Scala第十一章节

章节目标

1. 掌握模式匹配相关内容
2. 掌握option类型及偏函数的用法
3. 掌握异常处理的用法
4. 理解正则表达式的运用
5. 理解提取器的用法
6. 掌握随机职业案例

---

1. 模式匹配

Scala中有一个非常强大的模式匹配机制，应用也非常广泛, 例如:

- 判断固定值
- 类型查询
- 快速获取数据

1.1 简单模式匹配

一个模式匹配包含了一系列备选项，每个备选项都开始于关键字 **case**。且每个备选项都包含了一个模式及一到多个表达式。箭头符号 **=>** 隔开了模式和表达式。

格式

```
变量 match {
    case "常量1" => 表达式1
    case "常量2" => 表达式2
    case "常量3" => 表达式3
    case _ => 表达式4       // 默认匹配项
}
```

执行流程

1. 先执行第一个case, 看 `变量值` 和 `该case对应的常量值` 是否一致.
2. 如果一致, 则执行该case对应的表达式.
3. 如果不一致, 则往后执行下一个case, 看 `变量值` 和 `该case对应的常量值` 是否一致.
4. 以此类推, 如果所有的case都不匹配, 则执行 `case _` 对应的表达式.

需求

1. 提示用户录入一个单词并接收.
2. 判断该单词是否能够匹配以下单词，如果能匹配，返回一句话
3. 打印结果.

单词	返回
hadoop	大数据分布式存储和计算框架
zookeeper	大数据分布式协调服务框架
spark	大数据分布式内存计算框架
未匹配	未匹配

参考代码

```
import scala.io.StdIn

//案例: 模式匹配之简单匹配
object ClassDemo01 {
  def main(args: Array[String]): Unit = {
    //1. 提示用户录入字符串并接收.
    println("请录入一个字符串: ")
    var str = StdIn.readLine()

    //2. 判断字符串是否是指定的内容, 并接收结果.
    val result = str match {
      case "hadoop" => "大数据分布式存储和计算框架"
      case "zookeeper" => "大数据分布式协调服务框架"
      case "spark" => "大数据分布式内存计算框架"
      case _ => "未匹配"
    }
    //3. 打印结果.
    println(result)
    println("-" * 15) //分割线.

    //简写形式
    str match {
      case "hadoop" => println("大数据分布式存储和计算框架")
      case "zookeeper" => println("大数据分布式协调服务框架")
      case "spark" => println("大数据分布式内存计算框架")
      case _ => println("未匹配")
    }
  }
}
```

1.2 匹配类型

除了匹配数据之外, match表达式还可以进行类型匹配。如果我们要根据不同的数据类型，来执行不同的逻辑，也可以使用match表达式来实现。

格式

```
对象名 match {
    case 变量名1: 类型1 => 表达式1
    case 变量名2: 类型2 => 表达式2
    case 变量名3: 类型3 => 表达式3
    ...
    case _ => 表达式4
}
```

> 注意: 如果case表达式中无需使用到匹配到的变量，可以使用下划线代代替

需求

1. 定义一个变量为Any类型，然后分别给其赋值为"hadoop"、1、1.0
2. 定义模式匹配，然后分别打印类型的名称

参考代码

```
//案例: 模式匹配之匹配类型
object ClassDemo02 {
  def main(args: Array[String]): Unit = {
    //1. 定义一个变量为Any类型，然后分别给其赋值为"hadoop"、1、1.0
    val a:Any = 1.0
    //2. 定义模式匹配，然后分别打印类型的名称
    val result = a match {
      case x:String => s"${x} 是String类型的数据"
      case x:Double => s"${x} 是Double类型的数据"
      case x:Int => s"${x} 是Int类型的数据"
      case _ => "未匹配"
    }
    //3. 打印结果
    println(result)

    //4. 优化版，如果在case校验的时候，变量没有被使用，则可以用_替代.
    val result2 = a match {
      case _:String => "String"
      case _:Double => "Double"
      case _:Int => "Int"
      case _ => "未匹配"
    }
    //打印结果
    println(result2)
  }
}
```

1.3 守卫

所谓的守卫指的是`在case语句中添加if条件判断`，这样可以让我们的代码更简洁, 更优雅.

