







# 反编译

## apk是什么

**\*APK\*** (全称: Android application package, **Android应用程序包**) 是Android操作系统使用的一种应用程序包文件格式, 用于分发和安装移动应用。一个Android应用程序的代码想要在Android设备上运行, 必须先进行编译, 然后被打包成为一个被Android系统所能识别的文件才可以被运行, 而这种能被Android系统识别并运行的文件格式便是“APK”。一个APK文件内包含被编译的代码文件(.dex 文件), 文件资源 (resources), 原生资源文件 (assets), 证书 (certificates), 和清单文件 (manifest file)

虽然Android应用程序包的文件格式是.apk, 但是它其实也是一个压缩包。直接打开里面就会有6个文件

 kotlin	2020/7/15 3:26	文件夹	
 META-INF	2020/7/15 3:26	文件夹	
 res	2020/7/15 3:26	文件夹	
 AndroidManifest.xml	2020/7/15 2:30	XML 文档	3 KB
 classes.dex	2020/7/15 2:30	DEX 文件	3,523 KB
 resources.arsc	2020/7/15 2:30	ARSC 文件	264 KB

- **kotlin**: java的apk里面只有另外的5个文件, 但是我这个APK是用kotlin写的所以多了一个, 我猜这个应该只是kotlin里面的一些库, 感觉应该没有多大的影响吧。
- **META-INF文件夹**: 这里面储存的是关于apk的签名信息, 在安装apk时, 系统会校验apk的签名信息, 判断程序完整性(所以如果直接解压apk修改文件, 再重新打包的话, 是肯定安装不上的) (如果真能这样, 那白嫖也太简单了)。
- **res文件夹**: 做Android的肯定知道, 这里面存储着apk用到的资源文件, 但这样就想拿到里面的布局文件? 那是肯定不行的, 为了减小文件大小以及保证一定程度的安全, 里面的所有xml文件都是二进制的, 并不能直接拿到 (不过拿个图片还是可以的)。
- **AndroidManifest.xml**: 这东西大家肯定非常熟悉, 是一个存了一大堆程序配置信息的清单文件, 当然也是二进制的。
- **class.dex**(有时候不止一个dex文件): DEX文件 (DalvikVM executes), 顾名思义, 是Android Dalvik虚拟机上的执行程序, 也就是Dalvik字节码, 程序的代码都在里面(反编译看源码的关键)。
- **resources.arsc**: 编译后的二进制资源文件。

## 什么是反编译

要了解什么是反编译就要了解什么是编译, 要了解编译就要知道计算机语言。

目前计算机语言大概分为两种, 一种低级语言, 一种高级语言。可以这样简单的理解: 低级语言是计算机认识的语言、高级语言是程序员认识的语言。

那么, 怎么把程序员写出来的高级语言转换成计算机认识的低级语言然后让计算机执行呢?

这个过程其实就是编译!

编译的主要的目的是将便于人编写、阅读、维护的高级语言所写作的源代码程序, 翻译为计算机能解、运行的低级语言的程序, 也就是可执行文件。

那么反编译就是将这个过程反过来通过低级语言进行反向工程，获取其源代码。这个过程，就叫做反编译。

## 为什么要反编译

那肯定是为了白嫖学习别人的开发思想。那些破解软件基本上都是用了反编译，然后重新签名打包发布出来的。(不建议这样做)

## 开始反编译

对反编译有所了解了，我们就可以开始反编译了。

当然工欲善其事，必先利其器。所以你需要安装下面的软件

- *apktool*: 反编译apk的利器，还可重新打包。
- *dex2jar*: 把dex文件反编译成jar文件，获取源码。
- *jd-gui*: 查看jar源码，当然用其他工具也能看。

需要注意一下，将下载的apktool\_2.4.1.jar文件重命名为apktool.jar

就是需要自己新建一个文本文件输入下面的文字，然后重命名为apktool.bat

```
@echo off
if "%PATH_BASE%" == "" set PATH_BASE=%PATH%
set PATH=%CD%;%PATH_BASE%;
chcp 65001 2>nul >nul
java -jar -Duser.language=en -Dfile.encoding=UTF8 "%~dp0\apktool.jar" %*
```

