



Scala核心编程

-数据结构（应用）

尚硅谷研究院

数据结构(上)-集合

- 数据结构特点

scala集合基本介绍

- 1) Scala同时支持**不可变集合**和**可变集合**，不可变集合可以安全的并发访问
- 2) 两个主要的包：

不可变集合： `scala.collection.immutable`

可变集合： `scala.collection.mutable`

- 3) 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(`mutable`)和不可变(`immutable`)的版本
- 4) Scala的集合有三大类：序列Seq、集Set、映射Map，所有的集合都扩展自**Iterable**特质，在Scala中集合有可变（`mutable`）和不可变（`immutable`）两种类型。



图片1.z

● 数据结构特点

可变集合和不可变集合举例

- 1) 不可变集合: scala不可变集合, 就是这个**集合本身**不能动态变化。(类似java的数组, 是不可以动态增长的)
- 2) 可变集合: 可变集合, 就是这个**集合本身**可以动态变化的。(比如: ArrayList, 是可以动态增长的)

//不可变集合类似java的数组

```
int[] nums = new int[3];
```

```
nums[2] = 11; //?
```

```
//nums[3] = 90; //?
```

```
String[] names = {"bj", "sh"};
```

```
System.out.println(nums + " " + names);
```

//可变集合举例

```
ArrayList al = new ArrayList<String>();
```

```
al.add("zs");
```

```
al.add("zs2");
```

```
System.out.println(al + " " + al.hashCode()); //地址
```

```
al.add("zs3");
```

```
System.out.println(al + " " + al.hashCode()); //地址
```



