Redis 哨兵模式之执行过程解析篇

Original Edwin 彬 星河之码 8/8

> **由经验而得的智慧，胜于书本而得的理论。**

在之前的两篇文章**Redis 哨兵模式之实现原理篇**和**Redis 哨兵模式之实践篇**中简单介绍了Redis的哨兵模式的原理和搭建的全过程，本文就基于哨兵模式聊一聊它的执行过程。

一、什么是哨兵（Sentinel）

Sentinel 其实也是一个 redis 的服务端程序，它也会定时执行 serverCron 函数，只是里面其他的程序用不到，用到的是对普通 redis 节点的监控以及故障转移模块。

Sentinel 初始化的时候会清空原来的命令表，写入自己独有的命令进去，所以普通 redis 节点支持的数据读写命令，对 Sentinel 来说都是找不到命令，因为它根本就没有初始化这些命令的执行器。

二、启动并初始化Sentinel

启动使用命令

```
redis-sentinel sentinel.conf
```

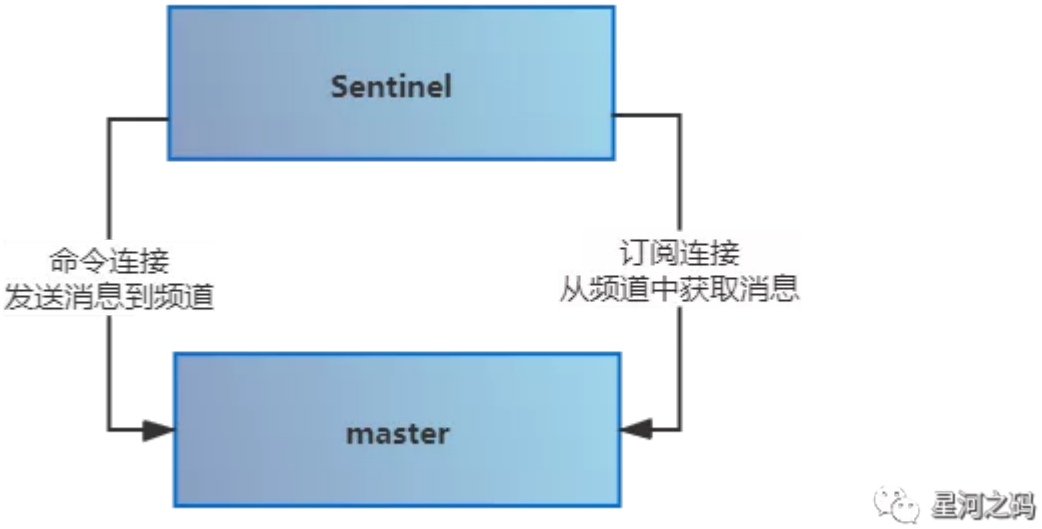
启动时，它需要执行以下几个步骤

- 初始化服务器
- 初始化sentinel状态

- 根据给定的配置文件，来初始化sentinel的监视器主服务列表
- 创建连向主服务器的连接

  Sentinel实例启动后，每个Sentinel会创建2个连向主服务器的网络连接

  - **命令连接**：用于向主服务器发送命令，并接收响应
  - **订阅连接**：用于订阅主服务器的—sentinel—:hello频道



三、获取服务器信息

- **获取主服务器信息**

  Sentinel默认会以10s一次的频率向主服务器发送INFO命令，通过主服务器返回的信息来获取主服务器本身的信息（包括run_id域记录的服务器运行ID）和主服务器下的所有从服务器信息（一个由salve字符开头的行记录）

