AndroidGradle 用户指南

极客学院出版

前言

Android Studio 使用 Gradle 构建工具，而 Gradle 继承了强大、灵活的 Ant 和 Maven 丰富的依赖管理，配置管理简单，脚本编写方便灵活，插件模块化。本指南是 Android 官方的 Gradle Plugin User Guide (http://tools.android.com/tech-docs/new-build-system/user-guide) 中文翻译版。

Android Studio 使用 Gradle 构建工具，Eclipse 的 ADT 插件使用的是 Ant 构建工具。因为两个构建工具的区别，导致习惯了 Eclipse 开发环境的开发者刚开始比较难适应 Android Studio。如果要迁移到 Android Studio，建议最好了解下 Gradle 构建工具。Gradle 构建工具是任务驱动型的构建工具，并且可以通过各种 Plugin 插件扩展功能以适应各种构建任务。对应 Android 项目的 Gradle 插件就是 Android Gradle Plugin。

本文档适用于 0.9 版本的 Gradle plugin。由于我们在 1.0 版本之前介绍的不兼容，所以早期版本可能与本文档有所不同。

> 致谢
> 内容撰写：http://blog.csdn.net/qinxiandiqi/article/category/2394347
> 文档整理：https://github.com/AvatarQing/Gradle-Plugin-User-Guide-Chinese-Verision

更新日期	更新内容
2014-04-03	Android 官方 Gradle Plugin User Guide 中文译版发布

目录

1

简介 – Introduction

本文档适用于 0.9 版本的 Gradle plugin。由于我们在 1.0 版本之前介绍的不兼容，所以早期版本可能与本文档有所不同。

Gradle 构建系统的目标

采用 Gradle 作为新构建系统的目标：

- 让重用代码和资源变得更加容易
- 让创建同一应用程序的不同版本变得更加容易，无论是多个 apk 发布版本还是同一个应用的不同定制版本
- 让构建过程变得更加容易配置，扩展和定制。
- 整合优秀的 IDE

Gradle 为什么使用

Gradle 是一个优秀的构建系统和构建工具，它允许通过插件创建自定义的构建逻辑。 我们基于 Gradle 以下的一些特点而选择了它：

- 采用了 Domain Specific Language(DSL 语言) 来描述和控制构建逻辑。
- 构建文件基于 Groovy，并且允许通过混合声明 DSL 元素和使用代码来控制 DSL 元素以控制自定义的构建逻辑。
- 支持 Maven 或者 Ivy 的依赖管理。
- 非常灵活。允许使用最好的实现，但是不会强制实现的方式。
- 插件可以提供自己的 DSL 和 API 以供构建文件使用。
- 良好的 API 工具供 IDE 集成。

2

要求 – Requirements

- Gradle 1.10 或者 Gradle 1.11 或者 Gradle 1.12，并使用 0.11.1 插件版本。
- SDK build tools 要求版本 19.0.0。一些新的特征可能需要更高版本。

3

基本项目 – Basic Project

一个Gradle项目的构建过程定义在build.gradle文件中，位于项目的根目录下。

简单的构建文件

一个最简单的Gradle纯Java项目的build.gradle文件包含以下内容：

```
apply plugin: 'java'
```

这里引入了Gradle的Java插件。这个插件提供了所有构建和测试Java应用程序所需要的东西。

最简单的Android项目的build.gradle文件包含以下内容：

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:0.11.1'
    }
}

apply plugin: 'android'

android {
    compileSdkVersion 19
    buildToolsVersion "19.0.0"
}
```

这里包括了Android build file的3个主要部分：

`buildscrip{...}` 这里配置了驱动构建过程的代码。

在这个部分，它声明了使用Maven仓库，并且声明了一个maven文件的依赖路径。这个文件就是包含了0.11.1版本android gradle插件的库。

> 注意：这里的配置只影响控制构建过程的代码，不影响项目源代码。项目本身需要声明自己的仓库和依赖关系，稍后将会提到这部分。

接下来，跟前面提到的Java Plugin一样添加了 `android` plugin。

最后， `andorid{...}` 配置了所有android构建过程需要的参数。这里也是Android DSL的入口点。

默认情况下，只需要配置目标编译SDK版本和编译工具版本，即 compileSdkVersion 和 buildToolsVersion 属性。 这个 complieSdkVersion 属性相当于旧构建系统中project.properites文件中的 target 属性。这个新的属性可以跟旧的 target 属性一样指定一个int或者String类型的值。

> 重要：
> 你只能添加 android plugin。同时添加 java plugin会导致构建错误。
>
> 注意：
> 你同样需要在相同路径下添加一个local.properties文件，并使用sdk.dir属性来设置SDK路径。 另外，你也可以通过设置 ANDROID_HOME 环境变量，这两种方式没有什么不同，根据你自己的喜好选择其中一种设置。

项目结构

上面提到的基本的构建文件需要一个默认的文件夹结构。Gradle遵循约定优先于配置的概念，在可能的情况尽可能提供合理的默认配置参数。

基本的项目开始于两个名为“source sets”的组件，即main source code和test code。它们分别位于：

- src/main/
- src/androidTest/

里面每一个存在的文件夹对应相应的源组件。 对于Java plugin和Android plugin来说，它们的Java源代码和资源文件路径如下：

- java/
- resources/

但对于Android plugin来说，它还拥有以下特有的文件和文件夹结构：

- AndroidManifest.xml
- res/
- assets/
- aidl/
- rs/
- jni/

> 注意：
> src/androidTest/AndroidManifest.xml是不需要的，它会自动被创建。

配置结构

当默认的项目结构不适用的时候，你可能需要去配置它。根据Gradle文档，重新为Java项目配置*sourceSets*可以使用以下方法：

```
sourceSets {
    main {
        java {
            srcDir 'src/java'
        }
        resources {
            srcDir 'src/resources'
        }
    }
}
```

> 注意：
> `srcDir` 将会被添加到指定的已存在的源文件夹中（这在Gradle文档中没有提到，但是实际上确实会这样执行）。

替换默认的源代码文件夹，你可能想要使用能够传入一个路径数组的 `srcDirs` 来替换单一的 `srcDir` 。以下是使用调用对象的另一种不同方法：

```
sourceSets {
    main.java.srcDirs = ['src/java']
    main.resources.srcDirs = ['src/resources']
}
```

想要获取更多信息，可以参考Gradle文档中关于Java Pluign (http://gradle.org/docs/current/userguide/java_plugin.html) 的部分。

Android Plugin使用的是类似的语法。但是由于它使用的是自己的*sourceSets*，这些配置将会被添加在 `android` 对象中。

以下是一个示例，它使用了旧项目结构中的main源码，并且将 `androidTest` *sourceSet*组件重新映射到*tests*文件夹。

