# 安装

配置hadoop.Env.sh

JAVA\_HOME=/usr/local/src/jdk1.8.0\_45

SCALA\_HOME=/opt/scala-2.10.4

HADOOP\_HOME=/opt/hadoop-2.7.3

HADOOP\_CONF\_DIR=/opt/hadoop-2.7.3/etc/hadoop

SPARK\_MASTER\_HOST=hadoop

配置slaves

仅把slaver的hostname加进去

# SparkContext

Spark上下文对象是spark程序的主入口，负责连接到spark cluster，可用创建RDD，在集群上创建累加器和广播变量，每个JVM只能激活一个sparkContext对象，创建新的sparkContext对象是必须stop原来的sparkContext对象

RDD

弹性分布式数据集

类名：org.apache.spark.rdd.RDD

是不可变的，可分区的集合，可进行并行操作，该类包含了所有可用于RDD之上的基本操作，如map，filter，persist

在rdd内部，每个RDD都有5个特征

1 有一个分区列表

2 对每个切片的计算函数

3 存放parent的RDD依赖列表

4 （可选）基于key-value的分区器

5 （可选）首选位置列表

Spark-shell

1 默认使用local模式运行spark程序，没有用到spark的集群，类似于hadoop的本地模式

Spark-shell --master local[4] //表示在本地模式启动线程出5，模拟集群

Val lines=sc.textFile(“file:///x/xxx/x/xx.txt”)

Val rdd2=lines.map(x=>{

Val tname=Thread.currentThread().getName();

Println(tname+”:”+x)

x

}

Rdd2.count

2 每个spark应用有driver构成，由它启动各种并行操作

Driver含有main函数和分布式数据集，并对他们应用各种操作。

Spark-shell就是driver实例（包含main方法）

Driver通过sparkContext（简写sc）来访问spark，代表和spark集群的链接

运行程序时，驱动程序需要管理一些叫做executor 的节点

Standalone Applications

不用spark-shell，写scala app

使用maven编译运行scala的spark程序

1 编写scala源代码

# RDD

1 不可变基础

2 每个RDD切割成分区，每个分区在不同的节点上计算

3 创建RDD的两种方式

A 加载外部数据集，sc.textFile（）从hadoop或者hive，hbase等

B 从父类rdd转换得到新的rdd

4 RDD的两种操作类型

Transformations：从前一个rdd产生一个新的rdd，例如map，filter操作

Lazy延迟操作，只有actions操作才触发运算

Actions:基于rdd计算一个结果/返回值给driver/存储文件到

Count/first/take 会触发spark提交job

## 持久化

rdd.persist() 持久化到内存（默认），也可以到磁盘，为了rdd的重用

Rdd.unpsersist()解除持久化

1 spark默认持久化对象到jvm head中是没有串行化的

2 如果是off-head或者磁盘存储必须是串行化

3 持久化级别：

MEMORY\_ONLY

MEMORY\_ONLY\_SER

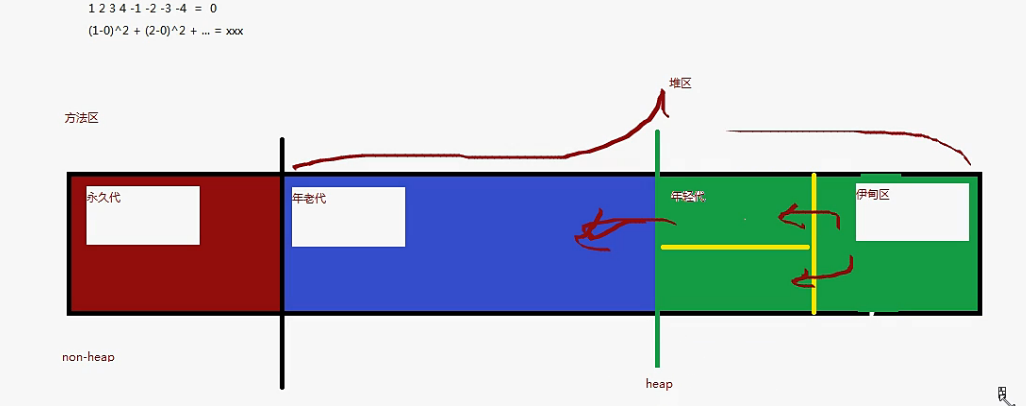
MEMORY\_AND\_DISK

MEMORY\_AND\_DISK\_SER

非串行化占用内存空间高，但占用cpu时间少

串行化占用内存少，但占用cpu时间高

|  |
| --- |
| **val** *rdd*=sc.parallelize(*List*(**"tom"**,**"gao"**,**"jian"**)) **import** org.apache.spark.storage.\_ *rdd*.persist(StorageLevel.DISK\_ONLY) |



## 基本的action

1 count()获取元素的个数

2 rdd.collect()返回rdd所有的元素

3 rdd.countByValue()计算每个元素出现的次数，返回Map

|  |
| --- |
| **val** *rdd*=sc.parallelize(*List*(**"wode"**,**"wode"**,**"spark"**,**"hadoop"**,**"hadoop"**,**"hadoop"**)) *rdd*.countByValue() 输出：scala.collection.Map[String,Long] = *Map*(spark -> 1, hadoop -> 3, wode -> 2) |

4 rdd.first() 提取rdd的第一个元素

5 rdd.take(n) 提取rdd的前n个元素

6 rdd.top(n)提取末尾的n个元素

7 rdd.takeOrdered(3) 提取前三个元素并排序

8rdd.Sample() 采样

9 fold(n)(x=>T) 类似于reduce不过带了初始值

10 aggregate((n,n),((n=>T),(n=>T)))

11 foreach(func)

均值：mean()

集合中所有数的平均值

方差：variance()

集合中每个数减去平均数后平方再相加，方法越下波动越小

方根：

集合中每个数减去平均数后平方后再开方再相加

堆区：

## Key-value操作

1创建pairRDD

**val** *pair*=rdd.map(x=>(x.split(**" "**),x))

2 pairRDD操作

### GroupByKey

|  |
| --- |
| **var** *rdd1* = sc.makeRDD(*Array*((**"A"**,0),(**"A"**,2),(**"B"**,1),(**"B"**,2),(**"C"**,1))) *rdd1*.groupByKey().collect *//结果：Array[(String, Iterable[Int])] = Array((B,CompactBuffer(1, 2)), (A,CompactBuffer(0, 2)), (C,CompactBuffer(1)))* |

GroupBy

groupBy（）用于不成对的数据，我们希望使用不同的条件除了当前键的相等性。需要传入一个函数

### reduceByKey

|  |
| --- |
| **val** *rdd2*=sc.parallelize(*List*((**"gao"**,2),(**"jian"**,1),(**"le"**,1),(**"yue"**,2),  (**"yue"**,1),(**"le"**,3))) *rdd2*.reduceByKey((x,y)=>x+y).collect  结果：Array[(String, Int)] = Array((jian,1), (le,4), (yue,3), (gao,2))  *//指定分区数* **val** *rdd4*=*rdd2*.reduceByKey(**new** org.apache.spark.HashPartitioner(2),\_+\_) *//查看分区数量* rdd3.partitions.size |

