2019

企业级数据仓库实战

Spark RDD

学习资料：

Spark RDD基础：http://spark.apache.org/docs/latest/rdd-programming-guide.html

Scala基础：https://www.runoob.com/scala/scala-tutorial.html

主要内容：

1、什么是RDD
2、获取RDD的三种方式
3、常用的RDD算子
4、RDD缓存
5、共享变量
6、闭包
7、依赖
8、shuffle与优化

什么是RDD

The main abstraction Spark provides is a ***resilient distributed dataset*** (RDD), which is a collection of elements **partitioned** across the nodes of the cluster that can be **operated** on in **parallel**. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and **transforming** it. Users may also ask Spark to ***persist*** an RDD in **memory**, allowing it to be reused efficiently across parallel operations. Finally, RDDs **automatically recover** from node failures.

关键词：

RDD：resilient(弹性) distributed(分布式) dataset(数据集)

partition(分区)：每一个RDD包含的数据都会被存储在系统中的不同节点

parallel(并行)：天然支持并行化操作

transform(转换)：支持多种算子

persist(缓存)：支持多种缓存方式

recover(故障恢复)：可以根据DAG进行自动故障恢复

获取RDD的三种方式

1、并行化现有集合

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

```
scala> val data = Array(1, 2, 3, 4, 5)
data: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val distData = sc.parallelize(data)
distData: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[177] at parallelize at <console>:26
```

2、引用外部存储系统中的数据集

```
val lines = sc.textFile("data.txt")
```

```
scala> val lines = sc.textFile("/Users/[redacted]/data.txt")
lines: org.apache.spark.rdd.RDD[String] = /Users/[redacted]/data.txt MapPartitionsRDD[179] at textFile at <console>:24
```

3、已有的RDD转换获得

```
val data = lines.map(s => s.length)
```

```
scala> val data = lines.map(s => s.length)
data: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[180] at map at <console>:25
```

常用算子-transformation

map：将原有数据集中的每个元素经过函数处理，返回一个新的分布式数据集

map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.

```
scala> sc.parallelize(Array(1,2,3,4,5,6)).map(_ *2).collect
res105: Array[Int] = Array(2, 4, 6, 8, 10, 12)
```

filter：返回满足条件的新的数据集

filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.

```
scala> sc.parallelize(Array(1,2,3,4,5,6)).filter(_ >2).collect
res106: Array[Int] = Array(3, 4, 5, 6)
```

常用算子-transformation

flatMap:与map类型，但是每个输入项都可以映射到一个或者多个输出项

flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).

```
scala> sc.parallelize(Array(1,2,3,4,5,6)).flatMap(x => Array(x,x,x)).collect
res108: Array[Int] = Array(1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6)
```

MapPartitions:与map类型，但是是应用在每个分区

mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.

```
scala> sc.parallelize(List(1,2,3,4,5,6),2).mapPartitions{ x=>{
     |     var result = List[Int]()
     |     var i = 0
     |     while(x.hasNext){
     |         i += x.next()
     |     }
     |     result.::(i).iterator
     | }}.collect
res117: Array[Int] = Array(6, 15)
```

常用算子-transformation

mapPartitionsWithIndex:类似mapPartitions,只是func中多一个分区索引值

mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.

```
scala> sc.parallelize(List(1,2,3,4,5,6),2).mapPartitionsWithIndex{ (x,iter)=>{
     |     var result = List[String]()
     |     var i = 0
     |     while(iter.hasNext){
     |         i += iter.next()
     |     }
     |     result.::(x+":"+i).iterator
     | }}.collect
res119: Array[String] = Array(0:6, 1:15)
```

sample：采样，比例取决于fraction,withReplacement控制是否采用回置采样，seed为随机数种子

sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.

```
scala> val a = sc.parallelize(1 to 10000, 3)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[193] at parallelize at <console>:24
scala> a.sample(false, 0.1, 0).count
res120: Long = 1032
```

常用算子-transformation

union：返回源数据集与参数数据集的并集

union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.

```
scala> val a = sc.parallelize(1 to 3, 1)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[195] at parallelize at <console>:24
scala> val b = sc.parallelize(5 to 7, 1)
b: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[196] at parallelize at <console>:24
scala> a.union(b).collect
res121: Array[Int] = Array(1, 2, 3, 5, 6, 7)
scala> (a++b).collect
res122: Array[Int] = Array(1, 2, 3, 5, 6, 7)
```

intersection:返回源数据集与参数数据集的交集

intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.

```
scala> val a = sc.parallelize(1 to 5, 1)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[213] at parallelize at <console>:24
scala> val b = sc.parallelize(5 to 7, 1)
b: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[214] at parallelize at <console>:24
scala>  a.intersection(b).collect
res125: Array[Int] = Array(5)
```

