尚硅谷大数据技术之 Scala
数据结构

官网：www.atguigu.com

ShangGuigu Technologies Co., Ltd.

尚硅谷技术有限公司

一 数据结构

1.1 数据结构特点

Scala 同时支持可变集合和不可变集合，不可变集合从不可变，可以安全的并发访问。

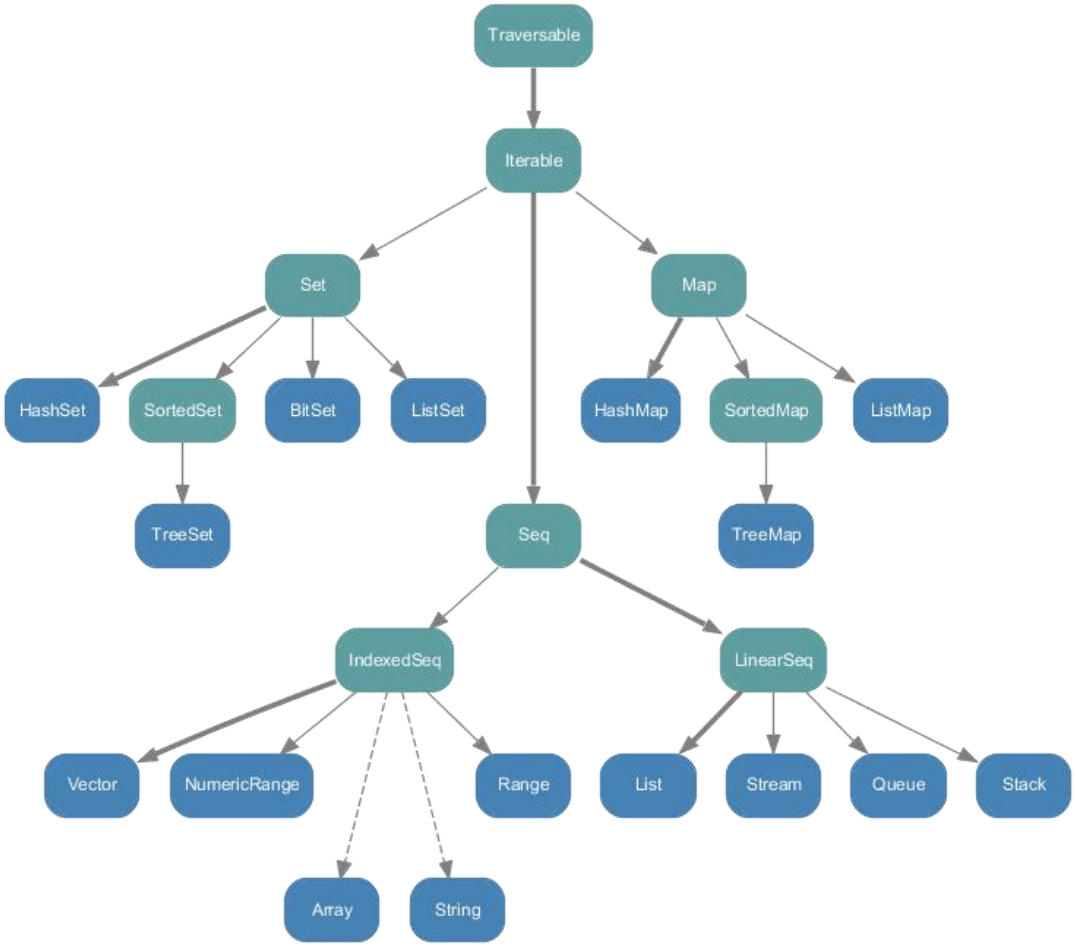
两个主要的包：

不可变集合：scala.collection.immutable

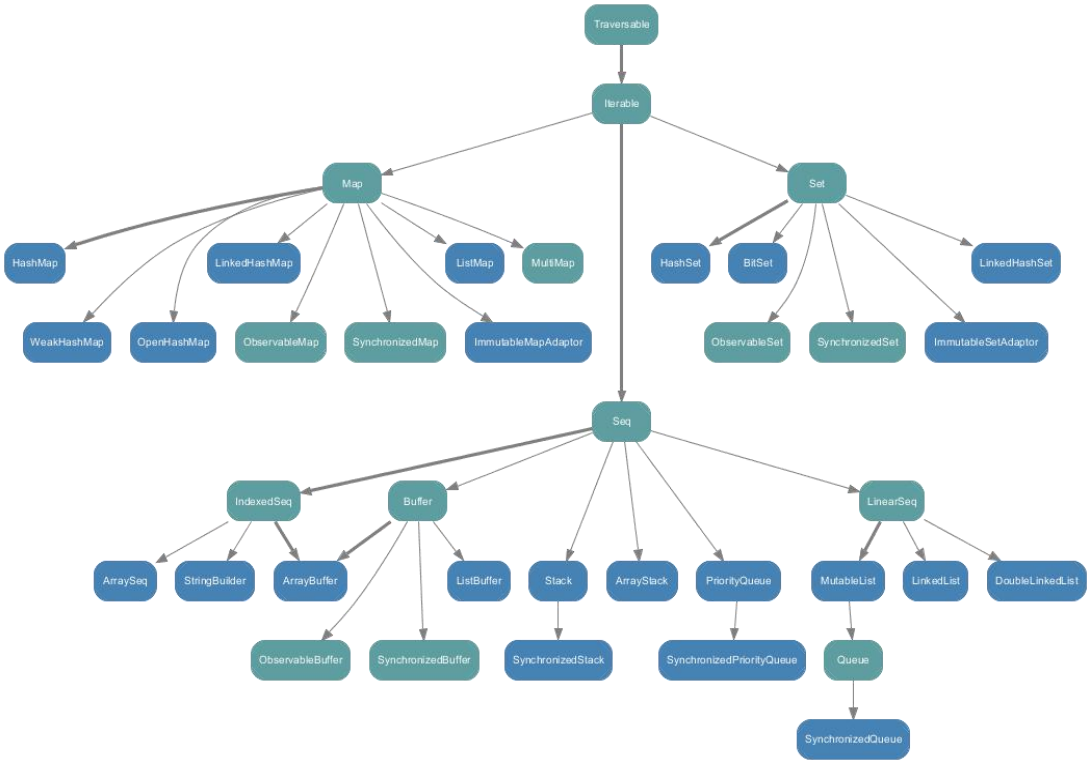
可变集合： scala.collection.mutable

Scala 默认采用不可变集合，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。

不可变集合继承层次：



可变集合继承层次：



1.2 数组 Array

1) 定长数组（声明泛型）

```
//第一种方式定义数组，这里的数组等同于 Java 中的数组
//中括号中的类型其实就是 Java 数组的类型
val arr1 = new Array[Int](10)
//赋值
arr1(1) = 7 // 集合元素采用小括号访问
中括号用来做泛型；数组的索引是从 0 开始的，不要被上面的例子误导；直接打印 println(arr1)结果是[I@47f37ef1，其中[代表数组；想要打印出数组的内容，这样 println(arr1.toBuffer)其实就是后面转换为变长数组
或：
//第二种方式定义数组    伴生对象 apply 方法
```

```
val arr1 = Array(1, 2，3)
```

2) 变长数组（声明泛型）

```
//定义
val arr2 = ArrayBuffer[Int]()
//追加值
arr2.append(7)
//重新赋值
arr2(0) = 7
```

3) 定长数组与变长数组的转换

```
arr1.toBuffer
arr2.toArray
```

4) 多维数组

```
//定义
val arr = Array.ofDim[Double](3,4)
//赋值
arr(1)(1) = 11.11
```

5) 与 Java 数组的互转

Scal 数组转 Java 数组：

```
val arr = ArrayBuffer("1", "2", "3")
//Scala to Java
import scala.collection.JavaConversions.bufferAsJavaList 导入了隐式转换规则
val javaArr = new ProcessBuilder(arr)  这里的 arr 已经是 java.util.list
println(javaArr.command())
```

Java 数组转 Scala 数组：

```
import scala.collection.JavaConversions.asScalaBuffer
import scala.collection.mutable.Buffer
val scalaArr: Buffer[String] = javaArr.command()
```

```
println(scalaArr)
```

