# ****一、AOP****

AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑  
各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

## ****1.1 主要功能****

日志记录，性能统计，安全控制，事务处理，异常处理等等。

## ****1.2 主要目标****

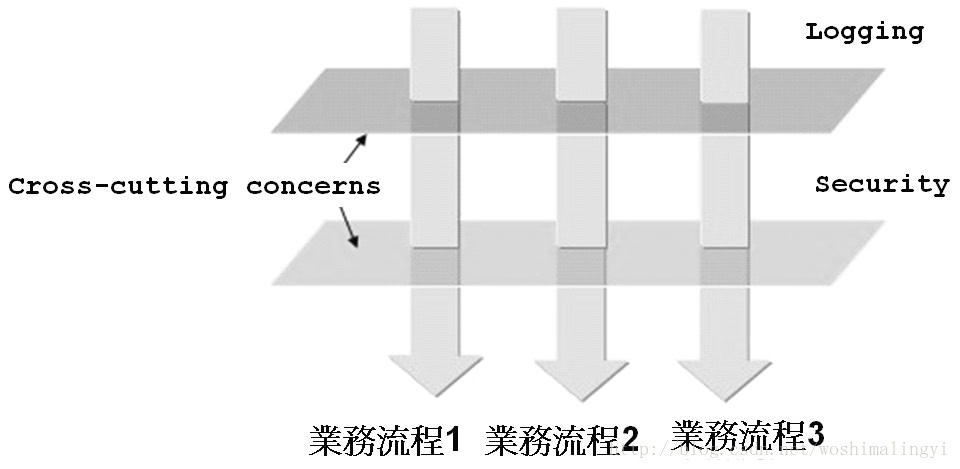
将日志记录，性能统计，安全控制，事务处理，异常处理等代码从业务逻辑代码中划分出来，通过对这些行为的分离，我们希望可以将它们独立到非指导业务逻辑的方法中，进而改变  
这些行为的时候不影响业务逻辑的代码。

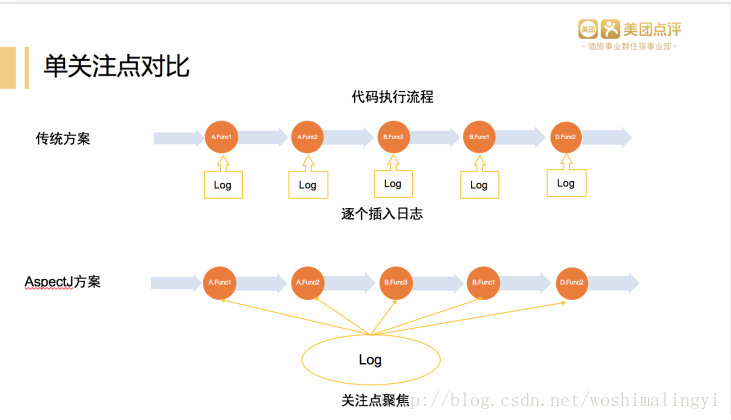
## ****1.3 适用对象****

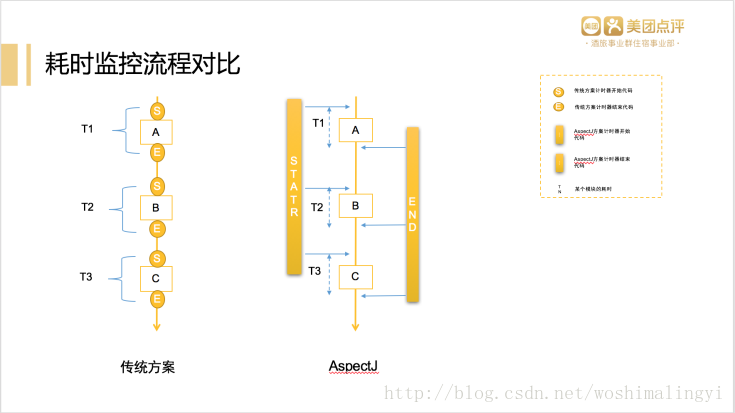
比较大型的项目，而且迭代较快，使用OOP太消耗内力。  
有日志、性能、安全、异常处理等横切关注点需求。

