**Cassandra**

## 一、Cassandra框架

**1 Cassandra是社交网络理想的数据库，适合于实时事务处理和提供交互型数据。**以Amazon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，P2P去中心化的存储，目前twitter和digg中都有使用。Apache Cassandra 是一种分布式非关系型数据库，具有高性能、可扩展、无中心化等特征。

Cassandra 是适用于社交网络业务场景的数据库，适合实时事务处理和提供交互型数据。

以 Amazon 完全分布式的 Dynamo 数据库作为基础，结合 Google BigTable 基于列族（Column Family）的数据模型，实现 P2P 去中心化的存储。

**2 在CAP特性上，HBase选择了CP，Cassandra更倾向于AP，而在一致性上有所减弱。**

Cassandra最初源自Facebook，结合了Google BigTable面向列的特性和[Amazon Dynamo](http://en.wikipedia.org/wiki/Dynamo(storagesystem) ) 分布式哈希（DHT）的P2P特性于一身，具有很高的性能、可扩展性、容错、部署简单等特点。

它虽然有多的优点，但国内使用的公司貌似不多，远没有Hbase和MongoDB火，从百度指数上可以明显看到这三个系统在国内的热度对比。相对国内冷静的市场来说，Cassandra在国外发展的倒是如火如荼，国外这个专门对数据库进行评分的网站DB-Engines显示Cassandra排进了前十名，比Hbase的名次要高好几位，从2013年开始有了突飞猛进的增长，目前已有超过1500多家公司在使用Cassandra，可惜的是没有多少国内公司，只有一家做云存储的创业公司云诺名列榜单。这也证明了网上的中文资源为何相对匮乏，我不得不找英文资料来看，倒是顺便加强了我的英文阅读能力，也算是失之东隅得之桑榆。

**CAP原则又称CAP定理，指的是在一个分布式系统中，**

**Consistency（一致性）、**

**Availability（可用性）、**

**Partition tolerance（分区容错性），**

**三者不可兼得。**

**CAP原理指的是，CAP的三个要素最多只能同时实现两点，不可能三者兼顾。因此在进行分布式架构设计时，必须做出取舍。而对于分布式数据系统，分区容忍性是基本要求，否则就失去了价值。**

**因此设计分布式数据系统，就是在一致性和可用性之间取一个平衡。对于大多数web应用，其实并不需要强一致性，因此牺牲一致性而换取高可用性，是目前多数分布式数据库产品的方向。**

**当然，并不是完全不管数据的一致性。牺牲一致性，只是不再要求关系型数据库中的强一致性，而是只要系统能达到最终一致性即可，考虑到客户体验，这个最终一致的时间窗口，要尽可能的对用户透明，也就是需要保障“用户感知到的一致性”。通常是通过数据的多份异步复制来实现系统的高可用和数据的最终一致性的，“用户感知到的一致性”的时间窗口则取决于数据复制到一致状态的时间。**

**3 Cassandra的类Dynamo特性有以下几点**

（1）对称的，P2P架构

n 无特殊节点，无单点故障

（2）基于Gossip的分布式管理

（3）通过分布式hash表放置数据

n 可插拔的分区

n 可插拔的拓扑发现

n 可配置的放置策略

1. 可配置的，最终一致性

**4 类BigTable特性**

（1）列族数据模型

n 可配置，2级maps，Super Colum Family

（2）SSTable磁盘存储

n Append-only commit log

n Mentable (buffer and sort)

n 不可修改的SSTable文件

（3）集成Hadoop

## 二、核心技术

**1 高级特性**

**（1）极高的读写性能**

Cassandra写数据时，首先会将请求写入Commit Log以确保数据不会丢失，然后再写入内存中的Memtable，超过内存容量后再将内存中的数据刷到磁盘的SSTable，并定期异步对SSTable做数据合并(Compaction)以减少数据读取时的查询时间。因为写入操作只涉及到顺序写入和内存操作，因此有非常高的写入性能。而进行读操作时，Cassandra支持像LevelDB一样的实现机制，数据分层存储，将热点数据放在Memtable和相对小的SSTable中，所以能实现非常高的读性能。

