# Scala基础知识

## 10. Scala单例对象、伴生对象

### 10.1 Scala单例对象

Scala中没有java中的静态类、静态方法、静态成员，相应的scala中提供了Object对象，在Object对象中所有的成员和方法都是静态的。

package lesson10  
  
/\*\*  
 \* Created by hipparchus on 2017/2/9.  
 \*/  
  
class College{  
 val id = College.studentNo  
 private var number = 0  
 def addClass(number:Int){this.number += number}  
}  
  
object College{  
 private var studentNo = 0  
 def newStudentNo = {  
 studentNo += 1  
 studentNo  
 }  
}  
  
object ObjectOps {  
 def main(args: Array[String]) {  
 println(College.newStudentNo)  
 println(College.newStudentNo)  
 }  
}

在以上代码中，Object College就是一个Object对象，它的方法可以通过*对象名.方法名*的方式来调用，但是需要注意的是在main方法中调用的Object对象的方法或成员不能使用private关键字修饰。

输出结果：

1

2

由于Object对象内的方法和成员都是静态的，所以Object对象可以作为静态变量区，存放一些公共的变量和方法。例如一些配置信息需要在Object对象中定义。

与java的静态类不同的是，java的静态类只要加载静态类的时候会对静态类的方法执行，但是scala中的Object对象只用在第一次调用时才会执行其内部的方法和成员，而不是每次加载时都执行。其实Object对象自身也存在构造器，在上例中，Object College存在一个无参数的构造器，在第一次调用时会调用构造器来创建Object College对象。

### 10.2 类的伴生对象

如果存在一个类与一个Object对象名字相同，则称这个object对象是这个类的伴生对象，这个类是这个对象的伴生类。

class College{  
 val id = College.studentNo  
 private var number = 0  
 def addClass(number:Int){this.number += number}  
}  
  
object College{  
 private var studentNo = 0  
 def newStudentNo = {  
 studentNo += 1  
 studentNo  
 }  
}

在伴生对象中会对类的一些静态配置和一些隐式转换进行的封装。伴生类可以访问伴生对象的所有成员（包括私有成员与非私有成员），例如上例中class College可以访问伴生对象的私有成员 studentNo。值得一提的是伴生对象并不是在伴生类的作用范围内的，所以访问伴生类的成员时需要使用*对象名.成员名*的方式。

## 11.scala中的apply方法

在Scala中apply方法有两种用法一种是在Object伴生对象中调用apply方法，另一种是在类中调用apply方法,前者使用比较多。

### 11.1 Object中的apply

在Object中使用apply方法大多数是用于构造对象，利用半生对象的apply方法来实现伴生类的对象的实例化。

package lesson11

/\*\*

\* Created by hipparchus on 2017/2/9.

\* apply function

\*/

class ApplyTest{

def apply() = println("I am into Spark!")

def haveATry: Unit ={

println("Have a try on apply!")

}

}

object ApplyTest{

def apply() = {

println("I am into Scala!")

new ApplyTest

}

}

object ApplyOperation {

def main(args: Array[String]) {

val arr = Array(1,2,3,4,5)

val a = ApplyTest()

a.haveATry

}

}

这里有一个伴生类ApplyTest和其伴生对象object ApplyTest，并且中伴生对象中实现了apply方法返回一个ApplyTest的对象。

当在main方法中通过*val a =ApplyTest()* 方式创建对象时，scala会调用object ApplyTest 的apply方法，来创建对象。

输出：

I am into Scala!

Have a try on apply!

在构建集合时，一般都是采用这种方式来构建集合对象的，例如创建数组对象：

val arr = Array(1,2,3,4,5)

实际上是调用了Array类的半生对象的apply方法。使用这种方式有如下几个好处：1)进行单例控制，2)可以控制对象构建的权限控制，3）避免多次使用new来创建对象，4）通过在伴生类apply方法中构建抽象类的子类实例的方式来实例化抽象类。

### 11.2 Class中的apply方法

这种方式使用的比较少。

class ApplyTest{

def apply() = println("I am into Spark!")

def haveATry: Unit ={

println("Have a try on apply!")

}

}

object ApplyOperation {

def main(args: Array[String]) {

val a = new ApplyTest

a.haveATry

println(a())

}

}

一般调用class的apply的方法是通过*对象名()*方式来调用的。

结果：

Have a try on apply!

I am into Spark!

()

## 12. Scala中的继承

### 12.1超类的构造

Scala中的类与java的类似都是继承自一个默认顶级类Any（java是Object）。

package lesson12

/\*\*

\* Created by hipparchus on 2017/2/9.

\* scala inherit

\*/

class Person(val name:String, var age:Int){

println("The primary constructor of Person")

val school = "XDU"

def study = "study in the morning"

override def toString = "I am a Person,name:" + name +",age:" + age

}

class Student(name :String, age:Int, val grade:String) extends Person(name,age){

println("This is the subClass of Person, Primary constructor of Student")

override val school = "BJU"

override def toString = "I am a student,name:"+name+",age:" + age + ","+super.study

}

object OverrideOperations {

val w = new Student("hipparchus", 25, "grade four")

println("School:" + w.school)

println("Grade:"+ w.grade)

print(w.toString)

