Redis Sentinel

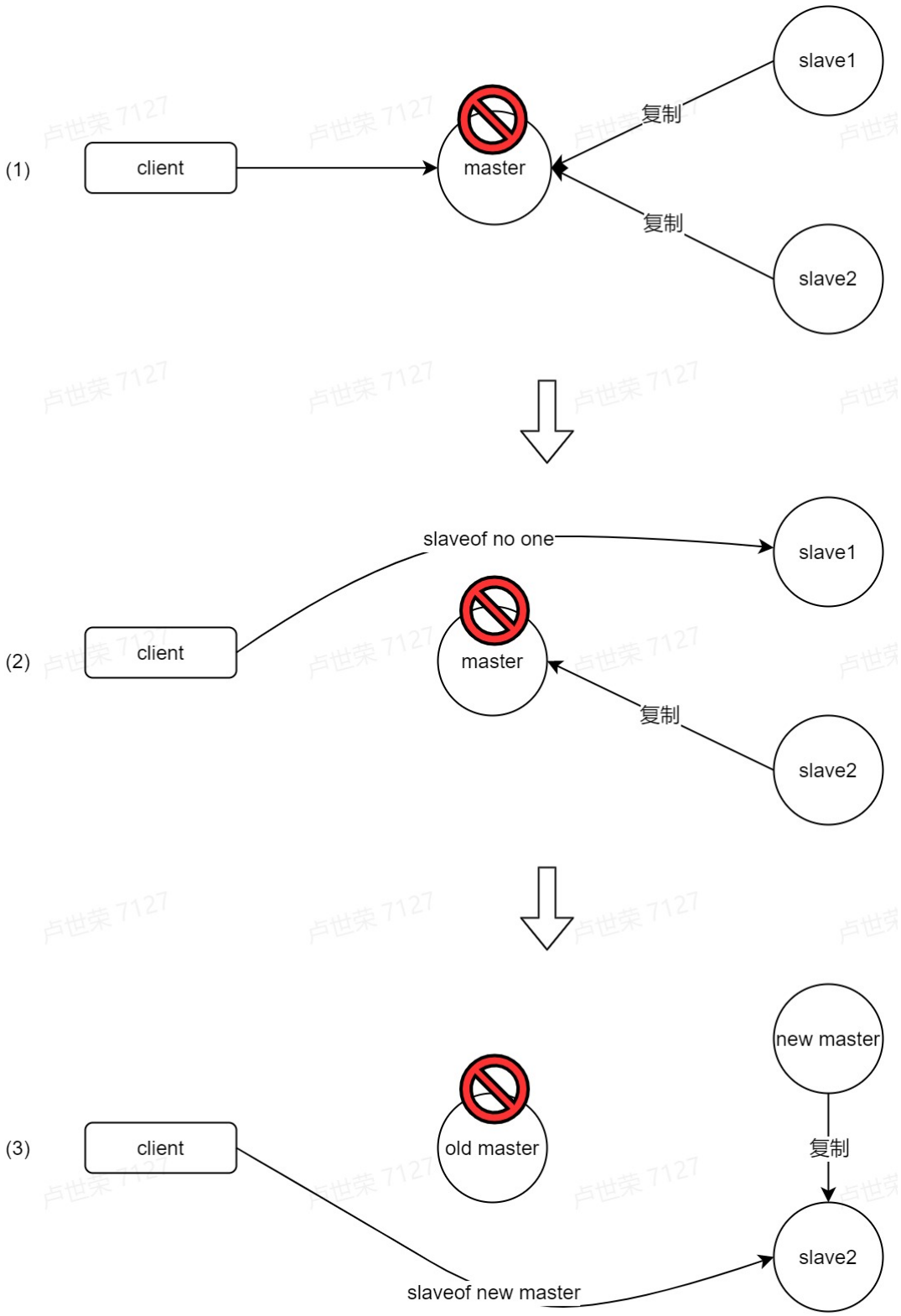
概述

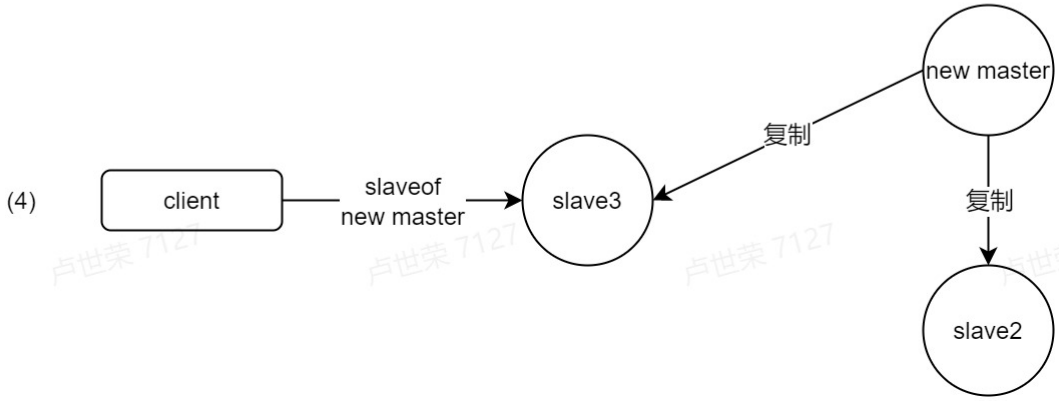
Redis sentinel是 redis 的高可用解决方案，在redis2.8之后的版本正式生产可用。Redis sentinel通过引入sentinel节点（也是一个redis进程）集合来对redis的数据节点集合进行监控，发现故障节点并自动进行故障转移，从而提高生产环境下redis的高可用性。

解决问题

常规的redis主从复制结构模式下，主节点可以将数据同步给从节点，这样做的原因是：一旦主节点发生了故障，从节点可以直接作为后备顶替上来成为新的主节点，并尽最大努力保证数据不丢失（由于主从复制之间有一定的时延，只能保证最终一执行）。 为了实现这一点，需要运维人员实时地去监控主节点的状态，通常情况下是很难及时地发现主节点的异常状态。并且，一旦发现主节点发生了故障，还需要运维人员手动的去进行故障转移，故障转移的实时性和准确性上都无法保障。以下为人工干预进行故障转移所需要的步骤：

1. 检测到某个主节点发生故障，客户端连接主节点失败，从节点与主节点间的复制失败。
2. 选择一个从节点，执行slave of no one，使其成为主节点。
3. 通知应用方，使其重新初始化客户端，连接新晋主节点。
4. 对其他从节点执行salve of new master，让这些从节点去复制新晋主节点的数据。
5. 监控之前的主机点，待其恢复后，执行slave of new master，让其成为新晋主节点的从节点，并开始执行复制操作。

