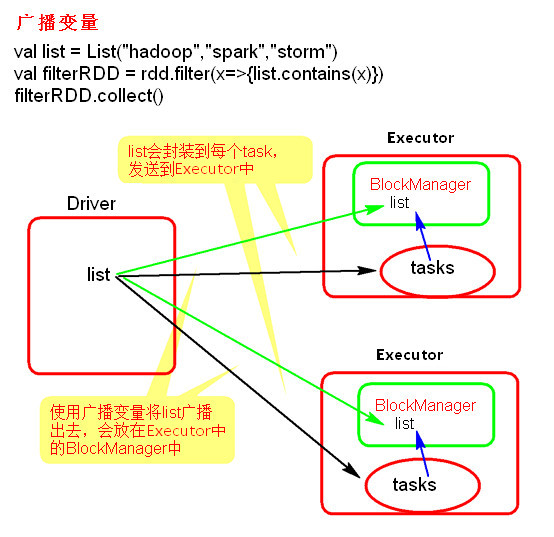
# Spark day04

1. **广播变量和累加器**
2. 广播变量

* 广播变量理解图



* 广播变量使用

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("brocast")  **val** sc = **new** SparkContext(conf)  **val** list = List("hello xasxt")  **val broadCast = sc.broadcast(list)**  **val** lineRDD = sc.textFile("./words.txt")  lineRDD.filter { x => **broadCast.value**.contains(x) }.foreach { println}  sc.stop() |

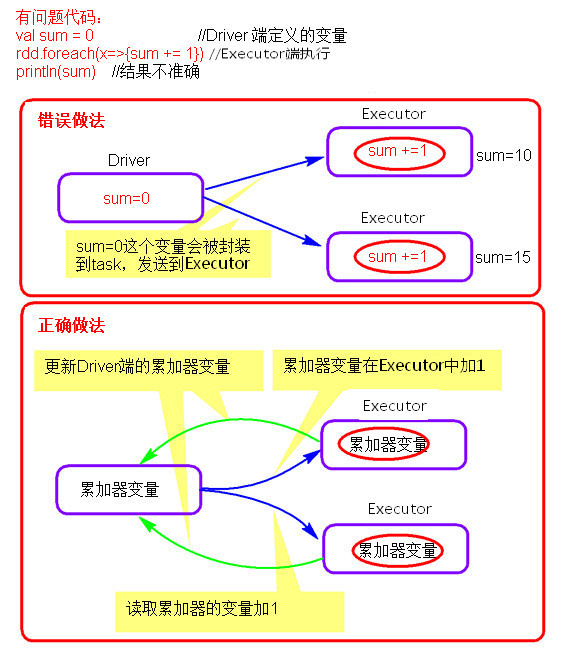
* 注意事项
* 能不能将一个RDD使用广播变量广播出去？

不能，因为RDD是不存储数据的。可以将RDD的结果广播出去。

* 广播变量只能在Driver端定义，不能在Executor端定义。
* 在Driver端可以修改广播变量的值，在Executor端无法修改广播变量的值。

1. 累加器

* 累加器理解图



* 累加器的使用

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("accumulator")  **val** sc = **new** SparkContext(conf)  **val** accumulator = sc.accumulator(0)  sc.textFile("./words.txt").foreach { x =>{accumulator.add(1)}}  println(accumulator.value)  sc.stop() |

* 注意事项
* 累加器在Driver端定义赋初始值，累加器只能在Driver端读取，在Excutor端更新。

1. **SparkUI**
2. SparkUI界面介绍

可以指定提交Application的名称

|  |
| --- |
| ./spark-shell --master spark://node1:7077 --name myapp |

1. 配置historyServer

* 临时配置，对本次提交的应用程序起作用

|  |
| --- |
| ./spark-shell --master spark://node1:7077  --name myapp1  **--conf spark.eventLog.enabled=true**  **--conf spark.eventLog.dir=hdfs://node1:9000/spark/test** |

