

## 第 1 章 Scala 编程基础

Scala 是一种混合编程语言，支持面向对象编程和函数式编程。它支持函数式编程概念，如不可变数据结构和函数作为一等公民。对于面向对象编程，它支持类、对象和特征等概念。它还支持封装、继承、多态性和其他重要的面向对象概念。



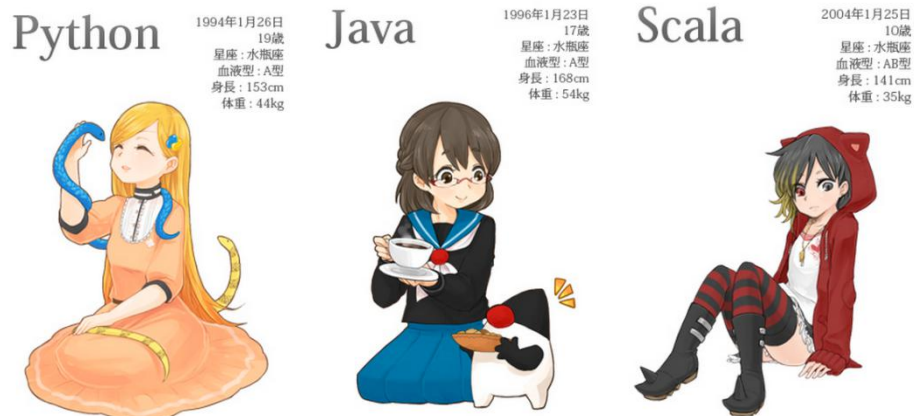
Scala 原作者 Martin Odersky

Scala 是一种静态类型的语言。Scala 应用程序是由 Scala 编译器编译的。它是一种类型安全的语言，Scala 编译器在编译时强制类型安全。这有助于减少应用程序中的错误数量。

Scala 是一种基于 Java 虚拟机(JVM)的语言。Scala 编译器将 Scala 应用程序编译成 Java 字节码，可以在任何 JVM 上运行。在字节码级别，Scala 应用程序与 Java 应用程序没有区别。因为 Scala 是基于 JVM 的，所以它可以与 Java 无缝地互操作。更重要的是，Scala 应用程序可以使用任何 Java 库，而不需要任何包装器或粘合代码。

本章的目标不是让读者成为 Scala 的专家，而是帮助读者学习足够的 Scala 知识，以便使用 Scala 来理解和编写 Spark 应用程序。

"Scala 是一种强大的语言。强大带来了复杂性"。Scala 结合了面向对象和函数式编程范式。如果没有 Java 或 c++ 等面向对象编程语言的经验，Scala 的语法可能很难掌握。在学习 Scala 之前，最好有 Java 或 Python 语言经验。



从生产实践来看，无论是老公司的新部门，还是新成立的公司，如果涉及到大数据、广告、

流媒体、游戏、银行以及一些领域模型比较复杂的场景，大多数企业还是会尝试调研一下 Scala 生态。国内的广告、电商、数据分析、流媒体、银行等行业，都有采用 Scala（及生态圈）作为基建语言和框架。

下面是 Scala 的官网地址、API 参考网址以及开发 Scala 程序的 IDE 工具网址：

- ❑ Scala 的官网：<http://scala-lang.org/>
- ❑ Scala API 地址：<https://www.scala-lang.org/api/2.11.11/>
- ❑ IDE 开发工具：<http://scala-ide.org>

## 1.1 Scala 简介

Scala 是一种非常适合开发大数据应用程序的语言，是使用 Apache Spark 的首选语言。使用 Scala 语言来学习 Spark，具有以下优点：

- ❑ 首先，开发人员可以通过使用 Scala 实现显著的生产力提升。
- ❑ 其次，它帮助开发人员编写健壮的代码，减少 bug。
- ❑ 第三，Spark 是用 Scala 编写的，因此 Scala 非常适合开发 Spark 应用程序。

Spark 本身是用 Scala 编写的，这是一种基于 Java 虚拟机（JVM）的函数式编程语言。Scala 编译器会生成在 JVM 上执行的字节码。因此，它可以与任何其他基于 JVM 的系统无缝集成，比如 HDFS，Cassandra，HBase 等等。Scala 是首选语言，因为它简洁的编程接口、交互式 shell，以及它捕获功能并高效地将它们传递到集群中的节点的能力。Scala 是一个可扩展的（scalable，因此得名），静态类型的，高效的多范式语言，它支持函数式语言和面向对象语言特性。

### 1.1.1 Scala 语言特性

Scala 编程语言的特性，可概括如下：

- ❑ 可扩展
  - 面向对象
  - 函数式编程
- ❑ 兼容 JAVA
  - 类库调用
  - 互操作
- ❑ 语法简洁
  - 代码行短
  - 类型推断
  - 抽象控制
- ❑ 静态类型化
  - 可检验
  - 安全重构
- ❑ 支持并发控制
  - 强计算能力
  - 自定义其他控制结构语言特点



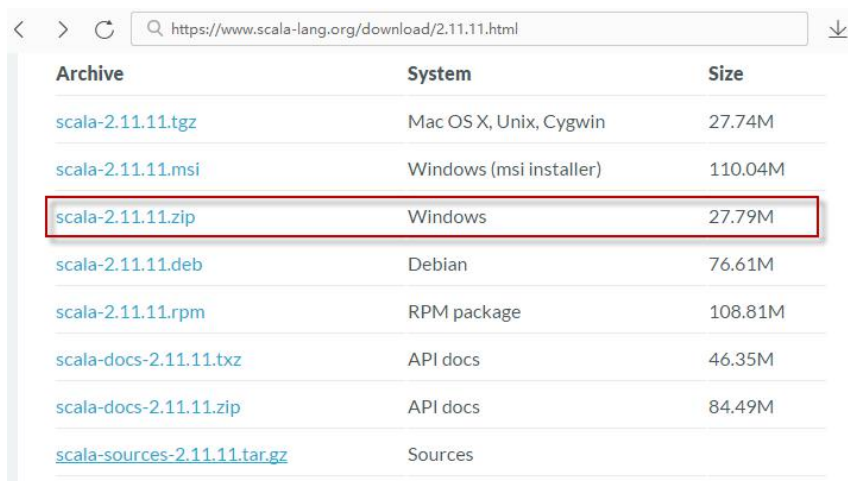
## 1.1.2 安装 Scala

Scala 可以安装在 Windows 和 Linux 操作系统下。下面我们分别介绍在这两个操作系统下安装 Scala 的步骤。（注：在安装 Scala 前，请确保已经安装好了 JDK 8，并配置好了环境变量。JDK 8 的安装和配置不属于本书内容，请自行查询相关资料）

### 在 Windows 平台上

在 Windows 上安装和配置 Scala 的步骤如下：

1) 首先从 Scala 的官网(<http://scala-lang.org/>)下载安装包。本教程使用的是 Scala 2.11.11，其下载链接是 <https://www.scala-lang.org/download/2.11.11.html>，如下图所示（注意选择 Windows 版本下载）：

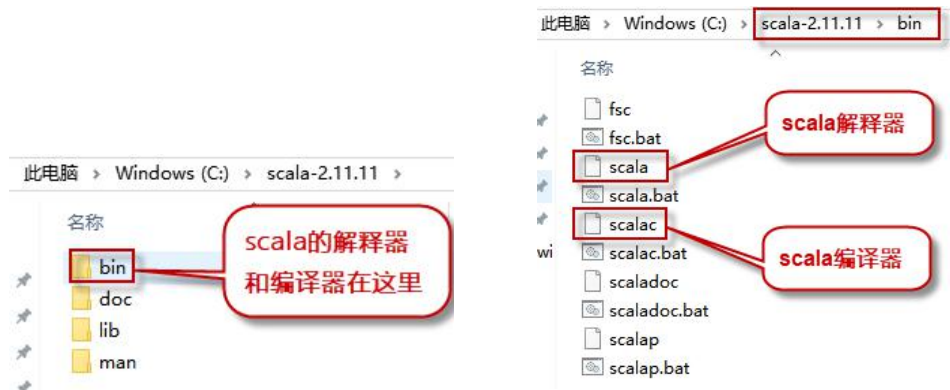


Archive	System	Size
<a href="#">scala-2.11.11.tgz</a>	Mac OS X, Unix, Cygwin	27.74M
<a href="#">scala-2.11.11.msi</a>	Windows (msi installer)	110.04M
<a href="#">scala-2.11.11.zip</a>	Windows	27.79M
<a href="#">scala-2.11.11.deb</a>	Debian	76.61M
<a href="#">scala-2.11.11.rpm</a>	RPM package	108.81M
<a href="#">scala-docs-2.11.11.txz</a>	API docs	46.35M
<a href="#">scala-docs-2.11.11.zip</a>	API docs	84.49M
<a href="#">scala-sources-2.11.11.tar.gz</a>	Sources	

2) 将下载的安装包解压缩到指定的位置。比如，笔者把它解压缩到 Windows 的 C 盘根目录下：



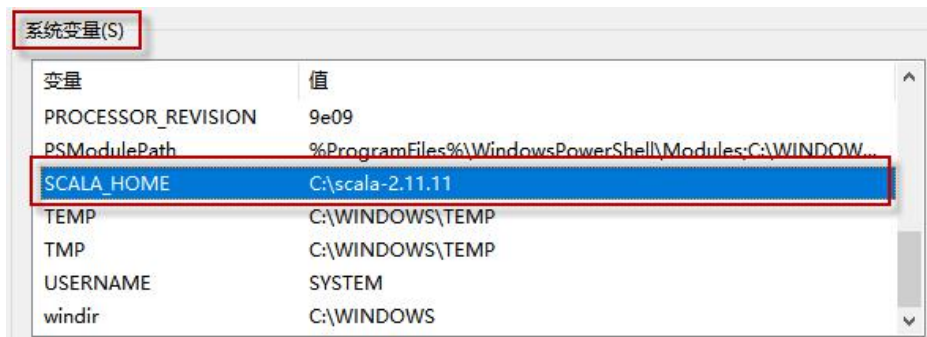
3) 了解一下 Scala 的目录结构，如下所示：



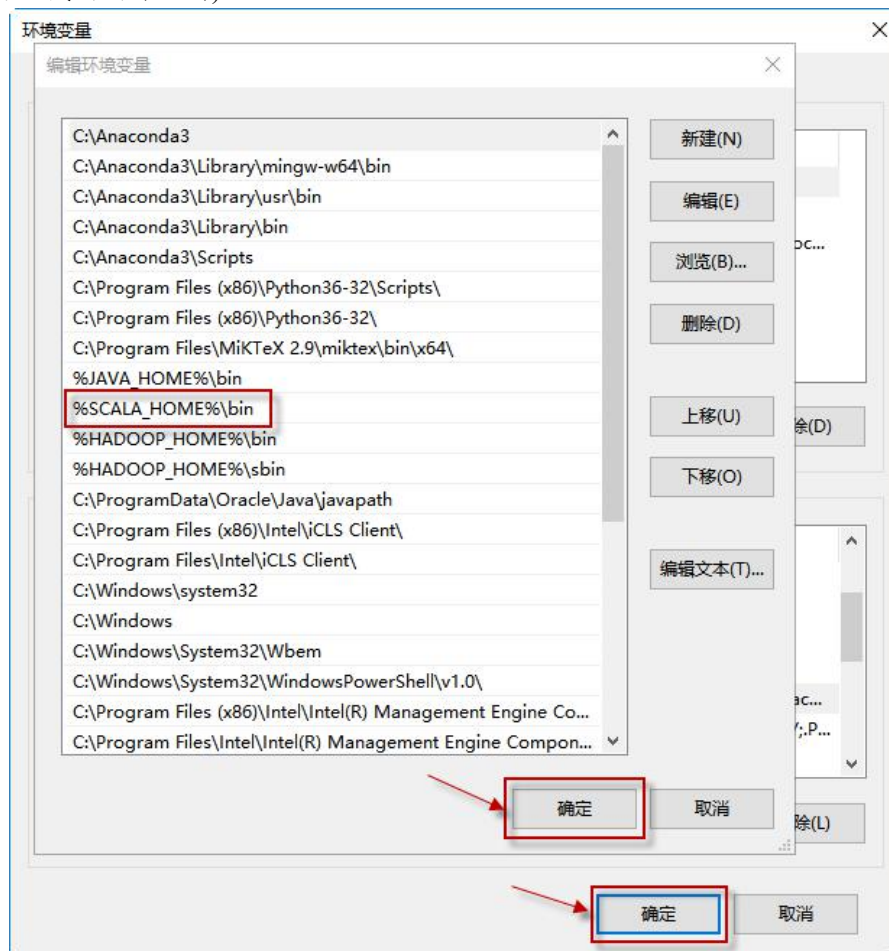
4) 配置环境变量。在"我的电脑"上单击右键，在弹出菜单中依次选择"属性 | 高级系统设置 | 高级 | 环境变量"，如下所示：



在打开的环境变量配置窗口中，选择下方的"系统变量"窗口，单击"新建"按钮创建新的系统变量。请先创建"SCALA\_HOME"环境变量，设置其变量值为刚才 Scala 解压缩后的主目录地址，如下图所示：



接着在"系统变量"窗口中，找到 PATH 变量，双击打开，在其原有值的最前方，添加如下内容(一定不要删除或修改已有的值)，添加的内容为 Scala 安装目录下的 bin 目录，目的是告诉操作系统，Scala 程序的编译器和解释器的位置(注：这里作者使用的是 Win10 系统，读者的电脑上可能会与此截图稍有不同)：



然后一路单击【确定】按钮，保存环境变量的设置。

5) 验证安装是否成功。打开命令行窗口(终端窗口)，分别键入以下命令。(说明：scalac 是 Scala 的编译器，scala 是 Scala 的解释器)

```
scalac -version  
scala -version
```

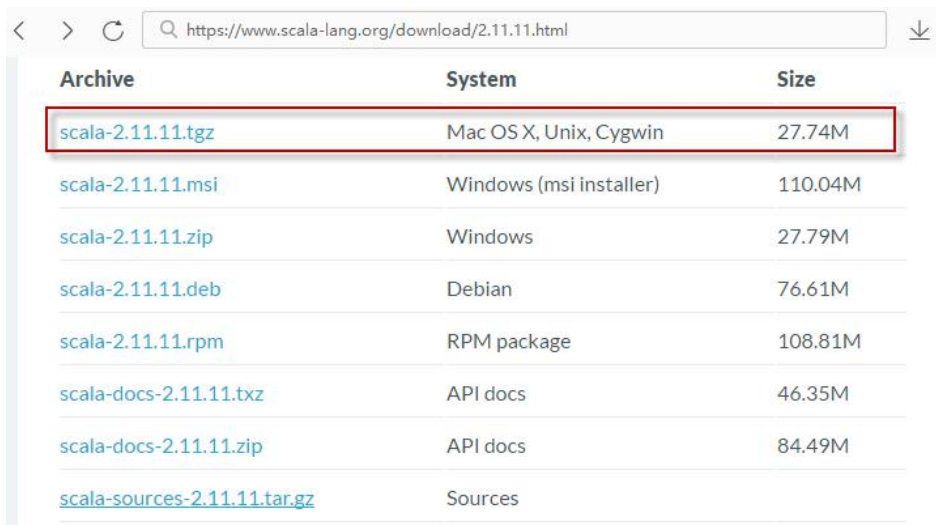
如果能正确显示 scala 的版本号,说明环境变量配置正确。否则,请重复以上的步骤,重新检查环境变量配置的路径和大小写及标点符号(英文半角)是否正确。

```
Windows PowerShell
PS C:\>
PS C:\> scalac -version
Scala compiler version 2.11.11 -- Copyright 2002-2017, LAMP/EPFL
PS C:\> scala -version
Scala code runner version 2.11.11 -- Copyright 2002-2017, LAMP/EPFL
PS C:\>
```

## 在 Linux 平台上

在 Linux 操作系统上安装和配置 Scala 的步骤如下:

1) 首先从 Scala 的官网(<http://scala-lang.org/>)下载安装包。本教程使用的是 Scala 2.11.11, 其下载链接是 <https://www.scala-lang.org/download/2.11.11.html>, 如下所示(注意选择 Linux 版本下载):



Archive	System	Size
<a href="#">scala-2.11.11.tgz</a>	Mac OS X, Unix, Cygwin	27.74M
<a href="#">scala-2.11.11.msi</a>	Windows (msi installer)	110.04M
<a href="#">scala-2.11.11.zip</a>	Windows	27.79M
<a href="#">scala-2.11.11.deb</a>	Debian	76.61M
<a href="#">scala-2.11.11.rpm</a>	RPM package	108.81M
<a href="#">scala-docs-2.11.11.tgz</a>	API docs	46.35M
<a href="#">scala-docs-2.11.11.zip</a>	API docs	84.49M
<a href="#">scala-sources-2.11.11.tar.gz</a>	Sources	

2) 将下载的安装包解压缩到指定的位置。假设我们把下载的安装包 `scala-2.11.11.tgz` 保存在 `~/software/` 目录下, 现在我们把它解压缩到 `/usr/local` 目录下。打开终端窗口, 在终端窗口中使用如下的命令进行解压缩:

```
$ cd /usr/local
$ sudo tar -zxvf ~/software/scala-2.11.11.tgz
$ ls
```

如下图所示:



```
[hduser@master ~]$ cd /usr/local
[hduser@master local]$ sudo tar -zxvf ~/software/scala-2.11.11.tgz
[hduser@master local]$ ll
总用量 0
drwxr-xr-x. 2 root root    6 4月 11 2018 bin
drwxr-xr-x. 2 root root    6 4月 11 2018 etc
drwxr-xr-x. 2 root root    6 4月 11 2018 games
drwxr-xr-x. 2 root root    6 4月 11 2018 include
drwxr-xr-x. 8  10 143 255 12月 20 2017 jdk1.8.0_162
drwxr-xr-x. 2 root root    6 4月 11 2018 lib
drwxr-xr-x. 2 root root    6 4月 11 2018 lib64
drwxr-xr-x. 2 root root    6 4月 11 2018 libexec
drwxr-xr-x. 2 root root    6 4月 11 2018 sbin
drwxrwxr-x. 6 1001 1001  50 4月 14 2017 scala-2.11.11
drwxr-xr-x. 5 root root   49 9月 27 03:47 share
drwxr-xr-x. 2 root root    6 4月 11 2018 src
[hduser@master local]$
```

