



Scala 教程



Scala 是一门多范式（multi-paradigm）的编程语言，设计初衷是要集成面向对象编程和函数式编程的各种特性。
Scala 运行在Java虚拟机上，并兼容现有的Java程序。
Scala 源代码被编译成Java字节码，所以它可以运行于JVM之上，并可以调用现有的Java类库。

谁适合阅读本教程？

本教程适合想从零开始学习 Scala 编程语言的开发人员。当然本教程也会对一些模块进行深入，让你更好的了解 Scala 的应用。

学习本教程前你需要了解

在继续本教程之前，你应该了解一些基本的计算机编程术语。如果你学习过Java编程语言，将有助于你更快的了解 Scala 编程。
学习 [Java 教程](#)。

第一个 Scala 程序：Hello World

以下是用 Scala 编写的典型 Hello World 程序：

实例（HelloWorld.scala）

```
object HelloWorld {  
  def main(args: Array[String]): Unit = {  
    println("Hello, world!")  
  }  
}
```

运行实例 »

将以上代码保存为 HelloWorld.scala 文件，执行以上 scala 程序（你也可以直接在线执行）：

```
$ scalac HelloWorld.scala  
  
$ scala HelloWorld.scala
```

输出结果为：

```
Hello, world!
```

相关文档推荐

以下是一份 [Scala语言规范.pdf](#) 文档，可作为学习参考：



Scala 简介 

 点我分享笔记

反馈/建议

Copyright © 2013-2018 菜鸟教程 runoob.com All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

 Scala 教程

Scala 安装及环境配置 

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。

谁使用了 Scala

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。
瑞银集团把Scala用于一般产品中。
Coursera把Scala作为服务器语言使用。

Scala Web 框架

以下列出了两个目前比较流行的 Scala 的 Web应用框架：

Lift 框架
Play 框架



Scala 安装

Scala 语言可以运行在Window、Linux、Unix、Mac OS X等系统上。

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

Mac OS X 和 Linux 上安装 Scala

第一步：Java 设置

确保你本地以及安装了 JDK 1.5 以上版本，并且设置了 JAVA_HOME 环境变量及 JDK 的bin目录。

我们可以使用以下命令查看是否安装了 Java:

```
$ java -version

java version "1.8.0_31"

Java(TM) SE Runtime Environment (build 1.8.0_31-b13)

Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)

$
```

接着，我们可以查看是否安装了 Java 编译器。输入以下命令查看：

```
$ javac -version

javac 1.8.0_31

$
```

如果还未安装，可以参考我们的[Java 开发环境配置](#)。

接下来，我们可以从 [Scala 官网地址](#) <http://www.scala-lang.org/downloads> 下载 Scala 二进制包，本教程我们将下载 2.11.7版本，如下图所示：

Choose one of three ways to get started with Scala!



Download Scala 2.11.7 binaries for your system ([All downloads](#)).



二进制包

[Need help installing?](#)

~ or ~

解压缩文件包，可将其移动至/usr/local/share下：

```
mv scala-2.11.7 scala          # 重命名 Scala 目录
mv /download/scalapath /usr/local/share # 下载目录需要按你实际的下载路径
```

修改环境变量，如果不是管理员可使用 **sudo** 进入管理员权限，修改配置文件profile:

```
vim /etc/profile

或

sudo vim /etc/profile
```

在文件的末尾加入：

```
export PATH="$PATH:/usr/local/share/scala/bin"
```

:wq!保存退出，重启终端，执行 **scala** 命令，输出以下信息，表示安装成功：

```
$ scala

Welcome 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.
```

注意：在编译的时候，如果有中文会出现乱码现象，解决方法查看：[Scala 中文乱码解决](#)

window 上安装 Scala

第一步：Java 设置

检测方法前文已说明，这里不再描述。

如果还为安装，可以参考我们的[Java 开发环境配置](#)。

接下来，我们可以从 [Scala 官网地址 http://www.scala-lang.org/downloads](http://www.scala-lang.org/downloads) 下载 [Scala 二进制包](#)(页面底部)，本教程我们将下载 2.11.7版本，如下图所示：

下载后，双击 `msi` 文件，一步步安装即可，安装过程你可以使用默认的安装目录。

安装好 `scala`后，系统会自动提示，单击 `finish`，完成安装。

右击我的电脑，单击"属性"，进入如图所示页面。下面开始配置环境变量，右击【我的电脑】－【属性】－【高级系统设置】－【环境变量】，如图：

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 (x86) \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。

设置 `Path` 变量：找到系统变量下的"Path"如图，单击编辑。在"变量值"一栏的最前面添加如下的路径：`%SCALA_HOME%\bin;%SCALA_HOME%\jre\bin`；

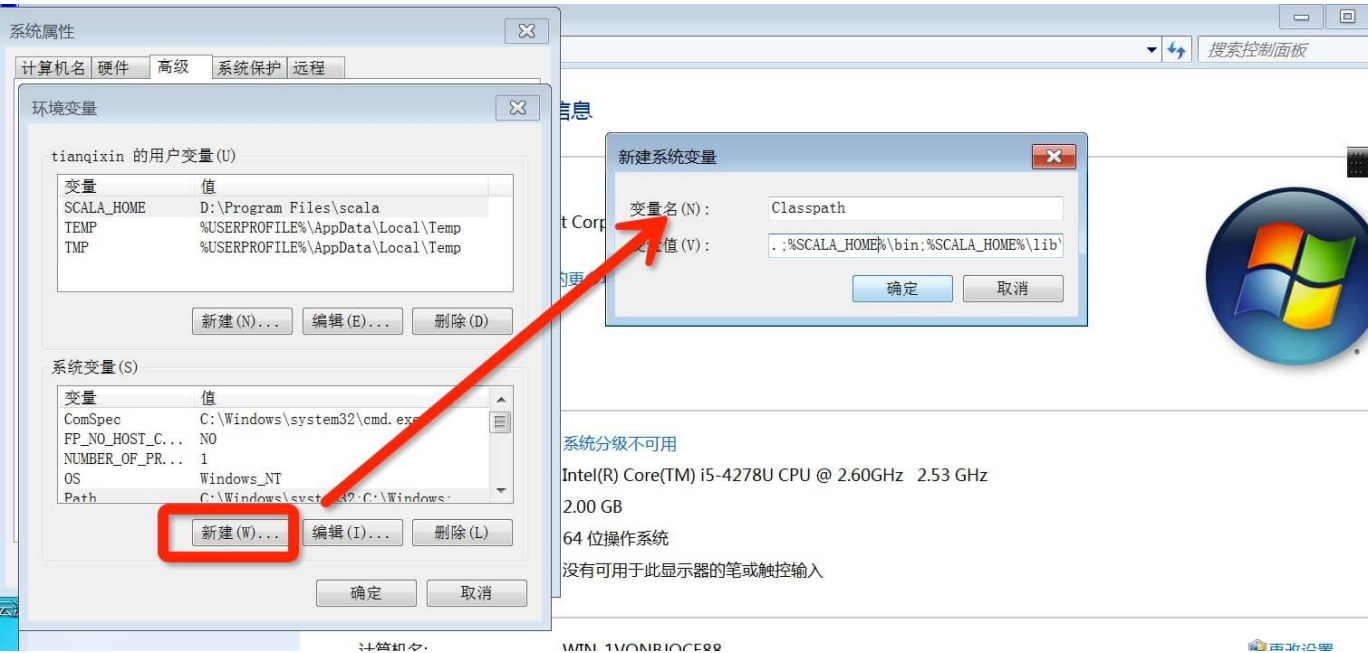
注意：后面的分号； 不要漏掉。

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"：

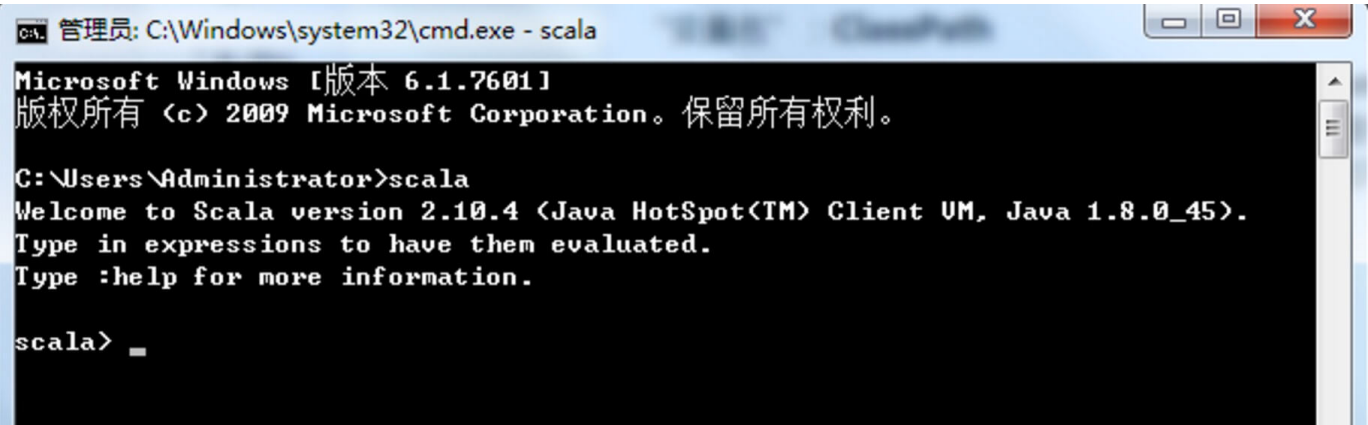
 "变量名"：`ClassPath`

 "变量值"：`.;%SCALA_HOME%\bin;%SCALA_HOME%\lib\dt.jar;%SCALA_HOME%\lib\tools.jar;`

注意："变量值"最前面的 `;`；不要漏掉。最后单击确定即可。



检查环境变量是否设置好了：调出"cmd"检查。单击【开始】，在输入框中输入 `cmd`，然后"回车"，输入 `scala`，然后回车，如环境变量设置ok，你应该能看到这些信息。



