1. 分析一下一段spark代码中哪些部分在Driver端执行,哪些部分在Worker端执行

Driver Program是用户编写的提交给Spark集群执行的application，它包含两部分

- **作为驱动**： Driver与Master、Worker协作完成application进程的启动、DAG划分、计算任务封装、计算任务分发到各个计算节点(Worker)、计算资源的分配等。
- **计算逻辑本身**，当计算任务在Worker执行时，执行计算逻辑完成application的计算任务。

一般来说transformation算子均是在worker上执行的,其他类型的代码在driver端执行。

2. 讲一下spark的几种部署方式

目前,除了local模式为本地调试模式以为, Spark支持三种分布式部署方式，分别是standalone、spark on mesos和 spark on YARN

- **Standalone模式**

  即独立模式，自带完整的服务，可单独部署到一个集群中，无需依赖任何其他资源管理系统。从一定程度上说，该模式是其他两种的基础。目前Spark在standalone模式下是没有任何单点故障问题的，这是借助zookeeper实现的，思想类似于Hbase master单点故障解决方案。将Spark standalone与MapReduce比较，会发现它们两个在架构上是完全一致的：

  - 都是由master/slaves服务组成的，且起初master均存在单点故障，后来均通过zookeeper解决（Apache MRv1的JobTracker仍存在单点问题，但CDH版本得到了解决）；
  - 各个节点上的资源被抽象成粗粒度的slot，有多少slot就能同时运行多少task。不同的是，MapReduce将slot分为map slot和reduce slot，它们分别只能供Map Task和Reduce Task使用，而不能共享，这是MapReduce资源利率低效的原因之一，而Spark则更优化一些，它不区分slot类型，只有一种slot，可以供各种类型的Task使用，这种方式可以提高资源利用率，但是不够灵活，不能为不同类型的Task定制slot资源。总之，这两种方式各有优缺点。
- **Spark On YARN模式**

  spark on yarn 的支持两种模式：

  - yarn-cluster：适用于生产环境；
  - yarn-client：适用于交互、调试，希望立即看到app的输出

  yarn-cluster和yarn-client的区别在于yarn appMaster，每个yarn app实例有一个appMaster进程，是为app启动的第一个container；负责从ResourceManager请求资源，获取到资源后，告诉NodeManager为其启动container。yarn-cluster和yarn-client模式内部实现还是有很大的区别。如果你需要用于生产环境，那么请选择yarn-cluster；而如果你仅仅是Debug程序，可以选择yarn-client。
- **Spark On Mesos模式**

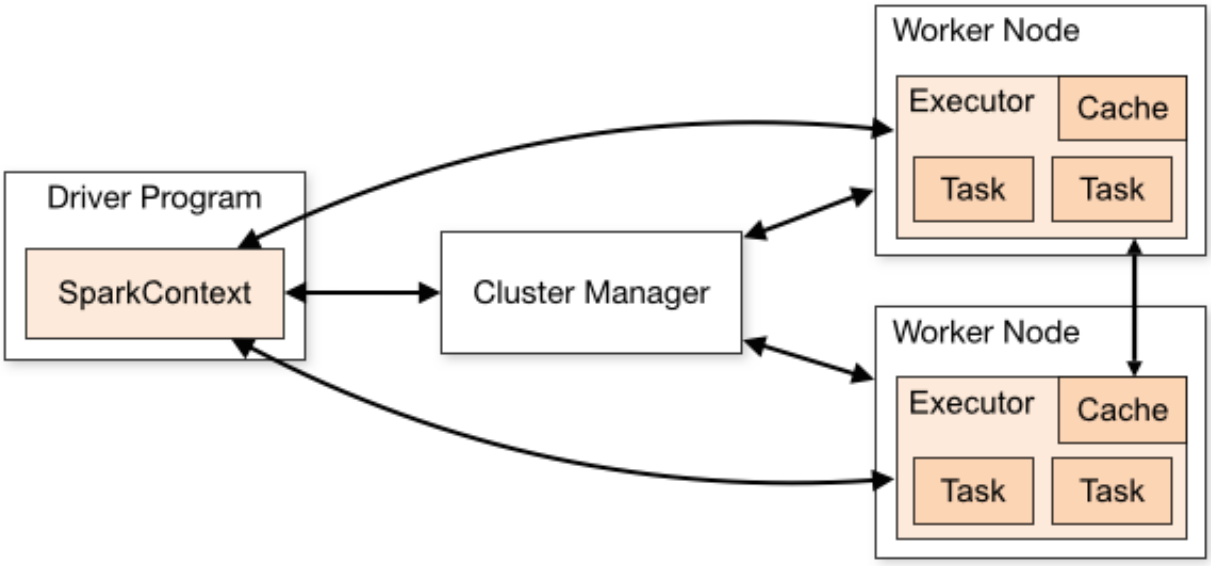
  Spark运行在Mesos上会比运行在YARN上更加灵活，更加自然。目前在Spark On Mesos环境中，用户可选择两种调度模式之一运行自己的应用程序

  - 粗粒度模式（Coarse-grained Mode）：每个应用程序的运行环境由一个Dirver和若干个Executor组成，其中，每个Executor占用若干资源，内部可运行多个Task（对应多少个“slot”）。应用程序的各个

