# Scala 简介

Scala 是 Scalable Language 的简写，是一门多范式的编程语言

联邦理工学院洛桑（EPFL）的Martin Odersky于2001年基于Funnel的工作开始设计Scala。

Funnel是把函数式编程思想和Petri网相结合的一种编程语言。

Odersky先前的工作是Generic Java和javac（Sun Java编译器）。Java平台的Scala于2003年底/2004年初发布。.NET平台的Scala发布于2004年6月。该语言第二个版本，v2.0，发布于2006年3月。

截至2009年9月，最新版本是版本2.7.6 。Scala 2.8预计的特性包括重写的Scala类库（Scala collections library）、方法的命名参数和默认参数、包对象（package object），以及Continuation。

2009年4月，Twitter宣布他们已经把大部分后端程序从Ruby迁移到Scala，其余部分也打算要迁移。此外， Wattzon已经公开宣称，其整个平台都已经是基于Scala基础设施编写的。

## Scala 特性

### **面向对象特性**

Scala是一种纯面向对象的语言，每个值都是对象。对象的数据类型以及行为由类和特质描述。

类抽象机制的扩展有两种途径：一种途径是子类继承，另一种途径是灵活的混入机制。这两种途径能避免多重继承的种种问题。

### **函数式编程**

Scala也是一种函数式语言，其函数也能当成值来使用。Scala提供了轻量级的语法用以定义匿名函数，支持高阶函数，允许嵌套多层函数，并支持柯里化。Scala的case class及其内置的模式匹配相当于函数式编程语言中常用的代数类型。

更进一步，程序员可以利用Scala的模式匹配，编写类似正则表达式的代码处理XML数据。

### **静态类型**

Scala具备类型系统，通过编译时检查，保证代码的安全性和一致性。类型系统具体支持以下特性：

* 泛型类
* 协变和逆变
* 标注
* 类型参数的上下限约束
* 把类别和抽象类型作为对象成员
* 复合类型
* 引用自己时显式指定类型
* 视图
* 多态方法

### **扩展性**

Scala的设计秉承一项事实，即在实践中，某个领域特定的应用程序开发往往需要特定于该领域的语言扩展。Scala提供了许多独特的语言机制，可以以库的形式轻易无缝添加新的语言结构：

* 任何方法可用作前缀或后缀操作符
* 可以根据预期类型自动构造闭包。

### **并发性**

Scala使用Actor作为其并发模型，Actor是类似线程的实体，通过邮箱发收消息。Actor可以复用线程，因此可以在程序中可以使用数百万个Actor,而线程只能创建数千个。在2.10之后的版本中，使用Akka作为其默认Actor实现。

## Scala 安装

Scala是基于java之上，大量使用java的类库和变量，必须使用Scala之前必须先安装 Java（>1.5版本）。

1 。设置 SCALA\_HOME 变量：单击新建，在变量名栏输入：**SCALA\_HOME**: 变量值一栏输入：**D:\Program Files\scala**也就是scala的安装目录，根据个人情况有所不同，如果安装在C盘，将"D"改成"C"即可。

2.设置 Classpath 变量：找到找到系统变量下的"Classpath"如图，单击编辑，如没有，则单击"新建":

* "变量名"：ClassPath
* "变量值"：.;%SCALA\_HOME%\bin;%SCALA\_HOME%\lib\dt.jar;%SCALA\_HOME%\lib\tools.jar.;

# Scala 基础语法

Scala 与 Java 的最大区别是：Scala 语句末尾的分号 ; 是可选的。

我们可以认为 Scala 程序是对象的集合，通过调用彼此的方法来实现消息传递。接下来我们来理解下，类，对象，方法，实例变量的概念：

## 第一个 Scala 程序

### **交互式编程**

交互式编程不需要创建脚本文件，可以通过以下命令调用：

$ scalaWelcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0\_31).Type in expressions to have them evaluated.Type :help for more information.

scala> 1 + 1

res0: Int = 2

scala> println("Hello World!")Hello World!

scala>

## 脚本形式

我们也可以通过创建一个 HelloWorld.scala 的文件来执行代码，HelloWorld.scala 代码如下所示：

object HelloWorld {

/\* 这是我的第一个 Scala 程序

\* 以下程序将输出'Hello World!'

\*/

def main(args: Array[String]) {

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

}}

接下来我们使用 scalac 命令编译它：

$ scalac HelloWorld.scala

$ lsHelloWorld$.class HelloWorld.scalaHelloWorld.class

编译后我们可以看到目录下生成了 HelloWorld.class 文件，该文件可以在Java Virtual Machine (JVM)上运行。

编译后，我们可以使用以下命令来执行程序：

$ scala HelloWorldHello, world!

## 基本语法

Scala 基本语法需要注意以下几点：

**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。

## 标识符

<-为值的提取符

Scala 可以使用两种形式的标志符，字符数字和符号。

字符数字使用字母或是下划线开头，后面可以接字母或是数字，符号"$"在 Scala 中也看作为字母。然而以"$"开头的标识符为保留的 Scala 编译器产生的标志符使用，应用程序应该避免使用"$"开始的标识符，以免造成冲突。

Scala 的命名规则采用和 Java 类似的 camel 命名规则，首字符小写，比如 toString。类名的首字符还是使用大写。此外也应该避免使用以下划线结尾的标志符以避免冲突。符号标志符包含一个或多个符号，如+，:，? 等，比如:

+ ++ ::: < ?> :->

Scala 内部实现时会使用转义的标志符，比如:-> 使用 $colon$minus$greater 来表示这个符号。因此如果你需要在 Java 代码中访问:->方法，你需要使用 Scala 的内部名称 $colon$minus$greater。

混合标志符由字符数字标志符后面跟着一个或多个符号组成，比如 unary\_+ 为 Scala 对+方法的内部实现时的名称。字面量标志符为使用"定义的字符串，比如 `x` `yield`。

你可以在"之间使用任何有效的 Scala 标志符，Scala 将它们解释为一个 Scala 标志符，一个典型的使用为 Thread 的 yield 方法， 在 Scala 中你不能使用 Thread.yield()是因为 yield 为 Scala 中的关键字， 你必须使用 Thread.`yield`()来使用这个方法。

## Scala 关键字

下表列出了 scala 保留关键字，我们不能使用以下关键字作为变量：

|  |  |  |  |
| --- | --- | --- | --- |
| 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 注释

Scala 类似 Java 支持单行很多行注释。多行注释可以嵌套，但必须正确嵌套，一个注释开始符号对应一个结束符号。注释在 Scala 编译中会被忽略，实例如下：

object HelloWorld {

/\* 这是一个 Scala 程序

\* 这是一行注释

\* 这里演示了多行注释

\*/

def main(args: Array[String]) {

// 输出 Hello World

// 这是一个单行注释

println("Hello, world!")

}}

## 空行和空格

一行中只有空格或者带有注释，Scala 会认为其是空行，会忽略它。标记可以被空格或者注释来分割。

## 换行符

Scala是面向行的语言，语句可以用分号（;）结束或换行符。Scala 程序里,语句末尾的分号通常是可选的。如果你愿意可以输入一个,但若一行里仅 有一个语句也可不写。另一方面,如果一行里写多个语句那么分号是需要的。例如

val s = "菜鸟教程"; println(s)

## Scala 包

### **定义包**

Scala 使用 package 关键字定义包，在Scala将代码定义到某个包中有两种方式：

第一种方法和 Java 一样，在文件的头定义包名，这种方法就后续所有代码都放在该报中。 比如：

package com.runoobclass HelloWorld

第二种方法有些类似 C#，如：

package com.runoob {

class HelloWorld }

第二种方法，可以在一个文件中定义多个包。

### **引用**

Scala 使用 import 关键字引用包。

import java.awt.Color // 引入Color

import java.awt.\_ // 引入包内所有成员

def handler(evt: event.ActionEvent) { // java.awt.event.ActionEvent

... // 因为引入了java.awt，所以可以省去前面的部分}

import语句可以出现在任何地方，而不是只能在文件顶部。import的效果从开始延伸到语句块的结束。这可以大幅减少名称冲突的可能性。

如果想要引入包中的几个成员，可以使用selector（选取器）：

import java.awt.{Color, Font}

// 重命名成员import java.util.{HashMap => JavaHashMap}

// 隐藏成员import java.util.{HashMap => \_, \_} // 引入了util包的所有成员，但是HashMap被隐藏了

***注意：****默认情况下，Scala 总会引入 java.lang.\_ 、 scala.\_ 和 Predef.\_，这里也能解释，为什么以scala开头的包，在使用时都是省去scala.的。*

# Scala 数据类型

## 数据类型

Scala 与 Java有着相同的数据类型，下表列出了 Scala 支持的数据类型：

|  |  |
| --- | --- |
| **数据类型** | **描述** |
| Byte | 8位有符号补码整数。数值区间为 -128 到 127 |
| Short | 16位有符号补码整数。数值区间为 -32768 到 32767 |
| Int | 32位有符号补码整数。数值区间为 -2147483648 到 2147483647 |
| Long | 64位有符号补码整数。数值区间为 -9223372036854775808 到 9223372036854775807 |
| Float | 32位IEEE754单精度浮点数 |
| Double | 64位IEEE754单精度浮点数 |
| Char | 16位无符号Unicode字符, 区间值为 U+0000 到 U+FFFF |
| String | 字符序列 |
| Boolean | true或false |
| Unit | 表示无值，和其他语言中void等同。用作不返回任何结果的方法的结果类型。  Unit只有一个实例值，写成()。 |
| Null | null 或空引用 |
| Nothing | Nothing类型在Scala的类层级的最低端；它是任何其他类型的子类型。 |
| Any | Any是所有其他类的超类 |
| AnyRef | AnyRef类是Scala里所有引用类(reference class)的基类 |

上表中列出的数据类型都是对象，也就是说scala没有java中的原生类型。在scala是可以对数字等基础类型调用方法的

Scala 基础字面量

Scala 非常简单且直观。接下来我们会详细介绍 Scala 字面量。

### **整型字面量**

整型字面量用于 Int 类型，如果表示 Long，可以在数字后面添加 L 或者小写 l 作为后缀。：

### **浮点型字面量**

如果浮点数后面有f或者F后缀时，表示这是一个Float类型，否则就是一个Double类型的。实例如下：

### **布尔型字面量**

布尔型字面量有 true 和 false。

### **符号字面量**

符号字面量被写成： **'<标识符>** ，这里 **<标识符>** 可以是任何字母或数字的标识（注意：不能以数字开头）。这种字面量被映射成预定义类scala.Symbol的实例。

