



Scala核心编程

-模式匹配

尚硅谷研究院



- **match**

基本介绍

Scala中的模式匹配类似于Java中的switch语法，但是**更加强大**。

模式匹配语法中，采用**match**关键字声明，每个分支采用**case**关键字进行声明，当需要匹配时，会从第一个**case**分支开始，如果匹配成功，那么执行对应的逻辑代码，如果匹配不成功，继续执行下一个分支进行判断。如果**所有case都不匹配**，那么会执行**case _** 分支，类似于Java中default语句。



- **match**

应用案例

➤ Java Switch的简单回顾

```
// Java  
int i = 1;  
switch ( i ) {  
    case 0 :  
        break;  
    case 1 :  
        break;  
    default :  
        break  
}
```



- **match**

应用案例

➤ Scala的模式匹配

// 模式匹配，类似于Java的switch语法

```
val oper = '#'  
val n1 = 20  
val n2 = 10  
var res = 0  
oper match {  
  case '+' => res = n1 + n2  
  case '-' => res = n1 - n2  
  case '*' => res = n1 * n2  
  case '/' => res = n1 / n2  
  case _ => println("oper error")  
}  
println("res=" + res)
```

● match

match的细节和注意事项

- 1) 如果所有case都不匹配，那么会执行**case _** 分支，类似于Java中default语句
- 2) 如果所有case都不匹配，又没有写**case _** 分支，那么会抛出MatchError
- 3) 每个case中，不用break语句，自动中断case
- 4) 可以在match中使用其它类型，而不仅仅是字符

```
val oper = 1
val n1 = 20
val n2 = 10
var res = 0
oper match {
  case '+' => res = n1 + n2
  case 1 => res = n1 / n2
  case _ => println("oper error")
  println("res=" + res)
```



图片1.z

- 5) => 等价于 java switch 的 :
- 6) => 后面的代码块到下一个 case，是作为一个整体执行，可以使用{} 扩起来，也可以不扩。



● 守卫

基本介绍

如果想要表达**匹配某个范围的数据**，就需要在模式匹配中增加条件守卫

应用案例

```
for (ch <- "+-3!") {  
  var sign = 0  
  var digit = 0  
  ch match {  
    case '+' => sign = 1  
    case '-' => sign = -1  
    // 说明..  
    case _ if ch.toString.equals("3") => digit = 3  
    case _ => sign = 2  
  }  
  println(ch + " " + sign + " " + digit)  
}
```

- 守卫

课堂思考题

如看下面的代码，会输出什么？

```
for (ch <- "+-3!") {  
  var sign = 0  
  var digit = 0  
  ch match {  
    case '+' => sign = 1  
    case '-' => sign = -1  
    // 说明..  
    case _ => digit = 3  
    case _ => sign = 2  
  }  
  //  
  println(ch + " " + sign + " " + digit)  
}
```

```
for (ch <- "+-3!") {  
  var sign = 0  
  var digit = 0  
  ch match {  
    case _ => digit = 3  
    case '+' => sign = 1  
    case '-' => sign = -1  
    // 说明..  
  }  
  println(ch + " " + sign + " " + digit)  
}
```



- 模式中的变量

基本介绍

如果在case关键字后跟变量名，那么match前表达式的值会赋给那个变量

应用案例

```
val ch = 'V'
ch match {
  case '+' => println("ok~")
  case mychar => println("ok~" + mychar)
  case _ => println ("ok~~")
}
```


● 类型匹配

基本介绍

可以匹配对象的任意类型，这样做避免了使用isInstanceOf和asInstanceOf方法

应用案例

```
// 类型匹配, obj 可能有如下的类型
val a = 7
val obj = if(a == 1) 1
else if(a == 2) "2"
else if(a == 3) BigInt(3)
else if(a == 4) Map("aa" -> 1)
else if(a == 5) Map(1 -> "aa")
else if(a == 6) Array(1, 2, 3)
else if(a == 7) Array("aa", 1)
else if(a == 8) Array("aa")
```

```
val result = obj match {
  case a : Int => a
  case b : Map[String, Int] => "对象是一个字符串-数字的Map集合"
  case c : Map[Int, String] => "对象是一个数字-字符串的Map集合"
  case d : Array[String] => "对象是一个字符串数组"
  case e : Array[Int] => "对象是一个数字数组"
  case f : BigInt => Int.MaxValue
  case _ => "啥也不是"
}
println(result)
```



