# http://www.yiibai.com/scala/scala\_strings.html

# 环境安装与设置

Scala语言可以安装在任何类UNIX或Windows系统。要安装Scala，必须先安装Java1.5或更高版本安装在计算机上。

**Windows上安装Scala：**

**步骤(1)：JAVA设置：**

首先，必须设置JAVA\_HOME环境变量和将JDK的bin目录添加到PATH变量。要验证是否可以工作，在命令提示符下，键入：java -version，然后按Enter。应该看到类似以下内容：

C:>java -version java version "1.6.0\_15" Java(TM) SE Runtime Environment (build 1.6.0\_15-b03) Java HotSpot(TM) 64-Bit Server VM (build 14.1-b02, mixed mode) C:>

接下来测试，看看已安装Java编译器。输入javac -version。应该看到类似以下内容：

C:>javac -version javac 1.6.0\_15 C:>

**步骤(2)：SCALA设置：**

接下来可以下载Scala从以下网址[http://www.scala-lang.org/downloads](http://www.scala-lang.org/downloads" \t "_blank). 在写这篇教程的时候，下载的是scala-2.9.0.1-installer.jar，并把它放在 C:/> 目录。请确保有管理员权限进行。现在在命令提示符下键入以下命令执行：

C:>java -jar scala-2.9.0.1-installer.jar C:>

上面的命令将显示一个安装向导，该向导将引导在Windows计算机上安装。在安装过程中，它会询问许可协议，接受它，并进一步询问Scala将要安装的 路径。选择的默认给出的路径 C:Program Filesscala，可以选择一个合适的路径。最后打开一个新的命令提示符，然后键入scala -version，然后按Enter键。应该会看到以下内容：

C:>scala -version Scala code runner version 2.9.0.1 -- Copyright 2002-2011, LAMP/EPFL C:>

恭喜已经成在Windows机器上安装Scala。下一节将演示如何在Mac OS X和Unix / Linux机器上安装Scala。

**在Mac OS X和Linux上安装Scala**

**步骤（1）：JAVA设置：**

请确保拥有Java JDK1.5或更高版本已经计算机上安装并设置JAVA\_HOME环境变量，以及JDK的bin目录添加到PATH变量。要验证是否安装正确，在命令提示符下，键入java -version，然后按Enter。应该看到类似以下内容：

$java -version java version "1.5.0\_22" Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0\_22-b03) Java HotSpot(TM) Server VM (build 1.5.0\_22-b03, mixed mode) $

接下来，测试，看看已安装的Java编译器。输入javac-version。应该看到类似以下内容：

$javac -version javac 1.5.0\_22 javac: no source files Usage: javac <options> <source files> ................................................ $

**步骤（2）：SCALA设置：**

接下来可以下载Scala 从以下网址 [http://www.scala-lang.org/downloads](http://www.scala-lang.org/downloads" \t "_blank). 在写这篇教程的时候，下载的是Scala-2.9.0.1-installer.jar，并把它放在/tmp目录。请确保拥有管理员权限才能进行操作。现在，在命令提示符下键入以下命令执行：

$java -jar scala-2.9.0.1-installer.jar Welcome to the installation of scala 2.9.0.1! The homepage is at: http://scala-lang.org/ press 1 to continue, 2 to quit, 3 to redisplay 1 ................................................ [ Starting to unpack ] [ Processing package: Software Package Installation (1/1) ] [ Unpacking finished ] [ Console installation done ] $

在安装过程中，它会询问许可协议，接受它键入1，它会问在哪里Scala安装。这里是在目录/usr/local/share，可以选择一个合适的路径。最后，打开一个新的命令提示符，然后键入Scala的-version，然后按Enter键。应该看到以下内容：

$scala -version Scala code runner version 2.9.0.1 -- Copyright 2002-2011, LAMP/EPFL $

恭喜，已经在UNIX/Linux机器上安装了Scala。

# 基础语法

如果有很好的了解Java语言，那么将很容易学习Scala。 Scala和Java间的最大语法的区别在于;行结束符是可选的。考虑Scala程序它可以被定义为通过调用彼此方法进行通信的对象的集合。现在，简要地看看什么叫做类，对象，方法和临时变量。

* 对象 - 对象有状态和行为。例如：狗有状态 - 颜色，名字，品种，它有行为 - 摇摆，吠叫，吃东西。对象是类的一个实例。
* 类 - 类可以被定义为一个模板/蓝本以描述行为/指示其类型支持对象。
* 方法 - 方法本质上是一个行为。类可以包含许多方法。它是在将逻辑写入方法中，数据进行操作和所有动作被执行。
* 字段 - 每个对象都有其独特的一组临时变量，这是所谓的字段。对象的状态是由分配给这些字段中的值创建的。

## 第一个Scala程序：

### 交互式模式编程：

调用解释不通过一个脚本文件作为一个参数会显示以下提示：

C:>scala

Welcome to Scala version 2.9.0.1

Type in expressions to have them evaluated.

Type :help for more information.

scala>

键入下列文字的Scala提示符，然后按Enter键：

scala> println("Hello, Scala!");

这将产生以下结果：

Hello, Scala!

### 脚本模式编程：

让我们来看一个简单的代码，用于打印简单的一句话：Hello, World!

object HelloWorld {

/\* This is my first java program.

\* This will print 'Hello World' as the output

\*/

def main(args: Array[String]) {

println("Hello, world!") // prints Hello World

}

}

让我们来看看如何保存文件，编译并运行该程序。请按照以下的步骤：

1. 打开记事本，并添加上面代码。
2. 将文件保存为：HelloWorld.scala。
3. 打开命令提示符窗口，然后转到保存程序文件的目录。假设它是 C:>
4. 键入“scalac HelloWorld.scala”，然后按回车编译代码。如果代码中没有错误，命令提示符下将自动换到下一行。
5. 上面的命令将在当前目录中生成几个类文件。其中一个名称为HelloWorld.class。这是一个字节码可以运行在Java虚拟机（JVM）。
6. 现在，键入“scala HelloWorld”来运行程序。
7. 可以看到“Hello, World!”打印在窗口上。

C:> scalac HelloWorld.scala

C:> scala HelloWorld

Hello, World!

## 基础语法

关于Scala程序，这是非常要注意以下几点。

* **区分大小写** -  Scala是大小写敏感的，这意味着标识Hello 和 hello在Scala中会有不同的含义。
* **类名** - 对于所有的类名的第一个字母要大写。  
    
  如果需要使用几个单词来构成一个类的名称，每个单词的第一个字母要大写。  
    
  示例：class MyFirstScalaClass
* **方法名称** - 所有的方法名称的第一个字母用小写。  
    
  如果若干单词被用于构成方法的名称，则每个单词的第一个字母应大写。  
    
  示例：def myMethodName()
* **程序文件名** - 程序文件的名称应该与对象名称完全匹配。  
    
  保存文件时，应该保存它使用的对象名称（记住Scala是区分大小写），并追加“.scala”为文件扩展名。 （如果文件名和对象名称不匹配，程序将无法编译）。  
    
  示例: 假设“HelloWorld”是对象的名称。那么该文件应保存为'HelloWorld.scala“
* **def main(args: Array[String])** - Scala程序从main()方法开始处理，这是每一个Scala程序的强制程序入口部分。

## Scala修饰符:

所有的Scala的组件需要名称。使用对象，类，变量和方法名被称为标识符。关键字不能用作标识符和标识是区分大小写的。Scala支持以下四种类型标识符：

### 文字标识符

字母数字标识符，开始以字母或下划线，可以使用字母，数字或下划线。“$”字符在Scala中是保留关键字，标识符不能使用。以下是合法的字母标识符：

age, salary, \_value, \_\_1\_value

以下是非法标识符：

$salary, 123abc, -salary

### 运算符标识

运算符识别符由一个或多个运算符字符。操作字符是可打印的ASCII字符，如+, :, ?, ~ 或#。以下是合法的运算符标识：

+ ++ ::: <?> :>

Scala编译器将在内部“轧”操作符标识符使它们成为合法的Java标识符，并嵌入$字符。例如，所述标识符:->将内部表示为$colon$minus$greater。

### 混合标识符

混合标识符由一个字母数字识别符，随后是一个下划线和运算符标识。以下是合法的混合标识符：

unary\_+, myvar\_=

在这里，作为一个方法名unary\_+定义了一个一元+运算符和myvar\_=用来作为方法名称定义了一个赋值运算符。

### 立即数标识符

一个文字标识是包含在反引号(` . . . `)的任意字符串。以下是合法的文字标识：

`x` `<clinit>` `yield`

## Scala关键字:

下面的列表显示在Scala中的保留字。这些保留关键字不可以被用作常量或变量，或任何其他的标识符名称。

|  |  |  |  |
| --- | --- | --- | --- |
| abstract | case | catch | class |
| def | do | else | extends |
| false | final | finally | for |
| forSome | if | implicit | import |
| lazy | match | new | null |
| object | override | package | private |
| protected | return | sealed | super |
| this | throw | trait | try |
| true | type | val | var |
| while | with | yield |  |
| - | : | = | => |
| <- | <: | <% | >: |
| # | @ |  |  |

## Scala中的注释

Scala支持单行和多行注释非常类似于Java。多行注释可以嵌套，但必须正确嵌套。可用的任何注释和所有字符都将被Scala编译器忽略。

object HelloWorld {

/\* This is my first java program.

\* This will print 'Hello World' as the output

\* This is an example of multi-line comments.

\*/

def main(args: Array[String]) {

// Prints Hello World

// This is also an example of single line comment.

println("Hello, world!")

}

}

## 空行和空格：

仅包含空格，可能带有注释行，被称为一个空行，并且Scala将会完全忽略它。标记可以通过空格和/或注释分开。

## 换行符：

Scala是面向行的语言，语句可以用分号（;）结束或换行符。分号在声明的结尾通常是可选的。可以键入一个想要的，如果语句出现其本身在一行。在另一方面，如果写在一行多个语句分号是必需的：

val s = "hello"; println(s)

## Scala包：

包是一个代码命名的模块。例如，Lift 实用包net.liftweb.util。包声明是在源文件中的第一个非注释行，如下所示：

package com.liftcode.stuff

Scala的包可以被导入，使他们能够在当前编译范围内被引用。下面的语句是导入scala.xml包的内容：

import scala.xml.\_

可以导入封装的类和对象，例如，从scala.collection.mutable导入HashMap：

import scala.collection.mutable.HashMap

可以从一个单一封装的scala.collection.immutable包导入多个类或对象，例如，TreeMap和TreeSet：

import scala.collection.immutable.{TreeMap, TreeSet}

# 数据类型

Scala的数据类型全部相同于Java中，具有相同的内存占用和精度。以下表是有关可在Scala中所有的数据类型的细节：

|  |  |
| --- | --- |
| **数据类型** | **描述** |
| Byte | 8位有符号值。范围从-128到127 |
| Short | 16位有符号值。范围从-32768至32767 |
| Int | 32 位有符号值。范围从 -2147483648 to 2147483647 |
| Long | 64位有符号值。 从-9223372036854775808到9223372036854775807 |
| Float | 32位IEEE754单精度浮点数 |
| Double | 64位IEEE754双精度浮点数 |
| Char | 16位无符号Unicode字符。范围由U+0000至U+FFFF |
| String | 字符序列 |
| Boolean | 无论是字面true或false字面 |
| Unit | 对应于没有值 |
| Null | 空或空引用 |
| Nothing | 每一个其他类型的子类型; 包括无值 |
| Any | Any类型的超类型;任何对象是任何类型 |
| AnyRef | 任何引用类型的超类型 |

上面列出的所有的数据类型是对象。没有原始的类型，如Java中。这意味着可以调用int，long等上的方法

**Scala基本字面值：**

Scala中使用的文字规则是简单和直观的。这部分解释了所有Scala中的基本文字。

**整数值**

整数文字通常是int类型，或类型龙其次是L或l后缀的时候。下面是一些整数文字：

0

035

21

0xFFFFFFFF

0777L

**浮点型**

浮点文字是Float类型后跟一个浮点类型后缀F或f时，否则是Double类型。下面是一些浮点文字：

0.0

1e30f

3.14159f

1.0e100

.1

**布尔类型**

布尔字面值true和false是Boolean类型的成员。

**符号字面值**

符号常量'x是简写的表达scala.Symbol(“X”)。Symbol是一个类，它的定义如下。

package scala

final case class Symbol private (name: String) {

override def toString: String = "'" + name

}

**字符**

字符文字是用引号括起来的单个字符。该字符是一个可打印的Unicode字符或转义序列描述。这里有一些字符文字：

'a'

'u0041'

'

'

' '

**字符串常量**

字符串常量是在双引号中的字符序列。字符或者是可打印Unicode字符或转义序列描述。下面是一些字符串：

"Hello,

World!"

"This string contains a " character."

**多行字符串**

多行字符串是包含在三重引号“”字符序列“...”“”。字符序列是任意的，不同之处在于它可以包含三个或更多个连续引号字符仅在未尾。

字符未必是可打印;新的一行或其他控制字符也是允许的。这里是一个多行字符串：

"""the present string

spans three

lines."""

**NULL值**

null是scala.Null类型一个值，因此这个引用类型兼容。它表示参考值是指一种特殊的“空”的对象。

**转义序列：**

以下转义序列在字符和字符串。

|  |  |  |
| --- | --- | --- |
| **转义序列** | **Unicode** | **描述** |
|  | u0008 | 退格BS |
|  | u0009 | 水平制表符HT |
|  | u000c | 换页FF |
| f | u000c | 换页FF |
|  | u000d | 回车CR |
| " | u0022 | 双引用 " |
| ' | u0027 | 单引用 . |
|  | u005c | 反斜线 |

介于0和255字符使用Unicode也可以通过一个八进制转义，也就是说，一个反斜杠“”之后的最多三个八进制字符的序列表示。以下为例子来说明一些转义字符序列：

object Test {

def main(args: Array[String]) {

println("Hello World

" );

}

}

当上述代码被编译和执行时，它产生了以下结果：

Hello World

# 变量

变量是只不过是保留内存位置来存储值。这意味着，当创建一个变量，会在内存中保留一些空间。

基于变量的数据类型，所述编译器分配存储器，并决定什么可以被存储在保留存储器。因此，通过分配不同的数据类型的变量，可以存储整数，小数，或字符在这些变量。

**变量声明**

Scala的变量声明有不同的语法，它们可以被定义为值，即，常量或变量。下面是使用var关键字来定义一个变量的语法：

var myVar : String = "Foo" //变量

在这里，myVar使用关键字var声明。这意味着它是一个可以改变值的变量，被称为可变变量。下面是使用val关键字来定义变量的语法：

val myVal : String = "Foo" //常量

这里，myVal是使用关键字val声明。这意味着，它是不能改变的变量，这是所谓的不可变变量。

**变量数据类型：**

变量类型在变量名后面指定，前在equals之前签署。可以通过它的数据类型定义任何类型的Scala变量如下：

val or val VariableName : DataType [= Initial Value]

如果不指定任何初始值变量，那么如下所示它是有效的：

var myVar :Int;

val myVal :String;

**变量类型推断（根据声明时赋的初始值推断）：**

当分配一个初始值给一个变量，Scala编译器可以计算出根据分配给它的值的变量类型。这就是所谓的变量类型推断。因此，可以写这些变量的声明如下这样：

var myVar = 10;

val myVal = "Hello, Scala!";

在这里，默认情况下，myVar是int类型，将设为myVal为字符串类型变量。

**多重任务：**

Scala支持多任务。如果一个代码块或方法返回一个元组，该元组可被分配给一个val变量。 [注：元组在以后的章节中学习。]

val (myVar1: Int, myVar2: String) = Pair(40, "Foo")

类型推断得到正确类型：

val (myVar1, myVar2) = Pair(40, "Foo")

