ProGuard使用说明

目录

[1 ProGuard简介和工作流程 3](#_Toc460504584)

[2 PrgGuard环境配置和使用 4](#_Toc460504585)

[2.1 运行PrgGuard需要以下依赖： 4](#_Toc460504586)

[2.2 如何运行ProGuard 4](#_Toc460504587)

[3 PrgGuard配置文件使用 5](#_Toc460504588)

[3.1 Entry points的概念 5](#_Toc460504589)

[3.2 会用到的指令参数说明 6](#_Toc460504590)

[3.2.1 Modifier 6](#_Toc460504591)

[3.2.2 Class Specifications 6](#_Toc460504592)

[3.3 基本指令 6](#_Toc460504593)

[3.3.1 -basedirectory directoryname： 6](#_Toc460504594)

[3.3.2 -injars class\_path 6](#_Toc460504595)

[3.3.3 -outjars class\_path 7](#_Toc460504596)

[3.3.4 -libraryjars class\_path 7](#_Toc460504597)

[3.3.5 -skipnonpubliclibraryclasses 7](#_Toc460504598)

[3.3.6 –dontskipnonpubliclibraryclasses 7](#_Toc460504599)

[3.3.7 -dontskipnonpubliclibraryclassmembers 7](#_Toc460504600)

[3.3.8 -keepdirectories [directory\_filter] 7](#_Toc460504601)

[3.3.9 -target version 8](#_Toc460504602)

[3.3.10 -forceprocessing 8](#_Toc460504603)

[3.3.11 -keep [,modifier,...] class\_specification 8](#_Toc460504604)

[3.3.12 -keepclassmembers [,modifier,...] class\_specification 8](#_Toc460504605)

[3.3.13 -keepclasseswithmembers [,modifier,...] class\_specification 8](#_Toc460504606)

[3.3.14 -keepnames class\_specification 9](#_Toc460504607)

[3.3.15 -keepclassmembernames class\_specification 9](#_Toc460504608)

[3.3.16 -keepclasseswithmembernames class\_specification 9](#_Toc460504609)

[3.3.17 -printseeds [filename] 9](#_Toc460504610)

[3.3.18 -dontshrink 9](#_Toc460504611)

[3.3.19 -printusage [filename] 9](#_Toc460504612)

[3.3.20 -dontoptimize 9](#_Toc460504613)

[3.3.21 Optimizations 10](#_Toc460504614)

[3.3.22 -optimizationpasses n 10](#_Toc460504615)

[3.3.23 -assumenosideeffects class\_specification 10](#_Toc460504616)

[3.3.24 –dontobfuscate 10](#_Toc460504617)

[3.3.25 -printmapping [filename] 10](#_Toc460504618)

[3.3.26 -applymapping filename 10](#_Toc460504619)

[3.3.27 -obfuscationdictionary filename 11](#_Toc460504620)

[3.3.28 -classobfuscationdictionary filename 11](#_Toc460504621)

[3.3.29 -packageobfuscationdictionary filename 11](#_Toc460504622)

[3.3.30 –useuniqueclassmembernames 11](#_Toc460504623)

[3.3.31 -dontusemixedcaseclassnames 11](#_Toc460504624)

[3.3.32 -keeppackagenames [package\_filter] 11](#_Toc460504625)

[3.3.33 -flattenpackagehierarchy [package\_name] 11](#_Toc460504626)

[3.3.34 -repackageclasses [package\_name] 11](#_Toc460504627)

[3.3.35 -keepattributes [attribute\_filter] 12](#_Toc460504628)

[3.3.36 -dontpreverify 12](#_Toc460504629)

[3.3.37 -verbose 12](#_Toc460504630)

[3.3.38 -dontnote [class\_filter] 12](#_Toc460504631)

[3.3.39 -dontwarn [class\_filter] 12](#_Toc460504632)

[3.3.40 -ignorewarnings 13](#_Toc460504633)

[3.3.41 -printconfiguration [filename] 13](#_Toc460504634)

