1 简介

这篇文档是基于0.9版本的Gradle插件，1.0以前的版本由于不兼容，可能会有所不同

1.1 新的构建系统的目标

新构建系统的目标是：

- 使得代码和资源的重用更加简单
- 使得创建同一应用程序的不同版本更加容易，不管是多个apk版本还是同一版本的多种定制
- 使得配置，扩展和自定义构建更加容易
- 良好的IDE集成

1.2 为什么使用Gradle

Gradle是一个高级构建系统和构建工具，允许通过插件自定义构建逻辑

以下一些功能使得我们选择Gradle：

- 使用特定领域语言(DSL)来描述和控制构建逻辑
- 构建脚本基于Groovy语言，允许通过DSL混合元素声明和通过代码控制DSL元素，来产生自定义的构建逻辑
- 支持Maven和(或者)Ivy管理依赖
- 非常灵活。允许使用最佳实践，但也不强制自己的实现方式
- 插件能够提供自己的DSL和API供构建脚本使用
- 提供优秀的工具API以供IDE集成

2 环境要求

- Gradle 1.10或者1.11或者1.12，以及插件版本0.11.1
- SDK以及Buid Tools 19.0.0，某些功能可能需要更新的版本

3 基本项目

Gradle项目通过项目根目录下的 build.gradle 文件来描述构建过程

3.1 简单的构建文件

最简单的Java项目构建文件 build.gradle

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

这个脚本应用了Gradle的Java插件。这个插件了提供构建和测试Java应用的所有功能

最简单的Android项目的构建文件包含以下内容：

```
buildscript {    repositories {        mavenCentral()    }    dependencies {
   classpath 'com.android.tools.build:gradle:0.11.1'    }} apply plugin:
'android' android {    compileSdkVersion 19    buildToolsVersion "19.0.0"}
```

Note(译注): 最新的android插件声明
`apply plugin: 'com.android.application'`

在这个Android构建脚本里包含了三个主要内容：

`buildscript { ... }` 配置了驱动构建过程的代码。在这个案例中，声明了使用Maven仓库，以及一个Maven文件(artifact)的依赖路径。这个文件就是包含了Android Gradle插件的库，版本为0.11.1

然后，`android` 插件被应用，像之前的Java插件一样

最后，`android { ... }` 配置了anroid构建过程需要的参数。这也是Adnroid DSL的入口。默认的情况下，只有编译目标SDK版本，和构建工具版本是必须的。在脚本中，对应的是`compileSdkVersion` 和 `buildtoolsVersion` 属性。`compileSdkVersion` 和旧编译系统中 `project.properties` 文件中的 `target` 属性对应。这个新属性 `compileSdkVersion` 可以是一个int值(API Level)或者一个和之前的 `target` 属性值一样的字符串

重点: 你应该只应用android插件，同时应用java插件会导致构建错误

注意: 你同样需要一个 `local.properties` 文件来指明SDK的路径，和 `build.gradle` 在同一路径下，在文件中使用 `sdk.dir` 属性指明。或者，你可以设置 `ANDROID_HOME` 环境变量。两者是一致的，你可以选择一种你喜欢的方式。

3.2 项目结构

前面的android构建脚本使用了默认的文件夹目录结构。Gradle遵循 `约定优于配置` 的原则，在可能的情况下提供了合理的默认配置参数。

基本的项目包含两个“source sets”组件。 `main source code` 和 `test code`，位于以下的目录中：

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

在这些目录中，都存在目录对应源码组件

不管是Java还是Android插件，源码目录和资源目录都如下：

```
java/ resources/
```

对于Android插件，还有特有的文件和目录

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

Note: `src/androidTest/AndroidManifest.xml` 不是必须的，会自动被创建。

3.2.1 配置项目结构

当默认项目结构不合适的时候，可以配置项目目录。根据Gradle文档，可以通过下面的脚本重新配置Java项目的sourceSets：

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

Note: srcDir 会添加指定的目录到源文件目录列表中(这在Gradele文档中没有提及，但是实际上是这样的)。

为了替换默认的源文件目录列表，可以使用 `srcDirs` 来指定目录数组。这也是一种不同的使用方式：

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

更多的信息，参考Gradle文档中的Java插件内容

Android插件使用相似的语法，但是由于使用是自己的`sourceSets`，相应的目录在(`build.gradle`文件中的)android对象中指定

下面是一个示例，它使用旧项目的源码结构，并且将androidTest sourceSet映射到tests目录

```
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')    }}
```

Note: 由于旧的结构将所有的源文件 (java, aidl, renderscript, and java资源文件)放在一个目录里，我们需要映射这些sourceSet组件到src目录。

Note: setRoot() 方法将整个sourceSet(包含子目录)指向新的目录。比如上面，将`src/androidTest/*`指向了`tests/*`