```
android {
    sourceSets {
```

```
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            resources.srcDirs = ['src']
            aidl.srcDirs = ['src']
            renderscript.srcDirs = ['src']
            res.srcDirs = ['res']
            assets.srcDirs = ['assets']
        }

        androidTest.setRoot('tests')
    }
}
```

> 注意：
> 由于旧的项目结构将所有的源文件（java,aidl,renderscripthe和java资源文件）都放在同一个目录里面，所以我们需要将这些*sourceSet*组件重新映射到 `src` 目录下。
>
> 注意：
> `setRoot()` 方法将移动整个组件（包括它的子文件夹）到一个新的文件夹。示例中将会移动 `src/androidTest/*` 到 `tests/*` 下。
> 以上这些是Android特有的，如果配置在Java的*sourceSets*里面将不会有作用。

以上也是将旧构建系统项目迁移到新构建系统需要做的迁移工作。

构建任务

通用任务

添加一个插件到构建文件中将会自动创建一系列构建任务(build tasks)去执行（注：gradle属于任务驱动型构建工具，它的构建过程是基于Task的）。Java plugin和Android plugin都会创建以下task：

- `assemble`
  这个task将会组合项目的所有输出。
- `check`
  这个task将会执行所有检查。
- `build`
  这个task将会执行assemble和check两个task的所有工作

- `clean`
  这个task将会清空项目的输出。

实际上`assemble`，`check`，`build`这三个task不做任何事情。它们只是一个Task标志，用来告诉android plugin添加实际需要执行的task去完成这些工作。

这就允许你去调用相同的task，而不需要考虑当前是什么类型的项目，或者当前项目添加了什么plugin。例如，添加了*findbugs* plugin将会创建一个新的task并且让`check` task依赖于这个新的task。当`check` task被调用的时候，这个新的task将会先被调用。

在命令行环境中，你可以执行以下命令来获取更多高级别的task：

```
gradle tasks
```

查看所有task列表和它们之间的依赖关系可以执行以下命令：

```
gradle tasks --all
```

> 注意：
> Gradle会自动监视一个task声明的所有输入和输出。
> 两次执行`build` task并且期间项目没有任何改动，gradle将会使用UP-TO-DATE通知所有task。这意味着第二次build执行的时候不会请求任何task执行。这允许task之间互相依赖，而不会导致不需要的构建请求被执行。

Java 项目的 Task

Java plugin主要创建了两个task，依赖于main task（一个标识性的task）：

- `assemble`
  - `jar`
    这个task创建所有输出
- `check`
  - `test`
    这个task执行所有的测试。

`jar` task自身直接或者间接依赖于其他task：`classes` task将会被调用于编译java源码。
`testClasses` task用于编译测试，但是它很少被调用，因为`test` task依赖于它（类似于`classes` task）。

通常情况下，你只需要调用到`assemble`和`check`，不需要其他task。

你可以在Gradle文档中查看java plugin (http://gradle.org/docs/current/userguide/java_plugin.html) 的全部task。

Android 任务

Android plugin使用相同的约定以兼容其他插件，并且附加了自己的标识性task，包括：

- `assemble`
  这个task用于组合项目中的所有输出。
- `check`
  这个task用于执行所有检查。
- `connectedCheck`
  这个task将会在一个指定的设备或者模拟器上执行检查，它们可以同时在所有连接的设备上执行。
- `deviceCheck`
  通过APIs连接远程设备来执行检查，这是在CL服务器上使用的。
- `build`
  这个task执行assemble和check的所有工作。
- `clean`
  这个task清空项目的所有输出。

这些新的标识性task是必须的，以保证能够在没有设备连接的情况下执行定期检查。 注意 `build` task不依赖于 `deviceCheck` 或者 `connectedCheck` 。

一个Android项目至少拥有两个输出：debug APK（调试版APK)和release APK（发布版APK）。每一个输出都拥有自己的标识性task以便能够单独构建它们。

- `assemble`
  - `assembleDebug`
  - `assembleRelease`

它们都依赖于其它一些tasks以完成构建一个APK需要多个步骤。其中assemble task依赖于这两个task，所以执行 `assemble` 将会同时构建出两个APK。

> 小提示：
> gradle在命令行终端上支持骆驼命名法的task简称，例如，执行
> `gradle aR`
> 命令等同于执行
> `gradle assembleRelease`

check task也拥有自己的依赖：

- `check`
  - `lint`
- `connectedCheck`
  - `connectedAndroidTest`
  - `connectedUiAutomatorTest`（目前还没有应用到）
- `deviceCheck`
  - 这个test依赖于test创建时，其它实现测试扩展点的插件。

最后，只要task能够被安装（那些要求签名的task），android plugin就会为所有构建类型（`debug`，`release`，`test`）安装或者卸载。

基本的构建定制

Android plugin提供了大量DSL用于直接从构建系统定制大部分事情。

Manifest 属性

通过SDL可以配置一下manifest选项：

- minSdkVersion
- targetSdkVersion
- versionName
- applicationId (有效的包名 -- 更多详情请查阅ApplicationId 对比 PackageName (http://tools.android.com/tech-docs/new-build-system/applicationid-vs-packagename))
- package Name for the test application
- Instrumentation test runner

例如：

```
android {
    compileSdkVersion 19
    buildToolsVersion "19.0.0"
```

```
    defaultConfig {
        versionCode 12
        versionName "2.0"
        minSdkVersion 16
        targetSdkVersion 16
    }
}
```

在 android 元素中的 defaultConfig 元素中定义所有配置。

之前的Android Plugin版本使用packageName来配置manifest文件中的packageName属性。从0.11.0版本开始，你需要在build.gradle文件中使用applicationId来配置manifest文件中的packageName属性。 这是为了消除应用程序的packageName（也是程序的ID）和java包名所引起的混乱。

在构建文件中定义的强大之处在于它是动态的。 例如，可以从一个文件中或者其它自定义的逻辑代码中读取版本信息：

```
def computeVersionName() {
    ...
}

android {
    compileSdkVersion 19
    buildToolsVersion "19.0.0"

    defaultConfig {
        versionCode 12
        versionName computeVersionName()
        minSdkVersion 16
        targetSdkVersion 16
    }
}
```

> 注意：
> 不要使用与在给定范围内的getter方法可能引起冲突的方法名。例如，在defaultConfig{...}中调用getVersionName()将会自动调用defaultConfig.getVersionName()方法，你自定义的getVersionName()方法就被取代掉了。

如果一个属性没有使用DSL进行设置，一些默认的属性值将会被使用。以下表格是可能使用到的值：

Property Name	Default value in DSL object	Default value
versionCode	-1	value from manifest if present

Property Name	Default value in DSL object	Default value
versionName	null	value from manifest if present
minSdkVersion	-1	value from manifest if present
targetSdkVersion	-1	value from manifest if present
applicationId	null	value from manifest if present
testApplicationId	null	applicationId + “.test”
testInstrumentationRunner	null	android.test.InstrumentationTestRunner
signingConfig	null	null
proguardFile	N/A (set only)	N/A (set only)
proguardFiles	N/A (set only)	N/A (set only)

如果你在构建脚本中使用自定义代码逻辑请求这些属性，那么第二列的值将非常重要。例如，你可能会写：

