Scala总结

# Scala简介

Scala是一个函数式静态编程语言，设计初衷是要集成面向对象编程和函数式编程的各种特性。

* 面向对象风格。
* 函数式静态编程语言。
* 自动化类型推导，不需要写变量类型和函数返回值类型，Scala的哲学是编译器能做的就不需要程序员做，所以代码可以做到尽可能的简洁。
* 极度方便的集合操作
* 支持闭包操作
* 可以直接使用Java类库，运行在JVM上，同时拥有动态语言的编程效率和Java的运行效率。

# 与Java异同

如果直接从C/C++开始学习Scala语言可能会比较难，如果你是Java程序员的话这就会比较简单了，接下来通过跟Java语法的对比来开始了解Scala这门强大的语言。

## 与Java相比增加的语法

* 纯OO
* 操作符重载
* closure
* 使用trait进行mixin组合（可作为多重继承的简单替代方案）
* 模式匹配

## 与java相比减少的语法

* 静态成员
* 原生数据类型
* break、continue
* 接口
* enum枚举

# Scala使用

了解了Scala的语法后，我们开始进入Scala语言的编写吧。我一直都认为通过代码来学习语言是最直接最快速的。

## 关于“;” semmicolon

Scala假定任何从句法上独立的语句的两行之间都有一个分号，除非第一行以中缀结尾，或者两行被括在括号或方括号中。

## 关于“()”

Scala遵循统一访问原则："Uniform Access Principle" 意味着变量和无参数的函数都是以同样方式访问。

def loadDate() = {return someDate}

println(loadDate)

## 变量定义

* var 可变，可重新赋值，赋值为"\_"表示缺省值(0, false, null)。
* val 不可变，相当于const/final，但如果val为数组或者List，val的元素可以赋值。
* lazy 定义时不求值，第一次使用时完成求值，保持不变。
* def 定义时不求值，每次使用时都重新求值（可以把这个变量理解为方法，通过反编译的结果可以看出来）

## 集合操作

以下是Scala的Iterable

|  |  |
| --- | --- |
| map | m->m |
| flatMap | m->n |
| indices | m->m |
| foreach | m->Unit |
| for (... if ...) yield | m->n |
| collect { case ... if ... => ... } | m->n |
| filter, filterNot | m->n |
| take | m->n |
| takeWhile | m->n |
| forall | m->1 (true|false) |
| reduceLeft, foldLeft | m->1 |
| scanLeft | m->m+1 |
| exists | m->1 (true|false) |
| find | m->1 (或者None) |
| count | m->1 |
| span, partition | m->2 |

for(i<-1 to 10; j=i\*i) println(j) 从0循环到10，然后打印出这个数字的平方。

List(1,2,3,4).filter(\_%2==1) 过滤List中的偶数

List(1,2,3,4).map(n => n\*n) 循环List中的每个数字然后取平方并返回一个新的List

List(1,2,3).foreach(println) 循环打印List中的数字

## Scala集合和Java集合互操作

### import scala.collection.JavaConverters.\_

引入这个包可将Scala集合与Java集合互相转换

val list: java.util.List[Int] = Seq(1,2,3,4).asJava; 将Scala的Seq转换为Java的List

val buffer: scala.collection.mutable.Buffer[Int] = list.asScala; 将java的List转换为Scala的Buffer

### import scala.collection.JavaConversions.\_

引入这个包Java集合就可以使用Scala集合的所有方便操作

val javaList: java.util.List[Int] = Seq(1,2,3,4).asJava

javaList.foreach(println)

java的集合在Scala代码中就可以像Scala集合那样方便操作了。

## implicit隐式转换

我们经常会在Scala源码中看到implicit这个关键字，对于初学者比较难理解，其实这个是一个很强大的功能，隐式转换是支撑scala的易用、容错以及灵活语法的基础。

### implicit parameters

实现将一个类型的对象在需要的时候转换为另一个类型的对象的功能

implicit def str2Int = (str: String) => str.toInt; 在需要的时候将String转换为Int

implicit def int2Str = (num: Int) => num.toString; 在需要的时候将Int转换为String

### implicit conversion

实现动态的给一个对象增加一些本身并不存在的方法

implicit def exObject(obj: Object) = new {

def asInt = obj.asInstanceOf[Int]

def asLong = obj.asInstanceOf[Long]

}

通过引入这个隐式转换，Object对象可以动态的添加两个方法asInt 和 asLong，将擦除类型信息后的Object再转换回相应的类型

## type做alias

相当于C语言的类型定义typedef，建立新的数据类型名（别名）；在一个函数中用到同名类时可以起不同的别名。

type JDate = java.util.Date

type SDate = java.sql.Date

val d1 = new JDate() // 相当于 val d = new java.util.Date()

val d2 = new SDate() // 相当于 val d = new java.sql.Date()

