Scala编程（讲师：霄嵩）

```
课程目标：
熟练使用Scala进行Spark开发
为阅读Spark内核源码做准备

课程内容：
    第一部分 Scala基础
    第二部分 控制结构和函数
    第三部分 数组和元组
    第四部分 类与对象
    第五部分 继承
    第六部分 特质
    第七部分 模式匹配和样例类
    第八部分 函数及抽象化
    第九部分 集合
    第十部分 隐式机制
```

第一部分 Scala基础

第1节 Scala语言概况

1.1 Scala语言起源

马丁·奥德斯基（Martin Odersky）是编译器及编程的狂热爱好者。

主流JVM的Javac编译器就是马丁·奥德斯基编写出来的，JDK5.0、JDK8.0的编译器就是他写的。

长时间的编程之后，他希望发明一种语言，能够让写程序这样的基础工作变得高效，简单。

当接触到Java语言后，对Java这门语言产生了极大的兴趣，所以决定将函数式编程语言的特点融合到Java中，由此发明了Scala。

1.2 Scala语言特点

Scala是一门以 JVM 为运行环境并将面向对象和函数式编程的最佳特性结合在一起的静态类型编程语言。

Scala源代码会被编译成Java字节码，然后运行于JVM之上，并可以调用现有的Java类库，实现两种语言的无缝互操作的。

- **面向对象**

Scala是一种面向对象的语言。

Scala中的每个值都是一个对象，包括基本数据类型（即布尔值、数字等）在内，连函数也是对象。

- **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中支持高阶函数，允许嵌套多层函数，并支持柯里化。

Scala提供了模式匹配，可以匹配各种情况，比如变量的类型、集合的元素、有值或无值。

- **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。

- **并发性**

Scala使用Actor作为其并发模型，Actor是类似线程的实体。

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。

1.3 为什么要学Scala

优雅：这是框架设计师第一个要考虑的问题，框架的用户是应用开发程序员，API是否优雅直接影响用户体验。

简洁：Scala语言表达能力强，一行代码抵得上Java多行，开发速度快。

融合大数据生态圈：Hadoop现在是大数据事实标准，(Kafka Spark源码都是用Scala编写的，Spark Flink都支持使用Scala进行开发)Spark并不是要取代Hadoop，而是要完善Hadoop生态。

第2节 环境准备

Scala官网：https://www.scala-lang.org/

```
1、下载Scala
2、Windows下安装Scala
3、配置IDEA开发环境
4、REPL
```

2.1 Windows下环境配置

访问Scala官网下载Scala 2.11.8安装包，下载地址：https://www.scala-lang.org/download/2.11.8.html

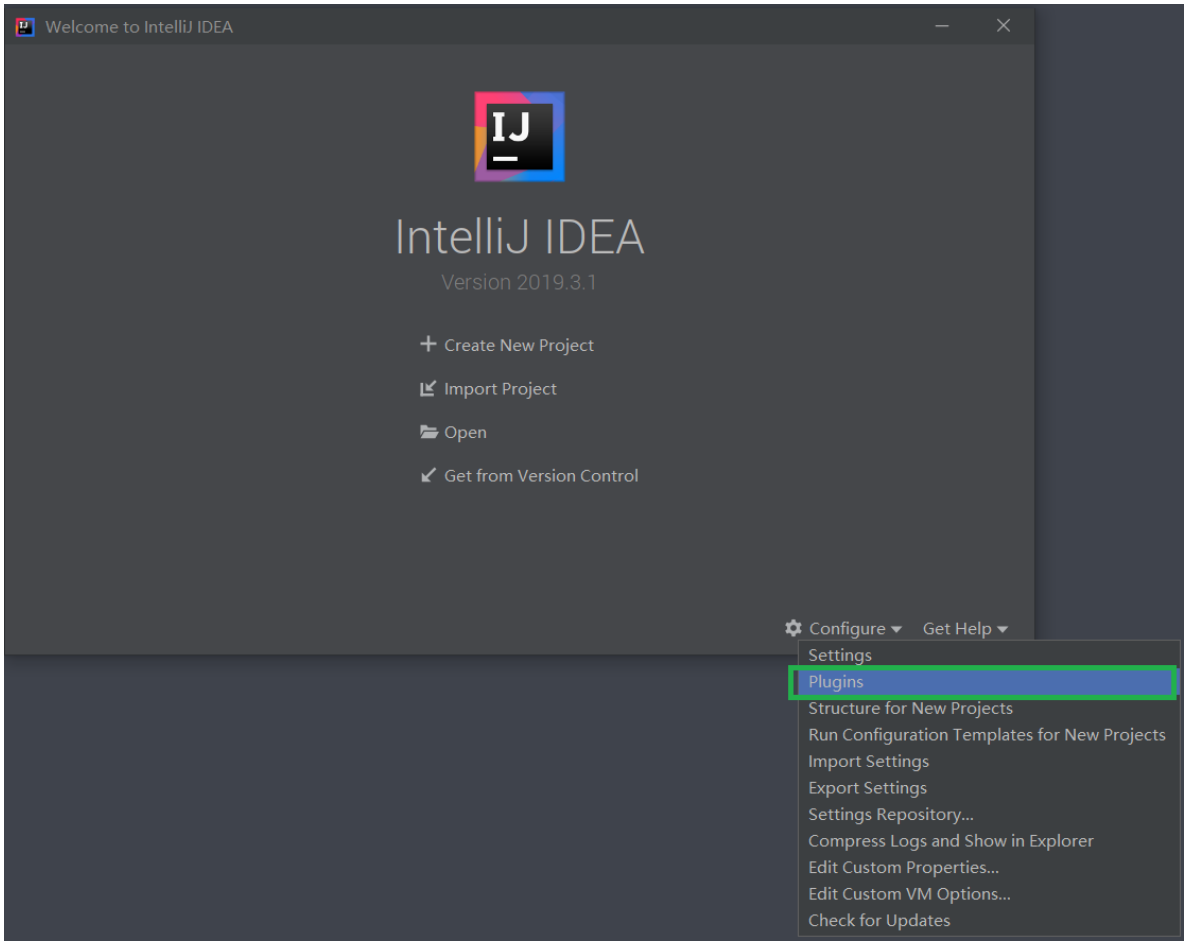
下载scala-2.11.8.msi后，点击下一步就可以了（自动配置上环境变量）。

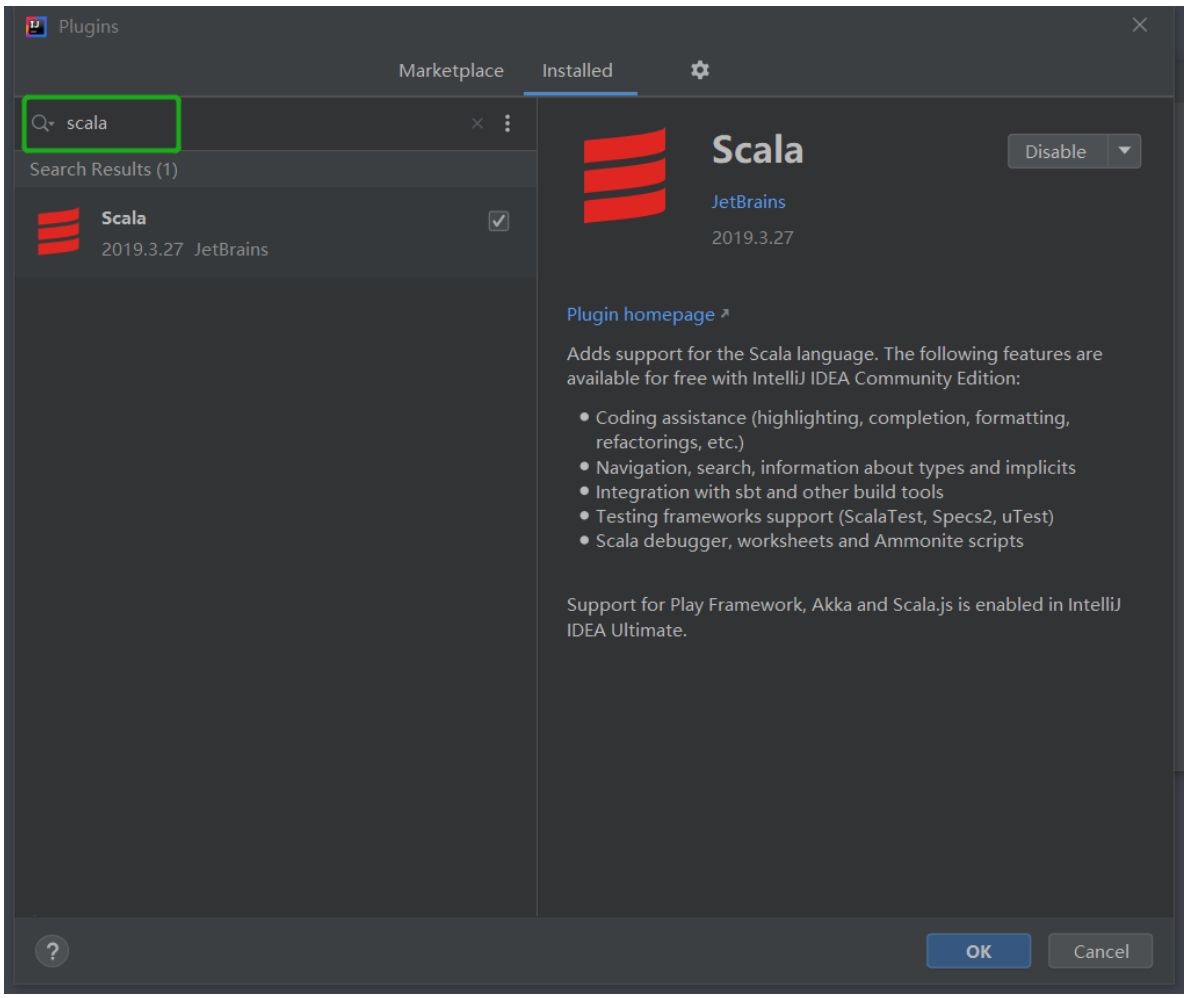
也可以下载 scala-2.11.8.zip，解压后配置上环境变量就可以了。

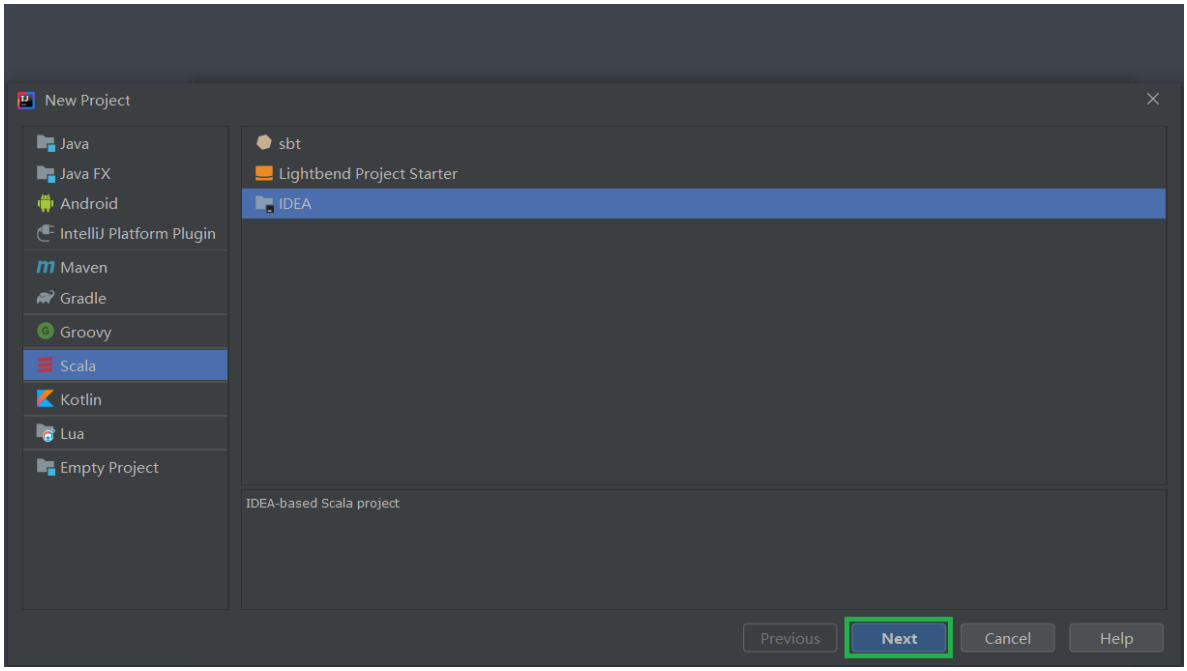
备注：安装Scala之前，Windows系统需要安装JDK。

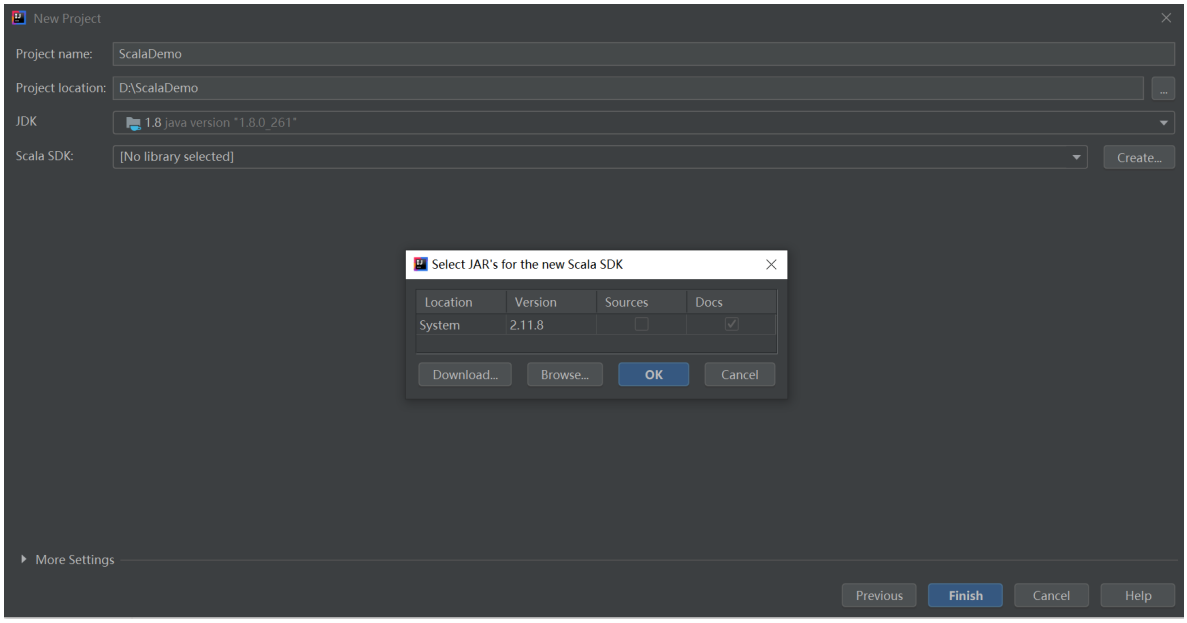
2.2 IDEA环境配置

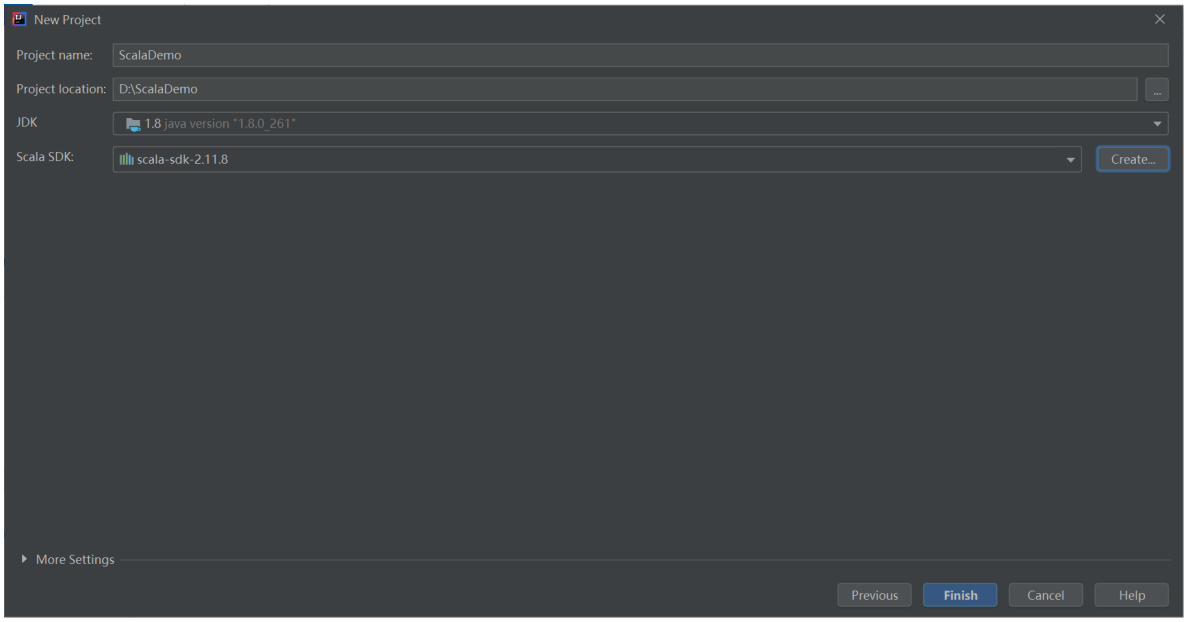
IDEA是 Java 的集成开发环境，要支持Scala开发，需要安装Scala插件；

