Scala

第一章 基本数据类型和运算符

1. 数据类型

类型支持

Scala 拥有下表所示的数据类型，其中 Byte、Short、Int、Long 和 Char 类型统称为整数类型，整数类型加上 Float 和 Double 统称为数值类型。Scala 数值类型的取值范围和 Java 对应类型的取值范围相同。

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，等同于 Java 中的 void。用作不返回任何结果的方法的结果类型。Unit 只有一个实例值，写成 ()。
Null	null 或空引用
Nothing	Nothing 类型在 Scala 的类层级的最低端；它是任何其他类型的子类型。
Any	Any 是所有其他类的超类
AnyRef	AnyRef 类是 Scala 里所有引用类 (reference class) 的基类

定义变量

Scala 的变量分为两种，val 和 var，其区别如下：

- **val**： 类似于 Java 中的 final 变量，一旦初始化就不能被重新赋值；
- **var**： 类似于 Java 中的非 final 变量，在整个声明周期内 var 可以被重新赋值；

```
scala> val a=1
a: Int = 1

scala> a=2
<console>:8: error: reassignment to val // 不允许重新赋值

scala> var b=1
b: Int = 1

scala> b=2
b: Int = 2
```

类型推断

在上面的演示中，并没有声明 a 是 Int 类型，但是程序还是把 a 当做 Int 类型，这就是 Scala 的类型推断。在大多数情况下，你都无需指明变量的类型，程序会自动进行推断。如果你想显式的声明类型，可以在变量后面指定，如下：

```
scala>   val c:String="hello scala"
c: String = hello scala
```

Scala解释器

在 scala 命令行中，如果没有对输入的值指定赋值的变量，则输入的值默认会赋值给 `resX` (其中 X 是一个从 0 开始递增的整数)，`res` 是 result 的缩写，这个变量可以在后面的语句中进行引用。

```
scala> 5
res0: Int = 5

scala> res0*6
res1: Int = 30

scala> println(res1)
30
```

2. 字面量

Scala 和 Java 字面量在使用上很多相似，比如都使用 F 或 f 表示浮点型，都使用 L 或 l 表示 Long 类型。下文主要介绍两者差异部分。

```
scala> 1.2
res0: Double = 1.2

scala> 1.2f
res1: Float = 1.2

scala> 1.4F
res2: Float = 1.4

scala> 1
res3: Int = 1

scala> 1l
res4: Long = 1

```

```
scala> 1L
res5: Long = 1
```

整数字面量

Scala 支持 10 进制和 16 进制，但不支持八进制字面量和以 0 开头的整数字面量。

```
scala> 012
<console>:1: error: Decimal integer literals may not have a leading zero. (Octal syntax is obsolete.)
```

字符串字面量

字符字面量

字符字面量由一对单引号和中间的任意 Unicode 字符组成。你可以显式的给出原字符、也可以使用字符的 Unicode 码来表示，还可以包含特殊的转义字符。

```
scala> '\u0041'
res0: Char = A

scala> 'a'
res1: Char = a

scala> '\n'
res2: Char =
```

字符串字面量

字符串字面量由双引号包起来的字符组成。

```
scala> "hello world"
res3: String = hello world
```

原生字符串

Scala 提供了 `""" ... """` 语法，通过三个双引号来表示原生字符串和多行字符串，使用该种方式，原生字符串中的特殊字符不会被转义。

```
scala> "hello \tool"
res4: String = hello      ool

scala> """hello \tool"""
res5: String = hello \tool

scala> """hello
     | world"""
res6: String =
hello
world
```

符号字面量

符号字面量写法为： `'标识符` ，这里 标识符可以是任何字母或数字的组合。符号字面量会被映射成 `scala.Symbol` 的实例，如:符号字面量 `'x` 会被编译器翻译为 `scala.Symbol("x")` 。符号字面量可选方法很少，只能通过 `.name` 获取其名称。

注意：具有相同 `name` 的符号字面量一定指向同一个 Symbol 对象，不同 `name` 的符号字面量一定指向不同的 Symbol 对象。

```
scala> val sym = 'ID008
sym: Symbol = 'ID008

scala> sym.name
res12: String = ID008
```

插值表达式

Scala 支持插值表达式。

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

scala> println(s"My name is $name,I'm ${2*9}.")
My name is xiaoming,I'm 18.
```

3. 运算符

Scala 和其他语言一样，支持大多数的操作运算符：

- 算术运算符（+，-，*，/，%）
- 关系运算符（==，!=，>，<，>=，<=）
- 逻辑运算符 (&&，||，!，&，|)
- 位运算符 (~，&，|，^，<<，>>，>>>)
- 赋值运算符 (=，+=，-=，*=，/=，%=，<<=，>>=，&=，^=，|=)

以上操作符的基本使用与 Java 类似，下文主要介绍差异部分和注意事项。

运算符即方法

Scala 的面向对象比 Java 更加纯粹，在 Scala 中一切都是对象。所以对于 `1+2`,实际上是调用了 Int 类中名为 `+` 的方法，所以 1+2,也可以写成 `1.+(2)` 。

```
scala> 1+2
res14: Int = 3

scala> 1.+(2)
res15: Int = 3
```

Int 类中包含了多个重载的 `+` 方法，用于分别接收不同类型的参数。

逻辑运算符

和其他语言一样，在 Scala 中 `&&`， `||` 的执行是短路的，即如果左边的表达式能确定整个结果，右边的表达式就不会被执行，这满足大多数使用场景。但是如果你需要在无论什么情况下，都执行右边的表达式，则可以使用 `&` 或 `|` 代替。

赋值运算符

在 Scala 中没有 Java 中的 `++` 和 `--` 运算符，如果你想要实现类似的操作，只能使用 `+=1`，或者 `-=1`。

```
scala> var a=1
a: Int = 1

scala> a+=1

scala> a
res8: Int = 2

scala> a-=1

scala> a
res10: Int = 1
```

对象相等性

如果想要判断两个对象是否相等，可以使用 `==` 和 `!=`,这两个操作符可以用于所有的对象，包括 null。

```
scala> 1==2
res2: Boolean = false

scala> List(1,2,3)==List(1,2,3)
res3: Boolean = true

scala> 1==1.0
res4: Boolean = true

scala> List(1,2,3)==null
res5: Boolean = false

scala> null==null
res6: Boolean = true
```

第二章 流程控制语句

条件表达式if

Scala 中的 if/else 语法结构与 Java 中的一样，唯一不同的是，Scala 中的 if 表达式是有返回值的。

```
object ScalaApp extends App {

  val x = "scala"
  val result = if (x.length == 5) "true" else "false"
  print(result)

}
```

在 Java 中，每行语句都需要使用 `;` 表示结束，但是在 Scala 中并不需要。除非你在单行语句中写了多行代码。

块表达式

在 Scala 中，可以使用 `{}` 块包含一系列表达式，块中最后一个表达式的值就是块的值。

```
object ScalaApp extends App {

   val result = {
      val a = 1 + 1; val b = 2 + 2; a + b
   }
   print(result)
}

// 输出： 6
```

如果块中的最后一个表达式没有返回值，则块的返回值是 Unit 类型。

```
scala> val result ={ val a = 1 + 1; val b = 2 + 2 }
result: Unit = ()
```

循环表达式while

Scala 和大多数语言一样，支持 `while` 和 `do ... while` 表达式。

```
 object ScalaApp extends App {

    var n = 0

    while (n < 10) {
       n += 1
       println(n)
    }

    // 循环至少要执行一次
    do {
       println(n)
    } while (n > 10)
 }
```

循环表达式for

for 循环的基本使用如下：

```
object ScalaApp extends App {

   // 1.基本使用  输出[1,9)
   for (n <- 1 until 10) {print(n)}

   // 2.使用多个表达式生成器  输出: 11 12 13 21 22 23 31 32 33
   for (i <- 1 to 3; j <- 1 to 3) print(f"${10 * i + j}%3d")

   // 3.使用带条件的表达式生成器  输出: 12 13 21 23 31 32
   for (i <- 1 to 3; j <- 1 to 3 if i != j) print(f"${10 * i + j}%3d")

}
```

除了基本使用外，还可以使用 `yield` 关键字从 for 循环中产生 Vector，这称为 for 推导式。

```
scala> for (i <- 1 to 10) yield i * 6
res1: scala.collection.immutable.IndexedSeq[Int] = Vector(6, 12, 18, 24, 30, 36, 42, 48, 54, 60)
```

异常处理try

和 Java 中一样，支持 `try...catch...finally` 语句。

```
import java.io.{FileNotFoundException, FileReader}

object ScalaApp extends App {

   try {
      val reader = new FileReader("wordCount.txt")
   } catch {
      case ex: FileNotFoundException =>
         ex.printStackTrace()
         println("没有找到对应的文件!")
   } finally {
      println("finally 语句一定会被执行！")
   }
}
```

这里需要注意的是因为 finally 语句一定会被执行，所以不要在该语句中返回值，否则返回值会被作为整个 try 语句的返回值，如下：

```
scala> def g():Int = try return 1 finally   return   2
g: ()Int

// 方法 g() 总会返回 2
scala> g()
res3: Int = 2
```

条件选择表达式match

match 类似于 java 中的 switch 语句。

```
object ScalaApp extends App {

   val elements = Array("A", "B", "C", "D", "E")

   for (elem <- elements) {
      elem match {
         case "A" => println(10)
         case "B" => println(20)
         case "C" => println(30)
         case _ => println(50)
      }
   }
}

```

但是与 Java 中的 switch 有以下三点不同：

- Scala 中的 case 语句支持任何类型；而 Java 中 case 语句仅支持整型、枚举和字符串常量；
- Scala 中每个分支语句后面不需要写 break，因为在 case 语句中 break 是隐含的，默认就有；
- 在 Scala 中 match 语句是有返回值的，而 Java 中 switch 语句是没有返回值的。如下：

```
object ScalaApp extends App {

   val elements = Array("A", "B", "C", "D", "E")

   for (elem <- elements) {
      val score = elem match {
         case "A" => 10
         case "B" => 20
         case "C" => 30
         case _ => 50
      }
      print(elem + ":" + score + ";")
   }
}
// 输出： A:10;B:20;C:30;D:50;E:50;
```

没有break和continue

额外注意一下：Scala 中并不支持 Java 中的 break 和 continue 关键字。

输入与输出

在 Scala 中可以使用 print、println、printf 打印输出，这与 Java 中是一样的。如果需要从控制台中获取输入，则可以使用 `StdIn` 中定义的各种方法。

```
val name = StdIn.readLine("Your name: ")
print("Your age: ")
val age = StdIn.readInt()
println(s"Hello, ${name}! Next year, you will be ${age + 1}.")
```

第三章 函数和闭包

1. 函数

函数与方法

Scala 中函数与方法的区别非常小，如果函数作为某个对象的成员，这样的函数被称为方法，否则就是一个正常的函数。

```
// 定义方法
def multi1(x:Int) = {x * x}
// 定义函数
val multi2 = (x: Int) => {x * x}

println(multi1(3)) //输出 9
println(multi2(3)) //输出 9
```

也可以使用 `def` 定义函数：

```
def multi3 = (x: Int) => {x * x}
println(multi3(3))   //输出 9
```

`multi2` 和 `multi3` 本质上没有区别，这是因为函数是一等公民，`val multi2 = (x: Int) => {x * x}` 这个语句相当于是使用 `def` 预先定义了函数，之后赋值给变量 `multi2`。

函数类型

上面我们说过 `multi2` 和 `multi3` 本质上是一样的，那么作为函数它们是什么类型的？两者的类型实际上都是 `Int => Int`，前面一个 Int 代表输入参数类型，后面一个 Int 代表返回值类型。