以上是Android特有的，如果配置在Java sourceSets中就没有作用

'migrated' 示例(位于本页面底部)中展示了这部分内容

3.3 构建任务

3.3.1 通用任务

在build文件中应用一个插件将自动创建一系列构建任务。Java插件和Android插件都是这样。任务约定如下：

- assemble

组合项目输出

- check

执行所有检查

- build

执行assemble和check两个task的所有工作

- clean

清理项目输出

任务`assemble`，`check`和`build`不会做任何实际的事情。他们只是锚点任务(anchor tasks)，插件依赖他们来添加实际执行实际操作的任务。

这样就不需要考虑项目是什么类型，使用的是什么插件，都可以执行同样的任务。
例如，使用findbugs插件，会创建新的任务，并让`check`依赖这个任务，使得`check`被调用时这个任务就会被调用。

在终端(命令行，gradle项目目录下)中运行下面的任务可以查询到高级别的任务：

`gradle tasks`运行以下命令可以看到全部任务和任务依赖关系：

```
gradle tasks --all
```

Note: Gradle自动监视一个任务声明的输入输出文件。再次执行构建任务时，如果文件没有改变，Gradle会指明所有任务为`UP-TO-DATE`，意味着任务不需要执行。这样的话，任务可以正确地互相依赖，而不不会导致非必须的构建操作

3.3.2 Java项目的任务

Java插件主要创建两个任务，下面是这两个锚点任务的依赖关系

- assemble
  - jar
    这个任务创建输出
- check
  - test
    这个任务运行测试

`jar`任务直接或间接地依赖其他任务：例如`classes`任务将编译Java源码
`testClasses`任务用于编译测试的，但是这个任务很少被调用，因为`test`任务依赖于它(就像依赖`classes`任务一样)

通常来说，你只需要调用`assemble`或者`check`任务，而不需要调用其他任务。

你可以在Gradle Java插件文档看到Java插件的全部任务和它们的描述

3.3.3 Android任务

Android插件使用同样的约定来保持和其他插件的兼容，并且添加了额外的锚点任务：

- assemble
  这个任务组织项目的输出
- check
  这个项目运行所有检查
- connectedCheck
  运行检查需要一个已连接的设备或者模拟器。并在所有已连接的设备上异步运行。
- deviceCheck
  通过APIs连接远程设备并运行检查。这通常在CI服务器上运行。
- build
  运行`assemble`和`check`
- clean
  清理项目输出

新的锚点任务是必须的，以保证在不需要设备连接的情况下能运行常规检查。
需要注意的是，`build`任务并不依赖`deviceCheck`或者`connectedCheck`

一个Android项目至少有两个输出：debug APK 和 release APK。它们每一个都有自己的锚点任务来帮助它们完成独立的构建：

- assemble
  - assembleDebug
  - assembleRelease

它们都依赖其它任务来完成构建一个apk所需要的多个步骤。`assemble`任务依赖这两个任务，所以调用`assemble`会生成两个APK。

> Tip: Gradle支持在命令行中使用camel形式的任务名缩写。
> 例如：
> `gradle aR`和`gradle assembleRelease`是一样的，因为没有别的任务名有同样的缩写

锚点任务`check`也有自己的依赖：

- check
  - lint
- connectedCheck
  - connectedAndroidTest
  - connectedUiAutomatorTest (not implemented yet)
- deviceCheck
  - 依赖于当其它插件实现测试扩展点时所创建的任务。

最终，插件会为所有构建类型(debug, release, test)创建`install`/`uninstall`任务，如果输出文件可以安装的话(必须签名)。

3.4 基本的构建过程定制

Android插件提供了大量DSL来直接从构建系统中定制大多数事情。

3.4.1 Manifest属性

通过DSL，可以配置以下manifest属性：

- minSdkVersion
- targetSdkVersion
- versionCode
- versionName
- applicationId (实际的packageName – 前往 ApplicationId versus PackageName 查看更多)
- 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    }}
```

配置项位于`android`元素中的`defaultConfig`元素中。

之前版本的Android Plugin使用packageName来配置manifest文件的`packageName`属性。从0.11.1版本开始，你应该在build.gradle文件使用applicationId来配置manifest文件的`packageName`属性。
这是为了消除Android应用的`packageName`(作为Android应用的ID)和java包名之间的疑义。

在构建文件中定义的强大之处在于它可以是动态的。
例如，可以从一个文件中读取版本名称，或者使用自定义的逻辑：