任务正式运行之前，需要将运行环境中的资源全部申请好，且运行过程中要一直占用这些资源，即使不用，最后程序运行结束后，回收这些资源。

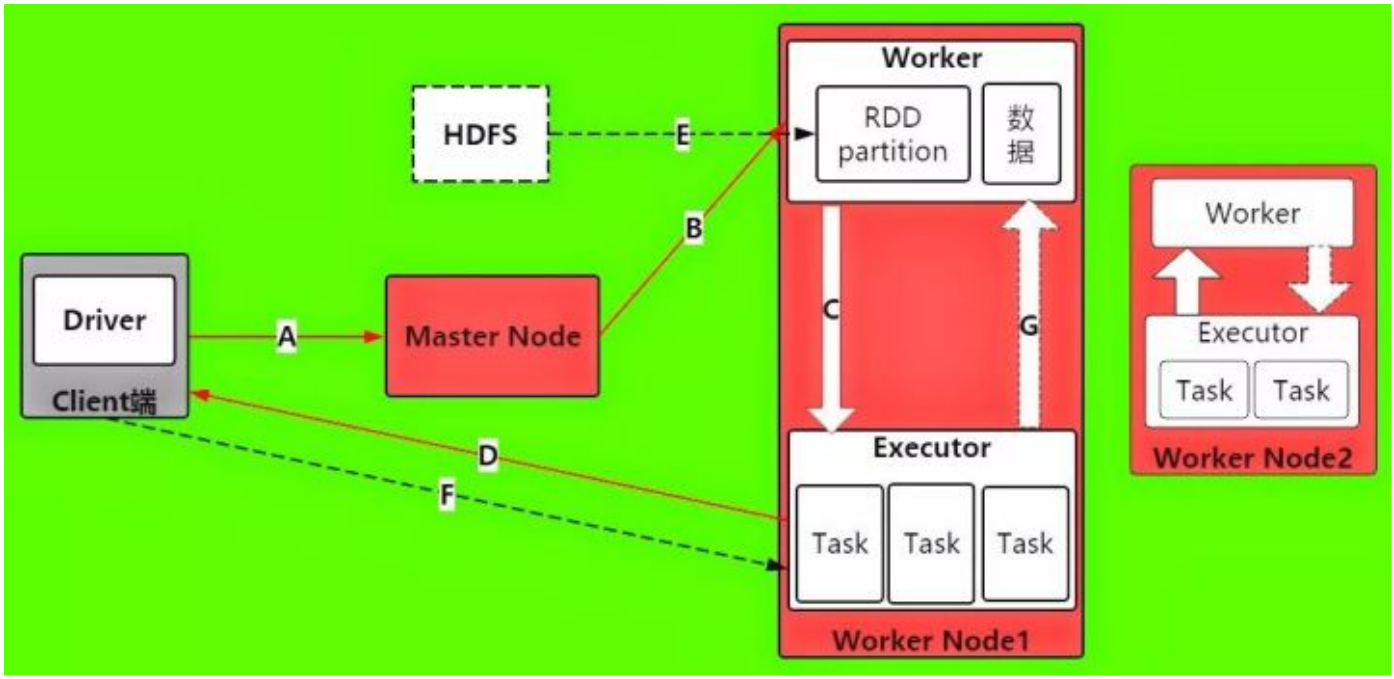
- 细粒度模式（Fine-grained Mode）：鉴于粗粒度模式会造成大量资源浪费，Spark On Mesos还提供了另外一种调度模式：细粒度模式，这种模式类似于现在的云计算，思想是按需分配。与粗粒度模式一样，应用程序启动时，先会启动executor，但每个executor占用资源仅仅是自己运行所需的资源，不需要考虑将来要运行的任务，之后，mesos会为每个executor动态分配资源，每分配一些，便可以运行一个新任务，单个Task运行完之后可以马上释放对应的资源。

3. 讲一下spark 的运行架构



- **Cluster Manager(Master)**：在standalone模式中即为Master主节点，控制整个集群，监控worker。在YARN模式中为资源管理器
- **Worker节点**：从节点，负责控制计算节点，启动Executor或者Driver。
- **Driver**： 运行Application 的main()函数
- **Executor**：执行器，是为某个Application运行在worker node上的一个进程

4. 一个spark程序的执行流程



- **A ->** 当 Driver 进程被启动之后,首先它将发送请求到Master节点上,进行Spark应用程序的注册
- **B ->** Master在接受到Spark应用程序的注册申请之后,会发送给Worker,让其进行资源的调度和分配.
- **C ->** Worker 在接受Master的请求之后,会为Spark应用程序启动Executor, 来分配资源
- **D ->** Executor启动分配资源好后,就会想Driver进行反注册,这是Driver已经知道哪些Executor为他服务了
- **E ->** 当Driver得到注册了Executor之后,就可以开始正式执行spark应用程序了. 首先第一步,就是创建初始RDD,读取数据源,再执行之后的一系列算子. HDFS文件内容被读取到多个worker节点上,形成内存中的分布式数据集,也就是初始RDD
- **F ->** Driver就会根据 Job 任务任务中的算子形成对应的task,最后提交给 Executor, 来分配给task进行计算的线程
- **G ->** task就会去调用对应的任务数据来计算,并task会对调用过来的RDD的partition数据执行指定的算子操作,形成新的RDD的partition,这时一个大的循环就结束了
- 后续的RDD的partition数据又通过Driver形成新的一批task提交给Executor执行,循环这个操作,直到所有的算子结束

5. spark2.0为什么放弃了akka 而用netty

1. 很多Spark用户也使用Akka，但是由于Akka不同版本之间无法互相通信，这就要求用户必须使用跟Spark完全一样的Akka版本，导致用户无法升级Akka。
2. Spark的Akka配置是针对Spark自身来调优的，可能跟用户自己代码中的Akka配置冲突。
3. Spark用的Akka特性很少，这部分特性很容易自己实现。同时，这部分代码量相比Akka来说少很多，debug比较容易。如果遇到什么bug，也可以自己马上fix，不需要等Akka上游发布新版本。而且，Spark升级Akka本身又因为第一点会强制要求用户升级他们使用的Akka，对于某些用户来说是不现实的。

6. spark的各种HA：master/worker/executor的HA

- **Master异常**

  spark可以在集群运行时启动一个或多个standby Master,当 Master 出现异常时,会根据规则启动某个standby master接管,在standlone模式下有如下几种配置

  - ZOOKEEPER

    集群数据持久化到zk中,当master出现异常时,zk通过选举机制选出新的master,新的master接管是需要从zk获取持久化信息

  - FILESYSTEM

    集群元数据信息持久化到本地文件系统, 当master出现异常时,只需要在该机器上重新启动master,启动后新的master获取持久化信息并根据这些信息恢复集群状态

  - CUSTOM

    自定义恢复方式,对 standloneRecoveryModeFactory 抽象类 进行实现并把该类配置到系统中,当master出现异常时,会根据用户自定义行为恢复集群

  - None

    不持久化集群的元数据, 当 master出现异常时, 新启动的Master 不进行恢复集群状态,而是直接接管集群

- **Worker异常**

  Worker 以定时发送心跳给 Master, 让 Master 知道 Worker 的实时状态,当worker出现超时时,Master 调用 timeOutDeadWorker 方法进行处理,在处理时根据 Worker 运行的是 Executor 和 Driver 分别进行处理

  - 如果是Executor, Master先把该 Worker 上运行的Executor 发送信息ExecutorUpdate给对应的Driver, 告知Executor已经丢失,同时把这些Executor从其应用程序列表删除, 另外, 相关Executor的异常也需要处理
  - 如果是Driver, 则判断是否设置重新启动,如果需要,则调用Master.shedule方法进行调度,分配合适节点重启Driver, 如果不需要重启, 则删除该应用程序