```
scala> val multi2 = (x: Int) => {x * x}
multi2: Int => Int = $$Lambda$1092/594363215@1dd1a777

scala> def multi3 = (x: Int) => {x * x}
multi3: Int => Int

// 如果有多个参数，则类型为：（参数类型，参数类型 ...）=>返回值类型
scala> val multi4 = (x: Int,name: String) => {name + x * x }
multi4: (Int, String) => String = $$Lambda$1093/1039732747@2eb4fe7
```

一等公民&匿名函数

在 Scala 中函数是一等公民，这意味着不仅可以定义函数并调用它们，还可以将它们作为值进行传递：

```
import scala.math.ceil
object ScalaApp extends App {
   // 将函数 ceil 赋值给变量 fun,使用下划线 (_) 指明是 ceil 函数但不传递参数
   val fun = ceil _
   println(fun(2.3456))   //输出 3.0

}
```

在 Scala 中你不必给每一个函数都命名，如 `(x: Int) => 3 * x` 就是一个匿名函数：

```scala
object ScalaApp extends App {
   // 1.匿名函数
  (x: Int) => 3 * x
   // 2.具名函数
   val fun = (x: Int) => 3 * x
   // 3.直接使用匿名函数
   val array01 = Array(1, 2, 3).map((x: Int) => 3 * x)
   // 4.使用占位符简写匿名函数
   val array02 = Array(1, 2, 3).map(_ * 3)
   // 5.使用具名函数
   val array03 = Array(1, 2, 3).map(fun)

}
```

特殊的函数表达式

可变长度参数列表

在 Java 中如果你想要传递可变长度的参数，需要使用 `String ...args` 这种形式，Scala 中等效的表达为 `args: String*` 。

```scala
object ScalaApp extends App {
   def echo(args: String*): Unit = {
      for (arg <- args) println(arg)
  }
   echo("spark","hadoop","flink")
}
// 输出
spark
hadoop
flink
```

传递具名参数

向函数传递参数时候可以指定具体的参数名。

```scala
object ScalaApp extends App {

   def detail(name: String, age: Int): Unit = println(name + ":" + age)

   // 1.按照参数定义的顺序传入
   detail("heibaiying", 12)
   // 2.传递参数的时候指定具体的名称,则不必遵循定义的顺序
   detail(age = 12, name = "heibaiying")

}
```

默认值参数

在定义函数时，可以为参数指定默认值。

```
object ScalaApp extends App {

  def detail(name: String, age: Int = 88): Unit = println(name + ":" + age)

  // 如果没有传递 age 值,则使用默认值
  detail("heibaiying")
  detail("heibaiying", 12)

}
```

2. 闭包

闭包的定义

```
var more = 10
// addMore 一个闭包函数:因为其捕获了自由变量 more 从而闭合了该函数字面量
val addMore = (x: Int) => x + more
```

如上函数 `addMore` 中有两个变量 x 和 more:

- **x** : 是一个绑定变量 (bound variable)，因为其是该函数的入参，在函数的上下文中有明确的定义；
- **more** : 是一个自由变量 (free variable)，因为函数字面量本生并没有给 more 赋予任何含义。

按照定义：在创建函数时，如果需要捕获自由变量，那么包含指向被捕获变量的引用的函数就被称为闭包函数。

修改自由变量

这里需要注意的是，闭包捕获的是变量本身，即是对变量本身的引用，这意味着：

- 闭包外部对自由变量的修改，在闭包内部是可见的；
- 闭包内部对自由变量的修改，在闭包外部也是可见的。

```
// 声明 more 变量
scala> var more = 10
more: Int = 10

// more 变量必须已经被声明，否则下面的语句会报错
scala> val addMore = (x: Int) => {x + more}
addMore: Int => Int = $$Lambda$1076/1844473121@876c4f0

scala> addMore(10)
res7: Int = 20

// 注意这里是给 more 变量赋值，而不是重新声明 more 变量
scala> more=1000
more: Int = 1000

scala> addMore(10)
res8: Int = 1010
```

自由变量多副本

自由变量可能随着程序的改变而改变，从而产生多个副本，但是闭包永远指向创建时候有效的那个变量副本。

```
// 第一次声明 more 变量
scala> var more = 10
```

```
more: Int = 10

// 创建闭包函数
scala> val addMore10 = (x: Int) => {x + more}
addMore10: Int => Int = $$Lambda$1077/1144251618@1bdaa13c

// 调用闭包函数
scala> addMore10(9)
res9: Int = 19

// 重新声明 more 变量
scala> var more = 100
more: Int = 100

// 创建新的闭包函数
scala> val addMore100 = (x: Int) => {x + more}
addMore100: Int => Int = $$Lambda$1078/626955849@4d0be2ac

// 引用的是重新声明 more 变量
scala> addMore100(9)
res10: Int = 109

// 引用的还是第一次声明的 more 变量
scala> addMore10(9)
res11: Int = 19

// 对于全局而言 more 还是 100
scala> more
res12: Int = 100
```

从上面的示例可以看出重新声明 `more` 后，全局的 `more` 的值是 100，但是对于闭包函数 `addMore10` 还是引用的是值为 10 的 `more`，这是由虚拟机来实现的，虚拟机会保证 `more` 变量在重新声明后，原来的被捕获的变量副本继续在堆上保持存活。

3. 高阶函数

使用函数作为参数

定义函数时候支持传入函数作为参数，此时新定义的函数被称为高阶函数。

```
object ScalaApp extends App {

   // 1.定义函数
   def square = (x: Int) => {
      x * x
   }

   // 2.定义高阶函数: 第一个参数是类型为 Int => Int 的函数
   def multi(fun: Int => Int, x: Int) = {
      fun(x) * 100
   }

   // 3.传入具名函数
   println(multi(square, 5)) // 输出 2500

   // 4.传入匿名函数
   println(multi(_ * 100, 5)) // 输出 50000
```

```
18
19    }
```

函数柯里化

我们上面定义的函数都只支持一个参数列表，而柯里化函数则支持多个参数列表。柯里化指的是将原来接受两个参数的函数变成接受一个参数的函数的过程。新的函数以原有第二个参数作为参数。

```
1   object ScalaApp extends App {
2     // 定义柯里化函数
3     def curriedSum(x: Int)(y: Int) = x + y
4     println(curriedSum(2)(3)) //输出 5
5   }
```

这里当你调用 curriedSum 时候，实际上是连着做了两次传统的函数调用，实际执行的柯里化过程如下：

- 第一次调用接收一个名为 `x` 的 Int 型参数，返回一个用于第二次调用的函数，假设 `x` 为 2，则返回函数 `2+y`；
- 返回的函数接收参数 `y`，并计算并返回值 `2+3` 的值。

想要获得柯里化的中间返回的函数其实也比较简单：

```
1   object ScalaApp extends App {
2     // 定义柯里化函数
3     def curriedSum(x: Int)(y: Int) = x + y
4     println(curriedSum(2)(3)) //输出 5
5
6     // 获取传入值为 10 返回的中间函数 10 + y
7     val plus: Int => Int = curriedSum(10)_
8     println(plus(3)) //输出值 13
9   }
```

柯里化支持多个参数列表，多个参数按照从左到右的顺序依次执行柯里化操作：

```
1   object ScalaApp extends App {
2     // 定义柯里化函数
3     def curriedSum(x: Int)(y: Int)(z: String) = x + y + z
4     println(curriedSum(2)(3)("name")) // 输出 5name
5
6   }
```

第四章 类和对象

Scala 的类与 Java 的类具有非常多的相似性，示例如下：

```
1    // 1. 在 scala 中，类不需要用 public 声明,所有的类都具有公共的可见性
2    class Person {
3
4      // 2. 声明私有变量,用 var 修饰的变量默认拥有 getter/setter 属性
5      private var age = 0
6
7      // 3.如果声明的变量不需要进行初始赋值，此时 Scala 就无法进行类型推断，所以需要显式指明类型
8      private var name: String = _
9
10
```

```
11      // 4. 定义方法,应指明传参类型。返回值类型不是必须的，Scala 可以自动推断出来，但是为了方便调用者，建议指明
12      def growUp(step: Int): Unit = {
13          age += step
14      }
15
16      // 5.对于改值器方法 (即改变对象状态的方法),即使不需要传入参数,也建议在声明中包含 ()
17      def growUpFix(): Unit = {
18          age += 10
19      }
20
21      // 6.对于取值器方法 (即不会改变对象状态的方法),不必在声明中包含 ()
22      def currentAge: Int = {
23          age
24      }
25
26      /**
27       * 7.不建议使用 return 关键字,默认方法中最后一行代码的计算结果为返回值
28       *    如果方法很简短，甚至可以写在同一行中
29       */
30      def getName: String = name
31
32  }
33
34
35  // 伴生对象
36  object Person {
37
38      def main(args: Array[String]): Unit = {
39          // 8.创建类的实例
40          val counter = new Person()
41          // 9.用 var 修饰的变量默认拥有 getter/setter 属性，可以直接对其进行赋值
42          counter.age = 12
43          counter.growUp(8)
44          counter.growUpFix()
45          // 10.用 var 修饰的变量默认拥有 getter/setter 属性，可以直接对其进行取值，输出: 30
46          println(counter.age)
47          // 输出: 30
48          println(counter.currentAge)
49          // 输出: null
50          println(counter.getName)
51      }
52
53  }
```

类

成员变量可见性

Scala 中成员变量的可见性默认都是 public，如果想要保证其不被外部干扰，可以声明为 private，并通过 getter 和 setter 方法进行访问。

getter和setter属性

getter 和 setter 属性与声明变量时使用的关键字有关：

- 使用 var 关键字：变量同时拥有 getter 和 setter 属性；
- 使用 val 关键字：变量只拥有 getter 属性；
- 使用 private[this]：变量既没有 getter 属性、也没有 setter 属性，只能通过内部的方法访问；

需要特别说明的是：假设变量名为 age,则其对应的 get 和 set 的方法名分别叫做 `age` 和 `age_=`。

```
class Person {

   private val name = "heibaiying"
   private var age = 12
   private[this] var birthday = "2019-08-08"
   // birthday 只能被内部方法所访问
   def getBirthday: String = birthday
}

object Person {
   def main(args: Array[String]): Unit = {
      val person = new Person
      person.age = 30
      println(person.name)
      println(person.age)
      println(person.getBirthday)
   }
}
```

解释说明：

示例代码中 `person.age=30` 在执行时内部实际是调用了方法 `person.age_=(30)`，而 `person.age` 内部执行时实际是调用了 `person.age()` 方法。想要证明这一点，可以对代码进行反编译。同时为了说明成员变量可见性的问题，我们对下面这段代码进行反编译：

```
class Person {
var name = ""
private var age = ""
}
```

依次执行下面编译命令：

```
> scalac Person.scala
> javap -private Person
```

编译结果如下，从编译结果可以看到实际的 get 和 set 的方法名 (因为 JVM 不允许在方法名中出现 =，所以它被翻译成$eq)，同时也验证了成员变量默认的可见性为 public。

