## 数据处理应用场景

批处理 bi分析，olap

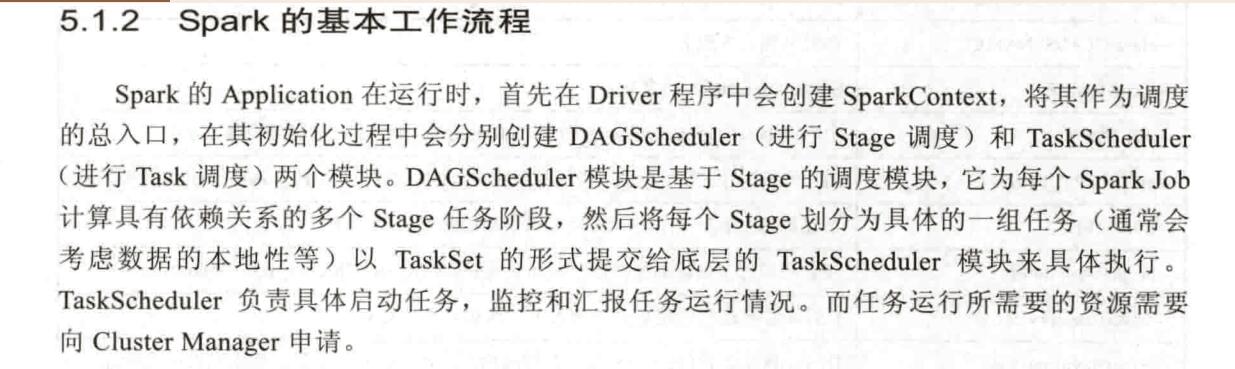
流处理 实时需求

交互式处理 开发调试，数据挖，掘模型计算，机器学期

## 数据处理方式

1. 函数式数据处理，使用某种语言，通过方法的调用完成数据的处理
2. sql

## Spark原理



## Scala连接spark

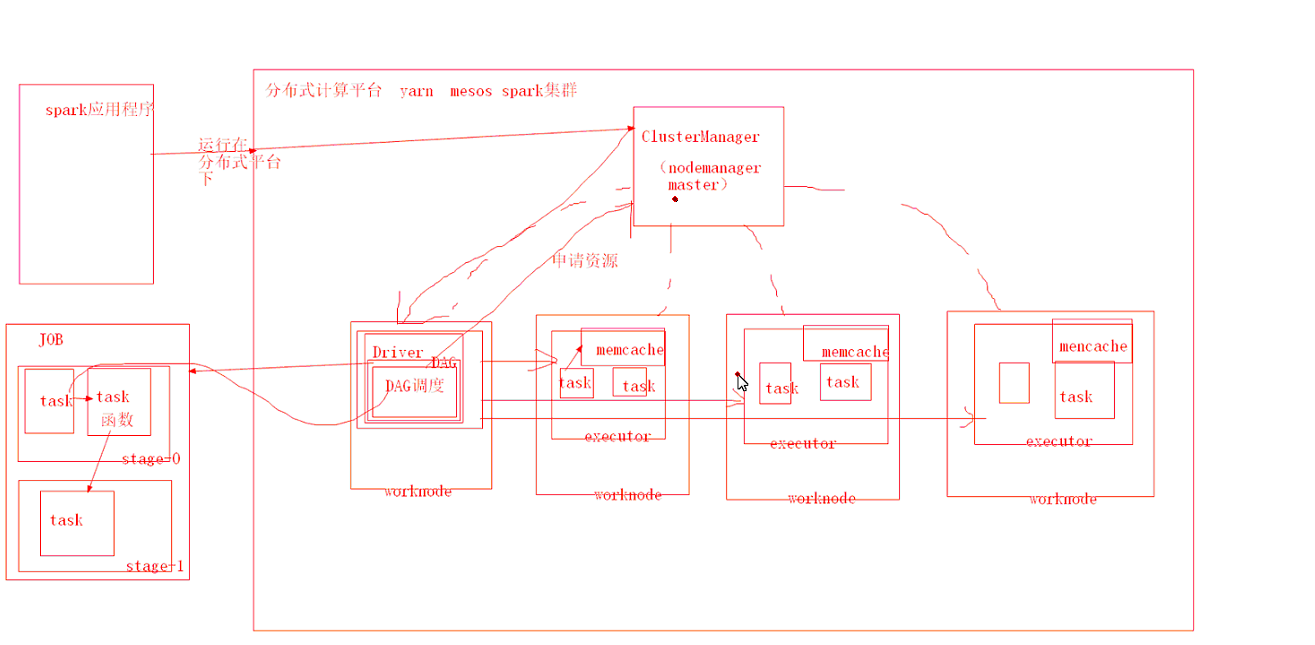
1. 创建scala maven项目
2. 添加spark依赖

|  |
| --- |
| *<!-- https://mvnrepository.com/artifact/org.apache.spark/spark-core -->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-core\_2.11</**artifactId**>  <**version**>2.2.0</**version**> </**dependency**> |