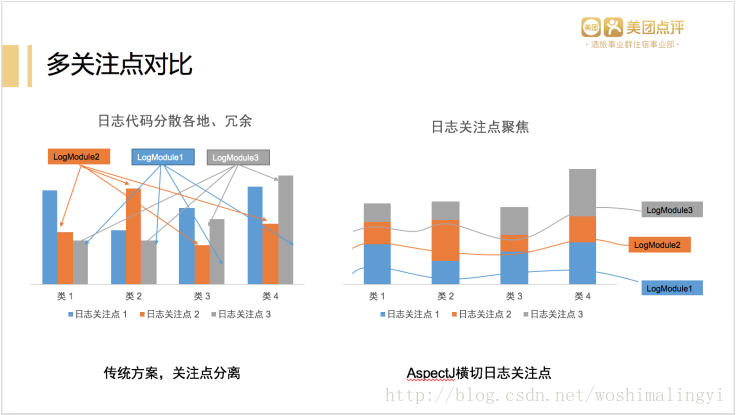
## ****1.4 AOP与OOP的关系****

OOP（面向对象编程）针对业务处理过程的实体及其属性和行为进行抽象封装，以获得更加清晰高效的逻辑单元划分。但是也有它的缺点，最明显的就是关注点聚焦时，面向对象无法简单的解决这个问题，一个关注点是面向所有而不是单一的类，不受类的边界的约束，因此OOP无法将关注点聚焦来解决，只能分散到各个类中。  
AOP（面向切面编程）则是针对业务处理过程中的切面进行提取，它所面对的是处理过程中的某个步骤或阶段，以获得逻辑过程中各部分之间低耦合性的隔离效果。这两种设计思想在目标上有着本质的差异。  
AOP并不是与OOP对立的，而是为了弥补OOP的不足。OOP解决了竖向的问题，AOP则解决横向的问题。因为有了AOP我们的调试和监控就变得简单清晰。它们之间的关系如下图所示：



****1.4.1 对比一——单一横切关注点****  




****1.4.2 对比二——多横切关注点****  


****结论：****  


# ****二、Android中使用@AspectJ****

AspectJ 意思就是Java的Aspect，Java的AOP。它其实不是一个新的语言，它的核心是ajc（编译器）\weaver（织入）。

* ajc编译器：基于Java编译器之上的，它是用来编译.aj文件，aspectj在Java编译器的基础上增加了一些它自己的关键字和方法。因此，ajc也可以编译Java代码。
* weaver织入器：为了在java编译器上使用AspectJ而不依赖于Ajc编译器，aspectJ 5出现了@AspectJ，使用注释的方式编写AspectJ代码，可以在任何Java编译器上使用。

由于AndroidStudio默认是没有ajc编译器的，所以在Android中使用@AspectJ来编写（包括SpringAOP也是如此）。它在代码的编译期间扫描目标程序，根据切点（PointCut）匹配,将开发者编写的Aspect程序编织（Weave）到目标程序的.class文件中，对目标程序作了重构（重构单位是JoinPoint），目的就是建立目标程序与Aspect程序的连接（获得执行的对象、方法、参数等上下文信息），从而达到AOP的目的。

## ****2.1 Gradle 配置示例****

要引入AspectJ到Android工程中，最重要的就是两个包：

*//在buildscript中添加该编织器，gradle构建时就会对class文件进行编织*

classpath 'org.aspectj:aspectjweaver:1.8.9'

*//在dependencies中添加该依赖，提供@AspectJ语法*

compile 'org.aspectj:aspectjrt:1.8.9'

此外还有一个工具包，用于Gradle构建时进行打日志等操作：

*//在buildscript中添加该工具包，在构建工程的时候执行一些任务：打日志等*

classpath 'org.aspectj:aspectjtools:1.8.9'import com.android.build.gradle.LibraryPlugin

import org.aspectj.bridge.IMessage

import org.aspectj.bridge.MessageHandler

import org.aspectj.tools.ajc.Main

*//打印gradle日志*

android.libraryVariants.all { variant ­>

LibraryPlugin plugin = project.plugins.getPlugin(LibraryPlugin)

JavaCompile javaCompile = variant.javaCompile

javaCompile.doLast {

String[] args = ["­showWeaveInfo",

"­1.5",

"­inpath", javaCompile.destinationDir.toString(),

"­aspectpath", javaCompile.classpath.asPath,

"­d", javaCompile.destinationDir.toString(),

"­classpath", javaCompile.classpath.asPath,

"­bootclasspath",

project.android.bootClasspath.join(

File.pathSeparator)]

