Cassandra介绍

畅享互联——李明建

2013-01-09

Cassandra介绍

Apache Cassandra 是一套开源分布式 Key-Value 存储系统。Cassandra 不是一个数据库，它是一个混合型的非关系的数据库。它以Amazon专有的完全分布式的Dynamo 为基础，结合了Google BigTable 基于列族(Column Family)的数据模型。

Cassandra介绍

- 特点
- 数据模型
- Cassandra Query Language
- 内部数据存储结构
- 集群机制
- Cassandra Client——Hector
- 集成Hadoop

Cassandra特点

- 1. 灵活的schema

　　不需要象数据库一样预先设计schema，增加或者删除字段非常方便。

- 2. 支持range 查询

　　可以对Key 进行范围查询。

- 3. 高可用，可扩展

　　单点故障不影响集群服务，可线性扩展。

数据模型

Cassandra 的数据模型是基于列族（Column Family）的四维或五维模型。它借鉴了 Amazon 的 Dynamo 和 Google's BigTable 的数据结构和功能特点，采用 Memtable 和 SSTable 的方式进行存储。在 Cassandra 写入数据之前，需要先记录日志 (CommitLog)，然后数据开始写入到 Column Family 对应的 Memtable 中，Memtable 是一种按照 key 排序数据的内存结构，在满足一定条件时，再把 Memtable 的数据批量的刷新到磁盘上，存储为 SSTable 。

数据模型

Cassandra的数据模型与传统的键值对类型不同，我们可以将Cassandra的数据模型想象成一个四维或者五维的HashMap。

在Cassandra中，数据类型有以下几种：Column，SuperColumn，ColumnFamily和Keyspace。

数据模型

- Column

column是Cassandra中最小的数据单元。它是一个三元的数据类型，包含name，value， timestamp。将一个column用JSON的形式表现出现，如下所示：

```
{//这是一个Column
        name:”email”,
        value:”hello@joyveb.com”,
        timestamp:123456789
}
```

数据模型

- SuperColumn

我们可以将SuperColumn想象成Column的数组，它包含一个name以及一系列相应的Column。将一个SuperColumn用JSON的形式表现出现，如下所示：

```
{//这是一个SuperColumn
    name:"email",
    value:{
      {name:"address", value:"hello@joyveb.com",timestamp:12},
      {name:"id", value:"hello",timestamp:12},
      {name:"passwd", value:"123456",timestamp:12}
    }
}
```

数据模型

- SuperColumn

Column和SuperColumn都是name与value的组合。最大的不同在于Column的value是一个“string”，而SuperColumn的value是Columns的Map。

还有一点需要注意的是：SuperColumn本身是不包含timestamp的。

数据模型

- ColumnFamily

ColumnFamily是一个包含了许多Row的结构，你可以将它想象成RDBMS中的Table。Row是由Key以及和该Key关联的一系列Column组成的。如下所示：

```
UserProfile = {   // 这是一个ColumnFamily
key1: {     // 这是对应ColumnFamily的key
    //这是Key下对应的Column
    {username: “hello“,email: “hello@joyveb.com”,timestamp:1},
    {username: “hehe“,email: “hehe@joyveb.com",timestamp:1}
}, // 第1个row结束
key2: {  // 这是ColumnFamily的另一个key
    //这是另一个Key对应的column
    {username: “world“,email: “world@joyveb.com", timestamp:1}
} // 第2个row结束
}
```

数据模型

- ColumnFamily

上面的例子是一个Standard类型的ColumnFamily。包含了一系列的Column。ColumnFamily也可以是Super类型。

```
UserProfile = {   // 这是一个Super类型的ColumnFamily
  key1: {     // 这是对应ColumnFamily的key
    {
      name:"joyveb1"//这是SuperColumn对应的name
      value{
            {name:"email",value="hello@joyveb.com",timestamp:1},
            {name:"phone",value="13900000000",timestamp:1}
      }
    },
...
```

数据模型

- ColumnFamily

```
{
    name:"joyveb2"//这是SuperColumn对应的name
    value{
        {name:"email",value="joyveb2@joyveb.com",timestamp:1},
        {name:"phone",value="13900000002",timestamp:1}
    }
} // 第1个row结束
... ...
```

数据模型

- Keyspace

Keyspace是我们的数据最外层，你所有的ColumnFamily都属于某一个Keyspace。一般来说，我们的一个程序应用只会有一个Keyspace。有点类似于RDBMS的数据库。

Cassandra Query Language

Cassandra内部数据存储结构

Cassandra数据信息共分为3类：

- data目录：用于存储真正的数据文件，是SSTable文件，可以指定多个目录。
- commitlog目录：用于存储未写入SSTable中的数据。
- cache目录：用于存储系统中的缓存数据。（在服务重启的时候从这个目录中加载缓存数据）