如： 符号字面量 **'x** 是表达式 **scala.Symbol("x")** 的简写，符号字面量定义如下：

package scala

final case class Symbol private (name: String) {

override def toString: String = "'" + name

}

### **字符字面量**

在scala中字符类型表示为半角单引号 ’’中的字符，与java相同如下：

其中 **\** 表示转移字符，其后可以跟 **u0041** 数字或者 **\r\n** 等固定的转义字符。

### **字符串字面量**

字符串表示方法是在双引号中””，与java相同包含一系列字符，如：

### **多行字符串的表示方法**

多行字符串用三个双引号来表示分隔符，格式为：**""" ... """**。

### **Null 值**

空值是 scala.Null 类型。

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

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

### **Scala 转义字符**

下表列出了常见的转义字符：

|  |  |  |
| --- | --- | --- |
| **转义字符** | **Unicode** | **描述** |
| \b | \u0008 | 退格(BS) ，将当前位置移到前一列 |
| \t | \u0009 | 水平制表(HT) （跳到下一个TAB位置） |
| \n | \u000c | 换行(LF) ，将当前位置移到下一行开头 |
| \f | \u000c | 换页(FF)，将当前位置移到下页开头 |
| \r | \u000d | 回车(CR) ，将当前位置移到本行开头 |
| \" | \u0022 | 代表一个双引号(")字符 |
| \' | \u0027 | 代表一个单引号（'）字符 |
| \\ | \u005c | 代表一个反斜线字符 '\' |

0 到 255 间的 Unicode 字符可以用一个八进制转义序列来表示，即反斜线‟\‟后跟 最多三个八进制。

在字符或字符串中，反斜线和后面的字符序列不能构成一个合法的转义序列将会导致 编译错误。

## Scala 变量

变量是一种使用方便的占位符，用于引用计算机内存地址，变量创建后会占用一定的内存空间。

基于变量的数据类型，操作系统会进行内存分配并且决定什么将被储存在保留内存中。因此，通过给变量分配不同的数据类型，你可以在这些变量中存储整数，小数或者字字母。

在 Scala 中，使用关键词 **"var"** 声明变量，使用关键词 **"val"** 声明常量。

Val生命不可变值得战略意义：

1. scala函数式编程中要求值不可变，val天然符合这一要求
2. 在分布式系统中，一般都要求值不可变，这样才能简化分布式系统的设计和实现，同时拥有更高的效率，val声明的内容都不可变，所以不用担心在机器之间传递数据的时候改变修改内容，这位分布式编程打下非常好的基础，包括spark的核心rdd也是补课改变的。

声明变量实例如下：

var myVar : String = "Foo"

var myVar : String = "Too"

以上定义了变量 myVar，我们可以修改它。

声明常量实例如下：

val myVal : String = "Foo"

以上定义了常量 myVal，它是不能修改的。如果程序尝试修改常量 myVal 的值，程序将会在编译时报错

### **变量类型声明**

变量的类型在变量名之后等号之前声明。定义变量的类型的语法格式如下：

var VariableName : DataType [= Initial Value]

val VariableName : DataType [= Initial Value]

变量声明不一定需要初始值，以下也是正确的：

var myVar :Int;

val myVal :String;

### 变量类型引用

在 Scala 中声明变量和常量不一定要指明数据类型，在没有指明数据类型的情况下，其数据类型是通过变量或常量的初始值推断出来的。

所以，如果在没有指明数据类型的情况下声明变量或常量必须要给出其初始值，否则将会报错。

var myVar = 10;

val myVal = "Hello, Scala!";

以上实例中，myVar 会被推断为 Int 类型，myVal 会被推断为 String 类型。

### **Scala 多个变量声明**

Scala 支持多个变量的声明：

val xmax, ymax = 100 // xmax, ymax都声明为100

如果方法返回值是元组，我们可以使用 val 来声明一个元组：

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

也可以不指定数据类型：

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

## Scala 访问修饰符

Scala 访问修饰符基本和Java的一样，分别有：private，protected，public。

如果没有指定访问修饰符符，默认情况下，Scala对象的访问级别都是 public。

Scala 中的 private 限定符，比 Java 更严格，在嵌套类情况下，外层类甚至不能访问被嵌套类的私有成员。

### 私有(Private)成员

用private关键字修饰，带有此标记的成员仅在包含了成员定义的类或对象内部可见，同样的规则还适用内部类。

class Outer{

class Inner{

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

class InnerMost{

f() // 正确

}

}

(new Inner).f() //错误}

(new Inner).f( ) 访问不合法是因为 f 在 Inner 中被声明为 private，而访问不在类Inner之内。

但在 InnerMost 里访问f就没有问题的，因为这个访问包含在 Inner 类之内。

Java中允许这两种访问，因为它允许外部类访问内部类的私有成员。

### 保护(Protected)成员

在 scala 中，对保护（Protected）成员的访问比 java 更严格一些。因为它只允许保护成员在定义了该成员的的类的子类中被访问。而在java中，用protected关键字修饰的成员，除了定义了该成员的类的子类可以访问，同一个包里的其他类也可以进行访问。

package p{class Super{

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

}

class Sub extends Super{

f()

}

class Other{

(new Super).f() //错误

}}

上例中，Sub 类对 f 的访问没有问题，因为 f 在 Super 中被声明为 protected，而 Sub 是 Super 的子类。相反，Other 对 f 的访问不被允许，因为 other 没有继承自 Super。而后者在 java 里同样被认可，因为 Other 与 Sub 在同一包里

### 公共(Public)成员

Scala中，如果没有指定任何的修饰符，则默认为 public。这样的成员在任何地方都可以被访问。

class Outer {

class Inner {

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

class InnerMost {

f() // 正确

}

}

(new Inner).f() // 正确因为 f() 是 public}

### 作用域保护

Scala中，访问修饰符可以通过使用限定词强调。格式为:

private[x]

或

protected[x]

这里的x指代某个所属的包、类或单例对象。如果写成private[x],读作"这个成员除了对[…]中的类或[…]中的包中的类及它们的伴生对像可见外，对其它所有类都是private。

这种技巧在横跨了若干包的大型项目中非常有用，它允许你定义一些在你项目的若干子包中可见但对于项目外部的客户却始终不可见的东西。

package bobsrocckets{

package navigation{

private[bobsrockets] class Navigator{

protected[navigation] def useStarChart(){}

class LegOfJourney{

private[Navigator] val distance = 100

}

private[this] var speed = 200

}

}

package launch{

import navigation.\_

object Vehicle{

private[launch] val guide = new Navigator

}

}}

上述例子中，类Navigator被标记为private[bobsrockets]就是说这个类对包含在bobsrockets包里的所有的类和对象可见。

比如说，从Vehicle对象里对Navigator的访问是被允许的，因为对象Vehicle包含在包launch中，而launch包在bobsrockets中，相反，所有在包bobsrockets之外的代码都不能访问类Navigator。

## 运算符优先级

在一个表达式中可能包含多个有不同运算符连接起来的、具有不同数据类型的数据对象；由于表达式有多种运算，不同的运算顺序可能得出不同结果甚至出现错误运算错误，因为当表达式中含多种运算时，必须按一定顺序进行结合，才能保证运算的合理性和结果的正确性、唯一性。

优先级从上到下依次递减，最上面具有最高的优先级，逗号操作符具有最低的优先级。

相同优先级中，按结合顺序计算。大多数运算是从左至右计算，只有三个优先级是从右至左结合的，它们是单目运算符、条件运算符、赋值运算符。

基本的优先级需要记住：

* 指针最优，单目运算优于双目运算。如正负号。
* 先乘除（模），后加减。
* 先算术运算，后移位运算，最后位运算。请特别注意：1 << 3 + 2 & 7 等价于 (1 << (3 + 2))&7
* 逻辑运算最后计算

# Scala 函数

## 函数介绍

函数是一组一起执行一个任务的语句。 您可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由您来决定的，但在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。

Scala 有函数和方法，二者在语义上的区别很小。Scala 方法是类的一部分，而函数是一个对象可以赋值给一个变量。换句话来说在类中定义的函数即是方法。

我们可以在任何地方定义函数，甚至可以在函数内定义函数（内嵌函数）。更重要的一点是 Scala 函数名可以由以下特殊字符：**+, ++, ~, &,-, -- , \, /, :** 等。

关于println打印出内容到控制台，底层借助了java IO的功能，一个事实情况是Scala在做很多比较底层的实现，使用java的实现来缩短开发时间，例如说操作数据（DB.NoSql(cassandra,Hbase)等）的JDBC，再例如关于线程Thread的操作，scala会直接使用java中的Thread。

按照OS的原理，程序的main入口方法都是运行在主线程中的，OS 的运行分为kernel space和User space，普通的应用程序都运行在User space中，应用程序的scala所在的进程一般都是透过OS Fork出来的，被fork出来的应用程序默认会有主线程，而main方法就是在主线程中的。

### 函数声明

Scala 函数声明格式如下：

def functionName ([参数列表]) : [return type]

如果你不写等于号和方法主体，那么方法会被隐式声明为"抽象(abstract)"，包含它的类型于是也是一个抽象类型。

### 函数定义

方法定义由一个def 关键字开始，紧接着是可选的参数列表，一个冒号"：" 和方法的返回类型，一个等于号"="，最后是方法的主体。

方法名和方法体之间没有返回值类型,则默认为Unit

方法和方法体之间没有= ，则方法的返回值也是Unit，打印出来为（）

### 参数

函数的参数可以有默认值def addInt( a:Int, b:Int=30)

也可以在调用方法的时候给参数名传值 addInt(a=20,b=40),因为函数背后就是类和对象。

***可变参数：***

***可变参数的数据在被收集时会被变成数组***

***1 to 10 的到的类型为range，不能传递参数，应该写成***1 to 1000: \_\****形式，***

***将range转变成参数序列***

|  |
| --- |
| *print*(*sum*(1 to 1000: \_\*))  **def** sum(numbers:Int\*)={  **var** result =0  **for**(number<-numbers)result+=number  result } |

Scala 函数定义格式如下：

def functionName ([参数列表]) : [return type] = {

function body

return [expr]}

以上代码中 **return type** 可以是任意合法的 Scala 数据类型。参数列表中的参数可以使用逗号分隔。

以下函数的功能是将两个传入的参数相加并求和：

object add{

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

var sum:Int = 0

sum = a + b

return sum

}}

如果函数没有返回值，可以返回为 **Unit**，这个类似于 Java 的 **void**, 实例如下：

object Hello{

def printMe( ) : Unit = {

println("Hello, Scala!")

