# 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