**（2）简单的部署结构**

相对Hbase等的主从结构，Cassandra是去中心化的P2P结构，所有节点完全一样没有单点，对小公司来说，我完全可以选择数据复制份数为2，先用两三台机器把Cassandra搭起来，既能保证数据的可靠性也方便今后机器的扩展，而Hbase起码得四五台机器吧。以后为了更好地支持客户可能需要在多个地方建立数据中心，而Cassandra对多数据中心的支持也很好，可以方便地部署多个数据中心，今早还看到一个俄罗斯最大电信公司的案例。另外我们的机器现在托管在一个小机房里，万一到时机器满了无法增加要考虑搬迁机房时，**使用多数据中心的方式也能做到无缝迁移。**

**（3）和Spark的结合**

Cassandra作为一个底层的存储系统，能够方便地和Spark集成进行实时计算，这一点对我们的业务场景有致命的吸引力，我看到国外有很多使用案例就是用Spark+Cassandra来实现Velocity计算，比如Ooyala（需自备梯子）。

**2 基本架构**

**（1）Cassandra**没有像BigTable或Hbase那样选择中心控制节点，**而选择了无中心的P2P架构**，网络中的所有节点都是对等的，它们构成了一个环，节点之间通过P2P协议每秒钟交换一次数据，这样每个节点都拥有其它所有节点的信息，包括位置、状态等，如下图所示。

**（2）Cassandra Ring**

客户端可以连接集群中的任一个节点，和客户端建立连接的节点叫协作者(coordinator)，它相当于一个代理，负责定位该次请求要发到哪些实际拥有本次请求所需数据的节点上去获取，但如何获取并返回，主要根据客户端要求的一致性级别（Consistency Level）来定，比如：ONE指只要有一个节点返回数据就可以对客户端做出响应，QUONUM指需要返回几个根据用户的配置数目，ALL指等于数据复制份数的所有节点都返回结果才能向客户端做出响应，对于数据一致性要求不是特别高的可以选择ONE，它是最快的一种方式。

**（3）Cassandra的核心组件包括：**

**Gossip：点对点的通讯协议**，用来相互交换节点的位置和状态信息。当一个节点启动时就立即本地存储Gossip信息，但当节点信息发生变化时需要清洗历史信息，比如IP改变了。通过Gossip协议，每个节点定期每秒交换它自己和它已经交换过信息的节点的数据，每个被交换的信息都有一个版本号，这样当有新数据时可以覆盖老数据，为了保证数据交换的准确性，所有的节点必须使用同一份集群列表，这样的节点又被称作seed。

**Partitioner：负责在集群中分配数据，**由它来决定由哪些节点放置第一份的copy，一般情况会使用Hash来做主键，将每行数据分布到不同的节点上，以确保集群的可扩展性。

**Replica placement strategy：复制策略**，确定哪个节点放置复制数据，以及复制的份数。

**Snitch：定义一个网络拓扑图**，用来确定如何放置复制数据，高效地路由请求。

**cassandra.yaml：主配置文件**，设置集群的初始化配置、表的缓存参数、调优参数和资源使用、超时设定、客户端连接、备份和安全

**3 数据复制和分发**

**（1）数据分发和复制通常是一起的**，数据用表的形式来组织，用主键来识别应该存储到哪些节点上，行的copy称作replica。当一个集群被创建时，至少要指定如下几个配置：Virtual Nodes，Partitioner，Replication Strategy，Snitch。

**（2）数据复制策略有两种**，一种是SimpleStrategy，适合一个数据中心的情况，第一份数据放在Partitioner确定的节点，后面的放在顺时针找到的节点上，它不考虑跨数据中心和机架的复制。另外一种是NetworkTopologyStargegy，第一份数据和前一种一样，第二份复制的数据放在不同的机架上，每个数据中心可以有不同数据的replicas。

**（3）Partitioner策略有三种**，默认是Murmur3Partitioner，使用MurmurHash。RandomPartitioner，使用Md5 Hash。ByteOrderedPartitioner使用数据的字节进行有顺分区。Cassandra默认使用MurmurHash，这种有更高的性能。

**（4）Snitch用来确定从哪个数据中心和哪个机架上写入或读取数据,有如下几种策略：**

@DynamicSnitch：监控各节点的执行情况，根据节点执行性能自动调节，大部分情况推荐使用这种配置

@SimpleSnitch：不会考虑数据库和机架的情况，当使用SimpleStategy策略时考虑使用这种情况

@RackInterringSnitch：考虑数据库中和机架

@PropertyFileSnitch：用cassandra-topology.properties文件来自定义@GossipPropertyFileSnitch:定义一个本地的数据中心和机架，然后使用Gossip协议将这个信息传播到其它节点，对应的配置文件是cassandra-rockdc.properties