}}

### 函数调用

Scala 提供了多种不同的函数调用方式：

以下是调用方法的标准格式：

functionName( 参数列表 )

如果函数使用了实例的对象来调用，我们可以使用类似java的格式 (使用 **.** 号)：

[instance.]functionName( 参数列表 )

以上实例演示了定义与调用函数的实例:

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

}}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala TestReturned Value : 12

解 Scala 编程：

|  |  |  |
| --- | --- | --- |
| **函数概念解析接案例** | |  |
| [函数传名调用(Call-by-Name)](http://www.runoob.com/scala/functions-call-by-name.html) | [指定函数参数名](http://www.runoob.com/scala/functions-named-arguments.html) |  |
| [函数 - 可变参数](http://www.runoob.com/scala/functions-variable-arguments.html) | [递归函数](http://www.runoob.com/scala/recursion-functions.html) |  |
| [默认参数值](http://www.runoob.com/scala/functions-default-parameter-values.html) | [高阶函数](http://www.runoob.com/scala/higher-order-functions.html) |  |
| [内嵌函数](http://www.runoob.com/scala/nested-functions.html) | [匿名函数](http://www.runoob.com/scala/anonymous-functions.html) |  |
| [偏应用函数](http://www.runoob.com/scala/partially-applied-functions.html) | [函数柯里化(Function Currying)](http://www.runoob.com/scala/currying-functions.html) |  |

### \*函数特性

在scala中函数事一等公民，可以向变量一样被传递，被赋值，同时函数可以赋值给变量，变量页可以赋值给函数，之所有可以这样，原因在于函数背后是类和对象，也就是说在运行的时候函数其实就是一个变量！！！当然，背后的类是scala语言自动帮我们生成的，切可以天然的许列化和反序列化，这个意义非常重要：

意义1：可以天然的序列化和反序列化的直接好处就是函数可以在分布式系统上传递！！！

意义2：因为函数背后其实是类和对象，所以可以和普通的变量完全一样的应用在任何普通变量可以运行的地方，包括参数传递，作为返回值，被变量赋值和赋值给变量。

函数的参数可以是函数

|  |
| --- |
| *//函数可以直接赋值给变量* **val** hiData=*bigData* \_ *println*(hiData(**"spark"**)) *//匿名函数，只需要函数参数和函数体，不要名称，一般吧匿名函数赋值给变量（其实是val常量）* **val** hiBig=(name:String)=>*println*(**"hi,"**+name) *println*(hiBig(**"kafka"**)) *//直接将函数作为参数传递给函数 //在java中是new一个接口实例，在借口实例的回调方法calback中来实现业务逻辑* **def** getName(func:(String)=>Unit,name:String): Unit ={  func(name) } getName(hiBig,**"Scala"**)  **val** mularray=*Array*(1 to 10: \_\*).map((item:Int)=>item\*2).foreach(x=>*println*(x)) *//当函数的返回类型是函数的时候，这个时候就表明了scala的函数实现了闭包， //闭包原理：scala函数后面是列和对象，所以，scala的参数都作为了对象的成员！！！！，所以可以后续访问* **def** funResult(message:String)=(name:String)=>*println*(message+**":"**+name) funResult(**"hello"**)(**"java"**)*//Curring函数写法 //上面的调用等同于下面两句* **val** result=funResult(**"hello"**) result(**"java"**) |

## Scala 闭包

闭包是一个函数，返回值依赖于声明在函数外部的一个或多个变量。

闭包通常来讲可以简单的认为是可以访问一个函数里面局部变量的另外一个函数。

如下面这段匿名的函数：

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

函数体内有一个变量 i，它作为函数的一个参数。如下面的另一段代码：

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

在 multiplier 中有两个变量：i 和 factor。其中的一个 i 是函数的形式参数，在 multiplier 函数被调用时，i 被赋予一个新的值。然而，factor不是形式参数，而是自由变量，考虑下面代码：

var factor = 3

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

这里我们引入一个自由变量 factor，这个变量定义在函数外面。

这样定义的函数变量 multiplier 成为一个"闭包"，因为它引用到函数外面定义的变量，定义这个函数的过程是将这个自由变量捕获而构成一个封闭的函数。

完整实例

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

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

muliplier(1) value = 3

muliplier(2) value = 6

# 控制语句

## If else

1 、Scala中的if条件表达式是可以有值的！！！，这和java不一样

2 、if条件表达式可以进行类型推到，类型推到的一般过程就是根据变量的值的类型来推到确定变量的类型，这在很多复杂算法实现的时候可以让我们省略变量的类型书写，为复杂算法的实现提供了非常大的便利。

3 、如果if后面没有else部分，默认的实现是if(...)... Else()

在很多情况下。Else部分虽然可以不写，但是返回的类型会是any，所以仍旧不会省略else，使用else{None}这种方式其目的是为了确保if条件表达式的类型为处理逻辑需要的类型Option,而不是any类型

4 、if表达式中的{}中有多条语句时，只有最后一条是计算结果

5 、如果表达式中没有值可以赋，则result的值为Unit

6 、补充说明：{。。。}代表了一个语句块，语句块是有值的，值就是最后一条语句的类型

|  |
| --- |
| **var** age = 10; *//当else前后的值类型相同，则result的值与其相同* **var** result = **if** (age > 25) **"gao" else "jian"** *//当else两边的类型不一样的时候，result的类型为any* **var** result2 = **if** (age > 18) **"adlt" else** 1 *//当没有else是相当于 if(age>18) "a" else (),result类型为any* **var** result3 = **if** (age>18) **"a"**  *//值为0 2 4 if在这里限制结果*  **for**(i <- 0 to 5 **if** i%2==0){  *println*(i) } |

## For，while

<-为值的提取符，to是一个方法,if是有值的，所以可以放在for等控制结构中用于限制结果

在for循环中能提取出什么内容取决于后面的集合类型

除了if条件守卫还能用return停止循环（return为方法级别的），for和while中不能用break

，一般都用标记位flag标记来跳出循环，break应该与breakable配合使用

|  |
| --- |
| *//输出为15，for循环中加入if叫做条件守卫，用于限制for循环（优化for循环，去掉不必要的执行步骤，或者说跳出for循环）* **for**(i<-0 to 6 **if** flag){  sum=sum+i  **if**(5==i)flag=**false** }  *//while循环（用的不多）*  **import** scala.util.control.Breaks.\_ **var** flag=**true** breakable{  **while**(flag){  **for**(item<-**"spark"**){  *println*(item)  **if**(item==**'r'**){  flag=**false** break  }  }  } }  *//双层循环* **for**(i<- 1 to 3;j<- 1 to 3)*println*(i\*j) |

## Scala 模式匹配

Scala 提供了强大的模式匹配机制，应用也非常广泛。

一个模式匹配包含了一系列备选项，每个都开始于关键字 **case**。每个备选项都包含了一个模式及一到多个表达式。箭头符号 **=>** 隔开了模式和表达式。

match 对应 Java 里的 switch，但是写在选择器表达式之后。即： **选择器 match {备选项}。**

match 表达式通过以代码编写的先后次序尝试每个模式来完成计算，只要发现有一个匹配的case，剩下的case不会继续匹配。

|  |
| --- |
| **class** DataFrameWork **case class** ComputationFramework(name:String,popular:Boolean)**extends** DataFrameWork **case class** StorageFramework(name:String,popular:Boolean)**extends** DataFrameWork **object** PatternMatch {  **def** main(args:Array[String]): Unit ={  *getSalary*(**"gdsfgf"**,10)  *getMatchType*(100.1f)  *getMatchColl*(*Array*[String](**"java"**,**"python"**,**"cc"**))  *getBigDataType*(ComputationFramework(**"gao"**,**true**))  *getValue*(**"spark"**,*Map*((**"spark"**,**"gao"**),(**"hadoop"**,**"jian"**)))  }  *//case class匹配* **def** getBigDataType(data:DataFrameWork){  data **match** {  **case** *ComputationFramework*(name,popular)=>*println*(**"ComputationFramework---name:"**+name+**",popular:"**+popular)  **case** *StorageFramework*(name,popular)=>*println*(**"StorageFramework---name:"**+name+**",popular:"**+popular)  **case** \_=>*println*(**"Unkown Type"**)  }  }  *//值匹配* **def** getSalary(name:String,age:Int): Unit ={  name **match**{  **case "spark"**=>*println*(**"$150000"**)  **case "hadoop"**=>*println*(**"$90000"**)  **case** \_ **if** name==**"scala"**=>*println*(**"$140000"**)  **case** \_ **if** name==**"mapreduce"**=>*println*(**"$70000"**)  **case** \_name **if** age>=15=>*println*(**"name:"**+\_name+**",age:"**+age+**",$20000"**)  **case** \_ =>*println*(**"$10000"**)  }  }  *//类型匹配* **def** getMatchType(msg:Any){  msg **match**{  **case** i:Int=>*println*(**"Integer"**)  **case** s:String=>*println*(**"String"**)  **case** d:Double=>*println*(**"Double"**)  **case** \_ =>*println*(**"Ubkown type"**)  }  }  *//集合匹配* **def** getMatchColl(msg:Array[String]){  msg **match**{  **case** *Array*(**"scala"**)=>*println*(**"scala"**)  **case** *Array*(**"java"**,**"python"**)=>*println*(**"java"**)  **case** *Array*(**"spark"**,\_\*)=>*println*(**"spark"**)  **case** \_ =>*println*(**"Ubkown type"**)  }  }  *//选项匹配* **def** getValue(key:String,content:Map[String,String]){  content.get(key) **match** {  **case** *Some*(value)=>*println*(value)  **case** None=>*println*(**"not fond!"**)  }  } } |

### 使用样例类

使用了case关键字的类定义就是就是样例类(case classes)，样例类是种特殊的类，经过优化以用于模式匹配。

在声明样例类时，下面的过程自动发生了：

* 构造器的每个参数都成为val，除非显式被声明为var，但是并不推荐这么做；
* 在伴生对象中提供了apply方法，所以可以不使用new关键字就可构建对象；
* 提供unapply方法使模式匹配可以工作；
* 生成toString、equals、hashCode和copy方法，除非显示给出这些方法的定义。

# Scala 集合

## \*数组笔记

### 1 数组Array

*1 最原始数组创建方式：val array=new Array[Int](5),指定数组长度为固定5个长度  
\* 2 最常用和经典访问的创建数组的方式形如Array[Int](1,2,3,4,5),直接通过Array类名并传入参数，这种方式的数组长度是不可变的，在背后的实现是调用Array的  
\* 工厂方法模式apply来构建数组和数组内容的  
\* 3 对数组元素访问的时候下标范围在0到length-1长度，超过会包异常*

