Cassandra 详解,第二部分：启动和集群

(君山，6/10/2010)

简介：本文主要讨论了 Cassandra 服务器的启动过程，包括 Cassandra 在启动过程中主要都完成了那些操作，为什么要执行这些操作，最终达到什么状态等。接着介绍如果在集群情况下，集群中节点如何自治理，节点间如何通信、如何控制数据在集群中的分布等关键问题。

一、 Cassandra 的启动过程

1. Cassandra 的功能模块

按照我的理解我将 Cassandra 的功能模块划分为三个部分：

1） 客户端协议解析。

目前这个版本 Cassandra 支持两个客户端 avro 和 thrift，使用的较多的是后者，它们都是通过 socket 协议作为网络层协议，然后再包装一层应用层协议，这个应用层协议的包装和解析都是由它们的客户端和相应的服务端模块来完成的。

这样设计的目的是解决多种多样的客户端的连接方式，既可以是短连接也可以是长连接。既可以是 Java 程序调用也可以是 PHP 调用或者多种其它编程语言都可以调用。

2） 集群 Gossip 协议。

集群中节点之间相互通信是通过 Gossip 协议来完成的，它的实现都在 org.apache.cassandra.gms.Gossiper 类中。它的主要作用就是每个节点向集群中的其它节点发送心跳，心跳携带的信息是本身这个节点持有的其它节点的状态信息包括本节点的状态，如果发现两边的状态信息不是不一致，则会用最新的状态信息替换，同时通过心跳来判断某个节点是否还在线，把这种状态变化通知感兴趣的事件监听者，以做出相应的修改，包括新增节点、节点死去、节点复活等。

除了维护节点状态信息外，还需做另外一些事，如集群之间的数据的转移，这些数据包括：读取的数据、写入的数据、状态检查的数据、修复的数据等等。

3） 数据的存储。

数据的存储包括，内存中数据的组织形式，它又包括 CommitLog 和 Memtable。磁盘的数据组织方式，它又包括 date、filter 和 index 的数据。

其它剩下的就是如何读取和操作这些数据了，可以用下图来描述 Cassandra 是如何工作的：

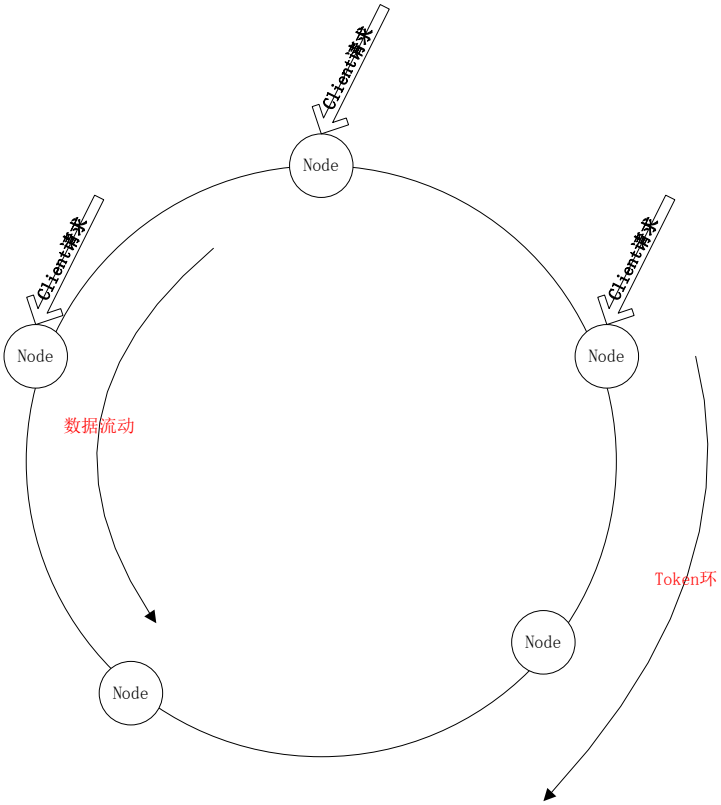

图 1 Cassandra 的工作模型

2. Cassandra 的启动过程

这里将详细介绍 Cassandra 的启动过程。Cassandra 的启动过程大概分为下面几个阶段：

2.1 storage-config.xml 配置文件的解析

配置文件的读取和解析都是在 org.apache.cassandra.config.DatabaseDescriptor 类中完成的，这个类的作用非常简单，就是读取配置文件中各个配置项所定义的值，经过简单的验证，符合条件就将其值赋给 DatabaseDescriptor 的私有静态常量。值得注意的是关于 Keyspace 的解析，按照 ColumnFamily 的配置信息构建成 org.apache.cassandra.config.CFMetaData 对象，最后把这些所有 ColumnFamily 放入 Keyspace 的 HashMap 对象 org.apache.cassandra.config.KSMetaData 中，每个 Keyspace 就是一个 Table。这些信息都是作为基本的元信息，可以通过 DatabaseDescriptor 类直接获取。DatabaseDescriptor 类相关的类结构如下图 2 所示：

