Android Annotation and AspectJ

by zhengxiaobin

- Annotation(注解)是 JDK5.0 就引入的一个功能。

- 注解是 Java 的一个新的类型(与接口很相似)，它与类、接口、枚举是在同一个层次，它们都称作为 Java 的一个类型(TYPE)。它可以声明在包、类、字段、方法、局部变量、方法参数等的前面，用来对这些元素进行说明、注释。它的作用非常多，例如：进行编译检查、生成说明文档、代码分析等

最常见的几个小伙伴：

Override
Deprecated
SuppressWarnings

- @Deprecated：该注解的作用是标记某个过时的类或方法

  用法：
  ```
  @Deprecated public void fun
  {
      ……
  }
  ```

- @Override：该注解用在方法前面，用来标识该方法是重写父类的某个方法

  用法：
  ```
  @Override
  public void fun()
  {
      ……
  }
  ```

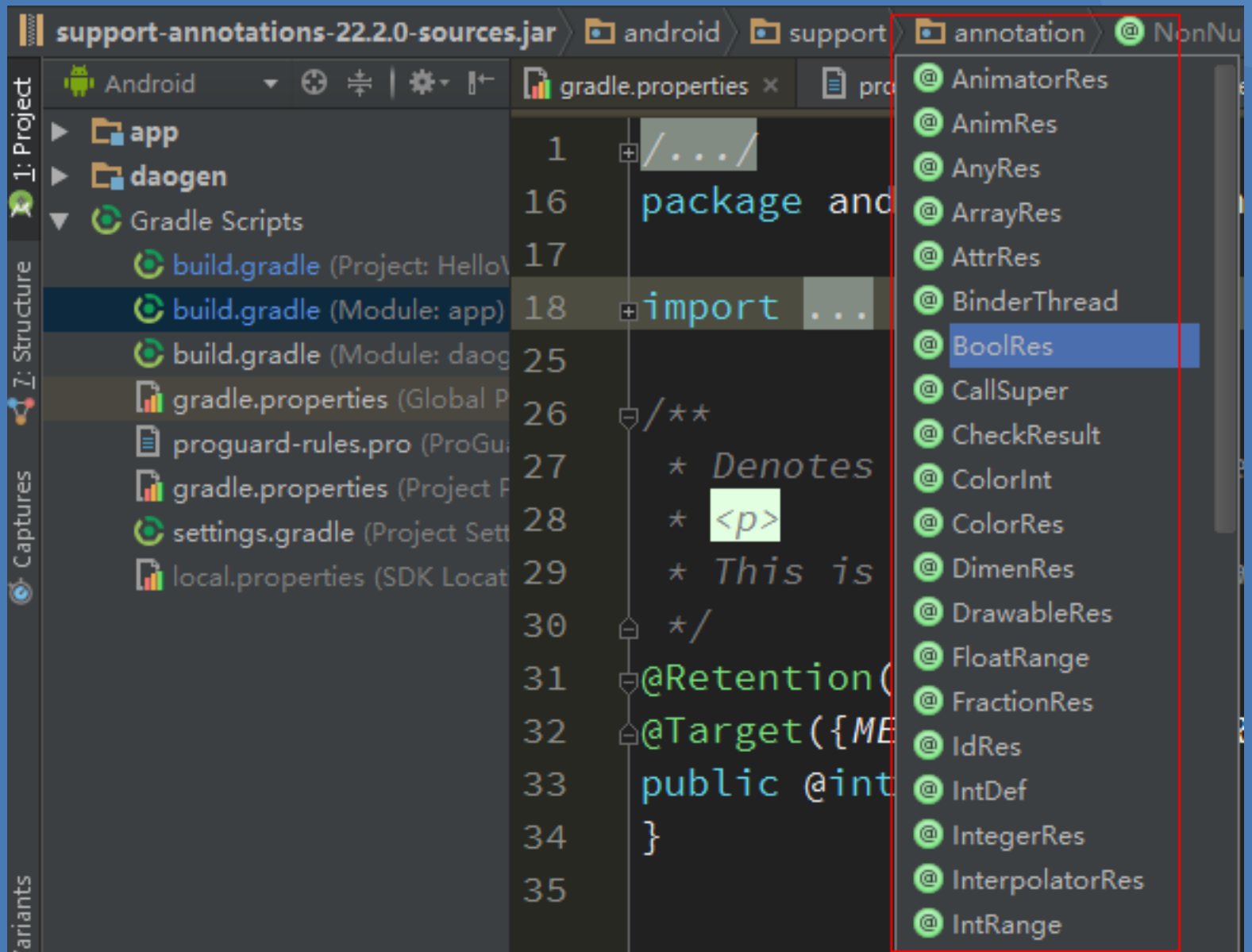
p @SuppressWarnings：可以注释一段代码，该注解的作用是阻止编译器发出某些警告信息。它可以有以下参数：