常用算子-transformation

distinct:返回源数据集做元素去重后的数据集

distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.

```
scala> val a = sc.parallelize(1 to 5, 1)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[232] at parallelize at <console>:24
scala> val b = sc.parallelize(5 to 7, 1)
b: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[233] at parallelize at <console>:24
scala> a.union(b).distinct.collect
res129: Array[Int] = Array(4, 6, 2, 1, 3, 7, 5)
```

groupByKey: 只对包含键值对的RDD有效，输入(K,V)对，返回一个包含(K,Iterable<V>)对

groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.

```
scala> sc.parallelize(List((1,2),(3,4),(3,6))).groupByKey().collect
res132: Array[(Int, Iterable[Int])] = Array((1,CompactBuffer(2)), (3,CompactBuffer(4, 6)))
```

常用算子-transformation

reduceByKey:输入(K,V)对，返回(K,V)对，其中key对应的value值是通过func聚合后的结果

reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.

```
scala> sc.parallelize(List((1,2),(3,4),(3,6))).reduceByKey(_+_).collect
res134: Array[(Int, Int)] = Array((1,2), (3,10))
```

aggregateByKey:输入(K,V)对，返回(K,V)对，其中key对应value是通过combOp函数与zeroValue聚合得到

aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.

```
scala> val rdd = sc.parallelize(List(("a",1),("a",2),("a",3),("b",4),("b",5),("c",6)),3)
rdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[292] at parallelize at <console>:24
scala> rdd.glom.collect
res163: Array[Array[(String, Int)]] = Array(Array((a,1), (a,2)), Array((a,3), (b,4)), Array((b,5), (c,6)))
scala> val res = rdd.aggregateByKey(0)(math.max(_,_),_+_)
res: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[294] at aggregateByKey at <console>:25
scala> res.collect
res164: Array[(String, Int)] = Array((c,6), (a,5), (b,9))
```

常用算子-transformation

sortByKey:输入(K,V),输出(K,V)，其中K是经过排序后的结果

sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.

```
scala> sc.parallelize(List((1,2),(3,4),(1,6))).sortByKey(true).collect
res140: Array[(Int, Int)] = Array((1,2), (1,6), (3,4))
```

join:内关联，原理同SQL中的inner join，类似好友leftOuterJoin,rightOuterJoin,fullOuterJoin

join(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.

```
scala> val a = sc.parallelize(List((1,'a'),(2,'b'),(3,'c')))
a: org.apache.spark.rdd.RDD[(Int, Char)] = ParallelCollectionRDD[261] at parallelize at <console>:24
scala> val b = sc.parallelize(List((1,'a'),(2,'b'),(3,'c'),(4,'d')))
b: org.apache.spark.rdd.RDD[(Int, Char)] = ParallelCollectionRDD[262] at parallelize at <console>:24
scala> a.join(b).collect
res145: Array[(Int, (Char, Char))] = Array((1,(a,a)), (2,(b,b)), (3,(c,c)))
```

常用算子-transformation

cogroup:输入(K,V)和(K,W)，返回(K,(Iterable<V>,Iterable<W>))，也被称之为groupWith

cogroup(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called `groupWith`.

```
scala> val a = sc.parallelize(List((1,'a'),(2,'b'),(3,'c')))
a: org.apache.spark.rdd.RDD[(Int, Char)] = ParallelCollectionRDD[266] at parallelize at <console>:24
scala> val b = sc.parallelize(List((1,'a'),(2,'b'),(3,'c'),(4,'d')))
b: org.apache.spark.rdd.RDD[(Int, Char)] = ParallelCollectionRDD[267] at parallelize at <console>:24
scala> a.cogroup(b).collect
res146: Array[(Int, (Iterable[Char], Iterable[Char]))] = Array((1,(CompactBuffer(a),CompactBuffer(a))), (2,
(CompactBuffer(b),CompactBuffer(b))), (3,(CompactBuffer(c),CompactBuffer(c))), (4,(CompactBuffer(),CompactBuffer(d))))
```

repartition:将RDD进行重新混洗(reshuffle)并随机分布到新的分区中，使得数据分布更加均衡，需要消耗网络带宽

repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

```
scala> sc.parallelize(List((1,'a'),(2,'b'),(3,'c')),2).repartition(3)
res148: org.apache.spark.rdd.RDD[(Int, Char)] = MapPartitionsRDD[275] at repartition at <console>:25
```

