

1 背景

在开始Gradle之前请务必保证自己已经初步了解了Groovy脚本，特别是闭包规则，如果还不了解Groovy则可以先看《[Groovy脚本基础全攻略](#)》这一篇博客速成一下Groovy基础，然后再看此文即可。关于Gradle速成干货基础详情也请参考[Geadle官方网站](#)，不好意思我太Low了。



Gradle核心是基于Groovy的领域特定语言(DSL，具体概念参见《[Groovy脚本基础全攻略](#)》)，具有非常好的扩展性，所以不管是简单的独立项目还是大型的多项目构建它都能高效的提高构建任务，尤其对多项目支持是非常牛逼的；Gradle还提供了局部构建功能，譬如构建一个单独子项目时它会构建这个子项目依赖的所有子项目；当然了他对远程仓库和本地库的支持也很到位；哎呀，总之后面你就明白他的牛逼之处了。

既然Gradle核心是Groovy，Groovy本质又是Java，所以很明显可以发现Gradle环境必须依赖JDK与Groovy库，具体如下：

- JDK版本必须是JDK6以上；
- 因为Gradle自带Groovy库，所以已安装的Groovy会被Gradle忽略；

具体Gradle环境配置好了以后如下图：

```
$ gradle -version

-----
Gradle 1.0-milestone-3
-----

Gradle build time: Thursday, September 8, 2011 4:06:52 PM UTC
Groovy: 1.8.6
Ant: Apache Ant(TM) version 1.8.2 compiled on December 3 2011
Ivy: non official version
JVM: 1.6.0_35 (Sun Microsystems Inc. 23.25-b01)
OS: Linux 3.13.0-58-generic amd64
```

【工匠若水 <http://blog.csdn.net/yanbober> 转载请注明出处。 [点我开始Android技术交流](#)】

2 Gradle DSL基础

Gradle的实质是配置脚本，执行一种类型的配置脚本时就会创建一个关联的对象，譬如执行Build script脚本就会创建一个Project对象，这个对象其实就是Gradle的代理对象。下面给出来各种类型Gradle对应的对象类型：

脚本类型	关联对象类型

Build script	Project
Init script	Gradle
Settings script	Settings

Gradle的三种主要对象解释如下：

- Project对象：每个build.gradle会转换成一个Project对象。
- Gradle对象：构建初始化时创建，整个构建执行过程中只有这么一个对象，一般很少去修改这个默认配置脚本。
- Settings对象：每个settings.gradle会转换成一个Settings对象。

可以看见，当我们编写指定类型Gradle脚本时我们可以直接使用关联对象的属性和方法；当然了，每个脚本也都实现了Script接口，也就是说我们也可以直接使用Script接口的属性与方法。

2-1 构建脚本Build script (Project)

在Gradle中每个待编译的工程都是一个Project（每个工程的build.gradle对应一个Project对象），每个Project在构建的时候都包含一系列Task，这些Task中很多又是Gradle的插件默认支持的。

PS：所谓的我们编写Gradle脚本，实质大多数时候都是在编写构建脚本Build script，所以说Project和Script对象的属性和方法等API非常重要。

每一个Project对象和build.gradle——对应，一个项目在构建时都具备如下流程：

1. 为当前项目创建一个Settings类型的实例。
2. 如果当前项目存在settings.gradle文件，则通过该文件配置刚才创建的Settings实例。
3. 通过Settings实例的配置创建项目层级结构的Project对象实例。
4. 最后通过上面创建的项目层级结构Project对象实例去执行每个Project对应的build.gradle脚本。

2-2 初始化脚本Init script (Gradle) 和设置脚本Settings script (Settings)

Gradle对象：

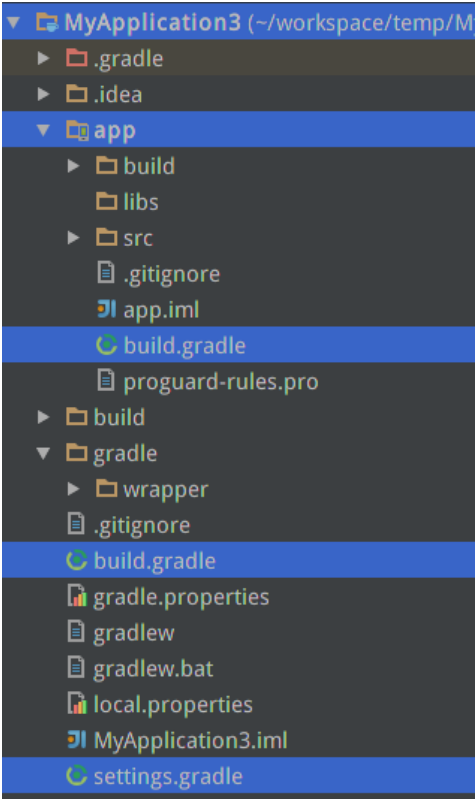
初始化脚本Init script（Gradle）类似于Gradle的其他类型脚本，这种脚本在构建开始之前运行，主要的用途是为接下来的Build script做一些准备工作。我们如果需要编写初始化脚本Init script，则可以把它按规则放置在USER_HOME/.gradle/相关目录下。譬如：

```
jerry@ThinkPad:~$ cat .gradle/wrapper/dists/gradle-2.2.1-all/c64ydeuar  
dnfqctvr1gm30w53/gradle-2.2.1/init.d/readme.txt  
You can add .gradle init scripts to this directory. Each one is execut  
ed at the start of the build. jerry@ThinkPad:~$
```

