

---

# SCALA 学习笔记

---

官网：<http://www.scala-lang.org/>

官网 API 文档 <http://www.scala-lang.org/api/2.11.8/#scala.package>

学习网站 <http://www.runoob.com/scala/scala-tutorial.html>



## 1 基础概念

---

### 1. 基本说明：Object-Oriented Meets Functional 面向对象和编程的语言

a>Scala 运行在 Java 虚拟机上，并兼容现有的 Java 程序。

b>Scala 源代码被编译成 Java 字节码，所以它可以运行于 JVM 之上，并可以调用现有的 Java 类库。

c>Scala 编译器是智能的,静态类型。大多数时候,你不需要告诉它你的变量的类型。相反,其强大的类型推断将为你计算出来。

d>在 Scala 中, `Future` 和 `promis` 可用于异步处理数据,使其更容易并行化甚至分发应用程序。

e> 在 Scala 中,可以将多个特征混合到一个接口结合和他们的实现类。

f> case 类被用来做模式匹配

测试代码：如下

```
class Person {
  //person类的两个继承类包含 name和sno字段
  //定义一个继承student类
  case class Student(name:String,sno:Int) extends Person
  //定义一个继承teacher类
  case class Teacher(name:String,tno:Int) extends Person
  //定义一个none
  case class None(name:String) extends Person

  //case 模式匹配
  object CaseClassTest {
    def caseClassMatch(p:Person) = p match{
      case Student(name,sno) => println(name + " is a student,no is:" + sno)
      case Teacher(name,tno) => println(name + " is a teacher,no is:" + tno)
      case None(name) => println("None matched")
    }

    def test(){
      val p = Student("yy",999999)
      caseClassMatch(p)
    }
  }
}
```

```
3 object Test2 {
4   //object下的成员默认都是静态的
5   def apply() = {
6     println("object person--apply()...")
7     new Person().CaseClassTest.test()
8   }
9   def main(args: Array[String]): Unit = {
10    //静态调用
11    apply()
12  }
13 }
```

Problems Tasks Console

```
<terminated> Test2$ [Scala Application] C:\Program Files\Java\jre7\bin\javaw.exe (2016年8月8日 上午11:1
object person--apply()...
yy is a student,no is:999999
```

**注：**Future 模式的核心在于：去除了主函数的等待时间，并使得原本需要等待的时间

段可以用于处理其他业务逻辑 → 个人：就像商品订单

Promis 异步模式 代表了一种可能会长时间运行而且不一定必须完整的操作的结果。这种模式不会阻塞和等待长时间的操作完成，而是返回一个代表了承诺的（promised）结果的对象 ->如 nodejs

## 2. 函数式编程有两种指导理念：

- 第一种理念是函数是**头等值**：函数作为头等值这种理念简化了操作符的抽象和新控制结构的创建。
- 函数式编程的第二种理念是程序的操作应该把输入值映射为输出值而不是就地修改数据：**不可变**数据结构是函数式语言的一块基石。

## 3. 简单的规则：如果方法使用**操作符来标注**，那么左操作符是方法的调用者 —— 除非——方法名以**冒号**结尾。这种情况下，方法被右操作数调用。

## 4. Scala 的一个**基本规则**：方法若只有一个参数，调用的时候就可以省略点及括号。

**注意**：这个语法只有在明确指定方法调用的接受者时才有效。

例如：不可以写成 “println 10” ，但是可以写成 “Console println 10” 。

## 5. 从技术层面上来说，Scala 没有操作符重载，因为它根本没有传统意义上的操作符。

## 6. 与 Java 相比，Scala 鲜有特例。与 Scala 其他的类一样，数组也只是类的实例。

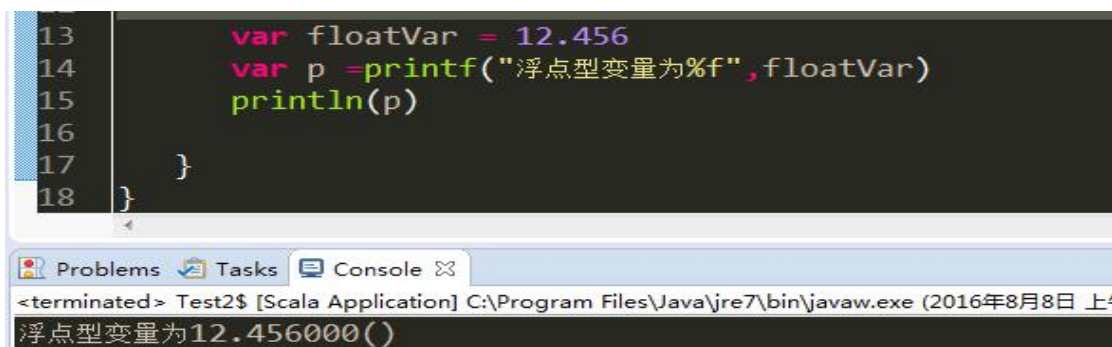
## 7. 一个**原则**：任何对于对象的**值参数应用**将都被转换为对 **apply** 方法的调用。当然前提是这个类型实际定义过 apply 方法。所以这不是特例，而是**通用法则**。

## 8. 与之类似的是，当对带有括号并包括一到若干参数的变量**赋值**时，编译器将使用对象的 **update** 方法对括号里的参数（索引值）和等号右边的对象执行调用。

array(0) = “hello” 将被转化为： array.update(0, “hello” )。

## 9. Scala **编译器**会尽可能在编译完成的代码里利用 Java 数组、原始类型和原生的数值计算方法。

10. 方法没有副作用是函数式风格编程的重要理念，计算并返回值应该是方法唯一的目的。
11. 变长参数列表，也可以称为重复参数（repeated parameters）。
12. Scala 的每个源文件都隐含了对包 java.lang、包 scala，以及单例对象 Predef 的成员引用。
13. Scala 本身没有 string 类 类 StringBuilder 和 String 均引用 Java 的 stringBuffer 和 string。其中 Scala 的 printf（）可以自动格式化字符串



```
13      var floatVar = 12.456
14      var p = printf("浮点型变量为%f", floatVar)
15      println(p)
16
17    }
18  }
```

Problems Tasks Console

<terminated> Test2\$ [Scala Application] C:\Program Files\Java\jre7\bin\javaw.exe (2016年8月8日 上

浮点型变量为12.456000()

## 1.1 BY-NAME 与 BY-VALUE

1. 如果操作符都只是方法的话，短路机制（关系运算符）是怎么工作的？

所有的 Scala 方法都有延迟其参数评估乃至取消其评估的机制，被称为传名参数（by-name parameter）。

个人-扩展：Cousera 的函数式编程原理公开课程。博客资料地址

<http://blog.csdn.net/i6448038/article/details/51344216>

by-name 、 by-value 对比。

## 1.2 对象相等性 ( SCALA 与 JAVA ) 的差异

---

《Scala 编程》：P90

1. Java 里的 == 既可以比较原始类型也可以比较引用类型。
  - 原始类型：Java 的 == 比较值的相等性，与 Scala 一致。
  - 引用类型：Java 的 == 比较了引用相等性 ( reference equality )，也就是说比较的是这两个变量是否都指向 JVM 堆里的同一个对象。Scala 也提供了这种机制，名字是 eq。不过，eq 和它的反义词 ne，仅仅应用于可以直接映射到 Java 的对象。
2. 自动检查机制不会检查右侧的参数，但是任何合理的 equals 方法都应在参数为 null 的时候返回 false。

## 1.3 源文件

---

1. Scala 对于源文件的命名没有硬性规定。然而通常情况下如果不是脚本，推荐的风格是 Java 那样按照所包含的类名来命名文件。
2. 脚本必须以结果表达式结束 —— 如果以定义结尾则不是脚本，此时以脚本方式执行时，Scala 解释器将会报错说不是以结果表达式结束的。

## 1.4 分号推断

---

1. 分号推断：Scala 程序里，语句末尾的分号通常是可选的。
  - 愿意可以加；
  - 如果一行里仅有一个语句时可以不加；
  - 如果一行包含多条语句时，分号则是必须的。

跨越多行的语句时，多数情况无须特别处理，Scala 将在正确的位置分割语句。

Scala 通常的风格是把操作符放在行尾而不是行头。

## 2. 分号推断的规则

分割语句的具体规则既出人意料地简单又非常有效。那就是，除非以下情况的一种成立，否则行尾被认为是一个分号：—— 即以下情况时，行尾不是分号.....

- 疑问行由一个不能合法作为语句结尾的字结束，如句点或中缀操作符。
- 下一行开始于不能作为语句开始的词。
- 行结束于括号(...)或方框[...]内部，因为这些符号不可能容纳多个语句。

**ps**：行尾、行首是否合法，以及是否在**配对符号**中间断开。

## 1.5 常量

---

Scala 与 Java 的习惯不一致的地方在于**常量名**。

Scala，constant 不等同于 val。

Java 中，习惯上常量名全部大写，Scala 中习惯只是第一个字母必须大写。即其常数也用驼峰式风格。

## 1.6 等效推论 ( EQUATIONAL REASONING )

---

1. 等效推论：在表达式没有副作用的前提下，引入的变量等效于计算它的表达式。因此，无论何时都可以用表达式替代变量名。
2. 使用 val 而不是 var 的一个好处：它能更好地支持等效推论。

---

## 1.7 函数

---

1. Scala 提供了许多 Java 中没有的**定义函数**的方式：除了成员函数外，还有内嵌在函数中的函数，函数字面量和函数值。
2. 定义在函数内的函数：这种**本地函数**仅在包含它的代码块中可见。

本地函数可以访问包含它的函数的参数 —— 对外层函数入参的直接使用。

3. **函数字面量**：被编译进类，并在运行期实例化为函数值（function value）。

因此，函数字面量和值的区别在于函数字面量存在于源代码，而函数值作为对象存在于运行期。

这个区别很像类（源代码）和对象（运行期）之间的关系。

如**函数字面量** `(x: Int) => x + 1`：指明这个函数把左边的东西转变为右边的东西。

4. 任何函数值都是某个扩展了 scala 包的若干 FunctionN 特质之一的类的实例。

5. 函数字面量的**短格式**：

- 一种方式是去除参数类型 —— 目标类型化：target typing，因为表达式的目标使用影响了表达式的类型化。

`xxx.filter((x) => x > 0)` 其中 x 的类型被省略。

- 某些参数的类型是被推断的，省略其外的括号，是第二种去除无用字符的方式。

Email : [bugcoder@163.com](mailto:bugcoder@163.com) 小胖 Q:294027471 欢迎加入群 413409965



`xxx.filter(x=>x>0)` 其中 x 两边的括号不是必须的。

- 第三种方式：可以把下划线当做一个或更多参数的占位符，主要每个参数在函数数字面量内进出现一次。

`xxx.filter(_>0)` 。

使用下划线当做参数的占位符时，有时编译器可能无法推断缺失的类型参数，这时可以使用冒号指定类型。

`val f = (_:Int) + (_: Int) : _+_` 将扩展成带两个参数的函数数字面量。

6. 高阶函数的好处之一是它们能让你创造控制抽象从而减少代码重复。
7. 高阶函数的另一个重要应用是把它们放在 API 里使客户代码更简洁。
8. Scala 不允许在运行期粘合代码。
9. 在拥有头等函数的语义中，即使语言的语法是固定的，你也可以有效地制作新的控制结构。所有你需要做的就是创建带有函数做参数的方法。
10. 借贷模式 ( loan pattern ) : 《Sccala 编程》P141
11. Scala 的任何方法调用，如果你确定只传入一个参数，就能可选地使用花括号替代小括号包围参数。

