> 图文详解 53 道 Redis 面试高频题，这次吊打面试官，我觉得稳了（手动 dog）。整理：楼仔，作者：三分恶，戳原文链接。

基础

1.说说什么是Redis?

Redis是一种基于键值对（key-value）的NoSQL数据库。

比一般键值对数据库强大的地方，Redis中的value支持string（字符串）、hash（哈希）、 list（列表）、set（集合）、zset（有序集合）、Bitmaps（位图）、 HyperLogLog、GEO（地理信息定位）等多种数据结构，因此 Redis可以满足很多的应用场景。

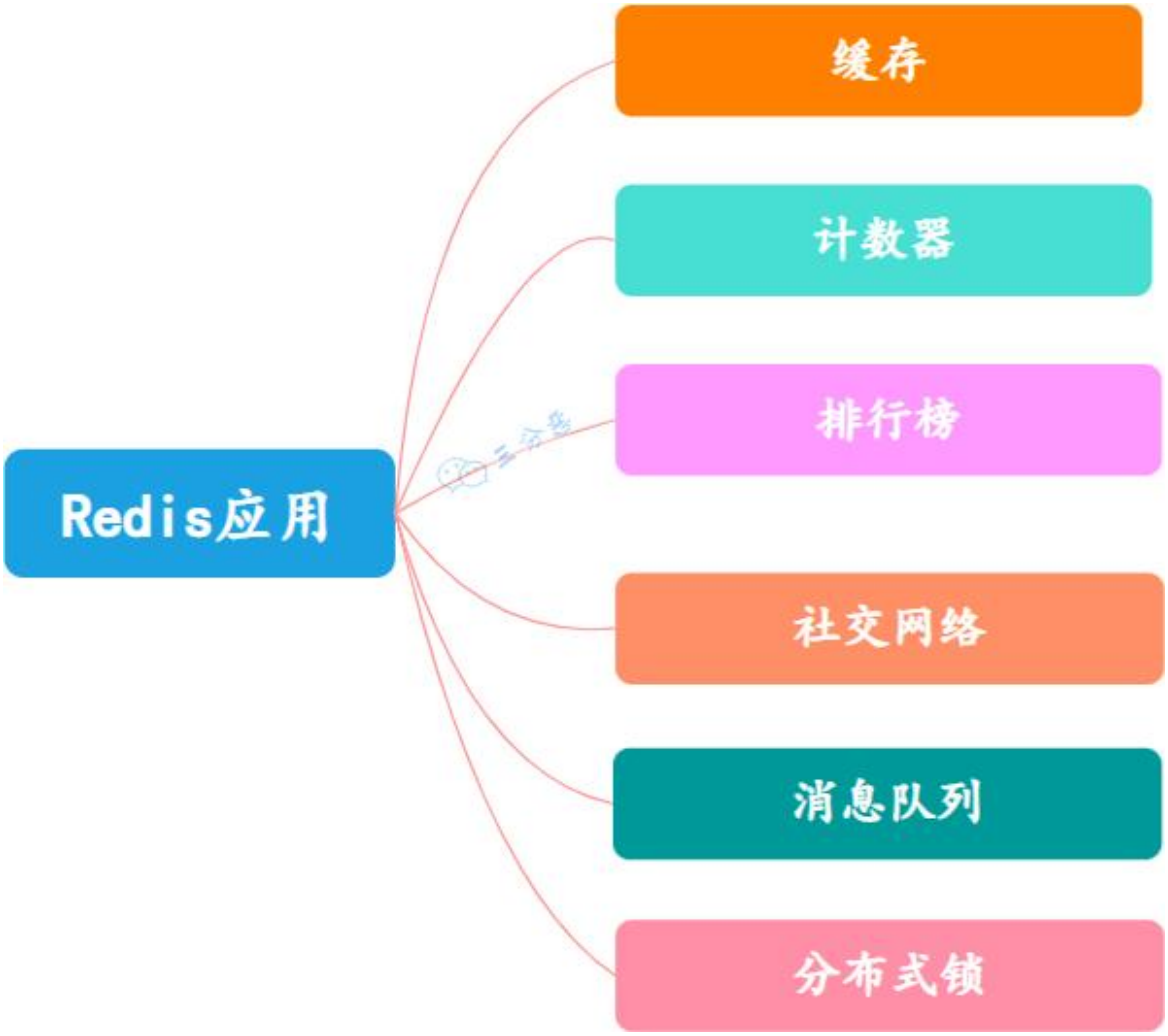
而且因为Redis会将所有数据都存放在内存中，所以它的读写性能非常出色。

不仅如此，Redis还可以将内存的数据利用快照和日志的形式保存到硬盘上，这样在发生类似断电或者机器故障的时候，内存中的数据不会“丢失”。

