Gradle学习系列

文档来自：<http://www.davenkin.me/post/2013-11-12/gradle-first-try>

git clone <https://github.com/davenkin/gradle-learning.git>

# Gradle快速入门

和Maven一样，Gradle只是提供了构建项目的一个框架，真正起作用的是Plugin。Gradle在默认情况下为我们提供了许多常用的Plugin，其中便包括构建Java项目的Plugin，还有War，Ear等。与Maven不同的是，Gradle不提供内建的项目生命周期管理，只是Java的Plugin向Project中添加了许多Task，这些Task依次执行，为我们营造了一种如同Maven般项目构建周期。

现在我们都在谈领域驱动设计，而Gradle本身的领域对象主要有Project和Task。Project为Task提供了执行上下文，所有的Plugin要么向Project中添加了用于配置的Property，要么向Project中添加不同的Task。一个Task表示一个逻辑上较为独立的执行过程，比如编译Java源代码，拷贝文件，打包Jar文件，甚至可以是执行一个系统命令或者调用Ant。另外，一个Task可以读取和设置Project的Property以完成特定的操作。

让我们来看一个最简单的Task，创建一个build.gradle文件，内容如下：

task helloWorld << {

println "Hello World!"

}

这里的“<<”表示向helloWorld中加入执行代码，其实就是groovy代码。Gradle向我们提供了一整套DSL，所以在很多时候我们写的代码似乎已经脱离了groovy，但是在底层依然是执行的groovy。比如上面的task关键字，其实就是一个groovy中的方法，而大括号之间的内容则表示传递给task方法的一个闭包。除了“<<”之外，我们还很多种方式可以定义一个Task，我们将在系列后续的文章中讲到。

在与build.gradle相同的目录下执行：

gradle helloWorld

命令行输出如下：

:helloWorld

Hello World!

BUILD SUCCESSFUL

Total time: 2.544 secs

在默认情况下，Gradle将当前目录下的build.gradle文件作为项目的构建文件。在上面的例子中，我们创建了一个名为helloWorld的Task，在执行gradle命令时，我们指定执行该helloWorld Task。这里的helloWorld是一个DefaultTask类型的对象，这也定义一个Task时的默认类型，当然我们也可以显式地声明一个Task的类型，甚至可以自定义一个Task类型（我们将在本系列的后续文章中讲到）。

比如，我们可以定义一个用于文件拷贝的Task：

task copyFile(type: Copy) {

from 'xml'

into 'destination'

}

以上copyFile将xml文件夹中的所有内容拷贝到destination文件夹中。这里的两个文件夹都是相对于当前Project而言的，即build.gradle文件所在的目录。

Task之间可以存在依赖关系，比如taskA依赖于taskB，那么在执行taskA时，Gradle会先执行taskB，然后再执行taskB。声明Task依赖关系的一种方式是在定义一个Task的时候：

task taskA（dependsOn:taskB）<< {

println "executing taskA"

}

Gradle在默认情况下为我们提供了几个常用的Task，比如查看Project的Properties，显示当前Project中定义的所有Task等。可以通过一下命令查看Project所有的Task：

gradle tasks

输出如下：

:tasks

------------------------------------------------------------

All tasks runnable from root project

------------------------------------------------------------

Build Setup tasks

-----------------

setupBuild - Initializes a new Gradle build. [incubating]

wrapper - Generates Gradle wrapper files. [incubating]

Help tasks

----------

dependencies - Displays all dependencies declared in root project 'gradle-blog'.

dependencyInsight - Displays the insight into a specific dependency in root project 'gradle-blog'.

help - Displays a help message

projects - Displays the sub-projects of root project 'gradle-blog'.

properties - Displays the properties of root project 'gradle-blog'.

tasks - Displays the tasks runnable from root project 'gradle-blog'.

Other tasks

-----------

copyFile

helloWorld

To see all tasks and more detail, run with --all.

BUILD SUCCESSFUL

Total time: 2.845 secs

可以看到，除了我们自己定义的copyFile和helloWorld之外，Gradle还默认为我们提供了dependencies,projects和properties等Task。dependencies用于显示Project的依赖信息，projects用于显示所有Project，包括根Project和子Project，而properties则用于显示一个Project所包含的所有Property。

在默认情况下，Gradle已经为一个Project添加了很多Property，我们可以调用以下命令进行查看：

gradle properties

输出如下：

:properties

------------------------------------------------------------

Root project

------------------------------------------------------------

allprojects: [root project 'gradle-blog']

ant: org.gradle.api.internal.project.DefaultAntBuilder@1342097

antBuilderFactory: org.gradle.api.internal.project.DefaultAntBuilderFactory@1eeb95e

artifacts: org.gradle.api.internal.artifacts.dsl.DefaultArtifactHandler@d2ad3c

asDynamicObject: org.gradle.api.internal.ExtensibleDynamicObject@198339a

buildDir: /home/davenkin/Desktop/gradle-blog/build

buildFile: /home/davenkin/Desktop/gradle-blog/build.gradle

buildScriptSource: org.gradle.groovy.scripts.UriScriptSource@13e9963

buildscript: org.gradle.api.internal.initialization.DefaultScriptHandler@89eb77

childProjects: {}

class: class org.gradle.api.internal.project.DefaultProject\_Decorated

components: []

configurationActions: org.gradle.configuration.project.DefaultProjectConfigurationActionContainer@b07848

configurations: []

convention: org.gradle.api.internal.plugins.DefaultConvention@11492ed

copyFile: task ':copyFile'

defaultTasks: []

dependencies: org.gradle.api.internal.artifacts.dsl.dependencies.DefaultDependencyHandler@1a5469e

dependsOnProjects: []

depth: 0

description: null

ext: org.gradle.api.internal.plugins.DefaultExtraPropertiesExtension@1b5d53a

extensions: org.gradle.api.internal.plugins.DefaultConvention@11492ed

fileResolver: org.gradle.api.internal.file.BaseDirFileResolver@19d44df

gradle: build 'gradle-blog'

group:

helloWorld: task ':helloWorld'

implicitTasks: [task ':dependencies', task ':dependencyInsight', task ':help', task ':projects', task ':properties', task ':tasks']

inheritedScope: org.gradle.api.internal.ExtensibleDynamicObject$InheritedDynamicObject@ab8796

logger: org.gradle.api.logging.Logging$LoggerImpl@1b33087

logging: org.gradle.logging.internal.DefaultLoggingManager@13f67a9

metaClass: org.codehaus.groovy.runtime.HandleMetaClass@11cac41[groovy.lang.MetaClassImpl@11cac41[class org.gradle.api.internal.project.DefaultProject\_Decorated]]

