23-从哨兵Leader选举学习Raft协议实现（上）

你好，我是蒋德钧。

在上节课，我们了解了哨兵实例的初始化过程。哨兵实例一旦运行后，会周期性地检查它所监测的主节点的运行状态。当发现主节点出现客观下线时，哨兵实例就要开始执行故障切换流程了。

不过，我们在部署哨兵实例时，通常会部署多个哨兵来进行共同决策，这样就避免了单个哨兵对主节点状态的误判。但是这同时也给我们带来了一个问题，即**当有多个哨兵判断出主节点故障后，究竟由谁来执行故障切换？**

实际上，这就和**哨兵Leader选举**有关了。而哨兵Leader选举，又涉及到分布式系统中经典的共识协议：Raft协议。学习和掌握Raft协议的实现，对于我们在分布式系统开发中实现分布式共识有着非常重要的指导作用。

所以接下来的两节课，我会带你了解Raft协议以及Redis源码中，基于Raft协议实现Leader选举的具体设计思路。今天我们先来学习下Raft协议的基本流程、它和哨兵Leader选举的关系，以及哨兵工作的整体执行流程，这部分内容也是我们学习哨兵Leader选举的必备知识。

哨兵Leader选举和Raft协议

当哨兵发现主节点有故障时，它们就会选举一个Leader出来，由这个Leader负责执行具体的故障切换流程。但因为哨兵本身会有多个实例，所以，在选举Leader的过程中，就需要按照一定的协议，让多个哨兵就“Leader是哪个实例”达成一致的意见，这也就是**分布式共识**。

而Raft协议可以用来实现分布式共识，这是一种在分布式系统中实现多节点达成一致性的算法，可以用来在多个节点中选举出Leader节点。为了实现这一目标，Raft协议把节点设计成了三种类型，分别是Leader、Follower和Candidate。

Raft协议对于Leader节点和Follower节点之间的交互有两种规定：

- 正常情况下，在一个稳定的系统中，只有Leader和Follower两种节点，并且Leader会向Follower发送心跳消息。
- 异常情况下，如果Follower节点在一段时间内没有收到来自Leader节点的心跳消息，那么，这个Follower节点就会转变为Candidate节点，并且开始竞选Leader。

然后，当一个Candidate节点开始竞选Leader时，它会执行如下操作：

- 给自己投一票；
- 向其他节点发送投票请求，并等待其他节点的回复；
- 启动一个计时器，用来判断竞选过程是否超时。

在这个Candidate节点等待其他节点返回投票结果的过程中，如果它**收到了Leader节点的心跳消息**，这就表明，此时已经有Leader节点被选举出来了。那么，这个Candidate节点就会转换为Follower节点，而它自己发起的这轮竞选Leader投票过程就结束了。

而如果这个Candidate节点，**收到了超过半数的其他Follower节点返回的投票确认消息**，也就是说，有超过半数的Follower节点都同意这个Candidate节点作为Leader节点，那么这个Candidate节点就会转换为Leader节点，从而可以执行Leader节点需要运行的流程逻辑。

这里，你需要注意的是，每个Candidate节点发起投票时，都会记录当前的投票轮次，Follower节点在投票过程中，每一轮次只能把票投给一个Candidate节点。而一旦Follower节点投过票了，它就不能再投票了。如果在一轮投票中，没能选出Leader节点，比如有多个Candidate节点获得了相同票数，那么Raft协议会让Candidate节点进入下一轮，再次开始投票。

好了，现在你就了解了Raft协议中Leader选举的基本过程和原则。不过你还要清楚一点，就是**Redis哨兵在实现时，并没有完全按照Raft协议来实现**，这主要体现在，Redis哨兵实例在正常运行的过程中，不同实例间并不是Leader和Follower的关系，而是**对等的关系**。只有当哨兵发现主节点有故障了，此时哨兵才会按照Raft协议执行选举Leader的流程。

接下来，我们就从代码层面来看下，哨兵是如何执行Raft协议来选举Leader的。

哨兵的时间事件处理函数sentinelTimer

我们先来看下哨兵的时间事件处理函数sentinelTimer（在sentinel.c文件中），因为哨兵Leader选举是在这个函数执行过程中触发的。

sentinelTimer函数本身是在serverCron函数（在server.c文件中）中调用的，如下所示：

```
int serverCron(struct aeEventLoop *eventLoop, long long id, void *clientData) {
…
if (server.sentinel_mode) sentinelTimer(); //如果当前运行的是哨兵，则运行哨兵的时间事件处理函数
…
}
```

serverCron函数每100ms执行一次，在执行过程中，它会检查**server.sentinel_mode配置项**，如果该配置项为1，就表明当前运行的是哨兵实例，紧接着它就会调用sentinelTimer函数。因此，sentinelTimer函数也会周期性执行。我在上节课给你介绍过server.sentinel_mode配置项的设置，你也可以再去回顾下。

