混淆

五步搞定

1

打开项目中的proguard-rules.pro，将下面的代码复制进去。我的思路是将混淆分为2个主要部分

定制化区域。这里边的内容是我们主要需要补充的部分，大致分为4个小部分，我们接下来的步骤主要是补充这4个部分的东西。

基本不用动区域。顾名思义，基本不用动。这块的内容我已经写好了，除非有特殊的需求，基本不用改动。。。请叫我雷锋。。。。。

#-------------------------------------------定制化区域----------------------------------------------

#---------------------------------1.实体类---------------------------------

#-------------------------------------------------------------------------

#---------------------------------2.第三方包-------------------------------

#-------------------------------------------------------------------------

#---------------------------------3.与js互相调用的类------------------------

#-------------------------------------------------------------------------

#---------------------------------4.反射相关的类和方法-----------------------

#----------------------------------------------------------------------------

#---------------------------------------------------------------------------------------------------

#-------------------------------------------基本不用动区域--------------------------------------------

#---------------------------------基本指令区----------------------------------

-optimizationpasses 5

-dontusemixedcaseclassnames

-dontskipnonpubliclibraryclasses

-dontskipnonpubliclibraryclassmembers

-dontpreverify

-verbose

-printmapping proguardMapping.txt

-optimizations !code/simplification/cast,!field/\*,!class/merging/\*

-keepattributes \*Annotation\*,InnerClasses

-keepattributes Signature

-keepattributes SourceFile,LineNumberTable

#----------------------------------------------------------------------------

#---------------------------------默认保留区---------------------------------

-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

-keep class android.support.\*\* {\*;}

-keepclasseswithmembernames class \* {

native <methods>;

}

-keepclassmembers class \* extends android.app.Activity{

public void \*(android.view.View);

}

-keepclassmembers enum \* {

public static \*\*[] values();

public static \*\* valueOf(java.lang.String);

}

-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);

}

-keepclasseswithmembers class \* {

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

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

}

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

public static final android.os.Parcelable$Creator \*;

}

-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();

}

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

\*;

}

-keepclassmembers class \* {

void \*(\*\*On\*Event);

}

#----------------------------------------------------------------------------

#---------------------------------webview------------------------------------

-keepclassmembers class fqcn.of.javascript.interface.for.Webview {

public \*;

}

-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);

}

#----------------------------------------------------------------------------

#---------------------------------------------------------------------------------------------------

2

理解完战略级思想后，我们开始第一部分补充-实体类。实体类由于涉及到与服务端的交互，各种gson的交互如此等等，是要保留的。将你项目中实体类都拎出来，用以下语法进行保留。

-keep class 你的实体类所在的包.\*\* { \*; }

如我的项目下类User的完整路径为：com.demo.login.bean.User, 那我的混淆如下

#---------------------------------1.实体类---------------------------------

-keep class com.demo.login.bean.\*\* { \*; }

#-------------------------------------------------------------------------

当然你的实体类肯定不止这一个，请用上边的方式一一添加，如果你的实体类都在一个包下，那你就幸福了。

3

第2部分是第三方包。打开你的build.gradle文件，查看你用了哪些第三方的包。

dependencies {

compile 'com.github.bumptech.glide:glide:3.7.0'

compile 'org.greenrobot:eventbus:3.0.0'

}

我这里用了glide，eventbus。我去他们的官网把已经写好的混淆copy下来。

#---------------------------------2.第三方包-------------------------------

#eventBus

-keepattributes \*Annotation\*

-keepclassmembers class \*\* {

@org.greenrobot.eventbus.Subscribe <methods>;

}

-keep enum org.greenrobot.eventbus.ThreadMode { \*; }

-keepclassmembers class \* extends org.greenrobot.eventbus.util.ThrowableFailureEvent {

<init>(java.lang.Throwable);

}

#glide

-keep public class \* implements com.bumptech.glide.module.GlideModule

-keep public enum com.bumptech.glide.load.resource.bitmap.ImageHeaderParser$\*\* {

\*\*[] $VALUES;

public \*;

}

#-------------------------------------------------------------------------

一般官网都是有混淆的，没有的话就google，也没有的话自己按照上面的写法自己写，还不会的话。。。。。只能换个包。。。。。如果你是直接包含的jar包的话，你这样写