```
def computeVersionName() {    ...} android {    compileSdkVersion 19
buildToolsVersion "19.0.0"     defaultConfig {        versionCode 12
versionName computeVersionName()        minSdkVersion 16        targetSdkVersion
16    }}
```

Note: 函数名不要与指定范围内已经存在的getter方法名冲突。例如，在`defaultConfig { ...}`中调用getVersionName()会自动使用defaultConfig.getVersionName()，而不是你自定义的其它getVersionName()。

如果属性没有通过DSL设置，那么默认的属性值会被使用。下面是默认的属性值列表：

Property Name	Default value in DSL object	Default value
versionCode	-1	value from manifest if present
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的逻辑读取这些属性，那么第二列的属性就很重要。例如，你可能这样写：

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

如果这个值是null，那么在构建过程中会被第三列的默认值替代，但是DSL元素不会包含这个默认值(第三列的值)，所以你查询不到这个值。这是为了防止解析应用的manifest文件，除非真的必要。

3.4.2 构建类型(Build Types)

默认情况下，Android插件会自动设置项目同时构建debug和release版本的应用程序。
这两个版本的不同之处主要在于能否在一个安全设备上调试程序，和APK如何签名。

debug版本使用一个自动创建的密钥/证书，并使用已知的name/password来签名(防止构建过程中出现请求提示)。release版本在构建过程中没有签名，需要稍后签名。

这些配置通过一个构建类型(BuildTpye)对象来设置。默认情况下，debug和release这两个构建类型都会被创建。

Android插件允许自定义这两个实例，也允许创建其它构建类型。这些都在buildTypes的DSL容器中配置：

```
android {    buildTypes {        debug {            applicationIdSuffix ".debug"
       }         jnidebug.initWith(buildTypes.debug)        jnidebug {
packageNameSuffix ".jnidebug"            jniDebuggable true        }    }}
```

上面的代码片段完成来以下功能：

- 配置默认的

  ```
  debug
  ```

  构建类型：
  - 将包名设置成`<app appliationId>.debug`，以便在同一设备上同时安装debug和release版本
- 创建了一个新的名为`jnidebug`的构建类型，是debug构建类型的一个副本
- 配置`jnidebug`构建类型，允许调试JNI组件，并且添加一个不同的包名后缀

创建一个新的构建类型就像在`buildTypes`容器中使用一个新的元素一样简单，可以通过调用`initWith()`或者使用闭包来配置

以下是可能用到的属性和它们的默认值：

Property name	Default values for debug	Default values for release / other
debuggable	true	false
jniDebuggable	false	false
renderscriptDebuggable	false	false
renderscriptOptimLevel	3	3
applicationIdSuffix	null	null
versionNameSuffix	null	null
signingConfig	android.signingConfigs.debug	null
zipAlignEnabled	false	true
minifyEnabled	false	false
proguardFile	N/A (set only)	N/A (set only)
proguardFiles	N/A (set only)	N/A (set only)

除了以上这些属性，*Build Types*还可以通过源码和资源来影响构建过程。
每一个构建类型都会创建一个匹配的*sourceSet*，默认的路径为：

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

这意味这新的构建类型的名字不能是*main*或者*androidTest*(这是插件强制要求的)，而构建类型的名称必须是唯一的。

像其它*sourceSet*一样，构建类型的*sourceSet*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`assemble<BuildTypeName>`任务。

在前面，`assembleDebug`和`assembleRelease`已经提到过了，这就是它们的来源。当debug和release构建类型被预创建的时候，它们相关的任务就被自动创建了，比如`assembleDebug`和`assembleRelease`

上面的*build.gradle*片段同样会创建`assembleJnidebug`任务，`assemble`会像依赖`assembleDebug`和`assembleRelease`任务一样依赖`assembleJnidebug`。

Tip: 你可以在命令行下输入`gradle aJ`来运行`assembleJnidebug`任务。

可能用到的场景：

- 只有debug模式才需要的权限，而release模式不需要
- 自定义debug实现
- debug模式使用不同的资源(例如，资源取值与签名证书绑定)

*BuildType*的源码和资源通过以下方式使用：

- manifest文件合并到app的manifest文件中
- 源码作为另一个源码目录
- 资源叠加到main的资源中，取代已经存在的值

3.4.3 签名配置

对一个应用程序签名需要以下：

- 一个keystore
- 一个keystore密码
- 一个key的别名
- 一个key密码
- 存储类型

位置，别名，两个密码和存储类型一个组成一个签名配置(*SigningConfig*)

默认情况下，`debug`签名配置使用一个debug keystore，已知的密码和已知的别名以及别名密码。debug keystore位于`$HOME/.android/debug.keystore`，如果没有的话会自动创建一个。

`debug`构建类型会自动使用`debug` *SigningConfig*。

可以创建其它签名配置或者自定义默认内建配置。通过`signingConfigs` 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            jniDebuggable true
signingConfig signingConfigs.myConfig        }    }}
```