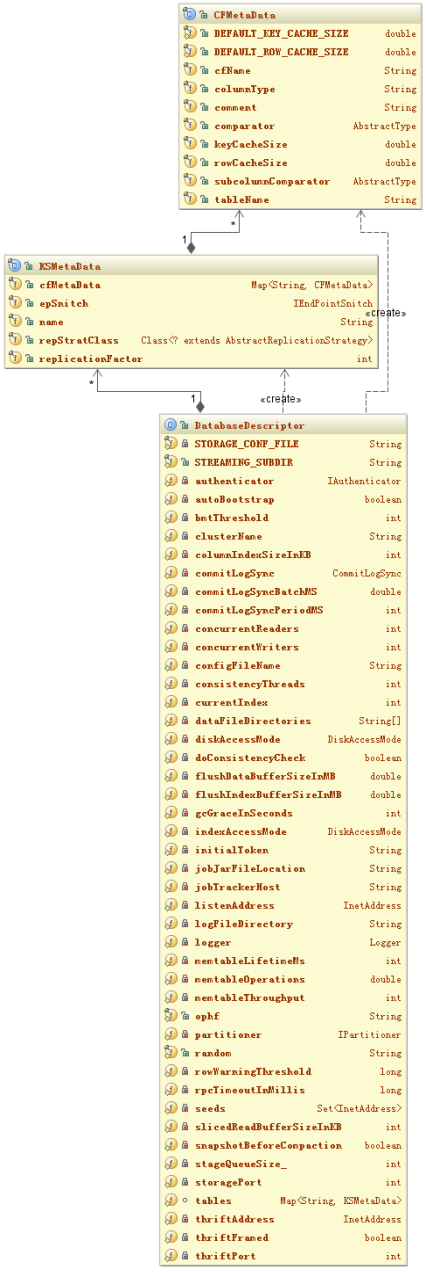

图 2 DatabaseDescriptor 类相关的类结构

2.2 创建每个 Table 的实例

创建 Table 的实例将完成：1）获取该 Table 的元信息 TableMatedate。2）创建改 Table 下每个 ColumnFamily 的存储操作对象 ColumnFamilyStore。3）启动定时程序，检查该 ColumnFamily 的 Memtable 设置的 MemtableFlushAfterMinutes 是否已经过期，过期立即写到磁盘。与 Table 相关的类如图 3 所示：

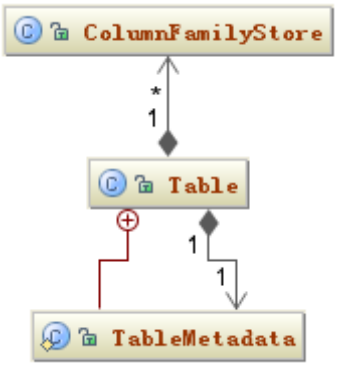

图 3 Table 相关的类图

一个 Keyspace 对应一个 Table，一个 Table 持有多个 ColumnFamilyStore，而一个 ColumnFamily 对应一个 ColumnFamilyStore。Table 并没有直接持有 ColumnFamily 的引用而是持有 ColumnFamilyStore，这是因为 ColumnFamilyStore 类中不仅定义了对 ColumnFamily 的各种操作而且它还持有 ColumnFamily 在各种状态下数据对象的引用，所以持有了 ColumnFamilyStore 就可以操作任何与 ColumnFamily 相关的数据了。与 ColumnFamilyStore 相关的类如图 4 所示