合理安排上面节点之间的位置以提高性能。

Commitlog

Commilog中包括两个部分：Commilog-XXXX.log和Commilog-XXXX.log.header。Commitlog-XXXX.log文件中保存上次更新操作的值，Commitlog-XXXX.log.header记录了那些数据已经从Memtable中写入SSTable中。

Cassandra有“周期”和“批量”两种方式记录Commitlog的方式。当一个Commitlog文件的大小超过一定阀值的时候就会新建一个Commitlog。Cassandra每次更新信息将写入Commitlog中，并且每隔一定时间间隔同步文件（还要同步缓存中的数据，使其全部保存在Commitlog中）。

Memtable

数据写入Commitlog中后会缓存在Memtable中（每个 Memtable为一个ColumnFamily服务）。在Memtable中容量、条数、时间间隔超过阀值后就会将数据写入磁盘形成一个SSTable文件。Memtable中保存的数据是ConcurrentSkipListMap<DecoratedKey, ColumnFamily>。写入时按照Key的顺序。使用Memtable的优势就是把随机的IO变成了顺序IO，降低大量写操作对存储系统的压力。

SSTable

当Memtable中缓存的某一个ColumnFamily中的数据量(大小和数量)或者超过上一次生成SSTable的时间后，Cassandra会将Memtable中对应的ColumnFamily的数据持久化到磁盘中，生成一个SSTable文件。

SSTable

Cf1的一个SSTable文件由如下文件组成：

- Cf1-hf-1-Data.db
- Cf1-hf-1-Filter.db
- Cf1-hf-1-Index.db
- Cf1-hf-1-Statistics.db

SSTable

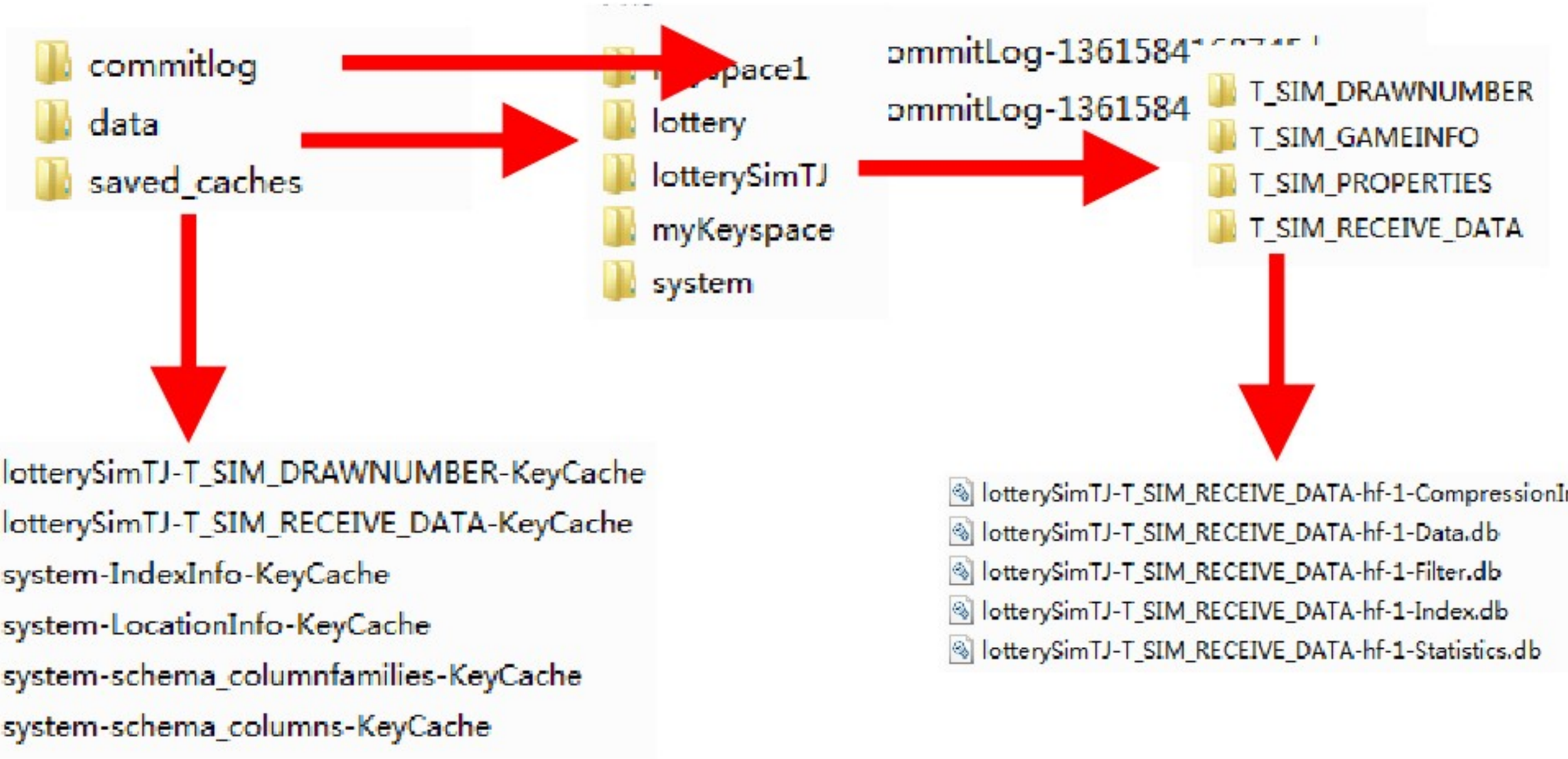
- Data文件存储数据与Key对应的一些Column的索引信息（利用索引快速地找到要找的值）。
- Filter文件的作用是快速定位某一个Key是否在该SSTable文件中（用布尔过滤器来做判断）。
- Index文件用于索引文件，保存Key和在Data文件中对应的位置。在内存中查找的时候先哈希再二分查找。
- Statistics文件用来存储SSTable中所包含的Column的个数和Row的个数。

