函数

1. 函数式编程风格的一个重要设计原则：程序应该被分解为许多小函数，每个函数只做明确定义的任务。每个函数通常都很小。

一、函数定义

1. 函数定义：以`def`开始，然后是函数名，然后是圆括号`()`和圆括号包围的、以逗号`,`分隔的参数列表，然后是冒号`:`，然后是返回类型，然后是等号`=`，最后是花括号`{}`包围的函数体。

```
def max(x:Int,y:Int): Int = {
    if (x>y) x
    else y
}
```

   - 参数列表中的每个参数都必须加上冒号`:`开始的类型标注，因为编译器无法推断函数参数的类型。

     如果函数不接收任何参数，则参数列表为空，此时使用空的圆括号`()`，表示不接收任何参数。

   - 圆括号包围的参数列表之后是返回结果的类型标注。

     如果函数没有返回结果，则结果类型标注为`:Unit`。`Unit`结果类型类似与`Java`的`void`类型，表示函数并不会返回任何有实际意义的结果。

   - 函数体之前的等号也有特别的含义，表示在函数式编程中，函数定义是一个表达式。

2. 一个返回结果类型为`Unit`的函数通常带有副作用（如：修改某个变量，或者输入/输出），这样的函数也被称作过程`procedure`。

   - 每个有用的`preocedure`都必须有某种形式的副作用，否则它对于外部世界没有任何价值。
   - 函数式编程倾向于使用没有副作用的函数，这鼓励你设计的代码副作用尽可能小。其好处是：更容易测试。

3. 一旦定义好函数，则可以通过按函数名来调用：`max(3,5)`。函数定义中的参数列表`x,y`称作形参，函数调用中的值`3,5`称作实参。

4. `Scala`函数的参数都是`val`而不是`var`，这意味着不能在函数体内部对入参重新赋值：

```
def add(b:Byte):Unit ={
    // b = 1; 错误：b 是一个val
    println(b)
}
```

5. 在没有任何显式的`return`语句时，`Scala`函数返回的是该函数计算出的最后一个表达式的值。

   `Scala`推荐的风格是：尽量避免使用任何显式的`return`语句，尤其是多个`return`语句。这样可以将每个函数视作一个最终提交某个值的表达式。这种哲学鼓励你编写短小的函数，将大的函数拆解成小的函数。

   - 当一个函数只会计算一个返回结果的表达式时，可以不写花括号。如果这个表达式很短，它甚至可以被放置在`def`的同一行。
   - 为了极致的精简，还可以省略掉结果类型，`Scala`会帮你推断出来。

但是为了代码的可读性，推荐对类中声明的公有方法（它是函数的一种）显式的给出结果类型。

1.1 定义缩略

1. 有时候没必要采用完整的函数定义，`Scala` 编译器可以帮助我们做推断。
2. 如果函数的返回结果的类型是已知的，则函数定义中可以省略掉结果类型部分。
   - 如果函数是递归的，则必须显式给出函数的结果类型。
   - 通常建议显式的给出函数的结果类型，虽然编译器不会强求，但是这会让代码更容易阅读。
3. 如果函数体只有一条语句，则可以省略掉花括号 `{}` 。

   如：

   ```
   def max(x:Int,y:Int) = if (x>y) x else y
   ```

4. 如果可以推断参数类型信息，则可以省略参数类型声明。

   ```
   val intList = List[Int]()
   intList.filter((x) => x > 0 )
   ```

   其中 `intList` 是一个整数列表。由于 `Scala` 编译器知道 `x` 必定是个整数，因此 `(x:Int)` 可以简写为 `(x)` 。

   这被称作目标类型 `target typing`，因为一个表达式的目标使用场景会影响该表达式的类型。该机制原理细节不必掌握，你可以不需指定任何参数类型，直接使用函数字面量，当编译器报错时再添加类型声明。

