Scala编程基础

山丘

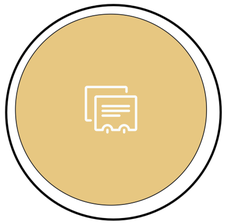
西普教育
SimpleEdu

主要内容

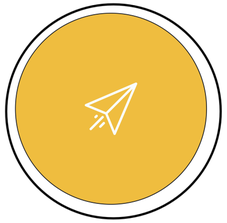
Scala基础

程序流程控制

函数式编程



Scala基本语法



集合和函数

面向对象编程

Scala基础

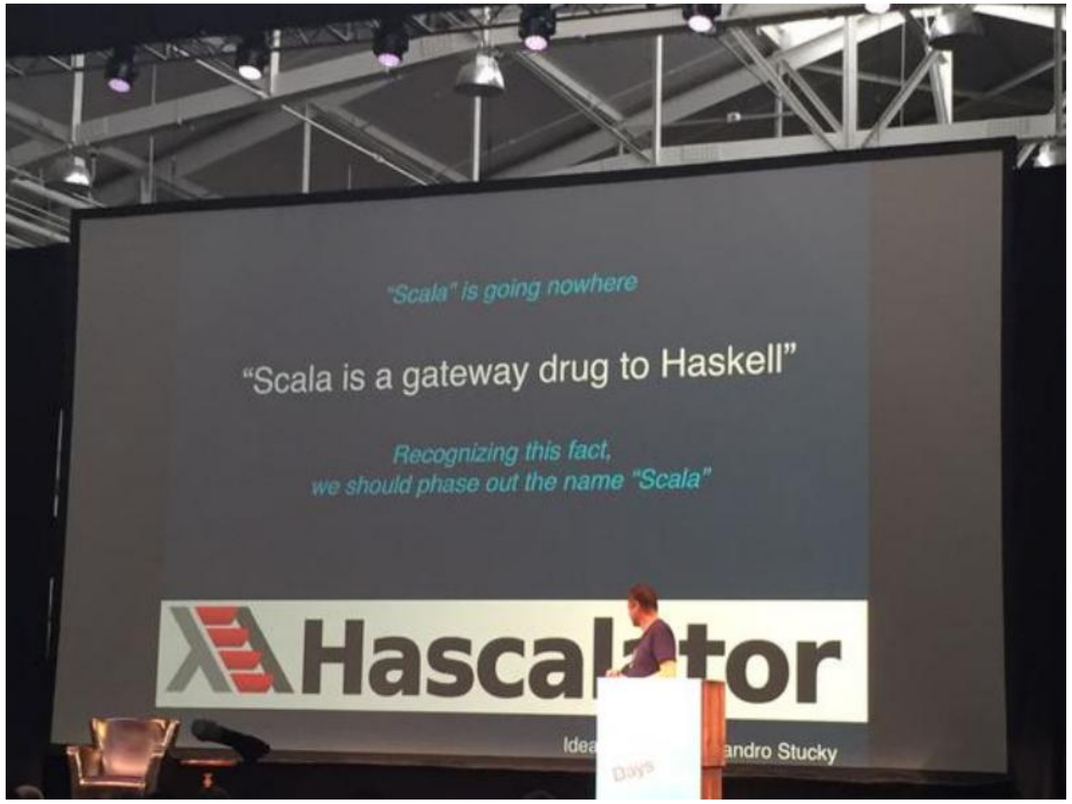
Scala基础

- Scala是一种混合编程语言，支持面向对象编程和函数式编程。
  - Scala是最热门的现代编程语言之一，被誉为编程语言中的凯迪拉克
  - 它支持函数式编程概念，如不可变数据结构和函数作为一等公民
  - Scala是一种非常适合开发大数据应用程序的语言
  - Scala的官网：http://scala-lang.org/
  - Scala下载地址：https://www.scala-lang.org/download/2.11.11.html
  - Scala API地址：https://www.scala-lang.org/api/2.11.11/

目标：不是成为Scala的专家，而是学习足够的Scala，以便使用Scala来理解和编写Spark应用程序

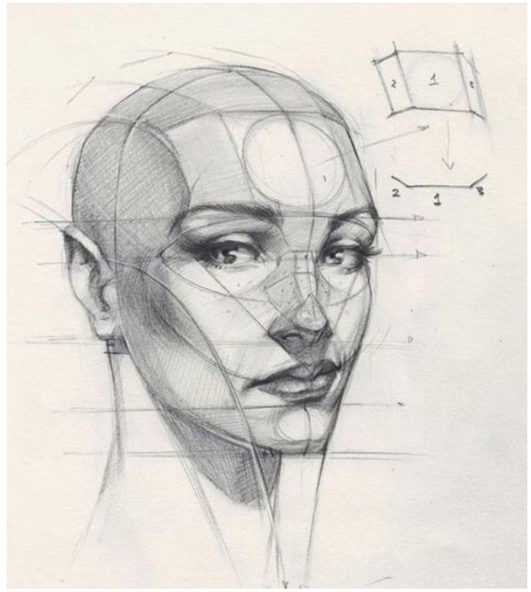
Scala基础

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



Scala基础

- 语言对比
  - 以下分别代表：C++，Python，Scala



C++

Python

Scala

Scala基础

- 使用Scala语言编写代码更简洁
  - 比如，获取订单中产品列表的代码：

```java
public List<Product> getProducts() {
    List<Product> products = new ArrayList<Product>();
    for (Order order : orders) {
        products.addAll(order.getProducts());
    }
    return products;
}
```

Java实现

```
def products = orders.flatMap(o => o.products)
```

甚至可以更简洁：

```
def products = orders.flatMap(_.products)
```

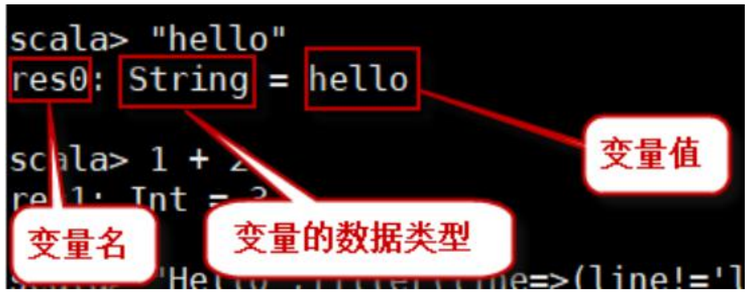
Scala实现

Scala基础

- 安装Scala
  - Scala可以安装在Windows和Linux操作系统下（要求先安装好JDK 8） - 2.12.14
  - Scala解释器: Scala解释器读到一个表达式，对它进行求值，将它打印出来，接着再继续读下一个表达式。这个过程被称做读取--求值--打印--循环，即：REPL。
  - Scala提供了一个REPL工具，叫做Scala Shell。

