# Hello World

|  |
| --- |
| object eveType{  def main(args:Array[String]){  println("hello , world!");  }  } |

可以将这段代码保存成一个文件，也可以直接在scala解释器中交互式执行，对于文件扩展名是没有什么要求的，可以任意指定。

如果保存成文件可以直接调用scala 文件名执行，也可以使用scalac命令先将文件编译成可执行文件，然后再使用scala命令执行

只有object定义的Singleton对象才可以直接执行，class中定义的main方法不能作为程序的入口

注意：Scala提供了一个快速编译代码的辅助命令fsc (fast scala compliler) ，使用这个命令，只在第一次使用fsc时启动JVM,之后fsc在后台运行，这样就避免每次使用scalac时都要载入相关库文件，从而提高编译速度。

# 交互式Scala解释器

输入Scala表达式，Scala解释器会立即解释执行该语句并输出结果

## 进入scala解释器

在cmd窗口中输入scala 回车即可进入scala交互式解释器

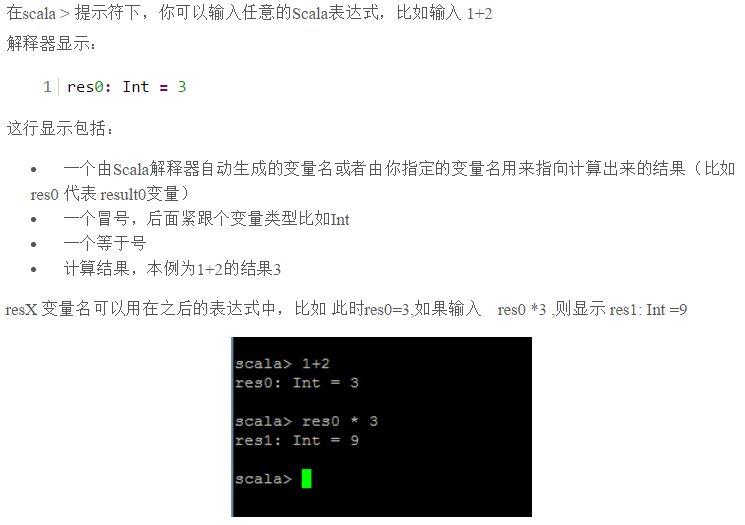
## 推出scala解释器

在解释器模式中输入：quit即可退出交互式解释器模式

## Scala解释器帮助

help 命令列出一些常用的Scala解释器命令

## 解释器中的scala表达式



# Scala脚本

Scala本身是设计用来编写大型应用的，但它也可以作为脚本语言来执行，脚本为一系列Scala表达式构成以完成某个任务，你可以使用脚本来实现一些比如复制文件，创建目录之类的任务。

# Scala语法

## 变量

Scala 定义了两种类型的变量 val 和 var

* val 类似于Java中的final 变量，一旦初始化之后，不可以重新赋值（我们可以称它为常变量）。
* var 类似于一般的非final变量。可以任意重新赋值。

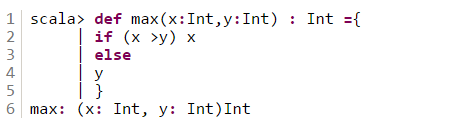
|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | scala> val msg="Hello,World" |  |  |  | | --- | --- | | msg: String = Hello,World |  | |

在scala中定义变量不需要指定变量的类型，scala会自己推测，称为type inference（类型推断）。

如果要指定变量的类型可以在变量后面加上类型，并用冒号将变量和变量类型隔开。如下所示： val msg：String = “hello world”

Scala中语句结尾不需要分号(；)

## 函数



scala函数以def定义，然后是函数的名称（如max)，然后是以逗号分隔的参数。Scala中变量类型是放在参数和变量的后面，以“：”隔开。这样做的一个好处是便于”type inference”。刚开始有些不习惯（如果你是Pascal程序员会觉的很亲切）。同样如果函数需要返回值，它的类型也是定义在参数的后面（实际上每个Scala函数都有返回值，只是有些返回值类型为Unit，类似java的void类型）。

此外每个scala表达式都有返回结果（这一点和Java，C#等语言不同），比如Scala的 if else 语句也是有返回值的，因此函数返回结果无需使用return语句。实际上在Scala代码应当尽量避免使用return语句。函数的最后一个表达式的值就可以作为函数的结果作为返回值。

