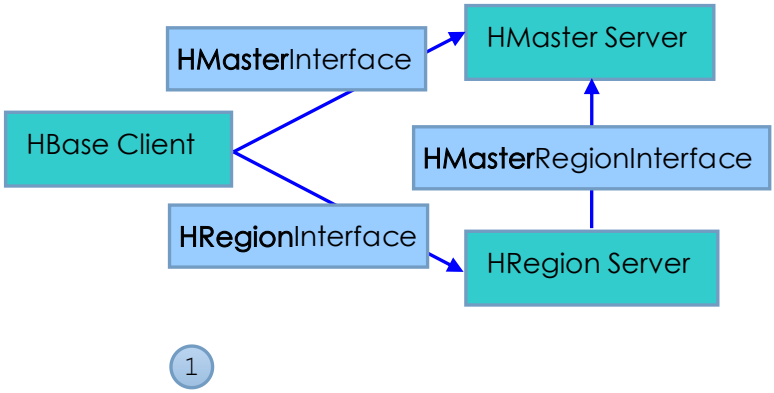
# HBase IPC

| | RPC | Server 内部类 | 服务器 | 客户端 |
|---|---|---|---|---|
| Hadoop | RPC | RPC.Server | Server | Client |
| HBase | HBaseRPC | WritableRpcEngine | HBaseServer | HBaseClient |

| 客户端 | 服务器 | 通信接口 |
|---|---|---|
| HBase Client | Master Server (HMaster) | HMasterInterface |
| HBase Client | Region Server (HRegionServer) | HRegionInterface |
| Region Server | Master Server (HMaster) | HMasterRegionInterface |



# Put

先来看一个 HBase 最基本的 put 存储操作:

```
        HTable table = new HTable(conf, tablename);
        Put put = new Put(Bytes.toBytes("row-001"));
        put.add(Bytes.toBytes("family-01"), Bytes.toBytes("col-01"), Bytes.toBytes("value-01"));
        put.add(Bytes.toBytes("family-01"), Bytes.toBytes("col-02"), Bytes.toBytes("value-02"));
        table.put(put);
```

调用 HTable 的构造函数:

```
public class HTable implements HTableInterface {
  private HConnection connection;                    // 连接实例
  private final byte [] tableName;                   // 操作的表名
  private volatile Configuration configuration;
  private final ArrayList<Put> writeBuffer = new ArrayList<Put>();  // 写入时的客户端缓冲区
  private long writeBufferSize;
  private boolean clearBufferOnFail;                 // 失败时是否清空缓冲区
  private boolean autoFlush;                         // 是否自动刷新
  private long currentWriteBufferSize;               // 客户端当前已经写入到缓冲区的大小
  protected int scannerCaching;                      // 扫描器的缓存数量
  private int maxKeyValueSize;
  private ExecutorService pool;                      // For Multi  多线程
  private boolean closed;
  private int operationTimeout;
  private final boolean cleanupPoolOnClose;          // shutdown the pool in close()
  private final boolean cleanupConnectionOnClose;    // close the connection in close()

  /** Creates an object to access a HBase table.
   * Shares zookeeper connection and other resources with other HTable instances created with the same conf instance.
   * Uses already-populated region cache if one is available, populated by any other HTable instances sharing this conf instance. */
  public HTable(Configuration conf, final byte [] tableName){
    this.tableName = tableName;
    this.cleanupPoolOnClose = this.cleanupConnectionOnClose = true;
    this.connection = HConnectionManager.getConnection(conf);
    this.configuration = conf;
    this.pool = getDefaultExecutor(conf);
    this.finishSetup();
  }

  private static ThreadPoolExecutor getDefaultExecutor(Configuration conf) {
    int maxThreads = conf.getInt("hbase.htable.threads.max", Integer.MAX_VALUE);
    long keepAliveTime = conf.getLong("hbase.htable.threads.keepalivetime", 60);
    // we only create as many Runnables as there are region servers. It means it also scales when new region servers are added.
    ThreadPoolExecutor pool = new ThreadPoolExecutor(1, maxThreads, keepAliveTime, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>(), Threads.newDaemonThreadFactory("hbase-table"));
    pool.allowCoreThreadTimeOut(true);
    return pool;
  }

  private void finishSetup()  {
```

```
    this.connection.locateRegion(tableName, HConstants.EMPTY_START_ROW);
    this.writeBufferSize = this.configuration.getLong("hbase.client.write.buffer", 2097152);
    this.clearBufferOnFail = true;
    this.autoFlush = true;
    this.currentWriteBufferSize = 0;
    this.scannerCaching = this.configuration.getInt("hbase.client.scanner.caching", 1);
    this.maxKeyValueSize = this.configuration.getInt("hbase.client.keyvalue.maxsize", -1);
    this.closed = false;
  }
```

实例化 HTable 时会从 HConnectionManager 管理类中获取一个 HConnection 实现类. 使用 Map 来缓存 HConnection 实例. Map 的 key 是由配置文件构造成的 HConnectionKey, value 是 HConnection 实现类: HConnectionImplementation.

```
public class HConnectionManager {
  // An LRU Map of HConnectionKey -> HConnection (TableServer).  All access must be synchronized.
  static final Map<HConnectionKey, HConnectionImplementation> HBASE_INSTANCES;

  public static HConnection getConnection(Configuration conf) {
    HConnectionKey connectionKey = new HConnectionKey(conf);
    synchronized (HBASE_INSTANCES) {
      HConnectionImplementation connection = HBASE_INSTANCES.get(connectionKey);
      if (connection == null) {
        connection = new HConnectionImplementation(conf, true, null);
        HBASE_INSTANCES.put(connectionKey, connection);
      }
      connection.incCount();
      return connection;
    }
  }
```

## HConnection

```
public Configuration getConfiguration();
```

获取 HConnection 实例使用的配置对象

```
public HTableInterface getTable(String tableName);
public HTableInterface getTable(byte[] tableName);
public HTableInterface getTable(String tableName, ExecutorService pool);
public HTableInterface getTable(byte[] tableName, ExecutorService pool);
```

参数可以是 String 或 byte[]的 tableName, 或可选的 ExecutorService pool 连接池

返回类型 HTableInterface 有: HTable, PolledTable, RemoteTable

```
public ZooKeeperWatcher getZooKeeperWatcher();
```

获得该连接可以使用的一个 ZooKeeperWrapper, 进而获取-ROOT-,.META.表信息, session 信息等

```
public HMasterInterface getMaster();
public boolean isMasterRunning();
```

获得一个到 HMaster Server 的连接

```
public boolean isTableEnabled(byte[] tableName);
public boolean isTableDisabled(byte[] tableName);
public boolean isTableAvailable(byte[] tableName);
```

```
public HTableDescriptor[] listTables();
```

获取所有的用户表，扫描.META.表，返回代表每个用户表的 HTableDescriptor[]数组

```
public HTableDescriptor getHTableDescriptor(byte[] tableName);
public HTableDescriptor[] getHTableDescriptors(List<String> tableNames);
```

参数是 byte[] tableName，获取指定表的 HTableDescriptor 对象．也有参数为列表返回数组

```
public HRegionLocation locateRegion(final byte [] tableName, final byte [] row);
public HRegionLocation relocateRegion(final byte [] tableName, final byte [] row);
public HRegionLocation locateRegion(final byte [] regionName);
public List<HRegionLocation> locateRegions(final byte[] tableName);
public List<HRegionLocation> locateRegions(final byte[] tableName, final boolean useCache, final boolean offlined);
HRegionLocation getRegionLocation(byte [] tableName, byte [] row, boolean reload)
```

根据 tableName 和 row 定位对应的 Region 位置信息．没有指定 row 时定位的是表的 Region 列表

因为一张表会分成多个 Region，指定 row 后，只会对应其中的一个 region.

```
public HRegionInterface getHRegionConnection(final String hostname, final int port)
public HRegionInterface getHRegionConnection(final String hostname, final int port, boolean getMaster)
```

指定 host 和 port 后，就能获取到 HRegionInterface: HRegionServer 的一个连接实例.

RegionServer 与 Region 的关系类似 NameNode 和 BlockLocation 以及 DataNode 和 Block 的关系.

RegionServer/NN/DN 是真正的物理机器，通过主机名/IP 地址和端口就能建立到 RegionServer 的连接.

```
public <T> T ~~getRegionServerWithRetries~~(ServerCallable<T> callable)
public <T> T ~~getRegionServerWithoutRetries~~(ServerCallable<T> callable)
```

传递一个 ServerCallable 对象，在其中的 call 方法可以实现自己的逻辑. with 表示会重试, without 不重试.

调用该方法，就能管理等待重试的过程，重新找到丢失的 region 信息(region 失效需要重新获取).

```
public void processBatch(List<? extends Row> actions, final byte[] tableName, ExecutorService pool, Object[] results);
public <R> void processBatchCallback(List<? extends Row> list, byte[] tableName,
    ExecutorService pool, Object[] results, Batch.Callback<R> callback);
```

批处理，动作有 Put, Delete, Get. 同一个 RegionServer 的所有动作只形成一次 RPC 调用.

```
public void clearRegionCache();
public void clearRegionCache(final byte [] tableName);
public void deleteCachedRegionLocation(final HRegionLocation location);
public void prewarmRegionCache(final byte[] tableName, final Map<HRegionInfo, ~~HServerAddress~~> regions);
public void clearCaches(final String servername);
```

客户端会缓存 Region 信息，也可以清除指定或所有的缓存信息.

主要方法:

| 方法 | 说明 |
| --- | --- |
| **HTableInterface** getTable | 获取 HTable 实例 |
| **HTableDescriptor** getHTableDescriptor | 获取表描述符 |
| **HMasterInterface** ~~getMaster()~~ | 建立到 Master Server 的连接 |
| **HRegionInterface** getHRegionConnection | 建立到 Region Server 的连接 |
| **HRegionLocation** locateRegion | 定位表某一行的 Region 位置信息 |
| T ~~getRegionServerWithRetries~~ | 尝试连接 RegionServer 并回调 ServerCallable |
| **void** processBatch | 批处理 |

# HConnectionManager

HConnectionManager 是对 HConnection 的管理类. 管理动作一般有添加, 删除 HConnection 实例等.
HConnectionImplementation 是 HConnectionManager 的内部类, 实现了 HConnection 接口.

```
/* Encapsulates connection to zookeeper and regionservers.*/
static class HConnectionImplementation implements HConnection, Closeable {
    private final Class<? extends HRegionInterface> serverInterfaceClass;     // HRegionServer,通过反射实例化对象
    private final long pause;
    private final int numRetries;
    private final int maxRPCAttempts;
    private final int rpcTimeout;
    private final int prefetchRegionLimit;

    private final Object masterLock = new Object();
    private volatile boolean closed;
    private volatile boolean aborted;
    private volatile boolean resetting;
    private volatile HMasterInterface master;                                 // HMaster
    private volatile ZooKeeperWatcher zooKeeper;                              // ZooKeeper reference
    private volatile MasterAddressTracker masterAddressTracker;               // ZooKeeper-based master address tracker
    private volatile RootRegionTracker rootRegionTracker;
    private volatile ClusterId clusterId;

    private final Object metaRegionLock = new Object();
    private final Object userRegionLock = new Object();
    private final Object resetLock = new Object();

    // thread executor shared by all HTableInterface instances created by this connection
    private volatile ExecutorService batchPool = null;
    private volatile boolean cleanupPool = false;
    private final Configuration conf;
    private RpcEngine rpcEngine;

    // Known region HServerAddress.toString() -> HRegionInterface(RegionServer)
    private final Map<String, HRegionInterface> servers = new ConcurrentHashMap<String, HRegionInterface>();
    private final ConcurrentHashMap<String, String> connectionLock = new ConcurrentHashMap<String, String>();

    // Map of table to table HRegionLocations.    The table key is made by doing a Bytes#mapKey(byte[]) of the table's name.
    private final Map<Integer, SoftValueSortedMap<byte [], HRegionLocation>>
      cachedRegionLocations = new HashMap<Integer, SoftValueSortedMap<byte [], HRegionLocation>>();
    // The presence of a server in the map implies it's likely that there is an entry in cachedRegionLocations that map to this server;
    // but the absence of a server in this map guarentees that there is no entry in cache that maps to the absent server.
    private final Set<String> cachedServers = new HashSet<String>();
    // region cache prefetch is enabled by default. this set contains all tables whose region cache prefetch are disabled.
    private final Set<Integer> regionCachePrefetchDisabledTables = new CopyOnWriteArraySet<Integer>();

    private int refCount;
    private final boolean managed;    // indicates whether this connection's life cycle is managed
```

建立到 Master 的连接:

```
public HMasterInterface getMaster() throws MasterNotRunningException, ZooKeeperConnectionException {
  if (master != null && master.isMasterRunning()) return master;   // Check if we already have a good master connection
  ensureZookeeperTrackers();          // 初始化ZooKeeper和主节点的地址管理器
  checkIfBaseNodeAvailable();         // 检查节点是否可用
  ServerName sn = null;               // 通过地址管理器可以获取主节点的地址：主机名和端口
  synchronized (this.masterLock) {    // 加锁
    if (master != null && master.isMasterRunning()) return master;
    this.master = null;
    for (int tries = 0; !this.closed && this.master == null && tries < numRetries; tries++) {
        sn = masterAddressTracker.getMasterAddress();
        InetSocketAddress isa = new InetSocketAddress(sn.getHostname(), sn.getPort());   // 建立到Master的连接
        HMasterInterface tryMaster=rpcEngine.getProxy(HMasterInterface.class,HMasterInterface.VERSION,isa,conf, rpcTimeout);
        if (tryMaster.isMasterRunning()) {
          this.master = tryMaster;
          this.masterLock.notifyAll();// 释放锁
          break;                      // 一旦建立到Master的连接，就不需要重试. 只有失败才需要重试
        }
        // Cannot connect to master or it is not running. Sleep & retry  建立到Master的连接失败，等待并重试
        this.masterLock.wait(ConnectionUtils.getPauseTime(this.pause, tries));
    }
    return this.master;
  }
}
```

建立到 HRegionServer 的连接:

```
HRegionInterface getHRegionConnection(final String hostname, final int port, final InetSocketAddress isa, final boolean master){
  if (master) getMaster();
  HRegionInterface server;
  String rsName = null;
  if (isa != null) rsName = Addressing.createHostAndPortStr(isa.getHostName(), isa.getPort());
  else rsName = Addressing.createHostAndPortStr(hostname, port);
  ensureZookeeperTrackers();
  server = this.servers.get(rsName);                  // See if we already have a connection (common case)
  if (server == null) {
    this.connectionLock.putIfAbsent(rsName, rsName);  // create a unique lock for this RS (if necessary)
    synchronized (this.connectionLock.get(rsName)) {  // get the RS lock  同步锁，并没有对servers进行同步
      server = this.servers.get(rsName);              // do one more lookup in case we were stalled above
      if (server == null) {   // definitely a cache miss. establish an RPC for this RegionServer  缓存失效，建立到RegionServer的连接
        InetSocketAddress address = isa != null? isa: new InetSocketAddress(hostname, port); // Only create isa when we need to.
        server = HBaseRPC.waitForProxy(this.rpcEngine, serverInterfaceClass, HRegionInterface.VERSION,
            address, this.conf, this.maxRPCAttempts, this.rpcTimeout, this.rpcTimeout);
        this.servers.put(Addressing.createHostAndPortStr(address.getHostName(), address.getPort()), server);
      }
    }
  }
  return server;
}
```

# Region 定位

用户表数据不断增大时，一张表会分成多个 Region 存储在多个 RegionServer 上.
一个 Region 只能对应一个 RegionServer，一个 RegionServer 可以存放多个 Region.
客户端查询数据时，需要先联系 ZooKeeper，依次访问-ROOT-表 → .META.表 → 最后确定数据的位置.

```
private HRegionLocation locateRegion(final byte [] tableName, final byte [] row, boolean useCache, boolean retry){
  ensureZookeeperTrackers();
  if (Bytes.equals(tableName, HConstants.ROOT_TABLE_NAME)) {
    ServerName servername = this.rootRegionTracker.waitRootRegionLocation(this.rpcTimeout);
    if (servername == null) return null;
    return new HRegionLocation(HRegionInfo.ROOT_REGIONINFO, servername.getHostname(), servername.getPort());
  } else if (Bytes.equals(tableName, HConstants.META_TABLE_NAME)) {
    return locateRegionInMeta(HConstants.ROOT_TABLE_NAME, tableName, row, useCache, metaRegionLock, retry);
  } else { // Region not in the cache - have to go to the meta RS
    return locateRegionInMeta(HConstants.META_TABLE_NAME, tableName, row, useCache, userRegionLock, retry);
  }
}
```

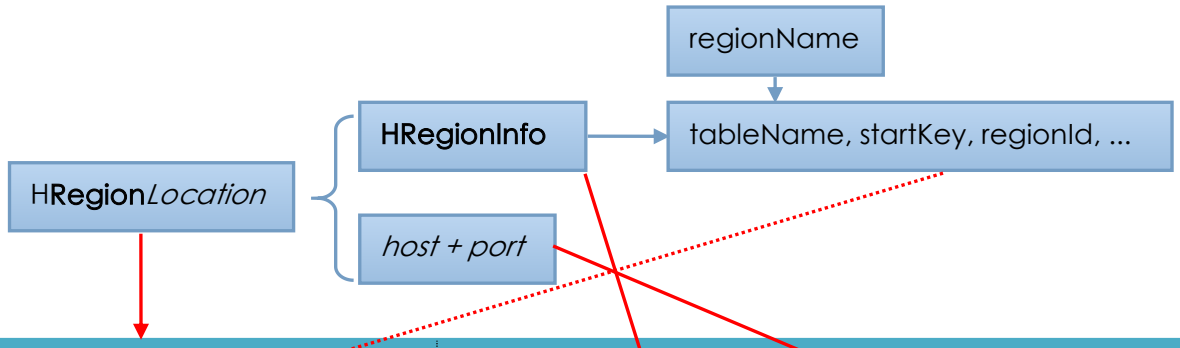
定位一个用户表某一行的 region 位置是个递归调用的过程, locateRegionInMeta 会调用 locateRegion

```
// Search one of the meta tables (-ROOT- or .META.) for the HRegionLocation info that contains the table and row we're seeking.
private HRegionLocation locateRegionInMeta(final byte [] parentTable, final byte [] tableName, final byte [] row,
  boolean useCache, Object regionLockObject, boolean retry) throws IOException {
  HRegionLocation location;
  if (useCache) {  // If we are supposed to be using the cache, look in the cache to see if we already have the region.
    location = getCachedLocation(tableName, row);
    if (location != null) return location;
  }
  int localNumRetries = retry ? numRetries : 1;
  // build the key of the meta region we should be looking for.
  // the extra 9's on the end are necessary to allow "exact" matches without knowing the precise region names.
  byte [] metaKey = HRegionInfo.createRegionName(tableName, row, HConstants.NINES, false);
  for (int tries = 0; true; tries++) {
    if (tries >= localNumRetries) throw new NoServerForRegionException("Unable to find region after " + numRetries + " tries.");
    HRegionLocation metaLocation = null;
      metaLocation = locateRegion(parentTable, metaKey, true, false);  // locate the root or meta region
      if (metaLocation == null) continue;    // If null still, go around again.
      HRegionInterface server = getHRegionConnection(metaLocation.getHostname(), metaLocation.getPort());

      Result regionInfoRow = null;
      // This block guards against two threads trying to load the meta region at the same time.
      // The first will load the meta region and the second will use the value that the first one found.
      synchronized (regionLockObject) {
        // Check the cache again for a hit in case some other thread made the same query while we were waiting on the lock.
        if (useCache) {
          location = getCachedLocation(tableName, row);
          if (location != null) return location;
          // If the parent table is META, we may want to pre-fetch some region info into the global region cache for this table.
          if (Bytes.equals(parentTable, HConstants.META_TABLE_NAME) && (getRegionCachePrefetch(tableName))) {
            prefetchRegionCache(tableName, row);
          }
```

```
                location = getCachedLocation(tableName, row);
                if (location != null) return location;
            } else {    // If we are not supposed to be using the cache, delete any existing cached location so it won't interfere.
                deleteCachedLocation(tableName, row);
            }
            // Query the root or meta region for the location of the meta region
            regionInfoRow = server.getClosestRowBefore(
                    metaLocation.getRegionInfo().getRegionName(), metaKey, HConstants.CATALOG_FAMILY);
        }
        if (regionInfoRow == null) throw new TableNotFoundException(Bytes.toString(tableName));
        byte [] value = regionInfoRow.getValue(HConstants.CATALOG_FAMILY, HConstants.REGIONINFO_QUALIFIER);
        // convert the row result into the HRegionLocation we need!
        HRegionInfo regionInfo = (HRegionInfo) Writables.getWritable(value, new HRegionInfo());
        value = regionInfoRow.getValue(HConstants.CATALOG_FAMILY, HConstants.SERVER_QUALIFIER);
        String hostAndPort = Bytes.toString(value);
        String hostname = Addressing.parseHostname(hostAndPort);
        int port = Addressing.parsePort(hostAndPort);
        location = new HRegionLocation(regionInfo, hostname, port);     // Instantiate the location
        cacheLocation(tableName, location);
        return location;
      }
    }
```

这个方法中涉及到了很多知识点:

1. -ROOT-, .META.也被当做 Table，表结构包括 row-key，列族名称为 info，列名有 regioninfo 和 server 等.
2. HRegionLocation 由 HRegionInfo 和 RegionServer 的 host+port 组成. 以及 HRegionInfo 的组成.
3. HRegionLocation 对应了上面 1.的表结构. 其中 HRegionInfo 对应 info:regioninfo, host+port 对应 info:server
4. 获取缓存 getCachedLocation 和设置缓存 cacheLocation
5. 给定表的某一行 row-key，如何找出离指定行最近的行 RegionServer.getClosestRowBefore
6. 递归调用




| row-key | info | | | history |
|---|---|---|---|---|
| | regioninfo | server | serverstartcode | |
| tableName, startKey, regionIdTS | startKey, endKey, family list... | host + port | | |
| .META., tableA,001,1222989,195856 | | RS1 | | |
| tableA, 001, 1223434 | | RS2 | | |

上面的表结构(-ROOT, .META.)的 row-key 实际上是 HRegionInfo 的 regionName.

递归调用对应了查找 Region 的过程:

```
private HRegionLocation locateRegion(final byte [] tableName, final byte [] row, boolean useCache, boolean retry){
  if (Bytes.equals(tableName, "-ROOT-")) {
    return new HRegionLocation(HRegionInfo.ROOT_REGIONINFO, servername.getHostname(), servername.getPort());   3
  } else if (Bytes.equals(tableName, ".META.")) {
    return locateRegionInMeta("-ROOT-", tableName, row, useCache, metaRegionLock, retry);                       2
  } else { // Region not in the cache - have to go to the meta RS
    return locateRegionInMeta(".META.", tableName, row, useCache, userRegionLock, retry);                       1
  }
}

private HRegionLocation locateRegionInMeta(final byte [] parentTable, final byte [] tableName, final byte [] row,...) {
  byte [] metaKey = HRegionInfo.createRegionName(tableName, row, HConstants.NINES, false);
  HRegionLocation metaLocation = locateRegion(parentTable, metaKey, true, false);       // locate the root or meta region

  HRegionInterface server = getHRegionConnection(metaLocation.getHostname(), metaLocation.getPort());
  Result regionInfoRow = server.getClosestRowBefore(metaLocation.getRegionInfo().getRegionName(), metaKey, "info");
  byte [] value = regionInfoRow.getValue("info", "regioninfo");    → info: regioninfo → HRegionInfo
  HRegionInfo regionInfo = (HRegionInfo) Writables.getWritable(value, new HRegionInfo());
  value = regionInfoRow.getValue("info", "server");                → info: server → RegionServer's host + port
  location = new HRegionLocation(regionInfo, hostname, port);    // Instantiate the location
  return location;                                                                                              6 5 4
```

假设客户端要查询 tableA, row-001 的 HRegionLocation:

locateRegion("tableA", "row-001")

- locateRegionInMeta(".META.", "tableA", "row-001") 1
  - locateRegion(".META.", "tableA,row-001,99999")
    - locateRegionInMeta("-ROOT-", ".META.", "tableA,row-001,99999") 2
      - locateRegion("-ROOT", ".META.,tableA,row-001,99999,99999")
        - return new HRegionLocation(new HRegionInfo(0, "-ROOT-"), host, port) 3
      - host + port → HRegionServer → server.getRow(".META.,tableA,row-001,99999,99999")
      - Result → info:regionInfo → HRegionInfo
      - Result → info:server → RegionServer's host + port
      - return new HRegionLoation(HRegionInfo, host, port)
    - **在-ROOT-表中 row-key~=".META.,tableA,row-001,99999,99999"Row 的 RegionServer U1** 4
  - 返回 RegionServer U1, 对应 U1 的 host+port → server.getRow("tableA,row-001,99999")
  - **在 U1 的.META.表中查找 row-key~="tableA,row-001,99999"的 Row 的 RegionServer U2** 5
- 返回 RegionServer U2 6

最后我们就可以到 U2 的用户表 tableA 中查找 row-key="row-001"的 Row

Table → RegionServer → Region

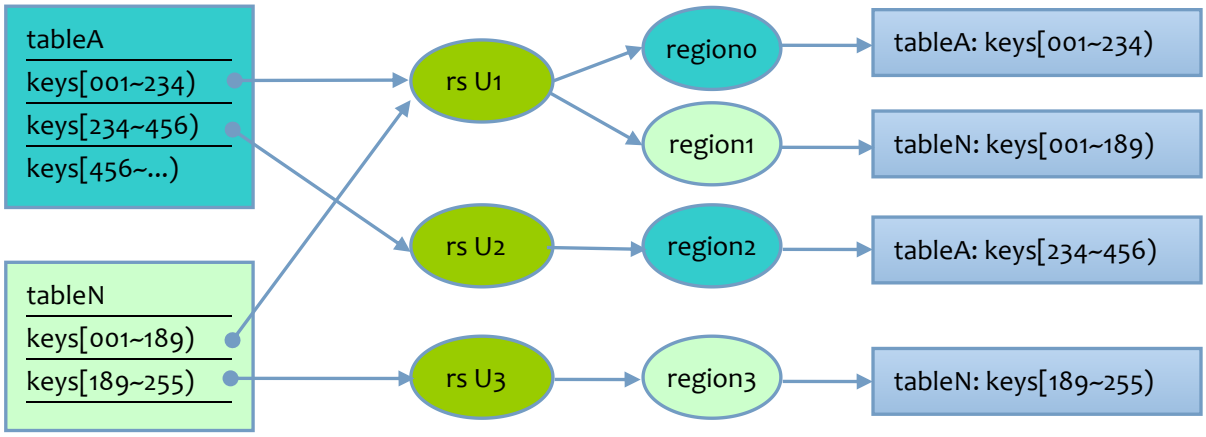
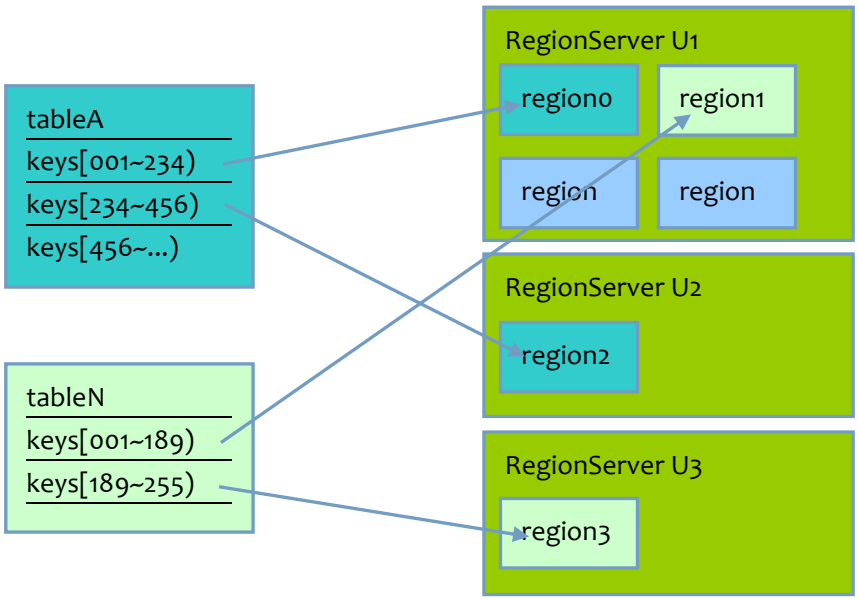
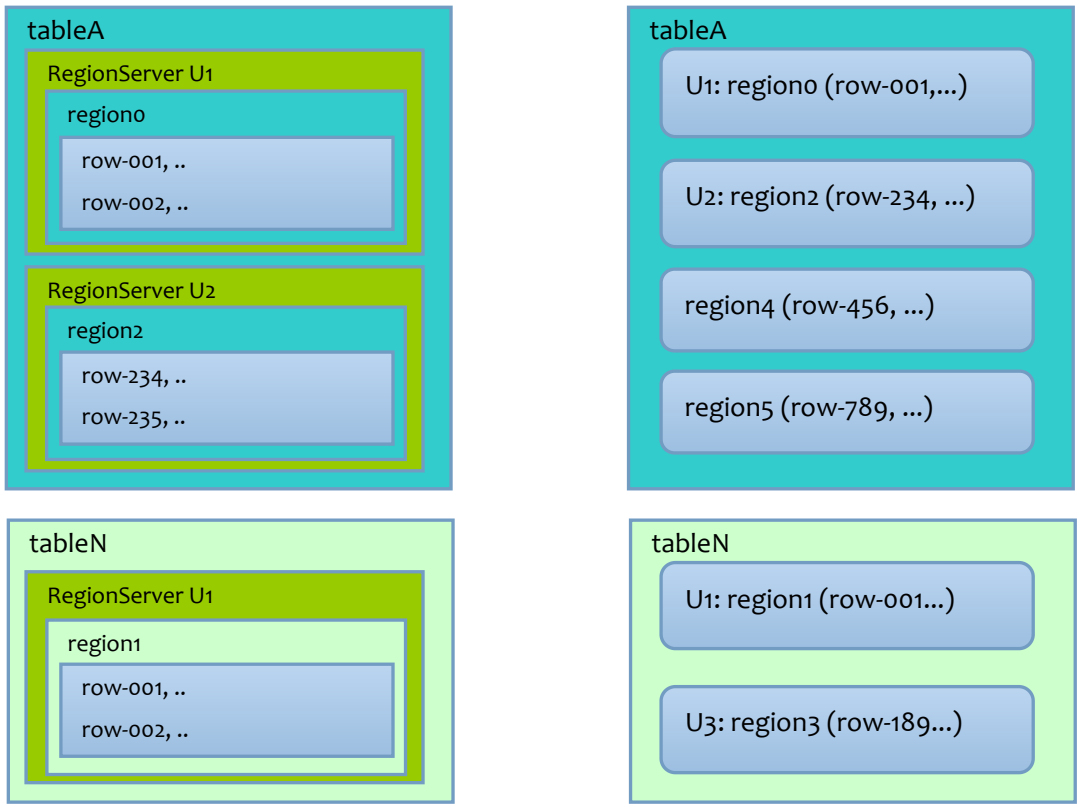


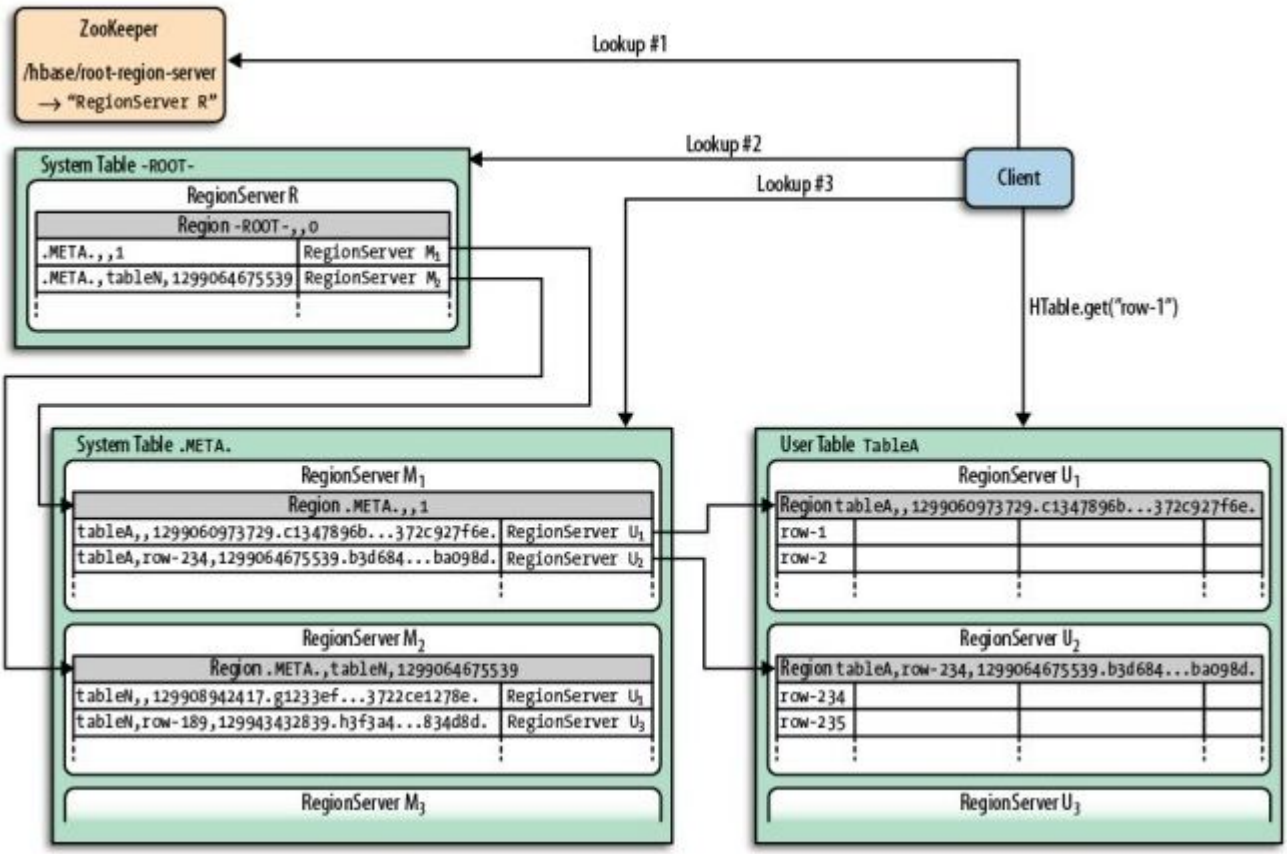

Table → RegionServer(Regions)

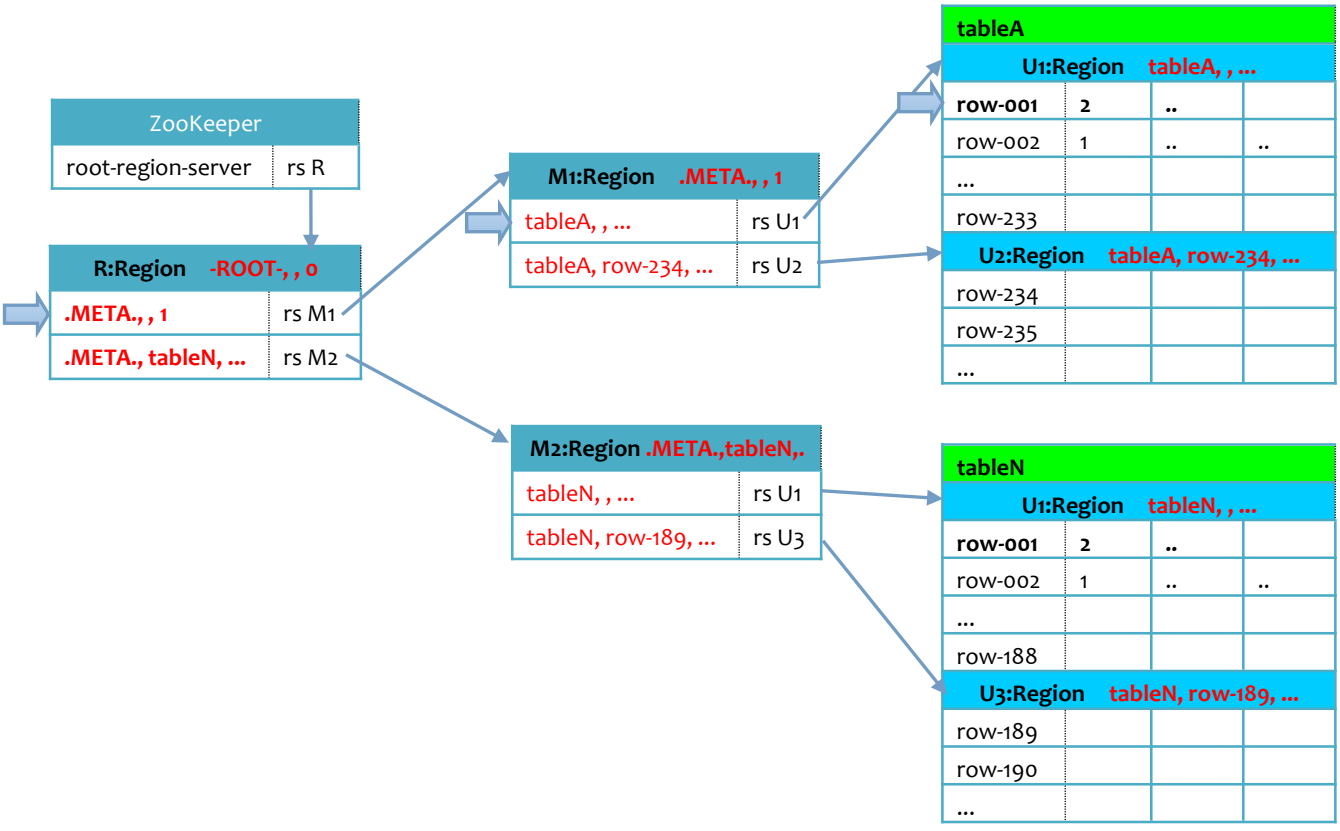



Table(RegionServer → Region)



LookUp Region




[ZooKeeper] **-ROOT-**, 0, ...

| row-key | info | |
|---|---|---|
| | regioninfo | server |
| **.META., tableA, row-001, 99999,99999** | tableA, row-001, row-234, ... | RegionServer U1 |
| **.META., tableA, row-234, ...** | tableA, row-234, row-456, ... | RegionServer U2 |
| **...** | ... | ... |
| **.META, tableN, row-189, ...** | tableN, row-001, row-189, ... | RegionServer U1 |

4

[RegionServer U1] **.META.**, tableA, row-001, ...

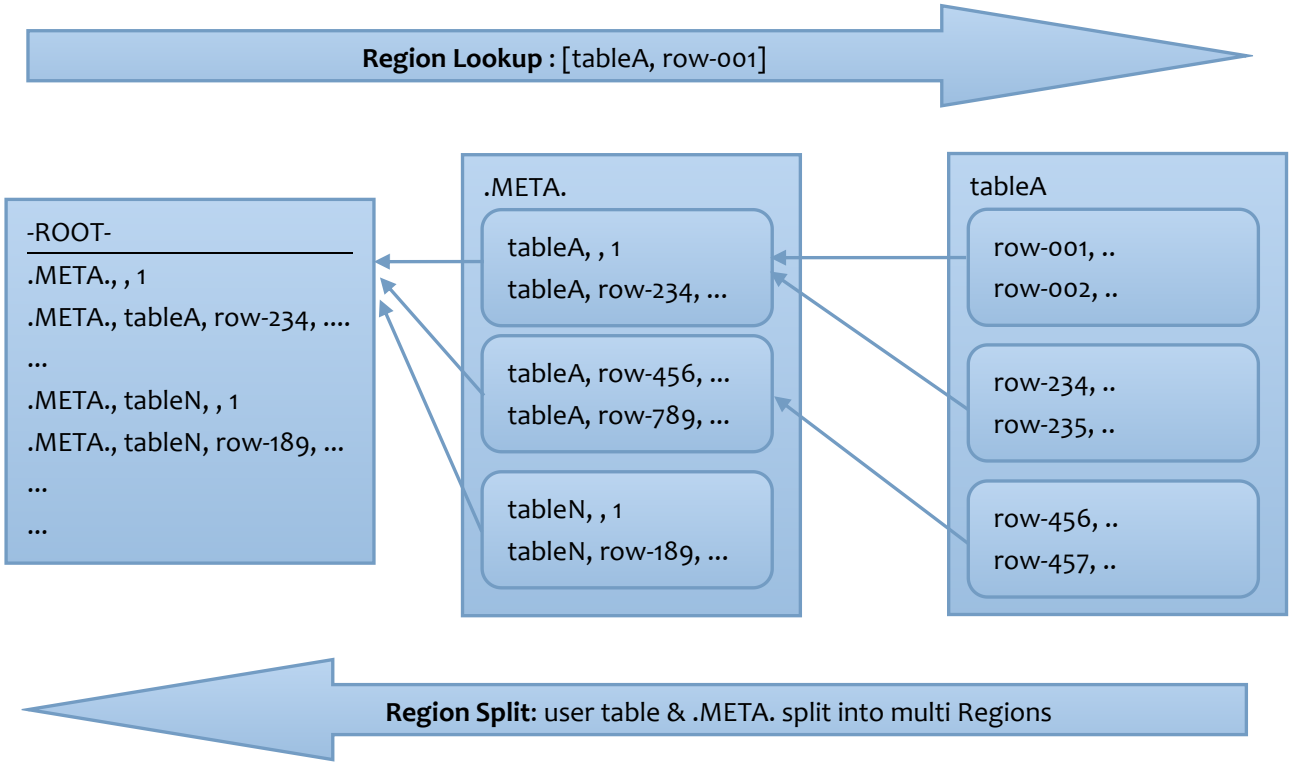
| row-key | info | |
|---|---|---|
| | regioninfo | server |
| **tableA,row-001,99999** | row-001, row-234, ... | RegionServer U2 |

5

[RegionServer U2] **tableA**, row-001, ... 6

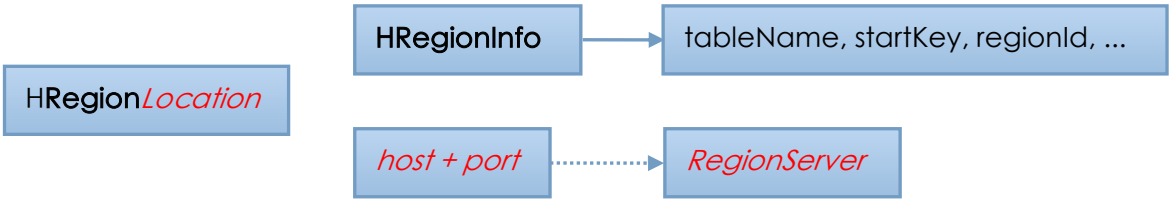
| row-key | family-01 | |
|---|---|---|
| | qual-01 | qual-02 |
| **row-001** | val-01 | |
| **row-002** | | val-02 |
| **...** | ... | ... |
| **row-233** | ... | |

用户表可以分成多个 Region，用户表的 Region 信息存储在.META.表中.
.META.表如果数据不断增大，也可以分成多个 Region, .META.的 region 信息存储在-ROOT-表中
-ROOT-表不会被分成多个 Region 的. HBase 认为-ROOT-表存储的信息不会大到能再分成 Region.



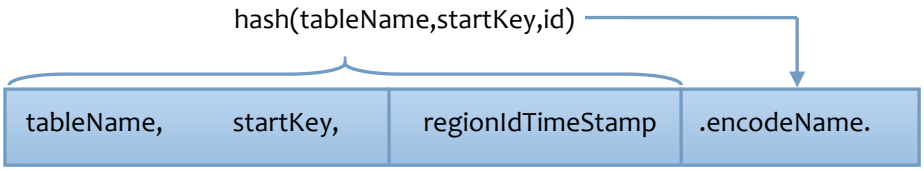
# HRegionLocation & HRegionInfo

前面我们已经图示了 HRegionLocation 由 Region: HRegionInfo 和 Location: host+port 组成.
表示了 Region 的位置信息, Region 是 RegionServer 上的数据结构，因此 host+port 能够区分哪个 RegionServer.
比较两个 HRegionLocation 是否相等，只要比较 host+port 即可．也就是说同一个 RegionServer 上
即使 HRegionInfo 不同．这两个 HRegionLocation 也是相同的．正如类名所示，比较的是 Region 的位置,而不是信息.




HRegionInfo 表示的是 Region 信息．前面说过一个 Table 由多个 Region 组成.
因此一个 Region 要能够代表 Table 的一部分信息．反过来 Table 要分成多个 Region，需要 Region 记住哪些信息.

假设 tableA 的 row-key 从 001 ~ 456，要等分成 2 份 Region．首先每个 Region 都应该知道自己对应的是 tableA.
每个 Region 都应该记住自己在 Table 中的起始位置 → startKey，以及结束位置 → endKey
为了快速找到 Table 的每个 Region，还要给每个 Region 分配一个 id: regionIdTimeStamp.




HRegionInfo 的 regionName 的组成格式: <tableName>,<startKey>,<regionIdTimeStamp>.<encodeName>.
注意: -ROOT-和.META.的 row-key 就是 regionName, info:regionInfo 才包括完整的 HRegionInfo 信息.

```
public class HRegionLocation implements Comparable<HRegionLocation> {
  private final HRegionInfo regionInfo;
  private final String hostname;
  private final int port;

  public HRegionLocation(HRegionInfo regionInfo, final String hostname, final int port) {
    this.regionInfo = regionInfo;
    this.hostname = hostname;
    this.port = port;
  }
}

/** HRegion information. Contains HRegion id, start and end keys, a reference to this HRegions' table descriptor, etc. */
public class HRegionInfo extends VersionedWritable implements WritableComparable<HRegionInfo> {
  /** HRegionInfo for root region */
  public static final HRegionInfo ROOT_REGIONINFO = new HRegionInfo(0L, Bytes.toBytes("-ROOT-"));
  /** HRegionInfo for first meta region */
  public static final HRegionInfo FIRST_META_REGIONINFO = new HRegionInfo(1L, Bytes.toBytes(".META."));

  private byte [] endKey = HConstants.EMPTY_BYTE_ARRAY;
  private long regionId = -1;
  private transient byte [] regionName = HConstants.EMPTY_BYTE_ARRAY;
  private String regionNameStr = "";
```

```
  private boolean split = false;
  private byte [] startKey = HConstants.EMPTY_BYTE_ARRAY;
  private volatile String encodedName = NO_HASH;
  private byte [] encodedNameAsBytes = null;
  private byte[] tableName = null;      // Current TableName

  public HRegionInfo(final byte[] tableName, final byte[] startKey, final byte[] endKey, final boolean split, final long regionid){
     this.tableName = tableName.clone();
     this.offLine = false;
     this.regionId = regionid;
     this.regionName = createRegionName(this.tableName, startKey, regionId, true);
     this.regionNameStr = Bytes.toStringBinary(this.regionName);
     this.split = split;
     this.endKey = endKey == null? HConstants.EMPTY_END_ROW: endKey.clone();
     this.startKey = startKey == null? HConstants.EMPTY_START_ROW: startKey.clone();
     this.tableName = tableName.clone();
     setHashCode();
  }

  public static byte [] createRegionName(final byte [] tableName, final byte [] startKey, final byte [] id, boolean newFormat) {
     byte [] b = new byte [tableName.length + 2 + id.length +       // tableName,startKey,regionIdTimeStamp.encodeName.
         (startKey == null? 0: startKey.length) + (newFormat ? (MD5_HEX_LENGTH + 2) : 0)];    // +2表示分隔符: 2个,和2个.
     int offset = tableName.length;
     System.arraycopy(tableName, 0, b, 0, offset);
     b[offset++] = DELIMITER;
     if (startKey != null && startKey.length > 0) {
        System.arraycopy(startKey, 0, b, offset, startKey.length);
        offset += startKey.length;
     }
     b[offset++] = DELIMITER;
     System.arraycopy(id, 0, b, offset, id.length);
     offset += id.length;
     if (newFormat) {
        String md5Hash = MD5Hash.getMD5AsHex(b, 0, offset);
        byte [] md5HashBytes = Bytes.toBytes(md5Hash);
        b[offset++] = ENC_SEPARATOR;
        System.arraycopy(md5HashBytes, 0, b, offset, MD5_HEX_LENGTH);      // now append the bytes '.<encodedName>.' to the end
        offset += MD5_HEX_LENGTH;
        b[offset++] = ENC_SEPARATOR;
     }
     return b;
  }

  public KVComparator getComparator() {
     return isRootRegion()? KeyValue.ROOT_COMPARATOR: isMetaRegion()? KeyValue.META_COMPARATOR: KeyValue.COMPARATOR;
  }
}
```

# CacheLocation

```
// Map of table to table HRegionLocations.    The table key is made by doing a Bytes#mapKey(byte[]) of the table's name.
private final Map<Integer, SoftValueSortedMap<byte [], HRegionLocation>>
  cachedRegionLocations = new HashMap<Integer, SoftValueSortedMap<byte [], HRegionLocation>>();

private void cacheLocation(final byte [] tableName, final HRegionLocation location) {
  byte [] startKey = location.getRegionInfo().getStartKey();
  Map<byte [], HRegionLocation> tableLocations = getTableLocations(tableName);
  boolean hasNewCache = false;
  synchronized (this.cachedRegionLocations) {
    cachedServers.add(location.getHostnamePort());
    hasNewCache = (tableLocations.put(startKey, location) == null);
  }
}
private SoftValueSortedMap<byte [], HRegionLocation> getTableLocations(final byte [] tableName) {
  // find the map of cached locations for this table
  Integer key = Bytes.mapKey(tableName);
  SoftValueSortedMap<byte [], HRegionLocation> result;
  synchronized (this.cachedRegionLocations) {
    result = this.cachedRegionLocations.get(key);
    if (result == null) {        // if tableLocations for this table isn't built yet, make one
      result = new SoftValueSortedMap<byte [], HRegionLocation>(Bytes.BYTES_COMPARATOR);
      this.cachedRegionLocations.put(key, result);
    }
  }
  return result;
}
```

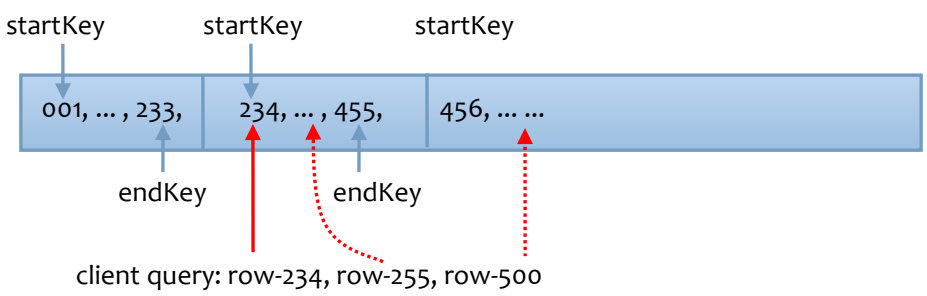
客户端会将 tableName 以及对应的 HRegionLocation 缓存起来. 这样就不用每次都要重新获取.

cachedRegionLocations 的组成: <hash(tableName), SoftValueSortedMap<startKey, HRegionLocation>>

key 使用 Integer 的 hash 值这样检索起来速度会快. SoftValueSortedMap 内部使用了 SoftReference 软引用.

| key | SoftValueSortedMap<startKey, HRegionLocation> | | 备注 |
|---|---|---|---|
| hash(tableA) | row-001 | tableA, row-001, ...., RegionServer-1 | [startKey, endKey)=[001, 234) |
| | row-234 | tableA, row-234, ...., RegionServer-2 | [startKey, endKey)=[234, 456) |
| | row-456 | tableA, row-456, ...., RegionServer-3 | [startKey, endKey)=[456, ...) |
| hash(tableN) | row-001 | tableA, row-001, ...., RegionServer-1 | [startKey, endKey)=[001, 189) |
| | row-189 | tableA, row-189, ...., RegionServer-3 | [startKey, endKey)=[189, ...) |

假设上述表格的信息都缓存起来. 当客户端要查询 tableA, row-234 的 HRegionLocation, 可以从 Map 中很快查出来. 但是如果客户端要查询的是 tableA, row-255/500 的 HRegionLocation. 由于不在 Map 中, 那么是不是要重新查询?