解压后

3) 了解一下 Scala 的目录结构，在终端窗口中使用命令如下：

```
$ ll scala-2.11.11/
```

如下所示：

```
[hduser@master local]$ ll scala-2.11.11/
总用量 4
drwxrwxr-x 2 1001 1001 162 4月 14 2017 bin
drwxrwxr-x 4 1001 1001  86 4月 14 2017 doc
drwxrwxr-x 2 1001 1001 4096 4月 14 2017 lib
drwxrwxr-x 3 1001 1001  18 4月 14 2017 man
[hduser@master local]$
```

scala的解释器  
和编译器在这里

4) 进一步查看 Scala 主目录下的 bin 目录，在终端窗口中使用如下命令：

```
$ ll scala-2.11.11/bin
```

如下图所示：

```
[hduser@master local]$ ll scala-2.11.11/bin
总用量 80
-rwxrwxr-x 1 1001 1001 5780 4月 14 2017 fsc
-rwxrwxr-x 1 1001 1001 4968 4月 14 2017 fsc.bat
-rwxrwxr-x 1 1001 1001 5784 4月 14 2017 scala
-rwxrwxr-x 1 1001 1001 4976 4月 14 2017 scala.bat
-rwxrwxr-x 1 1001 1001 5771 4月 14 2017 scalac
-rwxrwxr-x 1 1001 1001 4950 4月 14 2017 scalac.bat
-rwxrwxr-x 1 1001 1001 5775 4月 14 2017 scaladoc
-rwxrwxr-x 1 1001 1001 4958 4月 14 2017 scaladoc.bat
-rwxrwxr-x 1 1001 1001 5774 4月 14 2017 scalap
-rwxrwxr-x 1 1001 1001 4956 4月 14 2017 scalap.bat
[hduser@master local]$
```

scala解释器

scala编译器

5) 配置环境变量。在终端窗口中键入如下的命令，打开环境变量配置文件 profile：

```
$ sudo nano /etc/profile
```

在打开的文件最后，添加如下内容：

```
export SCALA_HOME=/usr/local/scala-2.11.11
export PATH=$PATH:$SCALA_HOME/bin
```

然后按下 Ctrl + X，保存退出。

6) 让环境变量设置生效。在终端窗口中键入如下的命令：

```
$ source /etc/profile
```

7) 验证安装是否成功。在终端窗口分别键入以下命令。(说明：scalac 是 Scala 的编译器，scala 是 Scala 的解释器)

```
scalac -version  
scala -version
```

如果能正确显示 scala 的版本号，说明环境变量配置正确。否则，请重复以上的步骤重新检查环境变量配置的路径和大小写及标点符号(英文半角)是否正确。

以上几步操作如下图所示：

```
[hduser@master ~]$ sudo nano /etc/profile  
[sudo] hduser 的密码：  
[hduser@master ~]$ source /etc/profile  
[hduser@master ~]$ scalac -version  
Scala compiler version 2.11.11 -- Copyright 2002-2017, LAMP/EPFL  
[hduser@master ~]$ scala -version  
Scala code runner version 2.11.11 -- Copyright 2002-2017, LAMP/EPFL  
[hduser@master ~]$
```

### 1.1.3 使用 Scala Shell

Scala 解释器读到一个表达式，对它进行求值，将它打印出来，接着再继续读下一个表达式。这个过程被称做读取--求值--打印--循环，即：REPL。

从技术上讲，scala 程序并不是一个解释器。实际发生的是，我们输入的内容被快速地编译成字节码，然后这段字节码交由 Java 虚拟机执行。正因为如此，大多数 scala 程序员更倾向于将它称做“REPL”。

Scala 提供了一个 REPL 工具，叫做 Scala Shell。我们可以使用 Scala Shell 进行交互式编程，这对于学习 Scala 这门语言来说非常方便。要进入 Scala Shell，请在终端窗口键入如下命令：

```
$ scala
```

回车，就会进入到 Scala Shell 界面，如下所示：


```
[hduser@master ~]$ scala  
Welcome to Scala 2.11.11 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_162).  
Type in expressions for evaluation. Or try :help.  
  
scala>
```

在这个界面中，我们可以交互式地执行 Scala 语句，如下所示：

```
scala> "hello"  
res0: String = hello  
  
scala> 1 + 2  
res1: Int = 3  
  
scala> "Hello".filter(line=>(line!='l'))  
res2: String = Heo
```



注意到在我们输入每个 Scala 语句后，它会输出一行信息，由三部分组成。其中输出的第一部分是 REPL 给表达式起的变量名。在这几个例子里，REPL 为每个表达式定义了一个新变量（res0 到 res2）。输出的第二部分（: 后面的部分）是变量的数据类型，比如字符串是 String 类型，整数是 Int 类型。输出的最后一部分是表达式求值后的结果，也就是变量的值。如下图所示：



The screenshot shows the Scala REPL interface with two lines of input and output. The first line is `scala> "hello"` followed by `res0: String = hello`. The second line is `scala> 1 + 2` followed by `res1: Int = 3`. Red boxes and arrows highlight specific parts: a box around 'res0' is labeled '变量名' (Variable Name); a box around 'String' is labeled '变量的数据类型' (Variable's Data Type); a box around 'hello' is labeled '变量值' (Variable Value).

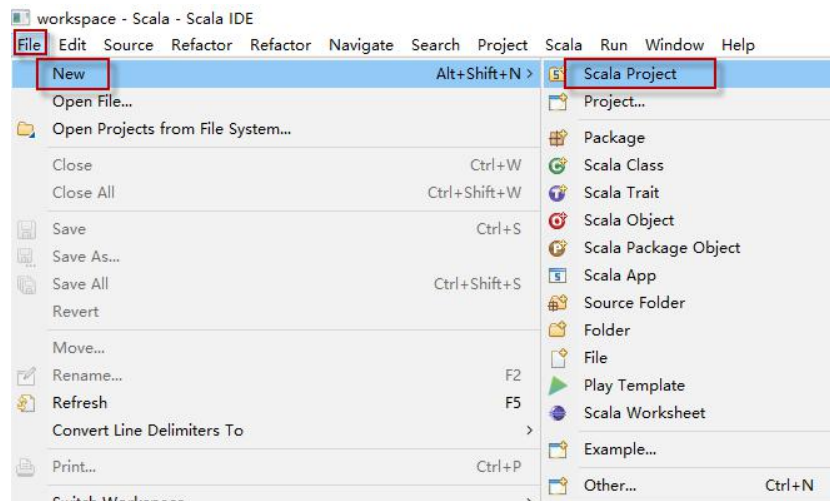
如果要退出 Scala Shell，键入 `":q"` 或 `":quit"` 命令即可：

```
scala> :q
```

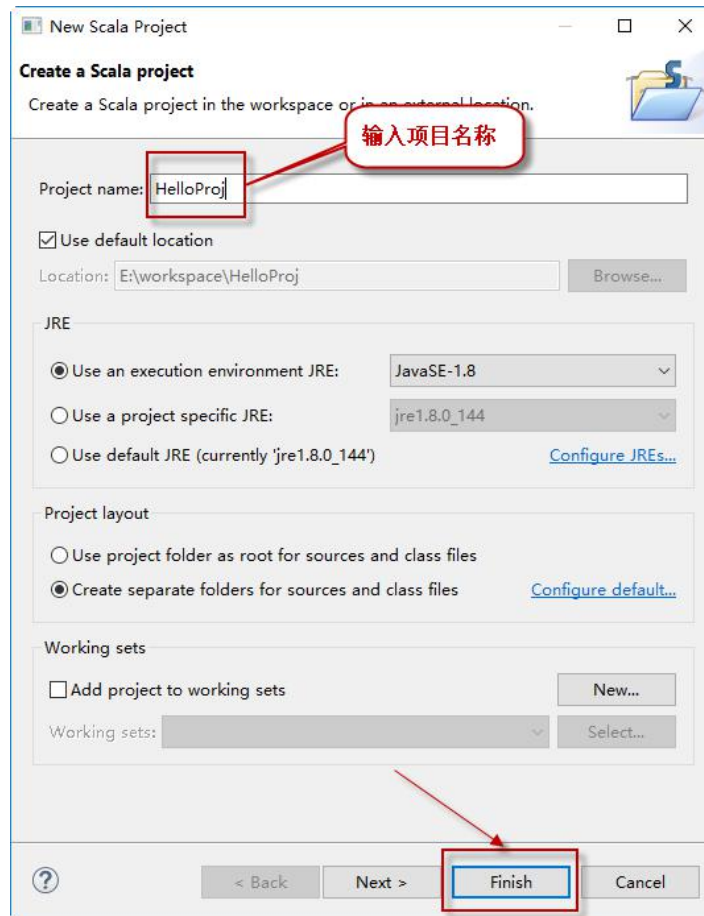
### 1.1.4 使用 Scala IDE

另外，也可以使用集成开发工具 Scala IDE 来开发 Scala 程序。安装和使用 Scala IDE 的步骤如下：

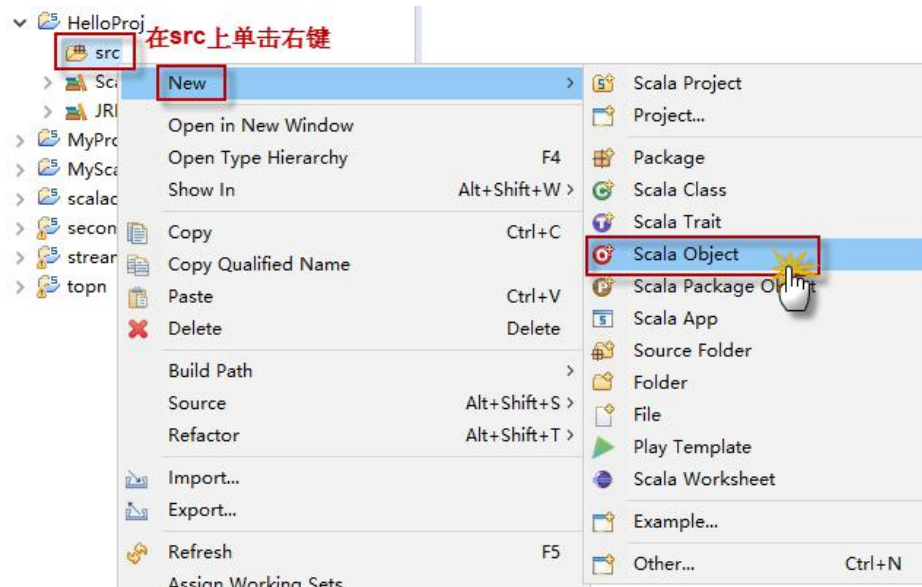
1、从 <http://scala-ide.org> 站点下载 Scala IDE，然后解压缩到指定位置。双击启动图标，打开 Scala IDE，新建一个 Scala 项目，如下图所示：



2、输入项目名称 "HelloProj"，然后点击【Finish】按钮，如下图所示：

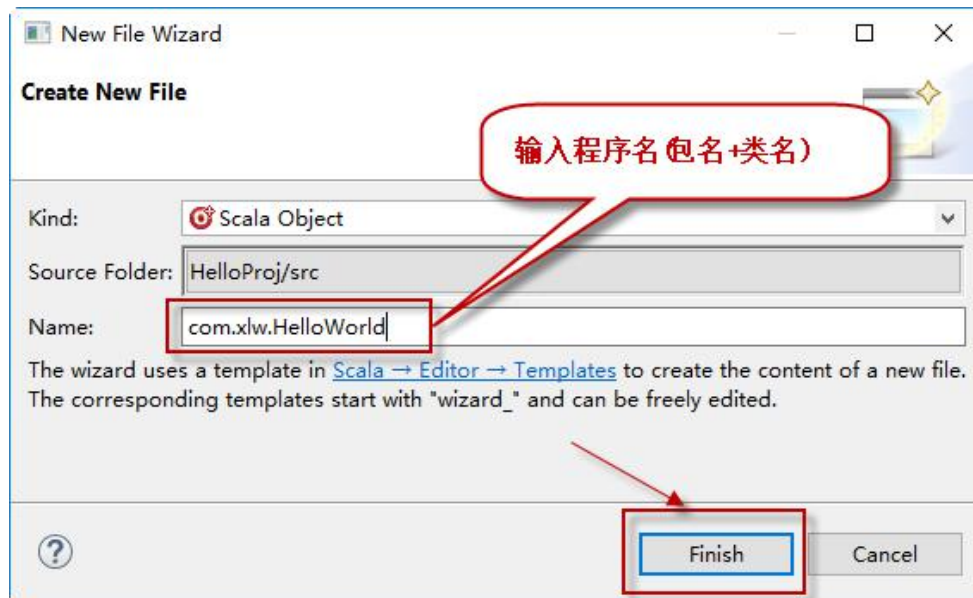


3、然后在项目的源目录(src 目录)下,单击右键,选择"Scala Object"菜单项,新建 Scala Object 文件（即为 scala 源代码文件），如下图所示：



4、在打开的"New File Wizard"新文件向导窗口中,输入程序的名称（包名+类名），例如

com.xlw.HelloWorld，然后点击【Finish】按钮，创建包含 main 方法的主程序，如下图所示：



5、然后在编辑窗口打开 HelloWorld 源文件（双击打开），如下图所示。

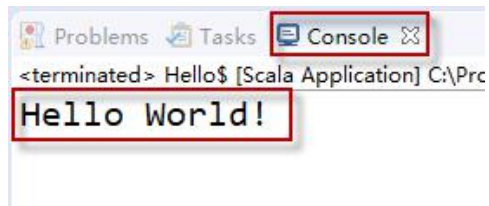


6、在打开的源文件中编辑源代码。代码编写完成后，点击 IDE 上侧工具栏中的绿色按钮，执行程序，如下图所示：



注：一个独立的 Scala 应用程序需要具有一个带有一个 main 方法的单例对象。这个 main 方法接收一个类型为 Array[String] 的输入参数，并且不返回任何值。这是 Scala 应用程序的入口点。该包含 main 方法的单例对象可以取任何名。

7、程序运行结果会出现在 IDE 下方的 console（控制台）窗口，如下图所示：



### 1.1.5 运行 Scala 程序

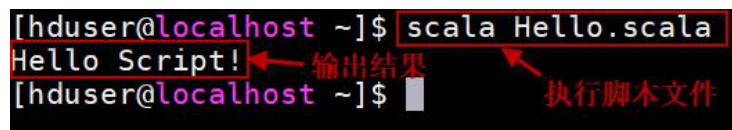
在运行方式上，Scala 非常灵活。它可以被当作一种脚本语言执行，也可以像 Java 一样，作为应用程序执行。

#### 作为脚本执行

我们可以将 Scala 表达式写在一个文件里，然后在命令行中使用 scala 解释器直接运行该文件，就可得到程序运行结果。例如，创建一个 Hello.scala 脚本文件，输入内容如下：

```
println("Hello Script!")
```

然后在打开命令行，使用如下命令直接执行该脚本文件。如下图所示：



#### 作为应用程序执行

作为应用程序执行时，我们需要在一个单例对象中定义入口函数 main，经过编译后就可以执行该应用程序了。例如，下面创建一个 HelloScala.java 源程序，包含 main 方法。这样的应用程序需要先编译为字节码文件，然后执行。代码如下所示：

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

可以在 Scala-IDE 中编辑这个应用程序并执行，参考上一小节内容(1.1.4 节)。

Scala 还提供了一个更简便的方式，直接继承另一个对象 App，无需定义 main 方法，编译即可运行。如下面的代码所示：

```
object HelloScala extends App {  
  println("Hello Scala!")  
}
```

注：但是在 Spark 中，这种方法有时可不会正确的运行（出现异常或者是数据不见了）。为了程序能够正常地运行，最好不要继承 App，直接用 main 方法。

#### 交互式执行

对于数据分析人员来说，通常需要交互式地对数据进行探索性分析。使用 Scala REPL，可

以轻松地交互式执行 scala 命令（请参考 1.1.3 小节：Scala 解释器）。如下图所示：

```
scala> "hello"
res0: String = hello

scala> 1 + 2
res1: Int = 3

scala> "Hello".filter(line=>(line!='l'))
res2: String = Heo
```

## 1.2 Scala 基本语法

Scala 语言的语法非常简洁，拥有其他语言编程经验的程序员很容易读懂 Scala 代码。Scala 语言异常精炼，实现同样功能的程序，在代码量上，使用 Scala 实现通常比 Java 实现少一半或者更多。短小精悍的代码常常意味着更易懂，更易维护。本文将为大家介绍 Scala 语言的基本语法。

### 1.2.1 变量

Scala 有两种类型变量：可变的和不可变的。不可变意味着变量的值一旦声明就不能更改。数据不变性帮助在管理数据时实现并发控制。强烈建议不要使用可变变量。一个纯函数程序永远也不要使用可变变量。不过，有时可变变量的使用可能会导致更少的代码复杂性，因此 Scala 也支持可变变量。可变变量应该被谨慎使用。

在 Scala 中，声明变量的关键字有三种：val、var 和 lazy val。其中：

- ❑ val：声明的变量是不可变的（只读的）。使用 val 关键字声明不可变变量就像在 Java 中声明 final 变量一样。
- ❑ var：声明的变量是可变的。在变量创建以后，可以重新赋值。
- ❑ lazy val：变量只被计算一次，在该变量第一次被访问时。

在下面的代码中，我们分别使用 val 定义不可变的变量，使用 var 来定义可变的变量。使用 val 关键字声明的变量在初始化以后，不允许重新赋值（否则编译器在编译时会出现错误）：



```
scala> var a = 3
a: Int = 3

scala> a = 4
a: Int = 4

scala> val b = 3
b: Int = 3

scala> b = 4
<console>:12: error: reassignment to val
b = 4
^

scala> lazy val c = 3
c: Int = <lazy>

scala> c
res6: Int = 3
```

a是可变的,所以可以重新赋值

b是不可变的,重新赋值会报错

c是延迟计算的,直到使用时才赋值

使用 `lazy` 关键字可实现惰性求值特性，这允许用户延迟任何表达式的执行。当使用 `lazy` 关键字声明表达式时，它只会在显式调用时执行。在下面的代码中，表达式 `sum` 是用 `lazy` 关键字定义的。

```
val x = 3
val y = 5
lazy val sum = a + b
print(sum)
```

因此，只有在调用它时才计算它。需要注意的是，惰性计算特性只能在 `val` 中使用。

```
scala> val a = 3
a: Int = 3

scala> val b = 5
b: Int = 5

scala> lazy val sum = a + b
sum: Int = <lazy>

scala> print(sum)
8
scala>
```

前面的代码中，我们在定义变量时，并没有指定数据类型。在 `Scala` 中，不强制显式地指定变量的数据类型，编译器可以通过内置的类型推断机制，根据变量的初始化来识别变量的类型，这称为“类型推断”。

如果想要明确地指定变量的数据类型，可以在变量名后面跟上一个冒号(:)，后面指定数据类型，像下面这样：

```
scala> val b:Byte = -128
b: Byte = -128

scala> val name:String = "张三"
name: String = 张三

scala> 
```