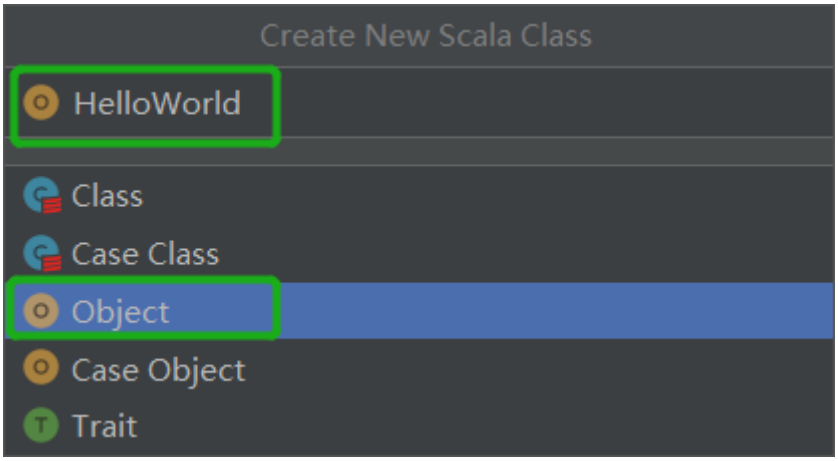



Plugins
Marketplace
Installed
scala
Search Results (1)
Scala
2019.3.27 JetBrains
Scala
JetBrains
2019.3.27
Disable
Plugin homepage
Adds support for the Scala language. The following features are available for free with IntelliJ IDEA Community Edition:
Coding assistance (highlighting, completion, formatting, refactorings, etc.)
Navigation, search, information about types and implicits
Integration with sbt and other build tools
Testing frameworks support (ScalaTest, Specs2, uTest)
Scala debugger, worksheets and Ammonite scripts
Support for Play Framework, Akka and Scala.js is enabled in IntelliJ IDEA Ultimate.
OK
Cancel


New Project
Java
Java FX
Android
IntelliJ Platform Plugin
Maven
Gradle
Groovy
Scala
Kotlin
Lua
Empty Project
sbt
Lightbend Project Starter
IDEA
IDEA-based Scala project
Previous
Next
Cancel
Help








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

2.3 Scala的REPL

在命令行输入Scala可启动Scala REPL。

REPL 是一个交互式解析器环境，R(read)、E(evaluate) 、P（print）、L（loop）

输入值，交互式解析器会读取输入内容并对它求值，再打印结果，并重复此过程。

第3节 基础语法

基础语法规则：

- **区分大小写** - Scala语言对大小写敏感；
- **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名，每个单词的第一个字母要大写；比如:ListDemo
- **方法名** - 所有方法名的第一个字母用小写。如果需要使用几个单词来构成方法名，除第一个单词外每个词的第一个字母应大写；比如：getResult
- **程序文件名** - Scala程序文件的后缀名是 .scala，程序文件的名称可以不与对象名称完全匹配。这点与Java有所区别。

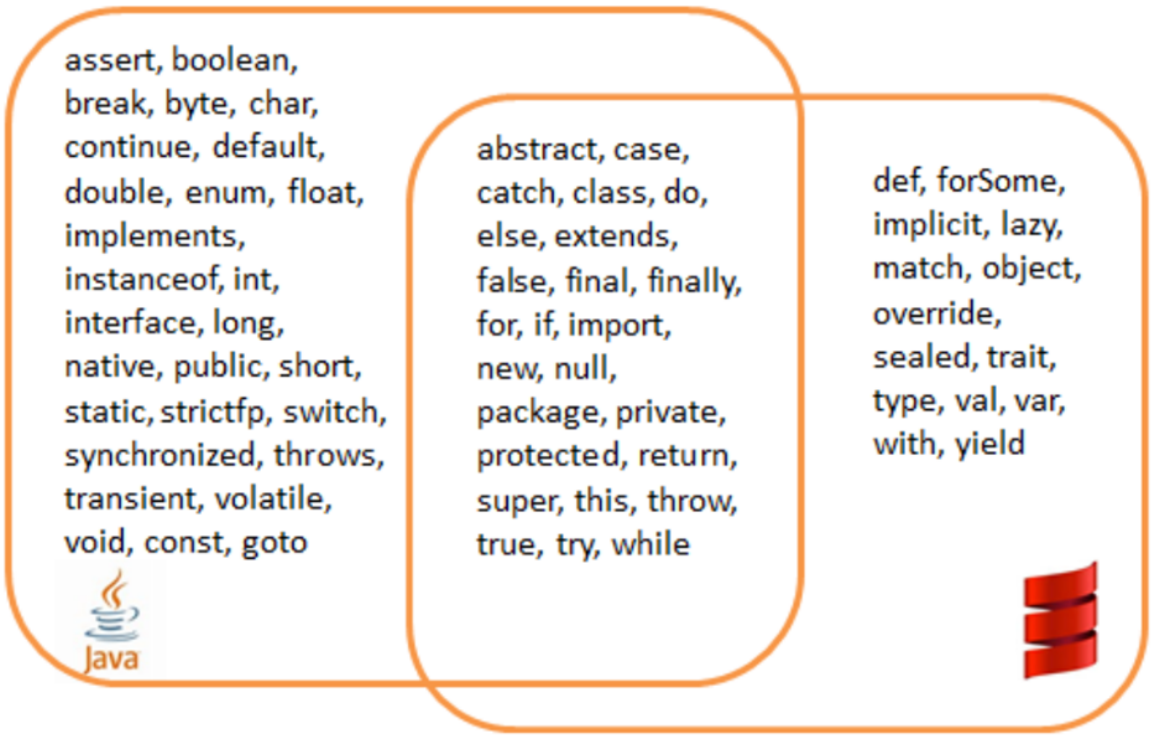
  备注：建议遵循 Java 的惯例，程序文件名称与对象名称匹配；
- **main()方法** - Scala程序从main()方法开始处理，这是每一个Scala程序的入口点。main()定义在object中；

标识符。所有Scala组件都需要名称，用于对象、类、变量和方法的名称称为标识符。

关键字不能用作标识符，标识符区分大小写；

标识符以字母或下划线开头，后面可以有更多的字母、数字或下划线；

$字符是Scala中的保留关键字，不能在标识符中使用；



注释。Scala使用了与Java相同的单行和多行注释；

换行符。Scala语句可以用分号作为一行的结束，语句末尾的分号通常可以省略，但是如果一行里有多个语句那么分号是必须的。

```
小结：
Scala的基础语法与Java比较类似，但是仍然有三点不一样的地方：
1、在Scala中换行符是可以省略的
2、Scala中main方法定义在object中
3、Scala中程序文件名可以不与对象名称相匹配，但是建议仍然遵循Java的规范，二者最好匹配
```

第4节 常用类型与字面量

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，相当于Java中的void，用于不返回任何结果的方法的返回类型。Unit写成()
Null	通常写成null
Nothing	Nothing类型在Scala类层级的最低端，它是任何其他类型的子类型
Any	Any是Scala中所有类的超类
AnyRef	AnyRef是Scala中所有引用类的超类

Scala和Java一样，有8种数值类型 Byte、Short、Int、Long、Float、Double、Char、Boolean 类型；

和 Java 不同的是 ，这些类型都是类，有自己的属性和方法。

Scala并不刻意的区分基本类型和引用类型。

String 直接引用 Java.lang.String 中的类型，String在需要时能隐式转换为StringOps，因此不需要任何额外的转换，String就可以使用StringOps中的方法。

每一种数据类型都有对应的Rich类型，如RichInt、RichChar等，为基本类型提供了更多的有用操作。

```
-- StringOps。 //toInt等方法都定义在StringLike中；StringOps实现了StringLike
"11".toInt

1.max(10)
1.min(10)
1.to(10)
1.until(10)
```

整数字面量。整数字面量有两种形式，十进制与十六进制(0X/0x开头)

```
-- 十六进制整数字面量
scala> val a = 0xa
a: Int = 10

scala> val a = 0X00FF
a: Int = 255

scala> val magic = 0xcafe
magic: Int = 51966

-- 十进制整数字面量
scala> val dec1 = 255
dec1: Int = 255

scala> val dec1 = 31
dec1: Int = 31

-- Long类型整数字面量
scala> val magic = 0xcafeL
magic: Long = 51966

scala> val long1 = 255L
long1: Long = 255

-- Short 或 Byte 类型，需要明确声明，否则编译器会推断为Int类型
scala> val little: Short = 32767
little: Short = 32767

scala> val littler: Byte = 127
littler: Byte = 127
```

浮点数字面量

```
-- 十进制数、可选的小数点、可选的e开头的指数
scala> val big = 3.1415926
big: Double = 3.1415926

scala> val bigger = 3.1415926e1
bigger: Double = 31.415926

-- 浮点数字面量以F/f结尾为Float类型；否则为Double类型；
scala> val litte = 0.31415926f
litte: Float = 0.31415927

scala> val litte = 0.31415926e1F
litte: Float = 3.1415925
```

字符字面量

```
scala> val a = 'A'
a: Char = A

-- 用字符的Unicode码来表示。Unicode码前128个字符就是ASCII码
scala> val b = '\u0042'
b: Char = B

-- 转义字符
scala> val mark = '\'
<console>:1: error: unclosed character literal
val mark = '\'
           ^

scala> val mark = '\\'
mark: Char = \
```

字符串字面量

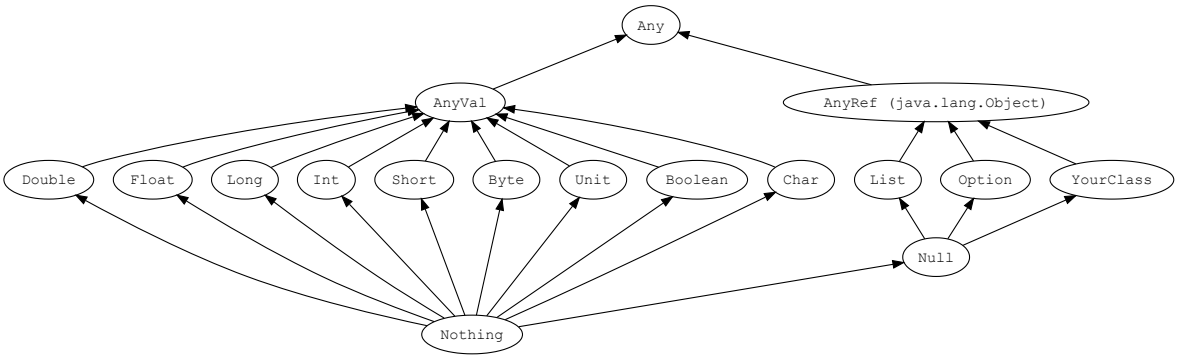
```
scala> val str = "Hello Scala"
str: String = Hello Scala
```

第5节 类层次结构

Scala中，所有的类，包括值类型和引用类型，都最终继承自一个统一的根类型Any。

Scala中定义了以下三个底层类：

- Any是所有类型共同的根类型，Any是AnyRef和AnyVal的超类
- AnyRef是所有引用类型的超类
- AnyVal是所有值类型的超类



上图中有三个类型需要注意：

- Null是所有引用类型的子类型

  Null类只有一个实例对象null。

  null可以赋值给任意引用类型，但是不能赋值给值类型。

- Nothing位于Scala类继承关系的底部，它是其他所有其他类型的子类型
  - Nothing对泛型结构有用 。比如，空列表Nil的类型就是List[Nothing]
  - Nothing的可以给出非正常终止的信号。比如，使用Nothing处理异常
- Unit类型用来标识过程，过程就是没有返回值的方法，Unit类似于Java里的void。**Unit只有一个实例()。**

```
-- null 不能赋值给值类型
scala> val i: Int = null
<console>:11: error: an expression of type Null is ineligible for implicit
conversion
       val i: Int = null

scala> val str: String = null
str: String = null

-- 使用 Nothing 处理异常
val test = false
val thing: Int = if (test) 42 else throw new Exception("ERROR!")

-- Unit类型只有一个实例()，该实例没有实际意义
scala> val a = ()
a: Unit = ()
```

第6节 值与变量&自动类型推断

Scala当中的声明变量可以使用以下两种方式：

- val，值 -- value，用val定义的变量，值是不可变的
- var，变量 -- variable，用var定义的变量，值是可变的

在Scala中，鼓励使用val。大多数程序并不需要那么多的var变量。

声明变量时，可以不指定变量的数据类型，编译器会根据赋值内容自动推断当前变量的数据类型。

备注：简单数据类型可以省略，对于复杂的数据类型建议明确声明；

声明变量时，可以将多个变量放在一起声明。

```
-- val定义的变量不可更改，变量的类型编译器可以进行自动类型推断
val name = "zhangsan"

-- 必要时可以指定数据类型
var name: String = null

-- 可以将多个值或变量放在一起声明
val x, y = 100;
var name, message: String = null
```

第7节 操作符

Scala的算术操作符、位操作符与 Java中的效果一样的。

需要特别注意一点：Scala中的操作符都是方法

```
a + b 等价 a.+(b)
1 to 10 等价 1.to(10)
```

书写时推荐使用：a + b 、1 to 10这种代码风格。

Scala 没有提供 ++、-- 操作符，但是可以使用+=、-=

第8节 块表达式和赋值语句

{} 块包含一系列表达式，其结果也是一个表达式，块中最后一个表达式的值就是块的值。

赋值语句返回Unit类型，代表没有值；

```
val x1 = 1
val y1 = 1
val x2 = 0
val y2 = 0
val distance = {
    val dx = x1 - x2
    val dy = y1 - y2
    math.sqrt(dx*dx + dy*dy)
}

-- 赋值语句的值是Unit类型，不要把它们串接在一起。x的值是什么?
var y = 0
val x = y = 1
```

第9节 输入和输出

通过readLine 从控制台读取一行输入。

如果要读取数字、Boolean或者字符，可以用readInt、readDouble、readByte、readShort、readLong、readFloat、readBoolean或者readChar。

print、println、printf 可以将结果输出到屏幕；

```
-- printf 带有C语言风格的格式化字符串的 printf 函数
printf("Hello, %s! You are %d years old.", "Scala", 18)
```

第10节 字符串插值器

Scala 提供了三种字符串插值器：

- s 插值器，对内嵌的每个表达式求值，对求值结果调用toString，替换掉字面量中的那些表达式
- f 插值器，它除s插值器的功能外，还能进行格式化输出，在变量后用%指定输出格式，使用java.util.Formatter中给出的语法
- raw 插值器，按照字符串原样进行输出

```
-- s插值器
val subject = "Spark"
val str1 = s"Hello, $subject"
println(str1)

val arr = (1 to 10).toArray
val str2 = s"arr.length = ${arr.length}"
println(str2)

println(s"The answer is ${6*6}")

-- f插值器
val year=2020
val month=6
val day=9
println(s"$year-$month-$day")

-- yyyy-MM-dd，不足2位用0填充
println(f"$year-$month%02d-$day%02d")

-- raw插值器
println("a\nb\tc")
println(raw"a\nb\tc")
println("""a\nb\tc""")
```

第11节 对象相等性

Java 中可以 == 来比较基本类型和引用类型：

- 对基本类型而言，比较的是值的相等性
- 对引用类型而言，比较的是引用相等性，即两个变量是否指向JVM堆上的同个对象

Scala中，要比较两个基础类型的对象是否相等，可以使用 == 或 !=；

```
1 == 1
1 != 2
2 == 2
```

== 或 != 可以比较同一类型的两个对象；

```
List(1,2,3) == List(1,2,3)
List(1,2,3) != Array(4,5,6)
```