### MapValues,flatMapValues

|  |
| --- |
| **val** *rdd1*=sc.parallelize(*List*((**"gao"**,1),(**"jian"**,2),(**"feng"**,3))) *rdd1*.mapValues(\_+1).collect *//结果: Array[(String, Int)] = Array((gao,1), (jian,2), (feng,3))*  rdd1.flatMapValues(x=>x to 5).collect *//结果：Array[(String, Int)] = Array((gao,1), (gao,2), (gao,3), (gao,4),* (gao,5), (jian,2), (jian,3), (jian,4), (jian,5), (feng,3), (feng,4), (feng,5)) |

### Keys(可排序的)

|  |
| --- |
| **val** *rdd1*=sc.parallelize(*List*((**"gao"**,1),(**"jian"**,2),(**"feng"**,3)))  rdd.keys.collect 结果：Array[String] = Array(gao, jian, feng) |

### Values

|  |
| --- |
| **val** *rdd1*=sc.parallelize(List((**"gao"**,1),(**"jian"**,2),  (**"feng"**,3))) *rdd1*.values.collect  结果： Array[Int] = Array(1, 2, 3) |

## Two Key-Value操作

**val** *rdd1*=sc.parallelize(*List*((**"le"**,1),(**"yue"**,2),(**"gao"**,3)))  
**val** *rdd2*=sc.parallelize(*List*((**"gao"**,1),(**"jian"**,2),(**"feng"**,3)))

### subtractByKey求差集

|  |
| --- |
| *rdd1*.subtractByKey(*rdd2*) *//结果：Array[(String, Int)] = Array((yue,2), (le,1))* |

### Join

|  |
| --- |
| rdd1.join(*rdd2*) *//结果：Array[(String, (Int, Int))] = Array((gao,(3,1)))* |

### rightOuterJoin与做外链接

|  |
| --- |
| **val** *rdd1*=sc.parallelize(*List*((**"le"**,1),(**"yue"**,2),(**"gao"**,3))) **val** *rdd2*=sc.parallelize(*List*((**"gao"**,1),(**"jian"**,2),(**"feng"**,3)))  *rdd1*.rightOuterJoin(*rdd2*).collect *//结果：Array[(String, (Option[Int], Int))] = Array((jian,(None,2)),  (feng,(None,3)), (gao,(Some(3),1)))*  *rdd1*.leftOuterJoin(*rdd2*).collect  *//结果：Array[(String, (Int, Option[Int]))] = Array((le,(1,None)), (yue,(2,None)), (gao,(3,Some(1))))* |

### Cogroup

对多个rdd的数据进行分组，可以实现块之间的链接

|  |
| --- |
| **val** *rdd1*=sc.parallelize(*List*((**"le"**,1),(**"yue"**,2),(**"gao"**,3))) **val** *rdd2*=sc.parallelize(*List*((**"gao"**,1),(**"jian"**,2),(**"feng"**,3)))  *rdd1*.cogroup(*rdd2*).collect *//结果：Array[(String, (Iterable[Int], Iterable[Int]))] = Array( // (jian,(CompactBuffer(),CompactBuffer(2))),  // (feng,(CompactBuffer(),CompactBuffer(3))),  // (le,(CompactBuffer(1),CompactBuffer())),  // (yue,(CompactBuffer(2),CompactBuffer())),  // (gao,(CompactBuffer(3),CompactBuffer(1))))* |

aggregations聚合

**val** *rdd1*=sc.parallelize(*List*((**"le"**,1),(**"yue"**,2),(**"gao"**,3),(**"le"**,2),(**"gao"**,2))

**MapValues操作key-value中的value值**

|  |
| --- |
| rdd1.mapValues(x=>(x,1)).reduceByKey((x,y)=>(x.\_1+y.\_1,x.\_2+y.\_2)).collect *//结果：Array[(String, (Int, Int))] = Array((le,(3,2)), (yue,(2,1)), (gao,(5,2)))* |

### \*CountByValue

**操作列表中的值，计算每个值出现的次数，速度比上面的reduceByKey更快**

|  |
| --- |
| **val** rdd1=sc.parallelize(*List*(**"gao"**,**"wo"**,**"ni"**,**"jian"**,**"ni"**,**"san"**,**"gao"**)) rdd1.countByValue()**//不用collect操作** scala.collection.Map[String,Long] = *Map*(gao -> 2, ni -> 2, san -> 1, wo -> 1, jian -> 1) |

### combineByKey（）

类似于hadoop的combine，大多数情况下用combineByKey比group+reduce块很多。

**遍历分区中的元素，每个元素要么有一个之前没有见过的键，要么具有与前一个元素相同的键，如果它是一个新的元素，combineByKey（）使用我们提供的函数，称为createCombiner（），为该键上的累加器创建初始值。 它的重要的是要注意，这发生在第一次在每个分区中找到一个键，而不是比只有第一次在RDD中找到key。如过当我们处理分区的时候发现一个以前没见过的值，则用提供的方法mergeValue（），把累加器中的该键的当前值+1由于每个分区是独立处理的，我们可以有多个累加器用于相同的密钥。 当我们合并来自每个分区的结果时，如果两个或更多分区具有相同键的累加器，我们使用用户提供的mergeCombiners（）函数合并累加器。我们可以在combineByKey（）中禁用map-side合，如果我们知道我们的数据不会受益于它。 例如，groupByKey（）禁用map-side聚合函数不会保存任何空间。 如果我们想禁用map-side combine，我们需要指定分区器; 现在你可以只使用**

|  |
| --- |
| **val** input=sc.parallelize(List((**"coffee"**,1),(**"coffee"**,2),(**"panda"**,3),(**"coffee"**,9)))//求平均数的例子 **val** result=input.combineByKey(  (v)=>(v,1),**//createCombine**  (acc:(Int,Int),v)=>(acc.\_1+v,acc:\_2+1),**//mergeValue** (acc1:(Int,Int),acc2(Int,Int))=>(acc1.\_1+acc2.\_1,acc1.\_2+acc2.\_2)**//mergeCombines** ).map{**case** (key,value)=>(key,value.\_1/value.\_2.toFloat)}  result.collectAsMap().map(*println*(\_))  执行过程：  **//partition1:**(**"coffee"**,1),(**"coffee"**,2),(**"panda"**,3)  则(**"coffee"**,1)->new key accumulators[coffee]=createCombiner(1)  (**"coffee"**,2)->existing key ccumulators[coffee]=mergeValue(accumulators[coffee],2)  (**"panda"**,3)->new key accumulators[panda]=createCombiner(3)  **//partition2**:(**"coffee"**,9)  (**"coffee"**,9)-> new key accumulators[coffee]=createCombiner(9)  **最后：merge Partitions:**  mergeCombines(partition1.accumulators[coffe],partiton2.accumulators[coffee] |