系统表空间

在Cassandra中除了用户自己定义的Keyspace之外还有一个特殊的Keyspace：system。

- 管理Cassandra的系统元数据信息。
- 缓存HINT数据。

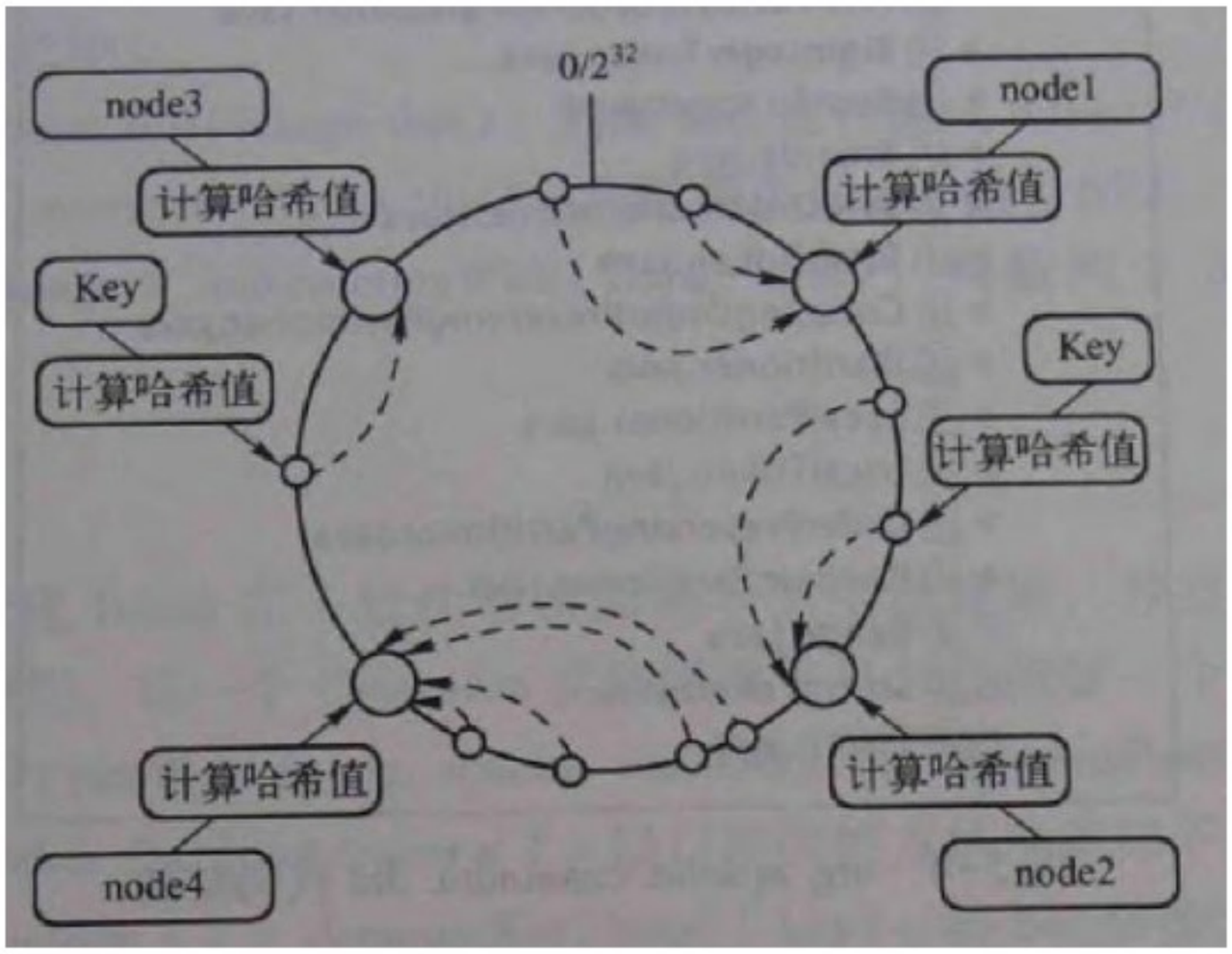
Cassandra内部数据存储结构