== 或 != 还可以比较不同类型的两个对象：

```
2 == 2.0
List(1,2,3) == "Scala"
```

第二部分 控制结构和函数

第1节 if 表达式

Scala中 if 表达式有返回值。

如果if 和 else 的返回值类型不一样，那么就返回两个返回值类型公共的父类。

```
-- if 语句有返回值
val x = 10
val s = if (x > 0)
  1
else
  -1

-- 多分支if 语句
val s = if (x==0)
  0
else if (x > 1)
  1
else
  0

-- 如果返回的类型不一致就返回公共的父类
val s = if (x > 0)
  "positive"
else
  -1

-- 缺省 else 语句：s1/s2的返回类型不同
val s1 = if (x > 0) 1                等价      val s1 = if (x > 0) 1 else ()
val s2 = if (x > 0) "positive"       等价      val s2 = if (x > 0) "positive" else
()
```

第2节 for 表达式

Scala中，for循环语法结构：for (i <- 表达式 / 集合)，让变量 i遍历<-右边的表达式/集合的所有值。

Scala为for循环提供了很多的特性，这些特性被称之为 for守卫式 或 for推导式。

```
-- 基本结构。使用to实现左右两边闭合的访问区间
```

```
for (i <- 1 to 10) {
  println(s"i = $i")
}

-- 基本结构。使用until实现左闭右开的访问区间
for (i <- 1 until 10) {
  println(s"i = $i")
}

-- 双重循环。条件之间使用分号分隔
for (i <- 1 until 5; j <- 2 until  5){
  println(i * j )
}

-- 使用变量
for (i <- 1 to 3 ;j = 4-i){
  println(i * j )
}

-- 守卫语句。增加 if 条件语句
for (i <- 1 to 10; j <- 1 to 10 if i==j){
  println(s"i * j = $i * $j = ${i * j}")
}

-- 使用 yield 接收返回的结果，这种形式被称为for推导式
val result = for (i <- 1 to 10) yield i

-- 使用大括号将生成器、守卫、定义包含在其中；并以换行的方式来隔开它们
for { i <- 1 to 3
   from = 4 - i
   j <- from to 3 }
   println(s"i = $i; j = $j")
```

第3节 while 表达式

Scala提供了与 Java 类似的while和do...while循环。while语句的本身没有任何返回值类型，即while语句的返回结果是Unit类型的 () 。

Scala内置控制结构特地去掉了 break 和 continue。

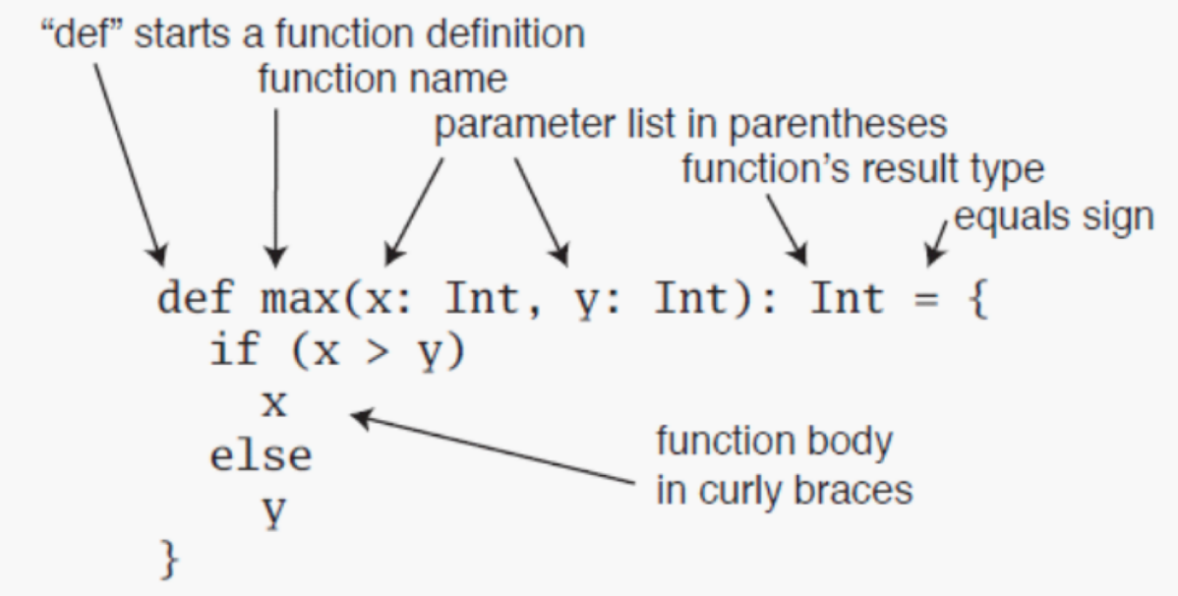
特殊情况下如果需要终止循环，可以有以下三种方式：

- 使用Boolean类型的控制变量
- 使用return关键字
- 使用breakable和break，需要导入scala.util.control.Breaks包

```
    // while循环
    var flag = true
    var result = 0
    var n = 0

    while (flag) {
      result += n
      n += 1
      println("res = " + result)
```

```
      println("n = " + n)
      if (n == 10) {
        flag = false
      }
    }

    // for循环
    var flag = true
    var res = 0

    for (i <- 0 until 10 if flag) {
      res += i
      println("res = " + res )
      if (i == 5) flag = false
    }


/**
    * 1 + 2 + 3 + 4
    *
    * @return
    */
   def addInner() {
     for (i <- 0 until 10) {
       if (i == 5) {
         return
       }
       res += i
       println("res = " + res)
     }
   }

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

def main(args: Array[String]): Unit = {
    // 需要导包
    import scala.util.control.Breaks._
    var res = 0
    breakable {
      for (i <- 0 until 10) {
        if (i == 5) {
          break
        }
        res += i
      }
    }
    println("res = " + res)
  }
```

第4节 函数



函数体中最后一句为返回值的话，可以将return 去掉；如果一个函数体只有一句代码，大括号可以去掉；

如果一个函数没有返回值，其返回类型为Unit , 并且 "=" 号可以去掉，这样的函数被称为过程；

可以不声明函数的返回类型，返回类型可通过自动类型推断来完成，但递归函数的返回类型必须声明；

备注：建议明确声明函数的返回值，即使为Unit

```
-- 定义函数
def add(x: Int, y: Int): Int = {
    x + y
}

-- 递归函数必须声明返回类型
def fibonacci(n: Int): Long = {
    if(n > 0){
        if(n==1 || n==2)
            1
        else
            fibonacci(n - 1) + fibonacci(n - 2)
    }
}

-- 参数的默认值
def add(x: Int, y: Int=10): Int = {
    x + y
}
add(1)
add(1, 20)

-- 带名参数
def add(x: Int=1, y: Int=10, z: Int): Int = {
    x + y + z
}
-- 现在要求 x=1, y=10, c=100
add(1, 10, 100)
add(z=100)

-- 变长参数。x的数据类型可以简单的认为是Array[Int]
def add(x: Int*): Int = {
```

```
    x.sum
}

// 告诉编译器这个参数被当做参数序列处理。使用 parameter: _* 的形式
val arr = (1 to 10).toArray
add(a:_*)

// 变长参数只能出现在参数列表的尾部，只能有一个
// Error: *-parameter must come last
def add1(x: Int*, y: String) = {
  ... ...
}
```

第5节 懒值

当 val 被声明为lazy时(var不能声明为lazy)，它的初始化将被推迟，直到首次对此取值，适用于初始化开销较大的场景。

```
// 语句立刻执行，发现文件不存在，报错
val file1 = scala.io.Source.fromFile("src/test111.scala")

// 文件不存在时，不会报错。因为语句此时并没有被执行
lazy val file2 = scala.io.Source.fromFile("src/test111.scala")
println("OK1")
file2.getLines().size
// 先打印OK!，才报错
```

第6节 文件操作

导入scala.io.Source后，可引用Source中的方法读取文本文件的内容

```
import scala.io.{BufferedSource, Source}

object FileDemo {
  def main(args: Array[String]): Unit = {
    //注意文件的编码格式，如果编码格式不对，那么读取报错
    val file: BufferedSource = Source.fromFile("... ...", "GBK");
    val lines: Iterator[String] = file.getLines()

    for (line <- lines) {
      println(line)
    }

    //注意关闭文件
    file.close()
  }
}
```

如果要将文件内容转数组，直接调用toArray。

读取网络资源

```
import scala.io.{BufferedSource, Source}

object FileDemo2 {
  def main(args: Array[String]): Unit = {
    val source: BufferedSource = Source.fromURL("http://www.baidu.com")
    val string: String = source.mkString

    println(string)
    source.close()
  }
}
```

Scala没有内建的对写入文件的支持。要写入文本文件，可使用 java.io.PrintWriter

```
import java.io.PrintWriter

object FileDemo3 {
  def main(args: Array[String]): Unit = {
    val writer = new PrintWriter("../a.txt")
    for(i <- 1 to 100){
      writer.println(i)
      writer.flush()
    }
    writer.close()
  }
}
```

第三部分 数组和元组

第1节 数组定义

数组几乎是所有语言中最基础的数据结构。数组可索引、类型一致、长度不变。

```
-- 长度为10的整型数组，初始值为0
val nums = new Array[Int](10)

-- 使用()访问数据元素；下标从0开始
nums(9) = 10

-- 长度为10的字符串数组，初始值为null
val strs = new Array[String](10)

-- 省略new关键字，定义数组，scala进行自动类型推断
val arrays = Array(1, 2, 3)

-- 快速定义数组，用于测试
val numsTest = (1 to 100).toArray
```

第2节 变长数组

长度按需要变换的数组ArrayBuffer。Scala 中很多数组类型都有可变、不可变两个版本，推荐使用不可变的数组类型，使用可变数组类型时需要显示声明；

使用ArrayBuffer时，需要导包 import scala.collection.mutable.ArrayBuffer；

```
import scala.collection.mutable.ArrayBuffer

object VarArrayDemo {
  def main(args: Array[String]){
    // 定义一个空的可变长Int型数组。注意：后面要有小括号
    val nums =  ArrayBuffer[Int]()

    // 在尾端添加元素
    nums += 1

    // 在尾端添加多个元素
    nums += (2,3,4,5)

    // 使用++=在尾端添加任何集合
    nums ++= Array(6,7,8)

   // 这些操作符，有相应的 -= ， --=可以做数组的删减，用法同+=，++=

   // 使用append追加一个或者多个元素
    nums.append(1)
    nums.append(2,3)

   // 在下标2之前插入元素
    nums.insert(2,20)
    nums.insert(2,30,30)

    // 移除最后2元素
    nums.trimEnd(2)
    // 移除最开始的一个或者多个元素
    nums.trimStart(1)

    // 从下标2处移除一个或者多个元素
    nums.remove(2)
    nums.remove(2,2)

  }
}
```

第3节 数组操作

数组转换

```
// Array <==> BufferArray定长数组与变长数组转换
//toArray，变长数组转换为定长数组
val array: Array[Int]=nums.toArray

//toBuffer，定长数组转换为变长数组
val arrayBuffer: mutable.Buffer[Int]=array.toBuffer
```

数组遍历

```
// 使用until，基于下标访问使用增强for循环进行数组遍历
for (i <- 0 until nums.length)
    println(nums(i))

// 使用to，基于下标访问使用增强for循环进行数组遍历
for (i <- 0 to nums.length-1)
    println(nums(i))

// 使用增强for循环遍历数组元素
for (elem <- nums)
    println(elem)
```

第4节 常见算法

在Scala中对数组进行转换非常简单方便，这些转换动作不会修改原始数组，而是产生一个全新的数组。

任务：将数组中偶数元素加倍，奇数元素丢弃

```
val arr = (1 to 10).toArray

-- 使用for推导式。注意：原来的数组并没有改变
val result1 = for (elem <- arr) yield if (elem % 2 == 0) elem * 2 else 0
val result2 = for (elem <- arr if elem %2 == 0) yield elem * 2

-- scala中的高阶函数
arr.filter(_%2==0).map(_*2)
```

```
// 取第一个元素
a1.head
// 取最后一个元素
a1.last
// 除了第一个元素，剩下的其他元素
a1.tail
// 除了最后一个元素，剩下其他元素
a1.init

// 数组常用算法
Array(1,2,3,4,5,6,7,8,9,10).sum //求和
Array(2,3,4).product //元素相乘
Array(1,2,3,4,5,6,7,8,9,10).max  //求最大值
Array(1,2,3,4,5,6,7,8,9,10).min  //求最小值
```

```
Array(1,3,2,7,6,4,8,9,10).sorted  //升序排列
// max、min、sorted方法，要求数组元素支持比较操作

Array(1,2,3,4,5,4,3,2,1).map(_*2)
Array(1,2,3,4,5,4,3,2,1).reduce(_+_)
Array(1,2,3,4,5,4,3,2,1).distinct //数据去重
Array(1,2,3,4,5,4,3,2,1).length
Array(1,2,3,4,5,4,3,2,1).size
Array(1,2,3,4,5,4,3,2,1).indices //数据索引

// count计数，需要注意的是count中必须写条件
Array(1,2,3,4,5,4,3,2,1).count(_>3)
Array(1,2,3,4,5,4,3,2,1).count(_%2==0)

// filter 过滤数据，原始数组中的数据保持不变，返回一个新的数组
Array(1,2,3,4,5,4,3,2,1).filter(_>3)
Array(1,2,3,4,5,4,3,2,1).filterNot(_%2==0)

// 在REPL环境中输入数组名称即可打印数组元素，非常方便
// 在IDEA中，print(a) / print(a.toString)都不能打印数组元素
// 使用mkString / toBuffer 是打印数组元素简单高效的方法
Array(10,9,8,7,6,5,4,3,2,1).toString
Array(10,9,8,7,6,5,4,3,2,1).mkString(" & ")
Array(10,9,8,7,6,5,4,3,2,1).mkString("<", " & ", ">")
Array(10,9,8,7,6,5,4,3,2,1).toBuffer

// take取前4个元素；takeRight取后4个元素
//原始数组中的数据保持不变，返回一个新的数组
Array(1,2,3,4,5,6,7,8,9,10).take(4)
Array(1,2,3,4,5,6,7,8,9,10).takeRight(4)

// takeWhile 从左向右提取列表的元素，直到条件不成立（条件不成立时终止）
Array(1,2,3,4,5,6,1,2,3,4).takeWhile(_<5)

// drop 删除前4个元素；dropRight删除后4个元素；
// dropWhile删除元素，直到条件不成立
Array(1,2,3,4,5,6,7,8,9,10).drop(4)
Array(1,2,3,4,5,6,7,8,9,10).dropRight(4)
Array(1,2,3,4,5,6,1,2,3,4).dropWhile(_<5)

// 将数组分为前n个，与剩下的部分
Array(1,2,3,4,5,6,7,8,9,10).splitAt(4)

// 数组切片。取下标第2到第4的元素（不包括第5个元素）
// 返回结果：Array(2, 3, 4)
Array(0,1,2,3,4,5,6,7,8,9,10).slice(2,5)

// 拉链操作；a1,a2的长度不一致时，截取相同的长度
val a1 = Array("A","B","C")
val a2 = Array(1,2,3,4)
val z1 = a1.zip(a2)

// 拉链操作；a1,a2的长度不一致时，a1用 * 填充，a2用 -1 填充
val z2 = a1.zipAll(a2, "*", -1)
val z3 = a1.zipAll(a2, -1, "*")

// 用数组索引号填充
val z4 = a1.zipWithIndex
```

```
// unzip 的逆操作，拆分成2个数组
val (l1,l2) = z4.unzip
// unzip3拆分成3个数组
val (l1,l2,l3) = Array((1, "one", '1'),(2, "two", '2'),(3, "three", '3')).unzip3

// 用于数组的操作符(:+、+:、++)
// :+ 方法用于在尾部追加元素；+: 方法用于在头部追加元素；
// 备注：冒号永远靠近集合类型，加号位置决定元素加在前还是后；
// ++ 该方法用于连接两个集合(数组、列表等)，arr1 ++ arr2；
val a = (1 to 4).toArray
val b = (5 to 8).toArray

// 分别在集合头部、尾部增加元素；连接两个集合
val c = 10 +: a
val d = c :+ 9
val e = a ++ b

// 说明：上述的很多方法不仅仅对Array适用，一般情况下对其他集合类型同样适用。
val list = (1 to 10).toList
list.sum
list.max
list.take(4)
list.drop(4)

//数组排序
val nums = Array(1, 3, 2, 6, 4, 7, 8, 5)
println(nums.sorted.toBuffer) //升序
println(nums.sorted.reverse.toBuffer) //降序
println(nums.sortWith(_ > _).toBuffer) //降序
println(nums.sortWith(_ < _).toBuffer) //升序
```

第5节 多维数组

通过Array的ofDim方法来定义一个多维的数组，多少行，多少列，都是自己说了算。

```
//创建一个3行4列的二维数组
val dim = Array.ofDim[Double](3,4)
dim(1)(1) = 11.11
for (i <- 0 to 2; j <- 0 to 3) {
    print(dim(i)(j) + "  ")
    if (j == 3) println()
}
```

