```
  <name>hive.zookeeper.quorum</name>
  <value>hadoop102,hadoop103,hadoop104</value>
  <description>The list of ZooKeeper servers to talk to. This is only
needed for read/write locks.</description>
</property>
<property>
  <name>hive.zookeeper.client.port</name>
  <value>2181</value>
  <description>The port of ZooKeeper servers to talk to. This is only
needed for read/write locks.</description>
</property>
```

1．案例一

目标：建立 Hive 表，关联 HBase 表，插入数据到 Hive 表的同时能够影响 HBase 表。

分步实现：

(1) 在 Hive 中创建表同时关联 HBase

```
CREATE TABLE hive_hbase_emp_table(
empno int,
ename string,
job string,
mgr int,
hiredate string,
sal double,
comm double,
deptno int)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH       SERDEPROPERTIES       ("hbase.columns.mapping"       =
":key,info:ename,info:job,info:mgr,info:hiredate,info:sal,info:co
mm,info:deptno")
TBLPROPERTIES ("hbase.table.name" = "hbase_emp_table");
```

提示：完成之后，可以分别进入 Hive 和 HBase 查看，都生成了对应的表

(2) 在 Hive 中创建临时中间表，用于 load 文件中的数据

提示：不能将数据直接 load 进 Hive 所关联 HBase 的那张表中

```
CREATE TABLE emp(
empno int,
ename string,
job string,
mgr int,
hiredate string,
sal double,
comm double,
deptno int)
row format delimited fields terminated by '\t';
```

(3) 向 Hive 中间表中 load 数据

```
hive> load data local inpath '/home/admin/softwares/data/emp.txt'
into table emp;
```

(4) 通过 insert 命令将中间表中的数据导入到 Hive 关联 HBase 的那张表中

```
hive> insert into table hive_hbase_emp_table select * from emp;
```

(5) 查看 Hive 以及关联的 HBase 表中是否已经成功的同步插入了数据

Hive：

```
hive> select * from hive_hbase_emp_table;
```

HBase：

```
hbase> scan ‘hbase_emp_table’
```

2．案例二

目标：在 HBase 中已经存储了某一张表 hbase_emp_table，然后在 Hive 中创建一个外部表来关联 HBase 中的 hbase_emp_table 这张表，使之可以借助 Hive 来分析 HBase 这张表中的数据。

注：该案例 2 紧跟案例 1 的脚步，所以完成此案例前，请先完成案例 1。

分步实现：

(1) 在 Hive 中创建外部表

```
CREATE EXTERNAL TABLE relevance_hbase_emp(
empno int,
ename string,
job string,
mgr int,
hiredate string,
sal double,
comm double,
deptno int)
STORED BY
'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" =
":key,info:ename,info:job,info:mgr,info:hiredate,info:sal,info:co
mm,info:deptno")
TBLPROPERTIES ("hbase.table.name" = "hbase_emp_table");
```

(2) 关联后就可以使用 Hive 函数进行一些分析操作了

```
hive (default)> select * from relevance_hbase_emp;
```

第 7 章 HBase 优化

7.1 高可用

在 HBase 中 Hmaster 负责监控 RegionServer 的生命周期，均衡 RegionServer 的负载，如果 Hmaster 挂掉了，那么整个 HBase 集群将陷入不健康的状态，并且此时的工作状态并不会维持太久。所以 HBase 支持对 Hmaster 的高可用配置。

1．关闭 HBase 集群（如果没有开启则跳过此步）

```
[atguigu@hadoop102 hbase]$ bin/stop-hbase.sh
```

2．在 conf 目录下创建 backup-masters 文件

```
[atguigu@hadoop102 hbase]$ touch conf/backup-masters
```

3．在 backup-masters 文件中配置高可用 HMaster 节点

```
[atguigu@hadoop102 hbase]$ echo hadoop103 > conf/backup-masters
```

4．将整个 conf 目录 scp 到其他节点

```
[atguigu@hadoop102 hbase]$ scp -r conf/ hadoop103:/opt/module/hbase/
[atguigu@hadoop102 hbase]$ scp -r conf/ hadoop104:/opt/module/hbase/
```

5．打开页面测试查看

http://hadooo102:16010

7.2 预分区

每一个 region 维护着 startRow 与 endRowKey，如果加入的数据符合某个 region 维护的 rowKey 范围，则该数据交给这个 region 维护。那么依照这个原则，我们可以将数据所要投放的分区提前大致的规划好，以提高 HBase 性能。

1．手动设定预分区

```
hbase> create 'staff1','info','partition1',SPLITS => ['1000','2000','3000','4000']
```