- **Executor异常**

  1. Executor发生异常时由ExecutorRunner捕获该异常并发送ExecutorStateChanged信息给Worker
  2. Worker接收到消息时, 在Worker的 handleExecutorStateChanged 方法中, 根据Executor状态进行信息更新,同时把Executor状态发送给Master
  3. Master在接受Executor状态变化消息之后,如果发现其是异常退出,会尝试可用的Worker节点去启动 Executor

7. spark的内存管理机制

spark的内存结构分为3大块:storage/execution/系统自留

- **storage 内存**：用于缓存 RDD、展开 partition、存放 Direct Task Result、存放广播变量。在 Spark Streaming receiver 模式中，也用来存放每个 batch 的 blocks
- **execution 内存**：用于 shuffle、join、sort、aggregation 中的缓存、buffer
- **系统自留**:
  - 在 spark 运行过程中使用：比如序列化及反序列化使用的内存，各个对象、元数据、临时变量使用的内存，函数调用使用的堆栈等

- 作为误差缓冲：由于 storage 和 execution 中有很多内存的使用是估算的，存在误差。当 storage 或 execution 内存使用超出其最大限制时，有这样一个安全的误差缓冲在可以大大减小 OOM 的概率
-

1.6版本以前的问题

- 旧方案最大的问题是 storage 和 execution 的内存大小都是固定的，不可改变，即使 execution 有大量的空闲内存且 storage 内存不足，storage 也无法使用 execution 的内存，只能进行 spill，反之亦然。所以，在很多情况下存在资源浪费
- 旧方案中，只有 execution 内存支持 off heap，storage 内存不支持 off heap

新方案的改进

- 新方案 storage 和 execution 内存可以互相借用，当一方内存不足可以向另一方借用内存，提高了整体的资源利用率
- 新方案中 execution 内存和 storage 内存均支持 off heap

8. Spark的 partitioner 都有哪些?

Partitioner主要有两个实现类：HashPartitioner和RangePartitioner,HashPartitioner是大部分transformation的默认实现，sortBy、sortByKey使用RangePartitioner实现，也可以自定义Partitioner.

- **HashPartitioner**

  numPartitions方法返回传入的分区数，getPartition方法使用key的hashCode值对分区数取模得到PartitionId，写入到对应的bucket中。

- **RangePartitioner**

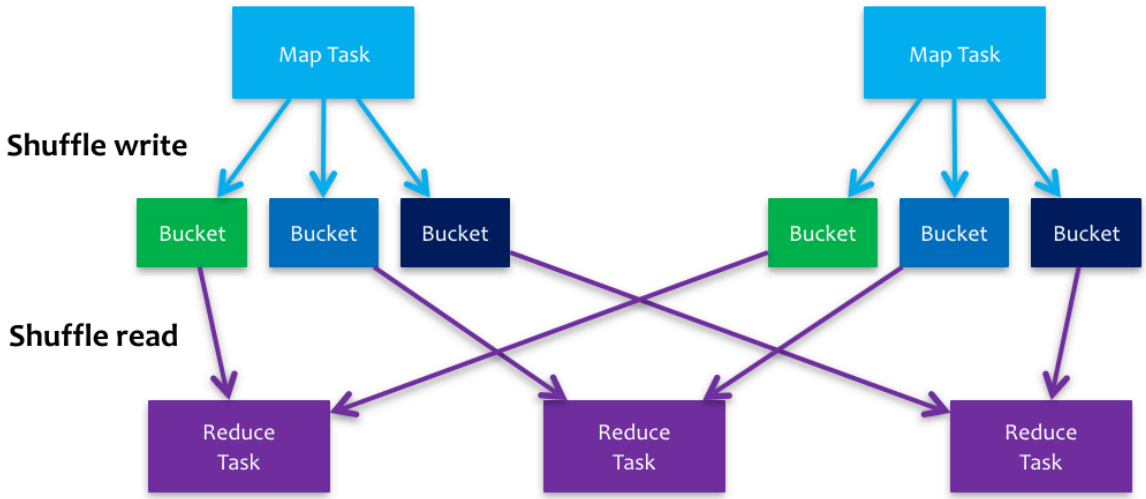
  RangePartitioner是先根据所有partition中数据的分布情况，尽可能均匀地构造出重分区的分隔符，再将数据的key值根据分隔符进行重新分区

  - 使用reservoir Sample方法对每个Partition进行分别抽样
  - 对数据量大(大于sampleSizePerPartition)的分区进行重新抽样
  - 由权重信息计算出分区分隔符rangeBounds
  - 由rangeBounds计算分区数和key的所属分区

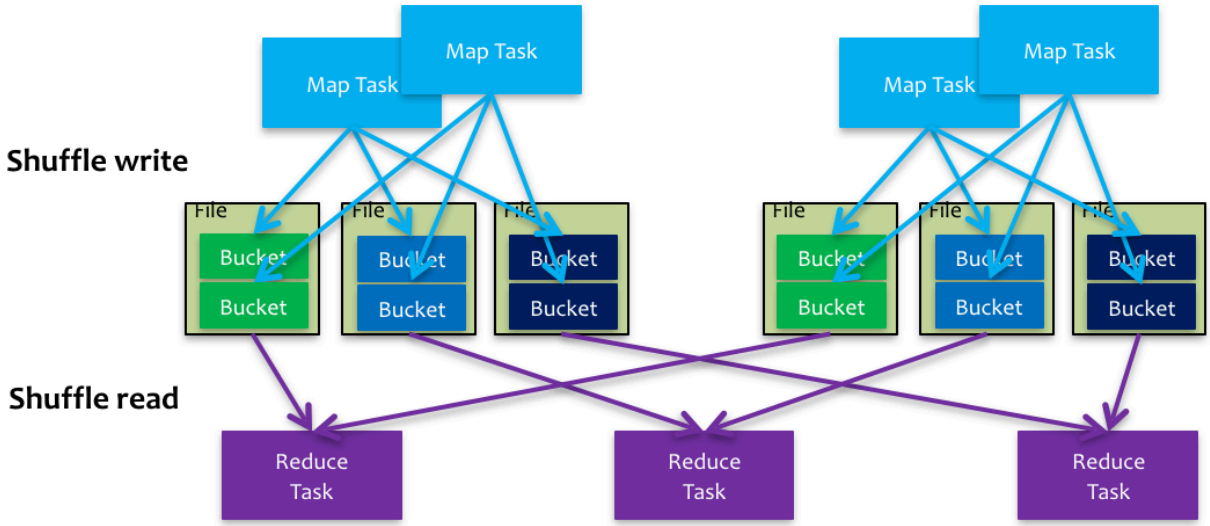
9. spark的shuffle介绍

spark中的shuffle主要有3种:

- **Hash Shuffle** 2.0以后移除



早期版本（1.1.0以前）的Hash Shuffle v1

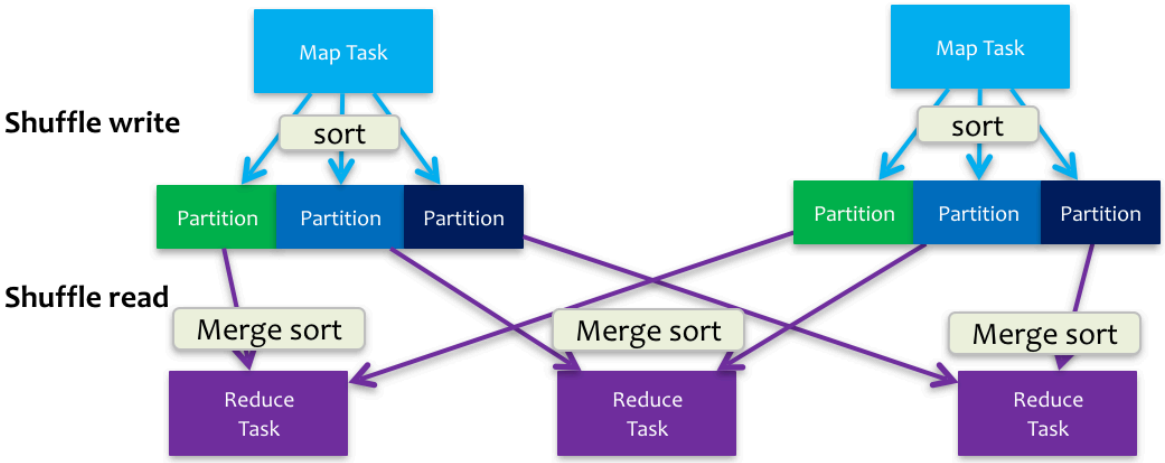


早期版本（1.1.0以前）的Hash Shuffle v2

在map阶段(shuffle write)，每个map都会为下游stage的每个partition写一个临时文件，假如下游stage有1000个partition，那么每个map都会生成1000个临时文件，一般来说一个executor上会运行多个map task，这样下来，一个executor上会有非常多的临时文件，假如一个executor上运行M个map task，下游stage有N个partition，那么一个executor上会生成MN个文件。*另一方面，如果一个executor上有K个core，那么executor同时可运行K个task，这样一来，就会同时申请*KN个文件描述符，一旦partition数较多，势必会耗尽executor上的文件描述符，同时生成K*N个write handler也会带来大量内存的消耗。

在reduce阶段(shuffle read)，每个reduce task都会拉取所有map对应的那部分partition数据，那么executor会打开所有临时文件准备网络传输，这里又涉及到大量文件描述符，另外，如果reduce阶段有combiner操作，那么它会把网络中拉到的数据保存在一个`HashMap`中进行合并操作，如果数据量较大，很容易引发OOM操作。

- **Sort Shuffle** 1.1开始(sort shuffle也经历过优化升级,详细见参考文章1)



后期版本（1.1.0以后）的 Sort Shuffle

在map阶段(shuffle write)，会按照partition id以及key对记录进行排序，将所有partition的数据写在同一个文件中，该文件中的记录首先是按照partition id排序一个一个分区的顺序排列，每个partition内部是按照key进行排序存放，map task运行期间会顺序写每个partition的数据，并通过一个索引文件记录每个partition的大小和偏移量。这样一来，每个map task一次只开两个文件描述符，一个写数据，一个写索引，大大减轻了Hash Shuffle大量文件描述符的问题，即使一个executor有K个core，那么最多一次性开K*2个文件描述符。

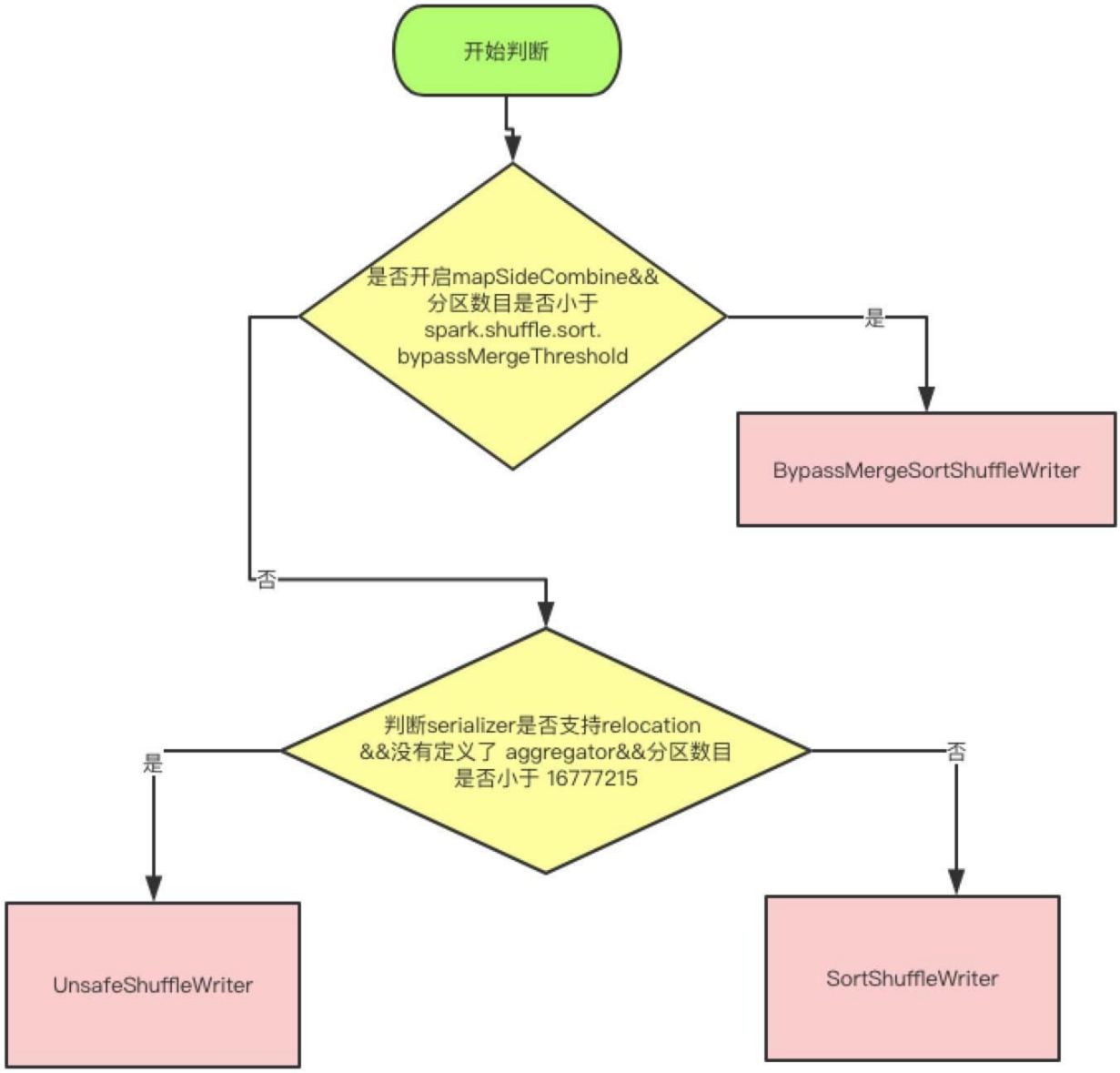
在reduce阶段(shuffle read)，reduce task拉取数据做combine时不再是采用`HashMap`，而是采用`ExternalAppendOnlyMap`，该数据结构在做combine时，如果内存不足，会刷写磁盘，很大程度的保证了鲁棒性，避免大数据情况下的OOM。