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

scala>
```

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

scala> 1 + 2
res1: Int = 3

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



Scala基础

- 使用IDE
  - 使用Scala IDE
  - IDE开发工具：http://scala-ide.org

Scala基本语法

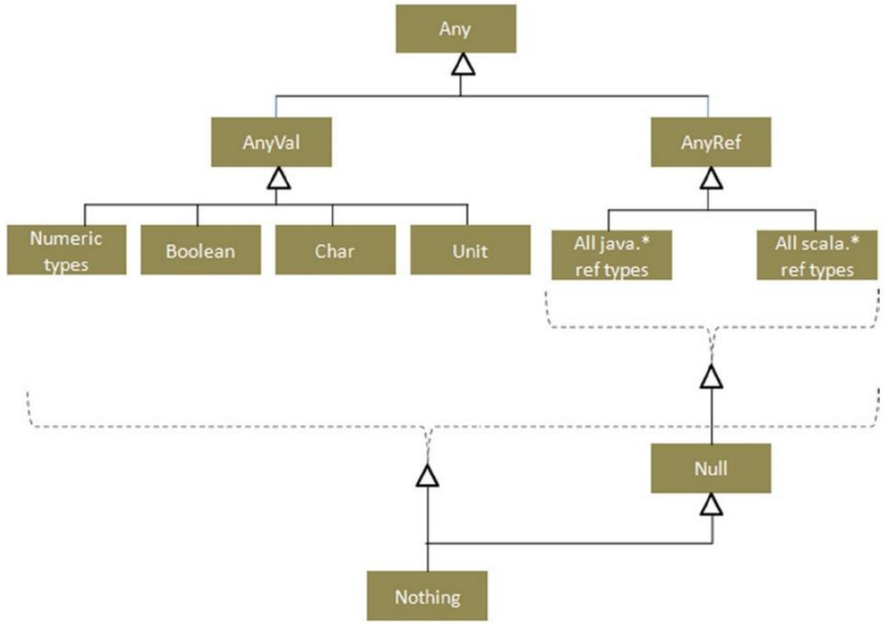
Scala基本语法

- Scala基本语法：变量
  - Scala有两种类型变量：可变的和不可变的。
  - 在Scala中，定义变量的方式有三种：val、var和lazy val。其中：
    - val：声明的变量是不可变的（只读的）。
    - var：声明的变量是可变的。在变量创建以后，可以重新赋值。
    - lazy val：变量只被计算一次，在该变量第一次被访问时。

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

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

scala>
```

Scala基本语法

- Scala基本语法：数据类型
  - 与Java语言类似，Scala 语言内置了基本的数据(变量)类型以及在这些类型上允许执行的运算符。
  - 与Java语言不同，Scala没有原始数据类型（基本数据类型）。在Scala中，所有的数据类型都是对象，这些对象具有操作其自身数据的方法。
  - Scala的所有数据类型，从数字到集合，都是类型层次结构的一部分



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

Scala基本语法

- Scala基本语法：运算符
  - Scala语言中提供了这几种运算符：算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符等。
  - 在实际进行计算的时候，经常会多个不同的运算符混合在一起进行运算
  - 下表中显示了Scala中不同运算符的优先级：

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

实际上，Scala没有传统意义上的运算符。在Scala中，一切皆对象，运算符是定义在对象上的方法。

Scala基本语法

- 数组
  - Scala中数组分为定长数组和变长数组。创建定长数组Array的两种方式：
    - 先创建，后赋值；
    - 创建的同时赋初值。
  - 例如，在下面的代码中，创建了一个名为names的数组，用来存储3个人的姓名：

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

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

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

val arrs1 = new Array[String](3)
arrs1(0)

val arrs2 = new Array[Boolean](3)
arrs2(0)
```

Scala程序流程控制

Scala程序流程控制

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

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

  - 在scala中，更经常是将if语句作为表达式来使用，在Scala中，每个表达式都有值：

```
val score = 60
if(score>=60) "及格" else "不及格"

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

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

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

输出结果如下：

```
score: Int = 60
res37: String =  及格
result: String =  及格
result1: Any =  及格
result2: Any = ()
```

Scala程序流程控制

- 循环结构
  - Scala拥有与Java和C++相同的while和do-while循环。
  - Scala 也有do-while循环，它和while循环类似，只是检查条件是否满足在循环体执行之后检查。
  - Scala也提供了for-each循环语句，可用来遍历数组

下面是使用 while 的循环语句：

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

下面是使用 do-while 的循环语句：

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

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

下面是使用 for-each 的语句：

```
val a = Array("apple", "banana", "orange")

for (e <- a) println(e)

for (e <- a) {
    val s = e.toUpperCase
    println(s)
}
```

Scala程序流程控制

- 循环中使用计数器
  - 使用for-each循环有一个缺点，就是无法获取当前索引值。为此，可使用循环计数器。
  - Scala集合还提供了一个zipWithIndex方法可用于创建循环计数器

```
val a = Array("张三","李四","王老五")
// 循环计数
for (i <- 0 until a.length) {
    println(s"$i：${a(i)}")
}
```