**变量类型：**

Scala变量可以有三种不同的范围，这取决于它们正在使用的地方。它们可以存在于字段，方法参数和局部变量。下面是关于每种类型的适用范围的详细信息：

**字段:**

字段是属于一个对象的变量。该字段是从对象中的所有方法中访问。字段也可以是具体取决于访问修饰符字段与声明的对象外访问。对象字段既可以是可变的或不可变的类型，可以使用var 或 val两种定义。

**方法参数：**

方法的参数是变量，其用于传递方法内的值在方法被调用时。方法参数被从方法唯一传入，但传递的对象可从外面，如果从外面的方法有一个引用对象。方法参数始终是可变的并由val关键字定义。

**局部变量：**

局部变量是在方法内声明的变量。局部变量只能在方法中访问，但可以访问创建方法的对象（如果从方法中返回它们）。局部变量既可以是可变的或不可变的类型，可以使用var或val两个关键字定义。

# 访问修饰符

包，类或对象的成员可以标记访问修饰符private和protected，如果我们不使用这两种关键字，那么访问将被默认设置为public。这些修饰 限制为成员的代码的某些区域访问。要使用访问修饰符，包括它的包，类或对象的成员定义的关键字，我们将在下一节中看到。

**私有成员：**

私有成员只能看到里面包含的成员定义的类或对象。下面是一个例子：

class Outer {

class Inner {

private def f() { println("f") }

class InnerMost {

f() // OK

}

}

(new Inner).f() // Error: f is not accessible

}

在Scala中，访问 (new Inner).f() 是非法的，因为f被声明为private内部类并且访问不是在内部类内。与此相反，到f第一接入类最内层是确定的，因为该访问包含在类内的主体。 Java将允许这两种访问，因为它可以让其内部类的外部类访问私有成员。

**保护成员：**

受保护的成员是从该成员定义的类的子类才能访问。下面是一个例子：

package p {

class Super {

protected def f() { println("f") }

}

class Sub extends Super {

f()

}

class Other {

(new Super).f() // Error: f is not accessible

}

}

类分给 f 访问是正常的，因为f被声明为受保护的超类和子类是超级的子类。相比之下，访问 f 在其他是不允许的，因为其他没有从超级继承。在Java中，后者访问将仍然允许的，因为其他的是在同一个包子。

**公共成员：**

未标示私有或受保护的每一个成员是公开的。不需要明确使用修饰符public。这样的成员可以从任何地方访问。下面是一个例子：

class Outer {

class Inner {

def f() { println("f") }

class InnerMost {

f() // OK

}

}

(new Inner).f() // OK because now f() is public

}

**保护范围：**

Scala中的访问修饰符可以增加使用修饰符。形式：private[X]或protected[X]的修饰符意味着访问私有或受保护“达到”X，其中X代表了一些封闭的包，类或单个对象。考虑下面的例子：

package society {

package professional {

class Executive {

private[professional] var workDetails = null

private[society] var friends = null

private[this] var secrets = null

def help(ex : Executive) {

println(ex.workDetails)

println(ex.secrets) //ERROR why？

}

}

}

}

注意，上面的例子中以下几点：

* 变量workDetails将可对任何一类封闭包professional范围内。
* 变量friends 将可对任何一类封闭包society中。
* 变量secrets 将可只在实例方法隐含的对象（this）。

# 运算符

运算符是一个符号，告诉编译器执行特定的数学或逻辑操作。 Scala有丰富的内置运算符，运算符提供了以下类型：

* 算术运算符
* 关系运算符
* 逻辑运算符
* 位运算符
* 赋值运算符

本章将学习研究算术，关系，逻辑，按位，分配赋值和其他运算符。

**算术运算符：**

Scala语言支持以下算术运算符：

假设变量A=10和变量B=20，则：

[查看例子](http://www.yiibai.com/scala/scala_arithmatic_operators.html" \o "算术运算符" \t "_blank)

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **示例** |
| + | 两个操作数相加 | A + B = 30 |
| - | 从第一操作减去第二操作数 | A - B = -10 |
| \* | 两个操作数相乘 | A \* B = 200 |
| / | 通过分子除以分子 | B / A = 2 |
| % | 模运算，整数除法后的余数 | B % A = 0 |

**关系运算符：**

Scala语言支持以下关系运算符

假设变量A=10和变量B=20，则：

[查看示例](http://www.yiibai.com/scala/scala_relational_operators.html" \o "关系运算符" \t "_blank)

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **示例** |
| == | 检查两个操作数的值是否相等，如果是的话那么条件为真。 | (A == B) 不为 true. |
| != | 检查两个操作数的值是否相等，如果值不相等，则条件变为真。 | (A != B) 为 true. |
| > | 检查左边的操作数的值是否大于右操作数的值，如果是的话那么条件为真。 | (A > B) 不为 true. |
| < | 检查左边的操作数的值是否小于右操作数的值，如果是的话那么条件为真。 | (A < B) 为 true. |
| >= | 检查左边的操作数的值是否大于或等于右操作数的值，如果是的话那么条件为真。 | (A >= B) 不为 true. |
| <= | 检查左边的操作数的值是否小于或等于右操作数的值，如果是的话那么条件为真。 | (A <= B) 为 true. |

**逻辑运算符(0为假，非0为真)：**

Scala语言支持以下逻辑运算符

假设变量A=1和变量B=0，则：

[查看示例](http://www.yiibai.com/scala/scala_logical_operators.html" \o "逻辑运算符" \t "_blank)

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **示例** |
| && | 所谓逻辑与操作。如果两个操作数为非零则条件为真。 | (A && B) 为 false. |
| || | 所谓的逻辑或操作。如果任何两个操作数是非零则条件变为真。 | (A || B) 为 true. |
| ! | 所谓逻辑非运算符。使用反转操作数的逻辑状态。如果条件为真，那么逻辑非操作符作出结果为假。 | !(A && B) 为  true. |

**位运算符：**

位运算符适用于位和位操作进行一下。 ：&, |, 和^ 其真值表如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

假设，如果A =60;B =13;现在以二进制格式它们将如下：

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011

Scala语言支持位运算符列在下表中。假设变量A=60和变量B=13，则：

[查看示例](http://www.yiibai.com/scala/scala_bitwise_operators.html" \o "位运算符" \t "_blank)

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **示例** |
| & | 二进制和运算符副本位的结果，如果它存在于两个操作数。 | (A & B) = 12, 也就是 0000 1100 |
| | | 二进制或操作拷贝，如果它存在一个操作数。 | (A | B) = 61, 也就是 0011 1101 |
| ^ | 二进制异或运算符的副本，如果它被设置在一个操作数而不是两个比特。 | (A ^ B) = 49, 也就是 0011 0001 |
| ~ | 二进制的补运算符是一元的，具有“翻转”位的效应。 | (~A ) = -61, 也就是 1100 0011在2补码形式，由于一个带符号二进制数。 |
| << | 二进制左移位运算符。左边的操作数的值向左移动由右操作数指定的位数。 | A << 2 = 240, 也就是 1111 0000 |
| >> | 二进制向右移位运算符。左边的操作数的值由右操作数指定的位数向右移动。 | A >> 2 = 15, 也就是 1111 |
| >>> | 右移补零操作。左边的操作数的值由右操作数指定的位数向右移动，并转移值以零填充。 | A >>>2 = 15 也就是 0000 1111 |

**赋值运算符：**

Scala语言支持以下赋值运算符：

[查看示例](http://www.yiibai.com/scala/scala_assignment_operators.html" \o "赋值运算符" \t "_blank)

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **示例** |
| = | 简单的赋值操作符，分配值从右边的操作数左侧的操作数 | C = A + B 将分配 A + B 的值到 C |
| += | 加法和赋值运算符，它增加了右操作数左操作数和分配结果左操作数 | C += A 相当于 C = C + A |
| -= | 减和赋值运算符，它减去右操作数从左侧的操作数和分配结果左操作数 | C -= A 相当于 C = C - A |
| \*= | 乘法和赋值运算符，它乘以右边的操作数与左操作数和分配结果左操作数 | C \*= A 相当于 C = C \* A |
| /= | 除法和赋值运算符，它把左操作数与右操作数和分配结果左操作数 | C /= A 相当于 C = C / A |
| %= | 模量和赋值运算符，它需要使用两个操作数的模量和分配结果左操作数 | C %= A 相当于 C = C % A |
| <<= | 左移位并赋值运算符 | C <<= 2 等同于 C = C << 2 |
| >>= | 向右移位并赋值运算符 | C >>= 2 等同于 C = C >> 2 |
| &= | 按位与赋值运算符 | C &= 2 等同于C = C & 2 |
| ^= | 按位异或并赋值运算符 | C ^= 2 等同于 C = C ^ 2 |
| |= | 按位或并赋值运算符 | C |= 2 等同于 C = C | 2 |

**Scala的运算符优先级：**

运算符优先级决定术语的表达式分组。这会影响一个表达式是如何进行计算。某些运算符的优先级高于其他;例如，乘法运算符的优先级比所述加法运算符优先级更高：

例如X =7 + 3\* 2;这里，x 被赋值13，而不是20，因为运算符\* 优先级高于+，所以它首先做乘法3 \* 2，然后加上7。

这里，具有最高优先级的运算符在表的顶部，那些优先低级排在底部。在一个表达式，优先级高的运算符将首先计算。

|  |  |  |
| --- | --- | --- |
| **分类** | **运算符** | **关联** |
| Postfix | () [] | 从左到右 |
| Unary | ! ~ | 从右到左 |
| Multiplicative | \* / % | 从左到右 |
| Additive | + - | 从左到右 |
| Shift | >> >>> << | 从左到右 |
| Relational | > >= < <= | 从左到右 |
| Equality | == != | 从左到右 |
| Bitwise AND | & | 从左到右 |
| Bitwise XOR | ^ | 从左到右 |
| Bitwise OR | | | 从左到右 |
| Logical AND | && | 从左到右 |
| Logical OR | || | 从左到右 |
| Assignment | = += -= \*= /= %= >>= <<= &= ^= |= | 从右到左 |
| Comma | , | 从左到右 |

# 判断语句（if...else...）

下面是一个典型的决策中IF...ELSE结构的一般形式使用在大多数的编程语言中：

**if 语句:**

if 语句包含一个布尔表达式后跟一个或多个语句。

**语法：**

一个 if 语句的语法：

if(Boolean\_expression){

// Statements will execute if the Boolean expression is true

}

如果布尔表达式的值为true，那么if语句里面的代码模块将被执行。如果不是这样，第一组码if语句结束后（右大括号后）将被执行。

**示例：**

object Test {

def main(args: Array[String]) {

var x = 10;

if( x < 20 ){

println("This is if statement");

}

}

}

这将产生以下输出结果：

C:/>scalac Test.scala

C:/>scala Test

This is if statement

C:/>

**if...else语句:**

if语句可以跟着一个可选的else语句，当 else 块执行时，布尔表达式条件是假的。

**语法：**

if...else的语法是：

if(Boolean\_expression){

//Executes when the Boolean expression is true

}else{

//Executes when the Boolean expression is false

}

**示例：**

object Test {

def main(args: Array[String]) {

var x = 30;

if( x < 20 ){

println("This is if statement");

}else{

println("This is else statement");

}

}

}

这将产生以下结果:

C:/>scalac Test.scala

C:/>scala Test

This is else statement

C:/>

**if...else if...else语句:**

if语句可以跟着一个可选的else if ... else语句，这是非常有用的使用 if...else if如果测试各种条件声明。

当使用 if , else if , else 语句有几点要牢记。

* if可以有零或一个else，它必须跟在else if后面。
* 一个if 可以有零到多个else if，并且它们必须在else之前。
* 一旦一个 else if 匹配成功，剩余的else if或else不会被测试匹配。

**语法：**

if...else if...else的语法是：

if(Boolean\_expression 1){

//Executes when the Boolean expression 1 is true

}else if(Boolean\_expression 2){

//Executes when the Boolean expression 2 is true

}else if(Boolean\_expression 3){

//Executes when the Boolean expression 3 is true

}else {

//Executes when the none of the above condition is true.

}

**示例：**

object Test {

def main(args: Array[String]) {

var x = 30;

if( x == 10 ){

println("Value of X is 10");

}else if( x == 20 ){

println("Value of X is 20");

}else if( x == 30 ){

println("Value of X is 30");

}else{

println("This is else statement");

}

}

}

这将产生以下结果:

C:/>scalac Test.scala

C:/>scala Test

Value of X is 30

C:/>

**if ... else语句嵌套：**

它始终是合法的嵌套 if-else 语句，这意味着可以使用一个 if 或 else if 在另一个if 或 else if 语句中。

**语法：**

语法嵌套 if...else 如下：

if(Boolean\_expression 1){

//Executes when the Boolean expression 1 is true

if(Boolean\_expression 2){

//Executes when the Boolean expression 2 is true

}

}

可以嵌套else if...else在if语句中，反之也可以。

**示例：**

object Test {

def main(args: Array[String]) {

var x = 30;

var y = 10;

if( x == 30 ){

if( y == 10 ){

println("X = 30 and Y = 10");

}

}

}

}

这将产生以下结果:

C:/>scalac Test.scala

C:/>scala Test

X = 30 and Y = 10

C:/>

# 循环类型

可能有一种情况，当需要多次执行代码的几个块。在一般情况下，语句顺序执行：在一个函数的第一条语句，首先执行，然后是第二个等等。

编程语言提供了各种控制结构，允许更多复杂的执行路径。

循环语句可以执行语句多次或多组，下面是在大多数编程语言和循环语句一般如下：

Scala编程语言提供了以下循环类型的处理循环需求。点击以下链接查看其详细信息。

|  |  |
| --- | --- |
| **循环类型** | **描述** |
| [while循环](http://www.yiibai.com/scala/scala_while_loop.html" \o "while循环" \t "_blank) | 重复声明语句或一组，当给定的条件为真。它测试条件执行循环体前。 |
| [do...while循环](http://www.yiibai.com/scala/scala_do_while_loop.html" \o "do...while循环" \t "_blank) | 像一个while语句，不同之处在于它测试条件在循环体的结尾 |
| [for循环](http://www.yiibai.com/scala/scala_for_loop.html" \o "for循环" \t "_blank) | 执行语句多次序列并简写管理循环变量的代码。 |

**循环控制语句：**

循环控制语句改变其正常的顺序执行。当执行离开一个范围，在该范围内创建的所有对象自动被销毁。但是Scala不支持break或continue语句，想要像Java，但从Scala2.8版本开始，有一种方法可以打退出循环。点击以下链接查看详细信息。

|  |  |
| --- | --- |
| **控制语句** | **描述** |
| [break语句](http://www.yiibai.com/scala/scala_break_statement.html" \o "Scala break 语句" \t "_blank) | 终止循环语句并将执行立刻循环的下面语句。 |

**无限循环：**

一个循环变成无限循环，如果条件永远不会为假。如果使用Scala，while循环是实现无限循环的最佳方式，如下：

object Test {

def main(args: Array[String]) {

var a = 10;

// An infinite loop.

while( true ){

println( "Value of a: " + a );

}

}

}

如果将上面的代码执行，它会在无限循环可以通过按Ctrl+ C键终止。

# while循环

while 循环语句多次执行，只要给定的条件为真执行目标语句。

**语法:**

Scala while循环的语法是：

while(condition){

statement(s);

}

在这里，声明可以是单个语句或语句块。所述条件可以是任何表达式，真值是任何非零值。当条件为true，则循环迭代。当条件为faklse，则程序控制进到紧接在循环之后的行。

**流程图：**

在这里，while循环的关键点是循环可能不会永远运行。当条件测试结果为false，循环体将跳过while循环后的第一个语句执行。

**示例：**

object Test {

def main(args: Array[String]) {

// Local variable declaration:

var a = 10;

// while loop execution

while( a < 20 ){

println( "Value of a: " + a );

a = a + 1;

}

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

C:/>

# 函数

函数是一组一起执行任务的语句。可以将代码放到独立的功能。如何划分你的代码不同功能？在逻辑上，通常是让每个函数执行特定的任务。

Scala有函数和方法，我们术语说的方法和函数互换用微小的差别。Scala方法是其中有一个名字，签名，任选一些注释，有的字节码，其中如在Scala中函数是可被分配给一个变量的完整对象类的一部分。换句话说，函数，其被定义为某些对象的一个成员，被称为方法。

函数定义可以出现在在源文件的任何地方，Scala允许嵌套函数的定义，那就是其他函数定义的内部函数定义。需要注意的最重要的一点是，Scala的函数名称可以类似+, ++, ~, &,-, -- , , /, : 等字符。

**函数声明：**

Scala函数声明有如下形式：

def functionName ([list of parameters]) : [return type]

如果保留关闭等号和方法体的方法则为隐式声明，抽象的封闭类型是抽象本身。

**函数定义：**

Scala函数定义有如下形式：

def functionName ([list of parameters]) : [return type] = {

function body

return [expr]

}

在这里，返回类型可以是任何有效的scala数据类型，参数列表将是用逗号和参数，返回值类型列表分离变量是可选的。非常类似于Java，一个返回语句可以在函数表达式可用情况下返回一个值。以下是这将增加两个整数并返回的函数：

object add{

def addInt( a:Int, b:Int ) : Int = {

var sum:Int = 0

sum = a + b

return sum

}

}

函数，它不返回任何东西，可以返回这相当于在Java中void，并表示该函数不返回任何单元。Scala中不返回任何东西函数被称为过程。以下是语法

object Hello{

def printMe( ) : Unit = {

println("Hello, Scala!")