以下列出了不同系统放置的目录（可作为参考）：

系统环境	变量	值 (举例)
------	----	--------

系统环境	变量	值 (举例)
Unix	\$SCALA_HOME	/usr/local/share/scala
	\$PATH	\$PATH:\$SCALA_HOME/bin
Windows	%SCALA_HOME%	c:\Progra~1\Scala
	%PATH%	%PATH%;%SCALA_HOME%\bin

Scala 简介

Scala 基础语法



2 篇笔记
#2

写笔记



Linux中修改/etc/profile文件后不会在当前会话窗口立马生效。
常用的生效方式有以下三种：
重启
再(重新)打开一个terminal窗口
source /etc/profile

ABCDer_Bi1年前 (2017-09-12)

#1



目前官网提供的下载版本是2.12.4。windows配置环境变量有部分变化；
1.设置 SCALA_HOME 变量,同上；
2.设置 Path 变量：只需要设置%SCALA_HOME%bin;即可
3.设置 Classpath 变量："变量值": .;%SCALA_HOME%bin;即可

带着代码走天涯8个月前 (02-01)

反馈/建议

Copyright © 2013-2018 菜鸟教程 runoob.com All Rights Reserved. 备案号：闽ICP备15012807号-1



首页 HTML CSS JS 本地书签

Scala 安装及环境配置

Scala 数据类型

Scala 基础语法

如果你之前是一名 Java 程序员，并了解 Java 语言的基础知识，那么你能很快学会 Scala 的基础语法。

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：

对象 - 对象有属性和行为。例如：一只狗的状态属性有：颜色，名字，行为有：叫、跑、吃等。对象是一个类的实例。

类 - 类是对象的抽象，而对象是类的具体实例。

方法 - 方法描述的基本的行为，一个类可以包含多个方法。

字段 - 每个对象都有它唯一的实例变量集合，即字段。对象的属性通过给字段赋值来创建。

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：

```
$ scala

Welcome 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

$ ls

HelloWorld$.class  HelloWorld.scala
```



```
HelloWorld.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 HelloWorld
```

```
Hello, world!
```

[在线实例 »](#)

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(新版本不需要了，但建议保留这种习惯)。

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

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** 支持单行和多行注释。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包中。 比如：

```
package com.runoob

class 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 安装及环境配置](#)

[Scala 数据类型](#)

[点我分享笔记](#)

反馈/建议

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Scala 基础语法](#)

[Scala 变量](#)

Scala 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, IEEE 754 标准的单精度浮点数
Double	64 位 IEEE 754 标准的双精度浮点数
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。：

```
0

035

21

0xFFFFFFFF

0777L
```

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：

```
0.0

1e30f

3.14159f

1.0e100

.1
```

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** 字符变量使用单引号 `'` 来定义，如下：

```
'a'

'\u0041'

'\n'

'\t'
```

其中 `\` 表示转义字符，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

在 **Scala** 字符串变量使用双引号 `"` 来定义，如下：

```
"Hello,\nWorld!"

"菜鸟教程官网：www.runoob.com"
```

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：`""" ... """`。

实例如下：

```
val foo = """菜鸟教程

www.runoob.com

www.w3cschool.cc

www.runnoob.com

以上三个地址都能访问"""
```

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
<code>\b</code>	<code>\u0008</code>	退格(BS)，将当前位置移到前一系列
<code>\t</code>	<code>\u0009</code>	水平制表(HT)（跳到下一个TAB位置）
<code>\n</code>	<code>\u000a</code>	换行(LF)，将当前位置移到下一行开头
<code>\f</code>	<code>\u000c</code>	换页(FF)，将当前位置移到下页开头
<code>\r</code>	<code>\u000d</code>	回车(CR)，将当前位置移到本行开头
<code>\"</code>	<code>\u0022</code>	代表一个双引号(")字符
<code>\'</code>	<code>\u0027</code>	代表一个单引号 (') 字符
<code>\\</code>	<code>\u005c</code>	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。

以下实例演示了一些转义字符的使用：

```
object Test {  
  
    def main(args: Array[String]) {  
  
        println("Hello\tWorld\n\n" );  
  
    }  
  
}
```

运行实例 »

执行以上代码输出结果如下所示：

```
$ scalac Test.scala  
  
$ scala Test  
  
Hello      World
```



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，小数或者字母。

变量声明

在学习如何声明变量与常量之前，我们先来了解一些变量与常量。

- 一、变量： 在程序运行过程中其值可能发生改变的量叫做变量。如：时间，年龄。

二、常量 在程序运行过程中其值不会发生变化的量叫做常量。如：数值 3，字符'A'。

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

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]
```

或

```
val VariableName : DataType [= Initial Value]
```

变量声明一定需要初始值，否则会报错。

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：


```
scala> val pa = (40,"Foo")

pa: (Int, String) = (40, Foo)
```

[Scala 数据类型](#)[Scala 访问修饰符](#)[点我分享笔记](#)[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1

[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)[Scala 变量](#)[Scala 运算符](#)

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，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象可见外,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 bobsrockets{

  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`。

[Scala 变量](#)

Scala 运算符 [Scala 运算符](#)

[点我分享笔记](#)

反馈/建议



Scala 运算符

一个运算符是一个符号，用于告诉编译器来执行指定的数学运算和逻辑运算。

Scala 含有丰富的内置运算符，包括以下几种类型：

算术运算符

关系运算符

逻辑运算符

位运算符

赋值运算符

接下来我们将为大家详细介绍以上各种运算符的应用。

算术运算符

下表列出了 Scala 支持的算术运算符。

假定变量 A 为 10，B 为 20：

运算符	描述	实例
+	加号	A + B 运算结果为 30
-	减号	A - B 运算结果为 -10
*	乘号	A * B 运算结果为 200
/	除号	B / A 运算结果为 2
%	取余	B % A 运算结果为 0

实例

```
object Test {  
  
  def main(args: Array[String]) {  
  
    var a = 10;  
  
    var b = 20;  
  
    var c = 25;  
  
    var d = 25;  
  
    println("a + b = " + (a + b) );  
  
    println("a - b = " + (a - b) );  
  
  }  
}
```

```
println("a * b = " + (a * b) );

println("b / a = " + (b / a) );

println("b % a = " + (b % a) );

println("c % a = " + (c % a) );


}

}
```

运行实例 »

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

a + b = 30

a - b = -10

a * b = 200

b / a = 2

b % a = 0

c % a = 5
```

关系运算符

下表列出了 **Scala** 支持的关系运算符。

假定变量 **A** 为 10, **B** 为 20:

运算符	描述	实例
==	等于	(A == B) 运算结果为 false
!=	不等于	(A != B) 运算结果为 true
>	大于	(A > B) 运算结果为 false
<	小于	(A < B) 运算结果为 true
>=	大于等于	(A >= B) 运算结果为 false
<=	小于等于	(A <= B) 运算结果为 true

实例

```
object Test {

    def main(args: Array[String]) {
```

```
var a = 10;

var b = 20;

println("a == b = " + (a == b) );

println("a != b = " + (a != b) );

println("a > b = " + (a > b) );

println("a < b = " + (a < b) );

println("b >= a = " + (b >= a) );

println("b <= a = " + (b <= a) );

}

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

a == b = false

a != b = true

a > b = false

a < b = true

b >= a = true

b <= a = false
```

逻辑运算符

下表列出了 **Scala** 支持的逻辑运算符。

假定变量 **A** 为 1，**B** 为 0:

运算符	描述	实例
&&	逻辑与	(A && B) 运算结果为 false
	逻辑或	(A B) 运算结果为 true
!	逻辑非	!(A && B) 运算结果为 true

实例

