在项目中缓存是如何使用的？缓存如果使用不当会造成什么后果？

Redis 和 Memcached 有什么区别？Redis 的线程模型是什么？为什么单线程的 Redis 比多线程的 Memcached 效率要高得多？*

数据类型

redis 主要有以下几种数据类型：

	String 字符串	**Hash**	**list**	**set**	**sort set**
存储方式	K,V	结构存储	有序	无序和去重】、	排序列表
应用	万物皆可字符串	存对象（但不可有对象的引用）	粉丝、关注、评论	点赞人，全局过滤去重	排行榜
技巧		可对象的操作字段	lrange分页，bpop阻塞队列	交集并集差集、	

为啥 redis 效率这么高？

- 纯内存操作。
- 核心是基于非阻塞的 IO 多路复用机制。
- C 语言实现，一般来说，C 语言实现的程序“距离”操作系统更近，执行速度相对会更快。
- 单线程反而避免了多线程的频繁上下文切换问题，预防了多线程可能产生的竞争问题。

过期策略

- 往 redis 写入的数据怎么没了？

内存是有限的，比如 redis 就只能用 10G，你要是往里面写了 20G 的数据，会干掉 10G 的数据，然后就保留 10G 的数据了。

- 数据明明过期了，怎么还占用着内存？

这是由 redis 的过期策略来决定。

redis 过期策略

redis 过期策略是：**定期删除+惰性删除。**

所谓**定期删除**，指的是 redis 默认是每隔 100ms 就随机抽取一些设置了过期时间的 key，检查其是否过期，如果过期就删除。

惰性删除:获取 key 的时候，如果此时 key 已经过期，就删除，不会返回任何东西。

如果定期删除漏掉了很多过期 key，然后你也没及时去查，也就没走惰性删除，大量过期 key 堆积在内存里，导致 redis 内存块耗尽了，咋整？答案是：**走内存淘汰机制。**

内存淘汰机制

redis 内存淘汰机制有以下几个：

- noeviction: 当内存不足以容纳新写入数据时，新写入操作会报错，这个一般没人用吧，实在是太恶心了。
- **allkeys-lru**：当内存不足以容纳新写入数据时，在**键空间**中，移除最近最少使用的 key（这个是**最常用**的）。
- allkeys-random：当内存不足以容纳新写入数据时，在**键空间**中，随机移除某个 key，这个一般没人用吧，为啥要随机，肯定是把最近最少使用的 key 给干掉啊。
- volatile-lru：当内存不足以容纳新写入数据时，在**设置了过期时间的键空间**中，移除最近最少使用的 key（这个一般不太合适）。
- volatile-random：当内存不足以容纳新写入数据时，在**设置了过期时间的键空间**中，**随机移除**某个 key。
- volatile-ttl：当内存不足以容纳新写入数据时，在**设置了过期时间的键空间**中，有**更早过期时间**的 key 优先移除。

持久化

- RDB：RDB 持久化机制，是对 redis 中的数据执行**周期性**的持久化。
- AOF：AOF 机制对每条写入命令作为日志，以 `append-only` 的模式写入一个日志文件中，在 redis 重启的时候，可以通过**回放** AOF 日志中的写入指令来重新构建整个数据集。

	如何持久化	丢失数据	写入性能	文件大小	恢复速度	场景
RDB	数据周期存储	容易丢失	低	小	快	冷备
AOF	指令重新构建	不容易（只丢失最近一秒）	高，没有磁盘寻址的开销	大，文件是逐渐大的	慢	误删的紧急恢复

RDB 快照（保存值）

触发机制

1.全量复制：主从复制的时候，主会自动生成RDB文件
2.debug reload：不需要将内存清空的重启也会触发RDB的生成
3.shutdown，关闭的时候会自动生成shutdown save

三种命令

save（同步）数据进行完整的拷贝的话可能会阻塞注命令
bgsave(异步)生成子进程去完成RDB的生成
自动：满足900秒有1个改变、300秒有10个，60秒有1000个改变一条件就进行save

场景：冷备

AOF 日志（保存命令）

这个文件是逐渐增大的。

三种策略：always everysec no（os决定）

AOF重写：把一些过期的命令进行优化。

场景：误删的紧急处理。比如某人不小心用 `flushall` 命令清空了所有数据，只要这个时候后台 `rewrite` 还没有发生，那么就可以立即拷贝 AOF 文件，将最后一条 `flushall` 命令给删了，然后再将该 `AOF` 文件放回去，就可以通过恢复机制，自动恢复所有数据。