```
Compiled from "Person.scala"
public class Person {
private java.lang.String name;
private java.lang.String age;

public java.lang.String name();
public void name_$eq(java.lang.String);

private java.lang.String age();
private void age_$eq(java.lang.String);

public Person();
}
```

@BeanProperty

在上面的例子中可以看到我们是使用 `.` 来对成员变量进行访问的，如果想要额外生成和 Java 中一样的 getXXX 和 setXXX 方法，则需要使用@BeanProperty 进行注解。

```
class Person {
   @BeanProperty var name = ""
}

object Person {
   def main(args: Array[String]): Unit = {
      val person = new Person
      person.setName("heibaiying")
      println(person.getName)
   }
}
```

主构造器

和 Java 不同的是，Scala 类的主构造器直接写在类名后面，但注意以下两点：

- 主构造器传入的参数默认就是 val 类型的，即不可变，你没有办法在内部改变传参；
- 写在主构造器中的代码块会在类初始化的时候被执行，功能类似于 Java 的静态代码块 `static{}`

```
class Person(val name: String, val age: Int) {

   println("功能类似于 Java 的静态代码块 static{}")

   def getDetail: String = {
      //name="heibai" 无法通过编译
      name + ":" + age
   }
}

object Person {
   def main(args: Array[String]): Unit = {
      val person = new Person("heibaiying", 20)
      println(person.getDetail)
   }
}

```

```
输出：
功能类似于 Java 的静态代码块 static{}
heibaiying:20
```

辅助构造器

辅助构造器有两点硬性要求：

- 辅助构造器的名称必须为 this；
- 每个辅助构造器必须以主构造器或其他的辅助构造器的调用开始。

```
class Person(val name: String, val age: Int) {

   private var birthday = ""

   // 1.辅助构造器的名称必须为 this
   def this(name: String, age: Int, birthday: String) {
      // 2.每个辅助构造器必须以主构造器或其他的辅助构造器的调用开始
      this(name, age)
      this.birthday = birthday
   }

   // 3.重写 toString 方法
   override def toString: String = name + ":" + age + ":" + birthday
}

object Person {
   def main(args: Array[String]): Unit = {
      println(new Person("heibaiying", 20, "2019-02-21"))
   }
}
```

方法传参不可变

在 Scala 中，方法传参默认是 val 类型，即不可变，这意味着你在方法体内部不能改变传入的参数。这和 Scala 的设计理念有关，Scala 遵循函数式编程理念，强调方法不应该有副作用。

```
class Person() {

   def low(word: String): String = {
      word="word" // 编译无法通过
      word.toLowerCase
   }
}
```

对象

Scala 中的 object(对象) 主要有以下几个作用：

- 因为 object 中的变量和方法都是静态的，所以可以用于存放工具类；
- 可以作为单例对象的容器；
- 可以作为类的伴生对象；
- 可以拓展类或特质；
- 可以拓展 Enumeration 来实现枚举。

工具类&单例&全局静态常量&拓展特质

这里我们创建一个对象 `Utils`,代码如下:

```
object Utils {

   /*
    *1. 相当于 Java 中的静态代码块 static,会在对象初始化时候被执行
    *    这种方式实现的单例模式是饿汉式单例,即无论你的单例对象是否被用到,
    *    都在一开始被初始化完成
    */
   val person = new Person

   // 2. 全局固定常量 等价于 Java 的 public static final
   val CONSTANT = "固定常量"

   // 3. 全局静态方法
   def low(word: String): String = {
      word.toLowerCase
   }
}
```

其中 Person 类代码如下:

```
class Person() {
   println("Person 默认构造器被调用")
}
```

新建测试类:

```
// 1.ScalaApp 对象扩展自 trait App
object ScalaApp extends App {

   // 2.验证单例
   println(Utils.person == Utils.person)

   // 3.获取全局常量
   println(Utils.CONSTANT)

   // 4.调用工具类
   println(Utils.low("ABCDEFG"))

}

// 输出如下:
Person 默认构造器被调用
true
固定常量
abcdefg
```

伴生对象

在 Java 中，你通常会用到既有实例方法又有静态方法的类，在 Scala 中，可以通过类和与类同名的伴生对象来实现。类和伴生对象必须存在与同一个文件中。

```
class Person() {

   private val name = "HEIBAIYING"
```

```
    def getName: String = {
      // 调用伴生对象的方法和属性
      Person.toLow(Person.PREFIX + name)
    }
}

// 伴生对象
object Person {

    val PREFIX = "prefix-"

    def toLow(word: String): String = {
      word.toLowerCase
    }

    def main(args: Array[String]): Unit = {
      val person = new Person
      // 输出 prefix-heibaiying
      println(person.getName)
    }

}
```

实现枚举类

Scala 中没有直接提供枚举类，需要通过扩展 `Enumeration`，并调用其中的 Value 方法对所有枚举值进行初始化来实现。

```
object Color extends Enumeration {

    // 1.类型别名,建议声明,在 import 时有用
    type Color = Value

    // 2.调用 Value 方法
    val GREEN = Value
    // 3.只传入 id
    val RED = Value(3)
    // 4.只传入值
    val BULE = Value("blue")
    // 5.传入 id 和值
    val YELLOW = Value(5, "yellow")
    // 6. 不传入 id 时,id 为上一个声明变量的 id+1,值默认和变量名相同
    val PINK = Value

}
```

使用枚举类：

```
// 1.使用类型别名导入枚举类
import com.heibaiying.Color.Color

object ScalaApp extends App {

    // 2.使用枚举类型,这种情况下需要导入枚举类
    def printColor(color: Color): Unit = {
```

```
        println(color.toString)
    }

    // 3.判断传入值和枚举值是否相等
    println(Color.YELLOW.toString == "yellow")
    // 4.遍历枚举类和值
    for (c <- Color.values) println(c.id + ":" + c.toString)
}

//输出
true
0:GREEN
3:RED
4:blue
5:yellow
6:PINK
```

第五章 继承和特质

继承

Scala中的继承结构

Scala 中继承关系如下图：

- Any 是整个继承关系的根节点；
- AnyRef 包含 Scala Classes 和 Java Classes，等价于 Java 中的 java.lang.Object；
- AnyVal 是所有值类型的一个标记；
- Null 是所有引用类型的子类型，唯一实例是 null，可以将 null 赋值给除了值类型外的所有类型的变量；
- Nothing 是所有类型的子类型。

extends & override

Scala 的集成机制和 Java 有很多相似之处，比如都使用 `extends` 关键字表示继承，都使用 `override` 关键字表示重写父类的方法或成员变量。示例如下：

```
//父类
class Person {

    var name = ""
    // 1.不加任何修饰词,默认为 public,能被子类和外部访问
    var age = 0
    // 2.使用 protected 修饰的变量能子类访问，但是不能被外部访问
    protected var birthday = ""
    // 3.使用 private 修饰的变量不能被子类和外部访问
    private var sex = ""

    def setSex(sex: String): Unit = {
        this.sex = sex
    }
    // 4.重写父类的方法建议使用 override 关键字修饰
```

```
    override def toString: String = name + ":" + age + ":" + birthday + ":" + sex

}
```

使用 `extends` 关键字实现继承：

```
// 1.使用 extends 关键字实现继承
class Employee extends Person {

    override def toString: String = "Employee~" + super.toString

    // 2.使用 public 或 protected 关键字修饰的变量能被子类访问
    def setBirthday(date: String): Unit = {
       birthday = date
     }

}
```

测试继承：

```
object ScalaApp extends App {

    val employee = new Employee

    employee.name = "heibaiying"
    employee.age = 20
    employee.setBirthday("2019-03-05")
    employee.setSex("男")

    println(employee)
}

// 输出： Employee~heibaiying:20:2019-03-05:男
```

调用超类构造器

在 Scala 的类中，每个辅助构造器都必须首先调用其他构造器或主构造器，这样就导致了子类的辅助构造器永远无法直接调用超类的构造器，只有主构造器才能调用超类的构造器。所以想要调用超类的构造器，代码示例如下：

```
class Employee(name:String,age:Int,salary:Double) extends Person(name:String,age:Int) {
    .....
}
```

类型检查和转换

想要实现类检查可以使用 `isInstanceOf`，判断一个实例是否来源于某个类或者其子类，如果是，则可以使用 `asInstanceOf` 进行强制类型转换。

```
object ScalaApp extends App {

    val employee = new Employee
    val person = new Person

    // 1. 判断一个实例是否来源于某个类或者其子类 输出 true
    println(employee.isInstanceOf[Person])
    println(person.isInstanceOf[Person])

```

```
    // 2. 强制类型转换
    var p: Person = employee.asInstanceOf[Person]

    // 3. 判断一个实例是否来源于某个类（而不是其子类）
    println(employee.getClass == classOf[Employee])

}
```

构造顺序和提前定义

构造顺序

在 Scala 中还有一个需要注意的问题，如果你在子类中重写父类的 val 变量，并且超类的构造器中使用了该变量，那么可能会产生不可预期的错误。下面给出一个示例：

```
// 父类
class Person {
   println("父类的默认构造器")
   val range: Int = 10
   val array: Array[Int] = new Array[Int](range)
}

//子类
class Employee extends Person {
   println("子类的默认构造器")
   override val range = 2
}

//测试
object ScalaApp extends App {
   val employee = new Employee
   println(employee.array.mkString("(", ",", ")"))

}
```

这里初始化 array 用到了变量 range，这里你会发现实际上 array 既不会被初始化 Array(10)，也不会被初始化为 Array(2)，实际的输出应该如下：

```
父类的默认构造器
子类的默认构造器
()
```

可以看到 array 被初始化为 Array(0)，主要原因在于父类构造器的执行顺序先于子类构造器，这里给出实际的执行步骤：

1. 父类的构造器被调用，执行 `new Array[Int](range)` 语句；
2. 这里想要得到 range 的值，会去调用子类 range() 方法，因为 `override val` 重写变量的同时也重写了其 get 方法；
3. 调用子类的 range() 方法，自然也是返回子类的 range 值，但是由于子类的构造器还没有执行，这也就意味着对 range 赋值的 `range = 2` 语句还没有被执行，所以自然返回 range 的默认值，也就是 0。

这里可能比较疑惑的是为什么 `val range = 2` 没有被执行，却能使用 range 变量，这里因为在虚拟机层面，是先对成员变量先分配存储空间并赋给默认值，之后才赋予给定的值。想要证明这一点其实也比较简单，代码如下：

```
class Person {
   // val range: Int = 10 正常代码 array 为 Array(10)
   val array: Array[Int] = new Array[Int](range)
   val range: Int = 10   //如果把变量的声明放在使用之后，此时数据 array 为 array(0)
}

object Person {
   def main(args: Array[String]): Unit = {
      val person = new Person
      println(person.array.mkString("(", ",", ")"))
   }
}
```

提前定义

想要解决上面的问题，有以下几种方法：

(1) . 将变量用 final 修饰，代表不允许被子类重写，即 `final val range: Int = 10`；

(2) . 将变量使用 lazy 修饰，代表懒加载，即只有当你实际使用到 array 时候，才去进行初始化；

```
lazy val array: Array[Int] = new Array[Int](range)
```

(3) . 采用提前定义，代码如下，代表 range 的定义优先于超类构造器。