}

这里，创建了一个超类Person，然后创建了一个继承自Person的子类Studenet。通过关键字extends来表明继承自哪个父类。Person类中有两个类型参数分别是name和age，而子类中有三个类型参数（name、age、grade）。子类通过super关键字来调用父类的成员。当子类有与父类相同的类型参数时，需要把类型参数同时传递给父类，例如：

class Student(name :String, age:Int, val grade:String) extends Person(name,age)

总结，1）子类在继承时，需要将父类主构造的所有参数都填充了；

2) 子类调用父类方法时，通过关键字super。

运行结果：

The primary constructor of Person

This is the subClass of Person, Primary constructor of Student

School:BJU

Grade:grade four

I am a student,name:hipparchus,age:25,study in the morning

### 12.2 重写字段

一般情况下，建议在子类重写父类的方法或者成员时添加override关键字。

如：

class Person(val name:String, var age:Int){

println("The primary constructor of Person")

val school = "XDU"

}

class Student(name :String, age:Int, val grade:String) extends Person(name,age){

println("This is the subClass of Person, Primary constructor of Student")

override val school = "BJU"

}

子类Student继承自父类Person，并且重写了父类的成员school。

### 12.3 重写方法

class Person(val name:String, var age:Int){

println("The primary constructor of Person")

override def toString = "I am a Person,name:" + name +",age:" + age

}

class Student(name :String, age:Int, val grade:String) extends Person(name,age){

println("This is the subClass of Person, Primary constructor of Student")

override def toString = "I am a student,name:"+name+",age:" + age + ","+super.study

}

由于Person类默认继承自scala的顶层类Any，而Any类中定义了toString方法，所以Person类重写了toString方法。类似子类Student也重写了父类Person的toString的方法。

## 13.抽象类、抽象字段、抽象方法

### 13.1抽象类

抽象类在scala的编程中常常用到。在面向对象编程和面向接口编程的角度来讲，抽象类是至关重要的。

声明抽象类使用关键字：*abstract*。

实现抽象类时，子类需要使用*extends*关键字继承抽象类，然后实现其中的抽象方法。

package lesson13

/\*\*

\* Created by hipparchus on 2017/2/10.

\* Abstract Class

\*/

class AbstractClassOps{

var id:Int = \_

}

abstract class SuperTeacher(val name:String){

var id:Int

var age:Int

def teach

}

class MathsTeacher(name:String) extends SuperTeacher(name){

override var id = name.hashCode

override var age = 25

override def teach: Unit ={

println("Teach Maths!")

}

}

object AbstractClassOps {

def main(args: Array[String]) {

val teacher = new MathsTeacher("Maths Teacher Hipparchus")

teacher.teach

println("teacher.id:" + teacher.id)

println("teach.name:" + teacher.name)

}

}

输出结果：

Teach Maths!

teacher.id:-143635800

teach.name:Maths Teacher Hipparchus

### 13.2抽象字段

当一个字段在声明的时候，不给字段赋值时，该字段就是抽象字段。需要注意的是非抽象类的每个字段在声明时必须要进行赋值（此时可以通过给字段赋值为具体值或者给*var*类型的字段赋值为一个占位符*\_*）。

class AbstractClassOps{

var id:Int = \_

}

abstract class SuperTeacher(val name:String){

var id:Int

var age:Int

}

### 13.3抽象方法

当一个方法没有方法体的时候，它就是一个抽象方法。与java不同的是scala中抽象方法不需要有*abstract*关键字来进行修饰。

abstract class SuperTeacher(val name:String){

def teach

}

## 14.scala中的trait（特质）

### 14.1作为接口使用的trait

与java中的interface接口类似，scala中的trait可以作为接口来讲，但是它要比java中的interface强大的多。

声明一个trait只需要使用关键字trait即可。

trait Logger{

def log (msg:String){}

def loginfo

}

与java的interface不同，trait中可以有抽象方法也可以有实例方法。

当一个类需要实现多个trait时，第一个trait使用*extends*关键字，而后的所有trait都使用*with*关键字。

class ConcreteLogger extends Logger with Cloneable{

override def log(msg:String) = print("Log:" + msg)

def concreteLog{

log("concrete log!")

}

}

举个例子：

package lesson14

/\*\*

\* Created by hipparchus on 2017/2/10.

\* trait

\*/

class UseTrait{

}

trait Logger{

def log (msg:String){}

}

class ConcreteLogger extends Logger with Cloneable{

override def log(msg:String) = print("Log:" + msg)

def concreteLog{

log("concrete log!")

}

}

object UseTrait {

def main(args: Array[String]) {

val logger = new ConcreteLogger

logger.concreteLog

}

}

输出结果：

Log:concrete log!

### 14.2在对象中混入trait

在实际的情况中有一个需要，在具体类实例化的时候，通过一些trait的具体的实现来提供具体的实现方法。

例如

trait Logger{

def log (msg:String){}

}

class ConcreteLogger extends Logger with Cloneable{

def concreteLog{

log("concrete log!")

}

}

在trait Logger和ConcreteLogger中都没有具体的写log方法的方法体。但在TraitLogger(TraitLogger 也是一个trait)中有具体的方法体内容。

trait TraitLogger extends Logger{

override def log (msg:String): Unit ={

print("TraitLogger Log content is : " + msg)

}

}

此时我们通过以下方式来构建实例对象。

object UseTrait {

def main(args: Array[String]) {

val logger = new ConcreteLogger with TraitLogger

logger.concreteLog

}

}

由于此时TraitLogger中对log方法进行了复写，此时可以通过以上方式将TraitLogger中log方法具体的实现混入到对象logger中，覆盖掉logger对象的log默认方法。此时需要注意的是这两个trait必须类型一致。

运行结果：

TraitLogger Log content is : concrete log!