常用算子-transformation

repartitionAndSortWithinPartitions:根据partitioner重新分区，并在每个分区上按照key进行排序

repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling `repartition` and then sorting within each partition because it can push the sorting down into the shuffle machinery.

```
scala> val data=sc.parallelize(Array(2,4,6,67,3,45,26,35,789,345),2)
data: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[312] at parallelize at <console>:27
scala> data.glom.collect
res191: Array[Array[Int]] = Array(Array(2, 4, 6, 67, 3), Array(45, 26, 35, 789, 345))
scala> data.zipWithIndex().repartitionAndSortWithinPartitions(new org.apache.spark.HashPartitioner(1)).foreach(println)
(2,0)
(3,4)
(4,1)
(6,2)
(26,6)
(35,7)
(45,5)
(67,3)
(345,9)
(789,8)
```

常用算子-action

reduce():将RDD中的元素按照func进行聚合

reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.

```
scala> sc.parallelize(List(1,2,3,3)).reduce((x,y)=>x+y)
res150: Int = 9
```

collect()：将数据集中的所有元素返回给驱动程序

collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

```
scala> sc.parallelize(List(1,2,3,3)).collect
res151: Array[Int] = Array(1, 2, 3, 3)
```

count()：返回数据集中的元素个数

count()	Return the number of elements in the dataset.

```
scala> sc.parallelize(List(1,2,3,3)).count
res152: Long = 4
```

常用算子-action

first:返回数据集中的首个元素

first()	Return the first element of the dataset (similar to take(1)).

```
scala> sc.parallelize(List(1,2,3,3)).first
res153: Int = 1
```

take:返回数据集中的前N个元素

take(*n*)	Return an array with the first *n* elements of the dataset.

```
scala> sc.parallelize(List(1,2,3,3)).take(2)
res154: Array[Int] = Array(1, 2)
```

takeSample:返回数据集中的随机采样子集，最多包含num个元素，其中withReplacement表示是否使用回置采样，seed为随机生成器的种子

takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.

```
scala> val a = sc.parallelize(1 to 10000, 3)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[320] at parallelize at <console>:25
scala> a.takeSample(false,10,0)
res198: Array[Int] = Array(6077, 6963, 331, 8232, 6777, 1684, 2760, 6992, 7826, 6975)
```

常用算子-action

takeOrdered:按元素进行排序，并返回前N个元素，其中ordering可以自定义排序规则

takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.

```
scala> sc.parallelize(List(1,2,3,3)).takeOrdered(2)
res155: Array[Int] = Array(1, 2)
```

saveAsTextFile:将数据集中的元素保存到文本文件中

saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.

```
scala> sc.parallelize(1 to 10, 3).saveAsTextFile("/Users/xxx/Downloads/data/20200404")

data xxx$ hadoop fs -ls /Users/xxx/Downloads/data/20200404
-rw-r--r--   1 xxx supergroup    0 2020-04-06 18:38 /Users/xxx/Downloads/data/20200404/_SUCCESS
-rw-r--r--   1 xxx supergroup    6 2020-04-06 18:38 /Users/xxx/Downloads/data/20200404/part-00000
-rw-r--r--   1 xxx supergroup    6 2020-04-06 18:38 /Users/xxx/Downloads/data/20200404/part-00001
-rw-r--r--   1 xxx supergroup    9 2020-04-06 18:38 /Users/xxx/Downloads/data/20200404/part-00002
```

常用算子-action

savaAsObjectFile：将RDD元素以Java序列化的格式保存为文件

saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.

```
scala> sc.makeRDD(1 to 10, 3).saveAsObjectFile("/Users/xxx/Downloads/data/20200403")
```

```
data xxx$ hadoop fs -ls /Users/xxx/Downloads/data/20200403
20/04/06 19:08:28 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using
builtin-java classes where applicable
Found 4 items
-rw-r--r--   1 xxx supergroup          0 2020-04-06 19:08 /Users/xxx/Downloads/data/20200403/_SUCCESS
-rw-r--r--   1 xxx supergroup        146 2020-04-06 19:08 /Users/xxx/Downloads/data/20200403/part-00000
-rw-r--r--   1 xxx supergroup        146 2020-04-06 19:08 /Users/xxx/Downloads/data/20200403/part-00001
-rw-r--r--   1 xxx supergroup        150 2020-04-06 19:08 /Users/xxx/Downloads/data/20200403/part-00002
```

savaAsSequenceFile：将数据集中的元素保存到指定目录下的Hadoop Sequence文件中

saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).