### sortByKey（）自定义排序

默认的排序为升序排序，可以接受一个自定义函数，自定义排序

|  |
| --- |
| **import** org.apache.spark.rdd.RDD  **val** rdd:RDD[(Int,String)]=sc.parallelize(Array((9,**"jian"**),(34,**"laohu"**),(6,**"xiaoyu"**),(21,**"dafeng"**))) **implicit val** sortIntegersByString=**new** Ordering[Int]{  **override def** compare(a:Int,b:Int)=a.toString.compare(b.toString) } rdd.sortByKey().collect |

### \*Tuning the level of parallelism

**调整并发度**

有时，我们想改变分组上下文之外的RDD的分区和聚合操作。 对于这些情况，Spark提供了repartition（）函数，其在网络上打乱数据以创建新的分区集合。 但是重新分区你数据是一个相当昂贵的操作。 Spark还有一个优化repartition（）的版本称为coalesce（）允许避免数据移动，但是它会减少RDD分区的数量。 可以使用rdd.partitions.size（）检查RDD的大小（在Python中用rdd.getNumPartitions（））来确定是否可以安全的调用coalesce，并确保你是合并它比目前有更少数量的分区。

rdd.repartition(4,**true**) //不推荐 true为重新洗牌

rdd.coalesce(4,**true**)

Repartition(4)==hadoop的job.setNumreduceTasks(4)

当大多数的key值都相同时，不管分区数为多少，相同key都会进入同一分区里面，其他区就闲置了，就会造成数据倾斜，（可以给key加前缀后缀来解决数据倾斜）

### \*优化：

有时候使用聚合函数比先使用groupByKey()然后使用reduce（）或者fold（）更高效，

例如：rdd.reduceByKey(func)能产生和下面的函数同样的结果：

rdd.groupByKey().mapValues(value => value.reduce(func))

但是它更高效，因为它避免了为每个键创建值列表的步骤。

## Actions Pair RDD

与转换一样，基本RDD上可用的所有传统操作也可用于对RDD。 对RDD上可以使用一些附加动作以利用数据的键/值本质

会触发计算

### CountByKey计算key出现的个数

|  |
| --- |
| **val** rdd=sc.parallelize(List((1,2),(3,4),(3,6))) rdd.countByKey() *//结果：scala.collection.Map[Int,Long] = Map(1 -> 1, 3 -> 2)* |

### collectAsMap 把元祖列表变成map

|  |
| --- |
| **val** rdd=sc.parallelize(*List*((1,2),(3,4),(3,6))) rdd.collectAsMap *//结果：scala.collection.Map[Int,Int] = Map(1 -> 2, 3 -> 6)* |

### Lookup（3）找出所有键锁对应的值

|  |
| --- |
| **val** rdd=sc.parallelize(List((1,2),(3,4),(3,6))) rdd.lookup(3) *//结果：Seq[Int] = WrappedArray(4, 6)* |

## 分区Data Partitioning

在分布式程序中，通信非常昂贵，所以最小化网络流量可以大大提高性能，Spark可以选择控制其rdd的分区来减少通讯，当数据集在面向键的操作（例如联接）中重复使用多次时分区才会有意义。 虽然Spark不明确控制每个键到哪个工作节点（部分是因为系统被设计为工作，即使特定节点失败），但是程序确保一组键将一起出现在某个节点上。

例如，定时合并用户表（UserId,UserInfo）和用户过去五分钟访问网站点击的信息表（UserId,LinkInfo）用join合并：

|  |
| --- |
| **def** sc=**new** SparkContext() **val** userData=sc.sequenceFile[UserID,UserInfo](**"hdfs://"**).persist() **def** processNewLogs(logFileName:String)={  **val** event=sc.sequenceFile[UserID,UserInfo](logFileName)  **val** joined=userData.join(events)  **val** offTopicVisits=joined.filter{  **case** (userId,(userInfo,linkInfo))=>!userInfo.topics.contains(linkInfo.topic)  }.count()  *print*(**"number of visits to non-subscribed topics:"**+offTopicVisits) } |

上面的代码可以正常运行但是效率低下，因为processNewLogs方法每次调用join操作时都不知道key在数据集上是怎么分区的，默认情况下，会散列两个数据集的所有键，并将key具有相同hash值的元素传输到同一台机器上，然后再把相同键的元素链接到一起，所以即使UserData表没有变化，每次调用join时UserData表都会被散列和重新在网络上传输。

解决这个问题很简单，就是在程序一开始的时候就使用partitionBy（）方法把UserData数据变换到hash-partition ，可以通过spark.HashPartitioner 对象来partitionBy：

|  |
| --- |
| **def** UserData=sc.sequenceFile[UserID,UserInfo](**"hdfs://..."**)  .partitionBy(**new** HashPartitioner(100))  .persist() |

这样在spark可以知道UserData已经提前哈希分区好了，再调用userData.join（events）时，spark只会对events RDD进行shuffle，把对应的events的元素传输到UserData相对应的散列分区的机器，减少了网络传输。另外：分区数的大小与有多少个并行的任务在rdd上执行有关，所以，通常分区数应该至少不小于集群的核数。

**注意：**

**用partition分区后没有持久化，将导致后续其他的操作都会重新对rdd进行分区和shuffle。许多的spark操作都会自动产生已知分区信息的rdd，而且除了join（）外还有的许多操作都会利用分区信息。例如sortByKEey产生rangepartitioned范围分区，groupByKey产生散列分区。而另外一些操作如map会导致rdd无视父级分区信息，因为理论上它可以改变键的值。**

### 查看是否分区：

用pairs.partitioner方法，返回一个scala.Option对象，调用isDefined()查看它是否有值，用get()获取这个值。可以用这种方法测试那些操作会影响rdd的分区。

|  |
| --- |
| scala> val pairs = sc.parallelize(List((1, 1), (2, 2), (3, 3))) pairs: spark.RDD[(Int, Int)] = ParallelCollectionRDD[0] at parallelize at <console>:12 scala> pairs.partitioner res0: Option[spark.Partitioner] = None scala> val partitioned = pairs.partitionBy(new spark.HashPartitioner(2)) partitioned: spark.RDD[(Int, Int)] = ShuffledRDD[1] at partitionBy at <console>:14 scala> partitioned.partitioner res1: Option[spark.Partitioner] = Some(spark.HashPartitioner@5147788d) |

### 分区对操作的好处

Spark的许多操作都涉及shuffle，如果rdd已分区，则下面的这些操作都会受益：

cogroup（），groupWith（），join（），leftOuterJoin（），rightOuterJoin（），

groupByKey（），reduceByKey（），combineByKey（）和lookup（）

对于单个rdd的操作如reduceByKey（），在已经分好区的rdd上操作将只会在单个节点行计算每个键的值，最后把每个worker计算出的值传给master节点。

对于二进制操作，两个rdd的，如cogroup（）和join操作，与分区使得至少有一个rdd不被shuffle，如果两个rdd具有相同的分区器，并且他们被缓存在同一节点上，则不会发生网络传输shuffle。

