

前言

Gradle是一个基于Apache Ant和Apache Maven概念的项目自动化建构工具。它使用一种基于Groovy的特定领域语言来声明项目设置，而不是传统的XML。当前其支持的语言限于Java、Groovy和Scala，计划未来将支持更多的语言。

怎么看上面都是一段很官方的解释，对于入门的人来说简直是一个噩梦般的解释（包括以前的我）。那下面我就用通俗一点语言说说我的理解。

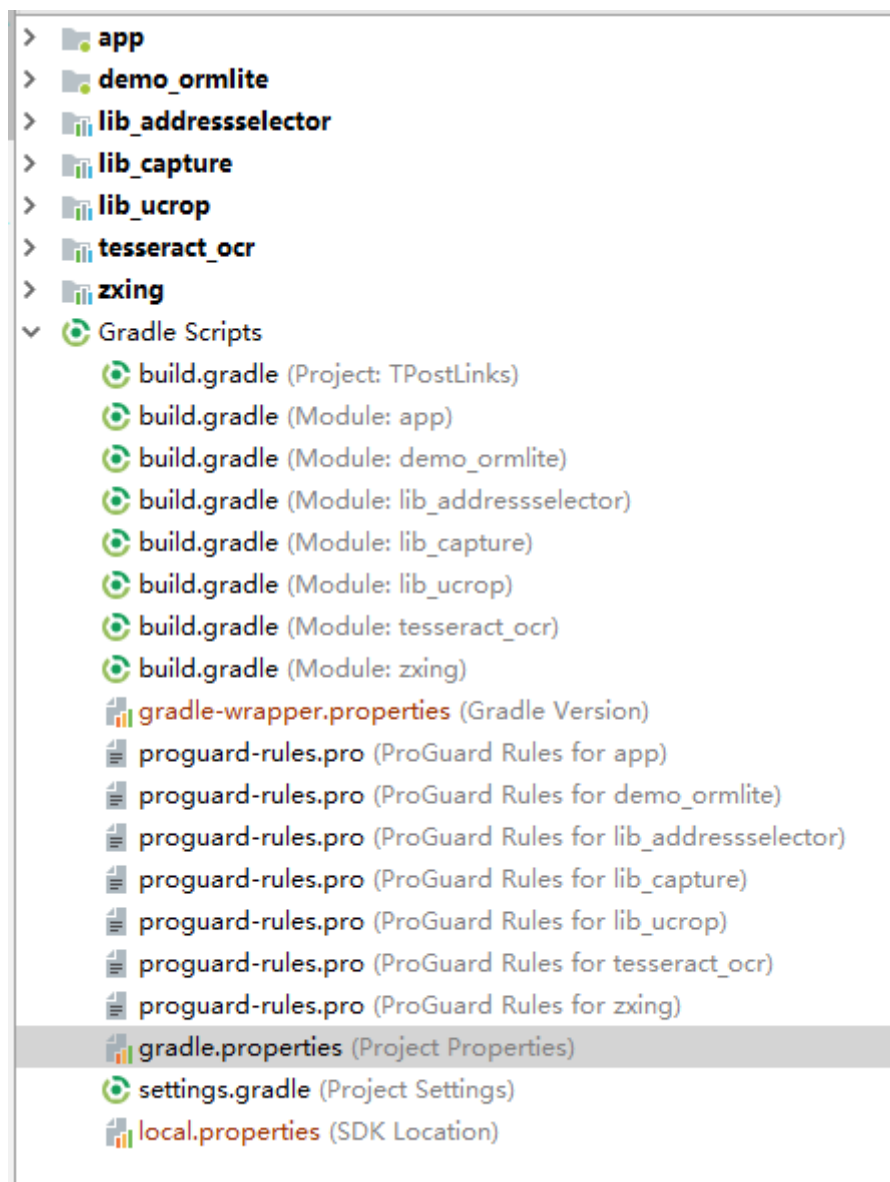
Gradle就是工程的管理，帮我们做了依赖,打包,部署,发布,各种渠道的差异管理等工作。举个例子形容，如果我是一个做大事的少爷平时管不了这么多小事情，那Gradle就是一个贴心的秘书或者管家，把一些杂七杂八的小事情都帮我们做好了，让我们可以安心的打代码，其他事情可以交给管家管。

那有人会问，既然工作都可以交给他做，为什么还要我们去了解。我想我们要管家做事，也要下达我们的命令，我们必须知道这些命令和管家的喜好才能跟他相处和谐，不然你不知道它的脾性下错命令，那后果可是很严重的。

在以前实习的时候，我还用eclipse，那是导入一个网上的下载的module还需要一步步的import。但自从用了Android Studio后，Gradle很贴心的帮我完成了这个繁杂的工作，而且往往只需要添加一句话，这太神奇了，当时我是这样想的，下面我们也会说到这个。