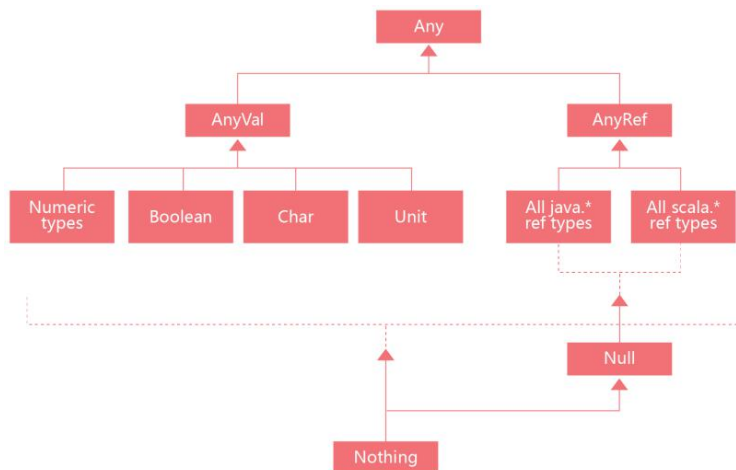
Scala 是一个静态类型语言，因此一切都有一个类型。不过，Scala 并不要求开发者一定要声明类型，它可以自己推断出来。因此，用 Scala 编码需要更少的输入，并且代码看上去更简洁。

## 1.2.2 数据类型

与 Java 语言类似，Scala 语言内置了基本的数据(变量)类型以及在这些类型上允许执行的运算符。

与 Java 语言不同，Scala 没有原始数据类型（基本数据类型）。在 Scala 中，所有的数据类型都是对象，这些对象具有操作其自身数据的方法。当一个 Scala 程序被编译为 Java 字节码时，编译器会自动地将 Scala 类型转换为 Java 的原始数据类型，以尽可能地优化程序性能。

Scala 的所有数据类型，从数字到集合，都是类型层次结构的一部分。在 Scala 中定义的每个类也将自动属于这个层次结构。Scala 核心数据类型层次结构如下图所示：



### Any、AnyVal 和 AnyRef

这三者为 Scala 类型层次结构的根。继承自 AnyVal 的数据类型称为“值类型”。继承自 AnyRef 的数据类型称为“引用类型”。

#### 1、值类型介绍如下：

Numeric 数据类型用来表示数值，如下表所示：

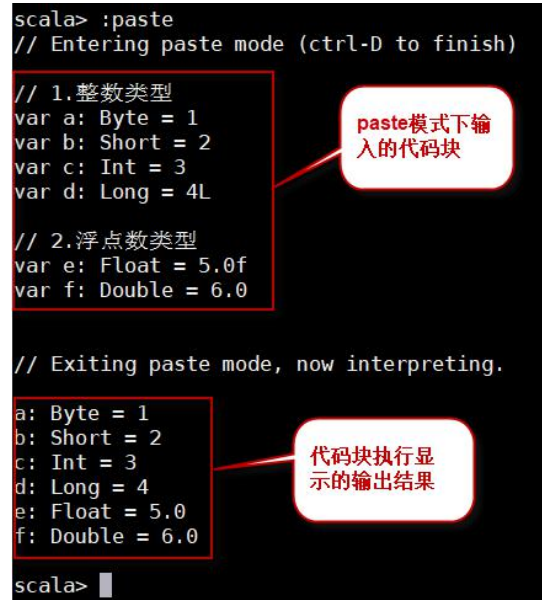
数据类型	描述	大小	最小值	最大值
Byte	有符号整数	1个字节	-128	127
Short	有符号整数	2个字节	-32768	32767
Int	有符号整数	4个字节	-2147483648	2147483647
Long	有符号整数	8个字节	-9223372036854775808	9223372036854775807

Float	有符号浮点数	4个字节	n/a	n/a
Double	有符号浮点数	8个字节	n/a	n/a

Scala 支持自动将数字从一种类型转换为另一种类型的能力，转换顺序如下：

Byte -> Short -> Int -> Long -> Float -> Double。

请看下面的代码：



```
scala> :paste
// Entering paste mode (ctrl-D to finish)

// 1. 整数类型
var a: Byte = 1
var b: Short = 2
var c: Int = 3
var d: Long = 4L

// 2. 浮点数类型
var e: Float = 5.0f
var f: Double = 6.0

// Exiting paste mode, now interpreting.

a: Byte = 1
b: Short = 2
c: Int = 3
d: Long = 4
e: Float = 5.0
f: Double = 6.0

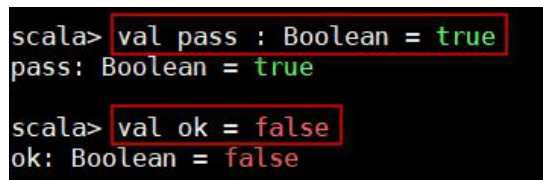
scala>
```

paste模式下输入的代码块

代码块执行显示的输出结果

如图中所示，Scala Shell 支持"paste"模式。当需要输入多行代码一起执行时，可以先键入":paste"，进入 paste 模式，然后输入多行代码。当按下 Ctrl + D 键时，退出 paste 模式，并执行代码。

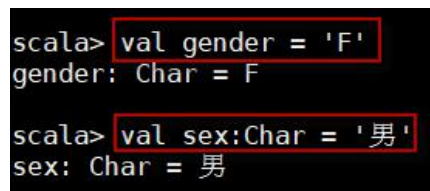
Boolean 类型代表逻辑值，可以是 true 或 false。如下所示：



```
scala> val pass : Boolean = true
pass: Boolean = true

scala> val ok = false
ok: Boolean = false
```

Char 类型代表字符，即用单引号括起来的单个字符。如下所示：



```
scala> val gender = 'F'
gender: Char = F

scala> val sex:Char = '男'
sex: Char = 男
```

Unit 类型用于定义不返回数据的函数。它与 Java 中的 void 关键字类。

2、常用的引用类型介绍如下：

String 类型表示字符串，即用双引号括起来的零个或多个字符。例如：

```
scala> val name = "张三"
name: String = 张三

scala> val address:String = "北京市中关村南大街"
address: String = 北京市中关村南大街

scala> 
```

字符串类型是我们在数据处理中涉及比较多的类型，关于字符串更详细的用法，在本章 1.8 小节将会有更详细的讲解。

### 1.2.3 运算符

数据存储 在变量中，要对数据进行运算，就需要使用运算符。Scala 语言中提供了这几种运算符：算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符等。

注：实际上，Scala 没有传统意义上的运算符。在 Scala 中，一切皆对象，运算符是定义在对象上的方法。但是为了学习上的方便，我们还在这里称为运算符。

算术运算符如下表所示：

运算符	描述	说明
+	加法运算	10 + 3，结果是13
-	减法运算	10 - 3，结果是7
*	乘法运算	10 * 3，结果是30
/	除法运算	10 / 3，结果是3
%	求余数运算	10 % 3，结果是1

在 scala shell 中输入以下示例代码，掌握算术运算符的使用：

```
// 算术运算符
scala> val a = 7
a: Int = 7

scala> val b = 3
b: Int = 3

scala> a + b
res0: Int = 10

scala> a - b
res1: Int = 4

scala> a * b
res2: Int = 21

scala> a / b
res3: Int = 2

scala> a % b
res4: Int = 1
```

关系运算符如下表所示：

运算符	描述	说明
==	判断两个值是否相等，结果为true或false	10 == 3，结果是false
!=	判断两个值是否不相等，结果为true或false	10 != 3，结果是true
>	判断左值是否大于右值，结果为true或false	10 > 3，结果是true
<	判断左值是否小于右值，结果为true或false	10 < 3，结果是false
>=	判断左值是否大于等于右值，结果为true或false	10 >= 3，结果是true
<=	判断左值是否小于等于右值，结果为true或false	10 <= 3，结果是false

在 scala shell 中输入以下示例代码，掌握关系运算符的使用：

```
scala> val a = 7
a: Int = 7

scala> val b = 3
b: Int = 3

scala> a == b
res5: Boolean = false

scala> a != b
res6: Boolean = true

scala> a > b
res7: Boolean = true

scala> a < b
res8: Boolean = false

scala> a >= b
res9: Boolean = true

scala> a <= b
res10: Boolean = false
```

逻辑运算符如下表所示：

运算符	描述	说明
&&	逻辑与运算，结果为true或false	10>3 && 3>5，结果是false
	逻辑或运算，结果为true或false	10>3    3>5，结果是true
!	逻辑非运算，结果为true或false	!true，结果是false

在 scala shell 中输入以下示例代码，掌握逻辑运算符的使用：

```
scala> true && true
res11: Boolean = true

scala> true && false
res12: Boolean = false

scala> false && true
res13: Boolean = false

scala> false && false
res14: Boolean = false
```



```
scala> true || true
res15: Boolean = true

scala> true || false
res16: Boolean = true

scala> false || true
res17: Boolean = true

scala> false || false
res18: Boolean = false

scala> !true
res19: Boolean = false

scala> !false
res20: Boolean = true
```

位运算符如下表所示：

运算符	描述	说明
&	按位与运算	0 & 1, 结果是0
	按位或运算	0   1, 结果是1
~	按位取反运算	~12, 结果是-13; ~(12), 结果是1
^	按位异或运算	1 ^ 1, 结果是0
<<	左移位运算	2 << 2, 结果是8
>>	右移位运算	4 >> 2, 结果是1
>>>	无符号右移位运算	4 >>> 2, 结果是1

位运算符在实际中使用较少。

赋值运算符如下表所示：

运算符	描述	说明
=	赋值运算，将右边的值赋给左边的变量	var a = 3, 为变量a赋值
+=	将左边变量值与右边值相加后，再赋值给左边的变量	a += 2, 相当于 a = a+2
-=	将左边变量值与右边值相减后，再赋值给左边的变量	a -= 2, 相当于 a = a-2
*=	将左边变量值与右边值相乘后，再赋值给左边的变量	a *= 2, 相当于 a = a*2
/=	将左边变量值与右边值相除后，再赋值给左边的变量	a /= 2, 相当于 a = a/2
%=	将左边变量值与右边值求余后，再赋值给左边的变量	a %= 2, 相当于 a = a%2
<<=	将左边的变量值按位左移位后，再赋值给左边的变量	a <<= 2, 相当于 a = a<<2
>>=	将左边的变量值按位右移位后，再赋值给左边的变量	a >>= 2, 相当于 a = a>>2
&=	将左边变量值与右边值按位与后，再赋值给左边的变量	a &= 2, 相当于 a = a&2
=	将左边变量值与右边值按位或后，再赋值给左边的变量	a  = 2, 相当于 a = a 2
^=	将左边变量值与右边值按位异或后，再赋值给左边的变量	a ^= 2, 相当于 a = a^2

在 scala shell 中输入以下示例代码，掌握赋值运算符的使用：

```
scala> var a = 3 // 注意这里。因为是重复赋值运算，所以使用关键字 var 声明变量
a: Int = 3

scala> a += 2; a
res24: Int = 5
```

```
scala> a -= 2; a
res25: Int = 3

scala> a *= 2; a
res26: Int = 6

scala> a /= 2; a
res27: Int = 3

scala> a %= 2; a
res28: Int = 1

scala> a << 2; a
res29: Int = 1

scala> a >> 2; a
res30: Int = 1

scala> a <=< 2; a
res31: Int = 4

scala> a >=> 2; a
res32: Int = 1

scala> a &= 2; a
res33: Int = 0

scala> a |= 2; a
res34: Int = 2

scala> a ^= 2; a
res35: Int = 0
```

对象相等性比较：在 Scala 中比较的是值，而不是 Java 概念中的地址值。请看下面的代码：

```
scala> ("he" + "llo") == "hello"
res33: Boolean = true

scala> 1 == 1.0
res34: Boolean = true
```

要比较 2 个不同的对象，使用三个等号（===）。

在实际进行计算的时候，经常会多个不同的运算符混合在一起进行运算，这时就需要考虑运算符的优先级问题。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。

下表中显示了 Scala 中不同运算符的优先级：

优先级	运算符	关联性
1	() []	从左到右
2	! ~	从右到左

3	* / %	从左到右
4	+ -	从左到右
5	>> >>> <<	从左到右
6	> >= < <=	从左到右
7	== !=	从左到右
8	&	从左到右
9	^	从左到右
10		从左到右
11	&&	从左到右
12		从左到右
13	= += -= *= /= %= >>= <<= &=  = ^=	从右到左
14		从左到右

【例】请编写 scala 代码，计算工人报酬。计算原则如下：前 40 个小时，按小时付报酬，每小时 50 元；超过 40 小时，应该付 1.5 倍报酬。

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val rate = 50.00          // 每小时薪酬
val totalHours = 45       // 工时

val overTimeHours = totalHours - 40    // 计算超工时

var pay = (rate * 40) + ((rate * 1.5) * overTimeHours)
println("工作时间" + totalHours + "小时，应获得报酬" + pay)
```

按下 Ctrl + D 键，退出:paste 模式，并执行以上代码。输出结果如下：

```
工作时间 45 小时，应获得报酬 2375.0
rate: Double = 50.0
totalHours: Int = 45
overTimeHours: Int = 5
pay: Double = 2375.0
```

## 1.2.4 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。Scala 中数组分为定长数组和变长数组。

创建定长数组 Array 的两种方式：

- ☐ 先创建，后赋值；
- ☐ 创建的同时赋初值。

例如，在下面的代码中，创建了一个名为 names 的数组，用来存储 3 个人的姓名：

```
val names = Array("张三","李四","王老五")
names(0)
names(1)
names(2)
names(0) = "张小三"          // 数组元素的值可以改变
names(0)
```

```
val numbers = Array(1,2,3,4,5)
numbers(0) + numbers(1)
```

输出如下所示：

```
names: Array[String] = Array(张三, 李四, 王老五)
res16: String = 张三
res17: String = 李四
res18: String = 王老五
res20: String = 张小三

numbers: Array[Int] = Array(1, 2, 3, 4, 5)
res21: Int = 3
```

也可以先创建一个空的数组，然后再赋值，如下面的代码所示：

```
// 创建一个长度为 3 的整型数组，所有元素初始化为 0
val arrs = new Array[Int](3)
arrs(0)
arrs(0) = 1
arrs(1) = 11
arrs

// 创建一个长度为 3 的字符串数组，所有元素初始化为 null
val arrs1 = new Array[String](3)
arrs1(0)

// 创建一个长度为 3 的布尔数组，所有元素初始化为 false
val arrs2 = new Array[Boolean](3)
arrs2(0)
```

输出结果如下：

```
arrs: Array[Int] = Array(0, 0, 0)
res9: Int = 0
res12: Array[Int] = Array(1, 11, 0)

arrs1: Array[String] = Array(null, null, null)
res13: String = null

arrs2: Array[Boolean] = Array(false, false, false)
res14: Boolean = false
```

也可以使用工具方法赋值(在 Scala 中，所有的工具方法都是在伴生对象中定义的)：

```
val array1 = Array.fill(5)(3.5)
val array2 = Array.fill(3)(math.random)
```

输出结果如下：

```
array1: Array[Double] = Array(3.5, 3.5, 3.5, 3.5, 3.5)
array2: Array[Double] = Array(0.6765717365372399, 0.936889692262062, 0.9403123661472736)
```

注：关于数组更详细的内容，我们将在 1.4 小节详细讲解。

## 1.3 程序流程控制

Scala 同样提供了对程序流程控制的语法。Scala 中的程序流程控制结构虽然与 Java 类似，但也有自己的一些独特的方法。

### 1.3.1 选择结构

Scala 的 if/else 语法结构和 Java 或者 C++ 一样。不过，在 Scala 中 if/else 表达式会返回一个值，这个值就是跟在 if 或 else 之后的表达式的值。

```
val x = if (a) y else z
```

请看下面的示例代码：

```
val age = 19

// 单分支选择结构
if (age < 20) {
  println(222)
}

// 双分支选择结构
if (age > 20) {
  println(444)
} else {
  println(555)
}

// 多分支选择结构
if (age < 18) {
  println(666)
} else if (age <= 36) {
  println(777)
} else if (age <= 60) {
  println(888)
} else {
  println(999)
}

// 三元运算符
if(!true) 1 else 0
```

输出结果如下：

```
age: Int = 19
222
555
777

res7: Int = 0
```



执行下面的代码：

```
val score = 60

// 作为表达式，值赋给变量 result
val result = if(score>=60) "及格" else "不及格"

// 注意：返回的是公共类型 Any
val result1 = if(score>=60) "及格" else 0

// 等同于 if(score<60) "不及格" else ()
val result2 = if(score<60) "不及格"
```

输出结果如下：

```
score: Int = 60
result: String = 及格
result1: Any = 及格
result2: Any = ()
```

### 1.3.2 循环结构

经常地，我们需要遍历集合中的元素，或者对集合中的每个元素进行操作，或者从现有集合创建一个新集合，这就需要用到循环结构。Scala 拥有与 Java 和 C++ 相同的 while 和 do-while 循环。

下面是使用 while 的循环语句：

```
var sum = 0
var i = 1
while(i < 11) {
    sum += i
    i += 1
}
sum
i
```

输出结果如下：

```
sum: Int = 55
i: Int = 11

res26: Int = 55
res27: Int = 11
```

请看下面的示例代码：

```
//输出 1 到 10 的偶数
//方法 1
var j = 1
while (j <= 10) {
    //如果 j 是偶数
```

```

    if (j % 2 == 0) {
        println(j)
    }
    j = j + 1
}

```

```

//方法 2
var k = 2
while (k <= 10) {
    println(k)
    k = k + 2
}

```

输出结果如下：

```

j: Int = 1
2
4
6
8
10

```

```

k: Int = 2
2
4
6
8
10

```

Scala 也有 do-while 循环，它和 while 循环类似，只是检查条件是否满足在循环体执行之后检查。例如：

```

var sum = 0
var i = 1
do {
    sum += i
    i += 1
} while(i <= 10)

println("1+...+10=" + sum)

```

Scala 也提供了 for 循环和 foreach 方法。下面的代码使用 for-each 循环来遍历数组：

```

// 定义一个字符串数组
val fruits = Array("apple", "banana", "orange")

// 输出数组中每个元素，每个一行(println 会输出换行符)
for (e <- fruits)
    println(e)

// 将数组中每个字符串转换为大写以后输出
for (e <- fruits) {

```

```
    val s = e.toUpperCase
    println(s)
}
```

可以在 for 表达式内部定义变量，然后，可以在 for 表达式的循环体中重用这些变量，这称为"变量绑定"。如下代码所示：

```
val books = Array("Scala 从入门到精通",
                  "Groovy 从入门到精通",
                  "Java 从入门到精通",
                  "24 小时精通 Scala",
                  "24 小时精通 Java")

for {book <- books
    bookVa1 = book.toUpperCase()
    bookVa2 = book.toLowerCase()
} println(bookVa1 + "," + bookVa2)
```

注意 to 和 until 的区别：

```
for( a <- 1 to 10){
    println( "a = " + a );
}

println("-----")

for( a <- 1 until 10){
    println( "a = " + a );
}
```

输出结果如下所示：

```
a = 1
a = 2
a = 3
a = 4
a = 5
a = 6
a = 7
a = 8
a = 9
a = 10
-----
a = 1
a = 2
a = 3
a = 4
a = 5
a = 6
a = 7
a = 8
a = 9
```

【示例】遍历数组。

```
// 声明数组
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);
```

【示例】登录用户名密码的游戏：三次机会，从控制台输入输入用户名密码。如果成功登录，返回登录成功，失败，则反馈错误信息！代码实现如下所示：