2．生成 16 进制序列预分区

```
create 'staff2','info','partition2',{NUMREGIONS => 15, SPLITALGO => 'HexStringSplit'}
```

3．按照文件中设置的规则预分区

创建 splits.txt 文件内容如下：

```
aaaa
bbbb
cccc
dddd
```

然后执行：

```
create 'staff3','partition3',SPLITS_FILE => 'splits.txt'
```

4．使用 JavaAPI 创建预分区

```
//自定义算法，产生一系列 Hash 散列值存储在二维数组中
byte[][] splitKeys = 某个散列值函数
//创建 HBaseAdmin 实例
HBaseAdmin hAdmin = new HBaseAdmin(HBaseConfiguration.create());
//创建 HTableDescriptor 实例
HTableDescriptor tableDesc = new HTableDescriptor(tableName);
//通过 HTableDescriptor 实例和散列值二维数组创建带有预分区的 HBase 表
hAdmin.createTable(tableDesc, splitKeys);
```

7.3 RowKey 设计

一条数据的唯一标识就是 rowkey，那么这条数据存储于哪个分区，取决于 rowkey 处于哪个一个预分区的区间内，设计 rowkey 的主要目的 ，就是让数据均匀的分布于所有的 region 中，在一定程度上防止数据倾斜。接下来我们就谈一谈 rowkey 常用的设计方案。

1．生成随机数、hash、散列值

```
比如：
原 本 rowKey 为 1001 的 , SHA1 后 变 成 ：dd01903921ea24941c26a48f2cec24e0bb0e8cc7
原 本 rowKey 为 3001 的 , SHA1 后 变 成 ：49042c54de64a1e9bf0b33e00245660ef92dc7bd
原 本 rowKey 为 5001 的 , SHA1 后 变 成 ：7b61dec07e02c188790670af43e717f0f46e8913
在做此操作之前，一般我们会选择从数据集中抽取样本，来决定什么样的 rowKey 来 Hash 后作为每个分区的临界值。
```

2．字符串反转

```
20170524000001 转成 10000042507102
```

20170524000002 转成 20000042507102

这样也可以在一定程度上散列逐步 put 进来的数据。

3．字符串拼接

```
20170524000001_a12e
20170524000001_93i7
```

7.4 内存优化

HBase 操作过程中需要大量的内存开销，毕竟 Table 是可以缓存在内存中的，一般会分配整个可用内存的 70%给 HBase 的 Java 堆。但是不建议分配非常大的堆内存，因为 GC 过程持续太久会导致 RegionServer 处于长期不可用状态，一般 16~48G 内存就可以了，如果因为框架占用内存过高导致系统内存不足，框架一样会被系统服务拖死。

7.5 基础优化

1．允许在 HDFS 的文件中追加内容

hdfs-site.xml、hbase-site.xml

属性：dfs.support.append
解释：开启 HDFS 追加同步，可以优秀的配合 HBase 的数据同步和持久化。默认值为 true。

2．优化 DataNode 允许的最大文件打开数

hdfs-site.xml

属性：dfs.datanode.max.transfer.threads
解释：HBase 一般都会同一时间操作大量的文件，根据集群的数量和规模以及数据动作，设置为 4096 或者更高。默认值：4096

3．优化延迟高的数据操作的等待时间

hdfs-site.xml

属性：dfs.image.transfer.timeout
解释：如果对于某一次数据操作来讲，延迟非常高，socket 需要等待更长的时间，建议把该值设置为更大的值（默认 60000 毫秒），以确保 socket 不会被 timeout 掉。

4．优化数据的写入效率

mapred-site.xml

属性：
mapreduce.map.output.compress
mapreduce.map.output.compress.codec
解释：开启这两个数据可以大大提高文件的写入效率，减少写入时间。第一个属性值修改为 true，第二个属性值修改为：org.apache.hadoop.io.compress.GzipCodec 或者其他压缩方式。

5．设置 RPC 监听数量

hbase-site.xml

属性：hbase.regionserver.handler.count
解释：默认值为 30，用于指定 RPC 监听的数量，可以根据客户端的请求数进行调整，读写请求较多时，增加此值。

6．优化 HStore 文件大小

hbase-site.xml

属性：hbase.hregion.max.filesize
解释：默认值 10737418240（10GB），如果需要运行 HBase 的 MR 任务，可以减小此值，因为一个 region 对应一个 map 任务，如果单个 region 过大，会导致 map 任务执行时间过长。该值的意思就是，如果 HFile 的大小达到这个数值，则这个 region 会被切分为两个 Hfile。