初始化脚本的Gradle对象代表了Gradle的调运，我们可以通过调用Project对象的getGradle()方法获得Gradle实例对象。

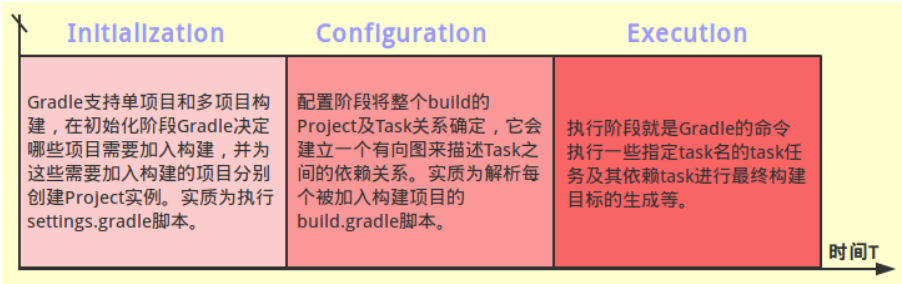
Settings对象：

在对工程进行配置（譬如多项目树构建）时Settings实例与settings.gradle文件——对应，它用来进行一些项目设置的配置。这个文件一般放置在工程的根目录。譬如：



2-3 Build生命周期

Gradle的构建脚本生命周期具备三大步，如下：



可以看见，生命周期其实和上面构建脚本Build script的执行流程是可以关联上的。有了这个流程图我们接下来详细看下每个过程。

settings.gradle文件：

除了构建脚本文件，Gradle还定义了一个约定名称的设置文件（默认为settings.gradle）。该文件在初始化阶段被执行，对于多项目构建必须保证在根目录下有settings.gradle文件，对于单项目构建设置文件是可选的，不过建议还是写上。

如下是单项目构建的一个例子：

```
1 //settings.gradleprintln 'This is executed during the initialization phase.'
2
1 //build.gradleprintln 'This is executed during the configuration phase.'
2
3 task configured {
4     println 'This is also executed during the configuration phase.'
5 }
6
7 task test << {
8     println 'This is executed during the execution phase.'
9 }
10
```

```

11 task testBoth {
12     doFirst {
13         println 'This is executed first during the execution phase.'
14     }
15     doLast {
16         println 'This is executed last during the execution phase.'
17     }
18     println 'This is executed during the configuration phase as well.'
19 }
20

```

运行构建结果：

```

1 > gradle test testBoth
2 This is executed during the initialization phase.
3 This is executed during the configuration phase.
4 This is also executed during the configuration phase.
5 This is executed during the configuration phase as well.
6 :test
7 This is executed during the execution phase.
8 :testBoth
9 This is executed first during the execution phase.
10 This is executed last during the execution phase.
11
12 BUILD SUCCESSFUL
13
14 Total time: 1 secs

```

Gradle多项目构建：

多项目构建总是需要指定一个树根，树中的每一个节点代表一个项目，每一个Project对象都指定有一个表示在树中位置的路径；在设置文件中我们还可以使用一套方法来来自定义构建项目树。

```

1 //分布局的多项目构建settings.gradle文件include 'project1', 'project2:child', 'project3:child1'
2

```

上面例子中把project的路径作为include方法的参数，譬如上面的' project3:child1' 参数就指定了物理路径的project3/child1（ project3/child1是相对于多项目根路径的相对路径），这也同时意味着会创建' project3' 和' project3:child1' 两个project。

```

1 //平面布局的多项目构建settings.gradle文件
2 includeFlat 'project3', 'project4'

```

上面例子中includeFlat方法接受目录名作为参数，但是特别注意，这些项目目录必须是根目录的兄弟目录。

当然了，设置文件中创建的多项目树其实是由项目描述符来描述的，我们可以在设置文件中随时修改这些描述符。如下：

```

1 //settings.gradle
2 rootProject.name = 'main'
3 project(':projectA').projectDir = new File(settingsDir, '../my-project-a')
4 project(':projectA').buildFileName = 'projectA.gradle'

```

可以看见，如上例子通过描述符更改名称和项目目录，并且建立了一个项目的文件。

Gradle构建初始化Initialization：

在初始化阶段如果我们在根路径下直接指明settings.gradle文件和相关配置则构建初始化就会直接按照我们的设置去构建项目，如果我们没指明settings.gradle文件则Gradle会以一定的规则去寻找settings.gradle文件，然后依据寻找结果的不同去决定如何构建项目。

【工匠若水 <http://blog.csdn.net/yanbober> 转载请注明出处。点我开始Android技术交流】

3 Gradle构建基础

通过上一章可以知道，每一个Gradle构建都是由一个或多个project构成，每一个project都是由一个或多个tasks构成，每个task的实质其实是一些更加细化的构建（譬如编译class、创建jar文件等）。