在传入一个参数时，可以用花括号替代小括号的机制，其目的是让客户程序员能写出包围在花括号内的函数数字面量。这可以让方法调用感觉更像控制抽象。

---

### 1.7.1 函数是第一类值

---



1. 在 Scala 中，第一类函数：first-class function。你不仅可以定义函数和调用它们，还可以把函数写成没有名字的文本：literal 并把它们像值：value 那样传递

因此函数文本和值的区别在于函数文本存在于源代码，而函数值存在于运行期对象。

这个区别很像类（源代码）和对象（运行期）的那样

```
(x: Int) => x + 1
```

=>指明这个函数把左边的东西（任何整数 x）转变成右边的东西（x + 1）。所以，这是一个把任何整数 x 映射为 x + 1 的函数。

```
var func = (x: Int) => x + 1  
println(func(1)) // --> 2
```

因为 func 是 var，你可以在之后重新赋给它不同的函数值

如果你想在函数文本中包括超过一个语句，用大括号包住函数体，一行放一个语句，就组成了一个代码块。与方法一样，当函数值被调用时，所有的语句将被执行，而函数的返回值就是最后一行产生的那个表达式

```
var func = (x: Int) => {  
    print("test")  
    x+1  
}  
println(func(1)) // --> test2
```

---

### 1.7.2 函数文本的短方式

---

Scala 提供了许多方法去除冗余信息并把函数文本写得更简短

```
val someNumbers = List(-11, -10, -5, 0, 5, 10)  
var s = someNumbers.filter((x: Int) => x > 0)  
println(s) // --> List(5, 10)
```

Scala 编译器知道  $x$  一定是整数，因为它看到你立刻使用了这个函数过滤整数列表（由 `someNumbers` 暗示）。这被称为**目标类型化**：，因为表达式的目标使用 `someNumbers.filter()` 的参数——影响了表达式的类型化——决定了  $x$  参数的类型。

```
var s1=someNumbers.filter(x => x > 0)
print(s1) //-->List(5, 10)
```

去除无用字符的方式是省略类型是被推断的参数之外的括号。前面例子里， $x$  两边的括号不是必须的

---

### 1.7.3 占位符

---

如果想让函数文本更简洁，可以把下划线当做一个或更多参数的占位符，只要每个参数在函数文本内仅出现一次。

比如， $\_ > 0$  对于检查值是否大于零的函数来说就是非常短的标注：

```
var s2= someNumbers.filter(_ > 0)
print(s2)//-->List(5, 10)
```

等价于

```
var s3= someNumbers.filter(x => x > 0)
print(s3)//-->List(5, 10)
```

**Ps:**可以把下划线看作表达式里需要被“填入”的“空白”。这个空白在每次函数被调用的时候用函数的参数填入

---

### 1.7.4 偏应用函数 PARTIAL FUNCTION

---

1. 尽管前面的例子里下划线替代的只是单个参数，你还可以使用一个下划线替换整个参数列表。

例如，写成 `println(_)`，或者更好的方法你还可以写成 `println _`

```
someNumbers.foreach(println _) //-->-11-10-50510
someNumbers.foreach(x => println(x)) //-->-11-10-50510
```

以这种方式使用下划线时，你就正在写一个偏应用函数：partially applied function。

Scala 里，当你调用函数，传入任何需要的参数，你就是在把函数应用到参数上

2. 偏应用函数是一种表达式，你不需要提供函数需要的所有参数。代之以仅提供部分，或不提供所需参数

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

```
val a = sum _ //等价于 sum(1, 2, 3)
print(a(1,2,3)) //-->6
```

**Q:为什么要使用尾下划线？**

Scala 的偏应用函数语法凸显了 Scala 与经典函数式语言如 Haskell 或 ML 之间，设计折中的差异。在经典函数式语言中，偏应用函数被当作普通的例子。更进一步，这些语言拥有非常严格的静态类型系统能够暴露出你在偏应用中可能犯的所有错误。Scala 与指令式语言如 Java 关系近得多，在这些语言中没有应用所有参数的方法会被认为是错误的。进一步说，子类型推断的面向对象的传统和全局的根类型接受一些被经典函数式语言认为是错误的程序。

举例来说，如果你误以为 List 的 `drop(n: Int)` 方法如 `tail()`，那么你会忘记你需要传递给 `drop` 一个数字。你或许会写，“`println(drop)`”。如果 Scala 采用偏应用函数在哪儿都 OK 的经典函数

式传统，这个代码就将通过类型检查。然而，你会惊奇地发现这个 `println` 语句打印的输出将总是 `<function>`！可能发生的事情是表达式 `drop` 将被看作是函数对象。因为 `println` 可以带任何类型对象，这个代码可以编译通过，但产生出乎意料的结果。

为了避免这样的情况，Scala 需要你指定显示省略的函数参数，尽管标志简单到仅用一个 `'_'`。

Scala 允许你仅在需要函数类型的地方才能省略这个仅用的 `_`。

---

### 1.7.5 柯里化函数 CURRYING

---

柯里化是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术

```
//普通的非柯里化的函数定义
// println(oldSum(1,2)) -->3
def oldSum(x:Int,y:Int):Int={
  return x+y
}
//柯里化的函数定义
//原来函数使用一个参数列表,柯里化把函数定义为多个参数列表
//println(newSum(1)(2)) -->3
def newSum(x:Int)(y:Int):Int={
  return x+y
}
```

```
//模拟柯里化函数
//定义一个firstSetup
// println(firstSetup(1)) --><function1>
def firstSetup(x:Int)=(y:Int)=> x+y
// println(secondSetup) --><function1>
//使用参数1调用这个函数来生成第二个函数
val secondSetup=firstSetup(1)
//println(secondSetup(2))-->3
```

PS : 类似闭包

```
//使用柯里化函数 生成函数one ,下划线“_” 作为第二参数列表的占位符
//println(one) --><function1>
val one = newSum(1)_
//println(one(2)) -->3
```

通过柯里化，可以定义多个类似 one 的函数

### 1.7.6 传名参数 ( BY-NAME PARAMETER )

1. 要实现一个传名函数，要定义参数的类型开始于=> 而不是()=>。
2. 传名类型中，空的参数列表，(),被省略，它仅在参数中被允许。
3. 没有什么传名变量或传名字段这样的东西。

### 1.7.7 闭包 CLOSURE

```
var more =1
var func= (x: Int) => x + more
println(func(1)) // --> 2
```

依照这个函数文本在运行时创建的函数值（对象）被称为闭包：\_。名称源自于通过“捕获”

自由变量的绑定对函数文本执行的“关闭”行动。不带自由变量的函数文本，如(x: Int) =>

x + 1，被称为**封闭术语**：\_，这里**术语**：指的是一小部分源代码。因此依照这个函数文本

在运行时创建的函数值严格意义上来讲就不是闭包，因为(x: Int) => x + 1 在编写的时候就

已经封闭了。但任何带有自由变量的函数文本，如`(x: Int) => x + more`，都是开放术语：

…。因此，任何依照`(x: Int) => x + more` 在运行期创建的函数值将必须捕获它的自由变量，`more`，的绑定。由于函数值是关闭这个开放术语`(x: Int) => x + more` 的行动的最终产物，得到的函数值将包含一个指向捕获的 `more` 变量的参考，因此被称为闭包。

```
var more = 1
var func = (x: Int) => x + more
println(func(1)) // --> 1+1=2
more = 10
println(func(1)) // --> 1+10=11
```

依照`(x: Int) => x + more` 创建的闭包看到了闭包之外做出的对 `more` 的变化。反过来也一样。闭包对捕获变量作出的改变在闭包之外也可见。

---

## 1.8 重复参数

1. Scala 中，你可以指明函数的最后一个参数是重复的。在参数的类型之后放一个星号。
2. 函数内部，重复参数的类型是声明参数类型的数组。
3. 如果要将数组传入重复参数，需要在数组参数后添加一个冒号和一个 `_*` 符号。这个标注告诉编译器把该传入数组的每个元素当做参数，而不是当作单一的参数。

---

## 1.9 尾递归



1. 在最后一个动作调用自己的函数，被称为尾递归：tail recursive。
2. Scala 编译器检测到尾递归就用新值更新函数参数，然后把它替换成一个回到函数开通的跳转。
3. 递归经常是比基于循环的更优美和简明的方案。
4. 如果方案是尾递归，就无须付出任何运行期开销。
5. 尾递归函数追踪

尾递归函数将不会为每个调用制造新的堆栈结构；所有的调用将在一个结构内执行。

这可能会让检查程序的堆栈跟踪信息并失败的程序员感到惊奇。

**尾递归开关项**：-g:notailcalls

传递该参数到 scala 的 shell 或者 scalac 编译器。

6. 尾递归的局限：Scala 里尾递归的使用局限很大，因为 JVM 指令集使实现更加先进的尾递归形式变得很困难。
7. Scala 仅优化了直接递归调用使其返回同一个函数。如果递归是间接的，就没有优化的可能性了。
8. 同样，如果最后一个调用是一个**函数值**你也不能获得尾递归调用优化。
9. 尾调用优化**限定**了方法或嵌套函数必须在最后一个操作调用本身，而不是转到某个函数值或什么其他的中间函数的情况。

---

## 1.10 不可变对象的权衡

---

不可变对象提供了若干强于可变对象的有点和一个潜在的缺点。



## 1. 优点

- 不可变对象常常比可变对象更易理清头绪，因为它们没有随着时间变化的复杂的状态空间。
- 可以很自由地传递不可变对象，但对于可变对象来说，传递之前需建副本以防万一。
- 一旦不可变对象完成构造之后，就不会有线程因为并发访问而破坏对象内部状态，因为根本没有线程可以改变不可变对象的状态。
- 不可变对象让哈希表键值更安全。比方说，如果可变对象在进入 HashSet 之后被改变，那么你下一次查找这个 HashSet 时就找不到这个对象了。

2. 缺点：有时需要复制很大的对象表而可变对象的更新可以在原址发生。

## 2 数组

---

1. Scala 数组是**可变**的同类对象序列。尽管数组在实例化之后**长度固定**，但它的元素值却是可变的。所以说数组是可变的。

## 3 列表

---

1. Scala 列表类 ( List ) 是**不可变**的同类对象序列。实际上，Scala 的列表是为了实现函数式风格的编程而设计的。
2. 列表类最常用的操作符或许是 “::” ，发音为 “cons” 。可以把新元素组合到现有列表的最前端，然后返回作为执行结果的新列表。

3. Nil : 是空列表的简写。
4. 要在最后用到 Nil 的理由是::是定义在 List 类上的方法。
5. 列表为什么不支持添加 ( append ) 操作 ?

List 类没有提供 append 操作 , 因为随着列表边长 , append 的耗时将呈线性增长 , 而使用::做前缀则仅耗用固定的时间。

6. ListBuffer : 一种提供 append 操作的可变列表 , 完成之后调用 toList。
7. 列表与数组的两点重要差别 :

- 首先 , 列表是不可变的。不能通过赋值改变列表的元素 ;
- 其次 , 列表具有递归结构 , 而数组是连续的。

8. 列表是同质 ( homogeneous ) 的 : 列表的所有元素都具有相同的类型。

9. Scala 的列表类型是协变 ( covariant ) 的。

10. 空列表的类型为 List[Nothing]。

11. 所有的列表都是由两个基础构造块 Nil 和:: ( 发音为 “cons” ) 构造出来的。

12. Nil 代表空列表。

13. 对于列表的所有操作都可以表达为以下三种形式 :

