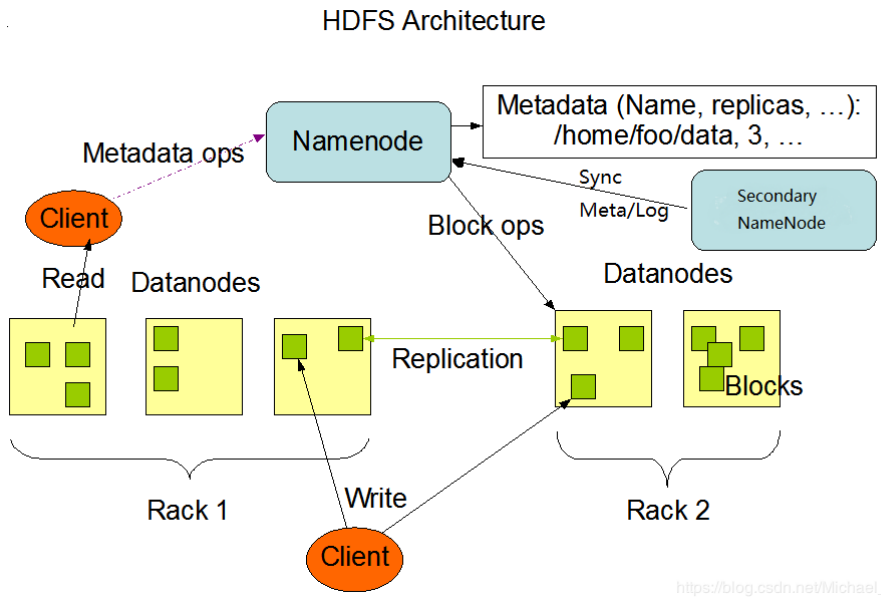
第二节：HDFS的体系结构

HDFS 采用的是master/slaves这种主从的结构模型来管理数据，这种结构模型主要由四个部分组成，分别是Client(客户端)、Namenode(名称节点)、Datanode(数据节点)和SecondaryNameNode。真正的一个HDFS集群包括一个Namenode和若干数目的Datanode。Namenode是一个中心服务器，负责管理文件系统的命名空间 (Namespace)及客户端对文件的访问。集群中的Datanode一般是一个节点运行一个DataNode进程，负责管理客户端的读写请求，在Namenode的统一调度下进行数据块的创建、删除和复制等操作。数据块实际上都是保存在Datanode本地的Linux文件系统中的。每个Datanode会定期的向Namenode发送数据，报告自己的状态(我们称之为心跳机制)。没有按时发送心跳信息的Datanode会被Namenode标记为“宕机”，不会再给他分配任何I/O请求。
用户在使用Client进行I/O操作时,仍然可以像使用普通文件系统那样，使用文件名去存储和访问文件，只不过，在HDFS内部，一个文件会被切分成若干个数据块，然后被分布存储在若干个Datanode上。

比如，用户在Client上需要访问一个文件时，HDFS的实际工作流程如此：客户端先把文件名发送给Namenode，Namenode根据文件名找到对应的数据块信息及其每个数据块所在的Datanode位置，然后把这些信息发送给客户端。之后，客户端就直接与这些Datanode进行通信，来获取数据（这个过程，Namenode并不参与数据块的传输）。这种设计方式，实现了并发访问，大大提高了数据的访问速度。

HDFS集群中只有唯一的一个Namenode,负责所有元数据的管理工作。这种方式保证了Datanode不会脱离Namenode的控制，同时，用户数据也永远不会经过Namenode，大大减轻了Namenode的工作负担，使之更方便管理工作。通常在部署集群中，我们要选择一台性能较好的机器来作为Namenode。当然，一台机器上也可以运行多个Datanode，甚至Namenode和Datanode也可以在一台机器上，只不过实际部署中，通常不会这么做的



1. 总结各个部分的作用
    1. NameNode:
        - 主节点只有一个（HA除外）
        - 管理HDFS的命名空间
        - 在内存中维护数据块的映射信息
        - 配置副本策略
        - 处理客户端的访问请求
    2. DateNode：
        - 存储真正的数据(块进行存储)
        - 执行数据块的读写操作
        - 心跳机制

3. SecondaryNameNode
    - 帮助NameNode合并fsimage和edits文件
    - 不能实时同步，不能作为热备份节点
4. Client:
    - HDFS实际上提供了各种语言操作HDFS的接口。
    - 与NameNode进行交互，获取文件的存储位置（读/写两种操作）
    - 与DataNode进行交互，写入数据，或者读取数据
    - 上传时分块进行存储，读取时分片进行读取

2. FsImage和EditLog的介绍

    命名空间指的就是文件系统树及整棵树内的所有文件和目录的元数据，每个Namenode只能管理唯一的一命名空间。HDFS暂不支持软链接和硬连接。Namenode会在内存里维护文件系统的元数据，同时还使用fsimage和editlog两个文件来辅助管理元数据，并持久化到本地磁盘上。

    fsimage：命名空间镜像文件，它是文件系统元数据的一个完整的永久检查点，内部维护的是最近一次检查点的文件系统树和整棵树内部的所有文件和目录的元数据，如修改时间，访问时间，访问权限，副本数据，块大小，文件的块列表信息等等。

    editlog：编辑日志文件，当hdfs文件系统发生打开、关闭、创建、删除、重命名等操作产生的信息除了在保存在内存中外，还会持久化到编辑日志文件。比如上传一个文件后，日志文件里记录的有这次事务的txid,文件的inodeid,数据块的副本数，数据块的id，数据块大小，访问时间，修改时间等

第三节：HDFS的设计思想

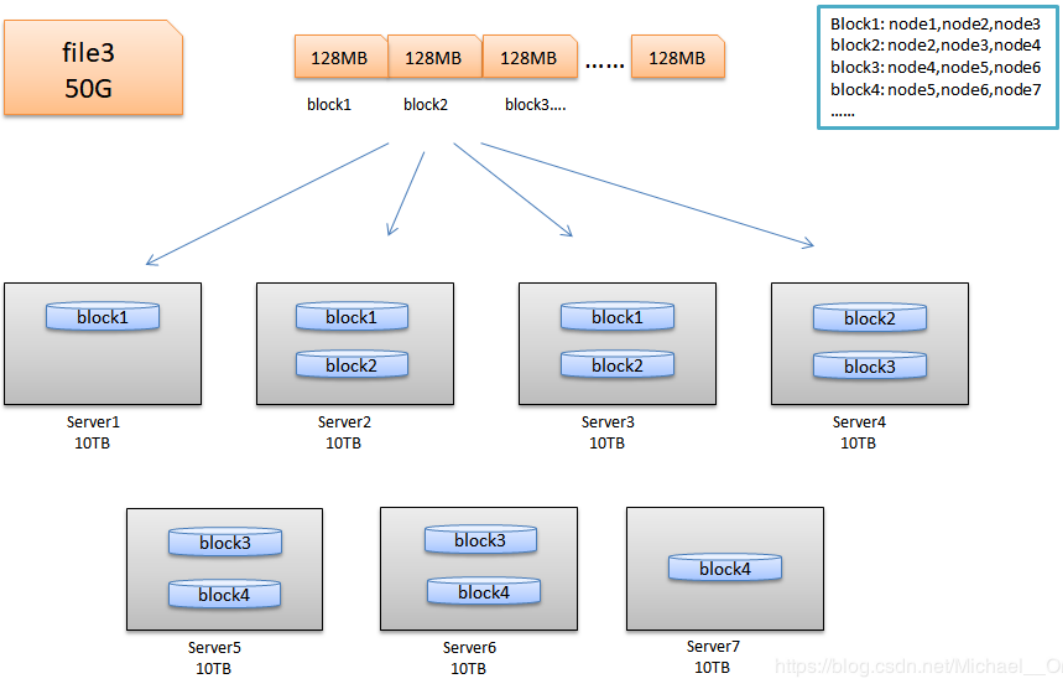
现在想象一下这种情况：有四个文件 0.5TB的file1，1.2TB的file2，50GB的file3，100GB的file4；有7个服务器，每个服务器上有10个1TB的硬盘。

在存储方式上，我们可以将这四个文件存储在同一个服务器上（当然大于1TB的文件需要切分），我们需要使用一个文件来记录这种存储的映射关系吧。用户是可以通过这种映射关系来找到节点硬盘相应的文件的。那么缺点也就暴露了出来：

第一、负载不均衡。因为文件大小不一致，势必会导致有的节点磁盘的利用率高，有的节点磁盘利用率低。

第二、网络瓶颈问题。一个过大的文件存储在一个节点磁盘上，当有并行处理时，每个线程都需要从这个节点磁盘上读取这个文件的内容，那么就会出现网络瓶颈，不利于分布式的数据处理。

我们来看看HDFS的设计思想：以下图为例，来进行解释。



1. HDFS的块的概念

   HDFS同样引入了块(Block)的概念，块是HDFS系统当中的最小存储单位，在hadoop2.0中默认大小为128MB。在HDFS上的文件会被拆分成多个块，每个块作为独立的单元进行存储。多个块存放在不同的DataNode上，整个过程中 HDFS系统会保证一个块存储在一个数据节点上 。但值得注意的是 如果某文件大小或者文件的最后一个块没有到达128M，则不会占据整个块空间 。

   当然块大小可以在配置文件中hdfs-default.xml中进行修改(此值可以修改)

   ```
   默认块大小，以字节为单位。可以使用以下后缀(不区分大小写):k，m，g，t，p，e以重新指定大小(例如128k，512m，1g等)
   每个文件的最大块数，由写入时的Namenode执行。这可以防止创建会降低性能的超大文件
   ```

   HDFS中的NameNode会记录文件的各个块都存放在哪个dataNode上，这些信息一般也称为元信息(MetaInfo) 。元信息的存储位置一般由dfs.namenode.name.dir来指定。

   而datanode是真实存储文件块的节点，块在datanode的位置一般由dfs.datanode.data.dir来指定

   HDFS上的块为什么远远大与传统文件系统，是有原因的。目的是为了最小化寻址开销时间。

   ```
   在I/O开销中，机械硬盘的寻址时间时最耗时的部分，一旦找到第一条记录，剩下的顺序读取效率是非常高的，因此以块为单位读写数据，可以把磁盘寻
   道时间分摊到大量数据中。
   HDFS寻址开销不仅包括磁盘寻道开销，还包括数据库的定位开销，当客户端需要访问一个文件时，首先从名称节点获取组成这个文件的数据块的位置列表，
   然后根据位置列表获取实际存储各个数据块的数据节点的位置，最后，数据节点根据数据块信息在本地Linux文件系统中找到对应的文件，并把数据返
   回给客户端，设计一个比较大的块，可以把寻址开销分摊到较多的数据中，相对降低了单位数据的寻址开销

   举个例子：    块大小为128MB，默认传输效率100M/s ，寻址时间为10ms，那么寻址时间只占传输时间的1%左右
   ```

第四节：HDFS的设计目标和优缺点

一. 设计目标

1. 大规模数据集

   HDFS用来处理很大的数据集。HDFS上的文件，大小一般都在GB至TB。因此同时，HDFS应该能提供整体较高的数据传输带宽，能在一个集群里扩展到数百个节点。一个单一的HDFS实例应该能支撑千万计的文件。目前在实际应用中，HDFS已经能用来存储管理PB级的数据了。