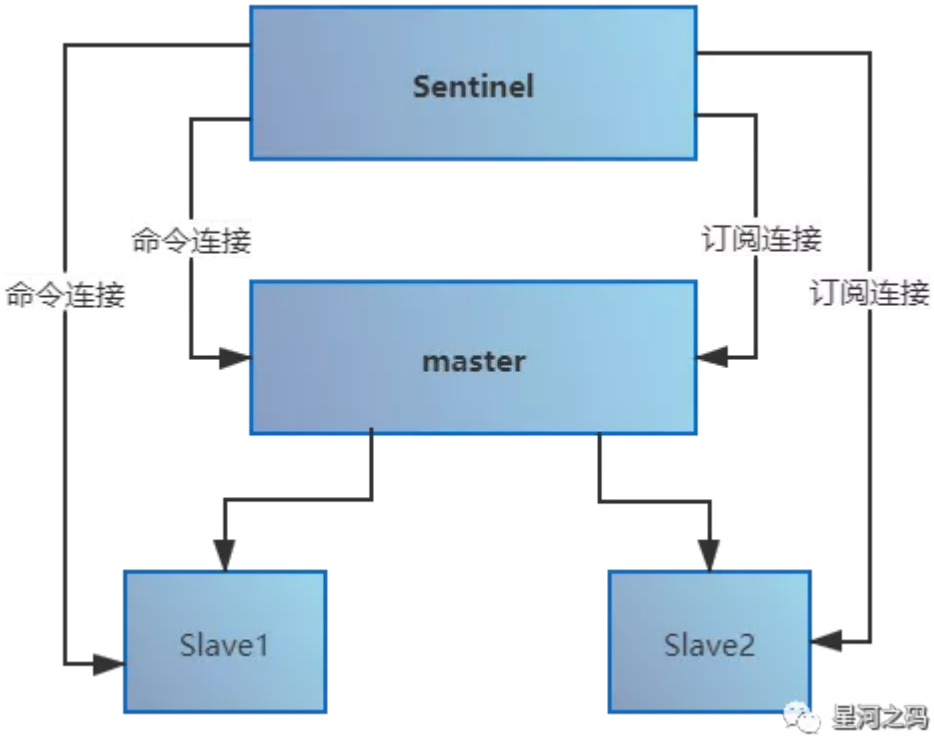
```
# Server 服务器信息
redis_version:6.0.9
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:f66e54835a6541a6
redis_mode:standalone
os:Linux 3.10.0-1127.19.1.el7.x86_64 x86_64
```

```
run_id:adc6faafdb5081218abb6cca08df6207bbc6e4f4

# Replication 主从复制信息
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=6380,state=online,offset=12352566,lag=0
slave1:ip=127.0.0.1,port=6381,state=online,offset=12352713,lag=0
master_replid:17fec3ccc1c354af16aa603c9c7569ba703fed5c
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:12352713
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:11304138
repl_backlog_histlen:1048576
```

- **获取从服务器信息**

  当Sentinel发现主服务器**有新的从服务器出现**时，Sentinel还会**向从服务器建立命令连接和订阅连接**。在命令连接建立之后，Sentinel还是默认10s一次，向从服务器发送info命令，并记录从服务器的信息。



- **创建连向其他Sentinel的命令连接**
  - 多个Sentinel可能同时监控同一个服务器，此时某一个Sentinel发送的信息会被其他Sentinel接受到，其他Sentinel就可以及时更新发送信息Sentinel和被监控服务器的信息。
  - 当Sentinel1通过频道信息发现一个新的Sentinel2时，

    这样监视同一服务器的sentinel将会形成一个相互连接的网络
    - Sentinel1会为它Sentinel2在Sentinels字典中创建一个实例，
    - Sentinel1会创建连向新Sentinel2 的连接，
    - Sentinel2也会创建连向Sentinel1的连接。

  因为Sentinel通过订阅主服务器或从服务器，就可以感知到新的Sentinel的加入，而一旦新Sentinel加入后，相互感知的Sentinel通过命令连接来通信就可以了。

  注意：Sentinel彼此之间只创建命令连接，而不创建订阅连接

四、向主/从服务器发送消息(订阅方式)

在默认情况下，**Sentinel会以每两秒一次的频率，** 通过命令连接向所有被监视的主服务器和从服务器发送以下格式的命令：

```
PUBLISH __sentinel__:hello
"<s_ip>,<s_port>,<s_runid>,<s_epoch>,<m_name>,<m_ip>,<m_port>,<m_epoch>"
```

这条命令向服务器的__sentinel__: hello频道发送了一条信息，信息的内容由多个参数组成：

- **以s_开头的参数记录的是Sentinel本身的信息**

参数	说明
s_ip	Sentinel的Ip
s_port	Sentinel的端口号
s_runid	Sentinel的runid
s_epoch	Sentinel的当前版本号

- **m_开头的参数记录的则是主服务器/从服务器的信息**

参数	说明
m_ip	Master/Slave的Ip
m_port	Master/Slave的端口号
m_runid	Master/Slave的runid
m_epoch	Master/Slave的当前版本号

  - 如果Sentinel正在监视的是主服务器，那么这些参数记录的就是主服务器的信息
  - 如果Sentinel正在监视的是从服务器，那么这些参数记录的就是从服务器正在复制的主服务器的信息

五、接收主/从服务器的频道信息

当Sentinel与一个主服务器或者从服务器建立起订阅连接之后，Sentinel就会通过订阅连接，向服务器发送命令：

```
subscribe —sentinel—:hello
```

Sentinel为主服务器创建的实例结构中的sentinels字典中除了保存本身以外，还会保存同样监视这个主服务器的其他Sentinel的资料，其中字典键为 ip:port 值为Sentinel实例结构。