- head : 返回列表的第一个元素 ;
- tail : 返回除第一个之外所有元素组成的列表 ;
- isEmpty : 如果列表为空 , 则返回真。

14. head 和 tail 方法仅能够作用在非空列表上。如果执行在空列表上 , 会抛出异常。

15. List 的模式匹配 :

- List(...) : 是由开发库定义的抽取器 ( extractor ) 模式的实例 ;

- “cons” 模式 `x::xs` 是中缀操作符模式的特例。即中缀操作符会被当做构造器模式。

16. Scala 中存在两个 `::`，一个存在于 `scala` 包中，另一个是 `List` 类的方法。

17. `::` 方法的目的是实例化 `scala::` 的对象。

18. 需要遍历整个列表的方法：

- `Length`：会遍历整个列表
- `Last`：返回（飞空）列表的最后一个元素；
- `init`：返回除了最后一个元素之外余下的列表。

19. 组织好数据，以便让所有的访问都集中在列表的头部，而不是尾部。

20. `reverse`：反转列表——频繁访问尾部时。

21. `drop` 和 `take` 操作泛化了 `tail` 和 `init`，它们可以返回列表任意长度的前缀或后缀。

22. `splitAt`：操作在指定位置拆分列表，并返回对偶（`pair`）列表。

23. `apply` 方法：实现了随机元素的选择；

24. `indices` 方法：可以返回指定列表的所有有效索引值组成的列表。

25. `zip`：操作可以把两个列表组成一个对偶列表；如果两个列表的长度不一致，那么任何不能匹配的元素将被丢掉。

26. `zipWithIndex` 方法：把列表的每个元素与元素在列表中的位置值组成一对。

27. `toString` 方法：返回列表的标准字符串表达形式；

28. `mkString` 方法：其他表达形式；

29. `addString` 变体：把构建好的字符串添加到 `StringBuilder` 对象中。

30. `elements`、`toArray`、`copyToArray`：

- 31. List 类的 toArray 和 Array 类的 toList 方法：让数据存储格式在连续存放的数组和递归存放的列表之间进行转化。
- 32. copyToArray：把列表元素复制到目标数组的一段连续空间。必须确保目标数组有足够的空间可以全部放下列表元素。
- 33. Elements：用枚举器访问列表元素；
- 34. 插入排序与归并排序 ( merge sort )

## 4 元组

---

- 1. 元组也是不可变的；
- 2. 与列表不同，元组可以包含不同类型的元素。
- 3. 元组实例化之后，可以用点号、下划线和基于 1 的索引访问其中的元素。其中点号“.”与访问字段或调用方法的点没有区别。
- 4. 元组的实际类型取决于它含有的元素数量和这些元素的类型。
- 5. 访问元组的元素：

你或许想知道为什么不能用访问列表的方法来访问元组，如 pari(0)。那是因为列表的 apply 方法始终返回同样的类型，但元组的类型不尽相同。\_1 的结果类型可能与 \_2 的不一致。诸如此类，因此两者的访问方法也不一样。

另外，这些 \_N 的索引是基于 1 的，而不是基于 0 的，这是因为对于拥有静态类型元组的其他语言，如 Haskell 和 ML，从 1 开始是传统的设定。

## 5 集(SET)和映射(MAP)

---

1. Scala 致力于充分利用函数式和指令式风格两方面的好处，因此它的集合 ( collection ) 库区分为可变类型和不可变类型。
2. 对于 set 和 map 来说，Scala 同样有可变的和不可变的，不过并非各提供两种类型，而是通过类继承的差别把可变性差异蕴含其中。
3. 只有可变集提供的才是真正的 += 方法，不可变集不是。
4. set 类层级

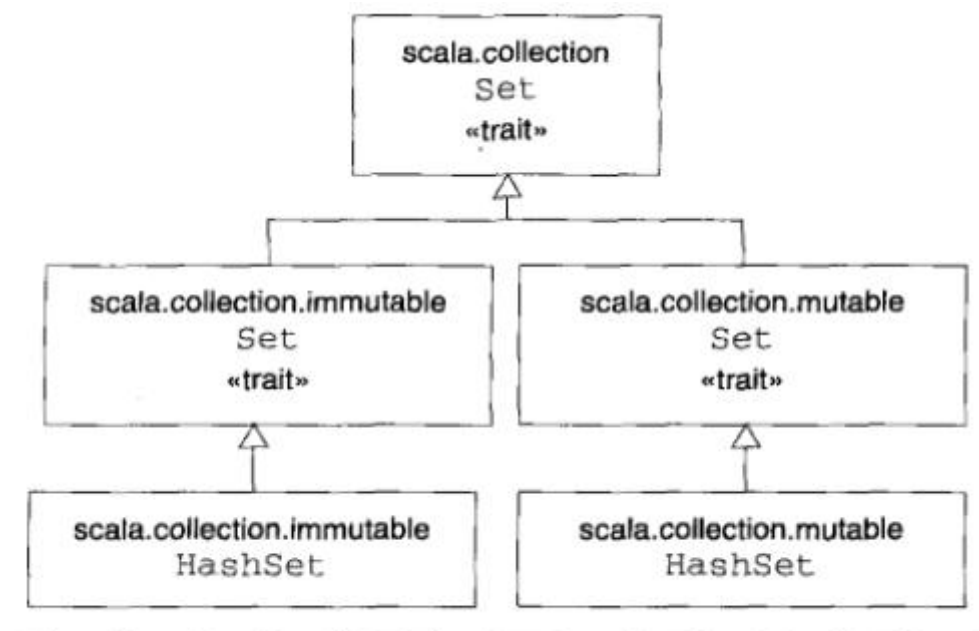


图 3.2 Scala 的 set 类层级

5. map 类层级

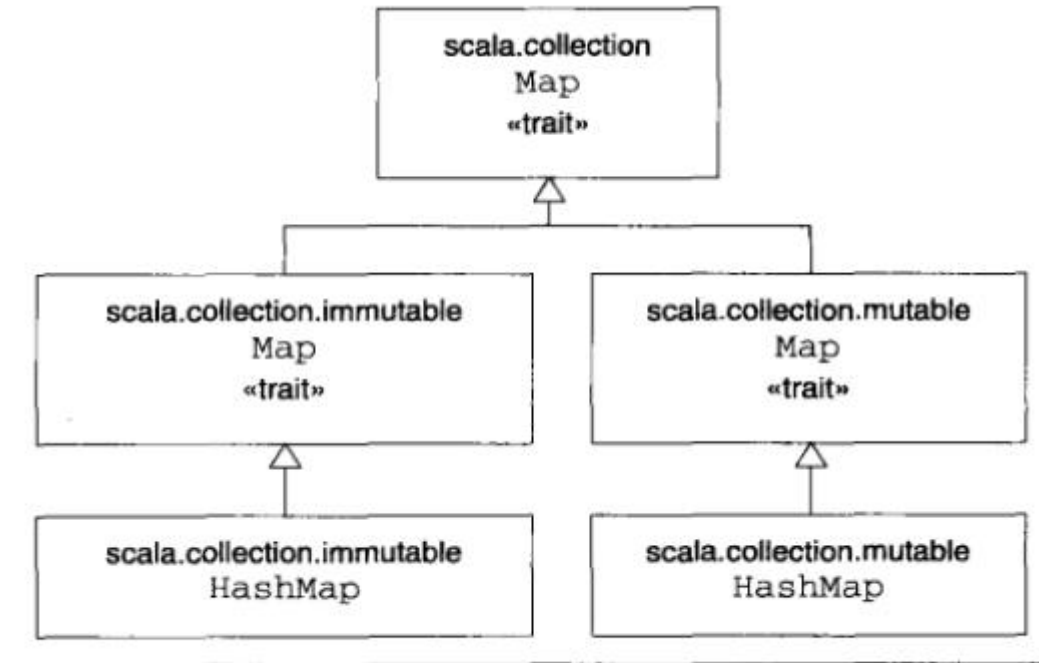


图 3.3 Scala 的 map 类层级

6. 向函数式风格转变的方式之一，就是尝试不用任何 var 编程。
- 7.
- 8.
- 9.

## 6 文件

---

1. `scala.io.Source`

## 7 注解

---

1. 注解可以在程序中的各项条目添加信息，这些信息可以被编译器或外部工具处理。
2. 注解是插入到代码中以便有工具可以对它们进行处理的标签。工具可以在代码级别运作，也可以处理被编译器加入了注解信息的类文件。

3. 可以对 Scala 类使用 Java 注解。也可以使用 Scala 特有的注解，通常由 Scala 编译器或编译器插件处理。
4. Java 注解并不影响编译器如何将源码翻译成字节码，仅仅往字节码中添加数据，以便外部工具可以利用它们。而在 Scala 中，注解可以影响编译过程，比如 `@BeanProperty` 注解。
5. Scala 中可以为类、方法、字段、局部变量和参数添加注解，与 Java 一样。
6. 可以同时添加多个注解，先后顺序没有影响。

```
@Entity class Credentials
@Test def testSomeFeature() {}
@BeanProperty var username = _
def doSomething(@NotNull message: String) {}
@BeanProperty @Id var username = _
```

7. 给主构造器添加注解时，需要将注解放置在构造器之前，并加上一对圆括号：

```
class Credentials @Inject() (var username: String, var password: String)
```

8. 为表达式添加注解，在表达式后加上冒号，然后是注解：

```
(myMap.get(key): @unchecked) match { ... }
```

9. 为类型参数添加注解：

```
class MyContainer[@specialized T]
```

10. 针对实际类型的注解应放置在类型名称之后：

```
String @c 个人[Unit]
```

11. 这个暂时不知道是什么东西，`@c 个人` 注解将会在 22 章介绍。

## 7.1 注解参数



1. Java 注解可以有带名参数。如果参数的名字是 value，参数名可以略去。如果注解不带参数，圆括号可以省去。大多数注解参数带有缺省值。
2. Java 注解的参数类型是有规定的，但 Scala 中参数可以是任何类型，不过只有很少几个注解使用了。

## 7.2 针对 JAVA 特性的注解

---

1. 不常用的 Java 特性，Scala 提供了相应的注解。

@volatile 注解标记为易失的；@transient 注解将字段标记为瞬态的；@strictfp 注解对应 strictfp 修饰符；@native 注解标记在 C 或 C++ 代码中实现的方法，对应 native 修饰符。

2. Scala 使用@cloneable 和@remote 注解来代替 Cloneable 和 java.rmi.Remote 标记接口。
3. Java 编译器会跟踪受检异常。那么如果从 Java 代码中调用 Scala 的方法时，需要包含那些可能抛出的受检异常。这时，需要使用@throws 注解来生成正确的签名：

```
class Book {  
    throws(classOf[IOException]) def read(filename: String) { ... }  
    ...  
}  
  
// 对应的 Java 代码  
void read(String filename) throws IOException
```

4. 使用@varargs 注解可以让 Java 调用 Scala 中带有变长参数的方法。

## 7.3 用于优化的注解

---

### 7.3.1 尾递归

1. 递归有时能被转化成循环，可以节省栈空间。在函数式编程中这很重要，因为通常会使用递归方法来遍历集合。
2. 尾递归计算过程的最后一步是递归调用同一个方法，可以变换成跳回到方法顶部的循环。比如：

```
def sum(xs: Seq[Int], partial: BigInt): BigInt =  
  if (xs.isEmpty) partial else sum(xs.tail, xs.head + partial)
```

