项目知识点总结(未完待续)

大数据本质：

在庞大数据集中找到一个模型，利用模型产生挖掘有价值的信息

云计算：

以服务为中心；超大规模的计算机集群，通过技术整合起来，能够按需按量提供**服务**

价格便宜，提供服务

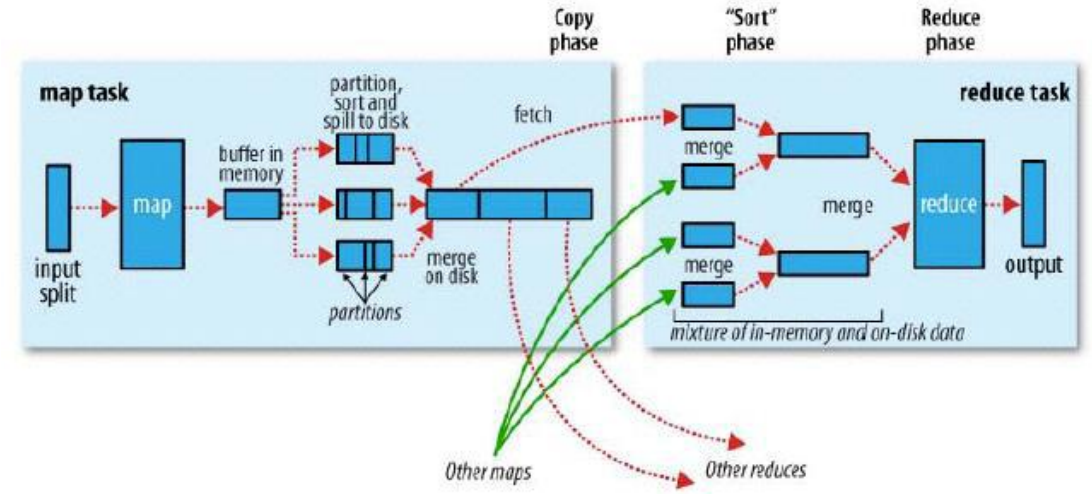
1、Hadoop 的调度器，以及其工作方法

a) FIFO 调度：先进先出

b) 计算能力调度(Capacity schedular)：选择占用最小、优先级高的执行，依此类推

c) 公平调度(Fair schedular)：所有 Job 具有相同的资源

2、Hadoop 核心



Map 端总结：

（1）map 过程的输出是写入本地磁盘而不是 HDFS，但是一开始数据并不是直接写入磁盘而是缓冲在内存中，缓存的好处就是 **减少磁盘 I/O 的开销，提高合并和排序的速度** 。又因为默认的内存缓冲大小是 100M（当然这个是可以配置的），所以 在编写 map 函数的时候要尽量减少内存的使用，为 shuffle 过程预留更多的内存 ，因为该过程是最耗时的过程。

（2）写磁盘前，要进行 partition、sort 和 combine 等操作。通过分区，将不同类型的数据分开处理，之后对不同分区的数据进行 排序，如果有 Combiner，还要对排序后的数据进行 combine。等最后记录写完，将全部溢出文件合并为一个分区且排序的文件。

（3）最后将磁盘中的数据送到 Reduce 中，从图中可以看出 Map 输出有三个分区，有一个分区数据被送到图示的 Reduce 任务中，剩下的两个分区被送到其他 Reducer 任务中。而图示的 Reducer 任务的其他的三个输入则来自其他节点的 Map 输出。

在写磁盘的时候采用**压缩**的方式将 map 的输出结果进行压缩是一个减少网络开销很有效的方法！

Reduce 端总结：

（1）Copy 阶段：Reducer 通过 Http 方式得到输出文件的分区。
reduce 端可能从 n 个 map 的结果中获取数据，而这些 map 的执行速度不尽相同，当其中一个 map 运行结束时，reduce 就会从 JobTracker 中获取该信息。map 运行结束后 TaskTracker 会得到消息，进而将消息汇报给 JobTracker，reduce 定时从 JobTracker 获取该信息，reduce 端默认有 5 个数据复制线程从 map 端复制数据。
（2）Merge 阶段：如果形成多个磁盘文件会进行合并
从 map 端复制来的数据首先写到 reduce 端的缓存中，同样缓存占用到达一定阈值后会将数据写到磁盘中，同样会进行 partition、 combine、排序等过程。如果形成了多个磁盘文件还会进行合并，最后一次合并的结果作为 reduce 的输入而不是写入到磁盘中。
（3）Reducer 的参数：最后将合并后的结果作为输入传入 Reduce 任务中。
总结： 当 Reducer 的输入文件确定后，整个 Shuffle 操作才最终结束。之后就是 Reducer 的执行了，最后 Reducer 会把结果存到 HDFS 上。

3、HDFS（Hadoop Distributed File System ）

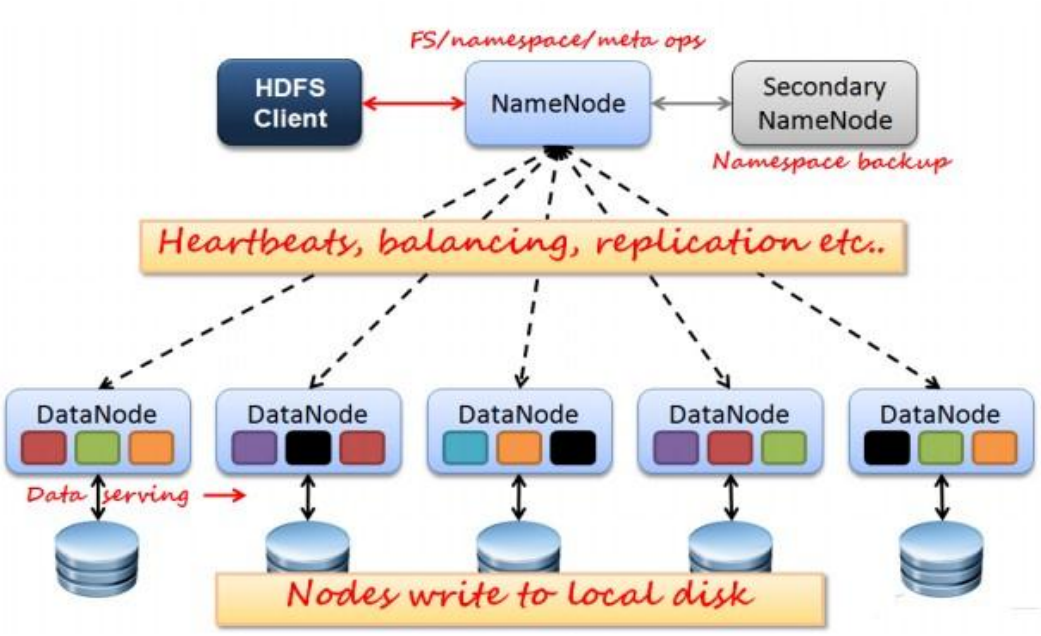
Hadoop 分布式文件系统。是根据 google 发表的论文翻版的。论文为 GFS（Google File System）Google 文件系统（中文，英文）。

HDFS 有很多特点：

① 保存多个副本，且提供容错机制，副本丢失或宕机自动恢复。默认存 3 份。

② 运行在廉价的机器上。

③ 适合大数据的处理。多大？多小？HDFS 默认会将文件分割成 block，64M 为 1 个 block。然后将 block 按键值对存储在 HDFS 上，并将键值对的映射存到内存中。如果小文件太多，那内存的负担会很重。



如上图所示，HDFS 也是按照 Master 和 Slave 的结构。分 NameNode、SecondaryNameNode、DataNode 这几个角色。

NameNode：是 Master 节点，是大领导。管理数据块映射；处理客户端的读写请求；配置副本策略；管理 HDFS 的名称空间；

SecondaryNameNode：是一个小弟，分担大哥 namenode 的工作量；是 NameNode 的冷备份；合并 fsimage 和 fsedits 然后再发给 namenode。

DataNode：Slave 节点，奴隶，干活的。负责存储 client 发来的数据块 block；执行数据块的读写操作。

热备份：b 是 a 的热备份，如果 a 坏掉。那么 b 马上运行代替 a 的工作。

冷备份：b 是 a 的冷备份，如果 a 坏掉。那么 b 不能马上代替 a 工作。但是 b 上存储 a 的一些信息，减少 a 坏掉之后的损失。

fsimage:元数据镜像文件（文件系统的目录树。）

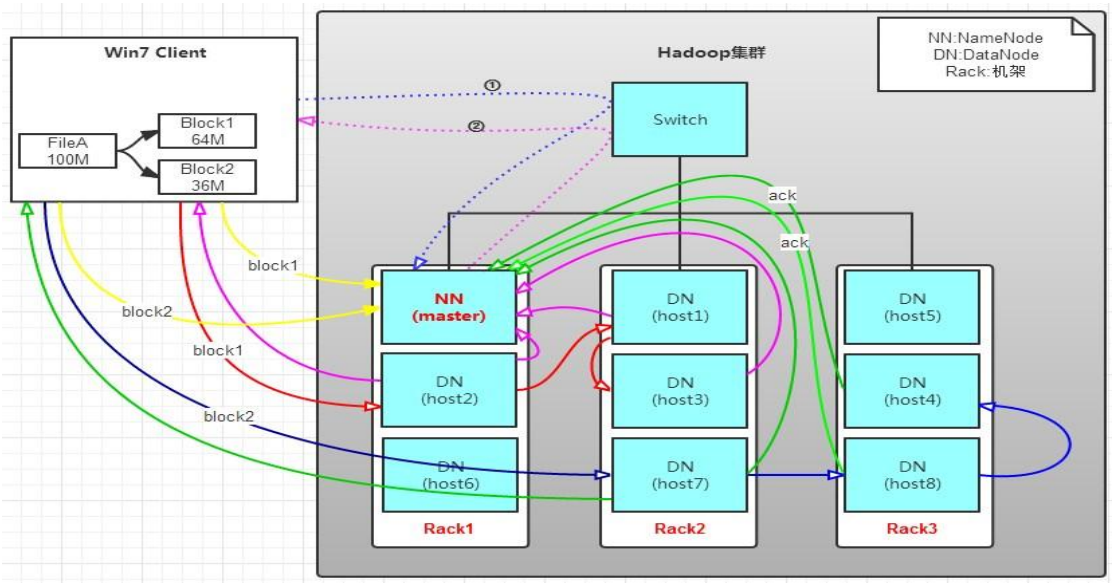
edits：元数据的操作日志（针对文件系统做的修改操作记录）

namenode 内存中存储的是=fsimage+edits。

SecondaryNameNode 负责定时默认 1 小时，从 namenode 上，获取 fsimage 和 edits 来进行合并，然后再发送给 namenode。减少 namenode 的工作量。

工作原理

写操作：



有一个文件 FileA，100M 大小。Client 将 FileA 写入到 HDFS 上。

HDFS 按默认配置。

HDFS 分布在三个机架上 Rack1，Rack2，Rack3。

a. Client 将 FileA 按 64M 分块。分成两块，block1 和 Block2;

b. Client 向 nameNode 发送写数据请求，如图蓝色虚线①------>。

c. NameNode 节点，记录 block 信息。并返回可用的 DataNode，如粉色虚线②--------->。

Block1: host2,host1,host3

Block2: host7,host8,host4

原理：

NameNode 具有 RackAware 机架感知功能，这个可以配置。

若 client 为 DataNode 节点，那存储 block 时，规则为：副本 1，同 client 的节点上；副本 2，不同机架节点上；副本 3，同第二个副本机架的另一个节点上；其他副本随机挑选。

若 client 不为 DataNode 节点，那存储 block 时，规则为：副本 1，随机选择一个节点上；副本 2，不同副本 1，机架上；副本 3，同副本 2 相同的另一个节点上；其他副本随机挑选。

d. client 向 DataNode 发送 block1；发送过程是以流式写入。

流式写入过程，

1>将 64M 的 block1 按 64k 的 package 划分;

2>然后将第一个 package 发送给 host2;

3>host2 接收完后，将第一个 package 发送给 host1，同时 client 想 host2 发送第二个 package;

4>host1 接收完第一个 package 后，发送给 host3，同时接收 host2 发来的第二个 package。

5>以此类推，如图红线实线所示，直到将 block1 发送完毕。

6>host2,host1,host3 向 NameNode，host2 向 Client 发送通知，说"消息发送完了"。如图粉红颜色实线所示。

7>client 收到 host2 发来的消息后，向 namenode 发送消息，说我写完了。这样就真完成了。如图黄色粗实线

8>发送完 block1 后，再向 host7，host8，host4 发送 block2，如图蓝色实线所示。

9>发送完 block2 后，host7,host8,host4 向 NameNode，host7 向 Client 发送通知，如图浅绿色实线所示。

10>client 向 NameNode 发送消息，说我写完了，如图黄色粗实线。。。这样就完毕了。

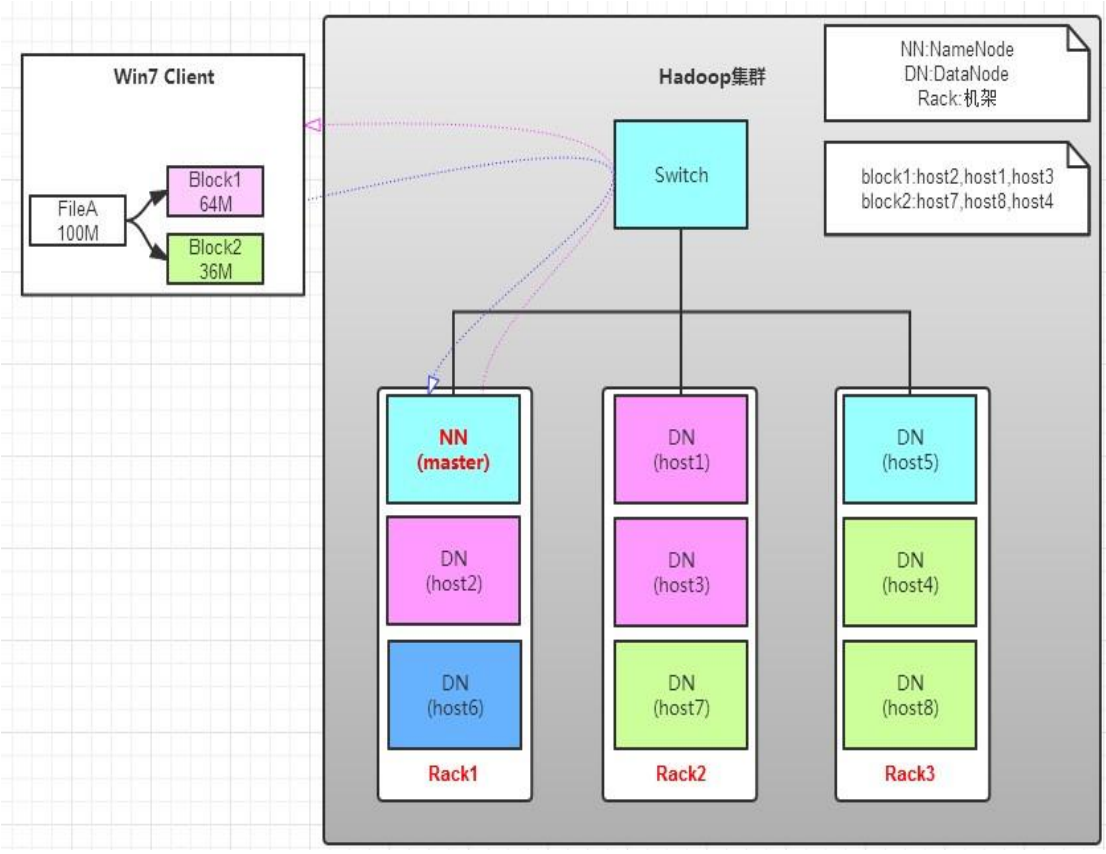
分析，通过写过程，我们可以了解到：

①写 1T 文件，我们需要 3T 的存储，3T 的网络流量贷款。

②在执行读或写的过程中，NameNode 和 DataNode 通过 HeartBeat 进行保存通信，确定 DataNode 活着。如果发现 DataNode 死掉了，就将死掉的 DataNode 上的数据，放到其他节点去。读取时，要读其他节点去。

③挂掉一个节点，没关系，还有其他节点可以备份；甚至，挂掉某一个机架，也没关系；其他机架上，也有备份。

读操作：



读操作就简单一些了，如图所示，client 要从 datanode 上，读取 FileA。而 FileA 由 block1 和 block2 组成。

那么，读操作流程为：

a. client 向 namenode 发送读请求。

b. namenode 查看 Metadata 信息，返回 fileA 的 block 的位置。

block1:host2,host1,host3

block2:host7,host8,host4

c. block 的位置是有先后顺序的，先读 block1，再读 block2。而且 block1 去 host2 上读取；然后 block2，去 host7 上读取；

上面例子中，client 位于机架外，那么如果 client 位于机架内某个 DataNode 上，例如,client 是 host6。那么读取的时候，遵循的规律是：

优选读取本机架上的数据。

HDFS 中常用到的命令

1、hadoop fs

```
hadoop fs -ls /
hadoop fs -lsr
hadoop fs -mkdir /user/hadoop
hadoop fs -put a.txt /user/hadoop/
hadoop fs -get /user/hadoop/a.txt /
hadoop fs -cp src dst
hadoop fs -mv src dst
hadoop fs -cat /user/hadoop/a.txt
hadoop fs -rm /user/hadoop/a.txt
hadoop fs -rmr /user/hadoop/a.txt
hadoop fs -text /user/hadoop/a.txt
hadoop fs -copyFromLocal localsrc dst 与 hadoop fs -put 功能类似。
hadoop fs -moveFromLocal localsrc dst 将本地文件上传到 hdfs，同时删除本地文件。
```

2、hadoop fsadmin

```
hadoop dfsadmin -report
hadoop dfsadmin -safemode enter | leave | get | wait
hadoop dfsadmin -setBalancerBandwidth 1000
```

