版本号	描述	日期(更新日期)/更新人
1.0	Android Studio Gradle技术	2019/05/23 **刘志保**
Github地址	https://github.com/MMLoveMeMM/ AngryPandaGradle	
	https://github.com/davenkin/gradle-learning	

<一>：先明确几个概念:

<1>：Android Studio 编译工作等是由Gradle组织的成的project下一系列task[任务项]去完成的.比如这些任务项包括编译之前先进行lint任务,unit任务等等.总而言之,gradle组织了一系列任务完成编译工作;

<2>：平时升级各种gradle-**.jar版本中,其实是有google工程师完成的一些编译Android等工程的插件,这个插件包含了各种所需要的任务,并且将各自的任务做了系统的安排.

<3>：既然google工程师可以开发一套复杂的插件进行优化编译过程,那么我们同样可以开发一套简单的工作去简化平时的工作任务,比如打包等.即开发自己的自定义插件plugin,而正如上面所说,插件显然可以包含多个Task任务,并且组织和协调各个Task联合起来完成一项复杂的工作.

<4>：既然Plugin是由很多Task完成的,那么就可以开发自定任务Task,所以Plugin和Task的关系类似于进程和线程的关系.

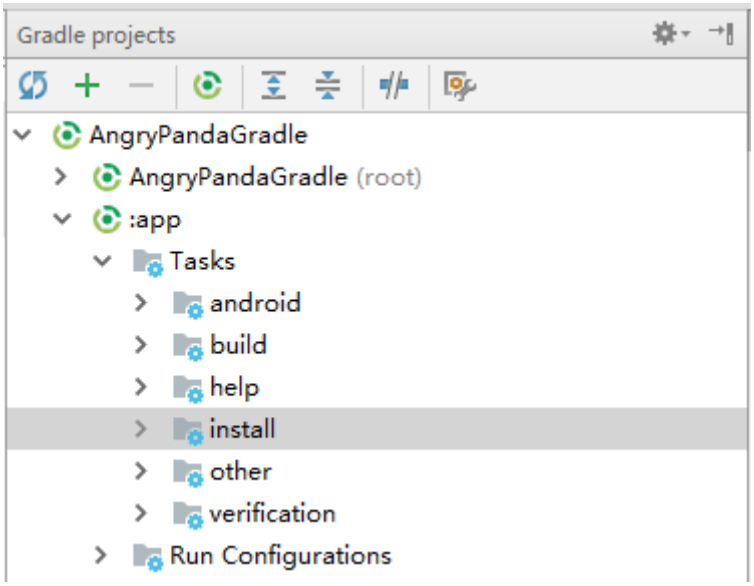
<5>：如何开发,开发gradle采用的是groovy,其语法相对简单,并且能够接纳java语言,混合编程,比如:

获取当前时间:

```
def curTime = new Date().format("yyyyMMddHHmm")
```

<6>：学习groovy基本知识或者开发自定义任务Task,或者补充google提供的build中的任务Task都可以直接在build.gradle文件下进行书写开发.

<7>：认识一个常规APP的任务Task列表清单:



其中每个文件夹下列出的各项都是一个Task,可想而知每次编译生成APK的时候,做了多少任务才能够完全生成.

<二>:其次明确一下groovy细节约定:

<1>:书写约定

```
String message = "develop gradle by groovy"
// def message = "develop gradle by groovy"
println("hello world ${message}")
// 建议不使用:
println("hello world $message")
println "hello world $message"
// 虽然都可以!
```

<2>:自定义gradle文件,比如自己新建了一个copy.gradle文件,将开发放在copy.gradle中,那么在build.gradle引入如下:

在

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

下面添加:

```
apply from: this.file('task.gradle')
```

注意文件从属关系,因为平时开发不可能全部堆叠在build.gradle中.