3. 上面的代码中 Scala 会自动尝试使用尾递归优化。不过有的时候可能会因为某些原因使得编译器无法进行优化。如果需要依赖于编译器去掉递归，给方法加上 `@tailrec` 注释。这样的话，如果编译器无法应用递归优化，就会报错。
4. 对于消除递归，一个更加通用的机制叫“蹦床”。蹦床会执行一个循环，不停地调用函数，每个函数都返回下一个将被调用的函数。尾递归是一个特例，每个函数都返回它自己。Scala 有一个名为 `TailCalls` 的工具对象，可以帮助实现蹦床。相互递归的函数返回类型为 `TailRec[A]`，要么返回 `done(result)`，要么返回 `tailcall(fun)`，`fun` 是下一个要被调用的函数。这个函数必须是不带额外参数且同样返回 `TailRec[A]` 的函数。示例：

```
import scala.util.control.TailCalls._  
  
def evenLength(xs: Seq[Int]): TailRec[Boolean] =  
  if (xs.isEmpty) done(true) else tailcall(oddLength(xs.tail))  
  
def oddLength(xs: Seq[Int]): TailRec[Boolean] =  
  if (xs.isEmpty) done(false) else tailcall(evenLength(xs.tail))  
  
// 获取结果使用 result 方法  
evenLength(1 to 1000000).result
```

### 7.3.2 跳转表生成与内联

1. 在 C++ 或 Java 中，switch 语句通常可以被编译成跳转表，这比一系列的 if/else 表达式更加高效。Scala 也会尝试对匹配语句生成跳转表。而 @switch 注解可以检查 match 语句是不是真的被编译成了跳转表。

```
(n: @switch) match {  
  case 0 => "Zero"  
  case 1 => "One"  
  case _ => "?"  
}
```

2. 方法内联是另一个常见的优化。内联将方法调用语句替换为被调用的方法体（减少了调用的开支，应该相当于 C++ 中的 inline 函数吧，适合于小方法）。
3. 使用 @inline 来建议编译器做内联，或者使用 @noinline 来告诉编译器不要内联。

### 7.3.3 可省略方法

1. @elidable 注解给可以在生产代码中移除的方法打上标记。这个等到使用时再回来查看详细吧。

### 7.3.4 基本类型的特殊化

1. 打包和解包基本类型的值不高效，不过这样的操作在泛型代码里很常见。当使用泛型代码时，可以使用 @specialized 注解来使编译器自动生成基本类型的重载版本。

```
def allDifferent[@specialized(Long, Double) T](x: T, y: T, z: T) = ...
```

2. @specialized 注解后的括号内，是指定特殊化的基本类型。如上面的例子，特殊化了 Long 和 Double 两种类型。如果没有括号及括号内的内容，则是全部生成。

## 7.4 用于错误和警告的注解

---

1. 加上了@deprecated 注解的特性如果被使用，编译器会生成一个警告信息。
  2. @implicitNotFound 注解用于当某个隐式参数不存在的时候生成有意义的错误提示，详细参见 21 章。
  3. @unchecked 注解用于在匹配不完整时取消警告信息。
  4. @uncheckedVariance 注解会取消与型变相关的错误提示。
- 

## 8 内建控制结构

---

1. Scala 内建的控制结构：仅有 if、while、for、try、match 和函数调用。  
如此少的理由是，Scala 从语法层面支持函数式编程。
2. 几乎所有的 Scala 的控制结构都会产生某个值。这是函数式语言所采用的方式。
3. While 和 do-while 结构之所以被称为“循环”，而不是表达式，是因为它们不能产生有意义的结果。结果的类型是 Unit，是表明存在并且唯一存在类型为 Unit 的值，称为 unit value，写成()。  
与此类似：对 var 再赋值等式本身也是 Unit 值。
4. ()的存在是 Scala 的 Unit 不同于 Java 的 void 的地方。

《Scala 编程》P107

## 9 隐式转换

---

### 9.1 基础知识

---

1. 要隐式转换起作用，需要定义在作用范围之内。
- 2.

## 10 编译与运行

---

### 1. 编译：

- `scalac xxx.scala yyy.scala` ： 编译
- Scala 发布包中包含了一个叫做 **fsc**( 快速 Scala 编译器 ,fast Scala compiler ) 的 Scala 编译器后台服务 ( daemon ) 。

使用方法：`fsc xxx.scala yyy.scala`

第一次执行 `fsc` 时，会创建一个绑定在你计算机端口上的本地服务器后台进程。然后它就会把文件列表通过端口发送给后台进程，由后台进程完成编译。

下一次执行 `fsc` 时，检测到后台进程已经在运行了，于是 `fsc` 将只把文件列表发给后台进程，它会立即开始编译文件。

使用 `fsc`，只需在首次运行的时候等待 Java 运行环境的启动。

停止 `fsc` 后台进程：`fsc -shutdown`

两种方式，最终都将完成 Java 类文件的创建，然后可以用 scala 命令，像之前的例子里调用解释器那样运行它。但这里用的是包含了正确签名的 **mian 方法的独立对象名**。

如 scala xxx params

2. scala xxx.scala params : 运行

Scala 程序用来“解释”Scala 源文件的真正机制是，它把 Scala 源码编译成字节码，然后立即通过类装载机装载它们，并执行它们。

---

## 10.1 SCALA 的标识符

---

1. 构建方式：

- 字母数字式：字母数字标识符 ( alphanumeric identifier ) 以字母或下划线开始，之后可以跟字母、数字或下划线。
- 操作符标识符 ( operator identifier ) 由一个或多个操作符字符组成。
- 混合标识符 ( mixed identifier )：由字母数字组成，后面跟着下划线和一个操作符标识符。

如 **unary\_+** 被用作定义一元的 “+” 操作符的方法名。

或 **myvar\_** 被用作定义 **赋值操作符** 的方法名。该混合标识符格式是由 Scala 编译器产生的用来支持属性 ( property ) 的。

- **字面量标识符** ( literal identifier )：用反引号 `...` 包括的任意字符串。

如：``x`` ``<clinit>`` ``yield``

思路是你可以把运行时环境认可的任意字符串放在反引号之间当做标识符。

在 Java 的 Thread 类中访问静态的 yield 方法是它的典型用例。你不能写 Thread.yield()，因为 yield 是 Scala 的保留字。但可以在反引号里引用方法的名称，例如：`Thread.`yield`()`。

2. Scala 遵循 Java 的驼峰式标识符习惯。
3. Scala 编译器将在内部“粉碎”操作符标识符以转换成合法的内置 '\$' 的 Java 标识符。如标识符:->将被内部表达为 \$colon\$minus\$greater。若你想从 Java 代码访问这个标识符，就应使用这种内部表达式方式。
4. 注意：  
不建议在标识符结尾使用下划线。比如：“val name\_: Int = 1”，将导致编译器错误。编译器会认为你在定义名为“name\_.”的变量。需要在冒号前插入额外的空格，才能编译通过。
- 5.

---

## 10.2 SCALA 表达式的操作符

---

---

### 10.2.1 优先级和关联性

---

《Scala 编程》：P90。

1. 操作符标注不仅限于像+这种其他语言里看上去像操作符的东西，任何方法都可以被当做操作符来标注。



Scala 里的操作符不是特殊的语法：任何方法都可以是操作符。到底是方法还是操作符取决于你如何使用它。即是否以操作符标注方式使用它。



## 2. 操作符标注方式：《Scala 编程》P85

- 前缀标注：这些前缀操作符与中缀操作符一致，是值类型对象调用方法的简写形式。然而这种情况下，**方法名**在操作符字符上前缀 **"unary\_"**。

标识符中能作为**前缀操作符**用的**只有** +、-、!和~。因此，如果对类型定义了名为 unary\_!的方法，就可以对值或变量用!p 这样的前缀操作符方式调用方法。

如果定义名为 unary\_\*的方法，由于\*不是**四种**可以当做前缀操作符用的标识符之一，因此没有办法将其用成前缀操作符。—— \*p，Scala 会理解成\*.p。

可以通过操作符和显示方法名两种方式调用方法。

- 中缀标注
- 后缀标注：后缀操作符是不用**点或括号**调用的**不带任何参数**的方法。

Scala 里，方法调用的空括号可以省略。惯例是如果方法带有副作用就加上括号。

3. 要想知道 Scala 的值类型有哪些操作符，只要在 Scala 的 API 文档里查下定义在值类型上的方法即可。
4. 操作符的优先级决定了表达式的哪个部分先于其他部分被评估。
5. Scala 没有操作符，实际上，操作符只是方法的一种表达式。
6. 对于以操作符形式使用的方法，Scala 根据操作符的第一个字符判断方法的优先级（这个规则有个例外）。

表 5.3 操作符优先级

(所有其他的特殊字符)	
* / %	
+ -	
:	
= !	
<>	
&	
^	
(所有字母)	
(所有赋值操作符)	

7. 除以上优先级规则外，还有以等号结束的**赋值操作符**（assignment operator）。  
如果操作符以等号字符（=）结束，且操作符并非比较操作符，那么这个 操作符的  
优先级与赋值操作符（=）相同。也就是说，它比任何其他操作符的优先级都低。
8. 当同样优先级的多个操作符并列出现在表达式时，操作符的**关联性**（associativity）  
决定了操作符**分组**的方式。
9. Scala 里操作符的**关联性**取决于它的最后一个字符。
10. 不论操作符具有什么样的关联性，它的操作数总是从左到右**评估**的。任何以 “:” 字  
符结尾的方法由它的右操作数调用，并传入左操作符。以其他字符结尾的方法与之  
相反。
- 11.