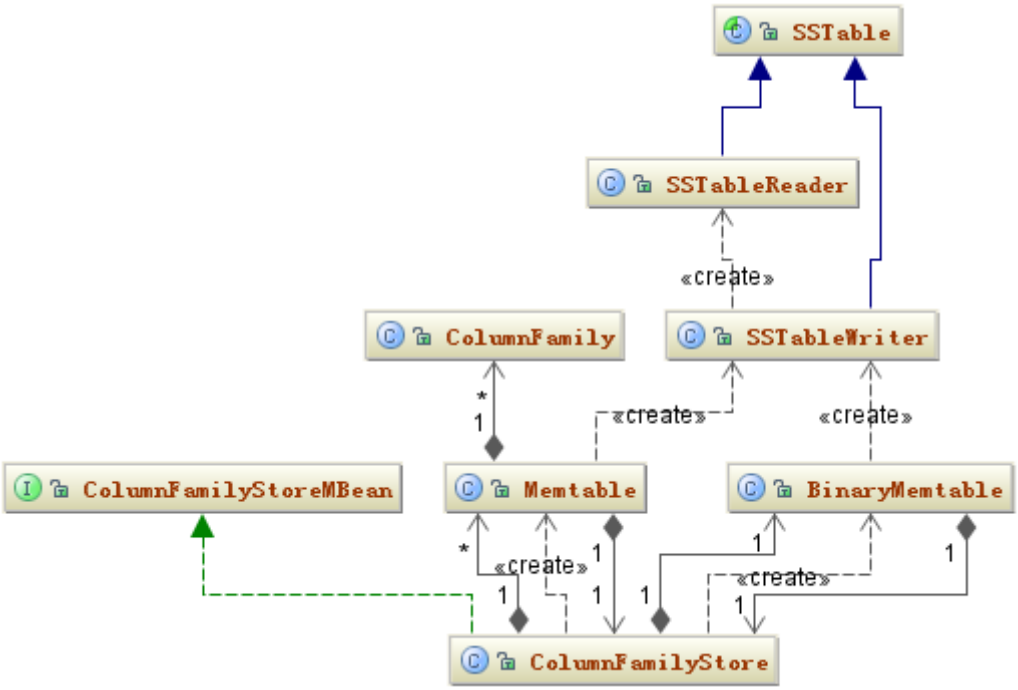

图 4 ColumnFamilyStore 相关的类

2.3 CommitLog 日志恢复

这里主要完成这几个操作，发现是否有没有被写到磁盘的数据，恢复这个数据，构建新的日志文件。CommitLog 日志文件的恢复策略是，在头文件中发现没有被序列化的最新的 ColumnFamily Id，然后取出这个这个被序列化 RowMutation 对象的起始地址，反序列化成为 RowMutation 对象，后面的操作和新添一条数据的流程是一样的，如果这个 RowMutation 对象中的数据被成功写到磁盘中，那么会在 CommitLog 去掉已经被持久化的 ColumnFamily Id。关于 CommitLog 日志文件的存储格式以及数据如何写到 CommitLog 文件中，将在后面第三部分详细介绍。

2.4 启动存储服务

这里是启动过程中最重要的一步。这里将会启动一系列服务，主要包括如下步骤。

1） 创建 StorageMetadata

StorageMetadata 将包含三个关键信息：本节点的 Token、当前 generation 以及 ClusterName，Cassandra 判断如果是第一次启动，Cassandra 将会创建三列分别存储这些信息并将它们存在在系统表的 LocationInfo ColumnFamily 中，key 是“L”。如果不是第一次启动将会更新这三个值。这里的 Token 是判断用户是否指定，如果指定了使用用户指定的，否则随机生成一个 Token。但是这个 Token 有可能在后面被修改。这三个信息被存在 StorageService 类的 storageMetadata_属性中，以便后面随时调用。

2） GCInspector.instance.start 服务

主要是统计统计当前系统中资源的使用情况，将这个信息记录到日志文件中，这个可以作为系统的监控日志使用。

3） 启动消息监听服务

这个消息监听服务就是监听整个集群中其它节点发送到本节点的所有消息，Cassandra 会根据每个消息的类型，做出相应的反应。关于消息的处理将在后面详细介绍。

4） StorageLoadBalancer.instance.startBroadcasting 服务

这个服务是每个一段时间会收集当前这个节点所存的数据总量，也就是节点的 load 数据。把这个数据更新到本节点的 ApplicationState 中，然后就可以通过这个 state 来和其它节点交换信息。这个 load 信息在数据的存储和新节点加入的时候，会有参考价值。

5） 启动 Gossiper 服务

在启动 Gossiper 服务之前，将 StorageService 注册为观察者，一旦节点的某些状态发生变化，而这些状态是 StorageService 感兴趣的，StorageService 的 onChange 方法就会触发。

Gossiper 服务就是一个定时程序，它会向本节点加入一个 HeartBeatState 对象，这个对象标识了当前节点是 Live 的，并且记录当前心跳的 generation 和 version。这个 StorageMetadata 和前面的 StorageMetadata 存储的 generation 是一致的，version 是从 0 开始的。

这个定时程序每隔一秒钟随机向 seed 中定义的节点发送一个消息，而这个消息是保持集群中节点状态一致的唯一途径。这个消息如何同步，将在后面详细介绍。

6） 判断启动模式

是否是 AutoBootstrap 模式启动，又是如何判断的，以及应作出那些相应的操作，在前面的第一部分中已有介绍，这里不再赘述。这里主要说一下，当是 Bootstrap 模式启动时，Cassandra 都做了那些事情。这一步很重要，因为它关系到后面的很多操作，对 Cassandra 的性能也会有影响。

这个过程如下：

1. 通过之前的消息同步获取集群中所有节点的 load 信息
2. 找出 load 最大的节点的 ip 地址
3. 向这个节点发送消息，获取其一半 key 范围所对应的 Token，这个 Token 是前半部分值。
4. 将这个 Token 写到本地节点
5. 本地节点会根据这个 Token 计算以及集群中的 Token 环，计算这个 Token 应该分摊集群中数据的一个范围（range）这个环应该就是，最大 load 节点的一半 key 的所对应的 range。
6. 向这个 range 所在的节点请求数据。发送 STREAM-STAGE 类型的消息，要经过 STREAM_REQUEST、STREAM_INITIATE、STREAM_INITIATE_DONE、STREAM_FINISHED 几次握手，最终才将正确的数据传输到本节点。
7. 数据传输完成时设置 SystemTable.setBootstrapped(true)标记 Bootstrap 已经启动，这个标记的目的是防止再次重启时，Cassandra 仍然会执行相同的操作。

