# Hadoop

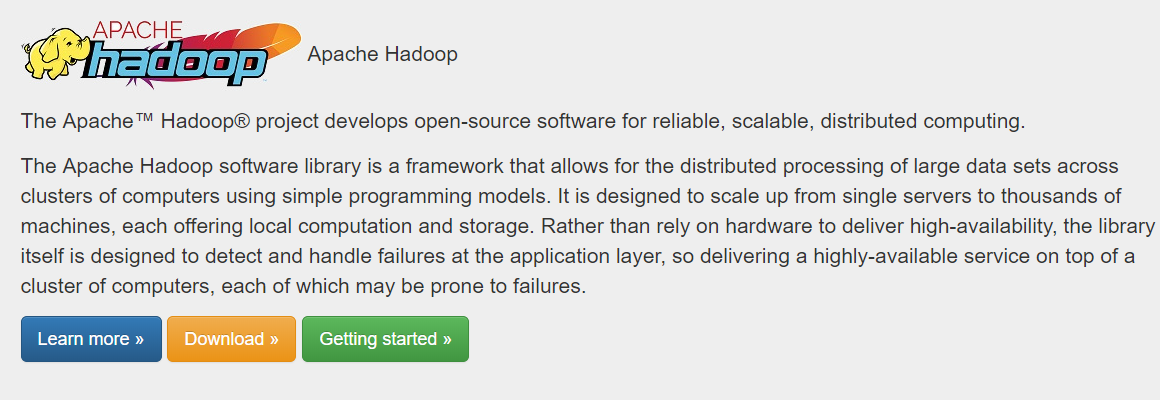
**01-初识Hadoop**

Mar 15, 2019

关于大数据的学习终于开始了，对java后台开发比较熟悉的同学会发现，部分的组件比如zookeeper就是从Hadoop生态圈中诞生的，但是纵观hadoop的生态圈，大多数还是比较陌生，本节来说一说比较核心的组件，找到学习的入口。

**一、概述**

我们来到它的官网来看看Hadoop到底是什么。



大概就是说：Haadoop项目旨在发展可靠的、可扩展的、分布式计算的开源软件。允许用简单的编程跨越集群来分布式处理大数据集。可以从单体服务扩展到上千台机器，每个机器都可以提供本地计算和存储。相比于依赖硬件来保证高可用，Hadoop是在应用层监测和处理错误从而保证可靠性，允许单个机器出现故障。

可以看到，Hadoop旨在提供高可用、可扩展、分布式计算一种能力，是应对海量数据的存储以及处理的一个方案。

这个Hadoop project包含了几个核心的子模块：

* Hadoop Common: 为其他的模块提供基础支持。
* Hadoop Distributed File System (HDFS™): 一个高吞吐量的分布式文件系统
* Hadoop YARN: 一个作业调度和集群资源管理的框架
* Hadoop MapReduce: 基于YARN实线的用于并行处理大数据集的系统

**二、核心组件之HDFS**

* 源自于谷歌的GFS论文，论文发表于2003年10月
* HDFS是GFS的克隆版
* HDFS特点：扩展性&容错性&海量数据存储

HDFS是一种分布式文件系统，文件系统是什么，我们需要知道他的基本功能是：管理和调度文件的存储空间，提供文件的逻辑结构、物理结构和存储方法;实现文件从标识到实际地址的映射。也就是对文件进行管理和调度的。

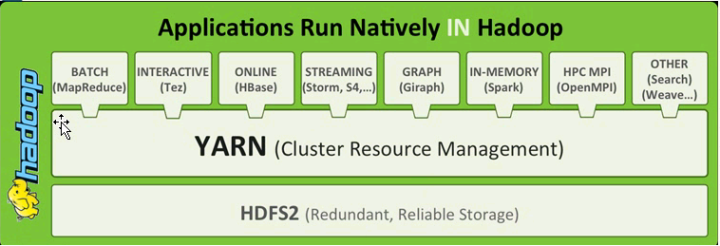
海量的数据，我们需要良好的扩展性来存放，也需要容错性来防止数据丢失，因此HDFS是在文件系统基本要求的基础上，实现更可靠的存储。

这里简单说一下的它的容错性是如何做到的：将文件切分为指定带线啊哦的数据块并以多副本的方式存储在多个机器上，而数据切分、多副本、容错等操作对用户是透明的。

**三、核心组件之资源调度系统YARN**

* 负责整个集群资源的管理和调度：比如有一个作业要提交上来执行，需要占用多少CPU，多少内存都是YARN来计算完成的。
* 具有扩展性&容错性&多框架资源统一调度

这里注意，YARN可以为上层应用提供统一的资源管理和调度，是什么意思呢？

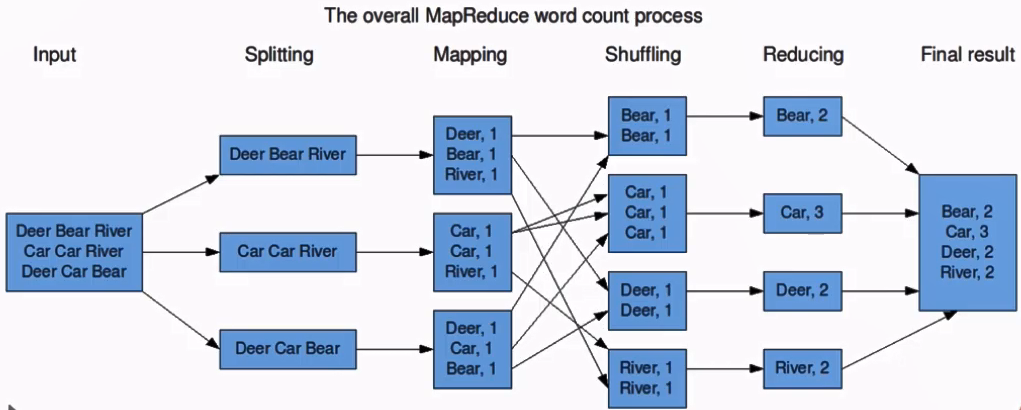


这张图从下面开始，HDFS2其实就是表示hadoop2.x，他是做存储的，往上看有一个YARN，他就是做集群的资源管理的（cluster resource management），再往上看，对于批处理我们可以用mapreduce这种执行引擎，对于交互式的我们可以使用tez，在线的我们可以使用hbase，流处理可以采用storm，图计算用giraph，内存式的计算我们可以使用spark。。。。。

有了YARN之后就可以执行不同类型的计算框架，我们可以把YARN理解为操作系统级别的资源调度框架，可以让更多的计算框架mapreduce，spark，storm等等都运行在同一个集群里面，而且不同的计算框架可以共享同一个hdfs上的数据，享受整体的资源调度。