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

break;

case IMessage.WARNING:

case IMessage.INFO:

log.info message.message, message.thrown

break;

case IMessage.DEBUG:

log.debug message.message, message.thrown

break;

}

}

}

}

****附：美团RoboAspectJ****

buildscript {

repositories {

mavenLocal()

}

dependencies {

classpath 'com.meituan.gradle:roboaspectj:0.9.2'

classpath 'jaop.gradle.plugin:gradle­plugin:1.0.2'

}

*// Exclude the version that the android plugin depends on.*

configurations.classpath.exclude group: 'com.android.tools.external.lombok'

}

****配置参数****

*// AspectJ*

aspectj {

disableWhenDebug true

javartNeeded true

*// 排除不需要AOP扫描的包*

exclude group: 'xxxx', module: 'xxxx'

compileOptions {

defaultJavaVersion = JavaVersion.VERSION\_1\_7

}

}

## ****2.2 基本概念****

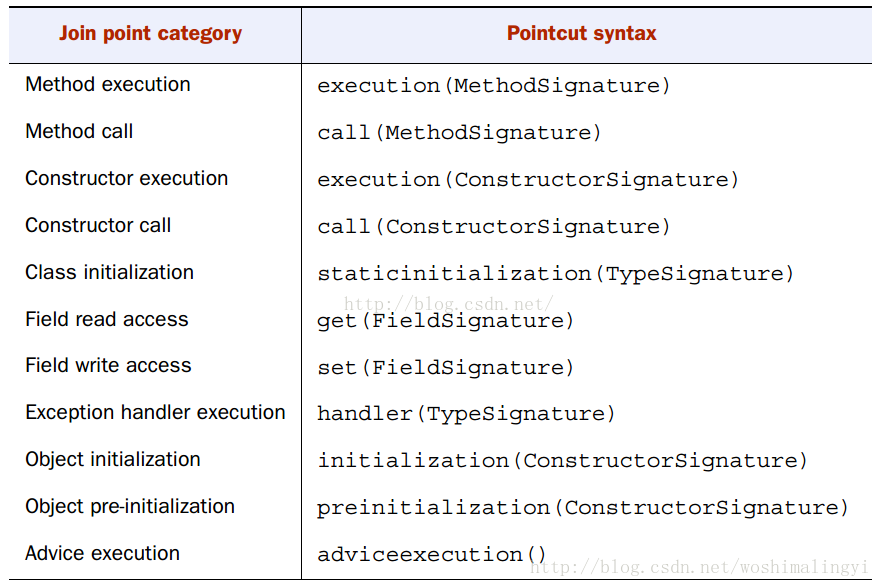
### ****2.2.1 切面——Aspect****

实现了cross­cutting功能，是针对切面的模块。最常见的是logging模块、方法执行耗时模块，这样，程序按功能被分为好几层，如果按传统的继承的话，商业模型继承日志模块的话需要插入修改的地方太多，而通过创建一个切面就可以使用AOP来实现相同的功能了，我们可以针对不同的需求做出不同的切面。

### ****2.2.2 连接点——JoinPoint****

连接点是切面插入应用程序的地方，该点能被方法调用，而且也会被抛出意外。连接点是应用程序提供给切面插入的地方，可以添加新的方法。比如：我们的切点可以认为是findInfo(String)方法。  
AspectJ将面向对象的程序执行流程看成是JoinPoint的执行链，每一个JoinPoint是一个单独的闭包，在执行的时候将上下文环境赋予闭包执行方法体逻辑。  
下面列表上的是被AspectJ认为是joinpoint的：  


### ****2.2.3 切点——PointCut****

切点的声明决定需要切割的JoinPoint的集合，就结果上来说，它是JoinPoint的一个实际子集合。  
pointcut可以控制你把哪些advice应用于jointpoint上去，通常通过正则表达式来进行匹配应用，决定了那个jointpoint会获得通知。分为call、execution、target、this、within等关键字，含义下面会附图。  
****1.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：  


****2.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：  


****3.匹配规则  
（1）类型匹配语法****  
首先让我们来了解下AspectJ类型匹配的通配符：

\*：匹配任何数量字符；  
..：匹配任何数量字符的重复，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。  
+：匹配指定类型的子类型；仅能作为后缀放在类型模式后边。  
AspectJ使用 且（&&）、或（||）、非（！）来组合切入点表达式。

