样例类和模式匹配

1. 样例类 `case class` 和模式匹配 `pattern matching` 是一组孪生语法，它们为我们编写规则的、未封装的数据结构提供了支持。它们对于表达树形的递归数据尤其有用。

一、样例类

1. 样例类是 `Scala` 用于对象模式匹配，并且它不需要大量的样板代码。

   样例类是带 `case` 修饰符的类。通过这个修饰符，`Scala` 编译器对我们的类添加了一些语法上的便利。

```
case class  C(name:String,age:Int)  // 样例类: name,age 都会被视为字段

val c = C("zhang san",20)           // 不需要 new
println("name:" + c.name + " ;age: " + c.age) // name,age 都会被视为字段
println(c.toString)                           // 自动实现了 toString
println(c.hashCode)                           // 自动实现了 hashCode

val c2 = c.copy(name="li si")
println(c2.toString)
println(c.equals(c2))                          // 自动实现了 equals
```

   - `Scala` 编译器会为样例类添加一个和类同名的工厂方法。这意味着可以使用 `C(...)` 来构造对象，而不是使用 `new C(...)` 来构造对象。这样使得代码中不再到处是 `new` 关键字。
   - `Scala` 编译器会为样例类参数列表中的参数都隐式获取一个 `val` 前缀，因此它们都会被当作字段处理
   - `Scala` 编译器会帮我们以"自然"的方式实现 `toString,hashCode,equals` 方法。这些方法会分别打印、哈希、比较包含类及所有入参的整棵树。

     因为 `Scala` 的 `==` 总是代理给 `equals` 方法，因此样例类总是以结构化的方式作比较。
   - `Scala` 编译器还会添加一个 `copy` 方法用于制作修改过的拷贝。这个方法支持带名字的参数，以及缺省参数：可以通过带名字的参数给出希望做的修改；然后对于任何未给出的参数，都会使用旧对象中的原值。

2. 样例类以及样例类的对象都比常规类和常规类的对象大一点，这是因为 `Scala` 编译器会生成额外的方法，并且对构造方法的每个参数都隐式添加了字段。
3. 样例类最大的好处是它支持模式匹配。

二、模式匹配

1. 模式匹配由 `match` 表达式实现，其语法为：

```
选择器 match {
    可选分支一
    可选分支二
    ...
}
```

- `选择器`是一个表达式，`match`将对该表达式的结果进行选择。
- `可选分支`是以`case`关键字打头的、包含一个模式以及一个或者多个表达式。
  - 模式和表达式通过箭头符`=>`分开。
  - 如果匹配到该模式，则这些表达式就会被求值，求值结果将作为`match`表达式整体的结果。
  - 如果没有表达式，则结果是`unit`值，即`()`。
- 一个`match`表达式的求值过程是按照给出的模式顺序逐一尝试的。
  - 一旦某个模式被匹配，则该模式后面跟着的表达式被执行。
  - 一旦某个模式被匹配，则后续的模式将不再尝试。

2. `match`和`Java`的`switch`很相似。但是它们有重要区别：

```
switch(选择器) {
    可选分支一
    可选分支二
    ...
}
```

   - `Scala`的`match`是一个表达式，它总会返回一个值。
   - `Scala`的可选分支并不会贯穿到下一个`case`。
   - 如果没有任何一个模式匹配上，则`Scala`会抛出`MatchError`异常。

     这意味着你需要确保所有的`case`都会被覆盖到。

3. 如果没有任何一个模式匹配上，此时你可以为选择器部分添加一个`@unchecked`注解。那么编译器对后续模式分支的覆盖完整性检查就会被抑制，这样就不会抛出`MatchError`异常。

```
(选择器 : @unchecked) match {
    可选分支一
    可选分支二
    ...
}
```

2.1 模式种类

1. 所有的模式跟相应的表达式看上去完全一样。
2. 模式会按照代码中的顺序逐个被尝试。通常要求捕获通用的`case`出现在更具体的`case`之后。

   如果我们将顺序颠倒过来，那么捕获通用的`case`就会在更具体的规则之前执行。在许多场景下，编译器甚至会拒绝编译。

2.1.1 通配模式

1. 通配符`_`可以匹配任何对象。
   - 它常用于缺省的、捕获所有剩余可选路径。

```
def f(expr : String) = expr match{
    case "name" => println("catch:"+expr)
    case _      => println("default")       // 默认 case
}
```

- 也可以用于忽略某个对象中你并不关心的局部信息。

```
abstract class Parent
case class  C(name:String,age:Int) extends Parent

def f(p: Parent) =  p match{
    case C(_,_) => println("catch:"+p)      // 不关心 C 的参数
    case _      => println("default")       // 默认 case
}
```