**四、核心组件之分布式计算框架MapReduce**

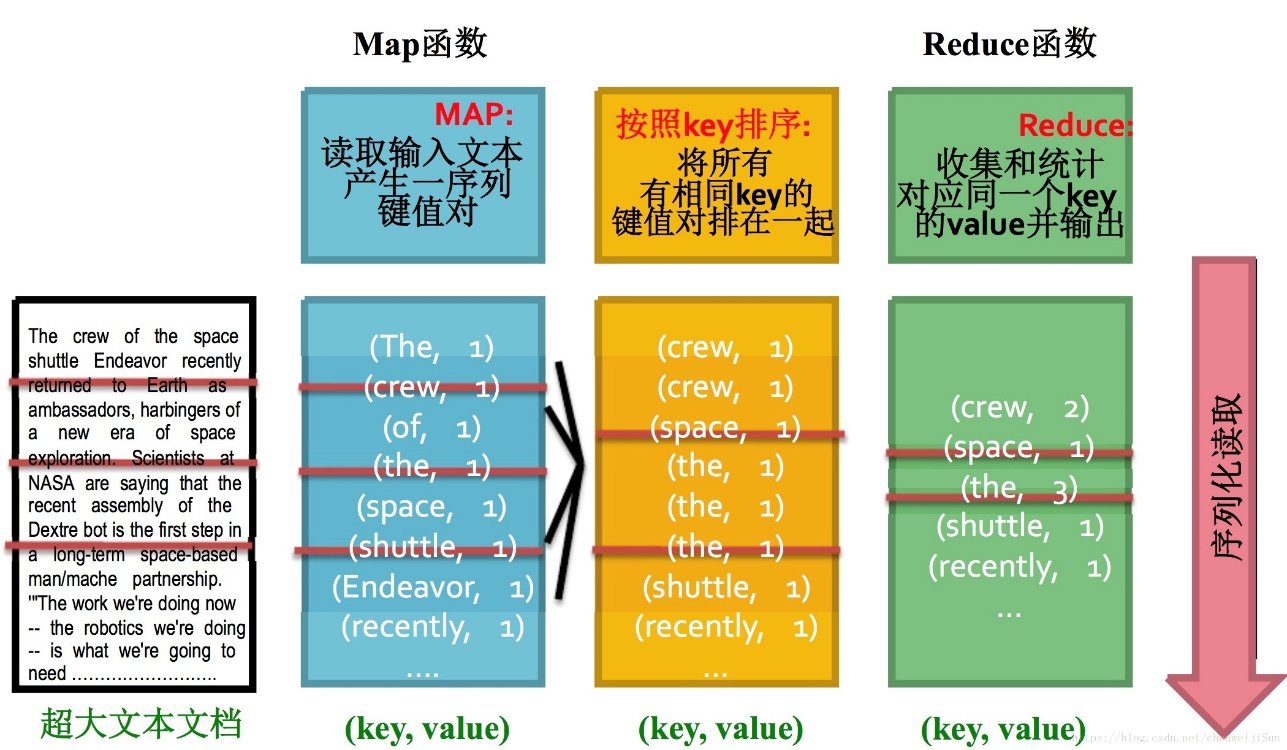
也是根据谷歌的论文实现的。不再赘述。下面我们来看看它是如何简单地统计单词出现频率的：



这是比较经典的一个案例。

大概是分成下面几个环节：

* Map阶段
  + 主要完成key-value对生成，这里是每看到一个单词，就输出(单词，1)的kv对
* 排序阶段
  + 对刚才的kv对进行排序，这样相同单词就在一块儿了
* Reduce阶段
  + 对同一个单词的次数进行汇总，得到(词，频次)对



**五、Hadoop优势**

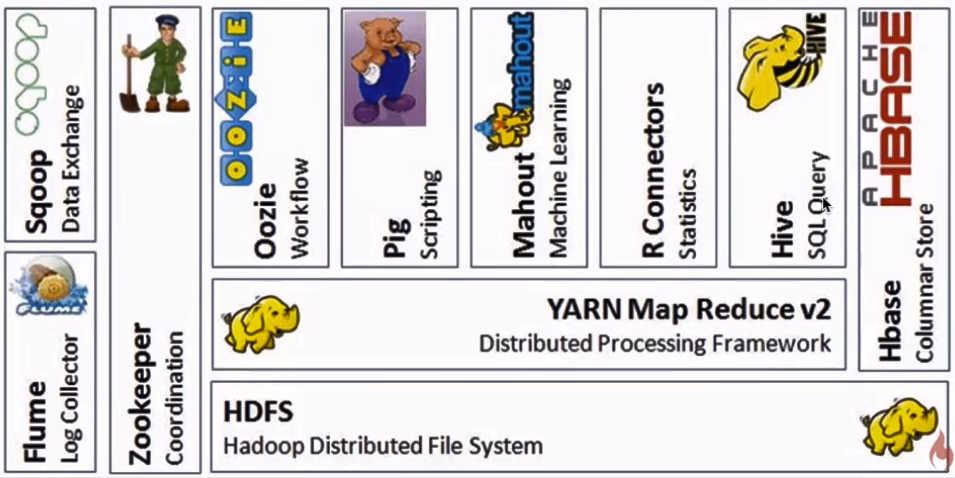
* 高可用性：
  + 数据存储：数据块多副本
  + 数据计算：发生异常的时候会重新调度作业计算
* 高扩展性
  + 可以横向增加机器
* 其他
  + 可以存储在廉价机器上，降低成本
  + 成熟的生态圈

**六、狭义Hadoop和广义Hadoop**

狭义上的Hadoop：是一个适合大数据分布式存储(HDFS)、分布式计算(MapReduce)和资源调度(YARN)的平台。

广义上的Hadoop：指的是Hadoop生态系统。我们要知道，完成一些任务仅靠上面几个是不够的，比如MapReduce只能做离线批处理，要想做实时计算时无法完成的，所以需要其他的软件来完成。Hadoop生态系统是一个很庞大的概念，Hadoop是其中最重要最基础的一个部分；生态系统中的每一个子系统只解决某一个特定的问题域，不搞统一型的一个全能系统，而实小而精的多个小系统。

**七、Hadoop生态圈介绍**



我们从图中可以大概知道每个组件的用途。下面我们来简单说一说。

HDFS是GFS的一种实现，他的完整名字是分布式文件系统，类似于FAT32，NTFS，是一种文件格式，是底层的。

Hive与Hbase的数据一般都存储在HDFS上。hadoop HDFS为他们提供了高可靠性的底层存储支持。

hive不支持更改数据的操作，Hive基于数据仓库，提供静态数据的动态查询。其使用类SQL语言，底层经过编译转为MapReduce程序，在Hadoop上运行，数据存储在HDFS上。