```
class Employee extends {
   //这里不能定义其他方法
   override val range = 2
} with Person {
   // 定义其他变量或者方法
   def pr(): Unit = {println("Employee")}
}
```

但是这种语法也有其限制：你只能在上面代码块中重写已有的变量，而不能定义新的变量和方法，定义新的变量和方法只能写在下面代码块中。

> **注意事项**：类的继承和下文特质 (trait) 的继承都存在这个问题，也同样可以通过提前定义来解决。虽然如此，但还是建议合理设计以规避该类问题。

抽象类

Scala 中允许使用 `abstract` 定义抽象类，并且通过 `extends` 关键字继承它。

定义抽象类：

```
abstract class Person {
    // 1.定义字段
    var name: String
    val age: Int

    // 2.定义抽象方法
    def geDetail: String

    // 3. scala 的抽象类允许定义具体方法
    def print(): Unit = {
        println("抽象类中的默认方法")
    }
}
```

继承抽象类：

```
class Employee extends Person {
    // 覆盖抽象类中变量
    override var name: String = "employee"
    override val age: Int = 12

    // 覆盖抽象方法
    def geDetail: String = name + ":" + age
}

```

特质

trait & with

Scala 中没有 interface 这个关键字，想要实现类似的功能，可以使用特质 (trait)。trait 等价于 Java 8 中的接口，因为 trait 中既能定义抽象方法，也能定义具体方法，这和 Java 8 中的接口是类似的。

```
// 1.特质使用 trait 关键字修饰
trait Logger {

    // 2.定义抽象方法
    def log(msg: String)

    // 3.定义具体方法
    def logInfo(msg: String): Unit = {
        println("INFO:" + msg)
    }
}
```

想要使用特质，需要使用 `extends` 关键字，而不是 `implements` 关键字，如果想要添加多个特质，可以使用 `with` 关键字。

```
// 1.使用 extends 关键字,而不是 implements,如果想要添加多个特质，可以使用 with 关键字
class ConsoleLogger extends Logger with Serializable with Cloneable {

    // 2. 实现特质中的抽象方法
    def log(msg: String): Unit = {
        println("CONSOLE:" + msg)
    }
}
```

特质中的字段

和方法一样，特质中的字段可以是抽象的，也可以是具体的：

- 如果是抽象字段，则混入特质的类需要重写覆盖该字段；
- 如果是具体字段，则混入特质的类获得该字段，但是并非是通过继承关系得到，而是在编译时候，简单将该字段加入到子类。

```
trait Logger {
   // 抽象字段
   var LogLevel:String
   // 具体字段
   var LogType = "FILE"
}
```

覆盖抽象字段：

```
class InfoLogger extends Logger {
   // 覆盖抽象字段
   override var LogLevel: String = "INFO"
}
```

带有特质的对象

Scala 支持在类定义的时混入 `父类 trait`，而在类实例化为具体对象的时候指明其实际使用的 `子类 trait`。

trait Logger：

```
// 父类
trait Logger {
   // 定义空方法 日志打印
   def log(msg: String) {}
}
```

trait ErrorLogger：

```
// 错误日志打印，继承自 Logger
trait ErrorLogger extends Logger {
   // 覆盖空方法
   override def log(msg: String): Unit = {
      println("Error:" + msg)
    }
}
```

trait InfoLogger：

```
// 通知日志打印，继承自 Logger
trait InfoLogger extends Logger {

   // 覆盖空方法
   override def log(msg: String): Unit = {
      println("INFO:" + msg)
    }
}
```

具体的使用类：

```
// 混入 trait Logger
class Person extends Logger {
   // 调用定义的抽象方法
   def printDetail(detail: String): Unit = {
      log(detail)
   }
}
```

这里通过 main 方法来测试：

```
 object ScalaApp extends App {

    // 使用 with 指明需要具体使用的 trait
    val person01 = new Person with InfoLogger
    val person02 = new Person with ErrorLogger
    val person03 = new   Person with InfoLogger with ErrorLogger
    person01.log("scala")   //输出 INFO:scala
    person02.log("scala")   //输出 Error:scala
    person03.log("scala")   //输出 Error:scala

 }
```

这里前面两个输出比较明显，因为只指明了一个具体的 `trait`，这里需要说明的是第三个输出，**因为 trait 的调用是由右到左开始生效的**，所以这里打印出 `Error:scala`。

特质构造顺序

`trait` 有默认的无参构造器，但是不支持有参构造器。一个类混入多个特质后初始化顺序应该如下：

```
// 示例
class Employee extends Person with InfoLogger with ErrorLogger {...}
```

1. 超类首先被构造，即 Person 的构造器首先被执行；
2. 特质的构造器在超类构造器之前，在类构造器之后；特质由左到右被构造；每个特质中，父特质首先被构造；
    - Logger 构造器执行（Logger 是 InfoLogger 的父类）；
    - InfoLogger 构造器执行；
    - ErrorLogger 构造器执行；
3. 所有超类和特质构造完毕，子类才会被构造。

第六章 集合类型

集合简介

Scala 中拥有多种集合类型，主要分为可变的和不可变的集合两大类：

- **可变集合**： 可以被修改。即可以更改，添加，删除集合中的元素；
- **不可变集合类**：不能被修改。对集合执行更改，添加或删除操作都会返回一个新的集合，而不是修改原来的集合。

集合结构

Scala 中的大部分集合类都存在三类变体，分别位于 `scala.collection`，`scala.collection.immutable`，`scala.collection.mutable` 包中。还有部分集合类位于 `scala.collection.generic` 包下。

- **scala.collection.immutable**： 包是中的集合是不可变的；
- **scala.collection.mutable**： 包中的集合是可变的；
- **scala.collection**： 包中的集合，既可以是可变的，也可以是不可变的。

```
val sortSet = scala.collection.SortedSet(1, 2, 3, 4, 5)
val mutableSet = collection.mutable.SortedSet(1, 2, 3, 4, 5)
val immutableSet = collection.immutable.SortedSet(1, 2, 3, 4, 5)
```

如果你仅写了 `Set` 而没有加任何前缀也没有进行任何 `import`，则 Scala 默认采用不可变集合类。

```
scala> Set(1,2,3,4,5)
res0: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)
```

Scala 中所有集合的顶层实现是 `Traversable` 。它唯一的抽象方法是 `foreach`：

```
def foreach[U](f: Elem => U)
```

实现 `Traversable` 的集合类只需要实现这个抽象方法，其他方法可以从 `Traversable` 继承。`Traversable` 中的所有可用方法如下：

方法	作用
Abstract Method:	
`xs foreach f`	为 xs 的每个元素执行函数 f
Addition:	
`xs ++ ys`	一个包含 xs 和 ys 中所有元素的新的集合。 ys 是一个 Traversable 或 Iterator。
Maps:	
`xs map f`	对 xs 中每一个元素应用函数 f，并返回一个新的集合
`xs flatMap f`	对 xs 中每一个元素应用函数 f，最后将结果合并成一个新的集合
`xs collect f`	对 xs 中每一个元素调用偏函数 f，并返回一个新的集合
Conversions:	
`xs.toArray`	将集合转化为一个 Array
`xs.toList`	将集合转化为一个 List
`xs.toIterable`	将集合转化为一个 Iterable
`xs.toSeq`	将集合转化为一个 Seq
`xs.toIndexedSeq`	将集合转化为一个 IndexedSeq
`xs.toStream`	将集合转化为一个延迟计算的流
`xs.toSet`	将集合转化为一个 Set
`xs.toMap`	将一个（key, value）对的集合转化为一个 Map。 如果当前集合的元素类型不是（key, value）对形式， 则报静态类型错误。
Copying:	
`xs copyToBuffer buf`	拷贝集合中所有元素到缓存 buf
`xs copyToArray(arr, s, n)`	从索引 s 开始，将集合中最多 n 个元素复制到数组 arr。 最后两个参数是可选的。
Size info:	
`xs.isEmpty`	判断集合是否为空
`xs.nonEmpty`	判断集合是否包含元素
`xs.size`	返回集合中元素的个数
`xs.hasDefiniteSize`	如果 xs 具有有限大小，则为真。
Element Retrieval:	
`xs.head`	返回集合中的第一个元素（如果无序，则随机返回）
`xs.headOption`	以 Option 的方式返回集合中的第一个元素， 如果集合为空则返回 None

方法	作用
`xs.last`	返回集合中的最后一个元素（如果无序，则随机返回）
`xs.lastOption`	以 Option 的方式返回集合中的最后一个元素， 如果集合为空则返回 None
`xs find p`	以 Option 的方式返回满足条件 p 的第一个元素， 如果都不满足则返回 None
Subcollection:	
`xs.tail`	除了第一个元素之外的其他元素组成的集合
`xs.init`	除了最后一个元素之外的其他元素组成的集合
`xs slice (from, to)`	返回给定索引范围之内的元素组成的集合 （包含 from 位置的元素但不包含 to 位置的元素）
`xs take n`	返回 xs 的前 n 个元素组成的集合（如果无序，则返回任意 n 个元素）
`xs drop n`	返回 xs 的后 n 个元素组成的集合（如果无序，则返回任意 n 个元素）
`xs takeWhile p`	从第一个元素开始查找满足条件 p 的元素， 直到遇到一个不满足条件的元素，返回所有遍历到的值。
`xs dropWhile p`	从第一个元素开始查找满足条件 p 的元素， 直到遇到一个不满足条件的元素，返回所有未遍历到的值。
`xs filter p`	返回满足条件 p 的所有元素的集合
`xs withFilter p`	集合的非严格的过滤器。后续对 xs 调用方法 map、flatMap 以及 withFilter 都只用作于满足条件 p 的元素，而忽略其他元素
`xs filterNot p`	返回不满足条件 p 的所有元素组成的集合
Subdivisions:	
`xs splitAt n`	在给定位置拆分集合，返回一个集合对 (xs take n, xs drop n)
`xs span p`	根据给定条件拆分集合，返回一个集合对 (xs takeWhile p, xs dropWhile p)。即遍历元素，直到遇到第一个不符合条件的值则结束遍历，将遍历到的值和未遍历到的值分别放入两个集合返回。
`xs partition p`	按照筛选条件对元素进行分组
`xs groupBy f`	根据鉴别器函数 f 将 xs 划分为集合映射
Element Conditions:	
`xs forall p`	判断集合中所有的元素是否都满足条件 p
`xs exists p`	判断集合中是否存在一个元素满足条件 p
`xs count p`	xs 中满足条件 p 的元素的个数
Folds:	
`(z /: xs) (op)`	以 z 为初始值，从左到右对 xs 中的元素执行操作为 op 的归约操作

方法	作用
`(xs :\ z) (op)`	以 z 为初始值，从右到左对 xs 中的元素执行操作为 op 的归约操作
`xs.foldLeft(z) (op)`	同 (z /: xs) (op)
`xs.foldRight(z) (op)`	同 (xs :\ z) (op)
`xs reduceLeft op`	从左到右对 xs 中的元素执行操作为 op 的归约操作
`xs reduceRight op`	从右到左对 xs 中的元素执行操作为 op 的归约操作
Specific Folds:	
`xs.sum`	累计求和
`xs.product`	累计求积
`xs.min`	xs 中的最小值
`xs.max`	xs 中的最大值
String:	
`xs addString (b, start, sep, end)`	向 StringBuilder b 中添加一个字符串， 该字符串包含 xs 的所有元素。start、seq 和 end 都是可选的，seq 为分隔符，start 为开始符号，end 为结束符号。
`xs mkString (start, seq, end)`	将集合转化为一个字符串。start、seq 和 end 都是可选的，seq 为分隔符，start 为开始符号，end 为结束符号。
`xs.stringPrefix`	返回 xs.toString 字符串开头的集合名称
Views:	
`xs.view`	生成 xs 的视图
`xs view (from, to)`	生成 xs 上指定索引范围内元素的视图

下面为部分方法的使用示例：