格式

```
变量 match {
    case 变量名 if条件1 => 表达式1
    case 变量名 if条件2 => 表达式2
    case 变量名 if条件3 => 表达式3
    ...
    case _ => 表达式4
}
```

需求

1. 从控制台读入一个数字a（使用StdIn.readInt）
2. 如果 a >= 0 而且 a <= 3，打印[0-3]
3. 如果 a >= 4 而且 a <= 8，打印[4,8]
4. 否则，打印未匹配

参考代码

```
//案例：模式匹配之守卫
object ClassDemo03 {
  def main(args: Array[String]): Unit = {
    //1. 从控制台读入一个数字a（使用StdIn.readInt）
    println("请录入一个整数：")
    var num = StdIn.readInt()
    //2. 模式匹配
    num match {
      //2.1 如果 a >= 0 而且 a <= 3，打印[0-3]
      case a if a >= 0 && a <= 3 => println("[0-3]")
      //2.2 如果 a >= 4 而且 a <= 8，打印[4,8]
      case a if a >= 4 && a <= 8 => println("[4-8]")
      //2.3 否则，打印未匹配
      case _ => println("未匹配")
    }
  }
}
```

1.4 匹配样例类

Scala中可以使用模式匹配来匹配样例类，从而实现可以快速获取样例类中的成员数据。后续，我们在开发Akka案例时，还会经常用到。

格式

```
对象名 match {
    case 样例类型1(字段1，字段2，字段n) => 表达式1
    case 样例类型2(字段1，字段2，字段n) => 表达式2
    case 样例类型3(字段1，字段2，字段n) => 表达式3
    ...
    case _ => 表达式4
}
```

> 注意:

> 1. 样例类型后的小括号中, 编写的字段个数要和该样例类的字段个数保持一致.
> 2. 通过match进行模式匹配的时候, 要匹配的对象必须声明为: Any类型.

需求

1. 创建两个样例类Customer(包含姓名, 年龄字段), Order(包含id字段)
2. 分别定义两个样例类的对象，并指定为Any类型
3. 使用模式匹配这两个对象，并分别打印它们的成员变量值

参考代码

```
//案例: 模式匹配之匹配样例类
object ClassDemo04 {

  //1. 创建两个样例类Customer、Order
  //1.1 Customer包含姓名、年龄字段
  case class Customer(var name: String, var age: Int)

  //1.2 Order包含id字段
  case class Order(id: Int)

  def main(args: Array[String]): Unit = {
    //2. 分别定义两个案例类的对象，并指定为Any类型
    val c: Any = Customer("糖糖", 73)
    val o: Any = Order(123)
    val arr: Any = Array(0, 1)

    //3. 使用模式匹配这两个对象，并分别打印它们的成员变量值
    c match {
      case Customer(a, b) => println(s"Customer类型的对象，name=${a}, age=${b}")
      case Order(c) => println(s"Order类型, id=${c}")
      case _ => println("未匹配")
    }
  }
}
```

1.5 匹配集合

除了上述功能之外, Scala中的模式匹配，还能用来匹配数组, 元组, 集合(列表, 集, 映射)等。

1.5.1 示例一: 匹配数组

需求

1. 依次修改代码定义以下三个数组

```
Array(1,x,y)    // 以1开头，后续的两个元素不固定
Array(0)        // 只匹配一个0元素的元素
Array(0, ...)   // 可以任意数量，但是以0开头
```

2. 使用模式匹配, 匹配上述数组.

