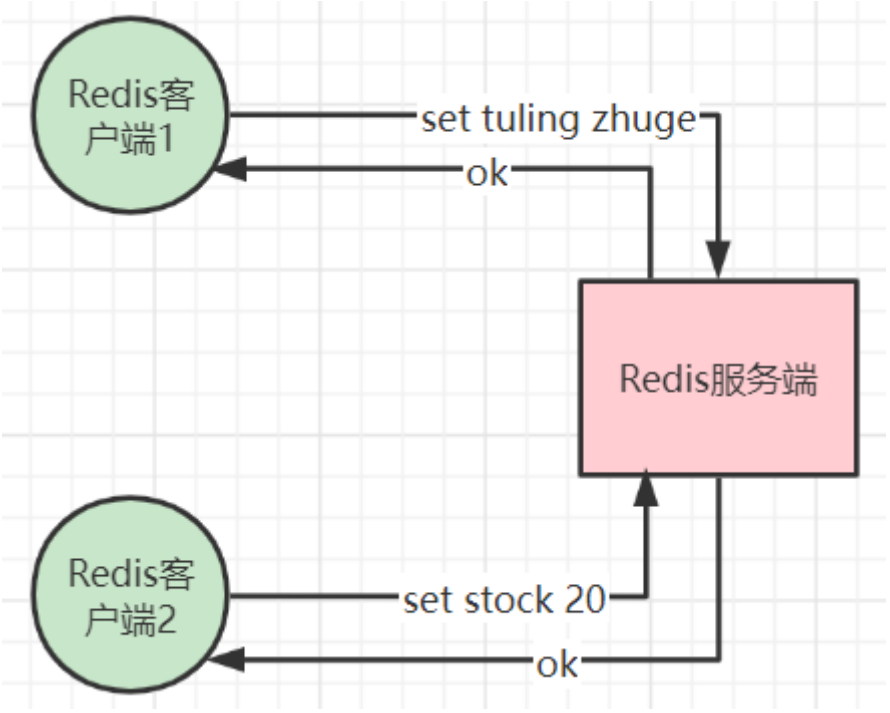
Redis到底是单线程还是多线程

Redis 6.0版本之前的单线程指的是其网络I/O和键值对读写是由一个线程完成的。

Redis6.0引入的多线程指的是网络请求过程采用了多线程，而键值对读写命令仍然是单线程处理的，所以Redis依然是并发安全的。

也就是只有网络请求模块和数据操作模块是单线程的，而其它的持久化、集群数据同步等，其实是由额外的线程执行的。



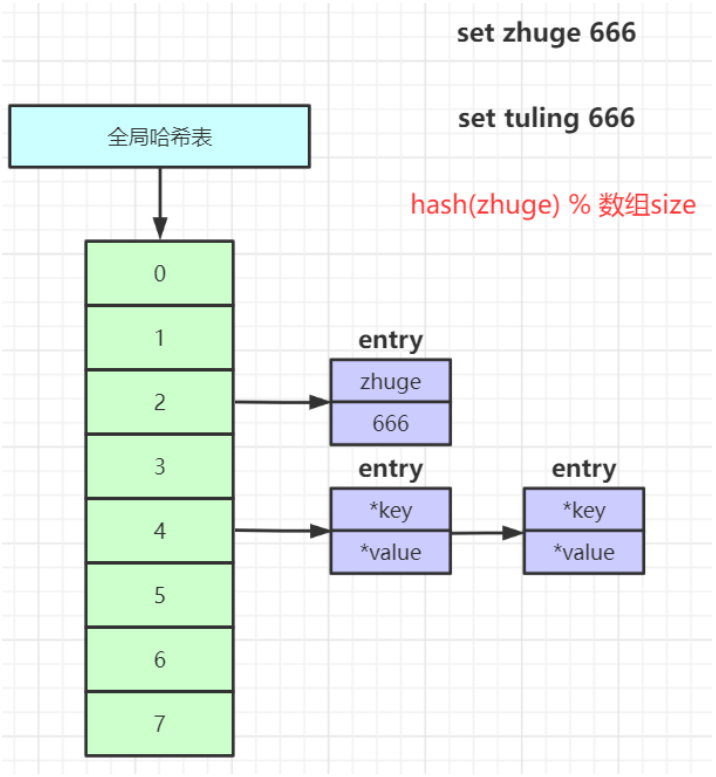
Redis单线程为什么还能这么快

1、命令执行基于内存操作，一条命令在内存里操作的时间是几十纳秒。

2、命令执行是单线程操作，没有线程切换开销。

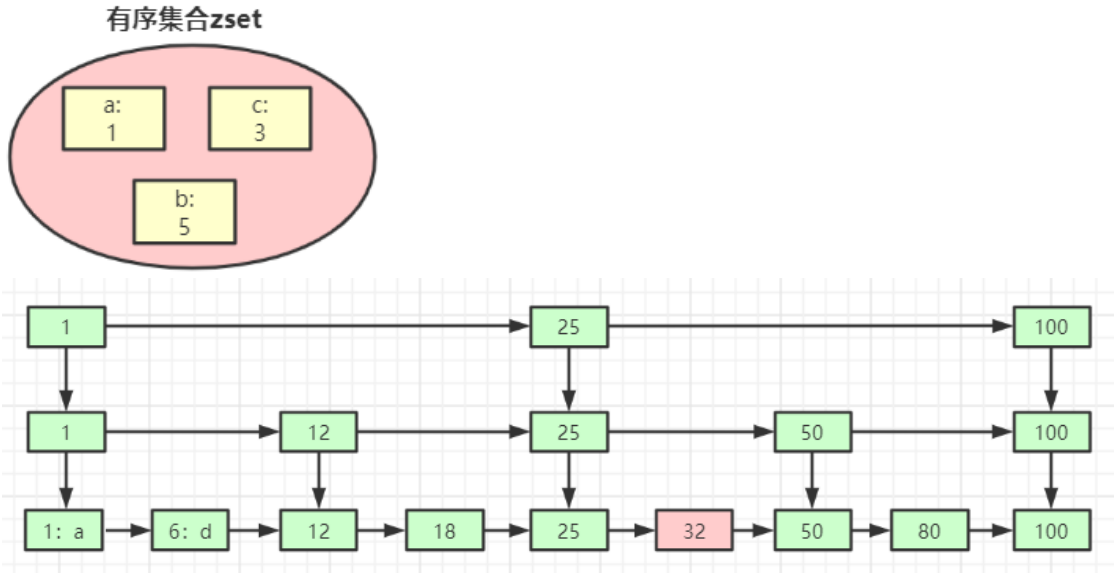
3、基于IO多路复用机制提升Redis的I/O利用率。

4、高效的数据存储结构：全局hash表以及多种高效数据结构，比如：跳表，压缩列表，链表等等。



Redis底层数据是如何用跳表来存储的

跳表：将有序链表改造为支持近似“折半查找”算法，可以进行快速的插入、删除、查找操作。



Redis Key过期了为什么内存没释放

你在使用 Redis 时，肯定经常使用 SET 命令。

SET 除了可以设置 key-value 之外，还可以设置 key 的过期时间，就像下面这样：

```
127.0.0.1:6379> SET tuling zhuge EX 120
OK
127.0.0.1:6379> TTL tuling
(integer) 117
```

此时如果你想修改 key 的值，但只是单纯地使用 SET 命令，而没有加上过期时间的参数，那这个 key 的过期时间将会被**擦除。**

```
127.0.0.1:6379> SET tuling zhuge666
OK
127.0.0.1:6379> TTL tuling // key永远不过期了！
(integer) -1
```

导致这个问题的原因在于：SET 命令如果不设置过期时间，那么 Redis 会自动擦除这个 key 的过期时间。

如果你发现 Redis 的内存持续增长，而且很多 key 原来设置了过期时间，后来发现过期时间丢失了，很有可能是因为这个原因导致的。

这时你的 Redis 中就会存在大量不过期的 key，消耗过多的内存资源。

所以，你在使用 SET 命令时，如果刚开始就设置了过期时间，那么之后修改这个 key，也务必要加上过期时间的参数，避免过期时间丢失问题。

Redis对于过期key的处理一般有惰性删除和定时删除两种策略。