同样由于Scala的”type inference”特点，本例其实无需指定返回值的类型。对于大多数函数Scala都可以推测出函数返回值的类型

### 函数字面量

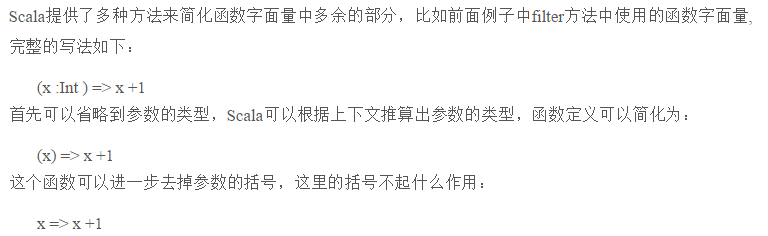
函数字面量的概念就像3是整数字面量一样

在scala中可以将一个函数字面量保存在一个函数变量中，例如

var add = (n1:Int,n2:Int) => println(n1+n2)

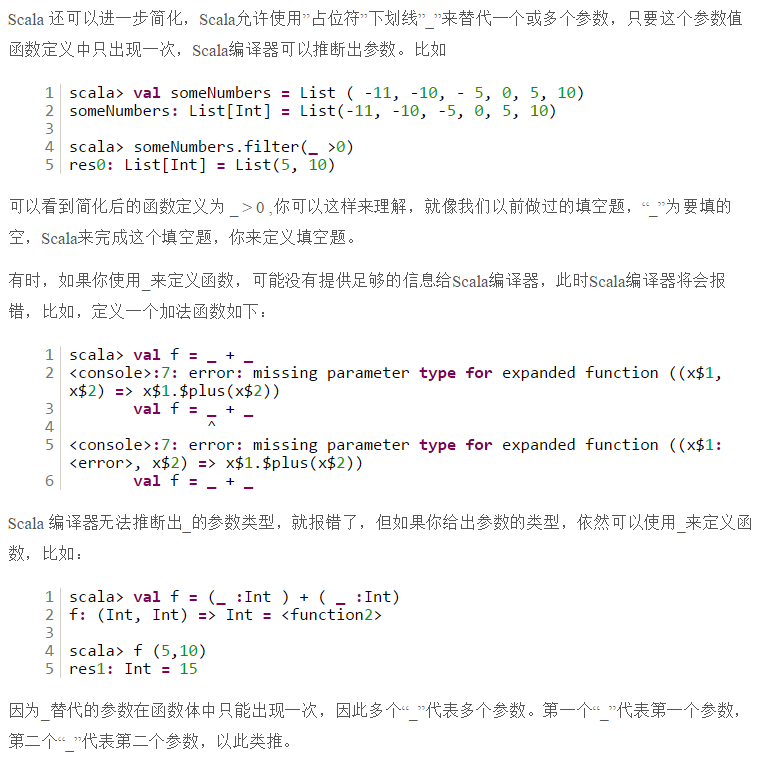
上面的表达式定义了一个含有两个Int型入参，作用为打印两个入参的和的函数字面量，然后将该函数字面量保存在了变量add中，在使用时，可以直接使用add(3,2)这种形式来使用该函数字面量