```
if (android.defaultConfig.testInstrumentationRunner == null) {
    // assign a better default...
}
```

如果这个值一直保持null，那么在构建执行期间将会实际替换成第三列的默认值。但是在DSL元素中并没有包含这个默认值，所以，你无法查询到这个值。
除非是真的需要，这是为了预防解析应用的manifest文件。

构建类型

默认情况下，Android Plugin会自动给项目设置同时构建应用程序的debug和release版本。 两个版本之间的不同主要围绕着能否在一个安全设备上调试，以及APK如何签名。

Debug版本采用使用通用的name/password键值对自动创建的数字证书进行签名，以防止构建过程中出现请求信息。Release版本在构建过程中没有签名，需要稍后再签名。

这些配置通过一个 `BuildType` 对象来配置。默认情况下，这两个实例都会被创建，分别是一个 `debug` 版本和一个 `release` 版本。

Android plugin允许像创建其他构建类型一样定制 `debug` 和 `release` 实例。这需要在 `buildTypes` 的DSL容器中配置：

```
android {
    buildTypes {
        debug {
            applicationIdSuffix ".debug"
```

```
        }

        jnidebug.initWith(buildTypes.debug)
        jnidebug {
            packageNameSuffix ".jnidebug"
            jnidebugBuild true
        }
    }
}
```

以上代码片段实现了以下功能：

- 配置默认的 `debug` 构建类型
    - 将debug版本的包名设置为.debug以便能够同时在一台设备上安装*debug*和*release*版本的apk。
- 创建了一个名为 `jnidebug` 的新构建类型，并且这个构建类型是 `debug` 构建类型的一个副本。
- 继续配置 `jnidebug` 构建类型，允许使用JNI组件，并且也添加了不一样的包名后缀。

创建一个新的构建类型就是简单的在 `buildType` 标签下添加一个新的元素，并且可以使用 `initWith()` 或者直接使用闭包来配置它。

以下是一些可能使用到的属性和默认值：

Property name	Default values for debug	Default values for release / other
debuggable	true	false
jniDebugBuild	false	false
renderscriptDebugBuild	false	false
renderscriptOptimLevel	3	3
applicationIdSuffix	null	null
versionNameSuffix	null	null
signingConfig	android.signingConfigs.debug	null
zipAlign	false	true
runProguard	false	false
proguardFile	N/A (set only)	N/A (set only)
proguardFiles	N/A (set only)	N/A (set only)

除了以上属性之外，*Build Type*还会受项目源码和资源影响： 对于每一个Build Type都会自动创建一个匹配的*sourceSet*。默认的路径为：

```
src/<buildtypename>/
```

这意味着*BuildType*名称不能是*main*或者*androidTest*（因为这两个是由plugin强制实现的），并且他们互相之间都必须是唯一的。

跟其他sourceSet设置一样，Build Type的source set路径可以重新被定向：

```
android {
    sourceSets.jnidebug.setRoot('foo/jnidebug')
}
```

另外，每一个*Build Type*都会创建一个新的assemble任务。

`assembleDebug` 和 `assembleRelease` 两个Task在上面已经提到过，这里要讲这两个Task从哪里被创建。当 `debug` 和 `release` 构建类型被预创建的时候，它们的tasks就会自动创建对应的这个两个Task。

上面提到的build.gradle代码片段中也会实现 `assembleJnidebug` task，并且 `assemble` 会像依赖于 `assembleDebug` 和 `assembleRelease` 一样依赖于 `assembleJnidebug` 。

提示：你可以在终端下输入gradle aJ去运行 `assembleJnidebug task` 。

可能会使用到的情况：

- release模式不需要，只有debug模式下才使用到的权限
- 自定义的debug实现
- 为debug模式使用不同的资源（例如当资源的值由绑定的证书决定）

*BuildType*的代码和资源通过以下方式被使用：

- manifest将被混合进app的manifest
- 代码行为只是另一个资源文件夹
- 资源将叠加到main的资源中，并替换已存在的资源。

签名配置

对一个应用程序签名需要以下：

- 一个Keystory
- 一个keystory密码
- 一个key的别名
- 一个key的密码

- 存储类型

位置，键名，两个密码，还有存储类型一起形成了签名配置。

默认情况下， `debug` 被配置成使用一个debug keystory。 debug keystory使用了默认的密码和默认key及默认的key密码。 debug keystory的位置在$HOME/.android/debug.keystroe，如果对应位置不存在这个文件将会自动创建一个。

`debug` *Build Type*(构建类型) 会自动使用 `debug` *SigningConfig* (签名配置)。

可以创建其他配置或者自定义内建的默认配置。通过 `signingConfigs` 这个DSL容器来配置：

```
android {
    signingConfigs {
        debug {
            storeFile file("debug.keystore")
        }

        myConfig {
            storeFile file("other.keystore")
            storePassword "android"
            keyAlias "androiddebugkey"
            keyPassword "android"
        }
    }

    buildTypes {
        foo {
            debuggable true
            jniDebugBuild true
            signingConfig signingConfigs.myConfig
        }
    }
}
```

以上代码片段修改debug keystory的路径到项目的根目录下。在这个例子中，这将自动影响其他使用到 `debug` 构建类型的构建类型。

这里也创建了一个新的 `Single Config` （签名配置）和一个使用这个新签名配置的新的 `Build Type` （构建类型）。

> 注意：只有默认路径下的debug keystory不存在时会被自动创建。更改debug keystory的路径并不会自动在新路径下创建debug keystory。如果创建一个新的不同名字的*SignConfig*，但是使用默认的debug keystore

> 路径来创建一个非默认的名字的SigningConing，那么还是会在默认路径下创建debug keystory。换句话说，会不会自动创建是根据keystory的路径来判断，而不是配置的名称。
>
> 注意：虽然经常使用项目根目录的相对路径作为keystore的路径，但是也可以使用绝对路径，尽管这并不推荐（除了自动创建出来的debug keystore）。
>
> 注意：如果你将这些文件添加到版本控制工具中，你可能不希望将密码直接写到这些文件中。下面Stack Overflow链接提供从控制台或者环境变量中获取密码的方法：
> http://stackoverflow.com/questions/18328730/how-to-create-a-release-signed-apk-file-using-gradle
> 我们以后还会在这个指南中添加更多的详细信息。

运行 Proguard

从Gradle Plugin for ProGuard version 4.10之后就开始支持ProGuard。ProGuard插件是自动添加进来的。如果*Build Type*的*runProguard*属性被设置为true，对应的task将会自动创建。

```
android {
    buildTypes {
        release {
            runProguard true
            proguardFile getDefaultProguardFile('proguard-android.txt')
        }
    }

    productFlavors {
        flavor1 {
        }
        flavor2 {
            proguardFile 'some-other-rules.txt'
        }
    }
}
```

发布版本将会使用它的Build Type中声明的规则文件，product flavor（定制的产品版本）将会使用对应flavor中声明的规则文件。

这里有两个默认的规则文件：

- proguard-android.txt
- proguard-android-optimize.txt

这两个文件都在SDK的路径下。使用*getDefaultProguardFile()*可以获取这些文件的完整路径。它们除了是否要进行优化之外，其它都是相同的。

精简资源

你能够在编译的时候自动化地移除不用的资源。更多的信息，请看文档 Resource Shrinking (http://tools.android.com/tech-docs/new-build-system/resource-shrinking)

4

依赖关系，Android 库和多项目设置 - Dependencies，Android Libraries and Multi-project setup

Gradle项目可以依赖于其它组件。这些组件可以是外部二进制包，或者是其它的Gradle项目。

依赖二进制包

本地包

配置一个外部库的jar包依赖，你需要在 compile 配置中添加一个依赖。