1. row-234，因为 Map 的 key 是 startKey，而客户端要查询的 row 正好是 Map 中的 startKey=row-234.
2. row-500，因为 Map 中不存在 startKey=row-500 的元素．所以我们找比 row-500 小的 startKey，
   找到 startKey=456，而且这个 RegionLocation 对应的 RegionInfo 的 endKey 为空，
   我们就可以认为 row-500 在 startKey=row-456 的 Region 里面
3. row-255．同上先找到比 row-255 小的 startKey，找到 row-234.
   同时因为这个 RegionInfo 有 endKey，那么 endKey 必须大于我们要查询的 row-255.
   因为 row-255 ∈ [row-234, row-456)，所以我们也认为 row-255 在[row-234, row-456)对应的 Region 里
4. 假设我们要查询 row-000，由于 row-000 不在和 startKey 相等的 SoftValueSoftedMap 中，
   而且比 row-000 小的 startKey 也都不存在，我们就认为没有找到 row-000 对应的 Region.

```
HRegionLocation getCachedLocation(final byte [] tableName, final byte [] row) {
  SoftValueSortedMap<byte [], HRegionLocation> tableLocations = getTableLocations(tableName);
  // start to examine the cache. we can only do cache actions if there's something in the cache for this table.
  if (tableLocations.isEmpty()) return null;
  HRegionLocation possibleRegion = tableLocations.get(row);
  if (possibleRegion != null) return possibleRegion;              // 就在缓存中，直接返回

  possibleRegion = tableLocations.lowerValueByKey(row);   // 查询比row小的Map中的startKey
  if (possibleRegion == null) return null;                        // 没有比row更小的，那就是真没有了
  // make sure that the end key is greater than the row we're looking for，确保候选的Location的endKey > 我们要查找的row
  // otherwise the row actually belongs in the next region, not this one. 否则(endKey < row)，这个Region就不是我们想要的
  // the exception case is when the endkey is HConstants.EMPTY_END_ROW，如果endKey是空的，也满足
  // signifying that the region we're checking is actually the last region in the table. 通常是Table中的最后一个Region
  byte[] endKey = possibleRegion.getRegionInfo().getEndKey();
  if (Bytes.equals(endKey, HConstants.EMPTY_END_ROW) ||
      KeyValue.getRowComparator(tableName).compareRows(endKey, 0, endKey.length, row, 0, row.length) > 0) {
    return possibleRegion;
  }
  return null;    // Passed all the way through, so we got nothin - complete cache miss
}
```

## 预获取缓存

当 parentTable=.META.时可以预先获取一部分的缓存．为什么是.META.表，而不是-ROOT-表?
因为.META.保存的是用户自定义的表的 Region 信息．假设 tableA 分成多个 Region.
而用户查询时可能会查询 tableA 的多个 Region，比如全表扫描就需要获取到 table 的每个 Region.
什么时候停止预获取? 当.META.的记录不是同一个 Table 时停止预获取．因为通常我们的查询是针对同一个表的.
-ROOT-表保存的是所有.META.的 Region 信息．显然这个层次太高级了．查询时主要针对的是.META.表

| row-key | info: regionInfo | info: server |
|---|---|---|
| tableA, , ... | [row-001, row-234) | RegionServer U1 |
| tableA, row-234, ... | [row-234, row-456) | RegionServer U2 |
| tableA, row-456, ... | [row-456, row-789) | RegionServer U3 |
| tableA, row-789, ... | [row-789, row-1000) | RegionServer U4 |
| tableA, row-1000, ... | [row-1000, row-1234) | RegionServer U5 |
| tableA, row-1234, ... | [row-1234, ...) | RegionServer U1 |
| ... | | |
| tableN, , ... | [row-001, row-189) | RegionServer U1 |
| tableN, row-189, ... | [row-189, ...) | RegionServer U3 |

```
/* Search .META. for the HRegionLocation info that contains the table and row we're seeking.
  * It will prefetch certain number of regions info and save them to the global region cache. */
private void prefetchRegionCache(final byte[] tableName, final byte[] row) {
    // Implement a new visitor for MetaScanner, and use it to walk through the .META.
    MetaScannerVisitor visitor = new MetaScannerVisitorBase() {
      public boolean processRow(Result result) throws IOException {
          byte[] value = result.getValue(HConstants.CATALOG_FAMILY, HConstants.REGIONINFO_QUALIFIER);
          HRegionInfo regionInfo = null;
          if (value != null) {
            regionInfo = Writables.getHRegionInfo(value);      // convert the row result into the HRegionLocation we need!
            if (!Bytes.equals(regionInfo.getTableName(), tableName)) {  // possible we got a region of a different table...
              return false;                                    // stop scanning
            }
            if (regionInfo.isOffline()) return true;  // don't cache offline regions
            value = result.getValue(HConstants.CATALOG_FAMILY, HConstants.SERVER_QUALIFIER);
            if (value == null) return true;                  // don't cache it
            final String hostAndPort = Bytes.toString(value);
            String hostname = Addressing.parseHostname(hostAndPort);
            int port = Addressing.parsePort(hostAndPort);
            HRegionLocation loc = new HRegionLocation(regionInfo, hostname, port);    // instantiate the location
            cacheLocation(tableName, loc);  // cache this meta entry
          }
          return true;
      }
    };
    // pre-fetch certain number of regions info at region cache.
    MetaScanner.metaScan(conf, this, visitor, tableName, row, this.prefetchRegionLimit, HConstants.META_TABLE_NAME);
}
```

MetaScanner 和 MetaScannerVisitor 用于扫描.META.表的 Region 信息. 获取.META.表和获取普通表的行一样. MetaScannerVisitor 是一个回调函数. 在 metaScan 中如果获取到一条记录, 就调用 processRow 回调函数将 Result 对应的.META.表结构的一行记录(info 列族)转成缓存需要的 HRegionLocation 加入到客户端缓存中.

# Put

在本章一开始的示例中创建了一个 Put 对象，并且调用 table.put()方法将 put 存储到 HBase 的表中.

| row-key | ts | family-01 | | family-02 | |
|---|---|---|---|---|---|
| | | col-01 | col-02 | col-01 | col-02 |
| row-001 | 2 | val-01 | | | |
| | 1 | value-001 | | | |
| | 1 | | val-02 | | |
| | 1 | | | val-01 | |
| | 1 | | | | val-02 |
| row-002 | 1 | val-01 | | | |
| | 1 | | val-02 | | |

HBase 的一个单元格由<row-key, family, qualifier, ts, value>=<row-002, family-01, col-01, 2, val-01>确定.
HBase 的一行数据(同一个 row-key)可以由多个版本组成. 比如要获取表中 row-key=row-001 的数据，在 HBase 的 Shell 中将会显示 5 行数据. 获取 row-key=row-001, column=family-01:col-01 的有 2 个数据单元格.

抽象基类 Mutation 包含了一行数据共同的字段: 行键:byte[] row，版本号:ts，锁 ID: lockId 以及最重要的 familyMap. familyMap 的 key 是 familyName, value 是一系列的 KeyValue. 即每一个数据单元格.

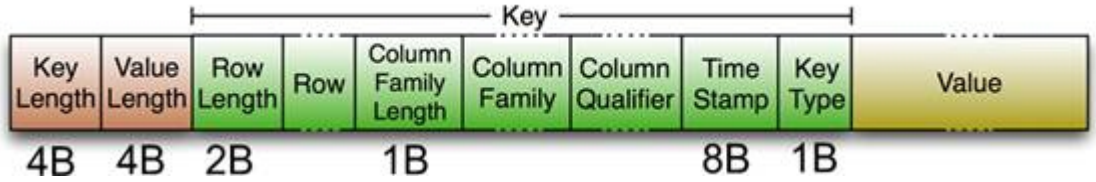
```
public abstract class Mutation extends OperationWithAttributes implements Row {
  protected byte [] row = null;
  protected long ts = HConstants.LATEST_TIMESTAMP;
  protected long lockId = -1L;
  protected boolean writeToWAL = true;
  protected Map<byte [], List<KeyValue>> familyMap =new TreeMap<byte [], List<KeyValue>>(Bytes.BYTES_COMPARATOR);
}
```

在客户端调用 API 时插入一格数据，需要调用 Put 的 add 方法(相当于往数据块中添加一行数据):

```
  /** Add the specified column and value, with the specified timestamp as its version to this Put operation.
   * 1. 获取指定列族family(add传入)从familyMap中获取已经存在的所有List<KeyValue>     [map.get(k)->list]
   * 2. 根据传入的列族family，列名qualifier，版本ts，值value构建一个KeyValue          [new]
   * 3. 将新创建的KeyValue加入List中                                              [list.add(new)]
   * 4. 重新将新的List集合放入familyMap中                                         [map.put(k,list)]      */
  public Put add(byte [] family, byte [] qualifier, long ts, byte [] value) {
    List<KeyValue> list = getKeyValueList(family);
    KeyValue kv = createPutKeyValue(family, qualifier, ts, value);
    list.add(kv);
    familyMap.put(kv.getFamily(), list);
    return this;
  }
  private List<KeyValue> getKeyValueList(byte[] family) {
    List<KeyValue> list = familyMap.get(family);
    if(list == null) list = new ArrayList<KeyValue>(0);
    return list;
  }
  private KeyValue createPutKeyValue(byte[] family, byte[] qualifier, long ts, byte[] value) {
    return new KeyValue(this.row, family, qualifier, ts, KeyValue.Type.Put, value);
  }
```

# KeyValue

KeyValue 的格式：由 row-key, family, qualifier, ts, type, value 组成．是 HBase 最底层的存储单元.




KeyValue 内部使用 byte[] bytes 来填充所有的信息．使用字节数组的好处还有:

1. 允许存储任意类型的数据，并且可以有效地只存储所需的字节，保证了最少的内部数据结构开销
2. 每个字节数组都有 offset 和 length，允许用户提交一个已经存在的字节数组．并进行效率很高的字节级别操作

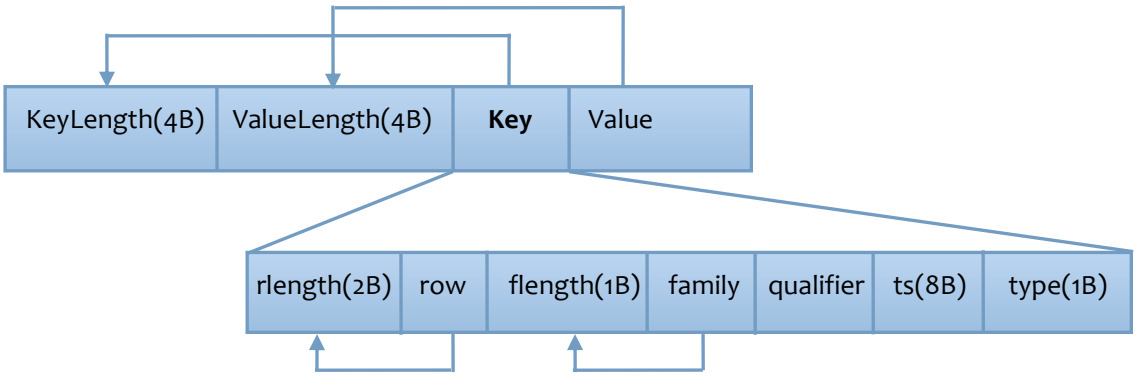
```
public class KeyValue implements Writable, HeapSize {
  private byte [] bytes = null;
  private int offset = 0;
  private int length = 0;

  public KeyValue(final byte [] bytes, final int offset, final int length) {
    this.bytes = bytes;
    this.offset = offset;
    this.length = length;
  }
  public byte [] getBuffer() { return this.bytes; }
  public int getOffset()       { return this.offset; }
  public int getLength()       {return length; }

  public KeyValue(final byte[] row, final byte[] family, final byte[] qualifier, final long timestamp, Type type, final byte[] value) {
    this(row, family, qualifier, 0, qualifier==null ? 0 : qualifier.length, timestamp, type, value, 0, value==null ? 0 : value.length);
  }
  public KeyValue(final byte [] row, final int roffset, final int rlength, final byte [] family, final int foffset, final int flength,
        final byte [] qualifier, final int qoffset, final int qlength, final long timestamp, final Type type,
        final byte [] value, final int voffset, final int vlength) {
    this.bytes = createByteArray(row, roffset, rlength, family, foffset, flength, qualifier, qoffset, qlength, timestamp, type,
          value, voffset, vlength);
    this.length = bytes.length;
    this.offset = 0;
  }
  static byte [] createByteArray(final byte [] row, final int roffset, final int rlength, final byte [] family, final int foffset, int flength,
        final byte [] qualifier, final int qoffset, int qlength, final long timestamp, final Type type,
        final byte [] value, final int voffset, int vlength) {
    flength = family == null ? 0 : flength;            // Family length
    qlength = qualifier == null ? 0 : qlength;         // Qualifier length
    // Key length 12=2+1+8+1=(short)rlength'+(byte)flength'+(long)ts+(byte)type
    long longkeylength = KEY_INFRASTRUCTURE_SIZE + rlength + flength + qlength;
    int keylength = (int)longkeylength;
    vlength = value == null? 0 : vlength;              // Value length
    // Allocate right-sized byte array.
    byte [] bytes = new byte[KEYVALUE_INFRASTRUCTURE_SIZE + keylength + vlength];
    // Write key, value and key row length.
    int pos = 0;
```

```
    pos = Bytes.putInt(bytes, pos, keylength);                  // key length [4]
    pos = Bytes.putInt(bytes, pos, vlength);                    // value length [4]
    pos = Bytes.putShort(bytes, pos, (short)(rlength & 0x0000ffff));  // row length [2]
    pos = Bytes.putBytes(bytes, pos, row, roffset, rlength);    // row [rlength]
    pos = Bytes.putByte(bytes, pos, (byte)(flength & 0x000000ff));    // family length [1]
    if(flength != 0) {
      pos = Bytes.putBytes(bytes, pos, family, foffset, flength);     // column family [flength]
    }
    if(qlength != 0) {
      pos = Bytes.putBytes(bytes, pos, qualifier, qoffset, qlength);  // column qualifier [qlength]
    }
    pos = Bytes.putLong(bytes, pos, timestamp);                 // ts [8]
    pos = Bytes.putByte(bytes, pos, type.getCode());            // type [1]
    if (value != null && value.length > 0) {
      pos = Bytes.putBytes(bytes, pos, value, voffset, vlength);      // value [vlength]
    }
    return bytes;
  }
```

Bytes 提供的 putXXX(bytes, offset, value)方法会将 value 放到 bytes 以 offset 开始的位置.
通过返回值 pos 不断后移指针，将 row-key, family, qualifier, ts, type, value 以及长度信息放入整个 bytes 数组.
createByteArray 方法填充 bytes 数组，需要计算 KeyValue 占用的大小=8 + KeyLength + ValueLength.

**KEYVALUE_INFRASTRUCTURE_SIZE**(KeyValue 基础大小)=keyLength(4B) + valueLength(4V) = =8B
**KEY_INFRASTRUCTURE_SIZE**(Key 的基础大小)=rlength(2B) + flength(1B) + ts(8B) + type(1B)=12B
注意：上面计算基础大小的值，是每个字段占用的字节数，而不是字段本身在 bytes 数组中的值.

填充字节数组 byte[] bytes 时，因为已知传入的 row, family, qualifier, ts, type, value.
所以可以计算出每一部分的长度，将其中的 row+family+qualifier+ts+type 组成 key，其长度填充到 KeyLength.




相对填充字节数组的是获取字节数组对应字段的值.
下面的 getLength()传入参数 byte[] bytes 和 offset，而不是默认的 getLength()无参方法直接返回 length.
给定字节数组，以及偏移量 offset，计算此字节数组的长度=KeyValue 基础大小(8B)+keyLength+valueLength.
keyLength 和 valueLength 需要通过 Bytes.toInt(bytes, offset)方法来读取(从 bytes 的 offset 开始读取一个 int 值).

```
  private static int getLength(byte [] bytes, int offset) {
    return ROW_OFFSET +   // (int)key-length + (int)value-length=8B
        Bytes.toInt(bytes, offset) +   // key-length's value  从bytes中的offset开始读取，一共读取int=4bytes
        Bytes.toInt(bytes, offset + Bytes.SIZEOF_INT); // value-length's value, offset+4，因为上一次读取的keyLength长度=4B
  }
```

其他字段的值的获取类似，都是通过 Bytes.toXXX(bytes, offset)从 bytes 的 offset 开始读取指定的字节数.

# put 存储

```
private final ArrayList<Put> writeBuffer = new ArrayList<Put>();
private long writeBufferSize;
private boolean autoFlush;
private long currentWriteBufferSize;

public void put(final Put put) throws IOException {
   doPut(put);
   if (autoFlush)    flushCommits();
}
public void put(final List<Put> puts) throws IOException {
   for (Put put : puts)    doPut(put);
   if (autoFlush)    flushCommits();
}

private void doPut(Put put) throws IOException{
   validatePut(put);
   writeBuffer.add(put);
   currentWriteBufferSize += put.heapSize();
   if (currentWriteBufferSize > writeBufferSize)    flushCommits();
}
public void flushCommits() throws IOException {
      Object[] results = new Object[writeBuffer.size()];
      try {
         this.connection.processBatch(writeBuffer, tableName, pool, results);
      } catch (InterruptedException e) {
         throw new IOException(e);
      } finally {
         // mutate list so that it is empty for complete success，如果所有记录都成功了，writeBuffer为空
         // or contains only failed records results are returned in the same order as the requests in list walk the list backwards,
         // so we can remove from list without impacting the indexes of earlier members
         for (int i = results.length - 1; i>=0; i--) {
            if (results[i] instanceof Result) { // 操作成功的Put记录，从writeBuffer中移除．失败的记录会存储在writeBuffer中.
               writeBuffer.remove(i);          // successful Puts are removed from the list here.
            }
         }
      }
}
```

调用 HConnection.processBatch()会调用到实现类 HConnectionImplementation 的 processBatchCallback():
这个方法比较长，我们一步一步分解，批处理可能失败，如果失败要进行重试.

```
   public <R> void processBatchCallback(List<? extends Row> list, byte[] tableName,
         ExecutorService pool, Object[] results, Batch.Callback<R> callback) {
      HRegionLocation [] lastServers = new HRegionLocation[results.length];
      List<Row> workingList = new ArrayList<Row>(list);   // 新建一个工作列表，而不是使用参数中的list
      boolean retry = true;
      int actionCount = 0;  // count that helps presize actions array
      for (int tries = 0; tries < numRetries && retry; ++tries) {
         if (tries >= 1) {       // sleep first, if this is a retry
```

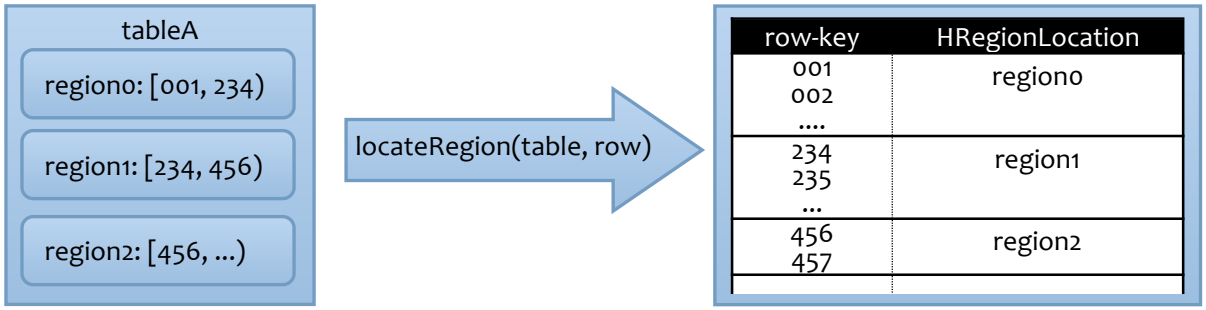
```
            long sleepTime = ConnectionUtils.getPauseTime(this.pause, tries);
            Thread.sleep(sleepTime);
          }
          // ..........................................
        }
```

# 第 1 步：将每个 Put 分解(break up)到对应的 RegionServer 上

```
        // step 1: break up into regionserver-sized chunks and build the data structs
        Map<HRegionLocation, MultiAction<R>> actionsByServer = new HashMap<HRegionLocation, MultiAction<R>>();
        for (int i = 0; i < workingList.size(); i++) {
          Row row = workingList.get(i);          // Row实现类，比如writeBuffer里的每个Put对象
          if (row != null) {
            HRegionLocation loc = locateRegion(tableName, row.getRow());
            byte[] regionName = loc.getRegionInfo().getRegionName();

            MultiAction<R> actions = actionsByServer.get(loc);
            if (actions == null) {
              actions = new MultiAction<R>();
              actionsByServer.put(loc, actions);
            }
            Action<R> action = new Action<R>(row, i);    // 每个Put对象对应一个Action动作
            lastServers[i] = loc;
            actions.add(regionName, action);
          }
        }
```

actionsByServer 实际上数据结构如下: Map< HRegionLocation, Map< regionName, List<Action> > >

每个被加入到 writeBuffer 的 Put 对象，都会首先根据 tableName 和构造 Put 时的 row-key 确定 HRegionLocation. locateRegion()方法前面已经分析过了，即要确定每个 Put 对象存放到哪个 RegionServer 上.




Map< HRegionLocation, Map< regionName, List<Action> > > **actionsByServer**

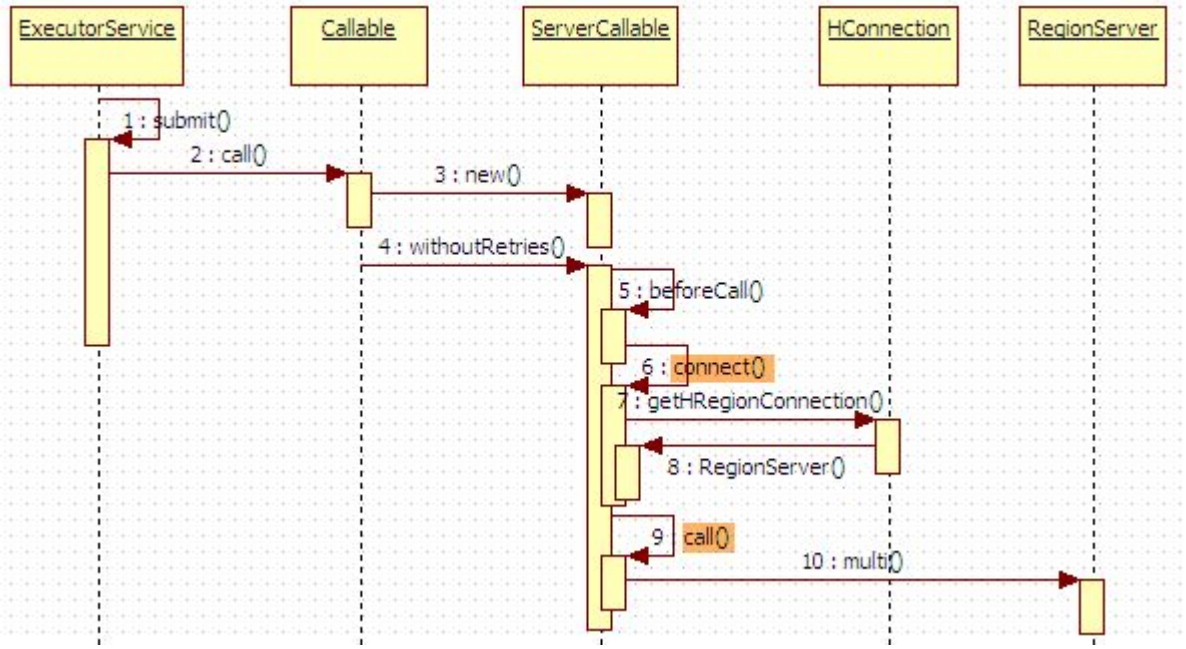
| HRegionLocation | regionName | List<Action> | RegionServer |
|---|---|---|---|
| region0 | tableA, 001, ... | put(001)<br>put(002)<br>... | RegionServer1 |
| region1 | tableA, 234, ... | put(234)<br>put(235)<br>... | RegionServer2 |
| region2 | tableA, 456, ... | put(456)<br>put(457)<br>... | RegionServer1 |

# 第 2 步：发起请求

```
// step 2: make the requests
Map<HRegionLocation, Future<MultiResponse>> futures = new HashMap(actionsByServer.size());
for (Entry<HRegionLocation, MultiAction<R>> e: actionsByServer.entrySet()) {
  futures.put(e.getKey(), pool.submit(createCallable(e.getKey(), e.getValue(), tableName)));
}
```

对每个不同的 HRegionLocation(含有 HRegionServer 的 host 和 port)，都发起一次 RPC 调用.
使用 Java 并发编程的 Future 以及连接池提交任务. poll.submit(**Callable**)会回调 Callable 的 call 方法
在 call()里面调用了 **ServerCallable**.withoutRetries() → 首先调用匿名 ServerCallable 的 connect()做准备工作，
最终调用到匿名 ServerCallable 的 call()方法 → 根据 connect()准备的 RegionServer 调用 RPC 方法()

```
private <R> Callable<MultiResponse> createCallable(final HRegionLocation loc, final MultiAction<R> multi, final byte [] tableName) {
  final HConnection connection = this;
  return new Callable<MultiResponse>() {
   public MultiResponse call() throws IOException {
     ServerCallable<MultiResponse> callable = new ServerCallable<MultiResponse>(connection, tableName, null) {
         public MultiResponse call() throws IOException {
           return server.multi(multi);
         }
         public void connect(boolean reload) throws IOException {
           server = connection.getHRegionConnection(loc.getHostname(), loc.getPort());
         }
      };
     return callable.withoutRetries();
   }
  };
}
```




注意：和 Get 不同，put 已经知道了 HRegionLocation，因此 connect()时只需根据 HRegionLocation 获取 RegionServer.

# ServerCallable

ServerCallable 实现了 Callable 接口，通常的使用方式是：建立一个 ServerCallable 对象 callable，在创建对象的同时实现 call()，并在 call()中写入自己的业务逻辑，然后调用 callable.withRetries()。 因为 ServerCallable 的 withRetries()会依次调用 beforeCall() → connect() → call()。即最终调用到创建自定义 ServerCallable 时的 call()。
实际上因为 ServerCallable 是个抽象类(没有 Callable 的 call 方法)，因此实现类(或匿名类)必须实现 call()

```
new ServerCallable<Result>(connection, tableName, row, operationTimeout) {
    public Result call() throws IOException {
        return server.XXX();
    }
}.withRetries();
```

ServerCallable 从名字我们知道是 Server 的回调对象。在 HBase 中 Server 可以是 HMaster 或 HRegionServer。
根据业务场景，put/delete/get 操作都是直接针对 RegionServer。所以 ServerCallable 针对的是 HRegionServer。

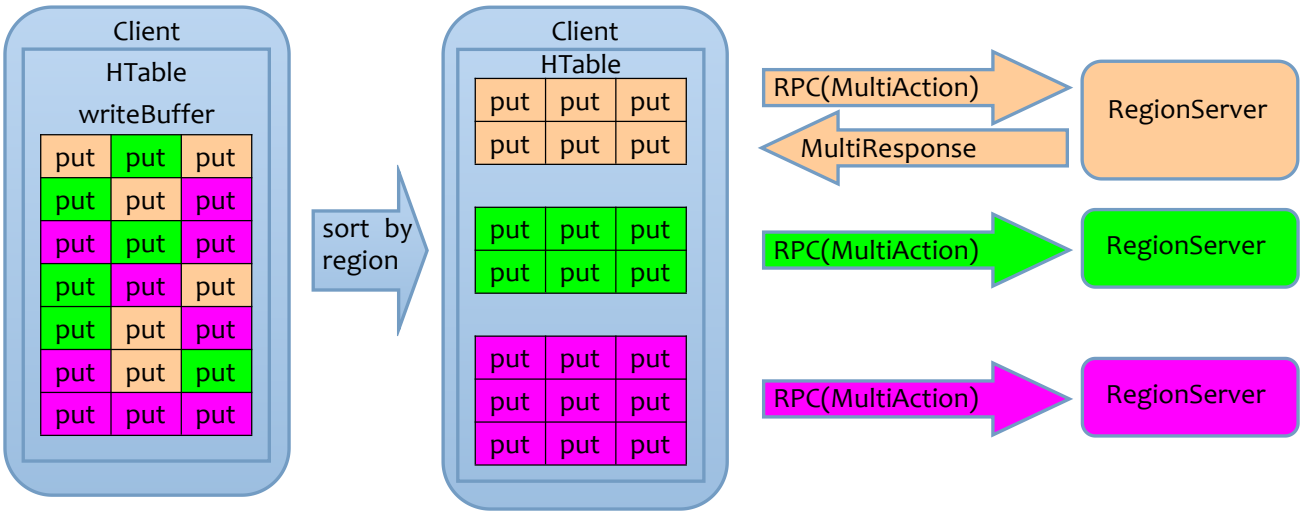
HConnection 是用来建立到 HRegionServer(HRegionInterface)的连接。如何连接？需要获取到 HRegionLocation。
对于 Put 而言，因为在 processBatch()中已经根据 tableName 和 row-key 获取到了 HRegionLocation。
所以创建一个 ServerCallable 对象时需要重写 connect()并且只需要获取到 HRegionInterface server 即可。
如果准备工作没有提供 HRegionLocation，则 connect()中还需要根据 tableName 和 row 确定 HRegionLocation。

```
public abstract class ServerCallable<T> implements Callable<T> {
  protected final HConnection connection;
  protected final byte [] tableName;
  protected final byte [] row;
  protected HRegionLocation location;
  protected HRegionInterface server;
  protected int callTimeout;

  public ServerCallable(HConnection connection, byte [] tableName, byte [] row, int callTimeout) {
    this.connection = connection;
    this.tableName = tableName;
    this.row = row;
    this.callTimeout = callTimeout;
  }

  public void connect(final boolean reload) throws IOException {
    this.location = connection.getRegionLocation(tableName, row, reload);
    this.server = connection.getHRegionConnection(location.getHostname(), location.getPort());
  }

  public T withRetries() {
    final int numRetries = c.getInt(HConstants.HBASE_CLIENT_RETRIES_NUMBER, 10);
    for (int tries = 0; tries < numRetries; tries++) {
      try {
        beforeCall();
        connect(tries != 0);
        return call();
      } finally {
        afterCall();
      }
    }
```

```
    return null;
  }
  public T withoutRetries() {
    try {
      beforeCall();
      connect(false);
      return call();
    } finally {
      afterCall();
    }
  }
```

## 第 3/4 步：收集结果并重试:

客户端启用缓存后，会将批量动作一次性送往服务器 RegionServer, MultiAction 对应了 MultiResponse.




注意：在客户端缓存 writeBuffer 会经过 Region 排序形成一个 HRegionLocation 对应多个 Action 的批处理资源.
每个 HRegionLocation 对应的 MultiAction 都会发起一次到 RegionServer 的 RPC 调用. 也就是说如果两个不同的 HRegionLocation, 如果他们的 host+port 即指向的 RegionServer 是同一个, RPC 调用次数为 2 次而不是一次.
因为在 processBatchCallback 方法中是针对每个 HRegionLocation 都调用 server.multi(MultiAction)的.
而 HRegionServer server 的获取是根据每个 HRegionLocation 的 host+port, 通过 HConnection 建立连接的.
因此 RPC 调用的次数决定于 HRegionLocation, 而不是决定于 HRegionServer 的.

### MultiAction & MultiResponse

actionsByServer 和 futures 类似, key 都是 HRegionLocation, value 则分别是 MultiAction 和 MultiResponse:

```
        Map<HRegionLocation, MultiAction<R>> actionsByServer = new HashMap();
        Map<HRegionLocation, Future<MultiResponse>> futures = new HashMap(actionsByServer.size());
```

MultiAction 和 MultiResponse 内部也是 Map, 其中 key 都是 regionName. value 分别是 Action 和 Result.

```
public final class MultiAction<R> implements Writable {
  // map of regions to lists of puts/gets/deletes for that region.
  public Map<byte[], List<Action<R>>> actions = new TreeMap(Bytes.BYTES_COMPARATOR);
  public void add(byte[] regionName, Action<R> a) {
    List<Action<R>> rsActions = actions.get(regionName);
    if (rsActions == null) {
      rsActions = new ArrayList<Action<R>>();
      actions.put(regionName, rsActions);
    }
```

```
      rsActions.add(a);
    }
  }

public class MultiResponse implements Writable {
    // map of regionName to list of (Results paired to the original index for that Result)
    private Map<byte[], List<Pair<Integer, Object>>> results = new TreeMap(Bytes.BYTES_COMPARATOR);
    public void add(byte[] regionName, Pair<Integer, Object> r) {
      List<Pair<Integer, Object>> rs = results.get(regionName);
      if (rs == null) {
        rs = new ArrayList<Pair<Integer, Object>>();
        results.put(regionName, rs);
      }
      rs.add(r);
    }
    public void add(byte []regionName, int originalIndex, Object resOrEx) {
      add(regionName, new Pair<Integer,Object>(originalIndex, resOrEx));
    }
}

public class Pair<T1, T2> implements Serializable {
    protected T1 first = null;
    protected T2 second = null;
    public Pair(T1 a, T2 b){
      this.first = a;
      this.second = b;
    }
}
```

RPC 在 HRegionServer 端的处理我们暂时不考虑，先来接着 processBatchCallback 的处理流程:

```
            // step 3: collect the failures and successes and prepare for retry
            for (Entry<HRegionLocation, Future<MultiResponse>> responsePerServer : futures.entrySet()) {
                HRegionLocation loc = responsePerServer.getKey();
                    Future<MultiResponse> future = responsePerServer.getValue();
                    MultiResponse resp = future.get();
                    if (resp == null) continue;
                    for (Entry<byte[], List<Pair<Integer,Object>>> e : resp.getResults().entrySet()) {
                      byte[] regionName = e.getKey();
                      List<Pair<Integer, Object>> regionResults = e.getValue();
                      for (Pair<Integer, Object> regionResult : regionResults) {
                        if (regionResult != null) {    // Result might be an Exception, including DNRIOE
                          results[regionResult.getFirst()] = regionResult.getSecond();
                          if (callback != null && !(regionResult.getSecond() instanceof Throwable)) {
                            callback.update(e.getKey(), list.get(regionResult.getFirst()).getRow(), (R)regionResult.getSecond());
                          }
                        }
                      }
                    }
            }
```

```
        // step 4: identify failures and prep for a retry (if applicable).
        // Find failures (i.e. null Result), and add them to the workingList (in order), so they can be retried.
        retry = false;
        workingList.clear();
        actionCount = 0;
        for (int i = 0; i < results.length; i++) {
          // if null (fail) or instanceof Throwable && not instanceof DNRIOE then retry that row. else dont.
          if (results[i] == null || (results[i] instanceof Throwable && !(results[i] instanceof DoNotRetryIOException))) {
            retry = true;                 // 可以重试
            actionCount++;                // 动作次数
            Row row = list.get(i);
            workingList.add(row);   // 重试，加入workingList列表中
            deleteCachedLocation(tableName, row.getRow());
          } else {                        // 无需重试：正常的返回结果，或者是DNRIOE
            if (results[i] != null && results[i] instanceof Throwable) {
              actionCount++;
            }
            // add null to workingList, so the order remains consistent with the original list argument.
            workingList.add(null);   // 加入null元素，保持列表的有序性
          }
        }
```

# RegionServer.multi

客户端发起的批处理的 RPC 调用在 RegionServer 端，循环处理参数 MultiAction<regionName, List<Action>>

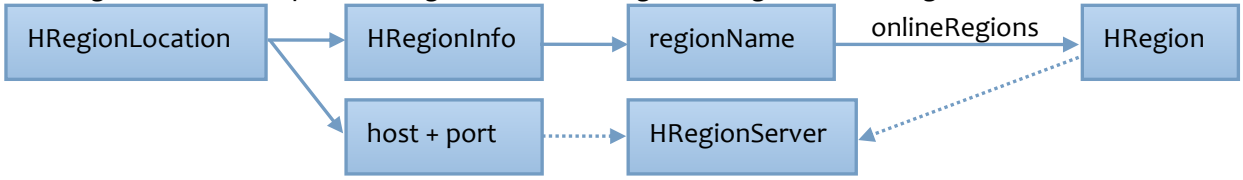
```
public <R> MultiResponse multi(MultiAction<R> multi) throws IOException {
  checkOpen();
  MultiResponse response = new MultiResponse();
  for (Map.Entry<byte[], List<Action<R>>> e : multi.actions.entrySet()) {
    byte[] regionName = e.getKey(); // 每个HRegionLocation都调用一次mutli，而一个RegionLocation对应一个regionName..??
    List<Action<R>> actionsForRegion = e.getValue();
    // sort based on the row id - this helps in the case where we reach the end of a region,
    // so that we don't have to try the rest of the actions in the list.
    Collections.sort(actionsForRegion);
    Row action;
    List<Action<R>> mutations = new ArrayList<Action<R>>();
    for (Action<R> a : actionsForRegion) {       // 处理MultiAction中的每个Action. 由originalIndex和Row实现类(Put, Delete..)组成
      action = a.getAction();                    // Row实现类，比如Put, Delete...
      int originalIndex = a.getOriginalIndex(); // 原始动作(Put)在客户端缓存(writeBuffer)中的位置(添加到 List 中的顺序)
        if (action instanceof Delete || action instanceof Put) {
          mutations.add(a);
        } else if (action instanceof Get) {
          response.add(regionName, originalIndex, get(regionName, (Get)action));
        } else if (action instanceof Exec) {
          ExecResult result = execCoprocessor(regionName, (Exec)action);
          response.add(regionName, new Pair<Integer, Object>(a.getOriginalIndex(), result.getValue()));
        } else if (action instanceof Increment) {
          response.add(regionName, originalIndex, increment(regionName, (Increment)action));
```

```
            } else if (action instanceof Append) {
                response.add(regionName, originalIndex, append(regionName, (Append)action));
            } else if (action instanceof RowMutations) {
                mutateRow(regionName, (RowMutations)action);
                response.add(regionName, originalIndex, new Result());
            } else {
                throw new DoNotRetryIOException("Invalid Action, row must be a Get, Delete, Put, Exec, Increment, or Append.");
            }
        }
```

如果动作类型是无效的，直接抛出 DoNotRetryIOException，因为无效的动作即使重试 N 次也是失败的.
另外如果在运行之前进行检查，比如检查表结构等，检查失败也会将 DNRIE 作为 Response 返回表示无需重试.
批处理可接受的动作类型有: Put , Delete, Append, Increment, Get, Exec, RowMutations...

```
        // We do the puts with result.put so we can get the batching efficiency we so need.
        // All this data munging doesn't seem great, but at least we arent copying bytes or anything.
        if (!mutations.isEmpty()) {
            HRegion region = getRegion(regionName);                 //  根据regionName获取对应的HRegion
            if (!region.getRegionInfo().isMetaTable()) this.cacheFlusher.reclaimMemStoreMemory();
            List<Pair<Mutation,Integer>> mutationsWithLocks = Lists.newArrayListWithCapacity(mutations.size());
            for (Action<R> a : mutations) {
                Mutation m = (Mutation) a.getAction();                  // Put对象
                Integer lock = getLockFromId(m.getLockId());            // put(Put)时, Put对象使用的Lock，没有显示指定使用系统自带的
                mutationsWithLocks.add(new Pair<Mutation, Integer>(m, lock));
            }

            this.requestCount.addAndGet(mutations.size());
            OperationStatus[] codes = region.batchMutate(mutationsWithLocks.toArray(new Pair[]{}));    // List转成数组
            for( int i = 0 ; i < codes.length ; i++) {
                OperationStatus code = codes[i];
                Action<R> theAction = mutations.get(i);
                Object result = null;

                if (code.getOperationStatusCode() == OperationStatusCode.SUCCESS) {
                    result = new Result();
                } else if (code.getOperationStatusCode() == OperationStatusCode.SANITY_CHECK_FAILURE) {
                    result = new DoNotRetryIOException(code.getExceptionMsg());
                } else if (code.getOperationStatusCode() == OperationStatusCode.BAD_FAMILY) {
                    result = new NoSuchColumnFamilyException(code.getExceptionMsg());
                }
                // FAILURE && NOT_RUN becomes null, aka: need to run again.
                response.add(regionName, theAction.getOriginalIndex(), result);
            }
        }
```

根据 regionName 从 Map: onlineRegions 中获取 HRegion. HRegion 才是 HRegionServer 中真正工作的类:



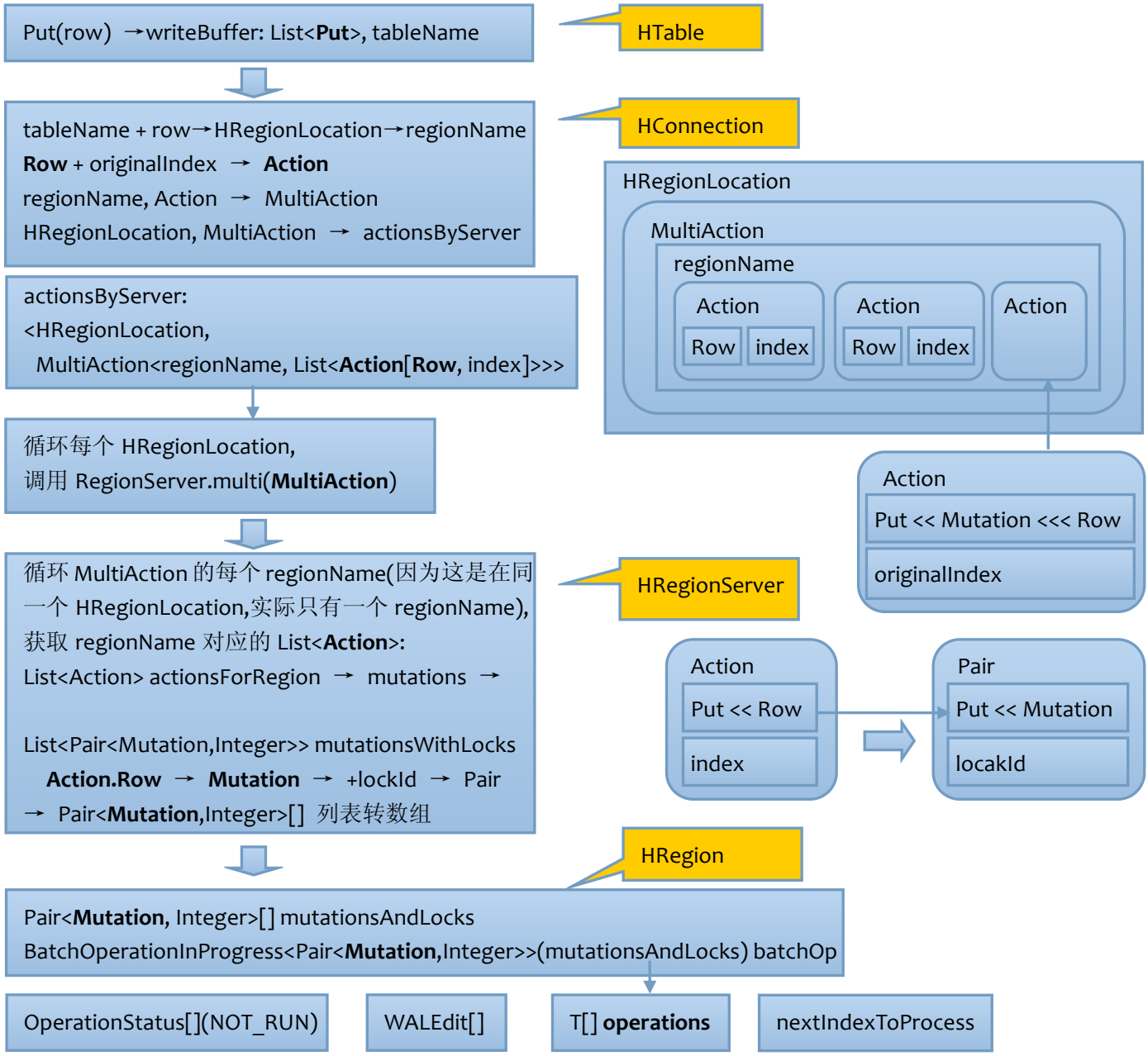

拆分 Region 时会将 HRegion 加入到 onlineRegions 中.

# HRegion.batchMutate

```
public OperationStatus[] batchMutate(Pair<Mutation, Integer>[] mutationsAndLocks) throws IOException {
  BatchOperationInProgress<Pair<Mutation, Integer>> batchOp = new BatchOperationInProgress(mutationsAndLocks);
  boolean initialized = false;
  while (!batchOp.isDone()) {        // 说明批处理不是一次性完成，会分成多次
    long newSize;
    startRegionOperation();          // 加锁
    try {
      if (!initialized) {            // 第一次会对所有的记录进行预处理
        this.writeRequestsCount.increment();
        this.opMetrics.setWriteRequestCountMetrics(this.writeRequestsCount.get());
        doPreMutationHook(batchOp);  // 协处理器预处理
        initialized = true;
      }
      long addedSize = doMiniBatchMutation(batchOp);              // 只处理一部分哦. 如果处理完成, batchOp.isDone
      newSize = this.addAndGetGlobalMemstoreSize(addedSize);    // 首先写到内存的缓冲区MemStore中
    } finally {
      closeRegionOperation();        // 释放锁
    }
    if (isFlushSize(newSize)) {      // 如果大小超过memStroe的大小，刷写到磁盘上
      requestFlush();
    }
  }
  return batchOp.retCodeDetails;
}

public void startRegionOperation()
  lock(lock.readLock());
}
public void closeRegionOperation(){
  lock.readLock().unlock();
}
public long addAndGetGlobalMemstoreSize(long memStoreSize) {
  if (this.rsAccounting != null) rsAccounting.addAndGetGlobalMemstoreSize(memStoreSize);
  return this.memstoreSize.getAndAdd(memStoreSize);
}
private boolean isFlushSize(final long size) {
  return size > this.memstoreFlushSize;
}
private void requestFlush() {
  if (this.rsServices == null) return;
  synchronized (writestate) {
    if (this.writestate.isFlushRequested()) return;
    writestate.flushRequested = true;
  }
  this.rsServices.getFlushRequester().requestFlush(this);
}
```

```
private static class BatchOperationInProgress<T> {
    T[] operations;
    int nextIndexToProcess = 0;
    OperationStatus[] retCodeDetails;
    WALEdit[] walEditsFromCoprocessors;

    public BatchOperationInProgress(T[] operations) {
        this.operations = operations;
        this.retCodeDetails = new OperationStatus[operations.length];
        this.walEditsFromCoprocessors = new WALEdit[operations.length];
        Arrays.fill(this.retCodeDetails, OperationStatus.NOT_RUN);
    }
    public boolean isDone() {
        return nextIndexToProcess == operations.length;
    }
}
```

**data-structure flow(HTable->HConnection->HRegionServer->HRegion)**

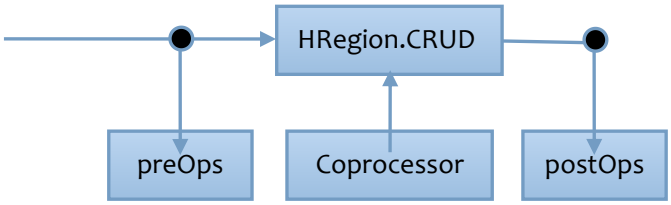


# Coprocessor 协处理器

在分析协处理器 Coprocessor 之前，我们先来总结下客户端发起 RPC 请求，最终到 HRegion 的过程.

