# 环境搭建1

## 软件清单

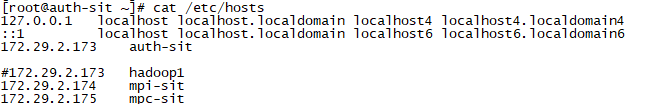
环境依赖：jdk8

软件：

https://www.cloudera.com/documentation/enterprise/release-notes/topics/cdh\_vd\_cdh\_package\_tarball\_59.html

1. zookeeper：zookeeper-3.4.6.tar.gz
2. hadoop：hadoop-2.6.0-cdh5.9.3.tar.gz (插件：hadoop-native-64-2.6.0.tar)
3. mysql：mysql-5.6.34.tar.gz
4. hbase：hbase-1.2.0-cdh5.9.3.tar.gz，插件phoenix（未安装）
5. hive：hive-1.1.0-cdh5.9.3.tar.gz
6. sqoop：sqoop-1.4.6-cdh5.9.3.tar.gz

## 准备工作

1. 配置三台机器的hosts, 如下 vim /etc/hosts  
   
2. 创建hadoop 用户并赋予root组权限
   * 1. useradd hadoop
     2. passwd hadoop
     3. gpasswd -a hadoop root
     4. 在创建文件夹 mkdir –p /apps/svr/bigdata
     5. 赋予权限

chown -R hadoop /apps/svr/bigdata

chgrp -R hadoop /apps/svr/bigdata

* + 1. su hadoop

1. 将hadoop用户1台master与 2台 slave 的ssh 互信
2. auth-sit机器上执行:
   1. ssh-keygen -t rsa
   2. ssh-copy-id auth-sit
   3. ssh-copy-id mpi-sit
   4. ssh-copy-id mpc-sit
3. mpi-sit机器上执行：
   1. ssh-keygen -t rsa
   2. ssh-copy-id mpc-sit

## 安装zookeeper

zookeeper配置如下: (/apps/svr/bigdata/zookeeper-3.4.6)

1. 配置zk环境变量

export ZOOKEEPER\_HOME=/apps/svr/bigdata/zookeeper-3.4.6

export PATH=$PATH:$ZOOKEEPER\_HOME/bin:$ZOOKEEPER\_HOME/conf

1. 配置修改$ ZOOKEEPER\_HOME/conf下的zoo.cfg ，如下文件



1. 在文件$ ZOOKEEPER\_HOME 新建文件夹
2. mkdir data
3. mkdir log
4. 将zookeeper文件同步到2台slave机器中
5. scp -r /apps/svr/bigdata/zookeeper-3.4.6 mpi-sit:/apps/svr/bigdata/
6. scp -r /apps/svr/bigdata/zookeeper-3.4.6 mpc-sit:/apps/svr/bigdata/
7. 并且在$ ZOOKEEPER\_HOME /data下建myid文件

vim myid auth-sit设置为1 mpi-sit设置为2、mpc-sit设置为3

1. 用zkServer.sh start 启动zk
2. 用zkServer.sh status 查看启动状态，如图  
   

## 安装hadoop

hadoop配置如下：(/apps/svr/bigdata/hadoop-2.6.0-cdh5.9.3)

1. 配置/etc/profile

export HADOOP\_HOME=/apps/svr/bigdata/hadoop-2.6.0-cdh5.9.3

export PATH=$PATH:$HADOOP\_HOME/bin:$HADOOP\_HOME/sbin

export HADOOP\_CONF\_DIR=$HADOOP\_HOME

export HADOOP\_COMMON\_LIB\_NATIVE\_DIR=$HADOOP\_HOME/lib/native

export HADOOP\_OPTS=-Djava.library.path=$HADOOP\_HOME/lib

1. 配置修改$HADOOP\_HOME/etc/hadoop下配置文件

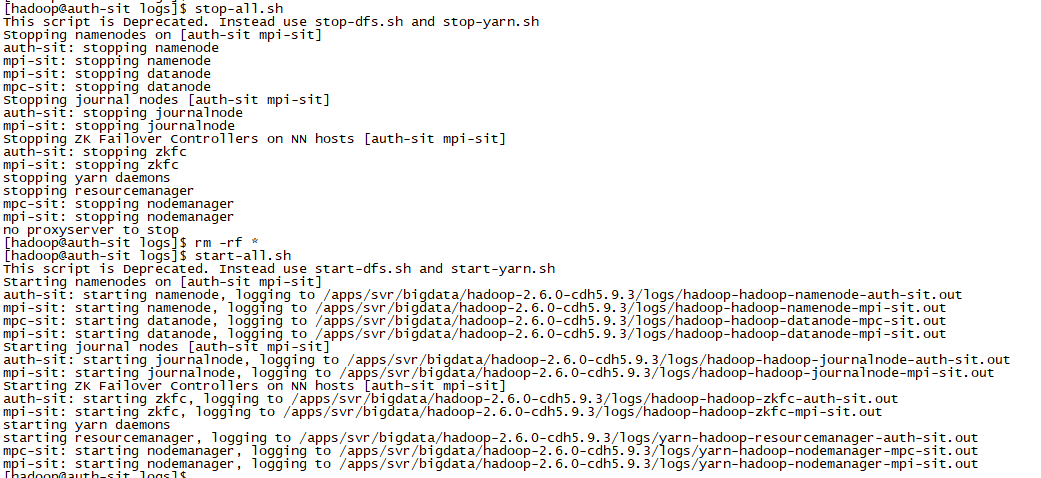


1. 安装hadoop-native-64-2.6.0.tar
   * 1. tar -xvf hadoop-native-64-2.6.0.tar –C /apps/svr/bigdata/hadoop-2.6.0-cdh5.9.3/lib/native
     2. tar -xvf hadoop-native-64-2.6.0.tar -C /apps/svr/bigdata/hadoop-2.6.0-cdh5.9.3/lib
2. mkdir -p /apps/svr/bigdata/hadoop/hadoop\_tmp

mkdir -p /apps/svr/bigdata/hadoop/journalnode

mkdir -p /apps/svr/bigdata/hadoop/hadoop\_tmp/dfs/name

1. 执行hdfs namenode -format
2. 将hadoop文件同步到2台机器中
   * 1. scp -r /apps/svr/bigdata/hadoop-2.6.0-cdh5.9.3 mpi-sit:/apps/svr/bigdata/
     2. scp -r /apps/svr/bigdata/hadoop-2.6.0-cdh5.9.3 mpc-sit:/apps/svr/bigdata/
     3. scp -r /apps/svr/bigdata/hadoop mpi-sit:/apps/svr/bigdata/
     4. scp -r /apps/svr/bigdata/hadoop mpc-sit:/apps/svr/bigdata/
3. ~~在auth-sit执行start-all.sh~~



1. 上一步使用start-all.sh启动时，会出现一些依赖未启动和同步问题，建议使用如下分部启动
   * 1. 确保已启动zk
     2. 启动journalNode，在auth-sit mpi-sit节点上执行hadoop-daemon.sh start journalnode
     3. 格式化hdfs，在auth-sit上执行命令: hdfs namenode -format
     4. 格式化高可用，在auth-sit上执行: hdfs zkfc -formatZK，查看zk根目录下，多了个ha的目录
     5. 同步NameNode的元数据信息

auth-sit执行： hdfs namenode

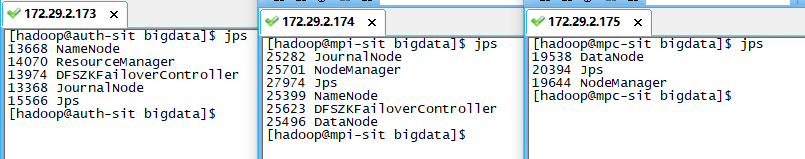
mpi-sit节点执行： hdfs namenode -bootstrapStandby，同步完成后，ctrl+c结束掉auth-sit的进程

* + 1. 启动hdfs，在auth-sit上执行，start-dfs.sh
    2. 启动yarn

在auth-sit上执行，start-yarn.sh

在mpi-sit上执行：yarn-daemon.sh start resourcemanager （单独启动rm进程）

1. jps查看启动信息



1. 安装完成

hadoop web服务器地址:

<http://auth-sit:50070/dfshealth.html#tab-overview>

http://auth-sit:8088/cluster

<http://mpi-sit:8042/node>

http://mpc-sit:8042/node

1. 常见问题
   * 1. Invalid URI for NameNode address (check fs.defaultFS): file:/// has no authority.

hadoop fs -chmod 577 ///

* + 1. java.io.IOException: NameNode is not formatted.

start-all.sh-> hdfs namenode -format-> stop-all.sh-> start-all.sh

或者

start-all.sh-> hdfs namenode -format-> hadoop-daemon.sh start namenode

## 安装mysql

172.29.1.208 root/root

## 安装hbase

hbase配置如下：(/apps/svr/bigdata/hbase-1.2.0-cdh5.9.3)

1. 配置/etc/profile

export HBASE\_HOME=/apps/svr/bigdata/hbase-1.2.0-cdh5.9.3

export PATH=$PATH:$HBASE\_HOME/bin

1. 配置修改$ HBASE\_HOME/etc/hadoop下的配置文件



1. 将hbase文件同步到2台slave机器中

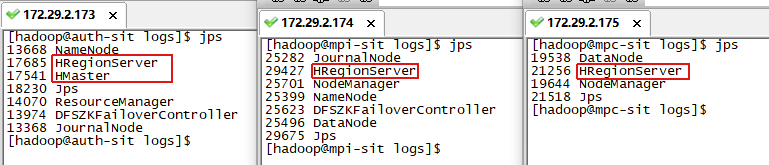
scp -r /apps/svr/bigdata/hbase-1.2.0-cdh5.9.3 mpi-sit:/apps/svr/bigdata/

scp -r /apps/svr/bigdata/hbase-1.2.0-cdh5.9.3 mpc-sit:/apps/svr/bigdata/

1. 启动hbase

auth-sit：start-hbase.sh

1. jps查看启动信息



1. 安装完成

http://auth-sit:60010/master-status

## 安装hive

hive从0.14开始，使用Zookeeper实现了HiveServer2的HA功能(ZooKeeper Service Discovery)