```
scala> List(1, 2, 3, 4, 5, 6).collect { case i if i % 2 == 0 => i * 10 }
res0: List[Int] = List(20, 40, 60)

scala> List(1, 2, 3, 4, 5, 6).withFilter(_ % 2 == 0).map(_ * 10)
res1: List[Int] = List(20, 40, 60)

scala> (10 /: List(1, 2, 3)) (_ + _)
res2: Int = 16

scala> List(1, 2, 3, -4, 5).takeWhile (_ > 0)
res3: List[Int] = List(1, 2, 3)

scala> List(1, 2, 3, -4, -5).span (_ > 0)
res4: (List[Int], List[Int]) = (List(1, 2, 3),List(-4, -5))

scala> List(1, 2, 3).mkString("[","-","]")
```

```
res5: String = [1-2-3]
```

Trait Iterable

Scala 中所有的集合都直接或者间接实现了 `Iterable` 特质，`Iterable` 拓展自 `Traversable`，并额外定义了部分方法：

方法	作用
Abstract Method:	
`xs.iterator`	返回一个迭代器，用于遍历 xs 中的元素，与 foreach 遍历元素的顺序相同。
Other Iterators:	
`xs grouped size`	返回一个固定大小的迭代器
`xs sliding size`	返回一个固定大小的滑动窗口的迭代器
Subcollections:	
`xs takeRigtht n`	返回 xs 中最后 n 个元素组成的集合（如果无序，则返回任意 n 个元素组成的集合）
`xs dropRight n`	返回 xs 中除了最后 n 个元素外的部分
Zippers:	
`xs zip ys`	返回 xs 和 ys 的对应位置上的元素对组成的集合
`xs zipAll (ys, x, y)`	返回 xs 和 ys 的对应位置上的元素对组成的集合。其中较短的序列通过附加元素 x 或 y 来扩展以匹配较长的序列。
`xs.zipWithIndex`	返回一个由 xs 中元素及其索引所组成的元素对的集合
Comparison:	
`xs sameElements ys`	测试 xs 和 ys 是否包含相同顺序的相同元素

所有方法示例如下：

```
scala> List(1, 2, 3).iterator.reduce(_ * _ * 10)
res0: Int = 600

scala> List("a","b","c","d","e") grouped 2 foreach println
List(a, b)
List(c, d)
List(e)

scala> List("a","b","c","d","e") sliding 2 foreach println
List(a, b)
List(b, c)
List(c, d)
List(d, e)

```

```
scala>   List("a","b","c","d","e").takeRight(3)
res1: List[String] = List(c, d, e)

scala> List("a","b","c","d","e").dropRight(3)
res2: List[String] = List(a, b)

scala> List("a","b","c").zip(List(1,2,3))
res3: List[(String, Int)] = List((a,1), (b,2), (c,3))

scala> List("a","b","c","d").zipAll(List(1,2,3),"",4)
res4: List[(String, Int)] = List((a,1), (b,2), (c,3), (d,4))

scala> List("a","b","c").zipAll(List(1,2,3,4),"d","")
res5: List[(String, Any)] = List((a,1), (b,2), (c,3), (d,4))

scala> List("a", "b", "c").zipWithIndex
res6: List[(String, Int)] = List((a,0), (b,1), (c,2))

scala> List("a", "b") sameElements List("a", "b")
res7: Boolean = true

scala> List("a", "b") sameElements List("b", "a")
res8: Boolean = false
```

修改集合

当你想对集合添加或者删除元素，需要根据不同的集合类型选择不同的操作符号：

操作符	描述	集合类型
coll(k) 即 coll.apply(k)	获取指定位置的元素	Seq, Map
coll :+ elem elem +: coll	向集合末尾或者集合头增加元素	Seq
coll + elem coll + (e1, e2, ...)	追加元素	Seq, Map
coll - elem coll - (e1, e2, ...)	删除元素	Set, Map, ArrayBuffer
coll ++ coll2 coll2 ++: coll	合并集合	Iterable
coll -- coll2	移除 coll 中包含的 coll2 中的元素	Set, Map, ArrayBuffer
elem :: lst lst2 :: lst	把指定列表 (lst2) 或者元素 (elem) 添加到列表 (lst) 头部	List
list ::: list2	合并 List	List
set | set2 set & set2 set &~ set2	并集、交集、差集	Set
coll += elem coll += (e1, e2, ...) coll ++= coll2 coll -= elem coll -= (e1, e2, ...) coll --= coll2	添加或者删除元素，并将修改后的结果赋值给集合本身	可变集合
elem +=: coll coll2 ++=: coll	在集合头部追加元素或集合	ArrayBuffer

第七章 数组

定长数组

在 Scala 中，如果你需要一个长度不变的数组，可以使用 Array。但需要注意以下两点：

- 在 Scala 中使用 `(index)` 而不是 `[index]` 来访问数组中的元素，因为访问元素，对于 Scala 来说是方法调用，`(index)` 相当于执行了 `.apply(index)` 方法。
- Scala 中的数组与 Java 中的是等价的，`Array[Int]()` 在虚拟机层面就等价于 Java 的 `int[]`。

```
// 10 个整数的数组，所有元素初始化为 0
scala> val nums=new Array[Int](10)
nums: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

// 10 个元素的字符串数组，所有元素初始化为 null
```

```
scala> val strings=new Array[String](10)
strings: Array[String] = Array(null, null, null, null, null, null, null, null, null, null)

// 使用指定值初始化，此时不需要 new 关键字
scala> val a=Array("hello","scala")
a: Array[String] = Array(hello, scala)

// 使用 () 来访问元素
scala> a(0)
res3: String = hello
```

变长数组

在 scala 中通过 ArrayBuffer 实现变长数组 (又称缓冲数组)。在构建 ArrayBuffer 时必须给出类型参数，但不必指定长度，因为 ArrayBuffer 会在需要的时候自动扩容和缩容。变长数组的构建方式及常用操作如下：

```
import scala.collection.mutable.ArrayBuffer

object ScalaApp {

   // 相当于 Java 中的 main 方法
   def main(args: Array[String]): Unit = {
      // 1.声明变长数组 (缓冲数组)
      val ab = new ArrayBuffer[Int]()
      // 2.在末端增加元素
      ab += 1
      // 3.在末端添加多个元素
      ab += (2, 3, 4)
      // 4.可以使用 ++=追加任何集合
      ab ++= Array(5, 6, 7)
      // 5.缓冲数组可以直接打印查看
      println(ab)
      // 6.移除最后三个元素
      ab.trimEnd(3)
      // 7.在第 1 个元素之后插入多个新元素
      ab.insert(1, 8, 9)
      // 8.从第 2 个元素开始,移除 3 个元素,不指定第二个参数的话,默认值为 1
      ab.remove(2, 3)
      // 9.缓冲数组转定长数组
      val abToA = ab.toArray
      // 10. 定长数组打印为其 hashcode 值
      println(abToA)
      // 11. 定长数组转缓冲数组
      val aToAb = abToA.toBuffer
   }
}
```

需要注意的是：使用 `+=` 在末尾插入元素是一个高效的操作，其时间复杂度是 O(1)。而使用 `insert` 随机插入元素的时间复杂度是 O(n)，因为在其插入位置之后的所有元素都要进行对应的后移，所以在 `ArrayBuffer` 中随机插入元素是一个低效的操作。

数组遍历

```
object ScalaApp extends App {

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

   // 1.方式一 相当于 Java 中的增强 for 循环
   for (elem <- a) {
      print(elem)
   }

   // 2.方式二
   for (index <- 0 until a.length) {
      print(a(index))
   }

   // 3.方式三, 是第二种方式的简写
   for (index <- a.indices) {
      print(a(index))
   }

   // 4.反向遍历
   for (index <- a.indices.reverse) {
      print(a(index))
   }

}
```

这里我们没有将代码写在 main 方法中，而是继承自 App.scala，这是 Scala 提供的一种简写方式，此时将代码写在类中，等价于写在 main 方法中，直接运行该类即可。

数组转换

数组转换是指由现有数组产生新的数组。假设当前拥有 a 数组，想把 a 中的偶数元素乘以 10 后产生一个新的数组，可以采用下面两种方式来实现：

```
object ScalaApp extends App {

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

   // 1.方式一 yield 关键字
   val ints1 = for (elem <- a if elem % 2 == 0) yield 10 * elem
   for (elem <- ints1) {
      println(elem)
   }

   // 2.方式二 采用函数式编程的方式,这和 Java 8 中的函数式编程是类似的，这里采用下划线标表示其中的每个元素
   val ints2 = a.filter(_ % 2 == 0).map(_ * 10)
   for (elem <- ints1) {
      println(elem)
   }
}
```

多维数组

和 Java 中一样，多维数组由单维数组组成。

```
object ScalaApp extends App {

   val matrix = Array(Array(11, 12, 13, 14, 15, 16, 17, 18, 19, 20),
      Array(21, 22, 23, 24, 25, 26, 27, 28, 29, 30),
      Array(31, 32, 33, 34, 35, 36, 37, 38, 39, 40))


   for (elem <- matrix) {

      for (elem <- elem) {
         print(elem + "-")
      }
      println()
   }

}

打印输出如下：
11-12-13-14-15-16-17-18-19-20-
21-22-23-24-25-26-27-28-29-30-
31-32-33-34-35-36-37-38-39-40-
```

与Java互操作

由于 Scala 的数组是使用 Java 的数组来实现的，所以两者之间可以相互转换。

```
import java.util

import scala.collection.mutable.ArrayBuffer
import scala.collection.{JavaConverters, mutable}

object ScalaApp extends App {

   val element = ArrayBuffer("hadoop", "spark", "storm")
   // Scala 转 Java
   val javaList: util.List[String] = JavaConverters.bufferAsJavaList(element)
   // Java 转 Scala
   val scalaBuffer: mutable.Buffer[String] = JavaConverters.asScalaBuffer(javaList)
   for (elem <- scalaBuffer) {
      println(elem)
   }
}
```

第八章 列表和集合

List字面量

List 是 Scala 中非常重要的一个数据结构，其与 Array(数组) 非常类似，但是 List 是不可变的，和 Java 中的 List 一样，其底层实现是链表。

```
scala>   val list = List("hadoop", "spark", "storm")
list: List[String] = List(hadoop, spark, storm)

// List 是不可变
scala> list(1) = "hive"
<console>:9: error: value update is not a member of List[String]
```

List类型

Scala 中 List 具有以下两个特性：

- **同构 (homogeneous)**：同一个 List 中的所有元素都必须是相同的类型；
- **协变 (covariant)**：如果 S 是 T 的子类型，那么 `List[S]` 就是 `List[T]` 的子类型，例如 `List[String]` 是 `List[Object]` 的子类型。

需要特别说明的是空列表的类型为 `List[Nothing]`：

```
scala> List()
res1: List[Nothing] = List()
```

构建List

所有 List 都由两个基本单元构成：`Nil` 和 `::`(读作"cons")。即列表要么是空列表 (Nil)，要么是由一个 head 加上一个 tail 组成，而 tail 又是一个 List。我们在上面使用的 `List("hadoop", "spark", "storm")` 最终也是被解释为 `"hadoop"::"spark":: "storm"::Nil`。

