第一章

**大数据的四个特征：**

数量大、数据类型多、处理速度快、价值密度低。

**大数据的四个关键技术：**

数据采集与预处理、数据存储和管理、数据处理和分析、数据安全与隐私保护

第二章

**Hadoop的七个特征：**

高可靠性、高容错性、高效性、高可扩展性、成本低、运行在Linux系统下、支持多种编程语言

**Hadoop的十大生态环境及其功能：**

MapReudce:是一种编程模型，用于大数据集的并行运算，将复杂的、运行于大规模集群上的并行运算过程高度抽象到两个函数map、reduce，并允许用户在完全不了解分布式系统底层的细节情况下开发并行应用程序，运行于廉价计算机集群上，完成海量数据计算。

HDFS：处理超大数据集、流式处理读取速度快、支持运行在廉价商用服务器上、高容错高可扩展。

Hbase：提供高可靠性、高性能、可伸缩、实时读写、分布式的列式数据库。

Zookeeper：提供高效和可靠的协同工作系统，提供分布式锁之类的基本服务，用于构建分布式应用，减轻分布式应用程序所承担的协调任务。

Hive：hadoop的数据仓库工具，用于对hadoop文件中的数据集进行数据整理、特殊查询和分析存储。

Flume：高可用的、高可靠的、分布式的海量日子采集、聚合和传输的系统。

Mahout：数据挖掘库

Sqoop：用于hadoop和关系数据库之间交换数据

Pig：提供一种更接近结构化查询语言的接口

Ambari：支持apache hadoop集群的安装、部署、配置和管理。

**Hadoop伪分布式启动后所具有的进程：**

NodeManager、ResourceManager、NameNode、DataNode、SecondaryNameNode

第三章

分布式文件系统以“块”为基本读写单位，其中**hadoop1.0**版本定义一块为64MB，**hadoop2.0**版本定义一块为128MB。

分布式文件系统在物理结构上是由计算机集群中的多个节点构成的，其中结点分为两类：主节点/名称节点、从节点/数据结点；

名称节点的功能：负责管理分布式文件系统的命名空间，记录分布式文件系统中的每个文件中各个块所在的数据节点的信息位置。(保存两个核心的数据结构：FsImage、EditLog。FsImage用于维护文件系统树以及文件系统中所有的文件和文件夹的元数据；EditLog记录了所有针对文件执行的操作；记录了每个文件块所在的数据结点的位置信息，并不持久化保存，而是每次启动时扫描数据结点重构这些信息。)

数据结点的功能：负责数据的存储和读取，会根据客户端或者名称节点的调度来进行数据的存储和检索，并向名称节点定时发送存储的块的列表。

第二名称节点的功能：(1)完成EditLog和FsImage的合并操作（EditLog随着操作越来越多，文件越来越大使得每次重启重载的信息太多、花销太大、影响用户的使用）；(2)作为名称节点的“检查点”，周期性的备份名称节点中记录的元数据信息，当名称节点故障时可用第二名称节点中的元数据进行数据恢复，但还是有部分元数据不能恢复。

HDFS采用主从结构模型，一个HDFS集群中包含唯一一个名称节点和多个数据节点。

用户访问文件的过程：

用户访问一个文件时，首先将文件名发送到名称节点，名称节点根据文件名找到对应的数据块，在根据数据块信息找到实际存储各个数据块的数据节点的位置，并把数据节点的位置发送到客户端，最后客户端直接访问数据节点获取信息。名称节点不参与数据的传输，一个文件的数据能够在不同的数据结点上实现并发访问，大大提高了数据的访问速度。

