Gradle Plugin使用手册

yeungeek

目錄

译者序

《Gradle Plugin User Guide》 官方地址
http://tools.android.com/tech-docs/new-build-system/user-guide

在线阅读译文
http://chaosleong.gitbooks.io/gradle-for-android/content/

译文 Github 地址
https://github.com/ChaosLeong/Gradle-Android-Plugin

本中文指南的翻译内容大部分参考

Avatar Qing 的 《Gradle Plugin User Guide 中文版》

flyouting 的 《Gradle Plugin User Guide 中文版》

并做了相应的修正以及更新，如有纰漏，望斧正。

简介

DSL 文档

如需了解所有在 build.gradle 文件可用的选项，请查看 DSL reference。

新构建系统的目标

新构建系统的目标:

- 让重用代码和资源变得更容易
- 使针对不同渠道构建多个 APK 或构建同一个应用的不同定制版本更容易
- 使构建过程更容易配置，扩展和自定义
- 优秀的 IDE 集成

为什么使用 Gradle?

Gradle 是一个能通过插件形式自定义构建逻辑的优秀构建工具。

以下的一些特性让我们选择了 Gradle：

- 使用领域专用语言（DSL）来描述和控制构建逻辑
- 构建文件基于 Groovy，并允许通过 DSL 来声明元素、使用代码操作 DSL 元素这样的混合方式来自定义构建逻辑
- 内置了 Maven 和 Ivy 来进行依赖管理
- 相当灵活。允许使用最好的实现，但是不会强制实现的形式
- 插件可以提供它们的 DSL 和 API 来定义构建文件
- 优秀的 API 工具与 IDE 集成

配置要求

- Gradle 2.2
- SDK Build Tools 19.0.0。一些特性可能需要更高版本。

译者注：

Gradle 版本可以关注：Gradle 官方网站

Android Plugin 版本可关注：Android Plugin for Gradle Release Notes

基础项目

Gradle 项目的构建描述定义在项目根目录下的 *build.gradle* 文件中。(查看 Gradle User Guide 了解更多 Gradle 相关知识)

构建文件示例

比较简单的 Android Gradle 项目的 *build.gradle* 如下：

```
buildscript {
    repositories {
        jcenter()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:1.3.1'
    }
}

apply plugin: 'com.android.application'

android {
    compileSdkVersion 23
    buildToolsVersion "23.1.0"
}
```

上述内容包含了 Android 构建文件的 3 个主要部分：

buildscript { ... } 配置了用于驱动构建的代码。上述代码声明了项目使用 jCenter 仓库，并且声明了一个 jCenter 文件的 classpath。该文件声明了项目的 Android Gradle 插件版本为 1.3.1。

> 注意：这里的配置只会影响构建过程所需的类库，而不会影响项目的源代码。项目自身需声明自身的仓库和依赖。这个后面会提到。

接着，使用了 **com.android.application** 插件。该插件用于编译 Android 应用。

最后，**android { ... }** 配置了所有 android 构建所需的参数，这也是 Android DSL 的入口点。默认情况下，只有 **compileSdkVersion** 和 **buildtoolsVersion** 这两个属性是必须的。**compileSdkVersion** 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` (api level) 或者 `String` 类型的值。

> 重要： **com.android.application** 插件不能与 **java** 插件同时使用，否则会导致构建错误。
>
> 注意： 你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。或通过设置 **ANDROID_HOME** 环境变量来设置 SDK 路径。这两种方式都是一样的，根据喜好选择其中一种。
> *local.properties* 文件示例：
>
> ```
> sdk.dir=/path/to/Android/Sdk
> ```

项目结构

上面提到的构建文件中有默认的文件夹结构。Gradle 遵循约定优先于配置的概念，在尽可能的情况下提供合理的默认配置参数。最基本的项目有两个 “source sets” 组件，分别存放应用代码及测试代码。它们分别位于：

- `src/main/`
- `src/androidTest/`

里面每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 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/`
- `jniLibs/`

这就意味着在 Android plugin 下 `*.java` 文件的 source set 路径是 `src/main/java`，而 manifest 则是 `src/main/AndroidManifest.xml`

