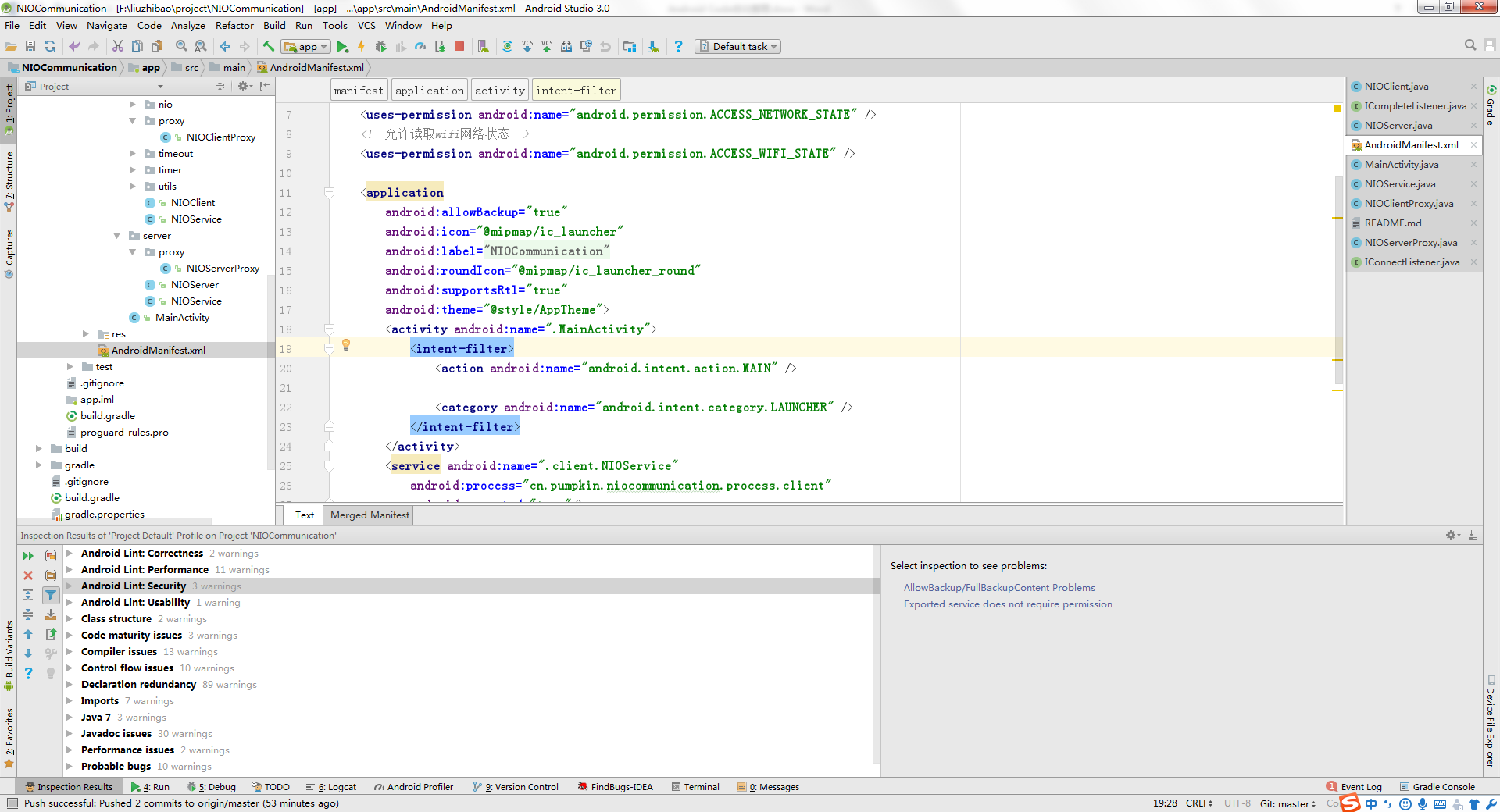
Android Code优化整理

|  |  |  |
| --- | --- | --- |
| 版本号 | 描述 | 日期(更新日期)/更新人 |
| 1.0 | Android code 初步优化整理 | 2018/07/11 刘志保 |
| 2.0 | Android Annotations | 2018/09/25 刘志保 |