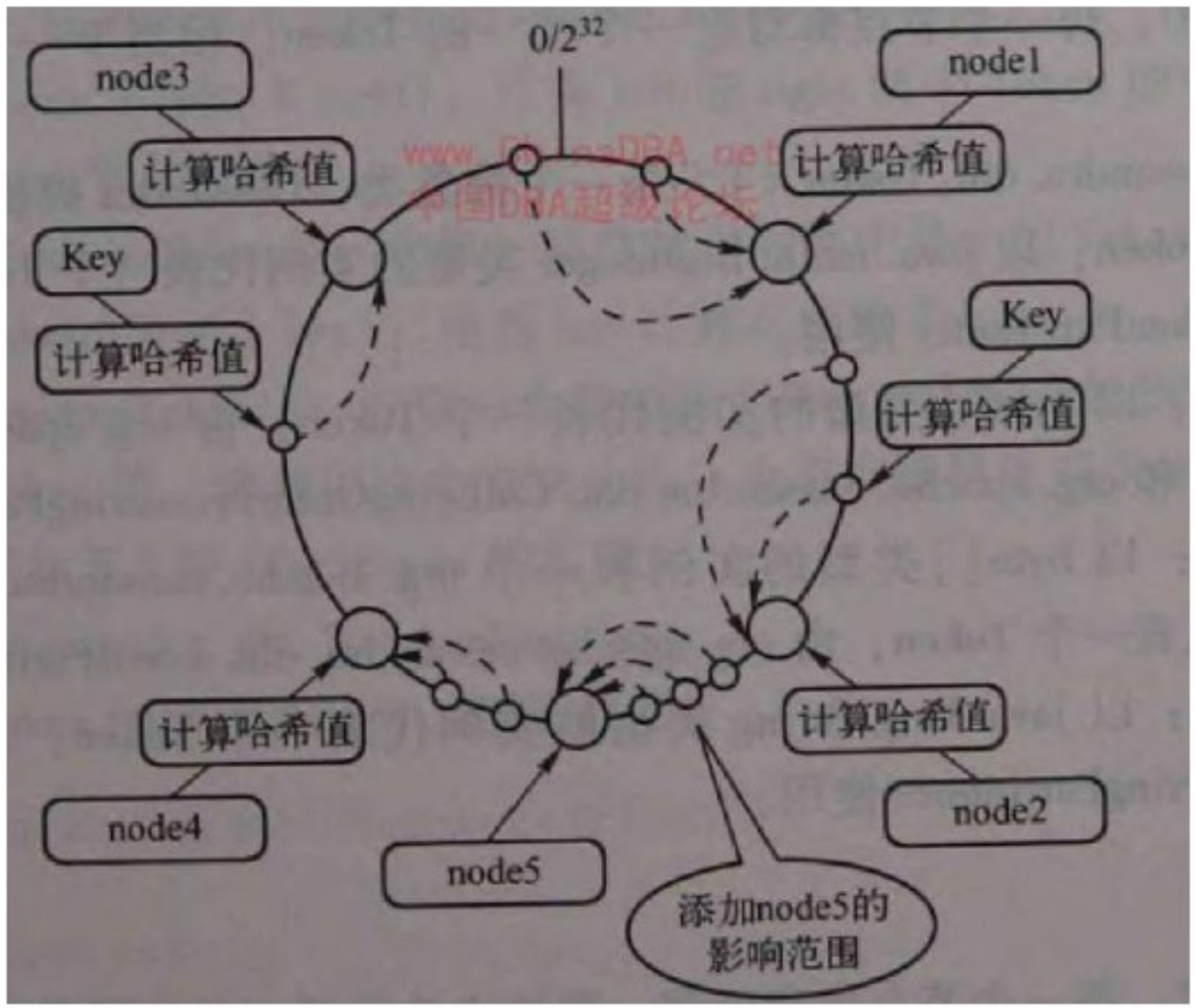


集群机制

- 一致性哈希
- Gossip
- 集群节点之间的通讯协议
- 集群的数据备份机制

集群机制——一致性哈希

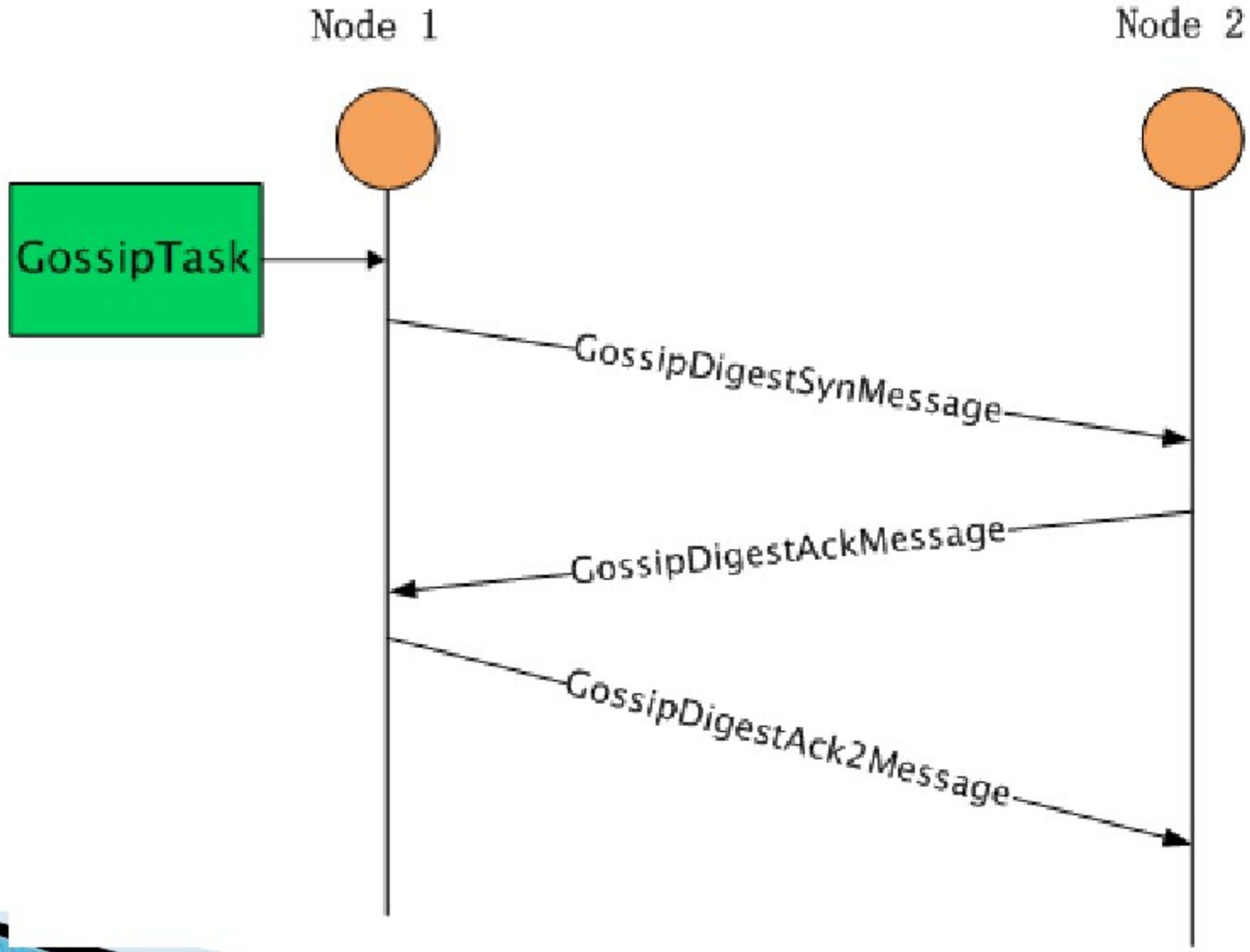

0/2^32
node3
node1
计算哈希值
计算哈希值
Key
计算哈希值
Key
计算哈希值
计算哈希值
计算哈希值
node4
node2


0/2^32
node3
node1
计算哈希值
计算哈希值
Key
Key
计算哈希值
计算哈希值
计算哈希值
计算哈希值
node4
node2
node5
添加node5的
影响范围