参考代码

```
//案例：模式匹配之匹配数组
object ClassDemo05 {
  def main(args: Array[String]): Unit = {
    //1. 定义三个数组.
    val arr1 = Array(1, 2, 3)
    val arr2 = Array(0)
    val arr3 = Array(1, 2, 3, 4, 5)
    //2. 通过模式匹配，找到指定的数组.
    arr2 match {
      //匹配：长度为3，首元素为1，后两个元素无所谓.
      case Array(1, x, y) => println(s"匹配长度为3，首元素为1，后两个元素是：${x}，${y}")
      //匹配：只有一个0元素的数组
      case Array(0) => println("匹配：只有一个0元素的数组")
      //匹配：第一个元素是1，后边元素无所谓的数组.
      case Array(1, _*) => println("匹配：第一个元素是1，后边元素无所谓的数组")
      //其他校验项
      case _ => println("未匹配")
    }
  }
}
```

1.5.2 示例二: 匹配列表

需求

1. 依次修改代码定义以下三个列表

```
List(0)             // 只保存0一个元素的列表
List(0,...)         // 以0开头的列表，数量不固定
List(x,y)           // 只包含两个元素的列表
```

2. 使用模式匹配, 匹配上述列表.

参考代码

```
//案例：模式匹配之匹配列表
object ClassDemo06 {
  def main(args: Array[String]): Unit = {
    //1. 定义列表.
    var list1 = List(0)
    var list2 = List(0, 1, 2, 3, 4, 5)
    var list3 = List(1, 2)

    //2. 通过match进行模式匹配
    //思路一：通过List()来实现.
    list1 match {
      case List(0) => println("匹配：只有一个0元素的列表")
      case List(0, _*) => println("匹配：0开头，后边元素无所谓的列表")
      case List(x, y) => println(s"匹配：只有两个元素的列表，元素为：${x}，${y}")
      case _ => println("未匹配")
    }
```

```
    //思路二：采用关键字优化  Nil, tail
    list1 match {
      case 0 :: Nil => println("匹配：只有一个0元素的列表")
      case 0 :: tail => println("匹配：0开头，后边元素无所谓的列表")
      case x :: y :: Nil => println(s"匹配：只有两个元素的列表，元素为：${x}，${y}")
      case _ => println("未匹配")
    }
  }
}
```

1.5.3 案例三: 匹配元组

需求

1. 依次修改代码定义以下两个元组

```
(1, x, y)   // 以1开头的、一共三个元素的元组
(x, y, 5)   // 一共有三个元素，最后一个元素为5的元组
```

2. 使用模式匹配, 匹配上述元组.

参考代码

```
//案例：模式匹配之匹配元组
object ClassDemo07 {
  def main(args: Array[String]): Unit = {
    //1. 定义两个元组.
    val a = (1, 2, 3)
    val b = (3, 4, 5)
    val c = (3, 4)

    //2. 通过模式匹配，匹配指定的元素
    a match {
      case (1, x, y) => println(s"匹配：长度为3，以1开头，后两个元素无所谓的元组，这里后两个元素是：${x}，${y}")
      case (x, y, 5) => println(s"匹配：长度为3，以5结尾，前两个元素无所谓的元素，这里前两个元素是：${x}，${y}")
      case _ => println("未匹配")
    }
  }
}
```

1.6 变量声明中的模式匹配

在定义变量时，可以使用模式匹配快速获取数据. 例如: `快速从数组,列表中获取数据`.

需求

1. 生成包含0-10数字的数组，使用模式匹配分别获取第二个、第三个、第四个元素

2. 生成包含0-10数字的列表，使用模式匹配分别获取第一个、第二个元素

参考代码

```
//案例: 演示变量声明中的模式匹配.
object ClassDemo08 {
  def main(args: Array[String]): Unit = {
    //1. 生成包含0-10数字的数组，使用模式匹配分别获取第二个、第三个、第四个元素
    //1.1 生成包含0-10数字的数组
    val arr = (0 to 10).toArray
    //1.2 使用模式匹配分别获取第二个、第三个、第四个元素
    val Array(_, x, y, z, _*) = arr;
    //1.3 打印结果.
    println(x, y, z)
    println("-" * 15)

    //2. 生成包含0-10数字的列表，使用模式匹配分别获取第一个、第二个元素
    //2.1 生成包含0-10数字的列表,
    val list = (0 to 10).toList
    //2.2 使用模式匹配分别获取第一个、第二个元素
    //思路一: List() 实现
    val List(a, b, _*) = list
    //思路二: ::, tail 实现.
    val c :: d :: tail = list
    //2.3 打印结果.
    println(a, b)
    println(c, d)
  }
}
```

1.7 匹配for表达式

Scala中还可以使用模式匹配来匹配for表达式，从而实现快速获取指定数据, 让我们的代码看起来更简洁, 更优雅.