2. 硬件错误

   我们应该知道，硬件组件发生故障是常态，而非异常情况。HDFS可能由成百上千的服务器组成，每一个服务器都是廉价通用的普通硬件，任何一个组件都有可能一直失效，因此错误检测和快速、自动恢复是HDFS的核心架构目标，同时能够通过自身持续的状态监控快速检测冗余并恢复失效的组件。

3. 流式数据访问

   流式数据，特点就是，像流水一样，不是一次过来而是一点一点“流”过来，而处理流式数据也是一点一点处理。

   HDFS的设计要求是：能够高速率、大批量的处理数据，更多地响应"一次写入、多次读取"这样的任务。在HDFS上一个数据集，会被复制分发到不同的存储节点中。而各式各样的分析任务多数情况下，都会涉及数据集中的大部分数据。为了提高数据的吞吐量，Hadoop放宽了POSIX的约束，使用流式访问来进行高效的分析工作

4. 简化一致性模型

HDFS应用需要一个“一次写入多次读取”的文件访问模型。一个文件经过创建、写入和关闭之后就不需要改变了。这一假设简化了数据一致性问题，并且使高吞吐量的数据访问成为可能。MapReduce应用或网络爬虫应用都非常适合这个模型。目前还有计划在将来扩充这个模型，使之支持文件的附加写操作。

5. 移动计算代价比移动数据代价低

   一个应用请求的计算，离它操作的数据越近就越高效，这在数据达到海量级别的时候更是如此。将计算移动到数据附近，比之将数据移动到应用所在之处显然更好，HDFS提供给应用这样的接口。

6. 可移植性

   HDFS在设计时就考虑到平台的可移植性，这种特性方便了HDFS作为大规模数据应用平台的推广。

二. 总结HDFS的优缺点

- 优点
  - 高容错性：数据自动保存多个副本，副本丢失后，会自动恢复。
  - 适合大数据集：GB、TB、甚至PB级数据、百万规模以上的文件数量，1000以上节点规模。
  - 流式数据访问： 一次性写入，多次读取；保证数据一致性
  - 构建成本低：可以构建在廉价机器上。
- 缺点
  - 不适合做低延迟数据访问：HDFS的设计目标有一点时：处理大型数据集，高吞吐率。这一点势必要以高延迟为代价的。因此HDFS不适合处理用户要求的毫秒级的低延迟应用请求
  - 不适合小文件存取：一个是大量小文件需要消耗大量的寻址时间，违反了HDFS的尽可能减少寻址时间比例的设计目标。第二个是内存有限，一个block元数据大内存消耗大约为150个字节，存储一亿个block和存储一亿个小文件都会消耗20G内存。因此相对来说，大文件更省内存
  - 不适合并发写入，文件随机修改：HDFS上的文件只能拥有一个写者，仅仅支持append操作。不支持多用户对同一个文件的写操作，以及在文件任意位置进行修改

第五节：HDFS的工作机制

1. HDFS的开机过程

Namenode在启动时，会将FsImage的内容加载到内存当中，然后执行EditLog文件中的各项操作，使得内存中的元数据保持最新。这个操作完成以后，就会创建一个新的FsImage文件和一个空的EditLog文件。名称节点启动成功并进入正常运行状态以后，HDFS中的更新操作都被写到EditLog，而不是直接写入FsImage，这是因为对于分布式文件系统而言，FsImage文件通常都很庞大，如果所有的更新操作都直接往FsImage文件中添加，那么系统就会变得非常缓慢。相对而言，EditLog通常都要远远小于FsImage，更新操作写入到EditLog是非常高效的。名称节点在启动的过程中处于“安全模式”，只能对外提供读操作，无法提供写操作。在启动结束后，系统就会退出安全模式，进入正常运行状态，对外提供写操作。

2. 安全模式

Namenode启动时，首先要加载fsimage文件到内存，并逐条执行editlog文件里的事务操作，在这个期间一但在内存中成功建立文件系统元数据的映像，就会新创建一个fsimage文件(该操作不需要SecondaryNamenode)和一个空的editlog文件。在这个过程中，namenode是运行在安全模式下的，Namenode的文件系统对于客户端来说是只读的，文件修改操作如写，删除，重命名等均会失败。

Namenode不会存储数据块的位置信息，因此Namenode在启动后，还会等待接受Datanode的blockreport(数据块的状态报告)。严格来说，只有接受到状态报告后，客户端的读操作才能成功完成。

PS:启动一个刚刚格式化完的集群时，HDFS还没有任何操作呢，因此Namenode不会进入安全模式。

管理员可以随时让Namenode进入或离开安全模式，这项功能在维护和升级集群时非常关键

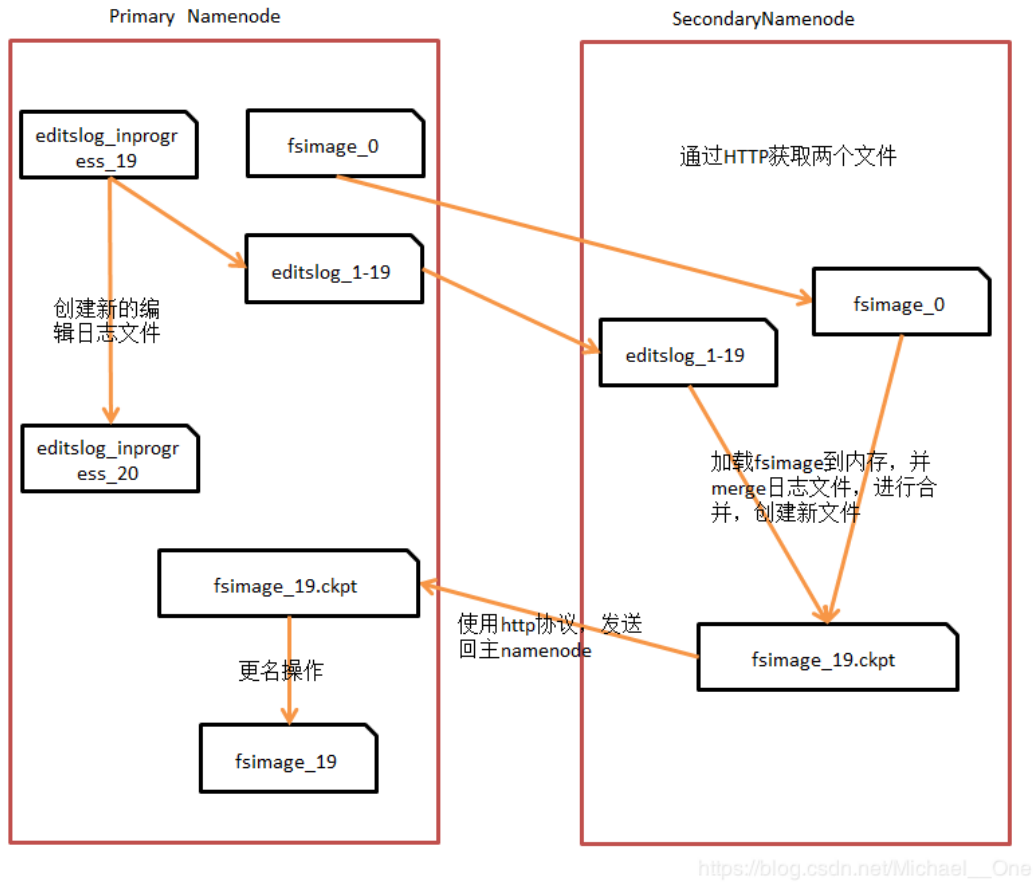
将下面的属性的值设置为大于1，将永远不会离开安全模式

有时，在安全模式下，用户想要执行某条命令，特别是在脚本中，此时可以先让安全模式进入等待状态

3. 检查点机制

SecondaryNamenode，是HDFS集群中的重要组成部分，它可以辅助Namenode进行fsimage和editlog的合并工作，减小editlog文件大小，以便缩短下次Namenode的重启时间，能尽快退出安全模式。

两个文件的合并周期，称之为检查点机制（checkpoint），是可以通过hdfs-default.xml配置文件进行修改的：



1）、SecondaryNamenode请求Namenode停止使用正在编辑的editlog文件，Namenode会创建新的editlog文件(小了吧)，同时更新seed_txid文件。
2）、SecondaryNamenode通过HTTP协议获取Namenode上的fsimage和editlog文件。
3）、SecondaryNamenode将fsimage读进内存当中，并逐步分析editlog文件里的数据，进行合并操作，然后写入新文件fsimage_x.ckpt文件中。
4）、SecondaryNamenode将新文件fsimage_x.ckpt通过HTTP协议发送回Namenode。
5）、Namenode再进行更名操作。

4. 存储策略

client 向 Active NN 发送写请求时，NN为这些数据分配DN地址，HDFS文件块副本的放置对于系统整体的可靠性和性能有关键性影响。一个简单但非优化的副本放置策略是，把副本分别放在不同机架，甚至不同IDC，这样可以防止整个机架、甚至整个IDC崩溃带来的错误，但是这样文件写必须在多个机架之间、甚至IDC之间传输，增加了副本写的代价，是否有较优的方案来解决这个问题呢?

hdfs 在缺省配置下副本数是3个，通常的策略是：

第一个副本放在和Client相同机架的Node里（如果Client不在集群范围，第一个Node是随机选取不太满或者不太忙的Node）
第二个副本放在与第一个Node不同的机架中的Node
第三个副本放在与第二个Node所在机架里不同的Node

5. 心跳机制

1. hdfs是master/slave结构，master包括namenode和resourcemanager，slave包括datanode和nodemanager
2. master启动时会开启一个IPC服务，等待slave连接
3. slave启动后，会主动连接IPC服务，并且每隔3秒链接一次，这个时间是可以调整的，设置heartbeat，这个每隔一段时间连接一次的机制，称为心跳机制。Slave通过心跳给master汇报自己信息，master通过心跳下达命令。
4. Namenode通过心跳得知datanode状态。Resourcemanager通过心跳得知nodemanager状态
5. 当master长时间没有收到slave信息时，就认为slave挂掉了。

   超长时间计算：默认为10分钟30秒
   而默认的 heartbeat.recheck.interval 大小为 5 分钟，dfs.heartbeat.interval 默认的大小为 3 秒。
   Recheck的时间单位为毫秒　heartbeat的时间单位为秒

   计算公式为2*recheck+10*heartbeat

6. 集群的动态上下线

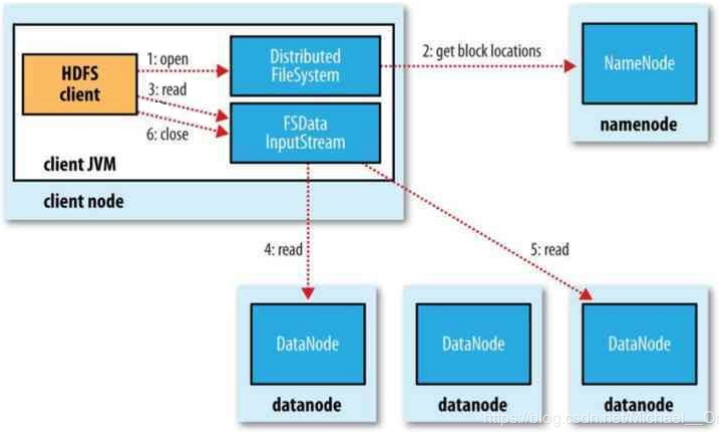
1. 下线步骤：
2. 将要下线的机器加入到dfs.hosts.exclude 指定的文件中，（使用主机名， ip基本不靠谱）， 然后刷新配置Hadoop dfsadmin -refreshNodes。
3. 通过hadoop dfsadmin -report或者web界面， 可以看到， 该datanode状态转化为Decommission In Progress。
4. 当decommission进程完成数据移动， datanode状态会转变为Decommissioned， 然后datanode会自动停止datanode进程。 然后你可以看见dead nodes下多了一个你想要下线的节点。
5. 然后删除include 和 exclude 中该节点的hosts， 重新刷新hadoop dfsadmin -refreshNodes。
6. 最后别忘了删除slaves中该节点的配置， 防止下次整个集群重启时， 该节点不能通过namenode自动启动。