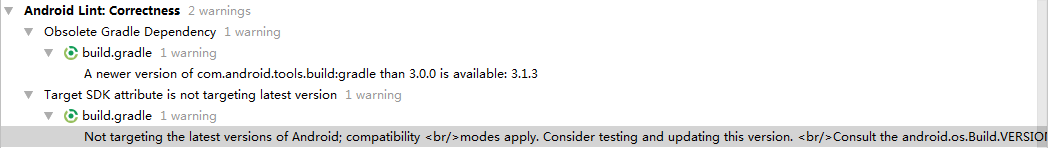
这里主要介绍根据android studio的Analyze菜单分析的结果进行评估和修正程序不合理.

先看看 Analyze->Inspect code,对整个工程(使用的代码: <https://github.com/MMLoveMeMM/NIOCommunication.git>),不要过滤了,全部改!



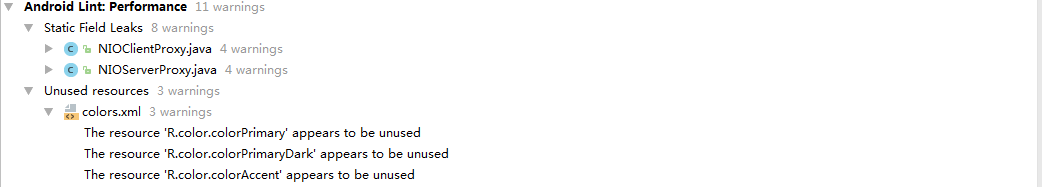
注意左下方框内的项目.

<1> : Android Lint Correctness

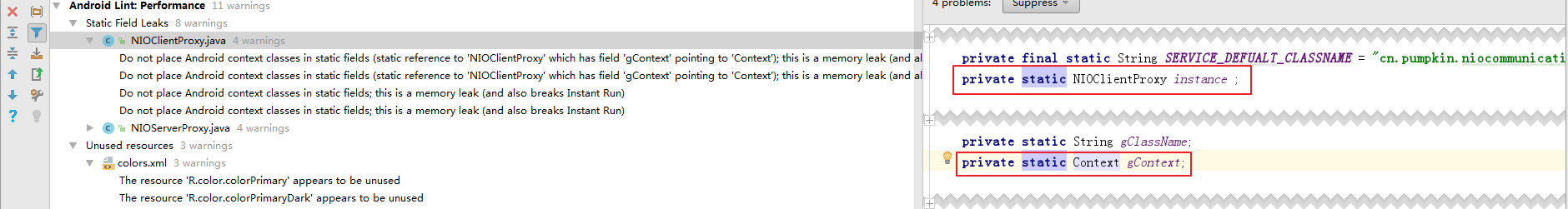


这个是两个警告warning,第一个是提示有更高版本的build.gradle---处理建议:不推荐更新gradle版本,忽略.第二个是提示目前没有SDK=26的目标设备---处理建议:忽略,当然你可以改为你当前的设备版本号,不过不是特别重要.

<2> : Android Lint Performance



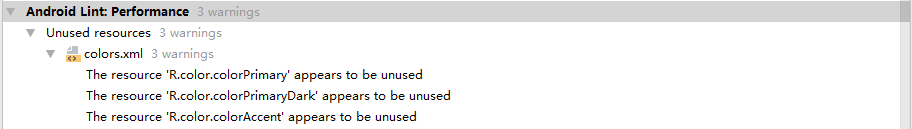
第一个static field leaks里面使用了静态变量:



这个里面由于开发的时候使用了单实例,导致Context上下报文也使用static,这样就有可能Activity销毁了,Context没有销毁,导致内存无法释放.

