链滴

Redis 笔记二

作者：matthewhan

原文链接：https://ld246.com/article/1625799019553

来源网站：链滴

<h2 id="Q-什么是主从模式">Q：什么是主从模式</h2>
那我们总说的 Redis 具有高可靠性，又是什么意思呢？
其实，这里有两层含义：一是数据尽量少丢失，二是服务尽量少中断。AOF 和 RDB 保证了前者而对于后者，Redis 的做法就是增加副本冗余量，将一份数据同时保存在多个实例上。即使有一个实出现了故障，需要过一段时间才能恢复，其他实例也可以对外提供服务，不会影响业务使用。
实际上，Redis 提供了主从库模式，以保证数据副本的一致，主从库之间采用的是读写分离的方。

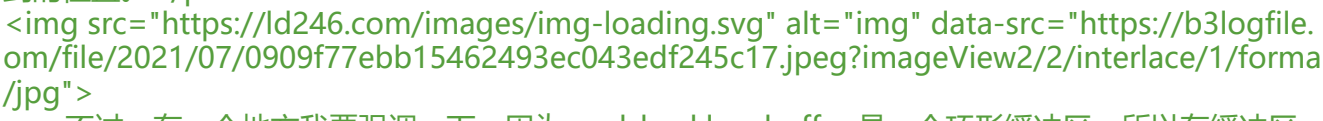
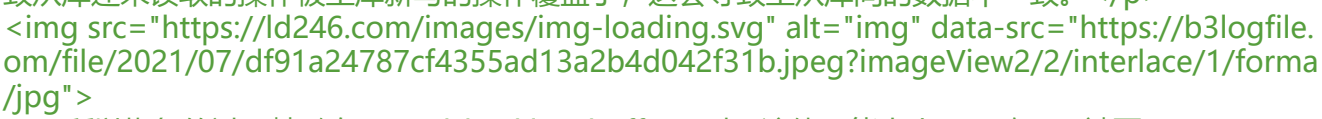
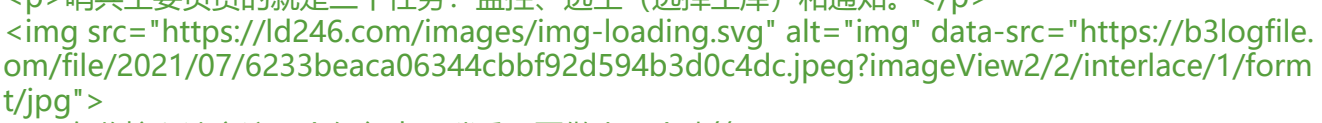
读操作：主库、从库都可以接收；
写操作：首先到主库执行，然后，主库将写操作同步给从库。

<h2 id="Q-主从之间什么时候进行第一次同步">Q：主从之间什么时候进行第一次同步</h2>
当我们启动多个 Redis 实例的时候，它们相互之间就可以通过 replicaof（Redis 5.0 之前使用 slveof）命令形成主库和从库的关系，之后会按照三个阶段完成数据的第一次同步。
例如，现在有实例 1（ip：172.16.19.3）和实例 2（ip：172.16.19.5），我们在实例 2 上执行下这个命令后，实例 2 就变成了实例 1 的从库，并从实例 1 上复制数据：
<pre><code class="language-bash highlight-chroma">replicaof 172.16.19.3 6379
</code></pre>

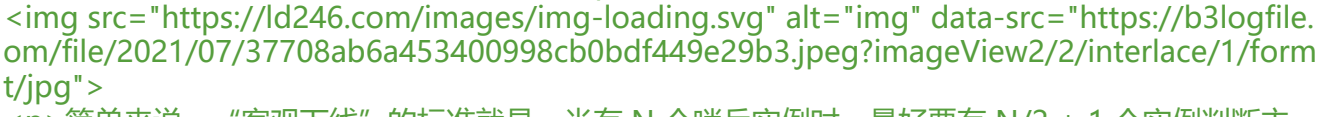
简单来说就是以下流程：
<ol start="0">
第一阶段，主从库间建立连接、协商同步的过程，主要是为全量复制做准备
第二阶段，主库将所有数据同步给从库。从库收到数据后，在本地完成数据加载。这个过程依赖内存快照生成的 RDB 文件。
第三阶段，主库会把第二阶段执行过程中新收到的写命令，再发送给从库。具体的操作是，当主完成 RDB 文件发送后，就会把此时 replication buffer 中的修改操作发给从库，从库再重新执行这操作。这样一来，主从库就实现同步了。