这在很多情况下是非常有用的，它可以随着程序的运行动态的混入更符合当时上下文的方法的实现。这要得益于trait中的很多工具方法（有实现的方法）。在scala中可以混入很多trait，这样就可以把各种各样的工具方法混入到类和方法中，这对于构造大型工程或者复杂的上下文环境很有必要。

## 15.scala多重继承与AOP

### 15.1 scala多重继承

Scala中实现多重继承的方式一般采用混入多个trait的方式。由于trait与java中的interface不同，它本身可以用方法的实现，这样在进行大型工程的一个具体模块时，可以让模块混入各种不同的trait来实现模块功能的不同方面。

class Human{

println("Human")

}

trait Teacher extends Human{

println("Teacher")

def teach

}

trait PianoPlayer extends Human{

println("PianoPlayer")

def playPiano = {

println("I'm playing piano.")

}

}

class PianoTeacher extends Human with Teacher with PianoPlayer{

override def teach = {

println("I'm training student.")

}

}

这里PianoTeacher类继承自Human类混入了trait Teacher和PianoTeacher的工具方法，并且复写了Teacher的teach方法。

### 15.2 scala多重构造器执行顺序

在scala的多重构继承时，会有一个构造顺序。这个顺序是从左往右进行构造。

class PianoTeacher extends Human with Teacher with PianoPlayer{

在这个例子中，在实例化PianoTeacher时，会先调用Human的构造器，然后调用Teacher的构造器，最后调用PianoPlayer的构造器。这里会有一个问题，由于Teacher和PianoPlayer都继承自Human，但是并不是每次都会调用Human的构造方法。在scala中多继承时，如果已经构造了后面trait的超类的话，那么后面的trait不会再重复构造其超类。

例如：

class Human{

println("Human")

}

trait Teacher extends Human{

println("Teacher")

def teach

}

trait PianoPlayer extends Human{

println("PianoPlayer")

def playPiano = {

println("I'm playing piano.")

}

}

class PianoTeacher extends Human with Teacher with PianoPlayer{

override def teach = {

println("I'm training student.")

}

}

object UseTrait {

def main(args: Array[String]) {

val t1 = new PianoTeacher

t1.playPiano

t1.teach

}

}

输出结果：

Human

Teacher

PianoPlayer

I'm playing piano.

I'm training student.

### 15.3 基于trait的AOP的实现

这里简单的介绍利用trait如何实现AOP编程。

举个例子：

trait Action {

def doAction

}

trait TBeforeAfter extends Action{

abstract override def doAction: Unit = {

println("Initialization")

super.doAction

println("Destroyed")

}

}

class Worker extends Action {

override def doAction = println("Working...")

}

这里有一个trait Action代表处理类具体执行的内容，trait TBeforeAfter代表处理类处理前、后所做的预备和收尾工作。由于TBeforeAfter的daAction方法需要调用其父类Action的daAction方法，而daAction方法是抽象方法，所以TBeforeAfter的daAction方法需要用*abstract override*关键字修饰。Worker类继承了Action类。

object UseTrait {

def main(args: Array[String]) {

val worker = new Worker with TBeforeAfter

worker.doAction

}

}

在具体实例化时，可以通过以上方式来实现AOP编程。首先Worker类复写了action方法，然后TBeforeAfter会被混入到对象worker中。所以在调用worker对象的doAction其实是调用TBeforeAfter的doAction方法；TBeforeAfter的doAction方法会调用父类的doAction方法，通过反射调用worker的doAction方法。

输出结果：

Initialization

Working...

Destroyed

## 16.scala中的包

### 16.1 scala中的包定义

包对于组织大型功能来说十分重要，在构建大型工程时一般都会将功能分成不同的模块，然后有不同的团队或者同一个不同人员负责，此时就需要使用包来分工模块、组织和管理代码。

在scala中使用关键字package来定义一个包。

package spark.navigation {

abstract class Navigator

package tests {

//I'm in package spark.Navigation.test

class NavigatorSuite

}

}

package hadoop {

package navigation {

class Navigator

}

package launch {

class Booster {

val nav = new navigation.Navigator

}

}

}

包可以使用链式的定义方式即支持包的嵌套，如package spark.navigation。在包中可以实现代码的组织管理，例如：

package spark.navigation {

abstract class Navigator{

def act

}

package tests {

//I'm in package spark.Navigation.test

class NavigatorSuite

}

package impls {

class Action extends Navigator{

override def act {

println("action")

}

}

}

}

在包spark.navigation中嵌套定义了两个包test和impls，此时就可以将spark.navigation中的抽象方法的实现放在包impls中，对于一些单元测试的方法可以放在test包中。例如spark.navigation中有一个类Navigator，其中的抽象方法act的具体实现放到了test包中。这样便可以清晰的管理整个代码。如果包的定义中有{}，则其作用域在代码快{…}中，如果没有{}的话，则其作用域是整个scala文件。

### 16.2 包对象

Scala中包存在包对象的概念，包对象一般使用关键字*package object 包名* 的方式定义，在包对象中我们可以定义方法和成员。当存在一个包的包名和包对象名称相同时，包中的类就可以直接访问包对象中成员。例如：

package object people {

val defaultName = "Scala"