集群机制——Gossip

Cassandra集群没有中心节点，各个节点的地位完全相同，节点之间通过Gossip协议进行通信，用于维护集群的状态。通过Gossip，每个节点都能知道集群中包含哪些节点，以及每一个节点的状态。这使得Cassandra集群中的任何一个节点都可以完成任意读取和写入操作，若任意一个节点失效，整个集群依旧正常工作。

集群机制——Gossip工作原理



集群机制——数据备份机制

Cassandra是一个支持容灾的系统，即数据会在集群中保留多份，这样当某一个机器失效的时候，其他机器仍然有数据备份，从而保证整个服务正常。

Cassandra主要包括两个部分的实现：EndpointSnitch(机架感应)与ReplicationStrategy(数据的备份策略)。

集群机制——数据备份机制

- EndpointSnitch

通过机架感应，Cassandra集群中的每一个节点都可以知道哪几台节点和自己属于一个机架，哪几台节点和自己属于一个数据中心。

  - **SimpleSnitch**：不提供机架和数据中心功能，对节点距离排序
  - **PropertyFileSnitch**：通过配置文件制定对应的数据中心和机架名称。
  - **RackInferringSnitch**：与PropertyFileSnitch类似，又有节点距离排序规则
  - **DynamicEndpointSnitch**：不能单独使用，必须与前面3中搭配使用

集群机制——数据备份机制

- ReplicationStrategy

通过ReplicationStrategy，Cassandra集群可以知道任意一份数据备份的节点信息，同时在节点失效的时候，还能够计算出应该接收HINT消息的节点。

- **SimpleStrategy**：它根据指定的Token在指定的一致性哈希圆环中按照顺时针方向找出下N个需要备份的节点。
- **OldNetworkTopologyStrategy**：与SimpleStrategy类似。不同的是它在寻找第二个备份节点的时候，会找一个与第一个备份节点不在同一个数据中心的节点进行备份；寻找第三个备份节点的时候，会找一个与第二个备份节点同数据中心，但是不同机架的节点备份；接下来所有的备份节点寻找策略就按照SimpleStrategy的备份策略寻找。
- **NetworkTopologyStrategy**：它在OldNetworkTopologyStrategy的基础上，可以更加详细地指定每一个数据中心需要备份的数据份数。