这个直接在网络上下载就行，资源还是挺多的。

准备好之后就可以开始行动了。

这里为了尊重他人的劳动成果，我就自己写了一个非常简单的app，当作示例

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello world!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/change"
        android:layout_width="wrap_content"
```

```

        android:layout_height="wrap_content"
        android:text="change"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>

```

```

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        change.setOnClickListener {
            text.setText("Hello Android!")
        }
    }
}

```

就是单击按钮然后会把TextView里面的文字换为“Hello Android! ”







#### • apktool的使用

将上述两个文件apktool.bat和apktool.jar文件放到同一文件夹下（任意路径），打开命令窗口（win+R-->cmd-->enter）；  
定位到apktool所在的文件夹；  
输入以下命令：

```
apktool.bat d -f ***.apk objectFolderPath
```

其中，objectFolderPath为可选项，如果此项不存在，软件将会在apktool文件夹下新建一个apk文件名的文件夹，否则存储到目标文件夹；

即可在该处产生一个同名的demo文件夹，反编译的结果就在里面。

	kotlin	2020/7/15 3:16	文件夹	
	original	2020/7/15 3:16	文件夹	
	res	2020/7/15 3:16	文件夹	
	smali	2020/7/15 3:16	文件夹	
	AndroidManifest.xml	2020/7/15 3:16	XML 文档	1 KB
	apktool.yml	2020/7/15 3:16	YML 文件	2 KB

然后 res下的所有xml文件以及AndroidManifest都是可读的了

smali 文件夹：里面存放着源代码反编译出的smali文件

通过文件名，都能大概看出这些smali文件来自哪些Java类了（\$表示内部类）

用文本编辑器打开这些文件看看，应该还是能大致看出具体什么意思的，但是肯定没Java看着爽，所以我们进入下一步。

Smali, Baksmali分别是指安卓系统里的Java虚拟机（Dalvik）所使用的一种.dex格式文件的汇编器，反汇编器。

其语法是一种宽松式的Jasmin/dedexer语法，而且它实现了.dex格式所有功能（注解，调试信息，线路信息等）。（百度百科）

- 使用dex2jar

把最开始解压出来的文件里面的class.dex文件放到dex2jar文件里面在控制台输入

```
.\d2j-dex2jar.bat classes.dex
```

会得到如下结果

名称	修改日期	类型	大小
lib	2014/10/27 17:32	文件夹	
classes.dex	2020/7/15 2:30	DEX 文件	3,523 KB
classes-dex2jar.jar	2020/7/15 3:29	JAR 文件	3,001 KB
d2j_invoke.bat	2014/10/27 17:32	Windows 批处理文件	1 KB
d2j_invoke.sh	2014/10/27 17:32	SH 文件	2 KB
d2j-baksmali.bat	2014/10/27 17:32	Windows 批处理文件	1 KB
d2j-baksmali.sh	2014/10/27 17:32	SH 文件	2 KB
d2j-dex2jar.bat	2014/10/27 17:32	Windows 批处理文件	1 KB
d2j-dex2jar.sh	2014/10/27 17:32	SH 文件	2 KB
d2j-dex2smali.bat	2014/10/27 17:32	Windows 批处理文件	1 KB

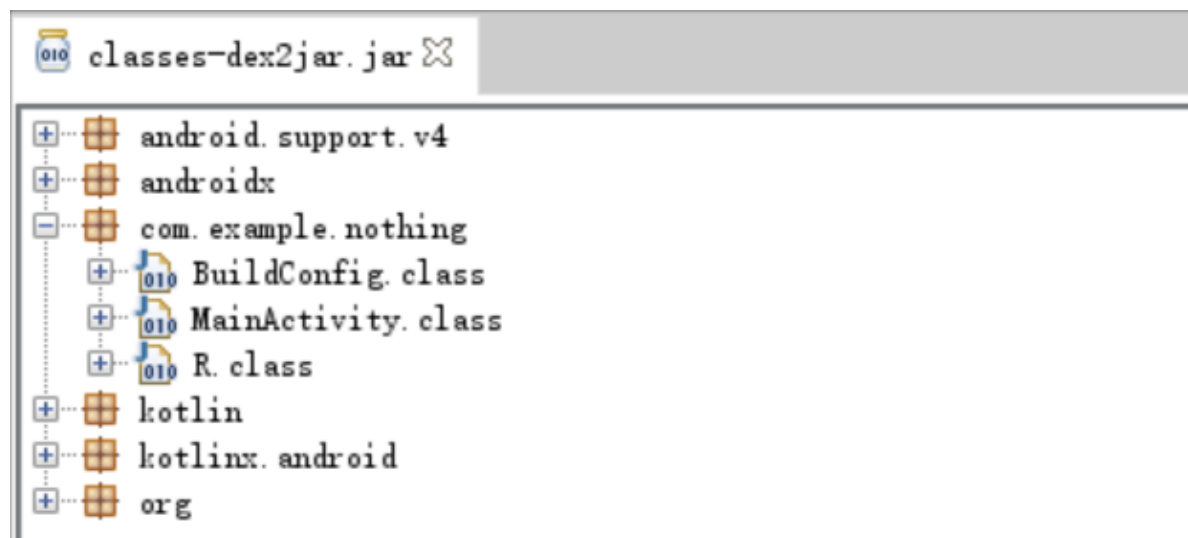
把得到的jar包用jd-gui打开就行

- 通过jd-gui查看java源码

在jd-gui的目录下，输入