注意： 当你下线一个datanode节点， 有可能该节点长时间处于Decommission In Progress 状态， 一直不能转变成Decommissioned。 请你用hadoop fsck / 检查下是否有些块少于指定的块数， 特别注意那些mapreduce的临时文件。 将这些删除， 并且从垃圾箱移除， 该节点就可以顺利下线， 这是我想到的解决办法。

2. 上线步骤：
3. 保证将要上线的机器不存在与dfs.hosts.exclude所对应的文件中， 并且存在于 dfs.hosts 所对应的文件中。
4. 在namenode上刷新配置： hadoop dfsadmin -refreshNodes。
5. 在要上线的节点重启 datanode， hadoop-daemon.sh start datanode.
6. 通过hadoop dfsadmin -report 或者web界面， 可以看到， 节点已经上线。
7. 还是老话最后别忘了修改slave

第二节：HDFS的读写流程

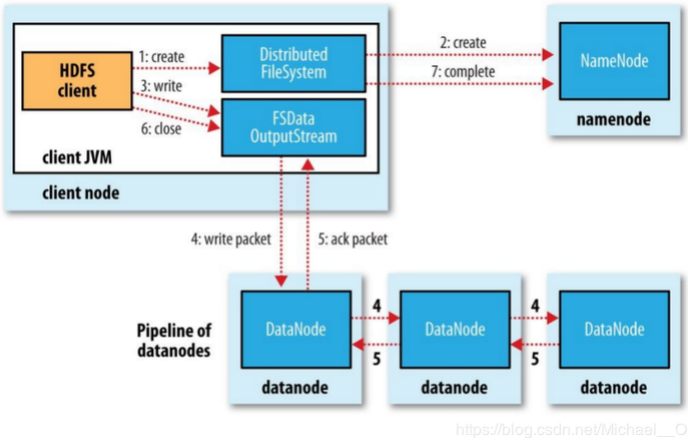
1. hdfs的读取流程



1. 客户端通过调用FileSystem对象的open()方法来打开希望读取的文件，对于HDFS来说，这个对象是DistributedFileSystem，它通过使用远程过程调用(RPC)来调用namenode,以确定文件起始块的位置
2. 对于每一个块,NameNode返回存有该块副本的DataNode地址,并根据距离客户端的远近来排序。
3. DistributedFileSystem实例会返回一个FSDataInputStream对象（支持文件定位功能）给客户端以便读取数据，接着客户端对这个输入流调用read()方法
4. FSDataInputStream随即连接距离最近的文件中第一个块所在的DataNode,通过对数据流反复调用read()方法，可以将数据从DataNode传输到客户端
5. 当读取到块的末端时，FSInputStream关闭与该DataNode的连接，然后寻找下一个块的最佳DataNode
6. 客户端从流中读取数据时，块是按照打开FSInputStream与DataNode的新建连接的顺序读取的。它也会根据需要询问NameNode来检索下一批数据块的DataNode的位置。一旦客户端完成读取，就对FSInputStream调用close方法

   在读取数据的时候，如果FSInputStream与DataNode通信时遇到错误，会尝试从这个块的最近的DataNode读取数据，并且记住那个故障的DataNode,保证后续不会反复读取该节点上后续的块。FInputStream也会通过校验和确认从DataNode发来的数据是否完整。如果发现有损坏的块，FSInputStream会从其他的块读取副本，并且将损坏的块通知给NameNode

2. hdfs的写流程



1. 客户端通过对DistributedFileSystem对象调用create()方法来新建文件
2. DistributedFileSystem对namenode创建一个RPC调用，在文件系统的命名空间中新建一个文件，此时该文件中还没有相应的数据块

3. namenode执行各种不同的检查，以确保这个文件不存在以及客户端有新建该文件的权限。如果检查通过，namenode就会为创建新文件记录一条事务记录(否则，文件创建失败并向客户端抛出一个IOException异常)。DistributedFileSystem向客户端返回一个FSDataOuputStream对象，由此客户端可以开始写入数据，
4. 在客户端写入数据时，FSOutputStream将它分成一个个的数据包(packet)，并写入一个内部队列，这个队列称为"数据队列"（data queue）。DataStreamer线程负责处理数据队列，它的责任是挑选出合适存储数据复本的一组datanode，并以此来要求namenode分配新的数据块。这一组datanode将构成一个管道，以默认复本3个为例，所以该管道中有3个节点.DataStreamer将数据包流式传输到管道中第一个datanode，该datanode存储数据包并将它发送到管道中的第2个datanode，同样，第2个datanode存储该数据包并且发送给管道中的第三个datanode。DataStreamer在将一个个packet流式传输到第一个Datanode节点后，还会将此packet从数据队列移动到另一个队列确认队列(ack queue)中。
5. datanode写入数据成功之后，会为ResponseProcessor线程发送一个写入成功的信息回执，当收到管道中所有的datanode确认信息后，ResponseProcessoer线程会将该数据包从确认队列中删除

   如果任何datanode在写入数据期间发生故障，则执行以下操作：

   1. 首先关闭管道，把确认队列中的所有数据包都添加回数据队列的最前端，以确保故障节点下游datanode不会漏掉任何一个数据包
   2. 为存储在另一正常datanode的当前数据块制定一个新标识，并将该标识传送给namenode，以便故障datanode在恢复后可以删除存储的部分数据块
   3. 从管道中删除故障datanode，基于两个正常datanode构建一条新管道，余下数据块写入管道中正常的datanode
   4. namenode注意到块复本不足时，会在一个新的Datanode节点上创建一个新的复本。

   在一个块被写入期间可能会有多个datanode同时发生故障，但概率非常低。只要写入了dfs.namenode.replication.min的复本数（默认1），写操作就会成功，并且这个块可以在集群中异步复制，直到达到其目标复本数dfs.replication的数量（默认3）

机架感知

如何想让HDFS知道自己的网络拓扑情况，那就是另外一个配置策略了，即机架感知。默认情况下，机架感知策略是关闭的。 需要进行对net.topology.script.file.name进行设置。如果不设置，namenode就会将datanode注册属于/default-rack机架。

使用了机架感知策略的副本存放：

```
第一个副本在client所处的节点上。如果客户端在集群外，随机选一个。
第二个副本与第一个副本不相同机架，随机一个节点进行存储
第三个副本与第二个副本相同机架，不同节点。
```

第三节：Zookeeper的介绍

1. Zookeeper的简介

1. zookeeper是一个为分布式应用程序提供的一个分布式开源协调服务框架。是Google的Chubby的一个开源实现，是Hadoop和Hbase的重要组件。主要用于解决分布式集群中应用系统的一致性问题。
2. 提供了基于类似Unix系统的目录节点树方式的数据存储。
3. 可用于维护和监控存储的数据的状态的变化，通过监控这些数据状态的变化，从而达到基于数据的集群管理
4. 提供了一组原语(机器指令)，提供了java和c语言的接口

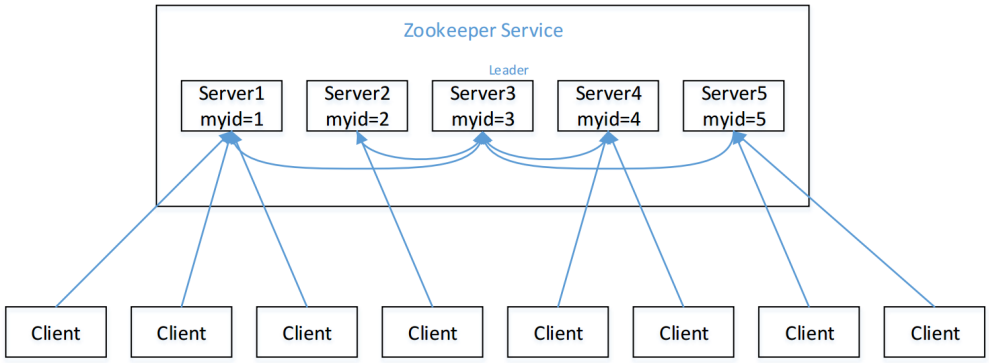
2. Zookeeper的特点

1. 也是一个分布式集群，一个领导者(leader),多个跟随者(follower).
2. 集群中只要有半数以上的节点存活，Zookeeper集群就能正常服务。
3. 全局数据一致性：每个server保存一份相同的数据副本，client无论连接到哪个server,数据都是一致的。
4. 更新请求按顺序进行：来自同一个client的更新请求按其发送顺序依次执行
5. 数据更新的原子性：一次数据的更新要么成功，要么失败
6. 数据的实时性：在一定时间范围内，client能读到最新数据。

3. zookeeper的应用场景

1. 统一配置管理、
2. 统一集群管理、
3. 服务器节点动态上下线感知、
4. 软负载均衡等

第四节：Zookeeper的工作原理



1. 选举机制

1. 基于节点在半数以上才能正常服务的要求，Zookeeper适合装在奇数台机器。
2. Zookeeper没有在配置文件中指定leader和follower，而是使用算法(Paxos)在内部通过选举机制来选择一个节点为leader，其他节点为follower。
3. 以一个简单的例子来说明整个选举的过程。假设有五台服务器组成的 zookeeper 集群，它们的 id 从 1-5，同时它们都是最新启动的，也就是没有历史数据，在存放数据量这一点上，都是一样的。假设这些服务器依序启动，来
   看看会发生什么。
   - 服务器 1 启动，此时只有它一台服务器启动了，它发出去的报没有任何响应，所以它的选举状态一直是 LOOKING 状态。
   - 服务器 2 启动，它与最开始启动的服务器 1 进行通信，互相交换自己的选举结果，由于两者都没有历史数据，所以 id 值较大的服务器 2 胜出，但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是 3)，所以服务器 1、 2 还是继续保持 LOOKING 状态。
   - 服务器 3 启动，根据前面的理论分析，服务器 3 成为服务器 1、 2、 3 中的老大，而与上面不同的是，此时有三台服务器选举了它，所以它成为了这次选举的 leader。
   - 服务器 4 启动，根据前面的分析，理论上服务器 4 应该是服务器 1、 2、 3、 4 中最大的，但是由于前面已经有半数以上的服务器选举了服务器 3，所以它只能接收当小弟的命了。
   - 服务器 5 启动，同 4 一样当小弟
4. 选举机制中的概念
   - serverid:服务器id
     比如有三台服务器，编号分别为1，2，3。编号越大在选择算法中的权重越大

