# 知识回顾

项目名称：xxx日志流量统计系统

技术架构：Hadoop、Hive、Flume、Kafka、Storm、Sqoop、MySQL

项目描述：xxx为电商管理平台，为提高管理决策能力，开发日志收集处理系统，通过大数据分析技术，实现对用户消费行为的准确分析。

负责模块：

略

<https://mp.weixin.qq.com/s/mmDo7_ogqxtoxwfKpe9cYg>

java技术栈

# Hbase

## 概述

HBase是一个分布式的、面向列的开源数据库，该技术来源于 Fay Chang 所撰写的Google论文“Bigtable：一个结构化数据的分布式存储系统”。就像Bigtable利用了Google文件系统（File System）所提供的分布式数据存储一样，HBase在Hadoop之上提供了类似于Bigtable的能力。HBase是Apache的Hadoop项目的子项目。HBase不同于一般的关系数据库，它是一个适合于非结构化数据存储的数据库（NoSql）。另一个不同的是HBase基于列的而不是基于行的模式。

是一种 NoSQL 非关系型的数据库，不符合关系型数据库的范式

适合存储 半结构化 非结构化 的稀疏数据

面向列(族\簇)进行存储

提供实时增删改查的能力 是一种真正的数据库

可以存储海量数据 性能也很强大 可以实现上亿条记录的毫秒级别的查询

但是不能提供严格的事务控制 只能在行级别保证事务

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技术可以在廉价的PC上搭建起大规模结构化存储集群。

HBase利用HadoopHDFS作为其文件存储系统，利用Hadoop的MapReduce来处理HBase中的海量数据，利用Zookeeper作为集群资源协调工具

## 逻辑结构

* **行键 – RowKey**

即hbase的主键，访问hbase中的数据只有三种方式

通过单一行键访问

通过一组行键访问

全表扫描

* **列族(簇) - Column Family**

是表的元数据的一部分，需要在建表时声明，不能后期增加，如果需要增加只能alter表，一个列族可以包含一个或多个列

* **列 - Column**

可以动态增加列,不需要提前声明,不是表的元数据的一部分

* **单元格与时间戳 - cell timestamp**

通过row和columns确定的一个存储单元为一个cell。每个cell都保存着一个数据的多个版本，版本通过时间戳来区别。

* **数据类型**

数据都以二进制形式存储，没有数据类型的区别。

# 安装配置

前提条件，安装jdk 和 hadoop，并配置了环境变量

## 单机模式

此模式底层文件系统为linux文件系统，并非HDFS，仅用于测试。

直接解压安装包

tar -zxvf xxxxx.tar.gz

修改conf/hbase-site.xml,配置hbase使用的数据文件的位置，默认在/tmp/hbase-[username],此目录是linux的临时目录，可能会被系统清空，所以最好修改一下

<property>

<name>hbase.rootdir</name>

<value>file:///home/app/hbase-0.98.17-hadoop2/tmp</value>

</property>

创建tmp目录

启动Hbase

./start-hbase.sh

关闭Hbase

./stop-hbase.sh

## 伪分布式模式

使用HDFS作为底层文件系统，但本身为伪分布式，不支持高可用，切性能较低。

修改conf/hbase-env.sh修改JAVA\_HOME

export JAVA\_HOME=xxxx

修改hbase-site.xml,配置使用hdfs

<property>

<name>hbase.rootdir</name>

<value>hdfs://hadoop01:9000/hbase</value>

</property>

<property>

<name>dfs.replication</name>

<value>1</value>

</property>

启动hbase

## 完全分布式模式

使用HDFS作为底层文件系统，支持高可用。

准备三台虚拟机hadoop01,hadoop02,hadoop03

其中hadoop01中需要有hadoop，jdk，zookeeper，hbase

Hadoop02：jdk，zookeeper，hbase

Hadoop03：jdk，zookeeper，hbase

### 修改conf/hbase-env.sh

修改JAVA\_HOME

export JAVA\_HOME=xxxx

禁用对zookeeper的自动管理

export HBASE\_MANAGES\_ZK false

### 修改hbase-site.xml，配置开启完全分布式模式

配置hbase.cluster.distributed为true。

配置hbase.rootdir设置为HDFS访问地址

<property>

<name>hbase.rootdir</name>

<value>hdfs://hadoop01:9000/hbase</value>

</property>

<property>

<name>dfs.replication</name>

<value>1</value>

</property>

<property>

<name>hbase.cluster.distributed</name>

<value>true</value>

</property>

<property>

<name>hbase.zookeeper.quorum</name>

<value>hadoop01:2181,hadoop02:2181,hadoop03:2181</value>

</property>

### 配置region服务器

修改conf/regionservers文件，其中配置所有hbase主机，每个主机名独占一行，hbase启动或关闭时会按照该配置顺序启动或关闭主机中的hbase

hadoop01

hadoop02

hadoop03

### HBASE配置文件说明

hbase-env.sh配置HBase启动时需要的相关环境变量

hbase-site.xml配置HBase基本配置信息

HBASE启动时默认使用hbase-default.xml中的配置，如果需要可以修改hbase-site.xml文件，此文件中的配置将会覆盖hbase-default.xml中的配置