***4 如果想在已经有的数组的基础上通过作用于每个元素来生成新元素构成的新数组，则可以用yield完成，这在大数据中意义重大：***

***第一点：在不修改已经有的array的基础上完成的非常适合与大数据的处理***

***第二点：在大数据处理中，例如spark操作中，业务操纵的核心思想就是yield，来通过function对每个元素操作来获取新的元素集合，尤其是新的RDD，例如MapPartitionsRDD***

***提供初始值构造数组时不要用new***

***用array（1）来访问数组元素***

|  |
| --- |
| *//传统方式* **val** array1=**new** Array[Int](5) *//通过类名+参数，可以去掉泛型是cala有类型推倒能力，可以根据值推倒出类型，长度不可变* **val** array2=*Array*[Int](1,2,3) *//同上，调用类的工厂方法* **val** array=Array.*apply*(1,2,3,4,5)  **for**(i<-0 until array.length)*print*(array(i)+**" "**) *//每次打印下标+2个step才打印* **for**(i<-0 until (array.length,2))*print*(array(i)+**" "**) *//翻转下标，打印* **for**(i<-(0 until array.length).reverse)*print*(array(i)+**" "**) *//求和 print*(**"sum:"**+array.sum) *//对数组进行升序排序* scala.util.Sorting.*quickSort*(array) *//打印结果为：quiksort1:1,2,3,4,5,6,7 print*(**"quiksort1:"**+array.mkString(**","**)) *//打印结果为：quiksort2:\*\*\*\*\*1,2,3,4,5,6,7\*\*\*\*\* print*(**"quiksort2:"**+array.mkString(**"\*\*\*\*\*"**,**","**,**"\*\*\*\*\*"**)) *//sparkmap函数的原型，用yield产生新的元素，产生为向量* **val** arrayAddedOne=**for**(item<-array) **yield** item +1 *println*(arrayAddedOne.mkString(**" "**)) *//添加守卫* **val** arrayEven= **for**(item<-array **if** item%2==0)**yield** item+1 *println*(arrayEven.mkString(**" "**)) *println*(array.filter(x=>x%2==0).mkString(**" "**)) *//如果针对所欲元素可以用\_取代 println*(array.filter( \_%2==0).mkString(**" "**)) *//map是便利集合中的所有元素，对每个元素做处理 println*(array.filter( \_%2==0).map(\_\*10).mkString(**" "**)) |

### 2 变长数组ArrayBuffer

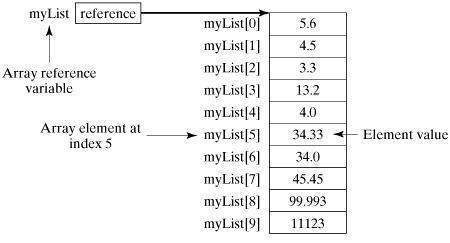
使用scala.Collection.javaConversions中的转换函数

*1 如果想使用可变数组要导入****scala.collection.mutable.ArrayBuffer****,用ArrayBuffer  
 2 ArrayBuffer增加元素默认是在末尾增加元素*

*3 在多线程操作时，吧arrayBuffer变成array很重要，虽然Array的元素不可删减，但是可以改变值，还需要考虑并发问题*

|  |
| --- |
| **import** scala.collection.mutable.ArrayBuffer **val** arrayBuffer=ArrayBuffer[Int]() arrayBuffer+=1 arrayBuffer+=2 arrayBuffer+=3 arrayBuffer+=(4,5,6,7,8) *//用两个++=来在末尾添加数组* arrayBuffer++=*Array*(10,11,12) *//在指定位置插入元素* arrayBuffer.insert(3,100,1000,10000) *//删除元素* arrayBuffer.remove(arrayBuffer.length-1) *//当需要多线程并发时，把arrayBuffer转换成Array，因为Array一般不可变* arrayBuffer.toArray  *//移除最后的2个元素*  arrayBuffer.trimEnd(2)  *//移除开始的2个元素*  arrayBuffer.trimStart(2)  *//数组转换成arraybuffer*  arr.toBuffer |

下图展示了一个长度为 10 的数组 myList，索引值为 0 到 9：



**多维数组**

多维数组一个数组中的值可以是另一个数组，另一个数组的值也可以是一个数组。矩阵与表格是我们常见的二维数组。

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

实例中数组中包含三个数组元素，每个数组元素又含有三个值。

**合并数组**

以下实例中，我们使用 concat() 方法来合并两个数组，concat() 方法中接受多个数组参数：

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)

// 输出所有数组元素

for ( x <- myList3 ) {

println( x )

}

}

}

### 创建区间数组

以下实例中，我们使用了 range() 方法来生成一个区间范围内的数组。range() 方法最后一个参数为步长，默认为 1：

|  |
| --- |
| **import** Array.\_ **object** Test {  **def** main(args: Array[String]) {  *//10到20,step为2*  **var** myList1 = *range*(10, 20, 2)  **var** myList2 = *range*(10,20)  *// 输出所有数组元素* **for** ( x <- myList1 ) {  *print*( **" "** + x )  }  *println*()  **for** ( x <- myList2 ) {  *print*( **" "** + x )  }  } } |

执行以上代码，输出结果为：

10 12 14 16 18

10 11 12 13 14 15 16 17 18 19

### 隐式转换

|  |
| --- |
| **import** scala.collection.JavaConversions.bufferAsJavaList **import** scala.collection.mutable.ArrayBuffer **val** com=ArrayBuffer(**"ls"**,**"-qa"**,**"/opt"**) *//scala转换为java* **val** pb=**new** ProcessBuilder(com) **import** scala.collection.JavaConversions.asScalaBuffer **import** scala.collection.mutable.Buffer *//java转换成buffer，不能用ArrayBuffer-包装起来的对象只能把奥正* |

### Scala 数组方法

下表中为 Scala 语言中处理数组的重要方法，使用它前我们需要使用 **import Array.\_** 引入包。

|  |  |
| --- | --- |
| **序号** | **方法和描述** |
| 1 | **def apply( x: T, xs: T\* ): Array[T]**  创建指定对象 T 的数组, T 的值可以是 Unit, Double, Float, Long, Int, Char, Short, Byte, Boolean。 |
| 2 | **def concat[T]( xss: Array[T]\* ): Array[T]**  合并数组 |
| 3 | **def copy( src: AnyRef, srcPos: Int, dest: AnyRef, destPos: Int, length: Int ): Unit**  复制一个数组到另一个数组上。相等于 Java's 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]**  返回指定长度数组，每个数组元素为指定函数的返回值。  以上实例数组初始值为 0，长度为 3，计算函数为**a=>a+1**：  scala> Array.iterate(0,3)(a=>a+1)  res1: Array[Int] = Array(0, 1, 2) |
| 6 | **def fill[T]( n: Int )(elem: => T): Array[T]**  返回数组，长度为第一个参数指定，同时每个元素使用第二个参数进行填充。 |
| 7 | **def fill[T]( n1: Int, n2: Int )( elem: => T ): Array[Array[T]]**  返回二数组，长度为第一个参数指定，同时每个元素使用第二个参数进行填充。 |
| 8 | **def ofDim[T]( n1: Int ): Array[T]**  创建指定长度的数组 |
| 9 | **def ofDim[T]( n1: Int, n2: Int ): Array[Array[T]]**  创建二维数组 |
| 10 | **def ofDim[T]( n1: Int, n2: Int, n3: Int ): Array[Array[Array[T]]]**  创建三维数组 |
| 11 | **def range( start: Int, end: Int, step: Int ): Array[Int]**  创建指定区间内的数组，step 为每个元素间的步长 |
| 12 | **def range( start: Int, end: Int ): Array[Int]**  创建指定区间内的数组 |
| 13 | **def tabulate[T]( n: Int )(f: (Int)=> T): Array[T]**  返回指定长度数组，每个数组元素为指定函数的返回值，默认从 0 开始。  以上实例返回 3 个元素：  scala> Array.tabulate(3)(a => a + 5)  res0: Array[Int] = Array(5, 6, 7) |
| 14 | **def tabulate[T]( n1: Int, n2: Int )( f: (Int, Int ) => T): Array[Array[T]]**  返回指定长度的二维数组，每个数组元素为指定函数的返回值，默认从 0 开始。 |

## Scala List(列表)

Scala 列表类似于数组，它们所有元素的类型都相同，但是它们也有所不同：列表是不可变的，值一旦被定义了就不能改变，其次列表 具有递归的结构（也就是链接表结构）而数组不是。。

列表的元素类型 T 可以写成 List[T]。例如，以下列出了多种类型的列表：

// 字符串列表

val site: List[String] = List("Runoob", "Google", "Baidu")

// 整型列表

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

// 空列表

val empty: List[Nothing] = List()

// 二维列表

val dim: List[List[Int]] =

List(

List(1, 0, 0),

List(0, 1, 0),

List(0, 0, 1)

)

构造列表的两个基本单位是 **Nil** 和 **::**

**Nil** 也可以表示为一个空列表。

以上实例我们可以写成如下所示：

// 字符串列表

val site = "Runoob" :: ("Google" :: ("Baidu" :: Nil))

// 整型列表

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

// 空列表

val empty = Nil

// 二维列表

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

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

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

### 列表基本操作

Scala列表有三个基本操作：

* head 返回列表第一个元素
* tail 返回一个列表，包含除了第一元素之外的其他元素
* isEmpty 在列表为空时返回true

对于Scala列表的任何操作都可以使用这三个基本操作来表达。实例如下:

object Test {

def main(args: Array[String]) {

val site = "Runoob" :: ("Google" :: ("Baidu" :: Nil))

val nums = Nil

println( "第一网站是 : " + site.head )

println( "最后一个网站是 : " + site.tail )

println( "查看列表 site 是否为空 : " + site.isEmpty )

println( "查看 nums 是否为空 : " + nums.isEmpty )

}}

执行以上代码，输出结果为：

$ vim Test.scala

$ scala Test.scala 第一网站是 : Runoob最后一个网站是 : List(Google, Baidu)查看列表 site 是否为空 : false查看 nums 是否为空 : true

### 连接列表

你可以使用 **:::** 运算符或 **List.:::()** 方法或 **List.concat()** 方法来连接两个或多个列表。实例如下:

|  |
| --- |
| *//空数组用Nil表示* **val** site1 = **"Runoob"** :: (**"Google"** :: (**"Baidu"** :: *Nil*)) **val** site2 = **"Facebook"** :: (**"Taobao"** :: *Nil*)  *// 使用 ::: 运算符,site1在前，site2在后* **var** fruit = site1 ::: site2 *println*( **"site1 ::: site2 : "** + fruit )  *// 使用 Set.:::() 方法，site2在前，site1在后* fruit = site1.:::(site2) *println*( **"site1.:::(site2) : "** + fruit )  *// 使用 concat 方法* fruit = *List*.concat(site1, site2) *println*( **"List.concat(site1, site2) : "** + fruit ) |