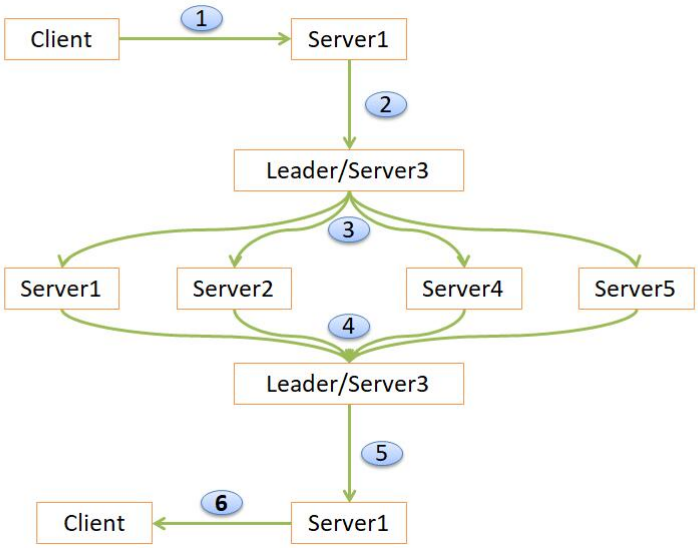
- zxid:数据id
  服务器中存放的最大数据ID。值越大说明数据越新，在选举算法中的权重越大
- Epoch:逻辑时钟
  也可以称之为每个服务器参加投票的次数。同一轮投票过程中的逻辑次数
- Server状态：选举状态
  LOOKING：竞选状态
  FOLLOWING:随从状态，同步leader状态，参与选票
  OBSERVING:观察状态，同步leader状态，不参与选票
  LEADER：领导者状态

2. 写数据流程

1. Client向Zookeeper的server1上写数据，发送一个写请求
2. 如果server1不是leader,那么server1会把请求进一步转发给leader。
3. 这个leader会将写请求广播给所有server。
4. 各个Server写成功后就会通知leader。
5. 当leader收到半数以上的server写成功的通知，就说明数据写成功了。写成功后，leader会告诉server1数据写成功了。
6. server1会进一步通知Client数据写成功了。这时就认为整个写操作成功。

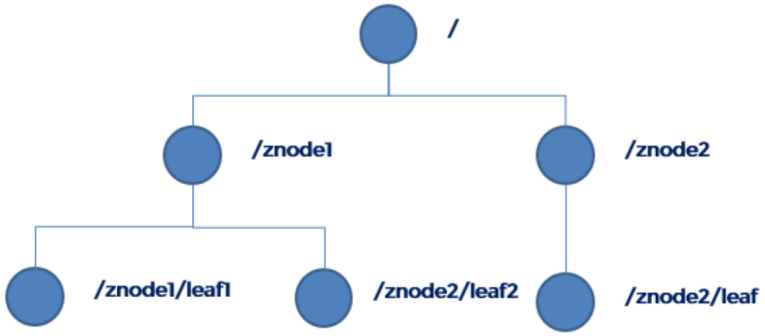
参考下图：

Zookeeper写数据原理图解



3. 数据模型

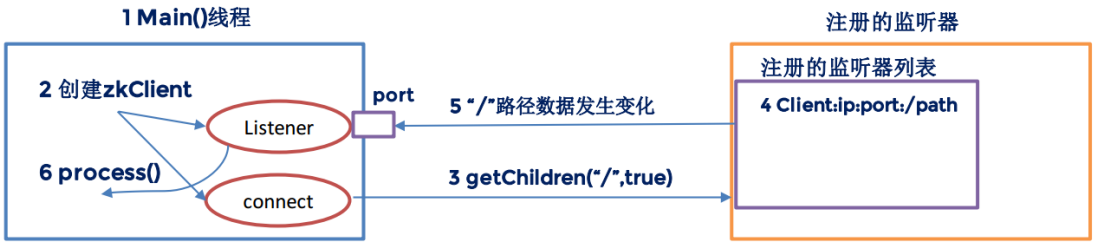
Zookeeper的数据模型采用的与Unix文件系统类似的层次化的树形结构。我们可以将其理解为一个具有高可用特征的文件系统。这个文件系统中没有文件和目录，而是统一使用"节点"(node)的概念，称之为znode。znode既可以作为保存数据的容器(如同文件),也可以作为保存其他znode的容器(如同目录)。所有的znode构成了一个层次化的命名空间。



数据结构图

- Zookeeper 被设计用来实现协调服务（这类服务通常使用小数据文件），而不是用于大容量数据存储，因此一个znode能存储的数据被限制在1MB以内，
- 每个znode都可以通过其路径唯一标识。

4. Zookeeper的监听原理

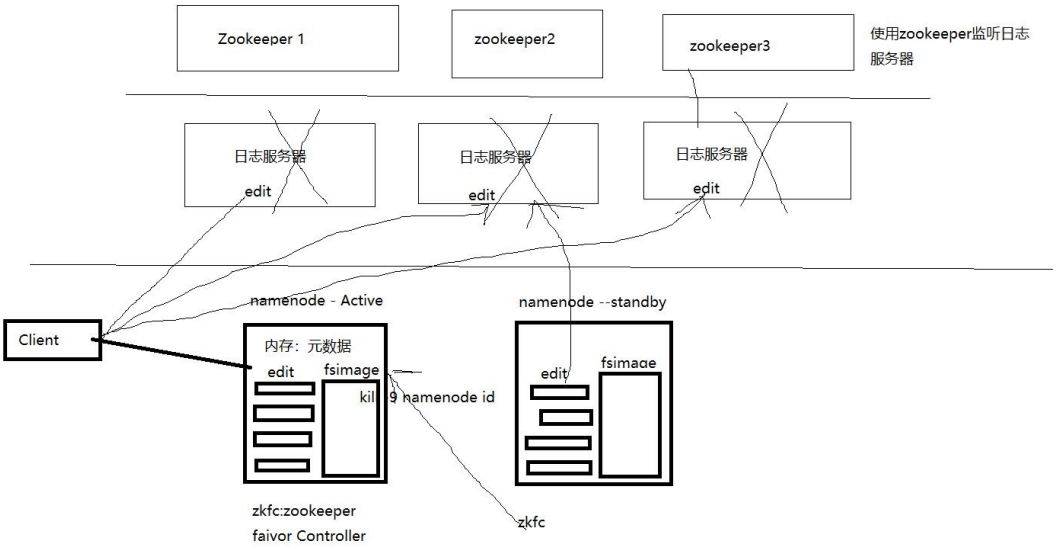


1. 图解：
    1. 首先要有一个main()线程
    2. 在main线程中创建Zookeeper客户端， 这时就会创建两个线程， 一个负责网络连接通信(connet),一个负责监听(listener)。
    3. 通过connect线程将注册的监听事件发送给Zookeeper。
    4. 在Zookeeper的注册监听器列表中将注册的监听事件添加到列表中。
    5. Zookeeper监听到有数据或路径变化， 就会将这个消息发送给listener线程。
    6. listener线程内部调用了process（） 方法。

第一节：HDFS的高可用(HA)

1. HA的原理

1. 问题所在：
    1. 计划外的事件（比如:单点故障）
    2. 计划内的事件 (比如：namenode上的硬件或软件升级)
2. 解决办法：
    1. 使用两个Namenode。一个是正在工作的namenode，可以称之为Active节点。另外一个namenode充当备份节点,称之为Standby节点。
    图解：



Zookeeper相关面试题汇总

1、请给zookeeper下一个定义。

ZooKeeper 是一个分布式应用程序协调服务，是一个为分布式应用提供一致性服务的软件

2、Zookeeper 中的容错机制，为什么是单数节点?

剩下的节点数必须大于宕掉的个数，也就是存活节点需要大于总数一半，zookeeper才可以继续使用。假设总结点为5，则需要3台存活，即使增加为6也是3台，也就是说只

有节点数为增加到奇数时最小存活节点才会增加，所以设置节点为奇数节约资源。

3、请阐述zookeeper的选举机制。

zk集群暂停对外服务。

1. 各节点会先选自己作为leader，然后将选票携带事务id:zxid发送出去
2. 各节点拿到选票后，先排除非本轮的票，然后比对自己的选票跟各个节点发来的选票，先比较zxid（越大说明数据越新），相同时比较myid，大的一方获胜，将票投给获

胜方，然后各自发回节点。（假如自己是1节点，3节点发来的票，比对后要么返回1要么返回3）

3. 投票后，各个节点会统计投票信息，判断如果有过半选票则认为选出了leader，更新自身状态为follow或leader，如果没有则一直重复2直到满足条件为止。
4. 选举出leader后，新节点或原leader节点宕机恢复后，会直接变为follow状态，不再进行选举。

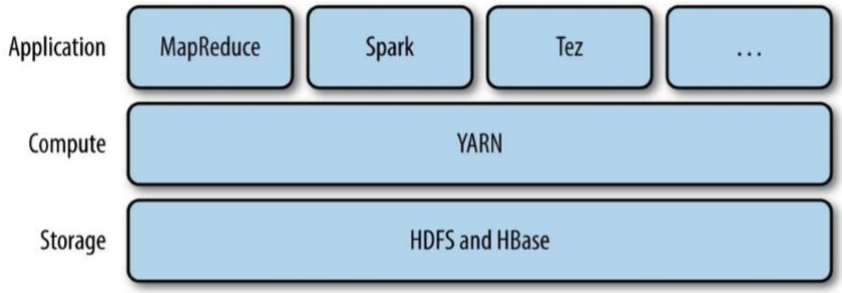
4、zookeeper中常用的命令有哪些?

ls、get、create

第一节：YARN的相关内容

1. yarn简介

1. 概述
   Apache YARN（Yet another Resource Negotiator的缩写）是Hadoop的集群资源管理系统。
   yarn被引入Hadoop 2,最初是为了改善MapReduce的实现，但是因为具有足够的通用性，同样可以支持其他的分布式计算模式



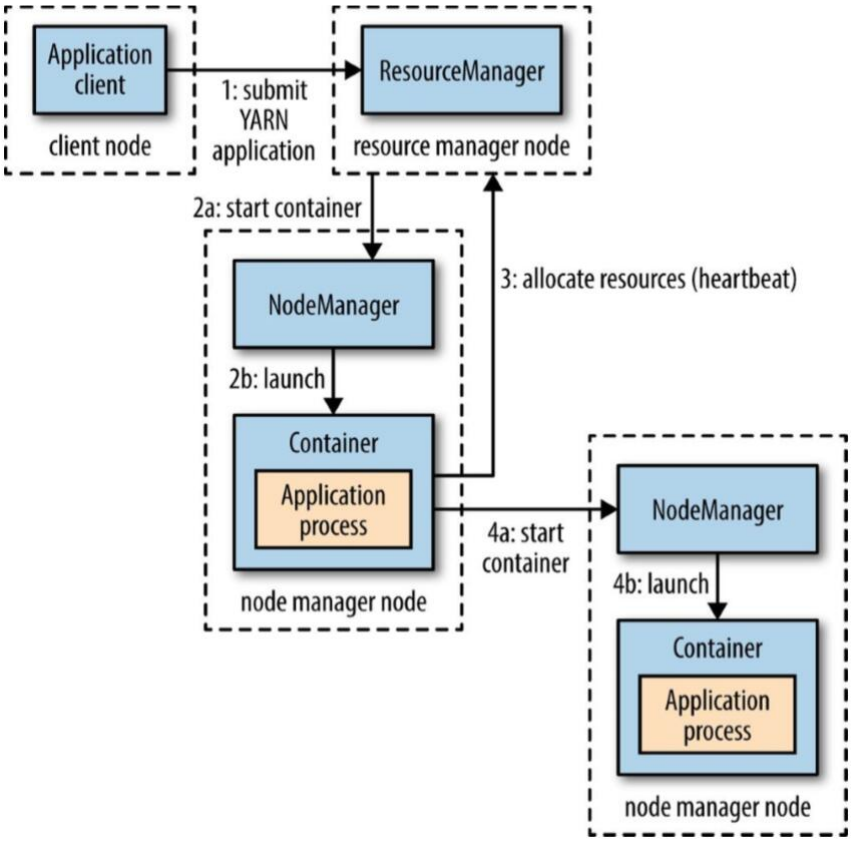
2. 思想

   yarn的基本思想是将资源管理和作业调度/监视功能划分为单独的守护进程。其思想是拥有一个全局ResourceManager (RM)和每个应用程序的ApplicationMaster (AM)。应用程序可以是单个作业，也可以是一组作业