我们这里简单部署一下， 只在auth-sit部署一台

hive配置如下：(/apps/svr/bigdata/ hive-1.1.0-cdh5.9.3)

1. 配置/etc/profile

export HIVE\_HOME=/apps/svr/bigdata/hive-1.1.0-cdh5.9.3

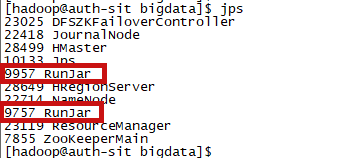
export PATH=$PATH:$HIVE\_HOME/bin

export PATH=$PATH:$HIVE\_HOME/lib配置修改$ HBASE\_HOME/etc/hadoop下的配置文件

1. 配置修改$ HIVE\_HOME/conf下的配置文件

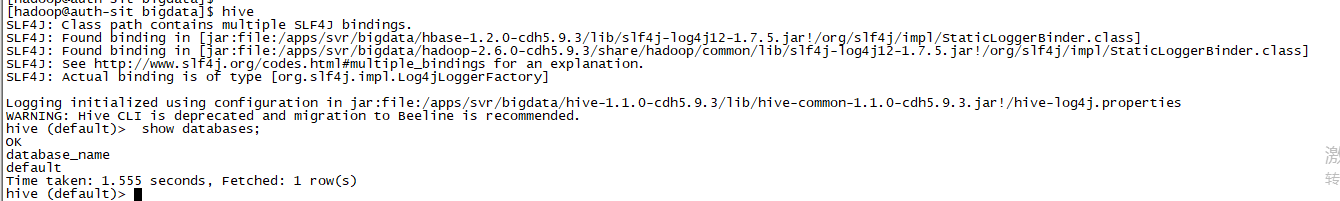


1. 启动hive
   * 1. schematool -initSchema -dbType mysql
     2. hive --service metastore &
     3. hiveserver2 &
2. jps查看启动信息



2个Runjar进程分别对应上一步的hive --service metastore &和hiveserver2 &

1. hive客户端



1. 启动hwi控制台

hive --service hwi

http://auth-sit:9999/hwi/

## 插件sqoop

Sqoop(发音：skup)是一款开源的工具，主要用于在Hadoop(Hive)与传统的数据库(mysql、postgresql...)间进行数据的传递，可以将一个关系型数据库（例如 ： MySQL ,Oracle ,Postgres等）中的数据导进到Hadoop的HDFS中，也可以将HDFS的数据导进到关系型数据库中。

sqoop2与sqoop1有很大的区别，使用的是server-client模式，在一个节点安装好后，在其他任何节点都可以访问。

sqoop配置如下：(/apps/svr/bigdata/sqoop2-1.99.5-cdh5.9.3，auth-sit)

1. 配置/etc/profile

export SQOOP\_HOME=/apps/svr/bigdata/sqoop2-1.99.5-cdh5.9.3

export PATH=$PATH:$SQOOP\_HOME/bin

export SQOOP\_SERVER\_EXTRA\_LIB=$SQOOP\_HOME/extra

export CATALINA\_BASE=$SQOOP\_HOME/server

export LOGDIR=$SQOOP\_HOME/logs/

export PATH

1. 修改/apps/svr/bigdata/sqoop2-1.99.5-cdh5.9.3/server/conf

## CDH快速搭建

软件：

# 环境搭建2

## 安装spark

# 技术总结

## zookeeper

### 基础

zkCli.sh

ls /

ls2 /

cd /hbase

ls /hbase

* 特点

最终一致性：为客户端展示同一视图，这是zookeeper最重要的功能。

可靠性：如果消息被到一台服务器接受，那么它将被所有的服务器接受。

实时性：zookeeper不能保证两个客户端能同时得到刚更新的数据，如果需要最新数据，应该在读数据之前调用sync()接口。

等待无关（wait-free）：慢的或者失效的client不干预快速的client的请求。

原子性：更新只能成功或者失败，没有中间状态。

顺序性：所有Server，同一消息发布顺序一致

* 节点类型

1. 持久节点（PERSISTENT）

所谓持久节点，是指在节点创建后，就一直存在，直到有删除操作来主动清除这个节点——不会因为创建该节点的客户端会话失效而消失。

1. 持久顺序节点（PERSISTENT\_SEQUENTIAL）

这类节点的基本特性和上面的节点类型是一致的。额外的特性是，在ZK中，每个父节点会为他的第一级子节点维护一份时序，会记录每个子节点创建的先后顺序。基于这个特性，在创建子节点的时候，可以设置这个属性，那么在创建节点过程中，ZK会自动为给定节点名加上一个数字后缀，作为新的节点名。这个数字后缀的范围是整型的最大值。   
在创建节点的时候只需要传入节点 “/test\_”，这样之后，zookeeper自动会给”test\_”后面补充数字。

1. 临时节点（EPHEMERAL）

和持久节点不同的是，临时节点的生命周期和客户端会话绑定。也就是说，如果客户端会话失效，那么这个节点就会自动被清除掉。注意，这里提到的是会话失效，而非连接断开。另外，在临时节点下面不能创建子节点。   
这里还要注意一件事，就是当你客户端会话失效后，所产生的节点也不是一下子就消失了，也要过一段时间，大概是10秒以内，可以试一下，本机操作生成节点，在服务器端用命令来查看当前的节点数目，你会发现客户端已经stop，但是产生的节点还在。

1. 临时顺序节点（EPHEMERAL\_SEQUENTIAL）

此节点是属于临时节点，不过带有顺序，客户端会话结束节点就消失。

### 案例

* 使用zookeeper有序节点实现分布式锁：

假设锁空间的根节点为/lock

1.客户端连接zookeeper，并在/lock下创建临时的且有序的子节点，第一个客户端对应的子节点为/lock/lock-0000000000，第二个为/lock/lock-0000000001，以此类推；

2.客户端获取/lock下的子节点列表，判断自己创建的子节点是否为当前子节点列表中序号最小的子节点，如果是则认为获得锁，否则监听刚好在自己之前一位的子节点删除消息，获得子节点变更通知后重复此步骤直至获得锁；

3.执行业务代码；

4.完成业务流程后，删除对应的子节点释放锁。

### 使用场景

* 数据发布与订阅（配置中心）
* 负载均衡
* 命名服务(Naming Service)
* 分布式通知/协调
* 集群管理与Master选举
* 分布式锁
* 分布式队列

## hadoop

### 基础

### 节点

* Namenode

Namenode管理者文件系统的Namespace。它维护着文件系统树(filesystemtree)以及文件树中所有的文件和文件夹的元数据(metadata)。管理这些信息的文件有两个，分别是Namespace镜像文件(Namespace image)和操作日志文件(edit log)，这些信息被Cache在RAM中，当然，这两个文件也会被持久化存储在本地硬盘。Namenode记录着每个文件中各个块所在的数据节点的位置信息，但是他并不持久化存储这些信息，因为这些信息会在系统启动时从数据节点重建。

* DataNode

Datanode是文件系统的工作节点，他们根据客户端或者是namenode的调度存储和检索数据，并且定期向namenode发送他们所存储的块(block)的列表（心跳机制）。

集群中的每个服务器都运行一个DataNode后台程序，这个后台程序负责把HDFS数据块读写到本地的文件系统。当需要通过客户端读/写某个数据时，先由NameNode告诉客户端去哪个DataNode进行具体的读/写操作，然后，客户端直接与这个DataNode服务器上的后台程序进行通信，并且对相关的数据块进行读/写操作。

* SecondaryNameNode

SecondaryNameNode是一个用来监控HDFS状态的辅助后台程序。就像NameNode一样，每个集群都有一个Secondary NameNode，并且部署在一个单独的服务器上。SecondaryNameNode不同于NameNode，它不接受或者记录任何实时的数据变化，但是，它会与NameNode进行通信，以便定期地保存HDFS元数据的快照。由于NameNode是单点的，通过Secondary NameNode的快照功能，可以将NameNode的宕机时间和数据损失降低到最小。同时，如果NameNode发生问题，Secondary NameNode可以及时地作为备用NameNode使用。这其中SecondaryNameNode作为NameNode备用节点，切换的过程完全不影响其它节点，举个栗子：NameNode就相当于皇帝，而SecondaryNameNode相当于东宫太子，皇帝即NameNode出现故障或者死去，东宫太子会立即继位，下面的大臣（DataNade）完全感觉不到皇位已经换人了。

* ResourceManager和NodeManager

一个为资源管理，另一个为节点管理，顾名思义，一个主要负责资源的管理，另一个负责节点的管理

ResourceManager：

1. 与客户端交互，处理来自客户端的请求

2. 启动和管理ApplicationMaster，并在它运行失败时重新启动它；资源管理和调度，接收来自ApplicationMaster的资源申请请求，并为之分配资源

3. 管理NodeManager，接收来自NodeManager的资源汇报信息，并向NodeManager下达管理指令（比如杀死Container等）

NodeManager：

Node Manager就是Weblogic Server提供的一个Java 程式，可以使让你在本地域控制台启动、关闭、重启及监管受管Server。Node Manager允许你从远程machine控制受管Server。要使用NodeManager的功能，必须在受管Server机器上启动Node Manager。

### hdfs

HDFS(Hadoop分布式文件系统)的分布式文件系统，基于 Hadoop 的应用程序使用 HDFS 。

HDFS 是专为存储超大数据文件，运行在集群的商品硬件上。

它是容错的，可伸缩的，并且非常易于扩展。

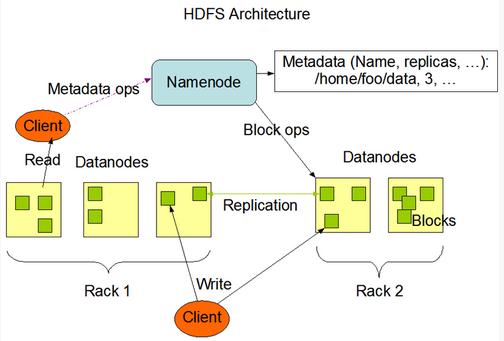
HDFS集群主要由 NameNode 管理文件系统 Metadata 和 DataNodes 存储的实际数据。