除了上述功能以外，Redis还提供了键过期、发布订阅、事务、流水线、Lua脚本等附加功能。总

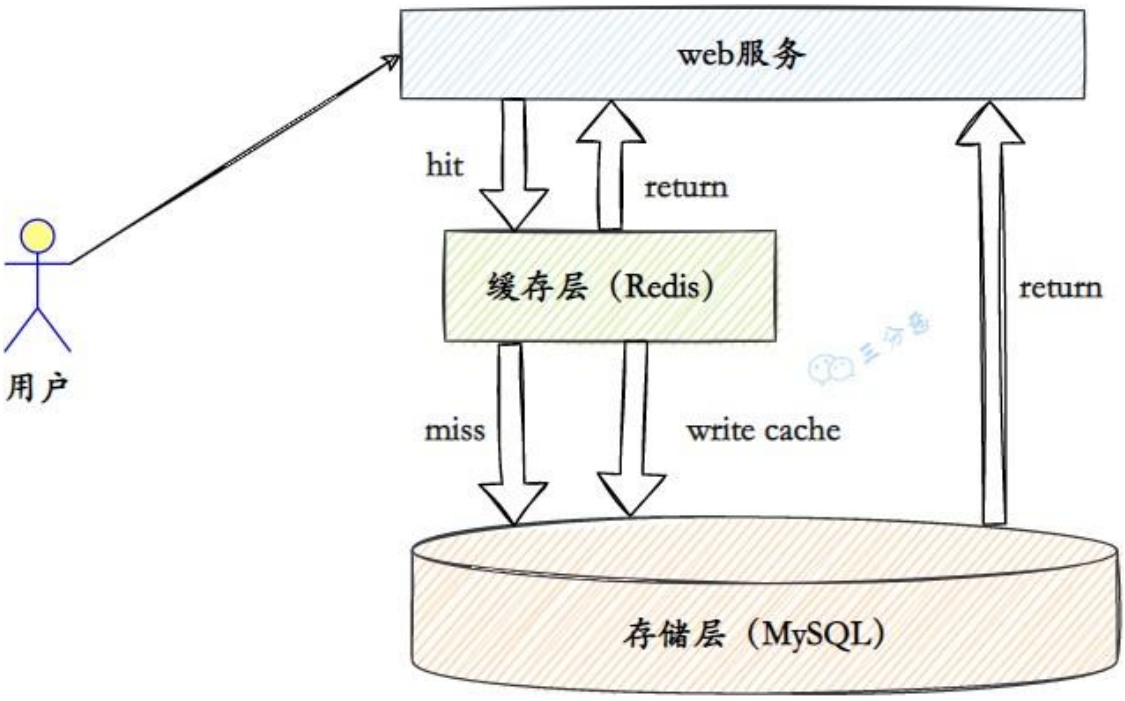
之，Redis是一款强大的性能利器。

2.Redis可以用来干什么？



1. 缓存

这是Redis应用最广泛地方，基本所有的Web应用都会使用Redis作为缓存，来降低数据源压力，提高响应速度。

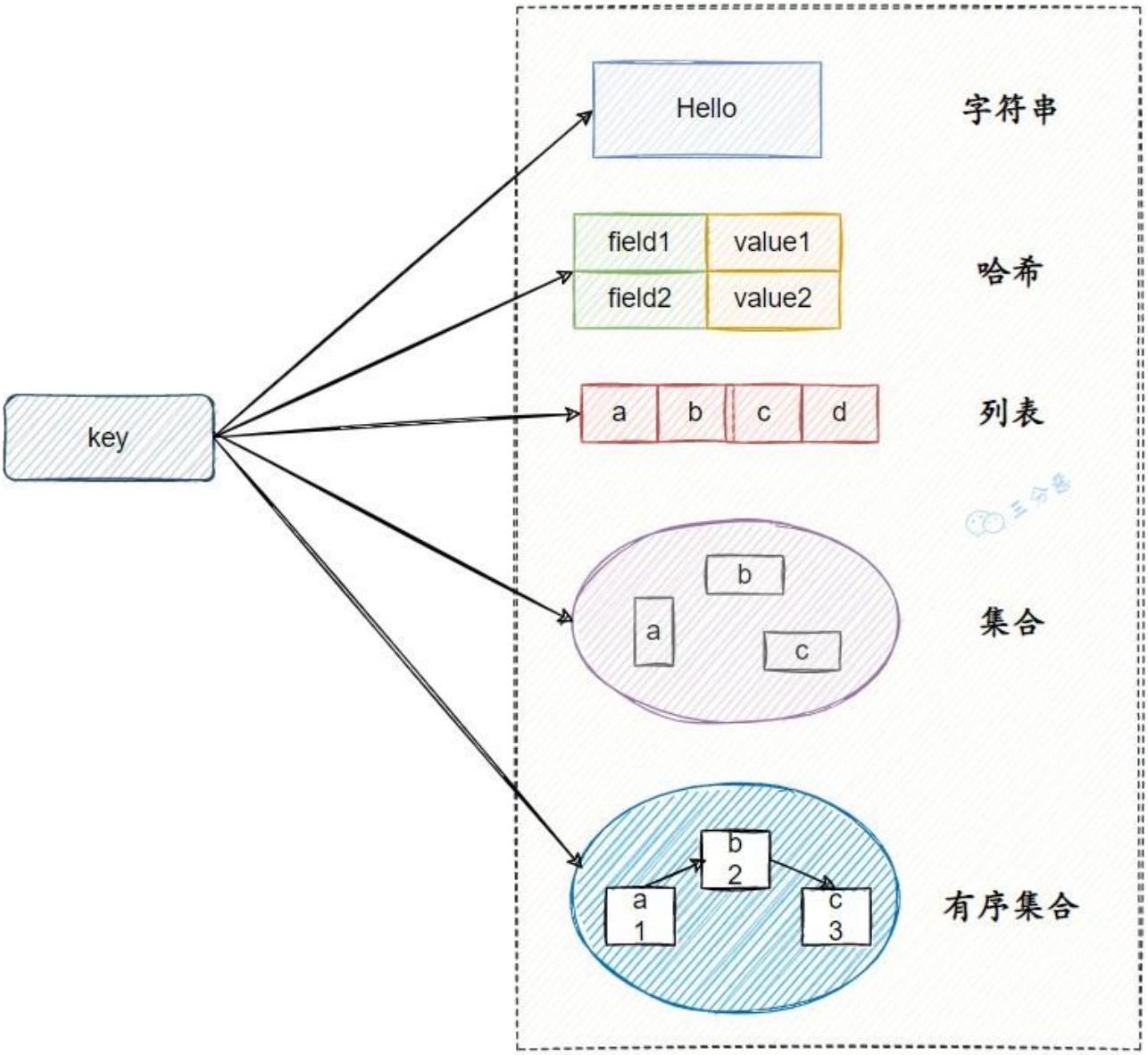


2. 计数器
   Redis天然支持计数功能，而且计数性能非常好，可以用来记录浏览量、点赞量等等。
3. 排行榜
   Redis提供了列表和有序集合数据结构，合理地使用这些数据结构可以很方便地构建各种排行榜系统。
4. 社交网络
   赞/踩、粉丝、共同好友/喜好、推送、下拉刷新。
5. 消息队列
   Redis提供了发布订阅功能和阻塞队列的功能，可以满足一般消息队列功能。
6. 分布式锁
   分布式环境下，利用Redis实现分布式锁，也是Redis常见的应用。

Redis的应用一般会结合项目去问，以一个电商项目的用户服务为例：

- Token存储：用户登录成功之后，使用Redis存储Token
- 登录失败次数计数：使用Redis计数，登录失败超过一定次数，锁定账号
- 地址缓存：对省市区数据的缓存
- 分布式锁：分布式环境下登录、注册等操作加分布式锁
- ……

3.Redis 有哪些数据结构？



Redis有五种基本数据结构。

string

字符串最基础的数据结构。字符串类型的值实际可以是字符串（简单的字符串、复杂的字符串（例如JSON、XML））、数字（整数、浮点数），甚至是二进制（图片、音频、视频），但是值最大不能超过512MB。字符串主要有以下几个典型使用场景：