分析

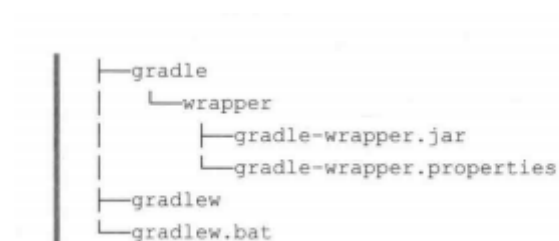
下面我就用自己项目中用到的Gradle慢慢分析：



我们看到，每个Module都会对应有一个Gradle文件，另外还有一个主Project的Gradle文件管理全局。下面我们先看看那个叫gradle-wrapper.properties的文件：

gradle-wrapper

Wrapper是对Gradle的一层包装，便于在团队开发过程中统一Gradle构建的版本号，这样大家都可以使用统一的Gradle版本进行构建。



上面我们看到的图就是Gradle提供内置的Wrapper task帮助我们自动生成Wrapper所需的目录文件。再看看我们Android项目里面自动生成的文件

✓ gradle.properties	2017/2/21 9:49	PROPERTIES 文...	1 KB
gradlew	2017/2/25 11:01	文件	5 KB
gradlew.bat	2017/2/25 11:01	Windows 批处理...	3 KB

TPostLinks > gradle > wrapper				
名称	修改日期	类型	大小	
gradle-wrapper.jar	2017/2/25 11:01	Executable Jar File	53 KB	
gradle-wrapper.properties	2018/4/16 15:03	PROPERTIES 文件	1 KB	

终于，我们知道这几个自动生成的文件原来是Gradle Wrapper创建出来的。

那下面我们看看gradle-wrapper.properties这个文件的作用

```
#Fri Oct 27 23:38:53 CST 2017
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-4.4-all.zip
```

看到项目里面的各个属性，下面再看看每个属性的作用

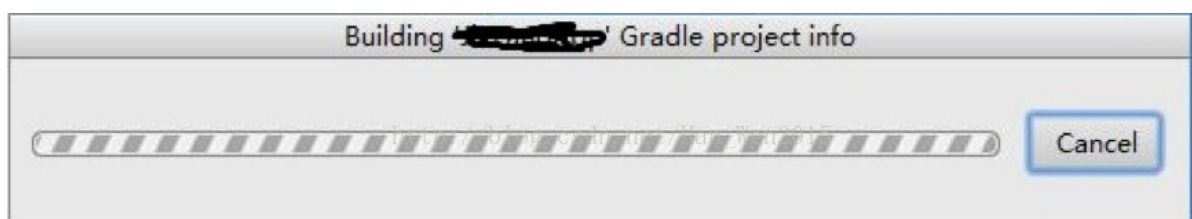
表 1-2 gradle-wrapper.properties 的配置字段	
字段名	说明
distributionBase	下载的 Gradle 压缩包解压后存储的主目录
distributionPath	相对于 distributionBase 的解压后的 Gradle 压缩包的路径
zipStoreBase	同 distributionBase，只不过是存放 zip 压缩包的
zipStorePath	同 distributionPath，只不过是存放 zip 压缩包的
distributionUrl	Gradle 发行版压缩包的下载地址

我们其实最关心的应该是distributionUrl这个属性，他是下载Gradle的路径，它下载的东西会出现在以下的文件夹中

.gradle > wrapper > dists >				
名称	修改日期	类型	大小	
gradle-2.14.1-all	2017/10/31 23:41	文件夹		
gradle-3.3-all	2017/3/7 10:05	文件夹		
gradle-3.5-all	2018/1/16 20:57	文件夹		
gradle-4.1-all	2017/10/31 21:26	文件夹		
gradle-4.4-all	2018/4/1 16:49	文件夹		
gradle-3.3-all.zip	2017/4/21 16:01	360压缩 ZIP 文件	177,428 KB	

看到了吧，这个文件夹包含了各个版本你下载的Gradle。

当我是初学者的时候老是会遇到一个问题，那就是下图：



导入项目的时候一直会停留在这个界面，这是为什么？其实原因很简单，就是你常用项目的Gradle版本跟你新导入项目的Gradle版本不一致造成的，那怎么解决？我本人自己是这么做的：

1. 网速好或者科学上网的时候，由它自己去下载，不过下载时间有长有短，不能保证。
2. 当你在公司被限网速的时候，当然也是我最常用的，就是把你最近常用项目的gradle-wrapper.properties文件替换掉你要导入项目的该文件，基本上我是这样解决的，当然有时候也会遇到替换掉报错的情况，不过比较少。

settings.gradle

下面我们讲讲settings.gradle文件，它其实是用于初始化以及工程树的配置的，放在根工程目录下。

设置文件大多数的作用都是为了配置自工程。在Gradle众多工程是通过工程树表示的，相当于我们在Android Studio看到的Project和Module概念一样。根工程相当于Android Studio的Project，一个根工程可以有多个自工程，也就是很多Module，这样就和Android Studio定义的Module概念对应上了。



我们可以看到这个项目我们添加了7个module，一一对应，如果你的项目添加了项目依赖，那就会出现在这个文件当中。

好了，我们说完settings.gradle文件之后就慢慢进入其他文件了，但是首先我们要解释一下什么是Groovy：

Groovy

Groovy是基于JVM虚拟机的一种动态语言，它的语法和Java非常相似，由Java入门学习Groovy基本没有障碍。Groovy完全兼容Java，又在此基础上增加了很多动态类型和灵活的特性，比如支持密保，支持DSL，可以说它就是一门非常灵活的动态脚本语言。

一开始我总把Gradle和Groovy搞混了，现在我总把他们的关系弄清楚了。Gradle像是一个软件，而Groovy就是写这个软件的语言，这就很简单明了吧。那下面我们说到的内容都是用Groovy语法写的，但是这个知识点我就暂时不科普了，有兴趣的小伙伴可以去了解一下更深入的Groovy语法。

build.gradle (Project)