1、惰性删除：当读/写一个已经过期的key时，会触发惰性删除策略，判断key是否过期，如果过期了直接删除掉这个key。

2、定时删除：由于惰性删除策略无法保证冷数据被及时删掉，所以Redis会定期(默认每100ms)主动淘汰一批已过期的key，这里的一批只是部分过期key，所以可能会出现部分key已经过期但还没有被清理掉的情况，导致内存并没有被释放。

Redis Key没设置过期时间为什么被Redis主动删除了

当Redis已用内存超过maxmemory限定时，触发主动清理策略。

主动清理策略在Redis 4.0之前一共实现了 6 种内存淘汰策略，在 4.0 之后，又增加了 2 种策略，总共8种：

a) 针对设置了过期时间的key做处理：

1. volatile-ttl：在筛选时，会针对设置了过期时间的键值对，根据过期时间的先后进行删除，越早过期的越先被删除。
2. volatile-random：就像它的名称一样，在设置了过期时间的键值对中，进行随机删除。
3. volatile-lru：会使用 LRU 算法筛选设置了过期时间的键值对删除。
4. volatile-lfu：会使用 LFU 算法筛选设置了过期时间的键值对删除。

b) 针对所有的key做处理：

5. allkeys-random：从所有键值对中随机选择并删除数据。
6. allkeys-lru：使用 LRU 算法在所有数据中进行筛选删除。
7. allkeys-lfu：使用 LFU 算法在所有数据中进行筛选删除。

c) 不处理：

8. noeviction：不会剔除任何数据，拒绝所有写入操作并返回客户端错误信息"(error) OOM command not allowed when used memory"，此时Redis只响应读操作。

Redis淘汰Key的算法LRU与LFU区别

LRU 算法(Least Recently Used，最近最少使用)：淘汰很久没被访问过的数据，以最近一次访问时间作为参考。
LFU 算法(Least Frequently Used，最不经常使用)：淘汰最近一段时间被访问次数最少的数据，以次数作为参考。
绝大多数情况我们都可以用LRU策略，当存在大量的热点缓存数据时，LFU可能更好点。

删除Key的命令会阻塞Redis吗

有可能的，我们看下DEL Key命令的时间复杂度：

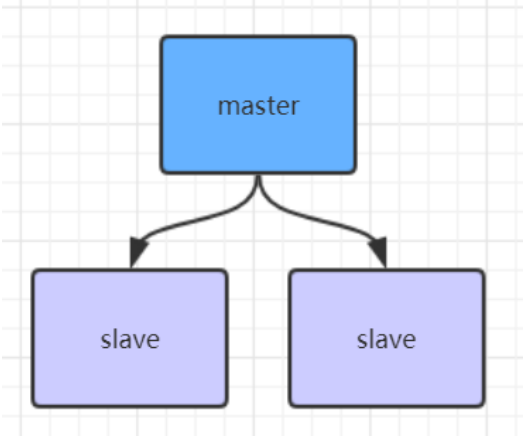
- 删除单个字符串类型的 key ，时间复杂度为 O(1)。
- 删除单个列表、集合、有序集合或哈希表类型的 key ，时间复杂度为 O(M)，M 为以上数据结构内的元素数量。

