# Gradle快速入门

# Groovy 概述

## 领域特定语言DSL介绍

DSL全称domain specific language（领域特定语言）。dsl并不是一个全新概念，像专门做计算的matlab，统一建模语言uml，超文本标记语言html，都是属于DSL的一种。Groovy也是DSL的一个分支，专门用来做脚本。

## DSL语言及特点

DSL解决的是特定领域的专有问题，和系统开发语言是相辅相成的。

## DSL与通用变成语言的区别

DSL语言的特点就是解决某一特定领域内的问题，DSL是纵向往深度方向发展，通用系统编程语言是横向以更广的应用范围发展，这是最大的区别。

DSL核心思想：求专不求全，解决特定问题。

## Groovy 介绍

1. Groovy是一种基于JVM的敏捷开发语言

可以将groovy编写的源文件编译成class字节码文件，然后交给JVM去执行，也可以直接将groovy源文件解释执行。

1. 结合了Python，Ruby和Smalltalk的许多强大的特性

Groovy是对Java的扩展，语法和Java很类似，可以以学习Java的方式去学习groovy，可以起到事半功倍的效果。

1. Groovy可以与Java完美结合，而且可以使用java所有的库

## Groovy特性

1. 语法上支持动态类型，闭包等新一代语言特性
2. 无缝集成所有已经存在的Java类库
3. 既支持面向对象编程也支持面向过程编程

基于面向对象编程是因为groovy是基于java的扩展，而Java是可以面向对象编程的，所以groovy也是支持面向对象编程的。

面向过程是因为groovy结合了Python，Ruby这些脚本语言的特性，所以也是支持面向过程编程的，主要看最后将groovy用在什么用途，如果使用groovy开发应用程序，就需要用groovy的面向对象编程这些特性，如果用groovy写脚本，使用的是groovy的面向过程特性。

## Groovy优势

1. 一种更加敏捷的编程语言

它在语法上做了非常多的语法糖，有许多在java中必须写的东西，在groovy中是可以省略的，我们可以用更少的代码写出同样的功能。

1. 入门非常容易，但功能非常强大

在掌握java的前提下，再去学习groovy，那是非常简单的，甚至可以搬用java那一套写groovy程序

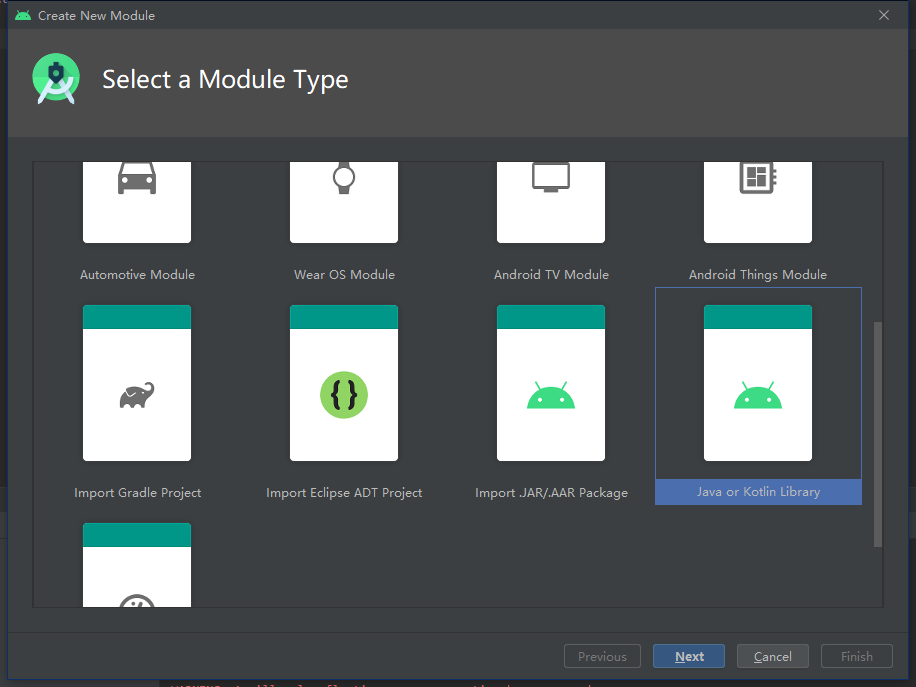
1. 既可以作为编程语言也可以作为脚本语言

groovy即结合了java特性，也结合了Python，Ruby等脚本语言的特性

1. 熟练掌握java会非常容易掌握groovy

# Android studio搭建groovy工程

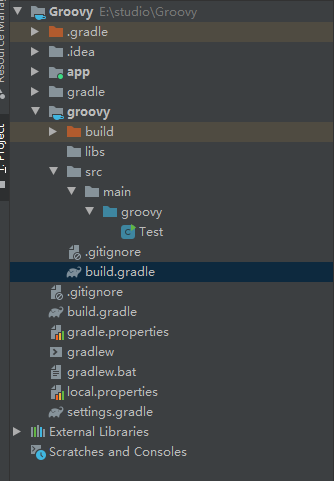
1. 新建一个android工程项目
2. 新建一个Library项目，选择Java or Kotlin Library



3．修改新建Library的build.gradle,修改后的内容如下

apply plugin: 'groovy'  
  
dependencies **{** implementation fileTree(dir: 'libs', include: ['\*.jar'])  
 implementation localGroovy()  
**}**sourceCompatibility = "1.7"  
targetCompatibility = "1.7"

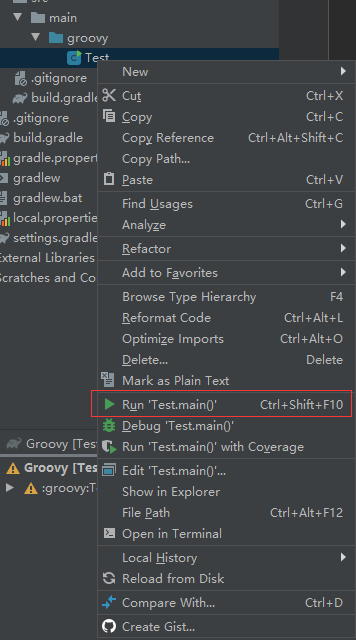
4.新建工程的main目录下创建groovy目录，删除掉原来的java目录



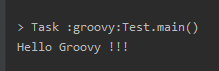
1. 在groovy目录下右键新建一个文件Test.groovy，输入如下内容

class Test {  
 // Groovy 中的 main 函数  
 def static main(def args) {  
 // 在 Groovy 中可以使用 Java 语法  
 System.*out*.println("Hello Groovy !!!")  
 }  
}

1. 右键Test.groovy文件运行



1. 在控制台查看运行结果



# Groovy 语法详解

## 变量

### 变量类型

Groovy变量没有基本类型变量，所有的变量都是对象类型。在定义变量的时候，无论定义类型是基本类型还是对象类型，最后都会被统一转换成对象类型。

打开android studio，创建一个包名variable,然后new file，新建一个variablestudy.groovy的文件，android studio创建的文件默认是没有添加包名的，idea创建的是自动添加了包名的，自己添加上包名package variable

int x = 10

println x.class  
  
double y = 2.0  
println y.class  
  
Float z = 2.0  
println z.class

输出结果：

class java.lang.Integer

class java.lang.Double  
class java.lang.Float

### 变量的定义

Groovy变量定义有强类型定义和弱类型定义2种方式。

强类型定义是在声明一个变量的时候，同时指定变量的类型。

弱类型定义不需要提前指定变量的类型，可以通过def关键字告诉编译器声明变量。

def a = 1

println a.class  
def b = 2.0  
println b.class  
def c = 'hello'  
println c.class

输出结果：

class java.lang.Integer

class java.math.BigDecimal  
class java.lang.String

一般而言，如果变量只有在本模块内部使用，推荐使用弱类型定义方式，可以方便转换成其他需要的类型。如果变量用于其他模块或者其他类，推荐使用强类型定义方式，这种方式定义的变量不能动态转换为其他类型，保证了外界传入的值是正确的。

使用def定义的变量能够随时改变它的类型

def a = 1

println a.class  
a = 'hello'  
println a.class

输出结果：

class java.lang.Integer

class java.lang.String

使用强类型定义的类型是具体的某一种类型。使用弱类型定义的类型是Object类型，后面可以根据赋值改变成任意类型。

def a = 1  
println a.class  
a = 2.0  
println a.class  
a = 'hello'  
println a.class

输出结果：

class java.lang.Integer  
class java.math.BigDecimal  
class java.lang.String

## 字符串

字符串对应Java中的String类，我们可以直接使用String。Groovy也提供了另外一个Gstring字符串类，可以通过GString类定义各种字符串。

### GString常用的三种定义方式

#### 单引号

在groovy中可以直接使用单引号’’定义字符串，使用单引号定义的字符串和Java中使用双引号定义的字符串是一样的，没有任何区别，而且不能更改。通过单引号定义的字符串类型实际上就是Java中的java.lang.String类。

def test = 'hello test'

println test.class

输出结果

class java.lang.String

如果字符串中有一些特殊字符，在groovy中是通过\转义，和Java是一样的。

def test = 'hello \'\'test'

println test

输出结果：

hello ''test

#### 三单引号

在groovy中也可以通过3个单引号来定义字符串。

def test = '''hello test'''

println test  
println test.class

输出结果：

hello test

class java.lang.String

使用1单引号定义字符串和使用3单引号定义字符串的类型都是java.lang.String。

使用1单引号定义的字符串是没有格式的，需要通过+号来对字符串进行拼接。而3单引号定义的字符串是允许有格式的，可以直接按照定义的格式进行输出，比如下面的换行：

def test2 = '''

hello

你好'''  
println test2

输出结果：

hello

你好

上面的定义方式字第一行会有一个换行，如果要把这个消除掉，只需要在每行末尾添加\，这样就能把格式忽略。

def test2 = '''\

hello\  
你好\  
'''  
println test2

输出结果：

hello你好

#### 双引号定义字符串

先看一段例子

def test = "hello test"

println test.class

输出结果：

class java.lang.String

使用双引号定义的字符串类型同样是java.lang.String，使用双引号定义的字符串又叫可扩展字符串，如果使用单引号或3单引号定义的字符串是不能包含变量的，内容是无法动态改变的。如果使用双引号定义的字符串内容是可以通过变量去填充的。

def variable = "你好"

def test = "hello ${variable}"  
println test  
println test.class

输出结果：

hello 你好

class org.codehaus.groovy.runtime.GStringImpl

当字符串使用这种可扩展表达式，类型也会被转换成GString的具体实现类 GStringImpl 类型，扩展表达式{}里并不局限使用变量，还可以是其他任意表达式类型：

def sum = "1 + 1 = ${1 + 1}"

println test

输出结果：

1 + 1 = 2

通过单引号，双引号，三单引号定义的字符串可以自由转换，编译器内部已经做了处理，编写代码时并不需要进行强制转换。

def sum = "1 + 1 = ${1 + 1}"

def msg = getMsg(sum)  
println msg  
println msg.class  
String getMsg(String msg) {  
 return msg  
}

输出结果：

1 + 1 = 2

class java.lang.String

## Groovy字符串常用API

Groovy对String的API进行了一系列扩展，这些方法主要来源如下：

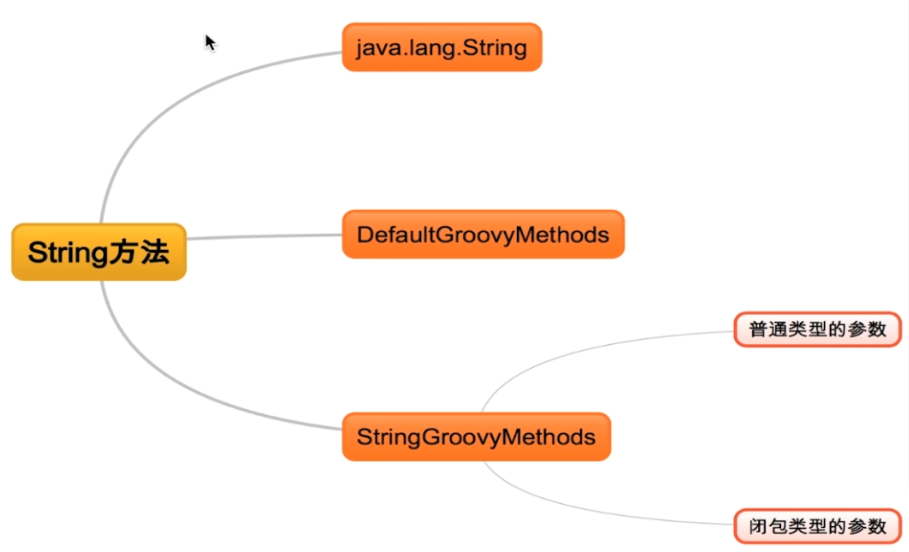
1. Java.lang.String
2. DefaultGroovyMethods

这是groovy对所有对象的扩展

1. StringGroovyMethods

重写了DefaultGroovyMethods里面的大部分方法，使之更适合String的特性。这里包含了2类方法，普通类型参数和闭包类型参数方法。

### String常用API



#### center

对已有字符串为中心向2边填充，当需要填充的字符数是单数时，字符串首填充的字符数是（（需要填充的字符数 – 1 ）/ 2），如果第二个参数不传入，默认使用空格填充

def str = "groovy"

println str.center(10, 'a')

输出结果：

aagroovyaa

#### padLeft

对已有字符串的左边进行填充。

def str = "groovy"

println str.padLeft(10, 'a')

输出结果：

aaaagroovy

#### 字符串比较

Grvooy除了可以使用java的compareTo方法对字符串进行比较，还可以使用==，>，<比较操作符对字符串进行比较。

def str = "groovy"

def str2 = 'hello'  
println str == str2

输出结果：

false

#### 通过索引获取字符

Groovy除了可以通过java方法getAt方法获取索引下的字符，还可以通过[下标]的方式进行获取。其中[]内部也可以传入范围，比如[0..3]，表示0-3的范围。

def str = "groovy"

println str[3..5]

