**Launcher Design Doc**

**编 写 人：桌面品牌产品线**

版本历史

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本号 | 作者 | 操作 | 日期 | 说明 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1 两个重要线程 4](#_Toc405383197)

[2 数据加载线程*sWorkerThread*与UI主线程的消息处理 4](#_Toc405383198)

[3 初始化加载 6](#_Toc405383199)

[4 初始化显示 6](#_Toc405383200)

### 1 ProGuard概念

ProGuard是一款免费的Java类文件压缩器、优化器和混淆器。它能发现并删除无用类、字段（field）、方法和属性值（attribute）。它也能优化字节码并删除无用的指令。最后，它使用简单无意义的名字来重命名你的类名、字段名和方法名。经过以上操作的jar文件会变得更小，并很难进行逆向工程。这里提到了ProGuard的主要功能是压缩、优化和混淆，下面我就先介绍一下这些概念，然后再介绍ProGuard的基本使用方法。

### 2 什么是压缩：

Java源代码（.java文件）通常被编译为字节码（.class文件）。而完整的程序或程序库通常被压缩和发布成Java文档（.jar文件）。字节码比Java源文件更简洁，但是它仍然包含大量的无用代码，尤其它是一个程序库的时候。ProGuard的压缩程序操作能分析字节码，并删除无用的类、字段和方法。程序只保留功能上的等价，包括异常堆栈描述所需要的信息。

### 3 什么是混淆：

通常情况下，编译后的字节码仍然包含了大量的调试信息：源文件名，行号，字段名，方法名，参数名，变量名等等。这些信息使得它很容易被反编译和通过逆向工程获得完整的程序。有时，这是令人厌恶的。例如像ProGuard这样的混淆器就能删除这些调试信息，并用无意义的字符序列来替换所有名字，使得它很难进行逆向工程，它进一步免费的精简代码。除了异常堆栈信息所需要的类名，方法名和行号外，程序只会保留功能上的等价。通过以上的了解，你应该明白为什么需要混淆了。

### 4 ProGuard的作用：

1.创建紧凑的代码文档是为了更快的网络传输,快速装载和更小的内存占用.

2.创建的程序和程序库很难使用反向工程.

3.所以它能删除来自源文件中的没有调用的代码

4.充分利用java6的快速加载的优点来提前检测和返回java6中存在的类文件.

### 5 ProGuard文件的配置属性：

参数：

-include {filename} 从给定的文件中读取配置参数

-basedirectory {directoryname} 指定基础目录为以后相对的档案名称

-injars {class\_path} 指定要处理的应用程序jar,war,ear和目录

-outjars {class\_path} 指定处理完后要输出的jar,war,ear和目录的名称

-libraryjars {classpath} 指定要处理的应用程序jar,war,ear和目录所需要的程序库文件

-dontskipnonpubliclibraryclasses 指定不去忽略非公共的库类。

-dontskipnonpubliclibraryclassmembers 指定不去忽略包可见的库类的成员。

保留选项

-keep {Modifier} {class\_specification} 保护指定的类文件和类的成员

-keepclassmembers {modifier} {class\_specification} 保护指定类的成员，如果此类受到保护他们会保护的更好

-keepclasseswithmembers {class\_specification} 保护指定的类和类的成员，但条件是所有指定的类和类成员是要存在。

-keepnames {class\_specification} 保护指定的类和类的成员的名称（如果他们不会压缩步骤中删除）

-keepclassmembernames {class\_specification} 保护指定的类的成员的名称（如果他们不会压缩步骤中删除）

-keepclasseswithmembernames {class\_specification} 保护指定的类和类的成员的名称，如果所有指定的类成员出席（在压缩步骤之后）

-printseeds {filename} 列出类和类的成员-keep选项的清单，标准输出到给定的文件

压缩

-dontshrink 不压缩输入的类文件

-printusage {filename}

-whyareyoukeeping {class\_specification}

优化

-dontoptimize 不优化输入的类文件

-assumenosideeffects {class\_specification} 优化时假设指定的方法，没有任何副作用

-allowaccessmodification 优化时允许访问并修改有修饰符的类和类的成员

混淆

-dontobfuscate 不混淆输入的类文件

-printmapping {filename}

-applymapping {filename} 重用映射增加混淆

-obfuscationdictionary {filename} 使用给定文件中的关键字作为要混淆方法的名称

-overloadaggressively 混淆时应用侵入式重载

-useuniqueclassmembernames 确定统一的混淆类的成员名称来增加混淆

-flattenpackagehierarchy {package\_name} 重新包装所有重命名的包并放在给定的单一包中

-repackageclass {package\_name} 重新包装所有重命名的类文件中放在给定的单一包中

-dontusemixedcaseclassnames 混淆时不会产生形形色色的类名

-keepattributes {attribute\_name,...} 保护给定的可选属性，例如LineNumberTable, LocalVariableTable, SourceFile, Deprecated, Synthetic, Signature, and InnerClasses.