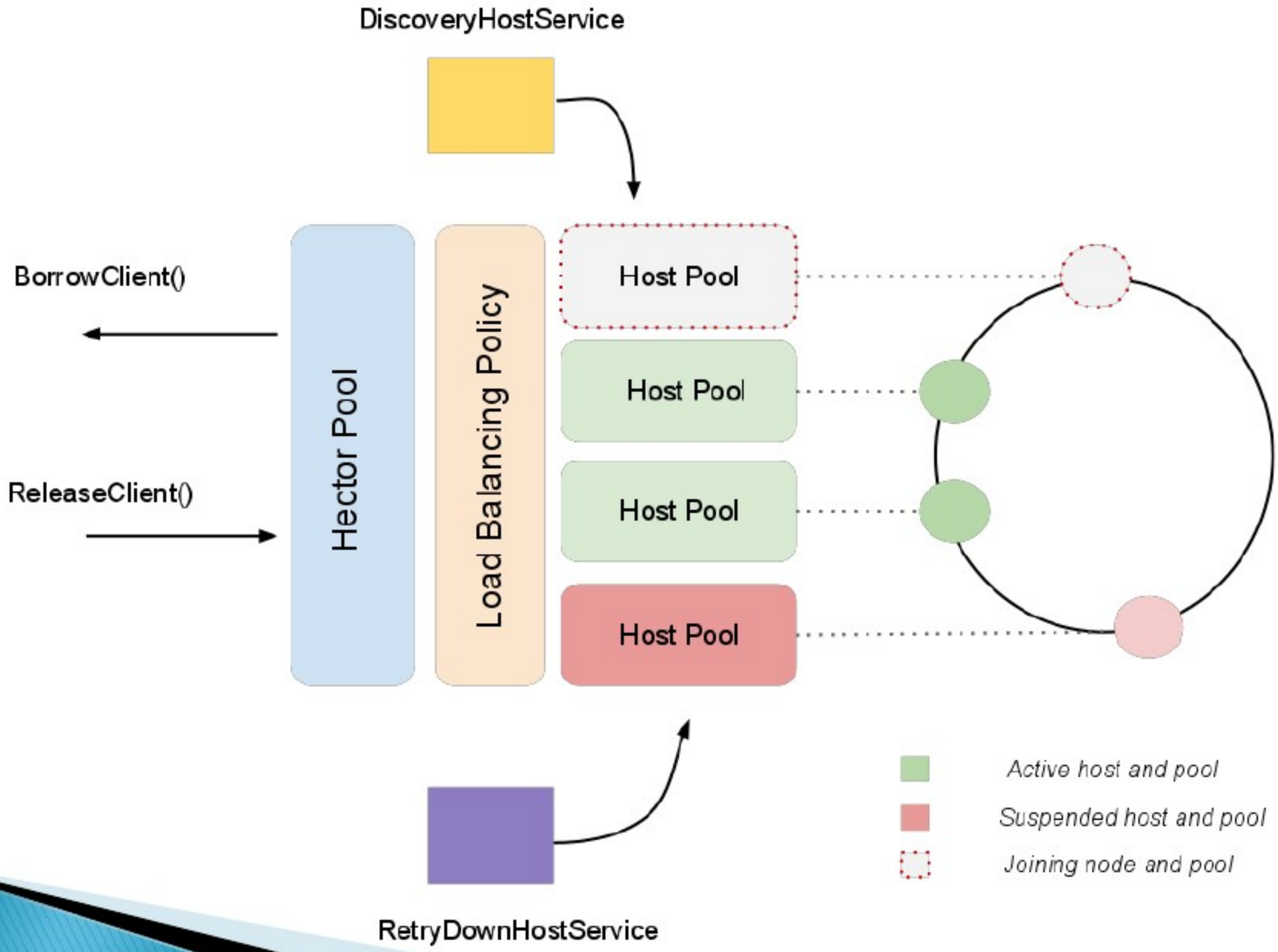
回顾

- 数据模型
- 内部数据存储结构
- 集群机制

Cassandra Client——Hector

- 特性
  - 对cassandra高层的、简单的面向对象接口
  - 客户端的故障转移行为
  - 提高性能和可扩展性的连接池
  - 用于监视和管理JMX计数器
  - 可配置和可扩展的负载均衡
  - 底层Thrift API和结构的完全封装
  - 自动重试下线的主机
  - 集群中自动发现其他主机
  - 多次超时后短时间内挂起主机
  - 简单的ORM层
  - 一个类型安全的方法处理Apache Cassandra的数据模型

Hector连接池架构



Hector实例——创建keyspace

```
protected Keyspace getOrCreateKeyspace(String keyspace) {
    boolean exists = false;
    List<KeyspaceDefinition> ksList = cluster.describeKeyspaces();
    for (KeyspaceDefinition ks : ksList) {
        if (keyspace.equals(ks.getName())) {
            exists = true;
            break;
        }
    }
    if (!exists) {
        KeyspaceDefinition ks_def = HFactory.createKeyspaceDefinition(
                keyspace, "org.apache.cassandra.locator.SimpleStrategy",
                cluster.getConnectionManager().getActivePools().size(),
                new ArrayList<ColumnFamilyDefinition>());
        cluster.addKeyspace(ks_def, true);
    }
    // Create a customized Consistency Level
    ConfigurableConsistencyLevel configurableConsistencyLevel = new ConfigurableConsistencyLevel();
    Map<String, HConsistencyLevel> clmap = new HashMap<String, HConsistencyLevel>();
    // Define CL.ONE for ColumnFamily "MyColumnFamily"
    clmap.put("cf_drawnumber", HConsistencyLevel.ONE);
    // In this we use CL.ONE for read and writes. But you can use different CLs if needed.
    configurableConsistencyLevel.setReadCfConsistencyLevels(clmap);
    configurableConsistencyLevel.setWriteCfConsistencyLevels(clmap);
    return HFactory.createKeyspace(keyspace, cluster, configurableConsistencyLevel);
```

Hector实例——批次插入

```
public void batchInsert(final List<DrawNumber> list){
    Mutator<String> mutator = HFactory.createMutator(ksp, stringSerializer);
    for(DrawNumber drawNumber : list){
        String key = drawNumber.getLtype()+"-"+drawNumber.getPeriod();
        mutator.addInsertion(key, columnFamily, HFactory.createStringColumn("ltype", drawNumber.getLtype()))
            .addInsertion(key, columnFamily, HFactory.createStringColumn("period", drawNumber.getPeriod()))
            .addInsertion(key, columnFamily, HFactory.createColumn("processstatus", drawNumber.getProcessstat
            .addInsertion(key, columnFamily, HFactory.createColumn("salestatus", drawNumber.getSalestatus(),s
            .addInsertion(key, columnFamily, HFactory.createStringColumn("region", drawNumber.getRegion()))
            .addInsertion(key, columnFamily, HFactory.createColumn("date",drawNumber.getDate(), stringSeriali
    }
    template.executeBatch(mutator);
}
```