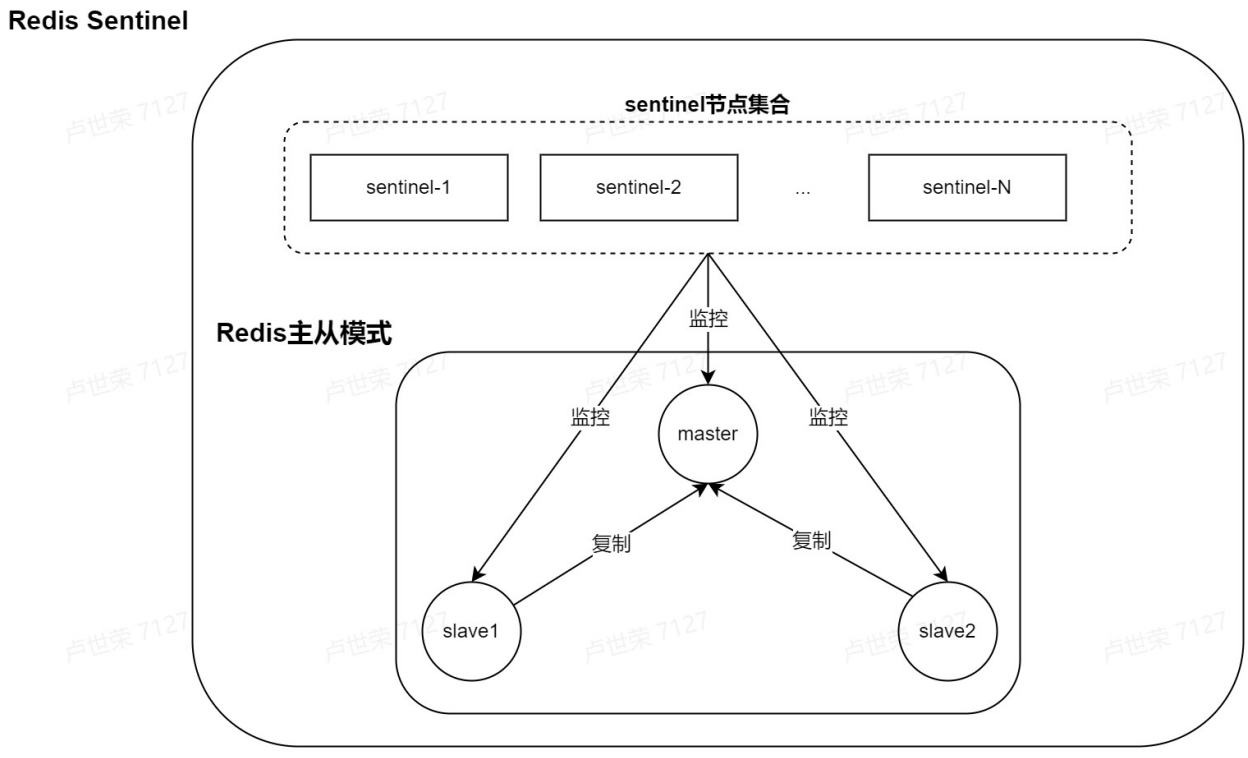




高可用性

从上图中可以看出手动进行故障转移需要运维人员手动执行一系列繁琐的操作，为了能够自动实现故障发现和故障转移，从而实现真正的高可用，redis给出了redis sentinel的解决方案。

redis sentinel通过引入一系列redis sentinel节点来对redis数据节点进行监控，当某个sentinel节点发现数据节点出现故障时，会对数据节点做主观下线操作，如果主观下线的是主节点，这个sentinel节点还会和redis sentinel节点集合中的其它sentinel节点进行协商，当大多数的sentinel节点认为某个节点发生了故障，会判定这个数据节点为客观下线，并继续协商投票选出一个sentinel节点来负责对故障节点进行故障转移的工作，这个过程都是自动化完成，实现了真正的高可用性。



从图中的拓扑结构上看，sentinel节点集合会定时对所有的数据节点进行监控，当发生故障，会自动实现图2-1中的操作。 Redis sentinel仅仅是在普通的redis主从复制模式增加

了一系列的sentinel节点，用于监控主从模式中的redis数据节点，并没有对redis的数据节点进行额外的处理。

实现原理

三个定时任务

1. 每隔10s，sentinel会向主节点和从节点发送info命令来获取最新的数据节点拓扑结构，用于及时感知节点之间的变化。就是因为有了这个定时任务，在配置redis sentinel节点的时候就只需要配置主节点的信息，从节点的信息可以由这个定时任务获取到。
2. 每隔2s，sentinel节点会向__sentinel__:hello频道发送自身的sentinel信息以及对主节点的故障判断，sentinel节点集合中的所有节点都会订阅这个频道。当有新的sentinel节点加入进来时，通过订阅这个频道，各个sentinel节点可以及时的获取到新加入的sentinel节点信息。此外，通过这个频道sentinel节点之间可以相互交换对主节点的状态信息，也就是与其它sentinel节点交换对主节点的客观下线判断，作为后续主节点客观下线以及领导者选举的依据。
3. 每隔1s，每个sentinel节点会向主节点、从节点和其它的sentinel节点发送ping命令做心跳检测，来确认这些节点是否可达，如果节点不可达，则标记节点下线。这个定时任务是判断节点故障的重要依据。

主观下线和客观下线

1. 主观下线：利用第三个定时任务，可以对所有节点进行故障检测。当sentinel向其它节点发送ping后，超过参数down-after-milliseconds配置的时间后仍没有收到有效的回复，就会判定这个节点为故障的，并标记这个节点下线，这个行为就叫做主观下线。主观下线是某一个sentinel节点对另外的节点的主观方面的故障判断，存在误判的可能性，因此不能作为节点实际下线的判断依据。
2. 客观下线：正如之前所说，主观下线存在误判的可能性，所有需要多个sentinel节点共同参与商讨对某个节点的下线判断，从而确定该节点是否真的处于不可达的状态。具体操作是：sentinel节点会向其它sentinel节点发送命令is-master-down-by-addr询问其他节点对主节点的故障判断，只有当达到quorum个sentinel节点对主节点做出故障判断时，才会执行对主节点的客观下线。

> - down-after-millisecondes：该参数可以在redis sentinel节点启动配置文件中设置。这个参数设置得越大，则误判节点故障的概率就越小，但这也意味着故障转移的操作延后了，降低了对故障节点的反应速度。反之，该参数设置得越小，虽然提高了对故障节点的反应速度，但是对故障判断的错误率也随之升高。
> - quorum：该参数也是在sentinel节点启动配置文件中设置的，可用于客观下线的判断和sentinel领导者的选举。如果该参数设置得太小，则客观下线的条件就越宽松，反之越严格。
> - is-master-down-by-addr：这个命令除了可以用于交换sentinel节点之间对主节点的主观下线判断，还可以用于sentinel领导者的选举，不同的行为由不同的参数决定。

Sentinel领导者选举

当sentienl节点对主节点做出了客观下线的判断后，不会立即执行故障转移的操作，此时sentinel节点结合会选举出一个领导者，由该领导者sentinel节点来执行后续的故障转移。Redis使用了Raft算法来选举领导者，这里见到介绍整个选举的流程。