6) 数组的遍历

```
for(x <- arr1) {
    println(x)
}
```

1.3 元组 Tuple

元组也是可以理解为一个容器，可以存放各种相同或不同类型的数据。

说的简单点，就是将多个无关的数据封装为一个整体

1) 元组的创建

```
val tuple1 = (1, 2, 3, "heiheihei")
println(tuple1)
```

2) 元组数据的访问，注意元素的访问有下划线，并且访问下标从 1 开始，而不是 0

```
val value1 = tuple1._4
println(value1)
```

3) 元组的遍历

```
for (elem <- tuple1.productIterator) {
    print(elem)
}
```

1.4 列表 List

1) 创建 List

```
val list1 = List(1, 2)
println(list1)
```

如果希望得到一个空列表，可以使用 Nil 对象。

```
val list1 = Nil
println(list1)
```

2) 访问 List 元素

```
val value1 = list1(1)
println(value1)
```

3) List 元素的追加

```
val list2 = list1 :+ 99      新的列表
println(list2)
val list3 = 100 +: list1
println(list3)
```

4) List 的创建与追加，符号“::”

```
val list4 = 1 :: 2 :: 3 :: list1 :: Nil
println(list4)
```

1.5 队列 Queue

队列数据存取符合先进先出策略

1) 队列的创建

```
import scala.collection.mutable
val q1 = new mutable.Queue[Int]
println(q1)
```

2) 队列元素的追加

```
q1 += 1;
println(q1)
```

3) 向队列中追加 List

```
q1 ++= List(2, 3, 4)      追加集合中的元素
println(q1)
```

4) 按照进入队列的顺序删除元素

```
q1.dequeue()
println(q1)
```

5) 塞入数据

```
q1.enqueue(9, 8, 7)
println(q1)
```

6) 返回队列的第一个元素

```
println(q1.head)
```

7) 返回队列最后一个元素

```
println(q1.last)
```

8) 返回除了第一个以外剩余的元素（尾部）

```
println(q1.tail)
```

1.6 映射 Map

1) 构造不可变映射

```
val map1 = Map("Alice" -> 10, "Bob" -> 20, "Kotlin" -> 30)
```

2) 构造可变映射

```
val map2 = scala.collection.mutable.Map("Alice" -> 10, "Bob" -> 20, "Kotlin" -> 30)
```

3) 空的映射

```
val map3 = new scala.collection.mutable.HashMap[String, Int]
```

4) 对偶元组（包含键值对的二元组）

```
val map4 = Map(("Alice", 10), ("Bob", 20), ("Kotlin", 30))
```

5) 取值

```
val value1 = map1("Alice")
println(value1)
```

如果映射中没有值，则会抛出异常，使用 contains 方法检查是否存在 key。（不推荐使用）

如果通过 映射.get(键) 这样的调用返回一个 Option 对象，要么是 Some，要么是 None。

默认值方法：getOrElse

6) 更新值

```
map2("Alice") = 99
```

```
println(map2("Alice"))
或：
map2 += ("Bob" -> 99)
map2 -= ("Alice", "Kotlin")
println(map2)
或：
val map5 = map2 + ("AAA" -> 10, "BBB" -> 20)
println(map5)
```

7) 遍历

```
for ((k, v) <- map1) println(k + " is mapped to " + v)
for (k <- map1.keys) println(k+"="+map(k))
for (v <- map1.values) println(v)
for(t <- map1) prinln(t._1+"="+t._2)
```

1.7 集 Set

集是不重复元素的结合。集不保留顺序，默认是以哈希集实现。

如果想要按照已排序的顺序来访问集中的元素，可以使用 SortedSet（已排序数据集），已排序的数据集是用红黑树实现的。 极限情况下放 11 个元素变成树

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

1) Set 不可变集合的创建

```
val set = Set(1, 2, 3)
println(set)
```

2) Set 可变集合的创建，如果 import 了可变集合，那么后续使用默认也是可变集合

```
import scala.collection.mutable.Set
val mutableSet = Set(1, 2, 3)
```

3) 可变集合的元素添加

```
mutableSet.add(4)
mutableSet += 6
// 注意该方法返回一个新的 Set 集合，而非在原有的基础上进行添加
mutableSet.+(5)
```