#---------------------------------2.第三方包-------------------------------

#log4j

-libraryjars log4j-1.2.17.jar

-dontwarn org.apache.log4j.\*\*

-keep class org.apache.log4j.\*\* { \*;}

#-------------------------------------------------------------------------

大致意思就是不混淆，不报warn。如果gradle报错的话，可以考虑注释掉-libraryjars log4j-1.2.17.jar这句。

4

第三部分与js互调的类，工程中没有直接跳过。一般你可以这样写

-keep class 你的类所在的包.\*\* { \*; }

如果是内部类的话，你可以这样

-keepclasseswithmembers class 你的类所在的包.父类$子类 { <methods>; }

例如

-keepclasseswithmembers class com.demo.login.bean.ui.MainActivity$JSInterface {

<methods>;

}

5

第四部分与反射有关的类，工程中没有直接跳过。类的话直接这样

-keep class 你的类所在的包.\*\* { \*; }

熟练后不用五分钟就能搞定。大家可以把自己写完混淆所用的时间写在评论里（0，0）

深入了解

先逐条介绍下基本指令区指令的含义

-optimizationpasses 5

代码混淆的压缩比例，值在0-7之间

-dontusemixedcaseclassnames

混淆后类名都为小写

-dontskipnonpubliclibraryclasses

指定不去忽略非公共的库的类

-dontskipnonpubliclibraryclassmembers

指定不去忽略非公共的库的类的成员

-dontpreverify

不做预校验的操作

-verbose

-printmapping proguardMapping.txt

生成原类名和混淆后的类名的映射文件

-optimizations !code/simplification/cast,!field/\*,!class/merging/\*

指定混淆是采用的算法

-keepattributes \*Annotation\*,InnerClasses

不混淆Annotation

-keepattributes Signature

不混淆泛型

-keepattributes SourceFile,LineNumberTable

抛出异常时保留代码行号

基本指令区基本介绍完了，说下2条用的最多的指令

-keep class XXXX

保留类名不变，也就是类名不混淆，而类中的成员名不保证。当然也可以是继承XXX类的所有类名不混淆，具体代码不贴了，重在理解。

-keepclasseswithmembers class XXXX

保留类名和成员名。当然也可以是类中特定方法，代码不贴了，理由同上。

Gradle：

Gradle 是一种DSL 领域相关语言

project和tasks

每一次构建都是有至少一个project来完成，所以Android studio中的project和Gradle中的project不是一个概念。每个project有至少一个tasks。每一个build.grade文件代表着一个project。tasks在build.gradle中定义。当初始化构建进程，gradle会基于build文件，集合所有的project和tasks,一个tasks包含了一系列动作，然后它们将会按照顺序执行，一个动作就是一段被执行的代码，很像Java中的方法。

构建的生命周期

一旦一个tasks被执行，那么它不会再次执行了，不包含依赖的Tasks总是优先执行，一次构建将会经历下列三个阶段：

初始化阶段：project实例在这儿创建，如果有多个模块，即有多个build.gradle文件，多个project将会被创建。

配置阶段：在该阶段，build.gradle脚本将会执行，为每个project创建和配置所有的tasks。

执行阶段：这一阶段，gradle会决定哪一个tasks会被执行，哪一个tasks会被执行完全依赖开始构建时传入的参数和当前所在的文件夹位置有关。

1.compileSdkVersion：编译该app时候，你想使用到的api版本。

2.buildToolsVersion：构建工具的版本号。

3.defaultConfig方法包含了该app的核心属性，该属性会重写在AndroidManifest.xml中的对应属性。

第一个属性是applicationId，该属性复写了AndroidManifest文件中的包名package

name，但是关于applicationId和packagename有一些不同。在gradle被用来作为Android构建工具之前，packagename在AndroidManifest.xml有两个作用：其作为一个app的唯一标示，并且其被用在了R资源文件的包名。

4.minSdkVersion和targetSdkVersion。这两个和AndroidManifest中的<uses-sdk>很像。minSdkVersion定义为最小支持api。

versionCode将会作为版本号标示，而versionName毫无作用。

5.buildTypes方法定义了如何构建不同版本的app

如果你想知道你多少tasks可以用，直接运行gradlew tasks，其会为你展示所有可用的tasks。当你创建了一个Android工程，那么将包含Android tasks，build tasks，build setup tasks，help tasks，install tasks，verification tasks等。

基本的tasks

android插件依赖于Java插件，而Java插件依赖于base插件。

base插件有基本的tasks生命周期和一些通用的属性。

base插件定义了例如assemble和clean任务，Java插件定义了check和build任务，这两个任务不在base插件中定义。

这些tasks的约定含义：

* assemble: 集合所有的output
* clean: 清除所有的output
* check: 执行所有的checks检查，通常是unit测试和instrumentation测试
* build: 执行所有的assemble和check