   ResourceManager和NodeManager构成数据计算框架。ResourceManager是在系统中的所有应用程序之间仲裁资源的最终权威。NodeManager是每台机器的框架代理，负责监视容器的资源使用情况(cpu、内存、磁盘、网络)，并向ResourceManager/Scheduler报告相同的情况

   每个应用程序ApplicationMaster实际上是一个特定于框架的库，它的任务是与ResourceManager协商资源，并与NodeManager一起执行和监视任务

2. yarn的运行机制



1. 角色
    1. resource manager 资源管理器：YARN的守护进程，管理集群资源使用
    2. node manager 节点管理器：YARN的守护进程，启动和监控容器，运行在集群中所有节点上
    3. container 容器：执行job进程，容器有资源限制(内存，CPU等)，具体容器取决于YARN配置
2. 运行机制

    yarn通过两类长期运行的守护进程提供自己的核心服务：管理集群上资源使用的资源管理器（ResourceManager）,运行再集群中所有节点上且能够启动和监控容器（container）的节点管理器（node manager）。容器用于执行特定应用程序的进程，每个容器都有资源限制（内存，cpu等）。一个容器可以适应unix进程，也可以是一个linux cgroup,取决于yarn的配置。

    上图描述了yarn是如何运行一个应用的

    1. 首先，客户端联系resoucemanager,要求他运行一个application master进程。
    2. 然后，resourcemanager找到一个能够在容器中启动application master的节点管理器（步骤2a和2b）.
    3. application master通过心跳机制向resourcemanager请求更多的容器资源
    4. application master运行起来之后需要做什么依赖于客户端传递的应用
        - 简单地运算后直接返回结果给客户端
        - 请求更多容器进行分布式计算

第二节: MR的概念

1. MR简介

Hadoop MapReduce是对google提出的分布式并行编程模型MapReduce论文的开源实现，以可靠，容错的方式运行在分布式文件系统HDFS上的并行处理数据的编程模型。MapReduce的优势在于处理大规模数据集（1TB以上）。

> - 在过去的很长一段时间里，CPU的性能都会遵循”摩尔定律“，在性能上每隔18个月左右就是提高一倍。那个时候，不需要对程序做任何改变，仅仅通过使用更优秀的CPU，就可以进行

> 性能提升。但是现在，在CPU性能提升的道路上，人类已经到达了制作工艺的瓶颈，因此，我们不能再把希望寄托在性能更高的CPU身上了。
> - 现在这个时候，大规模数据存储在分布式文件系统上，人们也开始采用分布式并行编程来提高程序的性能。分布式程序运行在大规模计算机集群上，集群是大量的廉价服务器，可以并行执行大规模数据处理任务，这样就获得了海量的计算能力
> - 分布式并行编程比传统的程序有明显的区别，它运行在大量计算机构成的集群上，可以充分利用集群的并行处理能力；同时，通过向集群中增加新的计算节点，就可以很容易的实现集群计算能力的扩展。

MapReduce 作业通常将输入数据集拆分为独立的块，这些块由map任务以完全并行的方式处理。框架对map的输出进行排序，然后输入到reduce任务。通常，作业的输入和输出都存储在文件系统中。该框架负责调度任务，监视任务并重新执行失败的任务。

通常，计算节点和存储节点是相同的，即MapReduce框架和Hadoop分布式文件系统在同一组节点上运行。此配置允许框架有效地在已存在数据的节点上调度任务，从而在集群中产生非常高的聚合带宽。

MapReduce框架由单个主ResourceManager，每个集群节点一个从NodeManager和每个应用程序的MRAppMaster组成（参见YARN简介）。

最低限度，应用程序通过适当的接口和/或抽象类的实现来指定输入/输出位置并提供映射和减少功能。这些和其他作业参数包括作业配置。

然后，Hadoop 作业客户端将作业（jar /可执行文件等）和配置提交给ResourceManager，然后ResourceManager负责将软件/配置分发给从站，调度任务并监视它们，为作业提供状态和诊断信息 -客户。

虽然Hadoop框架是用Java™实现的，但MapReduce应用程序不需要用Java编写。

2. 原型解析

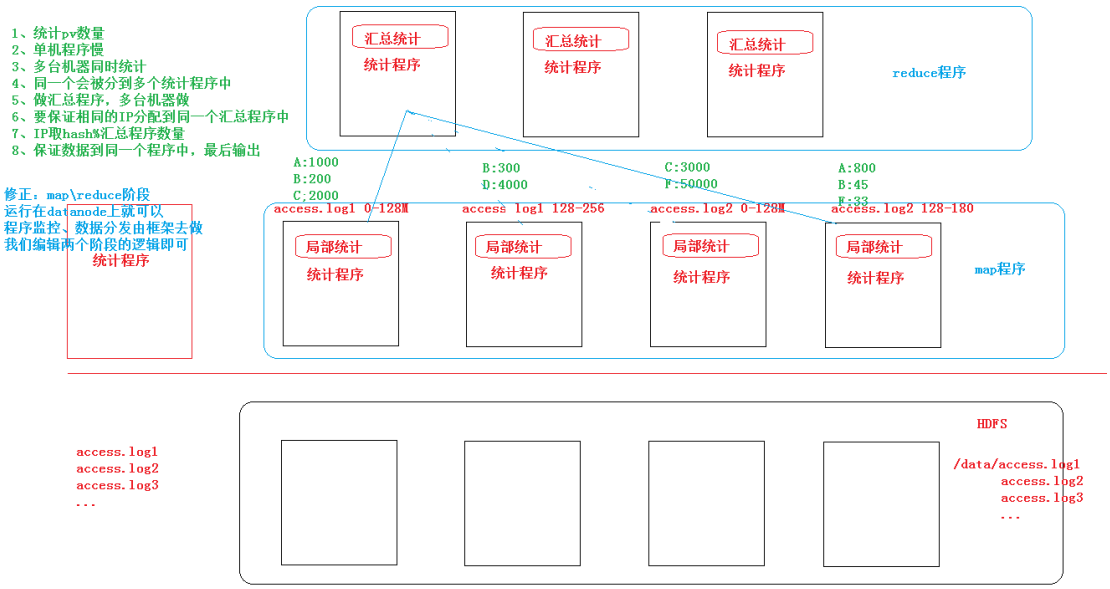
Hadoop的MapReduce核心技术起源于谷歌在2004年发表的关于MapReduce系统的论文介绍。论文中有这么一句话：Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages。这句话提到了MapReduce思想来源，大致意思是，MapReduce的灵感来源于函数式语言（比如Lisp）中的内置函数map（映射）和reduce（规约）。

简单来说，在函数式语言里，map表示对一个列表（List）中的每个元素做计算，reduce表示对一个列表中的每个元素做迭代计算。它们具体的计算是通过传入的函数来实现的，map和reduce提供的是计算的框架。我们想一下，reduce既然能做迭代计算，那就表示列表中的元素是相关的（比如我想对列表中的所有元素做相加求和，那么列表中至少都应该是数值吧）。而map是对列表中每个元素做单独处理的，这表示列表中可以是杂乱无章的数据。这样看来，就有点联系了。在MapReduce里，Map处理的是原始数据，自然是杂乱无章的，每条数据之间互相没有关系；到了Reduce阶段，数据是以key后面跟着若干个value来组织的，这些value有相关性，至少它们都在一个key下面，于是就符合函数式语言里map和reduce的基本思想了。

MapReduce任务过程分为两个处理阶段：map阶段和reduce阶段。每阶段都以键值对作为输入和输出，其类型由程序员来选择。程序员需要为每个阶段写一个函数，分别是map函数和reduce函数。

在执行MapReduce任务时，一个大规模的数据集会被划分成许多独立的等长的小数据块，称为输入分片（input split）或简称"分片"。Hadoop为每个输入分片分别构建一个map任务，并由该任务来运行用户自定义的map函数，从而处理分片中的每条记录。map任务处理后的结果会继续作为reduce任务的输入，最终由reduce任务输出最后结果，并写入分布式文件系统。

1. 案例分析

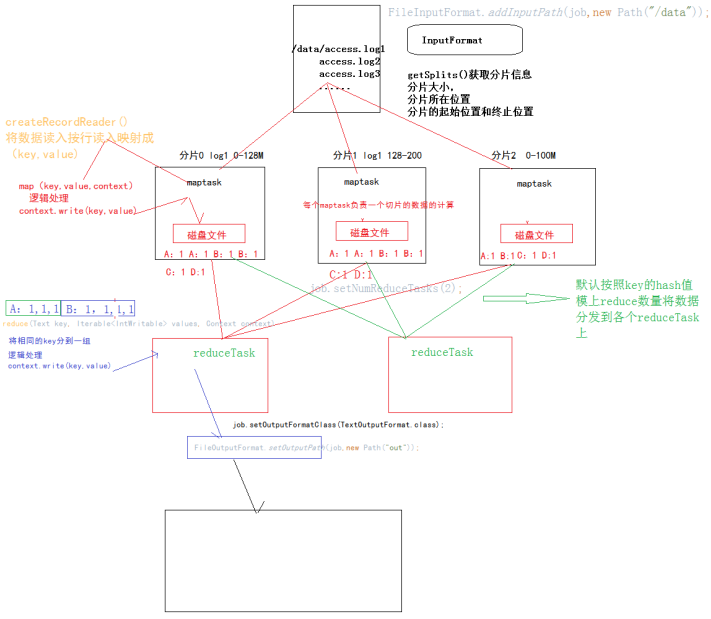


3. 核心思想

分而治之、移动计算不移动数据

MapReduce设计的一个理念是"计算向数据靠拢"（移动计算），而不是"数据向计算靠拢"（移动数据）。因为移动数据需要大量的网络传输开销，尤其是在大规模数据环境下，这种开销尤为惊人，所以移动计算要比移动数据更加经济。所以，在一个集群中，只要有可能，MapReduce框架就会将Map程序就近的在HDFS数据所在的节点上运行，即将计算节点和存储节点放在一起运行，从而减少节点间的数据移动开销。正因为这样，MapReduce可以并行的进行处理，解决计算效率问题。

第三节：wordcount案例



HDFS文件系统，多个文件/data/log1 log2 log3

程序提交时会先分片，分片信息包括文件路劲，分片起始位置，分片大小，分片数据所在块的信息，块的主机信息

maptask多个，每个maptask处理一个分片的数据

一行数据调用createRecorderReader方法返回键值对（由换行符决定）

一个（K,V）对调用一次map(k,v，context)方法，然后将数据交由maptask写出到磁盘

数据分发，默认的是key的hash值%reduceNum

reduce数量设置job.setNumReducetasks(2)

reduce保存数据到本地磁盘

分组读取数据（key相同为一组）

一组数据调用一次reduce（k,values迭代器，context）方法

第一节：MR的分片机制

1. 分片简介

Hadoop将MapReduce的输入数据划分成等长的小数据块，称之为输入分片（inputSplit）或者简称"分片"，Hadoop为每一个分片构建一个单独的map任务，并由该任务来运行用户自定义的map方法，从而处理分片中的每一条记录