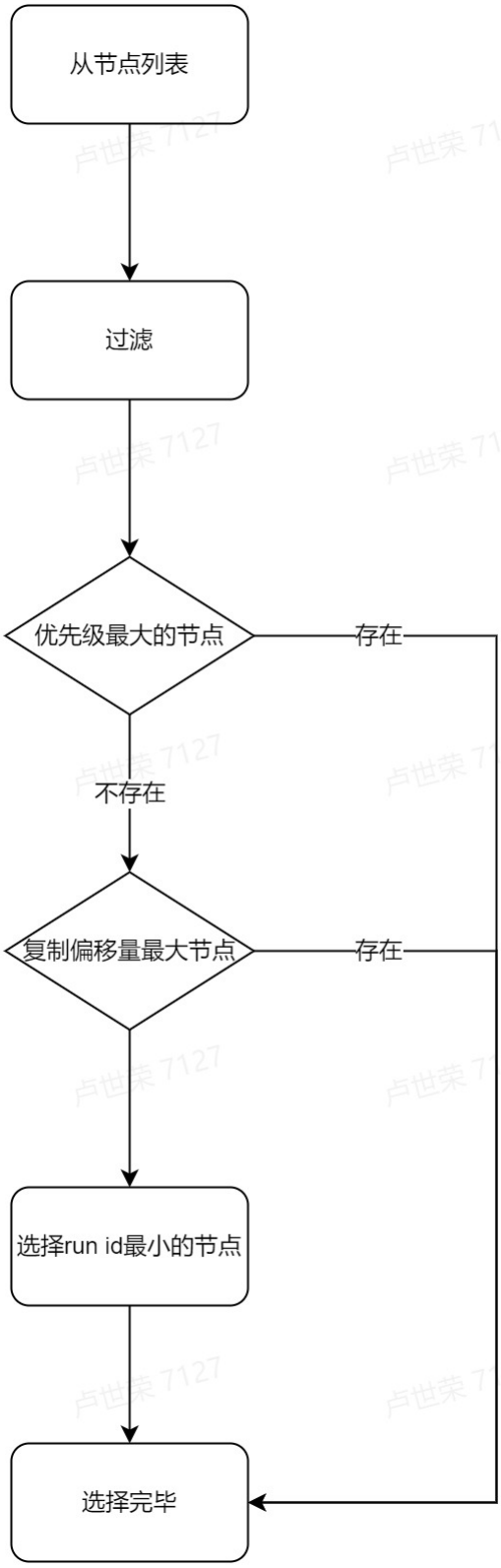
1. 只有没有标记为主观下线的sentinel节点才有资格成为候选人。sentinel节点通过向其它sentinel节点发送is-master-down-by-addr命令寻求对方将选票投个自己。
2. 收到命令的sentinel节点会判断自身在本轮(epoch)的投票中自己的选票是否还存在，如果选票还在，就同意对方成为领导者，反之拒绝。
3. 此时如果sentinel节点发现自己获得的选票达到max(quorum, num(sentinels)/2)，那么它将成为领导，由于每轮选举每个sentinel节点都只有一张票，因此只有人拿到半数以上的票数，其它人就不可能拿到半数以上的票。
4. 如果本轮投票没有选出领导者，开启下一轮选举。

为了达到半票以上，sentinel节点的个数至少为3个，并且为了最小化部署，通常选择sentinel节点的个数为奇数

故障转移

这里介绍一些sentinel进行故障转移中的细节，整个流程见下图。

1. sentinel会从过滤掉不健康的候选从节点，包括主观下线、5s没有回复sentinel节点ping响应、与主节点失联超过down-after-milliseconds*10秒的节点
2. 选择从节点优先级最高的从节点，如果存在直接返回
3. 选择复制偏移量最大的从节点，复制偏移量代表了主从同步的完整度
4. 选择run id最小的从节点



部署

Redis sentinel节点和普通的数据节点的启动并没有什么不同，都是读取配置文件，然后运行redis server，它们之间的区别在于配置文件内容。这里简单介绍如何搭建图2-2的redis sentinel服务。

1. 启动主节点

```
# redis-6379.conf

port 6379                       # 端口号
daemonize yes                   # 守护进程运行
logfile "6379.log"              # 日志名
dbfilename "dump-6379.rdb"      # 持久化文件名
dir "/opt/redis/data"           # 工作目录，存放rdb文件
```

2. 启动两个从节点

```
# redis-6380.conf

port 6380                       # 端口号
daemonize yes                   # 守护进程运行
logfile "6380.log"              # 日志名
dbfilename "dump-6380.rdb"      # 持久化文件名
dir "/opt/redis/data"           # 工作目录，存放rdb文件
slaveof 127.0.0.1 6379          # 作为6379的从节点
```

```
# redis-6381.conf

port 6381                       # 端口号
daemonize yes                   # 守护进程运行
logfile "6381.log"              # 日志名
dbfilename "dump-6381.rdb"      # 持久化文件名
dir "/opt/redis/data"           # 工作目录，存放rdb文件
slaveof 127.0.0.1 6379          # 作为6379的从节点
```

3. 部署sentinel节点，

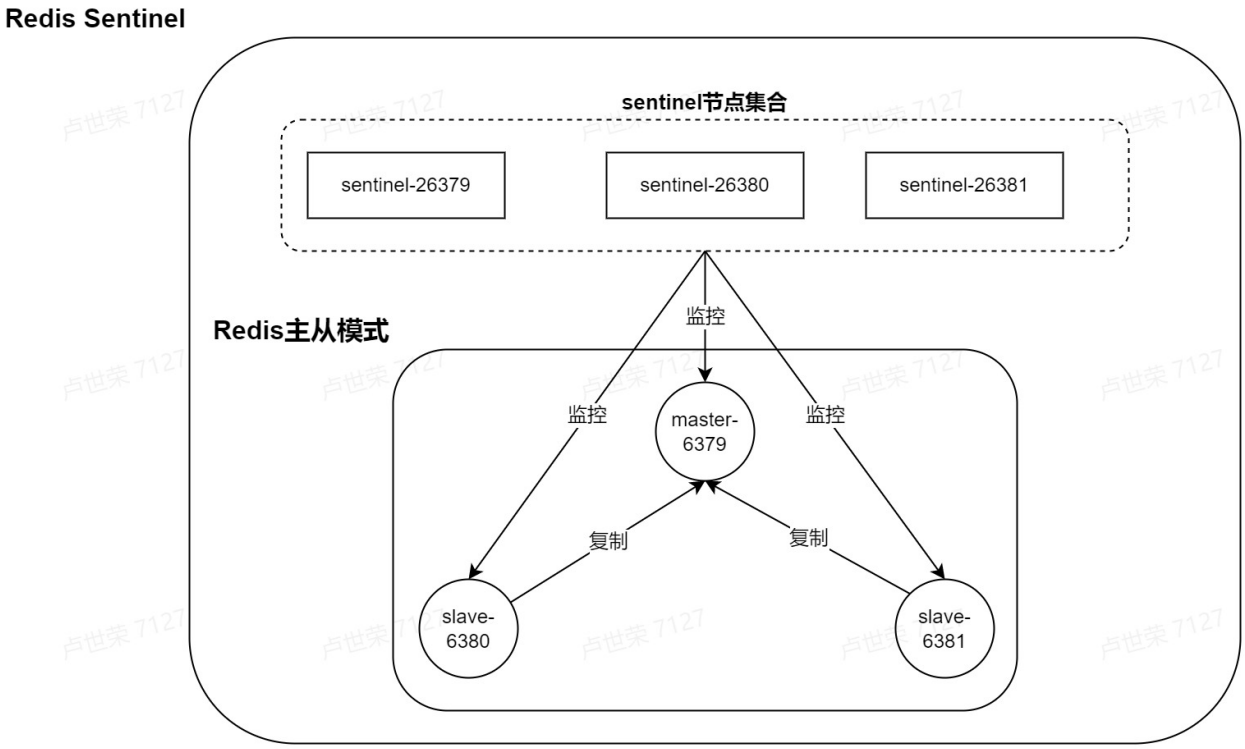
```
# redis-sentinel-26379.conf

port 26379                      # 端口号
daemonize yes                   # 守护进程运行
logfile "26379.log"             # 日志名
dir "/opt/redis/data"           # 工作目录，存放rdb文件
sentinel monitor master 127.0.0.1 6379 2      # 监控127.0.0.1:6379的主节点，名字为master，quorum为2
sentinel down-after-milliseconds master 30000 # 设置下线时间为30s
sentinel parallel-syncs master 1              # 设置故障转移后同时进行复制新主节点的从节点的个数
sentinel failover-timeout master 180000       # 设置故障转移超时时间为180s

# redis-sentinel-26380.conf
port 26380
...

# redis-sentinel-26381.conf
```

```
port 26381
...
```



