分布式系统作业 5

18340166　王若琪

1. Leader selection在分布式系统中具有重要的用途，主要用于容错，即当主节点失效后能够从备份节点中选择新的leader，但是新选择的leader需要得到其他节点的认同。主流的leader 选择算法有：Bully、 Ring算法，但不限于这些算法，调研以下软件，简述这些软件所采用的选举算法：Zookeeper、 Redis、 MongoDB、 Cassandra。

- **Zookeeper 选举：**
  - zookeeper节点状态：

    LOOKING
    LEADING
    FOLLOWERING
    OBSERVE

    当最初的时候，每个server的最初状态都是LOOKING，当leader服务器选举出来后，leader服务器状态变为LEADING，不是observe服务器的server的状态自动变为FOLLOWERING，当leader服务器挂掉之后，所有非observe的server将状态都改为LOOKING，进行新的选举，选举出新的leader服务器，然后leader服务器状态变为LEADING，不是observe的server的状态自动变为FOLLOWERING。
  - zxid（zookeeper事务id）

    zookeeper状态每次改变都会收到一个不同全局唯一的zxid，删除节点，创建节点都会使zookeeper状态改变，zxid不断递增。
  - leader服务器选取规则

    优先检查zxid，zxid大的作为leader服务器。zxid相同就比较myid大小，myid大的作为leader服务器，只有获取过半server的支持才能成为leader。
  - zookeeper选举算法

    basic paxos和fast paxos。
    basic paxos：选举线程向所有server发起一次询问，按照服务器选取规则去比较，选出下一次要询问的server，当被选取的server有一般server支持，则成为leader服务器，不然就一直选举，直到选出了leader。
    fast paxos：一个server声明自己要做leader，其它server将配合工作，解决zxid和epoch冲突，并向该server发送接收提议的消息。

- **Redis 选举：**

  当redis集群的主节点故障时，Sentinel集群将从剩余的从节点中选举一个新的主节点，有以下步骤：

  1. 故障节点主观下线

     Sentinel集群的每一个Sentinel节点会定时对redis集群的所有节点发心跳包检测节点是否正常。如果一个节点在`down-after-milliseconds`时间内没有回复Sentinel节点的心跳包，则该redis节点被该Sentinel节点主观下线。
  2. 故障节点客观下线

当节点被一个Sentinel节点记为主观下线时，并不意味着该节点肯定故障了，还需要Sentinel集群的其他Sentinel节点共同判断为主观下线才行。

该Sentinel节点会询问其他Sentinel节点，如果Sentinel集群中超过`quorum`数量的Sentinel节点认为该redis节点主观下线，则该redis客观下线。

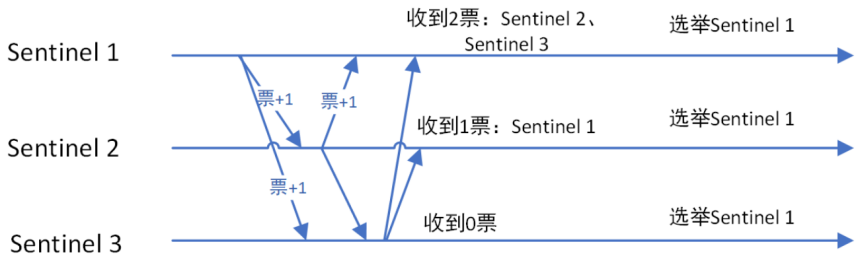
如果客观下线的redis节点是从节点或者是Sentinel节点，则操作到此为止，没有后续的操作了；如果客观下线的redis节点为主节点，则开始故障转移，从从节点中选举一个节点升级为主节点。

3. Sentinel集群选举Leader

   如果需要从redis集群选举一个节点为主节点，首先需要从Sentinel集群中选举一个Sentinel节点作为Leader。

   每一个Sentinel节点都可以成为Leader，当一个Sentinel节点确认redis集群的主节点主观下线后，会请求其他Sentinel节点要求将自己选举为Leader。被请求的Sentinel节点如果没有同意过其他Sentinel节点的选举请求，则同意该请求(选举票数+1)，否则不同意。