修改配置后要重启hbase才会起作用

### 启动集群

启动zookeeper

启动hadoop

启动hbase

访问http://xxxxx:60010来访问web界面，通过web见面管理hbase

也可以通过hbase shell脚本来访问bhase

### 启动备用master实现高可用

./hbase-daemon.sh start master

### 关闭集群

stop-hbase.sh

# 基本操作

启动hbase

bin/start-hbase.sh

开启hbase命令行工具

bin/hbase shell

查看状态

status

帮助文档

help

创建表

create 'testtable',''colfam1','colfam2'

查看表

list

查看表结构

desc 'testtable'

插入数据（修改操作与之相同）

put 'testtable','myrow-1','colfam1:q1','value-1'

全表扫描

scan 'testtable'

查询数据

get 'testtable','myrow-1'

删除数据

delete 'testtable','myrow-2','colfam1:q2'

禁用表

disable 'testtable'

删除表

drop 'testtable'

#建表时可以指定VERSIONS，配置的是当前列族在持久化到文件系统中时，要保留几个最新的版本数据，这并不影响内存中的历史数据版本

hbase>create 'testtable',{NAME=>'colfam1',VERSIONS=>3},{NAME=>'colfam2',VERSIONS=>1}

hbase>put 'testtable','myrow-1','colfam1:q1','value-1'

#直接使用scan而不加RAW=>true只能查询到最新版本的数据

hbase>scan 'testtable'

hbase>put 'testtable','myrow-1','colfam1:q1','value-2'

hbase>scan 'testtable'

hbase>put 'testtable','myrow-1','colfam1:q1','value-3'

hbase>scan 'testtable'

#可以在查询时加上RAW=>true来开启对历史版本数据的查询，VERSIONS=>3指定查询最新的几个版本的数据

hbase>scan 'testtable',{RAW=>true,VERSIONS=>3}

hbase>put 'testtable','myrow-1','colfam1:q1','value-4'

hbase>scan 'testtable'

hbase>scan 'testtable',{RAW=>true,VERSIONS=>3}

hbase>put 'testtable','myrow-1','colfam2:x1','value-1'

hbase>scan 'testtable'

hbase>put 'testtable','myrow-1','colfam2:x1','value-2'

hbase>scan 'testtable'

hbase>scan 'testtable',{RAW=>true,VERSIONS=>3}

#重启hbase

hbase>scan 'testtable',{RAW=>true,VERSIONS=>3}

hbase>exit

bin/stop-hbase.sh

hbase命令行下不能使用删除：

可以使用 ctrl+删除键 来进行删除

或

修改xshell配置：

文件->属性->终端->键盘

->delete键序列[VT220Del]

->backspace键序列[ASCII127]

# HBASE原理 参看Hbase.docx

## 总结

hbase表中的数据按照行键的字典顺序排序

hbase表中的数据按照行的的方向切分为多个region

最开始只有一个region 随着数据量的增加 产生分裂 这个过程不停的进行 一个表可能对应一个或多个region

region是hbase表分布式存储和负载均衡的基本单元 一个表的多个region可能分布在多台HRegionServer上

region是分布式存储的基本单元 但不是存储的基本单元 内部还具有结构。

一个region由多个Store来组成

有几个store取决于表的列族的数量 一个列族对应一个store 之所以这么设计 是因为 一个列族中的数据往往数据很类似 方便与进行压缩 节省存储空间

表的一个列族对应一个store store的数量由表中列族的数量来决定

一个store由一个memstore 和零个或多个storefile组成

storefile其实就是hdfs中的hfile 只能写入不能修改

数据写入hdfs时 先在hlog中记录日志 再修改memstore 直接返回成功 这样 不需要真正等待写入hdfs的过程 所以很快

memstore 内存有限 当写入数量达到一定的阈值的时候 就会创建一个新的memstore继续工作 而旧的memstore 会用一个单独的线程 写出到storefile中 最终清空旧的memstore 并在zookeeper中记录最后写出数据时间的redo point信息