module: org.gradle.api.internal.artifacts.ProjectBackedModule@8c0162

name: gradle-blog

parent: null

parentIdentifier: null

path: :

plugins: [org.gradle.api.plugins.HelpTasksPlugin@7359f7]

project: root project 'gradle-blog'

projectDir: /home/davenkin/Desktop/gradle-blog

projectEvaluationBroadcaster: ProjectEvaluationListener broadcast

projectEvaluator: org.gradle.configuration.project.LifecycleProjectEvaluator@1654982

projectRegistry: org.gradle.api.internal.project.DefaultProjectRegistry@1715c20

properties: {...}

repositories: []

resources: org.gradle.api.internal.resources.DefaultResourceHandler@1cd0037

rootDir: /home/davenkin/Desktop/gradle-blog

rootProject: root project 'gradle-blog'

scriptPluginFactory: org.gradle.configuration.DefaultScriptPluginFactory@10951aa

services: ProjectInternalServiceRegistry

standardOutputCapture: org.gradle.logging.internal.DefaultLoggingManager@13f67a9

state: project state 'EXECUTED'

status: release

subprojects: []

tasks: [task ':copyFile', task ':helloWorld']

version: unspecified

BUILD SUCCESSFUL

Total time: 2.667 secs

在以上Property中，allprojects表示所有的Project，这里之包含一个根Project，在多项目构建中，它将包含多个Project；buildDir表示构建结果的输出目录；我们自己定义的helloWorld和copyFile也成为了Project中的Property。另外，Project还包括用于执行Ant命令的DefaultAntBuilder（Property名为ant）和Project的描述属性description。

# 创建Task的多种方法

Gradle的Project和build.gradle存在着一对一的关系。

一个Project从本质上说只是含有多个Task的容器。

Any property or method which your script uses is delegated through to the associated Project object.

The tasks of the project. A method is added for each task, using the name of the task as the method name and taking a single closure parameter. The method calls the Task.configure(groovy.lang.Closure) method for the associated task with the provided closure. For example, if the project has a task called compile, then a method is added with the following signature: void compile(Closure configureClosure).代理

====================================

Gradle从入门到提高系列（二）——创建Task的多种方法

Gradle的Project从本质上说只是含有多个Task的容器，一个Task与Ant的Target相似，表示一个逻辑执行单元。我们可以通过

很多种方式定义Task，所有的Task都存放在Project的TaskContainer中。

（1）调用Project的task()方法创建Task

在使用Gradle时，创建Task最常见的方式便是：

task hello1 << {

println 'hello1'

}

这里的“<<”表示追加的意思，即向hello中加入执行过程。我们还可以使用doLast来达到同样的效果：

task hello2 {

doLast {

println 'hello2'}

}

另外，如果需要向Task的最前面加入执行过程，我们可以使用doFirst：

task hello3 {

doLast {

println 'hello3'}

}

在上面的3个例子中，Gradle的DSL向我们展示了一种非常自然的风格来创建Task，而事实上这些都只是一种内部DSL，也即必须符合groovy的语法要求。

上面的task关键字实际上是一个方法调用，该方法属于Project。Project中存在多个重载的task()方法完成多种类型的Task创建。

以上我们自定义的3个Task都位于TaskContainer中，Project中的tasks即表示该TaskContainer。为此，我们可以新建一个Task来显示这些信息：

task showTasks {

println tasks.class

println tasks.size()

}

将以上4个Task放在同一个build.gradle中，再执行gradle showTasks，命令行输出如下：

...

class org.gradle.api.internal.tasks.DefaultTaskContainer\_Decorated

4

...

上面的DefaultTaskContainer\_Decorated表示tasks类型，而4表示该TaskContainer中包含有4个自定义的Task——包括showTasks。

（2）通过TaskContainer的create()方法创建Task

在上文中我们讲到，通过task()方法创建的Task都被存放在了TaskContainer中，而Project又维护了一个TaskContainer类型的属性tasks，那么我们完全可以直接向TaskContainer里面添加Task。查查TaskContainer的API文档可以发现，TaskContainer向我们提供了大量的create()方法用于添加Task。

tasks.create(name: 'hello4') << {

println 'hello4'

}

（3）声明Task之间的依赖关系

Task之间是可以存在依赖关系，比如TaskA依赖TaskB，那么在执行TaskA时，Gradle会先执行TaskB，再执行TaskA。我们可以在定义一个Task的同时声明它的依赖关系：

task hello5(dependsOn:hello4) << {

println 'hello5'

}

当然，我们也可以在定义Task之后再声明依赖：

task hello6 << {

println 'hello6'

}

hello6.dependsOn hello5

（4）配置Task

一个Task出了执行操作之外，还包含有多个Property，其中有Gradle为每个Task默认定义的Property，比如description，logger等，每一个特定的Task类型还可以含有特定的Property，比如Copy的from和to等，当然，我们还可以动态地向Task中加入额外的Property。在执行一个Task之前，我们通常都需要先设定Property的值，Gradle提供了多种方法设置Task的Property值。

首先，我们可以在定义Task的时候对Property进行配置：

task hello7 << {

description = "this is hello7"

println description

}

我们还可以通过闭包的方式来配置一个已有的Task：

task hello8 << {

println description

}

hello8 {

description = "this is hello8"

}

需要注意的是，对hello8的description的设置发生在定义该Task之后，在执行gradle hello8时，命令行依然可以打印出正确的“this is hello8”，这是因为Gradle在执行Task时分为两个阶段，首先是配置阶段，然后才是实际执行阶段。所以在执行hello8之前，Gradle会扫描整个build.gradle文档，将hello8的description设置为“this is hello8”，再执行hello8，此时hello8的description已经包含了设置后的值。

另外，我们还可以通过调用Task的configure()方法完成Property的设置：

task hello9 << {

println description

}

hello9.configure {

description = "this is hello9"

}

实际上，通过闭包的方式在内部也是通过调用Task的configure()方法完成的，我们将在后续的博文中详细地讲到。

# 读懂Gradle语法

Gradle从入门到提高系列（三）—— 读懂Gradle

Gradle是一种声明式的构建工具。在执行时，Gradle并不会一开始便顺序执行build.gradle文件中的内容，而是分为两个阶段，第一个阶段是配置阶段，然后才是实际的执行阶段。在

配置阶段，Gradle将读取所有build.gradle文件的所有内容来配置Project和Task等，比如设置Project和Task的Property，处理Task之间的依赖关系等。

虽然很多时候我们只需要照着网上的例子写自己的DSL语句就行了，但是此时我们所知道的也就只有这么多。另一方面，如果我们能够了解Gradle DSL的内部工作机制，那么我们便可以达到