用法同上

常用算子-action

countByKey: 只适用于(K,V)的RDD，返回一个hashmap，包含(K,int)对，表示每个key的个数

countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.

```
scala> sc.parallelize(List(("a",1),("a",2),("a",3),("b",4),("b",5),("c",6)),3).countByKey().foreach(println)
(c,1)
(a,3)
(b,2)
```

foreach:在RDD的每个元素上运行func函数，通常被用于累加操作

foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the `foreach()` may result in undefined behavior. See Understanding closures for more details.

```
scala> sc.parallelize(List(("a",1),("a",2),("a",3),("b",4),("b",5),("c",6)),3).countByKey().foreach(println)
(c,1)
(a,3)
(b,2)
```

RDD缓存(持久化)

主要作用：复用已有数据集

主要方法：使用persist()或者cache()

存储级别：

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

RDD缓存(持久化)-如何选择存储级别

1、如果RDD能使用默认存储级别(MEMORY_ONLY)，尽量使用默认级别，这是CPU效率最高的方式

2、如果第一步为否，可以尝试MEMORY_ONLY_SER级别，并选择一个高效的序列化协议，这将大大节省数据对象的存储空间，同时速度还不错

3、尽量不要把数据回吐到磁盘上，除非：(1)你的数据集重新计算的代价很大 (2)你的数据集是从一个很大的数据源中过滤得到的结果。否则，重算一个分区的速度很可能比从磁盘上读取差不多

4、如果需要支持容错，可以考虑使用带副本的存储级别。所有的存储级别都能够以重算丢失数据的方式来提供容错性，但是带副本的存储级别可以让你的应用持续的运行，而不必等待重算丢失的分区