Hadoop1.0版本只有一个命名空间、名称节点，未实现磁盘配额、文件访问权限、文件的软连接、硬连接

HDFS体系结构的局限性：

命名空间的限制：命名空间保存在内存中，受内存大小的限制

性能的瓶颈：整个分布式系统的吞吐量受单个名称节点的吞吐量的限制

隔离问题：一个命名空间、一个名称节点，无法对不同应用程序进行隔离

集群可用性问题：唯一的名称节点发生故障导致整个集群不能使用

**HDFS的存储原理：**

**冗余存储的**优点：(1)加快数据传输速度 (2)容易检查数据错误 (3)保证数据的可靠性

数据存取策略（数据存放、数据读取、数据复制）

数据存放（HDFS默认存储因子为3，每一个文件会被同时保存到3个地方）：集群内发起写操作，第一个副本放置在发起写操作的数据节点上；集群外部的写操作请求从集群内部选择磁盘不太满、CPU不太忙的数据节点作为第一个副本的存放地。第二个副本放在与副本一不同的机架的数据节点上，第三副本放在第一副本相同机架不同的数据节点上，之后的副本随机存放。

数据读取：若客户端机架ID与所要访问的数据节点的某一副本机架ID相同，则优先选择该数据节点获取信息，若无，随机访问任一节点。

数据复制：流水线复制，客户端向HDFS中写入一个文件时，文件首先写入本地，被切分成多个块，每个块都向名称节点发起请求，名称节点根据数据节点的使用情况分配一个数据节点列表返回给客户端，客户端将数据首先写入列表中的第一个数据节点，同时把列表传给第一个数据节点，当数据节点接收到数据时写入本地，并且向第二个数据节点发送请求连接，把自己接收到的数据和列表传给第二个数据节点，以此类推，直到最后数据全部写完，复制也同时完成。

**HDFS的读取过程**

**读取文件：**

(1)客户端通过FileSystem.open()打开文件，相应的在HDFS文件系统中DistributedFileSystem具体实现了FileSystem。因此，调用open()方法后，DistributedFileSystem会创建输入流FSDataInputStream，对HDFS而言，具体的输入流就是DFSInputStream。

(2)在DFSInputStream的构造函数中，输入流通过ClientProtocal.getBlockLocations()远程调动名称节点，获得文件开始部分数据块的保存位置，对于该数据块，名称节点返回保存该数据块的所有数据节点的地址，同时根据距离客户端的远近对数据节点进行排序；然后，DistributedFileSystem会利用DFSInputStream来实例化FSDataInputStream，返回给客户端，同时返回了数据块的数据节点地址。

(3)获得输入流FSDataInputStream后，客户端调用read()函数开始读取数据，输入流根据前面的排序结果，选择距离客户端最近的数据节点建立连接并读取数据。

(4)数据从该数据节点读到客户端，当该数据块读取完毕时，FSDataInputStream关闭和该数据节点的连接。

(5)输入流通过getBlockLocation()方法查找下一个数据块

(6)找到数据块的最佳读取节点，读取数据。

(7)当客户端读取完毕数据的时候，调用FSDataInputStream的close()函数，关闭输入流。

**写数据：**

(1)客户端通过FileSystem.create()创建文件，相应的，在HDFS文件系统中，DistributedFileSystem实现FileSystem。因此，调用create()方法后，DistributedFileSystem会创建输入流FSDataOutputStream，对于HDFS而言，具体的输入流就是DFSOutputStream。

(2) DistributedFileSystem通过RPC远程调用名称节点，在文件系统的命名空间中创建一个新文件（名称节点会检查文件是否已存在、用户是否有权限等），检查通过后名称节点创建文件并添加文件信息。远程调用结束后，DistributedFileSystem会利用DFSOutputStream实例化FSDataOutputStream，返回给客户端，客户端使用此输出流输入数据。

(3)获得输出流后，客户端调用输出流的write()方法向HDFS中的文件写入数据。

(4)客户端向输出流FSDataOutputStram中写入的数据会首先被分成一个个分包，这些包被放入DFSOutputStream对象的内部队列。输出流FSDataOutputStram会向名称节点申请保存文件和副本数据块的若干数据节点，这些节点形成一个数据流管道，列队中的分包被打包成数据包发往管道的第一个数据节点，第一个数据节点将数据包发往第二个数据节点…

(5)为了保证每个数据节点的数据都是准确的，接收到数据的数据节点要向发送者发送“确认包”，确认包沿着数据流管道逆流而上，回到客户端，客户端收到时对应的分包从内部队列移出，不断执行3-5步，直到数据全部写完