Hbase是Hadoop database，即Hadoop数据库。它是一个适合于非结构化数据存储的数据库，HBase基于列的而不是基于行的模式。

HBase是Google Bigtable的开源实现，类似Google Bigtable利用GFS作为其文件存储系统，HBase利用Hadoop HDFS作为其文件存储系统；Google运行MapReduce来处理Bigtable中的海量数据，HBase同样利用Hadoop MapReduce来处理HBase中的海量数据。

Pig的语言层包括一个叫做PigLatin的文本语言,Pig Latin是面向数据流的编程方式。Pig和Hive类似更侧重于数据的查询和分析，底层都是转化成MapReduce程序运行。

Pig和Hive的区别是Hive是类SQL的查询语言，要求数据存储于表中，而Pig是面向数据流的一个程序语言。

总结：

Hadoop HDFS为HBase提供了高可靠性的底层存储支持，Hadoop MapReduce为HBase提供了高性能的计算能力，Zookeeper为HBase提供了稳定服务和failover机制(zookeeper = 通知机制 + 文件系统)。Pig和Hive还为HBase提供了高层语言支持，使得在HBase上进行数据统计处理变的非常简单。

**八、选型**

* Apache Hadoop
* CDH:Cloudera Distributed Hadoop
* HDP:Hortonworks Data Platform

优先选择CDH，比较全，解决了版本冲突问题。

**九、总结**

我们可以看到，Hadoop的生态圈是极其庞大的，每个组件都是小而精，是为了解决特定问题而产生的，因此需要具体情况具体分析选用什么组件，也需要对核心的组件原理和使用了如指掌，下面我们先逐个来学习一下狭义Hadoop核心组件，从HDFS开始。

# MapReduce（并行计算模型）

其实我们可以从word count这个实例来理解MapReduce。MapReduce大体上分为六个步骤：input, split, map, shuffle, reduce, output。细节描述如下：  
 **1. 输入(input)：**如给定一个文档，包含如下四行：  
Hello Java  
Hello C  
Hello Java

Hello C++

**2. 拆分(split)：**将上述文档中每一行的内容转换为key-value对，即：  
0 - Hello Java  
1 - Hello C  
2 – Hello Java

3 - Hello C++

**3. 映射(map)：**将拆分之后的内容转换成新的key-value对，即：  
(Hello , 1)  
(Java , 1)  
(Hello , 1)  
(C , 1)  
(Hello , 1)  
(Java , 1)  
(Hello , 1)

(C++ , 1)

**4. 派发(shuffle)：**将key相同的扔到一起去，即：  
(Hello , 1)  
(Hello , 1)  
(Hello , 1)  
(Hello , 1)  
(Java , 1)  
(Java , 1)  
(C , 1)  
(C++ , 1)  
注意：这一步需要移动数据，原来的数据可能在不同的datanode上，这一步过后，相同key的数据会被移动到同一台机器上。最终，它会返回一个list包含各种k-value对，即：  
{ Hello: 1,1,1,1}  
{Java: 1,1}  
{C: 1}

{C++: 1}

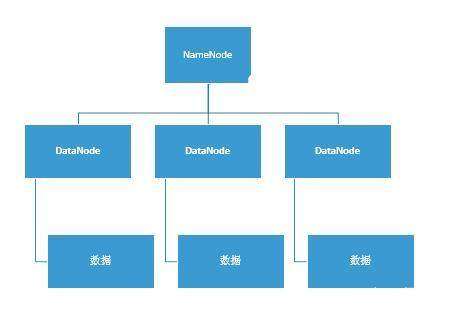
**5. 缩减(reduce)：**把同一个key的结果加在一起。如：  
(Hello , 4)  
(Java , 2)  
(C , 1)

(C++,1)

6. 输出(output): 输出缩减之后的所有结果。

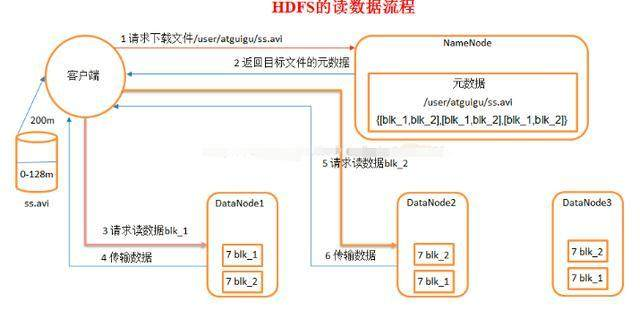
# HDFS（分布式文件系统）

HDFS是Hadoop中自带的分布式文件系统，并且是Hadoop工具的核心基础组件之一，之前的文章对HDFS有简单的介绍，没看过的小伙伴欢迎前往了解。今天，小鸟带大家对HDFS做一些深入的了解，对其内部概念进行阐述。

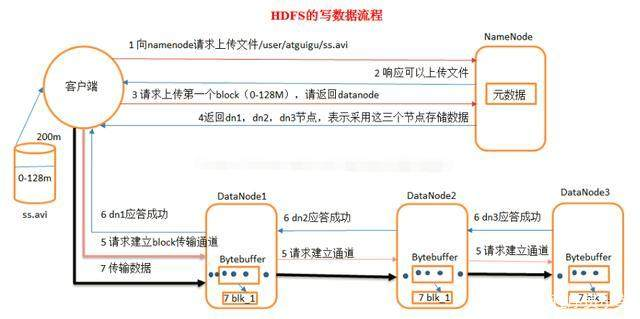
**一、namenode和datanode**

在HDFS中，每个服务器称之为一个节点。而所有的节点都分为两类，一个类是namenode，另一类是datanode。如上图所示：普通模式HDFS集群只有一个namenode和多个datanode，作用分别是管理和工作。

**namenode**

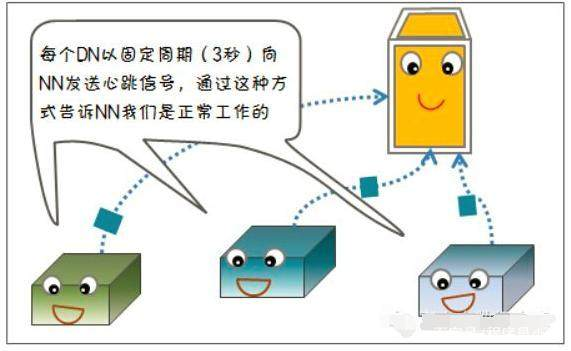
namenode是HDFS的管理节点，其维护着HDFS的命名空间。它会时刻监控着整个HDFS的树状系统以及系统内的所有目录和文件，并将它们以文件的形式保存在磁盘上。

namenode还维护着HDFS中每个数据块所在的节点的信息，其中位置信息不会永久保存。因为HDFS启动的时候每个datanode会向namenode汇报信息。

namenode是整个HDFS的外接接口，客户端在对HDFS进行交互请求之前都要先经过namenode验证。整个HDFS写数据的流程如下图所示：

**datanode**

datanode是HDFS的工作节点，每个datanode都维护着本节点的数据和剩余空间。datanode会随时向namenode汇报本节点的存储情况，这样在有客户端请求来临时，namenode才能对整个HDFS系统有清晰的把控。



namenode和datanode是整个HDFS中最核心的概念，但是如果namenode节点出现故障的话，整个HDFS文件系统就会失效，所有数据都会丢失。因此需要有一种止损方案来避免这种情况的出现，这个方案就是HDFS高可用。后续小鸟将持续介绍HDFS高可用以及其他HDFS概念，欢迎关注一起学习。