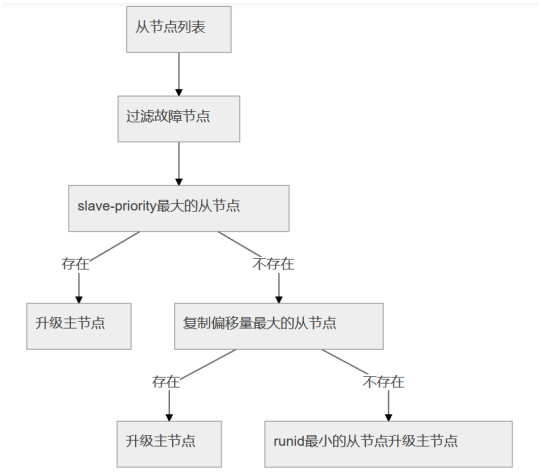
   如果一个Sentinel节点获得的选举票数达到Leader最低票数(`quorum`和`Sentinel`节点数/2+1的最大值)，则该Sentinel节点选举为Leader；否则重新进行选举。



4. Sentinel Leader决定新主节点

   当Sentinel集群选举出Sentinel Leader后，由Sentinel Leader从redis从节点中选择一个redis节点作为主节点：

   1. 过滤故障的节点。
   2. 选择优先级`slave-priority`最大的从节点作为主节点，如不存在则继续。
   3. 选择复制偏移量（数据写入量的字节，记录写了多少数据。主服务器会把偏移量同步给从服务器，当主从的偏移量一致，则数据是完全同步）最大的从节点作为主节点，如不存在则继续。
   4. 选择`runid`（redis每次启动的时候生成随机的`runid`作为redis的标识）最小的从节点作为主节点。



- **MongoDB 选举：**

MongoDB节点之间维护心跳检查，主节点选举由心跳触发。

- 心跳检查

  MongoDB复制集成员会向自己之外的所有成员发送心跳并处理响应信息，因此每个节点都维护着从该节点POV看到的其他所有节点的状态信息。节点根据自己的集群状态信息判断是否需要更换新的Primary。

  在实现的时候主要由两个异步的过程分别处理心跳响应和超时，抛开复杂的条件检查，核心逻辑主要包括：
  - Secondary节点权重比Primary节点高时，发起替换选举；
  - Secondary节点发现集群中没有Primary时，发起选举；
  - Primary节点不能访问到大部分(Majority)成员时主动降级；

  降级操作会断开链接，终止用户请求等。

- 选举发起

  发起选举的节点需要首先做一些条件判断，比如节点位于备选节点列表中、POV包含复制集Majority等，真实情况的条件判断更加复杂。然后将自己标记为选举过程中，并发起投票请求。

- 投票

  投票发起者向集群成员发起Elect请求，成员在收到请求后经过会一系列检查，如果通过检查则为发起者投一票。一轮选举中每个成员最多投一票，在PV0中用30秒"选举锁"避免为其他发起者重复投票，这导致了如果新选举的Primary挂掉，可能30秒内不会有新的Primary选举产生；在PV1中通过为投票引入单调递增的Term解决重复投票的问题。

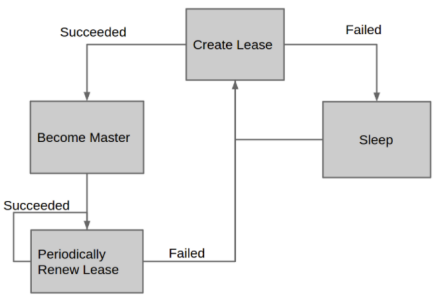
  如果投票发起者获得超过半数的投票，则选举通过成为Primary节点，否则重新发起投票。

- 注意

  - MongoDB选举需要获得大多数投票才能通过，在一轮选举中两个节点得票相同则重新选举，为避免陷入无限重复选举，MongoDB建议复制集的成员个数为奇数个，当Secondary节点个数为偶数时，可以增加一个Arbiter节点，。
  - PV0版本中，所有成员都可以投否决票，一个否决票会将得票数减少10000，所以一般可以认为只要有成员反对，则该节点不能成为Primary。PV1版本取消了否决票。
  - 选举过程中，复制集没有主节点，所有成员都是只读状态。

- **Cassandra 选举：**
  - 可以通过让想成为主节点的节点在一开始时就尝试获得租约的这种方法来选举领导者。
  - Cassandra 将相同的结果返回给所有的客户端，并且当选的 master 会每隔 N 秒进行续约。
  - 客户端会以轮询的方式来确认主服务器仍然是主服务器。
  - 如果 master 去世，clients 将会知道租约确实，并且等待参与者选出新的 master。

  此选举算法概括如下图：



2. 可靠多播在分布式系统中具有重要的用途，比如传播选举消息等，可靠多播的实现方式有多种，请从以下软件中选择一种，编译运行，观察是否可以实现可靠多播，并撰写报告。