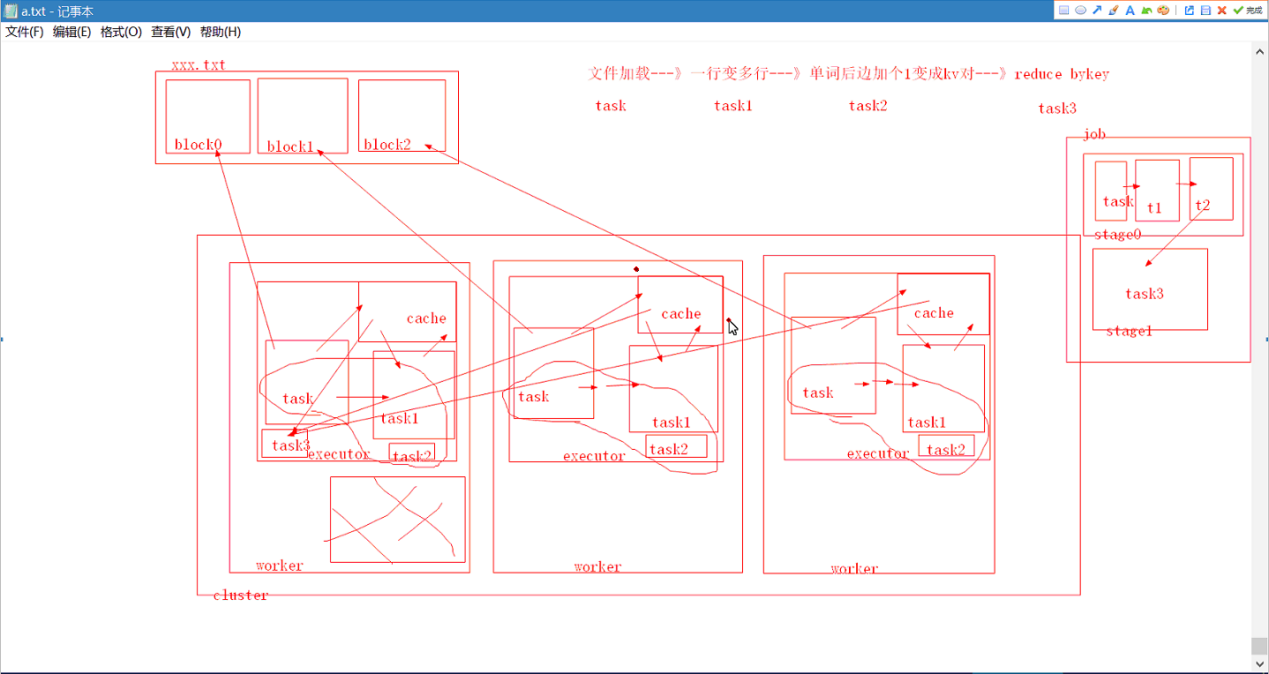
1. 编写WordCount

|  |
| --- |
| **package** top.xiesem.bd14 **import** org.apache.spark.{SparkConf, SparkContext}**object** WordCount {   **def** main(args: Array[String]): Unit = {  *// 创建SparkConf对象  // 设置分布式运行平台，和appname  // 使用master指定运行平台，yarn，standalong,mesos,local  // local:单线程;local[N]:n个线程;local[\*]:本地的cpu有多少个核就启动多少个线程* **val** conf = **new** SparkConf().setMaster(**"local[2]"**).setAppName(**"WordCount App"**)  *// 构建SparkContext对象* **val** sc = **new** SparkContext(conf)  *// 加载数据源，获取RDD对象  // 读取本地文件 file:///c:\\desktop\\read.md  // 读取hdfs上文件 hdfs://master:9000/file* **val** fileRdd = sc.textFile(**"/readme.md"**)  *// 数据处理开始* **val** wordRDD = fileRdd.flatMap(line => line.split(**"\\s"**))  **val** result = wordRDD.map(x => (x, 1)).reduceByKey(\_ + \_)  *// println(s"$result ")* result.saveAsTextFile(**"/spark/WordCountResult"**)   *// 简写形式 // fileRdd.flatMap(\_.split("\\s")).map((\_,1)).reduceByKey(\_ + \_).saveAsTextFile("/spark/WordCountResult")* } } |

Spark程序运行在分布式环境下，先找到Cluster manager，由Cluster manager指定worknode，由Driver进行执行，Driver进行解析生成DAG，同时生成DAG调度，由DAG调度执行job，job将具体的事务叫给task(实质上就是函数)进行执行。处理完这些driver之后，将具体的task分配到集群中交给executor进行具体的执行。在executor中task的结果保存在memcache(内存)中



