Day32 & Day33_Spark中RDD & pairRdd & 共享变量操作

大数据-张军锋 Day32 Day33 Spark 常用类的实例化 RDD pairRdd 共享变量

- Day32 & Day33_Spark中RDD & pairRdd & 共享变量操作
  - Spark中常用类的实例化
    - SparkContext
    - SparkConf
    - 构建RDD
    - 创建累加器
    - 创建广播变量
  - RDD
    - RDD容错
  - RDD API
    - 转换型
      - map
      - mapPartitions
      - flatMap
      - filter
      - keyBy
    - 聚合型
      - reduce
      - fold
      - aggregate
      - distinct
      - sortBy
      - takeOrdered & top
      - groupBy
      - 集合操作
    - 功能型
      - 缓存
      - 暂存rdd中的数据到本地文件中（容错机制）
    - 读值型
  - pairRdd
    - 转换
      - mapValues
      - flatMapValues
      - keys
      - values
    - 聚合
      - reduceByKey
      - foldByKey
      - aggregateByKey

Spark中常用类的实例化

SparkContext

Spark功能使用的主入口点
它负责连接spark集群，并且在spark集群上创建RDD，创建 accumulators ，broadcast variables（共享变量）
实例化方法
`new SparkContext(conf)`多例
`SparkContext.getOrCreate(conf)`单例

SparkConf

实例化方法
`new SparkConf()`

实例化时如果不指定是否加载默认，它会加载默认配置文件，如：core-site.xml，hive-site.xml，hbase-site.xml，如果不想引入这些文件，可以`new SparkConf(false)`

可以链式调用配置其参数：`conf.set("").set("")`

构建RDD

hadoop体系相关的文件中获取数据组装成rdd

```
binaryFiles
binaryRecords
hadoopFile[K, V](
hadoopRDD[K, V]
newAPIHadoopFile
newAPIHadoopRDD
objectFile[T]
sequenceFile[K, V]
textFile
wholeTextFiles
```

从内存中获取数据，组装成rdd

```
makeRDD
parallelize[T]
```

创建累加器

```
register(acc: AccumulatorV2[_, _], name: String)
doubleAccumulator(name: String)
collectionAccumulator[T]
longAccumulator
```

创建广播变量

```
broadcast[T](value: T)(implicit arg0: ClassTag[T])
```

RDD

1. 他有一组分区构成，不同的分区在不同的集群节点上
2. 可以通过调用它的方法（transformation和action），来对rdd的每一个分区进行各自的计算（对rdd进行分布式计算）
3. RDD之间是有依赖关系的，在spark中很少有rdd是独立的
4. 对于kv对的rdd可以使用分区器Partitioner来对其进行分区
5. 在对文件进行加载的时候，它会优先在拥有文件的节点上启动加载executor

RDD容错

rdd是通过依赖关系（DAG）来容错的

RDD API

转换型

map

一个输入对应一个输出，如果没有返回，对应元素的返回值会是()

```
def mapTest(sc:SparkContext) = {
  val file = sc.textFile("file:///D:\\svn\\user-logs-large.txt",3)
  val mapResult = file.map(x => {
    val infos = x.split("\\s")
    (infos(0),infos(1))
  })
  mapResult.take(10).foreach(println)
}
```

输出结果

```
(jim,logout)
(mike,new_tweet)
(bob,new_tweet)
(mike,logout)
(jim,new_tweet)
(marie,view_user)
(jim,login)
(marie,login)
(jim,logout)
(jim,logout)
```

> **take是一个action**，读取取出M条数据发送到Driver，一般应用于开发测试

mapPartitions

一个输入分区对应一个输出分区，新老rdd之间的元素可以不用一对一

```
def mapPartitionTest(sc:SparkContext) = {
  val file = sc.textFile("file:///D:\\svn\\user-logs-large.txt",3)
  val mapPartitionResult = file.mapPartitions(x => {
    var infos = new Array[String](3)
    for(line <- x) yield {
      infos = line.split("\\s")
      (infos(0),infos(1))
    }
  })
  mapPartitionResult.take(10).foreach(println)
}
```

