Hbase 高级知识

目录

1、Hbase 高级应用

1.1、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'}

作用：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，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、 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。

5、 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。

6、 describe/desc

这个命令查看了 create table 的各项参数或者是默认值。

使用方式：describe 'user_info'

7、 disable_all/enable_all

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 预分区

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 table with specific split points
hbase>create 'table1','f1',SPLITS => ['\x10\x00', '\x20\x00', '\x30\x00', '\x40\x00']

# create table with four regions based on random bytes keys
hbase>create 'table2','f1', { NUMREGIONS => 8 , SPLITALGO => 'UniformSplit' }

# create table with five regions based on hex keys
hbase>create 'table3','f1', { NUMREGIONS => 10, SPLITALGO => 'HexStringSplit' }
```

也可以使用 api 的方式:

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

hbase org.apache.hadoop.hbase.util.RegionSplitter splitTable HexStringSplit -c 10 -f info

参数：

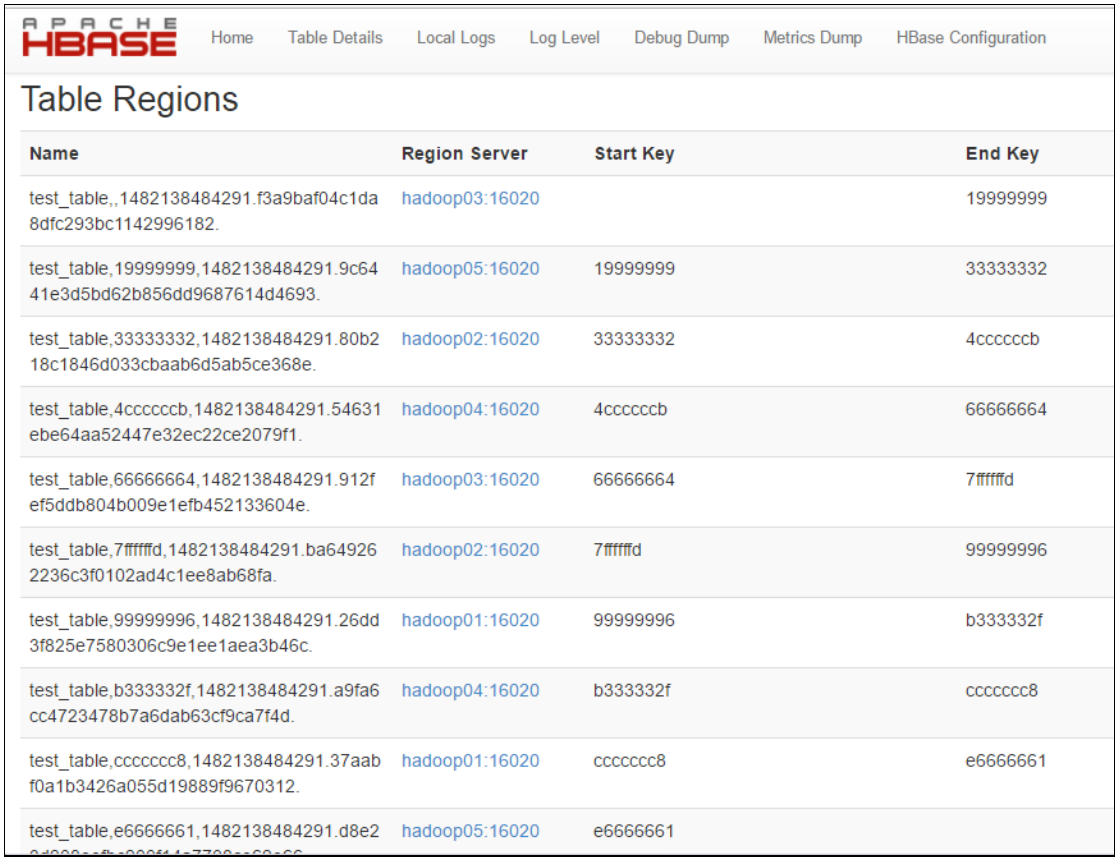
test_table 是表名

HexStringSplit 是 split 方式

-c 是分 10 个 region

-f 是 family

可在 UI 上查看结果，如图：



APACHE HBASE Home Table Details Local Logs Log Level Debug Dump Metrics Dump HBase Configuration

Table Regions

Name	Region Server	Start Key	End Key
test_table,,1482138484291.f3a9baf04c1da8dfc293bc1142996182.	hadoop03:16020		19999999
test_table,19999999,1482138484291.9c6441e3d5bd62b856dd9687614d4693.	hadoop05:16020	19999999	33333332
test_table,33333332,1482138484291.80b218c1846d033cbaab6d5ab5ce368e.	hadoop02:16020	33333332	4ccccccb
test_table,4ccccccb,1482138484291.54631ebe64aa52447e32ec22ce2079f1.	hadoop04:16020	4ccccccb	66666664