如果删除的是列表、集合、有序集合或哈希表类型的 key，如果集合元素过多，是会阻塞Redis的。对于这种情况我们可以借助scan这样的命令循环删除元素。

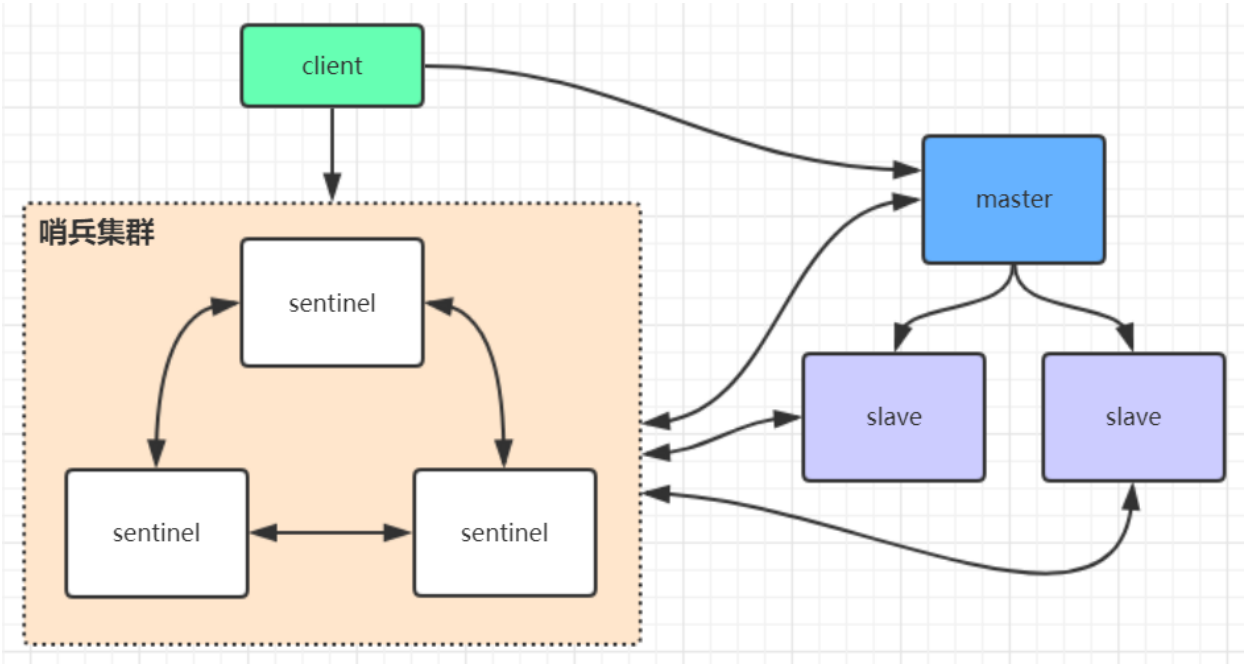
如果删除的是字符串类型的 key，但是key对应value比较大，比如有几百M，那么也是会阻塞Redis的。这种bigkey是我们要尽量减少出现的情况。

Redis主从、哨兵、集群架构优缺点比较

- **主从模式**

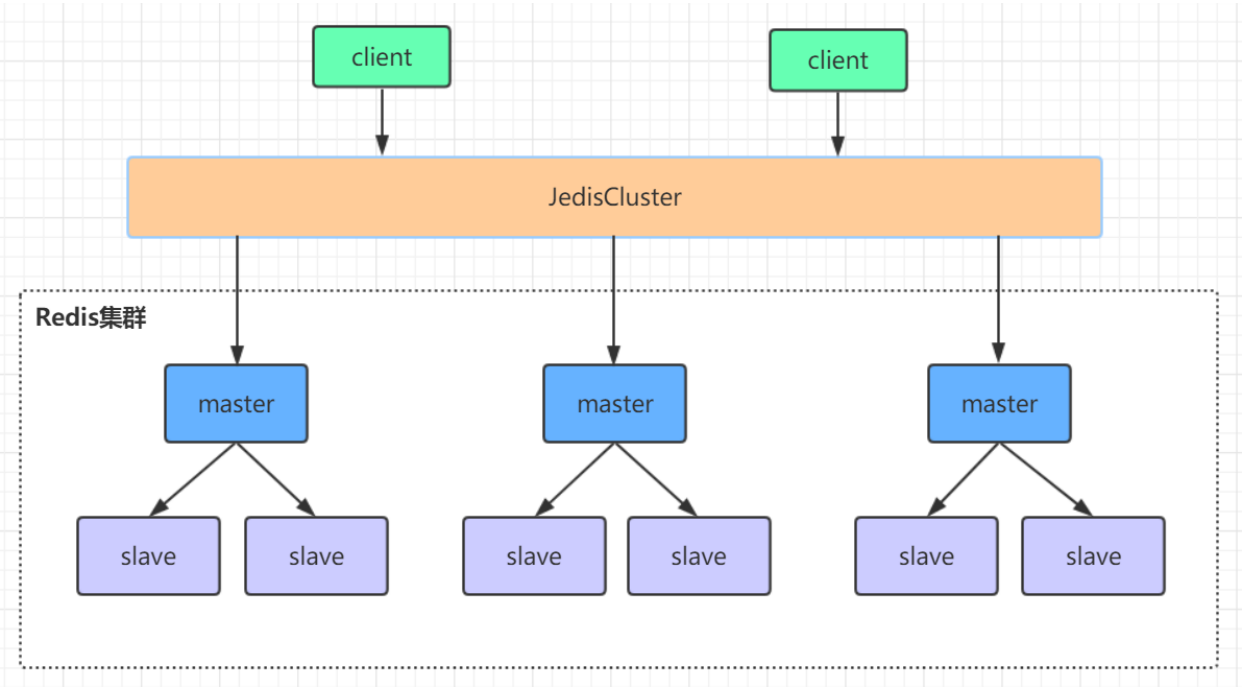


- **哨兵模式**



在redis3.0以前的版本要实现集群一般是借助哨兵sentinel工具来监控master节点的状态，如果master节点异常，则会做主从切换，将某一台slave作为master，哨兵的配置略微复杂，并且性能和高可用性等各方面表现一般，特别是在主从切换的瞬间存在访问瞬断的情况，而且哨兵模式只有一个主节点对外提供服务，没法支持很高的并发，且单个主节点内存也不宜设置得过大，否则会导致持久化文件过大，影响数据恢复或主从同步的效率