```
import scala.io.StdIn._

val dbUser = "cda"
val dbPassword = "emtf"
var count = 3

while(count > 0) {
    val name = readLine("亲，请输入用户名： ")
    val pwd = readLine("亲，请输入密码： ")
    if(name == dbUser && pwd == dbPassword) {
        println("登陆成功，正在为您跳转到主页呐，" + name + "^_^")
        count = 0
        // return
    } else {
        count -= 1
        println("连用户名和密码都记不住，你一天到底在弄啥嘞！您还有<" + count + ">次机会")
    }
}
```

在 for 循环中可以使用分号 (;) 来设置多个区间，它将迭代给定区间所有的可能值。以下示例代码演示了两个区间的循环实例：

```
for( a <- 1 to 3; b <- 1 to 3){
    print( "a = " + a )
```

```
        println( " , b = " + b )
    }
```

输出结果如下所示：

```
a = 1, b = 1
a = 1, b = 2
a = 1, b = 3
a = 2, b = 1
a = 2, b = 2
a = 2, b = 3
a = 3, b = 1
a = 3, b = 2
a = 3, b = 3
```

【示例】遍历二维数组。

```
// 处理多维数组
var myMatrix = Array.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; j <- 0 to 2) {
    print(" " + myMatrix(i)(j));
    if(j==2) println()
}

// 多维数组
val array = Array.ofDim[Int](2, 2)
array(0)(0) = 0
array(0)(1) = 1
array(1)(0) = 2
array(1)(1) = 3

for {
    i <- 0 to 1
    j <- 0 to 1
} println(s"($i)($j) = ${array(i)(j)}")
```

【示例】编程输出九九乘法表。

```
// 九九乘法表 1
for(i <- 1 to 9) {
    for(j <- 1 to i){
        var ret = i * j
        print(s"$i*$j=$ret\t")
    }
}
```



```

    }
    println()
}

// 九九乘法表 2
for(i <- 1 to 9; j <- 1 to i) {
    var ret = i * j
    print(s"$i*$j=$ret\t")
    if(i == j) {
        println
    }
}

```

### 1.3.3 中断循环

Scala 并没有提供 `break` 或 `continue` 语句来退出循环。那么如果需要 `break` 时我们该怎么做呢？有如下几个选项：

- 1) 使用 `Boolean` 型的控制变量。
- 2) 使用嵌套函数—可以从函数当中 `return`。
- 3) 使用 `Breaks` 对象中的 `break` 方法：

```

import scala.util.control.Breaks._

var n = 15
breakable {
    for(c <- "Spark Scala Storm") {
        if(n == 10) {
            println()
            break
        } else {
            print(c)
        }
        n -= 1
    }
}

```

输出内容如下：

```

import scala.util.control.Breaks._
n: Int = 15
Spark

```

### 1.3.4 循环中使用计数器

使用 `for-each` 循环有一个缺点，就是无法获取当前索引值。为此，可使用循环计数器。请看以下的代码：

```

val a = Array("张三", "李四", "王老五")

```

```
// 循环计数
for (i <- 0 until a.length) {
    println(s"$i: ${a(i)}")
}
```

输出结果如下：

```
0: 张三
1: 李四
2: 王老五
```

Scala 集合还提供了一个 `zipWithIndex` 方法可用于创建循环计数器：

```
for ((e, index) <- a.zipWithIndex) {
    println(s"$index: $e")
}
```

输出结果如下：

```
0: 张三
1: 李四
2: 王老五
```

实际上，对于数组/集合类型，它还有一个 `indices` 方法，能生成此序列的所有索引的范围：

```
val na = Array("张三","李四","王老五")

// 循环计数
for (i <- na.indices) {
    println(s"$i: ${na(i)}")
}
```

### 1.3.5 循环中使用条件过滤

可以遍历的同时过滤：

```
// for 语句
for (e <- 1 to 5) {
    println(e)
}

// 只输出偶数
for (e <- 1 to 10 if e % 2 == 0) {
    println(e)
}

// 迭代
val books = Array("Scala 从入门到精通", "Groovy 从入门到精通",
                  "Java 从入门到精通", "24 小时精通 Scala", "24 小时精通 Java")

for(book <- books){
    print(book + " || ")
}
println("\n-----")
```

```
// 过滤
for(book<-books if book.contains("Scala")){
    println(book)
}
```

Scala 可以使用一个或多个 if 语句来过滤一些元素：

```
val numList = Array(1,2,3,4,5,6,7,8,9,10);

// for 循环
for( a <- numList
    if a != 3; if a < 8 ){
    println( "a = " + a );
}
```

输出结果如下：

```
a = 1
a = 2
a = 4
a = 5
a = 6
a = 7
```

可以为嵌套循环通过 if 表达式添加条件：

```
for (i <- 1 to 3; j <- 1 to 3 if i != j)
    print ((10 * i + j) + " ")

println

// if 表达式是否添加括号，结果无变化：
for (i <- 1 to 3; j <- 1 to 3 if (i != j))
    print ((10 * i + j) + " ")
```

输出结果如下：

```
12 13 21 23 31 32
12 13 21 23 31 32
```

**【示例】** 在 for 条件表达式中使用过滤。

```
// 打印所有的偶数
for (i <- 1 to 10 if i % 2 == 0)
    println(i)

// 也可写成下面这样
for {
    i <- 1 to 10
    if i % 2 == 0
} println(i)

// 多条件
for {
```

```

    i <- 1 to 10
    if i > 3
    if i < 6
    if i % 2 == 0
} println(i)

```

### 1.3.6 简单模式匹配

Scala 中没有提供与 Java 语言中 `switch` 类似的语法，但是提供了一个更加强大的模式匹配功能。模式匹配是一个 Scala 概念，它看上去与其它语言中的一个 `switch` 语句很类似。不过，它是一个比 `switch` 语句更强大的工具。（根据值检查模式的过程称为模式匹配）

Scala 模式匹配使用关键字 `match`。每个可能的匹配由关键字 `case` 处理。如果有一个 `case` 被匹配到，那么  $\Rightarrow$  右侧的代码被执行。其中下划线 (`_`) 代表默认 `case`。如果前面没有一个 `case` 匹配上的话，默认 `case` 的代码会被执行。与 `switch` 语句不同，在每个 `case` 后的代码不需要 `break` 语句。只有匹配的 `case` 会被执行。另外，每个  $\Rightarrow$  右侧的代码是一个表达式，返回一个值。因此，一个模式匹配语句本身是一个表达式，返回一个值。

请看下面的示例：

```

// 根据输入的数字，给出对应的星期
val day = 8
val month = day match{
    case 1 => "星期一"
    case 2 => "星期二"
    case 3 => "星期三"
    case 4 => "星期四"
    case 5 => "星期五"
    case 6 => "星期六"
    case 7 => "星期天"
    case _ => "不正确"
}
println(month)

```

输出结果为：

不正确

【示例】从控制台输入一个整数，给出对应的月份描述。

```

import java.io._

print("控制台输入一个整数 ： ")
val br = new BufferedReader(new InputStreamReader(System.in));
val line = br.readLine().toInt
val month = line match {
    case 1  => "一月份"
    case 2  => "二月份"
    case 3  => "三月份"
    case 4  => "四月份"
    case 5  => "五月份"
}

```

```

        case 6 => "六月份"
        case 7 => "七月份"
        case 8 => "八月份"
        case 9 => "九月份"
        case 10 => "十月份"
        case 11 => "十一月份"
        case 12 => "十二月份"
        case _ => "无效的月份值" // 默认值
    }

    println("month:" + month)
    br.close()

```

说明：在 Scala 2.11 中，可以用 `scala.io.StdIn.readLine()` 函数从控制台读取一行输入。如果要读取数字、Boolean 或者是字符，可以用 `readInt`、`readDouble`、`readByte`、`readShort`、`readLong`、`readFloat`、`readBoolean` 或者 `readChar`。

```

import scala.io.StdIn.readLine

val input = readLine("请输入内容：")
print("您输入的是：")
println(input)
printf("您的输入内容是%s",input)

```

一个 case 语句也可以匹配多个条件。请看下面的代码：

```

val day = 6
val weekDay = day match{
    case 1|2|3|4|5 => "工作日"
    case 6|7 => "周末"
    case _ => "不正确"
}

```

匹配更多的类型：

```

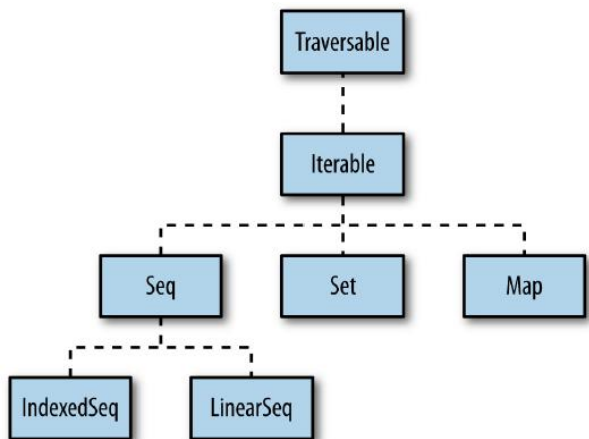
println("-----")
// 一个 case 语句匹配多个条件
// 匹配多个值
val i = 5
i match {
    case 1 | 3 | 5 | 7 | 9 => println("奇数")
    case 2 | 4 | 6 | 8 | 10 => println("偶数")
}

// 匹配字符串类型
val cmd = "stop"
cmd match {
    case "start" | "go" => println("starting")
    case "stop" | "quit" | "exit" => println("stopping")
    case _ => println("doing nothing")
}

```

## 1.4 集合

Scala 有一个丰富的集合库，包含很多不同类型的集合。此外，所有的集合都暴露出相同的接口。因此，一旦熟悉了其中一个 Scala 集合，就可以很容易地使用其它集合类型。



Scala 中的集合体系主要包括：Iterable、Seq (IndexSeq)、Set (SortedSet)、Map (SortedMap)。其中 Iterable 是所有集合 trait 的根 trait。实际上 Seq、Set、和 Map 都是子 trait。

- ❑ Seq: 是一个有先后次序的值的序列，比如数组或列表。IndexSeq 允许我们通过它们的下标快速的访问任意元素。举例来说，ArrayBuffer 是带下标的，但是链表不是。
- ❑ Set: 是一组没有先后次序的值。在 SortedSet 中，元素以某种排过序顺序被访问。
- ❑ Map: 是一组（键、值）对。SortedMap 按照键的排序访问其中的实体。

Scala 中的集合是分成可变和不可变两类集合的。可变集合的内容或引用可以更改，不可变集合不能更改。这两种类型的集合分别对应 Scala 中的 scala.collection.mutable 和 scala.collection.immutable 两个包。

Scala 中常用的集合如下表所示：

集合	描述
List	是一个相同类型元素的线性序列
Set	类型相同但没有重复的元素的集合
Map	键/值对的集合
Tuple	不同类型但大小固定的元素的集合
Option	包含0个或1个元素的容器

下面的代码简要描述了这几种集合形式：

```
val booksList = List("Spark", "Scala", "Python", "Spark")
val booksSet = Set("Spark", "Scala", "Python", "Spark")
val booksMap = Map(101 -> "Scala", 102 -> "Scala")
val booksTuple = new Tuple5(101, "Spark", "xinliwei", "机械工业出版社", 65.50)
```

### 1.4.1 Seq

序列表示一个以特定顺序排列的元素集合。因为该元素有个定义好的顺序，所以可以按照它们在集合中的位置进行访问。例如，可以请求序列中第 n 个元素。Seq 下包含了 Range、

ArrayBuffer、List 等子 trait。其中 Range 就代表了一个序列，通常可以使用“1 to 10”这种语法来产生一个 Range。ArrayBuffer 就类似于 Java 中的 ArrayList。

序列有三个实现类，分别是 Array、List 和 Vector。

### 1) Array (数组)

在 Scala 中，数组分为不可变数组和可变数组。不可变数组的实现类为 Array，可变数组的实现类为 ArrayBuffer。Array 是一个索引的元素序列，有一个固定的长度。在一个 Array 中的所有元素都是相同的类型。它是一个可变的数据结构；可以修改数组中的元素，不过，在一个 Array 被创建以后，不可以再向 Array 添加元素。Array 中的元素有一个基于零的索引。要获得或更新一个元素，在括号内指定其索引。例如：

```
val arr = Array(10,20,30,40);
arr(0) = 50;
val first = arr(0);
```

在一个数组上的基本操作包含：

- ❑ 根据索引提取一个元素
- ❑ 使用索引更新一个元素

在下面的示例代码中，演示了数组的创建及操作：

```
// 1.Array 创建
var arr1 = new Array[Int](10)
var arr2 = Array(1, 5, 3, 7)

// 2.ArrayBuffer 创建
var arr3 = new ArrayBuffer[Int]()
var arr4 = ArrayBuffer(10, 50, 30, 70)

// 3.共同方法
println("-----Array 和 ArrayBuffer 共同方法-----");
println(arr2.sum)    // 16
println(arr2.max)    // 7
println(arr2.min)    // 1
println(arr2.mkString("|"))    //结果是 1|3|5|7
println(arr2.sorted.toBuffer)    // 1 3 5 7
println(arr2.reverse.toBuffer)    // 7 3 5 1
println(arr4.toArray.getClass)    // class [I
println(arr2.toBuffer.getClass)    // class scala.collection.mutable.ArrayBuffer
println(Array(("cn", "china"), ("fr", "french")).toMap)    // 将数组(元素是元组类型)转换为 Map

// 4.ArrayBuffer 的独有方法
println("-----ArrayBuffer 独有方法-----");
arr4 += 20            // 增加一个元素
println(arr4)        // 10 50 30 70 20

arr4 ++= Array(50, 60)    // 增加一个数组集合
println(arr4)            // 10 50 30 70 20 50 60

arr4.trimEnd(3)        // 删除最后 3 个元素
println(arr4)          // 10 50 30 70
```



```

arr4.insert(2, 28, 29)    // 在索引 2 处插入两个元素
println(arr4)            // 10 50 28 29 30 70

arr4.remove(2, 3)        // 在索引 2 处删除三个元素
println(arr4)            // 10 50 70

arr4.clear()             // 清空数组
println(arr4)

// 5.遍历
println("-----5.遍历-----");
var arr5 = Array(1, 5, 3, 7)

// i) 直接取值
for (i <- arr5) {
    println(i)            // 1 5 3 7
}

// ii)通过下标
for (i <- arr5.indices) {
    println(arr5(i))      // 1 5 3 7
}

// 多维数组
println("-----多维数组-----")
var arr6 = Array(Array(1, 3, 5), Array(2, 4, 6))
for (i <- arr6) {
    for(j <- i) {
        print(j + " ")
    }
    print("\n")
}

```

下面的代码演示了在数组上的求和与排序：

```

// 1、求和与排序
println(Array(1,7,2,9).sum)

println("-----")
// 2、求最大值
println(ArrayBuffer("Mary","had","a","little","lamb").max)

println("-----")
// 3、排序
// 升序
val b = ArrayBuffer(1,7,2, 9)
val bSorted = b.sorted      //1,2,7,9
b.sortWith(<_<_).foreach(println)

```

```
println("-----")
// 降序
b.sortWith(_>_).foreach(println)

println("-----")
// 4、显示数组内容
println(b.mkString(" And "))           // 使用指定分隔符连接数组中的元素为字符串
println(b.mkString("<",">"))           // <1,7,2,9>, 指定前缀、分隔符、后缀
```

注：关于数组，在 1.2 节已经有详细讲解，此处不再赘述。

## 2) List (列表)

在 Scala 中，列表分为不可变的和可变的。不可变列表的实现类为 List，可变数组的实现类为 ListBuffer。List 是一个相同类型元素的线性序列。它是一个递归数据结构，不像数组，数组是一个扁平数据结构。另外，与数组不同，它是一个不可变的数据结构；List 被创建以后，不可以被修改，其大小以及元素不能被改变。List 是 Scala 中最常用的数据结构之一。

虽然可以通过元素的索引来访问 list 中的元素，但是通过索引访问元素不是一个高效的数据结构。访问时间与元素在 list 中的位置成正比。

Scala 的 List 是作为 Linked List 实现的，并提供有 head、tail 和 isEmpty 方法。因此，在 List 上的大多数操作涉及递归算法，将 list 拆分为 head 和 tail 部分。

创建 List 有两种方式：像 Array 一样，或者使用 :: 连接运算符。也可以将其他集合转换为 List 集合。下面的代码演示了创建一个列表的一些方式：

```
val xs = List(10,20,30,40);
val ys = (1 to 100).toList;
val zs = Array(1,2,3).toList;

// 创建一个空的 List
val empty: List[Nothing] = List()

// 也可用 Nil 创建空的列表
val empty = Nil

// 创建图书列表
val books: List[String] = List("Scala 从入门到精通", "Groovy 从入门到精通", "Java 从入门到精通")

// 使用 tail Nil 和 :: 来创建图书列表
val books = "Scala 从入门到精通" :: "Groovy 从入门到精通" :: "Java 从入门到精通" :: Nil

books.head    // 第一个元素
books.tail    // 除了第一个元素
```

可以使用 iterator 方法对集合进行迭代。iterator.hasNext 方法用于查找集合是否具有进一步的元素，而 iterator.next 方法用于访问集合中的元素。下面的代码描述了 iterator 方法：

```
val booksList = List("Spark","Scala","Python","Spark")

val iterator = booksList.iterator
while(iterator.hasNext){
    println(iterator.next)
```

```
}
```

输出如下图所示：

```
scala> iteratingList(booksList)
Spark
Scala
R Prog
Spark
```

在一个 list 上的基本操作包括：

- ❑ head 方法：提取第一个元素；
- ❑ tail 方法：提取除第一个元素之外的所有元；
- ❑ isEmpty 方法：判断一个 list 是否为空。如果一个 list 是空的，该方法返回 true。

下面的代码演示了 List 上的常用操作：

// 1、在 Scala 中，列表要么是 Nil（及空表），要么是一个 head 元素加上一个 tail，而 tail 又是一个列表

```
val list = List(1, 2, 3, 4, 5)
list.head
list.tail
list.isEmpty
list == Nil
```

// 下面是一个使用递归求 list 集合中和的例子：

```
def recursion(list:List[Int]):Int = {
    if(list.isEmpty) {
        return 0
    }
    list.head + recursion(list.tail)
}
```

// 增

```
/* A.++(B) --> 在列表 A 的尾部对添加另外一个列表 B,组成一个新的列表
 * A.++:(B) --> 在列表 A 的首部对添加另外一个列表 B,组成一个新的列表
 * A.:::(B) --> 在列表 A 的首部对添加另外一个列表 B,组成一个新的列表
 * -----
 * A.:+ (element) -->在列表 A 的尾部添加一个 element，组成一个新的集合
 * A.+:(element) -->在列表 A 的首部添加一个 element，组成一个新的集合
 * A.: (element) -->在列表 A 的首部添加一个 element，组成一个新的集合
 */
```