RDB 和 AOF 到底该如何选择

- 仅仅使用 RDB，因为那样会导致你丢失很多数据；
- 仅仅使用 AOF有两个问题：第一， AOF 没有 RDB 做冷备来的**恢复速度**更快；第二，RDB 每次简单粗暴生成数据快照，**更加健壮**，可以避免 AOF 这种复杂的备份和恢复机制的 bug；
- redis 支持同时开启开启两种持久化方式，我用 AOF 来保证数据不丢失，作为数据恢复的第一选择; 用 RDB 来做不同程度的冷备，在 AOF 文件都丢失或损坏不可用的时候，还可以使用 RDB 来进行快速的数据恢复。另外，如果同时使用 RDB 和 AOF 两种持久化机制，**redis** 重启的时候，会使用 **AOF** 来重新构建数据，因为 AOF 中的**数据更加完整**。

主从复制

数据流是单项的，从master到slave

全量复制与部分复制

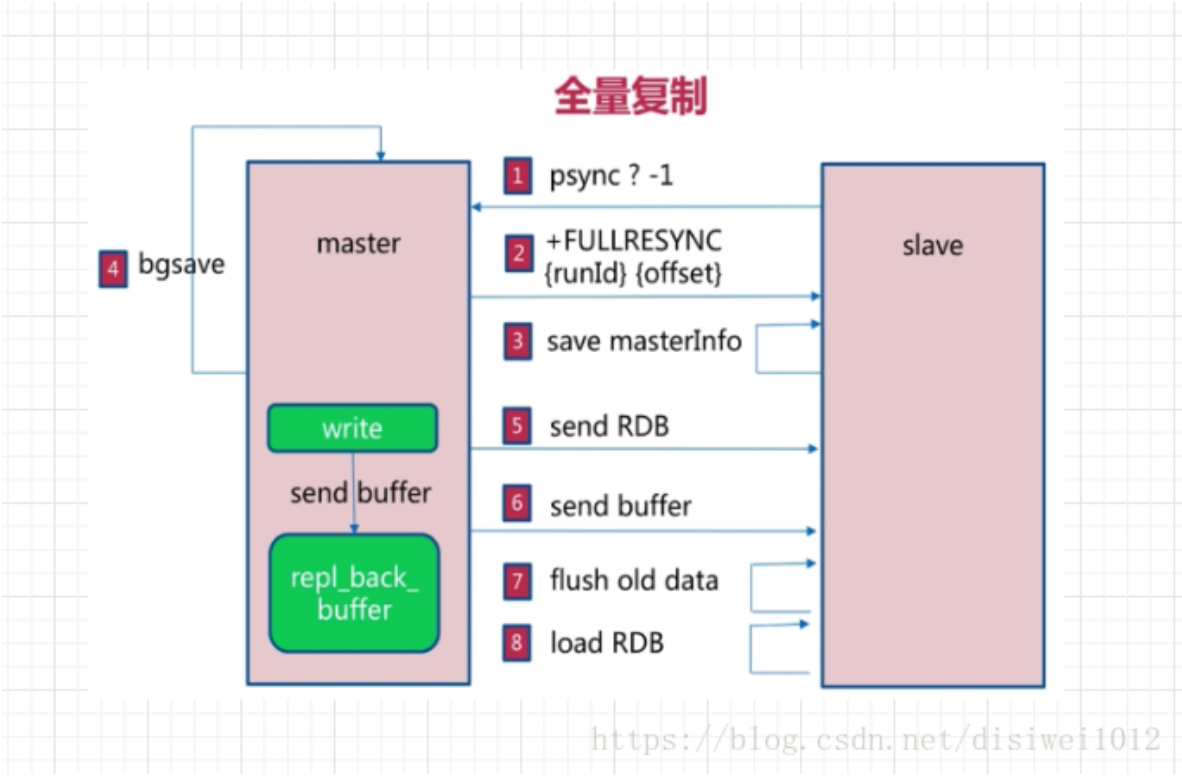
全量复制

- 从服务器向主服务器发送PSYNC命令，；第一次的话，slave不知道master 的runid，所以是？,偏移量是-1

2. 主服务器验证runid和自身runid是否一致，如不一致，则进行全量复制；
3. Master把自己的runid和offset（偏移量）发给slave，slave保存起来
4. 使用bgsave生成RDB文件。

