# Gradle插件更新说明URL

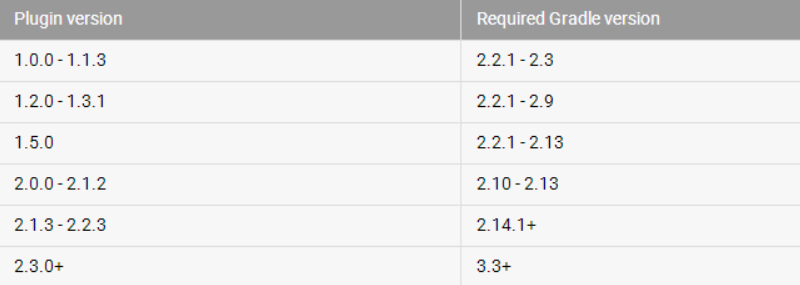
我对于插件的理解：Eclipse要想使用SDK来开发Android项目，需要ADT插件，AndroidStudio想要使用Gradle，则需要Gradle插件。也就是说，一个软件，想使用另外的软件，则需要对应软件的插件，而且这个插件会不断更新，所以就有了插件的版本变化，AS中的Gradle插件版本是在项目根目录的build.gradle中指定的，如下：



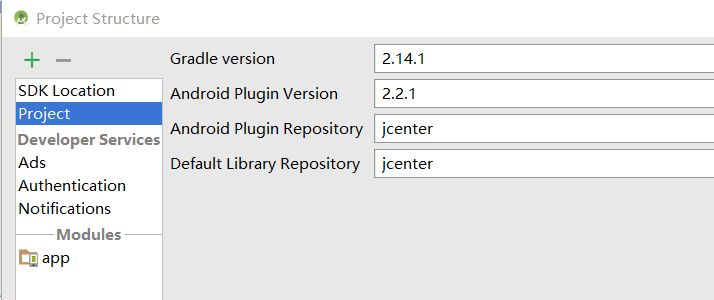
这个插件就叫“构建脚本”，它是从jcenter中下载到的。

关于插件的更新，会有更新说明，官网介绍网址：<https://developer.android.com/studio/releases/gradle-plugin.html>

不同的Gradle插件只能支持相应版本的Gradle，具体如下：



可以使用图形界面修改Gradle插件和Gradle的版本，在项目结构中，如下：



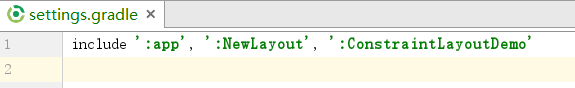
# 第一章 Android开发 Gradle基础

## 1.1 Android中的Gradle构建文件

创建一个新的Android项目时，会生成3个Gradle文件：settings.gradle、build.gradle、app/build.gradle

### settings.gradle

配置了项目中包含的模块，如下：



### 顶层的build.gradle

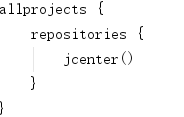


用于为所有的模块设置通用的配置

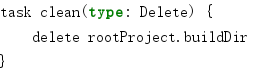


buildscript翻译过来为“构建脚本”，这个代码块就是用于配置从哪下载构建脚本插件，从图可知是从jcenter仓库中下载，下载的是gradle插件，这个gradle插件就是“构建脚本”，用于构建Android项目。

jcenter即Bintray Jcenter Artifactory库。也支持其他的库（特别是mavenCentral()，默认的Maven仓库），但是Jcenter现在已经成默认的了。Jcenter中的内容都使用HTTPS连接并通过CDN来提供，同时也更快速。



allprojects配置了所有模块都默认使用jcenter仓库来下载Java依赖库



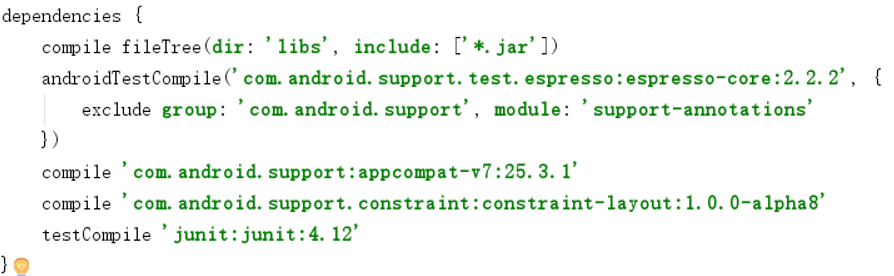
Gradle允许自定义task，上面代码就是一个自定义的task，type: Delete表示这个task是Gradle内置的Delete task的一个实例，在这个task 中，它会把项目根目录下的build目录删除。

### 模块里的build.gradle





通过Gradle的apply功能，将Android插件添加到构建系统中，从而能够使用Android的领域特定语言（Domain Specific Language, DSL）配置。



dependencies用于配置依赖，compile fileTree用于依赖一整个目录树里的文件，dir指定了依赖的目录，include为过滤器，表示只依赖\*.jar文件。

testComile 用于指定测试的依赖，也就是说这些依赖只能在src/AndroidTest/java或者src/test/java，这意味着打包apk时这个测试的依赖jar不会打包到apk中

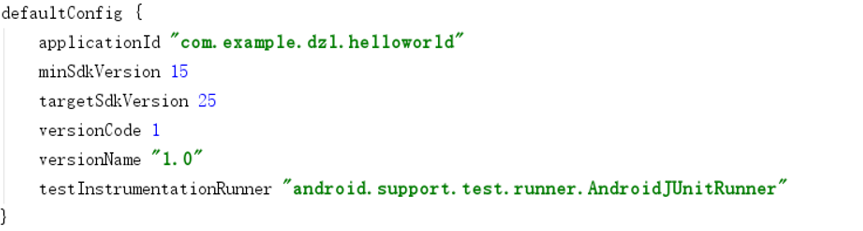
## 1.2 配置SDK版本以及其他的默认值

顶层的build.gradle通过buildscript块将Gradle的Android插件添加到了项目中，模块中的build.gradle通过apply应用了这个插件，因此可以将android块添加到Gradle的DSL中。在android块里面，你可以指定几个项目属性，如下：



android块是Android DSL的入口,compileSdkVersion和buildToolVersion指定编译SDK版本和构建工具版本，这些版本应该指定为最新的版本，因为它们是身后兼容的，并且包含了之前版本的漏洞修复。

defaultConfig块：



applicationId 应用的唯一标识。应用一但上线，这个id就不能修改。一修改就相当于是另一个应用了。

minSdkVersion 应用程序支持的最小Android SDK版本，小于这个版本的设备在访问Google Play商店时将不会看到这个应用程序。

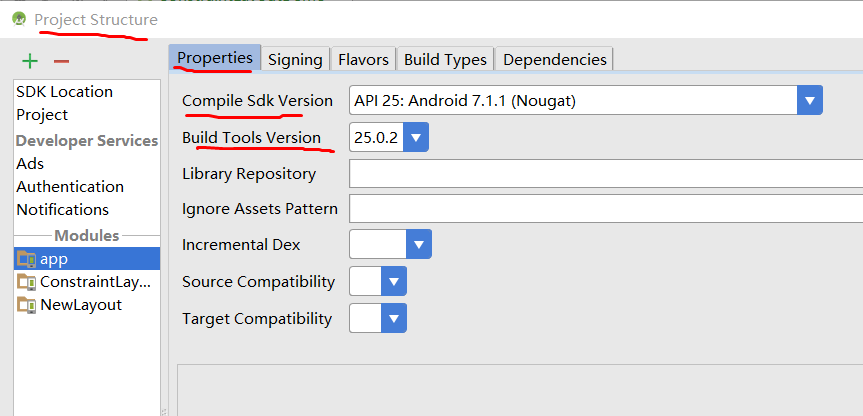
targetSdkVersion 应用程序的目标版本号。如果没有指定为最新版本，AS会给出警告，但是你可以作用任何的版本。

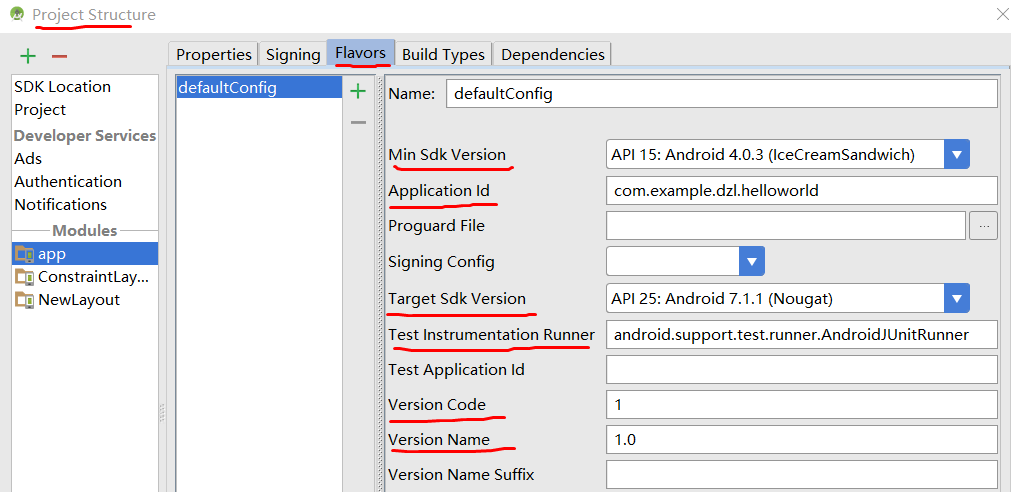
versionCode 应用程序的版本号，用于升级

versionName 版本号的名字，用于显示给用户看，通常的格式为：<major>.<minor>.<version>，就像大多数的项目一样。

testInstrumentationRunner

这些属性在AS的项目结构窗口中有图形化的显示，如下：





defaultConfig块的文档，以及其他元素的DSL，都可以在DSL参考中找到：<http://bit.ly/gradle-dsl> 或 <http://google.github.io/android-gradle-dsl/current/index.html>

