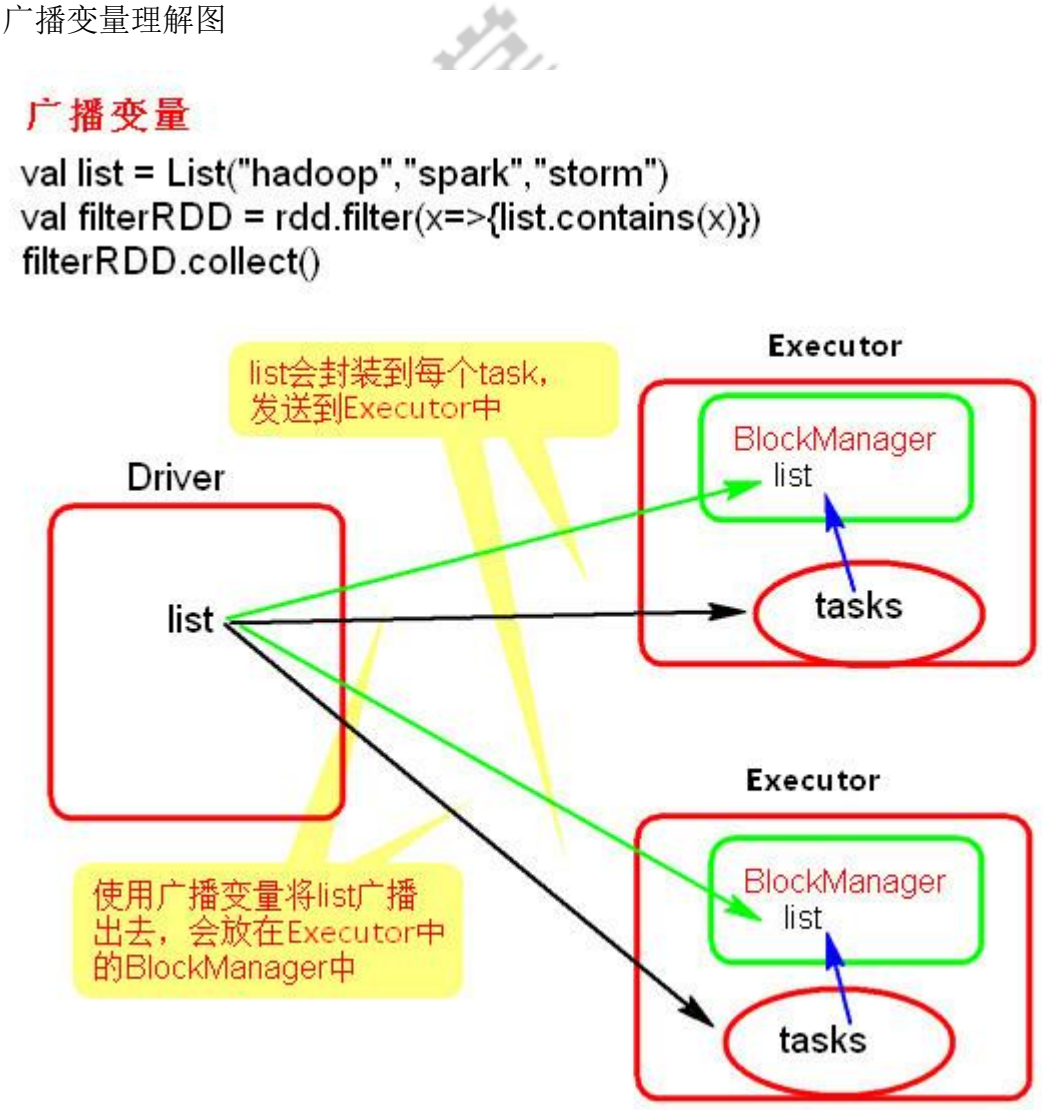
1. 广播变量和累加器

1. 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？

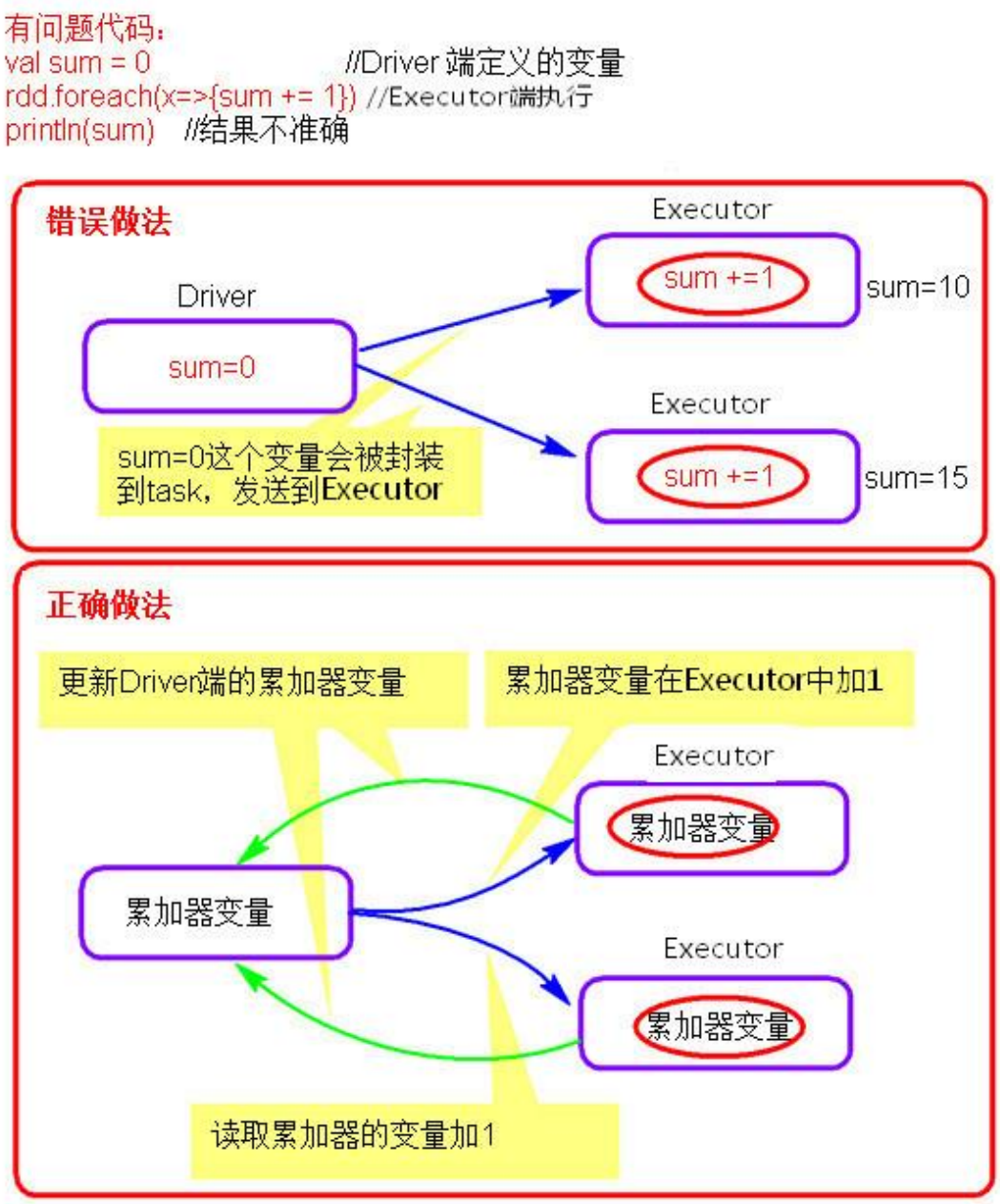
  不能，因为RDD是不存储数据的。可以将RDD的结果广播出去。

- ✶ 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。

2. 累加器

   ➢ 累加器理解图



- 累加器的使用