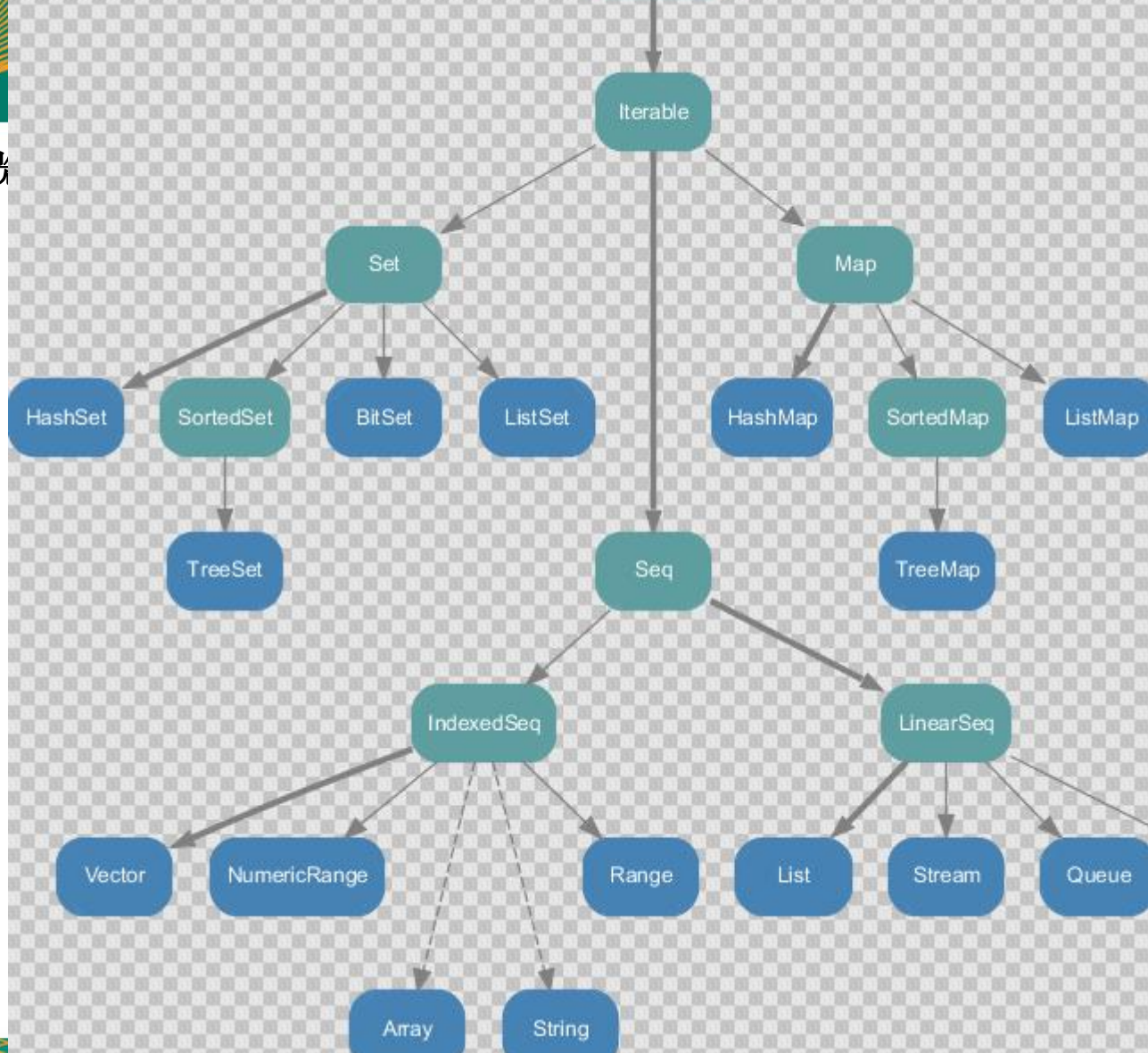

● 不可变集合继承层次一览

Scala不可变集合 继承关系一览表 老师小结:



图片1.z

1. Set、Map是Java中也有的集合
2. Seq是Java没有的，我们发现List归属到Seq了，因此这里的List就和java不是同一个概念了
3. 我们前面的for循环有一个 1 to 3，就是IndexedSeq 下的Vector
4. String也是属于IndexSeq
5. 我们发现经典的数据结构比如 Queue 和 Stack被归属到LinearSeq
6. 大家注意Scala中的Map体系有一个 SortedMap,说明Scala的Map可以支持排序
7. IndexedSeq 和 LinearSeq 的区别
[IndexedSeq是通过索引来查找和定位，因此速度快，比如String就是一个索引集合，通过索引即可定位]
[LinearSeq 是线型的，即有头尾的概念，这种数据结构一般是通过遍历来查找，它的价值在于应用到一些具体的应用场景 (电商网站, 大数据推荐系统:最近浏览的10个商品)]