需求

1. 定义变量记录学生的姓名和年龄, 例如: "张三" -> 23, "李四" -> 24, "王五" -> 23, "赵六" -> 26
2. 获取所有年龄为23的学生信息, 并打印结果.

参考代码

```
//案例: 模式匹配之匹配for表达式.
object ClassDemo09 {
  def main(args: Array[String]): Unit = {
    //1. 定义变量记录学生的姓名和年龄.
    val map1 = Map("张三" -> 23, "李四" -> 24, "王五" -> 23, "赵六" -> 26)
    //2. 获取所有年龄为23的学生信息.
    //2.1 格式一: 通过if语句实现.
    for((k,v) <- map1 if v == 23) println(s"${k} = ${v}")
    //分割线.
    println("-" * 15)
    //2.2 格式二: 通过固定值实现.
```

```
    for((k, 23) <- map1) println(k + " = 23")
  }
}
```

2. Option类型

2.1 概述

实际开发中, 在返回一些数据时, 难免会遇到空指针异常(NullPointerException), 遇到一次就处理一次相对来讲还是比较繁琐的. 在Scala中, 我们返回某些数据时，可以返回一个Option类型的对象来封装具体的数据，从而实现有效的避免空指针异常。

2.2 格式

Scala中，Option类型表示可选值。这种类型的数据有两种形式：

- Some(x)：表示实际的值

```
final case class Some[+A](x: A) extends Option[A] {
  def isEmpty : Boolean  = false
  def get : A  = x
}
```

- None：表示没有值

```
case object None extends Option[Nothing] {
  def isEmpty : Boolean  = true
  def get : Nothing  = throw new NoSuchElementException("None.get")
}
```

> 注意: 使用getOrElse方法，当值为None时可以指定一个默认值.

2.3 示例

需求

1. 定义一个两个数相除的方法，使用Option类型来封装结果
2. 打印结果
   - 不是除零，打印结果
   - 除零, 则打印异常错误

参考代码

```
//案例: 演示Option类型
object ClassDemo10 {
  //1. 定义一个两个数相除的方法，使用Option类型来封装结果
  def div(a:Int, b:Int) = {
    if (b == 0) {
      None      //除数为0，没有结果.
    } else {
      Some(a / b) //除数不为0，返回具体的结果.
    }
```

```
  }

  def main(args: Array[String]): Unit = {
    //2. 然后使用模式匹配来打印结果
    val result = div(10 , 0)
    //思路一: 通过模式匹配来打印结果.
    result match {
    //不是除零，打印结果
      case Some(x) => println(x)
      //除零打印异常错误
      case None => println("除数不能为0")
    }
    println("-" * 15)

    //思路二: 采用getOrElse()方法实现.
    println(result.getOrElse(0))
  }
}
```

3. 偏函数

3.1 定义

偏函数提供了更简洁的语法，可以简化函数的定义。配合集合的函数式编程，可以让代码更加优雅。

所谓的偏函数是指`被包在花括号内没有match的一组case语句`，偏函数是PartialFunction[A, B]类型的的一个实例对象, 其中A代表输入参数类型, B代表返回结果类型.

3.2 格式

```
val 对象名 = {     //这对大括号及其内部的一组case语句，就组成了一个偏函数.
    case 值1 => 表达式1
    case 值2 => 表达式2
    case 值3 => 表达式3
    ...
}
```

3.3 示例一: 入门案例

需求

定义一个偏函数，根据以下方式返回

输入	返回值
1	一
2	二
3	三
其他	其他

参考代码

```
//案例：演示偏函数
object ClassDemo11 {
  def main(args: Array[String]): Unit = {
    //1. 定义一个偏函数，根据指定格式返回
    val pf:PartialFunction[Int,String] = {
      case 1 => "一"
      case 2 => "二"
      case 3 => "三"
      case _ => "其他"
    }

    //2. 调用方法
    println(pf(1))
    println(pf(2))
    println(pf(3))
    println(pf(4))
  }
}
```

3.4 示例二: 结合map函数使用

需求

1. 定义一个列表，包含1-10的数字
2. 请将1-3的数字都转换为[1-3]
3. 请将4-8的数字都转换为[4-8]
4. 将其他的数字转换为(8-*]
5. 打印结果.