- **Unsafe Shuffle** 1.5开始, 1.6与Sort shuffle合并

  从spark 1.5.0开始，spark开始了钨丝计划(Tungsten)，目的是优化内存和CPU的使用，进一步提升spark的性能。为此，引入Unsafe Shuffle，它的做法是将数据记录用二进制的方式存储，直接在序列化的二进制数据上sort而不是在java 对象上，这样一方面可以减少memory的使用和GC的开销，另一方面避免shuffle过程中频繁的序列化以及反序列化。在排序过程中，它提供cache-efficient sorter，使用一个8 bytes的指针，把排序转化成了一个指针数组的排序，极大的优化了排序性能.

现在2.1 分为三种writer， 分为 BypassMergeSortShuffleWriter， SortShuffleWriter 和 UnsafeShuffleWriter

三种Writer的分类

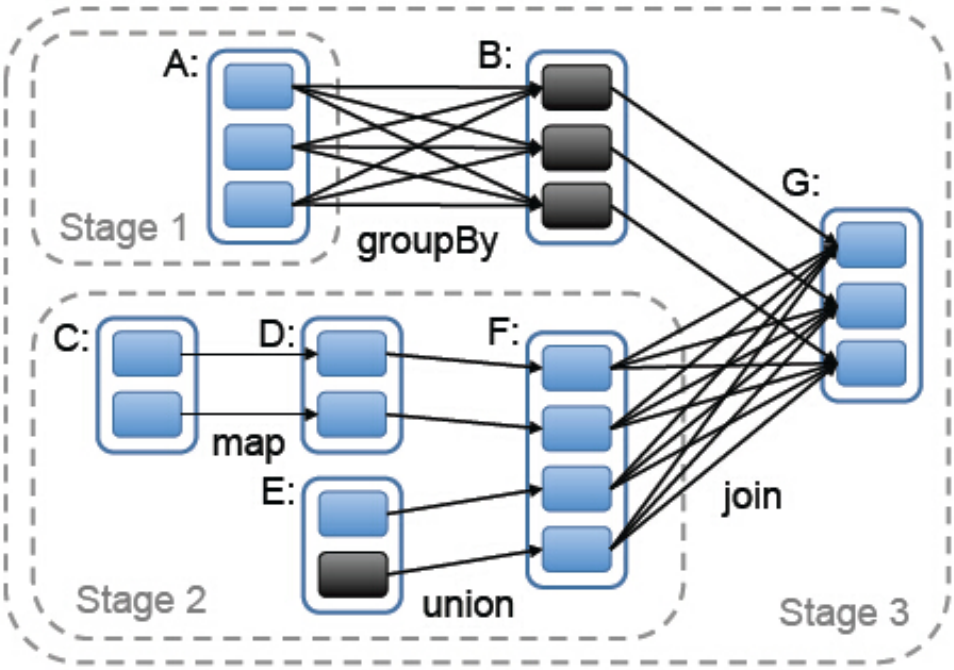


上面是使用哪种 writer 的判断依据， 是否开启 mapSideCombine 这个判断，是因为有些算子会在 map 端先进行一次 combine， 减少传输数据。 因为 BypassMergeSortShuffleWriter 会临时输出Reducer个（分区数目）小文件，所以分区数必须要小于一个阀值，默认是小于200

UnsafeShuffleWriter需要Serializer支持relocation，Serializer支持relocation：原始数据首先被序列化处理，并且再也不需要反序列，在其对应的元数据被排序后，需要Serializer支持relocation，在指定位置读取对应数据

10. spark的stage是如何划分的

stage的划分依据就是看是否产生了shuflle(即宽依赖),遇到一个shuffle操作就划分为前后两个stage。



11. spark有哪几种join

Spark 中和 join 相关的算子有：`join`、`fullOuterJoin`、`leftOuterJoin`、`rightOuterJoin`

- **join**

  join函数会输出两个RDD中key相同的所有项，并将它们的value联结起来，它联结的key要求在两个表中都存在，类似于SQL中的INNER JOIN。但它不满足交换律，a.join(b)与b.join(a)的结果不完全相同，值插入的顺序与调用关系有关。

