HBASE 数据库

1. Hbase 基础

1.1 hbase 数据库介绍

1、简介

hbase 是 bigtable 的开源 java 版本。是建立在 hdfs 之上，提供高可靠性、高性能、列存储、可伸缩、实时读写 nosql 的数据库系统。

它介于 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)。

主要用来存储结构化和半结构化的松散数据。

Hbase 查询数据功能很简单，不支持 join 等复杂操作，不支持复杂的事务（行级的事务）

Hbase 中支持的数据类型：byte[]

与 hadoop 一样，Hbase 目标主要依靠横向扩展，通过不断增加廉价的商用服务器，来增加计算和存储能力。

HBase 中的表一般有这样的特点：

- ✧ 大：一个表可以有上十亿行，上百万列
- ✧ 面向列:面向列(族)的存储和权限控制，列(族)独立检索。
- ✧ 稀疏:对于为空(null)的列，并不占用存储空间，因此，表可以设计的非常稀疏。

2、表结构逻辑视图

HBase 以表的形式存储数据。表有行和列组成。列划分为若干个列族(column family)

HBASE表结构：建表时，不需要限定表中的字段，只需要指定若干个列族
插入数据时，列族中可以存储任意多个列（KV，列名&列值）
要查询某一个具体字段的值，需要指定的坐标：表名---->行键---->列族(ColumnFamily)：列名(Qualifier)---->版本

3、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。

Hbase 会对表中的数据按照 rowkey 排序(字典顺序)

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。

4、列族

hbase 表中的每个列，都归属与某个列族。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。

访问控制、磁盘和内存的使用统计都是在列族层面进行的。

列族越多，在取一行数据时所要参与 IO、搜寻的文件就越多，所以，如果没有必要，不要设置太多的列族

5、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：

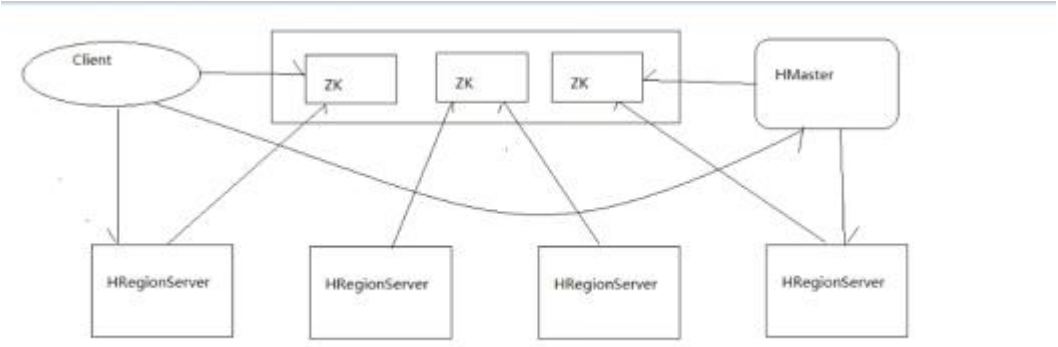
- ✧ 保存数据的最后 n 个版本
- ✧ 保存最近一段时间内的版本（设置数据的生命周期 TTL）。

用户可以针对每个列族进行设置。

6、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。

1.2 hbase 集群结构



REGION：是 HBASE 中对表进行切割的单元

HMASTER：HBASE 的主节点，负责整个集群的状态感知，负载分配、负责用户表的元数据管理
（可以配置多个用来实现 HA）
REGION-SERVER：HBASE 中真正负责管理 region 的服务器，也就是负责为客户端进行表数据读写的服务器

ZOOKEEPER:整个 HBASE 中的主从节点协调，主节点之间的选举，集群节点之间的上下线感知……都是通过 zookeeper 来实现

1.3 hbase 集群搭建

----先部署一个 zookeeper 集群

（1）上传 hbase 安装包

（2）解压

（3）配置 hbase 集群，要修改 3 个文件

注意：要把 hadoop 的 hdfs-site.xml 和 core-site.xml 放到 hbase/conf 下

（3.1）修改 hbase-env.sh

```
export JAVA_HOME=/usr/java/jdk1.7.0_55
//告诉 hbase 使用外部的 zk
export HBASE_MANAGES_ZK=false
```

（3.2）修改 hbase-site.xml

```
<configuration>
        <!-- 指定 hbase 在 HDFS 上存储的路径 -->
        <property>
                <name>hbase.rootdir</name>
                <value>hdfs://ns1/hbase</value>
        </property>
        <!-- 指定 hbase 是分布式的 -->
        <property>
                <name>hbase.cluster.distributed</name>
                <value>true</value>
        </property>
        <!-- 指定 zk 的地址，多个用“,”分割 -->
        <property>
                <name>hbase.zookeeper.quorum</name>
                <value>weekend05:2181,weekend06:2181,weekend07:2181</value>
        </property>
</configuration>
```

（3.3）修改 regionservers

```
weekend03
weekend04
weekend05
weekend06
```

(3.3) 修改 backup-masters 来指定备用的主节点【不需要】

```
[root@mini1 conf]# vi backup-masters
mini2
```

（3.4）拷贝 hbase 到其他节点

```
scp -r /weekend/hbase-0.96.2-hadoop2/ weekend02:/weekend/
scp -r /weekend/hbase-0.96.2-hadoop2/ weekend03:/weekend/
scp -r /weekend/hbase-0.96.2-hadoop2/ weekend04:/weekend/
scp -r /weekend/hbase-0.96.2-hadoop2/ weekend05:/weekend/
scp -r /weekend/hbase-0.96.2-hadoop2/ weekend06:/weekend/
```

（4）将配置好的 HBase 拷贝到每一个节点并同步时间。

（5）启动所有的 hbase 进程

首先启动 zk 集群

```
./zkServer.sh start
```

启动 hdfs 集群

```
start-dfs.sh
```

启动 hbase，在主节点上运行：

```
start-hbase.sh
```

（6）通过浏览器访问 hbase 管理页面

192.168.1.201:60010 /16010

（7）为保证集群的可靠性，要启动多个 HMaster

```
hbase-daemon.sh start master
```

1.4 命令行演示

1.4.1 基本 shell 命令

进入 hbase 命令行

```
./hbase shell
```

显示 hbase 中的表

```
list
```

创建 user 表，包含 info、data 两个列族

```
create 'user', 'info1', 'data1'
create 'user', {NAME => 'info', VERSIONS => '3'}
```

向 user 表中插入信息，row key 为 rk0001，列族 info 中添加 name 列标示符，值为 zhangsan

```
put 'user', 'rk0001', 'info:name', 'zhangsan'
```

向 user 表中插入信息，row key 为 rk0001，列族 info 中添加 gender 列标示符，值为 female

```
put 'user', 'rk0001', 'info:gender', 'female'
```

向 user 表中插入信息，row key 为 rk0001，列族 info 中添加 age 列标示符，值为 20

```
put 'user', 'rk0001', 'info:age', 20
```

向 user 表中插入信息，row key 为 rk0001，列族 data 中添加 pic 列标示符，值为 picture
put 'user', 'rk0001', 'data:pic', 'picture'

获取 user 表中 row key 为 rk0001 的所有信息
get 'user', 'rk0001'

获取 user 表中 row key 为 rk0001，info 列族的所有信息
get 'user', 'rk0001', 'info'

获取 user 表中 row key 为 rk0001，info 列族的 name、age 列标示符的信息
get 'user', 'rk0001', 'info:name', 'info:age'

获取 user 表中 row key 为 rk0001，info、data 列族的信息
get 'user', 'rk0001', 'info', 'data'
get 'user', 'rk0001', {COLUMN => ['info', 'data']}

get 'user', 'rk0001', {COLUMN => ['info:name', 'data:pic']}

获取 user 表中 row key 为 rk0001，列族为 info，版本号最新 5 个的信息
get 'user', 'rk0001', {COLUMN => 'info', VERSIONS => 2}
get 'user', 'rk0001', {COLUMN => 'info:name', VERSIONS => 5}
get 'user', 'rk0001', {COLUMN => 'info:name', VERSIONS => 5, TIMERANGE => [1392368783980, 1392380169184]}

获取 user 表中 row key 为 rk0001，cell 的值为 zhangsan 的信息
get 'people', 'rk0001', {FILTER => "ValueFilter(=, 'binary:图片')"}

获取 user 表中 row key 为 rk0001，列标示符中含有 a 的信息
get 'people', 'rk0001', {FILTER => "(QualifierFilter(=,'substring:a'))"}

put 'user', 'rk0002', 'info:name', 'fanbingbing'
put 'user', 'rk0002', 'info:gender', 'female'
put 'user', 'rk0002', 'info:nationality', '中国'
get 'user', 'rk0002', {FILTER => "ValueFilter(=, 'binary:中国')"}

查询 user 表中的所有信息
scan 'user'