图片7.z



- 类型匹配

类型匹配注意事项

- 1) Map[String, Int] 和 Map[Int, String] 是两种不同的类型，其它类推。
- 2) 在进行类型匹配时，编译器会**预先检测是否有可能的匹配**，如果没有则报错。

```
val obj = 10
val result = obj match {
  case a : Int => a
  case b : Map[String, Int] => "Map集合"
  case _ => "啥也不是"
}
```



● 类型匹配

类型匹配注意事项

3) 一个说明:

```
val result = obj match {  
  case i : Int => i
```

} case i : Int => i 表示 将 **i = obj** (其它类推), 然后再判断类型

4) 如果 case _ 出现在match 中间, 则表示隐藏变量名, 即不使用, 而**不是表示默认匹配**。

// 类型匹配, obj 可能有如下的类型

```
val a = 7
```

```
val obj = if(a == 1) 1
```

```
else if(a == 2) "2"
```

```
else if(a == 3) BigInt(3)
```

```
else if(a == 4) Map("aa" -> 1)
```

```
else if(a == 5) Map(1 -> "aa")
```

```
else if(a == 6) Array(1, 2, 3)
```

```
else if(a == 7) Array("aa", 1)
```

```
else if(a == 8) Array("aa")
```

```
val result = obj match {
```

```
case a : Int => a
```

```
case _ : BigInt => Int.MaxValue //看这里!
```

```
case b : Map[String, Int] => "对象是一个字符串-数字的Map集合"
```

```
case c : Map[Int, String] => "对象是一个数字-字符串的Map集合"
```

```
case d : Array[String] => "对象是一个字符串数组"
```

```
case e : Array[Int] => "对象是一个数字数组"
```

```
case _ => "啥也不是"
```

```
}
```

```
println(result)
```

● 匹配数组

基本介绍

- 1) Array(0) 匹配只有一个元素且为0的数组。
- 2) Array(x,y) 匹配数组有两个元素，并将两个元素赋值为x和y。当然可以依次类推Array(x,y,z) 匹配数组有3个元素的等等....
- 3) Array(0,*) 匹配数组以0开始

应用案例

```
for (arr <- Array(Array(0), Array(1, 0), Array(0, 1, 0),  
Array(1, 1, 0), Array(1, 1, 0, 1))) {  
  val result = arr match {  
    case Array(0) => "0"  
    case Array(x, y) => x + "=" + y  
    case Array(0, _) => "以0开头和数组"  
    case _ => "什么集合都不是"  
  }  
  println("result = " + result)  
}
```



说明:

通过增加和删除for循环的数组，来看代码运行的结果,加强学员对匹配数组的理解



图片8.z



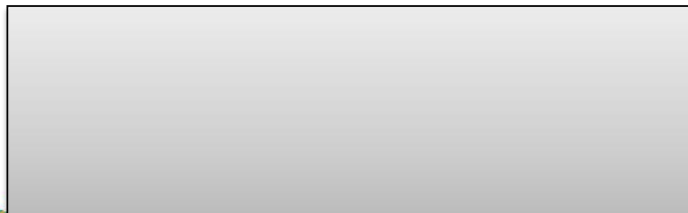
- 匹配列表

应用案例

```
for (list <- Array(List(0), List(1, 0), List(0, 0, 0), List(1, 0, 0))) {  
  val result = list match {  
    case 0 :: Nil => "0" //  
    case x :: y :: Nil => x + " " + y //  
    case 0 :: tail => "0 ..." //  
    case _ => "something else"  
  }  
  println(result)  
}
```

//案例演示+说明

//请思考，如果要匹配 **List(88)** 这样的只含有一个元素的列表,并原值返回.应该怎么写?