Redis Cluster

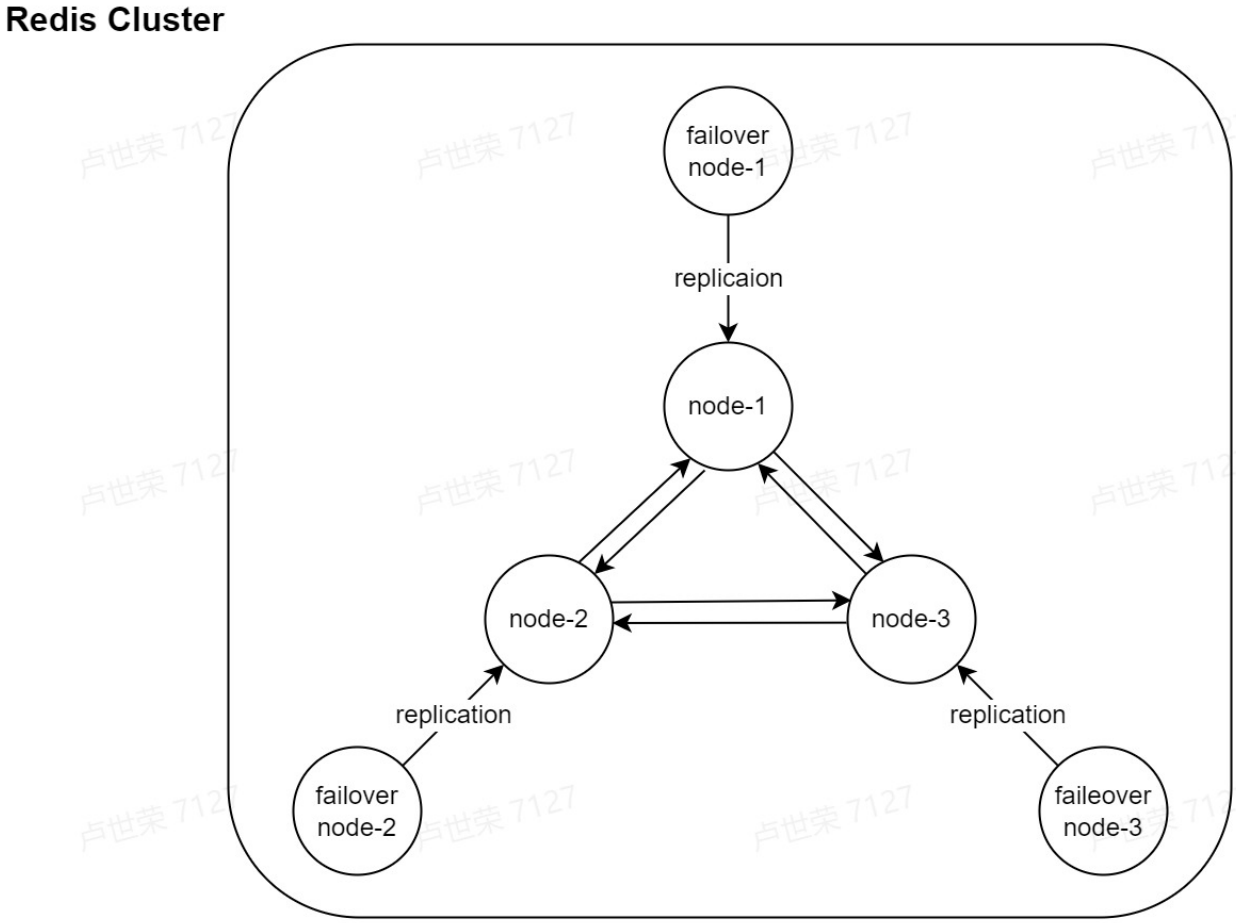
概述

Redis Cluster是Redis官方的分布式解决方案，在redis3.0之后生产环境正式可用。在实现了高可用的同时，分布式的架构极大扩展了redis的存储、读写能力。由于redis cluster所涉及的内容太多，本文只会对redis cluster中的节点通信、集群伸缩和故障转移进行介绍，其它部分包括集群搭建、请求路由、集群运维等内容可以参考《Redis开发与运维》一书。

解决问题

1. 单集群主节点的写能力受到限制，传统主从模式和redis sentinel无法提升写性能。
2. 单集群主节点的存储能力受到限制。

拓扑结构



从中可以看出redis cluster采用的是去中心化的分布式架构，它们之间通过不断的交换信息来获取集群整体的状态（图中集群的从节点也参与信息交换，为简化结构，忽略它们之间的连线)。其中redis cluster中的节点也区分主从，从节点负责处理当主节点出现故障时的自动故障转移，**默认情况下从节点不支持任何的读写操作，仅作为主节点的一个“热备”存在。**

> 注意：在Redis cluster模式下，成为从节点的salveof命令将会失效，应使用cluster replication命令

数据分区

分布式数据库首先要解决把整个数据集按照分区规则映射到多个节点的问题，即把数据集划分到多个节点上，每个节点负责整体数据的一个子集。 Redis cluster采用哈希分区规则，这里介绍两种常用的哈希分区规则