#### 函数字面量的一些简化写法

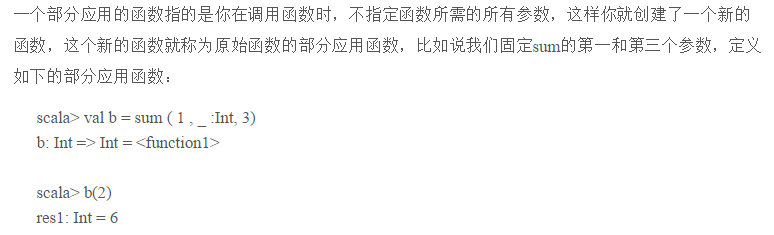


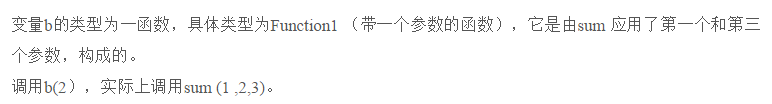
### 参数占位符

参数占位符主要用在含数字面量中和调用函数时通过占位符生成一个部分应用的函数，



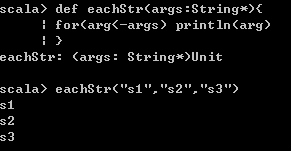
### 部分应用的函数

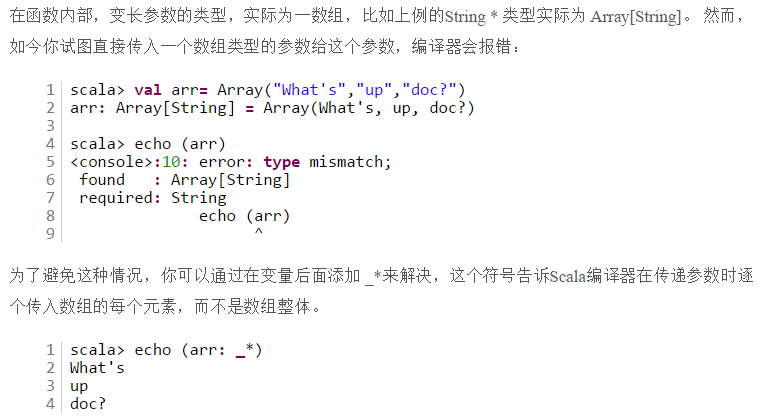




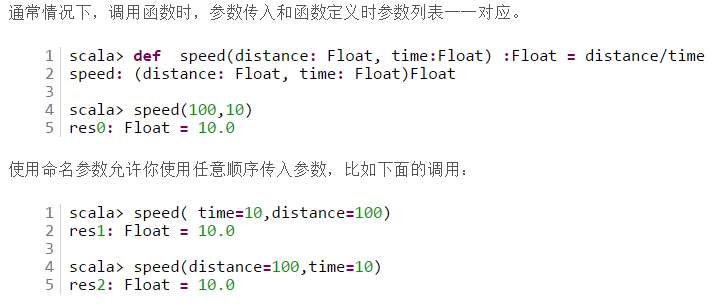
### 可变参数

scala在定义函数时允许指定最后一个参数可以重复（变长参数），从而允许函数调用者使用变长参数列表来调用该函数，Scala中使用“\*”来指明该参数为重复参数。

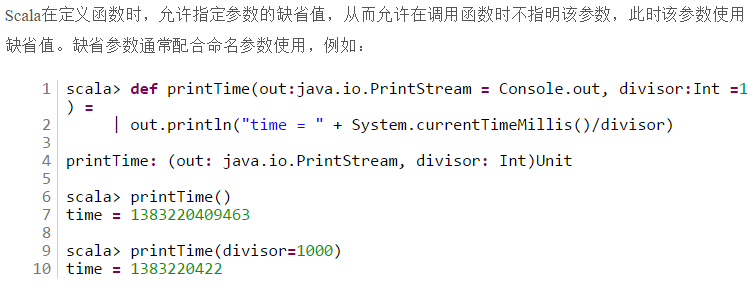




### 命名参数



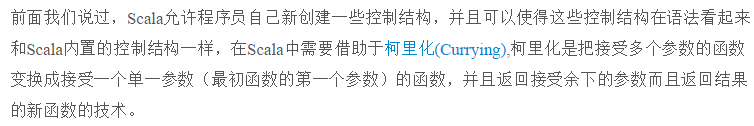
### 缺省参数值



### 语法糖

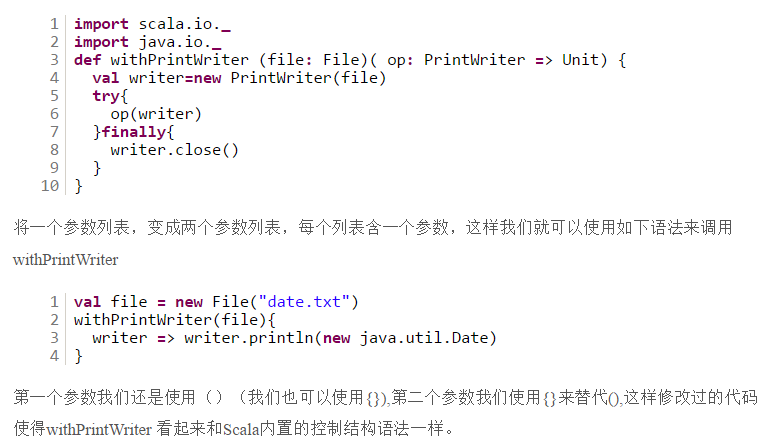
语法糖（Syntactic sugar），也译为糖衣语法，是由英国计算机科学家彼得·约翰·兰达（Peter J. Landin）发明的一个术语，指计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。通常来说使用语法糖能够增加程序的可读性，从而减少程序代码出错的机会。

### 柯里化函数



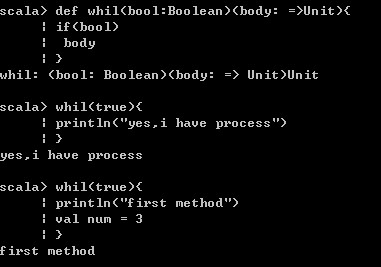
在Scala中如果你调用函数只有一个参数，你可以使用{}来替代().