test_table,66666664,1482138484291.912fef5ddb804b009e1efb452133604e.	hadoop03:16020	66666664	7fffffd
test_table,7fffffd,1482138484291.ba649262236c3f0102ad4c1ee8ab68fa.	hadoop02:16020	7fffffd	99999996
test_table,99999996,1482138484291.26dd3f825e7580306c9e1ee1aea3b46c.	hadoop01:16020	99999996	b333332f
test_table,b333332f,1482138484291.a9fa6cc4723478b7a6dab63cf9ca7f4d.	hadoop04:16020	b333332f	cccccccc8
test_table,cccccccc8,1482138484291.37aabf0a1b3426a055d19889f9670312.	hadoop01:16020	cccccccc8	e6666661
test_table,e6666661,1482138484291.d8e2	hadoop05:16020	e6666661	

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。

1.2、表设计

1、列簇设计

追求的原则是：在合理范围内能尽量少的减少列簇就尽量减少列簇。

最优设计是：将所有相关性很强的 key-value 都放在同一个列簇下，这样既能做到查询效率最高，也能保持尽可能少的访问不同的磁盘文件

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、RowKey 设计

HBase 中，表会被划分为 1...n 个 Region，被托管在 RegionServer 中。Region 二个重要的属性：StartKey 与 EndKey 表示这个 Region 维护的 rowKey 范围，当我们要读/写数据时，如果 rowKey 落在某个 start-end key 范围内，那么就会定位到目标 region 并且读/写到相关的数据

那怎么快速精准的定位到我们想要操作的数据，就在于我们的 rowkey 的设计了

Rowkey 设计三原则

一、 rowkey 长度原则

Rowkey 是一个二进制码流，Rowkey 的长度被很多开发者建议说设计在 10~100 个字节，不过建议是越短越好，不要超过 16 个字节。

原因如下：

1、数据的持久化文件 HFile 中是按照 KeyValue 存储的，如果 Rowkey 过长比如 100 个字节，1000 万列数据光 Rowkey 就要占用 100*1000 万=10 亿个字节，将近 1G 数据，这会极大影响 HFile 的存储效率；

2、MemStore 将缓存部分数据到内存，如果 Rowkey 字段过长内存的有效利用率会降低，系统将无法缓存更多的数据，这会降低检索效率。因此 Rowkey 的字节长度越短越好。

3、目前操作系统是都是 64 位系统，内存 8 字节对齐。控制在 16 个字节，8 字节的整数倍利用操作系统的最佳特性。

二、rowkey 散列原则

如果 Rowkey 是按时间戳的方式递增，不要将时间放在二进制码的前面，建议将 Rowkey 的高位作为散列字段，由程序循环生成，低位放时间字段，这样将提高数据均衡分布在每个 Regionserver 实现负载均衡的几率。如果没有散列字段，首字段直接是时间信息将产生所有新数据都在一个 RegionServer 上堆积的热点现象，这样在做数据检索的时候负载将会集中在个别 RegionServer，降低查询效率。

三、 rowkey 唯一原则