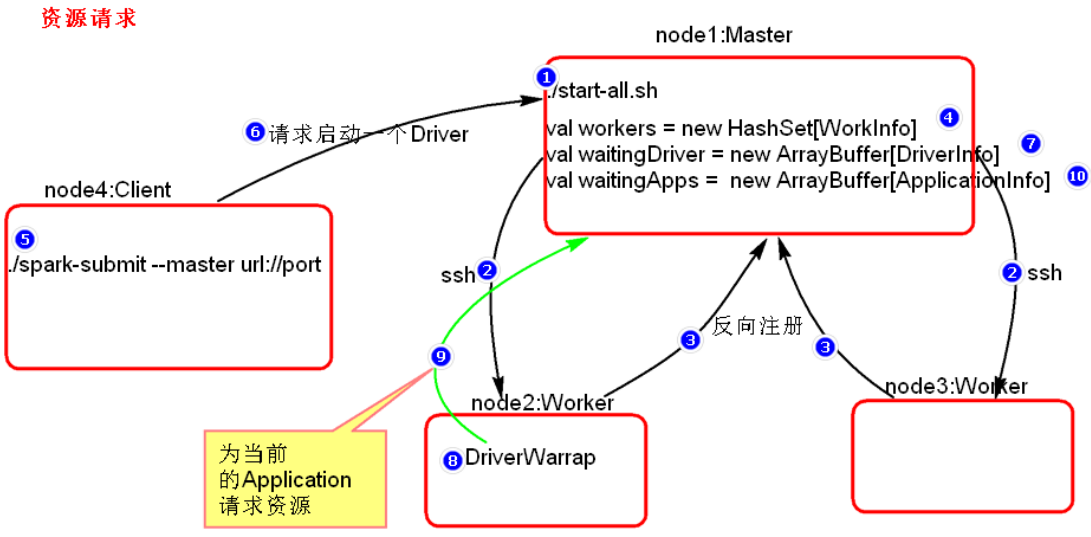
```
val conf = new SparkConf()
conf.setMaster("local").setAppName("accumulator")
val sc = new SparkContext(conf)
val accumulator = sc.accumulator(0)
sc.textFile("./words.txt").foreach { x =>{accumulator.add(1)}}
println(accumulator.value)
sc.stop()
```

