AOP之AspectJ在Android中的应用


Hilary_Lu 关注 IP属地: 重庆
1 2018.03.26 15:20:37 字数 2,505 阅读 6,175

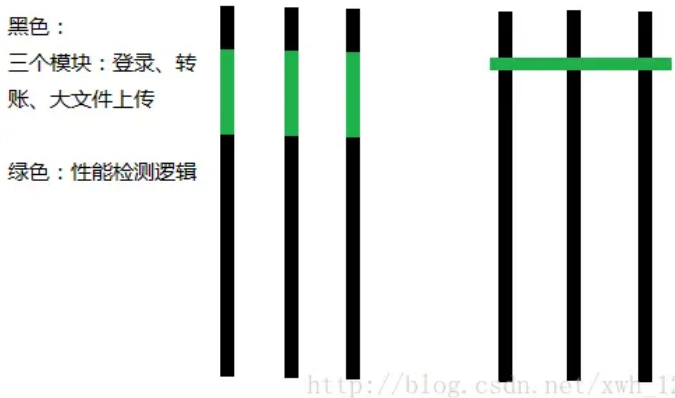
1 前言

1.1 什么是AOP，与OOP的区别

- OOP：即ObjectOriented Programming，面向对象编程。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。
- AOP：即Aspect Oriented Programming，面向切面编程。通过**预编译方式**和运行期动态代理实现程序功能的统一维护的一种技术。

	OOP	AOP
面向目标	面向名词领域	面向动词领域
思想结构	纵向结构	横向结构
注重方面	注重业务逻辑单元的划分	偏重业务处理过程的某个步骤或阶段

下图：有三个模块：登陆、转账、大文件上传，现在需要加入性能检测功能，统计这三个模块每个方法耗时多少，OOP思想做法是设计一个性能检测模块，提供接口供这三个模块调用。这样每个模块都要调用性能检测模块的接口，如果接口有改动，需要在这三个模块中每次调用的地方修改，这样做的弊端有：代码冗余，逻辑不清晰，重构不方便，违背单一原则。运用AOP的思想做法是：在这些独立的模块间，在特定的切入点进行hook，将共同的逻辑添加到模块中而不影响原有模块的独立性。如下图OOP实现转AOP实现，在不同的模块中加入性能检测功能，并不影响原有的架构。

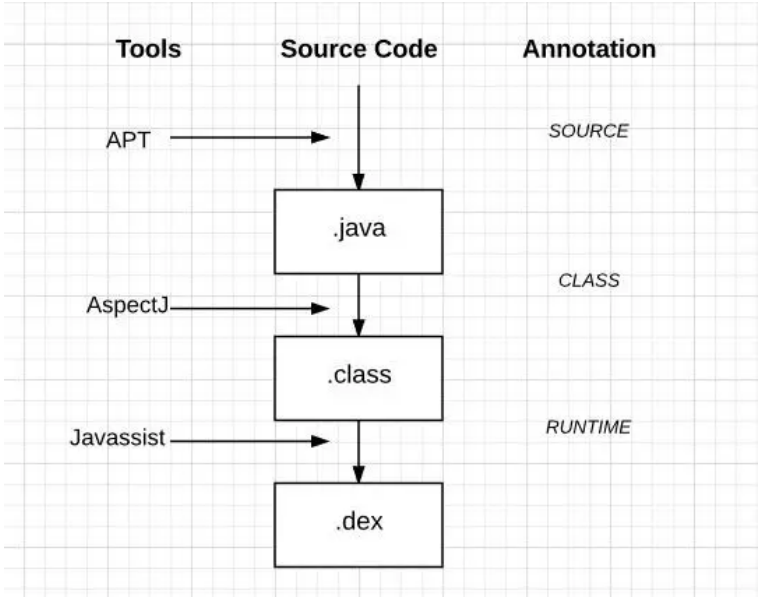


OOP实现转AOP实现.png

1.2 Android AOP主流框架

名称	描述
Xposed	ROOT社区著名开源项目，需要root权限（运行时）
Dexposed	阿里AOP框架，改造Xposed，只支持Android2.3 - 4.4（运行时）