输出结果：

ovy

#### 差运算

Groovy提供了字符串差运算方法minus（），表示把字符串包含另外字符串的内容去除。

def str = "hello groovy"

def str2 = "hello"  
println str.minus(str2)

输出结果：

groovy

groovy可以对字符串进行减法运算，相当于minus（）的功能。

def str = "hello groovy"

def str2 = "hello"  
println str - str2

输出结果：

groovy

#### 字符串倒序

使用reverse()方法对字符串进行倒序。

def str = "hello groovy"

println str.reverse()

输出结果：

yvoorg olleh

#### 首字母大写

使用capitalize()方法对字符串首字母大写

def str = "hello groovy"

println str.capitalize()

输出结果：

Hello groovy

#### 判断是否是数字类型

通过isNumber判断是否是数字类型的字符串

转换方法toIngeger()

def str = "hello groovy"

def str2 = '123456'  
println str.isNumber()  
println str2.isNumber()

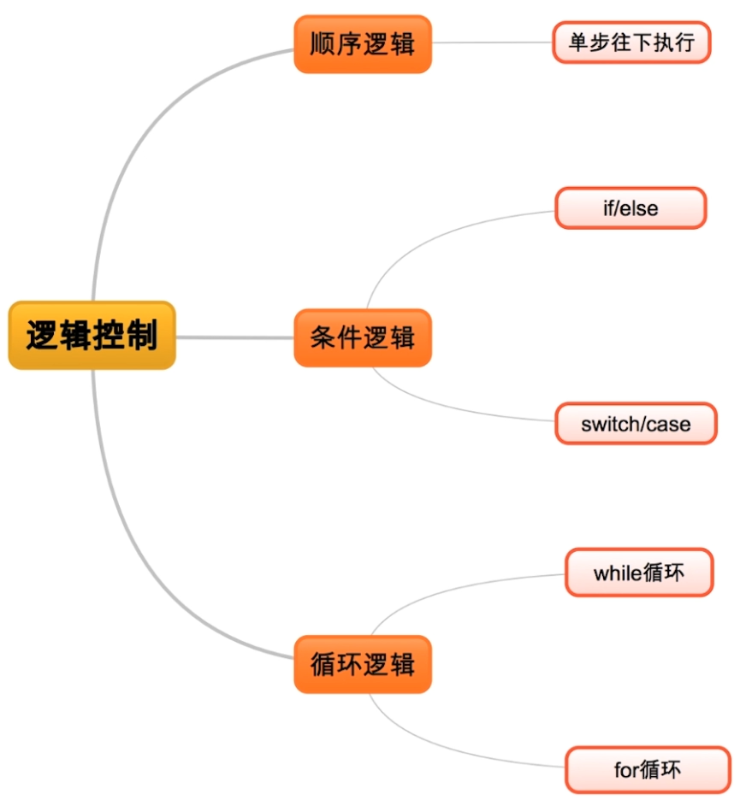
输出结果：

false

true

# Groovy逻辑控制

## Groovy逻辑控制介绍



Grvoovy逻辑控制主要包括：

### 顺序逻辑

按照正常顺序但不往下执行

### 条件逻辑

包括if/else和switch/case，其中if/else和Java使用基本一样，groovy对switch/case做了比较大的扩展。

### 循环逻辑

包括while循环和for循环，其中while和Java使用基本一样，groovy对for循环新增了一些特性。

## 对switch/case的扩展

Groovy中的switch/case中的判断参数可以传入任意类型，包括对象类型，从这个特性上来讲，完全可以取代if/else。

def x = 1.23

def result  
switch(x) {  
 case 'foo':  
 result = 'found foo'  
 break  
 case 'bar':  
 result = 'bar'  
 break  
 case [4, 5, 6, 'inList']:  
 result = 'list'  
 break  
 case 12..30:  
 result = 'range'  
 break  
 case Integer:  
 result = 'integer'  
 break  
 case BigDecimal:  
 result = 'big decimal'  
 break  
 default:  
 result = 'default'  
}  
println result

输出结果

big decimal

switch/case允许有多个匹配的值，例如上面代码修改成：

case [1.23, 4, 5, 6, 'inList']:

result = 'list'  
 break

输出结果：

list

因为它优先匹配到list

## for循环

### 对范围的for循环

Groovy中for循环对返回数据结构循环写法如下：

def sum = 0

for(i in 0..9) {  
 sum += i  
}  
println sum

输出结果：

45

### 对List的循环

def sum = 0

for(i in [1, 2, 3, 4, 5, 6, 7, 8, 9]) {  
 sum += i  
}  
println sum

输出结果：

45

### 对Map循环

对Map的循环可以直接获取它的key和value进行循环，不需要像Java一样必须获取迭代器。

def sum = 0

for(i in ['a': 1, 'b': 2, 'c': 3]) {  
 sum += i.value  
}  
println sum

输出结果

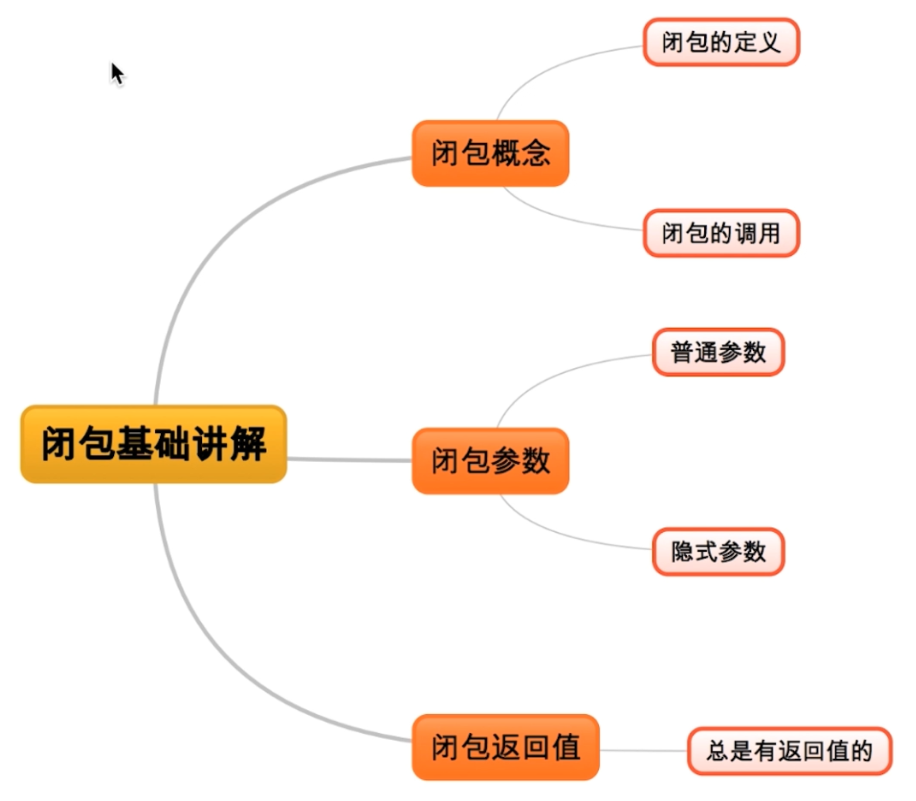
6

# 闭包

## Groovy中闭包基础知识

闭包其实就是一段代码块，在使用时和方法很类似，可是与方法有较大的差别。

主要包括闭包概念，闭包参数以及闭包返回值。



### 闭包的定义和调用

简单闭包定义：

def clouser = {println 'hello groovy'}

clouser.call()  
//clouser()

输出结果：

hello groovy

### 闭包参数

定义有参数的闭包，需要使用->，在->之前是参数，->之后是闭包体。

def clouser = {String name ->println "hello ${name}"}

//clouser.call('groovy')  
clouser('groovy')

输出结果：

hello groovy

如果有多个参数，使用逗号隔开。

def clouser = {String name, int age ->

println "hello ${name}, age is ${age}"}  
clouser('groovy', 4)

输出结果：

hello groovy, age is 4

任意闭包有一个默认参数，不需要显示申明，用it表示。

def clouser = {

println "hello ${it}"}  
clouser('groovy')

输出结果：

hello groovy

如果不想闭包中有it这个隐式参数，只需要定义一个显示参数，这样闭包就没有it这个隐式参数了。

### 闭包返回值

正常流程下，使用return返回

def clouser = {

return "hello ${it}"  
}  
def result = clouser('groovy')  
println result;

输出结果：

hello groovy

闭包一定会有返回值，默认返回最后一句代码的结果

def clouser = {

println "hello ${it}"  
}  
def result = clouser('groovy')  
println result

输出结果：

hello groovy

null

## Groovy闭包的使用详解

闭包用途非常多，这里列举闭包常用的4个方面：

### 与基本类型的结合使用

Groovy增加了upto和downto函数来进行循环。

int x = fab(5)

println x  
int y = fab(5)  
print y

// 求指定number的阶乘  
int fab(int number) {  
 int result = 1  
 1.upto(number, { num -> result \*= num})  
 return result  
}  
  
int fab2(int number) {  
 int result = 1  
 number.downto(1) {  
 num -> result \*= num  
 }  
 return result  
}

输出结果：

120

120

m.upto(n) 方法是从m开始循环到n，步长是1，循环n-m次。

m.downto(n) 方法是从m开始循环到n，步长是-1，循环m-n次。

如果闭包作为方法参数最后一个，可以写在括号体内，也可以写在括号体外，两种写法效果一样。

还有一个常用的方法n.times()方法, 表示循环n次，循环变量it从0开始到n结束，因为循环从0开始，所以不能用来求阶乘

int x = cal(5)

println x  
  
int cal(int number) {  
 int result = 0  
 number.times {  
 num ->  
 result += num  
 }  
 return result  
}

输出结果：

10

还有其他与基本类型使用的方法，具体可以从DefaultGroovyMethods.java中查看，闭包参数可以从源码查看，具体查看它调用闭包的时候有没有传入参数，定义的闭包需要匹配方法传入的参数。

### 与String结合使用

Groovy字符串提供了each()方法来进行遍历，返回值是字符串本身。代码：

String str = 'the 2 and 3 is 5'

str.each {  
 String temp -> print temp  
}

输出结果：

the 2 and 3 is 5

通过find()方法进行查找，find方法闭包必须返回一个布尔类型的值，满足条件时将会作为find方法的返回，它只查找第一个满足条件的值。

String str = 'the 2 and 3 is 5'

println str.find {  
 String s -> s.isNumber()  
}

输出结果：

2

findAll()方法，查找所有满足条件的字符，用法和find()方法一样

String str = 'the 2 and 3 is 5'

def list = str.findAll {  
 String s -> s.isNumber()  
}  
println list.toListString()

输出结果：

[2, 3, 5]

经常使用的还有any()和every()方法，any()方法表示字符串是否含有满足条件的字符。

String str = 'the 2 and 3 is 5'

def result = str.any {  
 String s -> s.isNumber()  
}  
println result

输出结果：

true

表示字符串里面还有数字字符。

every()方法表示字符串中是否每个字符都满足条件，如果是，返回true，否则返回false。

String str = 'the 2 and 3 is 5'

def result = str.every {  
 String s -> s.isNumber()  
}  
println result

输出结果：

false

collect()方法，表示将字符串每个元素都应用于闭包，闭包会产生一个新的结果，然后将结果添加到List中进行返回。

String str = 'the 2 and 3 is 5'

def list = str.collect {  
 it.toUpperCase()  
}  
println list

输出结果：

[T, H, E, , 2, , A, N, D, , 3, , I, S, , 5]

### 与数据结构结合使用

### 与文件等结合使用

## 闭包关键变量

闭包自带this，owner和delegate关键字。下面写一段测试代码，分别输出闭包这3个变量的值：

def testClouser = {

println "testClouser this:" + this  
 println "testClouser owner:" + owner  
 println "testClouser delegate:" + delegate  
}  
  
testClouser.call()

输出结果：

testClouser this:variable.variablestudy@662b4c69

testClouser owner:variable.variablestudy@662b4c69  
testClouser delegate:variable.variablestudy@662b4c69

可以看到这3个变量输出的值是一样的。

this：表示闭包定义处的类

owner：表示闭包定义处的类或者对象，如果定义在闭包中，那么将指向这个闭包对象

delegate：表示任意对象，有一个默认值和owner一致

下面多定义几个闭包，测试不同场景下3个参数所代表的的对象。

class Person {

def static *classClouser* = {  
 println "testClouser this:" + this  
 println "testClouser owner:" + owner  
 println "testClouser delegate:" + delegate  
 }  
  
 def static say() {  
 def classClouser = {  
 println "methodTestClouser this:" + this  
 println "methodTestClouser owner:" + owner  
 println "methodTestClouser delegate:" + delegate  
 }  
 classClouser.call()  
 }  
}  
Person.*classClouser*.call()  
Person.*say*()

输出结果：

testClouser this:class variable.Person

testClouser owner:class variable.Person  
testClouser delegate:class variable.Person  
methodTestClouser this:class variable.Person  
methodTestClouser owner:class variable.Person  
methodTestClouser delegate:class variable.Person

输出结果指向的是类的字节码，不是具体的对象，是因为我们定义的都是static的闭包。

把代码中的static去掉：

class Person {

def classClouser = {  
 println "testClouser this:" + this  
 println "testClouser owner:" + owner  
 println "testClouser delegate:" + delegate  
 }  
  
 def say() {  
 def classClouser = {  
 println "methodTestClouser this:" + this  
 println "methodTestClouser owner:" + owner  
 println "methodTestClouser delegate:" + delegate  
 }  
 classClouser.call()  
 }  
}  
Person p = new Person()  
p.classClouser.call()  
p.say()