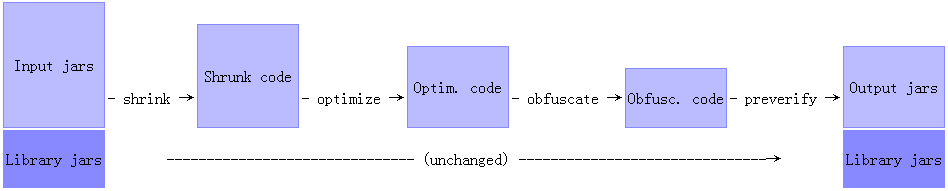
[3.3.42 -dump [filename] 13](#_Toc460504635)

[4 反射的处理 14](#_Toc460504636)

[5 PrgGuard的基本使用demo 14](#_Toc460504637)

# ProGuard简介和工作流程

ProGuard能够通过压缩、优化、混淆、预检等操作，检测并删除未使用的类,字段,方法和属性，分析和优化字节码，使用简短无意义的名称来重命名类，字段和方法。从而使代码更小、更高效、更难进行逆向工程。



上图就是ProGuard的工作流程，分别会经过四个阶段：

1. **压缩（Shrink）:**在压缩处理这一步中，用于检测和删除没有使用的类，字段，方法和属性
2. **优化（Optimize）:**在优化处理这一步中，对字节码进行优化，并且移除无用指令
3. **混淆（Obfuscate）:**在混淆处理这一步中，使用a,b,c等无意义的名称，对类，字段和方法进行重命名
4. **预检（Preveirfy）:**在预检这一步中，主要是在Java平台上对处理后的代码进行预检

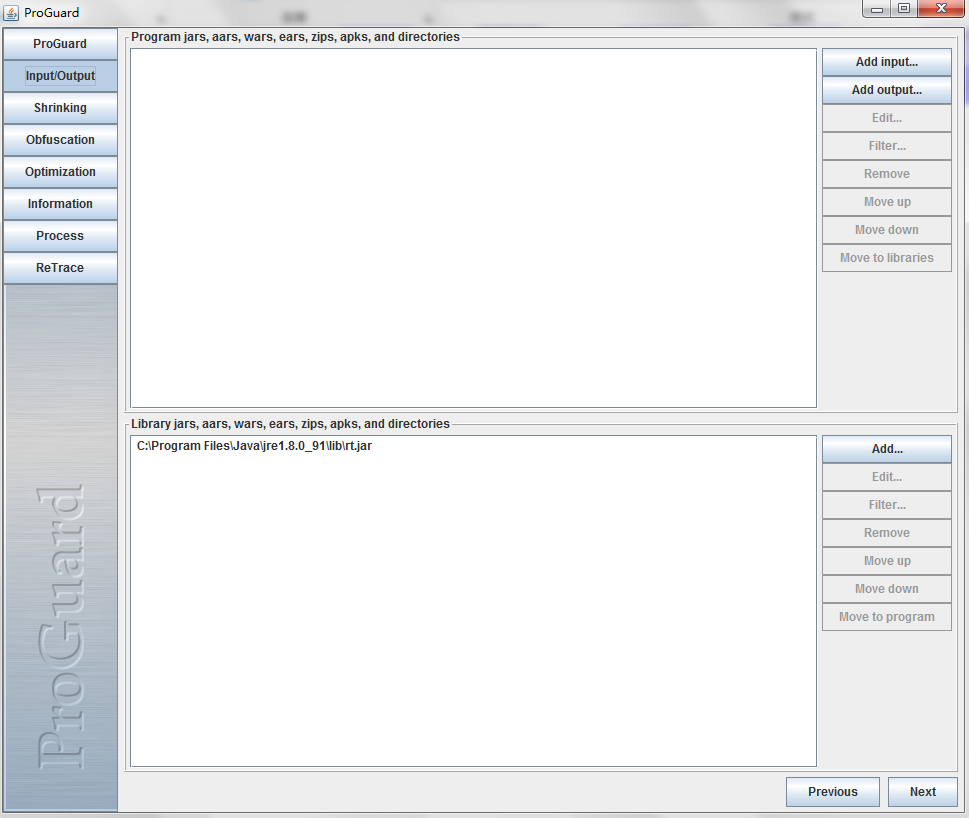
以上四个步骤都是**可选**的，我们可以通过配置脚本来决定其中的哪几个步骤。比如我们可以配置只压缩和混淆，不进行优化，不进行预检。