- 匹配元组

应用案例

```
// 元组匹配
// 元组匹配
for (pair <- Array((0, 1), (1, 0), (1, 1),(1,0,2))) {
  val result = pair match { //
    case (0, _) => "0 ..." //
    case (y, 0) => y //
    case _ => "other" //.
  }
  println(result)
}
```

//案例演示+说明

//思考，如果要匹配 (10, 30) 这样任意两个元素的**对偶元组**，应该如何写？

● 对象匹配

基本介绍

对象匹配，什么才算是匹配呢？，规则如下：

- 1) **case**中对象的**unapply**方法(对象提取器)返回**Some**集合则为匹配成功
- 2) 返回**none**集合则为匹配失败

应用案例1

```
object Square {  
  def unapply(z: Double): Option[Double] = Some(math.sqrt(z))  
  def apply(z: Double): Double = z * z  
}  
// 模式匹配使用：  
val number: Double = 36.0  
number match {  
  case Square(n) => println(n)  
  case _ => println("nothing matched")  
}[案例演示+代码说明+总结]
```

- 对象匹配

应用案例1的小结

- 1) 构建对象时`apply`会被调用，比如 `val n1 = Square(5)`
- 2) 当将 `Square(n)` 写在 `case` 后时`[case Square(n) => xxx]`，会默认调用 `unapply` 方法(对象提取器)
- 3) `number` 会被传递给`def unapply(z: Double)` 的 `z` 形参
- 4) 如果返回的是`Some`集合，则`unapply`提取器返回的结果会返回给 `n` 这个形参
- 5) `case`中对象的`unapply`方法(提取器)返回`some`集合则为匹配成功
- 6) 返回`none`集合则为匹配失败

- 对象匹配

应用案例2

```
object Names {  
  def unapplySeq(str: String): Option[Seq[String]] = {  
    if (str.contains(",")) Some(str.split(","))  
    else None  
  }  
}
```

【案例演示+代码说明+总结】

```
val namesString = "Alice,Bob,Thomas"  
//说明  
namesString match {  
  case Names(first, second, third) => {  
    println("the string contains three people's names")  
    // 打印字符串  
    println(s"$first $second $third")  
  }  
  case _ => println("nothing matched")  
}
```

- 对象匹配

应用案例2的小结

- 1) 当case 后面的对象提取器方法的参数为多个，则会默认调用def unapplySeq() 方法
- 2) 如果unapplySeq返回是**Some**，获取其中的值,判断得到的**sequence**中的元素的个数是否是三个,如果是三个，则把三个元素分别取出，赋值给**first**，**second**和**third**
- 3) 其它的规则不变.



- 变量声明中的模式

基本介绍

match中每一个case都可以单独提取出来，意思是一样的。

应用案例

```
val (x, y) = (1, 2)
val (q, r) = BigInt(10) /% 3 //说明 q = BigInt(10) / 3 r = BigInt(10) % 3
val arr = Array(1, 7, 2, 9)
val Array(first, second, _) = arr // 提出arr的前两个元素
println(first, second)
//案例演示+说明
```

- **for**表达式中的模式

基本介绍

for循环也可以进行模式匹配.

应用案例

```
val map = Map("A"->1, "B"->0, "C"->3)
for ( (k, v) <- map ) {
  println(k + " -> " + v)
}
//说明
for ((k, 0) <- map) {
  println(k + " --> " + 0)
}
//说明
for ((k, v) <- map if v == 0) {
  println(k + " ---> " + v)
}
```



- 样例类

样例类快速入门

abstract class Amount

case class Dollar(value: Double) extends Amount

case class Currency(value: Double, unit: String) extends Amount

case object NoAmount extends Amount

说明: 这里的 Dollar, Currency, NoAmount 是样例类。

案例演示+反编译

- 样例类

基本介绍

- 1) 样例类仍然是类
- 2) 样例类用**case**关键字进行声明。
- 3) 样例类是为**模式匹配而优化**的类
- 4) 构造器中的每一个参数都成为**val**——除非它被显式地声明为**var**（不建议这样做）
- 5) 在样例类对应的伴生对象中**提供apply方法**让你不用**new**关键字就能构造出相应的对象
- 6) **提供unapply方法**让模式匹配可以工作
- 7) 将**自动生成toString、equals、hashCode和copy方法**(有点类似**模板类**，直接给生成，供程序员使用)
- 8) 除上述外，样例类和其他类完全一样。你可以添加方法和字段，扩展它们



- 样例类

样例类最佳实践1:

当我们有一个类型为Amount的对象时，可以用模式匹配来匹配他的类型，并将属性值绑定到变量(即：**把样例类对象的属性值提取到某个变量,该功能有用**)

```
for (amt <- Array(Dollar(1000.0), Currency(1000.0, "RMB"), NoAmount)) {  
  val result = amt match {  
    //说明  
    case Dollar(v) => "$" + v  
    //说明  
    case Currency(v, u) => v + " " + u  
    case NoAmount => ""  
  }  
  println(amt + ": " + result)  
}
```