也就是说从库全量复制完了之后，之后都是加载 replication buffer 去实现数据同步。
<h2 id="Q-主从级联模式分担全量复制时的主库压力">Q：主从级联模式分担全量复制时的主库压力</h2>
一主多从的模式下，所有的从库都要和主库进行全量复制的话，就会导致主库忙于 fork 子进程成 RDB 文件，进行数据全量同步。fork 这个操作会阻塞主线程处理正常请求，从而导致主库响应应程序的请求速度变慢。此外，传输 RDB 文件也会占用主库的网络带宽，同样会给主库的资源使用带压力。
那么，有没有好的解决方法可以分担主库压力呢？
主 - 从 - 从模式。这个过程也称为基于长连接的命令传播，可以避免频繁建立连接的开销。
<pre><code class="language-bash highlight-chroma">replicaof 所选从库的IP 6379
</code></pre>

<h2 id="Q-主从库间网络断了怎么办-">Q：主从库间网络断了怎么办？</h2>
采用增量模式同步
当主从库断连后，主库会把断连期间收到的写操作命令，写入 replication buffer，同时也会把些操作命令也写入 repl_backlog_buffer 这个缓冲区。
repl_backlog_buffer 是一个环形缓冲区，主库会记录自己写到的位置，从库则会记录自己已经

到的位置。

不过，有一个地方我要强调一下，因为 repl_backlog_buffer 是一个环形缓冲区，所以在缓冲区满后，主库会继续写入，此时，就会覆盖掉之前写入的操作。如果从库的读取速度比较慢，就有可能致从库还未读取的操作被主库新写的操作覆盖了，这会导致主从库间的数据不一致。

所以恢复的过程精髓在于 repl_backlog_buffer，对于该值不能太小，导致圆环被覆写。
<h2 id="Q-哨兵机制的基本流程">Q：哨兵机制的基本流程</h2>
哨兵主要负责的就是三个任务：监控、选主（选择主库）和通知。

在监控和选主这两个任务中，哨兵需要做出两个决策：

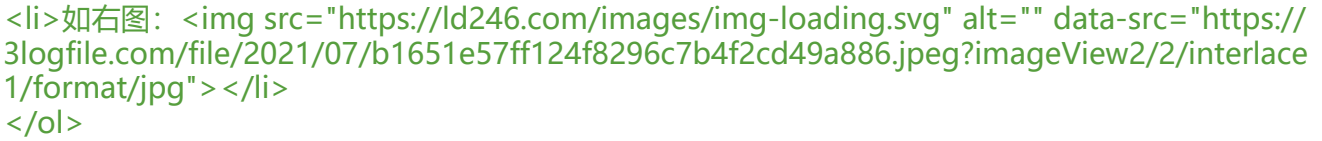
在监控任务中，哨兵需要判断主库是否处于下线状态；
在选主任务中，哨兵也要决定选择哪个从库实例作为主库。

<h3 id="主观下线与客观下线">主观下线与客观下线</h3>
哨兵进程会使用 PING 命令检测它自己和主、从库的网络连接情况，用来判断实例的状态。如果兵发现主库或从库对 PING 命令的响应超时了，那么，哨兵就会先把它标记为“主观下线”。
哨兵机制通常会采用多实例组成的集群模式进行部署，这也被称为哨兵集群。引入多个哨兵实例起来判断，就可以避免单个哨兵因为自身网络状况不好，而误判主库下线的情况。
在判断主库是否下线时，不能由一个哨兵说了算，只有大多数的哨兵实例，都判断主库已经“主下线”了，主库才会被标记为“客观下线”

简单来说，“客观下线”的标准就是，当有 N 个哨兵实例时，最好要有 N/2 + 1 个实例判断主为“主观下线”，才能最终判定主库为“客观下线”。
所以最好采用奇数个（大于等于 3）个哨兵节点。
<h3 id="如何筛选---打分-">如何筛选 + 打分？</h3>