参考代码

```
//案例：偏函数使用，结合map函数
object ClassDemo12{
  def main(args: Array[String]): Unit = {
    //1. 定义一个列表，包含1-10的数字
    val list1 = (1 to 10).toList
    //核心：通过偏函数结合map使用，来进行模式匹配
    val list2 = list1.map {
      //2 请将1-3的数字都转换为[1-3]
      case x if x >= 1 && x <= 3 => "[1-3]"
```

```
        //3 请将4-8的数字都转换为[4-8]
        case x if x >= 4 && x <= 8 => "[4-8]"
        //4 将其他的数字转换为(8-*]
        case _ => "(8-*]"
      }

      //5. 打印结果.
      println(list2)
    }
}
```

4. 正则表达式

4.1 概述

所谓的正则表达式指的是`正确的，符合特定规则的式子`，它是一门独立的语言, 并且能被兼容到绝大多数的编程语言中. 在scala中, 可以很方便地使用正则表达式来匹配数据。具体如下:

1. Scala中提供了`Regex类`来定义正则表达式.
2. 要构造一个Regex对象，直接使用`String类的r方法`即可.
3. 建议使用三个双引号来表示正则表达式，不然就得对正则中的反斜杠进行转义.

4.2 格式

```
val 正则对象名 = """具体的正则表达式""".r
```

> 注意: 使用findAllMatchIn方法可以获取到所有正则匹配到的数据(字符串).

4.3 示例一: 校验邮箱是否合法

需求

1. 定义一个字符串, 表示邮箱.
2. 定义一个正则表达式，来匹配邮箱是否合法.
3. 合法邮箱测试：qq12344@163.com
4. 不合法邮箱测试：qq12344@.com
5. 打印结果.

参考代码

```
//案例: 校验邮箱是否合法.
object ClassDemo13 {
  def main(args: Array[String]): Unit = {
    //需求: 定义一个正则表达式，来匹配邮箱是否合法
    //1. 定义一个字符串，表示邮箱.
    val email = "qq12344@163.com"
    //2. 定义一个正则表达式，用来校验邮箱.
    /*
        .   表示任意字符
        +   数量词，表示前边的字符出现至少1次，至多无所谓.
```

```
        @  表示必须是@符号，无特殊含义.
        \. 因为.在正则中有特殊的含义，所以要转移一下，使它变成普通的.
     */
    val regex = """.+@.+\..+""".r
    //3. 打印结果.
    if(regex.findAllMatchIn(email).size != 0) {
      //合法邮箱
      println(s"${email} 是一个合法的邮箱!")
    } else {
      println(s"${email} 是一个非法的邮箱!")
    }
  }
}
```

4.4 示例二: 过滤所有不合法邮箱

需求

1. 找出以下列表中的所有不合法的邮箱.
2. "38123845@qq.com", "a1da88123f@gmail.com", "zhansan@163.com", "123afadff.com"

参考代码

```
//案例: 过滤所有不合法的邮箱.
object ClassDemo14 {
  def main(args: Array[String]): Unit = {
    //1. 定义列表，记录邮箱.
    val emlList = List("38123845@qq.com", "a1da88123f@gmail.com", "zhansan@163.com",
"123afadff.com")
    //2. 定义正则表达式.
    val regex = """.+@.+\..+""".r
    //3. 通过 过滤器 获取所有的不合法的邮箱.
    val list = emlList.filter(x => regex.findAllMatchIn(x).size == 0)
    //4. 打印结果.
    println(list)
  }
}
```

4.5 示例三: 获取邮箱运营商

需求

1. 定义列表, 记录以下邮箱:

   ```
   "38123845@qq.com", "a1da88123f@gmail.com", "zhansan@163.com", "123afadff.com"
   ```

2. 使用正则表达式进行模式匹配，匹配出来邮箱运营商的名字。

   > 例如:
   >
   > 邮箱zhansan@163.com，需要将163(运营商的名字)匹配出来.
   >
   > 提示:

> 1. 使用括号来匹配分组.
> 2. 打印匹配到的邮箱以及运营商.