如下，使用柯里化技术，定义一个含有两个参数的函数，然后第一个参数用（）传值，第二个参数用{}传值，这样就跟内置的控制结构语法看起来一样了



#### 一个自定义的whil循环结构体

使用到了传名参数的概念



第一条语句为定义一个柯里化函数，第一个参数为一个Boolean类型的变量，第二个参数为一个函数类型的变量，

### 传名参数

Scala的解释器在解析函数参数(function arguments)时有两种方式：

* 传值调用（call-by-value）：先计算参数表达式的值，再应用到函数内部；
* 传名调用（call-by-name）：将未计算的参数表达式直接应用到函数内部

在进入函数内部前，传值调用方式就已经将参数表达式的值计算完毕，而传名调用是在函数内部进行参数表达式的值计算的。

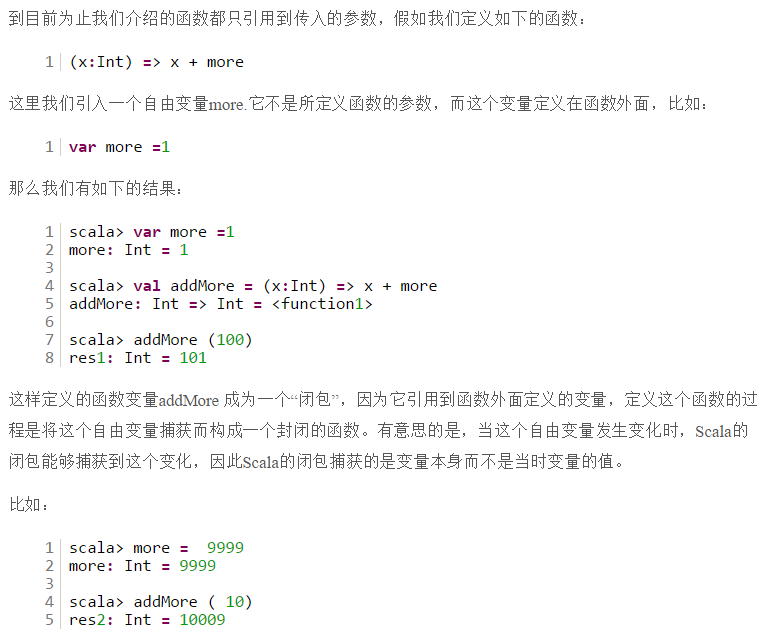
这就造成了一种现象，每次使用传名调用时，解释器都会计算一次表达式的值。

|  |
| --- |
| object Test {  def main(args: Array[String]) {  delayed(time());  }  def time() = {  println("获取时间，单位为纳秒")  System.nanoTime  }  def delayed( t: => Long ) = {  println("在 delayed 方法内")  println("参数： " + t)  t  }  } |

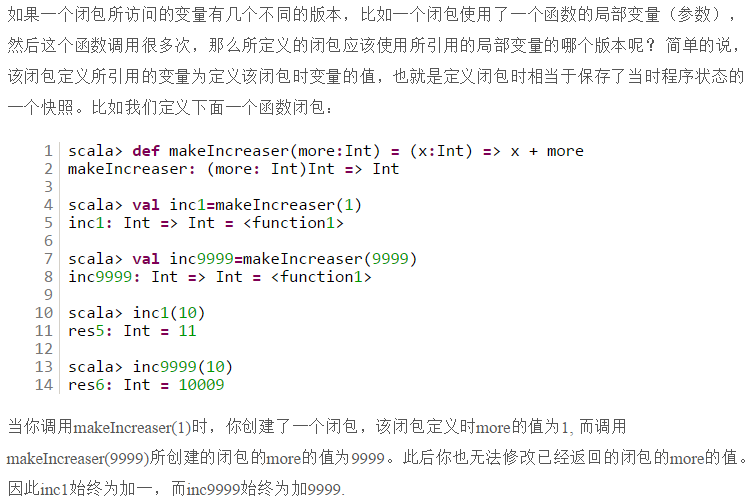
以上实例中我们声明了 delayed 方法， 该方法在变量名和变量类型使用 => 符号来设置传名调用。执行以上代码，输出结果如下：

|  |
| --- |
| $ scalac Test.scala  $ scala Test  在 delayed 方法内  获取时间，单位为纳秒  参数： 241550840475831  获取时间，单位为纳秒 |