上面的片段修改debug keystore的位置到项目根目录下。这会影响任何使用它的构建类型，在这个案例中，受影响的是`debug`构建类型。

这里也创建了一个新的签名配置和一个使用这个新签名配置的行的构建类型。

Note: 只有默认路径下debug keystore不存在的时候会被自动创建。改变debug keystore的路径则不会在新的路径下自动创建debug keystore。创建一个名字不同的签名配置，但是使用默认的debug keystore路径，会自动创建debug keystore。也就是说，是否自动创建debug keystore，是由keystore的位置决定，而不是配置的名称。

Note: keystore的路径通常是项目根目录的相对路径，但是也可以使用绝对路径，尽管不推荐这样(debug keystore除外，因为它会自动被创建)。

Note: 如果你要把这些文件添加到版本控制系统中，你可能不想把密码写在文件中。下面的Stack Overflow连接提供了从从控制台或者环境变量读取的方法。
http://stackoverflow.com/questions/18328730/how-to-create-a-release-signed-apk-file-using-gradle
我们以后会更新指南，提供更多的细节

3.4.4 运行ProGuard

ProGuard从Gradle plugin for ProGuard 4.10开始支持的(since Gradle plugin 0.4)。如果构建类型的`minifyEnabled`属性被设置为true，那么Progruard插件会自动被添加进来，对应的任务也自动被创建。

Note: 从Gradle插件版本0.14.0开始BuildType.runProguard更改为minifyEnabled属性。具体请参考 Release notes

```
android {    buildTypes {        release {            minifyEnabled true
   proguardFile getDefaultProguardFile('proguard-android.txt')        }    }
productFlavors {        flavor1 {        }        flavor2 {
proguardFile 'some-other-rules.txt'        }    }}
```

Variant会使用所有声明的规则文件，包括声明在相应的Build Type和flavor中的。

SDK中有两个默认的规则文件：

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

它们位于sdk路径下，使用*getDefaultProguardFile()*可以获取文件的完整路径。它们除了是否要进行优化之外，其它都是相同的。

3.4.5 压缩资源文件

构建时可以自动移除没有被使用的资源文件。更多详细信息请查看文档资源文件压缩

4 依赖关系，Android库项目和多项目设置

Gradle项目可以依赖其它组件，这些组件可以是外部二进制包，或者其它Gradle项目。

4.1 依赖二进制包

4.1.1 本地包

为了配置一个外部库jar依赖，你需要在`compile`配置中添加一个依赖

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

Note: `dependencies`DSL元素是标准Gradle API的一部分，并不属于`android`元素。

`compile`配置是用来编译main应用的。任何添加到编译路径中的东西都会被打包到最终的apk文件中。下面是其它一些在添加依赖时可能用到的配置：

- `compile`: 主module
- `androidTestCompile`: 测试module
- `debugCompile`: debug构建类型的编译
- `releaseCompile`: release构建类型的编译

因为构建一个apk必然有一个相关的构建类型，所以apk通常至少有两个编译配置：`compile`和`<buildtype>Compile`
创建一个构建类型时会自动创建一个基于它名字的编译配置`<buildtype>Compile`

当你在debug版本里需要使用一个自定义库（例如记录crash信息），而release版本不需要，或者他们依赖同一个库的不同版本的时候，会非常有用。

> 也可以通过添加一个目录来依赖目录下的所有jar文件：`compile fileTree(dir: 'libs', include: ['*.jar'])`

4.1.2 远程文件

Gradle支持从Maven或者Ivy仓库获取依赖文件。

首先，必须把仓库添加到列表中，其次，必须按照Maven或者Ivy的文件声明规范来声明依赖。

```
repositories {    mavenCentral()}  dependencies {    compile
'com.google.guava:guava:11.0.2'} android {    ...}
```

Note: `mavenCentral()` 是指定仓库URL的便捷方式。Gradle支持远程和本地仓库。
Note: Gradle遵循依赖关系的传递性。如果一个被依赖文件也依赖其它文件，那些被依赖的文件也会被拉取下来。

更多关于配置依赖的信息，请查看Gradle用户指南和DSL文档

4.2 多项目设置

Gradle项目可以通过多项目设置依赖其它gradle项目。

一个多项目设置通常把所有子项目作为子目录放在指定的项目根目录下。

例如，项目结构如下：

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

我们在这个结构中定义3个项目。Gradle将通过以下名字引用它们：

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

每个项目都有自己的`build.gradle`文件，声明来它怎样构建。另外，在根目录下还有一个`settings.gradle`文件，声明了所有的子项目。
目录结构如下：

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

`settings.gradle`文件的内容十分简单：

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

指明哪个文件夹是一个实际的Gradle项目。

`:app`项目或许依赖其它库项目，那么依赖关系声明如下：

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

更多关于多项目设置的信息在这里。

4.3 库项目

在上面的多项目设置中，`:libraries:lib1`和`:libraries:lib2`可能是Java项目，`:app`Android项目将会使用它们输出的jar文件。