****（2）匹配模式****  
call(<注解？> <修饰符?> <返回值类型> <类型声明?>.<方法名>(参数列表) <异常列表>？)

* 精确匹配

*//表示匹配 com.davidkuper.MainActivity类中所有被@Describe注解的public void方法。*

@Pointcut("call(@Describe public void com.davidkuper.MainActivity.init(Context))")

public void pointCut(){}

* 单一模糊匹配

*//表示匹配 com.davidkuper.MainActivity类中所有被@Describe注解的public void方法。*

@Pointcut("call(@Describe public void com.davidkuper.MainActivity.\*(..)) ")

public void pointCut(){}

*//表示匹配调用Toast及其子类调用的show方法，不论返回类型以及参数列表，并且该子类在以com.meituan或者com.sankuai开头的包名内*

@Pointcut("call(\* android.widget.Toast+.show(..)) && (within(com.meituan..\*)|| within(com.sankuai..\*))")

public void toastShow() {

}

* 组合模糊匹配

*//表示匹配任意Activity或者其子类的onStart方法执行，不论返回类型以及参数列表，且该类在com.meituan.hotel.roadmap包名内*

@Pointcut("execution(\* \*..Activity+.onStart(..))&& within(com.meituan.hotel.roadmap.\*)")

public void onStart(){}

****（3）获取参数****

* 通过声明参数语法arg()显示获取参数

@Around(value = "execution(\* BitmapFacade.picasso.init(java.lang.String,java.lang.String)) && args(arg1,arg2)"

public Object aroundArgs(String arg1,String arg2,ProceedingJoinPoint joinPoint){

System.out.println("aspects arg = " + arg1.toString()+" " + arg2);

Object resutObject = null;

try {

resutObject = joinPoint.proceed(new Object[]{arg1,arg2});

} catch (Throwable e) {

e.printStackTrace();

}

return resutObject;

}

* 通过joinPoint.getArg()获取参数列表

@Around("execution(static \* tBitmapFacade.picasso.init(..)) && !within(aspectj.\*) ")

public void pointCutAround(ProceedingJoinPoint joinPoint){

Object resutObject = null;

try {

*//获取参数列表*

Object[] args = joinPoint.getArgs();

resutObject = joinPoint.proceed(args);

} catch (Throwable e) {

e.printStackTrace();

}

return resutObject;

};

* 12

****（4）异常匹配****

*/\*\**

*\* 截获Exception及其子类报出的异常。*

*\* @param e 异常参数*

*\*/*

@Pointcut("handler(java.lang.Exception+)&&args(e)")

public void handle(Exception e) {}

### ****2.2.4 通知——Advise****

advice是我们切面功能的实现，它是切点的真正执行的地方。比如像写日志到一个文件中，会在pointcut匹配到的连接点中插入advice（包括：before、after、around等）代码到应用程序中。  
****（1）@Before、@After****

*//所有实例方法调用截获*

private static final String INSTANCE\_METHOD\_CALL =

"call(!static \* com.meituan.hotel.roadmap..\*.\*(..))&&target(Object)";

@Pointcut(INSTANCE\_METHOD\_CALL) public void instanceMethodCall() {

}

*//实例方法调用前后Advice*

@Before("instanceMethodCall()") public void beforInstanceCall(JoinPoint joinPoint) {

printLog(joinPoint, "before instance call");

}

@After("instanceMethodCall()") public void afterInstanceCall(JoinPoint joinPoint) {

printLog(joinPoint, "after instance call");

}

****（2）@Around****

*//横切项目中所有Activity的子类，以Layout命名、以及它的子类的所有方法的执行*

private static final String POINTCUT\_METHOD =

"(execution(\* android.app.Activity+.\*(..)) ||execution(\* \*..Layout+.\*(..)))&& within(com.meituan.hotel.roadmap.\*)";

@Pointcut(POINTCUT\_METHOD) public void methodAnnotated() {

}

@Around("methodAnnotated()") public Object weaveJoinPoint(ProceedingJoinPoint joinPoint)throws Throwable{

*//调用原方法的执行。*

Object result = joinPoint.proceed();

return result;