名称	描述
APT	注解处理器，通过注解生成源代码，代表框架：DataBinding,Dagger2, ButterKnife, EventBus3 、DBFlow、AndroidAnnotation
AspectJ	AspectJ定义了AOP语法，所以它有一个专门的编译器用来生成遵守Java字节编码规范的Class文件，在编译期注入代码。代表框架：Hugo(Jake Wharton)
Javassist、ASM	执行字节码操作的库。它可以在一个已经编译好的类中添加新的方法，或者是修改已有的方法，可以绕过编译，直接操作字节码，从而实现代码注入。代表框架：热修复框架HotFix 、InstantRun



APT,AspectJ,Javassist对应的编译时期.jpg

2 AspectJ

2.1 环境配置

aspectJ常规配置

在app/build.gradle下配置：

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

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.aspectj:aspectjtools:1.8.9'
        classpath 'org.aspectj:aspectjweaver:1.8.9'
    }
}

variants.all { variant ->
    if (!variant.buildType.isDebuggable()) {
        log.debug("Skipping non-debuggable build type '${variant.buildType.name}'.")
        return;
    }

    JavaCompile javaCompile = variant.javaCompile
    javaCompile.doLast {
        String[] args = ["-showWeaveInfo",
                         "-1.8",
                         "-inpath", javaCompile.destinationDir.toString(),
                         "-aspectpath", javaCompile.classpath.asPath,
```

```
                        "-d", javaCompile.destinationDir.toString(),
                        "-classpath", javaCompile.classpath.asPath,
                        "-bootclasspath", project.android.bootClasspath.join(File.pat
        log.debug "ajc args: " + Arrays.toString(args)

        MessageHandler handler = new MessageHandler(true);
        new Main().run(args, handler);
        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
                    break;
                case IMessage.WARNING:
                    log.warn message.message, message.thrown
                    break;
                case IMessage.INFO:
                    log.info message.message, message.thrown
                    break;
                case IMessage.DEBUG:
                    log.debug message.message, message.thrown
                    break;
            }
        }
    }
}

dependencies {
    compile files('libs/aspectjrt.jar')  //将aspectjrt.jar包拷贝至app/libs目录下
}
```

aspectjx插件配置

在根build.gradle下配置：

```
 dependencies {
        classpath 'com.hujiang.aspectjx:gradle-android-plugin-aspectjx:1.0.10'
 }
```

在app/build.gradle下配置：

```
apply plugin: 'android-aspectjx'
dependencies {
    compile 'org.aspectj:aspectjrt:1.8.9'
}
```

aspectjx默认会遍历项目编译后所有的.class文件和依赖的第三方库去查找符合织入条件的切点，为了提升编译效率，可以加入过滤条件指定遍历某些库或者不遍历某些库。

```
aspectjx {
    //织入遍历符合条件的库
    includeJarFilter 'universal-image-loader', 'AspectJX-Demo/library'
    //排除包含‘universal-image-loader’的库
    excludeJarFilter 'universal-image-loader'
}
```

配置注意：

1. 黑名单必须放在app/build.gralde下才生效
2. app下的代码会自动加入白名单
3. gradle插件版本目前不支持3.0以上
4. 只支持AspectJ annotation的方式
5. 建议配置白名单，否则编译时会遇到冲突

两种配置区别

AspectJ常规配置不支持AAR或者JAR切入的，只会对编译的代码进行织入，AspectJX插件配置支持AAR, JAR及Kotlin的应用。这里需要注意的，在AspectJ常规配置中有这样的代码：`"-inpath", javaCompile.destinationDir.toString()`，代表只对源文件进行织入。在查看Aspectjx源码时，发现在“-inputs”配置加入了.jar文件，使得class类可以被织入代码。这么理解来看，AspectJ也是支持对class文件的织入的，只是需要对它进行相关的配置，而配置比较繁琐，所以诞生了AspectJx等插件。

aspectjx github链接点此

2.2 语法

通常AspectJ需要编写aj文件，然后把AOP代码放到aj后缀名文件中，如下：

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

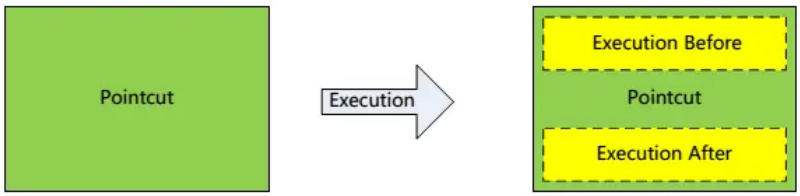
在Android开发中，建议不要使用aj文件。因为aj文件只有AspectJ编译器才认识，而Android编译器不认识这种文件。所以当更新了aj文件后，编译器认为源码没有发生变化，不会编译它。所以AspectJ提供了一种基于注解的方法，如下：