举一反三的效果。

在前面的文章中我们讲到，Gradle的DSL只是Groovy语言的内部DSL，也即必须遵循Groovy的语法规则。现在，让我们先来看看以下非常简单的Task：

task showDescription1 << {

description = 'this is task showDescription'

println description

}

task showDescription2 << {

println description

}

showDescription2.description = 'this is task showDescription'

task showDescription3 << {

println description

}

showDescription3 {

description = 'this is task showDescription'

}

以上3个Task完成的功能均相同，即先设置Task的description属性，在将其输出到命令行。但是，他们对description的设置方式是不同的。对于showDescription1，我们在定义一个Task的时候

便对该Task的description属性设置；对于showDescription2，其本身便是Project的一个Property；而对于showDescription3，我们是在一个和它同名的方法中设置description。

事实上，对于每一个Task，Gradle都会在Project中创建一个同名的Property，所以我们可以将该Task当作Property来访问，showDescription2便是这种情况。

另外，Gradle还会创建一个同名的方法，该方法接受一个闭包，我们可以使用该方法来配置Task，showDescription3便是这种情况。

要读懂Gradle，我们首先需要了解Groovy语言中的两个概念，一个Groovy中的Bean概念，一个是Groovy闭包的delegate机制。

Groovy中的Bean和Java中的Bean有一个很大的不同，即Groovy为每一个字段都会自动生成getter和setter，并且我们可以通过像访问字段本身一样调用getter和setter，比如：

class GroovyBeanExample {

private String name

}

def bean = new GroovyBeanExample()

bean.name = 'this is name'

println bean.name

我们看到，GroovyBeanExample只定义了一个私有的name属性，并且没有getter和setter。但是在使用时，我们可以直接对name进行访问，无论时读还是写。事实上，我们并不是

在直接访问name属性，当我们执行"bean.name = 'this is name'"时，我们实际调用的是"bean.setName('this is name')"，而在调用"println bean.name"时，我们实际调用的是

"println bean.getName()"。这里的原因在于，Groovy动态地位name创建了getter和setter，采用像直接访问的方式的目的是为了增加代码的可读性，是它更加自然，而在内部，Groovy依然

是在调用setter和getter方法。这样，我们便可以理解上面对showDescription2的description设置原理。

另外，Gradle大量地使用了Groovy闭包的delegate机制。简单来说，delegate机制可以使我们将一个闭包中的执行代码的作用对象设置成任意其他对象。比如：

class Child {

private String name

}

class Parent {

Child child = new Child();

void configChild(Closure c) {

c.delegate = child

c.setResolveStrategy Closure.DELEGATE\_FIRST

c()

}

}

def parent = new Parent()

parent.configChild {

name = "child name"

}

println parent.child.name

在上面的例子中，当我们调用configChild()方法时，我们并没有指出name属性是属于Child的，但是它的确时设置Child的name。事实上光从该方法的调用中，我们根本不知道name是属于

哪个对象的，你可能会认为它是属于Parent的。真实情况是，在默认情况下，name的确被认为是属于Parent的，但是我们在configChild()方法的定义中做了手脚，使其不再访问Parent中的name（Parent也没有name属性），而是

Child的name。在configChild()方法中，我们将该方法接受的闭包的delegate设置成了child，然后将该闭包的ResolveStrategy设置成了DELEGATE\_FIRST。这样，在调用configChild()时，所跟闭包中

代码被代理到了child上，即这些代码实际上是在child上执行的。此外，闭包的ResolveStrategy在默认情况下是OWNER\_FIRST，即它会先查找闭包的owner（这里基Parent），如果owner存在，则在owner上执行闭包中的

代码。这里我们将其设置成了DELEGATE\_FIRST，即该闭包会首先查找delegate（本例中即child），如果找到，该闭包便会在delegate上执行。对于上面的showDescription3，便是这种情况，当然，实际情况会稍微复杂一点，比如

showDescription3()方法会在内部调用showDescription3的configure()方法，再在configure()方法中设置执行闭包中的代码。

你可能会发现，在使用Gradle时，我们并没有想上面的parent.configChild()一样指明方法调用的对象，而是在build.gradle文件中直接调用task()，apply()和configuration()方法等，这是

因为在没有说明调用对象的情况下，Gradle会自动将调用对象设置成当前Project。比如调用apply()方法和调用project.apply()方法的效果是一样的。查查Gradle的Project文档，你会发现这些方法都是Project类的方法。

又举个例子，对于configurations()方法（它的作用我们将在后面的文章中讲到），该方法实际上会将所跟闭包的delegate设置成ConfigurationContainer，然后在该ConfigurationContainer上执行闭包中的代码。

再比如，dependencies()方法，该方法会将所跟闭包的delegate设置成DependencyHandler。还有，Project还定义了configure(Object object,Closure configureClosure)方法，该方法是专门用来配置对象的（比如Task），它会将configureClosure

的delegate设置成object，之后configureClosure中的执行代码其实是在object上执行的。和Groovy Bean一样，delegate机制的一个好处是可以增加所创建DSL的可读性。

# 增量式构建

如果我们将Gradle的Task看作一个黑盒子，那么我们便可以抽象出输入和输出的概念，一个Task对输入进行操作，然后产生输出。比如，在使用java插件编译源代码时，输入即为Java源文件，输出则为class文件。如果多次执行一个Task时的输入和输出是一样的，那么我们便可以认为这样的Task是没有必要重复执行的。此时，反复执行相同的Task是冗余的，并且是耗时的。

为了解决这样的问题，Gradle引入了增量式构建的概念。在增量式构建中，我们为每个Task定义输入（inputs）和输入（outputs），如果在执行一个Task时，它的输入和输出与前一次执行时没有发生变化，那么Gradle便会认为该Task是最新的（UP-TO-DATE），因此Gradle将不予执行。一个Task的inputs和outputs可以是一个或多个文件，可以是文件夹，还可以是Project的某个Property，甚至可以是闭包所定义的条件。

每个Task都拥有inputs和outputs属性，他们的类型分别为TaskInputs和TaskOutputs。

在下面的例子中，我们展示了这么一种场景：名为combineFileContent的Task从sourceDir目录中读取所有的文件，然后将每个文件的内容合并到destination.txt文件中。让我们先来看看没有定义Task输入和输出的情况：

task combineFileContentNonIncremental {

def sources = fileTree('sourceDir')

def destination = file('destination.txt')

doLast {

destination.withPrintWriter { writer ->

sources.each {source ->

writer.println source.text

}

}

}

}

多次执行“gradle combineFileContentNonIncremental”时，整个Task都会反复执行，即便在第一次执行后我们已经得到了所需的结果。如果该combineFileContentNonIncremental是一个繁重的Task，那么多次重复执行势必造成没必要的时间耗费。