4) 可变集合的元素删除

```
mutableSet -= 1
mutableSet.remove(2)
println(mutableSet)
```

5) 遍历

```
for(x <- mutableSet) {
    println(x)
}
```

6) Set 更多常用操作

序号	方法	描述
1	def +(elem: A): Set[A]	为集合添加新元素，并创建一个新的集合，除非元素已存在
2	def -(elem: A): Set[A]	移除集合中的元素，并创建一个新的集合
3	def contains(elem: A): Boolean	如果元素在集合中存在，返回 true，否则返回 false。
4	def &(that: Set[A]): Set[A]	返回两个集合的交集
5	def &~(that: Set[A]): Set[A]	返回两个集合的差集
6	def ++(elems: A): Set[A]	合并两个集合
7	def drop(n: Int): Set[A]]	返回丢弃前 n 个元素新集合
8	def dropRight(n: Int): Set[A]	返回丢弃最后 n 个元素新集合

9	def dropWhile(p: (A) => Boolean): Set[A]	从左向右丢弃元素，直到条件 p 不成立
10	def max: A	查找最大元素
11	def min: A	查找最小元素
12	def take(n: Int): Set[A]	返回前 n 个元素

1.8 集合元素的映射&筛选

1) map：将集合中的每一个元素通过指定功能（函数）映射（转换）成新的结果集合

这里其实就是所谓的将函数作为参数传递给另外一个函数,这是函数式编程的特点

```
val names = List("Alice", "Bob", "Nick")
def upper( s : String ) : String = {
    s. toUpperCase
}
println(names.map(upper))
```

2) flatmap：flat 即压扁，压平，扁平化，效果就是将集合中的每个元素的子元素映射到某个函数并返回新的集合

```
val names = List("Alice", "Bob", "Nick")
def upper( s : String ) : String = {
    s. toUpperCase
}
println(names.flatMap(upper))
```

3) filter：将符合要求的数据(筛选)放置到新的集合中

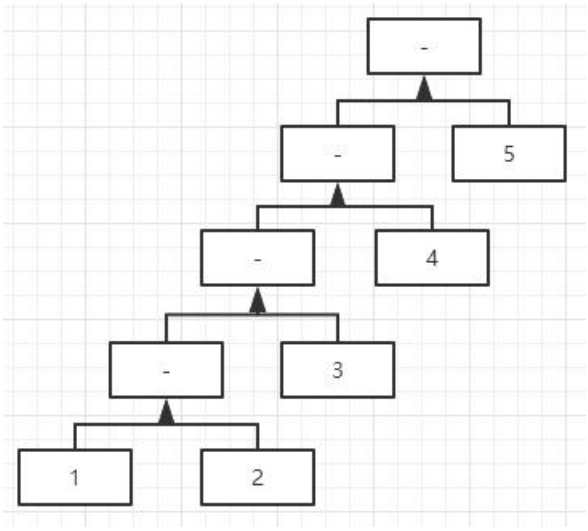
```
val names = List("Alice", "Bob", "Nick")
def find( s : String ) : Boolean = {
    s. startWith("A")
}
println(names.filter(find))
```

1.9 化简、折叠、扫描

1) 折叠，化简：将二元函数引用于集合中的函数

```
val list = List(1, 2, 3, 4, 5)
def minus( num1 : Int, num2 : Int ) : Int = {
    num1 - num2
}
val i1 = list.reduceLeft(minus)
println(i1)
val i2 = list.reduceRight(minus)
println(i2)
```

.reduceLefft(_ - _)这个函数的执行逻辑如图所示：



.reduceRight(_ - _)反之同理

2) 折叠，化简：fold

fold 函数将上一步返回的值作为函数的第一个参数继续传递参与运算，直到 list 中的所有元素被遍历。可以把 reduceLeft 看做简化版的 foldLeft。相关函数：fold，foldLeft，foldRight，可以参考 reduce 的相关方法理解。

```
val list2 = List(1, 9, 2, 8)
def minus( num1 : Int, num2 : Int ) : Int = {
```

```
    num1 - num2
}
val i4 = list2.fold(5)(minus)
println(i4)
```

foldRight

```
val list3 = List(1, 9, 2, 8)
def minus( num1 : Int, num2 : Int ) : Int = {
    num1 - num2
}
val i5 = list3.foldRight(100)(minus)
println(i5)
```