2.1.2 常量模式

1. 常量模式仅仅匹配自己：
   - 任何字面量都可以作为常量模式使用。如：`"+"` 和 `1` 这样的常量模式可以匹配那些按照 `==` 的要求跟它们相等的值。
   - 任何 `val` 或单例对象也可以被当作常量模式使用。如：`Nil` 这个单例对象仅能匹配空列表。

```
val S = "world"
def f(x:Any) =  x match{
    case 1 => "catch:1"
    case "hello" => "catch:hello"
    case Nil => "catch:empty list"          // 单例对象作为常量模式
    case S => "catch:"+S                    // val 作为常量模式
    case _ => "default"
}
```

2.1.3 变量模式

1. 类似 `e` 这样的变量模式可以匹配任何值。匹配后，在右侧的表达式中，这个变量 `e` 将绑定成该匹配的值。

   在绑定之后，你可以用这个变量来做进一步处理。

```
def f(x:Any) =  x match{
    case e => println(e)
}
```

2. 通配符模式 `_` 也可以匹配任何值，但是它并不会引入一个变量名来指向这个值。
3. 采用变量模式之后，就没必要继续跟着通配符模式 `_`。因为变量模式已经可以匹配任何值了，因此不可能走到后面的 `case`。
4. 常量模式中，也有可能出现符号形式的名字，如 `Nil`。当我们将 `Nil` 当作一个模式的时候，实际上就是在用一个符号名称来引用常量。

   `Scala` 通过一个简单的词法规则来区分：
   - 以一个小写字母打头的简单名称会被当作模式变量处理。
   - 所有其它引用都是常量。
   - 如果需要用小写的名称作为模式常量，有两个办法：
     - 如果常量是某个对象的字段，则可以在字段名之前加上限定词。如 `this.n`。

       尽管它们是以小写开头，但是由于 `.` 的存在，它们会被解析为常量模式。

- 使用反引号将这个名字包围起来。

> `Scala` 中反引号有两个用途：
>
> - 将小写字母打头的标识符用作模式匹配中的常量。
> - 将关键字当作普通的标识符。

```
val name = "hello"
def f(x:Any) =  x match{
  case `name` => println("catch hello")
  case Nil => println("catch Nil")
  case nil => println("catch:"+nil)
}
f("hello")          // 匹配到 name:  常量模式
f(List())           // 匹配到 Nil : 常量模式
f("world")          // 匹配到 nil : 变量模式
```

2.1.4 构造方法模式

1. 构造方法模式看上去就像 `UnOP("_",e)` 。这个模式匹配所有类型为 `UnOP`，并且首个入参匹配 `"_"`、第二个入参匹配 `e` 的值。

   构造方法模式由一个名称和一组圆括号中的模式组成。假设这里的名称是一个样例类，则这样的一个模式：

   - 首先检查被匹配的对象是否是以这个名称命名的样例类的实例。
   - 然后检查这个对象的构造方法参数是否匹配这些额外给出的模式。
     - 这些额外的模式意味着 `Scala` 的模式支持深度匹配。这样的模式不仅检查给定的对象的顶层，还将进一步检查对象的内容是否匹配额外的模式要求。
     - 由于额外的模式也可能是构造方法模式，因此检查对象内部时可以到任意的深度。
     - 额外的模式可以通过通配符 `_` 或者其它的模式来匹配。尤其可以通过变量模式来绑定匹配的值。

```
case class Worker(name:String,life_time:Double)
case class System(worker:Worker,lift_time:Double)
def f(x:Any) = x match{
  case System(worker(name,10.0),t) =>
        println("a:worker name="+ name + " ; system life="+t)
  case System(worker(name,_),t) =>
        println("b:worker name="+ name + " ; system life="+t)
  case other =>
        println("catch other: "+other)
}
f(System(Worker("first",10.0),100.0))  // 匹配到第一个 case
f(System(Worker("second",11.0),100.0)) // 匹配到第二个 case
f(Worker("second",11.0))               // 匹配到第三个 case
```

2.1.5 序列模式

1. 和样例类匹配一样，也可以和序列类型做匹配，如 `List` 或 `Array` ，使用的语法是相同的。
   - 可以在模式中给出任意数量的元素。
   - 如果想匹配一个序列，但又不想给出很长的元素，则可以用 `*_` 作为模式的最后一个元素。

     这种方式可以匹配序列中任意数量的元素，包括 `0` 个元素。

```
def f(x:Any) = x match {
  case List(0,_) => println("2 elements list,start with 0")
  case List(0,_,_) => println("3 elements list,start with 0")
  case List(e,_,_,_) => println("4 elements list,catch :"+e)   // 绑定到 e
  case List(0,_*) => println("various elements list,start with 0") // 以 0
开始的任意长度
  case _ => println("default")
}
f(List(0))          // case List(0,_*)
f(List(0,1))        // case List(0,_)
f(List(0,1,2))      // case List(0,_,_)
f(List(0,1,2,3))    // case List(e,_,_,_) : e 为 0
f(List(0,1,2,3,4)) // case List(0,_*)
f(Array(0,1))       // case _
```

