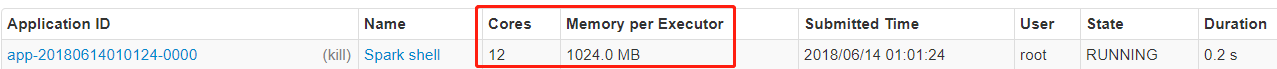
# spark任务运行的资源DIY



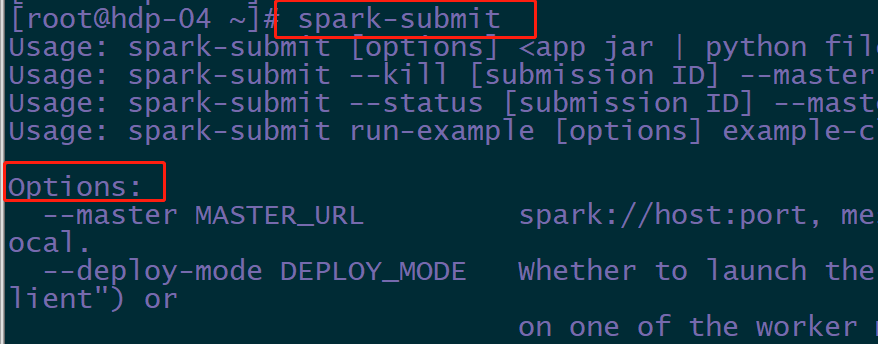
默认启动任务时，executor占用worker中的所有的cores,每一个executor占用1g内存。

默认情况下，一个worker启动一个executor

## spark-submit任务提交时的常用选项：

spark-submit 选项 jar包 参数

spark-submit –master spark://hdp-01:7077 --class xxxx jar包 参数



--master 指定使用哪一种部署模式

--class application使用的main class

--name application 使用的名称

--jars 使用第三方的jar包 写入mysql

--conf k-v 指定参数配置 k-v yarn的配置

--executor-cores 每一个executor使用的cores

--executor-memory 每一executor使用memory

--total-executor-cores 所有的executor使用的cores

如果我们在程序中

val conf = new SparkConf()

conf.setAppName(“xxx”)

这种指定优先级更高。 conf.setxxx --name --master

conf.setMaster(“local[\*]”)

--master spark://hdp-01:7077

spark任务的调度方式： FIFO

## 需要资源自定义：

executor的cores executor的memory

--executor-cores 每一个executor使用的cores 资源的最低要求

--executor-memory 每一executor使用memory 资源的最低要求

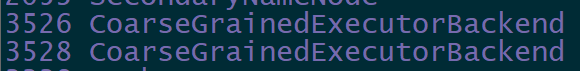
--total-executor-cores 所有的executor使用的cores 一个application使用的所有的cores 最高标准

我的资源：

3台worker 4cores 2.7g 内存

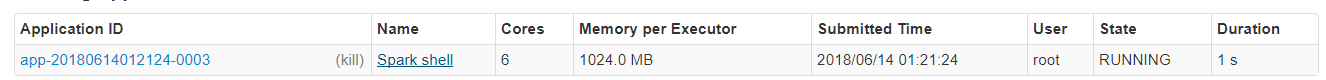
executor-cores 2 executor-memory 1g

启动几个executor呢? 3 6 5 6个executor



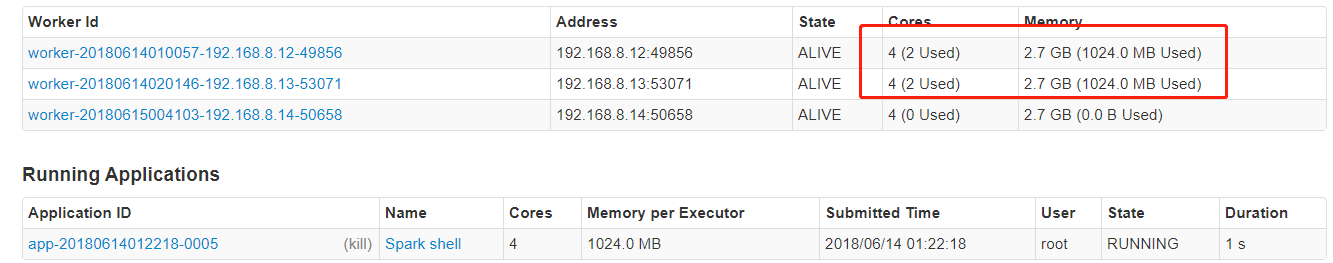
executor-cores 3 executor-memory 2g 3个executor

executor-cores 2 executor-memory 1g --total-executor-cores 7 3个 executor



executor-cores 2 executor-memory 1g --total-executor-cores 1000 还是6个

executor-cores 2 executor-memory 1g --total-executor-cores 4 2个 executor 一台机器上启动2个，还是在2台机器上，各启动一个。