}

}

**调用函数：**

Scala提供了一些语法的变化来调用方法。以下是调用一个方法的标准方法：

functionName( list of parameters )

如果函数被对象的一个实例调用使用，那么使用类似于Java点标记如下：

[instance.]functionName( list of parameters )

下面是一个例子用来定义，然后调用函数：

object Test {

def main(args: Array[String]) {

println( "Returned Value : " + addInt(5,7) );

}

def addInt( a:Int, b:Int ) : Int = {

var sum:Int = 0

sum = a + b

return sum

}

}

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

Returned Value : 12

C:/>

Scala函数是Scala编程的核心，这就是为什么Scala被假定为一个函数式编程语言。以下是Scala程序员理解相关Scala函数的一些重要的概念。

## 函数按名称调用

通常情况下，函数的参数是传值参数;即，参数的值在它被传递给函数之前被确定。但是，如果我们需要编写一个接收参数不希望马上计算，直到调用函数内的表达式。对于这种情况，Scala提供按名称参数调用函数。

按名称调用机制传递一个代码块给被调用者并且每次被调用方传接入参数，代码块被执行，值被计算。

object Test {

def main(args: Array[String]) {

delayed(time());

}

def time() = {

println("Getting time in nano seconds")

System.nanoTime

}

def delayed( t: => Long ) = {

println("In delayed method")

println("Param: " + t)

t

}

}

在这里，我们声明delayed方法，它通过=>符号变量的名称和类型，需要一个按名称调用参数。当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

In delayed method

Getting time in nano seconds

Param: 81303808765843

Getting time in nano seconds

C:/>

这里，delayed打印的消息声明，该方法已被输入。接下来，delayed打印一个与其消息的值。最后delayed方法返回 t。

## 使用命名参数函数

在一个正常的函数调用，调用参数在调用函数的参数的顺序一一匹配。命名参数允许将参数以不同的顺序传递给的函数。语法很简单，每个参数前面有一个参数名和等号。下面是一个简单的例子来说明这一概念：

object Test {

def main(args: Array[String]) {

printInt(b=5, a=7);

}

def printInt( a:Int, b:Int ) = {

println("Value of a : " + a );

println("Value of b : " + b );

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Value of a : 7

Value of b : 5

C:/>

## 函数使用可变参数

Scala允许指出的最后一个参数的函数可以被重复。能够通过可变长度参数列表传递到函数。下面是一个简单的例子来说明这个概念。

object Test {

def main(args: Array[String]) {

printStrings("Hello", "Scala", "Python");

}

def printStrings( args:String\* ) = {

var i : Int = 0;

for( arg <- args ){

println("Arg value[" + i + "] = " + arg );

i = i + 1;

}

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Arg value[0] = Hello

Arg value[1] = Scala

Arg value[2] = Python

C:/>

在这里，printStrings函数，这被声明为类型为“String\*”里的args类型实际上是数组[字符串]。

## 递归函数

递归起着纯粹函数式编程很大的作用，Scala支持递归功能非常好。递归是指的函数可以反复调用自身。以下是递归，用于计算阶乘数的一个很好的例子：

object Test {

def main(args: Array[String]) {

for (i <- 1 to 10)

println( "Factorial of " + i + ": = " + factorial(i) )

}

def factorial(n: BigInt): BigInt = {

if (n <= 1)

1

else

n \* factorial(n - 1)

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Factorial of 1: = 1

Factorial of 2: = 2

Factorial of 3: = 6

Factorial of 4: = 24

Factorial of 5: = 120

Factorial of 6: = 720

Factorial of 7: = 5040

Factorial of 8: = 40320

Factorial of 9: = 362880

Factorial of 10: = 3628800

C:/>

## 默认参数值

Scala可以指定默认值函数的参数。对于这样的一个参数，可以任选地从一个函数调用，在这种情况下对应的参数将被填充使用默认参数值。下面是指定默认参数的一个例子：

object Test {

def main(args: Array[String]) {

println( "Returned Value : " + addInt() );

}

def addInt( a:Int=5, b:Int=7 ) : Int = {

var sum:Int = 0

sum = a + b

return sum

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Returned Value : 12

C:/>

如果指定一个参数，那么第一个参数将使用该参数传递，第二将采用默认值。

## 高阶函数

Scala允许高阶函数的定义。这些都是采取其他函数参数，或它的结果是一个功能的函数。例如在下面的代码，适用于apply()函数将另一个函数f和v值并应用函数f到v：

object Test {

def main(args: Array[String]) {

println( apply( layout, 10) )

}

def apply(f: Int => String, v: Int) = f(v)

def layout[A](x: A) = "[" + x.toString() + "]"

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

[10]

C:/>

## 嵌套函数

Scala允许在一个函数内部定义其他函数定义的函数，并可被局部函数调用。这里是一个阶乘计算器，在这里使用调用第二个函数，嵌套方式是传统技术的一个实现：

object Test {

def main(args: Array[String]) {

println( factorial(0) )

println( factorial(1) )

println( factorial(2) )

println( factorial(3) )

}

def factorial(i: Int): Int = {

def fact(i: Int, accumulator: Int): Int = {

if (i <= 1)

accumulator

else

fact(i - 1, i \* accumulator)

}

fact(i, 1)

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

1

1

2

6

C:/>

像在许多语言中的局部变量声明，嵌套方法只封闭方法内可见。如果试图调用 fact() 在factorial()之外，将会得到一个编译器错误。

## 匿名函数

Scala中提供相对轻便的语法定义匿名函数。在源代码中的匿名函数被调用函数文本，并在运行时，函数文本被实例化为调用的函数值的对象。

Scala支持一流的功能，这意味着可以表达功能文本语法功能，即，(x: Int) => x + 1，函数都可以通过对象，这是所谓的函数值来表示。下面的表达式创建一个后继函数的整数：

var inc = (x:Int) => x+1

变量inc现在可以使用以通常的方式的函数：

var x = inc(7)-1

另外，也可以用多个参数定义的函数如下：

var mul = (x: Int, y: Int) => x\*y

变量mul现在可以使用以通常的方式的函数：

println(mul(3, 4))

另外，也可以用无参数定义函数如下：

var userDir = () => { System.getProperty("user.dir") }

变量userDir现在可以使用以通常的方式的函数：

println( userDir )

## 部分应用函数

当调用的函数，可以说成是将参数给函数。如果通过所有预期的参数，已经完全把它应用。如果只发送了几个参数，会得到一个部分应用的函数。这可以绑定一些参数，留下其余稍后填补，提供了方便。下面是一个简单的例子来说明这一概念：

import java.util.Date

object Test {

def main(args: Array[String]) {

val date = new Date

log(date, "message1" )

Thread.sleep(1000)

log(date, "message2" )

Thread.sleep(1000)

log(date, "message3" )

}

def log(date: Date, message: String) = {

println(date + "----" + message)

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Mon Dec 02 12:52:41 CST 2013----message1

Mon Dec 02 12:52:41 CST 2013----message2

Mon Dec 02 12:52:41 CST 2013----message3

C:/>

在这里，log()方法有两个参数：date和message。想调用这个方法多次，对于date使用不同的值，但message的值相同。我们可以消除部分采用的参数传递给log()方法传递的date给每个调用会有干扰。要做到这一点，我们首先绑定一个值到date参数，并把下划线在其位置第二个参数绑定。其结果是，我们已经存储在一个变量的部分应用函数。现在可以与未结合的mesage参数调用这个新方法如下：

import java.util.Date

object Test {

def main(args: Array[String]) {

val date = new Date

val logWithDateBound = log(date, \_ : String)

logWithDateBound("message1" )

Thread.sleep(1000)

logWithDateBound("message2" )

Thread.sleep(1000)

logWithDateBound("message3" )

}

def log(date: Date, message: String) = {

println(date + "----" + message)

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Mon Dec 02 12:53:56 CST 2013----message1

Mon Dec 02 12:53:56 CST 2013----message2

Mon Dec 02 12:53:56 CST 2013----message3

C:/>

## 柯里函数

柯里转换函数接受多个参数成一条链的函数，每次取一个参数。柯里函数是具有多个参数列表定义，如下：

def strcat(s1: String)(s2: String) = s1 + s2

另外，还可以使用以下语法定义柯里函数：

def strcat(s1: String) = (s2: String) => s1 + s2

以下是语法来调用一个柯里函数：

strcat("foo")("bar")

可以根据柯里函数需求定义两个以上的参数。让我们以一个完整的例子来说明柯里的概念：

object Test {

def main(args: Array[String]) {

val str1:String = "Hello, "

val str2:String = "Scala!"

println( "str1 + str2 = " + strcat(str1)(str2) )

}

def strcat(s1: String)(s2: String) = {

s1 + s2

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

str1 + str2 = Hello, Scala!

C:/>

# 闭包

闭包是函数，它的返回值取决于此函数之外声明一个或多个变量的值。考虑下面的一块使用匿名函数的代码：

val multiplier = (i:Int) => i \* 10

在这里，在函数体中使用的唯一变量， i \* 0, 为i，其被定义为一个参数的函数。现在，让我们来看另一块代码：

val multiplier = (i:Int) => i \* factor

有两个自由变量的乘数：i和factor。其中一个i是一个正式函数参数。因此，它被绑定到一个新的值在乘数每次调用。然而，factor不是一个正式的参数，那么这是什么？让我们增加一行代码：

var factor = 3

val multiplier = (i:Int) => i \* factor

现在，factor具有参考变量在函数之外，但为封闭范围的变量。让我们试试下面的例子：

object Test {

def main(args: Array[String]) {

println( "muliplier(1) value = " + multiplier(1) )

println( "muliplier(2) value = " + multiplier(2) )

}

var factor = 3

val multiplier = (i:Int) => i \* factor

}

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

muliplier(1) value = 3

muliplier(2) value = 6

C:/>

上面的函数引用factor并读取每个时间的当前值。如果函数没有外部引用，那么它就是封闭了自己。无需外部环境是必需的。

# 字符串

考虑下面这个简单的例子，我们给一个字符串类型的 val 变量：

object Test {

val greeting: String = "Hello, world!"

def main(args: Array[String]) {

println( greeting )

}

}

在这里，上面的值类型是从Java的java.lang.String借来的，因为Scala的字符串也是Java字符串。这一点是非常好的，要注意：每 一个Java类可用在Scala中。因此，Scala没有String类并直接利用Java字符串String类。因此，这一章已被写入保持Java String的基础。

在Scala中与在Java中一样，一个字符串是不可变的对象，也就是，这是不能被修改的对象。另一方面，对象是可以修改的，如数组对象，被称为可变对 象。由于字符串是非常有用的对象，在本节的其余部分，我们目前最重要了解类java.lang.String的方法定义。

**创建字符串：**

最直接的方法来创建一个字符串的编写方法是：

var greeting = "Hello world!";

or

var greeting:String = "Hello world!";

每当遇到一个字符串在代码中，编译器创建String对象及其值，在这个示例中使用：“Hello world!”，但如果你喜欢，可以给字符串，因为这里我已经在声明中交替显示。

object Test {

val greeting: String = "Hello, world!"

def main(args: Array[String]) {

println( greeting )

}

}

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

Hello, world!

C:/>

正如前面提到的，String类是不可变的，所以一旦它被创建,String 对象不能改变。如果需要做大量修改字符的字符串，那么应该使用在Scala提供字符串构建器类（ [String Builder](http://www.yiibai.com/scala/scala_string_builder.html) ）。

**字符串长度:**

用于获得关于对象的信息的方法是已知的存取方法。可以使用字符串使用一个存取方法是length()方法，它返回包含在字符串对象中的字符数。

下面两行代码被执行之后，len 等于17：

object Test {

def main(args: Array[String]) {

var palindrome = "Dot saw I was Tod";

var len = palindrome.length();

println( "String Length is : " + len );

}

}

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

String Length is : 17

C:/>

**连接字符串：**

String类包括用于连接两个字符串的方法：

string1.concat(string2);

这返回一个string1将string2添加到它的末尾的一个新字符串。也可以使用concat()方法用字符串，如：

"My name is ".concat("Zara");

字符串比较常用连接在一起，使用+运算符，如：

"Hello," + " world" + "!"

这将产生：

"Hello, world!"