## 闭包



### 闭包的场景

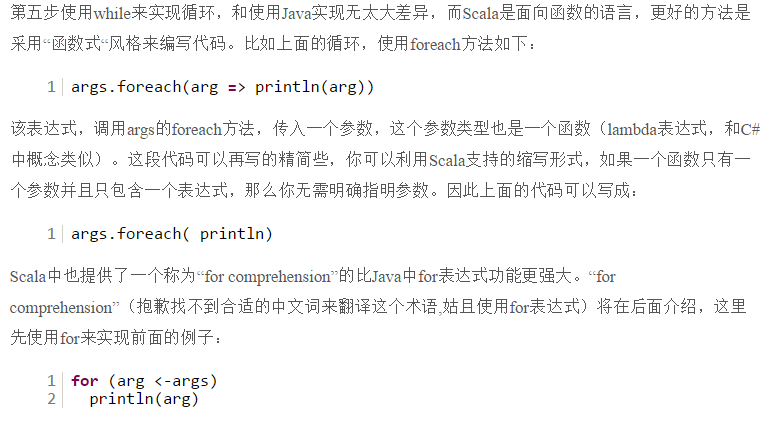


### 自己对闭包的理解

闭包这个概念我觉得是在函数进行传递的过程中产生的，比如，在一个方法中生成了一个函数，这个函数引用了方法的一个变量，当这个函数作为返回值被传递出去之后这个函数就是一个独立的函数字面量了，这个时候这个函数中引用的方法中的变量就一起被包含到了函数中并且在函数字面量中被引用过的变量不再和之前的方法中的变量有什么关系了，也就是说，被传递出去的函数字面量中的被引入的变量的值不会跟谁方法中的变量的值得变化而变化了，在Java中是没有闭包这一说的，只有在面向过程的语言中才会有这一概念。

## Foreach和for

Foreach为一个方法，需要一个函数类型的参数，就相当于回调函数



## For循环

### 最简单的scala for语句

for(i <- 1 to 10) {

    println("i is " + i);

}

上面的for i从1到10（包含10）来做循环

scala在for循环中对循环变量i的赋值用了“<-”符号，1 to 10指定了一个范围，i在这个范围内做循环，这个范围包括1和10

这里的1 to 10本质上是一个方法，这个方法返回的是一个Range类型，它等价于下面的代码

(1).to(10);

在scala中一切皆是对象所以，这里的1就是一个对象，他有一个to方法，返回一个Range。

如果在for循环中只有一条语句，那么可以省略掉花括号，如下代码

for(i <- 1 to 10)

    println("i is " + i);

### until 和 to关键字

在scala中还有一个和上面的to关键字有类似作用的关键字until，它的不同之处在于不包括最后一个元素

for(i <- 1 to 10) {

    println("i is " + i);

}

for(i <- 1 until 10) {

    println("i is " + i);

}

上面代码中的第一个循环会循环打印1到10包括10，而第二个循环打印1到9，不包含10

### for语句遍历数组

可以通过下面两种方法遍历数组元素

var myArray : Array[String] = new Array[String](10);

for(i <- 0 until myArray.length){

    myArray(i) = "value is: " + i;

}

for(value : String <- myArray ) {

    println(value);

}

上面代码中第一个for是通过数组下标来做遍历的，而第二个for是直接遍历数组的元素

### for语句和if条件结合过滤部分循环

在scala的for循环语句中可以使用if语句来做for条件过滤，例如

var myArray : Array[String] = new Array[String](10);

for(i <- 0 until myArray.length){

    myArray(i) = "value is: " + i;

}

for(value : String <- myArray if value.endsWith("5")) {

    println(value);

}

注意第二个for语句的括号内有if关键字，它后面紧跟if的条件使只有满足条件的元素来执行for语句块。上面的语句和下面的语句是类似的，不过更精简了一些

for(value : String <- myArray) {

    if value.endsWith("5"){

        println(value);

    }

}

scala的for语句中支持多个if条件，多个if之间用;来分割,如下例子

for(value : String <- myArray

    if value.endsWith("5");

    if value.indexOf("value") != -1 ) {

    println(value);