必须在设计上保证其唯一性。rowkey 是按照字典顺序排序存储的，因此，设计 rowkey 的时候，要充分利用这个排序的特点，将经常读取的数据存储到一块，将最近可能会被访问的数据放到一块。

数据热点

HBase 中的行是按照 rowkey 的字典顺序排序的，这种设计优化了 scan 操作，可以将相关的行以及会被一起读取的行存取在临近位置，便于 scan。然而糟糕的 rowkey 设计是热点的源头。 热点发生在大量的 client 直接访问集群的一个或极少数个节点（访问可能是读，写或者其他操作）。大量访问会使热点 region 所在的单个机器超出自身承受能力，引起性能下降甚至 region 不可用，这也会影响同一个 RegionServer 上的其他 region，由于主机无法服务其他 region 的请求。 设计良好的数据访问模式以使集群被充分，均衡的利用。
为了避免写热点，设计 rowkey 使得不同行在同一个 region，但是在更多数据情况下，数据应该被写入集群的多个 region，而不是一个。

防止数据热点的有效措施：

加盐

这里所说的加盐不是密码学中的加盐，而是在 rowkey 的前面增加随机数，具体就是给 rowkey 分配一个随机前缀以使得它和之前的 rowkey 的开头不同。分配的前缀种类数量应该和你想使用数据分散到不同的 region 的数量一致。加盐之后的 rowkey 就会根据随机生成的前缀分散到各个 region 上，以避免热点。

哈希

哈希会使同一行永远用一个前缀加盐。哈希也可以使负载分散到整个集群，但是读却是可以预测的。使用确定的哈希可以让客户端重构完整的 rowkey，可以使用 get 操作准确获取某一个行数据

反转

第三种防止热点的方法是反转固定长度或者数字格式的 rowkey。这样可以使得 rowkey 中经常改变的部分（最没有意义的部分）放在前面。这样可以有效的随机 rowkey，但是牺牲了 rowkey 的有序性。

反转 rowkey 的例子以手机号为 rowkey，可以将手机号反转后的字符串作为 rowkey，这样的就避免了以手机号那样比较固定开头导致热点问题

时间戳反转

一个常见的数据处理问题是快速获取数据的最近版本，使用反转的时间戳作为 rowkey 的一部分对这个问题十分有用，可以用 Long.Max_Value - timestamp 追加到 key 的末尾，例如 [key][reverse_timestamp] , [key] 的最新值可以通过 scan [key]获得[key]的第一条记录，因为 HBase 中 rowkey 是有序的，第一条记录是最后录入的数据。比如需要保存一个用户的操作记录，按照操作时间倒序排序，在设计 rowkey 的时候，可以这样设计
[userId 反转][Long.Max_Value - timestamp]，在查询用户的所有操作记录数据的时候，直接指定反转后的 userId， startRow 是 [userId 反转][000000000000],stopRow 是 [userId 反转][Long.Max_Value - timestamp]
如果需要查询某段时间的操作记录，startRow 是[user 反转][Long.Max_Value - 起始时间]，stopRow 是[userId 反转][Long.Max_Value - 结束时间]

2、Hbase 高级编程

2.1、协处理器—Coprocessor

1、 起源

Hbase 作为列族数据库最经常被人诟病的特性包括：无法轻易建立“二级索引”，难以执行求和、计数、排序等操作。比如，在旧版本的(<0.92)Hbase 中，统计数据表的总行数，需要使用 Counter 方法，执行一次 MapReduce Job 才能得到。虽然 HBase 在数据存储层中集成了 MapReduce，能够有效用于数据表的分布式计算。然而在很多情况下，做一些简单的相加或者聚合计算的时候，如果直接将计算过程放置在 server 端，能够减少通讯开销，从而获得很好的性能提升。于是，HBase 在 0.92 之后引入了协处理器(coprocessors)，实现一些激动人心的新特性：能够轻易建立二次索引、复杂过滤器(谓词下推)以及访问控制等。