## 函数

函数的地位和一般的变量是同等的，可以作为函数的参数，可以作为返回值函数和方法一般用def定义；也可以用val定义匿名函数，或者定义函数别名。

除了递归函数和重载的函数以外，其他函数的返回类型都不用写。（scala只能在代码编译的时候做类型推测）

不需要返回值的函数，可以使用def f() {...}，永远返回Unit（即使使用了return）,即：

def f() {...} 等价于 def f():Unit = {...}

需要返回值的函数，用 def f() = {...} 或者 def f = {...}

三种定义方式的区别：

|  |  |  |
| --- | --- | --- |
| def f() = ... | 调用：f, f()皆可 | 返回Unit或者值 |
| def f() { return .. } | 调用：f, f()皆可 | 始终返回：Unit |
| def f = ... | 调用：f | 返回Unit或者值 |

### 匿名函数

val increase = (x: Int) => x + 1

匿名函数式定义一个没有名称的函数，可以赋值给一个变量也可以直接传递给某个方法。

### 简写函数

简写是编译器根据上下文进行自动推测，在有足够信息的情况下可以不用写参数类型或参数名称，简写只能用在向其他函数传递匿名函数时，在其他情况下，编译器没有足够的信息来推测，所以不能使用。

val list = (0 to 10).toList

list.reduceLeft((v1: Int, v2: Int) => v1 + v2)

reduceLeft接受一个接受两个参数的函数，这里是(v1: Int, v2: Int) => v1 + v2，reduceLeft会依次把前一次得到的结果和下一个列表元素传递给函数，最后得到一个单个元素。

通过简写参数类型可以将其改写为：

list.reduceLeft((v1, v2) => v1 + v2)

继续简写参数名称可以将其改写为：

list.reduceLeft(\_ + \_)

### 部分应用函数

def mul(x:Int, y:Int)={x\*y} //定义一个需要两个参数的方法

val mulPart = mul(2, \_: Int) //定义一个偏函数

println(mulPart(5)) // 结果是10

### 柯里化

把原来需要接受两个参数的函数变成两个接受一个参数的函数，新的函数接受一个参数并返回一个接受一个参数的函数。

def mul(x:Int, y:Int)={x\*y} //原来需要传入两个参数

def mul(x:Int)(y:Int)={x\*y} //柯里化后，变成两个接收一个参数的函数

def mul2 = mul(2)(\_) //通过偏函数，创建一个新的函数

println(mul2(5)) //实际上的效果等于 2 \* 5

结果是10

### 闭包（值捕获或引用捕获？）

闭包是可以包含自由（未绑定到特定对象）变量的代码块；这些变量不是在这个代码块内或者任何全局上下文中定义的，而是在定义代码块的环境中定义（局部变量）。“闭包” 一词来源于以下两者的结合：要执行的代码块（由于自由变量被包含在代码块中，这些自由变量以及它们引用的对象没有被释放）和为自由变量提供绑定的计算环境（作用域）。

val times = 2 // 定义一个局部变量

val mulTimes = (x: Int) => x \* times // 定义一个方法，它捕获了局部变量times的值。

println(mulTimes(5)) // 结果是10

def makeMul(x: Int) = {

(y: Int) => y \* x //定义一个返回函数的函数，返回的函数会捕获传入的参数x

}

def mul2 = makeMul(2)

println(mul2(5)) // 结果是10

### 尾递归

尾递归是指一个递归的递归语句出现在最后一个地方，尾递归在最后递归的地方并没有创建一个新的栈，而是跳转到函数的开头，因此可以大幅度的减小递归调用的开销。

如：def factorial(n: Int) = n \* factorial (n - 1)

### 高阶函数

可以接受函数或者返回函数的函数叫做高阶函数。（偏部分应用、柯里化 都是高阶函数的一种）

这儿有一个概念叫做函数的类型，一个函数的类型就是：(参数列表类型)=>返回值类型。

高阶函数可以提供抽象程度更高的处理逻辑，

比如list的方法 def reduceLeft[B >: A](op: (B, A) ⇒ B): B

reduceLeft函数定义了遍历数据的方式，通过传入一个类型为(Int, Int) => Int的函数作为数据处理方式，两者组合起来就成了一个强大的功能。

## 重载和特殊函数名++，+，--，-，%等

方法重载和Java一摸一样，不过Scala中所有的非受保护的符号都可以作为方法名称，所以你可以看到非常函数化的操作方式。

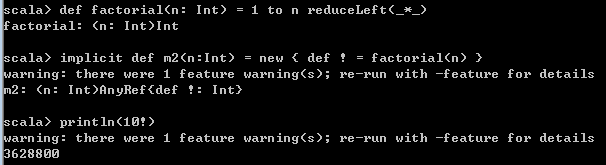
//定义阶乘方法

def factorial(n: Int) = 1 to n reduceLeft(\_\*\_)