7．优化 hbase 客户端缓存

hbase-site.xml

属性：hbase.client.write.buffer
解释：用于指定 HBase 客户端缓存，增大该值可以减少 RPC 调用次数，但是会消耗更多内存，反之则反之。一般我们需要设定一定的缓存大小，以达到减少 RPC 次数的目的。

8．指定 scan.next 扫描 HBase 所获取的行数

hbase-site.xml

属性：hbase.client.scanner.caching
解释：用于指定 scan.next 方法获取的默认行数，值越大，消耗内存越大。

9．flush、compact、split 机制

当 MemStore 达到阈值，将 Memstore 中的数据 Flush 进 Storefile；compact 机制则是把 flush 出来的小文件合并成大的 Storefile 文件。split 则是当 Region 达到阈值，会把过大的 Region 一分为二。

涉及属性：

即：128M 就是 Memstore 的默认阈值

```
hbase.hregion.memstore.flush.size：134217728
```

即：这个参数的作用是当单个 HRegion 内所有的 Memstore 大小总和超过指定值时，flush 该 HRegion 的所有 memstore。RegionServer 的 flush 是通过将请求添加一个队列，模拟生产消费模型来异步处理的。那这里就有一个问题，当队列来不及消费，产生大量积压请求时，可能会导致内存陡增，最坏的情况是触发 OOM。

```
hbase.regionserver.global.memstore.upperLimit：0.4
hbase.regionserver.global.memstore.lowerLimit：0.38
```

即：当 MemStore 使用内存总量达到 hbase.regionserver.global.memstore.upperLimit 指定值时，将会有多个 MemStores flush 到文件中，MemStore flush 顺序是按照大小降序执行的，直到刷新到 MemStore 使用内存略小于 lowerLimit

第 8 章 HBase 实战之谷粒微博

8.1 需求分析

1) 微博内容的浏览，数据库表设计
2) 用户社交体现：关注用户，取关用户

3) 拉取关注的人的微博内容

8.2 代码实现

8.2.1 代码设计总览：

1) 创建命名空间以及表名的定义
2) 创建微博内容表
3) 创建用户关系表
4) 创建用户微博内容接收邮件表
5) 发布微博内容
6) 添加关注用户
7) 移除（取关）用户
8) 获取关注的人的微博内容
9) 测试

8.2.2 创建命名空间以及表名的定义

```
//获取配置 conf
private Configuration conf = HBaseConfiguration.create();

//微博内容表的表名
private static final byte[] TABLE_CONTENT =
Bytes.toBytes("weibo:content");
//用户关系表的表名
private static final byte[] TABLE_RELATIONS =
Bytes.toBytes("weibo:relations");
//微博收件箱表的表名
private static final byte[] TABLE_RECEIVE_CONTENT_EMAIL =
Bytes.toBytes("weibo:receive_content_email");
public void initNamespace(){
    HBaseAdmin admin = null;
    try {
        admin = new HBaseAdmin(conf);
        //命名空间类似于关系型数据库中的 schema，可以想象成文件夹
        NamespaceDescriptor weibo = NamespaceDescriptor
                .create("weibo")
                .addConfiguration("creator", "Jinji")
                .addConfiguration("create_time",
System.currentTimeMillis() + "")
                .build();
        admin.createNamespace(weibo);
    } catch (MasterNotRunningException e) {
        e.printStackTrace();
    } catch (ZooKeeperConnectionException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        if(null != admin){
            try {
                admin.close();
            } catch (IOException e) {
                e.printStackTrace();
```

```
            }
        }
    }
}
```

8.2.3 创建微博内容表

表结构：

方法名	creatTableeContent
Table Name	weibo:content
RowKey	用户 ID_时间戳
ColumnFamily	info
ColumnLabel	标题,内容,图片
Version	1 个版本

代码：

```
/**
 * 创建微博内容表
 * Table Name:weibo:content
 * RowKey:用户 ID_时间戳
 * ColumnFamily:info
 * ColumnLabel:标题   内容       图片 URL
 * Version:1 个版本
 */
public void createTableContent(){
    HBaseAdmin admin = null;
    try {
        admin = new HBaseAdmin(conf);
        //创建表表述
        HTableDescriptor           content           =           new
HTableDescriptor(TableName.valueOf(TABLE_CONTENT));
        //创建列族描述
        HColumnDescriptor            info            =           new
HColumnDescriptor(Bytes.toBytes("info"));
        //设置块缓存
        info.setBlockCacheEnabled(true);
        //设置块缓存大小
        info.setBlocksize(2097152);
        //设置压缩方式
//        info.setCompressionType(Algorithm.SNAPPY);
        //设置版本确界
        info.setMaxVersions(1);
        info.setMinVersions(1);

        content.addFamily(info);
        admin.createTable(content);

    } catch (MasterNotRunningException e) {
        e.printStackTrace();
    } catch (ZooKeeperConnectionException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        if(null != admin){
```

```
            try {
                admin.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

8.2.4 创建用户关系表

表结构：

方法名	createTableRelations
Table Name	weibo:relations
RowKey	用户 ID
ColumnFamily	attends、fans
ColumnLabel	关注用户 ID，粉丝用户 ID
ColumnValue	用户 ID
Version	1 个版本

代码：