- 缓存功能
- 计数
- 共享Session
- 限速

hash

哈希类型是指键值本身又是一个键值对结构。

哈希主要有以下典型应用场景：

- 缓存用户信息

- 缓存对象

list

列表（list）类型是用来存储多个有序的字符串。列表是一种比较灵活的数据结构，它可以充当栈和队列的角色列

表主要有以下几种使用场景：

- 消息队列
- 文章列表

set

集合（set）类型也是用来保存多个的字符串元素，但和列表类型不一 样的是，集合中不允许有重复元素，并且集合中的元素是无序的。

集合主要有如下使用场景：

- 标签（tag）
- 共同关注

sorted set

有序集合中的元素可以排序。但是它和列表使用索引下标作为排序依据不同的是，它给每个元素设置一个权重（score）作为排序的依据。

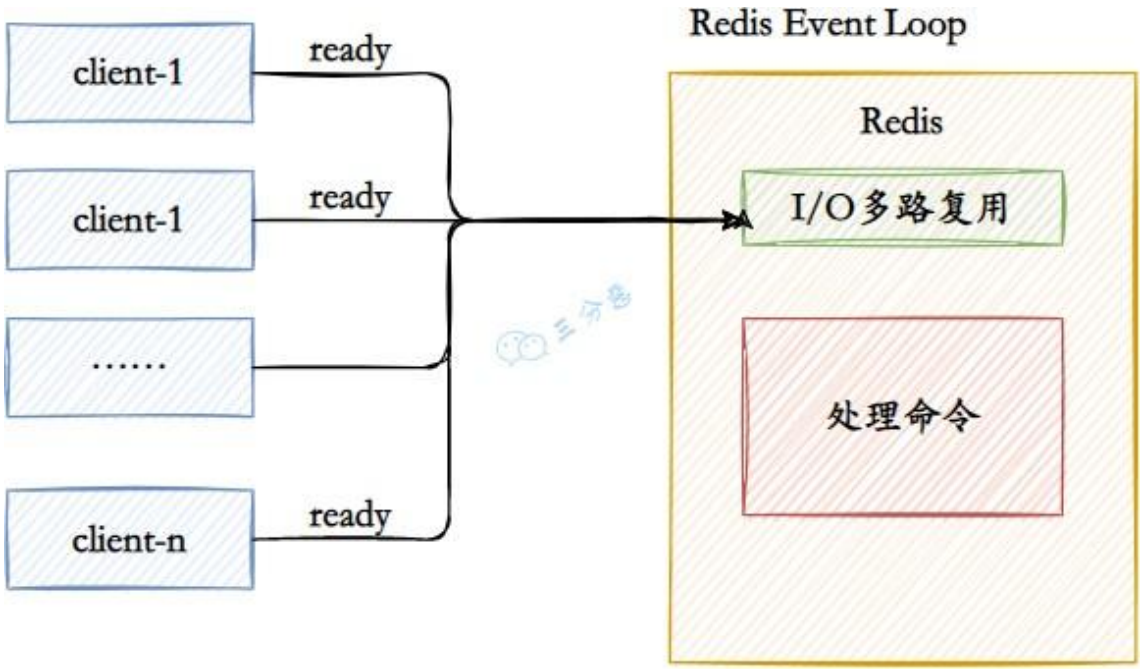
有序集合主要应用场景：

- 用户点赞统计
- 用户排序

4.Redis为什么快呢？

Redis的速度非常的快，单机的Redis就可以支撑每秒十几万的并发，相对于MySQL来说，性能是MySQL的几十倍。速度快的原因主要有几点：

1. **完全基于内存操作**
2. 使用单线程，避免了线程切换和竞态产生的消耗
3. 基于非阻塞的IO多路复用机制
4. C语言实现，优化过的数据结构，基于几种基础的数据结构，redis做了大量的优化，性能极高