Hector实例——更新数据

```
public void update(DrawNumber drawNumber){
    String key = drawNumber.getLtype()+"-"+drawNumber.getPeriod();
    ColumnFamilyUpdater<String, String> updater = template.createUpdater(key);
    updater.setString("ltype", drawNumber.getLtype());
    updater.setString("period", drawNumber.getPeriod());
    updater.setLong("processstatus", drawNumber.getProcessstatus());
    updater.setInteger("salestatus", drawNumber.getSalestatus());
    updater.setString("region", drawNumber.getRegion());
    updater.setDate("date", drawNumber.getDate());
    template.update(updater);
}
```

Hector实例——删除数据

```
public void delete(String key){
    template.deleteRow(key);
}
```

Hector实例——简单条件查询1

```
public List<DrawNumber> selectByProcessstatus(long processstatus){
    IndexedSlicesPredicate<String, String, Long> processP =
            new IndexedSlicesPredicate<String, String, Long>(stringSerializer, stringSeriali:
    processP.addExpression("processstatus", IndexOperator.EQ, processstatus);
    ColumnFamilyResult<String, String> result = template.queryColumns(processP);
    List<DrawNumber> drawNumbers = new ArrayList<DrawNumber>();
    if(result.hasResults()){
        DrawNumber drawNumber = cfResult2Pojo(result);
        drawNumbers.add(drawNumber);
        System.out.println(drawNumber);
        while(result.hasNext()){
            result.next();
            drawNumber = cfResult2Pojo(result);
            drawNumbers.add(drawNumber);
            System.out.println(drawNumber);
        }
    }
    return drawNumbers;
}
```

Hector实例——简单条件查询2

```
public List<T> selectByCQL(String cql) {
    long start = System.currentTimeMillis();
    Assert.notNull(cql, "cql is null");
    CqlQuery<String, String, String> cqlQuery = new CqlQuery<String, String, String>(
            this.keyspace, stringSerializer, stringSerializer,
            stringSerializer);
    cqlQuery.setQuery(cql);
    QueryResult<CqlRows<String, String, String>> result = cqlQuery
            .execute();
    List<T> datas = null;
    if (result.get() != null) {
        datas = new ArrayList<T>();
        List<Row<String, String, String>> rows = result.get().getList();
        for (Row<String, String, String> row : rows) {
            T data = row2POJO(row);
            datas.add(data);
        }
    }
    log.debug(this.columnFamilyName + ": cql[" + cql + "], cost:"
            + (System.currentTimeMillis() - start) + "ms");
    return datas;
}
```

Hector实例——简单条件查询3

```
public List<DrawNumber> selectRangeBySalestatus(int start, int end, int pagesize, String startKey, String endKey) {
    RangeSlicesQuery<String, String, Integer> rangeSlicesQuery =
            HFactory.createRangeSlicesQuery(ksp, stringSerializer, stringSerializer, IntegerSerializer.get());
    rangeSlicesQuery.addGtExpression("salestatus", start);
    rangeSlicesQuery.addLtExpression("salestatus", end);
    rangeSlicesQuery.setColumnNames("salestatus");
    rangeSlicesQuery.setColumnFamily(columnFamily);
    rangeSlicesQuery.setKeys(startKey, endKey);
    rangeSlicesQuery.setRowCount(pagesize);
    QueryResult<OrderedRows<String, String, Integer>> resultQuery = rangeSlicesQuery.execute();
    List<Row<String, String, Integer>> rows = resultQuery.get().getList();
    List<String> keys = new ArrayList<String>();
    for(Row<String, String, Integer> row : rows){
        keys.add(row.getKey());
    }
    ColumnFamilyResult<String, String> result = template.queryColumns(keys);
    return cfResult2List(result);
}
```

Hector在数据一致性的问题和实践

- Hector不支持事务
- 操作数据时抛出HTimedOutException、HUnavailableException等异常，但并不等于操作不成功

Hector一致性级别

- Hector提供不同的一致性级别
  - **ANY**:：等待直到有任何节点响应.
  - **ONE**：等待直到1个节点响应.
  - **TWO**：等待直到2个节点响应.
  - **THREE**：等待直到3个节点响应.
  - **LOCAL_QUORUM**：等待数据中心法定个数的连接建立.
  - **EACH_QUORUM**：等待每个数据中心法定个数的连接建立.
  - **QUORUM**：等待法定个数节点响应 (不管哪个数据中心).
  - **ALL**：等待所有节点响应.
- Hector可以设置每个*Column Family* 和每一个操作类型(读写)的一致性级别。

End.