Разработка → Groovy за 15 минут – краткий обзор

JAVA*

Groovy — объектно-ориентированный язык программирования разработанный для платформы Java как альтернатива языку Java с возможностями Python, Ruby и Smalltalk.

Groovy использует Java-подобный синтаксис с динамической компиляцией в JVM байт-код и напрямую работает с другим Java кодом и библиотеками. Язык может использоваться в любом Java проекте или как скриптовый язык.

Возможности Groovy (отличающие его от Java):

— Статическая и динамическая типизация
— Встроенный синтаксис для списков, ассоциативных массивов, массивов и регулярных выражений
— Замыкания
— Перегрузка операций

[http://ru.wikipedia.org/wiki/Groovy]

Более того, почти всегда java-код — это валидный groovy-код.

Установка

Для установки нужно скачать архив с оф. сайта , распаковать его в удобное место и добавить переменую окружения GROOVY_HOME, и добавить путь до groovy/bin в PATH:

```
export GROOVY_HOME=~/path/to/groovy/
export PATH=$GROOVY_HOME/bin:$PATH
```

В IDE NetBeans 7.0 поддержка groovy идет из коробки, для IDE Eclipse существует очень хороший плагин, который можно взять здесь.

Groovy

Самое главное отличие от java: в Groovy всё является объектами. Все примитивные типы сразу же упаковываются в объекты. Т.е. «int x» на самом деле является «Integer x»

```
println 1.class
int a = 10
println a.class
```

```
class java.lang.Integer
class java.lang.Integer
```

Следует не забывать о том, что все упаковочные типы — неизменяемые, поэтому каждый раз при каком-либо вычислении будет создаваться новый объект.

Строки в Groovy

1) Java Strings — строки в одинарных кавычках
2) Groovy Strings, они же GStrings — в обычных кавычках
В строках groovy можно вставлять параметры, в обычные строки — нельзя

```
javaString = 'java'
groovyString = "${javaString}"
j = '${javaString}'
bigGroovyString = """
   ${javaString}
   ${groovyString}
   ${j}
   ${2 + 2}
```

```
"""

println bigGroovyString
```

java
java
${**javaString**}
4

К строкам применимы операции + и *

```
groovy:000> a = "a"
===> a
groovy:000> a + "123"
===> a123
groovy:000> a * 5
===> aaaaa
```

Так же к строкам применимы ++ и - — (т.к. в groovy поддерживается перегрузка операторов)

```
groovy:000> a = 'abc'
===> abc
groovy:000> a++
===> abd
groovy:000> a--
===> abc
```

В groovy есть поодержка регулярных выражений на уровне конструкций языка:

```
groovy:000> r =~ '^a$'
===> java.util.regex.Matcher[pattern=^a$ region=0,1 lastmatch=]
```

Встроенная поддержка maps + lists

Так же на уровне конструкций языка поддерживаются словари (maps) и списки:

```
groovy:000> a = [1, 3, 5]
===> [1, 3, 5]
groovy:000> b = [1: true, 0: false]
===> {1=true, 0=false}
```

Ranges

Получать доступ к элементам списков в groovy можно следующим образом:

```
groovy:000> a = "0123456789"
===> 0123456789
groovy:000> a[1..4]
===> 1234
groovy:000> a[1..-1]
===> 123456789
groovy:000> a[-1..0]
===> 9876543210
groovy:000> a[1..<9]
===> 12345678
groovy:000> a[1, 3, 5]
===> 135
groovy:000> b = 1..5
===> 1..5
groovy:000> a[b]
===> 12345
```

Range — это такой же объект, поэтому возможны конструции, подобные последней. Отрицательные индексы, как в python, возвращают элементы с конца списка.

Range можно составить из строк:

```
groovy:000> 'a'..'aa'
===> a..aa
```

Более того, range можно сделать из любого объекта, у которого есть методы next() и prev().

Циклы

Циклы в groovy точно такие же, как и в java, плюс к ним добавляется еще один «foreach»:

```
for (i in 0..9) {
    print i
}

for (int i = 0; i < 9; ++i) {
    print i
}

for (Integer i : 0..9) {
    print i
}
```

Функции

```
def functionA(argA) {
    print ArgA
}

int functionB(int argB) {
    print argB
    return argB
}

String fuctionC() {
    "Hello World"
}
```

Ключевое слово return указывать не обязательно — по умолчанию будет возвращено значение последней упомянутой переменной в фукции.