# RDD 简述

RDD是spark中的基本的抽象的计算模型。

RDD( Resilient Distributed Dataset) **弹性分布式的数据集**。

弹性： 容错的概念。

分布式： 分区的概念。不同的分区中的数据，可能运行在不同机器上。

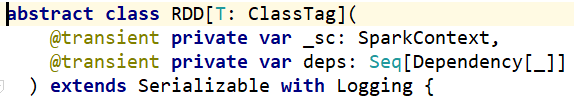
数据集合： 存储数据。

不可变，只读的，被分区的数据集。

类似于scala的本地集合，有很多的方法可以调用。操作起来和本地集合一样。

RDD还有5大特性。

落地到代码中，是一个抽象类： RDD



# RDD创建

创建rdd有3种方式：

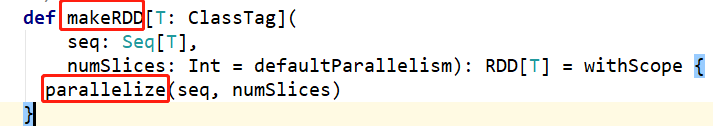
## 集合并行化

测试

把本地集合(Seq) ----》 RDD

|  |
| --- |
| scala> val arr = Array(List(1,3),List(4,6))  arr: Array[List[Int]] = Array(List(1, 3), List(4, 6))  scala> val rdd2 = sc.makeRDD(arr)  rdd2: org.apache.spark.rdd.RDD[List[Int]] = ParallelCollectionRDD[2] at makeRDD at <console>:26  scala> val rdd3 = sc.parallelize(arr) |

把本地集合中的外层集合去掉，生成RDD[数据类型]

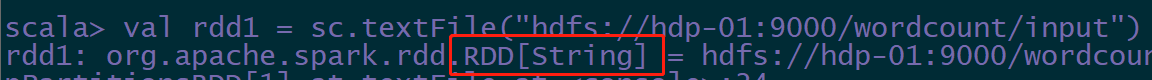


## 读取外部文件系统

实际中最常用的方法。

sc.textFile(“”)

可以读hdfs文件，也可以读取本地文件



## 调用转换类的算子

只要调用transformation类的算子，都会生成一个新的RDD

RDD中的数据类型，由传入给算子的函数的返回值类型决定。

一个是集合类型（RDD）,但是数据类型

|  |
| --- |
| **val** file: RDD[String] = sc.textFile(input) **val** map: RDD[Array[String]] = file.map(\_.split(**" "**)) *// 切分并压平* **val** words: RDD[String] = file.flatMap(\_.split(**" "**)) |

强调： action类的算子，不会生成rdd。

# RDD的分区

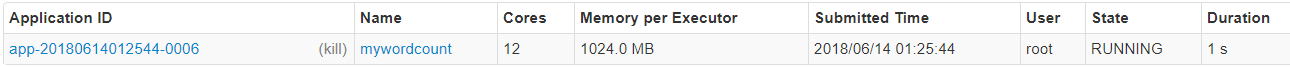
## 查看分区的API

被分区。

查看rdd的分区，rdd1.**partitions**.size

## 集合并行化得到的rdd的分区：

|  |
| --- |
| scala> val arr = Array(List(1,3),List(4,6))  arr: Array[List[Int]] = Array(List(1, 3), List(4, 6))  scala> val rdd3 = sc.parallelize(arr)  rdd3: org.apache.spark.rdd.RDD[List[Int]] = ParallelCollectionRDD[3] at parallelize at <console>:26  scala> rdd3.partitions.size  res0: Int = 12 |



默认情况下，一个application使用多个cores，就有多少个分区。

**分区的数量 = 运行任务的可用的cores**

（默认一个cores，能处理一个任务）

可以指定分区的数量：

val rdd3 = sc.makeRDD(arr,**分区的数量**)

|  |
| --- |
| scala> val rdd3 = sc.parallelize(arr,3)  rdd3: org.apache.spark.rdd.RDD[List[Int]] = ParallelCollectionRDD[4] at parallelize at <console>:26  scala> rdd3.partitions.size  res1: Int = 3  scala> val rdd3 = sc.parallelize(arr,11113)  rdd3: org.apache.spark.rdd.RDD[List[Int]] = ParallelCollectionRDD[5] at parallelize at <console>:26  scala> rdd3.partitions.size  res2: Int = 11113 |

