 09:46:20.055  3654  3654 E DemoAspect: execution(AopDemoActivity.onStart())
10-24 09:46:20.055  3654  3654 E AopDemoActivity: onStart
10-24 09:46:20.055  3654  3654 E DemoAspect: execution(AopDemoActivity.onResume())
10-24 09:46:20.055  3654  3654 E AopDemoActivity: onResume
10-24 09:46:20.055  3654  3654 E AopDemoActivity: Read Phone State succeed
10-24 09:46:20.239   579   597 I ActivityManager: Displayed com.androidaop.demo/.AopDemoActivity:
10-24 09:46:23.729  3654  3654 E DemoAspect: execution(AopDemoActivity.onPause())
10-24 09:46:23.730  3654  3654 E AopDemoActivity: onPause
10-24 09:46:23.745  3654  3654 E DemoAspect: execution(AopDemoActivity.onStop())
10-24 09:46:23.745  3654  3654 E AopDemoActivity: onStop
10-24 09:46:23.746  3654  3654 E DemoAspect: execution(AopDemoActivity.onDestroy())
10-24 09:46:23.746  3654  3654 E AopDemoActivity: onDestroy
```

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：

- 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。
- 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。

- 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：

```
package com.androidaop.demo;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;


//第一个@Target 表示这个注解只能给函数使用

//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。

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface SecurityCheckAnnotation {//@interface 用于定义一个注解。

    public String declaredPermission();    //declarePermssion 是一个函数，其实代表了注解里的参数

}
```

怎么使用注解呢？接着看代码：

```
//为 checkPhoneState 使用 SecurityCheckAnnotation 注解，并指明调用该函数的人需要声明的权限

    @SecurityCheckAnnotation(declaredPermission="android.permission.READ_PHONE_STATE")

    private void checkPhoneState(){

        //如果不使用 AOP，就得自己来检查权限

        if(checkPermission("android.permission.READ_PHONE_STATE") == false){

            Log.e(TAG,"have no permission to read phone state");

            return;

        }

        Log.e(TAG,"Read Phone State succeed");

        return;

    }
```

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。

```
/*
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
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
SecurityCheckAnnotation，参数名为 ann
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
AspectJ，这个 ann 是一个注解
*/
   @Pointcut("execution(@SecurityCheckAnnotation public * *..*.*(..)) && @annotation(ann)")
   public void checkPermssion(SecurityCheckAnnotation ann){};


/*
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
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。
*/
      @Before("checkPermssion(securityCheckAnnotation)")
      public void check(JoinPoint joinPoint,SecurityCheckAnnotation securityCheckAnnotation){
            //从注解信息中获取声明的权限。
            String neededPermission = securityCheckAnnotation.declaredPermission();
            Log.e(TAG, joinPoint.toShortString());
            Log.e(TAG, "\tneeded permission is " + neededPermission);
            return;
      }
```

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

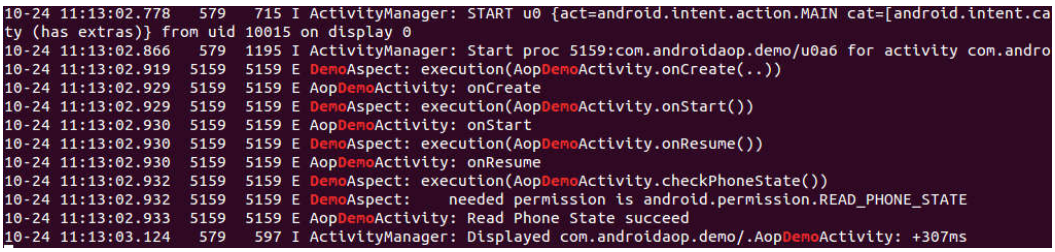

图 9　权限检查的例子

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：

- 把真正进行权限检查的地方封装到一个模块里，比如 SecurityCheck 中。
- 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。
- 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：

