分布式数据库Hbase

BIG DATA

Hbase

CONTENTS

重庆大学 CHONGQING UNIVERSITY

1.1 从BigTable说起

Google

- BigTable是谷歌公司研发的一个分布式存储系统，用于解决其大规模网页存储和搜索问题

- BigTable建立网页索引流程
  - 网页爬虫持续不断地抓取新页面，这些页面每页一行地存储到BigTable里
  - MapReduce计算作业运行在整张表上，生成索引，为网络搜索应用做准备
- 网页搜索过程：

用户发起网络搜索请求 → 网络搜索应用查询建立好的索引，从BigTable得到网页 → 搜索结果提交给用户

- BigTable的底层架构在GFS（谷歌文件系统）之上，并通过Chubby进行协同管理

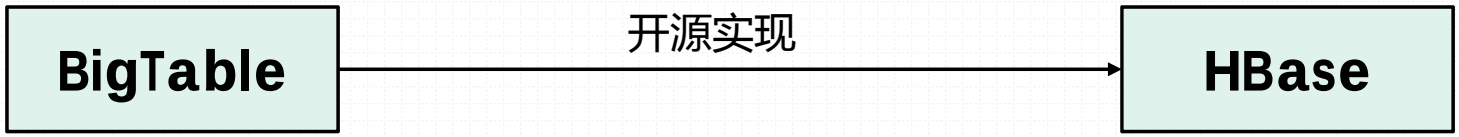
目前，谷歌的BigTable不仅用于网页搜索，还用于谷歌地图、财经、社交、视频等方面

1.1 从BigTable说起

Google

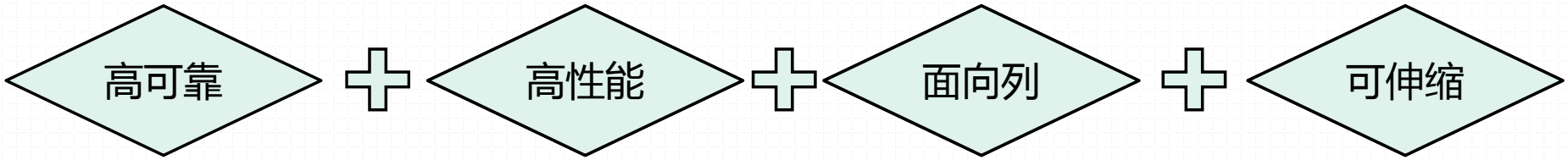
■ 谷歌BigTable为什么能受到如此广泛的关注呢？

- 它具有非常好的性能（可以支持PB级别的数据）
- 它具有非常好可扩展性（用几千台服务器组成的集群完成大规模数据的分布式存储）



1.2 HBase是什么？

■ HBase是BigTable的开源实现，具有如下特点：



HBase → **分布式数据库** → **适合存储半结构化和非结构化的松散数据**

HBase支持水平扩展，允许几千台服务器去存储海量数据和文件

规模：十亿行、几百万列的数据库

1.2 HBase是什么？

■ HBase与BigTable底层技术的关系对比

对比项	HBase	BigTable
文件存储系统	HDFS	GFS
海量数据处理	Hadoop MapReduce	MapReduce
协同管理服务器	ZooKeeper	Chubby

1.3 HBase的产生背景

■ 为什么需要设计HBase这么一个数据库产品？

原因

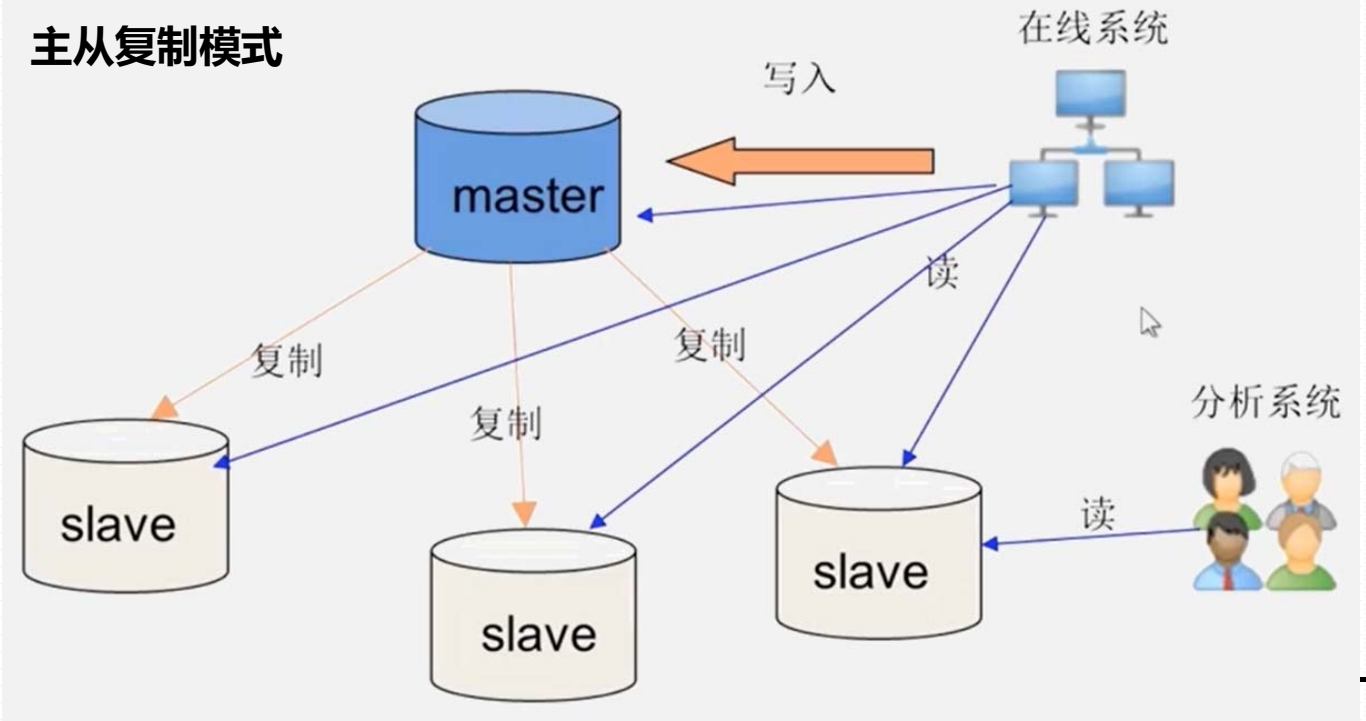
- 对于Hadoop来说，虽然已经有了HDFS和MapReduce，但MapReduce主要用于解决大规模数据的离线批处理，无法满足大规模数据的实时处理需求
- 对于传统关系型数据库来说，随着大数据时代数据规模的爆炸式增长，其扩展能力非常有限

1.3 HBase的产生背景

■ 传统关系型数据库的扩展方法

- **主从复制**（不能解决数据写入负载的问题）
- **分库**（不同部门的数据单独作为一个库）
- **分表**（把同一张表中的记录分散存储到多张表）

主从复制模式



上述方式还是无法从根源上解决数据库的可扩展性问题

- 操作不便利
- 效率低

1.3 HBase的产生背景

■ 总结：

- Hadoop MapReduce面向离线批处理模式，无法满足大规模数据实时处理应用的需求
- 传统关系型数据库无法应对数据规模剧增时导致的系统扩展性和性能问题
- 传统关系型数据库在数据结构变化时一般需要停机维护
- 传统关系型数据库存在大量空列浪费存储空间

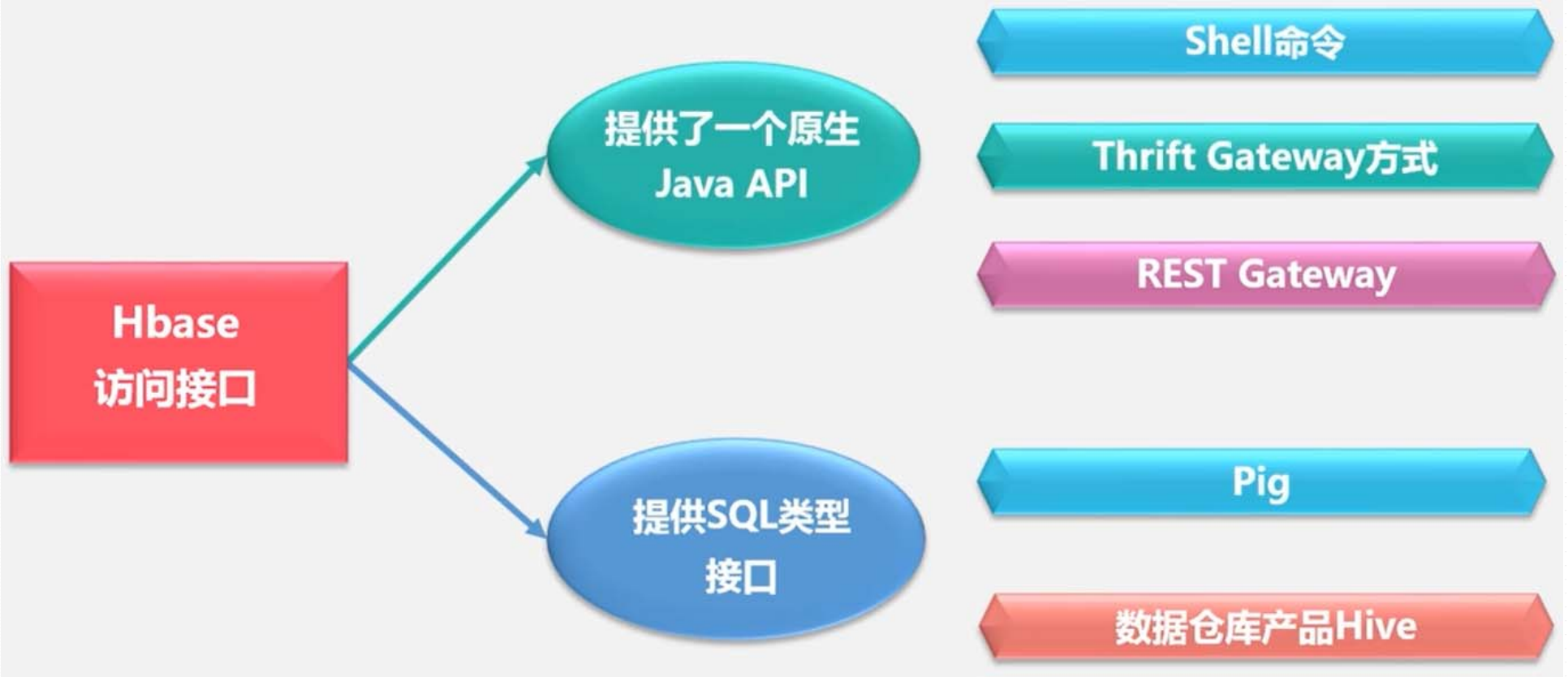
因此，业界出现了一类面向半结构化数据存储和处理的高可扩展、低写入/查询延迟的系统，例如，键值数据库、文档数据库和列族数据库（如BigTable和HBase等）

HBase的数据分片与扩展都是全自动的，不需要任何人工参与，具有极高的可扩展性

1.4 Hbase与传统关系型数据库的联系和区别

对比项	关系型数据库	HBase
数据类型方面	使用经典的关系数据模型，具有丰富的数据类型和存储方式	采用更加简单的数据模型，把数据存储为未经解释的字符串
数据操作方面	包含丰富的操作，涉及复杂的多表连接	不存在复杂的表与表之间的关系，避免了复杂的表和表之间的关系
存储模式方面	基于行模式存储	基于列模式存储，每个列族都由几个文件保存，不同列族的文件是分离的
数据索引方面	可以针对不同列构建复杂的多个索引，以提高数据访问性能	只支持对行键的简单索引
数据维护方面	数据更新操作会覆盖旧的值	不会删除旧数据，会保存数据的多个版本，并通过时间戳标识
可伸缩性方面	很难实现水平扩展，最多可实现纵向扩展（CPU、内存、存储等）	具有很好的水平可扩展性