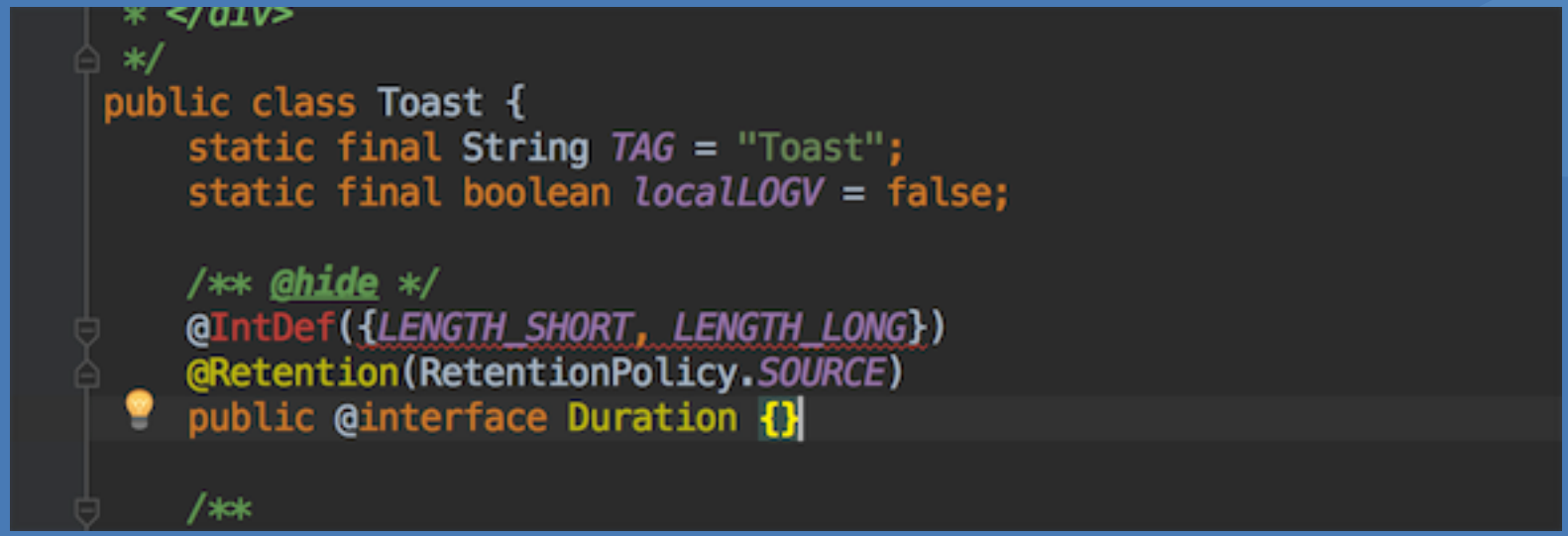
- ➢deprecation：过时的类或方法警告
- ➢unchecked：执行了未检查的转换时警告
- ➢fallthrough：当 switch 程序块直接通往下一种情况而没有 break 时的警告
- ➢path：在类路径、源文件路径等中有不存在的路径时的警告
- ➢serial：当在可序列化的类上缺少 serialVersionUID 定义时的警告
- ➢finally：任何 finally 子句不能完成时的警告
- ➢all：关于以上所有情况的警告

用法： @SuppressWarnings(value={"unchecked"})

……代码

安卓注解有8种类型，分别是Nullness注解、资源类型注解、线程注解、变量限制注解、权限注解、结果检查注解、CallSuper注解、枚举注解(IntDef和StringDef)。





常用的几个：

@NoNull

@Nullable

@BoolRes, @IdRes, @IntegerRes, @StringRes, @ColorRes

@RequiresPermission(Manifest.permission.SET_WALLPAPER)

@IntDef and @StringDef

p 元注解：专门负责注解其它的注解

p 元注解的种类
- ➢@Target
- ➢@Retention
- ➢@Documented
- ➢@Inheried

@Target：它是被定义在一个注解类的前面，用来说明该注解可以被声明在哪些元素前。它有以下参数：

- ElementType.TYPE：说明该注解只能被声明在一个类、接口(包括注解类型)、 enum前
- ElementType.FIELD：说明该注解只能被声明在一个类的字段(域)前
- ElementType.METHOD：说明该注解只能被声明在一个类的方法前
- ElementType.PARAMETER：说明该注解只能被声明在一个方法参数前
- ElementType.CONSTRUCTOR：说明该注解只能声明在一个类的构造方法前
- ElementType.LOCAL_VARIABLE：说明该注解只能声明在一个局部变量前
- ElementType.ANNOTATION_TYPE：说明该注解只能声明在一个注解类型前
- ElementType.PACKAGE：说明该注解只能声明在一个包名前