```
val left = List(1, 2, 3, 4)
val right = List(5, 6, 7)
```

```
left.++(right)
left.++:(right)
left.:::(right)
left.:+(10)
left.+:(10)
left.: (10)
```

```

// 删除
// drop(n)          --->删除 list 的前 n 个元素(首部开始删除)
// dropRight(n)     --->删除 list 的后 n 个元素(尾部开始删除)
// dropWhile(p: A => Boolean) --->逐个匹配去除符合条件的元素，直到不符合条件，之后的元素不再判断

val list = List(1, 2, 3, 4, 5, 6, 7)
list.drop(2)
list.dropRight(3)
list.dropWhile(_ <= 3)
list.dropWhile(_ > 3)    // 第一个元素就不符合条件，后面的不再判断，所以一个也没有删除

// 修改与查询
val list = List(1, 2, 3, 4, 5, 6, 7)
list(0)
// list(0) = 10          // 值不能被修改

val list = List(1, 2, 13, 14, 15, 6, 7)
list.take(5)
list.takeWhile(_ <= 3)
list.takeWhile(_ > 3)    // 获取的是最满足条件的最长前缀
list.mkString
list.mkString(";")
list.count(_ % 2 == 0)

val fruit = "apples" :: ("oranges" :: ("pears" :: Nil)) // ::操作符从给定的头和尾部创建一个新的列表

val list1 = List(1, 2, 3)
val list2 = List(4, 5, 6)

list1 ++ list2    // ++两个集合之间的操作

list1.sum

```

下面的代码示例中，演示了列表的创建及操作：

```

//1. List 创建
// 方式一：
var list1 = List("aa", "bb", "ccc")
list1(0)
list1(1)
list1(2)
// list1(0) = "aaaaa"    // 不可以，因为 List 不可以改变

//方式二：右操作符：当方法名以:结尾时，为右操作符.先做右边
// :: 右操作符，拼接元素
var list3 = "aa" :: "bb" :: "cc" :: Nil          // Nil 是空集合，先做"cc"::Nil，其中::是 Nil 的方法
var list4 = list1 ::: "dd" :: Nil                // :::拼接集合

```

```
// 还可以
val x = List.tabulate(5)(n => n * n)
x

val y = List.range(1, 10)
Y

// 2.方法
// 检索数据
// head() 第一个元素
list1.head

// tail() 除了第 1 个元素外的全部元素
list1.tail
list1.tail.head
list1.tail.tail

// take(n) 取前 n 个元素 - 子集
list1.take(2)

// init() 除了最后的一个全部元素
list1.init

// 3.ListBuffer 创建
import scala.collection.mutable.ListBuffer
var list5 = ListBuffer("111", "222", "333")

// 增
// +=或 append 追加元素
list5.append("444")
list5 += "555"
// list5 += ("666","777")
list5 :+ "000"
"000" +=: list5

// += 追加数组/列表
list5 +=: List("666", "777")

// 丢弃前 3 个元素
list5.drop(3)

// 判断是否为空
list5.isEmpty

// 翻转
list5.reverse

// splitAt(m): 把列表前 m 个做成一组, 后面做为另一组
list5.splitAt(3)
```

```

// flatten: 扁平化
var list6 = List(List('a', 'b'), List('c'), List('d'))
list6.flatten

// 转换成数组
list5.toArray

// zip:两个 List 内部的元素合并 - 拉链方法
var list7 = List(1, 2, 3).zip(List(4, 5, 6))

// grouped(n): 每 n 个元素分成 1 组,返回是迭代器 Iterator, 可以再用 toList 转成 list 类型
List(1, 3, 5, 7, 9).grouped(2).foreach(println)
List(1, 3, 5, 7, 9).grouped(2).toList

println("-----")
// 遍历方式一: 不推荐
for (e <- List("88", "99")) {
  println(e)
}

println("-----")
// 遍历方式二: 使用 foreach, 推荐,这才是采用函数式编程
List("88", "99").foreach(e => println(e))    // foreach 是高阶函数
println("-----")
List("88", "99").foreach(println(_))        // 简化写法 _占位符
println("-----")
List("88", "99").foreach(println)           // 最简方法

```

### 3) Vector (向量)

Vector 类是 List 和 Array 类的混搭。它结合了 Array 和 List 的执行特性。它提供了恒定时间复杂度的索引访问和线性访问。它既允许快速的随机访问，也允许快速的更新功能。

使用向量是简单而直接的：

```

val v = Vector(1, 2, 3)
v.sum           // 6
v.filter(_ > 1) // Vector(2, 3)
v.map(_ * 2)     // Vector(2, 4, 6)

```

下面是使用示例：

```

val v1 = Vector(0,10,20,30,40);
val v2 = v1 :+ 50;
val v3 = v2 :+ 60;
val v4 = v3(4);
val v5 = v3(5);

```

下面这个示例代码中，演示了 Vector 的创建和操作：

```

// 创建一个 Vector
val x = IndexedSeq(1,2,3)
// 通过索引进行访问
x(0)

```

```

// 不可修改，创建新的 Vector
val a = Vector(1,2,3)
val b = a ++ Vector(4,5)

// 使用 updated 方法替换一个元素
val c = b.updated(0,10)

// 使用常用的过滤方法
val a = Vector(1,2,3,4,5)
val b = a.take(2)
val c = a.filter(_>2)

// 貌似可变：var 声明，结果赋给自身
var a = Vector(1,2,3)
a = a ++ Vector(4,5)
a

// 混合可变变量(var)和一个不可变集合
var int = Vector(1)
int = int :+ 2
int = int :+ 3
int.foreach(println)

```

#### 4) Range（范围）

Range 定义一个范围，指定开始、结束和步长，通常用于填充数据结构和遍历 for 循环。

```

// 使用方法 to 来创建 Range(包含上限)
1 to 5

// 使用方法 until 来创建 Range(不包含上限)
1 until 5

// 还可以指定步长
1 to 21 by 4
1 until 21 by 4

'a' to 'c'

// 读取值
val r = 1 to 5
r(1)
val rr = 1.to(5)
rr(1)

// 使用 Range 创建一个连续序列
val x = (1 to 10).toList

```

#### 5) Stream（流）

与 List 类似，但是是延迟计算的，所以可以非常非常长。

```
// 创建
val stream = 1 #:: 2 #:: 3 #:: Stream.empty

// very very long
val stream = (1 to 100000000).toStream
stream.head
stream.tail
```

## 1.4.2 Set

Set 是一个不重复元素的无序集合。它不包含重复元素。此外，它不允许通过索引访问一个元素，因为它并没有索引。

下面是一个 set 的例子：

```
val fruits = Set("apple", "orange", "pear", "banana")
val set1 = Set(1,2,3,2,3,4,5,5)
```

Sets 支持两个基本操作：

- ❑ contains: 如果 set 包含输入参数，则返回 true;
- ❑ isEmpty: 如果 set 是空的，返回 true;

Set 是不重复元素的集合。尝试将已有元素加入进来是没有效果的。比如：

```
(Set(2,0,1) + 1).foreach(println(_))
```

Set 不保留元素插入的顺序。默认情况下，Set 是以 HashSet 实现的，其元素根据 hashCode 方法的值进行组织。

```
Set(1,2,3,5,7,8).foreach(println(_))
```

LinkedHashSet 可以记住元素被插入的顺序。它会维护一个链表来达到这个目的。

```
val weeks= scala.collection.mutable.LinkedHashSet("Mo","Tu","We","Th","Fr")
weeks.foreach(println(_))
```

如果想要按照已排序的顺序来访问其中的元素，使用 SortedSet：

```
scala.collection.immutable.SortedSet(1,2,3,4,5,6).foreach(println(_))
```

## 1.4.3 Map

Map 是一个 key-value 对的集合。在其它语言中，它被称为词典、关联数组、或哈希 map。这是一个根据 key 查找 value 的高效的数据结构。

下面的代码段演示了怎样创建和使用一个 Map：

```
val capitals = Map("美国" -> "华盛顿", "英国" -> "伦敦", "中国" -> "北京")
val indiaCapital:String = capitals("中国")
println(indiaCapital)
```



下面这个示例是对 Map 进行迭代。

```
val booksMap = Map(101 -> "Scala", 102 -> "Spark")

val iterator: Iterator[Int] = booksMap.keySet.iterator
while (iterator.hasNext) {
    val key = iterator.next
    // println(s"图书 Id:$key,图书名称:${booksMap.get(key)}")
    println(s"图书 Id:$key,图书名称:${booksMap(key)}")
}
```

输出如下图所示：

```
图书Id:101, 图书名称:{booksMap.get(key)}
图书Id:102, 图书名称:{booksMap.get(key)}
```

下面的代码示例演示的 Map 的创建和操作：

```
// 创建
var m1 = Map("zhang3" -> 20, "li4" -> 23, "wang5" -> 21)
var m2 = Map(("zhang3", 20), ("li4", 23), ("wang5", 21)) //使用元组

var m = scala.collection.mutable.Map(("zhang3", 20), ("li4", 23), ("wang5", 21))
// var m3 = scala.collection.mutable.Map[String, Int]() // 定义空 map，要用可变的才有意义。

// 添加和更新: += +=+=
m += (("zhao6", 22))
m +=+= Map(("a" -> 1), ("b" -> 2))
m

// updated, 没有增加，有了替换
m = m.updated("zheng7", 19) //和上面方法一样
m

m.put("wu", 22)
m

// 删除
m -= ("zhang3") // 删除一个元素：按指定的 key 删除

m.clear() // 清空
m

/* -- 查询 -- */
println("-----查询-----")
var m4 = Map(("zhang3", 20), ("li4", 23), ("wang5", 21))
m4("zhang3")
// m4("aaa") // 会出错，不友好
m4.get("aaa") // 友好方式，推荐
m4.getOrElse("aaa", 23)
m4.getOrElse("zhang3", 23)
```

```

/* -- 遍历 -- */
var m5 = Map(("zhang3", 20), ("li4", 23), ("wang5", 21))

// 方法 1:直接打印 entry
for (entry <- m5) {
  println(entry)
}

println("-----")
// 方法 2: 直接打印 key, 顺手输出 value
for (key <- m5.keySet) {
  println(key + ":" + m5(key))
}

println("-----")
for((k,v) <- m5){
  println(k + "," + v)
}

println("-----")
// 方法 3.foreach
m5.foreach(e => {
  val (k, v) = e
  println(k + "," + v)
})

println("-----")
// 方法 4
m5.foreach(e => println(e._1 + "," + e._2))

```

## 1.4.4 Tuple

一个 tuple(元组)是一个容器，用来存储两个或多个不同类型的元素。元组与 List 类似，大小和值不可变，它在创建以后不能被修改，但可以容纳不同数据类型。

可以通过两种方式创建元组：

- ❑ 通过编写由一对括号包围的、包含用逗号分开的值
- ❑ 通过使用关系操作符(->)

```

val tuple = (1, false, "Scala")
println(s"${tuple._1}, ${tuple._2}, ${tuple._3}")

val tuple2 ="title" -> "Scala 从入门到精通"
println(s"${tuple2._1}, ${tuple2._2}")

```

元组在我们想要将一组不相关的元素组织在一起时很有用。如果这些元素是相同类型，可以使用集合，比如数组或列表。如果元素是不同类型，但是是相关的，在这种情况下，元组可能更合适。

元组是不可迭代的，其目的只是作为能容纳多个值的容器。但在一个元组中的元素有一个基于 1 的索引，可以通过索引访问元组中的元素。下面的代码演示了访问元组中元素的语法：

```

val twoElements = ("10", true);
val threeElements = ("10", "harry", true);

val first = threeElements._1
val second = threeElements._2
val third = threeElements._3

```

下面的示例代码演示了元组的使用：

```

// 元组：容器，可以容纳多个不同类型的元素，不可改变，不可以迭代
// 主要应用场景：作为数据容器，在函数中做为返回值，可以一次返回多个值
val t1 = ("aa",3,3.45,true,'c')

// 获取其中的元素:基于 1 的位置索引
t1._1
t1._2
t1._3
t1._4
t1._5

println("-----")
val (a,b,c,d,e) = t1
a
b
c
d
e

```

### 1.4.5 Option

在 Scala 中，Option[T]是给定类型的 0 或 1 个元素的容器。Option 是一个数据类型，用来表明一些数据的"有"或"无"，可以是 Some[T]或 None[T]，其中 T 可以是任何给定类型。一个 Some 实例可以存储任何类型的数据。None 对象表示数据的缺失。

例如，如果 key 存在，Scala Map 总是返回值 Some[<given\_type>]，如果键不存在，则返回 None。

```

val booksMap = Map(101 -> "Scala", 102 -> "Scala")
println(booksMap.get(101))
println(booksMap.get(105))

```

输出结果如下图所示：

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val booksMap = Map(101 -> "Scala", 102 -> "Scala")
println(booksMap.get(101))
println(booksMap.get(105))

// Exiting paste mode, now interpreting.

Some(Scala)
None
```

Option 类型提供有一个 getOrElse 方法。当值不存在时，使用 getOrElse() 来访问值或使用默认值，可以得到比较友好的反馈（不会抛出空值异常）。例如：

```
println(booksMap.get(101).getOrElse("此图书不存在"))
println(booksMap.get(105).getOrElse("此图书不存在"))
```

输出结果如下图所示：

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

println(booksMap.get(101).getOrElse("此图书不存在"))
println(booksMap.get(105).getOrElse("此图书不存在"))

// Exiting paste mode, now interpreting.

Scala
此图书不存在
```

Option 数据类型与一个函数或方法一起使用，这个函数或方法可选地返回一个值。它可以返回 Some(x) - 这里 x 是实际返回的值，或者 None 对象 - 该对象代表一个缺失的值。由一个函数返回的可选值可以使用模式匹配读取。

下面的代码演示了使用方法：

```
def colorCode(color:String): Option[Int] = {
  color match{
    case "red" => Some(1)
    case "blue" => Some(2)
    case "green" => Some(3)
    case _ => None
  }
};

val code = colorCode("orange");
code match{
  case Some(c) => println("code for orange is:" + c);
  case None => println("code not defined for orange");
}
```

Option 数据类型帮助防止 null 指针异常。在许多语言中，null 被用来表示数据缺失。在 Scala 中联合使用强类型检查和 Option 类型以防止 null 引用错误。

Option 类型用来表示可能存在也可能不存在的值。Some 类（这是一个 Case class 类）包装

了某个值，而 `None`（这也是一个 `Case class` 类）表示没有值。`Option` 支持泛型。（`Option` 的语法必须掌握，因为 `Spark` 源码中有大量的实例是关于 `Option` 语法的）

下面是一个使用了 `Option` 类型的代码：

```
// 使用模式匹配提取 Option 变量中的值
def show(x: Option[String]) = x match {
  case Some(s) => s
  case None => "?"
}

// 定义一个函数
def optionOps: Unit = {
  val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo", "BeiJing" -> "通州")

  println(capitals.get("Japan") + "," + show(capitals.get("Japan")))
  println(capitals.get("BeiJing") + "," + show(capitals.get("BeiJing")))
}

optionOps
```

输出结果为：

```
Some(Tokyo),Tokyo
Some(通州),?
```

`Option` 类型另外还提供了一个 `isEmpty()` 方法，用来判断 `Option` 中是否有值：

```
val a:Option[Int] = Some(5)
val b:Option[Int] = None

// 使用 isEmpty()方法:
println("a.isEmpty: " + a.isEmpty) // false
println("b.isEmpty: " + b.isEmpty) // true
```

## 1.4.6 集合类上的高阶方法

`Scala` 集合的真正强大之处在于带来了其高阶方法。一个高阶方法使用一个函数作为其输入参数。需要特别注意的是，一个高阶方法并不改变集合。下面是 `Scala` 集合的一些最主要的高阶方法。

### 1、map

`Scala` 集合的 `map` 方法将其输入函数应用到集合中所有元素上，并返回另一个集合。返回的集合具有与调用 `map` 方法的那个集合相同数量的元素。不过，在返回的集合中的元素并不是原始集合中相同的类型。示例如下：

```
val xs = List(1,2,3,4);
val ys = xs.map((x:Int) => x*10.0);
```

在上面的代码中，`xs` 的类型是 `List[Int]`，而 `ys` 的类型是 `List[Double]`。

如果一个函数只有一个参数，那么圆括号可以被花括号替换。下面的两个语句等价：

```
val ys = xs.map((x:Int) => x*10.0);
```

```
val ys = xs.map{(x:Int) => x*10.0};
```

正如前面讲过的，Scala 允许使用运算符标记调用任何方法。要进一步提高可读性，前面的代码也可以写成下面这样：

```
val ys = xs map {(x:Int) => x*10.0};
```

Scala 可以从集合的类型推断出传入的参数类型，因此可以忽略掉参数类型。下面两个语句是等价的：

```
val ys = xs map {(x:Int) => x*10.0};
```

```
val ys = xs map {x => x*10.0};
```

如果一个函数字面量的输入参数只在函数体中使用一次，那么右箭头和该箭头的左侧可以从函数字面量中被删除。我们可以只编写函数字面量的函数体。下面的两个语句是等价的：

```
val ys = xs map {x => x*10.0};
```

```
val ys = xs map { _ * 10.0};
```

上面代码中，下划线(`_`)字符代表函数字面量的输入传给 `map` 方法。上面的代码可以被理解为集合 `xs` 中的每个元素乘以 10。

总结来说，下面的代码分别代表了相同语句的冗长的版本和简洁的版本：

```
val ys = xs.map{(x:Int) => x*10.0};
```

```
val ys = xs map { _ * 10.0};
```

下面是使用 `map` 方法的另一个示例：

```
// 快速产生 0.1, 0.2, 0.3 等方式的数字
```

```
(1 to 9).map(0.1 * _).foreach(println(_))
```

```
// 打印三角形
```

```
(1 to 9).map(" *" * _).foreach(println(_))
```

## 2、flatMap

Scala 集合的 `flatMap` 将集合中的所有集合元素展开，并形成单个集合。`flatMap` 方法类似于 `map`。它接收一个函数作为输入，并将该输入函数应用到集合中的每个元素，并返回另一个集合作为结果。不过，传递给 `flatMap` 的函数为原始集合中的每个元素都生成一个集合。因此，应用该输入函数的结果是一个集合的集合。如果相同的输入函数被传递给 `map` 方法，它将返回一个集合的集合。而 `flatMap` 方法则返回一个 `flattened` 集合。

下面的例子描述了 `flatMap` 的使用：

```
val numbersList = List(List(1,2,3), List(4,5,6), List(7,8,9))
```

```
numbersList.flatMap(list => list)
```

输出结果如下：

```
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

请看下面的代码：

```
val line = "Scala is fun";
```

```
val words = line.split(" "); // 按空格分割
```

```
val arrayOfChars = words flatMap {_.toList};
```