### List.fill()

我们可以使用 List.fill() 方法来创建一个指定重复数量的元素列表：

|  |
| --- |
| **val** site=*List*.fill(3)(**"Runoob"**) *println*(site)  //List(Runoob, Runoob, Runoob) |

### List.tabulate()

List.tabulate() 方法是通过给定的函数来创建列表。

方法的第一个参数为元素的数量，可以是二维的，第二个参数为指定的函数，我们通过指定的函数计算结果并返回值插入到列表中，起始值为 0，实例如下：

object Test {

def main(args: Array[String]) {

// 通过给定的函数创建 56个元素

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

println( "一维 : " + squares )

// 创建二维列表

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

println( "多维 : " + mul )

}}

执行以上代码，输出结果为：

一维 : List(0, 1, 4, 9, 16, 25)多维 : 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))

### List.reverse

List.reverse 用于将列表的顺序反转，实例如下：

object Test {

def main(args: Array[String]) {

val site = "Runoob" :: ("Google" :: ("Baidu" :: Nil))

println( "site 反转前 : " + site )

println( "site 反转前 : " + site.reverse )

}}

执行以上代码，输出结果为：

site 反转前 : List(Runoob, Google, Baidu)

site 反转前 : List(Baidu, Google, Runoob)

### Scala List 常用方法

下表列出了 Scala List 常用的方法：

|  |  |
| --- | --- |
| **序号** | **方法及描述** |
| 1 | **def +:(elem: A): List[A]**  为列表预添加元素  scala> val x = List(1)  x: List[Int] = List(1)  scala> val y = 2 +: x  y: List[Int] = List(2, 1)  scala> println(x)List(1) |
| 2 | **def ::(x: A): List[A]**  在列表开头添加元素 |
| 3 | **def :::(prefix: List[A]): List[A]**  在列表开头添加指定列表的元素 |
| 4 | **def :+(elem: A): List[A]**  复制添加元素后列表。  scala> val a = List(1)  a: List[Int] = List(1)  scala> val b = a :+ 2  b: List[Int] = List(1, 2)  scala> println(a)List(1) |
| 5 | **def addString(b: StringBuilder): StringBuilder**  将列表的所有元素添加到 StringBuilder |
| 6 | **def addString(b: StringBuilder, sep: String): StringBuilder**  将列表的所有元素添加到 StringBuilder，并指定分隔符 |
| 7 | **def apply(n: Int): A**  通过列表索引获取元素 |
| 8 | **def contains(elem: Any): Boolean**  检测列表中是否包含指定的元素 |
| 9 | **def copyToArray(xs: Array[A], start: Int, len: Int): Unit**  将列表的元素复制到数组中。 |
| 10 | **def distinct: List[A]**  去除列表的重复元素，并返回新列表 |
| 11 | **def drop(n: Int): List[A]**  丢弃前n个元素，并返回新列表 |
| 12 | **def dropRight(n: Int): List[A]**  丢弃最后n个元素，并返回新列表 |
| 13 | **def dropWhile(p: (A) => Boolean): List[A]**  从左向右丢弃元素，直到条件p不成立 |
| 14 | **def endsWith[B](that: Seq[B]): Boolean**  检测列表是否以指定序列结尾 |
| 15 | **def equals(that: Any): Boolean**  判断是否相等 |
| 16 | **def exists(p: (A) => Boolean): Boolean**  判断列表中指定条件的元素是否存在。  判断l是否存在某个元素:  scala> l.exists(s => s == "Hah")  res7: Boolean = true |
| 17 | **def filter(p: (A) => Boolean): List[A]**  输出符号指定条件的所有元素。  过滤出长度为3的元素:  scala> l.filter(s => s.length == 3)  res8: List[String] = List(Hah, WOW) |
| 18 | **def forall(p: (A) => Boolean): Boolean**  检测所有元素。  例如：判断所有元素是否以"H"开头：  scala> l.forall(s => s.startsWith("H")) res10: Boolean = false |
| 19 | **def foreach(f: (A) => Unit): Unit**  将函数应用到列表的所有元素 |
| 20 | **def head: A**  获取列表的第一个元素 |
| 21 | **def indexOf(elem: A, from: Int): Int**  从指定位置 from 开始查找元素第一次出现的位置 |
| 22 | **def init: List[A]**  返回所有元素，除了最后一个 |
| 23 | **def intersect(that: Seq[A]): List[A]**  计算多个集合的交集 |
| 24 | **def isEmpty: Boolean**  检测列表是否为空 |
| 25 | **def iterator: Iterator[A]**  创建一个新的迭代器来迭代元素 |
| 26 | **def last: A**  返回最后一个元素 |
| 27 | **def lastIndexOf(elem: A, end: Int): Int**  在指定的位置 end 开始查找元素最后出现的位置 |
| 28 | **def length: Int**  返回列表长度 |
| 29 | **def map[B](f: (A) => B): List[B]**  通过给定的方法将所有元素重新计算 |
| 30 | **def max: A**  查找最大元素 |
| 31 | **def min: A**  查找最小元素 |
| 32 | **def mkString: String**  列表所有元素作为字符串显示 |
| 33 | **def mkString(sep: String): String**  使用分隔符将列表所有元素作为字符串显示 |
| 34 | **def reverse: List[A]**  列表反转 |
| 35 | **def sorted[B >: A]: List[A]**  列表排序 |
| 36 | **def startsWith[B](that: Seq[B], offset: Int): Boolean**  检测列表在指定位置是否包含指定序列 |
| 37 | **def sum: A**  计算集合元素之和 |
| 38 | **def tail: List[A]**  返回所有元素，除了第一个 |
| 39 | **def take(n: Int): List[A]**  提取列表的前n个元素 |
| 40 | **def takeRight(n: Int): List[A]**  提取列表的后n个元素 |
| 41 | **def toArray: Array[A]**  列表转换为数组 |
| 42 | **def toBuffer[B >: A]: Buffer[B]**  返回缓冲区，包含了列表的所有元素 |
| 43 | **def toMap[T, U]: Map[T, U]**  List 转换为 Map |
| 44 | **def toSeq: Seq[A]**  List 转换为 Seq |
| 45 | **def toSet[B >: A]: Set[B]**  List 转换为 Set |
| 46 | **def toString(): String**  列表转换为字符串 |

## Scala Set(集合)

Scala Set(集合)是没有重复的对象集合，所有的元素都是唯一的。

Scala 集合分为可变的和不可变的集合。

默认情况下，Scala 使用的是不可变集合，如果你想使用可变集合，需要引用 **scala.collection.mutable.Set** 包。

默认引用 scala.collection.immutable.Set，不可变集合实例如下：

|  |
| --- |
| **val** set=*Set*(1,5,3)  *println*(set.getClass.getName)*//scala.collection.immutable.Set$Set3  println*(set.exists(\_%2==0))*//false  println*(set.drop(1))*//Set(5,3)* |

如果需要使用可变集合需要引入 scala.collection.mutable.Set：

import scala.collection.mutable.Set // 可以在任何地方引入 可变集合

val mutableSet = Set(1,2,3)

println(mutableSet.getClass.getName) // scala.collection.mutable.HashSet

mutableSet.add(4)

mutableSet.remove(1)

mutableSet += 5

mutableSet -= 2

println(mutableSet) // Set(5, 3, 4)

val another = mutableSet.toSet

println(another.getClass.getName) // scala.collection.immutable.Set

***注意：****虽然可变Set和不可变Set都有添加或删除元素的操作，但是有一个非常大的差别。对不可变Set进行操作，会产生一个新的set，原来的set并没有改变，这与List一样。 而对可变Set进行操作，改变的是该Set本身，与ListBuffer类似。*

### 集合基本操作

Scala集合有三个基本操作：

* head 返回集合第一个元素
* tail 返回一个集合，包含除了第一元素之外的其他元素
* isEmpty 在集合为空时返回true

对于Scala集合的任何操作都可以使用这三个基本操作来表达。实例如下:

object Test {

def main(args: Array[String]) {

val site = Set("Runoob", "Google", "Baidu")

val nums: Set[Int] = Set()

println( "第一网站是 : " + site.head )

println( "最后一个网站是 : " + site.tail )

println( "查看列表 site 是否为空 : " + site.isEmpty )

println( "查看 nums 是否为空 : " + nums.isEmpty )

}

}

### 连接集合

你可以使用 **++** 运算符或 **Set.++()** 方法来连接两个集合。如果元素有重复的就会移除重复的元素。

|  |
| --- |
| **val** site1 = *Set*(**"Runoob"**, **"Google"**, **"Baidu"**) **val** site2 = *Set*(**"Faceboook"**, **"Taobao"**)  *// ++ 作为运算符使用* **var** site = site1 ++ site2 *println*( **"site1 ++ site2 : "** + site )  *// ++ 作为方法使用* site = site1.++(site2) *println*( **"site1.++(site2) : "** + site ) |

### 查找集合中最大与最小元素

你可以使用 **Set.min** 方法来查找集合中的最小元素，使用 **Set.max** 方法查找集合中的最大元素。实

|  |
| --- |
| **val** num = *Set*(5,6,9,20,30)  *// 查找集合中最大与最小元素 println*( **"Set(5,6,9,20,30,45) 集合中的最小元素是 : "** + num.min ) *println*( **"Set(5,6,9,20,30,45) 集合中的最大元素是 : "** + num.max ) |

### 交集

你可以使用 **Set.&** 方法或 **Set.intersect** 方法来查看两个集合的交集元素。实例如下：

执行以上代码，输出结果为：

|  |
| --- |
| **val** num1 = *Set*(5,6,9,20,30,45) **val** num2 = *Set*(50,60,9,20,35,55)  *// 交集 println*( **"num1.&(num2) : "** + num1.&(num2) ) *println*( **"num1.intersect(num2) : "** + num1.intersect(num2) ) |