用法：

```
@Target(ElementType.METHOD)
@Target(value=ElementType.METHOD)
@Target(ElementType.METHOD,ElementType.CONSTRUCTOR)
```

p @Retention：它是被定义在一个注解类的前面，用来说明该注解的生命周期。它有以下参数：

- ➢RetentionPolicy.SOURCE：指定注解只保留在一个源文件当中
- ➢RetentionPolicy.CLASS：指定注解只保留在一个 class 文件中
- ➢RetentionPolicy.RUNTIME：指定注解可以保留在程序运行期间

- 注解有三个生命周期，它默认的生命周期是保留在一个 class 文件中，但它也可以由一个 @Retention 的元注解指定它的生命周期。
  - ➢Java源文件：
    当在一个注解类前定义了一个 @Retention(RetentionPolicy.SOURCE) 的注解，那么说明该注解只保留在一个源文件当中，当编译器将源文件编译成 class 文件时，它不会将源文件中定义的注解保留在 class 文件中
  - ➢class 文件中：
    当在一个注解类前定义了一个 @Retention(RetentionPolicy.CLASS)的注解，那么说明该注解只保留在一个 class 文件当中，当加载 class 文件到内存时，虚拟机会将注解去掉，从而在程序中不能访问
  - ➢程序运行期间：
    当在一个注解类前定义一个 @Retention(RetentionPolicy.RUNTIME)的注解，那么说明该注解在程序运行期间都会存在内存当中。此时，我们可以通过反射来获得定义在某个类上的所有注解

- 注解的使用分为三个过程：定义注解-->声明注解-->得到注解

- 注解大多是用做对某个类、方法、字段进行说明，标识的。以便在程序运行期间我们通过反射获得该字段或方法的注解的实例，来决定该做些什么处理或不该进行什么处理

p 一个简单的注解：

```
public @interface Annotation01
{
    //定义公共的 final 静态属性
    ……
    //定义公共的抽象方法
    ……
}
```

p 注解的成员：注解和接口相似，它只能定义 final 静态属性和公共抽象方法

p 注解的方法：

➢方法前默认会加上 public abstract

➢在声明方法时可以定义方法的默认返回值，如：

String color() default “blue”;

String color() default {“blue”, “red”, ……}

➢方法的返回值的类型：可以有8种基本类型(byte, short, int, long, float, double, boolean, char,)、String、Class、枚举、注解及这些类型的数组

```java
/**
 * 水果名称注解
 * @author peida
 *
 */
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface FruitName {
    String value() default "";
}
```

```java
/**
 * 水果供应者注解
 * @author peida
 *
 */
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface FruitProvider {
    /**
     * 供应商编号
     * @return
     */
    public int id() default -1;

    /**
     * 供应商名称
     * @return
     */
    public String name() default "";

    /**
     * 供应商地址
     * @return
     */
    public String address() default "";
}
```

注解处理器类库(java.lang.reflect.AnnotatedElement)：

java.lang.reflect 包下主要包含一些实现反射功能的工具类，实际上，java.lang.reflect 包所有提供的反射API扩充了读取运行时Annotation信息的能力。

AnnotatedElement 接口是所有程序元素（Class、Method和Constructor）的父接口，所以程序通过反射获取了某个类的AnnotatedElement对象之后，程序就可以调用该对象的如下四个个方法来访问Annotation信息：

方法1：<T extends Annotation> T getAnnotation(Class<T> annotationClass): 返回改程序元素上存在的、指定类型的注解，如果该类型注解不存在，则返回null。

方法2：Annotation[] getAnnotations():返回该程序元素上存在的所有注解。

方法3：boolean is AnnotationPresent(Class<?extends Annotation> annotationClass):判断该程序元素上是否包含指定类型的注解，存在则返回true，否则返回false.

方法4：Annotation[] getDeclaredAnnotations()：返回直接存在于此元素上的所有注释。与此接口中的其他方法不同，该方法将忽略继承的注释。（如果没有注释直接存在于此元素上，则返回长度为零的一个数组。）该方法的调用者可以随意修改返回的数组；这不会对其他调用者返回的数组产生任何影响。

