使用Observer实现HBase到Elasticsearch的数据同步、

# 概述

多个数据源中的数据同步问题，无非就三种解决方式：

（1）客户端双写，分别将数据写入两个数据源（同步、异步）

（2）主数据源在收到数据后推给辅数据源（同步、异步）

（3）辅数据源从主数据源中拉取数据（轮训、监听，全量、增量）

具体到 HBase 同步到 Elasticsearch 时，后两种方式具体对应的方案就是 HBase 的 Observer 和 Elasticsearch 的 River，这两种方式都可以使开发者在数据源中嵌入自己的业务逻辑，并且依托于集群可以轻松地保证高可用

但是非常遗憾的是，要使用 River 高效的同步数据，必须要有一种拉取增量数据的方式，而在 HBase 中这并没有很好的方法实现，所以本文将会介绍使用 Observer 的方法。

题外话：Elasticsearch 的 MySQL River 有两种实现：elasticsearch-river-jdbc和elasticsearch-river-mysql。前者简单的通过 SQL 查询数据同步到 Elasticsearch，所以必须要在表中定义更新时间的字段才能完成增量更新，而且它无法得知哪些数据删除掉了，除非增加并使用逻辑删除字段。而后者则通过 MySQL 的主从复制机制，读取 Binlog 完成增量数据的同步，要更加方便和实用很多。

# **什么是 Observer**

HBase 0.92 版本引入了协处理器（Coprocessor），可以使开发者将自己的代码嵌入到 HBase 中，其中协处理器分为两大块，一个是终端（Endpoint），另一个是本文将要介绍的观察者（Observer）。

Observer 有些类似于 MySQL 中的触发器（Trigger），它可以为 HBase 中的操作添加钩子，并在事件发生后实现自己的的业务逻辑。

Observer 主要分为三种：

RegionObserver：增删改查相关，例如 Get、Put、Delete、Scan 等

WALObserver：WAL 操作相关

MasterObserver：DDL-类型相关，例如创建、删除、修改数据表等

数据同步将会使用 RegionObserver 监听 Put 和 Delete 事件。

## **如何实现自己的 Observer**

每一个 Observer 都是一个 Jar 包。首先需要引入hbase-server包，并实现如BaseRegionObserver等 HBase 提供的相关接口，重写需要监听对应事件的方法。

实现数据同步功能可以重写postPut和putDelete方法监听 Put 和 Delete 事件。

下面就是一个最简单的例子，在这两个方法中分别得到表名和 RowKey，然后输出到 HBase 默认的日志中：

public class SimpleObserver extends BaseRegionObserver {  
  
 private static final Log logger = LogFactory.getLog (SimpleObserver.class);  
  
 @Override  
 public void postPut (ObserverContext<RegionCoprocessorEnvironment> e, Put put, WALEdit edit, Durability durability) throws IOException {  
 // 拿到表名  
 String table = e.getEnvironment ().getRegion ().getRegionInfo ().getTable ().getNameAsString ();  
 // 拿到 row key  
 String rowKey = new String (delete.getRow ());  
 logger.info ("a put event! table: " + table + ", row key: " + rowKey);  
 }  
  
 @Override  
 public void postDelete (ObserverContext<RegionCoprocessorEnvironment> e, Delete delete, WALEdit edit, Durability durability) throws IOException {  
 // 拿到表名  
 String table = e.getEnvironment ().getRegion ().getRegionInfo ().getTable ().getNameAsString ();  
 // 拿到 row key  
 String rowKey = new String (delete.getRow ());  
 logger.info ("a delete event! table: " + table + ", row key: " + rowKey);  
 }  
}

之后将项目打包，上传到 HDFS 中：

hdfs dfs -mkdir /observers  
hdfs dfs -put simple-observer.jar /observers

使用 HBase Shell 创建一个表，将这个 Observer 挂到该表中：

create 'test\_observer'  
  
disable 'test\_observer'  
  
alter ‘test\_observer', METHOD => 'table\_att', 'coprocessor' => 'hdfs:///observers/simple-observer.jar|com.scienjus.observer.SimpleObserver|'  
  
enable 'test\_observer'

coprocessor的值是一个字符串，由以下几个部分组成：

jar 地址（如果在配置文件中定义了 CLASS\_PATH 可以不填）|类名（包含包路径）|优先级|自定义属性

此时通过describe可以看到这个表已经挂上了观察者：

describe 'test\_observer'  
  