【案例演示+代码说明】



- 样例类

样例类最佳实践2:

样例类的**copy**方法和带名参数

copy创建一个与现有对象值**相同的新对象**，并可以通过**带名参数**来修改某些属性。

```
val amt = Currency(29.95, "RMB")
val amt1 = amt.copy() //创建了一个新的对象，但是属性值一样
val amt2 = amt.copy(value = 19.95) //创建了一个新对象，但是修改了货币单位
val amt3 = amt.copy(unit = "英镑")//..
println(amt)
println(amt2)
println(amt3)
//案例演示+说明
```

- **case**语句的中置(缀)表达式

基本介绍

什么是中置表达式？**1 + 2**，这就是一个中置表达式。如果**unapply**方法产生一个元组，你可以在**case**语句中使用中置表示法。比如可以匹配一个**List**序列

应用实例

```
List(1, 3, 5, 9) match { //修改并测试
//1.两个元素间::叫中置表达式,至少first, second两个匹配才行.
//2.first 匹配第一个 second 匹配第二个, rest 匹配剩余部分(5,9)
case first :: second :: rest => println(first + second + rest.length) //
case _ => println("匹配不到...")
}
```

- 匹配嵌套结构

基本介绍

操作原理类似于正则表达式

最佳实践案例-商品捆绑打折出售

现在有一些商品，请使用**Scala**设计相关的样例类，完成商品捆绑打折出售。要求

- 1) 商品捆绑可以是单个商品，也可以是多个商品。
- 2) 打折时按照折扣x元进行设计。
- 3) 能够统计出所有捆绑商品打折后的最终价格

- 匹配嵌套结构

- 创建样例类

```
abstract class Item // 项
```

```
case class Book(description: String, price: Double) extends Item
```

```
//Bundle 捆 , discount: Double 折扣 , item: Item* ,
```

```
case class Bundle(description: String, discount: Double, item: Item*) extends Item
```

- 匹配嵌套结构(就是**Bundle**的对象)

```
//给出案例表示有一捆数, 单本漫画 (40-10) +文学作品(两本书) (80+30-20)  
= 30 + 90 = 120.0
```

```
val sale = Bundle("书籍", 10, Book("漫画", 40), Bundle("文学作品", 20, Book("《  
《阳关》", 80), Book("《围城》", 30)))
```

- 匹配嵌套结构

- 知识点1-将descr绑定到第一个Book的描述

请思考：如何取出

```
val sale = Bundle("书籍", 10, Book("漫画", 40), Bundle("文学作品", 20, Book("《阳关》", 80), Book("《围城》", 30)))
```

这个嵌套结构中的 "漫画"

同学们想一想

```
val res = sale match {  
  //如果我们进行对象匹配时，不想接受某些值，则使用_ 忽略即可，_* 表示所有  
  case Bundle(_, _, Book(desc, _), _*) => desc  
}
```




● 匹配嵌套结构

➤ 知识点2-通过@表示法将嵌套的值绑定到变量。_*绑定剩余Item到rest

请思考：如何将 "漫画" 和

```
val sale = Bundle("书籍", 10, Book("漫画", 40), Bundle("文学作品", 20,  
Book("《阳关》", 80), Book("《围城》", 30)))
```

这个嵌套结构中的 "漫画" 和 紫色的部分 绑定到变量，即赋值到变量中。

```
val result2 = sale match {  
  case Bundle(_, _, art @ Book(_, _), rest @ _*) => (art, rest)  
}  
println(result2)  
println("art =" + result2._1)  
println("rest=" + result2._2)
```



- 匹配嵌套结构

- 知识点3-不使用_*绑定剩余Item到rest

请思考：如何将 "漫画" 和 紫色部分

```
val sale = Bundle("书籍", 10, Article("漫画", 40), Bundle("文学作品", 20, Article("《阳关》", 80), Article("《围城》", 30)))
```

这个嵌套结构中的 "漫画" 和 紫色的部分 绑定到变量，即赋值到变量中。

```
val result2 = sale match {  
  //说明因为没有使用_* 即明确说明没有多个Bundle,所以返回的rest, 就不是WrappedArray了。  
  case Bundle(_, _, art @ Book(_, _), rest) => (art, rest)  
}  
println(result2)  
println("art =" + result2._1)  
println("rest=" + result2._2)
```