输出结果：

testClouser this:variable.Person@28f2a10f

testClouser owner:variable.Person@28f2a10f  
testClouser delegate:variable.Person@28f2a10f  
methodTestClouser this:variable.Person@28f2a10f  
methodTestClouser owner:variable.Person@28f2a10f  
methodTestClouser delegate:variable.Person@28f2a10f

这时候的输出就指向了Person具体对象。

这表明闭包对象this，owner，delegate默认指向最近的封闭类。

下面在闭包中定义闭包，看看这3个变量的值是否一样。

def nestClouser = {

def innerClouser = {  
 println "innerClouser this:" + this  
 println "innerClouser owner:" + owner  
 println "innerClouser delegate:" + delegate  
 }  
 innerClouser.call()  
}  
nestClouser.call()

输出结果：

innerClouser this:variable.variablestudy@28f2a10f

innerClouser owner:variable.variablestudy$\_run\_closure2@7205765b  
innerClouser delegate:variable.variablestudy$\_run\_closure2@7205765b

可以看到，this指向了闭包定义处的类，owner和delegate一样，都指向了闭包。

如果在类或方法中定义闭包，this，owner，delegate3个变量的值是一样的。如果在闭包里面嵌套定义了闭包，this与owner和delegate的值就不一样了。this还会指向闭包定义处的类的实例或类本身。owner和delegate会指向最近的闭包对象。

那什么情况下owner和delegate会不同了，在代码中手动修改delegate的值：

class Person {

}  
Person p = new Person()  
  
def nestClouser = {  
 def innerClouser = {  
 println "innerClouser this:" + this  
 println "innerClouser owner:" + owner  
 println "innerClouser delegate:" + delegate  
 }  
 innerClouser.delegate = p  
 innerClouser.call()  
}  
nestClouser.call()

输出结果：

innerClouser this:variable.variablestudy@7133da86

innerClouser owner:variable.variablestudy$\_run\_closure2@20ccf40b  
innerClouser delegate:variable.Person@2fb3536e

当认为修改delegate的值时，owner和delegate就会出现不一样的情况。

总结：

在大多数场景下，this，owner和delegate的值都是一样的。

在闭包中定义闭包时，this与owner和delegate的值不一样。

在修改闭包的delegate参数时，owner和delegate的值不一样。

this和owner是不可以修改的，delegate可以做做任意修改。

## 闭包委托策略

先看一段简单的代码：

class Student {

String name  
 def pretty = {"My name is ${name}"}  
 String toString() {  
 pretty.call()  
 }  
}  
  
class Teacher {  
 String name  
}  
  
def stu = new Student(name : 'Sarash')  
def tea = new Teacher(name : 'Teacher')  
println stu.toString()

输出结果：

My name is Sarash

把上面代码的pretty闭包的delegate和resolveStrategy修改一下：

class Student {

String name  
 def pretty = {"My name is ${name}"}  
 String toString() {  
 pretty.call()  
 }  
}  
  
class Teacher {  
 String name  
}  
  
def stu = new Student(name : 'Sarash')  
def tea = new Teacher(name : 'Teacher')  
stu.pretty.delegate = tea  
stu.pretty.resolveStrategy = Closure.*DELEGATE\_FIRST*println stu.toString()

输出结果：

My name is Teacher

从结果可以看出闭包优先从tea查找变量，找到之后进行输出。

如果把Teacher的name改成其他命名，由于我们设置了Closure.DELEGATE\_FIRST，它先从委托对象寻找，然后从owner对象寻找，最后在Student找到了name属性，进行输出。如果改成Closure.DELEGATE\_ONLY，name在Teacher对象找不到name属性，代码会报错。

# 数据结构

## groovy列表

### 列表定义

在java中我们定义List的方式如下：

def list = new ArrayList() // java 定义方式

在groovy直接使用[]定义List：

def list = [1, 2, 3, 4, 5]

println list.class  
println list.size()

输出结果：

class java.util.ArrayList

5

可以看到这样定义的List，groovy初始化的是ArrayList。

groovy中定义数组：

def list = [1, 2, 3, 4, 5] as int[]

int [] array = [1, 2, 3, 4, 5]  
println list.class  
println list.size()  
println array.class  
println array.size()

输出结果：

class [I

5  
class [I  
5

### 列表操作

#### 增加

列表增加元素：

def list = [1, 2, 3, 4, 5]

list.add(6)  
list.leftShift(7)  
list << 8  
println list.toListString()  
def plusList = list + 9  
println plusList.toListString()

输出结果：

class java.util.ArrayList

[1, 2, 3, 4, 5, 6, 7, 8]  
[1, 2, 3, 4, 5, 6, 7, 8, 9]

#### 删除

list.remove(7)

list.remove((Object) 7)  
list.removeAt(7)  
list.removeElement(6)  
list.removeAll { return it % 2 == 0 }  
println list - [6, 7]  
println list.toListString()

输出结果：

class java.util.ArrayList

[1, 3, 5, 9]  
[1, 3, 5, 9]

#### 查找

可以通过find方法查找符合第一个条件的元素，下面实现查找列表中第一个偶数：

def findList = [6, -3, 9, 2, -7, 1, 5]

int result = findList.find {  
 return it % 2 == 0  
}  
println result

输出结果：

6

通过findAll方式查找所有满足条件的元素：

def findList = [6, -3, 9, 2, -7, 1, 5]

def result = findList.findAll {  
 return it % 2 != 0  
}  
println result.toListString()

输出结果：

[-3, 9, -7, 1, 5]

通过any()方法判断列表中是否有一个满足条件的原色。

def findList = [6, -3, 9, 2, -7, 1, 5]

def result = findList.any {  
 return it % 2 != 0  
}  
println result

输出结果：

true

通过every()判断列表中所有的元素是否满足条件：

def findList = [6, -3, 9, 2, -7, 1, 5]

def result = findList.every {  
 return it % 2 == 0  
}  
println result

输出结果：

false

通过min()和max()查找列表中最小值和最大值：

def findList = [6, -3, 9, 2, -7, 1, 5]

println findList.min()  
println findList.max()

输出结果：

-7

9

min()和max()方法也能够传入闭包，查找绝对值最小和最大的例子：

def findList = [6, -3, 9, 2, -7, 1, 5]

println findList.min{return Math.*abs*(it)}  
println findList.max{return Math.*abs*(it)}

输出结果：

1

9

可以通过count对列表中的元素进行统计

def findList = [6, -3, 9, 2, -7, 1, 5]

int num = findList.count {return it % 2 == 0}  
println num

输出结果：

2

#### 排序

使用Coloections.sort()会使用默认的规则（从小到大）对列表进行排序。

def sortList = [6, -3, 9, 2, -7, 1, 5]

Collections.*sort*(sortList)  
println sortList

输出结果：

[-7, -3, 1, 2, 5, 6, 9]

Coloections.sort()还有一个重载的函数，我们可以自定义排序规则，下面实现按照绝对值从小到大输出：

def sortList = [6, -3, 9, 2, -7, 1, 5]

Comparator mc = {a, b ->  
 a == b ? 0:  
 Math.abs(a) < Math.abs(b) ? -1 : 1  
}  
Collections.*sort*(sortList, mc)  
println sortList

输出结果：

[1, 2, -3, 5, 6, -7, 9]

在groovy中有可以直接调用sort()方法进行排序。

def sortList = [6, -3, 9, 2, -7, 1, 5]

sortList.sort()  
println sortList

输出结果：

[-7, -3, 1, 2, 5, 6, 9]

可以在sort方法中传入一个闭包，自定义排序规则。

sortList.sort{a, b ->

a == b ? 0:  
 Math.*abs*(a) < Math.*abs*(b) ? 1 : -1  
}  
println sortList

输出结果：

[9, -7, 6, 5, -3, 2, 1]

对字符串类型列表的排序：

def sortList = ['abc', 'z', 'Hello', 'groovy', 'java']

sortList.sort{  
 return it.size()  
}  
println sortList

输出结果：

[z, abc, java, Hello, groovy]

## groovy映射

### map定义

groovy中映射定义如下：

def colors = [

red : 'ff0000',  
 green : '00ff00',  
 blue : '0000ff']

这和列表方式定义一样，使用[]进行定义，不同的是在[]中Map是以key-value的形式给出没给元素的初始值。

### 索引方式

groovy提供了下标操作符的方式获取对应元素的值：

println colors['red']

输出结果：

ff000000

也可以通过以下方式获取map对应元素的值：

println colors.green

输出结果：

00ff00

### 添加元素

groovy可以直接通过map.key=value的方式进行添加元素

colors.yellow = 'ffff00'  
println colors.yellow

输出结果：

ffff00

map既可以添加同类型的元素，也可以添加不同类型的元素，这个是groovy中map比较强大的特性：

colors.complex = **[**a : 1, b : 2**]**

println colors.toMapString()

输出结果：

[red:ff0000, green:00ff00, blue:0000ff, yellow:ffff00, complex:[a:1, b:2]]

注意问题：

1. map中的key的类型一般是不可变字符串或者是Number。
2. key不用单引号和有单引号效果相同，没有单引号编译器会默认使用单引号字符串。

groovy定义的map默认是LinkedHashMap，如果不想使用LinkedHashMap，可以在定义的时候指定类型，或者使用关键字as指定类型。

HashMap colors = []

def colors = [] as HashMap

获取map类型时需要通过getClass()方法进行获取，不能直接通过class方法获取，因为通过class方法获取会被指定为在map中查找class的值。

### 删除元素

groovy中map删除元素使用remove()方法：

def colors = [

red : 'ff0000',  
 green : '00ff00',  
 blue : '0000ff']  
colors.remove('red')  
println colors.toMapString()

输出结果：

[green:00ff00, blue:0000ff]

### map常用操作

首先定义一个map如下：

def students = [

1 : [number : '0001', name : 'Bob', score : 55, sex : 'male'],  
 2 : [number : '0002', name : 'Johnny', score : 60, sex : 'female'],  
 3 : [number : '0003', name : 'Claire', score : 65, sex : 'female'],  
 4 : [number : '0004', name : 'Amy', score : 70, sex : 'male']  
]

#### 遍历

使用each()方法进行遍历：

students.each { def student ->

println "the key is ${student.key}, " +  
 "the value is ${student.value}"  
}

输出结果：

the key is 1, the value is [number:0001, name:Bob, score:55, sex:male]

the key is 2, the value is [number:0002, name:Johnny, score:60, sex:female]  
the key is 3, the value is [number:0003, name:Claire, score:65, sex:female]  
the key is 4, the value is [number:0004, name:Amy, score:70, sex:male]

也可以使用eachWithIndex()，和each()不同的是，eachWithIndex()多了一个index参数：

students.eachWithIndex { def student, int index ->

println "index is ${index}, the key is ${student.key}, " +  
 "the value is ${student.value}"  
}

输出结果：

index is 0, the key is 1, the value is [number:0001, name:Bob, score:55, sex:male]

index is 1, the key is 2, the value is [number:0002, name:Johnny, score:60, sex:female]  
index is 2, the key is 3, the value is [number:0003, name:Claire, score:65, sex:female]  
index is 3, the key is 4, the value is [number:0004, name:Amy, score:70, sex:male]

还可以使用each()方法，只不过闭包参数时key，value的方法进行遍历：

students.each { key, value ->

println "the key is ${key}, " +  
 "the value is ${value}"  
}

输出结果：

the key is 1, the value is [number:0001, name:Bob, score:55, sex:male]

the key is 2, the value is [number:0002, name:Johnny, score:60, sex:female]  
the key is 3, the value is [number:0003, name:Claire, score:65, sex:female]  
the key is 4, the value is [number:0004, name:Amy, score:70, sex:male]

使用上面的方式也可以带第三个参数index：

students.eachWithIndex { key, value, index ->

println "this index is ${index}, the key is ${key}, " +  
 "the value is ${value}"  
}

输出结果：

this index is 0, the key is 1, the value is [number:0001, name:Bob, score:55, sex:male]

this index is 1, the key is 2, the value is [number:0002, name:Johnny, score:60, sex:female]  
this index is 2, the key is 3, the value is [number:0003, name:Claire, score:65, sex:female]  
this index is 3, the key is 4, the value is [number:0004, name:Amy, score:70, sex:male]

#### 查找

通过find()方法进行查找：

def entry = students.find { def student ->

return student.value.score >= 60  
}  
println entry

输出结果：

2={number=0002, name=Johnny, score=60, sex=female}

使用findAll查找全部满足条件的数据:

def entrys = students.findAll { def student ->

return student.value.score >= 60  
}  
println entrys

输出结果：

[2:[number:0002, name:Johnny, score:60, sex:female], 3:[number:0003, name:Claire, score:65,

sex:female], 4:[number:0004, name:Amy, score:70, sex:male]]

使用count()进行数据统计：

def count = students.count { def student ->

return student.value.score >= 60 &&  
 student.value.sex == 'male'  
}  
println count

输出结果：

1

查找所有score大于等于60的姓名：

def names = students.findAll { def student ->

return student.value.score >= 60  
}.collect {  
 return it.value.name  
}  
println names.toListString()

输出结果：

[Johnny, Claire, Amy]

使用groupBy对map中元素进行分组：

def group = students.groupBy { def student ->

return student.value.score >= 60 ? '及格': '不及格'  
}  
println group.toMapString()

输出结果：

[不及格:[1:[number:0001, name:Bob, score:55, sex:male]], 及格:[2:[number:0002, name:Johnny,

score:60, sex:female], 3:[number:0003, name:Claire, score:65, sex:female], 4:[number:0004, name:Amy, score:70, sex:male]]]

#### 排序

使用sort()方法对Map的元素进行排序，会返回一个新的Map，与List不同的是，List中的sort()方法只对List中的元素进行排序。

def sort = students.sort { def student1, def student2 ->

Number score1 = student1.value.score  
 Number score2 = student2.value.score  
 return score1 == score2 ? 0 : score1 < score2 ? -1 : 1  
}  
println sort.toMapString()

输出结果：

[1:[number:0001, name:Bob, score:55, sex:male], 2:[number:0002, name:Johnny, score:60,

sex:female], 3:[number:0003, name:Claire, score:65, sex:female], 4:[number:0004, name:Amy, score:70, sex:male]]

## 范围-range

### 范围的定义

范围的定义如下：

def range = 1..10

println range[0] // 索引  
println range.contains(10) // 判断是否包含某个元素  
println range.from // 范围起始值  
println range.to // 范围结束值

输出结果：

1

true  
1  
10

range是List的直接子类，所以操作方法和列表完全一样。

### 范围的操作

#### 范围遍历

范围遍历的列子：

def range = 1..10

range.each {  
 println it  
}  
  
for (i in range) {  
 println i  
}

结果都是循环输出1~10.