查询 user 表中列族为 info 的信息
scan 'user', {COLUMNS => 'info'}
scan 'user', {COLUMNS => 'info', RAW => true, VERSIONS => 5}
scan 'persion', {COLUMNS => 'info', RAW => true, VERSIONS => 3}
查询 user 表中列族为 info 和 data 的信息

scan 'user', {COLUMNS => ['info', 'data']}
scan 'user', {COLUMNS => ['info:name', 'data:pic']}

查询 user 表中列族为 info、列标示符为 name 的信息
scan 'user', {COLUMNS => 'info:name'}

查询 user 表中列族为 info、列标示符为 name 的信息,并且版本最新的 5 个
scan 'user', {COLUMNS => 'info:name', VERSIONS => 5}

查询 user 表中列族为 info 和 data 且列标示符中含有 a 字符的信息
scan 'user', {COLUMNS => ['info', 'data'], FILTER => "(QualifierFilter(=,'substring:a'))"}

查询 user 表中列族为 info，rk 范围是[rk0001, rk0003)的数据
scan 'people', {COLUMNS => 'info', STARTROW => 'rk0001', ENDROW => 'rk0003'}

查询 user 表中 row key 以 rk 字符开头的
scan 'user',{FILTER=>"PrefixFilter('rk')"}

查询 user 表中指定范围的数据
scan 'user', {TIMERANGE => [1392368783980, 1392380169184]}

删除数据
删除 user 表 row key 为 rk0001，列标示符为 info:name 的数据
delete 'people', 'rk0001', 'info:name'
删除 user 表 row key 为 rk0001，列标示符为 info:name，timestamp 为 1392383705316 的数据
delete 'user', 'rk0001', 'info:name', 1392383705316

清空 user 表中的数据
truncate 'people'

修改表结构
首先停用 user 表（新版本不用）
disable 'user'

添加两个列族 f1 和 f2
alter 'people', NAME => 'f1'
alter 'user', NAME => 'f2'
启用表
enable 'user'

```
###disable 'user'(新版本不用)
删除一个列族：
alter 'user', NAME => 'f1', METHOD => 'delete'  或  alter 'user', 'delete' => 'f1'

添加列族 f1 同时删除列族 f2
alter 'user', {NAME => 'f1'}, {NAME => 'f2', METHOD => 'delete'}

将 user 表的 f1 列族版本号改为 5
alter 'people', NAME => 'info', VERSIONS => 5
启用表
enable 'user'


删除表
disable 'user'
drop 'user'


get 'person', 'rk0001', {FILTER => "ValueFilter(=, 'binary:中国')"}
get 'person', 'rk0001', {FILTER => "(QualifierFilter(=,'substring:a'))"}
scan 'person', {COLUMNS => 'info:name'}
scan 'person', {COLUMNS => ['info', 'data'], FILTER => "(QualifierFilter(=,'substring:a'))"}
scan 'person', {COLUMNS => 'info', STARTROW => 'rk0001', ENDROW => 'rk0003'}

scan 'person', {COLUMNS => 'info', STARTROW => '20140201', ENDROW => '20140301'}
scan 'person', {COLUMNS => 'info:name', TIMERANGE => [1395978233636, 1395987769587]}
delete 'person', 'rk0001', 'info:name'

alter 'person', NAME => 'ffff'
alter 'person', NAME => 'info', VERSIONS => 10


get 'user', 'rk0002', {COLUMN => ['info:name', 'data:pic']}
```

1.5 hbase 代码开发（基本，过滤器查询）

1.5.1 基本增删改查 java 实现

```
public class HbaseDemo {

    private Configuration conf = null;
```

```
@Before
public void init(){
	conf = HBaseConfiguration.create();
	conf.set("hbase.zookeeper.quorum", "weekend05,weekend06,weekend07");
}

@Test
public void testDrop() throws Exception{
	HBaseAdmin admin = new HBaseAdmin(conf);
	admin.disableTable("account");
	admin.deleteTable("account");
	admin.close();
}

@Test
public void testPut() throws Exception{
	HTable table = new HTable(conf, "person_info");
	Put p = new Put(Bytes.toBytes("person_rk_bj_zhang_000002"));
	p.add("base_info".getBytes(), "name".getBytes(), "zhangwuji".getBytes());
	table.put(p);
	table.close();
}


@Test
public void testDel() throws Exception{
	HTable table = new HTable(conf, "user");
	Delete del = new Delete(Bytes.toBytes("rk0001"));
	del.deleteColumn(Bytes.toBytes("data"), Bytes.toBytes("pic"));
	table.delete(del);
	table.close();
}

@Test
public void testGet() throws Exception{
	HTable table = new HTable(conf, "person_info");
	Get get = new Get(Bytes.toBytes("person_rk_bj_zhang_000001"));
	get.setMaxVersions(5);
	Result result = table.get(get);

	List<Cell> cells = result.listCells();

	for(Cell c:cells){
	}
```

```
        //result.getValue(family, qualifier);  可以从 result 中直接取出一个特定的
value

        //遍历出 result 中所有的键值对
        List<KeyValue> kvs = result.list();
        //kv   ---> f1:title:superise....        f1:author:zhangsan      f1:content:asdfasldgkjsldg
        for(KeyValue kv : kvs){
            String family = new String(kv.getFamily());
            System.out.println(family);
            String qualifier = new String(kv.getQualifier());
            System.out.println(qualifier);
            System.out.println(new String(kv.getValue()));

        }
        table.close();
    }
```

1.5.2 过滤器查询

引言：过滤器的类型很多，但是可以分为两大类——比较过滤器，专用过滤器

过滤器的作用是在服务端判断数据是否满足条件，然后只将满足条件的数据返回给客户端；

hbase 过滤器的比较运算符：

```
LESS  <
LESS_OR_EQUAL <=
EQUAL =
NOT_EQUAL <>
GREATER_OR_EQUAL >=
GREATER >
NO_OP 排除所有
```

Hbase 过滤器的比较器（指定比较机制）：

```
BinaryComparator  按字节索引顺序比较指定字节数组，采用 Bytes.compareTo(byte[])
BinaryPrefixComparator 跟前面相同，只是比较左端的数据是否相同
NullComparator 判断给定的是否为空
BitComparator 按位比较
RegexStringComparator 提供一个正则的比较器，仅支持 EQUAL 和非 EQUAL
SubstringComparator 判断提供的子串是否出现在 value 中。
```

Hbase 的过滤器分类

- 比较过滤器

1.1 行键过滤器 RowFilter

```
Filter filter1 = new RowFilter(CompareOp.LESS_OR_EQUAL, new BinaryComparator(Bytes.toBytes("row-22")));
scan.setFilter(filter1);
```

1.2 列族过滤器 FamilyFilter

```
Filter filter1 = new FamilyFilter(CompareFilter.CompareOp.LESS, new BinaryComparator(Bytes.toBytes("colfam3")));
scan.setFilter(filter1);
```

1.3 列过滤器 QualifierFilter

```
filter = new QualifierFilter(CompareFilter.CompareOp.LESS_OR_EQUAL, new BinaryComparator(Bytes.toBytes("col-2")));
scan.setFilter(filter1);
```

1.4 值过滤器 ValueFilter

```
Filter filter = new ValueFilter(CompareFilter.CompareOp.EQUAL, new SubstringComparator(".4") );
scan.setFilter(filter1);
```

- 专用过滤器

2.1 单列值过滤器 SingleColumnValueFilter ----会返回满足条件的整行

```
SingleColumnValueFilter filter = new SingleColumnValueFilter(
        Bytes.toBytes("colfam1"),
        Bytes.toBytes("col-5"),
        CompareFilter.CompareOp.NOT_EQUAL,
        new SubstringComparator("val-5"));
filter.setFilterIfMissing(true);  //如果不设置为 true，则那些不包含指定 column 的行也会返回
scan.setFilter(filter1);
```

2.2 SingleColumnValueExcludeFilter

与上相反

2.3 前缀过滤器 PrefixFilter----针对行键

```
Filter filter = new PrefixFilter(Bytes.toBytes("row1"));
scan.setFilter(filter1);
```

2.4 列前缀过滤器 ColumnPrefixFilter

```
Filter filter = new ColumnPrefixFilter(Bytes.toBytes("qual2"));
scan.setFilter(filter1);
```