}

package people {

class people {

var name = defaultName

}

}

包people可以访问包对象people中的成员 defaultName。

一般情况下包不可以单独定义成员和方法，所以使用包对象来代替。

### 16.3 包引用

一般在同一个scala文件中不需要显示的导入其他包，当不在同一个scala文件需要通过关键字import导入。例如：

import java.awt.{color,Font}

### 16.4 包的隐式引用

Scala中存在一个隐式的包引用，默认情况下是不需要引入直接可使用：

import java.lang.\_ //java.lang包

import scala.\_ //scala包

import Predef.\_ //Predef包

这里”\_”等同于java中的”\*”,而且默认情况下scala可以引用所有的java包。

import java.awt.{color,Font}

为了避免scala包与java中类名冲突可以使用重命名引入包的方式。

import java.util.{HashMap=>JavaHashMap}

如果不想引入一个包中的特定类可以使用如下方法。

import scala.{StringBuilder => \_}

## 17.scala中的访问权限

由于在大型面向对象的项目中存在许多包、类、对象等等，进行合理的访问控制是组织和构建项目必须要考虑的。

### 17.1 包、类、对象、成员的访问权限

在包、类、对象、成员中有如下几个访问控制。

**private[包名/类名]**，是指限定一个类、类的成员或者方法的可见度，即这个类的可见度可以扩展到包名下的所有包。如：

package spark {

package navigation{

private[spark] class Navigator{

}

package launch{

object Vehicle {

private[launch] val guide = new Navigator

}

}

}

}

类Navigator的可见度为整个spark包，所以在包launch中可以直接使用类Navigator。而lauch包中的对象Vehicle中的成员guide是整个lauch包可见的，lauch包中的所有类都可以访问它。

**protected[包名/类名]**，指这个方法或者类不仅其和其子类可以访问，并且包和包的子包也可以访问。

**private[this],**指方法、成员、类仅对象可见或者对象私有，也就是仅有该类的实例化的同一对象内部可以访问，类的其他对象无法访问。可以通过这一关键字来限定某一类、方法、成员不能被同一类的其他对象访问，这对于实际编程十分有用。

package spark {

package navigation{

private[spark] class Navigator{

protected [navigation] def useStarChar(): Unit = {

class LegOfJourney {

private[Navigator] val distance = 100

}

private[this] var speed = 200

}

}

}

}

这里类Navigator中的成员speed的访问修饰符是private[this],就是仅仅是对象可见的，同一类的其他对象是无法访问的。

### 17.2 伴生类、伴生对象的访问权限

伴生类可以访问伴生对象的私有成员，伴生对象也可以访问伴生类的私有成员，即伴生类和伴生对象可以相互访问。唯一的差别在于protect修饰的成员，由于伴生对象不存在子类，所以不存在伴生类的子类可访问的问题。

class PackageOps\_Advanced{

import PackageOps\_Advanced.power

private def canMakeItTrue = power > 10001

}

object PackageOps\_Advanced{

private def power = 10000

def makeItTrue(p:PackageOps\_Advanced):Boolean = {

val result = p.canMakeItTrue

result

}

}

## 18.scala中的文件操作

### 18.1 文件的读取、写入操作

一般情况下，编写scala工程都会涉及到文件的读入和写入的操作。它是编程中需要具备的一项基础能力。

通常情况下，读取文件都是都过scala.io包中的Source.fromFile方法来实现的。该方法的入参为代表路径的字符串，返回值是一个BufferSource。BufferSource是一个文件的迭代器。迭代器的内容是文件的内容，按行迭代。

package lesson18

import scala.io.Source

/\*\*

\* Created by hipparchus on 2017/2/12.

\* file Options

\*/

object FileOps {

def main(args: Array[String]) {

val file = Source.fromFile("D:\\work\test.txt")

for (line <- file.getLines()){

println(line)

}

file.close()

}

}

输出结果：

test

file

Source对象还可以读取URL内容，例如

val webFile = Source.fromURL("http://spark.apache.org/")

webFile.foreach(print)

webFile.close()

文件的存入，此时可以使用java的PrintWriter对象来实现文件写入磁盘的操作。

val writer = new PrintWriter(new File("scala.txt"))

for (i <- 1 to 100) writer.println(i)

writer.close()



### 18.2 读取控制台输入

有些时候需要动态的输入一些参数，例如从控制台输入参数，scala程序读取控制台的输入，这时就需要能够实现scala程序读取控制台输入的信息。

println("please enter your input:")

val line = Console.readLine

println("Thank you ,just typed:" + line)

输出结果：

please enter your input:

1

Thank you ,just typed:1

补充：在scala中的输入与输出流基本上都是调用java来进行操作的，所以scala中的输入与输出内容可以参照java的输入与输出操作。

## 19 scala中的正则表达式

### 19.1 正则表达式

正则表达式对于代码编写来说十分有用，它描述了一种规则，这种规则可以适用与具体的实例的匹配或者过滤，例如字符串的查找、匹配等等。这在数据进行清洗（ETL）时十分有用。

与其他语言类似，scala通过如下方法生成正则表达式

val regex = """([0-9]+) ([a-z]+)""".r

val numPattern = """[0-9]+""".r

val numberPattern = """\s+[0-9]+\s+""".r

这里的”””是指其中的内容作为原始字符串来使用，不含转义字符。通过字符串的***.r***方法来生成一个正则表达式。

package lesson19

/\*\*

\* Created by hipparchus on 2017/2/12.