输出结果和map一样

flatMap

一输入对多输出 输入是元素，输出是集合对象

```
//通过转换把一条new_tweet的记录转换成2条login记录
def flatMap(sc:SparkContext) = {
  val file = sc.textFile("file:///D:\\svn\\user-logs-large.txt",3)
  val flatMapResult = file.flatMap(x => {
    val infos = x.split("\\s")
    infos(1) match {
      case "new_tweet" => for (i <- 1 to 2) yield s"${infos(0)}  lo
gin  ${infos(2)}"
      case _ => Array(x)
    }
  })
  flatMapResult.take(10).foreach(println)
}
```

输出结果

```
jim logout  93.24.237.12
mike  login  87.124.79.252
mike  login  87.124.79.252
bob  login  58.133.120.100
bob  login  58.133.120.100
mike    logout  55.237.104.36
jim  login  93.24.237.12
jim  login  93.24.237.12
marie   view_user   122.158.130.90
jim login   198.184.237.49
```

filter

过滤，算子返回true，该条记录被保留，false，该条记录移除

```
def filterTest(sc:SparkContext) = {
  val file = sc.textFile("file:///D:\\svn\\user-logs-large.txt",3)
  val loginFilter = file.filter(x => x.split("\\s")(1)=="login")
  loginFilter.take(10).foreach(println)
}
```

输出结果

```
jim login   198.184.237.49
marie   login   58.133.120.100
bob login   198.184.237.49
jim login   93.24.237.12
bob login   231.112.6.146
marie   login   93.24.237.12
jim login   235.200.255.154
marie   login   122.158.130.90
bob login   198.184.237.49
marie   login   198.184.237.49
```

keyBy

输入一个元素x输出一个kv，v就是原来的元素，k是由x经过算子计算而来，算子把x转成k

```
def keyByTest(sc:SparkContext) = {
  val file = sc.textFile("file:///D:\\svn\\user-logs-large.txt",3)
  val userActionType = file.keyBy(x => {
    val infos = x.split("\\s")
    s"${infos(0)}-${infos(1)}"
  })
  userActionType.take(10).foreach(println)
}
```

输出结果

```
(jim-logout,jim logout  93.24.237.12)
(mike-new_tweet,mike    new_tweet   87.124.79.252)
(bob-new_tweet,bob  new_tweet   58.133.120.100)
(mike-logout,mike   logout  55.237.104.36)
(jim-new_tweet,jim  new_tweet   93.24.237.12)
(marie-view_user,marie  view_user   122.158.130.90)
(jim-login,jim  login   198.184.237.49)
(marie-login,marie  login   58.133.120.100)
(jim-logout,jim logout  58.133.120.100)
(jim-logout,jim logout  93.24.237.12)
```

聚合型

聚合计算函数

reduce

方便使用，只需要定义一个迭代算子，聚合结果的类型要和原rdd的泛型中的数据类型保持一致

fold

比reduce多了一个初值的指定，其他的和reduce一样

aggregate

使用复杂，但功能强大，功能上可以代替reduce和fold，计算效率上三者一致

```
def aggSumTest(sc:SparkContext) = {
  val list = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
  var rdd = sc.parallelize(list,3)
  //reduce计算sum
  val reduceResult = rdd.reduce(_+_)
  //fold计算sum
  val foldResult = rdd.fold(0)(_+_)

  //aggregate把元素连成一个字符串
  val aggregateResult = rdd.aggregate("")((c,v) => {
    c match {
      case "" => v.toString
      case _ => s"$c,$v"
    }
  },(c1,c2) => c1 match {
    case "" => c2.toString
    case _ => s"$c1,$c2"
  })

  println(s"reduceResult:$reduceResult")
  println(s"foldResult:$foldResult")
  println(s"aggregateResult:$aggregateResult")

}
```

输出结果

```
reduceResult:45
foldResult:45
aggregateResult:1,2,3,4,5,6,7,8,9
```

distinct

去重

```
def distinctTest(sc:SparkContext) = {
  val file = sc.textFile("file:///D:\\svn\\user-logs-large.txt",3)
  val userRdd = file.map(x => x.split("\\s")(0)).distinct()
  userRdd.foreach(println)
}
```