在switch..catch中使用

def result = getGrade(75)

println result  
  
def getGrade(Number number) {  
 def result  
 switch(number) {  
 case 0..<60:  
 result = '不及格'  
 break  
 case 60..<70:  
 result = '及格'  
 break  
 case 70..<80:  
 result = '良好'  
 break  
 case 80..100:  
 result = '优秀'  
 break  
 }  
 result  
}

输出结果：

良好

# 面向对象

## groovy中类，接口等的定义和使用

### 类

右键->New->Groovy Class创建一个groovy类，名字是Person。groovy默认所有的类都是public，所以创建完成的class前面没有public进行修饰。

*/\*\**

*\* 1.groovy中默认都是public  
 \* 2.所有的groovy类都默认集成了一个GroovyObject类  
 \* 这是groovy和java中类的区别  
 \*/*class Person {  
  
 String name  
  
 Integer age  
  
 def increaseAge(Integer years) {  
 this.name += years  
 }  
}

新建一个脚本groovy script，在脚本里面进行Person的使用。

def person = new Person(name: 'android', age: 26)

println "this name is ${person.name}, " +  
 "thi age is ${person.age}"

输出结果：

this name is android, thi age is 26

在groovy中直接通过.操作符操作的并不是对象的属性，实际上是调用getName()和getAge()方法。无论是直接通过.还是get/set，最终调用的都是get/set方法，这是和java不同的地方。

调用方法形式如下：

person.increaseAge(10)

### 接口

右键->New->Groovy Class，类型是Interface，创建一个Interface类型的class。groovy接口和java最大的不同是groovy不允许定义非public类型的方法。

Action.groovy

*/\*\**

*\* 接口不允许定义非public类型的额接口  
 \*/*interface Action {  
 void eat()  
  
 void drink()  
  
 void play()  
}

Person.groovy

*/\*\**

*\* 1.groovy中默认都是public  
 \* 2.所有的groovy类都默认集成了一个GroovyObject类  
 \* 这是groovy和java中类的区别  
 \*/*class Person implements Action{  
  
 String name  
  
 Integer age  
  
 def increaseAge(Integer years) {  
 this.name += years  
 }  
  
 @Override  
 void eat() {  
  
 }  
  
 @Override  
 void drink() {  
  
 }  
  
 @Override  
 void play() {  
  
 }  
}

groovy接口除了不定定义非public类型的方法外，其他和java接口使用几乎一样。

### Trait

trait和interface类似，不同之处在于trait可以为方法提供默认的实现，如果不提供默认实现，需要为方法添加abstract关键字修饰。

DefaultAction.groovy:

trait DefaultAction {

abstract void eat()  
 void play() {  
 println 'I can play'  
 }  
}

Peroson.groovy

class Person implements DefaultAction{

String name  
  
 Integer age  
  
 def increaseAge(Integer years) {  
 this.name += years  
 }  
  
 @Override  
 void eat() {  
  
 }  
  
}

使用Person对象：

def person = new Person(name: 'android', age: 26)

person.play()

输出结果：

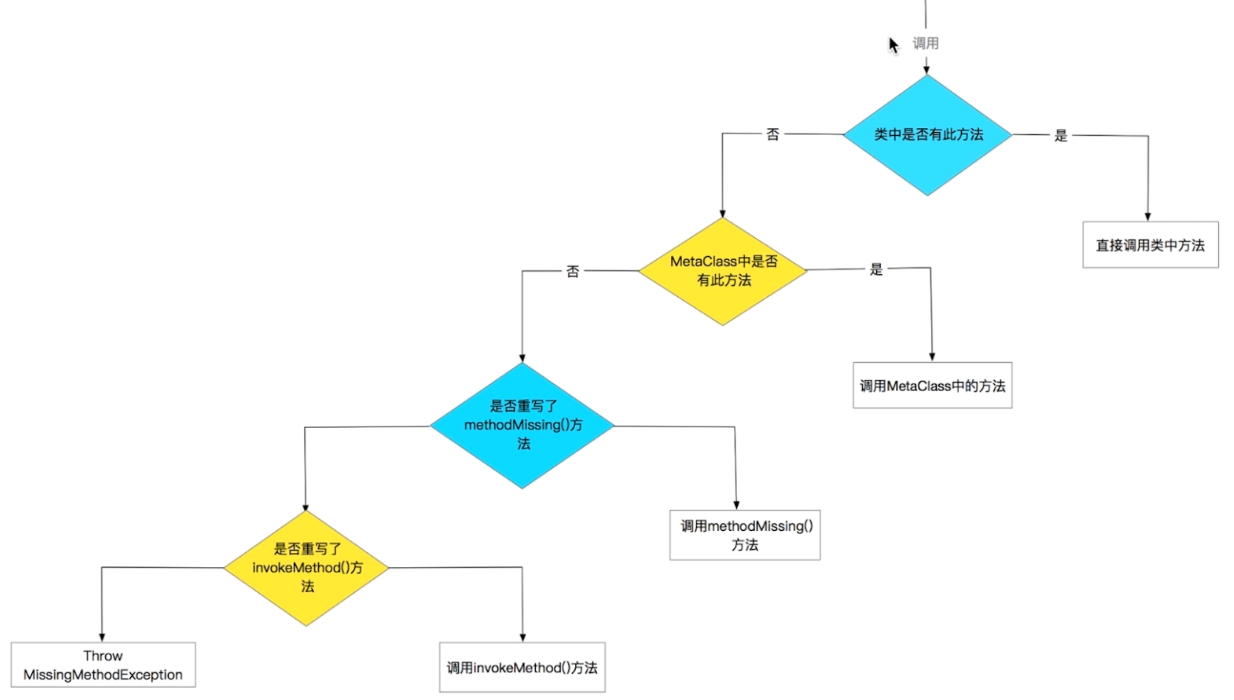
this name is android, thi age is 26

I can play

## groovy中的元编程

所谓元编程，是指编写的代码所执行的时机。

无论在java还是groovy，都会调用其他类的方法。当调用一个类的方法时，编译器会判断类中是否存在这个方法，如果存在就直接调用类中的方法。在java中找不到类方法会直接报错，但是groovy即使类中没有对应的方法，编译也是可以通过的，groovy编译器会尝试从MetaClass查找是否有此方法。如果MetaClass中包含了此方法，就会调用MetaClass中的方法。如果MetaClass也没有对应的方法，groovy编译器会继续判断是否重写了methodMissing()方法，如果重写了，就调用methodMissing()方法。如果没有重写会继续判断是否重写了invokeMethod()方法，如果重写了invokeMethod()方法，就调用该方法，否则抛出MissingMethodException异常，这是一个运行时的异常。可以用下面一幅图进行概括，同样也适用于属性调用。



### 基本流程验证

person.groovy

class Person {

String name  
  
 Integer age  
  
 def increaseAge(Integer years) {  
 this.name += years  
 }  
  
 def invokeMethod(String name, Object args) {  
 return "the methos is ${name}, the params is ${args}"  
 }  
  
 def methodMissing(String name, Object args){  
 return "the methos is ${name} is missing"  
 }  
}

使用person：

def person = new Person(name: 'android', age: 26)

println person.cry()

当没有invokeMethod()和methodMissing()方法时，运行会报出MissingMethodException，当重写了上述2个方法之后，则会按照先后书序调用对应的方法。

### 为类动态添加属性

metaClass可以为一个类动态添加属性和方法。

// 为类动态添加一个属性

Person.metaClass.sex = 'male'  
def person = new Person(name: 'android', age: 26)  
println person.sex  
person.sex = 'female'  
println "the new sex is: " + person.sex

输出结果：

male

the new sex is: female

### 为类动态添加方法

Person.metaClass.sex = 'male'

Person.metaClass.sexUpperCase = { -> sex.toUpperCase()}  
def person = new Person(name: 'android', age: 26)  
println person.sexUpperCase()

输出结果：

MALE

### 为类添加静态方法

Person.metaClass.static.createPerson = {String name, int age ->

new Person(name: name, age: age)  
}  
def person3 = Person.createPerson("androidtest", 30)  
println person3.name + " and " + person3.age

输出结果：

androidtest and 30

### 实现全局有效的功能

上面的实现只是针对本模块内有效，模块外部并不会生效，如果想实现全局有效的功能，需要使用调用ExpandoMetaClass.enableGlobally()，让其全局生效。

代码如下：

Person.groovy:

package expand

class Person {

String name  
  
 Integer age  
  
 def increaseAge(Integer years) {  
 this.name += years  
 }  
  
 def invokeMethod(String name, Object args) {  
 return "the methos is ${name}, the params is ${args}"  
 }  
  
 def methodMissing(String name, Object args){  
 return "the methos is ${name} is missing"  
 }  
}

ApplicationManager.groovy

package expand

*/\*\**

*\* 模拟应用管理类  
 \*/*class ApplicationManager {  
  
 static void init() {  
 ExpandoMetaClass.*enableGlobally*()  
 // 为第三方类添加方法  
 Person.metaClass.static.createPerson = {String name, int age ->  
 new Person(name: name, age: age)  
 }  
 }  
  
}

PersonManager.groovy

package expand

*/\*\**

*\* Person管理类  
 \*/*class PersonManager {  
 static Person createPerson(String name, int age) {  
 return Person.createPerson(name, age)  
 }  
}

Entry.groovy

package expand

class Entry {

static void main(def args) {  
 println '应用程序正在启动...'  
 //初始化  
 ApplicationManager.*init*()  
 println '应用程序初始化完成...'  
  
 def person = PersonManager.  
 *createPerson*('renzhiqiang', 26)  
 println "the person name is ${person.name} " +  
 "and the age is ${person.age}"  
 }  
}

运行Entry中的main方法，输出结果：

应用程序正在启动...

应用程序初始化完成...  
the person name is renzhiqiang and the age is 26

# Groovy 高级用法

## groovy json操作

### 转换成json

使用JsonOutput类可以直接将对象转换成Json字符串

示例：

Person.groovy

package objectorention

class Person {  
  
 String name  
  
 Integer age  
  
 def increaseAge(Integer years) {  
 this.name += years  
 }  
  
 def invokeMethod(String name, Object args) {  
 return "the methos is ${name}, the params is ${args}"  
 }  
  
 def methodMissing(String name, Object args){  
 return "the methos is ${name} is missing"  
 }  
}

jsonstudy.groovy

package file

import groovy.json.JsonOutput  
import objectorention.Person  
  
def list = [new Person(name: 'John', age: 25),  
 new Person(name: 'Major', age: 26)]  
println JsonOutput.*toJson*(list)

输出结果：

[{"age":25,"name":"John"},{"age":26,"name":"Major"}]

也可以直接输出带格式的json字符串，将代码修改如下

def list = [new Person(name: 'John', age: 25),

new Person(name: 'Major', age: 26)]  
def json = JsonOutput.*toJson*(list)  
println JsonOutput.*prettyPrint*(json)

输出结果：

[

{  
 "age": 25,  
 "name": "John"  
 },  
 {  
 "age": 26,  
 "name": "Major"  
 }  
]

### json转换成对象

groovy使用JsonSlurper类可以进行Json解析

def jsonSlpuer = new JsonSlurper()

jsonSlpuer.parse()

也可以使用第三方类库进行json解析，以使用Gson为例，创建以src同级的libs目录，把Gson的jar包拷贝到该目录下，右键将jar包Add as Library。

json转换成Object例子

Person.groovy

package objectorention

class Person {  
  
 String name  
  
 Integer age  
  
 def increaseAge(Integer years) {  
 this.name += years  
 }  
  
 def invokeMethod(String name, Object args) {  
 return "the methos is ${name}, the params is ${args}"  
 }  
  
 def methodMissing(String name, Object args){  
 return "the methos is ${name} is missing"  
 }  
}

jsonstudy.groovy

package file

import groovy.json.JsonOutput  
import groovy.json.JsonSlurper  
import objectorention.Person  
  
def list = [new Person(name: 'John', age: 25),  
 new Person(name: 'Major', age: 26)]  
def json = JsonOutput.*toJson*(list)  
println JsonOutput.*prettyPrint*(json)  
  
def jsonSlpuer = new JsonSlurper()  
def list1 = jsonSlpuer.parseText(json)  
println list1.getClass()

使用网络请求获取在线json

import groovy.json.JsonOutput

import groovy.json.JsonSlurper  
import objectorention.Person  
  
def reponse =  
 getNetworkData(  
 'http://baike.baidu.com/api/openapi/BaikeLemmaCardApi?scope=103&format=json&appid=379020&bk\_key=%E5%85%B3%E9%94%AE%E5%AD%97&bk\_length=600')  
  
println reponse.data.head.name  
  
def getNetworkData(String url) {  
 //发送http请求  
 def connection = new URL(url).openConnection()  
 connection.setRequestMethod('GET')  
 connection.connect()  
 def response = connection.content.text  
 //将json转化为实体对象  
 def jsonSluper = new JsonSlurper()  
 return jsonSluper.parseText(response)  
}