\* regExpress

\*/

object RegExpressOps {

def main(args: Array[String]) {

val regex = """([0-9]+) ([a-z]+)""".r

val numPattern = "[0-9]+".r

val numberPattern = """\s+[0-9]+\s+""".r

//findAllIn 返回遍历所有匹配项的迭代器

for (matchString <- numPattern.findAllIn("999 scala,1234 spark")) println(matchString)

//找到首个匹配项 findFirstIn 返回的是一个Option类型的数据，找到是Some(匹配内容),未找到是None

println(numberPattern.findFirstIn("999 scala,1234 spark"))

val numitemPattern = """([0-9]+) ([a-z]+)""".r

val numitemPattern(num, item) = "99 hadoop"

println("num:" + num + ",item:" + item)

}

}

输出结果：

999

1234

None

num:hadoop,item:99

### 19.2 模式匹配与Reg结合

在构建工程时，常常将正则表达式和模式匹配一起使用，来进行不同的结构控制，类似与java的switch…case…表达式。

val line = "90809 spark"

line match{

case numitemPattern(num,blog) => println(num + "\t" + blog)

case \_ => println("defalut!")

}

输出结果：

90809 spark

会这里首先会通过模式匹配，匹配上“90809 spark”,并将90809赋值给num，”spark”赋值给blog，然后执行case表达式的println(num + "\t" + blog)。

## 20 scala中的本地函数

### 20.1 本地函数

在使用函数进行操作的封装时，随着函数的数量和嵌套增多会带来一个副作用就是不容易控制函数封装的粒度，对于一些函数可见性的控制会越来越难。

虽然此时可以使用一些访问控制符例如*private、protect*等，但是更常见的就使用在函数内部定义函数的方法即本地函数的方法来实现这类功能。

本地函数是定义在函数内部的函数，只有在函数内部可以访问，函数外部无法访问。而本地函数是可以访问外部函数传入的参数的。

例如：

def processData(filename:String, width:Int){

def processLine(line:String){

if(line.length > width)

println(filename + ":" + line)

}

val source = Source.fromFile(filename)

for(line <- source.getLines)

processLine(line)

}

这里本地函数processLine是函数processData的一个本地函数。一般在使用时不希望过多的暴露processDate的内部实现细节，而本地函数只有内部能使用外部不可见，并且processLine可以使用外部函数processDate的参数width，这样就可以做到封装processDate内部的实现细节，从而达到高内聚低耦合的效果。

### 20.2 一等函数

在scala中函数是可以当作变量来使用的，即在能使用变量的地方都可以用函数代替，此时这些函数就叫做一等函数。这也是scala支持本地函数的原因，因为函数内部可以使用变量，从而函数内部也就可以使用函数了。例如：

object FunctionOps {

def main(args: Array[String]) {

val width = args(0).toInt

for (arg <- args.drop(1))

processData(arg,width)

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

increase(10)

increase = (x:Int) => x + 9999

}

}

在main函数中定义了一个匿名函数*(x:Int) => x + 1*并将它赋值给了变量increse。这个匿名函数需要出入一个整型参数x，并把这个参数映射(加工)成x+1即自增1。实际上这里的匿名函数会被scala转换成一个函数字面量来处里。当我们调用这个函数时，可以通过变量incerese(x)的方式来调用。

在scala中提供了很多是代码变得简洁的方法，例如：

someNumbers.foreach((x:Int) => println(x))

someNumbers.filter((x:Int) => x > 0)

someNumbers.filter((x) => x > 0)

someNumbers.filter(x => x > 0)

someNumbers.filter(\_ > 0)

由于scala的类型推断等一些特性，最终*someNumbers.filter(\_ > 0)*代码就代表了someNumbers.foreach((x:Int) => println(x))

## 21 scala中的偏函数

### 21.1 偏函数

偏函数实际上是一种表达式，这种表达式在使用时只需提供部分参数即可。例如：

def sum(a:Int, b:Int, c:Int) = a + b + c

println(sum(1,2,3))

val fp\_a = sum \_

println(fp\_a(1,2,3))

println(fp\_a.apply(1,2,3))

这里，先定义了一个函数sum，然后将sum函数原型(sum \_)赋值给了常量fp\_a,这里的占位符\_ 指代了sum的三个参数。实际上，scala编译器会先生成一个函数类，类中的apply方法会接受3个参数然后调用sum来进行计算。所以当时用函数fp\_a时，scala编译器会产生一个类的实例，调用这个对象的apply来进行计算。

输出结果：

6

6

偏函数的定义：

val fp\_b = sum(1, \_ :Int ,3)

println(fp\_b(2))

println(fp\_b(10))

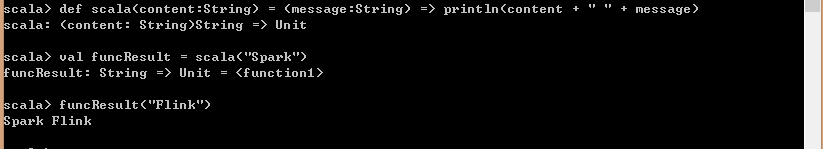
这里定义了一个偏函数fp\_b,此时sum函数的第一个和第三个参数默认给出，而第二个参数使用占位符\_来代替，并限定了参数类型为Int。当我们调用时，只需传入一个参数即可。