- 由于bgsave是异步的过程，master还可以继续写数据，这一段时间的操作放入缓冲区。
- 通过网络磁盘传去，再把缓冲区数据传去。
- Salve会先清除原来的数据，加载RDB和缓冲区数据，写入本地磁盘，然后再从本地磁盘加载到内存

过程如下图：

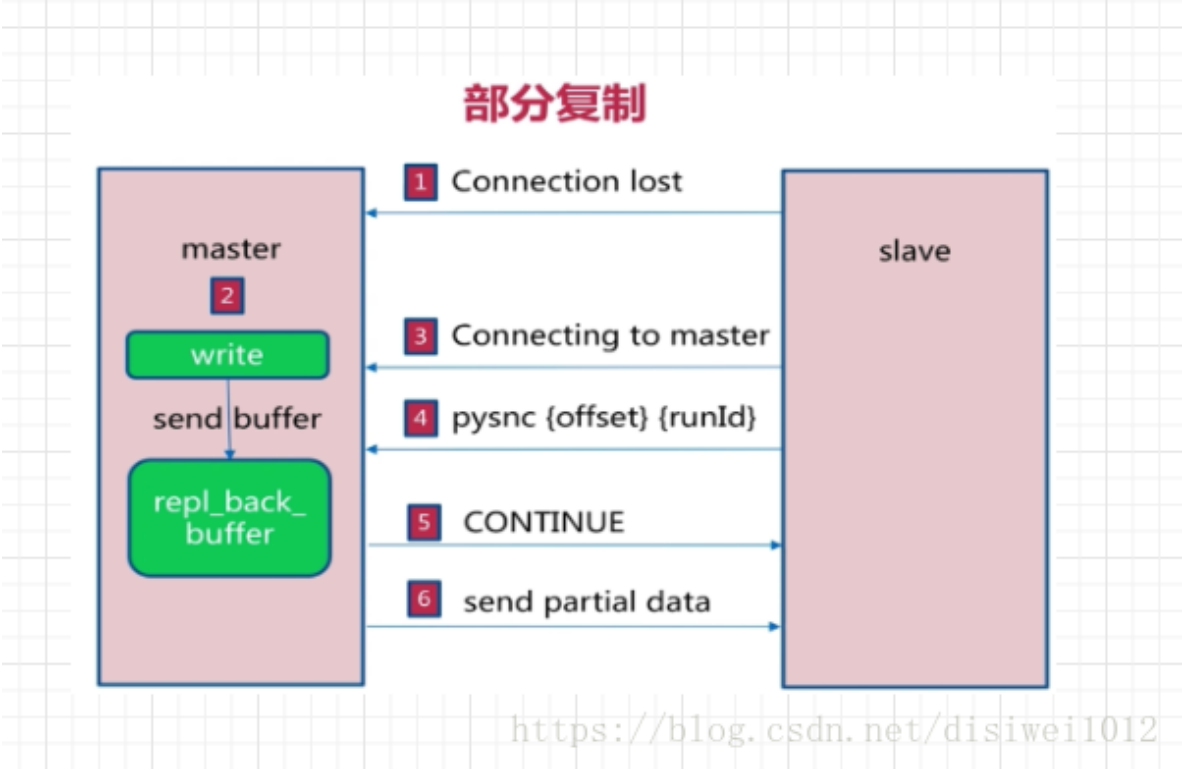


全量复制的开销包括以下几个方面：

1. bgsave时间
2. RDB文件网络传输时间
3. 从节点清空数据时间
4. 从节点加载RDB数据时间

部分复制（增量复制）

部分复制过程如下图：



部分复制的过程：

1. 当网络发生抖动，slave向master发送连接
2. 要求同步master数据，同时自己的传递偏移量和master runid
3. 如果runid一致，则查看slave的偏移量和master的偏移量是否一直。如果不一致，则观察偏移量是否超过repl_back_buffer中能存储的数据。
4. 如果超过则可能进行全量复制
5. 如果未超过则，将repl_back_buffer中存储的数据发送给slave，slave完成数据的同步