- 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。

3. 资源调度源码分析

- 资源请求简单图



- 资源调度 Master 路径：

```
CLASS="org.apache.spark.deploy.master.Master"
```

路径：spark-1.6.0/core/src/main/scala/org.apache.spark/deploy/Master/Master.scala

- 提交应用程序，submit 的路径：

```
org.apache.spark.deploy.SparkSubmit
```

路径：spark-1.6.0/core/src/main/scala/org.apache.spark/ deploy/SparkSubmit.scala

- 总结：

1. Executor 在集群中分散启动，有利于 task 计算的数据本地化。
2. 默认情况下（提交任务的时候没有设置--executor-cores 选项），每一个 Worker 为当前的 Application 启动一个 Executor,这个 Executor 会使用这个 Worker 的所有的 cores 和 1G 内存。
3. 如果想在 Worker 上启动多个 Executor，提交 Application 的时候要加--executor-cores 这个选项。
4. 默认情况下没有设置--total-executor-cores,一个 Application 会使用 Spark 集群中所有的 cores。

➢ 结论演示

使用 Spark-submit 提交任务演示。也可以使用 spark-shell

1. 默认情况每个 worker 为当前的 Application 启动一个 Executor，这个 Executor 使用集群中所有的 cores 和 1G 内存。

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

2. 在 workr 上启动多个 Executor,设置--executor-cores 参数指定每个 executor 使用的 core 数量。

```
./spark-submit
 --master   spark://node1:7077
 --executor-cores 1
 --class org.apache.spark.examples.SparkPi
../lib/spark-examples-1.6.0-hadoop2.6.0.jar
10000
```

3. 内存不足的情况下启动 core 的情况。Spark 启动是不仅看 core 配置参数，也要看配置的 core 的内存是否够用。

```
./spark-submit
--master   spark://node1:7077
--executor-cores 1
--executor-memory 3g
--class org.apache.spark.examples.SparkPi
 ../lib/spark-examples-1.6.0-hadoop2.6.0.jar
```

```
10000
```

4. --total-executor-cores 集群中共使用多少 cores

注意：一个进程不能让集群多个节点共同启动。

```
./spark-submit
--master    spark://node1:7077
--executor-cores 1
--executor-memory 2g
--total-executor-cores 3
--class org.apache.spark.examples.SparkPi
  ../lib/spark-examples-1.6.0-hadoop2.6.0.jar
10000
```