1.5 HBase的访问接口



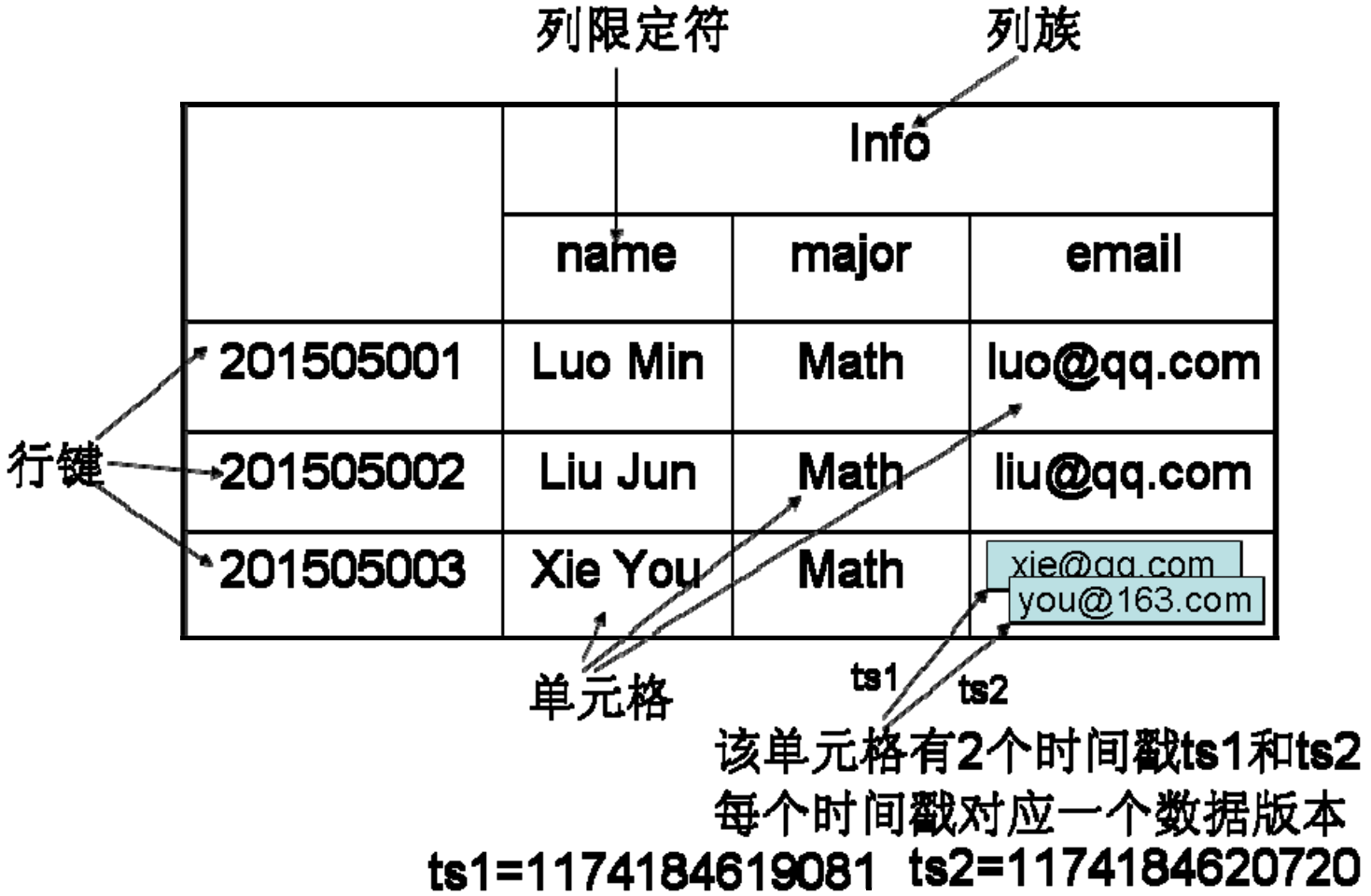
1.5 HBase的访问接口

接口类型	特点	场合
Native Java API	最常规和高效的访问方式	适合Hadoop MapReduce作业并行批处理HBase表数据
HBase Shell	HBase的命令行工具，最简单的接口	适合HBase管理使用
Thrift Gateway	利用Thrift序列化技术，支持C++、PHP、Python等多种语言	适合其他异构系统在线访问HBase表数据
REST Gateway	解除了语言限制	支持REST风格的Http API访问HBase
Pig	使用Pig Latin流式编程语言来处理HBase中的数据	适合做数据统计
Hive	简单	当需要以类似SQL语言方式来访问HBase的时候

Hbase

CONTENTS

2.1 HBase是一个稀疏的多维度的排序的映射表



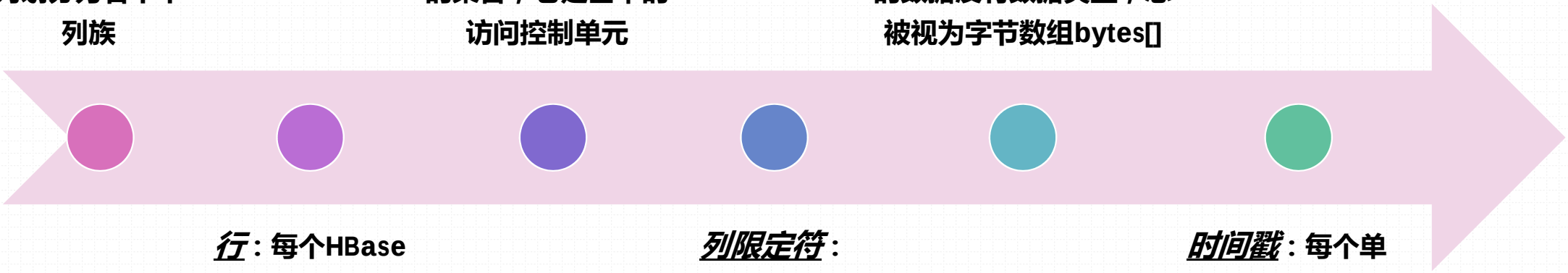
2 HBase的数据模型

2.1 HBase是一个稀疏的多维度的排序的映射表

表：HBase采用表来组织数据，表由行和列组成，列划分为若干个列族

列族：一个HBase表被分组成许多“列族”（Column Family）的集合，它是基本的访问控制单元

单元格：在HBase表中，通过行、列族和列限定符确定一个“单元格”（cell），单元格中存储的数据没有数据类型，总被视为字节数组bytes[]



行：每个HBase表都由若干行组成，每个行由行键（row key）来标识。

列限定符：列族里的数据通过列限定符（或列）来定位

时间戳：每个单元格都保存着同一份数据的多个版本，这些版本采用时间戳进行索引

2.1 HBase是一个稀疏的多维度的排序的映射表

HBase

- 列限定符（列）
- 每一个值都是未指定类型的Bytes数组，程序员需要自己对数据类型进行解析和处理
- 一个行可以有一个行键和任意多个列
- 列族是HBase中非常核心的概念

2.2 HBase数据模型的核心——列族

■ 列族特点

支持动态扩展

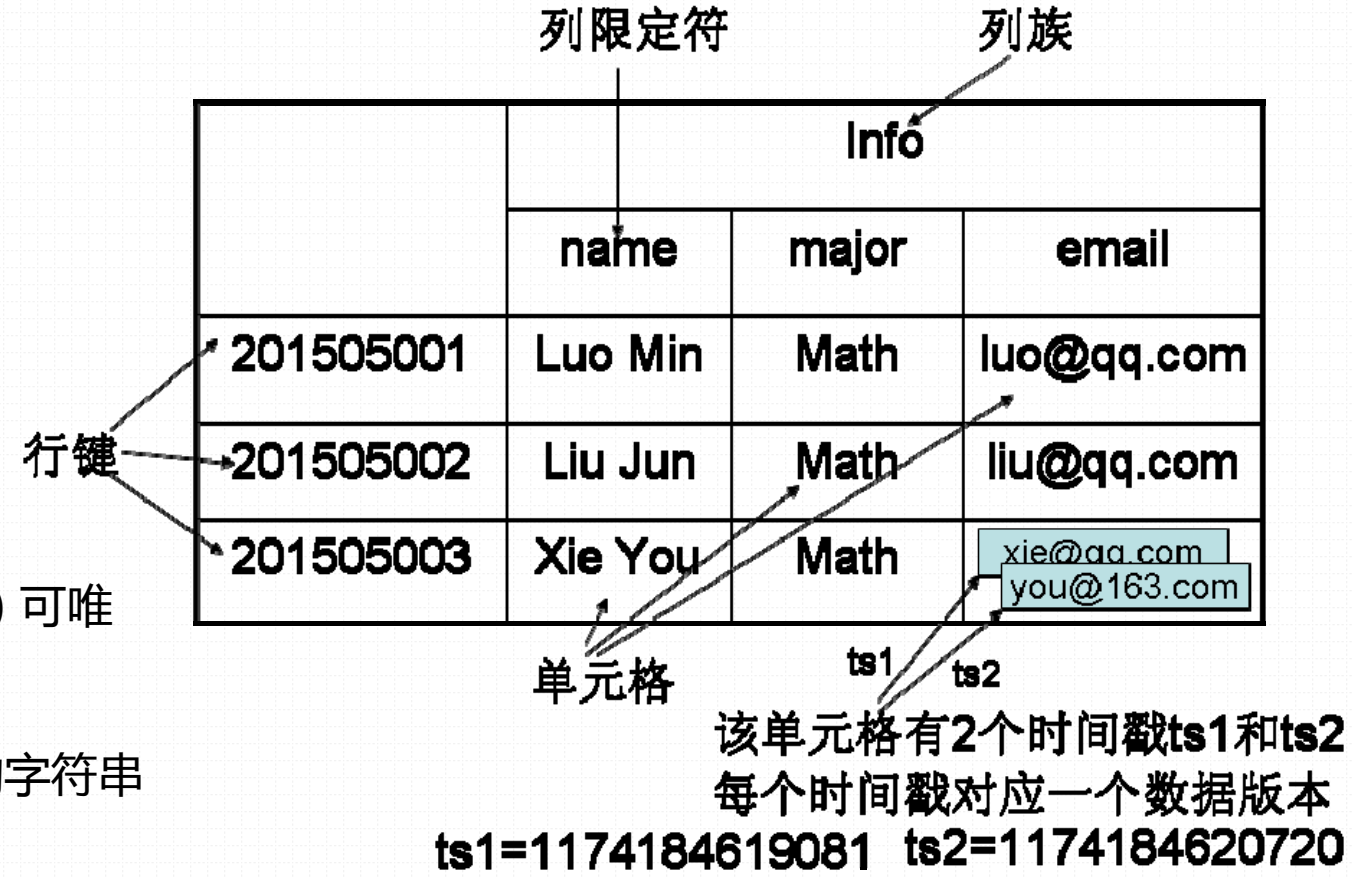
可以很轻松地添加一个列族或列，无需预先定义列的数量以及类型，所有列均以字符串形式存储，用户需要自行进行数据类型转换

