# 第四篇 Gradle的高级使用

## 4.1 基础知识

1，gradle中有一个功能叫做变体「productflavors」，这是来为APP设置不同的打包配置，以实现多渠道打包的一种方案。

##### 4.1.1 android目录下几个比较重要的节点

##### defaultConfig节点: 默认配置

defaultConfig {  
 applicationId "com.example.pver.high\_gradle"  
 minSdkVersion 15  
 targetSdkVersion 26  
 versionCode 1  
 versionName "1.0"  
 testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"  
}

##### sourceSets节点：目录指向配置

与我们的项目结构是相关的，默认情况下，跟maven一样，是标准的java目录结构，只不过maven是不可以修改的。

1. Java默认sourceSets配置如下，不用在gradle里写出来：

sourceSets {  
 main {  
 manifest.srcFile 'AndroidManifest.xml'  
 java.srcDirs = ['src']  
 resources.srcDirs = ['src']  
 aidl.srcDirs = ['src']  
 renderscript.srcDirs = ['src']  
 res.srcDirs = ['res']  
 assets.srcDirs = ['assets']  
 jniLibs.srcDirs = ['libs']

}

}

2，可以自己指定源码和资源文件目录

//指定源码目录  
java {  
 srcDir 'src/...'  
}  
//制定资源目录  
resources {  
 srcDir 'src/main/...'  
}

3，打包时不想包含的文件和资源

java {  
 exclude ''  
}  
resources {  
 exclude ''  
}

4，默认layout下太多了，添加两个目录分类，也可以分开添加

main.res.srcDirs += ['src/main/res-activity', 'src/main/res-fragment']

##### buildTypes 节点：build类型

1. 常用配置如下：

buildTypes {  
 debug {  
 buildConfigField  
 minifyEnabled  
 zipAlignEnabled  
 shrinkResources  
 proguardFiles  
 signingConfig  
 }  
 release {  
 ...  
 }  
}

1. **buildConfigField属性**

字段参数配置,android的gradle会根据buildTypes的配置自动生成BuildConfig.java文件,直接在项目接口配置的java文件内通过类名调用获取,可以避免每次调试与正式打包时容易忘记切换服务器的问题,省去出错麻烦.

//服务器头字段public static final String SERVER\_HEADER = BuildConfig.SERVERHEAD;

1，有两种配置方式：

buildConfigField 'boolean', 'ISB', 'true'

buildConfigField 'String', 'val', '"content"'

这样就可以在app中使用boolean类型的变量BuildConfig.ISB=true

1. 配置完成后的BuildConfig文件

public final class BuildConfig {  
 public static final boolean DEBUG = Boolean.parseBoolean("true");  
 public static final String APPLICATION\_ID = "com.example.pver.high\_gradle";  
 public static final String BUILD\_TYPE = "debug";  
 public static final String FLAVOR = "A";  
 public static final int VERSION\_CODE = 1;  
 public static final String VERSION\_NAME = "1.0";  
 // Fields from build type: debug  
 public static final String SERVERHEAD = "https://www.baidu.com/";  
}

1. **minifyEnabled属性**

debug与release的默认值都为false,如果需要混淆需要手动配置为true;

混淆文件在proguardFiles后面配置,默认为app目录下的 “proguard-rules.pro”

1. **zipAlignEnabled属性**

是否对齐app所有资源,对齐后,会使得在运行时Android与应用程序间的交互更加有效率;

配置此属性必须要有足够的信息签名程序包，Ant才会执行对齐操作，因为对齐处理发生在签名之后。为了能够签名程序包，进而执行对齐操作，Ant必须知道keystore的位置以及build.properties中key的名字。相应的属性名为key.store和key.alias。如果这些属性为空，签名工具会在编译过程中提示输入store/key的密码，然后脚本会执行签名及apk文件的对齐。如果这些属性都没有，Release程序包不会进行签名，自然也就不会进行对齐了。

1. **shrinkResources属性**

配置true,移除无用的resource文件,可以减小apk包的大小,有瘦身效果;

Tor Norbye在Google+上还透露了Android Gradle插件清除无用资源的效果：

在构建时，自动移除无用资源的功能能够大幅度减小APK文件的大小（最高可减小34%）；当前能够移除的无用资源包括图片、布局、菜单等资源文件，但不包括value资源文件。

1. **signingConfig属性**

可以直接在android{}内配置签名信息,不用在buildType里配置，代码格式如下:

signingConfigs{  
 debug{  
 ...  
 }  
 release{  
 ...  
 }  
}

##### productFlavors节点：配置多渠道打包的工具

我们都知道gradle非常强大，功能太多，今天就来给大家介绍下productFlavors的逆天用法。productFlavors通常是指同一软件的不同版本，比如收费版，免费版。这个版本的逻辑肯定是不一样的。