<三>:最后开始看看groovy具体如何开发:

<1>:Closure(闭包)Closure是一段单独的代码块,它可以接收参数,返回值,也可以被赋值给变量:

```
//无参数
def Closure1 = {
println 'Hello world'
```

```
}
Closure1() // 执行闭包，输出Hello world

//接收一个参数
def Closure2 = {
String str -> println str //箭头前面是参数定义，后面是执行代码，str为外部传入的参数
} //如果只有一个参数可以用it代替，也可写作：
def Closure2 = {
println it
}
Closure2('Hello world')// 执行闭包，输出Hello world

//接收多个参数
def Closure3 = {
String str , int n -> println "$str : $n" //参数前加$
} //也可以写作：
def Closure3 = {
str , n -> println "$str : $n"
}
Closure3('Hello world’, 1) // 执行闭包，输出Hello world : 1

//使用变量
def var = "Hello world"
def Closure4 = {
println var
}
Closure4() // 执行闭包，输出Hello world

//改变上下文
def Closure5 = {
println Var //这时还不存在
}
MyClass m = new MyClass()
Closure5.setDelegate(m) // 改变上下文，这时Var已经有了，在执行之前改变了
Closure5() //执行闭包，输出Hello world
class MyClass {
def Var = 'Hello world'
}

```

阅读,其实类似于程序的方法或者函数.

<2>：Property[属性] Gradle在默认情况下已经为Project定义了很多Property，如下：

```
project：Project本身
name：Project的名字
path：Project的绝对路径
description：Project的描述信息
buildDir：Project构建结果存放目录
version：Project的版本号
```

其中,通过ext来自定义Property：

```
ext.property1 = "this is property1"
或
ext {
property2 = "this is property2"
}

task showProperties << {
println property1 //直接访问
println property2
 }
```

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，比如Task也实现了该接口。

通过“-p”命令行参数定义Property：

```
task showCommandLineProperties << {
println propertyTest
}

gradle -P propertyTest ="this is propertyTest" showCommandLineProperties
```

通过JVM系统参数定义Property（需要以“org.gradle.project”为前缀）：

```
gradle -D org.gradle.project.propertyTest="this is another propertyTest"
showCommandLineProperties

另一种方式：

写入参数：gradle -DpropertyTest="this is another propertyTest"
读取参数：def propertyTest = System.properties['propertyTest']
```

通过环境变量设置Property（需要以“ORG_ GRADLE_ PROJECT_ ”为前缀）：

```
export ORG_GRADLE_PROJECT_propertyTest = "this is yet another propertyTest"

gradle showCommandLineProperties
```

<3>：数据实体Bean(类似于javabean)

```
> Groovy会为每一个字段自动生成getter和setter，我们可以通过像访问字段本身一样调用getter和setter，如：

class GroovyBeanExample {
private String name
}

def bean = new GroovyBeanExample()
bean.name = 'this is name' //Groovy动态地为name创建了getter和setter
println bean.name
```

<4>：代理delegate机制: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

```
class Child {
private String name
}

class Parent {
Child child = new Child();

void configChild(Closure c) {
c.delegate = child
 c.setResolveStrategy Closure.DELEGATE_FIRST //默认情况下是OWNER_FIRST，即它会先查找闭包的owner（这里即parent）
 c()
 }
 }

 def parent = new Parent()
 parent.configChild {
 name = "child name"
 }

 println parent.child.name
```

<5> Task任务[重点]

task有两个生命周期，***配置阶段和执行阶段***。

gradle在执行task时，都会先对task进行配置，task中最顶层的代码就是配置代码，在配置阶段执行，其他代码实在执行阶段执行的；

task关键字实际上是一个方法调用，我们不用将参数放在括号里面。

这个一定要熟悉Task.class里面的程序,熟悉Task里面的接口.

```
task Task1 {
println "hello” // 这段代码是在配置阶段执行的
}

task Task2 {
def name = "hello” // 这段代码是在配置阶段执行的
doLast {
println name
} // 这段代码是在执行阶段执行的，相当于：

 // doLast({
 // println 'Hello world!'
 // })
 }

 task Task3 << {
 println name
 }
 // “<<”语法糖，表示追加执行过程，相当于doLast，因此整个代码都是在执行阶段执行的；与之相反的是doFirst。
 //如果代码没有加“<<”，则这个任务在脚本initialization的时候执行（也就是你无论执行什么任务，这个任务都会被执行，“hello”都会被输出）；
 //如果加了“<<”，则在输入命令gradle Task3后才执行

```

通过TaskContainer的create()方法创建Task

```
tasks.create(name: 'hello') << {
println 'hello'
}
```

<6> 自定义Task[重点]

```
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"
 }
```

其中,@TaskAction表示该Task要执行的动作，@Optional表示在配置该Task时，message是可选的

<7>：任务Task配置:

方法一：在定义Task的时候对Property进行配置

```
task hello1 << {
description = "this is hello1"
println description
}
```

方法二：通过闭包的方式来配置一个已有的Task

```
task hello2 << {
println description
}

hello2 {
description = "this is hello2"
} //Gradle会为每一个task创建一个同名的方法，该方法接受一个闭包

或

 hello2.description = "this is hello2"//Gradle会为每一个task创建一个同名的Property，所以可以将该Task当作Property来访问
```

注：对hello2的description的设置发生在定义该Task之后，在执行gradle hello2时，命令行依然可以打印出正确的“this is hello2”，这是因为Gradle在执行Task时分为两个阶段：配置阶段、执行阶段。
所以在执行hello2之前，Gradle会扫描整个build.gradle文档，将hello2的description设置为“this is hello2”，再执行hello2。

方法三：通过Task的configure()方法完成Property的设置

```
task hello3 << {
println description
}

hello3.configure {
description = "this is hello3"
}
```

<8>：任务依赖关系,Task dependence on

```
task A << {
println 'Hello from A'
}
task B << {
println 'Hello from B'
}
B.dependsOn A

或

 task A << {
 println 'Hello from A'
 }
 task B {
 dependsOn A
 doLast {
 println 'Hello from B'
 }
 }

 或

 task B(dependsOn: A) {
```

```
 println 'Hello from B'
 }
```

<9>：顺序执行任务:

```
task unit << {
println 'Hello from unit tests'
}
task ui << {
println 'Hello from UI tests'
}
task tests << {
println 'Hello from all tests!'
}
 task mergeReports << {
 println 'Merging test reports'
 }

 tests.dependsOn unit //单元测试
 tests.dependsOn ui //ui测试
 ui.mustRunAfter unit //ui测试必须在单元测试之后执行
 tests.finalizedBy mergeReports //表示tests执行完后，再执行mergeReports，等价于mergeReports.dependsOn tests

```

<10>：增量式构建:为一个Task定义输入（inputs）和输出（outputs），在执行该Task时，如果它的输入和输出与前一次执行时没有变化，那么Gradle便会认为该Task是最新的（日志会输出“UP-TO-DATE“），因此不会重复执行.

```
task combineFileContent {
def sources = fileTree('sourceDir')
def destination = file('destination.txt')

inputs.dir sources // 将sources声明为该Task的inputs
outputs.file destination // 将destination声明为outputs

doLast {
destination.withPrintWriter { writer ->
 sources.each {source ->
 writer.println source.text
 }
 }
```

```
14  }
15  }
```

当首次执行combineFileContent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：

```
1   :combineFileContent UP-TO-DATE //被标记为UP-TO-DATE，表示该Task是最新的，不执行
2   BUILD SUCCESSFUL
3   Total time: 2.104 secs
```

如果修改inputs（上述即sourceDir文件夹）中的任何一个文件或删除destination.txt，再次调用“gradle combineFileContent”时，该Task又会重新执行.

<11>：自定义插件[重点,gradle就是google开发给android studio开发人员使用的工具插件],在当前工程中的buildSrc/src/main/groovy/davenkin目录下创建DateAndTimePlugin.groovy文件和DateAndTimePluginExtension.groovy文件,这个具体可以参考[Android Studio gradle技术.docx]文档.

```
1   // DateAndTimePlugin.groovy
2
3   package com.gradle.test
4
5   import org.gradle.api.Plugin
6   import org.gradle.api.Project
7
8   class DateAndTimePlugin implements Plugin<Project> {
9   void apply(Project project) {
10
11   //每个Gradle的Project都维护了一个ExtenionContainer，我们可以通过project.extentions访问额外的Property和定义额外的Property
12   project.extensions.create("dateAndTime", DateAndTimePluginExtension)
13
14   project.task('showTime') << {
15   println "Current time is " + new Date().format(project.dateAndTime.timeFormat)
16   }
17
18   project.tasks.create('showDate') << {
19   println "Current date is " + new Date().format(project.dateAndTime.dateFormat)
20   }
21   }
```

```
}

// DateAndTimePlugin.groovy

package com.gradle.test

class DateAndTimePluginExtension {
String timeFormat = "MM/dd/yyyyHH:mm:ss.SSS"
String dateFormat = "yyyy-MM-dd"
}

```

build.gradle文件中，再apply该Plugin

```
apply plugin: com.gradle.test.DateAndTimePlugin

// 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'
}

```

<四>：gradle常用打包命令:

注：执行“./gradlew xxx”等同于执行“gradle xxx”，但执行“gradle xxx”需配置环境变量

不过在windows系统,直接在终端执行 gradlew xxx即可.