> 注意：`src/androidTest/AndroidManifest.xml` 会被自动创建，所以一般情况下不需要手动创建。

配置项目结构

当默认的项目结构不适用时，可以自定义配置。查看 Gradle 文档中 Java plugin 部分以了解如何在纯 Java 项目中进行配置。

Android plugin 使用了类似的语法，但因为 Android 有自己的 *sourceSets*，所以需要配置到 **android** 块中。下面的例子使用了旧的项目结构(Eclipse)，并把 **androidTest** 的 *sourceSet* 映射到 tests 目录中。

```
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：因为旧的结构把所有的源文件（Java, AIDL, Renderscript）放在同一个目录中，所以我们需要重新映射所有的 *sourceSet* 新组件到同一个 **src** 目录下。
>
> 注意：**setRoot()** 会移动所有的 sourceSet（包括它的子目录）到新的目录。例子中把 **src/androidTest/*** 移动到 **tests/***

Android 特有的 *sourceSets* 在 Java *sourceSets* 中不起作用。

构建任务

通用 Task

添加一个 plugin 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，构建过程基于 Task）。
Java 插件和 Android 插件都会创建以下 Task:

- **assemble** 组合项目所有输出任务
- **check** 执行所有检查任务
- **build** 执行 **assemble** 和 **check** 两个 task 的所有工作
- **clean** 会清空项目的输出

实际上 **assemble**，**check** 和 **build** 这三个 task 不做任何事情。它们只是一个 Task 标志，用来告诉 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 会自动监听 task 声明的输入和输出。

多次执行 **build** task 并且期间项目没有任何改动，Gradle 将会提示所有 task `UP-TO-DATE`，这意味着不需要再做任何编译工作。这允许 task 之间互相依赖，而不需要额外的构建操作。

Java 项目的 Task

这里有两个由 Java plugin 创建的十分重要的 task，它们依赖于前面所述的标志性 task：

- **assemble**
  - **jar** 该 task 创建所有输出
- **check**
  - **test** 该 task 执行所有测试

jar task 本身直接或者间接依赖于其他 task: **classes** task 将会被调用于编译 Java 代码。**testClasses** task 用于编译测试，但是很少被调用，因为 **test** task 依赖于它（以及 **classes** task）。

通常情况下，你可能只需要调用 **assemble** 和 **check**，而不需要其他 task。你可以在 Gradle 文档中查看 Java plugin 的全部 task 以及它们的依赖关系。

Android Tasks

Android plugin 使用相同的约定以兼容其他插件，并且附加了标志性的 task，包括:

- **assemble** 组合项目所有输出
- **check** 执行所有检查
- **connectedCheck** 在一个连接的设备或者模拟器上执行检查，它们可以在所有连接的设备上并行执行检查
- **deviceCheck** 通过 APIs 连接远程设备来执行检查，主要用于 CI（Continuos integration，持续集成）服务上。
- **build** 执行 **assemble** 和 **check** 的所有工作
- **clean** 清空项目的输出

这些新的标志性 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 和 release APK。每个输出都有各自的标志性 task 以便单独构建它们。

- **assemble**
  - **assembleDebug**
  - **assembleRelease**

它们都依赖于其它一些 tasks 以完成构建一个APK所需的多个步骤。其中 **assemble** task 依赖于上述两个 task，所以执行 **assemble** 将会同时构建出两个 APK。

> 提示: Gradle 在命令行上支持驼峰命名法的 task 简称，例如，执行
>
> ```
> gralde aR
> ```
>
> 等同与输入
>
> ```
> gradle assembleRelease
> ```
>
> 只要没有其它命令匹配 `aR`
>
> 译者注：assR 同样能运行 assembleRelease task，即简称不一定要是首字母缩写 _(:3」∠)_

check task 也有拥有依赖:

- **check**

    - **lint**
- **connectedCheck**
    - **connectedAndroidTest**
- **deviceCheck**
    - 进行测试时才会触发

最后，只要可被安装（需要签名），插件会为所有的构建类型（**debug, release, test**）创建 install 及 uninstall 相关的 task。例：

- **installDebug**
- **installRelease**
- **uninstallAll**
    - **uninstallDebug**
    - **uninstallRelease**
    - **uninstallDebugAndroidTest**

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功能。

Manifest 属性

通过 DSL 可以配置以下 manifest 属性：

- minSdkVersion
- targetSdkVersion
- versionCode
- versionName
- applicationId（有效的包名--查看 ApplicationId 与 PackageName 了解更多信息）
- testApplicationId（测试应用的包名）
- testInstrumentationRunner

例子：

```
android {
       compileSdkVersion 23
       buildToolsVersion "23.0.1"

       defaultConfig {
           versionCode 12
           versionName "2.0"
           minSdkVersion 16
           targetSdkVersion 23
       }
}
```

可查看 Android Plugin DSL Reference 了解可配置的属性以及它们的默认值。

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 23
    buildToolsVersion "23.0.1"

    defaultConfig {
        versionCode 12
        versionName computeVersionName()
        minSdkVersion 16
        targetSdkVersion 23
    }
}
```

> 注意: 不要使用在给定范围内同名的 getter 方法，否则可能引起冲突。例如，在 **defaultConfig{...}** 中调用 **getVersionName()** 将会自动调用 **defaultConfig.getVersionName()** 方法而不是自定义的 **getVersionName()** 方法。

构建类型

默认情况下，Android Plugin 会自动给项目构建 debug 和 release 版本。两个版本的区别在于能否在安全设备（非 dev）上调试，以及 APK 如何签名。debug 使用通过通用的 `name/password` 对生成的密钥证书进行签名（为了防止在构建过程中出现认证请求）。release 在构建过程中不进行签名，需要自行签名。

这些配置是通过 **BuildType** 对象来完成的。默认情况下，**debug** 和 **release** 实例都会被创建。Android plugin 允许像创建其他 *Build Type* 一样自定义这两种类型。在 **buildTypes** 的 DSL 容器中进行配置：

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

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

上面的代码片段实现了以下功能：

- 配置默认的 **debug** *Build Type*:
  - 设置包名为 `<app appliationId>.debug`，以便能够在同一个设备上安装 *debug* 和 *release* 版的 apk
- 创建了名为 **jnidebug** 的新 *BuildType*，并且以 **debug** 的配置作为默认配置。
- 继续配置 **jnidebug**，开启了 JNI 组件的 debug 功能，并添加了包名后缀。