注：foldLeft 和 foldRight 有一种缩写方法对应分别是：/:和:\

例如：

foldLeft

```
val list4 = List(1, 9, 2, 8)
def minus( num1 : Int, num2 : Int ) : Int = {
    num1 - num2
}
val i6 = (0 /: list4)(minus)
println(i6)
```

3) 小应用：使用映射集合，统计一句话中，各个文字出现的次数

```
val sentence = "AAAAAAAAAABBBBBBBBCCCCCDDDDDDD"
// Map[Char, Int]()
```

4) 折叠，化简，扫描

这个理解需要结合上面的知识点，扫描，即对某个集合的所有元素做 fold 操作，但是会把产生的所有中间结果放置于一个集合中保存。

```
def minus( num1 : Int, num2 : Int ) : Int = {
```

```
    num1 - num2
}
val i8 = (1 to 5).scanLeft(0)(minus)
println(i8)
```

1.10 拉链

```
val list1 = List("1", "2", "3")
val list2 = List(4, 5)
var z1 = list1 zip list2
println(z1)
```

1.11 迭代器

你可以通过 iterator 方法从集合获得一个迭代器，通过 while 循环和 for 表达式对集合进行遍历。

```
val iterator = List(1, 2, 3, 4, 5).iterator
while (iterator.hasNext) {
  println(iterator.next())
}
或：
for(enum <- iterator) {
  println(enum)
}
```

1.12 流 Stream

stream 是一个集合。这个集合，可以用于存放，无穷多个元素，但是这无穷个元素并不会一次性生产出来，而是需要用到多大的区间，就会动态的生产，末尾元素遵循 **lazy** 规则。

1) 使用#::得到一个 stream

```
def numsForm(n: BigInt) : Stream[BigInt] = n #:: numsForm(n + 1)
```

2) 传递一个值，并打印 stream 集合

```
val tenOrMore = numsForm(10)
println(tenOrMore)
```

3) tail 的每一次使用，都会动态的向 stream 集合按照规则生成新的元素

```
println(tenOrMore.tail)
println(tenOrMore)
```

4) 使用 map 映射 stream 的元素并进行一些计算

```
def multi(x:Int) : Int {
    x * x
}
println(numsForm(5).map(multi))
```

1.13 视图 View

Stream 的懒加载特性，也可以对其他集合应用 view 方法来得到类似的效果，该方法产出一个其方法总是被懒执行的集合。但是 view 不会缓存数据，每次都要重新计算。

例如：我们找到 100 以内，所有数字倒序排列还是它本身的数字。

```
def multiple( num : Int ) : Int = {
    num
}
def eq( i : Int ) : Boolean = {
    i.toString.equals(i.toString.reverse)
}
val viewSquares = (1 to 100)
    .view
    .map(multiple)
    .filter(eq)
println(viewSquares)
```

```
for(x <- viewSquares){
    print(x + ", ")
}
```

1.14 线程安全的集合

所有线程安全的集合都是以 Synchronized 开头的集合，例如：

```
SynchronizedBuffer
SynchronizedMap
SynchronizedPriorityQueue
SynchronizedQueue
SynchronizedSet
SynchronizedStack
```

1.15 并行集合

Scala 为了充分使用多核 CPU，提供了并行集合（有别于前面的串行集合），用于多核环境的并行计算。

主要用到的算法有：

Divide and conquer：分治算法，Scala 通过 splitters，combiners 等抽象层来实现，主要原理是将计算工作分解很多任务，分发给一些处理器去完成，并将它们处理结果合并返回

Work stealin：算法，主要用于任务调度负载均衡（load-balancing），通俗点完成自己的所有任务之后，发现其他人还有活没干完，主动（或被安排）帮他人一起干，这样达到尽早干完的目的。

1) 打印 1~5

```
(1 to 5).foreach(println(_))
println()
(1 to 5).par.foreach(println(_))
```

2) 查看并行集合中元素访问的线程

```
val result1 = (0 to 100).map{case _ => Thread.currentThread.getName}.distinct
val result2 = (0 to 100).par.map{case _ => Thread.currentThread.getName}.distinct
println(result1)
println(result2)
```

1.16 操作符

这部分内容没有必要刻意去理解和记忆，语法使用的多了，自然就会产生感觉，该部分内容暂时大致了解一下即可。

1) 如果想在变量名、类名等定义中使用语法关键字（保留字），可以配合反引号反引号：

```
val `val` = 42
```

2) 这种形式叫中置操作符，A 操作符 B 等同于 A.操作符(B)