```
java -jar jd-gui.exe
```

我不知道为什么我直接点会打不开，只有这样能打开



- 修改程序，二次打包

新建一个activity

```
class AnotherActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_another)  
    }  
}
```

```

<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:background="#2830CA">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="破解成功"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>

```

- 将java文件转化为smali

有三种方法

1. AS或者idea有个叫做ava2smali的插件,装上后直接Build=>Compile to smali搞定,但由于用了androidx库,这个插件似乎不能成功运行。遂选择第二种方法。
2. 建立一个新项目,写好java文件和layout文件,直接打包apk,然后再反编译(我反编译我自己),也能得到smali文件。如果java文件里有内部类(比如按键监听器),会生成多个smali文件。
3. 直接手写smali (不会,跳过)

得到的java转为smali文件如下

```

.class public final Lcom/example/nothing/AnotherActivity;
.super Landroidx/appcompat/app/AppCompatActivity;
.source "AnotherActivity.kt"

# annotations
.annotation runtime Lkotlin/Metadata;
    bv = {
        0x1,
        0x0,
        0x3
    }
    d1 = {

        "\u0000\u0018\n\u0002\u0018\u0002\n\u0002\u0018\u0002\n\u0002\u0018\u0002\n\u0002\u0008\u0002\n\u0002\u0010\u0002\n\u0000\u0000\u0018\u0002\n\u0000\u0002\u0018\u0002\n\u0000\u0000\u0018\u0002\u0002\u0002\u001B\u0005\u00a2\u0006\u0002\u0010\u0002\u0012\u0012\u0010\u0003\u001a\u0002\u0004\u0008\u0008\u0010\u0005\u001a\u0004\u0018\u0001\u0006\u0014\u00a8\u0006\u0007"

    }
    d2 = {
        "Lcom/example/nothing/AnotherActivity;",
        "Landroidx/appcompat/app/AppCompatActivity;",
        "()V",
        "onCreate",
        "",
    }

```

```

        "savedInstanceState",
        "Landroid/os/Bundle;",
        "app_release"
    }
    k = 0x1
    mv = {
        0x1,
        0x1,
        0x10
    }
.end annotation