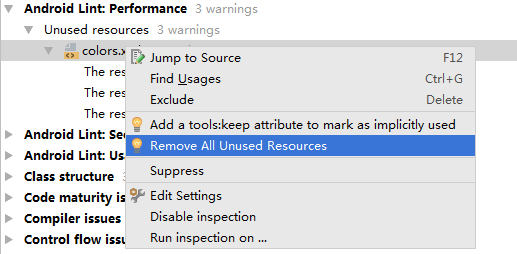
改进措施: NIOClientProxy.java和NIOServerProxy.java类不使用单实例吧,改为普通的对象即可.

改完以后:

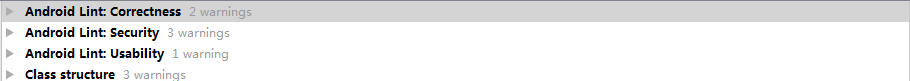


就已经没有了.

下面是colors.xml文件,那些颜色值没有使用,直接在colors.xml右击光标选择下面所示菜单即可.

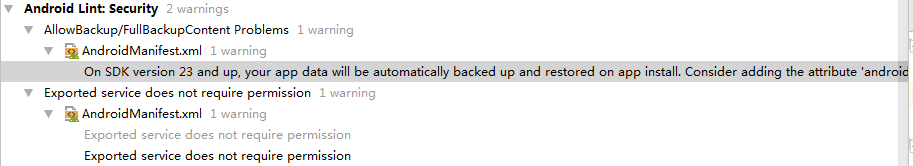


执行后,重新Inspect code菜单:

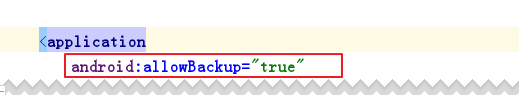


那么Android Lint: Performance就消除了.

<3> : Android Lint: Security



第一个点:



红色框内可以在高于SDK 23的版本可以不需要显示指明(我的工程是26),所以可以删除.

第二个是远程服务对外暴露,可以设置一个权限,就可以消除这个了

先在<manifest>标签下加入

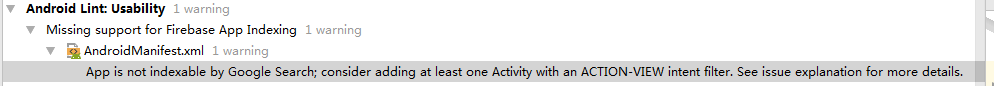
<permission android:protectionLevel="normal" android:name="oem.permission.SENDMAIL"></permission>

然后在<service>标签下

android:permission="oem.permission.SENDMAIL"

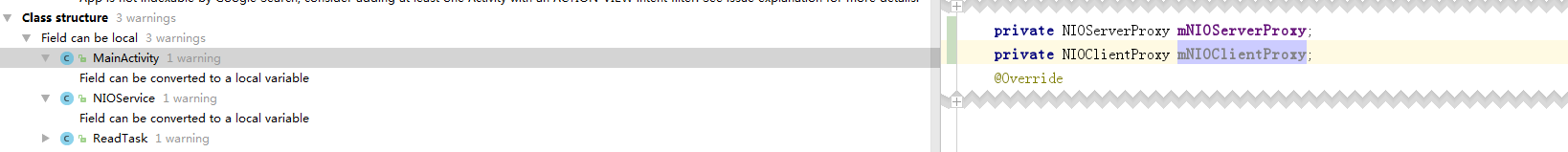
不过上面这两点可以忽略.

<4> Android Lint: Usability



这个可以忽略.

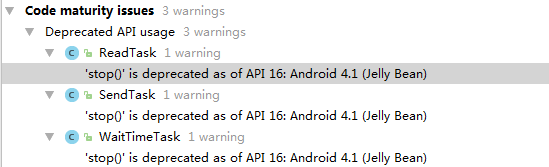
<5>Class Structure



上面三个警告1.变量定义了,但是没有真正使用;2.变量在局部中使用,但是定义为全部变量了,即扩大作用域了.这个修改太容易了,将变量调整一下定义位置即可,这个在后面的文档中会说关于变量生命周期的分析.