不同的stage是按照shuffle进行划分的。



Spark程序的数据处理过程，是在driver上进行解析编译的，在具体对数据进行处理的时候需要一个触发点来触发执行计算过程

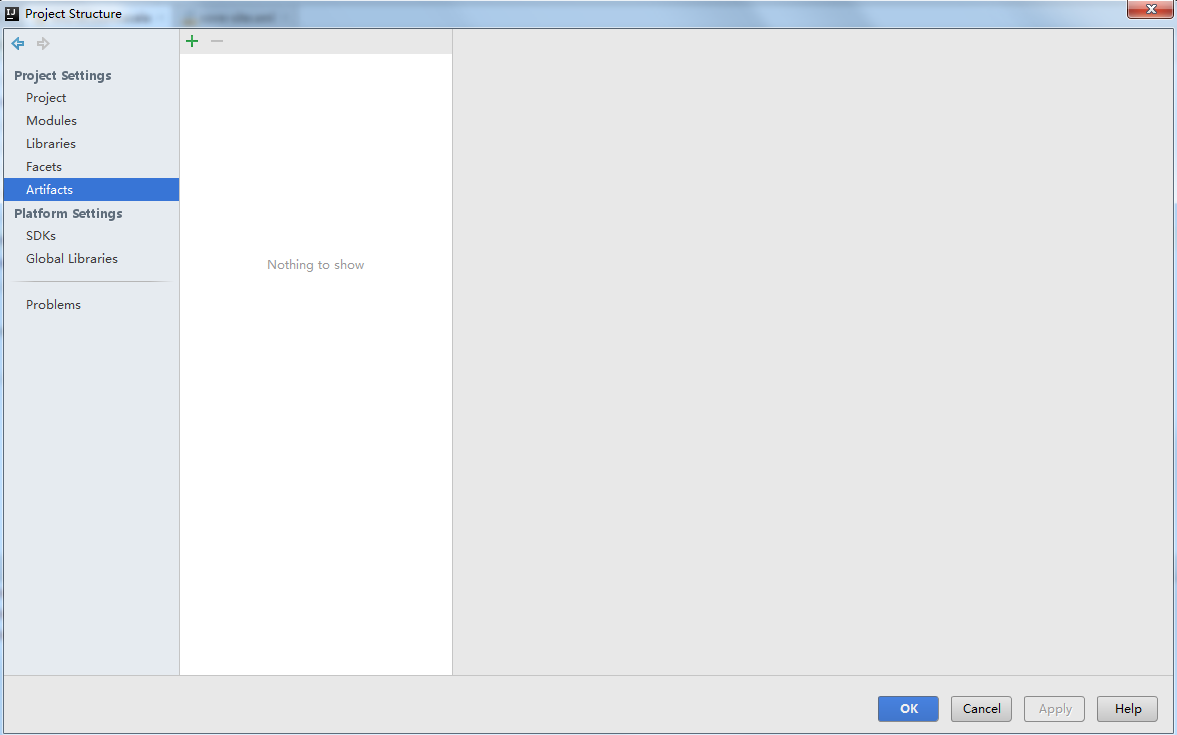
Rdd处理方法(函数)有两种

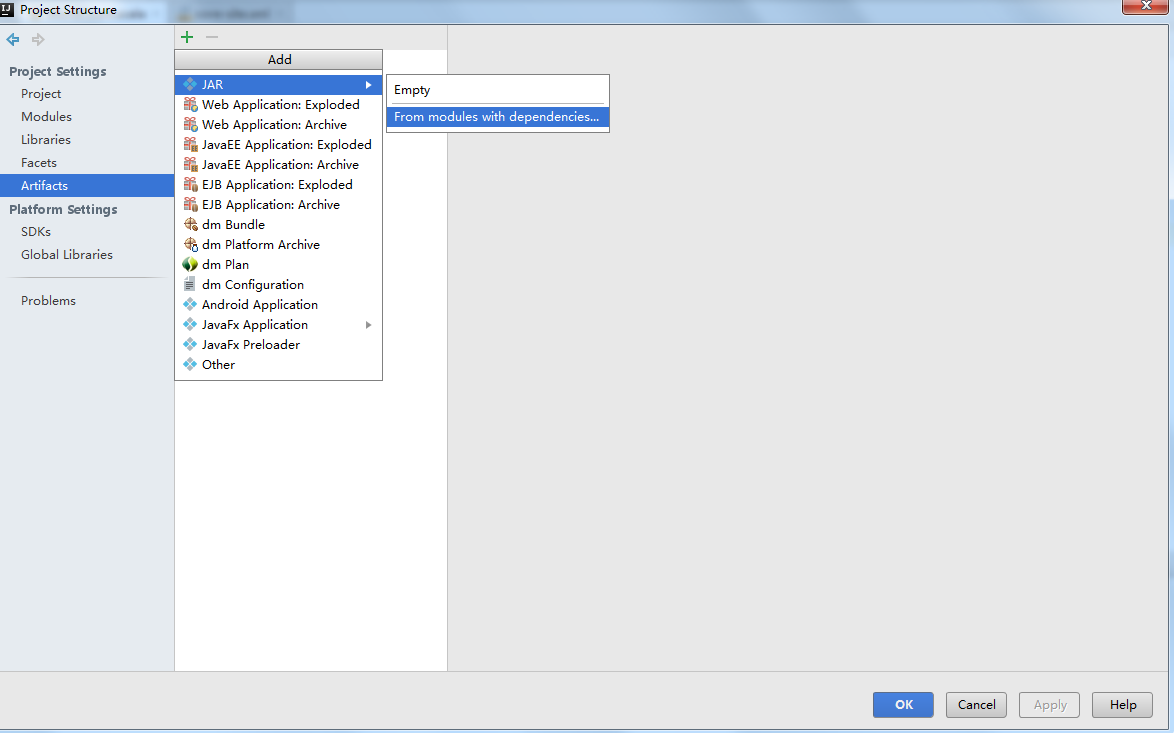
Transformation 转换 对数据进行计算转换，只记录处理过程而不真正计算数据例如，一行变多行，一个单词变成kv，把一个key下的所有value累加

Action 行为 对数据进行读值，触发对数据的计算过程，触发后把action前面的所有的transformation启动执行，一般不改变数据，只是对数据的结果值进行读取

Intellj中的程序直接运行在spark的standalone模式下

1. 创建artifact(项目编辑打包)





## Spark API

### sparkContext

Spark功能使用的主入口点，负责连接spark集群，并且创建RDD，accumulators(累加器) and broadcast(广播变量)，这两个是共享变量

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)

可以链式调用配置参数

Conf.setMaster(“”).setAppName(””)

|  |
| --- |
| **val** conf = **new** SparkConf().setMaster(**"spark://master:7077"**).setAppName(**"WordCount App"**) |

### 构建RDD

从hadoop体系相关的文件中获取数据组装成rdd

binaryFiles()

hadoopFile()

hadoopRDD()

newAPIHadoopFile()

newAPIHadoopRDD()

objectFile()

sequenceFile()

textFile():一行一个记录

wholeTextFiles():一个文件一条记录

从内存中获取数据，组装成rdd

makeRDD()

Parallelize()

创建累加器：register(acc: AccumulatorV2[\_, \_], name: String)

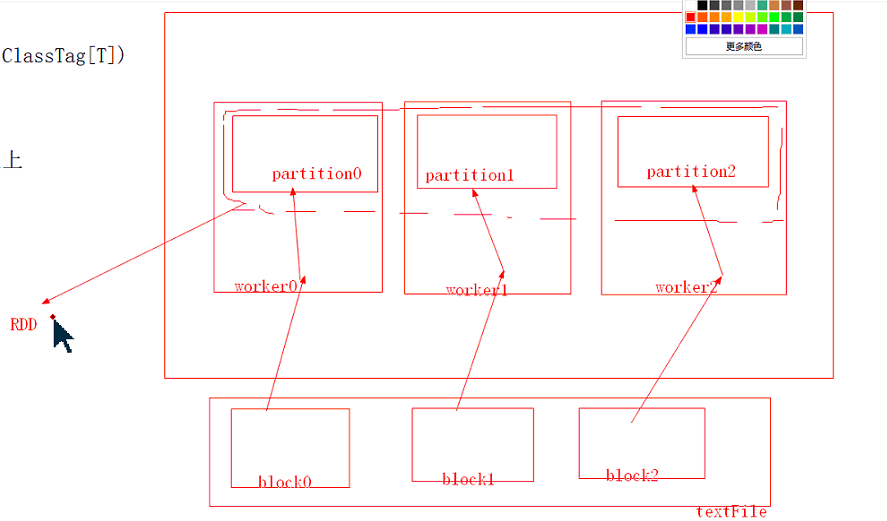
创建广播变量：broadcast[T](value: T)(implicit arg0: ClassTag[T]): Broadcast[T]