3、hadoop fsck

4、start-balancer.sh

4、Hadoop 调优

1）hdfs-site.xml 配置文件

dfs.block.size： 块大小的设置

dfs.replication： 复制数量的设置，不能为 0，并不是备份数量，而是存放数据的分数

2）mapred-site.xml 配置文件

mapred.tasktracker.map.tasks.maximum、

mapred.tasktracker.reduce.tasks.maximum：

用来设置的 map 和 reduce 的并发数量，实际作用的就是控制同时运行的 task 的数量

mapred.child.java.opts：置每个 map 或 reduce 使用的内存数量。默认的是 200M，如果配置的太小，则有可能出现”无可分配内存”的错误。

mapred.reduce.tasks： 设置 reduce 的数量

3）core-sit.xml 配置文件

webinterface.private.actions： 这个参数实际上就是为了方便测试用。允许在 web 页面上对任务设置优先级以及 kill 任务

调优主要有三种优化思路：

http://wenku.baidu.com/link?url=o_GsOBBwPAwYfGBL6b3hA8-giD9Kqvmv047_ssH4bs9YxXPxjOYO3n2_oYGOhgv_bR-ASDfEh1n7m73EciuTOuqBeTyDWbPNPzwGNWBddMC

从应用程序角度进行优化：

1）**避免不必要的 reduce 任务。**如果要处理的数据是排序且已经分区，或者对于一份数据需要多次处理，可以先排序分区，然后自定义 InputSplit，将单个分区作为 map 的输入；在 map 中处理数据，Reducer 设置为空，既重用了已有的“分区”，也避免多余的 reduce 任务

2）**外部文件引入。**有些应用程序要使用外部文件，如字典，配置文件等，这些文件需要在所有 task 之间共享，可以放到分布式缓存 DistributedCache 中（或直接采用-files 选项，机制相同）。更多的这方面的优化方法，还需要在实践中不断积累。

3）**为 job 添加一个 Combiner。**为 job 添加一个 combiner 可以大大减少 shuffle 阶段从 map task 拷贝给远程 reduce task 的数据量。一般而言，combiner 与 reducer 相同。

4）**根据处理数据特征使用最适合和简洁的 Writable 类型。**Text 对象使用起来很方便，但它在由数值转换到文本或是由 UTF8 字符串转换到文本时都是低效的，且会消耗大量的 CPU 时间。当处理那些非文本的数据时，可以使用二进制的 Writable 类型，如 IntWritable，FloatWritable 等。二进制 writable 好处：避免文件转换的消耗；使 map task 中间结果占用更少的空间。

5）**使用 StringBuffer 而不是 String。**当需要对字符串进行操作时，使用 StringBuffer 而不是 String，String 是 read-only 的，如果对它进行修改，会产生临时对象，而 StringBuffer 是可修改的，不会产生临时对象。

6）**调试。**最重要，也是最基本的，是要掌握 MapReduce 程序调试方

法，跟踪程序的瓶颈。

从参数配置角度进行优化：

1）参数自动调优

a) Linux 文件系统参数调整

i. **noatime 和 nodiratime 属性。**文件挂载时设置这两个属性可以明显提高性能

ii. readahead buffer。调整 linux 文件系统中预读缓冲区地大小，可以明显提高顺序读文件的性能。默认 buffer 大小为 256 sectors，可以增大为 1024 或者 2408 sectors（注意，并不是越大越好）。可使用 blockdev 命令进行调整。

iii. 避免 RAID 和 LVM 操作。避免在 TaskTracker 和 DataNode 的机器上执行 RAID 和 LVM 操作，这通常会降低性能。

2）参数手动设置

a) **Dfs.namenode.handler.count、**

mapred.job.tracker.handler.count。

namenode 或者 jobtracker 中用于处理 RPC 的线程数，默认是 10，较大集群，可调大些，比如 64。

b) **dfs.datanode.handler.count。**datanode 上用于处理 RPC 的线程数。默认为 3，较大集群，可适当调大些，比如 8。需要注意的是，每添加一个线程，需要的内存增加。

c) **tasktracker.http.threads。**HTTP server 上的线程数。运行在每个 TaskTracker 上，用于处理 map task 输出。大集群，可以将其设为 40~50。

3）HDFS 相关参数设置

a) dfs.replication。文件副本数，通常设为 3，不推荐修改。

b) **dfs.block.size。**HDFS 中数据 block 大小，默认为 64M，对于较大集群，可设为 128MB 或者 256MB。（也可以通过参数 mapred.min.split.size 配置）

c) **mapred.local.dir 和 dfs.data.dir 。**这两个参数 mapred.local.dir 和 dfs.data.dir 配置的值应当是分布在各个磁盘上目录，这样可以充分利用节点的 IO 读写能力。运行 Linux sysstat 包下的 iostat -dx 5 命令可以让每个磁盘都显示它的利用率。

4）**MapReduce**

a) **{map/reduce}.tasks.maximum。**同时运行在 TaskTracker 上的最大 map/reduce tas 数，一般设为(core_per_node)/2~2*（cores_per_node）。

b) **io.sort.factor。**当一个 map task 执行完之后，本地磁盘上(mapred.local.dir)有若干个 spill 文件，map task 最后做的一件事就是执行 merge sort，把这些 spill 文件合成一个文件（partition）。执行 merge sort 的时候，每次同时打开多少个 spill 文件由该参数决定。打开的文件越多，不一定 merge sort 就越快，所以要根据数据情况适当的调整。

c) **mapred.child.java.opts。**设置 JVM 堆的最大可用内存，需

从应用程序角度进行配置。

5）Map Task 相关配置

a） **io.sort.mb。**Map task 的输出结果和元数据在内存中所占的 buffer 总大小。默认为 100M，对于大集群，可设为 200M。当 buffer 达到一定阈值，会启动一个后台线程来对 buffer 的内容进行排序，然后写入本地磁盘(一个 spill 文件)。

b） **io.sort.spill.percent。**这个值就是上述 buffer 的阈值，默认是 0.8，即 80%，当 buffer 中的数据达到这个阈值，后台线程会起来对 buffer 中已有的数据进行排序，然后写入磁盘。

c） **io.sort.record。**Io.sort.mb 中分配给元数据的内存百分比，默认是 0.05。这个需要根据应用程序进行调整。

d） **mapred.compress.map.output/ Mapred.output.compress。**中间结果和最终结果是否要进行压缩，如果是指定压缩式 Mapred.compress.map.output.codec/
Mapred.output.compress.codec。推荐使用 LZO 压缩。Intel 内部测试表明，相比未压缩，使用 LZO 压缩的 TeraSort 作业运行时间减少 60%，且明显快于 Zlib 压缩。

6）Reduce Task 相关配置

a） **Mapred.reduce.parallel。**Reduce shuffle 阶段 copier 线程数。默认是 5，对于较大集群，可调整为 16~25。

从系统实现角度进行优化：

1）在可移植性和性能之间进行权衡

HDFS 性能低下的两个原因：调度延迟和可移植性假设[3]。

1.调度延迟。Hadoop 采用的是动态调度算法，即：当某个 tasktracker 上出现空 slot 时，它会通过 HEARBEAT（默认时间间隔为 3s，当集群变大时，会适当调大）告诉 jobtracker，之后 jobtracker

采用某种调度策略从待选 task 中选择一个，再通过 HEARBEAT 告诉 tasktracker。从整个过程看，HDFS 在获取下一个 task 之前，一直处于等待状态，这造成了资源利用率不高。此外，

由于 tasktracker 获取新 task 后，其数据读取过程是完全串行化的，即：tasktracker 获取 task 后，依次连接 namenode，连接 datanode 并读取数据，处理数据。

在此过程中，当 tasktracker 连接 namenode 和 datanode 时，HDFS 仍在处于等待状态。为了解决调度延迟问题，可以考虑的解决方案有：重叠 I/O 和 CPU 阶段（pipelining），

task 预取（task prefetching），数据预取（data prefetching）等.

2.可移植性假设。

为了增加 Hadoop 的可移植性，它采用 java 语言编写，这实际上也潜在的造成了 HDFS 低效。Java 尽管可以让 Hadoop 的可移植性增强，但是它屏蔽了底层文件系统，

这使它没法利用一些底层的 API 对数据存储和读写进行优化。首先，在共享集群环境下，大量并发读写会增加随机寻道，这大大

降低读写效率；另外，并发写会增加磁盘碎片，这将增加读取代价
（HDFS 适合文件顺序读取）。为了解决该问题，可以考虑的解决方案有：修改 tasktracker 上的线程模型，现在 Hadoop 上的采用的模型是 one thread per client，
即每个 client 连接由一个线程处理（包括接受请求，处理请求，返回结果）；修改之后，可将线程分成两组，一组用于处理 client 通信
（Client Thread），一组用于存取数据（Disk Threads，可采用 one thread per disk）。

2) Five Factors

影响 Hadoop 性能主要有 5 个因素，分别为计算模型，I/O 模型，数据解析，索引和调度

①计算模型。在 Hadoop 中，map task 产生的中间结果经过 sort-merge 策略处理后交给 reduce task。而这种处理策略（指 sort-merge）不能够定制，这对于有些应用而言（有些应用程序可能不需要排序处理），性能不佳。此外，即使是需要排序归并处理的，sort-merge 也并不是最好的策略。我们可以实现 Fingerprinting Based Grouping（基于 hash）策略，该方法可以明显提高 Hadoop 的性能。

② I/O 模型。Reader 可以采用两种方式从底层的存储系统中读取数据：direct I/O 和 streaming I/O。direct I/O 是指 reader 直接从本地文件中读取数据；streaming I/O 指使用某种进程间通信方式（如 TCP 或者 JDBC）从另外一个进程中获取数据。从性能角度考虑，direct I/O 性能更高，各种数据库系统都是采用 direct I/O 模式。但从存储独立性考虑，streaming I/O 使 Hadoop 能够从任何进程获取数据，如 datanode 或 database，此外，如果 reader 不得不从远程节点上读取数据，streaming I/O 是仅有的选择。

我们对 hadoop 的文件读写方式进行改进，当文件位于本地时，采用 direct I/O 方式；当文件位于其它节点上时，采用 streaming I/O 方式。（改进之前，hadoop 全是采用 streaming I/O 方式）。改进后，效率约提高 10%。

③数据解析。在 hadoop 中，原始数据要被转换成 key/value 的形式以便进一步处理，这就是数据解析。现在有两种数据解析方法：immutable decoding and mutable decoding。Hadoop 是采用 java 语言编写的，java 中很多对象是 immutable，如 String。当用户试图修改一个 String 内容时，原始对象会被丢弃而新对象会被创建以存储新内容。在 Hadoop 中，采用了 immutable 对象存储字符串，这样每解析一个 record 就会创建一个新的对象，这就导致了性能低下。

④索引。HDFS 设计初衷是处理无结构化数据，既然这样，怎么可能为数据添加索引。实际上，考虑到以下几个因素，仍可以给数据添加索引：hadoop 提供了结构将数据记录解析成 key/value 对，这样也许可以给 key 添加索引；如果作业的输入是一系列索引文件，可以实现一个新的 reader 高效处理这些文件；

⑤**调度。**Hadoop 采用的是动态调度策略，即每次调度一个 task 运行，这样会带来部分开销。而 database 采用的静态调度的策略，即在编译的时候就确定了调度方案。当用户提交一个 sql 时，优化器会生成一个分布式查询计划交给每一个节点进行处理。可以使用一个 benchmark 评估运行时调度的代价，最终发现运行时调度策略从两个角度影响性能：需要调度的 task 数；调度算法。对于第一个因素，可以调整 block 的大小减少 task 数，对于第二个因素，需要做更多研究，设计新的算法。通过调整 block 大小(从 64 增大到 5G)，发现 block 越大，效率越高，提升性能约 20%~30%。

对于第一种思路，需要根据具体应用需求而定，同时也需要在长期实践中积累和总结；
对于第二种思路，大部分采用的方法是根据自己集群硬件和具体应用调整参数，找到一个最优的。
对于第三种思路，难度较大，但效果往往非常明显，总结这方面的优化思路，主要有以下几个：

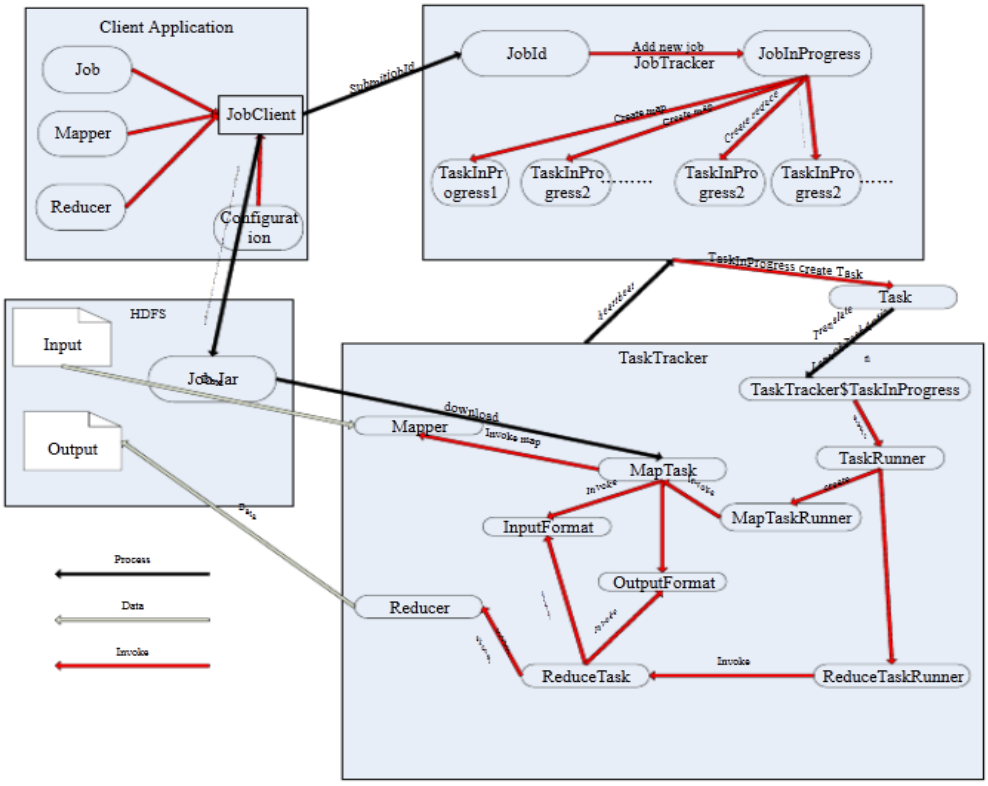
①对 namenode 进行优化，包括增加其吞吐率和解决其单点故障问题。
当前主要解决方案有 3 种：分布式 namenode，namenode 热备和 zookeeper。
②HDFS 小文件问题。当 Hadoop 中存储大量小文件时，namenode 扩展性 和性能受到极大制约。现在 Hadoop 中已有的解决方案包括：Hadoop Archive，Sequence file 和 CombineFileInputFormat。
③调度框架优化。在 Hadoop 中，每当出现一个空闲 slot 后，tasktracker 都需要通过 HEARBEAT 向 jobtracker 所要 task，这个过程的延迟比较大。可以用 task 预调度的策略解决该问题。
④共享环境下的文件并发存取。在共享环境下，HDFS 的随机寻道次数增加，这大大降低了文件存取效率。可以通过优化磁盘调度策略的方法改进。
⑤索引。索引可以大大提高数据读取效率，如果能根据实际应用需求， 为 HDFS 上的数据建立索引，将大大提高效率。

5、Hadoop 中 job 作业流程

http://wenku.baidu.com/link?url=s5BPqZl2EQhAr96Lo9FtXd_K9ttYO61zRlsUodVArIWib8cpjXKq9Jz09tnVBXwV-snNaqw3M9YSrua5K0NutaVWyJKI3ZJGO3msCsrtRNG

http://blog.sina.com.cn/s/blog_3fe961ae01019j0s.html

a) **概述：**



Job 执行整体流程图

b) **Job 创建与提交过程**

1）Configuration 类：所有客户端程序中配置的类的信息和其他运行信息，都会保存在这个类里。

2）JobClient.runJob() 开始运行 job 并分解输入数据集

3）JobClient.submitJob()提交 job 到 JobTracker

c） Job 执行过程

job 统一由 JobTracker 来调度的，具体的 Task 分发给各个 TaskTracker 节点来执行。

1）JobTracker 初始化 Job 和 Task 队列过程

a) **JobTracker.submitJob() 收到请求**
b) **JobTracker.setJobPriority() 设置优先级**
c) **JobTracker.InitJob 通知初始化线程**
d) **JobInProgress.initTasks() 初始化 TaskInProgress**

2）TaskTracker 执行 Task 的过程

a) **TaskTracker.run() 连接 JobTracker**
b) **TaskTracker.offerService() 主循环**

c) **TaskTracker.transmitHeartBeat() 获取 JobTracker 指令**
d) **TaskTracker.startNewTask() 启动新任务**
e) **TaskTracker.localizeJob() 初始化 job 目录等**
f) **TaskTracker.launchTaskForJob() 执行任务**
g) **TaskTracker$TaskInProgress.launchTask() 执行任务**
h) **MapTask.run()与 ReduceTask.run();**

6、Hdfs 如何存储图片和视频

流的方式存储

Eg:公司的一个服务需要存储大量的图片服务器，考虑使用 hadoop 的 hdfs 来存放图片文件.以下是整个架构思路:

使用 hadoop 作为分布式文件系统，hadoop 是一个实现了 HDFS 文件系统和 MapReduce 的开源项目，我们这里只是

使用了它的 hdfs.首先从 web 页面上上传的文件直接调用 hadoop 接口将图片文件存入 hadoop 系统中，hadoop 可以设定备份

数，这样在 hadoop 系统中某个 datanode 死掉并不会造成图片不可能，系统会从其他 datanode 上拿到数据。以下我们编写的一个 hadoop 的 java 的访问封装类:

```
import java.io.File;
import java.io.IOException;
import java.io.InputStream;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FileUtil;
import org.apache.hadoop.fs.Path;
import org.apache.log4j.Logger;

public class HadoopFileUtil {
 static Logger logger = Logger.getLogger(HadoopFileUtil.class);
 /**
  * @param args
  */
 public static void main(String[] args) {

  String src=args[0];
  String dst=args[1];
  String tag=args[2];
  HadoopFileUtil util=new HadoopFileUtil();
```

```
  if(tag!=null&&tag.equals("1")){
   System.out.println(util.createFile(src, dst));
  }
  else{
   util.deleteFile(dst);
  }

}

/**
  *  拷贝一个本地文件到 hadoop 里面
  * @param localFile  本地文件和路径名
  * @param hadoopFile hadoop 文件和路径名
  * @return
  */
public   boolean createFile(String localFile,String hadoopFile){
 try {
  Configuration conf=new Configuration();
  FileSystem src=FileSystem.getLocal(conf);
  FileSystem dst= FileSystem.get(conf);
  Path srcpath = new Path(localFile);
  Path dstpath = new Path(hadoopFile);
  FileUtil.copy(src, srcpath, dst, dstpath,false,conf);
 } catch (Exception e) {
  e.printStackTrace();
  return false;
 }

 return true;
}


/**将一个流作为输入，生成一个 hadoop 里面的文件
  * @param inStream  输入流
  * @param hadoopFile hadoop 路径及文件名字
  * @return
  */
public boolean createFileByInputStream(InputStream inStream,String hadoopFile){
 try {
  Configuration conf=new Configuration();
  FileSystem dst= FileSystem.get(conf);
  Path dstpath = new Path(hadoopFile);
  FSDataOutputStream oStream=dst.create(dstpath);
  byte[] buffer = new byte[400];
  int length = 0;
```

```
  while((length = inStream.read(buffer))>0){
   oStream.write(buffer,0,length);
  }
  oStream.flush();
  oStream.close();
  inStream.close();
 } catch (Exception e) {
  e.printStackTrace();
  return false;
 }
 return true;
}
/**
  *  删除 hadoop 里面的一个文件
  * @param hadoopFile
  * @return
  */
public   boolean deleteFile(String hadoopFile){
 try {
  Configuration conf=new Configuration();
  FileSystem dst= FileSystem.get(conf);
  FileUtil.fullyDelete(dst,new Path(hadoopFile));
 } catch (Exception e) {
  e.printStackTrace();
  return false;
 }

 return true;
}
/**
  *  从 hadoop 中读取一个文件流
  * @param hadoopFile
  * @return
  */
public FSDataInputStream getInputStream(String hadoopFile){
 FSDataInputStream iStream=null;
 try {
  Configuration conf=new Configuration();
  FileSystem dst= FileSystem.get(conf);
  Path p=new Path(hadoopFile);
  iStream=dst.open(p);
 } catch (Exception e) {
  e.printStackTrace();
  logger.error("getInputStream error:", e);
```

```
  }
  return iStream;
 }

}
```
通过调用这个类可以将图片存入 hadoop 系统.