# instance fields
.field private _$_findViewCache:Ljava/util/HashMap;

# direct methods
.method public constructor <init>()V
    .locals 0

    .line 11
    invoke-direct {p0}, Landroidx/appcompat/app/AppCompatActivity; -> <init>()V

    return-void
.end method

# virtual methods
.method public _$_clearFindViewByIdCache()V
    .locals 1

    iget-object v0, p0, Lcom/example/nothing/AnotherActivity; -
    > _$_findViewCache:Ljava/util/HashMap;

    if-eqz v0, :cond_0

    invoke-virtual {v0}, Ljava/util/HashMap; -> clear()V

    :cond_0
    return-void
.end method

.method public _$_findCachedViewById(I)Landroid/view/View;
    .locals 2

    iget-object v0, p0, Lcom/example/nothing/AnotherActivity; -
    > _$_findViewCache:Ljava/util/HashMap;

    if-nez v0, :cond_0

    new-instance v0, Ljava/util/HashMap;

    invoke-direct {v0}, Ljava/util/HashMap; -> <init>()V

    iput-object v0, p0, Lcom/example/nothing/AnotherActivity; -
    > _$_findViewCache:Ljava/util/HashMap;

```

```

        :cond_0
        iget-object v0, p0, Lcom/example/nothing/AnotherActivity;-
>_$_findViewCache:Ljava/util/HashMap;

        invoke-static {p1}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;

        move-result-object v1

        invoke-virtual {v0, v1}, Ljava/util/HashMap;-
>get(Ljava/lang/Object;)Ljava/lang/Object;

        move-result-object v0

        check-cast v0, Landroid/view/View;

        if-nez v0, :cond_1

        invoke-virtual {p0, p1}, Landroidx/fragment/app/FragmentActivity;-
>findViewById(I)Landroid/view/View;

        move-result-object v0

        iget-object v1, p0, Lcom/example/nothing/AnotherActivity;-
>_$_findViewCache:Ljava/util/HashMap;

        invoke-static {p1}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;

        move-result-object p1

        invoke-virtual {v1, p1, v0}, Ljava/util/HashMap;-
>put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;

        :cond_1
        return-object v0
    .end method

    .method protected onCreate(Landroid/os/Bundle;)V
        .locals 0

        .line 14
        invoke-super {p0, p1}, Landroidx/appcompat/app/AppCompatActivity;-
>onCreate(Landroid/os/Bundle;)V

        const p1, 0x7f0a001c

        .line 15
        invoke-virtual {p0, p1}, Lcom/example/nothing/AnotherActivity;-
>setContentView(I)V

        return-void
    .end method

```

把得到的smali文件放入smali文件夹里面，如果有内部类就会有多个smali文件

名称	修改日期	类型	大小
AnotherActivity.smali	2020/7/15 18:47	SMALI 文件	4 KB
BuildConfig.smali	2020/7/15 18:47	SMALI 文件	1 KB
MainActivity\$onCreate\$1.smali	2020/7/15 18:47	SMALI 文件	3 KB
MainActivity.smali	2020/7/15 18:47	SMALI 文件	4 KB
R\$anim.smali	2020/7/15 18:47	SMALI 文件	3 KB
R\$attr.smali	2020/7/15 18:47	SMALI 文件	22 KB
R\$bool.smali	2020/7/15 18:47	SMALI 文件	1 KB

在manifest添加启动新添加的activity

AndroidManifest.xml - 记事本

文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest
xmlns:android="http://schemas.android.com/apk/res/android"
android:compileSdkVersion="29" android:compileSdkVersionCodename="10"
package="com.example.nothing" platformBuildVersionCode="29"
platformBuildVersionName="10">
  <application android:allowBackup="true"
android:appComponentFactory="androidx.core.app.CoreComponentFactory"
android:icon="@mipmap/ic_launcher" android:label="@string/app_name"
android:roundIcon="@mipmap/ic_launcher_round" android:supportsRtl="true"
android:theme="@style/AppTheme">
    <activity android:name="com.example.nothing.MainActivity"/>
    <activity android:name="com.example.nothing.AnotherActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>
</manifest>
```