任务task基础：

如下例子我们先来直观感受一下task的概念，具体细节后面会探讨：

```
1 //创建一个名为build.gradle的文件
2 task hello {
3     doLast {
4         println 'Hello world!'
5     }
6 }
7
8 //这是快捷写法，用<<替换doLast，后面解释
9 task hl << {
10     println 'Hello world!'
11 }
12
13 //创建upper的task，使用Groovy语言编写
14 task upper << {
15     String someString = 'mY_nAmE' println "Original: " + someString
16     println "Upper case: " + someString.toUpperCase()
17 }
18
```

通过如下命令运行构建上面名为hello的task，具体如下：

```
1 xxx@XXX:~/ $ gradle hello
2 :helloHello world!
3
4 BUILD SUCCESSFULTotal time: 1.037 secs
5
```

可以看见，gradle命令会在当前目录中查找一个叫build.gradle的构建脚本文件，这个构建脚本定义了一个叫做hello的独立task，并且添加了一个action，我们执行了这个task就得到了想要的结果。

在这里再多嘴一句，我们看下task有无action的区别，如下：

```
1 //有Action的tasktask actionTask << {
2     println 'I am actionTask'
3 }
4 //无Action的tasktask noActionTask {
5     println 'I am noActionTask'
6 }
7
```

一定要记住，在上面这个例子中如果task没有加<<则这个任务在脚本初始化initialization阶段（即无论执行啥task都被执行，具体参见上一章的第一个例子）被执行，如果加了<<则在gradle actionTask后才执行。因为没有加<<则闭包在task函数返回前会执行，而加了<<则变成调用actionTask.doLast()，所以会等到gradle actionTask时执行。

任务task依赖：

我们通过上面task基础感受的例子可以发现，一个build.gradle文件中定义多个task互相没有关系，决定执行的是我们gradle命令后面跟的task名字；那我们要是让他们之间有依赖关系咋办呢？如下：

```
1 task taskX(dependsOn: 'taskY') << {
2
```

```

3     println 'taskX'
4 }
5 task taskY << {
6     println 'taskY'
7 }

```

运行结果如下：

```

1 xxx@XXX:~/$ gradle taskX
2 :taskY
3 taskY
4 :taskX
5 taskX
6
7 BUILD SUCCESSFULTotal time: 1.039 secs
8

```

动态任务task：

我们还可以在Gradle中使用Groovy来创建动态task，如下：

```

1 4.times { counter ->
2     task "task$counter" << {
3         println "I'm task number $counter"
4     }
5 }

```

运行结果如下：

```

1 xxx@XXX:~/$ gradle task1
2 :task1I'm task number 1
3
4 BUILD SUCCESSFUL
5
6 Total time: 1.397 secs
7

```

使用已存在任务task：

我们除过在上面定义任务task时指明依赖以外还可以通过API为任务加入一个依赖，如下：

```

1 4.times { counter ->
2     task "task$counter" << {
3         println "I'm task number $counter"
4     }
5 }
6 task0.dependsOn task2, task3

```

运行结果如下：

```

1 xxx@XXX:~/$ gradle task0:task0I'm task number 2
2 I'm task number 3I'm task number 0
3
4 BUILD SUCCESSFUL
5
6 Total time: 1.397 secs
7

```

或者我们还可以通过API为任务加入一些新行为，如下：

```

1 task hello << {
2

```

```

3     println 'Hello Earth'
4 }hello.doFirst {
5     println 'Hello Venus'
6 }hello.doLast {
7     println 'Hello Mars'
8 }
9 hello << {
10    println 'Hello Jupiter'
11 }
12

```

运行结果如下：

```

1 xxx@XXX:~/ $ gradle hello
2 :helloHello VenusHello EarthHello MarsHello JupiterBUILD SUCCESSFULTotal time: 1.397 secs
3

```

可以发现，doFirst和doLast可以被执行多次，<<操作符实质就是doLast。

任务task短标记：

我们可以通过美元符将一个task作为另一个task的属性，如下：

```

1 task hello << {
2     println 'Hello world!'
3 }
4 hello.doLast {
5     println "Greetings from the $hello.name task."
6 }

```

执行结果如下：

```

1 xxx@XXX:~/ $ gradle hello
2 :helloHello world!
3 Greetings from the hello task.
4
5 BUILD SUCCESSFULTotal time: 1.397 secs
6

```

可以看见，上面脚本中使用的name其实是任务的默认属性，代表当前任务的名称。

自定义任务task属性：

我们还可以给任务task加入自定义的属性，如下例子：

```

1 task myTask {
2     ext.myProperty = "myValue"
3 }
4
5 task printTaskProperties << {
6     println myTask.myProperty
7 }

```

执行结果如下：

```

1 xxx@XXX:~/ $ gradle printTaskProperties
2 :printTaskProperties
3 myValue
4
5 BUILD SUCCESSFULTotal time: 1.397 secs
6

```

定义默认任务task：

Gradle允许在脚本中定义一个或多个默认任务，如下：

```
1 defaultTasks 'clean', 'run' task clean << {
2     println 'Default Cleaning!'
3 }
4
5 task run << {
6     println 'Default Running!'
7 }
8
9 task other << {
10     println "I'm not a default task!"
11 }
12
```

