Gradle Plugin使用手册

yeungeek

Table of Contents

说明

Gradle Plugin User Guide中文版

Gradle Plugin的使用，并结合例子说明

- Gradle Plugin User Guide中文版 正在翻译当中 欢迎大家一起加入
- github: https://github.com/yeungeek/GradlePlugin_UserGuide
- 使用了gitbook进行编辑：http://yeungeek.gitbooks.io/gradle-plugin-user-guide/
- 原文地址：http://tools.android.com/tech-docs/new-build-system/user-guide
- 我会开放权限给需要加入的同学，联系我: yeungeek#gmail.com

翻译进度

章节	时间	译者	实例
1	14.09.29	yeungeek	HelloWorld
2	14.09.29	yeungeek	-
3.1	14.10.08	yeungeek	
3.2	14.10.09	yeungeek	
3.3	14.10.10	yeungeek	
3.4	14.10.20	yeungeek	
4.1	14.11.03	yeungeek	
4.2	14.11.05	yeungeek	
4.3	14.12.01	flyouting	

特色

我们是有实例的人

gradle对应的示例代码，可以fork Samples.

Summary

简介

本文档适用于0.9版本的Gradle plugin。在我们引入1.0版本之前，内容可能会与之前的版本不兼容。

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加容易
- 使创建同一个应用程序的多个版本根据容易，不管是多apk的发布还是同一个应用的不同定制版本
- 使构建过程根据容易配置，扩展和自定义
- 优秀IDE的集成

为什么使用**Gradle?**

Gradle是一个优秀的根据系统和构建工具，它允许通过插件来创建自定义的构建逻辑。

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声明和使用代码混合来定义DSL元素和自定义的构建逻辑
- 内置通过Maven和Ivy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。
- 插件提供DSL和API来定义构建文件
- 优秀的API工具与IDE集成

配置

- Gradle1.10 1.11 1.12使用插件0.11.1版本
- SDK Build Tools 版本19.0.0.一些特性需要更高版本。

译者注：gradle目前已经是2.1版本，插件0.12.+ 最新可以关注: http://www.gradle.org/

基础工程

一个Gradle工程的构建描述，定义在工程根目录下的build.gradle文件中.

简单构建文件

一个最简单Gradle纯Java工程的build.gradle文件包含了以下内容:

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

这是Gradle包装的Java插件。该插件提供了所有构建和测试Java应用程序的东西。 最简单的Android工程的build.gradle描述：

```
buildscript {
    repositories {
        mavenCentral()
    }

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

apply plugin: 'android'

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

译者注：目前gradle tools版本为0.13.+(2014.10.08)

上述内容包含了Android构建文件的3个主要部分：

buildscript { ... }配置了驱动构建的代码.

在这个例子中，他申明了使用Maven中央库，并且申明了一个Maven构件的依赖classpath。这个构件声明了Gradle的Android插件版本为0.11.1.

注意：这里的配置只影响了构建过程的代码，而不是整个工程的代码.工程本身需要声明它自己的仓库和依赖.这个后面会提到.

然后，跟前面提到的Java插件一样，添加了**android**插件.

最后，**android { ... }**配置了所有android构建的参数.也是Android DSL的入口点. 默认情况下，只有编译的target和build-tools版本是必须的.就是**compileSdkVersion**和**buildtoolsVersion** 两个属性. 编译的target属性相当于在老的构建系统中`project.properties`中的target属性.这个新属性和老的target属性一样可以指定一个int(api等级)或者string类型的值.

重要： 你只能使用**android**插件.如果同时使用**java**插件，会导致构建错误.

注意： 你还需要添加*local.properties*文件，使用**sdk.dir**属性，来设置已经存在的SDK路径. 另外，你也可以设置环境变量**ANDROID_HOME**.这两种方式没有什么区别,可以根据你自己的喜好来选择一种.

工程结构

上面提到的基本构建文件需要一个默认的文件结构.Gradle遵循约定优于配置的概念. 在尽可能的情况下提供合理的默认参数. 基本的工程有两个名为"source sets"组件.就是main source code和test code.它们分别位于:

- src/main/
- src/androidTest/

里面的每个文件目录都对应了相应的源组件. 对于Java插件和Android插件，他们对应的Java源代码和Java资源目录：

- java/
- resources/

对于Android插件，有额外的文件和文件目录：

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

注意： src/androidTest/AndroidManifest.xml是不需要的，因为它会自动创建.

配置工程结构

当默认的工程结构不适用时，就可能需要去配置它.根据Gradle文档，根据下面的代码可以重新配置Java工程的sourceSets：

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

> 注意： **srcDir**将会被添加到已存在的源文件目录中(这个在Gradle文档中没有提到，但是实际上确实是这样执行了)

要替换默认的源文件目录，你需要使用一个数组路径的**srcDirs**来替代.下面是使用调用对象的另外一种不同的方法：

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

想了解更多的信息，可以查看Gradle文档中Java插件部分.

Android插件使用了类似的语法，因为使用了它自己的sourceSets，这些配置会被添加到**android**对象中. 下面这个例子,使用了旧工程结构的main代码，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, and java resources)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/***

这是在Android中特有的，在Java sourceSets中不起作用.

上述的就是工程迁移的简单示例.

构建任务

通用任务

添加一个插件到构建文件中，就会自动创建一组可执行的构建任务.Java和Android插件都有此功能. 下面的是约定的构建任务:

- **assemble** 组合工程所有输出的任务
- **check** 执行所有检查的任务
- **build** 执行assemble和check两个任务
- **clean** 这个任务会清空工程的输出

assemble，**check**和**build**这三个任务实际上不做任何事.它们只是一个标记，目的是让plugins添加实际需要的可以完成工作的任务.

这就允许你去调用相同的任务，无论是什么类型的工程，或者是工程应用了任何插件. 例如,使用了*findbugs*插件将会创建一个新的任务，并且让**check**任务依赖它， 当 **check** task被调用的时候,这个新的task 先会被调用.

在命令行中执行以下命令,你可以获取更多高级别的任务:

```
gradle tasks
```

查看task之间依赖关系的完整列表，可以执行以下命令

```
gradle tasks --all
```

> 注意：Gradle会自动监视一个声明了输入和输出的task. 执行两次工程未变化的**build**，Gradle会使用UP-TO-DATE通知所有任务，也就意味着不需要做任何工作.这让任务之间可以相互正确的依赖，不用去执行不必要的构建操作.

Java工程任务

Java插件创建了两个主要的任务，是main标识任务的依赖

- **assemble**
  - **jar** 这个任务创建了所有输出
- **check**
  - **test** 这个任务执行所有的测试

jar任务本身直接或者间接依赖于其他任务: 像**classes**将会编译Java源码. **testClasses**会编译所有的测试，却很少被调用，因为**test**依赖于它(与**classes**任务类似)

通常情况下，你可能只需要调用**assemble**和**check**任务，可以忽略其他的任务.

你可以在件中Java 插件中查看完整的任务列表和他们描述.

Android任务

Android插件使用了相同的约定，使它可以兼容其他插件，并且额外增加了标识性的任务:

- **assemble** 组合工程所有输出的任务
- **check** 执行所有检查的任务
- **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** 使用APIs连接远程设备执行检查.主要用于CI(持续集成)服务上.
- **build** 执行assemble和check两个任务
- **clean** 这个任务会清空工程的输出

新的标志性任务是必须的，为了能够保证在没有设备连接的情况下执行定期检查. 要注意的是**build**并不依赖于**deviceCheck**，或者**connectedCheck**

一个android工程只要有两种输出：debug apk和release apk.每个输出都有各自标志性的任务，可以方便的单独构建它们:

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

它们都依赖于构建一个apk所需要执行多个步骤的其他一些任务. **assemble**任务依赖了上面的两个，所以会构建出两个APK.

> 提示:Gradle在命令行上支持驼峰语法来命名它们的任务名称.例如.
>
> ```
> gralde aR
> ```
>
> 等同与输入
>
> ```
> gradle assembleRelease
> ```
>
> 只要没有其它命令匹配 `aR`

check任务也有他自己的依赖:

- **check**
  - **lint**
- **connectedCheck**
  - **connectedAndroidTest**
  - **connectedUiAutomatorTest**(还么有实现)
- **deviceCheck**
  - 这个任务依赖于其它实现了测试扩展点的插件被创建的时候

最后，插件为所有的构建类型(**debug, release, test**)创建了 `install/uninstall` 任务，只要它们是可以被安装的(需要签名过的).

基本的构建自定义

Android插件提供了大量DSL，直接从构建系统中自定义大部分事情.

Manifest属性

通过DSL可以配置下面的manifest属性:

- minSdkVersion
- targetSdkVersion
- versionCode
- versionName
- applicationId(有效的包名--查看ApplicationId versus PackageName了解更多信息)
- Package Name for the test application
- Instrumentation test runner

例子：

```
android {
    compileSdkVersion 19
    buildToolsVersion "19.0.0"

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

在**android**元素中的**defaultConfig**定义了所有的配置.

在android插件的先前版本使用`packageName`来配置manifest的`packageName`属性. 从0.11.0版本开始,你应该在`build.gradle`中使用`applicationId`来配置manifest的`packageName`属性. 这是被用来消除在应用程序的packageName(程序的ID)和java包名之间的混乱.