第 1 行, 第 1 列100%Windows (CRLF)UTF-8

修改smali\com\example\nothing下面的R&layout.smali文件



```
R$layout.smali - 记事本
文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)

.field public static final select_dialog_item_material:I = 0x7f0a0024

.field public static final select_dialog_multichoice_material:I = 0x7f0a0025

.field public static final select_dialog_singlechoice_material:I = 0x7f0a0026

.field public static final support_simple_spinner_dropdown_item:I = 0x7f0a0027

.field public static final activity_another:I = 0x7f0a0028

# direct methods
.method private constructor <init>()V
    .locals 0

    invoke-direct {p0}, Ljava/lang/Object;.> <init>()V

    return-void
.end method
```

建议在最后添加，然后直接把上面一个的值加一就行。

然后再在AnotherActivity里面修改setContentView()里面的id

```
AnotherActivity.smali - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

    return-object v0
.end method

.method protected onCreate(Landroid/os/Bundle;)V
    .locals 0

    .line 14
    invoke-super {p0, p1}, Landroidx/appcompat/app/AppCompatActivity;.>
    onCreate(Landroid/os/Bundle;)V

    const p1 0x7f0a0028

    .line 15
    invoke-virtual {p0, p1}, Lcom/example/nothing/AnotherActivity;.>
    setContentView(I)V

    return-void
.end method
```

改成上面添加的值。

修改完毕

开始打包

```
.\apktool.bat b app-release -o new.apk
# b表示打包 demo指的是用于打包的文件夹 -o xxx\xxx.apk 表示输出apk的路径以及文件名
# 如果不指定-o参数的话 apk默认生成于app-release\dist目录下
```

打包完成，并不能直接安装，因为还需要签名。

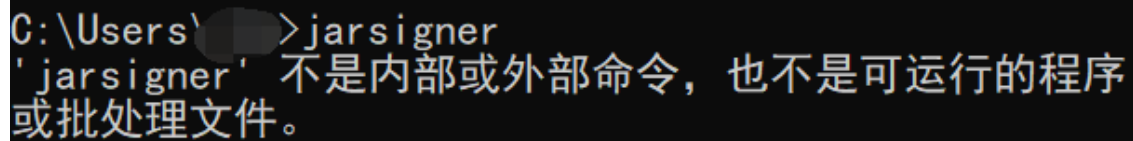
接下来讲一下签名apk

我用的是Java自带的签名工具jarsigner.exe 位于xxx\jdk\bin目录下（当然安装jdk的时候一般都设置了环境变量的，所以应该能直接调用）。

做Android的也肯定知道，生成一个签名的apk需要一个签名证书，xxx.jks,如果没有可以用AS生成一个

然后把jks文件拷贝一份到要签名的apk目录下

这里可能会出现一个问题，就是



这个可能是2个方面的问题，一个就是你jdk里面没有jarsigner.exe这个文件，另外一个就是你的环境变量没有配置好。

```
#可以先看看apk是否被签名
jarsigner -verify new.apk
#签名
jarsigner -digestalg SHA1 -sigalg MD5withRSA -keystore fhh.jks -storepass
xxxxxxxxxx -signedjar signed.apk new.apk fenghaha
#参数含义
jarsigner -digestalg SHA1 -sigalg MD5withRSA -keystore [签名证书文件名] -storepass
[签名证书密码] -signedjar [签名后生成apk的文件] [被签名的apk文件] [签名证书的别名]
```

签名完成后就会在当前目录下生成一个signed.apk，安装到手机上

Nothing

破解成功



可以看到破解成功了。

整个流程下来，虽然看上去很顺利，但实际操作上还是有很多问题的，比如想反编译的apk是被混淆过的，或者被加固过，又或者在native层做了签名校验，反编译的难度就会难很多。但安全攻防，有加固那也有脱壳，也有.so库的反编译，不过作为一个做开发而不是做安全的，也没有必要研究得太深入），通过这次反编译实战，理解一下Android项目的结构，编译过程，发布流程以及一点点的安全防护方面的知识，感觉也足够了