Groovy的json便不需要事先定义一个对象才能进行json解析，如上面的repoonse，在不定义对象的情况下也能直接进行解析，解析完成之后可以获取实际json中存在的字段，如果获取不存在的字段则会报错。

## xml文件操作

### groovy解析xml格式数据

groovy使用xmlSluper对xml进行解析，先定一个xml格式的字符串：

final String xml = '''

<response version-api="2.0">  
 <value>  
 <books id="1" classification="android">  
 <book available="20" id="1">  
 <title>疯狂Android讲义</title>  
 <author id="1">李刚</author>  
 </book>  
 <book available="14" id="2">  
 <title>第一行代码</title>  
 <author id="2">郭林</author>  
 </book>  
 <book available="13" id="3">  
 <title>Android开发艺术探索</title>  
 <author id="3">任玉刚</author>  
 </book>  
 <book available="5" id="4">  
 <title>Android源码设计模式</title>  
 <author id="4">何红辉</author>  
 </book>  
 </books>  
 <books id="2" classification="web">  
 <book available="10" id="1">  
 <title>Vue从入门到精通</title>  
 <author id="4">李刚</author>  
 </book>  
 </books>  
 </value>  
 </response>  
'''

#### 基本使用

下面是xml解析的一个例子，分别输出了节点和属性：

//开始解析此xml数据

def xmlSluper = new XmlSlurper()  
def response = xmlSluper.parseText(xml)  
  
println response.value.books[0].book[0].title.text()  
println response.value.books[0].book[0].author.text()  
println response.value.books[1].book[0].@available

输出结果

疯狂Android讲义

李刚  
10

#### 查询

查询所有作者是李刚的书名：

def xmlSluper = new XmlSlurper()

def response = xmlSluper.parseText(xml)  
  
def list = []  
response.value.books.each { books ->  
 //下面开始对书结点进行遍历  
 books.book.each { book ->  
 def author = book.author.text()  
 if (author.equals('李刚')) {  
 list.add(book.title.text())  
 }  
 }  
}  
println list.toListString()

输出结果：

[疯狂Android讲义, Vue从入门到精通]

groovy为我们提供了一个深度遍历的方法，通过该方法一样可以实现上述功能。

def xmlSluper = new XmlSlurper()

def response = xmlSluper.parseText(xml)  
//深度遍历xml数据  
def titles = response.depthFirst().findAll { book ->  
 return book.author.text() == '李刚' ? true : false  
}  
println titles.toListString()

深度遍历另一种写法是把depthFirst()使用’\*\*’替换，效果是一样的。

使用广度遍历方法遍历xml数据：

def xmlSluper = new XmlSlurper()

def response = xmlSluper.parseText(xml)

//广度遍历xml数据  
def name = response.value.books.children().findAll { node ->  
 node.name() == 'book' && node.@id == '2'  
}.collect { node ->  
 return node.title.text()  
}  
  
println name

输出结果：

[第一行代码]

广度遍历children()方法可以使用’\*’替代，效果一样。

### groovy创建xml格式数据

groovy生成xml主要使用MarkupBuilder类，该类是生成xml的核心类。

#### 基础使用

使用MarkupBuilder类生成xml的例子：

def sw = new StringWriter()

def xmlBuilder = new MarkupBuilder(sw) //用来生成xml数据的核心类  
//根结点langs创建成功  
xmlBuilder.langs(type: 'current', count: '3',  
 mainstream: 'true') {  
 //第一个language结点  
 language(flavor: 'static', version: '1.5', 'java') {  
 age('16')  
 }  
 language(flavor: 'dynamic', version: '1.6') {  
 age('10')  
 }  
 language(flavor: 'dynamic', version: '1.9', 'JavaScript')  
}  
  
println sw

输出结果：

<langs type='current' count='3' mainstream='true'>

<language flavor='static' version='1.5'>java  
 <age>16</age>  
 </language>  
 <language flavor='dynamic' version='1.6'>  
 <age>10</age>  
 </language>  
 <language flavor='dynamic' version='1.9'>JavaScript</language>  
</langs>

#### 将实体类数据转换成xml格式数据

def sw = new StringWriter()

def xmlBuilder = new MarkupBuilder(sw) //用来生成xml数据的核心类  
def langs = new Langs()  
xmlBuilder.langs(type: langs.type, count: langs.count,  
 mainstream: langs.mainstream) {  
 //遍历所有的子结点  
 langs.languages.each { lang ->  
 language(flavor: lang.flavor,  
 version: lang.version, lang.value)  
 }  
}  
println sw  
//对应xml中的langs结点  
class Langs {  
 String type = 'current'  
 int count = 3  
 boolean mainstream = true  
 def languages = [  
 new Language(flavor: 'static',  
 version: '1.5', value: 'Java'),  
 new Language(flavor: 'dynamic',  
 version: '1.3', value: 'Groovy'),  
 new Language(flavor: 'dynamic',  
 version: '1.6', value: 'JavaScript')  
 ]  
}  
//对应xml中的languang结点  
class Language {  
 String flavor  
 String version  
 String value  
}

输出结果：

<langs type='current' count='3' mainstream='true'>

<language flavor='static' version='1.5'>Java</language>  
 <language flavor='dynamic' version='1.3'>Groovy</language>  
 <language flavor='dynamic' version='1.6'>JavaScript</language>  
</langs>

## groovy文件处理

### java文件处理

java文件处理包括

1. 节点流，InputStream，OutputStream及其子类
2. 处理流，Reader，Writer及其子类

### groovy文件处理

所有java对文件的处理类，groovy都可以使用。

groovy拓展了许多更加快捷和强大的方法。

groovy把文件看做是一种资源，提供的方法都是通过ResourceGroovyMethods提供的。

#### 遍历文件内容

##### readLine()方法

groovy提供了eachLine()方法，可以遍历文件中每行内容：

def file = new File('../../GroovySpecIflcation.iml')

file.eachLine { line ->  
 println line  
}

##### getText()方法

groovy提供了另外一种方法获取文件的内容，这个方法就是getText()：

def file = new File('../../GroovySpecIflcation.iml')

def text = file.getText()  
println text

##### readLines()方法

也可以通过readLines()方法获取，这个方法返回的是List，它会把文件每一行内容都当做一个元素存储在List中返回。

def result = file.readLines()

##### withReader()方法

groovy通过withReader()方法可以读取某一部分的内容，与之对应的方法是withWriter()方法，提供写入一部分内容功能。

//读取文件部分内容

def reader = file.withReader { reader ->  
 char[] buffer = new char[100]  
 reader.read(buffer)  
 return buffer  
}  
println reader

#### 实现复制功能

通过withReader()和withWriter()方法实现复制文件功能：

def result = copy('../../GroovySpecIflcation.iml'

, '../../GroovySpecification2.iml')  
println result  
  
def copy(String sourcePath, String destationPath) {  
 try {  
 //首先创建目标文件  
 def desFile = new File(destationPath)  
 if (!desFile.exists()) {  
 desFile.createNewFile()  
 }  
  
 //开始copy  
 new File(sourcePath).withReader { reader ->  
 def lines = reader.readLines()  
 desFile.withWriter { writer ->  
 lines.each { line ->  
 writer.append(line + "\r\n")  
 }  
 }  
 }  
 return true  
 } catch (Exception e) {  
 e.printStackTrace()  
 }  
 return false  
}

groovy提供的文件操作方法会默认帮助处理关闭流，所以并不需要我们自己去关闭相关的文件流。

#### 对象读写

groovy使用withObjectOutputStream和withObjectInputStream方法实现对象的读写，对象存储是按照字节存储的。

def person = new Person(name: 'Qndroid', age: 26)

saveObject(person, '../../person.bin')  
  
def result = (Person) readObject('../../person.bin')  
println "the name is ${result.name} and the age is ${result.age}"  
  
def saveObject(Object object, String path) {  
 try {  
 //首先创建目标文件  
 def desFile = new File(path)  
 if (!desFile.exists()) {  
 desFile.createNewFile()  
 }  
 desFile.withObjectOutputStream { out ->  
 out.writeObject(object)  
 }  
 return true  
 } catch (Exception e) {  
 }  
 return false  
}  
  
def readObject(String path) {  
 def obj = null  
 try {  
 def file = new File(path)  
 if (file == null || !file.exists()) return null  
 //从文件中读取对象  
 file.withObjectInputStream { input ->  
 obj = input.readObject()  
 }  
 } catch (Exception e) {  
  
 }  
 return obj  
}

Person.groovy

class Person implements Serializable {

String name  
  
 Integer age  
  
 def increaseAge(Integer years) {  
 this.name += years  
 }  
  
 def invokeMethod(String name, Object args) {  
 return "the methos is ${name}, the params is ${args}"  
 }  
  
 def methodMissing(String name, Object args){  
 return "the methos is ${name} is missing"  
 }  
}

# gradle 生命周期

## gradle基本概念

使用过android studio都知道，android studio默认是使用gradle来构建程序的。gradle是一种构建工具，也是一种编程框架。

gradle由三部分组成：

1. groovy核心语法
2. build script block
3. gradle api

gradle可以通过api去灵活实现构建需求，这也是gradle核心和特色。

## gradle优势

灵活性，gradle可以修改构建脚本，使其能够按照我们的需求进行构建。

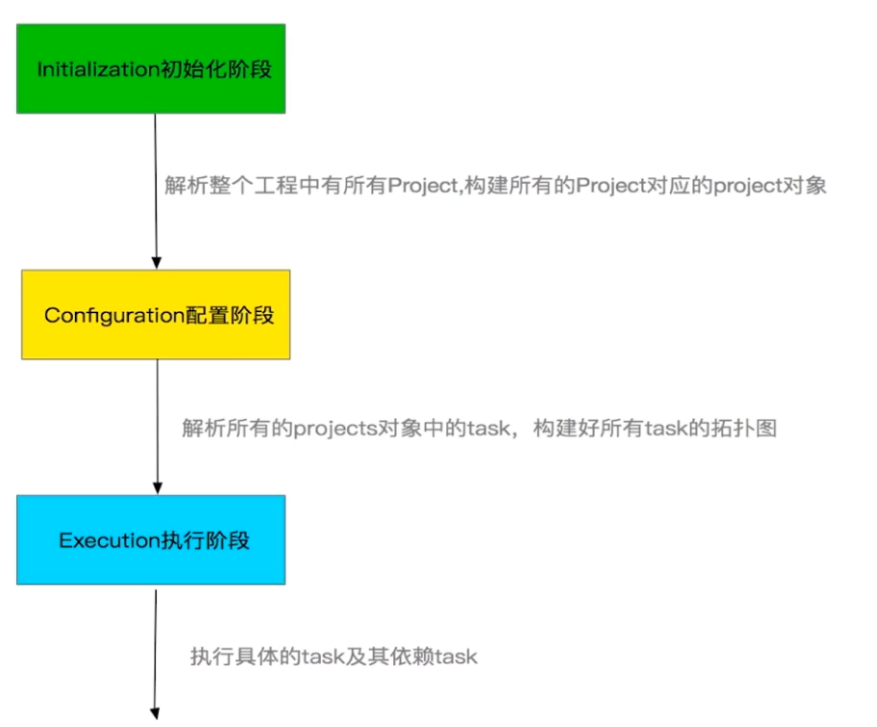
粒度性，gradle从源代码编译，资源编译，生成apk都是通过一系列的task实现的，task内部源码都是开源的。

拓展性，gradle支持插件机制

兼容性，兼容其他打包工具（例如art，meven）的功能

## gradle生命周期

gradle生命周期大致流程如下：



gradle的生命周期包括3个阶段：

1. Initialization初始化阶段：解析整个工程中所有的Project，构建所有的Project对应的project对象。
2. Configuration 配置阶段：解析所有的projects对象中的task，构建好所有task的拓扑图。
3. Execution 执行阶段：执行具体的task及其依赖task。

## 主要生命周期监听方法

beforeEvaluate：在初始化阶段和配置阶段之间的监听，配置阶段执行之前。

afterEvaluate：在配置阶段之后，执行阶段之前监听。

gradle.buildFinished：gradle生命周期执行完成之后的监听

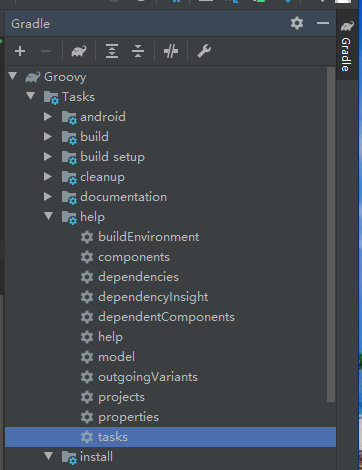
gradle.beforProject:等同beforeEvaluate

gradle. afterProject: 等同afterEvaluate

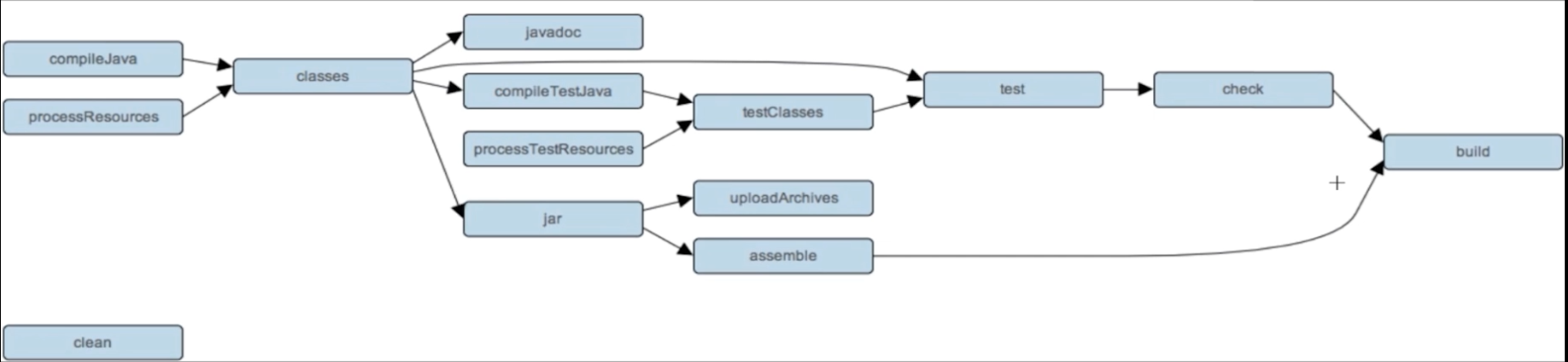
还有一种添加监听的方式，在这里不做详细讲解。

初始化阶段监听，在setting.gradle里面进行监听，初始化阶段实际上就是执行了setting.gradle

为什么执行gradlew build 会有其他那么的输出，因为build其实不是一个命令，它是一个Task。如果要查看gradle的task，可以使用命令 gradlew task进行查看，也可以使用gradle面板的进行查看