### Scala Set 常用方法

下表列出了 Scala Set 常用的方法：

|  |  |
| --- | --- |
| **序号** | **方法及描述** |
| 1 | **def +(elem: A): Set[A]**  为集合添加新元素，x并创建一个新的集合，除非元素已存在 |
| 2 | **def -(elem: A): Set[A]**  移除集合中的元素，并创建一个新的集合 |
| 3 | **def contains(elem: A): Boolean**  如果元素在集合中存在，返回 true，否则返回 false。 |
| 4 | **def &(that: Set[A]): Set[A]**  返回两个集合的交集 |
| 5 | **def &~(that: Set[A]): Set[A]**  返回两个集合的差集 |
| 6 | **def +(elem1: A, elem2: A, elems: A\*): Set[A]**  通过添加传入指定集合的元素创建一个新的不可变集合 |
| 7 | **def ++(elems: A): Set[A]**  合并两个集合 |
| 8 | **def -(elem1: A, elem2: A, elems: A\*): Set[A]**  通过移除传入指定集合的元素创建一个新的不可变集合 |
| 9 | **def addString(b: StringBuilder): StringBuilder**  将不可变集合的所有元素添加到字符串缓冲区 |
| 10 | **def addString(b: StringBuilder, sep: String): StringBuilder**  将不可变集合的所有元素添加到字符串缓冲区，并使用指定的分隔符 |
| 11 | **def apply(elem: A)**  检测集合中是否包含指定元素 |
| 12 | **def count(p: (A) => Boolean): Int**  计算满足指定条件的集合元素个数 |
| 13 | **def copyToArray(xs: Array[A], start: Int, len: Int): Unit**  复制不可变集合元素到数组 |
| 14 | **def diff(that: Set[A]): Set[A]**  比较两个集合的差集 |
| 15 | **def drop(n: Int): Set[A]]**  返回丢弃前n个元素新集合 |
| 16 | **def dropRight(n: Int): Set[A]**  返回丢弃最后n个元素新集合 |
| 17 | **def dropWhile(p: (A) => Boolean): Set[A]**  从左向右丢弃元素，直到条件p不成立 |
| 18 | **def equals(that: Any): Boolean**  equals 方法可用于任意序列。用于比较系列是否相等。 |
| 19 | **def exists(p: (A) => Boolean): Boolean**  判断不可变集合中指定条件的元素是否存在。 |
| 20 | **def filter(p: (A) => Boolean): Set[A]**  输出符合指定条件的所有不可变集合元素。 |
| 21 | **def find(p: (A) => Boolean): Option[A]**  查找不可变集合中满足指定条件的第一个元素 |
| 22 | **def forall(p: (A) => Boolean): Boolean**  查找不可变集合中满足指定条件的所有元素 |
| 23 | **def foreach(f: (A) => Unit): Unit**  将函数应用到不可变集合的所有元素 |
| 24 | **def head: A**  获取不可变集合的第一个元素 |
| 25 | **def init: Set[A]**  返回所有元素，除了最后一个 |
| 26 | **def intersect(that: Set[A]): Set[A]**  计算两个集合的交集 |
| 27 | **def isEmpty: Boolean**  判断集合是否为空 |
| 28 | **def iterator: Iterator[A]**  创建一个新的迭代器来迭代元素 |
| 29 | **def last: A**  返回最后一个元素 |
| 30 | **def map[B](f: (A) => B): immutable.Set[B]**  通过给定的方法将所有元素重新计算 |
| 31 | **def max: A**  查找最大元素 |
| 32 | **def min: A**  查找最小元素 |
| 33 | **def mkString: String**  集合所有元素作为字符串显示 |
| 34 | **def mkString(sep: String): String**  使用分隔符将集合所有元素作为字符串显示 |
| 35 | **def product: A**  返回不可变集合中数字元素的积。 |
| 36 | **def size: Int**  返回不可变集合元素的数量 |
| 37 | **def splitAt(n: Int): (Set[A], Set[A])**  把不可变集合拆分为两个容器，第一个由前 n 个元素组成，第二个由剩下的元素组成 |
| 38 | **def subsetOf(that: Set[A]): Boolean**  如果集合中含有子集返回 true，否则返回false |
| 39 | **def sum: A**  返回不可变集合中所有数字元素之和 |
| 40 | **def tail: Set[A]**  返回一个不可变集合中除了第一元素之外的其他元素 |
| 41 | **def take(n: Int): Set[A]**  返回前 n 个元素 |
| 42 | **def takeRight(n: Int):Set[A]**  返回后 n 个元素 |
| 43 | **def toArray: Array[A]**  将集合转换为数字 |
| 44 | **def toBuffer[B >: A]: Buffer[B]**  返回缓冲区，包含了不可变集合的所有元素 |
| 45 | **def toList: List[A]**  返回 List，包含了不可变集合的所有元素 |
| 46 | **def toMap[T, U]: Map[T, U]**  返回 Map，包含了不可变集合的所有元素 |
| 47 | **def toSeq: Seq[A]**  返回 Seq，包含了不可变集合的所有元素 |
| 48 | **def toString(): String**  返回一个字符串，以对象来表示 |

## Scala Map(映射)

Map 也叫哈希表（Hash tables）。

Map 有两种类型，可变与不可变，区别在于可变对象可以修改它，而不可变对象不可以。

默认情况下 Scala 使用不可变 Map。如果你需要使用可变集合，你需要显式的引入

**import scala.collection.mutable.Map**类

在 Scala 中 你可以同时使用可变与不可变 Map，不可变的直接使用 Map，可变的使用 mutable.Map。以下实例演示了不可变 Map 的应用：

// 空哈希表，键为字符串，值为整型

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

// Map 键值对演示

val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF")

定义 Map 时，需要为键值对定义类型。如果需要添加 key-value 对，可以使用 + 号，如下所示：

A += ('I' -> 1)

A += ('J' -> 5)

A += ('K' -> 10)

A += ('L' -> 100)

### Map 基本操作

Scala Map 有三个基本操作：

|  |  |
| --- | --- |
| **方法** | **描述** |
| keys | 返回 Map 所有的键(key) |
| values | 返回 Map 所有的值(value) |
| isEmpty | 在 Map 为空时返回true |

**实例**

以下实例演示了以上三个方法的基本应用：

object Test {

def main(args: Array[String]) {

val colors = Map("red" -> "#FF0000",

"azure" -> "#F0FFFF",

"peru" -> "#CD853F")

val nums: Map[Int, Int] = Map()

println( "colors 中的键为 : " + colors.keys )

println( "colors 中的值为 : " + colors.values )

println( "检测 colors 是否为空 : " + colors.isEmpty )

println( "检测 nums 是否为空 : " + nums.isEmpty )

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

colors 中的键为 : Set(red, azure, peru)

colors 中的值为 : MapLike(#FF0000, #F0FFFF, #CD853F)

检测 colors 是否为空 : false

检测 nums 是否为空 : true

### Map 合并

你可以使用 **++** 运算符或 **Map.++()** 方法来连接两个 Map，Map 合并时会移除重复的 key。以下演示了两个 Map 合并的实例:

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")

// ++ 作为运算符

var colors = colors1 ++ colors2

println( "colors1 ++ colors2 : " + colors )

// ++ 作为方法

colors = colors1.++(colors2)

println( "colors1.++(colors2)) : " + colors )

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ 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)

### 输出 Map 的 keys 和 values

以下通过 foreach 循环输出 Map 中的 keys 和 values：

object Test {

def main(args: Array[String]) {

val sites = Map("runoob" -> "http://www.runoob.com",

"baidu" -> "http://www.baidu.com",

"taobao" -> "http://www.taobao.com")

sites.keys.foreach{ i =>

print( "Key = " + i )

println(" Value = " + sites(i) )}

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

Key = runoob Value = http://www.runoob.com

Key = baidu Value = http://www.baidu.com

Key = taobao Value = http://www.taobao.com

### 查看 Map 中是否存在指定的 Key

你可以使用 **Map.contains** 方法来查看 Map 中是否存在指定的 Key。实例如下：

object Test {

def main(args: Array[String]) {

val sites = Map("runoob" -> "http://www.runoob.com",

"baidu" -> "http://www.baidu.com",

"taobao" -> "http://www.taobao.com")

if( sites.contains( "runoob" )){

println("runoob 键存在，对应的值为 :" + sites("runoob"))

}else{

println("runoob 键不存在")

}

if( sites.contains( "baidu" )){

println("baidu 键存在，对应的值为 :" + sites("baidu"))

}else{

println("baidu 键不存在")

}

if( sites.contains( "google" )){

println("google 键存在，对应的值为 :" + sites("google"))

}else{

println("google 键不存在")

}

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

runoob 键存在，对应的值为 :http://www.runoob.com

baidu 键存在，对应的值为 :http://www.baidu.com

google 键不存在

### Scala Map 方法

下表列出了 Scala Map 常用的方法：

|  |  |
| --- | --- |
| **序号** | **方法及描述** |
| 1 | **def ++(xs: Map[(A, B)]): Map[A, B]**  返回一个新的 Map，新的 Map xs 组成 |
| 2 | **def -(elem1: A, elem2: A, elems: A\*): Map[A, B]**  返回一个新的 Map, 移除 key 为 elem1, elem2 或其他 elems。 |
| 3 | **def --(xs: GTO[A]): Map[A, B]**  返回一个新的 Map, 移除 xs 对象中对应的 key |
| 4 | **def get(key: A): Option[B]**  返回指定 key 的值 |
| 5 | **def iterator: Iterator[(A, B)]**  创建新的迭代器，并输出 key/value 对 |
| 6 | **def addString(b: StringBuilder): StringBuilder**  将 Map 中的所有元素附加到StringBuilder，可加入分隔符 |
| 7 | **def addString(b: StringBuilder, sep: String): StringBuilder**  将 Map 中的所有元素附加到StringBuilder，可加入分隔符 |
| 8 | **def apply(key: A): B**  返回指定键的值，如果不存在返回 Map 的默认方法 |
| 9 | **def clear(): Unit**  清空 Map |
| 10 | **def clone(): Map[A, B]**  从一个 Map 复制到另一个 Map |
| 11 | **def contains(key: A): Boolean**  如果 Map 中存在指定 key，返回 true，否则返回 false。 |
| 12 | **def copyToArray(xs: Array[(A, B)]): Unit**  复制集合到数组 |
| 13 | **def count(p: ((A, B)) => Boolean): Int**  计算满足指定条件的集合元素数量 |
| 14 | **def default(key: A): B**  定义 Map 的默认值，在 key 不存在时返回。 |
| 15 | **def drop(n: Int): Map[A, B]**  返回丢弃前n个元素新集合 |
| 16 | **def dropRight(n: Int): Map[A, B]**  返回丢弃最后n个元素新集合 |
| 17 | **def dropWhile(p: ((A, B)) => Boolean): Map[A, B]**  从左向右丢弃元素，直到条件p不成立 |
| 18 | **def empty: Map[A, B]**  返回相同类型的空 Map |
| 19 | **def equals(that: Any): Boolean**  如果两个 Map 相等(key/value 均相等)，返回true，否则返回false |
| 20 | **def exists(p: ((A, B)) => Boolean): Boolean**  判断集合中指定条件的元素是否存在 |
| 21 | **def filter(p: ((A, B))=> Boolean): Map[A, B]**  返回满足指定条件的所有集合 |
| 22 | **def filterKeys(p: (A) => Boolean): Map[A, B]**  返回符合指定条件的的不可变 Map |
| 23 | **def find(p: ((A, B)) => Boolean): Option[(A, B)]**  查找集合中满足指定条件的第一个元素 |
| 24 | **def foreach(f: ((A, B)) => Unit): Unit**  将函数应用到集合的所有元素 |
| 25 | **def init: Map[A, B]**  返回所有元素，除了最后一个 |
| 26 | **def isEmpty: Boolean**  检测 Map 是否为空 |
| 27 | **def keys: Iterable[A]**  返回所有的key/p> |
| 28 | **def last: (A, B)**  返回最后一个元素 |
| 29 | **def max: (A, B)**  查找最大元素 |
| 30 | **def min: (A, B)**  查找最小元素 |
| 31 | **def mkString: String**  集合所有元素作为字符串显示 |
| 32 | **def product: (A, B)**  返回集合中数字元素的积。 |
| 33 | **def remove(key: A): Option[B]**  移除指定 key |
| 34 | **def retain(p: (A, B) => Boolean): Map.this.type**  如果符合满足条件的返回 true |
| 35 | **def size: Int**  返回 Map 元素的个数 |
| 36 | **def sum: (A, B)**  返回集合中所有数字元素之和 |
| 37 | **def tail: Map[A, B]**  返回一个集合中除了第一元素之外的其他元素 |
| 38 | **def take(n: Int): Map[A, B]**  返回前 n 个元素 |
| 39 | **def takeRight(n: Int): Map[A, B]**  返回后 n 个元素 |
| 40 | **def takeWhile(p: ((A, B)) => Boolean): Map[A, B]**  返回满足指定条件的元素 |
| 41 | **def toArray: Array[(A, B)]**  集合转数组 |
| 42 | **def toBuffer[B >: A]: Buffer[B]**  返回缓冲区，包含了 Map 的所有元素 |
| 43 | **def toList: List[A]**  返回 List，包含了 Map 的所有元素 |
| 44 | **def toSeq: Seq[A]**  返回 Seq，包含了 Map 的所有元素 |
| 45 | **def toSet: Set[A]**  返回 Set，包含了 Map 的所有元素 |
| 46 | **def toString(): String**  返回字符串对象 |