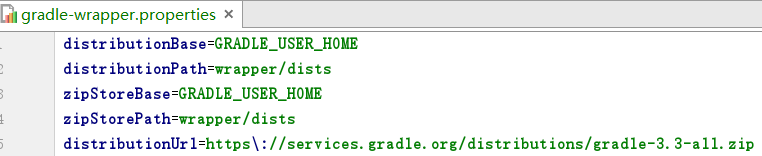
## 1.3 在命令行执行Gradle构建

在命令行执行Gradle构建，使用的是AS提供的Gradle wrapper或者安装Gradle直接运行。你不需要为了构建Android项目而安装Gradle，AS包含了一个Gradle的发行版本（以一个插件的形式）并包含了专门的功能来支持它。

“Gradle wrapper”引用gradlew脚本和gradlew.bat脚本，bradlew给UNIX使用，gradlew.bat给Windows使用，这两个项目在项目的根目录中，结尾的“w”代表“wrapper”。

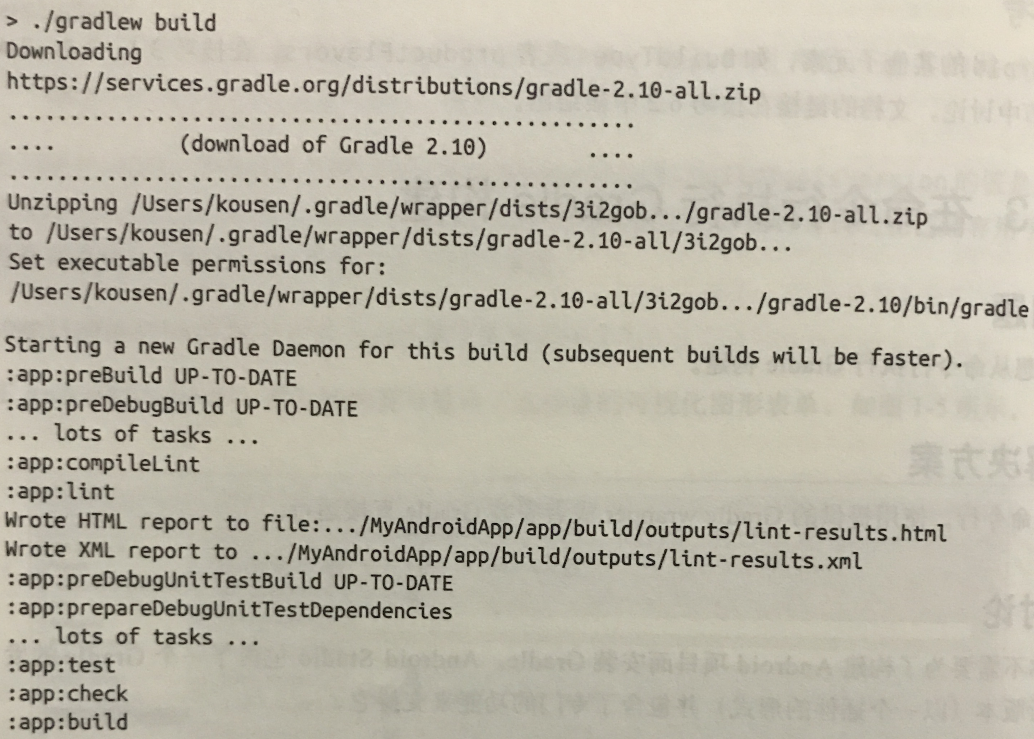
Gradle Wrapper的第一个目的就是允许客户端无须安装Gradle即可运行Gradle脚本（其实所谓的无需安装只是说不需要我们手动去安装，AS会根据wrapper中的配置去下载gradle并安装）。wrapper使用了项目根目录下的gradle/wrapper下的gradle-wrapper.jar和gradle.wrapper.properties文件来启动进程。

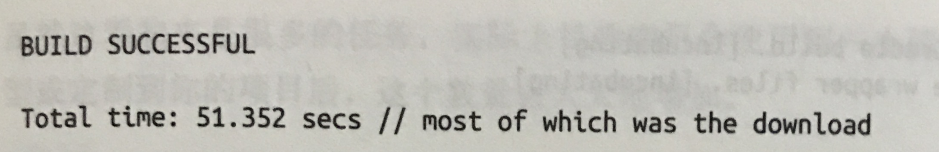
gradle.wrapper.properties文件内容如下：



distributionUrl显示了wrapper会下载并安装Gradle 3.3(可以修改为任意可用的版本),下载保存到zipStoreBase下的zipStorePath目录中，这样3.3版本只需要下载一次，以后如果是相同的版本就不需要再下载了。变量GRADLE\_USER\_HOME即电脑上的用户目录，如：C:\Users\EvenDai

假设把上面的版本修改为2.10，然后在命令行执行gradlew build，输出结果如下：





从上面截图可知，gradlew build是执行了一个脚本命令，并不是Gradle任务哦，这个脚本命令首先要下载gradle 2.10，然后解压，然后开启了一个Gradle的后台进程，然后开始执行Gradle tasks（只有开启了Gradle进程，才能执行Gradle tasks）。

可以在命令行运行任何支持的Gradle tasks，包括你自定义的tasks。编译的代码在app/build目录下，生成的apk(Android package)在app/build/apk目录下。

gradlew tasks 命令显示了构建中可以使用的任务，这个输出很长，可查看“可用任务.txt”

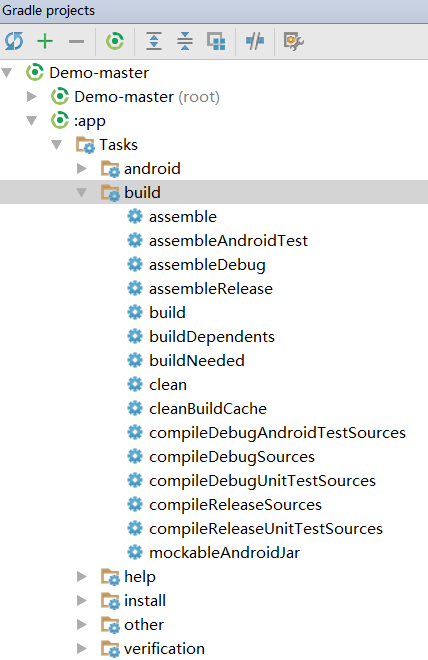
gradlew tasks –all 作了显示可用任务外，也显示每个任务的依赖。

可以使用任务的名字的缩写，只要这个缩写在所有任务中是唯一的即可，如：gradlew anDep表示gradlew androidDependencies。

如果你的构建文件不叫build.gradle，可以使用 –b 标志来指定一个构建文件名：gradlew –b app.gradle

## 1.4 在AndroidStudio中执行Gradle构建

当你创建一个Android项目时，AS产生了用于多项目构建的Gradle构建文件，IDE还提供了一个Gradle视图，其中整理了所有任务，如下：



这里的gradle任务被分类，双击任务即可执行。

运行项目，从Gradle console中查看可知，其实运行本质上只是Gradle的一个前端。任何执行，从构建到测试，再到部署，实际上就是从命令运行Gradle任务。

## 1.5 添加Java库的依赖

Gradle支持几种不同的方式来定义依赖。最常用的方式就是使用引号，并通过冒号分隔开组名、名字和版本号。（Gradle使用Groovy，其即支持单引号字符串，也支持双引号字符串。双引号允许修改，或者变量替换，其他方面是完全一样的。

每个依赖都关联了一个configuration。Android项目包含compile、runtime、testCompile、testRuntime等配置。插件可以添加额外的配置，你也可以定义你自己的。

依赖的完整语法：

testCompile group: ‘junit’, name: ‘junit’, version: ‘4.12’

依赖快捷语法：

testCompile ‘junit:junit:4.12’ 推荐

testCompile ‘junit:junit:4.+’ 这是合法的，但是不推荐使用，指定一个带加号的版本号，这是告诉Gradle任何高于4.0版本的Junit在编译项目的测试是是必须的。虽然这是可以工作的，但是这让构建的可确定性降低，并因此变得更不可重现。显示的版本号也在以后的版本中特定的API的变化中起到保护作用。

如果你想要添加一系列的文件配置中，可以使用files或者fileTree语法，如下：

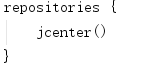
dependencies {

compile files(‘libs/a.jar’, ‘libs/b.jar’)

compile fileTree(dir: ‘libs’, include: ‘\*.jar’)

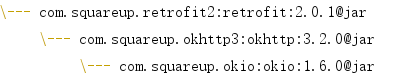
}

compile 'com.android.support:design:25.0.1' ，像这样的依赖，Gradle需要知道去哪里搜索并解析依赖，这通过一个repositories块实现，在项目根目录的build.gradle中指定了这个仓库，如下：



**传递依赖**

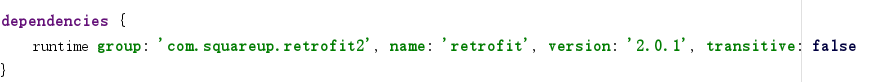
传递依赖，即一个库本身又依赖于另外的库（可能另外的库又依赖另外的库）。比如依赖compile 'com.squareup.retrofit2:retrofit:2.0.1'，这个库会以传递依赖的形式添加okhttp和okio两个库，这两个库在dependencies中是看不到的。如果想查看传递依赖有哪些库，可执行gradlew androidDependencies（注：双击对应的命令看不到相同的结果）：



从这个图中知，retrofit依赖于okhttp，okhttp又依赖于okio。通过查看完整控制台可知：v7依赖于v4和support-vector-drawable和animated-vector-drawable