这个过程可以用下面的时序图来描述：

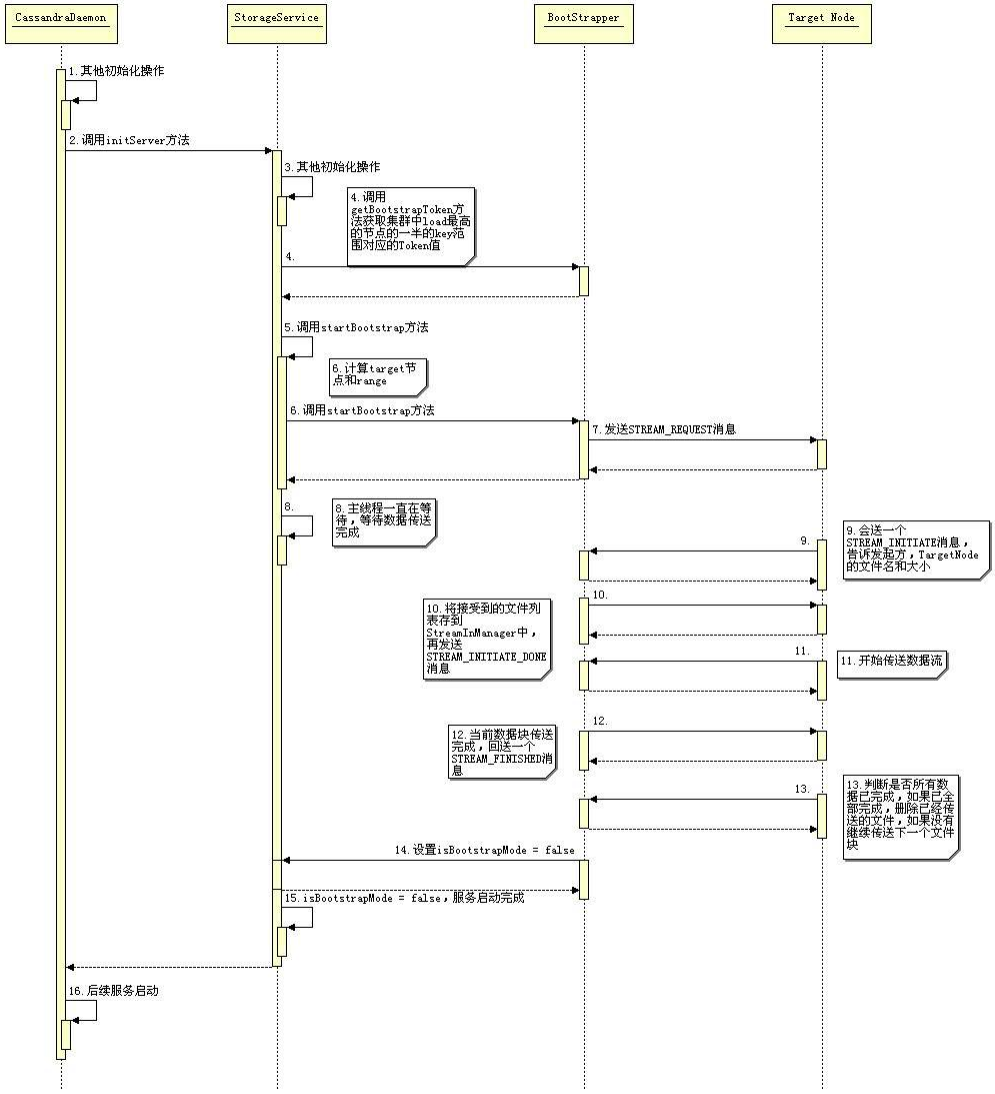

图 5 StorageService 服务启动时序图

以上是 AutoBootstrap 模式启动，如果是以非 AutoBootstrap 模式启动，那么启动将会非常简单，这个过程如下：

1） 检查配置项 InitialToken 有没有指定，如果指定了初始 Token，使用用户指定的 Token，否则将根据 Partitioner 配置项指定的数据分配策略生成一个默认的 Token，并把它写到系统表中。

2） 更新 generation=generation+1 到系统表中

3） 设置 SystemTable.setBootstrapped(true)，标记启动方式，防止用户再修改 AutoBootstrap 的启动模式。

二、 Cassandra 集群中的节点状态的同步策略

我们知道 Cassandra 集群中节点是通过自治理来对外提供服务的，它不像 Hadoop 这种 Master/Slave 形式的集群结构，会有一个主服务节点来管理所有节点中的原信息和对外提供服务的负载均衡。这种方式管理集群中的节点逻辑上比较简单也很方便，但是也有其弱点，那就是这个 Master 容易形成瓶颈，其稳定性也是一种挑战。而 Cassandra 的集群管理方式就是一种自适应的管理方式，集群中的节点没有 Master、Slave 之分，它们都是平等的，每个节点都可以单独对外提供服务，某个节点 Crash 也不会影响到其它节点。但是一旦某个节点的状态发生变化，整个集群中的所有节点都要知道，并且都会执行预先设定好的应对方案，这会造成节点间要发送大量的消息交换各自状态，这样也增加了集群中状态和数据一致性的复杂度，但是优点是它是一个高度自治的组织，健壮性比较好。

1. 消息交换

那么 Cassandra 是如何做到这么高度自治的呢？这个问题的关键就是它们如何同步各自的状态信息，同步消息的前提是它们有一种约定的消息交换机制。这个机制就是 Gossip 协议，Cassandra 就是通过 Gossip 协议相互交换消息。