```
/**
 * 用户关系表
 * Table Name:weibo:relations
 * RowKey:用户 ID
 * ColumnFamily:attends,fans
 * ColumnLabel:关注用户 ID，粉丝用户 ID
 * ColumnValue:用户 ID
 * Version：1 个版本
 */
public void createTableRelations(){
    HBaseAdmin admin = null;
    try {
        admin = new HBaseAdmin(conf);
        HTableDescriptor          relations          =          new
HTableDescriptor(TableName.valueOf(TABLE_RELATIONS));

        //关注的人的列族
        HColumnDescriptor          attends          =          new
HColumnDescriptor(Bytes.toBytes("attends"));
        //设置块缓存
        attends.setBlockCacheEnabled(true);
        //设置块缓存大小
        attends.setBlocksize(2097152);
        //设置压缩方式
//          info.setCompressionType(Algorithm.SNAPPY);
        //设置版本确界
        attends.setMaxVersions(1);
        attends.setMinVersions(1);

        //粉丝列族
        HColumnDescriptor          fans          =          new
HColumnDescriptor(Bytes.toBytes("fans"));
        fans.setBlockCacheEnabled(true);
```

```
        fans.setBlocksize(2097152);
        fans.setMaxVersions(1);
        fans.setMinVersions(1);


        relations.addFamily(attends);
        relations.addFamily(fans);
        admin.createTable(relations);

    } catch (MasterNotRunningException e) {
        e.printStackTrace();
    } catch (ZooKeeperConnectionException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        if(null != admin){
            try {
                admin.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

8.2.5 创建微博收件箱表

表结构：

方法名	createTableReceiveContentEmails
Table Name	weibo:receive_content_email
RowKey	用户 ID
ColumnFamily	info
ColumnLabel	用户 ID
ColumnValue	取微博内容的 RowKey
Version	1000

代码：

```
/**
 * 创建微博收件箱表
 * Table Name: weibo:receive_content_email
 * RowKey:用户 ID
 * ColumnFamily:info
 * ColumnLabel:用户 ID-发布微博的人的用户 ID
 * ColumnValue:关注的人的微博的 RowKey
 * Version:1000
 */
public void createTableReceiveContentEmail(){
    HBaseAdmin admin = null;
    try {
        admin = new HBaseAdmin(conf);
        HTableDescriptor      receive_content_email      =      new
HTableDescriptor(TableName.valueOf(TABLE_RECEIVE_CONTENT_EMAIL));
        HColumnDescriptor           info           =           new
HColumnDescriptor(Bytes.toBytes("info"));
```

```
        info.setBlockCacheEnabled(true);
        info.setBlocksize(2097152);
        info.setMaxVersions(1000);
        info.setMinVersions(1000);

        receive_content_email.addFamily(info);;
        admin.createTable(receive_content_email);
    } catch (MasterNotRunningException e) {
        e.printStackTrace();
    } catch (ZooKeeperConnectionException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        if(null != admin){
            try {
                admin.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

8.2.6 发布微博内容

a、微博内容表中添加 1 条数据

b、微博收件箱表对所有粉丝用户添加数据

代码：Message.java

```
package com.atguigu.weibo;

public class Message {
    private String uid;
    private String timestamp;
    private String content;

    public String getUid() {
        return uid;
    }
    public void setUid(String uid) {
        this.uid = uid;
    }
    public String getTimestamp() {
        return timestamp;
    }
    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }
    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
    }
    @Override
    public String toString() {
```

```
        return "Message [uid=" + uid + ", timestamp=" + timestamp + ",
content=" + content + "]";
    }
}
```

代码：public void publishContent(String uid, String content)

```
/**
 * 发布微博
 * a、微博内容表中数据+1
 * b、向微博收件箱表中加入微博的 Rowkey
 */
public void publishContent(String uid, String content){
    HConnection connection = null;
    try {
        connection = HConnectionManager.createConnection(conf);
        //a、微博内容表中添加 1 条数据，首先获取微博内容表描述
        HTableInterface                contentTBL                =
connection.getTable(TableName.valueOf(TABLE_CONTENT));
        //组装 Rowkey
        long timestamp = System.currentTimeMillis();
        String rowKey = uid + "_" + timestamp;