传递依赖的好处是Gradle很擅长解决依赖之间的版本问题。同时，Gradle也提供了语法来包含或者不包含指定的库。

Gradle默认遵循传递依赖，如果你想要针对一个特定的库关掉这个功能，使用transitive标志，如下：



transitive为false表示阻止传递依赖，则你自己需要手动添加任何需要的传递依赖。

如果你想要依赖一个模块的jar，不引入任何额外的依赖，可以这样：

dependencies {

compile ‘org.codehaus.groovy:groovy-all:2.4.4@jar’ (快捷语法， @符号即表示 ext: ‘jar’)

compile group: ‘ogr.codehaus.groovy’, name: ‘groovy-all’, version: ‘2.4.4’, ext: ‘jar’ (完整语法)

}

你也可以从dependencies块中排除一个传递依赖，如下：

dependencies {

androidTestCompile(‘org.spockframework:spock-core:1.0-groovy-2.4’) {

exclude group: ‘org.codehaus.groovy’

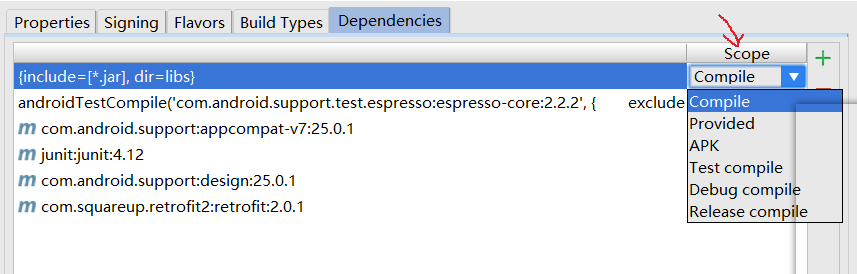
exclude group: ‘junit’

}

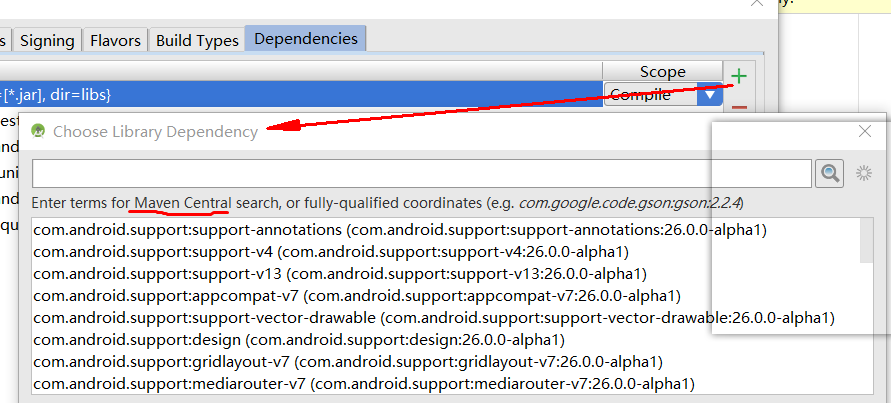
}

这个例子中，spock-core项目并没有包含Groovy依赖和Junit库，须要用其他的方式引入。

## 1.6 使用AS添加库的依赖



Scope列允许你指定哪处配置对依赖有需要。



这里是在Maven中央仓库中搜索，默认显示了所有可选的支持库和Google Play服务。

## 1.7 配置仓库

jcenter() 代表Bintray JCenter仓库，位于：<https://jcenter.bintray.com>

mavenCentral() 代表公共的Maven中央仓库，位于：<http://repo1.maven.org/maven2>

mavenLocal() 代表引用本地的Maven缓存

Maven仓库不止一个，任何的一个Maven仓库都可以使用maven并且跟随一个url块来添加到默认的仓库列表中去，如下：

repositories {

maven {

url ‘http://repo.spring.io/milestone’

}

}

通过密码保护的仓库可以使用一个credentials块，如：

repositories {

maven {

credentials {

username ‘username’

password ‘password’

}

url ‘http://repo.mycompany.com/maven2’

}

}

# 第二章 从项目导入到发布

## 2.1 设置项目属性

Gradle构建文件支持用一个简单的ext语法来定义属性，在这里”ext”代表”extra”。这使得一次定义变量可以全局使用。

下面是<http://androidannotations.org/>项目的示例Gradle构建文件：

ext {

def AAVersion = ‘4.0-SNAPSHOT’ // 将这个改成你希望的版本

}

dependencies {

apt “org.androidannotations:androidannotations:$AAVersion”

compile “org.androidannotations:androidannotations-api:$AAVersion”

}

这里应用的是普通的Groovy俗语，意味着AAVersion这个变量没有类型，但是在赋值时作为一个String。

ext中的没有类型的变量被添加到了构建相关的Project实例上，然而，如果你想要将实际的值挪出构建文件呢？想想看，一个需要登录认证的Maven仓库，我们是不希望把用户名和密码值放在构建文件中的，我们可以把它们保存在gradle.properties文件中，如下：

login=’user’

pass=’my\_long\_and\_highly\_complex\_password’

在build.gradle中配置仓库时可使用上面的变量：

repositories {

maven {

credentials {

username login

password pass

}

url ‘http://repo.mycompany.com/maven2’

}

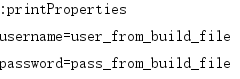
}

使用命令行的方式提供变量，把之前在gradle.properties中定义的变量删除，然后在项目根目录的builde.gradle文件中添加如下代码：

ext {  
 **if** (!project.hasProperty(**'user'**)) { *// 判断属性是否存在* user = **'user\_from\_build\_file'** *// 属性不存在给于默认值* }  
 **if** (!project.hasProperty(**'pass'**)) {  
 pass = **'pass\_from\_build\_file'** }  
}  
  
task printProperties() { *// 定义一个打印变量的属性* doLast {  
 println **"username=**$user**"** println **"password=**$pass**"** }  
}