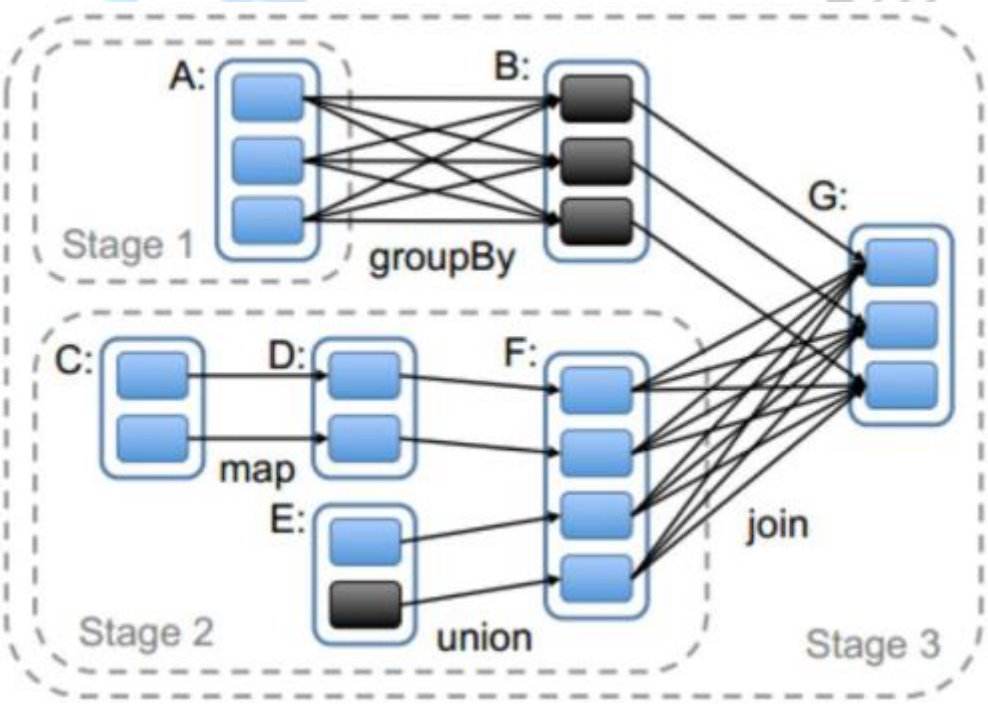
4. 任务调度源码分析

- Action 算子开始分析

任务调度可以从一个 Action 类算子开始。因为 Action 类算子会触发一个 job 的执行。

- 划分 stage,以 taskSet 形式提交任务

DAGScheduler 类中 getMessingParentStages()方法是切割 job 划分 stage 。可以结合以下这张图来分析：



5. SparkShuffle

1. 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？

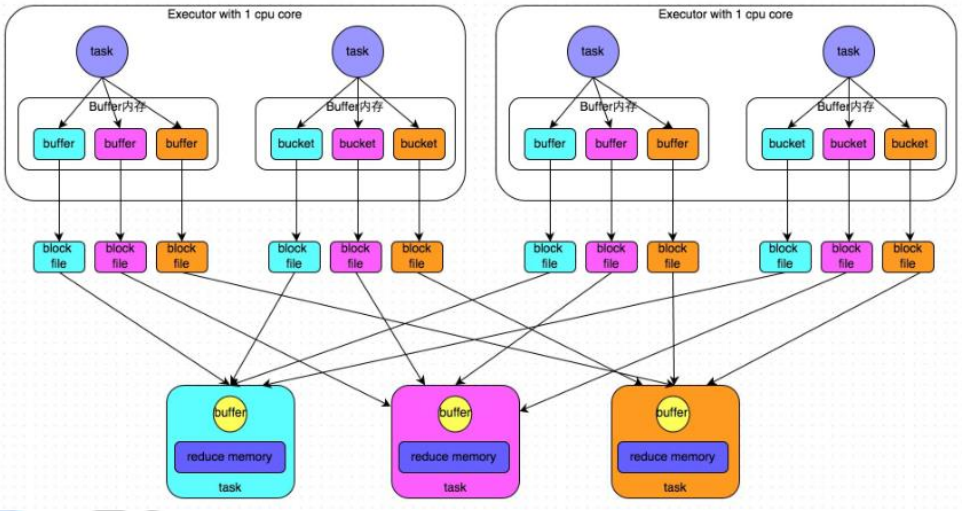
- **Shuffle Write**：上一个 stage 的每个 map task 就必须保证将自己处理的当前分区的数据相同的 key 写入一个分区文件中，可能会写入多个不同的分区文件中。
- **Shuffle Read**：reduce task 就会从上一个 stage 的所有 task 所在的机器上寻找属于己的那些分区文件，这样就可以保证每一个 key 所对应的 value 都会汇聚到同一个节点上去处理和聚合。

Spark 中有两种 Shuffle 类型，HashShuffle 和 SortShuffle，Spark1.2 之前是 HashShuffle 默认的分区器是 HashPartitioner，Spark1.2 引入 SortShuffle 默认的分区器是 RangePartitioner。