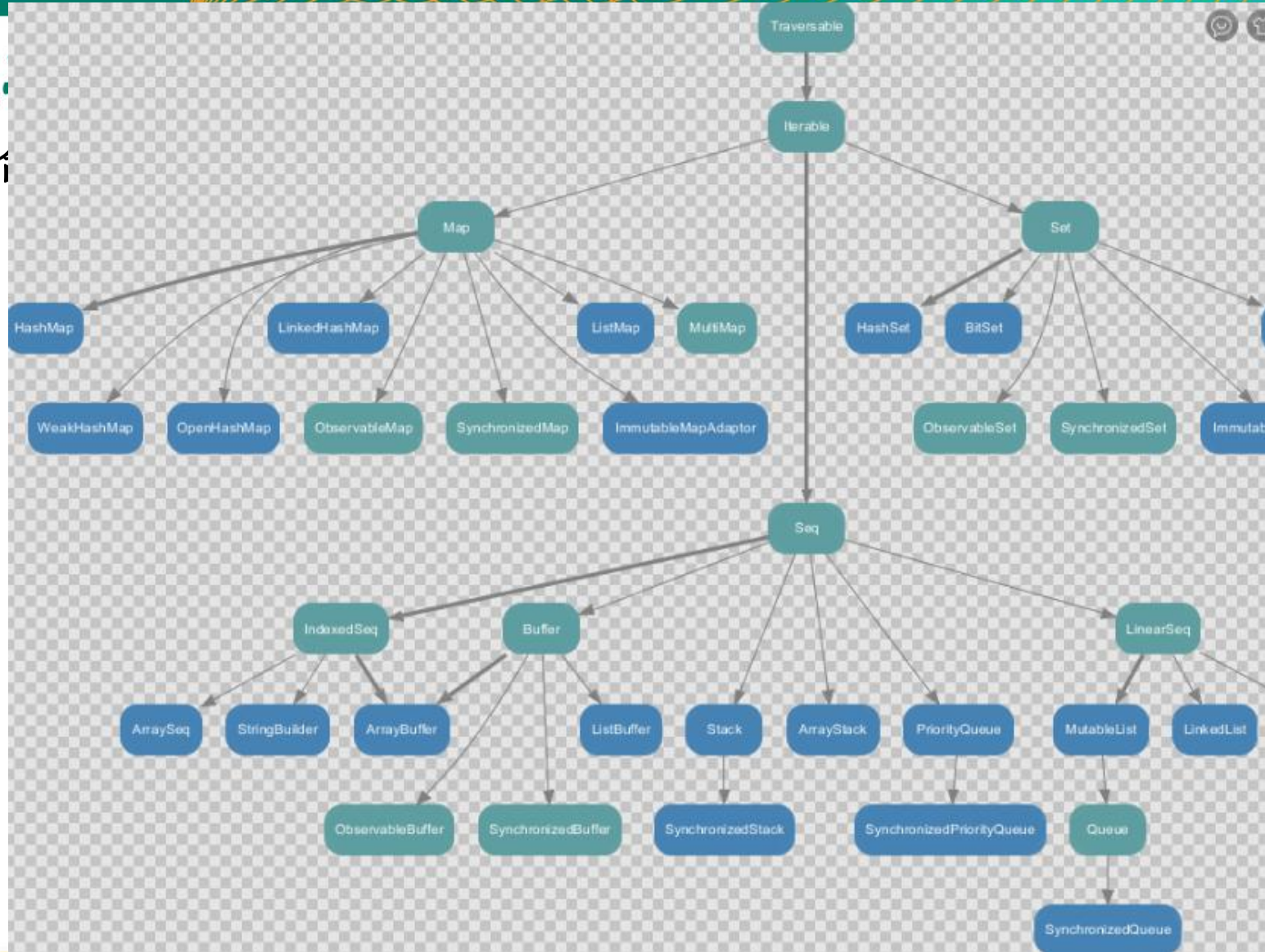
• 可变集合

Scala可变集合继承关系
一览图

小结:



可变集合





- 数组-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,中括号的类型就是数组的类型

```
val arr1 = new Array[Int](10)
```

//赋值,集合元素采用小括号访问

```
arr1(1) = 7
```

案例演示+反编译

```
val arr01 = new Array[Int](4)
println(arr01.length)
```

```
println("arr01(0)=" + arr01(0))
for (i <- arr01) {
  println(i)
}
println("-----")
arr01(3) = 10
for (i <- arr01) {
  println(i)
}
```




- 数组-定长数组(声明泛型)

第二种方式定义数组

在定义数组时，直接赋值
//使用apply方法创建数组对象
val arr1 = Array(1, 2)

```
var arr02 = Array(1, 3, "xxx")  
for (i <- arr02) {  
    println(i)  
}
```




- 数组-变长数组(声明泛型)

基本使用和应用案例

//定义/声明

```
val arr2 = ArrayBuffer[Int]()
```

//追加值/元素

```
arr2.append(7)
```

//重新赋值

```
arr2(0) = 7
```

//学习集合的流程(创建,查询,修改,删除)

案例演示+反编译

```
val arr01 = ArrayBuffer[Any](3, 2, 5)
```

```
println("arr01(1)=" + arr01(1))
```

```
for (i <- arr01) {
```

```
println(i)
```

```
}
```

```
println(arr01.length) //?
```

```
println("arr01.hash=" + arr01.hashCode())
```

```
arr01.append(90.0,13)
```

```
println("arr01.hash=" + arr01.hashCode())
```

```
arr01(1) = 89 //修改
```

```
println("-----")
```

```
for (i <- arr01) {
```

```
println(i)
```

```
}
```

```
//删除
```

```
arr01.remove(0)
```

```
println("-----")
```

```
for (i <- arr01) {
```

```
println(i)
```

```
}
```

```
println("最新的长度=" + arr01.length)
```

- 数组-变长数组(声明泛型)

变长数组分析小结

- 1) ArrayBuffer是变长数组，类似java的ArrayList
- 2) `val arr2 = ArrayBuffer[Int]()` 也是使用的apply方法构建对象
- 3) `def append(elems: A*) { appendAll(elems) }` 接收的是可变参数.
- 4) 每append一次，arr在底层会重新分配空间，进行扩容，arr2的内存地址会发生变化，也就成为新的ArrayBuffer

- 数组-变长数组(声明泛型)

定长数组与变长数组的转换

`arr1.toBuffer` //定长数组转可变数组

`arr2.toArray` //可变数组转定长数组

说明:

- 1) `arr2.toArray` 返回结果才是一个定长数组,
`arr2`本身没有变化
- 2) `arr1.toBuffer`返回结果才是一个可变数组,
`arr1`本身没有变化

```
val arr2 = ArrayBuffer[Int]()
```

```
// 追加值
```

```
arr2.append(1, 2, 3)
```

```
println(arr2)
```

```
val newArr = arr2.toArray;
```

```
println(newArr)
```

```
val newArr2 = newArr.toBuffer
```

```
newArr2.append(123)
```

```
println(newArr2)
```

```
//案例演示+说明
```