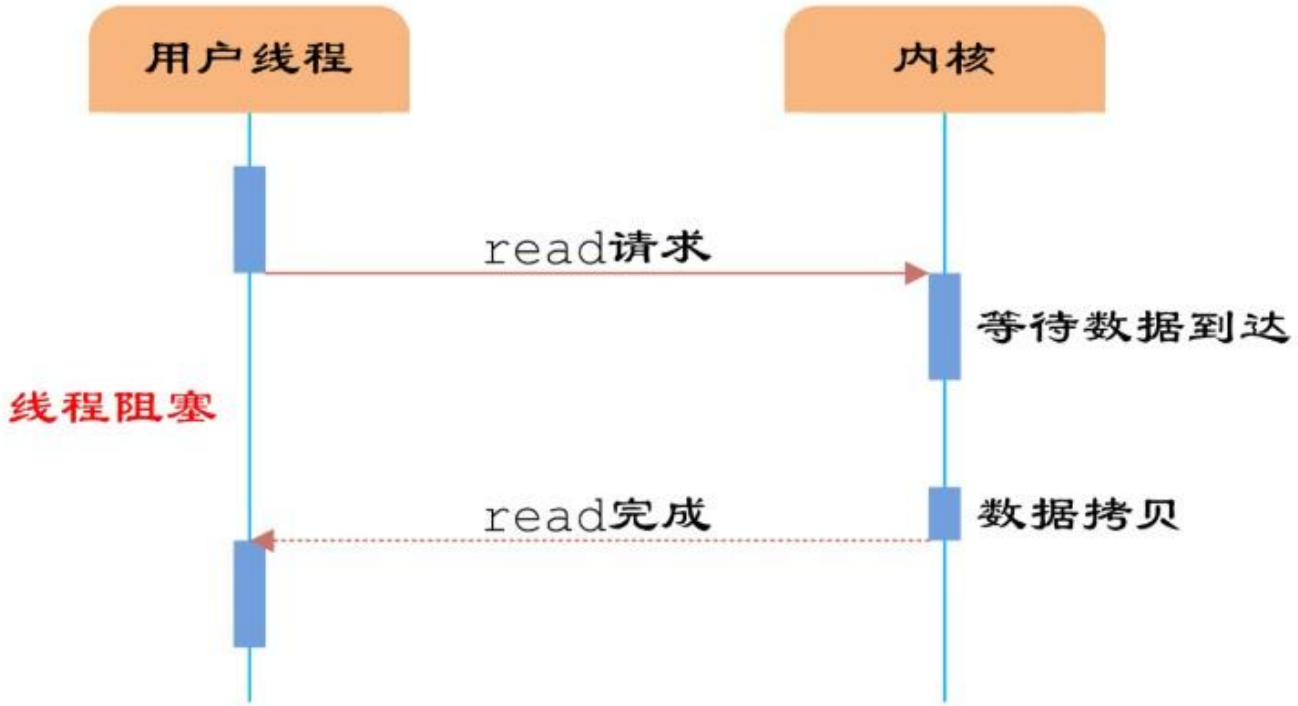
5.能说一下I/O多路复用吗？

引用知乎上一个高赞的回答来解释什么是I/O多路复用。假设你是一个老师，让30个学生解答一道题目，然后检查学生做的是否正确，你有下面几个选择：

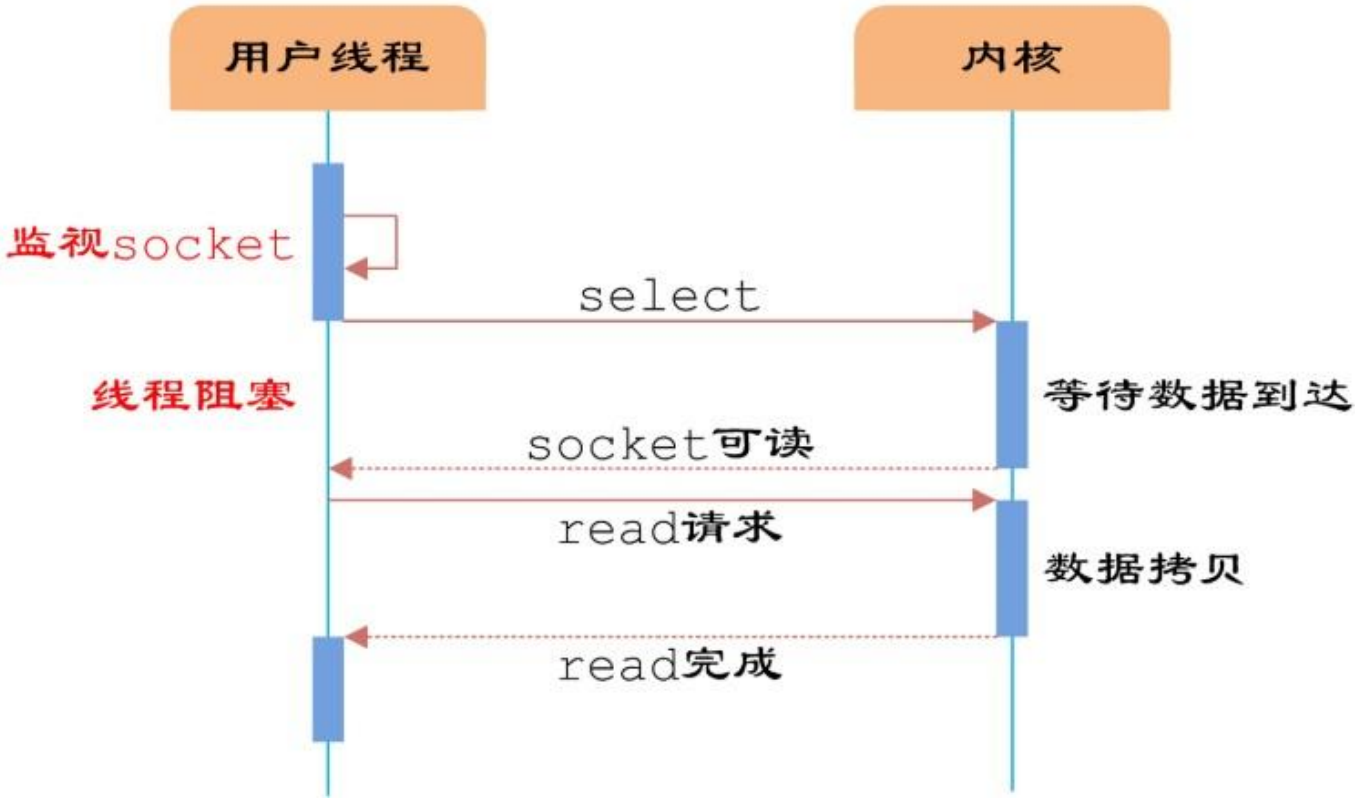
- 第一种选择：按顺序逐个检查，先检查A，然后是B，之后是C、D。。。这中间如果有一个学生卡住，全班都会被耽误。这种模式就好比，你用循环挨个处理socket，根本不具有并发能力。
- 第二种选择：你创建30个分身，每个分身检查一个学生的答案是否正确。 这种类似于为每一个用户创建一个进程或者线程处理连接。
- 第三种选择，你站在讲台上等，谁解答完谁举手。这时C、D举手，表示他们解答问题完毕，你下去依次检查C、D的答案，然后继续回到讲台上等。此时E、A又举手，然后去处理E和A。

第一种就是阻塞IO模型，第三种就是I/O复用模型。

阻塞I/O模型



I/O多路复用模型



Linux系统有三种方式实现IO多路复用：select、poll和epoll。

例如epoll方式是将用户socket对应的fd注册进epoll，然后epoll帮你监听哪些socket上有消息到达，这样就避免了大量的无用操作。此时的socket应该采用非阻塞模式。

这样，整个过程只在进行select、poll、epoll这些调用的时候才会阻塞，收发客户消息是不会阻塞的，整个进程或者线程就被充分利用起来，这就是事件驱动，所谓的reactor模式。

6. Redis为什么早期选择单线程？

官方解释：https://redis.io/topics/faq

> **Redis is single threaded. How can I exploit multiple CPU / cores?**
>
> It's not very frequent that CPU becomes your bottleneck with Redis, as usually Redis is either memory or network bound. For instance, using pipelining Redis running on an average Linux system can deliver even 1 million requests per second, so if your application mainly uses O(N) or O(log(N)) commands, it is hardly going to use too much CPU.
>
> However, to maximize CPU usage you can start multiple instances of Redis in the same box and treat them as different servers. At some point a single box may not be enough anyway, so if you want to use multiple CPUs you can start thinking of some way to shard earlier.
>
> You can find more information about using multiple Redis instances in the Partitioning page.
>
> However with Redis 4.0 we started to make Redis more threaded. For now this is limited to deleting objects in the background, and to blocking commands implemented via Redis modules. For future releases, the plan is to make Redis more and more threaded.