默认最小分区：defaultMinPartitions: Int

部署模式：deployMode: String

## RDD

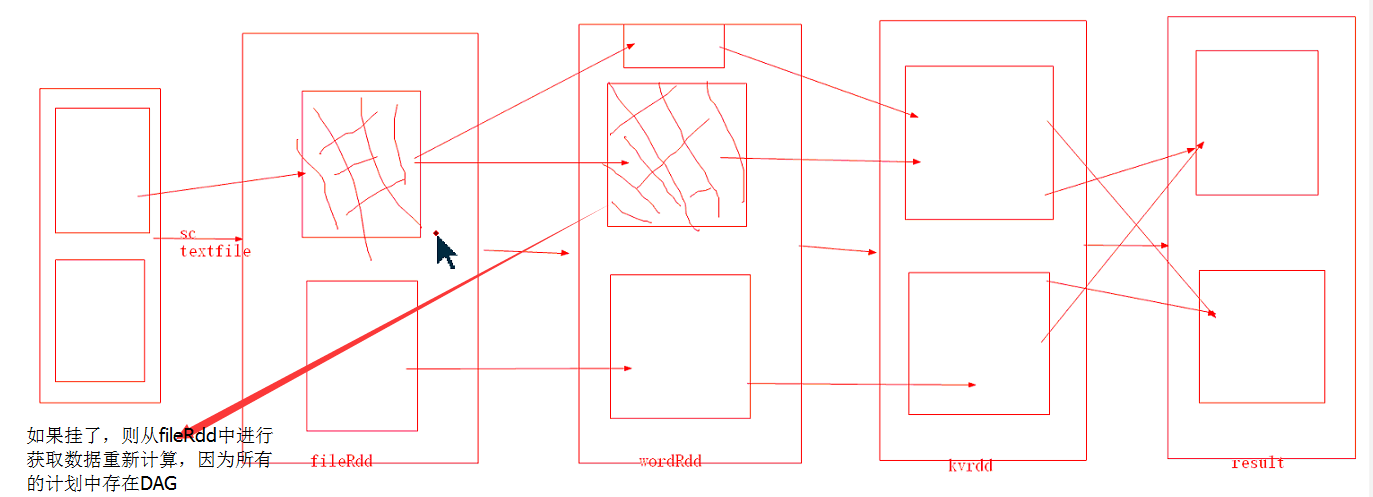
1. 它有一组分区构成，不同的分区在不同的集群节点上



1. 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计算(对rdd进行分布式计算)
2. RDD之间是有依赖关系的，在spark中很少有rdd是独立的
3. 在对文件加载的时候，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

flatMap-- 一输入多输出 输入是元素，输出是集合对象

Map----一个输入对应一个输出，如果没有返回值，对应元素的返回值会是()

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

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转换成k

聚合型

Aggregate

Distinct：去重

sortBy：排序---输入是原rdd，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，只需要将numPartitions设置成1即可

如果数据量大，需要自定义partitions保证分区有序

takeOrdered(num: Int)： 升序求topN

Top： 降序求topN

Group by 输入是每个元素，输出是分组条件

分组一般是为了计算，分组计算在partition里面提供了更为高效的方式，而且分组，容易数据倾斜，能不用就不使用

集合操作型

Cartesian

聚合计算

Reduce 方便使用，只需要定义一个迭代算子，聚合结果的类型和原rdd的数据类型保持一致

Fold 比reduce多一个初始值的指定，其他的和reduce一样

aggregate 操作更加灵活，使用复杂，功能上可以代替reduce和fold，计算效率三者一样

-------------------------------------------------------------------------------------------------------------------------

功能型

----------把rdd添加缓存中，缓存中的rdd被调用和转换时有利于提升速度

如果一个rdd在整个计算中会被用到多次，那么最好把这个rdd添加到cache中

cache() 他是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

cartesian 两个rdd之间 笛卡尔乘积

union 两个rdd合集 不去重

subtract 两个rdd 差积

zip 连个rdd相应位置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

----------占存rdd中的数据到本地文件中(容错)

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

-------分区一般在rdd执行了filter，flatMap，聚合等操作之后对新的rdd的一种操作，目的是调整下一次计算的平行度和数据均衡

Repartition:重分区，使用宽依赖

Coalesce：重分区，使用窄依赖

读值型

collect()

count()

countByValue()

只要返回值不是一个新的RDD的就不是transformation

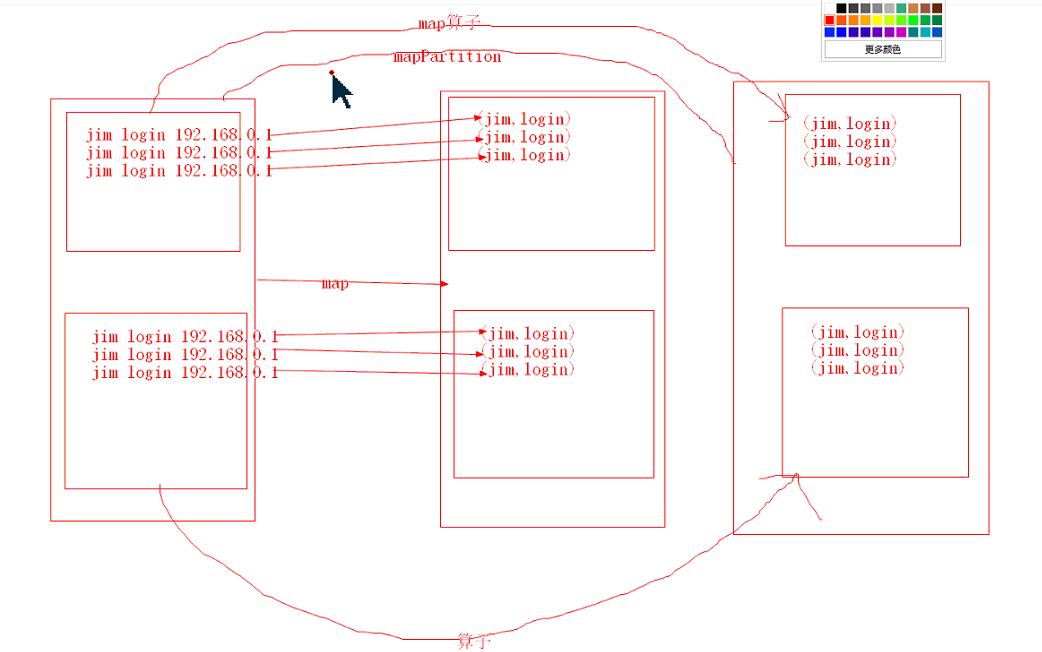
数据模型资源手册

数据模型工具：ea、powerdesigener、uml

#### Map抽取数据

Map----一个输入对应一个输出，如果没有返回值，对应元素的返回值会是()

|  |
| --- |
| *// jim logout 93.24.237.12 ==> jim logout*  *// take(n) 取出前n条数据发送到driver，一般应用于开发测试* **def** mapTest(sc: SparkContext) = {  **val** file = sc.textFile(**"/bd14/user-logs-large.txt"**, 3)  **val** mapresult = file.map(x => {  **val** info = x.split(**"\\t"**)  (info(0),info(1))  })  mapresult.take(10).foreach(*println*) } |