当需要访问某个图片时，先访问 jsp 服务器(如:tomcat)的一个 servlet，这个 servlet 从 hadoop 里面读出图片，并

返回给浏览器.以下是我们的 servlet:

```
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.io.IOUtils;
import org.apache.log4j.Logger;

import com.tixa.dfs.hadoop.util.HadoopFileUtil;

public class HadoopServlet extends javax.servlet.http.HttpServlet implements
javax.servlet.Servlet {
 static Logger logger = Logger.getLogger(HadoopServlet.class);

 public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException{

  PrintWriter out=res.getWriter();
  res.setContentType("image/jpeg");
  java.util.Date date = new java.util.Date();
  res.setDateHeader("Expires",date.getTime()+1000*60*60*24);
  String path=req.getPathInfo();
  path=path.substring(1,path.length());
  HadoopFileUtil hUtil=new HadoopFileUtil();
  FSDataInputStream inputStream=hUtil.getInputStream(path);
  OutputStream os = res.getOutputStream();

  byte[] buffer = new byte[400];
  int length = 0;
  while((length = inputStream.read(buffer))>0){
```

```
    os.write(buffer,0,length);
  }
  os.flush();
  os.close();
  inputStream.close();
 }

}
```

另外，为了避免对 hadoop 的频繁读取，可以再 jsp 服务器前放一个 squid 进行对图片的缓存。

这就是我们图片服务器的架构.

另外其实这里也可以用一些其他的分布式文件系统，如 kfs 等，很多分布式文件系统可能会比 hadoop 更为

稳定一些。

7、hadoop 怎么样实现二级排序

a) 出现的场景：

i. 当我们有对值进行排序的需求时，正常是在 reduce 阶段收集，但是当要处理的 values 过大时会导致内存溢出等问题，所有此时需要进行二次排序.

b) 解决方式：

在 mr 的 shuffle 阶段进行自定义 sort 排序

8、简述 hadoop 实现 join 的几种方法？

1) reduce side join

reduce side join 是一种最简单的 join 方式，其主要思想如下：

在 map 阶段，map 函数同时读取两个文件 File1 和 File2，为了区分两种来源的 key/value 数据对，对每条数据打一个标签 （tag）,比如：tag=0 表示来自文件 File1，tag=2 表示来自文件 File2。即：map 阶段的主要任务是对不同文件中的数据打标签。

在 reduce 阶段，reduce 函数获取 key 相同的来自 File1 和 File2 文件的 value list，然后对于同一个 key，对 File1 和 File2 中的数据进行 join（笛卡尔乘积）。即：reduce 阶段进行实际的连接操作。

2) map side join

之所以存在 reduce side join，是因为在 map 阶段不能获取所有需要的 join 字段，即：同一个 key 对应的字段可能位于不同 map 中。Reduce side join 是非常低效的，因为 shuffle 阶段要进行大量的数据传输。

Map side join 是针对以下场景进行的优化：两个待连接表中，有一个表非常大，而另一个表非常小，以至于小表可以直接存放到内存中。这样，我们可以将小表复制多 份，让每个 map task 内存中存在一份（比如存放到 hash table 中），然后只扫描大表：对于大表中的每一条记录 key/value，在 hash table 中查找是否有相同的 key 的记录，如果有，则连接后输出即可。

为了支持文件的复制，Hadoop 提供了一个类 DistributedCache，使用该类的方法如下：

（1）用户使用静态方法 DistributedCache.addCacheFile()指定要复制的文件，它的

参数是文件的 URI（如果是 HDFS 上的文件，可以这样：hdfs://namenode:9000/home/XXX/file，其中 9000 是自己配置的 NameNode 端口号）。JobTracker 在作业启动之前会获取这个 URI 列表，并将相应的文件拷贝到各个 TaskTracker 的本地磁盘上。（2）用户使用 DistributedCache.getLocalCacheFiles()方法获取文件目录，并使用标准的文件读写 API 读取相应的文件。

3) SemiJoin

SemiJoin，也叫半连接，是从分布式数据库中借鉴过来的方法。它的产生动机是：对于 reduce side join，跨机器的数据传输量非常大，这成了 join 操作的一个瓶颈，如果能够在 map 端过滤掉不会参加 join 操作的数据，则可以大大节省网络 IO。

实现方法很简单：选取一个小表，假设是 File1，将其参与 join 的 key 抽取出来，保存到文件 File3 中，File3 文件一般很小，可以放到内存中。在 map 阶段，使用 DistributedCache 将 File3 复制到各个 TaskTracker 上，然后将 File2 中不在 File3 中的 key 对应的记录过滤掉，剩下的 reduce 阶段的工作与 reduce side join 相同。

4) reduce side join + BloomFilter

在某些情况下，SemiJoin 抽取出来的小表的 key 集合在内存中仍然存放不下，这时候可以使用 BloomFiler 以节省空间。

BloomFilter 最常见的作用是：判断某个元素是否在一个集合里面。它最重要的两个方法是：add() 和 contains()。最大的特点是不会存在 false negative，即：如果 contains()返回 false，则该元素一定不在集合中，但会存在一定的 true negative，即：如果 contains()返回 true，则该元素可能在集合中。

因而可将小表中的 key 保存到 BloomFilter 中，在 map 阶段过滤大表，可能有一些不在小表中的记录没有过滤掉（但是在小表中的记录一定不会过滤掉），这没关系，只不过增加了少量的网络 IO 而已。

10、请简述 mapreduce 中，combiner，partition 作用？

InputFormat 类：该类的作用是将输入的文件和数据分割成许多小的 split 文件，并将 split 的每个行通过 LineRecorderReader 解析成<Key,Value>，通过 job.setInputFromatClass()函数来设置，默认的情况为类 TextInputFormat，其中 Key 默认为字符偏移量，value 是该行的值。

Map 类：根据输入的<Key,Value>对生成中间结果，默认的情况下使用 Mapper 类，该类将输入的<Key,Value>对原封不动的作为中间按结果输出，通过 job.setMapperClass()实现。实现 Map 函数。

Combine 类：实现 combine 函数，该类的主要功能是合并相同的 key 键，通过 job.setCombinerClass()方法设置，默认为 null，不合并中间结果。实现 map 函数

Partitioner 类： 该该主要在 Shuffle 过程中按照 Key 值将中间结果分成 R 份，其中每份都有一个 Reduce 去负责，可以通过 job.setPartitionerClass()方法进行设置，默认的使用 hashPartitioner 类。实现 getPartition 函数

Reducer 类：将中间结果合并，得到中间结果。通过 job.setReduceCalss()方法进行设置，默认使用 Reducer 类，实现 reduce 方法。

OutPutFormat 类：该类负责输出结果的格式。可以通过 job.setOutputFormatClass()

方法进行设置。默认使用 TextOUtputFormat 类，得到<Key,value>对。
note：hadoop 主要是上面的六个类进行 mapreduce 操作，使用默认的类，处理的数据和文本的能力很有限，具体的项目中，用户通过改写这六个 类（重载六个类），完成项目的需求。说实话，我刚开始学的时候，我怀疑过 Mapreudce 处理数据功能，随着学习深入，真的很钦佩 mapreduce 的 设计，基本就二个函数，通过重载，可以完成所有你想完成的工作。

11、用 mapreduce 怎么处理数据倾斜问题？

map /reduce 程序执行时，reduce 节点大部分执行完毕，但是有一个或者几个 reduce 节点运行很慢，导致整个程序的处理时间很长，这是因为某一个 key 的条数比其他 key 多很多（有时是百倍或者千倍之多），这条 key 所在的 reduce 节点所处理的数据量比其他节点就大很多，从而导致某几个节点迟 迟运行不完，此称之为**数据倾斜.**

解决方法：

(1)设置一个 hash 份数 N，用来对条数众多的 key 进行打散。

(2)对有多条重复 key 的那份数据进行处理：从 1 到 N 将数字加在 key 后面作为新 key，如果需要和另一份数据关联的话，则要重写比较类和分发类。如此实现多条 key 的平均分发。

(3)上一步之后，key 被平均分散到很多不同的 reduce 节点。如果需要和其他数据关联，为了保证每个 reduce 节点上都有关联的 key，对另一份单一 key 的数据进行处理：循环的从 1 到 N 将数字加在 key 后面作为新 key

用上述的方法虽然可以解决数据倾斜，但是当关联的数据量巨大时，如果成倍的增长某份数据，会导致 reduce shuffle 的数据量变的巨大，得不偿失，从而无法解决运行时间慢的问题.

在 两份数据中找共同点，比如两份数据里除了关联的字段以外，还有另外相同含义的字段，如果这个字段在所有 log 中的重复率比较小，则可以用这个字段作为计算 hash 的值，如果是数字，可以用来模 hash 的份数，如果是字符可以用 hashcode 来模 hash 的份数（当然数字为了避免落到同一个 reduce 上 的数据过多，也可以用 hashcode），这样如果这个字段的值分布足够平均的话，就可以解决上述的问题。

解决方法：http://my.oschina.net/leejun2005/blog/100922

1. 增加 reduce 的 jvm 内存
2. 增加 reduce 个数
3. customer partition
4. 其他优化的讨论.
5. reduce sort merge 排序算法的讨论
6. 正在实现中的 hive skewed join.
7. pipeline
8. distinct

9. index 尤其是 bitmap index

12、Hive：

本质：将 SQL 转换为 MapReduce 程序

Hive 重点：搭建、写 hive 的脚本语句(DDL<创建表、创建数据库----> (Data Definition Language, DDL)数据定义语言>、DML<*导入表(load data)----(insert into 表 select * from)*、删除表---->（Data Manipulation Language）数据操作语言>)

为什么使用 hive：

- 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，减少开发人员的学习成本
- 扩展功能很方便

Hive 执行脚本的三种方式：

1. hive -e 执行一行脚本
2. Hive -f 执行一个脚本文件
3. Hive jdbc 执行代码脚本

Hive 与 DBMS 区别：

存储文件系统不同：hive 存在 mr 的 hdfs 上，dbms 存在服务器的本地文件系统上.

Hive 使用 mr 的计算模型，dbms 使用自己设计的计算模型

Hive 实时性很差，dbms 实时性很好

Hive 容易扩展，dbms 不易扩展

Hive 自定义函数的三种类型：

UDF：一对一

UDAF：多对一(count、max)

UDTF：一对多(实现 initialize, process, close 三个方法)

13、Hive 元数据的三种存储方式

1） **默认存储(Single User Mode：derby)：**默认安装 hive，hive 是使用 derby 内存数据库保存 hive 的元数据，这样是不可以并发调用 hive 的.

2） **本地存储(Multi User Mode)：**通过网络连接到一个数据库中，是最经常使用到的模式。假设使用本机 mysql 服务器存储元数据。这种存储方式需要在本地运行一个 mysql 服务器，并作如下配置（需要将 mysql 的 jar 包拷贝到$HIVE_HOME/lib 目录下）.

3） **远程存储(Remote Server Mode)：** 在服务器端启动一个 MetaStoreServer，客户端利用 Thrift 协议通过 MetaStoreServer 访问元数据库.

14、Hive 内部表和外部表的区别？

1、在导入数据到外部表，数据并没有移动到自己的数据仓库目录下，也就是说外部表中的数据并不是由它自己来管理的！而表则不一样；

2、在删除表的时候，Hive 将会把属于表的元数据和数据全部删掉；而删除外部表的时候，Hive 仅仅删除外部表的元数据，数据是不会删除的！

那么，应该如何选择使用哪种表呢？在大多数情况没有太多的区别，因此选择只是个人喜好的问题。但是作为一个经验，如果所有处理都需要由 Hive 完成，那么你应该创建表，否则使用外部表！

15、Hive 创建自定义函数流程

a） 自定义 final 类

b) 继承 UDF
c) 写方法 evaluate(方法的参数和返回值类型必须是 MR 类型)
d) 写好后把该类打成 jar 包
e) 上传到 hive 的 lib 包下
f) 在虚拟机上启动 hive、
g) 去创建这个函数(create temporary function 自定义函数名 as ‘类名<全限定名>(com.example.hive.udf.Lower)’ ;)
Eg:create temporary function my_lower as ’com.example.hive.udf.Lower’;
h) 在 hql 中使用该自定义函数
select my_lower(title), sum(freq) from titles group by my_lower(title);

16、Hive 优化：

- 减少 job 数
- 对小文件进行合并，是行至有效的提高调度效率的方法
- 优化时把握整体，单个作业最优不如整体最优
- 数据量较大的情况下，慎用 count(distinct)

Hive 的数据类型方面的优化：

•按照一定规则分区（例如根据日期）。通过分区，查询的时候指定分区，会大大减少在无用数据上的扫描，同时也非常方便数据清理

合理的设置 Buckets。在一些大数据 join 的情况下，map join 有时候会内存不够。如果使用 Bucket Map Join 的话，可以只把其中的一个 bucket 放到内存中，内存中原来放不下的内存表就变得可以放下。这需要使用 buckets 的键进行 join 的条件连结，并且需要如下设置

set hive.optimize.bucketmapjoin = true

Hive 的操作方面的优化:

•控制 Hive 的 Map 数

•设定合理的 reducer 数量--启动 reduce 会消耗实际和资源

合并 MapReduce 操作：

JOIN 原则：小的文件放 join 左边，左边的会加载到内存

合并小文件：hadoop 烦过多的小文件

数据倾斜：用 partition 做分区，增加多个 reduce tasker 任务,partition 将数据分成多个目录，加快查询的速度

17、Hbase 的 rowkey 怎么创建比较好？列族怎么创建比较好？

HBase 是一个分布式的、面向列的数据库，它和一般关系型数据库的最大区别是：HBase 很适合于存储非结构化的数据，还有就是它基于列的而不是基于行的模式.