HDFS中的读/写操作运行在块级。HDFS数据文件被分成块大小的块，这是作为独立的单元存储。默认块大小为64 MB。

HDFS操作上是数据复制的概念，其中在数据块的多个副本被创建，分布在整个节点的群集以使在节点故障的情况下数据的高可用性。

注:在HDFS的文件，比单个块小，不占用块的全部存储。

* 基本构架图



* 数据存储方式

1. 按行存储 textfile

按列存储 orc/parqurt

--orcfile

每列数据有类似于元数据的索引信息,可以确定列内容,需要某列信息时可以直接锁定列内容,效率优于按行存储

压缩出来的文件比例最小,以时间换存储

--parquet

比较复杂,支持嵌套数据结构和高效其种类丰富的算法(以应对不同值分布特征的压缩)

压缩率不如orcfile,时间与压缩比适中

1. 压缩

压缩率:TEXTFILE(不压缩) RCFILE(14%) parquet(62%) orcfile(78%) 其中ORCFILE是RCFILE一个升级

减少网络IO

减少了磁盘IO存储

注意压缩必须有可分割性(在map输出经过shuffle到reduce时需要解压缩,保证单个数据还可以被解压)

hadoop支持的压缩格式:zlib/gzip/bzip2/lzo/lz4/snappy

<name>io.compression.codecs</name>

* 一致模型

文件系统的一致模型描述了读/写的数据可见性。HDFS 为了性能牺牲了一些 Posix 要求，也就是默认情况下，写入文件的内容并不保证能立即可见，即使流已经刷新（调用 flush() 方法）并存储。

当写入的数据超过一个块后，第一个数据块对新的 reader 可见。当前正在写入的块对其他 reader 不可见

HDFS 提供了一种强行将所有缓存刷新到 datanode 中的手段，即调用 hflush() 方法。当 hflush() 方法返回成功后，对所有新的 reader 而言，HDFS 能保证文件中到目前为止写入的数据均达到所有 datanode 的写入管道并且对所有新的 reader 可见

hflush() 不保证数据写入磁盘，为了确保数据写入磁盘，可以使用 hsync()

选择 hflush(), hsync() 合适的调用频率

* 数据完整性

Hadoop数据的完整性检测，都是通过校验和的比较来完成，在创建新文件时（也就是在上传数据到hdfs上时）将校验和的值和数据一起保存起来。NameNode会收到来自client、DataNode的检验和信息，根据这两个信息来维护文件的块存储及向客户端提供块读取服务。

HDFS会对写入的所有数据计算校验和，并在读取数据时验证校验和。   
常用的错误检测码是CRC-32(循环冗余校验)，任何大小的数据输入均计算得到一个32位的整数校验和。   
在写入文件时，hdfs为每个数据块都生成一个crc文件。客户端读取数据时生成一个crc与数据节点存储的crc做比对，如果不匹配则说明数据已经损坏了。数据节点在后台运行一个程序定期（默认为21天）检测数据，防止物理存储介质中位衰减而造成的数据损坏。

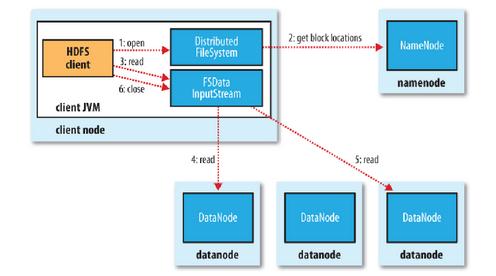
1.DataNode在写入时计算出校验和，然后每次读的时候再计算校验和进行检验hdfs会为每一个固定长度的数据（一个个数据包）执行一次校验和这个值由io.bytes.per.checksum指定，默认是512字节。因为CRC32是32位即4个字节，这样校验和占用的空间就会少于原数据的1%。

2.datanode在存储收到的数据前会校验数据的校验和，比如收到客户端的数据或者其他副本传过来的数据。如hdfs数据流中客户端写入数据到hdfs时的数据流，如果发现错误，就会抛出ChecksumException到客户端从datanode读数据的时候一样要检查校验和，而且每个datanode还保存了检查校验和的日志，每次校验都会记录到日志中

3.除了读写操作会检查校验和以外，datanode还跑着一个后台进程（DataBlockScanner）定期校验存在在它上面的block，因为除了读写过程中会产生数据错误以外，硬件本身也会产生数据错误，比如位衰减（bit rot）

### hdfs读写

* 读操作



根据HDFS地址，客户端首先获取FileSystem的一个实例，也就是HDFS对应的实例。

1.客户端调用FileSystem实例的open方法，获取这个这个文件对应对应的输入流，在HDFS上就是DFSInputStream。

2.构造第一步的输入流DFSInputStream时，通过RPC远程调用NameNode可以获得NameNode中此文件对应数据块保存位置在输入流中会按照网络拓扑结构，根据与客户端距离对DataNode进行简单的排序。

3-4.获得输入流后，客户端调用read方法读取数据。输入流会根据前面的排序，选择最近的DataNode建立连接并读取数据。如果客户端和其中一个DataNode位于同一机器(比如MapReduce过程中的mapper和reducer)，那么就直接从本地读取数据。

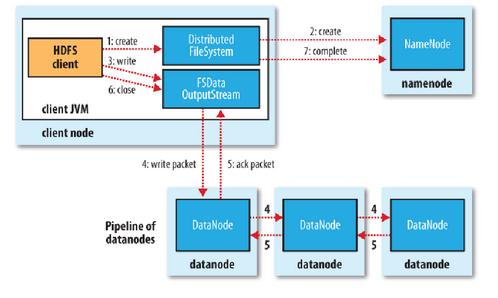
5.如果已经达到了数据块末端，那么关闭与这个DataNode的连接，然后重新查找下一个数据块。

不断执行2-5步直到读完文件的所有数据块，然后调用close。

6.客户端调用close，关闭输入流DFSInputStream。

注：如果DFSInputStream和DataNode通信时遇到错误，或者数据校验出错，则DFSInputStream会重新连接次近的DataNode读取数据。

* 写操作



根据HDFS地址，客户端首先获取FileSystem的一个实例，也就是HDFS对应的实例。

1-2.客户端调用FileSystem实例的create方法，创建文件。NameNode通过一些检查，比如文件是否存在，客户端是否拥有创建权限等；通过检查之后，在NameNode中添加文件信息。注意，因为这时的文件还没有数据，故NameNode上也没有文件数据块的信息。创建 结束后，HDFS会返回输出流DFSOutputStream给客户端。

3.客户端调用输出流DFSOutputStream的write方法向对应的HDFS中对应的文件写数据。数据会首先拆包，这学分包会写入输出流的内部Data队列，接收完整数据分包后，输出流DFSOutputStream会向NameNode申请保存文件和副本数据块的若干个DataNode信息，这若干个DataNode会形成一个数据传输管道。

4.DFSOutputStream会(根据网络拓扑结构排序)将数据传输给离自己最近的DataNode，这个DataNode接收到数据包后传递给下一个DataNode。数据在各个DataNode之间通过管道流动，而不是全部由输出流分发，减少了传输开销。

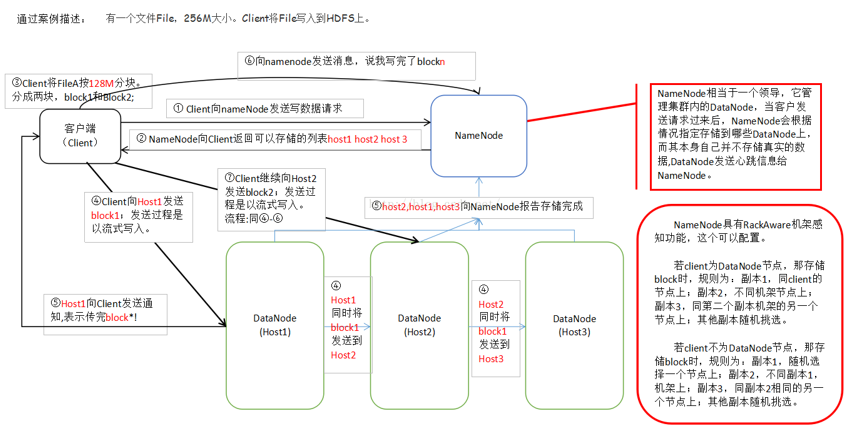
5.各个DataNode位于不同机器上，数据需要通过网络传输，为了保证数据的完整性，接收到数据的DataNode需要向发送者发送确认包(ACK Packet)。对于某个数据块，  只有当DFSOutputStream接收到了所有的DataNode的正确ACK，才能确认传输结束。DFSOutputStream内部专门维护了一个等待ACK队列，这一队列保存已经进入管道传输数据、但是并未被完全确认的数据包。

不断执行3-5步直到数据全部写完，客户端调用close关闭文件。

6.客户端调用close方法，DFSOutputStream继续等待直到所有数据写入完毕并被确认，调用complete方法通知NameNode文件写入完成。

7.NameNode接收到complete消息后，等待相应数量的副本写入完毕，告知客户端。

* 示例



### hdfs副本摆放策略

第一副本：放置在上传文件的DataNode上；如果是集群外提交，则随机挑选一台磁盘不太慢、CPU不太忙的节点上；  
第二副本：放置在于第一个副本不同的机架的节点上；  
第三副本：与第二个副本相同机架的不同节点上；  
如果还有更多的副本：随机放在节点中；

注意：

HDFS中存储的文件的副本数由上传文件时设置的副本数决定。无论以后怎么更改系统副本系数，这个文件的副本数都不会改变；

在上传文件时优先使用启动命令中指定的副本数，如果启动命令中没有指定则使用hdfs-site.xml中dfs.replication设置的默认值；

### hdfs负载均衡

* Hadoop的HDFS集群非常容易出现机器与机器之间磁盘利用率不平衡的情况，例如：当集群内新增、删除节点，或者某个节点机器内硬盘存储达到饱和值。当数据不

平衡时Map任务可能会分配到没有存储数据的机器，这将导致网络带宽的消耗，也无法很好的进行本地计算。

当HDFS负载不均衡时，需要对HDFS进行数据的负载均衡调整，即对各节点机器上数据的存储分布进行调整。从而，让数据均匀的分布在各个DataNode上，均衡IO性能，防止热点的发生。进行数据的负载均衡调整，必须要满足如下原则：