Show Me The Code:

http://git.meiyou.im/Android/Android/wikis/annotation

运行时注释声明：

```
@Target(ElementType.TYPE)//ElementType.TYPE只能在类中使用此注解
@Retention(RetentionPolicy.RUNTIME)// @Retention(RetentionPolicy.RUNTIME) 注解可以在运行
时通过反射获取一些信息
@Documented
public @interface FindViewByIdLayout {
    int value();
}
```

```java
public class ViewUtils {
    /**
     * 保存传入的activity
     */
    private static Class<?> activityClass;

    /**
     * 初始化activity和所有注解
     *
     * @param obj 你需要初始化的activity
     */
    public static void inject(Object obj) {
        activityClass = obj.getClass();
        injectContent(obj);
    }

    // 初始化activity布局文件
    private static void injectContent(Object obj) {
        FindViewByIdLayout annotation = activityClass
                .getAnnotation(FindViewByIdLayout.class);
        if (annotation != null) {
            // 获取注解中的对应的布局id 因为注解只有个方法 所以@XXX(YYY)时会自动赋值给注解
            int id = annotation.value();
            try {
                // 得到activity中的方法 第一个参数为方法名 第二个为可变参数 类型为 参数类
                Method method = activityClass.getMethod("setContentView",
                        int.class);
                // 调用方法 第一个参数为哪个实例去调用 第二个参数为 参数
                method.invoke(obj, id);
            } catch (Exception e) {

                e.printStackTrace();
            }
        }
```

```java
@FindViewByIdLayout(R.layout.activity_main)
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ViewUtils.inject(this);
    }

}
```

注解的分类：

从取值的方式上来说，注解可以分成两类——编译时注解和运行时注解

运行时注解

运行时注解表示你只能在程序运行时去操作它，那怎么样才能在运行时去操作呢?
很简单，使用反射。

典型：Retrofit ， ORMLite

http://git.meiyou.im/Android/jet

编译时注解

因为运行时注解毕竟是在程序运行时去进行操作的，用到了反射，在效率方面会有一定损耗。因此追求性能的更多使用了APT；

典型： dagger ， butterKnife

APT的全称是Annotation Processing Tool，从Java5开始，JDK就自带了注解处理器APT。主要用于在编译期根据不同的注解类生成或者修改代码。APT运行于独立的JVM进程中（编译之前），并且在一次编译过程中可以会被多次调用。

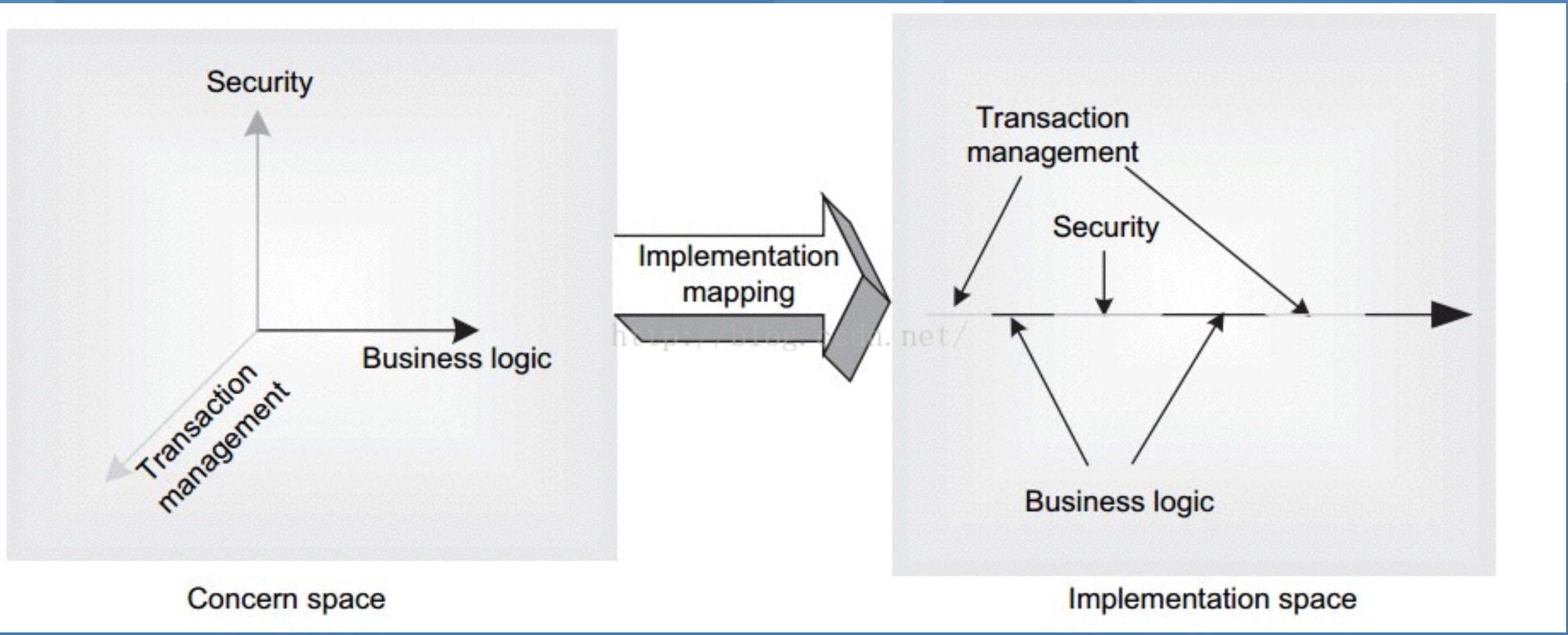
核心就是： 编译时读取注解＋APT＋动态生成源代码

Show Me The Code:

http://git.meiyou.im/Android/Android/wikis/apt%E4%BD%BF%E7%94%A8demo

什么是Aop编程

AOP为Aspect Oriented Programming的缩写，意为：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。