前面在 Cassandra 服务启动时提到了 Gossiper 服务的启动，一旦 Cassandra 启动成功，Gossiper 服务就是一直执行下去，它是一个定时程序。这个服务的代码在 org.apache.cassandra.gms.Gossiper 类中，下面是定时程序执行的关键代码如清单 2 所示：

清单 2 Gossiper.GossipTimerTask.run

```
public void run(){
                synchronized( Gossiper.instance ){
                 endPointStateMap .get(localEndPoint ).getHeartBeatState().updateHeartBeat();
                 List<GossipDigest> gDigests = new ArrayList<GossipDigest>();
                 Gossiper.instance.makeRandomGossipDigest(gDigests);
                 if ( gDigests.size() > 0 ){
                    Message message = makeGossipDigestSynMessage(gDigests);
                    boolean gossipedToSeed = doGossipToLiveMember(message);
                    doGossipToUnreachableMember(message);
                    if (!gossipedToSeed || liveEndpoints_.size() < seeds_.size())
                       doGossipToSeed(message);
                    doStatusCheck();
                 }
              }
}
```

Cassandra 通过向其它节点发送心跳来证明自己仍然是活着的，心跳里面包含有当前的 generation，用来表示有的节点是不是死了又复活的。

本地节点所保存的所有其它节点的状态信息都被放在了 GossipDigest 集合中。一个 GossipDigest 对象将包含这个节点的 generation、maxVersion 和节点地址。接着将会组装一个 Syn 消息（关于 Cassandra 中的消息格式将在后面介绍），同步一次状态信息 Cassandra 要进行三次会话，这三次会话分别是 Syn、Ack 和 Ack2。当组装成 Syn 消息后 Cassandra 将随机在当前活着的节点列表中选择一个向其发送消息。

Cassandra 中的消息格式如下：

1) header：消息头 org.apache.cassandra.net.Header，消息头中包含五个属性：消息编号（messageId）、发送方地址（from）、消息类型（type）、所要做的动作（verb）和一个 map 结构（details）
2) body：消息内容，是一个 byte 数组，用来存放序列化的消息主体。

可以用下面的图 3 更形象的表示：

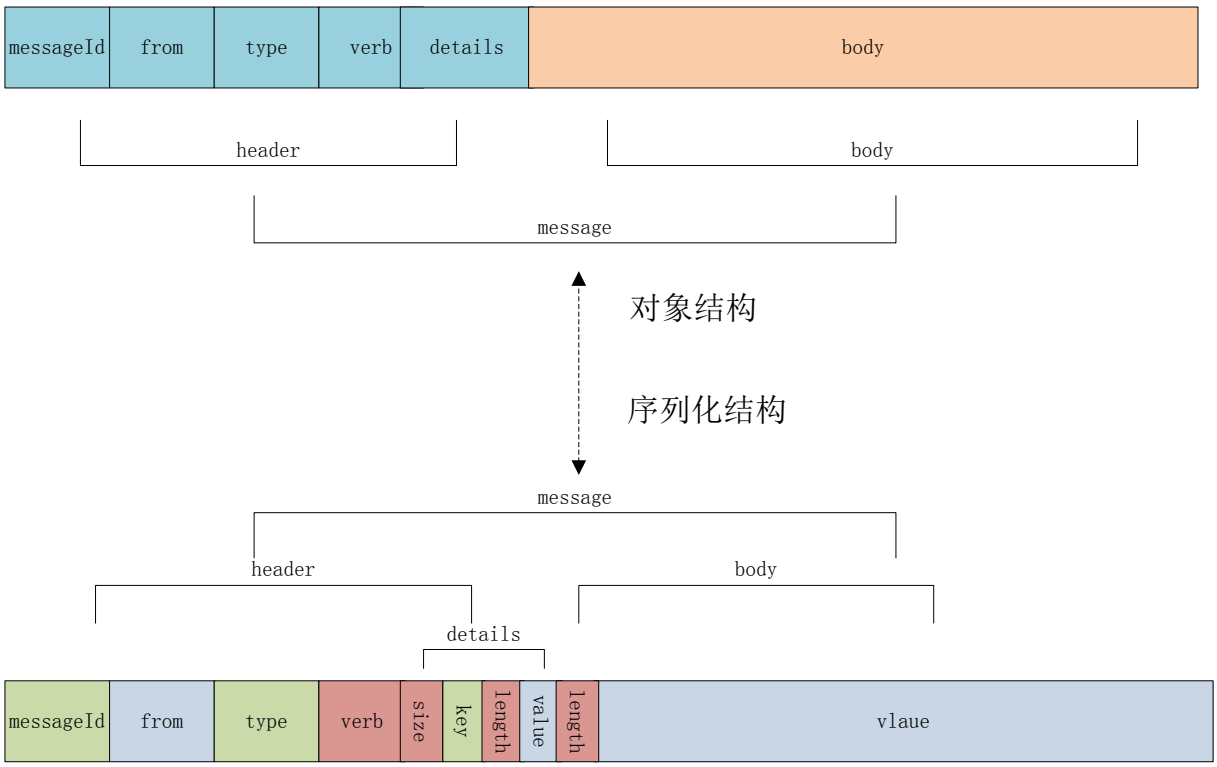

图 6 message 消息结构

当组装成一个 message 后，再将这个消息按照 Gossip 协议组装成一个 pocket 发送到目的地址。关于这个 pocket 数据包的结构如下：

1） header：包头，4 bytes。前两个是 serializer type；第三个是是否压缩包，默认是否；最后一个 byte 表示是否是 streaming mode。