停止程序，在Web Ui中Completed Applications对应的ApplicationID中能查看history。

* spark-default.conf配置文件中配置HistoryServer，对所有提交的Application都起作用

在客户端节点，进入../spark-1.6.0/conf/ spark-defaults.conf最后加入:

|  |
| --- |
| **//开启记录事件日志的功能**  **spark.eventLog.enabled true**  **//设置事件日志存储的目录**  **spark.eventLog.dir hdfs://node1:9000/spark/test**  **//设置HistoryServer加载事件日志的位置**  **spark.history.fs.logDirectory hdfs://node1:9000/spark/test**  **//日志优化选项,压缩日志**  **spark.eventLog.compress true** |

启动HistoryServer：

|  |
| --- |
| ./start-history-server.sh |

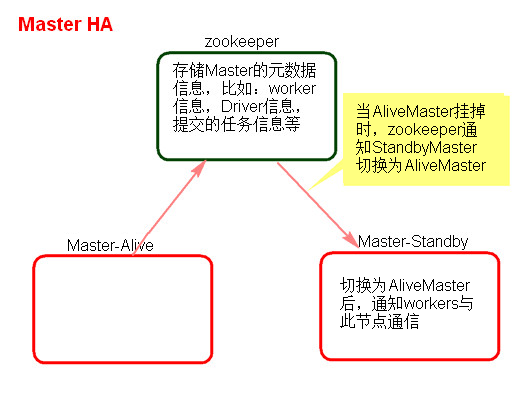
访问HistoryServer：node4:18080,之后所有提交的应用程序运行状况都会被记录。

1. **Master HA**
2. Master的高可用原理

Standalone集群只有一个Master，如果Master挂了就无法提交应用程序，需要给Master进行高可用配置，Master的高可用可以使用fileSystem(文件系统)和zookeeper（分布式协调服务）。

fileSystem只有存储功能，可以存储Master的元数据信息，用fileSystem搭建的Master高可用，在Master失败时，需要我们手动启动另外的备用Master，这种方式不推荐使用。

zookeeper有选举和存储功能，可以存储Master的元素据信息，使用zookeeper搭建的Master高可用，当Master挂掉时，备用的Master会自动切换，推荐使用这种方式搭建Master的HA。



1. Master高可用搭建
2. 在Spark Master节点上配置主Master，配置spark-env.sh

|  |
| --- |
| **export SPARK\_DAEMON\_JAVA\_OPTS="**  **-Dspark.deploy.recoveryMode=ZOOKEEPER**  **-Dspark.deploy.zookeeper.url=node3:2181,node4:2181,node5:2181**  **-Dspark.deploy.zookeeper.dir=/sparkmaster0821"** |



1. 发送到其他worker节点上





1. 找一台节点（非主Master节点）配置备用 Master,修改spark-env.sh配置节点上的MasterIP



1. 启动集群之前启动zookeeper集群：

|  |
| --- |
| ../zkServer.sh start |

1. 启动spark Standalone集群，启动备用Master
2. 打开主Master和备用Master WebUI页面，观察状态。
3. 注意点

* 主备切换过程中不能提交Application。
* 主备切换过程中不影响已经在集群中运行的Application。因为Spark是粗粒度资源调度。

1. 测试验证

提交SparkPi程序，kill主Master观察现象。

|  |
| --- |
| ./spark-submit  **--master spark://node1:7077,node2:7077**  --class org.apache.spark.examples.SparkPi  ../lib/spark-examples-1.6.0-hadoop2.6.0.jar  10000 |

1. **SparkShuffle**
2. SparkShuffle概念

reduceByKey会将上一个RDD中的每一个key对应的所有value聚合成一个value，然后生成一个新的RDD，元素类型是<key,value>对的形式，这样每一个key对应一个聚合起来的value。