执行结果如下：

```
1 xxx@XXX:~/ $ gradle
2 :clean, run
3 Default Cleaning!
4 Default Running!
5
6 BUILD SUCCESSFUL Total time: 1.397 secs
7
```

【工匠若水 <http://blog.csdn.net/yanbober> 转载请注明出处。 [点我开始Android技术交流](#)】

4 Gradle依赖管理基础

大多数项目都不是完全独立的，它们需要依赖其他项目进行编译等，Gradle允许你告诉它你项目的依赖关系，以便找到这些依赖关系，并在你的构建中维护这些依赖关系，依赖关系可能需要从远程的Maven等仓库中下载，也可能是在本地文件系统中，或者是通过多项目构建另一个构建，我们称这个过程为依赖解析。

Gradle依赖声明：

关于依赖声明不解释，直接给个例子，如下：

```
1 apply plugin: 'java'
2
3 repositories {
4     mavenCentral()
5 }
6
7 dependencies {
8     compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
9     testCompile group: 'junit', name: 'junit', version: '4.+
10 }
```

Gradle依赖配置：

在Gradle中依赖可以组合成configurations（配置），一个配置简单地讲就是一系列的依赖，通俗说也就是依赖配置；我们可以使用它们声明项目的外部依赖，也可以被用来声明项目的发布。下面我们给出几种Java插件中常见的配置，如下：

- compile

用来编译项目源代码的依赖；

- runtime

在运行时被生成的类需要的依赖，默认项，包含编译时的依赖；

- testCompile

编译测试代码依赖，默认项，包含生成的类运行所需的依赖和编译源代码的依赖；

- testRuntime

运行测试所需要的依赖，默认项，包含上面三个依赖；

各种各样的插件支持许多标准的配置，我们还可以定义自己的配置。

Gradle外部依赖：

我们可以用Gradle声明许多种依赖，其中有一种是外部依赖（external dependency），它是在当前构建之外的一种依赖，一般存放在远程（譬如Maven）或本地的仓库里。如下是一个外部依赖的例子：

```
1 dependencies {
2     compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
3 }
```

可以看见，引用一个外部依赖需要用到group、name、version属性。上面的写法还有一种简写，如下规则：

```
1 group:name:version
```

这是一个简写的例子：

```
1 dependencies {
2     compile 'org.hibernate:hibernate-core:3.6.7.Final'
3 }
```

Gradle仓库：

有了上面的外部依赖，你指定会想Gradle是咋找到那些外部依赖文件的。其实Gradle会在一个仓库（repository）里找这些依赖文件，仓库其实就是很多依赖文件的集合服务器，他们通过group、name、version进行归类存储，好在Gradle可以解析好几种不同的仓库形式（譬如Maven等），但是Gradle默认不提前定义任何仓库，我们必须手动在使用外部依赖之前定义自己的仓库。

下面是一个使用MavenCentral仓库的例子：

```
1 repositories {
2     mavenCentral()
3 }
```

这是一个使用远程Maven仓库的例子：

```
1 repositories {
2     maven {
3         url "http://repo.mycompany.com/maven2"
4     }
5 }
```

这是一个使用本地文件系统里库的例子：

```

1 repositories {
2     ivy {
3         // URL can refer to a local directory
4         url "../local-repo"
5     }
6 }

```

当然了，一个项目可以有好几个库，Gradle会根据依赖定义的顺序在各个库里寻找它们，在第一个库里找到了就不会再在第二个库里找它了，否则在第二个库找。

Gradle发布artifacts：

依赖配置也可以用来发布文件，我们可以通过在uploadArchives任务里加入仓库来完成。下面是一个发布到Maven 库的例子，Gradle将生成和上传pom.xml，如下：

```

1 apply plugin: 'maven'
2
3 uploadArchives {
4     repositories {
5         mavenDeployer {
6             repository(url: "file://localhost/tmp/myRepo/")
7         }
8     }
9 }

```

【工匠若水 <http://blog.csdn.net/yanbober> 转载请注明出处。点我开始Android技术交流】

5 Gradle命令

多任务调用命令：

```

1 gradle task1 task2 [...]

```

排除任务命令：

```

1 gradle -x task1 task2 [...]

```

失败后继续执行构建命令：

只要有任务调用失败Gradle默认就会中断执行，我们可以使用-continue选项在一次调用中不中断执行，然后发现所有失败原因。

简化任务名命令：

当我们调用某个任务时如果名字太长我们可以采用简化操作，但是必须保证可以唯一区分出该任务的字符，譬如：

```

1 //简写
2 gradle -x t1
3 //替换
4 gradle -x task1

```

选择执行构建命令：

调用gradle命令默认会构建当前目录下的build.gradle文件，我们可以使用-b参数选择其他目录的构建文件且当使用此参数时settings.gradle将不会生效。如下：

```
1 //选择文件构建subdir/myproject.gradle
2 task hello << {
3     println "using build file '$buildFile.name' in '$buildFile.parentFile.name'."
4 }
```

执行过程：

```
1 xxx@XXX:~/$ gradle -b subdir/myproject.gradle hello
2 :hello
3 using build file 'myproject.gradle' in 'subdir'.
4
5 BUILD SUCCESSFULTotal time: 1.397 secs
6
```