输出结果

```
jim
dave
mike
john
alison
mary
joe
dude
bob
marie
```

sortBy

排序 —— 单个分区排序 输入是原rdd的元素，而输出是排序的依据，算子是把x转换成排序条件
如果数据量小，想进行全排序，只需要把numPartitions设置成1即可
如果数据量大，想进行全排序，需要自定义parititoner保证分区间有序，然后再用sortby保证单分区有序

```
def sortByTest(sc:SparkContext) = {
  val file = sc.textFile("file:///D:\\svn\\user_info_spark\\test.tx
t")
  val sortByResult = file.sortBy(x => x.split("\\s+")(1).toInt,numP
artitions = 1)
  //从大到小
  //val sortByResult = file.sortBy(x => x.split("\\s+")(1).toInt,fa
lse,numPartitions = 1)
  sortByResult.foreach(println)
}
```

输出结果

```
gv 34
as  67
q  78
j 123
yuiu 568
asd  789
oii 998
```

takeOrdered & top

takeOrdered ：升序取topN
top：降序取topN

```
def topNTest(sc:SparkContext) = {
  val list = List(5,234,67,9,1,3,12)
  val rdd = sc.parallelize(list,2)
  //从小到大取topN
  //implicit val tonorderd = new Ordering[Int]{
  //  override def compare(x: Int, y: Int): Int = y.compare(x)
  //}
  val takeOrdered = rdd.takeOrdered(3)
  takeOrdered.foreach(println)
  //从大到小取topN
  val topN = rdd.top(3)
  topN.foreach(println)
}
```

输出结果

```
1
3
5

234
67
12
```

分组

groupBy

- 输入是每一个元素，输出是分组条件
- 分组一般是为了计算，分组计算在pairRDD里面提供更高效的方式来进行，而且分组

容易倾斜，所以尽可能避免开使用该方法来对数据进行计算

```
def groupByTest(sc:SparkContext) = {
  val file = sc.textFile("file:///D:\\svn\\user-logs-large.txt")
  val groupByResult = file.groupBy(x => x.split("\\s")(0))
  groupByResult.foreachPartition(x => {
    println(s"groupByRDD分区，该分区数据，${x.size} 条")
  })

  groupByResult.foreach(x => {
    println(s"groupByRDD的一条记录，key为${x._1}，value上的条数
为${x._2.size}条")
  })

  //计算用户登录的次数
  groupByResult.foreach(x => {
    var sum = 0
    x._2.foreach(line => {
      line.split("\\s")(1) match {
        case "login" => sum += 1
        case _ =>
      }
    })
    println(s"用户：${x._1}登录次数${sum}")
  })

}
```

输出结果

```
用户：bob登录次数311
用户：john登录次数0
用户：mary登录次数59
用户：alison登录次数42
用户：jim登录次数872
用户：dave登录次数2
用户：mike登录次数332
用户：dude登录次数1
用户：joe登录次数9
用户：marie登录次数864
```

集合操作

cartesian 两个rdd之间 笛卡尔乘积
union 两个rdd 合集 不去重
subtract 两个rdd 差集
zip 两个rdd相应位置处的数据并成一个元组，形成一个新的rdd
RDD[Int] zip RDD[String] —> RDD[(Int,String)] 拉链操作

功能型

缓存

- 把rdd添加到缓存中，缓存中的rdd在被调用和转换时有利于提升速度
- 如果一个rdd在整个计算过程中会被用到多次，name最好把这个rdd cache到缓存中

cache 它是persist的特例，它等同于persist(StorageLevel.MEMORY_ONLY)
persist 根据级别来设置缓存的存储位置，

```
def persistTest(sc:SparkContext) = {
  val file = sc.textFile("file:///D:\\svn\\user-logs-large.txt")
  //file.cache()
  file.persist(StorageLevel.MEMORY_ONLY)
  //其他操作.......
}
```