**4 失败检测和修复（Failure detection and recovery）**

Cassandra从Gossip信息中确认某个节点是否可用，避免客户端请求路由到一个不可用的节点，或者执行比较慢的节点，这个通过dynamic snitch可以判断出来。Cassandra不是设定一个固定值来标记失败的节点，而是通过连续的计算单个节点的网络性能、工作量、以及其它条件来确定一个节点是否失败。节点失败的原因可能是硬件故障或者网络中断等，节点的中断通常是短暂的但有时也会持续比较久的时间。节点中断并不意味着这个节点永久不可用了，因此不会永久地从网络环中去除，其它节点会定期通过Gossip协议探测该节点是否恢复正常。如果想永久的去除，可以使用nodetool手工删除。

当节点从中断中恢复过来后，它会缺少最近写入的数据，这部分数据由其它复制节点暂为保存，叫做Hinted Handoff，可以从这里进行自动恢复。但如果节点中断时间超过maxhintwindowinms（默认3小时）设定的值，这部分数据将会被丢弃，此时需要用nodetool repair在所有节点上手工执行数据修复，以保证数据的一致性。

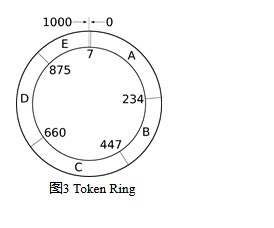
**5 动态扩展**

Cassandra最初版本是通过一致性Hash来实现节点的动态扩展的，这样的好处是每次增加或减少节点只会影响相邻的节点，但这个会带来一个问题就是造成数据不均匀，比如新增时数据都会迁移到这台机的机器上，减少时这台机器上的数据又都会迁移到相邻的机器上，而其它机器都不能分担工作，势必会造成性能问题。从1.2版本开始，Cassandra引入了虚拟节点(Virtual Nodes)的概念，为每个真实节点分配多个虚拟节点（默认是256），这些节点并不是按Hash值顺序排列，而是随机的，这样在新增或减少一个节点时，会有很多真实的节点参与数据的迁移，从而实现了负载匀衡。

**6 分区策略**

**（1）Token，Partitioner**

Cassandra中，Token是用来分区数据的关键。每个节点都有一个第一无二的Token，表明该节点分配的数据范围。节点的Token形成一个Token环。例如使用一致性HASH进行分区时，键值对将根据一致性Hash值来判断数据应当属于哪个Token。分区策略的不同，Token的类型和设置原则也有所不同。



**（2）Cassandra (0.6版本)本身支持三种分区策略：**

**1）RandomPartitioner：**随机分区是一种hash分区策略，使用的Token是大整数型(BigInteger)，范围为0~2^127，Cassandra采用了MD5作为hash函数，其结果是128位的整数值(其中一位是符号位，Token取绝对值为结果)。因此极端情况下，一个采用随机分区策略的Cassandra集群的节点可以达到2^127+1个节点。采用随机分区策略的集群无法支持针对Key的范围查询。

**2）OrderPreservingPartitioner：**如果要支持针对Key的范围查询，那么可以选择这种有序分区策略。该策略采用的是字符串类型的Token。每个节点的具体选择需要根据Key的情况来确定。如果没有指定InitialToken，则系统会使用一个长度为16的随机字符串作为Token，字符串包含大小写字符和数字。

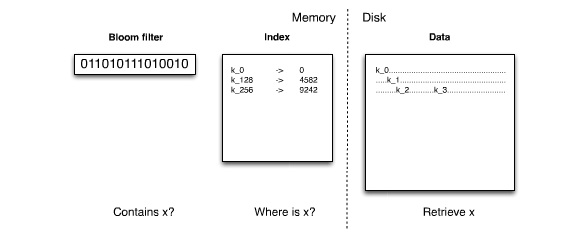
**3）CollatingOrderPreservingPartitioner：**和OrderPreservingPartitioner一样是有序分区策略。只是排序的方式不一样，采用的是字节型Token，支持设置不同语言环境的排

序方式，代码中默认是en\_US。分区策略和每个节点的Token(Initial Token)都可以在storage-conf.xml配置文件中设置。

**（3） bloom-filter, HASH**

Bloom Filter是一种空间效率很高的随机数据结构，本质上就是利用一个位数组来表示一个集合，并能判断一个元素是否属于这个集合。Bloom Filter的这种高效是有误差的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter不适合那些“零错误”的应用场合，而在能容忍低错误率的场合下，Bloom Filter通过极少的错误换取了存储空间的极大节省。