(6)客户端调用close()方法关闭输入流；当DFSOutputStream对象内部队列中的分包都收到答应以后，就可以使用ClientProtocl.complete()方法通知名称节点关闭文件，完成一次正常的写文件过程。

**使用HDFS的好处：**

兼容廉价的硬件设备、流数据读写（数据吞吐量高）、能够处理大数据集、简单的文件模型（一次写入、多次读取）、强大的跨平台兼容性。

**不适用HDFS的好处：（HDFS的缺陷）**

不适合低延迟数据访问、无法高效储存大量小文件、不支持多用户写入及修改文件任意（不支持多用户同时执行写操作，hadoop1.0不支持文件的修改，hadoop2.0仅支持文件的追加）

第四章

Hbase是一高可靠性、高性能、面向列、可伸缩的分布式数据库，主要用来存储非结构化和半结构化的松散矩阵

Hbase和传统的关系数据库的比较（相对比于关系数据库，Hbase不支持事务，因此无法实现跨行的原子性）：

|  |  |  |
| --- | --- | --- |
|  | Hbase | 传统关系数据库 |
| 数据类型 | 未经解释的字符串 | 丰富的数据类型和储存方式 |
| 数据操作 | 简单的插入、查询、删除、清空等，只采用单表的主键查询，无表之前的连接操作 | 除了插入、删除、更新、查询等，表与表之间可建立复杂的连接操作 |
| 存储模式 | 基于列储存 | 基于行存储 |
| 数据索引 | 只有一个索引——行键 | 针对不同的列建立复杂的索引 |
| 数据维护 | 更新操作保留原始版本 | 更新操作直接替换原始值，无保留 |
| 可伸缩性 | 灵活的水平扩展 | 难以实现横向扩展，纵向扩展的空间有限 |

Hbase是一个稀疏、多维度、排序的映射表，这张表的索引是行键、列族、列限定符、时间戳，每个值都是未经解释的字符串，没有数据类型。**列族支持动态扩展**，每一行数据都可以有截然不同的列，所以对于整个映射表的每行数据而言，有些列是空的，所以说Hbase是**稀疏矩阵**。Hbase执行更新操作时，不删除旧的数据版本，通过时间戳来保存。列名都以列族为前缀，列族是基本的访问控制单元；列族里的数据都通过列限定符来确定，没有数据类型，总被视为**字节数组byte[];**在Hbase表中，通过行、列族、列限定符确定一个单元格，单元格中的数据没有数据类型，总被**视为byte[]**，每个单元格可以保存一个数据的多个版本，每个版本对应一个**时间戳**；每次对单元格进行更新操作都会隐式地自动生成并储存一个时间戳作为旧版本的索引；四维坐标[行键，列族，列限定符，时间戳]确定一个值。

概念视图层面，每个表是由多行组成的，物理视图层面，每个表时由多列组成的。

行式数据库：适用于小批量的数据处理，如**联机事务型数据处理**。

列式数据库：适用于批量数据处理和即席查询，主要用于数据挖掘、**决策**支持和地理信息系统等查询**密集型**系统中。优点是：降低I/O开销、支持大量并发用户查询、处理速度快、比行式数据库更加有效；缺点是：执行连接操作时需要昂贵的元祖重构代价。

**Hbase的功能组件：**

库函数：链接到每个客户端；

一个Master主服务器(管理和维护Hbase表的分区信息、维护Region服务器列表、重新分配故障Region服务器中的Region、确保Region服务器间的负载均衡、处理模式变化)

多个Region服务器：管理和维护分配给自己的Region，处理来自客户端的读写请求

**分区机制：**

Hbase采用分区存储，一个大的表会被拆分成许多个Region，这些Region会被分发到布桶的服务器上实现分布式存储

**Region定位：**

客户端访问用户数据前，可以先访问缓存是否有该Region的位置信息，若没有需要首先访问Zookeeper，获取-ROOT-表的位置信息，然后访问-ROOT-表，获得.METE.表的信息，接着访问.METE.表，找到需要的Region具体位于哪个Region服务器，最后到Region服务器上读取数据，并将Region的位置信息存储缓存中。