参考代码

```
//案例: 获取邮箱运营商.
object ClassDemo15 {
  def main(args: Array[String]): Unit = {
    //1. 定义列表, 记录邮箱.
    val emlList = List("38123845@qq.com", "a1da88123f@gmail.com", "zhansan@163.com",
"123afadff.com")
    //2. 定义正则表达式.
    val regex = """.+@(.+)\..+""".r
    //3. 根据 模式匹配 匹配出所有合法的邮箱及其对应的运营商.
    val result = emlList.map {
      //email就是emlList这个列表中的每一个元素.
      //company表示: 正则表达式中你用()括起来的内容, 也就是分组的数据.
      case email @ regex(company) => email -> s"${company}"
      case email => email -> "未匹配"
    }
    //4. 打印结果
    println(result)
  }
}
```

5. 异常处理

5.1 概述

来看看下面这一段代码:

```
def main(args: Array[String]): Unit = {
  val i = 10 / 0

  println("你好! ")
}

Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ForDemo$.main(ForDemo.scala:3)
    at ForDemo.main(ForDemo.scala)
```

执行程序，可以看到scala抛出

了异常，而且没有打印出来"你好! "。说明程序出现错误后就终止了。

那怎么解决该问题呢?

在Scala中，可以使用异常处理来解决这个问题. 而异常处理又分为两种方式:

- 方式一: 捕获异常.

> 注意: 该方式处理完异常后, 程序会继续执行.

- 方式二: 抛出异常.

  > 注意: 该方式处理完异常后, 程序会终止执行.

5.2 捕获异常

格式

```
try {
    //可能会出现问题的代码
}
catch{
    case ex:异常类型1 => //代码
    case ex:异常类型2 => //代码
}
finally {
    //代码
}
```

> 解释:
>
> 1. try中的代码是我们编写的业务处理代码.
> 2. 在catch中表示当出现某个异常时，需要执行的代码.
> 3. 在finally中，写的是不管是否出现异常都会执行的代码.

5.3 抛出异常

我们也可以在一个方法中，抛出异常。格式如下:

格式

```
throw new Exception("这里写异常的描述信息")
```

5.4 示例

需求

1. 通过try.catch来处理 除数为零异常.
2. 在main方法中抛出一个异常.

参考代码

```
//案例: 演示异常处理.
object ClassDemo16 {
  def main(args: Array[String]): Unit = {
    //1. 通过try.catch来处理 除数为零异常.
    try {
      //可能出问题的代码
      val i = 10 / 0
    } catch {
      //出现问题后的解决方案.
      //case ex:Exception => println("代码出问题了!")
      case ex:Exception => ex.printStackTrace()
```

```
    }
    println("你好! ")
    println("-" * 15)        //我是分割线.

    //2. 抛出一个异常对象.
    throw new Exception("我是一个Bug!")
    println("Hello, Scala!")    //这行代码并不会被执行.
  }
}
```

6. 提取器(Extractor)

6.1 概述

我们之前已经使用过Scala中非常强大的模式匹配功能了，通过模式匹配，我们可以快速获取样例类对象中的成员变量值。例如：

```
// 1. 创建两个样例类
case class Person(name:String, age:Int)
case class Order(id:String)

def main(args: Array[String]): Unit = {
    // 2. 创建样例类对象，并赋值为Any类型
    val zhangsan:Any = Person("张三", 20)
    val order1:Any = Order("001")

    // 3. 使用match...case表达式来进行模式匹配
    // 获取样例类中成员变量
    order1 match {
        case Person(name, age) => println(s"姓名: ${name} 年龄: ${age}")
        case Order(id1) => println(s"ID为: ${id1}")
        case _ => println("未匹配")
    }
}
```

那是不是所有的类都可以进行这样的模式匹配呢？答案是：

`不是`。一个类要想支持模式匹配，则必须要实现一个**提取器**。

> 注意:
>
> 1. 提取器指的就是`unapply()方法`.
> 2. 样例类自动实现了apply()、unapply()方法, 无需我们手动定义.