此外我们还可以使用-p参数来指定构建的目录，譬如在多项目构建中可以用-p替代-b参数。如下执行过程：

```
1 xxx@XXX:~/$ gradle -p subdir hello
2 :hello
3 using build file 'build.gradle' in 'subdir'.
4
5 BUILD SUCCESSFULTotal time: 1.397 secs
6
```

获取构建信息：

- gradle projects命令：列出子项目名称列表。
- gradle tasks命令：列出项目中所有任务。
- gradle help -task someTask命令：可以显示指定任务的详细信息。
- gradle dependencies命令：列出项目的依赖列表，所有依赖会根据任务区分，以树型结构展示。

【工匠若水 <http://blog.csdn.net/yanbober> 转载请注明出处。[点我开始Android技术交流](#)】

6 编写Gradle脚本

Gradle是以Groovy语言为基础，基于DSL语法的自动化构建工具，一个构建脚本能够包含任何Groovy语言元素，每个脚本都是UTF-8编码的文件。

6-1 Project对象API

前面我们说过，Gradle在构建脚本中定义了一个project，对于构建脚本中每个project其实Gradle都创建了一个 Project类型的对象来关联，当构建脚本执行时它会去配置所关联的Project对象；构建脚本中每个被调用的方法和属性都委托给了当前Project对象。

如下我们看一个使用Project属性的例子：

```
1 println name
2 println project.name
```

上面两个println语句的输出是一样的；由于name属性没有在当前脚本中定义，所以可以像第一个那样使用自动委托，通常我们使用第二中写法。

Project对象提供了一些标准的属性，我们可以在构建脚本中很方便的使用他们，如下：

Name	Type	Default Value
project	Project	Project实例对象
name	String	项目目录的名称
path	String	项目的绝对路径

description	String	项目描述
projectDir	File	包含构建脚本的目录
build	File	projectDir/build
group	Object	未具体说明
version	Object	未具体说明
ant	AntBuilder	Ant实例对象

具体关于Project的方法详情参阅Project的API文档。这里我们给出Project的apply方法的一个例子，如下：

```

1 //加载一个gradle文件
2 apply from: rootProject.getRootDir().getAbsolutePath() + "/common.gradle"
```

6-2 Script对象API

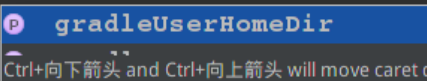
当Gradle执行一个脚本时它会将这个脚本编译为实现了Script的类（在上篇博客《Groovy脚本基础全攻略》Groovy的本质编译class代码那块有介绍），也就是说所有的属性和方法都是在Script的接口中声明。

6-3 Gradle对象API

关于Gradle对象的详细属性和API介绍[点我即可](#)。这里直接给出一个使用Gradle对象的例子，如下：

```

2 println "gradleHomeDir:" + gradle.gradleHomeDir
3 println "Test:" + gradle.
4
```



```

le Console
gradleHomeDir:/usr/myself/android-studio/gradle/gradle-2.4
```

6-4 Gradle变量声明

在Gradle脚本中有两种类型的变量可以声明，如下：

- 局部变量
- 扩展变量

局部变量使用关键字def声明，它只在声明的地方可见，如下：

```

1     def dest = "dest"
2
3     task copy(type: Copy) {
4         form "source"
5         into dest
6
7     }
```

在Gradle中所有被增强的对象可以拥有自定义属性（譬如projects、tasks、source sets等），使用ext扩展块可以一次添加多个属性。如下：

```

1 apply plugin: "java"
2
3 ext {
4     springVersion = "3.1.0.RELEASE"
5     emailNotification = "build@master.org"
6 }
7
8 sourceSets.all { ext.purpose = null }
9
```

```

10
11 sourceSets {
12     main {
13         purpose = "production"
14     }
15     test {
16         purpose = "test"
17     }
18     plugin {
19         purpose = "production"
20     }
21 }
22
23 task printProperties << {
24     println springVersion
25     println emailNotification
26     sourceSets.matching { it.purpose == "production" }.each { println it.name}
27 }

```

上面我们用一个ext扩展块向Project对象添加两个扩展属性，当这些扩展属性被添加后，它们就像预定义的属性一样可以被读写。

6-5 Gradle中Groovy使用

这个没啥说的，具体可以参考《[Groovy脚本基础全攻略](#)》这篇博客，里面有详细介绍。我们这里粗略总结回忆一下即可：

- Groovy会自动将一个属性的引用转换为相应的getter/setter方法。
- Groovy调用方法时圆括号可有可无。
- Groovy为List和Map集合提供了一些操作捷径，譬如apply plugin: 'java' 中的plugin: 'java' 其实就是Groovy中的Map，apply是一个方法，省略了括弧而已。

哎呀，详细的还是去看前一篇博客吧。

【工匠若水 <http://blog.csdn.net/yanbober> 转载请注明出处。 [点我开始Android技术交流](#)】

7 Gradle文件操作基础

实际使用Gradle过程中大多数时候需要操作文件，好在Gradle给我们提供了一些API来快捷处理。

定位文件：

我们可以使用Project.file()方法来定位一个文件获取File对象（详情参考Project的API），如下：

```

1 //相对路径File configFile = file('src/config.xml')
2 //绝对路径File configFile = file(configFile.absolutePath)
3 //项目路径的文件对象 File configFile = file(new File('src/config.xml'))
4