这时，我们可以将sources声明为该Task的inputs，而将destination声明为outputs，重新创建一个Task如下：

task combineFileContentIncremental {

def sources = fileTree('sourceDir')

def destination = file('destination.txt')

inputs.dir sources

outputs.file destination

doLast {

destination.withPrintWriter { writer ->

sources.each {source ->

writer.println source.text

}

}

}

}

相比之下，后一个Task只比前一个Task多了两行代码：

inputs.dir sources

outputs.file destination

当首次执行combineFileContentIncremental时，Gradle会完整地执行该Task。但是紧接着再执行一次，命令行显示：

:combineFileContentIncremental UP-TO-DATE

BUILD SUCCESSFUL

Total time: 2.104 secs

我们发现，combineFileContentIncremental被标记为UP-TO-DATE，表示该Task是最新的，Gradle将不予执行。在实际应用中，你将遇到很多这样的情况，因为Gradle的很多插件都引入了增量式构建机制。

如果我们修改了inputs（即sourceDir文件夹）中的任何一个文件或删除掉了destination.txt，当调用“gradle combineFileContentIncremental”时，Gradle又会重新执行，因为此时的Task已经不再是最新的了。对于outputs，我们还可以使用upToDateWhen()方法来决定一个Task的outputs是否为最新的，该方法接受一个闭包作为检查条件，感兴趣的读者可以自行练习。

# 自定义Property

Gradle从入门到提高（五）—— 定义Project的Property

在前面的文章中我们讲到，设置和读取Project的Property是使用Gradle的一个很重要的方面。比如，很多Plugin都会向Project中加入额外的Property，在使用这些Plugin时，我们

需要对这些Property进行赋值。

Gradle在默认情况下已经为Project定义了很多Property，其中比较常用的有：

project Project本身

name Project的名字

path Project的绝对路劲

description Project的描述信息

buildDir Project构建结果存放目录

version Project的版本号

以下，我们首先设置Project的version和description属性，再定义showProjectProperties打印这些属性：

version = 'this is the project version'

description = 'this is the project description'

task showProjectProperties << {

println version

println project.description

}

请注意，在打印description时，我们使用了project.description，而不是直接使用description。原因在于，Project和Task都拥有description属性，而定义Task的闭包将delegate设置

成了当前的Task，故如果直接使用description，此时打印的是showProjectProperties的description，而不是Project的，所以我们需要显示地指明project。有关delegate的更多知识，请参考

本系列的上一篇文章。

Gradle还为我们提供了多种方法来自定义Project的Property。

（1）在build.gradle文件中定义Property

在build.gradle文件中向Project添加额外的Property时，我们并不能直接定义，而是应该通过ext来定义。比如如果要添加一个名为property1的Property，我们可以：

ext.property1 = "this is property1"

另外，我们也可以通过闭包的方式：

ext {

property2 = "this is property2"

}

在定义了Property后，在使用这些Property时我们则不需要ext，而是可以直接访问：

task showProperties << {

println property1

println property2

}

事实上，任何实现了ExtensionAware接口的Gradle对象都可以通过这种方式来添加额外的Property。

（2）通过命令行参数定义Property

Gradle还提供了-P命令行参数来设置Property，比如：

task showCommandLieProperties << {

println property3

}

在执行“gradle showCommandLieProperties”时，终端输出如下：

\* What went wrong:

Execution failed for task ':showCommandLieProperties'.

> Could not find property 'property3' on task ':showCommandLieProperties'.

表示property3并没有被定义，在调用gradle命令时，通过-P参数传入该Property：

gradle -Pproperty3="this is property3" showCommandLieProperties

此时终端显示：

:showCommandLieProperties

this is property3

BUILD SUCCESSFUL

（3）通过JVM系统参数定义Property

我们知道，在java中，我们可以通过-D参数定义JVM的系统参数，然后在代码中可以可以通过System.getProperty()进行获取。在Gradle中，我们也可以通过-D的方式向Project传入

Property，只是此时我们需要遵循一些约定：每一个通过-D方式声明的Property都需要以“org.gradle.project”为前缀，对于上面的showCommandLieProperties，我们也可以通过以下方式

为property3设置：

gradle -Dorg.gradle.project.property3="this is another property3" showCommandLieProperties

（4）通过环境变量设置Property

我们还可以通过设置环境变量的方式为Project定义Property。这种方式和（3）一样，需要我们遵循一些约定：在定义环境变量时，每一个Property都需要以“ORG\_GRADLE\_PROJECT\_”为前缀：

export ORG\_GRADLE\_PROJECT\_property3="this is yet another property3"

在调用showCommandLieProperties时，我们便不需要传入命令行参数了：

gradle showCommandLieProperties

在笔者所工作的项目中，我们的持续集成服务器便是通过这种方式向Gradle定义Property的。

# 使用java Plugin

Gradle最常用的Plugin便是java Plugin。和其他Plugin一样，java Plugin并没有什么特别的地方，只是向Project中引入了多个Task和Property。当然，java Plugin也有

比较与众不同的地方，其中之一便是在项目中引入了构建生命周期的概念，就像Maven一样。但是，和Maven不同的是，Gradle的项目构建生命周期并不是Gradle的内建机制，而是由Plugin

引入的。

请下载本系列文章的示例代码：git clone https://github.com/davenkin/gradle-learning.git

java Plugin引入的主要Task

执行“gradle build”，我们已经可以看到java Plugin所引入的主要Task：

:compileJava

:processResources

:classes

:jar

:assemble

:compileTestJava

:processTestResources

:testClasses

:test

:check

:build

BUILD SUCCESSFUL

Total time: 4.813 secs

build也是java Plugin所引入的一个Task，它依赖于其他Task，其他Task又依赖于另外的Task，所以有了以上Task执行列表。以上Task执行列表基本上描述了java Plugin向项目中

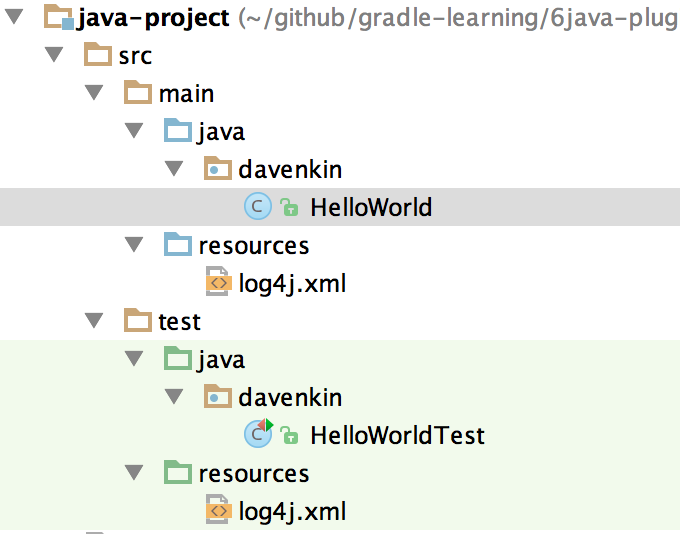
所引入的构建生命周期概念。

除了定义众多的Task外，java Plugin还想Project中加入了一些额外的Property。比如，sourceCompatibility用于指定在编译Java源文件时所使用的Java版本，archivesBaseName用于