2） body：包体，message 的序列化字节数据。

这个 pocket 的序列化字节结构如下：

图 7 通信协议包的结构

当另外一个节点接受到 Syn 消息后，反序列化 message 的 byte 数组，它会取出这个消息的 verb 执行相应的动作，Syn 的 verb 就是解析出发送节点传过来的节点的状态信息与本地节点的状态信息进行比对，看哪边的状态信息更新，如果发送方更新，将这个更新的状态所对应的节点加入请求列表，如果本地更新，则将本地的状态再回传给发送方。回送的消息是 Ack，当发送方接受到这个 Ack 消息后，将接受方的状态信息更新的本地对应的节点。再将接收方请求的节点列表的状态发送给接受方，这个消息是 Ack2，接受方法接受到这个 Ack2 消息后将请求的节点的状态更新到本地，这样一次状态同步就完成了。

不管是发送方还是接受方每当节点的状态发生变化时都将通知感兴趣的观察者做出相应的反应。消息同步所涉及到的类由下面图 8 的关系图表示：

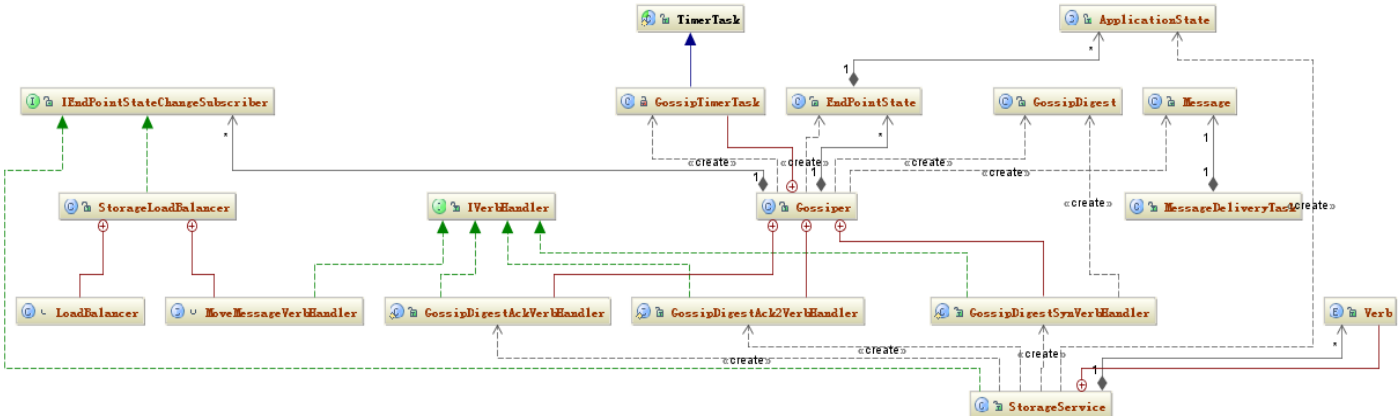

图 8 节点状态同步相关类结构图

节点的状态同步操作有点复杂，如果前面描述的还不是很清楚的话，再结合下面的时序图，你就会更加明白了，如图 9 所示：