**问题：**聚合之前，每一个key对应的value不一定都是在一个partition中，也不太可能在同一个节点上，因为RDD是分布式的弹性的数据集，RDD的partition极有可能分布在各个节点上。

**如何聚合？**

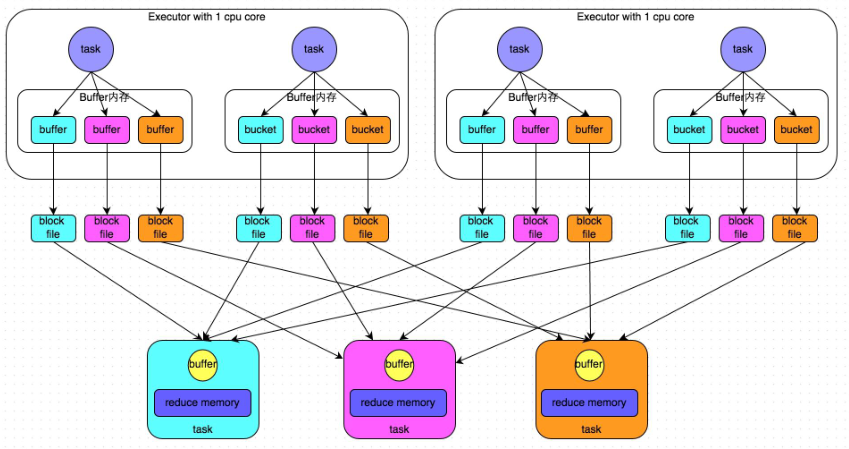
**– Shuffle Write：**上一个stage的每个map task就必须保证将自己处理的当前分区的数据相同的key写入一个分区文件中，可能会写入多个不同的分区文件中。

**– Shuffle Read：**reduce task就会从上一个stage的所有task所在的机器上寻找属于己的那些分区文件，这样就可以保证每一个key所对应的value都会汇聚到同一个节点上去处理和聚合。

Spark中有两种Shuffle管理类型，HashShufflManager和SortShuffleManager，Spark1.2之前是HashShuffleManager， Spark1.2引入SortShuffleManager,在Spark 2.0+版本中已经将HashShuffleManager丢弃。

1. HashShuffleManager
2. 普通机制

* 普通机制示意图



* 执行流程

1. 每一个map task将不同结果写到不同的buffer中，每个buffer的大小为32K。buffer起到数据缓存的作用。
2. 每个buffer文件最后对应一个磁盘小文件。
3. reduce task来拉取对应的磁盘小文件。

* 总结

1. .map task的计算结果会根据分区器（默认是hashPartitioner）来决定写入到哪一个磁盘小文件中去。ReduceTask会去Map端拉取相应的磁盘小文件。
2. .产生的磁盘小文件的个数：

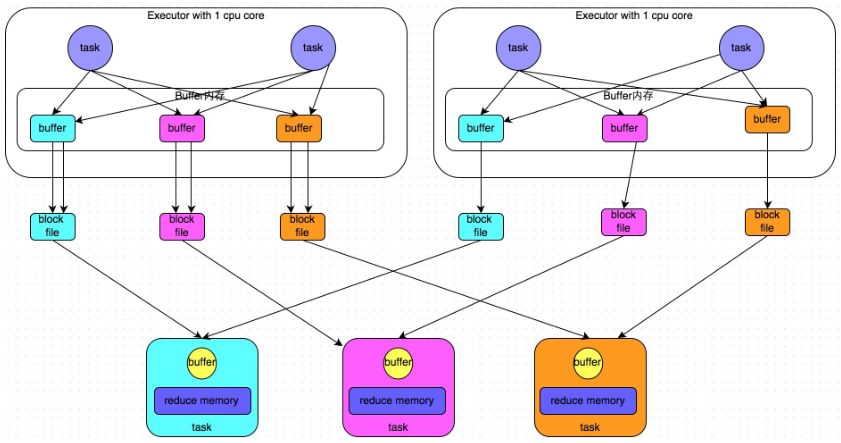
M（map task的个数）\*R（reduce task的个数）

* 存在的问题

产生的磁盘小文件过多，会导致以下问题：

1. 在Shuffle Write过程中会产生很多写磁盘小文件的对象。
2. 在Shuffle Read过程中会产生很多读取磁盘小文件的对象。
3. 在JVM堆内存中对象过多会造成频繁的gc,gc还无法解决运行所需要的内存 的话，就会OOM。
4. 在数据传输过程中会有频繁的网络通信，频繁的网络通信出现通信故障的可能性大大增加，一旦网络通信出现了故障会导致shuffle file cannot find 由于这个错误导致的task失败，TaskScheduler不负责重试，由DAGScheduler负责重试Stage。
5. 合并机制