可以在 **buildTypes** 标签下添加新的元素来创建新的 *Build Types* ，并且通过调用 **initWith()** 或者直接使用闭包来配置它。查看 DSL Reference 了解 *build type* 中可以配置的属性。

除了用于修改构建属性外，*Build Types* 还能用于添加特定的代码及资源。每个 Build Type 都会有匹配的 *sourceSet*。默认的路径为 `src/<buildtypename>/`，例如，`src/debug/java` 目录的资源只会编译于 debug APK 中。这意味着 *BuildType* 名称不能是 *main* 或者 *androidTest*（因为它们是由 plugin 内部实现的），并且他们都必须是唯一的。

跟其他 sourceSet 设置一样，Build Type 的 source set 路径也可以重新定向：

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

另外，每个 *Build Type* 都会创建一个新的 `assemble<BuildTypeName>` 任务。
例如：**assembleDebug**。当 **debug** 和 **release** 构建类型被预创建的时候，**assembleDebug** 和 **assembleRelease** 会被自动创建。同样的，上面提到的 *build.gradle* 代码片段中也会创建 **assembleJnidebug** task，并且 **assemble** 会像依赖 **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。

使用场景：

- debug 模式下才用到的权限
- 自定义的 debug 实现
- 为 debug 模式使用不同的资源（例如，当一个资源的值由绑定的证书决定）

BuildType 的代码和资源会在以下场景中被用到：

- manifest 将被合并到 app 的 manifest
- 代码只是换了一个源文件夹
- 资源将叠加到 main 的资源中，并替换已存在的资源

签名配置

签名一个应用程序需要以下文件（查看 Signing Your Application 了解更多 APK 签名的知识）：

- keystore 文件
- keystore 密码
- key alias
- key 密码
- 签名文件的类型

签名文件的路径、类型、key 名及一组密码构成了 SigningConfig。默认情况下，**debug** 使用 debug keystore，debug keystore 使用默认的 storePassword、keyAlias 及 keyPassword。debug keystore 位于 `$HOME/.android/debug.keystore`，如果文件不存在，则会自动创建。**debug** *Build Type* 默认使用 **debug** 的 *SigningConfig*。

可以通过 **signingConfigs** DSL 容器来创建其他配置或者自定义内建构建类型（debug、release）的默认配置：

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

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

    buildTypes {
        foo {
            signingConfig signingConfigs.myConfig
        }
    }
}
```

以上代码片段指定了 debug keystore 的路径在项目根目录下。其他使用了相同配置的 *Build Types* 亦会受影响，在该例子中为 **debug** *Build Type*。该代码片段同时还创建了新的 *Signing Config* 及使用该签名配置的 *Build Type*。

> 注意：只有默认路径下的 debug keystore 不存在时才会自动创建。改变 debug keystore 的路径并不会在新路径下自动创建 debug keystore。如果创建新的 *SigningConfig*，并使用默认的 debug keystore 路径，那么 debug keystore 会自动创建。换句话说，会不会自动创建是根据 keystore 的路径来判断，而不是配置的名称。
>
> 注意：虽然我们经常用 keystore 的相对路径，但也可以用绝对路径，尽管这并不推荐（自动创建的 debug keystore 除外）
>
> 注意：如果将这些文件添加到版本控制，你可能不希望将密码直接写到这些文件。可查看 Stack Overflow post 了解如何从控制台或者环境变量中获取密码。

依赖、**Library** 和多项目

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。

包依赖

本地包依赖

配置 jar 包需要在 **compile** 中添加响应依赖。下面的代码添加了 `libs` 文件夹中的所有 jar 作为依赖。

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
}

android {
    ...
}
```

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

compile 配置将被用于编译 main application。里面的所有依赖都会被添加到编译 classpath 中，同时也会被打包到最终的 APK 内。以下是添加依赖时可能用到的其他配置:

- **compile** 编译主 moudle
- **androidTestCompile** 编译主 moudle 的测试
- **debugCompile** debug Build Type 的编译
- **releaseCompile** release Build Type 的编译

因为不可能去构建没有关联任何 *Build Type* 的 APK，所以 APK 默认有两个（或更多）的编译配置：**compile** 和 `<buildtype>Compile`。创建新的 *Build Type* 将会自动创建基于该名字的编译配置。如果 debug 版要用自定义库（为了反馈实例化的崩溃信息等），但 release 版不需要，又或者 debug、release 依赖于同一个库的不同版本时，`<buildtype>Compile` 会非常有用（查看 Gradle documentation 了解更多版本冲突时的处理细节）。

远程包依赖

Gradle 支持从 Maven 或 Ivy 仓库中拉取依赖文件。首先必须将仓库添加到列表中，然后必须在 dependencies 中添加 Maven 或 Ivy 声明的包。

```
repositories {
    jcenter()
}


dependencies {
    compile 'com.google.guava:guava:18.0'
}

android {
    ...
}
```

> 注意：**jcenter()** 是指定仓库 URL 的便捷方法。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库，这些库也会一并被拉取回来。

更多关于设置 dependencies 的信息，请参考 Gradle 用户指南 和 DSL 文档。

多项目设置

Gradle 项目可以通过多项目配置依赖于其它 Gradle 项目。通常使用多项目配置会将所有库项目（如 lib1、lib2）添加到指定的根项目（如 libraries）。例如，给定以下项目结构：