原理：位数组 + K个独立hash（y）函数。将位数组中hash函数对应的值的位置设为1，查找时如果发现所有hash函数对应位都是1说明存在，很明显这个过程并不保证查找的结果是完全正确的。



**（4）在Cassandra中，每个键值对使用1Byte的位数组来实现bloom-filter。**

**7 节点间通信gossip**

Cassandra使用点对点通讯协议gossip在集群中的节点间交换位置和状态信息。

gossip进程每秒运行一次，与至多3个其他节点交换信息，这样所有节点可很快了解集群中的其他节点信息。

gossip协议的具体表现形式就是配置文件中的seeds种子节点. 一个注意点是同一个集群的所有节点的种子节点应该一致。否则如果种子节点不一致, 有时候会出现集群分裂, 即会出现两个集群. 一般先启动种子节点,尽早发现集群中的其他节点。每个节点都和其他节点交换信息, 由于随机和概率,一定会穷举出集群的所有节点.同时每个节点都会保存集群中的所有其他节点。这样随便连到哪一个节点,都能知道集群中的所有其他节点. 比如cql随便连接集群的一个节点,都能获取集群所有节点的状态。也就是说任何一个节点关于集群中的节点信息的状态都应该是一致的!

## 三、 Cassandra数据模型

**1 Colum / Colum Family, SuperColum / SuperColum Family**

**2 Column是数据增量最底层（也就是最小）的部分。**它是一个包含名称（name）、值（value）和时间戳（timestamp）的三重元组。

下面是一个用JSON格式表示的column：

{ // 这是一个Column

name: "emailAddress",

value: "arin@example.com",

timestamp: 123456789

}

需要注意的是，name和value都是二进制的（技术上指byte[]），并且可以是任意长度。

**3 与HBase相比，除了Colum/Colum Family外，Cassandra还支持SuperColum/SuperColum Family。**

**4 SuperColum与Colum的区**别就是，标准Column的value是一个“字符串”，而 SuperColumn的value是一个包含多个Column的map，另一个细微的差别是：SuperColumn没有时间戳。

{ // 这是一个SuperColumn

name: "homeAddress",

// 无限数量的Column

value: {

street: {name: "street", value: "1234 x street", timestamp: 123456789},

city: {name: "city", value: "san francisco", timestamp: 123456789},

zip: {name: "zip", value: "94107", timestamp: 123456789},

}

}

Column Family（CF）是某个特定Key的Colum集合，是一个行结构类型，每个CF物理上被存放在单独的文件中。从概念上看，CF像数据库中的Table。SuperColum Family概念上和Column Family（CF）相似，只不过它是Super Colum的集合。

**5 Colum排序**

不同于数据库可以通过Order by定义排序规则，Cassandra取出的数据顺序是总是一定的，数据保存时已经按照定义的规则存放，所以取出来的顺序已经确定了。另外，Cassandra按照column name而不是column value来进行排序。

Cassandra可以通过Colum Family的CompareWith属性配置Colume值的排序，在SuperColum中，则是通过SuperColum Family的CompareSubcolumnsWith属性配置Colum的排序。

Cassandra提供了以下一些选：BytesType，UTF8Type，LexicalUUIDType，TimeUUIDType，AsciiType， Column name识别成为不同的类型，以此来达到灵活排序的目的。

## 四、副本存储

**1 Cassandra不像HBase是基于HDFS的分布式存储**，它的数据是存在每个节点的本地文件系统中。

**2 Cassandra有三种副本配置策略：**

**1) SimpleStrategy （RackUnawareStrategy）：**

副本不考虑机架的因素，按照Token放置在连续下几个节点。如图3所示，假如副本数为3，属于A节点的数据在B.C两个节点中也放置副本。

**2) OldNetworkTopologyStrategy （RackAwareStrategy）:**

考虑机架的因素，除了基本的数据外，先找一个处于不同数据中心的点放置一个副本，其余N-2个副本放置在同一数据中心的不同机架中。

**3) NetworkTopologyStrategy （DatacenterShardStrategy）：**

将M个副本放置到其他的数据中心，将N-M-1的副本放置在同一数据中心的不同机架中。

## 五、网络嗅探

**1 网络嗅探主要用来计算不同host的相对距离**，进而告诉Cassandra网络拓扑结构，以便更高效地对用户请求进行路由。

**2 主要有三种配置策略：**

**1)org.apache.cassandra.locator.SimpleSnitch：**

将不同host逻辑上的距离（Cassandra Ring）作为他们之间的相对距离。

**2)org.apache.cassandra.locator.RackInferringSnitch:**

相对距离是由rack和data center决定的，分别对应ip的第3和第2个八位组。即，如果两个节点的ip的前3个八位组相同，则认为它们在同一个rack（同一个rack中不同节点，距离相同）；如果两个节点的ip的前两个八位组相同，则认为它们在同一个数据中心（同一个data center中不同节点，距离相同）。