2.4 分页过滤器 PageFilter

```
    public static void main(String[] args) throws Exception {
        Configuration conf = HBaseConfiguration.create();
        conf.set("hbase.zookeeper.quorum", "spark01:2181,spark02:2181,spark03:2181");

        String tableName = "testfilter";
        String cfName = "f1";
```

```
        final byte[] POSTFIX = new byte[] { 0x00 };
        HTable table = new HTable(conf, tableName);
        Filter filter = new PageFilter(3);
        byte[] lastRow = null;
        int totalRows = 0;
        while (true) {
            Scan scan = new Scan();
            scan.setFilter(filter);
            if(lastRow != null){
                //注意这里添加了 POSTFIX 操作，用来重置扫描边界
                byte[] startRow = Bytes.add(lastRow,POSTFIX);
                scan.setStartRow(startRow);
            }
            ResultScanner scanner = table.getScanner(scan);
            int localRows = 0;
            Result result;
            while((result = scanner.next()) != null){
                System.out.println(localRows++ + ":" + result);
                totalRows ++;
                lastRow = result.getRow();
            }
            scanner.close();
            if(localRows == 0) break;
        }
        System.out.println("total rows:" + totalRows);
    }
```

```
    /**
     * 多种过滤条件的使用方法
     * @throws Exception
     */
    @Test
    public void testScan() throws Exception{
        HTable table = new HTable(conf, "person_info".getBytes());
        Scan scan = new Scan(Bytes.toBytes("person_rk_bj_zhang_000001"),
Bytes.toBytes("person_rk_bj_zhang_000002"));

        //前缀过滤器----针对行键
        Filter filter = new PrefixFilter(Bytes.toBytes("rk"));
```

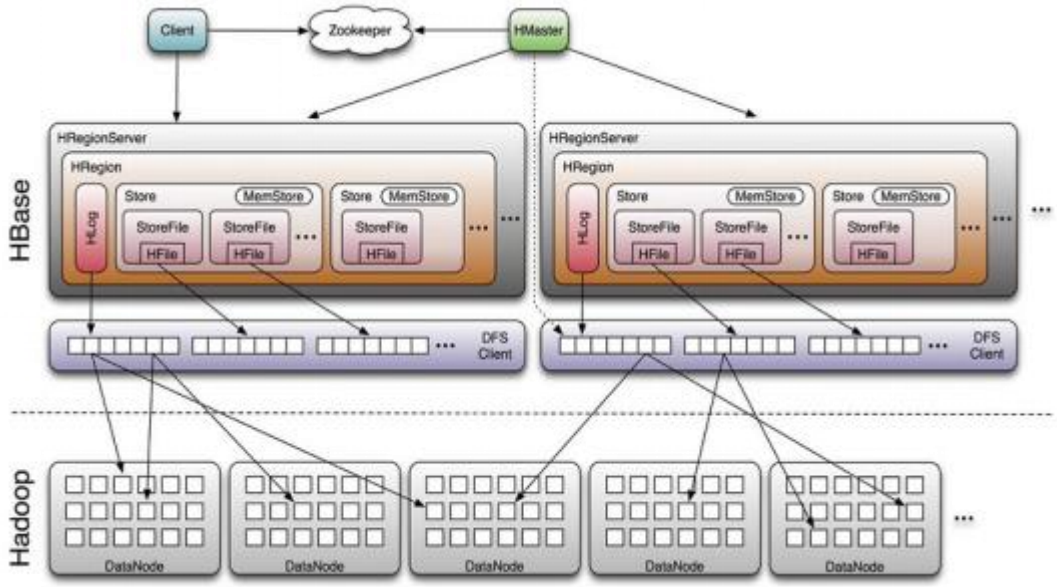
```
        //行过滤器  ---针对行键
        ByteArrayComparable rowComparator = new BinaryComparator(Bytes.toBytes("person_rk_bj_zhang_000001"));
        RowFilter rf = new RowFilter(CompareOp.LESS_OR_EQUAL, rowComparator);

        /**
         * 假设 rowkey 格式为：创建日期_发布日期_ID_TITLE
         * 目标：查找  发布日期  为  2014-12-21  的数据
         *
sc.textFile("path").flatMap(line=>line.split("\t")).map(x=>(x,1)).reduceByKey(_+_).map((_(2),_(1))).sortByKey().map((_(2),_(1))).saveAsTextFile("")
         *
         *
         */
        rf = new RowFilter(CompareOp.EQUAL , new SubstringComparator("_2014-12-21_"));


        //单值过滤器 1 完整匹配字节数组
        new SingleColumnValueFilter("base_info".getBytes(), "name".getBytes(), CompareOp.EQUAL, "zhangsan".getBytes());
        //单值过滤器 2 匹配正则表达式
        ByteArrayComparable comparator = new RegexStringComparator("zhang.");
        new SingleColumnValueFilter("info".getBytes(), "NAME".getBytes(), CompareOp.EQUAL, comparator);

        //单值过滤器 3 匹配是否包含子串,大小写不敏感
        comparator = new SubstringComparator("wu");
        new SingleColumnValueFilter("info".getBytes(), "NAME".getBytes(), CompareOp.EQUAL, comparator);

        //键值对元数据过滤-----family 过滤----字节数组完整匹配
        FamilyFilter ff = new FamilyFilter(
                CompareOp.EQUAL ,
                new BinaryComparator(Bytes.toBytes("base_info"))    //表中不存在 inf 列族，过滤结果为空
                );
        //键值对元数据过滤-----family 过滤----字节数组前缀匹配
        ff = new FamilyFilter(
                CompareOp.EQUAL ,
                new BinaryPrefixComparator(Bytes.toBytes("inf"))    //表中存在以 inf 打头的列族 info，过滤结果为该列族所有行
                );
```

```
        //键值对元数据过滤-----qualifier 过滤----字节数组完整匹配

        filter = new QualifierFilter(
                CompareOp.EQUAL ,
                new BinaryComparator(Bytes.toBytes("na"))   //表中不存在 na 列，过滤结果为空
                );
        filter = new QualifierFilter(
                CompareOp.EQUAL ,
                new BinaryPrefixComparator(Bytes.toBytes("na"))   //表中存在以 na 打头的列 name，过滤结果为所有行的该列数据
                );

        //基于列名(即 Qualifier)前缀过滤数据的 ColumnPrefixFilter
        filter = new ColumnPrefixFilter("na".getBytes());

        //基于列名(即 Qualifier)多个前缀过滤数据的 MultipleColumnPrefixFilter
        byte[][] prefixes = new byte[][] {Bytes.toBytes("na"), Bytes.toBytes("me")};
        filter = new MultipleColumnPrefixFilter(prefixes);

        //为查询设置过滤条件
        scan.setFilter(filter);



        scan.addFamily(Bytes.toBytes("base_info"));
        //一行
//      Result result = table.get(get);
        //多行的数据
        ResultScanner scanner = table.getScanner(scan);
        for(Result r : scanner){
            /**
            for(KeyValue kv : r.list()){
                String family = new String(kv.getFamily());
                System.out.println(family);
                String qualifier = new String(kv.getQualifier());
                System.out.println(qualifier);
                System.out.println(new String(kv.getValue()));
            }
            */
            //直接从 result 中取到某个特定的 value
            byte[] value = r.getValue(Bytes.toBytes("base_info"), Bytes.toBytes("name"));
            System.out.println(new String(value));
```

```
        }
        table.close();
    }
```

1.6 hbase 内部原理

1.6.1 系统架构



Client

1 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----root 表在哪台服务器上。

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 职责

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 HDFS 上的垃圾文件回收

5 处理 schema 更新请求

Region Server 职责

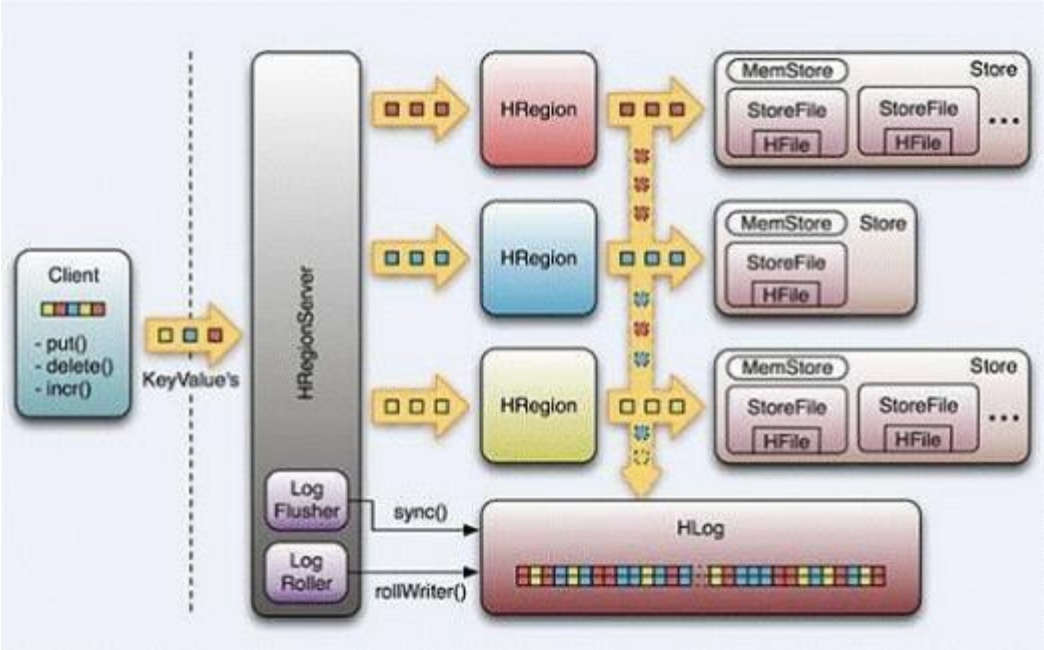
1 Region server 维护 Master 分配给它的 region，处理对这些 region 的 IO 请求