1. 哈希取余分区 这种是最简单的哈希分区方式，通过计算key的哈希值对目前分区大小进行取余，来决定数据应该存储在哪一个分区节点上，这种分区方式优点在于简单，但缺点也十分明显：当需要改变分区的大小时，也就是对分布式数据库进行扩缩容时，数据到分区节点的映射关系将会打乱，这会导致有大量的数据需要迁移。例如，假设有10个节点，数据key通过哈希取余的方式进行映射：hash(key)%10，当节点缩减到9时，此时通过hash(key)%9计算得到的值就会发生变化，这几乎导致了所有数据到节点都失效了，需要进行大量的数据迁移。
2. 一致性哈希算法 一致性哈希算法就是为了解决普通的哈希取余算法带了的数据迁移的问题。通过计算每个节点的哈希值并将值映射到一个环上（哈希环，大小一般取

2^32），当进行数据读写的时候，会先计算出数据key的哈希值，然后放置到哈希环上进行顺时针查找，此时遇到的第一个节点就是该key所将要映射节点。 这种方案的优点在于当集群进行扩缩容时，影响到的只有扩缩容节点顺时针相邻的节点。例如增加一个节点，那么只需要将其顺时针相邻的节点中部分数据迁移至新增节点中。反之如果是删除节点，则只需要将删除节点中的数据迁移至顺时针相邻节点中。

一致性哈希算法也存在一些问题：

- 当使用少量节点时，若节点哈希在环上分布不均匀，节点变化有肯能将大范围影响哈希环中数据映射，因此这种方式不适合少量数据节点的分布式方案。
- 普通的一致性哈希分区无法保证数据和负载的均衡，需要增加虚拟节点。

因此，在此基础之上，redis使用虚拟槽的分区规则。其主要思想就是每个节点负责一部分虚拟槽，而数据经过哈希计算后直接映射到对应的槽上，每个槽又维护着一定量的数据。 Redis cluster使用的槽范围为0～16383，共计16384个槽。数据到槽的映射可以通过hash(key)&16383的方式进行计算。使用虚拟槽的方式，可以很好的解决一致性哈希的问题：通过均匀分配给每个节点虚拟槽，当节点较少时，至少可以保证只有16384/num(node)个槽中的数据需要进行迁移。此外，均匀分配的槽可以比较好的保证负载均衡。

> redis cluster是去中心化的分布式架构，它们之间通过gossip进行通信，因此每个redis节点都保存集群中其他节点的槽信息。

节点间通信

在分布式存储中需要提供维护节点元数据信息的机制，所谓元数据是指：节点负责哪些数据，是否出现故障等状态信息。常见的元数据维护方式分为：集中式和P2P方式。Redis集群采用P2P的Gossip协议（是一种去中心化的协议，见图3-1的拓扑结构），Gossip协议工作原理就是节点彼此不断通信交换信息，一段时间后所有的节点都会知道集群完整的信息，这种方式类似流言传播。

Gossip协议

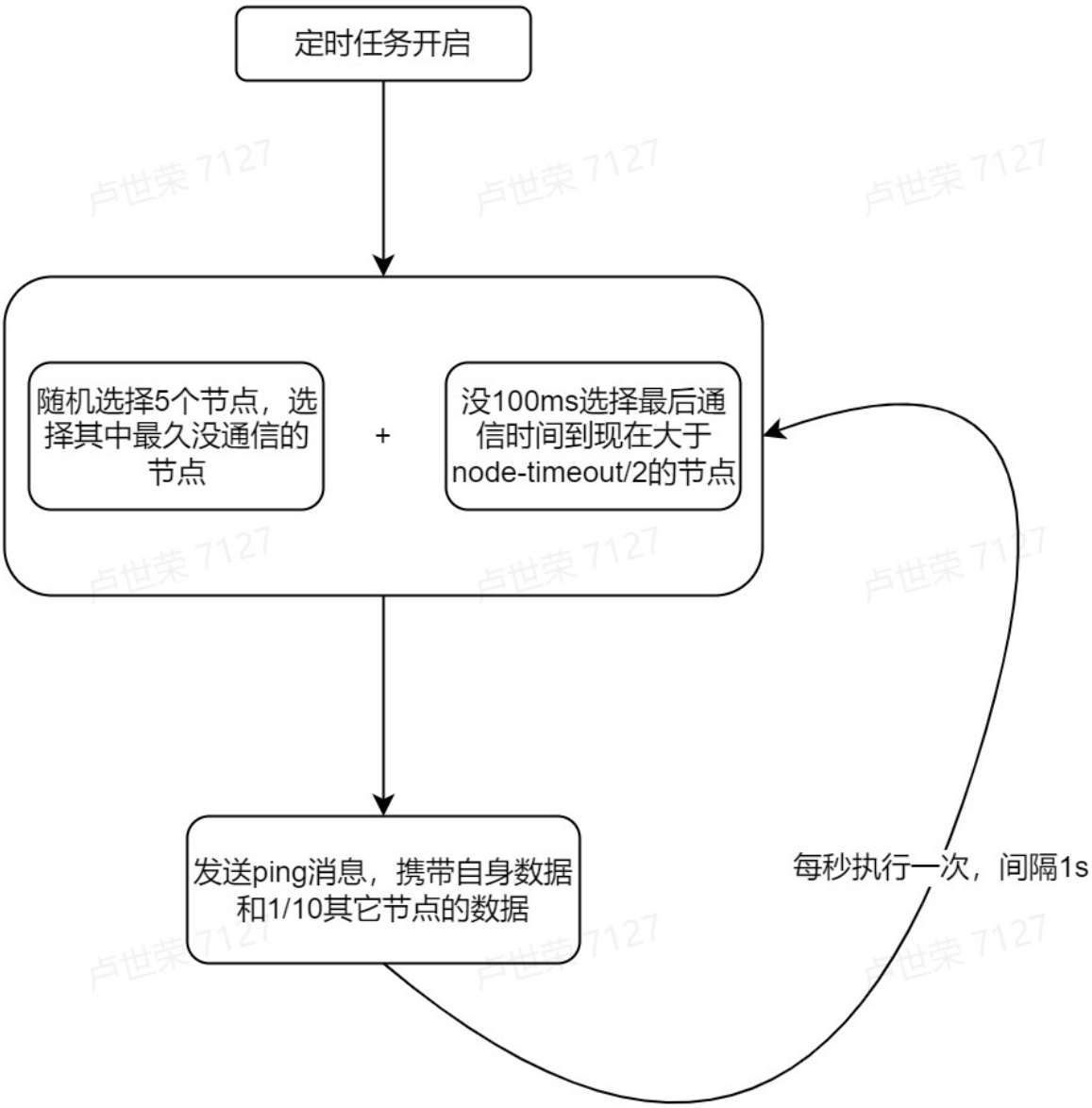
Gossip协议的主要职责就是信息交换。信息交换的载体就是节点彼此发送的Gossip消息。常用的Gossip消息可分为：ping消息、pong消息、meet消息、fail消息等。