}

****（3）@AfterThrowing****

*/\*\**

*\* 在异常抛出后，该操作优先于下一个切点的@Before()*

*\* @param joinPoint*

*\* @param e 异常参数*

*\*/*

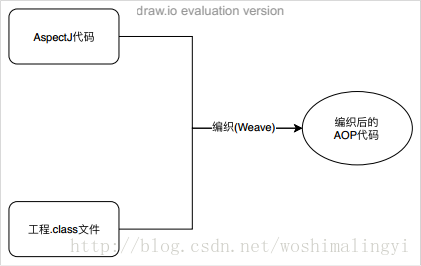
@AfterThrowing(pointcut = "afterThrow()",throwing = "e")

public void afterThrowing(JoinPoint joinPoint,Exception e){

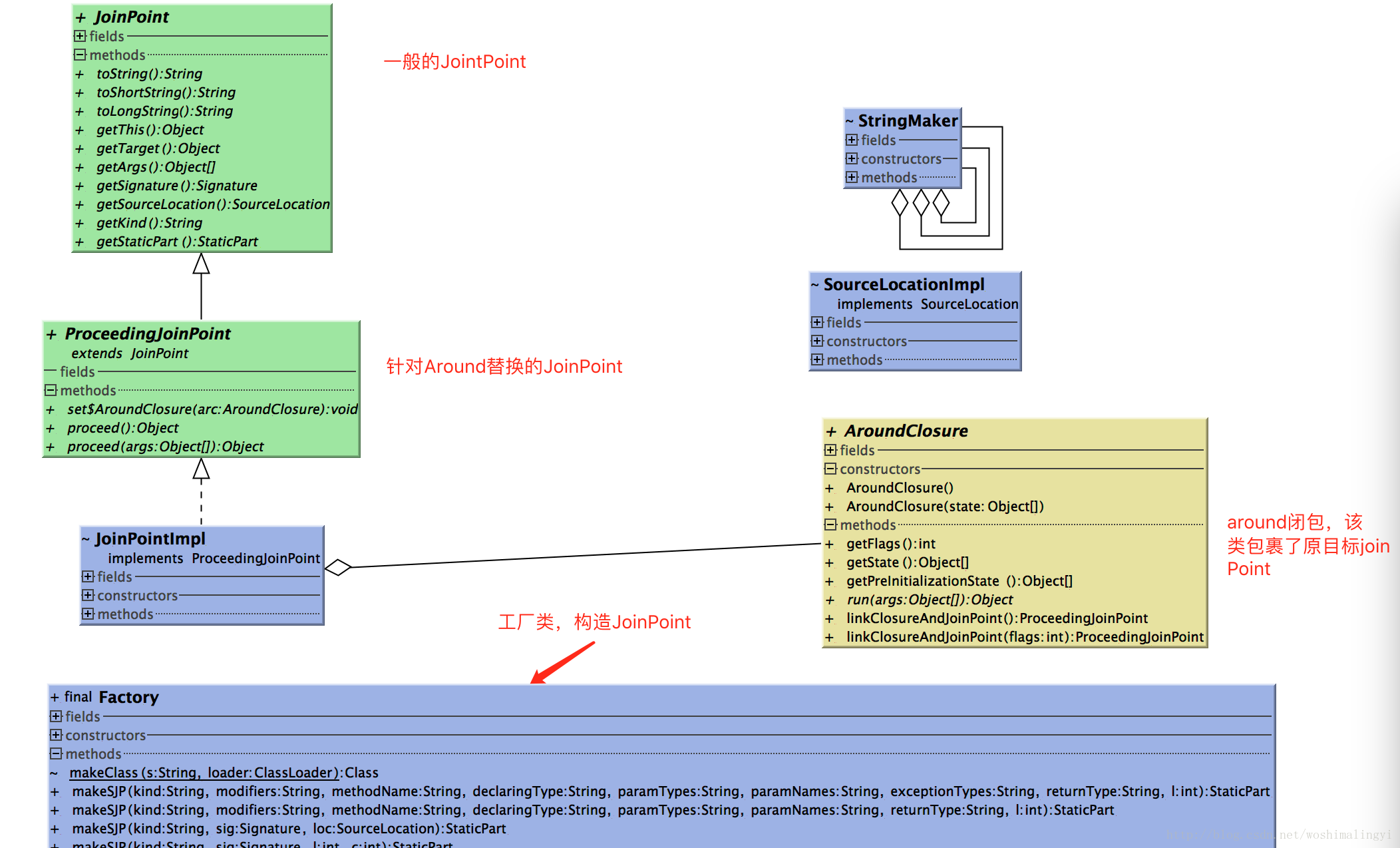
Log.e(TAG,joinPoint.getTarget().getClass().getSimpleName() + " afterThrowing() :" + e.toString());