第6节 元组及操作

Tuple，元组。Map是键值对的集合。对偶是元组的最简单形态；

元组是不同类型的值的集合，元组中的元素可以是不同的数据类型，元组在Scala中的应用非常广泛。

```
//报错，元组的元素个数上限是22个
val a = Tuple23(1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3)
```

```
// 定义元组
val a = (1, 1.2, "ad", 'd')
val b = Tuple4(1, 1.2, "ad", 'd')
println(a==b)

// Tuple的访问形式比较特殊。元组的下标从1开始
a._1
a _2
a._3
a _4

// 从元组接收数据
val (a1, a2, a3, a4), a5 = a
val (b1, _, b2, _), b5 = a

// 遍历元组，第一种方式
for(x <- a.productIterator){
  println(x)
}

// 遍历元组，第二种方式
a.productIterator.foreach(x => println(x))
```

第四部分 类与对象

第1节 类和无参构造器

在Scala中，类并不用声明为public；

Scala源文件中可以包含多个类，所有这些类都具有公有可见性；

val修饰的变量（常量），值不能改变，只提供getter方法，没有setter方法；

var修饰的变量，值可以改变，对外提供getter、setter方法；

如果没有定义构造器，类会有一个默认的无参构造器；

```
class Person {
  // Scala中声明一个字段，必须显示的初始化，然后根据初始化的数据类型自动推断其类型，字段类型可以省略
  var name = "jacky"

  // _ 表示一个占位符，编译器会根据变量的数据类型赋予相应的初始值
  // 使用占位符，变量类型必须指定
  // _ 对应的默认值:整型默认值0；浮点型默认值0.0；String与引用类型，默认值null; Boolean默认值false
  var nickName: String = _
  var age=20
  // 如果赋值为null,则一定要加数据类型，因为不加类型， 该字段的数据类型就是Null类型
  // var address = null
  // 改为：
  var address: String = null

  // val修饰的变量不能使用占位符
```

```
  val num = 30

  // 类私有字段,有私有的getter方法和setter方法，
  // 在类的内部可以访问，其伴生对象也可以访问
  private var hobby: String = "旅游"

  // 对象私有字段,访问权限更加严格，只能在当前类中访问
  private[this] val cardInfo = "123456"

  //自定义方法
   def hello(message: String): Unit = {
    //只能在当前类中访问cardInfo
    println(s"$message,$cardInfo")
  }
  //定义一个方法实现两数相加求和
  def addNum(num1: Int, num2: Int): Int = {
    num1 + num2
  }
}
```

类的实例化以及使用：

```
object ClassDemo {
  def main(args: Array[String]): Unit = {
    //创建对象两种方式，这里都是使用的无参构造器来进行创建对象的
    val person = new Person()
    //创建类的对象时，小括号()可以省略
    val person1 = new Person
    //给类的属性赋值
    person.age = 50
    //注意：如果使用对象的属性加上 _= 给var修饰的属性进行重新赋值，其实就是调用age_=这个
setter方法
    person.age_=(20)
    //直接调用类的属性，其实就是调用getter方法
    println(person.age)
    //调用类中的方法
    person.hello("hello")
    val result = person.addNum(10, 20)
    println(result)
  }
}
```

第2节 自定义getter和setter方法

对于 Scala 类中的每一个属性，编译后会有一个私有的字段和相应的getter、setter方法生成。

```
//getter方法
println(person age)

//setter方法
person age_= (18)

//getter方法
println(person.age)
```

可以不使用自动生成的方式，自己定义getter和setter方法

```
class Dog {
  private var _leg = 0
  //自定义getter方法
  def leg = _leg
  //自定义setter方法
  def leg_=(newLeg: Int) {
    _leg = newLeg
  }
}

// 使用自定义getter和setter方法
val dog = new Dog
dog.leg_=(4)
println(dog.leg)
```

自定义变量的getter和setter方法需要遵循以下原则：

- 字段属性名以“_”作为前缀，如： _leg
- getter方法定义为：def leg = _leg
- setter方法定义为：def leg_=(newLeg: Int)

第3节 Bean属性

JavaBean规范把Java属性定义为一堆getter和setter方法。

类似于Java，当将Scala字段标注为 @BeanProperty时，getFoo和setFoo方法会自动生成。

使用@BeanProperty并不会影响Scala自己自动生成的getter和setter方法。

在使用时需要导入包scala.beans.BeanProperty

```
import scala.beans.BeanProperty

class Teacher {
    @BeanProperty var name:String = _
}

object BeanDemo{
  def main(args: Array[String]): Unit = {
    val tea: Teacher = new Teacher
    tea.name = "zhagnsan"
    tea.setName("lisi")    //BeanProperty生成的setName方法
    println(tea.getName)   //BeanProperty生成的getName方法
  }
```

```
}
```

上述Teacher类中共生成了四个方法：

```
1. name: String
2. name_= (newValue: String): Unit
3. getName(): String
4. setName (newValue: String): Unit
```

第4节 构造器

如果没有定义构造器，Scala类中会有一个默认的无参构造器；

Scala当中类的构造器分为两种：主构造器和辅助构造器；

主构造器的定义与类的定义交织在一起，将主构造器的参数直接放在类名之后。

当主构造器的参数不用var或val修饰时，参数会生成类的私有val成员。

Scala中，所有的辅助构造器都必须调用另外一个构造器，另外一个构造器可以是辅助构造器，也可以是主构造器。

```
//主构造器直接定义在类中，其代码不包含在任何方法中
//Scala中的主构造器与类名交织在一起，类名后面的参数即为主构造器的参数
class Dog(name: String, age: Int) {
  //类中不在任何方法中的代码，都属于主构造器的代码。
  //创建类的对象时会去执行主构造器的代码。下面的println代码就是主构造器的一部分
  println(name)
  println(age)

  var gender: String = ""

  def this(name: String, age: Int, gender: String) {
    //每个辅助构造器，都必须以其他辅助构造器，或者主构造器的调用作为第一句代码
    this(name, age)
    this.gender = gender
  }

  var color = ""

  def this(name: String, age: Int, gender: String, color: String) {
    //调用上面的辅助构造器
    this(name, age, gender)
    this.color = color
  }
}

object Dog {
  def main(args: Array[String]): Unit = {
    val dog1=new Dog("狗蛋",4)
    val dog2=new Dog("旺才",3,"雄性")
    val dog3=new Dog("小六",5,"雄性","黑色")
  }
}
```

第5节 对象

5.1 单例对象

Scala并没有提供Java那样的静态方法或静态字段；

可以采用object关键字实现单例对象，具备和Java静态方法同样的功能；

使用object语法结构【object是Scala中的一个关键字】达到静态方法和静态字段的目的；对象本质上可以拥有类的所有特性，除了不能提供构造器参数；

对于任何在Java中用单例对象的地方，在Scala中都可以用object实现：

- 作为存放工具函数或常量的地方
- 高效地共享单个不可变实例

```
class Session {
  def hello(first: Int): Int = {
    println(first)
    first
  }
}

object SessionFactory {
  val session = new Session

  def getSession(): Session = {
    session
  }

  def main(args: Array[String]): Unit = {
    for (x <- 1 to 10) {
      //通过直接调用，产生的对象都是单例的
      val session = SessionFactory.getSession()
      println(session)
    }
  }
}
```

Scala中的单例对象具有如下特点：

1、创建单例对象不需要使用new关键字

2、object中只有无参构造器

3、主构造代码块只能执行一次，因为它是单例的

```
object ObjectDemo {
    println("这是单例对象的代码！")

    def main(args: Array[String]): Unit = {
        val object1=ObjectDemo
        val object2=ObjectDemo
    }
}
```

5.2 伴生类与伴生对象

当单例对象与某个类具有相同的名称时，它被称为这个类的“伴生对象”；

类和它的伴生对象必须存在于同一个文件中，而且可以相互访问私有成员（字段和方法）；

```
class ClassObject {
  val id = 1
  private var name = "lagou"
  def printName(): Unit ={
    //在ClassObject类中可以访问伴生对象ClassObject的私有字段
    println(ClassObject.CONSTANT + name )
  }
}

object ClassObject{
  //伴生对象中的私有字段
  private val CONSTANT = "汪汪汪"
  def main(args: Array[String]) {
    val p = new ClassObject
    //访问伴生类的私有字段name
    p.name = "123"
    p.printName()
  }
}
```

5.3 应用程序对象

每个Scala应用程序都必须从一个对象的main方法开始，这个方法的类型为 Array[String] => Unit；

备注：main方法写在class中是没有意义的，在IDEA中这样的 class 连run的图标都不能显示

除了main方法以外，也可以扩展App特质（trait）

```
object Hello extends App {
  if (args.length > 0)
    println(s"Hello World; args.length = ${args.length}")
  else
    println("Hello World")
}
```

5.4 apply方法

object 中有一个非常重要的特殊方法 -- apply方法；

- **apply方法通常定义在伴生对象中**，目的是通过伴生类的构造函数功能，来实现伴生对象的构造函数功能；
- 通常我们会在类的伴生对象中定义apply方法，**当遇到类名(参数1,...参数n)时apply方法会被调用**；
- 在创建伴生对象或伴生类的对象时，通常不会使用new class/class() 的方式，而是**直接使用class()隐式的调用伴生对象的 apply 方法**，这样会让对象创建的更加简洁；

```scala
//class Student为伴生类
class Student(name: String, age: Int) {
  private var gender: String = _

  def sayHi(): Unit ={
    println(s"大家好，我是$name,$gender 生")
  }
}

//object Student是class class的伴生对象
object Student {
  //apply方法定义在伴生对象中
  def apply(name: String, age: Int): Student = new Student(name, age)

  def main(args: Array[String]): Unit = {
    //直接利用类名进行对象的创建，这种方式实际上是调用伴生对象的apply方法实现的
    val student=Student("jacky",30)
    student.gender="男"

    student.sayHi()
  }
}
```

问题：在Scala中实现工厂方法，让子类声明哪种对象应该被创建，保持对象创建在同一位置。例如，假设要创建Animal工厂，让其返回Cat和Dog类的实例，基于这个需求，通过实现Animal伴生对象的apply方法，工厂的使用者可以像这样创建新的Cat和Dog实例。

```scala
abstract class Animal {
  def speak
}

class Dog extends Animal {
  override def speak: Unit = {
    println("woof")
  }
}

class Cat extends Animal {
  override def speak: Unit = {
    println("meow")
  }
}

object Animal {
  def apply(str: String): Animal = {
    if (str == "dog")
      new Dog
    else
      new Cat
  }

  def main(args: Array[String]): Unit = {
    val cat = Animal("cat")
    cat.speak

    val dog = Animal("dog")
```

```
    dog.speak
  }
}
```

第五部分 继承

第1节 继承的概念

Scala中继承类的方式和Java一样，也是使用extends关键字：

```
class Employee extends Person{
    var salary=1000
}
```

和Java一样，可在定义中给出子类需要而父类没有的字段和方法，或者重写父类的方法。

```
//Person类
class Person(name:String,age:Int)
//Student继承Person类
class Student(name:String,age:Int,var studentNo:String) extends Person(name,age)
object Demo{
  def main(args: Array[String]): Unit = {
     val student=new Student("john",18,"1024")
  }
}
```

上面继承部分的代码等效于下面的Java代码

```
//Person类
class Person{
    private String name;
    private int age;
    public Person(String name,int age){
       this.name=name;
       this.age=age;
    }
}
//Student继承Person类
class Student extends Person{
    private String studentNo;
    public Student(string name,int age,String studentNo){
        super(name,age);
        this.sutdentNo=studentNo;
    }
}
```

第2节 构造器执行顺序

Scala在继承的时候构造器的执行顺序：首先执行父类的主构造器，其次执行子类自身的主构造器。

类有一个主构造器和任意数量的辅助构造器，而每个辅助构造器都必须以对先前定义的辅助构造器或主构造器的调用开始。

子类的辅助构造器最终都会调用主构造器。只有主构造器可以调用父类的构造器。

```
//Person类
class Person(name:String,age:Int){
  println("这是父类Person")
}
//Student继承Person类
class Student(name:String,age:Int,studentNo:String) extends Person(name,age){
  println("这是子类Student")
}

object Demo{
  def main(args: Array[String]): Unit = {
     //下面的语句执行时会打印下列内容:
     //这是父类Person
     //这是子类Student
     //也就是说，构造Student对象之前，首先会调用Person的主构造器
     val student=new Student("john",18,"1024")
  }
}
```

第3节 override方法重写

方法重写指的是当子类继承父类的时候，从父类继承过来的方法不能满足子类的需要，子类希望有自己的实现，这时需要对父类的方法进行重写，方法重写是实现多态的关键。
Scala中的方法重写同Java一样，也是利用override关键字标识重写父类的方法。

```
class Programmer(name:String,age:Int){
  def coding():Unit=println("我在写代码...")
}
//ScalaProgrammer继承Programmer类
class ScalaProgrammer(name:String,age:Int,workNo:String) extends
Programmer(name,age){
  override def coding():Unit={
    //调用父类的方法
    super.coding()
    //增加了自己的实现
    println("我在写Scala代码...")
  }
}

object ExtendsDemo {
  def main(args: Array[String]): Unit = {
    val scalaProgrammer=new ScalaProgrammer("张三",30,"1001")
    scalaProgrammer.coding()
  }
}
//代码运行输出内容:
我在写代码...
我在写Scala代码...
```

需要强调一点：如果父类是抽象类，则override关键字可以不加。如果继承的父类是抽象类（假设抽象类为AbstractClass，子类为SubClass），在SubClass类中，AbstractClass对应的抽象方法如果没有实现的话，那SubClass也必须定义为抽象类，否则的话必须要有方法的实现。

```
//抽象的Person类
abstract class Person(name:String,age:Int){
  def walk():Unit
}
//Student继承抽象Person类
class Student(name:String,age:Int,var studentNo:String) extends Person(name,age)
{
  //重写抽象类中的walk方法，可以不加override关键字
  def walk():Unit={
    println("walk like a elegant swan")
  }
}
object Demo{
  def main(args: Array[String]): Unit = {
     val stu=new Student("john",18,"1024")
     stu.walk()
  }
}
```

第4节 类型检查与转换

要测试某个对象是否属于某个给定的类，可以用isInstanceOf方法。如果测试成功，可以用asInstanceOf方法进行类型转换。

```
if(p.isInstanceOf[Employee]){
    //s的类型转换为Employee
    val s = p.asInstanceOf[Employee]
}
```

如果p指向的是Employee类及其子类的对象，则p.isInstanceOf[Employee]将会成功。

如果p是null，则p.isInstanceOf[Employee]将返回false，且p.asInstanceOf[Employee]将返回null。

如果p不是一个Employee，则p.asInstanceOf[Employee]将抛出异常。

如果想要测试p指向的是一个Employee对象但又不是其子类，可以用：

```
if(p.getClass == classOf[Employee])
```

classOf方法定义在scala.Preder对象中，因此会被自动引入。

不过，与类型检查和转换相比，模式匹配通常是更好的选择。

```
p match{
    //将s作为Employee处理
    case s: Employee => ...
    //p不是Employee的情况
    case _ => ....
}
```

第六部分 特质

第1节 作为接口使用的特质

Scala中的trait特质是一种特殊的概念。

首先可以将trait作为接口来使用，此时的trait就与Java中的接口非常类似。

在trait中可以定义抽象方法，与抽象类中的抽象方法一样，只要不给出方法的具体实现即可。

类可以使用extends关键字继承trait。

注意：在Scala中没有implement的概念，无论继承类还是trait特质，统一都是extends。

类继承trait特质后，必须实现其中的抽象方法，实现时**可以省略override关键字**。

Scala不支持对类进行多继承，但是**支持多重继承trait特质，使用with关键字即可。**

```
//定义一个trai特质
trait HelloTrait {
  def sayHello
}
//定义一个trai特质
trait MakeFriendTrait {
  def makeFriend
}
//继承多个trait,第一个trait使用extends关键字，其它trait使用with关键字
class Person(name: String) extends HelloTrait with MakeFriendsTrait with
Serializable {
  override def sayHello() = println("Hello, My name is " + name)
  //override关键字也可以省略
  def makeFriend() = println("Hello, " + name)
}
```

第2节 带有具体实现的特质

具体方法

Scala中的trait特质不仅仅可以定义抽象方法，还可以定义具体实现的方法，这时的trait更像是包含了通用工具方法的类。比如，trait中可以包含一些很多类都通用的功能方法，比如打印日志等等，Spark中就使用了trait来定义通用的日志打印方法。