下面是一张java的构件图

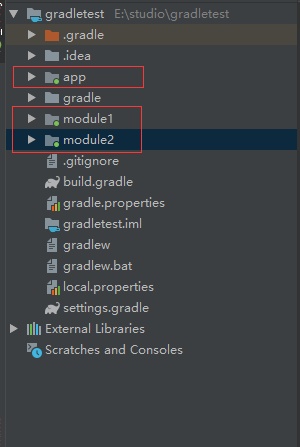


如果放到android，这里会更加复杂。从上图可以看到，build依赖的task其实有很多，当我们执行build的时候，需要把依赖的task都执行完，所以在控制台可以看到很多的输出信息。这个就是一个有向无环图，这个图就是在配置阶段执行完成之后生成的。

# project详解

## project了解

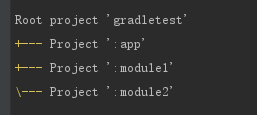
Project在gradle中包括根Project和子Project，例如下面的项目包括3个project，分别是app，module1，module2.



我们可以在命令上输入以下命令验证一下：

gradlew project

输出：

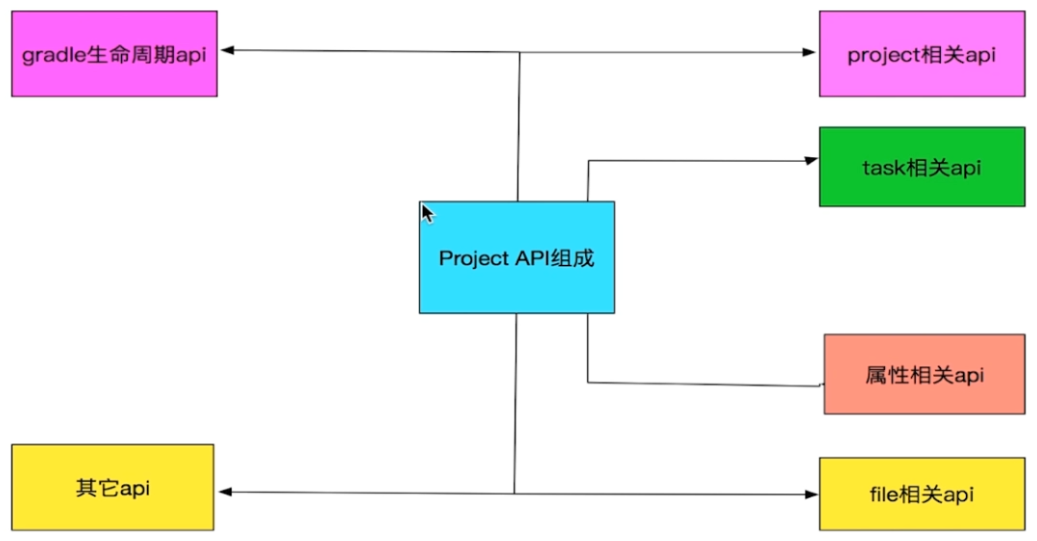


而且每一个project下都包含一个build.gradle文件，这是一个Project的标配，Project的配置和管理，最终都是由build.gradle文件实现。

## project 核心API讲解

### project 核心api概述

project的api可以使用一下一张图进行总结：



project相关api：让当前project拥有操作父project以及管理子project的能力

task相关api：为当前project提供新增task以及使用当前project已有task的能力

属性相关api：让我们拥有为project添加额外属性的能力

file相关api：操作当前project下文件的处理

生命周期api：和gradle生命周期相关的api

其他api：除上述api以外比较零散的api

### project相关api详解

每个project都会对应一个build.gradle，每一个groovy脚本都会被编译器编译成script字节码，每个build.gradle都会被gradle编译成project字节码。我们在build.gradle中所写的所有逻辑就相当于在project类内部进行书写。

#### getAllprojects()

输出所有的的project。

this.getProjects()

def getProjects() {  
 println '--------------------------'  
 println 'Root Project'  
 println '--------------------------'  
 this.getAllprojects().eachWithIndex { Project project, int index ->  
 if(index == 0) {  
 println "Root project:'${project.name}'"  
 }else {  
 println "+---- project:'${project.name}'"  
 }  
 }  
}

执行gradle任务都会执行build.gradle相关配置，执行gradlew clean，输出结果：

> Configure project :

--------------------------  
Root Project  
--------------------------  
Root project:'gradletest'  
+---- project:'app'  
+---- project:'module1'  
+---- project:'module2'

build.gralde中的方法，代码都是在project的配置阶段执行的。

#### getSubprojects()

获取当前项目的所有子project：

this.getProjects()

def getProjects() {  
 println '--------------------------'  
 println 'Sub Project'  
 println '--------------------------'  
 this.getSubprojects().eachWithIndex { Project project, int index ->  
 println "+---- project:'${project.name}'"  
 }  
}

输出结果：

> Configure project :

--------------------------  
Sub Project  
--------------------------  
+---- project:'app'  
+---- project:'module1'  
+---- project:'module2'

上面的代码是写在根project，gradletest下面的build.gradle下面的，如果写在app的build.gradle里面输出如下：

> Configure project :app

--------------------------  
Sub Project  
--------------------------

#### gatParent()

获取父project，由于gradletest没有父project，所以测试代码写在子project下：

this.getParentProject()

def getParentProject() {  
 def name = this.getParent().name  
 println "this parent project name is: ${name}"  
  
}

输出结果：

> Configure project :app

this parent project name is: gradletest

#### getRootProject()

获取根project：

this.getRootPro()

def getRootPro() {  
 def name = this.getRootProject().name  
 println "this root project name is: ${name}"  
  
}

输出结果：

> Configure project :app

this parent project name is: gradletest

project是以树的数据结构来管理的，当在根project的build.gradle获取父project时，由于当前节点已经是根节点，所以返回父节点为null，使用getRootProject()返回的总是根节点。

通过gradle管理的工程都会有一个根工程，根工程负责管理所有的子工程。子工程可以通过getParent()获取父工程节点，父工程可以通过getSubProject()获取所有的子工程节点。

#### project()

可以在父工程中完成子工程的所有配置，传入的参数时相对于根工程的，传入app时表示直接从根工程下面查找该名称。

project('app') { Project project ->

apply plugin : 'com.android.application'  
 group 'com.test'  
 version '1.0.0'  
 dependencies {  
  
 }  
  
 android {  
  
 }  
}

#### allprojects()

配置当前节点工程和其subproject的所有project，在根build.gradle配置下面代码：

// 配置当前节点工程和其subproject的所有project

allprojects {  
 group 'com.test'  
 version '1.0.0-release'  
}  
println project('module1').group

使用gradlew clean命令，输出结果如下：

> Configure project :

com.test

#### subprojects()

不包括当前节点工程，只包括它的subproject,下面的判断不起作用，不知道为什么

// 不包括当前节点工程，只包括它的subproject

subprojects { Project project ->  
 if(project.plugins.hasPlugin('com.android.library')) {  
 apply from : './publishToMavent.gradle.'  
 }  
}

### 属性相关api

可以打开Project.java查看Project的相关属性，gradle 4.6的路径是：

.gradle\wrapper\dists\gradle-4.10.1-all\455itskqi2qtf0v2sja68alqd\gradle-4.10.1\src\core-api\org\gradle\api

Project.java开始为我们定义了一系列的属性。

比如：

*/\*\*  
 \* The default project build file name.  
 \*/*String DEFAULT\_BUILD\_FILE = "build.gradle";  
  
*/\*\*  
 \* The hierarchy separator for project and task path names.  
 \*/*String PATH\_SEPARATOR = ":";  
  
*/\*\*  
 \* The default build directory name.  
 \*/*String DEFAULT\_BUILD\_DIR\_NAME = "build";  
  
String GRADLE\_PROPERTIES = "gradle.properties";  
  
String SYSTEM\_PROP\_PREFIX = "systemProp";  
  
String DEFAULT\_VERSION = "unspecified";  
  
String DEFAULT\_STATUS = "release";

#### ext定义扩展属性

在一个build.gradle里，可以通过变量定义来实现相关字符串的替换，比如：

apply plugin: 'com.android.application'

def mCompileSdkVersion = 28  
def libAndroidAppcompat = 'com.android.support:appcompat-v7:28.0.0'  
android {  
 compileSdkVersion mCompileSdkVersion  
}  
  
dependencies {  
 implementation libAndroidAppcompat  
}

gradle支持扩展属性，通过扩展属性也可以达到上述的目的：

apply plugin: 'com.android.application'

// 扩展属性  
ext {  
 compileSdkVersion = 28  
 libAndroidAppcompat = 'com.android.support:appcompat-v7:28.0.0'  
}  
  
android {  
 compileSdkVersion this.compileSdkVersion  
}  
  
dependencies {  
 implementation this.libAndroidAppcompat  
}

为每个子工程配置扩展属性，在根build.gradle中添加如下代码，子工程相关配置删除。

subprojects {

ext {  
 compileSdkVersion = 28  
 libAndroidAppcompat = 'com.android.support:appcompat-v7:28.0.0'  
 }  
}

上述方法相当于每个子project定义了扩展属性，如果想定义一份，需要把根build.gradle改成如下：

ext {

compileSdkVersion = 28  
 libAndroidAppcompat = 'com.android.support:appcompat-v7:28.0.0'  
}

子工程build.gradle改成如下：

apply plugin: 'com.android.application'

android {  
 compileSdkVersion this.rootProject.compileSdkVersion  
}  
  
dependencies {  
 implementation this.rootProject.libAndroidAppcompat  
}

如果把rootProject去掉，也是可以的

apply plugin: 'com.android.application'

android {  
 compileSdkVersion this.compileSdkVersion  
}  
  
dependencies {  
 implementation this.libAndroidAppcompat  
}

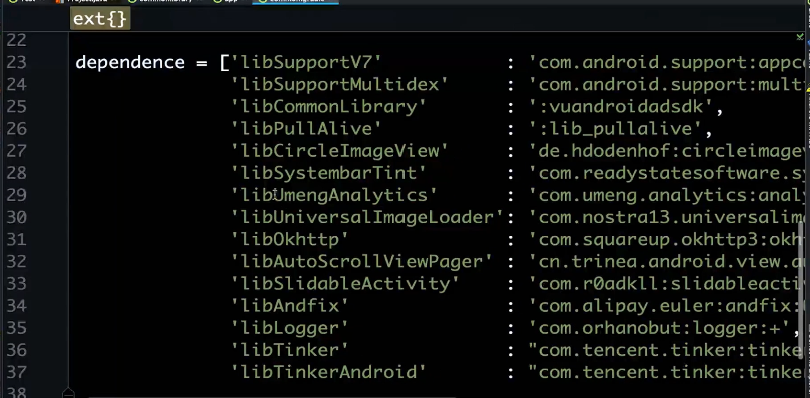
gradle规定，父project所有的属性都会被根project继承，所以可以直接在子project使用父project的属性。

可以把所有的扩展属性定义到一个独立的gradle文件中，在需要使用的build.gradle文件中使用apply from进行引入。

apply from : file('common.gradle')

可以用一个独立文件定义好下面的属性，然后在需要使用的地方进行使用。





#### gradle.properties定义属性

扩展属性也可以定义在gradle.properties中，在这个文件中只能定义key-value形式的扩展属性，而不能使用类似Map方式的定义，在使用上有一定的限制。

下面通过在gradle.properties定义开关控制一个模块是否引入项目中，在gradle.properties中定义：

isLoadTest = false

setting.gradle：

include ':app', ':module1', ':module2'

if(hasProperty('isLoadTest') ? isLoadTest.toBoolean() : false) {  
 include 'test'  
}

所以gradle.properties也可以定义扩展属性，在使用的时候转换成对应的类型。

例如在gradle.properties定义：

mCompileSdkVersion = 28

使用的时候：

compileSdkVersion mCompileSdkVersion.toInteger()

在gradle.properties定义的属性不能和build.gradle已经存在的方法同名，否则编译的时候不报错，但是在使用时会提示属性找不到。

### 文件相关api

#### 文件相关API

#### 路径获取相关api

// 根工程文件路径api

println "the root file path is: " + getRootDir().absolutePath  
// 获取build文件路径  
println "the build file path is: " + getBuildDir().absolutePath  
// 获取当前工程路径  
println "the project file path is: " + getProjectDir().absolutePath

输出结果：

> Configure project :

the root file path is: E:\studio\gradletest  
the build file path is: E:\studio\gradletest\build  
the project file path is: E:\studio\gradletest

#### 文件操作相关api

##### 文件定位

file()方法

在当前project下查找对应的文件，比如当前gradle文件是跟工程，使用file方法，就会从根工程下面查找对应的文件。如果查找到了，返回对应的file对象，如果查找不到，报异常。