* 合并机制示意图

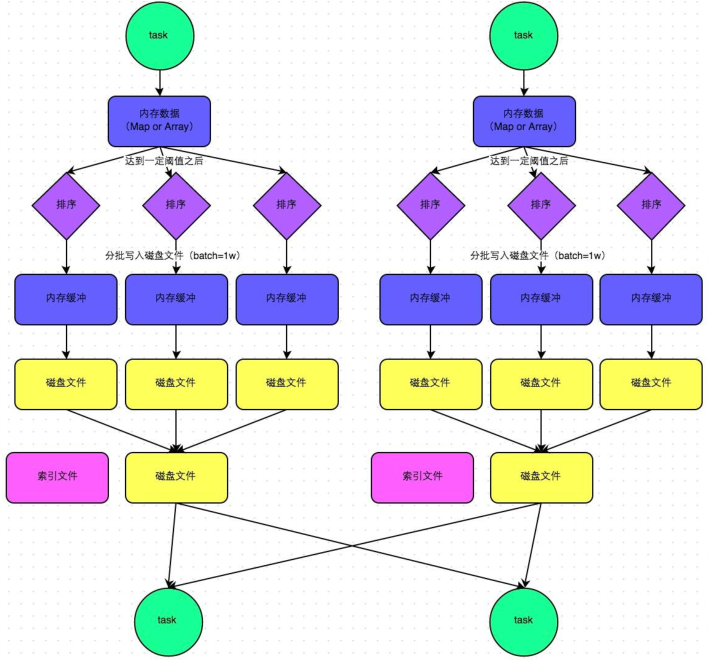


* 总结

产生磁盘小文件的个数：C(core的个数)\*R（reduce的个数）

1. SortShuffleManager
2. 普通机制

* 普通机制示意图



* 执行流程

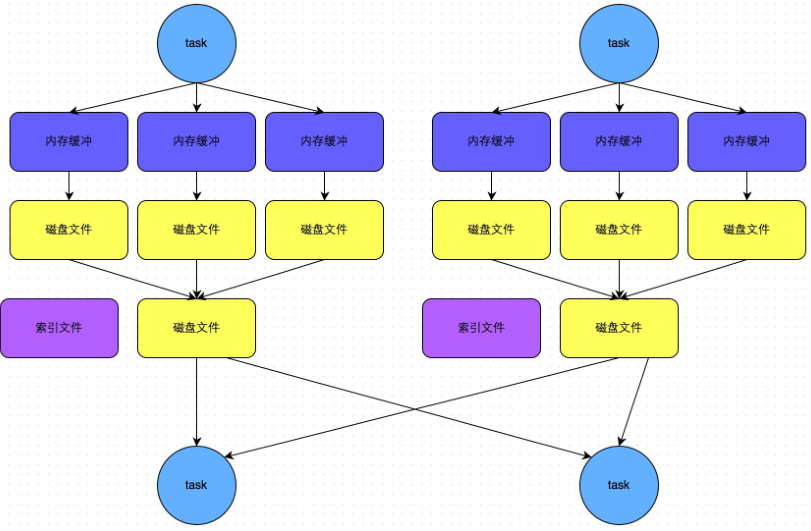
1. map task 的计算结果会写入到一个内存数据结构里面，内存数据结构默认是5M
2. 在shuffle的时候会有一个定时器，不定期的去估算这个内存结构的大小，当内存结构中的数据超过5M时，比如现在内存结构中的数据为5.01M，那么他会申请5.01\*2-5=5.02M内存给内存数据结构。
3. 如果申请成功不会进行溢写，如果申请不成功，这时候会发生溢写磁盘。
4. 在溢写之前内存结构中的数据会进行排序分区
5. 然后开始溢写磁盘，写磁盘是以batch的形式去写，一个batch是1万条数据，
6. map task执行完成后，会将这些磁盘小文件合并成一个大的磁盘文件，同时生成一个索引文件。
7. reduce task去map端拉取数据的时候，首先解析索引文件，根据索引文件再去拉取对应的数据。