        Put put = new Put(Bytes.toBytes(rowKey));
        put.add(Bytes.toBytes("info"),    Bytes.toBytes("content"),
timestamp, Bytes.toBytes(content));

        contentTBL.put(put);

        //b、向微博收件箱表中加入发布的 Rowkey
        //b.1、查询用户关系表，得到当前用户有哪些粉丝
        HTableInterface                relationsTBL                =
connection.getTable(TableName.valueOf(TABLE_RELATIONS));
        //b.2、取出目标数据
        Get get = new Get(Bytes.toBytes(uid));
        get.addFamily(Bytes.toBytes("fans"));

        Result result = relationsTBL.get(get);
        List<byte[]> fans = new ArrayList<byte[]>();

        //遍历取出当前发布微博的用户的所有粉丝数据
        for(Cell cell : result.rawCells()){
            fans.add(CellUtil.cloneQualifier(cell));
        }
        //如果该用户没有粉丝，则直接 return
        if(fans.size() <= 0) return;
        //开始操作收件箱表
        HTableInterface                recTBL                =
connection.getTable(TableName.valueOf(TABLE_RECEIVE_CONTENT_EMAIL
));
        List<Put> puts = new ArrayList<Put>();
        for(byte[] fan : fans){
            Put fanPut = new Put(fan);
            fanPut.add(Bytes.toBytes("info"),    Bytes.toBytes(uid),
timestamp, Bytes.toBytes(rowKey));
            puts.add(fanPut);
        }
        recTBL.put(puts);
```

```
    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        if(null != connection){
            try {
                connection.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

8.2.7 添加关注用户

a、在微博用户关系表中，对当前主动操作的用户添加新关注的好友

b、在微博用户关系表中，对被关注的用户添加新的粉丝

c、微博收件箱表中添加所关注的用户发布的微博

代码实现：public void addAttends(String uid, String... attends)

```
/**
 * 关注用户逻辑
 * a、在微博用户关系表中，对当前主动操作的用户添加新的关注的好友
 * b、在微博用户关系表中，对被关注的用户添加粉丝（当前操作的用户）
 * c、当前操作用户的微博收件箱添加所关注的用户发布的微博 rowkey
 */
public void addAttends(String uid, String... attends){
    //参数过滤
    if(attends == null || attends.length <= 0 || uid == null ||
uid.length() <= 0){
        return;
    }
    HConnection connection = null;
    try {
        connection = HConnectionManager.createConnection(conf);
        //用户关系表操作对象（连接到用户关系表）
        HTableInterface                relationsTBL                =
connection.getTable(TableName.valueOf(TABLE_RELATIONS));
        List<Put> puts = new ArrayList<Put>();
        //a、在微博用户关系表中，添加新关注的好友
        Put attendPut = new Put(Bytes.toBytes(uid));
        for(String attend : attends){
            //为当前用户添加关注的人
            attendPut.add(Bytes.toBytes("attends"),
Bytes.toBytes(attend), Bytes.toBytes(attend));
            //b、为被关注的人，添加粉丝
            Put fansPut = new Put(Bytes.toBytes(attend));
            fansPut.add(Bytes.toBytes("fans"),    Bytes.toBytes(uid),
Bytes.toBytes(uid));
            //将所有关注的人一个一个的添加到 puts（List）集合中
            puts.add(fansPut);
        }
        puts.add(attendPut);
        relationsTBL.put(puts);

        //c.1、微博收件箱添加关注的用户发布的微博内容（content）的 rowkey
```

```
        HTableInterface                    contentTBL                    =
connection.getTable(TableName.valueOf(TABLE_CONTENT));
        Scan scan = new Scan();
        //用于存放取出来的关注的人所发布的微博的 rowkey
        List<byte[]> rowkeys = new ArrayList<byte[]>();

        for(String attend : attends){
            //过滤扫描 rowkey，即：前置位匹配被关注的人的 uid_
            RowFilter             filter             =             new
RowFilter(CompareFilter.CompareOp.EQUAL,                            new
SubstringComparator(attend + "_"));
            //为扫描对象指定过滤规则
            scan.setFilter(filter);
            //通过扫描对象得到 scanner
            ResultScanner result = contentTBL.getScanner(scan);
            //迭代器遍历扫描出来的结果集
            Iterator<Result> iterator = result.iterator();
            while(iterator.hasNext()){
                //取出每一个符合扫描结果的那一行数据
                Result r = iterator.next();
                for(Cell cell : r.rawCells()){
                    //将得到的 rowkey 放置于集合容器中
                    rowkeys.add(CellUtil.cloneRow(cell));
                }

            }
        }
```