## Scala 元组

与列表一样，元组也是不可变的，但与列表不同的是元组可以包含不同类型的元素。

元组的值是通过将单个的值包含在圆括号中构成的。例如：

val t = (1, 3.14, "Fred")

以上实例在元组中定义了三个元素，对应的类型分别为[Int, Double, java.lang.String]。

此外我们也可以使用以上方式来定义：

val t = new Tuple3(1, 3.14, "Fred")

元组的实际类型取决于它的元素的类型，比如 (99, "runoob") 是 Tuple2[Int, String]。 ('u', 'r', "the", 1, 4, "me") 为 Tuple6[Char, Char, String, Int, Int, String]。

目前 Scala 支持的元组最大长度为 22。对于更大长度你可以使用集合，或者扩展元组。

访问元组的元素可以通过数字索引，如下一个元组：

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

我们可以使用 t.\_1 访问第一个元素， t.\_2 访问第二个元素，如下所示：

object Test {

def main(args: Array[String]) {

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

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

println( "元素之和为: " + sum )

}}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test元素之和为: 10

### 迭代元组

你可以使用 **Tuple.productIterator()** 方法来迭代输出元组的所有元素：

object Test {

def main(args: Array[String]) {

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

t.productIterator.foreach{ i =>println("Value = " + i )}

}}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala TestValue = 4Value = 3Value = 2Value = 1

### 元组转为字符串

你可以使用 **Tuple.toString()** 方法将元组的所有元素组合成一个字符串，实例如下：

object Test {

def main(args: Array[String]) {

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

println("连接后的字符串为: " + t.toString() )

}}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test连接后的字符串为: (1,hello,scala.Console$@4dd8dc3)

### 元素交换

你可以使用 **Tuple.swap** 方法来交换元组的元素。如下实例：

object Test {

def main(args: Array[String]) {

val t = new Tuple2("www.google.com", "www.runoob.com")

println("交换后的元组: " + t.swap )

}}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test交换后的元组: (www.runoob.com,www.google.com)

## Scala Option(选项)

Scala Option(选项)类型用来表示一个值是可选的（有值或无值)。

Option[T] 是一个类型为 T 的可选值的容器： 如果值存在， Option[T] 就是一个 Some[T] ，如果不存在， Option[T] 就是对象 None 。

接下来我们来看一段代码：

// 虽然 Scala 可以不定义变量的类型，不过为了清楚些，我还是

// 把他显示的定义上了

val myMap: Map[String, String] = Map("key1" -> "value")

val value1: Option[String] = myMap.get("key1")

val value2: Option[String] = myMap.get("key2")

println(value1) // Some("value1")

println(value2) // None

在上面的代码中，myMap 一个是一个 Key 的类型是 String，Value 的类型是 String 的 hash map，但不一样的是他的 get() 返回的是一个叫 Option[String] 的类别。

Scala 使用 Option[String] 来告诉你：「我会想办法回传一个 String，但也可能没有 String 给你」。

myMap 里并没有 key2 这笔数据，get() 方法返回 None。

Option 有两个子类别，一个是 Some，一个是 None，当他回传 Some 的时候，代表这个函式成功地给了你一个 String，而你可以透过 get() 这个函式拿到那个 String，如果他返回的是 None，则代表没有字符串可以给你。

另一个实例：

object Test {

def main(args: Array[String]) {

val sites = Map("runoob" -> "www.runoob.com", "google" -> "www.google.com")

println("sites.get( \"runoob\" ) : " + sites.get( "runoob" )) // Some(www.runoob.com)

println("sites.get( \"baidu\" ) : " + sites.get( "baidu" )) // None

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

sites.get( "runoob" ) : Some(www.runoob.com)

sites.get( "baidu" ) : None

你也可以通过模式匹配来输出匹配值。实例如下：

object Test {

def main(args: Array[String]) {

val sites = Map("runoob" -> "www.runoob.com", "google" -> "www.google.com")

println("show(sites.get( \"runoob\")) : " +

show(sites.get( "runoob")) )

println("show(sites.get( \"baidu\")) : " +

show(sites.get( "baidu")) )

}

def show(x: Option[String]) = x match {

case Some(s) => s

case None => "?"

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

show(sites.get( "runoob")) : www.runoob.com

show(sites.get( "baidu")) : ?

### getOrElse() 方法

你可以使用 getOrElse() 方法来获取元组中存在的元素或者使用其默认的值，实例如下：

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) )

}

}

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Test

a.getOrElse(0): 5

b.getOrElse(10): 10

### isEmpty() 方法

你可以使用 isEmpty() 方法来检测元组中的元素是否为 None，实例如下：

object Test {

def main(args: Array[String]) {

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

val b:Option[Int] = None

println("a.isEmpty: " + a.isEmpty )//a.isEmpty: false

println("b.isEmpty: " + b.isEmpty )//a.isEmpty: true

}

}

### Scala Option 常用方法

下表列出了 Scala Option 常用的方法：

|  |  |
| --- | --- |
| **序号** | **方法及描述** |
| 1 | **def get: A**  获取可选值 |
| 2 | **def isEmpty: Boolean**  检测可选类型值是否为 None，是的话返回 true，否则返回 false |
| 3 | **def productArity: Int**  返回元素个数， A(x\_1, ..., x\_k), 返回 k |
| 4 | **def productElement(n: Int): Any**  获取指定的可选项，以 0 为起始。即 A(x\_1, ..., x\_k), 返回 x\_(n+1) ， 0 < n < k. |
| 5 | **def exists(p: (A) => Boolean): Boolean**  如果可选项中指定条件的元素是否存在且不为 None 返回 true，否则返回 false。 |
| 6 | **def filter(p: (A) => Boolean): Option[A]**  如果选项包含有值，而且传递给 filter 的条件函数返回 true， filter 会返回 Some 实例。 否则，返回值为 None 。 |
| 7 | **def filterNot(p: (A) => Boolean): Option[A]**  如果选项包含有值，而且传递给 filter 的条件函数返回 false， filter 会返回 Some 实例。 否则，返回值为 None 。 |
| 8 | **def flatMap[B](f: (A) => Option[B]): Option[B]**  如果选项包含有值，则传递给函数 f 处理后返回，否则返回 None |
| 9 | **def foreach[U](f: (A) => U): Unit**  如果选项包含有值，则将每个值传递给函数 f， 否则不处理。 |
| 10 | **def getOrElse[B >: A](default: => B): B**  如果选项包含有值，返回选项值，否则返回设定的默认值。 |
| 11 | **def isDefined: Boolean**  如果可选值是 Some 的实例返回 true，否则返回 false。 |
| 12 | **def iterator: Iterator[A]**  如果选项包含有值，迭代出可选值。如果可选值为空则返回空迭代器。 |
| 13 | **def map[B](f: (A) => B): Option[B]**  如果选项包含有值， 返回由函数 f 处理后的 Some，否则返回 None |
| 14 | **def orElse[B >: A](alternative: => Option[B]): Option[B]**  如果一个 Option 是 None ， orElse 方法会返回传名参数的值，否则，就直接返回这个 Option。 |
| 15 | **def orNull**  如果选项包含有值返回选项值，否则返回 null。 |

## Scala Iterator（迭代器）

Scala Iterator（迭代器）不是一个集合，它是一种用于访问集合的方法。

迭代器 it 的两个基本操作是 **next** 和 **hasNext**。

调用 **it.next()** 会返回迭代器的下一个元素，并且更新迭代器的状态。

调用 **it.hasNext()** 用于检测集合中是否还有元素。

让迭代器 it 逐个返回所有元素最简单的方法是使用 while 循环：

object Test {

def main(args: Array[String]) {

val it = Iterator("Baidu", "Google", "Runoob", "Taobao")

while (it.hasNext){

println(it.next())

}

}}

### 查找最大与最小元素

你可以使用 **it.min** 和 **it.max** 方法从迭代器中查找最大与最小元素，实例如下:

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("最大元素是：" + ita.max )

println("最小元素是：" + itb.min )

}}

### 获取迭代器的长度

你可以使用 **it.size** 或 **it.length** 方法来查看迭代器中的元素个数。实例如下：

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("ita.size 的值: " + ita.size )

println("itb.length 的值: " + itb.length )