Closures

Closure — это анонимная функция

```
def cl = {a, b ->
    println a
    println b
}

cl(1, 2)
```

У многих объектов есть методы, в качестве параметров которым передаются closure:

```
1.upto 10, {
    print it
}

10.times {
    print it
}
```

Доступно большое количество методов для обработки последовательностей, к которым можно применять замыкания:

```
'qwerty'.each {
    print it
}

('a'..'z').each {
    print it
}

('a'..'z').findAll { el -> // = filter
    el in ['e', 'y', 'u', 'i', 'o', 'a']
}.each {
    print it + ' '
}
```

```
(0..10).collect { el -> // = map
   el * 10
}.each {
   print it + ' '
}

def sum = (0..10).inject(0) { prev, elem -> // = reduce
   return prev + elem
}
```

В closure так же не обязательно использовать ключевое слово return. Если явно не задано имя параметру, то по умолчанию используется it.

Так как closure является объектом, то ничего не мешает возвращать его из другого closure, и таким образом создавать функции высших порядков:

```
def cloA = {param ->
   def cloB = {
      return param * 10
   }
}

def b = cloA(10)
println b(10)
```

Файлы

Для директорий есть функции eachFile и eachFileRecursive:

```
new File('.').eachFile {
   println it
}
```

```
./.project
./src
./.settings
./.classpath
./bin
```

Для обработки текстовых файлов — функция eachLine:

```
new File('textfile.txt').eachLine {
   println it
}
```

Писать в файлы так же очень удобно:

```
def pw = new File('textfile.txt').newPrintWriter()
pw.println("new line")
```

Классы

```
class Account {
   String name
   BigDecimal value
}

// конструктор по умолчанию добавляется автоматически
// такой конструктор - синтаксический сахар для
// a = new Account()
// a.setName("Account #1")
// a.setValue(new BigDecimal(10))
a = new Account(name : "Account #1", value : new BigDecimal(10))

// геттеры и сеттеры генерируются автоматически
def name = a.getName()
a.setName("Account #2")
println "${a.name}"

class Person {
   def first
   def last
```

```
    // явно задаем сеттер
    void setFirst(first) {
        println "${this.first} is becoming ${first}"
        this.first = first
    }
}

p = new Person(first: "A", last: "G")
// если обращаться к полю, то будет использоваться сеттер
p.first = "C"

println "${p.first} ${p.last}"

// наследование как в java
class ExtendedAccount extends Account {
    def debt

    // задаем конструктор
    ExtendedAccount(name, value, debt) {
        setName(name)
        setValue(value)
        setDebt(debt)
    }

    def String toString() {
        "${name} ${value} ${debt}"
    }
}

// тут будет ошибка "Could not find matching constructor for: ExtendedAccount()"
//e = new ExtendedAccount()

println new ExtendedAccount("A", new BigDecimal(10), 1)
```

Неизменяемые классы задаются с помощью аннотации Immutable:

```
@Immutable
class ImmutableClass {
    String a
    Integer b
}

def ic = new ImmutableClass(a : "a", b : 1)
```

При использовании этой аннотации нужно явно указывать, какого типа данных поле.

Операторы

"?:" Elvis operator
```
def b = a ?: "b"
```

Проверяет переменную a, и если в ней null или false, то берет указанное следом значение. Иначе берется значение переменной a.

"?." Safe navigation
Используется для избежания ошибки NullPointerException
```
def user = Users.get("a")
def posts = user?.posts
println posts
```

Вернет null, если если в user содержится null вместо того, чтобы бросать NullPointerException.

"*." Spread operator
Применяет указанный метод для всех элементов какой-либо коллекции.
Эквивалент следующему:
```
parent*.action == parent.collect {ch -> child?.action}
```

Пример использования:

```
def sizes = ['string', 'long string']*.size()
println sizes

[6, 11]
```

Так же монжо использовать для составления списков и словарей:

```
def x = [2, 3]
def y = [0, 1, *x, 4]
println y

def a = [3 : 'c', 4 : 'd']
def b = [1 : 'a', 2: 'b', * : a, 5 : 'e']
println b

[0, 1, 2, 3, 4]
[1:a, 2:b, 3:c, 4:d, 5:e]
```

В Groovy можно перегружать операторы +, -, * и т.п. Для этого нужно определить соотвествующий метод для класса. Например, для перегрузки оператора ++ нужно переопределить метод next():

```
class RandomVal {
   // для этого поля не будут сгенерированы сеттеры и геттеры
   private def value
   private Random randomGen = new Random()

   def next() {
      this.value = randomGen.nextInt()
   }

   RandomVal() {
      this.value = randomGen.nextInt()
   }

   def String toString() {
      "${this.value}"
   }
}

def r = new RandomVal()
println(r)
r++
println(r)
```

Оператор "==" уже перегружен для всех объектов — и вызывает метод «isEquals()». Полный список методов, которые нужно переопределить для перегрузки операторов, доступен здесь: http://groovy.codehaus.org/Operator+Overloading.