heartbeat

主从节点互相都会发送 heartbeat 信息。

master 默认每隔 10秒 发送一次 heartbeat，slave node 每隔 1秒 发送一个 heartbeat。

哨兵保证高可用

三个定时任务

（1）每隔10s每个sentinel会对master节点和slave节点执行info命令。作用就是发现slave节点，并且确认主从关系

（2）每隔两秒，sentinel都会通过master节点内部的channel来**交换信息**（基于发布订阅）每个哨兵都会往 `__sentinel__:hello` 这个 channel 里发送一个消息，这时候所有其他哨兵都可以消费到这个消息，并感知到其他的哨兵的存在。

（3）每隔一秒每个sentinel对其他的redis节点（master，slave，sentinel）执行ping操作，对于master来说，若超过30s内没有回复，就对该master进行主观下线并询问其他的Sentinel节点是否可以客观下线

主观下线和 客观下线

- sdown 是主观宕机，就一个哨兵如果自己觉得一个 master 宕机了，那么就是主观宕机
- odown 是客观宕机，如果 quorum 数量的哨兵都觉得一个 master 宕机了，那么就是客观宕机

sdown 达成的条件很简单，如果一个哨兵 ping 一个 master，超过了 `is-master-down-after-milliseconds` 指定的毫秒数之后，就主观认为 master 宕机了；如果一个哨兵在指定时间内，收到了 quorum 数量的其它哨兵也认为那个 master 是 sdown 的，那么就认为是 odown 了。

sentinel领导选举与法定人数

。原因:只有一个sentinel节点完成故障转移
。选举:通过sentinel is-master-down-by-addr命令都希望成为领导者
　1,每个做主观下线的Sentinel节点向其他Sentinel节点发送命令,要求将它设置为领导者.
　2,收到命令的Sentinel节点如果没有同意通过其他Sentinel节点发送命令,那么将同意该请求，否则拒绝
　3,如果该Sentinel节点发现自己的票数已经超过Sentinel集合半数且超过quorum,那么它将成为领导者
　4,如果此过程有多个Sentinel节点成为了领导者,那么将等待一段时间重新进行选举

备注：quorum可以在sentinel的conf里配置,后面的2就是法定人数

```
sentinel monitor mymaster 127.0.0.1 6379 2
```

法定人数需要比一般的哨兵数还大，如果小于，法定人数为一半以上的哨兵数。

如果 quorum < majority，比如 5 个哨兵，majority 就是 3，quorum 设置为 2，那么就 3 个哨兵授权就可以执行切换。

但是如果 quorum >= majority，那么必须 quorum 数量的哨兵都授权，比如 5 个哨兵，quorum 是 5，那么必须 5 个哨兵都同意授权，才能执行切换。

master选举算法

如果一个 master 被认为 odown 了，而且 majority 数量的哨兵都允许主备切换

首先如果一个 slave 跟 master 断开连接的时间已经超过了 `down-after-milliseconds` 的 10 倍，外加 master 宕机的时长，那么 slave 就被认为不适合选举为 master。

```
(down-after-milliseconds * 10) +
milliseconds_since_master_is_in_SDOWN_state
```

接下来会对 slave 进行排序：

- 按照 slave 优先级进行排序，slave priority 越低，**优先级**就越高。
- 如果 slave priority 相同，那么看 replica **offset**，哪个 slave 复制了越多的数据，offset 越靠后，优先级就越高。
- 如果上面两个条件都相同，那么选择一个 **run id** 比较小的那个 slave。

slave 配置的自动纠正

哨兵会负责自动纠正 slave 的一些配置，比如 slave 如果要成为潜在的 master 候选人，哨兵会确保 slave 复制现有 master 的数据；如果 slave 连接到了一个错误的 master 上，比如故障转移之后，那么哨兵会确保它们连接到正确的 master 上。

slave上升为master日志中可以发现，这一条配置重写的日志

```
989:M 21 Aug 19:20:42.729 # CONFIG REWRITE executed with success.
```

数据丢失

脑裂

脑裂，某个 master 掉线了一会，但是实际上还运行着。此时哨兵可能就会认为 master 宕机了，然后开启选举，集群里就会有两个 master ，也就是所谓的脑裂。client 还没来得及切换到新的 master，还继续向旧 master 写数据。因此旧 master 再次恢复的时候，会被作为一个 slave 挂到新的 master 上去，自己的数据会清空，重新从新的 master 复制数据。而新的 master 并没有后来 client 写入的数据，因此，这部分数据也就丢失了。