```
@Pointcut(“call(public  *  *.println(..)) && !within(TestAspect)")//方法切入点
public void testAll() { }
```

Join Points

Join Points 就是程序运行时的一些执行点，例如：我要打印所有Activity的onCreate方法，onCreate方法被调用就是一个Join Points。除了方法被调用，还有很多，例如方法内部“读、写”变量，异常处理等，如下表：

Join Point	说明	Pointcuts语法
Method call	方法被调用	call(MethodPattern)
Method execution	方法执行	execution(MethodPattern)
Constructor call	构造函数被调用	call(ConstructorPattern)
Constructor execution	构造函数执行	execution(ConstructorPattern)
Field get	读取属性	get(FieldPattern)
Field set	写入属性	set(FieldPattern)
Pre-initialization	与构造函数有关，很少用到	preinitialization(ConstructorPattern)
Initialization	与构造函数有关，很少用到	initialization(ConstructorPattern)
Static initialization	static 块初始化	staticinitialization(TypePattern)
Handler	异常处理	handler(TypePattern)
Advice execution	所有 Advice 执行	adviceexcution()



call和execution区别.png

Pointcuts

上表中，同一个函数，还分为call类型和execution类型的JPoint，如何选择自己想要的JPoint呢，这就是Pointcuts的功能：提供一种方法使得开发者能够选择自己感兴趣的JoinPoints。例如：我要打印所有Activity的onCreate方法，Pointcuts需要筛选的就是所有Activity的onCreate方法，而不是任意类的onCreate方法。除了上表与Join Point 对应的选择外，Pointcuts 还有其他选择方法：

Pointcuts 语法	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类中所有JPoint
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(Pointcut)	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)	Join Point 所属的 this 对象是否 instanceOf Type 或者 Id 的类型	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。

Pointcut 表达式还可以！、&&、|| 来组合，语义和java一样。上面 Pointcuts 的语法中涉及到一些 Pattern，下面是这些 Pattern 的规则，[]里的内容是可选的：

Pattern	规则
MethodPattern	[!] [@Annotation] [public,protected,private] [static] [final] 返回值类型 [类名.]方法名(参数类型列表) [throws 异常类型]
ConstructorP	[!] [@Annotation] [public,protected,private] [final] [类名.]new(参数类型列表) [throws 异常类型]

Pattern	规则
attern	
FieldPattern	[!] [@Annotation] [public,protected,private] [static] [final] 属性类型 [类名.]属性名
TypePattern	其他 Pattern 涉及到的类型规则也是一样，可以使用 '!'、''、'..'、'+'，'!' 表示取反，'' 匹配除 . 外的所有字符串，'*' 单独使用事表示匹配任意类型，'..' 匹配任意字符串，'..' 单独使用时表示匹配任意长度任意类型，'+' 匹配其自身及子类，还有一个 '...'表示不定个数。也可以使用 &&、|| 操作符

下面主要介绍下上表中的**MethodPattern**。
MethodPattern对应的一个完整的表达式为：**@注解 访问权限 返回值的类型 包名.函数名(参数)**

- @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。
- 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
- 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：

```
1) java.*.Date：可以表示java.sql.Date，也可以表示java.util.Date
2) Test*：可以表示TestBase，也可以表示TestDervied
3) java..*：表示java任意子类
4) java..*Model+：表示Java任意package中名字以Model结尾的子类，比如TabelModel，TreeModel 等
```

- 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：