### 影响分区的操作

Spark知道那些操作会导致分区，而且会自动的对那些会产生分区rdd的操作设置分区器。例如spark会对join操作设置hash分区器，的结果是已经hash分区的rdd，则在这个结果rdd上进行reduceByKey操作，将会快很多。但是对于有些操作spark不会设置分区，例如对已经hash分区的rdd进行map（）操作，map（）理论上是可以改变key值的，所以spark不会对map（）的结果rdd设置分区器。代替的方法是mapValues和flatMapValues，这两个方法可以保证key值不变，所以spark会对其结果设置分区器。

Spark会设置分区器的操作：

cogroup（），groupWith（），join（），leftOuterJoin（），rightOuterJoin（），

groupByKey（），reduceByKey（），combineByKey（），partitionBy（），sort（），mapValues（如果父RDD有分区器），flatMapValues（）（如果父代有分区器）和

filter（）（如果父级有一个分区器）

其他的操作会产生没有分区的结果

对于两个rdd的操作，在输出上设置哪个分区器取决于父RDDs的分区器。默认情况下，它是一个哈希分区器，分区数设置为操作的并行性水平。如果两个父rdd中有一个有分区器设置，它将是分区器;如果父母都有分区器，它就会是第一个父的分区器。

### Pagerank例

Pagerank算法是通过计算有多少了文档指向一个网页来测量这个网页的排名。

PageRank是一个执行许多连接的迭代算法，因此是一个很好的rdd分区用例。这个算法维护两个数据集：

（pageID，LinkList）,包含每页的邻居页列表元素

（pageID,rank）包含每页的排名元素

1.将每页的排名初始化为1.0。

2.在每次迭代时，页面p发送rank（p）/ numNeighbors（p）的贡献到它的邻居（它链接的页面）。

3.将每页的排名设置为0.15 + 0.85 \* contributionReceived

最后两个步骤重复几次迭代，在此期间算法将收敛到每个页面的正确的PageRank值。 在实践中，它通常运行约10次迭代

|  |
| --- |
| **val** links **=** sc.objectFile[(**String**, **Seq**[**String**])](“links”) .partitionBy(**new HashPartitioner**(100)) .persist()**var** ranks **=** links.mapValues(v **=>** 1.0)**for** (i **<-** 0 until 10) { **val** contributions **=** links.join(ranks).flatMap { **case** (pageId, (links, rank)) **=>** links.map(dest **=>** (dest, rank / links.size)) }r anks **=** contributions.reduceByKey((x, y) **=>** x + y).mapValues(v **=>** 0.15 + 0.85\*v) } *//* ranks.saveAsTextFile(“ranks”) |

## 自定义分区

虽然Spark的HashPartitioner和RangePartitioner很适合许多场合，但有时候我们还是需要自定义分区器来实现想要的分区。

要实现自定义分区器，您需要继承org.apache.spark.Partitioner类并实现三种方法：

numPartitions：Int，返回要创建的分区数。

getPartition（key：Any）：Int，返回分区ID（0到numPartitions-1）。

equals（），标准的Java平等方法。 这是很重要的实现，因为Spark将需要测试你的Partitioner对象与其自身的实例它决定两个RDD是否以相同的方式分区！

例：哈希每个url的域名，基于域名的分区器

|  |
| --- |
| **class** DomainNamePartitioner(numParts: Int) **extends** Partitioner {  **override def** numPartitions: Int = numParts  **override def** getPartition(key: Any): Int = {  **val** domain = **new** Java.net.URL(key.toString).getHost()  **val** code = (domain.hashCode % numPartitions)  **if** (code < 0) {  code + numPartitions   } **else** {  code  }  }  **override def** equals(other: Any): Boolean = other **match** {  **case** dnp: DomainNamePartitioner =>  dnp.numPartitions == numPartitions  **case** \_ =>  **false** } } |

# Loading And Saving Data

## FileFormat

### Text Files

**val** input **=** sc.textFile(“file:///home/holden/repos/spark/README.md”)

如果是多个小文件有两种处理方式：1 .把所有的小文件加载到一个目录下，然后用textFile把路径传入。2. 如果文件足够的小，可以使用SparkContext.wholeTextFiles()方法获取键值对的rdd，其中键是文件名，值是文件的内容。当每个文件代表一定的时间段时的数据，wholeTextFiles（）可能非常有用。 如果我们有代表不同时期的销售数据的文件，我们可以很容易

计算每个周期的平均值

输入的文件支持通配符(e.g., part-\*.txt)

计算每个文件的平均值

|  |
| --- |
| **val** input=sc.wholeTextFiles(**"file://home/holden/salesFiles"**) **var** result=input.mapValues{  y=>**val** nums=y.split(**" "**).map(x=>x.toDouble)  nums.sum/nums.size.toDouble } |

存text数据

Result.saveAsTextFile(**"file://home/holden/salesFiles"**)

spark允许多个节点输同一目录下，但这种方法我们不能控制文件中最终以哪个数据段结束输出，但其他输出格式允许。

### Json

将json数据当做文本文件读取，然后解析json数据，可以使用mapPartitions（）来重用解析器。例如用jackson包

|  |
| --- |
| **import** com.fasterxml.jackson.module.scala.DefaultScalaModule **import** com.fasterxml.jackson.module.scala.experimental.ScalaObjectMapper **import** com.fasterxml.jackson.databind.ObjectMapper **import** com.fasterxml.jackson.databind.DeserializationFeature  **case class** Person(name: String, lovesPandas: Boolean) **val** result = input.flatMap(record => {  **try** {  *Some*(mapper.readValue(record, *classOf*[Person]))  } **catch** {  **case** e: Exception => None  }}) |

输出json数据 例如：过滤出那些喜欢熊猫的人

|  |
| --- |
| result.filter(p **=>** P.lovesPandas).map(mapper.writeValueAsString(**\_**)) .saveAsTextFile(outputFile) |

### 逗号分隔(csv)或制表符分隔的值

Comma-separated value (CSV)

加载csv文件

|  |
| --- |
| **case class Person**(name**: String**, favoriteAnimal**: String**) **val** input **=** sc.wholeTextFiles(inputFile) **val** result **=** input.flatMap{ **case** (**\_**, txt) **=> val** reader **= new CSVReader**(**new StringReader**(txt)); reader.readAll().map(x **=> Person**(x(0), x(1))) } |

写出csv文件

|  |
| --- |
| pandaLovers.map(person **=> List**(person.name, person.favoriteAnimal).toArray) .mapPartitions{people **=> val** stringWriter **= new StringWriter**(); **val** csvWriter **= new CSVWriter**(stringWriter); csvWriter.writeAll(people.toList) **Iterator**(stringWriter.toString) }.saveAsTextFile(outFile) |