一般情况下，在偏函数中使用占位符\_ 来代替部分参数列表或者全部参数列表。在实际运用中，可以都过给函数传递不同的参数来构造不同的偏函数，实现不同的功能。这对于scala实现函数的闭包，高阶函数的调用有很大作用。

## 22 scala中的闭包

### 22.1 闭包

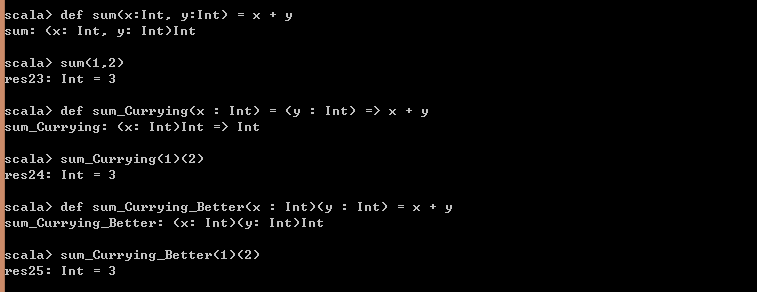
Scala中的闭包，简单来说就是指一个变量在其作用域以外仍能进行访问的现象。例如：



在这里我们首先定义了返回值为函数的一个函数，然后我们调用了该函数，传入的参数为Spark，通常当函数scala运行结束后，我们是不能在访问该参数了，因为局部变量Spark的作用域也就是生命周期结束了，但是当我们调用funcResult函数后，发现它是能访问参数Spark的。这就是函数的闭包。scala内部是创建了一个函数的内部对象，将参数Spark作为一个成员保留在了这个对象中。

科里化：将一个函数的两个参数，我们可以把它转换成两个函数，第一个函数会接收原函数的第一个参数，第二个的函数会接收原函数的第二个参数作为输入变量的函数。

例如：



其利用了闭包。

又如：

val data = List(1,2,3,4,5,6)

var sum = 0

data.foreach(sum += \_)

def add(more:Int) = (x:Int) => x + more

val a = add(1)

val b = add(999)

println(a(10))

println(b(10))

输出结果：

11

1009

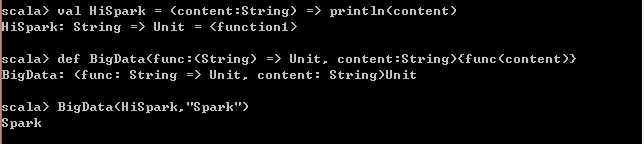
这里的函数a、b都是用了闭包，显而易见参数more是函数add的参变量，当a、b定义时调用了add，但是随着add的调用结束more的生命周期也就结束了，但是但我们调用a、b函数时发现参数more仍能被访问。实际上，scala编译器在这里创建了一个对象用来保存这些变量，由于add被调用时返回的是一个函数，而这个函数需要使用add的参数，所以新的对象中保存了add的参数，以便使该参数能有更长的生命周期。

## 23 scala高阶函数

### 23.1 高阶函数

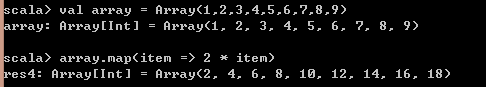
高阶函数是函数的参数也是函数。（因为函数的参数可以是变量，而函数又可以赋值给变量，即函数和变量地位一样，所以函数参数也可以是函数）

如：



首先我们定义了一个函数BigData,这个函数有两个参数，第一个参数是一个函数，函数名是func，他有一个String类型的参数并且返回值是unit空的；第二个参数是String类型的变量名为content的变量，函数体是将第二个参数作为第一个参数也就是函数func的参数，来调用第一个函数，整个函数返回值为unit空。这里只要传入的函数的格式与定义的一致就行。

又如：



Array.map()作用，他会遍历array中每一个元素，并将每个元素作为具体的值传给map中的作为参数的函数。

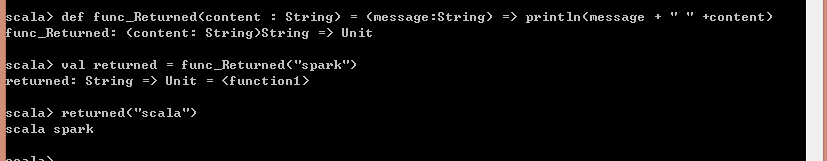
高阶函数的返回值是函数（更为重要）。



首先，我们定义一个函数作为返回值的一个函数，func\_Returned返回的是一个字符串类型的。



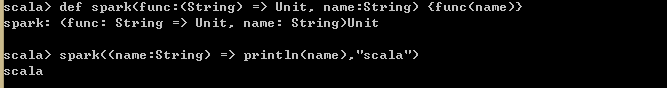
我们输入func\_Returned(“Spark”),生成了一个名为Spark的函数。



首先我们定义了一个返回值为函数的函数func\_Returned,然后我们将其返回值也就是一个函数赋值了变量returned，这样就得到了一个名为returned的函数，然后我们调用returned函数得到了打印结果。

高阶函数有个非常有用的特性是类型推断。其可以自动推断出参数的类型，而且对于只有一个的参数的函数，可以省略掉小括号，并且在函数的参数作用的函数体内只是用一次函数的输入参数的值话，就可省略掉函数名，用下划线（\_）代替。