当一个subscribe —sentinel—:hello频道收到一条信息时，Sentinel会分析提取出信息中的IP地址，端口号，运行ID等参数，并进行检查：

- 如果信息中记录的Sentinel运行ID和基尔兽信息的Sentinel的运行ID相同，那么说明这条信息时Sentinel自己发送的，Sentinel将丢弃这条信息，不做处理，
- 如果记录的Sentinel运行ID和接收消息的Sentinel的运行ID不相同，那么说明这条信息时监视同一个服务器的其他Sentinel发来的，接收信息的Sentinel将根据这些信息中的参数，对**sentinels字典中相应主服务器的实例结构进行更新。**

六、检测服务器下线

- **检测主观下线**
  - Sentinel每秒一次向所有与它建立了命令连接的实例(主服务器、从服务器和其他Sentinel)发送PING命令
  - 实例在down-after-milliseconds毫秒内返回无效回复(除了+PONG、-LOADING、-MASTERDOWN外)
  - 实例在down-after-milliseconds毫秒内无回复（超时）Sentinel就会认为该实例主观下线(SDown)，**在flags属性中打开SRI_S_DOWN标识，用来表示实例已经进入主观下线状态**
- **检测客观下线**

> 由于每个Sentinel 的down-after-milliseconds 可能不一致，这就会导致SentinelA认为主服务器主观下线了，但是SentinelB由于设置的参数down-after-milliseconds 比较大，SentinelB认为主服务器没下线

当一个Sentinel将一个主服务器判断为主观下线后，Sentinel会向同时监控这个主服务器的所有其他Sentinel发送查询命令

```
SENTINEL is-master-down-by-addr <ip> <port> <current_epoch> <runid>
#ip：被Sentinel判断为主观下线的主服务器的IP地址
#port：被Sentinel判断为主观下线的主服务器的端口号
#current_epoch：Sentinel当前的配置纪元，用于选举Leader Sentinel
#runid：Sentinel的运行id。运行id用于选举Leader Sentinel
```

当目标Sentinel收到源Sentinel发来的SENTINEL命令后，解析命令中的参数并根据主服务器的ip端口号检查主服务器是否下线，然后回复源Sentinel

```
<down_state><leader_runid ><leader_epoch >
#down_state：返回目标Sentinel对Master的检查结果，1代表主服务器已下线，0代表主服务器未下线
#leader_runid：Sentinel运行ID用于选举Leader Sentine
#leader_epoch：目标Sentinel的局部领头Sentinel的配置纪元，用于选举Leader Sentinel
```

根据其他Sentinel发回的SENTINEL命令回复，统计其他SENTINEL同一主服务器已下线的数量，当这一数量达到配置指定的判断客观下线所需要的数量时，**Sentinel会将主服务器实例结构flags属性的SRI_O_DOWN标识打开，标识主服务器已经进入客观下线状态。**则该主服务器就会被判定为客观下线(ODown)，并对主服务器进行故障转移。

七、哨兵leader选举

当一个主服务器被判定为客观下线后，监视这个主服务器的所有Sentinel会通过选举算法（raft），选出一个Leader Sentinel去执行failover（故障转移）操作

一致性算法Raft

Raft 是一种为了管理复制日志的一致性算法，主要用来解决分布式系统一致性问题。Raft提供了和Paxos算法相同的功能和性能，但是它的算法结构和Paxos不同。Raft算法更加容易理解并且更容易构建实际的系统。