选择优先级最高的

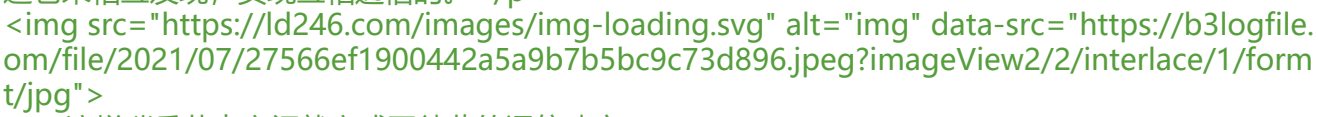
用户可以通过 slave-priority 配置项，给不同的从库设置不同优先级。比如，你有两个从库，它的内存大小不一样，你可以手动给内存大的实例设置一个高优先级。在选主时，哨兵会给优先级高的库打高分，如果有一个从库优先级最高，那么它就是新主库了。如果从库的优先级都一样，那么哨兵始第二轮打分。

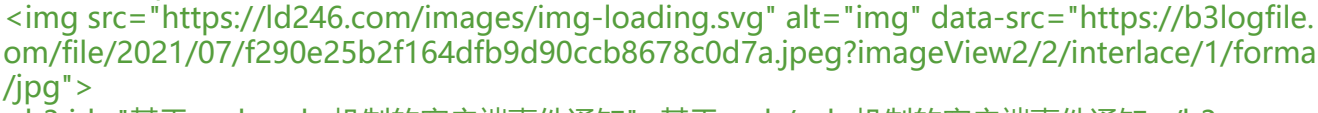
和旧主库同步程度最接近的从库得分高。

repl_backlog_buffer 中的位置，而从库会用 slave_repl_offset 这个值记录当前的复制进度。就下图所示，旧主库的 master_repl_offset 是 1000，从库 1、2 和 3 的 slave_repl_offset 分别是 95、990 和 900，那么，从库 2 就应该被选为新主库。
如右图： 

ID 号小的从库得分高。

在优先级和复制进度都相同的情况下，ID 号最小的从库得分最高，会被选为新主库。
也就是保证一定能选出一个从库来当新主库

<h2 id="Q-哨兵之间的选举">Q：哨兵之间的选举</h2>
<h3 id="基于-pub-sub-机制的哨兵集群组成">基于 pub/sub 机制的哨兵集群组成</h3>
哨兵实例之间可以相互发现，要归功于 Redis 提供的 pub/sub 机制，也就是发布 / 订阅机制。
只有订阅了同一个频道的应用，才能通过发布的消息进行信息交换。
在主从集群中，主库上有一个名为 <code>__sentinel__:hello</code> 的频道，不同哨兵就是过它来相互发现，实现互相通信的。


这样哨兵节点之间就完成了彼此的通信建立。
<h3 id="基于-pub-sub-机制的哨兵集群与主从库连接">基于 pub/sub 机制的哨兵集群与主从库连
</h3>
这是由哨兵向主库发送 INFO 命令来完成的。就像下图所示，哨兵 2 给主库发送 INFO 命令，库接受到这个命令后，就会把从库列表返回给哨兵。接着，哨兵就可以根据从库列表中的连接信息，每个从库建立连接，并在这个连接上持续地对从库进行监控。哨兵 1 和 3 可以通过相同的方法和从库立连接。


<h3 id="基于-pub-sub-机制的客户端事件通知">基于 pub/sub 机制的客户端事件通知</h3>
哨兵就是一个运行在特定模式下的 Redis 实例，只不过它并不服务请求操作，只是完成监控、选和通知的任务。所以，每个哨兵实例也提供 pub/sub 机制，客户端可以从哨兵订阅消息。哨兵提供消息订阅频道有很多，不同频道包含了主从库切换过程中的不同关键事件。
有了 pub/sub 机制，哨兵和哨兵之间、哨兵和从库之间、哨兵和客户端之间就都能建立起连接。
<h3 id="由哪个哨兵执行主从切换-">由哪个哨兵执行主从切换？</h3>
切换新主库需要选出一个 leader 来进行操作。
具体步骤（个人总结）：