2. 分片大小的选择

    1. 拥有许多分片，意味着处理每个分片所需要的时间要小于处理整个输入数据所花的时间(分而治之的优势)。
    2. 并行处理分片，且每个分片比较小。负载平衡，好的计算机处理的更快，可以腾出时间，做别的任务
    3. 如果分片太小，管理分片的总时间和构建map任务的总时间将决定作业的整个执行时间。
    4. 如果分片跨越两个数据块，那么分片的部分数据需要通过网络传输到map任务运行的节点，占用网络带宽，效率更低
    5. 因此最佳分片大小应该和HDFS上的块大小一致。hadoop2.x默认128M.
3.

```
`public abstract class FileInputFormat<K, V> implements InputFormat<K, V> {
    public static final String NUM_INPUT_FILES;
        public static final String INPUT_DIR_RECURSIVE;
    private static final double SPLIT_SLOP = 1.1;
    private long minSplitSize = 1;
    ........
    protected FileSplit makeSplit(Path file, long start, long length,
String[] hosts) {
        return new FileSplit(file, start, length, hosts);
    }
    .....
    public InputSplit[] getSplits(JobConf job, int numSplits)throws
IOException {

    //  获取文件的状态信息
    FileStatus[] files = listStatus(job);

    // Save the number of input files for metrics/loadgen
    job.setLong(NUM_INPUT_FILES, files.length);
    long totalSize = 0;                           // compute total size
    for (FileStatus file: files) {                // check we have valid
files
     .....
      totalSize += file.getLen();
    }

    long goalSize = totalSize / (numSplits == 0 ? 1 : numSplits);
    long minSize =
Math.max(job.getLong(org.apache.hadoop.mapreduce.lib.input.
      FileInputFormat.SPLIT_MINSIZE, 1), minSplitSize);

    // generate splits
    ArrayList<FileSplit> splits = new ArrayList<FileSplit>(numSplits);
    for (FileStatus file: files) {
      Path path = file.getPath();
      long length = file.getLen();
      if (length != 0) {
        FileSystem fs = path.getFileSystem(job);
        BlockLocation[] blkLocations;
        .......
        if (isSplitable(fs, path)) {
          long blockSize = file.getBlockSize();
          long splitSize = computeSplitSize(goalSize, minSize, blockSize);
          long bytesRemaining = length;
          while (((double) bytesRemaining)/splitSize > SPLIT_SLOP) {
```

```
            ......
            splits.add(makeSplit(path, length-bytesRemaining, splitSize,
                splitHosts[0], splitHosts[1]));
            bytesRemaining -= splitSize;
          }

          if (bytesRemaining != 0) {
            String[][] splitHosts =
getSplitHostsAndCachedHosts(blkLocations, length
                - bytesRemaining, bytesRemaining, clusterMap);
            splits.add(makeSplit(path, length - bytesRemaining,
bytesRemaining,
                splitHosts[0], splitHosts[1]));
          }
        } else {
          ......
        }
      } else {
        ......
      }
    }
    ......
    return splits.toArray(new FileSplit[splits.size()]);
  }
```

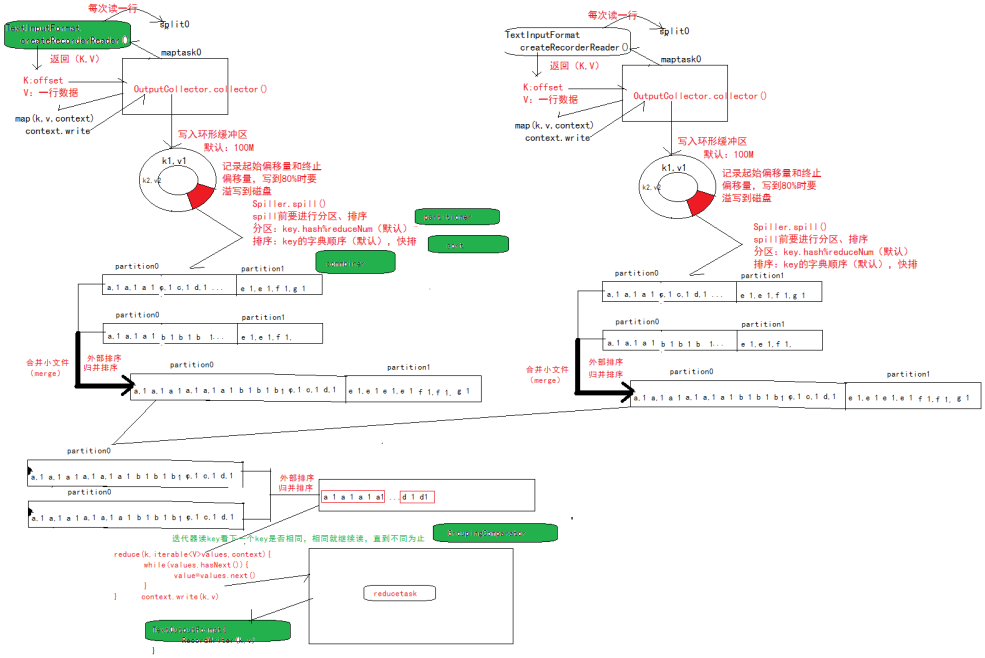
4. 创建分片的过程
    1. 获取文件大小及位置
    2. 判断文件是否可以分片（压缩格式有的可以进行分片，有的不可以）
    3. 获取分片的大小
    4. 剩余文件的大小/分片大小>1.1时，循环执行封装分片信息的方法，具体如下
        1. 封装一个分片信息(包含文件的路径，分片的起始偏移量，要处理的大小，分片包含的块的信息，分片中包含的块存在哪儿些机器上)
        2. 剩余文件的大小/分片大小<=1.1且 不等于0时，封装一个分片信息(包含文件的路径，分片的起始偏移量，要处理的大小，分片包含的块的信息，分片中包含的块存在哪儿些机器上)

    分片的注意事项：1.1倍的冗余。
    问题：260M文件分几个片?

考虑Hadoop应用处理的数据集比较大，因此需要借助压缩。按照效率从高到低排列的
（1）使用容器格式文件，例如：顺序文件、RCFile、Avro数据格式支持压缩和切分文件。另外在配合使用一些快速压缩工具，例如：LZO、LZ4或者Snappy.
（2）使用支持切分压缩格式，例如bzip2
（3）在应用中将文件切分成块，对每块进行任意格式压缩。这种情况确保压缩后的数据库接近HDFS块大小。
（4）存储未压缩文件，以原始文件存储。

3. 读取分片的细节：如果有多个分片
    - 第一个分片读到末尾再多读一行
    - 既不是第一个分片也不是最后一个分片第一行数据舍弃，末尾多读一行
    - 最后一个分片舍弃第一行，末尾多读一行

第一节：MapTask执行流程详解



1. maptask调用FileInputFormat的getRecordReader读取分片数据
2. 每行数据读取一次，返回一个(K,V)对，K是offset,V是一行数据
3. 将k-v对交给MapTask处理
4. 每对k-v调用一次map(K,V，context)方法，然后context.write(k,v)
5. 写出的数据交给收集器OutputCollector.collector()处理
6. 将数据写入环形缓冲区，并记录写入的起始偏移量，终止偏移量，环形缓冲区默认大小100M
7. 默认写到80%的时候要溢写到磁盘，溢写磁盘的过程中数据继续写入剩余20%
8. 溢写磁盘之前要先进行分区然后分区内进行排序
9. 默认的分区规则是hashpatitioner，即key的hash%reduceNum
10. 默认的排序规则是key的字典顺序，使用的是快速排序
11. 溢写会形成多个文件，在maptask读取完一个分片数据后，先将环形缓冲区数据刷写到磁盘
12. 将数据多个溢写文件进行合并，分区内排序（外部排序===》归并排序）

第二节：ReduceTask执行流程详解

1. 数据按照分区规则发送到reducetask
2. reducetask将来自多个maptask的数据进行合并，排序（外部排序===》归并排序）
3. 按照key相同分组（）
4. 一组数据调用一次reduce(k,iterablevalues,context)
5. 处理后的数据交由reducetask
6. reducetask调用FileOutputFormat组件
7. FileOutputFormat组件中的write方法将数据写出