### sequenceFile

SequenceFile是hadoop常用的格式

用sequenceFile(path, keyClass, valueClass, minPartitions) 读取sequence

或者sequenceFile [Key，Value]（path，minPartitions）自动转换Writable类型

|  |
| --- |
| **val** data=sc.sequenceFile(inFile,classOf[Text],classOf[IntWritable]). map{**case** (x,y)=>(x.toString,y.get())} |

输出sequenceFile

如果想要输出VIntWritable可以在save前进行map操作转换类型

|  |
| --- |
| **val** data = sc.parallelize(List((**"Panda"**, 3), (**"Kay"**, 6), (**"Snail"**, 2))) data.saveAsSequenceFile(outputFile) |

### ObjectFile

对象文件是一个关于SequenceFiles的一个看似简单的包装，它允许我们保存只包含值的RDD。 与SequenceFiles不同，对于对象文件，使用Java序列化写出值

读取：SparkContext上的函数objectFile（）

写出：saveAsObjectFile

### Hadoop Input and Output Formats

1

|  |
| --- |
| *//加载输入格式应用旧版hadoopAPI* **val** input=sc.hadoopFile[Text,Text,KeyValueTextInputFormat](inputPath).map{  **case** (x,y)=>(x.toString,y.toString) }  *//加载lzo压缩格式的json数据用Elephant Bird，MapWritable代表一个json对象* **val** input=sc.newAPIHadoopFile(inputPath,*classOf*[LzoJsonInputFormat],  *classOf*[LongWritable],*classOf*[MapWritable],conf) |

用hadoop的输出格式输出saveAsNewAPIHadoopFile

除了hadoopFile（）和saveAsHadoopFile（）家族的功能，你可以使用hadoopDataset / saveAsHadoopDataSet和newAPIHadoopDataset / saveAsNewAPIHadoopDataset访问Hadoop支持的存储不是文件系统的格式，例如hbase和mongoDB，hadoopDataSet需要一个Configuration对象设置数据源所需要的hadoop属性。

## 结构化数据hive和json

Spark SQL可以加载Hive支持的任何表。要将Spark SQL连接到现有Hive安装，您需要提供一个Hive组态。 可以通过将hive-site.xml文件复制到Spark的./conf/目录来实现

|  |
| --- |
| **def** sc=**new** SparkContext() **import** org.apache.spark.sql.hive.HiveContext **val** hiveCtx=**new** HiveContext(sc) **val** rows=hiveCtx.sql(**"SELECT name,age FROM users "**) **val** first=rows.first() *println*(first.getString(0)) |

用Spark SQL加载json数据

|  |
| --- |
| **val** tweets **=** hiveCtx.jsonFile(“tweets.json”) tweets.registerTempTable(“tweets”) **val** results **=** hiveCtx.sql(“SELECT user.name, text FROM tweets”) |

## Mysql hbase数据库

1 、用Jdbc读取数据，可以读取不同节点上不同范围的数据

MapRow参数把读到的每一行数据转换成我们想要的格式，如果不提供这个参数，则默认会转化成Array对象

|  |
| --- |
| **def** createConnection()={  Class.*forName*(**"com.mysql.jdbc.Driver"**).newInstance()  DriverManager.*getConnection*(**"jdbc:mysql://hadoop/test"**); } **def** extractValues(r:ResultSet)={  (r.getInt(1),r.getString(2)) } **val** *data*=**new** JdbcRDD(*sc*,*createConnection*,  **"SELECT \* FROM panda WHERE ? <= id AND id <= ?"**,  lowerBound = 1,upperBound = 3,numPartitions = 2,mapRow = *extractValues*) *print*(*data*.collect().toList) |

2 、Spark可以通过其Hadoop输入格式访问HBase，实现在org.apache.hadoop.hbase.mapreduce.TableInputFormat类。 这种输入格式返回键/值对，其中键是类型org.apache.hadoop.hbase.io.ImmutableBytesWritable

值为类型org.apache.hadoop.hbase.client.Result。

Result类包括各种方法用于根据其列系列获取值，如其API文档中所述。

要使用Spark与HBase，您可以调用SparkContext.newAPIHadoopRDD与正确输入格式，

|  |
| --- |
| **import** org.apache.hadoop.hbase.HBaseConfiguration **import** org.apache.hadoop.hbase.client.Result **import** org.apache.hadoop.hbase.io.ImmutableBytesWritable **import** org.apache.hadoop.hbase.mapreduce.TableInputFormat **val** conf=HBaseConfiguration.*create*() conf.set(TableInputFormat.*INPUT\_TABLE*,**"tablename"**) **val** rdd=*sc*.newAPIHadoopRDD(conf,*classOf*[TableInputFormat],  *classOf*[ImmutableBytesWritable],*classOf*[Result]) |

# 高级sprk编程

两种类型的共享变量，这是可以在Spark任务中使用的特殊类型的变量：

累加器来聚集信息和广播变量以有效地分布大值

当我们的任务涉及大的建立时间（例如创建数据库连接或随机数生成器）时，跨多个数据项共享此设置工作很有用。使用远程调用符号查找数据库，我们检查如何重用安装工作在每个分区上

## Accumulators

计数器有时可以更简单的聚合我们想要的数据，而不用单独的去运行filter或者reduce

计数器的value只对master节点可读，对于worker键值只有可写属性。

例 计算文本中的空行数：

|  |
| --- |
| **val** file=*sc*.textFile(**"file.txt"**) *//创建一个计数器并初始化为0* **val** balnkLines=*sc*.accumulator[Int](0) **val** callSigns=file.flatMap(line=>{  **if**(line==**""**){  balnkLines+=1  }  line.split(**""**) })  *//只有在执行action动作后才会看到正确的计数值* callSigns.saveAsTextFile(**"output.txt"**) *println*(**"Blank lines:"**+balnkLines.value) |

### 计数器和容错

Spark通过自动重新执行任务来处理失败或缓慢的任务。例如，如果运行map（）操作的分区的节点崩溃，Spark会重新运行在另一个节点上;并且即使节点不崩溃但是简单地慢得多比其他节点，Spark可以抢先启动任务的“推测”副本另一个节点，如果完成则取得其结果。Spark可能必须重新运行一个任务来重建一个内存不足的缓存值。 因此，最终结果是相同的函数可以对相同的数据运行多次，这取决于集群上发生的情况。

在action操作中使用的累加器，Spark将每个任务的更新只应用到每个累加器一次。所以我们要将累加器放入像foreach（）这样的方法中。

自定义计数器可以实现AccumulatorParam ，sum和max是常用的交换和关联操作计数器。

## Broadcast Variables广播变量

允许程序有效的像所有worker节点发送只读value用于一个或多个spark 操作。例如你的应用程序要发送只读的机器学习中的大特征向量到所有worker节点。Spark会自动将你的闭包中引用的所有变量发送到工作节点。虽然这很方便，但它也可能是低效的，因为（1）默认值任务启动机制针对小任务大小进行了优化，（2）你可能实际上使用了相同的变量在多个并行操作，但Spark会为它们分别发送它们操作。