2. HashShuffle

1) 普通机制

- 普通机制示意图



- 执行流程

a) 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。

b) 每个 buffer 文件最后对应一个磁盘小文件。

c) 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：

a) 在 Shuffle Write 过程中会产生很多写磁盘小文件的对象。

b) 在 Shuffle Read 过程中会产生很多读取磁盘小文件的对象。

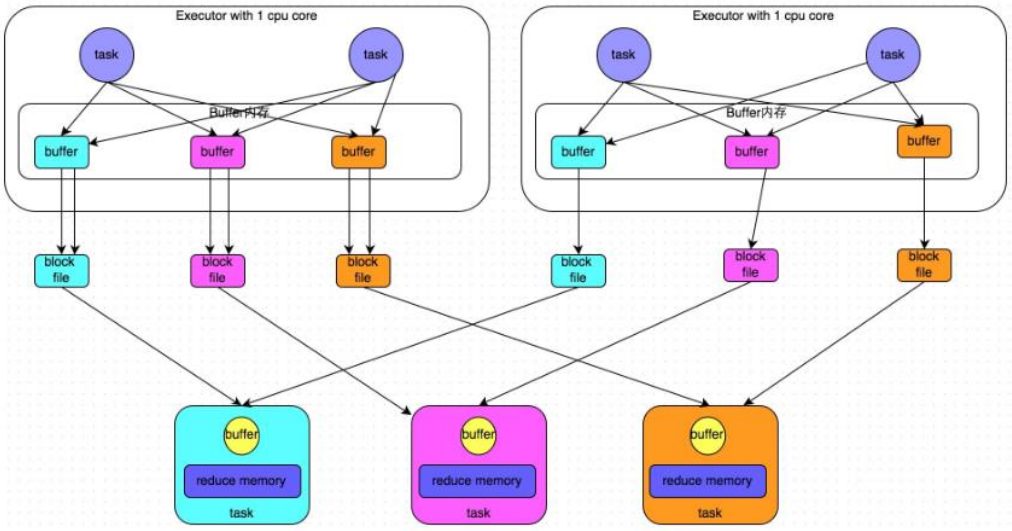
c) 在 JVM 堆内存中对象过多会造成频繁的 gc,gc 还无法解决运行所需要的内存 的话，就会 OOM。