下面我们就来讲讲主的build.gradle文件：

```

buildscript { 1
    ext.kotlin_version = '1.1.3-2' 2
    repositories { 3
        jcenter()
        maven{
            url 'http://maven.aliyun.com/nexus/content/groups/public'
        }
        google()
    }
    dependencies { 4
        classpath 'com.android.tools.build:gradle:3.1.1'
        classpath "org.jetbrains.kotlin:kotlin-android-extensions:1.1.2"
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

```

我们这里，分为四个标签来讲：

1.buildscript

buildscript中的声明是gradle脚本自身需要使用的资源。可以声明的资源包括依赖项、第三方插件、maven仓库地址等

2.ext

ext是自定义属性，现在很多人都喜欢把所有关于版本的信息都利用ext放在另一个自己新建的gradle文件中集中管理，下面我介绍一下ext是怎么用的：

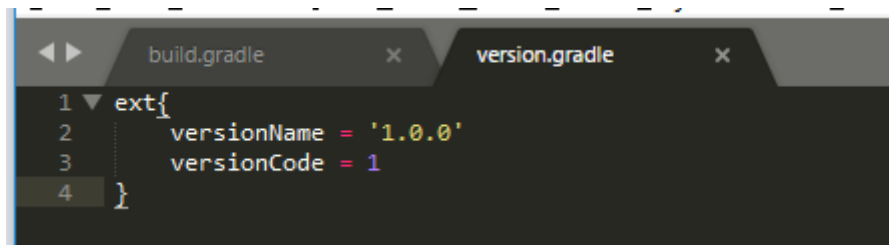
此电脑 > 系统 (E:) > AndroidProject > gradle project				
名称	修改日期	类型	大小	
.gradle	2018/4/16 22:05	文件夹		
build.gradle	2018/4/16 22:04	GRADLE 文件	1 KB	
version.gradle	2018/4/16 22:05	GRADLE 文件	1 KB	

1. 首先我们新建两个文件，分别叫build.gradle和version.gradle

```

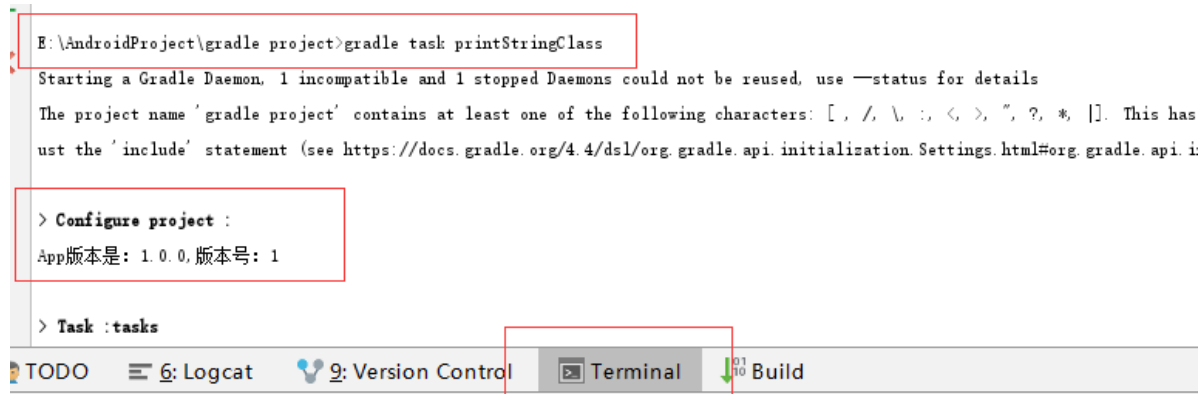
build.gradle
version.gradle
1
2  apply from:'version.gradle'
3
4  task printStringClass {
5
6
7      println "App版本是: ${versionName},版本号: ${versionCode}"
8
9
10 }
11

```



```
1 ext{
2     versionName = '1.0.0'
3     versionCode = 1
4 }
```

1. 然后分别在两个文件中打上相应的代码



```
E:\AndroidProject\gradle project>gradle task printStringClass
Starting a Gradle Daemon, 1 incompatible and 1 stopped Daemons could not be reused, use --status for details
The project name 'gradle project' contains at least one of the following characters: [ , /, \, :, <, >, ", ?, *, ]. This has
ust the 'include' statement (see https://docs.gradle.org/4.4/dsl/org.gradle.api.initialization.Settings.html#org.gradle.api.i

> Configure project :
App版本是: 1.0.0, 版本号: 1

> Task :tasks
```

1. 最后在Android Studio的Terminal移动到相应的文件夹中运行task。

我们可以很神奇的发现，当我们在build.gradle文件中输入了apply from:'version.gradle'这句话，我们就可以读取到该文件下ext的信息。

现在在项目中我也是这种方法统一管理所有第三方插件的版本号的，有兴趣的朋友也可以试试。

3.repositories

顾名思义就是仓库的意思啦，而jcenter()、maven()和google()就是托管第三方插件的平台

4.dependencies

当然配置了仓库还不够，我们还需要在dependencies{}里面的配置里，把需要配置的依赖用classpath配置上，因为这个dependencies在buildscript{}里面，所以代表的是Gradle需要的插件。