**3)org.apache.cassandra.locator.PropertyFileSnitch:**

相对距离是由rack和data center决定的，且它们是在配置文件cassandra-topology.properties中设置的。

## 六、一致性

**1 在一致性上，Cassandra采用了最终一致性**，可以根据具体情况来选择一个最佳的折衷，来满足特定操作的需求。Cassandra可以让用户指定读/插入/删除操作的一致性级别，一致性级别有多种，如图5所示。注：一致性级别是由副本数决定，而不是集群的节点数目决定。



**2 Quorum NRW**

N: 复制的节点数量，即副本数

R: 成功读操作的最小节点数

W: 成功写操作的最小节点数

**3 Quorum协议中，R 代表一次成功的读取操作中最小参与节点数量**，W 代表一次成功的写操作中最小参与节点数量。R + W>N ，则会产生类似quorum 的效果。该模型中的读(写)延迟由最慢的 R(W)复制决定，为得到比较小的延迟，R和W有的时候的和比N小。

**4 Quorum协议中，只需W + R > N，就可以保证强一致性。**因为读取数据的节点和被同步写入的节点是有重叠的。在一个RDBMS的复制模型中（Master/salve)，假如N=2,那么W=2,R=1此时是一种强一致性,但是这样造成的问题就是可用性的减低，因为要想写操作成功，必须要等 2个节点的写操作都完成以后才可以。在分布式系统中，一般都要有容错性，因此N一般大于3的，此时根据CAP理论，我们就需要在一致性和分区容错性之间做一平衡，如果要高的一致性，那么就配置N=W，R=1,这个时候可用性就会大大降低。如果想要高的可用性，那么此时就需要放松一致性的要求，此时可以配置W=1，这样使得写操作延迟最低，同时通过异步的机制更新剩余的N-W个节点。当存储系统保证最终一致性时，存储系统的配置一般是W+R<=N，此时读取和写入操作是不重叠的，不一致性的窗口就依赖于存储系统的异步实现方式，不一致性的窗口大小也就等于从更新开始到所有的节点都异步更新完成之间的时间。一般来说，Quorum中比较典型的NRW为（3,2,2）。

**5 维护最终一致性**

Cassandra 通过4个技术来维护数据的最终一致性，分别为逆熵（Anti-Entropy），读修复（Read Repair），提示移交（Hinted Handoff）和分布式删除。

**1) 逆熵**

这是一种备份之间的同步机制。节点之间定期互相检查数据对象的一致性，这里采用的检查不一致的方法是 Merkle Tree；

**2) 读修复**

客户端读取某个对象的时候，触发对该对象的一致性检查：读取Key A的数据时，系统会读取Key A的所有数据副本，如果发现有不一致，则进行一致性修复。如果读一致性要求为ONE，会立即返回离客户端最近的一份数据副本。然后会在后台执行Read Repair。这意味着第一次读取到的数据可能不是最新的数据；如果读一致性要求为QUORUM，则会在读取超过半数的一致性的副本后返回一份副本给客户端，剩余节点的一致性检查和修复则在后台执行；如果读一致性要求高(ALL)，则只有Read Repair完成后才能返回一致性的一份数据副本给客户端。可见，该机制有利于减少最终一致的时间窗口。

**3) 提示移交**

对写操作，如果其中一个目标节点不在线，先将该对象中继到另一个节点上，中继节点等目标节点上线再把对象给它：Key A按照规则首要写入节点为N1，然后复制到N2。假如N1宕机，如果写入N2能满足ConsistencyLevel要求，则Key A对应的RowMutation将封装一个带hint信息的头部（包含了目标为N1的信息），然后随机写入一个节点N3，此副本不可读。同时正常复制一份数据到N2，此副本可以提供读。如果写N2不满足写一致性要求，则写会失败。 等到N1恢复后，原本应该写入N1的带hint头的信息将重新写回N1。