Zookeeper服务器：由多台机器构成的集群来提供稳定可靠的协同服务

功能：(1)Region服务器都需要到Zookeeper处注册使得Zookeeper实时监控Region服务器的状态并反馈到Master；

(2)为Hbase选择唯一一个Master作为集群的总管。

(3)保存-ROOT-表（实现三级寻址）和Master位置信息，保存模式。

Master的功能：

(1)管理用户对表的增加、删除、修改、查询等操作

(2)实现不同Region服务器之间的负载均衡

(3)Region分裂或合并后重新发布Region

(4)对故障Region服务器上的Region进行迁移

Region服务器的功能：维护分配给自己的Region，响应客户端的读写操作。Hbase采用hdfs作为底层存储，所以自身不具备数据复制和维护数据副本的功能。

Region服务器（一个Hlog文件和多个Region对象，每个Region对象由多个Store（储存一个列族）组成，每个Store中有一个MemStore和多个StoreFile（在HDFS中以Hffile储存））的工作原理：

**读取数据的原理**：Region服务器会首先访问缓存MemStore，若数据不在缓存，再到磁盘上的StoreFile中找

**写入数据的原理**：客户端会被分配到相应的Region服务器去执行操作。用户数据先写入HLog和MemStore，当数据写入Hlog后，commit()调用返回到客户端。

缓存刷新：

MemStore的容量有限，系统会周期性的将其中的内容写入一个新的StoreFile中，清空缓存，并在Hlog中写入标记表示缓存中的内容已写入StoreFile文件中，所以每个Store有多个StoreFile。每次重启时，Region服务器会检查Hlog文件，最近一个缓存刷新后是否执行数据更新操作，如有新操作，则把更新写入MemStore中并执行缓存刷新写入磁盘中，最后删除旧的Hlog。

StoreFile的合并

每次缓存刷新都会生成新的StoreFile文件，使得其数量较多耗费查询时间，所以当其数量达到阈值时触发合并操作，将多个文件合并成一个大文件

**Store的工作原理**

Store对应了表中的一个列族的储存，每个Store包括一个MemStore和多个StoreFile。当用户写入数据时，系统首先把数据放入MemStore缓存，当缓存满时就会刷新到磁盘中的一个FileStore文件，当FileStore文件数量达到预先设定时触发合并操作，多个StoreFile合并成一个大的StoreFile，当单个StoreFile文件的大小超过一定的阈值时，触发文件分裂操作，同时当前的一个Region分类成两个Region，并被Master分配到相应的Region服务器上。

**Hlog的工作原理**(Hbase采用Hlog（预写式日志）来保证系统发生故障时能够恢复到正确的状态)

用户更新数据首先被记入日志后才能写入MemStore缓存，并直到MemStore缓存内容对应的日志已经被写入磁盘之后，缓存内容才会被刷新写入磁盘。

当Region服务器发生故障时，Master首先处理Hlog，一个Hlog记录了来自多个Region对象的日志文件，系统根据每条日志属于的Region对象对Hlog数据进行拆分，分别放到相应的Region对象的目录下，并为Region重新分配Region服务器，Region服务器领取分配到的Region及其相关的日志记录以后，重新做一遍日记记录中的各种操作，把日志记录中的数据写入MemStore缓存，然后刷新缓存，完成数据的恢复。