让我们来看看下面的例子：

object Test {

def main(args: Array[String]) {

var str1 = "Dot saw I was ";

var str2 = "Tod";

println("Dot " + str1 + str2);

}

}

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

Dot Dot saw I was Tod

C:/>

**创建格式化字符串：**

已经有两个方法：printf()和format()方法使用格式化数字打印输出。

String类有一个等价类的方法，format()，它返回一个String对象，而不是一个PrintStream对象。让我们看看下面的例子，这是使用printf()方法：

object Test {

def main(args: Array[String]) {

var floatVar = 12.456

var intVar = 2000

var stringVar = "Hello, Scala!"

var fs = printf("The value of the float variable is " +

"%f, while the value of the integer " +

"variable is %d, and the string " +

"is %s", floatVar, intVar, stringVar)

println(fs)

}

}让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

The value of the float variable is 12.456000, while the

value of the integer var iable is 2000, and the

string is Hello, Scala!()

C:/>

**字符串的方法：**

以下是由java.lang.String类中定义，并且可以直接在Scala程序中使用方法列表：

|  |  |
| --- | --- |
| **SN** | **方法及描述** |
| 1 | **char charAt(int index)**  返回指定索引处的字符 |
| 2 | **int compareTo(Object o)** 该字符串与另一个对象比较 |
| 3 | **int compareTo(String anotherString)** 按字典顺序比较两个字符串 |
| 4 | **int compareToIgnoreCase(String str)**  比较两个字符串字典顺序，忽略大小写差异 |
| 5 | **String concat(String str)** 将指定字符串添加这个字符串的结尾处 |
| 6 | **boolean contentEquals(StringBuffer sb)** 返回true当且仅当此String表示的字符与指定的StringBuffer相同的序列 |
| 7 | **static String copyValueOf(char[] data)** 返回表示所指定的数组中的字符序列的字符串 |
| 8 | **static String copyValueOf(char[] data, int offset, int count)** 返回表示所指定的数组中的字符序列的字符串。 |
| 9 | **boolean endsWith(String suffix)**  测试此字符串是否以指定的后缀结束。 |
| 10 | **boolean equals(Object anObject)** 比较该字符串和指定的对象。 |
| 11 | **boolean equalsIgnoreCase(String anotherString)** 比较该字符串到另一个字符串，不考虑大小写。 |
| 12 | **byte getBytes()**  将此String解码使用平台的默认字符集的字节序列，并将结果存储到一个新的字节数组。 |
| 13 | **byte[] getBytes(String charsetName** 将此String解码使用指定的字符集的字节序列，并将结果存储到一个新的字节数组。 |
| 14 | **void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** 复制字符从这个字符串到目标字符数组。 |
| 15 | **int hashCode()** 返回此字符串的哈希码。 |
| 16 | **int indexOf(int ch)** 返回此字符串指定字符第一次出现处的索引。 |
| 17 | **int indexOf(int ch, int fromIndex)**  返回此字符串指定的字符，开始搜索指定索引处的第一次出现处的索引。 |
| 18 | **int indexOf(String str)** 返回此字符串的指定子第一次出现处的索引。 |
| 19 | **int indexOf(String str, int fromIndex)** 返回此字符串中第一次出现的指定子字符串，开始在指定的索引处的索引。 |
| 20 | **String intern()** 返回字符串对象的规范表示。 |
| 21 | **int lastIndexOf(int ch)**  返回此字符串指定字符最后一次出现处的索引。 |
| 22 | **int lastIndexOf(int ch, int fromIndex)**  返回此字符串指定字符最后一次出现处的索引，搜索开始向后指定索引处。 |
| 23 | **int lastIndexOf(String str)** 返回此字符串指定子的最右边出现处的索引。 |
| 24 | **int lastIndexOf(String str, int fromIndex)**  返回此字符串的指定子最后一次出现处的索引，搜索开始向后指定索引处。 |
| 25 | **int length()**  返回此字符串的长度。 |
| 26 | **boolean matches(String regex)** 判断此字符串是否与给正则表达式匹配。 |
| 27 | **boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)** 测试两个字符串区域是否相等。 |
| 28 | **boolean regionMatches(int toffset, String other, int ooffset, int len)** 测试两个字符串区域是否相等。 |
| 29 | **String replace(char oldChar, char newChar)** 返回通过用newChar更换oldChar中出现的所有该字符串产生一个新的字符串。 |
| 30 | **String replaceAll(String regex, String replacement** 替换此字符串匹配给定的正则表达式以给定替换的每个子字符串。 |
| 31 | **String replaceFirst(String regex, String replacement)**  替换此字符串匹配给定的正则表达式以给定替换的第一子字符串。 |
| 32 | **String[] split(String regex)**  来此字符串围绕拆分给定的正则表达式的匹配。 |
| 33 | **String[] split(String regex, int limit)**  围绕拆分给定的正则表达式的匹配来此字符串。 |
| 34 | **boolean startsWith(String prefix)** 测试此字符串是否以指定的前缀开始。 |
| 35 | **boolean startsWith(String prefix, int toffset)** 测试这个字符串开头是否以指定索引指定的前缀开始。 |
| 36 | **CharSequence subSequence(int beginIndex, int endIndex)** 返回一个新的字符序列，它是此序列的子序列。 |
| 37 | **String substring(int beginIndex)** 返回一个新字符串，它是此字符串的一个子字符串。 |
| 38 | **String substring(int beginIndex, int endIndex)** 返回一个新字符串，它是此字符串的一个子字符串。 |
| 39 | **char[] toCharArray()**  这个字符串到一个新的字符数组转换。 |
| 40 | **String toLowerCase()** 将所有在这个字符串的字符使用默认语言环境的规则转换为小写。 |
| 41 | **String toLowerCase(Locale locale)** 将所有在此字符串中的字符使用给定Locale的规则转换为小写。 |
| 42 | **String toString()** 这个对象（这已经是一个字符串！）本身被返回。 |
| 43 | **String toUpperCase()**  将所有的字符在这个字符串使用默认语言环境的规则转换为大写。 |
| 44 | **String toUpperCase(Locale locale)**  将所有的字符在这个字符串使用给定的Locale规则转换为大写。 |
| 45 | **String trim()**  返回字符串的副本，以开头和结尾的空白忽略。 |
| 46 | **static String valueOf(primitive data type x)**  返回传递的数据类型参数的字符串表示。 |

# 数组

Scala中提供了一种数据结构-数组，其中存储相同类型的元素的固定大小的连续集合。数组用于存储数据的集合，但它往往是更加有用认为数组作为相同类型的变量的集合。

取替声明单个变量，如number0, number1, ..., 和number99，声明一个数组变量，如号码和使用numbers[0]，numbers[1]，...，numbers[99]表示单个变量。本教程 介绍了如何声明数组变量，创建数组和使用索引的过程变量数组。数组的第一个元素的索引是数字0和最后一个元素的索引为元素的总数减去1。

## 声明数组变量：

要使用的程序的数组，必须声明一个变量来引用数组，必须指定数组变量可以引用的类型。下面是语法声明数组变量：

var z:Array[String] = new Array[String](3)

or

var z = new Array[String](3)

在这里，z被声明为字符串数组，最多可容纳三个元素。可以将值分配给独立的元素或可以访问单个元素，这是可以做到通过使用类似于以下命令：

z(0) = "Zara";

z(1) = "Nuha";

z(4/2) = "Ayan"

在这里，最后一个例子表明，在一般的索引可以是产生一个全数字的表达式。定义数组还有另一种方式：

var z = Array("Zara", "Nuha", "Ayan")

下图展示了数组myList。在这里，myList中拥有10个double值,索引是从0到9。

## 处理数组：

当要处理数组元素，我们经常使用循环，因为所有的数组中的元素具有相同的类型，并且数组的大小是已知的。这里是展示如何创建，初始化和处理数组的完整的例子：

object Test {

def main(args: Array[String]) {

var myList = Array(1.9, 2.9, 3.4, 3.5)

// Print all the array elements

for ( x <- myList ) {

println( x )

}

// Summing all elements

var total = 0.0;

for ( i <- 0 to (myList.length - 1)) {

total += myList(i);

}

println("Total is " + total);

// Finding the largest element

var max = myList(0);

for ( i <- 1 to (myList.length - 1) ) {

if (myList(i) > max) max = myList(i);

}

println("Max is " + max);

}

}让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

1.9

2.9

3.4

3.5

Total is 11.7

Max is 3.5

C:/>

## 多维数组：

有很多情况下，需要定义和使用多维数组（即数组的元素数组）。例如，矩阵和表格结构的实例可以实现为二维数组。

Scala不直接支持多维数组，并提供各种方法来处理任何尺寸数组。以下是定义的二维数组的实例：

var myMatrix = ofDim[Int](3,3)

这是一个具有每个都是整数，它有三个元素数组3元素的数组。下面的代码展示了如何处理多维数组：

import Array.\_

object Test {

def main(args: Array[String]) {

var myMatrix = ofDim[Int](3,3) //创建二维数组

// build a matrix

for (i <- 0 to 2) {

for ( j <- 0 to 2) {

myMatrix(i)(j) = j;

}

}

// Print two dimensional array

for (i <- 0 to 2) {

for ( j <- 0 to 2) {

print(" " + myMatrix(i)(j));

}

println();

}

}

}

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

0 1 2

0 1 2

0 1 2

C:/>

## 联接数组：

以下是使用concat()方法来连接两个数组的例子。可以通过多个阵列作为参数传递给concat()方法。

import Array.\_

object Test {

def main(args: Array[String]) {

var myList1 = Array(1.9, 2.9, 3.4, 3.5)

var myList2 = Array(8.9, 7.9, 0.4, 1.5)

var myList3 = concat( myList1, myList2)

// Print all the array elements

for ( x <- myList3 ) {

println( x )

}

}

}

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

1.9

2.9

3.4

3.5

8.9

7.9

0.4

1.5

C:/>

## 创建具有范围数组：

下面是示例，这使得使用range() 方法来产生包含在给定的范围内增加整数序列的数组。可以使用最后一个参数创建序列; 如果不使用最后一个参数，然后一步将被假定为1。

import Array.\_

object Test {

def main(args: Array[String]) {

var myList1 = range(10, 20, 2) //步长为2

var myList2 = range(10,20) //默认步长为1

// Print all the array elements

for ( x <- myList1 ) {

print( " " + x )

}

println()

for ( x <- myList2 ) {

print( " " + x )

}

}

}

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

10 12 14 16 18

10 11 12 13 14 15 16 17 18 19

C:/>

## Scala中数组方法：

以下是重要的方法，可以同时使用数组。如上所示，则必须使用任何提及的方法之前，要导入Array.\_包。有关可用方法的完整列表，请Scala中的官方文件。

|  |  |
| --- | --- |
| **SN** | **方法及描述** |
| 1 | **def apply( x: T, xs: T\* ): Array[T]** 创建T对象，其中T可以是Unit, Double, Float, Long, Int, Char, Short, Byte, Boolean数组。 |
| 2 | **def concat[T]( xss: Array[T]\* ): Array[T]** 连接所有阵列成一个数组。 |
| 3 | **def copy( src: AnyRef, srcPos: Int, dest: AnyRef, destPos: Int, length: Int ): Unit** 复制一个数组到另一个。相当于Java的System.arraycopy(src, srcPos, dest, destPos, length). |
| 4 | **def empty[T]: Array[T]** 返回长度为0的数组 |
| 5 | **def iterate[T]( start: T, len: Int )( f: (T) => T ): Array[T]** 返回一个包含一个函数的重复应用到初始值的数组。 |
| 6 | **def fill[T]( n: Int )(elem: => T): Array[T]** 返回包含某些元素的计算的结果的次数的数组。 |
| 7 | **def fill[T]( n1: Int, n2: Int )( elem: => T ): Array[Array[T]]** 返回一个二维数组，其中包含某些元素的计算的结果的次数。 |
| 8 | **def iterate[T]( start: T, len: Int)( f: (T) => T ): Array[T]** 返回一个包含一个函数的重复应用到初始值的数组。 |
| 9 | **def ofDim[T]( n1: Int ): Array[T]** 创建数组给出的尺寸。 |
| 10 | **def ofDim[T]( n1: Int, n2: Int ): Array[Array[T]]** 创建了一个2维数组 |
| 11 | **def ofDim[T]( n1: Int, n2: Int, n3: Int ): Array[Array[Array[T]]]** 创建3维数组 |
| 12 | **def range( start: Int, end: Int, step: Int ): Array[Int]** 返回包含一些整数间隔等间隔值的数组。 |
| 13 | **def range( start: Int, end: Int ): Array[Int]** 返回包含的范围内增加整数序列的数组。 |
| 14 | **def tabulate[T]( n: Int )(f: (Int)=> T): Array[T]** 返回包含一个给定的函数的值超过从0开始的范围内的整数值的数组。 |
| 15 | **def tabulate[T]( n1: Int, n2: Int )( f: (Int, Int ) => T): Array[Array[T]]** 返回一个包含给定函数的值超过整数值从0开始范围的二维数组。 |

# 集合

Scala有一组丰富的集合库。集合是对事物的容器。这些容器可被测序，线性集像List, Tuple, Option, Map等集合的项目可具有元素的任意数量或有界到零个或一个元素(例如，Option)。

集合可能是严格或懒惰。懒集合有可能不消耗内存，直到他们被访问，就像范围元素。此外，集合可以是可变的(引用的内容可以更改)或不变(一个引用的东西指从未改变)。需要注意的是不可变的集合可能包含可变项。

对于一些问题，可变集合更好地工作，并为不可变集合更好地工作。如果有疑问，最好是先从一个不变的集合，如果需要可变进行更改。

本章给出最常用的集合类型对这些集合的信息和使用最频繁的操作。

|  |  |
| --- | --- |
| **SN** | **集合使用说明** |
| 1 | [Scala Lists](http://www.yiibai.com/scala/scala_lists.html" \t "_blank) Scala的List[T]是T类型的链表 |
| 2 | [Scala Sets](http://www.yiibai.com/scala/scala_sets.html" \t "_blank) 集是相同类型的配对的不同元素的集合。 |
| 3 | [Scala Maps](http://www.yiibai.com/scala/scala_maps.html" \t "_blank) 映射是键/值对的集合。任何值可以根据它的键进行检索。 |
| 4 | [Scala Tuples](http://www.yiibai.com/scala/scala_tuples.html" \t "_blank) 不像数组或列表，元组可以容纳不同类型的对象。 |
| 5 | [Scala Options](http://www.yiibai.com/scala/scala_options.html" \t "_blank) Option[T] 提供了一种容器，用于给定类型的零个或一个元素。 |
| 6 | [Scala Iterators](http://www.yiibai.com/scala/scala_iterators.html" \t "_blank) 迭代不是集合，而是一种由一个访问的集合之一的元素。 |

## 示例:

下面的代码片段是一个简单的例子来定义所有上述集合类型：

// Define List of integers.

val x = List(1,2,3,4)

// Define a set.

var x = Set(1,3,5,7)

// Define a map.

val x = Map("one" -> 1, "two" -> 2, "three" -> 3)

// Create a tuple of two elements.

val x = (10, "Scala")

// Define an option

val x:Option[Int] = Some(5)

## 1、Lists

Scala的List[T]是T类型的链表

Scala中列表是非常类似于数组，这意味着，一个列表的所有元素都具有相同的类型，但有两个重要的区别。首先，列表是不可变的，这意味着一个列表的元素可以不被分配来改变。第二，列表表示一个链表，而数组平坦的。

具有T类型的元素的列表的类型被写为List[T]。例如，这里有各种数据类型定义的一些列表：

// List of Strings

val fruit: List[String] = List("apples", "oranges", "pears")

// List of Integers

val nums: List[Int] = List(1, 2, 3, 4)

// Empty List.

val empty: List[Nothing] = List()

// Two dimensional list

val dim: List[List[Int]] =

List(

List(1, 0, 0),

List(0, 1, 0),

List(0, 0, 1)

)

所有的列表可以使用两种基本的构建模块来定义，一个无尾Nil和::，这有明显的缺点。Nil也代表了空列表。所有上述列表可以定义如下：

// List of Strings

val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))

// List of Integers

val nums = 1 :: (2 :: (3 :: (4 :: Nil)))

// Empty List.

val empty = Nil

// Two dimensional list

val dim = (1 :: (0 :: (0 :: Nil))) ::

(0 :: (1 :: (0 :: Nil))) ::

(0 :: (0 :: (1 :: Nil))) :: Nil

### 列表的基本操作：

上列出了所有的操作都可以体现在以下三个方法来讲：

|  |  |
| --- | --- |
| **方法** | **描述** |
| head | 此方法返回的列表中的第一个元素。 |
| tail | 此方法返回一个由除了第一个元素外的所有元素的列表。 |
| isEmpty | 如果列表为空，此方法返回true，否则为false。 |

以下是上述方法中的例子显示用法：

object Test {

def main(args: Array[String]) {

val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))

val nums = Nil

println( "Head of fruit : " + fruit.head )

println( "Tail of fruit : " + fruit.tail )

println( "Check if fruit is empty : " + fruit.isEmpty )

println( "Check if nums is empty : " + nums.isEmpty )

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Head of fruit : apples

Tail of fruit : List(oranges, pears)

Check if fruit is empty : false

Check if nums is empty : true

C:/>

### 串联列表：

可以使用:::运算符或列表List.:::()方法或List.concat()方法来添加两个或多个列表。下面是一个例子：

object Test {

def main(args: Array[String]) {

val fruit1 = "apples" :: ("oranges" :: ("pears" :: Nil))

val fruit2 = "mangoes" :: ("banana" :: Nil)

// use two or more lists with ::: operator

var fruit = fruit1 ::: fruit2

println( "fruit1 ::: fruit2 : " + fruit )