Security
Business logic
Transaction management
Implementation mapping
Transaction management
Security
Business logic
Concern space
Implementation space

常见的使用场景

性能监控: 在方法调用前后记录调用时间，方法执行太长或超时报警。

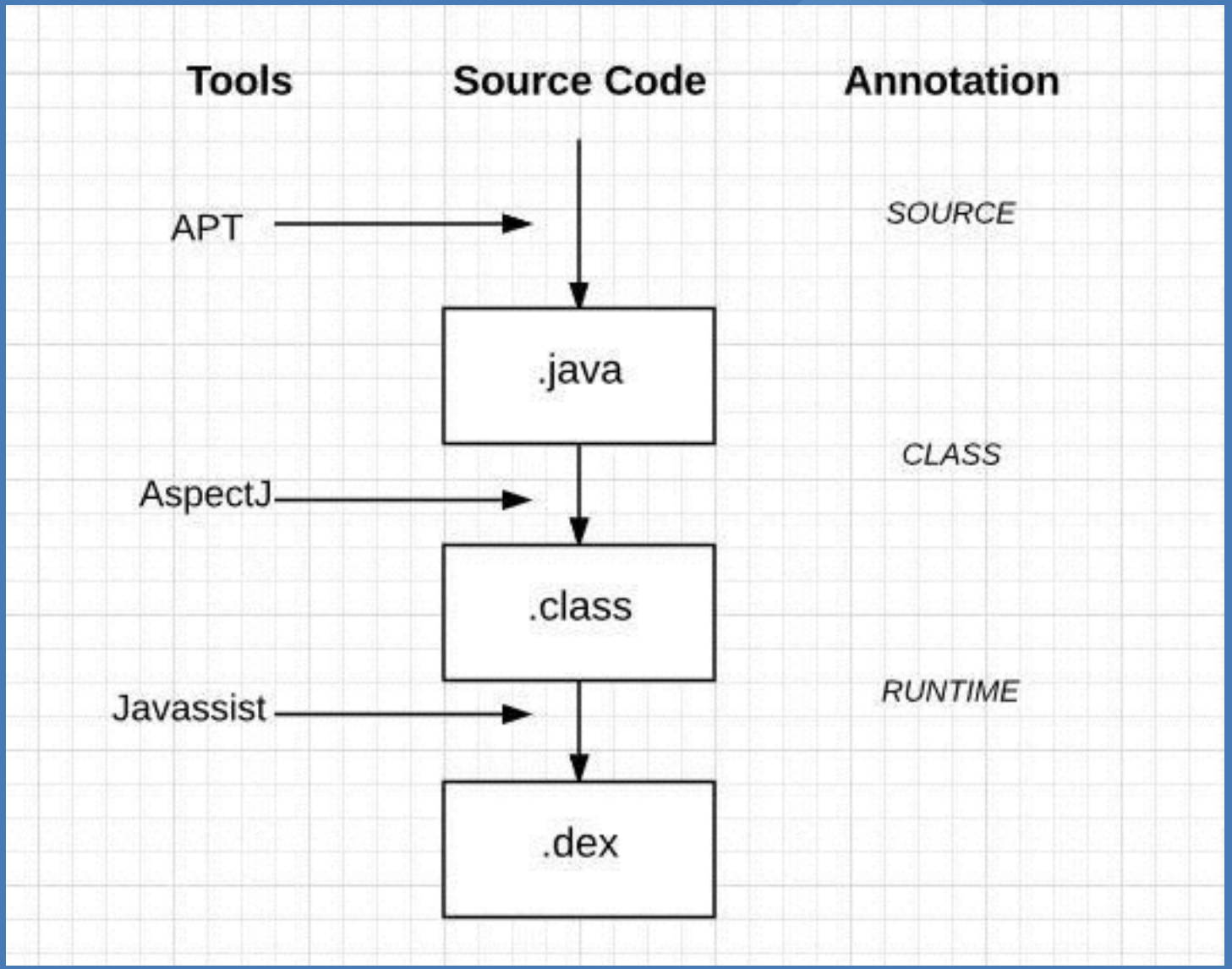
缓存代理: 缓存某方法的返回值，下次执行该方法时，直接从缓存里获取。

软件破解: 使用AOP修改软件的验证类的判断逻辑。

记录日志: 在方法执行前后记录系统日志。

工作流系统: 工作流系统需要将业务代码和流程引擎代码混合在一起执行，那么我们可以使用AOP将其分离，并动态挂接业务。

权限验证: 方法执行前验证是否有权限执行当前方法，没有则抛出没有权限执行异常，由业务代码捕捉。


Tools
Source Code
Annotation
APT
SOURCE
.java
CLASS
AspectJ
.class
RUNTIME
Javassist
.dex