HBase 的 RowKey 设计：

1 设计原则

1.1 Rowkey 长度原则

Rowkey 是一个二进制码流，Rowkey 的长度被很多开发者建议说设计在 10~100 个字节，不过建议是越短越好，不要超过 16 个字节。

原因如下：

（1）数据的持久化文件 HFile 中是按照 KeyValue 存储的，如果 Rowkey 过长比如 100 个字节，1000 万列数据光 Rowkey 就要占用 100*1000 万=10 亿个字节，将近 1G 数据，这会极大影响 HFile 的存储效率；

（2）MemStore 将缓存部分数据到内存，如果 Rowkey 字段过长内存的有效利用率会降低，系统将无法缓存更多的数据，这会降低检索效率。因此 Rowkey 的字节长度越短越好。

（3）目前操作系统是都是 64 位系统，内存 8 字节对齐。控制在 16 个字节，8 字节的整数倍利用操作系统的最佳特性。

1.2 Rowkey 散列原则

如果 Rowkey 是按时间戳的方式递增，不要将时间放在二进制码的前面，建议将 Rowkey 的高位作为散列字段，由程序循环生成，低位放时间字段， 这样将提高数据均衡分布在每个 Regionserver 实现负载均衡的几率。如果没有散列字段，首字段直接是时间信息将产生所有新数据都在一个 RegionServer 上堆积的热点现象，这样在做数据检索的时候负载将会集中在个别 RegionServer，降低查询效率。

1.3 Rowkey 唯一原则

必须在设计上保证其唯一性。

2 应用场景

2.1 针对事务数据 Rowkey 设计

事务数据是带时间属性的，建议将时间信息存入到 Rowkey 中，这有助于提示查询检索速度。对于事务数据建议缺省就按天为数据建表，这样设计的好处是多方 面的。按天分表后，时间信息就可以去掉日期部分只保留小时分钟毫秒，这样 4 个字节即可搞定。加上散列字段 2 个字节一共 6 个字节即可组成唯一 Rowkey。

2.2 针对统计数据的 Rowkey 设计

统计数据也是带时间属性的，统计数据最小单位只会到分钟（到秒预统计就没意义了）。同时对于统计数据我们也缺省采用按天数据分表，这样设计的好处无需多 说。按天分表后，时间信息只需要保留小时分钟，那么 0~1400 只需占用两个字节即可保存时间信息。由于统计数据某些维度数量非常庞大，因此需要 4 个字节 作为序列字段，因此将散列字段同时作为序列字段使用也是 6 个字节组成唯一 Rowkey。

2.3 针对通用数据的 Rowkey 设计

通用数据采用自增序列作为唯一主键，用户可以选择按天建分表也可以选择单表模式。这种模式需要确保同时多个入库加载模块运行时散列字段（序列字段）的唯一性。可以考虑给不同的加载模块赋予唯一因子区别。

2.4 支持多条件查询的 RowKey 设计

HBase 按指定的条件获取一批记录时，使用的就是 scan 方法。 scan 方法有以下特点：

（1）scan 可以通过 setCaching 与 setBatch 方法提高速度（以空间换时间）；

（2）scan 可以通过 setStartRow 与 setEndRow 来限定范围。范围越小，性能越高。

通过巧妙的 RowKey 设计使我们批量获取记录集合中的元素挨在一起（应该在同一个 Region 下），可以在遍历结果时获得很好的性能。

（3）scan 可以通过 setFilter 方法添加过滤器，这也是分页、多条件查询的基础。

在满足长度、三列、唯一原则后，我们需要考虑如何通过巧妙设计 RowKey 以利用 scan 方法的范围功能，使得获取一批记录的查询速度能提高。下例就描述如何将多个列组合成一个 RowKey，使用 scan 的 range 来达到较快查询速度。

18、HBASE 列族设计：

1、列簇的设计需要根据你的业务。那些可能被反复修改的数据表尽量使用单列簇。每个列簇在 HDFS 都有一个独立的 HFILE，当某个 ROWKEY 的某个列 簇数据被冲刷时，这个 ROWKEY 连带的其他列簇数据也会被一起冲刷，I/O 负担很大。APACHE 官方也提倡多列簇的设计方案，单列簇性能是最高的。
而持久型数据，也就是一次写入，从不修改的数据，可以使用多列簇，原理相同，但目前任然提倡单列簇设计模式

2、多列簇的效率问题参照 1

3、所谓列簇分组，就相当于关系习惯数据库中，两个表被纵向合并，形成一张双列簇的表

19、HBase 与关系型数据库的区别以及 hbase 的优化：

区别：

1.Hbase 是面向列存储的分布式存储系统，它的优点在于可以实现高性能的并发读写操作，同时 Hbase 还会对数据进行透明的切分，这样就使得存储本身具有了水平伸缩性。

RDBMS 是面向行存储，主要适合于事务性要求严格场合，或者说面向行存储的存储系统适合 OLTP，但是根据 CAP 理论，传统的 RDBMS，为了实现强一致性，通过严格的 ACID 事务来进行同步，这就造成了系统的可用性和伸缩性方面大大折扣，而目前的很多 NoSQL 产品，包括 Hbase，它们都是一种最终一致性的系统，它们为了高 的可用性牺牲了一部分的一致性。

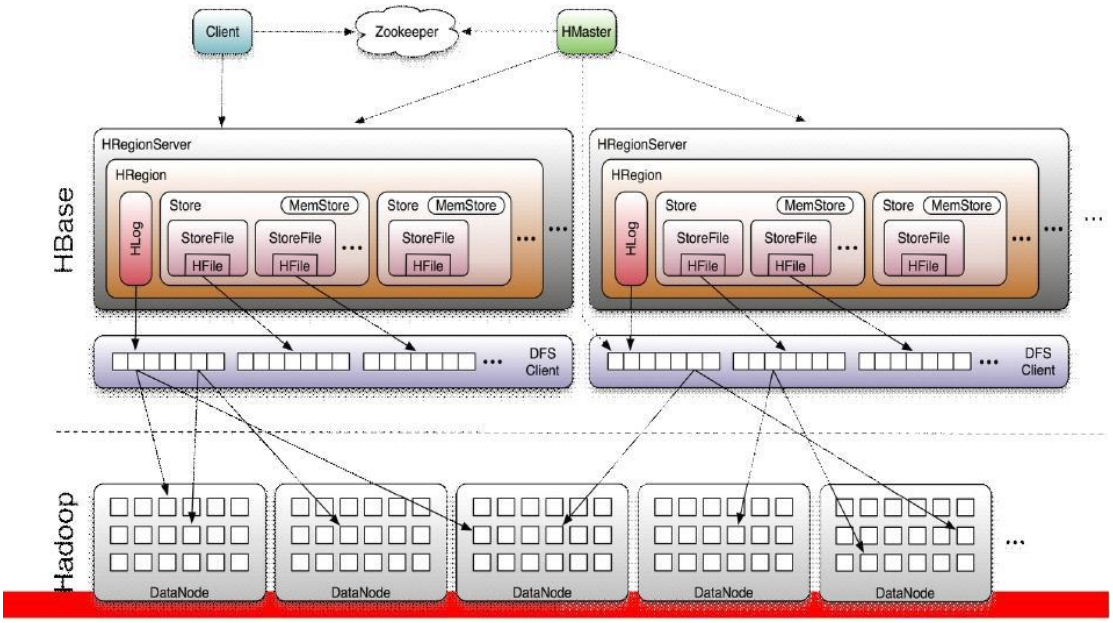
2.关系型数据存储数据时，哪怕值为空也是占用空间的，但 hbase 不会，对于内容为空的数据 hbase 会不存储，相比之下节省了空间

3.处理数据级别不同：hbase-->TB 级，RDBMS-->GB 级

Hbase 的优缺点：

优点：
1 列的可以动态增加，并且列为空就不存储数据,节省存储空间.
2 Hbase 自动切分数据，使得数据存储自动具有水平 scalability.
3 Hbase 可以提供高并发读写操作的支持

缺点：
1 不能支持条件查询，只支持按照 Row key 来查询.
2 暂时不能支持 Master server 的故障切换,当 Master 宕机后,整个存储系统就会挂掉.



Hbase 优化：

1. **表的设计**：

1 默认是一个 region 分区，所有的数据都存在该分区内，当足够大时才进行切分，一种加快写入的方法，是提前创建一些空的 regions，存储时分不同的 region 存储，在集群内做负载均衡

2. rowkey 不宜过长，建议不超过 16 个字节

3. 列族不宜太多，现在的 hbase 不适合处理超过 2-3 个列族的表：由于 flush 时会引起连锁效应，导致 IO 过多

4. In memory

创建表的时候，可以通过 HColumnDescriptor.setInMemory(true)将表放到 RegionServer 的缓存中，保证在读取的时候被 cache 命中。

5. Max Version<最大版本数>

6. Time To Live<存活时间>

7. Compact & Split< 裂变>

2. **写表操作**:

创建多个 htable 客户端用于写操作，提高写数据的吞吐量

2.1 参数设置 htable

1）auto flush：设置 htable 写的大小，可以一次性写入 hbase，不需一条一条处理

2）write buffer：设置每次写入的大小，如果文件小于该值则一次性写入

3）WAL fl 复，单对于不重要的数据在 put 和 delete 时可以省略先写入 hloag：默认是先入 hlog，再写入 memstore，以便宕机时数据恢

2.2. 批量写：HTable.put(Put)《-----》HTable.put(List<Put>)，针对实时性要求高的

2.3. 多线程并发写：

在客户端开启多个 HTable 写线程，每个写线程负责一个 HTable 对象的 flush 操作，这样结合定时 flush 和写 buffer（writeBufferSize），可以既保证在数据量小的时候，数据可以在较短时间内被 flush（如 1 秒内），同时又保证在数据量

大的时候，写 buffer 一满就及时进行 flush。

3. **读表操作：**

3.1 多个 tables 并发读

1）配置一次性抓取的数据条数，可减少 scan 的 next()开销

2）scan 时指定列族，否则返回全部

3）读取后关闭 result scanner，否则易出现对应的资源无法释放等问题

3.2 批量度

3.3 多线程并发读

4. 缓存查询结果

5. blockcache：hbase 上两个缓存：memstore 用于写，blockcache 用于读

20、Hbase 体系架构：

Client：包含访问 HBase 的接口并维护 cache 来加快对 HBase 的访问

Zookeeper：

- 保证任何时候，集群中只有一个 master
- 存贮所有 Region 的寻址入口。
- 实时监控 Region server 的上线和下线信息。并实时通知 Master
- 存储 HBase 的 schema 和 table 元数据

Master：

- 为 Region server 分配 region
- 负责 Region server 的负载均衡
- 发现失效的 Region server 并重新分配其上的 region
- 管理用户对 table 的增删改操作

RegionServer：

- Region server 维护 region，处理对这些 region 的 IO 请求
- Region server 负责切分在运行过程中变得过大的 region

当一个 region 所有 storefile 的大小和超过一定阈值后，会把当前的 region 分割为两个，并由 hmaster 分配到相应的 regionserver 服务器，实现负载均衡

21、数据存不进去了(HBase)

1.datanode 不够；

2.hbase 未做优化，可能正在做负载均衡<把一个 regionserver 上的数据移动到其他的 regionserver 上>《解决方案：在表的设计时使用预分区》；

3.hbase 在做合并

22、Redis：

基于内存的数据库!!!<如何持久化：将内存数据保存到硬盘，保证数据安全，方便进行数据的备份和恢复>读写数据不会受到磁盘 IO 的限制

独特的键值对模型：支持丰富的数据结构（除字符串外还有 5 种数据结构<**列表、散列、集合、有序集合、HyperLogLog**>）

过期键功能：为键设置一个过期时间，让其在指定时间后自动删除事物功能：使用乐观

锁

23、Storm：开源的分布式实时流式计算系统

Strom 架构：

Nimbus

Supervisor

Worker

编程模型：

DAG

Sqout

Bolt

数据传输：

Zmq

Netty

Storm 实时低延迟的原因

1.进程是常驻内存的，不像 hadoop 要反复启停，所以没有不断启停的开销

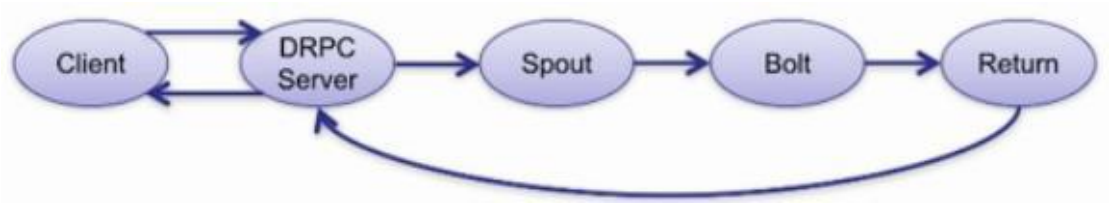
2.数据不经过磁盘，都在内存中，处理完就没有了；数据交互经过网络，避免了磁盘 IO 开销

实时请求应答服务(同步)：

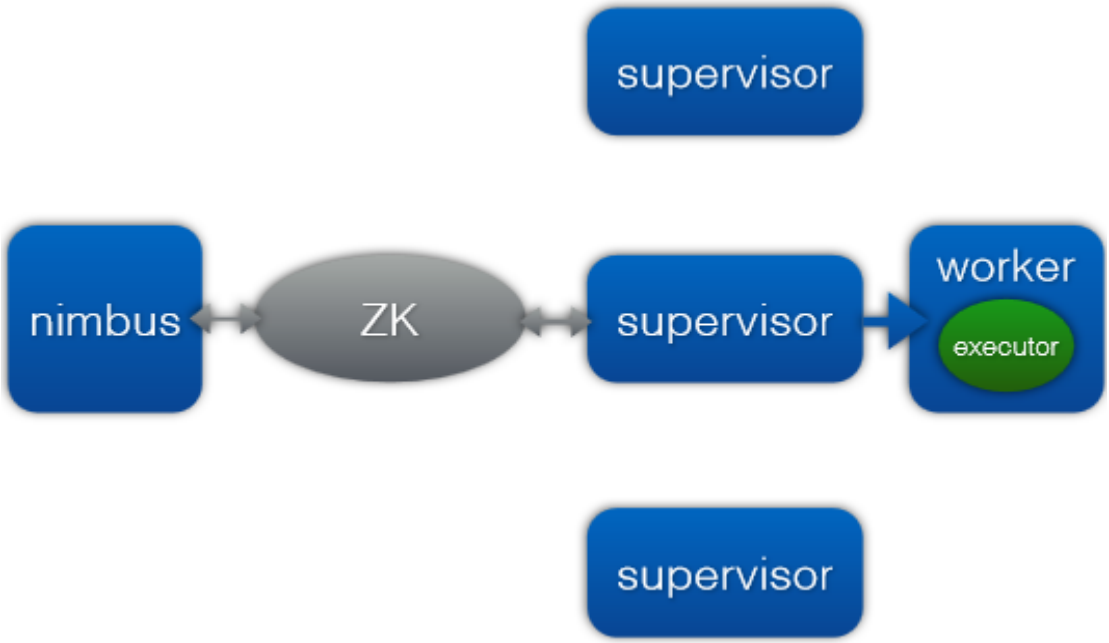
使用 DAG 模型提高请求处理速度<DRPC Server>

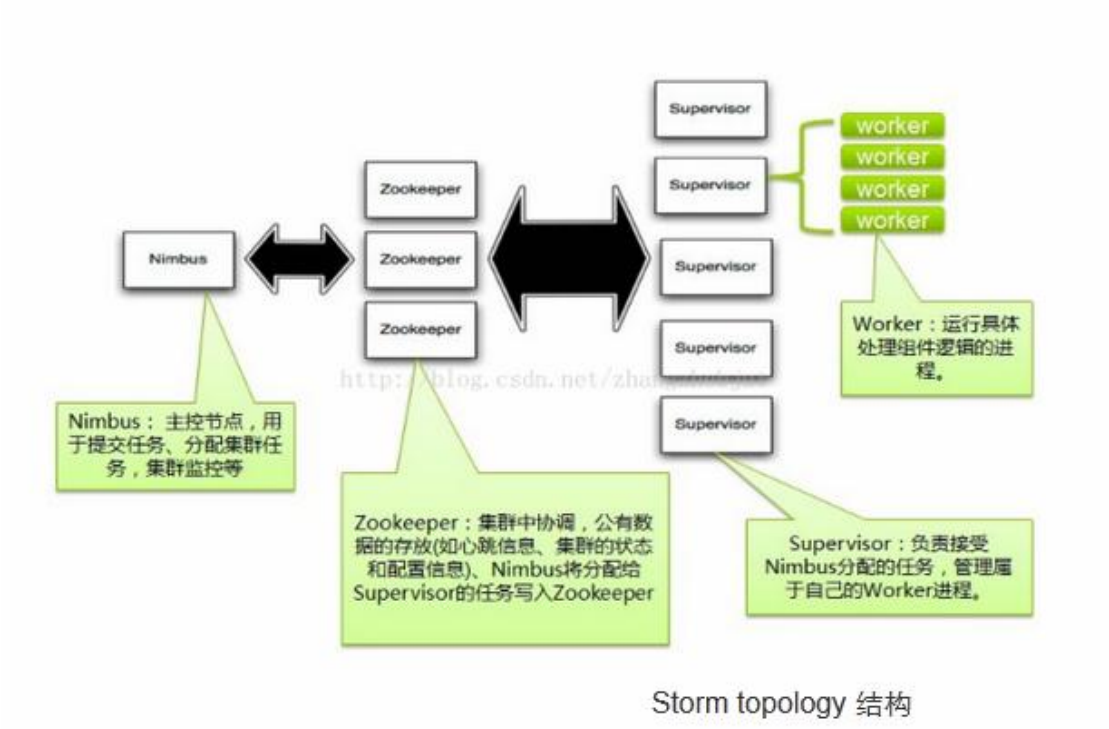
服务器四部分组成以及流程：

客户端--->DRPC Server--->DRPC Spout--->Topology--->ReturnResult



系统架构：





Storm topology 结构

架构特点：

无状态：nimbus/supervisor/worker 状态全存在 zk 中，谁挂掉都无所谓

单中心：nimbus 为主，一个人决定不需要仲裁；

单点失效问题：nimbus 和一个 supervisor 挂掉，没 nimbus 不会做重新调度

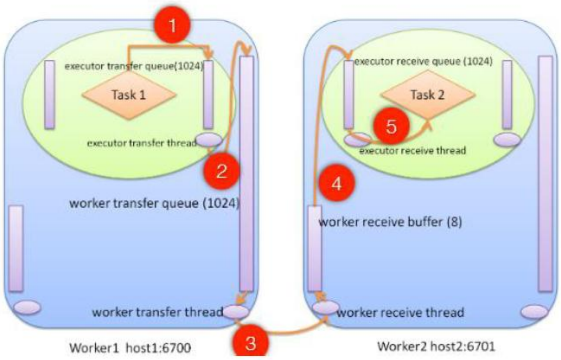
单点性能问题：所有 topology jar 包都通过 nimbus 分发，supervisor 去 nimbus 上下载，supervisor 多了，nimbus 会成为瓶颈，网卡很卡被打满，可以使用**把文件分发变成 p2p** 的

隔离性好：真正干活的在 executor 和 worker 中，nimbus 和 supervisor 只起到控制 topology 流程的作用，nimbus 和 supervisor 挂了无所谓，相比之下 hadoop 所有 tasktracker 存在，所有 shuffle 都经过 tasktracker，当其挂掉后，所有 map 需

要　　重新去执行

- **Nimbus**
  - **集群管理**,作为整个集群的 master，检查 supervisor/worker 是否正常，去 zk 中检查心跳是否正常就可以
  - **调度 topology**，将调度信息写入 zk
  - **作为接口处理 topology submit、kill、reblance 等请求**
  - **提供查询 cluster/topology 状态的 thrift 接口**---cluster 有多少机器，topology 状态，topology 有多少个 worker，有多少个 executor，worker 和 execu 运行的状态，例如在 webUI 通过 thrift 来调用 nimbus 的查询接口，展现在 web 上，可以用于做监控
  - **提供文件(如 topology 程序的 jar 包)上传下载服务**
    nimbus 提供该服务，通过 thrift 接口实现
- **Supervisor**
  - **启停 worker**：主要做两件事--下载 topology 的 jar 包，再解压，解压后创建相应的目录，以便后面使用这个 jar
  - **监控 worker**：worker 启动后会在本地写一些监控信息，supervisor 查看这些信息，长时间没消息则认为有问题了
  - **把自己的情况汇报出去**：默认 5 秒在 zk 上更新一次 supervisor 的心跳，为了告诉 nimbus 表明 supervisor 没问题
- **Worker**
  - **一个 JVM 进程资源分配的单位**
  - **启动 executor**，executor 真正的执行 task，运行 spout 和 bolt 代码；一个 worker 可能会启动多个 executor，多个 executor 必须属于一个 topology
  - **负责 worker 和 worker 之间的数据传输**；一个 topology 的 worker 之间建立网络--每个 worker 都有一个收发线程，进行实际数据 tuple 的收发工作，每个线程还有个 buffer，一个收 buffer，一个发送 buffer，来做数据缓冲，不同的 topology 之间没有网络连接
  - **把自己的信息定期往本地系统写一份，再往 zk 上写一份**；在本地会写个 WorkerHeartbeat 的数据结构--记录写心跳的时间，和 topology-id，worker 里面对应哪些 executor-id，worker 对应的是哪个端口；
- **Executor**
  - **实际干活的线程**
  - **去创建实际的 spout/bolt 的对象**
  - **创建执行线程**：执行 nextTuple()和 execute()两个回调函数，spout 的 nextTuple 是死循环，里面有个 sleep，当上一次调用没数据时 sleep 一会，或者当 spoutnextpending 到达，但还有数据没处理完时，做流控，睡一会再调用 nextTuple();Boltd 的 execute()是当收到一个新来的 Tuple 时才去调用
  - **创建传输线程**：负责将新产生的 tuple 放到 worker 的传输队列中，worker 是数据最后传输的地方，executor 本身不会做数据传输，只会处理完数据后把数据放到内存里