# HBase（分布式数据库）

undefined

**1. HBase简介**

**1.1 什么是HBase**

HBASE是一个高可靠性、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PC Server上搭建起大规模结构化存储集群。

HBASE的目标是存储并处理大型的数据，更具体来说是仅需使用普通的硬件配置，就能够处理由成千上万的行和列所组成的大型数据。

HBASE是Google Bigtable的开源实现，但是也有很多不同之处。比如：Google Bigtable使用GFS作为其文件存储系统，HBASE利用Hadoop HDFS作为其文件存储系统；Google运行MAPREDUCE来处理Bigtable中的海量数据，HBASE同样利用Hadoop MapReduce来处理HBASE中的海量数据；Google Bigtable利用Chubby作为协同服务，HBASE利用Zookeeper作为协同服务。

**1.2 与传统数据库的对比**

1、传统数据库遇到的问题：

  1）数据量很大的时候无法存储；  
  2）没有很好的备份机制；  
  3）数据达到一定数量开始缓慢，很大的话基本无法支撑；

2、HBASE优势：

  1）线性扩展，随着数据量增多可以通过节点扩展进行支撑；  
  2）数据存储在hdfs上，备份机制健全；  
  3）通过zookeeper协调查找数据，访问速度快。

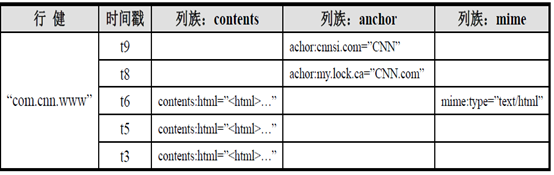
**1.3 HBase集群中的角色**

1、一个或者多个主节点，Hmaster；

2、多个从节点，HregionServer；

3、HBase依赖项，zookeeper；

**2. HBase数据模型**



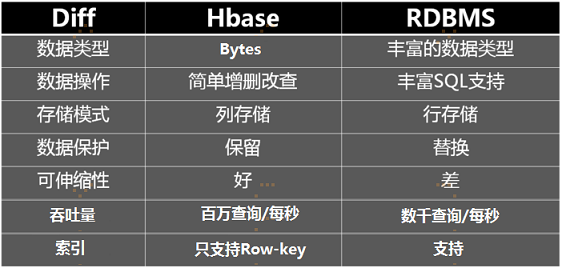
**2.1 HBase的存储机制**

HBase是一个面向列的数据库，在表中它由行排序。表模式定义只能列族，也就是键值对。一个表有多个列族以及**每一个列族可以有任意数量的列**。后续列的值连续存储在磁盘上。表中的每个单元格值都具有时间戳。总之，在一个HBase：

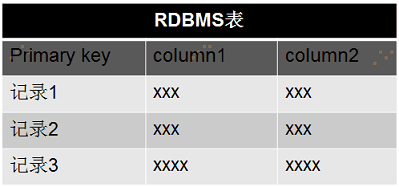
* 表是行的集合。
* 行是列族的集合。
* 列族是列的集合。
* 列是键值对的集合。

这里的列式存储或者说面向列，其实说的是列族存储，HBase是根据列族来存储数据的。列族下面可以有非常多的列，列族在创建表的时候就必须指定。

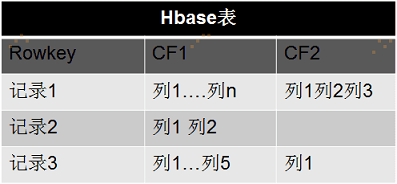
**HBase 和 RDBMS的比较**



RDBMS的表：



HBase的表：



**2.2 Row Key 行键**

与nosql数据库一样，row key是用来表示唯一一行记录的**主键**，HBase的数据时按照RowKey的**字典顺序**进行全局排序的，所有的查询都只能依赖于这一个排序维度。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（正则）

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10-1000bytes)，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)

**2.3 Columns  Family 列族**

列簇：HBASE表中的每个列，都归属于某个列族。列族是表的schema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。

**2.4 Cell**

由{row key，columnFamily，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储。

关键字：无类型、字节码

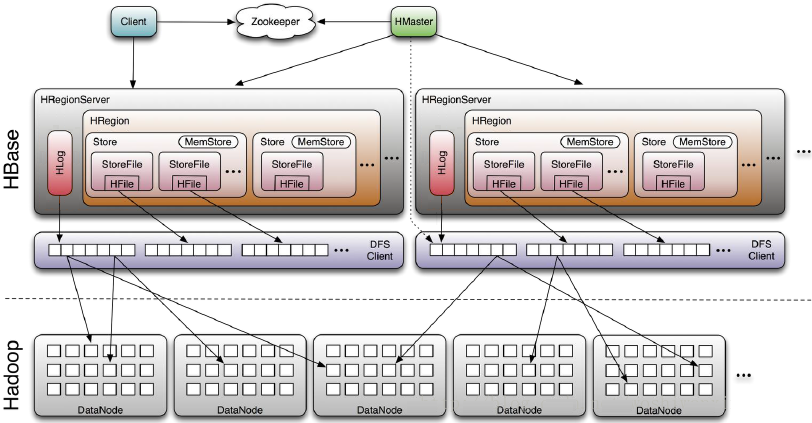
**2.5 Time Stamp 时间戳**

HBASE中通过rowkey和columns确定的为一个存储单元称为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示赋值。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管理(包括存储和索引)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，而是保存最近一段时间内的版本(比如最近7天)。用户可以针对每个列族进行设置。

**3. HBase原理**

**HBase系统架构体系图**



组成部件说明：

**Client：**

使用HBase RPC机制与HMaster和HRegionServer进行通信  
Client与HMaster进行管理类操作  
Client与HRegionServer进行数据读写类操作

**Zookeeper：**

Zookeeper Quorum存储-ROOT-表地址、HMaster地址  
HRegionServer把自己以Ephedral方式注册到Zookeeper中，HMaster随时感知各个HRegionServer的健康状况  
Zookeeper避免HMaster单点问题