这种方式，多用于测试。

## 读取外部文件RDD的分区

正常情况下，读取hdfs中的文件，默认情况下，读到的文件有几个block块，得到的rdd就有几个分区。

当读取一个文件，不足一个block块的时候，会是2个分区。

默认情况下，分区的数量 = 读取文件的block块的数量

分区的数量至少是2个。

|  |
| --- |
| scala> val rdd1 = sc.textFile("hdfs://hdp-01:9000/wordcount/input")  rdd1: org.apache.spark.rdd.RDD[String] = hdfs://hdp-01:9000/wordcount/input MapPartitionsRDD[7] at textFile at <console>:24  scala> rdd1.partitions.size  res3: Int = 3  scala> val rdd1 = sc.textFile("hdfs://hdp-01:9000/wordcount/input/a.txt")  rdd1: org.apache.spark.rdd.RDD[String] = hdfs://hdp-01:9000/wordcount/input/a.txt MapPartitionsRDD[9] at textFile at <console>:24  scala> rdd1.partitions.size  res4: Int = 2 |

textFile自身提供了修改分区的API

sc.textFile(path,**分区的数量**)

1. 这里的分区数量，不能少于 读取的数据的block块的数量。
2. 当设置分区的数量大于block块的数量的时候，读取数据的API会多我们的数据进行优化。

真正的想要改变分区的数量：

用算子。

repartition，coalesce，专用于修改分区数量。

读取hdfs上的数据，写入到hdfs中的数据，使用的APi，都是hadoop的API

总结：

默认情况下，分区的数量 = 读取文件的block块的数量

分区的数量至少是2个。

**rdd1.partitions.size**

## 通过转换类的算子

默认情况下，分区的数量是不变的。 map flatMap filter

groupByKey,reduceByKey 默认是不变的，但是可以通过参数来改变。

repartition(分区数量) ,coalesce(分区数量), 根据指定的分区数量重新分区。

union： 分区数量会增加。

|  |
| --- |
| scala> val rdd2 = rdd1.flatMap(\_.split(" ")).map((\_,1))  rdd2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[14] at map at <console>:26  scala> val rdd3 = rdd2.reduceByKey(\_+\_)  rdd3: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[15] at reduceByKey at <console>:28  scala> rdd2.partitions.size  res7: Int = 3  scala> rdd3.partitions.size  res8: Int = 3  scala> val rdd3 = rdd2.reduceByKey(\_+\_,6)  rdd3: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[16] at reduceByKey at <console>:28  scala> rdd3.partitions.size  res9: Int = 6 |

总结：

集合并行化：

val arr = Array[Int](1,4,5,6) –》 sc.makeRDD(arr) RDD[Int]

默认情况下， 分区数量 = application使用的 cores

sc.makeRDD(data,分区数量)

读取HDFS数据：

默认情况下， 分区数量 = 读取的数据的block块的数量

至少是2个

通过转换类的算子获取的RDD ：

默认情况下，分区的数量是不变的。

简单来说，rdd分区数量就决定了任务的并行的数量。

# RDD的算子

## 综述：

转换类的算子 Transformation

生成新的rdd，lazy执行的。

所有的transformation只有遇到action才能被执行。

行动类的算子 action

立即触发任务的执行，不会生成rdd

把数据写入到相应的介质，展示结果数据（把收集到driver）

## Transformation

### map

一一映射

对某一个rdd执行map，每一条数据执行操作。

返回值的数据类型，取决于 传递的函数的返回值类型。

rdd中有几条数据，就会被迭代几次。

|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List(1,4,2,5,7,8),3)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at makeRDD at <console>:24  scala> val rdd2 = rdd1.map(\_>5)  rdd2: org.apache.spark.rdd.RDD[Boolean] = MapPartitionsRDD[1] at map at <console>:26  scala> rdd2.collect  [Stage 0:> (0 + 0) / res1: Array[Boolean] = Array(false, false, false, false, true, true)  scala> rdd1.partitions.size  res2: Int = 3  scala> rdd2.partitions.size  res3: Int = 3 |