```

可以从Project的API发现file()方法能够接收任何形式的对象参数，它会将参数值转换为一个绝对文件对象，通常我们可以传一个String或File实例；如果传的路径是绝对路径，则会被直接构造为一个文件实例，否则会被构造为项目目录加上传递目录的文件对象；当然了，file()方法还能识别URL（譬如file:/some/path.xml等）。

文件集合：

文件集合其实是一组文件，Gradle使用FileCollection接口表示文件集合，Gradle API中许多类都实现了这个接口，譬如dependency configurations等。获取FileCollection实例的一种方法是Project.files()，我们可以传递任何数量的对象参数。如下：

```

1 FileCollection collection = files('src/file1.txt',
2                                 new File('src/file2.txt'),
3                                 ['src/file3.txt', 'src/file4.txt'])

```

使用迭代操作还能将其转换为其他的一些类型，同时我们还可以使用+操作将两个文件集合合并，使用-操作对一个文件集合做减法。如下例子：

```

1 // 对文件集合进行迭代
2 collection.each {File file ->
3     println file.name
4 }
5
6 // 转换文件集合为其他类型
7 Set set = collection.files
8 Set set2 = collection as Set
9 List list = collection as List
10 String path = collection.asPath
11 File file = collection.singleFile
12 File file2 = collection as File
13
14 // 增加和减少文件集合
15 def union = collection + files('src/file3.txt')
16 def different = collection - files('src/file3.txt')

```

我们也可以向files()方法传递闭包或者可回调的实例参数，当查询集合的内容时就会调用它，然后将返回值转换为一些文件实例，返回值可以是files()方法支持的任何类型的对象。如下例子：

```

1 task list << {
2     File srcDir
3
4     // 使用闭包创建一个文件集合
5     collection = files { srcDir.listFiles() }
6
7     srcDir = file('src')
8     println "Contents of $srcDir.name"
9     collection.collect { relativePath(it) }.sort().each { println it }
10
11     srcDir = file('src2')
12     println "Contents of $srcDir.name"
13     collection.collect { relativePath(it) }.sort().each { println it }
14 }

```

文件树：

文件树可以代表一个目录树结构或一个ZIP压缩文件的内容，FileTree继承自FileCollection，所以我们可以像处理文件集合一样处理文件树，使用Project.fileTree()方法可以得到FileTree实例，它会创建一个基于基准目录的对象。如下：

```

1 //以一个基准目录创建一个文件树
2 FileTree tree = fileTree(dir: 'src/main')
3
4 // 添加包含和排除规则
5 tree.include '**/*.java'
6 tree.exclude '**/Abstract*'
7 // 使用路径创建一个树
8 tree = fileTree('src').include('**/*.java')
9
10 // 使用闭包创建一个数
11 tree = fileTree('src') {
12     include '**/*.java'
13 }
14
15 // 使用map创建一个树
16 tree = fileTree(dir: 'src', include: '**/*.java')
17 tree = fileTree(dir: 'src', includes: ['**/*.java', '**/*.xml'])

```

```

18 tree = fileTree(dir: 'src', include: '**/*.java', exclude: '**/*test**')
19
20 // 遍历文件树
21 tree.each {File file ->
22     println file
23 }
24
25 // 过滤文件树
26 FileTree filtered = tree.matching {
27     include 'org/gradle/api/**'
28 }
29
30 // 合并文件树A
31 FileTree sum = tree + fileTree(dir: 'src/test')
32
33 // 访问文件数的元素
34 tree.visit {element ->
35     println "$element.relativePath => $element.file"
36 }
37

```

我们还可以使用ZIP或TAR等压缩文件的内容作为文件树，Project.zipTree()和Project.tarTree()方法可以返回一个FileTree实例。如下：

```

1 // 使用路径创建一个ZIP文件
2 FileTree zip = zipTree('someFile.zip')
3
4 // 使用路径创建一个TAR文件
5 FileTree tar = tarTree('someFile.tar')
6
7 //TarTree可以根据文件扩展名得到压缩方式，如果我们想明确的指定压缩方式则可以如下操作
8 FileTree someTar = tarTree(resources.gzip('someTar.ext'))

```

指定输入文件：

Gradle中有些对象的属性可以接收一组输入文件，譬如JavaCompile任务的source属性（定义编译的源文件）。如下：

```

1 //使用一个File对象设置源目录
2 compile {
3     source = file('src/main/java')
4 }
5
6 //使用一个字符路径设置源目录
7 compile {
8     source = 'src/main/java'
9 }
10
11 //使用一个集合设置多个源目录
12 compile {
13     source = ['src/main/java', '../shared/java']
14 }
15
16 //使用FileCollection或者FileTree设置源目录
17 compile {
18     source = fileTree(dir: 'src/main/java').matching {include 'org/gradle/api/**'}
19 }
20
21 //使用闭包设置源目录
22 compile {
23     source = {
24         // Use the contents of each zip file in the src dirfile('src').listFiles().findAll {it.name.endsWith('.zip')}.collect
25     }
26 }
27
28 compile {
29     //使用字符路径添加源目录source 'src/main/java', 'src/main/groovy' //使用File对象添加源目录source file('../shared/java')
30     //使用闭包添加源目录source { file('src/test/').listFiles() }
31 }