<1>：清除build文件夹

```
./gradlew clean
```

<2>：检查依赖并编译打包

```
./gradlew build
```

<3>：编译并打Debug包

```
./gradlew assembleDebug
```

<4>：编译并打Release包

```
./gradlew assembleRelease
```

<5>：获取gradle版本号

```
./gradlew -v
```

<6>：查看所有任务

```
./gradlew tasks 或 gradle tasks
```

<7>：查看所有工程

```
gradle projects
```

<8>：查看所有属性

```
gradle properties
```

<五> 常用的任务

<1>：执行任务

```
task A << {
 println 'Hello from A'
}
终端输入：gradle A
```

<2>：拷贝,移动,重命名文件

```
task copyFile(type: Copy) {
 from 'source'
 into 'destination'
 rename{
 newFilename
 }
}
将source文件夹中的所有内容拷贝到destination文件夹中，
这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

```

<3>：删除文件或文件夹

```
task deleteTest(type: Delete) {
 delete 'file' ,'dir'
}
文件和文件夹是相对于当前Project而言的，即build.gradle文件所在的目录
```

<4>：执行shell命令

```
task runShell1(type: Exec) {
 executable "sh"
 args "-c", "rm ./app/libs/test.jar" //路径是相对于当前build.gradle文件
}

或者
```

```
7
8  def cmd = 'date +%Y-%m-%d'
9  task shellTest << {
10   String date = cmd.execute().text.trim().toString() //带返回值
11   print date //打印系统日期
12  }
```

<5>：执行java代码

```
1  task runJava(type: JavaExec) {
2   classpath = sourceSets.main.runtimeClasspath //执行文件所在路径
3   main = 'com.example.MyClass' // 执行方法所在类
4   // arguments to pass to the application
5   args 'haha','xixi' //多个参数用逗号隔开
6  }
```

<六> 常见配置

<1>：设置全局参数（同额外属性设置）

```
1  ext {
2   compileSdkVersion = 22
3   buildToolsVersion = "22.0.1"
4  }
5
6  在module中引用全局参数：
7  android {
8   compileSdkVersion rootProject.ext.compileSdkVersion
9   buildToolsVersion rootProject.ext.buildToolsVersion
10  }
```

将属性或方法放入ext{}就可以被全局引用

<2>：设置全局编码

```
1  allprojects {
2   tasks.withType(JavaCompile) {
3   options.encoding = "UTF-8"
4   }
5  }
```

<3>：设置全局编译器的版本

```
allprojects {
 tasks.withType(JavaCompile) {
 sourceCompatibility = JavaVersion.VERSION_1_7
 targetCompatibility = JavaVersion.VERSION_1_7
 }
}
```

<4>：去掉重复依赖

```
compile 'com.alibaba.fastjson.latest.integration' { //latest.integration 获取服务器上最新版本
 exclude module: 'annotations', group: 'com.google.android'
}
```

<5>：本地aar包依赖

```
allprojects {
 repositories {
 jcenter()
 flatDir {
 dirs 'libs'
 }
 }
}

dependencies {
 compile(name:'本地库aar的名字，不带后缀', ext:'aar')
}
```

<七> 配置gradle，加快编译速度

<1>：~/.gradle/gradle.properties中添加如下配置（没有该文件则新建一个）：

```
org.gradle.daemon=true //独立进程，停止后台进程命令：gradle --stop
org.gradle.parallel=true //并行构建，需要将项目拆分成多个子项目，通过aar引用才能起效
org.gradle.configureondemand=true //按需配置，目前还在试验孵化阶段，默认是关闭的
```

<2>：设置离线编译：

```
打开settings->Build,Execution,Deployment->Build Tools->Gradle,
选中Offlie Work //更新依赖包时要取消它
```

<3>：命令行构建时在命令后面加上如下参数

```
--daemon
--parallel
--offline
--dry-run
```

参考来源:

```
https://github.com/davenkin/gradle-learning
https://blog.csdn.net/u010818425/article/details/52268126
```