先判断主观下线，发送主观下线的命令，等待其他哨兵节点回应。
收到其他哨兵节点的回应，当主观下线的票数大于 N / 2 + 1 时，标记主库是客观下线的事实，时，这个哨兵就可以再给其他哨兵发送命令，表明希望由自己来执行主从切换，并让所有其他哨兵进投票。
发送这个命令时会先投自己一票，并且只能投一次赞成票，后面接收到这个选 leader 的命令都否决票
在投票过程中，任何一个想成为 Leader 的哨兵，要满足两个条件：第一，拿到半数以上的赞成；第二，拿到的票数同时还需要大于等于哨兵配置文件中的 quorum 值。

投票选 leader 中的细节：
其他哨兵收到投票请求后，由于自己还没有询问进入判定“客观下线”的流程，所以该哨兵是可直接投票给“先做事”的哨兵，不会投 leader 票给自己。
存在的问题（我在群里的提问）：
<blockquote>
我提出的问题：假如哨兵集群主库挂了，所有哨兵实例在同一时刻判断主观下线，然后同时接收其他哨兵的消息，都到了客观下线的这一步，然后同时给自己投上一票 leader 票，是不是就没法选出eader 了。。这种小概率会发生吗？

</blockquote>
收到群友们的热心解答：

Q：假设有一个 Redis 集群，是“一主四从”，同时配置了包含 5 个哨兵实例的集群，quorum 设为 2。在运行过程中，如果有 3 个哨兵实例都发生故障了，此时，Redis 主库如果有故障，还能正地判断主库“客观下线”吗？如果可以的话，还能进行主从库自动切换吗？
A：这种情况，无论投票是怎么样，都没选出 leader 进行主从切换，因为哨兵实例数太少了，不足大于等于 3。

最后，一个经验：要保证所有哨兵实例的配置是一致的，尤其是主观下线的判断值 down-after-illiseconds。在项目中，因为这个值在不同的哨兵实例上配置不一致，导致哨兵集群一直没有对有故的主库形成共识，也就没有及时切换主库，最终的结果就是集群服务不稳定。
<h2 id="Q-切片集群-Redis-官网集群模式-">Q：切片集群（Redis 官网集群模式）</h2>
<h3 id="Redis-如何保存更多数据-">Redis 如何保存更多数据？</h3>

纵向扩展：升级单个 Redis 实例的资源配置，包括增加内存容量、增加磁盘容量、使用更高配置 CPU。就像下图中，原来的实例内存是 8GB，硬盘是 50GB，纵向扩展后，内存增加到 24GB，磁盘加到 150GB。
横向扩展：横向增加当前 Redis 实例的个数，就像下图中，原来使用 1 个 8GB 内存、50GB 磁的实例，现在使用三个相同配置的实例。

<h3 id="数据切片和实例的对应分布关系">数据切片和实例的对应分布关系</h3>
具体来说，Redis Cluster 方案采用哈希槽（Hash Slot，接下来我会直接称之为 Slot），来处理据和实例之间的映射关系。在 Redis Cluster 方案中，一个切片集群共有 16384 个哈希槽，这些哈希类似于数据分区，每个键值对都会根据它的 key，被映射到一个哈希槽中。
具体的映射过程分为两大步：首先根据键值对的 key，按照 CRC16 算法计算一个 16 bit 的值；后，再用这个 16bit 值对 16384 取模，得到 0~16383 范围内的模数，每个模数代表一个相应编号哈希槽。
手动指定每个实例上的哈希槽数量：使用 cluster meet 命令手动建立实例间的连接，形成集群再使用 cluster addslots 命令，指定每个实例上的哈希槽个数。
在手动分配哈希槽时，需要把 16384 个槽都分配完，否则 Redis 集群无法正常工作。/strong>
<h3 id="客户端如何定位数据-">客户端如何定位数据？</h3>
Redis 实例会把自己的哈希槽信息发给和它相连接的其它实例，来完成哈希槽分配信息的扩散。实例之间相互连接后，每个实例就有所有哈希槽的映射关系了。
客户端收到哈希槽信息后，会把哈希槽信息缓存在本地。当客户端请求键值对时，会先计算键所应的哈希槽，然后就可以给相应的实例发送请求了。
在集群中，实例和哈希槽的对应关系并不是一成不变的，最常见的变化有两个：