● 数组-多维数组

多维数组的定义和使用

➤ 说明

//定义

```
val arr = Array.ofDim[Double](3,4)
```

//说明：二维数组中有三个一维数组，
每个一维数组中有四个元素

//赋值

```
arr(1)(1) = 11.11
```

➤ 案例演示

```
val array1 = Array.ofDim[Int](3, 4)
array1(1)(1) = 9
for (item <- array1) {
  for (item2 <- item) {
    print(item2 + "\t")
  }
  println()
}
println("=====")
for (i <- 0 to array1.length - 1) {
  for (j <- 0 to array1(i).length - 1) {
    printf("arr[%d][%d]=%d\t", i, j, array1(i)(j))
  }
  println()
}
```




- 数组-Scala数组与Java的List的互转

Scala数组转Java的List

在项目开发中，有时我们需要将Scala数组转成Java数组，看下面案例：

```
// Scala集合和Java集合互相转换
val arr = ArrayBuffer("1", "2", "3")
import scala.collection.JavaConversions.bufferAsJavaList
val javaArr = new ProcessBuilder(arr) //为什么可以这样使用？
val arrList = javaArr.command()
println(arrList) //输出 [1, 2, 3]
```

//案例演示+说明

补充：

```
trait MyTrait01 {}
class A extends MyTrait01 {}
object B {
  def test(m: MyTrait01): Unit = {
    println("b ok..")
  }
}
```

//明确一个知识点

//当一个类继承了一个trait

//那么该类的实例，就可以传递给这个

```
val a01 = new A
```

```
B.test(a01)
```



- 数组-Scala数组与Java数组的互转

Java的List转Scala数组(mutable.Buffer)

在项目开发中，有时我们需要将Java的List转成Scala数组，看下面案例：

```
import scala.collection.JavaConversions.asScalaBuffer
import scala.collection.mutable
// java.util.List ==> Buffer
val scalaArr: mutable.Buffer[String] = arrList
scalaArr.append("jack")
println(scalaArr)
//案例演示+说明
```



● 元组Tuple-元组的基本使用

基本介绍

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，称为元组，最多的特点灵活，对数据没有过多的约束。

注意：元组中最大只能有22个元素

元组的创建

```
val tuple1 = (1, 2, 3, "hello", 4)
println(tuple1)
```

案例演示+反编译看类型

➤ 代码小结[后]



● 元组Tuple-元组的基本使用

元组的创建

➤ 代码小结

- 1) t1 的类型是 Tuple5类 是scala特有的类型
- 2) t1 的类型取决于 t1 后面有多少个元素, 有对应关系, 比如 4个元素=》 Tuple4
- 3) 给大家看一个Tuple5 类的定义, 大家就了然了

```
/*  
  final case class Tuple5[+T1, +T2, +T3, +T4, +T5](_1: T1, _2: T2, _3: T3, _4: T4, _5: T5)  
  extends Product5[T1, T2, T3, T4, T5]  
  {  
    override def toString() = "(" + _1 + "," + _2 + "," + _3 + "," + _4 + "," + _5 + ")"  
  }  
*/
```

- 4) 元组中最大只能有22个元素 即 Tuple1...Tuple22



● 元组Tuple-元组数据的访问

基本介绍

访问元组中的数据,可以采用顺序号（_顺序号），也可以通过索引（productElement）访问。

应用案例

```
object Tupleo1 {  
  def main(args: Array[String]): Unit = {  
  
    val t1 = (1, "a", "b", true, 2)  
    println(t1._1) //访问元组的第一个元素，从1开始  
    println(t1.productElement(0)) // 访问元组的第一个元素，从0开始  
  
  }  
}
```

案例演示+说明

- 元组Tuple-元组数据的遍历

Tuple是一个整体，遍历需要调其迭代器。

// Scala可以将多个无关的数据封装为一个整体，称之为元组

```
var t1 = (1, "a", "b", true, 2)
```

// 循环元组, 遍历元组

//tuple1是一个整体，遍历需要调其迭代器

```
for ( item <- t1.productIterator ) {  
    println(item)  
}
```



图片2.z



● 列表 List-创建List

基本介绍

Scala中的List 和Java List 不一样，在Java中List是一个接口，真正存放数据是ArrayList，而**Scala的List可以直接存放数据，就是一个object**，默认情况下Scala的List是不可变的，List属于序列Seq。

```
val List = scala.collection.immutable.List
```

```
object List extends SeqFactory[List]
```

创建List的应用案例

案例演示+分析

```
val list01 = List(1, 2, 3) //创建时，直接分配元素  
println(list01)  
val list02 = Nil //空集合  
println(list02)
```

思考题：为什么没有引入List
， Nil 的包就可以直接使用？

- 列表 List-创建List

创建List的应用案例小结

- 1) List默认为不可变的集合
- 2) List 在 scala包对象声明的,因此不需要引入其它包也可以使用
- 3) `val List = scala.collection.immutable.List`
- 4) List 中可以放任何数据类型, 比如 arr1的类型为 `List[Any]`
- 5) 如果希望得到一个空列表, 可以使用Nil对象, 在 scala包对象声明的,因此不需要引入其它包也可以使用

```
val Nil = scala.collection.immutable.Nil
```