接着，sentinelTimer会调用**sentinelHandleDictOfRedisInstances函数**。这个函数的原型如下，它的参数是一个哈希表：

```
void sentinelHandleDictOfRedisInstances(dict *instances)
```

实际上，当sentinelTimer调用sentinelHandleDictOfRedisInstances时，传入的哈希表参数，就是当前哨兵实例状态信息sentinelState结构中维护的master哈希表，其中记录了当前哨兵监听的主节点，如下所示：

```
void sentinelTimer(void) {
    ...
    //将当前哨兵监听的主节点作为参数传入sentinelHandleDictOfRedisInstances函数
    sentinelHandleDictOfRedisInstances(sentinel.masters);
    ...
}
```

sentinelHandleDictOfRedisInstances函数会执行一个循环流程，在该流程中，它会从sentinel.master哈希表中逐一取出监听的主节点，并调用sentinelHandleRedisInstance函数对该主节点进行处理，如下所示：

```
void sentinelHandleDictOfRedisInstances(dict *instances) {
   ...
   di = dictGetIterator(instances); //获取哈希表的迭代器
   while((de = dictNext(di)) != NULL) {
      //从哈希表中取出一个实例
      sentinelRedisInstance *ri = dictGetVal(de);
      //调用sentinelHandleRedisInstance处理实例
      sentinelHandleRedisInstance(ri);
      ...
    }
   ...
}
```

注意，这里的**sentinelHandleRedisInstance函数**是哨兵工作机制中的一个重要函数，它实现了哨兵实例工作的主体逻辑。下面我们就先来了解下它的主要执行步骤，然后我们再分别学习其中关键步骤的实现细节。