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

Scala程序流程控制

- 循环中使用条件过滤
  - 可以遍历的同时过滤
  - Scala可以使用一个或多个 if 语句来过滤一些元素

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

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

【示例】在 for 条件表达式中使用过滤。

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

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

// 多条件
for {
  i <- 1 to 10
  if i > 3
  if i < 6
  if i % 2 == 0
} println(i)
```

Scala程序流程控制

- 简单模式匹配
  - Scala中没有提供与Java语言中switch类似的语法，但是提供了一个更加强大的模式匹配功能。
  - 一个模式匹配的简单应用是作为多级if-else 语句的替代，这可以提高代码的可读性。
  - 模式匹配不使用关键字switch，Scala 使用关键字match。
  - 一个case语句也可以匹配多个条件。也可以匹配字符串

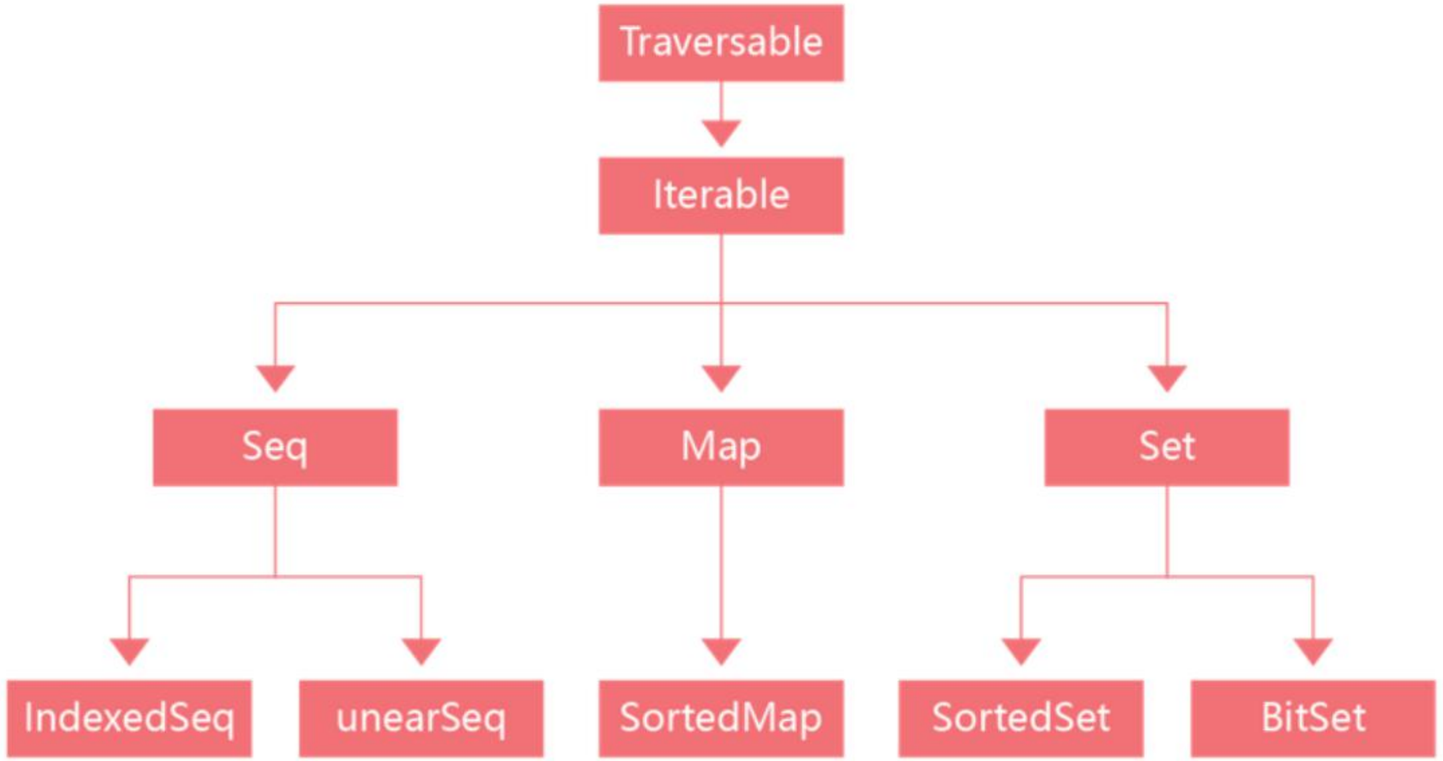
```
// 根据输入的数字，给出对应的星期
val day = 8
val month = day match{
      case 1 => "星期一"
      case 2 => "星期二"
      case 3 => "星期三"
      case 4 => "星期四"
      case 5 => "星期五"
      case 6 => "星期六"
      case 7 => "星期天"
      case _ => "不正确"
}
println(month)
```

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

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

集合

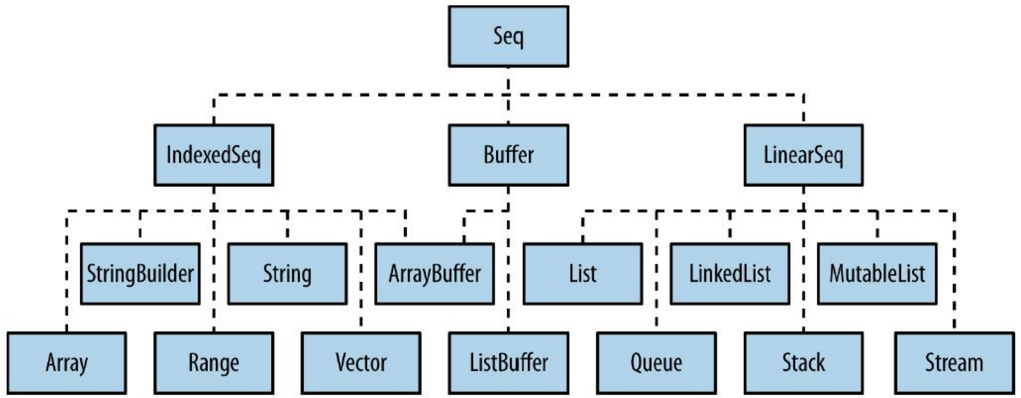
Scala常用集合类

- Scala 有一个丰富的集合库，包含很多不同类型的集合。
  - Scala中所有的集合都暴露出相同的接口。Scala中的集合体系主要包括：Iterable、Seq、Set、Map。
  - Scala中的集合是分成“可变”和“不可变”两类集合的，分别对应Scala中的scala.collection.mutable和scala.collection.immutable两个包。



Scala常用集合类

- Seq
  - 序列表示一个以特定顺序排列的元素集合。
  - 因为该元素有个定义好的顺序，所以可以按照它们在集合中的位置进行访问。
  - 序列有向个主要实现类，分别是Array、Range、List和Vector。




```
println("1: 初始化一个不可变的Seq")
// val list1: Seq[String] = Seq("苹果", "香蕉", "葡萄干")
val list1 = Seq("苹果","香蕉","葡萄干")   // 完全可以使用类型推
println(s"list1中的元素有 = $list1")

println("\n2: 在特定的索引访问元素的不可变列表")
println(s"Element at index 0 = ${list1(0)}")
println(s"Element at index 1 = ${list1(1)}")
println(s"Element at index 2 = ${list1(2)}")
println(s"last of list is = ${list1.last}")
println(s"take n of list is = ${list1.take(2)}")

println("\n3: List是一个递归数据结构")
println(s"head of list is = ${list1.head}")
println(s"tail of list is = ${list1.tail}")
```

```
println("\n4: 使用 :+ 在不可变列表末尾添加元素")
// val list2: Seq[String] = list1 :+ "草莓"
val list2 = list1 :+ "草莓"
println(s"在末尾追加元素，使用 :+ = $list2")

println("\n5: 初始化一个空的不可变列表")
// val emptyList: Seq[String] = Seq.empty[String]
val emptyList = Seq.empty[String]
println(s"Empty list = $emptyList")

println("\n6: 常用统计方法实现")
val numbers = Seq(1, 3, 5, 7, 9)
println(s"sum = ${numbers.sum}")
println(s"max = ${numbers.max}")
println(s"min = ${numbers.min}")
```

Scala常用集合类

- Array
  - 在Scala中，数组分为不可变数组和可变数组。
  - 不可变数组的实现类为Array，可变数组的实现类为ArrayBuffer。
  - 创建数组Array的两种方式：1) 先创建，后赋值；2) 创建的同时赋初值。
  - 在一个数组上的基本操作包含：1）根据索引提取一个元素；2）使用索引更新一个元素

```scala
// 创建的同时赋初值
val names = Array("张三","李四","王老五")
println(names(0))
println(names(1))
println(names(2))
```

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

Scala常用集合类

- List
  - 在Scala中，列表分为不可变的和可变的。不可变列表的实现类为List，可变列表的实现类为ListBuffer。
  - 在一个list 上的基本操作包括：
    - 1）head方法：提取第一个元素；
    - 2）tail方法：提取除第一个元素之外的所有元；
    - 3）isEmpty方法：判断一个list 是否为空。如果一个list 是空的，该方法返回true。