- 列表 **List**-访问**List**元素

应用实例

```
val value1 = list1(1) // 1是索引，表示取出第2个元素.  
println(value1)
```



● 列表 List-元素的追加

基本介绍

向列表中增加元素, **会返回新的列表**/集合对象。**注意**: Scala中List元素的追加形式非常独特, 和Java不一样。

方式1-在列表的最后增加数据

案例演示

```
var list1 = List(1, 2, 3, "abc")  
// :+运算符表示在列表的最后增加数据  
val list2 = list1 :+ 4  
println(list1) //list1没有变化  
println(list2) //新的列表结果是 [1, 2, 3, "abc", 4]
```

方式2-在列表的最前面增加数据

案例演示

```
var list1 = List(1, 2, 3, "abc")  
// :+运算符表示在列表的最后增加数据  
val list2 = 4 ++: list1  
println(list1) //list1没有变化  
println(list2) //新的列表结果是?
```



- 列表 **List**-元素的追加

方式3-在列表的最后增加数据

➤ 说明:

- 1) 符号`::`表示向集合中 新建集合添加元素。
- 2) 运算时，集合对象一定要放置在最右边，
- 3) 运算规则，从右向左。
- 4) `:::` 运算符是将集合中的每一个元素加入到空集合中去

➤ 应用案例:

案例演示+说明

```
val list1 = List(1, 2, 3, "abc")  
val list5 = 4 :: 5 :: 6 :: list1 :: Nil  
println(list5)
```

```
// 下面等价 4 :: 5 :: 6 :: list1  
val list7 = 4 :: 5 :: 6 :: list1 ::: Nil  
println(list7)  
//案例1 + 说明
```



- 列表 **List**-元素的追加

课堂练习题

```
val list1 = List(1, 2, 3, "abc")  
val list5 = 4 :: 5 :: 6 :: list1  
println(list5) // (4,5,6,1,2,3,"abc")  
//题1, 输出的结果是什么?
```

```
val list1 = List(1, 2, 3, "abc")  
val list5 = 4 :: 5 :: 6 :: list1 :: 9  
println(list5) //错误  
//题2, 输出的结果是什么?
```

```
val list1 = List(1, 2, 3, "abc")  
val list5 = 4 :: 5 :: 6 ::: list1 ::: Nil  
println(list5) // 错误 ::: 左右边为集合  
//题3, 输出的结果是什么?
```

```
val list1 = List(1, 2, 3, "abc")  
val list5 = 4 :: 5 :: list1 ::: list1 ::: Nil  
println(list5) // (4,5,1,2,3,"abc",1,2,3,"abc")  
//题4, 输出的结果是什么?
```




- 列表 **ListBuffer**

ListBuffer

ListBuffer是可变的list集合，可以添加，删除元素,ListBuffer属于序列

//追一下继承关系即可

Seq **var** listBuffer = ListBuffer(1,2)

```
val lst0 = ListBuffer[Int](1, 2, 3)
```

```
println("lst0(2)=" + lst0(2))  
for (item <- lst0) {  
  println("item=" + item)  
}
```

```
val lst1 = new ListBuffer[Int]  
lst1 += 4  
lst1.append(5)
```

```
lst0 ++= lst1  
val lst2 = lst0 ++ lst1  
val lst3 = lst0 :+ 5
```

```
println("====删除=====  
println("lst1=" + lst1)  
lst1.remove(1)  
for (item <- lst1) {  
  println("item=" + item)  
}
```

- 队列 Queue-基本介绍

队列的应用场景

银行排队的案例





- 队列 Queue-基本介绍

队列的说明

- 1) 队列是一个有序列表，在底层可以用数组或是链表来实现。
- 2) 其输入和输出要遵循先入先出的原则。即：先存入队列的数据，要先取出。后存入的要后取出
- 3) 在Scala中，由设计者直接给我们提供队列类型使用。
- 4) 在scala中, 有 `scala.collection.mutable.Queue` 和 `scala.collection.immutable.Queue` , 一般来说，我们在开发中通常使用可变集合中的队列。



- 队列 **Queue**-队列的创建

应用案例

```
import scala.collection.mutable
```

```
//说明: 这里的Int是泛型, 表示q1队列只能存放Int类型
```

```
//如果希望q1可以存放其它类型, 则使用 Any 即可。
```

```
val q1 = new mutable.Queue[Int]
```

```
println(q1)
```