解决方案

进行如下配置：

```
min-slaves-to-write 1
min-slaves-max-lag 10
```

表示，要求至少有 1 个 slave，数据复制和同步的延迟不能超过 10 秒。

如果说一旦所有的 slave，数据复制和同步的延迟都超过了 10 秒钟，那么这个时候，master 就不会再接收任何请求了。在脑裂场景下，最多就丢失 10 秒的数据。

异步复制

因为 master->slave 的复制是异步的，所以可能有部分数据还没复制到 slave，master 就宕机了，此时这部分数据就丢失了。

Redis 集群

数据分片：按某种规则对海量数据划分，分散存储多个结点上。

常见的数据分布方式有两种

1. 哈希分布：分散高，与业务无关，无法顺序访问
2. 顺序分布：分散低易倾斜，与业务有关，可顺序访问

redis哈希分布，但不是简单的取模，因为这样动态的添加/删除结点会大费周章

一致性哈希分区

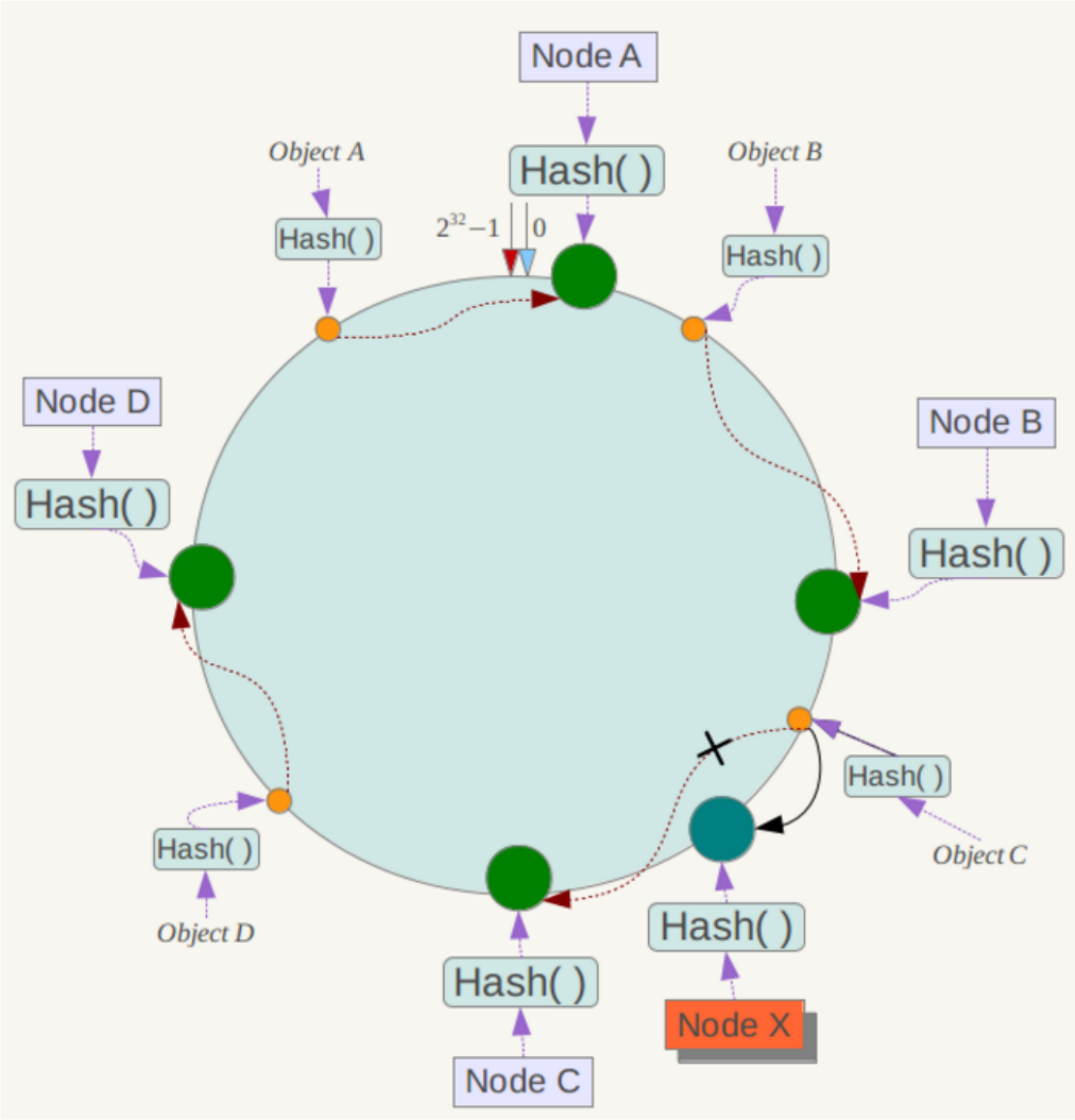
一致性 hash 算法将整个 hash 值空间组织成一个虚拟的圆环，整个空间按顺时针方向组织，下一步将各个 master 节点（使用服务器的 ip 或主机名）进行 hash。这样就能确定每个节点在其哈希环上的位置。