Raft将一致性算法分解成了3模块：

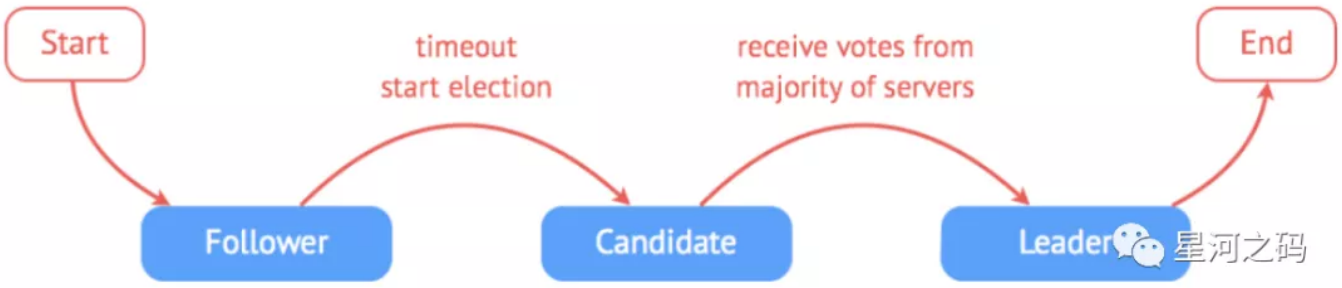
- leader选举
- 日志复制
- 安全性

Raft算法分为两个阶段，首先是选举过程，然后在选举出来的领导人带领进行正常操作，比如日志复制等，这里简单介绍一下leader选举。

- Raft协议描述的节点共有三种状态：
  - **领导者(leader)**：处理客户端交互，日志复制等动作，一般一次只有一个领导者
  - **候选者(candidate)**：候选者就是在选举过程中提名自己的实体，一旦选举成功，则成为领导者
  - **跟随者(follower)**：类似选民，完全被动的角色，这样的服务器等待被通知投票
- **Term**：Raft协议将时间切分为一个个的Term（任期），可以认为是一种**逻辑时间**。

leader选举选举过程

在Raft中，任何时候一个服务器都可以扮演下面的角色之一：Leader, Follower, Candidate



当server启动时，初始状态都是follower。**每一个server都有一个定时器，超时时间为election timeout（一般为150-300ms），如果某server没有超时的情况下收到来自领导者或者候选者的任何消息，定时器重启，如果超时，它就开始一次选举。**可以通过在线动画演示 体验这个过程。

Raft采用心跳机制触发Leader选举，系统启动后：

- 全部节点初始化为Follower，term为0。
- 节点如果收到了RequestVote或者AppendEntries，就会保持自己的Follower身份
  - **AppendEntries**：用于leader向follower发送心跳信号与更新同步日志
  - **RequestVote**：当follower转换成candidate的时候,就会向其他节点发起请求投票的请求
- 节点如果一段时间内没收到AppendEntries消息，在该节点的超时时间内还没发现Leader，Follower就 会转换成Candidate，自己开始竞选Leader。
- 一旦转化为Candidate，该节点立即开始下面几件事情：
  - 增加自己的term。
  - 启动一个新的定时器。
  - 给自己投一票。
  - 向所有其他节点发送RequestVote，并等待其他节点的回复。
- 如果在计时器超时前，节点收到多数节点的同意投票，就转换成Leader。同时向所有其他节点发送 AppendEntries，告知自己成为了Leader。
- 每个节点在一个term内只能投一票，**采取先到先得的策略**，Candidate前面说到已经投给了自己， Follower会投给第一个收到RequestVote的节点。

- Raft协议的定时器采取随机超时时间，这是选举Leader的关键。
- 在同一个term内，先转为Candidate的节点会先发起投票，从而获得多数票。

Sentinel选举leader流程

- 某Sentinel认定master客观下线后，该Sentinel会先看看自己有没有投过票，如果自己已经投过票给其他Sentinel了，在一定时间（election timeout）内自己就不会成为Leader。
- 如果该Sentinel还没投过票，那么它就成为Candidate。
- Sentinel需要完成几件事情：
  - 更新故障转移状态为start
  - 当前epoch加1，相当于进入一个新term，在Sentinel中epoch就是Raft协议中的term。
  - 向其他节点发送 is-master-down-by-addr 命令请求投票。命令会带上自己的epoch。
  - 给自己投一票（leader、leader_epoch）
- 当其它哨兵收到此命令时，可以同意或者拒绝它成为领导者；（通过判断epoch）
- Candidate会不断的统计自己的票数，直到发现认同自己成为Leader的票数超过一半而且超过它配 置的quorum，这时它就成为了Leader。
- 其他Sentinel等待Leader从slave选出master后，检测到新的master正常工作后，就会去掉客观下 线的标识。

八、故障转移

当选举出Leader Sentinel后，**Leader Sentinel会对下线的主服务器执行故障转移**操作，主要有三个步骤：

- 它会将失效Master的其中一个Slave升级为新的 Master，并让失效 Master的其他Slave改为复制新的Master
- 当客户端试图连接失效的 Master 时，集群也会向客户端返回新 Master 的地址，使得集群可以使用新的 Master 替换失效 Master 。

- Master 和 Slave 服务器切换后， Master 的 redis.conf 、 Slave 的 redis.conf 和 sentinel.conf 的配置文件的内容都会发生相应的改变。

> Master 主服务器的 redis.conf配置文件中会多一行 replicaof 的配置，sentinel.conf 的监控目标会随之调换

九、主服务器的选择

哨兵leader根据以下规则**从客观下线的主服务器的从服务器中选择出新的主服务器。**

- 过滤掉主观下线的节点
- 选择slave-priority最高的节点，如果由则返回没有就继续选择
- 选择出复制偏移量最大的系节点，因为复制偏移量越大则数据复制的越完整，如果由就返回了，没有就继续
- 选择run_id最小的节点，因为run_id越小说明重启次数越少