map总结：

1. map是一一映射，rdd中有几条数据，就会被迭代几次
2. map操作的返回值类型，由函数返回值类型来决定
3. rdd的分区的数量，是不变的。

### mapValues

scala中的mapValues Map集合

spark中的mapValues ，作用于 RDD[K,V] ,Key保持不变。

|  |
| --- |
| scala> val rdd = sc.makeRDD(List(("reba",100),("fengjie",80)))  rdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[2] at makeRDD at <console>:24  scala> rdd.mapValues(\_\*100)  res4: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at mapValues at <console>:27  scala> val rdd3 = rdd.mapValues(((t:Int)=> t\*100))  rdd3: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[4] at mapValues at <console>:26  scala> rdd3.collect  res5: Array[(String, Int)] = Array((reba,10000), (fengjie,8000)) |

mapValues总结；

mapValues得到的rdd的分区数量是不变的。

类似于map，作用于RDD[k-v]类似的v。key保持不变。

### mapPartitions

作用于每一个rdd的分区。

传递的函数是一个迭代器

有几个分区，就会迭代几次

|  |
| --- |
| **val** conf =**new** SparkConf() .setMaster(**"local[\*]"**) .setAppName(**this**.getClass.getSimpleName) **val** sc = **new** SparkContext(conf)  **val** rdd1 = sc.makeRDD(*List*(1,4,2,5,7,8),3)  *// zip 拉链操作 // zipWithIndex 索引 从0 开始* **val** index: RDD[(Int, Long)] = rdd1.zipWithIndex() index.mapValues(t=>t)  **val** partitions = rdd1.mapPartitions(t => {  t.map(\_ \* 10) }) partitions |

### mapPartitionsWithIndex

带分区编号的算子。分区编号从0开始。

|  |
| --- |
| **val** rdd1 = sc.makeRDD(*List*(1,4,2,5,7,8),3)  *// mapPartitions* **val** rdd2: RDD[Int] = rdd1.mapPartitions(t => {  t.map(\_ \* 10) }) *// 定义一个函数，返回rdd中的数据，以及对应的分区编号* **val** f=(i:Int,it:Iterator[Int])=> {  it.map(t=> **s"p=$**i**,v=$**t**"**) } **val** rdd3: RDD[String] = rdd1.mapPartitionsWithIndex(f) *println*(rdd3.collect().toBuffer) |

### flatMap

flatMap = map + flatten

得到新的rdd的分区数量不变。

|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List("hello spark","hello word"))  rdd1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at makeRDD at <console>:24  scala> val rdd2 = rdd1.flatMap(\_.split(" "))  rdd2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at flatMap at <console>:26  scala> rdd2.partitions.size  res1: Int = 12  scala> rdd1.partitions.size  res2: Int = 12 |

### filter

过滤出所有的满足条件的元素

分区数量不变，即使某些分区没有数据，但是分区是依然存在的。

|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List(1,3,4,5,6,7),3)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at makeRDD at <console>:24  scala> val rdd2 = rdd1.filter(\_>5)  rdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[3] at filter at <console>:26  scala> rdd2.partition  partitioner partitions  scala> rdd2.partitions.size  res3: Int = 3  scala> val f=(i:Int,it:Iterator[Int])=>  | it.map(t => s"p=$i,v=$t")  f: (Int, Iterator[Int]) => Iterator[String] = <function2>  scala> rdd2.mapPartitionsWithIndex(f).collect  res4: Array[String] = Array(p=2,v=6, p=2,v=7) |

### groupBy,groupByKey,reduceByKey

|  |
| --- |
| **def** groupBy[K](f: T => K)(**implicit** kt: ClassTag[K]): RDD[(K, Iterable[T])] |