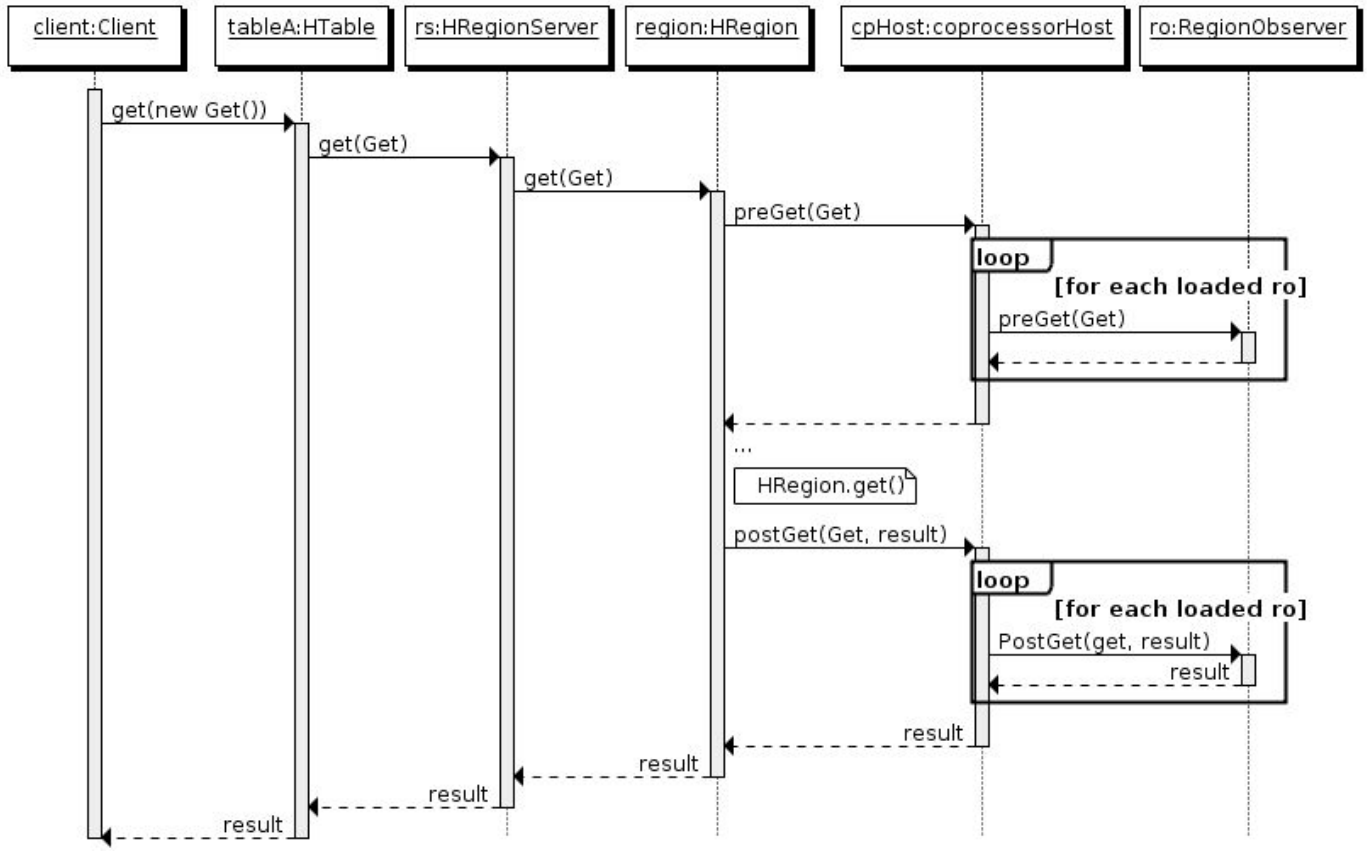

客户端操作表数据时，会先获取到该行对应的 HRegionLocation，然后联系 HRegionServer，发起 RPC 调用. HRegionServer 会将调用转发到 HRegion 上. 因为 HRegion 才是真正存储数据的地方.

如果客户端代码想要在 HRegion 执行 CRUD 操作前后植入自定义的逻辑可以采用类似触发器的方式：协处理器. 触发器可以在动作发生前后自定义一些动作，协处理器也类似，有 preOps 和 postOps 分别作用于事件前后.



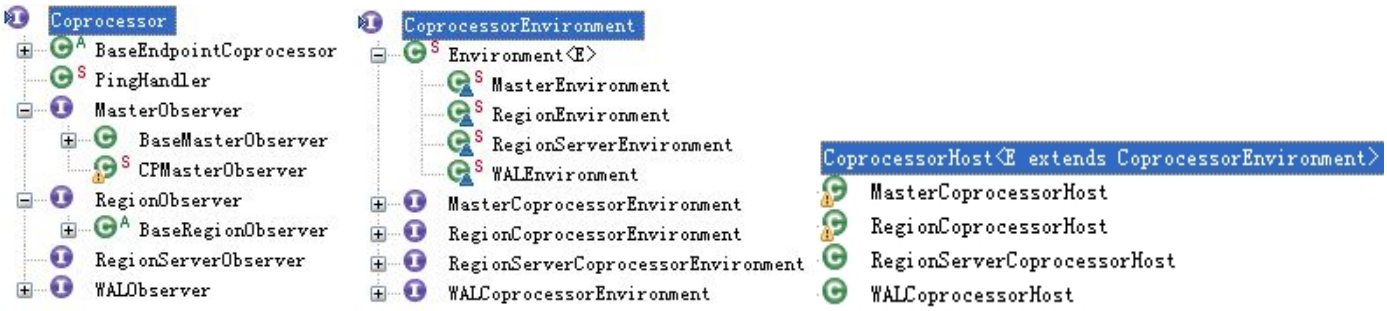

**NoSQL VS RDBMS:**
协处理器 Coprocessor 又分为 Observer 和 EndPoint. 分别对应 RDBMS 的触发器和存储过程.
我们知道 RDBMS 的触发器一般都是针对一行数据在插入/更新/删除前后自定义动作的.
与此类似, Coprocessor 的 Observer 也是针对单个 Region 的. 因为一张表分成多个 Region.
客户端针对 RegionServer 的一次 RPC 调用，只作用于 RegionServer 的一个 Region.
如果要跨 Region，比如自定义操作作用于 Table 的所有 Region，就要用到与存储过程相对应的 EndPoint.

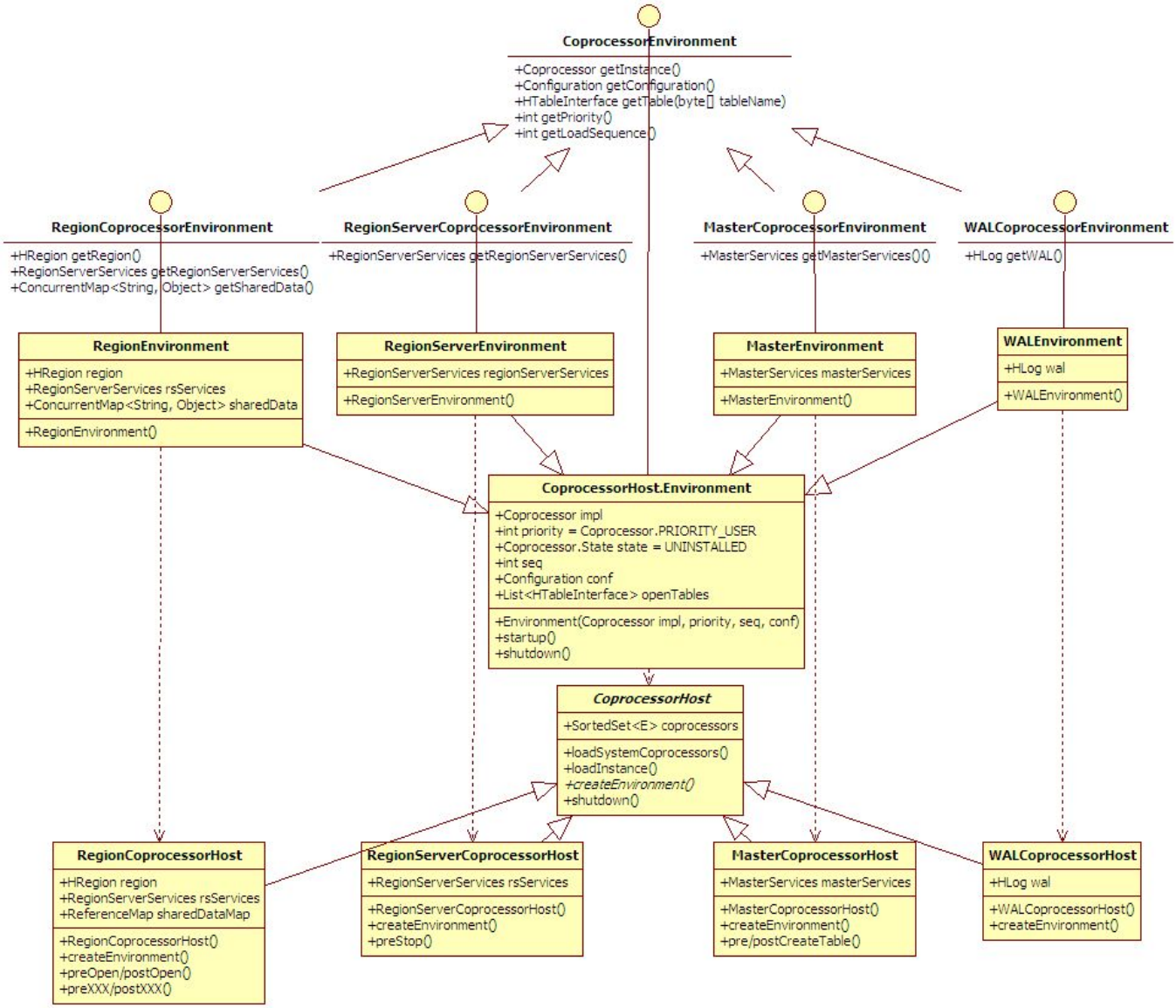
下图展示了 Get 流程，在 HRegion.get()操作前后分别执行了协处理器的 preGet 和 postGet():
注: HRegion 没有直接和 RegionObserver(即 Coprocessor 实现类)联系，而是经过了中间的 CoprocessorHost 主机.




图片来源: https://blogs.apache.org/hbase/entry/coprocessor_introduction

Coprocessor 相关的类的继承结构:




Coprocessor 相关类图

以 HRegion 上的 RegionCoprocessorHost 为例，当创建 HRegion 时(由 Master 控制, RegionServer 进行 open 或者 split), 会创建一个 RegionCoprocessorHost 对象, RegionCoprocessorHost 通过静态配置文件和表模式加载系统级别或者用户级别的协处理器: loadSystemCoprocessors 和 loadTableCoprocessors. 系统级加载在抽象父类 CoprocessorHost 中(其他 Host 比如 Master, RegionServer, WAL 也都会调用 loadSystemCoprocessors，只是传递的 key 不同而已).

以加载静态配置文件为例, RegionCoprocessorHost 会从 hbase-site.xml 配置项"hbase.coprocessor.region.classes", "hbase.coprocessor.user.region.classes"获取用户定义的协处理器的 Class 类型:implClass，然后在 loadInstance()中通过反射实例化协处理器对象 Coprocessor impl.

因为协处理器必须在专用的环境中才能运行，即一个协处理器对应一个协处理器环境，所以要把协处理器的 Class 类型，以及刚刚实例化的协处理器对象，还有一些环境需要的信息比如：该协处理器的优先级，加载顺序等，调用抽象方法 createEnvironment．各个 CoprocessorHost 实现类都要实现该抽象方法．

用户自定义的 Coprocessor 是有不同的作用范围的，因为 HBase 将各个组件分成 Master, RegionServer, Region. 因此不同的 Coprocessor 是要针对 HBase 中对应的组件．而 Coprocessor 必须在专用的环境中运行，因此 HBase 中的各个组件都应该有各自独立的运行环境，保证特定的 Coprocessor 只能运行在特定的专用环境中(类似隔离器).

| 配置属性 | 协处理器主机 | 使用环境 | 服务类型 |
|---|---|---|---|
| hbase.coprocessor.master.classes | MasterCoprocessorHost | MasterCoprocessorEnvironment | Master |
| hbase.coprocessor.regionserver.classes | RegionServerCoprocessorHost | RegionServerCoprocessorEnvironment | RegionServer |
| hbase.coprocessor.region.classes | RegionCoprocessorHost | RegionCoprocessorEnvironment | Region |
| hbase.coprocessor.wal.classes | WALCoprocessorHost | WALCoprocessorEnvironment | RegionServer |

Coprocessor, CoprocessorEnvironment, CoprocessorHost 以及 Region 的关系:

1. 一个 HRegion 只对应一个 CoprocessorHost
2. 一个 CoprocessorHost 可以有多个 CoprocessorEnvironment
3. 一个 CoprocessorEnvironment 运行一个 Coprocessor，一个 Coprocessor 运行在唯一的 CoprocessorEnvironment
4. 因此一个 CoprocessorHost 上会有多个 Coprocessor
5. 一个 HRegion 上就可以作用多个 Coprocessor(协处理器链)

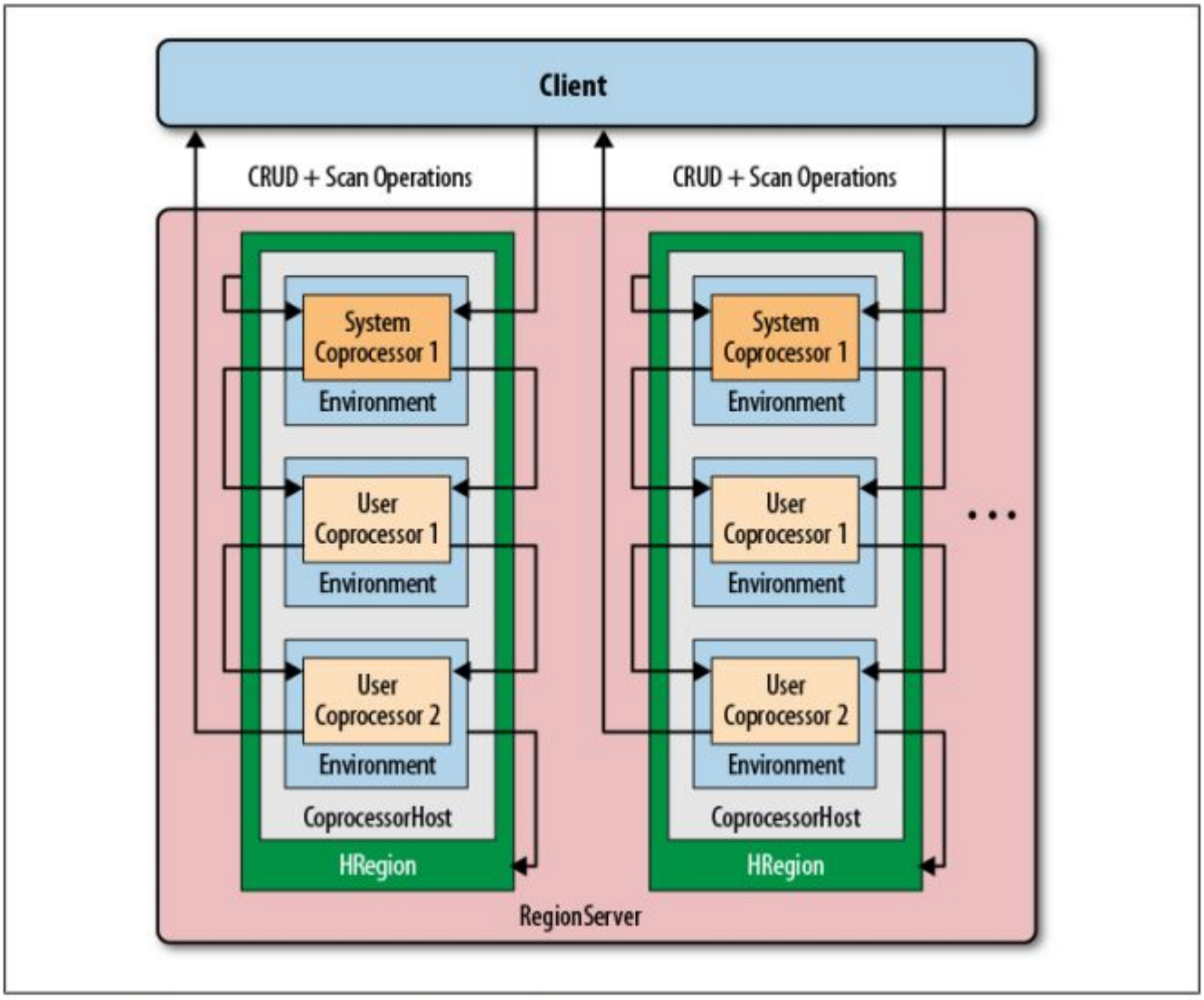


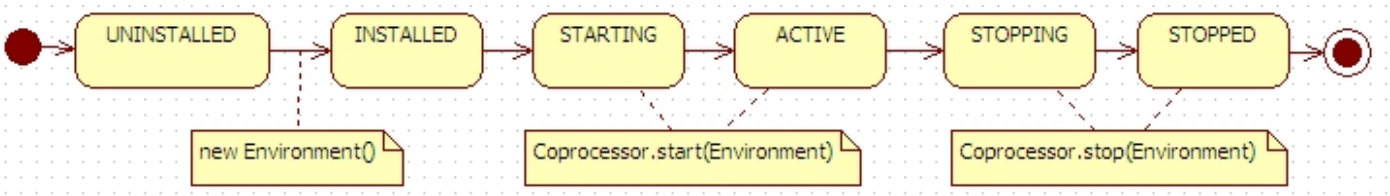

*Figure 4-3. Coprocessors executed sequentially, in their environment, and per region*

图片来源: HBase 权威指南 Chapter4 - 协处理器在他们每个 region 环境中按顺序执行

# Coprocessor 接口

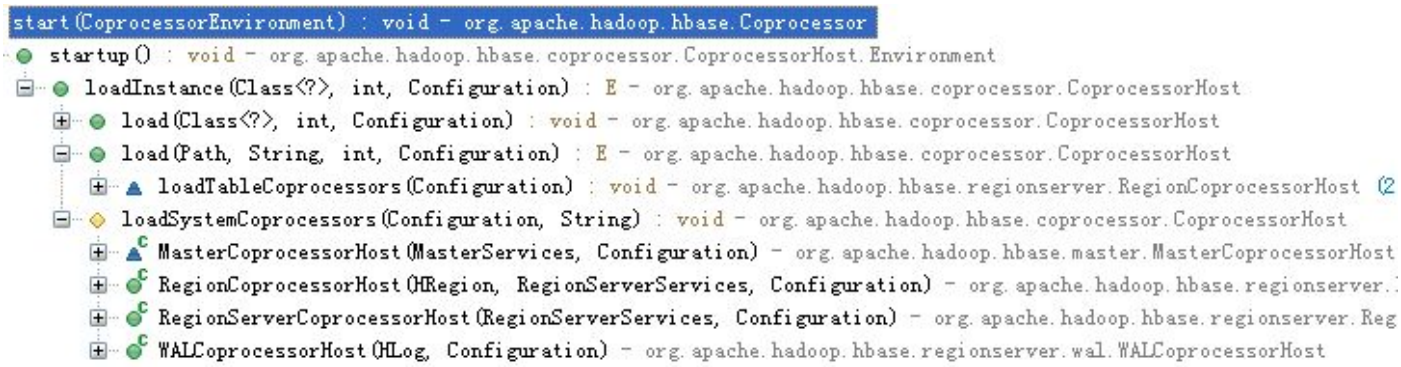
```
public interface Coprocessor {
  static final int VERSION = 1;
  static final int PRIORITY_HIGHEST = 0;                    /** Highest installation priority */
  static final int PRIORITY_SYSTEM = Integer.MAX_VALUE/4;   /** High (system) installation priority */
  static final int PRIORITY_USER = Integer.MAX_VALUE/2;     /** Default installation priority for user coprocessors */
  static final int PRIORITY_LOWEST = Integer.MAX_VALUE;     /** Lowest installation priority */
  public enum State {/** Lifecycle state of a given coprocessor instance.  协处理器的生命周期  */
    UNINSTALLED,    // 初始值
    INSTALLED,      // 装载了环境
    STARTING,       // 协处理器将要开始工作，即start()将要被调用
    ACTIVE,         // start()调用后
    STOPPING,       // close()调用前
    STOPPED         // close()调用后
  }

  // Interface
  void start(CoprocessorEnvironment env) throws IOException;
  void stop(CoprocessorEnvironment env) throws IOException;
}
```

事件的发生会影响协处理器的状态，事件包括用户产生的事件，也包括服务器内部自动产生的事件.



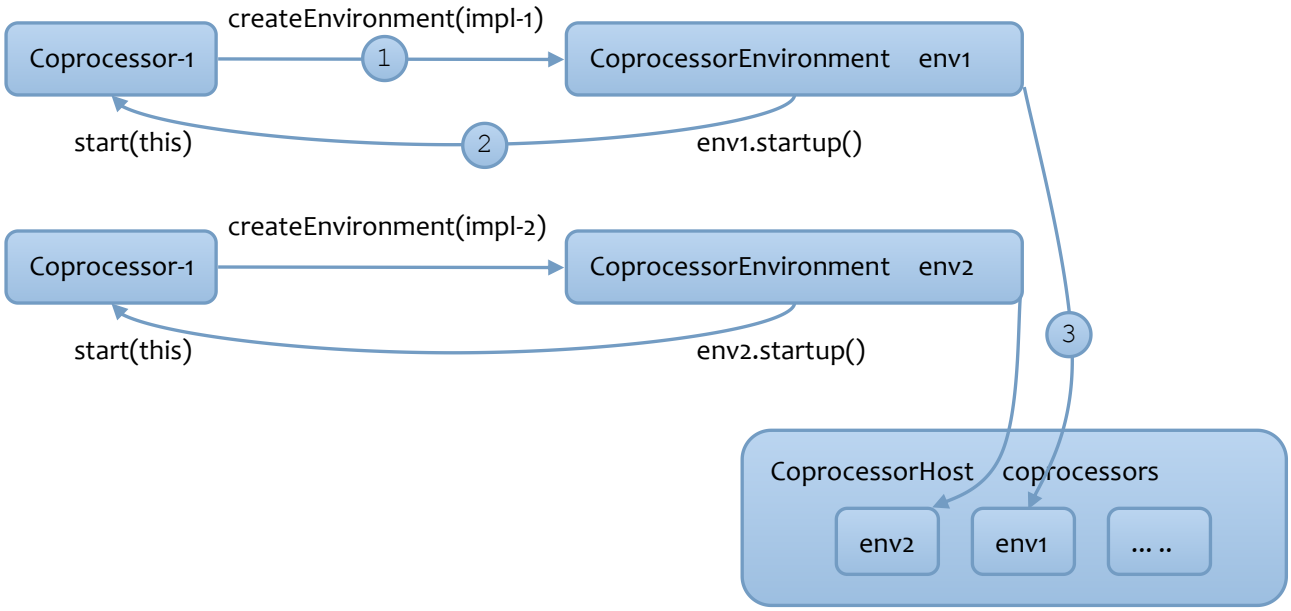

| 协处理器状态 | 说明 | 动作触发 |
|---|---|---|
| UNINSTALLED | 协处理器最初的状态，没有环境，没有初始化 | |
| INSTALLED | 协处理器实例装载了它的环境参数 | createEnvironment->super()->new Environment |
| STARTING | 协处理器将要开始工作，即 start()将要被调用 | Envronment.startup() → impl.start(env)前 |
| ACTIVE | start()方法调用完毕 | impl.start(env)后 |
| STOPPING | stop()方法被调用之前的状态 | Environment.shutdown() → impl.stop(env)前 |
| STOPPED | stop()方法将控制权交给框架后 | impl.stop(env)后 |

start()和 stop()通过和 CoprocessorEnvironment 进行关联，使得 Coprocessor 作用的范围只能限定在特定的环境中. 也就是说协处理器的生命周期是由 CoprocessorEnvironment 管理的. 一旦 Environment 消亡, Coprocessor 也不存在.



当启动系统，新建 Host 时，会加载协处理器并实例化，协处理器会装载对应的运行环境．获得环境后，就会立马调用 Environment.startup()，其中会调用协处理器的 start(this)．因为是在 Environment 中，this 正好是 start()的参数．
从上面的调用树可以看到，Master, RegionServer, Region, WAL 四种 CoprocessorHost 都会启动相应的 Coprocessor.

再次注意 Coprocessor, CoprocessorHost, CoprocessorEnvironment 之间的关系．
假设配置文件 hbase-site.xml 的配置项 hbase.coprocessor.region.classes 配置了以逗号分隔的多个协处理器．
那么 region 级别的每个协处理器都会对应自己的 CoprocessorEnvironment．　通过 CoprocessorEnvironment 启动 Coprocessor，也是每个 CoprocessorEnvironment 启动属于自己的内部 Coprocessor，保证了运行环境的隔离．




因为启动过程不管对于哪种类型的协处理器(类型指的是协处理器的作用范围是 Master,RegionServer, Region,WAL)都是必须的步骤．即一旦实例化好 Coprocessor，以及将 Coprocessor 装载到 CoprocessorEnvironment 之后，就要由框架(这里指包含 CoprocessorEnvironment 的 CoprocessorHost 类)让 CoprocessorEnvironment 启动 Coprocessor.

那么 Coprocessor 的 stop()方法是不是所有类型的 CoprocessorHost 都会调用到的呢?
从调用树可以看到只有关闭 HRegion 时，RegionCoprocessorHost 会调用 shutdown 方法进而停止 Coprocessor 进程.


问题: [Master|WAL|RegionServer]CoprocessorHost 是怎么保证关闭自己内部所有 Environment 的 Coprocessor 呢?

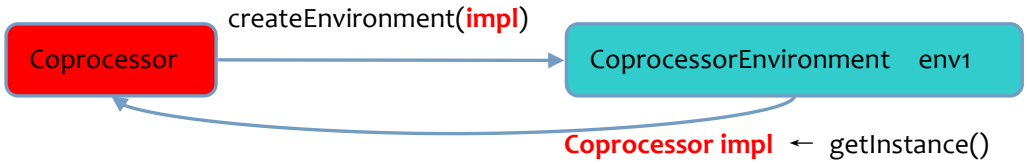
由于事件有多种多样，而且 HBase 不同的组件面向的操作级别也是不同的(比如 Master 是针对 Table, Column 的 DDL 操作，而 RegionServer 和 Region 就针对的是 Region 的 DML 操作)．肯定不能在 Coprocessor 接口中都一一列举出来．因此比 Coprocessor 粒度更细的接口有: MasterObserver, RegionServerObserver, RegionObserver, WALObserver.
至于每个 Observer 都有哪些相应的方法和具体的事件相关．后面再介绍.

对应 Coprocessor 的 CoprocessorEnvironment 也有不同的实现:
Coprocessor 应当只与提供给它们的 CoprocessorEnvironment 进行交互，这样的好处是可以保证没有会被恶意代码用来破坏数据的后门. CoprocessorEnvironment 用来在 Coprocessor 的生命周期中保持其(Coprocessor)状态.
Coprocessor 实例一直被保存在提供的 CoprocessorEnvironment 中．即 Coprocessor 的生命周期是由框架管理.

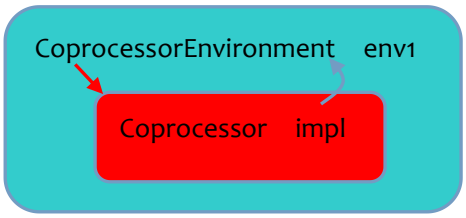
# CoprocessorEnvironment 协处理器环境

```
public interface CoprocessorEnvironment { /** Coprocessor environment state. */
  public int getVersion();                /** @return the Coprocessor interface version */
  public String getHBaseVersion();        /** @return the HBase version as a string (e.g. "0.21.0") */
  public int getPriority();               /** @return the priority assigned to the loaded coprocessor */
  public int getLoadSequence();           /** @return the load sequence number */
  public Configuration getConfiguration(); /** @return the configuration */

  public Coprocessor getInstance();       /** @return the loaded coprocessor instance */
  public HTableInterface getTable(byte[] tableName) throws IOException;   /** @return an interface for accessing the given table */
}
```

CoprocessorEnvironment 最重要的是 getInstance()和 getTable()方法.
上面我们已经分析了 Coprocessor 和 CoprocessorEnvironment 的关系: Coprocessor 必须在 Environment 里运行.
在 CoprocessorHost 中通过 createEnvironment 传递 Coprocessor 实例(impl)给 CoprocessorEnvironment 实现类.
这样 CoprocessorEnvironment 就可以通过 getInstance()获得传递给他的 Coprocessor impl 对象了.

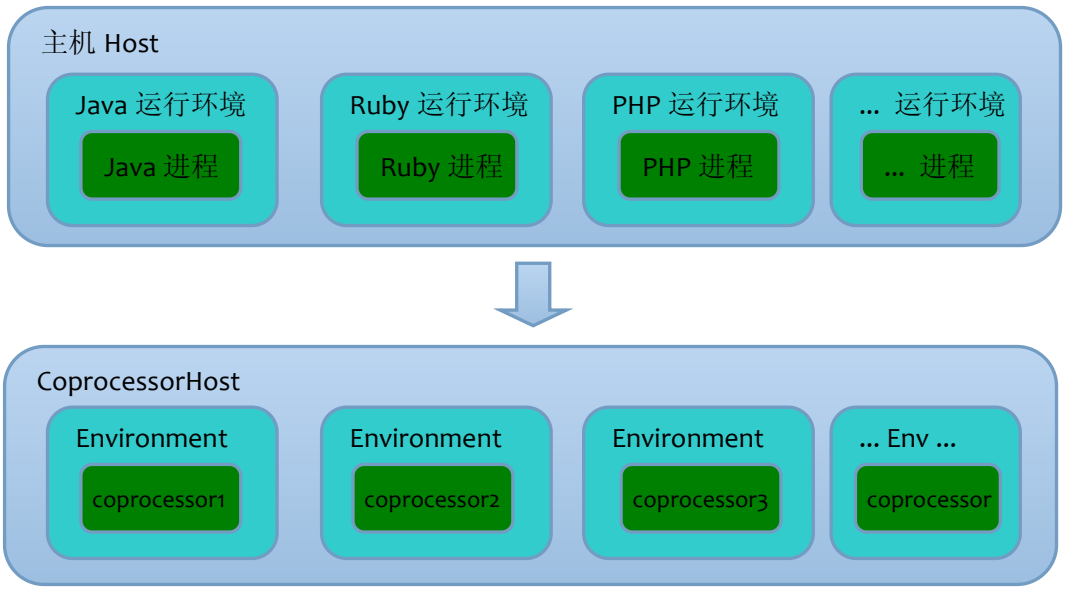



如果以组件之间的包含关系，下图看起来会更加直观:




1. 从上图您应该还可以看出，外界要访问 Coprocessor，必须先获得 CoprocessorEnvironment.
然后通过调用 CoprocessorEnvironment.getInstance()才能获取到运行在环境里面的 Coprocessor.
不允许直接调用 Coprocessor. 这也是 Coprocessor 的生命周期被 CoprocessorEnvironment 所管理的一个原因.

2. 上图是由外而内进行访问. 可以从 CoprocessorEnvironment 再往更高层次即 CoprocessorHost 进行考虑.
其实可以从宿主机 Host，运行环境 Environment，进程来理解 HBase 中 Coprocessor 相关的三级结构.
宿主机好比我们的机器，会有多个运行环境，每个运行环境运行自己独立的进程. 运行环境之间不能相互影响.



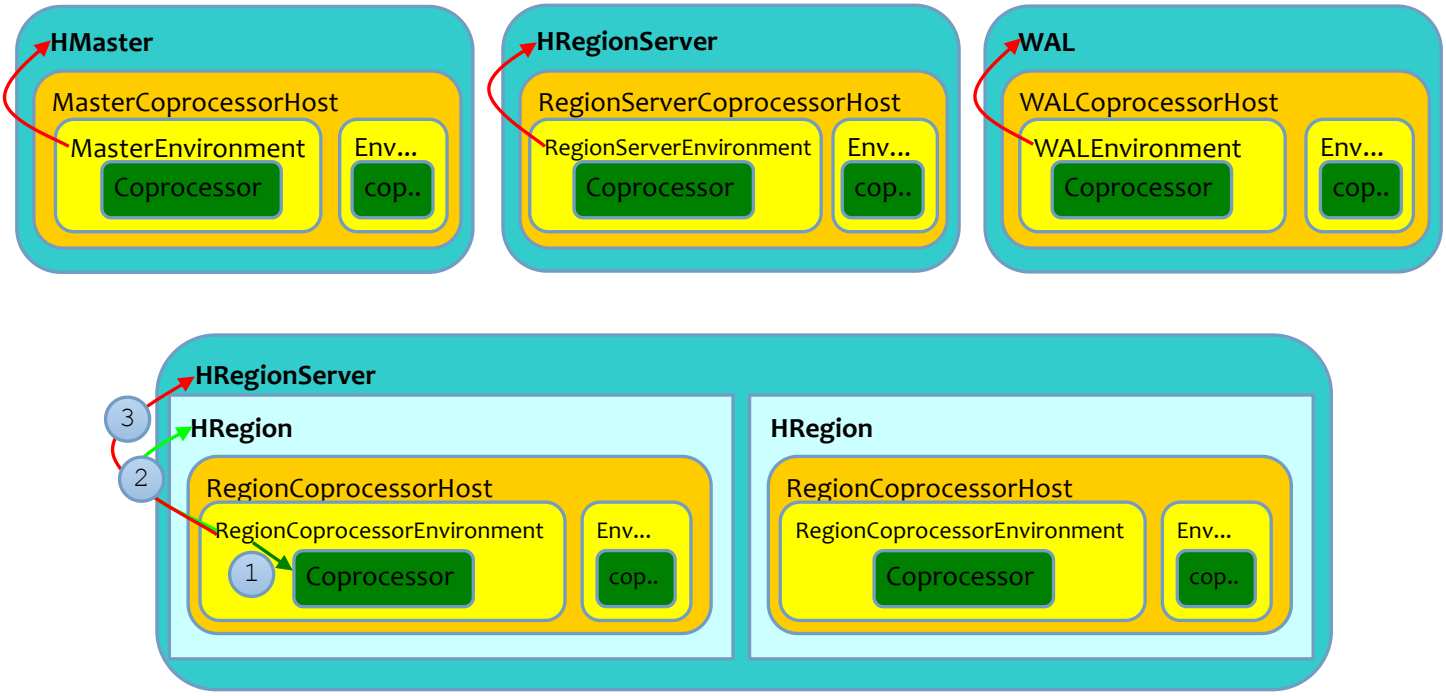
CoprocessorHost 的内部类 Environment 抽象了 CoprocessorEnvironment 接口：有了抽象类 Environment，对于不同的 CoprocessorEnvironment，如果有不同的逻辑就可以添加到自己的实现上，而无需改动框架内部的代码.

```
/** Encapsulation of the environment of each coprocessor  封装每个Coprocessor的环境. 即一个环境下只运行一个Coprocessor */
public static class Environment implements CoprocessorEnvironment {
  public Coprocessor impl;                                  /** The coprocessor */
  protected int priority = Coprocessor.PRIORITY_USER;       /** Chaining priority */
  Coprocessor.State state = Coprocessor.State.UNINSTALLED;  /** Current coprocessor state */
  /** Accounting for tables opened by the coprocessor */
  protected List<HTableInterface> openTables = Collections.synchronizedList(new ArrayList<HTableInterface>());
  private int seq;
  private Configuration conf;

  public Environment(final Coprocessor impl, final int priority, final int seq, final Configuration conf) {
    this.impl = impl;
    this.priority = priority;
    this.state = Coprocessor.State.INSTALLED;
    this.seq = seq;
    this.conf = conf;
  }
  public Coprocessor getInstance()        {return impl;}
  public int getPriority()                { return priority; }
  public int getLoadSequence()            {return seq; }
  public int getVersion()                 { return Coprocessor.VERSION; }
  public String getHBaseVersion()         { return VersionInfo.getVersion(); }
  public Configuration getConfiguration() { return conf; }
  /** Open a table from within the Coprocessor environment  在协处理器环境中打开一张表  */
  public HTableInterface getTable(byte[] tableName) throws IOException {
    return new HTableWrapper(tableName, CoprocessorHConnection.getConnectionForEnvironment(this));
  }

  public void startup() {                                   /** Initialize the environment */
    if (state == Coprocessor.State.INSTALLED || state == Coprocessor.State.STOPPED) {
      state = Coprocessor.State.STARTING;
      impl.start(this);
      state = Coprocessor.State.ACTIVE;
    }
  }
  protected void shutdown() {                               /** Clean up the environment */
    if (state == Coprocessor.State.ACTIVE) {
      state = Coprocessor.State.STOPPING;
      impl.stop(this);
      state = Coprocessor.State.STOPPED;
    }
    for (HTableInterface table: openTables) {
      ((HTableWrapper)table).internalClose();       // clean up any table references
    }
  }
}
```

比 CoprocessorEnvironment 更细粒度的接口分别有: MasterCoprocessorEnvironment, WALCoprocessorEnvironment, RegionServerCoprocessorEnvironment, RegionCoprocessorEnvironment. 这些接口扩展了通用方法提供面向针对特定级别的方法. 比如 RegionCoprocessorEnvironment 是针对 Region 级别,且 Region 是在 RegionServer 里的. 所以有 HRegion getRegion()和 RegionServerServices getRegionServerServices()方法.

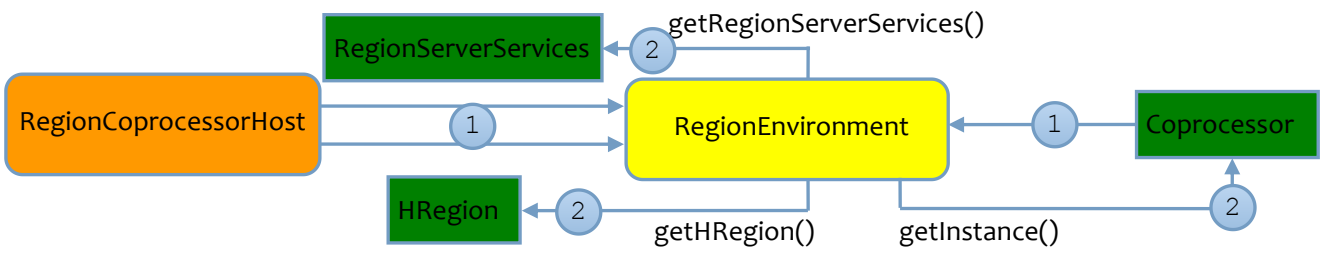
| CoprocessorEnvironment | 方法 | 说明 |
|---|---|---|
| **Region**CoprocessorEnvironment | **HRegion** getHRegion()<br>**RegionServerServices** getRegionServerServices() | 返回监听器监听的 Region 的引用<br>返回共享的 HRegionServer 实例 |
| **RegionServer**CoprocessorEnvironment | **RegionServerServices** getRegionServerServices() | 提供可访问的共享的 HRegionServer 实例 |
| **Master**CoprocessorEnvironment | **MasterServices** getMasterServices() | 提供可访问的共享的 HMaster 实例 |
| **WAL**CoprocessorEnvironment | **HLog** getWAL() | 提供可访问的共享的 HLog 实例 |

HBase 各组件和 Coprocessor 相关类的容器组成图:




CoprocessorEnvironment 承上启下: 下为 Coprocessor, 上就是 CoprocessorHost.
环境的含义: 从环境中获取您想要的东西. 比如①获取下级的 Coprocessor getInstance().
环境中运行着 Coprocessor, 因此需要有启动和结束 Coprocessor 的入口, 分别对应 startup 和 shutdown().
由于 Coprocessor 可以作用于不同级别(Master, WAL, RegionServer, Region). 因此包含 Coprocessor 的环境也是有不同的作用域的. 通过环境应该还能取到所属的位置信息, 这就是上面更细粒度的接口中提供的方法.

那么中间还有一个 Host 的作用是什么呢? Host 集中了所有的 Environment, 对所有的 Environment 进行管理.
Host 提供 Environment 需要的上层组件, 这样 Environment 通过 getXXX 就能获取到上层组件.
这和将 Coprocessor impl 提供给 Environment, 然后 Environment 通过 getInstance()就能获取到下层组件类似.
将什么样的上层组件提供给 Environment, Environment 就只能获取到对应的上层组件. 不会获取到其他组件的.



# CoprocessorHost

前面分析了 CoprocessorEnvironment 和 CoprocessorHost 的关系. 所以 CoprocessorHost 里有环境列表 SortSet<E>.

```
/**Provides the common setup framework and runtime services for coprocessor invocation from HBase services.
 * 提供了通用的启动框架，以及从HBase服务过来的，为协处理器的调用提供了运行时服务.
 * @param <E> the specific environment extension that a concrete implementation provides */
public abstract class CoprocessorHost<E extends CoprocessorEnvironment> {
  /** Ordered set of loaded coprocessors with lock */
  protected SortedSet<E> coprocessors = new SortedCopyOnWriteSet<E>(new EnvironmentPriorityComparator());
  protected Configuration conf;
  protected String pathPrefix;    // unique file prefix to use for local copies of jars when classloading
  protected volatile int loadSequence;

  /**Load system coprocessors. Read the class names from configuration. Called by constructor. */
  protected void loadSystemCoprocessors(Configuration conf, String confKey) {
    Class<?> implClass = null;                          // load default coprocessors from configure file
    String[] defaultCPClasses = conf.getStrings(confKey);
    if (defaultCPClasses == null || defaultCPClasses.length == 0) return;
    int priority = Coprocessor.PRIORITY_SYSTEM;
    List<E> configured = new ArrayList<E>();            // CoprocessorEnvironment列表，不是Coprocessor列表!
    for (String className : defaultCPClasses) {
      // 从coprocessors列表中循环每个Env->getInstance()->getClassName()和Conf的className比较.
      if (findCoprocessor(className) != null) continue;     // 已经存在，就不需要重复加载了
      ClassLoader cl = this.getClass().getClassLoader();
      implClass = cl.loadClass(className); (1)              // Coprocessor实现类
      // 加载Coprocessor实现类，返回对应的CoprocessorEnvironment
      configured.add(loadInstance(implClass, Coprocessor.PRIORITY_SYSTEM, conf));
    }
    coprocessors.addAll(configured); (4)                // add entire set to the collection for COW efficiency
  }

  public E loadInstance(Class<?> implClass, int priority, Configuration conf) throws IOException {
    // create the instance
    Object o = implClass.newInstance();
    Coprocessor impl = (Coprocessor)o;
    // create the environment 创建Coprocessor实现类对应的环境，并启动它! ->抽象类，由子类Host实现
    E env = createEnvironment(implClass, impl, priority, ++loadSequence, conf); (2)
    if (env instanceof Environment) {
      ((Environment)env).startup();
    }
    return env;
  }

  /** Called when a new Coprocessor class is loaded */
  public abstract E createEnvironment(Class<?> implClass, Coprocessor instance, int priority, int sequence, Configuration conf);

  public void shutdown(CoprocessorEnvironment e) {
    if (e instanceof Environment) {
      ((Environment)e).shutdown();
```

```
    }
  }

  /**Find a coprocessor implementation by class name */
  public Coprocessor findCoprocessor(String className) {
    for (E env: coprocessors)
      if (env.getInstance().getClass().getName().equals(className))
        return env.getInstance();
    return null;
  }
}
```

Java 泛型基础： List<User> users 我们很容易知道保存的是 User 列表． SortSet<E> coprocessors 对应的是 E 类型的列表．由于 E 是运行时才确定的． 所以要在类 CoprocessorHost 上添加<E extends CoprocessorEnvironment>.
以 RegionCoprocessorHost 类的声明为例，存储的就是 SortSet<RegionCoprocessHost.RegionEnvironment>列表:

```
RegionCoprocessorHost extends CoprocessorHost<RegionCoprocessorHost.RegionEnvironment>
```

在加载 Coprocessor，创建对应的 CoprocessorEnvironment，让 CoprocessorEnvironment 启动 Coprocessor 之后，就要将创建好的 CoprocessorEnvironment 加入到集合 coprocessors 中.

外界访问 CoprocessorHost 时，需要遍历 coprocessors 集合，获取 Host 里的所有运行时环境，
然后对每个 CoprocessorEnvironment 取出对应的 Coprocessor，最终调用到 Coprocessor 的方法.
以 RegionCoprocessorHost.prePut 为例(调用 HRegion.put 时会触发该动作的执行):

```
  public boolean prePut(Put put, WALEdit edit, final boolean writeToWAL) throws IOException {
    boolean bypass = false;
    ObserverContext<RegionCoprocessorEnvironment> ctx = null;
    for (RegionEnvironment env: coprocessors) {              // 循环Host上所有的CoprocessorEnvironment
      if (env.getInstance() instanceof RegionObserver) {    // Region级别的操作
        ctx = ObserverContext.createAndPrepare(env, ctx); // 获得在CoprocessorEnvironment里装载的Coprocessor.
        ((RegionObserver)env.getInstance()).prePut(ctx, put, edit, writeToWAL);   // 调用Coprocessor的相应方法
        bypass |= ctx.shouldBypass();                        // 是否需要跳过? 返回true则不会执行HRegion的put操作
        if (ctx.shouldComplete())    break;                  // 是否完成? true表示接下来的coprocessors不会执行了．到此为止.
      }
    }
    return bypass;
  }
```

这里面还有一个问题, CoprocessorHost 应该有对应 Coprocessor 事件发生的操作相同的方法．因为当事件发生时，是以 CoprocessorHost 为入口的．只有 CoprocessorHost 提供方法入口，才可以执行到 Coprocessor 相应的方法的.

## RegionCoprocessorHost

前面说过 Host 应该提供 Environment 需要的上层组件. Host 一般在 HBase 组件初始化时也一起初始化．比如新建 HRegion 的时候也会新建 RegionCoprocessorHost，这样就可以把当前 HRegion 对象赋值给 RegionCoprocessorHost.

```
public class HRegion implements HeapSize {
  final RegionServerServices rsServices;
  private RegionServerAccounting rsAccounting;
  private RegionCoprocessorHost coprocessorHost;    // Coprocessor host

  public HRegion(Path tableDir, HLog log, FileSystem fs, Configuration confParam,
    final HRegionInfo regionInfo, final HTableDescriptor htd, RegionServerServices rsServices) {
```

```
    if (rsServices != null) {
      this.rsAccounting = this.rsServices.getRegionServerAccounting();
      this.coprocessorHost = new RegionCoprocessorHost(this, rsServices, conf);
    }
  }
}
```

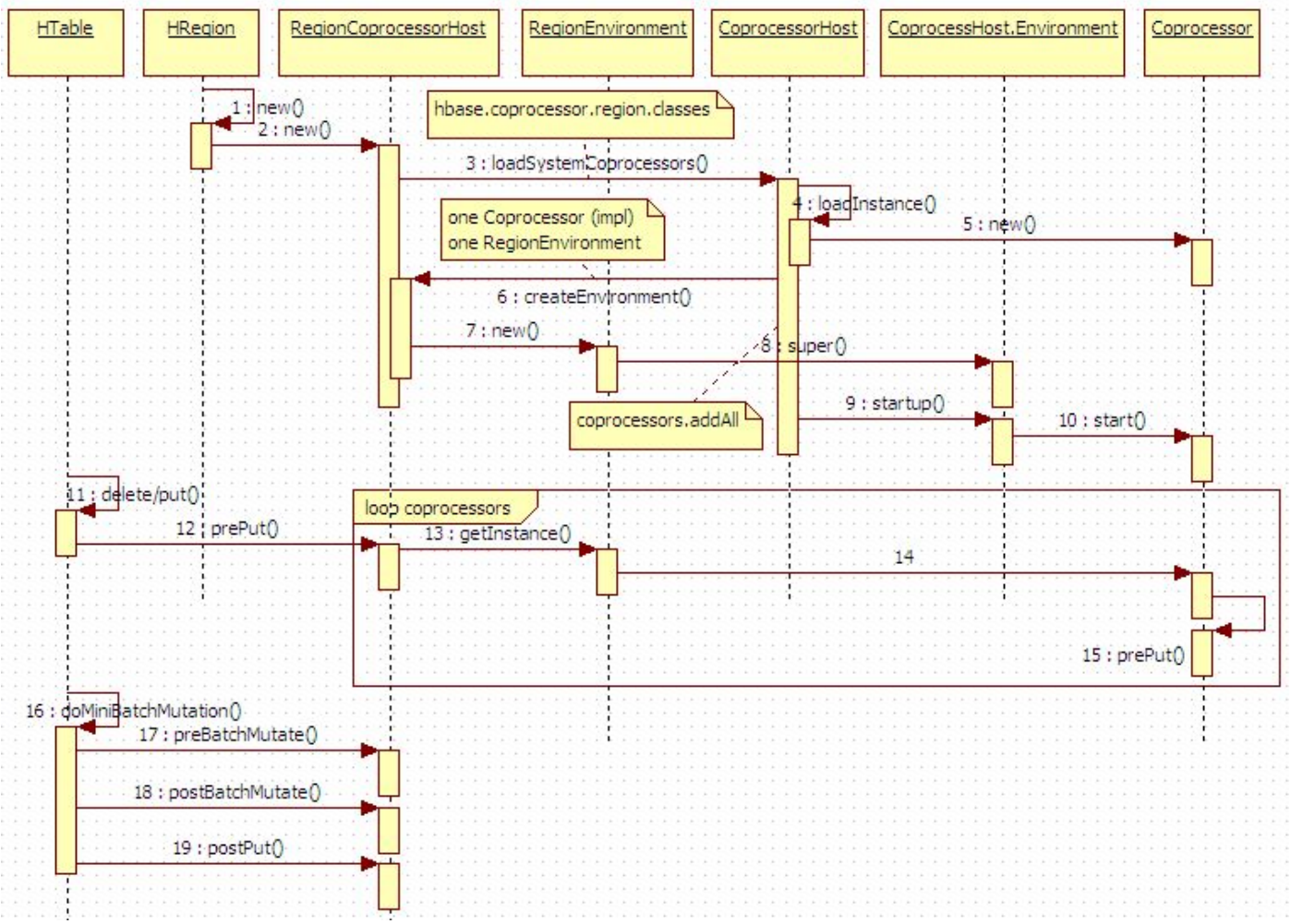
CoprocessorHost 实现类除了将上次组件设置到自己的成员变量中，还要负责加载配置的 Coprocessor.

```
public class RegionCoprocessorHost extends CoprocessorHost<RegionCoprocessorHost.RegionEnvironment>{
  // The shared data map. Host要提供HRegion和HMaster实例给Environment(创建时赋值), Environment才能获取到上层组件.
  private static ReferenceMap sharedDataMap = new ReferenceMap(AbstractReferenceMap.HARD, AbstractReferenceMap.WEAK);
  RegionServerServices rsServices;     /** The region server services */
  HRegion region;                      /** The region */

  public RegionCoprocessorHost(final HRegion region, final RegionServerServices rsServices, final Configuration conf) {
    this.conf = conf;
    this.rsServices = rsServices;
    this.region = region;
    this.pathPrefix = Integer.toString(this.region.getRegionInfo().hashCode());
    // load system default cp's from configuration.  从配置文件中加载系统默认的Coprocessor
    loadSystemCoprocessors(conf, REGION_COPROCESSOR_CONF_KEY);
    // load system default cp's for user tables from configuration.  加载用户表特定的系统级Coprocessor
    if (!HTableDescriptor.isMetaTable(region.getRegionInfo().getTableName())) {
      loadSystemCoprocessors(conf, USER_REGION_COPROCESSOR_CONF_KEY);
    }
    // load Coprocessor From HDFS  从HDFS中加载Coprocessor.  通过Configuration.setValue设置Coprocessor的jar包路径|类名|级别
    loadTableCoprocessors(conf);
  }

  public RegionEnvironment createEnvironment(Class<?> implClass, Coprocessor instance, int priority, int seq, Configuration conf) {
    ConcurrentMap<String, Object> classData;
    synchronized (sharedDataMap) {  // make sure only one thread can add maps
      classData = (ConcurrentMap<String, Object>)sharedDataMap.get(implClass.getName());
      if (classData == null) {
        classData = new ConcurrentHashMap<String, Object>();
        sharedDataMap.put(implClass.getName(), classData);
      }
    }
    return new RegionEnvironment(instance, priority, seq, conf, region, rsServices, classData);
  }
}
```

将 HRegion 传递给 RegionCoprocessorHost，在创建 CoprocessorEnvironment 时，赋值给 CoprocessorEnvironment. 这样通过 CoprocessorEnvironment 就可以获取到 HRegion 的引用对象了,同理 RegionServerServices 也是如此.