具体字段

Scala trait特质中的字段可以是抽象的，也可以是具体的。

```
trait People {
  //定义抽象字段
  val name: String
  //定义了age字段
  val age = 30

  def eat(message: String): Unit = {
    println(message)
```

```
  }
}

trait Worker {
  //这个trait也定义了age字段
  val age = 25

  def work: Unit = {
    println("Working......")
  }
}
// Student类继承了Worker、Person这两个特质，需要使用extends、with这两个关键字
class Student extends Worker with People{
  //重写抽象字段，override可以省略
  override val name: String = "张三"
  //继承的两个trait中都有age字段，此时需要重写age字段，override不能省略
  override val age = 20
}

object TraitDemoTwo {
  def main(args: Array[String]): Unit = {
    val stu = new Student
    stu.eat("吃饭")
    stu.work
    println(s"Name is ${stu.name}, Age is ${stu.age}")
  }
}
```

注意：特质Person和Worker中都有age字段，当Student继承这两个特质时，需要重写age字段，并且要用override关键字，否则就会报错。

第3节 特质构造顺序

在Scala中，trait特质也是有构造器的，也就是trait中的不包含在任何方法中的代码。

构造器以如下顺序执行：

1、执行父类的构造器；

2、执行trait的构造器，多个trait从左到右依次执行；

3、构造trait时会先构造父trait，如果多个trait继承同一个父trait，则父trait只会构造一次；

4、所有trait构造完毕之后，子类的构造器才执行

```
class Person2 { println("Person's constructor!") }

trait Logger { println("Logger's constructor!") }

trait MyLogger extends Logger { println("MyLogger's constructor!") }

trait TimeLogger extends Logger { println("TimeLogger's constructor!") }
//类既继承了类又继承了特质，要先写父类
class Student2 extends Person2 with MyLogger with TimeLogger {

  println("Student's constructor!")

}
```

上面代码的输出结果：

```
Person's constructor!
Logger's constructor!
MyLogger's constructor!
TimeLogger's constructor!
Student's constructor!
```

第4节 特质继承类

在Scala中，trait特质也可以继承class类，此时这个class类就会成为所有继承此trait的类的父类。

```
class MyUtil {
  def printMessage(msg: String) = println(msg)
}

// 特质Log继承MyUtil类
trait Log extends MyUtil {
  def log(msg: String) = printMessage(msg)
}

// Person3类继承Log特质，Log特质继承MyUtil类，所以MyUtil类成为Person3的父类
class Person3(name: String) extends Logger {
  def sayHello {
    log("Hello, " + name)
    printMessage("Hi, " + name)
  }
}
```

第5节 Ordered和Ordering

在Java中对象的比较有两个接口，分别是Comparable和Comparator。它们之间的区别在于：

实现Comparable接口的类，重写compareTo()方法后，其对象自身就具有了可比较性；
实现Comparator接口的类，重写了compare()方法后，则提供一个第三方比较器，用于比较两个对象。

在Scala中也引入了以上两种比较方法(Scala.math包下):

Ordered特质混入Java的Comparable接口，它定义了相同类型间的比较方式，但这种内部比较方式是单一的；

```
trait Ordered[A] extends Any with java.lang.Comparable[A]{.....}
```

Ordering特质混入Comparator接口，它是提供第三方比较器，可以自定义多种比较方式，在实际开发中也是使用比较多的，灵活解耦合。

```
trait Ordering[T] extends Comparator[T] with PartialOrdering[T] with
Serializable {......}
```

使用Ordered特质进行排序操作

```
case class Project(tag:String, score:Int) extends Ordered[Project] {
  def compare(pro:Project ) = tag.compareTo(pro.tag)
}

object OrderedDemo {
  def main(args: Array[String]): Unit = {
    val list = List(Project("hadoop",60), Project("flink",90),
                    Project("hive",70),Project("spark",80))
    println(list.sorted)
  }
}
```

使用Ordering特质进行排序操作

```
object OrderingDemo {
  def main(args: Array[String]): Unit = {
    val pairs = Array(("a", 7, 2), ("c", 9, 1), ("b", 8, 3))
    // Ordering.by[(Int,Int,Double),Int](_._2)表示从Tuple3转到Int型
    // 并按此Tuple3中第二个元素进行排序
    Sorting.quickSort(pairs)(Ordering.by[(String, Int, Int), Int](_._2))
    println(pairs.toBuffer)
  }
}
```

第七部分 模式匹配和样例类

第1节 模式匹配

Scala没有Java中的switch case，它有一个更加强大的模式匹配机制，可以应用到很多场合。

Scala的模式匹配可以匹配各种情况，比如变量的类型、集合的元素、有值或无值。

模式匹配的基本语法结构：**变量 match { case 值 => 代码 }**

模式匹配match case中，只要有一个case分支满足并处理了，就不会继续判断下一个case分支了，不需要使用break语句。这点与Java不同，Java的switch case需要用break阻止。如果值为下划线，则代表不满足以上所有情况的时候如何处理。

模式匹配match case最基本的应用，就是对变量的值进行模式匹配。match是表达式，与if表达式一样，是有返回值的。

除此之外，Scala还提供了样例类，对模式匹配进行了优化，可以快速进行匹配。

第2节 字符和字符串匹配

```
def main(args: Array[String]): Unit = {
  val charStr = '6'
  charStr match {
    case '+' => println("匹配上了加号")
    case '-' => println("匹配上了减号")
    case '*' => println("匹配上了乘号")
    case '/' => println("匹配上了除号")
    //注意：不满足以上所有情况，就执行下面的代码
    case _  => println("都没有匹配上，我是默认值")
  }
}

def main(args: Array[String]): Unit = {
  val arr = Array("hadoop", "zookeeper", "spark")
  val name = arr(Random.nextInt(arr.length))
  name match {
    case "hadoop"    => println("大数据分布式存储和计算框架...")
    case "zookeeper" => println("大数据分布式协调服务框架...")
    case "spark" => println("大数据分布式内存计算框架...")
    case _ => println("我不认识你...")
  }
}
```

第3节 守卫式匹配

```
// 所谓守卫就是添加if语句
object MatchDemo {
  def main(args: Array[String]): Unit = {
    //守卫式
    val character = '*'
    val num = character match {
      case '+' => 1
      case '-' => 2
      case _ if character.equals('*') => 3
      case _ => 4
    }
    println(character + " " + num)
}
```

第4节 匹配类型

Scala的模式匹配还有一个强大的功能，它可以直接匹配类型，而不是值。这一点是Java的switch case做不到的。
匹配类型的语法：**case 变量 : 类型 => 代码**，而不是匹配值的“case 值 => 代码”这种语法。

```
def main(args: Array[String]): Unit = {
  val a = 3
  val obj = if(a == 1) 1
  else if(a == 2) "2"
  else if(a == 3) BigInt(3)
  else if(a == 4) Map("aa" -> 1)
  else if(a == 5) Map(1 -> "aa")
  else if(a == 6) Array(1, 2, 3)
  else if(a == 7) Array("aa", 1)
  else if(a == 8) Array("aa")

  val r1 = obj match {
    case x: Int => x
    case s: String => s.toInt
    // case BigInt => -1 //不能这么匹配
    case _: BigInt => Int.MaxValue
    case m: Map[String, Int] => "Map[String, Int]类型的Map集合"
    case m: Map[_, _] => "Map集合"
    case a: Array[Int] => "It's an Array[Int]"
    case a: Array[String] => "It's an Array[String]"
    case a: Array[_] => "It's an array of something other than Int"
    case _ => 0
  }
  println(r1 + ", " + r1.getClass.getName)

}
```

第5节 匹配数组、元组、集合

```
def main(args: Array[String]): Unit = {
  val arr = Array(0, 3, 5)
  //对Array数组进行模式匹配，分别匹配：
  //带有指定个数元素的数组、带有指定元素的数组、以某元素开头的数组
  arr match {
    case Array(0, x, y) => println(x + " " + y)
    case Array(0) => println("only 0")
    //匹配数组以1开始作为第一个元素
    case Array(1, _*) => println("1 ...")
    case _ => println("something else")
  }

  val list = List(3, -1)
  //对List列表进行模式匹配，与Array类似，但是需要使用List特有的::操作符
  //构造List列表的两个基本单位是Nil和::，Nil表示为一个空列表
  //tail返回一个除了第一元素之外的其他元素的列表
  //分别匹配：带有指定个数元素的列表、带有指定元素的列表、以某元素开头的列表
  list match {
    case x :: y :: Nil => println(s"x: $x y: $y")
    case 0 :: Nil => println("only 0")
    case 1 :: tail => println("1 ...")
    case _ => println("something else")
  }

  val tuple = (1, 3, 7)
  tuple match {
```

```
    case (1, x, y) => println(s"1, $x , $y")
    case (_, z, 5) => println(z)
    case  _ => println("else")
  }
}
```

第6节 样例类

case class样例类是Scala中特殊的类。当声明样例类时，以下事情会自动发生：

- 主构造函数接收的参数通常不需要显式使用var或val修饰，Scala会自动使用val修饰
- 自动为样例类定义了伴生对象，并提供apply方法，不用new关键字就能够构造出相应的对象
- 将生成toString、equals、hashCode和copy方法，除非显示的给出这些方法的定义
- 继承了Product和Serializable这两个特质，也就是说样例类可序列化和可应用Product的方法

case class是多例的，后面要跟构造参数，case object是单例的。

此外，case class样例类中可以添加方法和字段，并且可用于模式匹配。

```
class Amount
//定义样例类Dollar，继承Amount父类
case class Dollar(value: Double) extends Amount
//定义样例类Currency，继承Amount父类
case class Currency(value: Double, unit: String) extends Amount
//定义样例对象Nothing，继承Amount父类
case object Nothing extends Amount

object CaseClassDemo {
  def main(args: Array[String]): Unit = {
    judgeIdentity(Dollar(10.0))
    judgeIdentity(Currency(20.2,"100"))
    judgeIdentity(Nothing)
  }
  //自定义方法，模式匹配判断amt类型
  def judgeIdentity(amt: Amount): Unit = {
    amt match {
      case Dollar(value) => println(s"$value")
      case Currency(value, unit) => println(s"Oh noes,I got $unit")
      case Nothing => println("Oh,GOD!")
    }
  }
}
```

第7节 Option与模式匹配

Scala Option选项类型用来表示一个值是可选的，有值或无值。

Option[T] 是一个类型为 T 的可选值的容器，可以通过get()函数获取Option的值。如果值存在，Option[T] 就是一个 Some。如果不存在，Option[T] 就是对象 None 。

Option通常与模式匹配结合使用，用于判断某个变量是有值还是无值。

```
object OptionMatch {
```

```
  val grades = Map("jacky" -> 90, "tom" -> 80, "jarry" -> 95)

  def getGrade(name: String): Unit = {
    val grade = grades.get(name)
    grade match {
      case Some(grade) => println("成绩：" + grade)
      case None => println("没有此人成绩！")
    }
  }

  def main(args: Array[String]): Unit = {
    getGrade("jacky")
    getGrade("张三")
  }
}
```

第八部分 函数及抽象化

第1节 函数字面量及函数的定义

Scala中函数为头等公民，不仅可以定义一个函数然后调用它，还可以写一个未命名的函数字面量，然后可以把它当成一个值传递到其它函数或是赋值给其它变量。

函数字面量体现了函数式编程的核心理念。字面量包括整数字面量、浮点数字面量、布尔型字面量、字符字面量、字符串字面量、符号字面量、函数字面量等。什么是函数字面量呢?

在函数式编程中，函数是“头等公民”，可以像任何其他数据类型一样被传递和操作。函数的使用方式和其他数据类型的使用方式完全一致，可以像定义变量那样去定义一个函数，函数也会和其他变量一样，有类型有值；

就像变量的“类型”和“值”是分开的两个概念一样，函数的“类型”和“值”也成为两个分开的概念；

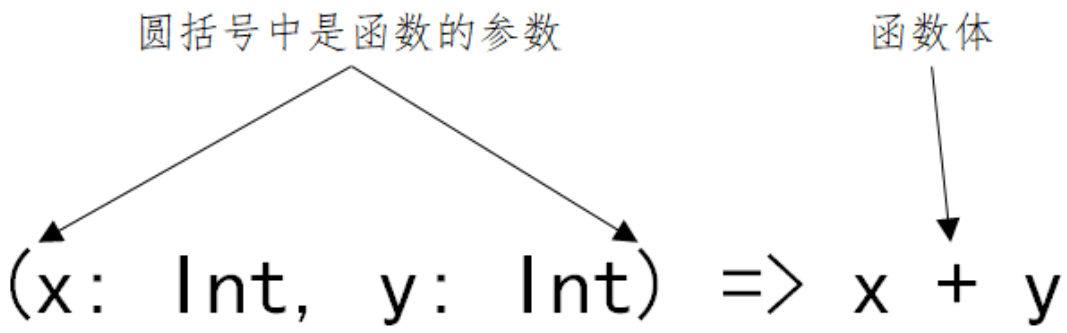
函数的“值”，就是“函数字面量”。

```
scala> def add1(x: Int): Int =  { x + 1 }
add1: (x: Int)Int
// 函数的类型为: (Int) => Int
// 输入参数列表只有一个括号，可以简写为: Int => Int

scala> def add2(x: Int, y: Int): Int = { x + y }
add2: (x: Int, y: Int)Int
// 函数的类型为: (Int, Int) => Int

scala> def add3(x: Int, y: Int, z: Int): Int = { x + y + z }
add3: (x: Int, y: Int, z: Int)Int
// 函数的类型为: (Int, Int, Int) => Int

scala> def add4(x: Int, y: Int, z: Int): (Int, Int) = { (x + y, y + z) }
add4: (x: Int, y: Int, z: Int)(Int, Int)
// 函数的类型为: (Int, Int, Int) => (Int, Int)
```

函数类型：(输入参数类型列表) => (输出参数类型列表)

只有一个参数时，小括号可省略；函数体中只有1行语句时，大括号可以省略；

把函数定义中的类型声明部分去除，剩下的就是函数的“值”，即函数字面量：



- 对 add1 而言函数的值为：(x) => x+1
- 对 add2 而言函数的值为：(x, y) => x+y
- 对 add3 而言函数的值为：(x, y, z) => x+y+z
- 对 add4 而言函数的值为：(x, y, z) => (x+y, y+z)

在Scala中我们这样定义变量：`val 变量名: 类型 = 值`;

我们可以用完全相同的方式定义函数：`val 函数名: 函数类型 = 函数字面量`

```
val add1: Int => Int = (x) => x+1
val add2: (Int, Int) => Int = (x, y) => x + y
val add3: (Int, Int, Int) => Int = (x, y, z) => x + y + z
val add4: (Int, Int, Int) => (Int, Int) = (x, y, z) => (x + y, y + z)
```

在Scala中有自动类型推断，所以可以省略变量的类型`val 变量名 = 值`。

同样函数也可以这样：`val 函数名 = 函数字面量`

```
val add1 = (x: Int) => x + 1
val add2 = (x: Int, y: Int) => x + y
val add3 = (x: Int, y: Int, z: Int) => x + y + z
val add4 = (x: Int, y: Int, z: Int) => (x + y, y + z)
```

备注：要让编译器进行自动类型推断，要告诉编译器足够的信息，所以添加了 x 的类型信息。

函数的定义：

```
val 函数名: (参数类型1, 参数类型2) => (返回类型) = 函数字面量
val 函数名 = 函数字面量
函数字面量:(参数1：类型1，参数2：类型2)  =>  函数体
val 函数名 = (参数1：类型1，参数2：类型2)  =>  函数体
```

第2节 函数与方法的区别

```
scala> def addm(x: Int, y: Int): Int = x + y
addm: (x: Int, y: Int)Int

scala> val addf = (x: Int, y: Int) => x + y
addf: (Int, Int) => Int = <function2>
```

严格的说：使用 val 定义的是函数(function)，使用 def 定义的是方法(method)。二者在语义上的区别很小，在绝大多数情况下都可以不去理会它们之间的区别，但是有时候有必要了解它们之间的不同。

Scala中的方法与函数有以下区别：

- Scala 中的方法与 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 中用 22 个特质(从 Function1 到 Function22)抽象出了函数的概念
- Scala 中用 val 语句定义函数，def 语句定义方法

```
// 下面用三种方式定义了函数，其中第二种方式最常见
val adder1: (Int, Int) => Int = (x, y) => x+y
val adder2 = (x: Int, y: Int) => x+y

// Function2是特质，不能直接new
// new Function2[Int,Int,Int]{ ... } 其实是定义并实例化一个实现了 Function2 特质的类的对象
val adder3 = new Function2[Int, Int, Int]{
  def apply(x: Int, y: Int): Int = {
    x + y
  }
}
```