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。

1.6.2 物理存储

1、整体结构



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。

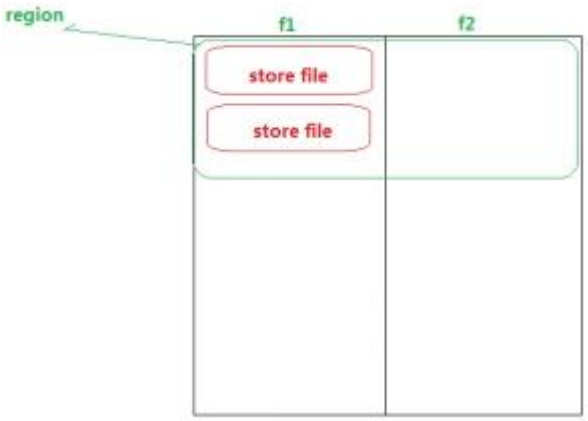
3 region 按大小分割的(默认 10G)，每个表一开始只有一个 region，随着数据不断插入表，region 不断增大，当增大到一个阀值的时候，Hregion 就会等分会两个新的 Hregion。当 table 中的行不断增多，就会有越来越多的 Hregion。

4 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。

5 HRegion 虽然是负载均衡的最小单元，但并不是物理存储的最小单元。

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

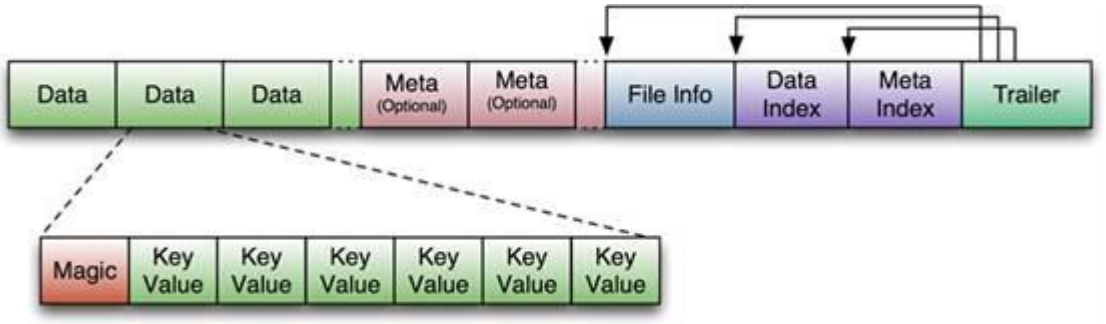
每个 Strore 又由一个 memStore 和 0 至多个 StoreFile 组成。如上图



2、STORE FILE & HFILE 结构

StoreFile 以 HFile 格式保存在 HDFS 上。

附：HFile 的格式为：



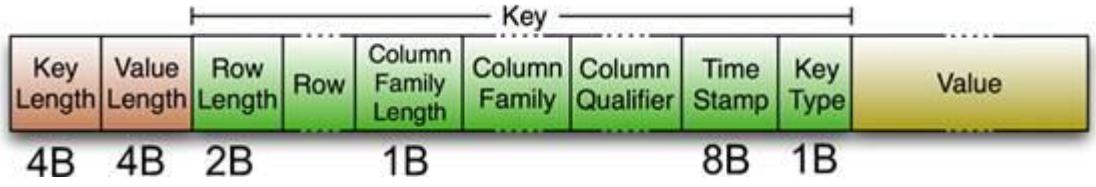
首先 HFile 文件是不定长的，长度固定的只有其中的两块：Trailer 和 FileInfo。正如图中所示的，Trailer 中有指针指向其他数 据块的起始点。

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。但是这个 byte 数组里面包含了很多项，并且有固定的结构。我们来看看里面的具体结构：



开始是两个固定长度的数值，分别表示 Key 的长度和 Value 的长度。紧接着是 Key，开始是固定长度的数值，表示 RowKey 的长度，紧接着是 RowKey，然后是固定长度的数值，表示 Family 的长度，然后是 Family，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了。

HFile 分为六个部分：
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 – 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。
HFile 的 Data Block，Meta Block 通常采用压缩方式存储，压缩之后可以大大减少网络 IO 和磁盘 IO，随之而来的开销当然是需要花费 cpu 进行压缩和解压缩。
目标 Hfile 的压缩支持两种方式：Gzip，Lzo。

3、Memstore 与 storefile

一个 region 由多个 store 组成，每个 store 包含一个列族的所有数据
Store 包括位于内存的 memstore 和位于硬盘的 storefile
写操作先写入 memstore,当 memstore 中的数据量达到某个阈值，Hregionserver 启动 flashcache 进程写入 storefile,每次写入形成单独一个 storefile
当 storefile 大小超过一定阈值后，会把当前的 region 分割成两个，并由 Hmaster 分配给相应的 region 服务器，实现负载均衡
客户端检索数据时，先在 memstore 找，找不到再找 storefile

4、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)

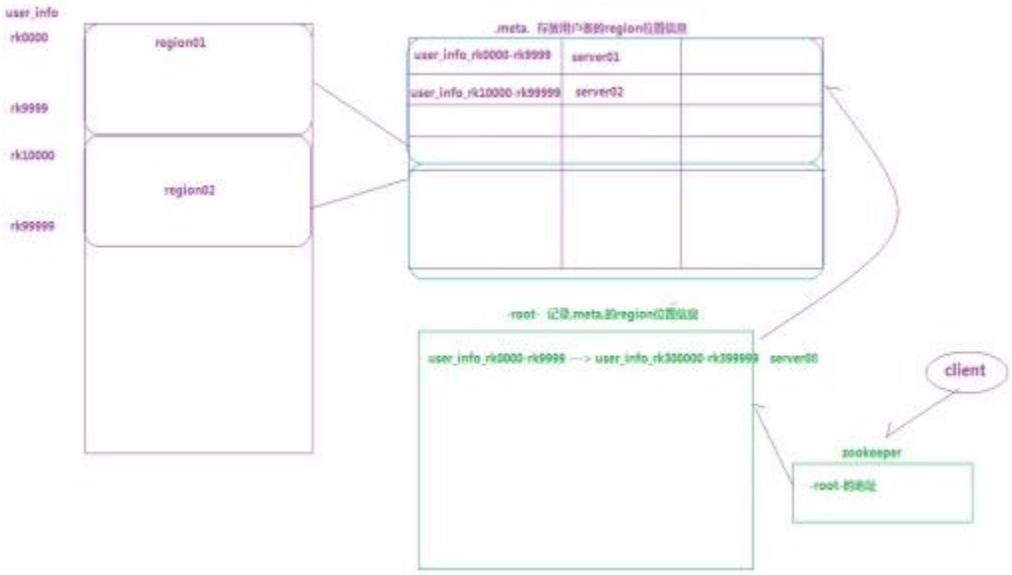
的日志会混在一起，这样做的目的是不断追加单个文件相对于同时写多个文件而言，可以减少磁盘寻址次数，因此可以提高对 table 的写性能。带来的麻烦是，如果一台 region server 下线，为了恢复其上的 region，需要将 region server 上的 log 进行拆分，然后分发到其它 region server 上进行恢复。

HLog 文件就是一个普通的 Hadoop Sequence File：

- ✧ HLog 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，可参见上文描述。

1.6.3 寻址机制

1、寻址示意图



2、-ROOT-和.META.表结构

RowKey	info			historian
	regioninfo	server	serverstartcode	
TableName,StartKey,TimeStamp	StartKey, EndKey, Family List (Family,BloomFilter,Compress,TTL,InMemory,BlockSize,BlockCache)	address		

.META.行记录结构

RowKey	info			historian
	regioninfo	server	server startcode	
Table1,RK0,12345678		RS1		
Table1, RK10000,12345687		RS2		
Table1, RK20000,12346578		RS3		
……	……	……	……	……
Table2, RK0,12345678		RS1		
Table2, RK30000,12348765		RS2		

3、寻址流程

现在假设我们要从 Table2 里面插寻一条 RowKey 是 RK10000 的数据。那么我们应该遵循以下步骤：

1. 从.META.表里面查询哪个 Region 包含这条数据。
2. 获取管理这个 Region 的 RegionServer 地址。
3. 连接这个 RegionServer, 查到这条数据。