一个Region服务器仅一个Hlog文件的好处：不需要同时打开、写入多个文件，减少磁盘寻址次数；坏处：故障时，需要对日志文件进行拆分。

第五章

NoSQL的三个特点：

灵活的可扩展性、灵活的数据类型、与云计算紧密结合

Web2.0的要求，关系数据库无法满足：

海量数据的管理需求、数据高并发的需求、高可扩展性和高可用性

Web2.0不要求：

严格的数据库事务、严格的读写实时性、大量复杂的SQL查询

|  |  |  |
| --- | --- | --- |
| 比较标准 | 关系数据库 | NoSQL |
| 数据规模 | 大 | 很大 |
| 数据库原理 | 支持 | 部分支持 |
| 数据库模式 | 固定 | 灵活 |
| 查询效率 | 快 | 实现简单查询，不具备高度结构化查询 |
| 一致性 | 强一致性 | 弱一致性 |
| 数据完整性 | 容易实现 | 很难实现 |
| 扩展性 | 一般 | 较好 |
| 可用性 | 好 | 很好 |
| 标准化 | 是 | 否 |
| 技术支持 | 高 | 低 |
| 可维护性 | 复杂 | 复杂 |

NoSQL的四大类型：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型 | 数据模型 | 优点 | 缺点 | 适用场合 |
| 键值数据库 | 键值对 | 扩展性好、灵活性好、大量写操作时性能高 | 无法存储结构化信息、条件查询效率低 | 内容缓存，如会话、配置文件、参数、购物车等 |
| 列族数据库 | 列族 | 查找快、扩展强、易进行分布式扩展、复杂性低 | 功能较少、大都不支持强事务一致性 | 分布式数据存储于管理 |
| 文件数据库 | 版本化的文档 | 性能好、灵活性高、复杂性低、数据结构灵活 | 缺乏统一的查询语言 | 存储、索引并管理面向文档的数据或类似半结构化数据 |
| 图数据库 | 图结构 | 灵活性高、适合复杂的图算法、用于构建复杂的关系图谱 | 复杂性高、只能支持一定的数据规模 | 大量复杂的图算法、互连接、低结构化的图结构场合，如社交网络、推荐系统等 |

NoSQL的三大基石：CAP、BASE、最终一致性

(C)一致性:任何一个读操作总是能够督导之前完成写操作的结果。

(A)可用性:快速获取数据，可以在确定的时间内返回操作结果

(P)分区容忍性:系统中的一部分节点无法与其他节点通信，分离的系统也能够正常运行

BASE:基本可用、软状态、最终一致性

第七章

**Mapreduce与Hadoop的关系：**

Hadoop是一个实现MapReduce模式的开源的分布式并行编程框架，在Hadoop的框架上采用MapReduce模式处理海量数据。

**🟊MapReduce的工作流程、shuffle过程的作用、map端reduce端的suffle过程（135-139全部）**

为什么采用Conbiner能够减少数据传输量？

合并操作可以将相同的键值对合并成一个键值对，减少溢写到磁盘的键值对的数量。

在进行文件归并时溢写文件数量超过定义的参数，对溢写文件进行合并可以减少写入磁盘的数据量。

**是否所有的MapReduce程序都能采用Conbiner？**

如果合并使得Reduce的结果与不适用合并的结果不同，不能使用合并。

如果溢写文件未达到规定的阈值，就不执行合并操作，以免得不偿失。

读文件：

**public** **class** reader {

**public** **static** **void** main(String[] args) {

**try** {

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

conf.set("fs.defaultFS", "hdfs://localhost:9000");

conf.set("fs.hdfs.impl","org.apache.hadoop.hdfs.DistributedFileSystem");

FileSystem fs=FileSystem.*get*(conf);

Path file =**new** Path("/user/kuang/input/test.txt");

FSDataInputStream getIt =fs.open(file);

BufferedReader d =**new** BufferedReader(**new** InputStreamReader(getIt));

String content="";

**while**((content=d.readLine())!=**null**)

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

d.close();

fs.close();

}**catch**(Exception e) {

e.printStackTrace();

}

}

}

写文件：

**public** **class** writer {

**public** **static** **void** main(String[] args) {

**try** {

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

conf.set("fs.defaultFS", "hdfs://localhost:9000");

conf.set("fs.hdfs.impl", "org.apache.hadoop.hdfs.DistributedFileSystem");

FileSystem fs=FileSystem.*get*(conf);

**byte**[] buff="Hello world\n".getBytes();

String filename="output/write";

FSDataOutputStream os=fs.create(**new** Path(filename));

os.write(buff,0,buff.length);

os.close();

fs.close();