```
dependencies {
    compile files('libs/foo.jar')
}

android {
    ...
}
```

> 注意：
> 这个 dependencies DSL标签是标准Gradle API中的一部分，所以它不属于 android 标签。

这个 compile 配置将被用于编译main application。它里面的所有东西都被会被添加到编译的classpath中，同时也会被打包进最终的APK。 以下是添加依赖时可能用到的其它一些配置选项：

- compile main application（主module）。
- androidTestCompile test application（测试module）。
- debugCompile debug Build Type（debug类型的编译）。
- releaseCompile release Build Type（发布类型的编译）。

因为没有可能去构建一个没有关联任何*Build Type*（构建类型）的APK，APK默认配置了两个或两个以上的编译配置： compile 和< buildtype >Compile. 创建一个新的*Build Type*将会自动创建一个基于它名字的新配置。

这对于debug版本需要使用一个自定义库（为了反馈实例化的崩溃信息等）但发布版本不需要，或者它们依赖于同一个库的不同版本时会非常有用。

远程文件

Gradle支持从Maven或者Ivy仓库中拉取文件。

首先必须将仓库添加到列表中，然后必须在依赖中声明Maven或者Ivy声明的文件。

```
repositories {
    mavenCentral()
}


dependencies {
    compile 'com.google.guava:guava:11.0.2'
}

android {
    ...
}
```

> 注意：
> `mavenCentral()` 是指定仓库URL的简单方法。Gradle支持远程和本地仓库。
>
> 注意：
> Gradle会遵循依赖关系的传递性。这意味着如果一个依赖本身依赖于其它东西，这些东西也会一并被拉取回来。

更多关于设置依赖关系的信息，请参考 Gradle 用户指南 (http://gradle.org/docs/current/userguide/artifact_dependencies_tutorial.html) 和 DSL (http://gradle.org/docs/current/dsl/org.gradle.api.artifacts.dsl.DependencyHandler.html) 文档。

多项目设置

Gradle项目也可以通过使用多项目配置依赖于其它Gradle项目。

多项目配置的实现通常是在一个根项目路径下将所有项目作为子文件夹包含进去。

例如，给定以下项目结构：

```
MyProject/
 + app/
 + libraries/
    + lib1/
    + lib2/
```

我们可以定义3个项目。Grand将会按照以下名字映射它们：

```
:app
:libraries:lib1
:libraries:lib2
```

每一个项目都拥有自己的build.gradle文件来声明自己如何构建。 另外，在根目录下还有一个*setting.gradle*文件用于声明所有项目。 这些文件的结构如下：

```
MyProject/
 | settings.gradle
 + app/
   | build.gradle
 + libraries/
   + lib1/
     | build.gradle
   + lib2/
     | build.gradle
```

其中setting.gradle的内容非常简单：

```
include ':app', ':libraries:lib1', ':libraries:lib2'
```

这里定义了哪一个文件夹才是真正的Gradle项目。

其中 `:app` 项目可能依赖于这些库，这是通过以下依赖配置声明的：

```
dependencies {
   compile project(':libraries:lib1')
}
```

更多关于多项目配置的信息请参考 这里 (http://gradle.org/docs/current/userguide/multi_project_builds.html) 。

库项目

在上面的多项目配置中， `:libraries:lib1` 和 `:libraries:lib2` 可能是一个Java项目，并且 `:app` 这个Android项目将会使用它们的jar包输出。

但是，如果你想要共享代码来访问Android API或者使用Android样式的资源，那么这些库就不能是通常的Java项目，而应该是Android库项目。

创建一个库项目

一个库项目与通常的Android项目非常类似，只是有一点小区别。

尽管构建库项目不同于构建应用程序，它们使用了不同的plugin。但是在内部这些plugin共享了大部分相同的代码，并且它们都由相同的com.android.tools.build.gradle.jar提供。

```
buildscript {

    repositories {
        mavenCentral()
    }


    dependencies {
        classpath 'com.android.tools.build:gradle:0.5.6'
    }

}

apply plugin: 'android-library'

android {
    compileSdkVersion 15
}
```

这里创建了一个使用API 15编译*SourceSet*的库项目，并且依赖关系的配置方法与应用程序项目的配置方法一样，同样也支持自定义配置。

普通项目和库项目之间的区别

一个库项目的main输出是一个.aar包（它代表Android的归档文件）。它组合了编译代码（例如jar包或者是本地的.so文件）和资源（manifest，res，assets）。一个库项目同样也可以独立于应用程序生成一个测试用的apk来测试。

标识Task同样适用于库项目（`assembleDebug`，`assembleRelease`），因此在命令行上与构建一个项目没有什么不同。

其余的部分，库项目与应用程序项目一样。它们都拥有build type和product flavor，也可以生成多个aar版本。记住大部分Build Type的配置不适用于库项目。但是你可以根据库项目是否被其它项目使用或者是否用来测试来使用自定义的sourceSet改变库项目的内容。

引用一个库项目

引用一个库项目的方法与引用其它项目的方法一样：

```
dependencies {
    compile project(':libraries:lib1')
```

```
    compile project(':libraries:lib2')
}
```

> 注意：
> 如果你要引用多个库，那么排序将非常重要。这类似于旧构建系统里面的project.properties文件中的依赖排序。

库项目发布

一般情况下一个库只会发布它的*release* Variant（变种）版本。这个版本将会被所有引用它的项目使用，而不管它们本身自己构建了什么版本。这是由于Gradle的限制，我们正在努力消除这个问题，所以这只是临时的限制。

你可以控制哪一个Variant版本作为发行版：

```
android {
    defaultPublishConfig "debug"
}
```

注意这里的发布配置名称引用的是完整的Variant版本名称。*Relesae*，*debug*只适用于项目中没有其它特性版本的时候使用。如果你想要使用其它Variant版本取代默认的发布版本，你可以：

```
android {
    defaultPublishConfig "flavor1Debug"
}
```

将库项目的所有Variant版本都发布也是可能的。我们计划在一般的项目依赖项目（类似于上述所说的）情况下允许这种做法，但是由于Gradle的限制（我们也在努力修复这个问题）现在还不太可能。 默认情况下没有启用发布所有Variant版本。可以通过以下启用：

```
android {
    publishNonDefault true
}
```

理解发布多个Variant版本意味着发布多个arr文件而不是一个arr文件包含所有Variant版本是非常重要的。每一个arr包都包含一个单一的Variant版本。 发布一个变种版本意味着构建一个可用的arr文件作为Gradle项目的输出文件。无论是发布到一个maven仓库，还是其它项目需要创建一个这个库项目的依赖都可以使用到这个文件。

Gradle有一个默认文件的概念。当添加以下配置后就会被使用到：