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

说明：
1 root region 永远不会被 split，保证了最需要三次跳转，就能定位到任意 region 。
2.META.表每行保存一个 region 的位置信息，row key 采用表名+表的最后一行编码而成。
3 为了加快访问，.META.表的全部 region 都保存在内存中。
4 client 会将查询过的位置信息保存缓存起来，缓存不会主动失效，因此如果

client 上的缓存全部失效，则需要进行最多 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)。

1.6.4 读写过程

1、读请求过程：

1 客户端通过 zookeeper 以及 root 表和 meta 表找到目标数据所在的 regionserver
2 联系 regionserver 查询目标数据
3 regionserver 定位到目标数据所在的 region，发出查询请求
4 region 先在 memstore 中查找，命中则返回
5 如果在 memstore 中找不到，则在 storefile 中扫描（可能会扫描到很多的 storefile----bloomfilter）

2、写请求过程：

1 client 向 region server 提交写请求
2 region server 找到目标 region
3 region 检查数据是否与 schema 一致
4 如果客户端没有指定版本，则获取当前系统时间作为数据版本
5 将更新写入 WAL log
6 将更新写入 Memstore
7 判断 Memstore 的是否需要 flush 为 Store 文件。

细节描述：
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。
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。

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(minor_compact, 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，compact 时，需要访问 Store 中全部的 StoreFile 和 MemStore，将他们按 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。

1.6.5 Region 管理

(1) region 分配
任何时刻，一个 region 只能分配给一个 region server。master 记录了当前有哪些可用的 region server。以及当前哪些 region 分配给了哪些 region server，哪些 region 还没有分配。当需要分配的新的 region，并且有一个 region server 上有可用空间时，master 就给这个 region server 发送一个装载请求，把 region 分配给这个 region server。region server 得到请求后，就开始对此 region 提供服务。

(2) region server 上线
master 使用 zookeeper 来跟踪 region server 状态。当某个 region server 启动时，会首先在 zookeeper 上的 server 目录下建立代表自己的 znode。由于 master 订阅了 server 目录上的变更消息，当 server 目录下的文件出现新增或删除操作时，master 可以得到来自 zookeeper 的实时通知。因此一旦 region server 上线，master 能马上得到消息。

(3) region server 下线
当 region server 下线时，它和 zookeeper 的会话断开，zookeeper 而自动释放代表这台 server 的文件上的独占锁。master 就可以确定：
1 region server 和 zookeeper 之间的网络断开了。
2 region server 挂了。
无论哪种情况，region server 都无法继续为它的 region 提供服务了，此时 master 会删除 server 目录下代表这台 region server 的 znode 数据，并将这台 region server 的 region 分配给其它还活着的同志。

1.6.6 Master 工作机制

- master 上线

master 启动进行以下步骤:
1 从 zookeeper 上获取唯一一个代表 active master 的锁，用来阻止其它 master 成为 master。
2 扫描 zookeeper 上的 server 父节点，获得当前可用的 region server 列表。
3 和每个 region server 通信，获得当前已分配的 region 和 region server 的对应关系。
4 扫描.META.region 的集合，计算得到当前还未分配的 region，将他们放入待分配 region 列表。

➢ master 下线

由于 master 只维护表和 region 的元数据，而不参与表数据 IO 的过程，master 下线仅导致所有元数据的修改被冻结(无法创建删除表，无法修改表的 schema，无法进行 region 的负载均衡，无法处理 region 上下线，无法进行 region 的合并，唯一例外的是 region 的 split 可以正常进行，因为只有 region server 参与)，表的数据读写还可以正常进行。因此 master 下线短时间内对整个 hbase 集群没有影响。

从上线过程可以看到，master 保存的信息全是可以冗余信息（都可以从系统其它地方收集到或者计算出来）

因此,一般hbase集群中总是有一个master在提供服务,还有一个以上的‘master’在等待时机抢占它的位置。

动手练习（增删改查）

2. Hbase 高级应用

2.1 建表高级属性

下面几个 shell 命令在 hbase 操作中可以起到很到的作用，且主要体现在建表的过程中，看下面几个 create 属性

1、BLOOMFILTER 默认是 NONE 是否使用布隆过虑及使用何种方式

布隆过滤可以每列族单独启用。

使用 HColumnDescriptor.setBloomFilterType(NONE | ROW | ROWCOL) 对列族单独启用布隆。

- ✧ Default = ROW 对行进行布隆过滤。
- ✧ 对 ROW，行键的哈希在每次插入行时将被添加到布隆。
- ✧ 对 ROWCOL，行键 + 列族 + 列族修饰的哈希将在每次插入行时添加到布隆

使用方法: create 'table',{BLOOMFILTER =>'ROW'}

启用布隆过滤可以节省读磁盘过程，可以有助于降低读取延迟

2、VERSIONS 默认是 1 这个参数的意思是数据保留 1 个 版本，如果我们认为我们的数据没有这么大的必要保留这么多，随时都在更新，而老版本的数据对我们毫无价值，那将此参数设为 1 能节约 2/3 的空间

使用方法: create 'table',{VERSIONS=>'2'}

附：MIN_VERSIONS => '0'是说在 compact 操作执行之后，至少要保留的版本

3、COMPRESSION 默认值是 NONE 即不使用压缩

这个参数意思是该列族是否采用压缩，采用什么压缩算法

使用方法: create 'table',{NAME=>'info',COMPRESSION=>'SNAPPY'}

建议采用 SNAPPY 压缩算法

HBase 中，在 Snappy 发布之前（Google 2011 年对外发布 Snappy），采用的 LZO 算法，目标是达到尽可能快的压缩和解压速度，同时减少对 CPU 的消耗；

在 Snappy 发布之后，建议采用 Snappy 算法（参考《HBase: The Definitive Guide》),具体可以根据实际情况对 LZO 和 Snappy 做过更详细的对比测试后再做选择。

Algorithm	% remaining	Encoding	Decoding
GZIP	13.4%	21 MB/s	118 MB/s
LZO	20.5%	135 MB/s	410 MB/s
Zippy/Snappy	22.2%	172 MB/s	409 MB/s

如果建表之初没有压缩，后来想要加入压缩算法，可以通过 alter 修改 schema

4、alter

使用方法：

如 修改压缩算法

```
disable 'table'
alter 'table',{NAME=>'info',COMPRESSION=>'snappy'}
enable 'table'
```

但是需要执行 major_compact 'table' 命令之后 才会做实际的操作。

5、TTL

默认是 2147483647 即:Integer.MAX_VALUE 值大概是 68 年

这个参数是说明该列族数据的存活时间，单位是 s

这个参数可以根据具体的需求对数据设定存活时间,超过存过时间的数据将在表中不在显示，待下次 major compact 的时候再彻底删除数据

注意的是 TTL 设定之后 MIN_VERSIONS=>'0' 这样设置之后,TTL 时间戳过期后，将全部彻底删除该 family 下所有的数据，如果 MIN_VERSIONS 不等于 0 那将保留最新的 MIN_VERSIONS 个版本的数据，其它的全部删除，比如 MIN_VERSIONS=>'1' 届时将保留一个最新版本的数据,其它版本的数据将不再保存。

6、describe 'table' 这个命令查看了 create table 的各项参数或者是默认值。

7、disable_all 'toplist.*' disable_all 支持正则表达式，并列出当前匹配的表的如下：

```
toplist_a_total_1001
toplist_a_total_1002
toplist_a_total_1008
```

toplist_a_total_1009
toplist_a_total_1019
toplist_a_total_1035
...
Disable the above 25 tables (y/n)? 并给出确认提示

8、drop_all 这个命令和 disable_all 的使用方式是一样的

9、hbase 表预分区----手动分区

默认情况下，在创建 HBase 表的时候会自动创建一个 region 分区，当导入数据的时候，所有的 HBase 客户端都向这一个 region 写数据，直到这个 region 足够大了才进行切分。一种可以加快批量写入速度的方法是通过预先创建一些空的 regions，这样当数据写入 HBase 时，会按照 region 分区情况，在集群内做数据的负载均衡。

命令方式:

```
create 't1', 'f1', {NUMREGIONS => 15, SPLITALGO => 'HexStringSplit'}
```

也可以使用 api 的方式:

```
bin/hbase org.apache.hadoop.hbase.util.RegionSplitter test_table HexStringSplit -c 10 -f info
```