2、介绍

协处理器有两种：observer 和 endpoint

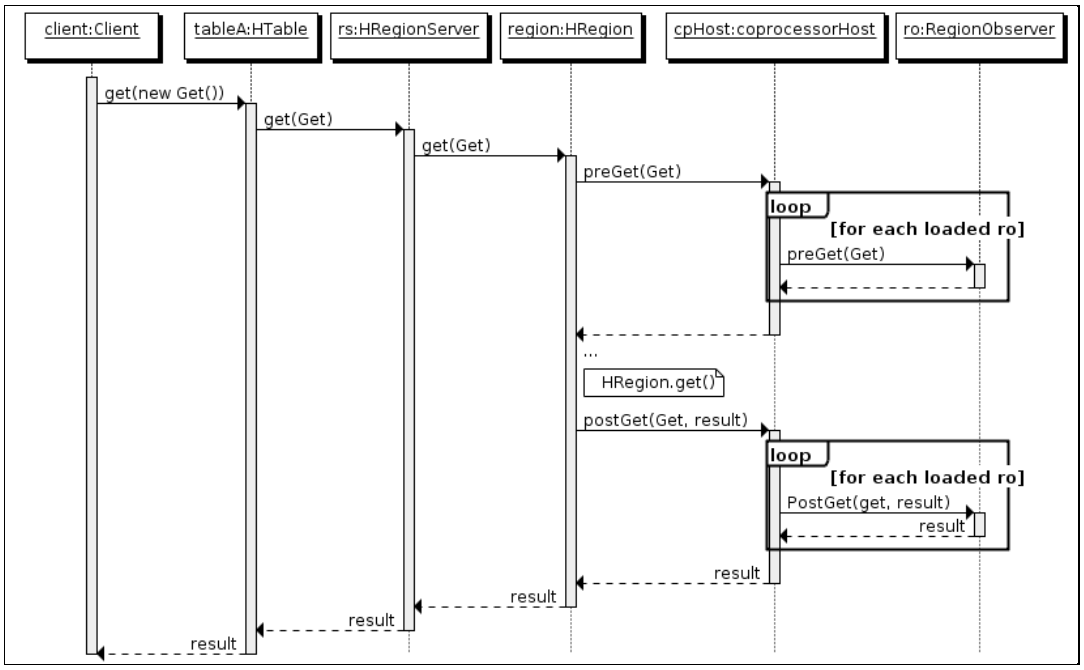
Observer 类似于传统数据库中的触发器，当发生某些事件的时候这类协处理器会被 Server 端调用。Observer Coprocessor 就是一些散布在 HBase Server 端代码中的 hook 钩子，在固定的事件发生时被调用。比如：put 操作之前有钩子函数 prePut，该函数在 put 操作执行前会被 Region Server 调用；在 put 操作之后则有 postPut 钩子函数

以 HBase0.92 版本为例，它提供了三种观察者接口：

- RegionObserver：提供客户端的数据操纵事件钩子：Get、Put、Delete、Scan 等。
- WALObserver：提供 WAL 相关操作钩子。
- MasterObserver：提供 DDL-类型的操作钩子。如创建、删除、修改数据表等。

到 0.96 版本又新增一个 RegionServerObserver

下图是以 RegionObserver 为例子讲解 Observer 这种协处理器的原理：



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()

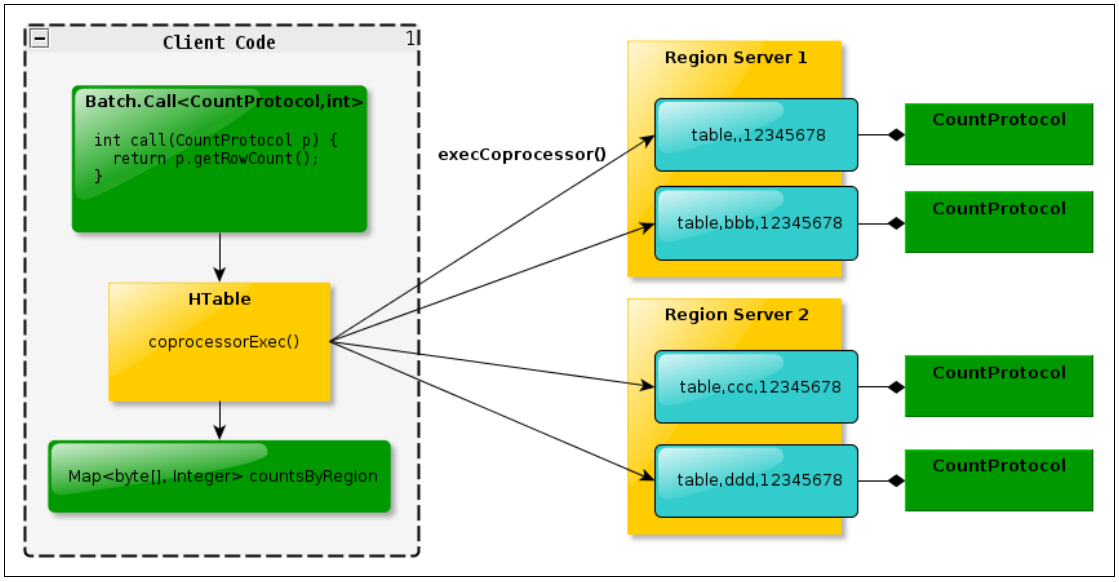
4、如果没有被 prePut()拦截，该请求继续送到 region，然后进行处理

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

Endpoint 协处理器类似传统数据库中的**存储过程**，客户端可以调用这些 Endpoint 协处理器执行一段 Server 端代码，并将 Server 端代码的结果返回给客户端进一步处理，最常见的用法就是进行聚集操作。如果没有协处理器，当用户需要找出一张表中的最大数据，即 max 聚合操作，就必须进行全表扫描，在客户端代码内遍历扫描结果，并执行求最大值的操作。这样的方法无法利用底层集群的并发能力，而将所有计算都集中到 Client 端统一执行，势必效率低下。利用 Coprocessor，用户可以将求最大值的代码部署到 HBase Server 端，HBase 将利用底层 cluster 的多个节点并发执行求最大值的操作。即在每个 Region 范围内执行求最大值的代码，将每个 Region 的最大值在 Region Server 端计算出，仅仅将该 max 值返回给客户端。在客户端进一步将多个 Region 的最大值进一步处理而找到其中的最大值。这样整体的执行效率就会提高很多

下图是 EndPoint 的工作原理：



3、总结

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 类似于 RDBMS 中的触发器，主要在服务端工作
endpoint 类似于 RDBMS 中的存储过程，主要在服务端工作

observer 可以实现权限管理、优先级设置、监控、ddl 控制、二级索引等功能
endpoint 可以实现 **min、max、avg、sum、distinct、group by** 等功能

2.2、协处理加载方式

协处理器的加载方式有两种，我们称之为**静态加载**方式（Static Load）和**动态加载**方式（Dynamic Load）。静态加载的协处理器称之为 **System Coprocessor**，动态加载的协处理器称之为 **Table Coprocessor**

1、 静态加载

通过修改 hbase-site.xml 这个文件来实现，启动全局 aggregation，能过操纵所有的表上的数据。只需要添加如下代码：