## 混淆

一个APK如果没有混淆，那么就会很容易被反编译代码、反编译资源、以及重新打包。

混淆代码并不是让代码无法被反编译，而是将代码中的类、方法、变量等信息进行重命名，把它们改成一些毫无意义的名字。

打出正式版的APK才会进行混淆，Debug版的APK是不会混淆的。

总结一下，混淆和反编译就类似于攻防战。

## 开始混淆

我还是准备就混淆上面那个例子

```
android {
    compileSdkVersion 29
    buildToolsVersion "29.0.2"

    defaultConfig {
        applicationId "com.example.nothing"
        minSdkVersion 16
        targetSdkVersion 29
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
}
```

```
}
```

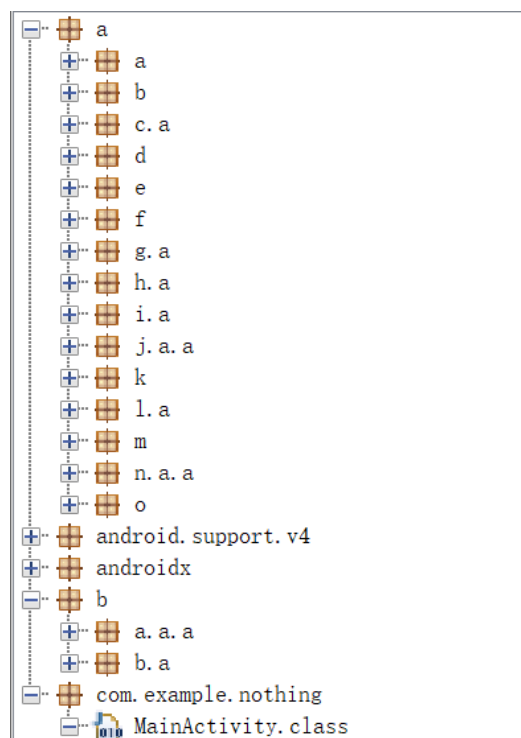
在Android Studio当中混淆APK比较简单，借助SDK中自带的Proguard工具，只需要修改build.gradle中的一行配置即可。可以看到，现在build.gradle中minifyEnabled的值是false，这里我们只需要把值改成true，打出来的APK包就会是混淆过的了。如下所示：

```
buildTypes {  
    release {  
        minifyEnabled false  
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'  
    }  
}
```

其中minifyEnabled用于设置是否启用混淆，proguardFiles用于选定混淆配置文件。注意这里是在release闭包内进行配置的，因此只有打出正式版的APK才会进行混淆，Debug版的APK是不会混淆的。当然这也是非常合理的，因为Debug版的APK文件我们只会用来内部测试，不用担心被人破解。

那么现在来打一个正式版的APK文件，在Android Studio导航栏中点击Build->Generate Signed APK，然后选择签名文件并输入密码，如果没有签名文件就创建一个，最终点击Finish完成打包，生成的APK文件会自动存放在app目录下。

这里同样通过上面的工具对他进行反编译。得到下面的结果



再看MainActivity里面的代码。

```

package com.example.nothing;

import a.b.k.h;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import b.b.a.a;
import java.util.HashMap;

public final class MainActivity extends h {
    public HashMap p;

    public View b(int paramInt) {
        if (this.p == null)
            this.p = new HashMap<Object, Object>();
        View view2 = (View)this.p.get(Integer.valueOf(paramInt));
        View view1 = view2;
        if (view2 == null) {
            view1 = findViewById(paramInt);
            this.p.put(Integer.valueOf(paramInt), view1);
        }
        return view1;
    }

    public void onCreate(Bundle paramBundle) {
        super.onCreate(paramBundle);
        setContentView(2131361821);
        ((Button)b(a.change)).setOnClickListener(new a(this));
    }

    public static final class a implements View.OnClickListener {
        public a(MainActivity param1MainActivity) {}

        public final void onClick(View param1View) { ((TextView)this.b.b(a.text)).setText("Hello Android!"); }
    }
}

```

可以看到，MainActivity的类名是没有混淆的，onCreate()方法也没有被混淆，但是定义的方法、全局变量、局部变量都被混淆了。kotlin里面的库都被混淆了。