官方FAQ表示，因为Redis是基于内存的操作，CPU成为Redis的瓶颈的情况很少见，Redis的瓶颈最有可能是内存的大小或者网络限制。

如果想要最大程度利用CPU，可以在一台机器上启动多个Redis实例。

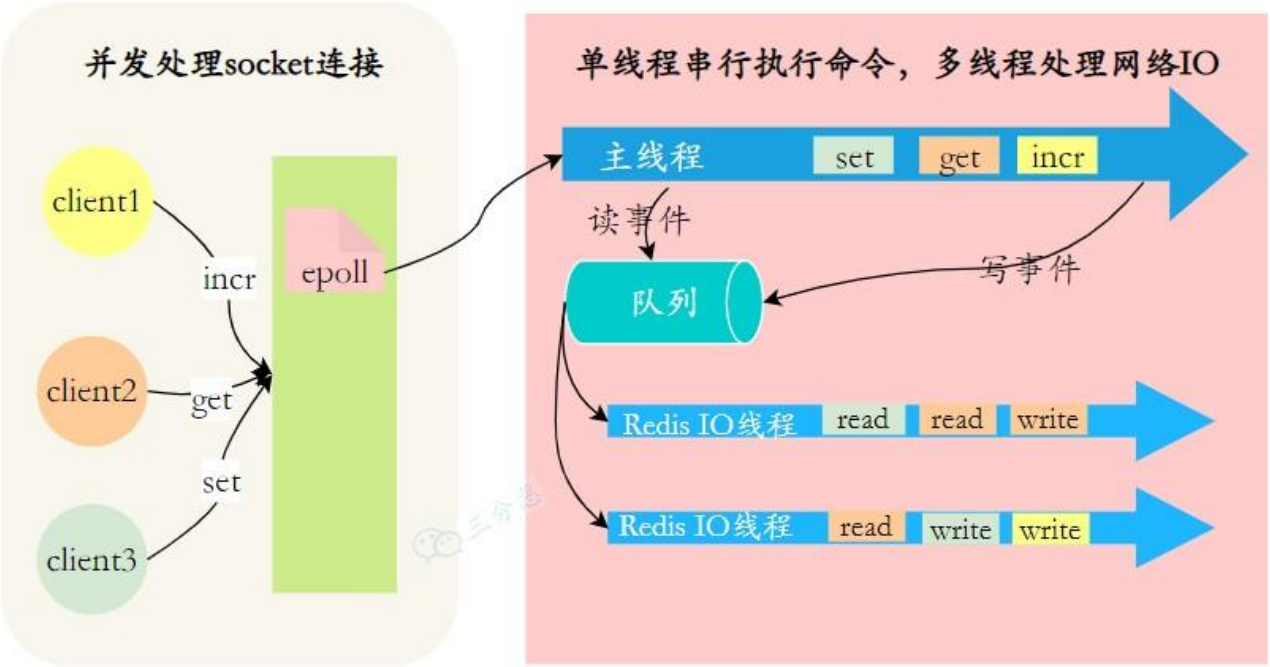
PS：网上有这样的回答，吐槽官方的解释有些敷衍，其实就是历史原因，开发者嫌多线程麻烦，后来这个CPU的利用问题就被抛给了使用者。

同时FAQ里还提到了， Redis 4.0 之后开始变成多线程，除了主线程外，它也有后台线程在处理一些较为缓慢的操作，例如清理脏数据、无用连接的释放、大 Key 的删除等等。