```
scala>   val list01 = "hadoop"::"spark":: "storm"::Nil
list01: List[String] = List(hadoop, spark, storm)

// :: 操作符号是右结合的，所以上面的表达式和下面的等同
scala> val list02 = "hadoop"::("spark":: ("storm"::Nil))
list02: List[String] = List(hadoop, spark, storm)
```

模式匹配

Scala 支持展开列表以实现模式匹配。

```
scala>   val list = List("hadoop", "spark", "storm")
list: List[String] = List(hadoop, spark, storm)

scala> val List(a,b,c)=list
a: String = hadoop
b: String = spark
c: String = storm
```

如果只需要匹配部分内容，可以如下：

```
scala> val a::rest=list
a: String = hadoop
rest: List[String] = List(spark, storm)
```

列表的基本操作

常用方法

```
object ScalaApp extends App {

   val list = List("hadoop", "spark", "storm")

   // 1.列表是否为空
   list.isEmpty

   // 2.返回列表中的第一个元素
   list.head

   // 3.返回列表中除第一个元素外的所有元素 这里输出 List(spark, storm)
   list.tail

   // 4.tail 和 head 可以结合使用
   list.tail.head

   // 5.返回列表中的最后一个元素 与 head 相反
   list.init

   // 6.返回列表中除了最后一个元素之外的其他元素 与 tail 相反 这里输出 List(hadoop, spark)
   list.last

   // 7.使用下标访问元素
   list(2)

   // 8.获取列表长度
   list.length

   // 9. 反转列表
   list.reverse

}
```

indices

indices 方法返回所有下标。

```
scala> list.indices
res2: scala.collection.immutable.Range = Range(0, 1, 2)
```

take & drop & splitAt

- **take**：获取前 n 个元素；
- **drop**：删除前 n 个元素；
- **splitAt**：从第几个位置开始拆分。

```
scala> list take 2
res3: List[String] = List(hadoop, spark)

scala> list drop 2
res4: List[String] = List(storm)

scala> list splitAt 2
res5: (List[String], List[String]) = (List(hadoop, spark),List(storm))
```

flatten

flatten 接收一个由列表组成的列表，并将其进行扁平化操作，返回单个列表。

```
scala>  List(List(1, 2), List(3), List(), List(4, 5)).flatten
res6: List[Int] = List(1, 2, 3, 4, 5)
```

zip & unzip

对两个 List 执行 `zip` 操作结果如下，返回对应位置元素组成的元组的列表，`unzip` 则执行反向操作。

```
scala> val list = List("hadoop", "spark", "storm")
scala> val score = List(10,20,30)

scala> val zipped=list zip score
zipped: List[(String, Int)] = List((hadoop,10), (spark,20), (storm,30))

scala> zipped.unzip
res7: (List[String], List[Int]) = (List(hadoop, spark, storm),List(10, 20, 30))
```

toString & mkString

toString 返回 List 的字符串表现形式。

```
scala> list.toString
res8: String = List(hadoop, spark, storm)
```

如果想改变 List 的字符串表现形式，可以使用 mkString。mkString 有三个重载方法，方法定义如下：

```
// start：前缀  sep：分隔符  end:后缀
def mkString(start: String, sep: String, end: String): String =
   addString(new StringBuilder(), start, sep, end).toString

// seq 分隔符
def mkString(sep: String): String = mkString("", sep, "")

// 如果不指定分隔符 默认使用""分隔
def mkString: String = mkString("")
```

使用示例如下：

```
scala> list.mkString
res9: String = hadoopsparkstorm

scala>  list.mkString(",")
res10: String = hadoop,spark,storm

scala> list.mkString("{",",","}")
res11: String = {hadoop,spark,storm}
```

iterator & toArray & copyToArray

iterator 方法返回的是迭代器，这和其他语言的使用是一样的。

```
object ScalaApp extends App {

   val list = List("hadoop", "spark", "storm")

   val iterator: Iterator[String] = list.iterator

   while (iterator.hasNext) {
      println(iterator.next)
   }

}
```

toArray 和 toList 用于 List 和数组之间的互相转换。

```
scala> val array = list.toArray
array: Array[String] = Array(hadoop, spark, storm)

scala> array.toList
res13: List[String] = List(hadoop, spark, storm)
```

copyToArray 将 List 中的元素拷贝到数组中指定位置。

```
object ScalaApp extends App {

   val list = List("hadoop", "spark", "storm")
   val array = Array("10", "20", "30")

   list.copyToArray(array,1)

   println(array.toBuffer)
}

// 输出 : ArrayBuffer(10, hadoop, spark)
```

列表的高级操作

列表转换：map & flatMap & foreach

map 与 Java 8 函数式编程中的 map 类似，都是对 List 中每一个元素执行指定操作。

```
scala> List(1,2,3).map(_+10)
res15: List[Int] = List(11, 12, 13)
```

flatMap 与 map 类似，但如果 List 中的元素还是 List，则会对其进行 flatten 操作。

```
scala> list.map(_.toList)
res16: List[List[Char]] = List(List(h, a, d, o, o, p), List(s, p, a, r, k), List(s, t, o, r, m))

scala> list.flatMap(_.toList)
res17: List[Char] = List(h, a, d, o, o, p, s, p, a, r, k, s, t, o, r, m)
```

foreach 要求右侧的操作是一个返回值为 Unit 的函数，你也可以简单理解为执行一段没有返回值代码。

```
scala> var sum = 0
sum: Int = 0

scala> List(1, 2, 3, 4, 5) foreach (sum += _)

scala> sum
res19: Int = 15
```

列表过滤：filter & partition & find & takeWhile & dropWhile & span

filter 用于筛选满足条件元素，返回新的 List。

```
scala> List(1, 2, 3, 4, 5) filter (_ % 2 == 0)
res20: List[Int] = List(2, 4)
```

partition 会按照筛选条件对元素进行分组，返回类型是 tuple(元组)。

```
scala> List(1, 2, 3, 4, 5) partition (_ % 2 == 0)
res21: (List[Int], List[Int]) = (List(2, 4),List(1, 3, 5))
```

find 查找第一个满足条件的值，由于可能并不存在这样的值，所以返回类型是 `Option`，可以通过 `getOrElse` 在不存在满足条件值的情况下返回默认值。

```
scala> List(1, 2, 3, 4, 5) find (_ % 2 == 0)
res22: Option[Int] = Some(2)

val result: Option[Int] = List(1, 2, 3, 4, 5) find (_ % 2 == 0)
result.getOrElse(10)
```

takeWhile 遍历元素，直到遇到第一个不符合条件的值则结束遍历，返回所有遍历到的值。

```
scala> List(1, 2, 3, -4, 5) takeWhile (_ > 0)
res23: List[Int] = List(1, 2, 3)
```

dropWhile 遍历元素，直到遇到第一个不符合条件的值则结束遍历，返回所有未遍历到的值。

```
// 第一个值就不满足条件,所以返回列表中所有的值
scala> List(1, 2, 3, -4, 5) dropWhile  (_ < 0)
res24: List[Int] = List(1, 2, 3, -4, 5)


scala> List(1, 2, 3, -4, 5) dropWhile (_ < 3)
res26: List[Int] = List(3, -4, 5)
```

span 遍历元素，直到遇到第一个不符合条件的值则结束遍历，将遍历到的值和未遍历到的值分别放入两个 List 中返回，返回类型是 tuple(元组)。

```
scala> List(1, 2, 3, -4, 5) span (_ > 0)
res27: (List[Int], List[Int]) = (List(1, 2, 3),List(-4, 5))
```

列表检查：forall & exists

forall 检查 List 中所有元素，如果所有元素都满足条件，则返回 true。

```
scala> List(1, 2, 3, -4, 5) forall ( _ > 0 )
res28: Boolean = false
```

exists 检查 List 中的元素，如果某个元素已经满足条件，则返回 true。

```
scala>  List(1, 2, 3, -4, 5) exists (_ > 0 )
res29: Boolean = true
```

列表排序：sortWith

sortWith 对 List 中所有元素按照指定规则进行排序，由于 List 是不可变的，所以排序返回一个新的 List。

```
scala> List(1, -3, 4, 2, 6) sortWith (_ < _)
res30: List[Int] = List(-3, 1, 2, 4, 6)

scala> val list = List( "hive","spark","azkaban","hadoop")
list: List[String] = List(hive, spark, azkaban, hadoop)

scala> list.sortWith(_.length>_.length)
res33: List[String] = List(azkaban, hadoop, spark, hive)
```

List对象的方法

上面介绍的所有方法都是 List 类上的方法，下面介绍的是 List 伴生对象中的方法。

List.range

List.range 可以产生指定的前闭后开区间内的值组成的 List，它有三个可选参数: start(开始值)，end(结束值，不包含)，step(步长)。

```
scala>  List.range(1, 5)
res34: List[Int] = List(1, 2, 3, 4)

scala> List.range(1, 9, 2)
res35: List[Int] = List(1, 3, 5, 7)

scala> List.range(9, 1, -3)
res36: List[Int] = List(9, 6, 3)
```

List.fill

List.fill 使用指定值填充 List。

```
scala> List.fill(3)("hello")
res37: List[String] = List(hello, hello, hello)

scala> List.fill(2,3)("world")
res38: List[List[String]] = List(List(world, world, world), List(world, world, world))
```

List.concat

List.concat 用于拼接多个 List。

```
scala> List.concat(List('a', 'b'), List('c'))
res39: List[Char] = List(a, b, c)

scala> List.concat(List(), List('b'), List('c'))
res40: List[Char] = List(b, c)

scala> List.concat()
res41: List[Nothing] = List()
```

处理多个List

当多个 List 被放入同一个 tuple 中时候，可以通过 zipped 对多个 List 进行关联处理。

```
// 两个 List 对应位置的元素相乘
scala> (List(10, 20), List(3, 4, 5)).zipped.map(_ * _)
res42: List[Int] = List(30, 80)

// 三个 List 的操作也是一样的
scala> (List(10, 20), List(3, 4, 5), List(100, 200)).zipped.map(_ * _ + _)
res43: List[Int] = List(130, 280)

// 判断第一个 List 中元素的长度与第二个 List 中元素的值是否相等
scala>  (List("abc", "de"), List(3, 2)).zipped.forall(_.length == _)
res44: Boolean = true
```

缓冲列表ListBuffer

上面介绍的 List，由于其底层实现是链表，这意味着能快速访问 List 头部元素，但对尾部元素的访问则比较低效，这时候可以采用 `ListBuffer`，ListBuffer 提供了在常量时间内往头部和尾部追加元素。

```
import scala.collection.mutable.ListBuffer

object ScalaApp extends App {

   val buffer = new ListBuffer[Int]
   // 1.在尾部追加元素
   buffer += 1
   buffer += 2
   // 2.在头部追加元素
   3 +=: buffer
   // 3. ListBuffer 转 List
   val list: List[Int] = buffer.toList
   println(list)
}

//输出：List(3, 1, 2)
```

集(Set)

Set 是不重复元素的集合。分为可变 Set 和不可变 Set。

可变Set

```
object ScalaApp extends App {

   // 可变 Set
   val mutableSet = new collection.mutable.HashSet[Int]