保留旧的版本

HBase中执行更新操作时，并不会删除数据旧的版本，而是生成一个新的版本，旧有的版本仍然保留（这是和HDFS只允许追加不允许修改的特性相关的）

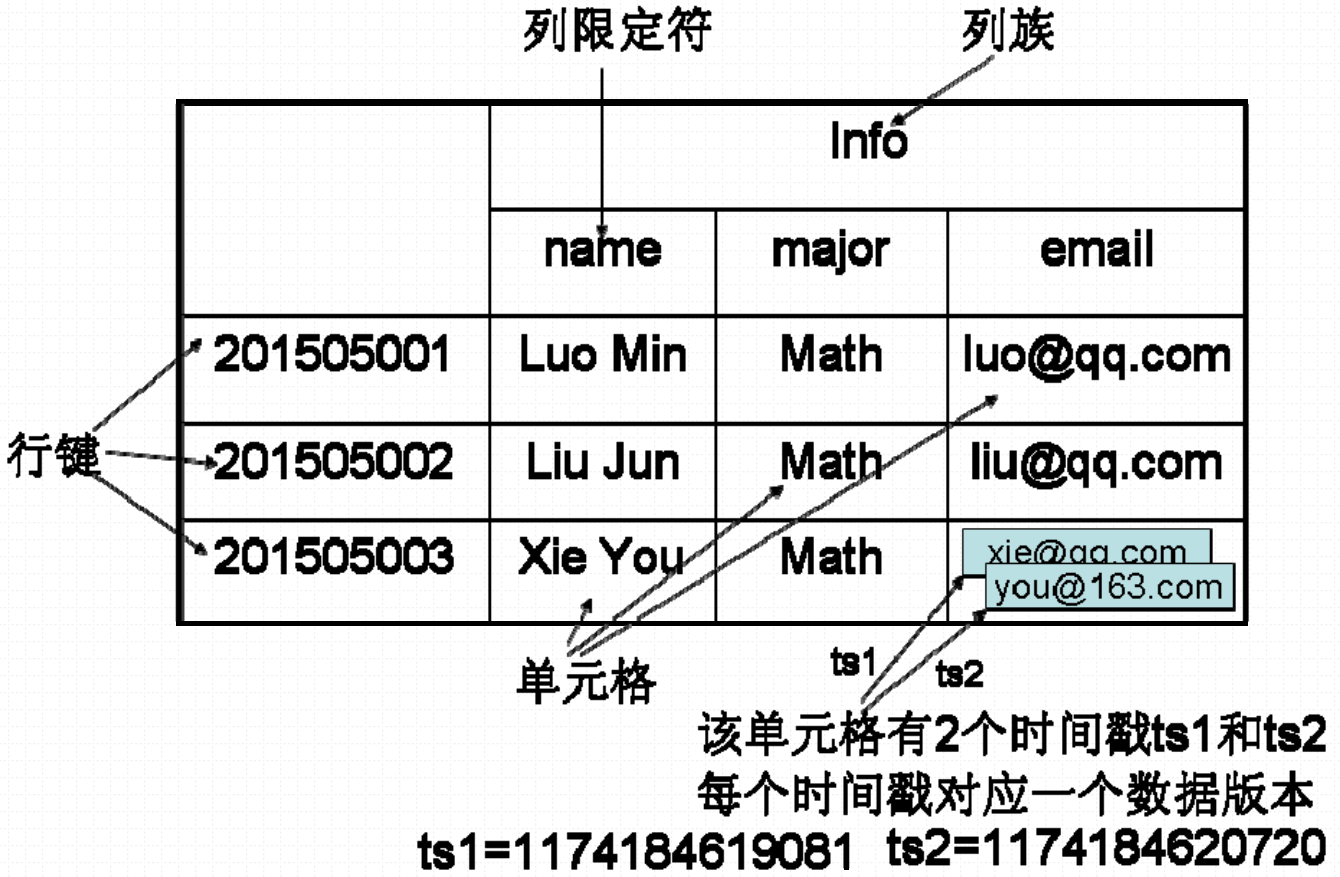
2.2 HBase数据模型——列限定符和单元格

- 列限定符（列）
  - name
  - major
  - email
- 单元格
  - 实际存储数据的地方
  - 通过 (行, 列族, 列限定符) 可唯一确定一个单元格
  - 单元格内容是未经解释的字符串（Bytes数组）

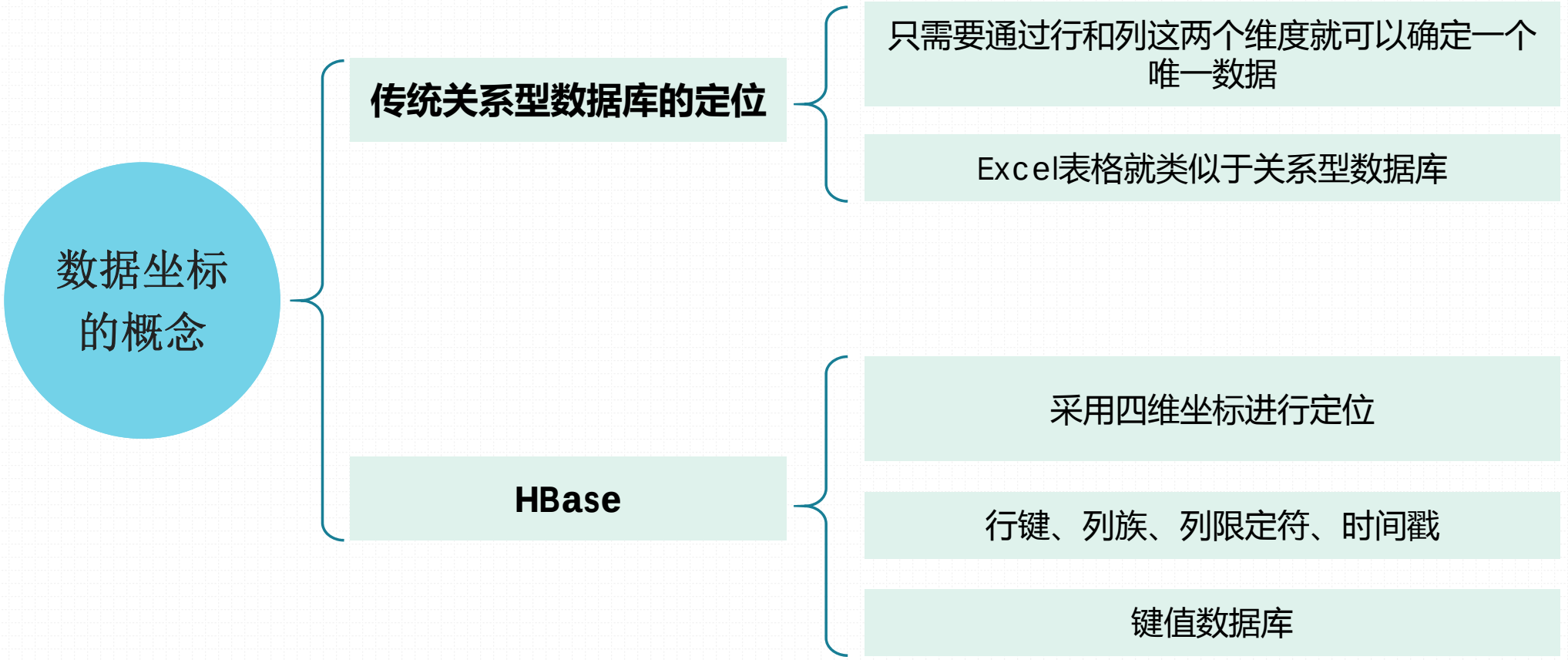


2.2 HBase数据模型——时间戳

- 时间戳
  - 用于区分同一单元格中数据的不同版本
- 数据坐标
  - 传统关系型数据库：(行, 列)
  - HBase：(行, 列族, 列, 时间戳)



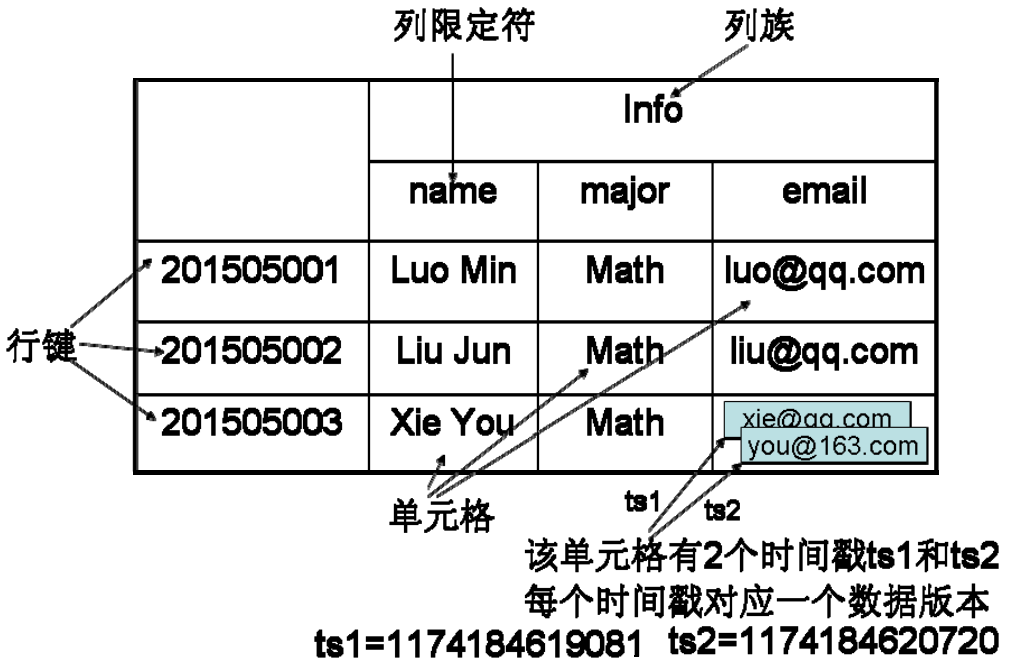
2.3 HBase数据坐标



2.3 HBase数据坐标

■ 数据坐标示例

键	值
[“201505003”, “Info”, “email”, 1174184619081]	“xie@qq.com”
[“201505003”, “Info”, “email”, 1174184620720]	“you@163.com”



2.3 HBase数据的概念视图

■ 概念视图：反映HBase数据的逻辑结构

HBase数据的概念视图

行键	时间戳	列族contents	列族anchor
"com.cnn.www"	t5		anchor:cnnsi.com=”CNN”
	t4		anchor:my.look.ca="CNN.com"
	t3	contents:html="<html>..."	
	t2	contents:html="<html>..."	
	t1	contents:html="<html>..."	

2.4 HBase数据的物理视图

■ 物理视图：反映HBase数据的实际存储方式

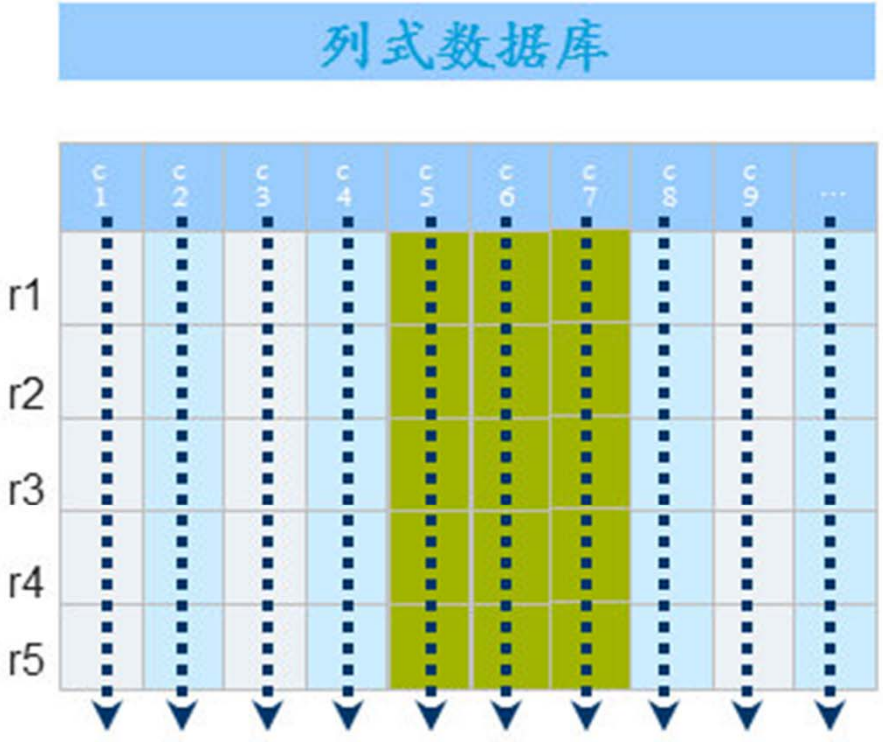
列族contents

行键	时间戳	列族contents
"com.cnn.www"	t3	contents:html="<html>..."
	t2	contents:html="<html>..."
	t1	contents:html="<html>..."