暂存rdd中的数据到本地文件中（容错机制）

checkpoint

- 数据冷备份
- 它会把rdd的数据持久化到文件系统（本地文件系统，hdfs）

分区一般在rdd执行了filter，flatmap，聚合等操作之后对新的rdd进行的一种操作，目的是调整下一步计算的平行度，和数据均衡

repartition —— 重分区，使用宽依赖的方式

coalesce —- 重分区，可以选择使用窄依赖的方式

```
def repartitionTest(sc:SparkContext) = {
  val file = sc.textFile("file:///D:\\svn\\user-logs-large.txt")
  val result = file.repartition(5)
  file.foreachPartition(x => {
    println(s"resultRDD分区，该分区数据，${x.size} 条")
  })

  //rePartition分区
  result.foreachPartition(x => {
    var sum = 0
    x.foreach(x => sum+=1)
    println(s"resultRDD分区，该分区数据，$sum 条")
  })

  //coalsce分区
  val coalResult = result.coalesce(3)
  coalResult.foreachPartition(x => {
    println(s"coalResultRDD分区，该分区数据，${x.size} 条")
  })

}
```

输出结果

```
resultRDD分区，该分区数据，5008 条
resultRDD分区，该分区数据，5003 条

rePartitionRDD分区，该分区数据，2001 条
rePartitionRDD分区，该分区数据，2003 条
rePartitionRDD分区，该分区数据，2003 条
rePartitionRDD分区，该分区数据，2003 条
rePartitionRDD分区，该分区数据，2001 条

coalResultRDD分区，该分区数据，4006 条
coalResultRDD分区，该分区数据，2003 条
coalResultRDD分区，该分区数据，4002 条
```

读值型

collect —— 把rdd的数据搜集到driver
count
countByValue
take(n) ——取出前N条数据发送到driver 一般应用于开发测试
foreach ——action，遍历rdd，每个分区

saveAsObjectFile —-action
saveAsTextFile —-action

分区器获取 partitioner
分区对象 partitions
依赖关系dependencies

pairRdd

如果一个rdd的每一个元素都是一个元组，并且这个元组只有两个元素，那么这个rdd会被隐式转换成pairRdd
RDD[(Int,String)]，RDD[(Int,List[String])]，RDD[(Int,(Int,String,Double))]

转换

数据准备

```
val map = List("小张" -> 8000, "小王" -> 5000, "小李" -> 3000)
val rdd = sc.parallelize(map)
```

mapValues

算子的输入：rdd的value
输出：新的value，新的value和原来的key构成新的kv

```
//map 每个人的薪水涨2000
val mapResult = rdd.mapValues(_ + 2000)
mapResult.foreach(println)
```

输出结果

```
(小王,7000)
(小李,5000)
(小张,10000)
```

flatMapValues

算子的输入：rdd的value
输出：集合对象，集合中的每个元素都会和原来的key组装成一个新的rdd的kv

```
//flatMapValues  只对value进行展平
//根据薪水对人打标签，比方说大于等于5000是高收入，否则是低收入
//大于7000是土豪，小于5k是穷屌丝
val flatMapResult = rdd.flatMapValues(x => {
  val tag1 = x match {
    case n if n < 5000 => "低收入"
    case _ => "高收入"
  }
  x match {
    case n if n > 7000 => Array(tag1, "土豪")
    case n if n > 4000 => Array(tag1, "屌丝")
    case _ => Array(tag1, "穷屌丝")
  }
})
flatMapResult.foreach(println)
```

输出结果

```
(小张,高收入)
(小张,土豪)
(小王,高收入)
(小王,屌丝)
(小李,低收入)
(小李,穷屌丝)
```

keys

把kvRdd的key抽出来作为新的rdd

values

把kvRdd的value抽出来作为新的rdd

```
val keysResult = rdd.keys
val valuesResult = rdd.values

keysResult.foreach(println)
valuesResult.foreach(println)
```

输出结果

```
小张
小王
小李

8000
3000
5000
```

聚合

数据准备

```
val scores = List("小张 语文 95", "小张 数学 100", "小张 英语 20", "小李
语文 70", "小李 数学 60", "小李 英语 50")
val rdd = sc.parallelize(scores)
    //计算每个学生的总分数
val reduceRdd = rdd.map(x => {
  val regex = "(.+)\\s(.+)\\s(.+)".r
  x match {
    case regex(studentName, className, score) => (studentName, scor
e.toInt)
  }
})
//val reduceRdd = rdd.map(x => {
//  val infos = x.split("\\s")
//  (infos(0), infos(2).toInt)
//})
```

reduceByKey

应用简单方便，计算效率高，但是需要聚合值和kv对的value的类型保持一致

```
val reduceResult = reduceRdd.reduceByKey(_ + _)
reduceResult.foreach(println)
```