2.1.6 元组模式

1. 模式匹配还支持元组。形如 `(a,b,c)` 这样的模式能够匹配任意的三元组。

```
def f(x:Any) = x match {
  case (a,b,c) => println("a:"+a+" ;b:"+b+" ;c:"+c)
  case _ => println("default")
}
f(1,"second",3.0)  // case (a,b,c)
```

2.1.7 带类型的模式

1. 可以用带类型的模式来替换类型测试和类型转换。
2. 假如需要设计一个函数，该函数返回不同类型的对象的大小或者长度。常规的设计思路时：通过类型测试以及类型转换：

```
def get_size(x:Any) = {
    if x.isInstanceOf[String]{          // 如果是字符串
        val s = x.asInstanceOf[String] // 强制类型转换
        s.length
    }else ...                           // 如果是其它序列，如 List,Map 等
}
```

   - 通过 `expr.isInstanceOf[T]` 可以判断 `expr` 是否是 `T` 类型，通过 `expr.asInstanceOf[T]` 可以将 `expr` 转换成 `T` 类型。

     这两个操作符会被当成 `Any` 类的预定义方法处理，它们接收一个用方括号括起来的类型参数。

   - `Scala` 中编写类型测试和类型检查会很罗嗦。这是有意为之，因为 `Scala` 并不鼓励这样做。`Scala` 推荐使用带类型的模式，尤其是当你需要同时执行类型测试和类型转换时。因为这两个操作所作的事情会被并在单个模式匹配中完成。

     ```
     def get_size(x:Any) = x match  {
         case s: String => s.length
         case m: Map[_,_] => m.size
         case _ => -1
     }
     ```

3. 在类型模式中，你也可以使用下划线来通配任意类型，这就像是其它模式中的通配符。如 `Map[_,_]` 。

4. 和 `Java` 一样，`Scala` 的泛型采取了类型擦除。这意味着运行时并不会保留类型参数的信息。因此我们在运行时无法判断某个给定的 `Map` 对象是由两个 `Int` 的类型参数创建，还是由其它类型参数创建。系统只能判断某个对象是不是 `Map` 。

   对这个规则唯一例外的是数组。因为 `Java` 和 `Scala` 都对它们进行了特殊处理，数组的元素类型和数组是一起保存的，因此可以对其进行模式匹配。

```
def isIntIntMap(x:Any) = x match{
  case m: Map[Int,Int] => true
  case _ => false
}
def isIntArray(x:Any) = x match{
  case a: Array[Int] => true
  case _ => false
}
println(isIntIntMap(Map[Int,Int](100 -> 1, 200 ->2)))  // 输出：  true
println(isIntIntMap(Map[String,Int]("a" -> 1,"b"->2))) // 输出：  true

println(isIntArray(Array[Int](1,2,3)))                 // 输出：  true
println(isIntArray(Array[String]("a","b","c")))        // 输出：  false
```

2.2 变量绑定

1. 除了独立存在的变量模式之外，还可以对任何其它模式添加变量。方式为：`变量名@模式` ，这就得到一个变量绑定模式。

   变量绑定模式和常规模式一样执行模式匹配。如果匹配成功，就将匹配的对象赋值给这个变量，就像简单的变量模式一样。

```
case class Worker(name:String,life_time:Double)
case class System(worker:Worker,lift_time:Double)
def f(x:Any) = x match{
  case System(worker@Worker(_,_),t) => println("worker:"+ worker) // 绑定到
worker
  case other => println("catch other: "+other)
}
f(System(Worker("first",10.0),100.0))
```

2.3 模式守卫

1. `Scala` 要求模式都是线性的：同一个模式变量在模式中只能出现一次。

```
case class Worker(name:String,life_time:Double)
case class System(worker:Worker,lift_time:Double)
def f(x:Any) = x match{
  case System(worker@Worker(_,t),t) => println(worker)
  // 希望两个 lift_time 是一样的，但是这里会编译失败
  case other => println("catch other: "+other)
}
```

如果希望模式变量出现多次，则可以用模式守卫来重新定义匹配逻辑。模式守卫出现在模式之后，并以 `if` 打头。

- 模式守卫可以是任意的布尔表达式，通常会引用到模式中的变量。
- 如果存在模式守卫，则这个匹配仅在模式守卫求值得到 `true` 时才会成功。

```
case class Worker(name:String,life_time:Double)
case class System(worker:Worker,lift_time:Double)
def f(x:Any) = x match{
  case System(worker@Worker(_,t1),t2) if t1==t2
          => println("worker:"+ worker)          // 模式守卫
  case other => println("catch other: "+other)
}
f(System(Worker("first",10.0),10.0))  // 匹配到第一个 case
f(System(Worker("second",10.0),20.0)) // 匹配到第二个 case
```