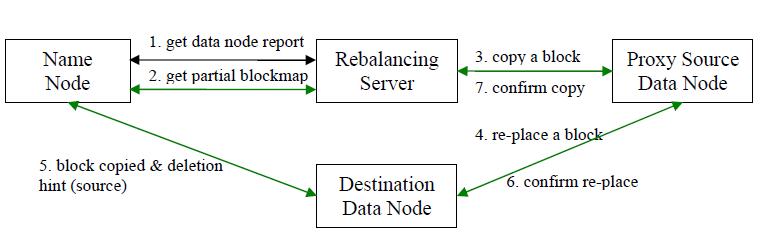
1.数据平衡不能导致数据块减少，数据块备份丢失

2.管理员可以中止数据平衡进程

3.每次移动的数据量以及占用的网络资源，必须是可控的

4.数据均衡过程，不能影响namenode的正常工作

* 数据均衡过程的核心是一个数据均衡算法，该数据均衡算法将不断迭代数据均衡逻辑，直至集群内数据均衡为止。该数据均衡算法每次迭代的逻辑如下：



步骤分析如下：

1.数据均衡服务（Rebalancing Server）首先要求 NameNode 生成 DataNode 数据分布分析报告,获取每个DataNode磁盘使用情况

2.Rebalancing Server汇总需要移动的数据分布情况，计算具体数据块迁移路线图。数据块迁移路线图，确保网络内最短路径

3.开始数据块迁移任务，Proxy Source Data Node复制一块需要移动数据块

4.将复制的数据块复制到目标DataNode上

5.删除原始数据块

6.目标DataNode向Proxy Source Data Node确认该数据块迁移完成

7.Proxy Source Data Node向Rebalancing Server确认本次数据块迁移完成。然后继续执行这个过程，直至集群达到数据均衡标准

* DataNode分组

在第2步中，HDFS会把当前的DataNode节点,根据阈值的设定情况划分到Over、Above、Below、Under四个组中。在移动数据块的时候，Over组、Above组中的块向Below组、Under组移动。

Over组：此组中的DataNode的均满足

DataNode\_usedSpace\_percent > Cluster\_usedSpace\_percent + threshold

Above组：此组中的DataNode的均满足

Cluster\_usedSpace\_percent + threshold > DataNode\_ usedSpace \_percent > Cluster\_usedSpace\_percent

Below组：此组中的DataNode的均满足

Cluster\_usedSpace\_percent > DataNode\_ usedSpace\_percent > Cluster\_ usedSpace\_percent – threshold

Under组：此组中的DataNode的均满足

Cluster\_usedSpace\_percent – threshold > DataNode\_usedSpace\_percent

* HDFS 数据自动平衡脚本使用方法

在hadoop中，包含一个start-balancer.sh脚本，通过运行这个工具，启动HDFS数据均衡服务。该工具可以做到热插拔，即无须重启计算机和 hadoop 服务。

启动命令为：bin/start-balancer.sh –threshold

1. 影响balancer的几个参数：

1.-threshold

默认设置：10，参数取值范围：0-100

参数含义：判断集群是否平衡的阈值。理论上，该参数设置的越小，整个集群就越平衡

2.dfs.balance.bandwidthPerSec

默认设置：1048576（1M/S）

参数含义：Balancer运行时允许占用的带宽

1. 示例如下：

#启动数据均衡，默认阈值为 10%

bin/start-balancer.sh

#启动数据均衡，阈值 5%

bin/start-balancer.sh –threshold 5

#停止数据均衡

bin/stop-balancer.sh

1. 在hdfs-site.xml文件中可以设置数据均衡占用的网络带宽限制

<property>

<name>dfs.balance.bandwidthPerSec</name>

<value>1048576</value>

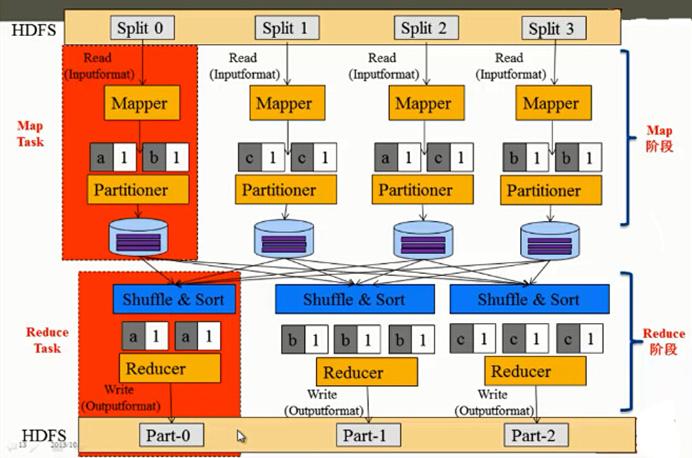
<description> Specifies the maximum bandwidth that each datanode can utilize for the balancing purpose in term of the number of bytes per second. </description>

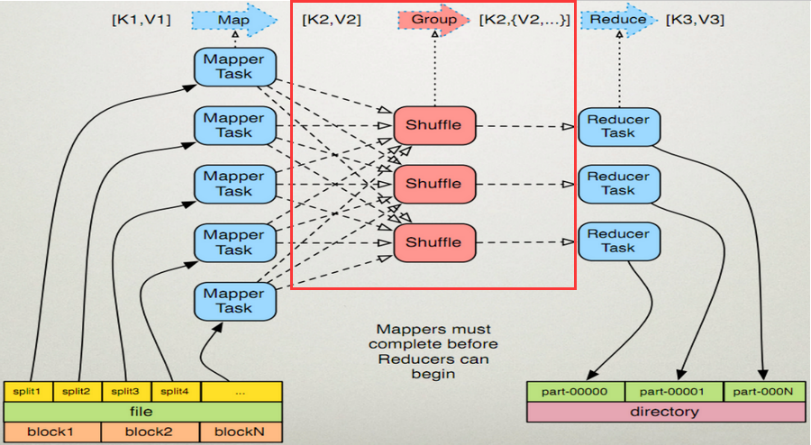
</property>

### mapreduce

MapReduce由JobTracker和TaskTracker组成。JobTracker负责资源管理和作业控制，TaskTracker负责任务的运行。

* 整体架构





1. combine：

combine分为map端和reduce端，作用是把同一个key的键值对合并在一起，能够自己定义。  
combine函数把一个map函数产生的<key,value>对（多个key,value）合并成一个新的<key2,value2>.将新的<key2,value2>作为输入到reduce函数中.

这个value2亦可称之为values，由于有多个。这个合并的目的是为了降低网络传输。

1. partition：是切割map每一个节点的结果，依照key分别映射给不同的reduce。也是能够自己定义的。这里事实上能够理解归类。
2. shuffle：

shuffle就是map和reduce之间的过程，包括了两端的combine和partition。

Map的结果，会通过partition分发到Reducer上，Reducer做完Reduce操作后。

通过OutputFormat，进行输出shuffle阶段的主要函数是fetchOutputs(),这个函数的功能就是将map阶段的输出，copy到reduce 节点本地。

1. input split：

在进行map计算之前，mapreduce会根据输入文件计算输入分片（input split），每个输入分片（input split）针对一个map任务。

输入分片（input split）存储的并非数据本身，而是一个分片长度和一个记录数据的位置的数组。

split的大小时默认和hdfs的block块大小一致，但是可以通过配置文件自己设置。

* 主要组件

InputFormat类：分割成多个splits和每行怎么解析。

Mapper类：对输入的每对<key,value>生成中间结果。

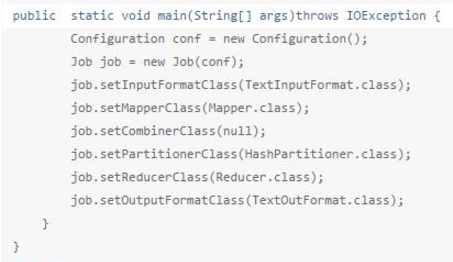
Combiner类：在map端，对相同的key进行合并。

Partitioner类：在shuffle过程中，将按照key值将中间结果分为R份，每一份都由一个reduce去完成。

Reducer类：对所有的map中间结果，进行合并。

OutputFormat类：负责输出结果格式。

编程框架如下：



* 针对MapReduce的缺点，YARN解决了什么？

MapReduce由以下缺点：

JobTracker挂掉，整个作业挂掉，存在单点故障

JobTracker既负责资源管理又负责作业控制，当作业增多时，JobTracker内存是扩展的瓶颈

map task全部完成后才能执行reduce task，造成资源空闲浪费

YARN设计考虑以上缺点，对MapReduce重新设计：

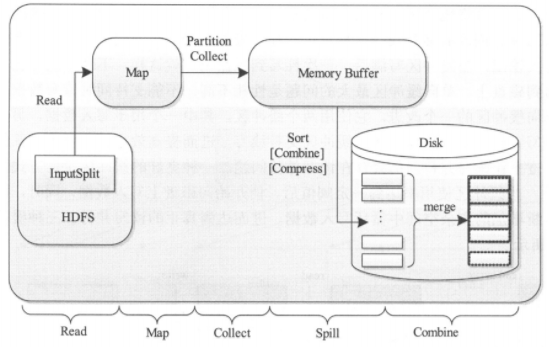
将JobTracker职责分离，ResouceManager全局资源管理，ApplicationMaster管理作业的调度