```
println("1: 初始化一个不可变的List")
// val list1: List[String] = List("苹果","香蕉","葡萄干")
val list1 = List("苹果","香蕉","葡萄干")    // 完全可以使用类型推断
println(s"list1中的元素有 = $list1")

println("\n2: 在特定的索引访问元素的不可变列表")
println(s"Element at index 0 = ${list1(0)}")
println(s"Element at index 1 = ${list1(1)}")
println(s"Element at index 2 = ${list1(2)}")

println("\n3: List是一个递归数据结构")
println(s"head of list is = ${list1.head}")
println(s"tail of list is = ${list1.tail}")
```

```
println("\n4: 使用 :+ 在不可变列表末尾添加元素")
// val list2: List[String] = list1 :+ "草莓"
val list2 = list1 :+ "草莓"
println(s"在末尾追加元素，使用 :+ = $list2")

println("\n5: 使用 +: 在不可变列表的前面添加元素")
val list3: List[String] = "菠萝" +: list1
// val list3 = "菠萝" +: list1
println(s"在不可变列表前端添加元素，使用 +: = $list3")

println("\n6: 使用 :: 将两个不可变列表添加在一起")
// val list4: List[Any] = list1 :: list2
val list4 = list1 :: list2
println(s"添加两个列表一起，使用 :: = $list4")
```

Scala常用集合类

- Vector
  - 根据Scala文档，Vector是类似于List的数据结构。但是，它解决了List中随机访问的低效问题。
  - 它结合了Array 和List 的执行特性，提供了恒定时间复杂度的索引访问和线性访问。
  - 它既允许快速的随机访问，也允许快速的更新功能。

```
println("1: 初始化一个具有3个元素的向量")
val vector1: Vector[String] = Vector("苹果","香蕉","葡萄干")
println(s"vector1的元素 = $vector1")

println("\n2: 访问向量在特定索引的元素")
println(s"索引0的元素 = ${vector1(0)}")
println(s"索引1的元素 = ${vector1(1)}")
println(s"索引2的元素 = ${vector1(2)}")

println("\n3: 使用:+ 在Vector末尾添加元素")
val vector2 = vector1 :+ "火龙果"
println(s"使用:+添加元素的向量 = $vector2")
```

```
println("\n4: 使用+:在Vector前面添加元素")
val vector3 = "草莓" +: vector1
println(s"使用+:在Vector前面添加元素的向量 = $vector3")

println("\n5: 使用++ 将两个向量添加在一起")
val vector4 = vector1 ++ Vector[String]("水蜜桃")
println(s"使用++将两个向量添加到一起后 = $vector3")

println("\n6: 初始化一个空向量")
val emptyVector: Vector[String] = Vector.empty[String]
println(s"String类型的空向量 = $emptyVector")
```

Scala常用集合类

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

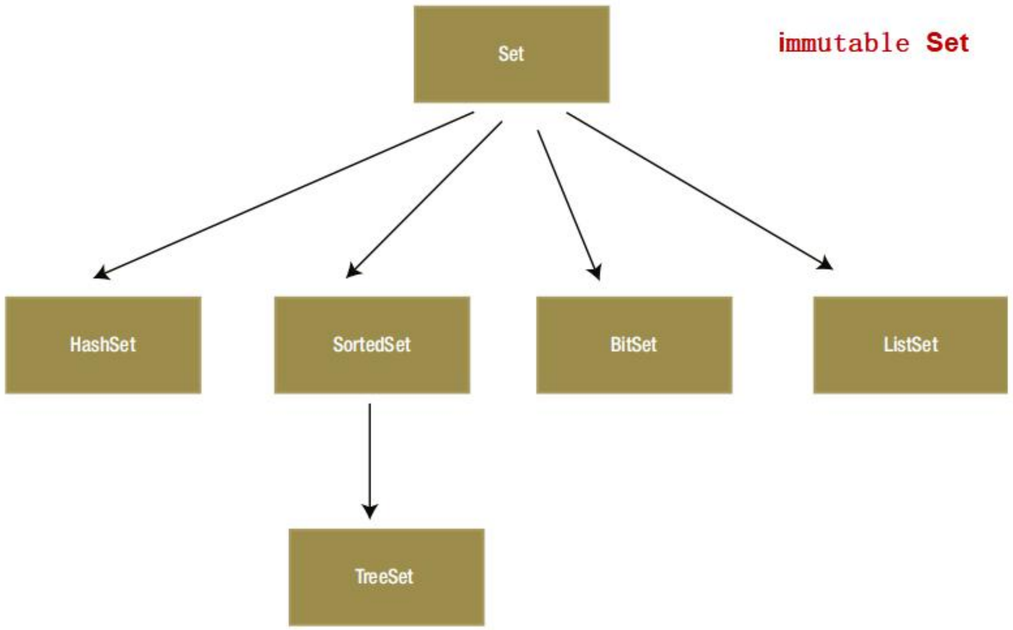
```
// 使用方法to来创建Range(包含上限)
for(n <- 1 to 5){
  println(n)
}

(1 to 5).foreach(println)

println("\n----------------------------")
// 使用方法until来创建Range(不包含上限)
for(n <- 1 until 5){
  println(n)
}
(1 until 5).foreach(println)
```

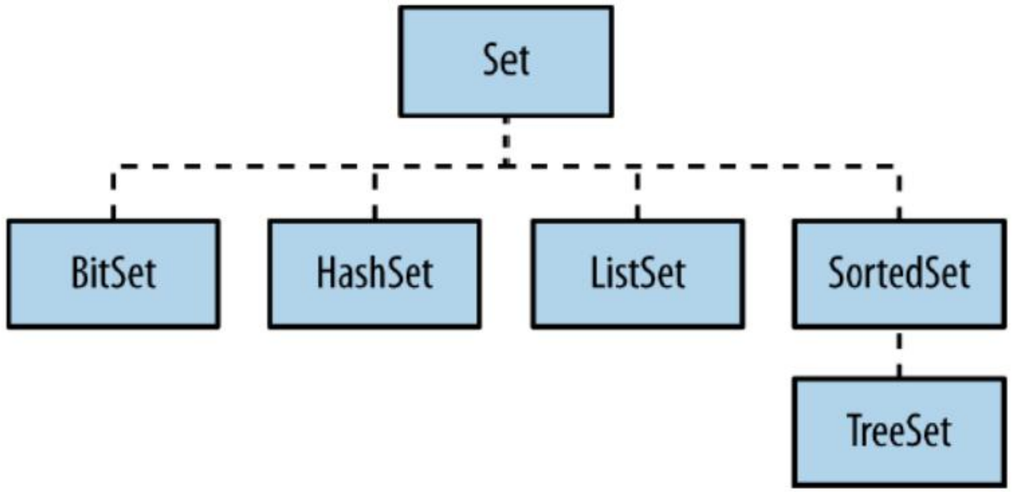
```
// 还可以指定步长
for(n <- 1 to 21 by 4){
  println(n)
}

println("\n----------------------------")
for(n <- 1 until 21 by 4){
  println(n)
}

println("\n----------------------------")
for(n <- 'a' to 'c'){
  println(n)
}
```