```
<property>
        <name>hbase.coprocessor.user.region.classes</name>
        <value>org.apache.hadoop.hbase.coprocessor.AggregateImplementation</value>
</property>
```

为所有 table 加载了一个 cp class，可以用”,”分割加载多个 class

2、 动态加载

启用表 aggregation，只对特定的表生效。通过 HBase Shell 来实现。

◆disable 指定表。hbase> disable 'mytable'

◆添加 aggregation

hbase> alter 'mytable', METHOD => 'table_att','coprocessor'=>
'|org.apache.Hadoop.hbase.coprocessor.AggregateImplementation||'

◆重启指定表 hbase> enable 'mytable'

3、 协处理器卸载

只需要三步即可：

```
disable 'mytable'
alter 'mytable',METHOD=>'table_att_unset',NAME=>'coprocessor$1'
enable 'mytable'
```

2.3、二级索引（ObserverCoprocessor 案例）

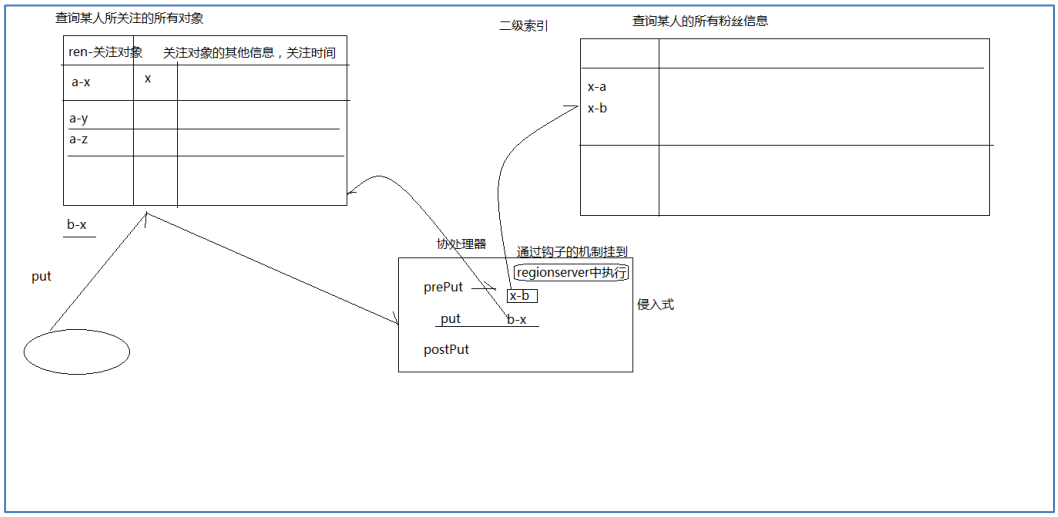
row key 在 HBase 中是以 B+ tree 结构化有序存储的，所以 scan 起来会比较效率。单表以 row key 存储索引，column value 存储 id 值或其他数据 ，这就是 Hbase 索引表的结构。

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

在社交类应用中，经常需要快速检索各用户的关注列表 guanzhu，同时，又需要反向检索各种户的粉丝列表 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：“guanzhu”： Rowkey: a f1:from b
另一个表为反向索引粉丝表：“fensi”： Rowkey: b f1:from a
建表语句: create 'guanzhu','cf1' create 'fensi','cf1'
实现效果：往 guanzhu 表插入一条数据，就会自动往 fensi 表插入一条数据 put 'guanzhu','a','cf1:from','b' put 'fensi','b','cf1:from','a'

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：



实现步骤：

1、 编写 TestCoprocessor 代码