- **高可用集群模式**



redis集群是一个由多个主从节点群组成的分布式服务器群，它具有复制、高可用和分片特性。Redis集群不需要sentinel哨兵·也能完成节点移除和故障转移的功能。需要将每个节点设置成集群模式，这种集群模式没有中心节点，可水平扩展，据官方文档称可以线性扩展到上万个节点(**官方推荐不超过1000个节点**)。redis集群的性能和高可用性均优于之前版本的哨兵模式，且集群配置非常简单。

Redis集群数据hash分片算法是怎么回事

Redis Cluster 将所有数据划分为 16384 个 slots(槽位)，每个节点负责其中一部分槽位。槽位的信息存储于每个节点中。

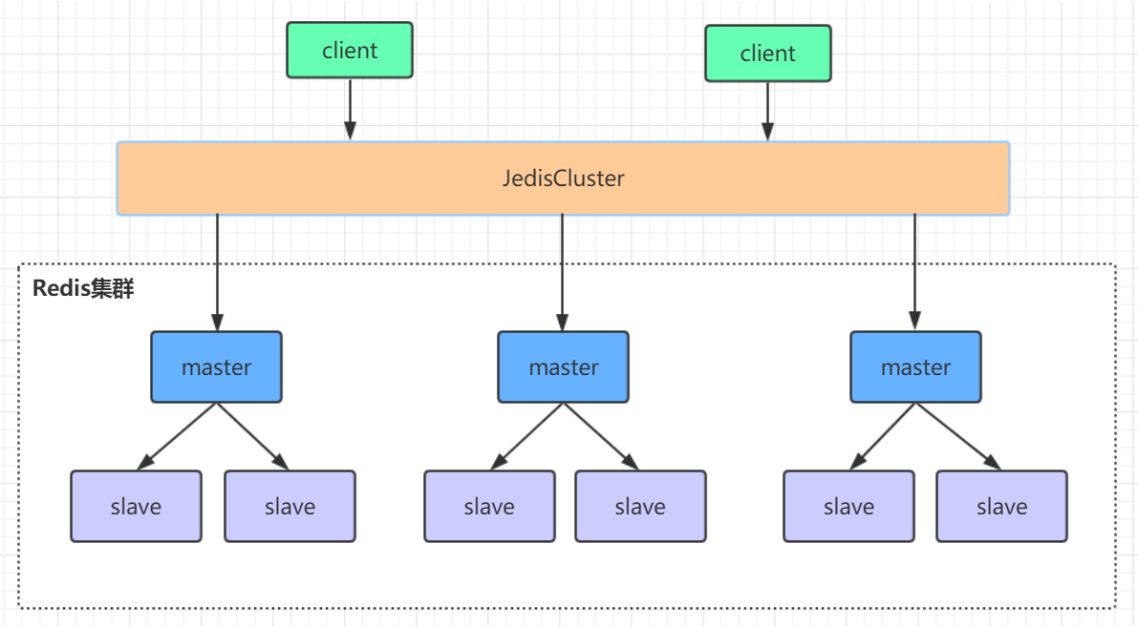
当 Redis Cluster 的客户端来连接集群时，它也会得到一份集群的槽位配置信息并将其缓存在客户端本地。这样当客户端要查找某个 key 时，可以根据槽位定位算法定位到目标节点。

槽位定位算法

Cluster 默认会对 key 值使用 crc16 算法进行 hash 得到一个整数值，然后用这个整数值对 16384 进行取模来得到具体槽位。

HASH_SLOT = CRC16(key) mod 16384

再根据槽位值和Redis节点的对应关系就可以定位到key具体是落在哪个Redis节点上的。



Redis执行命令竟然有死循环阻塞Bug

如果你想随机查看 Redis 中的一个 key，Redis里有一个 RANDOMKEY 命令可以从 Redis 中随机取出一个 key，这个命令可能导致Redis死循环阻塞。

前面的面试题讲过Redis对于过期Key的清理策略是定时删除与惰性删除两种方式结合来做的，而 RANDOMKEY 在随机拿出一个 key 后，首先会先检查这个 key 是否已过期，如果该 key 已经过期，那么 Redis 会删除它，这个过程就是惰性删除。但清理完了还不能结束，Redis 还要找出一个没过期的 key，返回给客户端。

此时，Redis 则会继续随机拿出一个 key，然后再判断它是否过期，直到找出一个没过期的 key 返回给客户端。

这里就有一个问题了，如果此时 Redis 中，有大量 key 已经过期，但还未来得及被清理掉，那这个循环就会持续很久才能结束，而且，这个耗时都花费在了清理过期 key 以及寻找不过期 key 上，导致的结果就是，RANDOMKEY 执行耗时变长，影响 Redis 性能。

以上流程，其实是在 master 上执行的。

如果在 slave 上执行 RANDOMEKY，那么问题会更严重。

slave 自己是不会清理过期 key，当一个 key 要过期时，master 会先清理删除它，之后 master 向 slave 发送一个 DEL 命令，告知 slave 也删除这个 key，以此达到主从库的数据

