# 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

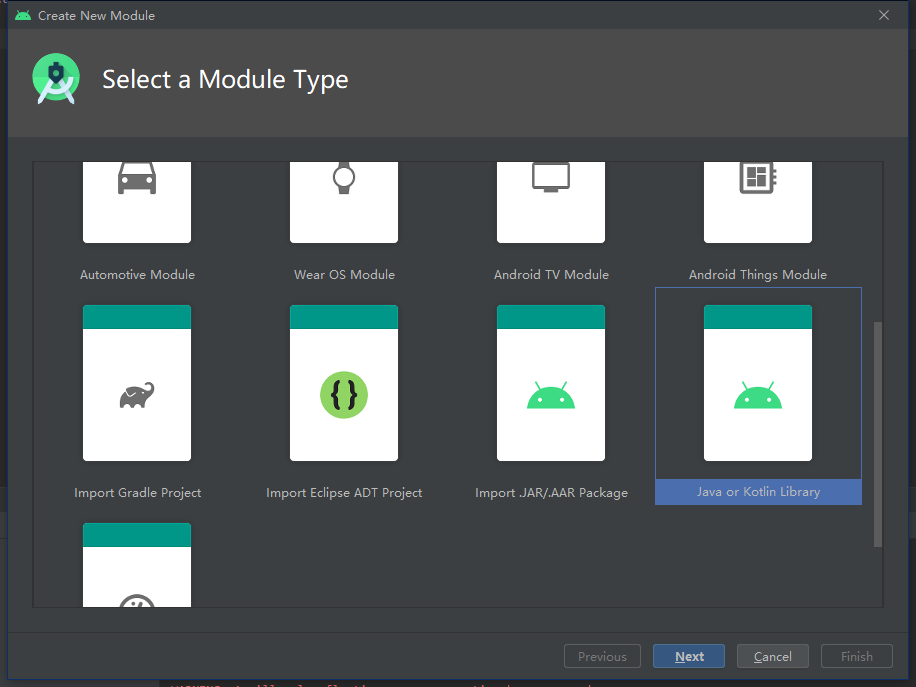
1. 既可以作为编程语言也可以作为脚本语言

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

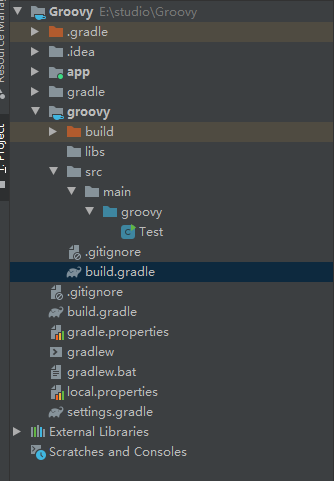
1. 新建一个android工程项目
2. 新建一个Library项目，选择Java or Kotlin Library



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"