广播变量只是类型spark.broadcast.Broadcast [T]的对象，它包装类型T的值。我们可以通过调用我们任务中的Broadcast对象的值来访问这个值。 值被发送到每个节点一次，使用高效的、BitTorrent-like通信机制。

广播变量只有driver可写，worker只可读。

广播的值的序列化格式很重要，要极快速又紧凑，可以通过spark.serializer属性选择不同的序列化库来优化。

|  |
| --- |
| **//创建一个broadcast，访问通过.value**  **val** signPrefixes = sc.broadcast(loadCallSignTable()) **val** countryContactCounts = contactCounts.map { **case** (sign, count) =>  **val** country = lookupInArray(sign, signPrefixes.value)  (country, count) }.reduceByKey((x, y) => x + y) countryContactCounts.saveAsTextFile(**" / countries.txt "**) |

## 以多个分区为基础工作

同时为rdd的多个分区上为分区建立例如数据库链接多很浪费资源的，我们可以进执行一次操纵就为多个分区建立链接，比如共享链接池，重用json解析器。可以使用mapPartitions（）方法来实现，它为rdd的每个分区提供一个iterator并且返回一个iterator的结果。

|  |
| --- |
| **val** validSigns=*sc*.makeRDD(*List*()) **val** contractsContactLists=validSigns.distinct().mapPartitions{  signs=>  *//创建一个解析json的mapper* **val** mapper=createMapper()  **val** client=**new** HttpClient()  client.start()  signs.map { *//创建请求* createExchangeForSign(sign)  }.map{**case** (sign,exchange)=> *//获取响应* (sign,readExchangeCallLog(mapper,exchange))  }.filter(x=>x.\_2!=**null**)*//过滤空日志* } |

MapPartitions f:(Iterator[T])->Iterator[U]

mapPartitionsWithIndex f:(Int,Iterator[T])->Iterator[T] 提供分区数量

foreachPartition f:(Iterator[T])->Nothing

## 数字rdd操作

StatsCounter的统计性方法：

Count() 返回rdd的元素数量

Mean（） 元素的平均值

Sum（）元素的和

Max（） min（） 最大值 和最小值

Variance() 方差

SampleVariance（）方法取样

Stdev() 标准偏差

SampleStdev 标准差取样

调动：**val** statObject=rdd.stats()  
 statObject.stdev

# Spark集群

## 运行结构

在集群模式上，spark使用master/workers即一个中央协调driver和其他多个分布式的executors，driver和executor都有自己独立的java进程。

Spark集群通过cluster manager管理集群，spark内置一个cluster master叫做Standalone

,spark也可以工作在hadoop的yarn和apache的Mesos上两个开源的集群管理。

### Driver

Driver是你程序运行main（）方法的进行。它是你使用代码创建SparkContext、创建RDDS、执行transformations和actions的进程。当你启动一个spark shell的时候实际上就是启动了以driver进程，并且初始化了一个SparkContext对象叫做sc，一旦你的driver终止，则你的应用程序也就停止了。

Driver运行的时候有两个职责：

1 、把你的程序代码转换成任务（tasks）

Spark根据你的输入创建RDDs，然后你可以进行transformations和actions操作来转换和保存数据。一个spark程序会隐式的创建一个又向无环图（DAG），当driver运行的时候，会把又向无环图转换成一个物理的执行计划。spark会执行几个优化：例如将执行图转换成一组阶段（stages），每个阶段又包含多个task，tasks被捆绑和准备发送到集群，task是Spark中最小的工作单元，一个程序可以启动数百上千个task。

2 、在executors上调度任务

给定一个执行计划，driver必须要协调调度executors上的tasks，当executors启动的时候，它会把自己注册到driver上，所以它能随时获取程序的完整视图。每一个executors都是一个执行任务和存储数据的进程，driver把 任务按照数据存储的配置合理的分配给executors，driver也会跟踪数据的缓存位置，并且在未来的任务调度中使用该数据。

### Executor

Spark执行器是负责在给定的Spark作业中运行单个任务的工作进程。 Executor在Spark应用程序开始时启动一次，通常运行在应用程序的整个生命周期，但是Spark应用程序可以继续当executor失败的时候。 Executor有两个角色。 首先，他们运行组成的任务应用程序并将结果返回给driver。 第二，他们通过每个executor中寄存的Block Manager服务来提供对RDD的内存缓存。 因为RDD直接在执行器内部缓存，任务可以与缓存的数据并排运行。

### Cluster Manager

Spark依赖集群管理器来启动executors，在某些情况下也会启动driver程序。集群管理器在spark中是一个可插拔的组件，spark可以运行在不同的集群管理器上，如YARN、MESOS还有自带的Standalone。

### 运行步骤

1. 用户通过spark-submit提交应用程序。
2. Spark-submit启动含有用户指定main（）的程序为driver进程。
3. Driver联系cluster Manager请求资源去启动executors。
4. Cluster Manager代表driver启动executors
5. Driver进程根据用户的程序运行，基于RDD的actions和transformations，driver将以task的形式向executor发送工作
6. Tasks运行在executor进程来计算和存储结果。
7. 如果driver的main方法exit或者调用SparkContext.stop() ，将会终止executors并且释放集群资源。

## 通过spark-submit来提交应用程序

如果不指定参数，则这个应用程序将会以本地模式运行，提交一个python应用程序：

Bin/spark-submit my\_script.py

不过想要运行在standalone cluster上，则需要加参数指定master，和运行在每个executer上的内存大小：

Bin/spark-submit -master spark://hadoop:7077 -executor-memory 10g my\_srcipt.py

-master参数如果是spark：//... 表示运行在Standalone cluster上，也可以是其他类型：

Spark://host:port 连接到StandAlone cluster上，默认端口为7077

Mesos://host:port 链接到MESOS Cluster上，默认端口为5050

Yarn 连接到hadoop集群上，当运行在yarn上的时候你需要设置HADOOP\_CONF\_DIR环境变量指定hadoop的配置文件的位置

Local 链接到本地模式用单核

Local[N] 链接到本地用n个核

Local[\*] 运行在本地模式，用本地机器上的所有核

### Spark-submit的格式

**Bin/spark-submit [options] <app jar|python file> [app options]**

App options是提供给应用程序的main函数的参数。

Options的参数列表：

-master 指定要链接的集群

-deploy-mode 指定是在本地启动driver（client模式）还是在集群上的worker节点上启动driver（cluster模式）。client模式spark将在调用spark-submit的机器上启动driver，cluster模式spark将把driver放在在集群中的worker上执行，默认为client模式。

-class 如果运行的是jar则用来指定main方法

-name 应用程序的名字

-jars 应用程序所依赖的jar

-files 将放置到应用程序的目录。是你想要放在集群每个节点上的数据

-executor-memory 每个executor占用的内存大小，以bytes不单位，可以指定为m或者g

-driver-memory driver所占用的内存大小

-total-executor-cores 总共的executor的核数

-conf 以prop-value的形式提供参数,优先级比代码中的低，如spark.shuffle.manager=hash

-properties-file 包含key/value

例如提交一个程序到standalone cluster

|  |
| --- |
| ./bin/spark-submit **\** —master spark://hostname:7077 **\** —deploy-mode cluster **\** —class com.databricks.examples.SparkExample **\** —name “Example Program” **\** —jars dep1.jar,dep2.jar,dep3.jar **\** —total-executor-cores 300 **\** —executor-memory 10g **\** myApp.jar “options” “to your application” “go here” |