一致性。

假设Redis 中存在大量已过期还未被清理的 key，那在 slave 上执行 RANDOMKEY 时，就会发生以下问题：

1、slave 随机取出一个 key，判断是否已过期。

2、key 已过期，但 slave 不会删除它，而是继续随机寻找不过期的 key。

3、由于大量 key 都已过期，那 slave 就会寻找不到符合条件的 key，此时就会陷入死循环。

也就是说，在 slave 上执行 RANDOMKEY，有可能会造成整个 Redis 实例卡死。

这其实是 Redis 的一个 Bug，这个 Bug 一直持续到 5.0 才被修复，修复的解决方案就是在slave中最多找一定的次数，无论是否能找到，都会退出循环。

一次线上事故，Redis主从切换导致了缓存雪崩

我们假设，slave 的机器时钟比 master 走得快很多。

此时，Redis master里设置了过期时间的key，从 slave 角度来看，可能会有很多在 master 里没过期的数据其实已经过期了。

如果此时操作主从切换，把 slave 提升为新的 master。

它成为 master 后，就会开始大量清理过期 key，此时就会导致以下结果：

1. master 大量清理过期 key，主线程可能会发生阻塞，无法及时处理客户端请求。
2. Redis 中数据大量过期，引发缓存雪崩。

当 master与slave 机器时钟严重不一致时，对业务的影响非常大。

所以，我们一定要保证主从库的机器时钟一致性，避免发生这些问题。

Redis持久化RDB、AOF、混合持久化是怎么回事

RDB快照（snapshot）

在默认情况下， Redis 将内存数据库快照保存在名字为 dump.rdb 的二进制文件中。

你可以对 Redis 进行设置， 让它在“ N 秒内数据集至少有 M 个改动”这一条件被满足时， 自动保存一次数据集。
比如说， 以下设置会让 Redis 在满足“ 60 秒内有至少有 1000 个键被改动”这一条件时， 自动保存一次数据集：
 # save 60 1000 //关闭RDB只需要将所有的save保存策略注释掉即可

还可以手动执行命令生成RDB快照，进入redis客户端执行命令**save**或**bgsave**可以生成dump.rdb文件，每次命令执行都会将所有redis内存快照到一个新的rdb文件里，并覆盖原有rdb快照文件。

bgsave的写时复制(COW)机制
Redis 借助操作系统提供的写时复制技术（Copy-On-Write, COW），在生成快照的同时，依然可以正常处理写命令。简单来说，bgsave 子进程是由主线程 fork 生成的，可以共享主线程的所有内存数据。bgsave 子进程运行后，开始读取主线程的内存数据，并把它们写入 RDB 文件。此时，如果主线程对这些数据也都是读操作，那么，主线程和 bgsave 子进程相互不影响。但是，如果主线程要修改一块数据，那么，这块数据就会被复制一份，生成该数据的副本。然后，bgsave 子进程会把这个副本数据写入 RDB 文件，而在这个过程中，主线程仍然可以直接修改原来的数据。

save与bgsave对比：

命令	**save**	**bgsave**
IO类型	同步	异步
是否阻塞redis其它命令	是	否(在生成子进程执行调用fork函数时会有短暂阻塞)
复杂度	O(n)	O(n)
优点	不会消耗额外内存	不阻塞客户端命令
缺点	阻塞客户端命令	需要fork子进程，消耗内存

配置自动生成rdb文件后台使用的是bgsave方式。

AOF（append-only file)
快照功能并不是非常耐久（durable）： 如果 Redis 因为某些原因而造成故障停机， 那么服务器将丢失最近写入、且仍未保存到快照中的那些数据。从 1.1 版本开始， Redis 增加了一种完全耐久的持久化方式： AOF 持久化，将**修改的**每一条指令记录进文件appendonly.aof中(先写入os cache，每隔一段时间fsync到磁盘)。

比如执行命令“**set zhuge 666**”，aof文件里会记录如下数据：

```
*3
$3
set
$5
zhuge
$3
666
```

这是一种resp协议格式数据，星号后面的数字代表命令有多少个参数，$号后面的数字代表这个参数有几个字符。

注意，如果执行带过期时间的set命令，aof文件里记录的是并不是执行的原始命令，而是记录key过期的**时间戳。**

比如执行“**set tuling 888 ex 1000**”，对应aof文件里记录如下：

```
*3
$3
set
$6
tuling
$3
888
*3
$9
PEXPIREAT
$6
tuling
$13
1604249786301
```

你可以通过修改配置文件来打开 AOF 功能：

```
# appendonly yes
```

从现在开始， 每当 Redis 执行一个改变数据集的命令时（比如 SET）， 这个命令就会被追加到 AOF 文件的末尾。

这样的话， 当 Redis 重新启动时， 程序就可以通过重新执行 AOF 文件中的命令来达到重建数据集的目的。

你可以配置 Redis 多久才将数据 fsync 到磁盘一次。

有三个选项：

```
appendfsync always：每次有新命令追加到 AOF 文件时就执行一次 fsync ，非常慢，也
非常安全。
```

```
appendfsync everysec：每秒 fsync 一次，足够快，并且在故障时只会丢失 1 秒钟的数据。
appendfsync no：从不 fsync ，将数据交给操作系统来处理。更快，也更不安全的选择。
```

推荐（并且也是默认）的措施为每秒 fsync 一次， 这种 fsync 策略可以兼顾速度和安全性。

AOF重写

AOF文件里可能有太多没用指令，所以AOF会定期根据**内存的最新数据**生成aof文件。

例如，执行了如下几条命令：

```
127.0.0.1:6379> incr readcount
(integer) 1
127.0.0.1:6379> incr readcount
(integer) 2
127.0.0.1:6379> incr readcount
(integer) 3
127.0.0.1:6379> incr readcount
(integer) 4
127.0.0.1:6379> incr readcount
(integer) 5
```