2.4 密封类

1. 当我们编写一个模式匹配时，需要确保完整地覆盖了所有可能的 `case` 。
   - 有时候可以通过在末尾添加一个缺省 `case` 来做到，但是这仅限于有合理兜底的场景。
   - 如果没有一个缺省的 `case` ，我们可以求助于 `Scala` 编译器，编译器帮我们检测出 `match` 表达式中缺失的模式组合。

     为了做到这一点，编译器需要分辨出所有可能的 `case` 有哪些。事实上这是不可能的。

```
def f(x:Any) = x match{
    case i:Int => println("catch int:"+i)
    case s:String => println("catch String:"+s)
    /*
    如果没有缺省 case，Scala 很难知道所有可能的 case 有哪些
    */
}
```

2. 解决这个问题的方法是：将这些样例类的超类标记为密封的。语法是：在类继承关系顶部的那个类的类名前加上 `sealed` 关键字。
   - 密封类除了在同一个文件中定义的子类之外，不能添加新的子类。这对于模式匹配而言非常有用，因此我们就只需要关心那些已知的样例类。
   - 此外编译器还能更好的支持模式匹配。如果对继承自密封类的样例类进行匹配，编译器会用警告消息标识出缺失的模式组合。

     因此如果你的类打算被用于模式匹配，则你应该将其作为密封类。这也是为什么 `sealed` 关键字通常被看作模式匹配的通行证的原因。

     ```
     sealed abstract class Man
     case class Worker(name:String) extends Man
     case class Leader(name:String) extends Man

     def f(x:Man) = x match{
       case w:Worker => println("catch worker:" + w)
     }
     f(Worker("worker_1"))
     ```

```
/*
编译器警告:
Warning:(5, 24) match may not be exhaustive.
It would fail on the following input: Leader(_)
def f(x:Man) = x match{
*/
```

解决办法是：添加一个缺省的`case`用于捕获所有的模式，或者补全缺失的样例类。

三、Option 类型

1. `Scala`有一个名为`Option`的标准类型表示可选值。这样的值可以有两种形式：
   - `Some(x)`，其中`x`就是那个实际的值。
   - `None`，表示没有值。
2. `Scala`的集合类中某些标准操作会返回可选值。如：`Scala`的`Map`有一个`get`方法，当传入的键有对应的值时，返回`Some(value)`；当传入的键在`Map`中没有定义时，返回`None`。

```
val map = Map("a"->1,"b"->2)
println(map.get("a"))   // 打印: Some(1)
println(map.get("c"))   // 打印: None
```

3. 将可选值解开的常见方式是通过模式匹配。

```
val map = Map("a"->1,"b"->2)

def get_option_value(x:Option[Int]) = x match{
  case Some(s) => {
    println("get value:"+s)
    s }
  case None => {
    println("get None")
    None}
}

get_option_value(map.get("a"))  // 打印: get value:1
get_option_value(map.get("c"))  // 打印: get None
```

4. `Scala`程序经常使用`Option`类型，这个类型可以和`Java`的`null`来表示空值来作比较。
   - `Java`的`null`很容易出错。因为在实践中，想要跟踪程序中哪些变量为`null`是一件很困难的事。

     如果某个变量允许为`null`，则必须记住每次使用它时都要判断是否为`null`。如果忘记做判断，则很容易出现`NullPointerException`。

     由于这一类异常可能并不是经常发生，因此在测试过程中很难发现。
   - `Scala`的`Option`则不容易出错。
     - 采用`Option`的代码更为直观。某个类型为`Option[String]`的变量提示了你：该变量可能为空；而在`Scala`中，类型为`String`的变量提示了你：该变量一定非空。
     - 如果返回了`Option`类型，则不检查是否为空的代码在`Scala`中会变成类型错误，因此无法编译通过。

```
val map = Map("a"->"first","b"->"second")

val opt_str:Option[String] = map.get("a") // 提醒你: opt_str 可能为空
val str:String = map.get("c")  // 编译失败，因为 get 方法返回类型 Option[String]
```

四、一切皆模式

1. `Scala` 中，很多地方都允许使用模式，而不仅仅是 `match` 表达式。

4.1 变量定义中的模式

1. 当定义一个`val` 或者`var` 时，都可以用模式而不是简单的标识符。如：可以将一个元组解开，并将其中每个元素分别赋值给不同的变量。

```
val tuple = (123,"abc")
val list = List(456,789)
val (num1,str1) = tuple       // 解包（必须包含圆括号，因为圆括号表示元组）
val List(num2,num3) = list    // 解包
println("num1:"+num1+";\tstr1:"+str1+";\tnum2:"+num2+";\tnum3:"+num3)
// 打印: num1:123;  str1:abc; num2:456; num3:789
```

2. 这种语法结构在处理样例类时非常有用。当你知道要处理的样例类是什么时，就可以用一个模式来解析它。

```
case class Worker(name:String)
case class System(worker:Worker,name:String)

val system = System(worker("worker_1"),"system_1")
val System(unpack_worker,system_name) = system     // 解包

println("unpack_worker:"+unpack_worker+"; system_name:"+system_name)
// 打印: unpack_worker:Worker(worker_1); system_name:system_1
```