第三节：shuffle流程

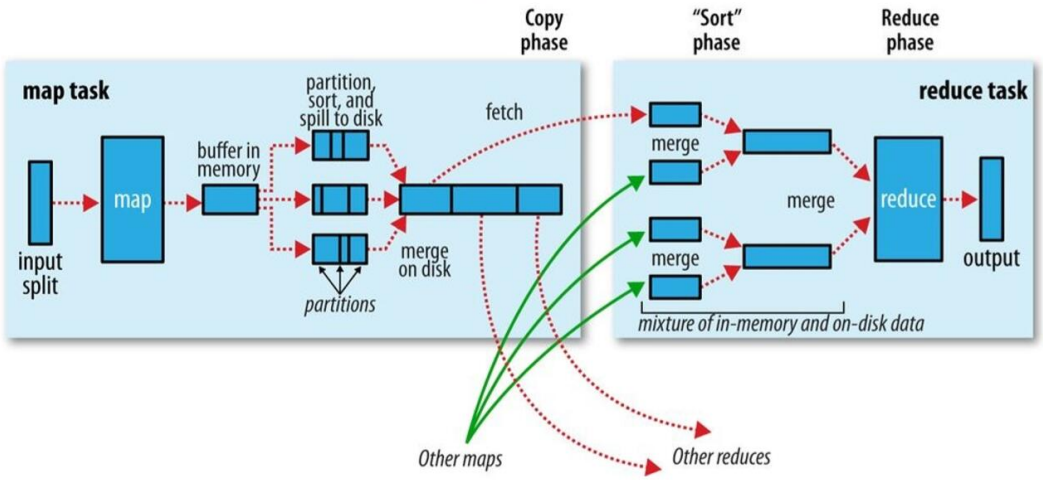


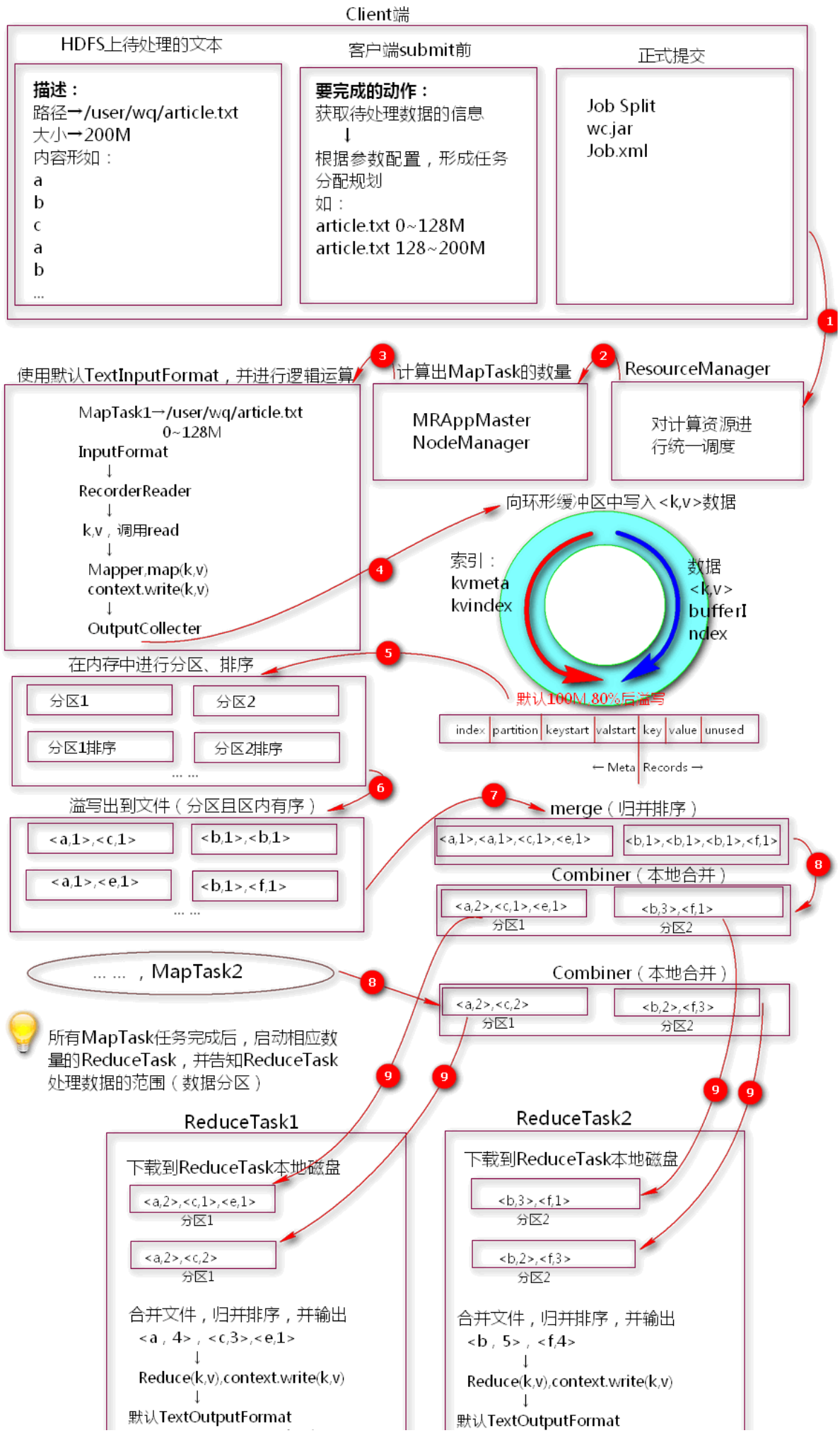
Figure 7-4. Shuffle and sort in MapReduce

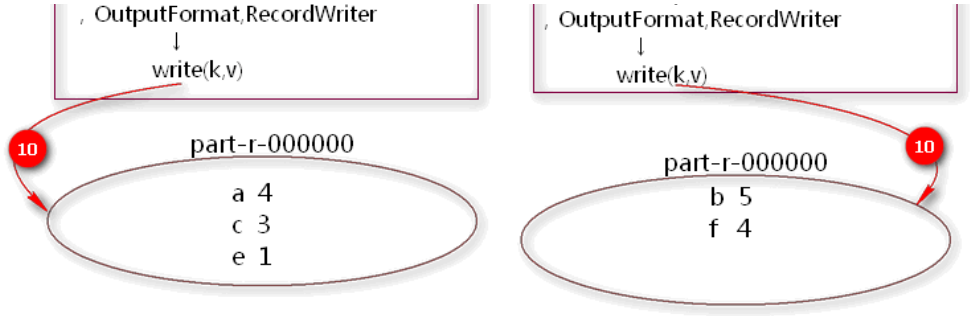
shuffle过程从map写数据到环形缓冲区到reduce读取数据合并(见maptask和reducetask执行过程)

1. 从map函数输出到reduce函数接受输入数据，这个过程称之为shuffle.
2. map函数的输出，存储环形缓冲区（默认大小100M,阈值80M）

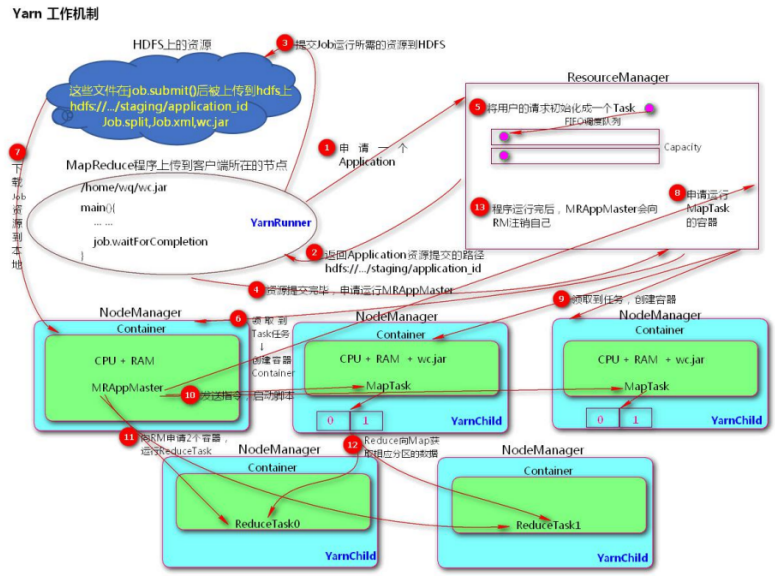
   环形缓冲区：其实是一个字节数组kvbuffer. 有一个sequator标记，kv原始数据从左向右填充(顺时针)，kvmeta是对kvbuffer的一个封装，封装成了int数组，用于存储kv原始数据的对应的元数据（valstart,keystart,partition,vallen）,从右向左(逆时针)。
3. 当达到阈值时，准备溢写到本地磁盘(因为是中间数据，因此没有必要存储在HDFS上)。在溢写前要进行对元数据分区(partition)整理，然后进行排序(quick sort,通过元数据找到出key，同一分区的所有key进行排序，排序完，元数据就已经有序了，在溢写时，按照元数据的顺序寻找原始数据进行溢写)
4. 如果有必要，可以在排序后，溢写前调用combiner函数进行运算，来达到减少数据的目的
5. 溢写文件有可能产生多个，然后对这多个溢写文件进行再次合并(也要进行分区和排序)。当溢写个数>=3时，可以再次调用combiner函数来减少数据。如果溢写个数<3时，默认不会调用combiner函数。
6. 合并的最终溢写文件可以使用压缩技术来达到节省磁盘空间和减少向reduce阶段传输数据的目的。（存储在本地磁盘中）
7. Reduce阶段通过HTTP写抓取属于自己的分区的所有map的输出数据(默认线程数是5，因此可以并发抓取)。
8. 抓取到的数据存在内存中，如果数据量大，当达到本地内存的阈值时会进行溢写操作，在溢写前会进行合并和排序(排序阶段)，然后写到磁盘中，
9. 溢写文件可能会产生多个，因此在进入reduce之前会再次合并(合并因子是10),最后一次合并要满足10这个因子，同时输入给reduce函数，而不是产生合并文件。reduce函数输出数据会直接存储在HDFS上。

MapReduce详细工作流程：





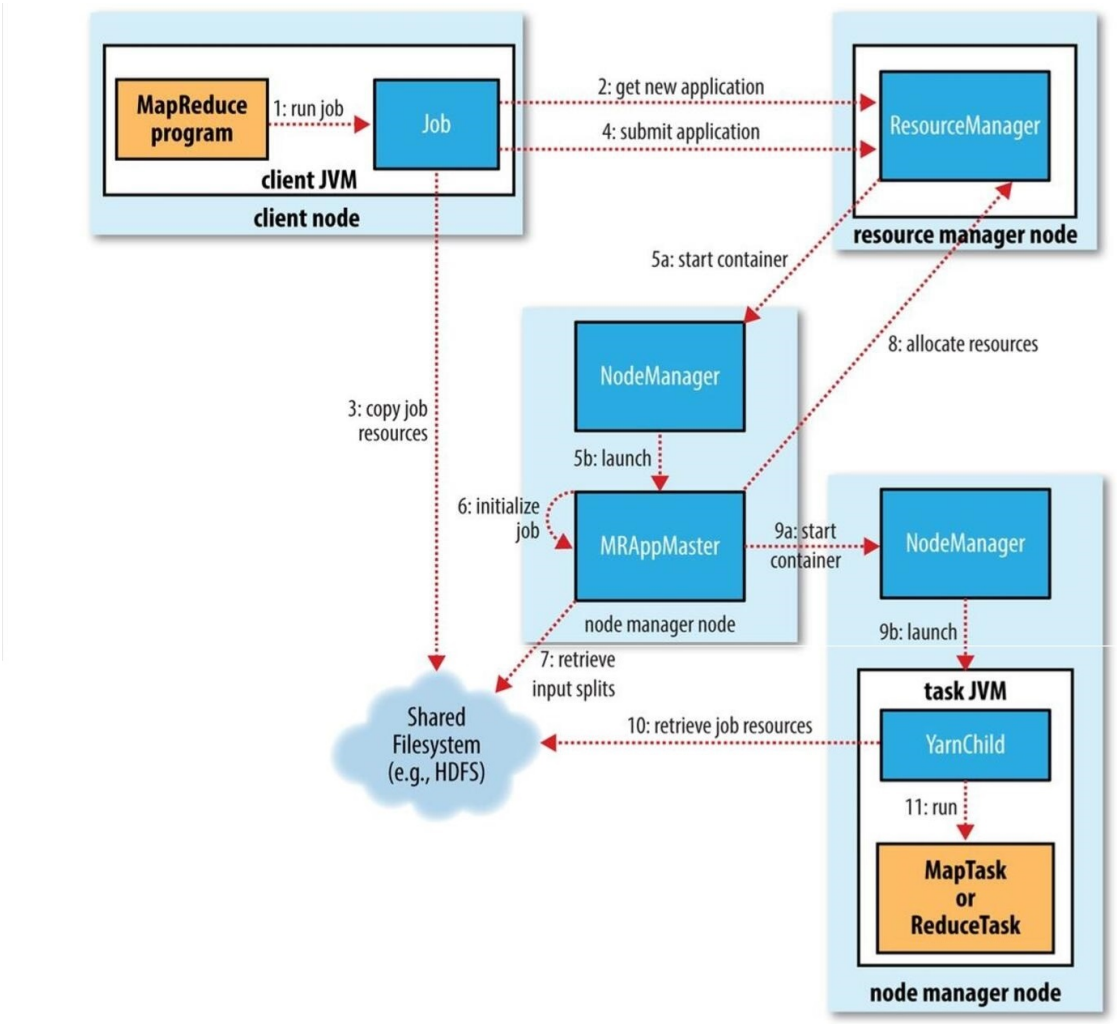
2、请阐述Yarn Job的提交流程。



第三节：job提交流程

整个过程如下图描述。在MR程序运行时，有五个独立的进程：

- YarnRunner:用于提交作业的客户端程序
- ResourceManager:yarn资源管理器，负责协调集群上计算机资源的分配
- NodeManager:yarn节点管理器，负责启动和监视集群中机器上的计算容器（container）
- Application Master:负责协调运行MapReduce作业的任务，他和任务都在容器中运行，这些容器由资源管理器分配并由节点管理器进行管理。
- HDFS:用于共享作业所需文件。