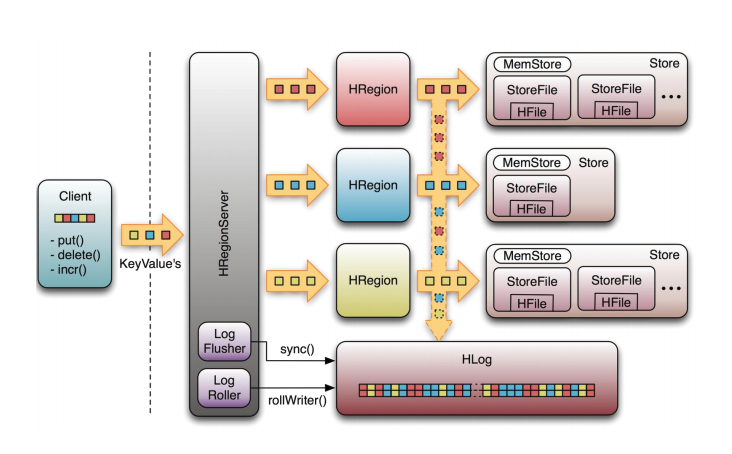
Zookeeper的主要作用：客户端首先联系ZooKeeper子集群（quorum）（一个由ZooKeeper节点组成的单独集群）查找行健。上述过程是通过ZooKeeper获取含有-ROOT-的region服务器名（主机名）来完成的。通过含有-ROOT-的region服务器可以查询到含有.META.表中对应的region服务器名，其中包含请求的行健信息。这两处的主要内容都被缓存下来了，并且都只查询一次。最终，通过查询.META服务器来获取客户端查询的行健数据所在region的服务器名。一旦知道了数据的实际位置，即region的位置，HBase会缓存这次查询的信息，同时直接联系管理实际数据的HRegionServer。所以，之后客户端可以通过缓存信息很好地定位所需的数据位置，而不用再次查找.META.表。

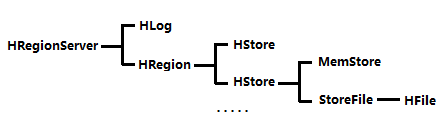
**HMaster：**

HMaster没有单点问题，HBase可以启动多个HMaster，通过Zookeeper的Master Election机制保证总有一个Master在运行  
主要负责Table和Region的管理工作：  
1. 管理用户对表的增删改查操作  
2. 管理HRegionServer的负载均衡，调整Region分布  
3. Region Split后，负责新Region的分布  
4. 在HRegionServer停机后，负责失效HRegionServer上Region迁移

**HRegionServer：**

HBase中最核心的模块，主要负责响应用户I/O请求，向HDFS文件系统中读写





HRegionServer管理一系列HRegion对象；  
每个HRegion对应Table中一个Region，HRegion由多个HStore组成；  
每个HStore对应Table中一个Column Family的存储；  
Column Family就是一个集中的存储单元，故将具有相同IO特性的Column放在一个Column Family会更高效。

可以看到，client访问hbase上的数据并不需要master参与（寻址访问zookeeper和region server，数据读写访问region server），master仅仅维护table和region的元数据信息（table的元数据信息保存在zookeeper上），负载很低。HRegionServer存取一个子表时，会创建一个HRegion对象，然后对表的每个列族创建一个Store实例，每个Store都会有一个MemStore和0个或多个StoreFile与之对应，每个StoreFile都会对应一个HFile，HFile就是实际的存储文件。因此，一个HRegion（表）有多少个列族就有多少个Store。一个HRegionServer会有多个HRegion和一个HLog。

**HRegion：**

table在行的方向上分隔为多个Region。Region是HBase中分布式存储和负载均衡的最小单元，即不同的region可以分别在不同的Region Server上，但同一个Region是不会拆分到多个server上。

Region按大小分隔，每个表一般是只有一个region。随着数据不断插入表，region不断增大，当region的某个列族达到一个阀值（默认256M）时就会分成两个新的region。

每个region由以下信息标识：

1. <表名，startRowKey，创建时间>
2. 由目录表(-ROOT-和.META.)记录该region的endRowKey

HRegion定位：Region被分配给哪个RegionServer是完全动态的，所以需要机制来定位Region具体在哪个region server。

HBase使用三层结构来定位region：

1. 通过zookeeper里的文件/hbase/rs得到-ROOT-表的位置。-ROOT-表只有一个region。
2. 通过-ROOT-表查找.META.表的第一个表中相应的region的位置。其实-ROOT-表是.META.表的第一个region；.META.表中的每一个region在-ROOT-表中都是一行记录。
3. 通过.META.表找到所要的用户表region的位置。用户表中的每个region在.META表中都是一行记录。

注意：

 -ROOT-表永远不会被分隔为多个region，保证了最多需要三次跳转，就能定位到任意的region。client会将查询的位置信息缓存起来，缓存不会主动失效，因此如果client上的缓存全部失效，则需要进行6次网络来回，才能定位到正确的region，其中三次用来发现缓存失效，另外三次用来获取位置信息。

table和region的关系

table默认最初只有一个region，随着记录数的不断增加而变大，起初的region会逐渐分裂成多个region，一个region有【startKey, endKey】表示，不同的region会被master分配给相应的regionserver管理。

region是hbase分布式存储和负载均衡的最小单元，不同的region分不到不同的regionServer。

注意：region虽然是分布式存储的最小单元，但并不是存储的最小单元。region是由一个或者多个store组成的，每个store就是一个column family。每个store又由memStore和1至多个store file 组成(memstore到一个阀值会刷新，写入到storefile，有hlog来保证数据的安全性，一个regionServer有且只有一个hlog)

**HStore：**

HBase存储的核心。由MemStore和StoreFile组成。MemStore是Stored Memory Buffer。  
**HLog：**

引入HLog原因：在分布式系统环境中，无法避免系统出错或者宕机，一旦HRegionServer意外退出，MemStore中的内存数据就会丢失，引入HLog就是防止这种情况。

工作机制：  
每个HRegionServer中都会有一个HLog对象，HLog是一个实现Write Ahead Log的类，每次用户操作写入MemStore的同时，也会写一份数据到HLog文件，HLog文件定期会滚动出新，并删除旧的文件(已持久化到StoreFile中的数据)。当HRegionServer意外终止后，HMaster会通过Zookeeper感知，HMaster首先处理遗留的HLog文件，将不同region的log数据拆分，分别放到相应region目录下，然后再将失效的region重新分配，领取到这些region的HRegionServer在Load Region的过程中，会发现有历史HLog需要处理，因此会Replay HLog中的数据到MemStore中，然后flush到StoreFiles，完成数据恢复。