}

## ****2.3 执行原理****

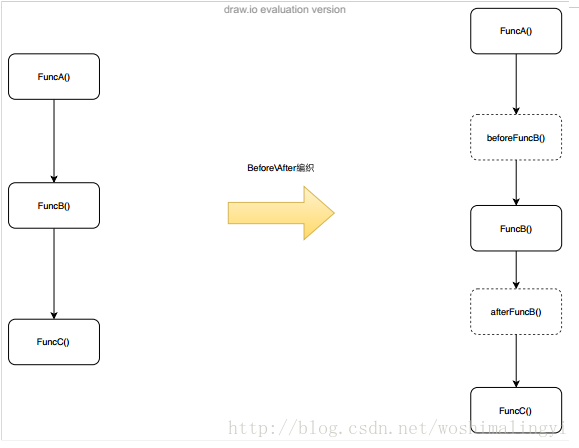
AspectJ是通过对目标工程的.class文件进行代码注入的方式将通知(Advise)插入到目标代码中。  
第一步：根据pointCut切点规则匹配的joinPoint；  
第二步：将Advise插入到目标JoinPoint中。  
这样在程序运行时被重构的连接点将会回调Advise方法，就实现了AspectJ代码与目标代码之间的连接。  


JoinPoint类包UML：

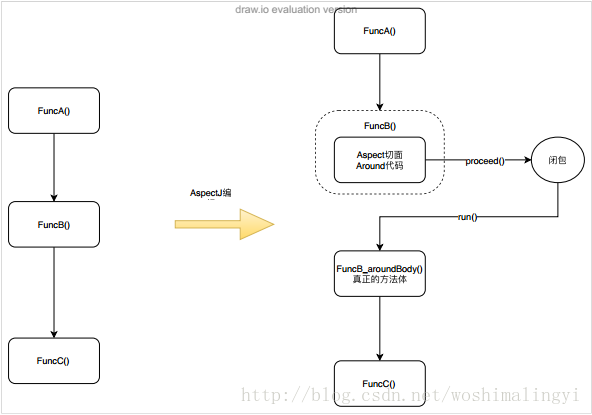


### ****2.3.1 Before、After(AfterThrowing)插入示意图****

Before和After的插入其实就是在匹配到的JoinPoint调用的前后插入我们编写的Before\After的Advise方法，以此来达到在目标JoinPoint执行之前先进入Advise方法，执行之后进入Advise方法。  
如下图所示，目标JoinPoint为FuncB()方法，需要在他执行前后进行AOP截获：



### ****2.3.2 Around替换逻辑示意图****

总体来说，使用了代理+闭包的方式进行替换，将原方法体放置到新的函数中替换，通过一个单独的闭包拆分来执行，相当于对目标JoinPoint进行了一个代理。  


### ****2.3.3 代码分析****

下面的Example作为目标源码，我们对它的printLog()方法进行替换、对getValue()方法调用前后插入Advise方法。

public class Example {

String value = "value";

public void printLog() {

String str = getValue();

}

public String getValue() {

return value;

}

}

****切面代码:****

@Aspect

public class LogAspect{

*//所有实例方法调用截获*

private static final String INSTANCE\_METHOD\_CALL = "call(!static \* com.meituan.hotel.roadmap..\*.\*(..))&&target(Object)"

@Pointcut(INSTANCE\_METHOD\_CALL) public void instanceMethodCall() {

}

*//实例方法调用前后Advice*

@Before("instanceMethodCall()") public void beforInstanceCall(JoinPoint joinPoint) {

printLog(joinPoint, "before instance call");

}

@After("instanceMethodCall()") public void afterInstanceCall(JoinPoint joinPoint) {

printLog(joinPoint, "after instance call");

}

*//所有实例方法执行截获*

private static final String INSTANCE\_METHOD\_EXECUTING = "execution(!static \* com.meituan.hotel.roadmap..\*.\*(..))&&target(Object)"