3) 后置操作符，A 操作符等同于 A.操作符，如果操作符定义的时候不带()则调用时不能加括号

4) 前置操作符，+、-、！、~等操作符 A 等同于 A.unary_操作符

5) 赋值操作符，A 操作符=B 等同于 A=A 操作符 B

二 模式匹配

2.1 switch

Scala 中的模式匹配类似于 Java 中的 switch 语法，但是更加强大。

模式匹配语法中，采用 match 关键字声明，每个分支采用 **case** 关键字进行声明，当需要匹配时，会从第一个 case 分支开始，如果匹配成功，那么执行对应的逻辑代码，如果匹配不成功，继续执行下一个分支进行判断。如果所有 case 都匹配，那么会执行 **case _** 分支，类似于 Java 中 default 语句。

如果没有任何模式匹配成功，那么会抛出 MatchError。

每个 case 中，不用 break 语句。

可以在 match 中使用任何类型，而不仅仅是数字。

```
// Java
```

```
int i = 1;
switch ( i ) {
        case 0 :
            break;
        case 1 :
            break;
        default :
            break
}
```

```
// Scala
var result = 0;
val op : Char = '-'
op match {
    case '+' => result = 1
    case '-' => result = -1
    case _ => result = 0
}
println(result)
```

2.2 守卫

如果想要表达匹配某个范围的数据，就需要在模式匹配中增加条件守卫

```
for (ch <- "+-3!") {
    var sign = 0
    var digit = 0
    ch match {
        case '+' => sign = 1
        case '-' => sign = -1
```

```
        case _ if ch.toString.equals("3") => digit = 3
        case _ => sign = 0
    }
    println(ch + " " + sign + " " + digit)
}
```

2.3 模式中的变量

如果在 case 关键字后跟变量名，那么 match 前表达式的值会赋给那个变量。

```
val str = "+-3!"
for (i <- str.indices) {
    var sign = 0
    var digit = 0

    str(i) match {
        case '+' => sign = 1
        case '-' => sign = -1
        case ch if Character.isDigit(ch) => digit = Character.digit(ch, 10)
        case _ =>
    }
    println(str(i) + " " + sign + " " + digit)
}
```

2.4 类型匹配

可以匹配对象的任意类型，这样做避免了使用 isInstanceOf 和 asInstanceOf 方法。

```
val a = 6
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 => Int.MaxValue
    case m: Map[String, Int] => "Map[String, Int]类型的 Map 集合"
    case a: Array[String] => "It's an Array[String]"
    case _ => 0
}
println(r1 + ", " + r1.getClass.getName)
```

注：类型不能直接匹配泛型类型

```
val a = 5
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 => Int.MaxValue
        case m: Map[String, Int] => "Map[String, Int]类型的 Map 集合"
        case m: Map[Int, String] => "Map[Int, String]类型的 Map 集合"
        case a: Array[String] => "It's an Array[String]"
        case a: Array[Int] => "It's an Array[Int]"
        case _ => 0
}
println(r1 + ", " + r1.getClass.getName)
```

2.5 匹配数组、列表、元组

Array(0) 匹配只有一个元素且为 0 的数组。

Array(x,y) 匹配数组有两个元素，并将两个元素赋值为 x 和 y。

Array(0,_*) 匹配数组以 0 开始。

1) 匹配数组

```
for (arr <- Array(Array(0), Array(1, 0), Array(0, 1, 0), Array(1, 1, 0), Array(1, 1, 0, 1))) {
   val result = arr match {
      case Array(0) => "0"
      case Array(x, y) => x + " " + y
      case Array(x, y, z) => x + " " + y + " " + z
      case Array(0, _*) => "0..."
      case _ => "something else"
   }
   println(result)
}
```

2) 匹配列表

与匹配数组相似，同样可以应用于列表

```
for (list <- Array(List(0), List(1, 0), List(0, 0, 0), List(1, 0, 0))) {
    val result = list match {
        case 0 :: Nil => "0"
        case x :: y :: Nil => x + " " + y
        case 0 :: tail => "0 ..."
        case _ => "something else"
    }
    println(result)
}
```