这里我们从来没有定义过user这个变量，但是可以直接使用。

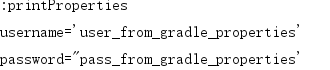
在终端输入：gradlew printProperties，打印结果如下：



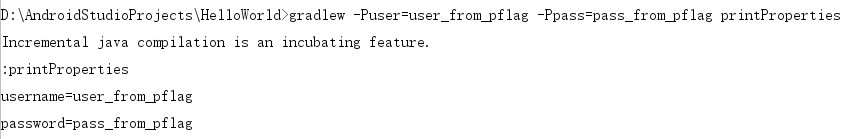
在gradle.properties文件中配置上面的属性：

**user**=**'user\_from\_gradle\_properties'  
pass**=**"pass\_from\_gradle\_properties'**

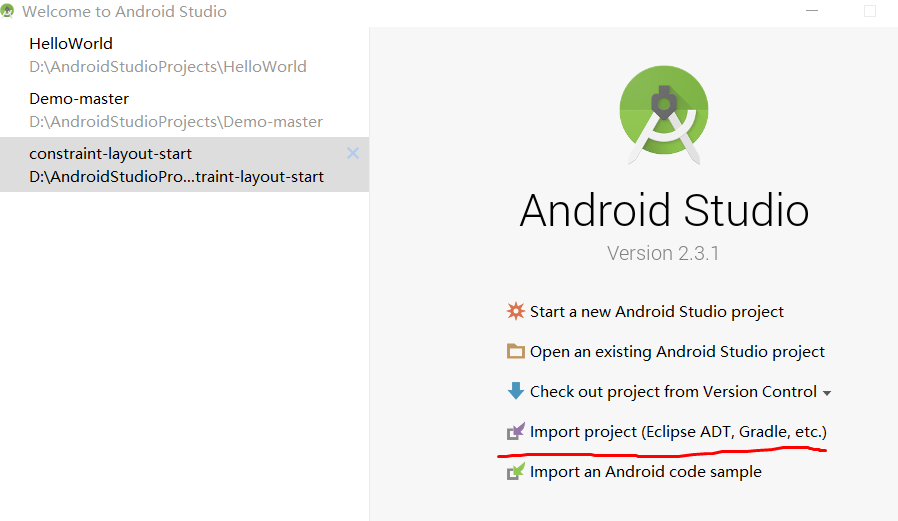
再执行gradlew printProperties任务，打印的结果如下：

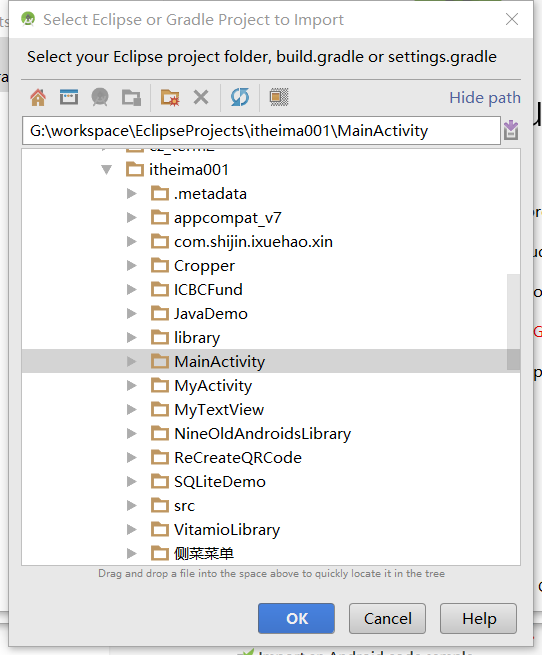


这些值也可以从命令行设置，而且拥有最高优先级：

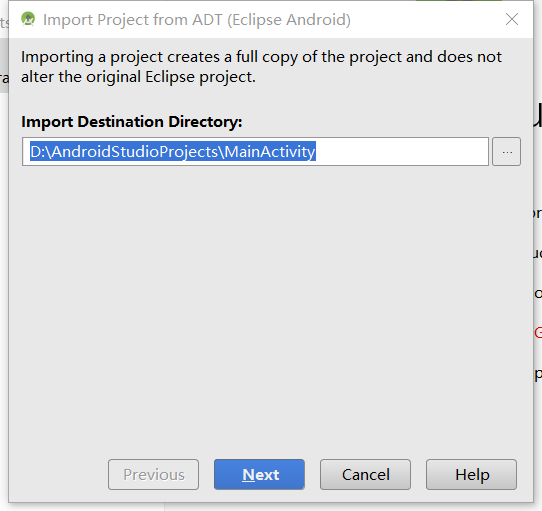


## 2.2 从Eclipse ADT中迁移应用程序到AndroidStudio

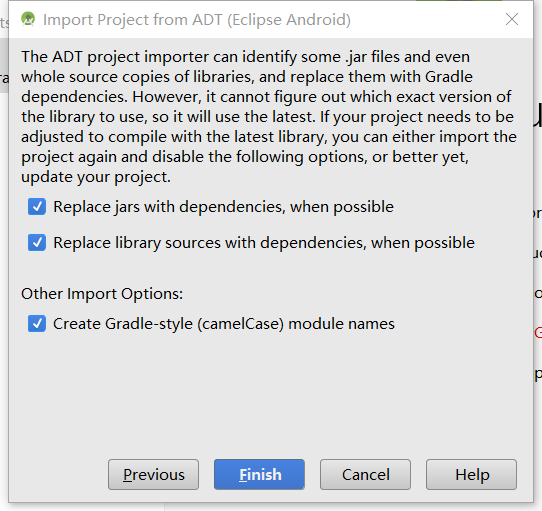




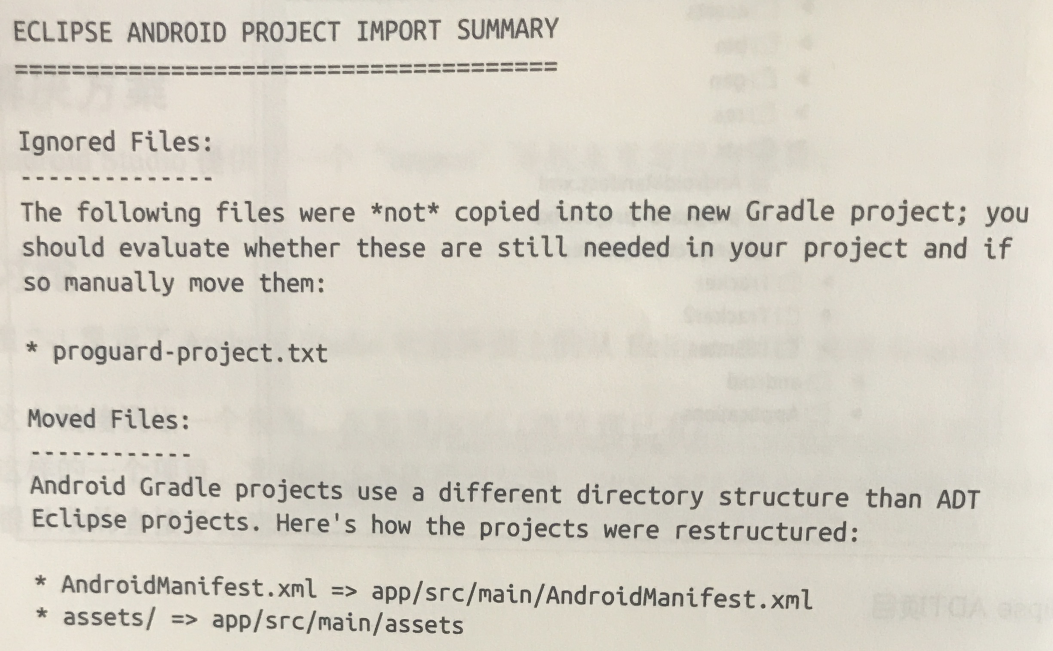
选择要导入的项目,点击OK

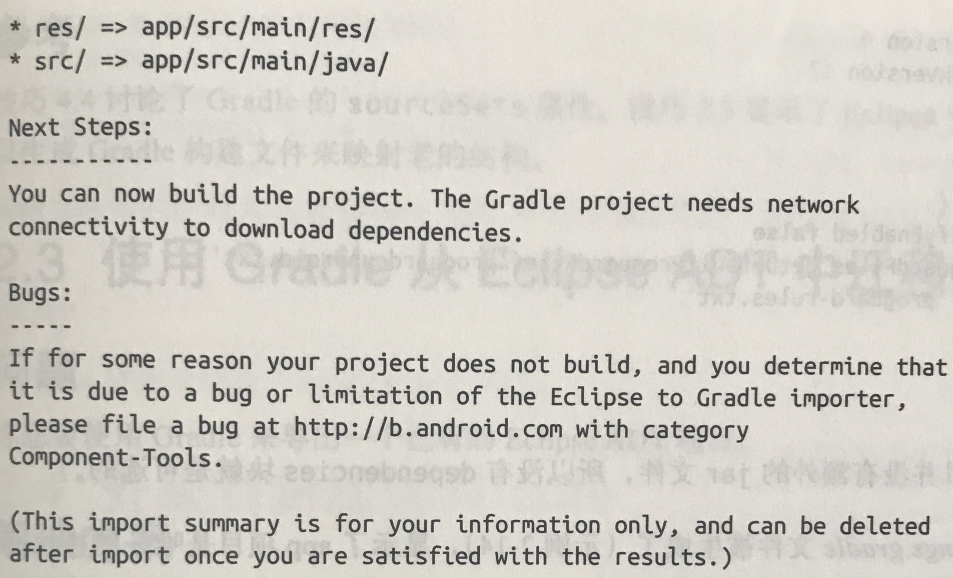


这里用于设置AS项目的保存位置，用于创建导入项目的完整副本，而且不会改变原来的Eclipse项目。



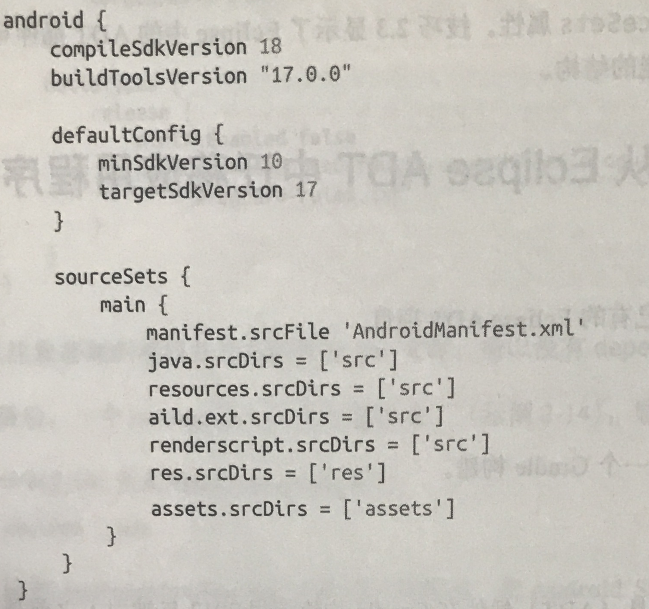
这里提示转换lib目录下的jar文件到Gradle构建文件中的依赖，还有一些其它选项，无需修改，点击Finish按钮，AS会重新组织项目结构，在项目的根目录会生成一个import-summary.txt文件，该文件显示了所有主要的变化：



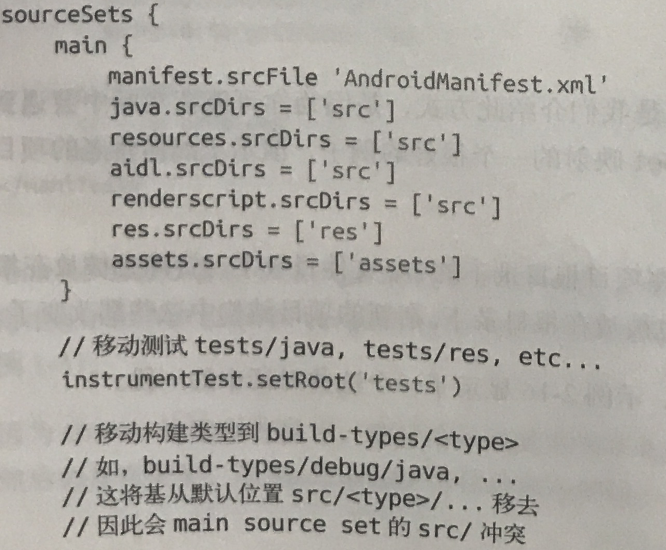


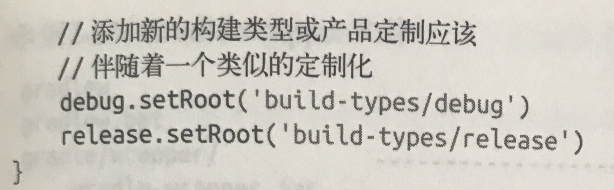
除了关于ProGuard文件的推荐以外，剩下的改动大多是移动文件。

## 2.3 使用Gradle从Eclipse ADT中迁移应用程序

目前直接导入Eclipse项目即可，所以这种方式已经不推荐了。这里还讲这种方式，只是因为这是一个Gradle sourceSet映射的一个很好的例子，ADT插件可以我们生成Gradle构建，如下：  