```
println("\n----------------------------")
// 读取值
val r = 1 to 5
// val r = 1.to(5)      // 等价
println(r(0),r(1),r(2),r(3),r(4))

println("\n----------------------------")
// 使用Range创建一个连续序列
val x = (1 to 10).toList
x.foreach(println)
```

Scala常用集合类

- Set
  - Set 是一个不重复元素的无序集合。它不包含重复元素。尝试将已有元素加入进来是没有效果的。
  - 此外，它不允许通过索引访问一个元素，因为它并没有索引。
  - Set不保留元素插入的顺序。默认情况下, Set是以HashSet实现的, 其元素根据hashCode方法的值进行组织。





Scala常用集合类

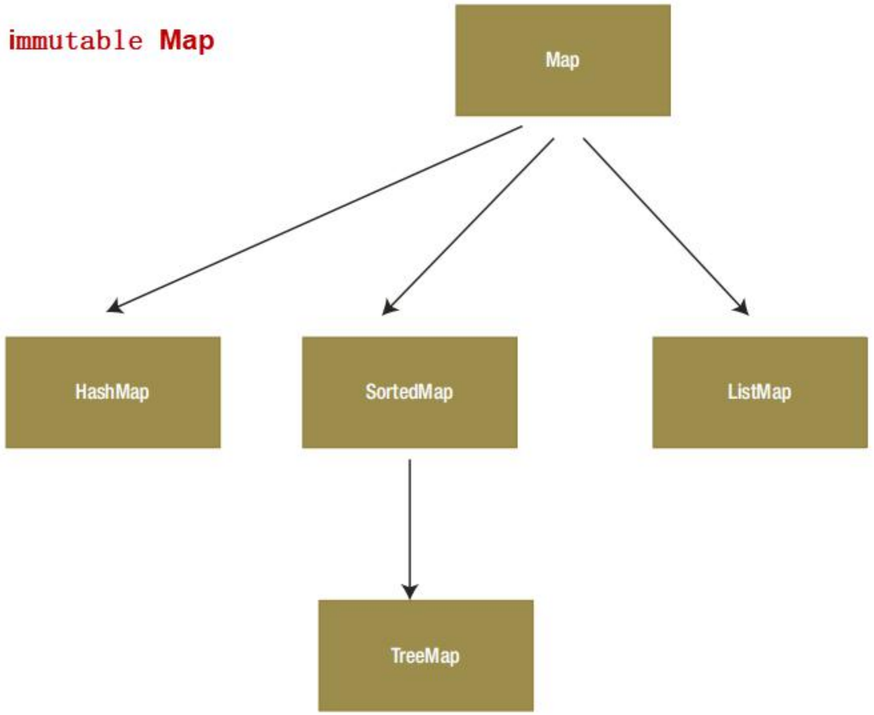
- Set
  - Sets 支持两个基本操作：
    - contains：如果set 包含输入参数，则返回true；
    - isEmpty：如果set 是空的，返回true；

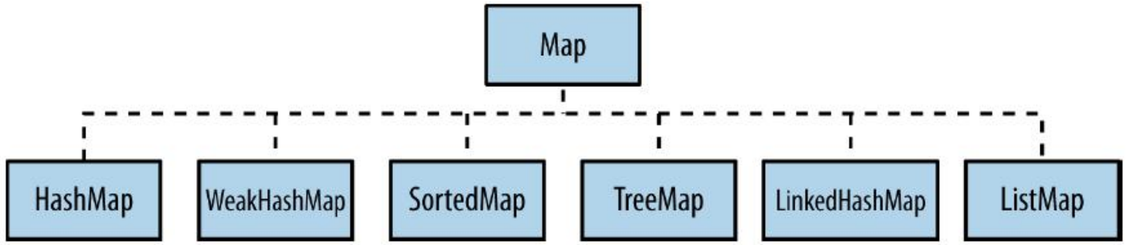
```
println("1: 初始化一个有3个元素的集合")
val set1: Set[String] = Set("苹果","香蕉","葡萄干","苹果")
println(s"set1的元素 = $set1")

println("\n2: 检查集合中存在的特定元素")
println(s"是否有苹果 = ${set1("苹果")}")
println(s"是否有香蕉 = ${set1("香蕉")}")
println(s"是否有葡萄干 = ${set1("葡萄干")}")
println(s"是否有火龙果 = ${set1("火龙果")}")
```

```
println("\n3: 使用+ 在Set 中添加元素")
val set2: Set[String] = set1 + "火龙果" + "葡萄干"
println(s"使用 + 向Set集合中添加元素 = $set2")

println("\n4: 使用 ++ 将两个Set集合添加到一起")
val set3: Set[String] = set1 ++ Set[String]("火龙果", "葡萄干")
println(s"使用 ++ 将两个Set集合添加到一起 = $set3")

println("\n5: 使用 - 从Set 中删除元素")
val set4: Set[String] = set1 - "苹果"
println(s"去掉苹果后的Set集合元素 = $set4")

println("\n6: 使用& 找到两个集合之间的交集")
val set5: Set[String] = Set("火龙果", "香蕉", "苹果")
println(s"set1和set5的交集 = ${set1 & set5}")

println("\n7: 使用 &~ 找出两个集合的差集")
println(s"set1和set5的差集 = ${set1 &~ set5}")

println("\n8: 初始化一个空集")
val emptySet: Set[String] = Set.empty[String]
println(s"Empty Set = $emptySet")
```

Scala常用集合类

- Map
  - Map是一个key-value 对的集合。在其它语言中，它被称为词典、关联数组、或HashMap。
  - 这是一个根据key查找value的高效的数据结构。
  - Map的常见实现类有：HashMap, TreeMap, ListMap等





Scala常用集合类

- Map
  - Map是一个根据key查找value的高效的数据结构。
  - 下面演示了Map的基本用法。

```
println("1: 初始化一个带有3个元素的Map - 使用key/value元组")
// val map1: Map[String, String] = Map(("PG","苹果"),("XG","西瓜"),("PTG","葡萄干"))
val map1 = Map(("PG","苹果"),("XG","西瓜"),("PTG","葡萄干")) // 类型推断
println(s"map1的元素 = $map1")

println("\n2: 使用key->value表示法初始化Map")
val map2: Map[String, String] = Map("PG" -> "苹果", "XG" -> "西瓜", "PTG" -> "葡萄干")
// val map2 = Map("PG" -> "苹果", "XG" -> "西瓜", "PTG" -> "葡萄干")
println(s"map2的元素 = $map2")

println("\n3: 通过特定键访问元素")
println(s"Key是PG的元素 = ${map2("PG")}")
println(s"Key是XG的元素 = ${map2("XG")}")
println(s"Key是PTG的元素 = ${map2("PTG")}")
println(s"Key是HLG的元素 = ${map2.get("HLG")}")      // 访问不存在的key
println(s"Key是HLG的元素 = ${map2.getOrElse("HLG","火龙果")}")

println("\n4: 使用 + 添加元素")
// val map3: Map[String, String] = map1 + ("HLG" -> "火龙果")
val map3: Map[String, String] = map1 + ("PTG" -> "火龙果") // 也可用来修改元素值
println(s"map3 = $map3")