然而，如果你想要共享使用了Android API或者Android资源文件的代码(在库项目中使用了Android API或Android资源文件)，这些库项目就不能是常规的Java项目，必须是Android库项目。

4.3.1 创建一个库项目

一个库项目和常规的Android项目很相似，只有很少的区别。

因为构建库项目和构建应用程序不一样，所以使用不同的插件。构建库项目的插件和构建应用程序的插件在内部共享大部分的代码，并且它们都是由`com.android.tools.build.gradle`jar库提供。

```
buildscript {    repositories {        mavenCentral()    }    dependencies {
   classpath 'com.android.tools.build:gradle:0.5.6'    }} apply plugin: 'android-
library' android {    compileSdkVersion 15}
```

这是一个使用API 15编译的库项目。`SourceSets`和依赖关系的处理跟应用程序项目中一样，而且定制方式也一样。

4.3.2 普通项目和库项目的区别

一个库项目的主要输出是一个`.aar`包（它代表Android的归档文件）。它包含编译好的源码（例如jar文件或者本地.so文件）以及资源文件（manifest, res, assets）。
一个库项目也可以生成一个测试apk来测试，而不依赖应用程序。

由于使用同样的锚点任务（`assembleDebug`，`assembleRelease`），所以在命令行中构建库项目和普通项目没有区别。

其余部分，库项目和应用程序项目一样。都有构建类型和product flavors，可以生成多个版本的aar。要注意的是，多数Build Type配置不适用于库项目。然而，你可以定制`sourceSet`来改变所依赖库的内容，不论它是被普通项目使用还是被测试。

4.3.3 引用一个库项目

引用一个库项目和引用其它项目的方式一样：

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

Note: 如果你由多个库项目，那么顺序是很重要的。这和旧构建系统中的`project.properties`文件中的依赖顺序一样重要。

4.3.4 库项目发布

默认的情况下，库项目只会发布release变种版本(release variant)。这个版本会被所有引用了库项目的项目使用，不管它们自己构建的是什么版本。这是Gradle导致的限制，我们正努力消除这个限制。

可以如下控制哪个版本会被发布：

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

要注意的是，这个发布配置名称必须是完整的variant名称，`release`和`debug`这两个名称只有在没有flavor的时候才使用。如果想要在有flavor的时候改变默认的发布版本，你必须这样写：

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

发布库项目的所有版本也是可能的。我们计划在普通的项目依赖项目的工程中允许这种做法，但是由于Gradle的限制，现在还不能这么做（我们也在努力修复这个问题）。

发布所有版本的功能默认没有开启。开启如下：

```
android {    publishNonDefault true}
```

必须认识到发布多个variant版本意味着发布多个aar文件，而不是在一个aar文件中包含了多个variant版本。每一个aar文件就是一个独立的variant。

发布一个variant版本意味着构建出了一个可用的aar文件，作为Gradle项目的输出文件。这个文件可以发布到maven仓库，或者在其他项目依赖该库项目时作为依赖目标。

Gradle有默认文件的概念。下面这个就使用了默认文件：

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

为了依赖其他的发布版本，你必须指定具体使用哪一个：

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

重要: 要注意已发布的配置是完整的variant版本，包含了构建类型，因此引用的时候也必须是完整的。

重要: 当开启了无默认版本发布，Maven发布插件会把这些额外的版本作为扩展包（按分类器）发布。这意味着并不是真正兼容地发布到maven仓库。你应该发布一个独立的vatiant到仓库，或者开启发布所有配置来支持跨项目依赖。

5 测试

构建一个测试应用已经内置在应用项目内。不需要再创建单独的测试项目。

5.1 单元测试

试验性的单元测试功能支持已经加入到1.1中，具体请看这个页面。本节其他部分讲述的是”instrumentation tests”

5.2 基础和配置

正如前面提到的，紧邻着`main` *sourceSet* 的就是 `androidTest` *sourceSet*，默认在`src/androidTest/`路径下。

从这个*sourceSet* 会构建出一个使用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。这里面可以包含单元测试，集成测试，和后续UI自动化测试。

测试应用的`<instrumentation>`节点是自动生成的，但是你也可以创建一个`src/androidTest/AndroidManifest.xml`，并在这个manifest文件中添加其他组件。

下面是一些测试应用可以配置的值：

- testPackageName
- testInstrumentationRunner
- testHandleProfiling
- testFunctionalTest

正如前面所看到的，这些在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`任务来构建。assembleTest不依赖于main中的`assemble`任务，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"}
```

5.3 运行测试

正如前面提到的，检查通过锚点任务`connectedCheck`启动，这需要一个设备已连接。
这个过程依赖于androidTest任务，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，这个构建就失败。

所有测试结果都被保存为XML文档，路径为：