#### mapPartitions

|  |
| --- |
| *// mapPartitions* **def** mapPartitionsTest(sc: SparkContext) = {  **val** file = sc.textFile(**"/bd14/user-logs-large.txt"**, 3)  **val** mapPartitionResult = file.mapPartitions(x => {  **var** info = **new** Array[String](3)  */\*val mapR = \*/***for**(line <- x) **yield** {  info = line.split(**"\\t"**)  (info(0),info(1))  } *// mapR.filter(\_.\_2 == "login")* })  mapPartitionResult.take(10).foreach(*println*)  } |

#### flatMap

|  |
| --- |
| *// flatMap  // 通过转换把一条new\_tweet的记录转换成两条login* **def** flatMapTest(sc:SparkContext) = {  **val** file = sc.textFile(**"/bd14/user-logs-large.txt"**, 3)  **val** flatMapResult = file.flatMap(x => {  **val** info = x.split(**"\\t"**)  info(1) **match** {  **case "new\_tweet"** => **for**(i <- 1 to 2) **yield s"$**{info(0)} **login $**{info(2)}**"  case** \_ => *Array*(x)  }  }) *// flatMapResult.take(10).foreach(println)  println*(file.count())  *println*(flatMapResult.count())  } |

#### Distinct

|  |
| --- |
| *// distinct* **def** distinctTest(sc: SparkContext) = {  **val** file = sc.textFile(**"/bd14/user-logs-large.txt"**, 3)  **val** distinctResult = file.map(x => x.split(**"\\t"**)(0)).distinct()  distinctResult.foreach(*println*) } |

#### Filter

|  |
| --- |
| *// filter* **def** filterTest(sc: SparkContext) = {  **val** file = sc.textFile(**"/bd14/user-logs-large.txt"**, 3)  **val** filterResult = file.filter(x => x.split(**"\\t"**)(1) == **"login"**)  filterResult.take(10).foreach(*println*)  *println*(filterResult.count()) } |

#### 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

|  |
| --- |
| *// 将logout 93.24.237.12 ==》 (jim--logout,jim logout 93.24.237.12) // k上是我们定义的值，v是全部记录* **def** keyByTest(sc: SparkContext) = {  **val** file = sc.textFile(**"/bd14/user-logs-large.txt"**, 3)  **val** keyByResult = file.keyBy(x => {  **val** info = x.split(**"\\t"**)  **s"$**{info(0)}**--$**{info(1)}**"** })  keyByResult.take(10).foreach(*println*) } |

#### sortBy

|  |
| --- |
| *// 分区排序，保证每个分区有序,设置ascending: Boolean = false降序,默认升序* **def** sortByTest(sc: SparkContext) = {  **val** file = sc.textFile(**"/spark/WordCountResult/part-00001"**)  **val** sortByResult = file.sortBy(x => x.split(**","**)(1).substring(0, 1).toInt, **false**, numPartitions = 1)  sortByResult.foreach(*println*) } |

#### topN

|  |
| --- |
| *// topN问题* **def** topNTest(sc: SparkContext) = {  **val** list = *List*(1, 2, 3, 4, 12, 6, 56, 2)  **val** rdd = sc.parallelize(list, 2)  *// 从小到大* **val** topNResult = rdd.takeOrdered(3)  topNResult.foreach(*println*)  *println*(**"----------------------"**)  *// 从大到小* **val** topN = rdd.top(3)  topN.foreach(*println*) } |

宽依赖：父rdd一个分区里面的数据流向子rdd多个分区，这种关系叫宽依赖

窄依赖：父rdd的一个分区里的数据只流向子rdd的一个分区，这种关系叫窄依赖

### 重新分区

|  |
| --- |
| *// 重新分区* **def** repatitionTest(sc: SparkContext) = {  **val** file = sc.textFile(**"/bd14/user-logs-large.txt"**)  **val** result = file.repartition(5)  file.foreachPartition(x => {  *println*(**s"fileRdd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"**)  })  **val** coalResult = result.coalesce(3)  coalResult.foreachPartition(x => {  *println*(**s"cocalResultRdd分区，该分区数据: $**{x.size} **条"**)  })  } |

#### Group by

|  |
| --- |
| *// 分组* **def** groupByTest(sc: SparkContext) = {  **val** file = sc.textFile(**"/bd14/user-logs-large.txt"**)  **val** groupedBy = file.groupBy(x => x.split(**"\\t"**)(0))  groupedBy.foreachPartition(x => {  *println*(**s"groupbyRdd分区，该分区共有: $**{x.size} **条数据"**)  })   groupedBy.foreach(x => {  *println*(**s"groupbyRdd的一条记录，key为 $**{x.\_1}**,value上集合的记录条数是: $**{x.\_2.size} **条"**)  })   *// 计算登录次数* groupedBy.foreach(x => {  **var** sum = 0  x.\_2.foreach(line => {  line.split(**"\\t"**)(1) **match** {  **case "login"** => sum += 1  **case** \_ =>  }  })  *println*(**s"用户$**{x.\_1}**,登录次数$**{sum}**"**)  }) } |

#### Reduce、fold、aggregate

|  |
| --- |
| **def** aggSumTest(sc: SparkContext) = {  **val** list = *List*(1,2,3,4,5,23,31,12,121)  **val** rdd = sc.parallelize(list,3)  *// reduce计算sum* **val** reduceResult = rdd.reduce(\_ + \_)  *// fold计算* **val** foldResult = rdd.fold(0)(\_ + \_)  *// aggregate 把元素拼接字符串* **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}**"**) } |

foreach ---- action 变量rdd，这个是每个分区各自执行

Collect --- 把rdd的数据搜集到driver

foreachPartition --- action，遍历rdd，这个是每个分区各自执行一个算子

saveAsObjectFile ------ action

saveAsTextFile --- action

分区器获取 Partitions

依赖关系 dependent

## pairRdd

如果一个rdd的每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会隐式转换成pairRdd。例如：RDD[(Int,String)]、RDD[(Int,List[String])],RDD[(Int,(Int,String,Double))]