```
1) (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
2) (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.
3) ..代表任意参数个数和类型
4) (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
```

Pointcuts 示例

以下示例表示在aspectjx插件下，相同包是指同一个aar/jar包，AspectJ常规配置下不同包不能执行“execution”织入

execution

1. `execution(* com.howtodoinjava.EmployeeManager.*( .. ))`
   匹配EmployeeManger接口中所有的方法
2. `execution(* EmployeeManager.*( .. ))`
   当切面方法和EmployeeManager接口在相同的包下时，匹配EmployeeManger接口中所有的方法
3. `execution(public * EmployeeManager.*(..))`
   当切面方法和EmployeeManager接口在相同的包下时，匹配EmployeeManager接口的所有**public**方法
4. `execution(public EmployeeDTO EmployeeManager.*(..))`
   匹配EmployeeManager接口中权限为public并返回类型为EmployeeDTO的所有方法。
5. `execution(public EmployeeDTO EmployeeManager.*(EmployeeDTO, ..))`
   匹配EmployeeManager接口中权限为public并返回类型为EmployeeDTO，第一个参数为EmployeeDTO类型的所有方法。
6. `execution(public EmployeeDTO EmployeeManager.*(EmployeeDTO, Integer))`
   匹配EmployeeManager接口中权限为public、返回类型为EmployeeDTO，参数明确定义为EmployeeDTO,Integer的所有方法。

7. `"execution(@com.xyz.service.BehaviorTrace * *(..))"`

   匹配注解为"@com.xyz.service.BehaviorTrace"，返回值为任意类型，任意包名下的任意方法。

within

任意连接点：包括类/对象初始化块,field,方法,构造器

1. `within(com.xyz.service.*)`

   com.xyz.service包下任意连接点
2. `within(com.xyz.service..*)`

   com.xyz.service包或子包下任意连接点
3. `within(TestAspect)`

   TestAspect类下的任意连接点
4. `within(@com.xyz.service.BehavioClass *)`

   持有com.xyz.service.BehavioClass注解的任意连接点

withincode

假设方法functionA, functionB都调用了dummy，但只想在functionB调用dummy时织入代码。

```
public void functionA() { dummy() }
public void functionB() { dummy() }
public void dummy() {}  // 只在functionB调用的时候织入代码
```

```
@Aspect  // 加上@Aspect注解表示此类会被aspectj编译器编译，相关的Pointcut才会被织入
public class MethodTracer {
  // withincode: 在functionB方法内
  @Pointcut("withincode(void org.sdet.aspectj.MainActivity.functionB(..))")
  public void invokeFunctionB() {}

  // call: 调用dummy方法
  @Pointcut("call(void org.sdet.aspectj.MainActivity.dummy(..))")
  public void invokeDummy() {}

  // 在functionB内调用dummy方法
  @Pointcut("invokeDummy() && invokeFunctionB()")
  public void invokeDummyInsideFunctionB() {}

  // 在functionB方法内，调用dummy方法之前invoke下面代码（目前仅打印xxx）
  @Before("invokeDummyInsideFunctionB()")
  public void beforeInvokeDummyInsideFunctionB(JoinPoint joinPoint) {
    System.out.printf("Before.InvokeDummyInsideFunctionB.advice() called on '%s'", joi
  }
}
```

Advice

之前介绍的是如何找到切点，现在介绍的Advice就是告诉我们如何切，换个说法就是告诉我们要插入的代码以何种方式插入，比如说有以下几种：

名称	描述
Before	在方法执行之前执行要插入的代码
After	在方法执行之后执行要插入的代码
AfterReturning	在方法执行后，返回一个结果再执行，如果没结果，用此修辞符修辞是不会执行的
AfterThrowing	在方法执行过程中抛出异常后执行，也就是方法执行过程中，如果抛出异常后，才会执行此切面方法。
Around	在方法执行前后和抛出异常时执行（前面几种通知的综合）

Before、After示例

Before和After原理和用法一样，只是一个在方法前插入代码，一个在方法后面插入代码，在此只介绍Before。例如：在"com.luyao.aop.aspectj.AspectJActivity"执行onCreate里的代码之前打印"hello world"

```
package com.luyao.aop.aspectj;
public class AspectJActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

```
@Before("execution(* com.luyao.aop.aspectj.AspectJActivity.on*(android.os.Bundle))")
public void onActivityMethodBefore(JoinPoint joinPoint) throws Throwable {
    Log.e("luy", "hello world");
}}
```