//c.2、将取出的微博 rowkey 放置于当前操作用户的收件箱中

```
        if(rowkeys.size() <= 0) return;
        //得到微博收件箱表的操作对象
        HTableInterface                      recTBL                      =
connection.getTable(TableName.valueOf(TABLE_RECEIVE_CONTENT_EMAIL
));
        //用于存放多个关注的用户的发布的多条微博 rowkey 信息
        List<Put> recPuts = new ArrayList<Put>();
        for(byte[] rk : rowkeys){
            Put put = new Put(Bytes.toBytes(uid));
            //uid_timestamp
            String rowKey = Bytes.toString(rk);
            //借取 uid
            String       attendUID       =       rowKey.substring(0,
rowKey.indexOf("_"));
            long                        timestamp                       =
Long.parseLong(rowKey.substring(rowKey.indexOf("_") + 1));
            //将微博 rowkey 添加到指定单元格中
            put.add(Bytes.toBytes("info"), Bytes.toBytes(attendUID),
timestamp, rk);
            recPuts.add(put);
        }

        recTBL.put(recPuts);

    } catch (IOException e) {
        e.printStackTrace();
```

```
    }finally{
        if(null != connection){
            try {
                connection.close();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

8.2.8 移除（取关）用户

a、在微博用户关系表中，对当前主动操作的用户移除取关的好友(attends)

b、在微博用户关系表中，对被取关的用户移除粉丝

c、微博收件箱中删除取关的用户发布的微博

代码：public void removeAttends(String uid, String... attends)

```
/**
 * 取消关注（remove）
 * a、在微博用户关系表中，对当前主动操作的用户删除对应取关的好友
 * b、在微博用户关系表中，对被取消关注的人删除粉丝（当前操作人）
 * c、从收件箱中，删除取关的人的微博的 rowkey
 */
public void removeAttends(String uid, String... attends){
    //过滤数据
    if(uid == null || uid.length() <= 0 || attends == null ||
attends.length <= 0) return;
    HConnection connection = null;

    try {
        connection = HConnectionManager.createConnection(conf);
        //a、在微博用户关系表中，删除已关注的好友
        HTableInterface                relationsTBL                =
connection.getTable(TableName.valueOf(TABLE_RELATIONS));

        //待删除的用户关系表中的所有数据
        List<Delete> deletes = new ArrayList<Delete>();
        //当前取关操作者的 uid 对应的 Delete 对象
        Delete attendDelete = new Delete(Bytes.toBytes(uid));
        //遍历取关，同时每次取关都要将被取关的人的粉丝-1
        for(String attend : attends){
            attendDelete.deleteColumn(Bytes.toBytes("attends"),
Bytes.toBytes(attend));
            //b
            Delete fansDelete = new Delete(Bytes.toBytes(attend));
            fansDelete.deleteColumn(Bytes.toBytes("fans"),
Bytes.toBytes(uid));
            deletes.add(fansDelete);
        }

        deletes.add(attendDelete);
        relationsTBL.delete(deletes);

        //c、删除取关的人的微博 rowkey 从 收件箱表中
```

```
        HTableInterface                    recTBL                    =
connection.getTable(TableName.valueOf(TABLE_RECEIVE_CONTENT_EMAIL
));

        Delete recDelete = new Delete(Bytes.toBytes(uid));
        for(String attend : attends){
            recDelete.deleteColumn(Bytes.toBytes("info"),
Bytes.toBytes(attend));
        }
        recTBL.delete(recDelete);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