- 队列 **Queue**-队列元素的追加数据

向队列中追加单个元素和List

案例演示

```
val q1 = new Queue[Int]
q1 += 20 // 底层?
println(q1)
```

```
q1 ++= List(2,4,6) //
println(q1)
```

```
//q1 += List(1,2,3) //泛型为Any才ok
println(q1)
//案例演示+说明
```

```
//补充操作符重载...
val cat = new Cat
println(cat.age)
cat += 9
println(cat.age)
class Cat {
  var age: Int = 10
  def +=(n: Int): Unit = {
    this.age += n
    println("xxx")
  }
}
```

- 队列 **Queue**-删除和加入队列元素

说明

按照进入队列的顺序删除元素（队列先进先出）

应用案例

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

```
val q1 = new mutable.Queue[Int]//  
q1 += 12  
q1 += 34  
q1 ++= List(2,9)  
q1.dequeue() //队列头  
println(q1)  
q1.enqueue(20,60) //队列位  
println(q1)
```



- 队列 **Queue**-给队列添加元素

说明

按照队列的算法，会将数据添加到队列的最后。

应用案例

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



- 队列 **Queue**-返回队列的元素

返回队列的第一个元素

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

返回队列最后一个元素

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

返回队列的尾部

即：返回除了第一个以外剩余的元素，可以级联使用，这个在递归时使用较多。

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

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




- 映射 **Map**-基本介绍

Java中的Map回顾

HashMap 是一个散列表(数组+链表)，它存储的内容是键值对(key-value)映射，Java中的HashMap是无序的，key不能重复。案例演示：

```
public class TestJavaMap {  
    public static void main(String[] args) {  
        HashMap<String,Integer> hm = new HashMap();  
        hm.put("no1", 100);  
        hm.put("no2", 200);  
        hm.put("no3", 300);  
        hm.put("no4", 400);  
  
        System.out.println(hm);  
        System.out.println(hm.get("no2"));  
    }  
}
```

- 映射 Map-基本介绍

Scala中的Map介绍

- 1) Scala中的Map 和Java类似，也是一个散列表，它存储的内容也是键值对 (key-value)映射，Scala中不可变的Map是有序的，可变的Map是无序的。
- 2) Scala中，有可变Map (`scala.collection.mutable.Map`) 和 不可变Map(`scala.collection.immutable.Map`)



● 映射 Map-构建Map

方式1-构造不可变映射

Scala中的不可变Map是有序，构建Map中的元素底层是Tuple2类型。

➤ 案例

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

➤ 小结

- 1.从输出的结果看到，输出顺序和声明顺序一致
- 2.构建Map集合中，集合中的元素其实是Tuple2类型
- 3.默认情况下（即没有引入其它包的情况下），Map是不可变map
- 4.为什么说Map中的元素是Tuple2 类型 [反编译或看对应的apply]

- 映射 Map-构建Map

方式2-构造可变映射

//需要指定可变Map的包

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

说明

1.从输出的结果看到，输出顺序和声明顺序不一致



- 映射 **Map**-构建Map

方式3-创建空的映射

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

方式4-对偶元组

即创建包含键值对的二元组， 和第一种方式等价， 只是形式上不同而已。

对偶元组 就是只含有两个数据的元组。

```
val map4 = mutable.Map( ("A", 1), ("B", 2), ("C", 3),("D", 30) )  
println("map4=" + map4)  
println(map4("A"))
```



- 映射 Map-取值

方式1-使用map(key)

```
val value1 = map2("Alice")  
println(value1)
```

说明:

- 1) 如果key存在，则返回对应的值
- 2) 如果key不存在，则抛出异常[java.util.NoSuchElementException]
- 3) 在Java中,如果key不存在则返回null



- 映射 **Map**-取值

方式2-使用**contains**方法检查是否存在**key**

// 返回Boolean

// 1.如果key存在，则返回true

// 2.如果key不存在，则返回false

map4.contains("B")

说明:

使用**contains**先判断在取值，可以防止异常，并加入相应的处理逻辑

```
val map4 = mutable.Map( ("A", 1), ("B", 2), ("C", 3),("D", 30.9) )  
if( map4.contains("B") ) {  
    println("key存在 值= " + map4("B"))  
} else {  
    println("key不存在")  
}
```



● 映射 Map-取值

方式3-使用map.get(key).get取值

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

```
var map4 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )
println(map4.get("A")) //Some
println(map4.get("A").get) //得到Some在取出
```

说明和小结:

- 1) map.get方法会将数据进行包装
- 2) 如果 map.get(key) key存在返回some,如果key不存在，则返回None
- 3) 如果 map.get(key).get key存在，返回key对应的值,否则，抛出异常
java.util.NoSuchElementException: None.get



● 映射 Map-取值

方式4-使用map4.getOrElse()取值

getOrElse 方法 : `def getOrElse[V1 >: V](key: K, default: => V1)`

说明:

- 1) 如果key存在, 返回key对应的值。
- 2) 如果key不存在, 返回默认值。在java中底层有很多类似的操作。

```
val map4 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )  
println(map4.getOrElse("A","默认"))
```

如何选择取值方式建议

- 1) 如果我们确定map有这个key ,则应当使用map(key), 速度快
- 2) 如果我们不能确定map是否有key ,而且有不同的业务逻辑, 使用map.contains() 先判断在加入逻辑
- 3) 如果只是简单的希望得到一个值, 使用map4.getOrElse("ip","127.0.0.1")



- 映射 **Map**-对**map**修改、添加和删除

更新map的元素

案例:

```
val map4 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )  
map4("AA") = 20  
println(map4)
```

说明:

- 1) **map** 是可变的, 才能修改, 否则报错
- 2) 如果**key**存在: 则修改对应的值,**key**不存在, 等价于添加一个**key-val**



- 映射 **Map**-对map修改、添加和删除

添加map元素

➤ 方式1-增加单个元素

```
val map4 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )  
map4 += ( "D" -> 4 )  
map4 += ( "B" -> 50 )  
println(map4)
```

思考：如果增加的key 已经存在会怎么样？

➤ 方式2-增加多个元素

```
val map4 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )  
val map5 = map4 + ("E"->1, "F"->3)  
map4 += ("EE"->1, "FF"->3)
```



- 映射 **Map**-对map修改、添加和删除

删除map元素

```
val map4 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )  
map4 -= ("A", "B")  
println("map4=" + map4)
```

说明

- 1) "A","B" 就是要删除的key, 可以写多个.
- 2) 如果key存在, 就删除, 如果key不存在, 也不会报错.



● 映射 Map-对map遍历

对map的元素(元组Tuple2对象)进行遍历的方式很多，具体如下：

```
val map1 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )  
for ((k, v) <- map1) println(k + " is mapped to " + v)  
for (v <- map1.keys) println(v)  
for (v <- map1.values) println(v)  
for(v <- map1) println(v) //v是Tuple?
```

说明

- 1.每遍历一次，返回的元素是Tuple2
- 2.取出的时候，可以按照元组的方式来取



● 集 Set-基本介绍

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

Java中Set的回顾

java中，HashSet是实现Set<E>接口的一个实体类，数据是以哈希表的形式存放的，里面的**不能包含重复数据**。Set接口是一种不包含重复元素的 collection，HashSet中的数据也是**没有顺序的**。

案例演示：

```
HashSet hs = new HashSet<String>();  
hs.add("jack");  
hs.add("tom");  
hs.add("jack");  
hs.add("jack2");  
System.out.println(hs);
```

Scala中Set的说明

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



- 集 **Set**-创建

Set不可变集合的创建

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

Set可变集合的创建

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

```
import scala.collection.mutable
```

```
object ScalaSet01 {  
  def main(args: Array[String]): Unit = {  
    val set01 = Set(1,2,4,"abc")  
    println(set01)  
    val set02 = mutable.Set(1,2,4,"abc")  
    println(set02)  
  }  
}
```



- 集 **Set**-可变集合的元素添加和删除

可变集合的元素添加

```
mutableSet.add(4) //方式1
```

```
mutableSet += 6 //方式2
```

```
mutableSet.+=(5) //方式3
```

说明：如果添加的对象已经存在，则不会重复添加，也不会报错

```
val set02 = mutable.Set(1,2,4,"abc")  
set02.add(90)  
set02 += 78  
set02 += 90  
println(set02)
```



- 集 **Set**-可变集合的元素添加和删除

可变集合的元素删除

```
val set02 = mutable.Set(1,2,4,"abc")
```

```
set02 -= 2 // 操作符形式
```

```
set02.remove("abc") // 方法的形式，scala的Set可以直接删除值
```

```
println(set02)
```