|  |
| --- |
| **val** sc: SparkContext = *MySpark*(**this**.getClass.getSimpleName)  **val** rdd1: RDD[Int] = sc.makeRDD(*List*(1, 4, 5, 6)) *// groupBy RDD[K] RDD[K,V]* **val** groupedRdd: RDD[(Int, Iterable[Int])] = rdd1.groupBy(t => t) **val** groupedRdd2: RDD[(String, Iterable[Int])] = rdd1.groupBy(t => t.toString)  *// groupBy 返回值类型[(K,Iterable[T])] K：是传入的函数的返回值类型 T 是rdd的元素类型* **val** rdd2: RDD[(String, Int)] = sc.makeRDD(*List*((**"rb"**, 1000), (**"baby"**, 990),  (**"yangmi"**, 980), (**"bingbing"**, 5000), (**"bingbing"**, 1000), (**"baby"**, 2000)), 3)   *// 返回值的类型* **val** rdd3: RDD[(String, Iterable[(String, Int)])] = rdd2.groupBy(\_.\_1)  **val** result1: RDD[(String, Int)] = rdd3.mapValues(\_.map(\_.\_2).sum)   *// groupByKey RDD[K,V] Iterable[990,2000]* **val** rdd4: RDD[(String, Iterable[Int])] = rdd2.groupByKey() *println*(**s"rdd4 part = $**{rdd4.partitions.size}**"**)  **val** result3: RDD[(String, Int)] = rdd4.mapValues(\_.sum)  *println*(rdd2.groupByKey(10).partitions.size)   *// reduceByKey RDD[K,V]* **val** rdd6: RDD[(String, Int)] = rdd2.reduceByKey(\_ + \_) *// 指定生成的rdd的分区的数量* **val** rdd7: RDD[(String, Int)] = rdd2.reduceByKey(\_ + \_, 10)  **val** rdd5: RDD[(String, List[Int])] = sc.makeRDD(*List*((**"a"**, *List*(1, 3)), (**"b"**, *List*(2, 4)))) rdd5.reduceByKey(\_ ++ \_) |

重点比较reduceByKey和groupByKey:

1,都作用于 RDD[K,V]

2，都是根据key来分组聚合

1. 默认，分区的数量都是不变的，但是都可以通过参数来指定分区数量

不同点：

1. groupByKey默认没有聚合函数，得到的返回值类型是RDD[ k,Iterable[V]]
2. reduceByKey 必须传聚合函数 得到的返回值类型 RDD[(K,聚合后的V)]
3. groupByKey().map() = reduceByKey

最重要的区别：

reduceByKey 会进行分区内聚合，然后再进行网络传输

groupByKey 不会进行局部聚合

结论：

**如果这两个算子，都可以使用，优先使用reduceByKey**

### sortBy 和 sortByKey

sortBy

按照指定条件进行排序

sortByKey

按照key进行排序

|  |
| --- |
| **val** rdd1: RDD[(String, Int)] = sc.makeRDD(*List*((**"a"**, 1), (**"b"**, 11), (**"c"**, 123)))  *// rdd1.sortBy(-\_.\_2)* **val** result1: RDD[(String, Int)] = rdd1.sortBy(\_.\_2, **false**)  *// rdd1.sortBy(\_.\_2,false).collect().foreach(println)* rdd1.sortByKey(**false**).collect().foreach(*println*) |

### 集合的交集并集和差集

并集 **union**

#intersection求交集

# subtract 求差集

**union得到的rdd的分区的数量 = 参与union的rdd的分区数量之和**

|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List(1,3,2,4,6,7),3)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[21] at makeRDD at <console>:24  scala> val rdd2 = sc.makeRDD(List(1,11),3)  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[22] at makeRDD at <console>:24  scala> val rdd3 = rdd1 union rdd2  rdd3: org.apache.spark.rdd.RDD[Int] = UnionRDD[23] at union at <console>:28  scala> rdd3.collect  res37: Array[Int] = Array(1, 3, 2, 4, 6, 7, 1, 11)  scala> rdd3.partitions.size  res38: Int = 6  scala> rdd1.intersection(rdd2).collect  [Stage 41:> (0 + 3) / [Stage 41:======================================> (2 + 1) / res39: Array[Int] = Array(1)  scala> rdd1.subtract(rdd2).collect  res40: Array[Int] = Array(3, 6, 4, 7, 2) |

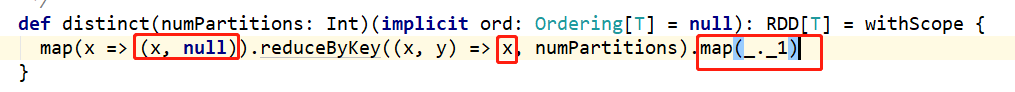
### distinct

集合中的元素去重的算子,distinct算子，默认分区数量是不变的，但是可以传参数改变分区的数量。

rdd1:RDD[Int]

map(x=>(x,null)).reduce

底层调用的是reduceByKey



## action

### foreach

一一映射， 对集合中的每一条数据执行某些操作。

foreach 和map 有什么区别：

foreach: Unit 常用于打印结果

map 有返回值

map 是转换类的算子，foreach是action类的算子。

### foreachPartition

每次迭代一个分区的数量。

|  |
| --- |
| scala> val rdd2 = sc.makeRDD(Array(1, 3, 4, 5, 6, 7),3)  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[5] at makeRDD at <console>:24  scala> rdd2.foreach(println)  [Stage 3:=======================================> (2 + 1) /  scala> rdd2.foreachPartition  foreachPartition foreachPartitionAsync  scala> rdd2.foreachPartition(println)  scala> rdd2.foreachPartition(it=>println(it.mkString("-"))) |