```
compile project(':libraries:lib2')
```

创建一个其它发布文件的依赖，你需要指定具体使用哪一个：

```
dependencies {
    flavor1Compile project(path: ':lib1', configuration: 'flavor1Release')
    flavor2Compile project(path: ':lib1', configuration: 'flavor2Release')
}
```

> 重要：
> 注意已发布的配置是一个完整的Variant版本，其中包括了build type，并且需要像以上一样被引用。
>
> 重要：
> 当启用非默认发布，maven发布插件将会发布其它Variant版本作为扩展包（按分类器分类）。这意味着不能真正的兼容发布到maven仓库。你应该另外发布一个单一的Variant版本到仓库中，或者允许发布所有配置以支持跨项目依赖。

5

测试 – Testing

构建一个测试程序已经被集成到应用项目中，没有必要再专门建立一个测试项目。

单元测试

单元测试的支持被试验性地添加到 1.1 版本中，请看 这页 (http://tools.android.com/tech-docs/unit-testing-support) 。本章节剩余部分描述了可以在真机（或模拟器）上运行的 “instrumentation tests”，以及需要构建的独立、测试性的 APK。

基本知识和配置

正如前面所提到的，紧邻 `main` *sourceSet*的就是 `androidTest` *sourceSet*，默认路径在src/androidTest/下。 在这个测试*sourceSet*中会构建一个使用Android测试框架，并且可以部署到设备上的测试apk来测试应用程序。这里面包含单元测试，集成测试，和后续UI自动化测试。 这个测试sourceSet不应该包含AndroidManifest.xml文件，因为这个文件会自动生成。

下面这些值可能会在测试应用配置中使用到：

- `testPackageName`
- `testInstrumentationRunner`
- `testHandleProfiling`
- `testfunctionalTest`

正如前面所看到的，这些配置在defaultConfig对象中配置：

```
android {
    defaultConfig {
        testPackageName "com.test.foo"
        testInstrumentationRunner "android.test.InstrumentationTestRunner"
        testHandleProfiling true
        testFunctionalTest true
    }
}
```

在测试应用程序的manifest文件中，instrumentation节点的targetPackage属性值会自动使用测试应用的package名称设置，即使这个名称是通过 `defaultConfig` 或者*Build Type*对象自定义的。这也是manifest文件需要自动生成的一个原因。

另外，这个测试*sourceSet*也可以拥有自己的依赖。 默认情况下，应用程序和他的依赖会自动添加的测试应用的classpath中，但是也可以通过以下来扩展：

```
dependencies {
    androidTestCompile 'com.google.guava:guava:11.0.2'
}
```

测试应用通过 assembleTest task来构建。 assembleTest 不依赖于main中的 assemble task，需要手动设置运行，不能自动运行。

目前只有一个*Build Type*被测试。默认情况下是 debug *Build Type*，但是这也可以通过以下自定义配置：

```
android {
    ...
    testBuildType "staging"
}
```

运行测试

正如前面提到的，标志性task connectedCheck 要求一个连接的设备来启动。 这个过程依赖于 androidTest task，因此将会运行 androidTest 。这个task将会执行下面内容：

- 确认应用和测试应用都被构建（依赖于 assembleDebug 和 assembleTest ）。
- 安装这两个应用。
- 运行这些测试。
- 卸载这两个应用。

如果有多于一个连接设备，那么所有测试都会同时运行在所有连接设备上。如果其中一个测试失败，不管是哪一个设备算失败。

所有测试结果都被保存为XML文档，路径为：

```
_build/androidTest-results_
```

（这类似于JUnit的运行结果保存在build/test-results）

同样，这也可以自定义配置：

```
android {
    ...

    testOptions {
        resultsDir = "$project.buildDir/foo/results"
    }
}
```

这里的 android.testOptions.resultsDir 将由 Project.file(String) 获得。

测试 Android 库

测试Android库项目的方法与应用项目的方法类似。

唯一的不同在于整个库（包括它的依赖）都是自动作为依赖库被添加到测试应用中。结果就是测试APK不单只包含它的代码，还包含了库项目自己和库的所有依赖。 库的manifest被组合到测试应用的manifest中（作为一些项目引用这个库的壳）。

androidTest task的变改只是安装（或者卸载）测试APK（因为没有其它APK被安装）。

其它的部分都是类似的。

测试报告

当运行单元测试的时候，Gradle会输出一份HTML格式的报告以方便查看结果。 Android plugin也是基于此，并且扩展了HTML报告文件，它将所有连接设备的报告都合并到一个文件里面。

独立项目

一个项目将会自动生成测试运行。默认位置为：build/reports/androidTests

这非常类似于JUnit的报告所在位置build/reports/tests，其它的报告通常位于build/reports/< plugin >/。

这个路径也可以通过以下方式自定义：

```
android {
    ...

    testOptions {
        reportDir = "$project.buildDir/foo/report"
    }
}
```

报告将会合并运行在不同设备上的测试结果。

多项目报告

在一个配置了多个应用或者多个库项目的多项目里，当同时运行所有测试的时候，生成一个报告文件记录所有的测试可能是非常有用的。

为了实现这个目的，需要使用同一个依赖文件（译注：指的是使用android gradle插件的依赖文件）中的另一个插件。可以通过以下方式添加：

```
buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:0.5.6'
    }
}

apply plugin: 'android-reporting'
```

这必须添加到项目的根目录下，例如与settings.gradle文件同个目录的build.gradle文件中。

之后，在命令行中导航到项目根目录下，输入以下命令就可以运行所有测试并合并所有报告：

```
gradle deviceCheck mergeAndroidReports --continue
```

> 注意：
> 这里的--continue选项将允许所有测试，即使子项目中的任何一个运行失败都不会停止。如果没有这个选项，第一个失败测试将会终止全部测试的运行，这可能导致一些项目没有执行过它们的测试。

Lint 支持

> Lint支持，译者注：Lint是一个可以检查Android项目中存在的问题的工具

从0.7.0版本开始，你可以为项目中一个特定的Variant（变种）版本运行lint，也可以为所有Variant版本都运行lint。它将会生成一个报告描述哪一个Variant版本中存在着问题。

你可以通过以下lint选项配置lint。通常情况下你只需要配置其中一部分，以下列出了所有可使用的选项：

```
android {
    lintOptions {
        // set to true to turn off analysis progress reporting by lint
```

```
        quiet true
        // if true, stop the gradle build if errors are found
        abortOnError false
        // if true, only report errors
        ignoreWarnings true
        // if true, emit full/absolute paths to files with errors (true by default)
        //absolutePaths true
        // if true, check all issues, including those that are off by default
        checkAllWarnings true
        // if true, treat all warnings as errors
        warningsAsErrors true
        // turn off checking the given issue id's
        disable 'TypographyFractions','TypographyQuotes'
        // turn on the given issue id's
        enable 'RtlHardcoded','RtlCompat', 'RtlEnabled'
        // check *only* the given issue id's
        check 'NewApi', 'InlinedApi'
        // if true, don't include source code lines in the error output
        noLines true
        // if true, show all locations for an error, do not truncate lists, etc.
        showAll true
        // Fallback lint configuration (default severities, etc.)
        lintConfig file("default-lint.xml")
        // if true, generate a text report of issues (false by default)
        textReport true
        // location to write the output; can be a file or 'stdout'
        textOutput 'stdout'
        // if true, generate an XML report for use by for example Jenkins
        xmlReport false
        // file to write report to (if not specified, defaults to lint-results.xml)
        xmlOutput file("lint-report.xml")
        // if true, generate an HTML report (with issue explanations, sourcecode, etc)
        htmlReport true
        // optional path to report (default will be lint-results.html in the builddir)
        htmlOutput file("lint-report.html")