通常你会看到这种类型的映射也会引起测试目录和构建类型的改变，如下：





构建文件中的注释其实是Eclipse ADT作为转换过程的一部分而添加进去的。

## 2.4 升级到新版本Gradle

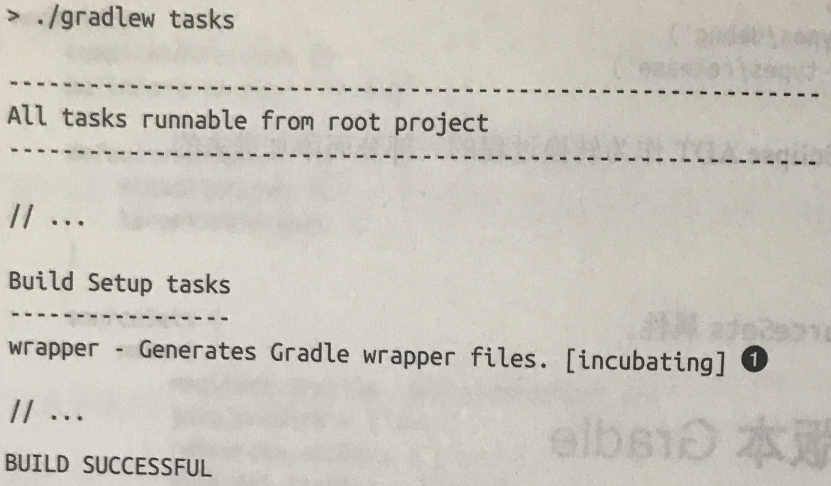
Android Studion包含一个Gradle的发行版本（在AS根目录的gradle目录中）。当你创建一个新的Android应用程序，AS会自动生成gradlew和gradlew.bat文件，这些就是“wrapper”脚本，其允许你省略手动安装的过程，由wrapper脚本为你下载并安装一个Gradle的版本。

两种升级Gradle版本的方式：

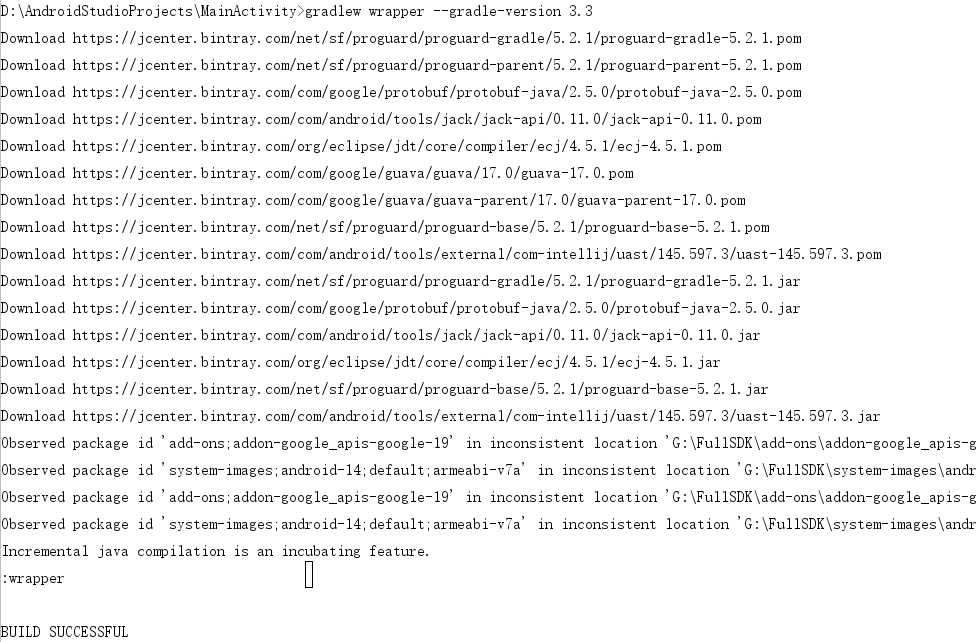
1、添加一个wrapper任务到你的build.gradle文件并生成新的wrapper脚本

2、直接修改gradle-wrapper.properties文件中的distributionUrl的值(这种方式应该会允许你仍然能成功运行已有的wrapper脚本)

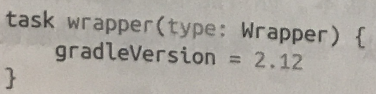
如果你的AS已经有当前版本的Gradle，那么第一个选择就最合适。默认情况下Gradle构建已经包含了一个所谓的wrapper任务，你可以通过运行gradlew tasks命令查看，如下：



gradlew wrapper任务支持一个--gradle-version参数，用于生成所期望版本的wrapper，如下：



另一种方式是：在项目根目录的build.gradle中添加一个wrapper任务，并设置一个gradleVersion的值，如下：

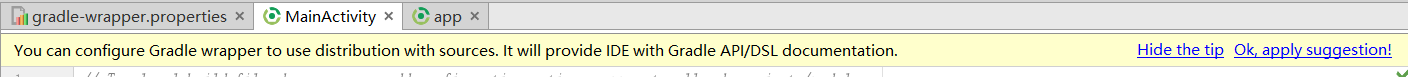


有了这个修改，运行gradlew wrapper任务就会生成一些新的wrapper文件。

不论是在命令行加参数，还是添加wrapper任务为升级Gradle，都仅仅是添加了Gradle的二进制的分发包，没有包含源码，如下：



这时AS会有如下提示：



点击“OK, apply suggestion!” 下载Gradle的完整分发包，此时查看之前的URL地变成：



## 2.5 在项目之间分享设置

问题：你想要去除多个模块中的重复设置

解决方案：在顶层的Gradle构建文件中使用allprojects或者subprojects块。

allprojects {  
 repositories {  
 jcenter()  
 }  
}

这个块来自Gradle的DSL，因此所有基于Gradle的项目都能工作，不只是Android项目。allprojects属性来自于Gradle的Project API，在这个API里它是org.gradle.api.Project 类的一个属性。这个属性由一个集合组成，其中包含了当前项目和所有子项目。有一个同样名字的方法，允许你配置当前项目和所有其子项目。Gradle API中包含同样名字的属性和方法是很常见的，使用的上下文决定了你使用的是方法还是属性。

这段代码的行为就是为allrojects集合返回的所有项目（根目录算一个项目，所有的模块都是项目，模块在Gradle中称为子项目）都应用这里的闭包参数。在这个例子中，简单地意味着你不需要在app模块中重复定义repositories块，因为其已经被设置过了。

另一个做法是使用subproject块。比如，假设你有多个Android库项目，每个项目会在其自己的构建文件中应用library插件。如果所有的子项目都是Android库，你可以通通过在顶层应用这个插件来消除重复：

subprojects {

apply plugin: ‘com.android.library’

}

subprojects返回的项目是不包含根目录项目的。

**高级用法**

如果你在Gradle DSL参考中查看关于Project类的allprojects该当的文档，你会发现这个方法接收一个类型为org.gradle.api.Action的参数，如下：

void allprojects(Action<? user Project> action) 将给定的Action在所有项目上执行。

Action<T>是一个只有一个方法的接口，这个方法叫execute，其接受一个单独的通用参数，所以文档暗示你需要创建一个实现了Action接口的类，然后实例化这个类，用其作为参数。在Java （Java SE 8之前）里面，这通常是使用匿名内部类的方式实现的，如下：

project.allprojects(new Action<Project>() {

void excute(Project p) {

// 做任何你想对这个项目做的事情

}

});

在Groovy中，你可以通过简单地提供一个闲包作为参数来实现一个单一方法的接口。这个闭包会成为这个方法的实现。Gradle实现的allprojects和subprojects方法就是在集合中的每一个项目上调用这个闭包参数。

allprojects {  
 repositories {  
 jcenter()  
 }  
}

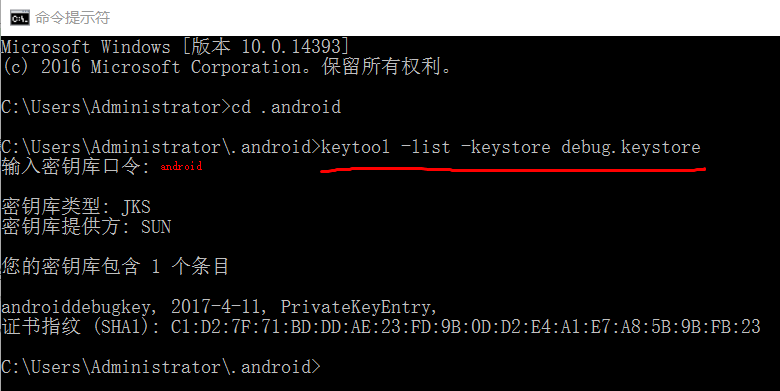
上面的代码提供了一个闭包参数给allprojects方法，其表明配置repositories块来使用jcenter作为其仓库。

注意Java SE 8引入了lambdas，其工作方式类似。Java 8 lambdas可以被赋值给一个所谓的functional interfaces，即只包含一个方法的接口。Groovy在最开始就已经拥有了闭包。Gradle 2.0及以上版本支持Java SE 8。但是Android SDK仍然不能，尽管Android N和Android Studio2.1 有计划支持一些lambda的功能。

## 2.6 为发布apk签名

数字签名证书是使用Java的keytool命令创建的。所有APK文件在部署之前都需要被数字签名。

列举调试密钥库：



密钥库类型是JKS（Java KeyStore），用于公钥和私钥。Java提供另一种类型叫JCEKS（Java Cryptography Extensions KeyStore），其可以被用于共享密钥，但是没有被Android应用程序使用。

这个密钥库中有一个签名的证书，别名为androiddebugkey，当APK被部署到连接的设备或者模拟器上时，其被用于签名调试用的apk。（为了重设调试用的密钥库，可以直接删除debug.keystore文件，下次部署app时会重新创建）

使用命令生成keystore（keytool -genkey -v -keystore myapp.keystore -alias my\_alias -keyalg RSA -keysize 2048 –v alidity 10000），如下：



RSA算法被用于生成公私钥对，大小为2KB，使用SHA256进行签名，10000天（27年多）有效期。

你现在可以使用jarsigner和zipalign工具来为你的apk签名了，但是让Gradle来做会更容量，添加signingConfigs块作为Android闭包的一个子块，如下：

android {  
 signingConfigs {  
 release {  
 keyAlias **'my\_alias'** keyPassword **'123456'** storeFile file(**'C:\\Users\\Administrator\\.android\\myapp.keystore'**)  
 storePassword **'123456'** }  
 }  
}