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

我们可以找出3个项目。Gradle 将会按照以下名字进行映射：

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

每个项目都有 `build.gradle` 文件来声明自身构建逻辑。另外，在项目根目录下还有一个 *setting.gradle* 文件用于声明所有项目。整个项目结构如下：

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

其中 `setting.gradle` 的内容非常简单。该文件定义了各 Gradle 项目的位置：

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

其中 **:app** 项目可能依赖于这些库，可以通过以下依赖配置声明：

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

更多关于多项目配置的信息请参考 Multi-project Builds。

Library 项目

在之前提到的多项目配置中，**:libraries:lib1** 和 **:libraries:lib2** 可以是 Java 项目，并且 **:app** 这个 Android 项目将会使用它们输出的 *jar* 包。然而，如果你想这部分代码能够访问 Android APIs 或者使用 Android 的资源，那么这些 Library 项目就不能是 Java 项目，只能是 Android Library 项目。

创建 Library 项目

Library 项目跟常规的 Android 项目只有小部分差异。

既然构建 Library 跟构建应用不同，那肯定用不同的插件，但是两个插件内部其实共享大部分同样的代码，且由同一个 jar 包提供：`com.android.tools.build.gradle`

```
buildscript {
    repositories {
        jcenter()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:1.3.1'
    }
}

apply plugin: 'com.android.library'

android {
    compileSdkVersion 23
    buildToolsVersion "23.0.1"
}
```

这里创建了一个使用 API 23 编译的 Library 项目，并且 *SourceSet*、*build types* 及 dependencies 的处理、配置方法都与普通项目一样。

普通项目和 **Library** 项目的区别

Library 项目会输出 *.aar* 包（标准的 Android 包）。包含编译文件（以 jar 包或者 .so 文件形式）和资源文件（manifest, res, assets）。Library 项目同样也可以借助普通项目生成测试 apk 进行测试。标识 Task 同样适用于 Library 项目（**assembleDebug**，**assembleRelease**），因此在命令行构建的方式都是一样的。至于其他方面，Library 项目与 Application 项目相同。它们都拥有 `build types` 和 `product flavors`（后面会提到），也可以生成多个不同版本的 `aar`。注意大部分 *Build Type* 的配置不适用于 Library 项目。但可以根据 Library 项目是被普通项目使用还是被用来测试来自定义 *sourceSet* 中的内容。

引用 Library 项目

引用 Library 项目与引用其他项目一样：

```
dependencies {
    compile project(':libraries:lib1')
    compile project(':libraries:lib2')
}
```

> 注意: 如果要引用多个 Library，那么引用的先后顺序将非常重要。这类似于旧的构建系统的 `project.properties` 文件中的依赖顺序。

Library 项目发布

默认 Library 只会发布 *release* variant（变种）版本。该版本将会被所有引用它的项目使用，无论编译出多少个版本。由于 Gradle 的限制，我们正在努力解决这个问题。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对应的值是完整的 variant 版本名称。*relesae*，*debug* 只适用于项目中没有其它 flavor（特性）时使用。如果想要用其它使用了 flavor 的 variant 版本取代默认的发布版本，你可以：

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

也可以发布 Library 项目的所有 variant 版本。我们计划在一般的 `项目依赖项目` （类似于前面所述）情况下允许这种做法，但是由于 Gradle 的限制（我们在努力克服这问题）现在还不太可能。
默认情况下不会发布所有 variant 版本，但可以通过以下代码启用：

```
android {
    publishNonDefault true
}
```

发布多个 variant 版本意味着发布多个 aar 文件而不是一个 aar 文件包含所有 variant 版本的资源。每个 aar 包都只包含一个 variant 版本。发布一个 variant 版本意味着将对应的 aar 文件作为 Gradle 项目的输出文件。无论是发布到 maven 仓库，还是其它项目需要依赖该 Library 项目都可以直接使用该 aar。

Gradle 有 `默认文件` 的概念。当添加以下配置后就会自动使用：

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

添加其它依赖，则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 configuration 是一个完整的 variant 版本，其中包括了 build type，并且需要像上面那样被引用。
>
> 重要：当发布配置非默认时，Maven 发布插件将会发布其它 variant 版本作为扩展包（含 classifier）。意思是指不能真正符合发布到 Maven 仓库。应该发布只含一个 variant 版本的 aar 到仓库中，或者发布所有配置以支持跨项目依赖。

测试

测试项目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

Android Studio 1.1 添加了单元测试支持，详细请看 Unit testing support。本章的其余部分描述的是 “instrumentation tests”。利用 Instrumentation 测试框架可以构建独立的测试 APK 并运行在真实设备（或模拟器）中进行测试。

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。在测试 *sourceSet* 中会构建使用 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。这里面可包含 unit tests，instrumentation tests 及 uiautomater tests。`<instrumentation>` 节点会包含在测试构建过程中自动生成的 AndroidManifest.xml，但是你也可以自己创建一个 `src/androidTest/AndroidManifest.xml` 文件并添加其他组件。

下面的值可以用来配置测试应用，相当于配置 `<instrumentation>` 节点（查看 DSL reference 了解详情）：

- `testApplicationId`
- `testInstrumentationRunner`
- `testHandleProfiling`
- `testFunctionalTest`