d) 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。

2) 合并机制

➢ 合并机制示意图



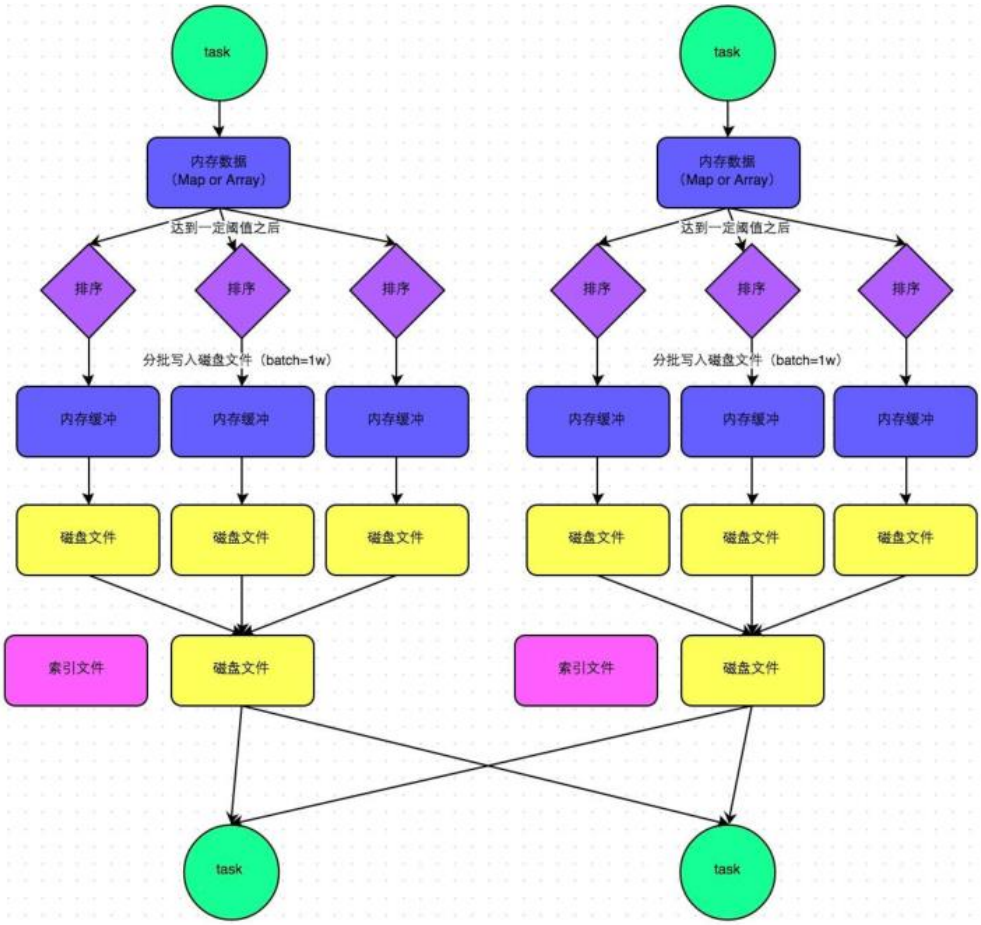
➢ 总结

产生磁盘小文件的个数：C(core 的个数)*R（reduce 的个数）

3. SortShuffle

1) 普通机制

➢ 普通机制示意图



- 执行流程

a) 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

b) 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。

c) 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。

d) 在溢写之前内存结构中的数据会进行排序分区

e) 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，

f) 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。

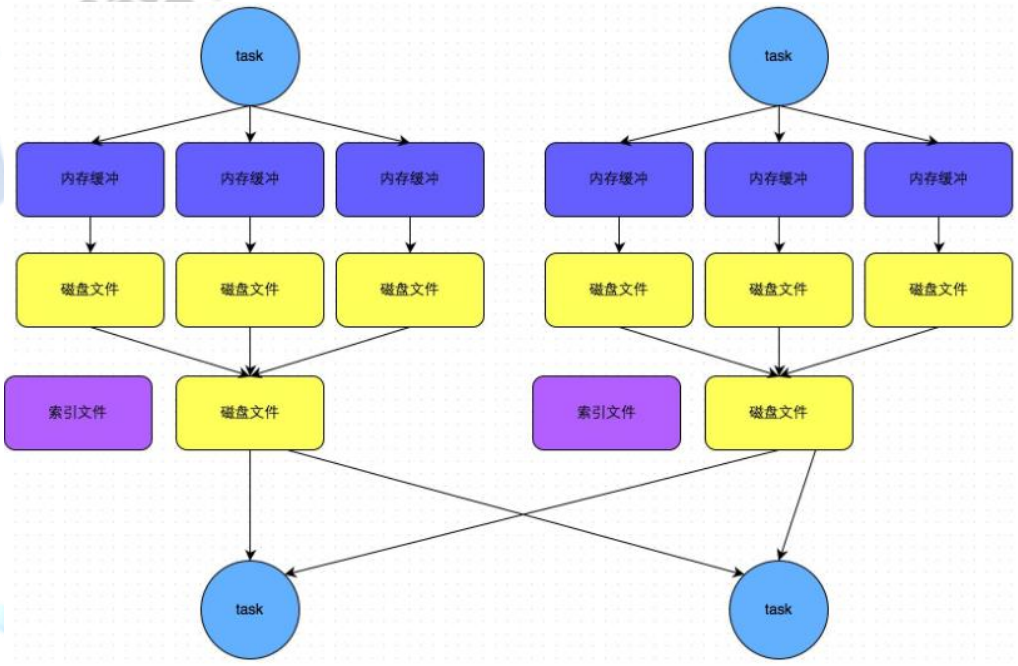
g) reduce task 去 map 端拉取数据的时候，首先解析索引文件，根据索引文件再去拉取对应的数据。

➢ 总结

产生磁盘小文件的个数： 2*M（map task 的个数）

2) 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。

② .产生的磁盘小文件为：2*M（map task 的个数）

4. Shuffle 文件寻址

1) 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。

2) 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。
- BlockManagerWorker，从对象，存在于 Excutor 中。