    // set to true to have all release builds run lint on issues with severity=fatal
    // and abort the build (controlled by abortOnError above) if fatal issues are found
    checkReleaseBuilds true
        // Set the severity of the given issues to fatal (which means they will be
        // checked during release builds (even if the lint target is not included)
        fatal 'NewApi', 'InlineApi'
        // Set the severity of the given issues to error
        error 'Wakelock', 'TextViewEdits'
        // Set the severity of the given issues to warning
```

```
        warning 'ResourceAsColor'
        // Set the severity of the given issues to ignore (same as disabling the check)
        ignore 'TypographyQuotes'
    }
}
```

6

构建变种版本 - Build Variants

新构建系统的一个目标就是允许为同一个应用创建不同的版本。

这里有两个主要的使用情景：

1. 同一个应用的不同版本。 例如一个免费的版本和一个收费的专业版本。
2. 同一个应用需要打包成不同的apk以发布Google Play Store。 点击此处 (http://developer.android.com/google/play/publishing/multiple-apks.html) 查看更多详细信息。
3. 综合1和2两种情景。

这个目标就是要让在同一个项目里生成不同的APK成为可能，以取代以前需要使用一个库项目和两个及两个以上的应用项目分别生成不同APK的做法。

不同定制的产品

一个product flavor定义了从项目中构建了一个应用的自定义版本。一个单一的项目可以同时定义多个不同的flavor来改变应用的输出。

这个新的设计概念是为了解决不同的版本之间的差异非常小的情况。虽然最项目终生成了多个定制的版本，但是它们本质上都是同一个应用，那么这种做法可能是比使用库项目更好的实现方式。

Product flavor需要在 `productFlavors` 这个DSL容器中声明：

```
android {
    ....

    productFlavors {
        flavor1 {
            ...
        }

        flavor2 {
            ...
        }
    }
}
```

这里创建了两个flavor，名为 `flavor1` 和 `flavor2` 。

> 注意：
> flavor的命名不能与已存在的*Build Type*或者 `androidTest` 这个*sourceSet*有冲突。

构建类型 + 定制产品 = 构建变种版本

正如前面章节所提到的，每一个*Build Type*都会生成一个新的APK。

*Product Flavor*同样也会做这些事情：项目的输出将会拼接所有可能的*Build Type*和*Product Flavor*（如果有Flavor定义存在的话）的组合。

每一种组合（包含*Build Type*和*Product Flavor*）就是一个*Build Variant*（构建变种版本）。

例如，在上面的Flavor声明例子中与默认的 `debug` 和 `release` 两个*Build Type*将会生成4个*Build Variant*：

- Flavor1 – debug
- Flavor1 – release
- Flavor2 – debug
- Flavor2 – release

项目中如果没有定义flavor同样也会有*Build Variant*，只是使用的是默认的flavor和配置。`default`(默认)的flavor/config是没有名字的，所以生成的Build Variant列表看起来就跟*Build Type*列表一样。

Product Flavor 的配置

每一个flavor都是通过闭包来配置的：

```
android {
    ...

    defaultConfig {
        minSdkVersion 8
        versionCode 10
    }

    productFlavors {
        flavor1 {
            packageName "com.example.flavor1"
            versionCode 20
        }

        flavor2 {
            packageName "com.example.flavor2"
            minSdkVersion 14
```

```
        }
    }
}
```

注意*ProductFlavor*类型的 android.productFlavors.* 对象与 android.defaultConfig 对象的类型是相同的。这意味着它们共享相同的属性。

defaultConfig 为所有的flavor提供基本的配置，每一个flavor都可以重设这些配置的值。在上面的例子中，最终的配置结果将会是：

```
* `flavor1`
    * `packageName`: com.example.flavor1
    * `minSdkVersion`: 8
    * `versionCode`: 20
* `flavor2`
    * `packageName`: com.example.flavor2
    * `minSdkVersion`: 14
    * `versionCode`: 10
```

通常情况下，*Build Type*的配置会覆盖其它的配置。例如，*Build Type*的 packageNameSuffix 会被追加到*Product Flavor*的 packageName 上面。

也有一些情况是一些设置可以同时在*Build Type*和*Product Flavor*中设置。在这种情况下，按照个别为主的原则决定。

例如， signingConfig 就这种属性的一个例子。 signingConfig 允许通过设置 android.buildTypes.release.signingConfig 来为所有的 release 包共享相同的*SigningConfig*。也可以通过设置 android.productFlavors.*.signingConfig 来为每一个release包指定它们自己的*SigningConfig*。

源组件和依赖关系

与*Build Type*类似，*Product Flavor*也会通过它们自己的*sourceSet*提供代码和资源。

上面的例子将会创建4个*sourceSet*：

- android.sourceSets.flavor1 位于src/flavor1/
- android.sourceSets.flavor2 位于src/flavor2/
- android.sourceSets.androidTestFlavor1 位于src/androidTestFlavor1/
- android.sourceSets.androidTestFlavor2 位于src/androidTestFlavor2/

这些*sourceSet*用于与 android.sourceSets.main 和*Build Type*的*sourceSet*来构建APK。

下面的规则用于处理所有使用的sourceSet来构建一个APK：

- 多个文件夹中的所有的源代码（ src/*/java ）都会合并起来生成一个输出。
- 所有的Manifest文件都会合并成一个Manifest文件。类似于*Build Type*，允许*Product Flavor*可以拥有不同的的组件和权限声明。
- 所有使用的资源（ Android res和assets ）遵循的优先级为*Build Type*会覆盖*Product Flavor*，最终覆盖 main *sourceSet*的资源。
- 每一个*Build Variant*都会根据资源生成自己的R类（或者其它一些源代码）。Variant互相之间没有什么是共享的。

最终，类似*Build Type*，*Product Flavor*也可以有它们自己的依赖关系。例如，如果使用flavor来生成一个基于广告的应用版本和一个付费的应用版本，其中广告版本可能需要依赖于一个广告SDK，但是另一个不需要。