● 匹配嵌套结构

最佳实践案例-商品捆绑打折出售

现在有一些商品，请使用**Scala**设计相关的样例类，完成商品可以捆绑打折出售。
要求

- 1) 商品捆绑可以是单个商品，也可以是多个商品。
- 2) 打折时按照折扣xx元进行设计。
- 3) 能够统计出所有捆绑商品打折后的最终价格

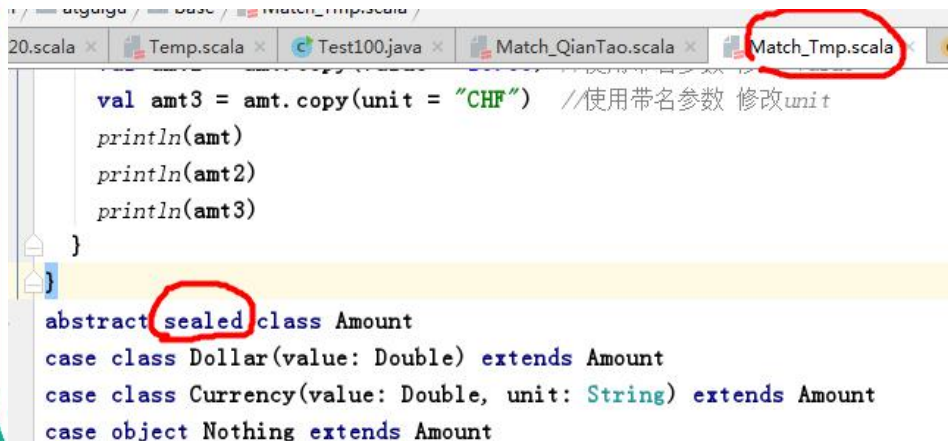
```
def price(it: Item): Double = {  
  it match {  
    case Book(_, p) => p  
    //生成一个新的集合,_是将its中每个循环的元素传递到price中it中。递归操作,分析一个简单的流程  
    case Bundle(_, disc, its @ _) => its.map(price _).sum - disc  
  }  
}
```

● 密封类

基本介绍

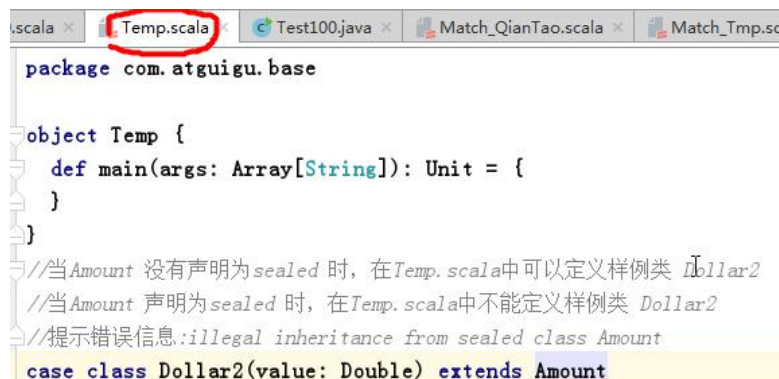
- 1) 如果能让case类的所有子类都必须在申明该类的**相同的源文件中定义**，可以将样例类的通用超类声明为**sealed**，这个超类称之为密封类。
- 2) 密封就是不能在其他文件中定义子类。

案例演示



```
val amt3 = amt.copy(unit = "CHF") //使用带名参数 修改unit
println(amt)
println(amt2)
println(amt3)
}

abstract sealed class Amount
case class Dollar(value: Double) extends Amount
case class Currency(value: Double, unit: String) extends Amount
case object Nothing extends Amount
```



```
package com.atguigu.base

object Temp {
  def main(args: Array[String]): Unit = {
  }
}

//当Amount 没有声明为sealed 时, 在Temp.scala中可以定义样例类 Dollar2
//当Amount 声明为sealed 时, 在Temp.scala中不能定义样例类 Dollar2
//提示信息:illegal inheritance from sealed class Amount
case class Dollar2(value: Double) extends Amount
```



图片9.z



谢谢！ 欢迎收看！