```
build/androidTest-results
```

（这和常规的JUnit类似，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"    }}
```

`android.testOptions.resultsDir`由Project.file(String)获得。

5.4 测试Android库

测试Android库项目的方法与应用项目的测试方法基本一样。

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（和其他项目引用这个库时一样）。

`androidTest`变成只是安装（或者卸载）测试APK（因为没有其它APK要安装）。

其它的部分都是类似的。

5.5 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。

5.5.1 独立项目

项目将会自动生成测试运行，测试报告默认位置：

```
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。

5.5.2 多项目测试报告

在一个配置了多个应用或者多个库项目的项目中，当同时运行所有测试的时候，生成一个单一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。

然后，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：`--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。

5.6 Lint支持

从0.7.0版本开始，你可以为项目中一个特定的variant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 构建不同版本(Build Variants)

新构建系统的一个目标就是为一个应用构建不同的版本。

有两个主要的场景：

- 同一个应用的不同版本。例如，免费版和收费版
- 同一个应用，为了在Google Play Store上发布并适配多钟设备，打包出不同的apk。
- 以上两种情况的综合

也就是说，从同一个项目中生成这些不同的apk，而不是使用一个库工程和2个以上的主应用工程。

6.1 产品定制(Product flavors)

一个`product flavor`定义了项目构建输出的一个自定义应用版本。一个单独项目可以有不同的flavor，来生成不同的应用。

这个新概念(flavor)是用来解决不同应用版本间差异很小的情形。如果"这是否同一个应用？"的回答是肯定的话，这是比使用库项目更好的做法。

flavor使用`productFlavors`这个DSL容器来声明：

```
android {    ....     productFlavors {        flavor1 {            ...        }
      flavor2 {            ...        }    }}
```

这里创建了两个flavor，分别是 `flavor1` 和 `flavor2`。
注意： flavor的名字不能喝已有的构建类型*(Build Type)*名字冲突，或者和`androidTest`这个*sourceSet*的名字冲突。

6.2 构建类型 + 产品定制 = 变种版本(Build Type + Product Flavor = Build Variant)

前面已经提到，每一个构建类型都会生成一个apk。忘了的话，请看*3.4.2*

Product Flavors 也会做同样的事情，实际上，项目输出来自所有可能的，*Build Types*和*Product Flavors*的组合，如果有*Product Flavors*的话。

每种*Build Types*和*Product Flavors*的组合就是一个*Build Variant*。

例如，默认的`debug` 和 `release`这两个*Build Types*，和上面创建的两个flavor会生成4个*Build Variants*：

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

没有flavor的项目也有*Build Variants*，使用默认的没有名字的flavor配置，使得*Build Variants*列表看起来和 *Build Types*一样。

6.3 ProductFlavor配置

每个flavor在下面这样的闭包结构中配置：