- **leftOuterJoin**

  leftOuterJoin会保留对象的所有key，而用None填充在参数RDD other中缺失的值，因此调用顺序会使结果完全不同。如下面展示的结果，

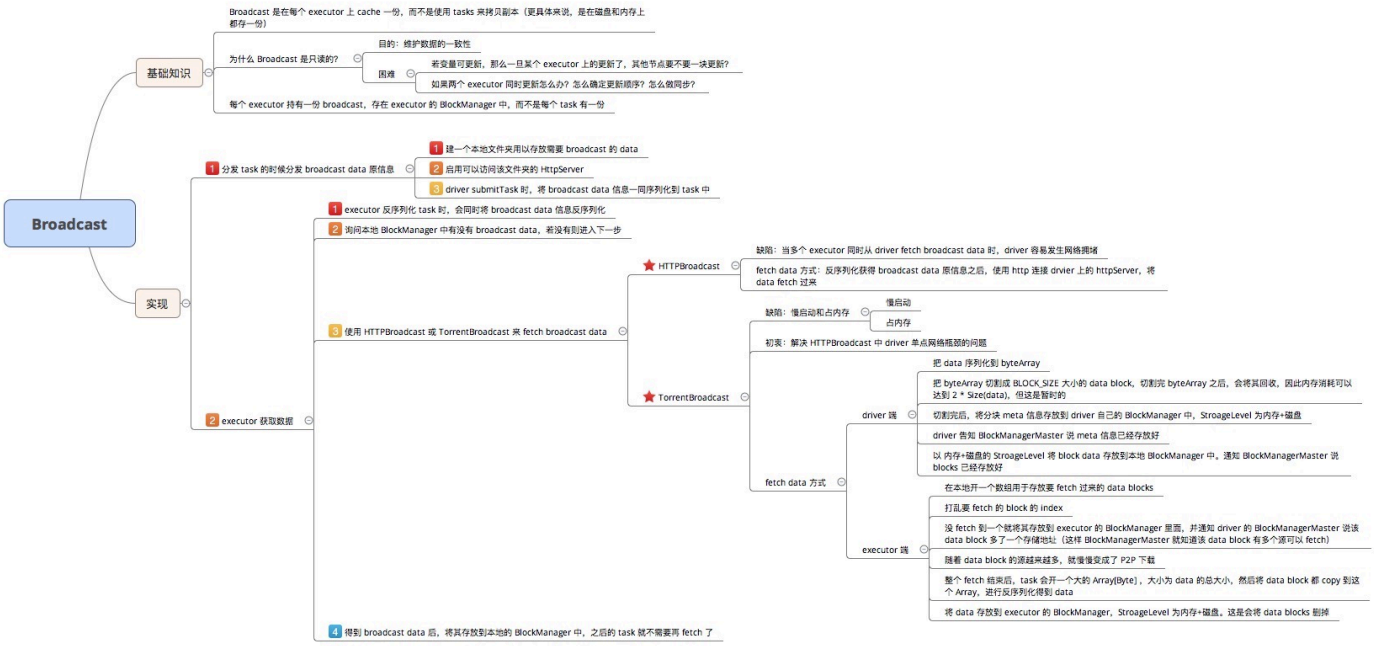
- **rightOuterJoin**

  rightOuterJoin与leftOuterJoin基本一致，区别在于它的结果保留的是参数other这个RDD中所有的key。

- **fullOuterJoin**

  fullOuterJoin会保留两个RDD中所有的key，因此所有的值列都有可能出现缺失的情况，所有的值列都会转为Some对象。

12. spark中的广播变量



顾名思义，broadcast 就是将数据从一个节点发送到其他各个节点上去。这样的场景很多，比如 driver 上有一张表，其他节点上运行的 task 需要 lookup 这张表，那么 driver 可以先把这张表 copy 到这些节点，这样 task 就可以在本地查表了。如何实现一个可靠高效的 broadcast 机制是一个有挑战性的问题。先看看 Spark 官网上的一段话：

Broadcast variables allow the programmer to keep a **read-only** variable cached on each **machine** rather than shipping a copy of it with **tasks**. They can be used, for example, to give every node a copy of a **large input dataset** in an efficient manner. Spark also attempts to distribute broadcast variables using **efficient** broadcast algorithms to reduce communication cost.

问题：为什么只能 broadcast 只读的变量?

这就涉及一致性的问题，如果变量可以被更新，那么一旦变量被某个节点更新，其他节点要不要一块更新？如果多个节点同时在更新，更新顺序是什么？怎么做同步？还会涉及 fault-tolerance 的问题。为了避免维护数据一致性问题，Spark 目前只支持 broadcast 只读变量。

问题：broadcast 到节点而不是 broadcast 到每个 task?

因为每个 task 是一个线程，而且同在一个进程运行 tasks 都属于同一个 application。因此每个节点（executor）上放一份就可以被所有 task 共享。

问题： 具体怎么用 broadcast?

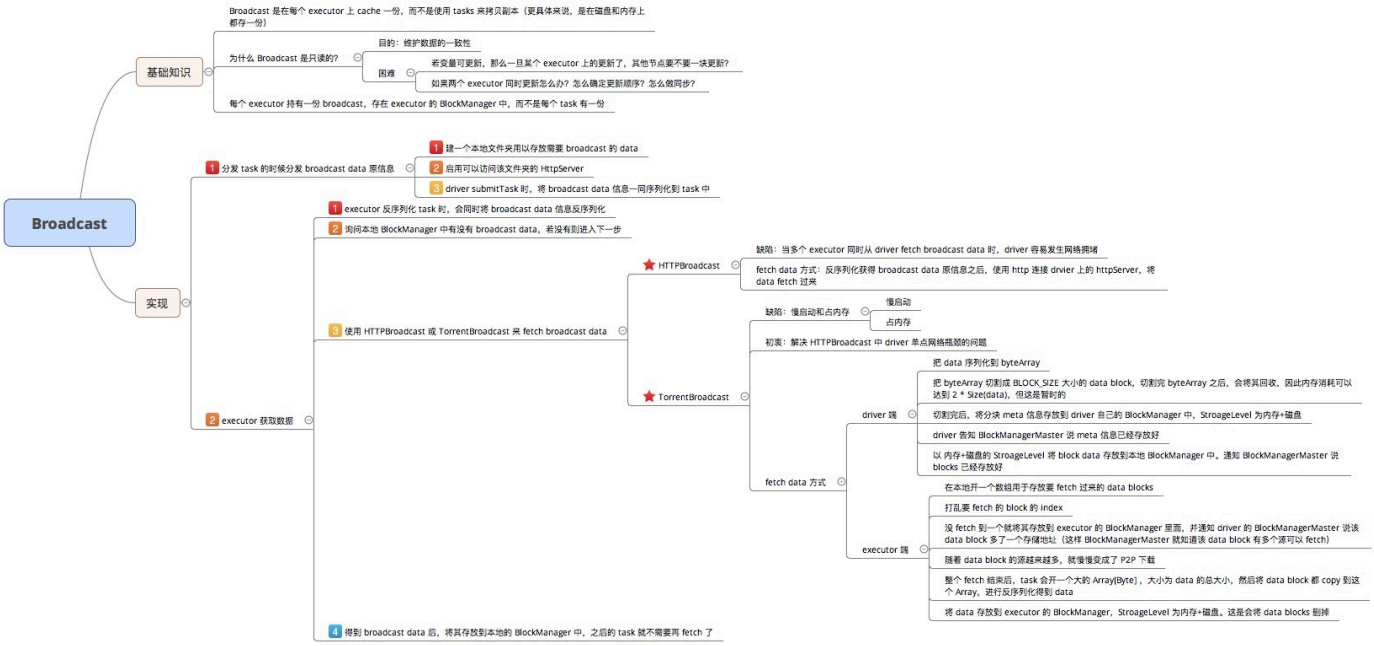
driver program 例子：

```
val data = List(1, 2, 3, 4, 5, 6)
val bdata = sc.broadcast(data)

val rdd = sc.parallelize(1 to 6, 2)
val observedSizes = rdd.map(_ => bdata.value.size)
```

driver 使用 `sc.broadcast()` 声明要 broadcast 的 data，bdata 的类型是 Broadcast。

当 `rdd.transformation(func)` 需要用 bdata 时，直接在 func 中调用，比如上面的例子中的 map() 就使用了 bdata.value.size。