对ResouceManager做了HA设计

设计了更细粒度的抽象资源容器Container

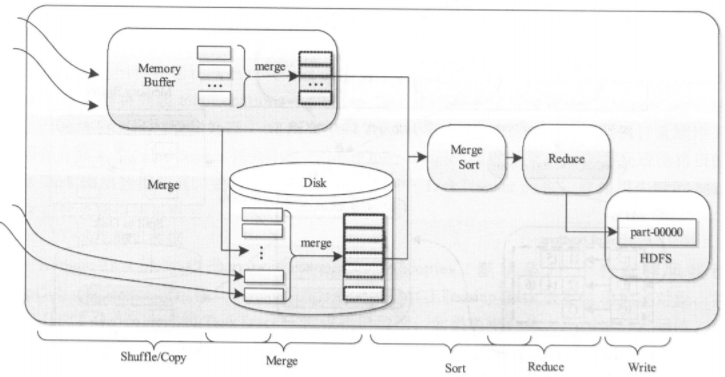
* MapTask

MapTask的整体计算流程如下图所示，共分为5个阶段，分别是：   
1.Read阶段：Map Task通过用户编写的RecordReader，从输入InputSplit中解析出一个个key/value   
2.Map阶段：该阶段主要是将解析出的key/value交给用户编写的map()函数处理，并产生一系列新的key/value   
3.Collect阶段：在用户编写的map()函数中，当数据处理完成后，一般会调用OutputCollector.collect()输出结果。在该函数内部，它会将生成的key/value分片（通过调用Partitioner），并写入一个环形内存缓冲区中。   
4.Spill阶段：即“溢写”，当环形缓冲区满后，MapReduce会将数据写到本地磁盘上，生成一个临时文件。需要注意的是，将数据写入本地磁盘之前，先要对数据进行一次本地排序，并在必要时对数据进行合并、压缩等操作。   
5.Combine阶段：当所有数据处理完成后，Map Task对所有临时文件进行一次合并，以确保最终只会生成一个数据文件。



* ReduceTask

Reduce Task的整体计算流程如图所示，共分为5个阶段   
1.Shuffle阶段：也称为Copy阶段。Reduce Task从各个MapTask上远程拷贝一片数据，并针对某一片数据，如果其大小超过一定阈值，则写到磁盘上，否则直接放到内存中。   
2.Merge阶段：在远程拷贝数据的同时，Reduce Task启动了两个后台线程对内存和磁盘上的文件进行合并，以防止内存使用过多或磁盘上文件过多。   
3.Sort阶段：按照MapReduce语义，用户编写的reduce（）函数输入数据是按key进行聚集的一组数据。为了将key相同的数据聚在一起，Hadoop采用了基于排序的策略。由于各个MapTask已经实现对自己的处理结果进行了局部排序，因此，ReduceTask只需对所有数据进行一次归并排序即可。   
4.Reduce阶段：在该阶段中，ReduceTask将每组数据依次交给用户编写的reduce（）函数处理   
5.Write阶段：reduce（）函数将计算结果写到HDFS上



### yarn

YARN（Yet Another Resource Negotiator）是一个通用的资源管理平台，可为各类计算框架提供资源的管理和调度。

其核心出发点是为了分离资源管理与作业调度/监控，实现分离的做法是拥有一个全局的资源管理器（ResourceManager，RM），以及每个应用程序对应一个的应用管理器（ApplicationMaster，AM），应用程序由一个作业（Job）或者Job的有向无环图（DAG）组成。

YARN可以将多种计算框架(如离线处理MapReduce、在线处理的Storm、迭代式计算框架Spark、流式处理框架S4等) 部署到一个公共集群中，共享集群的资源。

1. 资源的统一管理和调度：

集群中所有节点的资源(内存、CPU、磁盘、网络等)抽象为Container。计算框架需要资源进行运算任务时需要向YARN申请Container， YARN按照特定的策略对资源进行调度进行Container的分配。

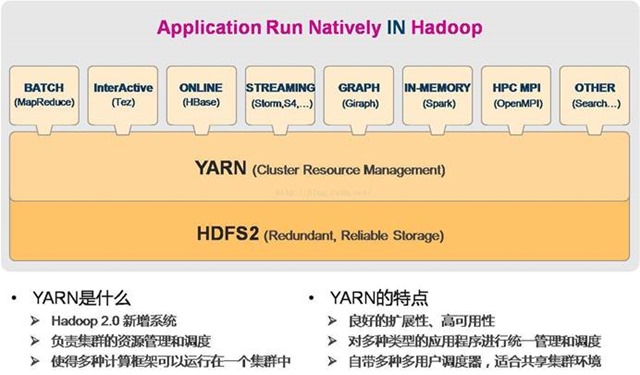
1. 资源隔离：

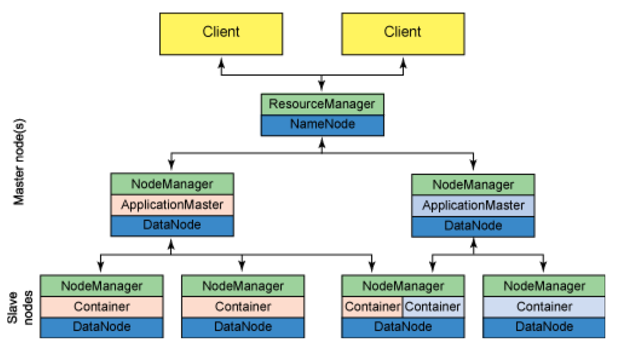
YARN使用了轻量级资源隔离机制Cgroups进行资源隔离以避免相互干扰，一旦Container使用的资源量超过事先定义的上限值，就将其杀死。

YARN是对Mapreduce V1重构得到的，有时候也成为MapReduce V2。

YARN可以看成一个云操作系统，由一个ResourceManager和多个NodeManager组成， 它负责管理所有NodeManger上多维度资源， 并以Container(启动一个Container相当于启动一个进程)方式分配给应用程序启动ApplicationMaster(相当于主进程中运行逻辑) 或运行ApplicationMaster切分的各Task(相当于子进程中运行逻辑)。

* 系统架构





### 使用场景

* 大数据量存储：分布式存储
* 日志处理：Hadoop擅长这个
* 海量计算：并行计算
* ETL：数据抽取到oracle、mysql、DB2、mongdb及主流数据库
* 使用HBase做数据分析:用扩展性应对大量的写操作—Facebook构建了基于HBase的实时数据分析系统
* 机器学习：比如Apache Mahout项目
* 搜索引擎：hadoop + lucene实现
* 数据挖掘：目前比较流行的广告推荐,
* 用户细分特征建模
* 个性化广告推荐,
* 智能仪器推荐

## hbase

HBase是Hadoop的生态系统，是建立在Hadoop文件系统（HDFS）之上的分布式、面向列的数据库，通过利用Hadoop的文件系统提供容错能力。如果你需要进行实时读写或者随机访问大规模的数据集的时候，请考虑使用HBase！

### 基础

数十亿行\*数百万列\*数千个版本=TB级或PB级的存储

<https://www.w3cschool.cn/hbase_doc/>



checkAndPut

row lock

HBase是一个高可靠、高性能、面向列、可伸缩的分布式存储系统，利用HBase技术可在廉价的PC Server上搭建大规模结构化存储集群。

HBase利用Hadoop HDFS作为其文件存储系统；

HBase利用Hadoop MapReduce来处理HBase中的海量数据；

HBase利用Zookeeper作为协同服务。

介于nosql和RDBMS之间，仅能通过主键(row key)和主键的range来检索数据，仅支持单行事务(可通过hive支持来实现多表join等复杂操作)。主要用来存储非结构化和半结构化的松散数据。

1. row key

与nosql数据库们一样,row key是用来检索记录的主键。访问hbase table中的行，只有三种方式：

1.通过单个row key访问

2.通过row key的range

3.全表扫描

Row key行键 (Row key)可以是任意字符串(最大长度是 64KB，实际应用中长度一般为 10-100bytes)，在hbase内部，row key保存为字节数组。

存储时，数据按照Row key的字典序(byte order)排序存储。设计key时，要充分排序存储这个特性，将经常一起读取的行存储放到一起。(位置相关性)

注意：

字典序对int排序的结果是1,10,100,11,12,13,14,15,16,17,18,19,2,20,21,…,9,91,92,93,94,95,96,97,98,99。要保持整形的自然序，行键必须用0作左填充。

行的一次读写是原子操作 (不论一次读写多少列)。这个设计决策能够使用户很容易的理解程序在对同一个行进行并发更新操作时的行为。

1. 列族

hbase表中的每个列，都归属与某个列族。列族是表的chema的一部分(而列不是)，必须在使用表之前定义。列名都以列族作为前缀。例如courses:history ， courses:math 都属于 courses 这个列族。

访问控制、磁盘和内存的使用统计都是在列族层面进行的。实际应用中，列族上的控制权限能 帮助我们管理不同类型的应用：我们允许一些应用可以添加新的基本数据、一些应用可以读取基本数据并创建继承的列族、一些应用则只允许浏览数据（甚至可能因 为隐私的原因不能浏览所有数据）。

1. 时间戳

HBase中通过row和columns确定的为一个存贮单元称为cell。每个 cell都保存着同一份数据的多个版本。版本通过时间戳来索引。时间戳的类型是 64位整型。时间戳可以由hbase(在数据写入时自动 )赋值，此时时间戳是精确到毫秒的当前系统时间。时间戳也可以由客户显式赋值。如果应用程序要避免数据版本冲突，就必须自己生成具有唯一性的时间戳。每个 cell中，不同版本的数据按照时间倒序排序，即最新的数据排在最前面。

为了避免数据存在过多版本造成的的管理 (包括存贮和索引)负担，hbase提供了两种数据版本回收方式。一是保存数据的最后n个版本，二是保存最近一段时间内的版本（比如最近七天）。用户可以针对每个列族进行设置。

1. cell

由{row key, column( =<family> + <label>), version} 唯一确定的单元。cell中的数据是没有类型的，全部是字节码形式存贮。



1. 常用命令

hadoop fs -ls /hbase

1. 常用命令

启动HBase集群：  
bin/start-hbase.sh  
单独启动一个HMaster进程：  
bin/hbase-daemon.sh start master  
单独停止一个HMaster进程：  
bin/hbase-daemon.sh stop master  
单独启动一个HRegionServer进程：  
bin/hbase-daemon.sh start regionserver  
单独停止一个HRegionServer进程：  
bin/hbase-daemon.sh stop regionserver