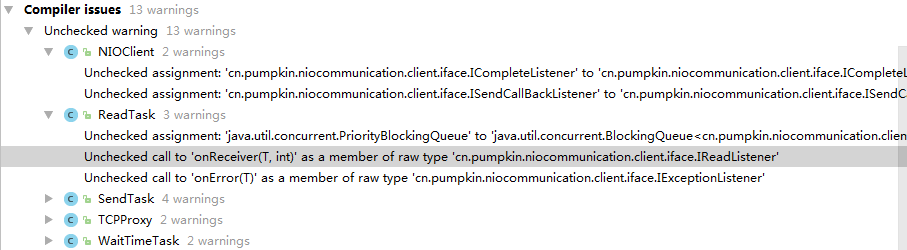
<6> : Code maturity issues

版本或其他兼容性问题:



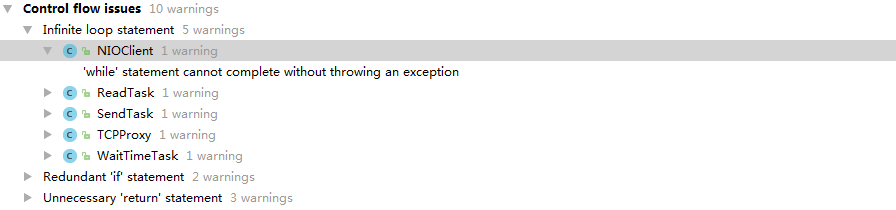
<7> : Compiler issues

这种基本上是可能,可能出现编译性问题,可能需要增加大量判断不为null才行.



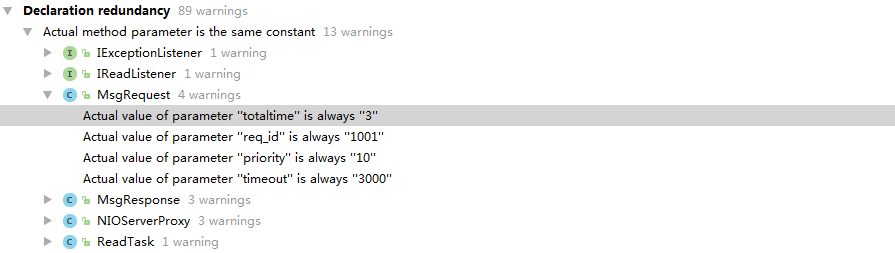
<8> : Control flow issues

这些主要是程序控制流程语句可能存在不合理



<9> : Declaration redundancy

这些很好理解吧,基本是那些本身是常量的,结果还依赖多余的声明变量去处理



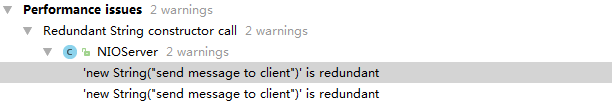
<10> Imports

这个最简单,也比较实用,就是很多java类中开发过程导入多余没有用到的类库,这个建议检查代码的时候使用.



<11> Performance issues

这个一定要处理,至少尽力处理一下.



其他还有几个,按照提示进行修改就可以了,尽量保持代码的干净

对上面进行扩展内容 :

下面通过注解库 support-annotations 来帮助开发者及早发现问题

首先配置环境:

compile 'com.android.support:support-annotations:24.2.0'

详细如下:

Android Studio 内置了代码检查工具 Lint，可在菜单栏选择  Analyze > Inspect Code 执行相应的代码检查，代码检查能够根据推断一些不合法的潜在问题，有助于在开发阶段发现开发者因为主管原因导致的一下代码问题，Android 官方提供了注解库 support-annotations 来帮助开发者及早发现问题，下面是常用的一些注解，主要内容如下：[来自网络,自己不做详细分析了,都是基本的]