在构建文件中定义的强大之处是它的动态性. 例如.可以从文件中或者是自定义的逻辑代码中读取版本名称:

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

android {
    compileSdkVersion 19
    buildToolsVersion "19.0.0"

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

> 注意: 不要使用使用在给定范围内已经存在的getter方法可能引起冲突的方法名.例如，在`defaultConfig { ...}`中调用`getVersionName()`会自动使用defaultConfig.getVersionName()方法去替代自定义的方法.

如果一个属性不是通过DSL来设置的，一些默认的值将被使用.下表是可能用到的默认值:

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

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了自定义逻辑来查询这些属性，第二列的值就变得很重要.例如,你可能会写:

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

如果这值一直为 `null` ,那么在构建的时候将被替换成第3列的默认值,但是在DSL中没有包含该值,所以你无法查询该值 除非是真的有必要才会如此定义， 这是为了预防解析应用的manifest文件.

构建类型

默认情况下，Android插件会自动建立构建应用程序的debug和release版本的工程. 它们的区别主要是能否在一个安全(non dev)的设备上调试,以及APK是如何签名.

debug版本使用了通用的 `name/password` 对自动创建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版本在构建过程中不进行签名，而是需要后面再进行签名.

这个配置是通过一个叫**BuildType**对象来完成的.默认情况下,两个实例会被创建,一个**debug**和一个**release**

Android插件允许像创建其他构建类型那样来自定义这两个实例.可以在**buildTypes**的DSL容器中完成:

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

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

上面的代码片段实现了以下功能：

- 配置默认的**debug**构建类型:
  - 设置包名为 `<app appliationId>` .debug,以便能够在同一个设备上安装*debug*和*release*版本的apk
- 创建了名为**jnidebug**的新*BuildType*,并且配置为是**debug**构建类型的一个副本.
- 继续配置**jnidebug**，可以构建JNI组件,而且增加了一个不同的包名后缀.

签名配置

签名一个应用程序需要以下文件:

- keystore
- keystore密码
- key的别名(alias)
- key密码
- 存储类型

位置,键名,两个密码和存储类型一起组成了这个签名配置(*SigningConfig*)

默认情况下, **debug**被配置成使用debug keystore,keystore使用了已知的密码和一个已知密码的默认key. debug keystore的位置在`$HOME/.android/debug.keystore`， 如果不存在则会自动创建该文件.

debug构建类型会自动使用**debug**的签名配置.

默认配置中可以创建其他配置或者自定义构建.通过**signingConfigs** DSL容器来完成:

```
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
        }
    }
}
```

上面的代码片段修改了debug keystore的位置到工程的根目录下.设置使用了上述的配置会自动影响其他的构建类型,在上述的例子就是**debug**的构建类型.

上述的代码片段使用了新的配置来创建新的签名配置和新的构建类型.

> 注意: 只有在默认的路径下的debug keystore会自动创建.改变了debug keystore路径不会按需进行创建.使用默认debug keystore的路径来创建一个不同名称的SigningConfig， 还是会在默认路径下创建keystore.换句话说,是否会自动创建keystore， 不是根据配置的名称， 而是根据keystore的路径.

注意:

> keystore的路径经常使用工程的根路径,也可以使用绝对路径， 虽然这样是不推荐的(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中， 你可能不想把密码存储在文件中.下面的`Stack Overflow`解答提供了如何从控制台或者从环境变量中读取密码的方法: http://stackoverflow.com/questions/18328730/how-to-create-a-release-signed-apk-file-using-gradle 我们以后会在这个指南中更新更多的详细信息

运行ProGuard

从 `Gradle Plugin for ProGuard version 4.10` 后就开始支持 `ProGuard` .ProGuard插件是自动应用，而且任务是自动创建的,如果构建类型的 `minifyEnabled` 属性被设置成运行ProGuard.

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

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

构建变种使用在它的构建类型和产物定制中声明的规则文件.

这里有两个默认规则文：

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

两个文件在SDK的路径下.使用 `getDefaultProguardFile()` 方法会返回文件的全路径.它们除了是否进行优化之外,其它都是相同的.

依赖,Android库和多项目设置

Gradle工程可以依赖于其他组件.这些组件可以是外部的二进制包,或者是其他Gradle工程.

依赖二进制包

本地包

配置一个外部的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： 该dependencies的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.它里面的所有依赖都会被添加到编译classpath中,同时也会被打包最终的APK.以下是添加依赖时可能用到的其他一些配置选项:

- compile 主程序
- androidTestCompile 测试程序
- debugCompile debug构建类型
- releaseCompile release构建类型

因为不可能去构建一个没有关联任何构建类型的APK,APK默认配置了两个或两个以上的编译配置：compile 和 `<buildtype>` Compile.创建一个新的构建类型将会自动创建一个基于该名字的新配置.