指定打包成Jar文件时的文件名称。

Java项目的目录结构

在默认情况下，Gradle采用了与Maven相同的Java项目目录结构：



更多的Maven标准目录结构，请参考http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

当然，跟Maven一样，以上只是默认的目录结构，我们可以通过配置来修改这些目录结构。

配置已有source set

Gradle在采用了Maven目录结构的同时，还融入了自己的一些概念，即source set。对于上图中的目录结构，Gradle实际上为我们创建了2个source set，一个名为main，一个名为test。

请注意，这里的source set的名字main与上图目录结构中的main文件夹并无必然的联系，只是在默认情况下，Gradle为了source set概念到文件系统目录结构的映射方便，才采用了相同的名字。

对于test， 也是如此。我们完全可以在build.gradle文件中重新配置这些source set所对应的目录结构，同时，我们还可以创建新的source set。

从本质上讲，Gradle的每个source set都包含有一个名字，并且包含有一个名为java的Property和一个名为resources的Property，他们分别用于表示该source set所包含的Java源文件集合和资源文件集合。在实际应用时，我们

可以将他们设置成任何目录值。比如，我们可以重新设置main的目录结构：

sourceSets {

main {

java {

srcDir 'java-sources'

}

resources {

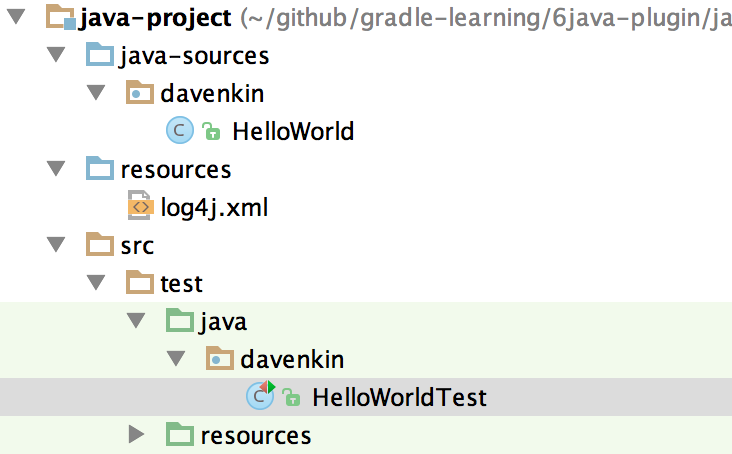
srcDir 'resources'

}

}

}

此时所对应的项目目录结构如下：



我们重新设置了main的目录结构，而对于test，我们保留了Gradle默认的目录结构。

创建新的source set

要创建一个新的source set也是非常简单的，比如，我们可以创建一个名为api的source set来存放程序中的接口类：

sourceSets {

api

}

当然，以上配置也可以与main放在一起。在默认情况下，该api所对应的Java源文件目录被Gradle设置为${path-to-project}/src/api/java，而资源文件目录则被设置成了${path-to-project}/src/api/resources。

我们也可以像上面的main一样重新对api的目录结构进行配置。

Gradle会自动地为每一个新创建的source set创建相应的Task，创建规律为：对于名为mySourceSet的source set，Gradle将为其创建compile<mySourceSet>Java、process<mySourceSet>Resources和<mySourceSet>Classes这3个Task。

对于这里api而言，Gradle会为其创建名为compileApiJava、processApiResource和apiClasses Task。我们可以在命令行中执行"gradle apiClasses"。

你可能会注意到，对于main而言，Gradle并没有相应的compileMainJava，原因在于：由于main是Gradle默认创建的source set，并且又是及其重要的source set，Gradle便省略掉了其中的“Main”，而是直接使用了compileJava作为main的编译Task。

对于test来说，我们看到，Gradle依然采用了compileTestJava。

通常的情况是，我们自己创建的名为api的source set会被其他source set所依赖，比如main中的类需要实现api中的某个接口等。此时我们需要做两件事情。第一，我们需要在编译main之前对api进行编译，即编译main中Java源文件的Task应该依赖于api中的Task：

classes.dependsOn apiClasses

第二，在编译main时，我们需要将api编译生成的class文件放在main的classpath下。此时，我们可以对main和test做以下配置：

sourceSets {

main {

compileClasspath = compileClasspath + files(api.output.

classesDir)

}

test {

runtimeClasspath = runtimeClasspath + files(api.output.

classesDir)

}

}

之所以需要对test的runtimeClasspath进行设置，是因为在运行测试时我们也需要加载api中的类。

# 依赖管理

Gradle从入门到提高（七）—— 依赖管理

一个Java项目总会依赖于第三方，要么是一个第三方类库，比如Apache commons；要么是你自己开发的另外一个Java项目，比如你的web项目依赖于另一个核心的业务项目。

通常来说，这种依赖的表示形式都是将第三方的Jar文件放在自己项目的classpath下，要么是编译时的classpath，要么是运行时的classpath。

在声明对第三方类库的依赖时，我们需要告诉Gradle在什么地方去获取这些依赖，即配置Gradle的Repository。在配置好依赖之后，Gradle会自动地从下载这些依赖到本地。

Gradle可以使用Maven和Ivy的Repository，同时它还可以使用本地文件系统作为Repository。

在本文中，我们将以Maven的Repository为例进行讲解，要配置Maven的Repository是非常简单的，我们只需要在build.gradle文件中加入以下代码即可：

repositories {

mavenCentral()

}

Gradle将对依赖进行分组，比如编译Java时使用的是这组依赖，运行Java时又可以使用另一组依赖。每一组依赖称为一个Configuration，在声明依赖时，我们实际上时在设置

不同的Configuration。值得一提的是，将依赖称为Configuration并不是一个好的名字，更好的应该叫作诸如“DependencyGroup”之类的。但是，习惯了就好的。

要定义一个Configuration，我们可以通过以下方式完成：

configurations {

myDependency

}

以上只是定义了一个名为myDependency的Configuration，我们并未向其中加入依赖。我们可以通过dependencies()方法向myDependency中加入实际的依赖项：

dependencies {

myDependency 'org.apache.commons:commons-lang3:3.0'