注：上面的signingConfigs块必须声明在buildTypes块的前面。

上面的alias和密码可以配置到gradle.properties文件中，前面的知识点已经介绍过了。

从DSL文档中，signingConfigs块委托给一个SigningConfig类，其包含4个常用的属性：

keyAlias 当签发一个特定的密钥时在keytool中被使用

keyPassword 在签发过程中使用的一个特定密钥的密码

storeFile 包含密钥和证书的磁盘文件，由keytool生成

storePassword 密钥库文件自身使用的密码。

还有一个storeType属性（默认为JKS），但是这个属性很少使用。为了使用新的配置，添加一个signingConfig属性到release构建类型，如下：

android {

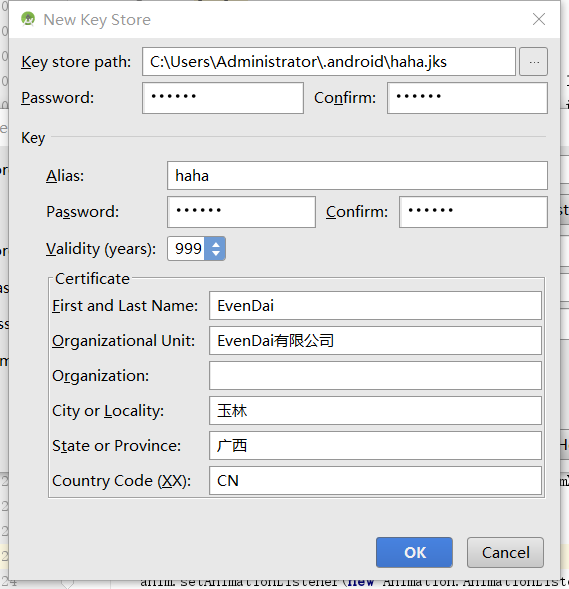
buildTypes {  
 release {  
 minifyEnabled **false** proguardFiles getDefaultProguardFile(**'proguard-android.txt'**), **'proguard-rules.txt'** signingConfig signingConfig.release  
 }  
 }  
}

注：

执行 gradlew assembleRelease任务，构建会在app/build/output/apk目录下生成一个发布版本的apk。

**使用图形界面来生成密钥和密钥配置**

菜单Build🡪Generate Signed APK🡪Create New…



Organizational Unit填写单位名称

Organization填写单位名称

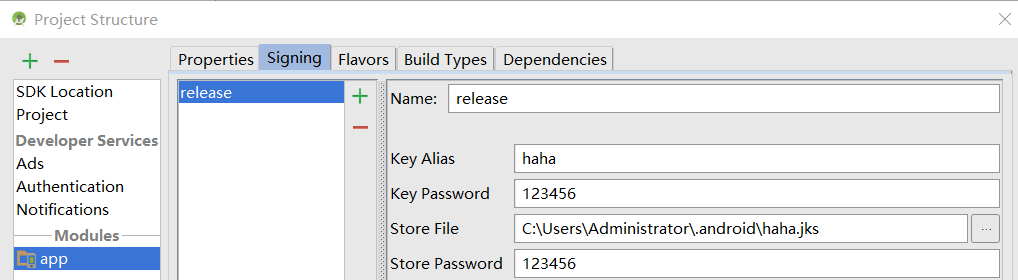
不知道这两都有什么区别，发现可以不写。

City or locality填写城市或区域名称

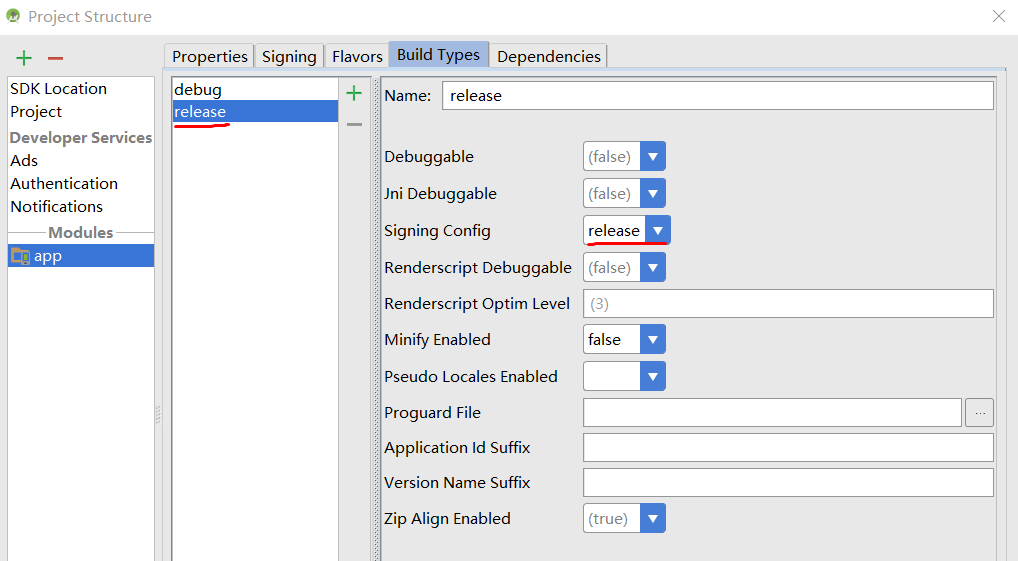
State or Province填写省/市/自治区名称

Contry Code(XX)填写国家/地区代码，这个固定填写CN，不要写86，因为（XX）代表是要两个字母不是数字。

在项目结构中配置keystore

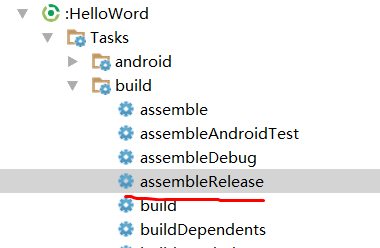


默认这里是没有的，点击加号创建一个。



这里默认就已经存在debug和release了，只需要在releae中的Signing Config中选择我们之前创建的release名称即可。

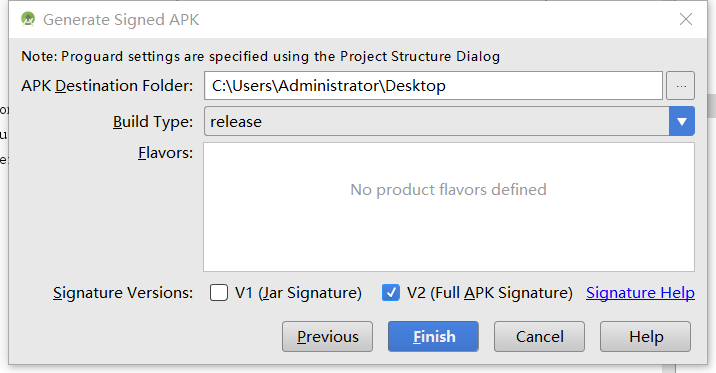
如果要生成release的apk，双击如下任务：



apk位置：build/output/apk/HelloWord-release.apk

这跟执行Build🡪Build Signed APK的区别：

这会弹出对话框，步骤比较多，需要输入keystore密码，在下面的对话框需要选择apk位置，需要选择签名版本



Android 7.0 引入一项新的应用签名方案 APK Signature Scheme v2，它能提供更快的应用安装时间和更多针对未授权 APK 文件更改的保护。在默认情况下，Android Studio 2.2 和 Android Plugin for Gradle 2.2 会使用 APK Signature Scheme v2 和传统签名方案来签署您的应用。

虽然我们建议您对您的应用采用 APK Signature Scheme v2，但这项新方案并非强制性的。如果您的应用在使用 APK Signature Scheme v2 时不能正确开发，您可以停用这项新方案。禁用过程会导致 Android Studio 2.2 和 Android Plugin for Gradle 2.2 仅使用传统签名方案来签署您的应用。要仅用传统方案签署，打开模块级 build.gradle 文件，然后将行 v2SigningEnabled false 添加到您的版本签名配置中：

  android {  
    ...  
    defaultConfig { ... }  
    signingConfigs {  
      release {  
        storeFile file("myreleasekey.keystore")  
        storePassword "password"  
        keyAlias "MyReleaseKey"  
        keyPassword "password"  
        **v2SigningEnabled false**  
      }  
    }  
  }

**注意：**如果您使用 APK Signature Scheme v2 签署您的应用，并对应用进行了进一步更改，则应用的签名将无效。出于这个原因，请在使用 APK Signature Scheme v2 签署您的应用之前、而非之后使用 zipalign 等工具。

如需了解详细信息，请阅读相关的 Android Studio 文档，这些文档介绍了如何在 Android Studio 中[签署应用](https://developer.android.com/studio/publish/app-signing.html#release-mode)以及如何使用 Android Plugin for Gradle [为签署应用配置构建文件](https://developer.android.com/studio/build/build-variants.html#signing)。

# 第三章 构建类型和定制

## 3.1 处理构建类型

问题：你想要自定义调试和发布构建类型，或者创建额外的你自己的构建类型

解决方案：android块内部的buildTypes块可以被用来配置构建类型。

一个构建类型决定app如何被打包。Gradle的Android插件默认支持两种不同类型的构建：debug和release。

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

上面是新项目的构建文件中默认的buildTypes块，如果你还想添加一个debug来配置默认的设置块也同样非常方便。每个块都支持一系列的属性。完整的属性和方法可以在com.Android.build.gradle.internal.BuildType类的DSL参考中找到。在release块中，minifyEnabled 代表在打包app时自动删除无用的资源。如果设置成true，Gradle还会删除无须用到的依赖库。这只会在shrinkResources属性同时设置成true时生效，如下：

buildTypes {  
 release {  
 minifyEnabled **true** *// 打开代码压缩* shrinkResources **true***// 打开资源压缩* proguardFiles getDefaultProguardFile(**'proguard-android.txt'**), **'proguard-rules.pro'** }  
}

更多细节参见“Resource Shrinking” 网页：<https://developer.android.com/studio/build/shrink-code.html>