**4) 分布式删除**

单机删除非常简单，只需要把数据直接从磁盘上去掉即可，而对于分布式，则不同，分布式删除的难点在于：如果某对象的一个备份节点 A 当前不在线，而其他备份节点删除了该对象，那么等 A 再次上线时，它并不知道该数据已被删除，所以会尝试恢复其他备份节点上的这个对象，这使得删除操作无效。Cassandra 的解决方案是：本地并不立即删除一个数据对象，而是给该对象标记一个hint，定期对标记了hint的对象进行垃圾回收。在垃圾回收之前，hint一直存在，这使得其他节点可以有机会由其他几个一致性保证机制得到这个hint。Cassandra 通过将删除操作转化为一个插入操作，巧妙地解决了这个问题。

## 七、存储机制

**1 Cassandra的存储机制借鉴了Bigtable的设计，采用Memtable和SSTable的方式。**

**2 CommitLog**

和HBase一样，Cassandra在写数据之前，也需要先记录日志，称之为Commit Log，然后数据才会写入到Column Family对应的MemTable中，且MemTable中的数据是按照key排序好的。SSTable一旦完成写入，就不可变更，只能读取。下一次Memtable需要刷新到一个新的SSTable文件中。所以对于Cassandra来说，可以认为只有顺序写，没有随机写操作。

**3 MenTable**

MemTable是一种内存结构，当数据量达到块大小时，将批量flush到磁盘上，存储为SSTable。这种机制，相当于缓存写回机制(Write-back Cache)，优势在于将随机IO写变成顺序IO写，降低大量的写操作对于存储系统的压力。所以我们可以认为Cassandra中只有顺序写操作，没有随机写操作。

**4 SSTable**

SSTable是Read Only的，且一般情况下，一个CF会对应多个SSTable，当用户检索数据时，Cassandra使用了Bloom Filter，即通过多个hash函数将key映射到一个位图中，来快速判断这个key属于哪个SSTable。

为了减少大量SSTable带来的开销，Cassandra会定期进行compaction，简单的说，compaction就是将同一个CF的多个SSTable合并成一个SSTable。

**5 在Cassandra中compaction主要完成的任务是：**

1） 垃圾回收： cassandra并不直接删除数据，因此磁盘空间会消耗得越来越多，compaction 会把标记为删除的数据真正删除；

2） 合并SSTable：compaction 将多个 SSTable 合并为一个（合并的文件包括索引文件，数据文件，bloom filter文件），以提高读操作的效率；

3） 生成 MerkleTree：在合并的过程中会生成关于这个 CF 中数据的 MerkleTree，用于与其他存储节点对比以及修复数据。

**6 单体、模块化**

Cassandra和HBase的一个重要区别是， Cassandra在每个节点是是一个单 Java 进程，而完整的HBase 解决方案却由不同部分组成：有数据库进程本身，它可能会运行在多个模式；一个配置好的 hadoop HDFS 分布式文件系统，以及一个 Zookeeper 系统来协调不同的 HBase 进程。

## 八、Java接口

**1 maven依赖**

<dependency>

<groupId>com.datastax.cassandra</groupId>

<artifactId>cassandra-driver-core</artifactId>

<version>3.1.0</version>

</dependency>

<dependency>

<groupId>net.jpountz.lz4</groupId>

<artifactId>lz4</artifactId>

<version>1.2.0</version>

</dependency>

**2 cassandra客户端**

package cassandra;

import com.datastax.driver.core.BoundStatement;

import com.datastax.driver.core.Cluster;

import com.datastax.driver.core.HostDistance;

import com.datastax.driver.core.PoolingOptions;

import com.datastax.driver.core.PreparedStatement;

import com.datastax.driver.core.ProtocolOptions;

import com.datastax.driver.core.ResultSet;

import com.datastax.driver.core.Session;

import com.datastax.driver.core.SocketOptions;

import com.datastax.driver.core.policies.DCAwareRoundRobinPolicy;

import com.datastax.driver.core.policies.DefaultRetryPolicy;

import com.google.common.base.Preconditions;

import java.util.List;

public class CassandraClient {

/\*\*

\* cluster

\*/

private static Cluster cluster;

/\*\*

\* session

\*/

private static Session session;

/\*\*

\* table name

\*/

private String keyspace;