println("\n5: 使用 ++ 将两个Map 添加到一起")
val map4: Map[String, String] = map3 ++ map2
println(s"map4 = $map4")

println("\n6: 使用 - 从Map 删除key及其value")
val map5: Map[String, String] = map4 - "PG"
println(s"Map移除了PG key及其value = $map5")

println("\n7: 初始化一个空的Map")
val emptyMap: Map[String,String] = Map.empty[String,String]
println(s"Empty Map = $emptyMap")
```

Scala常用集合类

- Tuple
  - 一个tuple(元组)是一个容器，用来存储两个或多个不同类型的元素。
  - 元组与List类似，大小和值不可变，它在创建以后不能被修改，但可以容纳不同数据类型。
  - 可以通过两种方式创建元组:
    - 通过编写由一对括号包围的、包含用逗号分开的值
    - 通过使用关系操作符(->)
  - 元组是不可迭代的，其目的只是作为能容纳多个值的容器。但在一个元组中的元素有一个基于1 的索引，可以通过索引访问元组中的元素。

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

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

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

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

Scala常用集合类

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

高阶方法	说明
map	Scala 集合的map 方法将其输入函数应用到集合中所有元素上，并返回另一个集合
flatMap	Scala 集合的flatMap 方法类似于map。它接收一个函数作为输入，并将该输入函数应用到集合中的每个元素，并返回另一个集合作为结果。
filter	使用filter方法，按照过滤条件，将原集合中不符合条件的数据过滤掉，输出所有匹配某个特定条件的元素
foreach	Scala 集合的foreach 方法在集合的每个元素上调用其输入函数，但并不返回任何东西
reduce	方法reduce 返回单个值。正如其名所暗示的，它将一个集合归约为一个单个的值。方法reduce 的输入函数同时接收两个输入并返回一个值。
mapValues	对于 (key,value) 类型的元素，只对 value 部分进行 map 转换
sortWith	使用sortWith函数对集合进行排序

Scala常用集合类

- for/yield组合创建新集合
  - 在Scala的for表达式中，可以使用yield关键字来生成新的集合
  - yield的主要作用是记住每次迭代中的有关值，并逐一存入到一个集合中，循环结束后将返回该集合。
  - 其语法如下：for {子句} yield {变量或表达式}

```
var a = 0;
val numList = List(1,2,3,4,5,6,7,8,9,10);

// for 循环
var retVal = for( a <- numList
                  if a != 3; if a < 8
                ) yield a

// 输出返回值
for( a <- retVal){
      println( "a = " + a );
}
```

yield关键字：
1)针对每一次for循环的迭代，yield会产生一个值，被循环记录下来；
2)当循环结束后，会返回所有yield的值组成的集合；
3)返回集合的类型与被遍历的集合类型是一致的。

Option类型

- 在Scala中，Option[T]是给定类型的0或1个元素的容器。
  - Option 是一个数据类型，用来表示可能存在也可能不存在的值。
  - Option类型可以是Some[T]或None，其中T可以是任何给定类型。
  - 一个Some 实例可以存储任何类型的数据。None 对象表示数据的缺失。
  - Option 数据类型用来防止null 指针异常。在许多语言中，null 被用来表示数据缺失。在Scala 中联合使用强类型检查和Option 类型以防止null 引用错误。
  - Option类型提供有一个getOrElse方法。当没有值时，使用getOrElse()来访问值或使用默认值，可以得到友好的反馈。另外还提供了一个isEmpty()方法，用来判断Option中是否有值

```scala
val a: Option[String] = Some("aaaa")
println(a.get)      // "aaaa"

val b: Option[String] = None
// println(b.get)    // 抛出异常
println(b.getOrElse(""))

val c: Option[String] = Some(null)
println(c.get)      // null
```

```scala
// Option类型的模式匹配
val title:Option[String] = Some("老板")
// val title:Option[String] = None
title match{
  case Some(a) => println(s"title是$a。")    // 提取
  case None => println(s"没有title，屌丝一个。")
}

// 巧用map/foreach函数
title.foreach(t => println(s"我的title是 = $t"))
```

Scala常用集合类

- 作业：词频统计
  - 需求描述：读取/加载多段文本内容，请统计其中每个单词出现的频次。 - 单词计数
  - 要求：将统计结果按单词出现频率由多到少输出。
  - 程序结构如下：

```
object WordCount {
  def main(args: Array[String]): Unit = {
    // 需要进行统计的文本
    val text = List("good good study","day day up")

    // 请在这里实现词频统计的代码 ......
  }
}
```

  - 期待输出结果

```
(day,2)
(good,2)
(study,1)
(up,1)
```

函数

Scala函数

- Scala是一个函数语言，它将函数当作一等公民
  - 一个函数可以像一个变量一样被使用。
  - 函数可以作为输入参数传给另一个函数。
  - 函数可以定义为一个匿名函数字面量，就像字符串字面量
  - 可以在一个函数内定义函数。
  - 函数可以作为另外一个函数的输出返回值。
  - 在Scala中，一切皆对象，因此函数也必是对象。

Scala函数

- 函数字面量
  - 函数字面量指的是在源代码中的一个未命名函数或匿名函数。
  - 在程序中可以像使用一个字符串变量一样使用它。另外，它也可以被赋给一个变量。
  - 它还可以作为一个输入参数传递给一个高阶方法或高阶函数。
  - 函数字面量经常作为高阶函数的输入参数
  - 函数执行以后可以有返回值。在Scala中，函数体的最后一行表达式的值，就是函数返回值

```
// 最简单的函数
() => println("hello")     // 无参

(i:Int) => {println("Hello"); println(i * i); i * i}     // 传入一个 int 参数
```

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

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

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

Scala函数

- 函数方法
  - 也可以使用def关键字来定义有名字的函数。在Scala 中，使用关键字def 定义函数，其语法格式如下:

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

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

```
// 定义方法 add
def add(x:Int,y:Int):Int = {
    println("这是一个对两个整数进行求和的函数")
    x + y
}

// 通过函数名来调用函数（方法）
add(3,5)
```

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

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

Scala函数

- 函数参数
  - 函数定义时指定的参数称为"形参"，函数被调用时实际传入的参数称为"实参"。
  - 默认在函数调用时，实参和形参是按顺序一一对应的，否则有可能出现错误。
  - 为了避免这种错误，可以在调用函数时指定参数名称（这称为名称参数）
  - 也可以混用未命名参数和名称参数，只要那些未命名的参数是排在前面的即可
  - 也可以在定义函数时指定默认参数。如果指定了默认参数，那么在调用函数时如果没有为该参数传入实参值，则默认传入的是默认参数。
  - Scala在定义函数时允许指定最后一个参数可以重复（变长参数），从而允许函数调用者使用变长参数列表来调用该函数。

```
def func3(name:String, age:Int) : String = {
      println("name=> " + name + "\t age=> " + age)
      "欢迎" + name + "光临，您的年龄是 => " + age
}

func3("张三",23)            // OK
```

Scala函数

- 高阶函数
  - 在scala中，函数可以作为参数来传递
  - 能接收函数参数的函数，或者返回值为函数的函数，称为"高阶函数"