重写后AOF文件里变成

```
*3
$3
SET
$2
readcount
$1
5
```

如下两个配置可以控制AOF自动重写频率

```
# auto-aof-rewrite-min-size 64mb //aof文件至少要达到64M才会自动重写，文件太小恢复速度本来就很快，重写的意义不大
# auto-aof-rewrite-percentage 100 //aof文件自上一次重写后文件大小增长了100%则再次触发重写
```

当然AOF还可以手动重写，进入redis客户端执行命令**bgrewriteaof**重写AOF。

注意，**AOF重写redis会fork出一个子进程去做(与bgsave命令类似)，不会对redis正常命令处理有太多影响。**

RDB 和 AOF ，我应该用哪一个?

命令	RDB	AOF

启动优先级	低	高
体积	小	大
恢复速度	快	慢
数据安全性	容易丢数据	根据策略决定

生产环境可以都启用，redis启动时如果既有rdb文件又有aof文件则优先选择aof文件恢复数据，因为aof一般来说数据更全一点。

Redis 4.0 混合持久化

重启 Redis 时，我们很少使用 RDB来恢复内存状态，因为会丢失大量数据。我们通常使用 AOF 日志重放，但是重放 AOF 日志性能相对 RDB来说要慢很多，这样在 Redis 实例很大的情况下，启动需要花费很长的时间。 Redis 4.0 为了解决这个问题，带来了一个新的持久化选项——混合持久化。

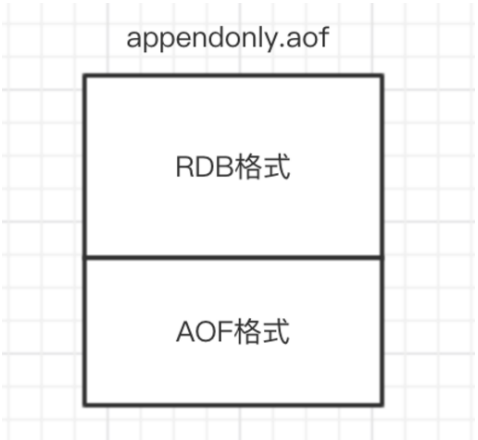
通过如下配置可以开启混合持久化(**必须先开启aof**)：

```
# aof-use-rdb-preamble yes
```

如果开启了混合持久化，**AOF在重写时**，不再是单纯将内存数据转换为RESP命令写入AOF文件，而是将重写**这一刻之前**的内存做RDB快照处理，并且将RDB快照内容和**增量的**AOF修改内存数据的命令存在一起，都写入新的AOF文件，新的文件一开始不叫appendonly.aof，等到重写完新的AOF文件才会进行改名，覆盖原有的AOF文件，完成新旧两个AOF文件的替换。

于是在 Redis 重启的时候，可以先加载 RDB 的内容，然后再重放增量 AOF 日志就可以完全替代之前的 AOF 全量文件重放，因此重启效率大幅得到提升。

混合持久化AOF文件结构如下：



线上Redis持久化策略一般如何设置

如果对性能要求较高，在Master最好不要做持久化，可以在某个Slave开启AOF备份数据，策略设置为每秒同步一次即可。

一次线上事故，Redis主节点宕机导致数据全部丢失

如果你的 Redis 采用如下模式部署，就会发生数据丢失的问题：

- master-slave + 哨兵部署实例。
- master 没有开启数据持久化功能。
- Redis 进程使用 supervisor 管理，并配置为进程宕机，自动重启。

如果此时 master 宕机，就会导致下面的问题：

- master 宕机，哨兵还未发起切换，此时 master 进程立即被 supervisor 自动拉起。
- 但 master 没有开启任何数据持久化，启动后是一个空实例。
- 此时 slave 为了与 master 保持一致，它会自动清空实例中的所有数据，slave 也变成了一个空实例。

在这个场景下，master / slave 的数据就全部丢失了。

这时，业务应用在访问 Redis 时，发现缓存中没有任何数据，就会把请求全部打到后端数据库上，这还会进一步引发缓存雪崩，对业务影响非常大。

这种情况下我们一般不应该给Redis主节点配置进程宕机马上自动重启策略，而应该等哨兵把某个Redis从节点切换为主节点后再重启之前宕机的Redis主节点让其变为slave节点。