Java插件同时也添加了source sets的概念。

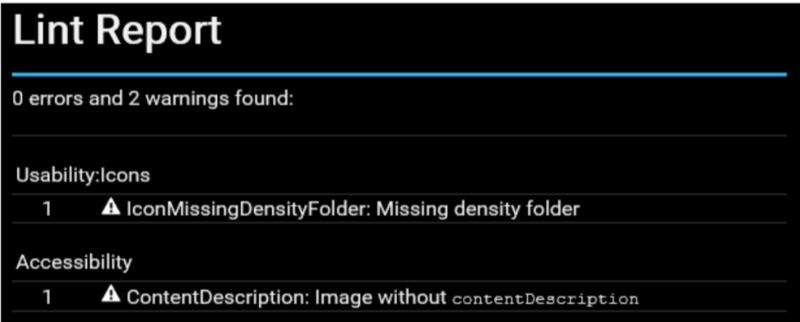
Android tasks

android插件继承了这些基本tasks,并且实现了他们自己的行为：

* assemble 针对每个版本创建一个apk
* clean 删除所有的构建任务，包含apk文件
* check 执行Lint检查并且能够在Lint检测到错误后停止执行脚本
* build 执行assemble和check

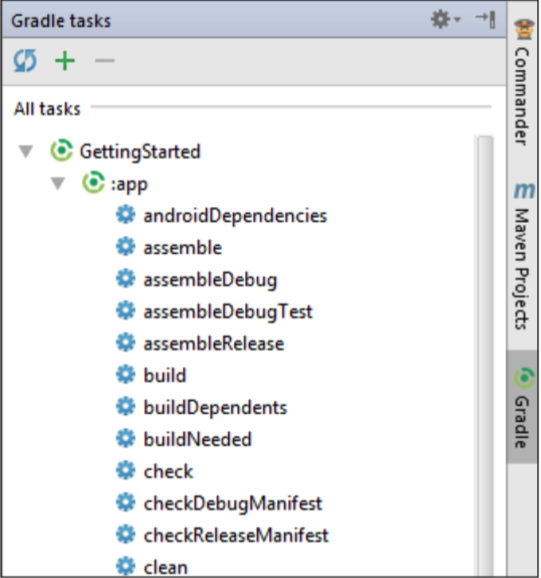
默认情况下assemble tasks定义了assembleDebug和assembleRelease，当然你还可以定义更多构建版本。除了这些tasks,android 插件也提供了一些新的tasks:

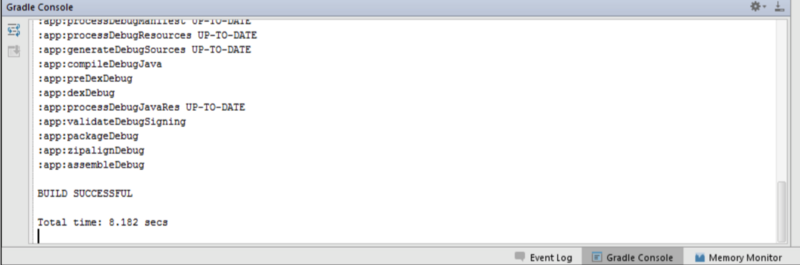
* connectedCheck 在测试机上执行所有测试任务
* deviceCheck 执行所有的测试在远程设备上
* installDebug和installRelease 在设备上安装一个特殊的版本
* 所有的install task对应有uninstall 任务

build task依赖于check任务，但是不依赖于connectedCheck或者deviceCheck，执行check任务的使用Lint会产生一些相关文件，这些报告可以在app/build/outputs中查看：

android studio的tasks

你根本不必要去执行gradle脚本在命令行中，Android studio有其对应的工具：



在这个界面，你要做的就是双击了。当然你也可以在Android studio中打开命令行，执行相关命令，具体操作就不介绍了。  


自定义构建

依赖管理：的build文件，Gradle会自动从远程仓库为你下载相关的jar包，并且保证你能够正确使用它们。Gradle甚至可以为你做的更多，包括当你在你的工程里添加了多个相同的依赖，gradle会为你排除掉相同的jar包

仓库

Gradle支持三种不同的仓库，分别是：Maven和Ivy以及文件夹

Gradle会默认预定义三个maven仓库：Jcenter和mavenCentral以及本地maven仓库。

repositories {

mavenCentral()

jcenter()

mavenLocal()

}

远程仓库

有些组织，创建了一些有意思的插件或者library,他们更愿意把这些放在自己的maven库，而不是maven中心库或jcenter。那么当你需要是要这些仓库的时候，你只需要在maven方法中加入url地址就好：

repositories {

maven {

url "http://repo.acmecorp.com/maven2"