ProGuard的官网有使用指导：<http://proguard.sourceforge.net/>

# PrgGuard环境配置和使用

## 运行PrgGuard需要以下依赖：

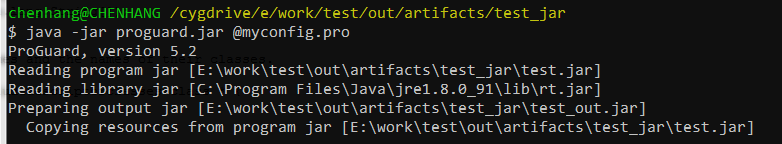
1. Java运行环境
2. proguard.jar或者proguardgui.jar。proguardgui提供了一个简单的配置界面（如下图），可以在上面进行配置，而progua.jar则是使用配置文件进行处理



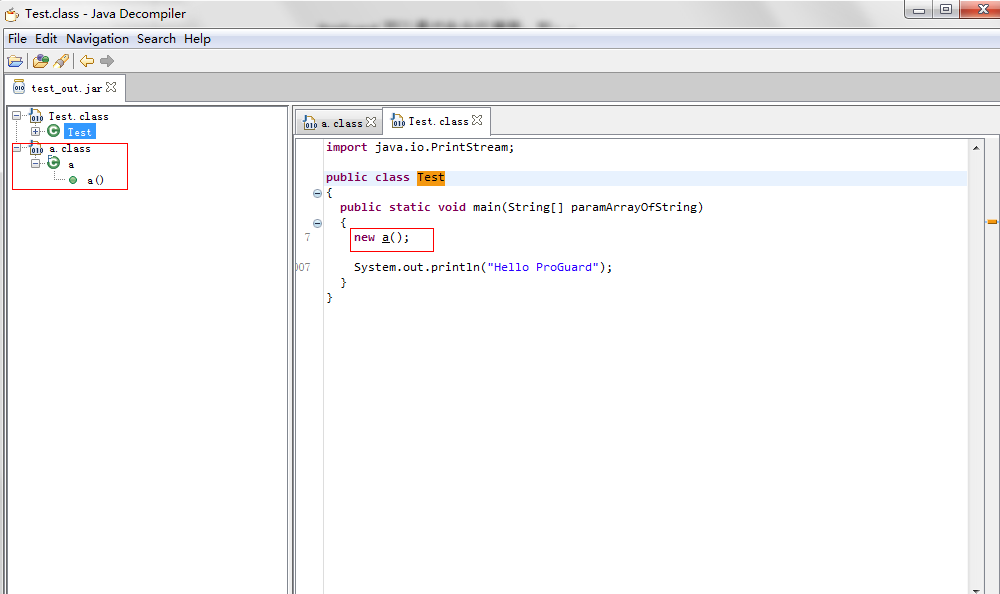
## 如何运行ProGuard

ProGuard可以通过命令行调用，如：

* **java -jar proguardgui.jar**：启动图形化配置界面
* **java -jar proguard.jar @config.file –options ...**：通过配置文件进行ProGuard处理



执行成功后，用jd-gui打开处理后的jar文件：



可以发现，类已经被混淆处理了。

# PrgGuard配置文件使用

## Entry points的概念

这里，我们引入Entry points的概念。Entry points是在ProGuard过程中**不会处理**的类或者方法。

在Shrink的步骤中，ProGuard会递归遍历，搜索使用了哪些类和成员，对于没有使用的类和类成员，就会在压缩阶段丢弃。

接下来在Optimize阶段，那些非Entry points的类、方法都会被设置为private、static或者final，没有使用的参数会被移除，此外，有些方法会被标记为内联。