# SparkSQL

## DataFrame和DataSet

DataSet是分布式数据集合，DataSet是spark在1.6版本中添加的新接口，它提供了Rdd的优点以及SparkSql的执行引擎的优点，DataSet可以有jvm对象创建，然后使用map、flatmap、filter等转换操作。DataSet类似于rdd，但是，dataset使用专门的编译器来序列化对象进行处理或者网络传输。虽然编译器和标准序列化都是将对象转换为字节，但是编译器是动态生成的代码，允许spark在不反序列化成对象就能够进行filter、hashing、sorting等操作。

DataFrame是组织成命名列的DataSet，概念类似于关系型数据库的表，但是具有更丰富的优化。DataFrame可以从各种来源构建，例如结构化数据文件，hive中的表，外部数据库或现有的rdd。在scala和java中DataFrame由Rows的DataSet表示，在scala中DataFrame只是DataSet[Row] 的类型别名，而在java中用DataSet<Row>表示DataFrame.

SparkSession取代了原来的SqlContext和HiveContext。SparkSession可以兼容两者，原来的sqlContext和hiveContext仍然保留。

### 初始化sparkSql

可以依赖hivesql构建:

<**groupId**>org.apache.spark</**groupId**>  
<**artifactId**>spark-hive\_2.10</**artifactId**>  
<**version**>1.6.2</**version**>

完全使用sparksql不适用hive依赖：

<**groupId**>org.apache.spark</**groupId**>  
<**artifactId**>spark-sql\_2.10</**artifactId**>  
<**version**>1.6.2</**version**>

要将SparkSql链接到现有的hive，必须将hivesite.xml文件复制到Spark配置目录conf下，如果没有hive，spark仍然能够启动，spark将创建自己的 hive Metastore元数据库在程序工作目录。

创建hiveContext或者hiveContext需要创建一个SparkContext

|  |
| --- |
| **val** sc=**new** SparkContext() **val** hiveCtx=**new** HiveContext(sc)  *// 如果您有现有的Hive安装，并且已将hive-site.xml文件复制到$ SPARK\_HOME / conf， // 你也可以只运行hiveCtx.sql来查询你现有的Hive表* **val** schemaRdd=hiveCtx.read.json(**"inputpath"**) **val** result=hiveCtx.sql(**"select text,retweetcount from tweet order by text limit 10"**)  **////或者用spark2.0后的**  **val** session=SparkSession  .*builder*()  .appName(**"Spark Sql Example"**)  .config(**"spark.some.config.option"**,**"value"**)  .getOrCreate() *//隐式转换，例如将rdd转换为dataFrame* **import** session.implicits.\_ |

### 创建Dataframe和 operations

*//用rdd创建一个dataFrame*

|  |
| --- |
| **val** colors=*List*(**"while"**,**"green"**,**"yellow"**,**"red"**,**"brown"**,**"pink"**) **val** color\_df=sc.parallelize(colors).map(x=>(x,x.length)).toDF(**"color"**,**"length"**) color\_df.show |

*//创建DataFrame From JSON*

**val** df=sqlContext.read.json(**"./authors.json"**)  
 df.show

*//创建FataFrame From Mysql*

将mysql驱动jar放入$SPARK\_HOME/lib/下

启动spark-shell带mysqlDriver

spark-shell --driver-class-path /path-to-mysql-jar/mysql-connector-java-5.1.34-bin.jar

|  |
| --- |
| **Val** df2=sqlContext.read.format(**"jdbc"**).options(Map(**"url"**->**"jdbc:mysql://ha**  **doop"**,**"dbtable"**->**"mydb1.people"**,**"user"**->**"root"**,**"password"**->**"root"**)).load() df2.show |

*//创建DataFrame From Parquet*

|  |
| --- |
| **val** df=sqlContext.read.parquet(**"/root/users.parquet"**)  scala> df.show +------+--------------+----------------+ | name|favorite\_color|favorite\_numbers| +------+--------------+----------------+ |Alyssa| **null**| [3, 9, 15, 20]| | Ben| red| []|  +------+--------------+----------------+} |

|  |
| --- |
| **val** colors=*List*(**"while"**,**"green"**,**"yellow"**,**"red"**,**"brown"**,**"pink"**) **val** colorDf=sc.parallelize(colors).map(x=>(x,x.length)).toDF(**"color"**,**"length"**) *//查看类型* colorDf.dtypes *//查看row count* colorDf.count *//检索全表数据* colorDf.show *//检索所有列名* colorDf.columns *//删除列* colorDf.drop(**"length"**).show  //查找列  df.select(**"name"**).show()  df.select(**$"name"**, **$"age"** + 1).show() *//转换成json格式* colorDf.toJSON.first() *//filter类似以SQL中的where条件,alias类似于SQL中的as起别名* colorDf.filter(colorDf(**"length"**).between(4,5)).select(colorDf(**"color"**).alias(**"mid\_length"**)).show colorDf.filter(colorDf(**"length"**)>4).filter(colorDf(**"color"**)!==**"white"**).show *//排序* colorDf..sort(**"color"**).show colorDf.filter(colorDf(**"length"**)>=4).sort(**$"length"**,**$"color"**.desc).show *//orderBy可以代替sort* colorDf.orderBy(**"length"**,**"color"**).take(4) *//对一个或多个列内部排序* colorDf.sort(colorDf(**"lentgh"**).desc,colorDf(**"color"**).asc).show *//groupBy,操作多行，必须是宽依赖，而且经常涉及shuffling* colorDf.groupBy(**"length"**).count().show *//删除一行数据* **val** df1=sqlContext.read.json(**"./authors.txt"**) **val** df2=df1.na.drop() df2.show |

### 临时表

创建临时表并查询

|  |
| --- |
| df.createOrReplaceTempView(**"people"**) df.createTempView(**"people"**)  **val** sqlDF=session.sql(**"select \* from people"**) sqlDF.show() |

创建dataSet

|  |
| --- |
| **case class** Person(name:String,age:Long)  **val** caseClassDS=Seq(Person(**"andy"**,32)).toDS()  **val** primitiveDS = *Seq*(1, 2, 3).toDS() primitiveDS.map(\_ + 1).collect()  **val** path = **"examples/src/main/resources/people.json" val** peopleDS = session.read.json(path).as[Person] peopleDS.show() |