// use two lists with Set.:::() method

fruit = fruit1.:::(fruit2)

println( "fruit1.:::(fruit2) : " + fruit )

// pass two or more lists as arguments

fruit = List.concat(fruit1, fruit2)

println( "List.concat(fruit1, fruit2) : " + fruit )

}

}

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

fruit1 ::: fruit2 : List(apples, oranges, pears, mangoes, banana)

fruit1.:::(fruit2) : List(mangoes, banana, apples, oranges, pears)

List.concat(fruit1, fruit2) : List(apples, oranges, pears, mangoes, banana)

C:/>

### 创建统一列表：

可以使用List.fill()方法创建，包括相同的元素如下的零个或更多个拷贝的列表：

object Test {

def main(args: Array[String]) {

val fruit = List.fill(3)("apples") // Repeats apples three times.

println( "fruit : " + fruit )

val num = List.fill(10)(2) // Repeats 2, 10 times.

println( "num : " + num )

}

}让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

fruit : List(apples, apples, apples)

num : List(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)

C:/>

### 制成表格一个功能：

可以使用一个函数连同List.tabulate()方法制表列表之前的列表中的所有元素以应用。它的参数是一样List.fill：第一个参数列表给出的列表的尺寸大小，而第二描述列表的元素。唯一的区别在于，代替的元素被固定，它们是从一个函数计算：

object Test {

def main(args: Array[String]) {

// Creates 5 elements using the given function.

val squares = List.tabulate(6)(n => n \* n)

println( "squares : " + squares )

//

val mul = List.tabulate( 4,5 )( \_ \* \_ )

println( "mul : " + mul )

}

}

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

squares : List(0, 1, 4, 9, 16, 25)

mul : List(List(0, 0, 0, 0, 0), List(0, 1, 2, 3, 4),

List(0, 2, 4, 6, 8), List(0, 3, 6, 9, 12))

C:/>

### 反向列表顺序：

可以使用List.reverse方法来扭转列表中的所有元素。以下为例子来说明的用法：

object Test {

def main(args: Array[String]) {

val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))

println( "Before reverse fruit : " + fruit )

println( "After reverse fruit : " + fruit.reverse )

}

}让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

Before reverse fruit : List(apples, oranges, pears)

After reverse fruit : List(pears, oranges, apples)

C:/>

### Scala列表方法：

以下是重要的方法，可以在使用列表时。有关可用方法的完整列表，请Scala的官方文件。

|  |  |
| --- | --- |
| **SN** | **方法及描述** |
| 1 | **def +(elem: A): List[A]** 前置一个元素列表 |
| 2 | **def ::(x: A): List[A]** 在这个列表的开头添加的元素。 |
| 3 | **def :::(prefix: List[A]): List[A]** 增加了一个给定列表中该列表前面的元素。 |
| 4 | **def ::(x: A): List[A]** 增加了一个元素x在列表的开头 |
| 5 | **def addString(b: StringBuilder): StringBuilder** 追加列表的一个字符串生成器的所有元素。 |
| 6 | **def addString(b: StringBuilder, sep: String): StringBuilder** 追加列表的使用分隔字符串一个字符串生成器的所有元素。 |
| 7 | **def apply(n: Int): A** 选择通过其在列表中索引的元素 |
| 8 | **def contains(elem: Any): Boolean** 测试该列表中是否包含一个给定值作为元素。 |
| 9 | **def copyToArray(xs: Array[A], start: Int, len: Int): Unit** 列表的副本元件阵列。填充给定的数组xs与此列表中最多len个元素，在位置开始。 |
| 10 | **def distinct: List[A]** 建立从列表中没有任何重复的元素的新列表。 |
| 11 | **def drop(n: Int): List[A]** 返回除了第n个的所有元素。 |
| 12 | **def dropRight(n: Int): List[A]** 返回除了最后的n个的元素 |
| 13 | **def dropWhile(p: (A) => Boolean): List[A]** 丢弃满足谓词的元素最长前缀。 |
| 14 | **def endsWith[B](that: Seq[B]): Boolean** 测试列表是否使用给定序列结束。 |
| 15 | **def equals(that: Any): Boolean** equals方法的任意序列。比较该序列到某些其他对象。 |
| 16 | **def exists(p: (A) => Boolean): Boolean** 测试谓词是否持有一些列表的元素。 |
| 17 | **def filter(p: (A) => Boolean): List[A]** 返回列表满足谓词的所有元素。 |
| 18 | **def forall(p: (A) => Boolean): Boolean** 测试谓词是否持有该列表中的所有元素。 |
| 19 | **def foreach(f: (A) => Unit): Unit** 应用一个函数f以列表的所有元素。 |
| 20 | **def head: A** 选择列表的第一个元素 |
| 21 | **def indexOf(elem: A, from: Int): Int** 经过或在某些起始索引查找列表中的一些值第一次出现的索引。 |
| 22 | **def init: List[A]** 返回除了最后的所有元素 |
| 23 | **def intersect(that: Seq[A]): List[A]** 计算列表和另一序列之间的多重集交集。 |
| 24 | **def isEmpty: Boolean** 测试列表是否为空 |
| 25 | **def iterator: Iterator[A]** 创建一个新的迭代器中包含的可迭代对象中的所有元素 |
| 26 | **def last: A** 返回最后一个元素 |
| 27 | **def lastIndexOf(elem: A, end: Int): Int** 之前或在一个给定的最终指数查找的列表中的一些值最后一次出现的索引 |
| 28 | **def length: Int** 返回列表的长度 |
| 29 | **def map[B](f: (A) => B): List[B]** 通过应用函数以g这个列表中的所有元素构建一个新的集合 |
| 30 | **def max: A** 查找最大的元素 |
| 31 | **def min: A** 查找最小元素 |
| 32 | **def mkString: String** 显示列表的字符串中的所有元素 |
| 33 | **def mkString(sep: String): String** 显示的列表中的字符串中使用分隔串的所有元素 |
| 34 | **def reverse: List[A]** 返回新列表，在相反的顺序元素 |
| 35 | **def sorted[B >: A]: List[A]** 根据排序对列表进行排序 |
| 36 | **def startsWith[B](that: Seq[B], offset: Int): Boolean** 测试该列表中是否包含给定的索引处的给定的序列 |
| 37 | **def sum: A** 概括这个集合的元素 |
| 38 | **def tail: List[A]** 返回除了第一的所有元素 |
| 39 | **def take(n: Int): List[A]** 返回前n个元素 |
| 40 | **def takeRight(n: Int): List[A]** 返回最后n个元素 |
| 41 | **def toArray: Array[A]** 列表以一个数组变换 |
| 42 | **def toBuffer[B >: A]: Buffer[B]** 列表以一个可变缓冲器转换 |
| 43 | **def toMap[T, U]: Map[T, U]** 此列表的映射转换 |
| 44 | **def toSeq: Seq[A]** 列表的序列转换 |
| 45 | **def toSet[B >: A]: Set[B]** 列表到集合变换 |
| 46 | **def toString(): String** 列表转换为字符串 |

## 2、Sets

集是相同类型的配对的不同元素的集合。

Scala集合为相同类型的配对的不同元素的集合。换句话说，集合是不包含重复元素的集合。有两种集合，不可改变的和可变的。可变和不可变的对象之间的区别在于，当一个对象是不可变的，对象本身不能被改变。

默认情况下，Scala中使用不可变的集。如果想使用可变集，必须明确地导入scala.collection.mutable.Set类。如果想在同一 个同时使用可变和不可变的集合，那么可以继续参考不变的集合，但可以参考可变设为mutable.Set。以下是声明不变集合示例：

// Empty set of integer type

var s : Set[Int] = Set()

// Set of integer type

var s : Set[Int] = Set(1,3,5,7)

or

var s = Set(1,3,5,7)

在定义空集，类型注释是必要的，因为系统需要指定一个具体的类型变量。

### 集合基本操作：

集合所有操作可以体现在以下三个方法：

|  |  |
| --- | --- |
| **方法** | **描述** |
| head | 此方法返回集合的第一个元素。 |
| tail | 该方法返回集合由除第一个以外的所有元素。 |
| isEmpty | 如果设置为空，此方法返回true，否则为false。 |

以下是上述方法中的例子显示的用法：

object Test {

def main(args: Array[String]) {

val fruit = Set("apples", "oranges", "pears")

val nums: Set[Int] = Set()

println( "Head of fruit : " + fruit.head )

println( "Tail of fruit : " + fruit.tail )

println( "Check if fruit is empty : " + fruit.isEmpty )

println( "Check if nums is empty : " + nums.isEmpty )

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Head of fruit : apples

Tail of fruit : Set(oranges, pears)

Check if fruit is empty : false

Check if nums is empty : true

C:/>

### 串联集合：

可以使用++运算符或集。++()方法来连接两个或多个集，但同时增加了集它会删除重复的元素。以下是这个例子来连接两个集合：

object Test {

def main(args: Array[String]) {

val fruit1 = Set("apples", "oranges", "pears")

val fruit2 = Set("mangoes", "banana")

// use two or more sets with ++ as operator

var fruit = fruit1 ++ fruit2

println( "fruit1 ++ fruit2 : " + fruit )

// use two sets with ++ as method

fruit = fruit1.++(fruit2)

println( "fruit1.++(fruit2) : " + fruit )

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

fruit1 ++ fruit2 : Set(banana, apples, mangoes, pears, oranges)

fruit1.++(fruit2) : Set(banana, apples, mangoes, pears, oranges)

C:/>

### 查找集合中最大，最小的元素：

可以使用Set.min方法找出最小元素，Set.max方法找出一组可用最大元素。以下为例子来说明的用法：

object Test {

def main(args: Array[String]) {

val num = Set(5,6,9,20,30,45)

// find min and max of the elements

println( "Min element in Set(5,6,9,20,30,45) : " + num.min )

println( "Max element in Set(5,6,9,20,30,45) : " + num.max )

}

}

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

Min element in Set(5,6,9,20,30,45) : 5

Max element in Set(5,6,9,20,30,45) : 45

C:/>

### 查找集合的共同值：

可以使用Set.&方法或Set.intersect方法找出两个集合之间的共同值。以下的例子来说明的用法：

object Test {

def main(args: Array[String]) {

val num1 = Set(5,6,9,20,30,45)

val num2 = Set(50,60,9,20,35,55)

// find common elements between two sets

println( "num1.&(num2) : " + num1.&(num2) )

println( "num1.intersect(num2) : " + num1.intersect(num2) )

}

}

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

num1.&(num2) : Set(20, 9)

num1.intersect(num2) : Set(20, 9)

C:/>

### Scala集合方法：

以下是可以同时使用集合的重要方法。有关可用方法的完整列表，请Scala官方文档。

|  |  |
| --- | --- |
| **SN** | **方法及描述** |
| 1 | **def +(elem: A): Set[A]** 创建一组新的具有附加元件，除非该元件已经存在 |
| 2 | **def -(elem: A): Set[A]** 创建一个新的从这个集合中删除一个给定的元素 |
| 3 | **def contains(elem: A): Boolean** 如果elem包含在这个集合返回true，否则为false。 |
| 4 | **def &(that: Set[A]): Set[A]** 返回新的集合组成在这个集合，并在给定的集合，所有的元素。 |
| 5 | **def &~(that: Set[A]): Set[A]** 返回此集合和另一个集合的差异 |
| 6 | **def +(elem1: A, elem2: A, elems: A\*): Set[A]** 创建一个新的不可变的集合与来自传递集合额外的元素 |
| 7 | **def ++(elems: A): Set[A]** 连接此不可变的集合使用另一个集合到这个不可变的集合的元素。 |
| 8 | **def -(elem1: A, elem2: A, elems: A\*): Set[A]** 返回包含当前不可变的集合，除了每一个给定参数的元素之一，较少出现的所有元素的不可变的集合。 |
| 9 | **def addString(b: StringBuilder): StringBuilder** 这追加不可变的集到一个字符串生成器的所有元素。 |
| 10 | **def addString(b: StringBuilder, sep: String): StringBuilder** 这追加不可变的集合使用分隔字符串一个字符串生成器的所有元素。 |
| 11 | **def apply(elem: A)** 测试如果一些元素被包含在这个集合。 |
| 12 | **def count(p: (A) => Boolean): Int** 计算在不可变的集合满足谓词的元素数。 |
| 13 | **def copyToArray(xs: Array[A], start: Int, len: Int): Unit** 这种不可变的集合到一个数组的副本元素。 |
| 14 | **def diff(that: Set[A]): Set[A]** 计算这组和另一组的差异。 |
| 15 | **def drop(n: Int): Set[A]]** 返回除了第n个的所有元素。 |
| 16 | **def dropRight(n: Int): Set[A]** 返回除了最后的n个的所有元素。 |
| 17 | **def dropWhile(p: (A) => Boolean): Set[A]** 丢弃满足谓词的元素最长前缀。 |
| 18 | **def equals(that: Any): Boolean** equals方法的任意序列。比较该序列到某些其他对象。 |
| 19 | **def exists(p: (A) => Boolean): Boolean** 测试谓词是否持有一些这种不可变的集合的元素。 |
| 20 | **def filter(p: (A) => Boolean): Set[A]** 返回此不可变的集合满足谓词的所有元素。 |
| 21 | **def find(p: (A) => Boolean): Option[A]** 找到不可变的集合满足谓词，如有第一个元素 |
| 22 | **def forall(p: (A) => Boolean): Boolean** 测试谓词是否持有这种不可变的集合中的所有元素。 |
| 23 | **def foreach(f: (A) => Unit): Unit** 应用一个函数f这个不可变的集合中的所有元素。 |
| 24 | **def head: A** 返回此不可变的集合的第一个元素。 |
| 25 | **def init: Set[A]** 返回除了最后的所有元素。 |
| 26 | **def intersect(that: Set[A]): Set[A]** 计算此set和另一组set之间的交叉点。 |
| 27 | **def isEmpty: Boolean** 测试此集合是否为空。 |
| 28 | **def iterator: Iterator[A]** 创建一个新的迭代器中包含的可迭代对象中的所有元素。 |
| 29 | **def last: A** 返回最后一个元素。 |
| 30 | **def map[B](f: (A) => B): immutable.Set[B]** 通过应用函数这一不可变的集合中的所有元素构建一个新的集合。 |
| 31 | **def max: A** 查找最大的元素。 |
| 32 | **def min: A** 查找最小元素。 |
| 33 | **def mkString: String** 显示此不可变的集合字符串中的所有元素。 |
| 34 | **def mkString(sep: String): String** 显示此不可变的集合在一个字符串使用分隔字符串的所有元素。 |
| 35 | **def product: A** 返回此不可变的集合相对于\*操作在num的所有元素的产物。 |
| 36 | **def size: Int** 返回此不可变的集合元素的数量。 |
| 37 | **def splitAt(n: Int): (Set[A], Set[A])** 返回一对不可变的集合组成这个不可变的集的前n个元素，以及其他元素。 |
| 38 | **def subsetOf(that: Set[A]): Boolean** 返回true，如果此set就是一个子集，也就是说，如果这集合的每个元素也是一个元素。 |
| 39 | **def sum: A** 返回此不可变的集合的所有元素的总和使用对于+运算符在num。 |
| 40 | **def tail: Set[A]** 返回一个不可变的集合组成这个不可变的集合的所有元素，除了第一个。 |
| 41 | **def take(n: Int): Set[A]** 返回前n个元素。 |
| 42 | **def takeRight(n: Int):Set[A]** 返回最后n个元素。 |
| 43 | **def toArray: Array[A]** 返回一个包含此不可变的集合的所有元素的数组。 |
| 44 | **def toBuffer[B >: A]: Buffer[B]** 返回一个包含此不可变的集合中的所有元素的缓冲区。 |
| 45 | **def toList: List[A]** 返回一个包含此不可变的集合中的所有元素的列表。 |
| 46 | **def toMap[T, U]: Map[T, U]** 这种不可变的集合转换为映射 |
| 47 | **def toSeq: Seq[A]** 返回一个包含此不可变的集的所有元素的序列。 |
| 48 | **def toString(): String** 返回对象的字符串表示。 |