- 方法不能作为单独的表达式而存在，而函数可以；
- 函数必须要有参数列表，而方法可以没有参数列表；
- 方法名是方法调用，而函数名只是代表函数对象本身；
- 在需要函数的地方，如果传递一个方法，会自动把方法转换为函数

```
// 方法不能作为单独的表达式而存在，而函数可以
scala> def addm(x: Int, y: Int): Int = x + y
addm: (x: Int, y: Int)Int

scala> val addf = (x: Int, y: Int) => x + y
addf: (Int, Int) => Int = <function2>

scala> addm
<console>:13: error: missing argument list for method addm

scala> addf
res8: (Int, Int) => Int = <function2>

// 函数必须要有参数列表，而方法可以没有参数列表
scala> def m1 = "This is lagou edu"
m1: String

// 函数必须有参数列表
scala> val f1 = () => "This is lagou edu"
f1: () => String = <function0>

// 方法名是方法调用
```

```
scala> m1
res16: String = This is lagou edu

// 函数名代表函数对象
scala> f1
res17: () => String = <function0>

// 这才代表函数调用
scala> f1()
res18: String = This is lagou edu

// 需要函数的地方，可以传递一个方法
scala> val list = (1 to 10).toList
lst: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> def double(x: Int) = x*x
double: (x: Int)Int

scala> list.map(double(_))
res20: List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

将方法转换为函数：

```
scala> def f1 = double _   //注意：方法名与下划线之间有一个空格
f1: Int => Int

scala> f1
res21: Int => Int = <function1>
```

写程序的时候是定义方法、还是定义函数?

一般情况下，不对二者做区分，认为都是函数，**更多的时候使用def定义函数。**

第3节 匿名函数与占位符

函数没有名字就是匿名函数;

匿名函数，又被称为 Lambda 表达式。 Lambda表达式的形式如下：

(参数名1：类型1, 参数名2：类型2, ...) => 函数体

```
// 定义匿名函数
scala> (x: Int) => x + 1
res0: Int => Int = <function1>

// 函数没有名字，在集成开发环境中是无法被调用的
scala> res0(10)
res1: Int = 11

scala> val list = (1 to 10).toList
lst: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

// 将匿名函数作为参数传递给另一个函数
scala> list.map((x: Int) => x + 1)
```

```
res2: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10, 11)

// x一定是Int类型，这里可以省略
scala> list.map((x) => x + 1)
res3: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10, 11)

// 只有一个参数，小括号可以省略
scala> list.map(x => x + 1)
res4: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10, 11)

// 使用占位符简化函数字面量
scala> list.map(_ + 1)
res5: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10, 11)

// 实现将List中的每个元素*2 + 1，但是出错了
scala> list.map(_ + _ + 1)
<console>:13: error: missing parameter type for expanded function ((x$1, x$2) =>
x$1.$plus(x$2).$plus(1))

// 这样是可行的
scala> list.map(2 * _ + 1)

// 通过reduce这个高阶函数，将list列表中的元素相加求和
scala> list.reduce((x,y) => x + y)
res0: Int = 55

// 使用占位符简化函数字面量
// 第一个下划线代表第一个参数，第二个下划线代表第二个参数
scala> list.reduce(_ + _)
res1: Int = 55
```

多个下划线指代多个参数，而不是单个参数的重复运用

- 第一个下划线代表第一个参数
- 第二个下划线代表第二个参数
- 第三个……，如此类推

第4节 高阶函数

高阶函数：接收一个或多个函数作为输入 或 输出一个函数。

函数的参数可以是变量，而函数又可以赋值给变量，由于函数和变量地位一样，所以函数参数也可以是函数；

常用的高阶函数：map、reduce、flatMap、foreach、filter、count ... ... (接收函数作为参数)

```
object HighFunction {
  def main(args: Array[String]): Unit = {
    //定义一个函数
    val func = (n) => "*" * n
    //接收函数作为输入
    (1 to 5).map(func(_)).foreach(println)

    //输出一个函数
    val urlBuilder= (ssl: Boolean, domainName: String) => {
      val schema = if (ssl) "https://" else "http://"
```

```
      //返回一个匿名函数
      (endPoint: String, query: String) => s"$schema$domainName/$endPoint?
$query"
    }
    val domainName = "www.lagou.com"

    def getURL: (String, String) => String = urlBuilder(true, domainName)

    val endPoint: String = "show"
    val query: String = "id=1"
    val url: String = getURL(endPoint, query)
    println(url)
  }
}
```

第5节 闭包

闭包是一种函数，一种比较特殊的函数，它和普通的函数有很大区别：

```
    // 普通的函数
    val addMore1 = (x: Int) => x + 10
    // 外部变量，也称为自由变量
    var more = 10
    // 闭包
    val addMore2 = (x: Int) => x + more

    // 调用addMore1函数
    println(addMore1(5))
    // 每次addMore2函数被调用时，都会去捕获外部的自由变量
    println(addMore2(10))
    more = 100
    println(addMore2(10))
    more = 1000
    println(addMore2(10))
```

闭包是在其上下文中引用了自由变量的函数；

闭包引用到函数外面定义的变量，定义这个函数的过程就是将这个自由变量捕获而构成的一个封闭的函数，也可理解为"把函数外部的一个自由变量关闭进来"。

何为闭包？需满足下面三个条件：

1、闭包是一个函数

2、函数必须要有返回值

3、返回值依赖声明在函数外部的一个或多个变量，用Java的话说，就是返回值和定义的全局变量有关

第6节 柯里化

函数编程中，**接收多个参数的函数都可以转化为接收单个参数的函数**，这个转化过程就叫柯里化(Currying)。

Scala中，柯里化函数的定义形式和普通函数类似，区别在于柯里化函数拥有多组参数列表，每组参数用小括号括起来。

Scala API中很多函数都是柯里化的形式。

```
// 使用普通的方式
def add1(x: Int, y: Int) = x + y

// 使用闭包的方式，将其中一个函数作为返回值
def add2(x: Int) = (y:Int) => x + y

// 使用柯里化的方式
def add(x: Int)(y: Int) = x + y

//调用柯里化函数add
scala> add(1)(2)
res1: Int = 3
//add(1)(2)实际上第一次调用使用参数x，返回一个函数类型的值，第二次使用参数y调用这个函数类型的值。
//实际上最先演变成这样的函数：def add(x: Int) = (y:Int) => x + y
//在这个函数中，接收一个x为参数，返回一个匿名函数，这个匿名函数的定义是：接收一个Int型参数y，函数体是x+y。
//调用过程如下：
scala> val result=add(1)
result: Int => Int = <function1>

scala> val sum=result(2)
sum: Int = 3

scala> sum
res0: Int = 3
```

```
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1 = foldLeft(z)(op)

def aggregate[B](z: =>B)(seqop: (B, A) => B, combop: (B, B) => B): B = foldLeft(z)(seqop)
```

```
def sortBy[B](f: A => B)(implicit ord: Ordering[B]): Repr = sorted(ord on f)
```

第7节 部分应用函数

部分应用函数（Partial Applied Function）也叫偏应用函数，与偏函数从名称上看非常接近，但二者之间却有天壤之别。

部分应用函数是指缺少部分（甚至全部）参数的函数。

如果一个函数有n个参数, 而为其提供少于n个参数, 那就得到了一个部分应用函数。

```
// 定义一个函数
def add(x:Int, y:Int, z:Int) = x+y+z

// Int不能省略
def addX = add(1, _:Int, _:Int)
addX(2,3)
addX(3,4)
```

```
def addXAndY = add(10, 100, _:Int)
addXAndY(1)

def addZ = add(_:Int, _:Int, 10)
addZ(1,2)

// 省略了全部的参数，下面两个等价。第二个更常用
def add1 = add(_: Int, _: Int, _: Int)
def add2 = add _
```

第8节 偏函数

偏函数（Partial Function）之所以“偏”，原因在于它们并不处理所有可能的输入，而只处理那些能与至少一个 case 语句匹配的输入；

在偏函数中只能使用 **case 语句**，整个函数必须用**大括号**包围。这与普通的函数字面量不同，普通的函数字面量可以使用大括号，也可以用小括号；

被包裹在**大括号**中的一组case语句是一个偏函数，是一个并非对所有输入值都有定义的函数；

Scala中的Partial Function是一个trait，其类型为PartialFunction[A,B]，表示：接收一个类型为A的参数，返回一个类型为B的结果。

```
// 1、2、3有对应的输出值，其它输入打印 Other
val pf: PartialFunction[Int, String] = {
  case 1 => "One"
  case 2 => "Two"
  case 3 => "Three"
  case _=> "Other"
}
pf(1) // 返回: One
pf(2) // 返回: Two
pf(5) // 返回: Other
```

需求：过滤List中的String类型的元素，并将Int类型的元素加1。

通过偏函数实现上述需求。

```
package cn.lagou.edu.scala.section3

object PartialFunctionDemo1 {
  def main(args: Array[String]): Unit = {
    // PartialFunction[Any, Int]: 偏函数接收的数据类型是Any，返回类型为Int
    val partialFun = new PartialFunction[Any, Int] {
      // 如果返回true，就调用 apply 构建实例对象；如果返回false，过滤String数据
      override def isDefinedAt(x: Any): Boolean = {
        println(s"x = $x")
        x.isInstanceOf[Int]
      }

      // apply构造器，对传入值+1，并返回
      override def apply(v1: Any): Int = {
        println(s"v1 = $v1")
```

```
        v1.asInstanceOf[Int] + 1
      }
    }

    val lst = List(10, "hadoop", 20, "hive", 30, "flume", 40, "sqoop")

    // 过滤字符串，对整型+1
    // collect通过执行一个并行计算（偏函数），得到一个新的数组对象
    lst.collect(partialFun).foreach(println)

    // 实际不用上面那么麻烦
    lst.collect{case x: Int => x+1}.foreach(println)
  }
}
```

第九部分 集合

主要内容：

1、Scala中的可变和不可变集合
2、集合的三大类：Seq、Set、Map
3、集合的常用算子
4、Scala与Java之间的集合转换

第1节 可变和不可变集合

根据容器中元素的组织方式和操作方式，可以分为有序和无序、可变和不可变等不同的容器类别；

不可变集合是指集合内的元素一旦初始化完成就不可再进行更改，任何对集合的改变都将生成一个新的集合；
可变集合提供了改变集合内元素的方法；

Scala同时支持可变集合和不可变集合，主要下面两个包：

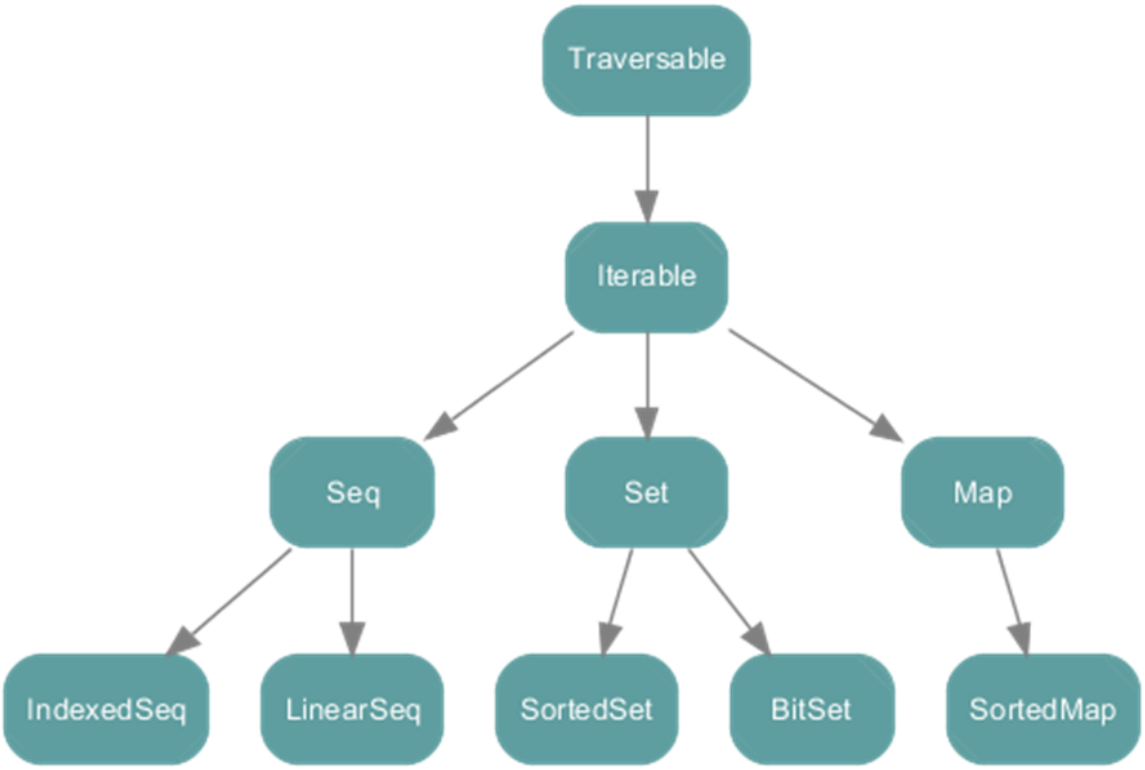
- scala.collection.mutable：定义了可变集合的特质和具体实现类
- scala.collection.immutable：定义了不可变集合的特质和具体实现类

对于几乎所有的集合类，Scala都同时提供了可变和不可变的版本。

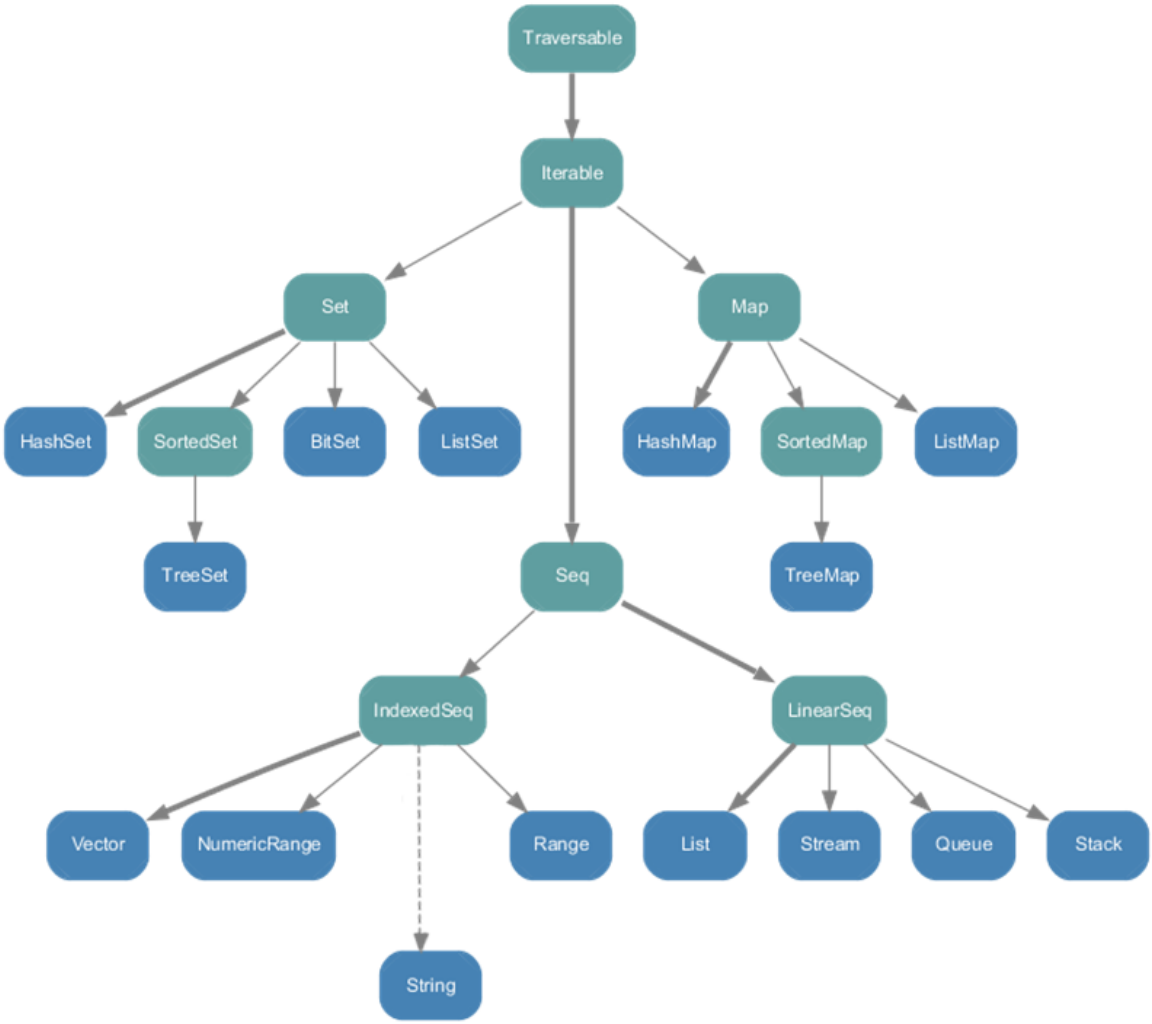
Scala优先采用不可变集合，不可变集合元素不可更改，可以安全的并发访问。

Scala集合有三大类：Seq（序列）、Set（集）、Map（映射）；

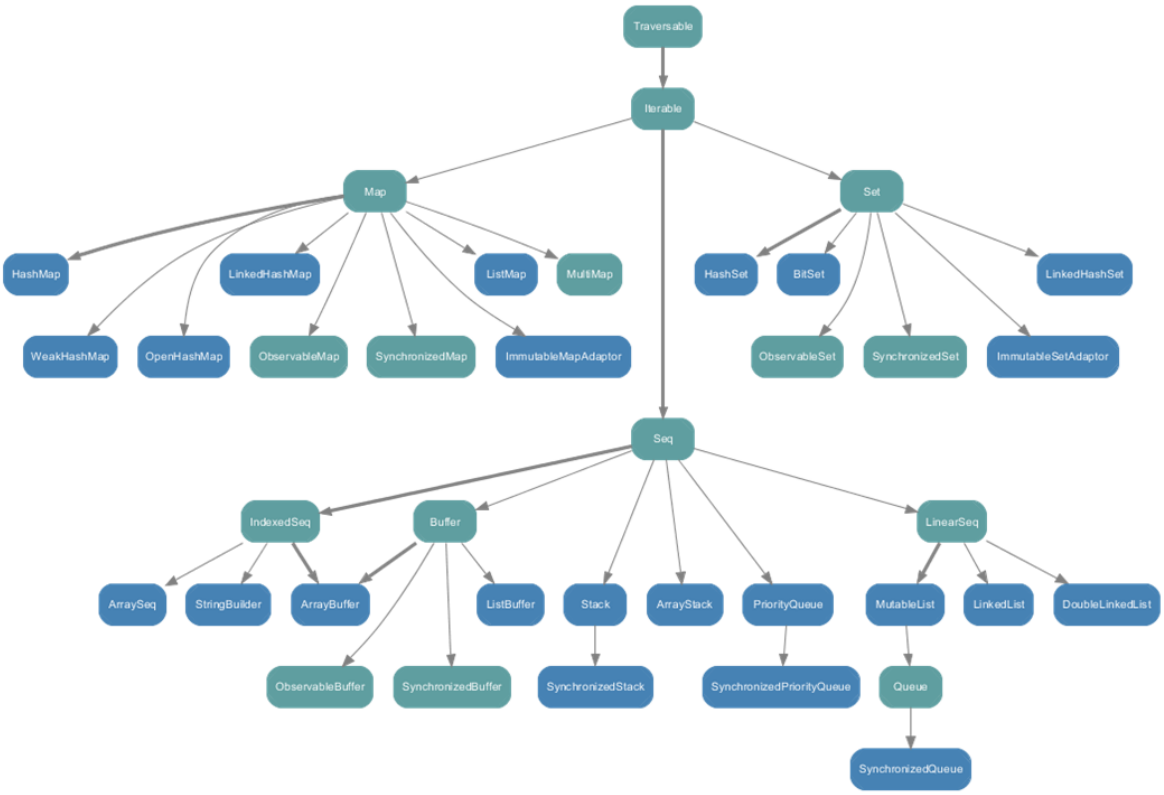
所有的集合都扩展自Iterable特质。



immutable不可变集合：



mutable可变集合：



小结：
String属于IndexedSeq
Queue队列和Stack堆这两个经典的数据结构属于LinearSeq
Map体系下有一个SortedMap，说明Scala中的Map是可以支持排序的
mutable可变集合中Seq中的Buffer下有ListBuffer，它相当于可变的List列表；
List列表属于Seq中的LinearSeq

第2节 Seq

Seq代表按照一定顺序排列的元素序列；

该序列是一种特别的可迭代集合，包含可重复的元素；

元素的顺序是确定的，每个元素对应一个索引值；

Seq提供了两个重要的子特质：

- IndexedSeq：提供了快速随机访问元素的功能，它通过索引来查找和定位的
- LinearSeq：提供了访问head、tail的功能，它是线型的，有头部和尾部的概念，通过遍历来查找。

2.1 List

List代表元素顺序固定的不可变的链表，它是Seq的子类，在Scala编程中经常使用。

List是函数式编程语言中典型的数据结构，与数组类似，可索引、存放类型相同的元素。

List一旦被定义，其值就不能改变。

List列表有头部和尾部的概念，可以分别使用head和tail方法来获取：

- head返回的是列表第一个元素的值
- tail返回的是除第一个元素外的其它元素构成的新列表

这体现出列表具有递归的链表结构。

Scala定义了一个空列表对象Nil，定义为List[Nothing]

借助 Nil 可将多个元素用操作符 :: 添加到列表头部，常用来初始化列表；

操作符 ::: 用于拼接两个列表；