下面我们再看看build.gradle (Project) 的另一部分代码



```
allprojects {
    repositories {
        jcenter()
        maven {
            url 'https://dl.bintray.com/jetbrains/anko'
        }
        google()
    }
}
```

allprojects

allprojects块的repositories用于多项目构建，为所有项目提供共同所需依赖包。而子项目可以配置自己的repositories以获取自己独需的依赖包。

奇怪，有人会问，为什么同一个build.gradle (Project) 文件中buildscript和allprojects里面的内容基本上是一样的呢，他们的区别在哪？

buildscript和allprojects的作用和区别

buildscript中的声明是gradle脚本自身需要使用的资源，就是说他是管家自己需要的资源，跟你这个大少爷其实并没有什么关系。而allprojects声明的却是你所有module所需要使用的资源，就是说如果大少爷你的每个module都需要用同一个第三库的时候，你可以在allprojects里面声明。这下解释应该可以明白了吧。

好了，下面该说说build.gradle (Project) 文件的最后一个一段代码了

```
task clean(type: Delete) {  
    delete rootProject.buildDir  
}
```

运行gradle clean时，执行此处定义的task。该任务继承自Delete，删除根目录中的build目录。相当于执行Delete.delete(rootProject.buildDir)。其实这个任务的执行就是可以删除生成的Build文件的，跟Android Studio的clean是一个道理。

build.gradle (Module)

讲完Project的build文件，就来讲讲最后也是内容最多的文件了。

apply plugin

```
apply plugin: 'com.android.application'  
apply plugin: 'kotlin-android'  
apply plugin: 'kotlin-android-extensions'
```

首先要说下apply plugin: 'xxx'

这种叫做引入Gradle插件，而Gradle插件大致分为分为两种：

1. **apply plugin: 'xxx'**: 叫做二进制插件，二进制插件一般都是被打包在一个jar里独立发布的，比如我们自定义的插件，再发布的时候我们也可以为其指定plugin id，这个plugin id最好是一个全限定名称，就像你的包名一样；
2. **apply from: 'xxx'**: 叫做应用脚本插件，其实这不能算一个插件，它只是一个脚本。应用脚本插件，其实就是把这个脚本加载进来，和二进制插件不同的是它使用的是from关键字.后面紧跟的站一个脚本文件，可以是本地的，也可以是网络存在的，如果是网络上的话要使用HTTP URL。虽然它不是一个真正的插件，但是不能忽视它的作用.它是脚本文件模块化的基础，我们可以把庞大的脚本文件.进行分块、分段整理.拆分成一个个共用、职责分明的文件，然后使用apply from来引用它们，比如我们可以把常用的函数放在一个Utils.gradle脚本里，供其他脚本文件引用。示例中我们把 App的版本名称和版本号单独放在一个脚本文件里，清晰、简单、方便、快捷.我们也可以使用自动化对该文件自动处理，生成版本。

说说Gradle插件的作用

把插件应用到你的项目中，插件会扩展项目的功能，帮助你在项目的构建过程中做很多事情。1.可以添加任务到你的项目中，帮你完成一些事情，比如测试、编译、打包。2.可以添加依赖配置到你的项目中，我们可以通过它们配置我们项目在构建过程中需要的依赖。比如我们编译的时候依赖的第三方库等。3.可以向项目中现有的对象类型添加新的扩展属性、方法等，让你可以使用它们帮助我们配置、优化构建，比如android{}这个配置块就是Android Gradle插件为Project对象添加的一个扩展。4.可以对项目进行一些约定，比如应用Java插件之后，约定src/main/java目录下是我们的源代码存放位置，在编译的时候也是编译这个目录下的Java源代码文件。

然后我们说说'com.android.application'

Android Gradle插件的分类其实是根据Android工程的属性分类的。在Android中有3类工程，一类是App应用工程，它可以生成一个可运行的apk应用；一类是Library库工程，它可以生成AAR包给其他的App工程公用，就和我们的Jar一样，但是它包含了Android的资源等信息，是一个特殊的Jar包；最后一类是Test测试工程，用于对App工程或者Library库工程进行单元测试。

1. App插件id: com.android.application.
2. Library插件id: com.android.library.
3. Test插件id: com.android.test.

一般一个项目只会设置一个App插件，而module一般是会设置为Library插件。

```
android {  
    compileSdkVersion 24  
  
    defaultConfig {  
        applicationId  
        minSdkVersion 15  
        targetSdkVersion 22  
        versionCode  
        versionName  
        flavorDimensions "applicationId"  
        flavorDimensions "versionName"  
        multiDexEnabled true  
        ndk {  
            //设置支持的SO库架构  
            abiFilters 'armeabi', 'x86', 'armeabi-v7a', 'x86_64'  
        }  
    }  
  
    sourceSets {  
        main {  
            jniLibs.srcDirs = ['libs']  
        }  
    }  
}
```