## 3、Maps

映射是键/值对的集合。任何值可以根据它的键进行检索。

Scala中的映射是键/值对的集合。任何值可以根据它的键进行检索。键是在映射唯一的，但值不一定是唯一的。映射也被称为哈希表。有两种类型的映射，不可变以及可变的。可变和不可变的对象之间的区别在于，当一个对象是不可变的，对象本身不能被改变。

默认情况下，Scala中使用不可变的映射。如果想使用可变集，必须明确地导入scala.collection.mutable.Map类。如果想在同 一个同时使用可变和不可变的映射，那么可以继续参考不可变的映射作为映射，但可以参考可变集合为mutable.Map。以下是该示例声明不可变的映射如 下：

// Empty hash table whose keys are strings and values are integers:

var A:Map[Char,Int] = Map()

// A map with keys and values.

val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF")在定义空映射，类型注释是必要的，因为系统需要指定一个具体的类型变量。如果我们要一个键值对添加到映射，我们可以使用运算符+如下：

A += ('I' -> 1) A += ('J' -> 5) A += ('K' -> 10) A += ('L' -> 100)

### 映射的基本操作：

在映射上的所有操作可被表示在下面的三种方法：

|  |  |
| --- | --- |
| **方法** | **描述** |
| keys | 这个方法返回一个包含映射中的每个键的迭代。 |
| values | 这个方法返回一个包含映射中的每个值的迭代。 |
| isEmpty | 如果映射为空此方法返回true，否则为false。 |

以下是上述方法中的例子显示的用法：

object Test { def main(args: Array[String]) { val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F") val nums: Map[Int, Int] = Map() println( "Keys in colors : " + colors.keys ) println( "Values in colors : " + colors.values ) println( "Check if colors is empty : " + colors.isEmpty ) println( "Check if nums is empty : " + nums.isEmpty ) } }

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala C:/>scala Test Keys in colors : Set(red, azure, peru) Values in colors : MapLike(#FF0000, #F0FFFF, #CD853F) Check if colors is empty : false Check if nums is empty : true C:/>

### 串联映射

可以使用++运算符或映射。++()方法来连接两个或更多的映射，但同时增加了映射，将删除重复的键。下面是一个例子来连接两个映射：

object Test { def main(args: Array[String]) { val colors1 = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F") val colors2 = Map("blue" -> "#0033FF", "yellow" -> "#FFFF00", "red" -> "#FF0000") // use two or more Maps with ++ as operator var colors = colors1 ++ colors2 println( "colors1 ++ colors2 : " + colors ) // use two maps with ++ as method colors = colors1.++(colors2) println( "colors1.++(colors2)) : " + colors ) } }

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala C:/>scala Test colors1 ++ colors2 : Map(blue -> #0033FF, azure -> #F0FFFF, peru -> #CD853F, yellow -> #FFFF00, red -> #FF0000) colors1.++(colors2)) : Map(blue -> #0033FF, azure -> #F0FFFF, peru -> #CD853F, yellow -> #FFFF00, red -> #FF0000) C:/>

### 打印映射的键和值：

可以通过使用foreach循环重复Map的键和值。以下为例子来说明的用法：

object Test { def main(args: Array[String]) { val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F") colors.keys.foreach{ i => print( "Key = " + i ) println(" Value = " + colors(i) )} } }

在这里，我们使用迭代器相关的foreach遍历键方法。当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala C:/>scala Test Key = red Value = #FF0000 Key = azure Value = #F0FFFF Key = peru Value = #CD853F C:/>

### 检查映射中的键：

可以使用 Map.contains 方法来测试，如果给定的键存在于映射或没有。以下为例子来说明的用法：

object Test { def main(args: Array[String]) { val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F") if( colors.contains( "red" )){ println("Red key exists with value :" + colors("red")) }else{ println("Red key does not exist") } if( colors.contains( "maroon" )){ println("Maroon key exists with value :" + colors("maroon")) }else{ println("Maroon key does not exist") } } }

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala C:/>scala Test Red key exists with value :#FF0000 Maroon key does not exist C:/>

### Scala映射的方法：

以下是可以使用映射的重要方法。有关可用方法的完整列表，请Scala的官方文件。

|  |  |
| --- | --- |
| **SN** | **方法及描述** |
| 1 | **def ++(xs: Map[(A, B)]): Map[A, B]** 返回包含此映射的映射和那些xs提供了一个新的映射。 |
| 2 | **def -(elem1: A, elem2: A, elems: A\*): Map[A, B]** 返回包含除具有一个键等于映射elem1，elem2时或任何元素此映射的所有映射的新映射。 |
| 3 | **def --(xs: GTO[A]): Map[A, B]** 返回此映射，除映射一键等于从遍历对象xs的一个键所有键/值映射的新映射。 |
| 4 | **def get(key: A): Option[B]** 可选择返回一个包含键关联的值。 |
| 5 | **def iterator: Iterator[(A, B)]** 创建一个新的迭代器的所有键/值对在此映射 |
| 6 | **def addString(b: StringBuilder): StringBuilder** 追加可收缩集合到一个字符串生成器的所有元素。 |
| 7 | **def addString(b: StringBuilder, sep: String): StringBuilder** 追加可收缩集合到使用分隔字符串一个字符串生成器的所有元素。 |
| 8 | **def apply(key: A): B** 返回给定键或者映射的默认方法的结果相关联的，如果不存在值。 |
| 9 | **def clear(): Unit** 从映射中删除所有绑定。在此之后操作已完成时，映射将是空的。 |
| 10 | **def clone(): Map[A, B]** 创建接收器对象的副本。 |
| 11 | **def contains(key: A): Boolean** 如果有一个绑定在该映射的键返回true，否则为false。 |
| 12 | **def copyToArray(xs: Array[(A, B)]): Unit** 复制这个可收缩集合值的数组。填充给定的数组xs与此可收缩集合值。 |
| 13 | **def count(p: ((A, B)) => Boolean): Int** 计算满足谓词在可收缩集合元素的数量。 |
| 14 | **def default(key: A): B** 定义默认值计算为映射，当找不到一个键返回。 |
| 15 | **def drop(n: Int): Map[A, B]** 返回除了第n个的所有元素。 |
| 16 | **def dropRight(n: Int): Map[A, B]** 返回除了最后n个的所有元素 |
| 17 | **def dropWhile(p: ((A, B)) => Boolean): Map[A, B]** 丢弃满足谓词的元素最长前缀。 |
| 18 | **def empty: Map[A, B]** 返回相同类型的，因为这映射的空映射。 |
| 19 | **def equals(that: Any): Boolean** 返回true，如果两个映射包含完全相同的键/值，否则为false。 |
| 20 | **def exists(p: ((A, B)) => Boolean): Boolean** 返回true如果给定的断言p成立了一些这方面可收缩集合的元素，否则为false。 |
| 21 | **def filter(p: ((A, B))=> Boolean): Map[A, B]** 返回此可收缩集合满足谓词的所有元素。 |
| 22 | **def filterKeys(p: (A) => Boolean): Map[A, B]** 返回一个不可变的映射只包含那些键值对这个映射，重点满足谓词p |
| 23 | **def find(p: ((A, B)) => Boolean): Option[(A, B)]** 查找可收缩集合满足谓词，任何的第一要素 |
| 24 | **def foreach(f: ((A, B)) => Unit): Unit** 应用一个函数f这种可收缩集合中的所有元素 |
| 25 | **def init: Map[A, B]** 返回除了最后的所有元素 |
| 26 | **def isEmpty: Boolean** 测试映射是否为空 |
| 27 | **def keys: Iterable[A]** 返回迭代所有键 |
| 28 | **def last: (A, B)** 返回最后一个元素 |
| 29 | **def max: (A, B)** 查找最大的元素 |
| 30 | **def min: (A, B)** 查找最小元素 |
| 31 | **def mkString: String** 显示此可收缩集合字符串中的所有元素 |
| 32 | **def product: (A, B)** 返回此可收缩集合相对于所述运算符\*在num所有元素的乘积 |
| 33 | **def remove(key: A): Option[B]** 移除此映射一个键，返回先前与该键作为一个选项相关联的值 |
| 34 | **def retain(p: (A, B) => Boolean): Map.this.type** 只保留那些映射其中谓词p返回true |
| 35 | **def size: Int** 返回在此映射的元素的数量。 |
| 36 | **def sum: (A, B)** 返回此可收缩集合中的所有元素的总和使用+运算符在num |
| 37 | **def tail: Map[A, B]** 返回除了第一元素外的所有元素 |
| 38 | **def take(n: Int): Map[A, B]** 返回前n个元素 |
| 39 | **def takeRight(n: Int): Map[A, B]** 返回最后n个元素 |
| 40 | **def takeWhile(p: ((A, B)) => Boolean): Map[A, B]** 满足谓词的元素最长前缀 |
| 41 | **def toArray: Array[(A, B)]** 这个可收缩集合转换成数组 |
| 42 | **def toBuffer[B >: A]: Buffer[B]** 返回包含此映射中的所有元素的缓冲区 |
| 43 | **def toList: List[A]** 返回包含此映射中的所有元素的列表 |
| 44 | **def toSeq: Seq[A]** 返回包含此映射中的所有元素的序列 |
| 45 | **def toSet: Set[A]** 返回一组包含此映射中的所有元素 |
| 46 | **def toString(): String** 返回对象的字符串表示 |

## 4、Tuples

不像数组或列表，元组可以容纳不同类型的对象。

Scala的元组结合件多个固定数量在一起，使它们可以被传来传去作为一个整体。不像一个数组或列表，元组可以容纳不同类型的对象，但它们也是不可改变的。这里是一个元组持有整数，字符串和Console，如下的一个例子：

val t = (1, "hello", Console)

这是语法修饰(快捷方式)以下：

val t = new Tuple3(1, "hello", Console)

一个元组的实际类型取决于它包含的元素和这些元素的类型的数目。因此，该类型 (99, "Luftballons") 是 Tuple2[Int, String].  ('u', 'r', "the", 1, 4, "me") 的类型是 Tuple6[Char, Char, String, Int, Int, String]

元组类型Tuple1，Tuple2，Tuple3等。至少目前为22的上限在Scala，如果需要更多，那么可以使用一个集合，而不是一个元组。对于每 个TupleN类型，其中1<= N <= 22，Scala定义了许多元素的访问方法。给出了以下定义：

val t = (4,3,2,1)

要访问的元组 t 的元素，可以使用的方法t.\_1访问的第一个元素，t.\_2进入第二个，依此类推。例如，下面的表达式计算t的所有元素的总和：

val sum = t.\_1 + t.\_2 + t.\_3 + t.\_4

可以使用三元组来写一个方法，它接受一个List[Double]，并返回计数的总和，并在三个元素的元组返回平方和，Tuple3[Int, Double, Double]。它们也非常有用传递的数据值的列表并发编程之间的消息。下面是一个元组的例子显示使用：

object Test { def main(args: Array[String]) { val t = (4,3,2,1) val sum = t.\_1 + t.\_2 + t.\_3 + t.\_4 println( "Sum of elements: " + sum ) } }

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala C:/>scala Test Sum of elements: 10 C:/>

### 遍历元组：

可以使用Tuple.productIterator()方法来遍历一个元组的所有元素。下面是一个例子来连接两个图：

object Test { def main(args: Array[String]) { val t = (4,3,2,1) t.productIterator.foreach{ i =>println("Value = " + i )} } }

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala C:/>scala Test Value = 4 Value = 3 Value = 2 Value = 1 C:/>

### 转换为字符串：

可以使用Tuple.toString()方法来连接的元组的所有元素为一个字符串。以下为例子来说明的用法：

object Test { def main(args: Array[String]) { val t = new Tuple3(1, "hello", Console) println("Concatenated String: " + t.toString() ) } }

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala C:/>scala Test Concatenated String: (1,hello,scala.Console$@281acd47) C:/>

### 交换元素：

可以使用Tuple.swap方法来交换一个Tuple2的元素。以下为例子来说明的用法：

object Test { def main(args: Array[String]) { val t = new Tuple2("Scala", "hello") println("Swapped Tuple: " + t.swap ) } }

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala C:/>scala Test Swapped tuple: (hello,Scala) C:/>

## 5、Options

Option[T] 提供了一种容器，用于给定类型的零个或一个元素。

Scala的Option[T]是容器对于给定的类型的零个或一个元件。Option[T]可以是一些[T]或None对象，它代表一个缺失值。例 如，Scala映射get方法产生，如果给定的键没有在映射定义的一些(值)，如果对应于给定键的值已经找到，或None。选项Option类型常用于 Scala程序，可以比较这对null值Java可用这表明没有任何值。例如，java.util.HashMap中的get方法将返回存储在 HashMap的值，或null，如果找到没有任何值。

比方说，我们有一个检索来自基于主键数据库中的记录的方法：

def findPerson(key: Int): Option[Person]

该方法会返回一些 [人士] 如果找到记录，如果找不到记录那么为None。让我们来看看一个真实的例子：

object Test { def main(args: Array[String]) { val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo") println("capitals.get( "France" ) : " + capitals.get( "France" )) println("capitals.get( "India" ) : " + capitals.get( "India" )) } }

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala C:/>scala Test capitals.get( "France" ) : Some(Paris) capitals.get( "India" ) : None C:/>

使用可选值最常用的方法是模式匹配。例如：

object Test { def main(args: Array[String]) { val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo") println("show(capitals.get( "Japan")) : " + show(capitals.get( "Japan")) ) println("show(capitals.get( "India")) : " + show(capitals.get( "India")) ) } def show(x: Option[String]) = x match { case Some(s) => s case None => "?" } }

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala C:/>scala Test show(capitals.get( "Japan")) : Tokyo show(capitals.get( "India")) : ? C:/>

### 使用getOrElse()方法:

以下是演示如何使用getOrElse()来访问值或使用默认值，当没有值的例子：

object Test { def main(args: Array[String]) { val a:Option[Int] = Some(5) val b:Option[Int] = None println("a.getOrElse(0): " + a.getOrElse(0) ) println("b.getOrElse(10): " + b.getOrElse(10) ) } }

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala C:/>scala Test a.getOrElse(0): 5 b.getOrElse(10): 10 C:/>

### 使用isEmpty()方法:

以下是演示如何使用isEmpty()检查该选项是否为 None 的例子：

object Test { def main(args: Array[String]) { val a:Option[Int] = Some(5) val b:Option[Int] = None println("a.isEmpty: " + a.isEmpty ) println("b.isEmpty: " + b.isEmpty ) } }

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala C:/>scala Test a.isEmpty: false b.isEmpty: true C:/>