SQL

SQL запросы обрабатываются очень просто:

```
import groovy.sql.Sql

def final ADDRESS = "jdbc:jtds:sqlserver://serverName/dbName"
def final USERNAME = "username"
def final PASSWD = "password"
def final DRIVER = "net.sourceforge.jtds.jdbc.Driver"
sql = Sql.newInstance(ADDRESS, USERNAME, PASSWD, DRIVER)

sql.eachRow("select * from tableName") { el ->
   println "${el.id} -- ${el.firstName}"
}

def firstName = "A"
def lastName = "G"
sql.execute("insert into tableName (firstName, lastName) " +
   "values (${firstName}, ${lastName})")
```

```
sql.execute("insert into tableName (firstName, lastName) " +
    "values (?, ?)", [firstName, lastName])
```

XML

В groovy существуют билдеры, которые можно использовать для генерации XML. Для генерации создается экземпляр объекта MarkupBuilder, на котором вызываются псевдо-методы — название этого метода и переданные параметры будет использоваться для генерации тега:

```
import groovy.xml.MarkupBuilder

def mb = new MarkupBuilder()

mb.html() {
    head() {
        title("This is the title")
    }

    body() {
        div("class" : "main") {
            p("this is the body")
        }
    }
}
```

Вывод:

```
<html>
  <head>
    <title>This is the title</title>
  </head>
  <body>
    <div class='main'>
      <p>this is the body</p>
    </div>
  </body>
</html>
```

В качестве параметра конструктору MarkupBuilder можно передавать любой PrintWriter:

```
def fb = new MarkupBuilder(new File("index.html").newPrintWriter())
```

Парсинг XML так же очень простой:

```
import groovy.xml.MarkupBuilder
import java.io.StringWriter

def sw = new StringWriter()
def mb = new MarkupBuilder(sw)

mb.html() {
    body() {
        div("class" : "main") {
            p("this is the body")
        }

        div() {
            p("this is the body 1")
            p("this is the body 2")
            p("this is the body 3")
        }
    }
}

def xml = sw.toString()

println xml

import groovy.util.XmlParser;

def parser = new XmlParser()
def doc = parser.parseText(xml)
//def doc = parser.parse("index.html")
```

```
println doc.body.div[1].p[1] // возвращает Node
println doc.body.div[1].p // возвращает список, состоящий из Node
println doc.body.div["@class"] // список значений аттрибута class для всех div
```

Вывод:

```
<html>
  <body>
    <div class='main'>
      <p>this is the body</p>
    </div>
    <div>
      <p>this is the body 1</p>
      <p>this is the body 2</p>
      <p>this is the body 3</p>
    </div>
  </body>
</html>
p[attributes={}; value=[this is the body 2]]
[p[attributes={}; value=[this is the body 1]], p[attributes={}; value=[this is the body 2]],
   p[attributes={}; value=[this is the body 3]]]
[main, null]
```

Groovlets

С помощью класса GroovyServlet возможно запускать скрипты на Groovy как сервлеты.
В первую очередь, для этого нужно добавить несколько строчек в web.xml:

```
<servlet>
   <servlet-name>GroovyServlet</servlet-name>
   <servlet-class>groovy.servlet.GroovyServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>GroovyServlet</servlet-name>
   <url-pattern>*.groovy</url-pattern>
</servlet-mapping>
```

Теперь все запросы для файлов .groovy будут обрабатываться классом GroovyServlet.
В этих скриптах уже доступны для использования следующие переменные:

— request & response
— context, application, session
— out (= response.getWriter())
— sout (= response.getOutputStream())
— html (= new MarkupBuilder(out))

```
html.html() {
   body() {
      div("class" : "main") {
         p("this is the body")
      }

      div() {
         p("this is the body 1")
         p("this is the body 2")
         p("this is the body 3")
      }
   }
}
```

Отдаст браузеру сгенерированную html-страницу.

Список используемых источников:

Kenneth Barclay, John Savage «Groovy programming: an introduction for Java developers»
http://groovy.codehaus.org/

Источник <https://habrahabr.ru/post/122127/>