5. 如果省略了参数类型，则参数两侧的圆括号也可以省略。

   ```
   intList.filter(x => x > 0 )
   ```

6. 为了让函数字面量更精简，还可以使用下划线作为占位符，用于表示一个或者多个参数，只需要满足每个参数只在函数字面量中只出现一次即可。

   ```
   intList.filter(_ > 0 )
   ```

   可以将下划线当作是表达式中的需要被“填”的“空”。函数每次被调用时，该“空”会被入参填上。

   事实上，当 `_` 表示一个参数，且该参数恰好也是一个函数时，其语法会比较怪异：

   ```
   actionList.foreach(_ ())
   ```

   其中 `actionList` 是一个列表，列表中每个元素都是一个函数。这些函数的参数列表都为空。因此上式等于：

   ```
   actionList.foreach(f => f())
   ```

7. 有时候使用 `_` 作为参数占位符时，编译器可能没有足够多的信息来推断缺失的参数类型。此时可以通过冒号来显式给出参数类型。

   ```
   val f = _ + _ // 错误：无法知道参数类型
   val f = (_:Int) + (_:Int) // 正确，知道参数类型
   ```

`Scala` 编译器将 `_ + _` 展开为一个接收两个参数的函数字面量。

多个下划线意味着多个参数，而不是一个参数的多次使用：第一个下划线代表第一个参数，第二个下划线代表第二个参数，依次类推。

8. 下划线 `_` 不仅可以替换掉单个参数，它甚至可以替换掉整个参数列表。如：

```
intList.foreach(println _)
```

这等价于 `intList.foreach(x => println(x))` 。这里的下划线 `_` 是整个参数列表的占位符。注意：需要保留 `println` 和 `_` 之间的空格。

这种语法是部分应用函数 `partially applied function` 。在 `Scala` 中，当你调用某个函数，传入任何需要的参数时，你实际上是 `apply` 那个函数到这些参数上。

1.2 部分应用函数

1. 部分应用函数是一个表达式，在这个表达式中，并不需要函数给出所有的参数，而是部分给出甚至完全不给出。如：

```
def sum(a:Int,b:Int,c:Int) = a + b +c
val i = sum(1,2,3) // 函数调用
val a = sum _ // partially applied function
println(a(1,2,3)) // 输出：6
```

2. 部分应用函数将返回一个新的函数。背后的原理是：
   - `Scala` 编译器根据 `partially applied function` 创建一个类，并实例化一个值函数，并将这个新的值函数的引用赋值给变量 `a` 。

     该类有一个接收 `3` 个参数的 `apply` 方法，这是因为表达式 `sum _` 缺失的参数个数为 `3` 。
   - 然后编译器将表达式 `a(1,2,3)` 翻译成对值函数 `apply` 方法的调用：`a.apply(1,2,3)` 。

     这个 `apply` 方法只是简单的将三个缺失的参数转发给 `sum` ，然后返回结果。
3. 你也可以通过给出一些实参来表达一个部分应用的函数。如：

```
val a = sum(1,_:Int,3) // partially applied function
```

由于只有一个参数缺失，因此 `Scala` 编译器生成的新的函数类，这个函数类的 `apply` 方法接收一个参数。

4. 如果你的部分应用函数表达式并不给出任何参数，则可以简化成：`sum _` ，甚至连下划线也不用写。

   如果什么都不写，则要求在明确需要函数的地方，否则会引起编译错误：

```
intList.foreach(println) // foreach 的参数要求是个函数
```

1.3 可变长度参数列表

1. `Scala` 允许你标识出函数的最后一个参数可以被重复，这使得调用方可以传入一个可变长度的参数列表。

   可变长度参数列表的语法为：在参数的类型之后加上一个星号 `*` 。

```
def echo(args : String*) =  for (arg <- args) println(arg)
```