4.2 case 序列

1. 用花括号包起来的一系列`case` （即可选分支）可以用在任何允许出现函数字面量的地方。

   本质上讲，`case` 序列就是一个函数字面量，只是更加通用。不像普通函数那样只有一个入口和参数列表，`case` 序列可以有多个入口，每个入口都有自己的参数列表。每个`case` 对应该函数的一个入口，该入口的参数列表通过模式来指定。每个入口的逻辑主体是`case` 右边的部分。

```
val func: Any => Unit = {
  case (i,j) => println("catch tuple:("+i+","+j+")")
  case s:String => println("catch string:"+s)
  case Some(x) => println("catch Opiton:"+x)
  case e => println("catch other:"+e)
}

func(Tuple2("hello",1))  // 匹配第一个 case: i="hello",j=1
func("hello",2)          // 匹配第一个 case: i="hello",j=2
func("hello",2,0.0)      // 匹配第四个 case: e=("hello",2,0.0)
func("A String")         // 匹配第二个 case: s="A String"
func(Some(0.0))          // 匹配第三个 case: x=0.0
func(Map("a"->1))        // 匹配第四个 case: e=Map("a"->1)
```

2. 通过 `case` 序列得到的是一个偏函数。如果我们将这样一个函数应用到它不支持的值上，则会产生一个运行时异常。

```
val second_val: List[Int] => Int = {
    case x :: y :: _ => y
}
println(second_val(List(1,2,3)))  // 打印: 2
println(second_val(List()))       // 运行时异常：抛出MatchError
```

编译时，编译器会警告：

```
Warning:(1, 44) match may not be exhaustive.
It would fail on the following inputs: List(_), Nil
val second_val: List[Int] => Int = {
```

3. 如果希望检查某个偏函数是否对某个入参有定义，则必须首先告诉编译器：你知道你要处理的是偏函数。但是`List[Int] => Int`这个类型涵盖了所有从`List[Int]`到`Int`的函数，不论这个函数是偏函数还是全函数。

   仅仅涵盖从`List[Int]`到`Int`的偏函数的类型写作`PartialFunction[List[Int],Int]`。偏函数定义了一个`isDefinedAt`方法，用于检查该函数是否对某个特定的参数值有定义。

```
val second_val: PartialFunction[List[Int], Int] = {
  case x :: y :: _ => y
}
println(second_val.isDefinedAt(List()))      // 打印: false
println(second_val.isDefinedAt(List(1)))     // 打印: false
println(second_val.isDefinedAt(List(1,2)))   // 打印: true
println(second_val.isDefinedAt(List(1,2,3))) // 打印: true
```

4. 偏函数的典型应用场景是模式匹配函数字面量，如这里的示例。事实上，这样的表达式会被`Scala`编译器翻译成偏函数。这样的翻译发生了两次：一次是实现真正的函数，另一次是测试这个函数是否对指定参数值有定义。

   如，函数字面量`{ case x :: y :: _ => y}`被翻译为如下的值偏函数：

```
new PartialFunction[List[Int],Int]{
    def apply(xs:List[Int]) = xs match{
        case x :: y :: _ => y
    }
    def isDefinedAt(xs:List[Int]) = xs match {
        case x :: y :: _ => true
        case _ => false
    }
}
```

- 如果函数字面量声明的类型是`PartialFunction`，则这样的翻译就会生效。
- 如果声明的类型是`Function1`，或者没有声明，则函数字面量对应的就是一个全函数。

5. 一般来说推荐使用全函数，因为偏函数允许运行时出现错误，而这个错误编译器无法帮助我们。

   不过有时候偏函数也特别有用。比如：你确信不会有无法处理的值传入。也有的框架可能会用到偏函数，每次函数调用前都会用`isDefinedAt`做一次检查。

4.3 for 表达式中的模式

1. 可以在`for`表达式中使用模式。

```
for ((name,age) <- Map("zhang san" -> 20,"li si" -> 23))
  println("name:"+name+";age:"+age)

/*
输出:
name:zhang san;age:20
name:li si;age:23
*/
```

   这个`Map`每次迭代时交出一个对偶，然后该对偶与`(name,age)`进行匹配。这个匹配永远是成功的。

2. 也有匹配失败的情况，迭代中无法匹配的值会被直接丢弃。

```
for(Some(name) <- List(Some("zhang san"),Some("li si"),None))
  println("name:"+name)

/*
输出:
name:zhang san
name:li si
*/
```

   由于`None`无法匹配`Some(name)`，因此`None`不会出现在输出中。

五、提取器

1. 目前为止，构造方法模式都和样例类有关。如`Some(x)`是一个合法的模式，因为`Some`是一个样例类。有时候我们需要提取模式，但是不希望创建相关的样例类，此时需要用到提取器。提取器是模式匹配的进一步泛化。