```
object Test {

  def main(args: Array[String]) {

    var a = true;

    var b = false;
```

```

println("a && b = " + (a&&b) );

println("a || b = " + (a||b) );

println("!(a && b) = " + !(a && b) );

}

}

```

执行以上代码，输出结果为：

```

$ scalac Test.scala

$ scala Test

a && b = false

a || b = true

!(a && b) = true

```

位运算符

位运算符用来对二进制位进行操作， \sim , $\&$, $|$, \wedge 分别为取反，按位与与，按位与或，按位与异或运算，如下表实例：

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

如果指定 A = 60; 及 B = 13; 两个变量对应的二进制为：

```
A = 0011 1100
```

```
B = 0000 1101
```

```
-----位运算-----
```

```
A&B = 0000 1100
```

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

Scala 中的按位运算法则如下：

运算符	描述	实例
&	按位与运算符	(a & b) 输出结果 12，二进制解释：0000 1100
	按位或运算符	(a b) 输出结果 61，二进制解释：0011 1101
^	按位异或运算符	(a ^ b) 输出结果 49，二进制解释：0011 0001
~	按位取反运算符	(~a) 输出结果 -61，二进制解释：1100 0011， 在一个有符号二进制数的补码形式。
<<	左移动运算符	a << 2 输出结果 240，二进制解释：1111 0000
>>	右移动运算符	a >> 2 输出结果 15，二进制解释：0000 1111
>>>	无符号右移	A >>> 2 输出结果 15, 二进制解释: 0000 1111

实例

```
object Test {  
  
  def main(args: Array[String]) {  
  
    var a = 60;          /* 60 = 0011 1100 */  
  
    var b = 13;          /* 13 = 0000 1101 */  
  
    var c = 0;  
  
  
    c = a & b;           /* 12 = 0000 1100 */  
  
    println("a & b = " + c );  
  
  
    c = a | b;           /* 61 = 0011 1101 */  
  
    println("a | b = " + c );  
  
  
    c = a ^ b;           /* 49 = 0011 0001 */  
  
    println("a ^ b = " + c );  
  
  
    c = ~a;              /* -61 = 1100 0011 */  
  
  }  
}
```



```
println("~a = " + c );

c = a << 2;          /* 240 = 1111 0000 */

println("a << 2 = " + c );

c = a >> 2;          /* 15 = 1111 */

println("a >> 2 = " + c );

c = a >>> 2;         /* 15 = 0000 1111 */

println("a >>> 2 = " + c );

}

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

a & b = 12

a | b = 61

a ^ b = 49

~a = -61

a << 2 = 240

a >> 2 = 15

a >>> 2 = 15
```

赋值运算符

以下列出了 **Scala** 语言支持的赋值运算符：

运算符	描述	实例
=	简单的赋值运算，指定右边操作数赋值给左边的操作数。	C = A + B 将 A + B 的运算结果赋值给 C
+=	相加后再赋值，将左右两边的操作数相加后再赋值给左边的操作数。	C += A 相当于 C = C + A
-=	相减后再赋值，将左右两边的操作数相减后再赋值给左边的操作数。	C -= A 相当于 C = C - A
*=	相乘后再赋值，将左右两边的操作数相乘后再赋值给左边的操作数。	C *= A 相当于 C = C * A
/=	相除后再赋值，将左右两边的操作数相除后再赋值给左边的操作数。	C /= A 相当于 C = C / A

%=	求余后再赋值，将左右两边的操作数求余后再赋值给左边的操作数。	C %= A is equivalent to C = C % A
<<=	按位左移后再赋值	C <<= 2 相当于 C = C << 2
>>=	按位右移后再赋值	C >>= 2 相当于 C = C >> 2
&=	按位与运算后赋值	C &= 2 相当于 C = C & 2
^=	按位异或运算符后再赋值	C ^= 2 相当于 C = C ^ 2
=	按位或运算后再赋值	C = 2 相当于 C = C 2

实例

```
object Test {

  def main(args: Array[String]) {

    var a = 10;

    var b = 20;

    var c = 0;

    c = a + b;

    println("c = a + b = " + c );

    c += a ;

    println("c += a = " + c );

    c -= a ;

    println("c -= a = " + c );

    c *= a ;

    println("c *= a = " + c );

    a = 10;

    c = 15;

    c /= a ;

    println("c /= a = " + c );

    a = 10;
```

```

    c = 15;

    c %= a ;

    println("c %= a  = " + c );

    c <<= 2 ;

    println("c <<= 2  = " + c );

    c >>= 2 ;

    println("c >>= 2  = " + c );

    c >>= 2 ;

    println("c >>= a  = " + c );

    c &= a ;

    println("c &= 2  = " + c );

    c ^= a ;

    println("c ^= a  = " + c );

    c |= a ;

    println("c |= a  = " + c );
}
}

```

执行以上代码，输出结果为：

```

$ scalac Test.scala

$ scala Test

c = a + b  = 30

c += a  = 40

c -= a = 30

c *= a = 300

c /= a  = 1

```

```
c %= a = 5

c <<= 2 = 20

c >>= 2 = 5

c >>= a = 1

c &= 2 = 0

c ^= a = 10

c |= a = 10
```

运算符优先级取决于所属的运算符组，它会影响算式的的计算。

实例： $x = 7 + 3 * 2$; 这里， x 计算结果为 13, 而不是 20, 因为乘法（*）高于加法（+），所以它先计算 $3*2$ 再加上 7。

查看以下表格，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。

类别	运算符	关联性
1	() []	左到右
2	! ~	右到左
3	* / %	左到右
4	+ -	左到右
5	>> >>> <<	左到右
6	> >= < <=	左到右
7	== !=	左到右
8	&	左到右
9	^	左到右
10		左到右
11	&&	左到右
12		左到右
13	= += -= *= /= %= >>= <<= &= ^= =	右到左
14	,	左到右

☐ Scala 访问修饰符

Scala IF...ELSE 语句 ☐

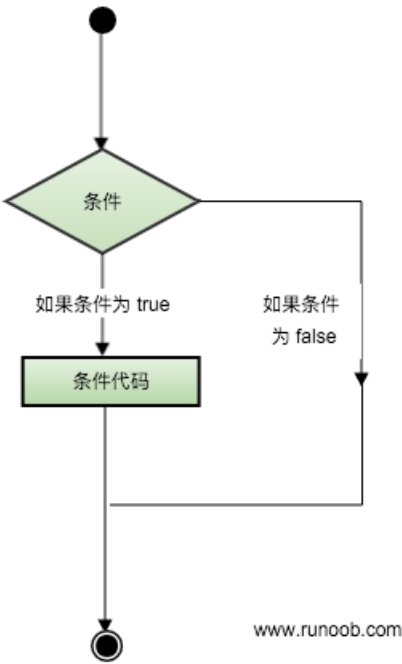
☐ 点我分享笔记

反馈/建议

Scala IF...ELSE 语句

Scala IF...ELSE 语句是通过一条或多条语句的执行结果（True或者False）来决定执行的代码块。

可以通过下图来简单了解条件语句的执行过程：



if 语句

if 语句有布尔表达式及之后的语句块组成。

语法

if 语句的语法格式如下：

```
if(布尔表达式)
{
    // 如果布尔表达式为 true 则执行该语句块
}
```

如果布尔表达式为 **true** 则执行大括号内的语句块，否则跳过大括号内的语句块，执行大括号之后的语句块。

实例

```
object Test {

    def main(args: Array[String]) {

        var x = 10;
```

```
    if( x < 20 ){

        println("x < 20");

    }

}

}
```

运行实例 »

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

x < 20
```

if...else 语句

if 语句后可以紧跟 **else** 语句，**else** 内的语句块可以在布尔表达式为 **false** 的时候执行。

语法

if...**else** 的语法格式如下：

```
if(布尔表达式){

    // 如果布尔表达式为 true 则执行该语句块

}else{

    // 如果布尔表达式为 false 则执行该语句块

}
```

实例

```
object Test {

    def main(args: Array[String]) {

        var x = 30;

        if( x < 20 ){

            println("x 小于 20");

        }else{

            println("x 大于 20");

        }

    }

}
```

```
}  
  
}  
  
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala  
  
$ scala Test  
  
x 大于 20
```

if...else if...else 语句

if 语句后可以紧跟 **else if...else** 语句，在多个条件判断语句的情况下很有用。

语法

if...else if...else 语法格式如下：

```
if(布尔表达式 1){  
  
    // 如果布尔表达式 1 为 true 则执行该语句块  
  
}else if(布尔表达式 2){  
  
    // 如果布尔表达式 2 为 true 则执行该语句块  
  
}else if(布尔表达式 3){  
  
    // 如果布尔表达式 3 为 true 则执行该语句块  
  
}else {  
  
    // 如果以上条件都为 false 执行该语句块  
  
}
```