android{}

是Android插件提供的一个扩展类型，可以让我们自定义Android Gradle工程，是Android Gradle工程配置的唯一入口。

compileSdkVersion

是编译所依赖的Android SDK的版本，这里是API Level。

buildToolsVersion

是构建该Android工程所用构建工具的版本。

defaultConfig{}

defaultConfig是默认的配置，它是一个ProductFlavor。ProductFlavor允许我们根据不同的情况同时生成多个不同的apk包。

applicationId

配置我们的包名，包名是app的唯一标识，其实他跟AndroidManifest里面的package是可以不同的，他们之间并没有直接的关系。

package指的是代码目录下路径；applicationId指的是app对外发布的唯一标识，会在签名、申请第三方库、发布时候用到。

minSdkVersion

是支持的Android系统的api level，这里是15，也就是说低于Android 15版本的机型不能使用这个app。

targetSdkVersion

表明我们是基于哪个Android版本开发的，这里是22。

versionCode

表明我们的app应用内部版本号，一般用于控制app升级，当然我在使用的bugly自动升级能不能接受到升级推送就是基于这个。

versionName

表明我们的app应用的版本名称，一般是发布的时候写在app上告诉用户的，这样当你修复了一个bug并更新了版本，别人却发现说怎么你这个bug还在，你这时候就可以自信的告诉他自已看下app的版本号。
(亲身经历在撕逼的时候可以从容的应对)

multiDexEnabled

用于配置该BuildType是否启用自动拆分多个Dex的功能。一般用程序中代码太多，超过了65535个方法的时候。

ndk{}

多平台编译，生成有so包的时候使用，包括四个平台'armeabi', 'x86', 'armeabi-v7a', 'mips'。一般使用第三方提供的SDK的时候，可能会附带so库。

sourceSets

源代码集合，是Java插件用来描述和管理源代码及资源的一个抽象概念，是一个Java源代码文件和资源文件的集合，我们可以通过sourceSets更改源集的Java目录或者资源目录等。

譬如像上图，我通过sourceSets告诉了Gradle我的关于jni so包的存放路径就在app/libs上了，叫他编译的时候自己去找。

```
buildTypes {
    debug {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
    release {
        minifyEnabled true
        // Zipalign优化
        zipAlignEnabled true
        // 移除无用的resource文件
        shrinkResources true
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
}
```

name: build type的名字

applicationIdSuffix: 应用id后缀

versionNameSuffix: 版本名称后缀

debuggable: 是否生成一个debug的apk

minifyEnabled: 是否混淆

proguardFiles: 混淆文件

signingConfig: 签名配置

manifestPlaceholders: 清单占位符

shrinkResources: 是否去除未利用的资源，默认false，表示不去除。

zipAlignEnable: 是否使用zipalign工具压缩。

multiDexEnabled: 是否拆成多个Dex

multiDexKeepFile: 指定文本文件编译进主Dex文件中

multiDexKeepProguard: 指定混淆文件编译进主Dex文件中

buildType

构建类型，在Android Gradle工程中，它已经帮我们内置了debug和release两个构建类型，两种模式主要区别在于，能否在设备上调试以及签名不一样，其他代码和文件资源都是一样的。一般用在代码混淆，而指定的混淆文件在下图的目录上，minifyEnabled=true就会开启混淆：

- proguard-rules.pro (ProGuard Rules for app)
- proguard-rules.pro (ProGuard Rules for demo_ormlite)
- proguard-rules.pro (ProGuard Rules for lib_addressselector)
- proguard-rules.pro (ProGuard Rules for lib_capture)
- proguard-rules.pro (ProGuard Rules for lib_ucrop)
- proguard-rules.pro (ProGuard Rules for tesseract_ocr)
- proguard-rules.pro (ProGuard Rules for zxing)

```
signingConfigs {
    debug {
        storeFile file("debug.keystore")
        storePassword "android"
        keyAlias "androiddebugkey"
        keyPassword "android"
    }
    release {
        storeFile file("release.keystore")
        storePassword "release"
        keyAlias "releasekey"
        keyPassword "release"
        // 自定义输出配置
        android.applicationVariants.all { variant ->
            variant.outputs.all {
                outputFileName = "app-release.apk"
            }
        }
    }
}
```

signingConfigs

签名配置，一个app只有在签名之后才能被发布、安装、使用，签名是保护app的方式，标记该app的唯一性。如果app被恶意删改，签名就不一样了，无法升级安装，一定程度保护了我们的app。而signingConfigs就很方便为我们提供这个签名的配置。storeFile签名文件，storePassword签名证书文件的密码，storeType签名证书类型，keyAlias签名证书中秘钥别名，keyPassword签名证书中改密钥的密码。