列族anchor

行键	时间戳	列族anchor
"com.cnn.www"	t5	anchor:cnnsi.com=”CNN”
	t4	anchor:my.look.ca="CNN.com"

2.5 行式存储结构和列式存储结构

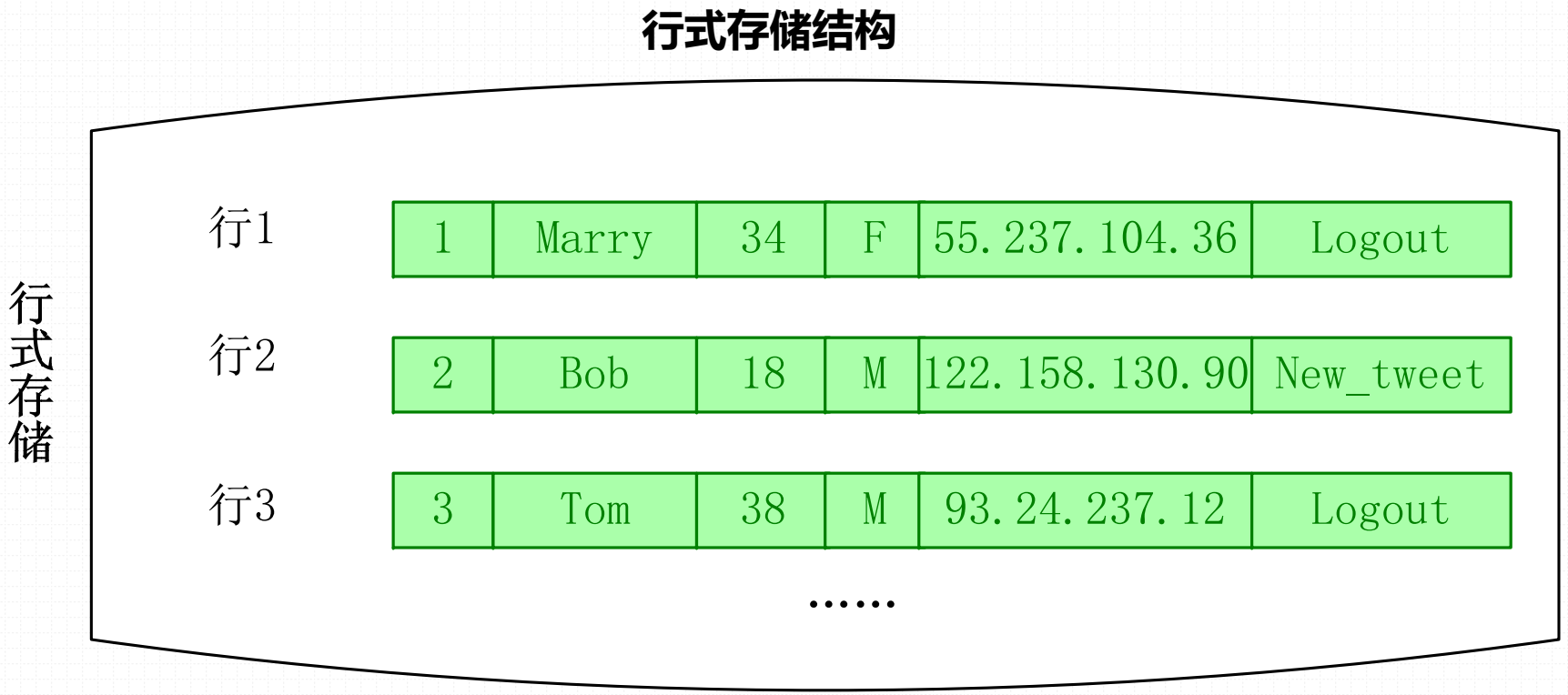


2.5 行式存储结构和列式存储结构

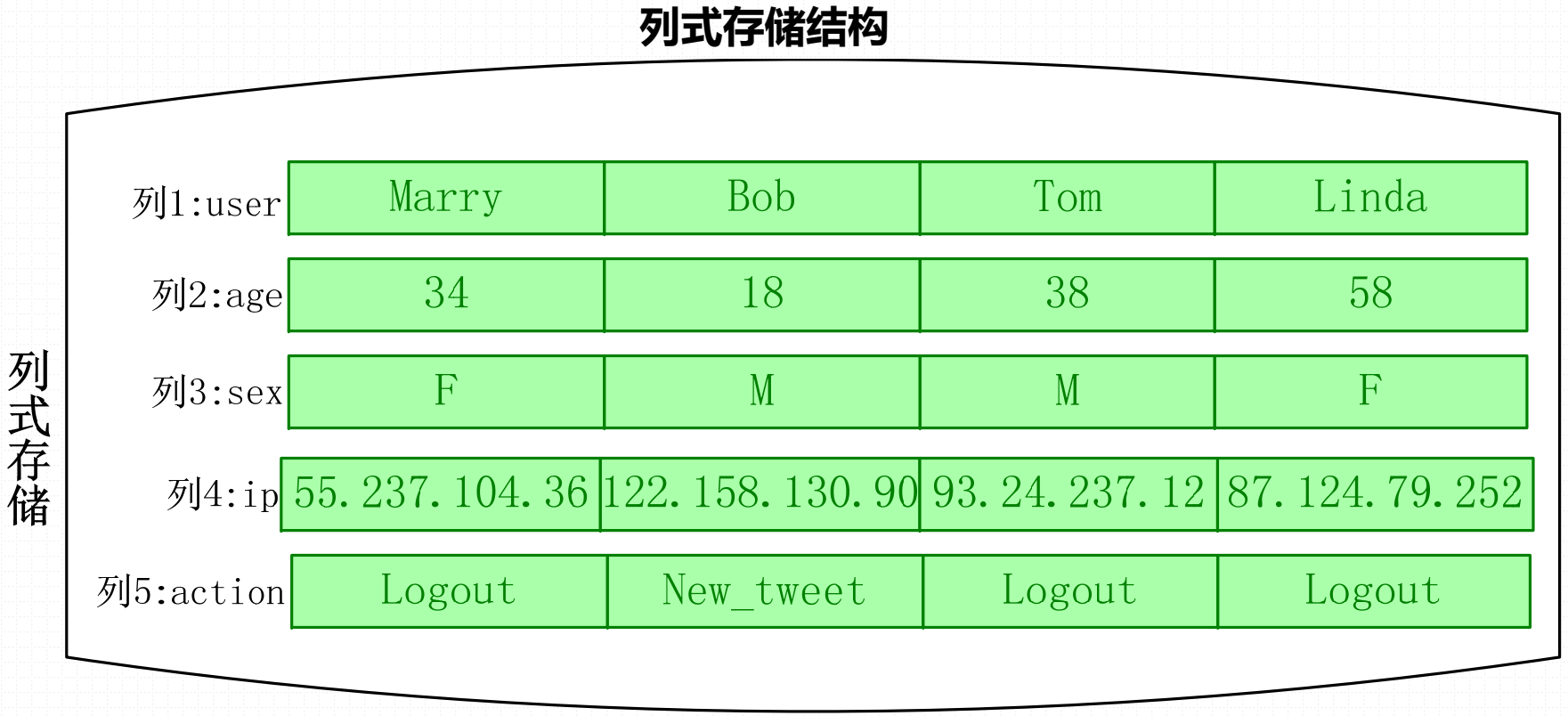
SQL模式

Log					
Log_id	user	age	sex	ip	action
1	Marry	34	F	55.237.104.36	Logout
2	Bob	18	M	122.158.130.90	New_tweet
3	Tom	38	M	93.24.237.12	Logout
4	Linda	58	F	87.124.79.252	Logout

2.5 行式存储结构和列式存储结构



2.5 行式存储结构和列式存储结构



行式存储结构的优缺点

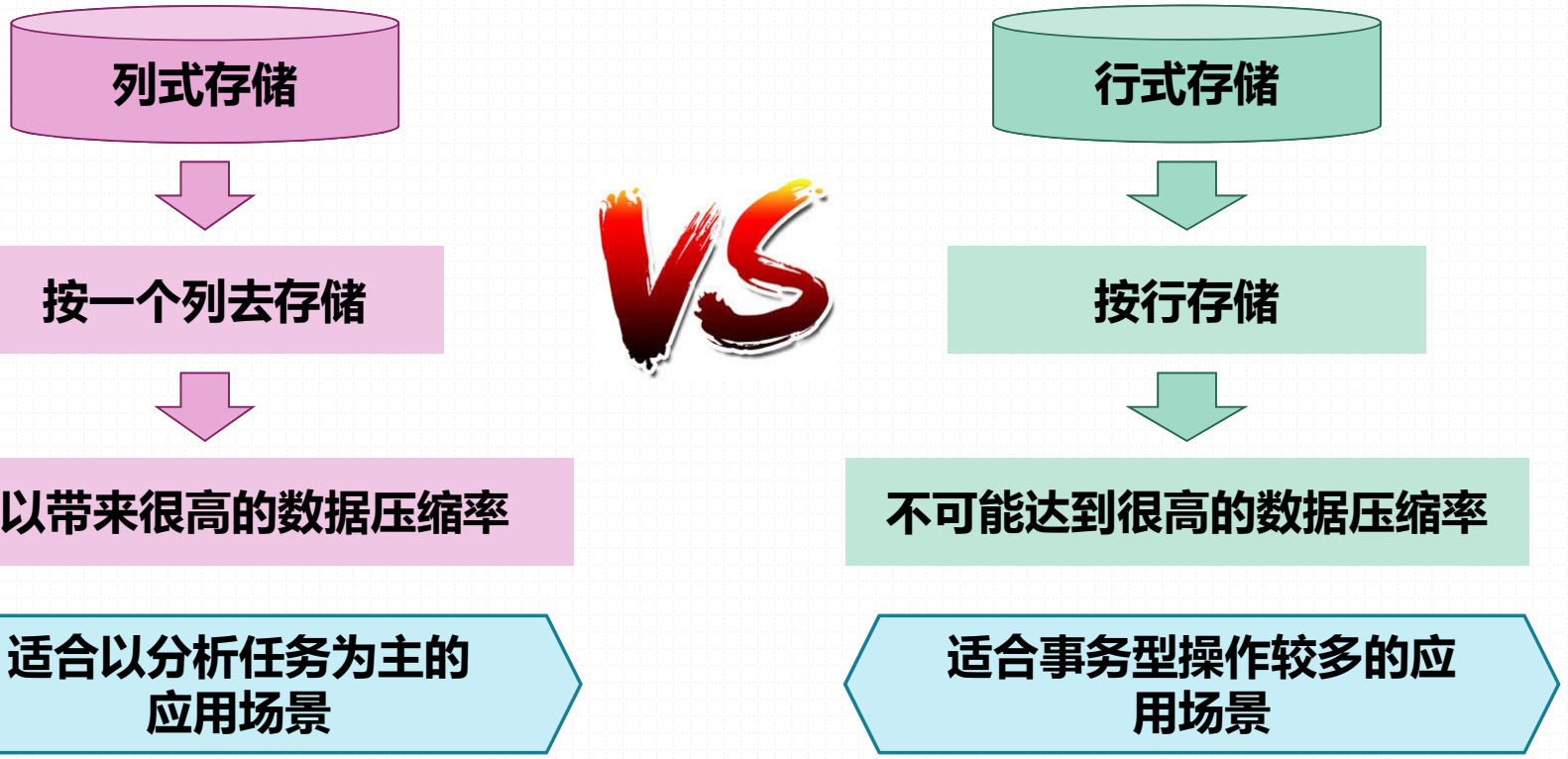
优点

- 对于传统的事务型操作，每次需要插入一条记录的时候，会把这条记录的各个信息都存入数据库
- 在OLTP系统中，每一次都生成一个完整的记录

缺点

- 对于行存储模式来说，为了取出某一个特定列的数据，需要逐行扫描每一条记录，并取出该字段
- 大多数情况下，数据分析任务都是在列上进行的，如分析用户的年龄、性别、行业分布等