Table test\_observer is ENABLED  
  
test\_observer, {TABLE\_ATTRIBUTES => {coprocessor$1 => 'hdfs:///observers/simple-observer.jar|com.scienjus.observer.SimpleObserver|'}  
COLUMN FAMILIES DESCRIPTION  
{NAME => 'info', DATA\_BLOCK\_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATION\_SCOPE => '0', VERSIO  
NS => '1', COMPRESSION => 'NONE', MIN\_VERSIONS => '0', TTL => 'FOREVER', KEEP\_DELETED\_CELLS => 'FALSE'  
, BLOCKSIZE => '65536', IN\_MEMORY => 'false', BLOCKCACHE => 'true'}  
1 row (s) in 0.2600 seconds

向这个表中进行 Put 和 Delete 操作，就可以看到对应的日志了：

## 如何同步数据到 Elasticsearch

Elasticsearch 官方的 Java 客户端提供了一个名为BulkProcessor的接口，这个接口可以轻易的实现一个批量发送请求的缓冲池。

下面这段代码创建了一个缓冲池，它会定期批量发送堆积的请求，触发条件为：

* 每 2 秒触发一次
* 当堆积的请求数量达到 1000 个时，触发一次
* 当堆积的请求达到 100mb 时，触发一次

processor = BulkProcessor.builder (client, new BulkProcessor.Listener () {  
 @Override  
 public void beforeBulk (long executionId, BulkRequest request) {  
 logger.info ("before bulk !!!");  
 }  
  
 @Override  
 public void afterBulk (long executionId, BulkRequest request, BulkResponse response) {  
 }  
  
 @Override  
 public void afterBulk (long executionId, BulkRequest request, Throwable failure) {  
 }  
})  
 .setBulkActions (1000)  
 .setBulkSize (new ByteSizeValue (100, ByteSizeUnit.MB))  
 .setFlushInterval (TimeValue.timeValueSeconds (2))  
 .setConcurrentRequests (5)  
 .build ();

同时它还提供了一个监听器，可以在发送请求前、发送请求后、发送请求出现异常时监听到对应事件并进行处理。可以在其中处理失败情况，例如重发或是记录日志。

将 Observer 和 BulkProcessor 结合起来，只需要在 postPut 时将文档转为 JSON 生成 Upsert 请求加入缓冲池，在 postDelete 时将 RowKey 作为 id 生成删除请求加入缓冲池即可，例如：

@Override  
public void postPut (ObserverContext<RegionCoprocessorEnvironment> e, Put put, WALEdit edit, Durability durability) throws IOException {  
 try {  
 // 拿到表名  
 String table = e.getEnvironment ().getRegion ().getRegionInfo ().getTable ().getNameAsString ();  
 // 拿到 id  
 String id = new String (put.getRow ());  
 logger.info ("a put! table: " + table + ", key: " + id);  
 // 拿到文档内容  
 Map<String, String> doc = new HashMap<>();  
  
 for (List<Cell> cells : put.getFamilyCellMap ().values ()) {  
 for (Cell cell : cells) {  
 doc.put (new String (CellUtil.cloneQualifier (cell)), new String (CellUtil.cloneValue (cell)));  
 }  
 }  
 processor.add (new UpdateRequest ()  
 .index (index)  
 .type (type)  
 .id (id)  
 .doc (doc)  
 .docAsUpsert (true)  
 );  
 } catch (RuntimeException ex) {  
 // TODO 记录运行异常  
 logger.info ("error!");  
 }  
}  
  
@Override  
public void postDelete (ObserverContext<RegionCoprocessorEnvironment> e, Delete delete, WALEdit edit, Durability durability) throws IOException {  
 try {  
 // 拿到表名  
 String table = e.getEnvironment ().getRegion ().getRegionInfo ().getTable ().getNameAsString ();  
 // 拿到 id  
 String id = new String (delete.getRow ());  
 logger.info ("a delete! table: " + table + ", key: " + id);  
 processor.add (new DeleteRequest ()  
 .index (index)  
 .type (type)  
 .id (id)  
 );  
 } catch (RuntimeException ex) {  
 // TODO 记录运行异常  
 logger.info ("error!");  
 }  
}

最后别忘了监听stop事件，将缓冲池和客户端都关闭：

@Override  
public void stop (CoprocessorEnvironment e) throws IOException {  
 processor.close ();  
 client.close ();  
}