- ping消息：集群内交换最频繁的消息，集群内每个节点每秒向多个其他节点发送ping消息，用于检测节点是否在线和交换彼此状态信息，ping消息中封装了自身节点和部分其他节点的状态数据。
- meet消息：用于通知新节点加入。发送该消息的节点通知接收节点自己要加入到当前集群，meet消息通信正常完成后，接收节点会加入到集群中并进行周期性的ping、pong消息交换，从而把自己的加入到集群的信息传播到整个集群。
- pong消息：当接收到ping、meet消息时，作为响应消息回复给发送方确认消息正常通信。pong消息内部也封装了自身状态数据。节点也可以向集群内广播自身的pong消息来通知整个集群对自身状态进行更新。

- fail消息：当节点判定集群内另一个节点客观下线时，会向集群内广播一个fail消息，其他节点接收到fail消息之后把对应节点更新为客观下线状态，从节点收到fail消息会执行自动故障转移。

通信节点选择

虽然Gossip协议的信息交换机制具有天然的分布式特性，但它是有成本的。由于内部需要频繁地进行节点信息交换，而ping/pong消息会携带当前节点和部分其他节点的状态数据，势必会加重带宽和计算的负担。Redis集群内节点通信采用固定频率（定时任务每秒执行10次）。因此节点每次选择需要通信的节点列表变得非常重要。通信节点选择过多虽然可以做到信息及时交换但成本过高。节点选择过少会降低集群内所有节点彼此信息交换频率，从而影响故障判定、新节点发现等需求的速度。因此Redis集群的Gossip协议需要兼顾信息交换实时性和成本开销，通信节点选择的规则如图所示。



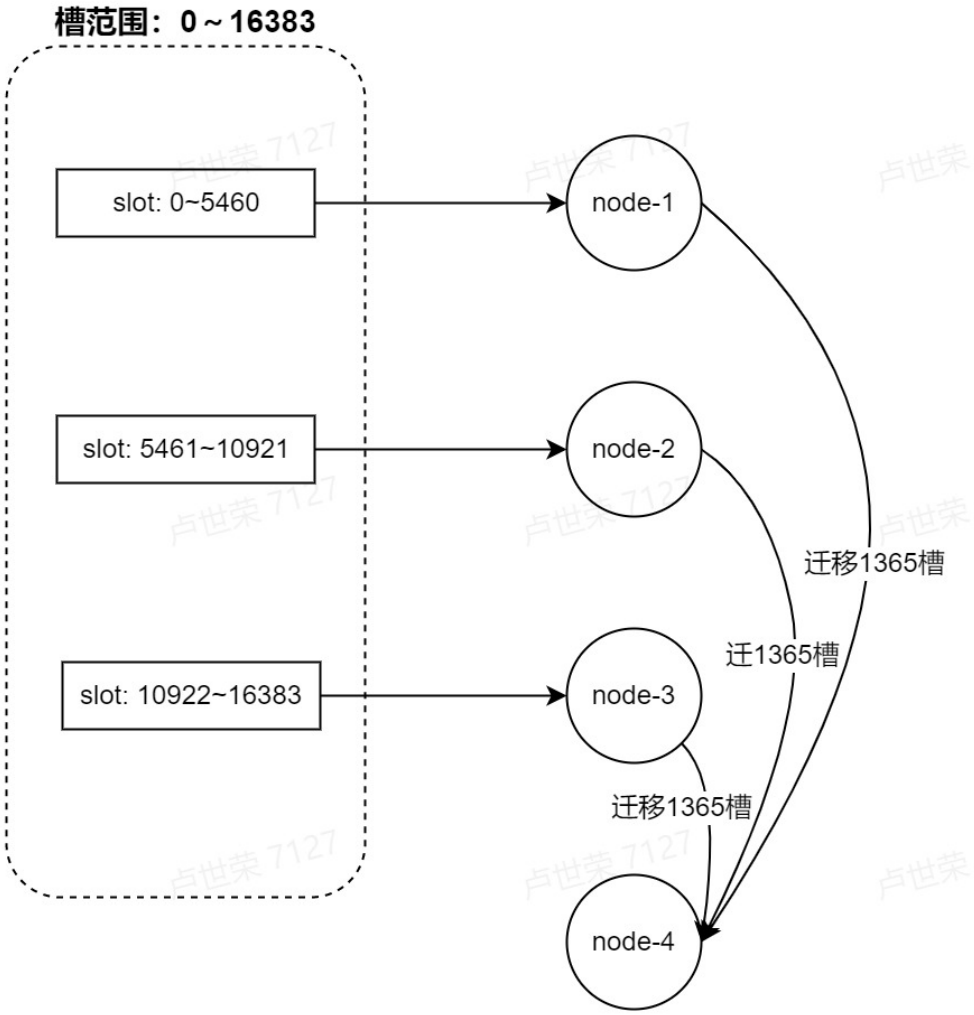
从图中可以得出每个节点每秒会向1+10*num(最后通信时间>node_timeout/2)个节点发送ping消息来告知自身和其它部分节点的信息。 当集群规模不断增大时，集群间会产生大量的gossip消息，这限制cluster集群的大小。

集群扩缩容

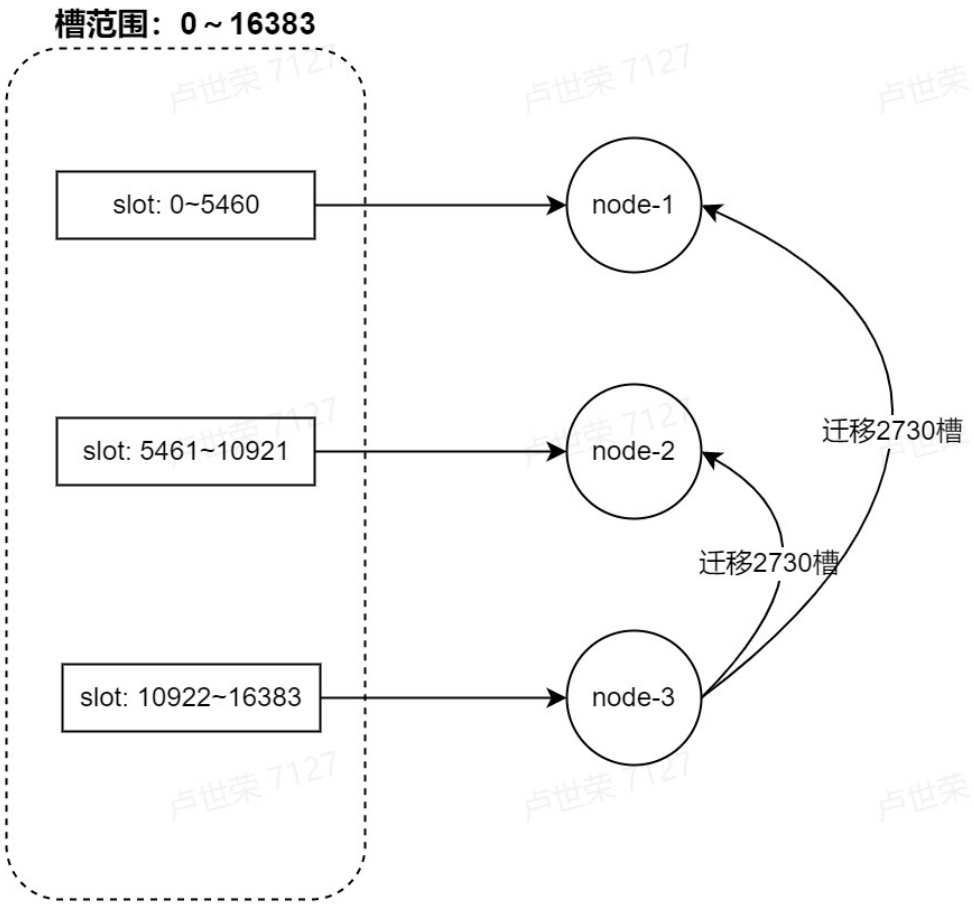
Redis cluster提供了灵活的集群扩缩容方案，主要得益于redis抽象了虚拟槽这一存储单元，集群的扩缩容的底层操作实际上就是虚拟槽在各个redis节点之间的移动，如图所