### 转换

flatMapValues：算子输入，rdd的value 输出：集合对象，集合对象中每个元素会和原来的key组装成一个新的rdd的kv

|  |
| --- |
| *// flatMapValue // 根据薪水对人打标签，大于等于5000，高收入;小于5000，低收入;大于7000，土豪;* **val** flatMapResult = rdd.flatMapValues(x => {  **val** tag1 = x **match** {  **case** n **if** n < 5000 => **"低收入"  case** \_ => **"高收入"** }  **val** result = x **match** {  **case** n **if** n > 7000 => *Array*(tag1, **"土豪"**)  **case** \_ => *Array*(tag1)  }  result }) flatMapResult.foreach(*println*) |

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

|  |
| --- |
| **val** map = *List*(**"小张"** -> 6000, **"小王"** -> 13500, **"小李"** -> 4000) **val** rdd = *sc*.parallelize(map) *// mapValue 每个人的薪水加2000 // val mapResult = rdd.map(v => v.\_2 + 2000)* **val** mapResult = rdd.mapValues(\_ + 2000) mapResult.foreach(*println*) |

Keys ： 把kvrdd的key抽取出来作为新的rdd

|  |
| --- |
| **val** map = List(**"小张"** -> 6000, **"小王"** -> 13500, **"小李"** -> 4000) **val** rdd = *sc*.parallelize(map)  **val** keysResult = rdd.keys  keysResult.foreach(*println*) |

Value : 把kvrdd的value抽取出来最为新的rdd

|  |
| --- |
| **val** map = List(**"小张"** -> 6000, **"小王"** -> 13500, **"小李"** -> 4000) **val** rdd = *sc*.parallelize(map)  **val** valuesResult = rdd.values  valuesResult.foreach(*println*) |

### 聚合

#### reduceByKey

应用简单方便，计算效率高，但是需要聚和值和kv的value的类型保持一致

|  |
| --- |
| *// reduceByKey* **val** scores = *List*(**"小张 语文 20"**, **"小张 数学 80"**, **"小李 语文 70"**, **"小李 英语 70"**, **"小王 语文 10"**) **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, score.toInt)  } }) **val** reduceResult = reduceRdd.reduceByKey(\_ + \_) reduceResult.foreach(*println*) |

#### foldByKey

和reduceByKey的区别在于，需要指定一个初始值，每个value的计算过程和reduceByKey一致

|  |
| --- |
| **val** scores = *List*(**"小张 语文 20"**, **"小张 数学 80"**, **"小李 语文 70"**, **"小李 英语 70"**, **"小王 语文 10"**) **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, score.toInt)  } })  *// foldBykey* **val** foldResult = reduceRdd.foldByKey(0)(\_ + \_) foldResult.foreach(*println*) |

#### aggregateByKey

比较灵活,功能强大，但是需要聚合值和kv对的value类型保持一致

|  |
| --- |
| **val** scores = *List*(**"小张 语文 20"**, **"小张 数学 80"**, **"小李 语文 70"**, **"小李 英语 70"**, **"小王 语文 10"**) **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, score.toInt)  } })  **val** aggregateResult = reduceRdd.aggregateByKey(0)(  seqOp = (c, v) => c + v  , combOp = (c1, c2) => c1 + c2 ) aggregateResult.foreach(*println*) |

#### groupbyKey

对kv的rdd进行分组，按照key把同一个key下所有value放入到一个集合对象中，和k形成一个新的kv

|  |
| --- |
| **val** scores = *List*(**"小张 语文 20"**, **"小张 数学 80"**, **"小李 语文 70"**, **"小李 英语 70"**, **"小王 语文 10"**) **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, score.toInt)  } })  *// groupByKey* **val** groupByKey = reduceRdd.groupByKey() groupByKey.foreach(x => {  *println*(**s"$**{x.\_1} **总分数: $**{x.\_2.sum}**"**) }) |

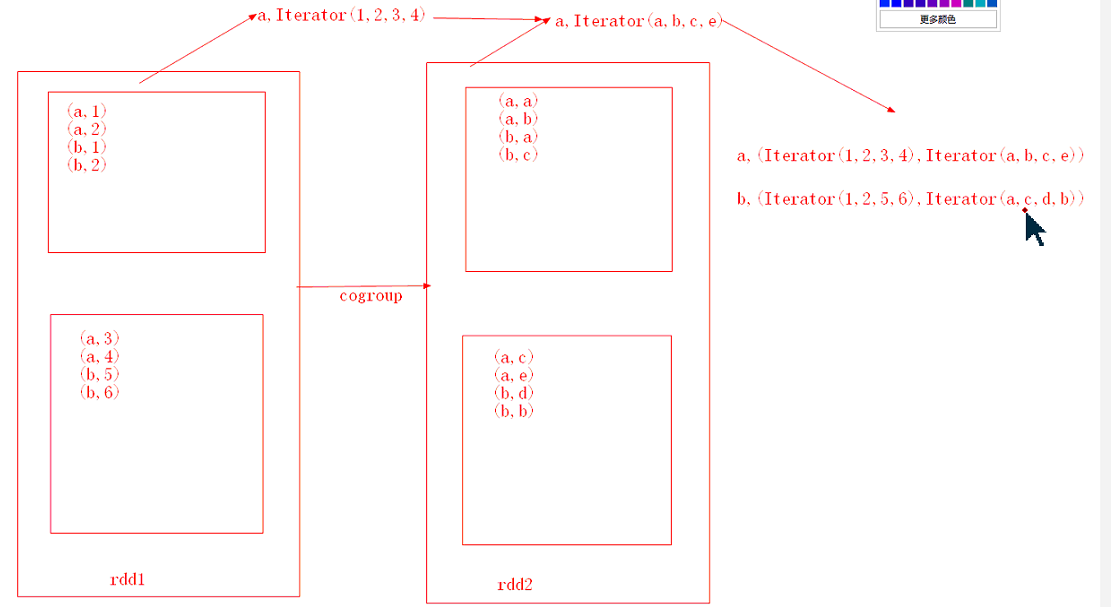
#### combineBykey

它和aggregateByKey基本一样，不同点在于初始值的获取方式,aggregateByKey的初始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，由函数生成初始值，它的输入是迭代的元素

|  |
| --- |
| **val** scores = *List*(**"小张 语文 20"**, **"小张 数学 80"**, **"小李 语文 70"**, **"小李 英语 70"**, **"小王 语文 10"**) **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, score.toInt)  } })  *// combineBykey* **val** combineResult = reduceRdd.combineByKey(  (initValue: Int) => initValue  , (c: Int, v: Int) => c + v  , (c1: Int, c2: Int) => c1 + c2 ) combineResult.foreach(*println*) |