最原始的方法 ：



由于scala可以类型推断，而且func只有一个参数，由于在spark定义是一定义了变量类型，所以我们就可以把name后的类型String去掉，



由于只有一个参数故小括号也可以去掉，

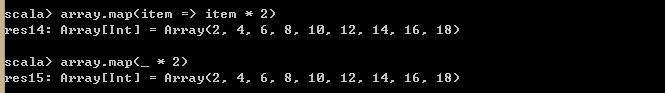


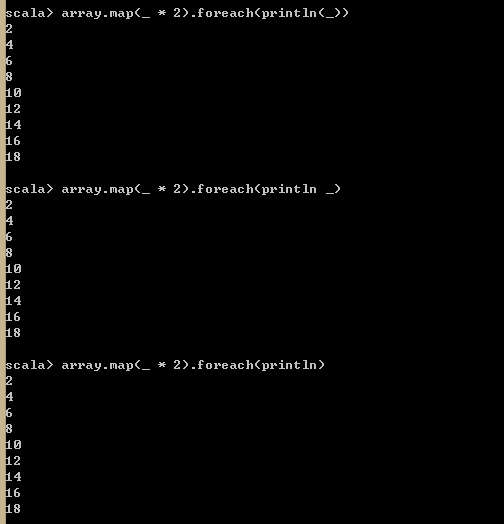
有由于只有一个参数，所以其参数可以省略，用下划线代替

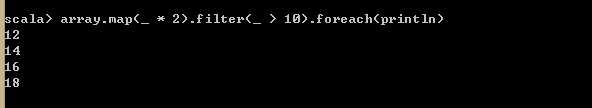




例如：







## 24 scala中SAM转换

### 24.1 SAM转换

SAM是single abstract method的缩写，是指通过指定抽象方法的具体实现来实例化一个抽象接口的方法。例如：

var data = 0

val frame = new JFrame("SAM Testing")

val jButton = new JButton("Counter")

jButton.addActionListener(new ActionListener {

override def actionPerformed(e: ActionEvent): Unit = {

data += 1

println(data)

}

})

这里的new ActionListener{…}就是SAM转换代码，这其中的new ActionListener{…}都是样板代码，如果接口需要实现的方法很多事会显得很复杂。这里更关注的是实现的逻辑即data加一、打印操作，而不是监听器ActionListener和回掉方法是什么。这里更聚焦于数字的处理。

Scala中可以通过引入隐式转换的方法实现更加简洁的SAM。

implicit def convertedAction(action:(ActionEvent) => Unit) =

new ActionListener {

override def actionPerformed(e: ActionEvent): Unit = {action(e)}

}

jButton.addActionListener((e:ActionEvent) => {data += 1; println(data)})

这里定义了一个隐式转换convertedAction，是将一个接收ActionEvent类型参数，返回值为unit的函数转换为一个SAM实例化的接口，即将java中的样例代码封装了以下。这样当在调用jButton.addActionListener方法时，只需传入处理逻辑的函数。jButton.addActionListener方法执行时发现内部的方法类型不符合，scala编译器扫描上下文，通过隐式转换的方法将函数转换为合适的SAM类型。

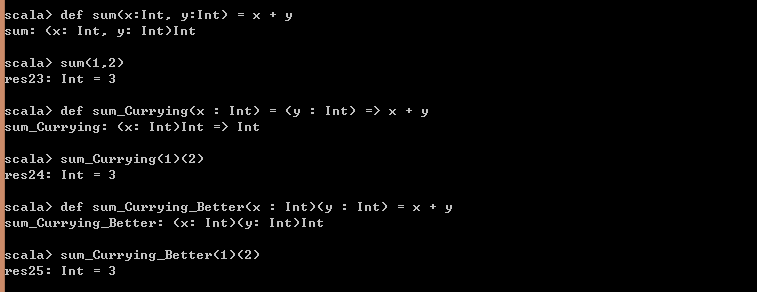
从java提供的样板代码转换为关注于数据处理的代码，这就是SAM转换的优点。这里通过隐式转换复写了原来的java样例代码。

## 25 Scala中的Curring

### 25.1 Curring

Curring即科里化，可以把接受多个参数的函数转换为一个参数的方法。例如将一个函数的两个参数，我们可以把它转换成两个函数，第一个函数会接收原函数的第一个参数，第二个的函数会接收原函数的第二个参数作为输入变量的函数。

例如：



其利用了闭包。

又如：

def multiple(x:Int,y:Int) = x + y

def multipleOne(x:Int) = (y:Int) => x \* y

println(multipleOne(6)(7))

这里将原本需要两个参数的函数multiple转换为了只需一个参数的multipleOne，然后通过multipleOne(arg1)(arg2)的方式调用这个函数。实际上在scala编译器编译时，会先将multiple(arg1)转换为一个函数*(y:Int) => arg1 \* y，*然后这个函数传入参数arg2进行计算。

Curring的一个用途是进行类型推到更方便。

val a = Array("Hello","Spark")

val b = Array("hello","scala")

println(a.corresponds(b)(\_.equalsIgnoreCase(\_)))

def corresponds[B](that: GenSeq[B])(p: (A,B) => Boolean): Boolean = {

val i = this.iterator

val j = that.iterator

while (i.hasNext && j.hasNext)

if (!p(i.next(), j.next()))

return false

!i.hasNext && !j.hasNext

}

这里corresponds函数中由于第一个参数已制定了类型，第二个参数就可以适用类型推到而无需指定具体的参数类型。

## 26 Scala中的模式匹配

### 26.1 模式匹配

Scala中在进行分布式编程时，使用模式匹配和消息通信相结合的方式可以使得分布式编程更加简洁。

模式匹配类似与java的switch…case表达式，但是有些许不同。

模式匹配的使用方法如下：

val data = 30

data match {

case 1 => println("First")

case 2 => println("Second")

case \_ => println("Not Known Number")