示。

扩容


槽范围：0～16383
slot: 0~5460
slot: 5461~10921
slot: 10922~16383
node-1
node-2
node-3
node-4
迁移1365槽
迁1365槽
迁移1365槽

缩容


槽范围：0～16383
slot: 0~5460
slot: 5461~10921
slot: 10922~16383
node-1
node-2
node-3
迁移2730槽
迁移2730槽

- 扩容集群 扩容集群分为一下几步：
    - 准备新节点，为了高可用，至少准备两个节点，启动这两个redis-server，新启动的节点作为孤儿节点运行，没有与任何节点通信。
    - 加入集群，新节点向集群中任意节点发送meet消息，请求加入到节点，集群中节点收到消息后回复pong消息，并在与集群中其它节点正常的ping/pong通信时将新节点的信息传播到整个集群。
    - 执行槽迁移，给其中一个节点分配槽，并开始执行节点与节点之间的槽中数据的迁移。
    - 执行cluster replication命令，使新节点中的其中一个成为另一个具有槽的节点的从节点。
- 收缩集群 收缩集群的操作与扩容集群相反：
    - 如果需要将主从节点一起下线，需要先下线从节点，否则主节点下线后从节点会被分配给从节点最少的主节点并执行复制操作，增加了额外的带宽压力。
    - 下线主节点，将槽迁移至集群中的其它的节点。
    - 对集群中其它的节点执行cluster forget命令来忘记下线节点，忘记节点后，节点就不会向下线节点发送gossip消息。
    - 停止下线节点进程。