### Scala Option方法:

以下可以使用Options的一些重要方法。有关可用方法的完整列表，请Scala的的官方文档。

|  |  |
| --- | --- |
| **SN** | **方法以及描述** |
| 1 | **def get: A** 返回选项的值 |
| 2 | **def isEmpty: Boolean** 如果该选项为None返回true，否则为false |
| 3 | **def productArity: Int** 该产品的尺寸大小。对于一个产品A(x\_1, ..., x\_k), 返回k |
| 4 | **def productElement(n: Int): Any** 该产品的第n个元素，0为基础。换言之，对于一个产品A(x\_1, ..., x\_k),，返回 x\_(n+1)，其中 0 < n < k。 |
| 5 | **def exists(p: (A) => Boolean): Boolean** 返回true，如果此选项非空，当应用于此选项的值谓词p返回true。否则返回false。 |
| 6 | **def filter(p: (A) => Boolean): Option[A]** 返回此选项，如果它不为空，并应用谓词p这个选项的值返回true。否则返回None。 |
| 7 | **def filterNot(p: (A) => Boolean): Option[A]** 返回此选项，如果它不为空，并应用谓词p这个选项的值返回false。否则返回None。 |
| 8 | **def flatMap[B](f: (A) => Option[B]): Option[B]** 返回将f应用这个选项的值，如果这个选项不为空的结果。返回None如果此选项是空。 |
| 9 | **def foreach[U](f: (A) => U): Unit** 应用过程f给到该选项的值，如果它不为空。否则，什么都不做。 |
| 10 | **def getOrElse[B >: A](default: => B): B** 返回选项的值，如果选择不为空，否则返回默认计算的结果。 |
| 11 | **def isDefined: Boolean** 如果该选项实例为一些返回true，否则为false。 |
| 12 | **def iterator: Iterator[A]** 如果它不为空，返回一个单例迭代器返回选项的值，或者如果该选项为空返回空迭代器。 |
| 13 | **def map[B](f: (A) => B): Option[B]** 如果这个选项不为空的结果，返回一个包含一些将f应用这个选项的值。否则返回None。 |
| 14 | **def orElse[B >: A](alternative: => Option[B]): Option[B]** 返回此选项Option，如果它不为空，否则返回计算替代的结果。 |
| 15 | **def orNull** 如果它不为空返回该选项的值，如果它是空则返回null。 |

## 6、Iterators

迭代不是集合，而是一种由一个访问的集合之一的元素。

迭代器不是集合，而是一种由一个访问的集合之一的元素。在一个迭代的两种基本操作：next和hasNext。调用 it.next()将返回迭代器的下一个元素，推进迭代器的状态。可以找出是否有更多的元素使用迭代器的it.hasNext方法返回。

最简单的方法是使用while循环“单步”将迭代器返回所有的元素。来看一个真实的例子：

object Test { def main(args: Array[String]) { val it = Iterator("a", "number", "of", "words") while (it.hasNext){ println(it.next()) } } }

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala C:/>scala Test a number of words C:/>

### 查找最小和最大值的元素：

可以使用 it.min 和 it.max 方法在一个迭代中找出最小和最大值元素。下面是使用示例：

object Test { def main(args: Array[String]) { val ita = Iterator(20,40,2,50,69, 90) val itb = Iterator(20,40,2,50,69, 90) println("Maximum valued element " + ita.max ) println("Minimum valued element " + itb.min ) } }

这里，我们用ita和itb执行两个不同的操作，因为迭代可以被通过一次。当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala C:/>scala Test Maximum valued element 90 Minimum valued element 2 C:/>

### 找出迭代器的长度：

可以使用以下任一方法：it.size 或 it.length 找出在一个迭代可用元素的数目。下面是使用：

object Test { def main(args: Array[String]) { val ita = Iterator(20,40,2,50,69, 90) val itb = Iterator(20,40,2,50,69, 90) println("Value of ita.size : " + ita.size ) println("Value of itb.length : " + itb.length ) } }

这里，我们用ita和itb执行两个不同的操作，因为迭代可以被通过一次。当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala C:/>scala Test Value of ita.size : 6 Value of itb.length : 6 C:/>

### Scala迭代器方法：

以下是可以使用迭代器的一些重要方法。有关可用方法的完整列表，请Scala的的官方文档。

|  |  |
| --- | --- |
| **SN** | **方法及描述** |
| 1 | **def hasNext: Boolean** 测试此迭代器是否能提供另一个元素。 |
| 2 | **def next(): A** 产生这个迭代的下一个元素。 |
| 3 | **def ++(that: => Iterator[A]): Iterator[A]** 此迭代器与另一个连接。 |
| 4 | **def ++[B >: A](that :=> GenTraversableOnce[B]): Iterator[B]** 此迭代器与另一个连接。 |
| 5 | **def addString(b: StringBuilder): StringBuilder** 返回字符串生成器b到元素被追加。 |
| 6 | **def addString(b: StringBuilder, sep: String): StringBuilder** 返回字符串生成器b到元素被追加使用分隔字符串。 |
| 7 | **def buffered: BufferedIterator[A]** 从缓冲迭代器创建此迭代器。 |
| 8 | **def contains(elem: Any): Boolean** 测试此迭代器是否包含给定值作为一个元素。 |
| 9 | **def copyToArray(xs: Array[A], start: Int, len: Int): Unit** 将选定这个迭代器数组产生的值。 |
| 10 | **def count(p: (A) => Boolean): Int** 计算在遍历或迭代满足谓词的元素数。 |
| 11 | **def drop(n: Int): Iterator[A]** 这个推进过去迭代的前n个元素，或迭代器中的较小值长度。 |
| 12 | **def dropWhile(p: (A) => Boolean): Iterator[A]** 跳过此迭代器满足给定谓词P元素的最长序列，并返回剩余的元素的迭代器。 |
| 13 | **def duplicate: (Iterator[A], Iterator[A])** 创建两个新的迭代，这两个迭代相同的元件，因为这迭代（以相同的顺序）。 |
| 14 | **def exists(p: (A) => Boolean): Boolean** 返回true如果给定的断言p成立一些由该迭代器所产生的值，否则为false。 |
| 15 | **def filter(p: (A) => Boolean): Iterator[A]** 返回一个迭代在这个迭代器，满足谓词p的所有元素。元素的顺序被保留。 |
| 16 | **def filterNot(p: (A) => Boolean): Iterator[A]** 创建一个迭代器在这个迭代器不符合谓词p的所有元素。 |
| 17 | **def find(p: (A) => Boolean): Option[A]** 查找由该迭代器满足谓词，如果任意所产生的第一个值。 |
| 18 | **def flatMap[B](f: (A) => GenTraversableOnce[B]): Iterator[B]** 创建一个新的迭代器通过应用功能，通过这个迭代器产生的所有值和串联的结果。 |
| 19 | **def forall(p: (A) => Boolean): Boolean** 返回true如果给定的断言p成立了由该迭代器所产生的所有值，否则为false。 |
| 20 | **def foreach(f: (A) => Unit): Unit** 应用一个函数f通过这个迭代器生成的所有值。 |
| 21 | **def hasDefiniteSize: Boolean** 返回true如果是空迭代器，否则为false。 |
| 22 | **def indexOf(elem: B): Int** 返回此迭代的对象的第一个出现的指定对象的索引。 |
| 23 | **def indexWhere(p: (A) => Boolean): Int** 返回第一个生产值满足谓词的索引，或-1。 |
| 24 | **def isEmpty: Boolean** 如果hasNext是false返回true，否则为false。 |
| 25 | **def isTraversableAgain: Boolean** 测试这个迭代器是否可以反复运行。 |
| 26 | **def length: Int** 返回此迭代的元素数。迭代器就是在这个方法返回后结束 |
| 27 | **def map[B](f: (A) => B): Iterator[B]** 返回一个新的迭代器，其将通过这个迭代器生产应用函数 f 给它的每一个值 |
| 28 | **def max: A** 查找最大元素。迭代器就是在这个方法返回后结束 |
| 29 | **def min: A** 查找最小元素，迭代器就是在这个方法返回后结束 |
| 30 | **def mkString: String** 显示此遍历的迭代器在一个字符串的所有元素 |
| 31 | **def mkString(sep: String): String** 使用分隔字符串显示在一个字符串这个遍历的迭代器的所有元素 |
| 32 | **def nonEmpty: Boolean** 测试可遍历迭代器是不是为空 |
| 33 | **def padTo(len: Int, elem: A): Iterator[A]** 追加的元素值到这个迭代器的一个给定目标长度 |
| 34 | **def patch(from: Int, patchElems: Iterator[B], replaced: Int): Iterator[B]** 返回此迭代器补丁值 |
| 35 | **def product: A** 这个集合中的元素相乘 |
| 36 | **def sameElements(that: Iterator[\_]): Boolean** 返回true，如果两个迭代器产生相同的顺序相同的元素，否则为false |
| 37 | **def seq: Iterator[A]** 返回集合的顺序视图 |
| 38 | **def size: Int** 返回此遍历或迭代的元素数 |
| 39 | **def slice(from: Int, until: Int): Iterator[A]** 创建一个迭代器返回由这个迭代器所产生的值区间 |
| 40 | **def sum: A** 返回此遍历的迭代器的所有元素的总和使用对于+运算符在num |
| 41 | **def take(n: Int): Iterator[A]** 返回一个迭代只生产这个迭代器，否则整个迭代器的前n个值，如果它产生少于n个值 |
| 42 | **def toArray: Array[A]** 返回包含此遍历的迭代器的所有元素的数组 |
| 43 | **def toBuffer: Buffer[B]** 返回包含此遍历的迭代器的所有元素的缓冲区 |
| 44 | **def toIterable: Iterable[A]** 返回包含此遍历的迭代器的所有元素一个可迭代，这不会终止无限迭代器 |
| 45 | **def toIterator: Iterator[A]** 返回包含此遍历的迭代器的所有元素的迭代器，这不会终止无限迭代器 |
| 46 | **def toList: List[A]** 返回包含此遍历的迭代器的所有元素的列表 |
| 47 | **def toMap[T, U]: Map[T, U]** 返回包含此遍历的迭代器的所有元素的映射 |
| 48 | **def toSeq: Seq[A]** 返回包含此遍历的迭代器的所有元素的序列 |
| 49 | **def toString(): String** 此迭代器转换为字符串 |
| 50 | **def zip[B](that: Iterator[B]): Iterator[(A, B)** 返回一个新的包含迭代器对由此迭代器和相应的元素。由新的迭代器返回的元素数为最小值，通过此迭代器和返回元素的数目 |

# 类和对象

类是一个对象的蓝图。一旦定义一个类可以创建从类蓝图使用关键字new创建对象。下面是一个简单的语法在Scala中定义一个类：

class Yiibai(xc: Int, yc: Int) {

var x: Int = xc

var y: Int = yc

def move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

println ("Yiibai x location : " + x);

println ("Yiibai y location : " + y);

}

}

这个类定义了两个变量x和y和方法：move，没有返回值。类变量被调用，类的字段和方法被称为类方法。

类名可以作为一个类的构造函数，可以采取一些参数。上面的代码定义了两个构造函数的参数：xc和yc;它们都在类的主体内可见。

正如前面提到的，可以使用关键字new创建对象，然后可以按照下面的例子所示访问类的字段和方法：

import java.io.\_

class Yiibai(val xc: Int, val yc: Int) {

var x: Int = xc

var y: Int = yc

def move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

println ("Yiibai x location : " + x);

println ("Yiibai y location : " + y);

}

}

object Test {

def main(args: Array[String]) {

val pt = new Yiibai(10, 20);

// Move to a new location

pt.move(10, 10);

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Yiibai x location : 20

Yiibai y location : 30

C:/>

## 扩展一个类：

可以扩展scala类以类似的方式，如在Java中的一样，但有两个限制：方法重载需要override关键字，只有主构造可以传递参数给基构造。现在扩展上面的类，并增加一个类的方法：

class Yiibai(val xc: Int, val yc: Int) {

var x: Int = xc

var y: Int = yc

def move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

println ("Yiibai x location : " + x);

println ("Yiibai y location : " + y);

}

}

class Location(override val xc: Int, override val yc: Int,

val zc :Int) extends Yiibai(xc, yc){

var z: Int = zc

def move(dx: Int, dy: Int, dz: Int) {

x = x + dx

y = y + dy

z = z + dz

println ("Yiibai x location : " + x);

println ("Yiibai y location : " + y);

println ("Yiibai z location : " + z);

}

}

**extends**子句有两种作用：它使类Location继承类Yiibai所有非私有成员，它使Location类 作为Yiibai类的子类。 因此，这里的Yiibai类称为超类，而Location类被称为子类。扩展一个类，继承父类的所有功能，被称为继承，但 scala允许继承，只能从一个唯一的类。让我们看看完整的例子，显示继承的用法：

import java.io.\_

class Yiibai(val xc: Int, val yc: Int) {

var x: Int = xc

var y: Int = yc

def move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

println ("Yiibai x location : " + x);

println ("Yiibai y location : " + y);

}

}

class Location(override val xc: Int, override val yc: Int,

val zc :Int) extends Yiibai(xc, yc){

var z: Int = zc

def move(dx: Int, dy: Int, dz: Int) {

x = x + dx

y = y + dy

z = z + dz

println ("Yiibai x location : " + x);

println ("Yiibai y location : " + y);

println ("Yiibai z location : " + z);

}

}

object Test {

def main(args: Array[String]) {

val loc = new Location(10, 20, 15);

// Move to a new location

loc.move(10, 10, 5);

}

}

需要注意的是方法**move，**不会覆盖 **move** 方法相应的定义，因为它们是不同的定义（例如，前两个参数，而后者则需要三个参数）。  
让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

Yiibai x location : 20

Yiibai y location : 30

Yiibai z location : 20

C:/>

## 单例对象：

Scala比Java更面向对象，因为在Scala中不能有静态成员。相反，Scala有单例的对象。单例就是只能有一个实例，即，类的对象。可以使用关 键字object代替class关键字，而不是创建单例。因为不能实例化一个单独的对象，不能将参数传递给主构造。前面已经看到全部采用单一对象，调用 Scala的main方法的例子。以下是单例显示的相同的例子：

import java.io.\_

class Yiibai(val xc: Int, val yc: Int) {

var x: Int = xc

var y: Int = yc

def move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

}

}

object Test {

def main(args: Array[String]) {

val yiibai = new Yiibai(10, 20)

printYiibai

def printYiibai{

println ("Yiibai x location : " + yiibai.x);

println ("Yiibai y location : " + yiibai.y);

}

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Yiibai x location : 10

Yiibai y location : 20

C:/>

# 特征

特性封装方法和字段定义，然后可以通过将它们混合成类被重用。不同于类继承，其中每个类都必须继承只有一个父类，一类可以在任意数量特质混合。

特征用于通过指定支持的方法的签名定义的对象类型。Scala中也允许部分实现特性但可能不具有构造参数。

一个特征定义看起来就像不同的是它使用关键字trait如下类定义：

trait Equal {

def isEqual(x: Any): Boolean

def isNotEqual(x: Any): Boolean = !isEqual(x)

}

这种特质由两个方法的isEqual和isNotEqual。这里，我们没有给出任何实现的isEqual其中作为另一种方法有它的实现。扩展特性的子类可以给实施未实现的方法。因此，一个特点是非常相似Java的抽象类。下面是一个完整的例子来说明特性的概念：

trait Equal {

def isEqual(x: Any): Boolean

def isNotEqual(x: Any): Boolean = !isEqual(x)

}

class Yiibai(xc: Int, yc: Int) extends Equal {

var x: Int = xc

var y: Int = yc

def isEqual(obj: Any) =

obj.isInstanceOf[Yiibai] &&

obj.asInstanceOf[Yiibai].x == x

}

object Test {

def main(args: Array[String]) {

val p1 = new Yiibai(2, 3)

val p2 = new Yiibai(2, 4)

val p3 = new Yiibai(3, 3)

println(p1.isNotEqual(p2))

println(p1.isNotEqual(p3))

println(p1.isNotEqual(2))

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

false

true

true

C:/>

**什么时候使用特性？**

没有严格的规定，但这里有一些指导原则需要考虑：

* 如果行为不被重用，则要使它成为一个具体的类。它毕竟不是可重复使用的行为。
* 如果它可能在多个不相关的类被重用，使它成为一个性状。只有特性可混入的类层次结构的不同部分。
* 如果想它从继承Java代码，使用抽象类。
* 如果打算在已编译的形式分发，而且希望外部组织编写的类继承它，可能会倾向于使用抽象类。
* 如果效率是非常重要的，倾向于使用类。

# 模式匹配

模式匹配是Scala中第二个最广泛使用的功能，经过函数值和闭包。Scala中大力支持模式匹配处理消息。

模式匹配包括替代的序列，每个开始使用关键字case。每个备选中包括模式和一个或多个表达式，如果模式匹配将被计算。一个箭头符号=>分开的表达模式。这里是一个小例子，它展示了如何匹配一个整数值：

object Test {

def main(args: Array[String]) {

println(matchTest(3))

}

def matchTest(x: Int): String = x match {

case 1 => "one"

case 2 => "two"

case \_ => "many"

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

many

C:/>

使用case语句块定义一个函数，该函数映射整数字符串。匹配关键字提供应用函数(如模式匹配函数以上)为一个对象的一个方便的方法。下面是第二个示例，它匹配针对不同类型的模式值：

object Test {

def main(args: Array[String]) {

println(matchTest("two"))

println(matchTest("test"))

println(matchTest(1))

}

def matchTest(x: Any): Any = x match {

case 1 => "one"

case "two" => 2

case y: Int => "scala.Int"

case \_ => "many"

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

2

many

one

C:/>

第一个case ，如果 x 指的是整数值1. 如果x等于字符串“2”的第二case相匹配匹配。第三种case 是由一个输入模式;它匹配针对任何整数，并结合选择值xto整数类型的变量y。以下为文字相同的匹配... case 表达式用括号 {...} 另一种形式：

object Test {

def main(args: Array[String]) {

println(matchTest("two"))

println(matchTest("test"))

println(matchTest(1))

}

def matchTest(x: Any){

x match {

case 1 => "one"

case "two" => 2

case y: Int => "scala.Int"

case \_ => "many"

}

}

}

**匹配使用case 类：**

case classes是用于模式匹配与case 表达式指定类。这些都是标准类具有特殊修饰：case。下面是一个简单的模式使用case class匹配示例：

object Test {

def main(args: Array[String]) {

val alice = new Person("Alice", 25)

val bob = new Person("Bob", 32)

val charlie = new Person("Charlie", 32)

for (person <- List(alice, bob, charlie)) {

person match {

case Person("Alice", 25) => println("Hi Alice!")

case Person("Bob", 32) => println("Hi Bob!")

case Person(name, age) =>

println("Age: " + age + " year, name: " + name + "?")