输出common.gradle的内容，根工程build.gradle添加：

println getContent('common.gradle')

def getContent(String path) {  
 try {  
 def file = file(path)  
 return file.text  
 }catch(GradleException e){  
 println 'file not found'  
 }  
}

输出结果：

ext {

// compileSdkVersion = 28  
 libAndroidAppcompat = 'com.android.support:appcompat-v7:28.0.0'  
}

file()参数传入的路径是相对于当前工程的，和使用new file()的区别是new File()对于传入的参数不会自动转成相对于当前工程的相对路径，而file会自动转换成相对当前工程的相对路径。

files()方法

files()定义一个或多个文件，返回一个collection，我们可以通过collection进行遍历处理。

##### 文件拷贝

把app工程下的build.gradle拷贝到根工程的build目录下，在app的build.gradle添加代码：

copy {

from file('build.gradle')  
 into getRootProject().getBuildDir()  
}

也可以拷贝整个文件夹：

copy {

from file('build/outputs/apk/')  
 into getRootProject().getBuildDir().path + '/apk/'  
}

也可以在拷贝的时候使用exclude进行排除，使用rename进行重命名。

##### 文件夹拷贝

文件夹拷贝可以使用exclude排除一些不想拷贝的文件，可以使用rename进行重名文件。

// 文件夹拷贝  
copy **{**

println("===================> 文件夹拷贝开始")  
 from file('build/outputs/apk/')  
 into getRootProject().getBuildDir().path + 'apk'  
 exclude **{** // 某些文件不需要拷贝可以在这里排除  
  
 **}** rename **{}** // 指定文件重命名  
  
 println("===================> 文件夹拷贝结束")  
**}**

##### 文件树遍历

文件树遍历使用fileTree实现：

fileTree('build/outputs/apk/'){ FileTree fileTree ->

fileTree.visit { FileTreeElement element ->  
 println 'the file name is: ' + element.file.name  
 copy {  
 from element.file  
 into getRootProject().getBuildDir().path + '/test/'  
 }  
  
 }  
}

上述提供的文件操作API只是针对在本工程进行文件操作，如果需要跨工程进行文件操作，需要使用groovy提供的api。

## 依赖相关api

### buildscript

buildscript { ScriptHandler scriptHandler ->

// 配置工程的仓库地址  
 scriptHandler.repositories { RepositoryHandler repositoryHandler ->  
 repositoryHandler.jcenter()  
 repositoryHandler.mavenCentral()  
 repositoryHandler.mavenLocal()  
 repositoryHandler.ivy {}  
 repositoryHandler.maven {  
 name 'personal'  
 url 'http://localhost:8081:/nexus/repositories/'  
 credentials {  
 username = 'admin'  
 password = 'admin123'  
 }  
 }  
 }  
  
 // 配置工程的"插件"地址  
 scriptHandler.dependencies {  
 classpath 'com.android.tools.build:gradle:2.2.2'  
 }  
}

buildscript里面的dependencies和子工程下的dependencies是不一样的，buildscript里面的dependencies是表示gradle本身对第三方的依赖，子工程下的dependencies是应用程序对第三方的依赖。

为什么闭包可以去掉参数里面的方法仍然可以被调用，因为闭包的delegate就是去掉的参数，

Groovy的闭包有owner，this，delegate这3个属性，只要这三个属性有的方法，都可以调用，区别是他们有一个执行顺序不一样。

子工程的dependencies可以添加依赖类型:

// 本地

implementation fileTree(include: ['\*.jar'], dir: 'libs')  
// 远程jar包  
implementation 'com.android.support:appcompat-v7:28.0.0'

// 依赖library工程  
implementation project(':module1')

依赖冲突的解决，使用exclude进行排除，如在引入包的后面添加

{

exclude module: 'support-v4' // 排除依赖  
 exclude group: 'com.android.support'  
 transitive false // 禁止传递依赖  
}

### compile()

使用compile()引入的库，在打包的时候类和资源都会打包在apk中。

### provided()

占位编译，在编译的时候gradle会读取引入第三方库的类和资源，但是在打包的时候是不会将这个库的类和资源打入到输出工程中的。在以下2中场景下使用provided()

1. 只使用第三方库生成的类，第三方库只在编译的时候起作用就可以了
2. 当前工程如果是类库，引入了主工程相同版本的库

## 外部命令执行

定义一个task，把apk拷贝到固定目录：

task(name: 'apkcopy') {

doLast {  
 def sourcePath = this.buildDir.path + '/outputs/apk'  
 def desationPath = '/users/Downloads/'  
 def command = "mv -f ${sourcePath} ${desationPath}"  
 exec {  
 try {  
 executable 'bash'  
 args '-c', command

println "the command is execute success."  
 } catch (GradleException e){  
 println "the command is execute failed."  
 }  
 }  
 }  
}

# Task

## Task定义及配置

由于工程已经引入了很多第三方插件，可以使用gradlew task查看当前工程有哪些task

E:\studio\Groovy>gradlew task

### 定义

直接使用task函数进行定义：

task helloTask {

println 'I am helloTask'  
}

通过TaskContainer创建Task

this.tasks.create(name: 'helloTask2') {

println 'I am helloTask2'  
}

gradle是通过TaskContainer管理当前Project中所有的Task。

执行task

E:\studio\Groovy>gradlew helloTask

结果：

I am helloTask  
I am helloTask2

上面2中创建方式没有任何区别。

### 配置：

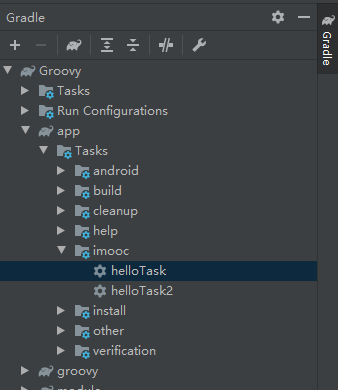
为task设置组名和描述信息：

task helloTask (group: 'imooc', description: 'task study') {

println 'I am helloTask'  
}  
  
this.tasks.create(name: 'helloTask2') {  
 setGroup('imooc')  
 setDescription('task study')  
 println 'I am helloTask2'  
}

上述2中创建task的方法效果都是一样的，最终都会把创建的task添加到task TaskContainer中。

group分组后，gradle会把相同group的task放一起，方便管理，可以在右侧Gradle面板中看到imooc分组，刚才创建的task就在这个分组下。如果不添加分组，就会被gradle放到other分组。



description是普通描述信息，说明该task的作用

## Task的执行详解

定义中的例子输出都是在task的配置阶段进行输出的，所以上述例子会输出task中的2个task中的打印语句。任何task的执行整个project中所有的配置代码都会被这行。使用task的doFirst和doLast会运行在gradle的执行周期，下面看看如何让task里面的代码在执行阶段运行。

task helloTask (group: 'imooc', description: 'task study') {

println 'I am helloTask'  
 doFirst {  
 println 'the task group is: ' + group  
 }  
 // 可以多次调用  
 doFirst {}  
}  
  
helloTask.doFirst {  
 println 'the task description is: '+ description  
}

在task内部可以多次调用doFirst，也可以在task外部进行调用doFirst，执行时先执行外部的doFirst，再执行task内部定义的。

E:\studio\gradletest>gradlew helloTask

Starting a Gradle Daemon, 1 incompatible Daemon could not be reused, use --status for details  
  
> Configure project :app  
I am helloTask  
  
> Task :app:helloTask  
the task description is: task study  
the task group is: imooc

doFirst和doLast的区别，gradle已经为我们默认实现了许多task，可以通过doFirst可以为已经有的task之前添加相应的逻辑，通过doLast可以为已经有的task之后添加相应的逻辑。

下面实现一个计算build执行时长的功能：

// 在这个方法执行的时候，project所有的task都已经被有序地创建好了

// 如果直接在脚本找，有些task没有被创建完毕，会出现task找不到报错  
this.afterEvaluate {  
 def preBuildTask = project.tasks.getByName('preBuild')  
 preBuildTask.doFirst {  
 startBuildTime = System.*currentTimeMillis*()  
 println 'the start time is: ' + startBuildTime  
 }  
 def buildTask = project.tasks.getByName('build')  
 buildTask.doLast {  
 endBuildTime = System.*currentTimeMillis*()  
 println "the build time is: ${endBuildTime - startBuildTime}"  
 }  
}

执行构建

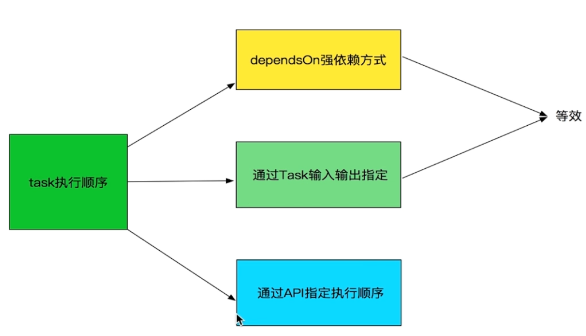
gradlew build

输出结果

the build time is: 34603

## Task的依赖和执行顺序

决定task执行顺序主要有3中方式：



1. dependsOn强依赖方式
2. 通过Task输入输出指定
3. 通过API指定实行顺序

其中方式1和方式2是等效的。

task依赖示例：

*/\*\**

*\* task 依赖  
 \*/*task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}  
task taskZ(dependsOn : [taskX, taskY]) {  
 doLast {  
 println 'taskZ'  
 }  
}

使用dependsOn指定依赖，如果多个依赖，使用列表进行定义，输出

E:\studio\gradletest>gradlew taskZ

> Task :app:taskX  
taskX  
  
> Task :app:taskY  
taskY  
  
> Task :app:taskZ  
taskZ

也可以使用下面方式指定依赖：

taskZ.dependsOn(taskX, taskY)

被依赖的task如果没有依赖关系，他们的执行的时机是随机的。

使用<<的作用完全等于doLast方法。

task动态依赖：

task lib1 << {

println 'lib1'  
}  
  
task lib2 << {  
 println 'lib2'  
}  
  
task noLib << {  
 println 'noLib'  
}  
  
task taskZ {  
 dependsOn this.tasks.findAll { task ->  
 println 'dependsOn: ' + task.name  
 task.name.startsWith('lib')  
 }  
 doLast {  
 println 'taskZ'  
 }  
}

taskZ需要放到依赖task的后面，如果放在前面，在执行dependsOn会发现tasks中没有在taskZ后面task，导致依赖失败。

实际运用，解析xml并输出到对应的文件：

xml的内容：

<releases>

<release>  
 <versionCode>100</versionCode>  
 <versionName>1.0.0</versionName>  
 <versionInfo>App的第1个版本，上线了一些最基础核心的功能.</versionInfo>  
 </release>  
   
   
 <release>  
 <versionCode>110</versionCode>  
 <versionName>1.1.0</versionName>  
 <versionInfo>App的第2个版本，上线了一些最基础核心的功能.</versionInfo>  
 </release>  
   
   
 <release>  
 <versionCode>1</versionCode>  
 <versionName>1.2.0</versionName>  
 <versionInfo>第8个版本。。。</versionInfo>  
 </release>  
</releases>

build.gradle

task handleReleaseFile {

def srcFile = file('releases.xml')  
 def destDir = new File(this.buildDir, 'generated/release/')  
 doLast {  
 println '开始解析对应的xml文件...'  
 destDir.mkdir()  
 def releases = new XmlParser().parse(srcFile)  
 releases.release.each{releaseNode ->  
 // 解析每个release节点的内容  
 def name = releaseNode.versionName.text()  
 def versionCode = releaseNode.versionCode.text()  
 def versionInfo = releaseNode.versionInfo.text()  
 // 创建文件，并写入节点数据  
 def destFile = new File(destDir, "release-${name}.txt")  
 destFile.withWriter { writer ->  
 writer.write("${name} -> ${versionCode} -> ${versionInfo}")  
  
 }  
 }  
 }  
}  
  
task handleReleaseFileTest (dependsOn : handleReleaseFile){  
 def dir = fileTree( 'build/generated/release/')  
 doLast {  
 dir.each {  
 println 'the file name is: ' + it  
 }  
 println '输出完成...'  
 }  
}

在控制台输入命令运行task：

gradlew handleReleaseFileTest

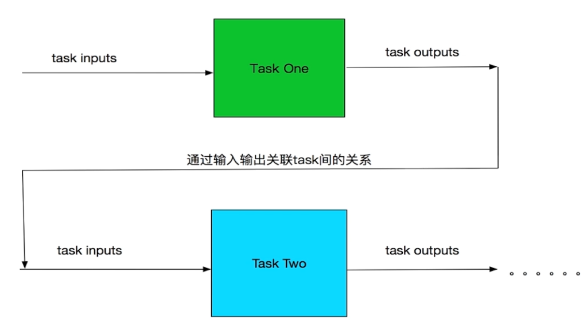
输出结果：

> Task :app:handleReleaseFile

开始解析对应的xml文件...  
  
> Task :app:handleReleaseFileTest  
the file name is: E:\studio\gradletest\app\build\generated\release\release-1.0.0.txt  
the file name is: E:\studio\gradletest\app\build\generated\release\release-1.1.0.txt  
the file name is: E:\studio\gradletest\app\build\generated\release\release-1.2.0.txt  
输出完成...