我选择：https://github.com/daeyun/reliable-multicast-chat

- **实验过程**
  - 下载源代码，因为系统版本的问题，直接运行源代码会有如下报错：

    ```
    AttributeError: module 'socket' has no attribute 'SO_REUSEPORT'
    ```

    只需将 socket.SO_REUSEPORT 修改为 socket.SO_REUSEADDR，即可成功编译运行。
  - 编译运行方法：

    在 reliable_multicast_chat 文件夹下输入命令：

    ```
    python main.py [process ID] [delay time] [drop rate]
    ```

    在本实验中，为了方便观察，可以将 [delay time] 以及 [drop rate] 都设为0，运行四个线程，所以[process ID] 分别为 0、 1、 2、 3 。
  - 分析原理

    分析源码中实现可靠多播功能的部分，可知原理大致如下：

    当一个进程单播消息时，它将其储存在未确认的消息缓冲区中，并将其保存在那里，直到收到 ack 消息。我们有一个名为 `ack_handler` 的线程，周期性地跟踪接收到缓冲区中每个项目的 ack 消息，并根据接收进程是否知道消息，重新发送消息或者从缓冲区中删除消息，接受进程发送 ack 消息。

    每次它收到一条消息，但实际上并不处理重复消息（用消息 id 标示）。

- **实验结果**
  - 运行四个线程，线程号分别为 0、 1、 2、 3 ，其他选项均设为 0 。

    ```
    wrq@wrq-Inspiron-7490:~/Desktop/hw5/reliable-multicast-chat-master/reliable_mult
    icast_chat$ python main.py 0 0 0
    ```

    ```
    wrq@wrq-Inspiron-7490:~/Desktop/hw5/reliable-multicast-chat-master/reliable_mult
    icast_chat$ python main.py 1 0 0
    ```

    ```
    wrq@wrq-Inspiron-7490:~/Desktop/hw5/reliable-multicast-chat-master/reliable_mult
    icast_chat$ python main.py 2 0 0
    ```

    ```
    wrq@wrq-Inspiron-7490:~/Desktop/hw5/reliable-multicast-chat-master/reliable_mult
    icast_chat$ python main.py 3 0 0
    ```
  - 以 0 号线程为例，展示实验结果，0 号进程发送消息，其他进程都能收到，并能够显示该消息来自 0 号进程：

    0 号线程截图：

    ```
    wrq@wrq-Inspiron-7490:~/Desktop/hw5/reliable-multicast-chat-master/reliable_mult
    icast_chat$ python main.py 0 0 0
    hi!
    ```

    1 号线程截图：

    ```
    wrq@wrq-Inspiron-7490:~/Desktop/hw5/reliable-multicast-chat-master/reliable_mult
    icast_chat$ python main.py 1 0 0
    0 says:  hi!
    ```

    2 号线程截图：

```
wrq@wrq-Inspiron-7490:~/Desktop/hw5/reliable-multicast-chat-master/reliable_mult
icast_chat$ python main.py 2 0 0
0 says:  hi!
```

3 号线程截图：

```
wrq@wrq-Inspiron-7490:~/Desktop/hw5/reliable-multicast-chat-master/reliable_mult
icast_chat$ python main.py 3 0 0
0 says:  hi!
```

可见正确实现可靠多播效果。

- 所有线程都进行可靠多播测试结果截图：

  0 号线程截图：

```
wrq@wrq-Inspiron-7490:~/Desktop/hw5/reliable-multicast-chat-master/reliable_mult
icast_chat$ python main.py 0 0 0
hi!
1 says:  hello~
2 says:  wow
3 says:  hahahahahahaha
```

  1 号线程截图：

```
wrq@wrq-Inspiron-7490:~/Desktop/hw5/reliable-multicast-chat-master/reliable_mult
icast_chat$ python main.py 1 0 0
0 says:  hi!
hello~
2 says:  wow
3 says:  hahahahahahaha
```

  2 号线程截图：

```
wrq@wrq-Inspiron-7490:~/Desktop/hw5/reliable-multicast-chat-master/reliable_mult
icast_chat$ python main.py 2 0 0
0 says:  hi!
1 says:  hello~
wow
3 says:  hahahahahahaha
```

  3 号线程截图：

```
wrq@wrq-Inspiron-7490:~/Desktop/hw5/reliable-multicast-chat-master/reliable_mult
icast_chat$ python main.py 3 0 0
0 says:  hi!
1 says:  hello~
2 says:  wow
hahahahahahaha
```

  根据以上实验结果，可知本程序能够实现可靠多播。

参考网页：

https://blog.51cto.com/13917261/2163268

https://blog.csdn.net/sz85850597/article/details/86751215

https://blog.csdn.net/wentyoon/article/details/78986174

https://www.datastax.com/blog/consensus-cassandra