### 系统架构

* Client

包含访问Hbase的接口，client维护着一些cache来加快对Hbase的访问，比如regione的位置信息。

* Zookeeper

1. 保证任何时候，集群中只有一个master
2. 存贮所有Region的寻址入口。
3. 实时监控Region Server的状态，将Region server的上线和下线信息实时通知给Master
4. 存储Hbase的schema,包括有哪些table，每个table有哪些column family

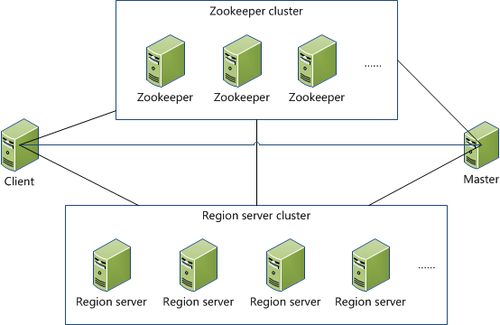
* Master

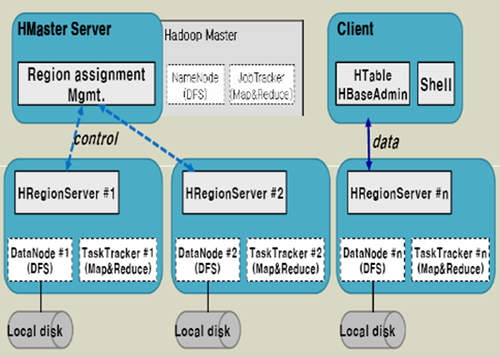
1. 为Region server分配region
2. 负责region server的负载均衡
3. 发现失效的region server并重新分配其上的region
4. GFS上的垃圾文件回收
5. 处理schema更新请求

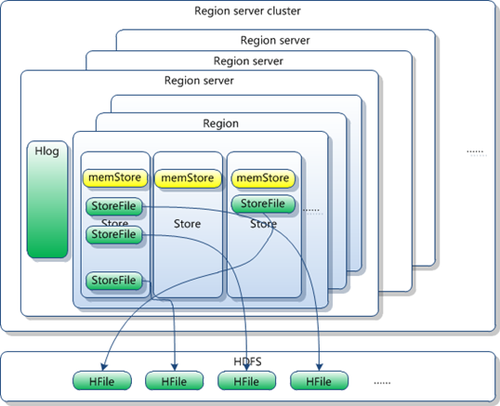
* Region Server

1. Region server维护Master分配给它的region，处理对这些region的IO请求
2. Region server负责切分在运行过程中变得过大的region

可以看到，client访问Hbase上数据的过程并不需要master参与(寻址访问zookeeper和region server，数据读写访问regione server)，master仅仅维护者table和region的元数据信息，负载很低。







### 存储架构

HBase上的数据是以StoreFile(HFile)二进制流的形式存储在HDFS上block块儿中

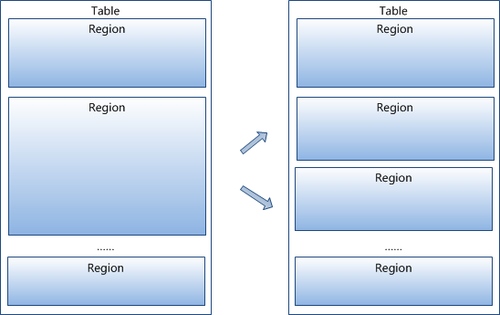
* 存储结构

HBase中的存储包括HMaster、HRegionSever、HRegion、HLog、Store、MemStore、StoreFile、HFile等

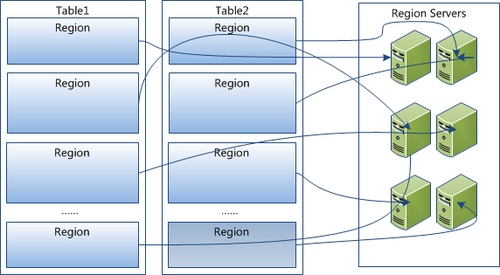
1. Table中的所有行都按照row key的字典序排列。
2. Table 在行的方向上分割为多个Hregion。

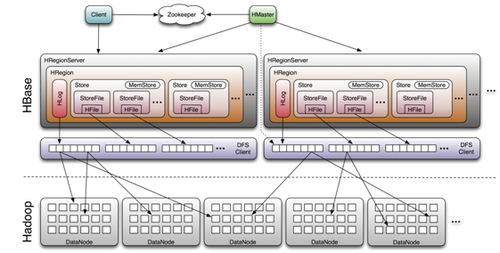


1. region按大小分割的，每个表一开始只有一个region，随着数据不断插入表，region不断增大，当增大到一个阀值的时候，Hregion就会等分会两个新的Hregion。当table中的行不断增多，就会有越来越多的Hregion。



1. HRegion是Hbase中分布式存储和负载均衡的最小单元。最小单元就表示不同的Hregion可以分布在不同的HRegion server上。但一个Hregion是不会拆分到多个server上的。

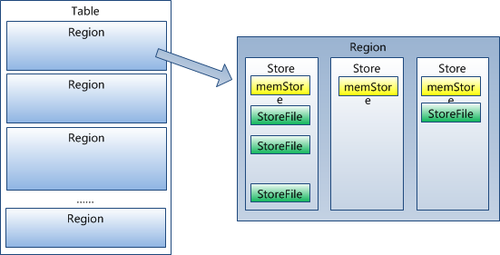




1. HRegion虽然是分布式存储的最小单元，但并不是存储的最小单元。

　事实上，HRegion由一个或者多个Store组成，每个store保存一个columns family。

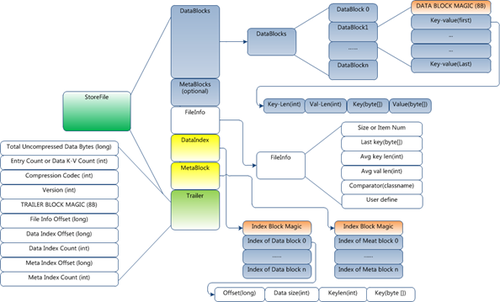
　每个Strore又由一个memStore和0至多个StoreFile组成。如图：



1. StoreFile以HFile格式保存在HDFS上。

在分布式文件系统hdfs上，HFile对应列族，一个列族可以有多个HFile文件，一个HFile文件是一个列族中的内容。

HFile的格式：



HFile分为6个部分：

Data Block 段–保存表中的数据，这部分可以被压缩

Meta Block 段 (可选的)–保存用户自定义的kv对，可以被压缩。

File Info 段–Hfile的元信息，不被压缩，用户也可以在这一部分添加自己的元信息。

Data Block Index 段–Data Block的索引。每条索引的key是被索引的block的第一条记录的key。

Meta Block Index段 (可选的)–Meta Block的索引。

Trailer段–这一段是定长的。保存了每一段的偏移量，读取一个HFile时，会首先读取Trailer，Trailer保存了每个段的起始 位置(段的Magic Number用来做安全check)，然后，DataBlock Index会被读取到内存中，这样，当检索某个key时，不需要扫描整个HFile，而只需从内存中找到key所在的block，通过一次磁盘io将整个 block读取到内存中，再找到需要的key。DataBlock Index采用LRU机制淘汰。

1. HLog(WAL log)

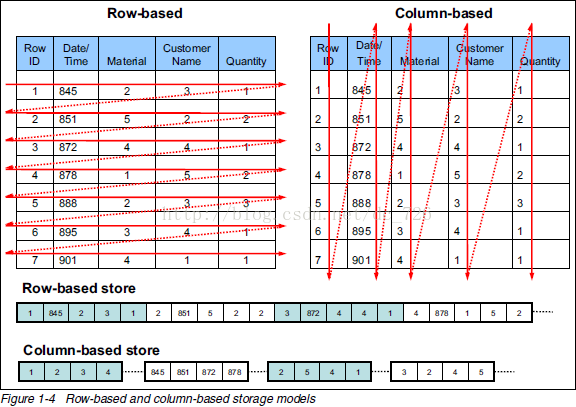
　　WAL 意为Write ahead log(http://en.wikipedia.org/wiki/Write-ahead\_logging)，类似mysql中的binlog,用来 做灾难恢复只用，Hlog记录数据的所有变更,一旦数据修改，就可以从log中进行恢复。

　　每个Region Server维护一个Hlog,而不是每个Region一个。这样不同region(来自不同table)的日志会混在一起，这样做的目的是不断追加单个 文件相对于同时写多个文件而言，可以减少磁盘寻址次数，因此可以提高对table的写性能。带来的麻烦是，如果一台region server下线，为了恢复其上的region，需要将region server上的log进行拆分，然后分发到其它region server上进行恢复。

HLog文件就是一个普通的Hadoop Sequence File，Sequence File 的Key是HLogKey对象，HLogKey中记录了写入数据的归属信息，除了table和region名字外，同时还包括 sequence number和timestamp，timestamp是”写入时间”，sequence number的起始值为0，或者是最近一次存入文件系统中sequence number。HLog Sequece File的Value是Hbase的KeyValue对象，即对应HFile中的KeyValue。