/\*\*

\* server ip

\*/

private String serverIp;

public CassandraClient(){

}

public CassandraClient(String keyspace, String serverIp){

this.keyspace = keyspace;

this.serverIp = serverIp;

createClient();

}

/\*\*

\* 创建cassandra客户端

\*/

public void createClient() {

Preconditions.checkNotNull(keyspace, "keyspace must not be null.");

Preconditions.checkNotNull(serverIp, "serverIp must not be null.");

Cluster.Builder builder = Cluster.builder().addContactPoint(serverIp);

// 设置连接池

PoolingOptions poolingOptions = getPoolingOptions();

builder.withPoolingOptions(poolingOptions);

// socket 链接配置

SocketOptions socketOptions = new SocketOptions().setKeepAlive(true).setReceiveBufferSize(1024

\* 1024).setSendBufferSize(1024

\* 1024).setConnectTimeoutMillis(5

\* 1000).setReadTimeoutMillis(1000);

builder.withSocketOptions(socketOptions);

// 设置压缩方式

builder.withCompression(ProtocolOptions.Compression.LZ4);

// 负载策略

DCAwareRoundRobinPolicy dCAwareRoundRobinPolicy = DCAwareRoundRobinPolicy.builder().withLocalDc("myLocalDC").withUsedHostsPerRemoteDc(2).allowRemoteDCsForLocalConsistencyLevel().build();

builder.withLoadBalancingPolicy(dCAwareRoundRobinPolicy);

// 重试策略

builder.withRetryPolicy(DefaultRetryPolicy.INSTANCE);

cluster = builder.build();

session = cluster.connect(keyspace); // (2)

Preconditions.checkNotNull(session, "session must not be null.");

}

/\*\*

\* 构建连接池

\*

\* @return 连接池

\*/

private PoolingOptions getPoolingOptions() {

PoolingOptions poolingOptions = new PoolingOptions().setCoreConnectionsPerHost(HostDistance.LOCAL,

4).setMaxConnectionsPerHost(HostDistance.LOCAL,

10).setCoreConnectionsPerHost(HostDistance.REMOTE,

2).setMaxConnectionsPerHost(HostDistance.REMOTE,

20);

return poolingOptions;

}

/\*\*

\* 执行预处理sql

\*

\* @param sql 数据库语句

\* @param params 占位符值

\* @return 执行结果

\*/

public ResultSet executePreparedSQL(String sql, List<String> params) {

Preconditions.checkNotNull(session, "session must not be null.");

Preconditions.checkNotNull(sql, "sql must not be null.");

Preconditions.checkNotNull(params, "params must not be null.");

PreparedStatement prepared = session.prepare(sql);

BoundStatement bound = prepared.bind(params.toArray());

return session.execute(bound);

}

/\*\*

\* 执行sql语句

\*

\* @param sql 数据库语句

\* @return 执行结果

\*/

public ResultSet executeSQL(String sql) {

Preconditions.checkNotNull(session, "session must not be null.");

Preconditions.checkNotNull(sql, "sql must not be null.");

return session.execute(sql);

}

public void close() {

if (session != null) {

session.close();

}

if (null != cluster) {

cluster.close();

}

}

public String getKeyspace() {

return keyspace;

}

public void setKeyspace(String keyspace) {

this.keyspace = keyspace;

}

public String getServerIp() {

return serverIp;

}

public void setServerIp(String serverIp) {

this.serverIp = serverIp;

}

}

**3 运行main函数**

package cassandra;

import com.datastax.driver.core.ResultSet;

import com.datastax.driver.core.Row;

import com.google.common.collect.Lists;

import java.util.List;

public class CassandraTest {

public static void main(String[] args) {

// 创建客户端

CassandraClient cassandraClient = new CassandraClient("demodb", "127.0.0.1");

// 插入数据

cassandraClient.executeSQL("INSERT INTO users (user\_name, password, gender) VALUES ('zhangsan', 'admin', 'male')");

List<String> params = Lists.newArrayList("root", "root", "male");

cassandraClient.executePreparedSQL("INSERT INTO users (user\_name, password, gender) VALUES (?, ?, ?)", params);

// 查询数据

ResultSet rs = cassandraClient.executeSQL("select \* from users where user\_name = 'root'");

// 输出

Row row = rs.one();

System.out.println("row=" + row.toString());

System.out.println("user\_name=" + row.getString("user\_name"));

System.out.println("password=" + row.getString("password"));

System.out.println("gender=" + row.getString("gender"));

// 关闭

cassandraClient.close();

}

}

**4 运行结果**

2016-12-12 23:43:21,812 - com.datastax.driver.core.ClockFactory -0 [main] INFO - Using native clock to generate timestamps.