}**catch**(Exception e) {

e.printStackTrace();

}

}

}

矩阵乘法：

**public** **class** Matrixm{

**public** **static** **int** *row*=3;

**public** **static** **int** *col*=2;

**public** **static** **void** main(String[]args) **throws** IOException, ClassNotFoundException, InterruptedException{

String input1="hdfs://localhost:9000/user/kuang/input/m1";

String input2="hdfs://localhost:9000/user/kuang/input/n1";

String output="hdfs://localhost:9000/user/kuang/output/mn";

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

conf.set("fs.defaultFS", "hdfs://localhost:9000");

Job job=Job.*getInstance*(conf,"Matrix");

job.setJarByClass(Matrixm.**class**);

job.setMapperClass(MyMapper.**class**);

job.setMapOutputKeyClass(Text.**class**);

job.setMapOutputValueClass(Text.**class**);

job.setReducerClass(MyReducer.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(IntWritable.**class**);

FileInputFormat.*setInputPaths*(job,**new** Path(input1),**new** Path(input2));

Path outpath=**new** Path(output);

outpath.getFileSystem(conf).delete(outpath,**true**);

FileOutputFormat.*setOutputPath*(job,outpath);

System.*exit*(job.waitForCompletion(**true**) ?0:1);

}

**public** **static** **class** MyMapper **extends** Mapper<LongWritable, Text, Text, Text>{

String flag=**null**;

**public** MyMapper() {

}

@Override

**protected** **void** setup(Context context) **throws** IOException,

InterruptedException {

flag = ((FileSplit) context.getInputSplit()).getPath().getName();// 获取文件名称

}

@Override

**protected** **void** map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException{

String[] token1=value.toString().split(" ");

**if**(flag.equals("m1")) {

String[] token2=token1[0].split(",");

**for**(**int** i=1;i<=*col*;i++) {

Text k=**new** Text(token2[0] + "," + i);

Text v=**new** Text("m1,"+ token2[1] + "," + token1[1]);

System.***out***.println("k:"+k);

System.***out***.println("v:"+v);

context.write(k,v);

}

}**else** **if**(flag.equals("n1")) {

String[] token2=token1[0].split(",");

**for**(**int** i=1;i<=*row*;i++) {

Text k=**new** Text(i+ "," + token2[1] );

Text v=**new** Text("n1,"+ token2[0] + "," + token1[1]);

System.***out***.println("k:"+k);

System.***out***.println("v:"+v);

context.write(k, v);

}

}

**else** {

System.***out***.println("not found file");

}

}

}

**public** **static** **class** MyReducer **extends** Reducer<Text, Text, Text, IntWritable>{

**public** MyReducer() {

}

@Override

**protected** **void** reduce(Text key, Iterable<Text> values, Context context) **throws** IOException, InterruptedException{

Map<String, String> mapA = **new** HashMap<String, String>();

Map<String, String> mapB = **new** HashMap<String, String>();

**for** (Text value : values) {

String[] val = value.toString().split(",");

**if** ("m1".equals(val[0])) {

mapA.put(val[1], val[2]);

} **else** **if** ("n1".equals(val[0])) {

mapB.put(val[1], val[2]);

}

}

**int** result = 0;

Iterator<String> mKeys = mapA.keySet().iterator();

**while** (mKeys.hasNext()) {

String mkey = mKeys.next();

**if** (mapB.get(mkey)==**null**) {

**continue**;

}

result += Integer.*parseInt*(mapA.get(mkey))

\* Integer.*parseInt*(mapB.get(mkey));

}

context.write(key, **new** IntWritable(result));

}

}

}

Hbase例子：

**public** **class** ExampleForHbase {

**public** **static** Configuration *configuration*;

**public** **static** Connection *connection*;

**public** **static** Admin *admin*;

**public** **static** **void** main(String[] args) **throws** IOException{

*createTable*("Score",**new** String[] {"sname","course"});//1 创建一个表，表名为score，列族为name，course

// insertRow("Score","001","sname","","Mary");//name列族没有子列，所以第四个参数为空

// insertRow("Score","001","course","Math","88");