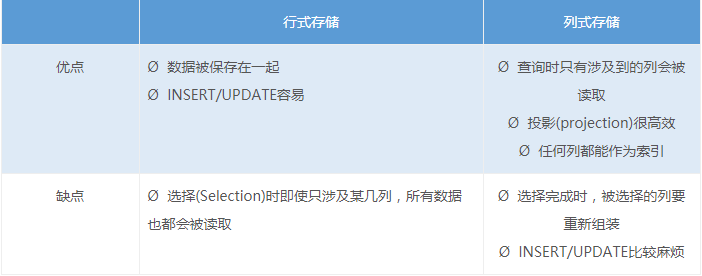
* 列式存储

示例：



可以看出，行式存储下一张表的一条数据都是放在一起的，但列式存储下都被分开保存了

优缺点：



### 关键算法

* region定位

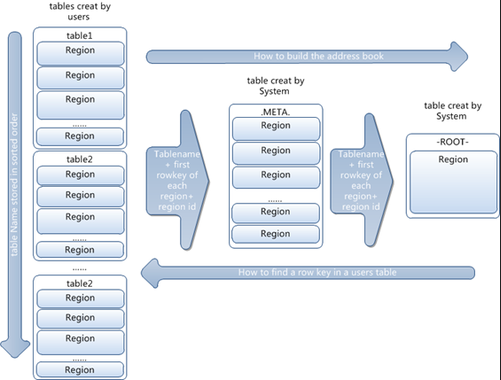
系统如何找到某个row key (或者某个 row key range)所在的region

bigtable 使用三层类似B+树的结构来保存region位置。

第一层是保存zookeeper里面的文件，它持有root region的位置。

第二层root region是.META.表的第一个region，其中保存了.META.表其它region的位置。通过root region，我们就可以访问.META.表的数据。

第三层.META.，它是一个特殊的表，保存了Hbase中所有数据表的region 位置信息。



* 写过程

Hbase使用MemStore和StoreFile存储对表的更新。

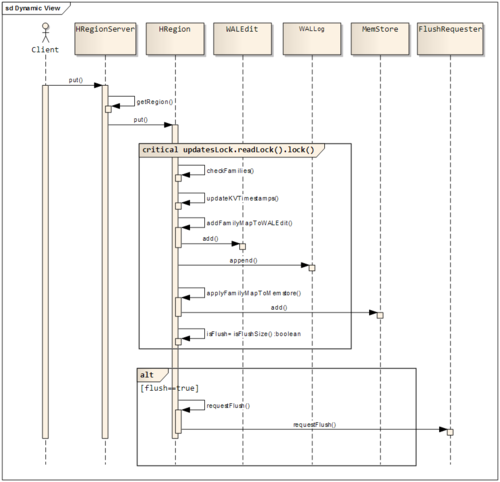
数据在更新时首先写入Log(WAL log)和内存(MemStore)中，MemStore中的数据是排序的，当MemStore累计到一定阈值时，就会创建一个新的MemStore，并且将老的MemStore添加到flush队列，由单独的线程flush到磁盘上，成为一个StoreFile。于此同时，系统会在zookeeper中 记录一个redo point，表示这个时刻之前的变更已经持久化了。(minor compact)

当系统出现意外时，可能导致内存(MemStore)中的数据丢失，此时使用Log(WAL log)来恢复checkpoint之后的数据。

前面提到过StoreFile是只读的，一旦创建后就不可以再修改。因此Hbase的更新其实是不断追加的操作。当一个Store中的 StoreFile达到一定的阈值后，就会进行一次合并(major compact),将对同一个key的修改合并到一起，形成一个大的StoreFile，当StoreFile的大小达到一定阈值后，又会对 StoreFile进行split，等分为两个StoreFile。

　　由于对表的更新是不断追加的，处理读请求时，需要访问Store中全部的StoreFile和MemStore，将他们的按照row key进行合并，由于StoreFile和MemStore都是经过排序的，并且StoreFile带有内存中索引，合并的过程还是比较快。

写请求处理过程：



* 合并

删除一条记录，就会在该记录上打上标记，被打上标记的记录就成了墓碑记录，该记录使用get和scan查询不到，但还是在HFile中。只有进行大合并的时候才会删除HFile中的墓碑记录。

大合并指定region的一个列族的所有HFile.合并完成后，这个列族的所有HFile文件合并成一个HFile文件，可以在shell中手动触发，但该动作相当耗资源。

小合并是将多个小的HFile文件内容读取出来合并生成一个大的HFile,把新文件设置成激活状态，然后删除小的HFile。

### 使用场景

* 海量数据存储，低延时查询

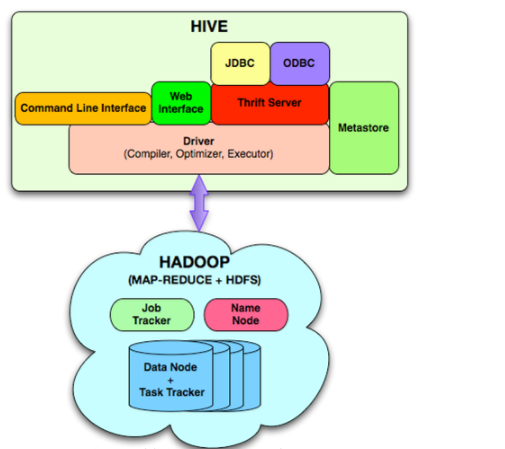
## hive

### 基础

Hive是一个构建在Hadoop基础设施之上的数据仓库。通过Hive可以使用HQL语言查询存放在HDFS上的数据。HQL是一种类SQL语言，这种语言最终被转化为Map/Reduce.

### 系统架构

* 架构



1. client

1.CLI(hive shell)、command line interface（命令行接口）  
2.JDBC/ODBC(java访问hive)   
3.WEBUI(浏览器访问hive)

1. Meta store

hive元数据存放地，包括metastore服务、后台数据存储两部分。  
元数据包括：表名、表所属的数据库（默认是default）、表的拥有者、列、分区字段、表的类型（是否是外部表）、表的数据所在的目录等。  
默认存储在自带的derby数据库中，推荐使用采用MySQL存储Metastore。

1. Driver

包含：解析器、编译器、优化器、执行器；  
1.解析器：将SQL字符串转换成抽象语法树AST，这一步一般都用第三方工 具库完成，比如antlr；对AST进行语法分析，比如表是否存在、字段是否 存在、SQL语义是否有误(比如select中被判定为聚合的字段在group by中是 否有出现)  
2.编译器：将AST编译生成逻辑执行计划

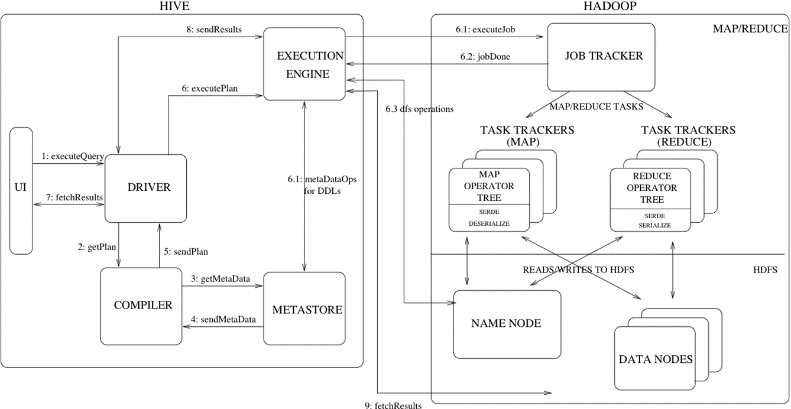
3.优化器：对逻辑执行计划进行优化  
4.执行器：把逻辑执行计划转换成可以运行的物理计划。对于Hive来说，就是MR/TEZ/Spark

1. 举例

select substring(ip,0,4) ip\_prex from bg\_log

执行顺序：首先在metastore查询-->sql解析-->查询优化--->物理计划-->执行MapReduce

* 执行流程



### 存储结构

* Metastore

metastore中几个重要的表：

1. dbs：存放hive所有数据库信息

2. tbls：存放hive所有表格信息

3. table\_params：存放hive所有表格的参数信息

4. columns\_v2：存放表格的字段信息

* Hdfs

### 关键算法

### 使用场景

* 数据的离线处理；比如：日志分析，海量结构化数据离线分析
* Hive的执行延迟比较高，因此hive常用于数据分析的，对实时性要求不高的场合；
* Hive优势在于处理大数据，对于处理小数据没有优势，因为Hive的执行延迟比较高。

## redis

### 基础

* 单线程-多路复用io模型

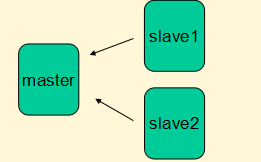
单一线程只能用到一个cpu核心，所以可以在同一个多核的服务器中，可以启动多个实例。

select，poll，epoll都是IO多路复用的机制。所谓I/O多路复用机制，就是说通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

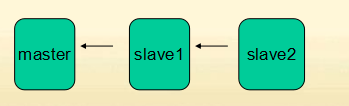
* 集群扩容与收缩redis-trib.rb
* 管理及监控工具treeNMS
* 主从结构中，slave不能写数据

主从复制在redis中的实现非常简单，redis持久化的实现原理是rdb内存快照配合aof日志文件两种方式，本质上只需要将rdb的dump文件和aof日志文件在redis集群中共享就可以实现redis集群，从而达到负载的作用。另外redis主从有两种方式，分别是：

1）一个master跟着多个slave



2）master只跟着一个slave1，slave1跟着另外一个slave2，于此同理



两种方式相比较而言，第二种会相对好一点，因为当master宕机的时候，只需要将master切换到slave1即可，redis集群中其他的slave不需要做任何改变。而对于第一种方式而言，相对来说就略微麻烦一点，因为在所有slave间选择其一作为新任master之后，剩余的slave需要指向新的master。

* sentienl监控的使用

sentienl的作用就是在master宕机的时候，由sentienl来切换新的master。

其工作原理是：在指定的连接超时次数时，确认master为宕机状态，根据slave的优先级，选择优先级最高的slave，将其更改为master，并且将其他的slave改为指向该新的master上。

### 案例

* 实现分布式锁

1. setnx key value

将 key 的值设为 value ，当且仅当 key 不存在。

返回值：

设置成功，返回 1

设置失败，返回 0

1. DEL key

* 实现高并发更新

1. 单线程；主从结构中，slave不能写数据；hash slots保证同一个key一定在同一台机器上
2. redis+lua的原子操作
3. lua脚本update\_sum.lua:

--[[

KEYS[1]: 要更新的key

ARGV[1]: 要增加的值

ARGV[2]: 最大值

返回累计结果，-1超过最大值

--]]

local k = KEYS[1];

local v1 = ARGV[1];

local v2 = ARGV[2];

local v3 = redis.call("GET",k);

local v4 = v1 + v3;

if(v4 > v2) then

return -1;

end

redis.call("SET",k,v4);

return v4;