输出结果如下：

```
arrayOfChars: Array[Char] = Array(S, c, a, l, a, i, s, f, u, n)
```

集合的 `toList` 方法创建集合中所有元素的一个列表。对于将一个字符串、数组、`set`、或任

何其它集合类型转换为一个 list，该方法很有用。

### 3、filter

方法 `filter` 应用一个断言到集合中的每个元素，并返回另一个集合(由断言返回 `true` 的元素组成)。一个断言是返回一个 `Boolean` 值的函数。它返回 `true` 或 `false`。

```
val xs = (1 to 100).toList;
val even = xs filter { _ % 2 == 0};
```

使用 `filter` 方法，按照过滤条件，将原集合中不符合条件的数据过滤掉，输出所有匹配某个特定条件的元素，得到一个序列中的所有偶数：

```
(1 to 9).filter(line => line % 2 == 0).foreach(println(_))
(1 to 9).filter(_ % 2 == 0).foreach(println)    // 与上一句等价
```

### 4、foreach

Scala 集合的 `foreach` 方法在集合的每个元素上调用其输入函数，但并不返回任何东西。这类似于 `map` 方法。这两个方法间的唯一区别在于 `map` 返回一个集合，而 `foreach` 并不返回任何东西。由于它的副作用，这是一个很少使用的方法。

```
val words = "Scala is fun".split(" ");
words.foreach(println);
```

### 5、reduce

方法 `reduce` 返回单个值。正如其名所暗示的，它将一个集合归约为一个单个的值。方法 `reduce` 的输入函数同时接收两个输入并返回一个值。本质上来说，输入函数是一个可组合和可交换的二元运算符。

请看下面的示例：

```
val xs = List(2,4,6,8,10)

val sqrtSum = xs reduce {(x,y) => x + y*y}
println("sqrtSum = " + sqrtSum)
```

下面这个示例找出一个语句中最长的单词：

```
val words2 = "Scala is fun".split(" ")
val longestWord = words2.reduce{(w1,w2) => if(w1.length > w2.length) w1 else w2}
println(longestWord)
```

`reduce` 和 `reduceLeft` 都是从左边的操作数开始，而 `reduceRight` 是从右边的操作数开始：

```
(1 to 9).reduce((v1:Int, v2:Int) => v1 + v2)
(1 to 9).reduce(_ + _)

(1 to 9).reduceLeft(_ + _)

(1 to 9).reduceRight(_ + _)
```

### 6、foldLeft

方法 `foldLeft` 返回单个值。与 `reduce` 类似，但它会提供一个初始值。请看下面的示例：

```
val xs = List(2,4,6,8,10)

val sqrtSum = xs reduce {(x,y) => x + y*y}
```

```
println("sqrtSum = " + sqrtSum)           // sqrtSum = 218

val sqrtSum2 = xs.foldLeft(0){(x,y) => x + y*y}
println("sqrtSum2 = " + sqrtSum2)         // sqrtSum2 = 220
```

## 7、sortWith

使用 sortWith 函数对集合进行排序：

```
(1 to 9).sortWith((v1:Int, v2:Int) => v1 > v2).foreach(println(_))

println("=====")
(1 to 9).sortWith(_ > _).foreach(println)
```

## 1.5 函数

函数是一个可执行代码块，它接收输入参数返回一个值。它概念上与数学中的函数相似，它接收输入并返回一个输出。

Scala 是一个函数语言，它将函数当作一等公民；一个函数可以像一个变量一样被使用。函数可以作为输入参数传给另一个函数。函数可以定义为一个匿名函数字面量，就像字符串字面量；函数可以被赋给一个变量。可以在一个函数内定义函数。函数可以作为另外一个函数的输出返回值。

在 Scala 中，一切皆对象，因此函数也是对象。

### 1.5.1 函数字面量

函数字面量。函数字面量指的是在源代码中的一个未命名函数或匿名函数。在程序中可以像使用一个字符串变量一样使用它。它还可以作为一个输入参数传递给一个高阶方法或高阶函数。另外，它也可以被赋给一个变量。

字面量函数的定义是使用一个小括号，里面是输入参数，后跟一个右箭头和一个函数体。函数字面量的函数体是封闭在一个可选地花括号中的。其语法为：(参数列表):返回值 => 函数体。

下面是一个函数字面量的例子：

```
(x:Int) => {
  x + 100;
}
```

如果该函数体由单行语句组成，那么可以省略花括号。上面代码的一个简写版本是：

```
(x:Int) => x + 100;
```

最简单的函数字面量是无参函数：

```
() => println("hello") // 无参
```

一切皆对象，当然函数也是对象，所以可以将它赋给一个变量

```
val func1 = () => println("hello")
func1()           // 执行函数
```



```
val func = (i:Int) => {println("Hello"); println(i * i)}  
func(3)           // 执行函数
```

函数执行以后可以有返回值。在 Scala 中，函数体的最后一行表达式的值，就是函数返回值。

```
val func2 = (x:Int) => {  
    println("这是 func2 的函数体")  
    x*x  
}  
val result = func2(2)    // 将函数的返回值赋给变量 result  
println(result)
```

函数变量经常作为高阶函数的输入参数。

## 1.5.2 函数方法

也可以使用 `def` 关键字来定义具名函数。在 Scala 中，使用关键字 `def` 定义函数，其语法格式如下：

```
def 函数名(参数 1:数据类型,参数 2:数据类型):函数返回类型= {  
    函数体  
}
```

这种方式，通常是用在一个类中，用来定义方法用的。例如，下面定义方法 `add`：

```
// 定义方法 add  
def add(x:Int,y:Int):Int = {  
    println("这是一个对两个整数进行求和的函数")  
    x + y  
}  
  
// 通过函数名来调用函数（方法）  
add(3,5)
```

输出结果如下：

```
这是一个对两个整数进行求和的函数  
add: (x: Int, y: Int)Int  
res81: Int = 8
```

有的方法可以有返回值。在 Scala 中，函数体的最后一行表达式的值，就是返回值。下面定义一个有返回值的方法：

```
def func1(name:String, age:Int) : String = {  
    println("name=> " + name + "\t age=> " + age)  
    "欢迎" + name + "光临，您的年龄是 => " + age    // 函数的最后一行是返回值  
}  
func1("张三", 23)
```

输出结果如下：

```
name=> 张三      age=> 23  
func1: (name: String, age: Int)String  
res82: String = 欢迎张三光临，您的年龄是 => 23
```

有的方法无返回值。下面定义一个无返回值的方法：

```
def func2(): Unit = {  
    println("这个函数执行没有返回值")  
}  
func2()
```

输出结果如下：

```
这个函数执行没有返回值  
func2: ()Unit
```

有的方法比较简单，只有一行执行代码，可以写在一行上，称为"单行函数"，如下所示：

```
def printMsg(name:String) = println("Hello, " + name + ", welcome happy day!!!")  
printMsg("李四")
```

输出结果如下：

```
Hello, 李四, welcome happy day!!!
```

### 1.5.3 函数参数

函数定义时指定的参数称为"形参"，函数被调用时实际传入的参数称为"实参"。默认在函数调用时，实参和形参是按顺序一一对应的，否则有可能出现错误。请看下面的代码：

```
def func3(name:String, age:Int) : String = {  
    println("name=> " + name + "\t age=> " + age)  
    "欢迎" + name + "光临，您的年龄是 => " + age  
}  
  
func3("张三",23)           // OK  
  
// func3(23,"张三")       // 出错，类型不匹配
```

为了避免这种错误，可以在调用函数时指定参数名称（这称为名称参数），这样就不必一定要按顺序传入实参了。如下代码所示：

```
func3(age=23,name="张三")   // OK
```

也可以混用未命名参数和名称参数，只要那些未命名的参数是排在前面的即可：

```
func3("李四",age=23)
```

也可以在定义函数时指定默认参数。如果指定了默认参数，那么在调用函数时如果没有为该参数传入实参值，则默认传入的是默认参数。如下所示：

```
func3(age=23)              // 出错，参数 name 未给值
```

下面的代码在定义函数时，为形参指定了默认值，如下：

```
def func5(name:String="匿名", age:Int=18) : String = {  
    println("name=> " + name + "\t age=> " + age)  
    "欢迎" + name + "光临，您的年龄是 => " + age  
}
```

```

}

// func5(age=23)
func5()

```

Scala 在定义函数时允许指定最后一个参数可以重复（变长参数），从而允许函数调用者使用变长参数列表来调用该函数。使用可变参数的语法如下：`add(a:Int,b:Int,c:Int*)`。Scala 中使用 `"*"` 来指明该参数为变长参数。当它被定义为 `Int*` 时，则必须将所有参数作为 `Int` 传递。可以将其他参数与可变长度参数一起传递，但是可变长度参数应该是参数列表中的最后一个。函数只能接受一个可变长度参数。

请看下面的代码：

```

def echo(name:String, age:Int, args:String*) = {
    println(name)
    println(age)
    for (arg <- args) println(arg)
}

echo("张三", 23, "hello", "123", "123", "456")

```

练习：请定义一个名为 `add` 的函数，可以实现对任意多个整数的求各计算。

```

def add(values:Int*)={
    var sum =0;
    for (v <- values){
        sum = sum + v
    }

    sum
}

val sum = add(1,2,3,4,5,6,7,8,9)
println(s"所有参数的和是: $sum")

```

在函数内部，变长参数的类型，实际为一个数组，比如上例的 `String*` 类型实际上为 `Array[String]`。然而，如果试图直接传入一个数组类型的参数值给这个参数，编译器会报错：

```

val arrs = Array("hello","吃了吗")
// echo("张三",23,arrs)    // 出错

```

为了避免这种情况，可以通过在变量后面添加 `_*` 来解决，这个符号告诉 Scala 编译器在传递参数时逐个传入数组的每个元素，而不是数组整体。如下所示：

```

val arrs = Array("hello","123","123","456")
echo("张三",23, arrs:_)

```

## 1.5.4 高阶函数

在 scala 中，函数可以作为参数来传递。下面的代码定义一个接受函数作为参数的函数：

```
// 函数定义
def operation(func:(Int, Int) => Int) = {
    val result = func(4,4)
    println(result)
}

// 接下来，定义一个与预期签名匹配的函数：
val add = (x: Int, y:Int) => { x + y }

// 将 add 函数作为参数传递给 operation 函数
operation(add)
```

输出结果如下：

```
8
```

以上可以简写，将函数参数作为匿名函数传入，如下所示：

```
// 函数定义
def operation(func:(Int, Int) => Int) = {
    val result = func(4,4)
    println(result)
}

operation((x: Int, y:Int) => { x + y })
```

能接收参数为函数的函数，或者返回值为函数的函数，称为"高阶函数"，比如这里的 `operation` 就是一个高阶函数。

下面定义一个返回一个函数的高阶函数：

```
// 首先定义一个匿名函数：
// (name: String) => {
//     println("这是一个匿名函数")
//     "hello" + " " + name    // 最后一行，是函数的返回值
// }

// 现在我们将定义一个返回刚才定义的匿名函数的方法：
// def greeting() = (name: String) => {"hello" + " " + name}

def greeting() = {
    println("any any ...")
    (name: String) => {"hello" + " " + name}    // 最后一行，是函数的返回值
}

// 调用函数 greeting，得到它的返回值，这个返回值本身又是一个函数：
// 相当于 val func = (name: String) => {"hello" + " " + name}
val func = greeting()

// 调用 func
func("朋友")
```

在这里，`greeting` 函数就是一个“高阶函数”，因为它返回了一个函数。所以可以这样调用：

```
greeting()(“朋友”)
```

下面是使用高阶函数的示例：

```
// 1.匿名函数作为参数
def highOrderFunc(name:String, func:(String) => Unit): Unit = {
    func(name)
}

highOrderFunc("张三", (name:String) => println("Hello, " + name))

// 2.匿名函数作为返回值
def getGoodBayFunction(gMsg: String) = (gName: String) => println(gMsg + ", " + gName)

val goodbyeFunction = getGoodBayFunction("good bye")
goodbyeFunction("李四")
```

输出结果如下所示：

```
Hello, 张三
good bye, 李四
```

## 1.7 面向对象编程

类是一个面向对象编程概念，提供了一个高级编程抽象。从本质上来说，类是代码组织技术，将数据和所有数据的操作绑定在一起。从概念上来说，类代表一个属性和行为的实体。类是在运行时创建对象的模板，对象是类的具体实例。在源代码中定义一个类，而一个对象则存在于运行时。

面向对象编程并不新鲜，但是 **Scala** 增加了一些其他静态类型语言中没有的新特性。

### 1.7.1 类和构造函数

在 **Scala** 中类与其它面向对象语言相似。类定义由字段声明和方法定义组成。字段是一个变量，用于存储对象的状态（数据），方法包含可执行代码，它是定义在类中的一个函数，可以提供对字段的访问，并更改对象的状态。

使用关键字 **class** 定义一个类。类的定义从类名开始，后跟一对小括号，括号内由逗号分隔的类参数，然后是一对花括号，花括号内包含字段和方法。

类声明与在 **Java** 或 **c#** 中声明的方式不同：不仅声明了类，而且声明了它的主构造函数。下面的示例代码定义了一个代表汽车类的 **Car** 及其主构造函数：

```
class Car(val make: String, val model: String, val color: String)
```

在 **Scala** 中，类体是可选的。可以创建没有任何类主体的类。与 **Java** 或 **c#** 一样，**Scala** 也使用 **new** 关键字来创建类的实例。

```
val mustang = new Car("福特","野马","红色")
mustang.make
mustang.model
mustang.color
```

当构造函数的参数以 `var` 为前缀时，Scala 创建可变实例变量。`val` 和 `var` 前缀是可选的，当它们都缺失时，它们被当作私有实例值，不能从类的外面进行访问：

```
class Car(val make: String, val model: String, color: String)

val mustang = new Car("福特", "野马", "红色")
mustang.make
mustang.model
// mustang.color    // 不能访问
```

注意，当 Scala 创建实例值或变量时，它还为它们创建访问器。当使用 `val` 和 `var` 定义一个字段时，默认会创建一个 `getter` 方法；并且对于 `var`，还创建了一个额外的 `setter` 方法。

```
class Car(val make: String, val model: String, var color: String)

val mustang = new Car("福特", "野马", "红色")
mustang.color

mustang.color = "黄色"
mustang.color
```

在类中也可以定义成员方法，如下面的代码所示：

```
class Car(val make: String, val model: String, var color: String) {

    // 成员方法：重新油漆
    def repaint(newColor: String) = {
        color = newColor
    }

    // 成员方法：输出汽车的详细信息
    def printCarDetails() = println(s"制造商:$make, 车型:$model, 颜色:$color")
}
```

下面的代码使用关键字 `new` 创建类 `Car` 的实例：

```
val mustang = new Car("福特", "Mustang", "红色")
mustang.printCarDetails

val corvette = new Car("通用", "Corvette", "黑色")
corvette.printCarDetails

mustang.repaint("黄色")
mustang.printCarDetails
```

在 Scala 中，类并不声明为 `public`。Scala 源文件可以包含多个类，所有这些类都具有 `public` 可见性。类中的成员方法默认是 `public` 的。

Scala 类的构造器分为主构造函数和辅助构造函数。Scala 中把类名后面的一对 `{}` 括起来的内容称为主构造函数的函数体，默认的主构造函数为无参构造器。在 Scala 中，类的主构造函数与类定义内联编码。主构造函数是在创建对象实例时需要直接或间接地从重载构造函数调用

的构造函数。

当主构造器满足不了我们的需求之后，我们就可以创建更多的辅助构造器来配合我们的业务，辅助构造器的函数名是 `this`。在一个类中只能有一个主构造器，可以有若干重载的辅助构造器。注意，在辅助构造中的第一句，必须是调用该类的主构造器或者其他辅助构造器 - `this`(参数)。

```
class Employee(var name:String, var age:Int) {    // 主构造器

    private var married:Boolean = false

    // 辅助构造器 1
    def this() {
        this("新员工", 0)    // 在辅助构造中的第一句话，必须是调用该类的主构造器或者其他辅助构造器
        println("this is 辅助构造器")
    }

    // 辅助构造器 2
    def this(isMarried:Boolean) {
        this()    // 在辅助构造中的第一句话，调用了该类的辅助构造器 1
        married = isMarried
        println(s"this($isMarried) 是另外一个构造器")
    }

    def show() = println(s"姓名:$name, 年龄:$age, 婚否:$married")
}
```

下面的代码测试上面这个类：

```
val emp = new Employee("张三", 23)
emp.show

val emp2 = new Employee()
emp2.show

val emp3 = new Employee(true)
emp3.show
```

练习：定义一个银行账户类，具有存款和取款的方法。参考实现如下：

```
class BankAccount(var balance:Double=0) {

    // 存款方法
    def deposit(amount: Int): Unit = {
        if (amount > 0) balance = balance + amount
    }

    // 取款方法
    def withdraw(amount: Double): Double =
        if (0 < amount && amount <= balance) {
            balance = balance - amount
        }
    }
}
```

```
    balance
  } else throw new Error("insufficient funds")
}
```

因为 Scala 运行在 JVM 上，并不需要显式地删除一个对象。Java 垃圾回收器自动地删除不再使用的对象。

### 1.7.2 单例对象

在面向对象编程中一个常见的设计模式是定义一个只能被实例化一次的类。一个只能被实例化一次的类叫做“单例(singleton)”。

Scala 不提供任何静态修饰符，这与构建纯面向对象语言的设计目标有关，在这种语言中，每个值都是一个对象，每个操作都是一个方法调用，每个变量都是某个对象的成员。拥有静态并不完全符合这个目标，与此同时，在代码中使用静态也有很多缺点。相反，Scala 支持所谓的单例对象。单例对象允许将类的实例化限制为一个对象。

Scala 类不能有静态变量和方法。相反，Scala 类可以有一个单例对象或伴生对象。当只需要一个类的一个实例时，可以使用 singleton 对象。singleton 也是一个 Scala 类，但是它只有一个实例。单例对象不能被实例化(对象创建)。可以使用 object 关键字创建它。单例对象中的函数和变量可以直接调用，而无需创建对象。

Scala 提供了关键字 object 用于定义一个单例对象。下面的代码定义一个单例对象：

```
// 定义单例对象
object HelloWorld{
  def greet(){
    println("Hello World!")
    println("这是在单例对象中的输出内容。")
  }
}

// 调用
HelloWorld.greet
```

通常，main 方法是在单例对象中创建的。因此，编译器在执行时不需要创建对象来调用 main 方法。例如，下面是带有 main 方法的一个 Scala 应用程序，可以直接编译运行：

```
object HelloScala {
  def main(args: Array[String]) {
    println("这是应用程序 main 方法里输出的内容。")
  }
}
```

### 1.7.3 样例类

Scala 中提供了一种特殊的类，用 case class 进行声明，中文也可以称作“样例类”。样例



类是种特殊的类，经过优化以用于模式匹配。样例类类似于常规类，带有一个 `case` 修饰符的类，在构建不可变类时，样例类非常有用，特别是在并发性和数据传输对象的上下文中。示例如下：

```
case class Message(from:String, to:String, content:String)
```

一个样例类相当于一个 `JavaBean` 风格的域对象，带有 `getter` 和 `setter` 方法，以及构造器、`hashCode`、`equals` 和 `toString` 方法。在创建 `case class` 的对象实例时，不需要使用 `new` 关键字。