由于storefile 不能修改 所以数据的更新其实是不停创建新的storefile的过程

这样多个storefile中可能存在对同一个数据的多个版本 其中旧的版本其实是垃圾数据 时间一长 垃圾数据就可能很多 浪费磁盘空间

所以当达到一定的阈值的时候 会自动合并storefile 在合并的过程中将垃圾数据清理

而当合并出来的文件达到一定程度时 再从新进行切分 防止文件过大

虽然看起来是小变大再变小 但是经过这个过程垃圾数据就被清理掉了

所以store中的数据 其实是memstore和storefile来组成的

而memstore由于是内存中的数据 一旦断电就会丢失

为了解决可能的意外造成数据丢失的问题 hbase在整个hregionserver中 通过记录hlog 来保存了所有数据操作的记录

当hbase启动时 会检查zookeeper中的redopoint信息 从hlog中恢复 这个时间点之后的数据 解决数据容易丢失的问题

hlog整个hregionServer中只有一个 所有这台机器中的所有HRegion都公用这个文件 这样整个机器的磁盘性能都可以为这一个文件提供支持 提升文件的读写效率

hlog文件最终对应的是hdfs中的文件 也是分布式存储的 保证了日志文件的可靠性

而在数据读取时 会将store的中memstore和storefile中的数据进行合并 提供查询

此处所谓的合并 并不是真正的数据的合并 而是将数据的索引进行合并

由于hbase中的数据天然排序 再加上索引 整个查询也可以非常的快

## 集群结构信息

hbase中的老大叫hmaster 小弟叫hregionServer

客户端叫Client

Zookeepr为hbase提供集群协调

client

访问hbase 保留一些缓存信息提升效率

zookeeper

保证任何时候集群只有一个HMaster

监控regionServer的状态 将其上线下线信息通知mater

存储所有Region的寻址地址

存储hbase的元数据信息 包括 有哪些表 有哪些列族等等

Mater

为RegionServer分配Region

为RegionServer进行负载的均衡

GFS上的垃圾回收

处理对Schema数据的更新请求

RegionServer

维护Master分配给它的region，处理对这些region的IO请求

负责切分在运行过程中变得过大的region

## 问题

**为什么hbase可以很快：**

从逻辑结构上来说：

表按照行键进行了排序，所以查询时可以很快定位

从物理结构上来说：

数据按照行键切分为多个HRegion，分布在多个RegionServer中，查询大量数据时，多个RegionServer可以一起工作，从而提高速度

**为什么hbase可以存储很多数据：**

基于hdfs，所以支持可扩展性，可以通过增加大量的廉价的硬件提高存储容量

空的数据不占用空间，当存储稀疏数据时，不会浪费空间

按列族存储，同一列族的数据存放在一起，而同一列族的数据一般都是同样的类型的内容相似的数据，可以实现非常高效的压缩，节省空间

**为什么hbase的数据是可靠的：**

基于hdfs，由hdfs的可靠性保证了hbase的可靠性--即数据可以有多个备份

利用zookeeper实现了HA，即使某一台机器挂掉另外的机器也可以很快的替换它

**hbase和传统的关系型数据库有何异同：**

比起传统的关系型数据库，可以存储半结构化非结构化的数据，可以存储和处理更大级别的数据，提供高效的查询，对于稀疏数据的处理更好，具有更好的横向扩展性，免费开源性价比很高。但是不能支持非常好的事务特性，只支持行级的事务。只能通过行键来查询，表设计时难度更高。而mysql用来存储结构化的数据提供更好的事务控制。

**Hbase和Hive有何异同**

比起hive，hive只是在mapreduce上包了一层壳，本质上还是离线数据的处理的工具，实时查询性能有限，本质上是一个基于hadoop的数据仓库工具，不能支持行级别的新增修改和删除。hbase可以提供实时的数据的处理能力，适用于在线数据查询处理，本质上是一种数据库工具。

# HBase的表设计

HBase是用来存放半结构化 非结构化数据的数据库。

对HBase表的设计 会直接影响hbase使用的效率 和 使用的便利性

对HBase表的设计 主要是 列族的设计 和 行键的设计

随机数+时间戳+内容信息（taobao、baidu、wangyi）

123124124\_1231233\_taobao

123123123\_1233234\_baidu

1.列族的设计

在设计hbase表时候，列族不宜过多，尽量的要少使用列族，官方推荐hbase表的列族不宜超过3个。

经常要在一起查询的数据最好放在一个列族中，尽量的减少跨列族的数据访问。

如果有多个列族 多个列族中的数据应该设计的比较均匀

2.行键的设计

hbase表中行键是唯一标识一个表的字段，所以行键设计的好不好将会直接影响未来对hbase的查询的性能和查询的便利性

所以hbase中的行键是需要进行精心设计的

行键设计的基本原则：

行键必须唯一

必须唯一才能唯一标识数据

行键必须有意义

这样才能方便数据的查询

行键最好是字符串类型

因为数值类型在不同的系统中处理的方式可能不同

行键最好具有固定的长度

不同长度的数据可能会造成自然排序时排序的结果和预期不一致

行键不宜过长

行键最多可以达到64KB,但是最好是在10~100字节之间，最好不要超过16字节，越短越好，最好是8字节的整数倍。

行键的最佳实践：

散列原则：

行键的设计将会影响数据在hbase表中的排序方式，这会影响region切分后的结果，要注意，在设计行键时应该让经常要查询的数据分散在不同的region中，防止某一个或某几个regionserver成为热点。