```

复制文件：

我们可以使用复制任务（Copy）进行文件复制操作，复制任务扩展性很强，它可以过滤复制文件的内容，使用复制任务要提供想要复制的源文件和一个目标目录，如果要指定文件被复制时的转换方式则可以使用复制规则，复制规则是一个CopySpec接口的实现，我们使用CopySpec.from()方法指定源文件，CopySpec.into()方法指定目标目录即可。如下：

```
1 task copyTask(type: Copy) {
2     from 'src/main/webapp' into 'build/ explodedWar'
3 }
4
5 task anotherCopyTask(type: Copy) {
6     //复制src/main/webapp目录下的所有文件from 'src/main/webapp' //复制一个单独文件from 'src/staging/index.html' //复制一个任
7     //显式使用任务的outputs属性复制任务的输出文件from copyTaskWithPatterns.outputs
8     //复制一个ZIP压缩文件的内容from zipTree('src/main/assets.zip')
9     //指定目标目录into { getDestDir() }
10 }
11
12 task copyTaskWithPatterns(type: Copy) {
13     from 'src/main/webapp' into 'build/ explodedWar' include '**/*.html' include '**/*.jsp'
14     exclude { details -> details.file.name.endsWith('.html') &&
15         details.file.text.contains(' staging') }
16 }
17
18 task copyMethod << {
19     copy {
20         from 'src/main/webapp' into 'build/ explodedWar' include '**/*.html' include '**/*.jsp'
21     }
22 }
23
24 //在复制时重命名文件
25 task rename(type: Copy) {
26     from 'src/main/webapp' into 'build/ explodedWar' //使用闭包映射文件名rename { String fileName ->
27         fileName.replace('-staging-', '')
28     }
29     // 使用正则表达式映射文件名rename '(.+)-staging-(.+)', '$1$2' rename(/(.+)-staging-(.+)/, '$1$2')
30 }
31
```

文件同步任务：

同步任务（Sync）继承自复制任务（Copy），当执行时会复制源文件到目标目录，然后从目标目录删除所有非复制文件。如下：

```
1 task libs(type: Sync) {
2     from configurations.runtime
3     into "$buildDir/libs"
4 }
```

创建归档文件：

使用归档任务可以创建Zip、Tar、Jar、War、Ear等归档文件，如下：

```
1 apply plugin: 'java'
2
3 task zip(type: Zip) {
4     from 'src/dist' into('libs') {
5         from configurations.runtime
6     }
7 }
8
```


关于文件操作的其他请参考API文档。

【工匠若水 <http://blog.csdn.net/yanbober> 转载请注明出处。 [点我开始Android技术交流](#)】

8 Gradle插件

8-1 Gradle插件概述

插件基础：

关于Gradle支持的插件可以[点我搜索](#)。其实Gradle的核心只是一个空空的框架，所谓的Gradle构建便捷脚本其实都是由插件提供支持的，插件添加了新的任务。在Gradle中一般有两种类型的插件，如下：

- 脚本插件

是额外的构建脚本，它会进一步配置构建，通常会在构建内部使用。脚本插件可以从本地文件系统或远程获取，如果从文件系统获取则是相对于项目目录，如果是远程获取则是由HTTP URL指定。

- 二进制插件

是实现了Plugin接口的类，并且采用编程的方式来操纵构建。

插件需要通过Project.apply()方法完成声明应用，相同的插件可以应用多次。如下例子：

```
1 //脚本插件
2 apply from: 'other.gradle' //二进制插件
3 apply plugin: 'java'
```

插件还可以使用插件ID，插件的id作为给定插件的唯一标识符，我们可以给插件注册一个缩写字符的id。譬如下面例子：

```
1 //通过Java插件的id进行引用
2 apply plugin: JavaPlugin
```

使用构建脚本块应用插件：

我们可以向构建脚本中加入插件的类路径然后再应用插件和使用插件的任务，如下：

```
1 buildscript {
2     repositories {
3         jcenter()
4     }
5     dependencies {
6         classpath "com.jfrog.bintray.gradle:gradle-bintray-plugin:0.4.1"
7     }
8 }
9
10 apply plugin: "com.jfrog.bintray"
```

Gradle插件拓展：

可以看见，Gradle其实是依托于各种插件壮大的，譬如Java插件用来构建Java工程，Android插件用来构建打包Android工程，我们只需要选择合适的插件即可，插件会为我们提供丰富的任务用来快捷处理构建，具体详情参考各插件API即可。

8-2 Gradle的Java插件构建实例

上面说了，插件是Gradle的扩展，它会通过某种方式配置我们的项目（譬如加入一些task）；Gradle自带许多插件，我们也可以编写自己的插件然后开

源.，Java 插件就是这样的一个插件，该插件已经给项目定义了默认的参数（譬如Java源文件位置），所以通常我们不需要在脚本中加入太多东西。

单个基础Java项目构建：

```
1 //把Java插件加入到项目中，也就是许多预定制的任务被自动加入到了项目里
2 apply plugin: 'java'
```