### 总结

集群模式下，foreach 和 foreachPartition打印的结果，都在executor中。

map(t=>{})

mapParititons()

foreach()

**foreachPartition**()

collect 之后，再写 最low 效率最低

数据，分析之后，存入到mysql中，哪一种算子最合适？

不需要返回值，

如果使用mapPartititions算子，还需要调用action类的算子。

foreach() 每一条数据，都要获取mysql的连接，

foreachPartition 一个分区的数据，共用一个连接。

10分区 100000 条数

### 常用的action算子：

reduce 归并： 得到的结果数据顺序是不确定的。

数据分布在不同的executor，收集的时候，顺序不确定。

|  |
| --- |
| scala> val rdd2 = sc.makeRDD(List("a","b","c"))  rdd2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[17] at makeRDD at <console>:24  scala> rdd2.reduce(\_++\_)  res26: String = bac  scala> rdd2.reduce(\_++\_)  res27: String = cba  scala> rdd2.reduce(\_++\_)  res28: String = abc  scala> rdd2.partitions.size  res29: Int = 12 |

|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List(1,3,4,5))  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[6] at makeRDD at <console>:24  scala> rdd1.reduce(\_+\_)  res10: Int = 13  scala> rdd1.count  res11: Long = 4  scala> val rdd1 = sc.makeRDD(List(11,13,14,5,1,6))  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[7] at makeRDD at <console>:24  scala> rdd1.first()  res12: Int = 11  scala> rdd1.take(3)  res13: Array[Int] = Array(11, 13, 14)  scala> rdd1.top(3)  res14: Array[Int] = Array(14, 13, 11)  scala> rdd1.takeOrdered(3)  res15: Array[Int] = Array(1, 5, 6)  scala> val rdd2 = sc.makeRDD(List(11,13,11,1,1,6))  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[10] at makeRDD at <console>:24  scala> rdd2.zipWI  zipWithIndex zipWithUniqueId  scala> val rdd3 = rdd2.zipWithIndex  rdd3: org.apache.spark.rdd.RDD[(Int, Long)] = ZippedWithIndexRDD[11] at zipWithIndex at <console>:26  scala> rdd3.countByKey()  res16: scala.collection.Map[Int,Long] = Map(13 -> 1, 1 -> 2, 6 -> 1, 11 -> 2) |

action类的算子，会触发任务的执行？

spark-submit spark-shell ------🡪 Application

正常情况下，调用一次action，就会产生一个job

还有一些特殊的算子：

take

sortBy

zipWithIndex

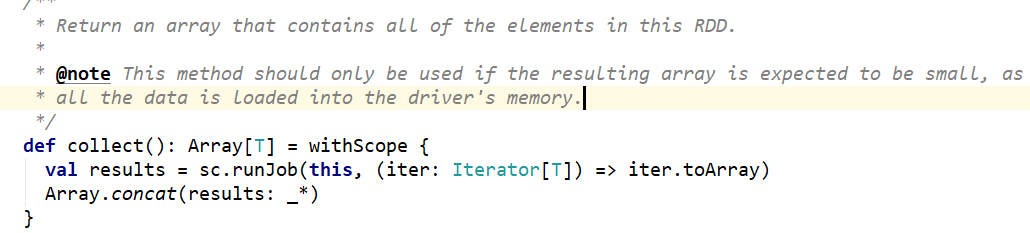
checkpoint

### collect,collectAsMap

collect: 把数据从executor端收集到Driver，返回值类型是Array

collectAsMap 返回值类型是Map 只能作用于RDD[K,V]

|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List(("a",1),("b",2)))  rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[15] at makeRDD at <console>:24  scala> rdd1.collect  res21: Array[(String, Int)] = Array((a,1), (b,2))  scala> rdd1.collectAsMap  res24: scala.collection.Map[String,Int] = Map(b -> 2, a -> 1) |



collect:

把数据收集到Driver端，有序的。

# 今日重点：

spark-submit的资源DIY

RDD 如何创建 rdd的分区

rdd的算子

wordcount --- 》 groupBy groupByKey mapValues

作业题：

求全局的TopN,分组TopN