输出结果

```
(小李,180)
(小张,215)
```

foldByKey

和reduceByKey的区别在于，需要指定一个初始值，每个value经过计算的计算过程都一样，其余和reduceByKey一样

```
val foldResult = reduceRdd.foldByKey(0)(_ + _)
foldResult.foreach(println)
```

输出结果

```
(小李,180)
(小张,215)
```

aggregateByKey

比较灵活，功能强大，计算效率高，没有聚合值需要和value类型保持一致的限制

```
val aggregateResult = reduceRdd.aggregateByKey(0)(
  seqOp = (c, v) => c + v
  , combOp = (c1, c2) => c1 + c2
)
aggregateResult.foreach(println)
```

输出结果

```
(小李,180)
(小张,215)
```

combineByKey

它和aggregateByKey基本一样，不同点在于初值的获取方式，aggregateByKey的初值是作为参数我们直接指定
combineByKey的初始值需要我们定义一个函数，由这个函数来生成初始值，它的输入是迭代计算的第一个元素

```
val combineByKeyResult = reduceRdd.combineByKey[Int](
  (initValue: Int) => initValue
  , (c: Int, v: Int) => c + v
  , (c1: Int, c2: Int) => c1 + c2
)
combineByKeyResult.foreach(println)
```

输出结果

```
(小李,180)
(小张,215)
```

要求输出：小李:语文:70,数学:60,英语:50

```
val combbineRDD = rdd.map(x => {
  val infos = x.split("\\s")
  (infos(0), s"${infos(1)} ${infos(2)}")
})

val result = combbineRDD.combineByKey(
  (fv: String) => fv.split("\\s").mkString(":")
  , (c: String, v: String) => s"$c,${v.split("\\s").mkString(":")}"
  , (c1: String, c2: String) => s"$c1,$c2"
)
result.foreach(x => println(s"${x._1}:${x._2}"))
```

输出结果

```
小李:语文:70,数学:60,英语:50
小张:语文:95,数学:100,英语:20
```

groupByKey

对kv的rdd进行分组，按照key把同一个key下的所有value放入一个集合对象中，和key形成一个新的kv

```
val groupByKeyResult = reduceRdd.groupByKey()
groupByKeyResult.foreach(x => {
  println(s"${x._1}总分数：${x._2.sum}")
})
```

输出结果

```
小李总分数：180
小张总分数：215
```

cogroup

对多个rdd中的数据按照key进行分组,key相同的属于多个rdd之间的value都会被打包到一个集合对象中,然后形成一个元组
这个元组会和key形成一个新的kv来作为结果的一条记录

```
def cogroupTest() = {
  val list1 = List("小张" -> "语文 95", "小张" -> "数学 100", "小王" ->
"英语 20", "小李" -> "语文 70", "小李" -> "数学 60")
  val list2 = List("小张" -> "旷课 32", "小张" -> "迟到 21", "小王" ->
"迟到 20", "小李" -> "旷课 10", "小李" -> "迟到 3")
  val rdd1 = sc.parallelize(list1)
  val rdd2 = sc.parallelize(list2)
  val cogroupRdd = rdd1.cogroup(rdd2)
  cogroupRdd.foreach(x => {
    println(s"姓名：${x._1}，成绩信息：${x._2._1}，违纪信
息：${x._2._2}")
  })
  println(s"结果数据记录数：${cogroupRdd.count}")
}
```

输出结果

```
姓名：小李，成绩信息：CompactBuffer(语文 70, 数学 60)，违纪信息：CompactBu
ffer(旷课 10, 迟到 3)
姓名：小王，成绩信息：CompactBuffer(英语 20)，违纪信息：CompactBuffer(迟到
20)
姓名：小张，成绩信息：CompactBuffer(语文 95, 数学 100)，违纪信息：CompactB
uffer(旷课 32, 迟到 21)

结果数据记录数：3
```