```
android {    ...     defaultConfig {        minSdkVersion 8        versionCode 10
   }     productFlavors {        flavor1 {            packageName
"com.example.flavor1"            versionCode 20        }         flavor2 {
     packageName "com.example.flavor2"            minSdkVersion 14        }
}}
```

注意到`android.productFlavors.*`和`android.defaultConfig`的配置项类型相同，这意味着他们共享相同的属性。

`defaultConfig`为所有的flavor提供默认的配置，每个flavor都可以覆盖配置项的值。上面的例子中，最终的配置如下：

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

通常，*Build Type*配置会覆盖其他配置。例如*Build Type*'的`packageNameSuffix`会添加到*Product Flavor*'的 `packageName`上。

也有一些情况下，一个配置项可以同时在Build Type 和 Product Flavor都进行配置，这时，就要具体情况具体分析了。

例如，`signingConfig`就是这样一个配置项。
可以设置`android.buildTypes.release.signingConfig`让所有release版本使用同一个*SigningConfig*，也可以单独设置`android.productFlavors.*.signingConfig`让各release使用各自的*SigningConfig*。

6.4 源集合和依赖关系

和构建类型类似，产品flavor也可以通过他们自己的sourceSets影响最终的代码和资源

在上面的例子中，创建了四个sourceSet：

- android.sourceSets.flavor1
  位置`src/flavor1/`
- android.sourceSets.flavor2
  位置`src/flavor2/`
- android.sourceSets.androidTestFlavor1
  位置`src/androidTestFlavor1/`
- android.sourceSets.androidTestFlavor2
  位置`src/androidTestFlavor2/`

这些*sourceSet* 都会用来创建apk，和`android.sourceSets.main`以及构建类型的*sourceSet*一起。

下面是构建apk时，所有*sourceSet*的处理原则：

- 所有文件夹里的源码(src/*/java)都会被合并起来构建一个输出。
- 多个Manifest文件会合并成一个。这样使得flavor和构建类型一样，可以有不同的组件和permission
- 所有资源的使用遵循优先级覆盖，Product Flavor资源覆盖main sourceSet资源，Build Type资源覆盖Product Flavor资源
- 每个Build Variant会从资源中生成各自的R文件（或者其他生成的源码）。各个Build Variant不会共享任何东西。

最后，和Build Type一样，Product Flavor还可以有自己的依赖。例如，flavor包含了一个广告版本和一个支付版本，那么就会依赖广告sdk，而其他版本不依赖。

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

在这个特别的情况下，`src/flavor1/AndroidManifest.xml`也许需要添加一个网络权限

每个variant也会包含额外的sourceset：

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

这些sourceset比build type的sourceset有更高的优先级，允许variant级别的定制。

6.5 构建和任务

前面提到，每个Build Type有自己的`assemble<name>`任务。但是Build Variant是Build Type 和Product Flavor组合。

当使用Product Flavor的时候，更多的assemble-type任务会被创建出来，分别是：

1. assemble
   允许直接构建一个Variant版本，例如assembleFlavor1Debug。
2. assemble
   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
   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。

`assemble`任务会构建所有可能的variant版本。

6.6 测试

测试多flavor的项目和简单项目十分类似。

`androidTest` sourceset被用来定义所有flavor的通用测试，同时，每个flavor也可以有各自的测试。

正如上面提到的，测试各flavor的sourceSet会被创建：

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

同样，他们也可以有他们自己的依赖：

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

可以通过锚点任务`deviceCheck`来运行测试，或者`androidTest`任务（当使用flavor时，它作为锚点任务）。

每个flavor有自己的任务运行测试：`androidTest<VariantName>`。例如：

- androidTestFlavor1Debug
- androidTestFlavor2Debug

类似的，每个variant都有构建测试apk和安装/卸载任务。

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

最后，生成的HTML报告支持按照flavor合并。
测试结果和报告位置如下，首先是每个flavor版本的，然后是合并的。

- build/androidTest-results/flavors/
- build/androidTest-results/all/
- build/reports/androidTests/flavors
- build/reports/androidTests/all/

改变任一个路径，只会影响根目录，仍然会为每个flavor和合并后的结果创建子目录。

6.7 多flaver维度的版本(Multi-flavor variants)

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multi-apk支持4个不同的过滤器。为每一个过滤器而创建的apk要求使用多个Product Flavor维度。

假设有一个游戏项目，有demo和付费版本，想要使用multi-apk中的ABI过滤器。由于要兼顾3种ABI和两个版本，所以需要生成6个apk（没有计算多个Build Type产生的版本）。

然而，付费版本的代码对于所有三个ABI都是一样，因此创建简单的6个flavor不是一个好方法。

相反的，将flavor分为两个维度，并自动构建所有可能的组合variant。

这个功能通过Flavor Dimensions来实现。flavor都被分配到一个特定的维度

```
android {    ...     flavorDimensions "abi", "version"     productFlavors {
freeapp {            flavorDimension "version"            ...        }
x86 {            flavorDimension "abi"            ...        }    }}
```

在`android.flavorDimensions`数组中定义可能的维度，并且每个flavor都指定一个维度。

根据已经划分维度的flavor([freeapp, paidapp] 和 [x86, arm, mips])，和Build Type[debug, release]，会创建以下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.flavorDimensions`中定义维度的顺序非常重要。

每个variant配置由多个Product Flavor对象决定：

- android.defaultConfig
- One from the abi dimension
- One from the version dimension

维度的顺序决定哪个flavor的配置会覆盖另一个，这对资源来说很重要，高优先级flavor中的资源会替换低优先级的。flavor维度定义时高优先级在前。所以上面的例子中：

```
abi > version > defaultConfig
```

多维flavor项目也有额外的sourceset，和variant类似，但是没有build type：

- android.sourceSets.x86Freeapp Location `src/x86Freeapp/`
- android.sourceSets.armPaidapp Location `src/armPaidapp/`
- etc...

这些sourceset允许在flavor-combination的级别进行定制。他们比基础的flavor sourceset优先级高，但是比build type sourceset优先级低。

7 高级构建定制

7.1 构建选项

7.1.1 Java编译选项

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

默认值是1.6。影响所有编译java源码的任务。

7.1.2 aapt选项

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

影响所有使用aapt的任务。

7.1.3 dex选项