在Obfuscate的步骤中，ProGuard会对非Entry points的类和方法进行重命名。

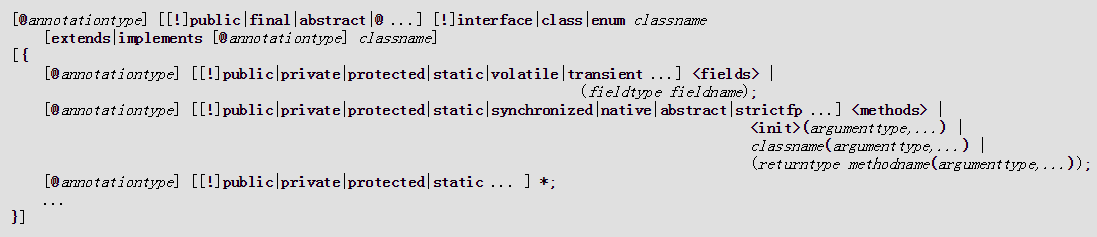
## 会用到的指令参数说明

### Modifier

* Includedescriptorclasses：一般用于保证native方法名，确保方法的参数类型不会重命名，确保方法签名不会被改变，这样才能跟native libraries相匹配。
* Allowshrinking：允许压缩
* Allowoptimization：允许优化
* Allowobfuscation：允许混淆名称

### Class Specifications

Class Specifications是用来描述类和方法的模板，下面是这个模板的格式：

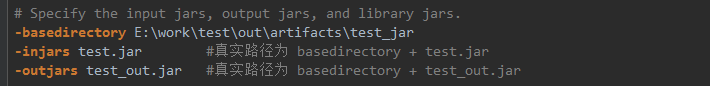


其中，[]中的内容是可选，名称可以使用通配符，<init>匹配构造函数、<fields>匹配成员、<methods>匹配方法，详细请参考：<http://proguard.sourceforge.net/manual/usage.html#keepoptionmodifiers>

## 基本指令

### -basedirectory directoryname：

在配置文件中出现的相对路径均是相对于该路径，如图：



### -injars class\_path

指定处理的jar包（或者aars, wars, ears, zips, apks, directories）等，这个jar包里面的类将会被ProGuard处理并写入到输出的jar包里去。一般非class文件会不做任何处理直接直接复制到输出文件中，injars可以多次使用，引入不同的需要处理的文件。

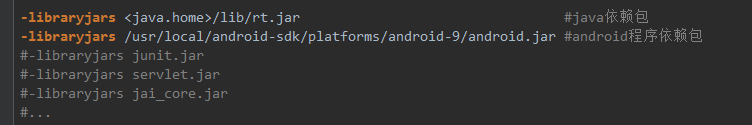
注意，该选项可以指定一个目录，那么该目录下所有文件都会被当作input file处理。

### -outjars class\_path

设置处理完成后的输出文件路径

### -libraryjars class\_path

指定要处理应用程序的jar(或者aars, wars, ears, zips, apks, directories)，这些文件不会包含到输出文件中。一般是指被处理文件所依赖的一些jar包，而那些jar包是不需要被处理以及写入到输出文件的。比如：



### -skipnonpubliclibraryclasses

忽略library里面非public修饰的类。从而加快ProGuard的处理速度和降低ProGuard的使用内存。一般而言，library里的非公开类是不能被程序使用的，忽略掉这些类可以加快混淆速度。但是请注意，**有一种特殊情况：有些人编写的代码与类库中的类在同一个包下，而且对该包的非public类进行了使用，在这种情况下，就不能使用该选项了。**

### –dontskipnonpubliclibraryclasses

**不忽略**library里面非public修饰的类

### -dontskipnonpubliclibraryclassmembers

指定不忽略非public类里面的成员和方法。ProGuard**默认会忽略类库里非public类里的成员和方法**，但是由于一些3.2.5里面的一些原因，应用程序里可能会用到这些，这时候就需要这个选项来指定不忽略它们。