2. 在 `Scala` 中，提取器是拥有名字叫 `unapply` 成员方法的对象。这个 `unapply` 方法的目的是跟某个值做匹配，并将其拆解开。

   通常提取器对象还会定义一个跟 `unapply` 相对应的 `apply` 方法用于构建值，不过这不是必须的。

   如下所示为一个用于处理 `email` 地址的提取器对象：

   ```
   object Email{
     def apply(user: String, domain: String) = user + "@" + domain  // apply 方法不是必须的
     def unapply(str: String) :Option[(String,String)] = {
       val parts = str split "@"
       if(parts.length == 2) Some(parts(0), partes(1)) else None
     }
   }
   ```

   这个对象同时定义了 `apply` 方法和 `unapply` 方法。

   - `apply` 方法使得我们可以像调用函数一样调用 `Email` 对象：`Email("huaxz","163.com")` 。

     如果想要更明显的表明意图，还可以让 `Email` 继承自 `Scala` 的函数类型：

     ```
     object Email extends ((String,String) => String){
       ...
     }
     ```

     这个对象声明当中的 `(String,String) => String`，其含义跟 `Function2[String, String, String]` 一样。

   - `unapply` 方法就是将 `Email` 变成提取器的核心方法。从某种意义上讲，它是 `apply` 方法的逆运算。但是 `unapply` 还需要处理输入的字符串不是 `email` 的情况，这也是为什么 `unapply` 方法的返回类型为 `Option` 类型。

3. 每当模式匹配遇到引用提取器对象的模式时，它都会调用提取器的 `unapply` 方法。例如：

   ```
   str match{
     case Email(user, domain) => ...
     case ...
   }
   ```

   这将会引发如下调用 `Email.unapply(str)` 。调用结果要么返回 `None`，要么返回 `Some(u,d)` 。

   - 如果返回 `Some(u,d)`，则模式能够匹配，则变量 `user` 会被绑到返回值 `u`，变量 `domain` 会绑到返回值 `d` 。
   - 如果返回 `None`，则模式未能匹配。系统会继续尝试匹配下一个模式。如果没有下一个模式，则系统返回 `MatchError` 异常。

   另外，这里的 `str` 的类型可以是 `Email.unapply` 的参数类型，但也可以不是。我们可以用提取器来匹配更笼统的类型表达式：

```
val x: Any = ....
x match{
  case Email(user, domain) => ...
  case ...
}
```

此时：

- 系统首先会检查给定的值 `x` 是否满足 `Email.unapply` 参数类型的要求。如果满足要求，则 `x` 就会被转成 `String`（`Email.unapply` 的参数类型），然后继续上述讨论的模式匹配流程。
- 如果类型不满足要求，则模式未能匹配，系统继续尝试匹配下一个模式。

4. 在对象 `Email` 中，`apply` 方法称作注入 `injection`，因为它接收某些入参，并交出给定的集合（在我们这里表示能够代表 `email` 地址的字符串集合）；`unapply` 方法称作提取 `extraction`，因为它接收上述集合的一个元素，并将它的某些组成部分提取出来。

   `injection` 和 `extraction` 通常成对地出现在对象中，因为这样一来我们就可以同时表示构造方法和模式。但是我们也可以在不定义 `injection` 的情况下单独定义 `extraction`。

   - 含有 `extraction` 的对象被称作提取器 `extractor`，无论它是否有 `apply` 方法。
   - 如果同时包含了 `injection` 和 `extraction` 方法，则它们是一对 `dual` 对偶方法。虽然这不是 `scala` 语法必须的，但是我们建议这么做。

     ```
     Email.unapply(Email.apply(user, domain)) // 应该返回 Some(user,domain)
     ```

5.1 提取 0 个或 1个变量的模式

1. 前面的 `unapply` 方法在成功的时候返回一对元素的值，这很容易推广到多个变量的模式。如果需要绑定 `N` 个变量，则 `unapply` 方法可以返回一个以 `Some` 包起来的 `N` 个元素的元组。

   - 但是，当模式只绑定一个变量时，处理逻辑是不同的。`Scala` 并没有单个元素的元组，因此 `unapply` 方法只是简单地将元素本身放到 `Some` 里。

     如以下提取器提取连续相同的子串：

     ```
     object Twice{
       def apply(s: String): String = s + s
       def unapply(s: String): Option[String] = {
         val len= s.length / 2
         val halfStr = s.substring(0, len)
         if(halfStr == s.substring(len)) Some(halfStr) else None
       }
     }
     ```

   - 也可能某个提取器模式不绑定任何变量，这时对应的 `unapply` 方法返回布尔值（`true` 表示成功，`false` 表示失败）。

     ```
     object UpperCase{
       def unapply(s: String): Boolean = s.toUpperCase == s
     }
     ```