集合操作

subtractByKey

根据key值减去rdd中的kv对记录

```
def subtractTest() = {
  val list1 = List("小张" -> "语文 95", "小张" -> "数学 100", "小王" ->
"英语 20", "小李" -> "语文 70", "小李" -> "数学 60")
  val list2 = List("小张" -> 111, "小刘" -> 666)
  val rdd1 = sc.parallelize(list1)
  val rdd2 = sc.parallelize(list2)
  val subtractResult = rdd1.subtractByKey(rdd2)
  subtractResult.foreach(println)
}
```

输出结果
关于小张和小刘的记录全都减去了

```
(小王,英语 20)
(小李,语文 70)
(小李,数学 60)
```

关联操作

数据准备

```
//两个rdd要进行join操作只需保证key的类型一致就可以
val list1 = List("小张" -> "男", "小李" -> "女", "小赵" -> "男")
val list2 = List("小张" -> 23, "小李" -> 21, "小刘" -> 33)
val rdd1 = sc.parallelize(list1)
val rdd2 = sc.parallelize(list2)
```

join

内关联，两个rdd根据key相互排除

```
val innerJoinResult = rdd1.join(rdd2)
innerJoinResult.foreach(x => {
  println(s"姓名：${x._1}，性别：${x._2._1}，年龄：${x._2._2}岁")
})
```

输出结果

```
姓名：小张，性别：男，年龄：23岁
姓名：小李，性别：女，年龄：21岁
```

leftOuterJoin

左边对象为主表，根据key去排除右边对象的元素值，然后关联

```
val leftJoinResult = rdd1.leftOuterJoin(rdd2)
leftJoinResult.foreach(x => {
  println(s"姓名：${x._1}，性别：${x._2._1}，" +
    s"年龄：${x._2._2 match {
      case None => "不详"
      case Some(a) => a
    }
  }")
})
```

输出结果

```
姓名：小张，性别：男，年龄：23
姓名：小赵，性别：男，年龄：不详
姓名：小李，性别：女，年龄：21
```

rightOuterJoin

右3边对象为主表，根据key去排除左边对象的元素值，然后关联

```
val rightJoinResult = rdd1.rightOuterJoin(rdd2)
rightJoinResult.foreach(x => {
  println(s"姓名：${x._1}，" +
    s"性别：${x._2._1 match {
      case None => "不详"
      case Some(a) => a
    }}，" +
    s"年龄：${x._2._2}")
})
```

输出结果

```
姓名：小李，性别：女，年龄：21
姓名：小刘，性别：不详，年龄：33
姓名：小张，性别：男，年龄：23
```

fullOuterJoin

连个rdd相互补充形成新的rdd

```
val fullOutterJoinResult = rdd1.fullOuterJoin(rdd2)
fullOutterJoinResult.foreach(x => {
  println(s"姓名：${x._1}，" +
    s"性别：${x._2._1 match {
      case None => "不详"
      case Some(a) => a
    }}，" +
    s"年龄：${x._2._2 match {
      case None => "不详"
      case Some(a) => a
    }}")
})
```

输出结果

```
姓名：小李，性别：女，年龄：21
姓名：小赵，性别：男，年龄：不详
姓名：小刘，性别：不详，年龄：33
姓名：小张，性别：男，年龄：23
```

action

数据准备

```
val map = List("小张" -> ("语文", 99), "小王" -> ("英语", 77), "小张" -
> ("数学", 100))
val rdd = sc.parallelize(map)
```

lookup

根据key值，获取rdd中所有key等于查询值的结果
如果rdd是根据key来进行分区的话，那么lookup会根据分区器定位查询条件所在的分区，减少扫描数据量，提高查询效率

```
//key是小张的所有记录
val seq = rdd.lookup("小张")
seq.foreach(println)
```

输出结果

```
(语文,99)
(数学,100)
```

collectAsMap

把rdd转换成map
如果rdd中一个key有多个value，那么转换成的map中，一个key只保留其中一个value

```
val cmap = rdd.collectAsMap()
cmap.foreach(println)
```

输出结果

```
(小王,(英语,77))
(小张,(数学,100))
```

countByKey

计算每一个key对应的元素的个数

```
val countByKeyMap = rdd.countByKey()
countByKeyMap.foreach(println)
```