在函数内部，该重复参数的类型是一个所声明的参数类型的 `Array`。因此在 `echo` 内部，`args` 的类型其实是 `Array[String]`。

- 如果你有一个 `Array[String]` 变量，则它不能作为 `echo` 的实参。因为 `echo` 的参数必须都是 `String` 类型。
- 你可以为数组实参 `arr` 的后面加上一个冒号和一个 `_*` 符号，这种表示法告诉编译器：将数组实参 `arr` 的每个元素作为参数传给函数，而不是将数组实参 `arr` 作为单个参数传入。

```
def echo(args : String*) =  for (arg <- args) println(arg)
val arr = Array("Hello","World")
echo(arr)       // 编译失败，参数不匹配
echo(arr : _*)  // OK
```

1.4 带名字的参数

1. 在一个普通的函数调用中，实参是根据被调用的函数的参数定义，逐一匹配起来的。`Scala` 支持带名字的参数，使得调用方可以用不同的顺序将实参传给函数。其语法是：在每个实参之前加上参数名和等号。

```
def getArea(width:Float,height:Float) = width * height
println(getArea(3.0,4.0))              // width = 3.0, height = 4.0
println(getArea(height=4.0,width=3.0)) // width = 3.0, height = 4.0
```

2. 通过带名字的参数，实参可以在不改变含义的情况下交换位置。
3. 可以混合使用按位置的参数和带名字的参数，此时按位置的参数需要放在前面。
4. 带名字的参数最常用的场合是跟默认参数一起使用。

1.5 默认参数

1. `Scala` 允许你给函数参数指定默认值。这些带有默认值的参数可以不出现在函数调用中，此时这些参数被填充为默认值。其语法是：在函数定义时，形参类型之后使用 `= 默认值` 的格式。

```
def printTime(out: java.io.PrintStream = Console.out) = {
    out.println("time = " + System.currentTimeMillis())
}
```

1.6 高阶函数

1. 高阶函数 `higher-order function`：接收函数作为参数的函数。

```
def filesMatching(query : String,
                  matcher : (String,String) => Boolean) = {
    // 这里是函数定义
}
```

其中形参 `matcher` 是一个函数，因此类型声明中有个 `=>` 符号。这个函数接收两个字符串类型的参数并返回一个布尔值。

2. 高阶函数的优势：

   - 高阶函数用于创建减少重复代码的控制抽象。

     如：返回指定模式的文件名：

     ```
     def filesMatching(query : String,
                       matcher : (String,String) => Boolean) = {
         // 这里是函数定义
     }
     ```

     这种方式避免了重复定义一些函数：

     ```
     def files_begin(query : String)= {
         // 这里是函数定义
     }
     def files_end(query : String)= {
         // 这里是函数定义
     }
     ...
     ```

   - 在`API`中采用高阶函数，从而使得调用代码更精简。

     如：列表的`exists`方法就是高阶函数：

     ```
     val has_neg = intList.exists(_ < 0) // 列表是否包含负数
     ```

     由于`exists`是`Scala collection API`中的公共函数，因此它减少了`API`使用方的代码重复。大量的循环逻辑都被抽象到了`exists`方法里了。

1.7 **柯里化**curring

1. 常规的函数定义包含一个参数列表，而一个柯里化的函数定义包含多个参数列表。

   柯里化函数调用时，每个参数列表都需要提供对应的实参。

   ```
   def regular_f(x:Int,y:Int,z:Int) = x + y + z    // 常规函数定义
   def currying_f(x:Int)(y:Int)(z:Int) = x + y + z // 柯里化函数定义

   println(regular_f(1,2,3))    // 常规函数调用
   println(currying_f(1)(2)(3)) // 柯里化函数调用
   ```

   当调用`currying_f`时，实际上是连续进行了三次传统的函数调用：

   - 第一次调用接收了一个名为`x`的`Int`参数。
   - 第二次调用接收了一个名为`y`的`Int`参数。
   - 第三次调用接收了一个名为`z`的`Int`参数。

2. 可以通过占位符表示法来获取柯里化函数内部的“传统函数”的引用。

   ```
   def f(x:Int)(y:Int)(z:Int) = x + y + z // 柯里化函数定义
   ```

   - 无法通过下面的方式获取，会引发编译失败：

```
val f1 = f(1)      // 编译失败
val f2 = f(1)(2)   // 编译失败
```