例如，下面的代码是有效的：

```
val request = Message("北京","上海","高铁");
```

下面的代码定义了一个简单的样例类：

```
// 定义一个样例类
case class Stuff(name:String, var age:Int)

val s = Stuff("张三",45)    // 不需要使用 new 来实例化

println(s)
println(s.name)
println(s.age)

s.age = s.age + 1
println(s.age)

// 也可以对两个样例类进行比较
// case 类是按结构比较的，而不是按引用比较。
println(s == Stuff("张三",45))
println(s == Stuff("张三",43))
```

输出结果如下所示：

```
Stuff(张三,45)
张三
45
46
false
false
```

另外，在一个样例类的定义中指定的所有输入参数不需要使用 `var` 或 `val` 修饰，`Scala` 自动就会使用 `val` 修饰。可以从该类的外部访问它。

此外，`Scala` 添加如下四个方法到一个样例类：`toString`、`hashCode`、`equals` 和 `copy`。这些方法使得使用一个样例类更容易。样例类对于创建不可变对象很有用。此外，它们支持模式匹配。

下面是使用 `case class` 的示例代码：

```
// 定义一个 case class
case class Money(amount:Int=1, currency:String="¥")

// 创建类的实例。注意：这里不需要用 new 关键字来实例化
val defaultAmount = Money()
val fifteenYuans = Money(15,"¥")
val fifteenYuans = Money(15)
```

```
val someYuans = Money(currency="¥")
val twentyYuans = Money(amount=20,currency="¥")

// 使用 copy 方法构造实例:
val tenEuros = twentyEuros.copy(10)
val twentyDollars = twentyEuros.copy(currency="USD")
```

```
// 带有参数时, 参数是 public val 类型的
case class Message(sender: String, recipient: String, body: String)
val message1 = Message("aa@qq.com", "bb@163.com", "hello?")

println(message1.sender) // 输出 aa@qq.com
message1.sender = "cc@sina.com" // 这一行不会编译

// 拷贝
// 可以简单地使用 copy 方法创建一个 case 类实例的浅拷贝。可以改变构造器参数(可选地)
val message2 = Message("aa@qq.com", "bb@163.com", "good good study")
val message3 = message2.copy(sender = message2.recipient, recipient = "cc@sina.com")
message3.sender           // bb@qq.com
message3.recipient        // cc@sina.com
message3.body             // "good good study"
```

当一个类被声名为 `case class` 的时候, `scala` 会帮助我们做下面几件事情:

- ❑ 构造器中的参数如果不被声明为 `var` 的话, 它默认的话是 `val` 类型的, 但一般不推荐将构造器中的参数声明为 `var`。
- ❑ 自动创建伴生对象, 同时在伴生对象里实现 `apply` 方法, 使得我们在使用的时候可以不直接显示地 `new` 对象。
- ❑ 伴生对象中同样会实现 `unapply` 方法, 从而可以将 `case class` 应用于模式匹配。
- ❑ 实现自己的 `toString`、`hashCode`、`copy`、`equals` 方法。

除此之外, `case class` 与其它普通的 `scala` 类没有区别。

`Scala` 自动为 `case class` 定义了伴生对象, 也就是 `object`, 并且定义了 `apply()` 方法负责对象创建, 该方法接收与主构造函数相同的参数, 并返回一个 `case class` 对象。

## 1.7.4 模式匹配

在 1.3.6 小节已经讲了 `Scala` 中的简单模式匹配。模式匹配是一个表达式, 因此它会生成一个值, 该值可能被分配或返回。例如:

```
44 match {
  case 44 => true    // 如果匹配了 44,则结果为 true
  case _ => false    // 否则, 结果是 false
}
```

也可以匹配字符串, 例如:

```
"过期商品" match {
```

```

    case "过期商品" => .45
    case "未过期商品" => .77
    case _ => 1.0
}

```

可用模式匹配实现 java 中 switch 语句功能：

```

val status = 500
val message = status match {
    case 200 => "ok"
    case 400 => {
        println("ERROR - 调用的服务不正确")
        "error"
    }
    case 500 => {
        println("ERROR - 服务器遇到了问题")
        "error"
    }
}

val day = "MON"
val kind = day match {
    case "MON" | "TUE" | "WED" | "THU" | "FRI" => "工作日"
    case "SAT" | "SUN" => "周末"
}

```

Scala 强大的模式匹配机制，可以应用在 switch 语句、类型检查以及“析构”等场合。

```

var sign = 0
val ch: Char = '+'

ch match {
    case '+' => sign = 1
    case '-' => sign = -1
    case _ => sign = 0
}

println("sign==> " + sign)

```

上面代码中，case \_ 模式对应于 switch 语句中的 default，能够捕获剩余的情况。如果没有模式能匹配，会抛出 MatchError。在 scala 中不需要这种 break 语句。

另外，match 是表达式，不是语句，所以是有返回值的，故可将代码简化如下：

```

val ch = '+'

sign = ch match {
    case '+' => 1
    case '-' => -1
    case _ => 0
}

println("sign====> " + sign)

```

Scala 中的任何表达式都是有返回值的，模式匹配也不例外，我们可以直接获取对应的返回

值进行操作。如果不写 `case _` 的操作，匹配不上，会抛出相关异常：`scala.MatchError`。

```
val ch = '1'
val sign = ch match {
  case '+' => 1
  case '-' => 0
  // case _ => 2
}
println(sign)
```

执行上面的代码，会抛出了几下的异常信息：

```
scala.MatchError: 1 (of class java.lang.Character)
... 40 elided
```

如果在 `case` 关键字后跟着一个变量名，那么匹配的表达式会被赋值给那个变量。`case _` 是这个特性的一个特殊情况，变量名是 `_`。

```
// 将要进行匹配的值，赋值给 case 后面的变量，我们可以对变量进行各种操作
"Hello, word!" foreach(c => println(
  c match {
    case ' ' => "space"
    case ch => "Char: " + ch
  }
))
```

输出结果如下所示：

```
Char: H
Char: e
Char: l
Char: l
Char: o
Char: ,
space
Char: w
Char: o
Char: r
Char: d
Char: l
```

## 基本模式匹配

下面的代码演示了基本模式匹配的使用：

```
def printNum(int: Int) {
  int match {
    case 0 => println("Zero")
    case 1 => println("One")
    case _ => println("more than one")
  }
}

printNum(0)
printNum(1)
```

```
printNum(2)
```

输出结果如下所示：

```
Zero
```

```
One
```

```
more than one
```

下面的代码演示了如何计算斐波那契数列（Fibonacci sequence）：

```
def fibonacci(in: Int): Int = in match {  
  case 0 | -1 | -2 => 0  
  case 1 => 1  
  case n => fibonacci(n - 1) + fibonacci(n - 2)  
}  
val result = fibonacci(10)  
println(result)
```

输出结果如下所示：

```
55
```

在 case 子句中也可以使用条件守护。例如，上面的斐波那契数列代码也可以像下面这样：

```
def fibonacci2(in: Int): Int = in match {  
  case n if n <= 0 => 0  
  case 1 => 1  
  case n => fibonacci2(n - 1) + fibonacci2(n - 2)  
}  
val result = fibonacci2(10)  
println(result)
```

输出结果如下所示：

```
55
```

## 匹配 Any 类型

下面的代码演示了如何将模式匹配应用于 Any 类型的数据：

```
val anyList= List(1, "A", 2, 2.5, 'a')  
for (m <- anyList) {  
  m match {  
    case i: Int      => println("Integer: " + i)  
    case s: String   => println("String: " + s)  
    case f: Double   => println("Double: " + f)  
    case other       => println("other: " + other)  
  }  
}
```

输出结果如下所示：

```
Integer: 1
```

```
String: A
```

```
Integer: 2
```

```
Double: 2.5
```

```
other: a
```

使用模式匹配可以代替 isInstanceOf 和 asInstanceOf 来进行使用。相比使用 isInstanceOf 来

判断类型，使用模式匹配更好。请看下面的代码：

```
def typeOps(x: Any): Int = {  
  val result = x match {  
    case i: Int           => i  
    case s: String        => Integer.parseInt(s)  
    case z: scala.math.BigInt => Int.MaxValue  
    case c: Char           => c.toInt  
    case _                => 0  
  }  
  result  
}
```

```
println(typeOps("12345") == 12345)
```

输出结果如下所示：

```
true
```

下面的代码同样演示了怎样通过模式匹配判断变量的数据类型：

```
def test2(in: Any) = in match {  
  case s: String      => "字符串，长度是" + s.length  
  case i: Int if i > 0 => "自然整数"  
  case i: Int          => "整数"  
  case a: AnyRef       => a.getClass.getName  
  case _              => "null"  
}
```

## 匹配 List 类型

也可以使用模式匹配来匹配数组、列表和元组：

```
val arr = Array(0, 1)  
  
arr match {  
  // 匹配只有一个元素的数组，且元素就是 0  
  case Array(0) => println("0")  
  
  // 匹配任何带有两个元素的数组，并将元素绑定到 x 和 y  
  case Array(x, y) => println(x + " " + y)  
  
  // 匹配任何以 0 开始的数组  
  case Array(10, _) => println("10 ...")  
  
  case _ => println("something else")  
}
```

输出结果如下所示：

```
0 1
```

下面的代码演示了如何将模式匹配应用于 List 类型的数据：

```
val rest = List(1,2,3,4)
```

```
// 模式匹配：即可以比较，也可以提取值
rest match {
  case Nil => println(0)
  case x :: rest => println(x); println(rest)
}
```

输出结果如下所示：

```
1
List(2, 3, 4)
```

下面的代码使用模式匹配对所有奇数求和：

```
def sumOdd(in: List[Int]): Int = in match {
  case Nil => 0
  case x :: rest if x % 2 == 1 => x + sumOdd(rest)
  case _ :: rest => sumOdd(rest) // 忽略第 1 个元素
}
```

```
val list1 = List(2,3,4)
val sum = sumOdd(list1)
println(sum)
```

输出结果如下所示：

```
3
```

下面的代码匹配 List 中任意数量的元素：

```
def noPairs[T](in: List[T]): List[T] = in match {
  case Nil => Nil
  case a :: b :: rest if a == b => noPairs(a :: rest) // 如果前两个元素相同，排除连续重复的元素
  case a :: rest => a :: noPairs(rest)
}
```

```
val noPairsList = noPairs(List(1,2,3,3,3,4,1,1))
println(noPairsList)
```

输出结果如下所示：

```
List(1, 2, 3, 4, 1)
```

使用类测试/强制转换机制查找 List[Any]中的所有字符串：

```
def getStrings(in: List[Any]): List[String] = in match {
  case Nil => Nil
  case (s: String) :: rest => s :: getStrings(rest)
  case _ :: rest => getStrings(rest)
}
```

```
val mixList = List(12, "张三", 23, "李四", 26, "王老五")
val strList = getStrings(mixList)
println(strList)
```

输出结果如下所示：

```
List(张三, 李四, 王老五)
```

## 模式匹配与 case class

模式匹配也可以对两个 case class 进行匹配：

```
// 定义 case class 类
case class Person(name: String, age: Int, valid: Boolean)

// 创建一个对象实例，不必使用关键字 new
val p = Person("张三", 45, true)

// 也可以
val m = new Person("李四", 24, true)

// 定义函数，找出年龄大于 35 的用户信息
// 使用模式匹配提取 case class
def older(p: Person): Option[String] = p match {
    case Person(name, age, true) if age > 35 => Some(name)
    case _                                => None
}

// 测试
val name1 = older(p).get
println(name1)

val name2 = older(p).getOrElse("匿名")
println(name2)

// older(m).get      // 会出现异常
val name3 = older(m).getOrElse("匿名")
println(name3)
```

输出结果如下所示：

```
张三
张三
匿名
```

下面这个示例代码，演示了如何进行样例类（case class）的模式匹配：

```
abstract class Expr
case class Var(name:String) extends Expr
case class UnOp(operator:String, arg:Expr) extends Expr
case class BinOp(operator:String, left:Expr, right:Expr) extends Expr

def test(expr:Expr) = expr match {
    case Var(name)                => println(s"Var($name)...")
    case UnOp(operator, e)        => println(s"$e ... $operator")
    case BinOp(operator, left, right) => println(s"$left $operator $right")
    case _                        => println("default")
}

test(BinOp("+", Var("1"), Var("2")))
```

输出结果如下所示：



```
Var(1) + Var(2)
```

下面这个示例代码，演示了如何匹配多个 case object:

```
trait Command

case object Start extends Command
case object Go extends Command
case object Stop extends Command
case object Whoa extends Command

def executeCommand(cmd: Command) = cmd match {
  case Start | Go    => println("starting")
  case Stop | Whoa   => println("stopping")
  case default       => println("You gave me: " + default) // 可访问 default 值
}

executeCommand(Start)
executeCommand(Whoa)
```

输出结果如下所示:

```
starting
stopping
```

### 模式匹配作为参数

模式匹配可作为参数传递给其他函数或方法，编译器会将模式匹配编译为函数:

```
val list = List("aa",123,"ss",456,"dd")
```

```
list.filter(a => a match {
  case s: String => true
  case _         => false
}).foreach(println)
```

// 上面的代码可简化为

```
list.filter {
  case s: String => true
  case _         => false
}.foreach(println)
```

输出结果如下所示:

```
aa
ss
dd
```

## 1.8 字符串处理

在 Scala 中，String 是一个不可变的对象，所以该对象不可被修改。这就意味着如果修改

字符串就会产生一个新的字符串对象。

## 1.8.1 字符串基本使用

Scala 中字符串的数据类型是 `java.lang.String`。在下面的代码中，获得字符串的数据类型：

```
println("Hello, world".getClass.getName)    // java.lang.String
```

实际上，Scala `String` 就是 Java `String`，所以可以使用所有常规的 Java 字符串方法。和 Java 中一样，在 Scala 中字符串对象有个 `length` 方法，可返回该字符串的长度，即该字符串包含的字符个数。如下代码所示：

```
val s = "Hello, world"
println(s.length)           // 12
```

两个字符串可以使用 `+` 号连接起来：

```
println("Hello" + " world")    // "Hello world"
```

但是因为 Scala 提供了隐式转换的魔法，`String` 实例还可以访问 `StringOps` 类的所有方法，因此可以使用它们做许多其他事情，比如将 `String` 实例视为字符序列。因此，可以使用 `foreach` 方法遍历字符串中的每个字符，Scala 将字符串当作一个字符序列。例如，我们可以遍历字符串，访问字符串中的每个字符：

```
"hello".foreach(println)    // 遍历字符串中的每个字符
```

输出结果如下所示：

```
h
e
l
l
o
```

也可以将字符串视为 `for` 循环中的字符序列。如下面的代码所示：

```
for (c <- "hello") println(c)
```

输出结果如下所示：

```
h
e
l
l
o
```

Scala 中字符串有一个 `getBytes` 方法，返回每个字符对应的 `ASCII` 值数组。代码如下所示：

```
s.getBytes.foreach(println)
```

输出结果如下所示：

```
72
101
108
108
111
44
```

```
32
119
111
114
108
100
```

因为顺序集合上有很多可用的方法，所以也可以使用其他函数方法，如 `filter`，以及其他操作方法。例如：

```
val result = "hello world".filter(_ != 'l')
println("result=" + result)           // result=heo word

"scala".drop(2).take(2).capitalize    // Al
```

可通过索引访问字符串中的一个字符。如下面的代码所示：

```
// 访问字符串中的一个字符
println("hello".charAt(0))    // 'h'
println("hello"(0))          // 'h'
println("hello"(1))          // 'e'
```

字符串也可以使用运算符，例如：

```
val greeting = "Hello, " + "World"
val matched = (greeting == "Hello, World")
val theme = "Na " * 16 + "Batman!"
```

输出结果如下所示：

```
greeting: String = Hello, World
matched: Boolean = true
theme: String = Na Na Na Na Na Na Na Na Na Na Na Na Na Na Na Na Batman!
```

## 1.8.2 判断字符串相等性

在 Scala 中，字符串是按照内容来进行比较的。下面的代码演示了如下判断字符串的相等性：

```
val s1 = "Hello"
val s2 = "Hello"
val s3 = "H" + "ello"
println("s1 == s2: " + (s1 == s2))    // true
println("s1 == s3: " + (s1 == s3))    // true

println("-----")
val s4: String = null
println("s3 == s4: " + (s3 == s4))    // false
println("s4 == s3: " + (s4 == s3))    // false

println("-----")
val s11: String = null
val s22: String = null
```

```
println("s11 == s22: " + (s11 == s22)) // true

println("-----")
val a = "Marisa"
val b = "marisa"
println("a == b: " + (a == b))          // false
println("a.equalsIgnoreCase(b): " + (a.equalsIgnoreCase(b))) // true
```

在 Scala 中，使用 `==` 方法测试对象是否相等。这与 Java 不同，在 Java 中使用 `equals` 方法比较两个对象。在 Scala 中，`AnyRef` 类中定义的 `==` 方法首先检查 `null` 值，然后调用第一个对象(即 `this`)上的 `equals` 方法看这两个对象是否相等。因此，在比较字符串时不必检查 `null` 值。

### 1.8.3 字符串插值

Scala 中的 `String` 构建在 Java 中的 `String` 之上，并添加了额外的特性，如字符串插值（字符串插值是一种将字符串内的值与变量相结合的机制）。字符串插值是根据数据创建字符串的过程。用户可以将任何变量的引用直接嵌入处理后的字符串文本并格式化字符串。

Scala 中的插值符号是在字符串的第一个双引号之前添加的 `s` 前缀。然后，美元符号操作符 `$` 可以用来引用变量。例如，下面的代码就应用了字符串插值：

```
val name = "张三"
s"我的朋友叫${name}"
```

输出结果如下所示：

```
scala> val name = "张三"
name: String = 张三

scala> s"我的朋友叫${name}"
res12: String = 我的朋友叫张三
```

Scala 提供了如下几种可用的字符串插值方法：

- ☐ `s` 插值器。
- ☐ `f` 插值器。
- ☐ `raw` 插值器。

#### 字符串 `s` 插值器

使用插值器 `s`，字符串字面量允许用户使用引用变量直接追加数据。下面的代码演示了 `s` 插值器和及显示结果：

```
var bookName = "Spark 大数据处理技术"
println(s"最畅销的 Spark 图书是$bookName")
```

执行结果如下所示：

```
scala> var bookName = "Spark大数据处理技术"
bookName: String = Spark大数据处理技术

scala> println(s"最畅销的Spark图书是$bookName")
最畅销的Spark图书是Spark大数据处理技术
```

可以同时引入多个变量，如下面的代码所示（当引用变量与其它文字字面量之间没有空格时，需要用花括号将引用变量括起来）：

```
val bookName = "Spark 大数据处理技术"
val pages = 330           // 页码
val price = 65.00         // 单价
println(s"${bookName}这本书共有${pages}页，零售价格为${price}。")
```