```scala
// 定义一个高阶函数
def operation(func:(Int, Int) => Int) = {
  val a = 3
  val b = 5
  val result = func(a,b)
  println(s"$a + $b = $result")
}
```

```scala
def main(args: Array[String]): Unit = {
  // 接下来，定义一个与预期签名匹配的函数：
  val add = (x: Int, y:Int) => { x + y }

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

  // 或者也可以简写如下
  operation((x: Int, y:Int) => { x + y })
}
```

Scala函数

- 闭包
  - 闭包是一个函数，其返回值依赖于在该函数外声明的一个或多个变量的值。
  - Scala编译器创建一个闭包，它包含封闭作用域中的变量。
  - 函数的变量不在其作用域内被调用，就是闭包的概念

```
var y = 3
val multiplier = (x:Int) => x * y
multiplier(2)
```

```
def closePackage: Unit ={
      def mulBy(factor:Double) = (x:Double) => factor * x
      val triple = mulBy(3)
      val half = mulBy(0.5)
      println(triple(14) +" " + half(14)) //42, 7
}
```

Scala函数

- 函数柯里化
  - 柯里化（Curring）转换使用多个参数的函数，创建一个函数链，链中的每个函数都期望有一个参数。
  - 例如，有一个函数func(a,b)，它有两个参数a和b。对其进行柯里化转换之后，变为func(a)(b)。
  - 因此，柯里化指的是将原来接受2个参数的函数变成新的接受一个参数的函数的过程。新的函数返回一个以原有第二个参数作为参数的函数。
  - 在柯里化以后，在函数调用的过程中，就变为了两个函数连续调用的形式。在Spark源码中，也有体现，所以对()()这种形式的柯里化函数，一定要掌握。

```
// 普通函数
val add = (x: Int, y: Int) => x + y
add(3,5)

// 在 Scala 中，curried 函数定义为多个参数列表，如下:
def add(x: Int)(y: Int) = x + y
add(4)(6)

// 还可以使用如下语法来定义一个柯里化函数
def add(x: Int) = (y: Int) => x + y        // 函数返回一个函数
add(7)(8)
```

函数式编程

Scala函数式编程

- 函数是一等公民
  - 在函数式编程中，应用程序完全是由函数组装而成的。
  - 尽管许多编程语言支持函数的概念，但函数式编程语言将函数视为一等公民。
  - 此外，在函数式编程中，函数是可组合的，没有副作用。
  - 在函数式编程中，函数是一等公民，意味着函数具有与变量或值相同的状态，可以像使用变量一样使用函数（例如，将函数赋给变量）
  - FP允许将一个函数作为输入传递给另一个函数。它允许一个函数作为另一个函数的返回值返回。一个函数可以在任何地方定义，包括在另一个函数内部。它可以定义为未命名的函数字面量，就像字符串字面量一样，并作为输入传递给函数。
  - 函数编程中的函数是可组合的。将简单的函数组合成复杂的函数。
  - 函数编程中的函数没有副作用。函数返回的结果只依赖于函数的输入参数。函数的行为不会随时间而改变。对于给定的输入，无论调用多少次，它每次都返回相同的输出。

Scala函数式编程

- 不可变的数据结构
  - 函数式编程强调不可变数据结构的使用。纯函数式程序不使用任何可变的数据结构或变量。
  - 换句话说，与命令式编程语言(如C/ c++、Java和Python)不同，数据永远不会被就地修改。
  - 不可变的数据结构提供了许多好处。
    - 首先，它们减少了bug。使用不可变数据结构编写的代码很容易推理。
    - 此外，函数式语言提供了允许编译器强制不变性的构造。因此，许多错误是在编译时捕获的。
    - 其次，不可变的数据结构使得编写多线程应用程序更加容易。编写一个利用所有核的应用程序并不容易。竞争条件和数据损坏是多线程应用程序的常见问题。使用不可变数据结构有助于避免这些问题。

Scala函数式编程

- 一切都是表达式
  - 在函数式编程中，每条语句都是一个返回值的表达式。
  - 例如，Scala中的if-else控制结构是一个返回值的表达式。
  - 这种行为不同于命令式语言，命令式语言中，只能在if-else中对一组语句进行分组。
  - 这个特性对于编写没有可变变量的应用程序非常有用。

面向对象编程

Scala面向对象编程

- 类的定义
  - 在Scala 中类与其它面向对象语言相似。
  - 类定义由字段声明和方法定义组成。字段是一个变量，用于存储对象的状态（数据），方法包含可执行代码，它是定义在类中的一个函数，可以提供对字段的访问，并更改对象的状态。
  - 使用关键字class 定义一个类。使用关键字new 创建一个类的实例。

```
class Car(mk:String, ml:String, cr:String){
    val make = mk;
    val model = ml;
    val color = cr;

    def repaint(newColor:String) = {
        color = newColor;
    }
}
```

```
val mustang = new Car("福特","Mustang","红色");
val corvette = new Car("通用","Corvette","黑色");
```

Scala面向对象编程

- 单例对象
  - 在面向对象编程中一个常见的设计模式是定义一个只能被实例化一次的类。
  - 一个只能被实例化一次的类叫做“单例(singleton)”。
  - Scala 提供关键字object 用于定义一个单例类。
  - 使用关键字object来定义单例对象。object是只有一个实例的类。当它被引用时，是延迟创建的。
  - 下面的代码定义一个单例对象：

```
// 定义单例对象
object HelloWorld{
    def greet(){
        println("Hello World!")
    }
}

// 调用
HelloWorld.greet
```

Scala面向对象编程

- 伴生对象
  - 当单例对象与某个类共享同一个名称时，它就被称为是这个类的伴生对象。类和它的伴生对象必须定义在同包同一个源文件里。类被称为是这个单例对象的伴生类。类和它的伴生对象可以互相访问其私有成员。
  - 从Java程序员的角度，可以把单例对象当作是Java中可能会用到的静态方法工具类。（在Scala类中不能定义静态成员）

```
// 伴生类
class Fruit(name: String, productCode: Long) {
  def print = println(s"水果名=$name, 产品编码=$productCode")
}

// 伴生对象
object Fruit{
  def apply(name: String, productCode: Long): Fruit = new Fruit(name, productCode)
}
```

Scala面向对象编程

- 类的继承
  - 继承是一种描述类与类之间的关系，反映的是is a这种关系，一个类是另外一种类型。
  - 子类通过关键字extends继承了父类的字段和方法，同时可以自定义相应的字段和方法。

```
// 基类
class Vehicle(speed:Int){
      val mph:Int = speed
      def race() = println("Racing")
}

// 派生类:只有主构造函数可以将参数传递给基构造函数
class Car(speed:Int) extends Vehicle(speed){
      override val mph:Int = speed
      override def race() = println("Racing Car")
}

// 另一个子类
class Bike(speed:Int) extends Vehicle(speed){
      override val mph:Int = speed
      override def race() = println("Racing Bike")
}
```