另一个在构建类型中可用的属性是debuggable。调试构建类型自动设置debuggable为true，其他类型默认为false。

为了能够在一台机器上安装多种构建的类型，Android必须能够分辨出它们的应用程序ID。applicationIDSuffix允许Gradle生成多个apk，每个都有自己的不同ID，如下：

buildTypes {  
 release {  
 minifyEnabled **true** *// 打开代码压缩* shrinkResources **true***// 打开资源压缩* proguardFiles getDefaultProguardFile(**'proguard-android.txt'**), **'proguard-rules.pro'** }  
 debug {  
 applicationIdSuffix **'.debug'** versionNameSuffix **'-debug'** }  
}

这样发布版本和调试版本都可以被部署到同一台设备上了。

## 3.2 产品定制和变种

问题：你想要创建多个apk，本质上你想要构建同一个应用程序，但是使用不同的资源或类

解决方案：产品定制允许你创建多个不同版本的相同的应用程序

定制允许你构建同一个app的多个版本。这可能会发生在当你需要根据不同的客户定制不同的样式和感觉的时候，或者你需要同时拥有一个免费版本和付费版本的同一app。

使用Android闭包的productFlavors块来声明一个产品定制。

创建一个最简单的Android项目，然后在app的build.gradle的android块中增加如下代码：

productFlavors {  
 arrogant{  
 applicationId **'com.example.administrator.hello.arrg'** }  
 friendly{  
 applicationId **'com.example.administrator.hello.frnd'** }  
 obsequious{  
 applicationId **'com.example.administrator.hello.obsq'** }  
}

上面我们定制了3个产品，每个产品都有自己的不同applicationId，当我们在打包apk时，只能打包这3个id的apk，默认的applicationId的apk无法打包了，如果想打包一个默认applicationId的apk产品，可以在这里的productFlavors中再增加一个applicationId和默认的一样即可。因为applicationId不同，所以上面的3个可以同时安装在同一个设备上。

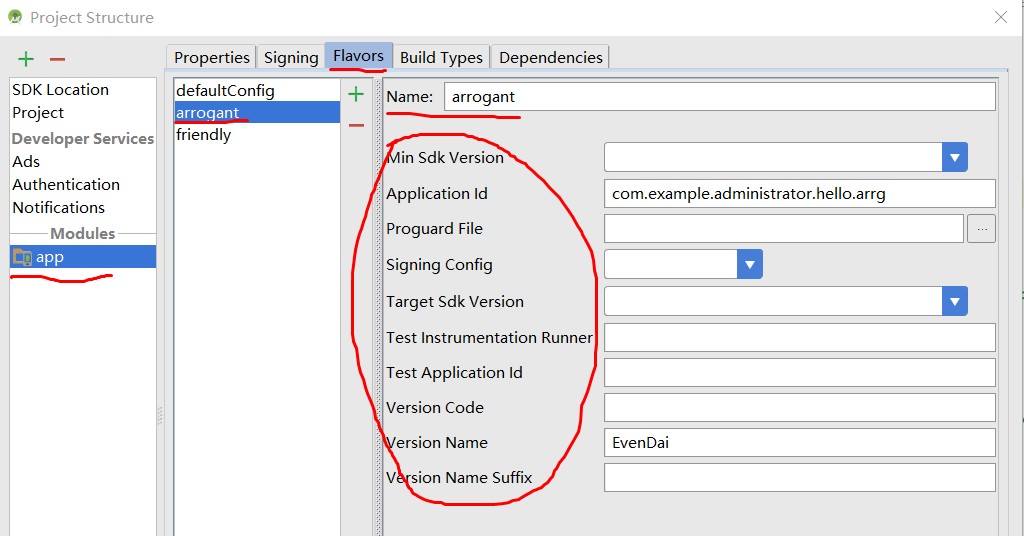
每个产品定制可以拥有其自己的值，如下面的属性，还有一些其他的基于默认的defaultConfig的相同属性。

* applicationId
* minSdkVersion
* targetSdkVersion
* versionCode
* versionName
* sigingConfig

这些属性的定制跟appliationId的定制是一样的，如下：

productFlavors {  
 arrogant{  
 applicationId **'com.example.administrator.hello.arrg'** versionName **'EvenDai-3.14'** }  
}

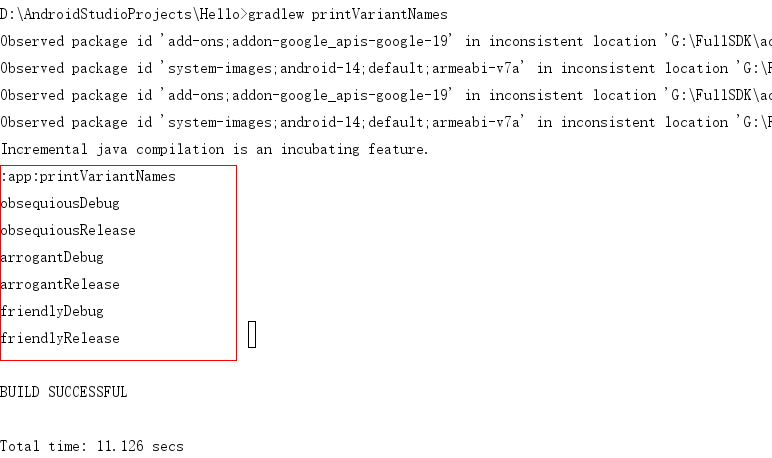
这些属性也都可以通过图形界面设置，如下：



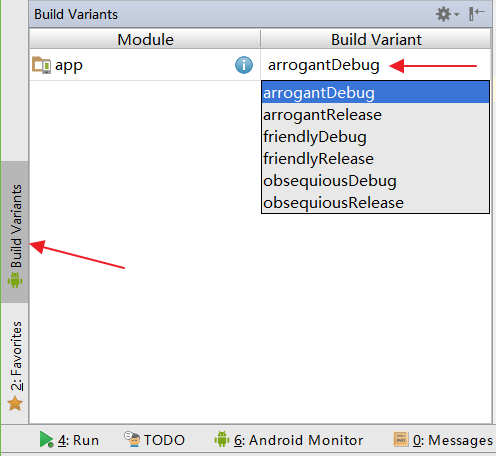
添加打印变种名称的任务，在app的build.gradle文件的根节点添加如下任务：

task printVariantNames {  
 doLast {  
 android.applicationVariants.all { variant ->  
 println variant.name  
 }  
 }  
}

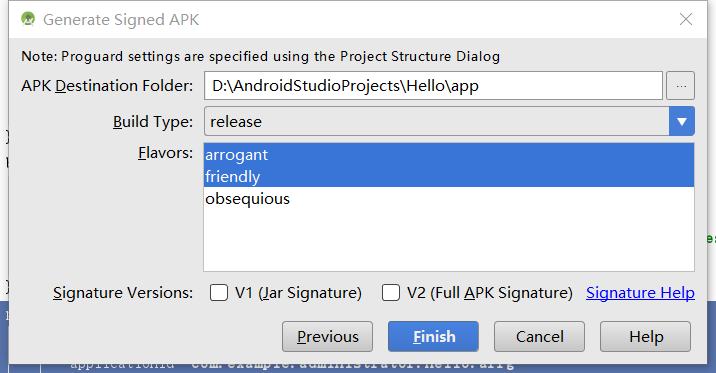
然后在终点执行如下命令：



对于两种默认的构建类型（debug和release）和这里的3个定制(arrogant、friendly、obsequious)组合成了6个不同的变种，当我们直接点击运行时，会打包哪一个变种的apk呢？查看如下：

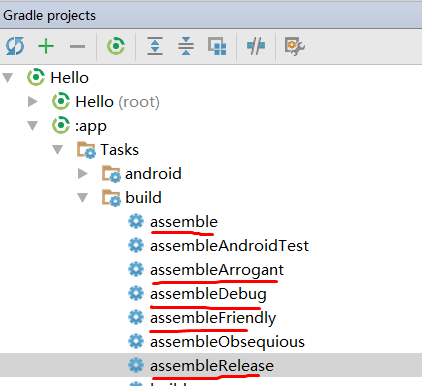
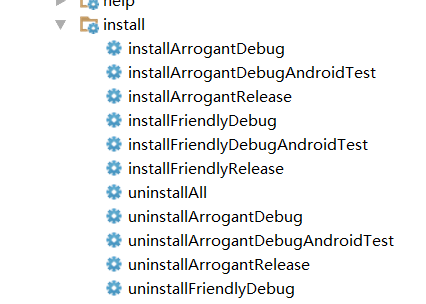


在这个Build Variants窗口中选择需要的构建变种，则默认运行的就是选择的变种。 在打包apk的时候，可以选择：



从上面的对话框我们可以选择构建类型，在Flavors中可以选择一个或多个定制的产品。

在终端执行下面的任务：

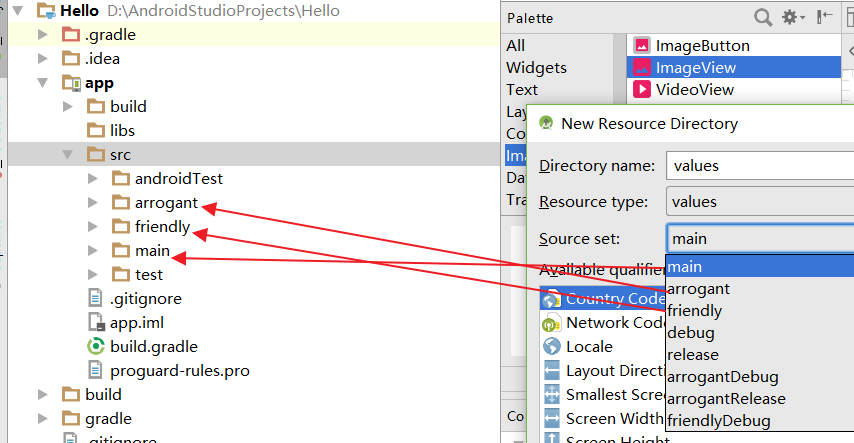
* assemble构建所有可能的的变种，assemble有装配的意思，可以理解组装apk
* gradlew assemble + 构建类型名称，如：gradlew assembleRelease 构建release类型的变种，如果要构建debug，则执行gradlew assembledebug
* gradlew assemble + 产品定制名称，如：gradlew assembleArrogant构建arrogant的debug和release变种
* gradlew assemble + 变种名称，如：gradlew assembleArrogantDebug构建arrogantDebug变种
* 注：上面的assemble后面跟的单词首字母大小写均可
* 上面的任务可以在任务窗口找到，如下：  
    