JAVA中Aop的具体实现方式

1、JDK动态代理

java通过实现InvocationHandler接口，可以实现对一个类的动态代理，通过动态代理，我们可以生成代理类从而在代理类方法中，在执行被代理类方法前后，添加自己的实现内容，从而实现Aop。

优点：动态代理java自身支持，不需要引入外部库，在运行期通过接口动态生成代理类

缺点：首先代理类必须实现一个接口，如果没实现接口会抛出一个异常。第二性能影响，因为动态代理使用反射的机制实现的，首先反射肯定比直接调用要慢

http://rejoy.iteye.com/blog/1627405

2、动态字节码生成CGLib

在运行期，目标类加载后，动态构建字节码文件生成目标类的子类，将切面逻辑加入到子类中，没有接口也可以织入，但扩展类的实例方法为final时，则无法进行织入。

可以使用Cglib来实现动态字节码生成，这是一个强大的，高性能，高质量的Code生成类库，它可以在运行期扩展Java类与实现Java接口。CGLIB包的底层是通过使用一个小而快的字节码处理框架ASM，来转换字节码并生成新的类。

优点：可以织入没有接口的类；运行时生成，减少不必要的生成开销；通过字节码生成子类，而不是反射方式去调用代理类

缺点：不能织入final方法；运行时生成子类，说明会有生成开销，并且可能生成大量子类

http://blog.csdn.net/zghwaicsdn/article/details/50957474

3、自定义类加载器,Javassist

在运行期，目标加载前，将切面逻辑加到目标字节码里。

可以对绝大部分类进行织入，但代码中如果使用了其他类加载器，则这些类将不会被织入。

Javassist是一个编辑字节码的框架，可以让你很简单地操作字节码。它可以在运行期定义或修改Class。使用Javassist实现AOP的原理是在字节码加载前直接修改需要切入的方法。这比使用Cglib实现AOP更加高效，并且没太多限制.

优点：可以织入绝大部分类；运行时生成，减少不必要的生成开销；通过将切面逻辑写入字节码，减少了生成子类的开销，不会产生过多子类

缺点：运行时加入切面逻辑，产生开销；比CgLib慢点

代表框架：热修复框架HotFix 、CDN加速等

http://yonglin4605.iteye.com/blog/1396494

4、ASM

ASM 是一个 Java 字节码操控框架。它能够以二进制形式修改已有类或者动态生成类。ASM 可以直接产生二进制 class 文件，也可以在类被加载入 Java 虚拟机之前动态改变类行为。ASM 从类文件中读入信息后，能够改变类行为，分析类信息，甚至能够根据用户要求生成新类。

从上面的描述可以看出，ASM可以在编译期直接修改编译出的字节码文件，也可以像javassit一样，在运行期，类文件加载前，去修改字节码。两者的区别在于，一个将所有需要AOP的类都事先修改了，一个在运行时需要才去修改。

优点：可以织入绝所有类；两者生成方式，可以根据需求选择

缺点：修改字节码，需要对class文件比较熟悉，编写过程复杂

http://www.cnblogs.com/mosthink/p/6295806.html

5、AspectJ

AspectJ是一个面向切面的框架，它扩展了Java语言。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。比较成熟，在Java Web开发中大量使用。
和ASM一样，Aspectj有静态编译和动态编译的优点，供程序员选择。另外Aspectj其编码更为简洁，是Android开发中，实现AOP的首选。

优点：可以织入绝所有类；两者生成方式，可以根据需求选择；编写简单，功能强大

缺点：需要使用ajc编译器编译，ajc编译器是java编译器的扩展，具有其所有功能

代表框架： Hugo(Jake Wharton) ,

http://git.meiyou.im/Android/JetAop

6、APT

自定义一个AbstractProcessor，在编译期去解析编译的类，并且根据需求生成一个实现了特定接口的子类(代理类)，和JDK动态代理不同的是，代理类是在编译期生成的。常见的一些Android的IOC框架中有大量应用(就是通过注解代替findviewbyid等方法)。