}

}

本地依赖

默认情况下，新建的Android项目会有一个lib文件夹，并且会在依赖中这么定义（即添加所有在libs文件夹中的jar）：

dependencies {

compile fileTree(dir: 'libs', include: ['\*.jar'])

}

native包（so包）

用c或者c++写的library会被叫做so包，Android插件默认情况下支持native包，你需要把.so文件放在对应的文件夹中：

app

├── AndroidManifest.xml

└── jniLibs

├── armeabi

│ └── nativelib.so

├── armeabi-v7a

│ └── nativelib.so

├── mips

│ └── nativelib.so

└── x86

└── nativelib.so

使用aar文件

如果你想复用你的library，那么你就可以创建一个aar文件，并将其作为你的工程依赖。当你构建你的library项目，aar文件将会在 build/output/aar/下生成。把该文件作为你的依赖包，你需要创建一个文件夹来放置它，我们就叫它aars文件夹吧，然后把它拷贝到该文件夹里面，然后添加该文件夹作为依赖库：

repositories {

flatDir {

dirs 'aars'

}

}

这样你就可以把该文件夹下的所有aar文件作为依赖，同时你可以这么干：

dependencies {

compile(name:'libraryname', ext:'aar')

}

依赖的概念

为了能顺利编译你的代码，你需要添加SDK到你的编译环境。你不需要将sdk包含在你的APK中，因为它早已经存在于设备中，所以配置来啦，我们会有5个不同的配置：

compile

apk

provided

testCompile

androidTestCompile

compile是默认的那个，其含义是包含所有的依赖包，即在APK里，compile的依赖会存在。

apk的意思是apk中存在，但是不会加入编译中，这个貌似用的比较少。

provided的意思是提供编译支持，但是不会写入apk。

testCompile和androidTestCompile会添加额外的library支持针对测试。

构建版本

在Gradle的Android插件中，一个构建版本意味着定义一个app或者依赖库如何被构建。每个构建版本都要特殊的一面，比如是否需要debug，application id是什么，是否不需要的资源被删除等等。你可以定义一个构建版本通过buildTypes方法。例如：

android {

buildTypes {

release {

minifyEnabled false

proguardFiles getDefaultProguardFile

('proguard-android.txt'), 'proguard-rules.pro'

}

}

}

这个文件定义了该模块是release版本，然后定义了proguard的位置。该release版本不是唯一的构建版本，默认情况下，还有个debug版本。Android studio把它视为默认构建版本。

创建自己的构建版本

当默认的构建版本不够用的时候，创建版本也是很容易的一件事，创建构建版本你只需要在buildTypes写入自己的版本。如下所示：

android {

buildTypes {

staging {

applicationIdSuffix ".staging"

versionNameSuffix "-staging"

buildConfigField "String", "API\_URL",

"\"http://staging.example.com/api\""

}

}

}

我们定义了一个staging版本，该版本定义了一个新的application id，这让其与debug和release版本的applicationID不同。假设你使用了默认的配置，那么applicationID将会是这样的：

Debug: com.package

Release: com.package

Staging: com.package.staging

product flavors

和构建版本不同，product flavors用来为一个app创建不同版本。典型的例子是，一个app有付费和免费版。product flavors极大简化了基于相同的代码构建不同版本的app。

android {

productFlavors {

red {

applicationId 'com.gradleforandroid.red'

versionCode 3

}

blue {

applicationId 'com.gradleforandroid.blue'

minSdkVersion 14

versionCode 4

}

}

}

product flavors和构建版本的配置不同。因为product flavors有自己的ProductFlavor类，就像defaultConfig，这意味着你的所有productFlavors都分享一样的属性。

多个flavors构建变体

在一些例子中，你可能需要创建一些product flavors的合并版本。举个例子，client A和client B可能都想要一个free和paid的版本，而他们又都是基于一样的代码，但是有不一样的颜色等。创建四个不同的flavors意味着有重复的配置。合并flavors最简单的做法可能是使用flavor dimensions，就像这样：

android {

flavorDimensions "color", "price"

productFlavors {

red {

flavorDimension "color"

}

blue {

flavorDimension "color"

}

free {

flavorDimension "price"

}

paid {

flavorDimension "price"

}

}

}

签名配置

在你发布你的应用之前，你需要为你的app私钥签名。如果你有付费版和免费版，你需要有不同的key去签名不同的变体。这就是配置签名的好处。配置签名可以这样定义：

android {

signingConfigs {

staging.initWith(signingConfigs.debug)

release {

storeFile file("release.keystore")

storePassword"secretpassword"

keyAlias "gradleforandroid"

keyPassword "secretpassword"

}

}

}