}

}

}

// case class, empty one.

case class Person(name: String, age: Int)

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Hi Alice!

Hi Bob!

Age: 32 year, name: Charlie?

C:/>

增加 case 关键字使编译器自动添加了许多实用的功能。关键字建议与模式匹配的情况下表达式的关联。

首先，编译器会自动转换的构造函数的参数为不可变的字段（vals）。val关键字是可选的。如果想可变字段，使用var关键字。因此，构造函数的参数列表现在更短。

其次，编译器自动实现equals, hashCode, 和toString方法的类，它使用指定为构造函数参数的字段。因此，不再需要自己的toString方法。

最后，还消失Person类的主体部分，因为没有需要定义的方法！

# 正则表达式

Scala支持通过Regex类的scala.util.matching封装正则表达式。让我们看看一个例子，我们将尝试从Scala中一个语句中找出单词：

import scala.util.matching.Regex

object Test {

def main(args: Array[String]) {

val pattern = "Scala".r

val str = "Scala is Scalable and cool"

println(pattern findFirstIn str)

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Some(Scala)

C:/>

我们创建一个字符串，并调用r()方法就可以了。Scala中字符串隐式转换为一个RichString并调用该方法来获得正则表达式的一个实例。找到第 一个正则表达式匹配，只需调用findFirstIn()方法。而非只找到第一次出现。如果想找到匹配的单词的所有事件，可以使用findAllIn() 方法，并在情况下，有目标字符串中使用多个Scala的单词，这将返回所有匹配的集合单词。

可以使用mkString()方法来连接所产生的服务，可以使用管道(|)搜索Scala中小型和资本的情况下，使用正则表达式构造来代替或r()方法创建一个模式如下：

import scala.util.matching.Regex

object Test {

def main(args: Array[String]) {

val pattern = new Regex("(S|s)cala")

val str = "Scala is scalable and cool"

println((pattern findAllIn str).mkString(","))

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Scala,scala

C:/>

如果想更换匹配的文本，可以使用replaceFirstIn()以取代第一个匹配项或replaceAllIn()，以取代所有出现如下：

object Test {

def main(args: Array[String]) {

val pattern = "(S|s)cala".r

val str = "Scala is scalable and cool"

println(pattern replaceFirstIn(str, "Java"))

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Java is scalable and cool

C:/>

**形成正则表达式：**

Scala继承了Java，这反过来又继承了大部分的Perl的功能，它的正则表达式语法。这里只是一些例子，应该是足够的说明：

下面是表，列出了所有的正则表达式元字符的语法可用在Java中：

|  |  |
| --- | --- |
| **子表达式** | **匹配** |
| ^ | 匹配行头 |
| $ | 匹配行尾 |
| . | 匹配除换行符任何单个字符。用m选项也允许使之匹配换行符。 |
| [...] | 匹配括号内任何单个字符。 |
| [^...] | 匹配任何单个字符不是在括号中 |
| A | 整个字符串的开始 |
| z | 整个字符串结束 |
|  | 最终，除了允许的最后行结束整个字符串。 |
| re\* | 匹配0或多次出现前面表达式。 |
| re+ | 匹配1个或多个的先前东西 |
| re? | 匹配0或1发生前表达式。 |
| re{ n} | 精确匹配n个前面表达式的数量。 |
| re{ n,} | 匹配n或多次出现前面的表达。 |
| re{ n, m} | 至少匹配n和在前面的表现最为m次出现。 |
| a|b | 匹配a或b。 |
| (re) | 组正则表达式并记住匹配的文本。 |
| (?: re) | 组正则表达式而不记住匹配的文本。 |
| (?> re) | 匹配独立模式而不反向追踪。 |
| w | 匹配单词字符。 |
| W | 匹配非单词字符。 |
| s | 匹配空白。相当于 [ f]. |
| S | 匹配非空白。 |
| d | 匹配数字。相当于 [0-9]. |
| D | 匹配非数字。 |
| A | 匹配开始的字符串。 |
|  | 匹配字符串的结尾。如果一个换行符存在，它只是换行之前匹配。 |
| z | 匹配字符串的结尾。 |
| G | 匹配点，最后一次匹配结束。 |
|  | 反向引用以捕获组编号 "n" |
|  | 匹配单词边界之外时，括号内。匹配退格（0×08）括号里面。 |
| B | 匹配非单词边界。 |
| , , etc. | 匹配换行符，回车，制表符等 |
| Q | 转义（引用）所有字符为 E |
| E | 尾部引用开始 Q |

**正则表达式的例子：**

|  |  |
| --- | --- |
| **示例** | **描述** |
| . | 匹配除了换行符的任何字符 |
| [Rr]uby | 匹配 "Ruby" 或"ruby" |
| rub[ye] | 匹配"ruby" 或 "rube" |
| [aeiou] | 匹配任何一个小写元音 |
| [0-9] | 匹配任何数字;同 [0123456789] |
| [a-z] | 匹配任意小写ASCII字母 |
| [A-Z] | 匹配任意大写ASCII字母 |
| [a-zA-Z0-9] | 匹配任何上述 |
| [^aeiou] | 匹配元音以外的任何一个小写字符 |
| [^0-9] | 匹配数字以外的任何其他 |
| d | 匹配一个数字: [0-9] |
| D | 匹配一个非数字: [^0-9] |
| s | 匹配一个空白字符: [ f] |
| S | 匹配非空白: [^ f] |
| w | 匹配一个字符: [A-Za-z0-9\_] |
| W | 匹配 一个非单词字符: [^A-Za-z0-9\_] |
| ruby? | 匹配 "rub" or "ruby": the y is optional |
| ruby\* | 匹配 "rub" plus 0 or more ys |
| ruby+ | 匹配 "rub" plus 1 or more ys |
| d{3} | 匹配只有 3 个数字 |
| d{3,} | 匹配 3 个或多个数字 |
| d{3,5} | 匹配3, 4, 或5 个数字 |
| Dd+ | 不分组: + repeats d |
| (Dd)+/ | 分组: + repeats Dd 对 |
| ([Rr]uby(, )?)+ | 匹配 "Ruby", "Ruby, ruby, ruby", 等. |

需要注意的是每一个反斜杠上述字符串中出现两次。这是因为在Java和Scala一个反斜杠是一个转义字符的字符串，而不是一个普通字符显示出来的字符串。所以不是.. 需要写. 。得到的字符串中的一个反斜杠。请查看下面的例子：

import scala.util.matching.Regex

object Test {

def main(args: Array[String]) {

val pattern = new Regex("abl[ae]d+")

val str = "ablaw is able1 and cool"

println((pattern findAllIn str).mkString(","))

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

able1

C:/>

# 异常处理

Scala的异常的工作像许多其他语言，如Java异常。而不是正常方式返回的值，方法可以通过抛出一个异常终止。然而，Scala实际上并没有检查异常。

当要处理异常，那么可使用try{...}catch{...} 块，就像在Java中除了catch块采用匹配识别和处理异常。

**抛出异常：**

抛出一个异常看起来类似于Java。创建一个异常对象，然后使用throw关键字把它抛出：

throw new IllegalArgumentException

**捕获异常：**

Scala中try/catch在一个单独的块捕捉任何异常，然后使用case块进行模式匹配，如下图所示：

import java.io.FileReader

import java.io.FileNotFoundException

import java.io.IOException

object Test {

def main(args: Array[String]) {

try {

val f = new FileReader("input.txt")

} catch {

case ex: FileNotFoundException =>{

println("Missing file exception")

}

case ex: IOException => {

println("IO Exception")

}

}

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Missing file exception

C:/>

这种try-catch表达的行为在其他语言处理异常是一样的。body是执行体，如果它抛出一个异常，每个catch子句都依次尝试捕获。

**finally子句：**

如果想知道引起一些代码是如何表达的终止执行，可以用一个finally子句包住一个表达式，finally块什么时候都会执行。

import java.io.FileReader

import java.io.FileNotFoundException

import java.io.IOException

object Test {

def main(args: Array[String]) {

try {

val f = new FileReader("input.txt")

} catch {

case ex: FileNotFoundException => {

println("Missing file exception")

}

case ex: IOException => {

println("IO Exception")

}

} finally {

println("Exiting finally...")

}

}

}

当上述代码被编译和执行时，它产生了以下结果：

C:/>scalac Test.scala

C:/>scala Test

Missing file exception

Exiting finally...

C:/>

# 提取器

提取器在Scala中是一个对象，有一个叫非应用作为其成员的一种方法。即不应用方法的目的是要匹配的值，并把它拆开。通常，提取对象还限定了双方法申请构建值，但是这不是必需的。

下面的例子显示电子邮件地址的提取器对象：

object Test {

def main(args: Array[String]) {

println ("Apply method : " + apply("Zara", "gmail.com"));

println ("Unapply method : " + unapply("Zara@gmail.com"));

println ("Unapply method : " + unapply("Zara Ali"));

}

// The injection method (optional)

def apply(user: String, domain: String) = {

user +"@"+ domain

}

// The extraction method (mandatory)

def unapply(str: String): Option[(String, String)] = {

val parts = str split "@"

if (parts.length == 2){

Some(parts(0), parts(1))

}else{

None

}

}

}

这个对象定义了 apply 和unapply 方法。该apply 方法具有相同的含义：它原来的测试为可以被应用到的参数在括号中的方法所应用的相同的方式的对象。所以，可以写为Test("Zara", "gmail.com") 来构造字符串“Zara@gmail.com”。

unapply方法使测试类成为一个提取器并反转应用的构造过程。应用需要两个字符串，并形成了一个电子邮件地址以找到它们，非应用unapply需要一个电子邮件地址，并可能返回两个字符串：用户和地址的域名。

unapply还必须处理中给定的字符串不是一个电子邮件地址的情况。这就是为什么unapply返回一个选项型过对字符串。其结果要么是一些(用户域) 如果字符串str使用给定电子邮件地址的用户和域的部分，或None，如果str不是一个电子邮件地址。下面是一些例子：

unapply("Zara@gmail.com") equals Some("Zara", "gmail.com")

unapply("Zara Ali") equals None

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

Apply method : Zara@gmail.com

Unapply method : Some((Zara,gmail.com))

Unapply method : None

C:/>

**模式匹配使用提取器：**

当一个类的实例后跟括号使用零个或多个参数的列表，所述编译器调用应用的方法在该实例上。我们可以定义同时适用对象和类。

如上述所提到的，unapply方法的目的是提取我们寻找一个特定的值。它相反的操作和apply一样。当比较使用匹配语句中unapply方法的提取对象将被自动执行，如下所示：

object Test {

def main(args: Array[String]) {

val x = Test(5)

println(x)

x match

{

case Test(num) => println(x+" is bigger two times than "+num)

//unapply is invoked

case \_ => println("i cannot calculate")

}

}

def apply(x: Int) = x\*2

def unapply(z: Int): Option[Int] = if (z%2==0) Some(z/2) else None

}

让我们编译和运行上面的程序，这将产生以下结果：

C:/>scalac Test.scala

C:/>scala Test

10

10 is bigger two times than 5

C:/>

# 文件I/O

Scala打开文件是利用Java对象和java.io.File，它们都可在Scala编程中用来读取和写入文件。以下是写入文件的一个例子：

import java.io.\_

object Test {

def main(args: Array[String]) {

val writer = new PrintWriter(new File("test.txt" ))

writer.write("Hello Scala")

writer.close()

}

}

当上面的代码被编译和执行，它会创建一个有“Hello Scala”内容的文件。

C:/>scalac Test.scala

C:/>scala Test

C:/>

**从屏幕读取一行：**

有时需要从屏幕上读取用户输入，然后进行某些进一步的处理。下面的例子说明了如何从屏幕上读取输入：

object Test {

def main(args: Array[String]) {

print("Please enter your input : " )

val line = Console.readLine

println("Thanks, you just typed: " + line)

}

}

当上面的代码被编译和执行，它会提示输入内容，并继续进行，直到按ENTER(回车)键。

C:/>scalac Test.scala

C:/>scala Test

scala Test

Please enter your input : Scala is great

Thanks, you just typed: Scala is great

C:/>

**读取文件内容：**

从文件中读取是非常简单的。可以使用Scala的Source 类和它配套对象读取文件。以下是这些显示如何从之前创建“test.txt”文件中读取内容的示例：

import scala.io.Source

object Test {

def main(args: Array[String]) {

println("Following is the content read:" )

Source.fromFile("test.txt" ).foreach{

print

}

}

}

当上述代码被编译和执行时，它将读取test.txt文件并在屏幕上显示内容：

C:/>scalac Test.scala

C:/>scala Test

scala Test

Following is the content read:

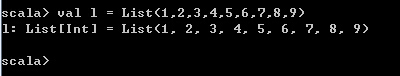
Hello Scala

C:/>

# 高阶函数

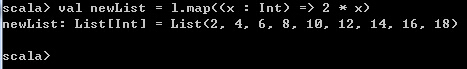
常用的高阶函数有 list、map、filter、reduce 等

声明一个 List 集合实例:



List 集合所在的包已经被预定义自动导入,所以此处不需要导入包,同时,这里直接使 用 List 实例化对象,其实是用来 List 的 object 对象的 apply 方法;

我们使用 map 函数把 List 中的每个值都乘以 2:



在上面的代码中,x 表示 l 中每个一个元素,map 对 l 中的每一个元素进行遍历操作, 由于 List 中只有一种类型的元素,所以我们在执行 map 操作的时候可以省略掉其类型,如 下所示:



List 集合中只有一个参数的时候,我们可以去掉参数中的括号:



在只有一个参数的情况下,更简洁和正常的写法如下所示:

常用



### 集合

集合主要有 List、Set、Tuple、Map 等