```
dependencies {
    flavor1Compile "..."
}
```

在这个例子中，src/flavor1/AndroidManifest.xml文件中可能需要声明访问网络的权限。

每一个Variant也会创建额外的sourceSet：

- `android.sourceSets.flavor1Debug` 位于src/flavor1Debug/
- `android.sourceSets.flavor1Release` 位于src/flavor1Release/
- `android.sourceSets.flavor2Debug` 位于src/flavor2Debug/
- `android.sourceSets.flavor2Release` 位于src/flavor2Release/

这些sourceSet拥有比Build Type的sourceSet更高的优先级，并允许在Variant的层次上做一些定制。

构建和任务

我们前面提到每一个*Build Type*会创建自己的assemble< name >task，但是*Build Variant*是*Build Type*和*Product Flavor*的组合。

当使用*Product Flavor*的时候，将会创建更多的assemble-type task。分别是：

1. assemble< Variant Name > 允许直接构建一个Variant版本，例如 `assembleFlavor1Debug` 。
2. assemble< Build Type Name > 允许构建指定Build Type的所有APK，例如 `assembleDebug` 将会构建Flavor1Debug和Flavor2Debug两个Variant版本。

3. assemble< Product Flavor Name > 允许构建指定flavor的所有APK，例如 `assembleFlavor1` 将会构建Flavor1Debug和Flavor1Release两个Variant版本。

另外 `assemble` task会构建所有可能组合的Variant版本。

测试

测试multi-flavors项目非常类似于测试简单的项目。

`androidTest` *sourceSet*用于定义所有flavor共用的测试，但是每一个flavor也可以有它自己特有的测试。

正如前面提到的，每一个flavor都会创建自己的测试sourceSet：

- `android.sourceSets.androidTestFlavor1` 位于src/androidTestFlavor1/
- `android.sourceSets.androidTestFlavor2` 位于src/androidTestFlavor2/

同样的，它们也可以拥有自己的依赖关系：

```
dependencies {
    androidTestFlavor1Compile "..."
}
```

这些测试可以通过main的标志性 `deviceCheck` task或者main的 `androidTest` task（当flavor被使用的时候这个task相当于一个标志性task）来执行。

每一个flavor也拥有它们自己的task来这行这些测试：androidTest< VariantName >。例如：

- `androidTestFlavor1Debug`
- `androidTestFlavor2Debug`

同样的，每一个Variant版本也会创建对应的测试APK构建task和安装或卸载task：

- `assembleFlavor1Test`
- `installFlavor1Debug`
- `installFlavor1Test`
- `uninstallFlavor1Debug`
- ...

最终的HTML报告支持根据flavor合并生成。 下面是测试结果和报告文件的路径，第一个是每一个flavor版本的结果，后面的是合并起来的结果：

- build/androidTest-results/flavors/< FlavorName >
- build/androidTest-results/all/
- build/reports/androidTests/flavors< FlavorName >
- build/reports/androidTests/all/

Multi-flavor variants

在一些情况下，一个应用可能需要基于多个标准来创建多个版本。例如，Google Play中的multi-apk支持4个不同的过滤器。区分创建的不同APK的每一个过滤器要求能够使用多维的Product Flavor。

假如有个游戏需要一个免费版本和一个付费的版本，并且需要在multi-apk支持中使用ABI过滤器（译注：ABI，应用二进制接口，优点是不需要改动应用的任何代码就能够将应用迁移到任何支持相同ABI的平台上）。这个游戏应用需要3个ABI和两个特定应用版本，因此就需要生成6个APK（没有因计算不同*Build Types*生成的Variant版本）。 然而，注意到在这个例子中，为三个ABI构建的付费版本源代码都是相同，因此创建6个flavor来实现不是一个好办法。 相反的，使用两个flavor维度，并且自动构建所有可能的Variant组合。

这个功能的实现就是使用Flavor Groups。每一个Group代表一个维度，并且flavor都被分配到一个指定的Group中。

```
android {
    ...

    flavorGroups "abi", "version"

    productFlavors {
        freeapp {
            flavorGroup "version"
            ...
        }

        x86 {
            flavorGroup "abi"
            ...
        }

        ...
    }
}
```

`andorid.flavorGroups` 数组按照先后排序定义了可能使用的group。每一个*Product Flavor*都被分配到一个group中。

上面的例子中将*Product Flavor*分为两组（即两个维度），为别为abi维度[x86,arm,mips]和version维度[freeapp,paidapp]，再加上默认的*Build Type*有[debug,release]，这将会组合生成以下的Build Variant：

- x86-freeapp-debug
- x86-freeapp-release
- arm-freeapp-debug
- arm-freeapp-release
- mips-freeapp-debug
- mips-freeapp-release
- x86-paidapp-debug
- x86-paidapp-release
- arm-paidapp-debug
- arm-paidapp-release
- mips-paidapp-debug
- mips-paidapp-release

`android.flavorGroups` 中定义的group排序非常重要（Variant命名和优先级等）。

每一个Variant版本的配置由几个 `Product Flavor` 对象决定：

- android.defaultConfig
- 一个来自abi组中的对象
- 一个来自version组中的对象

flavorGroups中的排序决定了哪一个flavor覆盖哪一个，这对于资源来说非常重要，因为一个flavor中的值会替换定义在低优先级的flavor中的值。

flavor groups使用最高的优先级定义，因此在上面例子中的优先级为：

```
abi > version > defaultConfig
```

Multi-flavors项目同样拥有额外的sourceSet，类似于Variant的sourceSet，只是少了Build Type：

- `android.sourceSets.x86Freeapp` 位于src/x86Freeapp/

- `android.sourceSets.armPaidapp` 位于src/armPaidapp/
- 等等...

这允许在flavor-combination的层次上进行定制。它们拥有过比基础的flavor sourceSet更高的优先级，但是优先级低于Build Type的sourceSet。

7

高级构建定制 – Advanced Build Customization

构建选项

Java 编译选项)

```
android {
    compileOptions {
        sourceCompatibility = "1.6"
        targetCompatibility = "1.6"
    }
}
```

默认值是“1.6”。这个设置将影响所有task编译Java源代码。

aapt 选项

```
android {
    aaptOptions {
        noCompress 'foo', 'bar'
        ignoreAssetsPattern "!.svn:!.git:!.ds_store:!*.scc:.*:<dir>_*:!CVS:!thumbs.db:!picasa.ini:!*~"
    }
}
```

这将影响所有使用aapt的task。

dex 选项

```
android {
    dexOptions {
        incremental false

        preDexLibraries = false

        jumboMode = false

    }
}
```

这将应用所有使用dex的task。

操作 task

基础Java项目有一组有限的task用于互相处理生成一个输出。 `classes` 是一个编译Java源代码的task。可以在build.gradle文件中通过脚本很容易使用 `classes` 。这是 `project.tasks.classes` 的缩写。

在Android项目中，相比之下这就有点复杂。因为Android项目中会有大量相同的task，并且它们的名字基于*Build Types*和*Product Flavor*生成。

为了解决这个问题， `android` 对象有两个属性：

- `applicationVariants` （只适用于app plugin ）
- `libraryVariants` （只适用于library plugin ）
- `testVariants` （两个plugin都适用 ）

这三个都会分别返回一个ApplicationVariant、LibraryVariant和TestVariant对象的DomainObjectCollection (http://www.gradle.org/docs/current/javadoc/org/gradle/api/DomainObjectCollection.html) 。

注意使用这三个collection中的其中一个都会触发生成所有对应的task。这意味着使用collection之后不需要更改配置。

DomainObjectCollection可以直接访问所有对象，或者通过过滤器进行筛选。