1. 调用waitForCompletion方法每秒轮询作业的进度，内部封装了submit()方法，用于创建JobCommiter实例，并且调用其的submitJobInternal方法。提交成功后，如果有状态改变，就会把进度报告到控制台。错误也会报告到控制台
2. JobCommiter实例会向ResourceManager申请一个新应用ID，用于MapReduce作业ID。这期间JobCommiter也会进行检查输出路径的情况，以及计算输入分片。
3. 如果成功申请到ID,就会将运行作业所需要的资源（包括作业jar文件，配置文件和计算所得的输入分片元数据文件）上传到一个用ID命名的目录下的HDFS上。此时副本个数默认是10.
4. 准备工作已经做好，再通知ResourceManager调用submitApplication方法提交作业。
5. ResourceManager调用submitApplication方法后，会通知Yarn调度器（Scheduler），调度器分配一个容器，在节点管理器的管理下在容器中启动 application master进程。
6. application master的主类是MRAppMaster，其主要作用是初始化任务，并接受来自任务的进度和完成报告。
7. 然后从HDFS上接受资源，主要是split。然后为每一个split创建MapTask以及参数指定的ReduceTask，任务ID在此时分配
8. 然后Application Master会向资源管理器请求容器，首先为MapTask申请容器，然后再为ReduceTask申请容器。（5%）
9. 一旦ResourceManager中的调度器（Scheduler），为Task分配了一个特定节点上的容器，Application Master就会与NodeManager进行通信来启动容器。
10. 运行任务是由YarnChild来执行的，运行任务前，先将资源本地化（jar文件，配置文件，缓存文件）
11. 然后开始运行MapTask或ReduceTask。
12. 当收到最后一个任务已经完成的通知后，application master会把作业状态设置为success。然后Job轮询时，知道成功完成，就会通知客户端，并把统计信息输出到控制台

3、请阐述Yarn的默认调度器、调度器分类、以及他们之间的区别

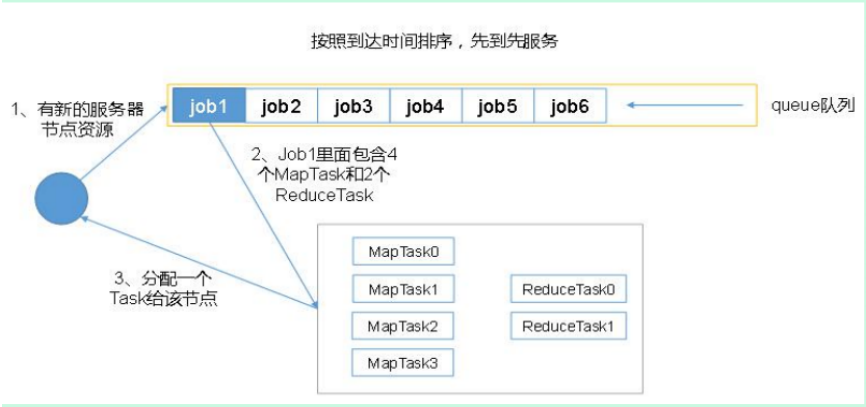
1）Hadoop调度器重要分为三类：

FIFO 、Capacity Scheduler（容量调度器）和Fair Sceduler（公平调度器）。

hadoop yarn本身默认使用容量调度器（Capacity），但是一些分布式项目（如CDH）默认使用公平调度器（Fair）

2）区别：

→ FIFO调度器：先进先出，同一时间队列中只有一个任务在执行。



3、请阐述Yarn的默认调度器、调度器分类、以及他们之间的区别（续）

2）区别（续）：

→ 容量调度器：多队列；每个队列内部先进先出，同一时间队列中只有一个任务在执行。队列的并行度为队列的个数。

3、请阐述Yarn的默认调度器、调度器分类、以及他们之间的区别（续）

2）区别（续）：

→ 公平调度器：多队列；每个队列内部按照缺额大小分配资源启动任务，同一时间队列中有多个任务执行。队列的并行度大于等于队列的个数。

一定要强调生产环境中不是使用的FifoScheduler，面试的时侯会发现候选人大概了解这几种调度器的区别，但是问在生产环境用哪种，却说使用的FifoScheduler（企业生产环境一定不会用这个调度的）

1. hive出现的原因

FaceBook网站每天产生海量的数据。为了对这些数据进行管理，并且因为机器学习的需求，产生了hive这门技术，并继续发展成为一个成功的Apache项目。

2. 什么是hive

hive是一个构建在Hadoop上的数据仓库框架（工具），可以将结构化的数据映射成一张数据库表，并可以使用类sql的方式来对大数据集进行读，写以及管理（元数据），这套HIVE SQL 简称HQL。hive的执行引擎可以是MR、spark、tez。如果执行引擎是MR的话，hive会将sql翻译成MR进行数据的计算。用户可以使用命令行工具和JDBC驱动程序来连接到hive

3. 为什么使用hive

- 直接使用MapReduce所面临的问题是：
    - 人员学习成本高
    - 项目周期要求太短
    - MapReduce实现复杂查询逻辑开发难度大
- 使用hive的优势：
    - 操作接口采用类sql语法，提供快速开发的能力。
    - 避免了去写MapReduce（适合java语言不好的，sql熟练的人），减少开发人员的学习成本。
    - 功能扩展很方便
    - 适合进行离线分析处理

4. hive的优缺点

1. hive的优点
    - 学习成本低:
      提供了类SQL查询语言HQL
    - 可扩展
      为超大数据集设计了计算/扩展能力（MR作为计算引擎，HDFS作为存储系统），Hive可以自由的扩展集群的规模，一般情况下不需要重启服务。
    - 延展性
      Hive支持用户自定义函数，用户可以根据自己的需求来实现自己的函数。
    - 容错
      良好的容错性，节点出现问题SQL仍可完成执行
    - 提供了统一的元数据管理
2. hive的缺点
    - hive的HQL表达能力有限

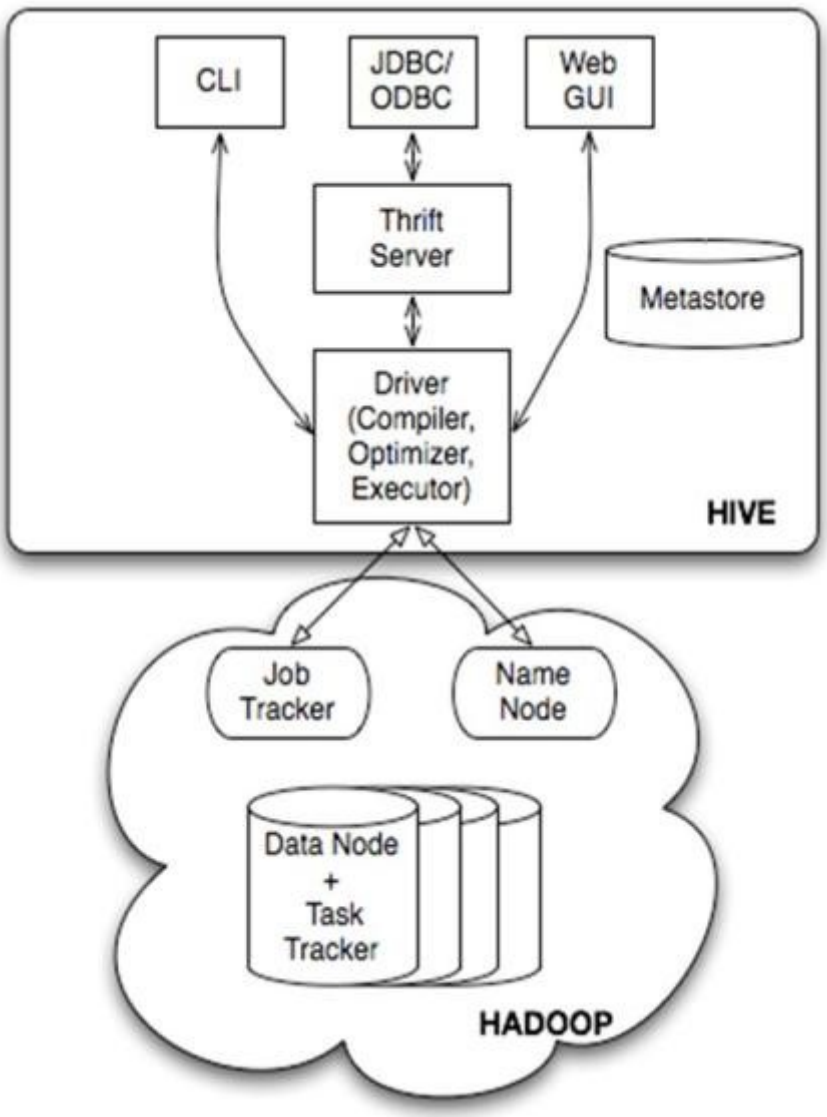
```
    - 迭代式算法无法表达，比如pagerank
    - 数据挖掘方面，比如kmeans

- hive的效率比较低

    - hive自动生成的mapreduce作业，通常情况下不够智能化
    - hive调优比较困难，粒度较粗
```

第二节：hive的架构

1. hive的架构简介



从上图可以看出，Hive的体系结构分为以下几部分:

1. 用户连接工具

   主要有三个：CLI，Client 和 WUI。其中最常用的是CLI，Cli启动的时候，会同时启动一个Hive副本。Client是Hive的客户端，用户连接至Hive Server。在启动 Client模式的时候，需要指出Hive Server所在节点，并且在该节点启动Hive Server。 WUI是通过浏览器访问Hive。

- thriftserver：第三方服务
- 元数据

  Hive将元数据存储在数据库中，如mysql、derby。Hive中的元数据库名、表名、字段名、字段类型、分区、分桶、创建时间、创建人等）。
- 解释器：将hql抽象成表达式树
- 编译器：

  对hql语句进行词法、语法、语义的编译(需要跟元数据关联)，编译完成后会生成一个执行计划。hive上就是编译成mapreduce的job。
- 优化器：

  将执行计划进行优化，减少不必要的列、使用分区、使用索引等。优化job。
- 执行器：将优化后的执行计划提交给hadoop的yarn上执行。提交job。

- hadoop

```
Hive的数据存储在HDFS中，大部分的查询、计算由MapReduce完成（包含*的查询，比如select * from tbl不会生成MapRedcue任务）。
```

2. hive和hadoop的关系

- hive本身其实没有多少功能，hive就相当于在hadoop上面包了一个壳子，就是对hadoop进行了一次封装。
- hive的存储是基于hdfs/hbase的，hive的计算是基于mapreduce。

3. hive与数据库的区别

1. Hive采用了SQL的查询语言HQL，因此很容易将Hive理解为数据库。其实从结构上来看，Hive和数据库除了拥有类似的查询语言，再无类似之处。
2. 数据库可以用在Online的应用中，但是Hive是为数据仓库而设计的，清楚这一点，有助于从应用角度理解Hive的特性。
3. Hive 不适合用于联机(online) 事务处理，也不提供实时查询功能。它最适合应用在基于大量不可变数据的批处理作业。Hive 的特点是可 伸缩（在Hadoop 的集群上动态的添加设备），可扩展、容错、输入格式的松散耦合。Hive 的入口是DRIVER ，执行的 SQL 语句首先提交到 DRIVER 驱动，然后调用 COMPILER 解释驱动， 最终解释成 MapReduce 任务执行，最后将结果返回。
4. MapReduce 开发人员可以把自己写的 Mapper 和 Reducer 作为插件支持 Hive 做更复杂的数据分析。 它与关系型数据库的 SQL 略有不同，但支持了绝大多数的语句（如 DDL、DML）以及常见的聚合函数、连接查询、条件查询等操作。
5. Hive和数据库的比较如下表：

比较项	SQL	HiveQL
ANSI SQL	支持	不完全支持
更新	UPDATE\INSERT\DELETE	insert OVERWRITE\INTO TABLE
事务	支持	不支持
模式	写模式	读模式
数据保存	块设备、本地文件系统	HDFS
延时	低	高
多表插入	不支持	支持
子查询	完全支持	只能用在From子句中
视图	Updatable	Read-only
可扩展性	低	高
数据规模	小	大
....	.....	.....