## 11 类

---

### 11.1 基础概念

---

1. `val`、`var` : 定义成员变量。
2. `def` : 定义成员方法, 包含可执行代码。
3. 保持对象健壮性的重要方法之一就是保证对象的状态——即实例变量的值——在对象整个生命周期中持续有效。
4. 基本类型: 除了 `String` 归于 `java.lang` 包之外, 其余所有的基本类型都是包 `scala` 的成员。
5. Scala 的基本类型与 Java 的对应类型范围完全一样。
6. Scala 编译器将把类内部的任何既不是字段也不是方法定义的代码编译至主构造器中。
7. `toString` 方法: 类会继承 `java.lang.Object` 类的 `toString` 实现, 只会打印类名、`@`符号和十六进制数。
8. **先决条件**: 先决条件是对传递给方法或构造器的值的限制, 是调用者必须满足的需求。

为主构造器定义先决条件 ( `precondition` ) 。

在 `scala` 包的孤立对象 `Predef` 上定义的 `require` 方法,

9. 自指向: **关键字** `this` 指向当前执行方法被调用的对象实例, 或者如果使用在构造器里的话, 就是正被构建的对象实例。

10. 辅助构造器 ( auxiliary constructor ) : Scala 里主构造器之外的构造器被称为辅助构造器。

Scala 的辅助构造器定义**开始于** `def this(...)`。

Scala 里的每个辅助构造器的**第一个动作**都是调用同类的别的构造器。

规则的**根本结果**是：每个 Scala 的构造器调用终将结束于对主构造器的调用。因此主构造器是类的唯一入口点。

比较：

- Java 的构造器的第一个动作只有两个选择：要么调用同类的其他构造器，要么直接调用超类的构造器。
- Scala 的类里面只有主构造器可以调用超类的构造器。

11. Scala 里，对象的每个非私有的 var 类型成员变量都隐含定义了 getter 和 setter 的方法。Var 变量 x 的 getter 方法命名为 "x"，它的 setter 方法命名为 "x\_="。

字段始终被标记为 `private[this]`，表示它只能被包含它的对象访问。

但是 getter 和 setter 方法获得了原本 var 变量的可见性。

12. 字段的**初始化器** `"="_`：把零值赋给该字段。这里的“零”的取值取决于字段的类型。对于数值类型来说是 0，布尔类型是 false，引用类型则是 null。

初始化器不可以随意省略，省略后将定义为**抽象变量**，而不是未初始化的变量。

## 11.2 嵌套与作用域 ( SCOPE )

---

1. 嵌套与作用域原则可以应用于所有的 Scala 架构，包括函数。

2. 类的主构造器：

主构造器的参数在该类范围内有效，但对外无效。

添加 val、var 后，主构造器的参数会自动变成字段，此时，对外有效。

即，这些参数作为字段时才能被访问。

《Scala 编程》：P96

3. 与 Java 的差异：Scala 允许在嵌套范围内定义同名变量。

4. Scala 程序里所有的变量定义都存在有效作用范围（scope）。

5. 花括号通常引入了新的作用范围，所以任何定义在花括号里的东西超出括号之后就脱离了范围。

6. 在解释器里：概念上，解释器为每次输入的新语句都创建了新的嵌套范围。

。

### 11.3 富包装类

---

表 5.4 一些富操作

代码	结果
0 max 5	5
0 min 5	0
-2.7 abs	2.7
-2.7 round	-3L
1.5 isInfinity	false
(1.0/0)isInfinity	true
4 to 6	Range(4,5,6)
"bob" capitalize	"Bob"
"robert" drop 2	"bert"

表 5.5 富包装类

基本类型	富包装
Byte	scala.runtime.RichByte
Short	scala.runtime.RichShort
Int	scala.runtime.RichInt
Long	scala.runtime.RichLong
Char	scala.runtime.RichChar
String	scala.runtime.RichString
Float	scala.runtime.RichFloat
Double	scala.runtime.RichDouble
Boolean	scala.runtime.RichBoolean

## 11.4 SCALA 的原字符串和符号字符串。

12. Scala 为原始字符串 ( raw string ) 引入了一种特殊的语法。它以同一行里的三个引号 ( " " " ) 作为开始和结束。内部的原始字符串可以包含无论何种任意字符，包括新行、引号和特殊字符。

同时字符串类引入了 stripMargin 方法。使用的方式是，把管道符号 ( | ) 放在每行前面，然后对整个字符串调用 stripMargin。

13. **符号字面量**：被写成 ' <标识符> ，这里<标识符>可以是任何字母或数字的标识符。

这种字面量被映射成预定义类 scala.Symbol 的实例。

具体地说，就是字面量 ' cymbal 将被编译器**扩展为工厂方法**调用：

Symbol( "cymbal" )。

符号字面量典型的应用场景是在动态类型语言中实用一个标识符。

《Scala 编程》P83。

符号字面量除了显示名字之外，什么都不能做。

还有就是符号是被限定 ( interned ) 的。如果同一个符号字面量出现两次，那么两个字面量指向的是同一个 Symbol 对象。

**个人** — 扩展：参考 ruby 中的符号。

---

## 11.5 方法

---

1. 如果没有发现任何显式的返回语句，Scala 方法将返回方法中最后一次计算得到的值。

方法的推荐风格是尽量避免使用返回语句，尤其是多条返回语句。待之以把每个方法当做是创建返回值的表达式。这种**逻辑鼓励你分层简化方法**，把较大的方法分解为多个更小的方法。

2. 另一种简写方式：假如某个方法仅计算单个结果表达式，则可以**去掉花括号**。如果结果表达式很短，甚至可以把它放在 def 的同一行里。
3. 另一种表达方式是**去掉结果类型和等号**，把方法体放在花括号里。在这种形式下，方法看上去很像过程 ( procedure )，一种仅为了副作用而执行的方法。

通常我们定义**副作用**为能够改变方法之外的某处状态或执行 I/O 活动的方法。

比较容易出错的地方是如果去掉方法体前面的等号 那么方法的结果类型就必定是 Unit。这种说法不论方法体里包含什么都成立，因为 **Scala 编译器可以把任何类型转换为 Unit**。

**个人**：不是因为 Unit 的类继承层次，而是因为 Unit 对象的隐式转换方法 unbox?

就是说，带有花括号但没有等号的，本质上当做 Unit 结果类型的方法。



4. 如果去掉结果类型，但有等号的话，结果类型可以自动推导。

---

## 11.6 OBJECT

---

1. Scala 比 Java 更为面向对象的特点之一是 Scala 不能定义静态成员，而是待之以定义单例对象（singleton object）。

除了用 object 关键字替换了 class 关键字之外，单例对象的定义看上去与类定义一致。

2. 不与伴生类共享名称的单例对象被称为**独立对象**（standalone object）。它可以在很多地方，例如作为相关功能方法的工具类，或者定义 scala 应用的入口点。

➤ 任何拥有合适签名的 main 方法的单例对象都可以用来作为程序的入口点。

**个人**:问题,共享名称但又不在于同一个文件,即不是伴生对象的单例对象应该叫啥???

3. 伴生对象与伴生类

当单例对象与某个类共享同一个名称时，它就被称为是这个类的伴生对象（companion object）。

类和它的伴生对象必须定义在一个源文件里。

类被称为是这个单例对象的伴生类（companion class）。

类和它的伴生对象可以**互相访问其私有成员**。

4. 对于 Java 程序员来说，可以把单例对象当做是 Java 中可能会用到的静态方法 工具类。也可以用类似的语法做方法调用：单例对象名，点，方法名。

5. 然而单例对象不只是静态方法的工具类。它同样是头等的对象。因此单例对象的名字可以被看做是贴在对象上的“名签”。
  6. 定义单例对象并没有定义类型（在 Scala 的抽象层次上说）。
  7. 或者**可以认为**，单例对象的类型是由单例对象的伴生类定义的。然而，**单例对象扩展了父类并可以混入特质**。因此，可以使用类型调用单例对象的方法，或者用类型的实例变量指代单例对象，并把它传递给需要类型参数的方法。
  8. 类和单例对象间的**差别**是，单例对象不带参数，而类可以。因为单例对象不是用 new 关键字实例化的，所以没有机会传递给它实例化参数。
  9. 每个单例对象都被实现为**虚构类**（synthetic class — 类名为对象名上加\$）的实例，并指向静态的变量，因此它们与 Java 静态类有这相同的**初始化语义**。
- 个人** — 扩展：静态类是指在一个类的内部，又定义了一个用 static 修饰的类。那静态类的功能又体现在哪里呢？可以用 C 中的结构体内嵌结构体来理解，其次需要了解 2 个概念：内部类和静态修饰符 static。
- A，首先，用内部类是因为内部类与所在外部类有一定的关系，往往只有该外部类调用此内部类。所以没有必要专门用一个 Java 文件存放这个类。
- B，静态都是用来修饰类的内部成员的。比如静态方法、静态成员变量。它唯一的作用就是随着类的加载（而不是随着对象的产生）而产生，以致可以用类名+静态成员名直接获得。这样静态内部类就可以理解了，它可以直接被用 外部类名+内部类名 获得。
10. 特别要指出的是，单例对象在第一次被访问的时候才会被初始化。

---

## 11.7 APPLICATION 特质

---

1. Scala 提供了特质 `scala.Application` , 可以减少一些输入工作。可以把想要执行的代码直接放在单例对象的花括号之间。
2. 特质 `Application` 声明了带有合适签名的 `main` 方法 , 并被你写的单例对象继承 , 使它可以像 Scala 程序那样。花括号之间的代码被收集进了单例对象的主构造器 ( `primary constructor` ) , 并在类被初始化时执行。
3. 缺点 :
  - 如果想访问命令行参数的话即不能用它 , 因为 `args` 数组不可访问。
  - 因为某些 JVM 线程模型里的局限 , 如对于多线程的程序需要自行编写 `main` 方法 ;
  - 某些 JVM 的实现没有优化被 `Application` 特质执行的对象的初始化代码。

因此 , 只有当程序相对简单并且是单线程的情况下才可以继承 `Application` 特质。

**个人扩展** : JVM 线程模型、JVM 的优化。

4.

---

## 11.8 访问修饰符

---

1. Scala 的默认访问级别 : `public`。
2. Scala 大体上遵守 Java 对访问修饰符的对待方式 , 但也有一些重要的差异。
3. Private 修饰符 :
  - a) 私有成员的处理与 Java 的相同。标记为 `private` 的成员仅在包含了成员定义类或对象的内部可见。
  - b) Scala 里 , 同样的规则还应用到了内部类上 —— 即保持一致性。

即 Scala 的内部类可以访问外部类的私有成员（内部类在外部类的定义之内），但外部类不能访问内部类的私有成员（因为外部类调用点在内部类定义之外）。

Java：两种访问都支持，因为 java 允许外部类访问其内部类的私有成员。

—— 嵌套作用域是否在当前作用域可见的差异。

#### 4. Protected 修饰符：

Scala 里，对包含成员的访问也同样比 Java 严格一些。

Scala：保护成员只在定义了成员的类的子类中可以被访问；

Java：还允许在同一个包的其他类中进行这种访问。

#### 5. Private[x]或 Protected[x]修饰符 —— 保护的作用域：

表示直到 x 的私有或保护，这里 x 指代某个所属的包、类或单例对象。

带限定的访问修饰符提供给你非常细粒度的可见度控制。

#### 6. Private[this]：对象私有（Object-private）

7. 对于私有或保护访问来说，Scala 的访问规则给予了伴生对象和类一些特权。

## 12 组合与继承

---

1. 如果你省略 extends 子句, Scala 编译器将隐式地假设你的类扩展自 scala.AnyRef , 这与 Java 平台上的 java.lang.Object 相同。
2. 继承 ( inheritance ) 表示超类的所有成员也是子类的成员, 但以下两种情况例外:
  - a) 第一, 超类的私有成员不会被子类继承;
  - b) 第二, 超类中的成员若与子类中实现的成员具有相同名称和参数则不会被子类继承。 —— 这种情况称为子类的成员重新 ( override ) 了超类的成员。如果子类中的成员是具体的而超类中的是抽象的, 也可以说具体的成员实现 ( implement ) 了抽象的成员。
3. 子类型化 ( subtyping ) 是指子类的值可以在任何需要其超类的值的地方使用。  
**个人**-扩展: 设计模式---K...替换模式
4. 这种组合操作符也经常被称为连结符 ( combinator ), 因为它们能把某些区域的元素组合成新的元素。
5. 以连结符的方式思考问题通常是实现库的设计的好方法: 可以让你只要考虑以**基本的方式**在应用领域中构建对象即可。
6. 抽象方法: 被声明为没有实现的方法。不像 Java , 方法的声明中不需要 ( 也不允许 ) 有抽象修饰符。
7. 具体的 ( concrete ) 方法: 拥有实现的方法。
8. 具有抽象成员类本身必须被声明为抽象的, 只要在 class 关键字之前加上 abstract 修饰符即可。
9. 术语声明 ( declaration ) 和定义 ( definition ) 及其之间的区别。

10. 无参数方法

11. 带有空括号的方法定义，被称为空括号方法（empty-paren method）。

12. **推荐的惯例**是无论何时，只要方法中没有参数并且方法**仅**能通过读取所包含对象的属性去访问可变状态（特指方法不能改变可变状态），就使用无参数方法。

这个惯例支持**统一访问原则**（uniform access principle），就是说客户代码不应由属性是通过字段实现还是方法实现而受到影响。

13. **统一访问原则**（uniform access principle）只是 Scala 在对待字段和方法上比 Java 更统一的一个方面。另一个差异是 Scala 里的字段和方法属于**相同的命名空间**。这让字段可以重写无参数的方法。另一方面，Scala 里禁止在同一个类里使用同样的名称定义字段和方法，尽管 Java 允许这样做。

14. Scala 在遇到混合了无参数和空括号方法的情况时很自由。特别是，你可以用空括号方法**重写**无参数方法，并且反之亦可。

15. 原则上，Scala 的函数调用中可以省略所有的空括号。但无论是直接的还是非直接的使用可变对象时，都应该添加空括号。

16.