}

以上，我们将Apache的commons加入了myDependency中。之后，如果有Task需要将Apache commons加入到classpath中，我们可以通过以下方式进行获取：

task showMyDependency << {

println configurations.myDependency.asPath

}

执行“gradle showMyDependency”命令，在笔者的电脑上终端将显示：

:showMyDependency

/Users/twer/.gradle/caches/artifacts-26/filestore/org.apache.commons/commons-lang3/3.0/jar/8873bd0bb5cb9ee37f1b04578eb7e26fcdd44cb0/commons-lang3-3.0.jar

BUILD SUCCESSFUL

Total time: 4.405 secs

在实际应用时，比如我们需要调用Ant的某个target，而该target在执行时需要设置正确的classpath，那么我们便可以通过以上方式进行设置。

下面，我们来看一个Java项目，该项目依赖于SLF4J，而在测试时依赖于Junit。在声明依赖时，我们可以通过以下方式进行设置：

dependencies {

compile 'org.slf4j:slf4j-log4j12:1.7.2'

testCompile 'junit:junit:4.8.2'

}

。。。

我们并没有定义名为compile和testCompile的Configuration，这是这么回事呢？原因在于，java Plugin会自动定义compile和testCompile，分别用于编译Java源文件和编译Java测试源文件。

另外，java Plugin还定义了runtime和testRuntime这两个Configuration，分别用于在程序运行和测试运行时加入所配置的依赖。

再举个来自Gradle官网的例子：在Gradle中调用Ant，首先我们通过Configuration声明一组依赖，然后在Ant定义中将该Configuration所表示的classpath传给Ant：

configurations {

pmd

}

dependencies {

pmd group: 'pmd', name: 'pmd', version: '4.2.5'

}

task check << {

ant.taskdef(name: 'pmd', classname: 'net.sourceforge.pmd.ant.PMDTask', classpath: configurations.pmd.asPath)

ant.pmd(shortFilenames: 'true', failonruleviolation: 'true', rulesetfiles: file('pmd-rules.xml').toURI().toString()) {

formatter(type: 'text', toConsole: 'true')

fileset(dir: 'src')

}

}

如果存在依赖冲突，在默认情况下，Gradle会选择最新版本，这个Maven是不同的，Maven会选择离依赖树最近的版本。当然，我们可以通过设置Configuration的resolutionStrategy来

重新设置依赖冲突的处理规则，对此本文将不予讲解。

除了可以加入Maven和Ivy的Repository中的依赖之外，Gradle还允许我们声明对其他Project或者文件系统的依赖。比如，如果ProjectA的compileJava依赖于ProjectB，那么可以在ProjectA中声明如下：

dependencies {

compile project(':ProjectB')

}

在下一篇文章中，我们将对此再做讲解。

另外，对于本地文件系统中的Jar文件，我们可以通过以下方式声明对其的依赖：

dependencies {

compile files('spring-core.jar', 'spring-aap.jar')

compile fileTree(dir: 'deps', include: '\*.jar')

}

# 构建多个Project

Gradle为每个build.gradle都会创建一个相应的Project领域对象，在编写Gradle脚本时，我们实际上是在操作诸如Project这样的Gradle领域对象。

在多Project的项目中，我们会操作多个Project领域对象。Gradle提供了强大的多Project构建支持。

要创建多Project的Gradle项目，我们首先需要在根（Root）Project中加入名为settings.gradle的配置文件，该文件应该包含各个子Project的名称。

比如，我们有一个根Project名为root-project，它包含有两个子Project，名字分别为sub-project1和sub-project2，此时对应的文件目录结构如下：

root-project/

sub-project1/

build.gradle

sub-project2/

build.gradle

build.gradle

settings.gradle

root-project本身也有自己的build.gradle文件，同时它还拥有settings.gradle文件位于和build.gradle相同的目录下。此外，两个子Project也拥有他们

自己的build.gradle文件。

要将sub-project1和sub-project2加入到root-project的子Project中，我们需要在settings.gradle中加入：

include 'sub-project1', 'sub-project2'

接下来，我们来定义一个Task用于显示每个Project各自的名称。我们可以在每个build.gradle进行定义，但是这却是一种比较笨的方法，此时我们也完全没有享受到Gradle

的多Project构建功能所带来的好处。在Gradle中，我们可以通过根Project的allprojects()方法将配置一次性地应用于所有的Project，当然也包括定义Task。比如，在root-project

的build.gradle中，我们可以做以下定义：

allprojects {

apply plugin: 'idea'

task allTask << {

println project.name

}

}

以上Gradle脚本，将闭包中的代码应用在所有的Project中，包括root-project本身。我们首先将应用了idea Plugin用于生成IntelliJ工程，其次我们定义了名为allTask的Task，该

Task应用于每个Project，作用是输出各个Project的名称。执行“gradle allTask”，命令行输出如下：

:allTask

root-project

:sub-project1:allTask

sub-project1

:sub-project2:allTask

sub-project2

我们看到，该allTask对于每个Project都执行了一次，在执行时输出了当前Project的名称。

除了allprojects()之外，Project还提供了subprojects()方法用于配置所有的子Project（不包含根Project）。比如，我们可以定义Task来只输出各个子Project的名字：

subprojects {

task subTask << {

println project.name

}

}

执行“gradle subTask”，命令行输出如下：

:sub-project1:subTask

sub-project1

:sub-project2:subTask

sub-project2

此时的输出中不包含root-project的名字。

上文中已经提到，在Gradle脚本中，我们实际上是在操作一些领域对象，因此我们可以将groovy的所有语言特性用在Gradle的领域对象上，比如我们可以对Project进行过滤：

configure(allprojects.findAll { it.name.startsWith('sub') }) {

subTask << {

println 'this is a sub project'

}

}

在上面的代码中，我们先找到所有Project中名字以“sub”开头的Project，然后再对这些Project进行配置，在配置中，我们向这些Project的subTask中加入了一条额外的打印语句。

此时如果再执行“gradle subTask”，命令行输出如下：

:sub-project1:subTask

sub-project1

this is a sub project

:sub-project2:subTask

sub-project2

this is a sub project

到此为止，我们所有的Task定义工作都是在root-project中进行的，而sub-project1和sub-project2中的build.gradle文件依然什么都没有。事实上，我们可以将所有子Project的配置

均放在根Project中进行。在上面的例子中，我们通过allprojects()和subprojects()将所有的子Project都包含在了配置之内，其实我们还可以对某个Project进行单独配置。比如，在root-project

的build.gradle中加入：

project(':sub-project1') {

task forProject1 << {

println 'for project 1'