在集群中，实例有新增或删除，Redis 需要重新分配哈希槽；
为了负载均衡，Redis 需要把哈希槽在所有实例上重新分布一遍。

<blockquote>
说白了就是，客户端先存一个默认值（默认该值在某个实例上），但是哈希槽会移动，所以请求

</blockquote>

结果不一定会直接得到数据，需要对返回的结果进行二次请求。
</blockquote>
Redis Cluster 方案提供了一种重定向机制，所谓的“重定向”，就是指，客户端给一个实例发数据读写操作时，这个实例上并没有相应的数据，客户端要再给一个新实例发送操作命令。
<h4 id="MOVED命令">MOVED 命令</h4>
实例给客户端返回下面的 MOVED 命令响应结果，这个结果中就包含了新实例的访问地址：</p

<pre><code class="language-bash highlight-chroma">GET hello:key
(error) MOVED 13320 172.16.19.5:6379
</code></pre>
其中，MOVED 命令表示，客户端请求的键值对所在的哈希槽 13320，实际是在 172.16.19.5 个实例上。通过返回的 MOVED 命令，就相当于把哈希槽所在的新实例的信息告诉给客户端了。这一来，客户端就可以直接和 172.16.19.5 连接，并发送操作请求了。
会改变本地缓存，下次会直接请求正确的实例，如果哈希槽没变的话。</p

<h4 id="ASK-命令">ASK 命令</h4>
<pre><code class="language-bash highlight-chroma">GET hello:key
(error) ASK <span class="highlight-m"13320 172.16.19.5:6379
</code></pre>
这个结果中的 ASK 命令就表示，客户端请求的键值对所在的哈希槽 13320，在 172.16.19.5 这实例上，但是这个哈希槽正在迁移。此时，客户端需要先给 172.16.19.5 这个实例发送一个 ASKING 令。这个命令的意思是，让这个实例允许执行客户端接下来发送的命令。然后，客户端再向这个实例送 GET 命令，以读取数据。
不同点：
客户端 ASK 重定向命令和 MOVED 命令不同，ASK 命令并不会更新客户端缓存的哈希槽分配信。所以，在上图中，如果客户端再次请求 Slot 2 中的数据，它还是会给实例 2 发送请求。这也就是，ASK 命令的作用只是让客户端能给新实例发送一次请求，而不像 MOVED 命令那样，会改本地缓存，让后续所有命令都发往新实例。
<h3 id="Redis-Cluster不采用把key直接映射到实例的方式-而采用哈希槽的方式原因">Redis Cluste 不采用把 key 直接映射到实例的方式，而采用哈希槽的方式原因</h3>
<blockquote>
摘自评论区 @Kaito
</blockquote>
<ol start="0">
整个集群存储 key 的数量是无法预估的，key 的数量非常多时，直接记录每个 key 对应的实例射关系，这个映射表会非常庞大，这个映射表无论是存储在服务端还是客户端都占用了非常大的内存间。
Redis Cluster 采用无中心化的模式（无 proxy，客户端与服务端直连），客户端在某个节点访一个 key，如果这个 key 不在这个节点上，这个节点需要有纠正客户端路由到正确节点的能力（MOVD 响应），这就需要节点之间互相交换路由表，每个节点拥有整个集群完整的路由关系。如果存储的是 key 与实例的对应关系，节点之间交换信息也会变得非常庞大，消耗过多的网络资源，而且就算交完成，相当于每个节点都需要额外存储其他节点的路由表，内存占用过大造成资源浪费。
当集群在扩容、缩容、数据均衡时，节点之间会发生数据迁移，迁移时需要修改每个 key 的映射系，维护成本高。
而在中间增加一层哈希槽，可以把数据和节点解耦，key 通过 Hash 计算，只需要关心映射到了个哈希槽，然后再通过哈希槽和节点的映射表找到节点，相当于消耗了很少的 CPU 资源，不但让数分布更均匀，还可以让这个映射表变得很小，利于客户端和服务端保存，节点之间交换信息时也变得量。

当集群在扩容、缩容、数据均衡时，节点之间的操作例如数据迁移，都以哈希槽为基本单位进行作，简化了节点扩容、缩容的难度，便于集群的维护和管理。
[poll1625797833561]