17.

## 12.1 命名空间

---

1. Java 为定义准备了四个命名空间（分别是字段，方法，类型和包）；

2. 一般来说，Scala 仅有两个命名空间：

a) 值（字段，方法，**包**还有单例对象）。

- b) 类型（类和特质名）。
- 3. Scala 里包与字段和方法共享相同命名空间是为了让你能够引入包，而不仅仅是引入类型名以及单例对象的字段和方法。
- 4. 可以通过在参数化字段（parametric field）定义中组合参数和字段 —— 同时定义同名的参数和字段。
- 5. 参数化字段可以添加诸如 private、protected，或 override 这类的修饰符。
- 6. 要调用超类构造器，只要简单地把要传递的参数或参数列表放在超类名之后的括号里即可。
- 7. 修饰符 Override：
  - a) Scala 要求，若子类成员所有重写了父类的具体成员则必须带有这个修饰符；
  - b) 若成员实现的是同名的抽象成员时，则这个修饰符是可选的；
  - c) 若成员并未重写或实现什么其他基类（我们通常称为超类）里的成员则禁用这个修饰符。

这条规则给编译器提供了有用的信息来帮助避免某些难以捕捉的错误并使得对系统的改进更加安全。

- 8. “脆弱基类”问题：这些“意外的重写”是这类问题的最通常的表现：《Scala 编程》P152。即如果你在类的层次结构中，为基类添加了新的成员，你会有破坏客户代码的风险。

只有继承才受累于脆基类问题。

- 9. Final 修饰符：
  - a) 修饰成员：确保一个成员不被子类重写 —— 和 Java 一样。

b) 修饰类：确保整个类不会有子类。

10. 在 Scala 中，可以在类和单例对象的内部定义其他的类和单例对象。

11. 定义工厂对象：《Scala 编程》P158

12.

---

## 13 SCALA 类层次

---

1. Scala 里，每个类都继承自通用的名为 Any 的超类。

2. Scala 还在层级的底端定义了一些有趣的类，如 Null 和 Nothing，扮演通用的子类。

如同 Any 是所有其他类的超类，Nothing 是所有其他类的子类。

scala.Null 和 scala.Nothing：它们是用统一的方式处理 Scala 面向对象类型系统的某些“边界情况”的特殊类型。

Null 类是 null 引用对象的类型，他是每个引用类（继承自 AnyRef 的类）的子类。Null 不兼容值类型。

Nothing 类型在 Scala 的类层次的最低端；它是任何其他类型的子类型。然而，根本没有这个类型的任何值。Nothing 的一个用处是它标明了不正常的终止。

3. 顶层的 Any 类，定义了下列的方法：

```
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean
def equals(that: Any): Boolean
def hashCode: Int
def toString: String
```

4. 实际上，==总是与 equals 相同，!=总是与 equals 相反。因此独立的类可以通过重写 equals 方法改变==或!=的意义。



5. 根类 Any 有两个子类：

- a) AnyVal：是 Scala 里每个内建值类的父类；对应 Java 的 8 个基本类型，加上 Unit 类。

对应的 8 个基本类型，它们的值在运行时表示成 Java 的基本类型的值。

值类都被定义为即是抽象的又是 final 的。

另一个值类，Unit，大约对应于 Java 的 void 类型；被用作不返回任何有趣结果的方法的结果类型。Unit 只有一个实例值，写成()。

- b) AnyRef：Scala 里所有引用类( reference class )的基类。在 Java 平台上 AnyRef 实际就是类 java.lang.Object 的别名。

尽管在 Java 平台上的 Scala 程序里 Object 和 AnyRef 的使用是可交换的，推荐的风格是在任何地方都只使用 AnyRef。

6. Scala 类与 Java 类的不同在于它们还继承自一个名为 ScalaObject 的特别的记号特质。是想要通过 ScalaObject 包含的 Scala 编译器定义和实现的方法让 Scala 程序的执行更高效。到现在为止，ScalaObject 只包含了一个方法，名为 \$tag，在内部使用以加速模式匹配。

7. Scala 里的相等操作 == 被设计为对类型表达透明。对值类型来说，就是自然的（数学或布尔）相等；对于引用类型，== 被视为继承自 Object 的 equals 方法的别名。这个方法被初始化地定义为引用相等，但被许多子类重写以实现它们自然理念上的相等性。

8. 需要使用引用相等代替用户定义的相等时：AnyRef 类定义了附加的 eq 方法。对应的反义词被称为 ne。

9.

## 14 特质

---

1. 特质 ( trait ) 是 Scala 代码里复用的基础单元。
2. 特质封装了方法和字段的定义，并可以通过混入到类中重用它们。
3. 与类的继承时每个类都只能继承唯一的超类不同，类可以混入任意多个特质。
4. 最常用到的两种方式：
  - a) 拓宽瘦接口为胖接口：可以根据类已有的方法自动为类添加方法。
  - b) 定义可堆叠的改变：特质让你改变类的方法，它们能够通过堆叠这些改动的方式做到这点。

瘦接口与胖接口的对阵体现了面向对象设计中常会面临的在实现者与接口用户之间的权衡。

5. Java 的接口常常是过瘦而非过胖。
6. 在特质中添加具体方法使得胖瘦对阵的权衡大大倾向于胖接口。
7. 特质的定义除了使用关键字 trait 之外，与类定义无异。
8. 一旦特质被定义了，就可以使用 extends 或 with 关键字，把它混入类中。
9. 你可以使用 extends 关键字混入特质；这种情况下隐式地继承了特质的超类。
10. 特质同样也是类型。
11. 如果想把特质混入显式扩展超类的类里，可以用 extends 指明待扩展的超类，用 with 混入特质。
12. 特质就像是带有具体方法的 Java 接口。不过它其实能做更多的事情。实际上，你可以用特质的定义做任何用类定义能做的事，并且，除了以下两点之外连语法都是一样的。

- a) 第一点：特质不能有任何“类”参数，即传递给类的主构造器的参数。
- b) 类和特质的另一个差别在于不论在类的哪个角落，**super** 调用都是静态绑定的，而在特质中，它们是**动态绑定**的。

在你定义特质的时候，**super** 调用的方法实现尚未被定义。调用的实现将在**每一次**特质被混入到**具体类**的时候才被决定。

这种处理 **super** 的有趣的行为是使得特质能以**可堆叠的改变** ( stackable modifications ) 方式工作的**关键**。

13. Ordered 特质并没有为你定义 equals 方法，因为它无法做到。问题在于要通过使用 compare 实现 equals 需要**检查**传入对象的类型，但是因为**类型擦除**，Ordered 本身无法做这种测试，因此，即使你继承了 Ordered，也还是需要自己定义 equals。

14. Super 的解释：

- a) 对于多重继承来说，**super** 调用导致的方法调用可以在调用发生的地方明确决定；
- b) 而对于特质来说，方法调用是由类和被混入到类的特质的线性化( linearization ) 所决定的。

15. 线性化的精确次序由语言的规格说明书描述。

16. 主旨是，在任何的线性化中，某个类总是被线性化在其所有超类和混入特质之前。

17. 线性化过程中，每个类仅出现一次。

18. 特质、抽象类选择规则：

- a) 如果行为不会被重用，那么就把它做成具体类。具体类没有可重用的行为。
- b) 如果要在多个不相关的类中重用，就做成特质。只有特质可以混入到不同的类层次中。

- c) 如果你希望从 Java 代码中继承它，就使用抽象类。
- 19. 因为特质和它的代码没有近似的 Java 模拟，在 Java 类里继承特质是很笨拙的。
- 20. 而继承 Scala 的类和继承 Java 的类完全一样。
- 21. 除了一个例外，只含有抽象成员的 Scala 特质将直接翻译成 Java 接口，因此即使你想用 Java 代码继承，也可以随心地定义这样的特质。
- 22. 当特质获得或失去成员，所有继承自它的类就算没有改变也都要被重新编译。
- 23. 如果效率非常重要，则应该倾向于使用类。
- 24. 大多数 Java 运行时都能让类成员的虚方法调用快于接口方法调用。特质被编译成接口，因此会付出微小的性能代价。
- 25.

## 15 包和引用

---

### 15.1 包

---

- 1. Scala 的代码采用了 Java 平台完整的包机制。
- 2. 可以用两种方式把代码放在命名包中：
  - a) 一种是通过 package 子句放在文件顶端的方式把整个文件内容放进包里 —— 更一般性的嵌套语法的语法糖。
  - b) 另一种把代码放进包里的方式更像 C# 的命名空间。可以在 package 子句 之后把要放到包里的定义用花括号括起来。除此之外，这种语法还能让你把文件的不同部分放在不同的包里。
- 3. Scala 的包是嵌套的。

4. Java 包，尽管是分级的，确不是嵌套的。Java 里命名一个包的时候，必须从包层级的根开始。
5. Scala 的**作用域划分规则**产生的另一个结果就是内部作用域的包可以隐匿被定义在外部作用域的同名包。

问题及解决：在内嵌的同名包遮盖了**顶层包**的情况下，如何访问？

Scala 在所有用户可创建的包之外提供了**名为\_root\_的包**。换句话说就是，任何你能写出来的顶层包都被当做是\_root\_包的成员。

---

## 15.2 引用

---

6. 在 Scala 里，包和其成员可以用 import 子句来引用。
7. Import 的三种方式举例：《Scala 编程》：P186。
  - a) Import 具体类：与 Java 的单类型引用一致；
  - b) Import 包.\_：对应于 Java 的按需( on-demand )引用，唯一的差别是 Scala 的按需引用写成以下划线 ( \_ ) 而不是星号 ( \* ) 结尾 ( 毕竟\*是合法的 Scala 标识符！ )。
  - c) Import 具体类.\_：与 Java 的静态字段引用一致。

Scala 引用实际可以更为普遍。

Scala 引用可以出现在任何地方，而不是仅仅在编译单一的开始处。

同样，它们可以指向**任意值**。

另一个体现了 Scala 引用灵活性的方面是它们可以引用包自身，而不只是除了包之外的其他成员。

**个人**：即，可以引用**对象**，也可以引用一个**包**（不用指定引用包中的具体类等）

## 8. Scala 的灵活的引用：

Scala 的 import 子句比 Java 的更为灵活。在它们之间存在三点主要差异。在 Scala 中，引用：

- 1) 可以出现在任何地方。
- 2) 可以指的是（单例或正统的）对象及包。
- 3) 可以重命名或隐藏一些被引用的成员：使用**引号选择器子句**（import selector clause）。

重命名子句的格式：<原始名> => <新名>

<原始名> => \_：格式的子句会从被引用的名字中排除<原始名>。即重命名为\_表示隐藏。

## 9. 引用选择器可以包括下列模式：

- a) 简单名 x：把 x 包含进引用名集；
- b) 重命名子句 x=>y：让名为 x 的成员以名称 y 出现；
- c) 隐藏子句 x=>\_：把 x 排除在引用名集之外；
- d) 全包括\_：引用除了前面子句提到的之外的全体成员。如果存在**全包括**，那么**必须是引用选择的最后一个**。

## 10. 隐式引用：

- a) `Import java.lang._` // java.lang 包的所有东西
- b) `Import scala._` // scala 包的所有东西
- c) `Import Predef._` // Predef 对象的所有东西

这三个引用子句与其他的稍有不同，出现在靠后位置的引用将覆盖靠前的引用。如：

Scala 包的东西覆盖 java.lang 中的东西。

11. Predef 对象包含了许多 Scala 程序中常用到的类型、方法和隐式转换的定义。

12.