```
// 构建List
val lst1 = 1 :: 2:: 3:: 4 :: Nil

// :: 是右结合的
val lst2 = 1 :: (2:: (3:: (4 :: Nil)))

// 使用 ::: 拼接List
val lst3 = lst1 ::: lst2

// 使用 head、tail获取头尾
lst3.head //返回第一个元素
lst3.tail //返回除第一个元素外的其它元素构成的新列表
lst3.init //返回除最后一个元素外的其它元素构成的新列表
lst3.last //返回最后一个元素
```

列表递归的结构，便于编写递归的算法：

```
package cn.lagou.edu.scala.section3

import scala.util.Random

object test1 {
  def main(args: Array[String]): Unit = {
    val random = new Random(100)
    val lst = List.fill(100)(random.nextInt(200))
    println(lst)

    println(s"sum(lst) = ${sum(lst)}; sum(lst) = ${lst.sum}")

    println(quickSort(lst))
  }

  // 快排
  def quickSort(lst: List[Int]): List[Int] = {
    lst match {
      case Nil => Nil
      case head :: tail =>
        val (less, greater) = tail.partition(_ < head)
        quickSort(less) ::: head :: quickSort(greater)
    }
  }
}
```

2.2 Queue

队列Queue是一个先进先出的结构。

队列是一个有序列表，在底层可以用数组或链表来实现。

先进先出的原则，就是先存入的数据，要先取出，后存入的数据后取出。

在Scala中，有scala.collection.mutable.Queue和scala.collection.immutable.Queue，一般来说，我们使用的是scala.collection.mutable.Queue

```
//创建可变的队列
val queue1 = new collection.mutable.Queue[Int]()
println(queue1)

//队列当中添加元素
queue1 += 1
//队列当中添加List
queue1 ++= List(2,3,4)
println(queue1)

// 按照进入队列顺序，删除队列当中的元素（弹出队列）
// 返回队列中的第一个元素，并从队列中删除该元素。
val dequeue = queue1.dequeue()
println(dequeue)
println(queue1)

// 向队列当中加入元素（入队列操作）
// 元素入队列
queue1.enqueue(5,6,7)
println(queue1)

//获取第一个、最后一个元素
println(queue1.head)
println(queue1.last)
```

第3节 Set

Set(集合)是没有重复元素的对象集合，Set中的元素是唯一的；

Set分为可变的和不可变的集合；

默认情况下，使用的是不可变集合(引用 scala.collection.immutable.Set)；

使用可变集合，需要引用 scala.collection.mutable.Set 包；

```
object SetDemo {
  def main(args: Array[String]): Unit = {
    // 判断元素是否存在
    val set = Set(1, 2, 3, 4, 5, 6, 7)
    println(set.exists(_ % 2 == 0))
    // 删除元素
    set.drop(1)
```

```
    // 引入可变的Set
    import scala.collection.mutable.Set
    val mutableSet = Set(4, 5, 6)

    // 增加元素、删除元素；执行成功返回true，否则返回false
    mutableSet.add(7)
    println(mutableSet)
    mutableSet.remove(7)
    println(mutableSet)

    // 使用 += / -= 增加、删除元素，表达更简洁
    mutableSet += 5
    mutableSet -= 2

    // 集合典型操作交、并、差
    //交集（&、intersect）
    println(Set(1, 2, 3) & Set(2, 3, 4))
    println(Set(1, 2, 3).intersect(Set(2, 3, 4)))
    println(Set(1, 2, 3) intersect (Set(2, 3, 4)))

    //并集(++、|、union)
    println(Set(1, 2, 3) ++ Set(2, 3, 4))
    println(Set(1, 2, 3) | Set(2, 3, 4))
    println(Set(1, 2, 3).union(Set(2, 3, 4)))

    //差集(--、&~、diff)
    //返回:包含本集合中不包含在给定集合中的元素的集合
    println(Set(1, 2, 3) -- Set(2, 3, 4))
    println(Set(1, 2, 3) &~ Set(2, 3, 4))
    println(Set(1, 2, 3).diff(Set(2, 3, 4)))
  }
}
```

第4节 Map

Map(映射)是一系列键值对的容器；Scala 提供了可变的和不可变的两种版本的Map，

分别定义在包 scala.collection.mutable 和 scala.collection.immutable 里；

默认情况下，Scala中使用不可变的 Map；

如果要使用可变Map，必须导入scala.collection.mutable.Map；

在Map中，键的值是唯一的，可以根据键来对值进行快速的检索。

```
// 可使用两种方式定义Map
// Map缺省是不可变的，值不能更改
val a = Map("a" -> 1, "b" -> 2, "c" -> 3)
val a = Map(("a", 1),("b", 2),("c", 3))
a.keys
a.values

// 获取Map中的值：
a("a")

// 访问不存在的key时，会抛出异常。Java.util.NoSuchElementException: key not found: x
```

```
a("x")

// 使用get方法，返回一个Option对象，要么是Some（键对应的值），要么是None
a.get("a")

// 获取键对应的值，如果键不存在返回给定的值（这里是0）
a.getOrElse("a", 0)

// 更新Map中的值(要使用可变的Map)
val b = scala.collection.mutable.Map("a" -> 1, "b" -> 2, "c" -> 3)
b("a") = 2

// 增加了一个新元素
b("d") = 4

// 用 + 添加新的元素；用 - 删除元素
b += ("e" -> 1, "f" -> 2)
b -= "a"

// 增加元素
val b = Map("a" -> 1, "b" -> 2, "c" -> 3)
val c = b + ("a" -> 10, "b" -> 20)

// 通过包含键值对的二元组创建Map集合
val a = Map(("a", 1),("b", 2),("c", 3))

// 逐个访问 value
for(v <- a.values) println(v)

// key 和 value 做了交换
val b = for((k,v) <- a) yield (v,k)

// 下面才是具有scala风格的写法，推荐
a.map(x=>(x._2, x._1))

// 拉链操作创建Map
val a = Array(1,2,3)
val b = Array("a","b","c")
//c: Array[(Int, String)]
val c = a.zip(b)
//c: scala.collection.immutable.Map[Int,String]
val c = a.zip(b).toMap
```

第5节 集合常用算子

5.1 map、foreach & mapValues

集合对象都有 foreach、map 算子。

两个算子的共同点在于：都是用于遍历集合对象，并对每一项执行指定的方法；

两个算子的差异点在于：

foreach无返回值（准确说返回void），用于遍历集合

map返回集合对象，用于将一个集合转换成另一个集合

```
// 使用 foreach 打印集合元素
val numlist = (1 to 10).toList
numlist.foreach(elem=>print(elem+" "))
numlist.foreach(print _)
numlist.foreach(print)

// 使用 map 对集合进行转换
numlist.map(_ > 2)
numlist.map(_ * 2)
```

操作 Map集合时，mapValues用于遍历value，是map操作的一种简化形式；

```
// Range(20, 0, -2)用给定的步长值设定一个范围，从开始到结束(不包含)。
//Map(20 -> 0,18 -> 1,16 -> 2,14 -> 3,12 -> 4,10 -> 5,8 -> 6,6 -> 7,4 -> 8,2 ->
9)
val map = Range(20, 0, -2).zipWithIndex.toMap

// 将map集合中的value值+100
map.map(elem => (elem._1, elem._2 + 100))
map.map{case (k,v) => (k, v+100)}
// mapValues的表达最简洁
map.mapValues(_+100)
```

5.2 flatten & flatMap

flatten的作用是把嵌套的结构展开，把结果放到一个集合中；

在 flatMap 中传入一个函数，该函数对每个输入都返回一个集合（而不是一个元素），最后把生成的多个集合“拍扁”成为一个集合；

```
scala> val lst1 = List(List(1,2), List(3,4))
lst1: List[List[Int]] = List(List(1, 2), List(3, 4))

scala> lst1.flatten
res5: List[Int] = List(1, 2, 3, 4)

// flatten 把一个字符串的集合展开为一个字符集合，因为字符串本身就是字符的集合
scala> val lst4 = List("Java", "hadoop")
lst4: List[String] = List(Java, hadoop)

scala> lst4.flatten
res8: List[Char] = List(J, a, v, a, h, a, d, o, o, p)

// flatten 有效的处理 Some 和 None 组成的集合。它可以展开Some元素形成一个新的集合，同时去掉
None元素
scala> val x = Array(Some(1), None, Some(3), None)
x: Array[Option[Int]] = Array(Some(1), None, Some(3), None)

// 方法很多，flatten最简单
scala> x.flatten
res9: Array[Int] = Array(1, 3)

scala> x.collect{case Some(i) => i}
```

```
res10: Array[Int] = Array(1, 3)

scala> x.filter(!_.isEmpty).map(_.get)
res11: Array[Int] = Array(1, 3)
```

```
// 下面两条语句等价
val lst = List(List(1,2,5,6),List(3,4))

// 将 lst 中每个元素乘2，最后作为一个集合返回
// 此时 flatMap = flatten + map
//List(1,2,5,6,3,4)
lst.flatten.map(_*2)
lst.flatMap((x: List[Int]) => x.map(_*2))
lst.flatMap(_.map(_*2))

// 将字符串数组按空格切分，转换为单词数组
val lines = Array("Apache Spark has an advanced DAG execution engine",
"Spark offers over 80 high-level operators")
// 下面两条语句效果等价
//map算子产生的结果：Array(Array(Apache, Spark, has, an, advanced, DAG, execution,
engine), Array(Spark, offers, over, 80, high-level, operators))
// flatten算子产生的结果：Array(Apache, Spark, has, an, advanced, DAG, execution,
engine, Spark, offers, over, 80, high-level, operators)
lines.map(_.split(" ")).flatten
// 此时 flatMap = map + flatten
lines.flatMap(_.split(" "))
```

备注：flatMap = flatten + map 或 flatMap = map + flatten

5.3 collect

collect通过执行一个并行计算（偏函数），得到一个新的数组对象

```
object CollectDemo {
  //通过下面的偏函数，把chars数组的小写a转换为大写的A
  val fun: PartialFunction[Char, Char] = {
    case 'a' => 'A'
    case x => x
  }

  def main(args: Array[String]): Unit = {
    val chars = Array('a', 'b', 'c')
    val newchars = chars.collect(fun)
    println("newchars:" + newchars.mkString(","))
  }
}
```

5.4 reduce

reduce可以对集合当中的元素进行归约操作；

还有 reduceLeft 和 reduceRight ， reduceLeft 从左向右归约，reduceRight 从右向左归约；

```
val lst1 = (1 to 10).toList
lst1.reduce(_+_)

// 为什么这里能出现两个占位符？
lst1.reduce(_+_)

// 我们说过一个占位符代表一个参数，那么两个占位符就代表两个参数。根据这个思路改写等价的语句
// x类似于buffer，缓存每次操作的数据；y每次操作传递新的集合元素
lst1.reduce((x, y) => x + y)


// 利用reduce操作，查找 lst1 中的最大值
lst1.reduce((x,y) => if (x>y) x else y)

// reduceLeft、reduceRight
lst1.reduceLeft((x,y) => if (x>y) x else y)
lst1.reduceRight((x,y) => if (x>y) x else y)
```

5.5 sorted sortwith & sortby

Scala中对于集合的排序有三种方法：sorted、sortBy、sortWith

```
object SortDemo {
  def main(args: Array[String]): Unit = {
    val list = List(1, 9, 3, 8, 5, 6)
    //sorted方法对一个集合进行自然排序
    //sorted源码：def sorted[B >: A](implicit ord: Ordering[B]): Repr
    //源码中有两点值得注意的地方：
    // 1.sorted方法中有个隐式参数ord: Ordering。
    // 2.sorted方法真正排序的逻辑是调用的java.util.Arrays.sort
    val numSort: List[Int] = list.sorted
    println(numSort)
    //sortBy源码：def sortBy[B](f: A => B)(implicit ord: Ordering[B]): Repr =
sorted(ord on f)
    //sortBy最后调用的sorted方法
    println(list.sortBy(x => x).reverse)

    //sortWith源码：def sortWith(lt: (A, A) => Boolean): Repr = sorted(Ordering
fromLessThan lt)
    print(list.sortWith(_ > _))
  }
}
```

第6节 与Java集合的转换

使用 scala.collection.JavaConverters 与Java集合交互。它有一系列的隐式转换，添加了asJava和asScala的转换方法。