1. Nullness注解
2. 资源注解
3. 线程注解
4. 值约束注解
5. 权限注解
6. 返回值注解
7. CallSuper注解
8. Typedef注解
9. 可访问性注解

Nullness注解

使用 Nullness 注解可以检查给定变量、参数和返回值是否允许 null 值，具体如下：

* @Nullable ：表示可以为 null 的变量、参数或返回值，
* @NonNull ：表示不可为 null 的变量、参数或返回值。

1@NonNull  
2@Override  
3public View onCreateView(String name, @NonNull Context context,@NonNull AttributeSet attrs) {  
4    //...  
5}

资源注解

资源注解的使用可使得在源码阶段让编辑器检查书写的不规范，也可在一定程度上优化代码结构，下面是常见的资源注解如下：

* @StringRes：表示检查是否包含R.string引用
* @ColorRes：表示检查是否包含R.color引用
* @ColorInt：表示检查是否包含表示颜色的整型
* @DrawableRes：表示检查是否包含R.drawable引用
* @DimenRes：表示检查是否包含R.dimen引用
* @InterpolatorRes：表示检查是否包含插值器引用

线程注解

线程注解可以检查某个方法是否从某个特定类型的线程中调用，支持一下线程注解,具体如下：

* @MainThread：表示主线程
* @UiThread：表示 UI 线程
* @WorkerThread：表示工作线程
* @BinderThread：表示Binder线程
* @AnyThread：表示任何一个线程

上述注解中 @MainThread 和 @UiThread 在大多时候表示的都是同一线程，如果应用中带有多个试图，UI 线程可与主线程不同，故可使用 @UIThread 标注与应用的视图层次相关联的方法，使用 @MainThread 仅标注与应用生命周期相关联的方法。线程注解最常用的一个用途是 AsyncTask 使用中的方法替换，因为 AsyncTask 会执行后台操作并将结果发布到 UI 线程。

值约束注解

使用值约束注解可验证传递的参数的值的合法性，可以借此指定参数的设置范围，可在一定程度上减少代码在主观程度上出现的错误，常见的值约束注解如下：

* @IntRange：表示可以验证整型参数是否在指定范围内
* @FloatRange：表示可以验证浮点型参数是否在指定范围内
* @Size：表示可以验证集合、数组、字符串参数是否在指定范围内，可指定最大值、最小值以及确切值

上面的注解有一些可使用的参数，如 from、to、min 等，使用时具体在某个注解体重查看定义即可。

权限注解

权限注解 @RequiresPermission 可以验证方法调用方的权限，即当使用了权限注解的方法时会检查有没有指定的权限，如果没有则会提示要在 AndroidManifest.xml 文件中申明权限，如果是危险权限还有进行权限动态申请，使用方式参考如下：

1/\*\*  
 2 \* 单个权限检查  
 3 \* @param message  
 4 \*/  
 5@RequiresPermission(Manifest.permission.WRITE\_EXTERNAL\_STORAGE)  
 6public void setMessage(String message) {  
 7}  
 8  
 9/\*\*  
10 \* 全部权限检查  
11 \* @param message  
12 \*/  
13@RequiresPermission(allOf = {  
14        Manifest.permission.WRITE\_EXTERNAL\_STORAGE,  
15        Manifest.permission.READ\_EXTERNAL\_STORAGE})  
16public void setMesage(String message) {  
17}  
18  
19/\*\*  
20 \* 某个权限检查  
21 \* @param message  
22 \*/  
23@RequiresPermission(anyOf = {  
24        Manifest.permission.WRITE\_EXTERNAL\_STORAGE,  
25        Manifest.permission.READ\_EXTERNAL\_STORAGE})  
26public void setMesage(String message) {  
27}

返回值注解

返回值注解 @CheckResult 会检查某个方法的返回值是否被使用，如果没有被使用，则会根据 suggest 配置建议使用相同公民没有返回值的另一个方法，如果返回值使用了，则和未加该注解的方法一样，使用方式参考如下：