实例

```
object Test {  
  
    def main(args: Array[String]) {  
  
        var x = 30;  
  
  
        if( x == 10 ){  
  
            println("X 的值为 10");  
  
        }else if( x == 20 ){
```

```
println("X 的值为 20");

}else if( x == 30 ){

    println("X 的值为 30");

}else{

    println("无法判断 x 的值");

}

}

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

X 的值为 30
```

if...else 嵌套语句

if...else 嵌套语句可以实现在 if 语句内嵌入一个或多个 if 语句。

语法

if...else 嵌套语句语法格式如下：

```
if(布尔表达式 1){

    // 如果布尔表达式 1 为 true 则执行该语句块

    if(布尔表达式 2){

        // 如果布尔表达式 2 为 true 则执行该语句块

    }

}
```

else if...else 的嵌套语句 类似 if...else 嵌套语句。

实例

```
object Test {

    def main(args: Array[String]) {

        var x = 30;

        var y = 10;
```



```
    if( x == 30 ){

        if( y == 10 ){

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

        }

    }

}

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

X = 30 , Y = 10
```

[☐ Scala 运算符](#)

[Scala 循环](#) ☐

[☐ 点我分享笔记](#)

反馈/建议

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

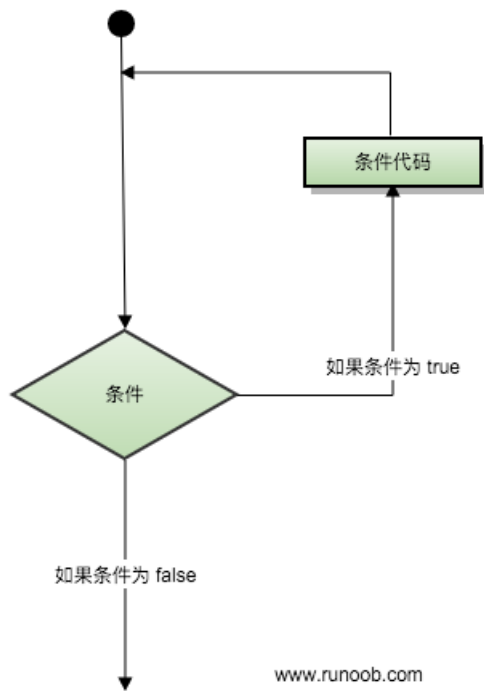
[☐ Scala IF...ELSE 语句](#)

[Scala while 循环](#) ☐

Scala 循环

有的时候，我们可能需要多次执行同一块代码。一般情况下，语句是按顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。编程语言提供了更为复杂执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的流程图：



循环类型

Scala 语言提供了以下几种循环类型。点击链接查看每个类型的细节。

循环类型	描述
while 循环	运行一系列语句，如果条件为true，会重复运行，直到条件变为false。
do..while 循环	类似 while 语句区别在于判断循环条件之前，先执行一次循环的代码块。
for 循环	用来重复执行一系列语句直到达成特定条件达成，一般通过在每次循环完成后增加计数器的值来实现。

循环控制语句

循环控制语句改变你代码的执行顺序，通过它你可以实现代码的跳转。Scala 以下几种循环控制语句：

Scala 不支持 break 或 continue 语句，但从 2.8 版本后提供了一种中断循环的方式，点击以下链接查看详情。

控制语句	描述
break 语句	中断循环

无限循环

如果条件永远为 true，则循环将变成无限循环。我们可以使用 while 语句来实现无限循环：

```
object Test {

  def main(args: Array[String]) {

    var a = 10;

    // 无限循环

    while( true ){

      println( "a 的值为 : " + a );

    }

  }

}
```

```
}
```

以上代码执行后循环会永久执行下去，你可以使用 **Ctrl + C** 键来中断无限循环。

[Scala IF...ELSE 语句](#)

[Scala while 循环](#)

[点我分享笔记](#)

[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1

□

[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Scala break 语句](#)

[Scala 函数传名调用](#)

Scala 方法与函数

Scala 有方法与函数，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。换句话说来说在类中定义的函数即是方法。

Scala 中的方法跟 **Java** 的类似，方法是组成类的一部分。

Scala 中的函数则是一个完整的对象，**Scala** 中的函数其实就是继承了 **Trait** 的类的对象。

Scala 中使用 **val** 语句可以定义函数，**def** 语句定义方法。

```
class Test{

    def m(x: Int) = x + 3

    val f = (x: Int) => x + 3

}
```

注意：有些翻译上函数(function)与方法(method)是没有区别的。

方法声明

Scala 方法声明格式如下：

```
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。

方法定义

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。

Scala 方法定义格式如下：

```
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。

以下方法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  
    }  
  
}
```

如果方法没有返回值，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!")  
    }  
  
}
```

方法调用

Scala 提供了多种不同的方法调用方式：

以下是调用方法的标准格式：

```
functionName( 参数列表 )
```

如果方法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方法的实例：

```
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 Test

Returned Value : 12
```

Scala 也是一种函数式语言，所以函数是 Scala 语言的核心。以下一些函数概念有助于我们更好的理解 Scala 编程：

函数概念解析接案例	
函数传名调用(Call-by-Name)	指定函数参数名
函数 - 可变参数	递归函数
默认参数值	高阶函数
内嵌函数	匿名函数
偏应用函数	函数柯里化(Function Currying)

❏

1 篇笔记

#1

❏

方法和函数的区别

1、函数可作为一个参数传入到方法中，而方法不行。

❏ 写笔记

```
scala> def m2(f: (Int, Int) => Int) = f(2,6)
m2: (f: (Int, Int) => Int)Int 1.定义一个方法

scala> val f2 = (x: Int, y: Int) => x - y
f2: (Int, Int) => Int = <function2> 2. 定义一个函数

scala> m2(f2) 3.将函数作为参数传入到方法中
res0: Int = -4
```

```
object MethodAndFunctionDemo {

    //定义一个方法

    //方法 m1 参数要求是一个函数，函数的参数必须是两个Int类型

    //返回值类型也是Int类型

    def m1(f:(Int,Int) => Int) : Int = {

        f(2,6)

    }

    //定义一个函数f1,参数是两个Int类型，返回值是一个Int类型

    val f1 = (x:Int,y:Int) => x + y

    //再定义一个函数f2

    val f2 = (m:Int,n:Int) => m * n


    //main方法

    def main(args: Array[String]): Unit = {

        //调用m1方法，并传入f1函数

        val r1 = m1(f1)

        println(r1)

        //调用m1方法，并传入f2函数

        val r2 = m1(f2)

        println(r2)

    }

}
```

运行结果：

8

12

2、在Scala中无法直接操作方法，如果要操作方法，必须先将其转换成函数。有两种方法可以将方法转换成函数：

```
val f1 = m _
```

在方法名称m后面紧跟一个空格和下划线告诉编译器将方法m转换成函数，而不是要调用这个方法。也可以显示地告诉编译器需要将方法转换成函数：

```
val f1: (Int) => Int = m
```

通常情况下编译器会自动将方法转换成函数，例如在一个应该传入函数参数的地方传入了一个方法，编译器会自动将传入的方法转换成函数。

```
scala> def m1(x: Int, y: Int) : Int = x * y
m1: (x: Int, y: Int)Int 方法

scala> val f1 = m1 _
f1: (Int, Int) => Int = <function2> 函数

scala> _
```

神奇的下划线将m1这个方法变成了函数

```
object TestMap {
```

```
def ttt(f: Int => Int): Unit = {
```

```
    val r = f(10)
```

```
    println(r)
```

```
}
```

```
val f0 = (x : Int) => x * x
```

```
//定义了一个方法
```

```
def m0(x: Int) : Int = {
```

```
    //传递进来的参数乘以10
```

```
    x * 10
```

```
}
```

```
//将方法转换成函数，利用了神奇的下划线
```

```
val f1 = m0 _
```

```
def main(args: Array[String]): Unit = {

    ttt(f0)

    //通过m0 _将方法转化成函数

    ttt(m0 _);

    //如果直接传递的是方法名称，scala相当于把方法转成了函数

    ttt(m0)

    //通过x => m0(x)的方式将方法转化成函数,这个函数是一个匿名函数，等价：(x:Int) => m0(x)

    ttt(x => m0(x))

}

}
```

输出结果为：

```
100

100

100

100
```

3、函数必须要有参数列表，而方法可以没有参数列表

```
scala> def m1 = 100
m1: Int

scala> def m2<>=100
m2: <>Int

scala> val f1 =<>=>100
f1: <> => Int = <function0>

scala> val f1 = =>100
<console>:1: error: illegal start of simple expression
    val f1 = =>100
              ^
```

