Gradle

随着 Google 对 Eclipse 的无情抛弃以及 Studio 的不断壮大， Android 开发者逐渐拜倒在 Studio 的石榴裙下。
而作为 Studio 的默认编译方式， Gradle 已逐渐普及。

接下来我们就系统的学习一下 Gradle 。

简介

Gradle 是以 Groovy 语言为基础，面向 Java 应用为主。基于 DSL(Domain Specific Language) 语法的自动化构建工具。

Gradle 集合了 Ant 的灵活性和强大功能，同时也集合了 Maven 的依赖管理和约定，从而创造了一个更有效的构建方式。凭借 Groovy 的 DSL 和创新打包方式， Gradle 提供了一个可声明的方式，并在合理默认值的基础上描述所有类型的构建。 Gradle 目前已被选作许多开源项目的构建系统。

因为 Gradle 是基于 DSL 语法的，如果想看到 build.gradle 文件中全部可以选项的配置，可以看这里 DSL Reference

基本的项目设置

一个 Gradle 项目通过一个在项目根目录中的 build.gradle 文件来描述它的构建。

简单的 Build 文件

最简单的 Android 应用中的 build.gradle 都会包含以下几个配置：
Project 根目录的 build.gradle：

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.5.0'

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

Module 中的 build.gradle：

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

android {
    compileSdkVersion 23
    buildToolsVersion "23.0.3"
    ...
}
```

- `buildscript { ... }`配置了编译时的代码驱动. 这种情况下，它声明所使用的是`jCenter`仓库。还有一个声明所依赖的在`Maven`文件的路径。这里声明的包含了`Android`插件所使用的1.5.0版本的`Gradle`. 注意:这只会影响`build`中运行的代码，不是项目中。项目中需要声明它自己所需要仓库和依赖关系。
- `apply plugin : com.android.application`，声明使用`com.androdi.application`插件。这是构建`Android`应用所需要的插件。
- `android{...}`配置了所有`Android`构建时的参数。默认情况下，只有编译的目标版本以及编译工具的版本是需要的。

重要: 这里只能使用`com.android.application`插件。如果使用`java`插件将会报错。

目录结构

`module/src/main`下的目录结构，因为有时候很多人把`so`放到`libs`目录就会报错:

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

配置目录结构

如果项目的结构不标准的时候，可能就需要去配置它。`Android`插件使用了相似的语法，但是因为它有自己的`sourceSets`，所以要在`android`代码块中进行配置。下面就是一个从`Eclipse`的老项目结构中配置主要代码并且将`androidTest`的`sourceSet`设置给`tests`目录的例子:

```
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')
    }
}
```

就像有些人就是要把`so`放到`libs`目录中(这类人有点犟)，那就需要这样进行修改。
注意:因为在旧的项目结构中所有的源文件(`Java`,`AIDL`和`RenderScript`)都放到同一个目录中，我们需要将`sourceSet`中的这些新部件都设置给`src`目录。

Build Tasks

对构建文件声明插件时通常或自动创建一些列的构建任务去执行。不管`Java`插件还是`Android`插件都是这样。`Android`常规的任务如下:

- `assemble` 生成项目 `output` 目录中的内容的任务。
- `check` 执行所有的检查的任务。
- `build` 执行 `assemble` 和 `check` 的任务。
- `clean` 清理项目 `output` 目录的任务。

在 `Android` 项目中至少会有两种 `output` 输出:一个 `debug apk` 和一个 `release apk`。他们都有自己的主任务来分别执行构建:

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

提示: `Gradle` 支持通过命令行执行任务首字母缩写的方式。例如:
在没有其他任务符合 `aR` 的前提下， `gradle aR` 与 `gradle assembleRelease` 是相同的。

最后，构建插件创建了为所有 `build type(debug, release, test)` 类型安装和卸载的任务，只要他们能被安装(需要签名)。

- `installDebug`
- `installRelease`
- `uninstallAll`
    - `uninstallDebug`
    - `uninstallRelease`
    - `uninstallDebugAndroidTest`

基本的 `Build` 定制

`Android` 插件提供了一些列的 `DSL` 来让直接从构建系统中做大部分的定制。

`Manifest` 整体部分

`DSL` 提供了很多重要的配置 `manifest` 文件的参数，例如:

- `minSdkVersion`
- `targetSdkVersion`
- `versionCode`
- `versionName`
- `applicationId`
- `testApplicationId`
- `testInstrumentationRunnder`

Android Plugin DSL Reference提供了一个完整的构建参数列表。

把这些 `manifest` 属性放到 `build` 文件中的一个重要功能就是它可以被动态的设置。例如，可以通过读取一个文件或者其他逻辑来获取版本名称。

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

android {
    compileSdkVersion 23
    buildToolsVersion "23.0.1"
```