参数：

test_table 是表名
HexStringSplit 是 split 方式
-c 是分 10 个 region
-f 是 family

可在 UI 上查看结果，如图：

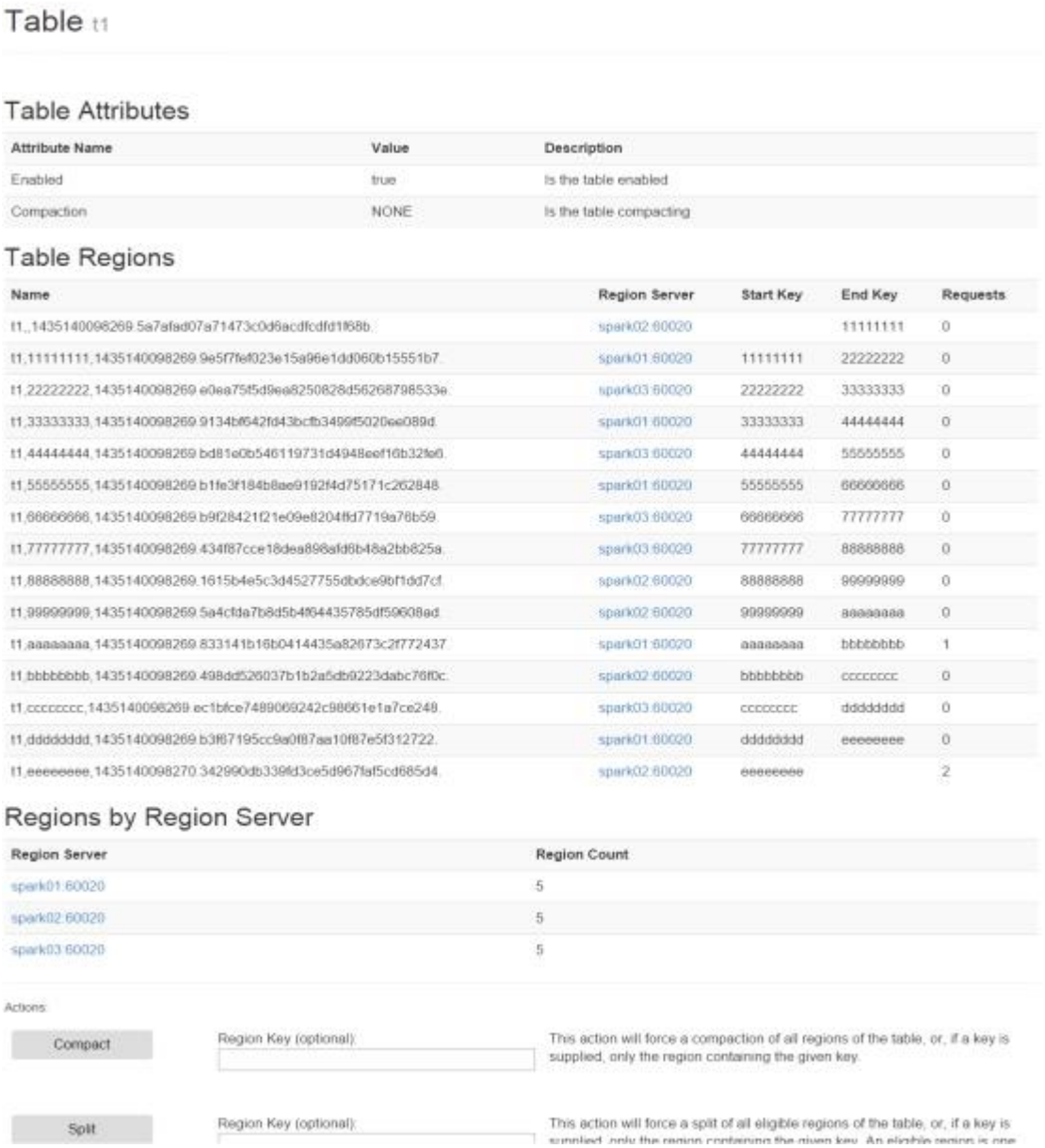


Table t1

Table Attributes

Attribute Name	Value	Description
Enabled	true	Is the table enabled
Compaction	NONE	Is the table compacting

Table Regions

Name	Region Server	Start Key	End Key	Requests
t1,,1435140098269.5a7afad07a71473c0d6acdfcdfd1f68b.	spark02:60020		11111111	0
t1,11111111,1435140098269.9e5f7fef023e15a96e1dd060b15551b7.	spark01:60020	11111111	22222222	0
t1,22222222,1435140098269.e0ea75f5d9ea8250828d56268798533e.	spark03:60020	22222222	33333333	0
t1,33333333,1435140098269.9134bf642fd43bcfb3499f5020ee089d.	spark01:60020	33333333	44444444	0
t1,44444444,1435140098269.bd81e0b546119731d4948eef16b32fe6.	spark03:60020	44444444	55555555	0
t1,55555555,1435140098269.b1fe3f184b8ae9192f4d75171c262848.	spark01:60020	55555555	66666666	0
t1,66666666,1435140098269.b9f28421f21e09e8204ffd7719a76b59.	spark03:60020	66666666	77777777	0
t1,77777777,1435140098269.434f87cce18dea898afd6b48a2bb825a.	spark03:60020	77777777	88888888	0
t1,88888888,1435140098269.1615b4e5c3d4527755dbdce9bf1dd7cf.	spark02:60020	88888888	99999999	0
t1,99999999,1435140098269.5a4cfda7b8d5b4f64435785df59608ad.	spark02:60020	99999999	aaaaaaaa	0
t1,aaaaaaaa,1435140098269.833141b16b0414435a82673c2f772437.	spark01:60020	aaaaaaaa	bbbbbbbb	1
t1,bbbbbbbb,1435140098269.498dd526037b1b2a5db9223dabc76f0c.	spark02:60020	bbbbbbbb	cccccccc	0
t1,cccccccc,1435140098269.ec1bfce7489069242c98661e1a7ce249.	spark03:60020	cccccccc	dddddddd	0
t1,dddddddd,1435140098269.b3f67195cc9a0f87aa10f87e5f312722.	spark01:60020	dddddddd	eeeeeeee	0
t1,eeeeeeee,1435140098270.342990db339fd3ce5d967faf5cd685d4.	spark02:60020	eeeeeeee		2

Regions by Region Server

Region Server	Region Count
spark01:60020	5
spark02:60020	5
spark03:60020	5

Actions:

Compact — Region Key (optional): — This action will force a compaction of all regions of the table, or, if a key is supplied, only the region containing the given key.

Split — Region Key (optional): — This action will force a split of all eligible regions of the table, or, if a key is supplied, only the region containing the given key. An eligible region is one

这样就可以将表预先分为 15 个区，减少数据达到 storefile 大小的时候自动分区的时间消耗，并且还有以一个优势，就是合理设计 rowkey 能让各个 region 的并发请求平均分配(趋于均匀) 使 IO 效率达到最高，但是预分区需要将 filesize 设置一个较大的值，设置哪个参数呢

hbase.hregion.max.filesize 这个值默认是 10G 也就是说单个 region 默认大小是 10G

这个参数的默认值在 0.90 到 0.92 到 0.94.3 各版本的变化：256M--1G--10G

但是如果 MapReduce Input 类型为 TableInputFormat 使用 hbase 作为输入的时候，就要注意了，每个 region 一个 map，如果数据小于 10G 那只会启用一个 map 造成很大的资源浪费，这时候可以考虑适当调小该参数的值，或者采用预分配 region 的方式，并将检测如果达到这个值，再手动分配 region。

2.2 hbase 应用案例看行键设计

表结构设计
1、列族数量的设定
以用户信息为例，可以将必须的基本信息存放在一个列族，而一些附加的额外信息可以放在另一列族；
2、行键的设计
语音详单：
13877889988-20150625
13877889988-20150625
13877889988-20150626
13877889988-20150626
13877889989
13877889989
13877889989
----将需要批量查询的数据尽可能连续存放
CMS 系统----多条件查询
尽可能将查询条件关键词拼装到 rowkey 中，查询频率最高的条件尽量往前靠
20150230-zhangsan-category…
20150230-lisi-category…
(每一个条件的值长度不同，可以通过做定长映射来提高效率)

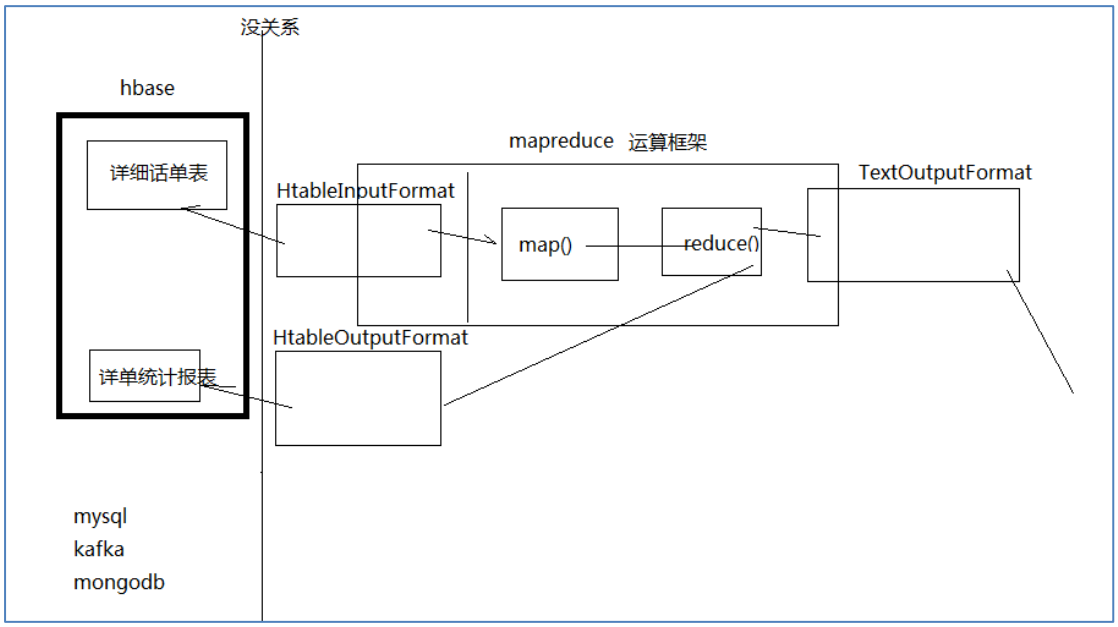
参考：《hbase 实战》----详细讲述了 facebook /GIS 等系统的表结构设计