  这里貌似没有构建具体某个变种名称的任务。
* gradlew install + 变种名称，如：installArrogantDebug （注：install任务只能针对某个具体的变种的），这些命令也能在工具窗口中找到，如下：  
    
  注：如果没有在build.gradle中配置release的keystore相关信息，则执行assemble生成的release类型的apk是没有签名的，则上面的窗口不会有安装Release的命令，如果自己在终端输入类型进行安装，也会提示安装失败，原因就是找不到这样的任务。 没有签名的apk是无法安装到手机上的。

## 3.3 合并资源

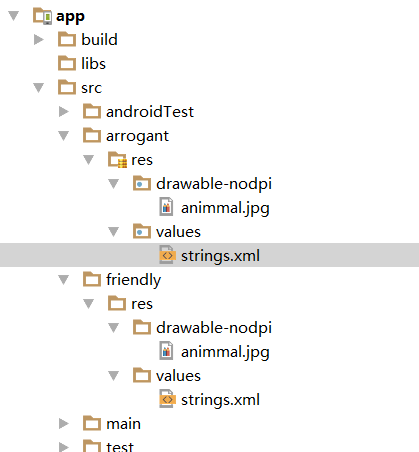
问题：你想要改变一个产品定制的图片、文本或者其它资源

解决方案：给你的定制添加适配的资源目录、添加相应的文件，并修改其包含的值。

可以给不同的变种创建相应的源码集，在src目录上右击🡪New 🡪Android Resource Directory，如下：



可以看到，我们可以为构建类型、产品定制、变种等分别创建SourceSet，然后各自的SourceSet中定义自己的源代码和资源。比如，我们把main中的strings .xml复制到不同的定制目录下，然后修改其String值，把drawable的图片相同名字，但是内容不同分别放到定制里面，如下：



## 3.4 定制维度

问题：一个产品定制还不够，你需要另一个标准类区分你应用程序的不同版本

解决方案：在你的产品定制中添加flavorDimensions属性

假设，不同的客户针对应用程序的每个定制需要有自己品牌特色的版本。本质上每个版本的源码是一样的，只是一些小的资源有不同。为了避免有太多的重复，引入一个额外的定制维度，如下：

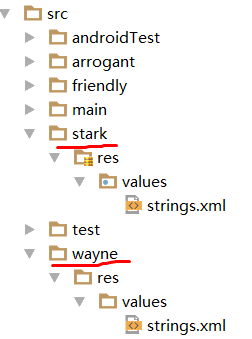
flavorDimensions **'attitude'**, **'client'**productFlavors {  
 arrogant {  
 dimension **'attitude'** applicationId **'com.example.administrator.hello.arrg'** versionName **'EvenDai-3.14'** }  
 friendly {  
 dimension **'attitude'** applicationId **'com.example.administrator.hello.frnd'** }  
 stark {  
 dimension **'client'** }  
 wayne {  
 dimension **'client'** }  
}

这里arrogant、friendly是在attitude维度上的，stark、wayne是在client类型的定制。

两类型种构建类型、两种attitude定制、两种clien定制，一共产生2 \* 2 \* 2 = 8 种不同的变种，如下：



为了让每个client的变种做一些看得到的事情，为每个client定制添加目录绿，如下：



不是很理解这个维度的功能有什么用，从实验结果看，不同维度的定制产品会互相组合，而且组合的名称中，一个维度靠前的名字会在前面，如上面的代码中attitude维度声明的时候是在前面的（flavorDimensions 'attitude', 'client'），而arrogant和friendly属性attitude维度，所以在组合变种的时候，arrogant、friendly的名字会在stark、wayne的前面，上面的截图中，src下有4个目录分别是：friendly、arrogant、stark、wayne，在它们下面都有strings.xml，里面有同样名字的资源，只是资源的值不同，以哪个为准呢？以friendly和arrogant的为准，因为这两个是属于attitude维度，而这个维度在声明的时候靠前。

总的感觉上就是两个产品，每个产品有两套资源，只要把维度的顺序调整一下，则切换资源。

所以上面变种arrogantStartRelease和arrogantWayneRelease这两个apk将会一模一样，没有区别。

当然，如果attitude维度的产品没有的资源，如果在client维度上有，则client上的资源也会覆盖main上的资源，但是这样有什么应用场景呢？

## 3.5 合并不同定制间的Java源代码

问题：你想要为单独的产品定制添加Activity，或者其它Java类。

解决方案：创建合适的源目录，添加Java类，并与main源码集合并。

虽然定制和构建类型中的资源与main中的资源会覆盖，Java类却不同，在构建类型和定制中有的类，则在main中就不能有。而且在一个构建类型或者产品定制中有的类，在其它的构建类型或者产品定制中可以没用，比如friendly、arrogant都有SencondActivity，arrogant中的SencondActivity使用到了一个Util类，而friendly中的SencondActivity并没有使用到Util类，所以，在arrogant中有Util类存在，在friendly中就没什么必要存在了。

# 第4章 自定义任务

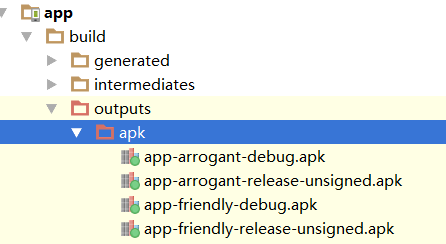
## 4.1 编写自定义任务

Gradle的DSL支持一个task块来定义你自己的任务。API包括很多已经存在任务（如Copy、Wrapper、Exec），这些任务你可以简单地设置些属性然后使用。比如Copy任务包含from和into属性，from块可以配置成不包含某个特定的文件名模式。为了拷贝所有的apk到一个新的文件夹，并且不包含未签名或未对齐的apk，在app的build.gralde的根节点中添加如下代码：

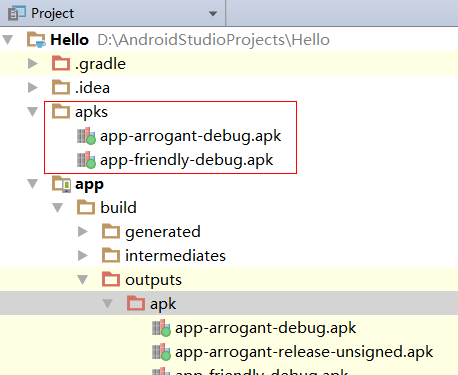
task copyApks(**type**: Copy) {  
 from(**"**$buildDir**/outputs/apk"**) {  
 exclude **'\*\*/\*unsigned.apk'**, **'\*\*/\*unaligned.apk'** }  
 into **'../apks'**}

buildDir属性引用默认的构建目录（app/build），$符号用来将其注入到Groovy的字符串中。文档显示Copy任务的from中exclude块支持Ant风格的目录名，意味着\*\*匹配所有子目录。

有如下apk：



在终端输入：gradlew copyApks，执行后结果如下：



如果你不想简单地配置一个已有的Gradle任务,你需要理解configuration和execution阶段的区别。在configuration阶段，Gradle根据其依赖构建一个DAG，然后执行目标任务，随着依赖一起，所有任务在执行之前先被配置。

Gradle推荐描述性的任务，如上面的Copy任务，你指定了你需要做的事情，而不是如何去做。如果你需要执行命令，添加一个doLast块到你的Gradle任务中，比如下面的示例：

task printVariantNames {  
 doLast {  
 android.applicationVariants.all { variant ->  
 println variant.name  
 }  
 }  
}

这个任务中无论在doLast之前还是之后的所有事情都在配置阶段执行。在doLast块中的代码在运行阶段执行。Android插件添加了一个Android属性，其中包含一个applicationVariants的属性，其返回所有的构建类型或定制组合，在这个例子中，它们都被打印到了终端。

applicationVariants属性只针对com.android.application插件有效。libraryVariants属性只针对Android库项目可用，两者都可以使用testVariants属性。

为了安装所有的调试定制到一台设备，编写下面的任务：

task installDebugFlavors() {  
 android.applicationVariants.all { v ->  
 **if** (v.name.endsWith(**'Debug'**)) {  
 String name = v.name.capitalize()  
 dependsOn **"install**$name**"** }  
 }  
}

在这个例子中，dependsOn属性显示了是配置阶段的一部分，而不是运行阶段。每一个变种名字，如friendlyDebug，都被产字母大写了（FriendlyDebug），（注：这可能是这本书出的时候是首字母大写的，现在已经不是大写了），并且相应地安装任务(installFriendlyDebug）也被添加成installDeubFlavors任务的一个依赖。

运行该任务即可安装所有debug类型apk。

**延长ADP超时时长**

作为一个旁白，虽然构建过程相对来说很快，但是部署过程却未必。Android标签支持一个adbOptions标签来增加在部署过程超时限制之前的所允许的时间，如下：

android {  
  
 adbOptions {  
 timeOutInMs = 30 \* 1000  
 }

}

此处延长超时限制到了30秒，如果你遇到了Shell Command UnresponsiveException失败，你可以调整这个值。

## 4.2 添加自定义任务到构建过程

问题：你想要在全局构建过程中的某部分调用你的自定义任务

解决方案：使用denpendOn属性将任务插入到有向无环图中（Directed Acyclic Graph, DGA）。

在初始化阶段，Gradle根据依赖关系将多个任务组装成一个顺序序列。结果就是一个DGA。比如，Gradle文档构造了Java插件的DAG，如下：