#### 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都会打包成一个集合对象中，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



|  |
| --- |
| *// coGroup* **def** coGroupTest() = {  **val** list1 = *List*(**"小张"** -> **"语文 50"**, **"小张"** -> **"数学 60"**, **"小王"** -> **"语文 70"**, **"小王"** -> **"英语 50"**, **"小李"** -> **"数学 80"**)  **val** list2 = *List*(**"小张"** -> **"迟到 50"**, **"小张"** -> **"旷课 20"**, **"小王"** -> **"迟到 2"**, **"小王"** -> **"旷课 0"**)  **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()}**"**) } |

### 输出保存(action)

#### Lookup

根据key值，获取rdd中所有key等于查询的结果。如果rdd是根据key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，提升查询效率

|  |
| --- |
| **val** map = *List*(**"小张"** -> (**"语文"**, 88), **"小王"** -> (**"英语"**, 92), **"小张"** -> (**"数学"**, 20)) **val** rdd = *sc*.parallelize(map) *// 找出小张的所有记录* **val** lookUpResult = rdd.lookup(**"小张"**) *// lookUpResult.foreach(println)* |

#### 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** map = List(**"小张"** -> (**"语文"**, 88), **"小王"** -> (**"英语"**, 92), **"小张"** -> (**"数学"**, 20)) **val** rdd = *sc*.parallelize(map)  *// collectAsMap* **val** collectAsMapResult = rdd.collectAsMap() collectAsMapResult.foreach(*println*) |

#### countByKey

计算每一个key对应的元素的个数

|  |
| --- |
| **val** map = List(**"小张"** -> (**"语文"**, 88), **"小王"** -> (**"英语"**, 92), **"小张"** -> (**"数学"**, 20)) **val** rdd = *sc*.parallelize(map)  *// countByKey* **val** countByKeyResult = rdd.countByKey() countByKeyResult.foreach(*println*) |

#### saveAsNewAPIHadoopDataset

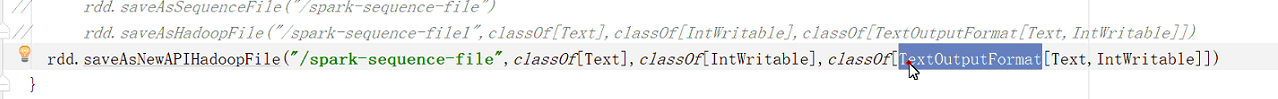
把rdd的数据保存到hbase的方法

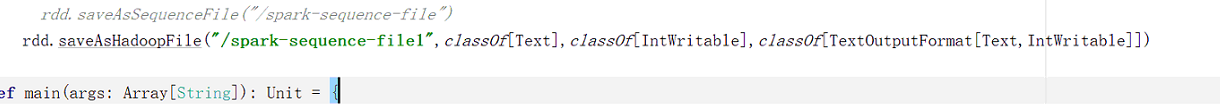
#### saveAsNewAPIHadoopFile

把rdd的数据按照指定的格式保存到hdfs上

#### saveAsSequenceFile

把rdd的数据以sequence格式存储到hdfs上





### 集合操作

#### subtractByKey

(差集)计算每一个key对应的元素的个数

|  |
| --- |
| **def** subtractTest() = {  **val** list1 = *List*(**"小张"** -> **"语文 50"**, **"小张"** -> **"数学 60"**, **"小王"** -> **"语文 70"**, **"小王"** -> **"英语 50"**, **"小李"** -> **"数学 80"**)  **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*) } |

#### 关联操作

1. Join 内关联 两个rdd根据key相互排除

|  |
| --- |
| *// 两个rdd进行关联，只需要key的类型相同即可，value不做要求* **val** list1 = *List*(**"小张"** -> **"男"**, **"小李"** -> **"女"**, **"小王"** -> **"男"**, **"小刘"** -> **"女"**) **val** list2 = *List*(**"小张"** -> 23, **"小王"** -> 18, **"小刘"** -> 19, **"小赵"** -> 22) **val** rdd1 = *sc*.parallelize(list1) **val** rdd2 = *sc*.parallelize(list2)  *// join 内关联* **val** innerJoinResult = rdd1.join(rdd2) innerJoinResult.foreach(x => {  *println*(**s"姓名:$**{x.\_1}**,性别:$**{x.\_2.\_1}**,年龄:$**{x.\_2.\_2}**"**) }) |

2. leftOuterJoin 左边对象为主表，根据key排除右边对象的元素值，然后关联

|  |
| --- |
| *// 两个rdd进行关联，只需要key的类型相同即可，value不做要求* **val** list1 = *List*(**"小张"** -> **"男"**, **"小李"** -> **"女"**, **"小王"** -> **"男"**, **"小刘"** -> **"女"**) **val** list2 = *List*(**"小张"** -> 23, **"小王"** -> 18, **"小刘"** -> 19, **"小赵"** -> 22) **val** rdd1 = *sc*.parallelize(list1) **val** rdd2 = *sc*.parallelize(list2)  *// leftOuterJoin* **val** leftOuterJoinResult = rdd1.leftOuterJoin(rdd2) leftOuterJoinResult.foreach(x => {  **val** age = x.\_2.\_2 **match** {  **case** None => **"不详"  case** *Some*(a) => a  }  *println*(**s"姓名:$**{x.\_1}**,性别:$**{x.\_2.\_1}**,年龄:$**{age}**"**) }) |

3. rightOuterJoin 右边对象为主表，根据key排除左边对象的元素值，然后关联

|  |
| --- |
| *// 两个rdd进行关联，只需要key的类型相同即可，value不做要求* **val** list1 = *List*(**"小张"** -> **"男"**, **"小李"** -> **"女"**, **"小王"** -> **"男"**, **"小刘"** -> **"女"**) **val** list2 = *List*(**"小张"** -> 23, **"小王"** -> 18, **"小刘"** -> 19, **"小赵"** -> 22) **val** rdd1 = *sc*.parallelize(list1) **val** rdd2 = *sc*.parallelize(list2)  *// rightOuterJoin* **val** rightOuterJoinResult = rdd1.rightOuterJoin(rdd2) rightOuterJoinResult.foreach(x => {  **val** gender = x.\_2.\_1 **match** {  **case** None => **"不详"  case** *Some*(a) => a  }  *println*(**s"姓名:$**{x.\_1}**,性别:$**{gender}**,年龄:$**{x.\_2.\_2}**"**) }) |