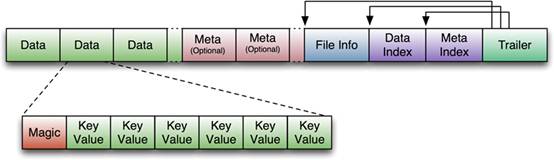
**3.1 HBase的存储格式**

HBase中的所有数据文件都存储在Hadoop HDFS文件系统上，格式主要有两种：

1. HFile，HBase中Key-Value数据的存储格式，HFile是Hadoop的二进制格式文件，实际上StoreFile就是对HFile做了轻量级包装，即StoreFile底层就是HFile。

2. HLog File，HBase中WAL(Write Ahead Log)的存储格式，物理上是Hadoop的Sequence File

**HFile**



图片解释：

HFile文件不定长，长度固定的块只有两个：Trailer和FileInfo

Trailer中指针指向其他数据块的起始点

File Info中记录了文件的一些Meta信息，例如：AVG\_KEY\_LEN, AVG\_VALUE\_LEN, LAST\_KEY, COMPARATOR, MAX\_SEQ\_ID\_KEY等

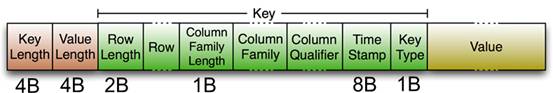
Data Index和Meta Index块记录了每个Data块和Meta块的起始点

Data Block是HBase I/O的基本单元，为了提高效率，HRegionServer中有基于LRU的Block Cache机制

每个Data块的大小可以在创建一个Table的时候通过参数指定，大号的Block有利于顺序Scan，小号Block利于随机查询

每个Data块除了开头的Magic以外就是一个个KeyValue对拼接而成, Magic内容就是一些随机数字，目的是防止数据损坏

HFile里面的每个KeyValue对就是一个简单的byte数组。这个byte数组里面包含了很多项，并且有固定的结构。



KeyLength和ValueLength：两个固定的长度，分别代表Key和Value的长度

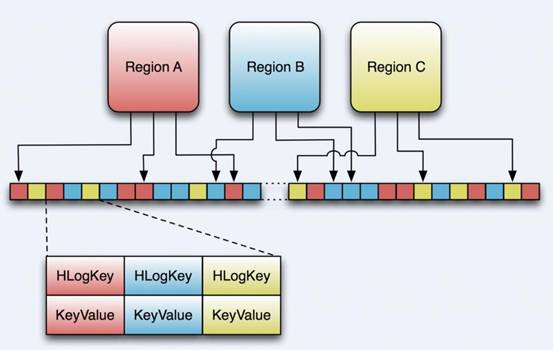
Key部分：Row Length是固定长度的数值，表示RowKey的长度，Row 就是RowKey

Column Family Length是固定长度的数值，表示Family的长度

接着就是Column Family，再接着是Qualifier，然后是两个固定长度的数值，表示Time Stamp和Key Type（Put/Delete）

Value部分没有这么复杂的结构，就是纯粹的二进制数据

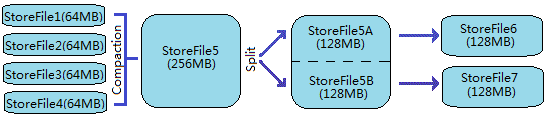
**HLog File**



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

**3.2 写流程**



1) Client通过Zookeeper的调度，向RegionServer发出写数据请求，在Region中写数据；

2) 数据被写入Region的MemStore，知道MemStore达到预设阀值(即MemStore满)；

3) MemStore中的数据被Flush成一个StoreFile；

4) 随着StoreFile文件的不断增多，当其数量增长到一定阀值后，触发Compact合并操作，将多个StoreFile合并成一个StoreFile，同时进行版本合并和数据删除；

5) StoreFiles通过不断的Compact合并操作，逐步形成越来越大的StoreFile；

6) 单个StoreFile大小超过一定阀值后，触发Split操作，把当前Region Split成2个新的Region。父Region会下线，新Split出的2个子Region会被HMaster分配到相应的RegionServer上，使得原先1个Region的压力得以分流到2个Region上。

可以看出HBase只有增添数据，所有的更新和删除操作都是在后续的Compact历程中举行的，使得用户的写操作只要进入内存就可以立刻返回，实现了HBase I/O的高性能。

**3.3 读流程**

1) Client访问Zookeeper，查找-ROOT-表，获取**.**META**.**表信息；

2) 从**.**META**.**表查找，获取存放目标数据的Region信息，从而找到对应的RegionServer；

3) 通过RegionServer获取需要查找的数据；

4) RegionServer的内存分为MemStore和BlockCache两部分，MemStore主要用于写数据，BlockCache主要用于读数据。读请求先到MemStore中查数据，查不到就到BlockCache中查，再查不到就会到StoreFile上读，并把读的结果放入BlockCache。

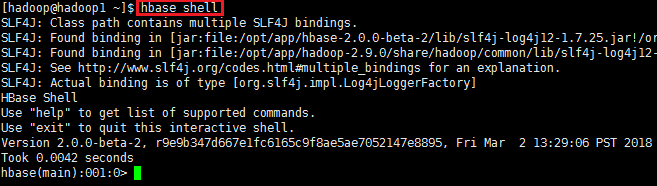
寻址过程：client—>Zookeeper—>ROOT表—>**.**META**.** 表—>RegionServer—>Region—>client

**4. HBASE命令**

**4.1 命令的进退**

1、hbase提供了一个shell的终端给用户交互

*hbase shell*



2、如果退出执行*quit*命令



**4.2 命令**

|  |  |
| --- | --- |
| 名称 | 命令表达式 |
| 查看hbase状态 | status |
| 创建表 | create '表名','列族名1','列族名2','列族名N' |
| 查看所有表 | list |
| 描述表 | describe '表名' |
| 判断表存在 | exists '表名' |
| 判断是否禁用启用表 | is\_enabled '表名' is\_disabled '表名' |
| 添加记录 | put '表名','rowkey','列族：列'，'值' |
| 查看记录rowkey下的所有数据 | get '表名','rowkey' |
| 查看所有记录 | scan '表名' |
| 查看表中的记录总数 | count '表名' |
| 获取某个列族 | get  '表名','rowkey','列族：列' |
| 获取某个列族的某个列 | get '表名','rowkey','列族：列' |
| 删除记录 | delete '表名','行名','列族：列' |
| 删除整行 | deleteall '表名','rowkey' |
| 删除一张表 | 先要屏蔽该表，才能对该表进行删除 第一步 disable '表名'，第二步 drop '表名' |
| 清空表 | truncate '表名' |
| 查看某个表某个列中所有数据 | scan '表名',{COLUMNS=>'列族名：列名'} |
| 更新记录 | 就是重新一遍，进行覆盖，hbase没有修改，都是追加 |