@Pointcut(INSTANCE\_METHOD\_EXECUTING) public void instanceMethodExecuting() {

}

*//实例方法执行Advice*

@Around("instanceMethodExecuting()") public Object InstanceMethodExecutingAround(ProceedingJoinPoint joinPoint){

Log.e(getClass().getSimpleName(),

"InstanceMethodExecuting()");

Object result = printLog(joinPoint, "instance

executing");

return result;

}

}

****反编译后的结果****  
网上给的反编译过程都是apktool——>dex2jar——>jd­gui，这个我用hotel\_road\_map的debug包试过，反编译出来的jar包里面只有几个系统类，不知道什么原因,其他的包又可以正常反编译。  
推荐一个反编译工具：jadx（可以直接反编译apk）。

public class Example {

private static final StaticPart ajc$tjp\_0 = null;

private static final StaticPart ajc$tjp\_1 = null;

private static final StaticPart ajc$tjp\_2 = null;

String TAG = "Example";

String value = "value";

static {

ajc$preClinit();

}

*//初始化连接点静态部分：方法名、参数列表、返回值、包路径等等。*

private static void ajc$preClinit() {

Factory factory = new Factory("Example.java", Example.class);

ajc$tjp\_0 = factory.makeSJP(JoinPoint.METHOD\_CALL, factory.makeMethodSig("1", "getValue", "com.meituan.hotel.roadmap.Ex");

ajc$tjp\_1 = factory.makeSJP(JoinPoint.METHOD\_EXECUTION, factory.makeMethodSig("1", "printLog", "com.meituan.hotel.roadm");

ajc$tjp\_2 = factory.makeSJP(JoinPoint.METHOD\_EXECUTION, factory.makeMethodSig("1", "getValue", "com.meituan.hotel.roadm");

}

*//原方法的闭包，在Aspect切面中joinPoint.proceed()会调用该闭包*

public class AjcClosure1 extends AroundClosure {

public AjcClosure1(Object[] objArr) {

super(objArr);

}

public Object run(Object[] objArr) {

Object[] objArr2 = this.state;

Example.printLog\_aroundBody0((Example) objArr2[0], (JoinPoint) objArr2[1]);

return null;

}

}

*//原方法真正的方法体,在闭包中被调用*

static final void printLog\_aroundBody0(Example ajc$this, JoinPoint joinPoint) {

JoinPoint makeJP = Factory.makeJP(ajc$tjp\_0, ajc$this, ajc$this);

try {

*//@Before的Advise被插入了目标代码调用之前*

LogAspect.aspectOf().beforInstanceCall(makeJP);

String str = ajc$this.getValue();

} finally {

*//@After的Advise被插入到目标代码调用之后,通过Finally强制执行After逻辑，每一个Before*

LogAspect.aspectOf().afterInstanceCall(makeJP);

}

}

*//原来的printLog()方法被AspectJ给替换了，替换成为链接AspectJ和源代码的桥梁，真正的方法体被放在了新的方法中。*

public void printLog() {

*//连接点构造*

JoinPoint makeJP = Factory.makeJP(ajc$tjp\_1, this, this);

*//将连接点与原方法的闭包连接，这样就可以在AspectJ的JoinPoint中通过joinPoint.proceed()调用闭包执行原方法。*

LogAspect.aspectOf().InstanceMethodExecutingAround(new AjcClosure1(new Object[]{this, makeJP}).linkClosureAndJoinPoint(

}

public String getValue() {

JoinPoint makeJP = Factory.makeJP(ajc$tjp\_2, this, this);

return (String)LogAspect

.InstanceMethodExecutingAround(new AjcClosure3(new Object[]{this, makeJP})

.linkClosureAndJoinPoint();

}

}

****Before、After(AfterThrowing)插入分析****

Before\After的插入调用比较简单，通过PointCut定位匹配到JoinPoint之后，将我们编写的Before\After的切面方法直接插入到目标JoinPoint前后即可。这样就可以改变原有的代码调用轨  
迹，在目标方法调用前后增加我们自己的AOP方法。