  BlockManagerWorker 会与 BlockManagerWorker 之间通信。

✶ 无论在 Driver 端的 BlockManager 还是在 Excutor 端的 BlockManager 都含有四个对象：

① DiskStore:负责磁盘的管理。

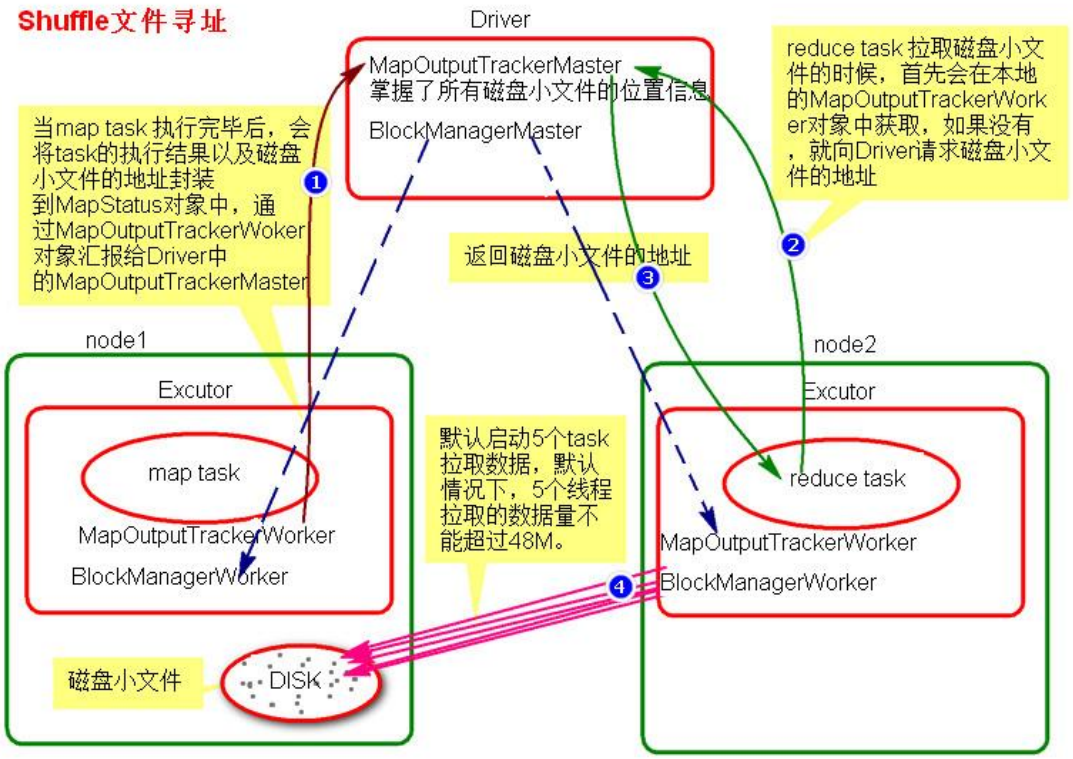
② MemoryStore：负责内存的管理。

③ ConnectionManager：负责连接其他的

BlockManagerWorker。

④ BlockTransferService:负责数据的传输。

3) Shuffle 文件寻址图



4) Shuffle 文件寻址流程

a) 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。

b) 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。

c) 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。

d) 获取到磁盘小文件的地址后，会通过 BlockManager 中的 ConnectionManager 连接数据所在节点上的 ConnectionManager,然后通过 BlockTransferService 进行数据的传输。

e) 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。

6. Spark 内存管理

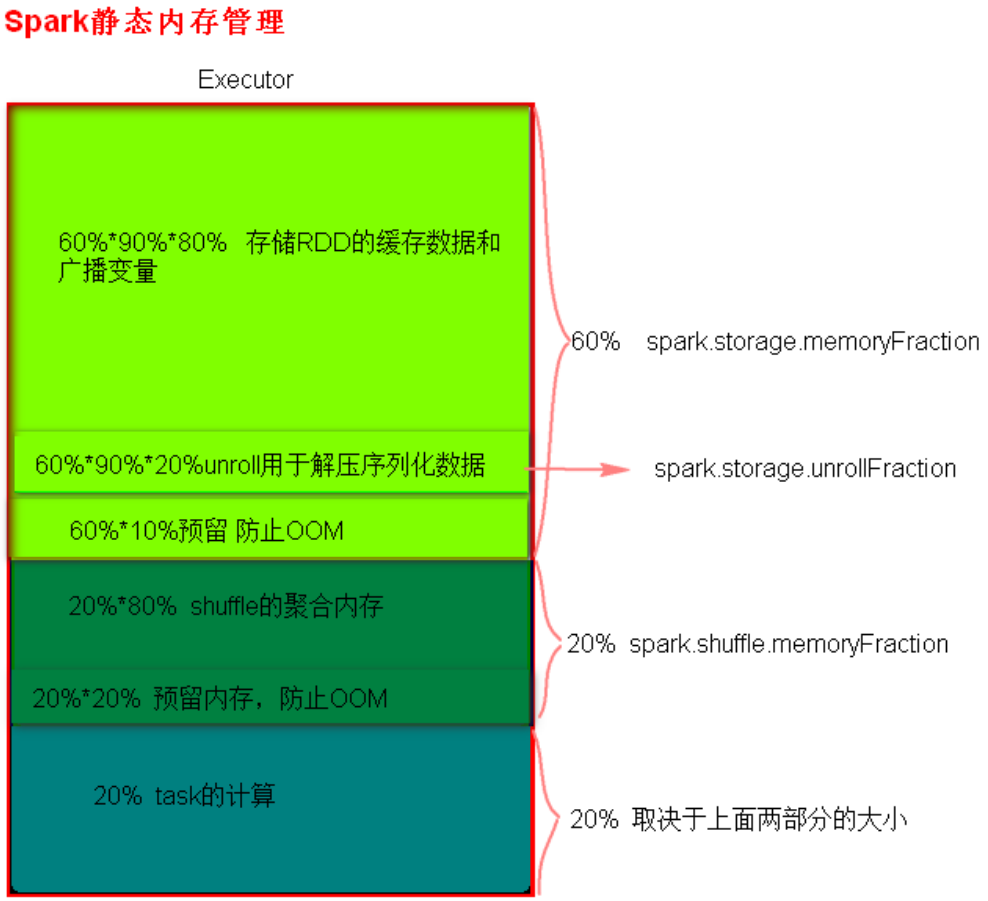
Spark 执行应用程序时，Spark 集群会启动 Driver 和 Executor 两种 JVM 进程，Driver 负责创建 SparkContext 上下文，提交任务，task 的分发等。Executor 负责 task 的计算任务，并将结果返回给 Driver。同时需要为需要持久化的 RDD 提供储存。Driver 端的内存管理比较简单，这里所说的 Spark 内存管理针对 Executor 端的内存管理。