8.2.9 获取关注的人的微博内容

a、从微博收件箱中获取所关注的用户的微博 RowKey

b、根据获取的 RowKey，得到微博内容

代码实现：public List<Message> getAttendsContent(String uid)

```
/**
 * 获取微博实际内容
 * a、从微博收件箱中获取所有关注的人的发布的微博的 rowkey
 * b、根据得到的 rowkey 去微博内容表中得到数据
 * c、将得到的数据封装到 Message 对象中
 */
public List<Message> getAttendsContent(String uid){
    HConnection connection = null;
    try {
        connection = HConnectionManager.createConnection(conf);
        HTableInterface                    recTBL                    =
connection.getTable(TableName.valueOf(TABLE_RECEIVE_CONTENT_EMAIL
));
        //a、从收件箱中取得微博 rowKey
        Get get = new Get(Bytes.toBytes(uid));
        //设置最大版本号
        get.setMaxVersions(5);
        List<byte[]> rowkeys = new ArrayList<byte[]>();
        Result result = recTBL.get(get);
        for(Cell cell : result.rawCells()){
            rowkeys.add(CellUtil.cloneValue(cell));
        }
        //b、根据取出的所有 rowkey 去微博内容表中检索数据
        HTableInterface                  contentTBL                  =
connection.getTable(TableName.valueOf(TABLE_CONTENT));
        List<Get> gets = new ArrayList<Get>();
        //根据 rowkey 取出对应微博的具体内容
        for(byte[] rk : rowkeys){
            Get g = new Get(rk);
            gets.add(g);
        }
        //得到所有的微博内容的 result 对象
        Result[] results = contentTBL.get(gets);

        List<Message> messages = new ArrayList<Message>();
        for(Result res : results){
```

```
            for(Cell cell : res.rawCells()){
                Message message = new Message();

                String                    rowKey                    =
Bytes.toString(CellUtil.cloneRow(cell));
                String       userid       =       rowKey.substring(0,
rowKey.indexOf("_"));
               String timestamp = rowKey.substring(rowKey.indexOf("_")
+ 1);
                String                    content                   =
Bytes.toString(CellUtil.cloneValue(cell));

                message.setContent(content);
                message.setTimestamp(timestamp);
                message.setUid(userid);

                messages.add(message);
            }
        }
        return messages;
    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        try {
            connection.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return null;
}
```

8.2.10 测试

-- 测试发布微博内容

public void testPublishContent(WeiBo wb)

-- 测试添加关注

public void testAddAttend(WeiBo wb)

-- 测试取消关注

public void testRemoveAttend(WeiBo wb)

-- 测试展示内容

public void testShowMessage(WeiBo wb)

代码：

```
/**
 * 发布微博内容
 * 添加关注
 * 取消关注
 * 展示内容
 */
public void testPublishContent(WeiBo wb){
    wb.publishContent("0001", "今天买了一包空气，送了点薯片，非常开心！！
");
    wb.publishContent("0001", "今天天气不错。");
}
```

```
public void testAddAttend(WeiBo wb){
    wb.publishContent("0008", "准备下课！");
    wb.publishContent("0009", "准备关机！");
    wb.addAttends("0001", "0008", "0009");
}

public void testRemoveAttend(WeiBo wb){
    wb.removeAttends("0001", "0008");
}

public void testShowMessage(WeiBo wb){
    List<Message> messages = wb.getAttendsContent("0001");
    for(Message message : messages){
        System.out.println(message);
    }
}

public static void main(String[] args) {
    WeiBo weibo = new WeiBo();
    weibo.initTable();

    weibo.testPublishContent(weibo);
    weibo.testAddAttend(weibo);
    weibo.testShowMessage(weibo);
    weibo.testRemoveAttend(weibo);
    weibo.testShowMessage(weibo);
}
```

第 9 章 扩展

9.1 HBase 在商业项目中的能力

每天：

1) 消息量：发送和接收的消息数超过 60 亿
2) 将近 1000 亿条数据的读写
3) 高峰期每秒 150 万左右操作
4) 整体读取数据占有约 55%，写入占有 45%
5) 超过 2PB 的数据，涉及冗余共 6PB 数据
6) 数据每月大概增长 300 千兆字节。

9.2 布隆过滤器

在日常生活中，包括在设计计算机软件时，我们经常要判断一个元素是否在一个集合中。比如在字处理软件中，需要检查一个英语单词是否拼写正确（也就是要判断它是否在已知的字典中）；在 FBI，一个嫌疑人的名字是否已经在嫌疑名单上；在网络爬虫里，一个网址是否被访问过等等。最直接的方法就是将集合中全部的元素存在计算机中，遇到一个新元素时，将它和集合中的元素直接比较即可。一般来讲，计算机中的集合是用哈希表（hash table）来存储的。它的好处是快速准确，缺点是费存储空间。当集合比较小时，这个问题不显著，

但是当集合巨大时，哈希表存储效率低的问题就显现出来了。比如说，一个像 Yahoo,Hotmail 和 Gmai 那样的公众电子邮件（email）提供商，总是需要过滤来自发送垃圾邮件的人（spamer）的垃圾邮件。一个办法就是记录下那些发垃圾邮件的 email 地址。由于那些发送者不停地在注册新的地址，全世界少说也有几十亿个发垃圾邮件的地址，将他们都存起来则需要大量的网络服务器。如果用哈希表，每存储一亿个 email 地址， 就需要 1.6GB 的内存（用哈希表实现的具体办法是将每一个 email 地址对应成一个八字节的信息指纹 googlechinablog.com/2006/08/blog-post.html，然后将这些信息指纹存入哈希表，由于哈希表的存储效率一般只有 50%，因此一个 email 地址需要占用十六个字节。一亿个地址大约要 1.6GB， 即十六亿字节的内存）。因此存贮几十亿个邮件地址可能需要上百 GB 的内存。除非是超级计算机，一般服务器是无法存储的。