5、在一些需要大量内存或者并行多个应用的场景下，实验性的OFF_HEAP会有如下优势

（1）这个级别下，可以允许多个执行器共享同一个Tachyon中的内存池

（2）可以有效的减少垃圾回收的开销

（3）即使单个执行器挂了，缓存数据也不会丢失

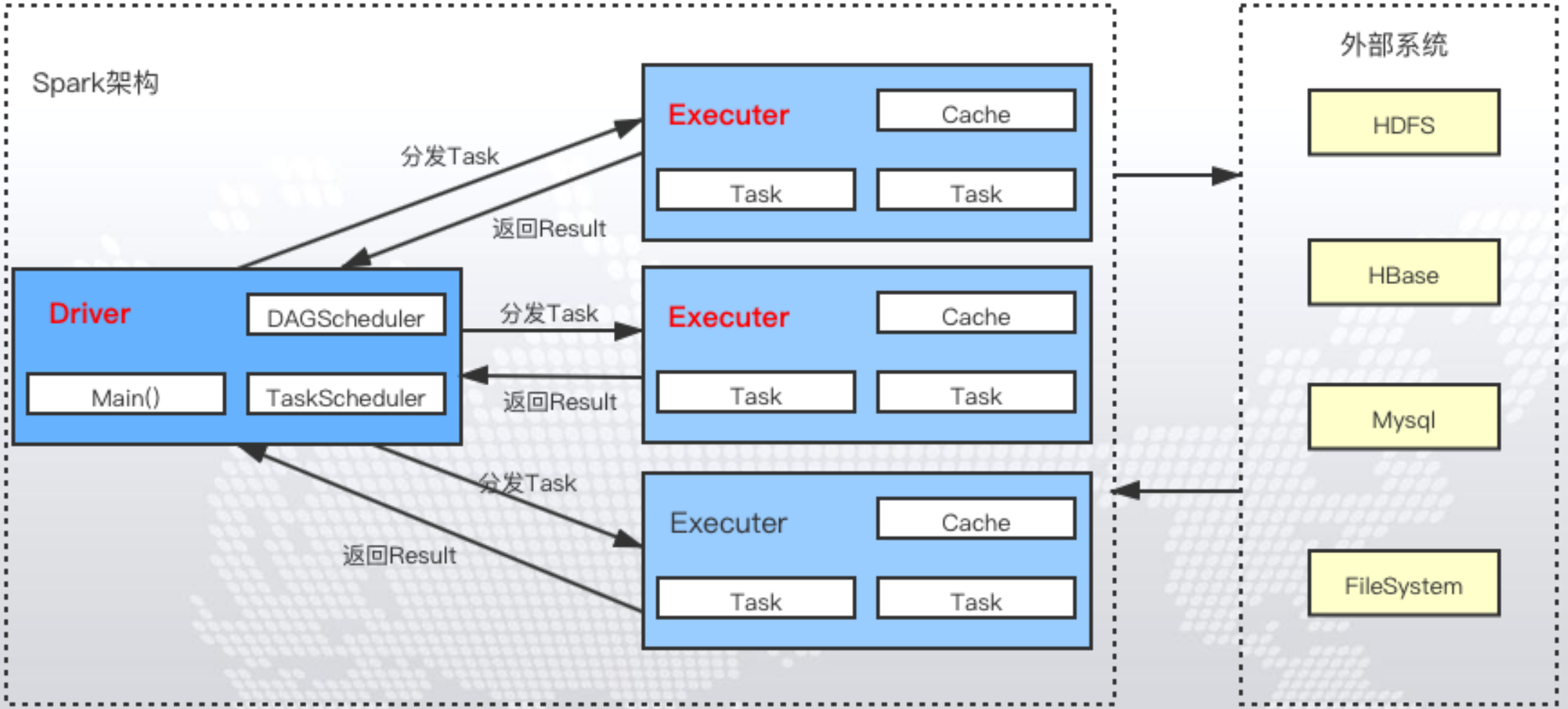
闭包

```
scala> var counter = 0
counter: Int = 0
scala> var rdd = sc.parallelize(List(1,2,3,4,5,6,7,8,9))
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[346] at parallelize at <console>:25
scala> rdd.foreach(x=>counter +=x)
scala> println(counter)
0
```

闭包是由执行器执行RDD算子（本例中的foreach()）时所需要的变量和方法组成的。闭包将会被序列化，并发送给每个执行器。由于本地模式下，只有一个执行器，所有任务都共享同样的闭包。而在其他模式下，情况则有所不同，每个执行器都运行于不同的worker节点，并且都拥有独立的闭包副本。

在上面的例子中，闭包中的变量会跟随不同的闭包副本，发送到不同的执行器上，所以等到foreach真正在执行器上运行时，其引用的counter已经不再是驱动器上所定义的那个counter副本了，驱动器内存中仍然会有一个counter变量副本，但是这个副本对执行器是不可见的！执行器只能看到其所收到的序列化闭包中包含的counter副本。因此，最终驱动器上得到的counter将会是0。

Spark 架构


Spark架构
Driver
DAGScheduler
Main()
TaskScheduler
分发Task
返回Result
分发Task
返回Result
分发Task
返回Result
Executer
Cache
Task
Task
Executer
Cache
Task
Task
Executer
Cache
Task
Task
外部系统
HDFS
HBase
Mysql
FileSystem

Spark 架构

Client：客户端进程，负责提交作业。

Driver：一个Spark作业有一个Spark Context，一个Spark Context对应一个Driver进程，作业的main函数运行在Driver中。Driver主要负责Spark作业的解析，以及通过DAGScheduler划分Stage，将Stage转化成TaskSet提交给TaskScheduler任务调度器，进而调度Task到Executor上执行。

Executor：负责执行Driver分发的Task任务。集群中一个节点可以启动多个Executor，每一个Executor可以执行多个Task任务。

Catche：Spark提供了对RDD不同级别的缓存策略，分别可以缓存到内存、磁盘、外部分布式内存存储系统Tachyon等。

Application：提交的一个作业就是一个Application，一个Application只有一个Spark Context。

Job：RDD执行一次Action操作就会生成一个Job。

Task：Spark运行的基本单位，负责处理RDD的计算逻辑。

Stage：DAGScheduler将Job划分为多个Stage，Stage的划分界限为Shuffle的产生，Shuffle标志着上一个Stage的结束和下一个Stage的开始。