4. fullOuterJoin 两个rdd相互补充形成新的rdd

|  |
| --- |
| *// 两个rdd进行关联，只需要key的类型相同即可，value不做要求* **val** list1 = *List*(**"小张"** -> **"男"**, **"小李"** -> **"女"**, **"小王"** -> **"男"**, **"小刘"** -> **"女"**) **val** list2 = *List*(**"小张"** -> 23, **"小王"** -> 18, **"小刘"** -> 19, **"小赵"** -> 22) **val** rdd1 = *sc*.parallelize(list1) **val** rdd2 = *sc*.parallelize(list2)  *// fullOuterJoin 全外连接* **val** fullOuterJoinResult = rdd1.fullOuterJoin(rdd2) fullOuterJoinResult.foreach(x => {  **val** gender = x.\_2.\_1 **match** {  **case** None => **"不详"  case** *Some*(a) => a  }  **val** age = x.\_2.\_2 **match** {  **case** None => **"不详"  case** *Some*(a) => a  }  *println*(**s"姓名:$**{x.\_1}**,性别:$**{gender}**,年龄:$**{age}**"**) }) |

## 共享变量

### 累加器(accumulator)

register(acc: AccumulatorV2[\_, \_], name: String): Unit

longAccumulator(name: String): LongAccumulator

doubleAccumulator(name: String): DoubleAccumulator

collectionAccumulator[T](name: String): CollectionAccumulator[T]

1. 在driver上定义，在executor上写

2. Accumulate对executor来说只是写的

3. 只有在driver上才能读取出accumulate的正确值

|  |
| --- |
| **val** *conf* = **new** SparkConf().setMaster(**"local[\*]"**).setAppName(**"ShareVariable"**) **val** *sc* = SparkContext.*getOrCreate*(*conf*)  **def** accumulateTest() = {  *// 统计每个用户的访问次数，同时计算大于50和小于50的用户数有多少* **val** rdd = *sc*.textFile(**"/bd14/user-logs-large.txt"**)  **val** recordNumberAccumulator = *sc*.longAccumulator(**"recordNumber"**)   **val** result = rdd.map(x => {  recordNumberAccumulator.add(1)  (x.split(**"\\t"**)(0), 1)  }).reduceByKey(\_ + \_)   **var** sumBt50 = 0  **var** sumLt50 = 0  *// /bd14/user-logs-large.txt文件中总共有多少记录   // 使用累加器来完成统计* **val** sumBtAccumulator = *sc*.longAccumulator(**"bt50"**)  **val** sumLtAccumulator = *sc*.longAccumulator(**"lt50"**)    result.foreach(x => {  **if** (x.\_2 > 50) sumBt50 += 1 **else** sumLt50 += 1  **if** (x.\_2 > 50) sumBtAccumulator.add(1) **else** sumLtAccumulator.add(1)  *println*(x)  })  *println*(**s"大于50的用户数:$**{sumBt50}**,小于50的用户数:$**{sumLt50}**"**)  *println*(**s"大于50的用户数:$**{sumBtAccumulator.value}**,小于50的用户数:$**{sumLtAccumulator.value}**"**)  *println*(recordNumberAccumulator.value) } |

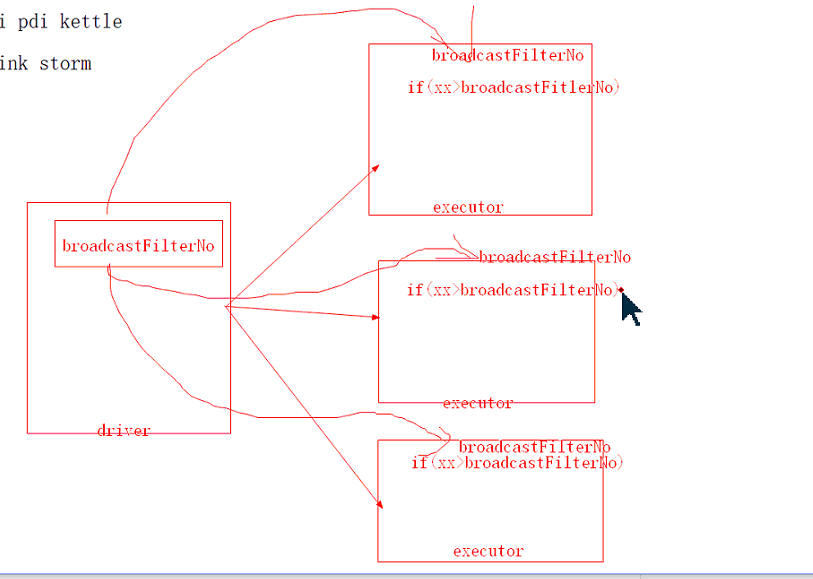
### 广播变量(broadcast variable)

定义格式：broadcast[T](value: T)(implicit arg0: ClassTag[T]): Broadcast[T]

1. 在driver上定义赋值，在execute上读取使用

2. Broadcast variable对execute来说是只读的

3. 一般在driver上计算或者获取某个值，然后广播给execute来使用



|  |
| --- |
| **val** *conf* = **new** SparkConf().setMaster(**"local[\*]"**).setAppName(**"ShareVariable"**) **val** *sc* = SparkContext.*getOrCreate*(*conf*)  *// 广播变量* **def** broadcastTest(filterNo: Int) = {  *// 计算每个用户访问次数,然后过滤掉访问次数小于filterNo的记录* **val** rdd = *sc*.textFile(**"/bd14/user-logs-large.txt"**)  .map(x => {  (x.split(**"\\t"**)(0), 1)  })  .reduceByKey(\_ + \_)  *// 把filterNo声明成广播变量* **val** broadcastFilterNo = *sc*.broadcast(filterNo)  *// 在execute使用广播变量，而不直接使用变量值* **val** result = rdd.filter(x => x.\_2 > broadcastFilterNo.value)  result.foreach(*println*) } |