来了一个 key，首先计算 hash 值，并确定此数据在环上的位置，从此位置沿环**顺时针“行走”**，遇到的第一个 master 节点就是 key 所在位置。

在一致性哈希算法中，如果一个节点挂了，受影响的数据仅仅是此节点到环空间前一个节点（沿着逆时针方向行走遇到的第一个节点）之间的数据，其它不受影响。增加一个节点也同理。

一致性Hash算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

燃鹅，一致性哈希算法在**节点太少**时，容易因为节点分布不均匀而造成**缓存热点/数据倾斜**的问题。为了解决这种热点问题，一致性 hash 算法引入了虚拟节点机制，即对每一个节点计算多个 hash，每个计算结果位置都放置一个**虚拟节点**。这样就实现了数据的均匀分布，负载均衡。

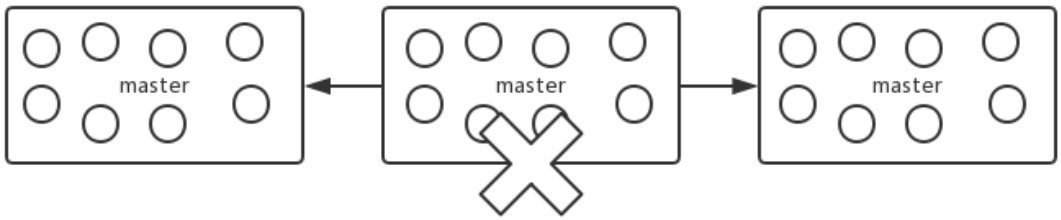


hash slot 算法

redis cluster 有固定的 `16384` 个 hash slot，对每个 `key` 计算 `CRC16` 值，然后对 `16384` 取模，可以获取 key 对应的 hash slot。

redis cluster 中每个 master 都会持有部分 slot，比如有 3 个 master，那么可能每个 master 持有 5000 多个 hash slot。hash slot 让 node 的增加和移除很简单，增加/减少一个 master，就将它的 hash slot 移动。移动 hash slot 的成本是非常低的。客户端的 api，可以对指定的数据，让他们走同一个 hash slot，通过 `hash tag` 来实现。

与取模分片的区别在于，hashsolt把%的数据量变大了，任何一台机器宕机，另外两个节点，不影响的。因为 key 找的是 hash slot，不是机器。



1. 虚拟槽分区

高可用

redis cluster 的高可用的原理，几乎跟哨兵是类似的。

判断节点宕机，主观+客观下线

cluster-node-timeout`内，某个节点一直没有返回`pong`，那么就被认为`pfail`。

`pfail`，**主观宕机**。类似odown。

fail，**客观宕机**。类似sdown。超过半数的节点都认为 `pfail` 了，那么就会变成 `fail`。

新master 选举

对宕机的 master node，从其所有的 slave node 中，选择一个切换成 master node。

- 断开连接的时间超过了 `cluster-node-timeout * cluster-slave-validity-factor`，**没有资格**切换成 `master`。
- offset越大，优先进行选举。
- 选举投票，拥有N/2 + 1票的结点选举通过成为master。
- 从节点执行主备切换，从节点切换为主节点。