RegionCoprocessorHost 的 createEnvironment 实现了基类 CoprocessorHost 的抽象方法.
CoprocessorHost 中使用抽象方法的原因是：对于不同的 CoprocessorHost 实现类有不同的 CoprocessorEnvironment. 所以创建 CoprocessorEnvironment 的具体实现应该交给子类来实现. CoprocessorHost 基础框架把加载的过程都完整地确定了下来(实例化,装载环境,启动 Coprocessor 等)，具体里面是创建哪个环境是交由不同 Host 实现类确定的.

至此 Coprocessor 相关类都介绍完毕，下图是 HTable.put()经过 Coprocessor 相关类的流程图:

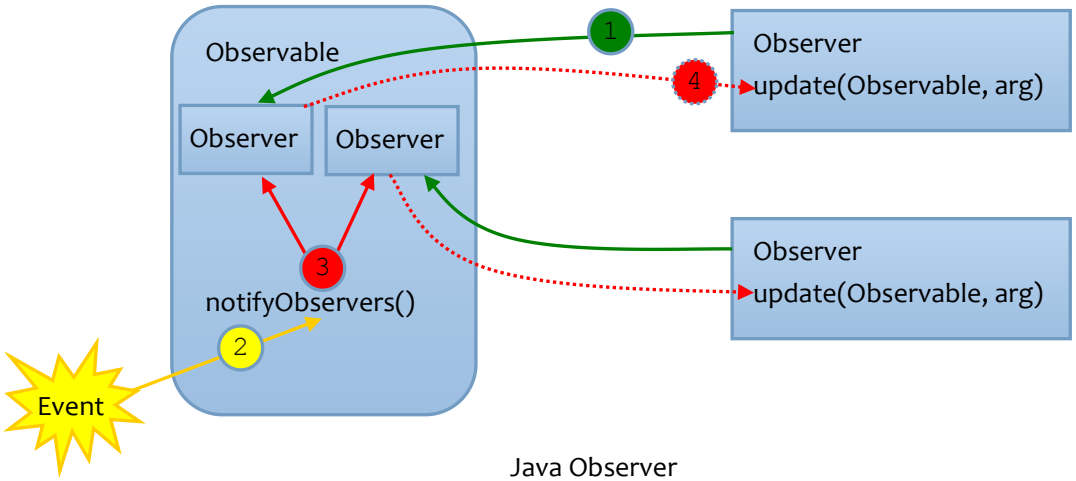


# Observer-Coprocessor 监听器类型的协处理器

Coprocessor 分为 Observer 监听器和 EndPoint 终端. Observer 又分为[Master|RegionServer|Region|WAL]ObServer.
这些 Observer 都是接口类型，同时还有接口的空实现：抽象类 BaseRegionObserver, BaseMasterObserver.

提供抽象类的好处是用户只要继承(extends)抽象类，重载合适的事件函数来实现自己的功能.
如果实现(implements)接口，则用户需要实现接口中定义的所有方法，但是并不是所有的方法我们都需要.
注意到 RegionServerObserver 和 WALObserver 并没有抽象实现，是因为这两个 Observer 里的方法个数本身就很少.

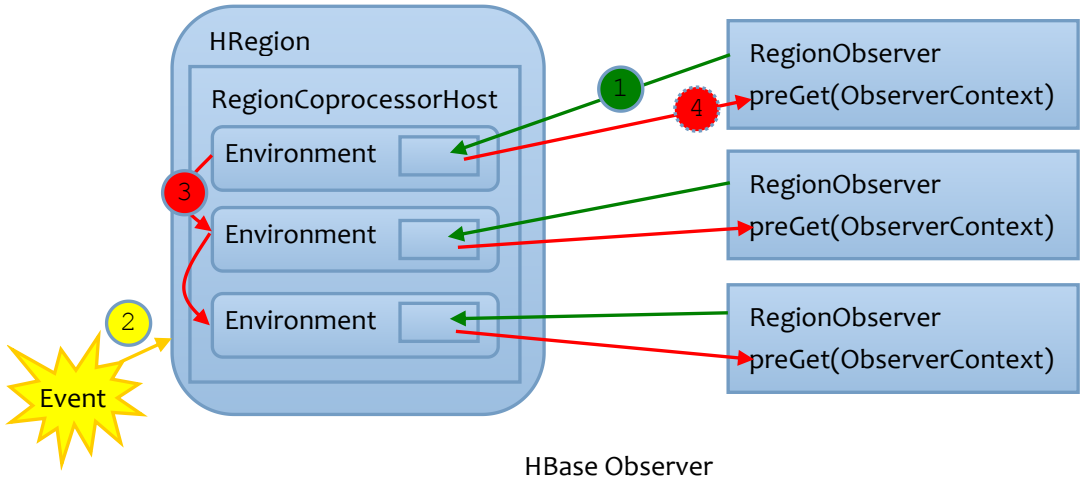
Java 观察者模式:




Java Observer

① Observable.addObserver(Observer) 注册观察者 Observe 到 Observable 上
② Event change on Observable. 有事件作用于 Observable，导致 Observable 状态发生改变
③ notifyObservers() 通知所有注册到 Observable 上的 Observers
④ for each Observer: update(Observable, arg) 每个观察者就都可以收到 Observable 改变后的状态.

问题：既然 Coprocessor 中的一种实现是 Observer. 那么 HBase 中充当 Observable 的哪个/哪些类?




HBase Observer

① CoprocessorHost.createEnvironment() 将 Coprocessor 装载环境，并让环境启动 Coprocessor.
② HTable 客户端事件发生/或者 region 生命周期变化，比如调用了 preGet()
③ CoprocessorHost.preGet() → for each CoprocessEnvironment in CoprocessorHost
④ 对每个 CoprocessorEnvironment, getInstance()→Coprocessor，并调用自定义的 Coprocessor.preGet()

# RegionObserver

RegionObserver 针对的是一个特定的 Region 级别的事件发生时，它们的钩子函数会被触发调用到.
客户端 API 调用都显示地从客户端应用中通过 RPC 调用传输到 RegionServer，并进而作用于特定的 Region.
除了客户端 API 调用，Region 自身生命周期的变化也会触发 RegionServer 中的某些钩子函数.

| 状态 | 描述 | 触发者/执行者 |
|---|---|---|
| Offline | Region 下线 | |
| Pending Open | 打开 Region 的请求已经发送到了服务器(RegionServer) | HMaster |
| Opening | 服务器开始打开/正在打开 Region | RegionServer |
| Open | Region 已经打开，并且完全可以使用 | |
| Pending Close | 关闭 Region 的请求已经发送到服务器 | HMaster |
| Closing | 服务器正在处理要关闭的 Region | RegionServer |
| Closed | Region 已经关闭了 | |
| Splitting | 服务器开始切分 Region | RegionServer |
| Split | Region 已经切分完毕 | |

Region 的各种状态均通过 HMaster 触发，使用 AssignmentManager 进行管理. 由于 Region 存在于 RegionServer.
所以 HMaster 触发 Region 状态改变，通过 RPC 调用发送给 RegionServer，由 RegionServer 完成这个操作.
Region 状态的改变也可以由 RegionServer 直接发起，比如 Region 进行拆分的过程(Splitting).

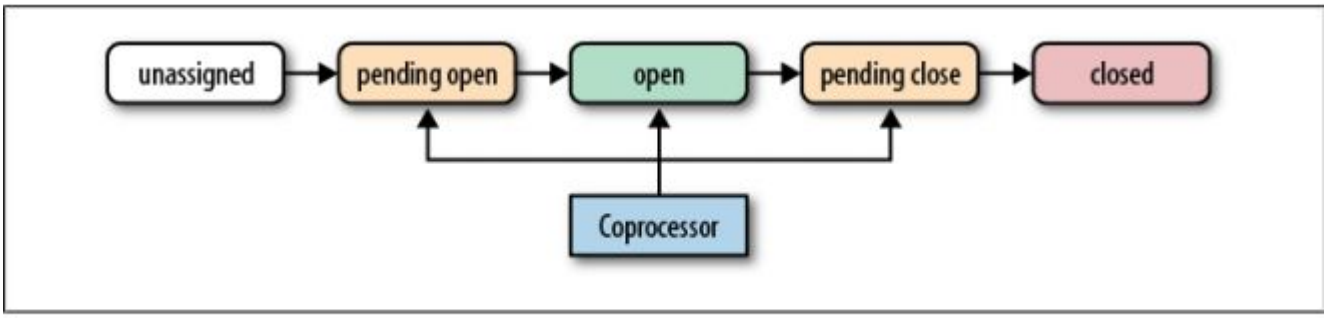



*Figure 4-4. The coprocessor reacting to life-cycle state changes of a region*

1. region 将要被打开时处于 Pending Open 状态.
   RegionObserver 的 preOpen/postOpen()会在 region 被打开前和刚刚打开后被调用.
2. region 经过 Pending Open，在 Open 状态之前，RegionServer 可能需要从 WAL 中 replay 记录到 region 中
   (RegionServer 失效，将 HLog 分发到每个 Region，每个 Region 被 HMaster 分配到其他的 RegionServer)
   这时 RegionServer 的 pre/postWALRestore()可以细粒度地控制在 WAL 重做时哪些修改需要被实施.
3. 当一个 Region 被部署到 RegionServer 上，并可以正常工作时，处于 Open 状态. 这时客户端 API 就可以运用到
   这个 Region 上. 对于 Region 自身而言的事件有: flush, compact, split 都有对应的 pre 和 post 方法.
4. region 在关闭前后，RegionServer 的 pre/postClose()会被触发执行.
   比如 region 由于负载均衡被 HMaster 移动到其他 RegionServer 上. 原先的 RegionServer 就应该关闭 region.
   或者 RegionServer 被撤销而失效，其上的所有 Region 都应该转移到其他 RegionServer 上，并关闭原先的 region.

客户端 API 调用最终调用到 HRegion 的相应方法，用户可以在这些调用执行前后拦截它们. 类似于拦截器，拦截器在
预处理时可以决定是否处理 HRegion 的相应方法，或者执行到某个拦截器后，不想执行其他的拦截器了.
HBase 使用了 ObserverContext 上下文来控制拦截器对 HBase 当前服务进程的影响. 比如跳过当前服务进程的机制.

# ObserverContext 上下文

所有的 Coprocessor 在执行时共用一个 ObserverContext，并随着环境一起变化.

```
/**Carries the execution state for a given invocation of an Observer coprocessor method.
 * The same ObserverContext instance is passed sequentially to all loaded coprocessors for a given Observer method trigger,
 * with the CoprocessorEnvironment reference swapped out for each coprocessor. */
public class ObserverContext<E extends CoprocessorEnvironment> {
  private E env;
  private boolean bypass;
  private boolean complete;

  public E getEnvironment() { return env; }
  public void prepare(E env) { this.env = env; }

  /**Call to indicate that the current coprocessor's return value should be used in place of the normal HBase obtained value. */
  public void bypass() {
    bypass = true;
  }
  /**Call to indicate that additional coprocessors further down the execution chain do not need to be invoked. */
  public void complete() {
    complete = true;
  }

  /**Instantiates a new ObserverContext instance if the passed reference is null and sets the environment in the new or existing instance.
   * This allows deferring the instantiation of a ObserverContext until it is actually needed.
   * @param env The coprocessor environment to set
   * @param context An existing ObserverContext instance to use, or null to create a new instance
   * @param <T> The environment type for the context
   * @return An instance of ObserverContext with the environment set */
  public static <T extends CoprocessorEnvironment> ObserverContext<T> createAndPrepare(T env, ObserverContext<T> context) {
    if (context == null) { context = new ObserverContext<T>(); }
    context.prepare(env);
    return context;
  }
}
```

以 RegionCoprocessorHost.preGet()为例, RegionServer 的每个回调函数都有参数 ObserverContext ctx:

```
  public boolean preGet(final Get get, final List<KeyValue> results) {
    boolean bypass = false;
    ObserverContext<RegionCoprocessorEnvironment> ctx = null;
    for (RegionEnvironment env: coprocessors) {
      if (env.getInstance() instanceof RegionObserver) {
        ctx = ObserverContext.createAndPrepare(env, ctx);   // 如果已经存在ctx(循环第一次会实例化), 接下来会使用这个ctx
        ((RegionObserver)env.getInstance()).preGet(ctx, get, results);   // 但是ctx其中的Environment是每次都不一样的.
        bypass |= ctx.shouldBypass();
        if (ctx.shouldComplete()) break;
      }
    }
    return bypass;
  }
```

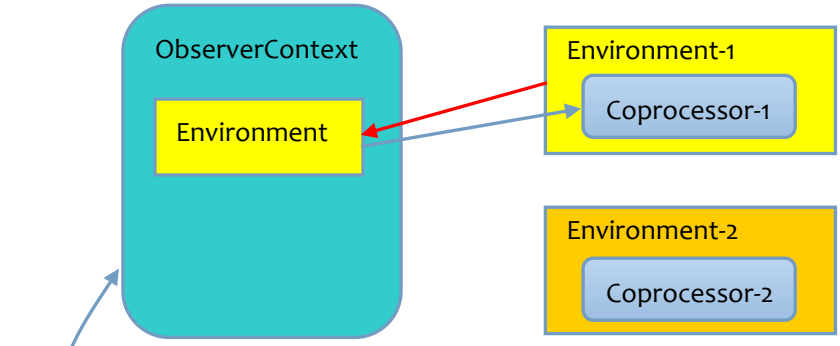
假设我们为 hbase.coprocessor.region.classes 配置了两个自定义的 Coprocessor 实现了 RegionObserver 接口.
每个 Coprocessor 都有属于自己的 RegionEnvironment，由 RegionCoprocessorHost 加载后加入到 coprocessors 中.
当客户端 API 发起 HTable.get()时，在调用 HRegion.get()之前，会调用 RegionCoprocessorHost.preGet()方法:

1. 首先获取 coprocessors 集合. 集合的每个元素是 RegionEnvironment.
2. 通过 getInstance()可以获取到对应的 Coprocessor: 即用户自定义的 RegionObserver 实现类.
3. 获取到 Coprocessor 实现类后，调用自定义的回调方法.
4. 通常，在回调函数里我们想要获得当前对象在整个运行环境中的其他对象.
   因为我们知道 CoprocessorEnvironment 里通过 get()保存了很多其他对象的引用.
5. 可以在回调函数里直接传递当前正在处理的 CoprocessorEnvironment. 但是...
6. 我想要在当前 Coprocessor 运行完后，不想执行其他的 Coprocessor 了，怎么办?
   或者执行完当前 Coprocessor，我不想执行当前服务进程本身的过程. 直接返回当前 Coprocessor 的结果.
7. ObserverContext 封装了 CoprocessorEnvironment，并且提供了标志位 bypass 和 complete 用来控制上述情景.

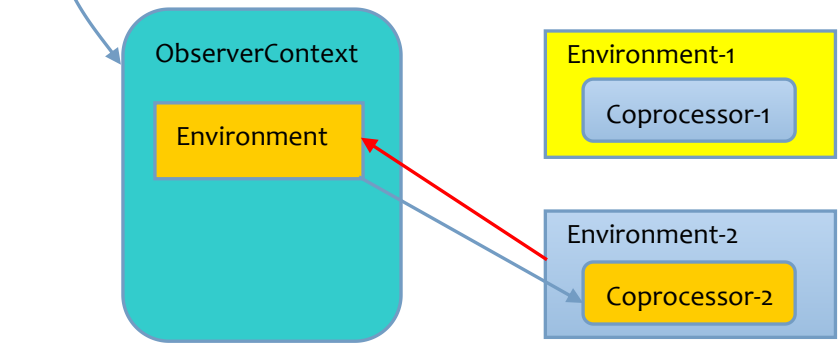
**ObserverContext VS CoprocessorHost**

ObserverContext 在回调函数执行过程中，一次只允许存在一个 CoprocessorEnvironment.
而 CoprocessorHost 是多个 CoprocessorEnvironment 的载体. 保存了所有 Coprocessor 对应的所有运行时环境.

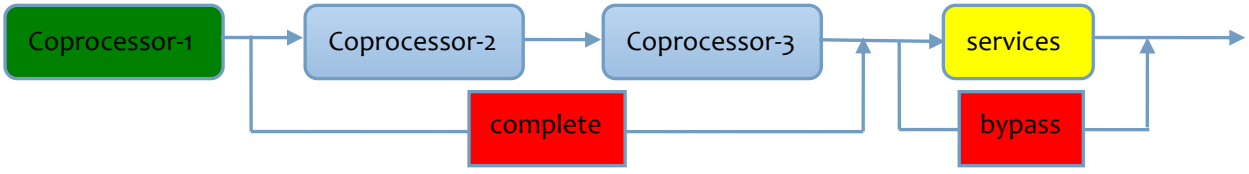
第一个 Coprocessor 运行时, ObserverContext 搭载的是第一个 Coprocessor 对应的 CoprocessorEnvironment-1
因为初始时, ObserverContext=null，所以第一个 Coprocessor 工作时，会初始化 ObserverContext 对象:




第二个 Coprocessor 工作时, ObserverContext 搭载的是第二个 Coprocessor 对应的 CoprocessorEnvironment-2.
由于 ObserverContext 对象已经存在，所以不会再创建对象了，而只是更新了里面的 CoprocessorEnvironment.




ObserverContext 的两个标志位的运用场景图:



## EndPoint(TODO)

# RowLock 行锁

HBase 对 put(), delete(), checkAndPut()这样的修改操作是独立的，对于每一行保证行级别的操作是原子性的.
RegionServer 提供了行锁 row lock(从名称可以看出针对单独一行的锁)的特性.
这个特性保证了只有一个客户端能获取一行数据相应的锁，同时对该行数据进行修改.
RowLock 分为服务器端隐式加锁，比如创建 Put 时没有传递 RowLock 参数. 服务器会在调用期间创建一个锁.
客户端也可以显示地对单行数据的多次操作进行加锁，注意一旦不需要锁时，必须通过 unlockRow()来释放锁.

当一个锁被服务器端或客户端显示获取后，其他所有想要对这行数据加锁的客户端将会等待，直到当前锁被释放或者锁的租约超时. 客户端使用 lockRow()但忘记 unlockRow()，锁的租约可以确保超时或释放长时间被占用的锁.
下面的示例展示了客户端显示地创建 RowLock，并且在拥有行锁的情况下，对同一行数据进行了多次 put 操作:

```
    public static void rowlock() throws Exception {
        Configuration conf = HBaseConfiguration.create();
        HTable table = new HTable(conf, "testtable");
        RowLock lock = table.lockRow(Bytes.toBytes("row-001"));

        // 客户端显示对单行数据的多次操作进行加锁.
        Put put1 = new Put(Bytes.toBytes("row-001"), lock);
        put1.add(Bytes.toBytes("fam1"), Bytes.toBytes("qual01"), Bytes.toBytes("value1"));
        table.put(put1);

        Put put2 = new Put(Bytes.toBytes("row-001"), lock);
        put2.add(Bytes.toBytes("fam1"), Bytes.toBytes("qual01"), Bytes.toBytes("value2"));
        table.put(put2);

        table.unlockRow(lock);      // 释放锁
    }
```

# HTable.lockRow(row)

客户端通过 lockRow 对某一行进行加锁，需要指定特定行的行键. 并返回 RowLock 对象.
接下来对 Put, Delete 等对象的构造函数传入这个 RowLock，就可以确保操作在 RowLock 之间进行.
最后当锁不需要时，调用 unlockRow()，并传递当前使用的 RowLock 对象释放锁.

```
public RowLock lockRow(final byte [] row) {
   return new ServerCallable<RowLock>(connection, tableName, row, operationTimeout) {
        public RowLock call() throws IOException {
           long lockId = server.lockRow(location.getRegionInfo().getRegionName(), row);
           return new RowLock(row,lockId);
        }
     }.withRetries();
}
public void unlockRow(final RowLock rl) {
   new ServerCallable<Boolean>(connection, tableName, rl.getRow(), operationTimeout) {
        public Boolean call() throws IOException {
           server.unlockRow(location.getRegionInfo().getRegionName(), rl.getLockId());
           return null;
        }
     }.withRetries();
}
```

RowLock 由行键 row 和唯一的 lockId 组成. lockId 会由 RegionServer 对当前行锁定并返回给客户端.
HTable.lockRow()回调函数中的 server 和 location 前面已经说过在 connect()中时通过行键 row 被定位到.

```
public long lockRow(byte[] regionName, byte[] row) throws IOException {
    requestCount.incrementAndGet();
        HRegion region = getRegion(regionName);
        Integer r = region.obtainRowLock(row);                    // 客户端显示获取锁
        long lockId = addRowLock(r, region);
        return lockId;
}

Map<String, Integer> rowlocks = new ConcurrentHashMap<String, Integer>();
private Leases leases;                    // Leases  客户端加锁后持有一个租约，租约超时后, RegionServer 会强制释放锁
protected long addRowLock(Integer r, HRegion region) {
    long lockId = -1L;
    lockId = rand.nextLong();
    String lockName = String.valueOf(lockId);
    rowlocks.put(lockName, r);          // key构成RowLock的lockId,返回给HTable, value是上面obtainRowLock的返回值
    this.leases.createLease(lockName, new RowLockListener(lockName, region));
    return lockId;
}
```

因为行键 row 实际存放于 HRegion 中，所以要通过 HRegion 来为指定的 row 获取 lockId. 因为锁是要加在行上的.

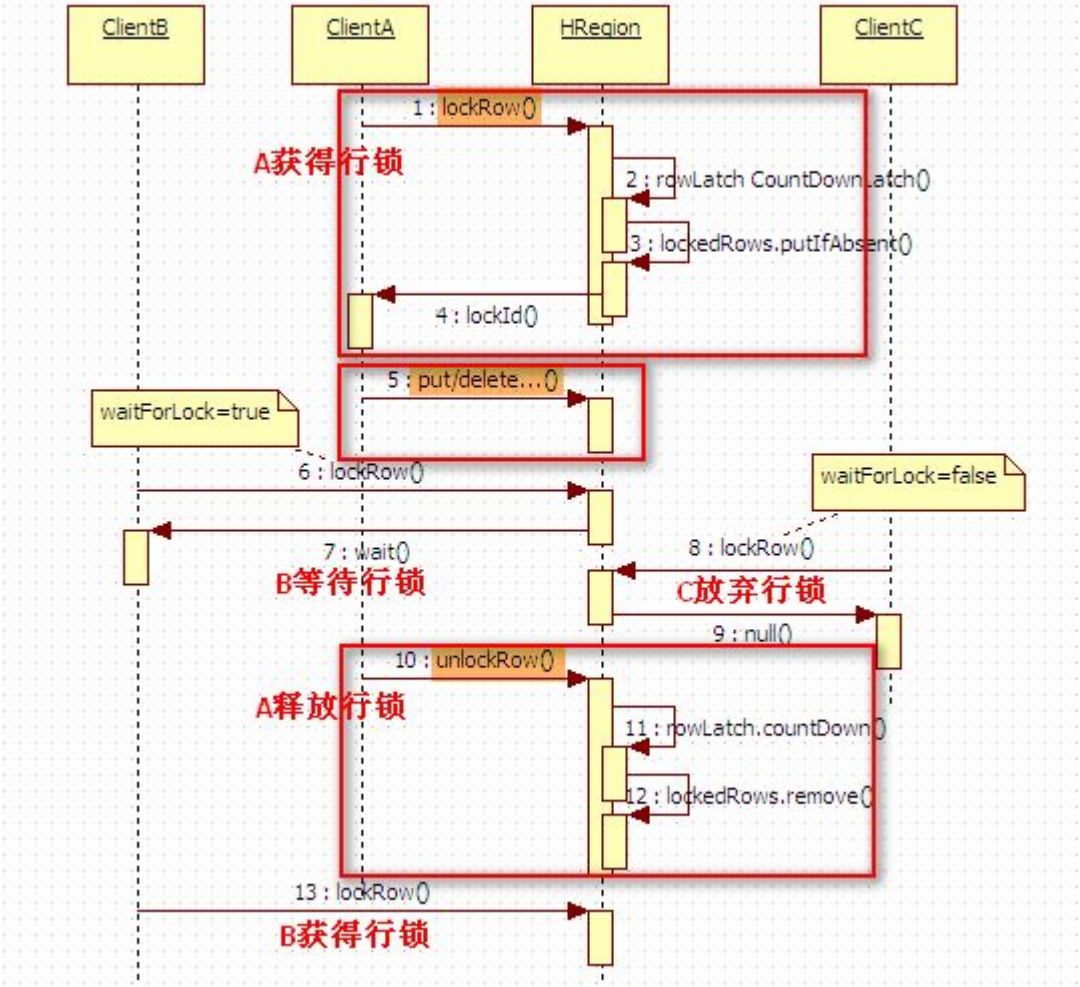
```
public Integer obtainRowLock(final byte [] row) throws IOException {
    startRegionOperation();              // 对下面的方法进行加锁，类似于同步
    this.writeRequestsCount.increment();
    this.opMetrics.setWriteRequestCountMetrics( this.writeRequestsCount.get());
    try {
        return internalObtainRowLock(new HashedBytes(row), true);   // 客户端显示获取锁，锁如果被占用，会等待，而不是放弃
    } finally {
        closeRegionOperation();          // 释放锁. 注意这是方法上的锁，不是行锁哦.
    }
}
```

HRegion 获取行锁使用了 CountDownLatch 闭锁来实现，初始时计数器的值为 1. 锁只有两种状态：加锁和释放.
CountDownLatch 可以看作一个计数器，只不过这个计数器的操作是原子操作，同时只能有一个线程去操作这个计数器，也就是同时只能有一个线程去减这个计数器里面的值. 你可以向 CountDownLatch 对象设置一个初始的数字作为计数值，任何调用这个对象上的 await()方法都会阻塞，直到这个计数器的计数值被其他的线程减为 0 为止.

CountDownLatch 的一个非常典型的应用场景是：有一个任务想要往下执行，但必须要等到其他的任务执行完毕后才可以继续往下执行. 假如我们这个想要继续往下执行的任务调用一个 CountDownLatch 对象的 await()方法，其他的任务执行完自己的任务后调用同一个 CountDownLatch 对象上的 countDown()方法，这个调用 await()方法的任务将一直阻塞等待，直到这个 CountDownLatch 对象的计数值减到 0 为止.

当一个客户端 A 占用了当前行的锁时，另一个客户端 B 也要操作那一行时，必须等待客户端 A 释放行锁才可以继续. 当然如果客户端 B 不想等待 waitForWork=false，那么就可以放弃这个锁，即放弃对这一行的操作.

下图模拟了三个客户端对同一行(显示)获取锁的过程. 其中 A 首先调用 lockRow 最先获得行锁.
在并发环境下，要对某一行进行操作必须先获得这一行的锁. 当指定的行被 A 锁住之后, B 和 C 都想要对这一行加锁. 其中 B 会等待获取这个行锁，而 C 不想等待获取：如果这次调用获取不到，我就不想要这个锁了.
B 会一直等待直到当前占用锁的 A 显示调用 unlockRow 或者租约超时释放掉这个锁之后才能获取到行锁.




加锁时会将行键和每次加锁新创建的 CountDownLatch 保存到 Map lockedRows 中，表示已经加锁的行.
当释放锁时，会从 lockedRows 中取出行键对应的 CountDownLatch，调用其 countDown()方法使其 value=0.
这样处于等待状态的其他客户端(B)就可以接着在 wait()方法后继续执行获取锁的过程.

也就是说对于同一个 CountDownLatch 对象，当不断调用 countDown()使 value=0 时，会触发 wait()方法被调用.

ClientA→lockRow():

```
CountDownLatch rowLatch = new CountDownLatch(1);
while (true) {
  CountDownLatch existingLatch = lockedRows.putIfAbsent(rowKey, rowLatch);
  if (existingLatch == null) break;
}
```

put

ClientB→lockRow():

```
CountDownLatch rowLatch = new CountDownLatch(1);
while (true) {
  CountDownLatch existingLatch = lockedRows.putIfAbsent(rowKey, rowLatch);
  if (existingLatch != null)  {
    if (!waitForLock) { return null; }
    existingLatch.await(this.rowLockWaitDuration, TimeUnit.MILLISECONDS);
  }
}
```

get

notify

ClientA→unlockRow():

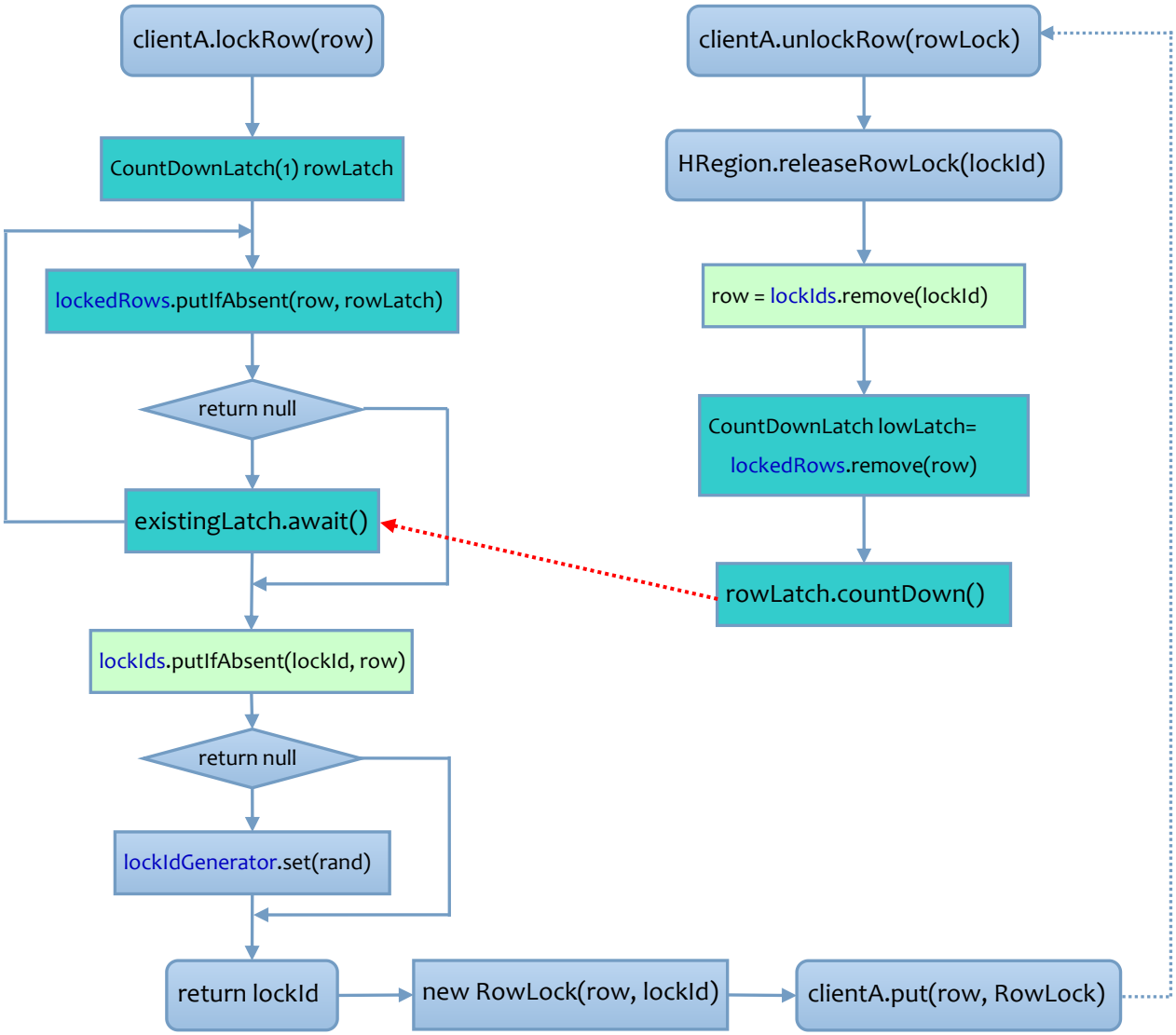
```
CountDownLatch rowLatch = lockedRows.remove(rowKey);
rowLatch.countDown();
```

ConcurrentHashMap 的 putIfAbsent(key, value)顾名思义: 1)如果 Map 中不存在 key，则将<key,value>放入 Map 中，2) 如果存在则直接取出 key 对应的 value. 因此 value 可以看做是 key 的初始值.
特别要注意这个方法的返回值. 1)Map 中不存在 key 时，返回 null; 2) 存在 key 时，返回已经存在的 key 对应的 value.
另外还有一个知识点: 往 Map 中放入一个不存在的 key 时，返回值为 null，而不是新放入的 key 对应的 value.

```
if (!map.containsKey(key))
    return map.put(key, value);    // A 调用时放入 Map，返回值=null.
else
    return map.get(key);           // 在 A 持有锁的过程中, B 调用时，由于 Map 中存在 key 的引用，返回 A 持有的锁
```

对应到获取行锁的过程, key 对应行键. 2) 如果 key 已经存在，表示有其他客户端正在对这一行进行加锁. 需等待!
1) 如果 key 不存在，表示目前为止没有其他客户端占用这一行的锁，当前的客户端就可以对这一行进行加锁.
那么就要跳出 while 循环，生成一个唯一未使用过的 lockId 放入 lockIds 中.

```
private final ConcurrentHashMap<HashedBytes, CountDownLatch> lockedRows = new ConcurrentHashMap();
private final ConcurrentHashMap<Integer, HashedBytes> lockIds = new ConcurrentHashMap();
private final AtomicInteger lockIdGenerator = new AtomicInteger(1);

/** Obtains or tries to obtain the given row lock.
 * waitForLock if true, will block until the lock is available. Otherwise, just tries to obtain the lock and returns null if unavailable. */
private Integer internalObtainRowLock(final HashedBytes rowKey, boolean waitForLock) {
  checkRow(rowKey.getBytes(), "row lock");   // 检查row-key在当前Region的范围内
  startRegionOperation();
  try {
    CountDownLatch rowLatch = new CountDownLatch(1);     // 客户端每次要获取行锁，都要创建新的闭锁
    // loop until we acquire the row lock (unless !waitForLock) 循环，直到获取到行锁(除非不想等待)
    while (true) {
      CountDownLatch existingLatch = lockedRows.putIfAbsent(rowKey, rowLatch);
      if (existingLatch == null) {     // lockedRows之前没有rowKey，即目前为止没有其他的客户端占用这一行的锁
        break;                         // 当前调用的客户端获得锁了!
      } else {                         // row already locked，当前行已经被其他的客户端加锁了. 要么等待，要么放弃.
        if (!waitForLock) { return null; }
        existingLatch.await(this.rowLockWaitDuration, TimeUnit.MILLISECONDS);
      }
    }

    // loop until we generate an unused lock id 循环，直到生成一个未使用过的lockId
    while (true) {
      Integer lockId = lockIdGenerator.incrementAndGet();
      HashedBytes existingRowKey = lockIds.putIfAbsent(lockId, rowKey);
      if (existingRowKey == null) { // lockIds中不存在生成的lockId，则立即返回. 因为lockId对于客户端必须是唯一的.
        return lockId;
      } else {                         // lockId already in use, jump generator to a new spot
        lockIdGenerator.set(rand.nextInt());
      }
    }
  } finally {
    closeRegionOperation();
  }
}
```




HTable.lockRow() → HTable.put() → HTable.unlockRow()

## HTable.unlockRow(RowLock)

客户端释放锁时调用 HTable.unlockRow(RowLock)，通过 RowLock 获得 lockId. 然后对指定的行键进行解锁操作.

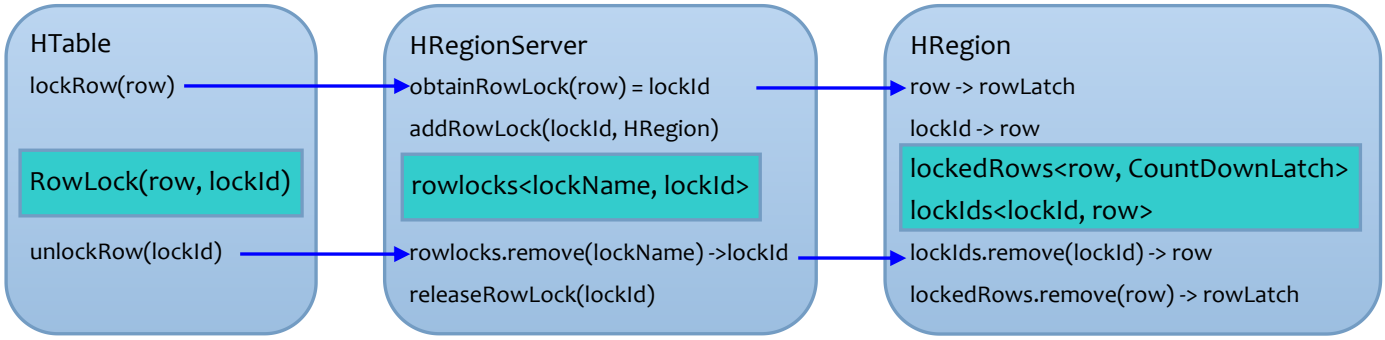
```
public void unlockRow(byte[] regionName, long lockId) throws IOException {
        HRegion region = getRegion(regionName);
        String lockName = String.valueOf(lockId);
        Integer r = rowlocks.remove(lockName);          // 从rowlocks中移除lockId，返回值为obtainRowLock中的lockId
        region.releaseRowLock(r);                       // 客户端显示释放锁
        this.leases.cancelLease(lockName);
}
```

HRegionServer 的 rowlocks<lockName, lockId> key 是随机的 lockId, value 是 HRegion 获得的 lockId，这两个值不同.

```
protected long addRowLock(Integer r, HRegion region) {
    long lockId = -1L;
    lockId = rand.nextLong();
    String lockName = String.valueOf(lockId);
    rowlocks.put(lockName, r);        // key构成RowLock的lockId,返回给HTable, value是上面obtainRowLock的返回值
    this.leases.createLease(lockName, new RowLockListener(lockName, region));
    return lockId;
}
```

HRegion.releaseRowLock(rockId)

```
public void releaseRowLock(final Integer lockId) {
    HashedBytes rowKey = lockIds.remove(lockId);                  // 移除lockIds的lockId，返回row-key
    CountDownLatch rowLatch = lockedRows.remove(rowKey);  // 移除lockedRows的row-key，返回CountDownLatch
    rowLatch.countDown();              // 调用CountDownLatch.countDown()使value=0，阻塞在该对象上的其他客户端可以继续工作
}
```

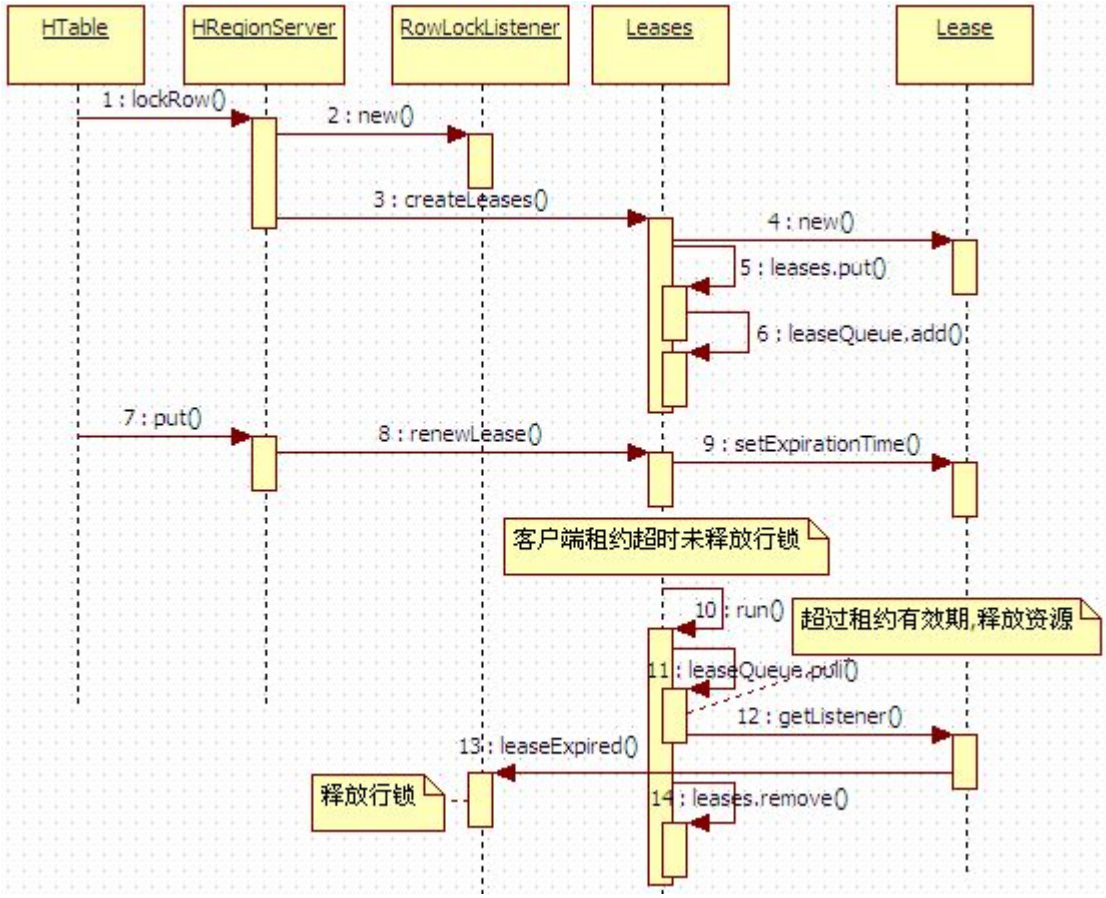



注: HTable.lockId = HRegionServer.lockName,  HRegionServer.lockId = HRegion.lockId

## Lease

外部客户端请求服务器(比如发送心跳，对行进行加锁等)，一般会在服务器占用一些资源．正常情况下当不需要这些资源后，外部客户端应该手动调用相关方法来释放自己先前占用的资源．但是有些情况下，客户端调用相关方法释放资源失败，或者忘记了释放资源．那么服务器端应该有一定的轮询机制来确保被占用的资源及时释放．

这种机制的实现：在客户端请求服务器资源时，就启动另外一个租约线程来跟踪客户端的活动情况．当客户端继续向服务器发送心跳时，会不断更新租约信息，表示客户端当前处于活动状态．租约有一个过期时间，如果超过这个过期时间，客户端没有向服务器发送心跳，服务器就认为客户端已经死亡，服务器会手动释放客户端占用的资源．



## RowLockListener租约监听器

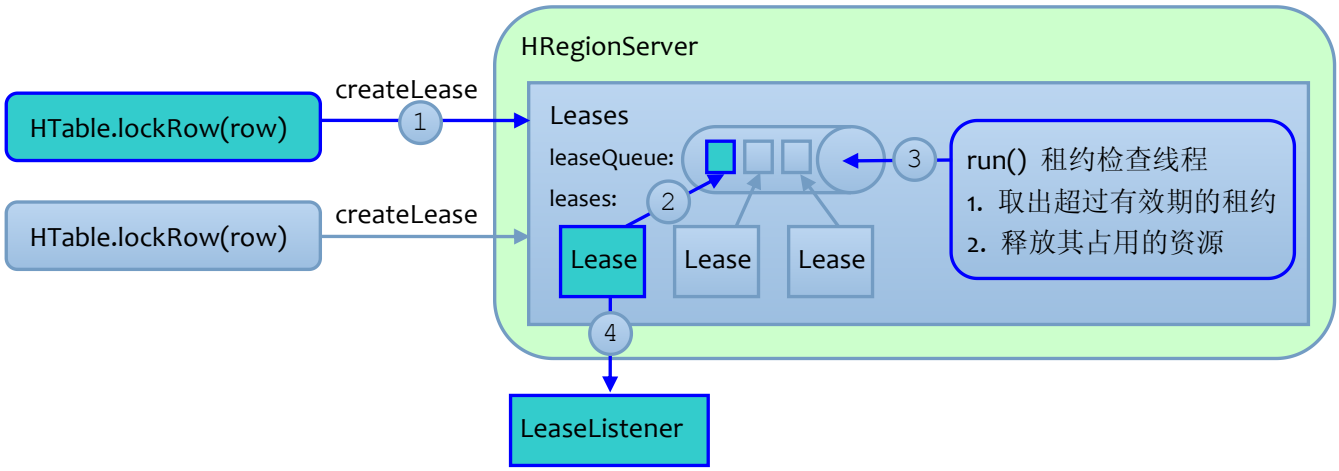
HTable对行进行加锁时，创建相应的租约线程. LeaseListener的实现类为RowLockListener:

```
this.leases.createLease(lockName, new RowLockListener(lockName, region));
```

HTable 对指定的行解锁时，会取消租约.

```
public void unlockRow(byte[] regionName, long lockId) throws IOException {
    HRegion region = getRegion(regionName);
    String lockName = String.valueOf(lockId);
    Integer r = rowlocks.remove(lockName);        // 从rowlocks中移除lockId，返回值为obtainRowLock中的lockId
    region.releaseRowLock(r);                     // 释放锁
    this.leases.cancelLease(lockName);
}
```

那么租约监听器完成的工作应该和释放行锁的工作是一样的. 前者为线程自动完成，后者为客户端手动调用.