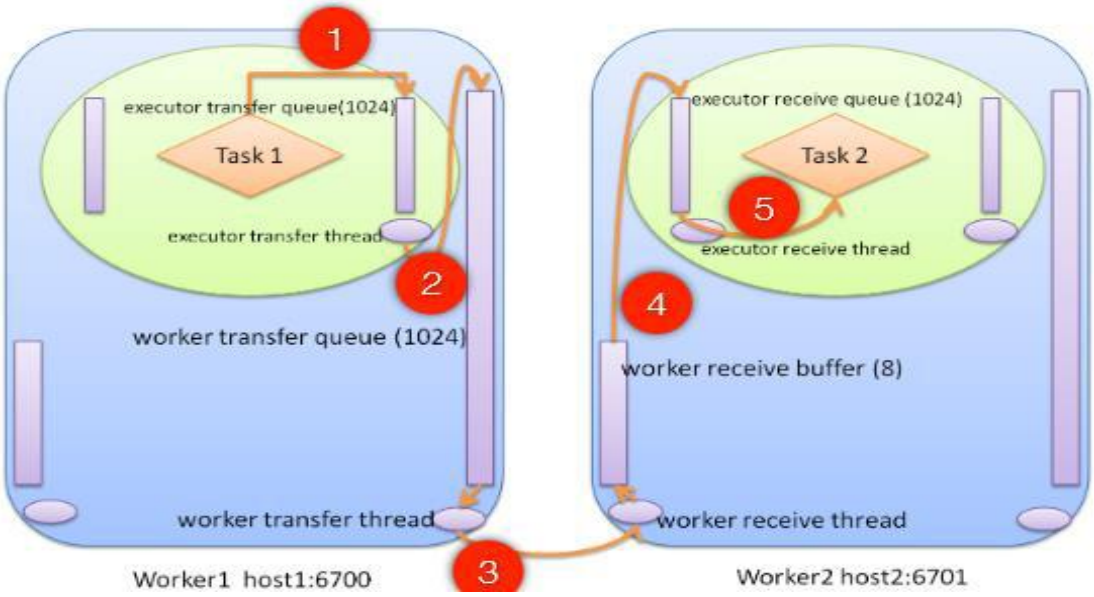
- 1.executor执行spout.nextTuple()/bolt.execute(),emit tuple放入executor transfer queue
- 2.executor transfer thread把自己transfer queue里面的tuple放到worker transfer queue
- 3.worker transfer thread把transfer queue里面的tuple序列化发送到远程的worker
- 4.worker receiver thread分别从网络收数据，反序列化成tuple放到对应的executor的receive queue
- 5.executor receive thread从自己的receive queue取出tuple，调用bolt.execute()



第三步找到对应的 worker receive thread 是通过 zk 来找的

第四步找到 worker 中的 task 是通过序列化数据里面找到的，序列化有两部分：task-id 和 tuple 本身数据

worker 内部的 task 传输不需要网络



•Zookeeper：

1.**存储心跳**(supervisor 心跳，worker 心跳)，把心跳信息写进去，等读取的人去 zookeeper 上读取信息

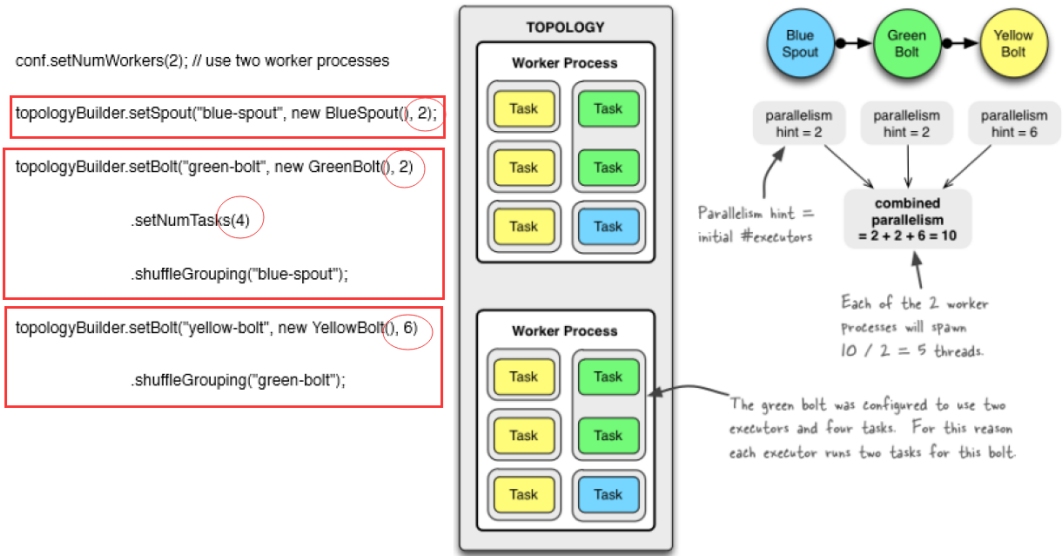
2.**存储调度信息**(topology 基本信息<name，id，状态的信息，几个 worker，component 有哪些个 executors>、topology 调度信息--topology 分配了多少个 worker，每个 worker 的 id 号，worker 分配到哪些机器上等信息；**supervisor ID+worker 的 port 确定唯一一个 worker**)、错误信息(task 执行的错误信息--spout/bolt 错误信息---》每个 component 就是一个 spout/bolt，最多写最近 20 条信息，防止往 zk 里面存储过多的数据，导致 zk 压力太大，毕竟 zk 不是专门用来存储数据的)，这样的话，在 webUI 上就可以读取该信息，把错误显示出来

并发模型：

Hadoop 就两级，map task 和 reduce task，task 中运行的就是 mapreduce 函数

Storm 大到小有 5 个层次：cluster--supervisor--worker--executor--task

一个 worker 对应一个进程



执行访问端口 http://zkhp01:8080/index.html

部署：

dataDir=/zookeepertest/data：zk 所有数据写在该目录下

autopurge.purgeInterval=1：每隔一小时，它来清理 dataDir 里面的数据；是因为 ZK 会产生 snapshot 和 binlog，产生的速度非常快，用不了几天就会把磁盘给打满，我们一小时清理一次就可以有效的避免这个问题，清理的规则是清理的时候它会保留最新的 3 个文件，当然这个 3 也是可以配置的

默认接收客户端端口是 2181

启动流程：

1.启动 ZK

2.启动 nimbus

3.启动 UI

4.启动 supervisor

5.启动 logviewer

6.打开 ui 界面 8080 验证

7.启动 topology

启动 topology

```
#  ./src/storm/strom-core/src/py/storm/distributedrpc-remote -h localhost:3772
-f execute exclaim test
```

1.Ispout 中 nextTuple()循环回调，没有输出或 max.pending 到达时触发 sleep

2.Tuple 处理完时调用 ack()

3.Tuple 超时或提前失败时调用 fail()

上面的三个回调函数任意时刻只有一个会被调用

生成 tuple 后，用 collector.emit 输出，spout 调用 nextTuple()后会睡 10 毫秒，避免死循环

如果不流控会出现 bolt 的 OOM 问题

Spout 中 open()为初始化方法，activate()和 deactivate()方法在 topology 调用这些方法的时候 spout 里面的方法才会被调用，譬如 deactivate()就会暂时不去数据源里拿数据了，activate()就又会继续去数据源拿数据了

ISpoutOutputCollector

emit 有三个参数，第一个是 Id，默认是 default 输出流，第二个是 tuple，输出的数据，后面我们会讲怎么生成 tuple，messageId 是后面 fail 或者 ack 的时候要使用的 messageId，其实 emit 还有一个变种就是如果第一个参数没有的话，就会自动填入 default

Bolt 核心是 execute()，还有两个 prepare 和 cleanup 方法

Execute 是来一个处理一个

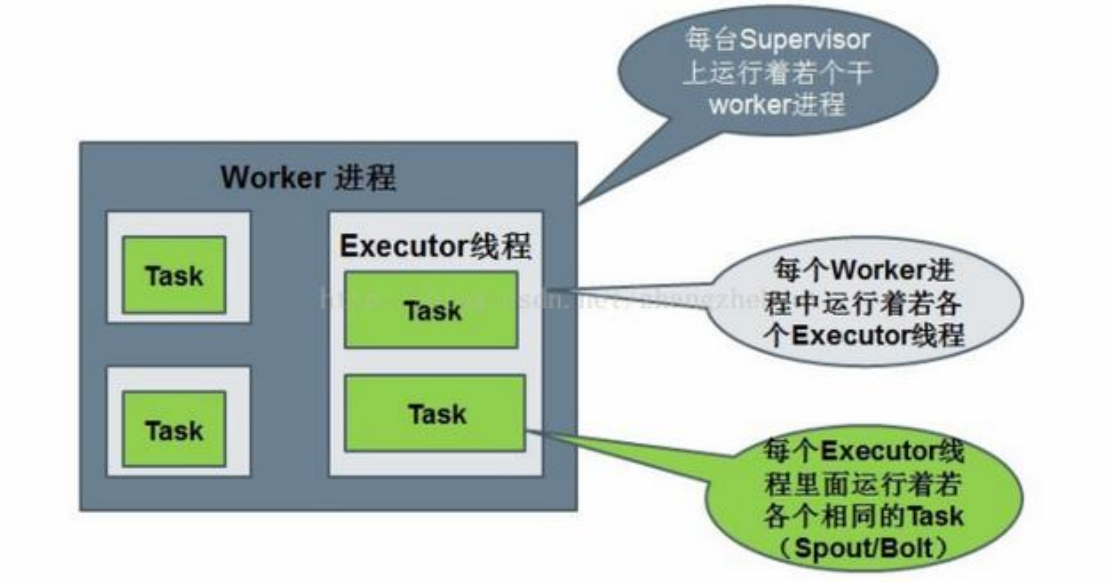
获取 tuple 的数据方式

1.通过下标：tuple.getString(0);

2.通过名字：tuple.getStringByFields("word");

Topology：

运行中的Topology主要由以下三个组件组成的： Worker processes、Executors threads以及 Tasks



集群中如何运行 topology：

1.定义 topology(java--用 TopologyBuilder**)**

2. 使用 StormSubmitter 来把 topology 提交到集群。StormSubmitter 的参数有：topology 的名字，topology 的配置对象，以及 topology 本身

StormSubmitter.submitTopology("name", conf, topology);

3.创建一个包含你的程序代码以及你代码所依赖的依赖包的 jar 包

4.用 storm 客户端去提交 jar 包

storm jar allmycode.jar org.me.MyTopology arg1 arg2 arg3

Config.TOPOLOGY_MAX_SPOUT_PENDING: 这个设置一个 spout task 上面最多有多少个没有处理的 tuple（没有 ack/failed）回复， 我们推荐你设置这个配置，以防止 tuple 队列爆掉。

终止一个 topology

Storm kill {stormname}

其中{stormname}是提交 topology 给 storm 集群的时候指定的名字

更新一个运行中的 topology

为了更新一个正在运行的 topology, 唯一的选择是杀掉正在运行的 topology 然后重新提交一个新的。一个计划中的命令是实现一个 storm swap 命令来运行时更新 topology， 并且保证前后两个 topology 不会同时在运行，同时保证替换所造成的“停机”时间最少。

Storm 支持的组分配策略如下：

ShuffleGrouping ： 随机选择一个 Task 来发送。

FiledGrouping： 根据 Tuple 中 Fields 来做一致性 hash，相同 hash 值的 Tuple 被发送到相同的 Task。

DRPC：rpc(远程过程调用协议)请求流式、并进行处理，请求参数当输入流，结果当输出流

Storm 只能获取数据，不能接请求和发响应，所以这里借助一个 DRPC Server 来帮助完成

DRPC 把大量请求分布式的去做，一次请求如果串行的话可能会比较慢，我并行的来处理，另一方面通过来降低平均一次请求的时间，解决了响应的吞吐

Kafka+Strom:

为什么使用 kafka：

Storm 没有自己的接收器、没有自己的存储，放到 KV 里面去

在写入数据库之前先放到缓冲区，避免大量数据直接写入数据时发生堵塞

Kafka：消费者生产者模式

Kafka 架构：

producer：生产者

Sonsumer：消费者

Broker：kafka 集群的 server，负责处理消息读、写请求，存储消息

Topic：消息队列/分类

Queue：里面有生产者消费者模型

Kafka 的消息存储和生产消费模型

一个 topic 分成多个 partition

每个 partition 内部都有序，其中每个消息有个序号叫 offset

一个 partition 只对应一个 broker

一个 broker 可以管多个 partition

消息不经过内存缓冲，直接写入文件

根据实际策略删除，而不是消费完就删除

Producer 自己决定往哪个 partition 中写消息

Kafka 精髓：

kafka 里面的消息是有 topic 来组织的，简单的我们可以想象为一个队列，一个队列就是一个 topic，然后它把每个 topic 又分为很多个 partition，这个是为了做并行的，在每个 partition 里面是有序的，相当于有序的队列，其中每个消息都有个序号，比如 0 到 12，从前面读往后面写，

一个 partition 对应一个 broker，一个 broker 可以管多个 partition，比如说，topic 有 6 个 partition，有两个 broker，那每个 broker 就管 3 个 partition

Kafka 特点：

生产者消费者模型：FIFO(先进先出)

高性能：单节点支持上千个客户端，百 M/s 吞吐

持久性：消息直接持久化在普通磁盘上切性能好

分布式：数据副本冗余、流量负载均衡、可扩展--一份数据写到不同的 broker

很灵活：消息长时间持久化+client 维护消费状态

1）消息持久化时间跨度比较长，一天或一星期等

2）消费状态自己维护消费到哪个地方了

Kafka 与 Redis 比较：

redis：单机、纯内存性好，持久化较差

kafka：分布式，较长时间持久化，高性能，轻量灵活

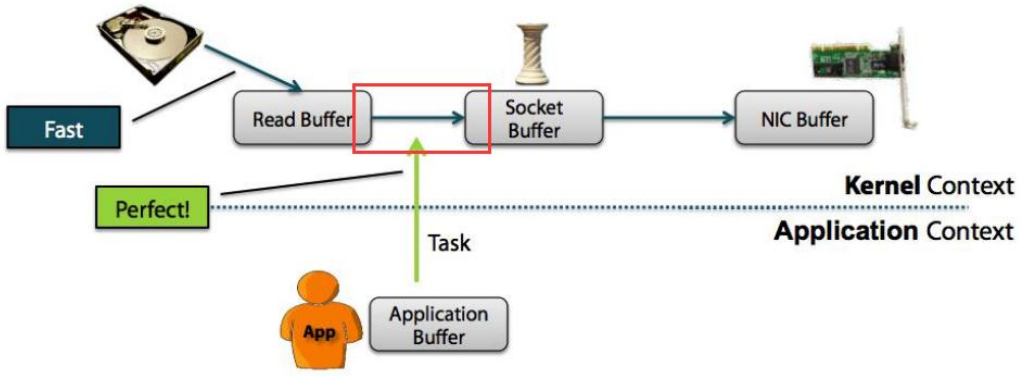
Storm+kafka 有什么好处:

- **满足获取输入。产生输出数据的基本需求**
- **kafka 的分布式、产生输出数据的基本需求**
- **kafka 的分布式、高性能和 storm 吻合**
- **pub-sub 模型可以让多个 storm 业务共享输入数据**
- **kafka 灵活消费的模式能配合 storm 实现不丢不重（exactly-once）的处理模型**
- **exactly-once，精准一次，这种模型在很多时候也是很有用的**

0 拷贝技术：

从 WIKI 的定义中，我们看到“零拷贝”是指计算机操作的过程中，CPU 不需要为数据在内存之间的拷贝消耗资源。而它通常是指计算机在网络上发送文件时，不需要将文件内容拷贝到用户空间（User Space）而直接在内核空间（Kernel

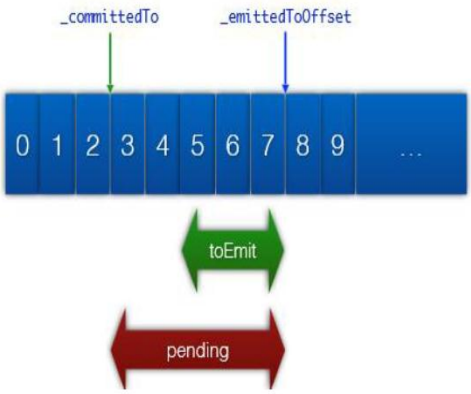
Space）中传输到网络的方式



storm 怎么从 kafka 里面去取数据，维护 offset 的

- 看起来比较简单，最核心的问题就是怎么去维护这个offset，第二个就是怎么把topic的partition均匀分给spout的task

- emittedtoOffest：已经从kafka读到最大offest
- committedTo：已经确认处理完的最大offest
- pending：已经从kafka读到，但是还没有ack的offest队列
- waitingToEmit：已从kafka读到，但是还没有emit出去的数据



Scala：函数式编程--IntelliJ IDEA

一种针对 JVM 将函数和面向对象技术组合在一起的编程语言

面向对象编程是把对象穿来穿去，函数式编程是把函数传来传去即高阶函数

特点：

函数式编程、兼容 JAVA、代码行短、支持并发控制

函数式编程与传统命令的典型区别:

在函数编程中函数是基本单位；变量只是一个名称，而不是一个存储单元

Clojure 能够吸引人的很重要一点是它是 JVM 之上的语言，这个决定非常关键

Clojure 的设计原则可以概括成 5 个词汇：**简单、专注、实用、一致和清晰**。这不是我概括的，而是《The joy of clojure》概括的

（1）简单： 鼓励纯函数，极简的语法（少数 special form），个人也认为 clojure 不能算是多范式的语言（有部分 OO 特性），为了支持多范式引入的复杂度，我们在 C++和 Scala 身上都看到了。

（2）专注：前缀运算符不需要去考虑优先级，也没有什么菱形继承的问题，动态类型系统（有利有弊），REPL 提供的探索式编程方法（告别修改/编译/运行的死循环，所见即所得）。

（3）实用：前面提到，构建在 JVM 之上，跟 Java 语言的互操作非常容易。直接调用 Java 方法，不去发明一套新的调用语法，努力规避 Java 语言中繁琐的地方(doto,箭头宏等等）。

（4）清晰：纯函数（前面提到），immutable var，immutable 数据结构，STM 避免锁问题。不可变减少了心智的负担，降低了多线程编程的难度，纯函数也更利于测试和调试。

（5）一致：语法的一致性：例如 doseq 和 for 宏类似，都支持 destructring，支持相同的 guard 语句（when,while）。数据结构的一致性：sequence 抽象之上的各种高阶函数。

val：常量声明--不可变
var：变量声明--可改变
def 函数声明；def v3 = v1*v2 只定义 v1*v2 表达式的名字，并不求值，在使用时求值

<-：用于 for 循环
=>：用于匿名函数，也可用于 import 中定义别名：import javax.swing.{JFrame=>jf}
->：用于 Map 初始化，也可以不用->而写成 Map((1,”a”),(2,”b”))

数据类型：
Unit：表示无值，和其他语言中的 void 等同
Null：空值或者空引用；是所有引用类型的底类型，即所有 AnyRef 的类型的空值都是 Null
None：Option 的两个子类之一，另一个是 Some，用于安全的函数返回值
Option：比使用空字符串的意图更加清晰，比使用 null 来表示缺少某值的做法更加安全
Nil：长度为 0 的 list
Nothing：所有其他类型的子类型，表示没有值，没有实例
Any：所有类型的超类，任何实例都属于 Any
AnyRef：所有引用类型的超类
AnyVal：所有值类型的超类

匿名函数的定义不需要定义返回值类型

1. scala 语言有什么特点，什么是函数式编程、有什么优点
特点：函数式编程、兼容 JAVA、代码行短、支持并发控制
函数式编程：本质是将函数作为抽象单位，而反应成代码形式
优点：无状态、单中心
2. scala 伴生对象有什么作用

伴生对象：scala 中没有 static 成员存在，允许以某种方式去使用 static 成员，即为伴生

Object 中的东西在 class 中可以直接使用

3. scala 并发编程是怎么弄得，你对 actor 模型怎么理解有何优点

通过 actor 实现的

Actor：将消息分发多个线程，每个处理线程处理完将结果发给主线程。主线程将结果接收

1. 导入 actors 包

```
import scala.actors._, scala.actors.Actor._
```

2. 创建 Actor 对象，receive 接受消息

```
val badActor = actor {
      receive {
      //dosomething
      }
}
```

3. 创建 case 类携带实际消息本身

```
case class Speak(line: String)
case class Gesture(bodyPart: String, action: String)
case class NegotiateNewContract;
case class ThatsAWrap;
```

4. 发送消息

说明：Actor 对象名 ！ case 类(消息)

```
badActor ! NegotiateNewContract
badActor ! Speak("丽仪 Speak")
badActor ! Gesture("丽仪", "Gesture")
badActor ! Speak("阿凤 Speak");
badActor ! "彩霞!";
badActor ! ThatsAWrap;
```

note:

```
!   发送异步消息，没有返回值。
!? 发送同步消息，等待返回值。（会阻塞发送消息语句所在的线程）
!! 发送异步消息，返回值是 Future[Any]。
?   不带参数。查看 mailbox 中的下一条消息。
```

5. 接收消息,跟 Scala case 类匹配处理

```
val badActor = actor {
   var done = false
```

```
        while (!done) {
          //跟 Scala case 类匹配处理
          receive {
            case NegotiateNewContract =>
              println("if won't do it for less than $1 mailior")
            case Speak(line) =>
              println(line)
            case Gesture(bodyPart, action) =>
              println("(" + action + "s " + bodyPart + ")")
            case ThatsAWrap =>
              println("Great cast party,everybody! see ya!")
              done = true
            case _ =>
              println("Hub? I'll be in my trailer.")
          }
        }
    }
```

4. scala case class 有什么重要

主要应用于模式匹配上

5. scala akka 框架有没有接触过，有什么重要

Akka 是一个开发库(scala 编写的库)和运行环境

6. scala 为什么设计 var 和 val

可变不可变

Pagerank 算法：

PageRank 算法计算每一个网页的 PageRank 值，然后根据这个值的大小对网页的重要性进行排序.**它的思想**是模拟一个悠闲的上网者，上网者首先随机选择一个网页打开，然后在这个网页上呆了几分钟后，跳转到该网页所指向的链接，这样无所事事、漫无目的地在网页上跳来跳去，PageRank 就是估计这个悠闲的上网者分布在各个网页上的概率。

终止点问题<某一个网页没有出链>:

图必须是强连通的，即从任意网页可以到达其他任意网页

陷阱问题<某一个网页只有自己到自己的出链，没有其他的出链>

解决以上问题：

算法改进：

正常为 v=mv(n-1)

改进为 v=a*mv+(1-a)e

Akka:

Akka 是一个用 Scala 编写的库，用于简化编写容错的、高可伸缩性的 Java 和 Scala 的 Actor 模型应用。

Akka 是一个开发库和运行环境，可以用于构建高并发、分布式、可容错、事件驱动的

基于 JVM 的应用。使构建高并发的分布式应用更加容易。

特性：

•1）易于构建并行和分布式应用（Simple Concurrency & Distribution）Akka 在设计时采用了异步通讯和分布式架构，并对上层进行抽象，如 Actors、Futures ， STM 等。

2）可靠性（Resilient by Design）

•系统具备自愈能力，在本地/远程都有监护。

•3）高性能（High Performance）

•在单机中每秒可发送 50000000 个消息。内存占用小，1GB 内存中可保存 2500000 个 actors。

•4）弹性，无中心（Elastic —Decentralized）

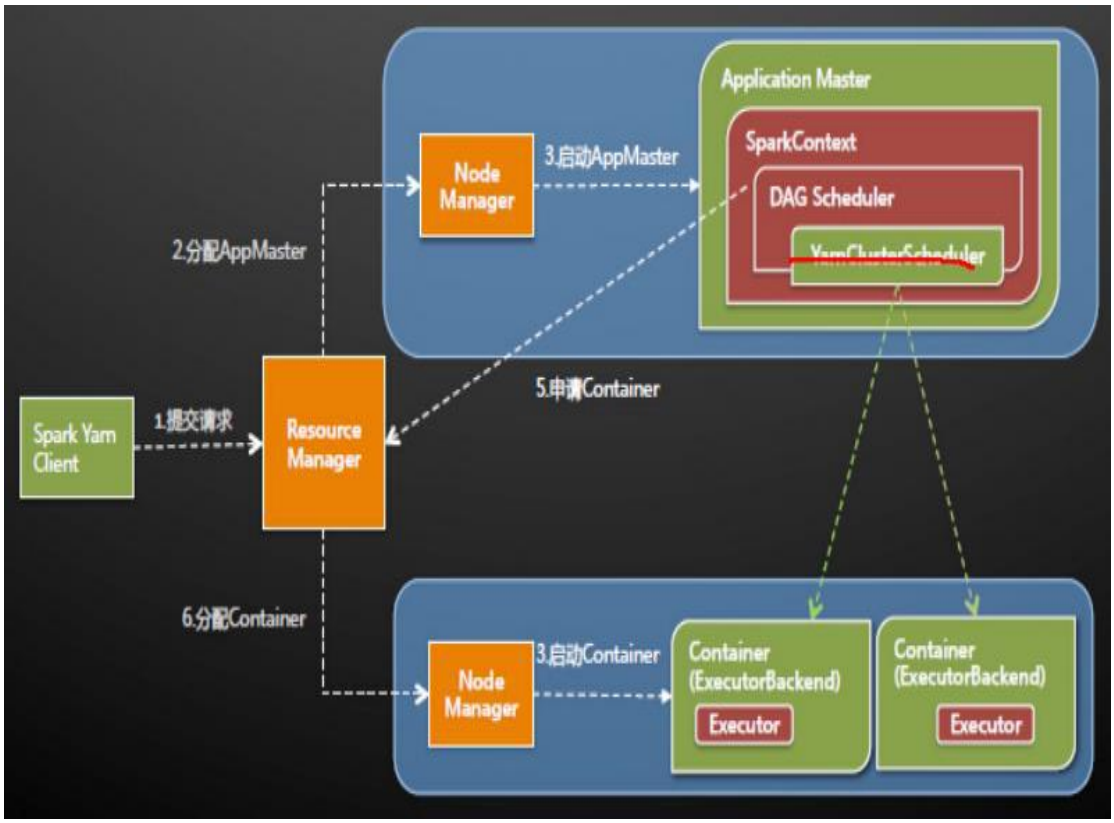
•自适应的负责均衡，路由，分区，配置

5）可扩展（Extensible）

•可以使用 Akka 扩展包进行扩展。

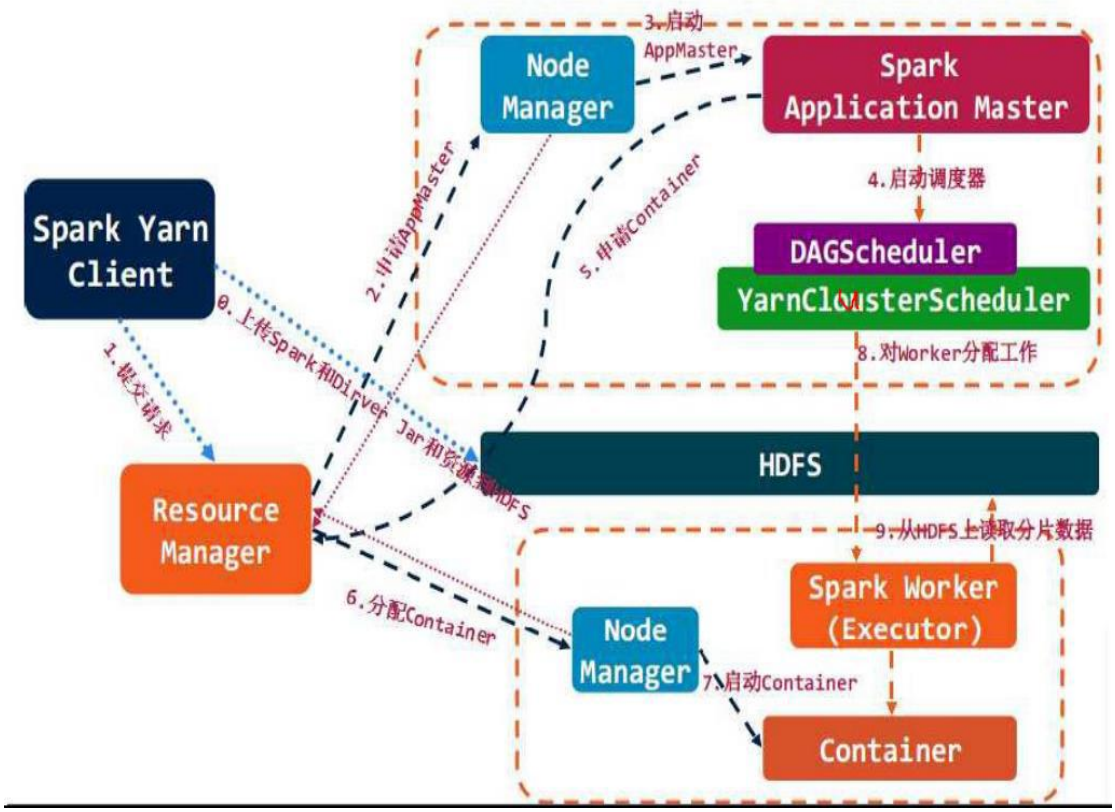
什么场景适合使用：

23、Spark：基于内存的计算框架



..每个 SparkContext 对应一个 ApplicationMaster

..每个 Executor 对应一个 Container



Spark 擅长迭代计算，这可以使 mllib 运行的更快，另外，mllib 也包含高效的算法，利用 spark 的迭代优势，从而达到百倍的效果

RDD:弹性分布式数据集<Resilient Distributed Dataset >

Spark 快的原因：在内存中处理、计算；还有 DAG 优化 Task
Spark 中 spark context 在哪　Driver 就在哪执行
Spark 两种执行方式：

Transformations 延迟执行，执行后没结果显示
Actions 开始执行，有结果显示

缓存策略：

userDisk、useMemory、useOffHeap、deserialized、replication

血统：可以帮助 Spark 中 RDD 快速恢复过来，会记录每个 RDD 之前依赖的 RDD，或者容错可以代码里面指定错 checkpoint：早碰到宽依赖的时候，也就是 shuffle 之前，帮咱们把 RDDs 先缓存起来
转换算子：

Transformations:

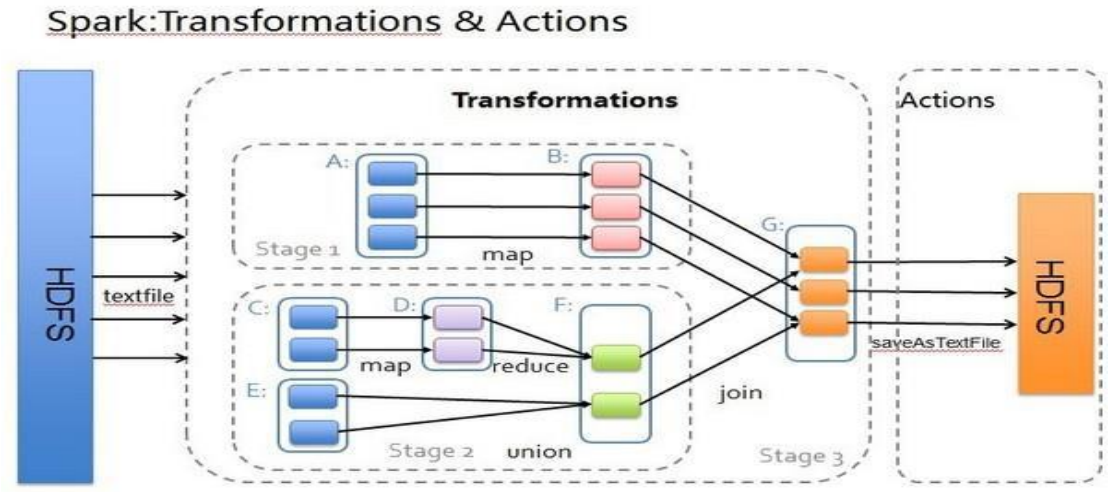
map
mapValues
filter
flatMap
sortByKey
reduceByKey

Actions：

count()
collect()
reduce
save

流程：

Hdfs-->textFile -->处理-->saveAstextFile-->Hdfs



Rdd 容错机制：

Lineage：记录过程，可用于恢复丢失的数据并进行计算

checkpoint：当 lineage 过长时，对 rdd 做 doCheckpoint()

DAG：对 mr 的优化，尽量不挪动数据，记录那个 RDD 或 Stage 输出被持久化了；将 taskset 传下去

窄依赖和宽依赖：

窄依赖：父类的一个 partition 走向子类中一个 partition；数据好恢复

Eg：map，filter

宽依赖：类似于 shuffle，父类的一个 partition 走向子类中多个 partition；数据不好恢复

Eg：join

Straggle 任务：任务不失败，跑的慢，其他节点的任务跑完了，就剩它自己了，机器负载太重了

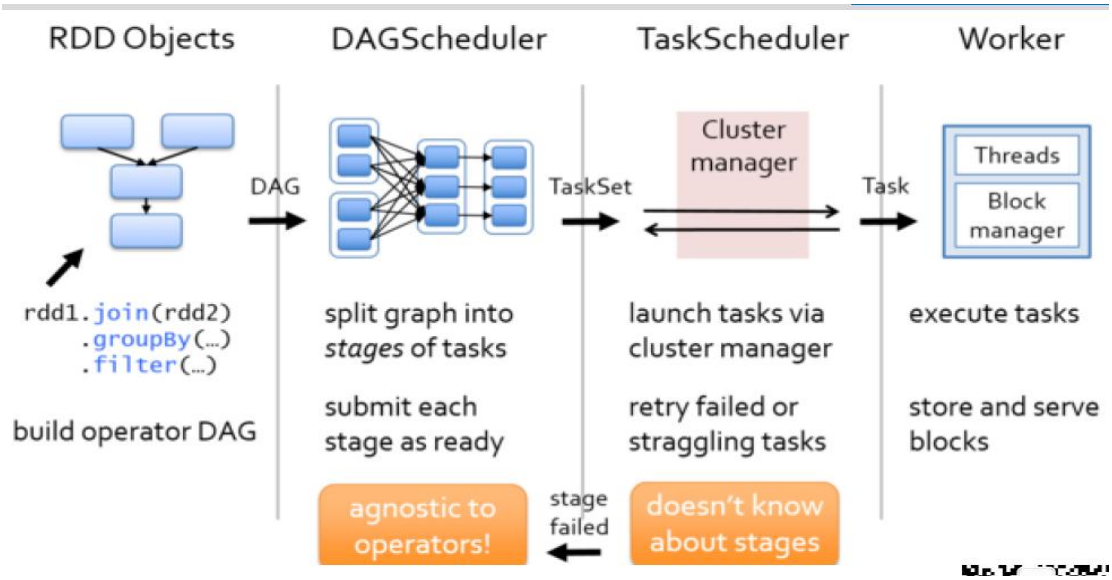
解决：

在 Hadoop 上配置 specular

Spark 会在新的 worker 上再执行一份该任务，两者竞争，谁先完成，最后完成的 kill 掉

调度器根据目标 RDD 的 Lineage 图创建一个由 stage 构成的**无回路有向图（DAG）**。每个 stage 内部尽可能多地包含一组具有窄依赖关系的转换，并将它们流水线并行化（**pipeline**）。stage 的边界有两种情况：一是宽依赖上的 Shuffle 操作；二是已缓存分区，它可以缩短父 RDD 的计算过程。例如图 6。父 RDD 完成计算后，可以在 stage 内启动一组任务计算丢失的分区。