```
// 对象
val vehicle1 = new Car(200)
vehicle1.mph
vehicle1.race

val vehicle2 = new Bike(100)
vehicle2.mph
vehicle2.race
```

Scala面向对象编程

- Trait
  - 一个trait代表一个接口，由相关类的层级所支持。它是一个抽象机制，帮助开发模块化、可重用和可扩展的代码。
  - Scala trait 与Java 的接口类似。不过Java 中的接口只包括方法签名，而一个Scala 的trait 可以包含方法的实现。
  - 此外，一个Scala trait 也可以包含字段。一个类可以重用在trait 中实现的这些字段和方法。

```scala
// 定义一个trait，提供DAO层的方法签名
trait ProductShoppingCartDao{
  def add(productName: String): Long
  def update(productName: String): Boolean
  def search(productName: String): String
  def delete(productName: String): Boolean
}
```

```scala
// 创建一个ProductShoppingCart类，它扩展上面的trait并实现其方法
class ProductShoppingCart extends ProductShoppingCartDao {

  override def add(productName: String): Long = {
    println(s"ProductShoppingCart-> add method -> productName: $productName")
    1
  }

  override def update(productName: String): Boolean = {
    println(s"ProductShoppingCart-> update method -> productName: $productName")
    true
  }

  override def search(productName: String): String = {
    println(s"ProductShoppingCart-> search method -> productName: $productName")
    productName
  }

  override def delete(productName: String): Boolean = {
    println(s"ProductShoppingCart-> delete method -> productName: $productName")
    true
  }
}
```

```scala
// 主类
object TraitDemo2 {
  def main(args: Array[String]): Unit = {
    // 现在我们可以创建ProductShoppingCart的实例
    val productShoppingCart1: ProductShoppingCart = new ProductShoppingCart()

    // 并调用相应的add、update、search和delete方法
    productShoppingCart1.add("苹果")
    productShoppingCart1.update("苹果")
    productShoppingCart1.search("苹果")
    productShoppingCart1.delete("苹果")

    println("----------------------------------------------")
    // 因为我们的ProductShoppingCart类扩展了特性ProductShoppingCartDao，
    // 也可以将ProductShoppingCart对象的类型分配给trait ProductShoppingCartDao
    val productShoppingCart2: ProductShoppingCartDao = new ProductShoppingCart()
    productShoppingCart2.add("苹果")
    productShoppingCart2.update("苹果")
    productShoppingCart2.search("苹果")
    productShoppingCart2.delete("苹果")
  }
}
```

Scala面向对象编程

- 样例类(case class)
  - 一个样例类是带有一个case 修饰符的类。示例如下：
    - **case class Message(from:String, to:String, content:String)**
  - 一个样例类相当于一个JavaBean风格的域对象，带有getter和setter方法，以及构造器、hashCode、equals和toString方法。但是可以不使用关键字new 来创建一个类实例。例如，下面的代码是有效的：
    - **val request = Message("harry","sam","fight");**
  - Scala 添加如下四个方法到一个样例类：toString, hashCode, equals 和copy
  - 样例类是一种特殊的类，经过优化以用于模式匹配。

```
case class Stuff(name:String, var age:Int)

val s = Stuff("张三",45)

s.toString
s.hashCode
s == Stuff("张三",45)
s == Stuff("张三",43)
s.name
s.age
s.age = s.age + 1
```

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

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

除此之外，case class与其它普通的scala类没有区别。

Scala面向对象编程

- 模式匹配
  - 模式匹配是一个表达式，因此它会导致一个值，该值可能被分配或返回。
  - 模式匹配也可以对两个case class进行匹配
  - 模式匹配可作为参数传递给其他函数或方法，编译器会将模式匹配编译为函数

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

也可以匹配字符串，例如：

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

```
// 定义一个case class
case class Person(name: String, age: Int, salary: Double)

// 判断方法
def older(person: Person): Option[Boolean] = person match {
  case Person(_, age, _) if age > 35 => Some(true)
  case _ => None
}
```

```
def main(args: Array[String]): Unit = {
  // 有一批员工信息
  val peoples = List(
    Person("张三", 40, 23000.00),
    Person("李四", 25, 18000.00),
    Person("王老五", 45, 35000.00))

  // 指出“老”员工（即年龄超过35岁的员工）
  peoples
    .filter(p => older(p).getOrElse(false))
    .foreach(println)
}
```

小结

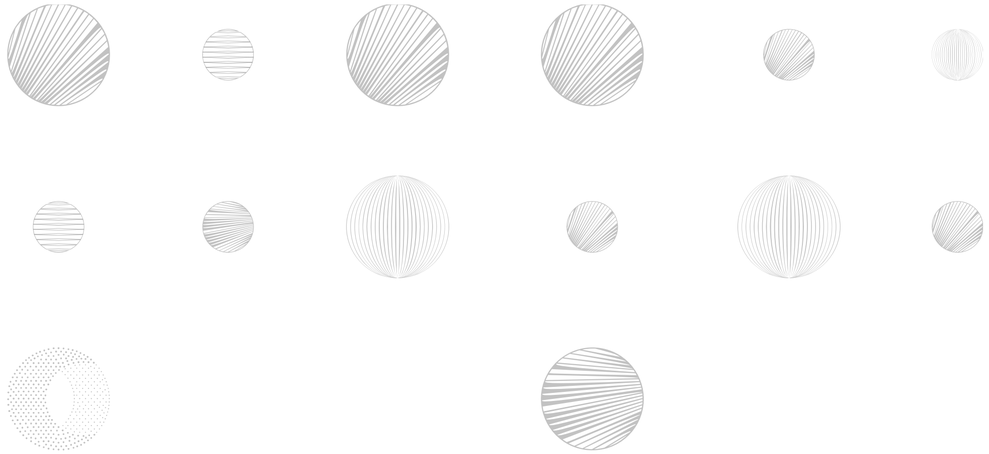
- 本章知识点小结
  - Scala是一种混合编程语言，支持面向对象编程和函数式编程。
  - Scala是最热门的现代编程语言之一，被誉为编程语言中的凯迪拉克。
  - 它支持函数式编程概念，如不可变数据结构和函数作为一等公民。
  - 对于面向对象编程，它支持类、对象和特征等概念。它还支持封装、继承、多态性和其他重要的面向对象概念。
  - Scala 有一个丰富的集合库，包含很多不同类型的集合。此外，所有的集合都暴露出相同的接口。因此，一旦熟悉了一个Scala 集合，就可以很容易地使用其它集合类型。
  - Scala中的集合体系主要包括：Iterable、Seq（IndexSeq）、Set（SortedSet）、Map（SortedMap）。
  - Scala是一个函数语言，它将函数当作一等公民；在Scala中，一切皆对象，因此函数也必是对象。

巩固与提高

- 请完成以下练习：
  - 练习1：使用Scala语言实现单词计数。
  - 练习2：Web日志信息提取
    - 有一行Web日志信息：

      199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] “GET /history/apollo/ HTTP/1.0” 200 6245

    - 请提取其中的用户IP地址及访问的URL。
    - 提示：请使用字符串正则表达式提取。使用方法可参考文档 scala.util.matching.Regex。



THANKS !