① 客户端加行锁时，创建租约，以及租约监听器(用于租约失效时的具体操作，比如释放行锁)
② 租约对象 Lease 加入到 Leases 的队列和 Map 中. 由 Leases 同一管理(比如采用优先级队列)
③ 租约线程 Leases 要检查超过租约有效期的 Lease. 如果超时要释放其占用的资源
④ 获取超过租约有效期的租约的监听器，由监听器执行释放资源的具体操作.

```
Map<String, Integer> rowlocks = new ConcurrentHashMap<String, Integer>();

/**Instantiated as a row lock lease. If the lease times out, the row lock is released */
private class RowLockListener implements LeaseListener {
  private final String lockName;
  private final HRegion region;

  RowLockListener(final String lockName, final HRegion region) {
    this.lockName = lockName;
    this.region = region;
  }

  public void leaseExpired() {              // 租约失效时的操作: 释放被客户端占用的行锁
    Integer r = rowlocks.remove(this.lockName);
    if (r != null) {
      region.releaseRowLock(r);
    }
  }
}
```

## Lease 租约对象

```
static class Lease implements Delayed {          /** This class tracks a single Lease. */
    private final String leaseName;               // 租约名字, RowLock的lockId
    private final LeaseListener listener;         // 租约监听器，用于租约超时后的具体动作
    private long expirationTime;                  // 租约失效时间(失效的那一刻的时间点)，当前时间>失效时间时，租约失效

    Lease(final String leaseName, LeaseListener listener) {
      this(leaseName, listener, 0);               // 客户端创建租约时，失效时间还没有指定
    }

    public long getDelay(TimeUnit unit) {         // Delayed实现方法，值>0表示还有效，需等待. <=0表示租约到期,
      return unit.convert(this.expirationTime - System.currentTimeMillis(), TimeUnit.MILLISECONDS);
    }
    public int compareTo(Delayed o) {             // 实现比较方法.
      long delta = this.getDelay(TimeUnit.MILLISECONDS) - o.getDelay(TimeUnit.MILLISECONDS);
      return this.equals(o) ? 0 : (delta > 0 ? 1 : -1);
    }
    public void setExpirationTime(long expirationTime) {
      this.expirationTime = expirationTime;       // 设置租约失效时间，当将Lease添加到Leases时，设置该值
    }
  }
```

## Leases 租约管理器

```
public class Leases extends HasThread {
  private final int leasePeriod;    // length of time (milliseconds) that the lease is valid  租约有效期(1min). 在这个时间段内租约有效
  private final int leaseCheckFrequency;    // how often the lease should be checked (milliseconds)  租约检查间隔(10s)
  private volatile DelayQueue<Lease> leaseQueue = new DelayQueue<Lease>();
  protected final Map<String, Lease> leases = new HashMap<String, Lease>();
  private volatile boolean stopRequested = false;

  /** Creates a lease monitor  创建一个租约监听器. 由HRegionServer初始化时创建并启动线程  */
  public Leases(final int leasePeriod, final int leaseCheckFrequency) {
    this.leasePeriod = leasePeriod;
    this.leaseCheckFrequency = leaseCheckFrequency;
    setDaemon(true);
  }
  public void createLease(String leaseName, final LeaseListener listener) {
    addLease(new Lease(leaseName, listener));
  }
  public void addLease(final Lease lease) {         /** Inserts lease.    Resets expiration before insertion. */
    if (this.stopRequested) return;
    lease.setExpirationTime(System.currentTimeMillis() + this.leasePeriod);   // 租约失效时间=当前时间+租约有效期
    synchronized (leaseQueue) {
      leases.put(lease.getLeaseName(), lease);
      leaseQueue.add(lease);
    }
  }
```

更新租约：客户端在创建租约后，如果有继续对持有的资源操作，每次操作都会更新已经创建的租约的失效时间.
失效时间是用于租约管理器检查租约是否失效的判断依据．如果租约失效(租约检查时刻的当前时间>租约失效时间)，则租约管理器就会认为持有该租约的客户端很久没有对持有的资源进行操作，可能是客户端挂掉了.
租约管理就会对此租约对应的客户端的资源进行释放操作．由于检查时间一般配置的比较长，所以如果在检查时间间隔内，客户端在不需要资源的情况下没有手动释放资源，这些资源会一直存在直到被检查出来并被释放.

```
public void renewLease(final String leaseName) throws LeaseException {
    synchronized (leaseQueue) {
        Lease lease = leases.get(leaseName);
        if (lease == null || !leaseQueue.remove(lease)) {
            throw new LeaseException("lease '" + leaseName + "' does not exist or has already expired");
        }
        lease.setExpirationTime(System.currentTimeMillis() + leasePeriod);     // 更新租约时，更新租约的失效时间
        leaseQueue.add(lease);   // 先移除，然后更新时间，再添加，这个过程并没有创建新的Lease对象.
    }
}
```

下面举一个例子模拟租约到期时间和租约检查时间的比较，用于判断租约是否到期:

| 时刻 | 客户端 | 动作/说明 | 租约到期时间 | 租约检查时间(20s) | 租约超时? |
|---|---|---|---|---|---|
| 1 | A | createLease/创建租约 | 2013-12-17 17:02:00 | 2013-12-17 17:01:20 | N |
| 1 | B | createLease/创建租约 | 2013-12-17 17:02:00 | 2013-12-17 17:01:20 | N |
| 2 | A | | 2013-12-17 17:02:00 | 2013-12-17 17:01:40 | N |
| 2 | B | | 2013-12-17 17:02:00 | 2013-12-17 17:01:40 | N |
| 3 | A | put/更新租约 | 2013-12-17 17:03:00 | | |
| 4 | A | | 2013-12-17 17:03:00 | 2013-12-17 17:02:00 | N |
| 4 | B | | 2013-12-17 17:02:00 | 2013-12-17 17:02:00 | Y |
| 5 | A | | 2013-12-17 17:03:00 | 2013-12-17 17:02:20 | N |

上表中在时刻 1 时客户端 A,B 都创建了租约，租约到期时间=17:02:00．租约检查线程每个 20s 检查租约是否到期.
租约检查线程经过三个周期，当前时间=17:02:00，而客户端 B 的到期时间刚好也是这个时刻，所以此时 B 过期了.
由于在时刻 3 时，客户端 A 有其他的动作更新了其租约到期时间(延长了到期时间)．所以在时刻 4, A 并没有过期.

实现 Delayed 接口的 Lease 类的 getDelay()用于判断租约是否到期的算法:

```
    public long getDelay(TimeUnit unit) {          // Delayed实现方法，值>0表示还有效，需等待. <=0表示租约到期,
        return unit.convert(this.expirationTime - System.currentTimeMillis(), TimeUnit.MILLISECONDS);
    }
```

convert 用于计算值是否>0, <=0 的两个值分别是 expirationTime 和 currentTime.
其中 expirationTime 对应上表的租约到期时间, currentTime 对应租约检查时间为系统当前时间.
getDelay()的返回值如果>0，说明当前租约对象还没有到期．否则<=0 表示租约到期.

## LeaseCheck 租约检查

现在看下租约检查线程是如何工作的. 也许你会很奇怪, 这里怎么没有检查租约是否超时? 实际上 DelayedQueue 延迟队列在内部已经实现了这种逻辑. 参考温少的博文:http://www.cnblogs.com/jobs/archive/2007/04/27/730255.html

```
public void run() {
  while (!stopRequested || (stopRequested && leaseQueue.size() > 0) ) {
    Lease lease = leaseQueue.poll(leaseCheckFrequency, TimeUnit.MILLISECONDS); // 删除队列头元素, 并返回这个头元素
    if (lease == null) continue;
    // A lease expired.  Run the expired code before removing from queue since its presence in queue is used to see if lease exists still.
    if (lease.getListener() != null) {
      lease.getListener().leaseExpired();
    }
    synchronized (leaseQueue) {
      leases.remove(lease.getLeaseName());
    }
  }
  close();
}
```

DelayedQueue.poll 的参数 leaseCheckFrequency 为线程检查的间隔时间, 默认 10s 检查一次.
从延迟队列中 poll 出来的元素如果=null, 说明此时系统中的所有租约都还没超时(租约都还没到期, 比如时刻 1~2)
如果 poll 出来的租约对象不为 null, 那么 poll 出来的队列头部的元素是超时时间最长的租约. 这是因为延迟队列内部采用了优先级队列, 优先级队列的比较基准值为时间(即租约到期时间和当前时间的差值越大越先被取出来).
取出租约对象后, 首先获取其在创建租约对象时传入的租约监听器, 调用监听器的 leaseExpired()做具体的工作.

现在我们回到 HRegionServer.multi(MultiAction multi), 从 multi 中取出 List<Action> mutations 转换为带 Lock 的:

```
          List<Pair<Mutation,Integer>> mutationsWithLocks = Lists.newArrayListWithCapacity(mutations.size());
          for (Action<R> a : mutations) {
            Mutation m = (Mutation) a.getAction();
            Integer lock = getLockFromId(m.getLockId());  // 获取HRegion内部使用的lockId. 如果不是客户端加锁, 返回null
            mutationsWithLocks.add(new Pair<Mutation, Integer>(m, lock));
          }
```

getLockFromId 的 lockId(lockName 在 RegionServer 中生成)是 RowLock 的 lockId, 方法返回的是 HRegion 中的 lockId.
获取 HRegion 内部使用的 lockId 时, 也会更新客户端租约信息的租约失效时间.
如果 HTable 创建 Put 时, 不是使用客户端加锁 lockRow()的方式, 那么 Put 对象的 lockId=-1, 存储到 Pair 里为 null.

```
/**Method to get the Integer lock identifier used internally from the long lock identifier used by the client.
 * @param lockId long row lock identifier from client
 * @return intId Integer row lock used internally in HRegion */
Integer getLockFromId(long lockId) throws IOException {
  if (lockId == -1L) return null;                  // 没有采用客户端加锁, 返回null
  String lockName = String.valueOf(lockId);
  Integer rl = rowlocks.get(lockName);             // 注意返回的是HRegion内部使用的lockId, 而不是RowLock的lockId.
  if (rl == null) throw new UnknownRowLockException("Invalid row lock");
  this.leases.renewLease(lockName);
  return rl;
}
```

# HRegion.batchMutate 批处理

在 HRegion.doMiniBatchMutation 真正开始工作之前，如果有协处理器钩子，要先执行钩子函数:
注意：预处理是对批处理的数据一次性全部执行完成，而 doMiniBatchMutation()一次只处理批处理的一部分.
这个从 for 循环的次数可以看出来，预处理是从 0~batch.operations.length. 而 doMinitBatchMutation 不是.

```
private void doPreMutationHook(BatchOperationInProgress<Pair<Mutation, Integer>> batchOp) {
    /* Run coprocessor pre hook outside of locks to avoid deadlock */
    WALEdit walEdit = new WALEdit();
    if (coprocessorHost != null) {       // RegionCoprocessorHost，因为服务于HRegion
      for (int i = 0; i < batchOp.operations.length; i++) {
        Pair<Mutation, Integer> nextPair = batchOp.operations[i];
        Mutation m = nextPair.getFirst();
        if (m instanceof Put) {
          if (coprocessorHost.prePut((Put) m, walEdit, m.getWriteToWAL())) {
            // pre hook says skip this Put: mark as success and skip in doMiniBatchMutation
            batchOp.retCodeDetails[i] = OperationStatus.SUCCESS;
          }
        } else if (m instanceof Delete) {
          if (coprocessorHost.preDelete((Delete) m, walEdit, m.getWriteToWAL())) {
            // pre hook says skip this Delete: mark as success and skip in doMiniBatchMutation
            batchOp.retCodeDetails[i] = OperationStatus.SUCCESS;
          }
        } else {    // Mutation还有一个实现类是Append，但是不支持协处理器的
          batchOp.retCodeDetails[i] = new OperationStatus(OperationStatusCode.FAILURE,"Put/Delete supporte batchMutate() now");
        }
        if (!walEdit.isEmpty()) {
          batchOp.walEditsFromCoprocessors[i] = walEdit;   // 如果预处理对WAL添加了额外的逻辑，要设置到batchOp中
          walEdit = new WALEdit();   // 重置
        }
      }
    }
}
```

batchMutate 会循环调用多次 doMimiBatchMutation 完成所有批处理记录的录入，每次范围: [firstIndex, lastIndex )

```
private long doMiniBatchMutation(BatchOperationInProgress<Pair<Mutation, Integer>> batchOp) throws IOException {
    Set<byte[]> putsCfSet = null;              // The set of columnFamilies first seen for Put.
    boolean putsCfSetConsistent = true;        // variable to note if all Put items are for the same CF -- metrics related
    Set<byte[]> deletesCfSet = null;           // The set of columnFamilies first seen for Delete.
    boolean deletesCfSetConsistent = true;// variable to note if all Delete items are for the same CF -- metrics related
    long startTimeMs = EnvironmentEdgeManager.currentTimeMillis();

    WALEdit walEdit = new WALEdit();       // 一次批处理只有一个WALEdit对象(同一个region多行row-key共用一个WAL)
    MultiVersionConsistencyControl.WriteEntry w = null;   // 同时也只有一个WriteEntry,表示在同一次写操作中
    long txid = 0;                         // 事务ID，用于同步WAL
    boolean walSyncSuccessful = false;
    boolean locked = false;

    /** Keep track of the locks we hold so we can release them in finally clause */
    List<Integer> acquiredLocks = Lists.newArrayListWithCapacity(batchOp.operations.length);
    Set<HashedBytes> rowsAlreadyLocked = Sets.newHashSet();
```

```
// reference family maps directly so coprocessors can mutate them if desired
Map<byte[],List<KeyValue>>[] familyMaps = new Map[batchOp.operations.length];
// We try to set up a batch in the range [firstIndex,lastIndexExclusive)
int firstIndex = batchOp.nextIndexToProcess;       // 每次批处理开始是，设置本次的开始为上一次的结束
int lastIndexExclusive = firstIndex;               // 本次批处理的结束位置初始=开始，并通过第一步来逐步递增结束位置
boolean success = false;
int noOfPuts = 0, noOfDeletes = 0;
```

## +step1：获取行锁

```
    // STEP 1. Try to acquire as many locks as we can, and ensure we acquire at least one. 获取尽可能多的锁. 要保证至少获取一个锁
    int numReadyToWrite = 0;     // 准备写的数量
    long now = EnvironmentEdgeManager.currentTimeMillis();
    while (lastIndexExclusive < batchOp.operations.length) {
      Pair<Mutation, Integer> nextPair = batchOp.operations[lastIndexExclusive];
      Mutation mutation = nextPair.getFirst();
      Integer providedLockId = nextPair.getSecond();   // 提供的lockId，如果是客户端显示加锁为RowLock的lockId，否则=null
      Map<byte[], List<KeyValue>> familyMap = mutation.getFamilyMap();
      familyMaps[lastIndexExclusive] = familyMap;         // store the family map reference to allow for mutations

      if (batchOp.retCodeDetails[lastIndexExclusive].getOperationStatusCode() != OperationStatusCode.NOT_RUN) {
        lastIndexExclusive++;
        continue;   // skip anything that "ran" already. 比如Coprocessor的预处理可能让statusCode=SUCCESS
      }
      // If we haven't got any rows in our batch, we should block to get the next one. 第一条记录要加锁，下一个记录会被阻塞
      boolean shouldBlock = numReadyToWrite == 0;
      boolean failedToAcquire = false;
      Integer acquiredLockId = null;
      HashedBytes currentRow = new HashedBytes(mutation.getRow());   // row-key
      try {
        if (providedLockId != null || !rowsAlreadyLocked.contains(currentRow)) {
          acquiredLockId = getLock(providedLockId, currentRow, shouldBlock);
          if (acquiredLockId == null) {   // 非客户端显示加锁, shouldBlock=false时，获取已经被占用的行锁才有可能返回null
            failedToAcquire = true;
          } else if (providedLockId == null) {   // 非客户端显示加锁，一旦获得行锁. 在同一个客户端内对同一行操作都接受
            rowsAlreadyLocked.add(currentRow);
          }
        }
      } catch (IOException ioe) {   failedToAcquire = true;   }
      if (failedToAcquire) break;            // We failed to grab another lock, stop acquiring more rows for this batch
      if (providedLockId == null) {
        acquiredLocks.add(acquiredLockId);
      }
      lastIndexExclusive++;
      numReadyToWrite++;
    }
```

根据给定的 lockid(HRegion 内部使用的 lockid，客户端显示加锁有值，不是客户端显示加锁则为 null)，行键，是否等待获取 lockid. 对于客户端显示加锁，则直接返回 lockid. 不是的话，调用 internalObtainRowLock 获取 lockid.
这个方法和前面 HTable.lockRow()调用的都是 internalObtainRowLock. 参数 waitForLock 表示是否等待获取锁.

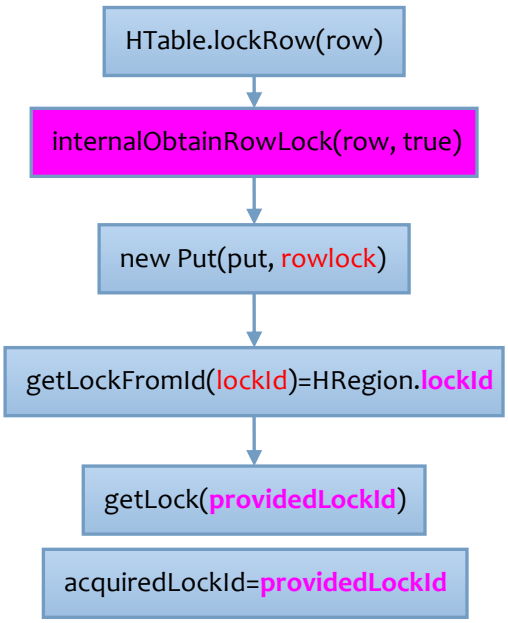
```
/**Returns existing row lock if found, otherwise obtains a new row lock and returns it.
  * @param lockid requested by the user, or null if the user didn't already hold lock  客户端显示加锁对应的HRegion的lockId.
  * @param row the row to lock
  * @param waitForLock if true, will block until the lock is available, otherwise will
  * simply return null if it could not acquire the lock.
  * @return lockid or null if waitForLock is false and the lock was unavailable. */
protected Integer getLock(Integer lockid, HashedBytes row, boolean waitForLock) {
    Integer lid;
    if (lockid == null) {      // 创建Put时，不提供RowLock(客户端不持有锁)，因此lockid=null.
        lid = internalObtainRowLock(row, waitForLock);      // 内部获取一个锁.
    } else {                   // 客户端显示创建RowLock，并传给Put. 注意：参数的lockid是HRegion内部使用的lockId
        HashedBytes rowFromLock = lockIds.get(lockid);      // 加锁时, <lockId, row>保存在lockIds中
        if (!row.equals(rowFromLock)) throw new IOException("Invalid row lock");
        lid = lockid;
    }
    return lid;
}
```

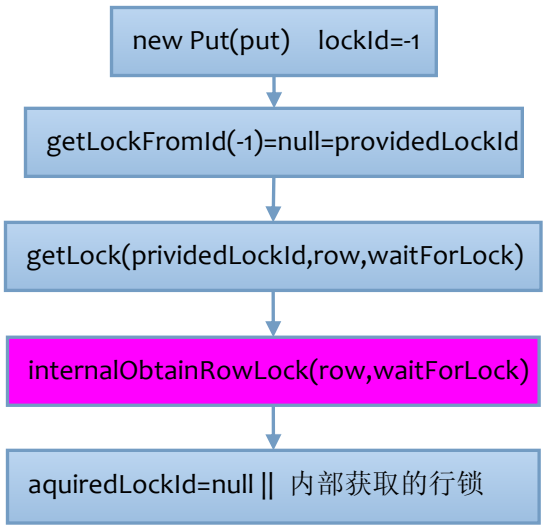
下左图为客户端显示加锁，右图为服务器端隐式加锁.
客户端加锁采用 HTable.lockRow()，不同客户端进程每次加锁都会调用 HTable.lockRow()，然后传递给 Put 对象.
服务器加锁其操作的 Put 没有 RowLock，但是也是来自客户端 HTable. 只不过加锁是在服务器内部完成的.

获取一个行锁的过程都是一样的 internalObtainRowLock:

1. 当某一行被某个进程加锁后，其他进程如果要对同一行进行操作(获得锁)，要么等待要么放弃锁.
2. 对于不同行(row-key)，不同进程都可以顺利获得锁，只要这个时候这一行没有被其他进程加锁.

这里的进程指的既可以是客户端主动加锁，或者服务器隐式加锁. 客户端和服务器加锁在并发下可以任意顺序进行.




**批处理的前提就是：在最先拥有行锁的前提下，可以一次处理多条记录.**
在并发环境下，会有多个客户端，如果多个客户端同时操作同一行，最先获得这一行锁的客户端拥有批处理的权利.

# example(串行与并行批处理)

## 1. 串行批处理

现在我们先来看下不存在并发的情况下，单个客户端操作相同行和不同行的步骤:

同一个客户端，串行方式执行批处理，只发生一次 doMiniBatchMutation: [firstIndex, lastIndexExclusive)

| Time | row-key | ts | rowsAlreadyLocked | numReadyToWrite \|shouldBlock | acquiredLockId \| failedToAcquire | lastIndexExclusive |
|---|---|---|---|---|---|---|
| 1 | row-001 | 1 | N | 0 \| true | != null \| false | 1 |
| 2 | row-001 | 2 | Y | 1 \| - | - | 2 |
| 3 | row-002 | 1 | N | 2 \| false | !=null \| false | 3 |
| 4 | row-003 | 1 | N | 3 \| false | !=null \| false | 4 |

在没有其他客户端竞争同一个行锁的情况下，类似于串行执行，以下是模拟代码(针对的都是非客户端加锁):

另外要注意一点: doMiniBatchMutation 的变量范围是某一次批处理，而 HRegion 的变量是全局的.

比如 doMiniBatchMutation 的 rowsAlreadyLocked 在其他客户端的批处理是不可见的.

但是 HRegion 的全局变量 lockedRows(已经加锁的行)则是针对所有客户端都可见的. 在后面分析并发写时要注意.

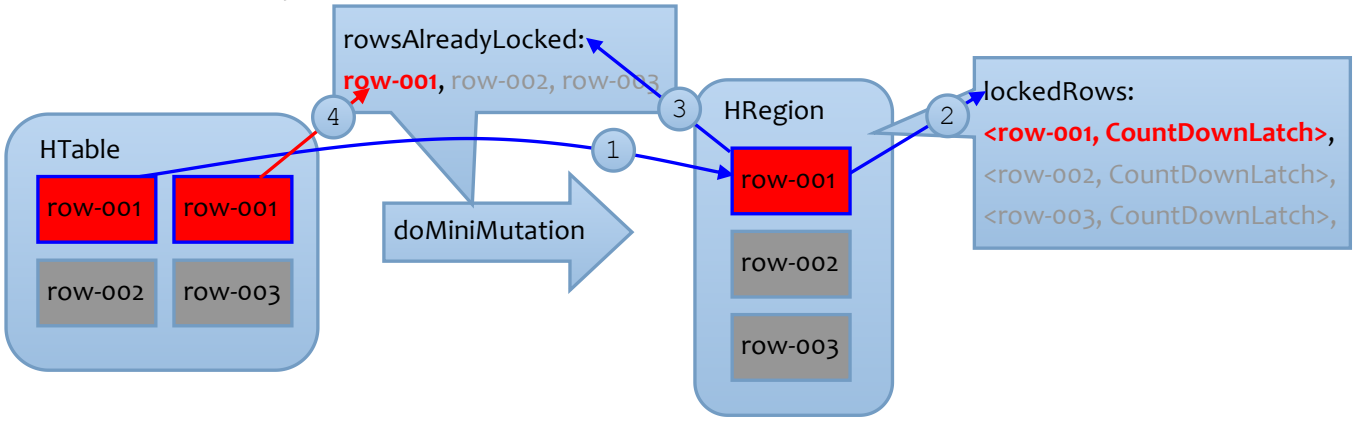
```
Put put1 = new Put(Bytes.toBytes("row-001"));
put1.add(Bytes.toBytes("fam1"), Bytes.toBytes("qualo1"), Bytes.toBytes("value1"));
table.put(put1);
Put put2 = new Put(Bytes.toBytes("row-001"));
put1.add(Bytes.toBytes("fam1"), Bytes.toBytes("qualo1"), Bytes.toBytes("value2"));
table.put(put2);
Put put3 = new Put(Bytes.toBytes("row-002"));
put3.add(Bytes.toBytes("fam1"), Bytes.toBytes("qualo1"), Bytes.toBytes("value1"));
table.put(put3);
Put put4 = new Put(Bytes.toBytes("row-003"));
put4.add(Bytes.toBytes("fam1"), Bytes.toBytes("qualo1"), Bytes.toBytes("value1"));
table.put(put4);
```

T1: 服务器操作 row-001，隐式加锁，(1)因为 HRegion 的 lockedRows 之前没有存在 key=row-001 的记录，所以服务器可以获得 row-001 的行锁.(2)获得锁之后，将当前行键 row-001 加入到集合 rowsAlreadyLocked 中.(3)这个集合变量用于同一个客户端的批处理范围内，如果是针对同一行的操作，则非第一次的操作不会再获取行锁了. 比如接下来的 order2

T2: 因为 rowsAlreadyLocked 已经包括了这一次操作的 row-001 即: rowsAlreadyLocked.contains(currentRow) (4)而且都是在同一个客户端的同一次批处理中. 所以 if 里的语句都不会执行，即已经有行锁了，不再获取. 只要递增 lastIndexExclusive 和 numReadyToWrite 索引变量即可

T3: 服务器隐式加锁,获得 row-002 的行锁. 类似于 order1. 将 row-002 加入到 rowsAlreadyLocked 并递增索引

T4: 同上 order1 和 order3 的操作.




一个客户端，即一个 HTable 实例的批处理示意图. 对于不同 row-key，在 HRegion 都会保存已经被锁住的行.

在 doMiniMutation 中保存的 rowsAlreadyLocked 用于同一行的多次操作. 同一行的第二次操作不会获得锁的.

## 2. 并行批处理

下表描述了三个客户端可能同时操作相同的行(假设都是针对同一个 HRegion).

| ClientA | ClientB | ClientC |
|---|---|---|
| row-001, V1 | row-001, V2 | row-003, V1 |
| row-001, V3 | row-002, V2 | row-003, V2 |
| row-002, V1 | row-002, V3 | row-004, V1 |
| row-003, V3 | row-003, V4 | row-005, V1 |

假设三个客户端按照如下的顺序写:

| Time | Client | row-key | block? | exist? | getLock | rowsAlreadyLocked | lockedRows |
|---|---|---|---|---|---|---|---|
| 1 | A | row-001, V1 | Y | N | Y | N->row-001 | row-001(A) |
| 2 | B | row-001, V2 | Y | Y | wait.. | wait.. | row-001(A) |
| 3 | A | row-001, V3 | -- | | -- | Y->row-001 | row-001(A) |
| 4 | B | ... ... ... | wait.. | | N | wait.. | |
| 5 | A | row-002, V1 | N | N | Y | row-001, N->row-002 | row-001(A), row-002(A) |
| 6 | C | row-003, V1 | Y | N | Y | N->row-003 | row-001, row-002, row-003(C) |
| 7 | C | row-003, V2 | -- | | -- | Y->row-003 | row-001, row-002, row-003(C) |
| 8 | A | row-003, V3 | N | Y | null | | |

**执行过程分析:**

T1: HRegion.lockedRows 中不存在 key=row-001 的行锁. 因此客户端 A 首先获得 row-001 的行锁.(1)(2)(3)

T2: 客户端 A 已经占用了 row-001 行锁. 其他客户端如果也要操作 row-001, 是获取不到 row-001 这一行的行锁的. 由于客户端 B 的批处理记录中, row-001 是批处理的第一条记录, 所以会在此处等待获取 row-001 的行锁.

T3: 还是客户端 A, 由于在这次批处理中的上一次已经为 row-001 加锁(T1)了. 所以在同一客户端同一次批处理中是可以批处理同一行记录的. 主要根据批处理方法中的 rowsAlreadyLocked 集合变量.(4)

T4: 客户端 B 由于在第一行记录, 即要操作 row-001 获得锁时已经处于等待状态, 接下来的记录都被阻塞住了.

T5: 客户端 A 继续处理 row-002, 由于 HRegion.lockedRows 目前为止不存在 key=row-002 的行锁. 处理方式同 T1

T6: A 在处理 row-003 之前, C 抢占处理 row-003. C 的处理方式同 T1, T5, 只不过 row--003 被 C 加锁.(1)(2)(3)

T7: C 在拥有 row-003 行锁的前提下, 在同一个客户端 C 同一次批处理中可以处理同一行记录. 处理方式同 T3 (4)

T8: row-003 的行锁被 C 占用. A 也要操作 row-003, 但是 row-003 不是 A 本次批处理的第一条记录. shouldBlock=false 不同于 B 处理 row-001 时行锁被 A 占用(row-001 是 B 第一行记录), A 会放弃 row-003 的行锁, 本次批处理结束

**难点在于 T2 和 T8:**

T2: B 要操作 row-001, 就需要获取到对应行 row-001 的行锁. 但是 row-001 已经被别的进程: A 锁住了.

由于 B 要处理的 row-001 是客户端 B 此次批处理的第一条记录. 所以 B 会等待获取 row-001 的行锁.

当 A 的一次批处理全部处理完成后,会释放掉占用的 row-001 的行锁, 将占用的行锁 row-001 从 lockedRows 中移除. 而 B 在 row-001 这一行上一直循环等待. 只要 row-001 在 HRegion.lockedRows 中不存在, 就不再处于循环等待的状态, 而是重新把 row-001 放入 HRegion.lockedRows 中. 此时 row-001 这一行的行锁被 B 锁住了.

T8: 轮到 A 要操作 row-003, 也想要获取 row-003 在 HRegion 中的行锁. 但是不幸的是 row-003 已经被 C 锁住了.

由于 A 要处理的 row-003 不是 A 此次批处理的第一条记录, 所以 A 会放弃等待获取 row-003 这一行的行锁.

此时通过 getLock()返回的 acquiredLockId=null, 设置 failedToAcquire=true. 并 break 出 while 循环的 lastIndex...

即 A 本次批处理最多只处理到 row-003 这一条记录(不包括 row-003, 因为 A 要操作的 row-003 已经被其他进程锁住).

通过 break 跳出 while 循环, 限制了 A 本次批处理的 lastIndexExclusive. 即只处理[firstIndex, lastIndexExclusive)条.

A 要操作的 row-003 会被交到 A 的下一次批处理去完成, 而且必须在 C 占用的 row-003 的那次批处理完成后才进行.

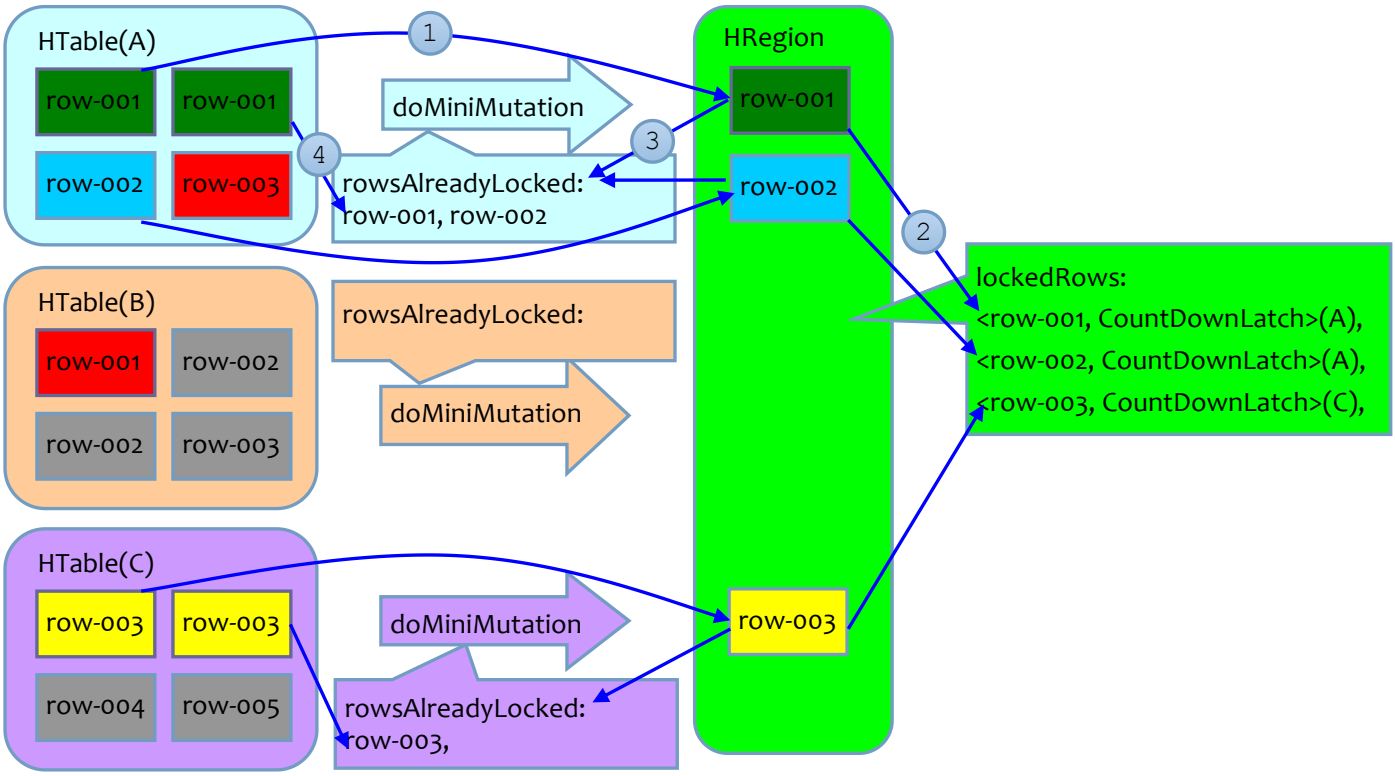
类似于 B 要处理 row-001 要等到被 A 占用的 row-001 的那一次批处理完成, B 获得 row-001 的行锁才可以继续进行.

**问题解答:**

那么问题是: 为什么如果是批处理的第一条记录, 行锁被占用, 则等待. 非第一条记录, 行锁被占用, 则放弃?

1. 如果是批处理的第一条记录, 行锁被占用, 必须等待. 如果不等待, 而是直接放弃. 那么有可能下次你重新开始的时候又获取不到行锁了. 而只有处于等待状态, 才可能在这一行对应的行锁被释放后, 可以立即开始.

2. 如果不是批处理的第一条记录，行锁被占用，直接放弃，这一行的操作被交给了下次批处理去完成.
   如果不放弃，而是继续等待．有可能别人正在等着被你占用的锁,而你又在等待被别人占用的锁,容易造成死锁.
   因为不是批处理的第一条记录，说明你前面已经成功地获取到了至少一行记录的锁了．该开始的赶紧开始!!!

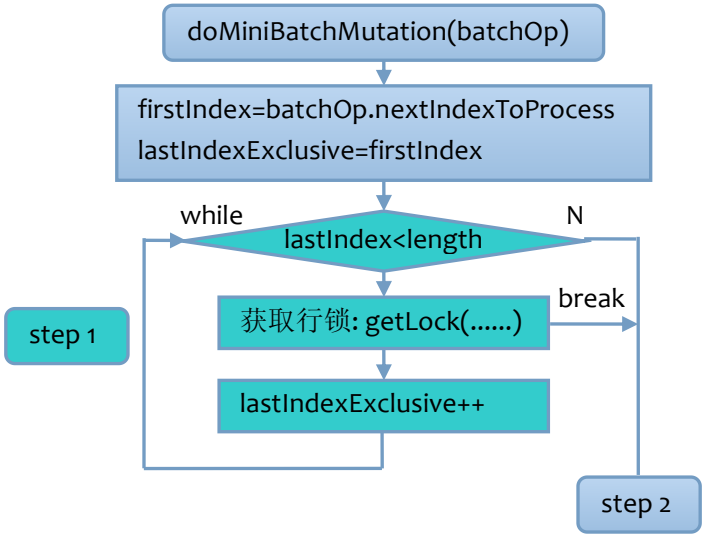



**总结下获取锁的过程:**

1. 获取尽可能多的锁，并且保证至少获得一个锁．在行锁被占用时，如果是批处理第一条记录则等待获取.
2. 对于相同 row-key 的多次修改，如果第一次获取到锁，则接下来的多次都发生在同一次批处理中.
   集合 rowsAlreadyLocked 保存了所有已经被加锁的行，如果已经存在了，则不会获取行锁.
   在处理相同行的多次修改操作(每次操作都是一个 Mutation)，非第一次操作只需要增加索引即可.

**注意:**

1. 在 HRegionServer 端已经对批处理的所有记录按照行键进行了排序．所以到达 HRegion 时行键都是排好序的.
2. 每个客户端自己的批处理的 rowsAlreadyLocked 的可见范围都是自己对象.只是决定自己后续同一行的操作.
   HRegion 的 lockedRows 是所有客户端都可见的．决定了其他客户端是否在这一行上等待，还是放弃行锁.
3. 客户端的全部批处理数据在服务器端一次只执行一部分批处理: [firstIndex, lastIndexExclusive)
   服务器的迷你/部分批处理每次开始时，设置 firstIndex 为上次批处理的结束位置．结束位置=开始位置，
   然后每接受一条记录(获取锁或者同一行已经有所了)就递增 lastIndexExclusive．用于确定本次批处理的结束点.
4. 放弃行锁的那一行被分配到客户端的下次批处理操作是由 HRegion.batchMutate 控制循环的.
   batchOp 只要还没有完成，就会循环多次批处理，直到 batchOp.nextIndexToProcess 达到全部数据的长度.



## +step2: 更新时间戳

```
// we should record the timestamp only after we have acquired the rowLock, 在获取到行锁后记录时间戳
// otherwise, newer puts/deletes are not guaranteed to have a newer timestamp
now = EnvironmentEdgeManager.currentTimeMillis();
byte[] byteNow = Bytes.toBytes(now);
if (numReadyToWrite <= 0) return 0L;       // Nothing to put/delete -- an exception in the above such as NoSuchColumnFamily?
// We've now grabbed as many mutations off the list as we can
// ------------------------------------
// STEP 2. Update any LATEST_TIMESTAMP timestamps
// ----------------------------------
for (int i = firstIndex; i < lastIndexExclusive; i++) {
  if (batchOp.retCodeDetails[i].getOperationStatusCode() != OperationStatusCode.NOT_RUN) continue;   // skip invalid
  Mutation mutation = batchOp.operations[i].getFirst();
  if (mutation instanceof Put) {
    updateKVTimestamps(familyMaps[i].values(), byteNow);
    noOfPuts++;
  } else {
    prepareDeleteTimestamps(familyMaps[i], byteNow);
    noOfDeletes++;
  }
}

lock(this.updatesLock.readLock(), numReadyToWrite);   // 对下面的操作进行加锁.
locked = true;

w = mvcc.beginMemstoreInsert();          // Acquire the latest mvcc number 获取最新的MVCC版本号

if (coprocessorHost != null) {                // calling the pre CP hook for batch mutation 协处理器的预处理
  MiniBatchOperationInProgress<Pair<Mutation, Integer>> miniBatchOp =
    new MiniBatchOperationInProgress<Pair<Mutation, Integer>>(batchOp.operations,
    batchOp.retCodeDetails, batchOp.walEditsFromCoprocessors, firstIndex, lastIndexExclusive);
  if (coprocessorHost.preBatchMutate(miniBatchOp)) return 0L;
}
```

除了更新时间戳，还有两个操作:

1. 准备 MVCC
2. 协处理器的 preBatchMuatte 预处理. 其中 batchOp 的几个变量在 doPreMutationHook 被全部赋值了.
   因为在 doMiniBatchMutation 中只处理一部分，所以 firstIndex 和 lastIndexExclusive 限制了预处理记录的数量.
   MiniBatchOperationInProgress 相比 BatchOperationInProgress 正是多了这两个限制索引.

## MVCC intro

http://blogs.apache.org/hbase/entry/apache_hbase_internals_locking_and

**Writes and Write-Write Synchronization 同步写**

| RowKey | Info:Company | Info:Role |
|---|---|---|
| Greg | Cloudera | Engineer |

| RowKey | Info:Company | Info:Role |
|---|---|---|
| Greg | Restaurant | Waiter |

*Image 1. Two writes to the same row 对同一行的两次写操作*

(1) Write to Write-Ahead-Log (WAL)
(2) Update MemStore: write each data cell [the (row, column) pair] to the memstore
*List 1. Simple list of write steps 写的步骤: 分别写入 WAL 日志文件(HLog) 和 MemStore 内存*

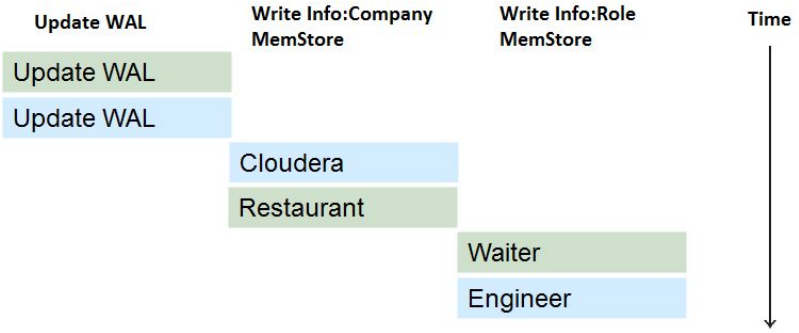



*Image 2. One possible order of events for two writes 并发写的一种可能顺序*

| RowKey | Info:Company | Info:Role |
|---|---|---|
| Greg | Restaurant | Engineer |

*Image 3. Inconsistent result in absence of write-write synchronization 两次写的不一致性(没有同步)*

**解决方案: 同步写--每次写都要先获取锁, 然后在这个锁范围内操作这条记录, 操作完毕才释放锁. 其他线程这个时候只能等到当前锁释放后才能更新同一条记录. 保证了同一行记录的事务要求.**

(0) Obtain Row Lock
(1) Write to Write-Ahead-Log (WAL)
(2) Update MemStore: write each cell to the memstore
(3) Release Row Lock

*List 2: List of write-steps with write-write synchronization 同步写的步骤: 锁的获取和释放*

多次写之间的同步的一种可能的事件序列图参考下图.
针对同一条记录的第二个事件的更新必须等到第一个事件更新完毕才能进行.
注意: 这里的同步都是针对同一条记录(即 HBase 中 row-key 相同的一行记录, 而不是跨行记录!)

写同步保证了写操作的事务, 但是如果在写的同时发生读事件呢?....
PS: 锁和同步: 一个是 lock, 一个是 synchronized. 都是多线程并发环境下的实现方式.

## Read-Write Synchronization 读写同步

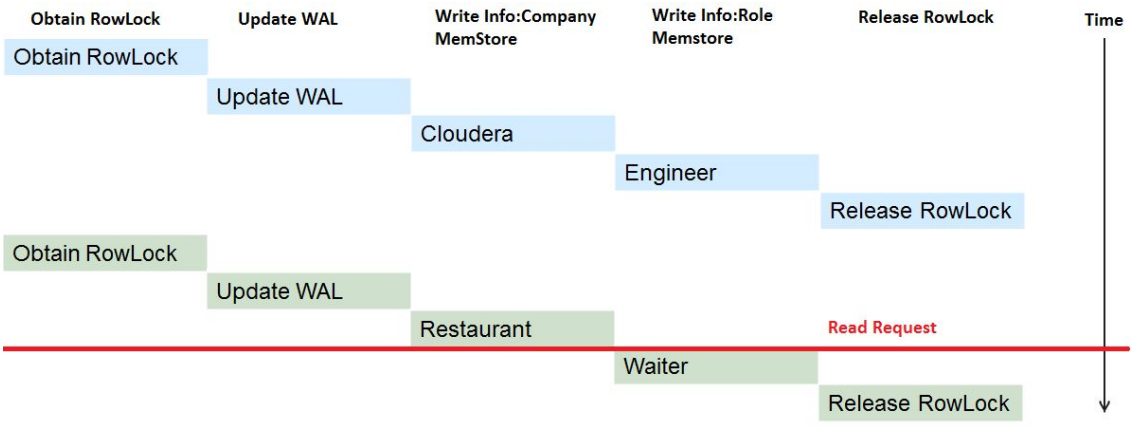



*Image 4. One possible order of operations for two writes and a read 两次写(同步)以及一次读*

| RowKey | Info:Company | Info:Role |
|---|---|---|
| Greg | Restaurant | Engineer |

*Image 5. Inconsistent result in absence of read-write synchronization 没有读写同步引起的不一致性*

解决方案 1：读操作也采取和写同步一样的锁机制，但是这样的代价是读写都要获取锁，造成系统性能下降.
解决方案 2--MVCC:

For writes:
(w1) After acquiring the RowLock, each write operation is immediately assigned a write number
(w2) Each data cell in the write stores its write number.
(w3) A write operation completes by declaring it is finished with the write number.

For reads:
(r1) Each read operation is first assigned a read timestamp, called a read point.
(r2) The read point is assigned to be the highest integer such that all writes with write number <= x have been completed.
(r3) A read r for a certain (row, column) combination returns the data cell with the matching (row, column) whose write number is the largest value that is less than or equal to the read point of r.

*List 3. Multiversion Concurrency Control steps*

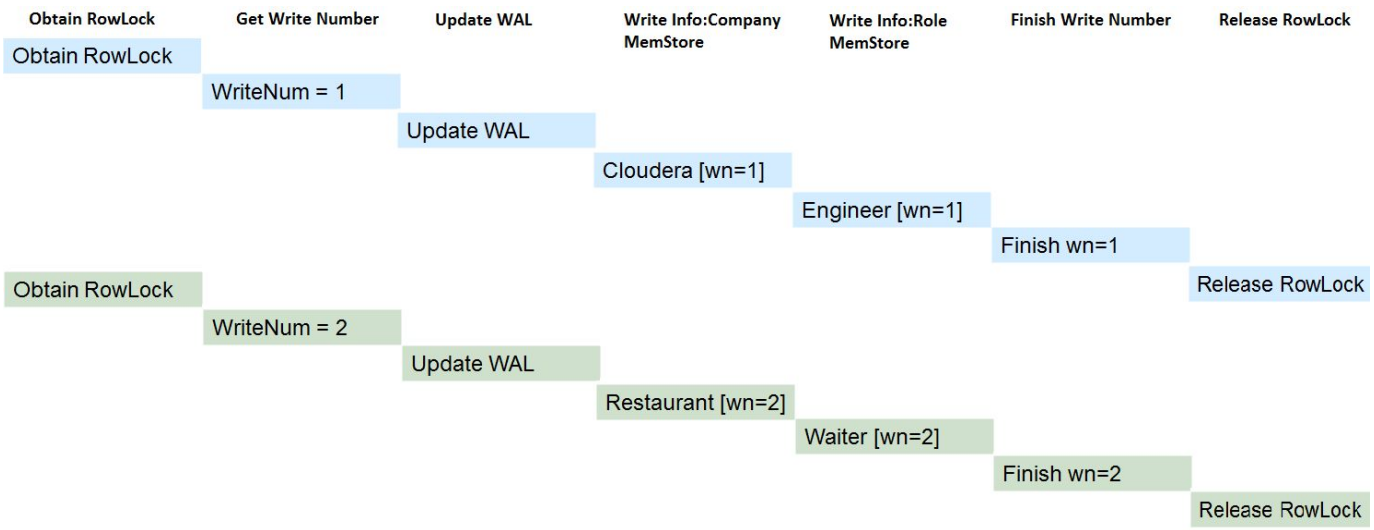



*Image 6. Write steps with Multiversion Concurrency Control 采用 MVCC 的写步骤*

Each write is assigned a write number (step w1), 每次写都被赋值一个 write number，写入内存的每个单元格
each data cell is written to the memstore with its write number (step w2, e.g. "Cloudera [wn=1]") 都携带了这个值
and each write completes by finishing its write number (step w3). 一次写入完成时，结束这个 write number

Now, let's consider the read in Image 4, i.e. a read that begins after step "Restaurant [wn=2]" but before the step "Waiter [wn=2]". From rule r1 and r2, its read point will be assigned to 1. 根据规则 r1,r2, write number=1
From r3, it will read the values with write number of 1, leaving us with: 只读取 write number=1 的记录(=2 的还未提交)

| RowKey | Info:Company | Info:Role |
|---|---|---|
| Greg | Cloudera | Engineer |

*Image 7. Consistent answer with Multiversion Concurrency Control 采用 MVCC 的一致性结果*

A consistent response without requiring locking the row for the reads! 不需要对读进行加锁的一种解决方案!
Let's put this all together by listing the steps for a write with Multiversion Concurrency Control: (new steps required for read-write synchronization are in red):

(0) Obtain Row Lock
(0a) Acquire New Write Number
(1) Write to Write-Ahead-Log (WAL)
(2) Update MemStore: write each cell to the memstore
(2a) Finish Write Number
(3) Release Row Lock

## +step2: HBase MVCC

```
/** Manages the read/write consistency within memstore. 在内存存储中管理读和写的一致性.
 * This provides an interface for readers to determine what entries to ignore, 读取者可以忽略未提交的条目
 * and a mechanism for writers to obtain new write numbers, 写入者获取一个新的write number
 * then "commit" the new writes for readers to read (thus forming atomic transactions).
 * 然后提交新写入的结果，以便于读取者来读取，这样形成了原子事务.
 * 简单说就是：写时获取编号，写后提交这个编号. 读取时只读取已提交的最大编号.
 * 写入完毕提交后，可以通知读取线程，获取到最新的编号对应的结果. */
public class MultiVersionConsistencyControl {      // 多版本一致性控制
  private volatile long memstoreRead = 0;          // 原子变量，内存可读的编号
  private volatile long memstoreWrite = 0;         // 原子变量，内存可写的编号
  private final Object readWaiters = new Object();   // 锁
  private final LinkedList<WriteEntry> writeQueue = new LinkedList<WriteEntry>();    // This is the pending queue of writes.

  public MultiVersionConsistencyControl() {        /** Default constructor. Initializes the memstoreRead/Write points to 0. */
    this.memstoreRead = this.memstoreWrite = 0;
  }
  public void initialize(long startPoint) {            /**Initializes the memstoreRead/Write points appropriately. */
    synchronized (writeQueue) {
      this.memstoreRead = this.memstoreWrite = startPoint;
    }
  }

  public WriteEntry beginMemstoreInsert() {
    synchronized (writeQueue) {
      long nextWriteNumber = ++memstoreWrite;
      WriteEntry e = new WriteEntry(nextWriteNumber);
      writeQueue.add(e);
      return e;
    }
  }
}
```

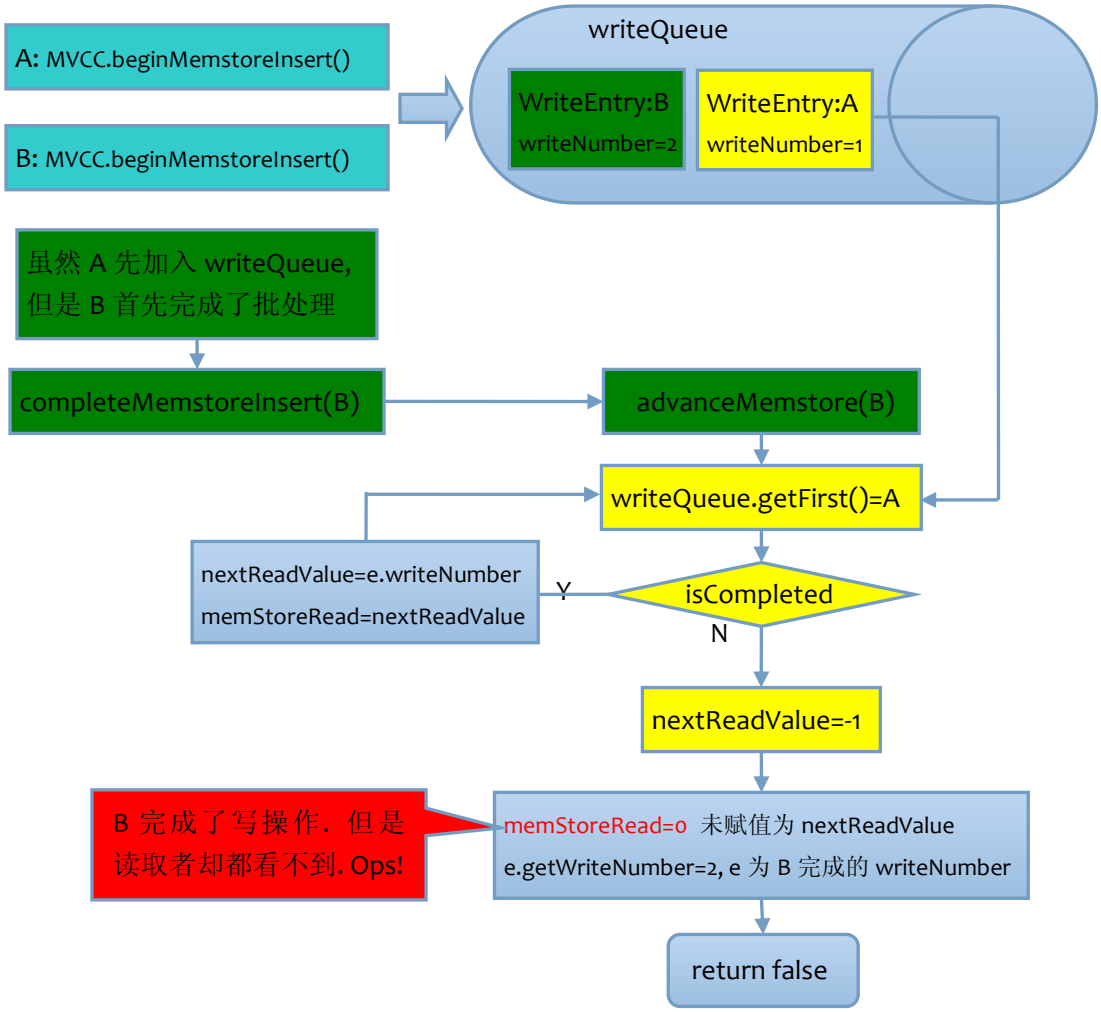
WriteEntry 两个属性 writeNumber 和标记 completed. 创建 WriteEntry 对象时传入 writeNumber.
当一次写完成提交后设置 completed=true. 此时读取线程就可以获取到本次写入的最新结果.

```
  public static class WriteEntry {
    private long writeNumber;
    private boolean completed = false;
  }
```

每一次写都会产生一个新的 WriteEntry 对象，并加入到 writeQueue 队列中.
注意：如果在一次批处理中，处理不同行，则每一行的 WriteEntry 的 writeNumber 都是相同的.