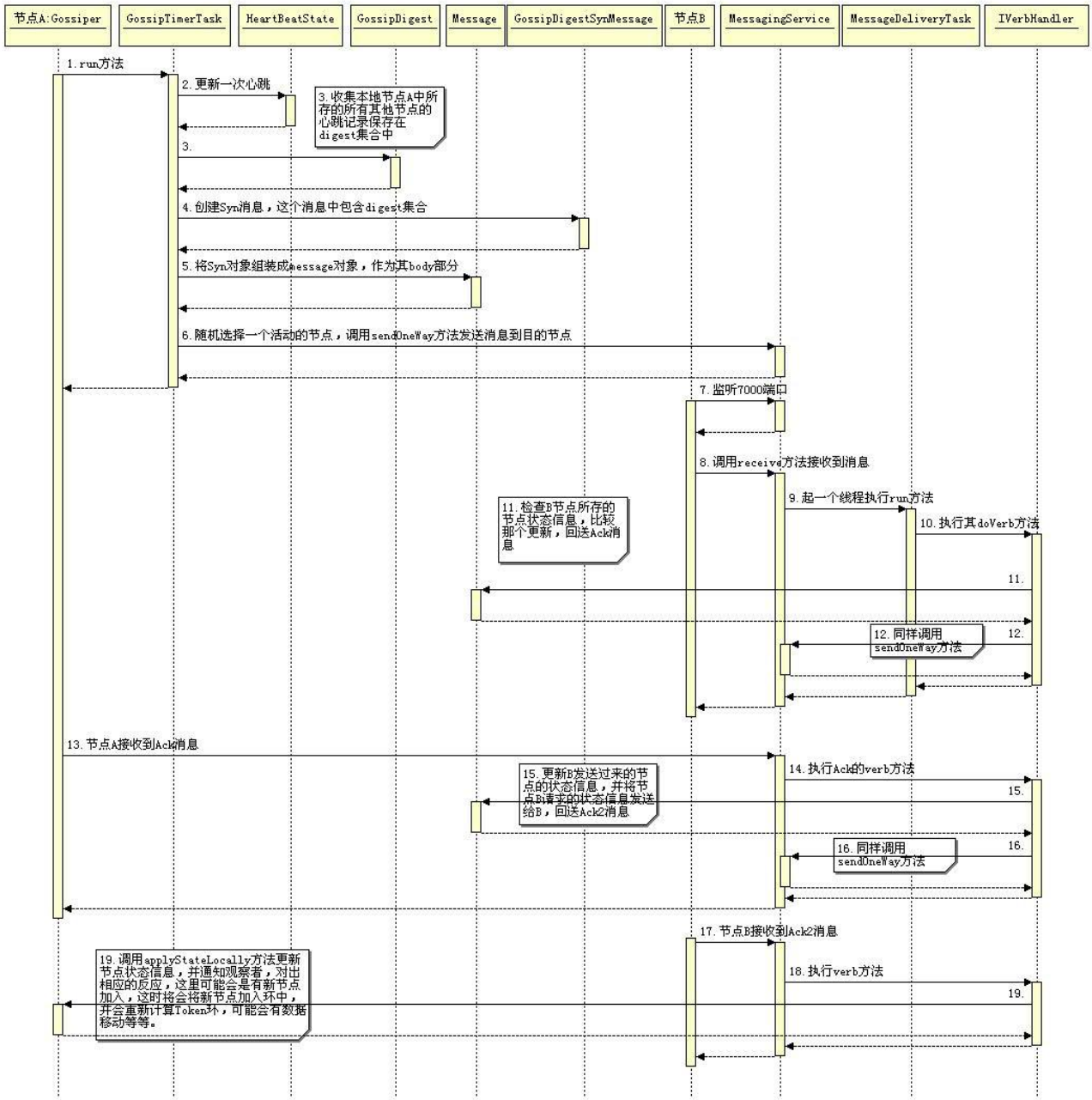

图 9 节点状态同步时序图

上图中省去了一部分重复的消息，还有节点是如何更新状态也没有在图中反映出来，这些部分在后面还有介绍，这里也无法完整的描述出来。

2. 状态更新

前面提到了消息的交换，它的目的就是可以根据交换的信息更新各自的状态。Cassandra 更新状态是通过观察者设计模式来完成的，订阅者被注册在 Gossiper 的集合中，当交换的消息中的节点的状态和本地节点不一致时，这时就会更新本地状态，更改本地状态本身并没有太大的意义，有意义的是状态发生变化这个动作，这个动作发生时，就会通知订阅者来完成这个状态发生变化后应该做出那些相应的改动，例如，发现某个节点已经不在集群中时，那么对这个节点应该要在本地保存的 Live 节点列表中移去，防止还会有数据发送到这个无法到达的节点。和状态相关的类如下：

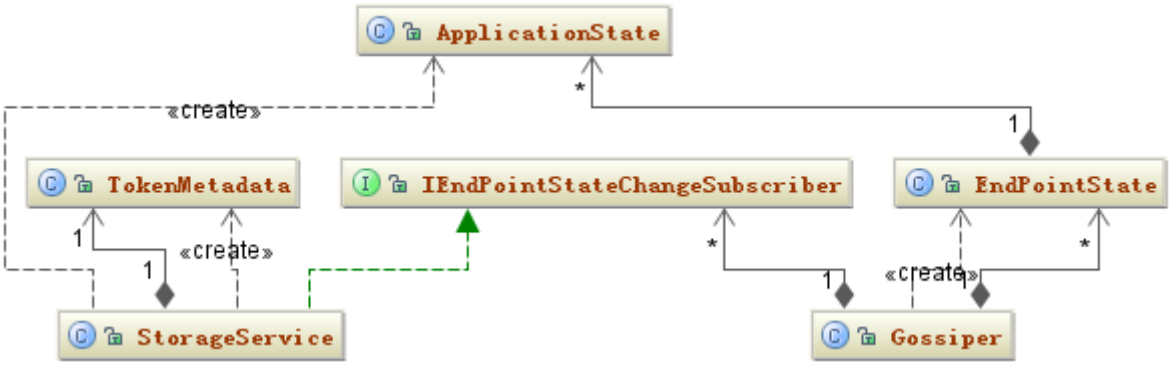

图 10 更新状态相关的类

从上图可以看出节点的状态信息由 ApplicationState 表示，并保存在 EndPointState 的集合中。状态的修改将会通知 IendPointStateChangeSubscriber，继而再更新 Subscriber 的具体实现类修改相应的状态。