sentinelHandleRedisInstance函数的执行流程

首先你要知道，sentinelHandleRedisInstance函数会被周期性执行，用来检测哨兵监听的节点的状态。这个函数主要会依次执行以下四个步骤。

第一步：重建连接

sentinelHandleRedisInstance会调用sentinelReconnectInstance函数，尝试和断连的实例重新建立连接。

第二步：发送命令

sentinelHandleRedisInstance会调用sentinelSendPeriodicCommands函数，向实例发送PING、INFO等命令。

第三步：判断主观下线

sentinelHandleRedisInstance会调用sentinelCheckSubjectivelyDown函数，检查监听的实例是否主观下线。

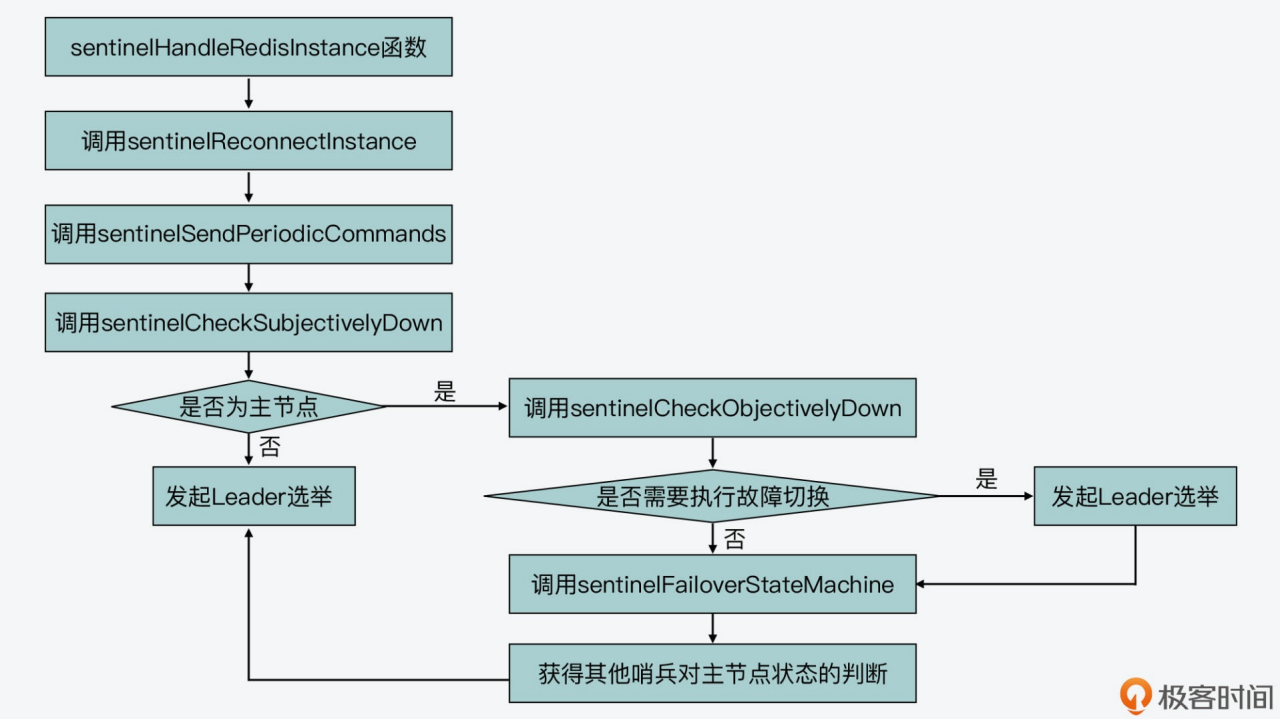
第四步：判断客观下线和执行故障切换

在这一步中，sentinelHandleRedisInstance函数的运行逻辑主要是针对被监听的主节点来执行的，而这一步又可以分成以下四个小步骤：

- 首先，针对监听的主节点，调用sentinelCheckObjectivelyDown函数检查其是否客观下线。
- 紧接着，调用sentinelStartFailoverIfNeeded函数判断是否要启动故障切换。如果要启动故障切换，就调用sentinelAskMasterStateToOtherSentinels函数，获取其他哨兵实例对主节点状态的判断，并向其他哨兵发送is-master-down-by-addr命令，发起Leader选举。
- 然后，调用sentinelFailoverStateMachine执行故障切换。
- 最后，再次调用sentinelAskMasterStateToOtherSentinels函数，获取其他哨兵实例对主节点状态的判断。

这里你需要注意下，因为sentinelHandleRedisInstance函数处理的对象是sentinelRedisInstance结构的实例，而sentinelRedisInstance结构可以表示主节点、从节点以及哨兵实例。在刚才介绍的四个大步骤中，第一、二和三步会对主节点、从节点和哨兵实例都执行，而第四步只有在当前sentinelRedisInstance表示主节点时，才会执行。

下图也展示了sentinelHandleRedisInstance函数执行的基本逻辑。



现在，我们就了解了sentinelHandleRedisInstance函数的基本执行过程。

另外，就像刚才给你介绍的，因为sentinelHandleDictOfRedisInstances函数接收的参数，是当前哨兵监听的主节点哈希表，而每个主节点又会记录同时监听它的其他哨兵实例以及它的从节点，这分别对应了主节点数据结构sentinelRedisInstance中的sentinels和slaves成员变量，这两个变量本身也是用哈希表来保存其他哨兵和从节点信息的，如下所示：

```
typedef struct sentinelRedisInstance {
```

```
...
 dict *sentinels;    //监听同一个主节点的其他哨兵实例
 dict *slaves;      //当前主节点的从节点
...
}
```

所以，哨兵在sentinelHandleDictOfRedisInstances函数中，调用sentinelHandleRedisInstance处理完每个主节点后，还会针对监听主节点的其他哨兵实例，以及主节点的从节点，分别调用sentinelHandleDictOfRedisInstances函数进行处理，如下所示：

```
//如果当前是主节点，那么调用sentinelHandleDictOfRedisInstances分别处理该主节点的从节点，以及监听该主节点的其他哨兵
if (ri->flags & SRI_MASTER) {
   sentinelHandleDictOfRedisInstances(ri->slaves);
   sentinelHandleDictOfRedisInstances(ri->sentinels);
   ...
}
```

也就是说，**sentinelTimer周期性执行的一个重要任务，就是sentinelHandleDictOfRedisInstances函数。**

那么，sentinelTimer除了调用sentinelHandleDictOfRedisInstances以外，它一开始还会调用**sentinelCheckTiltCondition函数**检查是否需要进入TILT模式。这里，你需要注意下，对于哨兵来说，TILT模式是一种特殊的运行模式，当哨兵连续两次的时间事件处理间隔时长为负值，或是间隔时长过长，那么哨兵就会进入TILT模式。在该模式下，哨兵只会定期发送命令收集信息，而不会执行故障切换流程。