任务调度：



DAG Scheduler

- 基于 Stage 构建 DAG，决定每个任务的最佳位置
- 记录哪个 RDD 或者 Stage 输出被物化
- 将 taskset 传给底层调度器 TaskScheduler
- 重新提交 shuffle 输出丢失的 stage

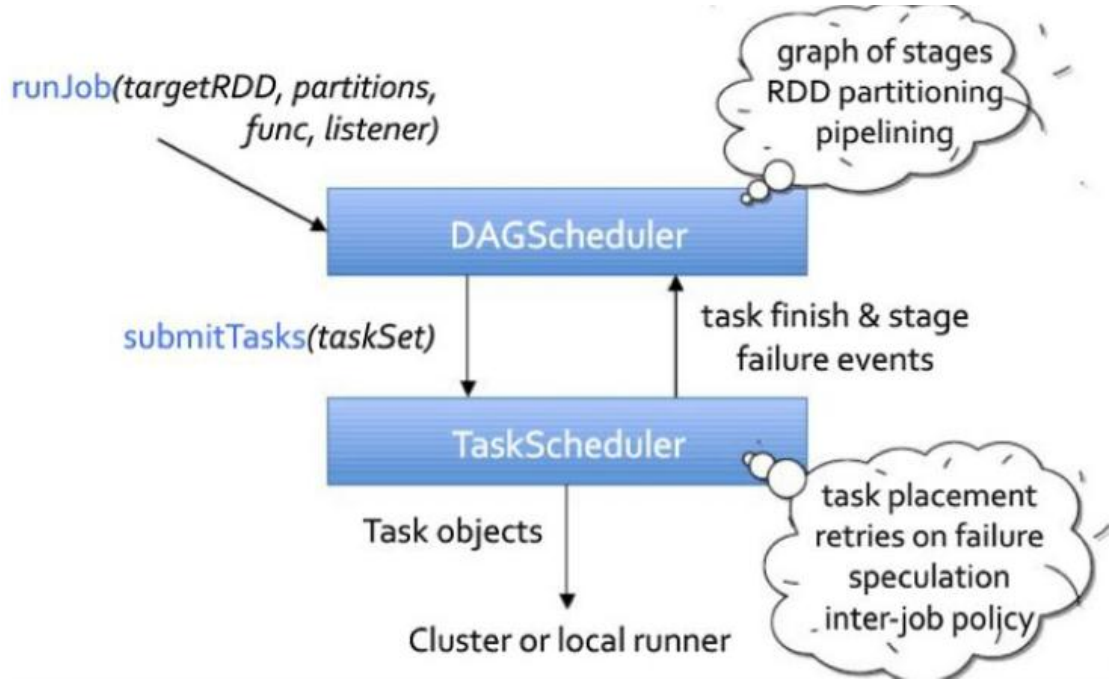
Task Scheduler

- 提交 taskset(一一组 task)到集群运行行并汇报结果
- 出现 shuffle 输出 lost 要报告 fetch failed 错误

•碰到 straggle 任务需要放到别的节点上重试

•为每一一个 TaskSet 维护一一个 TaskSetManager(追踪本地性及错误信息)

Job 调度流程:



job 生成的简单流程如下

1. 首先应用程序创建 SparkContext 的实例，如实例为 sc
2. 利用 SparkContext 的实例来创建生成 RDD
3. 经过一连串的 transformation 操作，原始的 RDD 转换成为其它类型的 RDD
4. 当 action 作用于转换之后 RDD 时，会调用 SparkContext 的 runJob 方法
5. sc.runJob 的调用是后面一连串反应的起点，关键性的跃变就发生在此处

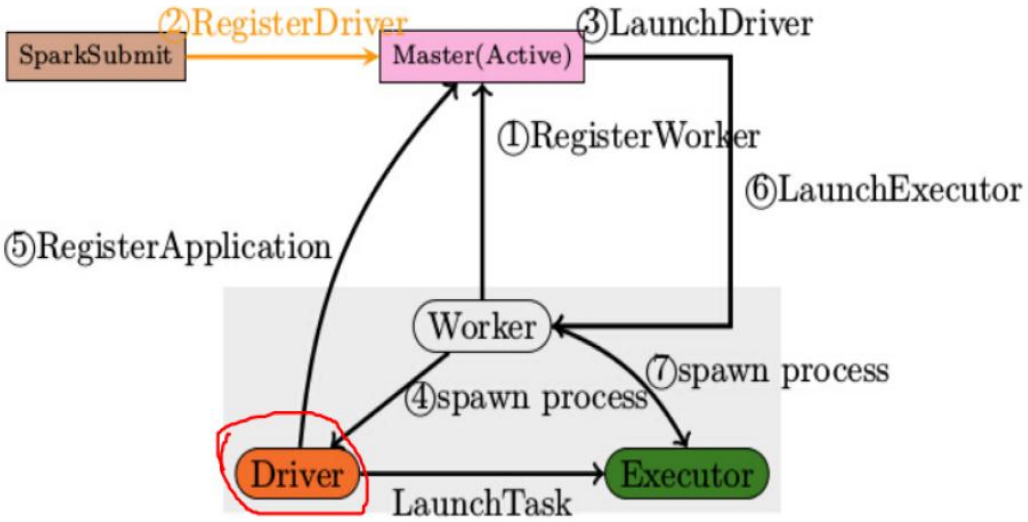
调用路径大致如下

1. sc.runJob->dagScheduler.runJob->submitJob
2. DAGScheduler::submitJob 会创建 JobSummitted 的 event 发送给内嵌类 **eventProcessActor**
3. eventProcessActor 在接收到 JobSubmmitted 之后调用 **processEvent** 处理函数
4. job 到 stage 的转换，生成 finalStage 并提交运行，关键是调用 **submitStage**
5. 在 submitStage 中会计算 stage 之间的依赖关系，依赖关系分为宽依赖和窄依赖两种
6. 如果计算中发现当前的 stage 没有任何依赖或者所有的依赖都已经准备完毕，则提交 task
7. 提交 task 是调用函数 **submitMissingTasks** 来完成
8. task 真正运行在哪个 worker 上面是由 TaskScheduler 来管理，也就是上面的 submitMissingTasks 会调用 TaskScheduler::submitTasks
9. TaskSchedulerImpl 中会根据 Spark 的当前运行模式来创建相应的 backend,如果是在单机运行则创建 LocalBackend

10. LocalBackend 收到 TaskSchedulerImpl 传递进来的 **ReceiveOffers** 事件

11. receiveOffers->executor.launchTask->TaskRunner.run

Cluster：



性能优化：

1.产生过多的空任务或小任务：使用 reparation 减少 RDD 中 partition 的数量

2.每条记录开销太大：能共用的共用，eg：db 连接

3.任务执行行速度倾斜：

1）数据倾斜：partition key 取得不好；在并行行中间加一步 aggregation

2）Worker 倾斜：某些 worker 上的 executor 不给力，设置 spark.speculation=true 把那些持续不给力力的 node 去掉

4.不设置 spark.local.dir--spark 写 shuffle 输出的地方：

设置一组磁盘

增加 IO 即加快速度

5.reducer 数量不合适：实际情况实际调整

太多--产生过多的小任务，产生很多的启动任务开销

太少--任务执行速度慢

Reduce 的任务数还会影响到内存

6.Collect 输出大量结果慢：直接输出到分布式文件系统

7.序列化<默认：ObjectOutputStream>：兼容性好，体积大，速度慢

使用 avro 或 kryo：体积小，速度快

Stage 如何划分出来的：

Task、宽窄依赖

Stage 有两种：

ShuffleMapStage

这种 Stage 是以 Shuffle 为输出边界

其输入边界可以是从外部获取数据，也可以是另一个 ShuffleMapStage 的输出

其输出可以是另一个 Stage 的开始

ShuffleMapStage 的最后 Task 就是 ShuffleMapTask
在一个 Job 里可能有该类型的 Stage，也可以能没有该类型 Stage。

ResultStage
这种 Stage 是直接输出结果
其输入边界可以是从外部获取数据，也可以是另一个
ShuffleMapStage 的输出
ResultStage 的最后 Task 就是 ResultTask
在一个 Job 里必定有该类型 Stage。
一个 Job 含有一个或多个 Stage，但至少含有一个 ResultStage。

spark 为什么快，你能说出你的几点看法吗？

Spark 对小数据集能达到亚秒级的延迟，这对于 Hadoop MapReduce（以下简称 MapReduce）是无法想象的（由于“心跳”间隔机制，仅任务启动就有数秒的延迟）。就大数据集而言，对典型的迭代机器 学习、即席查询（ad-hoc query）、图计算等应用，Spark 版本比基于 MapReduce、Hive 和 Pregel 的实现快上十倍到百倍。其中内存计算、数据本地性 （locality）和传输优化、调度优化等该居首功，也与设计伊始即秉持的轻量理念不无关系。

spark 里面的 transformation action 是干什么的，有什么区别，能举例说明几个常用方法吗？

简介：

1，transformation 是得到一个新的 RDD，方式很多，比如从数据源生成一个新的 RDD，从 RDD 生成一个新的 RDD

2，action 是得到一个值，或者一个结果（直接将 RDDcache 到内存中）所有的 transformation 都是采用的懒策略，就是如果只是将 transformation 提交是不会执行计算的，计算只有在 action 被提交的时候才被触发。

transformation 操作：

map(func):对调用 map 的 RDD 数据集中的每个 element 都使用 func，然后返回一个新的 RDD,这个返回的数据集是分布式的数据集

filter(func): 对调用 filter 的 RDD 数据集中的每个元素都使用 func，然后返回一个包含使 func 为 true 的元素构成的 RDD

flatMap(func):和 map 差不多，但是 flatMap 生成的是多个结果

mapPartitions(func):和 map 很像，但是 map 是每个 element，而 mapPartitions 是每个 partition

mapPartitionsWithSplit(func):和 mapPartitions 很像，但是 func 作用的是其中一个 split 上，所以 func 中应该有 index

sample(withReplacement,faction,seed):抽样

union(otherDataset)：返回一个新的 dataset，包含源 dataset 和给定 dataset 的元素的集合

distinct([numTasks]):返回一个新的 dataset，这个 dataset 含有的是源 dataset 中的 distinct 的 element

groupByKey(numTasks):返回(K,Seq[V])，也就是 hadoop 中 reduce 函数接受的 key-valuelist

reduceByKey(func,[numTasks]):就是用一个给定的 reducefunc 再作用在 groupByKey 产生的(K,Seq[V]),比如求和，求平均数

sortByKey([ascending],[numTasks]):按照 key 来进行排序，是升序还是降序，ascending 是 boolean 类型

join(otherDataset,[numTasks]):当有两个 KV 的 dataset(K,V)和(K,W)，返回的是(K,(V,W))的 dataset,numTasks 为并发的任务数

cogroup(otherDataset,[numTasks]):当有两个 KV 的 dataset(K,V)和(K,W)，返回的是(K,Seq[V],Seq[W])的 dataset,numTasks 为并发的任务数

cartesian(otherDataset)：笛卡尔积就是 m*n，大家懂的

action 操作：

reduce(func)：说白了就是聚集，但是传入的函数是两个参数输入返回一个值，这个函数必须是满足交换律和结合律的

collect()：一般在 filter 或者足够小的结果的时候，再用 collect 封装返回一个数组

count():返回的是 dataset 中的 element 的个数

first():返回的是 dataset 中的第一个元素

take(n):返回前 n 个 elements，这个士 driverprogram 返回的

takeSample(withReplacement，num，seed)：抽样返回一个 dataset 中的 num 个元素，随机种子 seed

saveAsTextFile（path）：把 dataset 写到一个 textfile 中，或者 hdfs，或者 hdfs 支持的文件系统中，spark 把每条记录都转换为一行记录，然后写到 file 中

saveAsSequenceFile(path):只能用在 key-value 对上，然后生成 SequenceFile 写到本地或者 hadoop 文件系统

countByKey()：返回的是 key 对应的个数的一个 map，作用于一个 RDD

foreach(func):对 dataset 中的每个元素都使用 func

sdd 你怎么理解的

RDD，全称为 Resilient Distributed Datasets，是一个容错的、并行的数据结构，可以让用户显式地将数据存储到磁盘和内存中，并能控制数据的分区。同时，RDD 还提供了一组丰富的操作来操作这些数据。在这些操作中，诸如 map、flatMap、filter 等转换操作实现了 monad 模式，很好地契合了 Scala 的集合操作。除此之外，RDD 还提供了诸如 join、groupBy、reduceByKey 等更为方便的操作（注意，reduceByKey 是 action，而非 transformation），以支持常见的数据运算。

通常来讲，针对数据处理有几种常见模型，包括：Iterative Algorithms，Relational Queries，MapReduce，Stream Processing。例如 Hadoop MapReduce 采用了 MapReduces 模型，Storm 则采用了 Stream Processing 模型。RDD 混合了这四种模型，使得 Spark 可以应用于各种大数据处理场景。

RDD 作为数据结构，本质上是一个只读的分区记录集合。一个 RDD 可以包含多个分区，每个分区就是一个 dataset 片段。RDD 可以相互依赖。如果 RDD 的每个分区最多只能被一个 Child RDD 的一个分区使用，则称之为 narrow dependency；若多个 Child RDD 分区都可以依赖，则称之为 wide dependency。不同的操作依据其特性，可能会产生不同的依赖。例如 map 操作会产生 narrow dependency，而 join 操作则产生 wide dependency。

spark 作业提交流程是怎么样的，client 和 cluster 有什么区别，各有什么作用？

Driver 运行在 Worker 上:

作业执行流程描述：

1.客户端提交作业给 Master

2.Master 让一个 Worker 启动 Driver，即 SchedulerBackend。Worker 创建一个 DriverRunner 线程，DriverRunner 启动 SchedulerBackend 进程。

3.另外 Master 还会让其余 Worker 启动 Exeuctor，即 ExecutorBackend。Worker 创建一个 ExecutorRunner 线程，ExecutorRunner 会启动 ExecutorBackend 进程。

4.ExecutorBackend 启动后会向 Driver 的 SchedulerBackend 注册。SchedulerBackend 进程中包含 DAGScheduler，它会根据用户程序，生成执行计划，并调度执行。对于每个 stage 的 task，都会被存放到 TaskScheduler 中，ExecutorBackend 向 SchedulerBackend 汇报的时候把 TaskScheduler 中的 task 调度到 ExecutorBackend 执行。

5.所有 stage 都完成后作业结束。

Driver 运行在客户端:

作业执行流程描述：

1.客户端启动后直接运行用户程序，启动 Driver 相关的工作：DAGScheduler 和 BlockManagerMaster 等。

2.客户端的 Driver 向 Master 注册。

3.Master 还会让 Worker 启动 Exeuctor。Worker 创建一个 ExecutorRunner 线程，ExecutorRunner 会启动 ExecutorBackend 进程。

4.ExecutorBackend 启动后会向 Driver 的 SchedulerBackend 注册。Driver 的 DAGScheduler 解析作业并生成相应的 Stage，每个 Stage 包含的 Task 通过 TaskScheduler 分配给 Executor 执行。

5.所有 stage 都完成后作业结束。

spark on yarn 作业执行流程，yarn-client 和 yarn cluster 有什么区别

yarn-client 相当于是命令行，会将你输入的代码提交到 yarn 上面执行；yarn-cluster 是将你写好的程序打成 jar 包然后提交到 yarn 上面去执行，然后 yarn 会将 jar 包分发到各个节点，并负责资源分配和任务管理

spark streamning 工作流程是怎么样的?