//隐式转换，通过匿名类，在需要时把n转换为new m1(n)

implicit def m2(n:Int) = new { def ! = factorial(n) }

//调用



## 类

Scala的类定义方式和Java基本类似，只是构造函数可以直接在类名后面写入，而且构造函数用this定义。

class c1(name:String, age:Int) { // (1)直接在类定义处

def this() { this("anonymous", 20) } // (2)用this定义

def m1() = { printf("%s=%d\n", name, age) }

}

## Object

Scala的Object类似于Java的静态类，里面所有的方法，其他的类或者对象都可以直接调用。

Object Functions {

def add(m: Int, n: Int) => m + n

}

## trait超级接口

和Java的Interface类似，但可以定义实体方法，有点类似C++的多继承。

当定义好trait t1, t2, t3后，如果有一个类需要同时使用t1, t2, t3的特性，那么可以直接继承和组合这三个接口，如下：

trait t1 {def foo1() {prinltn(“I am trait1”)}}

trait t1 {def foo2() {prinltn(“I am trait2”)}}

trait t1 {def foo3() {prinltn(“I am trait3”)}}

class c1 extends t1 with t2 with t3 { def foo(){println(“I am class”)}}

val obj = new c1()

//c1中就继承了3个trait中的方法

obj.foo –> I am class

obj.foo1 --> I am trait1

obj.foo2 --> I am trait2

obj.foo3 --> I am trait3

## match模式匹配

### 常量匹配

区分一些常量，比如：

times match {

case 1 => "one"

case 2 => "two"

case \_ => "some other number"

}

### 使用守卫进行匹配

匹配时后面可以加一个if语句，只有if为true，才会匹配成功，比如：

times match {

case i if i == 1 => "one"

case i if i == 2 => "two"

case \_ => "some other number"

}

### 变量匹配

单纯的变量模式没有匹配判断的过程，只是把传入的对象给起了一个新的变量名。通常不会单独使用，而是在多种模式组合时使用，比如：

List(1,2) match{

case List(x,2) => println(x)

}

### 类型匹配

区分不同的类型，比如：

obj match {

case i: Int => "int"

case d: Double => "double"

case text: String => "string"

case \_ => "unknow"

}

### 构造模式匹配

它的存在使得模式匹配真正变得强大。它由名称及若干括号之内的模式构成。如BinOp("+" , e , Number(0))。这种模式对于树型递归数据尤其有用。

abstract class Expr

case class Var(name:String) extends Expr

case class Number(num:Double) extends Expr

case class UnOp(operator : String , arg : Expr) extends Expr

case class BinOp(operator : String , left : Expr , right : Expr) extends Expr

def simplifyTop(expr : Expr) : Expr = expr match{

case UnOp("-" , UnOp("-" , e)) => e

case BinOp("+" , e , Number(0)) => e

case BinOp("\*" , e , Number(1)) => e

case \_ => expr

}

Console println simplifyTop(UnOp("-", Number(0))) //UnOp(-,Number(0.0))

Console println simplifyTop(UnOp("-", UnOp("-", Number(0)))) //Number(0.0)

Console println simplifyTop(BinOp("\*", Var("bbb"), Number(1))) //Var(bbb)

Console println simplifyTop(BinOp("\*", Var("bbb"), Number(2))) //BinOp(\*,Var(bbb),Number(2.0))

**样本类：添加了case的类便是样本类。这种修饰符可以让Scala编译器自动为这个类添加一些语法上的便捷设定。如下：**

1：添加与类名一致的工厂方法。也就是说，可以写成Var("x")来构造Var对象。

2：样本类参数列表中的所有参数隐式获得了val前缀，因此它被当作字段维护。

3：编译器为这个类添加了方法toString,hashCode和equals等方法。

# Scala进阶

## XML

Scala原生支持xml, 这就让生成和解析xml很简单优雅：

跟声明变量一样，声明XML对象

val name = "james"

val age = 10

val html = <html>name={name}, age="{age}"</html> toString // <html>name=james, age=&quot;10&quot;</html>

从XML中抽XML

val users = <users><user name="qh"/></users>

val <users>{u}</users> = users // u: scala.xml.Node = <user name="qh"></user>

从XML中抽取数据

val x = <r>{(1 to 5).map(i => <e>{i}</e>)}</r>

// <r><e>1</e><e>2</e><e>3</e><e>4</e><e>5</e></r>

(x \ "e") map (\_.text.toInt) // List(1, 2, 3, 4, 5)

## Actor

Scala中处理并发的架构，设计时借鉴Erlang语言的并发框架。actor不共享数据，actor之间通过message通讯。

【!】 发送异步消息，没有返回值。

【!?】 发送同步消息，等待返回值。（会阻塞发送消息语句所在的线程）

import actors.Actor, Actor.\_

val server = Actor.actor {

loop {

react {

case x:Int => println("a1 stop: " + x); exit()

case msg:String => println("a1: " + msg)

}

}

}

server ! "hello" // "a1: hello"

server ! "world" // "a1: world"

server ! -1 // "a1 stop: -1"

server ! "no response :("

# 最后

## 优点

通过语法糖、方便的集合操作、类型推导，可以减少冗余代码、提高程序员的生产力。

通过高阶函数，提供更高层次的抽象，可以设计出更精炼、更漂亮的代码。

## 缺点

语法、用法比较复杂，多范式学习（面向对象、函数式）导致开发应用的门槛比较高。