此外，sentinelTimer函数在调用执行完sentinelHandleDictOfRedisInstances函数后，还会依次调用sentinelRunPendingScripts、sentinelCollectTerminatedScripts和sentinelKillTimedoutScripts这三个函数，来运行待执行的脚本、收集结束的脚本以及将超时的脚本kill掉。

最后，sentinelTimer函数会**调整server.hz配置项**，它会在server.hz默认值的基础上增加一个随机值，而这个配置项决定了sentinelTimer本身的执行频率。因此在调整后，sentinelTimer函数就会按照修改后的运行频率再次执行。

下面的代码展示了sentinelTimer函数的整体执行流程，你可以再回顾下。

```
void sentinelTimer(void) {
    sentinelCheckTiltCondition();
    sentinelHandleDictOfRedisInstances(sentinel.masters);
    sentinelRunPendingScripts();
    sentinelCollectTerminatedScripts();
    sentinelKillTimedoutScripts();
    server.hz = CONFIG_DEFAULT_HZ + rand() % CONFIG_DEFAULT_HZ;
}
```

好了，到这里，我们就了解了哨兵实例的时间事件处理函数sentinelTimer。在该函数的执行流程中，你需

要重点关注的是sentinelHandleRedisInstance函数，这是哨兵周期性检测主节点下线状态和执行故障切换的主要函数。并且一旦需要执行故障切换，哨兵的Leader选举也会发生在这里。所以接下来，我们就来具体学习下sentinelHandleRedisInstance函数的实现。

sentinelHandleRedisInstance函数的内部实现

通过前面针对sentinelHandleRedisInstance函数执行流程的介绍，现在我们知道，该函数首先会依次调用sentinelReconnectInstance、sentinelSendPeriodicCommand和sentinelCheckSubjectiveDown这三个函数。所以这里，我们先来看下这三个函数的实现和主要作用。然后在下节课，我会给你详细介绍sentinelHandleRedisInstance中其他函数的实现，以此帮助你全面掌握哨兵工作过程中的关键操作。

sentinelReconnectInstance函数

sentinelReconnectInstance函数的主要作用是**判断哨兵实例和主节点间连接是否正常**，如果发生了断连情况，它会重新建立哨兵和主节点的连接。

其实，哨兵在使用sentinelRedisInstance结构保存主节点信息时，在该结构中有一个instanceLink类型的成员变量**link**，该变量就记录了哨兵和主节点间的两个连接，分别对应用来发送命令的连接cc和用来发送Pub/Sub消息的连接pc，如下所示：

```
typedef struct instanceLink {
...
redisAsyncContext *cc; //用于发送命令的连接
redisAsyncContext *pc; //用于发送pub-sub消息的连接
...
}
```

sentinelReconnectInstance函数执行时会**检查这两个连接是否为NULL**。如果是的话，那么它就会调用redisAsyncConnectBind函数（在async.c文件中），重新和主节点建立这两个连接。

这是因为，哨兵在监听主节点状态过程中，正是要通过命令连接cc向主节点发送命令，而通过Pub/Sub连接pc，订阅主节点的Hello频道，从而就可以通过这个频道再发现监听同一主节点的其他哨兵实例。

这样，在完成了和主节点的连接重建后，哨兵会继续调用sentinelSendPeriodicCommands函数。

sentinelSendPeriodicCommands函数

sentinelSendPeriodicCommands的逻辑比较简单，它先是调用**redisAsyncCommand函数**（在async.c文件中），通过哨兵和主节点间的命令连接cc，向主节点发送INFO命令。然后，再通过**sentinelSendPing函数**（在sentinel.c文件中）向主节点发送PING命令（PING命令的发送也是通过哨兵和主节点的命令连接cc来完成的）。

最后，sentinelSendPeriodicCommands函数会调用**sentinelSendHello函数**（在sentinel.c文件中），通过哨兵和主节点的命令连接cc，向主节点发送PUBLISH命令，将哨兵自身的IP、端口号和ID号信息发送给主节点。