*//原方法真正的方法体,在闭包中被调用*

static final void printLog\_aroundBody0(Example ajc$this, JoinPoint joinPoint) {

JoinPoint makeJP = Factory.makeJP(ajc$tjp\_0, ajc$this, ajc$this);

try {

*//@Before的Advise被插入了目标代码调用之前*

LogAspect.aspectOf().beforInstanceCall(makeJP);

String str = ajc$this.getValue();

} finally {

*//@After的Advise被插入到目标代码调用之后,通过Finally强制执行After逻辑，每一个Before*

LogAspect.aspectOf().afterInstanceCall(makeJP);

}

}

****Around替换代码分析****  
JoinPoint为printLog()方法，是被Around替换的，反编译后的部分代码如下：  
首先，在静态初始化的时候，通过Factory会为每一个JPoint先构造出静态部分信息StaticPart。

*//初始化连接点静态部分：方法名、参数列表、返回值、包路径等等。*

private static void ajc$preClinit() {

Factory factory = new Factory("Example.java", Example.class);

ajc$tjp\_0 = factory.makeSJP(JoinPoint.METHOD\_CALL, factory.makeMethodSig("1", "getValue", "com.meituan.hotel.roadmap.Exa");

ajc$tjp\_1 = factory.makeSJP(JoinPoint.METHOD\_EXECUTION, factory.makeMethodSig("1", "printLog", "com.meituan.hotel.roadma");

ajc$tjp\_2 = factory.makeSJP(JoinPoint.METHOD\_EXECUTION, factory.makeMethodSig("1", "getValue", "com.meituan.hotel.roadma");

}

public JoinPoint.StaticPart makeSJP(String kind, String modifiers, String methodName, String declaringType, String paramTypes,St

*//构造方法签名实例，其中存储着方法的静态信息*

Signature sig = this.makeMethodSig(modifiers, methodName, declaringType, paramTypes, paramNames, "", returnType);

return new JoinPointImpl.StaticPartImpl(count++, kind, sig, makeSourceLoc(l, ­1));

}

其次，将printLog方法体替换，以XXX\_aroundBodyN(args)命名，原方法体被替换如下：

*//原来的printLog()方法被AspectJ给替换了，替换成为链接AspectJ和源代码的桥梁，真正的方法体被放在了新的方法中。*

public void printLog() {

*//连接点构造*

JoinPoint makeJP = Factory.makeJP(ajc$tjp\_1, this, this);

*//将连接点与原方法的闭包连接，这样就可以在AspectJ的JoinPoint中通过joinPoint.proceed()调用闭包执行原方法。*

LogAspect.aspectOf().InstanceMethodExecutingAround(new AjcClosure1(new Object[]{this, makeJP}).linkClosureAndJoinPoint(

}

AroundClosure闭包，将运行时对象和当前连接点JP对象传入其中，调用linkClosureAndJoinPoint()进行两端的绑定，这样在Around中就可以通过ProceedingJoinPoint.proceed()调用AroundClosure，进而调用目标方法。

public abstract class AroundClosure {

protected Object[] state;

protected Object[] preInitializationState;

public AroundClosure() {

}

public AroundClosure(Object[] state) {

this.state = state;

}

public ProceedingJoinPoint linkClosureAndJoinPoint() {

*//获取执行链接点，默认数组最后一个是连接点*

ProceedingJoinPoint jp =

(ProceedingJoinPoint)state[state.length­1];

*//设置执行时闭包*

jp.set$AroundClosure(this);

return jp;

}

}

JoinPointImpl，包括一个JoinPoint的静态部分和实例部分:

class JoinPointImpl implements ProceedingJoinPoint {

*//JP静态部分*

static class StaticPartImpl implements

JoinPoint.StaticPart {

String kind;

Signature signature;

SourceLocation sourceLocation;

private int id;

*//省略*

}

Object \_this;

Object target;

Object[] args;

org.aspectj.lang.JoinPoint.StaticPart staticPart;

*//省略....*

*// To proceed we need a closure to proceed on*

private AroundClosure arc;

public void set$AroundClosure(AroundClosure arc) {

this.arc = arc;

}

*//通过proceed()调用闭包arc的run方法，并且传入JP的执行状态：参数列表等。进而调用原方法体执行*

public Object proceed() throws Throwable {

*//when called from a before advice, but be a no­op*

if (arc == null)

return null;

else

return arc.run(arc.getState());

}

*//省略....*

}

## ****2.4 AspectJ切面编写****