这是AS打正式APK时默认的混淆规则，那么这些混淆规则是在哪里定义的呢？其实就是刚才在build.gradle的release闭包下配置的proguard-android.txt文件，这个文件存放于/tools/proguard目录下，我们打开来看一下：

```

# This is a configuration file for ProGuard.
# http://proguard.sourceforge.net/index.html#manual/usage.html
#
# This file is no longer maintained and is not used by new (2.2+) versions of
the
# Android plugin for Gradle. Instead, the Android plugin for Gradle generates
the
# default rules at build time and stores them in the build directory.

-dontusemixedcaseclassnames
-dontskipnonpubliclibraryclasses
-verbose

# Optimization is turned off by default. Dex does not like code run
# through the ProGuard optimize and preverify steps (and performs some
# of these optimizations on its own).
-dontoptimize
-dontpreverify
# Note that if you want to enable optimization, you cannot just
# include optimization flags in your own project configuration file;
# instead you will need to point to the
# "proguard-android-optimize.txt" file instead of this one from your
# project.properties file.

-keepattributes *Annotation*

```

```

-keep public class com.google.vending.licensing.ILicensingService
-keep public class com.android.vending.licensing.ILicensingService

# For native methods, see
http://proguard.sourceforge.net/manual/examples.html#native
-keepclasseswithmembernames class * {
    native <methods>;
}

# keep setters in Views so that animations can still work.
# see http://proguard.sourceforge.net/manual/examples.html#beans
-keepclassmembers public class * extends android.view.View {
    void set*(***);
    *** get*();
}

# We want to keep methods in Activity that could be used in the XML attribute
onClick
-keepclassmembers class * extends android.app.Activity {
    public void *(android.view.View);
}

# For enumeration classes, see
http://proguard.sourceforge.net/manual/examples.html#enumerations
-keepclassmembers enum * {
    public static **[] values();
    public static ** valueOf(java.lang.String);
}

-keepclassmembers class * implements android.os.Parcelable {
    public static final android.os.Parcelable$Creator CREATOR;
}

-keepclassmembers class **.R$* {
    public static <fields>;
}

# The support library contains references to newer platform versions.
# Don't warn about those in case this app is linking against an older
# platform version. We know about them, and they are safe.
-dontwarn android.support.**

# Understand the @Keep support annotation.
-keep class android.support.annotation.Keep

-keep @android.support.annotation.Keep class * {*;}

-keepclasseswithmembers class * {
    @android.support.annotation.Keep <methods>;
}

-keepclasseswithmembers class * {
    @android.support.annotation.Keep <fields>;
}

-keepclasseswithmembers class * {
    @android.support.annotation.Keep <init>(...);
}

```

这个就是默认的混淆配置文件了，我们来一起逐行阅读一下

`-dontusemixedcaseclassnames` 表示混淆时不使用大小写混合类名。

`-dontskipnonpubliclibraryclasses` 表示不跳过library中的非public的类。

`-verbose` 表示打印混淆的详细信息。

`-dontoptimize` 表示不进行优化，建议使用此选项，因为根据proguard-android-optimize.txt中的描述，优化可能会造成一些潜在风险，不能保证在所有版本的Dalvik上都正常运行。

`-dontpreverify` 表示不进行预校验。这个预校验是作用在Java平台上的，Android平台上不需要这项功能，去掉之后还可以加快混淆速度。

`-keepattributes *Annotation*` 表示对注解中的参数进行保留。

```
-keep public class com.google.vending.licensing.ILicensingService
-keep public class com.android.vending.licensing.ILicensingService
```

表示不混淆上述声明的两个类，这两个类我们基本也用不上，是接入Google原生的一些服务时使用的。

```
-keepclasseswithmembernames class * {
    native <methods>;
}
```

表示不混淆任何包含native方法的类的类名以及native方法名。

#### 什么是native方法

简单地讲，一个Native Method就是一个java调用非java代码的接口。一个Native Method是这样一个java的方法：该方法的实现由非java语言实现，比如C。这个特征并非java所特有，很多其它的编程语言都有这一机制，比如在C++中，你可以用extern "C"告知C++编译器去调用一个C的函数。

```
-keepclassmembers public class * extends android.view.View {
    void set*(***);
    *** get*();
}
```