输出结果如下所示：

```
scala> val bookName = "Spark大数据处理技术"
bookName: String = Spark大数据处理技术

scala> val pages = 330
pages: Int = 330

scala> val price = 65.00
price: Double = 65.0

scala> println(s"${bookName}这本书共有${pages}页，零售价格为${price}。")
Spark大数据处理技术这本书共有330页，零售价格为65.0。
```

此外，可以使用字符串插值器计算任意表达式，如下面的代码所示：

```
val bookName = "Spark 大数据处理技术"
val quantity = 5           // 购买数量
val price = 65.00         // 单价
println(s"${bookName}这本书单价为${price}，一共买了${quantity}本，则总金额是：${price * quantity}")
```

输出结果如下所示：

```
scala> val bookName = "Spark大数据处理技术"
bookName: String = Spark大数据处理技术

scala> val quantity = 5
quantity: Int = 5

scala> val price = 65.00
price: Double = 65.0

scala> println(s"${bookName}这本书单价为${price}，一共买了${quantity}本，则总金额是：${price * quantity}")
Spark大数据处理技术这本书单价为65.0，一共买了5本，则总金额是：325.0
```

字符串 s 插值器还可以使用表达式。下面的代码演示了这种用法：

```
val age = 33
println(s"过一年，长一岁：${age + 1}")
println(s"你今年是 33 岁吗？ ${age == 33}")
println(s"仰天大笑出门去，${"哈"*3}")
```

输出结果如下所示：

```
scala> val age = 33
age: Int = 33

scala> println(s"过一年，长一岁: ${age + 1}")
过一年，长一岁: 34

scala> println(s"你今年是33岁吗? ${age == 33}")
你今年是33岁吗? true

scala> println(s"仰天大笑出门去，${"哈"*3}")
仰天大笑出门去， 哈哈
```

字符串 s 插值器还可以引用对象字段。下面的代码演示了这种用法：

```
case class Student(name: String, score: Int)
val zhangsan = Student("张三", 95)
println(s"${zhangsan.name}的成绩是${zhangsan.score}")
```

输出结果如下所示：

```
scala> case class Student(name: String, score: Int)
defined class Student

scala> val zhangsan = Student("张三", 95)
zhangsan: Student = Student(张三,95)

scala> println(s"${zhangsan.name}的成绩是${zhangsan.score}")
张三的成绩是95
```

### 字符串 f 插值器

Scala 提供了一种从数据创建字符串的新机制。对字符串字面量使用 f 插值器允许用户创建格式化的字符串，并在处理后的字符串文字中直接嵌入变量引用。下面的代码演示了 f 插值器的使用：

```
var bookPrice = 65.00
val bookName = "Spark 大数据处理技术"
println(f"${bookName}这本书的价格是${bookPrice}元")
println(f"${bookName}这本书的价格是${bookPrice}%1.1f 元")
println(f"${bookName}这本书的价格是${bookPrice}%1.2f 元")
```

输出结果如下所示：

```
scala> var bookPrice = 65.00
bookPrice: Double = 65.0

scala> val bookName = "Spark大数据处理技术"
bookName: String = Spark大数据处理技术

scala> println(f"${bookName}这本书的价格是${bookPrice}元")
Spark大数据处理技术这本书的价格是65.0元

scala> println(f"${bookName}这本书的价格是${bookPrice}%1.1f元")
Spark大数据处理技术这本书的价格是65.0元

scala> println(f"${bookName}这本书的价格是${bookPrice}%1.2f元")
Spark大数据处理技术这本书的价格是65.00元
```

%之后允许的格式基于 Java 提供的格式字符串。

另外，字符串 f 插值还可以这样用：

```
val item = "apple"
println(f"我今天开发了一款新的${item}%.3s")
println(f"PI 值是: ${355/113.0}%.2f")
```

输出结果如下所示：

```
scala> val item = "apple"
item: String = apple

scala> println(f"我今天开发了一款新的${item}%.3s")
我今天开发了一款新的app

scala> println(f"PI值是: ${355/113.0}%.2f")
PI值是: 3.14
```

也可以先将带有字符串插值的字符串保存到一个变量中，然后再输出。代码如下所示：

```
val content = f"${bookName}这本书的价格是${bookPrice}%.2f 元"
println(content)
```

输出结果如下所示：

```
scala> val content = f"${bookName}这本书的价格是${bookPrice}%.2f 元"
content: String = Spark大数据处理技术这本书的价格是65.00元

scala> println(content)
Spark大数据处理技术这本书的价格是65.00元
```

### 字符串 raw 插值器

raw 插值器不允许转义字符。也就是说，当使用 raw interpolator 时，在字符串内不执行字面量的转义。例如，使用 \n 与 raw 插值器不返回换行字符。下面的代码演示了 raw 插值器的使用：

```
val bookId = 101
val bookName = "Spark 大数据处理技术"
println(s"图书编号是$bookId。 \n 图书名称是: $bookName")
println(raw"图书编号是$bookId。 \n 图书名称是: $bookName")
```

输出结果如下所示：

```
scala> val bookId = 101
bookId: Int = 101

scala> val bookName = "Spark大数据处理技术"
bookName: String = Spark大数据处理技术

scala> println(s"图书编号是$bookId。 \n图书名称是: $bookName")
图书编号是101。
图书名称是: Spark大数据处理技术

scala> println(raw"图书编号是$bookId。 \n图书名称是: $bookName")
图书编号是101。 \n图书名称是: Spark大数据处理技术
```

## 1.8.4 处理字符串，一次一个字符

可以有多个方法一次一个字符地处理字符串。如下所示：

```
// 迭代字符串中的每个字符
val upper = "hello, world".map(c => c.toUpper)
println(upper)
val upper2 = "hello, world".map(_.toUpper)
println(upper2)
val upper3 = "hello, world".filter(_ != '!').map(_.toUpper)
println(upper3)

// for/yield 循环生成新的字符串，等价于前两个 map
val upper4 = for (c <- "hello, world") yield c.toUpper
println(upper4)

// for/yield 加上过滤
val result4 = for {
  c <- "hello, world"
  if c != '!'
} yield c.toUpper
println(result4)
```

## 1.8.5 使用正则表达式

字符串中正则表达式模式匹配

通过在 `String` 上调用 `.r` 方法来创建一个 `scala.util.matching.Regex` 对象，然后在 `findFirstIn` 中使用该模式来查找一个匹配，在 `findAllIn` 中使用该模式来查找所有的匹配。

```
// 使用正则表达式匹配
val numPattern = "[0-9]+".r
val address = "123 Main Street Suite 101"

val match1 = numPattern.findFirstIn(address) // 返回的是一个 Option[String]
println(match1.get) // 123

val matches = numPattern.findAllIn(address) // 返回的是一个 Option[String]
matches.foreach(println) // 123 101

// 将匹配结果转为 Array 数组: toArray, toList, toSeq, toVector
val matches2 = numPattern.findAllIn(address).toArray // Array(123, 101)

// 另一种方法
import scala.util.matching.Regex

val numPattern3 = new Regex("[0-9]+")
val address3 = "123 Main Street Suite 101"
```



```

val match3 = numPattern.findFirstIn(address3)
println(match3)           // Some(123)
println(match3.get)       // 123

// 处理返回的值: Option/Some/None
val result5 = numPattern3.findFirstIn(address3).getOrElse("no match")
println(result5)          // 123

// 因为一个 Option 是零个或一个元素的集合
numPattern3.findFirstIn(address3).foreach { e => println(e)}    // 123
numPattern3.findFirstIn(address3).foreach(println)              // 123

// 还可以这样处理: 使用模式匹配
match1 match {
  case Some(s) => println(s"发现: $s")           // 123
  case None => println("什么也没找到")
}

```

提取与正则表达式相匹配的部分(一个或多个)

使用 `group` 可以进行字符串提取。

```

// 提取与正则表达式相匹配的部分(一个或多个)
val pattern9 = "[0-9]+ ([A-Za-z]+)".r

// 下面的代码从给定的字符串提取数值字段和字母字段作为两个变量: count 和 fruit
val pattern9(count, fruit) = "100 Bananas"
println(count,fruit)

val datePattern = """"(\d\d\d\d)-(\d\d)-(\d\d)"""".r
val dates = "历史上重要的时刻: 2004-01-20, 1958-09-05, 2010-10-06, 2011-07-15"
val firstDate = datePattern.findFirstIn(dates).getOrElse("No date found.")
val firstYear = for (m <- datePattern.findFirstMatchIn(dates)) yield m.group 1

```

## 1.8.6 字符串拆分

可以使用 `split` 方法对字符串按指定的分割符进行分割。请看下面的示例代码:

```

"hello world".split(" ")
"hello world".split(" ").foreach(println)

val s33 = "鸡蛋, 牛奶, 面包, 可口 可乐"
s33.split(",")      // 返回一个数组
s33.split(",").map(_.trim).foreach(println)

// 基于正则表达式拆分(按逗号或空格分割)
"hello world,this is AI".split(",\\s+").foreach(println)

```

# 1.8.7 字符串替换

在字符串上调用 `replaceAll` 方法执行一个替换，返回一个新的字符串。

```
// 字符串替换
val address6 = "123 Main Street".replaceAll("[0-9]", "x")
println(address6) // xxx Main Street

// 也可以先定义一个正则表达式，然后执行 replaceAllIn 方法
val regex = "[0-9]".r
val newAddress = regex.replaceAllIn("123 Main Street", "x")
println(newAddress) // xxx Main Street

// 只替换第一个匹配到的
val result7 = "123".replaceFirst("[0-9]", "x")
println(result7) // x23

// 还可以在 Regex 上调用 replaceFirstIn
val regex8 = "H".r
val result8 = regex8.replaceFirstIn("Hello Hi world", "J")
println(result8) // Jello Hi world
```

# 1.8.8 转义字符

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位
<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(FP)，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 \

# 1.8.9 多行字符串

在 Scala 中，可以通过三个双引号将文本包围起来创建多行字符串：

```
val text = """This is
a multiline
String"""
```

如下所示：

```
scala> val text = """This is
| a multiline
| String"""
text: String =
This is
a multiline
String

scala> println(text)
This is
a multiline
String
```

## 1.9 数值处理

在 Scala 中，所有的数值类型都是对象，包括 Byte、Char、Double、Float、Int、Long 和 Short。这七个数值类型继承自 AnyVal trait。它们的取值范围与 Java 一样：

数据类型	描述	大小	最小值	最大值
Char	无符号Unicode字符	2个字节	0	65536
Byte	有符号整数	1个字节	-128	127
Short	有符号整数	2个字节	-32768	32767
Int	有符号整数	4个字节	-2147483648	2147483647
Long	有符号整数	8个字节	-9223372036854775808	9223372036854775807
Float	有符号单精度浮点数	4个字节	n/a	n/a
Double	有符号单精度浮点数	8个字节	n/a	n/a

除了以上这些类型，还有代表真/假的 Boolean 类型。Boolean 类型可以有 true 或 false 值。Unit 和 Boolean 类被认为是“非数值类型”。

如果想要知道准确的值的范围，可以象下面这样：

```
// 获取值的范围
println(Byte.MinValue)           // -128
println(Byte.MaxValue)           // 127

println(Short.MinValue)          // -32768
println(Short.MaxValue)          // 32767

println(Int.MinValue)             // -2147483648
println(Int.MaxValue)             // 2147483647

println(Long.MinValue)           // -9223372036854775808
println(Long.MaxValue)           // 9223372036854775807

println(Float.MinValue)           // -3.4028235E38
println(Float.MaxValue)           // 3.4028235E38

println(Double.MinValue)           // -1.7976931348623157E308
println(Double.MaxValue)           // 1.7976931348623157E308
```

```
println(Char.MinValue.toInt)    // 0
println(Char.MaxValue.toInt)    // 65535
```

还可以用标准数值类型的 `PositiveInfinity` 和 `NegativeInfinity` 来表示正无穷和负无穷:

```
Double.PositiveInfinity        // Double = Infinity
Double.NegativeInfinity        // Double = -Infinity
1.7976931348623157E308 > Double.PositiveInfinity    // Boolean = false
Double.MaxValue > Double.PositiveInfinity          // Boolean = false
```

### 1.9.1 字符串转数值(解析)

Scala 提供了将字符串解析为数值的方法。请看下面的代码:

```
// 字符串转数值(解析)
println("100".toInt)          // 100
println("100".toDouble)       // 100.0
println("100".toFloat)        // 100.0
println("1".toLong)           // 1
println("1".toShort)          // 1
println("1".toByte)           // 1
```

需要注意的是, 如果字符串无法解析为数据的话, 那么这些方法可能会抛出通常的 `Java NumberFormatException` 异常。

`BigInt` 和 `BigDecimal` 实例也可以直接从字符串创建(也可能抛出 `NumberFormatException`):

```
val bi = BigInt("1")
val bd = BigDecimal("3.14159")
println(bi)                // 1
println(bd)                // 3.14159
```

### 1.9.2 数值之间类型的转换

不同类型的数值之间也可以相互转换:

```
// 数值之间类型的转换
println(19.45.toInt)        // 19
println(19.toFloat)         // 19.0
println(19.toDouble)        // 19.0
println(19.toLong)          // 19

val a = 3
val b = a.toFloat
println(b)                  // 3.0

// 判断是否可转换
val a2 = 1000L
println(a2.isValidByte)     // false
println(a2.isValidShort)    // true
```

## 1.10 异常处理

Scala 的异常处理和其它语言比如 Java 类似，一个方法可以通过抛出异常的方法而不返回值的方式终止相关代码的运行。调用函数可以捕获这个异常作出相应的处理或者直接退出，在这种情况下，异常会传递给调用函数的调用者，依次向上传递，直到有方法处理这个异常。

Scala 中的所有异常都是未检查的；没有检查异常的概念。Scala 在选择是否捕获异常方面提供了很大的灵活性。

抛出异常在 Scala 和 Java 中是一样的。

```
import scala.io.Source
import java.io.FileNotFoundException

try {
  val line = Source.fromFile("./wc.txt").mkString
  val ret = 1 / 0
  println(line)
} catch {
  case fNFE:FileNotFoundException => {
    println("FileNotFoundException: 文件找不到了，传的路径有误。。。")
  }
  case e:Exception => {
    println("Exception: " + e.getMessage)
  }
  case _ : Throwable => {
    println("default 处理方式")
  }
} finally {
  println("this is 必须要执行的语句")
}
```

Scala 中的 try/catch 结构与 Java 不同，因为 Scala 中的 try/catch 是一个表达式，它会产生一个值，而且 Scala 中的异常可以在 catch 块中进行模式匹配，而不是为每个不同的异常提供单独的 catch 子句。因为 Scala 中的 try/catch 是一个表达式，所以如果调用失败，可以将调用包装在 try/catch 中，并赋值一个默认值。

下面的代码在 try/catch 中包装调用并在调用失败时分配默认值：

```
try{Integer.parseInt("dog")} catch{case _ => 0}
```

## 1.11 文件读写和处理

### 1.11.1 基本的输入输出

可以用 `scala.io.StdIn.readLine()` 函数从控制台读取一行输入。如果要读取数字、Boolean 或者是字符，可以用 `readInt`、`readDouble`、`readByte`、`readShort`、`readLong`、`readFloat`、`readBoolean`

或者 readChar。

```
val input = readLine("请输入内容: ")
print("您输入的是: ")
println(input)
printf("您的输入内容是%s",input)
```

执行过程及结果如下：

```
scala> val input = readLine("请输入内容: ")
warning: there was one deprecation warning; re-run with -deprecation for details
请输入内容: input: String = hello world

scala> print("您输入的是:")
您输入的是:

scala> println(input)
hello world

scala> printf("您的输入内容是%s",input)
您的输入内容是hello world
scala> 
```

### 1.11.2 打开和读取文本文件

有两种方式打开和读取文本文件。

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

```
val filename = "fileopen.scala"
for (line <- Source.fromFile(filename).getLines) {
    println(line)
}

// 变化的方式
val lines = Source.fromFile("/Users/Al/.bash_profile").getLines.toList
val lines = Source.fromFile("/Users/Al/.bash_profile").getLines.toArray

val fileContents = Source.fromFile(filename).getLines.mkString
```

关闭文件：

```
val bufferedSource = Source.fromFile("example.txt")
for (line <- bufferedSource.getLines) {
    println(line.toUpperCase)
}
bufferedSource.close
```

处理异常：

```
import scala.io.Source
import java.io.{FileNotFoundException, IOException}

val filename = "no-such-file.scala"
try {
```

```

    for (line <- Source.fromFile(filename).getLines) {
        println(line)
    }
} catch {
    case e: FileNotFoundException => println("Couldn't find that file.")
    case e: IOException => println("Got an IOException!")
}

```

### 1.11.3 写入文本文件

Scala 并没有提供任何特殊的文件写入能力，因此使用 Java 的 `PrintWriter` 或 `FileWriter`:

```

// PrintWriter
import java.io._

val pw = new PrintWriter(new File("hello.txt" ))
pw.write("Hello, world")
pw.close

// FileWriter
val file = new File(canonicalFilename)
val bw = new BufferedWriter(new FileWriter(file))
bw.write(text)
bw.close()

```

### 1.11.4 读写二进制文件

Scala 并没有提供任何特殊的方式来读写二进制文件，所以使用 Java `FileInputStream` 和 `FileOutputStream` 类:

```

import java.io._

object CopyBytes extends App {
    var in = None: Option[FileInputStream]
    var out = None: Option[FileOutputStream]
    try {
        in = Some(new FileInputStream("/tmp/Test.class"))
        out = Some(new FileOutputStream("/tmp/Test.class.copy"))
        var c = 0
        while ({c = in.get.read; c != -1}) {
            out.get.write(c)
        }
    } catch {
        case e: IOException => e.printStackTrace
    } finally {
        println("entered finally ...")
        if (in.isDefined) in.get.close
    }
}

```

```

        if (out.isDefined) out.get.close
    }
}

```

### 1.11.5 处理 CSV 文件

例如：

```

January, 10000.00, 9000.00, 1000.00
February, 11000.00, 9500.00, 1500.00
March, 12000.00, 10000.00, 2000.00

```

按行处理：

```

object CSVDemo extends App {
    println("Month, Income, Expenses, Profit")
    val bufferedSource = io.Source.fromFile("/tmp/finance.csv")
    for (line <- bufferedSource.getLines) {
        val cols = line.split(",").map(_.trim)
        // do whatever
        println(s"${cols(0)}|${cols(1)}|${cols(2)}|${cols(3)}")
    }
    bufferedSource.close
}

```

如果不想使用数组访问，可以使用如下的 for 循环：

```

for (line <- bufferedSource.getLines) {
    val Array(month, revenue, expenses, profit) = line.split(",").map(_.trim)
    println(s"$month $revenue $expenses $profit")
}

```

如果有标题行，可以这样处理：

```

for (line <- bufferedSource.getLines.drop(1)) {
    // ...
}

```

如果愿意，还可以写成 foreach 循环：

```

bufferedSource.getLines.foreach { line =>
    rows(count) = line.split(",").map(_.trim)
    count += 1
}

```