2.3 Hbase 和 mapreduce 结合

为什么需要用 mapreduce 去访问 hbase 的数据？

——加快分析速度和扩展分析能力

Mapreduce 访问 hbase 数据作分析一定是在离线分析的场景下应用



2.3.1 从 Hbase 中读取数据、分析，写入 hdfs

```
/**
public abstract class TableMapper<KEYOUT, VALUEOUT>
extends Mapper<ImmutableBytesWritable, Result, KEYOUT, VALUEOUT> {
}
 * @author duanhaitao@itcast.cn
 *
 */
public class HbaseReader {

	public static String flow_fields_import = "flow_fields_import";
	static class HdfsSinkMapper extends TableMapper<Text, NullWritable>{

		@Override
		protected void map(ImmutableBytesWritable key, Result value, Context context) throws
IOException, InterruptedException {

			byte[] bytes = key.copyBytes();
			String phone = new String(bytes);
			byte[] urlbytes = value.getValue("f1".getBytes(), "url".getBytes());
			String url = new String(urlbytes);
			context.write(new Text(phone + "\t" + url), NullWritable.get());
```

```
        }

    }

    static class HdfsSinkReducer extends Reducer<Text, NullWritable, Text, NullWritable>{

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

            context.write(key, NullWritable.get());
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = HBaseConfiguration.create();
        conf.set("hbase.zookeeper.quorum", "spark01");

        Job job = Job.getInstance(conf);

        job.setJarByClass(HbaseReader.class);

//          job.setMapperClass(HdfsSinkMapper.class);
        Scan scan = new Scan();
        TableMapReduceUtil.initTableMapperJob(flow_fields_import,                    scan,
HdfsSinkMapper.class, Text.class, NullWritable.class, job);
        job.setReducerClass(HdfsSinkReducer.class);

        FileOutputFormat.setOutputPath(job, new Path("c:/hbasetest/output"));

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);

        job.waitForCompletion(true);
    }

}
```

2.3.2 从 **hdfs** 中读取数据写入 Hbase

```
/**
public abstract class TableReducer<KEYIN, VALUEIN, KEYOUT>
extends Reducer<KEYIN, VALUEIN, KEYOUT, Writable> {
```

```
}
 * @author duanhaitao@itcast.cn
 *
 */
public class HbaseSinker {

	public static String flow_fields_import = "flow_fields_import";
	static class HbaseSinkMrMapper extends Mapper<LongWritable, Text, FlowBean, NullWritable>{
		@Override
		protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

			String line = value.toString();
			String[] fields = line.split("\t");
			String phone = fields[0];
			String url = fields[1];

			FlowBean bean = new FlowBean(phone,url);

			context.write(bean, NullWritable.get());
		}
	}

	static class HbaseSinkMrReducer extends TableReducer<FlowBean, NullWritable, ImmutableBytesWritable>{

		@Override
		protected void reduce(FlowBean key, Iterable<NullWritable> values, Context context) throws IOException, InterruptedException {

			Put put = new Put(key.getPhone().getBytes());
			put.add("f1".getBytes(), "url".getBytes(), key.getUrl().getBytes());

			context.write(new ImmutableBytesWritable(key.getPhone().getBytes()), put);

		}

	}

	public static void main(String[] args) throws Exception {
		Configuration conf = HBaseConfiguration.create();
		conf.set("hbase.zookeeper.quorum", "spark01");
```

```
        HBaseAdmin hBaseAdmin = new HBaseAdmin(conf);

        boolean tableExists = hBaseAdmin.tableExists(flow_fields_import);
        if(tableExists){
            hBaseAdmin.disableTable(flow_fields_import);
            hBaseAdmin.deleteTable(flow_fields_import);
        }
        HTableDescriptor desc = new HTableDescriptor(TableName.valueOf(flow_fields_import));
        HColumnDescriptor hColumnDescriptor = new HColumnDescriptor ("f1".getBytes());
        desc.addFamily(hColumnDescriptor);

        hBaseAdmin.createTable(desc);



        Job job = Job.getInstance(conf);

        job.setJarByClass(HbaseSinker.class);

        job.setMapperClass(HbaseSinkMrMapper.class);
        TableMapReduceUtil.initTableReducerJob(flow_fields_import, HbaseSinkMrReducer.class, job);

        FileInputFormat.setInputPaths(job, new Path("c:/hbasetest/data"));

        job.setMapOutputKeyClass(FlowBean.class);
        job.setMapOutputValueClass(NullWritable.class);

        job.setOutputKeyClass(ImmutableBytesWritable.class);
        job.setOutputValueClass(Mutation.class);

        job.waitForCompletion(true);


    }

}
```

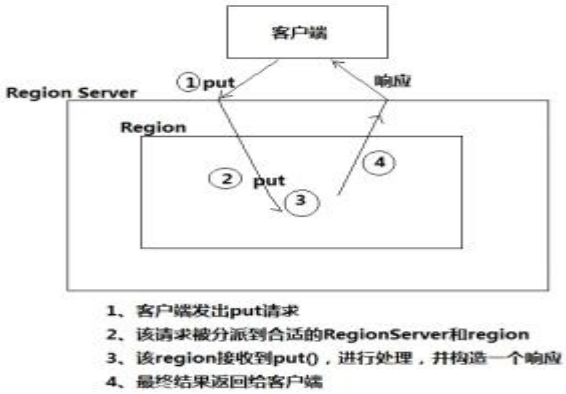
2.3 hbase 高级编程

2.3.1 协处理器---- Coprocessor

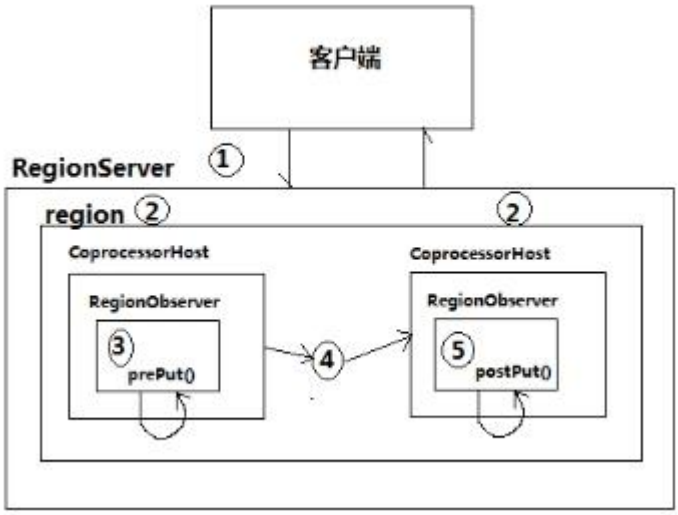
协处理器有两种：observer 和 endpoint

Observer 允许集群在正常的客户端操作过程中可以有不同的行为表现
Endpoint 允许扩展集群的能力，对客户端应用开放新的运算命令

- ✓ Observer 协处理器
  - ✧ 正常 put 请求的流程：



  - ✧ 加入 Observer 协处理后的 put 流程：



1 客户端发出 put 请求
2 该请求被分派给合适的 RegionServer 和 region
3 coprocessorHost 拦截该请求，然后在该表上登记的每个 RegionObserver 上调用 prePut()
4 如果没有被 prePut()拦截，该请求继续送到 region，然后进行处理
5 region 产生的结果再次被 CoprocessorHost 拦截，调用 postPut()
6 假如没有 postPut()拦截该响应，最终结果被返回给客户端

- ✓ Observer 的类型

1、RegionObs——这种 Observer 钩在数据访问和操作阶段，所有标准的数据操作命令都可以被 pre-hooks 和 post-hooks 拦截
2、WALObserver——WAL 所支持的 Observer;可用的钩子是 pre-WAL 和 post-WAL
3、MasterObserver——钩住 DDL 事件，如表创建或模式修改