如果在使用gradle之前，可能需要比较头痛的写2套代码，但现在，完全可以在一套代码中解决。

实战：不同的APP名称图标和UI

1. 动态使用APP名称

Manifest中配置动态属性

xmlns:tools="http://schemas.android.com/tools"

<application  
 android:label="${APP\_NAME}"  
 tools:replace="android:label"

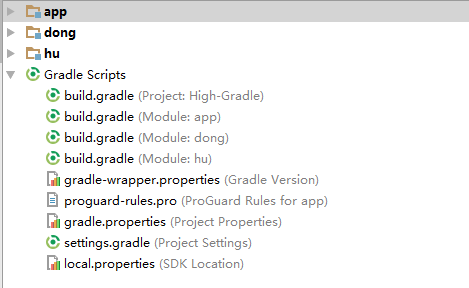
Gradle中使用

productFlavors {  
 Dong {  
 manifestPlaceholders = [APP\_NAME: "Dong"]  
 }  
 Hu {  
 manifestPlaceholders = [APP\_NAME: "Hu"]  
 }  
}

这样打不同的包就会有不用的App名称

1. APP图标和UI

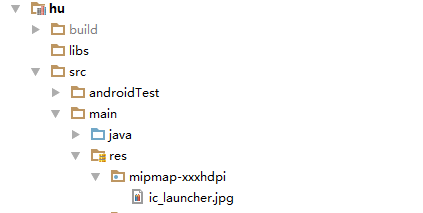
首先创建两个lib mudle，让app去依赖他们，这里是为了阅读直观，也可以在java/main里存放公共代码，创建两个与main同级的目录作为两个app目录。



App图标是声明在manifest中

<application  
 android:allowBackup="true"  
 android:icon="@mipmap/ic\_launcher"

然后在各自目录下创建bitmap目录，放入图片就行，如下



不过这样出线了一个问题，就是DongDebug和HuDebug版本的app图片是一样的，找了半天，上面有两个地方需要修改一下

一个是app依赖dong和hu两个模块时，不能使用系统自动生成的依赖格式

compile project(':dong')

compile project(':hu')

必须修改为如下格式，才能让app指向相应的mudle

dongCompile project(':dong')

huCompile project(':hu')

第二个是dongCompile这个格式不能是DongCompile，第一个字幕必须小写，所以我们在创建productFlavors时，就需要用小写了，上面的修改如下

productFlavors {  
 dong {  
 ...

}  
 hu {  
 ...  
 }  
}

这样再运行相应的版本就可以得到不用的图片，不过由于debug和release版本没有设置签名，目前只能运行debug版本

##### signingConfigs：签名配置(还有点混乱)

使用gradle自动签名有几种方式

1，比如最直接的使用方式

signingConfigs {//签名配置  
  
 release {//发布版签名配置  
 storeFile file('gh.keystore')//密钥文件路径  
 storePassword "123"//密钥文件密码  
 keyAlias "fk"//key别名  
 keyPassword "123"//key密码  
 }  
 debug {//debug版签名配置  
 storeFile file('gh.keystore')  
 storePassword "123"  
 keyAlias "fk"  
 keyPassword "123"  
 }  
}

直接在signingConfigs 里的release和debug版本里设置密钥位置和名称密码等，这里直接使用gh.keystore，是默认在app/目录下的gh.keystore文件，如果是其他位置，就填写具体位置

gh.keystore签名工具是通过studio里create的。

然后使用在buildType里使用签名，这里注意buildType必须卸载signingConfigs下面，否则会报错

buildTypes {  
  
 debug {

...  
 signingConfig signingConfigs.debug  
 }  
 release {  
 ...  
 signingConfig signingConfigs.release  
 }  
}

然后build APK即可，在G:\dym\_workspace\High-Gradle\app\build\outputs\apk目录下可以看到我们编译的版本。

2，对上面的方式进行加工

创建一个signing.properties文件，在里面配置签名工具的信息

RELEASE\_STORE\_FILE = dym.jks  
RELEASE\_KEY\_ALIAS = dongyiming  
RELEASE\_STORE\_PASSWORD= dongyiming  
RELEASE\_KEY\_PASSWORD= dongyiming  
  
DEBUG\_KEYSTORE= dym.jks  
DEBUG\_ALIAS= dongyiming  
DEBUG\_KEYSTORE\_PWD= dongyiming  
DEBUG\_KEY\_PWD= dongyiming