1@CheckResult(suggest="#enforcePermission(String,int,int,String)")  
2public  int checkPermission(@NonNull String permission, int pid, int uid){  
3    return 0;  
4}

如果没有使用返回值提示如下：

返回值注解

当返回值没有被使用，则会建议使用相同功能没有返回值的另一个方法，简而言之，返回值注解 @CheckResult 能够表示某个方法实际使用的时方法本身的处理还是方法最终的处理结果。

CallSuper注解

使用 @CallSuper 注解会验证子类的重写方法是否调用父类的实现，这样约束的好处是可保证父类的实现不会修改，当然，如果不使用该注解，子类重写父类的方法可以不调用弗父类的默认实现，具体参考如下：

1/\*\*  
 2 \* 父类  
 3 \* @CallSuper注解的使用  
 4 \*/  
 5public class Test {  
 6    //使用@CallSuper注解，子类重写该方法时必须调用该方法  
 7    @CallSuper  
 8    protected void onCreate(){  
 9  
10    }  
11}

下面是 Test 类的实现类：

1/\*\*  
 2 \* 子类  
 3 \* @CallSuper注解的使用  
 4 \*/  
 5public class TestImpl extends Test{  
 6    @Override  
 7    protected void onCreate() {  
 8        super.onCreate();  
 9        /\*\*  
10         \* 如果不调用父类的方法，则会提示  
11         \* Some methods, such as View#onDetachedFromWindow, require that you also call the super implementation as part of your method.  
12         \*/  
13    }  
14}

Typedef注解

使用 @IntDef 和 @StringDef 注解 可以创建整型和字符串的枚举注解来验证其他代码中使用的某些整型和字符串，可以保证代码中的某些常量整型或常量字符串是某些具体定义的常量集，这两个注解的位置只能是注解。

开发中总会使用到枚举，枚举在一定程度上可使得代码结构更清晰，但枚举的使用会增加内存的开销，这里可以用 Typedef 注解的方式来代替枚举，下面是 Tyoedef 注解的使用，参考如下：

1/\*\*  
 2 \* Typedef 注解的定义  
 3 \*/  
 4public class ActionType {  
 5  
 6    public static final int ACTION\_TYPE\_0 = 0;  
 7    public static final int ACTION\_TYPE\_1 = 1;  
 8    public static final int ACTION\_TYPE\_2 = 2;  
 9  
10    @Retention(RetentionPolicy.SOURCE)  
11    @IntDef({ACTION\_TYPE\_0,ACTION\_TYPE\_1,ACTION\_TYPE\_2})  
12    public @interface ActionTypeDef{  
13  
14    }  
15}

下面是上述 Typedef 注解的使用方式，参考如下：

1/\*\*  
 2 \* Typedef注解的使用  
 3 \* @param value  
 4 \*/  
 5private void setValue(@ActionType.ActionTypeDef int value) {  
 6    switch (value) {  
 7        case ActionType.ACTION\_TYPE\_0:  
 8            break;  
 9        case ActionType.ACTION\_TYPE\_1:  
10            break;  
11        case ActionType.ACTION\_TYPE\_2:  
12            break;  
13  
14//        case 100://不能使用未定义的整型  
15//            break;  
16    }  
17}

可见 Typedef 注解约束了使用到的某些整型，当然还可以是字符串，这样也能达到枚举的作用。

可访问性注解

可访问性注解是 @VisibleForTesting 和 @Keep 可以表示方法、字段、类的可访问性。具体如下：

* @VisibleForTesting：表示注解的某个代码块的可见性高于能够测试时需要的水平
* @Keep：表示被注解的代码块将不会被混淆。

总结

最常用的可能就是资源注解，如 @StringRes、@ColorRes、@ColorInt等，还有Typeof 注解，该注解可以在替换枚举在 Android 开发中带来的性能影响，如果平时留意这些注解在 Android 源码中也经常使用，所以可在开发过程中尝试去使用这些注解以进行必要的代码检查。