   // 1.添加元素
   mutableSet.add(1)
   mutableSet.add(2)
   mutableSet.add(3)
   mutableSet.add(3)
   mutableSet.add(4)

   // 2.移除元素
   mutableSet.remove(2)

   // 3.调用 mkString 方法 输出 1,3,4
   println(mutableSet.mkString(","))

   // 4. 获取 Set 中最小元素
   println(mutableSet.min)

   // 5. 获取 Set 中最大元素
   println(mutableSet.max)

}
```

不可变Set

不可变 Set 没有 add 方法，可以使用 `+` 添加元素，但是此时会返回一个新的不可变 Set，原来的 Set 不变。

```
object ScalaApp extends App {

   // 不可变 Set
   val immutableSet = new collection.immutable.HashSet[Int]

   val ints: HashSet[Int] = immutableSet+1

   println(ints)

}

// 输出 Set(1)
```

Set间操作

多个 Set 之间可以进行求交集或者合集等操作。

```scala
object ScalaApp extends App {

    // 声明有序 Set
    val mutableSet = collection.mutable.SortedSet(1, 2, 3, 4, 5)
    val immutableSet = collection.immutable.SortedSet(3, 4, 5, 6, 7)

    // 两个 Set 的合集  输出：TreeSet(1, 2, 3, 4, 5, 6, 7)
    println(mutableSet ++ immutableSet)

    // 两个 Set 的交集  输出：TreeSet(3, 4, 5)
    println(mutableSet intersect immutableSet)

}
```

第九章 映射和元组

映射(Map)

构造Map

```scala
// 初始化一个空 map
val scores01 = new HashMap[String, Int]

// 从指定的值初始化 Map（方式一）
val scores02 = Map("hadoop" -> 10, "spark" -> 20, "storm" -> 30)

// 从指定的值初始化 Map（方式二）
val scores03 = Map(("hadoop", 10), ("spark", 20), ("storm", 30))
```

采用上面方式得到的都是不可变 Map(immutable map)，想要得到可变 Map(mutable map)，则需要使用：

```scala
val scores04 = scala.collection.mutable.Map("hadoop" -> 10, "spark" -> 20, "storm" -> 30)
```

获取值

```scala
object ScalaApp extends App {

    val scores = Map("hadoop" -> 10, "spark" -> 20, "storm" -> 30)

    // 1.获取指定 key 对应的值
    println(scores("hadoop"))

    // 2. 如果对应的值不存在则使用默认值
    println(scores.getOrElse("hadoop01", 100))
}
```

新增/修改/删除值

可变 Map 允许进行新增、修改、删除等操作。

```
object ScalaApp extends App {

   val scores = scala.collection.mutable.Map("hadoop" -> 10, "spark" -> 20, "storm" -> 30)

   // 1.如果 key 存在则更新
   scores("hadoop") = 100

   // 2.如果 key 不存在则新增
   scores("flink") = 40

   // 3.可以通过 += 来进行多个更新或新增操作
   scores += ("spark" -> 200, "hive" -> 50)

   // 4.可以通过 -= 来移除某个键和值
   scores -= "storm"

   for (elem <- scores) {println(elem)}
}

// 输出内容如下
(spark,200)
(hadoop,100)
(flink,40)
(hive,50)
```

不可变 Map 不允许进行新增、修改、删除等操作，但是允许由不可变 Map 产生新的 Map。

```
object ScalaApp extends App {

   val scores = Map("hadoop" -> 10, "spark" -> 20, "storm" -> 30)

   val newScores = scores + ("spark" -> 200, "hive" -> 50)

   for (elem <- scores) {println(elem)}

}

// 输出内容如下
(hadoop,10)
(spark,200)
(storm,30)
(hive,50)
```

遍历Map

```
object ScalaApp extends App {

   val scores = Map("hadoop" -> 10, "spark" -> 20, "storm" -> 30)

   // 1. 遍历键
   for (key <- scores.keys) { println(key) }

   // 2. 遍历值
   for (value <- scores.values) { println(value) }
```

```
10
11      // 3. 遍历键值对
12      for ((key, value) <- scores) { println(key + ":" + value) }
13
14  }
```

yield关键字

可以使用 `yield` 关键字从现有 Map 产生新的 Map。

```
1   object ScalaApp extends App {
2
3       val scores = Map("hadoop" -> 10, "spark" -> 20, "storm" -> 30)
4
5       // 1.将 scores 中所有的值扩大 10 倍
6       val newScore = for ((key, value) <- scores) yield (key, value * 10)
7       for (elem <- newScore) { println(elem) }
8
9
10      // 2.将键和值互相调换
11      val reversalScore: Map[Int, String] = for ((key, value) <- scores) yield (value, key)
12      for (elem <- reversalScore) { println(elem) }
13
14  }
15
16  // 输出
17  (hadoop,100)
18  (spark,200)
19  (storm,300)
20
21  (10,hadoop)
22  (20,spark)
23  (30,storm)
```

其他Map结构

在使用 Map 时候，如果不指定，默认使用的是 HashMap，如果想要使用 `TreeMap` 或者 `LinkedHashMap`，则需要显式的指定。

```
1   object ScalaApp extends App {
2
3       // 1.使用 TreeMap,按照键的字典序进行排序
4       val scores01 = scala.collection.mutable.TreeMap("B" -> 20, "A" -> 10, "C" -> 30)
5       for (elem <- scores01) {println(elem)}
6
7       // 2.使用 LinkedHashMap,按照键值对的插入顺序进行排序
8       val scores02 = scala.collection.mutable.LinkedHashMap("B" -> 20, "A" -> 10, "C" -> 30)
9       for (elem <- scores02) {println(elem)}
10  }
11
12  // 输出
13  (A,10)
14  (B,20)
15  (C,30)
16
17  (B,20)
18  (A,10)
19  (C,30)
```

可选方法

```
object ScalaApp extends App {

   val scores = scala.collection.mutable.TreeMap("B" -> 20, "A" -> 10, "C" -> 30)

   // 1. 获取长度
   println(scores.size)

   // 2. 判断是否为空
   println(scores.isEmpty)

   // 3. 判断是否包含特定的 key
   println(scores.contains("A"))

}
```

与Java互操作

```
import java.util
import scala.collection.{JavaConverters, mutable}

object ScalaApp extends App {

   val scores = Map("hadoop" -> 10, "spark" -> 20, "storm" -> 30)

   // scala map 转 java map
   val javaMap: util.Map[String, Int] = JavaConverters.mapAsJavaMap(scores)

   // java map 转 scala map
   val scalaMap: mutable.Map[String, Int] = JavaConverters.mapAsScalaMap(javaMap)

   for (elem <- scalaMap) {println(elem)}
}
```

元组(Tuple)

元组与数组类似，但是数组中所有的元素必须是同一种类型，而元组则可以包含不同类型的元素。

```
scala> val tuple=(1,3.24f,"scala")
tuple: (Int, Float, String) = (1,3.24,scala)
```

模式匹配

可以通过模式匹配来获取元组中的值并赋予对应的变量：

```
scala> val (a,b,c)=tuple
a: Int = 1
b: Float = 3.24
c: String = scala
```

如果某些位置不需要赋值，则可以使用下划线代替：

```
scala> val (a,_,_)=tuple
a: Int = 1
```

zip方法

```
object ScalaApp extends App {

     val array01 = Array("hadoop", "spark", "storm")
    val array02 = Array(10, 20, 30)

    // 1.zip 方法得到的是多个 tuple 组成的数组
    val tuples: Array[(String, Int)] = array01.zip(array02)
    // 2.也可以在 zip 后调用 toMap 方法转换为 Map
    val map: Map[String, Int] = array01.zip(array02).toMap

    for (elem <- tuples) { println(elem) }
    for (elem <- map) {println(elem)}
}

// 输出
(hadoop,10)
(spark,20)
(storm,30)

(hadoop,10)
(spark,20)
(storm,30)
```

第十章 模式匹配

模式匹配

Scala 支持模式匹配机制，可以代替 swith 语句、执行类型检查、以及支持析构表达式等。

更好的switch

Scala 不支持 switch，可以使用模式匹配 `match...case` 语法代替。但是 match 语句与 Java 中的 switch 有以下三点不同：

- Scala 中的 case 语句支持任何类型；而 Java 中 case 语句仅支持整型、枚举和字符串常量；
- Scala 中每个分支语句后面不需要写 break，因为在 case 语句中 break 是隐含的，默认就有；
- 在 Scala 中 match 语句是有返回值的，而 Java 中 switch 语句是没有返回值的。如下：

```
object ScalaApp extends App {

    def matchTest(x: Int) = x match {
       case 1 => "one"
       case 2 => "two"
       case _ if x > 9 && x < 100 => "两位数"    //支持条件表达式 这被称为模式守卫
       case _ => "other"
   }

    println(matchTest(1))    //输出 one
    println(matchTest(10))   //输出 两位数
    println(matchTest(200)) //输出 other
}
```

用作类型检查

```
object ScalaApp extends App {

   def matchTest[T](x: T) = x match {
      case x: Int => "数值型"
      case x: String => "字符型"
      case x: Float => "浮点型"
      case _ => "other"
   }

   println(matchTest(1))       //输出 数值型
   println(matchTest(10.3f)) //输出 浮点型
   println(matchTest("str")) //输出 字符型
   println(matchTest(2.1))    //输出 other
}
```

匹配数据结构

匹配元组示例：

```
object ScalaApp extends App {

   def matchTest(x: Any) = x match {
      case (0, _, _) => "匹配第一个元素为 0 的元组"
      case (a, b, c) => println(a + "~" + b + "~" + c)
      case _ => "other"
   }

   println(matchTest((0, 1, 2)))                  // 输出: 匹配第一个元素为 0 的元组
   matchTest((1, 2, 3))                              // 输出: 1~2~3
   println(matchTest(Array(10, 11, 12, 14))) // 输出: other
}
```

匹配数组示例：

```
object ScalaApp extends App {

   def matchTest[T](x: Array[T]) = x match {
      case Array(0) => "匹配只有一个元素 0 的数组"
      case Array(a, b) => println(a + "~" + b)
      case Array(10, _*) => "第一个元素为 10 的数组"
      case _ => "other"
   }

   println(matchTest(Array(0)))              // 输出: 匹配只有一个元素 0 的数组
   matchTest(Array(1, 2))                       // 输出: 1~2
   println(matchTest(Array(10, 11, 12))) // 输出: 第一个元素为 10 的数组
   println(matchTest(Array(3, 2, 1)))      // 输出: other
}
```

提取器

数组、列表和元组能使用模式匹配，都是依靠提取器 (extractor) 机制，它们伴生对象中定义了 `unapply` 或 `unapplySeq` 方法：

- **unapply**：用于提取固定数量的对象；
- **unapplySeq**：用于提取一个序列；

这里以数组为例，`Array.scala` 定义了 `unapplySeq` 方法：

```
def unapplySeq[T](x : scala.Array[T]) : scala.Option[scala.IndexedSeq[T]] = { /* compiled code */ }
```

`unapplySeq` 返回一个序列，包含数组中的所有值，这样在模式匹配时，才能知道对应位置上的值。

样例类

样例类

样例类是一种的特殊的类，它们被经过优化以用于模式匹配，样例类的声明比较简单，只需要在 `class` 前面加上关键字 `case`。下面给出一个样例类及其用于模式匹配的示例：