TaskSet：划分的Stage会转换成一组相关联的任务集。

DAG（Directed Acyclic Graph）：有向无环图。Spark实现了DAG的计算模型，DAG计算模型是指将一个计算任务按照计算规则分解为若干子任务，这些子任务之间根据逻辑关系构建成有向无环图。

共享变量

一般而言，当我们给Spark算子（如 map 或 reduce）传递一个函数时，这些函数将会在**远程的集群节点**上运行，并且这些函数所引用的变量都是各个节点上的**独立副本**。这些变量都会以**副本**的形式复制到各个机器节点上，如果更新这些变量副本的话，这些更新并不会传回到**驱动器（driver）**程序。通常来说，支持跨任务的可读写共享变量是比较低效的。不过，Spark还是提供了两种比较通用的共享变量：**广播变量**和**累加器**。

广播变量

广播变量提供了一种只读的共享变量，它是把在每个机器节点上保存一个缓存，而不是每个任务保存一份副本。通常可以用来在每个节点上保存一个较大的输入数据集，这要比常规的变量副本更高效（一般的变量是每个任务一个副本，一个节点上可能有多个任务）。Spark还会尝试使用高效的广播算法来分发广播变量，以减少通信开销。

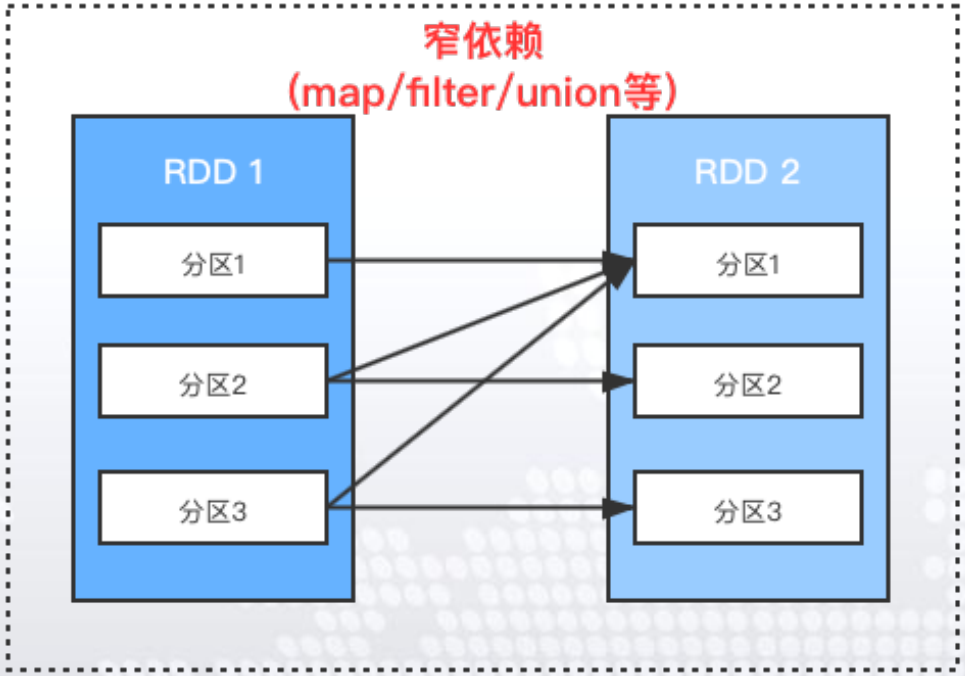
```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(224)
scala> broadcastVar.value
res224: Array[Int] = Array(1, 2, 3)
```