4、在函数出现的地方我们可以提供一个方法

在需要函数的地方，如果传递一个方法，会自动进行ETA展开（把方法转换为函数）

```
scala> def m<x:Int>:Int=x+1
m: <x: Int>Int

scala> val f1=m
<console>:8: error: missing arguments for method m;
follow this method with '_' if you want to treat it as a partially applied function
    val f1=m
          ^
```

如果我们直接把一个方法赋值给变量会报错。如果我们指定变量的类型就是函数，那么就可以通过编译，如下：

```
scala> val f1:<Int>=>Int = m
f1: Int => Int = <function1>
```

当然我们也可以强制把一个方法转换给函数，这就用到了 scala 中的部分应用函数：


```
scala> val f1=m _  
f1: Int => Int = <function1>
```

tianqixin4个月前 (05-28)

反馈/建议

Copyright © 2013-2018 菜鸟教程 runoob.com All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Scala 函数柯里化\(Currying\)](#)

[Scala 字符串](#)

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( "multiplier(1) value = " + multiplier(1) )  
  
    println( "multiplier(2) value = " + multiplier(2) )  
  
  }  
}
```

```
}

var factor = 3

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

}
```

运行实例 »

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

multiplier(1) value = 3

multiplier(2) value = 6
```

[Scala 函数柯里化\(Currying\)](#)

[Scala 字符串](#)

[点我分享笔记](#)

反馈/建议

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Scala 闭包](#)

[Scala 数组](#)

Scala 字符串

以下实例将字符串赋值给一个常量：

```
object Test {

    val greeting: String = "Hello,World!"

    def main(args: Array[String]) {

        println( greeting )

    }

}
```

```
}
```

以上实例定义了变量 `greeting`，为字符串常量，它的类型为 `String (java.lang.String)`。

在 `Scala` 中，字符串的类型实际上是 `Java String`，它本身没有 `String` 类。

在 `Scala` 中，`String` 是一个不可变的对象，所以该对象不可被修改。这就意味着你如果修改字符串就会产生一个新的字符串对象。

但其他对象，如数组就是可变的对象。接下来我们会为大家介绍常用的 `java.lang.String` 方法。

创建字符串

创建字符串实例如下：

```
var greeting = "Hello World!";
```

或

```
var greeting:String = "Hello World!";
```

你不一定为字符串指定 `String` 类型，因为 `Scala` 编译器会自动推断出字符串的类型为 `String`。

当然我们也可以直接显示的声明字符串为 `String` 类型，如下实例：

```
object Test {  
  
    val greeting: String = "Hello, World!"  
  
    def main(args: Array[String]) {  
  
        println( greeting )  
  
    }  
  
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala  
  
$ scala Test  
  
Hello, world!
```

我们前面提到过 `String` 对象是不可变的，如果你需要创建一个可以修改的字符串，可以使用 `String Builder` 类，如下实例：

```
object Test {  
  
    def main(args: Array[String]) {
```

```
    val buf = new StringBuilder;

    buf += 'a'

    buf += "bcdef"

    println( "buf is : " + buf.toString );

}

}
```

运行实例 »

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

buf is : abcdef
```

字符串长度

我们可以使用 **length()** 方法来获取字符串长度：

```
object Test {

    def main(args: Array[String]) {

        var palindrome = "www.runoob.com";

        var len = palindrome.length();

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

    }

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

String Length is : 14
```

字符串连接

String 类中使用 **concat()** 方法来连接两个字符串：

```
string1.concat(string2);
```

实例演示：

```
scala> "菜鸟教程官网：".concat("www.runoob.com");

res0: String = 菜鸟教程官网： www.runoob.com
```

同样你也可以使用加号(+)来连接：

```
scala> "菜鸟教程官网： " + " www.runoob.com"

res1: String = 菜鸟教程官网： www.runoob.com
```

让我们看个完整实例：

```
object Test {

  def main(args: Array[String]) {

    var str1 = "菜鸟教程官网：";

    var str2 = "www.runoob.com";

    var str3 = "菜鸟教程的 Slogan 为：";

    var str4 = "学的不仅是技术，更是梦想！";

    println( str1 + str2 );

    println( str3.concat(str4) );

  }

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

菜鸟教程官网：www.runoob.com

菜鸟教程的 Slogan 为：学的不仅是技术，更是梦想！
```

创建格式化字符串

`String` 类中你可以使用 `printf()` 方法来格式化字符串并输出，`String format()` 方法可以返回 `String` 对象而不是 `PrintStream` 对象。以下实例演示了 `printf()` 方法的使用：

```
object Test {
```

```
def main(args: Array[String]) {  
  
    var floatVar = 12.456  
  
    var intVar = 2000  
  
    var stringVar = "菜鸟教程!"  
  
    var fs = printf("浮点型变量为 " +  
  
                    "%f, 整型变量为 %d, 字符串为 " +  
  
                    " %s", floatVar, intVar, stringVar)  
  
    println(fs)  
  
}  
  
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala  
  
$ scala Test  
  
浮点型变量为 12.456000, 整型变量为 2000, 字符串为  菜鸟教程!()
```

String 方法

下表列出了 java.lang.String 中常用的方法，你可以在 Scala 中使用：

序号	方法及描述
1	char charAt(int index) 返回指定位置的字符
2	int compareTo(Object o) 比较字符串与对象
3	int compareTo(String anotherString) 按字典顺序比较两个字符串
4	int compareToIgnoreCase(String str) 按字典顺序比较两个字符串，不考虑大小写
5	String concat(String str) 将指定字符串连接到此字符串的结尾
6	boolean contentEquals(StringBuffer sb) 将此字符串与指定的 StringBuffer 比较。
7	static String copyValueOf(char[] data) 返回指定数组中表示该字符序列的 String
8	static String copyValueOf(char[] data, int offset, int count) 返回指定数组中表示该字符序列的 String

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) 将此字符串与指定的对象比较
11	boolean equalsIgnoreCase(String anotherString) 将此 <code>String</code> 与另一个 <code>String</code> 比较，不考虑大小写
12	byte getBytes() 使用平台的默认字符集将此 <code>String</code> 编码为 <code>byte</code> 序列，并将结果存储到一个新的 <code>byte</code> 数组中
13	byte[] getBytes(String charsetName) 使用指定的字符集将此 <code>String</code> 编码为 <code>byte</code> 序列，并将结果存储到一个新的 <code>byte</code> 数组中
14	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) 将字符从此字符串复制到目标字符数组
15	int hashCode() 返回此字符串的哈希码
16	int indexOf(int ch) 返回指定字符在此字符串中第一次出现处的索引
17	int indexOf(int ch, int fromIndex) 返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索
18	int indexOf(String str) 返回指定子字符串在此字符串中第一次出现处的索引
19	int indexOf(String str, int fromIndex) 返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始
20	String intern() 返回字符串对象的规范化表示形式
21	int lastIndexOf(int ch) 返回指定字符在此字符串中最后一次出现处的索引
22	int lastIndexOf(int ch, int fromIndex) 返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行反向搜索
23	int lastIndexOf(String str) 返回指定子字符串在此字符串中最右边出现处的索引
24	int lastIndexOf(String str, int fromIndex) 返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索
25	int length() 返回此字符串的长度
26	boolean matches(String regex) 告知此字符串是否匹配给定的正则表达式
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) 返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的
30	String replaceAll(String regex, String replacement) 使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串
31	String replaceFirst(String regex, String replacement) 使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串
32	String[] split(String regex) 根据给定正则表达式的匹配拆分此字符串
33	String[] split(String regex, int limit) 根据匹配给定的正则表达式来拆分此字符串
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) 测试此字符串从指定索引开始的子字符串是否以指定前缀开始。
36	CharSequence subSequence(int beginIndex, int endIndex) 返回一个新的字符序列，它是此序列的一个子序列
37	String substring(int beginIndex) 返回一个新的字符串，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() 将此字符串转换为一个新的字符数组
40	String toLowerCase() 使用默认语言环境的规则将此 String 中的所有字符都转换为小写
41	String toLowerCase(Locale locale) 使用给定 Locale 的规则将此 String 中的所有字符都转换为小写
42	String toString() 返回此对象本身（它已经是一个字符串！）
43	String toUpperCase() 使用默认语言环境的规则将此 String 中的所有字符都转换为大写
44	String toUpperCase(Locale locale) 使用给定 Locale 的规则将此 String 中的所有字符都转换为大写
45	String trim() 删除指定字符串的首尾空白符
46	static String valueOf(primitive data type x) 返回指定类型参数的字符串表示形式

Scala 数组

Scala 语言中提供的数组是用来存储固定大小的同类型元素，数组对于每一门编辑应语言来说都是重要的数据结构之一。

声明数组变量并不是声明 `number0`、`number1`、...、`number99` 一个个单独的变量，而是声明一个就像 `numbers` 这样的变量，然后使用 `numbers[0]`、`numbers[1]`、...、`numbers[99]` 来表示一个个单独的变量。数组中某个指定的元素是通过索引来访问的。

数组的第一个元素索引为0，最后一个元素的索引为元素总数减1。

声明数组

以下是 Scala 数组声明的语法格式：

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

或

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

以上语法中，`z` 声明一个字符串类型的数组，数组长度为 3，可存储 3 个元素。我们可以为每个元素设置值，并通过索引来访问每个元素，如下所示：

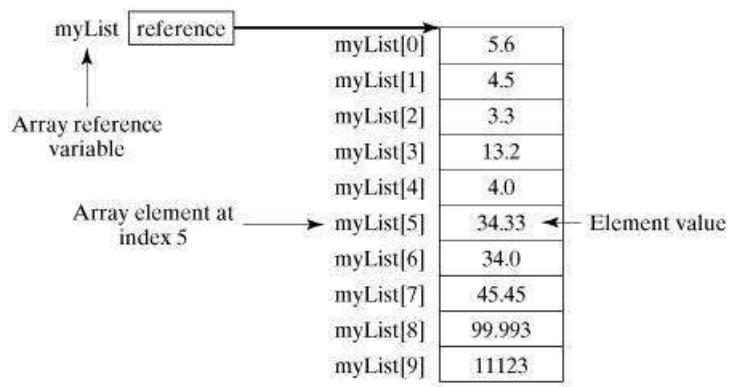
```
z(0) = "Runoob"; z(1) = "Baidu"; z(4/2) = "Google"
```

最后一个元素的索引使用了表达式 `4/2` 作为索引，类似于 `z(2) = "Google"`。

我们也可以使用以下方式来定义一个数组：