- 可以通过括号外的`_`来获取。

```
val f1 = f(1)_       // 指向一个函数的引用，类型：Int => Int => Int
                     //                  （参数为 Int，返回值为一个函数）
val f2 = f(1)(2)_    // 指向一个函数的引用，类型：Int => Int
```

理论上`_`之前要放置空格，但是由于这里`_`之前是圆括号，因此不必放置空格。如`println _`就必须放置空格，因为`println_`是另一个合法的标识符。

- 也可以通过括号内的`_`来获取，但是此时需要给定参数类型。

```
val f1 = f(_:Int)(_:Int)(_:Int)  //  指向一个函数的引用，类型：(Int, Int, Int) => Int
val f2 = f(1)(_:Int)(_:Int)      //  指向一个函数的引用，类型：(Int, Int) => Int
val f3 = f(1)(2)(_)              //  指向一个函数的引用，类型：Int => Int
```

1.8 传名参数

1. 当函数的参数是另一个函数，且该参数函数的参数列表为空时，参数函数可以退化为传名参数`by-name parameter`：

```
def filter_line1(file:File,op:() => Boolean) = {
    // 函数体
}
def filter_line2(file:File,op: => Boolean) = { // 退化为传名参数
    // 函数体
}
```

传名参数是一个以`=>`开头的类型声明的参数，而不是`() =>`开头。

2. 传名参数比常规的参数函数的优点在于：调用时可以去掉空的参数列表。

```
filter_line1(new File("data.txt"),() => 5>3 )  // 常规调用，必须带上空的参数列表
filter_line2(new File("data.txt"), 5>3 )       // 传名参数调用，不用带上空的参数列表
```

3. 传名参数是相对于传值参数`by-value parameter`来说的。

```
def filter_line3(file:File,op: Boolean) = { // 传值参数
    // 函数体
}
filter_line3(new File("data.txt"), 5>3 )       // 传值参数调用
```

- 传名参数和传值参数在调用时完全相同；在定义时，传名参数的类型声明多了一个`=>`。
- 传名参数本质上是一个值函数对象，因此`5>3`转换成一个返回`Boolean`值的值函数。

  因此：

- 对于传值参数，表达式`5>3`首先被求值。
- 对于传名参数，表达式`5>3`并不会立即求值，而是在这个值函数对象`apply`方法被应用时才会求值。

  如果该值函数的`apply`方法从未被调用，则表达式`5>3`始终不会被求值。

4. 传名参数和传值参数的典型比较：

```
def assert_by_p_name(enable_assert:Boolean,predicate: => Boolean)={
    if (enable_assert && ! predicate())
      throw new AssertionError
}   // 传名参数

def assert_by_p_value(enable_assert:Boolean,predicate: Boolean)={
    if (enable_assert && ! predicate)
      throw new AssertionError
}   // 传值参数

assert_by_p_name(false,5/0 == 0 )       // 正常执行，因为 5/0 不会被求值
assert_by_p_value(false,5/0 == 0 )      // 抛出异常，因为 5/0 被求值了
```

二、函数分类

1. `Scala`中的函数有几种类型：对象的方法、局部函数、函数字面量、值函数（注：它不是函数的返回值）。

3.1 对象的方法

1. 定义函数最常用的方式是作为某个对象的成员，这样的函数称作方法`method`。

```
class C{
    val field1 = 0;          // 字段
    var field2 = "hello";    // 字段
    def method(){};          // 方法
}
```

2. 如果一个方法只接收一个参数，则在调用时可以不使用英文句点`.`或者圆括号`()`。如：

```
for (i <- 0 to 2)
  print(xxx)
```