Redis线上数据如何备份

1、写crontab定时调度脚本，每小时都copy一份rdb或aof文件到另外一台机器中去，保留最近48小时的备份

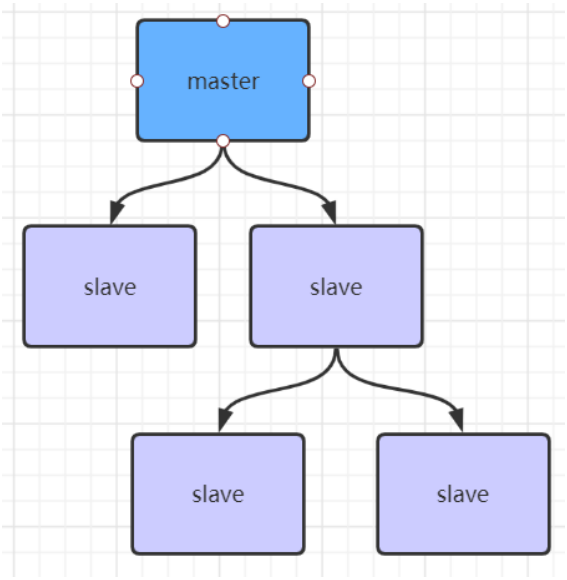
2、每天都保留一份当日的数据备份到一个目录中去，可以保留最近1个月的备份

3、每次copy备份的时候，都把太旧的备份给删了

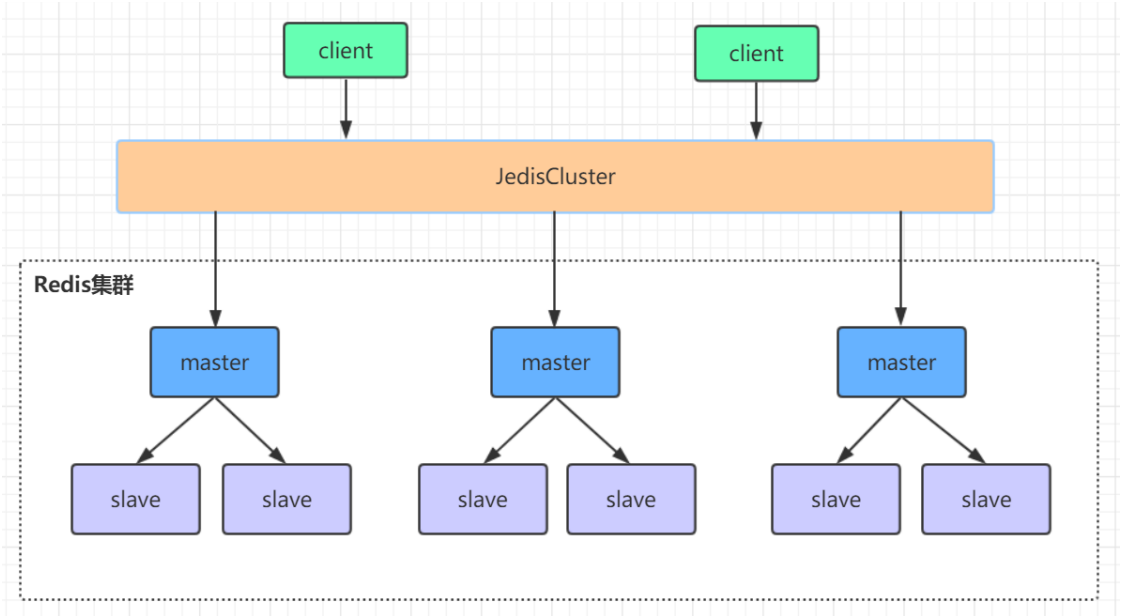
Redis主从复制风暴是怎么回事

如果Redis主节点有很多从节点，在某一时刻如果所有从节点都同时连接主节点，那么主节点会同时把内存快照RDB发给多个从节点，这样会导致Redis主节点压力非常大，这就是所谓的Redis**主从复制风暴问题。**

这种问题我们对Redis主从架构做一些优化得以避免，比如可以做下面这种树形复制架构。



Redis集群网络抖动导致频繁主从切换怎么处理



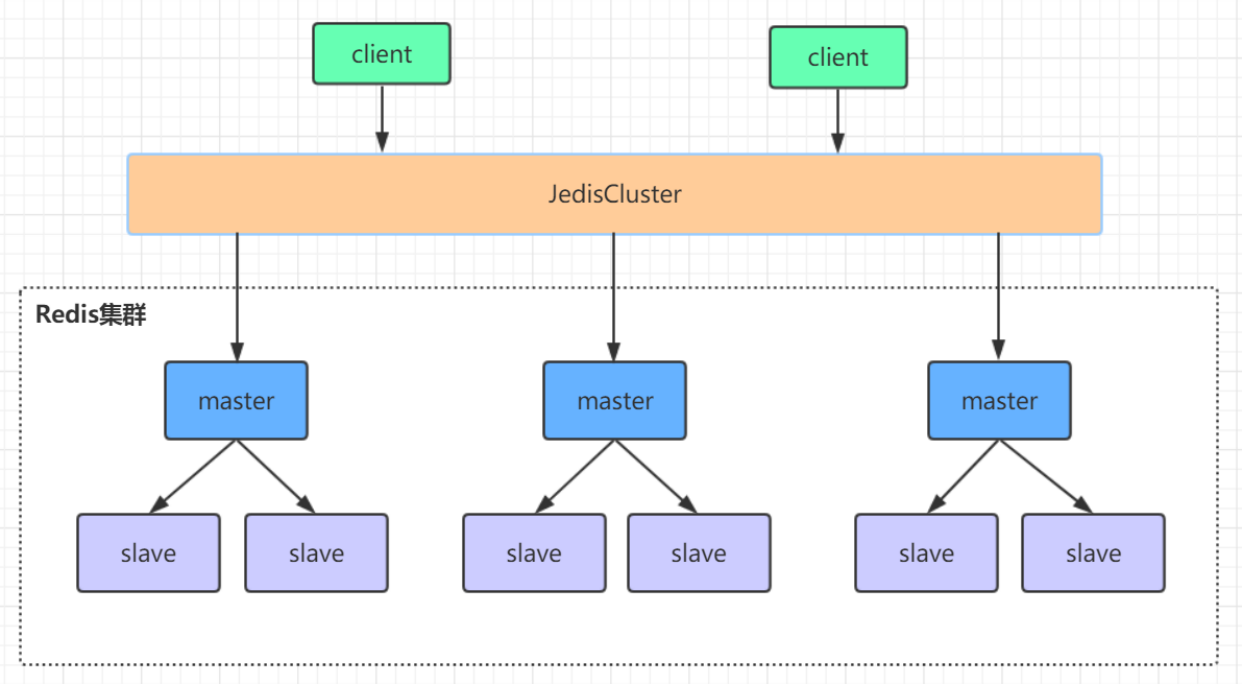
真实世界的机房网络往往并不是风平浪静的，它们经常会发生各种各样的小问题。比如网络抖动就是非常常见的一种现象，突然之间部分连接变得不可访问，然后很快又恢复正常。