正如前面所看到的，这些配置在 **defaultConfig** 对象中配置：

```
android {
    defaultConfig {
        testApplicationId "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包名设置，即使这个名称是通过 **defaultConfig** 或者 *Build Type* 对象自定义的。这也是 manifest 中这一部分需要自动生成的原因之一。

另外，*androidTest* 也可以拥有自己的依赖。默认情况下，应用程序的依赖均会自动添加到测试应用中，但是也可以通过以下来扩展：

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

测试应用由 **assembleAndroidTest** task 构建。**assemble** task 不依赖于它，当测试运行时它会替代 **assemble** 并自动调用。

目前只有一个 *Build Type* 会被测试。默认情况下是 **debug** *Build Type*，可以通过以下代码进行自定义配置：

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

解决 main APK 与 test APK 之间的冲突

> 注意：该章节的内容均为直译，由于个人测试的结果跟文章说的内容存在差异，所以并不确定字里行间的真正意思。如果存疑，请结合官网原文阅读。

当 instrumentation 测试用例运行时，main APK 与 test APK 共享相同的 classpath。如果 main APK 与 test APK 使用相同的库（例：Guava）但版本不同的时候，Gradle 将会构建失败。如果 Gradle 没有对这部分问题进行处理（译注：估计这里是指旧版本 Gradle 没处理这个问题，但我用 0.10.+ 的 android plugin、1.11 的 gradle 也没出现构建失败），你的应用在测试和正常运行过程中会出现不同的行为（包括崩溃的表现也可能不一样）。

为了构建成功，需确定使用了相同版本的库。如果是由于间接依赖（没有在你的 build.gradle 提及的库）而导致构建错误，只需添加对应的新版本库依赖到配置中（"compile" 或 "androidTestCompile"），你也可以使用 Gradle's resolution strategy mechanism 来解决错误。你可以通过运行 **./gradlew :app:dependencies** 及 **./gradlew :app:androidDependencies** 查阅依赖树形图。

> 译注：Gradle's resolution strategy mechanism，是 Gradle 用于定义解析依赖策略的 API，可用于强制指定依赖库的版本、库的替代策略、冲突解决、库缓存本地的时间等

运行测试

正如前面提到的，**connectedCheck** 需要一个已连接设备。这个过程依赖于 **connectedDebugAndroidTest** task，因此 **connectedDebugAndroidTest** task 也会运行。该 task 会执行以下内容：

- 确认应用和测试应用已被构建（依赖于 **assembleDebug** 和 **assembleDebugAndroidTest**）
- 安装这两个应用
- 运行测试
- 卸载这两个应用

如果有多个连接设备，测试会并行在所有连接设备上。如果其中一个测试失败，不管在哪个设备都算测试失败。

测试 Android Library 项目

测试 Android Library 项目类似于测试应用项目。唯一的不同点在于整个库（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 不单只包含自身的代码，还包含了 Library 项目以及它依赖的代码。Library 的 manifest 被组合到测试应用的 manifest 中（引用这个 Library 的项目作为容器）。**androidTest** task 变为只执行安装（或者卸载）测试 APK（因为没有其它 APK 可以被安装），至于其它部分都是一样的。

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 则将所有连接设备的测试报告都合并到一个 HTML 格式的报告文件中。所有测试结果都以 XML 文件形式保存到 `build/reports/androidTests/` 中（类似于 JUnit 的运行结果保存在 build/reports/tests 中)。可以自定义路径：

```
android {
    ...

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

android.testOptions.resultsDir 的值由 **Project.file(String)** 计算获得。

多项目报告

在一个配置了多个应用项目和多个 Library 项目的多项目里，当同时运行所有测试的时候，测试结果整合到一份测试报告中可能是非常有用的。

为了实现这个目的，需要在同一个配置中添加另一个插件。可以通过以下方式添加：

```
buildscript {
    repositories {
        jcenter()
    }

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

apply plugin: 'android-reporting'
```

必须添加在项目的根目录下，例如，与 *settings.gradle* 文件同级目录的 *build.gradle* 文件中。

之后，在命令行中进入项目根目录，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测试运行失败都不会停止。如果没有这个选项，其中一个测试失败则会终止所有测试的运行，此时部分项目可能还未执行测试。

Lint 支持

你可以为特定 variant 运行 lint，例如 **./gradlew lintRelease**，或为所有 variants 运行（**./gradlew lint**），这种情况下会生成一份包含特定版本存在的问题的详细报告。你可以像下面的代码片段那样通过配置 lintOptions 节点来配置 lint。一般只能配置小部分选项，查看 DSL reference 了解所有可修改的选项。

```
android {
    lintOptions {
        // 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'
    }
}
```

构建 **Variants**（变种）版本

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，免费的版本和收费的专业版本。
2. 同一个应用需要打包成不同的 apk 以发布到 Google Play Store，查看 Multiple APK Support 了解详情。
3. 综合 1 和 2 两种情景。

该目标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 Library 项目结合两个或以上的普通项目分别生成不同 APK 的做法。

产品定制

Product flavor 可定义应用的不同定制版本，一个项目可以同时定义多个不同的 flavor 来改变应用的输出。

不同定制版本之间的差异非常小的情况，可以考虑使用 Product flavor。虽然项目最终会生成多个定制版本，但是它们本质上都是同一个应用。这种做法可能是比使用 Library 项目更好的实现方式。