}}

### Scala Iterator 常用方法

下表列出了 Scala Iterator 常用的方法：

|  |  |
| --- | --- |
| **序号** | **方法及描述** |
| 1 | **def hasNext: Boolean**  如果还有可返回的元素，返回true。 |
| 2 | **def next(): A**  返回迭代器的下一个元素，并且更新迭代器的状态 |
| 3 | **def ++(that: => Iterator[A]): Iterator[A]**  合并两个迭代器 |
| 4 | **def ++[B >: A](that :=> GenTraversableOnce[B]): Iterator[B]**  合并两个迭代器 |
| 5 | **def addString(b: StringBuilder): StringBuilder**  添加一个字符串到 StringBuilder b |
| 6 | **def addString(b: StringBuilder, sep: String): StringBuilder**  添加一个字符串到 StringBuilder b，并指定分隔符 |
| 7 | **def buffered: BufferedIterator[A]**  迭代器都转换成 BufferedIterator |
| 8 | **def contains(elem: Any): Boolean**  检测迭代器中是否包含指定元素 |
| 9 | **def copyToArray(xs: Array[A], start: Int, len: Int): Unit**  将迭代器中选定的值传给数组 |
| 10 | **def count(p: (A) => Boolean): Int**  返回迭代器元素中满足条件p的元素总数。 |
| 11 | **def drop(n: Int): Iterator[A]**  返回丢弃前n个元素新集合 |
| 12 | **def dropWhile(p: (A) => Boolean): Iterator[A]**  从左向右丢弃元素，直到条件p不成立 |
| 13 | **def duplicate: (Iterator[A], Iterator[A])**  生成两个能分别返回迭代器所有元素的迭代器。 |
| 14 | **def exists(p: (A) => Boolean): Boolean**  返回一个布尔值，指明迭代器元素中是否存在满足p的元素。 |
| 15 | **def filter(p: (A) => Boolean): Iterator[A]**  返回一个新迭代器 ，指向迭代器元素中所有满足条件p的元素。 |
| 16 | **def filterNot(p: (A) => Boolean): Iterator[A]**  返回一个迭代器，指向迭代器元素中不满足条件p的元素。 |
| 17 | **def find(p: (A) => Boolean): Option[A]**  返回第一个满足p的元素或None。注意：如果找到满足条件的元素，迭代器会被置于该元素之后；如果没有找到，会被置于终点。 |
| 18 | **def flatMap[B](f: (A) => GenTraversableOnce[B]): Iterator[B]**  针对迭代器的序列中的每个元素应用函数f，并返回指向结果序列的迭代器。 |
| 19 | **def forall(p: (A) => Boolean): Boolean**  返回一个布尔值，指明 it 所指元素是否都满足p。 |
| 20 | **def foreach(f: (A) => Unit): Unit**  在迭代器返回的每个元素上执行指定的程序 f |
| 21 | **def hasDefiniteSize: Boolean**  如果迭代器的元素个数有限则返回true（缺省等同于isEmpty） |
| 22 | **def indexOf(elem: B): Int**  返回迭代器的元素中index等于x的第一个元素。注意：迭代器会越过这个元素。 |
| 23 | **def indexWhere(p: (A) => Boolean): Int**  返回迭代器的元素中下标满足条件p的元素。注意：迭代器会越过这个元素。 |
| 24 | **def isEmpty: Boolean**  检查it是否为空, 为空返回 true，否则返回false（与hasNext相反）。 |
| 25 | **def isTraversableAgain: Boolean**  Tests whether this Iterator can be repeatedly traversed. |
| 26 | **def length: Int**  返回迭代器元素的数量。 |
| 27 | **def map[B](f: (A) => B): Iterator[B]**  将 it 中的每个元素传入函数 f 后的结果生成新的迭代器。 |
| 28 | **def max: A**  返回迭代器迭代器元素中最大的元素。 |
| 29 | **def min: A**  返回迭代器迭代器元素中最小的元素。 |
| 30 | **def mkString: String**  将迭代器所有元素转换成字符串。 |
| 31 | **def mkString(sep: String): String**  将迭代器所有元素转换成字符串，并指定分隔符。 |
| 32 | **def nonEmpty: Boolean**  检查容器中是否包含元素（相当于 hasNext）。 |
| 33 | **def padTo(len: Int, elem: A): Iterator[A]**  首先返回迭代器所有元素，追加拷贝 elem 直到长度达到 len。 |
| 34 | **def patch(from: Int, patchElems: Iterator[B], replaced: Int): Iterator[B]**  返回一个新迭代器，其中自第 from 个元素开始的 replaced 个元素被迭代器所指元素替换。 |
| 35 | **def product: A**  返回迭代器所指数值型元素的积。 |
| 36 | **def sameElements(that: Iterator[\_]): Boolean**  判断迭代器和指定的迭代器参数是否依次返回相同元素 |
| 37 | **def seq: Iterator[A]**  返回集合的系列视图 |
| 38 | **def size: Int**  返回迭代器的元素数量 |
| 39 | **def slice(from: Int, until: Int): Iterator[A]**  返回一个新的迭代器，指向迭代器所指向的序列中从开始于第 from 个元素、结束于第 until 个元素的片段。 |
| 40 | **def sum: A**  返回迭代器所指数值型元素的和 |
| 41 | **def take(n: Int): Iterator[A]**  返回前 n 个元素的新迭代器。 |
| 42 | **def toArray: Array[A]**  将迭代器指向的所有元素归入数组并返回。 |
| 43 | **def toBuffer: Buffer[B]**  将迭代器指向的所有元素拷贝至缓冲区 Buffer。 |
| 44 | **def toIterable: Iterable[A]**  Returns an Iterable containing all elements of this traversable or iterator. This will not terminate for infinite iterators. |
| 45 | **def toIterator: Iterator[A]**  把迭代器的所有元素归入一个Iterator容器并返回。 |
| 46 | **def toList: List[A]**  把迭代器的所有元素归入列表并返回 |
| 47 | **def toMap[T, U]: Map[T, U]**  将迭代器的所有键值对归入一个Map并返回。 |
| 48 | **def toSeq: Seq[A]**  将代器的所有元素归入一个Seq容器并返回。 |
| 49 | **def toString(): String**  将迭代器转换为字符串 |
| 50 | **def zip[B](that: Iterator[B]): Iterator[(A, B)**  返回一个新迭代器，指向分别由迭代器和指定的迭代器 that 元素一一对应而成的二元组序列 |

# Scala 类和对象

## 类和对象

*\* 1 在定义Scala的class的时候可以直接在类名后面（）里加入类的构造参数，此时在apply方法中页必须有这些参数  
\* 2 在Scala的object中可以有多个apply  
\* 2 如果名称相同，则object中的内容都是class的静态内容，也就是说object中的内容class都可以在没有实例的时候直接调用  
\* 正是因为可以在没有类的实例的时候去调用object中的一切内容，所以可以使用object中特定方法来创建类的实例，而这个特定的  
\* 方法就是apply方法。  
\* 3 object 中的apply方式class对象生成的工厂方法，用于控制对象的生成。*

***Object中的内容不属于class的api（不属于类接口），所以对版本迭代很有用***

***因为类继承的时候不会继承伴生对象的。*** *\* 4 很多框架的代码一般直接调用抽象类的object的apply方法生成类的实例对象：  
\* 第一：其秘诀在于apply具有类对象生成的一切生杀大权，首相类不可以实例化，在apply中可以实例化抽象类的子类，以  
\* Spark的图计算为例，Graph的抽象的class，在object Graph中的apply方法实际上调用了Graph的子类GraphImpl来构建  
\* Graph类型的对象实例的，当然从Spark图计算的源码可以看出，GraphImpl的构造业使用了object GraphImpl的apply方法  
\* 第二：这种方式神奇的效应在于更加能够应对版本迭代或者修改的变化，这是更高意义的面向接口编程；*

|  |
| --- |
| *//是object HellOOP的伴生类，可以直接访问object HellOOP的一切内容* **class** HelloOOP(age:Int,name:String){ *// var name="Spark"* **def** sayHello=*println*(**"Hi,My name is "**+name+**" my name is "**+age) } *//是class HelloOOP的伴生对象，可以直接访问class HelloOOP的一切内容，private[this]是特例* **object** HelloOOP {  **var** *number*=0  **def** main(args:Array[String]):Unit={  *println*(**"Hello Scala OOp!"**)  **val** helloOOP=*HelloOOP*(18)  helloOOP.sayHello  }  **def** apply(age:Int):HelloOOP={  *println*(**"my number is :"**+*number*)  *number*+=1  **new** HelloOOP(age,**""**)  }  **def** apply(age:Int,name:String):HelloOOP={  **new** HelloOOP(age,name)  } } |

## 继承

重载父类方法一定要用**override**

|  |
| --- |
| **class** MapTest2 **extends** MapTest{  **override def** say(): Unit ={  *println*(**"my name is jian"**)  } } |

**override 还可以复写父类的属性**

## 接口trait

cala Trait(特征) 相当于 Java 的接口，实际上它比接口还功能强大。

与接口不同的是，它还可以定义属性和方法的实现。

一般情况下Scala的类只能够继承单一父类，但是如果是 Trait(特征) 的话就可以继承多个，从结果来看就是实现了多重继承。

Trait(特征) 定义的方式与类类似，但它使用的关键字是 **trait**，如下所示：

trait Equal {

def isEqual(x: Any): Boolean

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

以上Trait(特征)由两个方法组成：**isEqual** 和 **isNotEqual**。isEqual 方法没有定义方法的实现，isNotEqual定义了方法的实现。子类继承特征可以实现未被实现的方法。所以其实 Scala Trait(特征)更像 Java 的抽象类。

以下演示了特征的完整实例：

|  |
| --- |
| **trait** Equal {  **def** isEqual(x: Any): Boolean  **def** isNotEqual(x: Any): Boolean = !isEqual(x)} **class** Point(xc: Int, yc: Int) **extends** Equal {  **var** *x*: Int = xc  **var** *y*: Int = yc  **def** isEqual(obj: Any) =  obj.isInstanceOf[Point] &&  obj.asInstanceOf[Point].*x* == *x*} **object** Test {  **def** main(args: Array[String]) {  **val** p1 = **new** Point(2, 3)  **val** p2 = **new** Point(2, 4)  **val** p3 = **new** Point(3, 3)   *println*(p1.isNotEqual(p2))  *println*(p1.isNotEqual(p3))  *println*(p1.isNotEqual(2))  }} |

执行以上代码，输出结果为：

$ scalac Test.scala

$ scala Testfalsetruetrue

### 特征构造顺序

特征也可以有构造器，由字段的初始化和其他特征体中的语句构成。这些语句在任何混入该特征的对象在构造是都会被执行。

构造器的执行顺序：

* 调用超类的构造器；
* 特征构造器在超类构造器之后、类构造器之前执行；
* 特质由左到右被构造；
* 每个特征当中，父特质先被构造；
* 如果多个特征共有一个父特质，父特质不会被重复构造
* 所有特征被构造完毕，子类被构造。

构造器的顺序是类的线性化的反向。线性化是描述某个类型的所有超类型的一种技术规格。