默认情况下，debug模式的签名已经被配置好了，使用的就是Android SDK自动生成的debug证书，它一般位于\$HOME/.android/debug.keystore,其key和密码是已经知道的，一般情况下我们不需要单独配置debug模式的签名信息。

```

productFlavors {
    GZ_test {
        "applicationId" "com.example.test"
        "versionName" "0.7.58"
        manifestPlaceholders = [APP_NAME: "(测试)"]
        buildConfigField "String", "BASE_URL", "\"\\\"\""
        buildConfigField "String", "VERSION", "\"\\\"GZ\\\"\""
        buildConfigField "boolean", "isTest", "true"
    }

    GZ_verification {
        "applicationId" "com.example.test"
        "versionName" "1.5.8"
        manifestPlaceholders = [APP_NAME: "(验证)"]
        buildConfigField "String", "BASE_URL", "\"\\\"\""
        buildConfigField "String", "VERSION", "\"\\\"GD\\\"\""
        buildConfigField "boolean", "isTest", "true"
    }
}

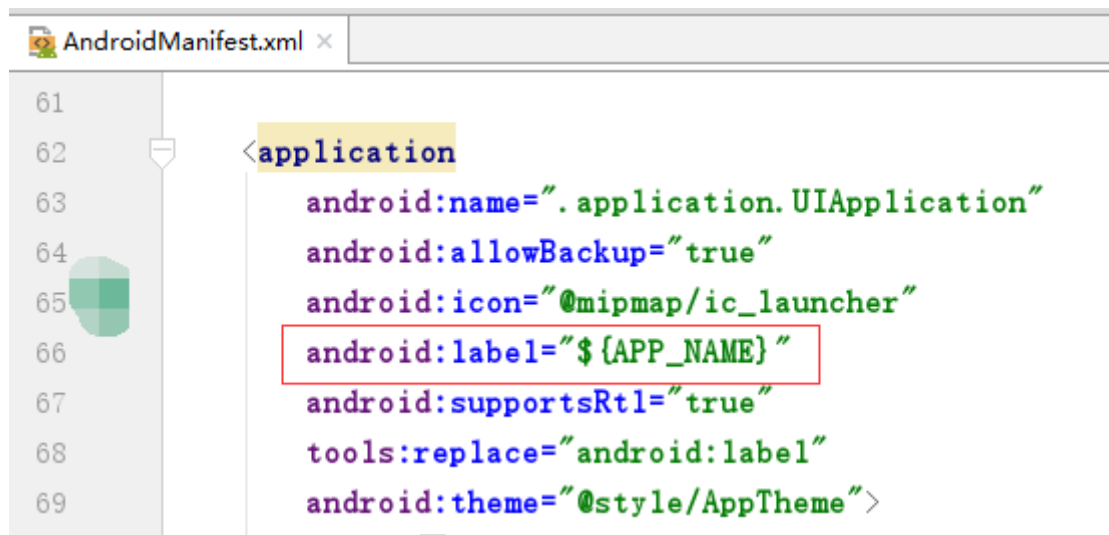
```

productFlavors

在我看来他就是Gradle的多渠道打包，你可以在不同的包定义不同的变量，实现自己的定制化版本的需求。

manifestPlaceholders

占位符，我们可以通过它动态配置AndroidManifest文件一些内容，譬如app的名字：



看看上图，我们就能发现我们在productFlavors中定义manifestPlaceholders = [APP_NAME: "(测试)"]之后，在AndroidManifest的label加上"\${APP_NAME}"，我们就能控制每个包打出来的名字是我们想要不同的名字，譬如测试服务器和生产服务器的包应该名字不一样。

buildConfigField

他是BuildConfig文件的一个函数，而BuildConfig这个类是Android Gradle构建脚本在编译后生成的。而buildConfigField就是其中的自定义函数变量，看下图我们分别定义了三个常量：

```

buildConfigField "String", "BASE_URL", "\"\\\"\""
buildConfigField "String", "VERSION", "\"\\\"GZ\\\"\""
buildConfigField "boolean", "isTest", "true"

```

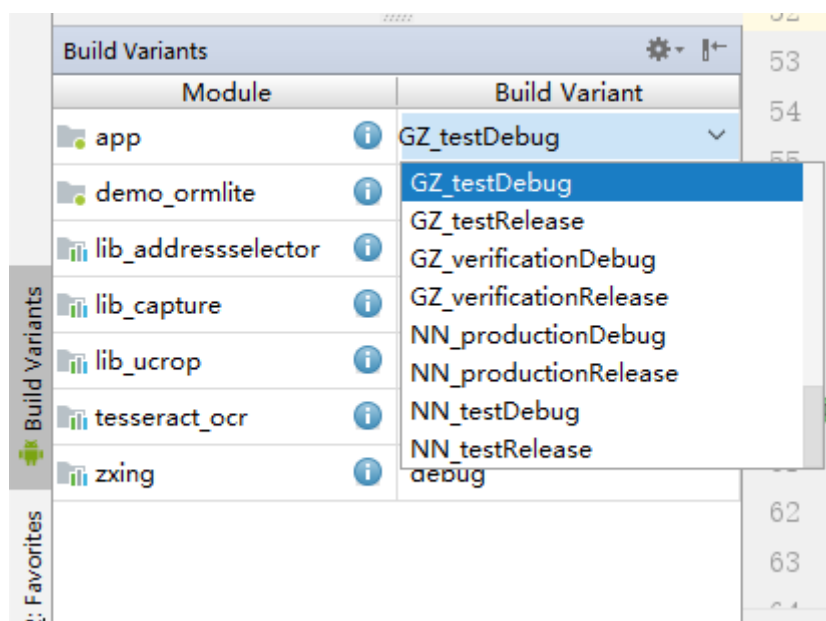
我们可以在BuildConfig文件中看到我们声明的三个变量

```
public final class BuildConfig {
    public static final boolean DEBUG = Boolean.parseBoolean(s: "true");
    public static final String APPLICATION_ID = "com.example.myapplication";
    public static final String BUILD_TYPE = "debug";
    public static final String FLAVOR = "GZ_test";
    public static final int VERSION_CODE = 1;
    public static final String VERSION_NAME = "0.7.58";
    // Fields from product flavor: GZ_test
    public static final String BASE_URL = "http://example.com/";
    public static final String VERSION = "GZ";
    public static final boolean isTest = true;
}
```

然后我们就可以在代码中用这些变量控制不同版本的代码：

```
if (BuildConfig.isTest) {
    params.put("money", "0.01"); //测试写死是0.01
} else {
    params.put("money", details.getPayment());
}
```