表示不混淆任何一个View中的setXxx()和getXxx()方法，因为属性动画需要有相应的setter和getter的方法实现，混淆了就无法工作了。

```
-keepclassmembers class * extends android.app.Activity {
    public void *(android.view.View);
}
```

表示不混淆Activity中参数是View的方法，因为有这样一种用法，在XML中配置android:onClick="buttonClick"属性，当用户点击该按钮时就会调用Activity中的buttonClick(View view)方法，如果这个方法被混淆的话就找不到了。

```
-keepclassmembers enum * {
    public static **[] values();
    public static ** valueOf(java.lang.String);
}
```



表示不混淆枚举中的values()和valueOf()方法，枚举我用的非常少，这个就不评论了。

```
-keepclassmembers class * implements android.os.Parcelable {
    public static final android.os.Parcelable$Creator CREATOR;
}
```

表示不混淆Parcelable实现类中的CREATOR字段，毫无疑问，CREATOR字段是绝对不能改变的，包括大小写都不能变，不然整个Parcelable工作机制都会失败。

```
-keepclassmembers class **.R$* {
    public static <fields>;
}
```

表示不混淆R文件中的所有静态字段，我们都知道R文件是通过字段来记录每个资源的id的，字段名要是被混淆了，id也就找不着了。

`-dontwarn android.support.**` 表示对android.support包下的代码不警告，因为support包中有很多代码都是在高版本中使用的，如果我们的项目指定的版本比较低在打包时就会给予警告。不过support包中所有的代码都在版本兼容性上做足了判断，因此不用担心代码会出问题，所以直接忽略警告就可以了。

这就是proguard-android.txt文件中所有默认的配置，而我们混淆代码也是按照这些配置的规则来进行混淆的。

proguard语法中还真有几处非常难理解的地方

proguard中一共有三组六个keep关键字，很多人搞不清楚它们的区别，这里通过一个表格来直观地看下：

关键字	描述
keep	保留类和类中的成员，防止它们被混淆或移除。
keepnames	保留类和类中的成员，防止它们被混淆，但当成员没有被引用时会被移除。
keepclassmembers	只保留类中的成员，防止它们被混淆或移除。
keepclassmembernames	只保留类中的成员，防止它们被混淆，但当成员没有被引用时会被移除。
keepclasseswithmembers	保留类和类中的成员，防止它们被混淆或移除，前提是指名的类中的成员必须存在，如果不存在则还是会混淆。
keepclasseswithmembernames	保留类和类中的成员，防止它们被混淆，但当成员没有被引用时会被移除，前提是指名的类中的成员必须存在，如果不存在则还是会混淆。

除此之外，proguard中的通配符也比较让人难懂，proguard-android.txt中就使用到了很多通配符，我们来看一下它们之间的区别：

通配符	描述
	匹配类中的所有字段
	匹配类中的所有方法
	匹配类中的所有构造函数
*	匹配任意长度字符，但不含包名分隔符(.). 比如说我们的完整类名是 com.example.test.MyActivity，使用 com.*，或者 com.exmaple.* 都是无法匹配的，因为 * 无法匹配包名中的分隔符，正确的匹配方式是 com.exmaple.*.*，或者 com.exmaple.test.*，这些都是可以的。但如果你不写任何其它内容，只有一个*，那就表示匹配所有的东西。
**	匹配任意长度字符，并且包含包名分隔符(.). 比如 proguard-android.txt 中使用的 -dontwarn android.support.** 就可以匹配 android.support 包下的所有内容，包括任意长度的子包。
***	匹配任意参数类型。比如 void set*(*) 就能匹配任意传入的参数类型，* get*() 就能匹配任意返回值的类型。
...	匹配任意长度的任意类型参数。比如 void test(...) 就能匹配任意 void test(String a) 或者是 void test(int a, String b) 这些方法。

写在最后

这是我第一次讲课，可能讲得不太好，希望大家多多理解。

这个反编译和混淆郭神都有讲过，比较通俗易懂。