```
package com.ghgj.hbase;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.Cell;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Durability;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.coprocessor.BaseRegionObserver;
import org.apache.hadoop.hbase.coprocessor.ObserverContext;
import org.apache.hadoop.hbase.coprocessor.RegionCoprocessorEnvironment;
import org.apache.hadoop.hbase.regionserver.wal.WALEdit;

public class TestCoprocessor extends BaseRegionObserver {

    static Configuration config = HBaseConfiguration.create();
    static HTable table = null;
    static{
        config.set("hbase.zookeeper.quorum",
"hadoop01:2181,hadoop02:2181,hadoop03:2181,hadoop04:2181,hadoop05:2181");
        try {
            table = new HTable(config, "guanzhu");
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
        }

        @Override
        public void prePut(ObserverContext<RegionCoprocessorEnvironment> e,
                Put put, WALEdit edit, Durability durability) throws IOException {
    //        super.prePut(e, put, edit, durability);

            byte[] row = put.getRow();
            Cell cell = put.get("f1".getBytes(), "from".getBytes()).get(0);

            Put putIndex = new Put(cell.getValueArray(),cell.getValueOffset(),cell.getValueLength());

            putIndex.addColumn("f1".getBytes(), "from".getBytes(), row);

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

2、 打成 jar 包（cppp.jar），上传到 hdfs 中的 hbasecp 目录下

```
[root@hadoop01 soft]# hadoop fs -put cppp.jar /hbasecp
```

3、 建 hbase 表，请按以下顺序操作

```
hbase(main):036:0> create 'guanzhu','f1'
hbase(main):036:0> create 'fensi','f1'
hbase(main):036:0> disable 'fensi'
hbase(main):036:0> alter 'fensi',METHOD => 'table_att','coprocessor' => 'hdfs://myha01/hbasecp/cpp.jar|com.ghgj.hbase.TestCoprocessor|1001|'

# 理解 coprocessor 的四个参数，分别用'|'隔开的
1、 你的协处理器 jar 包所在 hdfs 上的路径
2、 协处理器类全限定名
3、 协处理器加载顺序
4、 传参
hbase(main):036:0> enable 'fensi'
```

4、 现在插入数据进行验证，命令行和代码都可以

```
testPut("fensi", "c", "f1", "from", "b");
```

```
public static void testPut(String tableName, String rowkey,String cf,
        String key, String value)  throws Exception{
    HTable table = new HTable(config, tableName);
    Put put = new Put(rowkey.getBytes());
    put.addColumn(cf.getBytes(), key.getBytes(), value.getBytes());
    table.put(put);
    System.out.println("成功插入一条数据");
    table.close();
}
```

5、 结果演示：

```
hbase(main):036:0> scan 'guanzhu'
ROW                             COLUMN+CELL
 b                              column=f1:from, timestamp=1482125382074, value=a
1 row(s) in 0.0380 seconds

hbase(main):037:0> scan 'fensi'
ROW                             COLUMN+CELL
 a                              column=f1:from, timestamp=1482125381460, value=b
1 row(s) in 0.0260 seconds
```