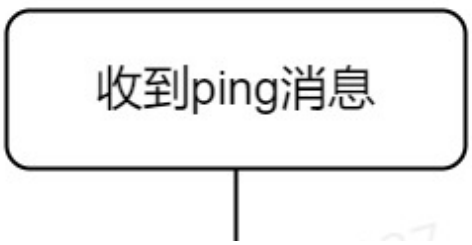
故障转移

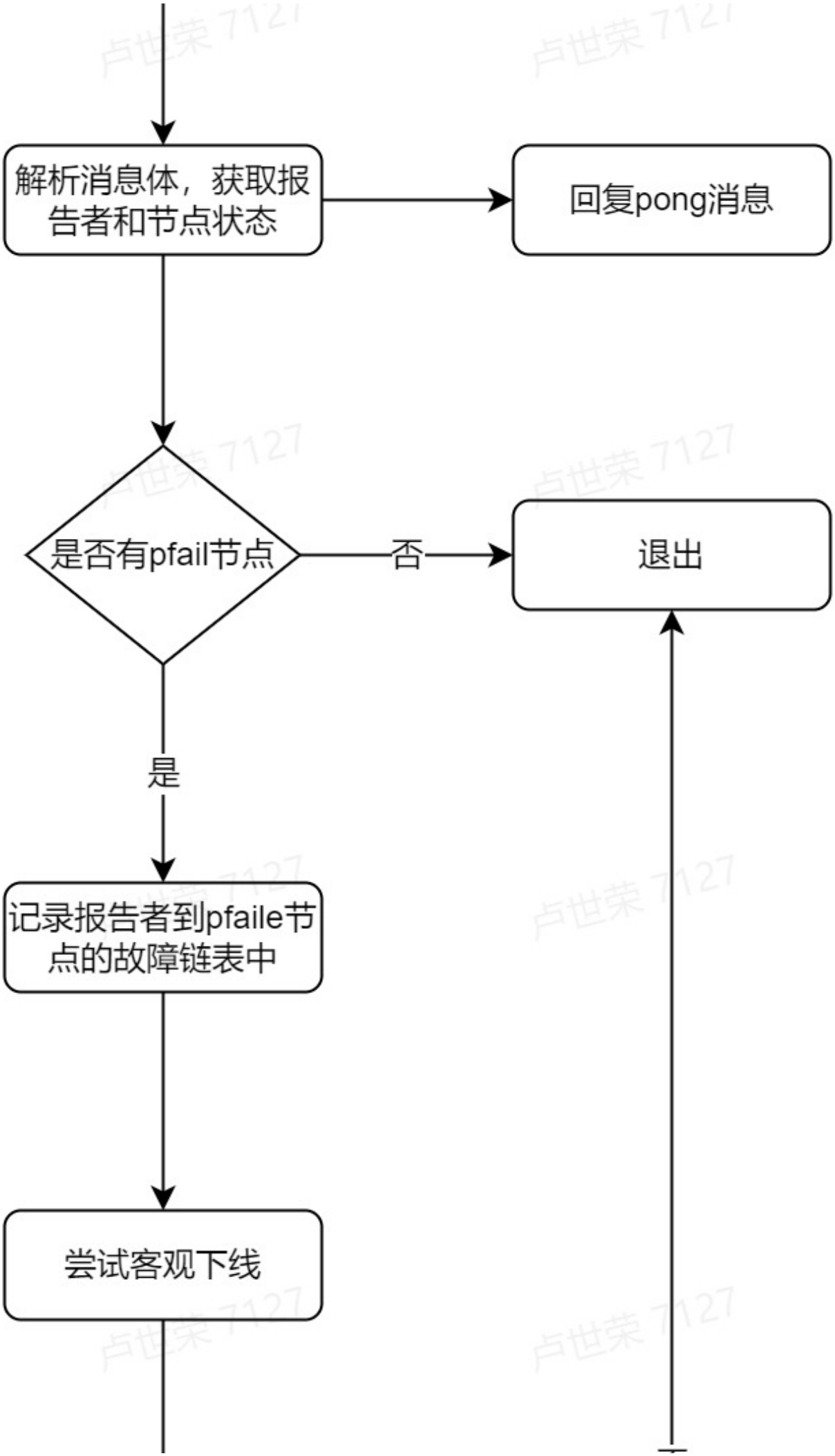
故障发现

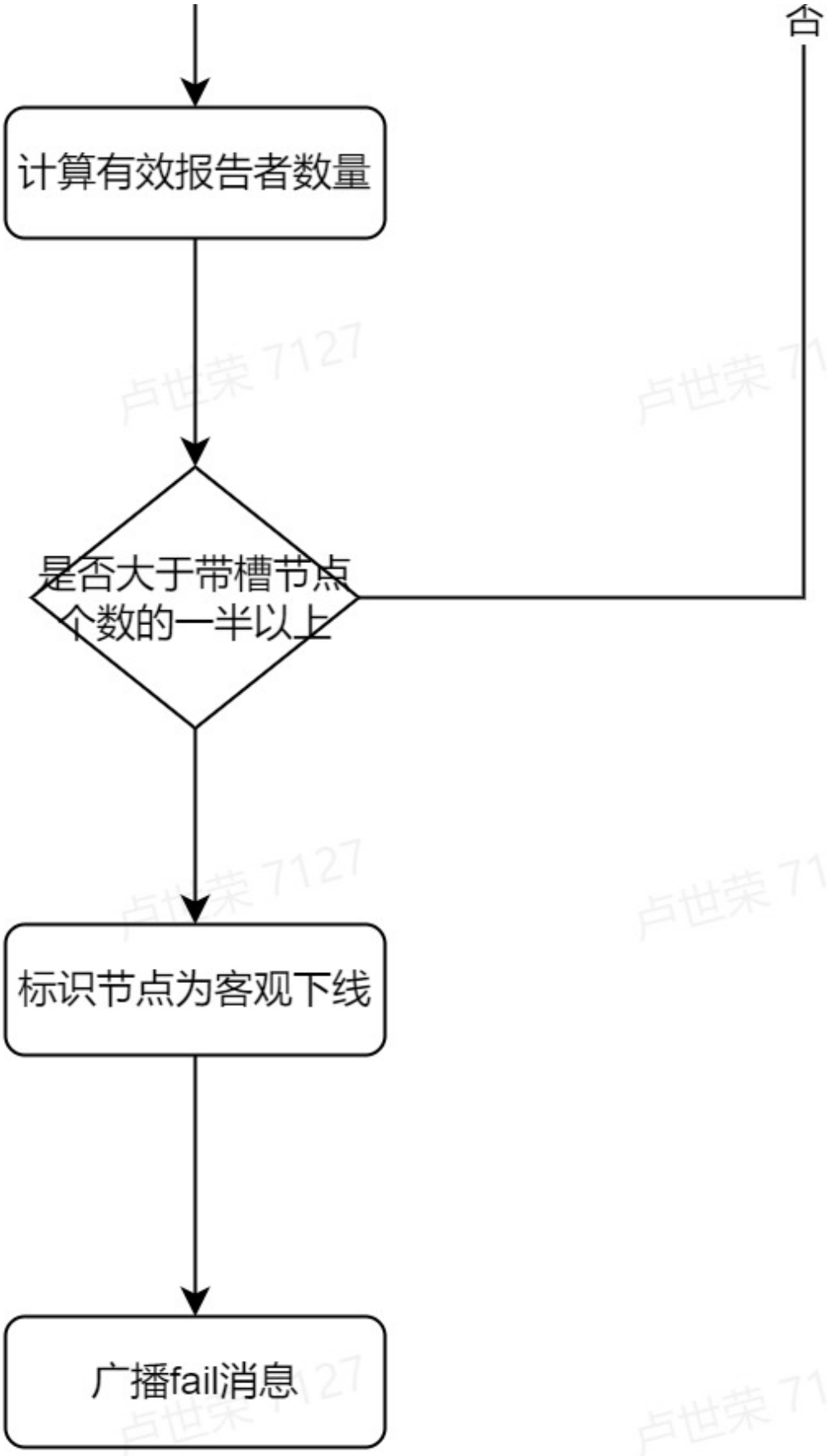
redis cluster中的ping/pong消息除了进行节点间信息交流，还可以检测节点是否可达。

1. 主观下线：集群中每个节点都会定期地向其它节点发送ping消息告诉自身节点和保存的其它节点的信息，收到消息的节点也会回复pong信息来告诉对方自己的节点和保存的其它节点的信息。如果在配置cluster-node-timeout时间一直没收到pong信息，则会认为这个节点存在故障，标识这个节点为主观下线（pfail）并保存在本地。
2. 客观下线：当某个节点判断另一个节点主观下线后，后续的ping消息会将这个主观下线判断传播到集群中的其它节点，其它节点收到这个主观下线报告后，会维护一个链表记录下线节点的报告者，并尝试进行客观下线。当这个下线节点链表中的报告者的数量大于num(nodes)/2 + 1时，执行客观下线（fail），并向集群中广播fail信息。

这个流程见图:




解析消息体，获取报告者和节点状态
回复pong消息
是否有pfail节点
否
退出
是
记录报告者到pfaile节点的故障链表中
尝试客观下线



这里进行决策的节点都是带有槽的主节点，不带有槽的节点和从节点不参与客观下线的决策。

故障恢复

故障节点被标志为客观下线后，为了保证系统的高可用，必须从从节点中选出一个成为新的主节点。从节点的内部定时任务发现其主节点被标志为客观下线后，将会触发故障恢复流程：

1. **资格检查**：每个从节点都要检查最后与主节点断线时间，判断是否有资格替换故障的主节点。如果短线时间过长，不会发起投票选举。
2. **准备选举延迟时间**：当从节点符合故障转移资格后，更新触发故障选举的时间，只有到达该时间后才能执行后续流程。从节点根据自身复制偏移量设置延迟选举时间，保证复制延迟低的从节点优先发起选举。
3. **发起选举**：递增全局配置epoch，并赋值给从节点本地epoch，并广播选举消息。
4. **选举投票**：每个具有槽的主节点在每个epoch内有一张选票，在收到从节点的选举消息后，如果选票还在，就会投出这一票。当从节点收到的票数达到N/2+1，从节点就可以执行替换主节点的操作，其中N为具有槽的主节点个数。为了能够达到选举票数，必须保证N大于等于3。如果在一个epoch内从节点没有获取到足够的选票，会更新epoch，并发起下一轮投票。
5. **替换主节点**：取消复制主节点，接管主节点的槽并广播pong消息通知集群当前节点以晋升为主节点并接管了原主节点的槽。

> epoch的作用：
>
> 1. 全局配置epoch用于标志当前的选举的轮数，只有在同一个epoch中的选票才会进行统计。
> 2. 另外，当选举完成后，新的主节点的本地epoch会设为当前的全局配置epoch，因此全局配置epoch也表示当前集群的最新版本。
> 3. 具有更大的本地epoch的主节点具有跟高的优先级，当故障主节点恢复后，由于已经进行了故障转移，新的主节点和老的主节点具有同样的虚拟槽，当虚拟槽信息在集群中传播时，以节点本地opoch最大的主节点为准。

常见缓存问题及解决方案

布隆过滤器