}

多个if条件必须同时满足，才会执行for语句块内的代码

上面的for语句和下面的语句是等价的

for(value : String <- myArray) {

    if( value.endsWith("5") && value.indexOf("value") != -1){

        println(value);

    }

}

### 嵌套循环

如下是一个嵌套for循环的例子，我们声明了一个二维的字符串数组，在for循环里面做遍历

var myArray : Array[Array[String]] = new Array[Array[String]](10);

for(i <- 0 until myArray.length){

    myArray(i) = new Array[String](10);

    for(j <- 0 until myArray(i).length){

      myArray(i)(j) = "value is: " + i + ", " + j;

    }

}

上面的嵌套for语句，也可以简写如下

for(anArray : Array[String] <- myArray;

    aString : String        <- anArray ) {

    println(aString);

}

上面的for语句中有两个范围的遍历，首先是anArray对myArray的遍历，然后是aString对anArray的遍历

scala的for循环中还支持中间变量的绑定

如下例子在嵌套的for循环中多了aStringUC的中间变量，这个中间变量是aString的大写,同时在这个for语句中还有两个if语句

for(anArray : Array[String] <- myArray;

    aString : String        <- anArray;

    aStringUC = aString.toUpperCase()

    if aStringUC.indexOf("VALUE") != -1;

    if aStringUC.indexOf("5") != -1     ) {

     println(aString);

}

## 数组

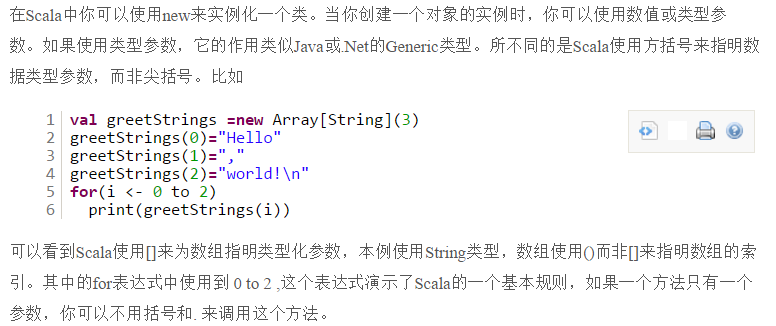
如果一个方法只有一个参数，你可以不用括号和. 来调用这个方法

比如：for(i<-0 to 2){

Println(i)

}

上例中0 to 2实则是0.to(2)。在scala中基本数据类型也是对象



实际上Scala中表达式 1+2 ,最终解释为(1).+(2) +也是Int的一个方法，和Java不同的是，Scala对方法的名称没有太多的限制，你可以使用符合作为方法的名称。

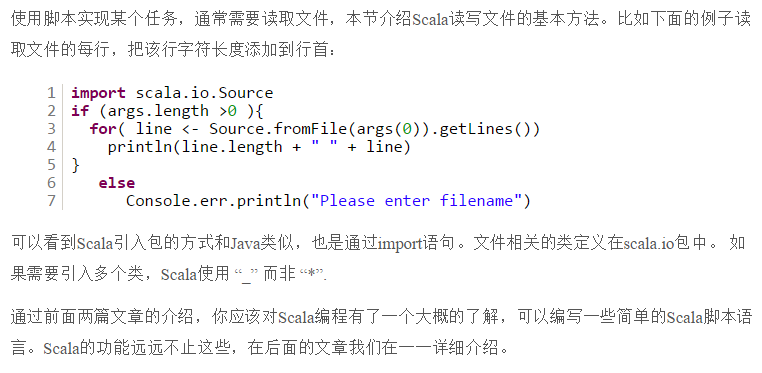
这里也说明为什么Scala中使用（）来访问数组元素，在Scala中，数组和其它普遍的类定义一样，没有什么特别之处，当你在某个值后面使用（）时，Scala将其翻译成对应对象的apply方法。因此本例中greetStrings(1) 其实调用greetString.apply(1)方法。这种表达方法不仅仅只限于数组，对于任何对象，如果在其后面使用（）,都将调用该对象的apply方法。