}

}

以上脚本向sub-project1中加入了名为forProject1的Task，在执行“gradle forProject1”时，终端输出如下：

:sub-project1:forProject1

for project 1

这里有一个问题：我们是在root-project下执行的命令，因此照理说Gradle会认为forProject1是定义在所有的Project上，而此时只有sub-project1才拥有该Task，Gradle应该抛出异常指示在root-project和sub-project2上找不到

该Task才对，为什么它还是执行成功了呢？原因在于：只有在一个Task没有在任何Project中定义时，Gradle才会将其当做异常。否则，Gradle会在所有拥有该Task的Project上执行该Task。

一旦有了多个Project，他们之间便会存在着依赖关系。Gradle的Project之间的依赖关系是基于Task的，而不是整个Project的。

现在，让我们来看一个Project依赖的例子。比如sub-project1中有taskA和taskB，taskA依赖于taskB：

task taskA << {

println 'this is taskA from project 1'

}

task taskB << {

println 'this is taskB from project 1'

}

taskA.dependsOn taskB

在执行“gradle taskA”时，终端输出：

:sub-project1:taskB

this is taskB from project 2

:sub-project1:taskA

this is taskA from project 1

这个很容易理解，两个Task都是属于sub-project1的。但是，让我们再向其中加入一些复杂性。我们在sub-project2中定义taskC和taskD，然后使taskA再依赖于taskC，又使taskB依赖于taskD：

//sub-project1:

taskA.dependsOn ':sub-project2:taskC'

taskB.dependsOn ':sub-project2:taskD'

//sub-project2:

task taskC << {

println 'this is taskC from project 2'

}

task taskD << {

println 'this is taskD from project 2'

}

此时再执行“gradle taskA”，终端输出如下：

:sub-project2:taskD

this is taskD from project 2

:sub-project1:taskB

this is taskB from project 1

:sub-project2:taskC

this is taskC from project 2

:sub-project1:taskA

this is taskA from project 1

分析一下：taskA依赖于taskB，而taskB又依赖于taskD，所以sub-project1的taskD首先得到了执行，然后再执行sub-project1的taskB。之后，又由于taskA依赖于taskC，故Gradle

再次转向sub-project1执行taskC，最后才执行taskA。

# 自定义Task类型

在前面的文章中我们讲到，Gradle本身只是一个架子，真正起作用的是Task和Plugin。要真正了解Task和Plugin的工作机制并熟练运用，学会自定义Task类型和Plugin是大有裨益的。

Gradle中的Task要么是由不同的Plugin引入的，要么是我们自己在build.gradle文件中直接创建的。在默认情况下，我们所创建的Task是DefaultTask类型，该类型是一个非常通用的Task类型，而在有些时候，我们希望创建一些具有特定功能的Task，比如Copy和Jar等。还有时候，我们希望定义自己创建的Task类型，在本文中，我们以定义一个简单的HelloWorldTask为例，讲解如何自定义一个Task类型，并且如何对其进行配置。

在Gradle中，我们有3种方法可以自定义Task类型。

（1）在build.gradle文件中直接定义

我们知道，Gradle其实就是groovy代码，所以在build.gradle文件中，我们便可以定义Task类。

class HelloWorldTask extends DefaultTask {

@Optional

String message = 'I am davenkin'

@TaskAction

def hello(){

println "hello world $message"

}

}

task hello(type:HelloWorldTask)

task hello1(type:HelloWorldTask){

message ="I am a programmer"

}

在上例中，我们定义了一个名为HelloWorldTask的Task，它需要继承自DefaultTask，它的作用是向命令行输出一个字符串。@TaskAction表示该Task要执行的动作，即在调用该Task时，hello()方法将被执行。另外，message被标记为@Optional，表示在配置该Task时，message是可选的。在定义好HelloWorldTask后，我们创建了两个Task实例，第一个hello使用了默认的message值，而第二个hello1在创建时重新设置了message的值。

在执行hello时，命令行输出如下：

:hellohello world I am davenkin

BUILD SUCCESSFUL

在执行hello1时，命令行输出如下：

:hello1hello world I am a programmer

BUILD SUCCESSFUL

（2）在当前工程中定义Task类型

在（1）中，我们在build.gradle中直接定义了Task的类型，这样将Task的定义和使用混在一起。在需要定义的Task类型不多时，我们可以采用这种方法，但是在项目中存在大量的自定义Task类型时，这就不见得是中好的做法了。一种改进方法是在另外的一个gradle文件中定义这些Task，然后再apply到build.gradle文件中。这里，我们将使用另一种方法：在buildSrc目录下定义Task类型，Gradle在执行时，会自动地查找该目录下所定义的Task类型，并首先编译该目录下的groovy代码以供build.gradle文件使用。

在当前工程的buildSrc/src/main/groovy/davenkin目录下创建HelloWorldTask.groovy文件，将（1）中对HelloWorldTask的定义转移到该文件中：

package davenkin

import org.gradle.api.\*

import org.gradle.api.tasks.\*

class HelloWorldTask extends DefaultTask {

@Optional

String message = 'I am davenkin'

@TaskAction

def hello(){

println "hello world $message"

}

}

这里，我们将HelloWorldTask定义在了davenkin包下，因此在build.gradle文件中引用该Task时，我们需要它的全名称：

task hello(type:davenkin.HelloWorldTask)

task hello1(type:davenkin.HelloWorldTask){

message ="I am a programmer"

}

以上的hello和hello1与（1）中的hello和hello1完成的功能相同。

（3）在单独的项目中定义Task类型

虽然（2）中的Task定义与build.gradle分离开了，但是它依然只能应用在当前工程中。如果我们希望所定义的Task能够用在另外的项目中，那么（2）中的方法便不可行的，此时我们可以将Task的定义放在单独的工程中，然后在所有使用Task的工程中通过声明依赖的方式引入这些Task。

创建另外一个项目，将（2）中buildSrc目录下的内容考到新建项目中，由于该项目定义Task的文件是用groovy写的，因此我们需要在该项目的build.gradle文件中引入groovy Plugin。另外，由于该项目的输出需要被其他项目所使用，因此我们还需要将其上传到repository中，在本例中，我们将该项目生成的包含了Task定义的jar文件上传到了本地的文件系统中。最终的build.gradle文件如下：

apply plugin: 'groovy'

apply plugin: 'maven'

version = '1.0'

group = 'davenkin'

archivesBaseName = 'hellotask'

repositories.mavenCentral()

dependencies {

compile gradleApi()

groovy localGroovy()

}

uploadArchives {

repositories.mavenDeployer {

repository(url: 'file:../lib')

}

}

执行“gradle uploadArchives”，所生成的jar文件将被上传到上级目录的lib(../lib)文件夹中。

在使用该HelloWorldTask时，客户端的build.gradle文件可以做以下配置：

buildscript {

repositories {

maven {

url 'file:../lib'