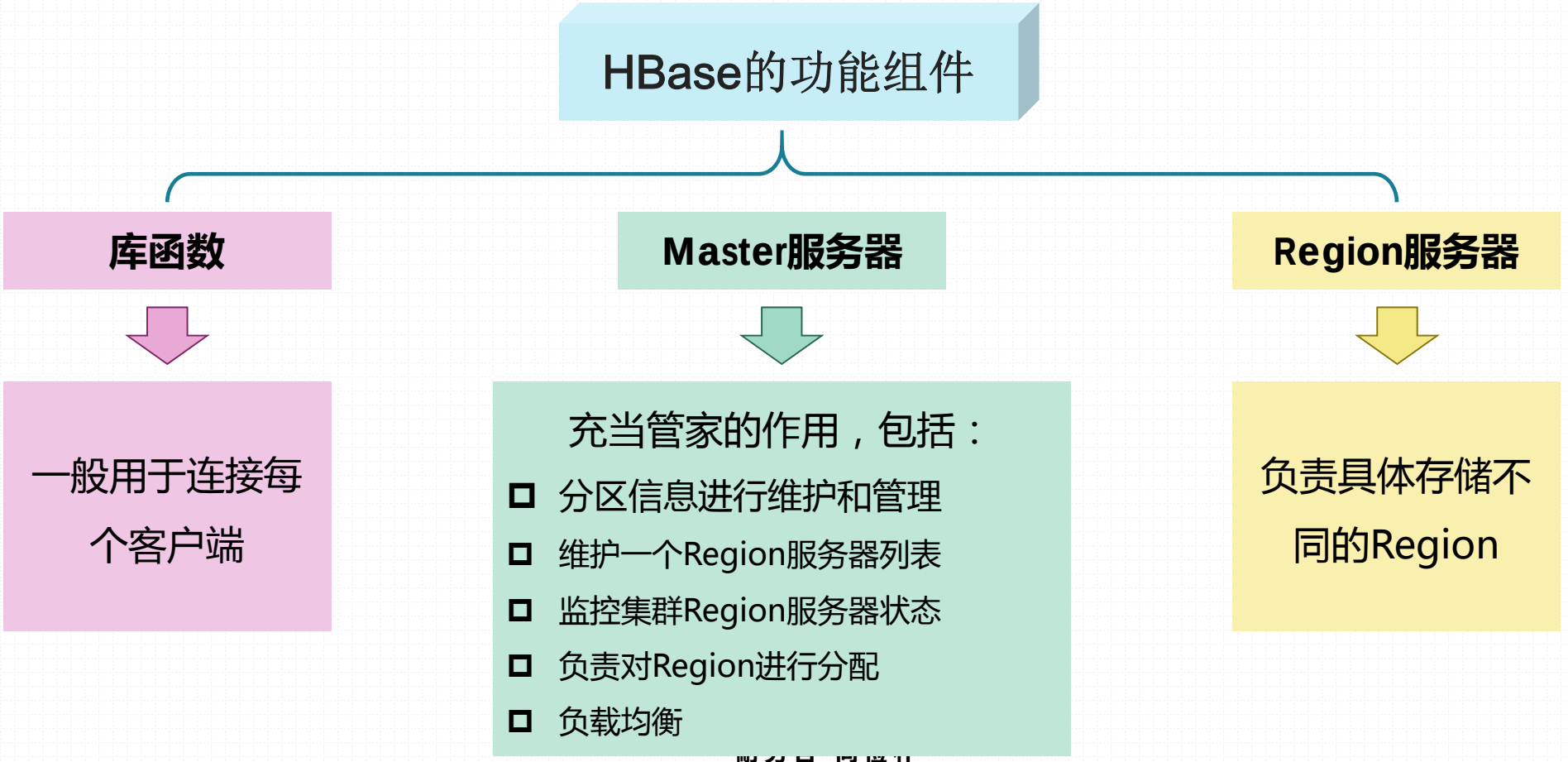
行式存储和列式存储对比



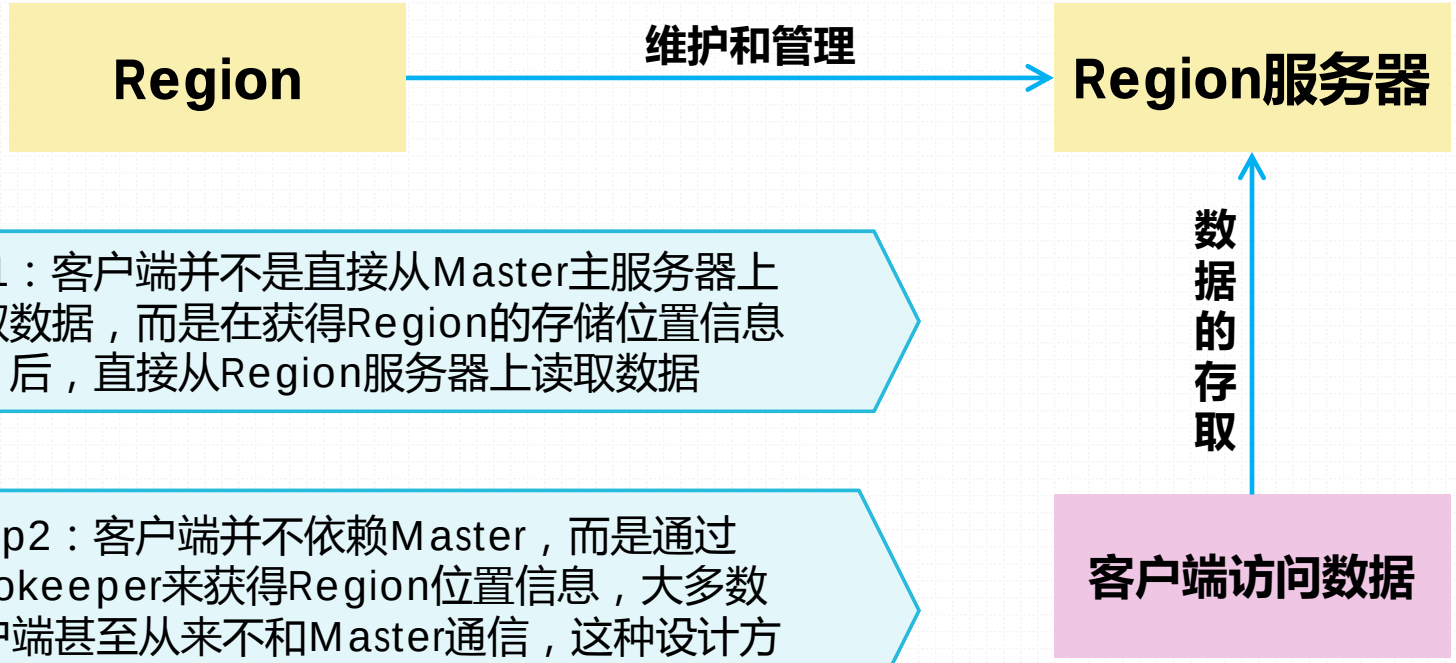
Hbase

CONTENTS

HBase的功能组件


HBase的功能组件
库函数
Master服务器
Region服务器
一般用于连接每个客户端
充当管家的作用，包括：
分区信息进行维护和管理
维护一个Region服务器列表
监控集群Region服务器状态
负责对Region进行分配
负载均衡
负责具体存储不同的Region

HBase的功能组件

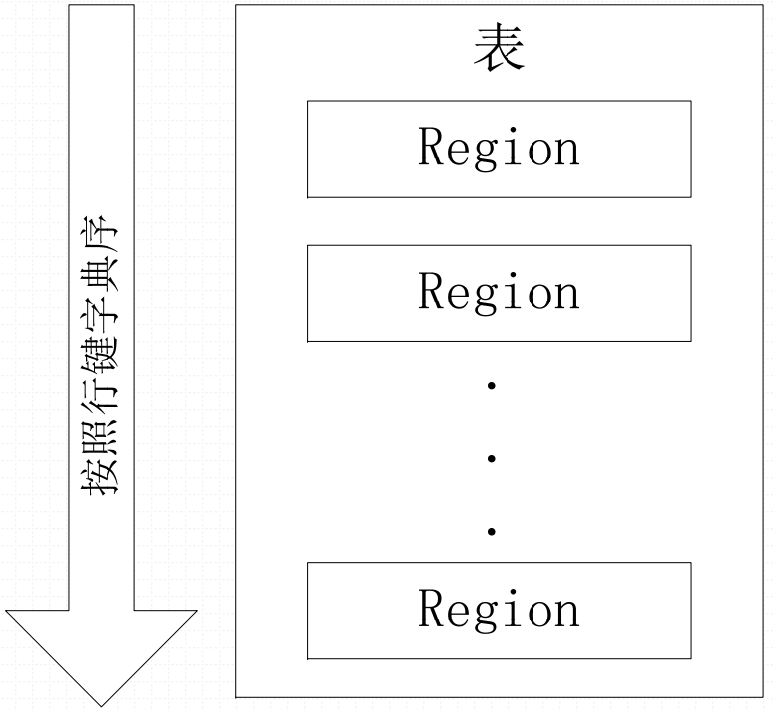


Tip1：客户端并不是直接从Master主服务器上读取数据，而是在获得Region的存储位置信息后，直接从Region服务器上读取数据

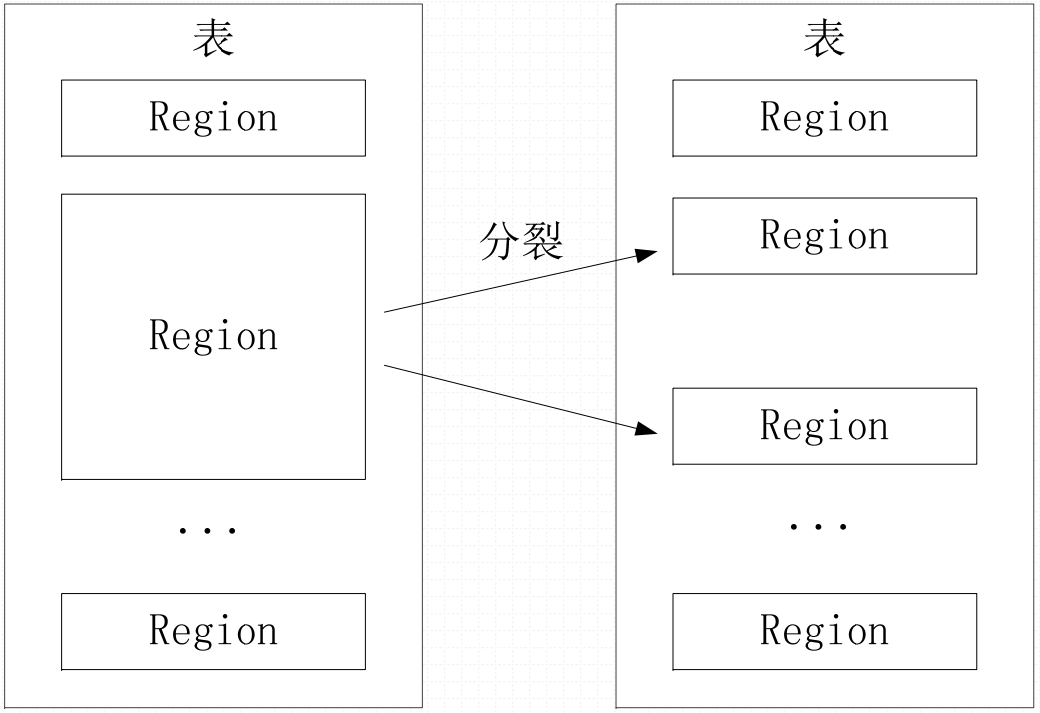
Tip2：客户端并不依赖Master，而是通过Zookeeper来获得Region位置信息，大多数客户端甚至从来不和Master通信，这种设计方式使得Master负载很小