我们这样加个if，就可以轻轻松松的控制测试和生产版本付费的问题了，再也不用手动的改来改去了，那问题来了，我怎么去选择不同的版本呢，看下图：



如果你是Android Studio，找到Build Variants就可以选择你当前要编译的版本啦。

flavorDimensions

顾名思义就是维度，Gradle3.0以后要用flavorDimensions的变量必须在defaultConfig{}中定义才能使用，不然会报错：

```
Error:All flavors must now belong to a named flavor dimension.
The flavor 'flavor_name' is not assigned to a flavor dimension.
```

```
flavorDimensions "applicationId"
flavorDimensions "versionName"
```

这样我们就可以在不同的包中形成不同的applicationId和versionName了。

```
testOptions {
    dexOptions {
        incremental true
        javaMaxHeapSize "4g"
    }
}
```

dexOptions{}

我们知道，Android中的Java源代码被编译成class字节码后，在打包成apk的时候被dx命令优化成Android虚拟机可执行的DEX文件。DEX文件比较紧凑，Android费尽心思做了这个DEX格式，就是为了能使我们的程序在Android中平台上运行快一些。对于这些生成DEX文件的过程和处理，Android Gradle插件都帮我们处理好了，Android Gradle插件会调用SDK中的dx命令进行处理。但是有的时候可能会遇到提示内存不足的错误，大致提示异常是java.lang.OutOfMemoryError: GC overhead limit exceeded,为什么会提示内存不足呢？其实这个dx命令只是一个脚本，它调用的还是Java编写的dx.jar库，是Java程序处理的，所以当内存不足的时候，我们会看到这个Java异常信息。默认情况下给dx分配的内存是一个G8,也就是1024MB。

所以我们只需要把内存设置大一点，就可以解决这个问题，上图我的项目就把内存设置为4g。

```
dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
    testCompile 'junit:junit:4.12'
    compile 'com.android.support:appcompat-v7:24.2.0'
    compile 'com.android.support:design:24.2.0'
    compile 'com.android.support:support-v4:24.2.0'
    compile 'com.chanven.lib:cptr:1.0.0'
    compile project(path: ':lib_capture')
    compile project(path: ':lib_ucrop')
```

dependencies{}

我们平时用的最多的大概就这个了，

1. 首先第一句**compile fileTree(include: ['*.jar'], dir: 'libs')**，这样配置之后本地libs文件夹下的扩展名为jar的都会被依赖，非常方便。
2. 如果你要引入某个本地module的话，那么需要用**compile project('xxx')**。
3. 如果要引入网上仓库里面的依赖，我们需要这样写

```
compile group: 'com.squareup.okhttp3',name:'okhttp',version:'3.0.1'
```

当然这样是最完整的版本，缩写就把group、name、version去掉，然后以":"分割即可。

```
compile 'com.squareup.okhttp3:okhttp:3.0.1'
```

表 6-1		gradle 提供的依赖配置	
名称	继承自	被哪个任务使用	意义
compile	-	compileJava	编译时依赖
runtime	compile	-	运行时依赖
testCompile	compile	compileTestJava	编译测试用例时依赖
testRuntime	runtime, testCompile	test	仅仅在测试用例运行时依赖
archives	-	uploadArchives	该项目发布构件（JAR 包等）依赖
default	runtime	-	默认依赖配置

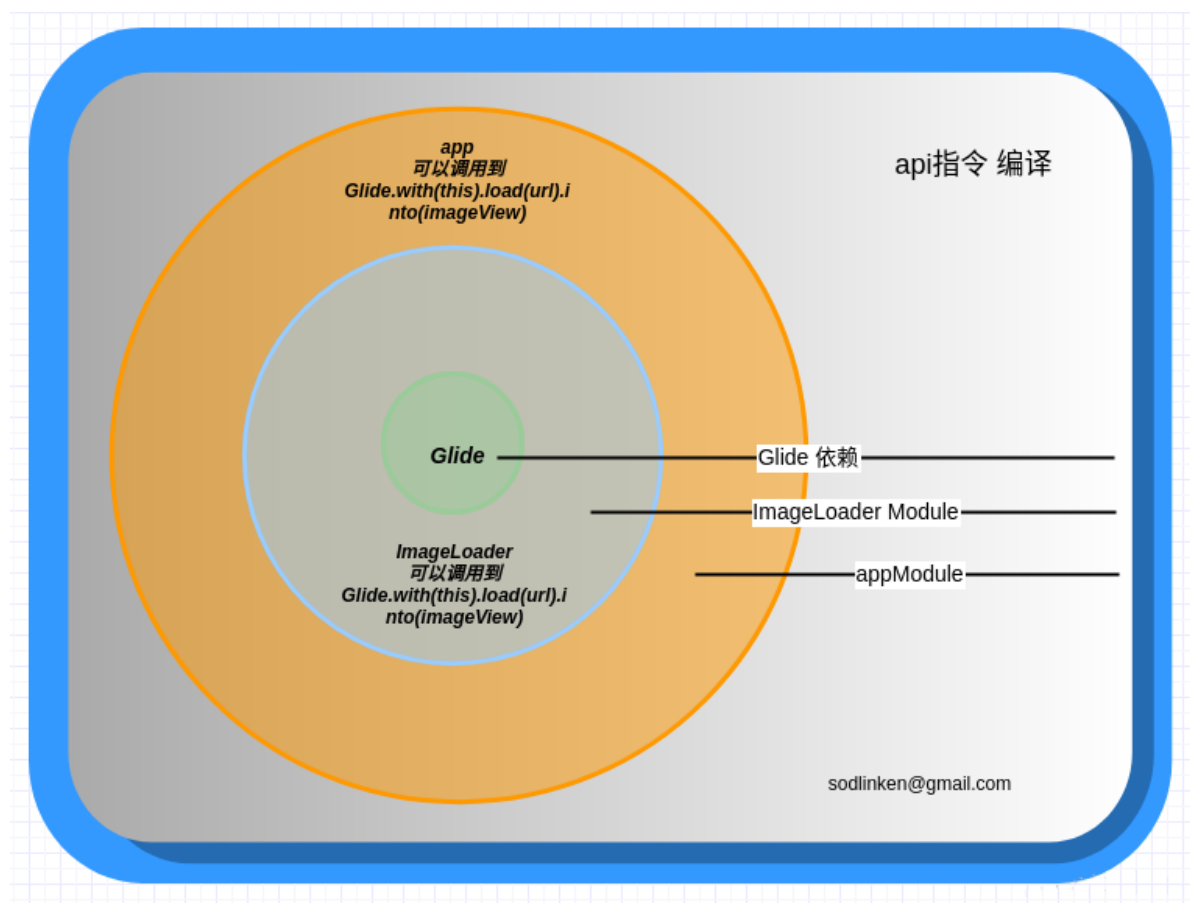
但是到了gradle3.0以后build.gradle中的依赖默认为implementation，而不是之前的compile。另外，还有依赖指令api。