3) 匹配元组

同样可以应用于元组

```
for (pair <- Array((0, 1), (1, 0), (1, 1))) {
    val result = pair match {
        case (0, _) => "0 ..."
        case (y, 0) => y + " 0"
        case _ => "neither is 0"
    }
    println(result)
}
```

2.6 对象匹配

对象匹配，什么才算是匹配呢？即，case 中对象的 unapply 方法返回 some 集合则为匹配成功，返回 none 集合则为匹配失败。

1) unapply

--调用 unapply，传入 number

--接收返回值并判断返回值是 None，还是 Some

--如果是 Some，则将其中的值赋值给 n（就是 case Square(n)中的 n）

--请参考伴生对象的 apply 方法理解

创建 object Square：

```
object Square {
    def unapply(z: Double): Option[Double] = Some(math.sqrt(z))
}
```

模式匹配使用：

```
val number: Double = 36.0
number match {
    case Square(n) => println(n)
    case _ => println("nothing matched")
}
```

2) unapplySeq

--调用 unapplySeq，传入 namesString

--接收返回值并判断返回值是 None，还是 Some

--如果是 Some，获取其中的值

--判断得到的 sequence 中的元素的个数是否是三个

--如果是三个，则把三个元素分别取出，赋值给 first，second 和 third

创建 object Names：

```
object Names {
    def unapplySeq(str: String): Option[Seq[String]] = {
        if (str.contains(",")) Some(str.split(","))
        else None
    }
}
```

模式匹配使用：

```
val namesString = "Alice,Bob,Thomas"
namesString match {
```

```
    case Names(first, second, third) => {
        println("the string contains three people's names")
        println(s"$first $second $third")
    }
    case _ => println("nothing matched")
}
```

2.7 变量声明中的模式

match 中每一个 case 都可以单独提取出来，意思是一样的，如下：

```
val (x, y) = (1, 2)
val (q, r) = BigInt(10) /% 3
val arr = Array(1, 7, 2, 9)
val Array(first, second, _*) = arr
println(first, second)
```

2.8 for 表达式中的模式

```
val map = Map("A"->1, "B"->0, "C"->3)
for ( (k, v) <- map ) {
        println(k + " -> " + v)
}
for ((k, 0) <- map) {
        println(k + " --> " + 0) // for 中匹配会自动忽略失败的匹配
}
for ((k, v) <- map if v == 0) {
        println(k + " ---> " + 0)
}
```

2.9 样例类

样例类首先是类，除此之外它是为模式匹配而优化的类，样例类用 case 关键字进行声

明：

1) 样例类的创建

```
abstract class Amount
case class Dollar(value: Double) extends Amount
case class Currency(value: Double, unit: String) extends Amount
case object Nothing extends Amount
```

2) 当我们有一个类型为 Amount 的对象时，我们可以用模式匹配来匹配他的类型，并将属性值绑定到变量：

```
for (amt <- Array(Dollar(1000.0), Currency(1000.0, "EUR"), Nothing)) {
  val result = amt match {
    case Dollar(v) => "$" + v
    case Currency(_, u) => u
    case Nothing => ""
  }
  println(amt + ": " + result)
}
```

当你声明样例类时，如下几件事情会自动发生：

--构造器中的每一个参数都成为 val——除非它被显式地声明为 var（不建议这样做）

--在伴生对象中提供 apply 方法让你不用 new 关键字就能构造出相应的对象，比如 Dollar(29.95)或 Currency(29.95, "EUR")

--提供 unapply 方法让模式匹配可以工作

--将生成 toString、equals、hashCode 和 **copy** 方法——除非显式地给出这些方法的定义。

除上述外，样例类和其他类型完全一样。你可以添加方法和字段，扩展它们。

样例类的 copy 方法和带名参数

copy 创建一个与现有对象值相同的新对象，并可以通过带名参数来修改某些属性。

```
val amt = Currency(29.95, "EUR")
val price = amt.copy(value = 19.95)
```

```
println(amt)
println(price)
println(amt.copy(unit = "CHF"))
```

2.10 case 语句的中置(缀)表达式

什么是中置表达式？1 + 2，这就是一个中置表达式。如果 unapply 方法产出一个元组，你可以在 case 语句中使用中置表示法。比如可以匹配一个 List 序列。

```
List(1, 7, 2, 9) match {
  case first :: second :: rest => println(first + second + rest.length)
  case _ => 0
}
```