```
var z = Array("Runoob", "Baidu", "Google")
```

下图展示了一个长度为 10 的数组 `myList`，索引值为 0 到 9：



处理数组

数组的元素类型和数组的大小都是确定的，所以当处理数组元素时候，我们通常使用基本的 **for** 循环。

以下实例演示了数组的创建，初始化等处理过程：

```
object Test {  
  
  def main(args: Array[String]) {  
  
    var myList = Array(1.9, 2.9, 3.4, 3.5)  
  
    // 输出所有数组元素  
  
    for ( x <- myList ) {  
  
      println( x )  
  
    }  
  
    // 计算数组所有元素的总和  
  
    var total = 0.0;  
  
    for ( i <- 0 to (myList.length - 1)) {  
  
      total += myList(i);  
  
    }  
  
    println("总和为 " + total);  
  
    // 查找数组中的最大元素  
  
    var max = myList(0);  
  
    for ( i <- 1 to (myList.length - 1) ) {  
  
      if (myList(i) > max) max = myList(i);  
  
    }  
  
    println("最大值为 " + max);  
  
  }  
}
```

```
}  
  
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala  
  
$ scala Test  
  
1.9  
2.9  
3.4  
3.5  
  
总和为 11.7  
  
最大值为 3.5
```

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。

以上是一个定义了二维数组的实例：

```
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。

接下来我们来看一个二维数组处理的完整实例：

```
import Array._  
  
object Test {  
  
  def main(args: Array[String]) {  
  
    var myMatrix = ofDim[Int](3,3)  
  
    // 创建矩阵  
  
    for (i <- 0 to 2) {  
  
      for ( j <- 0 to 2) {  
  
        myMatrix(i)(j) = j;  
  
      }  
  
    }  
  
  }  
  
}
```

```
// 打印二维阵列

for (i <- 0 to 2) {

    for ( j <- 0 to 2) {

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

    }

    println();

}

}
```

运行实例 »

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

0 1 2

0 1 2

0 1 2
```

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 )

}

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

1.9

2.9

3.4

3.5

8.9

7.9

0.4

1.5
```

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]) {

        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：

```
$ scalac Test.scala

$ scala Test

10 12 14 16 18

10 11 12 13 14 15 16 17 18 19
```

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 : <div><pre>scala> Array.iterate(0,3)(a=>a+1) res1: Array[Int] = Array(0, 1, 2)</pre></div>
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]] 返回二维数组，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： <pre>scala> Array.tabulate(3)(a => a + 5) res0: Array[Int] = Array(5, 6, 7)</pre>
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 字符串](#)

[Scala Collection\(集合\) ☐](#)

[☐ 点我分享笔记](#)

[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[☐ Scala 数组](#)

[Scala List\(列表\) ☐](#)

Scala Collection

Scala提供了一套很好的集合实现，提供了一些集合类型的抽象。

Scala 集合分为可变的和不可变的集合。

可变集合可以在适当的地方被更新或扩展。这意味着你可以修改，添加，移除一个集合的元素。

而不可变集合类，相比之下，永远不会改变。不过，你仍然可以模拟添加，移除或更新操作。但是这些操作将在每一种情况下都返回一个新的集合，同时使原来的集合不发生改变。

接下来我们将为大家介绍几种常用集合类型的应用：

序号	集合及描述
1	Scala List(列表) List 的特征是其元素以线性方式存储，集合中可以存放重复对象。 参考 API文档
2	Scala Set(集合) Set 是最简单的一种集合。集合中的对象不按特定的方式排序，并且没有重复对象。 参考 API文档
3	Scala Map(映射) Map 是一种把键对象和值对象映射的集合，它的每一个元素都包含一对键对象和值对象。 参考 API文档
4	Scala 元组 元组是不同类型的值的集合
5	Scala Option Option[T] 表示有可能包含值的容器，也可能不包含值。
6	Scala Iterator (迭代器) 迭代器不是一个容器，更确切的说是逐一访问容器内元素的方法。

实例

以下代码判断，演示了所有以上集合类型的定义实例：

```
// 定义整型 List

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


// 定义 Set

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


// 定义 Map

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


// 创建两个不同类型元素的元组

val x = (10, "Runoob")


// 定义 Option

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


1 篇笔记

#1

写笔记

Scala 程序使用 **Option** 非常频繁，在 **Java** 中使用 **null** 来表示空值，代码中很多地方都要添加 **null** 关键字检测，不然很容易出现 **NullPointerException**。因此 **Java** 程序需要关心那些变量可能是 **null**,而这些变量出现 **null** 的可能性很低，但一但出现，很难查出为什么出现 **NullPointerException**。
Scala 的 **Option** 类型可以避免这种情况，因此 **Scala** 应用推荐使用 **Option** 类型来代表一些可选值。使用 **Option** 类型，读者一眼就可以看出这种类型的值可能为 **None**。
参考链接：[Scala 使用Option、Some、None，避免使用 null](#)

starxhong2个月前 (07-23)

反馈/建议

首页HTMLCSSJS本地书签

Scala Option(选项)

Scala 类和对象

Scala Iterator（迭代器）

Scala 集合

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())  
  
        }  
  
    }  
  
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
```

```
$ scala Test
```

```
Baidu
```

```
Google
```

```
Runoob
```

```
Taobao
```

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 )  
  
  
    }  
  
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
```

```
$ scala Test
```

```
最大元素是: 90
```

```
最小元素是: 2
```

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 )


}

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

ita.size 的值: 6

itb.length 的值: 6
```

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

更多方法可以参考 [API文档](#)

[Scala 集合](#)

☐ Scala Option(选项)

Scala 类和对象 ☐

☐ 点我分享笔记

反馈/建议



☐ Scala Iterator（迭代器）

Scala Trait(特征) ☐

Scala 类和对象

类是对象的抽象，而对象是类的具体实例。类是抽象的，不占用内存，而对象是具体的，占用存储空间。类是用于创建对象的蓝图，它是一个定义包括在特定类型的对象中的方法和变量的软件模板。

我们可以使用 **new** 关键字来创建类的对象，实例如下：

```
class Point(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 ("x 的坐标点: " + x);  
  
        println ("y 的坐标点: " + y);  
  
    }  
  
}
```

Scala中的类不声明为public，一个Scala源文件中可以有多个类。

以上实例的类定义了两个变量 **x** 和 **y**，一个方法：**move**，方法没有返回值。

Scala 的类定义可以有参数，称为类参数，如上面的 **xc**, **yc**，类参数在整个类中都可以访问。

接着我们可以使用 **new** 来实例化类，并访问类中的方法和变量：

```
import java.io._

class Point(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 ("x 的坐标点: " + x);

        println ("y 的坐标点: " + y);

    }

}

object Test {

    def main(args: Array[String]) {

        val pt = new Point(10, 20);

        // 移到一个新的位置

        pt.move(10, 10);

    }

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

x 的坐标点: 20

y 的坐标点: 30
```

Scala 继承

Scala继承一个基类跟Java很相似, 但我们需要注意以下几点:

- 1、重写一个非抽象方法必须使用`override`修饰符。
- 2、只有主构造函数才可以往基类的构造函数里写参数。
- 3、在子类中重写超类的抽象方法时, 你不需要使用`override`关键字。