### -keepdirectories [directory\_filter]

指定要保留在输出文件内的目录。默认情况下，目录会被移除。这会减少输出文件的大小，但如果你的代码引用到它们时可能会导致程序崩溃（如mypackage.MyCalss.class.getResource("")）。这时就需要指定-keepdirectories mypackage。-keepdirectories mydirectory匹配 mydirectory 目录；-keepdirectories mydirectory/\*匹配 mydirectory 的直接子目录；-keepdirectorie mydirectory/\*\*匹配所有子目录，如果没有指定过滤器，所有目录会被保留。

### -target version

指定被处理class文件所使用的java版本，可选的有: 1.0, 1.1, 1.2, 1.3, 1.4, 1.5 (or just 5), 1.6 (or just 6), 1.7 (or just 7), or 1.8 (or just 8).

### -forceprocessing

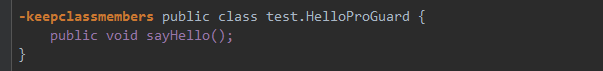
强制输出，即使输出文件已经是最新状态

### -keep [,modifier,...] class\_specification

指定该类以及类的成员和方法为entry points，不被ProGuard混淆

### -keepclassmembers [,modifier,...] class\_specification

指定类的某些成员不被混淆，**注意类名还是会被混淆**，如：

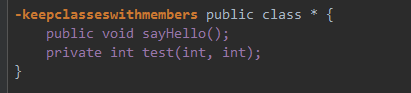


### -keepclasseswithmembers [,modifier,...] class\_specification

通过成员来指定哪些类不被混淆处理。比如可以用来保留包含main方法的类：



如果指定了多条规则，如下，那么必须同时包含sayHello和test两个方法的类才会被保留



.

### -keepnames class\_specification

-keepclassmembers,allowshrinking class\_specification的别名，保留名称不被混淆，但可以被压缩

### -keepclassmembernames class\_specification

-keepclasseswithmembers,allowshrinking class\_specification的别名，保留名称不被混淆，但可以被压缩

### -keepclasseswithmembernames class\_specification

-keepclasseswithmembers,allowshrinking class\_specification的别名

### -printseeds [filename]

把keep匹配的类和方法输出到文件中，可以用来验证自己设定的规则是否生效.

### -dontshrink

指定不进行压缩.

### -printusage [filename]

把没有使用的代码输出到文件中，方便查看哪些代码被压缩丢弃了。.

### -dontoptimize

指定不对输入代码进行优化处理。优化选项是默认打开的。

### -optimizations

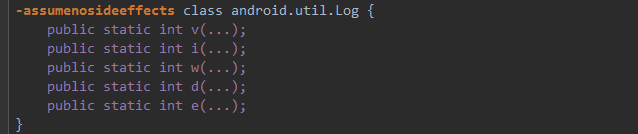
指定混淆是采用的算法，后面的参数是一个过滤器，这个过滤器是谷歌推荐的算法，一般不做更改

### -optimizationpasses n

指定优化的级别，在0-7之间，默认为5.

### -assumenosideeffects class\_specification

可以指定移除哪些方法没有副作用，如在android开发中，如想在release版本可以把所有log输出都移除，可以配置：



那么所有log代码将会在优化阶段被去除。

### –dontobfuscate

指定不进行混淆

### -printmapping [filename]

生成map文件，记录混淆前后的名称对应关系，注意，**这个比较重要**，因为混淆后运行的名称会变得不可读，只有依靠这个map文件来还原。

### -applymapping filename

主要是用来维持两次混淆公用一份mapping，确保相同的代码前后两次混淆后是一样的命名

### -obfuscationdictionary filename

指定外部模糊字典

### -classobfuscationdictionary filename

指定class模糊字典.