1. redis-cli --eval /update\_sum.lua

或

jedisCluster DefaultRedisScript

* 生成全局唯一id

INCRBY key increment

将 key 所储存的值加上增量 increment 。

如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 INCRBY 命令。

返回值：

加上 increment 之后， key 的值。

1. String key = dateNow("yyyymmdd")
2. String id = key + {INCRBY key 1}
3. EXPIRE key 24\*60\*60

### 案例-滑动时间窗算法

一种时间窗口累计的实现方法，首先通过滑动时间窗口累计算法对交易数据信息进行统计分析，然后判断时间窗口的位置变换范围、窗体或步长的大小来选择适用减法引擎算法或加法引擎算法，最后在Redis或Hbase中完成交易数据累计值和获取值的统计分析与快速查询。解决了多维度多场景组合下，SQL查询算法不灵活、查询吞吐量不高、查询粒度不够精确的问题，实现了获取时间窗口累计值的快速响应，提高了准确性。

* Redis实现滑动时间窗算法（相减）

1. 对于有时效性的统计，因为时间边界一直在变化，传统上，采用累加的方式，我们没法保存统计值，因为随着时间的推移，统计值已经混淆了，所以会取出所有符合

的请求，然后在jvm内存里计算，这样会出现2个不可控问题（尤其对于热度数据），1是redis的io和网络消耗大，2是jvm计算时间不确定，jvm内存消耗不可控。

这里采用相减的滑动时间窗来实现，计算结果=历史统计值-前推时间n的历史统计值，很大程度减轻了请求的网络负担。

1. 数据结构（有序集合）

key a:00000:xxxxxxx

zset score: yyyyMMddHHmm

value: 累计数据

1. 数据累计

清除过期数据，保留最大过期数据：

ct=zcount key 0 (<currenttime> -<expiretime>)

if ct>2则zremrangebyrank key 0 (ct-2)

获取最近的累计（acccumulated value）：

zrang key -1 -1

新增累计数据：

zadd key <time> (currentvalue + acccumulated value)

设置key 的过期时间：

expire key expires

1. 数据获取

获取过期的累计：

zrevrangebyscore key（<currenttime> -<expiretime>）0 limit 0 1

获取最近的累计：

zrang key -1 -1

将第二个值与第一个值相减再加上当前发生额即滑动时间窗口的累计数据

1. 示例

如下图所示，我们统计最近10分钟的交易笔数(或者交易金额)，计算结果=历史总交易笔数-10分钟前的历史总交易笔数



* Hbase实现滑动时间窗算法(相加)

采用相加的算法，步长1分钟，具体如下：

1. 数据结构

rowkey:a:00000:xxxxxxx

qualifier: yyyyMMddHHmm

ttl:时间窗范围

create 'cust1', {NAME => 'amount',TTL =>'1800'}

1. 数据累计

String qualifier= dateNow("yyyyMMddHHmm ")

Long value=要增加的值

incr 'cust1', 'rowkey', 'amount:qualifier', value

1. 数据获取

Long sum=0L

List< Long> list= get 'cust1',' rowkey'

for(Long li:list>) sum+=li

或

String qualifierStart= dateNow("yyyyMMddHHmm ")-ttl

String qualifierEnd= dateNow("yyyyMMddHHmm ")

List< Long> list = get 'cust1', ' rowkey', {FILTER => "QualifierFilter(>=, 'binary: qualifierStart') AND QualifierFilter(<=, 'binary: qualifierEnd')"}

### 使用场景

* 缓存——热数据
* 计数器
* 队列
* 位操作（大数据处理）
* 分布式锁与单线程机制
* 最新列表
* 排行榜
* 发布/订阅

## elasticsearch

### Elasticsearch

是一个实时的分布式搜索和分析引擎。是一个基于Lucene的搜索服务器。

1. 分布式实时文件存储，可将每一个字段存入索引，使其可以被检索到。
2. 实时分析的分布式搜索引擎。

分布式：索引分拆成多个分片，每个分片可有零个或多个副本。集群中的每个数据节点都可承载一个或多个分片，并且协调和处理各种操作；

负载再平衡和路由在大多数情况下自动完成。

1. 可以扩展到上百台服务器，处理PB级别的结构化或非结构化数据。
2. 支持插件机制，分词插件（elasticsearch-analysis-ik）、同步插件（elasticsearch-jdbc，logstash-input-jdbc）、Hadoop插件、可视化插件（elasticsearch-head）等。
3. settings修改索引库默认配置
4. mapping,就是对索引库中索引的字段名称及其数据类型进行定义，类似于mysql中的表结构信息。

### Lucene

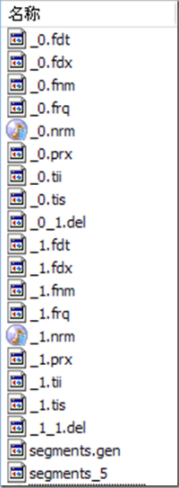
Lucene是一款高性能的、可扩展的信息检索（IR）工具库。信息检索是指文档搜索、文档内信息搜索或者文档相关的元数据搜索等操作。

### Lucene索引文件格式

Lucene的索引过程，就是按照全文检索的基本过程，将倒排表写成此文件格式的过程。

Lucene的搜索过程，就是按照此文件格式将索引进去的信息读出来，然后计算每篇文档打分(score)的过程。

* 下图就是Lucene生成的索引的一个实例：



Lucene的索引结构是有层次结构的，主要分以下几个层次：

1. 索引(Index)：

在Lucene中一个索引是放在一个文件夹中的。

如上图，同一文件夹中的所有的文件构成一个Lucene索引。

1. 段(Segment)：

一个索引可以包含多个段，段与段之间是独立的，添加新文档可以生成新的段，不同的段可以合并。

如上图，具有相同前缀文件的属同一个段，图中共两个段 "\_0" 和 "\_1"。

segments.gen和segments\_5是段的元数据文件，也即它们保存了段的属性信息。

1. 文档(Document)：

文档是我们建索引的基本单位，不同的文档是保存在不同的段中的，一个段可以包含多篇文档。

新添加的文档是单独保存在一个新生成的段中，随着段的合并，不同的文档合并到同一个段中。

1. 域(Field)：

一篇文档包含不同类型的信息，可以分开索引，比如标题，时间，正文，作者等，都可以保存在不同的域里。

不同域的索引方式可以不同，在真正解析域的存储的时候，我们会详细解读。

1. 词(Term)：

词是索引的最小单位，是经过词法分析和语言处理后的字符串。

* Lucene的索引结构中，即保存了正向信息，也保存了反向信息。

1. 所谓正向信息：

按层次保存了从索引，一直到词的包含关系：索引(Index) –> 段(segment) –> 文档(Document) –> 域(Field) –> 词(Term)

也即此索引包含了那些段，每个段包含了那些文档，每个文档包含了那些域，每个域包含了那些词。

既然是层次结构，则每个层次都保存了本层次的信息以及下一层次的元信息，也即属性信息，比如一本介绍中国地理的书，应该首先介绍中国地理的概况，以及中国包含多少个省，每个省介绍本省的基本概况及包含多少个市，每个市介绍本市的基本概况及包含多少个县，每个县具体介绍每个县的具体情况。

如上图，包含正向信息的文件有：

segments\_N保存了此索引包含多少个段，每个段包含多少篇文档。

XXX.fnm保存了此段包含了多少个域，每个域的名称及索引方式。

XXX.fdx，XXX.fdt保存了此段包含的所有文档，每篇文档包含了多少域，每个域保存了那些信息。

XXX.tvx，XXX.tvd，XXX.tvf保存了此段包含多少文档，每篇文档包含了多少域，每个域包含了多少词，每个词的字符串，位置等信息。

1. 所谓反向信息：

保存了词典到倒排表的映射：词(Term) –> 文档(Document)

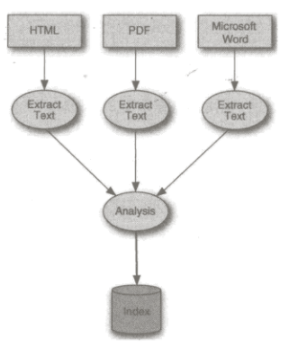
如上图，包含反向信息的文件有：

XXX.tis，XXX.tii保存了词典(Term Dictionary)，也即此段包含的所有的词按字典顺序的排序。

XXX.frq保存了倒排表，也即包含每个词的文档ID列表。

XXX.prx保存了倒排表中每个词在包含此词的文档中的位置。

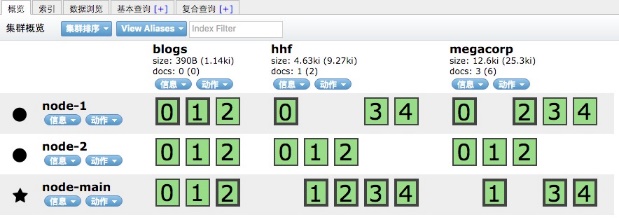
### Lucene索引过程



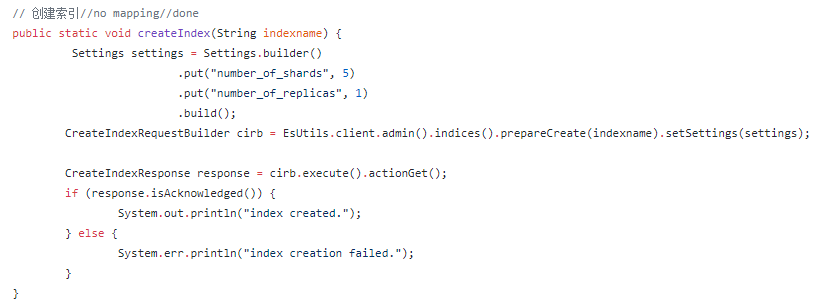
Lucene索引过程分为3个主要步骤：将原始文档转换成文本，分析文本，将分析好的文本保存至索引中

### 集群监控

elasticsearch-head插件



### 索引示例





# tensorflow