具体实例：

1、查看HBase运行状态  status

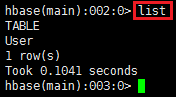


2、创建表  create <table>,{NAME => <family>, VERSIONS => <VERSIONS>}

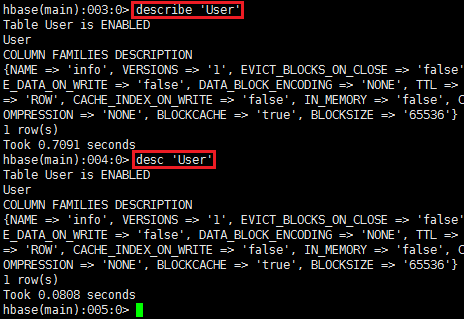
创建一个User表，并且有一个info列族



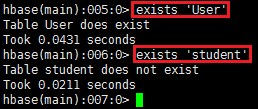
3、查看所有表  list



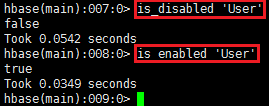
4、描述表详情  describe 'User'



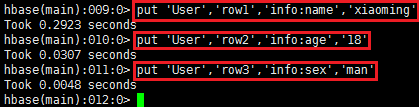
5、判断表是否存在 exists  'User'



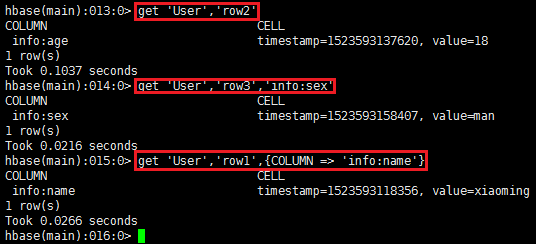
6、启用或禁用表 is\_disabled  'User'   is\_enabled  'User'



7、添加记录，即插入数据，语法：put <table>,<rowkey>,<family:column>,<value>

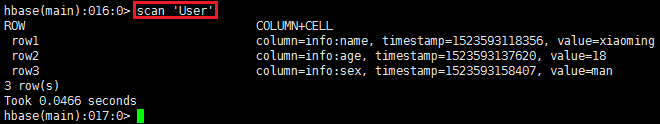


8、根据rowKey查询某个记录，语法：get <table>,<rowkey>,[<family:column>, ...]

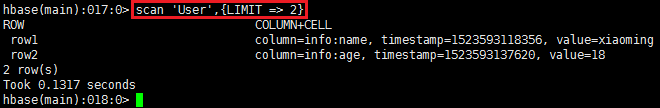


9、查询所有记录，语法：scan <table>,{COLUMNS  =>  [family:column, ...], LIMIT => num}

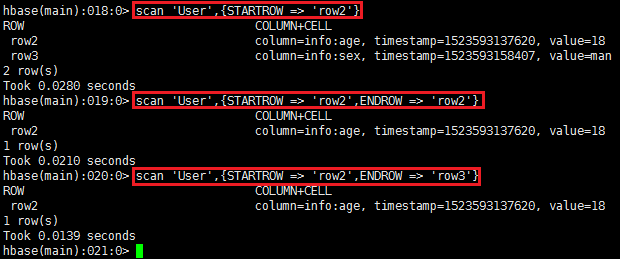
扫描所有记录



扫描前2条



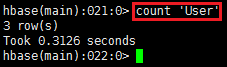
范围查询



另外，还可以添加TIMERANGE和FILTER等高级功能，STARTROW、ENDROW必须大写，否则报错，查询结果不包含等于ENDROW的结果集。

10、统计表记录数，语法：count <table>, {INTERVAL => intervalNum，CACHE => cacheNum}

 INTERVAL设置多少行显示一次及对应的rowkey，默认1000；CACHE每次去取的缓存区大小，默认是10，调整该参数可提高查询速度。



11、删除

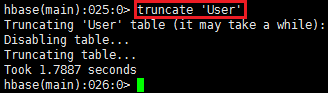
删除列



删除整行

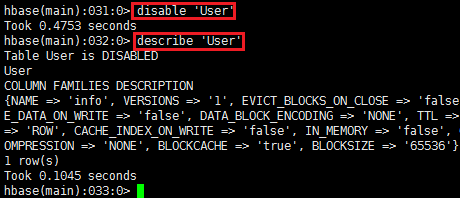


删除表中所有数据

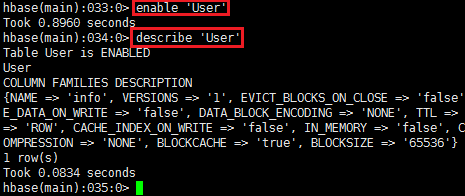


12、禁用或启用表

禁用表

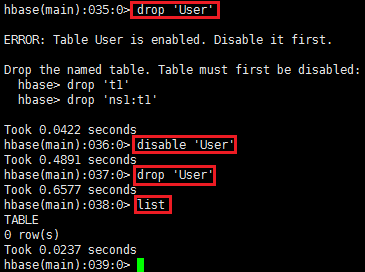


启用表



12、删除表

删除前，必须先disable



# Hadoop项目

## 倒排索引

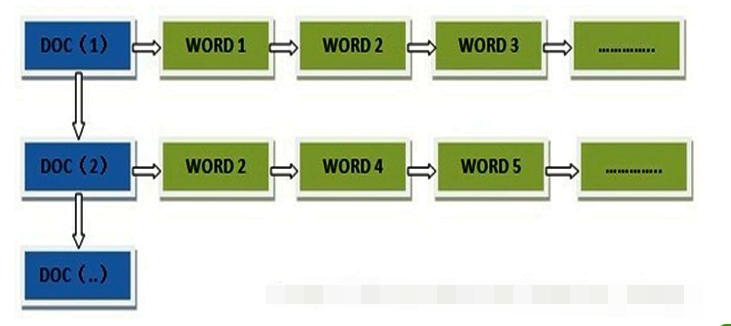
　　见其名知其意，有倒排索引，对应肯定，有正向索引。

     正向索引（forward index），反向索引（inverted index）更熟悉的名字是倒排索引。

     在**搜索引擎**中每个文件都对应一个文件ID，文件内容被表示为一系列关键词的集合（实际上在搜索引擎索引库中，关键词也已经转换为关键词ID）。例如“文档1”经过分词，提取了20个关键词，每个关键词都会记录它在文档中的出现次数和出现位置。

     得到**正向索引**的结构如下：

       “文档1”的ID > 单词1：出现次数，出现位置列表；单词2：出现次数，出现位置列表；…………。

       “文档2”的ID > 此文档出现的关键词列表。

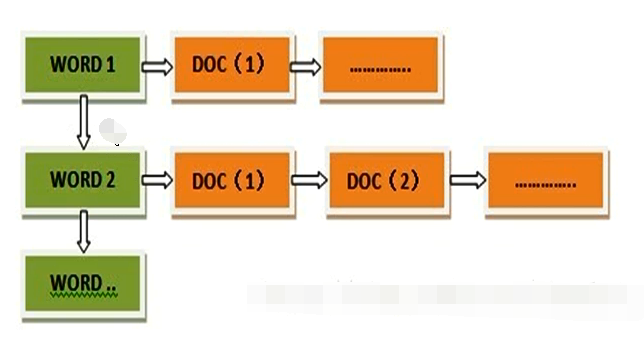
一般是通过key，去找value。

**当用户在主页上搜索关键词“华为手机”时，假设只存在正向索引（forward index），那么就需要扫描索引库中的所有文档，找出所有包含关键词“华为手机”的文档，再根据打分模型进行打分，排出名次后呈现给用户。**因为互联网上收录在搜索引擎中的文档的数目是个天文数字，**这样的索引结构根本无法满足实时返回排名结果的要求。**