}

}

dependencies {

classpath group: 'davenkin', name: 'hellotask', version: '1.0'

}

}

task hello(type: davenkin.HelloWorldTask)

首先，我们需要告诉Gradle到何处去取得依赖，即配置repository。另外，我们需要声明对HelloWorldTask的依赖，该依赖用于当前build文件。之后，对hello的创建与（2）中一样。

# 自定义Plugin

在Plugin中，我们可以向Project中加入新的Task，定义configurations和property等。我们3种方法可以自定义Plugin，这些方法和自定义Task类型的3种方法相似。在接下来的例子中，我们将分别通过这3种方法来创建一个DateAndTimePlugin，该Plugin定义了2个Task，分别用于输出系统当前的日期和时间，另外，我们可以配置日期和时间的输出格式。

（1）在build.gradle文件中直接定义Plugin

和在build.gradle文件中定义Task类型一样，我们可以将对Plugin的定义直接写在build.gradle中：

apply plugin: DateAndTimePlugin

dateAndTime {

timeFormat = 'HH:mm:ss.SSS'

dateFormat = 'MM/dd/yyyy'

}

class DateAndTimePlugin implements Plugin<Project> {

void apply(Project project) {

project.extensions.create("dateAndTime", DateAndTimePluginExtension)

project.task('showTime') << {

println "Current time is " + new Date().format(project.dateAndTime.timeFormat)

}

project.tasks.create('showDate') << {

println "Current date is " + new Date().format(project.dateAndTime.dateFormat)

}

}

}

class DateAndTimePluginExtension {

String timeFormat = "MM/dd/yyyyHH:mm:ss.SSS"

String dateFormat = "yyyy-MM-dd"

}

每一个自定义的Plugin都需要实现Plugin<T>接口，事实上，除了给Project编写Plugin之外，我们还可以为其他Gradle类编写Plugin。该接口定义了一个apply()方法，在该方法中，我们可以操作Project，比如向其中加入Task，定义额外的Property等。

在上例中，我们在DateAndTimePlugin中向Project添加了2个Task，一个名为showTime，一个名为showDate。请注意创建这2个Task所使用的不同方法，更多的创建Task的方法，请参考本系列这篇文章。

每个Gradle的Project都维护了一个ExtenionContainer，我们可以通过project.extentions进行访问，比如读取额外的Property和定义额外的Property等。在DateAndTimePlugin中，我们向Project中定义了一个名为dateAndTime的extension，并向其中加入了2个Property，分别为timeFormat和dateFormat，他们又分别用于showTime和showDate。在使用该Plugin时，我们可以通过以下方式对这两个Property进行重新配置：

dateAndTime {

timeFormat = 'HH:mm:ss.SSS'

dateFormat = 'MM/dd/yyyy'

}

（2）在当前工程中定义Plugin

在当前工程中的buildSrc/src/main/groovy/davenkin目录下创建DateAndTimePlugin.groovy文件，将build.gradle中定义DateAndTimePlugin的代码提取到给文件中，但是除去对DateAndTimePluginExtension的定义，因为我们将在另外一个单独的文件中定义DateAndTimePluginExtension。

package davenkin

import org.gradle.api.Plugin

import org.gradle.api.Project

class DateAndTimePlugin implements Plugin<Project> {

void apply(Project project) {

project.extensions.create("dateAndTime", DateAndTimePluginExtension)

project.task('showTime') << {

println "Current time is " + new Date().format(project.dateAndTime.timeFormat)

}

project.tasks.create('showDate') << {

println "Current date is " + new Date().format(project.dateAndTime.dateFormat)

}

}

}

再创建DateAndTimePluginExtension.groovy：

package davenkin

class DateAndTimePluginExtension {

String timeFormat = "MM/dd/yyyyHH:mm:ss.SSS"

String dateFormat = "yyyy-MM-dd"

}

这里，我们将2个类文件都放在了davenkin包下。Gradle在执行时，会自动扫描buildSrc目录，并会在执行Task之前构建该目录下的内容。在build.gradle文件中，在apply该Plugin时，我们需要声明对该Plugin的全名称，即包含报名：

apply plugin: davenkin.DateAndTimePlugin

dateAndTime {

timeFormat = 'HH:mm:ss.SSS'

dateFormat = 'MM/dd/yyyy'

}

执行“gradle showTime”，命令行输出如下：

:buildSrc:compileJava UP-TO-DATE

:buildSrc:compileGroovy UP-TO-DATE

:buildSrc:processResources UP-TO-DATE

:buildSrc:classes UP-TO-DATE

:buildSrc:jar UP-TO-DATE

:buildSrc:assemble UP-TO-DATE

:buildSrc:compileTestJava UP-TO-DATE

:buildSrc:compileTestGroovy UP-TO-DATE

:buildSrc:processTestResources UP-TO-DATE

:buildSrc:testClasses UP-TO-DATE

:buildSrc:test UP-TO-DATE

:buildSrc:check UP-TO-DATE

:buildSrc:build UP-TO-DATE

:showTime

Current time is 19:08:35.489

BUILD SUCCESSFUL

Total time: 2.995 secs

可以看到，Gradle会首先构建buildSrc目录，然后才执行showTime。

（3）在单独的项目中创建Plugin

新建一个项目，将（2）中buildSrc目录下的内容拷贝到该项目下，定义该项目的build.gradle文件如下：

apply plugin: 'groovy'

apply plugin: 'maven'

version = 1.0

group = 'davenkin'

archivesBaseName = 'datetimeplugin'

repositories.mavenCentral()

dependencies {

compile gradleApi()

groovy localGroovy()

}

uploadArchives {

repositories.mavenDeployer {

repository(url: 'file:../lib')

}

}

此外，我们还可以为该Plugin重新命名，如果我们希望将该Plugin命名为time，那么我们需要在src/main/resources/META-INF/gradle-plugins目录下创建名为time.properties的文件，内容如下：

implementation-class = davenkin.DateAndTimePlugin

在执行“gradle uploadArchives”时，Gradle会将该Plugin打包成jar文件，然后将其上传到上级目录下的lib目录中（../lib）。之后，在客户端的build.gradle文件中，我们需要做如下定义：

buildscript {

repositories {

maven {

url 'file:../lib'

} }

dependencies {

classpath group: 'davenkin', name: 'datetimeplugin',

version: '1.0'

}

}

apply plugin: 'time'

dateAndTime {

timeFormat = 'HH:mm:ss.SSS'

dateFormat = 'MM/dd/yyyy'

}

首先我们配置repository以执行lib目录，然后声明对DateAndTimePlugin的依赖，再apply该Plugin，此时我们应该使用“time”作为该Plugin的名称，最后对该Plugin进行配置。