2.11 匹配嵌套结构

操作原理类似于正则表达式

比如某一系列商品想捆绑打折出售

1) 创建样例类

```
abstract class Item
case class Article(description: String, price: Double) extends Item
case class Bundle(description: String, discount: Double, item: Item*) extends Item
```

2) 匹配嵌套结构

```
val sale = Bundle("书籍", 10,  Article("漫画", 40), Bundle("文学作品", 20, Article("《阳关》
", 80), Article("《围城》", 30)))
```

3) 将 descr 绑定到第一个 Article 的描述

```
val result1 = sale match {
  case Bundle(_, _, Article(descr, _), _*) => descr
}
println(result1)
```

4) 通过@表示法将嵌套的值绑定到变量。_*绑定剩余 Item 到 rest

```
val result2 = sale match {
    case Bundle(_, _, art @ Article(_, _), rest @ _*) => (art, rest)
}
println(result2)
```

5) 不使用_*绑定剩余 Item 到 rest

```
val result3 = sale match {
    case Bundle(_, _, art @ Article(_, _), rest) => (art, rest)
}
println(result3)
```

6) 计算某个 Item 价格的函数，并调用

```
def price(it: Item): Double = {
    it match {
        case Article(_, p) => p
        case Bundle(_, disc, its@_*) => its.map(price _).sum - disc
    }
}

println(SwitchBaseSyllabus.price(sale))
```

2.12 密封类

如果想让 case 类的所有子类都必须在申明该类的相同的源文件中定义，可以将样例类的通用超类声明为 sealed，这个超类称之为密封类，密封就是不能在其他文件中定义子类。

2.13 模拟枚举

样例类可以模拟出枚举类型

1) 创建样例类

```
abstract class TrafficLightColor
```

```
case object Red extends TrafficLightColor
case object Yellow extends TrafficLightColor
case object Green extends TrafficLightColor
```

2) 模拟枚举

```
for (color <- Array(Red, Yellow, Green))
  println(
    color match {
      case Red => "stop"
      case Yellow => "slowly"
      case Green => "go"
    })
```

2.14 偏函数

将集合中的数字加 1 并返回新的集合

```
val list = List(1, 2, 3, 4)
def add( obj : Any ) : Int = {
    obj match {
        case i:Int => i+1
    }
}
val list1 = list.map( add )
println(list1)
```

如果这个时候，集合的元素有字符串，会出现什么情况？

```
val list = List(1, 2, 3, 4, "ABC")
def add( obj : Any ) : Int = {
    obj match {
        case i:Int => i+1
```

```
    }
}
val list1 = list.map( add )
println(list1)
// 此时程序执行时会发生错误
```

在对符合某个条件，而不是所有情况进行逻辑操作时，使用偏函数是一个不错的选择

将包在大括号内的一组 case 语句封装为函数，我们称之为**偏函数**，它只对会作用于指定类型的参数或指定范围值的参数实施计算，超出范围的值会忽略（未必会忽略，这取决于你打算怎样处理）

偏函数在 Scala 中是一个特质 PartialFunction

```
// 定义一个将 List 集合里面数字加 1 的偏函数
// 构建特质的实现类
// [Any, Int]是泛型，第一个表示参数类型，第二个表示返回参数
val addOne= new PartialFunction[Any, Int] {
  def apply(any: Any) = any.asInstanceOf[Int] + 1
  def isDefinedAt(any: Any) = if (any.isInstanceOf[Int]) true else false
}
val list1 = list.map(addOne) // (X) map 函数不支持偏函数
val list1 = list.collect(addOne) // OK collect 函数支持偏函数
println(list1)
```

声明偏函数，需要重新特质中的方法，有的时候会略显麻烦，而 Scala 其实提供了简单的方法

```
def f2: PartialFunction[Any, Int] = {
  case i: Int => i + 1 // case 语句可以自动转换为偏函数
}
val rf2 = List(1, 2, 3, 4,"ABC") collect f2
println(rf2)
```

```
// 上面的代码也可以简化为
val rf2 = List(1, 2, 3, 4,"ABC") collect { case i: Int => i + 1 }
println(rf2)
```