查看反编译后的代码:

```
protected void onCreate(Bundle savedInstanceState) {
    AspectTest.aspectOf().onActivityMethodBefore(Factory.makeJP(ajc$tjp_0, (Object) this, (Object) this, (Object) savedInstanceState));
    super.onCreate(savedInstanceState);
}
```

Before示例

AfterReturning示例

在"com.luyao.aop.aspectj.AspectJActivity"执行getHeight()方法返回高度后打印这个高度值。

```
package com.luyao.aop.aspectj;
public class AspectJActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getHeight();
    }
    public int getHeight() {
            return 1280;
    }
}
```

```
@AfterReturning(pointcut = "call(* com.luyao.aop.aspectj.AspectJActivity.getHeight())"
public void getHeight(int height) {  // height必须和上面"height"一样
    Log.e("luy", "height:" + height);
}
```

反编译后的代码:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    AspectTest.aspectOf().getHeight(getHeight());
}
```

AfterReturning示例

AfterThrowing示例

如果我们经常需要收集抛出异常的方法信息，可以使用@AfterThrowing。比如我们要在任意类的任意方法抛出异常时，打印这个异常信息：

```
public class AspectJActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        divideZero();
```

```
    }
    public void divideZero() {
        int i = 2 / 0;
    }
}
```

```
@AfterThrowing(pointcut = "call(* *..*(..))", throwing = "throwable")  // "throwable"必
public void anyFuncThrows(Throwable throwable) {
    Log.e("luy", "throwable--->" + throwable);   // throwable--->java.lang.ArithmeticE
}
```

反编译后的代码：

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    try {
        divideZero();
    } catch (Throwable th) {
        AspectTest.aspectOf().anyFuncThrows(th);
    }
}
```

AfterThrowing示例

注意点：

1. @AfterThrowing 不支持 Field -> get & set，一般用在 Method 和 Constructor
2. 捕获的是抛出异常的方法，即使这个方法的调用方已经处理了此异常。上面例子中即使 divideZero()调用了try catch， 也能被anyFuncThrows织入。

Around 示例

例如我想在"com.luyao.aop.aspectj.AspectJActivity "执行setContentView方法前后打印当前系统时间：

```
package com.luyao.aop.aspectj;
public class AspectJActivity extends Activity {
    private static final String TAG = "luyao";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_aspect_j);
    }
}
```

```
@Around("call(* com.luyao.aop.aspectj.AspectJActivity.setContentView(..))")
public void invokeSetContentView(ProceedingJoinPoint proceedingJoinPoint) throws Throw
    Log.e("luy", "执行setContentView方法前:" + System.currentTimeMillis());
    proceedingJoinPoint.proceed();
    Log.e("luy", "执行setContentView方法后:" + System.currentTimeMillis());
}
```

反编译后的代码：

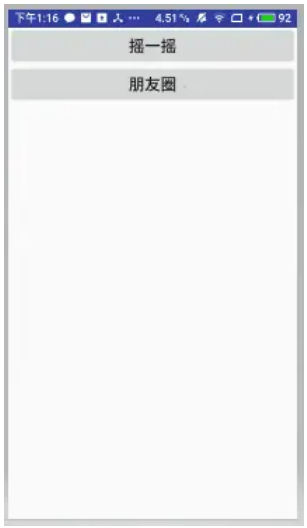
```
public class AjcClosure1 extends AroundClosure {
    public AjcClosure1(Object[] objArr) {
        super(objArr);
    }

    public Object run(Object[] objArr) {
        Object[] objArr2 = this.state;
        AspectJActivity.setContentView_aroundBody0((AspectJActivity) objArr2[0], (AspectJActivity) objArr2[1], Conversions.intValue(objArr2[2]), (JoinPoint) objArr2[3]);
        return null;
    }
}
```