HBase的表和Region

一个HBase表被划分为多个Region

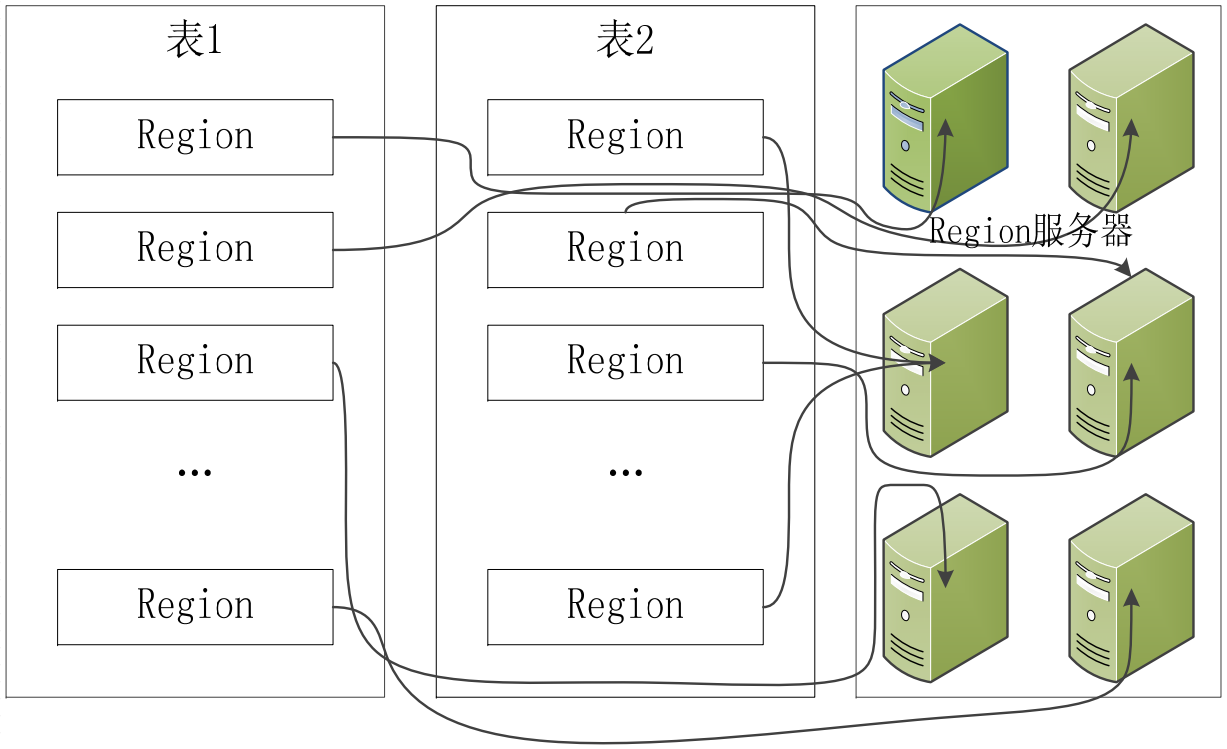


一个Region会分裂成多个新的Region



HBase的表和Region

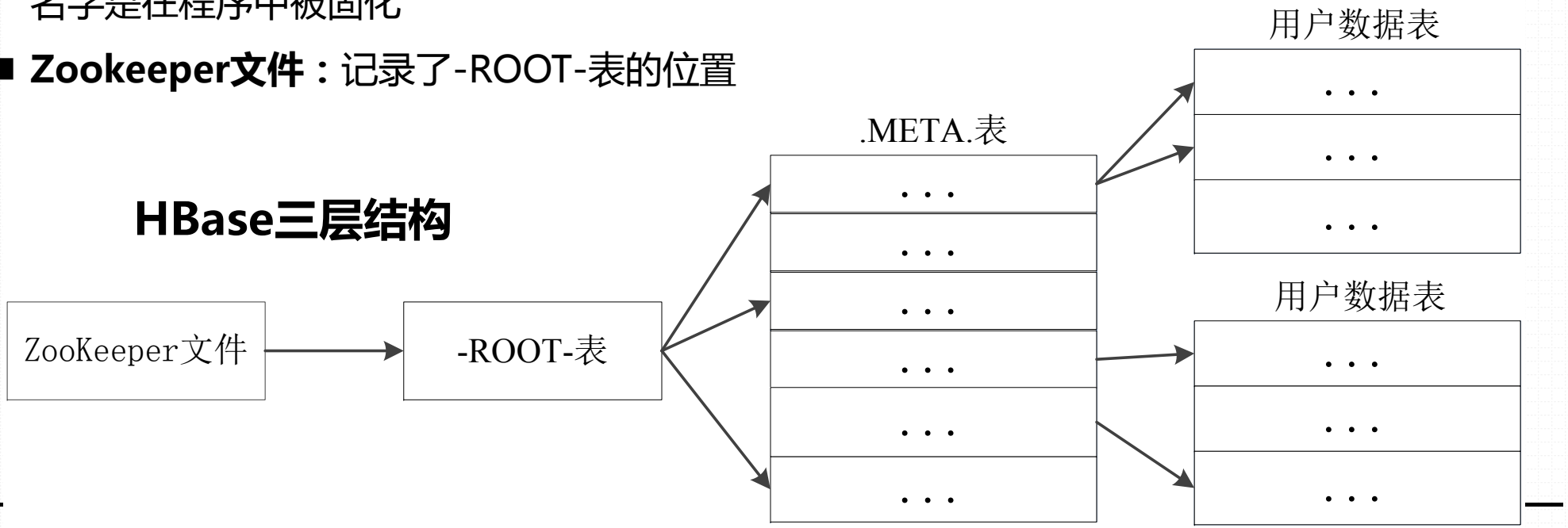
- 2006年以前，一般推荐一个Region的大小为100~200MB
- 目前，一个Region的最佳大小配置为1GB到2GB
- 实际Region的大小，取决于单台服务器的有效处理能力
- 同一个Region通常不会被拆分到不同的Region服务器上去
- 每一个Region服务器大概可以存10~1000个Region



HBase中Region的定位

- **元数据表**：又名.META.表，存储了Region和Region服务器的映射关系，当HBase表很大时， .META.表也会被分裂成多个Region
- **根数据表**：又名-ROOT-表，记录所有元数据的具体位置，-ROOT-表只有唯一一个Region，名字是在程序中被固化
- **Zookeeper文件**：记录了-ROOT-表的位置

HBase三层结构



HBase的三层结构中各层次的名称和作用

层次	名称	作用
第一层	Zookeeper文件	记录了-ROOT-表的位置信息
第二层	-ROOT-表	记录了.META.表的Region位置信息 -ROOT-表只能有一个Region。通过-ROOT-表，就可以访问.META.表中的数据
第三层	.META.表	记录了用户数据表的Region位置信息，.META.表可以有多个Region，保存了HBase中所有用户数据表的Region位置信息

HBase中Region的定位

Region定位

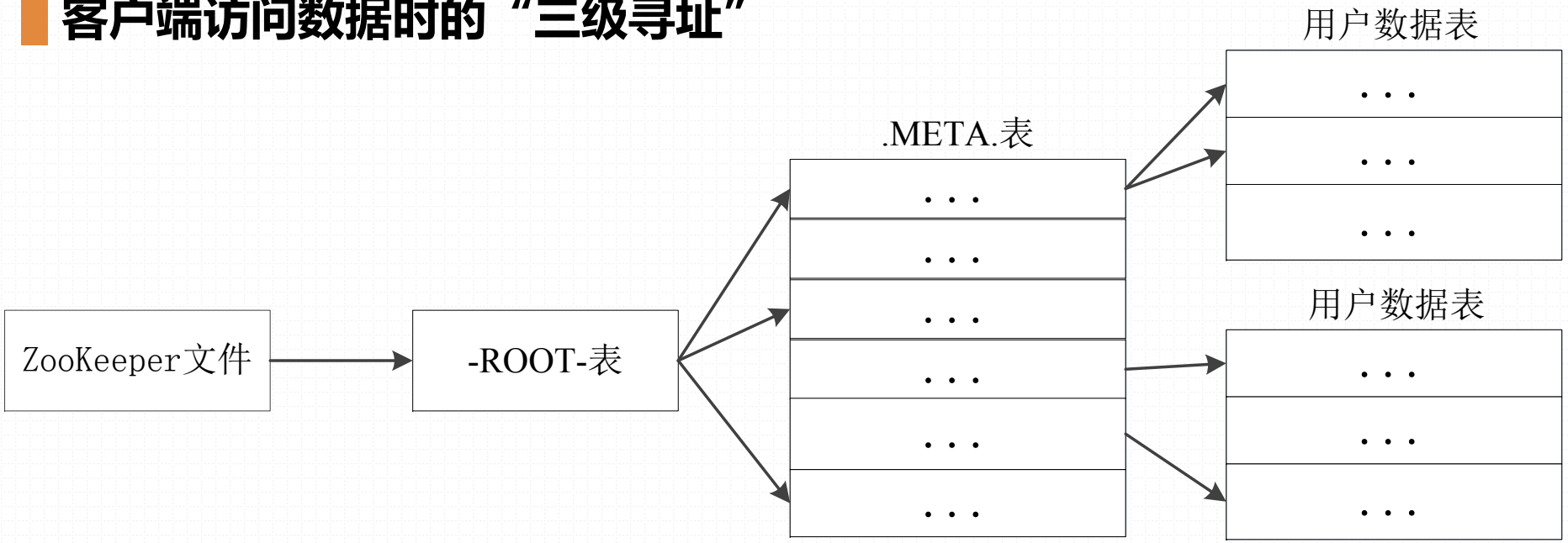
- 为了加快访问速度，.META.表的全部Region都会被保存在内存中
- 假设.META.表的每行（一个映射条目）在内存中大约占用1KB，并且每个Region限制为128MB，那么，上面的三层结构可以保存的用户数据表的Region数目的计算方法是：
- （-ROOT-表能够寻址的.META.表的Region个数）×（每个.META.表的Region可以寻址的用户数据表的Region个数）

HBase中Region的定位

Region定位

- 一个-ROOT-表最多只能有一个Region，也就是最多只能有128MB，按照每行（一个映射条目）占用1KB内存计算，128MB空间可以容纳128MB/1KB=2^{17}行，也就是说，一个-ROOT-表可以寻址2^{17}个.META.表的Region。
- 同理，每个.META.表的 Region可以寻址的用户数据表的Region个数是128MB/1KB=2^{17}个。
- 最终，三层结构可以保存的Region数目是(128MB/1KB) × (128MB/1KB) = 2^{34}个Region

客户端访问数据时的“三级寻址”



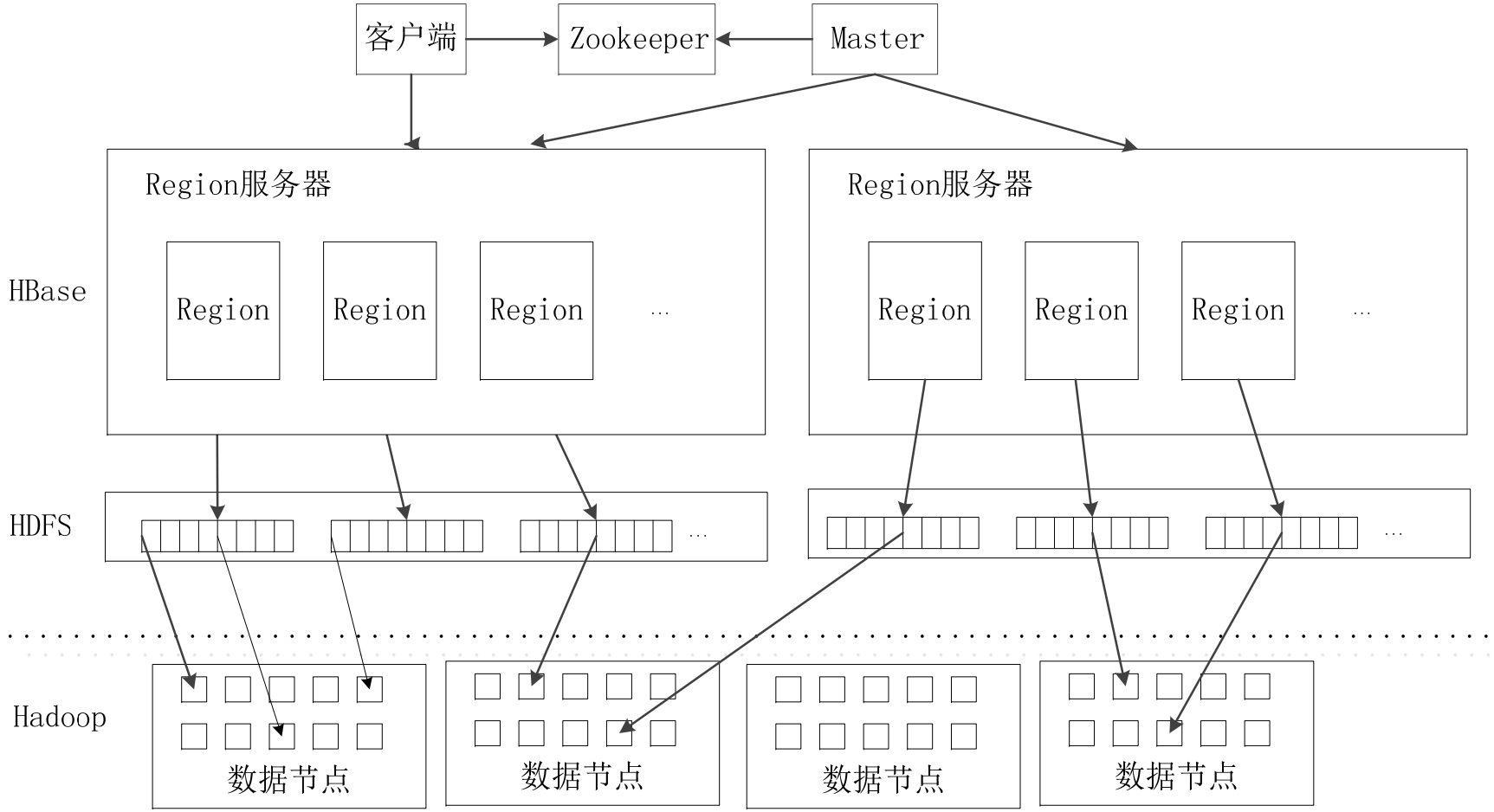
为了加速寻址，客户端会缓存位置信息，同时，需要解决缓存失效问题

寻址过程客户端只需要询问Zookeeper服务器，不需要连接Master服务器

Hbase

CONTENTS

HBase的系统架构


客户端
Zookeeper
Master
HBase
Region服务器
Region
Region
Region
…
Region服务器
Region
Region
Region
…
HDFS
…
…
Hadoop
数据节点
数据节点
数据节点
数据节点