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),但分布时不需要，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仓库中拉取构件. 首先必须将仓库添加到列表中，然后需要声明依赖的Maven或者Ivy构件名称.

```
repositories {
    mavenCentral()
}


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

android {
    ...
}
```

> 注意: mavenCentral()是指定仓库URL的快捷方法.Gradle支持远程和本地的仓库.
>
> 注意: Gradle遵循依赖递归.这意味着，如果一个依赖中又有自己的依赖，那么所有的构件都会被拉取下来.

更多关于设置依赖的信息,请查看Gralde用户指南，和DSL文档.

多项目设置

使用多项目设置，Gradle项目可以依赖其他Gradle项目.

多项目设置的实现通常是在一个根项目路径下，包含了所有子项目的文件夹.

例如,给定以下的项目结构:

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

我们可以定义3个项目.Gradle会按照下面的名称去映射:

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

每个项目都有属于自己的`build.gradle`来声明项目是如何构建的. 另外,在项目的根目录下有个`setting.gradle`. 这些文件的目录结构:

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

`setting.gradle`文件内容很简单：

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

这里定义了哪个文件夹才是真正的Gradle项目.

其中:app项目可能依赖一些库,可以通过声明下面的这些依赖来解决:

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

更多关于多项目配置的信息，请参考多项目配置

库工程

创建一个库工程

一个库工程跟一个常规的Android工程很相似，只是有部分不同.

既然构建库跟构建工程不同，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提供：com.android.tools.build.gradle

```
buildscript {
    repositories {
        mavenCentral()
    }

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

apply plugin: 'android-library'

android {
    compileSdkVersion 15
}
```

这里创建了一个库工程，使用API 15编译，SourceSets dependencies的处理就跟在应用工程中一样，且可以用同样的方式自定义.

普通项目和库项目的区别

一个库工程主要输出一个 `aar` 包(代表Android存档)，这是个编译文件(类似jar包或者.so文件)和资源文件(manifest，res，assets)的组合.

库工程还可以生成一个测试apk来独立测试.

assembleDebug assembleRelease会调起同样的anchor task，所以用通过命令行去构建是无差别的

至于其他的，库工程跟应用工程是一样的，都有 `build type`， `product flavors`，可以生成多个版本的aar

注意，大多数的Build Type配置不支持库工程，然而你可以使用自定义 `sourceSets` 来配置库内容是用于工程使用还是测试使用.

引用一个库工程

引用一个库就跟引用其他工程一样：

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

注意: 如果你有多个库，那么这个顺序很重要，这就好像旧的构建系统，在project.properties文件中的依赖顺序一样重要。

库工程发布

默认情况下,一个库只会发布它的*release* Variant(变种)版本.这个版本通过库项目被其他工程引用,不论它们本身构建了什么版本.这是由于Gradle的限制,我们正在努力消除这个临时的限制.

使用下面的代码,你可以控制哪个Variant版本为发行版

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

注意这里发布的配置名称引用的时完整的Variant版本名称.*Release*和*debug*只能在项目中没有其他特性版本下适用.如果你想改变默认的发布版本，你可以使用这个特性：

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

将库项目的所有变种版本发布也是有可能的。我们计划允许在一般的项目与项目进行依赖(类似于上面所说的情况)，但是由于Gradle的限制(我们正在努力修复这个问题)，目前还没有实现这种功能。
默认情况下没有启用发布所有变种版本的功能。使用下面的代码启用它:

```
android {
    publishNonDefault true
}
```

理解发布多个Variant版本意味着发布多个aar文件，而不是一个aar文件包含多个Variant版本这个很重要。每个aar包含了一个单独的Variant版本.
发布一个Variant版本意味着构建一个可用的aar文件作为Gradle项目的输出构件。无论是被发布到maven仓库，还是被作为另一个项目的库项目依赖。
Gradle有个默认构建的概念。当添加以下配置后就会使用到：

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

创建一个其他发布构建的依赖，你需要指定使用哪个：

```
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，包含了构建的类型，需要像上面一样的格式被引用
> 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。

测试

基本原理和配置

运行测试

测试Android库工程

测试报告

独立工程

多工程报告

Lint支持

构建变种版本

产物定制

产物定制配置

源组件和依赖

构建和任务

多定制的变种版本

高级构建的自定义

构建选项

Java编译选项

aapt选项

dex选项

操作**tasks**

构建类型和产物定制的属性引用

使用**sourceCompatibility 1.7**