```
android {    dexOptions {        incremental false        preDexLibraries = false
       jumboMode = false        javaMaxHeapSize "2048M"    }}
```

影响所有使用dex的任务。

7.2 修改构建任务

基础的Java项目有一套有限的任务共同工作来生成输出。

`classes`任务是一个编译Java源码的任务。很容易在`build.gradle`文件的脚本中用`classes`调用。这是`project.tasks.classes`的缩写。

在Android项目中，情况就有点复杂，因为存在大量同样的任务，他们的名字是基于Build Type 和 Product Flavor生成的。

为了解决这个问题，`android`有两个属性：

- applicationVariants (只适用于 app plugin)
- libraryVariants (只适用于 library plugin)
- testVariants (两者都适用)

这三者分别返回一个`ApplicationVariant`，`LibraryVariant`，和`TestVariant`对象的DomainObjectCollection。

要注意使用这些collection中的任一个都会触发创建所有的任务。这意味着使用collection之后不应该修改配置。

`DomainObjectCollection`可以直接访问所有对象，或者通过过滤器筛选（更方便）。

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

所有三种variant共享下面这些属性：

Property Name	Property Type	Description
name	String	variant的名字，必须是唯一的
description	String	variant的描述
dirName	String	Variant的子文件夹名，必须是唯一的。可能也会有多个子文件夹，例如“debug/flavor1”
baseName	String	variant输出的基本名字，必须唯一
outputFile	File	Variant的输出，这是一个可读写的属性
processManifest	ProcessManifest	处理Manifest的任务
aidlCompile	AidlCompile	编译AIDL文件的的任务
renderscriptCompile	RenderscriptCompile	编译Renderscript文件的任务
mergeResources	MergeResources	合并资源文件的任务
mergeAssets	MergeAssets	合并asset的任务
processResources	ProcessAndroidResources	处理并编译资源文件的任务
generateBuildConfig	GenerateBuildConfig	生成BuildConfig类的任务
javaCompile	JavaCompile	编译Java源代码的任务
processJavaResources	Copy	处理Java资源文件的任务
assemble	DefaultTask	variant的标志任务assemble

`ApplicationVariant`拥有以下额外属性：

Property Name	Property Type	Description
buildType	BuildType	variant的构建类型
productFlavors	List	Variant的ProductFlavor。一般不为空但允许空值
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已经具备了签名所需的所有信息
testVariant	BuildVariant	将会测试这个variant的TestVariant
dex	Dex	将代码打包成dex的任务，库工程该属性可以为空
packageApplication	PackageApplication	打包出最终apk的任务，库工程该属性可以为空
zipAlign	ZipAlign	对apk进行对齐(zipalign)的任务，库工程或者apk无法签名时，该属性可以为空
install	DefaultTask	安装apk的任务，可以为空
uninstall	DefaultTask	卸载任务

`LibraryVariant`拥有以下额外属性：

Property Name	Property Type	Description
buildType	BuildType	variant的构建类型
mergedFlavor	ProductFlavor	The defaultConfig values
testVariant	BuildVariant	用于测试这个variant的Variant
packageLibrary	Zip	打包成库工程AAR文件的任务，非库工程该属性为空

`TestVariant`拥有以下额外属性：

Property Name	Property Type	Description
buildType	BuildType	variant的构建类型
productFlavors	List	Variant的ProductFlavor。一般不为空但允许空值
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已经具备了签名所需的所有信息
testedVariant	BaseVariant	被当前TestVariant测试的BaseVariant
dex	Dex	将代码打包成dex的任务，库工程该属性可以为空
packageApplication	PackageApplication	打包出最终apk的任务，库工程该属性可以为空
zipAlign	ZipAlign	对apk进行对齐(zipalign)的任务，库工程或者apk无法签名时，该属性可以为空
install	DefaultTask	安装apk的任务，可以为空
uninstall	DefaultTask	卸载任务
connectedAndroidTest	DefaultTask	在已连接的设备上运行android测试的任务
providerAndroidTest	DefaultTask	使用扩展API运行android测试的任务

API for Android specific task types.

android特有任务的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。
- 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。

由于Gradle的工作方式和Android plugin的配置方式， 每个task类型的API是受限的。
首先，Gradle使得任务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，任务只定义一些输入或者输出。

其次，大多数任务的输入都很复杂，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，可读，我们的目标是让开发者通过略微改动DSL对象来修改构建过程，而不是深入到输入文件和任务选项中去。

另外需要注意，除了ZipAlign这个任务类型，其它所有类型都要求设置私有数据来让它们运行。这意味着不能手动创建这些任务的实例。

这些API也可能改变。大体来说，当前的API是围绕着修改任务的输入（可能的话）和输出来添加额外的处理过程（必要的话）。欢迎反馈，特别是那些没有预见到的问题。

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

7.3 BuildType和Product Flavor属性参考)

敬请期待。
对于Gradle任务(DefaultTask, JavaCompile, Copy, Zip)，请参考Gradle文档。

7.4 使用（JDK）1.7版本的sourceCompatibility)

使用Android KitKat（buildTools v19）就可以使用diamond operator，multi-catch，在switch中使用字符串，try with resource等等（jdk7中的新特性），要使用这些，需要修改你的构建文件如下：

```
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。