### -packageobfuscationdictionary filename

指定package模糊字典

### –useuniqueclassmembernames

类和成员混淆的时候，使用唯一的名字

### -dontusemixedcaseclassnames

不使用大小写混合类名，注意，windows用户必须为ProGuard指定该选项，因为windows对文件的大小写是不敏感的，也就是比如a.java和A.java会认为是同一个文件。如果不这样做并且你的项目中有超过26个类的话，那么ProGuard就会默认混用大小写文件名，导致class文件相互覆盖。

### -keeppackagenames [package\_filter]

保持packagename 不混淆

### -flattenpackagehierarchy [package\_name]

指定重新打包,所有包重命名,这个选项会进一步模糊包名，将包里的类混淆成n个再重新打包到一个个的package中

### -repackageclasses [package\_name]

将包里的类混淆成n个再重新打包到一个统一的package中，会覆盖flattenpackagehierarchy选项

### -keepattributes [attribute\_filter]

混淆时可能被移除下面这些东西，如果想保留，需要用该选项，对于一般注解处理如 -keepattributes \*Annotation\*。

attribute\_filter :

* Exceptions,
* Signature,
* Deprecated,
* SourceFile,
* SourceDir,
* LineNumberTable,
* LocalVariableTable,
* LocalVariableTypeTable,
* Synthetic,
* #EnclosingMethod,
* RuntimeVisibleAnnotations,
* RuntimeInvisibleAnnotations,
* RuntimeVisibleParameterAnnotations,
* RuntimeInvisibleParameterAnnotations,
* AnnotationDefault.

### -dontpreverify

指定不执行预检

### -verbose

把所有信息都输出，而不仅仅是输出出错信息

### -dontnote [class\_filter]

不输出指定类的错误信息.

### -dontwarn [class\_filter]

不打印指定类的警告信息

### -ignorewarnings

遇到警告的时候，忽略警告继续执行ProGuard，不建议添加此项。

### -printconfiguration [filename]

输出当前ProGuard所使用的配置

### -dump [filename]

指定输出所处理的类的结构

# 反射的处理

在代码中，如果用到了反射，混淆会改变类和成员的名字，导致反射找不到相应的类或者方法，所以开发者在混淆的时候，必须把用到了反射的类保留，不进行混淆。一般而言，使用反射一般会有以下方式，可以搜索代码，找到相关的类，然后在混淆配置里面进行保留：

* Class.forName("SomeClass")
* SomeClass.class
* SomeClass.class.getField("someField")
* SomeClass.class.getDeclaredField("someField")
* SomeClass.class.getMethod("someMethod", new Class[] {})
* SomeClass.class.getMethod("someMethod", new Class[] { A.class })
* SomeClass.class.getMethod("someMethod", new Class[] { A.class, B.class })
* SomeClass.class.getDeclaredMethod("someMethod", new Class[] {})
* SomeClass.class.getDeclaredMethod("someMethod", new Class[] { A.class })
* SomeClass.class.getDeclaredMethod("someMethod", new Class[] { A.class, B.class })
* AtomicIntegerFieldUpdater.newUpdater(SomeClass.class, "someField")
* AtomicLongFieldUpdater.newUpdater(SomeClass.class, "someField")
* AtomicReferenceFieldUpdater.newUpdater(SomeClass.class, SomeType.class, "someField")

# reflectClass类使用了反射，保留该类  
**-keep** class package.reflectClass { \*; }

# PrgGuard的基本使用demo

#代码混淆压缩比，在0~7之间，默认为5，一般不做修改  
**-optimizationpasses** 5  
#指定混淆是采用的算法，后面的参数是一个过滤器，这个过滤器是谷歌推荐的算法，一般不做更改  
**-optimizations** !code/simplification/cast,!field/\*,!class/merging/\*  
  
#混合时不使用大小写混合，混合后的类名为小写,windows下必须使用该选项  
**-dontusemixedcaseclassnames**#指定不去忽略非公共库的类和成员  
**-dontskipnonpubliclibraryclasses  
-dontskipnonpubliclibraryclassmembers**#输出详细信息  
**-verbose**#输出类名->混淆后类名的映射关系  
**-printmapping** map.txt  
  