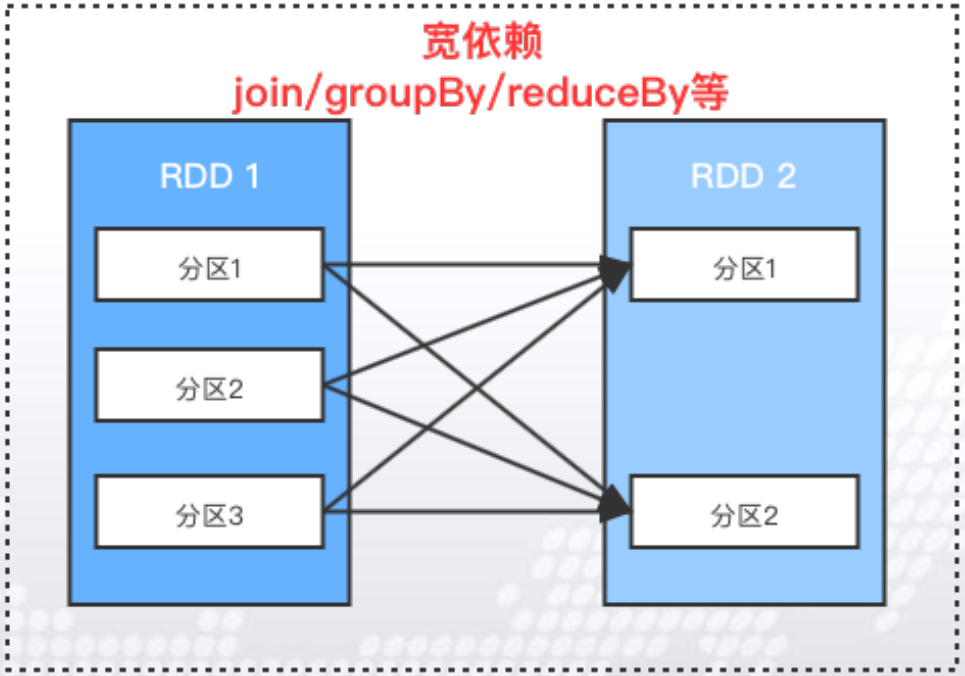
累加器

累加器是一种只支持满足结合律的“累加”操作的变量，因此它可以很高效地支持并行计算。利用累加器可以实现计数（类似MapReduce中的计数器）或者求和。Spark原生支持了数字类型的累加器，开发者也可以自定义新的累加器。如果创建累加器的时候给了一个名字，那么这个名字会展示在Spark UI上，这对于了解程序运行处于哪个阶段非常有帮助（注意：Python尚不支持该功能）

```
scala> var counter = sc.accumulator(0,"My accumulator")
counter: org.apache.spark.Accumulator[Int] = 0
scala> var rdd = sc.parallelize(List(1,2,3,4,5,6,7,8,9))
scala> rdd.foreach(x=>counter +=x)
scala> println(counter)
45
```

依赖：宽依赖与窄依赖





判断依据

窄依赖：父RDD的分区可以一一对应到子RDD的分区

宽依赖：父RDD的分区可以被多个子RDD的分区使用

疑问?

窄依赖的子RDD只能一一对应一个父RDD，对吗?

口诀：

窄依赖：一子多父、一子一父

宽依赖：一父多子

终极判断方法：

如果依赖连线是发散的，就是宽依赖

否则就是窄依赖

Shuffle与优化

背景

在Spark中，通常是由于为了进行某种计算操作，而将数据分布到所需要的各个分区当中。而在计算阶段，单个任务（task）只会操作单个分区中的数据 - 因此，为了组织好每个任务执行时所需的数据，Spark需要执行一个多对多操作。即，Spark需要读取RDD的所有分区，并找到所有key对应的所有values，然后跨分区传输这些values，并将每个key对应的所有values放到同一分区，以便后续计算各个key对应values的reduce结果 - 这个过程就叫做**混洗（Shuffle）**。

会导致Shuffle的算子

重分区（repartition）类算子，如： repartition 和 coalesce；ByKey 类算子(除了计数类的，如 countByKey) 如：groupByKey 和 reduceByKey；以及Join类算子，如：cogroup 和 join.

Shuffle慢的原因

混洗（Shuffle）之所以开销大，是因为混洗操作需要引入磁盘I/O，数据序列化以及网络I/O等操作。

Shuffle与优化

1、避免重复创建RDD
2、尽可能复用同一个RDD
3、对多次使用的RDD进行持久化
4、尽量避免使用shuffle类算子
5、尽量使用高性能算子
   groupByKey -》reduceByKey/aggregateByKey
   map -》mapPartitions
   foreach -》foreachPartitions
   repartition+sort -》repartitionAndSortWithinPartitions
6、使用广播变量

推荐课外阅读内容：
1、美团Spark优化基础篇 https://tech.meituan.com/2016/04/29/spark-tuning-basic.html
2、美团Spark优化高级篇https://tech.meituan.com/2016/05/12/spark-tuning-pro.html
3、书籍：企业大数据处理：Spark、Druid、Flume与Kafka应用实践

THANK YOU FOR YOUR GUIDANCE.

谢谢