* 总结

产生磁盘小文件的个数： 2\*M（map task的个数）

1. bypass机制

* bypass机制示意图



* 总结

1. .bypass运行机制的触发条件如下：

shuffle reduce task的数量小于spark.shuffle.sort.bypassMergeThreshold的参数值。这个值默认是200。

1. .产生的磁盘小文件为：2\*M（map task的个数）
2. Shuffle文件寻址
3. MapOutputTracker

MapOutputTracker是Spark架构中的一个模块，是一个主从架构。管理磁盘小文件的地址。

* MapOutputTrackerMaster是主对象，存在于Driver中。
* MapOutputTrackerWorker是从对象，存在于Excutor中。

1. BlockManager

BlockManager块管理者，是Spark架构中的一个模块，也是一个主从架构。

* BlockManagerMaster,主对象，存在于Driver中。

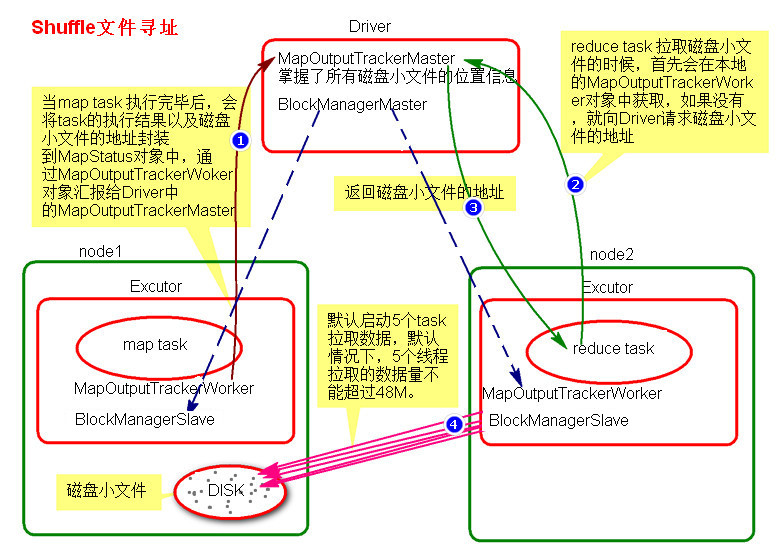
BlockManagerMaster会在集群中有用到广播变量和缓存数据或者删除缓存数据的时候，通知BlockManagerSlave传输或者删除数据。

* BlockManagerSlave，从对象，存在于Excutor中。

BlockManagerSlave会与BlockManagerSlave之间通信。

* 无论在Driver端的BlockManager还是在Excutor端的BlockManager都含有三个对象：

1. DiskStore:负责磁盘的管理。
2. MemoryStore：负责内存的管理。
3. BlockTransferService:负责数据的传输。
4. Shuffle文件寻址图



1. Shuffle文件寻址流程
2. 当map task执行完成后，会将task的执行情况和磁盘小文件的地址封装到MpStatus对象中，通过MapOutputTrackerWorker对象向Driver中的MapOutputTrackerMaster汇报。
3. 在所有的map task执行完毕后，Driver中就掌握了所有的磁盘小文件的地址。
4. 在reduce task执行之前，会通过Excutor中MapOutPutTrackerWorker向Driver端的MapOutputTrackerMaster获取磁盘小文件的地址。
5. 获取到磁盘小文件的地址后，会通过BlockManager连接数据所在节点，然后通过BlockTransferService进行数据的传输。
6. BlockTransferService默认启动5个task去节点拉取数据。默认情况下，5个task拉取数据量不能超过48M。
7. **Spark内存管理**

Spark执行应用程序时，Spark集群会启动Driver和Executor两种JVM进程，Driver负责创建SparkContext上下文，提交任务，task的分发等。Executor负责task的计算任务，并将结果返回给Driver。同时需要为需要持久化的RDD提供储存。Driver端的内存管理比较简单，这里所说的Spark内存管理针对Executor端的内存管理。