}

这里通过match…case来实现模式匹配，match前的data即为需要进行匹配的变量，然后通过case表达式来进行具体的匹配，case \_表示匹配其他所有情况类似于switch…case中的default。Case表达式的方法体默认是直接break的，不会继续执行后面的case表达式。

在进行模式匹配时，可以使用守卫的方式进行case过滤，例如：

val result = data match {

case i if i == 1 => "The First"

case number if number == 2 => "The Second" + number

case \_ => "Not Known Number"

}

println(result)

*case i if i == 1 => "The First"*中使用了一个常量 i来接受data match传入的data，然后使用这个常量i来进行守卫判断或者在case表示后的方法体内直接使用。

更多的是模式匹配与集合等其他方法相结合使用。

"Spark !"foreach{

c => println(

c match {

case " " => "space"

case ch => "Char:" + ch

}

)

}

这里*"Spark !"*会转换为一个字符串集合，通过便利字串集合同每一个字符进行模式匹配，然后打印不同的值。

输出结果：

Char:S

Char:p

Char:a

Char:r

Char:k

space

Char:!

## 27 Type、Array、List、Tuple的模式匹配

### 27.1 Type模式匹配

在scala中的模式匹配可以根据类型来进行模式匹配，不同的类型进行不同的数据处理。例如：

def match\_type (t:Any) = t match {

case p:Int => println("this is Integer")

case p:String => println("this is String")

case m:Map[\_, \_] => m.foreach(println)

case \_ => println("UnKnown type!")

}

match\_type(2)

match\_type(Map("Scala" -> "Spark"))

这里根据匹配的变量的不同类型进行匹配处理。需要提一下的是scala中对于泛型或者高级类型，例如Map，它会擦除Map中的k.v的类型，所以这里使用占位符\_来表示Map的k,v的类型。

输出结果：

this is Integer

(Scala,Spark)

### 27.2 Array模式匹配

模式匹配也可以使用在Array中，例如：

def match\_array(arr :Any) = arr match {

case Array(0) => println("Array: " + 0)

case Array(x,y) => println("Array: " + x + " " + y)

case Array(0,\_\*) => println("Array: " + "0......")

case \_ => println("something else")

}

match\_array(Array(0))

match\_type(Array(0,1))

match\_array(Array(0,1,2,3,1))

这里根据传入的数组的长度以及数组中的元素进行匹配，例如*case Array(0,\_\*)=>…*这里使用了可变参数，它会匹配以0作为数组下标为0的元素，长度至少为1的数组。

输出结果：

Array: 0

Array: 1 0

Array: 0......

### 27.3 List模式匹配

与Array类似，但是由于list是由头尾组合而成，语法略有不同。

def match\_list(list :Any) = list match {

case 0::Nil => println("List:" +"0")

case x::y::Nil => println("List:" + x + " " + y)

case 0::tail => println("List:" + "0...")

case \_ => println("something else")

}

match\_list(List(0))

match\_list(List(3,0))

match\_list(List(0,1,2,3))

输出结果：

List:0

List:3 0

List:0...

### 27.4 Tuple模式匹配

def match\_tuple(tuple:Any) = tuple match{

case (0,\_) => println("Tuple:"+ 0)

case (x,\_) => println("Tuple:" + x)

case \_ => println("something else")

}

match\_tuple((0,"scala"))

match\_tuple((2,0))

match\_tuple((1,2,3,"string",Map("scala"->"spark")))

输出结果：

Tuple:0

Tuple:2

something else

## 28 scala中的提取器Extractor

### 28.1 Extractor

提取器（Extractor）是指从表达式中提取出需要的值。它是使得scala中模式匹配如此强大的基础。

def match\_array(arr :Any) = arr match {

case Array(0) => println("Array: " + 0)

case Array(x,y) => println("Array: " + x + " " + y)

case Array(0,\_\*) => println("Array: " + "0......")

case \_ => println("something else")

}

match\_array(Array(0))

match\_array(Array(1,0))

match\_array(Array(0,1,2,3,1))

val pattern = "([0-9]+) ([a-z]+)".r

"1112 helloScala" match {

case pattern(num,string) => println(num + " " + string)

}

其实，在进行模式匹配时，通过提取器将符合Array或是正则表达的变量中的值提取出来，复制给相应的变量，以便在case表达式中使用。

一般会调用相应的类型的unapply方法，例如Array的unapply方法。

/\*\* Called in a pattern match like `{ case Array(x,y,z) => println('3 elements')}`.

\*

\* @param x the selector value

\* @return sequence wrapped in a [[scala.Some]], if `x` is a Seq, otherwise `None`

\*/

def unapplySeq[T](x: Array[T]): Option[IndexedSeq[T]] =

if (x == null) None else Some(x.toIndexedSeq)

// !!! the null check should to be necessary, but without it 2241 fails. Seems to be a bug

// in pattern matcher. @PP: I noted in #4364 I think the behavior is correct.

}