布隆过滤器只需要哈希表 1/8 到 1/4 的大小就能解决同样的问题。

Bloom Filter 是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。Bloom Filter 的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter 不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter 通过极少的错误换取了存储空间的极大节省。

下面我们具体来看 Bloom Filter 是如何用位数组表示集合的。初始状态时，Bloom Filter 是一个包含 m 位的位数组，每一位都置为 0，如图 9-5 所示。

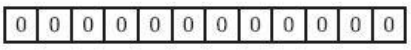

图 9-5

为了表达 S={x1, x2,…,xn}这样一个 n 个元素的集合，Bloom Filter 使用 k 个相互独立的哈希函数（Hash Function），它们分别将集合中的每个元素映射到{1,…,m}的范围中。对任意一个元素 x，第 i 个哈希函数映射的位置 hi(x)就会被置为 1（1≤i≤k）。注意，如果一个位置多次被置为 1，那么只有第一次会起作用，后面几次将没有任何效果。如图 9-6 所示，k=3，且有两个哈希函数选中同一个位置（从左边数第五位）。

图 9-6

在判断 y 是否属于这个集合时，我们对 y 应用 k 次哈希函数，如果所有 $h_i(y)$的位置都是 1（1≤i≤k），那么我们就认为 y 是集合中的元素，否则就认为 y 不是集合中的元素。如图 9-7 所示 y_1 就不是集合中的元素。y_2 或者属于这个集合，或者刚好是一个 false positive。

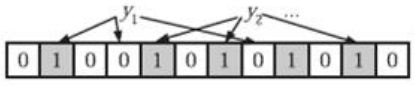

图 9-7

- 为了 add 一个元素，用 k 个 hash function 将它 hash 得到 bloom filter 中 k 个 bit 位，将这 k 个 bit 位置 1。
- 为了 query 一个元素，即判断它是否在集合中，用 k 个 hash function 将它 hash 得到 k 个 bit 位。若这 k bits 全为 1，则此元素在集合中；若其中任一位不为 1，则此元素比不在集合中（因为如果在，则在 add 时已经把对应的 k 个 bits 位置为 1）。
- 不允许 remove 元素，因为那样的话会把相应的 k 个 bits 位置为 0，而其中很有可能有其他元素对应的位。因此 remove 会引入 false negative，这是绝对不被允许的。

布隆过滤器决不会漏掉任何一个在黑名单中的可疑地址。但是，它有一条不足之处，也就是它有极小的可能将一个不在黑名单中的电子邮件地址判定为在黑名单中，因为有可能某个好的邮件地址正巧对应一个八个都被设置成一的二进制位。好在这种可能性很小，我们把它称为误识概率。

布隆过滤器的好处在于快速，省空间，但是有一定的误识别率，常见的补救办法是在建立一个小的白名单，存储那些可能个别误判的邮件地址。

布隆过滤器具体算法高级内容，如错误率估计，最优哈希函数个数计算，位数组大小计算，请参见 http://blog.csdn.net/jiaomeng/article/details/1495500。

9.2 HBase2.0 新特性

2017 年 8 月 22 日凌晨 2 点左右，HBase 发布了 2.0.0 alpha-2，相比于上一个版本，修复了 500 个补丁，我们来了解一下 2.0 版本的 HBase 新特性。

最新文档：

http://hbase.apache.org/book.html#ttl

官方发布主页：

http://mail-archives.apache.org/mod_mbox/www-announce/201708.mbox/<CADcMMgFzmX0xYYso-UAYbU7V8z-Obk1J4pxzbGkRzbP5Hps+iA@mail.gmail.com

举例：

1) region 进行了多份冗余

主 region 负责读写，从 region 维护在其他 HregionServer 中，负责读以及同步主 region 中的

信息，如果同步不及时，是有可能出现 client 在从 region 中读到了脏数据（主 region 还没来得及把 memstore 中的变动的内容 flush）。

2) 更多变动

https://issues.apache.org/jira/secure/ReleaseNote.jspa?version=12340859&styleName=&projectId=12310753&Create=Create&atl_token=A5KQ-2QAV-T4JA-FDED%7Ce6f233490acdf4785b697d4b457f7adb0a72b69f%7Clout