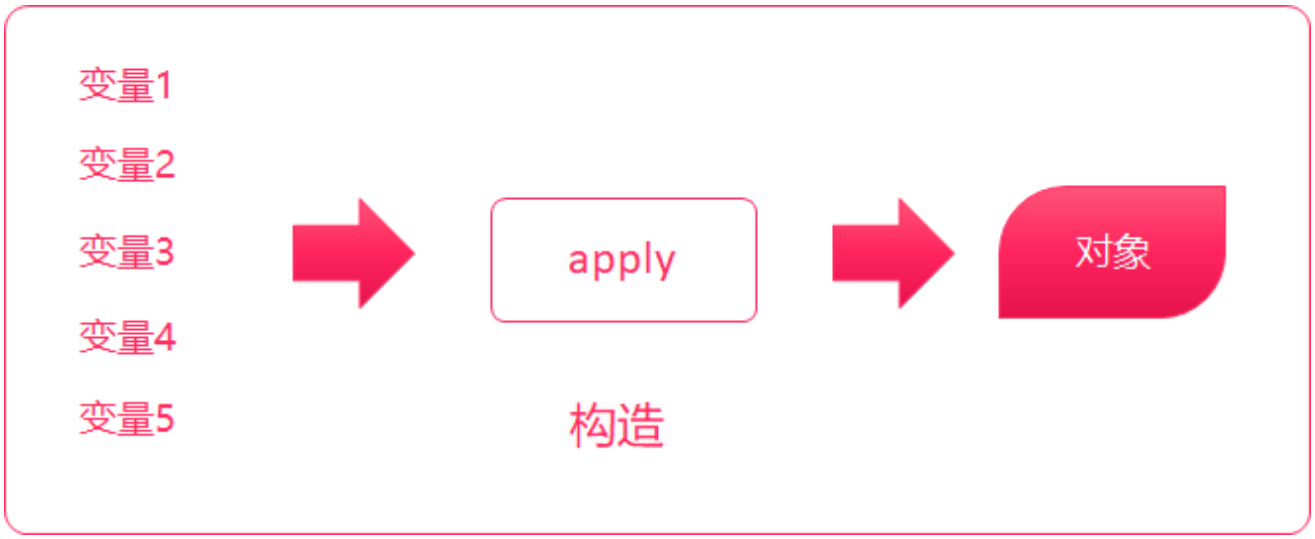
6.2 格式

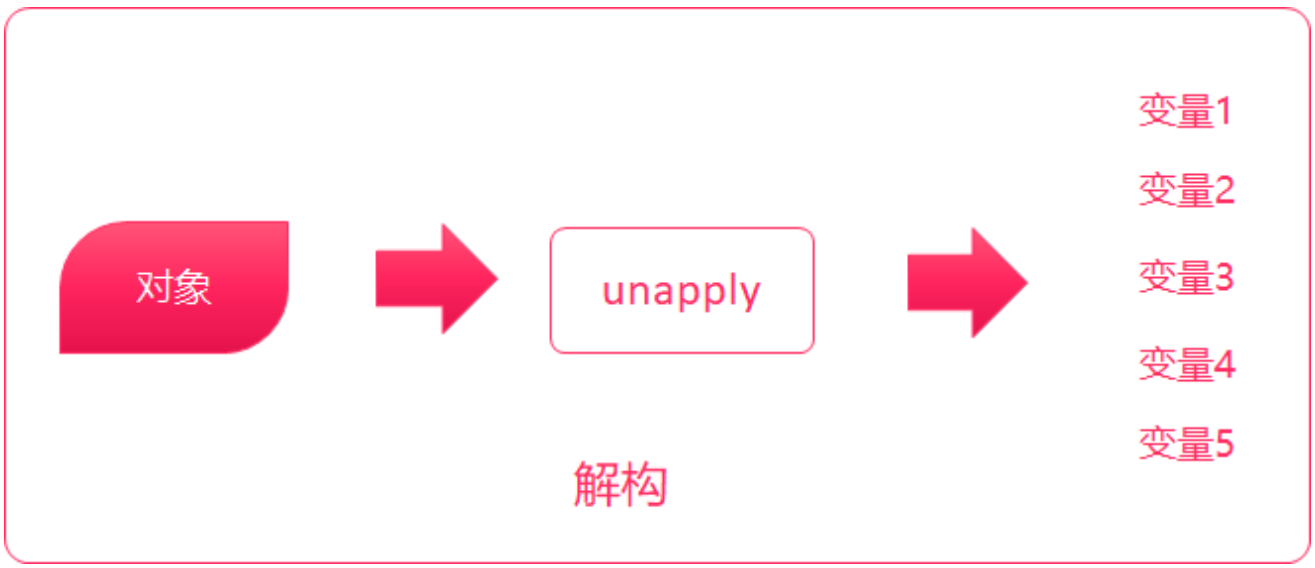
要实现一个类的提取器，只需要在该类的伴生对象中实现一个unapply方法即可。

语法格式

```
def unapply(stu:Student):Option[(类型1, 类型2, 类型3...)] = {
    if(stu != null) {
        Some((变量1, 变量2, 变量3...))
    }
    else {
        None
    }
}
```

图解





6.3 示例

需求

1. 创建一个Student类，包含姓名年龄两个字段
2. 实现一个类的提取器，并使用match表达式进行模式匹配，提取类中的字段。

参考代码

```
//案例: 演示Scala中的提取器.
//所谓的提取器就是: 在类的伴生对象中, 重写一个unapply()方法即可.
```

```
object ClassDemo17 {
  //1. 创建一个Student类，包含姓名年龄两个字段
  class Student(var name:String, var age:Int)

  //2. 实现一个类的提取器，并使用match表达式进行模式匹配，提取类中的字段。
  object Student {      //伴生对象.
    def apply(name:String, age:Int) = new Student(name, age)      //免new

    def unapply(s: Student): Option[(String, Int)] = {        //相当于把对象 拆解成 其各个
属性.
      if (s != null)
        Some(s.name, s.age)
      else
        None
    }
  }
  //main方法，作为程序的主入口.
  def main(args: Array[String]): Unit = {
    //3. 创建Student类的对象.
    val s = new Student("糖糖", 73)          //普通方式创建对象.