其中`0 to 2`等价于`0.to(2)`，它调用的是`Int`实例的`.to()`方法。

注意：这种方式必须显式的给出方法调用的目标对象时才有效。

3.2 局部函数

1. 可以在`Scala`的函数内部定义函数，这样的函数称作局部函数。就像局部变量一样，这样的局部函数只在包含它的代码块中可见，而不能从外部访问。

   局部函数可以直接访问包含它们的函数的参数。

```
import scala.io.Source
def processFile(filename:String,width:Int)={
    def processLine(line:String)={          // 局部函数
        if(line.length > width)
          println(filename+": "+line.trim) // 直接访问外部函数的参数
    }
    val source = Source.fromFile(filename)
    for(line <- source.getLines())
      processLine(line)
}
```

3.3 函数字面量&值函数

1. `Scala` 中，函数是一等公民。不仅可以定义函数并且调用它们，还可以用匿名的字面量来编写函数并将其作为`value`传递。
2. 函数字面量被编译成类，并且运行时实例化成值函数`function value`。因此，函数字面量和值函数的区别在于：函数字面量存在于源码，而值函数以对象形式存在于运行时。

   函数字面量简单示例：

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

   `=>` 表示函数将左侧的内容转换成右侧的内容。
3. 值函数是一个可调用对象（实现了`apply`方法），所以可以将它们存放在变量中。

   它们同时也是函数（因为具有`apply`方法），所以可以用常规的圆括号来调用。

   ```
   var f = (x:Int) => x+1
   f(10) // 结果为 11
   ```
4. 如果希望在函数字面量中包含多于`1`条语句，则可以将函数体用花括号`{}`括起来，每条语句占一行，组成一个代码块。

   跟方法一样，当值函数被调用时，所有的语句都会被执行，并且最后一个表达式求值结果就是该函数的返回值。

   ```
   var f = (x:Int) => {
       println("Hello")
       println("World")
       x+1
   }
   f(10) // 结果为 11
   ```

三、闭包

1. 闭包：运行时从函数字面量创建出来的值函数对象称做闭包`closure`。
2. 没有自由变量的函数字面量称作闭合语`closed term`。
   - 自由变量：函数字面量的函数体以及参数以外的变量。
   - 绑定变量：函数字面量的参数变量。

```
var more = 1
val f = (x:Int) => x + more // 闭包。 more：自由变量； x: 绑定变量
```

3. 如果在闭包创建之后再改变自由变量，则闭包能够看到这个改变。因为`Scala`的闭包捕获的是变量本身，而不是变量引用的值。

   同理，如果闭包对捕获的变量修改，则闭包之外也能看到这种修改。

```
val intList = List(1,2,3,4,5)
var sum = 0
val f = (x:Int) => sum += x          // 闭包
intList.foreach(f) // 闭包内部修改了变量 sum: sum 现在为 15
sum = 100          // 闭包外部修改了变量 sum
intList.foreach(f) // 闭包内部修改了变量 sum: sum 现在为 115
```

4. 如果闭包使用了某个函数的局部变量，而这个函数又被调用了多次，则：闭包引用的实例是在被创建时活跃的那个。

```
def make_f(more:Int) = (x:Int) => x+more // make_f 返回一个局部函数
val f = make_f(100) // 闭包引用的 more 是 10
println(1)          // 打印 11
```

   - `make_f`每调用一次就会创建一个新的闭包。每个闭包都会访问创建时活跃的变量`more`。其返回结果取决于闭包创建时的`more`的定义。
   - `Scala`编译器会重新组织和安排，让被捕获的参数在堆上继续存活（而不是由垃圾收集器自动回收）。

四、递归函数

1. 通常一个不断更新`var`的`while`循环可以修改为递归函数。

```
//************* while 版本 *****************//
def func(init_guess: Double) : Double = {
    var guess = init_guess
    while(!=is_ok(guess))         // 不满足条件，继续迭代
      guess = improve(guess)
    guess // 返回结果
}

//************* 递归函数 版本 *****************//
def func(guess : Double) : Double = {
    if (is_ok(guess)) guess        // 满足条件，返回
    else func(impove(guess))       // 不满足条件，继续递归
}
```