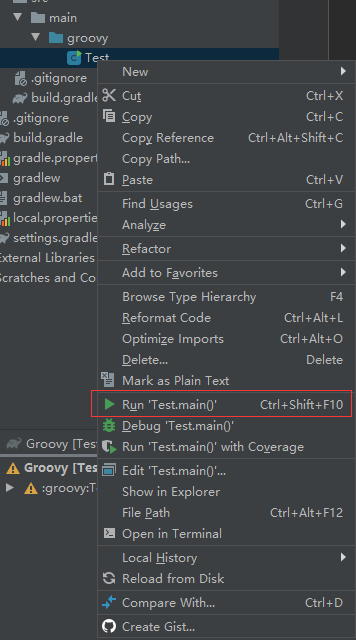
4.新建工程的main目录下创建groovy目录，删除掉原来的java目录



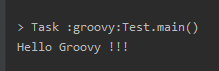
1. 在groovy目录下右键新建一个文件Test.groovy，输入如下内容

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：

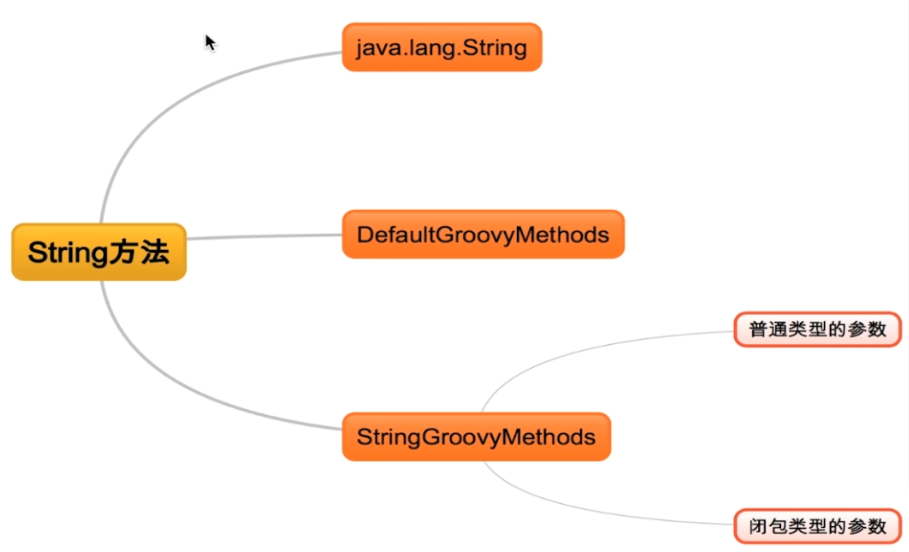
1. Java.lang.String
2. DefaultGroovyMethods

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()