说明：说明：如果删除的对象不存在，则不生效，也不会报错

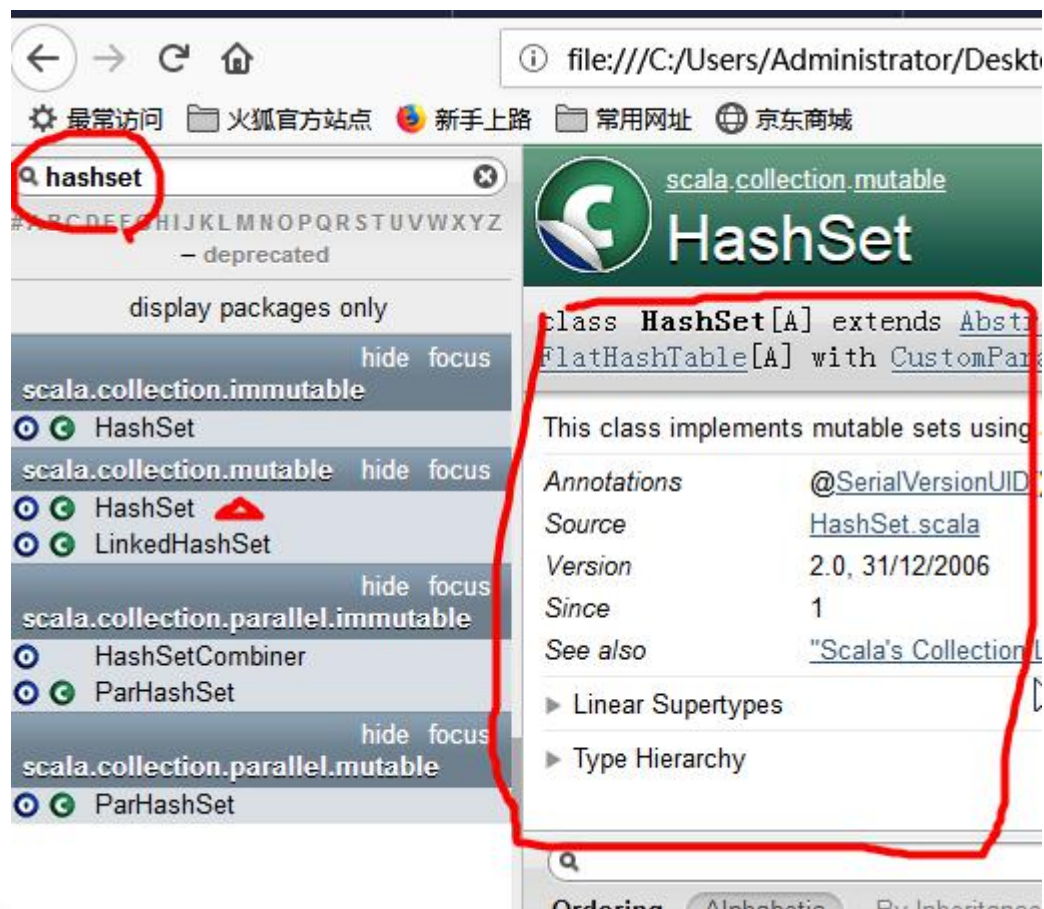


- 集 **Set**-遍历

集Set的遍历

```
val set02 = mutable.Set(1, 2, 4, "abc")
for(x <- set02) {
    println(x)
}
```


- 集 Set-更多操作



file:///C:/Users/Administrator/Desktop

最访问 火狐官方站点 新手上路 常用网址 京东商城

hashset

scala.collection.mutable

HashSet

class **HashSet**[A] extends [AbstractCollection](#)[A] with [FlatHashTable](#)[A] with [CustomParallelizable](#)

This class implements mutable sets using

Annotations	@SerialVersionUID
Source	HashSet.scala
Version	2.0, 31/12/2006
Since	1
See also	"Scala's Collection Library"

► Linear Supertypes

► Type Hierarchy

查看集 Set 的更多使用方法，
可以查看相关的文档。

● 集 Set-更多操作

序号	方法	描述
1	<code>def +(elem: A): Set[A]</code>	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	<code>def -(elem: A): Set[A]</code>	移除集合中的元素，并创建一个新的集合
3	<code>def contains(elem: A): Boolean</code>	如果元素在集合中存在，返回 <code>true</code> ，否则返回 <code>false</code> 。
4	<code>def &(that: Set[A]): Set[A]</code>	返回两个集合的交集
5	<code>def &~(that: Set[A]): Set[A]</code>	返回两个集合的差集
6	<code>def ++(elems: A): Set[A]</code>	合并两个集合
7	<code>def drop(n: Int): Set[A]</code>	返回丢弃前n个元素新集合
8	<code>def dropRight(n: Int): Set[A]</code>	返回丢弃最后n个元素新集合
9	<code>def dropWhile(p: (A) => Boolean): Set[A]</code>	从左向右丢弃元素，直到条件p不成立
10	<code>def max: A //演示下</code>	查找最大元素
11	<code>def min: A //演示下</code>	查找最小元素



谢谢！ 欢迎收看！