## 16 断言和单元测试

---

个人:断言学习官网：[http://www.scalatest.org/user\\_guide/using\\_assertions](http://www.scalatest.org/user_guide/using_assertions)

- 1. Assert 方法
- 2. Ensuring 方法：带一个函数做参数，该函数是接受一个结果类型对象并返回 Boolean 类型的论断函数（predicate function）。

由于存在隐式转换，ensuring 方法可以用在任何结果类型上。

- 3. AssertionError
- 4. 断言（以及 ensuring 检查）可以使用 JVM 的 -ea 和 -da 命令行标识开放和禁止。
- 5. Scala 的单元测试：
  - a) Java 实现的工具：JUnit 和 TestNG
  - b) Scala 的新工具：ScalaTest、specs 和 ScalaCheck

## 16.1 SCALATEST

---

1. ScalaTest 提供了若干编写测试的方法 ,最简单的就是创建扩展 `org.scalatest.Suite` 的类并在这些类中定义测试方法。
2. Suite 代表一个测试集。测试方法名以 “test” 开头。
3. 尽管 ScalaTest 包含了 Runner 应用 ,你还可以直接在 Scala 解释器中通过调用 `execute` 方法运行 Suite。
4. 特质 Suite 的 `execute` 方法使用反射发现测试方法并调用它们。

`(new ElementSuite).execute()`

5. 由于 `Execute` 方法可以在子类型中重载 ,因此 ScalaTest 为不同风格的测试提供了便利。
6. ScalaTest 提供了名为 `FunSuite` 的特质 ,重载了 `execute` ,从而可以让你以函数值的方式而不是方法定义测试。
7. “test” 是定义在 `FunSuite` 中的方法 ,它将被 `ElementSuite` 的主构造器调用。圆括号里的字符串指定了测试的名称 ,花括号之间的是测试代码。
8. 测试代码是被作为传名参数传递给 `test` 的函数 ,并由 `test` 函数注册之后运行。
9. 三等号操作符
10. Expect 方法
11. Intercept 方法 : 检查方法是否抛出期待的异常



## 17 样本类和模式匹配

---

1. 这两种在编写规范的、无封装数据结构时会用到的构件。它们对于树型递归数据尤其有用。
2. Scala 可以去掉围绕空类结构体的花括号，因此 `class C` 与 `class C {}` 相同。
3. Case 修饰符：带有这种修饰符的类被称为样本类 ( case class )。
4. 这种修饰符可以让 Scala 编译器自动为你的类添加一些句法上的便捷设定。
  - a) 首选，它会添加与类名一致的工厂方法。
  - b) 第二个句法便捷设定是样本类参数列表中的所以参数隐式获得了 `val` 前缀，因此它被当做字段维护。
  - c) 第三个，是编译器为你的类添加了方法 `toString`、`hashCode` 和 `equals` 的“自然”实现。
5. 样本类最大的好处还在于它们能够支持模式匹配。
6. Scala 与 Java :
  - a) 选择器 `match {备选项}`
  - b) `Switch(选择器) {备选项}`
7. 模式匹配
  - a) 一个模式匹配包含了一系列备选项 ( alternative )，每个都开始于关键字 `case`。
  - b) 每个备选项都包含了一个模式 ( pattern ) 及一到多个表达式，它们将在模式匹配过程中被计算。
  - c) 箭头符号 `=>` 隔开了模式和表达式。
8. 模式

- a) 常量模式 ( constant pattern ) : 匹配的值等于用 == 判断相等的常量。任何字面量都可以用作常量, 任何的 val 或单例对象也可以被用作常量。常量模式可以有符号名, 如 Nil 作为模式时。
- b) 变量模式 ( variable pattern ) : 匹配所有值。之后在 case 子句的右侧, 这个变量指代了被匹配的值, 即 Scala 把变量绑定在匹配的对象上。
- c) 通配模式 ( wildcard pattern ) : ( \_ ), 同样也匹配所有值, 不过没有引入指向那个值的变量名。除了用作默认的“全匹配 ( catch-all )”的备选项, 通配模式还可以用来忽略对象中你不关心的部分。
- d) 构造器模式 ( constructor pattern ) : 这种模式匹配所有类型指定的类, 并且其参数也进行模式匹配, 即传递给构造器的参数本身也是模式。

#### 9. Java 的 switch 与 Scala 的 match 三点不同 :

- a) 首先, match 是 Scala 的表达式, 即, 它始终以值作为结果;
- b) 第二, Scala 的备选项表达式永远不会“掉到”下一个 case ;
- c) 第三, 如果没有模式匹配, MatchError 异常会被抛出。

#### 10. Scala 模式匹配的文字规则 :

- a) 用小写字母开始的简单名被当作是模式变量;
- b) 所以其他的引用被认为是常量。

#### 11. 给模式常量使用小写字母名的两种手段 :

- a) 首先, 如果常量是某个对象的字段, 可以用限定符前缀。
- b) 或使用反引号包住变量名, 如 `pi`, 会解释成常量, 而不是变量。

#### 12. Scala 里标识符的反引号语法有两处不同目的的用法以帮助你在非常规的情况下解决编码问题。

- a) 用来处理小写字母标识符当做模式匹配常量的问题；
- b) 用来处理关键字被当做普通的标识符的问题，如 `Thread.`yield`()`。

13. 构造器模式——Scala 模式支持深度匹配 ( deep match )：这种模式不只检查顶层对象是否一致，还会检查对象的内容是否匹配内层的模式。由于额外的模式自身可以形成构造器模式，因此可以使用它们检查到对象内部的任意深度。

14. 序列模式：匹配如 List 或 Array 这样的序列类型。可以支持指定模式内任意数量的元素。

15. 如果想匹配一个不指定长度的序列，可以指定 `*_` 作为模式的最后元素。这种古怪的模式能匹配序列中零到任意数量的元素。

16. 元组模式：匹配元组

17. 类型模式 ( typed pattern )：可以当做类型测试和类型转换的简易替代。

Case s:String => ...

**个人**：相当于构造器模式+变量模式，可以调用匹配类型的成员，如 `s.length`

18. 能够得到与类型模式匹配相同效果但更为曲折的方式需要使用类型测试及类型转换。

Scala 使用了与 Java 不同的语法：如，

测试类型：`expr.isInstanceOf[String]`

转换类型：`expr.asInstanceOf[String]`

19. 操作符 `isInstanceOf` 和 `asInstanceOf` 被当做了带了类型参数( 方括号内 )的 Any 类预定义方法。

20. 类型擦除：特定元素类型的映射是否能匹配？

21. Scala 使用了泛型的擦除 ( erasure ) 模式 , 如 Java 那样。也就是说类型参数信息没有保留到运行期。
22. 擦粗规则的唯一例外就是数组 , 因为在 Scala 和 Java 里 , 它们都被特殊处理了。数组的元素类型与数组值保存在一起 , 因此它可以用模式匹配。
23. 命令行选项 : -unchecked 开始解释器 , 解释器会发出更多 “未检查警告” 的细节说明。
24. 变量绑定模式 : 除了独立的变量模式之外 , 还可以对任何其他模式添加变量 : 变量名、一个@符号 , 以及这个模式。

这种模式的意义在于它能像通常的那样做模式匹配 , 并且如果匹配成功 , 则把变量设置成匹配的对象。

25. Scala 要求模式是线性的 : 模式变量仅允许在模式中出现一次。可以使用模式守卫 ( pattern guard ) 重新制定匹配规则。

26. 模式守卫 : 模式守卫接在模式之后 , 开始于 if。

守卫可以是任意的引用模式中变量的布尔表达式。

如果存在模式守卫 , 那么只有在守卫返回 true 的时候匹配才成功。

---

## 17.1 封闭类

---

1. 让 Scala 编译器帮助你检测 match 表达式中遗漏的模式组合。

可选方案是让样本类的超类被封闭 ( sealed ) , 只要把关键字 sealed 放在最顶层类的前边即可。

2. **封闭类**除了类定义所在的文件之外不能再添加任何新的子类。
3. 如果你使用继承自封闭类的样本类做匹配，编译器将通过警告信息标识出缺失的模式组合。
4. 关闭检查：unchecked 注解

通常你可以用类似于添加类型的方式给表达式添加注解：表达式后跟一个冒号以及注解的名称（前缀@符号）。

```
(e:@unchecked) match {}
```

@unchecked 注解对于模式匹配来说有特定的意思。如果 match 的选择器表达式带有这个注解，那么对于随后的模式的穷举行检查将被抑制掉。

## 17.2 OPTION 类型

---

个人：总结在云笔记上：

<http://note.youdao.com/yws/public/redirect/share?id=b108cfb3a56c4388c3fa40689a441cb8&type=false>

1. Scala 为可选值定义了一个名为 Option 的标准类型。
2. Option 这种值有两种形式：
  - a) Some(x)形式，其中 x 为实际值；

b) None 对象，代表缺失的值。

```
/**
 * 打印map的value的长度
 */
def printValueLen(a:Option[String]) = {
  for(i<- a){
    println(i+" length--->" +i.length)
  }
}
/**
 * 模式匹配map的value
 */
def matchMap(a:Option[String])= a match{
  case Some(a) => a
  case None =>0
}
```

### 17.3 模式的使用

1. 模式在变量定义中：定义 val 或 var 的任何时候，都可以使用模式替代简单的标识符。
2. 用作偏函数的样本序列：花括号内的样本序列（备选项）可以用在能够出现函数数字面量的任何地方。

实际上，样本序列就是函数数字面量，而且只有更普遍。

函数数字面量只有一个入口点和参数列表，但样本序列可以有多个入口点，每个都有自己的参数列表。

每个样本都是函数的一个入口点，参数也被模式所特化。

每个入口点的函数体都在样本的右侧。

```
/**
 * 模式匹配
 */
def matchTest(a:Any):Any = a match{
  case 1 => "test1"
  case 2 => "case"
  case 3 => "6"
  case 4 => "play"
  case "c" => 6
}
```