优点：可以织入绝所有类；编译期代理，减少运行时消耗

缺点：需要使用apt编译器编译；需要手动拼接代理的代码(其实是整个字符串)；生成大量代理类

代表框架：DataBinding,Dagger2, ButterKnife, EventBus3 、DBFlow、AndroidAnnotation

其他的一些工具：

DexMaker: Dalvik 虚拟机上，在编译期或者运行时生成代码的 Java API。

ASMDEX: 一个类似 ASM 的字节码操作库，运行在Android平台，操作Dex字节码。

xposed: 需要root 是因为它要劫持zygote进程来hook系统方法；

Dexposed: 阿里实现，Dexposed支持从Android2.3到4.4 ,Dalvik

Byte Buddy byte-buddy （号称性能最快的； Mockito, Hibernate, Google's Bazel build system ）

Proxetta jodd

。。。。

See More: https://stackoverflow.com/questions/2261947/are-there-alternatives-to-cglib

Dexposed

支持的系统版本：

Dalvik 2.3
Dalvik 4.0~4.4
不支持的系统版本：

Dalvik 3.0
ART 5.1
ART M
测试中的系统版本：

ART 5.0
未经测试的系统版本：

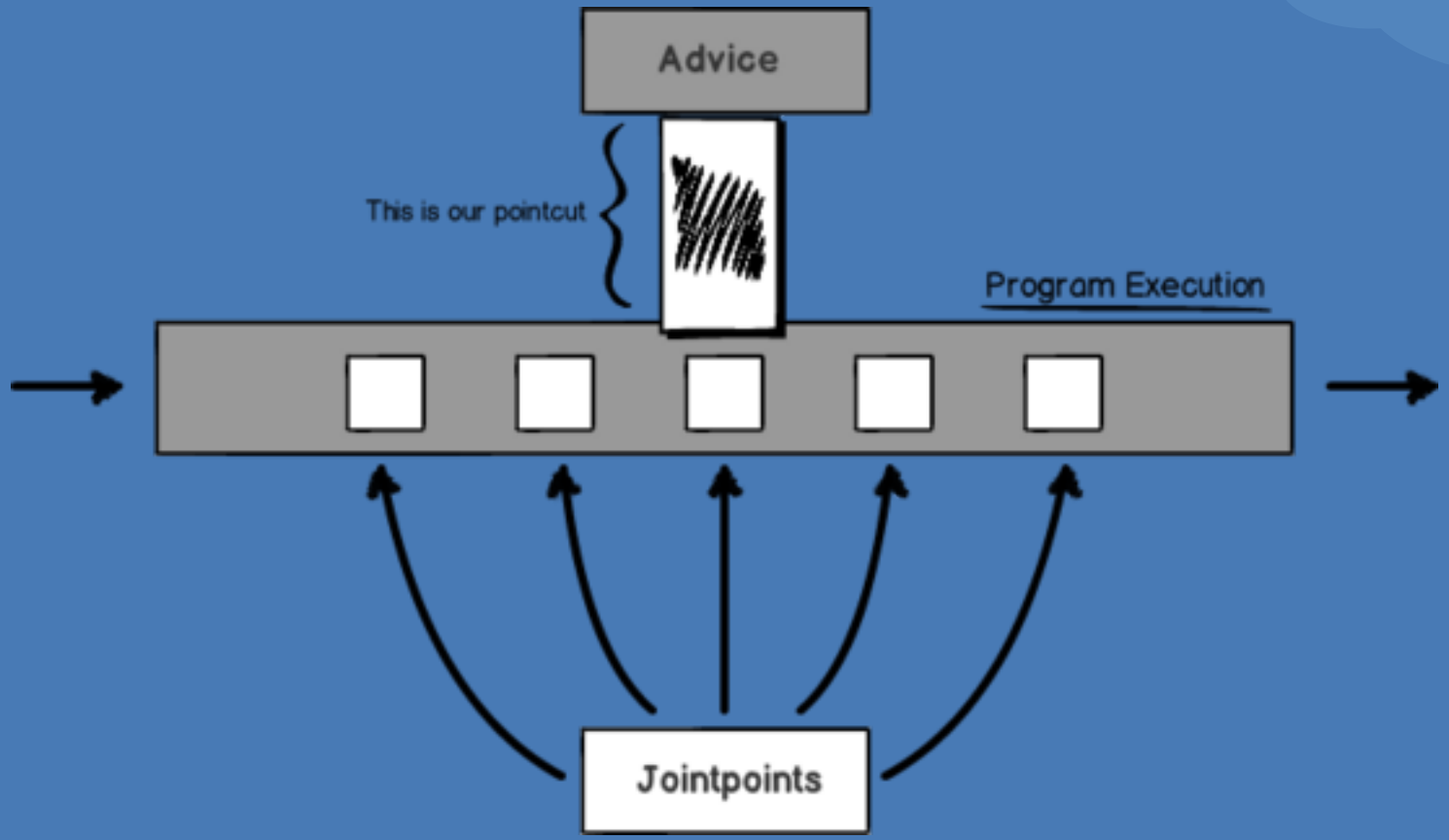
Dalvik 2.2

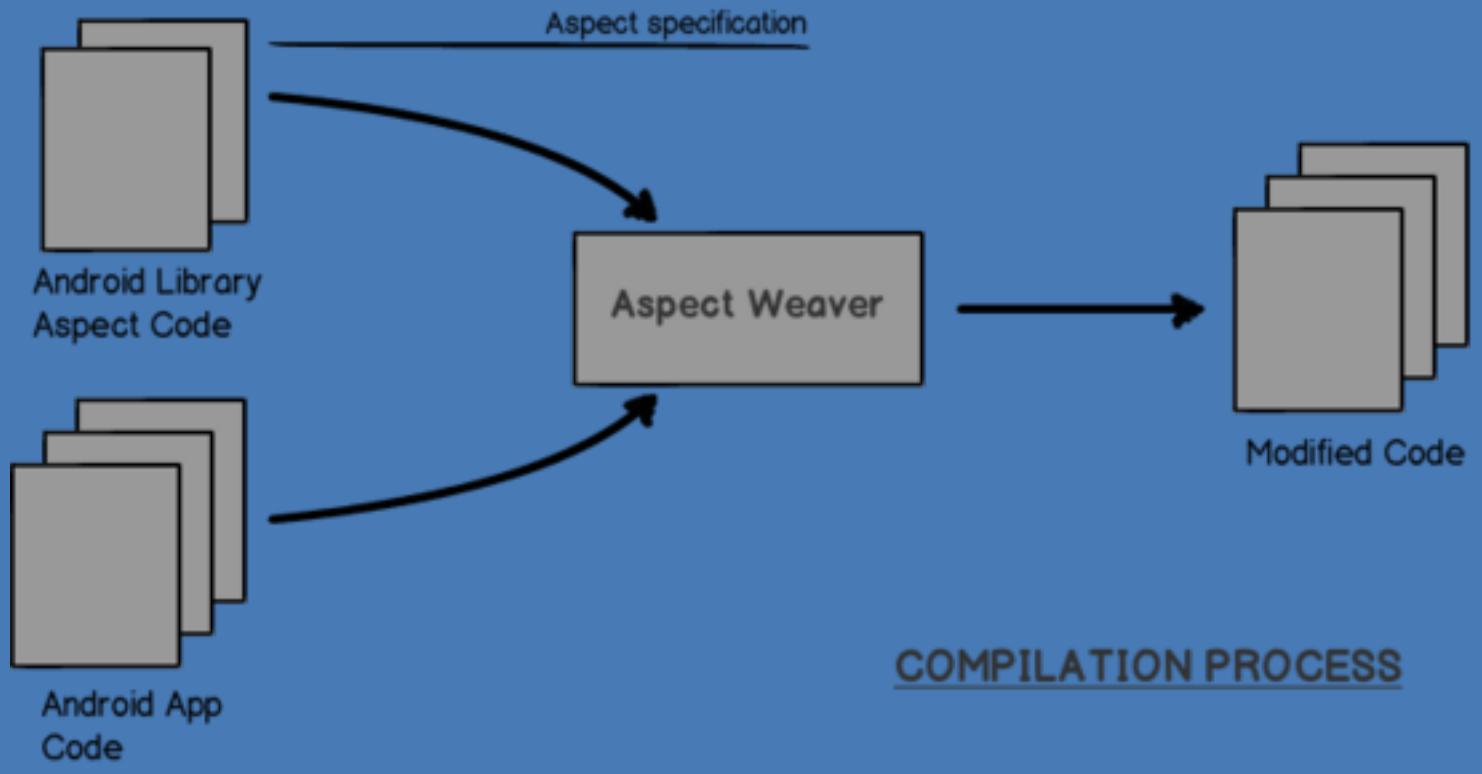
AspectJ是Eclipse出品的Aop框架，可以帮助我们进行很方便的Aop编程

官网链接: http://www.eclipse.org/aspectj/

优点

无侵入性
学习成本低
支持各种格式，并且很成熟


Advice
This is our pointcut
Program Execution
Jointpoints



COMPILATION PROCESS

简单理解：

在AspectJ中，工作流程为：

1. 寻找可以插入的点（joinPoint)
2. 我就插这了（PointCut）
3. 我要插什么（Advice），

主要是插入这些方法：

a. Before（前面干什么），
b. After（后面干什么），
c. Around（直接换了，用我的逻辑）

Show Me The Code

http://git.meiyou.im/Android/Android/wikis/aspectjdemo

THANKS