与主从复制和哨兵

二者都可以做到高并发高性，用哪个主要看数据量

如果你的数据量很少就几个 G，单机就足够了，可以自己搭建一个 sentinel 集群去保证 redis 主从架构的高可用性。

redis cluster，主要是针对**海量数据+高并发+高可用**的场景。redis cluster 还可以支持横向扩容更多的 master 节点。

节点间的内部通信机制

基本通信原理

集群元数据的维护有两种方式：集中式、Gossip 协议。redis cluster 节点间采用 gossip 协议进行通信。

- 集中式是将集群元数据（节点信息、故障等等）集中存储在某个节点上。
- gossip` 协议，所有节点都持有一份元数据，元数据的变更会发送给其它的节点通知变更。

集中式的好处在于，元数据时效常好。不好在于，集中存储元数据有更新访问压力。

gossip 正好相反。好处在于，元数据分散，降低了压力；不好在于，元数据更新滞后。

gossip 协议

- 10000 端口：每个节点都有一个专门用于节点间通信的端口，就是自己提供服务的端口号+10000，比如 7001，那么用于节点间通信的就是 17001 端口。每个节点每隔一段时间都会往另外几个节点发送 `ping` 消息，同时其它几个节点接收到 `ping` 之后返回 `pong`。
- 交换的信息：信息包括故障信息，节点的增加和删除，hash slot 信息等等。

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

- meet消息：**用于通知新节点加入**。消息发送者通知接收者加入到当前集群，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
- 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消息发送封装了自身节点和部分其他节点的状态数据
- 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：**用于主观下线到客观下线。**判断为主观节点后就发送 fail 给其它节点。

雪崩、穿透和击穿

雪崩：缓存管理，请求全打在数据库上，把数据库也打挂了。

- 事前：redis 高可用，主从+哨兵，redis cluster，避免全盘崩溃。
- 事中：本地 ehcache 缓存 + hystrix 限流&降级，避免 MySQL 被打死。
- 事后：redis 持久化，一旦重启，自动从磁盘上加载数据，快速恢复缓存数据。

用户发送一个请求，系统 A 收到请求后，先查本地 ehcache 缓存，如果没查到再查 redis。如果 ehcache 和 redis 都没有，再查数据库，将数据库中的结果，写入 ehcache 和 redis 中。

限流组件，可以设置每秒的请求，有多少能通过组件，剩余的未通过的请求，怎么办？**走降级**！可以返回一些默认的值，或者友情提示，或者空白的值。

好处：

- 数据库绝对不会死，限流组件确保了每秒只有多少个请求能通过。
- 只要数据库不死，就是说，对用户来说，2/5 的请求都是可以被处理的。
- 只要有 2/5 的请求可以被处理，就意味着你的系统没死，对用户来说，可能就是点击几次刷不出来页面，但是多点几次，就可以刷出来一次。

缓存穿透

缓存穿透，即黑客发几千个不存在的恶意攻击请求。缓存中查不到，会直接去数据库里查（当然也查不到）。但**这种恶意攻击场景的缓存穿透就会直接把数据库给打死。**

解决：每次系统 A 从数据库中只要没查到，就写一个空值到缓存里去，比如 `set -999 UNKNOWN`。然后设置一个过期时间，这样的话，下次有相同的 key 来访问的时候，在缓存失效之前，都可以直接从缓存中取数据。

但是如果**key是随机生成**的，这样的做法就用处不大了。可以**布隆过滤器**解决，请求过来，先调用布隆过滤器判断数据是否存在。如果不存在的数据，就不要把请求引向数据库。直接过滤掉了大量不存在的数据攻击。redis就带有bitmap哦，我猜就是做这个功能的。

缓存击穿

缓存击穿，就是说某个 key 非常热点，访问非常频繁，处于集中式高并发访问的情况，**当这个 key 在失效的瞬间**，大量的请求就击穿了缓存，直接请求数据库，就像是在一道屏障上凿开了一个洞。

解决方式也很简单，可以将热点数据设置为永远不过期；或者基于 redis or zookeeper 实现互斥锁，等待第一个请求构建完缓存之后，再释放锁，进而其它请求才能通过该 key 访问数据。

缓存与数据库的双写一致性

问题1：先更新数据库，再删除缓存。如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据就出现了不一致。

解决思路：先删除缓存，再更新数据库。如果数据库更新失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致。因为读的时候缓存没有，所以去读了数据库中的旧数据，然后更新到缓存中。（每天上亿的读请求，每秒并发读几万）