有序原则：

行键的设计将会影响数据在hbase表中的排序方式，所以一种策略是将经常连续查询的条件作为行键最前面的数据，这样一来可以方便批量查询

# java api操作

导入开发包

将hbase安装包中lib下包导入java项目

创建表

**public** **void** create() **throws** Exception{

//1.创建配置对象

Configuration conf = **new** Configuration();

conf.set("hbase.zookeeper.quorum", "hadoop01:2181,hadoop02:2181,hadoop03:2181");

//2.创建HbaseAdmin对象

HBaseAdmin admin = **new** HBaseAdmin(conf);

//3.创建表的描述器

//--创建表名对象

TableName name = TableName.*valueOf*("tabx1");

//--创建描述器

HTableDescriptor tabDesc = **new** HTableDescriptor(name );

//4.添加列族信息

//--创建列族描述器

HColumnDescriptor cf1 = **new** HColumnDescriptor("cf1".getBytes());

tabDesc.addFamily(cf1 );

//5.创建表

admin.createTable(tabDesc );

//6.关闭连接

admin.close();

}

插入数据

**public** **void** put() **throws** Exception{

//1.创建配置对象并指定zookeeper地址

Configuration conf = **new** Configuration();

conf.set("hbase.zookeeper.quorum", "hadoop01:2181,hadoop02:2181,hadoop03:2181");

//2.创建Htable对象

HTable table = **new** HTable(conf, "tabx".getBytes());

//3.创建put对象并指定行键

Put put = **new** Put("rk1".getBytes());

//--添加数据

put.add("cf1".getBytes(), "a1".getBytes(), "11111".getBytes());

put.add("cf1".getBytes(), "a2".getBytes(), "22222".getBytes());

//4.插入数据

table.put(put);

//5.关闭连接

table.close();

}

获取数据

**public** **void** get() **throws** Exception{

//1.创建配置对象并指定zookeeper地址

Configuration conf = **new** Configuration();

conf.set("hbase.zookeeper.quorum", "hadoop01:2181,hadoop02:2181,hadoop03:2181");

//2.创建Htable对象

HTable table = **new** HTable(conf, "tabx".getBytes());

//3.创建get对象并指定行键

Get get = **new** Get("rk1".getBytes());

//4.创建result对象接受结果并处理

Result result = table.get(get);

**byte**[] value = result.getValue("cf1".getBytes(), "a1".getBytes());

String rs = **new** String(value);

System.***out***.println(rs);

//5.关闭连接

table.close();

}

获取数据集

**public** **void** scan() **throws** Exception{

//1.创建配置对象并指定zookeeper地址

Configuration conf = **new** Configuration();

conf.set("hbase.zookeeper.quorum", "hadoop01:2181,hadoop02:2181,hadoop03:2181");

//2.创建Htable对象

HTable table = **new** HTable(conf, "tabx".getBytes());

//3.创建Scan对象

Scan scan = **new** Scan();

//4.创建ResultScanner对象接收结果集，并处理

ResultScanner scanner = table.getScanner(scan);

Iterator<Result> it = scanner.iterator();

**while**(it.hasNext()){

Result next = it.next();

**byte**[] value = next.getValue("cf1".getBytes(), "a1".getBytes());

String str = **new** String(value);

System.***out***.println(str);

}

//5.关闭连接

table.close();

}

删除数据

**public** **void** delete() **throws** Exception{

//1.创建配置对象并指定zookeeper地址

Configuration conf = **new** Configuration();

conf.set("hbase.zookeeper.quorum", "hadoop01:2181,hadoop02:2181,hadoop03:2181");

//2.创建Htable对象

HTable table = **new** HTable(conf, "tabx".getBytes());

//3.创建delete对象并且制定删除目标

Delete delete = **new** Delete("rk1".getBytes());

delete.deleteColumn("cf1".getBytes(), "a1".getBytes());

//4.删除数据

table.delete(delete);

//5.关闭连接

table.close();

}

删除表

**public** **void** drop() **throws** Exception{

//1.创建配置对象并指定zookeeper地址

Configuration conf = **new** Configuration();

conf.set("hbase.zookeeper.quorum", "hadoop01:2181,hadoop02:2181,hadoop03:2181");

//2.创建HbaseAdmin对象

HBaseAdmin admin = **new** HBaseAdmin(conf);

//3.禁用表

admin.disableTable("tabx".getBytes());

//4.删除表

admin.deleteTable("tabx".getBytes());

//5.关闭连接

admin.close();

}