2016-12-12 23:43:22,212 - com.datastax.driver.core.NettyUtil -400 [main] INFO - Did not find Netty's native epoll transport in the classpath, defaulting to NIO.

2016-12-12 23:43:23,678 - com.datastax.driver.core.policies.DCAwareRoundRobinPolicy -1866 [main] INFO - Using data-center name 'datacenter1' for DCAwareRoundRobinPolicy (if this is incorrect, please provide the correct datacenter name with DCAwareRoundRobinPolicy constructor)

2016-12-12 23:43:23,683 - com.datastax.driver.core.Cluster -1871 [main] INFO - New Cassandra host /127.0.0.1:9042 added

row=Row[root, root, male]

user\_name=root

password=root

gender=male

**5 通过客户端查看刚刚插入数据**

cqlsh> use demodb;

cqlsh:demodb> select \* from users ;

user\_name | gender | password

-----------+--------+----------

zhangsan | male | admin

zhengyong | male | 1234

root | male | root

zhangsan2 | male | admin

lisi | male | admin

zhangsan1 | male | admin

(6 rows)

## 九、MongoDB vs.Cassandra

**1 数据结构**

MongoDB使用文档型存储，其数据结构为与JSON类似的BSON结构，而Cassandra支持的是key-value式存储，而每个key-value还会保存一个时间戳，这个时间戳实际上起到了版本控制的作用。

**2 索引结构**

MongoDB的索引几乎与关系型数据库完全一样，其普通索引、联合索引、唯一索引的意义和实现上都可以参考对MySQL索引的理解。而Cassandra由于其是一个key-value结构的存储，如果你要对value进行条件查找，那么就必须建立反向索引，重新建立一个value-key的键值对。

**3 部署**

MongoDB提供了Replica Sets的高可用部署方式，配置好RS的节点后，整个集群会自动选举出Primary机器供写入操作，并自动复制数据到其它节点。它还具有故障后自动选举新的主机的机制。而Cassandra提供的策略更为灵活，它通过一种对网络结构可感知的机制，它让你可以配置数据是备份在本。地网络中的其它节点还是备份到远端的数据中心。

**4 .商业支持**

Cassandra和MongoDB都是开源产品，不同的是Cassandra是完全自由地在Apache孵化器中进行发展。而MongoDB是由商业公司10gen在背后支持，绝大部分的MongoDB贡献者都来自10gen。

**5 .技术支持**

对MongoDB来说，最好的技术支持肯定来源于10gen，因为MongoDB是他们开发的，所以他们应该最了解。在10gen的网站上，你可以看到相关的支持支持联系方式。而Cassandra也有一些第三方的机制提供技术支持服务，这些机构基本上都有一些Cassandra的核心开发者在其中，以保证其对Cassandra能有足够的了解。

**6 发展状态**

MongoDB在10gen的带领下一直稳步而快速地进行着BUG修复和新功能增加，比如他们会举行新功能投票来让你参与MongoDB发展中来。另外值得一提的是10gen对社区的关注热情很高，对于使用用户的需求了解非常充分。相对来说Cassandra在这方面就相对混乱一些，与10gen的全职开发工程师不同，Cassandra更多的是自发的开源项目参与者，像Eric Evans这样的全职开发者非常之少。

**7 技术文档**

Cassandra的官方文档并不太好，相对好一些的是Datastax自己上面的一个版本。但还是缺乏一个完整详尽的介绍。而MongoDB的文档相对来说比较完整而成体系。在其官方网站上你就能看到结构明了的文档了，而且多语言版本也都在跟进翻译中。

**8 社区活动**

这方面MongoDB做得非常出色，几乎可以用技术营销经典案例来形容。在去年一年，MongoDB相关的会议就举行了40场。相信在国内的很多朋友还记得在北京举行的Mongo

Beijing。而Cassandra的活动更多的是自发进行的。

**9 第三方客户端**

这也是boxedice选择MongoDB的另一个原因，MongoDB拥有非常广泛的客户端支持，几乎所有流行语言都有其操作客户端。而且这些客户端几乎都是由10gen官方开发并维护的。而Cassandra仅有Java和Python的官方客户端，剩下的都是一些自发的第三方客户端。