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。

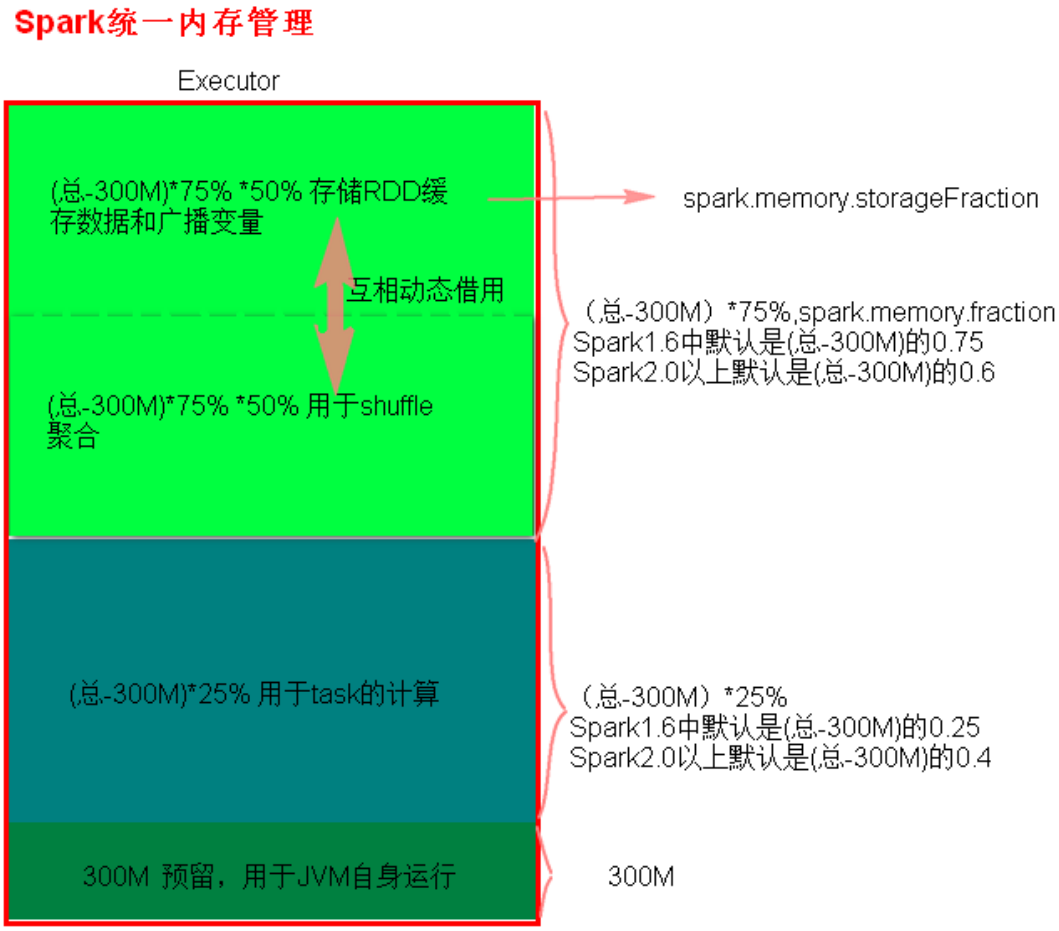
静态内存管理中存储内存、执行内存和其他内存的大小在 Spark 应用程序运行期间均为固定的，但用户可以应用程序启动前进行配置。
统一内存管理与静态内存管理的区别在于储存内存和执行内存共享同一块空间，可以互相借用对方的空间。

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


Spark静态内存管理
Executor
60%*90%*80% 存储RDD的缓存数据和广播变量
60%*90%*20%unroll用于解压序列化数据
60%*10%预留 防止OOM
20%*80% shuffle的聚合内存
20%*20% 预留内存，防止OOM
20% task的计算
60% spark.storage.memoryFraction
spark.storage.unrollFraction
20% spark.shuffle.memoryFraction
20% 取决于上面两部分的大小

2. 统一内存管理分布图



3. reduce 中 OOM 如何处理？

   1) 减少每次拉取的数据量

   2) 提高 shuffle 聚合的内存比例

   3) 提高 Excutor 的总内存

7. Shuffle 调优

1. SparkShuffle 调优配置项如何使用？

   1) 在代码中,不推荐使用，硬编码。

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

   2) 在提交 spark 任务的时候，推荐使用。

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

3) 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