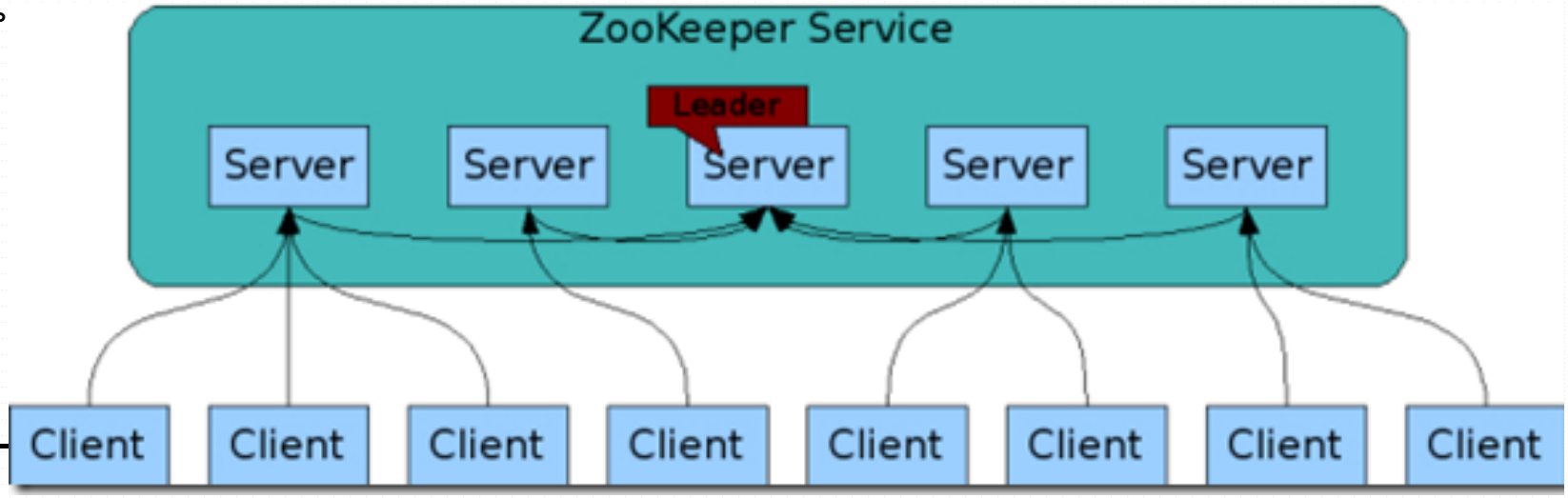
HBase的系统架构

1. 客户端

- 客户端包含访问HBase的接口，同时在缓存中维护着已经访问过的Region位置信息，用来加快后续数据访问过程

2. Zookeeper服务器

- 帮助选举出一个Master作为集群的总管，并保证在任何时刻总有唯一一个Master在运行，这就避免了Master的“单点失效”问题
- 是一个很好的集群管理工具，被大量用于分布式计算，提供配置维护、域名服务、分布式同步、组服务等。



HBase的系统架构

3. Master（主服务器）：主要负责表和Region的管理工作，包括：

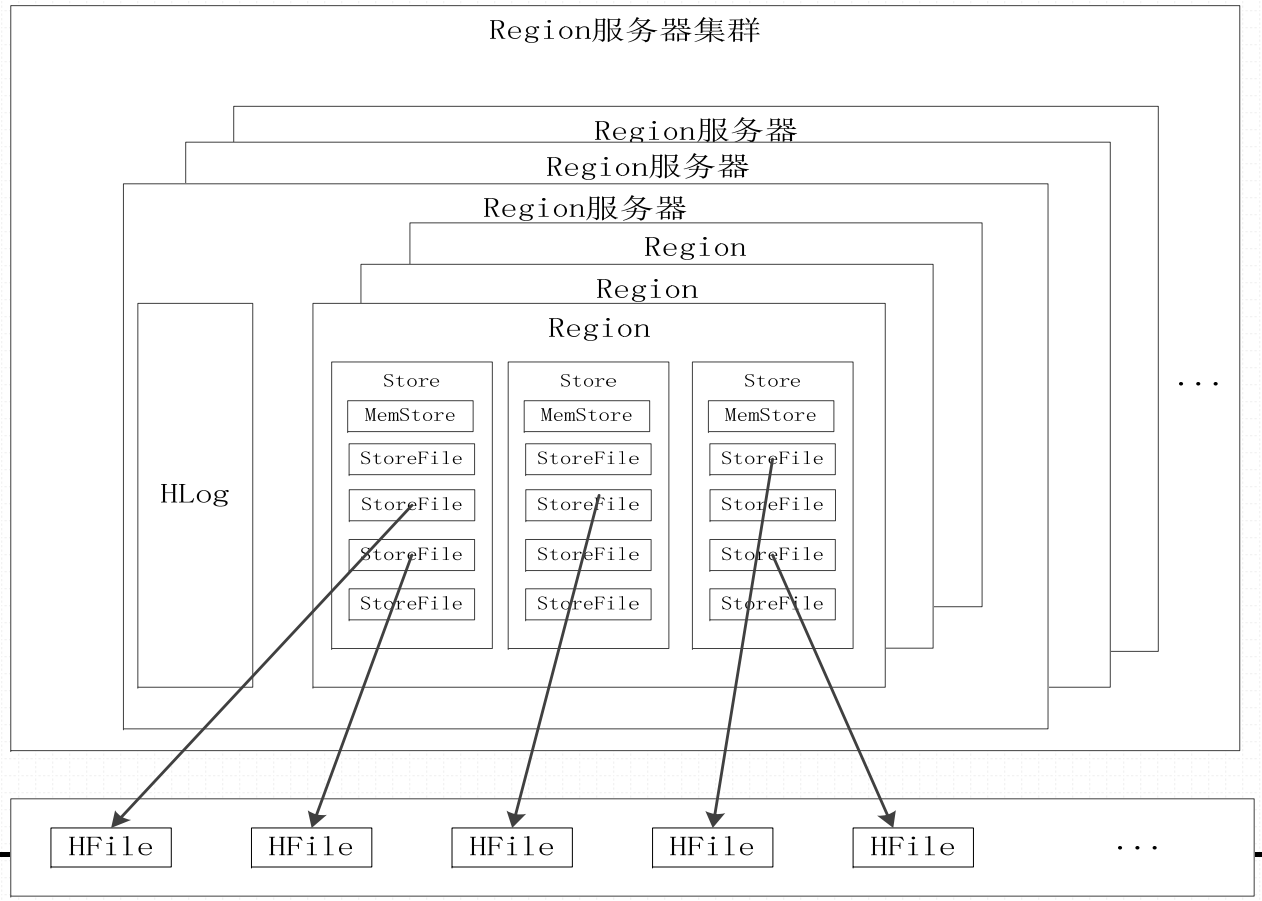
- 管理用户对表的增加、删除、修改、查询等操作
- 实现不同Region服务器之间的负载均衡
- 在Region分裂或合并后，负责重新调整Region的分布
- 对发生故障失效的Region服务器上的Region进行迁移

4. Region服务器

- Region服务器是HBase中最核心的模块，负责维护分配给自己的Region，并响应用户的读写请求
- 负责具体的用户数据的存储和管理，用户对数据的存取都是直接与Region服务器交互

Region服务器工作原理

Region服务器向HDFS文件系统中读写数据



每个Region服务器可以存多个Region，它们共用一个HLog

每个Region中存储了多个列族，每个列族对应一个Store

数据写入时，会先写到MemStore缓存中，缓存满了，再刷写到StoreFile中

StoreFile底层采用HDFS存储

用户读写数据过程

写入数据

Region服务器

用户写入数据时，被分配到相应Region服务器去执行

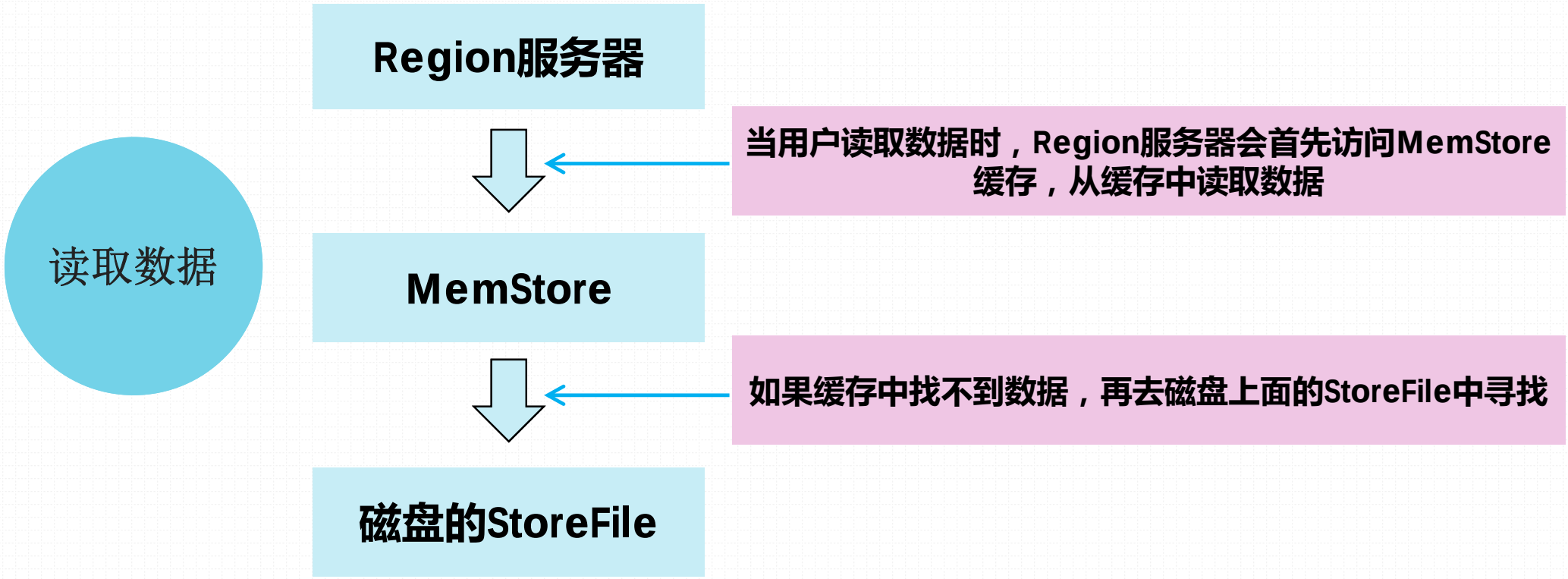
写入MemStore和HLog

用户数据首先被写入到MemStore和Hlog中

调用commit()

只有当操作写入Hlog之后，commit()调用才会将其返回给客户端

用户读写数据过程


Region服务器
读取数据
MemStore
磁盘的StoreFile
当用户读取数据时，Region服务器会首先访问MemStore缓存，从缓存中读取数据
如果缓存中找不到数据，再去磁盘上面的StoreFile中寻找