### **用样例类模式创建DataFrame**

Spark SQL的Scala接口支持将包含案例类的RDD自动转换为DataFrame。 case类定义了表的模式。 case类的参数的名称使用反射读取，并成为列的名称。 case类也可以嵌套或包含复杂类型，如Seqs或Arrays。 此RDD可以隐式转换为DataFrame，然后注册为表。 表可以在后续的SQL语句中使用

|  |
| --- |
| **case class** Person(name:String,age:Long) **object** DataSetCreate {  **def** main(args: Array[String]) {  **val** session=SparkSession  .*builder*()  .appName(**"people"**)  .master(**"spark://hadoop:7077"**)  .getOrCreate()  *//可以将rdd隐式转换为dataFrame* **import** session.implicits.\_  **val** peopleDF=session.sparkContext  .textFile(**"F:/mydata/wordcount.txt"**)  .map(\_.split(**","**))  .map(attribute=>*Person*(attribute(0),attribute(1).trim.toInt))  .toDF()  *//把dataframe注册为临时表,表以case为模式* peopleDF.createOrReplaceTempView(**"people"**)  *//SQL statement* **val** teenagerDF=session.sql(**"select name,age from people where age between 13 and 19"**)  *//可以通过下标访问* teenagerDF.map(teenager=>**"name:"**+teenager(0)).show()  *//也可以通过字段名访问* teenagerDF.map(teenager=>**"Name:"**+teenager.getAs[String](**"name"**)).show()  *//为数据集明确定义编码* **implicit val** mapEncoder=org.apache.spark.sql.Encoders.*kryo*[Map[String,Any]]  **implicit val** stringIntMapEncoder: Encoder[Map[String, Int]] = *ExpressionEncoder*()  *// row.getValuesMap [T]立即检索多个列到Map [String，T]* teenagerDF.map(teenager=>teenager.getValuesMap[Any](*List*(**"name"**,**"age"**))).collect()  } } |

### **根据RowsRdd创建dataframe**

当case类不能提前定义时（例如，记录的结构被编码在一个字符串中，或者一个文本数据集将被解析，字段将被不同的用户投射），一个DataFrame可以用三个步骤 。

1.从原始RDD创建行的RDD;

2.创建由与第1步中创建的RDD中的Rows结构匹配的StructType表示的模式。

3.通过SparkSession提供的createDataFrame方法将模式应用于Rows的RDD

|  |
| --- |
| **val** session=SparkSession  .*builder*()  .getOrCreate() **import** session.implicits.\_ *//创建一个rdd* **val** peopleRdd=session.sparkContext.textFile(**"file:///opt/mydata/people.txt"**) *//schema用字符串编码* **val** schemaString = **"name age"** *//根据字符串生成模式* **val** fields=schemaString.split(**" "**)  .map(fieldName=>*StructField*(fieldName,StringType,nullable = **true**)) **val** schema=*StructType*(fields) *//把rdd的记录转换成rows* **val** rowRdd=peopleRdd  .map(\_.split(**","**))  .map(attributes=>*Row*(attributes(0),attributes(1).trim)) *//用schema创建dataFrame* **val** peopleDF=session.createDataFrame(rowRdd,schema) *//创建临时表* peopleDF.createOrReplaceTempView(**"people"**) **val** result=session.sql(**"select \* from people"**) result.map(attributes=>**"Name:"**+attributes(0)+**",age:"**+attributes(1)).show() |

## 数据源

### 常用的加载保存函数

Parquet是一种由许多其他数据处理系统支持的列式格式。 Spark SQL提供对读取和写入Parquet文件的支持，这些文件自动保留原始数据的模式。 在写Parquet文件时，出于兼容性原因，所有列都会自动转换为可空

|  |
| --- |
| **val** session=SparkSession.*builder*().getOrCreate() **import** session.implicits.\_ **val** userDf=session.read.load(**"file:///opt/mydata/user.parquet"**) userDf.select(**"name"**,**"age"**).write.save(**"file:///opt/mydata/user.parquet"**) |

### 手动**指定数据源**

你还可以手动指定要与您要传递给数据源的任何其他选项一起使用的数据源。 数据源由其完全限定名（即org.apache.spark.sql.parquet）指定，但对于内置源，您还可以使用其短名称（json，parquet，jdbc）。 从任何数据源类型加载的DataFrames可以使用此语法转换为其他类型。

|  |
| --- |
| **val** peopleDf=session.read.format(**"json"**).load(**"file:///opt/mydata/user.json"**) peopleDf.select(**"name"**,**"age"**).write.format(**"parquet"**).save(**"namesAndAges.parquet"**) |

### 直接对文件执行sql

可以直接对文件执行sql，而不用加载成dataframe

**val** sqlDF = session.sql(**"SELECT \* FROM parquet.`users.parquet`"**)

### 持久化到hive表

DataFrames也可以使用saveAsTable命令作为持久表保存到Hive元数据仓中。 注意，使用此功能不需要现有的Hive部署。 Spark将为您创建一个默认的本地Hive元数据仓库（使用Derby）。 与createOrReplaceTempView命令不同，saveAsTable将实现DataFrame的内容并创建指向Hive Metastore中数据的指针。 即使在Spark程序重新启动后，持久表仍将存在，只要您保持与同一个存储区的连接。 可以通过使用表的名称调用SparkSession上的表方法来创建永久表的DataFrame。

默认情况下，saveAsTable将创建一个“托管表”（内部表），这意味着数据的位置将由元数据仓控制。 托管表也将在删除表时自动删除其数据。

Hive和parquet区别

从表模式处理的角度来看，Hive和Parquet之间有两个主要区别。

Hive不区分大小写，而Parquet不是

Hive将所有列视为可空，而Parquet中的可空性很重要

由于这个原因，当将Hive Metastore Parquet表转换为Spark SQL Parquet表时，我们必须调整Hive Metastore模式与Parquet模式。 对帐规则为：

在两个模式中具有相同名称的字段必须具有相同的数据类型，而不管可否为空。 已对帐字段应具有Parquet侧的数据类型，以便遵守可空性。

已协调模式恰好包含在Hive metastore模式中定义的那些字段。

只有出现在Parquet模式中的任何字段都会被删除在已对帐模式中。

只有出现在Hive Metastore模式中的任何字段才会在已调节模式中添加为可空字段。

如 果这些表是由Hive或其他外部工具更新的，则需要手动刷新它们以确保一致的元数据。

spark.catalog.refreshTable("my\_table")

### 表分区：

请注意，自动推断分区列的数据类型。目前，支持数字数据类型和字符串类型。有时用户可能不想自动推断分区列的数据类型。对于这些用例，自动类型推断可以由spark.sql.sources.partitionColumnTypeInference.enabled配置，默认为true。当禁用类型推断时，字符串类型将用于分区列。

从Spark 1.6.0开始，默认情况下分区发现仅查找给定路径下的分区。对于上面的示例，如果用户将path / to / table / gender = male传递给SparkSession.read.parquet或SparkSession.read.load，则性别将不会被视为分区列。如果用户需要指定分区发现应该从哪个基本路径开始，他们可以在数据源选项中设置basePath。例如，当path / to / table / gender = male是数据的路径并且用户将basePath设置为path / to / table /时，gender将是一个分区列。