-renamesourcefileattribute {string} 设置源文件中给定的字符串常量

### 6 例子：

### A complete Android application

These options shrink, optimize, and obfuscate all public activities, services, broadcast receivers, and content providers from the compiled classes and external libraries:

-injars bin/classes

-injars libs

-outjars bin/classes-processed.jar

-libraryjars /usr/local/java/android-sdk/platforms/android-9/android.jar

-dontpreverify

-repackageclasses ''

-allowaccessmodification

-optimizations !code/simplification/arithmetic

-keepattributes \*Annotation\*

-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.view.View {

public <init>(android.content.Context);

public <init>(android.content.Context, android.util.AttributeSet);

public <init>(android.content.Context, android.util.AttributeSet, int);

public void set\*(...);

}

-keepclasseswithmembers class \* {

public <init>(android.content.Context, android.util.AttributeSet);

}

-keepclasseswithmembers class \* {

public <init>(android.content.Context, android.util.AttributeSet, int);

}

-keepclassmembers class \* implements android.os.Parcelable {

static android.os.Parcelable$Creator CREATOR;

}

-keepclassmembers class \*\*.R$\* {

public static <fields>;

}

Most importantly, we're keeping all fundamental classes that may be referenced by the AndroidManifest.xml file of the application. If your manifest file contains other classes and methods, you may have to specify those as well.

We're keeping annotations, since they might be used by custom RemoteViews.

We're keeping any custom View extensions and other classes with typical constructors, since they might be referenced from XML layout files.

We're also keeping the required static fields in Parcelable implementations, since they are accessed by introspection.

Finally, we're keeping the static fields of referenced inner classes of auto-generated R classes, just in case your code is accessing those fields by introspection. Note that the compiler already inlines primitive fields, so ProGuard can generally remove all these classes entirely anyway (because the classes are not referenced and therefore not required).

If you're using additional Google APIs, you'll have to specify those as well, for instance:

-libraryjars /usr/local/android-sdk/add-ons/google\_apis-7\_r01/libs/maps.jar

If you're using Google's optional License Verification Library, you can obfuscate its code along with your own code. You do have to preserve its ILicensingService interface for the library to work:

-keep public interface com.android.vending.licensing.ILicensingService

If you're using the Android Compatibility library, you should add the following line, to let ProGuard know it's ok that the library references some classes that are not available in all versions of the API:

-dontwarn android.support.\*\*

If applicable, you should add options for processing [native methods](https://stuff.mit.edu/afs/sipb/project/android/sdk/android-sdk-linux/tools/proguard/docs/manual/examples.html#native), [callback methods](https://stuff.mit.edu/afs/sipb/project/android/sdk/android-sdk-linux/tools/proguard/docs/manual/examples.html#callback), [enumerations](https://stuff.mit.edu/afs/sipb/project/android/sdk/android-sdk-linux/tools/proguard/docs/manual/examples.html#enumerations), and [resource files](https://stuff.mit.edu/afs/sipb/project/android/sdk/android-sdk-linux/tools/proguard/docs/manual/examples.html#resourcefiles). You may also want to add options for producing[useful stack traces](https://stuff.mit.edu/afs/sipb/project/android/sdk/android-sdk-linux/tools/proguard/docs/manual/examples.html#stacktrace). You can find a complete sample configuration in examples/android.pro in the ProGuard distribution.

The build process of the Android SDK (version 2.3 and higher) already integrates ProGuard by default. You only need to enable it (for release builds) by addingproguard.config=proguard.cfg to the file build.properties. In case of problems, you may want to check if the automatically generated file proguard.cfg contains the settings discussed above. The generated Ant build file already sets the input and output files for you.

For more information, you can consult the official [Developer Guide](http://developer.android.com/guide/developing/tools/proguard.html) in the Android SDK.