问题：怎么实现 broadcast?

broadcast 的实现机制很有意思：

1. 分发 task 的时候先分发 bdata 的元信息

Driver 先建一个本地文件夹用以存放需要 broadcast 的 data，并启动一个可以访问该文件夹的 HttpServer。当调用`val bdata = sc.broadcast(data)`时就把 data 写入文件夹，同时写入 driver 自己的 blockManger 中（StorageLevel 为内存＋磁盘），获得一个 blockId，类型为 BroadcastBlockId。当调用`rdd.transformation(func)`时，如果 func 用到了 bdata，那么 driver submitTask() 的时候会将 bdata 一同 func 进行序列化得到 serialized task，**注意序列化的时候不会序列化 bdata 中包含的 data**。上一章讲到 serialized task 从 driverActor 传递到 executor 时使用 Akka 的传消息机制，消息不能太大，而实际的 data 可能很大，所以这时候还不能 broadcast data。

> driver 为什么会同时将 data 放到磁盘和 blockManager 里面？放到磁盘是为了让 HttpServer 访问到，放到 blockManager 是为了让 driver program 自身使用 bdata 时方便（其实我觉得不放到 blockManger 里面也行）。

那么什么时候传送真正的 data? 在 executor 反序列化 task 的时候，会同时反序列化 task 中的 bdata 对象，这时候会调用 bdata 的 readObject() 方法。该方法先去本地 blockManager 那里询问 bdata 的 data 在不在 blockManager 里面，如果不在就使用下面的两种 fetch 方式之一去将 data fetch 过来。得到 data 后，将其存放到 blockManager 里面，这样后面运行的 task 如果需要 bdata 就不需要再去 fetch data 了。如果在，就直接拿来用了。

下面探讨 broadcast data 时候的两种实现方式：

2. HttpBroadcast

顾名思义，HttpBroadcast 就是每个 executor 通过的 http 协议连接 driver 并从 driver 那里 fetch data。

Driver 先准备好要 broadcast 的 data，调用`sc.broadcast(data)`后会调用工厂方法建立一个 HttpBroadcast 对象。该对象做的第一件事就是将 data 存到 driver 的 blockManager 里面，StorageLevel 为内存＋磁盘，blockId 类型为 BroadcastBlockId。

同时 driver 也会将 broadcast 的 data 写到本地磁盘，例如写入后得到`/var/folders/87/grpn1_fn4xq5wdqmxk31v0l00000gp/T/spark-6233b09c-3c72-4a4d-832b-6c0791d0eb9c/broadcast_0`， 这个文件夹作为 HttpServer 的文件目录。

> Driver 和 executor 启动的时候，都会生成 broadcastManager 对象，调用 HttpBroadcast.initialize()，driver 会在本地建立一个临时目录用来存放 broadcast 的 data，并启动可以访问该目录的 httpServer。

Fetch data：在 executor 反序列化 task 的时候，会同时反序列化 task 中的 bdata 对象，这时候会调用 bdata 的 readObject() 方法。该方法先去本地 blockManager 那里询问 bdata 的 data 在不在 blockManager 里面，**如果不在就使用 http 协议连接 driver 上的 httpServer，将 data fetch 过来。**得到 data 后，将其存放到 blockManager 里面，这样后面运行的 task 如果需要 bdata 就不需要再去 fetch data 了。如果在，就直接拿来用了。

HttpBroadcast 最大的问题就是 **driver 所在的节点可能会出现网络拥堵**，因为 worker 上的 executor 都会去 driver 那里 fetch 数据。

3. TorrentBroadcast

为了解决 HttpBroadast 中 driver 单点网络瓶颈的问题，Spark 又设计了一种 broadcast 的方法称为 TorrentBroadcast，**这个类似于大家常用的 BitTorrent 技术。**基本思想就是将 data 分块成 data blocks，然后假设有 executor fetch 到了一些 data blocks，那么这个 executor 就可以被当作 data server 了，随着 fetch 的 executor 越来越多，有更多的 data server 加入，data 就很快能传播到全部的 executor 那里去了。

HttpBroadcast 是通过传统的 http 协议和 httpServer 去传 data，在 TorrentBroadcast 里面使用在上一章介绍的 blockManager.getRemote() => NIO ConnectionManager 传数据的方法来传递，读取数据的过程与读取 cached rdd 的方式类似。

driver 端：

Driver 先把 data 序列化到 byteArray，然后切割成 BLOCK_SIZE（由 `spark.broadcast.blockSize = 4MB` 设置）大小的 data block，每个 data block 被 TorrentBlock 对象持有。切割完 byteArray 后，会将其回收，因此内存消耗虽然可以达到 2 * Size(data)，但这是暂时的。

完成分块切割后，就将分块信息（称为 meta 信息）存放到 driver 自己的 blockManager 里面，StorageLevel 为内存＋磁盘，同时会通知 driver 自己的 blockManagerMaster 说 meta 信息已经存放好。**通知 blockManagerMaster 这一步很重要，因为 blockManagerMaster 可以被 driver 和所有 executor 访问到，信息被存放到 blockManagerMaster 就变成了全局信息。**

之后将每个分块 data block 存放到 driver 的 blockManager 里面，StorageLevel 为内存＋磁盘。存放后仍然通知 blockManagerMaster 说 blocks 已经存放好。到这一步，driver 的任务已经完成。

Executor 端：

executor 收到 serialized task 后，先反序列化 task，这时候会反序列化 serialized task 中包含的 bdata 类型是 TorrentBroadcast，也就是去调用 TorrentBroadcast.readObject()。这个方法首先得到 bdata 对象，**然后发现 bdata 里面没有包含实际的 data。怎么办？**先询问所在的 executor 里的 blockManager 是会否包含 data（通过查询 data 的 broadcastId），包含就直接从本地 blockManager 读取 data。否则，就通过本地 blockManager 去连接 driver 的 blockManagerMaster 获取 data 分块的 meta 信息，获取信息后，就开始了 BT 过程。