- ✓ Observer 应用场景示例
  见下节；

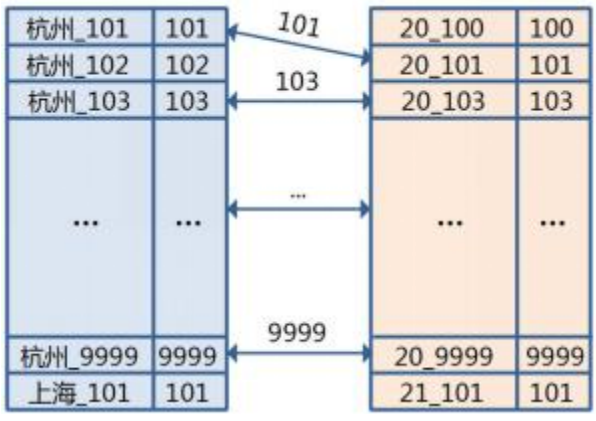
- ➢ Endpoint—参考《Hbase 权威指南》

2.3.2 二级索引

row key 在 HBase 中是以 B+ tree 结构化有序存储的，所以 scan 起来会比较效率。单表以 row key 存储索引，column value 存储 id 值或其他数据 ，这就是 Hbase 索引表的结构。

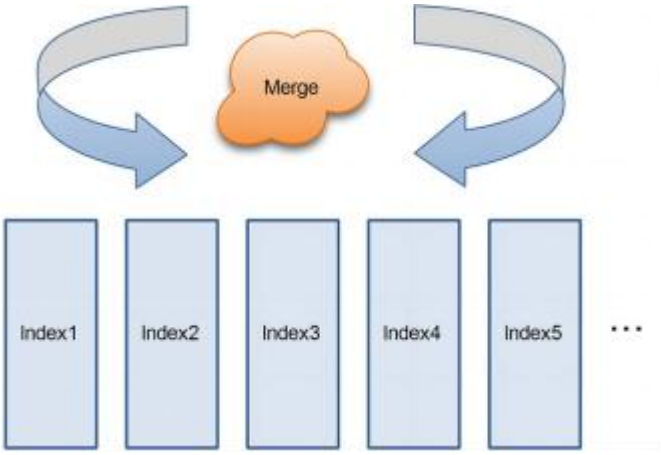
由于 HBase 本身没有二级索引（Secondary Index）机制，基于索引检索数据只能单纯地依靠 RowKey，为了能支持多条件查询，开发者需要将所有可能作为查询条件的字段一一拼接到 RowKey 中，这是 HBase 开发中极为常见的做法

比如，现在有一张 1 亿的用户信息表，建有出生地和年龄两个索引，我想得到一个条件是在杭州出生，年龄为 20 岁的按用户 id 正序排列前 10 个的用户列表。
有一种方案是，系统先扫描出生地为杭州的索引，得到一个用户 id 结果集，这个集合的规模假设是 10 万。然后扫描年龄，规模是 5 万，最后 merge 这些用户 id，去重，排序得到结果。
这明显有问题，如何改良？
保证出生地和年龄的结果是排过序的，可以减少 merge 的数据量？但 Hbase 是按 row key 排序，value 是不能排序的。
变通一下——将用户 id 冗余到 row key 里？OK，这是一种解决方案了，这个方案的图示如下：



merge 时提取交集就是所需要的列表，顺序是靠索引增加了_id，以字典序保证的。

2， 按索引查询种类建立组合索引。
在方案 1 的场景中，想象一下，如果单索引数量多达 10 个会怎么样？10 个索引，就要 merge 10 次，性能可想而知。



解决这个问题需要参考 RDBMS 的组合索引实现。
比如出生地和年龄需要同时查询，此时如果建立一个出生地和年龄的组合索引，查询时效率会高出 merge 很多。
当然，这个索引也需要冗余用户 id，目的是让结果自然有序。结构图示如下：

杭州_20_101	101
杭州_20_102	102
杭州_20_103	103
...	...
杭州_20_999	999
上海_20_100	100

这个方案的优点是查询速度非常快，根据查询条件，只需要到一张表中检索即可得到结果 list。缺点是如果有多个索引，就要建立多个与查询条件一一对应的组合索引

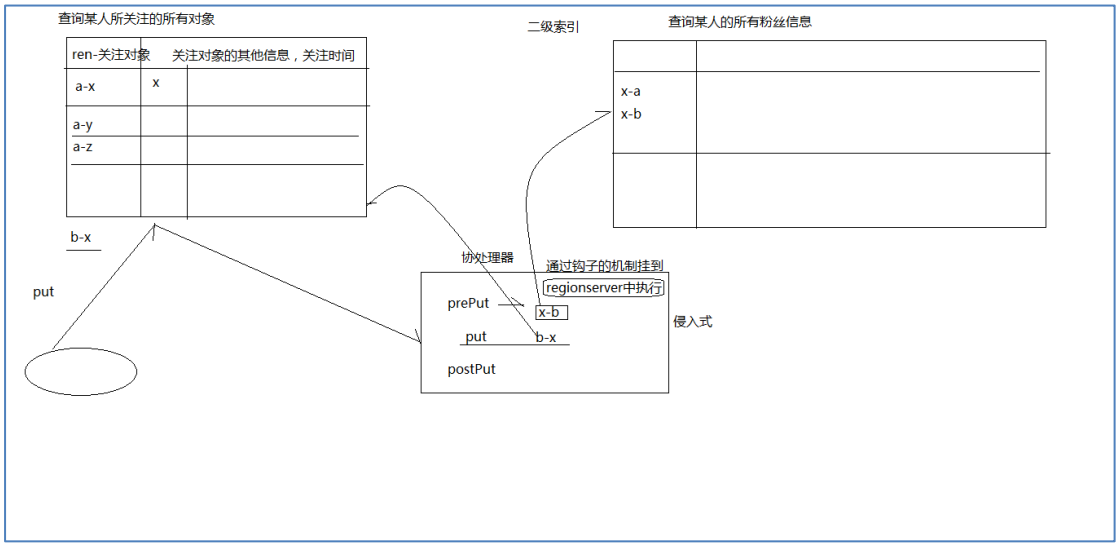
而索引表的维护如果交给应用客户端，则无疑增加了应用端开发的负担
通过协处理器可以将索引表维护的工作从应用端剥离

✓ 利用 Observer 自动维护索引表示例
在社交类应用中，经常需要快速检索各用户的关注列表 t_guanzhu，同时，又需要反向检索各种户的粉丝列表 t_fensi，为了实现这个需求，最佳实践是建立两张互为反向的表：

- 一个表为正向索引关注表 “t_guanzhu”：
  Rowkey: A-B
  f1:From
  f1:To
- 另一个表为反向索引粉丝表：“t_fensi”：

Rowkey: B—A
f1:From
f1:To

插入一条关注信息时，为了减轻应用端维护反向索引表的负担，可用 Observer 协处理器实现：



1、编写自定义 RegionServer

```
public class InverIndexCoprocessor extends BaseRegionObserver {

    @Override
    public void prePut(ObserverContext<RegionCoprocessorEnvironment> e, Put put,
WALEdit edit, Durability durability) throws IOException {
        // set configuration
        Configuration conf = HBaseConfiguration.create();
        // need conf.set...

        HTable table = new HTable(conf, "t_fensi");
        Cell fromCell = put.get("f1".getBytes(), "From".getBytes()).get(0);
        Cell toCell = put.get("f1".getBytes(), "To".getBytes()).get(0);
        byte[] valueArray = fromCell.getValue();
        String from = new String(valueArray);
        valueArray = toCell.getValue();
        String to = new String(valueArray);

        Put putIndex = new Put((to+"-"+from).getBytes());
        putIndex.add("f1".getBytes(), "From".getBytes(),from.getBytes());
        putIndex.add("f1".getBytes(), "To".getBytes(),to.getBytes());

        table.put(putIndex);
```

```
        table.close();

    }
}
```

2、打成 jar 包"fensiguanzhu.jar"上传 hdfs

hadoop fs -put fensiguanzhu.jar /demo/

3、修改 t_fensi 的 schema，注册协处理器

```
hbase(main):017:0>                  alter                  ' t_fensi                   ',METHOD                  =>
'table_att','coprocessor'=>'hdfs://spark01:9000/demo/  fensiguanzhu.jar|cn.itcast.bigdata.hbasecoprocessor.
InverIndexCoprocessor|1001|'
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
```

4、检查是否注册成功

```
hbase(main):018:0> describe 'ff'
DESCRIPTION                                                                                              ENABLED
 'ff', {TABLE_ATTRIBUTES => {coprocessor$1 => 'hdfs://spark01:9000/demo/fensiguanzhu.jar|cn.itcast.bi true
 gdata.hbasecoprocessor.TestCoprocessor|1001|'}, {NAME => 'f1', DATA_BLOCK_ENCODING => 'NONE', BLOOMF
 ILTER => 'ROW', REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSIONS => '0
 ', TTL => '2147483647', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY => 'false', B
 LOCKCACHE => 'true'}, {NAME => 'f2', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATIO
 N_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL => '2147483647', KE
 EP_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
1 row(s) in 0.0250 seconds
```

5、向正向索引表中插入数据进行验证