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    JoinPoint makeJP = Factory.makeJP(ajc$tjp_0, (Object) this, (Object) this, Conversions.intObject(R.layout.activity_aspect_j));
    AspectTest.aspectOf().invokeSetContentView(new AjcClosure1(new Object[]{this, this, Conversions.intObject(R.layout.activity_aspect_j)}, makeJP)).linkClosureAndJoinPoint(4112));
}
```

Around 示例

2.3 简单使用案例

需求：如图，假设有2个功能分别是"朋友圈"和"摇一摇"，功能很简单，点击按钮触发睡眠和打印日志。统计这2个功能的耗时。



思路：一般思路是在调用这2个方法之前后分别获取当前系统的时间戳，然后相减得到耗时，关键代码如下：

```
// 摇一摇点击事件处理
public void shake(View view) {
    long begin = SystemClock.currentThreadTimeMillis();

    Log.i(TAG, "进入摇一摇方法体");
    SystemClock.sleep(3000);

    long end = SystemClock.currentThreadTimeMillis();
    Log.i(TAG, "耗时:" + (end - begin));
}

// 朋友圈点击事件处理
public void friend(View view) {
    long begin = System.currentThreadTimeMillis();

    Log.i(TAG, "进入朋友圈方法体");
    SystemClock.sleep(2000);

    long end = System.currentThreadTimeMillis();
    Log.i(TAG, "耗时:" + (end - begin));
}
```

上面这种处理方法对于功能点少还好处理，如果很多方法都需要统计，每个方法都这样写无疑加了很大的工作量，导致代码阅读逻辑不清晰，重构不方便，违背单一原则。如果使用AspectJ，可以通过一行注解，解决所有需要统计耗时的方法。具体代码如下：

编写布局文件：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.luyao.aop.aspectj.AspectJActivity"
    android:orientation="vertical">

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="shake"
        android:text="摇一摇"
        android:textSize="20sp"/>

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="friend"
```

```
        android:textSize="20sp"
        android:text="朋友圈"/>

</LinearLayout>
```

定义注解

```
@Target(ElementType.METHOD) // 修饰的是方法
@Retention(RetentionPolicy.CLASS) // 编译时注解
public @interface BehaviorTrace {
    String value(); // 功能点名称
    int type(); // 唯一确定功能点的值
}
```

通过定义@BehaviorTrace 来给"摇一摇"和"朋友圈"方法添加注解

编写 Aspect

```
@Aspect // 此处一定要定义，否则不会该类不会参与编译
public class BehaviorAspect {

    @Pointcut("execution(@com.luyao.aop.aspectj.BehaviorTrace  * *(..))") // 定义切点
    public void annoBehavior() {
    }

    @Around("annoBehavior()") // 定义怎么切，也可以这么写 @Around("execution(@com.luyao.ao
    public void dealPoint(ProceedingJoinPoint point) throws Throwable {
        //方法执行前
        MethodSignature methodSignature = (MethodSignature) point.getSignature();
        BehaviorTrace behaviorTrace = methodSignature.getMethod().getAnnotation(Behavi
        long begin = System.currentTimeMillis();
        Log.i("luy", "拿到需要切的方法啦，执行前");

        point.proceed(); // 执行被切的方法

        //方法执行完成
        long end = System.currentTimeMillis();
        Log.i("luy", behaviorTrace.value() + "(" + behaviorTrace.type() + ")" + " 耗时:
    }

}
```

使用@BehaviorAspect

```
public class AspectJActivity extends Activity {
    private static final String TAG = "luy";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_aspect_j);
    }

    @BehaviorTrace(value = "摇一摇", type = 1)
    public void shake(View view) {
        Log.i(TAG, "进入摇一摇方法体");
        SystemClock.sleep(3000);
    }

    @BehaviorTrace(value = "朋友圈", type = 2)
    public void friend(View view) {
        Log.i(TAG, "进入朋友圈方法体");
        SystemClock.sleep(2000);
    }
}
```

编写完毕，接下来测试，点击摇一摇打印日志：

```
拿到需要切的方法啦，执行前
进入摇一摇方法体
摇一摇(1) 耗时：3000ms
```

点击朋友圈打印日志：

```
拿到需要切的方法啦，执行前
进入朋友圈方法体
朋友圈(2) 耗时：2000ms
```

写在最后

本文介绍了AOP的思想、AOP的几种工具和AspectJ的基本用法。在实际开发项目中，当有需求时，了解AOP可以多一种思维方式去解决问题。同时，AspectJ织入代码会增加编译时间，使用时也需要考虑。