```
//声明一个抽象类
abstract class Person{}
```

```
// 样例类 Employee
case class Employee(name: String, age: Int, salary: Double) extends Person {}
```

```
// 样例类 Student
case class Student(name: String, age: Int) extends Person {}
```

当你声明样例类后，编译器自动进行以下配置：

- 构造器中每个参数都默认为 `val`；
- 自动地生成 `equals, hashCode, toString, copy` 等方法；
- 伴生对象中自动生成 `apply` 方法，使得可以不用 new 关键字就能构造出相应的对象；
- 伴生对象中自动生成 `unapply` 方法，以支持模式匹配。

除了上面的特征外，样例类和其他类相同，可以任意添加方法和字段，扩展它们。

用于模式匹配

样例的伴生对象中自动生成 `unapply` 方法，所以样例类可以支持模式匹配，使用如下：

```
object ScalaApp extends App {

   def matchTest(person: Person) = person match {
      case Student(name, _) => "student:" + name
      case Employee(_, _, salary) => "employee salary:" + salary
      case _ => "other"
   }

   println(matchTest(Student("heibai", 12)))            //输出: student:heibai
   println(matchTest(Employee("ying", 22, 999999))) //输出: employee salary:999999.0
}
```

第十一章 隐式转换和隐式参数

隐式转换

使用隐式转换

隐式转换指的是以 `implicit` 关键字声明带有单个参数的转换函数，它将值从一种类型转换为另一种类型，以便使用之前类型所没有的功能。示例如下：

```
// 普通人
class Person(val name: String)

// 雷神
class Thor(val name: String) {
   // 正常情况下只有雷神才能举起雷神之锤
   def hammer(): Unit = {
      println(name + "举起雷神之锤")
  }
}

object Thor extends App {
   // 定义隐式转换方法 将普通人转换为雷神 通常建议方法名使用 source2Target,即：被转换对象 To 转换对象
   implicit def person2Thor(p: Person): Thor = new Thor(p.name)
   // 这样普通人也能举起雷神之锤
   new Person("普通人").hammer()
}

输出： 普通人举起雷神之锤
```

隐式转换规则

并不是你使用 `implicit` 转换后，隐式转换就一定会发生，比如上面如果不调用 `hammer()` 方法的时候，普通人就还是普通人。通常程序会在以下情况下尝试执行隐式转换：

- 当对象访问一个不存在的成员时，即调用的方法不存在或者访问的成员变量不存在；
- 当对象调用某个方法，该方法存在，但是方法的声明参数与传入参数不匹配时。

而在以下三种情况下编译器不会尝试执行隐式转换：

- 如果代码能够在不使用隐式转换的前提下通过编译，则不会使用隐式转换；
- 编译器不会尝试同时执行多个转换，比如 `convert1(convert2(a))*b` ；
- 转换存在二义性，也不会发生转换。

这里首先解释一下二义性，上面的代码进行如下修改，由于两个隐式转换都是生效的，所以就存在了二义性：

```
//两个隐式转换都是有效的
implicit def person2Thor(p: Person): Thor = new Thor(p.name)
implicit def person2Thor2(p: Person): Thor = new Thor(p.name)
// 此时下面这段语句无法通过编译
new Person("普通人").hammer()
```

其次再解释一下多个转换的问题：

```
class ClassA {
   override def toString = "This is Class A"
}

```

```
class ClassB {
   override def toString = "This is Class B"
   def printB(b: ClassB): Unit = println(b)
}

class ClassC

class ClassD

object ImplicitTest extends App {
   implicit def A2B(a: ClassA): ClassB = {
      println("A2B")
      new ClassB
   }

   implicit def C2B(c: ClassC): ClassB = {
      println("C2B")
      new ClassB
   }

   implicit def D2C(d: ClassD): ClassC = {
      println("D2C")
      new ClassC
   }

   // 这行代码无法通过编译，因为要调用到 printB 方法，需要执行两次转换 C2B(D2C(ClassD))
   new ClassD().printB(new ClassA)

   /*
    *  下面的这一行代码虽然也进行了两次隐式转换，但是两次的转换对象并不是一个对象,所以它是生效的:
    *  转换流程如下:
    *  1. ClassC 中并没有 printB 方法,因此隐式转换为 ClassB,然后调用 printB 方法;
    *  2. 但是 printB 参数类型为 ClassB,然而传入的参数类型是 ClassA,所以需要将参数 ClassA 转换为 ClassB,这是第二次;
    *  即: C2B(ClassC) -> ClassB.printB(ClassA) -> ClassB.printB(A2B(ClassA)) -> ClassB.printB(ClassB)
    *  转换过程 1 的对象是 ClassC,而转换过程 2 的转换对象是 ClassA,所以虽然是一行代码两次转换，但是仍然是有效转换
    */
   new ClassC().printB(new ClassA)
}

// 输出:
C2B
A2B
This is Class B
```

引入隐式转换

隐式转换的可以定义在以下三个地方:

- 定义在原类型的伴生对象中;
- 直接定义在执行代码的上下文作用域中;
- 统一定义在一个文件中，在使用时候导入。

上面我们使用的方法相当于直接定义在执行代码的作用域中，下面分别给出其他两种定义的代码示例:

定义在原类型的伴生对象中：

```
class Person(val name: String)
// 在伴生对象中定义隐式转换函数
object Person{
   implicit def person2Thor(p: Person): Thor = new Thor(p.name)
}
```

```
class Thor(val name: String) {
   def hammer(): Unit = {
      println(name + "举起雷神之锤")
  }
}
```

```
// 使用示例
object ScalaApp extends App {
   new Person("普通人").hammer()
}
```

定义在一个公共的对象中：

```
object Convert {
   implicit def person2Thor(p: Person): Thor = new Thor(p.name)
}
```

```
// 导入 Convert 下所有的隐式转换函数
import com.heibaiying.Convert._

object ScalaApp extends App {
   new Person("普通人").hammer()
}
```

> 注：Scala 自身的隐式转换函数大部分定义在 `Predef.scala` 中，你可以打开源文件查看，也可以在 Scala 交互式命令行中采用 `:implicit -v` 查看全部隐式转换函数。

隐式参数

使用隐式参数

在定义函数或方法时可以使用标记为 `implicit` 的参数，这种情况下，编译器将会查找默认值，提供给函数调用。

```
// 定义分隔符类
class Delimiters(val left: String, val right: String)

object ScalaApp extends App {

      // 进行格式化输出
   def formatted(context: String)(implicit deli: Delimiters): Unit = {
      println(deli.left + context + deli.right)
   }

   // 定义一个隐式默认值 使用左右中括号作为分隔符
   implicit val bracket = new Delimiters("(", ")")
   formatted("this is context") // 输出: (this is context)
}
```

关于隐式参数，有两点需要注意：

1.我们上面定义 `formatted` 函数的时候使用了柯里化，如果你不使用柯里化表达式，按照通常习惯只有下面两种写法：

```
// 这种写法没有语法错误，但是无法通过编译
def formatted(implicit context: String, deli: Delimiters): Unit = {
   println(deli.left + context + deli.right)
}
// 不存在这种写法，IDEA 直接会直接提示语法错误
def formatted( context: String,   implicit deli: Delimiters): Unit = {
   println(deli.left + context + deli.right)
}
```

上面第一种写法编译的时候会出现下面所示 `error` 信息,从中也可以看出 `implicit` 是作用于参数列表中每个参数的，这显然不是我们想要到达的效果，所以上面的写法采用了柯里化。

```
not enough arguments for method formatted:
(implicit context: String, implicit deli: com.heibaiying.Delimiters)
```

2.第二个问题和隐式函数一样，隐式默认值不能存在二义性，否则无法通过编译，示例如下：

```
implicit val bracket = new Delimiters("(", ")")
implicit val brace = new Delimiters("{", "}")
formatted("this is context")
```

上面代码无法通过编译，出现错误提示 `ambiguous implicit values`，即隐式值存在冲突。

引入隐式参数

引入隐式参数和引入隐式转换函数方法是一样的，有以下三种方式：

- 定义在隐式参数对应类的伴生对象中；
- 直接定义在执行代码的上下文作用域中；
- 统一定义在一个文件中，在使用时候导入。

我们上面示例程序相当于直接定义执行代码的上下文作用域中，下面给出其他两种方式的示例：

定义在隐式参数对应类的伴生对象中；

```
class Delimiters(val left: String, val right: String)

object Delimiters {
   implicit val bracket = new Delimiters("(", ")")
}
```

```
// 此时执行代码的上下文中不用定义
object ScalaApp extends App {

   def formatted(context: String)(implicit deli: Delimiters): Unit = {
      println(deli.left + context + deli.right)
  }
   formatted("this is context")
}
```

统一定义在一个文件中，在使用时候导入：

```scala
object Convert {
   implicit val bracket = new Delimiters("(", ")")
}
```

```scala
// 在使用的时候导入
import com.heibaiying.Convert.bracket

object ScalaApp extends App {
   def formatted(context: String)(implicit deli: Delimiters): Unit = {
      println(deli.left + context + deli.right)
   }
   formatted("this is context") // 输出: (this is context)
}
```

利用隐式参数进行隐式转换

```scala
def smaller[T] (a: T, b: T) = if (a < b) a else b
```

在 Scala 中如果定义了一个如上所示的比较对象大小的泛型方法，你会发现无法通过编译。对于对象之间进行大小比较，Scala 和 Java 一样，都要求被比较的对象需要实现 java.lang.Comparable 接口。在 Scala 中，直接继承 Java 中 Comparable 接口的是特质 Ordered，它在继承 compareTo 方法的基础上，额外定义了关系符方法，源码如下:

```scala
trait Ordered[A] extends Any with java.lang.Comparable[A] {
   def compare(that: A): Int
   def <  (that: A): Boolean = (this compare that) <  0
   def >  (that: A): Boolean = (this compare that) >  0
   def <= (that: A): Boolean = (this compare that) <= 0
   def >= (that: A): Boolean = (this compare that) >= 0
   def compareTo(that: A): Int = compare(that)
}
```

所以要想在泛型中解决这个问题，有两种方法:

使用视图界定

```scala
object Pair extends App {

 // 视图界定
   def smaller[T<% Ordered[T]](a: T, b: T) = if (a < b) a else b

   println(smaller(1,2)) //输出 1
}
```

视图限定限制了 T 可以通过隐式转换 `Ordered[T]`，即对象一定可以进行大小比较。在上面的代码中 `smaller(1,2)` 中参数 `1` 和 `2` 实际上是通过定义在 `Predef` 中的隐式转换方法 `intWrapper` 转换为 `RichInt`。

```scala
// Predef.scala
@inline implicit def intWrapper(x: Int)    = new runtime.RichInt(x)
```

为什么要这么麻烦执行隐式转换，原因是 Scala 中的 Int 类型并不能直接进行比较，因为其没有实现 `Ordered` 特质，真正实现 `Ordered` 特质的是 `RichInt`。

利用隐式参数进行隐式转换

Scala2.11+ 后，视图界定被标识为废弃，官方推荐使用类型限定来解决上面的问题，本质上就是使用隐式参数进行隐式转换。

```
object Pair extends App {

    // order 既是一个隐式参数也是一个隐式转换，即如果 a 不存在 < 方法，则转换为 order(a)<b
   def smaller[T](a: T, b: T)(implicit order: T => Ordered[T]) = if (a < b) a else b

   println(smaller(1,2)) //输出 1
}
```