#不做预校验，preverify是proguard的四个步骤之一，Android不需要preverify，去掉这一步能够加快混淆速度。  
**-dontpreverify**#保留Annotation不混淆  
**-keepattributes** \*Annotation\*,InnerClasses  
  
#避免混淆泛型  
**-keepattributes** Signature  
  
#抛出异常时保留代码行号  
**-keepattributes** SourceFile,LineNumberTable  
  
#保留本地native方法不被混淆  
**-keepclasseswithmembernames** class \* {  
 native <methods>;  
}  
  
#保留枚举类不被混淆  
**-keepclassmembers** enum \* {  
 public static \*\*[] values();  
 public static \*\* valueOf(java.lang.String);  
}  
  
#保留Serializable序列化的类不被混淆  
**-keepclassmembers** class \* implements java.io.Serializable {  
 static final long serialVersionUID;  
 private static final java.io.ObjectStreamField[] serialPersistentFields;  
 private void writeObject(java.io.ObjectOutputStream);  
 private void readObject(java.io.ObjectInputStream);  
 java.lang.Object writeReplace();  
 java.lang.Object readResolve();  
}

另外，由于Android平台在使用混淆的时候，还要特别注意要添加以下一些配置：

#保留我们使用的四大组件，自定义的Application等等这些类不被混淆  
**-keep** public class \* extends android.app.Activity  
**-keep** public class \* extends android.app.Application  
**-keep** public class \* extends android.app.Service  
**-keep** public class \* extends android.content.BroadcastReceiver  
**-keep** public class \* extends android.content.ContentProvider  
**-keep** public class \* extends android.app.backup.BackupAgentHelper  
**-keep** public class \* extends android.preference.Preference  
**-keep** public class \* extends android.view.View  
**-keep** public class com.android.vending.licensing.ILicensingService  
  
#保留support下的所有类及其内部类  
**-keep** class android.support.\*\* {\*;}  
  
#保留R下面的资源  
**-keep** class \*\*.R$\* {\*;}  
  
#保留在Activity中的方法参数是view的方法，  
**-keepclassmembers** class \* extends android.app.Activity{  
 public void \*(android.view.View);  
}  
  
#保留我们自定义控件（继承自View）不被混淆  
**-keep** public class \* extends android.view.View{  
 \*\*\* get\*();  
 void set\*(\*\*\*);  
 public <init>(android.content.Context);  
 public <init>(android.content.Context, android.util.AttributeSet);  
 public <init>(android.content.Context, android.util.AttributeSet, int);  
}  
  
#保留Parcelable序列化类不被混淆  
**-keep** class \* implements android.os.Parcelable {  
 public static final android.os.Parcelable$Creator \*;  
}  
  
#对于带有回调函数的onXXEvent的，不能被混淆  
**-keepclassmembers** class \* {  
 void \*(\*\*On\*Event);  
}  
  
#在我们的app中使用了webView需要进行特殊处理  
**-keepclassmembers** class \* extends android.webkit.webViewClient {  
 public void \*(android.webkit.WebView, java.lang.String, android.graphics.Bitmap);  
 public boolean \*(android.webkit.WebView, java.lang.String);  
}  
**-keepclassmembers** class \* extends android.webkit.webViewClient {  
 public void \*(android.webkit.webView, jav.lang.String);  
}  
  
# 在app中与HTML5的JavaScript的交互进行特殊处理，如  
# package com.ljd.example;  
#  
# public class JSInterface {  
# @JavascriptInterface  
# public void callAndroidMethod(){  
# // do something  
# }  
# }  
#我们需要确保这些js要调用的原生方法不能够被混淆，于是我们需要做如下处理  
**-keepclassmembers** class com.ljd.example.JSInterface {  
 <methods>;  
}  
  
#内嵌类经常被混淆，结果在调用的时候就崩溃了，如果需要保留内嵌类，则用以下方法来保留内嵌类，如暴力MyClass里面的内嵌类，$就是用来分割内嵌类和母体的标志  
**-keep** class com.test.MyClass$\* {\*;}

#-----------以下处理反射类---------------