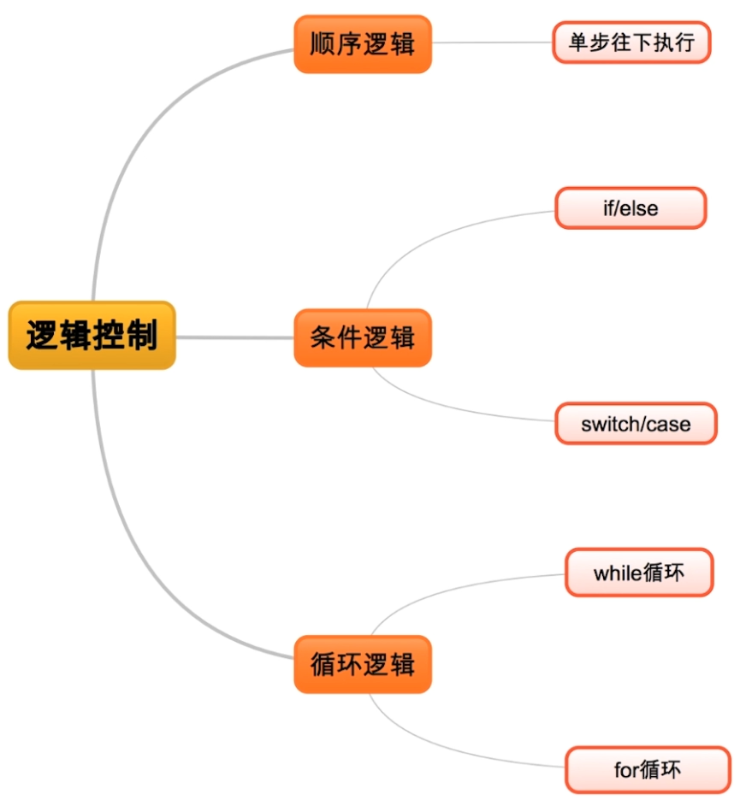
输出结果：

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

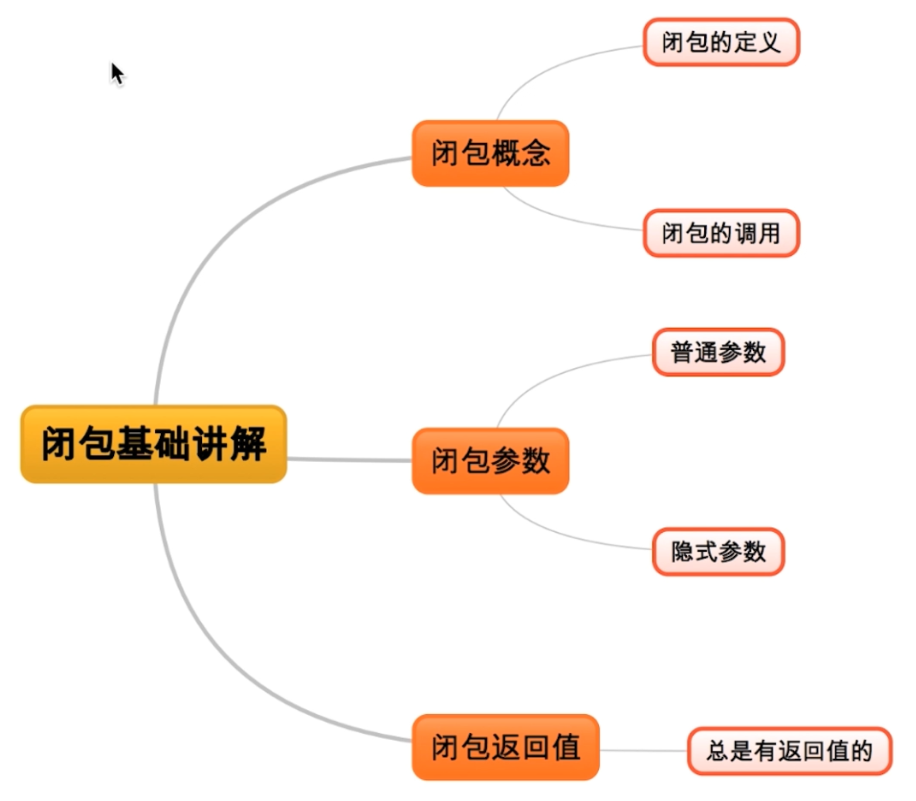
6

# 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：

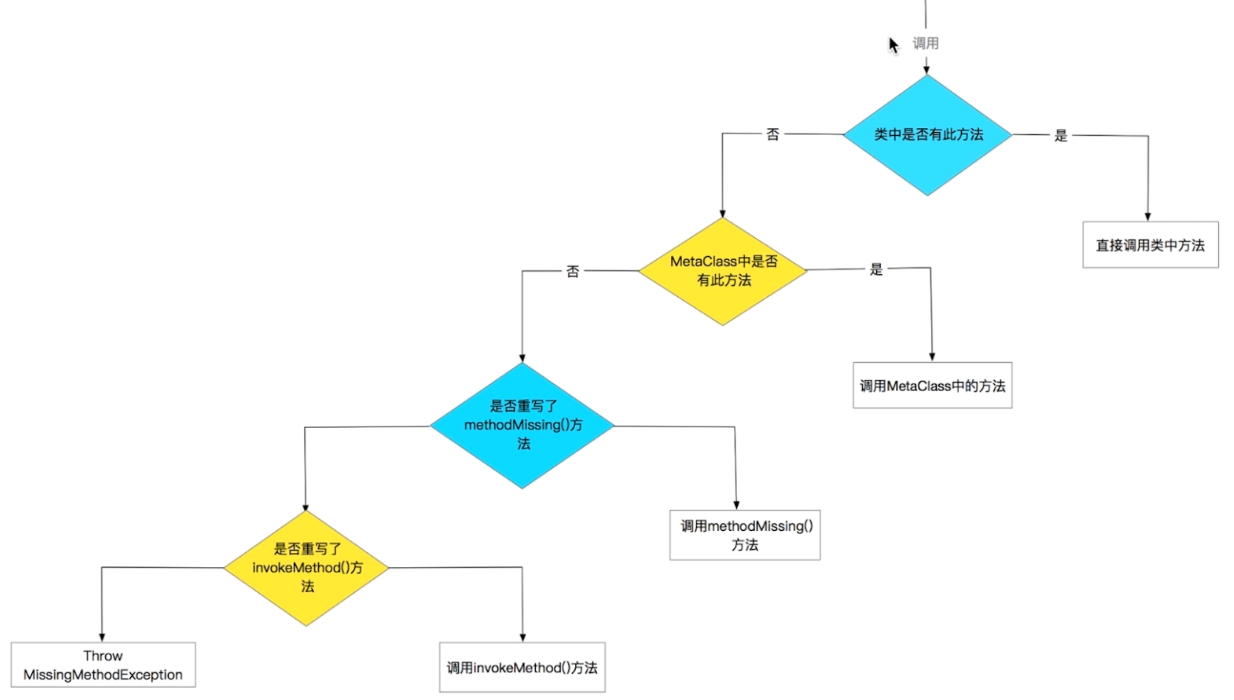
this name is android, thi age is 26

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 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。

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5