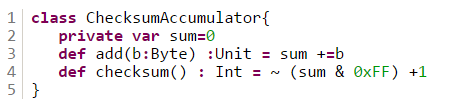
## 函数式编程风格

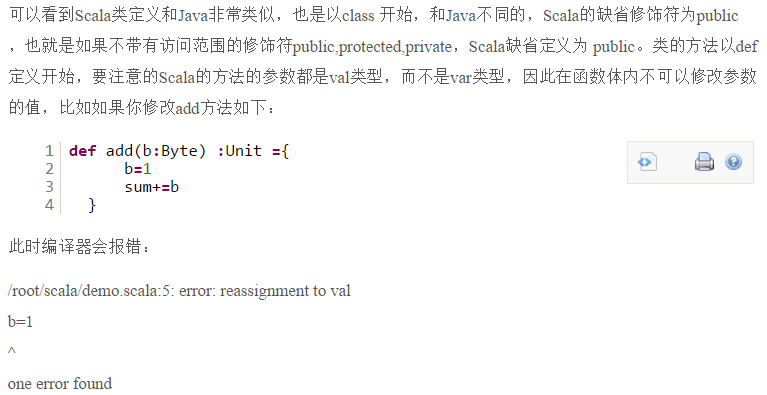


## 读取文件



## 类定义





### 类参数

Scala定义类时可以在类名后面指定类参数，比如：



Scala使用类参数，并把类定义和主构造函数合并在一起，在定义类的同时也定义了类的主构造函数。

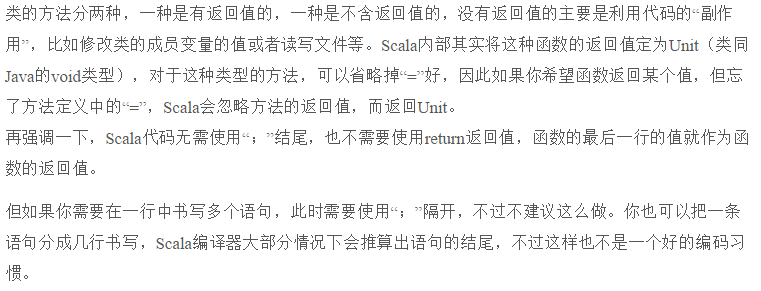
Scala编译器会编译Scala类定义包含的任何不属于类成员和类方法的其它代码，这些代码将作为类的主构造函数

### 辅助构造

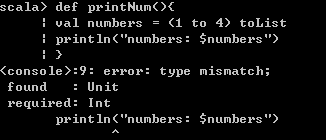
## 伴生对象

Scala中没有静态变量，静态方法这一说，但有一个伴生对象，伴生对象中的变量和方法都是静态的，在其他类中可以直接用类名.方法名的形式访问，伴生对象的声明用object关键字，如果有一个使用class关键字定义的类和伴生对象使用相同的名字定义，那么这个类就是这个伴生对象的伴生类，对于伴生类中的私有方法，可以在伴生对象的方法中new出一个伴生类的对象，然后调用其私有方法。同样，在伴生类的方法中可以使用类名.方法名的形式访问伴生对象的私有方法。伴生对象和伴生类要放在同一个文件中。