加入上面插件以后Gradle默认希望能在src/main/java路径下找到源代码，在 src/test/java路径下找到测试代码，任何src/main/resources路径的文件都会被包含在JAR文件里，任何src/test/resources路径的文件都会被加入到classpath中以运行测试代码，所有的输出文件将会被创建在构建目录里，JAR文件存放在 build/libs文件夹里。

加入Java插件后我们可以通过gradle tasks命令来列出项目的所有任务，这样就可以知道Java插件添加了哪些task。常用的task如下：

- build task
当运行gradle build命令时Gradle将会编译和测试你的代码，并且创建一个包含类和资源的JAR文件。
- clean task
当运行gradle clean命令时Gradle将会删除build生成的目录和所有生成的文件。
- assemble task
当运行gradle assemble命令时Gradle将会编译并打包代码，但是并不运行单元测试。
- check task
当运行gradle check命令时Gradle将会编译并测试你的代码，其他的插件会加入更多的检查步骤。

单个具有外部依赖的Java项目构建：

当然了，一个Java项目可能会有许多外部依赖（即调用第三方JAR），为了在项目里引用这些 JAR包，我们需要告诉Gradle去哪里找他们，好在Gradle支持许多仓库，这些仓库可以被用来提取或者放置依赖，我们可以很方便的从这些仓库中取得第三方Jar包。如下：

```
1 //加入Maven仓库
2 repositories {
3     mavenCentral()
4 }
```

接着加入一些编译阶段来自于mavenCentral仓库的依赖，如下：

```
1 dependencies {
2     //编译阶段
3     compile group: 'commons-collections', name: 'commons-collections', version: '3.2' //测试编译阶段
4     testCompile group: 'junit', name: 'junit', version: '4.+'
5 }
6
```

定制构建项目：

Java插件给项目加入了一些属性，这些属性已经被赋予了默认的值且已经够我们日常使用了，如果我们觉得这些默认属性不好也可以自己修改。如下：

```
1 //定制 MANIFEST.MF 文件
2 sourceCompatibility = 1.5version = '1.0'
3 jar {
4     manifest {
5         attributes 'Implementation-Title': 'Gradle Quickstart', 'Implementation-Version': version
6     }
7 }
```

```
8 }
```

默认Java插件加入的任务是常规性的任务，但是我们可以定制任务，譬如我们可以设置一个任务的属性、在任务中加入行为、改变任务的依赖、完全重写一个任务等。如下：

```
1 //测试阶段加入一个系统属性
2 test {
3     systemProperties 'property': 'value'
4 }
```

关于哪些属性是可用的问题，我们可以使用gradle properties命令列出项目的所有属性。

发布JAR文件：

通常JAR文件需要在某个地方发布，我们可以通过Gradle方便的进行发布，譬如下面例子将发布到一个本地的目录，如下：

```
1 //uploadArchives task
2 uploadArchives {
3     repositories {
4         flatDir {
5             dirs 'repos'
6         }
7     }
8 }
```

多Java项目构建：

在Gradle中为了定义一个多项目构建我们需要创建一个设置文件（settings.gradle），设置文件放在源代码的根目录，它用来指定要包含哪个项目且名字必须叫做settings.gradle。如下例子：

```
1 //多项目工程结构树：
2 multiproject/
3   api/
4   services/webservice/
5   shared/
```

```
1 //多项目构建settings.gradle文件include "shared", "api", "services:webservice", "services:shared"
2
```

对于大多数多项目构建有一些配置对所有项目都是通用的，所以我们将在根项目里定义一个这样的通用配置（配置注入技术 configuration injection）。根项目就像一个容器，subprojects方法遍历这个容器的所有元素并且注入指定的配置。如下：

```
1 //多项目构建通用配置
2 subprojects {
3     apply plugin: 'java'
4     apply plugin: 'eclipse-wtp'
5
6     repositories {
7         mavenCentral()
8     }
9
10    dependencies {
11        testCompile 'junit:junit:4.11'
12    }
13
14    version = '1.0'
15
16    jar {
17        manifest.attributes provider: 'gradle'
```

```
18     }
19 }
```

可以看见，上面通用配置把Java插件应用到了每一个子项目中。

我们还可以在同一个构建里加入项目之间的依赖，这样可以保证他们的先后关系。如下：

```
1 //api/build.gradle
2 dependencies {
3     compile project(':shared')
4 }
```

至此基础的Java插件使用就OK了，深入的请自行查看API。

【工匠若水 <http://blog.csdn.net/yanbober> 转载请注明出处。 [点我开始Android技术交流](#)】

4 Gradle基础总结

到此Gradle的基础知识就完全介绍完了，我们对Gradle的框架也有了一个直观的认识。其实编写Gradle无非也就是对类的属性和方法进行调运操作，至于如何调运操作依据具体插件而异，核心的生命周期和几个对象实例搞明白基本上就能驾驭Gradle脚本了，其他的无非就是熟练度和API查找。

来源：<http://blog.csdn.net/yanbober/article/details/49314255>