     这里仅定义了 `unapply`，并没有定义 `apply`。定义 `apply` 没有任何意义，因为没有任何东西需要构造。

2. 如果我们希望匹配那些大写的、重复出现的子串。因此可以这样使用提取器：

```
s math{
  case Twice(x @ UpperCase()) => println("match :"+x)
}
```

这里有两点注意：

- `UpperCase()` 的空参数列表不能省略，否则匹配的就是和 `UpperCase` 这个对象的相等性。
- 尽管 `UppserCase()` 本身不绑定任何变量，但是我们仍然可以将跟它匹配的整个模式关联一个变量。做法是标准变量绑定机制：`x @ UpperCase()` 。这样的写法将变量 `x` 跟 `UpperCase()` 匹配的模式关联起来。

5.2 提取可变长度参数的模式

1. 目前的 `unapply` 方法只支持固定长度的变量匹配。为支持可变长度的变量匹配，`scala` 允许我们定义另一个不同的 提取方法：`unapplySeq` 。

```
object Domain{
  def apply(parts: String*) : String = parts.reverse.mkString(".")
  def unapplySeq(str: String): Option[Seq[String]] =
Some(str.split("\\.").reverse)
}
```

这里 `unapplySeq` 的结果类型必须符合 `Option[Seq[T]]` 的要求，其中元素类型 `T` 可以为任意类型。

另外，这里的 `apply` 方法也不是必须的。

2. 从 `unapplySeq` 返回"固定元素 + 可变部分" 也是可行的。这是通过将所有元素放在一个元组里面来实现的，其中可变部分放在元组最后：

```
object Email{
  def unapplySeq(email: String): Option[(String,Seq[String])] ={
    val partes = email split "@"
    if(partes.length == 2) Some(partes(0), partes(1).split("\\.").reverse)
    else None
  }
}
```

5.3 提取器和序列模式

1. 可以使用序列模式来访问列表或数组的元素，如：

```
List()
List(x, y, _*)
Array(x, 0, 0, _)
```

事实上，`Scala` 标准库中的这些序列模式都是用提取器实现的。例如 `List(...)` 这样的模式之所以可行，是因为 `scala.List` 的伴生对象是一个定义了 `unapplySeq` 方法的提取器。

```
package scala
object List{
  def apply[T](elems: T*) = elems.toList
  def unapplySeq[T](x: List[T]): Option[Seq[T]] = Some(x)
}
```

`List` 伴生对象包含了一个接收可变数量的入参的 `apply` 方法，正是这个方法让我们可以编写 `List(), List(1, 2, 3)` 这样的表达式。

`List` 伴生对象还有一个以序列形式返回列表所有元素的 `unapplySeq` 方法。正是这个方法在支撑 `List(...)` 这样的模式。

`scala.Array` 对象中我们也能够找到类似的定义，这些定义支持对数组的 `injection` 和 `extraction` 。

5.4 提取器 vs 样例类

1. 样例类的缺点：样例类将数据的具体表现类型暴露给使用方。这意味着构造方法模式中使用的类名和选择器对象的具体表现类型相关。

   例如如下的模式匹配：

   ```
   a match{
     case C(...)
   }
   ```

   如果匹配成功了，那么你就知道选择器表达式是 `C` 这个类的实例。

   提取器打破了数据表现和模式之间的关联：提取器支持的模式，和被选择的对象的数据类型没有任何联系。这个性质被称作表现独立 `representation independence` 。这很重要，因为它允许我们修改某些组件的实现类型，同时又不影响这些组件的使用方。

   如果你的组件定义了样例类，你就难以修改这些样例类，因为使用方可能已经包含了对这些样例类的模式匹配。重命名这些样例类，或者改变类的继承关系都会影响到使用方的代码。

   提取器没有这个问题，因为它们介于数据表现层和使用方看到的内容之间。你可以改变某个类型的具体表现形式，而不影响使用方的代码。

2. 表现独立是提取器对于样例类的一个重要优势。另一方面，样例类也有一些相对于提取器的优势。
   - 首先，设置和定义样例类要比提取器简单得多，需要的代码也更少。
   - 其次，样例类比提取器能够带来更高效的模式匹配，因为 `scala` 编译器能够对使用样例类的模式做更好的优化。
   - 最后，如果你的样例类继承自一个 `sealed` 基类，那么 `scala` 编译器将检查你的模式匹配是否完整全面。如果有的值没有被覆盖到，则编译时会报错。对于提取器而言，并不存在这样的全面性检查。
3. 至于选择样例类还是提取器，视具体情况而定。
   - 如果你编写的是一个封闭的应用，则样例类通常更好，因为它们精简、快速、还可以有静态检查。