   - 从代码简洁性来讲，递归函数的版本更好。但是从执行效率上讲，通常`while`版本更好。
   - 如果递归函数是尾递归的，则这两个版本的执行效率一样高。尾递归`tail recursive`：递归函数的递归发生在函数的最后一步。

     因为`Scala`编译器会执行尾递归优化。当`Scala`检测到尾递归时，编译器将其替换为跳到函数的最开始，并且在跳转之前将参数更新为新的值。这样尾递归并不会在每次调用时都构建一个新的栈帧，所有的调用都会在同一个栈帧中执行。

这在调试尾递归函数时可能会是不利影响。如果你对调试信息感到困惑，可以将尾递归优化关闭，方法是 `scala` 命令或者 `scalac` 编译器添加参数 `-g:notailcalls`。 一旦调试结果，可以继续打开尾递归优化。

2. 在 `Scala` 中使用尾递归比较受限，因为 `Scala` 只能对那些在函数末尾直接调用自己的函数做优化。如果递归调用是间接的，则无法使用尾递归优化。
   - 最后一个表达式是递归调用之后再做了其它操作：无法使用尾递归优化。

```
def func(x: Int) : Int =
{
  if (x==0) 0
  else func(x-1) + 1   // 无法使用尾递归优化，因为最后一个表达式是 func 的间接调用：调用之后再加 1
}
```

   - 相互递归调用：无法使用尾递归优化。

```
def isEven(x: Int): Boolean = if (x==0) true else isOdd(x-1)
def isOdd(x: Int): Boolean = if (x==0) false else isEven(x-1)
```

   - 最后一步调用的是一个值函数（而不是发起调用的那个函数自己）：无法使用尾递归优化。

```
var funValue = (x :Int) => x+1
def func(x: Int) : Int = {
    if(x!=0) {println(x); funValue(x-1)}
    else 0
}
funValue = func _ // 现在 funValue 指向一个打包了 func 函数的值函数对象
```

五、自定义控制结构

1. 在拥有一等函数的语言中，可以有效地制作出新的控制接口，方法是：创建接收函数作为入参的方法。
2. 通常，当你发现某种模式在代码中多处出现，就应该考虑将这种模式实现为新的控制结构。

   如一个常见的编码模式：打开资源，对资源进行操作，然后关闭资源。这可以自定义一个控制抽象：

```
def withPrinter(file:File,op:Printer => Unit) = {
    val printer = new Printer(file)
    try {
        op(printer)
    } finally {
        printer.close()
    }
}
```

   然后可以方便的使用该方法来管理 `Printer`：

```
withPrinter(
    new File("data.txt"),             // file 实参：一个 File 对象
    printer => printer.printFile()    // op 实参：一个函数
)
```

这可以确保资源`printer`在最后关闭。这种技巧被称作`贷出模式 loan pattern`。

在例子中，`withPrinter`负责创建资源，然后将资源贷给`op`；当`op`完成时，表示不再需要资源，此时由`withPrinter`负责关闭资源。

3. 可以使用花括号而不是圆括号来表示参数列表，这样调用方的代码看上去更像是在使用内建的控制结构。

   注意：`Scala`中，这个花括号技巧仅对传入单个参数的场景适用。

```
println("Hello,scala!")  // 圆括号调用
println {"Hello,scala!"} // 花括号调用
val s = "Hello,scala!".substring {7,9} // 编译失败，必须用 substring(7,9)
```

   - 这个技巧是为了让调用方更便利的编写函数字面量，从而使得函数用起来更像是控制结构。
   - 如果有多个参数，则可以通过柯里化来适配该技巧。

```
def withPrinter(file:File)(op:Printer => Unit) = {
    val printer = new Printer(file)
    try {
        op(printer)
    } finally {
        printer.close()
    }
}
val file = new File("data.txt")
withPrinter(file) {                   // 现在 withPrinter 更像是控制结构了
    printer => printer.printFile()
}
```