输出结果

```
(小王,1)
(小张,2)
```

保存输出（action）

saveAsHadoopDataset
saveAsHadoopFile——老版本hadoop操作的api
saveAsNewAPIHadoopDataset——把rdd的数据保存到hbase的方法
saveAsNewAPIHadoopFile——把rdd的数据按照指定的格式保存到hbase上
saveAsSequenceFile——把rdd的数据以SequenceFile的格式保存到hbase

```
def saveFile() = {
  val list = List("apple","pear","banana")
  val rdd = sc.parallelize(list).map(x => (x,x.length))
  //保存成sequence文件
  //rdd.saveAsSequenceFile("/spark-sequence-file")
  rdd.saveAsHadoopFile("/spark-sequence-file",classOf[Text],classO
f[IntWritable],classOf[TextOutputFormat[Text,IntWritable]])

}
```

共享变量

在driver上定义，在executor上可以使用的变量，叫共享变量

```
def varTest() = {
  //统计每个用户的访问次数，然后过滤掉小于times的数据，并返回结果
  val times = 50
  val rdd = sc.textFile("file:///D:\\svn\\user-logs-large.txt")
  val result = rdd.map(x => (x.split("\\s")(0), 1))
                  .reduceByKey(_ + _)
                  .filter(x => x._2 > times)
  result.foreach(println)
}
```

输出结果

```
(alison,196)
(bob,1345)
(jim,3476)
(mike,1322)
(marie,3405)
(mary,220)
```

> **注意：**这样的话每次都要读取times的值，效率很低

累加器

accumulator

- 在driver上定义，在executor
- accumulator对executor来说是只写的
- 只有在driver上才能读取出accumulator的正确值

```
def accumulateTest() = {
  //统计每个用户的访问次数,同时计算出大于50和小于50的用户数有多少
  //计算总共有多少条记录
  val rdd = sc.textFile("file:///D:\\svn\\user-logs-large.txt")
  val result = rdd.map(x => {
      recordNumberAccmulator.add(1)
      (x.split("\\s")(0), 1)
    }).reduceByKey(_ + _)
  var sumBt50 = 0
  var sumLt50 = 0
  //使用累加器的方式来完成这种统计
  val sumBtAccmulator = sc.longAccumulator("bt50")
  val sumLtAccmulator = sc.longAccumulator("lt50")

  result.foreach(x => {
    if(x._2 > 50) sumBt50 += 1 else sumLt50 += 1
    if(x._2 > 50) sumBtAccmulator.add(1) else sumLtAccmulator.ad
d(1)
    println(x)
  })
  println(s"大于50的用户数：${sumBt50}")
  println(s"小于50的用户数：${sumLt50}")
  println(s"小于50的用户数：${sumBtAccmulator.value}")
  println(s"小于50的用户数：${sumLtAccmulator.value}")
}
```

输出结果

```
(bob,1345)
(alison,196)
(john,11)
(jim,3476)
(mary,220)
(dave,20)
(mike,1322)
(dude,1)
(joe,15)
(marie,3405)

大于50的用户数：0
小于50的用户数：0
小于50的用户数：6
小于50的用户数：4
总记录数：10011
```

广播变量

broadcast variable

- 在driver上定义赋值，在executor读取使用
- broadcast variable对executor来说是只读的
- 一般在driver上计算或者获取某个值，然后广播给每个executor来使用

```
//调用输入50
def broadcastTest(filterNo:Int) = {
  //统计每个用户的访问次数，然后过滤掉小于filterNo的数据，并返回结果
  val rdd = sc.textFile("file:///D:\\svn\\user-logs-large.txt")
              .map(x => {
                (x.split("\\s")(0),1)
              })
              .reduceByKey(_ + _)
  //把filterNo声明为广播变量
  val broadcastFilterNo = sc.broadcast(filterNo)
  //在executor上使用广播变量，而不直接使用变量值
  val result = rdd.filter(_._2 > broadcastFilterNo.value)
  result.foreach(println)
}
```

输出结果

```
(bob,1345)
(alison,196)
(mary,220)
(jim,3476)
(mike,1322)
(marie,3405)
```