问题2：数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改。一个请求过来，去读缓存，发现缓存空了，去查询数据库，查到了修改前的旧数据，放到了缓存中。随后数据变更的程序完成了数据库的修改。造成不一致
解决思路(1)：写请求先删除缓存，再去更新数据库，（异步等待段时间)再删除缓存（成功表示有脏数据出现）。这种方案读取快速，但会出现短时间的脏数据。

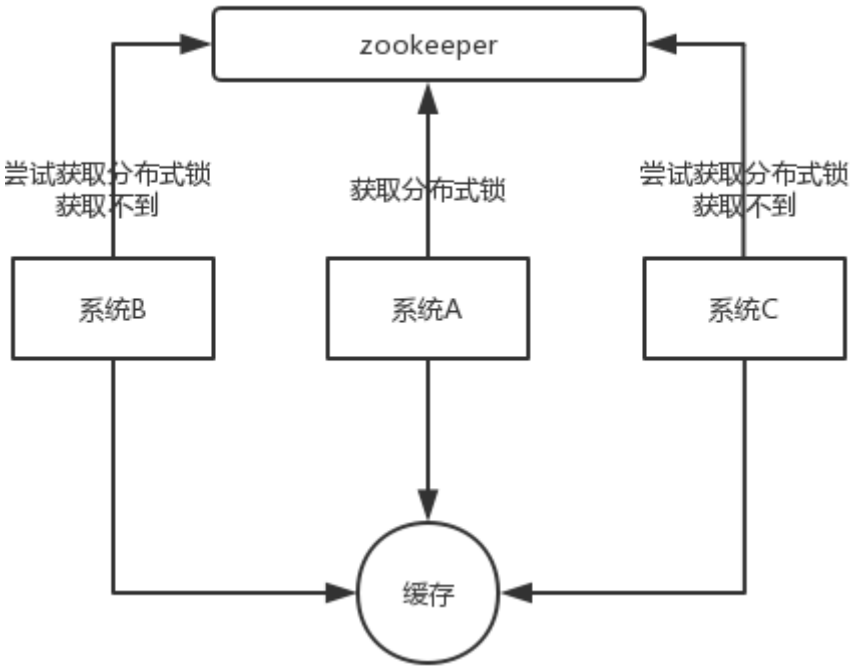
解决思路(2)：写请求先修改缓存为指定值，再去更新数据库，再更新缓存。读请求过来后，先读缓存，判断是指定值后进入循环状态，等待写请求更新缓存。如果循环超时就去数据库读取数据，更新缓存。这种方案保证了读写的一致性，但是读请求会等待写操作的完成，降低了吞吐量

Redis 并发竞争解决方案

问题：多客户端同时并发写一个 key，会造成更新丢失。

分布式锁（zookeeper）+时间戳

分布式锁保证同一时间只能有一个客户端实实例在操作数据。mysql 数据要有一个时间戳。**写之前，先判断**一下当前这个 value 的时间戳是否比缓存里的 value 的时间戳要新。不能用旧的数据覆盖新的数据。



生产环境中的 Redis 是怎么部署的？

其他

慢查询

1. FIFO
2. 固定长度，超过会丢弃
3. 保存在内存里

生命周期，慢查询不是发生在网络请求和排队上，可以看看图

两个配置

- showlog-log-slower-than
- showlog-max-len

三个命令

- 获取慢查询队列
- 获取慢查询队列长度
- 清空慢查询

运维经验：慢查询定期持久化

pipeline

流水线功能
什么是流水线 ，redis命令是超快，但网络时间慢。

把n次命令+n次网络请求变成次命令+1次网络

只能用客户端实现

使用建议
1.注意每次pipeline携带数据量
2.pipeline每次只能作用在一个Redis节点上
3.M操作和pipeline的区别

发布订阅

发布者(publisher)、订阅者(subscriber)、频道(channel) 发布者发布消息到频道上，订阅者订阅频道就会收到消息

API

publish
unsubscribe
subscribe

与消息队列的区别：不是专业的，新上线的订阅者不会收到一起的消息

与生产消费模式：生产消费模式中，发布者发布一条，只有一个订阅者能够收到。而redis相当于一个广播