把signing.properties这个文件添加到.gitignore忽略文件中

然后，在gradle里加载这个配置文件，读取文件里的字段

def loadSigningConfigs() {  
 def Properties props = new Properties()  
 def propFile = file('../signing.properties')  
 if (propFile.canRead()) {  
 props.load(new FileInputStream(propFile))  
 if (props != null && props.containsKey('RELEASE\_STORE\_FILE') && props.containsKey('RELEASE\_STORE\_PASSWORD') &&  
 props.containsKey('RELEASE\_KEY\_ALIAS') && props.containsKey('RELEASE\_KEY\_PASSWORD')) {  
 android.signingConfigs.release.storeFile = file(props['RELEASE\_STORE\_FILE'])  
 android.signingConfigs.release.storePassword = props['RELEASE\_STORE\_PASSWORD']  
 android.signingConfigs.release.keyAlias = props['RELEASE\_KEY\_ALIAS']  
 android.signingConfigs.release.keyPassword = props['RELEASE\_KEY\_PASSWORD']  
 } else if (props != null && props.containsKey('DEBUG\_KEYSTORE') && props.containsKey('DEBUG\_ALIAS') &&  
 props.containsKey('DEBUG\_KEYSTORE\_PWD') && props.containsKey('DEBUG\_KEY\_PWD')) {  
 android.signingConfigs.debug.storeFile = file(props['DEBUG\_KEYSTORE'])  
 android.signingConfigs.debug.storePassword = props['DEBUG\_KEYSTORE\_PWD']  
 android.signingConfigs.debug.keyAlias = props['DEBUG\_ALIAS']  
 android.signingConfigs.debug.keyPassword = props['DEBUG\_KEY\_PWD']  
 } else {  
 android.buildTypes.release.signingConfig = null  
 }  
 } else {  
 android.buildTypes.release.signingConfig = null  
 }  
}

在android{}里使用这个方法

android{

signingConfigs {//签名配置  
 release {  
 }  
 debug {  
 }

}

loadSigningConfigs()

buildTypes {  
 debug {

...  
 signingConfig signingConfigs.debug  
 }  
 release {  
 ...  
 signingConfig signingConfigs.release  
 }  
 }

}

最后同样build apk也能得到build Variant选择的版本。

##### 补充

android {

 ...

buildTypes {  
 debug {

...  
 }  
 beta {

...  
 }  
 release {

...  
 }  
 }  
 productFlavors {  
 A {}  
 B {}  
 C {}  
 }

这样的话，最终编译后可以打9种包，他们分别是

· ADebug

· ABeta

· ARelease

· BDebug

· BBeta

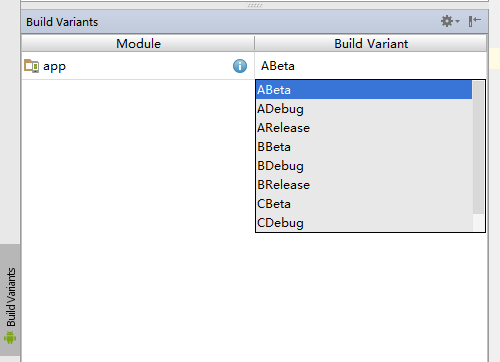
· BRelease

· CDebug

· CBeta

· CRelease

在Android Studio左下角可以找到并在每次build的时候选择不同种类的包



名字不能用纯数字

##### 补充2

我们项目使用到的框架，在debug版本上使用，但是在正式发布版本上不需要时，其依赖情况可以这么写，打包时就不用添加进去

debugCompile('...')

当我们初始化项目时，有时会在Application中加载SDK，当只有debug版本需要加载SDK时，在Application中是这么表达：

public class MyApplication extends Application {  
  
 @Override  
 public void onCreate() {  
 super.onCreate();  
 if (BuildConfig.DEBUG) {  
 XXXSDK.init(this);  
 }  
 }  
}

当我们编译release版本时会出现错误，因为我们正式打包时是不会将debug依赖的内容打进去，而我们在debug版本判断时，会使用到debug版本需要的依赖库，所以会提示找不到相应的XXX类

解决办法是在debug和release模块里，创建相同的包结构，在同一目录下创建一个SDKManager类，将debug相关写在debug模块下，release模块下的SDKManager类里写相同的方法，不过方法里不做任何处理，并且在/main/java主模块下不能有相同的类，最后在application里直接调用SDKManager的方法即可。

##### 补充3

不管使用什么IDE，Android 打包生成Apk主要都是由以下几步完成：

1，根据资源文件和AndroidManifest.xml生成R.java文件

2，处理aidl，生成对应的java文件，如果没有aidl，则跳过

3，编译工程源码（主项目，库）src目录下所有的源码，同时上边生成的R.java和aidl生成的java文件也会被编译生成相应的class文件

4，将第3步生成的class文件打包生成.dex文件

5，将资源文件打包，生成初始的apk

6，将第4步生成的.dex文件加入到apk中生成未签名的包

7，apk签名