BT 过程：task 先在本地开一个数组用于存放将要 fetch 过来的 data blocks `arrayOfBlocks = new Array[TorrentBlock](totalBlocks)`，TorrentBlock 是对 data block 的包装。然后打乱要 fetch 的 data blocks 的顺序，比如如果 data block 共有 5 个，那么打乱后的 fetch 顺序可能是 3-1-2-4-5。然后按照打乱后的顺序去 fetch 一个个 data block。fetch 的过程就是通过 “本地 blockManager －本地 connectionManager－driver/executor 的 connectionManager－driver/executor 的 blockManager－data” 得到 data，这个过程与 fetch cached rdd 类似。**每 fetch 到一个 block 就将其存放到 executor 的 blockManager 里面，同时通知 driver 上的 blockManagerMaster 说该 data block 多了一个存储地址。**这一步通知非常重要，意味着 blockManagerMaster 知道 data block 现在在 cluster 中有多份，下一个不同节点上的 task 再去 fetch 这个 data block 的时候，可以有两个选择了，而且会随机选择一个去 fetch。这个过程持续下去就是 BT 协议，随着下载的客户端越来越多，data block 服务器也越来越多，就变成 p2p下载了。关于 BT 协议，Wikipedia 上有一个动画。

整个 fetch 过程结束后，task 会开一个大 Array[Byte]，大小为 data 的总大小，然后将 data block 都 copy 到这个 Array，然后对 Array 中 bytes 进行反序列化得到原始的 data，这个过程就是 driver 序列化 data 的反过程。

最后将 data 存放到 task 所在 executor 的 blockManager 里面，StorageLevel 为内存＋磁盘。显然，这时候 data 在 blockManager 里存了两份，不过等全部 executor 都 fetch 结束，存储 data blocks 那份可以删掉了。

问题：broadcast RDD 会怎样?

不会怎样，就是这个rdd在每个executor中实例化一份。

Discussion

公共数据的 broadcast 是很实用的功能，在 Hadoop 中使用 DistributedCache，比如常用的`-libjars`就是使用 DistributedCache 来将 task 依赖的 jars 分发到每个 task 的工作目录。不过分发前 DistributedCache 要先将文件上传到 HDFS。这种方式的主要问题是**资源浪费**，如果某个节点上要运行来自同一 job 的 4 个 mapper，那么公共数据会在该节点上存在 4 份（每个 task 的工作目录会有一份）。但是通过 HDFS 进行 broadcast 的好处在于**单点瓶颈不明显**，因为公共 data 首先被分成多个 block，然后不同的 block 存放在不同的节点。这样，只要所有的 task 不是同时去同一个节点 fetch 同一个 block，网络拥塞不会很严重。

对于 Spark 来讲，broadcast 时考虑的不仅是如何将公共 data 分发下去的问题，还要考虑如何让同一节点上的 task 共享 data。

对于第一个问题，Spark 设计了两种 broadcast 的方式，传统存在单点瓶颈问题的 HttpBroadcast，和类似 BT 方式的 TorrentBroadcast。HttpBroadcast 使用传统的 client-server 形式的 HttpServer 来传递真正的 data，而 TorrentBroadcast 使用 blockManager 自带的 NIO 通信方式来传递 data。TorrentBroadcast 存在的问题是**慢启动**和**占内存**，慢启动指的是刚开始 data 只在 driver 上有，要等 executors fetch 很多轮 data block 后，data server 才会变得可观，后面的 fetch 速度才会变快。executor 所占内存的在 fetch 完 data blocks 后进行反序列化时需要将近两倍 data size 的内存消耗。不管哪一种方式，driver 在分块时会有两倍 data size 的内存消耗。

对于第二个问题，每个 executor 都包含一个 blockManager 用来管理存放在 executor 里的数据，将公共数据存放在 blockManager 中（StorageLevel 为内存＋磁盘），可以保证在 executor 执行的 tasks 能够共享 data。

其实 Spark 之前还尝试了一种称为 TreeBroadcast 的机制，详情可以见技术报告 Performance and Scalability of Broadcast in Spark。

更深入点，broadcast 可以用多播协议来做，不过多播使用 UDP，不是可靠的，仍然需要应用层的设计一些可靠性保障机制。

13. Spark中的算子都有哪些

总的来说,spark分为两大类算子:

- **Transformation 变换/转换算子：这种变换并不触发提交作业，完成作业中间过程处理**

  Transformation 操作是延迟计算的，也就是说从一个RDD 转换生成另一个 RDD 的转换操作不是马上执行，需要等到有 Action 操作的时候才会真正触发运算

- **Action 行动算子：这类算子会触发 SparkContext 提交 Job 作业**

  Action 算子会触发 Spark 提交作业（Job），并将数据输出 Spark系统

---

1. Value数据类型的Transformation算子

- 输入分区与输出分区一对一型
  - map算子
  - flatMap算子
  - mapPartitions算子
  - glom算子
- 输入分区与输出分区多对一型
  - union算子
  - cartesian算子
- 输入分区与输出分区多对多型
  - grouBy算子
- 输出分区为输入分区子集型
  - filter算子
  - distinct算子
  - subtract算子
  - sample算子
  - takeSample算子
- Cache型
  - cache算子
  - persist算子

2. Key-Value数据类型的Transfromation算子

- 输入分区与输出分区一对一
  - mapValues算子
- 对单个RDD或两个RDD聚集

    - combineByKey算子
    - reduceByKey算子
    - partitionBy算子
    - Cogroup算子
- 连接
    - join算子
    - leftOutJoin 和 rightOutJoin算子

3. Action算子

- 无输出
    - foreach算子
- HDFS算子
    - saveAsTextFile算子
    - saveAsObjectFile算子
- Scala集合和数据类型
    - collect算子
    - collectAsMap算子
    - reduceByKeyLocally算子
    - lookup算子
    - count算子
    - top算子
    - reduce算子
    - fold算子
    - aggregate算子
    - countByValue
    - countByKey

14. spark on yarn 模式下的 cluster模式和 client模式有什么区别

1. yarn-cluster 适用于生产环境。而 yarn-client 适用于交互和调试，也就是希望快速地看到 application 的输出.
2. yarn-cluster 和 yarn-client 模式的区别其实就是 **Application Master 进程**的区别，yarn-cluster 模式下，driver 运行在 AM(Application Master)中，它负责向 YARN 申请资源，并监督作业的运行状况。当用户提交了作业之后，就可以关掉 Client，作业会继续在 YARN 上运行。然而 yarn-cluster 模式不适合运行交互类型的作业。而 yarn-client 模式下，Application Master 仅仅向 YARN 请求 executor，Client 会和请求的 container 通信来调度他们工作，也就是说 Client 不能离开。

注：资料来源于网络。