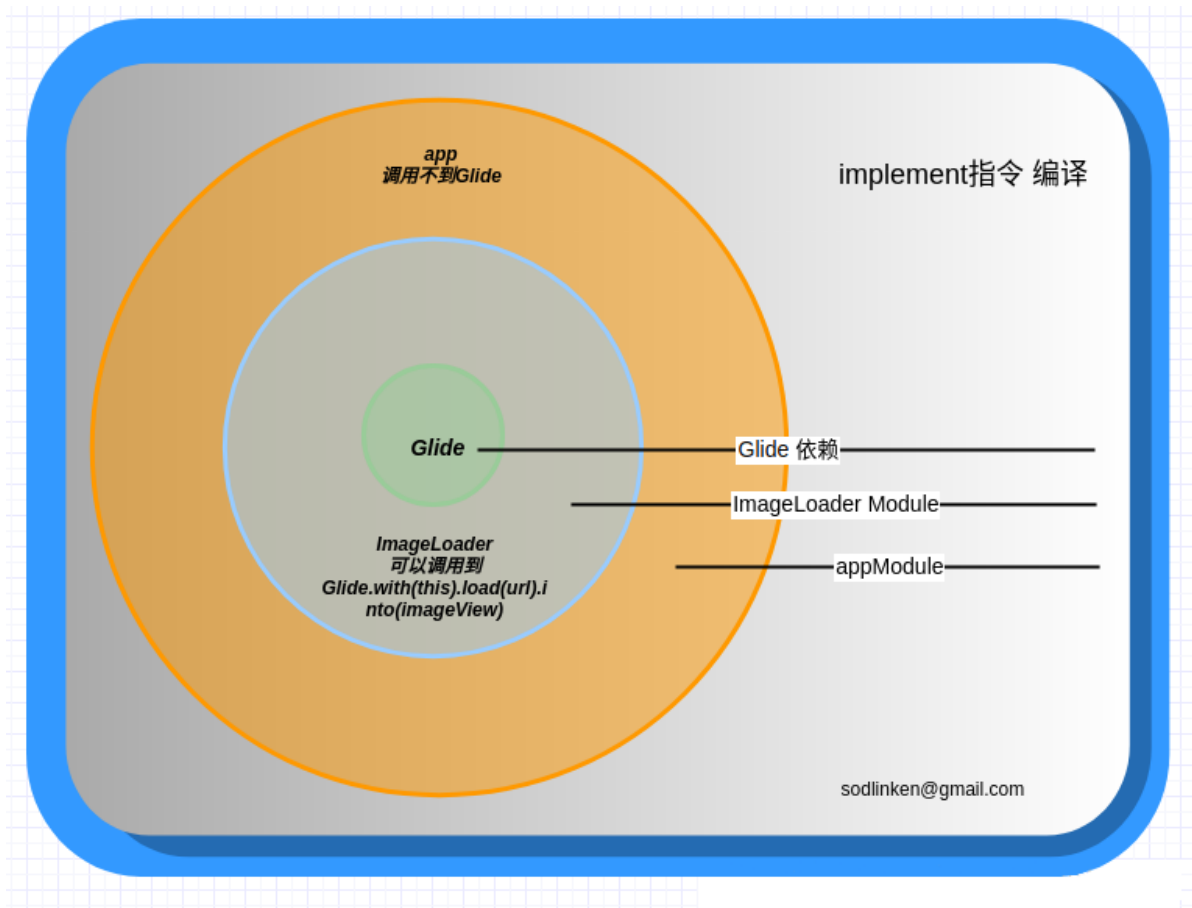
那么下面我们来说一说：

gradle 3.0中依赖implementation、api的区别：

其实api跟以前的compile没什么区别，将compile全部改成api是不会错的；

而implementation指令依赖是不会传递的，也就是说当前引用的第三方库仅限于本module内使用，其他module需要重新添加依赖才能用，下面用两个图说明：





相信看过图的人都会一目了然了。