Spark Streaming 是将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是 Spark，也就是把 Spark Streaming 的输入数据按照 batch size（如 1 秒）分成一段一段的数据（Discretized Stream），每一段数据都转换成 Spark 中的 RDD（Resilient Distributed Dataset），然后将 Spark Streaming 中对 DStream 的 Transformation 操作变为针对 Spark 中对 RDD 的 Transformation 操作，将 RDD 经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加，或者存储到外部设备

spark sql 你使用过没有，在哪个项目里面使用的

这个题目可以结合自己做过的项目来说，主要是了解 spark sql 是做什么的，如果对这块熟，又在项目中用过，可以说使用过，可以从下面的点回答：

SparkSQL 抛弃原有 Shark 的代码，汲取了 Shark 的一些优点，如内存列存储（In-Memory Columnar Storage）、Hive 兼容性等，重新开发了 SparkSQL 代码；由于摆脱了对 hive 的依赖性，SparkSQL 无论在数据兼容、性能优化、组件扩展方面都得到了极大的方便；

shark 的出现，使得 SQL-on-Hadoop 的性能比 hive 有了 10-100 倍的提高。

spark 机器学习和 spark 图计算接触过没，，能举例说明你用它做过什么吗？

这个问题，其实我也不会，但是可以之前就准备下这方面的资料，从 spark 的特点回答面试官，也可以跟 hadoop 做对比，比如，spark 是内存计算框架，运行速度要比 hadoop 框架快很多倍。

spark 图计算可以从下面几点来回答：

图存储模式，边分割（Edge-Cut），点分割（Vertex-Cut）

图计算模式，BSP（Bulk Synchronous Parallell）计算模式，Pregel 模型，GAS 模型，

GraphX 的图运算符，

一半找一两个熟悉的点说就够了。

spark sdd dag ,stage 你怎么理解的 sdd？不是 rdd

RDD，全称为 Resilient Distributed Datasets，是一个容错的、并行的数据结构，可以让用户显式地将数据存储到磁盘和内存中，并能控制数据的分区。

在图论中，如果一个有向图无法从任意顶点出发经过若干条边回到该点，则这个图是一个有向无环图（DAG 图）

DAG 可用于对数学和 计算机科学中得一些不同种类的结构进行建模。

DAG 也可以用来模拟信息沿着一个一 致性的方向通过处理器网络的过程。

DAG 中得可达性关系构成了一个局 部顺序，任何有限的局部顺序可以由 DAG 使用可达性来呈现

DAG 的可作为一个序列集合的高效利用空间的重叠的子序列的代表性。

面向 stage 的调度层，为 job 生成以 stage 组成的 DAG，提交 TaskSet 给 TaskScheduler 执行。

每一个 Stage 内，都是独立的 tasks，他们共同执行同一个 compute function，享有相同的 shuffledependencies。DAG 在切分 stage 的时候是依照出现 shuffle 为界限的。

Stage 类会传入 Option[ShuffleDependency[_,_]]参数，内部有一个 isShuffleMap 变量，以标识该 Stage 是 shuffle or result。

spark 宽依赖 窄依赖你怎么理解的

窄依赖指父 RDD 的每一个分区最多被一个子 RDD 的分区所用，表现为一个父 RDD 的分区对应于一个子 RDD 的分区，和两个父 RDD 的分区对应于一个子 RDD 的分区。

宽依赖指子 RDD 的分区依赖于父 RDD 的所有分区，这是因为 shuffle 类操作，如图 3 中的 groupByKey 和未经协同划分的 join

窄依赖对优化很有利。

stage 是基于什么原理分割为 task 的。。

要理解什么是 Stage，首先要搞明白什么是 Task。Task 是在集群上运行的基本单位。一个 Task 负责处理 RDD 的一个 partition。RDD 的多个 patition 会分别由不同的 Task 去处理。当然了这些 Task 的处理逻辑完全是一致的。这一组 Task 就组成了一个 Stage。有两种 Task：

org.apache.spark.scheduler.ShuffleMapTask

org.apache.spark.scheduler.ResultTask

ShuffleMapTask 根据 Task 的 partitioner 将计算结果放到不同的 bucket 中。而 ResultTask 将计算结果发送回 Driver Application。一个 Job 包含了多个 Stage，而 Stage 是由一组完全相同的 Task 组成的。最后的 Stage 包含了一组 ResultTask。

在用户触发了一个 action 后，比如 count，collect，SparkContext 会通过 runJob 的函数开始进行任务提交。最后会通过 DAG 的 event processor 传递到 DAGScheduler 本身的 handleJobSubmitted，它首先会划分 Stage，提交 Stage，提交 Task。至此，Task 就开始在运行在集群上了。

一个 Stage 的开始就是从外部存储或者 shuffle 结果中读取数据；一个 Stage 的结束就是由于发生 shuffle 或者生成结果时

一个 stage 的边界，输入是外部的存储或者一个 stage shuffle 的结果；输入则是 Job 的结果（result task 对应的 stage）或者 shuffle 的结果。

DAGScheduler 将 Stage 划分完成后，提交实际上是通过把 Stage 转换为 TaskSet，然后通过 TaskScheduler 将计算任务最终提交到集群。

接下来，将分析 Stage 是如何转换为 TaskSet，并最终提交到 Executor 去运行的。

Spark 机器学习：MLlib

MLlib 基本数据类型：

Vector：稠密向量--dense<很少有 0>和稀疏向量--sparse

LabeledPoint：LabeledPoint 其实就是在你向量之前加一个表示，正负分别用 1.0 和 0.0 来表示

libSVM：

本地矩阵：

分布式矩阵：

统计摘要：--summary

单例的几种写法：

懒汉式单例

```
//懒汉式单例类.在第一次调用的时候实例化自己
public class Singleton {
    private Singleton() {}
    private static Singleton single=null;
    //静态工厂方法
    public static Singleton getInstance() {
         if (single == null) {
             single = new Singleton();
         }
        return single;
    }
}
```

静态内部类

```
public class Singleton {
    private static class LazyHolder {
       private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton (){}
    public static final Singleton getInstance() {
       return LazyHolder.INSTANCE;
    }
}
```

饿汉式单例

```
//饿汉式单例类.在类初始化时，已经自行实例化
public class Singleton1 {
    private Singleton1() {}
    private static final Singleton1 single = new Singleton1();
    //静态工厂方法
    public static Singleton1 getInstance() {
        return single;
    }
}
```

hbase 读的效率高还是写的效率高，为什么；

http://www.programmer.com.cn/7246/

hadoop1.x 和 hadoop2.x 的区别等。

1.HDFS 的变化- 增强了 NameNode 的水平扩展及可用性；1 中只有一个 NN，2 中有多个 NN

2.配置文件的路径变化；1 中 hadoop 配置文件在$HADOOP_HOME/conf 下，2 中在$HADOOP_HOME/etc/hadoop

3.命令文件目录变化；1 中都在 bin 下，不区分客户端和服务端命令，2 中区分，将服务端命名单独存在 sbin 中

项目开发过程中遇到的困难，怎么解决的

聚类算法你怎么理解的，他有什么作用，你使用这个算法做过什么？

Storm：

难点为把无线网络优化的各个细节分析清楚，

spout 获取数据，第一个 bolt 怎么设计的，第二个 bolt 怎么设计的，spout、bolt、bolt 各启动了几个线程，是基于什么来设定启动线程的数量的；为什么采用 kafka。

Map 的输出的环形缓冲区怎么设计的，为什么这么设计

Hbase 的表怎么设计的，为什么不使用精确到小时设计，基站数量大概有多少

1. **hive 如何优化；**
2. **基于 Hive 写 Sql 的复杂查询怎么样？**
3. **MapReduce 如何优化；**
4. **Hadoop 如何优化；（主要想听配置文件如何优化）**
5. **HBase 的优点 + HBase 的优化。**
6. PageRank 算法
7. .基于 ZooKeeper 的 Redis 高可用如何开发的，原理；
8. 机器学习推荐算法中，电商企业选择基于物品的还是基于用户的；
9. 索引有几种，简述一下。（这个我不太会，回答了：倒排索引，二叉树，B 树；
10. 基于物品的特征性强还是基于用户的特征性强
11. Spark 的工作原理
12. Redis3.0 自带的高可用
13. 双重读写保证安全机制时，也就是既往 redis 写，又往 mysql 写时，怎么做到 redis 的键值 DB 和关系表结构对应的
    先往 redis 写，再往 mysql 写或者自己设计一个监听器
14. spark 的 RDD 是什么？运行原理？为何快？
15. 使用虚拟化技术时效率会降低，你能告诉我大概降低百分之多少吗？
16. 两个数据中心如何同步数据
17. Storm 里为什么用 kafka，即消息队列的作用；
18. 你以前的项目中 hive 的运用，听过全增量模型，还有什么增量模型我都没记住
19. Sqoop 导入数据用过了，导出数据用过吗？
20. 谈谈敏捷开发
21. **企业好像很重视 spark，不懂 spark 也会问你有没有意向；**
22. **4 家企业问道了项目经历，因此项目经历一定要天衣无缝；**
23. **大企业基本考算法，而且都是笔试基础的算法，我都忘的差不多了；**
24. **不会的技术：JVM 优化，R 语言，敏捷开发，两个数据中心如何同步数据，Redis3.0 高可用，索引有几种**
25. 单向链表和双向链表
    a) 单向：尾部存下一个的位置
    b) 双向：尾部存下一个位置，头部存前一个位置

26. **hadoop 小文件过多会怎么样**

任何一个文件，目录和 block，在 HDFS 中都会被表示为一个 object 存储在 namenode 的内存中，没一个 object 占用 150 bytes 的内存空间。所以，如果有 10million 个文件，没一个文件对应一个 block，那么就将要消耗 namenode 3G 的内存来保存这些 block 的信息。如果规模再大一些，那么将会超出现阶段计算机硬件所能满足的极限。

27. **storm 事务：**

在事务性拓扑中，Storm 以并行和顺序处理混合的方式处理元组。spout 并行分批创建供 bolt 处理的元组（译者注：下文将这种分批创建、分批处理的元组称做批次）。其中一些 bolt 作为提交者以严格有序的方式提交处理过的批次。这意味着如果你有每批五个元组的两个批次，将有两个元组被 bolt 并行处理，但是直到提交者成功提交了第一个元组之后，才会提交第二个元组。 NOTE： 使用事务性拓扑时，数据源要能够重发批次，有时候甚至要

重复多次。因此确认你的数据源——你连接到的那个 spout——具备这个能力。 这个过程可以被描述为两个阶段： 处理阶段 纯并行阶段，许多批次同时处理。 提交阶段 严格有序阶段，直到批次一成功提交之后，才会提交批次二。 这两个阶段合起来称为一个 Storm 事务

云计算服务分类

SaaS：软件即服务

是一种通过 Internet 提供软件的模式，用户不用再购买软件，而改用向提供商租用基于 Web 的软件，来管理企业经营活动，且无需对软件进行维护

PaaS：平台即服务

把计算环境、开发环境等平台作为一种服务提供的商业模式

提供基础架构服务 - 存储，主机，计算，带宽等等

IaaS：基础设施即服务

把数据中心、基础设施硬件资源通过 Web 分配给用户使用的商业模式

云计算的优点：

超大规模、虚拟化<不用管实际在哪运行>、高可靠性<多副本容错>、通用性<支撑不同的应用类型>、高可伸缩性<根据需求变化>、按需服务<资源池>、极其廉价<可使用廉价节点构件>

Openstack：

VMWare 三种网络模式：

桥接模式<直接连外网>、NAT 模式<ip 映射连外网>、HOST-ONLY<只能连主机>

模块：

Horizon：管理 WEB，控制面板，dashboard 服务

Nova：管理虚拟机

Neutron：网络模块,管理网络--quantum

Cinder：提供永久的块存储给运行中的实例，管理磁盘

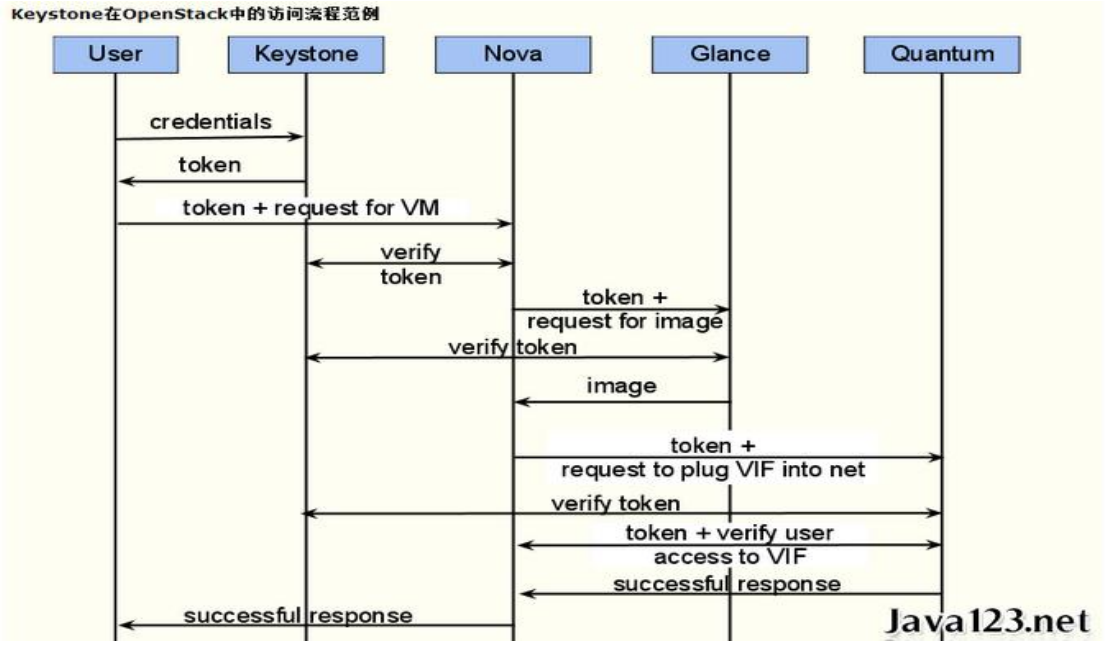
Keystone<核心>：提供认证和授权服务，管理权限，身份验证服务

Glance：存储虚拟机磁盘镜像，管理镜像

Swift：对象存储

Tenant：租户，资源池

EndPoint：端点，是一个服务暴露出来的访问点



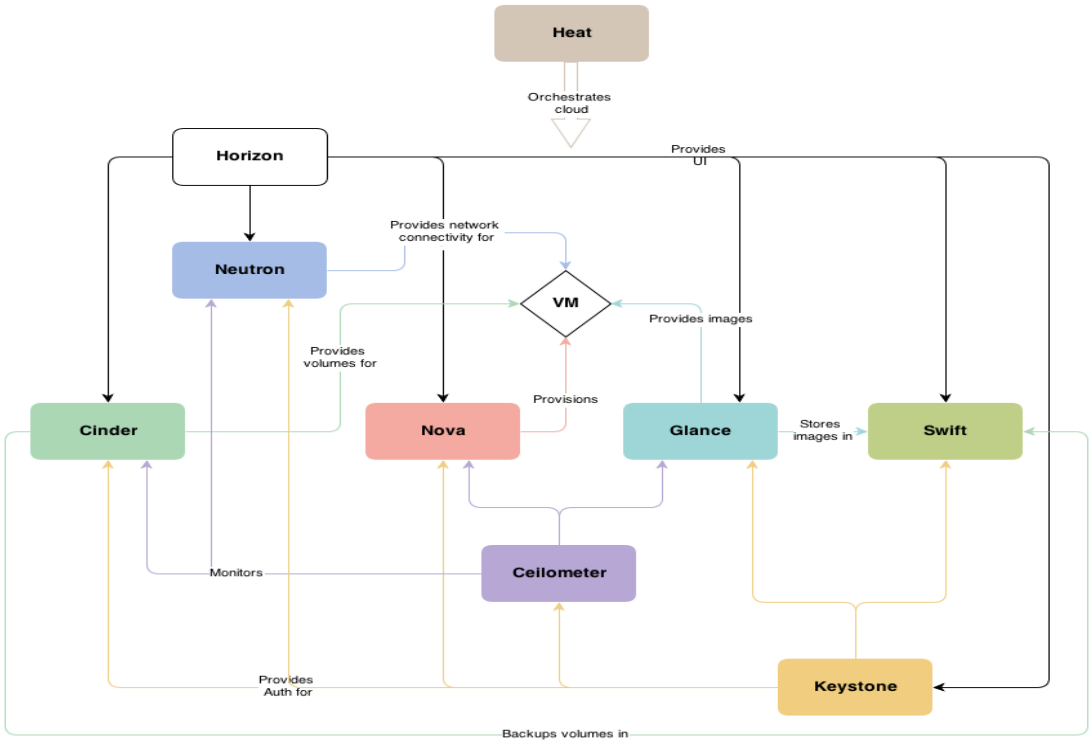
Openstack 是开源的云计算管理平台项目

四大开源云平台：

OpenStack、CoudStack、OpenNebula、Eucalyptus

核心模块：

Horzion、KeyStone



1.Horizon(Dashboard)与其他主要模块的关联（包括 Nova,Cinder,Glance,Swift,Neutron, keystone)---黑色线

2.通过 Ceilometer(监控功能）可以监控的模块(包括 Nova,Glance,Cinder,Neutron)--蓝色线

3.Keystone（身份验证功能）模块可以对其他模块进行相应操作进行身份及权限验证(包括 Nova,Glance,Cinder,Swift,Neutron,Ceilometer)--橙色线

1）Nova 为 VM 提供计算资源

2）Glance 为 VM 提供镜像

3）Cinder 为 VM 提供块存储资源

4）Neutron 为 VM 提供网络资源及网络连接

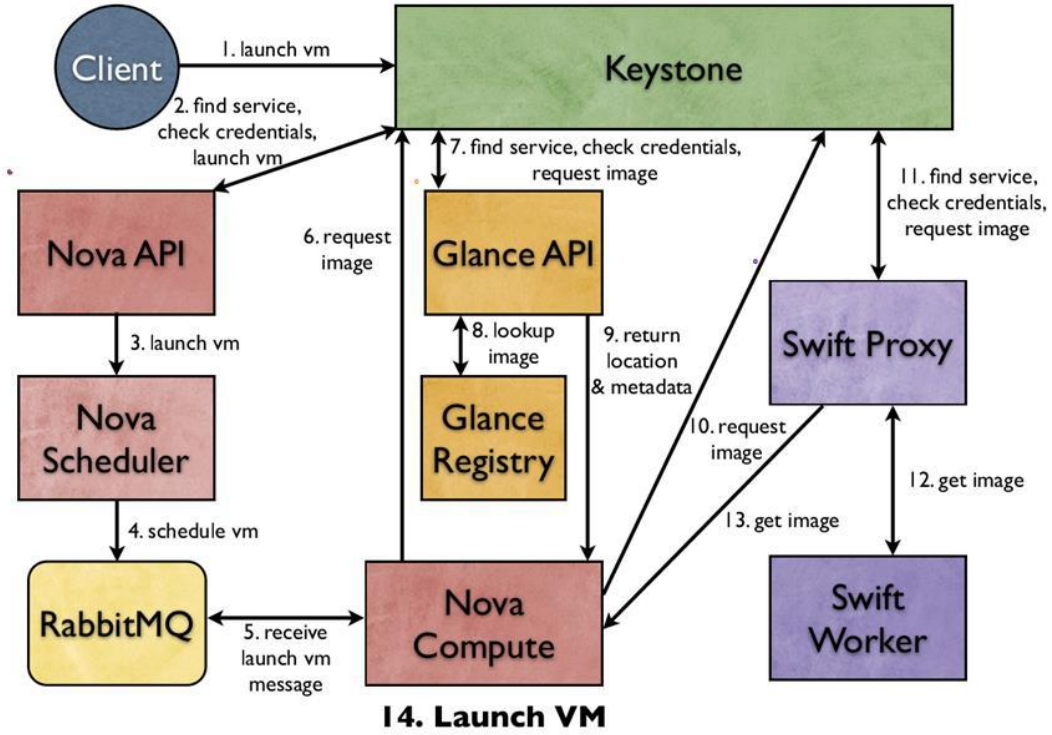
Heat 是一套业务流程平台，旨在帮助用户更轻松地配置以 OpenStack 为基础的云体系

测量(Metering)：Ceilometer。像一个漏斗一样，能把 OpenStack 内部发生的几乎所有的事件都收集起来，然后为计费和监控以及其它服务提供数据支撑

OpenStack 主要逻辑模块－**Horizon Dashboard** 服务：

在整个 Openstack 应用体系框架中，Horizon 就是整个应用的入口。它提供了一个模块化的，基于 web 的图形化界面服务门户。用户可以通过浏览器使用这个 Web 图形化界面来访问、控制他们的计算、存储和网络资源，如启动实例、分配 IP 地址、设置访问控制

请求一个虚拟机实例过程



Openstack 网络模式

- Flat
- Flatdhcp
- gre
- Vlan：为每个项目提供受保护的网段(虚拟 LAN)。(4096) 2^12
- vxlan：2^24