Product flavor 需要在 **productFlavors** 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* 或者与 **androidTest**、**test** *sourceSet* 有冲突。

构建类型 + 产品定制 = 构建变种版本

正如前面章节所提到的，每一个 *Build Type* 都会生成新的 APK。*Product Flavors* 同样也会做这些事情：项目的输出将会组合所有的 *Build Types* 和 *Product Flavors*（如果有定义 Flavor）。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，在之前的 Flavor 声明例子中与默认的 **debug** 和 **release** 两个 *Build Types* 将会生成 4 个 *Build Variants*：

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

项目中如果没有定义 flavor 同样也会有 *Build Variants*，只是使用的是 **default** 的 flavor/config，因为是无名称的，所以生成的 build variant 列表看起来就跟 *Build Type* 列表一样。

产品定制的配置

每个 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。就是说着它们之间属性共享。

defaultConfig 为所有的 flavor 提供基本的配置，每个 flavor 都可以重写这些配置的值。在前面的例子中，最终的配置结果将会是：

- **flavor1**
  - **applicationId**: com.example.flavor1
  - **minSdkVersion**: 8
  - **versionCode**: 20
- **flavor2**
  - **applicationId**: 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 **applicationIdSuffix** 会被追加到 *Product Flavor* 的 **applicationId** 上面。但有部分属性可以同时在 *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，通过设置 **android.buildTypes.release.signingConfig** 来为所有的 release 包共用相同的 *SigningConfig*。也可以通过设置 **android.productFlavors.*.signingConfig** 来为每个 release 包指定它们自己的 *SigningConfig*。

源组件和依赖

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

之前的例子将会创建 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 sourceSet 来构建 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。
- 所有资源（Android res 和 assets）遵循的优先级为 *Build Type* 覆盖 *Product Flavor*，最终覆盖 **main** *sourceSet* 的资源。
- 每个 *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 之间互相独立。

最终，类似 *Build Type*，*Product Flavor* 也可以有它们的依赖关系。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广告 SDK，但是付费版不需要。

```
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 Variants* 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 会构建所有的 variant 版本。

多定制的变种版本

某些情况下，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 种过滤器。根据每个过滤器来创建不同的 APK 需要用到 *Product Flavor*。

假如一个游戏有免费版和付费版，并且需要在 multi-apk 支持中使用 ABI 过滤器。该游戏应用需要 3 个 ABI 和两个特定版本，因此就需要生成 6 个 APK（忽略不同 *Build Types* 生成的 variant 版本）。
然而，3 个 ABI 付费版的源代码都是相同的，因此创建 6 个 flavor 来实现并不是一个好办法。
作为代替，可以使用两个 Flavor Dimensions，并且让它自动构建所有可能的 variant 组合。

对应的功能实现需要使用 Flavor Dimensions（译注：`Flavor Dimensions` 对应旧版中的 `Flavor Groups`），并将 flavor 分配到一个指定的 dimension 中。

```
android {
    ...


    flavorDimensions "abi", "version"


    productFlavors {
        freeapp {
            dimension "version"
            ...
        }

        paidapp {
            dimension "version"
            ...
        }


        arm {
            dimension "abi"
            ...
        }

        mips {
            dimension "abi"
            ...
        }

        x86 {
            dimension "abi"
            ...
        }
    }
}
```

andorid.flavorDimensions 数组按照先后排序定义了可能用到的 dimensions。（示例中）每个 *Product Flavor* 都被分配到一个 dimension 中。

依据 *Product Flavors* [freeapp, paidapp] 和 abi [x86, arm, mips] 以及 *Build Type* [debug,release]，最后会生成以下的 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`

andorid.flavorDimensions 中元素的顺序非常重要（variant 命名和优先级等）。

每个 variant 版本的配置由几个 *Product Flavor* 对象决定：

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

flavorDimensions 中的顺序决定了 flavor 的优先级，这对于资源来说非常重要，因为高优先级 flavor 的值会覆盖低优先级 flavor 的值。
flavor dimension 使用最高的优先级定义，因此前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。它们拥有比最基本的 flavor 的 sourceSet 更高的优先级，但是优先级低于 Build Type 的 sourceSet。

测试

测试 multi-flavors 项目与测试普通的项目相差无几。

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

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 **deviceCheck** task 或者 **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。

每个 flavor 也拥有自己的 task 来执行测试：`androidTest<VariantName>`。例如：

- **androidTestFlavor1Debug**
- **androidTestFlavor2Debug**

同样的，每个 variant 版本也会创建对应的测试 APK 构建 task 和 安装或卸载 task：

- **installFlavor1Debug**
- **assembleFlavor1Test**
- **installFlavor1Test**
- **uninstallFlavor1Debug**
- ...

最终的 HTML 报告根据每个 flavor 的报告汇总而成。
下面是测试结果和报告文件的路径，每组第一个是每个 flavor 版本的结果，第二个是汇总结果：

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

改变 report 或 results 的输出文件夹都只会改变对应的根目录，构建时仍会创建每个 flavor 的文件夹，并且仍会汇总测试结果以及报告文件。

BuildConfig

在编译时，Android Studio 会生成一个名为 **BuildConfig** 的类，它包含了一些编译特定 variant 时使用到的常量指。你可以通过检查这些常量的值来改变不同的 variant 的行为，例如：