## Task输入输出

先看一张图：



上图Task One和Task Two分别代表2个Task，inputs和outputs是Task的2个属性，Task One的输出是Task Two的输入。

TaskInputs代表Task的输入类，TaskInputs接收的输入参数类型包括：files，file，dir和key-value类型的属性，value可以是任何对象类型。也就是说Task输入可以是任意对象，文件，多个文件和文件夹。

TaskOutputs表示Task的输出类，输出类型包括：files，file，dir，也就是说TaskOutputs的输出是文件类型。

gradle规定，如果一个task同时依赖于另外两个task，并且另外两个task的其中一个task输出作为另外一个task的输入，那么会先执行生产者的task。

下面的代码功能：根据版本信息生成release.xml文件

*/\*\**

*\* task的输入输出详解  
 \*/*import groovy.xml.MarkupBuilder  
  
ext {  
 versionName = '1.1.0'  
 versionCode = '110'  
 versionInfo = 'App的第2个版本，上线了一些基础的核心功能'  
 destFile = file('releases.xml')  
 if(destFile != null && !destFile.exists()) {  
 destFile.createNewFile()  
 }  
}  
  
task readTask {  
 inputs.file destFile  
 doLast {  
 def file = inputs.files.singleFile  
 println file.text  
 }  
}  
  
task writeTask {  
 // 为Task指定输入  
 inputs.property('versionCode', this.versionCode)  
 inputs.property('versionName', this.versionName)  
 inputs.property('versionInfo', this.versionInfo)  
  
 outputs.file destFile  
  
 doLast {  
 def data = inputs.getProperties()  
 File file = outputs.getFiles().getSingleFile()  
  
 // 将Map转化为实体对象  
 def versionMsg = new VersionMsg(data)  
 def sw = new StringWriter()  
 def xmlBuilder = new MarkupBuilder(sw)  
 if(file.text != null && file.text.size() <= 0) {  
 // 文件中没有内容  
 xmlBuilder.releases {  
 release {  
 versionCode(versionMsg.versionCode)  
 versionName(versionMsg.versionName)  
 versionInfo(versionMsg.versionInfo)  
 }  
 }  
 file.withWriter{writer -> writer.append(sw.toString())}  
 } else {  
 // 已有版本信息  
 xmlBuilder.release {  
 versionCode(versionMsg.versionCode)  
 versionName(versionMsg.versionName)  
 versionInfo(versionMsg.versionInfo)  
 }  
 // 将生成的xml数据插入到根节点之前  
 def lines = file.readLines()  
 def lengths = lines.size() - 1  
 file.withWriter { writer ->  
 lines.eachWithIndex { String line, int index ->  
 if(index != lengths) {  
 writer.append(line + '\r\n')  
 } else if(index == lengths) {  
 writer.append('\r\r\n' + sw.toString() + '\r\n')  
 writer.append(lines.get(lengths))  
 }  
 }  
 }  
 }  
 }  
}  
  
class VersionMsg {  
 String versionCode  
 String versionName  
 String versionInfo  
}  
  
task taskTest {  
 dependsOn readTask, writeTask  
 doLast {  
 println '输入输出任务结束'  
 }  
}

上面的readTask和writeTask看起来没有依赖关系，实际上是通过输入输出的关系确定了依赖，readTask的输入是destFile，writeTask的输出是destTask，gradle规定输出属性对应的task会被首先执行，所以会优先执行writeTask。

这个例子输出的release1.xml打开之后看到的是乱码，暂时还不知道怎么处理。

## 挂接到构建生命周期

### task执行顺序

task执行顺序的指定除了使用dependsOn强依赖方式和通过Task输入输出指定，还可以通过API指定实行顺序。

mustRunAfter它可以指定一个，也可以指定多个，用法完全一样。

*/\*\**

*\* 执行顺序的指定  
 \*/*task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
  
task taskY {  
 mustRunAfter taskX  
 doLast {  
 println 'taskY'  
 }  
}  
  
task taskZ {  
 mustRunAfter taskY  
 doLast {  
 println 'taskZ'  
 }  
}

控制台输入顺序错乱的task：

gradlew taskY taskZ taskX

输出：

E:\studio\gradletest>gradlew taskY taskZ taskX

> Task :app:taskX  
taskX  
  
> Task :app:taskY  
taskY  
  
> Task :app:taskZ  
taskZ

shouldRunAfter和mustRunAfter用法完全一样，唯一区别是mustRunAfter是强制task必须在另外的task之后执行，而shouldRunAfter是非强制性的。

### 挂接到构建生命周期

this.project.afterEvaluate { project ->

def buildTask = project.tasks.getByName('build')  
 if(buildTask == null) {  
 throw GradleException('the build task is not found')  
 }  
 buildTask.doLast {  
 writeTask.execute()  
 }  
}

### 挂接到构建生命周期的中间

指定运行在某些task之后，某个task之前。通过查看tinker的源码实现。

## task类型

官方文档：https://docs.gradle.org/current/dsl/

## Task实战

# Gradle其他模块

## Settings类

Setting类是通过setting.gradle中的配置文件或者配置代码来完成初始化的，Setting类和setting.gradle的核心作用是决定工程中哪些子工程需要被gradle处理。Settring类占用了gradle的初始化流程，gradle的初始化阶段完全是在执行setting.gradle中的内容。

gradle.properties

isLoadTest = true

settting.gradle

if(hasProperty('isLoadTest') ? isLoadTest.toBoolean() : false) {  
 include ':module'  
}

## SourceSet类

gradle为什么会从src/main/java目录读取源码，然后生成字节码文件，其实就是从SourceSet这个类的配置中拿到的。SourceSet类决定了所有的代码，资源，第三方库等要存放的位置。

gradle有个最大的特点是约定大于配置，只要没有做出修改，默认按照约定的配置。gradle默认是从java文件夹下获取源码，从res文件夹下获取资源进行编译。

SourceSet类可以修改位置的文件夹类型包括：resources，java，manifest，res，asset，jni，jniLibs。

在app中的build.gradle进行配置SourceSets

把so存放目录执行libs文件夹：

sourceSets {

*main* {  
 jniLibs.srcDirs = ['libs'] // 修改so库存放位置  
 }  
}

资源分模块

配置多个不同的res资源目录，然后把不同业务的资源放到对应的资源目录中：

sourceSets {

*main* {  
 res.srcDirs = ['src/main/res', 'src/main/res-ad', 'src/main/res-player']  
 }  
}

把sourceSet放在android闭包外面：

this.android.sourceSets {

*main* {  
 res.srcDirs = ['src/main/res', 'src/main/res-ad', 'src/main/res-player']  
 }  
}

## Plugin及自定义Plugin

### gradle插件目录作用

自定义gradle插件工程，各文件夹的作用

buildSrc：这是插件工程默认的文件夹，gradle会自动把这个文件夹下面的内容编译为一个插件。

src/main/groovy：开发插件创建的groovy类

src/main/resources：

build.gradle的内容：

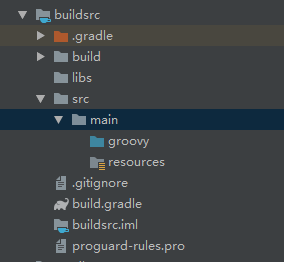
apply plugin: 'groovy'

sourceSets {  
 *main* {  
 groovy {  
 srcDir 'src/main/groovy'  
 }  
 resources {  
 srcDir 'src/main/resources'  
 }  
 }  
}

### 与android工程区别

1. groovy文件夹
2. resources文件夹
3. build.gradle里的内容

gradle插件工程目录结构:



### 自定义插件步骤

1. 在groovy目录下创建包名

com.imooc.gradle.study

1. 在该包名下创建一个自定义Plugin

package com.imooc.gradle.study

import org.gradle.api.Plugin

import org.gradle.api.Project  
  
*/\*\*  
 \* 自定义插件  
 \*/*class GradleStudyPlugin implements Plugin<Project>{  
  
 @Override  
 void apply(Project project) {  
 println 'Hello Project...' + project.name  
 }  
}

1. 使用该插件

在resources文件夹下创建META-INF目录，然后在META-INF目录下创建gradle-plugins目录

创建com.imooc.gradle.study. properties文件，文件内容：

implementation-class=com.imooc.gradle.study.GradleStudyPlugin

1. 引入插件

在build.gradle引入该插件

apply plugin: 'com.imooc.gradle.study'

同步一下，看下控制台输出：

> Configure project :app  
============> Hello Plugin ...

5.遇到问题

Plugin with id not found

原因是创建META-INF时，直接把目录创建成了META-INF. gradle-plugins。而实际上应该创建2级目录，META-INF/gradle-plugins,原来直接创建成了一个目录

### 定义扩展属性

1. 首先定义一个通信类ReleaseInfoExtension.groovy:

package com.imooc.gradle.study

*/\*\*  
 \* 与自定义PLugin进行参数传递  
 \*/*class ReleaseInfoExtension {  
  
 String versionCode  
 String versionName  
 String versionInfo  
 String fileName  
  
 ReleaseInfoExtension() {  
  
 }  
  
 @Override  
 String toString() {  
 """| versionCode = ${versionCode}  
 | versionName = ${versionName}  
 | versionInfo = ${versionInfo}  
 | fileName = ${fileName}  
 """.stripMargin()  
 }  
}

1. 在自定义插件apply方法进行创建

//创建扩展属性

project.extensions.create('imoocReleaseInfo',  
 ReleaseInfoExtension)

在build.gradle中使用扩展属性：

*/\*\**

*\* 为自定义插件传递参数  
 \*/*imoocReleaseInfo {  
 versionCode = rootProject.ext.android.versionCode  
 versionName = rootProject.ext.android.versionName  
 versionInfo = '第8个版本。。。'  
 fileName = 'releases.xml'  
}

### 自定义task

默认创建的Task都是集成DefaultTask。

下面介绍自定义Task的一些知识点：

@TaskAction：被这个注解修饰的方法会被执行于Task的执行阶段。

创建ReleaseInofTask类并继承于DefaultTask：

package com.imooc.gradle.study

import groovy.xml.MarkupBuilder  
import org.gradle.api.DefaultTask  
import org.gradle.api.tasks.TaskAction  
  
*/\*\*  
 \* 自定义Task,实现维护版本信息功能  
 \*/*class ReleaseInfoTask extends DefaultTask {  
 ReleaseInfoTask() {  
 group = 'imooc'  
 description = 'update the release info'  
 }  
  
 */\*\*  
 \* 执行于gradle执行阶段的代码  
 \*/* @TaskAction  
 void doAction() {  
 updateInfo()  
 }  
  
 //真正的将Extension类中的信息呢，写入指定文件中  
 private void updateInfo() {  
 //获取将要写入的信息  
 String versionCodeMsg = project.extensions.  
 imoocReleaseInfo.versionCode  
 String versionNameMsg = project.extensions.  
 imoocReleaseInfo.versionName  
 String versionInfoMsg = project.extensions.  
 imoocReleaseInfo.versionInfo  
 String fileName = project.extensions.  
 imoocReleaseInfo.fileName  
 def file = project.file(fileName)  
 //将实体对象写入到xml文件中  
 def sw = new StringWriter()  
 def xmlBuilder = new MarkupBuilder(sw)  
 if (file.text != null && file.text.size() <= 0) {  
 //没有内容  
 xmlBuilder.releases {  
 release {  
 versionCode(versionCodeMsg)  
 versionName(versionNameMsg)  
 versionInfo(versionInfoMsg)  
 }  
 }  
 //直接写入  
 file.withWriter { writer -> writer.append(sw.toString())  
 }  
 } else {  
 //已有其它版本内容  
 xmlBuilder.release {  
 versionCode(versionCodeMsg)  
 versionName(versionNameMsg)  
 versionInfo(versionInfoMsg)  
 }  
 //插入到最后一行前面  
 def lines = file.readLines()  
 def lengths = lines.size() - 1  
 file.withWriter { writer ->  
 lines.eachWithIndex { line, index ->  
 if (index != lengths) {  
 writer.append(line + '\r\n')  
 } else if (index == lengths) {  
 writer.append('\r\r\n' + sw.toString() + '\r\n')  
 writer.append(lines.get(lengths))  
 }  
 }  
 }  
 }  
  
 }  
}

在GradleStudyPlugin进行创建：

//创建Task

project.tasks.create('imoocReleaseInfoTask',  
 ReleaseInfoTask)

运行imoocReleaseInfoTask有2中方式：

1. 在android studio右侧的面板中找到imoocReleaseInfoTask，点击运行
2. 使用命令行gradlew imoocReleaseInfoTask

实际运行Task发现，使用右侧面板运行task生成的文件中文没有乱码，但是使用命令行生成的文件有乱码。解决方法是在运行命令的时候指定编码

gradlew imoocReleaseInfoTask -Dfile.encoding=UTF-8

### 插件供所有人使用

1. 创建一个library，在里面实现插件代码
2. 把生成的aar上传到远程仓库

## android插件对gradle扩展

android插件的属性配置可以到BaseExtension.class中进行查看

android task可以到BaseVariant.class查看，具体包名是com.android.build.gradle.api

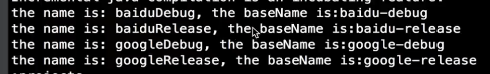
可以在下面这个文档查看gradle的一些说明：

<https://avatarqing.gitbooks.io/gradlepluginuserguidechineseverision/content/advanced_build_customization/manipulation_taskstask.html>

android变体name和baseName的区别：

this.afterEvaluate **{** this.android.applicationVariants.all **{** variant **->** def name = variant.name  
 def baseName = variant.baseName  
 println "==========> this name is $name, this baseName is $baseName"  
   
 **}  
}**

输出结果，测试例子没有添加多渠道，下面是视频老师的例子输出



修改输出apk文件名，下面的是gradle 3.0以前的操作，3.0以后的会有问题。

<https://blog.csdn.net/u014343528/article/details/79473065>

this.afterEvaluate **{** this.android.applicationVariants.all **{** variant **->** def output = variant.outputs.first()  
 def apkName = "app-$**{**variant.baseName**}**-$**{**variant.versionName**}**.apk"  
 output.outputFile = new File(output.outputFile.parent, apkName)  
 println "==========> this output name is $**{**output.outputFile.name**}**"  
 **}  
}**

## 如何迁移到gradle

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5