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

注意:不要使用可能与现有给定冲突的方法名。例如`defaultConfig{...}`中使用`getVersionName()`方法将会自动使用`defaultConfig.getVersionName()`来带起自定义的方法。

`Build Types`

默认情况下`Android`插件会自动将应用程序设置成有一个`debug`版本和一个`release`版本。
这就是通过调用`BuildType`对象完成。默认情况下会创建两个实例，一个`debug`实例和一个`release`实例。`Android`插件同样允许通过其他的`Build Types`来定制其他的实例。这就是通过`buildTypes`来设置的:

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


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

上面的代码执行了以下操作:

- 配置了默认`debug`的`Build Type`:
    - 设置了它的`applicationId`。这样`debug`模式就能与`release`模式的`apk`同时安装在同一手机上。
- 创建了一个新的`jnidebug`的`Build Type`，并且把它设置为`debug`的拷贝。
- 通过允许`JNI`组件的`debug`和增加一个新的包名后缀来继续定制该`Build Type`。

不管使用`initWith()`还是使用其他的代码块，创建一个新的`Build Types`都是非常简单的在`buildTypes`代码块中创建一个新的元素就可以了。

签名配置

为应用签名需要使用如下几个部分:

- `A keystore`
- `A keystore password`
- `A key alias name`
- `A key password`
- `The store type`

默认情况下有一个`debug`的配置，设置了一个`debug`的`keystore`，有一个已知的密码。`debug keystore`的位置是在`$HOME/.android/debug.keystore`，如果没有的话他会被默认创建。`Debug`的`Build Type`会默认使用该`debug`的签名设置。

当然也可以通过使用`DSL`语法中的`signingconfigs`部分来创建其他的配置来进行定制:

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


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


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

上面的设置将把`debug keystore`的位置改为项目的根目录。同样也创建了一个新的签名配置，并且有一个新的`Build Type`使用它。

Dependencies, Android Libraries and Multi-project setup

`Gradle`项目可以依赖其他的外部二进制包、或者其他的`Gradle`项目。

本地包

想要配置依赖一个外部`jar`包，需要在`compile`的配置中添加一个`dependency`。下面的配置是添加了所有在`libs`目录的`jar`包:

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


android {
    ...
}
```

注意: `DSL` 元素中的 `dependencies` 是 `Gradle API` 中的标准元素。不属于 `andorid` 元素。
`compile` 配置是用来编译主应用的。它配置的所有部分都会被打包到 `apk` 中。当然也有一些其他的配置:

- `compile`: `main application`
- `androidTestCompile`: `test application`
- `debugCompile`: `debug Build Type`
- `release Compile`: `release Build Type`

当然我们可以使用 `compile` 和 `<buildtype>.compile` 这两种配置。创建一个新的 `Build Type` 通常会自动基于它的名字创建一个新的配置部分。这样在像 `debug` 版本而 `release` 版本不适用的一些特别的 `library` 时非常有用。

远程仓库

`Gradle` 只是使用 `Maven` 和 `Ivy` 仓库。但是仓库必须要添加到列表中，并且必须声明所依赖仓库的 `Maven` 或者 `Ivy` 定义。

```
repositories {
     jcenter()
}


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


android {
    ...
}
```

注意: `jcenter()` 是指定仓库 `URL` 的快捷设置。 `Gradle` 支持远程和本地仓库。
注意: `Gradle` 会直接识别所有的依赖关系。这就意味着如果一个依赖库自身又依赖别的库时，他们会被一起下下来。

本地 `AAR` 库

```
dependencies {
    compile(name:'本地aar库的名字，不用加后缀', ext:'aar')
}
```

多项目设置

`Gradle` 项目通常使用多项目设置来依赖其他的 `gradle` 项目。例如:

- MyProject/
  - app/
  - libraries/
    - lib1/
    - lib2/

`Gradle`会通过下面的名字来引用他们：
`:app`
`:libraries:lib1`
`:libraries:lib2`

每个项目都会有一个单独的`build`文件，并且在项目的根目录还会有一个`setting.gradle`文件：

- MyProject/
  - settings.gradle
  - app/
    - build.gradle
  - libraries/
    - lib1/
      - build.gradle
    - lib2/
      - build.gradle

`setting.gradle`文件中的内容非常简单。它指定了哪个目录是`Gralde`项目：

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

`: app`这个项目可能会依赖其他的`libraries`，这样可以通过如下进行声明：

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

`Library`项目

上面用到了`:libraries:lib1`和`:libraries:lib2`可以是`Java`项目，`:app`项目会使用他们俩的输出的`jar`包。但是如果你需要使用`android`资源等，这些`libraries`就不能是普通的`Java`项目了，他们必须是`Android Library`项目。

创建一个`Library`项目

`Library`项目和普通的`Android`项目的区别比较少，由于`libraries`的构建类型与应用程序的构建不同，所有它会使用一个别的构建插件。但是他们所使用的插件内部有很多相同的代码，他们都是由`com.android.tools.build.gradle`这个`jar`包提供的。