```
private void javaCode() {
    if (BuildConfig.FLAVOR.equals("paidapp")) {
        doIt();
    else {
        showOnlyInPaidAppDialog();
    }
}
```

BuildConfig 包含的值如下：

- **boolean DEBUG** —— 当前构建是否开启了 debuggable
- **int VERSION_CODE**
- **String VERSION_NAME**
- **String APPLICATION_ID**
- **String BUILD_TYPE** —— build type 的名字，例："release"
- **String FLAVOR** —— flavor 的名字，例："paidapp"

如果项目使用了 flavor dimensions，会添加额外的值。按照之前的示例，则有如下 BuildConfig：

- **String FLAVOR = "armFreeapp"**
- **String FLAVOR_abi = "arm"**
- **String FLAVOR_version = "freeapp"**

过滤变种版本

当你添加了 dimensions 及 flavors 时，你可以移除无意义的 variants。比如你定义了一个使用 Web API 的 flavor 及一个为了更快地测试而硬编码假数据的 flavor。后者只会用于开发阶段而不会存在于发布阶段。你可以通过 **variantFilter** 闭包方法移除这个 variant：

```
android {
    productFlavors {
        realData
        fakeData
    }

    variantFilter { variant ->
        def names = variant.flavors*.name

        if (names.contains("fakeData") && variant.buildType.name == "release") {
            variant.ignore = true
        }
    }
}
```

像上面那样配置后，你的项目只会存在三个 variants：

- **realDataDebug**
- **realDataRelease**
- **fakeDataDebug**

查看 DSL reference 了解可以通过 **variant** 获取的所有属性。

高级构建的自定义

运行 ProGuard

ProGuard 插件是自动添加进来的，如果 *Build Type* 的 `minifyEnabled` 属性被设置为 true，对应的 task 将会自动创建。

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

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

Variants 会使用在 build type 及 flavors 声明的所有规则文件。

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。

清理资源

你也可以通过修改 `build.gradle` 文件来使得构建过程中自动移除未使用过的资源。查看 Resource Shrinking 文档了解详细信息。

操作 task

一般的 Java 项目中有一组 task 用于协同处理并最终生成一个输出。
classes task 用于编译 Java 源代码。
可以在 *build.gradle* 文件中使用 **classes** 访问 **classes** task 。**classes** 是 **project.tasks.classes** 的缩写。

相比之下在 Android 项目中这就有点复杂。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** 对象有三个属性：

- **applicationVariants**（只适用于 app plugin）
- **libraryVariants**（只适用于 library plugin）
- **testVariants**（app、library plugin 均适用）

这三个属性会分别返回一个 `ApplicationVariant` 、`LibraryVariant` 和 `TestVariant` 对象的 DomainObjectCollection。

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 之后不需要重新配置。

`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 类都拥有下面的属性：

属性名	属性类型	说明
name	String	Variant 的名字，唯一
description	String	Variant 的描述说明
dirName	String	Variant 的子文件夹名，唯一。可能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，该属性可读可写
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
mergeResources	MergeResources	合并资源文件的 task
mergeAssets	MergeAssets	合并 assets 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<ProductFlavor>	Variant 的 ProductFlavor，一般不为空但允许为空
mergedFlavor	ProductFlavor	android.defaultConfig 和 variant.productFlavors 的组合
signingConfig	SigningConfig	Variant 使用的 SigningConfig 对象
isSigningReady	boolean	如果是 true 则表明该 Variant 已经具备了所有需要签名的信息
testVariant	BuildVariant	将会测试该 Variant 的 TestVariant
dex	Dex	将代码打包成 dex 的 task。如果该 Variant 是 Library，该值可为空
packageApplication	PackageApplication	打包最终 APK 的 task。如果该 Variant 是 Library，该值可为空
zipAlign	ZipAlign	对 APK 进行 zipalign 的 task。如果该 Variant 是 Library 或者 APK 不能被签名时，该值可为空
install	DefaultTask	负责安装的 task，可为空
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	只有 android.defaultConfig
testVariant	BuildVariant	用于测试 Variant
packageLibrary	Zip	用于打包 Library 项目的 AAR 文件。如果是 Library 项目，该值不能为空

`TestVariant` 类还有以下属性：

属性名	属性类型	说明
buildType	BuildType	Variant 的 Build Type
productFlavors	List<ProductFlavor>	Variant 的 ProductFlavor。一般不为空但允许为空
mergedFlavor	ProductFlavor	android.defaultConfig 和 variant.productFlavors 的组合
signingConfig	SigningConfig	Variant 使用的 SigningConfig 对象
isSigningReady	boolean	如果是 true 则表明该 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。如果该 Variant 是 Library，该值可为空
packageApplication	PackageApplication	打包最终 APK 的 task。如果该 Variant 是 Library，该值可为空
zipAlign	ZipAlign	对 APK 进行 zipalign 的 task。如果该 Variant 是 Library 或者 APK 不能被签名时，该值可为空
install	DefaultTask	负责安装的 task，可为空
uninstall	DefaultTask	负责卸载的 task
connectedAndroidTest	DefaultTask	在连接设备上执行 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 都受 Gradle 的工作方式和 Android plugin 的配置方式限制。
首先，Gradle 中存在的 task 只能配置输入输出的路径以及部分可能使用的可选标识。因此，这些 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。所以为了保持构建文件的简洁和可读性，目标自然是让开发者通过 DSL 修改这些对象来影响构建的过程，而不是深入修改输入和 task 的选项。

另外需要注意，上面的 task 中除了 ZipAlign，其它都要求设置私有数据进行工作。这意味着不能手动创建这些类型的 task 实例。

这些 API 也可能会被修改。一般来说，目前的 API 是围绕着 task 的输出和输入（如果可能）来添加额外的处理（必要时）。欢迎反馈意见，特别是那些没有考虑过的需求。

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

设置 Java 的版本

你可以使用 **compileOptions** 块选择编译器使用的版本。默认由 **compileSdkVersion** 的值来决定。