接下来让我们来看个实例:

```
class Point(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 ("x 的坐标点: " + x);

        println ("y 的坐标点: " + y);

    }

}

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

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

    var z: Int = zc

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

        x = x + dx

        y = y + dy

        z = z + dz

        println ("x 的坐标点 : " + x);

        println ("y 的坐标点 : " + y);

        println ("z 的坐标点 : " + z);

    }

}
```

Scala 使用 `extends` 关键字来继承一个类。实例中 `Location` 类继承了 `Point` 类。`Point` 称为父类(基类), `Location` 称为子类。

override val xc 为重写了父类的字段。

继承会继承父类的所有属性和方法，**Scala** 只允许继承一个父类。

实例如下：

```
import java.io._

class Point(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 ("x 的坐标点 : " + x);

        println ("y 的坐标点 : " + y);

    }

}

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

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

    var z: Int = zc

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

        x = x + dx

        y = y + dy

        z = z + dz

        println ("x 的坐标点 : " + x);

        println ("y 的坐标点 : " + y);

        println ("z 的坐标点 : " + z);

    }

}

object Test {

    def main(args: Array[String]) {

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

```
// 移到一个新的位置

loc.move(10, 10, 5);

}

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

x 的坐标点 : 20

y 的坐标点 : 30

z 的坐标点 : 20
```

Scala重写一个非抽象方法，必须用`override`修饰符。

```
class Person {

    var name = ""

    override def toString = getClass.getName + "[name=" + name + "]"

}

class Employee extends Person {

    var salary = 0.0

    override def toString = super.toString + "[salary=" + salary + "]"

}

object Test extends App {

    val fred = new Employee

    fred.name = "Fred"

    fred.salary = 50000

    println(fred)

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
```

```
$ scala Test
```

```
Employee[name=Fred][salary=50000.0]
```

Scala 单例对象

在 **Scala** 中，是没有 **static** 这个东西的，但是它也为我们提供了单例模式的实现方法，那就是使用关键字 **object**。

Scala 中使用单例模式时，除了定义的类之外，还要定义一个同名的 **object** 对象，它和类的区别是，**object** 对象不能带参数。

当单例对象与某个类共享同一个名称时，他被称作是这个类的伴生对象：**companion object**。你必须在同一个源文件里定义类和它的伴生对象。类被称为是这个单例对象的伴生类：**companion class**。类和它的伴生对象可以互相访问其私有成员。

单例对象实例

```
import java.io._

class Point(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 point = new Point(10, 20)

        printPoint

        def printPoint{

            println ("x 的坐标点 : " + point.x);

            println ("y 的坐标点 : " + point.y);

        }

    }

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

x 的坐标点 : 10

y 的坐标点 : 20
```

伴生对象实例

```
/* 文件名: Marker.scala

 * author:菜鸟教程

 * url:www.runoob.com

 */


// 私有构造方法

class Marker private(val color:String) {

    println("创建" + this)

    override def toString(): String = "颜色标记: "+ color

}


// 伴生对象，与类共享名字，可以访问类的私有属性和方法

object Marker{

    private val markers: Map[String, Marker] = Map(

        "red" -> new Marker("red"),

        "blue" -> new Marker("blue"),

        "green" -> new Marker("green")

    )

    def apply(color:String) = {

        if(markers.contains(color)) markers(color) else null

    }

}
```

```
}

def getMarker(color:String) = {

    if(markers.contains(color)) markers(color) else null

}

def main(args: Array[String]) {

    println(Marker("red"))

    // 单例函数调用, 省略了.(点)符号

    println(Marker getMarker "blue")

}

}
```

执行以上代码, 输出结果为:

```
$ scalac Marker.scala

$ scala Marker

创建颜色标记: red

创建颜色标记: blue

创建颜色标记: green

颜色标记: red

颜色标记: blue
```

☐ [Scala Iterator（迭代器）](#)

[Scala Trait\(特征\)](#) ☐

☐ [点我分享笔记](#)

[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号: 闽ICP备15012807号-1



Scala Trait(特征)

Scala 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：

```
/* 文件名: Test.scala  
  
 * author:菜鸟教程  
  
 * url:www.runoob.com  
  
 */  
  
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 Test

false

true

true
```

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时都会被执行。

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；

特征由左到右被构造；

每个特征当中，父特征先被构造；

如果多个特征共有一个父特征，父特征不会被重复构造

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。

[☐ Scala 类和对象](#)

Scala 模式匹配 [☐](#)

[☐ 点我分享笔记](#)

反馈/建议



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。

以下是一个简单的整型值模式匹配实例：

```
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"  
  
    }  
  
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala  
  
$ scala Test  
  
many
```

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。

接下来我们来看一个不同数据类型的模式匹配：

```
object Test {  
  
    def main(args: Array[String]) {  
  
        println(matchTest("two"))  
  
    }  
  
}
```



```

println(matchTest("test"))

println(matchTest(1))

println(matchTest(6))

}

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

  case 1 => "one"

  case "two" => 2

  case y: Int => "scala.Int"

  case _ => "many"

}

}

```

执行以上代码，输出结果为：

```

$ scalac Test.scala

$ scala Test

2

many

one

scala.Int

```

实例中第一个 **case** 对应整型数值 **1**，第二个 **case** 对应字符串值 **two**，第三个 **case** 对应类型模式，用于判断传入的值是否为整型，相比使用 `isInstanceOf` 来判断类型，使用模式匹配更好。第四个 **case** 表示默认的全匹配选项，即没有找到其他匹配时的匹配项，类似 **switch** 中的 **default**。

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。

以下是样例类的简单实例：

```

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 + "?")

    }

}

// 样例类

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

}

```

执行以上代码，输出结果为：

```

$ scalac Test.scala

$ scala Test

Hi Alice!

Hi Bob!

Age: 32 year, name: Charlie?

```

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 Trait\(特征\)](#)

[Scala 正则表达式 ☐](#)

[☐ 点我分享笔记](#)

[反馈/建议](#)

Scala 正则表达式

Scala 通过 `scala.util.matching` 包中的 **Regex** 类来支持正则表达式。以下实例演示了使用正则表达式查找单词 **Scala**：

```
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)

  }

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

Some(Scala)
```

实例中使用 **String** 类的 `r()` 方法构造了一个 **Regex** 对象。

然后使用 `findFirstIn` 方法找到首个匹配项。

如果需要查看所有的匹配项可以使用 `findAllIn` 方法。

你可以使用 `mkString()` 方法来连接正则表达式匹配结果的字符串，并可以使用管道(`|`)来设置不同的模式：

```
import scala.util.matching.Regex

object Test {

  def main(args: Array[String]) {

    val pattern = new Regex("(S|s)cala")  // 首字母可以是大写 S 或小写 s

    val str = "Scala is scalable and cool"
```

```
println((pattern findAllIn str).mkString(","))    // 使用逗号 ， 连接返回结果

}

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

Scala,scala
```

如果你需要将匹配的文本替换为指定的关键词，可以使用 **replaceFirstIn()** 方法来替换第一个匹配项，使用 **replaceAllIn()** 方法替换所有匹配项，实例如下：

```
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"))