```
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"
}
```

普通项目与`Library`项目的区别

`Library`项目的主要输出我`.aar`包。它结合了代码(例如`jar`包或者本地`.so`文件)和资源(`manifest`,`res`,`assets`)。每个`library`也可以单独设置`Build Type`等来指定生成不同版本的`aar`。

`Lint Support`

你可以通过指定对应的变量来设置`lint`的运行。可以通过添加`lintOptions`来进行配置:

```
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'
    }
}
```

`Build`变量

构建系统的一个目标就是能对同一个应用创建多个不同的版本。

`Product flavors`

一个`product flavor`可以针对一个项目制定不同的构建版本。一个应用可以有多个不同的`falvors`来改变生成的应用。
`Product flavors`是通过`DSL`语法中的`productFlavors`来声明的:

```
android {
    ....


    productFlavors {
        flavor1 {
            ...
        }


        flavor2 {
            ...
        }
    }
}
```

`Build Type + Product Flavor = Build Variant`

像我们之前看到的，每个`Build Type`都会生成一个`apk`.`Product Flavors`也是同样的：项目的输出僵尸所有`Build Types`与`Product Flavors`的结合。每种结合方式称之为`Build Variant`。例如，如果有`debug`和`release`版本的`Build Types`，上面的例子就会生成4种`Build Variants`：

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

没有配置`flavors`的项目仍然有`Build Variants`，它只是用了一个默认的`flavor/config`，没有名字，这导致`variants`的列表和`Build Types`的列表比较相同。

`Product Flavor`配置

```
android {
    ...


    defaultConfig {
        minSdkVersion 8
        versionCode 10
    }


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


         flavor2 {
             applicationId "com.example.flavor2"
             minSdkVersion 14
         }
    }
}
```

注意`android.productFlavors.*`对象`ProductFlavor`有`android.defaultConfig`是相同的类型。这就意味着他们有相同的属性。
`defaultConfig`为所有的`flavors`提供了一些基本的配置，每个`flavor`都已重写他们。在上面的例子中，这些配置有:

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

通常，`Build Type`配置会覆盖其他的配置。例如，`Build Type`的`applicationIdSuffix`会添加到`Product Flavor`的`applicationId`上。

最后，就像`Build Types`一样，`Product Flavors`也可以有他们自己的依赖关系。例如，如果有一个单独的`flavors`会使用一些广告或者支付，那这个`flavors`生成的`apk`就会使用广告的依赖，而其他的`flavors`就不需要使用。

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

BuildConfig

在编译阶段，`Android Studio`会生成一个叫做`BuildConfig`的类，该类包含了编译时使用的一些变量的值。你可以观看这些值来改变不同变量的行为:

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

下面是`BuildConfig`中包含的一些值:

- `boolean DEBUG` - `if the build is debuggable`
- `int VERSION_CODE`
- `String VERSION_NAME`
- `String APPLICATION_ID`
- `String BUILD_TYPE` - `Build Type`的名字，例如`release`
- `String FLAVOR` - `flavor`的名字，例如`flavor1`

ProGuard配置

`Android`插件默认会使用`ProGuard`插件，并且如果`Build Type`中使用`ProGuard`的`minifyEnabled`属性开启的话，会默认创建对应的`task`。

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

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

Tasks控制

基本的Java项目有一系列的tasks一起制作输出文件。
classes task就是编译Java源码的任务。 我们可以在build.gradle中通过使用classes很简单的获取到它。就是project.tasks.classes.

在Android项目中，更多的编译task，因为他们的名字通过Build Types和Product Flavors生成。

为了解决这个问题，android对象有两种属性:

- applicationVariants - only for the app plugin
- libraryVariants - only for the library plugin
- testVariants - for both plugins
  这些都会返回一个ApplicationVariant, LibraryVariant,TestVariant的DomainObjectCollection接口的实现类对象。
  DomainObjectCollection提供了直接获取或者很方便的间接获取所有对象的方法。

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

设置编译语言版本

可以使用compileOptions代码块来设置编译时使用的语言版本。默认是基于compileSdkVersion的值。

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

Resource Shrinking

Gradle构建系统支持资源清理：对构建的应用会自动移除无用的资源。不仅会移除项目中未使用的资源，而且还会移除项目所以来的类库中的资源。注意，资源清理只能在与代码清理结合使用(例如ProGuad)。这就是为什么它能移除所依赖类库的无用资源。通常，类库中的所有资源都是使用的，只有类库中无用代码被移除后这些资源才会变成没有代码引用的无用资源。

```
android {
    ...

    buildTypes {
        release {
            minifyEnabled true
            shrinkResources true
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
        }
    }
}
```