     如果你之后决定需要修改你的类继承关系，则应用需要被重构。
   - 如果你需要将类型暴露给第三方，那么提取器可能是更好的选择，因为它们保持了表现独立。

   通常建议从样例类开始，随着需求变化再切换成提取器。

5.5 正则表达式

1. 提取器的一个应用场景是正则表达式。和 `Java` 一样， `scala` 也通过类库提供了正则表达式的支持，不过提取器让我们更容易使用正则表达式。
2. `Scala` 从 `Java` 继承了正则表达式语法，而 `Java` 又从 `Perl` 继承了大部分的正则表达式特性。
3. `Scala` 的正则表达式类位于 `scala.util.matching` 包 `scala.util.matching.Regex` 。创建一个正则表达式是将一个字符串传给 `Regex` 方法来创建的。如：

```
import scala.util.matching.Regex
val reg = new Regex("(-)?(\\d+)(\\.\\d*)?")
```

这里 `\` 出现多次。这是因为在 `Java` 和 `Scala` 中，字符串中的单个反斜杠表示转义字符，而不是字符串中的常规字符。所以你需要写出 `\\` 从而得到字符串里的一个 `\` 字符。

4. 如果正则表达式中有很多反斜杠，那么写起来和读起来都比较痛苦。可以使用 `Scala` 原生字符串来解决这个问题。

`Scala` 原生字符串是由一对三个连续双引号括起来的字符序列，它和普通字符串的区别在于：原生字符串中的字符和它本身一样，没有转义。

因此上述正则表达式也可以写成：

```
val reg = new Regex("""(-)?(\d+)(\.\d*)?""")
```

5. 在 `scala` 中一种更短的创建正则表达式的方式是：

```
val reg = """(-)?(\d+)(\.\d*)?""".r
```

这是由于 `StringOps` 类有一个名为 `.r` 的方法，它将字符串转换为正则表达式。其定义为：

```
package scala.runtime
import scala.util.matching.Regex
class StringOps(self: String)...{
  ...
  def r = new Regex(self)
}
```

6. 有几种利用正则表达式查找的方法：
   - `regex findFirstIn str`：查找 `str` 中正则表达式首次出现的结果，结果为 `Option` 类型（找不到就是 `None`）。

     ```
     val reg = new Regex("""(-)?(\d+)(\.\d*)?""")
     reg findFirstIn "123.5 abc 45.6" // 返回 Option[String] = Some(123.5)
     ```

   - `regex findAllIn str`：查找 `str` 中正则表达式所有出现的结果，结果为 `Iterator` 类型。

     ```
     val reg = new Regex("""(-)?(\d+)(\.\d*)?""")
     reg findFirstIn "123.5 abc 45.6" // 返回一个非空迭代器，迭代内容为
     ["123.5", "45.6"]
     ```

- `findPrefixOf str`：查找 `str` 以正则表达式开始（即 `str` 的头部）的结果，结果为 `Option` 类型（找不到就是 `None`）。

```
val reg = new Regex("""(-)?(\d+)(\.\d*)?""")
reg findFirstIn "de 123.5"  // 返回 Option[String] = Some(123.5)
reg findPrefixOf "de 123.5" // 返回 None
reg findPrefixOf "123.5 de" // 返回 Option[String] = Some(123.5)
```

7. 在 `scala` 中每个正则表达式都定义了一个提取器。该提取器用来识别正则表达式中的分组（正则表达式中每个括号 `()` 代表一个分组）匹配的子字符串。

   例如：

```
val reg = """(-)?(\d+)(\.\d*)?""".r
val reg(sign, intPart, deciPart) = "-1.23"
// sign 被绑定为 '-',   intPart 被绑定为 '1',  deciPart 被绑定为 '.23'
```

   在这里，模式 `reg(...)` 被用在了 `val` 的定义中。这里的 `reg` 正则表达式定义了一个 `unapplySeq` 方法，这个方法会匹配任何与 `reg` 匹配的字符串。

   - 如果字符串匹配了，那么正则表达式 `reg` 中的三个分组对应的部分就被作为模式的元素返回，进而被 `sign, intPart, deciPart` 绑定。
   - 如果某个分组缺失，则对应的元素值就被设为 `null` 。如：

```
val reg = """(-)?(\d+)(\.\d*)?""".r
val reg(sign, intPart, deciPart) = "1.23"
// sign 被绑定为 null ,   intPart 被绑定为 '1',  deciPart 被绑定为 '.23'
```

   - 如果无法匹配，则抛出 `MatchError` 异常。

     注意：这里要求字符串和正则表达式从头到尾完全匹配。这不同于 `findFirstIn/findAllIn/findPrefixOf` ，后者只需要字符串中部分匹配即可。

8. 我们也可以在 `for` 表达式中混用提取器和正则表达式的 `find` 。如下面的示例对字符串中找到的所有十进制数字进行拆解：

```
val reg = """(-)?(\d+)(\.\d*)?""".r
for (reg(s,i,d) <- reg.findAllIn(inputStr))
    println("sign: %s, integer: %s, decimal: %s".format(s, i, d))
```