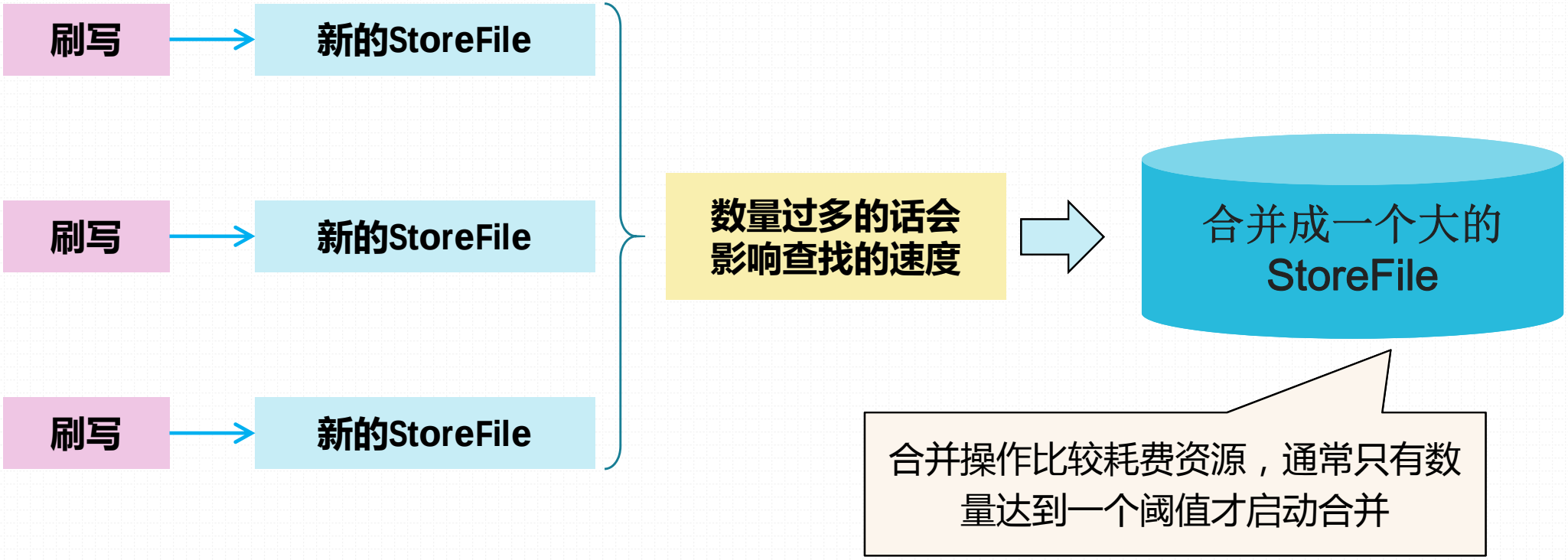
缓存的刷新

系统会周期性地把MemStore缓存里的内容刷写到磁盘的StoreFile文件中，清空缓存，并在Hlog里面写入一个标记

每次刷写都生成一个新的StoreFile文件，因此，每个Store包含多个StoreFile文件

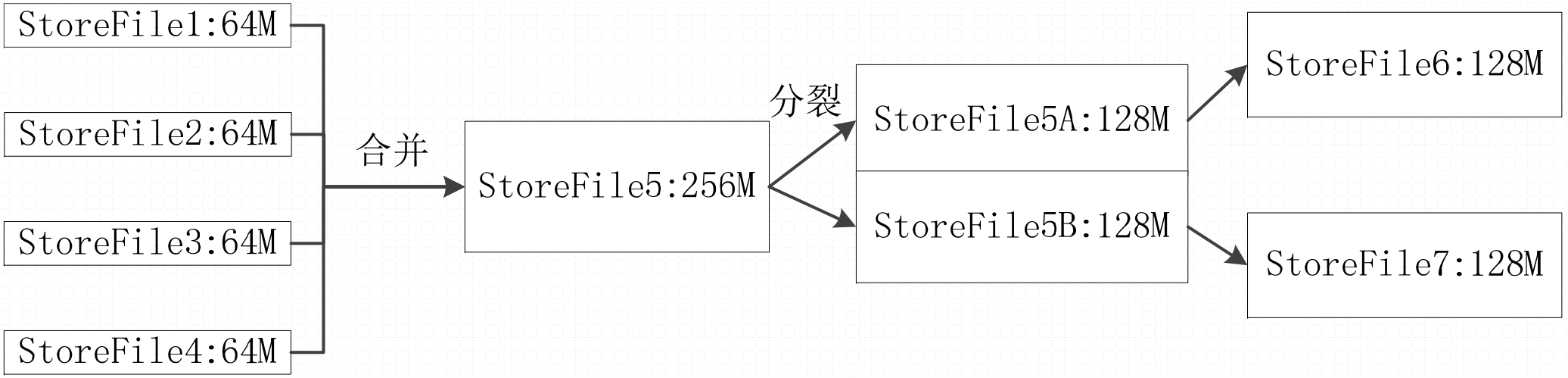
每个Region服务器都有一个自己的HLog 文件，每次启动都检查该文件，确认最近一次执行缓存刷新操作之后是否发生新的写入操作；如果发现更新，则先写入MemStore，再刷写到StoreFile，最后删除旧的Hlog文件，开始为用户提供服务

StoreFile的合并



StoreFile的分裂

- 当合并之后的单个StoreFile过大时，又触发分裂操作，1个父Region被分裂成两个子Region



HLog工作原理

为什么需要HLog？

- 分布式环境必须要考虑系统出错。HBase采用HLog保证系统恢复
- HBase系统为每个Region服务器配置了一个HLog文件，它是一种预写式日志（Write Ahead Log）
- 用户更新数据必须首先写入日志后，才能写入MemStore缓存，并且，直到MemStore缓存内容对应的日志已经写入磁盘，该缓存内容才能被刷写到磁盘

HLog数据恢复原理

- Zookeeper会实时监测每个Region服务器的状态，当某个Region服务器发生故障时，Zookeeper会通知Master
- Master首先会处理该故障Region服务器上面遗留的HLog文件，这个遗留的HLog文件中包含了来自多个Region对象的日志记录
- 系统会根据每条日志记录所属的Region对象对HLog数据进行拆分，分别放到相应Region对象的目录下，然后，再将失效的Region重新分配到可用的Region服务器中，并把与该Region对象相关的HLog日志记录也发送给相应的Region服务器
- Region服务器领取到分配给自己的Region对象以及与之相关的HLog日志记录以后，会重新做一遍日志记录中的各种操作，把日志记录中的数据写入到MemStore缓存中，然后，刷新到磁盘的StoreFile文件中，完成数据恢复

思考：同一个Region服务器上不同的Region为什么要共用HLog？

Hbase

CONTENTS

HBase的安装和配置

下载HBase安装文件 → 解压到/usr/local目录（Linux）

根据HBase与Hadoop版本的对应关系选择相应版本

单机版本 → 直接解压

伪分布式 → 需要进行配置

HBase的bin目录需要添加到系统的Path环境变量中

HBase常用Shell命令

- **create**：创建表
- **list**：列出HBase中所有的表信息
- 例子1：创建一个表，该表名称为tempTable，包含3个列族f1，f2和f3

```
hbase(main):002:0> create 'tempTable', 'f1', 'f2', 'f3'
0 row(s) in 1.3560 seconds

hbase(main):003:0> list
TABLE
tempTable
testTable
wordcount
3 row(s) in 0.0350 seconds
```

HBase常用Shell命令

- **put**：向表、行、列指定的单元格添加数据（一次只能为一个表的一行数据的一个列添加一个数据）
- **scan**：浏览表的相关信息
- 例子2：继续向表tempTable中的第r1行、第“f1:c1”列，添加数据值为“hello,dblab”

```
hbase(main):005:0> put 'tempTable', 'r1', 'f1:c1', 'hello, dblab'
0 row(s) in 0.0240 seconds

hbase(main):006:0> scan 'tempTable'
ROW                         COLUMN+CELL
 r1                         column=f1:c1, timestamp=1430036599391, value=hello, dblab
1 row(s) in 0.0160 seconds
```

在添加数据时，HBase会自动为添加的数据添加一个时间戳，当然，也可以在添加数据时人工指定时间戳的值

HBase常用Shell命令

- **get**：通过表名、行、列、时间戳、时间范围和版本号来获得相应单元格的值
- 例子3：
  - 从tempTable中，获取第r1行、第“f1:c1”列的值
  - 从tempTable中，获取第r1行、第“f1:c3”列的值

```
hbase(main):012:0> get 'tempTable', 'r1', {COLUMN=>'f1:c1'}
COLUMN                          CELL
 f1:c1                          timestamp=1430036599391, value=hello, dblab
1 row(s) in 0.0090 seconds

hbase(main):013:0> get 'tempTable', 'r1', {COLUMN=>'f1:c3'}
COLUMN                          CELL
0 row(s) in 0.0030 seconds
```

从运行结果可以看出： tempTable中第r1行、第“f1:c3”列的值当前不存在

HBase常用Shell命令

- **enable/disable**：使表有效或无效
- **drop**：删除表
- 例子4：使表tempTable无效、删除该表

```
hbase(main):016:0> disable 'tempTable'
0 row(s) in 1.3720 seconds

hbase(main):017:0> drop 'tempTable'
0 row(s) in 1.1350 seconds

hbase(main):018:0> list
TABLE
testTable
wordcount
2 row(s) in 0.0370 seconds
```

HBase常用Java API及应用实例

- HBase是Java编写的，它的原生的API也是Java开发的，不过，可以使用Java或其他语言调用API来访问HBase，使用HBase Java API时，需要先导入相应的JAR包
- **示例任务**：创建一个学生信息表，用来存储学生姓名（姓名作为行键，并且假设姓名不会重复）以及考试成绩，其中，考试成绩是一个列族，分别存储了各个科目的考试成绩。如下图所示：

表1 学生信息表的表结构

name	score		
	English	Math	Computer

表2 需要添加的数据

name	score		
	English	Math	Computer
zhangsan	69	86	77
lisi	55	100	88

重庆大学 CHONGQING UNIVERSITY

HBase常用Java API及应用实例

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.*;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;
import java.io.IOException;
public class Chapter4{
    public static Configuration configuration;
    public static Connection connection;
    public static Admin admin;

    public static void main(String[] args)throws IOException{
        createTable("student",new String[]{"score"});
        insertData("student","zhangsan","score","English","69");
        insertData("student","zhangsan","score","Math","86");
        insertData("student","zhangsan","score","Computer","77");
        getData("student", "zhangsan", "score", "English");
    }
……
public static void init(){……}//建立连接
public static void close(){……}//关闭连接
public static void createTable(){……}//创建表
public static void insertData() {……}//插入数据
public static void getData{……}//浏览数据
}
```

Configuration类

用于对配置信息进行管理

Connection类

用于对数据库连接进行管理

Admin类

用于对数据库进行管理

用于管理对表的创建删除等

HBase常用Java API及应用实例

```
//建立连接
   public static void init(){
      configuration  = HBaseConfiguration.create();
      configuration.set("hbase.rootdir","hdfs://localhost:9000/hbase");
      try{
         connection = ConnectionFactory.createConnection(configuration);
         admin = connection.getAdmin();
      }catch (IOException e){
         e.printStackTrace();
      }
   }
```

备注：hbase-site.xml

```
<configuration>
<property>
<name>hbase.rootdir</name>
<value>hdfs://localhost:9000/hbase</value>
</property>
</configuration>
```

HBase常用Java API及应用实例

```
//关闭连接
    public static void close(){
        try{
            if (admin != null){
                admin.close();
            }
            if (null != connection){
                connection.close();
            }
        }catch (IOException e){
            e.printStackTrace();
        }
    }
```

HBase常用Java API及应用实例

```
/*创建表*/
  /**
   * @param myTableName 表名
   * @param colFamily列族数组
   * @throws Exception
   */
      public static void createTable(String myTableName,String[] colFamily) throws IOException {

      TableName tableName = TableName.valueOf(myTableName);
       if (admin.tableExists(tableName)){
          System.out.println("table exists!");
       }else {
          HTableDescriptor hTableDescriptor = new HTableDescriptor(tableName);
          for (String str: colFamily){
             HColumnDescriptor hColumnDescriptor = new HColumnDescriptor(str);
             hTableDescriptor.addFamily(hColumnDescriptor);
          }
          admin.createTable(hTableDescriptor);
       }
}
```

HBase常用Java API及应用实例

```
    /*添加数据*/
    /**
             * @param tableName 表名
             * @param rowKey 行键
             * @param colFamily 列族
             * @param col 列限定符
             * @param val 数据
             * @throws Exception
             */
        public static void insertData(String tableName, String rowKey, String colFamily, String col, String val)
throws IOException {
            Table table = connection.getTable(TableName.valueOf(tableName));
            Put put = new Put(Bytes.toBytes(rowkey));
            put.addColumn(Bytes.toBytes(colFamily), Bytes.toBytes(col), Bytes.toBytes(val));
            table.put(put);
            table.close();
    }
```

函数调用示例：

```
insertData("student","zhangsan","score","English","69");
insertData("student","zhangsan","score","Math","86");
insertData("student","zhangsan","score","Computer","77");
```

HBase常用Java API及应用实例

```
/*获取某单元格数据*/
/**       * @param tableName 表名
          * @param rowKey 行键
          * @param colFamily 列族
          * @param col 列限定符
         * @throws IOException     */
    public static void getData(String tableName,String rowKey,String colFamily,String col)throws
IOException{
        Table table = connection.getTable(TableName.valueOf(tableName));
        Get get = new Get(Bytes.toBytes(rowkey));
        get.addColumn(Bytes.toBytes(colFamily),Bytes.toBytes(col));
        //获取的result数据是结果集，还需要格式化输出想要的数据才行
        Result result = table.get(get);
        System.out.println(new String(result.getValue(colFamily.getBytes(),col==null?null:col.getBytes())));
        table.close();
    }
```

Thank You!