下面是新节点加入的时序图，如图 11 所示：

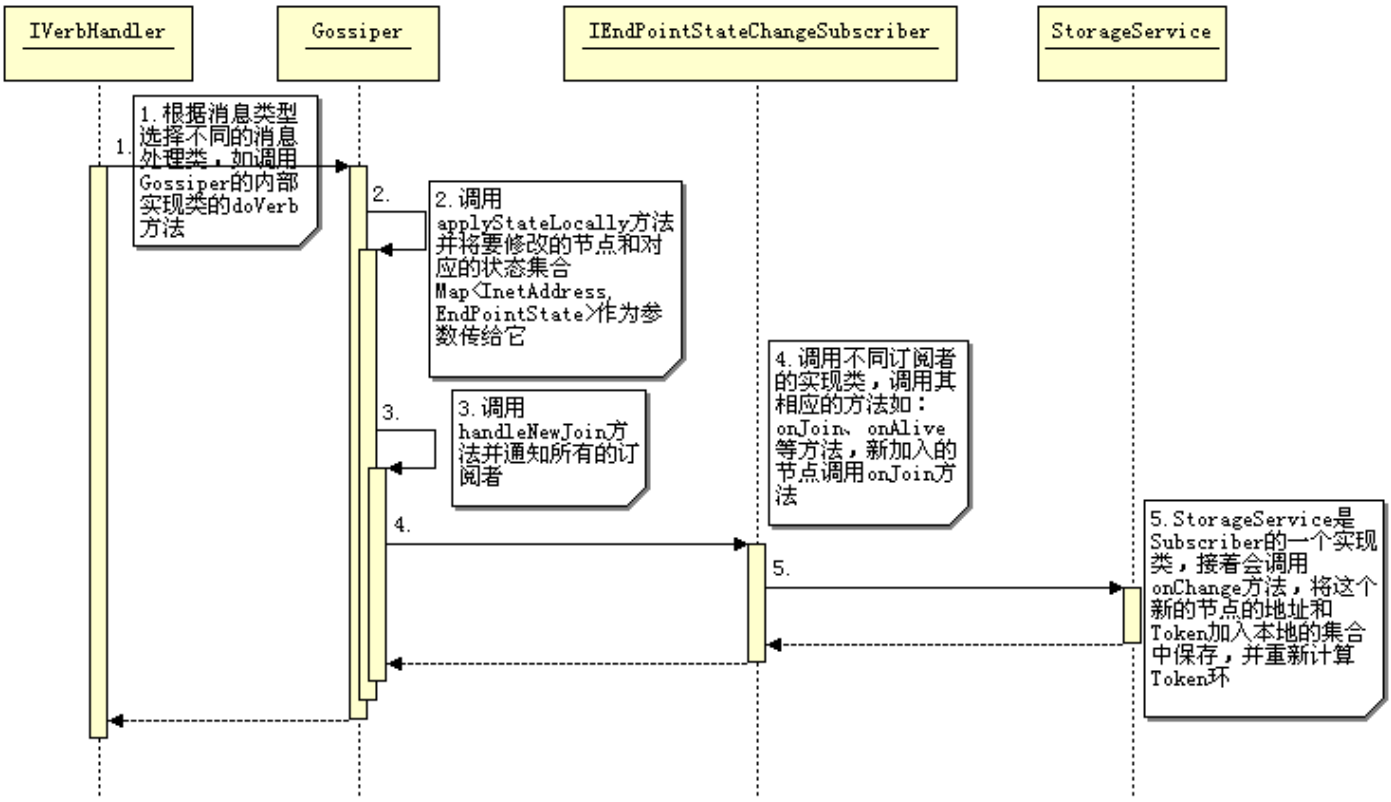

图 11 新加入节点的时序图

上图基本描述了 Cassandra 更新状态的过程，需要说明的点是，Cassandra 为何要更新节点的状态，这实际上就是关于 Cassandra 对集群中节点的管理，它不是集中管理的方式，所以每个节点都必须保存集群中所有其它节点的最新状态，所以将本节点所持有的其它节点的状态与另外一个节点交换，这样做有一个好处就是，并不需要和某个节点通信就能从其它节点获取它的状态信息，这样就加快了获取状态的时间，同时也减少了集群中节点交换信息的频度。另外，节点状态信息的交换的根本还是为了控制集群中 Cassandra 所维护的一个 Token 环，这个 Token 是 Cassandra 集群管理的基础。因为数据的存储和数据流动都在这个 Token 环上进行，一旦环上的节点发生变化，Cassandra 就要马上调整这个 Token 环，只有这样才能始终保持整个集群正确运行。

到底哪些状态信息对整个集群是重要的，这个主要就在图 10 中的 TokenMetadata 类中，它主要记录了当前这个集群中，哪些节点是 live 的哪些节点现在已经不可用了，哪些节点可能正在启动，以及每个节点它们的 Token 是多少。而这些信息都是为了能够精确控制集群中的那个 Token 环。只要每个集群中每个节点所保存的是同一个 Token 环，整个集群中的节点的状态就是同步的，反之，集群中节点的状态就没有同步。

三、 总结

本文主要介绍了，Cassandra 的启动过程，以及 Cassandra 是如何管理集群的。实际上 Cassandra 的启动和集群的管理是连在一起的，启动过程中的很多步骤都是集群管理的一部分，如节点以 AutoBootstrap 方式启动，在启动过程中就涉及到数据的重新分配，这个分配的过程正是在动态调整集群中 Token 环的过程。所以当你掌握了 Cassandra 是如何动态调整这个 Token 环，你也就掌握了 Cassandra 的集群是如何管理的了。