```
@Before("checkPermssion(securityCheckAnnotation)")
public void check(JoinPoint joinPoint,SecurityCheckAnnotation securityCheckAnnotation){
    String neededPermission = securityCheckAnnotation.declaredPermission();
    Log.e(TAG, "\tneeded permission is " + neededPermission);
```

```
        SecurityCheckManager manager = SecurityCheckManager.getInstanc();
        if(manager.checkPermission(neededPermission) == false){
            throw new SecurityException("Need to declare permission:" + neededPermission);
        }
        return;
    }
```

图 10 所示为最终的执行结果。

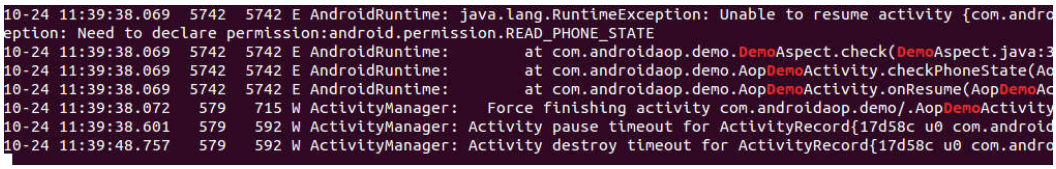

图 10　执行真正的权限检查

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

- 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，
- 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。
- 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：

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

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。

```
import org.aspectj.bridge.IMessage
import org.aspectj.bridge.MessageHandler
import org.aspectj.tools.ajc.Main

def ajc(String androidbootClassFiles,JavaCompile javaCompile){
     String[] args = ["-showWeaveInfo",
                      "-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
                      "-inpath", javaCompile.destinationDir.toString(),
                      "-aspectpath", javaCompile.classpath.asPath,
                      "-d", javaCompile.destinationDir.toString(),
                      "-classpath", javaCompile.classpath.asPath,
```

```
                            "-bootclasspath", androidbootClassFiles]
    MessageHandler handler = new MessageHandler(true);
    new Main().run(args, handler)


    def log = project.logger
    for (IMessage message : handler.getMessages(null, true)) {
        switch (message.getKind()) {
            case IMessage.ABORT:
            case IMessage.ERROR:
            case IMessage.FAIL:
                log.error message.message, message.thrown
                throw message.thrown
                break;
            case IMessage.WARNING:
            case IMessage.INFO:
                log.info message.message, message.thrown
                break;
            case IMessage.DEBUG:
                log.debug message.message, message.thrown
                break;
        }
    }
}
```

主要利用了 **https://eclipse.org/aspectj/doc/released/devguide/ajc-ref.html** 中 **The AspectJ 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。

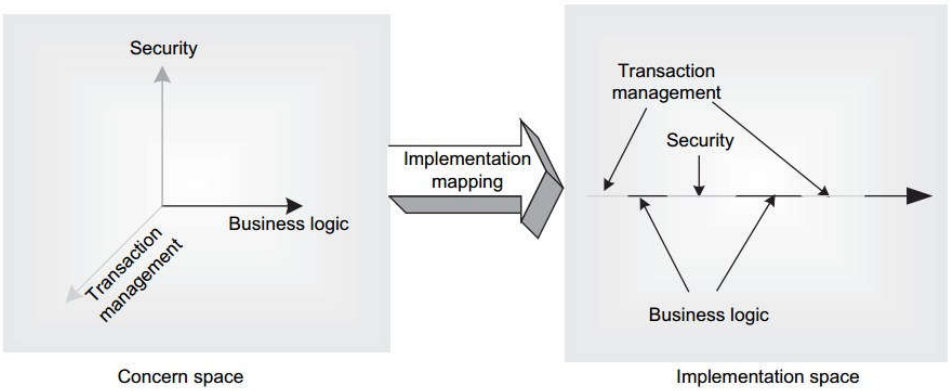

图 11 未使用 AOP 的情况

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。

[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。