## 异常情况

在批处理第一步时对行锁的获取进行了限制，是后面 MVCC 获取 write number 的保证.
前面示例中客户端 A 获取了行锁 row-001，那么客户端 B 就得等待 A 批处理完成才可以获取到 row-001的行锁.
如果没有行锁的等待机制，假设 B 也获取到了 row-001的行锁. 那么两者都会对 WriteEntry 的 writeNumber+1.
假设 A 首先加入 writeQueue，则获取到的 writeNumber=1, B 获取到的 writeNumber=2. 即 A 是队列的 first 元素.
但是两个客户端在服务器端的运行，可能 B 先完成，而队列的首元素 A 还没有完成. 这样 B 的结果会被无视了.
因为通过 writeQueue.getFirst()得到的是 A 的 WriteEntry，而 A 还没完成, break 出循环, nextReadValue 并未赋值.




模拟没有锁机制等待引起的数据写入丢失

## CompleteMemstoreInsert

```
    // STEP 8. Advance mvcc. This will make this put visible to scanners and getters.
    if (w != null) {
      mvcc.completeMemstoreInsert(w);
      w = null;
    }
```

完成 MemStore 的写入后，重置 WriteEntry w 对象为 null. 以备下次使用.

```
public void completeMemstoreInsert(WriteEntry e) {
  advanceMemstore(e);
  waitForRead(e);
}
boolean advanceMemstore(WriteEntry e) {
  synchronized (writeQueue) {
```

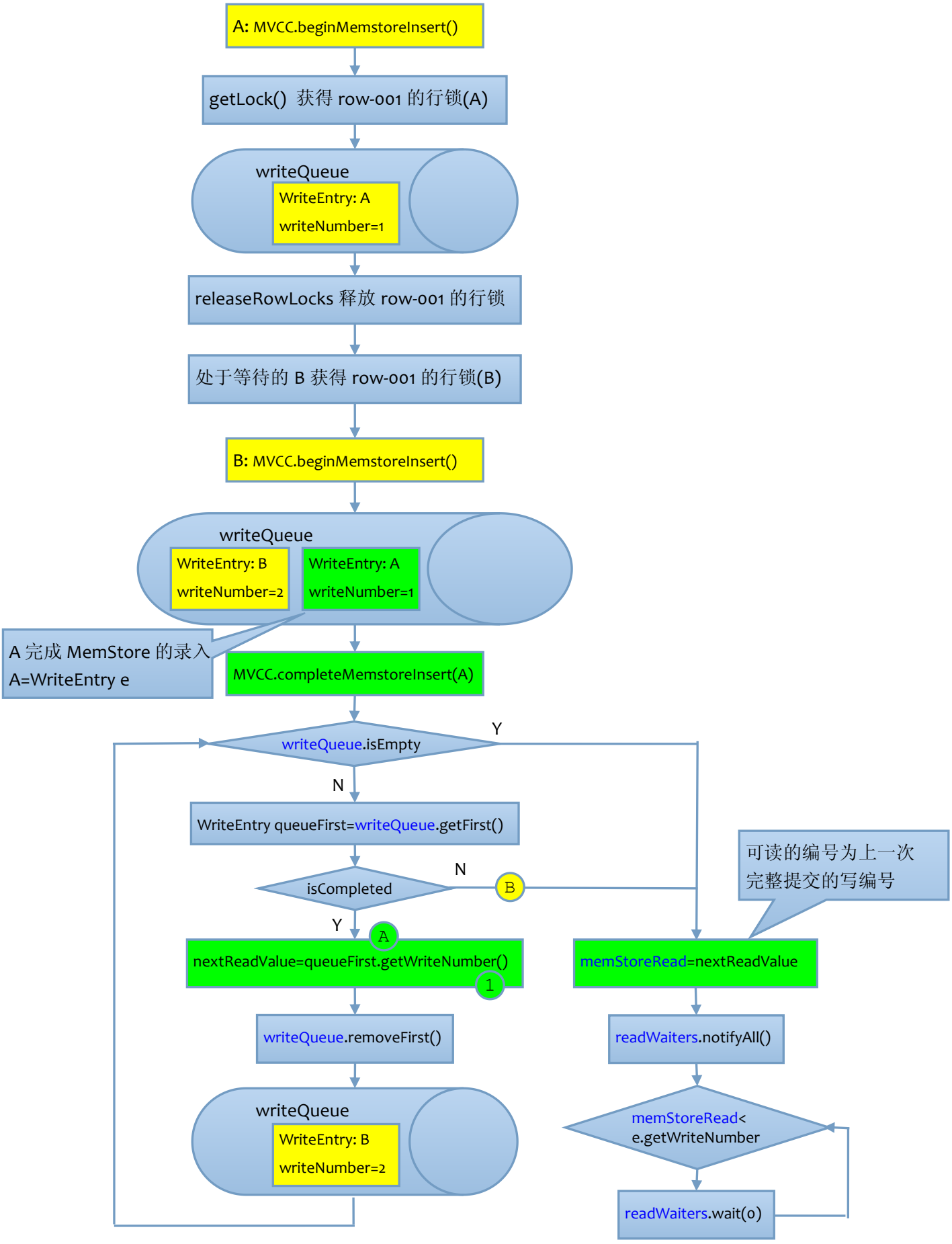
```
        e.markCompleted();

        long nextReadValue = -1;
        boolean ranOnce=false;
        while (!writeQueue.isEmpty()) {
          ranOnce=true;
          WriteEntry queueFirst = writeQueue.getFirst();

          if (nextReadValue > 0) {
            if (nextReadValue+1 != queueFirst.getWriteNumber())
              throw new RuntimeException("invariant in completeMemstoreInsert violated");
          }
          if (queueFirst.isCompleted()) {
            nextReadValue = queueFirst.getWriteNumber();
            writeQueue.removeFirst();
          } else {
            break;
          }
        }
        if (!ranOnce) throw new RuntimeException("never was a first");    // 队列writeQueue中没有元素，却调用了完成的方法. NO!
        if (nextReadValue > 0) {
          synchronized (readWaiters) {
            memstoreRead = nextReadValue;
            readWaiters.notifyAll();
          }
        }
        if (memstoreRead >= e.getWriteNumber()) return true;
        return false;
      }
    }

    /**Wait for the global readPoint to advance upto the specified transaction number. */
    public void waitForRead(WriteEntry e) {
      boolean interrupted = false;
      synchronized (readWaiters) {
        while (memstoreRead < e.getWriteNumber()) {
          readWaiters.wait(0);
        }
      }
    }
```

完成MVCC，是在写入MemStore, WAL，释放批处理占用的锁之后进行的(A占用了row-001的行锁). (A)一旦释放锁，其他进程(B)就可以继续写入(处于等待row-001行锁的B检测到HRegion.lockedRows中没有row-001对应的闭锁CountDownLatch．因此B就成功获得了row-001的行锁).

在A的批处理还没调用MVCC.completeMemstoreInsert时，B获得锁后调用了MVCC.beginMemstoreInsert将属于B的WriteEntry加入到MVCC的writeQueue中．这样当调用A的completeMemstoreInsert时，writeQueue有两个元素．已完成WriteEntry的A,位于writeQueue的队列首部．还没完成WriteEntry的B紧接着加入到writeQueue中．

调用completeMemstoreInsert的处理流程是：从writeQueue中取出第一个元素(A)，如果这个元素已经完成了(A).

设置临时变量nextReadValue=调用当前方法的WriteEntry的writeNumber(即A的writeNumber=1). 然后移除元素.
继续循环writeQueue, 取出writeQueue中剩余的队首元素(B), 如果WriteEntry没有完成(B), 则break跳出循环.
然后设置 MVCC 的 memStoreRead 为先前赋值过的临时变量 nextReadValue. 并通知 readWaiters 可以读取最新值.



## +step3: MemStore

```
        // STEP 3. Write back to memstore
        // Write to memstore. It is ok to write to memstore first without updating the HLog
        // because we do not roll forward the memstore MVCC.    The MVCC will be moved up when the complete operation is done.
        // These changes are not yet visible to scanners till we update the MVCC. The MVCC is moved only when the sync is complete.
        long addedSize = 0;
        for (int i = firstIndex; i < lastIndexExclusive; i++) {
          if (batchOp.retCodeDetails[i].getOperationStatusCode() != OperationStatusCode.NOT_RUN) continue;
          addedSize += applyFamilyMapToMemstore(familyMaps[i], w);
        }
```

在获取锁之后的批处理中，首先调用了 MVCC.beginMemstoreInsert 获取到一个最新 writeNumber 的 WriteEntry. 在接下来将批处理记录写入 MemStore 和 WAL 中都要携带这个 WriteEntry，主要是携带其中的 writeNumber.

```
  /**Atomically apply the given map of family->edits to the memstore.  将更新(编辑)写入内存中  */
  private long applyFamilyMapToMemstore(Map<byte[], List<KeyValue>> familyMap,
      MultiVersionConsistencyControl.WriteEntry localizedWriteEntry) {
    long size = 0;
    boolean freemvcc = false;
    try {
      if (localizedWriteEntry == null) {        //  如果参数为空，则在本方法内部创建一个MVCC事务
        localizedWriteEntry = mvcc.beginMemstoreInsert();
        freemvcc = true;
      }
      for (Map.Entry<byte[], List<KeyValue>> e : familyMap.entrySet()) {
        byte[] family = e.getKey();
        List<KeyValue> edits = e.getValue();

        Store store = getStore(family);      //  根据familyName获取在HRegion上的Store. Store包括MemStore和StoreFile.
        for (KeyValue kv: edits) {
          kv.setMemstoreTS(localizedWriteEntry.getWriteNumber());    //  设置KeyValue的MemStoreTS(也可以看做是版本号)
          size += store.add(kv);                //  往Store的MemStore对象中添加每一个KeyValue
        }
      }
    } finally {
      if (freemvcc) mvcc.completeMemstoreInsert(localizedWriteEntry);
    }
    return size;
  }
```

Store 中将 KeyValue 加入到内存中:

```
  protected long add(final KeyValue kv) {    /**Adds a value to the memstore */
    lock.readLock().lock();
    try {
      return this.memstore.add(kv);
    } finally {
      lock.readLock().unlock();
    }
  }
```

## +step4-5: WAL

写入 MemStoren(内存存储)中的数据如果在 HRegionServer 挂掉之后，内存里的数据会全部丢失．使用预写日志 WAL(预先写入日志，在之前的 HBase 版本中先写 WAL，再写 MemStore，不过新版本调换了顺序，但并不影响，实际上从 WAL 的字面来看，预写就是预先写入，发生在真正写入内存之前)．日志会存储在 HDFS 中不会丢失以便恢复．

```
      // STEP 4. Build WAL edit   构建WALEdit，即将KeyValue对写入WALEdit中
      Durability durability = Durability.USE_DEFAULT;          // 持久性，是否写入WAL日志中
      for (int i = firstIndex; i < lastIndexExclusive; i++) {
        if (batchOp.retCodeDetails[i].getOperationStatusCode() != OperationStatusCode.NOT_RUN) continue;
        batchOp.retCodeDetails[i] = OperationStatus.SUCCESS;

        Mutation m = batchOp.operations[i].getFirst();
        Durability tmpDur = m.getDurability();
        if (tmpDur.ordinal() > durability.ordinal()) durability = tmpDur;
        if (tmpDur == Durability.SKIP_WAL) {
          if (m instanceof Put) {
            recordPutWithoutWal(m.getFamilyMap());
          }
          continue;
        }
        // Add WAL edits by CP. 在batchMutate()的第一次批处理中，如果存在协处理器，会为batchOp每个元素创建WALEdit
        WALEdit fromCP = batchOp.walEditsFromCoprocessors[i];
        if (fromCP != null) {
          for (KeyValue kv : fromCP.getKeyValues()) {
            walEdit.add(kv);
          }
        }
        addFamilyMapToWALEdit(familyMaps[i], walEdit);
      }

      // STEP 5. Append the edit to WAL. Do not sync wal.
      Mutation first = batchOp.operations[firstIndex].getFirst();
      walEdit.addClusterIds(first.getClusterIds());
      txid = this.log.appendNoSync(regionInfo, this.htableDescriptor.getName(),
                 walEdit, first.getClusterId(), now, this.htableDescriptor);
```

针对 HRegion 的每次批处理，都生成一个 WALEdit 对象．并将本次批处理的更新(KeyValues)加入到 WALEdit 中．一次批处理可能包括多行记录(不同的 row-key)，但是只使用了一个 WALEdit 对象．

```
/**Append the given map of family->edits to a WALEdit data structure. This does not write to the HLog itself.
  * @param familyMap map of family->edits(即List<KeyValue>)
  * @param walEdit the destination entry to append into */
private void addFamilyMapToWALEdit(Map<byte[], List<KeyValue>> familyMap, WALEdit walEdit) {
    for (List<KeyValue> edits : familyMap.values()) {
      for (KeyValue kv : edits) {
        walEdit.add(kv);
      }
    }
}
```

# WALEdit+HLog

WALEdit: Used in HBase's transaction log (WAL) to represent the collection of edits (KeyValue objects) corresponding to a single transaction. The class implements "Writable" interface for serializing/deserializing a set of KeyValue items.

Previously, if a transaction contains 3 edits to c1, c2, c3 for a row R, the HLog would have three log entries as follows:
在旧版本中，一个事务对同一行 R 包含了三次更新, HLog 会记录三条日志:

<logseq1-for-edit1>:<KeyValue-for-edit-c1>
<logseq2-for-edit2>:<KeyValue-for-edit-c2>
<logseq3-for-edit3>:<KeyValue-for-edit-c3>

This presents problems because row level atomicity of transactions was not guaranteed. 行级别的事务没有被保证
If we crash after few of the above appends make it, then recovery will restore a partial transaction.只能恢复部分事务

In the new world, all the edits for a given transaction are written out as a single record, for example:
在新版本中，在一次事务中，对同一条记录(同一个 row-key)的多次更新只被记录为一条日志:

<logseq#-for-entire-txn>:<WALEdit-for-entire-txn>

where, the WALEdit is serialized as: WALEdit 会被序列化为<-1，更新数量，每次更新的 KeyValue>.

<-1, # of edits, <KeyValue>, <KeyValue>, ... > 注:-1 只是为了适配旧版本

For example: <-1, 3, <Keyvalue-for-edit-c1>, <KeyValue-for-edit-c2>, <KeyValue-for-edit-c3>>

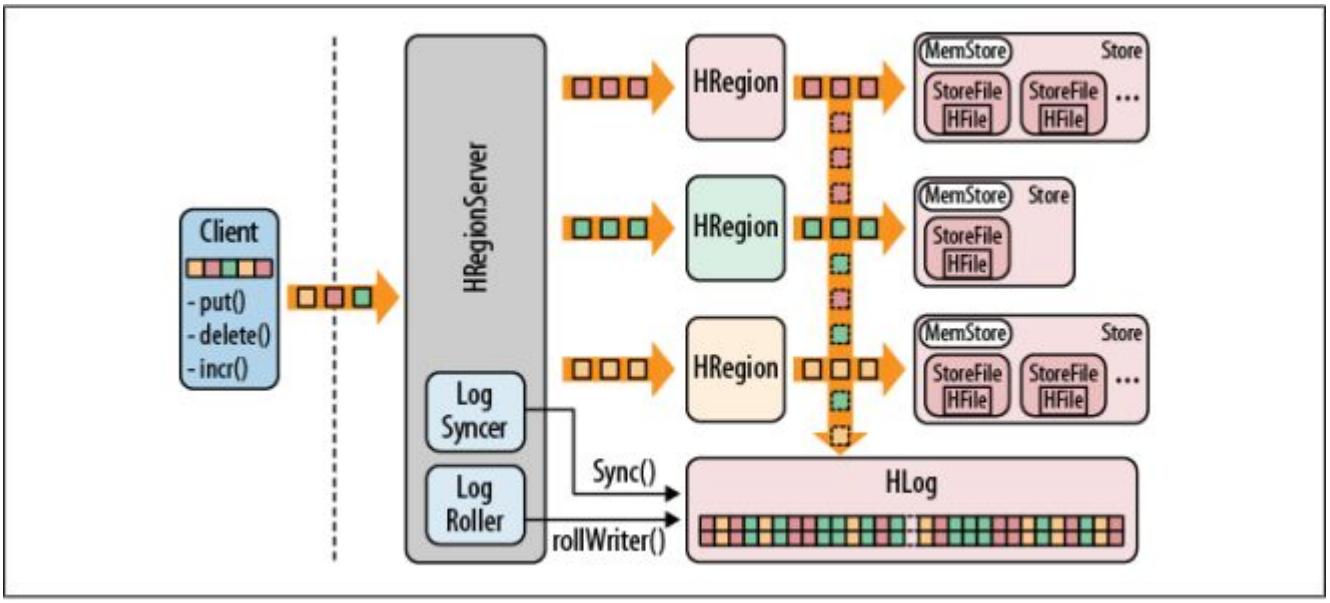



*Figure 8-8. All modifications saved to the WAL, and then passed on to the memstores*

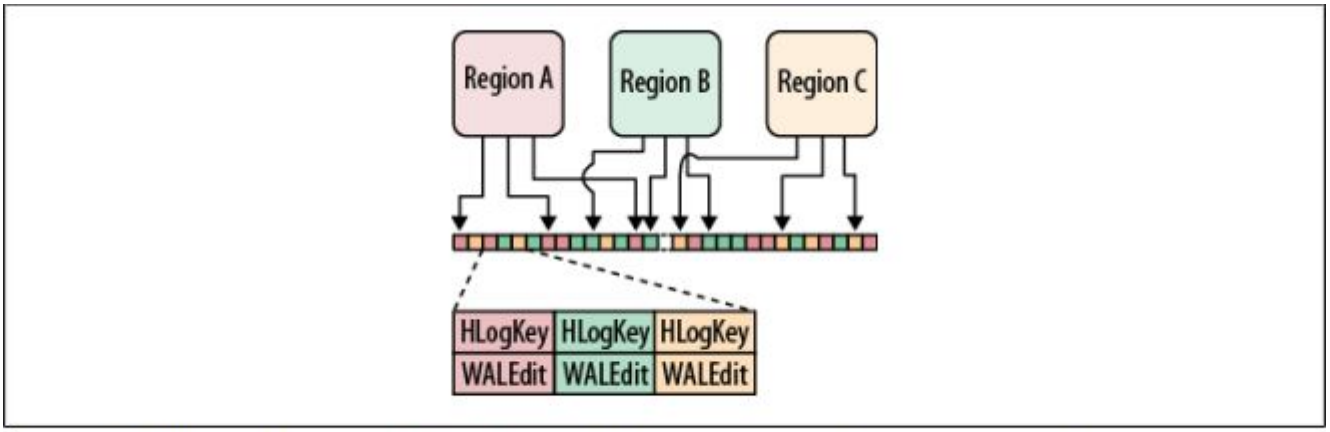
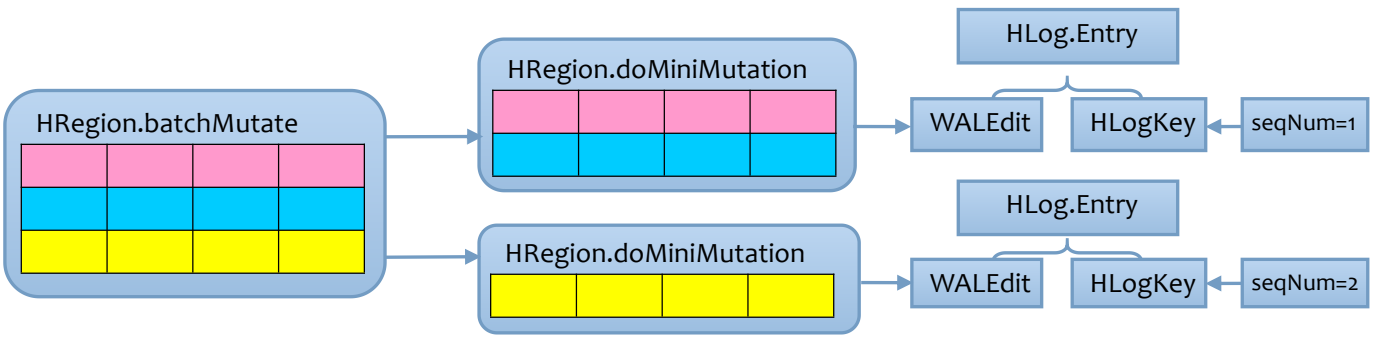



*Figure 8-9. The WAL saving edits in the order they arrive, spanning all regions of the same server*

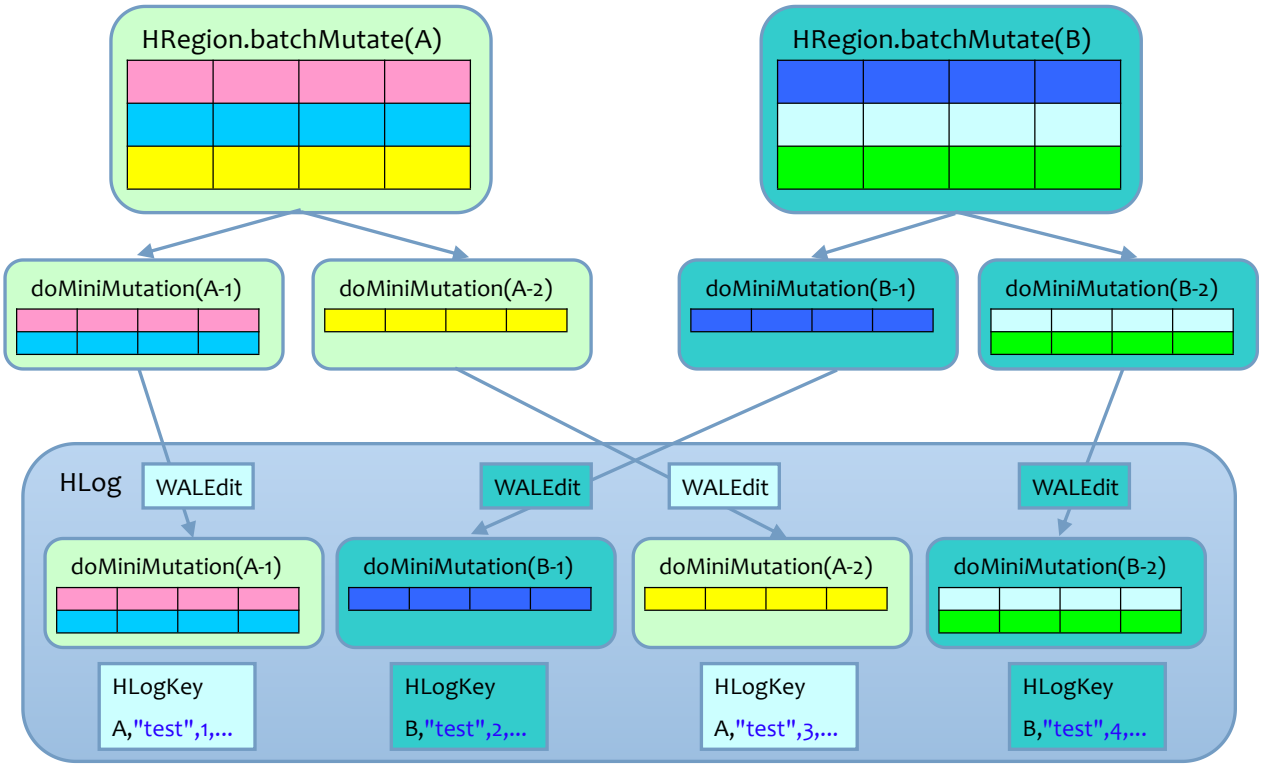
# HLog.append()

```
// Map of encoded region names to their most recent sequence/edit id in their memstore.
private final ConcurrentSkipListMap<byte [], Long> lastSeqWritten = new ConcurrentSkipListMap(Bytes.BYTES_COMPARATOR);
private final AtomicLong logSeqNum = new AtomicLong(0);
private final AtomicInteger numEntries = new AtomicInteger(0);      // number of transactions in the current Hlog  当前HLog的事务数
private final AtomicLong unflushedEntries = new AtomicLong(0);      //  当前HLog中未刷新的事务数，一次批处理产生一次事务

public long appendNoSync(HRegionInfo info, byte [] tableName, WALEdit edits, UUID clusterId, final long now, HTableDescriptor htd) {
   return append(info, tableName, edits, clusterId, now, htd, false);
}
private long append(HRegionInfo info, byte [] tableName, WALEdit edits, UUID clusterId,final long now, HTableDescriptor htd, boolean doSync) {
      if (edits.isEmpty()) return this.unflushedEntries.get();;
      long txid = 0;
      synchronized (this.updateLock) {
         long seqNum = this.logSeqNum.incrementAndGet();
         // The 'lastSeqWritten' map holds the sequence number of the oldest write for each region
         // (i.e. the first edit added to the particular memstore)
         // When the cache is flushed, the entry for the region being flushed is removed  当缓存被刷新(MemStore刷新到磁盘上),
         // if the sequence number of the flush is greater than or equal to the value in lastSeqWritten.
         byte [] encodedRegionName = info.getEncodedNameAsBytes();
         this.lastSeqWritten.putIfAbsent(encodedRegionName, seqNum);
         HLogKey logKey = makeKey(encodedRegionName, tableName, seqNum, now, clusterId);
         doWrite(info, logKey, edits, htd);
         this.numEntries.incrementAndGet();
         txid = this.unflushedEntries.incrementAndGet();            //  未刷新的条目的事务ID
         if (htd.isDeferredLogFlush()) lastDeferredTxid = txid;      //  延迟日志刷写
      }
      // Sync if catalog region, and if not then check if that table supports deferred log flushing
      if (doSync && (info.isMetaRegion() || !htd.isDeferredLogFlush())) {
         this.sync(txid);     // sync txn to file system
      }
      return txid;
}
```

客户端过来的一次完整的批处理可能会被分成多次只处理其中一部分的 mini 批处理.
HRegion.batchMutate(1)  ---> (*)doMiniMutation(1)  ---> (1)WALEdit(1)  ---> (1)HLogKey + (1)seqNum
HRegion 接受的每次 miniMutation 都会生成一个 WALEdit. 然后通过同一个 RegionServer 所有 Region 共同的 HLog 实例调用 append()将每次批处理的日志信息追加到 HLog 日志文件中.

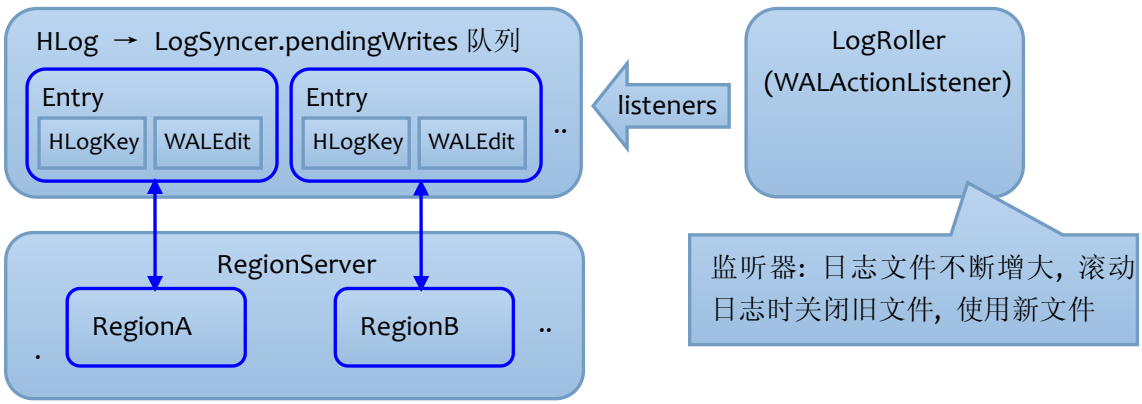


HLog 文件按照数据到达的顺序将日志写入. 而且 HLog 是 RegionServer 上所有 Region 公共的日志文件.
下图示例了两个 Region(A 和 B)按照一定的顺序(A-1, B-1, A-2, B-2)将多次批处理记录(A-1, A-2, B-1, B-2)写入到 HLog 中:




在 doMiniMutation 追加日志到 HLog 中, 每次批处理都生成一个唯一的 seqNum 和 HLogKey.
HLogKey 由 RegionName, tableName, seqNum 序列号, 当前时间, 集群 ID 组成.
HLogKey 和 WALEdit 构成 HLog 的一个 Entry. 即 HLog 中的每一条记录条目, 可以看做是 HLog 的一个元素.
写入时把所有 Region 日志写入 HLog, 拆分日志时(RegionServer 崩溃需要把 HLog 分发到其他 RegionServer 上),
根据 HLogKey 可以找到对应 Region 的所有更新记录即 WALEdit. 因为 HLogKey 的组成包括了 regionName.

```
protected void doWrite(HRegionInfo info, HLogKey logKey, WALEdit logEdit, HTableDescriptor htd){
    if (!this.enabled) return;
    if (!this.listeners.isEmpty()) {
        for (WALActionsListener i: this.listeners)
            i.visitLogEntryBeforeWrite(htd, logKey, logEdit);
    }
    if (!coprocessorHost.preWALWrite(info, logKey, logEdit)) {    // coprocessor hook:
        logSyncer.append(new HLog.Entry(logKey, logEdit));        // write to our buffer for the Hlog file. 先写入缓存中!
    }
    coprocessorHost.postWALWrite(info, logKey, logEdit);
}
```



# HRegionServer->HLog

因为一个 HRegionServer 对应一个 HLog. HRegionServer 初始化的时候会创建 HLog 文件用于写入 WAL 日志.

```
protected volatile HLog hlog;        // HLog, 数据追加到HLog是先存储到内部类LogSyncer的队列中
LogRoller hlogRoller;                // HLog roller.
private ReplicationSourceService replicationSourceHandler;      // Replication services. If no replication, this handler will be null.

/* Run init. Sets up hlog and starts up all server threads. */
protected void handleReportForDutyResponse(final MapWritable c){
      this.hlog = setupWALAndReplication();
}

/**Setup WAL log and replication if enabled. return A WAL instance.
 * Replication setup is done in here because it wants to be hooked up to WAL. */
private HLog setupWALAndReplication() throws IOException {
   final Path oldLogDir = new Path(rootDir, HConstants.HREGION_OLDLOGDIR_NAME);    // /hbase/.oldlogs
   Path logdir = new Path(rootDir, HLog.getHLogDirectoryName(this.serverNameFromMasterPOV.toString())); // /hbase/$regionName..
   if (this.fs.exists(logdir)) throw new RegionServerRunningException("Region server has already created directory");
   // Instantiate replication manager if replication enabled.    Pass it the log directories.  实例化副本管理器.
   createNewReplicationInstance(conf, this, this.fs, logdir, oldLogDir);
   return instantiateHLog(logdir, oldLogDir);
}
protected HLog instantiateHLog(Path logdir, Path oldLogDir){ // creating WAL instance  创建WAL实例, HLog是WAL的实现类
   return new HLog(this.fs.getBackingFs(), logdir, oldLogDir, this.conf,
      getWALActionListeners(), this.serverNameFromMasterPOV.toString());
}
/** setting up WAL instance. Add any WALActionsListeners you want inserted before WAL startup. */
protected List<WALActionsListener> getWALActionListeners() {
   List<WALActionsListener> listeners = new ArrayList<WALActionsListener>();
   this.hlogRoller = new LogRoller(this, this);    // Log roller  日志滚动
   listeners.add(this.hlogRoller);
   if (this.replicationSourceHandler != null && this.replicationSourceHandler.getWALActionsListener() != null) {
      // Replication handler is an implementation of WALActionsListener.
      listeners.add(this.replicationSourceHandler.getWALActionsListener());
   }
   return listeners;
}
```

HRegionServer 上的每个 HRegion 都会接收这个共用的 HLog 实例. HRegionServer 打开 HRegion 时会传入 HLog.

```
public static HRegion newHRegion(Path tableDir, HLog log, FileSystem fs, Configuration conf,
      HRegionInfo regionInfo, final HTableDescriptor htd, RegionServerServices rsServices) {
      Class<? extends HRegion> regionClass = (Class<? extends HRegion>) conf.getClass(HConstants.REGION_IMPL, HRegion.class);
      Constructor<? extends HRegion> c = regionClass.getConstructor(Path.class, HLog.class, FileSystem.class,
               Configuration.class, HRegionInfo.class, HTableDescriptor.class, RegionServerServices.class);
      return c.newInstance(tableDir, log, fs, conf, regionInfo, htd, rsServices);
}
```

LogSyncer 在 HLog 中，其生命周期和 HLog 一样，如果没有日志追加到 HLog，就不需要进行 LogSyncer 同步操作.
LogRoller 在 HRegionServer 中，并且在 startServiceThreads 中启动. 不像 LogSyncer 时时同步，而是在需要时刷新.

# HLog rollWriter

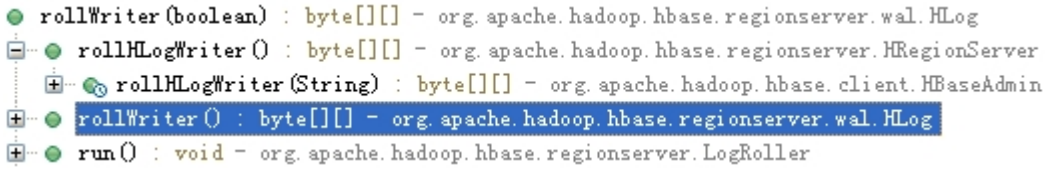
创建 HLog 对象，就立即启动日志写入器滚动，即创建日志文件和对应的 Writer 用于准备接受用户操作的日志.

```
Writer writer;                    // Current log file. 封装输出流的Writer，数据append到Writer中即可完成数据到文件的持久化
private FSDataOutputStream hdfs_out;   // FSDataOutputStream associated with the current SequenceFile.writer 输出流
private final long logrollsize;   // If > than this size, roll the log. This is typically 0.95 times the size of the default Hdfs block size.
// This lock prevents starting a log roll during a cache flush. synchronized is insufficient because a cache flush spans two method calls.
private final Lock cacheFlushLock = new ReentrantLock();
// We synchronize on updateLock to prevent updates and to prevent a log roll during an update locked during appends
private final Object updateLock = new Object();       // 追加时对updateLock进行同步，防止这个时候进行日志滚动
private final Object flushLock = new Object();        // 日志刷新到磁盘中
private volatile boolean logRollRunning;              // 是否正在进行日志滚动
// Map of all log files but the current one. 除了当前文件的所有日志文件
final SortedMap<Long, Path> outputfiles = Collections.synchronizedSortedMap(new TreeMap<Long, Path>());

public HLog(final FileSystem fs, final Path dir, final Path oldLogDir, final Configuration conf,
      final List<WALActionsListener> listeners, final boolean failIfLogDirExists, final String prefix, boolean forMeta){
   this.blocksize = conf.getLong("hbase.regionserver.hlog.blocksize", FSUtils.getDefaultBlockSize(this.fs, this.dir));
   float multi = conf.getFloat("hbase.regionserver.logroll.multiplier", 0.95f);       // Roll at 95% of block size.
   this.logrollsize = (long)(this.blocksize * multi);                                 // 达到64MB*0.95，就可以进行日志滚动
   this.optionalFlushInterval = conf.getLong("hbase.regionserver.optionallogflushinterval", 1 * 1000);   // 日志刷新间隔(1s)
   this.hlogFs = new HLogFileSystem(conf);
   if (listeners != null)
      for (WALActionsListener i: listeners)
         registerWALActionsListener(i);                  // 注册WAL动作监听器，有LogRoller和Replication
   HBaseFileSystem.makeDirOnFileSystem(fs, dir);         // 创建以./logs/regionServer, port, startCode 的目录
   HBaseFileSystem.makeDirOnFileSystem(fs, oldLogDir);   // 创建./oldlogs 目录，用于存放旧的日志文件，会被移除
   rollWriter();                                         // rollWriter sets this.hdfs_out if it can. 日志写入器滚动.
   logSyncer = new LogSyncer(this.optionalFlushInterval); // 日志同步器，参数表示同步(刷新到磁盘)时间间隔
   Threads.setDaemonThreadRunning(logSyncer.getThread(), Thread.currentThread().getName()+".logSyncer"); // 启动线程
   coprocessorHost = new WALCoprocessorHost(this, conf);
}
```

rollWriter()获取滚动日志需要的 Writer,该方法会新建一个新的日志文件，这样追加的日志数据会写入新的文件.

三种情况会触发该动作:

1. HBaseAdmin 客户端触发
2. 新创建 HLog 时：作用是初始化 HLog，新建第一个日志文件.
3. LogRoller 日志滚动线程触发时：已经存在旧的日志文件，新建新的日志文件，必要的话清理旧文件.



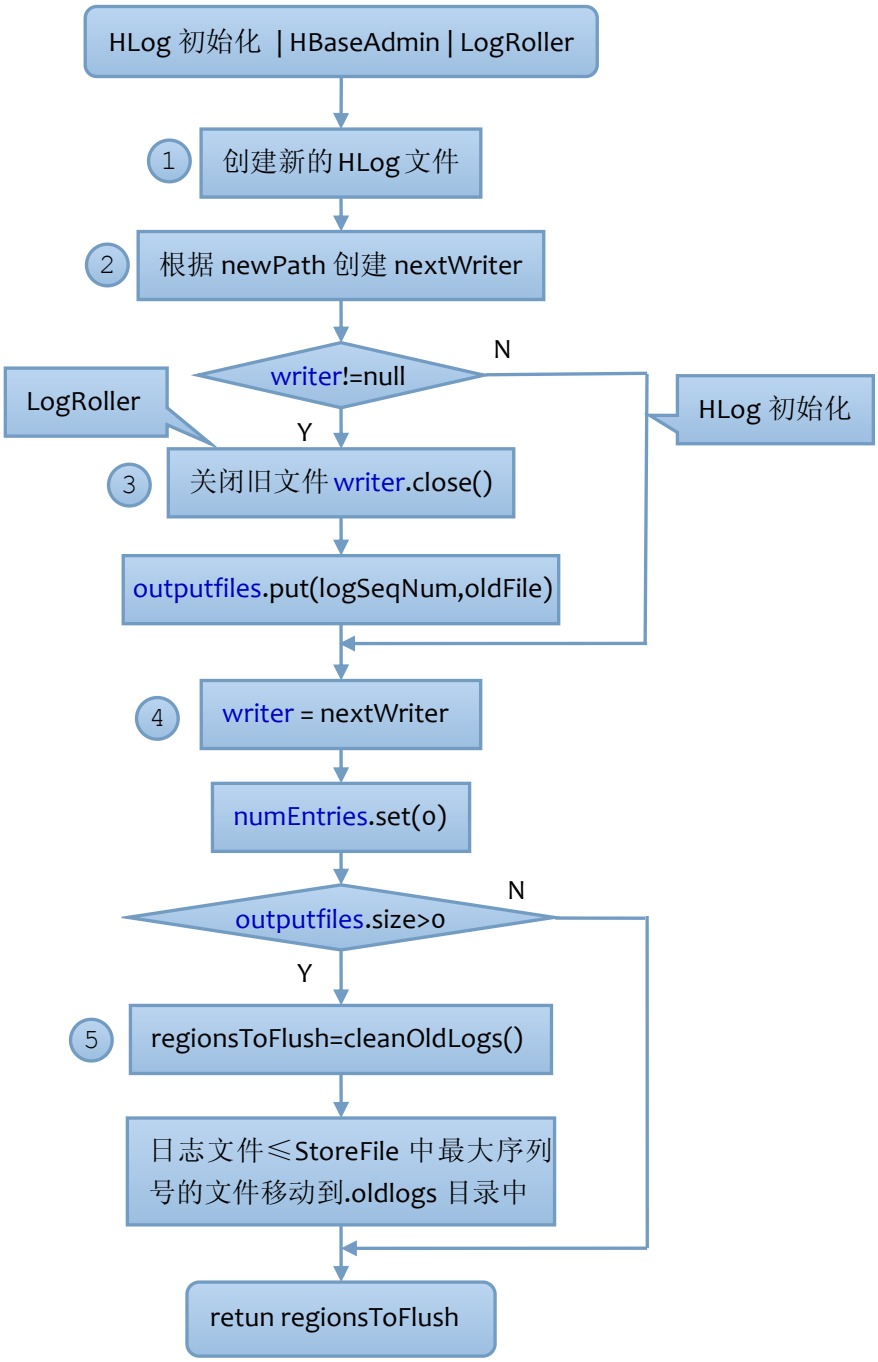

日志文件的设计思路:

日志表示客户端的操作数据，需要满足追加的特性．客户端产生的数据追加到日志文件，会使得日志文件不断增大，如果日志文件太大，则后期的读取日志文件中的内容会非常消耗内存(打开一个很大的文件)．因此日志需要不间断地进行分割(比如 Tomcat 的 catalina.out 日志文件可以每天分割一个文件，想象要是一个日志文件存放着一年的数据将是多么的可怕)．每次分割都会新创建一个文件用来存放客户端最新操作的日志．而旧的日志文件都不会再变化．这种分割日志也叫做滚动日志：每隔一定时间滚动一次日志，产生一个旧的日志文件，创建一个新的空日志文件.

## ☆rollWriter☆:

rollWriter()方法是为了获得日志文件的 Writer 以及输出流，做些准备或清理工作.

1. 每次滚动日志时，都要确定新生成的文件名，因为滚动日志之后，接下来的日志信息保存在新文件里
2. 创建新文件的 Writer 和输出流
3. 如果存在旧的 Writer(首次新建文件时没有，第一次及之后的滚动 writer 都不为空)，做些清理操作
4. 设置当前的 writer=最新的 writer
5. 检查是否需要清理旧的日志文件




```
public byte [][] rollWriter() throws FailedLogCloseException, IOException {
    return rollWriter(false);              // 不强制滚动日志
}
public byte [][] rollWriter(boolean force) {
    if (!force && this.writer != null && this.numEntries.get() <= 0) return null;    // Return if nothing to flush.
    byte [][] regionsToFlush = null;
    this.cacheFlushLock.lock();            // 滚动日志时，如果获得cacheFlushLock锁，则刷新缓存不能进行!
    this.logRollRunning = true;            // 正在进行日志滚动. 当然如果刷新缓存正在进行，则滚动日志不能进行，等待!...
```

```
    try {
      if (closed) return regionsToFlush;
      // Do all the preparation outside of the updateLock to block as less as possible the incoming writes
      long currentFilenum = this.filenum;
      Path oldPath = null;
      if (currentFilenum > 0)
        oldPath = computeFilename(currentFilenum);        //computeFilename will take care of meta hlog filename
      this.filenum = System.currentTimeMillis();
1     Path newPath = computeFilename();                   // 创建HLog日志文件，以便将日志写入到该文件中

      if (!this.listeners.isEmpty())
        for (WALActionsListener i : this.listeners)
          i.preLogRoll(oldPath, newPath);                 // Tell our listeners that a new log is about to be created
2     HLog.Writer nextWriter = this.createWriterInstance(fs, newPath, conf);
      FSDataOutputStream nextHdfsOut = null;
      if (nextWriter instanceof SequenceFileLogWriter) {
        nextHdfsOut = ((SequenceFileLogWriter)nextWriter).getWriterFSDataOutputStream();
      }

      synchronized (updateLock) {
        Path oldFile = cleanupCurrentWriter(currentFilenum);    // Clean up current writer.  3
4       this.writer = nextWriter;
        this.hdfs_out = nextHdfsOut;
        this.numEntries.set(0);              // 为本次日志写作准备，将numEntries清零. 注意unflushedEntries并没有清零
      }
      if (!this.listeners.isEmpty())
        for (WALActionsListener i : this.listeners)
          i.postLogRoll(oldPath, newPath);                  // Tell our listeners that a new log was created

      // Can we delete any of the old log files?
5     if (this.outputfiles.size() > 0) {
        if (this.lastSeqWritten.isEmpty()) {                // Last sequenceid written is empty. Deleting all old hlogs
          // If so, then no new writes have come in since all regions were flushed (and removed from the lastSeqWritten map).
          // Means can remove all but currently open log file.
          for (Map.Entry<Long, Path> e : this.outputfiles.entrySet()) {
            archiveLogFile(e.getValue(), e.getKey());
          }
          this.outputfiles.clear();
        } else {
          regionsToFlush = cleanOldLogs();
        }
      }
    } finally {
      this.logRollRunning = false;          // 日志滚动结束了
      this.cacheFlushLock.unlock();         // 释放cacheFlushLock锁
    }
    return regionsToFlush;
}
```

rollWriter()的①~④主要是为了新日志文件准备的 Writer. 如果是 HLog 初始化时，不会执行③，因为此时 writer=null，只有 LogRoller 或者 HBaseAdmin 对日志进行滚动时，因为初始化 HLog 时，在④设置了 writer. 所以滚动日志时 writer!=null，由于要生成新的日志文件来保存客户端对 HRegion 接下来的更新. 所以原先旧的 writer 就得进行清理. cleanupCurrentWriter()清理当前 writer，并将所有旧的日志文件(除了当前日志文件)保存到 outputfiles 中.

```
/*Cleans up current writer closing and adding to outputfiles. Presumes we're operating inside an updateLock scope.*/
Path cleanupCurrentWriter(final long currentfilenum) throws IOException {
  Path oldFile = null;
  if (this.writer != null) {              // Close the current writer, get a new one.
    try {
      // Wait till all current transactions are written to the hlog. No new transactions can occur because we have the updatelock.
      if (this.unflushedEntries.get() != this.syncedTillHere) sync();                  // 清理当前writer时，不允许新的事务发生
      this.writer.close();
      this.writer = null;
      closeErrorCount.set(0);
    } catch (IOException e) {}
    if (currentfilenum >= 0) {          // 存在旧的日志文件
      oldFile = computeFilename(currentfilenum);
      this.outputfiles.put(Long.valueOf(this.logSeqNum.get()), oldFile);          // 输出文件
    }
  }
  return oldFile;
}
```

Here We Go. **updateLock 更新锁**，用于日志更新的锁. 日志更新时通过 doWrite()写入 HLog. updateLock 使用地方:

1. append() → doWrite() 追加日志记录
2. rollWriter() → cleanupCurrentWriter() 关闭旧的日志文件，设置新的 writer
3. completeCacheFlush() 刷新内存完毕，将标记位 HBASE::CACHEFLUSH 写入 HLog 文件

将更新写入日志文件(1.3.)以及关闭旧的日志文件都需要同步 updateLock，日志只要作用于 writer 就需要同步.

每次进行 rollWriter()日志滚动时，都会将最新的 logSeqNum 对应旧的日志文件放入 outputfiles 中.
因为只有进行日志滚动时，旧的文件不再保存日志，生成新的日志文件用于保存新的日志.