```
import scala.collection.JavaConverters._

val list: Java.util.List[Int] = List(1,2,3,4).asJava
val buffer: scala.collection.mutable.Buffer[Int] = list.asScala
```

第十部分 隐式机制

主要内容：
1、隐式转换
2、隐式转换函数
3、隐式参数和隐式值

第1节 隐式转换

隐式转换和隐式参数是Scala中两个非常强大的功能，利用隐式转换和隐式参数，可以提供类库，对类库的使用者隐匿掉具体的细节。

Scala会根据隐式转换函数的签名，在程序中使用到隐式转换函数接收的参数类型定义的对象时，会自动将其传入隐式转换函数，转换为另外一种类型的对象并返回，这就是“**隐式转换**”。

- 首先得有一个隐式转换函数
- 使用到隐式转换函数接收的参数类型定义的对象
- Scala自动传入隐式转换函数，并完成对象的类型转换

隐式转换需要使用implicit关键字。

使用Scala的隐式转换有一定的限制：

- implicit关键字只能用来修饰方法、变量、参数
- 隐式转换的函数只在当前范围内才有效。如果隐式转换不在当前范围内定义，那么必须通过import语句将其导入

Spark源码中有大量的隐式转换和隐式参数，因此必须掌握隐式机制。

第2节 隐式转换函数

Scala的隐式转换最核心的就是定义隐式转换函数，即implicit conversion function。

定义的隐式转换函数，只要在编写的程序内引入，就会被Scala自动使用。

隐式转换函数由Scala自动调用，通常建议将隐式转换函数的名称命名为“one2one”的形式。

示例1：下面代码中定义了一个隐式函数

```
class Num {}

class RichNum(num: Num) {
  def rich(): Unit = {
    println("Hello Implicit!")
  }
}
```

```
object ImplicitDemo {
  // 定义一个名称为num2RichNum的隐式函数
  implicit def num2RichNum(num: Num): RichNum = {
    new RichNum(num)
  }

  def main(args: Array[String]): Unit = {
    val num = new Num
    // num对象并没有rich方法，编译器会查找当前范围内是否有可转换的函数
    // 如果没有则编译失败，如果有则会调用。
    num.rich()
  }
}
```

示例2：导入隐式函数

```
package test.implicitdemo

object Int2String {
  implicit def int2String(num: Int):String = num.toString
}
```

下面代码中调用了String类型的length方法，Int类型本身没有length方法，但是在可用范围内定义了可以把Int转换为String的隐式函数int2String，因此函数编译通过并运行出正确的结果。

此示例中隐式函数的定义必须定义在使用之前，否则编译报错。

```
import test.implicitdemo.Int2String._

object ImplicitTest {
  def main(args: Array[String]): Unit = {
    println(20.length)
  }
}
```

通过import test.implicitdemo.Int2String._，将Int2StringTest内部的成员导入到相应的作用域内，否则无法调用隐式函数。

要实现隐式转换，只要在程序可见的范围内定义隐式转换函数即可，Scala会自动使用隐式转换函数。隐式转换函数与普通函数的语法区别就是，要以implicit开头，而且最好要定义函数返回类型。

隐式转换案例：特殊售票窗口（只接受特殊人群买票，比如学生、老人等），其他人不能在特殊售票窗口买票。

```
class SpecialPerson(var name: String)

class Older(var name: String)

class Student(var name: String)

class Worker(var name: String)

object ImplicitDemoTwo {
  def buySpecialTickWindow(person: SpecialPerson): Unit = {
    if (peron != null) {
```

```
      println(person.name + "购买了一张特殊票！")
    } else {
      println("你不是特殊人群，不能在此买票！")
    }
  }

  //隐式转换函数
  //注意：any参数的类型是Any
  implicit def any2SpecialPerson(any: Any): SpecialPerson = {
    any match {
      case any: Older => new SpecialPerson(any.asInstanceOf[Older].name)
      case any: Student => new SpecialPerson(any.asInstanceOf[Student].name)
      case _ => null
    }
  }

  def main(args: Array[String]): Unit = {
    val stu = new Student("jacky")
    val older = new Older("old man")
    val worker = new Worker("tom")

    ImplicitDemoTwo.buySpecialTickWindow(stu)
    ImplicitDemoTwo.buySpecialTickWindow(older)
    ImplicitDemoTwo.buySpecialTickWindow(worker)
  }
}
```

第3节 隐式参数和隐式值

在函数定义的时候，支持在**最后一组参数**中使用 `implicit`，表明这是一组隐式参数。在调用该函数的时候，可以不用传递隐式参数，而编译器会自动寻找一个`implicit`标记过的合适的值作为参数。

Scala编译器会在两个范围内查找：

- 当前作用域内可见的val或var定义隐式变量
- 隐式参数类型的伴生对象内隐式值

```
object Doubly {
  //在print函数中定义一个隐式参数fmt
  def print(num: Double)(implicit fmt: String): Unit = {
    println(fmt format (num))
  }
  def main(args: Array[String]): Unit = {
    //此时调用print函数需要为第二个隐式参数赋值
    print(3.12)("%.1f")

    //定义一个隐式变量
    implicit val printFmt="%.3f"
    //当调用print函数时没有给第二个隐式参数赋值，
    //那么Scala会在当前作用域内寻找可见的val或var定义的隐式变量，一旦找到就会应用
    print(3.12)
  }
}
```

第十一部分 扩展部分

```
主要内容：
    1、类型参数
        泛型类、泛型函数、协变和逆变
    2、Akka
```

第1节 类型参数

Scala的类型参数与Java的泛型是一样的，可以在集合、类、函数中定义类型参数，从而保证程序更好的健壮性。

1.1 泛型类

泛型类，顾名思义，其实就是在类的声明中定义一些泛型类型，然后在类内部的字段或者方法，就可以使用这些泛型类型。

使用泛型类，通常是需要对类中的某些成员，比如某些字段和方法中的参数或变量进行统一的类型限制，这样可以保证程序更好的健壮性和稳定性。

如果不使用泛型进行统一的类型限制，那么在后期程序运行过程中难免会出现问题，比如传入了不希望的类型导致程序出问题。

在使用泛型类的时候，比如创建泛型类的对象，只需将类型参数替换为实际的类型即可。

Scala自动推断泛型类型特性：直接给使用泛型类型的字段赋值时，Scala会自动进行类型推断。

泛型类的定义如下：

```
//定义一个泛型类
class Stack[T1, T2, T3](name: T1) {
  var age: T2 = _
  var address: T3 = _

  def getInfo: Unit = {
    println(s"$name,$age,$address")
  }
}
```

使用上述的泛型类，只需要使用具体的类型代替类型参数即可。

```
object GenericityDemo {
  def main(args: Array[String]): Unit = {
    //创建泛型类对象
    val stack = new Stack[String, Int, String]("lisi")
    stack.age = 20
    stack.address = "北京"

    stack.getInfo
  }
}
```

1.2 泛型函数

泛型函数，与泛型类类似，可以给某个函数在声明时指定泛型类型，然后在函数体内，多个变量或者返回值之间，就可以使用泛型类型进行声明，从而对某个特殊的变量，或者多个变量，进行强制性的类型限制。

与泛型类一样，你可以通过给使用了泛型类型的变量传递值来让Scala自动推断泛型的实际类型，也可以在调用函数时，手动指定泛型类型。

案例：卡片售卖机，可以指定卡片的内容，内容可以是String类型或Int类型

```
object GenericityFunction {
  def getCard[T](content: T) = {
    content match {
      case content: Int => s"card:$content is Int "
      case content: String => s"card:$content is String"
      case _ => s"card:$content"
    }
  }

  def main(args: Array[String]): Unit = {
    println(getCard[String]("hello"))
    println(getCard(1001))
  }
}
```

1.3 协变和逆变

Scala的协变和逆变是非常有特色的，完全解决了Java中的泛型的一大缺憾！

举例来说，Java中，如果有Professional是Master的子类，那么Card[Professionnal]是不是Card[Master]的子类？答案是：不是。因此对于开发程序造成了很多的麻烦。

而Scala中，只要灵活使用协变和逆变，就可以解决Java泛型的问题。

协变定义形式如：trait List[+T] {}

当类型S是类型A的子类型时，则List[S]也可以认为是List[A}的子类型，即List[S]可以泛化为List[A]，也就是被参数化，类型的泛化方向与参数类型的方向是一致的，所以称为协变（covariance）。

逆变定义形式如：trait List[-T] {}
当类型S是类型A的子类型，则Queue[A]反过来可以认为是Queue[S}的子类型，也就是被参数化类型的泛化方向与参数类型的方向是相反的，所以称为逆变（contravariance）。

```
小结：
    如果A是B的子类，那么在协变中，List[A]就是List[B]的子类； 在逆变中，List[A]就是List[B]的父类。
```

协变案例：只有大师以及大师级别以下的名片都可以进入会场

```
package lagou.cn.part11
```

```scala
//大师
class Master

//专家
class Professor extends Master

//讲师
class Teacher

//这个是协变，Professor是Master的子类，此时Card[Profesor]也是Card[Master]的子类
class Card[+T]

object CovarianceDemo {

  def enterMeet(card: Card[Master]): Unit = {
    //只有Card[Master]及其子类Card[Professor]才能进入会场。
    println("欢迎进入会场！")
  }

  def main(args: Array[String]): Unit = {
    val masterCard = new Card[Master]
    val professorCard = new Card[Professor]
    val teacharCard = new Card[Teacher]

    enterMeet(masterCard)
    enterMeet(professorCard)
    //此处就会报错
    //enterMeet(teacharCard)
  }
}
```

第2节 Akka

Akka是Java虚拟机平台上构建高并发、分布式和容错应用的工具包和运行时。

Akka用Scala语言编写，同时提供了Scala和Java的开发接口。

Akka处理并发的方法基于Actor模型，Actor之间通信的唯一机制就是消息传递。

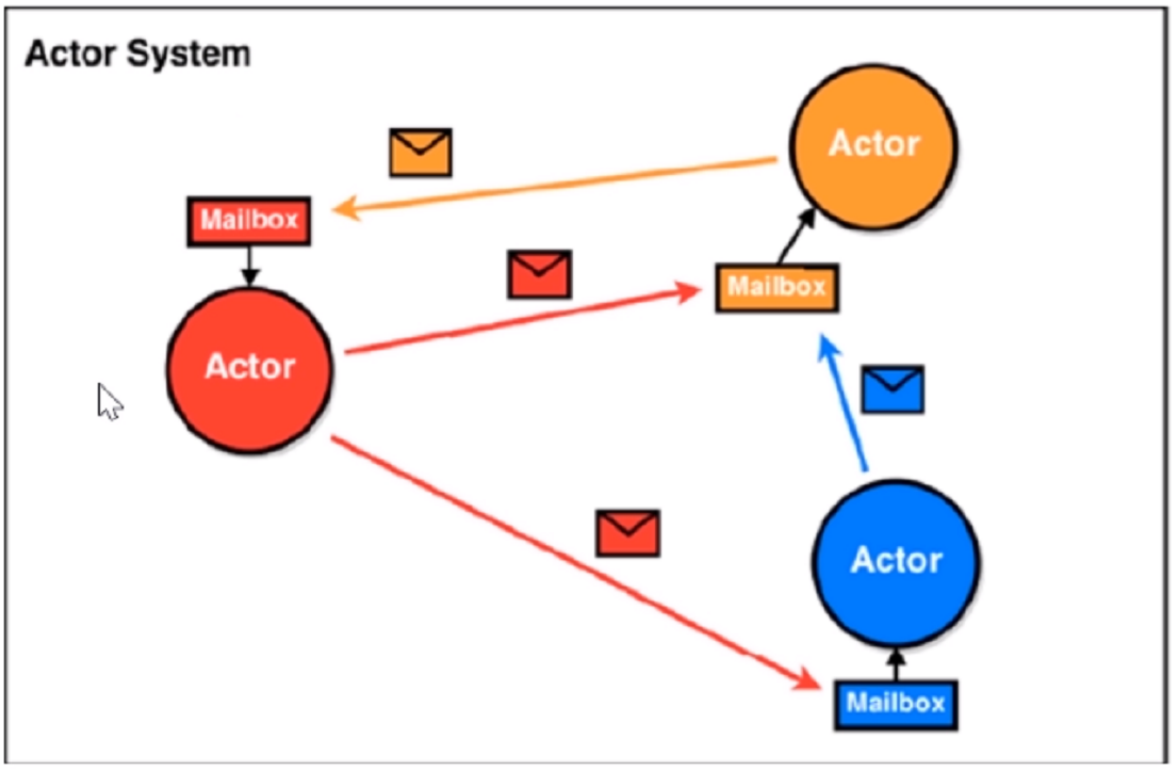
Actor

Scala的Actor类似于Java中的多线程编程。

但是不同的是，Scala的Actor提供的模型与多线程有所不同。Scala的Actor尽可能地避免锁和共享状态，从而避免多线程并发时出现资源争用的情况，进而提升多线程编程的性能。

Actor可以看作是一个个独立的实体，Actor之间可以通过交换消息的方式进行通信，每个Actor都有自己的收件箱（Mailbox）。

一个Actor收到其他Actor的信息后，根据需要作出各种相应。消息的类型可以是任意的，消息的内容也可以是任意的。



ActorSystem

在Akka中，ActorSystem是一个重量级的结构。

它需要分配多个线程，所以在实际应用中，ActorSystem通常是一个单例对象，我们可以使用这个ActorSystem创建很多Actor。

Akka案例：

创建一个maven项目，在项目的pom文件中增加如下依赖：

```
    <!-- 定义一下常量-->
    <properties>
        <encoding>UTF-8</encoding>
        <scala.version>2.12.3</scala.version>
        <scala.compat.version>2.11</scala.compat.version>
        <akka.version>2.4.17</akka.version>
    </properties>

    <dependencies>
        <!-- 添加akka的actor依赖 -->
        <dependency>
            <groupId>org.scala-lang</groupId>
            <artifactId>scala-actors</artifactId>
            <version>2.11.8</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/com.typesafe.akka/akka-actor -->
        <dependency>
            <groupId>com.typesafe.akka</groupId>
            <artifactId>akka-actor_2.11</artifactId>
            <version>2.3.16</version>
        </dependency>
```

```
        <!-- 添加akka的actor依赖 -->
        <dependency>
            <groupId>com.typesafe.akka</groupId>
            <artifactId>akka-actor_${scala.compat.version}</artifactId>
            <version>${akka.version}</version>
        </dependency>

        <!-- 多进程之间的Actor通信 -->
        <!-- https://mvnrepository.com/artifact/com.typesafe.akka/akka-remote --
>
        <dependency>
            <groupId>com.typesafe.akka</groupId>
            <artifactId>akka-remote_${scala.compat.version}</artifactId>
            <version>${akka.version}</version>
        </dependency>

    </dependencies>
```

```
package cn.lagou.edu.scala.section4

import akka.actor.{Actor, ActorSystem, Props}

import scala.io.StdIn

class HelloActor extends Actor {
  // 接收消息并处理
  override def receive: Receive = {
    case "吃了吗" => println("吃过了")
    case "吃的啥" => println("北京卤煮")
    case "拜拜" => {
      //关闭自己
      context.stop(self)
      //关闭ActorSystem
      context.system.terminate()
    }
  }
}

object HelloActor {
  //创建线程池对象MyFactor，myFactory为线程池的名称
  private val MyFactory = ActorSystem("myFactory")

  // 通过MyFactory.actorOf方法来创建一个actor；
  // 第一个参数传递自定义的HelloActor类，第二个参数是给actor起个名字
  private val helloActorRef = MyFactory.actorOf(Props[HelloActor], "helloActor")

  def main(args: Array[String]): Unit = {
    var flag = true
    while (flag) {
      print("请输入想发送的消息：")
      val consoleLine: String = StdIn.readLine()
      //通过！来发送消息
      helloActorRef ! consoleLine
      if (consoleLine.equals("拜拜")) {
```

```
          flag = false
          println("程序即将结束! ")
        }

        // 休眠100毫秒
        Thread.sleep(100)
      }
    }
}
```

课程总结

- 《Scala编程》本课程共十一部分

  课程目的：使用Scala进行Spark开发、阅读Spark源码

  ```
  第一部分 Scala基础
  第二部分 控制结构和函数
  第三部分 数组和元组
  第四部分 类与对象
  第五部分 继承
  第六部分 特质
  第七部分 模式匹配和样例类
  第八部分 函数及抽象化
  第九部分 集合
  第十部分 隐式机制
  第十一部分 扩展部分
  ```

- Scala功能强大，内容很多，还有一些内容课程中没有涉及
- 这门课程作为大家学习Scala的起点，而非终点