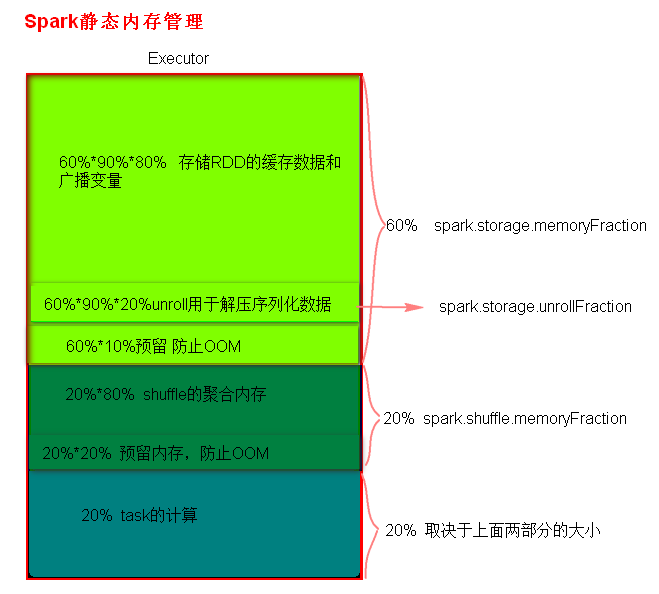
Spark内存管理分为静态内存管理和统一内存管理，Spark1.6之前使用的是静态内存管理，Spark1.6之后引入了统一内存管理。

**静态内存管理**中存储内存、执行内存和其他内存的大小在 Spark 应用程序运行期间均为固定的，但用户可以应用程序启动前进行配置。

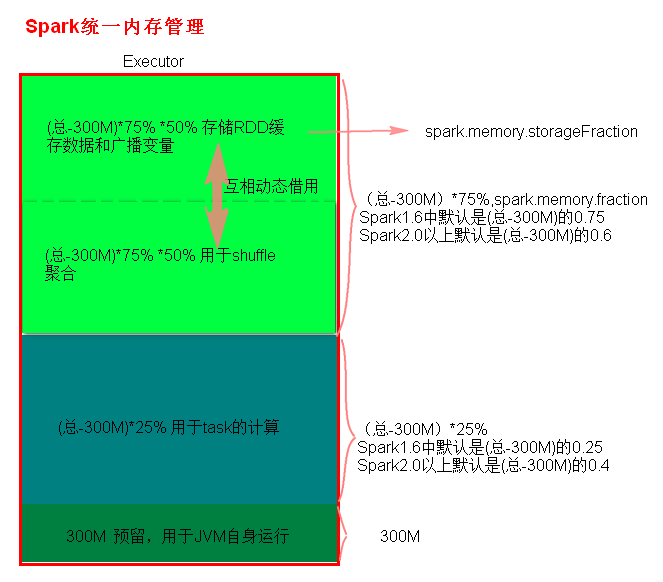
**统一内存管理**与静态内存管理的区别在于储存内存和执行内存共享同一块空间，可以互相借用对方的空间。

Spark1.6以上版本默认使用的是统一内存管理，可以通过参数spark.memory.useLegacyMode 设置为true(默认为false)使用静态内存管理。

1. 静态内存管理分布图

****

1. 统一内存管理分布图

****

1. reduce 中OOM如何处理？
2. 减少每次拉取的数据量
3. 提高shuffle聚合的内存比例
4. 提高Excutor的总内存
5. **Shuffle调优**
6. SparkShuffle调优配置项如何使用？
7. 在代码中,不推荐使用，硬编码。

new SparkConf().set(“spark.shuffle.file.buffer”,”64”)

1. 在提交spark任务的时候，推荐使用。

spark-submit --conf spark.shuffle.file.buffer=64 –conf ….

1. 在conf下的spark-default.conf配置文件中,不推荐，因为是写死后所有应用程序都要用。
2. Shuffle调优附件

|  |
| --- |
|  |