```
android.applicationVariants.each { variant ->
    ....
}
```

这三个variant类都共享下面的属性：

属性名	属性类型	说明
name	String	Variant的名字，必须是唯一的。
description	String	Variant的描述说明。
dirName	String	Variant的子文件夹名，必须也是唯一的。可能也会有不止一个子文件夹，例如“debug/flavor1”
baseName	String	Variant输出的基础名字，必须唯一。
outputFile	File	Variant的输出，这是一个可读可写的属性。
processManifest	ProcessManifest	处理Manifest的task。
aidlCompile	AidlCompile	编译AIDL文件的task。
renderscriptCompile	RenderscriptCompile	编译Renderscript文件的task。

属性名	属性类型	说明
mergeResources	MergeResources	混合资源文件的task。
mergeAssets	MergeAssets	混合asset的task。
processResources	ProcessAndroidResources	处理并编译资源文件的task。
generateBuildConfig	GenerateBuildConfig	生成BuildConfig类的task。
javaCompile	JavaCompile	编译Java源代码的task。
processJavaResources	Copy	处理Java资源的task。
assemble	DefaultTask	Variant的标志性assemble task。

ApplicationVariant类还有以下附加属性：

属性名	属性类型	说明
buildType	BuildType	Variant的BuildType。
productFlavors	List	Variant的ProductFlavor。一般不为空但也允许空值。
mergedFlavor	ProductFlavor	android.defaultConfig和variant.productFlavors的合并。
signingConfig	SigningConfig	Variant使用的SigningConfig对象。
isSigningReady	boolean	如果是true则表明这个Variant已经具备了所有需要签名的信息。
testVariant	BuildVariant	将会测试这个Variant的TestVariant。
dex	Dex	将代码打包成dex的task。如果这个Variant是个库，这个值可以为空。
packageApplication	PackageApplication	打包最终APK的task。如果这个Variant是个库，这个值可以为空。
zipAlign	ZipAlign	zip压缩APK的task。如果这个Variant是个库或者APK不能被签名，这个值可以为空。
install	DefaultTask	负责安装的task，不能为空。
uninstall	DefaultTask	负责卸载的task。

LibraryVariant类还有以下附加属性：

属性名	属性类型	说明
buildType	BuildType	Variant的BuildType.
mergedFlavor	ProductFlavor	The defaultConfig values
testVariant	BuildVariant	用于测试这个Variant。
packageLibrary	Zip	用于打包库项目的AAR文件。如果是个库项目，这个值不能为空。

TestVariant类还有以下属性：

属性名	属性类型	说明
buildType	BuildType	Variant的Build Type。

属性名	属性类型	说明
productFlavors	List	Variant的ProductFlavor。一般不为空但也允许空值。
mergedFlavor	ProductFlavor	android.defaultConfig和variant.productFlavors的合并。
signingConfig	SigningConfig	Variant使用的SigningConfig对象。
isSigningReady	boolean	如果是true则表明这个Variant已经具备了所有需要签名的信息。
testedVariant	BaseVariant	TestVariant测试的BaseVariant
dex	Dex	将代码打包成dex的task。如果这个Variant是个库，这个值可以为空。
packageApplication	PackageApplication	打包最终APK的task。如果这个Variant是个库，这个值可以为空。
zipAlign	ZipAlign	zip压缩APK的task。如果这个Variant是个库或者APK不能被签名，这个值可以为空。
install	DefaultTask	负责安装的task，不能为空。
uninstall	DefaultTask	负责卸载的task。
connectedAndroidTest	DefaultTask	在连接设备上行执行Android测试的task。
providerAndroidTest	DefaultTask	使用扩展API执行Android测试的task。

Android task特有类型的API：

- ProcessManifest
    - File manifestOutputFile
- AidlCompile
    - File sourceOutputDir
- RenderscriptCompile
    - File sourceOutputDir
    - File resOutputDir
- MergeResources
    - File outputDir
- MergeAssets
    - File outputDir
- ProcessAndroidResources

    - File manifestFile
    - File resDir
    - File assetsDir
    - File sourceOutputDir
    - File textSymbolOutputDir
    - File packageOutputFile
    - File proguardOutputFile
- GenerateBuildConfig
    - File sourceOutputDir
- Dex
    - File outputFolder
- PackageApplication
    - File resourceFile
    - File dexFile
    - File javaResourceDir
    - File jniDir
    - File outputFile
        - 直接在Variant对象中使用“outputFile”可以改变最终的输出文件夹。
- ZipAlign
    - File inputFile
    - File outputFile
        - 直接在Variant对象中使用“outputFile”可以改变最终的输出文件夹。

每个task类型的API由于Gradle的工作方式和Android plugin的配置方式而受到限制。 首先，Gradle意味着拥有的task只能配置输入输出的路径和一些可能使用的选项标识。因此，task只能定义一些输入或者输出。

其次，这里面大多数task的输入都不是单一的，一般都混合了*sourceSet*、*Build Type*和*Product Flavor*中的值。为了保持构建文件的简单和可读性，目标是要让开发者通过DSL语言修改这些对象来配饰构建的过程，而不是深入修改输入和task的选项。

另外需要注意，除了ZipAlign这个task类型，其它所有类型都要求设置私有数据来让它们运行。这意味着不可能自动创建这些类型的新task实例。

这些API也可能会被更改。一般来说，目前的API是围绕着给定task的输入和输出入口来添加额外的处理（如果需要的时候）。欢迎反馈意见，特别是那些没有预见过的需求。

对于Gradle的task（DefaultTask，JavaCompile，Copy，Zip），请参考Gradle文档。

BuildType 和 Product Flavor 属性参考

即将到来... 对于Gradle的task（DefaultTask，JavaCompile，Copy，Zip），请参考Gradle文档。

使用 JDK 1.7 版本的 sourceCompatibility

使用Android KitKat（19版本的buildTools）就可以使用diamond operator，multi-catch，switch中使用字符串，try with resource等等（译注：都是JDK7的一些新特性，详情请参考JDK7文档）。设置使用1.7版本，需要修改你的构建文件：

```
android {
    compileSdkVersion 19
    buildToolsVersion "19.0.0"

    defaultConfig {
        minSdkVersion 7
        targetSdkVersion 19
    }

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_7
        targetCompatibility JavaVersion.VERSION_1_7
    }
}
```

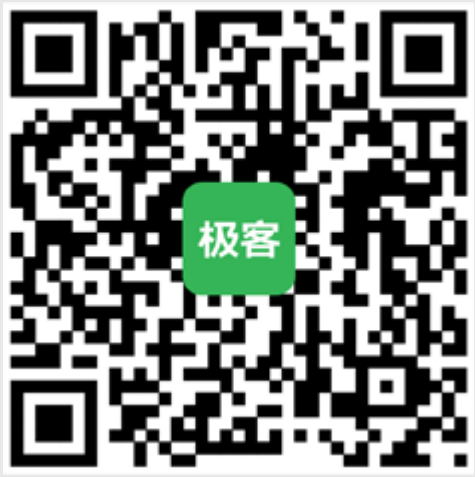
> 注意：
> 你可以将 `minSdkVersion` 的值设置为19之前的版本，只是你只能使用除了try with resources之外的其它新语言特性。如果你想要使用try with resources特性，你就需要把 `minSdkVersion` 也设置为19。

你同样也需要确认Gradle使用1.7或者更高版本的JDK（Android Gradle plugin也需要0.6.1或者更高的版本）。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台



更多信息请访问

http://wiki.jikexueyuan.com/project/android-gradle-guide/