滚动日志时如果存在旧的日志文件即 outputfiles.size()>0. 日志文件可能的状态由两种:

1. 滚动日志时所有的 regions 都已经刷新完成. 日志文件对应的 regions 只要刷新完成，日志就失效移动到.oldlogs

```
private void archiveLogFile(final Path p, final Long seqno) throws IOException {
  Path newPath = getHLogArchivePath(this.oldLogDir, p);
  if (!this.listeners.isEmpty()) // Tell our listeners that a log is going to be archived.
    for (WALActionsListener i : this.listeners)  i.preLogArchive(p, newPath);
  HBaseFileSystem.renameAndSetModifyTime(this.fs, p, newPath));          // 移动旧的日志文件到.oldlogs中
  if (!this.listeners.isEmpty()) // Tell our listeners that a log has been archived.
    for (WALActionsListener i : this.listeners)  i.postLogArchive(p, newPath);
}
```

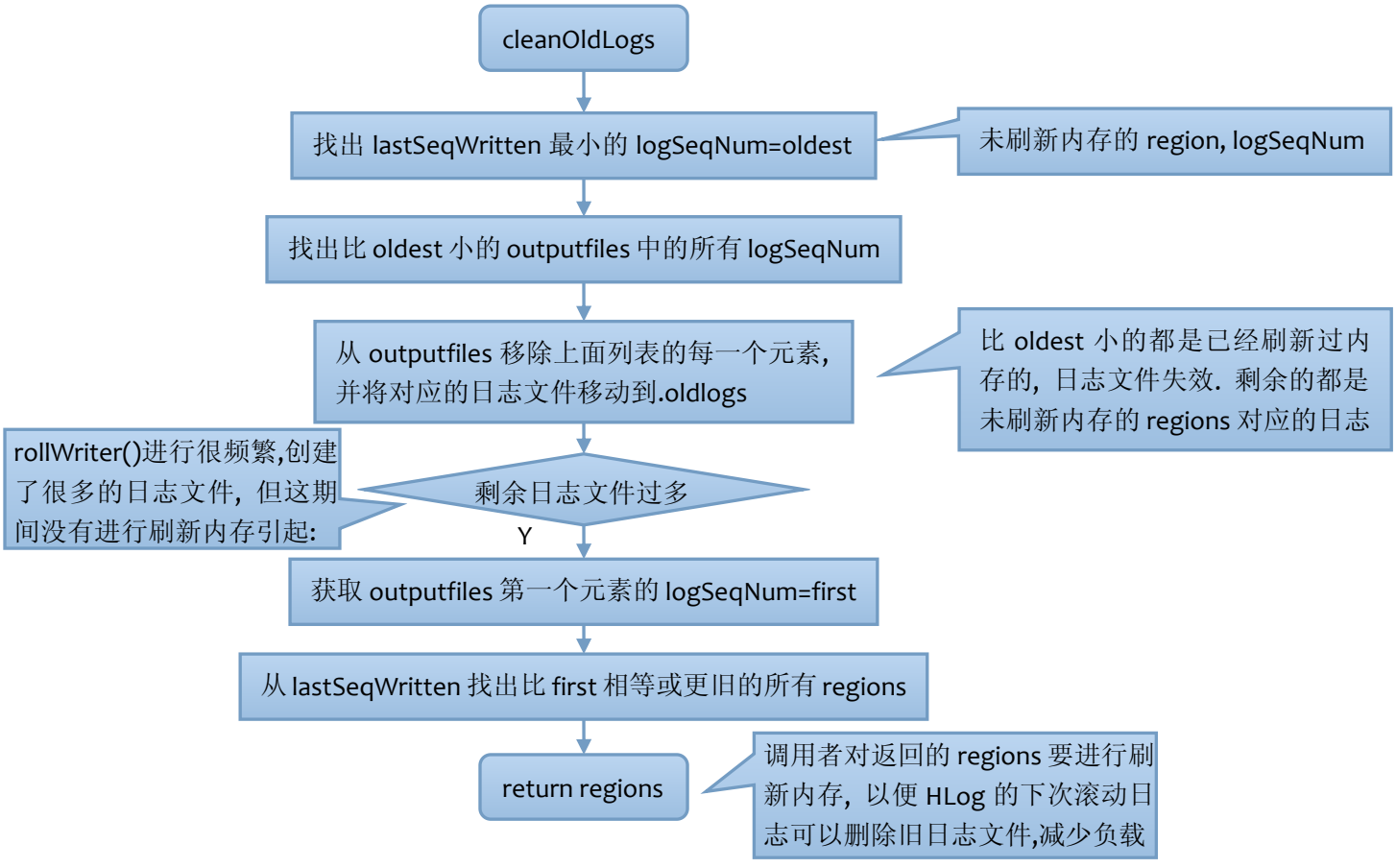
2. 有一部分日志文件对应的 regions 刷新，还有一部分没有刷新. 判断条件是 lastSeqWritten 不为空

lastSeqWrittent 保存的是没有刷新内存的日志. 如果已经刷新内存，会将<region,序列号>从中移除.
(开始刷新内存 startCacheFlush 时就移除，这样在刷新内存过程中仍然可以保证日志的更新).
lastSeqWritten 因为保存的是未刷新内存的日志对应的<region, seq>，所以只要比 lastSeqWritten 中保存的 value 最小的 logSeqNum 还小的日志文件都是已经刷新过内存的(cacheFlush)，这些日志文件可以移动到.oldlogs 中.

注意: 如果日志文件对应的 regions 有些 region 有刷新,有些没有刷新. 这个日志文件仍然不能移动到.oldlogs 中.

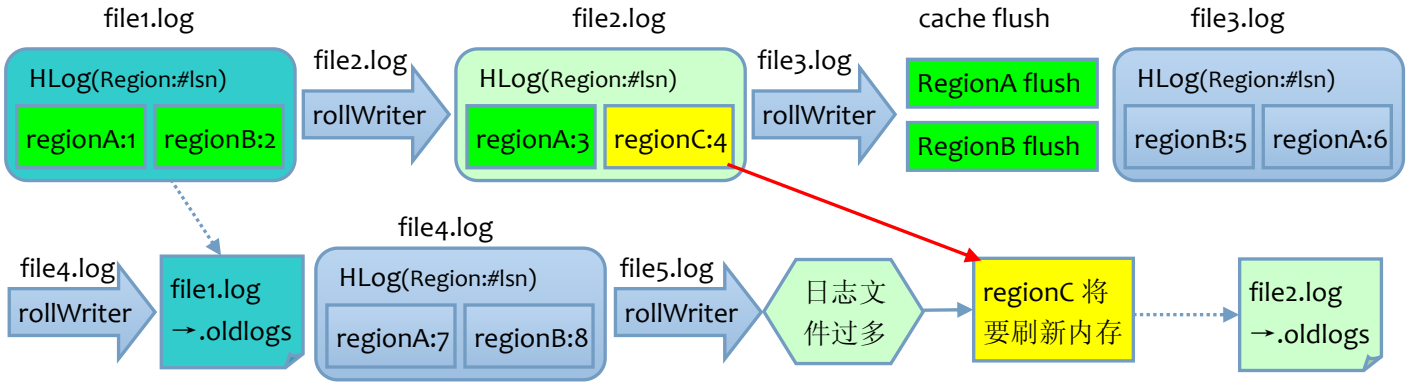
```
/* Clean up old commit logs.
 * @return If lots of logs, flush the returned region so next time through we can clean logs. Returns null if nothing to flush. */
private byte [][] cleanOldLogs() throws IOException {
  Long oldestOutstandingSeqNum = getOldestOutstandingSeqNum();   // outstanding未解决的，表示未刷新内存的
  // Get the set of all log files whose last sequence number is smaller than the oldest edit's sequence number.
  TreeSet<Long> sequenceNumbers = new TreeSet<Long>(this.outputfiles.headMap(
      (Long.valueOf(oldestOutstandingSeqNum.longValue()))).keySet());
  // Now remove old log files (if any)
  int logsToRemove = sequenceNumbers.size();
  if (logsToRemove > 0)
    for (Long seq : sequenceNumbers)
      archiveLogFile(this.outputfiles.remove(seq), seq);

  // If too many log files, figure which regions we need to flush. Array is an array of encoded region names.
  byte [][] regions = null;
  int logCount = this.outputfiles == null? 0: this.outputfiles.size();
  if (logCount > this.maxLogs && logCount > 0) {
    regions = findMemstoresWithEditsEqualOrOlderThan(this.outputfiles.firstKey(), this.lastSeqWritten);
  }
  return regions;
}
```

上面的清理工作还有一个：如果未刷新内存对应的日志文件过多(outputfiles.size()>maxLogs)，造成这种情况的原因可能是文件系统负载过大以至于它不能以新数据被添加进来的速率来存储数据(更新记录大批量地添加到 HLog 日志文件中，造成日志文件过多，造成文件系统负载过大，所以需要将这些文件移动到.oldlogs 中并最终被删除，但是这些更新记录由于还没有经过刷新内存这一步，所以不能移动!-只有刷新到内存后，日志文件才可以删除！所以这里我们返回需要进行刷新内存的 regions 列表，以便调用者触发刷新内存操作，在下一次 rollWriter 就可以删除旧文件).

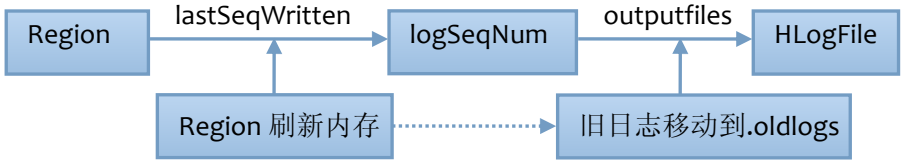


模拟 rollWriter() → cacheflush → rollWriter() → ... → rollWriter() → cleanoldlogs()
lastSeqWritten 保存的是<未刷新内存的 region, logSeqNum>. 如果已经刷新, 则会移除元素. 已经存在,不会再放入
而 outputfiles 保存的是<logSeqNum, 日志文件>. 如果要清理日志文件移动到.oldlogs, 则移除元素. 存在会覆盖.

| 动作 | 结果 |
|---|---|
| init | file1.log 初始化 HLog 时创建 file1.log 日志文件 |
| **regionA, 1; regionB,2** | lastSeqWritten:<regionA,1>,<regionB,2> |
| rollWriter() | →file2.log 每次日志滚动时,都会创建一个新的日志文件,并把旧日志文件加入 outputfiles<br>outputfiles:**<2,file1.log>** |
| **regionA,3, regionC,4** | lastSeqWritten:<regionA,1>,<regionB,2>,~~**<regionA,3>,<regionC,4>**~~ |
| rollWriter() | →file3.log 创建新日志文件 file3.log, 并把旧日志文件 file2.log 加入 outputfiles<br>outputfiles:<2,file1.log>,**<4,file2.log>** |
| RegionA /B cacheflush | lastSeqWritten:<regionC,4> 刷新缓存时, 把 region 对应元素从 lastSeqWritten 删除 |
| regionB,5, regionA,6 | lastSeqWritten:<regionC,4>,**<regionB,5>,<regionA,6>** |
| rollWriter() | →file4.log 创建新日志文件 file4.log, 并把旧日志文件 file3.log 加入 outputfiles<br>→outputfiles:<2,file1.log>,<4,file2.log>,**<6,file3.log>**<br>file1.log→.oldlogs 从 outputfiles 找出比 lastSeqWritten.#lsn=4 小的, 只有<1, file1.log><br>因为 RegionC 还没有刷新缓存, 所以 file2.log 还暂时不能被移动到.oldlogs(将被删除)<br>regionA, regionB 已经刷新过一次, 对应 file1.log 可以删除. 但 file3.log 则还不能删除.<br>→outputfiles:<4,file2.log>,<6,file3.log> |
| regionA,7, regionB,8 | lastSeqWritten:<regionC,4>,<regionB,5>,<regionA,6>,~~<regionA,7>,<regionB,8>~~ |
| rollWriter() | →file5.log 创建新日志文件 file5.log, 并把旧日志文件 file4.log 加入 outputfiles<br>→outputfiles:<4,file2.log>,<6,file3.log>,<8,file4.log><br>region 没有刷新, 不断 rollWriter 造成日志文件过多, 要让 region 进行内存刷新<br>→outputfiles 第一个元素的 logSeqNum=firstKey=4<br>→lastSeqWritten 中比 firstKey=4 相等或更旧的 region<br>→regionC → 让 regionC 进行内存刷新 → 下一次 rollWriter 时删除对应的旧日志文件 |
| RegionC cacheflush | lastSeqWritten:<regionB,5>,<regionA,6> |
| rollWriter() | →file6.log 创建新日志文件 file6.log, 并把旧日志文件 file5.log 加入 outputfiles<br>→outputfiles:<4,file2.log>,<6,file3.log>,<8,file4.log>,<9,file5.log><br>找出 lastSeqWritten 最小的#lsn=5, outputfiles 中比 5 小的只有 file2.log,<br>file2.log 会移动到.logs 中(file2.log 还有 regionA, 但其在第六步已经刷新过了) |




Region, #lsn(logSeqNum), HLogFile

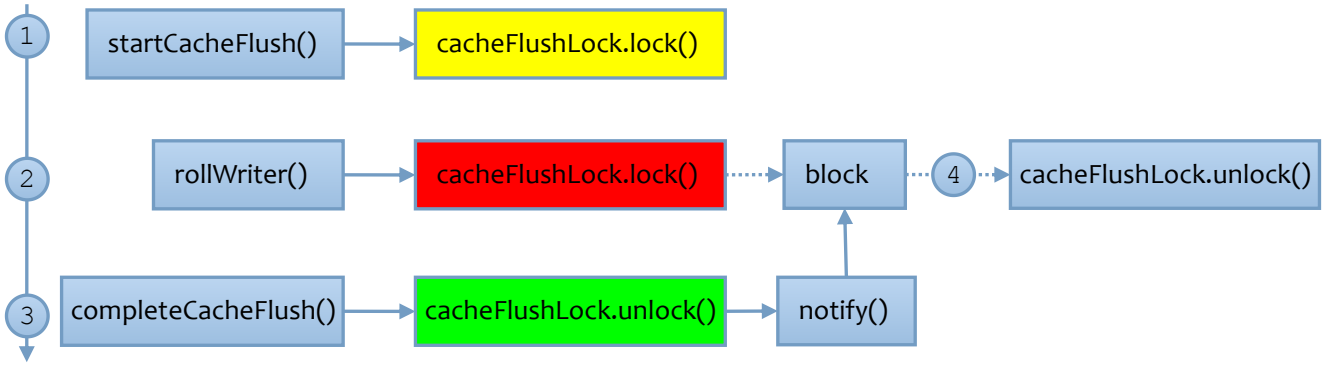


**rollWriter() & cacheFlush()**

```
/**Roll the log writer. That is, start writing log messages to a new file.开始写入日志信息到新文件中
 * Because a log cannot be rolled during a cache flush, 在缓存刷新时，不能进行日志滚动
 * and a cache flush spans two method calls, 缓存刷新会分成两个方法调用(startCacheFlush和completeCacheFlush).
 * a special lock needs to be obtained so that a cache flush cannot start
 * when the log is being rolled and the log cannot be rolled during a cache flush.
 *  缓存刷新和日志滚动共同竞争一个锁(cacheFlushLock)，两者只要其中一个获得锁，另外一个就不能进行.
 * <p>Note that this method cannot be synchronized because it is possible that  该方法不能同步
 * 1. startCacheFlush runs, obtaining the cacheFlushLock, 有可能缓存刷新刚开始，获得了cacheFlushLock
 * 2. then this method could start which would obtain the lock on this  然后该方法启动想要获得cacheFlushLock
 * but block on obtaining the cacheFlushLock  但是因为已经被缓存刷新进程占用了，所以该方法会阻塞
 * 3. and then completeCacheFlush could be called which would wait for the lock on this  然后HLog调用completeCacheFlush
 * and consequently never release the cacheFlushLock  继续等待cacheFlushLock，导致永远无法释放cacheFlushLock
 * @return If lots of logs, flush the returned regions so next time through  刷新返回regions的日志，下次就可以清理这些日志
 * we can clean logs. Returns null if nothing to flush.  Names are actual  返回 null 表示没有要刷新的日志 */
```

简单地说滚动日志和缓存刷新共同竞争锁 cacheFlushLock. 其中一个获得锁，另外一个就不能进行.
MemStore 缓存刷新 flush cache 会分成两个方法: startCacheFlush 和 completeCacheFlush.
所以 rollWriter()不能进行同步即方法加上 synchronized. 因为同步方法，并发条件下就不能获得锁，会造成死锁:

1. startCacheFlush()首先执行，对 cacheFlushLock 进行加锁
2. rollWriter()接着执行，如果是同步方法，则进入方法内，会尝试也对 cacheFlushLock 进行加锁.
   但是这个时候 cacheFlushLock 已经被 startCacheFlush()加锁了，所以 rollWriter()会被阻塞住.
   虽然 rollWriter()获取不到锁，但是因为使用的锁在同步方法里面，所以仍然占用了这个锁!
3. completeCacheFlush()想要释放 cacheFlushLock，要释放一个锁，也要保证持有锁的情况下才可以释放.
   但是因为 cacheFlushLock 在 rollWriter()中被同步了，导致这里无法释放.
   而 rollWriter()中又获取不到 cacheFlushLock 锁. 一个是想获取锁却被阻塞，一个要释放锁却被同步了.




刷新缓存和滚动日志共用一个 cacheFlushLock，是因为要保证两者不能同时进行(WHY?).
而刷新缓存在跟 cacheFlushLock 相关的操作上分成了两个方法 startCacheFlush 和 completeCacheFlush.
由 startCacheFlush 获得 cacheFlushLock，最后在 completeCacheFlush 中释放 cacheFlushLock.
因此在这个过程中间，如果有其他操作也要加 cacheFlushLock 这个锁，必须保证不能进行同步.
因为一旦同步了操作，相当于占用了 cacheFlushLock 锁，尽管你是无法获得这个锁的.
这样锁之间相互依赖，很容易造成死锁，导致无法释放锁.
注意到 cacheFlushLock 是个可重入锁 ReentrantLock.

问题:

1. 刷新缓存和滚动日志为什么不能同时进行?
2. 刷新缓存和滚动日志使用 cacheFlushLock，那么 updateLock, flushLock 分别用于哪些场景的锁策略?
3. 是不是可以得出结论: 当作用于同一个锁时, synchronized 和 ReentrantLock 不能同时使用?

**startCacheFlush() & completeCacheFlush()**

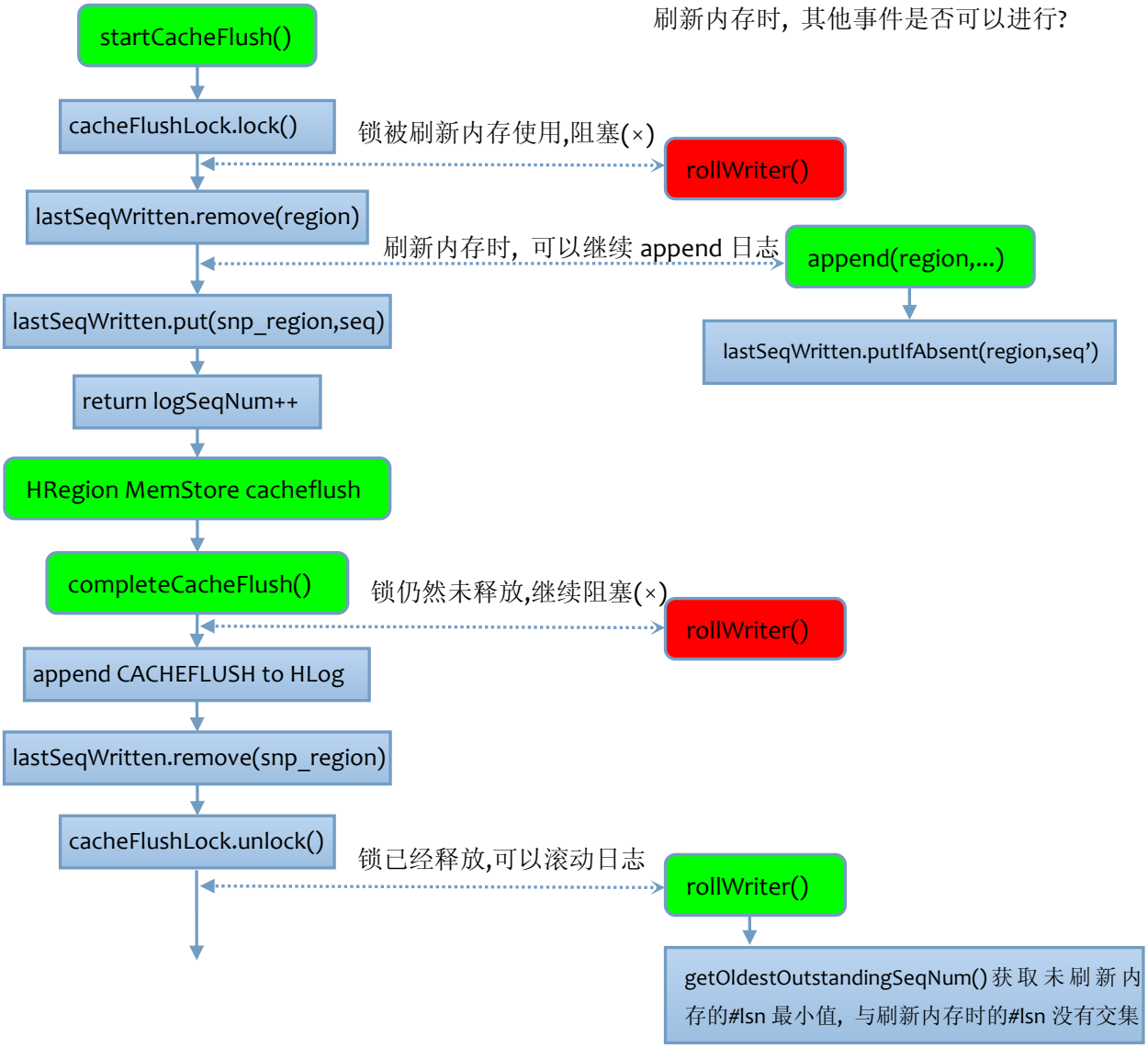
```
/**By acquiring a log sequence ID, we can allow log messages to continue while we flush the cache.允许刷新内存时继续接收日志更新
 * Acquire a lock so that we do not roll the log between the start and completion of a cache-flush. 开始和结束缓存之间,不能滚动日志
 * Otherwise the log-seq-id for the flush will not appear in the correct logfile. 否则刷新时刻的#lsn不会正确地出现在HLog文件中.
 *  即刷新缓存时，记录刷新的#lsn(会作为刷新完成的标记写入HLog中). 以便HLog知道是哪一次缓存的刷新.
 * Ensuring that flushes and log-rolls don't happen concurrently also allows us to  刷新缓存和滚动日志不能同时进行
 * temporarily put a log-seq-number in lastSeqWritten against the region being flushed  允许我们将#lsn临时放入lastSeqWritten
 * that might not be the earliest in-memory log-seq-number for that region. 防止region被刷新，而日志却不是内存中最早的#lsn.
 * By the time the flush is completed or aborted and before the cacheFlushLock is released  在锁释放前，保存的是最旧的.
 * it is ensured that lastSeqWritten again has the oldest in-memory edit's lsn for the region that was being flushed.
 * In this method, by removing the entry in lastSeqWritten for the region being flushed  从lastSeqWritten中移除条目
 * we ensure that the next edit inserted in this region will be correctly recorded in append  在刷新缓存的时候,仍可以接受日志更新
 * The #lsn of the earliest in-memory lsn - which is now in the memstore snapshot -  内存中旧的#lsn被放入内存的快照中.
 * is saved temporarily in the lastSeqWritten map while the flush is active.    */
public long startCacheFlush(final byte[] encodedRegionName) {
  this.cacheFlushLock.lock();
  Long seq = this.lastSeqWritten.remove(encodedRegionName);   // 在开始缓存刷新时就移除,append就可以对同region继续put
  // seq is the lsn of the oldest edit associated with this region. 当前region的最旧的更新. 因为对同一个region存在时不会再put了
  // If a snapshot already exists - because the last flush failed - then seq will be the lsn of the oldest edit in the snapshot
  if (seq != null) {   // keeping the earliest sequence number of the snapshot in lastSeqWritten maintains the correctness of
    // getOldestOutstandingSeqNum(). But it doesn't matter really because everything is being done inside of cacheFlush lock.
    lastSeqWritten.put(getSnapshotName(encodedRegionName), seq);
  }
  return obtainSeqNum();              // 获取一个新的logSeqNum，因为最后的标记位也要写入HLog中.
}
public void completeCacheFlush(final byte [] encodedRegionName,final byte [] tableName, final long logSeqId...) {
  try {
    long txid = 0;
    synchronized (updateLock) {   // 类似append，只要涉及到写入更新，就要加上updateLock更新锁
      WALEdit edit = completeCacheFlushLogEdit();       // WALEdit只有一个KeyValue: HBASE::CACHEFLUSH
      HLogKey key = makeKey(encodedRegionName, tableName, logSeqId,System.currentTimeMillis(), HConstants.DEFAULT_CLUSTER_ID);
      logSyncer.append(new Entry(key, edit));              // 将条目添加到LogSyncer的队列中. 由同步线程进行同步
      txid = this.unflushedEntries.incrementAndGet();
      this.numEntries.incrementAndGet();
    }
    this.sync(txid);                    // sync txn to file system  手动同步...
  } finally {   // updateLock not needed for removing snapshot's entry Cleaning up of lastSeqWritten is in the finally clause
    // because we don't want to confuse getOldestOutstandingSeqNum()  完成刷新，则该region对应的logSeqNum就没有用处了
    this.lastSeqWritten.remove(getSnapshotName(encodedRegionName));   // getOldest...获取的是所有未刷新缓存的最小lsn
    this.cacheFlushLock.unlock();
  }
}
```

lastSeqWritten 的 oldest logSeqNum 的定义：假设对同一个 region 有多次批处理，每次批处理都获取递增的序列号. 针对同一个 region, lastSeqWritten 使用 putIfAbsent(): 如果存在 key 则不会放入 Map. 因此对同一个 region 的第一次批处理获取到的 logSeqNum 是该 Region 最旧 oldest 的序列号.

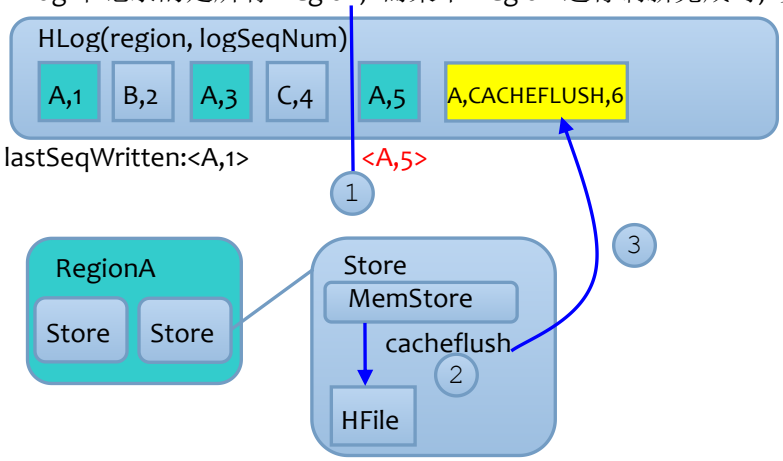
在开始刷新缓存时，就将 region 从 lastSeqWritten 移除(确保对该 region 的更新可以继续进行). 然后随后又把 region 对应的 snp_regionname 放入 lastSeqWritten. 为的是 rollWriter()方法调用 getOldestOutstandingSeqNum()获取到正确的 logSeqNum: 所有未刷新缓存的最小/最旧的 logSeqNum.

实际上不重新加入也没问题，因为 getOldestOutstandingSeqNum()是在 rollWriter()中被调用. 而 startCacheFlush(),

completeCacheFlush()和 rollWriter()竞争持有 cacheFlushLock 锁的，两者不可同时进行．而在 startCacheFlush 将 snp_regionname 和对应的#lsn 加入，最后在 completeCacheFlush()又移除掉．等价于 snp_regionname 及#lsn 是在缓存刷新期间可见的．那么 rollWriter()不管任何时候调用 getOldestOutstandingSeqNum()获取的结果都是正确的．




HLog 中记录的是所有 Region，而某个 Region 进行刷新完成时，要把该 region 及最新的#lsn 写入 HLog 中




在[C,4]后 RegionA 进行了缓存刷新．此时 HLog 仍然可以接受日志的更新(HLog 和 MemStore 此时没有必要的联系，实际上不管 MemStore 是否进行缓存刷新，HLog 任何时候都可以接受日志的更新，只不过如果 region 正在更新时，如果对同一个 region 还有更新过来，则 lastSeqWritten 保存的是该 region 在内存刷新开始时第一次的 logSeqNum).

**问题：为什么在刷新缓存完成时需要在 HLog 中添加一个结束标记?**

# SequenceFileLogWriter

日志滚动需要创建一个新文件，并创建相应的 Writer 对象，以便将日志的数据写入到日志文件中.
将文件名 newPath 传递给 createWriterInstance，生成该日志文件的写入流对象 SequenceFileLogWriter.
createWriterInstance() → HLogFileSystem.createWriter() → HLog.createWriter() → Writer.init()

```
public static Writer createWriter(final FileSystem fs, final Path path, Configuration conf){   /**Get a writer for the WAL. */
    if (logWriterClass == null)
      logWriterClass = conf.getClass("hbase.regionserver.hlog.writer.impl", SequenceFileLogWriter.class, Writer.class);
    HLog.Writer writer = (HLog.Writer) logWriterClass.newInstance();
    writer.init(fs, path, conf);
    return writer;
}
```

HLog 的 Writer 实现类是 SequenceFileLogWriter，在创建完对象后，就会调用其 init 初始化方法.

```
public void init(FileSystem fs, Path path, Configuration conf) {
  if (null == keyClass)   keyClass = HLog.getKeyClass(conf);   // HLogKey
  // Create a SF.Writer instance.  采用反射机制. Method.invoke(Object, params) --> SequenceFile.createWriter(parms....)
  this.writer = (SequenceFile.Writer) SequenceFile.class
      .getMethod("createWriter", new Class[] {FileSystem.class, Configuration.class, Path.class, Class.class, Class.class,
          Integer.TYPE, Short.TYPE, Long.TYPE, Boolean.TYPE, CompressionType.class, CompressionCodec.class, Metadata.class})
      .invoke(null, new Object[] {fs, conf, path, HLog.getKeyClass(conf), WALEdit.class,
          Integer.valueOf(fs.getConf().getInt("io.file.buffer.size", 4096)),
          Short.valueOf((short)conf.getInt("hbase.regionserver.hlog.replication", FSUtils.getDefaultReplication(fs, path))),
          Long.valueOf(conf.getLong("hbase.regionserver.hlog.blocksize", FSUtils.getDefaultBlockSize(fs, path))),
          Boolean.valueOf(false), SequenceFile.CompressionType.NONE, new DefaultCodec(), createMetadata(conf, compress)});
  this.codec = WALEditCodec.create(conf, compressionContext);   // setup the WALEditCodec
  this.writer_out = getSequenceFilePrivateFSDataOutputStreamAccessible();
  this.syncFs = getSyncFs();   // SequenceFile.Writer的syncFs方法
  this.hflush = getHFlush();   // FSDataOutputStream的hflush方法. 在Hadoop-0.20以及1.x中没有此方法.
}
private Method getSyncFs() {  // function pointer to writer.syncFs() method; present when sync is hdfs-200.
  return this.writer.getClass().getMethod("syncFs", new Class<?> []{});
}
private Method getHFlush() {   /**See if hflush (0.21 and 0.22 hadoop) is available. */
  Class<? extends OutputStream> c = getWriterFSDataOutputStream().getClass();
  return c.getMethod("hflush", new Class<?> []{});
}
```

通过 Class.getMethod 获取到 syncFs 和 hflush 的 Method 对象.  然后通过反射 Method.invoke(obj, params) 就可以调用的 Class 类的 Method 方法. syncFs 对应 SequenceFile.Writer, hflush 对应 FSDataOutputStream,都不需要参数.

```
public void sync() throws IOException {
  if (this.syncFs != null) {
      this.syncFs.invoke(this.writer, HLog.NO_ARGS);
  } else if (this.hflush != null) {
      this.hflush.invoke(getWriterFSDataOutputStream(), HLog.NO_ARGS);
  }
}
```

# LogSyncer

回到 HLog.appendNoSync()→LogSyncer.append(HLog.Entry)

```
// appends new writes to the pendingWrites. 追加新的日志写记录到LogSyncer的缓存列表中
// It is better to keep it in our own queue rather than writing it to the HDFS 先保存在自己的队列中而不是马上就写入HDFS
// output stream because HDFSOutputStream.writeChunk is not lightweight at all. 因为HDFS的输出流对象不是一个轻量级的方法
synchronized void append(Entry e) throws IOException {
  pendingWrites.add(e);
}
```

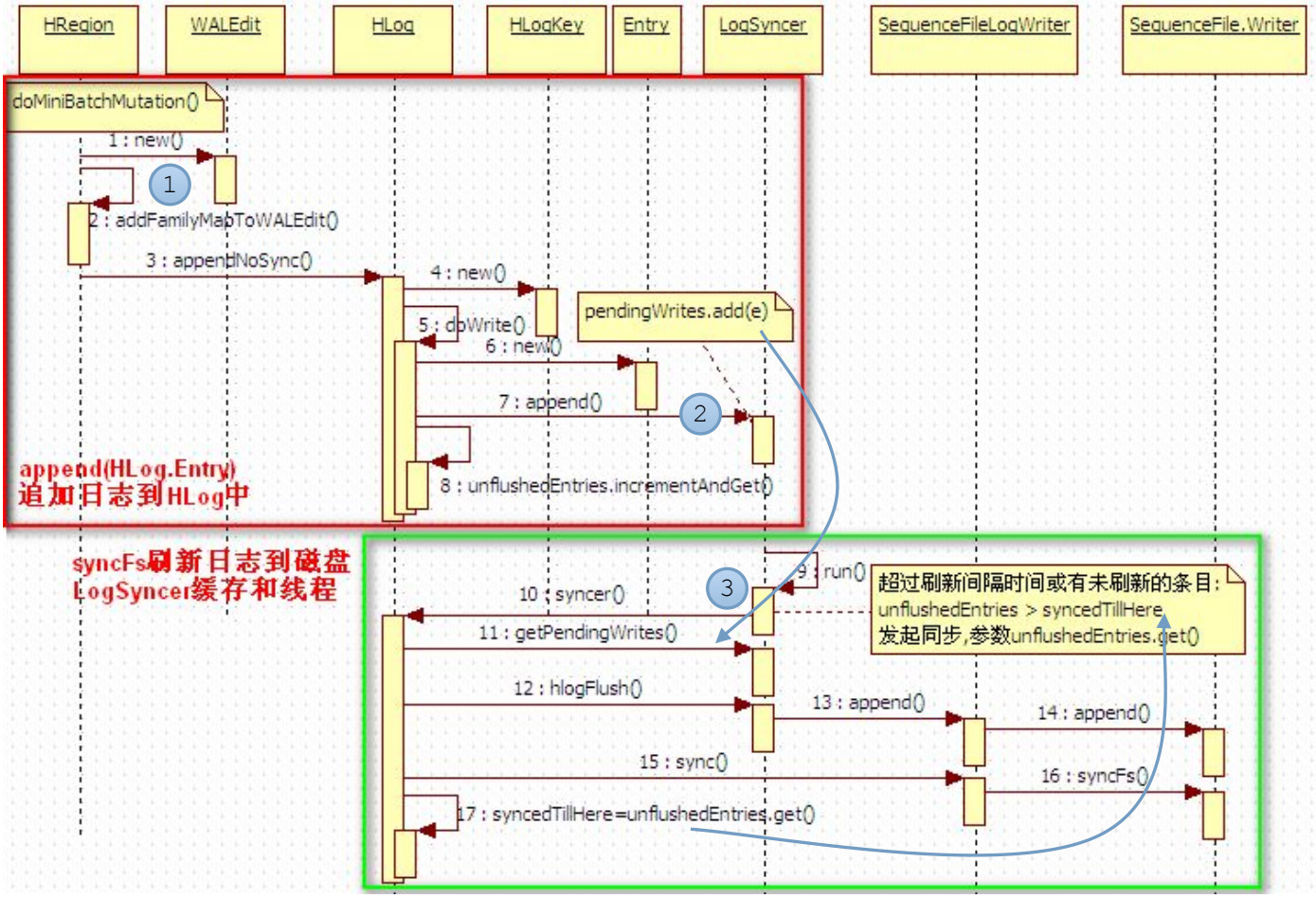
在 HLog.append()调用完 doWrite()后，递增了 numEntries 和 unflushedEntries 两个原子变量.

```
this.numEntries.incrementAndGet();
txid = this.unflushedEntries.incrementAndGet();          // 未刷新的条目的事务ID
```

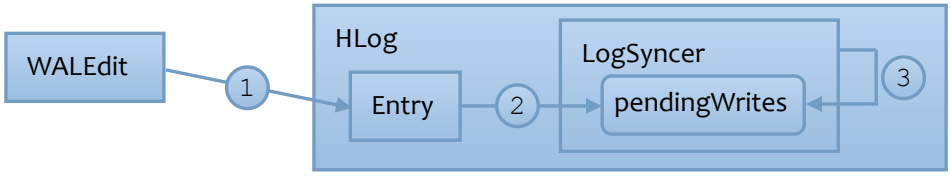
unflushedEntries 表示在当前 HLog 中未刷新的事务数. 调用一次 doMiniBatchMutation 产生一次事务.
unflushedEntries 变量会用于 LogSyncer 和 syncedTillHere 进行比较 HLog 中是否还有未刷新到磁盘上的事务.

下图示例了一次批处理(mini)追加到 HLog 中，并且被 LogSyncer 日志同步线程同步到 HDFS 上的过程:

1. HLog.append()创建 HLogKey，并在 doWrite()中以 HLogKey 和 WALEdit 构造 HLog.Entry 对象.
2. HLog 的 Entry 条目没有立即同步到 HDFS，而是先加入到 LogSyncer.pendingWrites 队列中
3. LogSyncer 线程启动后判断是否需要同步，达到同步的条件则调用 HLog.syncer()
   同步时获取 LogSyncer.pendingWrites 队列元素，使用 Writer 输出流对象同步到 HDFS.




下图是组件结构图，分别对应上面的流程图的相应阶段:



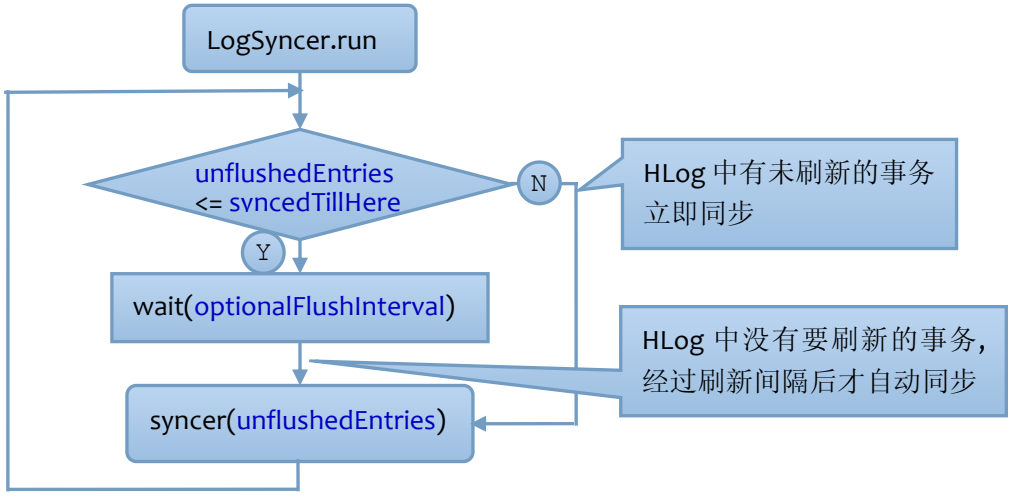
## LogSyncer 线程

LogSyncer 是一个线程. 在创建 HLog 的时候会创建 LogSyncer 对象，传入同步间隔时间，并启动 LogSyncer 线程:

```
/**This class is responsible to hold the HLog's appended Entry list
 * and to sync them according to a configurable interval.  定时同步HLog中的条目
 * Deferred log flushing works first by piggy backing on this process by simply not sync'ing the appended Entry.
 * It can also be sync'd by other non-deferred log flushed entries outside of this thread. */
class LogSyncer extends HasThread {
    private final long optionalFlushInterval;
    private AtomicBoolean closeLogSyncer = new AtomicBoolean(false);
    // List of pending writes to the HLog. There corresponds to transactions that have not yet returned to the client.
    // We keep them cached here instead of writing them to HDFS piecemeal，不是细碎地写入到HDFS中,而是先缓存起来
    // because the HDFS write method is pretty heavyweight as far as locking is concerned.  因为HDFS些方法是重量级的:锁
    // The goal is to increase the batchsize for writing-to-hdfs as well as sync-to-hdfs,
    // so that we can get better system throughput.  目标是增加写入到HDFS的批大小，以及同步到HDFS,来获取更好的系统吞吐量
    private List<Entry> pendingWrites = new LinkedList<Entry>();

    LogSyncer(long optionalFlushInterval) {
        this.optionalFlushInterval = optionalFlushInterval;
    }
    public void run() {
        while(!this.isInterrupted() && !closeLogSyncer.get()) {
            if (unflushedEntries.get() <= syncedTillHere) {
                synchronized (closeLogSyncer) {
                    closeLogSyncer.wait(this.optionalFlushInterval);   // 等待时间: 刷新间隔(1s，会不会太短了?)
                }
            }
            // Calling sync since we waited or had unflushed entries.  超过等待间隔时间或者有未刷新的条目,则发起同步操作
            // Entries appended but not sync'd are taken care of here AKA deferred log flush  未同步的，已经追加的条目会被处理
            sync();
        }
    }
```

发生同步的条件有两种:

1. unflushedEntries.get() > syncedTillHere 未刷新的事务编号>目前为止已经同步的事务编号
   HLog 中有未刷新的条目，立即同步
2. unflushedEntries.get() <= syncedTillHere HLog 中没有要刷新的条目，所以需要等待刷新间隔过后才同步一次.



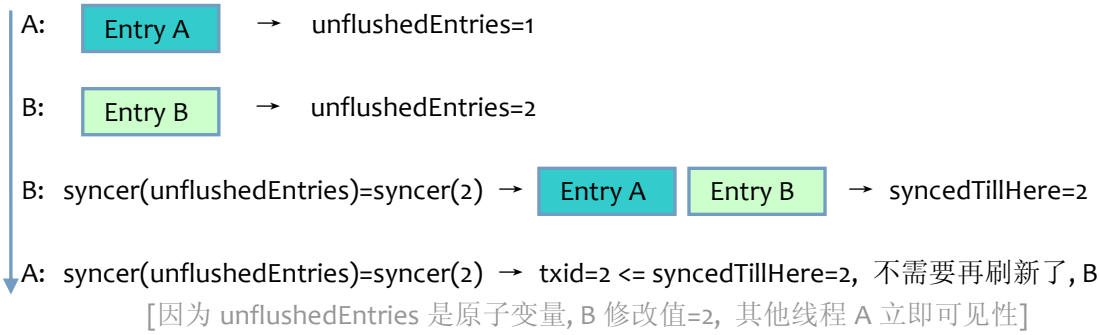
## syncer 同步

由 LogSyncer 产生的同步，需要传入 unflushedEntries 的值表示事务 ID. 在同步时如果有其他线程同步过了就返回.

```
public void sync() throws IOException {
    syncer();
}
private void syncer()   {                    // sync all known transactions  同步所有已知的事务
    syncer(this.unflushedEntries.get());     // sync all pending items
}
private void syncer(long txid) {             // sync all transactions upto the specified txid  同步最多到指定的txid的所有事务
    if (txid <= this.syncedTillHere) return; // if the transaction that we are interested in is already synced, then return immediately.
    Writer tempWriter;
    synchronized (this.updateLock) {
      if (this.closed) return;
      tempWriter = this.writer;              // guaranteed non-null  前面通过反射创建出Writer对象
    }
      long doneUpto;
      // First flush all the pending writes to HDFS. Then issue the sync to HDFS. If sync is successful,
      // then update syncedTillHere to indicate that transactions till this number has been successfully synced.
      List<Entry> pending = null;
      synchronized (flushLock) {
        if (txid <= this.syncedTillHere) return;
        doneUpto = this.unflushedEntries.get();
        pending = logSyncer.getPendingWrites();      //  获取批处理时append加入到LogSyncer队列的HLog.Entry
        logSyncer.hlogFlush(tempWriter, pending);    //  最终将LogSyncer队列所有Entry加入append到SequenceFile中
      }
      if (txid <= this.syncedTillHere) return;             // another thread might have sync'ed avoid double-sync'ing
      tempWriter.sync();                                   //  调用Writer实现类.syncFs(反射)→SequenceFile.Writer的syncFs刷新到HDFS
      this.syncedTillHere = Math.max(this.syncedTillHere, doneUpto);   //  赋值表示事务直到这个数字都已经成功地刷新到HDFS了
      if (!this.logRollRunning) {
            checkLowReplication();
            if (tempWriter.getLength() > this.logrollsize) {
              requestLogRoll();
            }
      }
}
```

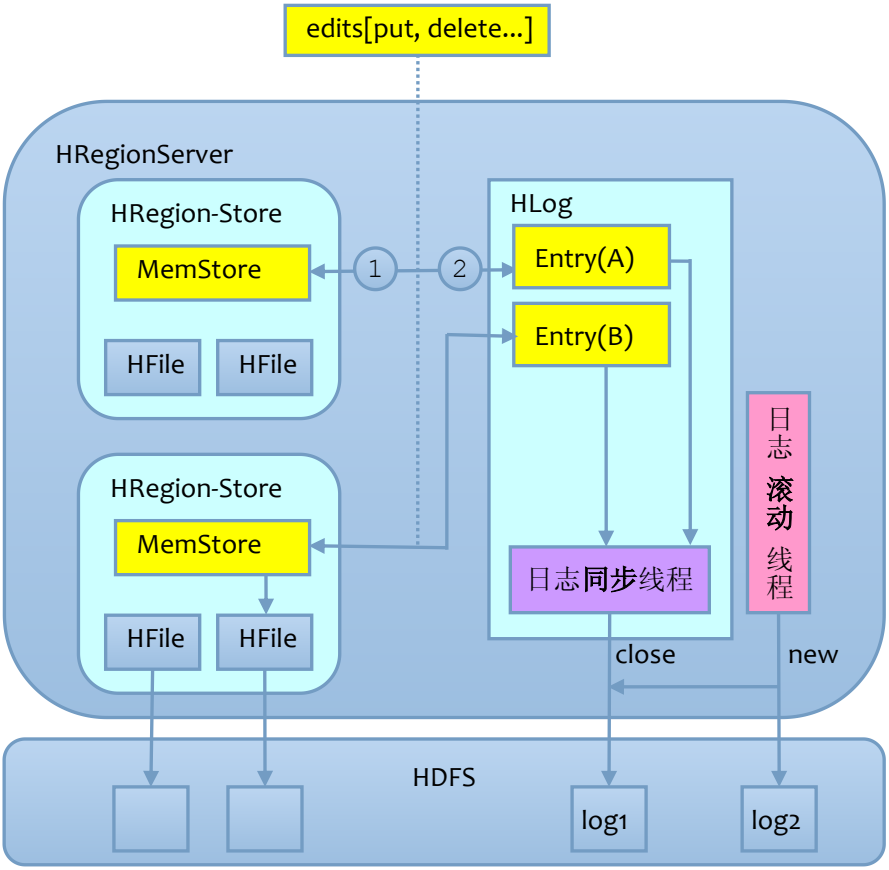
LogSyncer 的同步是将更新日志写入 HLog 中，客户端过来的日志需要时时刻刻地保存下来(因此同步线程默认 1s 同步一次)，而 MemStore 的内存刷新则是将内存中的数据持久化到磁盘的过程，这个过程很耗时，间隔就比较大了.

下面模拟两个线程并发调度，其中一个把所有未同步的刷新到 HDFS 后，另外一个线程就不需要再刷新了.



# LogRoller

客户端更新记录发送到 HRegion 进行批处理，会将更新发送到 MemStore 内存和 HLog 文件中分成两部分来存储. HLog 可以看做是 MemStore 内存形式的磁盘备份形式. 客户端不断更新记录, MemStore 内存会不断增大, HLog 日志文件也不会不断增大(刷新更新到 HDFS 磁盘的 HLog 文件使用了 HLog 的内部类 LogSyncer 每隔一秒刷新一次). 日志文件作为 MemStore 的备份，当 MemStore 刷新到 HDFS 永久存储到 HDFS 后，日志文件也就没有存在的必要了. 在分析 HLog 的 rollWriter()时提到刷新内存和滚动日志不可以同时进行，这两个操作共用了 cacheFlushLock.




```
/** Runs periodically to determine if the HLog should be rolled. */
class LogRoller extends HasThread implements WALActionsListener {
  private final ReentrantLock rollLock = new ReentrantLock();          // 可重入锁
  private final AtomicBoolean rollLog = new AtomicBoolean(false);      // 是否发起日志滚动请求，请求的话立即滚动
  private final Server server;                                         // RegionServer
  protected final RegionServerServices services;                       // RegionServer
  private volatile long lastrolltime = System.currentTimeMillis();    // 上一次进行日志滚动的时间
  private final long rollperiod;                                       // Period to roll log. 滚动日志的时间间隔(1h)
  private final int threadWakeFrequency;                               // 线程检查间隔(10s)

  public LogRoller(final Server server, final RegionServerServices services) {
    super();
    this.server = server;
    this.services = services;
    this.rollperiod = this.server.getConfiguration().getLong("hbase.regionserver.logroll.period", 3600000);
    this.threadWakeFrequency = this.server.getConfiguration().getInt(HConstants.THREAD_WAKE_FREQUENCY, 10 * 1000);
  }
  public void run() {
    while (!server.isStopped()) {
      long now = System.currentTimeMillis();
      boolean periodic = false;
```

```
        if (!rollLog.get()) {
          periodic = (now - this.lastrolltime) > this.rollperiod;
          if (!periodic) {                          // 滚动日志的时间间隔没到，继续等待，每隔10s检查一次
            synchronized (rollLog) {
              rollLog.wait(this.threadWakeFrequency);
            }
            continue;
          }
        }
        // Time for periodic roll  开始日志滚动
        rollLock.lock();        // FindBugs UL_UNRELEASED_LOCK_EXCEPTION_PATH
        try {
          this.lastrolltime = now;
          byte [][] regionsToFlush = getWAL().rollWriter(rollLog.get());    // This is array of actual region names.  需要刷新的Region
          if (regionsToFlush != null) {
            for (byte [] r: regionsToFlush)
              scheduleFlush(r);                     // 调度每个Region刷新：刷新HLog日志到磁盘上
          }
        } finally {
          rollLog.set(false);
          rollLock.unlock();
        }
      }
    }
```

HLog.rollWriter()进行日志滚动需要的工作．返回值 regions 表示因为日志文件过多需要刷新的 region 列表．
调度发起 Region 的 MemStore 的内存刷新，实现类是 MemStoreFlusher，这又是一个包含队列的线程．

```
  private void scheduleFlush(final byte [] encodedRegionName) {
    HRegion r = this.services.getFromOnlineRegions(Bytes.toString(encodedRegionName));
    FlushRequester requester = this.services.getFlushRequester();
    requester.requestFlush(r);
  }
```

# MemStoreFlusher

```
class MemStoreFlusher extends HasThread implements FlushRequester {
  // These two data members go together.    Any entry in the one must have a corresponding entry in the other.
  private final BlockingQueue<FlushQueueEntry> flushQueue = new DelayQueue<FlushQueueEntry>();
  private final Map<HRegion, FlushRegionEntry> regionsInQueue = new HashMap<HRegion, FlushRegionEntry>();

  public void requestFlush(HRegion r) {
    synchronized (regionsInQueue) {
      if (!regionsInQueue.containsKey(r)) {
        // This entry has no delay so it will be added at the top of the flush queue. It'll come out near immediately.
        FlushRegionEntry fqe = new FlushRegionEntry(r);    // 记录了Region的创建时间，失效时间，重试次数
        this.regionsInQueue.put(r, fqe);
        this.flushQueue.add(fqe);
      }
    }
  }
```

FlushRegionEntry 在添加到 flushQueue 中，会立即被 flushQueue 获取到，因为其失效时间=创建时间.
FlushRegionEntry 实现了 Delayed 接口，当超过失效时间时，就可以从队列中取出元素，否则一直等待.
(上一次出现 Delayed 接口的是 Lease 租约对象，在租约超时后，从队列中取出元素，释放其占用的资源)

```
interface FlushQueueEntry extends Delayed {}  // Delayed接口最重要的是getDelay()，返回值<=0表示超过失效时间,可从队列中取出

/**Datastructure used in the flush queue. Holds region and retry count. Keeps tabs on how old this object is.
 * Implements Delayed. On construction, the delay is zero. 构造函数，延迟时间=0，这样getDelay()就返回0:
 * When added to a delay queue, we'll come out near immediately. 一旦添加到延迟队列，就可以立即从队列中获取出来
 * Call #requeue(long) passing delay in milliseconds 在重新加入延迟队列前,传递需要延迟的时间给requeue方法
 * before readding to delay queue if you want it to stay there a while.  可以让元素在队列中保持一会儿(做些初始化工作等) */
static class FlushRegionEntry implements FlushQueueEntry {
  private final HRegion region;      // 要进行刷新的Region
  private final long createTime;     // 创建时间
  private long whenToExpire;         // 失效时间，如果失效时间=创建时间，则立即从队列中取出，马上刷新
  private int requeueCount = 0;      // 重试次数，如果想延迟从队列中取出时间，增加失效时间

  FlushRegionEntry(final HRegion r) {
    this.region = r;
    this.createTime = System.currentTimeMillis();
    this.whenToExpire = this.createTime;
  }
  public FlushRegionEntry requeue(final long when) {
    this.whenToExpire = System.currentTimeMillis() + when;  // 增加失效的时间. 就会等待when这些时间后，才从队列中取出
    this.requeueCount++;
    return this;
  }
  public long getDelay(TimeUnit unit) {
    return unit.convert(this.whenToExpire - System.currentTimeMillis(), TimeUnit.MILLISECONDS);
  }
}
```

另外还有一个 WakeupFlushThread，只是用来做标记，也会被加入 flushQueue，而且加入后也会立即被取出来.

```
/**Token to insert into the flush queue that ensures that the flusher does not sleep */
static class WakeupFlushThread implements FlushQueueEntry {
  public long getDelay(TimeUnit unit) {
    return 0;
  }
  public int compareTo(Delayed o) {
    return -1;
  }
}
```

标记元素的作用是：不带有逻辑的其他可能触发刷新内存操作的调用者想要立刻执行 run()方法里面的逻辑.
比如当 Region 的所有 MemStore 占用的内存超过限制时 isAboveHighWaterMark(),isAboveLowWaterMark()会触发:

```
private void wakeupFlushThread() {
  if (wakeupPending.compareAndSet(false, true)) {      // 如果wakeupPending=false，则设置值=true，表示正在唤醒
    flushQueue.add(new WakeupFlushThread());           // 加入到flushQueue中，以便立刻执行MemStoreFlusher.run()的逻辑判断
  }
}
```

前面大致分析了刷新内存，但是具体由哪些条件可以出发内存刷新?