  }

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

Java is scalable and cool
```

正则表达式

Scala 的正则表达式继承了 Java 的语法规则，Java 则大部分使用了 Perl 语言的规则。

下表我们给出了常用的一些正则表达式规则：

表达式	匹配规则
^	匹配输入字符串开始的位置。
\$	匹配输入字符串结尾的位置。
.	匹配除“\r\n”之外的任何单个字符。

[...]	字符集。匹配包含的任一字符。例如，"[abc]"匹配"plain"中的"a"。
[^...]	反向字符集。匹配未包含的任何字符。例如，"[^abc]"匹配"plain"中"p", "l", "i", "n"。
\\A	匹配输入字符串开始的位置（无多行支持）
\\z	字符串结尾(类似\$, 但不受处理多行选项的影响)
\\Z	字符串结尾或行尾(不受处理多行选项的影响)
re*	重复零次或更多次
re+	重复一次或更多次
re?	重复零次或一次
re{ n}	重复n次
re{ n,}	
re{ n, m}	重复n到m次
a b	匹配 a 或者 b
(re)	匹配 re,并捕获文本到自动命名的组里
(?: re)	匹配 re,不捕获匹配的文本，也不给此分组分配组号
(?> re)	贪婪子表达式
\\w	匹配字母或数字或下划线或汉字
\\W	匹配任意不是字母，数字，下划线，汉字的字符
\\s	匹配任意的空白符,等同于 [\t\n\r\f]
\\S	匹配任意不是空白符的字符
\\d	匹配数字，类似 [0-9]
\\D	匹配任意非数字的字符
\\G	当前搜索的开头
\\n	换行符
\\b	通常是单词分界位置，但如果在字符类里使用代表退格
\\B	匹配不是单词开头或结束的位置
\\t	制表符
\\Q	开始引号：\\Q(a+b)*3\\E 可匹配文本 "(a+b)*3"。
\\E	结束引号：\\Q(a+b)*3\\E 可匹配文本 "(a+b)*3"。

正则表达式实例

实例	描述
.	匹配除"\r\n"之外的任何单个字符。
[Rr]uby	匹配 "Ruby" 或 "ruby"

rub[ye]	匹配 "ruby" 或 "rube"
[aeiou]	匹配小写字母：aeiou
[0-9]	匹配任何数字，类似 [0123456789]
[a-z]	匹配任何 ASCII 小写字母
[A-Z]	匹配任何 ASCII 大写字母
[a-zA-Z0-9]	匹配数字，大小写字母
[^aeiou]	匹配除了 aeiou 其他字符
[^0-9]	匹配除了数字的其他字符
\\d	匹配数字，类似: [0-9]
\\D	匹配非数字，类似: [^0-9]
\\s	匹配空格，类似: [\\t\\n\\f]
\\S	匹配非空格，类似: [^\\t\\n\\f]
\\w	匹配字母，数字，下划线，类似: [A-Za-z0-9_]
\\W	匹配非字母，数字，下划线，类似: [^A-Za-z0-9_]
ruby?	匹配 "rub" 或 "ruby": y 是可选的
ruby*	匹配 "rub" 加上 0 个或多个的 y。
ruby+	匹配 "rub" 加上 1 个或多个的 y。
\\d{3}	刚好匹配 3 个数字。
\\d{3,}	匹配 3 个或多个数字。
\\d{3,5}	匹配 3 个、4 个或 5 个数字。
\\D\\d+	无分组：+ 重复 \\d
(\\D\\d)+/	分组：+ 重复 \\D\\d 对
([Rr]uby(,)?)+	匹配 "Ruby"、"Ruby, ruby, ruby"，等等

注意上表中的每个字符使用了两个反斜线。这是因为在 **Java** 和 **Scala** 中字符串中的反斜线是转义字符。所以如果你要输出 `\\.` ，你需要在字符串中写成 `\\.` 来获取一个反斜线。查看以下实例：

```
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(","))

}

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

able1
```

[Scala 模式匹配](#)

Scala 异常处理 [Scala](#)

[点我分享笔记](#)

反馈/建议

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Scala 正则表达式](#)

Scala 提取器(Extractor) [Scala](#)

Scala 异常处理

Scala 的异常处理和其它语言比如 **Java** 类似。

Scala 的方法可以通过抛出异常的方法来方式来终止相关代码的运行，不必通过返回值。

抛出异常

Scala 抛出异常的方法和 **Java**一样，使用 **throw** 方法，例如，抛出一个新的参数异常：

```
throw new IllegalArgumentException
```

捕获异常

异常捕捉的机制与其他语言中一样，如果有异常发生，**catch**字句是按次序捕捉的。因此，在**catch**字句中，越具体的异常越要靠前，越普遍的异常越靠后。 如果抛出的异常不在**catch**字句中，该异常则无法处理，会被升级到调用者处。

捕捉异常的**catch**子句，语法与其他语言中不太一样。在**Scala**里，借用了模式匹配的思想来做异常的匹配，因此，在**catch**的代码里，是一系列**case**字句，如下例所示：

```
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")

      }

    }

  }

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

Missing file exception
```

`catch` 字句里的内容跟 `match` 里的 `case` 是完全一样的。由于异常捕捉是按次序，如果最普遍的异常，`Throwable`，写在最前面，则在它后面的 `case` 都捕捉不到，因此需要将它写在最后面。

finally 语句

`finally` 语句用于执行不管是正常处理还是有异常发生时都需要执行的步骤，实例如下：

```
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...")  
  
    }  
  
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala  
  
$ scala Test  
  
Missing file exception  
  
Exiting finally...
```

[☐ Scala 正则表达式](#)

[Scala 提取器\(Extractor\) ☐](#)

[☐ 点我分享笔记](#)

[反馈/建议](#)

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



Scala 提取器(Extractor)

提取器是从传递给它的对象中提取出构造该对象的参数。

Scala 标准库包含了一些预定义的提取器，我们会大致的了解一下它们。

Scala 提取器是一个带有`unapply`方法的对象。`unapply`方法算是`apply`方法的反向操作：`unapply`接受一个对象，然后从对象中提取值，提取的值通常是用来构造该对象的值。

以下实例演示了邮件地址的提取器对象：

```
object Test {

    def main(args: Array[String]) {

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

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

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

    }

    // 注入方法（可选）

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

        user +"@"+ domain

    }

    // 提取方法（必选）

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

        val parts = str split "@"

        if (parts.length == 2){

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

        }else{

            None

        }

    }

}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala

$ scala Test

Apply 方法 : Zara@gmail.com

Unapply 方法 : Some((Zara, gmail.com))

Unapply 方法 : None
```

以上对象定义了两个方法：**apply** 和 **unapply** 方法。通过 **apply** 方法我们无需使用 **new** 操作就可以创建对象。所以您可以通过语句 `Test("Zara", "gmail.com")` 来构造一个字符串 `"Zara@gmail.com"`。

unapply方法算是**apply**方法的反向操作：**unapply**接受一个对象，然后从对象中提取值，提取的值通常是用来构造该对象的值。实例中我们使用 **Unapply** 方法从对象中提取用户名和邮件地址的后缀。

实例中 **unapply** 方法在传入的字符串不是邮箱地址时返回 **None**。代码演示如下：

```
unapply("Zara@gmail.com")  相等 于  Some("Zara", "gmail.com")

unapply("Zara Ali")  相等 于  None
```

提取器使用模式匹配

在我们实例化一个类的时，可以带上0个或者多个的参数，编译器在实例化的时会调用 **apply** 方法。我们可以在类和对象中都定义 **apply** 方法。

就像我们之前提到过的，**unapply** 用于提取我们指定查找的值，它与 **apply** 的操作相反。 当我们在提取器对象中使用 **match** 语句是，**unapply** 将自动执行，如下所示：

```
object Test {

  def main(args: Array[String]) {

    val x = Test(5)

    println(x)

    x match

    {

      case Test(num) => println(x + " 是 " + num + " 的两倍! ")

      //unapply 被调用

      case _ => println("无法计算")

    }

  }

  def apply(x: Int) = x*2

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

```
}
```

执行以上代码，输出结果为：

```
$ scalac Test.scala
```

```
$ scala Test
```

```
10
```

```
10 是 5 的两倍！
```

[Scala 异常处理](#)

[Scala 文件 I/O](#)

[点我分享笔记](#)

反馈/建议

Copyright © 2013-2018 菜鸟教程 [runoob.com](#) All Rights Reserved. 备案号：闽ICP备15012807号-1



[首页](#) [HTML](#) [CSS](#) [JS](#) [本地书签](#)

[Scala 提取器\(Extractor\)](#)

Scala 文件 I/O

Scala 进行文件写操作，直接用的都是 java 中的 I/O 类（**java.io.File**）：

```
import java.io._

object Test {

  def main(args: Array[String]) {

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

    writer.write("菜鸟教程")

    writer.close()

  }

}
```

执行以上代码，会在你的当前目录下生产一个 **test.txt** 文件，文件内容为"菜鸟教程"：

```
$ scalac Test.scala

$ scala Test

$ cat test.txt

菜鸟教程
```

从屏幕上读取用户输入

有时候我们需要接收用户在屏幕输入的指令来处理程序。实例如下：

```
import scala.io._

object Test {

    def main(args: Array[String]) {

        print("请输入菜鸟教程官网 : " )

        val line = StdIn.readLine()

        println("谢谢，你输入的是: " + line)

    }

}
```

*Scala 2.11 后的版本 **Console.readLine** 已废弃，使用 **scala.io.StdIn.readLine()** 方法代替。*

执行以上代码，屏幕上会显示如下信息：

```
$ scalac Test.scala

$ scala Test

请输入菜鸟教程官网 : www.runoob.com

谢谢，你输入的是: www.runoob.com
```

从文件上读取内容

从文件读取内容非常简单。我们可以使用 **Scala** 的 **Source** 类及伴生对象来读取文件。以下实例演示了从 "test.txt"(之前已创建过) 文件中读取内容：

```
import scala.io.Source
```

```
object Test {  
  
  def main(args: Array[String]) {  
  
    println("文件内容为:" )  
  
    Source.fromFile("test.txt" ).foreach{  
  
      print  
  
    }  
  
  }  
  
}
```

执行以上代码，输出结果为:

```
$ scalac Test.scala
```

```
$ scala Test
```

文件内容为:

菜鸟教程

☐ [Scala 提取器\(Extractor\)](#)

☐ [点我分享笔记](#)

[反馈/建议](#)