### 3. 使用偏函数类型：PartialFunction[T,T]

方法 isDefinedAt：可以用来测试是否函数对某个特定值有定义。

使用偏函数的时候，编译器没办法帮你避免运行期故障的出现。

### 4. Scala 编译器对模式匹配函数字面量的转译：《Scala 编程》：P218

这种翻译只有在函数字面量的声明类型为 PartialFunction 的时候才起效。否则函数字面量就会代而转译为完整的函数。

### 5. For 表达式里的模式

产生出来的不能匹配于模式的值将被丢弃。

---

## 18 类型参数化

---

1. 信息隐藏可以用来获得更为通用的类型参数化的变化型注解。
2. 类型参数化让你能够编写泛型类和特质。
3. 变化型定义了参数化类型的继承关系。

4. 理想情况下，函数式（不可变）队列不应该具有比指令式（可变）队列更高的基本开销。
5. 可以通过把 `private` 修饰符添加在类参数列表的前边把主构造器隐藏起来。
6. 夹在类名与其参数之间的 `private` 修饰符表明类的构造器是私有的：它只能被类本身及伴生对象访问。类名仍然是公开的，可以继续使用这个类，但不能调用它的构造器。
7. 另一种更为彻底的方式是直接把类本身隐藏掉，仅提供能够暴露类公共接口的特质。
8. 类型构造器：通过指定参数类型构造新的类型。
9. 类型构造器“产生”了一个类型的家族。
10. 术语“泛型”指的是通过一个能够广泛适用的类或特质定义了许多特定的类型。

## 18.1 参数的变化型

---

11. 无论类型参数是协变的，逆变的，还是非协变的，都被称为参数的变化型。
12. 可以放在类型参数前的+号和-号被称为变化型注解。
13. 在正常的类型参数前加上+号表明这个参数的子类型化是协变（弹性）的。
14. 前缀加上-号，表明是需要逆变的（contravariant）子类型化。
15. 在 Scala 中，泛型类型缺省的是非协变的（nonvariant）（或称为“严谨的”）子类型化。
16. 纯函数式中，许多类型都是自然协变（弹性）的。然而，一旦引入了可变数据，情况就改变了。
17. 只要泛型的参数类型被当做方法参数的类型，那么包含它的类或特质就可能不能与这个类型参数一起协变。



18. 规则：不允许使用+号注解的类型参数用作方法参数类型。

## 18.2 SCALA 编译器检查变化型注解的机制

---

1. 为了核实变化型注解的正确性，Scala 编译器会把类或特质结构体的所有位置分类为正，负，或中立。
2. 所谓的“位置”是指类或特质的结构体内可能会用到类型参数的地方。
3. 编译器检查类的类型参数的每一个用法。
  - a) 注解了+号的类型参数只能被用在正的位置上；
  - b) 而注解了-号的类型参数只能用在负的位置上；
  - c) 没有变化型注解的类型参数可以用于任何位置，因此它是唯一能被用在类或特质结构体的中性位置上的类型参数。
4. 处于**声明类的最顶层**被划为正的位置。默认情况下，更深的内嵌层的位置的分类会与它的外层一致，不过仍有屈指可数的几种例外会改变具体的分类。
5. **方法值参数**位置是方法外部的位置的翻转类别，这里正的位置翻转为负的，负的位置翻转为正的，而中性位置仍然保持中性。
- 6.

## 19 函数组合

---

Effective Scala

### 19.1 函数组合

---

```
def f(s: String) = "f"
def g(s: String) = "g"

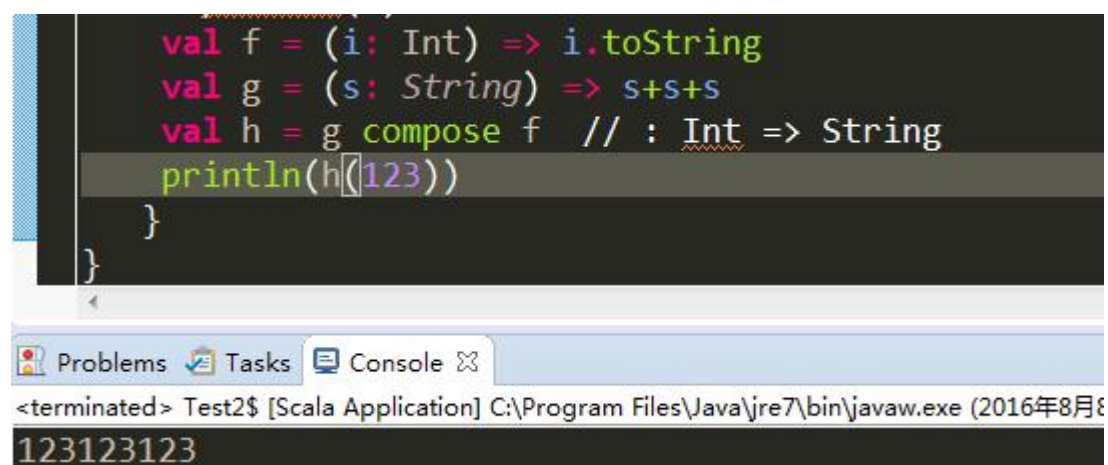
def f(i: Int)(s: String) = "f"+i+s
val fComposeG = f(1) _ compose g _
// print(fComposeG("1")) //-->f1g
val fAndThenG = f(1) _ andThen g _
// print(fAndThenG("1")) //-->g
```

compose 组合其他函数形成一个新的函数  $f(g(x))$

andThen 和 compose 很像，但是调用顺序是先调用第一个函数，然后调用第二个，即  $g(f(x))$

## 19.2 组合(composition)

让我们重新审视我们所说的组合：将简单的组件合成一个更复杂的。函数组合的一个权威的例子 给定函数  $f$  和  $g$  ,组合函数  $(g \circ f)(x) = g(f(x))$  ——结果先对  $x$  使用  $f$  函数，然后在使用  $g$  函数——用 Scala 来写：



```
val f = (i: Int) => i.toString
val g = (s: String) => s+s+s
val h = g compose f // : Int => String
println(h(123))
}
```

Problems Tasks Console

<terminated> Test2\$ [Scala Application] C:\Program Files\Java\jre7\bin\javaw.exe (2016年8月8)

123123123

复合函数  $h$ ，是个新的函数，由之前定义的  $f$  和  $g$  函数合成。

## 20 语法

---

1. 占位符语法：下划线\_，这是通常在 Scala 里作为占位符，来表示未知值的通配符。

个人:关于下划线请查看

<http://note.youdao.com/yws/public/redirect/share?id=11ddee548c9c8a12c3993bf2f83e29d5&type=false>

2. 不再使用 break 和 continue。
3. 在 Scala 里有时候可以用花括号代替小括号。For 表达式的可选语法就是这种用法的一个例子。

```
/**
 * 动态调用参数
 */
def dynamicparams(strs: String*)={
  var i = 0
  for(str <- strs) {
    println("the ["+i+" ]params--->"+str)
    i=i+1
  }
}
```

### 20.1 常用语法

### 20.2 递归调用

---

1. 尾调用优化——尾递归：递归调用在尾调用（tail-call）位置，编译器会产生出于while 循环类似的代码。每个递归调用将被实现为回到函数开始位置的跳转。

```
/**
 * 递归
 */
def recursive1(a: BigInt) : BigInt = {
  println("params ----->" + a)
  if(a <= 1) {
    1
  } else {
    a * recursive1(a - 1)
  }
}
```

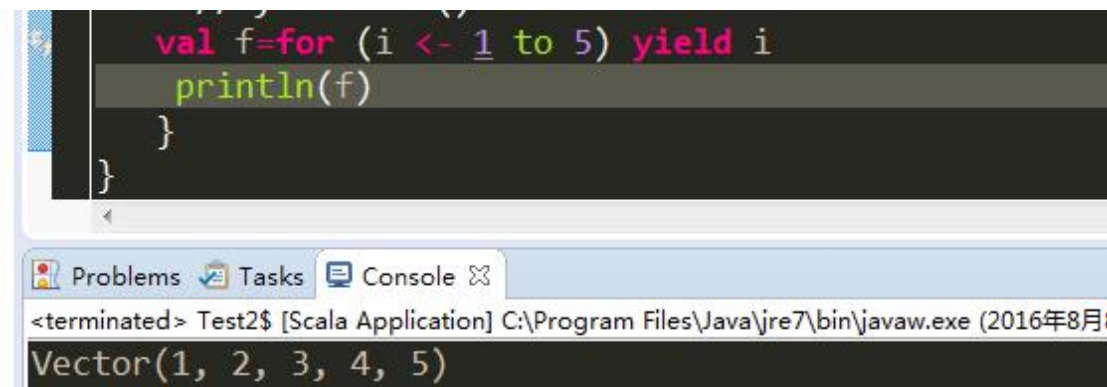
### 20.3 FOR 表达式

1. 发生器 ( generator ) 的语法 : "xxx <- xxxSeq" —— 参见 for 表达式。  
如果加入多个 <- 子句 , 就得到了嵌套的 "循环" 。
2. 过滤器 ( filter ) : for 表达式中的 if 字句。
3. 注意 : 如果在发生器中加入超过一个过滤器 , if 字句必须用分号分隔。
4. 可以使用花括号代替小括号包裹发生器和过滤器。使用花括号的好处是可以省略使用小括号时必须加的分号。
5. 流间 ( mid-stream ) 变量绑定 : 如 : for 表达式里的流间赋值。变量从半路引入 for 表达式。

《Scala 编程》P111

for-yield 表达式的语法 : **for** {子句} **yield** {变量或表达式}

制造新集合：在 for 表达式之前加上关键字 **yield**。每次循环产生一个值。



The screenshot shows an IDE window with a Scala file. The code is as follows:

```
val f=for (i <- 1 to 5) yield i
println(f)
}
```

The IDE's console window at the bottom shows the output:

```
<terminated> Test2$ [Scala Application] C:\Program Files\Java\jre7\bin\javaw.exe (2016年8月)
Vector(1, 2, 3, 4, 5)
```

## 20.4 TRY 表达式与异常

1. 把抛出的异常当做任何类型的值都是安全的。任何使用经 throw 返回值的尝试都不会起作用，因此这样做不会有害处。
2. 从技术角度上，抛出异常的类型是 Nothing。
3. 尽管 throw 不实际产生任何值，但你还是可以把当表达式。
4. 选择 catch 子句这种语法的原因是为了与 Scala 很重要的部分模式匹配（pattern matching）保持一致。
5. Scala 与 Java 的差别在于不需要补货检查异常（checked exception），或把它们声明在 throws 子句中。可以用注解声明@throws 子句，但不是必须的。
6. Finally 子句 —— Scala 可以使用另一种被称为出借模式（loan pattern）的技巧更简洁地达到同样的目的。
7. Try-catch-finally 也产生值：
  - 没有异常抛出：对应 try 子句；

- 抛出异常被捕获：对应相应的 catch 子句；
- 抛出异常单没有被捕获：表达式没有值。

由 finally 子句计算得到的值，即使有也会被抛弃。—— 即 finally 不应该修改主函数体或 catch 子句中计算的值。

与 Java 对比：Scala 的行为差异仅在于 Java 的 try-finally 不产生值。

在 Java 中，如果 finally 子句也包含返回语句，或抛出异常，这个返回值或异常将“凌驾”于任何之前的 try 子句或 catch 子句里产生的值或异常之上。

8. 在 Scala 中，被 “{}” 包含的一系列 case 语句可以被看成是一个函数字面量，它可以被用在任何普通的函数字面量适用的地方，例如被当做参数传递。

```
/**
 * 测试异常处理
 */
def tryTest(){
  try {
    var f = new FileReader("1.txt")
  } catch {
    case ex :FileNotFoundException => println(ex) // TODO: handle error
    case ex: IOException => println(ex)
  } finally {
    println("-----一定会执行的代码")
  }
}
```