**所以，搜索引擎会将正向索引重新构建为倒排索引**，即把文件ID对应到关键词的映射转换为**关键词到文件ID的映射**，每个关键词都对应着一系列的文件，这些文件中都出现这个关键词。

       得到**倒排索引**的结构如下：

       “关键词1”：“文档1”的ID，“文档2”的ID，…………。

       “关键词2”：带有此关键词的文档ID列表。

从词的关键字，去找文档。

1.单词——文档矩阵

 单词-文档矩阵是表达两者之间所具有的一种包含关系的概念模型，图1展示了其含义。图3-1的每列代表一个文档，每行代表一个单词，打对勾的位置代表包含关系。

   从纵向即文档这个维度来看，每列代表文档包含了哪些单词，比如文档1包含了词汇1和词汇4，而不包含其它单词。从横向即单词这个维度来看，每行代表了哪些文档包含了某个单词。比如对于词汇1来说，文档1和文档4中出现过单词1，而其它文档不包含词汇1。矩阵中其它的行列也可作此种解读。

搜索引擎的索引其实就是实现“单词-文档矩阵”的具体[数据结构](http://lib.csdn.net/base/datastructure)。可以有不同的方式来实现上述概念模型，比如“倒排索引”、“签名文件”、“后缀树”等方式。但是各项实验数据表明，“倒排索引”是实现单词到文档映射关系的最佳实现方式，所以本博文主要介绍“倒排索引”的技术细节。

**2.倒排索引基本概念**

       文档(Document)：一般搜索引擎的处理对象是互联网网页，而文档这个概念要更宽泛些，代表以文本形式存在的存储对象，相比网页来说，涵盖更多种形式，比如Word，PDF，html，XML等不同格式的文件都可以称之为文档。再比如一封邮件，一条短信，一条微博也可以称之为文档。在本书后续内容，很多情况下会使用文档来表征文本信息。

       文档集合(Document Collection)：由若干文档构成的集合称之为文档集合。比如海量的互联网网页或者说大量的电子邮件都是文档集合的具体例子。

       文档编号(Document ID)：在搜索引擎内部，会将文档集合内每个文档赋予一个唯一的内部编号，以此编号来作为这个文档的唯一标识，这样方便内部处理，每个文档的内部编号即称之为“文档编号”，后文有时会用DocID来便捷地代表文档编号。

       单词编号(Word ID)：与文档编号类似，搜索引擎内部以唯一的编号来表征某个单词，单词编号可以作为某个单词的唯一表征。

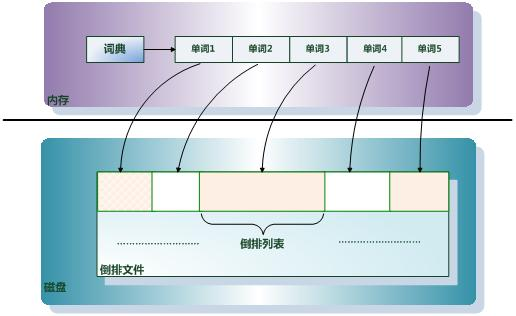
**倒排索引(Inverted Index)**：倒排索引是实现“单词-文档矩阵”的一种具体存储形式，通过倒排索引，可以根据单词快速获取包含这个单词的文档列表。倒排索引主要由两个部分组成：“单词词典”和“倒排文件”。

       单词词典(Lexicon)：搜索引擎的通常索引单位是单词，单词词典是由文档集合中出现过的所有单词构成的字符串集合，单词词典内每条索引项记载单词本身的一些信息以及指向“倒排列表”的指针。

       倒排列表(PostingList)：倒排列表记载了出现过某个单词的所有文档的文档列表及单词在该文档中出现的位置信息，每条记录称为一个倒排项(Posting)。根据倒排列表，即可获知哪些文档包含某个单词。

       倒排文件(Inverted File)：所有单词的倒排列表往往顺序地存储在磁盘的某个文件里，这个文件即被称之为倒排文件，倒排文件是存储倒排索引的物理文件。

     关于这些概念之间的关系，通过图2可以比较清晰的看出来。



## TF-IDF

项目介绍：TF-IDF

TF-IDF是一种统计方法，用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度。字词的重要性随着它在文件中出现的次数成正比增加，但同时会随着它在语料库中出现的频率成反比下降。TF-IDF加权的各种形式常被搜索引擎应用，作为文件与用户查询之间相关程度的度量或评级。除了TF-IDF以外，因特网上的搜索引擎还会使用基于链接分析的评级方法，以确定文件在搜寻结果中出现的顺序。

应用场景：

1. 权重计算方法经常会和余弦相似度(cosine similarity)一同使用于向量空间模型中，用以判断两份文件之间的相似性。

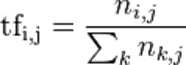
2. 广告投放，收集用户的朋友圈，或者博客，你会发现，朋友圈给你投放的广告，往往你跟发布的东西有关，例如一个人很喜欢旅游，那我们应该给他投放旅行社，或者风景秀丽的旅游景点。

3. 利用TF-IDF，对数据库的文本进行分词，建立索引后入库，提高用户全文检索的速率。

4. 等等…

接着我们看一下TF-IDF的公式，以及他的原理：

  我们首先看一下，何为TF

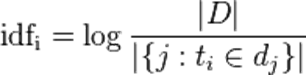
词频（term frequency，TF）指的是某一个给定的词语在该文件中出现的频率。这个数字是对词数(term count)的归一化，以防止它偏向长的文件。（同一个词语在长文件里可能会比短文件有更高的词数，而不管该词语重要与否。）对于在某一特定文件里的词语来说，它的重要性可表示为：

以上式子中分子是该词在文件中的出现次数，而分母则是在文件中所有字词的出现次数之

和。

  我们首先看一下，何为IDF

逆向文件频率（inverse document frequency，IDF）是一个词语普遍重要性的度量。某一特定词语的IDF，可以由总文件数目除以包含该词语之文件的数目，再将得到的商取对数得到。



分子：|D| 语料库中的文件总数

分母：包含词语的文件数目（即的文件数目）如果该词语不在语料库中，就会导致分母为零，因此一般情况下使用

è¿éåå¾çæè¿°

作为分母。

我们再看一下TF-IDF的公式：

è¿éåå¾çæè¿°