接下来，哨兵就会调用sentinelCheckSubjectivelyDown函数，来判断监听的主节点是否主观下线。

sentinelCheckSubjectivelyDown函数

sentinelCheckSubjectivelyDown函数首先会计算当前距离上次哨兵发送PING命令的时长elapsed，如下所示：

```
void sentinelCheckSubjectivelyDown(sentinelRedisInstance *ri) {
...
if (ri->link->act_ping_time)  //计算当前距离上一次发送PING命令的时长
        elapsed = mstime() - ri->link->act_ping_time;
else if (ri->link->disconnected) //如果哨兵和主节点的连接断开了，那么计算当前距离连接最后可用的时长
        elapsed = mstime() - ri->link->last_avail_time;
...
}
```

计算完elapsed之后，sentinelCheckSubjectivelyDown函数会分别检测哨兵和主节点的命令发送连接，以及Pub/Sub连接的活跃程度。如果活跃度不够，那么哨兵会调用instanceLinkCloseConnection函数（在sentinel.c文件中），断开当前连接，以便重新连接。

紧接着，sentinelCheckSubjectivelyDown函数会根据以下两个条件，判断主节点是否为主观下线。

- **条件一**：当前距离上次发送PING的时长已经超过down_after_period阈值，还没有收到回复。down_after_period的值是由sentinel.conf配置文件中，down-after-milliseconds配置项决定的，其默认值是30s。
- **条件二**：哨兵认为当前实例是主节点，但是这个节点向哨兵报告它将成为从节点，并且在down_after_period时长，再加上两个INFO命令间隔后，该节点还是没有转换成功。

当上面这两个条件有一个满足时，哨兵就判定主节点为主观下线了。然后，哨兵就会调用sentinelEvent函数发送“+sdown”事件信息。下面的代码展示了这部分的判断逻辑，你可以看下。

```
 if (elapsed > ri->down_after_period ||
  (ri->flags & SRI_MASTER && ri->role_reported == SRI_SLAVE
   &&  mstime() - ri->role_reported_time > (ri->down_after_period+SENTINEL_INFO_PERIOD*2)))
 {
        //判断主节点为主观下线
        if ((ri->flags & SRI_S_DOWN) == 0) {
            sentinelEvent(LL_WARNING,"+sdown",ri,"%@");
            ri->s_down_since_time = mstime();
            ri->flags |= SRI_S_DOWN;
        }
  }
```

好了，到这里，我们就先了解了sentinelHandleRedisInstance函数执行流程中的前三个关键操作。它们会分别用于重建哨兵和监控主节点的连接，向主节点发送检测命令，以及判断主节点主观下线状态。这三步也是哨兵每次执行周期性任务的必备操作。

小结

这节课，我主要是给你介绍了哨兵工作过程中的一个重要环节，也就是哨兵Leader的选举。这个选举过程是参考了分布式系统中常用的分布式共识协议Raft协议来实现的。所以，你需要先了解Raft协议的基本流程，包括**Leader、Follower、Candidate三种节点类型**，Follower成为Candidate的条件和具体操作，以及Leader投票的规则。

那么，对于哨兵Leader选举来说，它参考了Raft协议，但你需要注意的是，哨兵在正常运行时并不像Raft协议那样区分了三种节点类型，而是**所有哨兵都是对等的**。而当哨兵发现主节点故障，要执行故障切换时，会按照Raft协议中Leader选举的规则，进行投票选出Leader。这是哨兵Leader选举和Raft协议的区别与联系。

此外，我还介绍了哨兵的**时间事件处理函数sentinelTimer**，这个函数会对哨兵监听的每个主节点，周期性调用sentinelHandleRedisInstance函数，来检查主节点在线状态。当主节点客观下线了，哨兵会启动Leader选举并执行故障切换。这节课我们是先了解了sentinelHandleRedisInstance函数的整体执行流程，这样，你也能掌握哨兵的整体工作过程。同时，针对哨兵和主节点重建连接、发送命令和检查主观下线的三个函数，你也要有所了解，它们也是哨兵工作中的三个重要步骤。

那么，在下节课，我将带你了解哨兵Leader选举的具体过程以及故障切换的执行。

每课一问

哨兵实例执行的周期性函数sentinelTimer，它在函数执行逻辑的最后，会修改server.hz配置项，如下所示：

```
void sentinelTimer(void) {
...
server.hz = CONFIG_DEFAULT_HZ + rand() % CONFIG_DEFAULT_HZ;
}
```

你知道调整server.hz的目的是什么吗？欢迎在留言区分享你的答案和思考，也欢迎你把今天的内容分享给更多的朋友。