```
android {
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_6
        targetCompatibility JavaVersion.VERSION_1_6
    }
}
```

附录

附录主要为《Gradle Plugin User Guide》官方文档中提及的一些额外链接的翻译

ApplicationId 与 PackageName

> 基于 http://blog.csdn.net/maosidiaoxian/article/details/41719357 校对整理，感谢原文作者 貌似掉线

Android 应用都有自己的包名。包名是设备上每个应用程序的唯一标识，同样也是 Google Play 商店里的唯一标识。就是说，假如你已经使用某个包名来发布应用，就不能再去改变应用的包名，因为这样做会导致你的应用被视为一个全新的应用，你现有的用户也不会收到应用的更新通知。

旧版的 Android Gralde 构建系统中，应用的包名由 manifest 中根节点的 package 属性决定：

AndroidManifest.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.my.app"
    android:versionCode="1"
    android:versionName="1.0" >
```

然而，这里所定义的 package 还有另一个作用：用来命名资源类 R（以及用于解析相关的 Activity）。在上面的示例中，最终生成的 R 类为 **com.example.my.app.R**，所以如果你在其他包中的代码需要引用资源，对应的 .java 文件需要导入 `com.example.my.app.R` 。

在新的 Android Gradle 构建系统中，你可以轻松地构建多个不同版本的应用。例如，你可以同时构建免费版和专业版的应用（使用 flavor），并且它们在 Google Play 上也应该要有不同的包名，这样它们就能够在同一设备上安装并且能够单独购买使用等等。同样的，你也可以构建 “debug”、“alpha”、“beta” 版的应用（使用 build type），它们也同样可以有唯一的包名。

同时，代码中引用的 R 类要保持不变；在构建不同版本的应用时，对应的（引用了 R 的）.java 源文件也不能改动。

因此，我们将包名的两种作用解耦：

- “application id” 对应 apk 中 manifest 定义的应用包名，同时用于设备以及 Google Play 的应用唯一标识。
- “package” 用于在源码中引用 R 类以及解析注册相关的 activity/service，对应 Java 的包名概念。

你可以在 Gradle 文件中指定 application id，如下所示：
app/build.gradle:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 19
    buildToolsVersion "19.1"

    defaultConfig {
        applicationId "com.example.my.app"
        minSdkVersion 15
        targetSdkVersion 19
        versionCode 1
        versionName "1.0"
    }
    ...
```

像以前一样，你需要像前面的 AndroidManifest.xml 示例在 Manifest 中指定给代码用的“package”。

关键部分：参照上面的做法，即能解耦 **applicationId** 和 **package**。意思是你能够完全自由地重构你的代码，改变用于 Activity 和 Service 的内部包，改变 Manifest 的 package，重构导入语句。这都不会影响到 app 的最终 id，app 的 id 对应 Gradle 文件中 applicationId 的值。

你可以通过以下的 Gradle DSL 方法来为不同的 flavor 和 build type 定义不同的 applicationId：
app/build.gradle:

```
    productFlavors {
        pro {
            applicationId = "com.example.my.pkg.pro"
        }
        free {
            applicationId = "com.example.my.pkg.free"
        }
    }

    buildTypes {
        debug {
            applicationIdSuffix ".debug"
        }
    }
    ....
```

(在 Android Studio 中，你可以通过 Project Structure 图形化界面来进行这些配置。)

注意：出于兼容性考虑，如果没有在 build.gradle 文件中定义 applicationId，那么 applicationId 将默认为 AndroidManifest.xml 中所指定的 package 的值。在这种情况下，applicationId 和 package 显然未解耦，此时重构代码也将会更改应用的 id ！在 Android Studio 中，新建的项目会指定这两个值。

注意：package 始终必须在默认 AndroidManifest.xml 文件中指定。如果存在多个 manifest（例如一个 flavor 有特定的 manifest 或一个 buildType 有特定的 manifest），package 可不指定，但如果被指定，必须和主 manifest 中指定的 package 完全相同。

AAR 格式文件

'aar' 是由 Android Library Project 析出的可发布的二进制数据包。

该文件后缀名为 .aar，对应的 maven artifact type 应为 aar，但该文件实际是包含下列内容的一个 zip 包：

- /AndroidManifest.xml (必需)
- /classes.jar (必需)
- /res/ (必需)
- /R.txt (必需)
- /assets/ (可选)
- /libs/*.jar (可选)
- /jni//*.so (可选)
- /proguard.txt (可选)
- /lint.jar (可选)

这些内容均直接放置于 zip 包的根路径中。

其中 R.txt 通过指定 aapt 的 --output-text-symbols 选项而生成。