为解决这种问题，Redis Cluster 提供了一种选项`cluster-node-timeout`，表示当某个节点持续 timeout 的时间失联时，才可以认定该节点出现故障，需要进行主从切换。如果

没有这个选项，网络抖动会导致主从频繁切换 (数据的重新复制)。

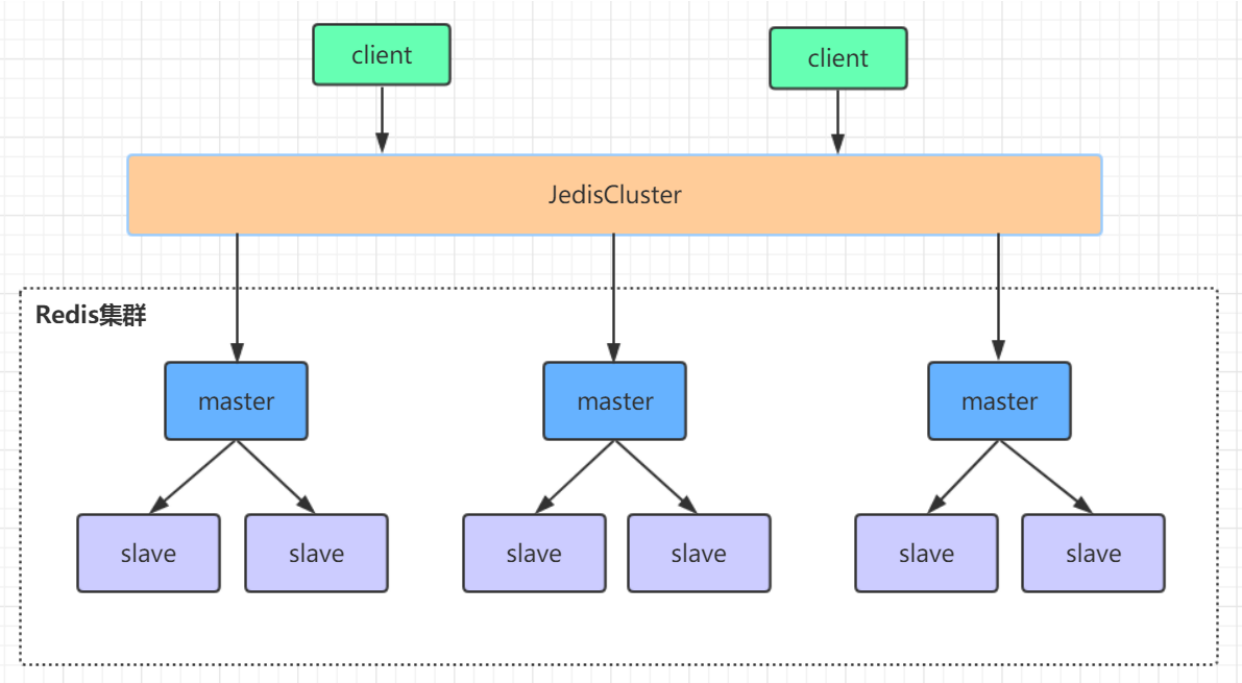
Redis集群为什么至少需要三个master节点

因为新master的选举需要大于半数的集群master节点同意才能选举成功，如果只有两个master节点，当其中一个挂了，是达不到选举新master的条件的。



Redis集群为什么推荐奇数个节点

因为新master的选举需要大于半数的集群master节点同意才能选举成功，奇数个master节点可以在满足选举该条件的基础上节省一个节点，比如三个master节点和四个master节点的集群相比，大家如果都挂了一个master节点都能选举新master节点，如果都挂了两个master节点都没法选举新master节点了，所以奇数的master节点更多的是**从节省机器资源角度出发**说的。



Redis集群支持批量操作命令吗

对于类似mset，mget这样的多个key的原生批量操作命令，redis集群只支持所有key落在同一slot的情况，如果有多个key一定要用mset命令在redis集群上操作，则可以在key的前面加上{XXX}，这样参数数据分片hash计算的只会是大括号里的值，这样能确保不同的key能落到同一slot里去，示例如下：

```
mset {user1}:1:name zhuge {user1}:1:age 18
```

假设name和age计算的hash slot值不一样，但是这条命令在集群下执行，redis只会用大括号里的 user1 做hash slot计算，所以算出来的slot值肯定相同，最后都能落在同一slot。

Lua脚本能在Redis集群里执行吗

Redis官方规定Lua脚本如果想在Redis集群里执行，需要Lua脚本里操作的所有Redis Key落在集群的同一个节点上，这种的话我们可以给Lua脚本的Key前面加一个相同的hash tag，就是{XXX}，这样就能保证Lua脚本里所有Key落在相同的节点上了。

下面的面试题偏重实战，以代码为主，代码在视频资料里

Redis主从切换导致分布式锁丢失问题是怎么回事

Redlock如何解决Redis主从切换分布式锁丢失问题

中小公司Redis缓存架构以及线上问题分析

大厂线上大规模商品缓存数据冷热分离实战

实战解决大规模缓存击穿导致线上数据库压力暴增

面试常问的缓存穿透是怎么回事

基于DCL机制解决突发性热点缓存并发重建问题实战

Redis分布式锁解决缓存与数据库双写不一致问题实战

大促压力暴增导致分布式锁串行争用问题优化实战

利用多级缓存架构解决Redis线上集群缓存雪崩问题