## 方法

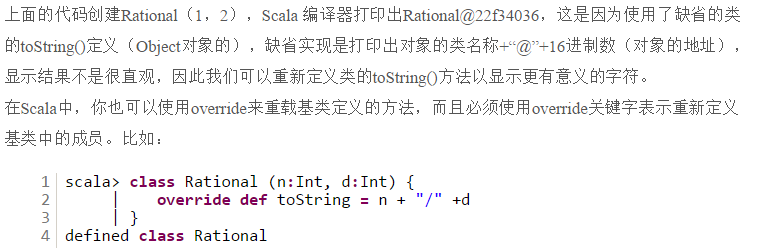


### 错误分析

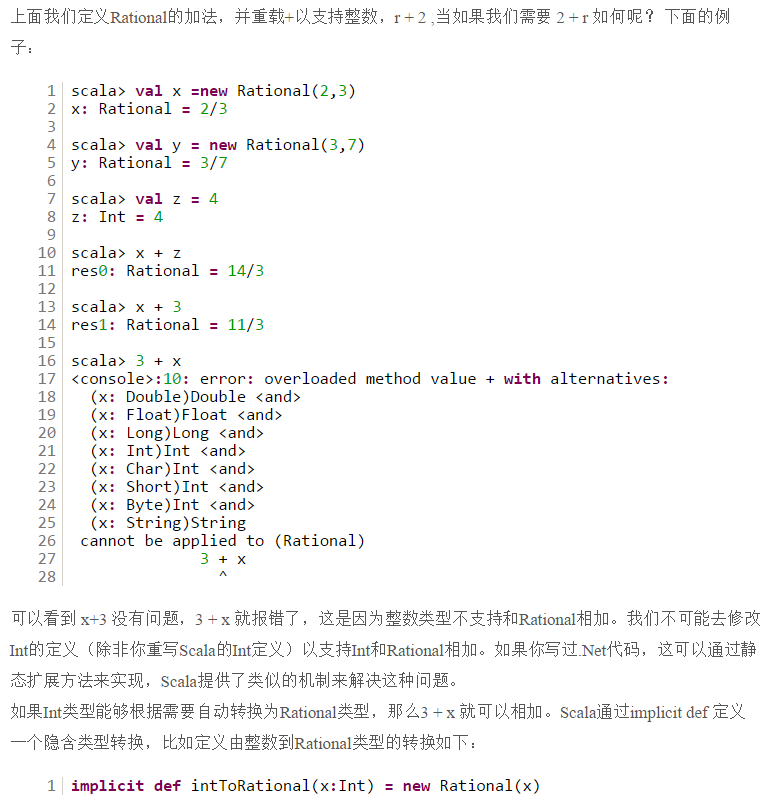


自定义一个printNum方法，方法体中，第一行获得一个list对象，第二行打印

## 方法重写



## 隐式类型转换



在scala语言中有很多隐式类型转换，如果我们想定义自己的隐式类型转换，可以使用implicit关键字定义一个方法，将该方法中的参数自动隐式转换为返回值类型的数据

## scala模块化语言特性

引入的原因是scala中的某些特性比较难懂或有些晦涩，或有些特性是试验性质。

这些特性在普通场景下不鼓励使用，但在特定场景下又可能需要。所以引入了“显式引入语言特性”的做法。

举例来说，在2.10的repl下定义隐式转换：

scala> implicit def string2MyString(s:String) = new MyString(s)

warning: there were 1 feature warning(s); re-run with -feature for details

string2MyString: (s: String)MyString

好，我们重新启动repl，启用 -feature 选项：

$ scala -feature

scala> class MyString(str:String) { def say() {println(str)} }

defined class MyString

scala> implicit def string2MyString(s:String):MyString = new MyString(s)

<console>:8: warning: implicit conversion method string2MyString should be enabled

by making the implicit value language.implicitConversions visible.

This can be achieved by adding the import clause 'import scala.language.implicitConversions'

or by setting the compiler option -language:implicitConversions.

See the Scala docs for value scala.language.implicitConversions for a discussion

why the feature should be explicitly enabled.

implicit def string2MyString(s:String):MyString = new MyString(s)

^

string2MyString: (s: String)MyString

这回给出了详细的警告信息，使用隐式转换方法，建议在代码中引入language单例的implicitConversions成员： import scala.language.implicitConversions 或者 编译时指定-language:implicitConversions参数

为什么要控制隐式转换这一特性(要显式启用)？是因为过度使用隐式转换会有很多陷阱，而现在已经有这个趋势，因为隐式转换的强大而被过度使用。另外，相对隐式转换，采用隐式参数对设计更好一些。

类似的控制还有后缀操作符：

scala> "hello" length //中间用空格，省略了点

也会给出警告提示import scala.language.postfixOps或编译时指定参数-language:postfixOps

这是因为后缀操作符与分号推理(semicolon inference)容易冲突，举个例子：

//有参方法：a.foo(b) 也可以写成 a foo b

//无参方法：a.foo() 也可以写成 a foo

//如果一个类中同时定义了上面两个方法

scala> class A { def foo(i:Int)=i+1; def foo()=2}

// 没有分号，解析为 a.foo(2)

scala> val r = { a foo 2 }

r: Int = 3

// 有分号的话解析为 a.foo() 分号后边的是下一条语句

scala> val r = { a foo; 2 }

warning: there were 1 feature warning(s); re-run with -feature for details

r: Int = 2

// 但在换行情况下

scala> :pas

// Entering paste mode (ctrl-D to finish)

val r = {

a foo // 这里没写分号

2

}

// Exiting paste mode, now interpreting.

r: Int = 3

结果被解析为 a.foo(3) ！！

如果没有分号，就无法解析为a.foo()，总会把下一条语句也当作参数，比如：

scala> :pas

// Entering paste mode (ctrl-D to finish)

var r = {

a foo // 这里没有写分号

println("another")

}

// Exiting paste mode, now interpreting.

<console>:19: error: type mismatch;

found : Unit

required: Int

println("another")

^

所以为了避免这种情况，后缀操作符也不鼓励，只有特定场景需要时(DSL)，显式的打开这个选项；否则会警告。

## 前提条件检查