1. HBaseAdmin.flush() → HRegionServer.flushRegion() → HRegion.flushcache()
2. HRegion.close() 关闭时，内存中的数据需要持久化到磁盘上
3. MemStoreFlusher 内存刷新线程．如果内存占用超过阀值，或者滚动日志时需要刷新内存

```
private AtomicBoolean wakeupPending = new AtomicBoolean();      // 是否正在唤醒
private final long threadWakeFrequency;              // 线程唤醒的时间间隔，即 flushQueue 每隔 10s 执行一次 poll
private final HRegionServer server;
private final ReentrantLock lock = new ReentrantLock();
private final Condition flushOccurred = lock.newCondition();
protected final long globalMemStoreLimit;            // 单个Region内所有的MemStore大小总和,高位,超过时全部刷新
protected final long globalMemStoreLimitLowMark; // 低位,超过该值时,选择一个region刷新
private static final float DEFAULT_UPPER = 0.4f;      // 强制block所有的更新并flush这些region以释放所有memstore占用的内存
private static final float DEFAULT_LOWER = 0.35f;     // 找一个memstore内存占用最大的region,做flush.此时写更新还是会被block
private static final String UPPER_KEY = "hbase.regionserver.global.memstore.upperLimit";
private static final String LOWER_KEY = "hbase.regionserver.global.memstore.lowerLimit";

public MemStoreFlusher(final Configuration conf, final HRegionServer server) {
   super();
   this.server = server;
   this.threadWakeFrequency = conf.getLong(HConstants.THREAD_WAKE_FREQUENCY, 10 * 1000);
   long max = ManagementFactory.getMemoryMXBean().getHeapMemoryUsage().getMax();
   this.globalMemStoreLimit = globalMemStoreLimit(max, DEFAULT_UPPER, UPPER_KEY, conf);
   long lower = globalMemStoreLimit(max, DEFAULT_LOWER, LOWER_KEY, conf);
   this.globalMemStoreLimitLowMark = lower;
}
```

用来表示 Region 所有的 MemStore 的两个水位限制分别是 lower 和 max．类似水库对水位有一个安全阀值的范围限制．如果超过低位，则只需打开一个水闸放水．如果超过最高警戒线：高位，则要打开所有水闸了．

MemStoreFlusher 的两个阀值变量确切地说是 **HRegionServer 的所有 Region 的所有 MemStore**(中间有个 Store).

```
private boolean isAboveHighWaterMark() {     /**Return true if global memory usage is above the high watermark*/
   return server.getRegionServerAccounting().getGlobalMemstoreSize() >= globalMemStoreLimit;
}
private boolean isAboveLowWaterMark() {      /**Return true if we're above the high watermark*/
   return server.getRegionServerAccounting().getGlobalMemstoreSize() >= globalMemStoreLimitLowMark;
}
```

安全阀值的判断在水流入之前就要进行判断．正如更新记录刷新到内存前就要判断 MemStore 的内存大小．

客户端发送批处理到 RegionServer.multi 后(其他操作也有)，在 HRegion 处理批处理 batchMutate 之前就判断:

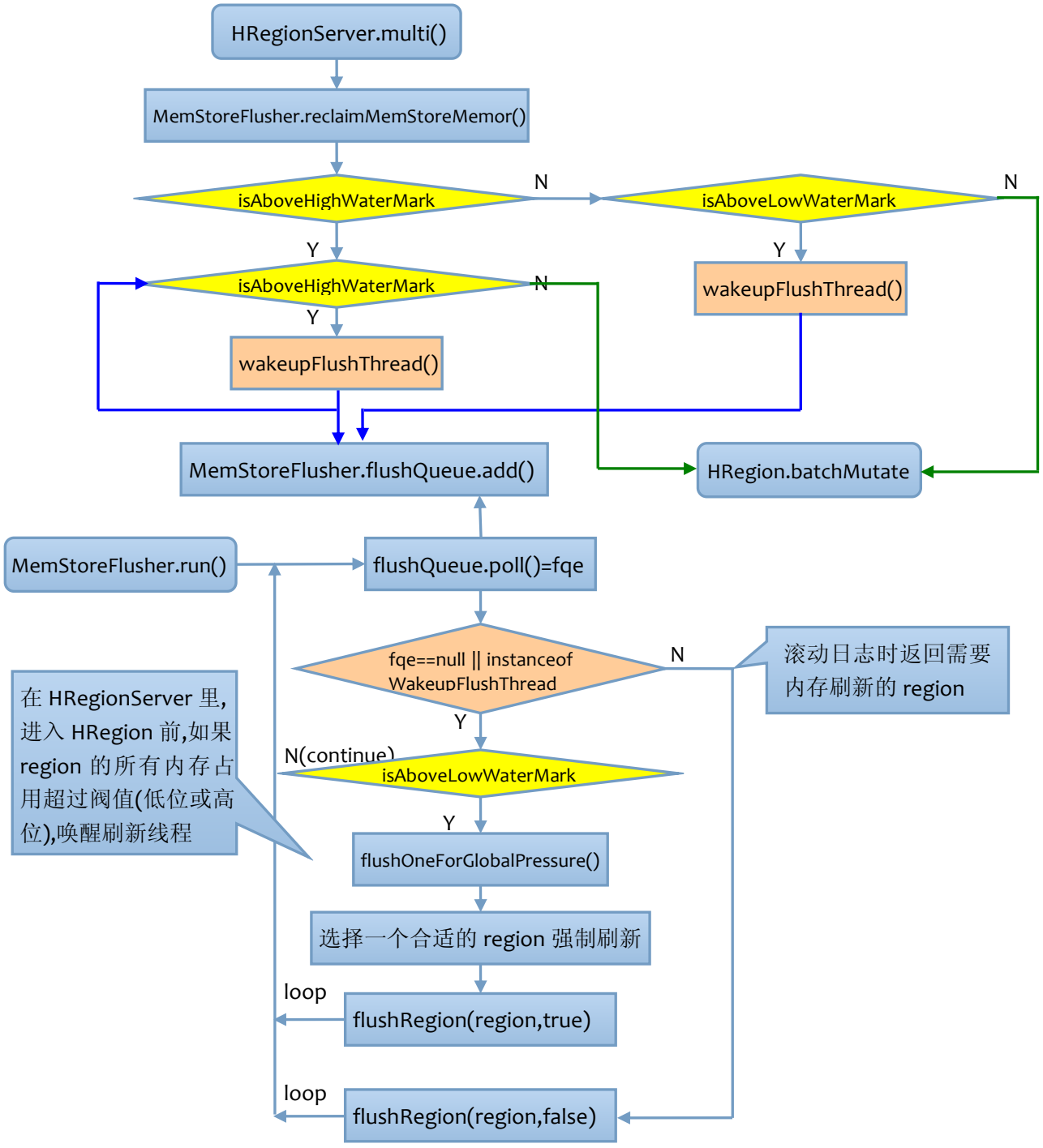
```
HRegion region = getRegion(regionName);
if (!region.getRegionInfo().isMetaTable()) {
   this.cacheFlusher.reclaimMemStoreMemory();   // 回收再利用MemStore的内存，刷新region就可以释放内存
}
```

在进入 HRegion 处理之前，在 HRegionServer 中就进行内存回收，是因为 RegionServer 要选择一个合适的 Region 进行内存刷新，如果进入 HRegion 之后才做，就被限制在当前 Region 里，因为我们要找到一个占用内存最大的 region.

```
/** Check if the regionserver's memstore memory usage is greater than the limit.
 * If so, flush regions with the biggest memstores until we're down to the lower limit.
 * This method blocks callers until we're down to a safe amount of memstore consumption. */
public void reclaimMemStoreMemory() {
   if (isAboveHighWaterMark()) {
      boolean blocked = false;
      long startTime = 0;
```

```
        while (isAboveHighWaterMark() && !server.isStopped()) {   // 超过最高警戒线，会一直唤醒刷新线程
          blocked = true;
          wakeupFlushThread();
        }
      } else if (isAboveLowWaterMark()) {   // 超过最低警戒线，则只需唤醒一次刷新线程：选择一个region进行内存刷新
        wakeupFlushThread();
      }
    }
```

回收 MemStore 占用的内存，如果只超过低位，则只需唤醒一次，选择一个 Region 进行内存刷新即可．如果超过的是最高水位，则要不断循环唤醒多次．但 MemStoreFlusher 线程里只根据低位来判断．超过低位就刷新一个 region．因为高水位的值肯定比低水位的值大，超过高水位的一定超过低水位．在 run()里只需要判断低水位，并且每次只执行一次：一个 region 的内存刷新．　要满足高水位唤醒多次内存刷新，在 reclaimMemStoreMemory()中通过 while 循环保证：仍然超过高水位,则继续往 flushQueue 中添加标记唤醒线程的元素，以便 run()继续从头开始执行判断方法．一旦没超过高水位(在低水位和高水位之间，或者在低水位下)就无需刷新 region 的内存了．



```
public void run() {
  while (!this.server.isStopped()) {
      wakeupPending.set(false);                                  // allow someone to wake us up again
      // 间隔threadWakeFrequency执行一次poll检查线程. 为空表示加入到队列中的元素还没有到达失效时间
      FlushQueueEntry fqe = flushQueue.poll(threadWakeFrequency, TimeUnit.MILLISECONDS);
      if (fqe == null || fqe instanceof WakeupFlushThread) {
        if (isAboveLowWaterMark()) {  // Flush thread woke up because memory above low water  刷新线程被唤醒因为内存超值
          if (!flushOneForGlobalPressure()) {        // Region的MemStore内存压力过大，所以要选择一个Region来刷新内存
            wakeupFlushThread();  // Enqueue another one of these tokens so we'll wake up again  失败的话需要再次唤醒
          }
          continue;  // 继续从队列中取元素，如果是超过高水位(就一定超过低水位)，要继续选择region来刷新
        }
        FlushRegionEntry fre = (FlushRegionEntry)fqe;      // 不为空，表示队列中的元素超过失效时间.
        if (!flushRegion(fre)) break;                      // 滚动日志时如果日志文件过多，会让region刷新内存，释放内存
  }
  this.regionsInQueue.clear();
  this.flushQueue.clear();
}
```

由于 RegionServer 的所有 region 的所有 MemStore 的内存占用负载过大，因此要选择一个 Region 刷新其内存，释放一部分内存(刷新内存，MemStore 会持久化到磁盘，MemStore 占用的内存就会被释放). 如何选择这样的 region? 如果 Region 已经存储的 StoreFiles 超过该 Region 可以存储的 StoreFile 数量的限制,就不选择该 Region. 因为如果该 Region 已经有很多 StoreFile 了，再选择该 Region 进行内存刷新，只会使 StoreFile 的数量更多. 因此要选择那些 StoreFile 数量不少很多的 Region 来刷新，这样可以保证 RegionServer 中所有的 Region 的 StoreFile 数量很平均.

选择一个 MemStore 内存占用最大的 Region 方法: getBiggestMemstoreRegion(). 参数 checkStoreFileCount 表示是否需要检查 Region 的 StoreFile 的数量，如果要检查的话，如果 region 存储的 StoreFile 数量超过某阀值则不选择它. 对于最合适的需要进行内存刷新的 Region(bestFlushableRegion)，这个参数=true.

```
private HRegion getBiggestMemstoreRegion(SortedMap<Long, HRegion> regionsBySize, // 所有的region
      Set<HRegion> excludedRegions, boolean checkStoreFileCount) { // 失败的在exclude中，是否检查StoreFile数量
  synchronized (regionsInQueue) {
    for (HRegion region : regionsBySize.values()) {
      if (excludedRegions.contains(region)) continue;  // 在不包含的列表中，则肯定不会选择了
      // 检查StoreFile数量的话，如果超过可以存储的StoreFile的数量，则不会选择该region.
      if (checkStoreFileCount && isTooManyStoreFiles(region)) continue;
      return region;
    }
  }
  return null;
}
```

有可能我们选择了一个最合适的 bestFlushableRegion，以及任意一个 region: bestAnyRegion. 如果 bestAnyRegion 占用的内存比 2*bestFlushableRegion 占用的内存还大. 则我们要选择这个 bestAnyRegion. 因为我们的目的是释放 RegionServer 中某个 Region 占用的内存. 对于占用内存过大的 Region 要优先选择. 其次才是根据 StoreFile 的数量.

```
/**The memstore across all regions has exceeded the low water mark. Pick one region to flush and flush it synchronously */
private boolean flushOneForGlobalPressure() {
  SortedMap<Long, HRegion> regionsBySize = server.getCopyOfOnlineRegionsSortedBySize();  // 根据region内存占用大小排序
  Set<HRegion> excludedRegions = new HashSet<HRegion>();  // 选择的region刷新失败，加入其中. 不会被再次选择
  boolean flushedOne = false;       // 刷新了一个region,设置为true，停止while循环.
  while (!flushedOne) {             // 只要找到一个Region可以刷新就停止. 否则找另外的region.
    // Find the biggest region that doesn't have too many storefiles (might be null!) 没有太多StoreFile. 若有太多，则刷新后更多.
```

```
        HRegion bestFlushableRegion = getBiggestMemstoreRegion(regionsBySize, excludedRegions, true);
        // Find the biggest region, total, even if it might have too many flushes.  任何一个region.
        HRegion bestAnyRegion = getBiggestMemstoreRegion(regionsBySize, excludedRegions, false);
        if (bestAnyRegion == null) return false;
        HRegion regionToFlush;
        if (bestFlushableRegion != null && bestAnyRegion.memstoreSize.get() > 2 * bestFlushableRegion.memstoreSize.get()) {
          // Even if it's not supposed to be flushed, pick a region if it's more than twice as big as the best flushable one    - otherwise
          // when we're under pressure we make lots of little flushes and cause lots of compactions, etc, which just makes life worse!
          regionToFlush = bestAnyRegion;
        } else {
          if (bestFlushableRegion == null) regionToFlush = bestAnyRegion;
          else regionToFlush = bestFlushableRegion;
        }
        Preconditions.checkState(regionToFlush.memstoreSize.get() > 0);
        flushedOne = flushRegion(regionToFlush, true);              // 内存压力过大，则要强制立刻进行内存刷新.
        if (!flushedOne) excludedRegions.add(regionToFlush);        // 选择的region刷新失败，加入到不包含列表中，重新while...
      }
      return true;
    }
```

## flushRegion()

flushOneForGlobalPressure()选择 region 时，如果 region 的 StoreFlie 数量过多不会选择．而 flushRegion()要刷新的 region 的文件过多，则要发起合并文件/拆分文件的请求．　选择一个 region 来刷新，碰到 region 的 StoreFile 文件数过多，就绕过这个 region．而 flushRegion()时已经确定是这个 region 了，没办法绕过，必须处理这个 region.

滚动日志时如果日志文件太多(Region 没有及时刷新，而滚动日志不断进行产生很多的日志文件使得系统负载过大)，则返回需要内存刷新的 Region(刷新完内存，就可以把对应的日志文件移动到.oldlogs 中删除减少文件系统负载)．选择的 region 进行内存刷新，也要首先检查 region 的 StoreFile 数量是否过多．如果过多，则要等待一段时间进行．

```
  /* A flushRegion that checks store file count. If too many, puts the flush on delay queue to retry later. */
  private boolean flushRegion(final FlushRegionEntry fqe) {
    HRegion region = fqe.region;
    if (!fqe.region.getRegionInfo().isMetaRegion() && isTooManyStoreFiles(region)) {
      if (!fqe.isMaximumWait(this.blockingWaitTime)) {
        if (fqe.getRequeueCount() <= 0) {
          if (!this.server.compactSplitThread.requestSplit(region))                    // 请求拆分Region.
            this.server.compactSplitThread.requestCompaction(region, getName());  // 请求合并Region
        }
        // Put back on the queue. Have it come back out of the queue after a delay of this.blockingWaitTime / 100 ms.
        this.flushQueue.add(fqe.requeue(this.blockingWaitTime / 100));
        return true;   // Tell a lie, it's not flushed but it's ok  延迟刷新．经过blockingWaitTime后就会调用flushRegion(region,false)
      }
    }
    return flushRegion(region, false);   // 如果要刷新的region没有过多的StoreFile，则可以立即进行内存刷新.
  }
```

将 FlushRegionEntry 重新加入队列中并使延迟时间增加 blockingWaitTime/100(0.9s)，为的是让 CompactSplitThread 线程由足够的时间发起合并和切分 Region 中过多的 StoreFile：延迟时间每次只增加 0.9s，而等待时间为 90s.
加入队列中的元素的延迟时间=0.9s,线程从队列中取出元素的时间为 10s，因此每次都可以取出,每次都再延迟 0.9s.
只有当等待时间超过 blockingWaitTime(与 createTime 比较而不是 whenToExpire)才发起 flushRegion()请求.

注意：在 run()循环里当从 flushQueue.poll()=fqe 时，就将 fqe 从 flushQueue 中移除！因此在上面的 flushQueue.add 时是一个删除后再添加的过程，由于 poll()的返回值就是我们想要的 fqe 对象，fqe.requeue()返回 this 也是 fqe 对象. 元素加入队列中并被取出后处理，下次队列再获取就不会获取已经处理过的元素相当于：add -> get then remove.

MemStoreFlusher 几个时间用于从队列中取出待内存刷新的 region:

| variable | value | desc |
|---|---|---|
| threadWakeFrequency | 10s | 线程唤醒频率，即每隔 10s 执行一次 run()里的 flushQueue.poll()获取元素<br>即每隔 10s 检查是否有需要内存刷新的 Region.<br>队列里的元素就是准备进行内存刷新的 region. |
| blockingWaitTime | 90s | 加入 flushQueue 队列中的 region 的 StoreFile 数量太多，<br>等待 90s 后才开始刷新 region 的缓存. 因为要对 region 发起拆分和合并. |
| requeue->whenToExpire | 0.9s | 将准备刷新的 region 重新加入 flushQueue，延迟时间+0.9s.<br>只要还没超过 blockingWaitTime，而且每次都延时 0.9s<10s,<br>因此每次 flushQueue.poll()，每隔 10s 都能从 flushQueue 取出后又重新放入 |

```
/* Flush a region.
 * @param emergencyFlush Set if we are being force flushed. If true the region needs to be removed from the flush queue.
 * If false, when we were called from the main flusher run loop and we got the entry to flush  在run()里获得要刷新的条目
 * by calling poll on the flush queue (which removed it).  通过调用flushQueue.poll()获得后就从flushQueue中移除. */
private boolean flushRegion(final HRegion region, final boolean emergencyFlush) {
    synchronized (this.regionsInQueue) {
        FlushRegionEntry fqe = this.regionsInQueue.remove(region);
        if (fqe != null && emergencyFlush) {    // 如果是WakeupFlushThread则不会执行. 必须是FlushRegioEntry
            // Need to remove from region from delay queue. When NOT an emergencyFlush, then item was removed via a flushQueue.poll.
            flushQueue.remove(fqe);
        }
        lock.lock();
    }
    try {
        boolean shouldCompact = region.flushcache();          // 刷新region的内存，返回值表示是否需要合并
        boolean shouldSplit = region.checkSplit() != null;        // We just want to check the size  检查是否需要拆分
        if (shouldSplit) this.server.compactSplitThread.requestSplit(region);
        else if (shouldCompact) server.compactSplitThread.requestCompaction(region, getName());
        server.getMetrics().addFlush(region.getRecentFlushInfo());
    } catch (DroppedSnapshotException ex) {
        return false;    // 返回false，如果是滚动日志触发，不再执行MemStoreFlusher，解决办法只能重启RegionServer
    } catch (IOException ex) {
        if (!server.checkFileSystem()) return false;    // 检查RegionServer文件系统失败，也要重启.
    } finally {
        flushOccurred.signalAll();
        lock.unlock();
    }
    return true;
}
```

flushRegion()如果遇到异常返回 false,如果是 flushOneForGlobalPressure()则会重新选择一个合适的 Region 再次刷新. 如果是由滚动日志返回需要刷新的内存，而返回 false，在 run()里直接 break 出 while 循环，然后清除 regionsInQueue 和 flushQueue. 那么问题是 MemStoreFlusher 此时就彻底不能执行 run()了. 肿么破？只能手动重启 RegionServer. 因为 MemStoreFlusher 是在 RegionServer 里初始化并启动的. 一旦不执行了，只能从入口处重新启动线程.

# Region.flushCache()

```
/**Flush the cache. When this method is called the cache will be flushed unless: 什么情况下不会刷新缓存(MemStore)
 *    the cache is empty 缓存为空
 *    the region is closed 关闭了Region
 *    a flush is already in progress 正在刷新缓存
 *    writes are disabled 写被禁用了
 * This method may block for some time, so it should not be called from a time-sensitive thread. 刷新内存很耗时，会阻塞一段时间
 * @return true if the region needs compaction 返回region是否需要合并
 * @throws IOException general io exceptions 根据这2个异常，如果返回false，目前的解决办法是重启RegionServer
 * @throws DroppedSnapshotException Thrown when replay of hlog is required because a Snapshot was not properly persisted.*/
```

flushCache()会调用 internalFlushcache(MonitoredTask )

```
Flush the memstore. Flushing the memstore is a little tricky.
We have a lot of updates in the memstore, all of which have also been written to the log. 内存中的更新也写入HLog
We need to write those updates in the memstore out to disk, 需要将内存中的更新刷新到磁盘上
while being able to process reads/writes as much as possible during the flush operation. 在刷新期间也要处理读写
Also, the log has to state clearly the point in time at which the memstore was flushed. 当内存刷新后,HLog中有明确的时间点来表示
(That way, during recovery, we know when we can rely on the on-disk flushed structures 在恢复时,我们就能根据磁盘上刷新过的结果
and when we have to recover the memstore from the log.) 并且从HLog中恢复内存(即内存刷新后的那个时间点(HLog的一个KeyValue)
So, we have a three-step process:
A. Flush the memstore to the on-disk stores, noting the current sequence ID for the log. 刷新内存到磁盘存储,记录当前序列号
B. Write a FLUSHCACHE-COMPLETE message to the log, using the sequence ID that was current at the time of memstore-flush. 标记完成
C. Get rid of清除the memstore structures that are now redundant多余的, as they've been flushed to the on-disk HStores. 清除内存数据
```

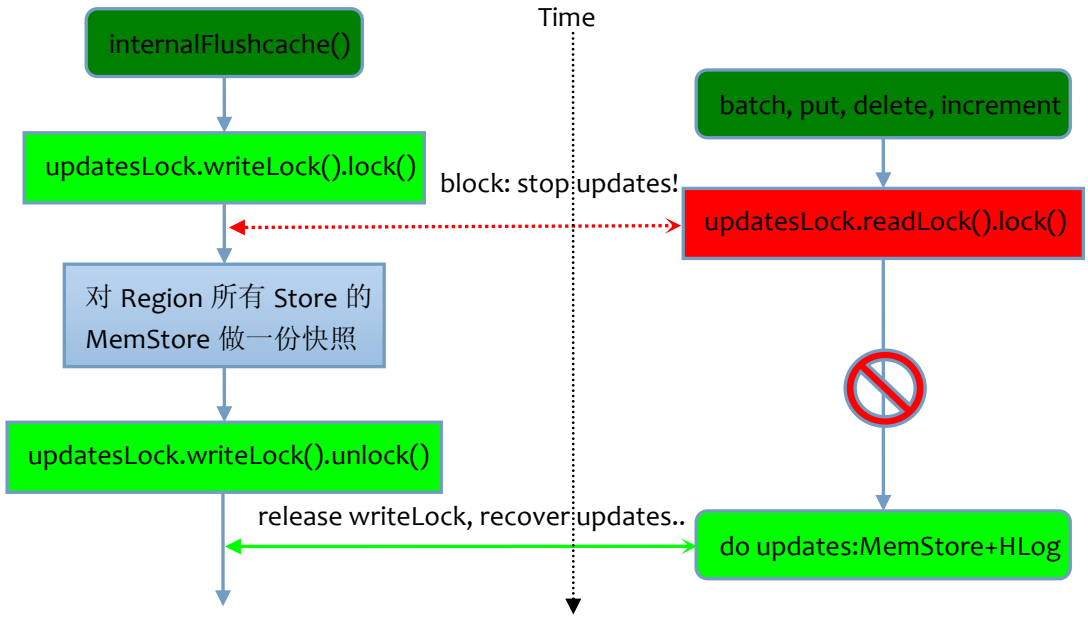
flushCache()时可以接受客户端的读写，当然也支持日志的写入．但是不支持在刷新内存时滚动日志．

```
Stop updates while we snapshot the memstore of all stores. 对一个Region中的所有Store进行快照时,停止更新(日志写入)
We only have to do this for a moment.  Its quick. 进入HLog的#lsn，在刷新完所有的快照后也会进入到info文件(???)
The subsequent sequence id that goes into the HLog after we've flushed all these snapshots
->调用HLog.completeCacheFlush时将completeSequenceId写入HLog中(completeSequenceId为startCacheFlush的返回值)
also goes into the info file that sits beside the flushed files. 刷新内存后, MemStore->StoreFile(HFile)
We also set the memstore size to zero here before we allow updates again 在允许再次更新之前,MemStore的大小=0
so its value will represent the size of the updates received during the flush
在刷新过程中,MemStore的大小表示接收到的更新的大小. MemStore表示占用内存的大小,更新到来时会写入内存. 没有更新,则大小=0
在刷新内存的时候，保存了快照，这样新的 MemStore 就仍然能够继续接受更新的到来.
```

刷新缓存时如果对 Region 的所有 Store 进行快照时，则不允许更新．使用了 ReentrantReadWriteLock updateLocks.
在 HRegion.batchMutate 的 step3 之前(获取 MVCC,写入 MemStore 之前)使用了 ReentrantReadWriteLock 的读取锁:
问题：批处理明明是写入操作，为什么获取的是读取锁，而不是写入锁？关键要弄清楚锁要保护的数据是什么....
HRegion 中只有 internalFlushcache()使用了 writeLock，其他方法如 put, delete, increment 都使用 readLock.

```
    lock(this.updatesLock.readLock(), numReadyToWrite);
    locked = true;
    w = mvcc.beginMemstoreInsert();
    applyFamilyMapToMemstore...
    addFamilyMapToWALEdit...
    this.log.appendNoSync(...)
    if (locked) {
      this.updatesLock.readLock().unlock();
      locked = false;
    }
```

刷新内存时在准备阶段即对 Region 的所有 Store 的 MemStore 做一份快照时，为这个过程加上写锁 writeLock. updatesLock 写锁发生的时候其他读锁，写锁如果也使用 updatesLock 则都会被阻塞住．直到快照完成释放写锁.




ReentrantReadWriteLock 的 readLock 和 writeLock 的一种猜测:
put, delete, increment 是客户端的操作，因为可能存在很多客户端，如果采用 writeLock，则同一时刻只有一个线程可以拥有 updatesLock，会使读写操作都阻塞住．而使用 readLock，则所有使用 updatesLock 的线程都可以并发执行，因为多个读取线程之间不会有问题．而刷新内存则是 Region 内部的动作，而且不是很常执行，所以使用 writeLock.
那么问题是：为什么刷新内存做快照时，客户端的更新操作不能进行? -- 后面说到 MemStore.snapshot 会举例分析.

```
protected boolean internalFlushcache(final HLog wal, final long myseqid, MonitoredTask status){
  long sequenceId = -1L;
  long completeSequenceId = -1L;
  MultiVersionConsistencyControl.WriteEntry w = null;   // MVCC: 涉及到并发写，需要使用MVCC来控制读写并发
  // We have to take a write lock during snapshot, or else a write could end up in both snapshot and memstore
  // (makes it difficult to do atomic rows then)  为了保证行级别的原子性，在做快照时需要同步写锁．即其他更新会被阻塞住
  status.setStatus("Obtaining lock to block concurrent updates");
  this.updatesLock.writeLock().lock();            // block waiting for the lock for internal flush  内部刷新时阻塞写
  long flushsize = this.memstoreSize.get();      // 要刷新的大小
  status.setStatus("Preparing to flush by snapshotting stores");
  List<StoreFlusher> storeFlushers = new ArrayList<StoreFlusher>(stores.size());
  try {
    // Record the mvcc for all transactions in progress. 记录下正在进行的所有事务的MVCC．即下面的写会携带memstoreWrite
    w = mvcc.beginMemstoreInsert();
    mvcc.advanceMemstore(w);

    sequenceId = (wal == null)? myseqid: wal.startCacheFlush(this.regionInfo.getEncodedNameAsBytes());
    completeSequenceId = this.getCompleteCacheFlushSequenceId(sequenceId);
    for (Store s : stores.values())    // region的所有Store
      storeFlushers.add(s.getStoreFlusher(completeSequenceId));   // 每个Store都有一个StoreFlusher
    for (StoreFlusher flusher : storeFlushers)
      flusher.prepare();              // prepare flush (take a snapshot)
  } finally {
    this.updatesLock.writeLock().unlock();
  }  // 快照结束，开始等待 MVCC---> MVCC.waitForRead()
```

构造 StoreFlusher 时，从 HLog 中获取最新的 logSeqNum，该序列号最终在内存刷新完毕后写入 HLog 作为一个标记.
因为一个 Region 由多个 Store 组成，所以刷新 Region 即刷新所有的 Store. 一个 Store 只有一个 MemStore.
获取 Region 的每个 StoreFlusher 后，要做准备工作 prepare: 对每个 Store 的 MemStore 进行快照.

```
public StoreFlusher getStoreFlusher(long cacheFlushId) {
  return new StoreFlusherImpl(cacheFlushId);
}
private class StoreFlusherImpl implements StoreFlusher {
  private long cacheFlushId;                    // 要刷新的ID(HLog分配的序列号)
  private SortedSet<KeyValue> snapshot;         // 刷新内存时，对MemStore进行快照的内容
  private StoreFile storeFile;                  // 内存MemStore会转换成StoreFile(HFile)
  private Path storeFilePath;                   // 内存MemStore的数据刷新到磁盘上HFile的路径
  private TimeRangeTracker snapshotTimeRangeTracker;
  private AtomicLong flushedSize;               // 已经刷新的大小

  private StoreFlusherImpl(long cacheFlushId) {
    this.cacheFlushId = cacheFlushId;
    this.flushedSize = new AtomicLong();        // 初始化时=0，因为还没开始刷新
  }
  public void prepare() {
    memstore.snapshot();                        // 对内存进行快照
    this.snapshot = memstore.getSnapshot();
    this.snapshotTimeRangeTracker = memstore.getSnapshotTimeRangeTracker();
  }
  public void flushCache(MonitoredTask status) throws IOException {
    storeFilePath = Store.this.flushCache(cacheFlushId, snapshot, snapshotTimeRangeTracker, flushedSize, status);
  }
  public boolean commit(MonitoredTask status) throws IOException {
    if (storeFilePath == null) return false;
    storeFile = Store.this.commitFile(storeFilePath, cacheFlushId, snapshotTimeRangeTracker, flushedSize, status);
    if (Store.this.getHRegion().getCoprocessorHost() != null)
      Store.this.getHRegion().getCoprocessorHost().postFlush(Store.this, storeFile);
    // Add new file to store files.  Clear snapshot too while we have the Store write lock.
    return Store.this.updateStorefiles(storeFile, snapshot);
  }
}
```

StoreFlusher 的 prepare()很简单，因为一个 Store 只有一个 MemStore，所以调用 MemStore.snapshot()并通过 getSnapshot()获取快照数据 snapshot. MemStore 的来源数据是 kvset，因此做快照时将 kvset 赋值给 snapshot.
除此之外，还要重置 MemStore 的 kvset, size, allocator 等. 这样 MemStore 中的数据就被清空，不再保存任何数据.

```
/**Creates a snapshot of the current memstore. Snapshot must be cleared by call to #clearSnapshot(SortedSet<KeyValue>) */
void snapshot() {
  this.lock.writeLock().lock();
  try {
    // If snapshot currently has entries, then flusher failed or didn't callcleanup. Log a warning. 快照时必须保证snapshot为空
    if (this.snapshot.isEmpty()) {
      if (!this.kvset.isEmpty()) {  // 当然MemStore的内容不能为空. 即快照的来源不为空，快照的目标必须为空
        this.snapshot = this.kvset;
        this.kvset = new KeyValueSkipListSet(this.comparator);  // 重置MemStore用于保存客户端数据的kvset
        this.snapshotTimeRangeTracker = this.timeRangeTracker;
```

```
            this.timeRangeTracker = new TimeRangeTracker();
            this.size.set(DEEP_OVERHEAD);         // Reset heap to not include any keys  不包括任何keyvalue
            if (allocator != null) this.allocator = new MemStoreLAB(conf); // Reset allocator so we get a fresh buffer for the new memstore
            timeOfOldestEdit = Long.MAX_VALUE;
          }
        }
      } finally {
        this.lock.writeLock().unlock();
      }
    }
```

采用快照的原因是刷新内存时要使用 MemStore 的内存数据，而刷新内存时仍然会有客户端的更新进来．在刷新内存时又要保证要操作的数据的不变性(否则数据是变的,刷新完这部分数据,还要再处理新来的数据显然是不现实的).因此对某一时刻 MemStore 中的内存数据进行快照，刷新内存只对这部分快照的数据进行刷新操作．同时清空 MemStore 已有的数据，客户端如果还有更新数据过来，尽管写入到已经被清空的 MemStore 中．对刷新内存的那份快照数据并不会有任何影响． 这样就确保了客户端可以继续写入更新数据，而刷新内存也仍然可以照样进行.

现在我们来看看前面的问题就很容易理解了：刷新内存在做快照时，不允许进行更新操作.
假设对 Region 的 MemStore 进行快照没有加上 writeLock，也就是允许其他线程的写和读.
举例：线程 A 要对 MemStore 进行快照，B 也要对相同的 Region 的 MemStore 快照，C 则进行更新操作.
A & C: A 做快照时会重置 kvset，然后 C 更新，往 kvset 中添加记录，A 又重置 size 不包括任何 KV，这样快照后的结果和 C 产生的 kvset 就产生了矛盾：kvset 中有 KV 数据，但是 size 却不包含任何的 KV.
A & B: A 正在做快照,进入 MemStore.snapshot()，还没为 snapshot 赋值，B 线程却抢先完成了，并重置 kvset 为空，当 A 为 snapshot 赋值时，得到的 kvset 为空，因为 B 线程已经做完快照了.

所以在对 MemStore 进行快照时，要加上写锁，不允许任何别的写线程和读线程操作．你可能会说那么读操作不是会阻塞吗？其实没关系的，因为做快照的过程是很快的，只有几个赋值操作，没有别的业务逻辑处理．而简单地赋值操作速度是很快的，只是内存之间引用的赋值．所以做快照时阻塞更新操作是可以接受的.

我们现在知道了 ReentrantReadWriteLock 的 writeLock 会阻塞其他任意线程的 readLock 和 writeLock.
那么如果 readLock 正在运行，writeLock 发生时可以继续进行，readLock 必须等待 writeLock 释放完才能继续执行.

| | readLock | writeLock |
|---|---|---|
| readLock | √ | √ |
| writeLock | × | × |

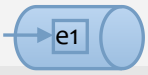
上表第一列为首先占用的锁，读锁发生时，其他读锁也可以执行，如果此时有写锁，也可以进行，不过其他读写锁这时候都会被阻塞，直到那个写锁释放为止． 即写锁发生时，会阻塞任何的读锁和写锁(第二行).

在对 MemStore 进行快照时，加上 writeLock 不允许读写更新． 但如果是更新正在进行(readLock)，然后对 MemStore 进行快照，快照会阻塞正在进行的更新：读锁发生时，写锁也可以进行，但是读锁此时会被阻塞住，为了防止正在进行的更新未提交就被刷新，在做快照后，要等待快照时刻正在进行的更新提交事务后才进行内存刷新.

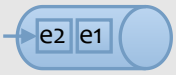
```
String s = "Finished snapshotting, commencing wait for mvcc, flushsize=" + flushsize;
status.setStatus(s);
if (wal != null && isDeferredLogSyncEnabled()) wal.sync();    // sync unflushed WAL changes when deferred log sync is enabled

// wait for all in-progress transactions to commit to HLog before we can start the flush. 在刷新前等待正在进行的事务提交到HLog
// This prevents uncommitted transactions from being written into HFiles. 防止未提交的事务写入HFile
// We have to block before we start the flush, otherwise keys that  在开始刷新前阻塞,否则通过rollbackMemstore
// were removed via a rollbackMemstore could be written to Hfiles. 移除了，但却还是会写入HFile.
mvcc.waitForRead(w);
status.setStatus("Flushing stores");
```

举例：线程 A 对 MemStore 的 kvset 添加了一个 KV，但是还没提交:未执行 MVCC.completeMemstoreInsert()更新 memstoreRead. 然后线程 B 对 MemStore 的 kvset 做了快照，快照里包含了 A 刚刚添加却未提交的 KV. 要刷新内存时，显然必须是对 MemStore 上已经提交的事务做快照，未提交的事务不能进行快照.

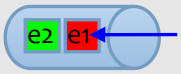
A: put(kv) → **beginMemStoreInsert**()
memstoreRead=0, memstoreWrite=1 writeQueue.add(E1[writeNumber=1])




B: cacheflush → **beginMemStoreInsert**()
memstoreRead=0, memstoreWrite=2 writeQueue.add(E2[writeNumber=2])

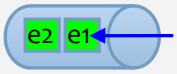



→ **advanceMemstore**(E2) E2 completed!
queueFirst = E1, but not complete yet! memstoreRead=0 still!

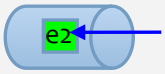



→ **waitForRead**(E2) memstoreRead=0<E2.writeNumber=2, so wait in while loop (>_<)

A: put(kv) → **completeMemstoreInsert**(E1)
1. queueFirst=E1, E1 complete now, memstoreRead=E1.writeNumber=1




notify readWaiters
2. queueFirst=E2, E2 already complete, memstoreRead=E2.writeNumber=2
waitForRead(E1) memstoreRead=2 >= E1.writeNumber=1, do not need to wait!



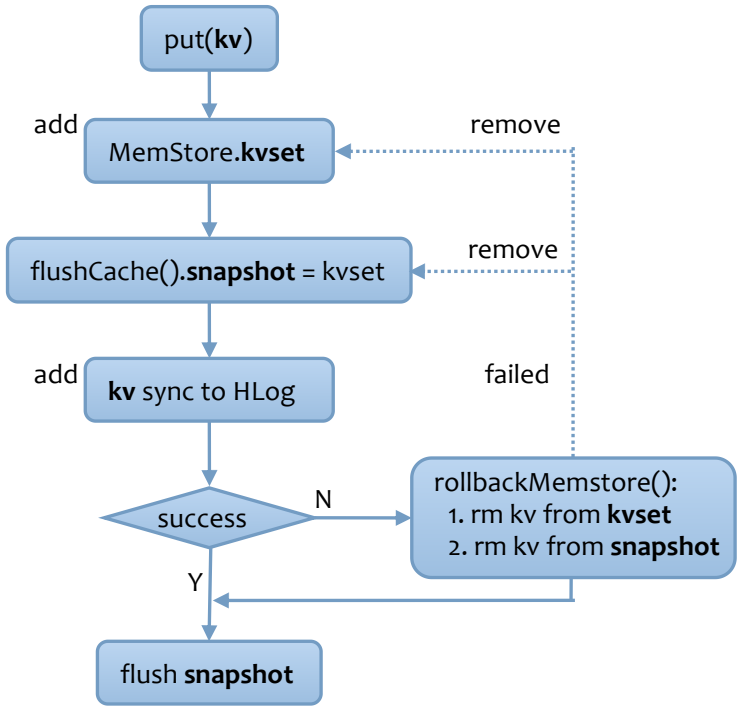

notify readWaiters again


B: cacheflush → **waitForRead**(E2) was notifyed...
1. memstoreRead=1 < E2.writeNumber=2, still go to wait
2. memstoreRead=2>=E2.writeNumber=2, Wow, I should't wait any more! (∩_∩)

在刷新内存对 MemStore 做快照时，如果有正在更新的进程 put(kv)，则做快照完毕后，要等待那个正在更新的进程提交到 HLog 后，才可以刷新内存. 假设正在更新的进程 put(kv)已经到了 MemStore.kvset，那么做快照时，这个 kv 也会被包含到 snapshot 中，之后会被刷新到 HFile 中. 更新数据 updates data 存在 MemStore 中，还必须保存一份到 HLog 中. 必须等到 put(kv)提交后，才可以刷新内存. 如果没有这个过程，刷新内存继续进行，会把 kv 也 flush 到 HFile 中. 但是 put(kv)继续进行的时候如果出现了同步失败 rollbackMemstore，则 kv 不会出现在 HLog 中. 但却已经被刷新到了 HFile 中. 造成了数据的不一致性：更新数据必须在 MemStore(或者刷新到 HFile)以及 HLog 中保持一致.

```
                              |----------------> MemStore ------------------------> StoreFile(HFile)
updates data (KeyValue) ------|                              flush
                              |----------------> HLog
```

下图表示 Put 更新记录时和刷新缓存做快照并行执行的一个过程:



什么时候同步成功:HRegion.doMiniMutation()的 walSyncSuccessful 变量:在同步到 WAL(HLog)完成后才算同步成功.
如果同步失败，说明更新数据写入到 HLog 中失败，由于数据一致性的限制，更新数据也不应该存在于 MemStore 中.

```
// if the wal sync was unsuccessful, remove keys from memstore
if (!walSyncSuccessful) {
  rollbackMemstore(batchOp, familyMaps, firstIndex, lastIndexExclusive);
}
```

从方法名字可以看出回滚 MemStore:数据写入 MemStore,但是写入 HLog 失败了,那么 MemStore 里的数据需要回滚:

```
/**Remove all the keys listed in the map from the memstore. This method is called when a Put/Delete has updated memstore
 * but subequently fails to update the wal. This method is then invoked to rollback the memstore. */
private void rollbackMemstore(BatchOperationInProgress<Pair<Mutation, Integer>> batchOp,
    Map<byte[], List<KeyValue>>[] familyMaps, int start, int end) {
  for (int i = start; i < end; i++) {
    // skip over request that never succeeded in the first place.  如果更新数据本身就没成功过，说明本身就没在MemStore中
    if (batchOp.retCodeDetails[i].getOperationStatusCode() != OperationStatusCode.SUCCESS) continue;
    // Rollback all the kvs for this row.
    Map<byte[], List<KeyValue>> familyMap = familyMaps[i];
    for (Map.Entry<byte[], List<KeyValue>> e : familyMap.entrySet()) {
      byte[] family = e.getKey();
      List<KeyValue> edits = e.getValue();
      // Remove those keys from the memstore that matches our key's (row, cf, cq, timestamp, memstoreTS).
      // The interesting part is that even the memstoreTS has to match for keys that will be rolleded-back.
      Store store = getStore(family);
      for (KeyValue kv: edits) {
        store.rollback(kv);
      }
    }
  }
}
```

回滚 MemStore 时，也会将 KeyValue 从 snapshot 中移除，所以这个操作应该发生在 flushCache(snapshot)之前:
通过在快照之后 mvcc.waitForRead()确保了如果发生错误回滚 MemStore 时, snapshot 和内存中的数据都是一致的.

```
void rollback(final KeyValue kv) {
  this.lock.readLock().lock();
  try {
    // If the key is in the snapshot, delete it.    The flush of this snapshot to disk has not yet started
    // because Store.flush() waits for all rwcc transactions to commit before starting the flush to disk.
    KeyValue found = this.snapshot.get(kv);
    if (found != null && found.getMemstoreTS() == kv.getMemstoreTS())
      this.snapshot.remove(kv);    //  从快照中也移除写入HLog失败的kv
    found = this.kvset.get(kv);       // If the key is in the memstore, delete it. Update this.size.
    if (found != null && found.getMemstoreTS() == kv.getMemstoreTS()) {
      removeFromKVSet(kv);
      long s = heapSizeChange(kv, true);
      this.size.addAndGet(-s);
    }
  } finally {
    this.lock.readLock().unlock();
  }
}
```

MemStore 刷新到 StoreFile 到此为止，接着 HRegion.doMiniMutation()接下来的流程. 是不是忘了出口在哪里了?

## +step6-9: release resouce

```
// step6: release row locks
if (locked) {
  this.updatesLock.readLock().unlock();
  locked = false;
}
if (acquiredLocks != null) {
  for (Integer toRelease : acquiredLocks) releaseRowLock(toRelease);
  acquiredLocks = null;
  rowsAlreadyLocked = null;
}

// step7: sync wal
if (walEdit.size() > 0)   syncOrDefer(txid, durability);
walSyncSuccessful = true;       // 同步到HLog完成，设置标记. 说明数据在MemStore和HLog中都有记录. 不需要回滚内存
if (coprocessorHost != null) {   // calling the post CP hook for batch mutation
  MiniBatchOperationInProgress<Pair<Mutation, Integer>> miniBatchOp = new MiniBatchOperationInProgress(
    batchOp.operations, batchOp.retCodeDetails, batchOp.walEditsFromCoprocessors, firstIndex, lastIndexExclusive);
  coprocessorHost.postBatchMutate(miniBatchOp);
}

// STEP 8. Advance mvcc. This will make this put visible to scanners and getters.
if (w != null) {
  mvcc.completeMemstoreInsert(w);
  w = null;
}

// STEP 9. Run coprocessor post hooks. This should be done after the wal is synced so that the coprocessor contract is adhered to.
if (coprocessorHost != null) {
  for (int i = firstIndex; i < lastIndexExclusive; i++) {
    if (batchOp.retCodeDetails[i].getOperationStatusCode() != OperationStatusCode.SUCCESS) continue; // only for successful
    Mutation m = batchOp.operations[i].getFirst();
    if (m instanceof Put) coprocessorHost.postPut((Put) m, walEdit, m.getWriteToWAL());
    else coprocessorHost.postDelete((Delete) m, walEdit, m.getWriteToWAL());
  }
}
success = true;
```

s6: acquiredLocks 是第一步获取到的行锁集合.释放某个行锁后,在该行上等待获取锁的其他线程就可以获取到行锁.在释放行锁前,也释放了 updatesLock 读锁.即 updatesLock 读取锁的 lock 和 unlick 之间的部分会被 writeLock 阻塞.

s7/9: 协处理器贯穿在正常的业务流程中，类似于拦截器的作用定义在各个事件前后:
batchMutate() → prePut() → ... → preBatchMutate() → ... → postBatchMutate() → ... → postPut() → END

最后在 finally 里设置下一次批处理的开始位置为本次批处理的 lastIndexExclusive: 因为客户端过来的一次完整的批处理可能会被分成多次 mini 的批处理(如果锁无法获取到，且不是本次批处理的第一条记录就等待下次批处理).

```
batchOp.nextIndexToProcess = lastIndexExclusive;
```

Wow, finally, Then Where shall we go NEXT?