// insertRow("Score","001","course","English","85");

// deleteRow("Score","001","course","Math");

// deleteRow("Score","001","course","");

// deleteRow("Score","001","","");

// getData("Score","001","course","Math");

// getData("Score","001","sname","");

// listTable();

// deleteTable("Score");

// listTable();

}

//建立连接

**public** **static** **void** inti() {

*configuration* =HBaseConfiguration.*create*();

*configuration*.set("hbase.rootdir", "hdfs://localhost:9000/hbase");

**try** {

*connection*=ConnectionFactory.*createConnection*(*configuration*);

*admin*=*connection*.getAdmin();

}**catch**(IOException e) {

e.printStackTrace();

}

}

//关闭连接

**public** **static** **void** close() {

**try** {

**if**(*admin*!=**null**)

*admin*.close();

**if**(**null**!=*connection*)

*connection*.close();

}**catch**(IOException e) {

e.printStackTrace();

}

}

//建表

**public** **static** **void** createTable(String myTableName,String[] colFamily) **throws** IOException{

*inti*();

TableName tableName=TableName.*valueOf*(myTableName);

**if**(*admin*.tableExists(tableName))

System.***out***.println("table is exists!");

**else** {

HTableDescriptor hTableDescriptor=**new** HTableDescriptor(tableName);

**for**(String str:colFamily) {

HColumnDescriptor hColumnDescriptor= **new** HColumnDescriptor(str);

hTableDescriptor.addFamily(hColumnDescriptor);

}

*admin*.createTable(hTableDescriptor);

System.***out***.println("create table success");

}

*close*();

}

//删除指定表

**public** **static** **void** deleteTable(String tableName) **throws** IOException{

*inti*();

TableName tn=TableName.*valueOf*(tableName);

**if**(*admin*.tableExists(tn)) {

*admin*.disableTable(tn);

*admin*.deleteTable(tn);

}

*close*();

}

//查看已有表

**public** **static** **void** listTable() **throws** IOException{

*inti*();

HTableDescriptor hTableDescriptors[]=*admin*.listTables();

**for** (HTableDescriptor hTableDescriptor:hTableDescriptors) {

System.***out***.println(hTableDescriptor.getNameAsString());

}

*close*();

}

//向某行某列插入数据

**public** **static** **void** insertRow(String tableName,String rowKey,String colFamily,String col,String val)**throws** IOException{

*inti*();

Table table=*connection*.getTable(TableName.*valueOf*(tableName));

Put put =**new** Put(rowKey.getBytes());

put.addColumn(colFamily.getBytes(), col.getBytes(), val.getBytes());

table.put(put);

table.close();

*close*();

}

//删除数据

**public** **static** **void** deleteRow(String tableName,String rowKey,String colFamily,String col) **throws** IOException{

*inti*();

Table table =*connection*.getTable(TableName.*valueOf*(tableName));

Delete delete=**new** Delete(rowKey.getBytes());

table.delete(delete);

table.close();

*close*();

}

//根据行键查找数据

**public** **static** **void** getData(String tableName,String rowKey,String colFamily,String col)**throws** IOException{

*inti*();

Table table =*connection*.getTable(TableName.*valueOf*(tableName));

Get get=**new** Get(rowKey.getBytes());

get.addColumn(colFamily.getBytes(), col.getBytes());

Result result=table.get(get);

*showCell*(result);

table.close();

*close*();

}

**public** **static** **void** showCell(Result result) {

Cell[] cells=result.rawCells();

**for**(Cell cell:cells) {

System.***out***.println("RowName:"+**new** String(CellUtil.*cloneRow*(cell))+" ");

System.***out***.print("Timetamp:"+cell.getTimestamp()+" ");

System.***out***.println("column Family:"+**new** String(CellUtil.*cloneFamily*(cell))+" ");

System.***out***.println("row Name:"+**new** String(CellUtil.*cloneQualifier*(cell))+" ");

System.***out***.println("value:"+**new** String(CellUtil.*cloneValue*(cell))+" ");

}

}

}