    val s2 = Student("糖糖", 73)             //免new，创建对象，   apply方法保证

    //4. 打印对象的属性值
    println(s2.name + "..." + s.age)      //普通方式获取对象的属性值

    //5. 通过提取器获取对象中的方法.
    val result = Student.unapply(s2)
    println(result)
  }
}
```

7. 案例: 随机职业

7.1 需求

1. 提示用户录入一个数字(1~5), 然后根据用户录入的数字, 打印出他/她上辈子的职业.
2. 假设: 1-> 一品带刀侍卫, 2 -> 宰相, 3 -> 江湖郎中, 4 -> 打铁匠, 5 -> 店小二.

7.2 目的

考察 `键盘录入，模式匹配` 相关内容.

7.3 步骤

1. 提示用户录入数字, 并接收.
2. 通过模式匹配获取该用户上辈子的职业.
3. 打印结果.

7.4 参考代码

```scala
//案例: 随机职业.
object ClassDemo18 {
  def main(args: Array[String]): Unit = {
    //1. 提示用户录入数字, 并接收.
    println("请录入一个数字(1~5), 我来告诉您上辈子的职业: ")
    val num = StdIn.readInt()
    //2. 通过模式匹配获取该用户上辈子的职业.
    //假设: 1-> 一品带刀侍卫, 2 -> 宰相, 3 -> 江湖郎中, 4 -> 打铁匠, 5 -> 店小二.
    val occupation = num match {
      case 1 => "一品带刀侍卫"
      case 2 => "宰相"
      case 3 => "江湖郎中"
      case 4 => "打铁匠"
      case 5 => "店小二"
      case _ => "公公"
    }
    //3. 打印结果.
    println(s"您上辈子的职业是: ${occupation}")
  }
}
```