7. Redis6.0使用多线程是怎么回事？

Redis不是说用单线程的吗？怎么6.0成了多线程的？

Redis6.0的多线程是用多线程来处理数据的**读写和协议解析**，但是Redis**执行命令**还是单线程的。

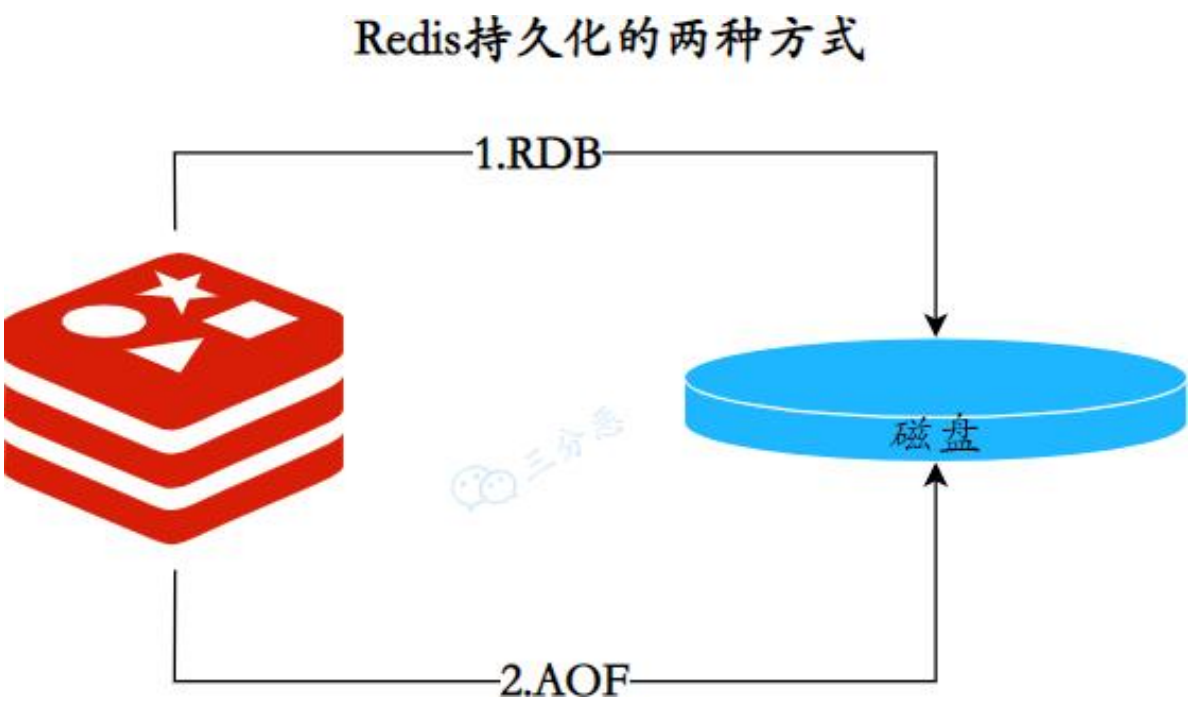


这样做的目的是因为Redis的性能瓶颈在于网络IO而非CPU，使用多线程能提升IO读写的效率，从而整体提高Redis的性能。

持久化

8.Redis持久化方式有哪些？有什么区别？

Redis持久化方案分为RDB和AOF两种。

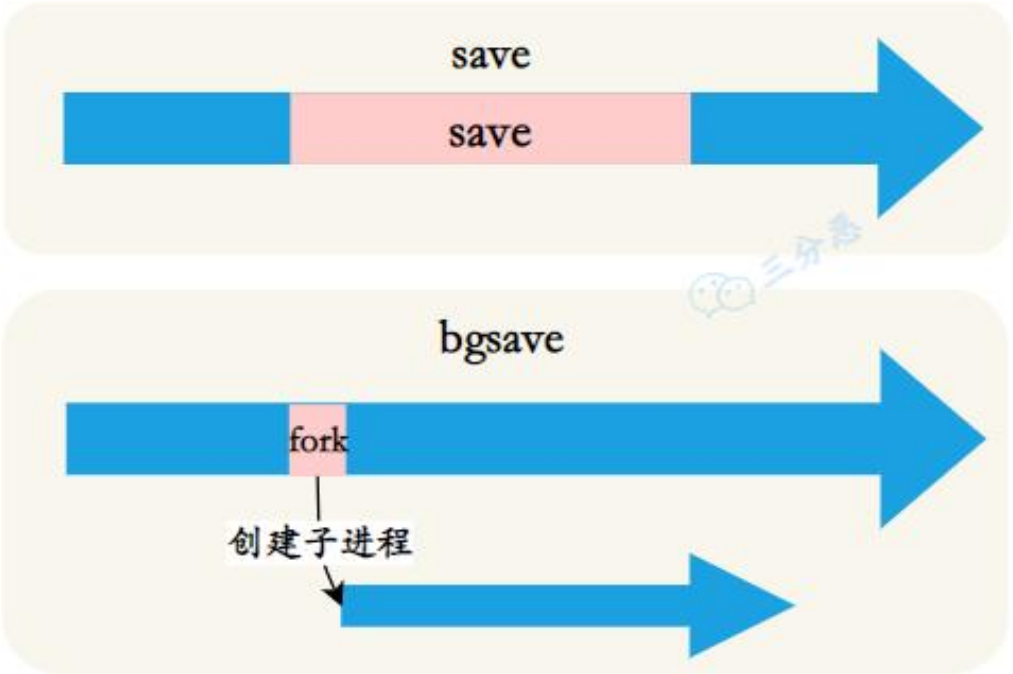


RDB

RDB持久化是把当前进程数据生成**快照**保存到硬盘的过程，触发RDB持久化过程分为手动触发和自动触发。

RDB文件是一个压缩的二进制文件，通过它可以还原某个时刻数据库的状态。由于RDB文件是保存在硬盘上的，所以即使Redis崩溃或者退出，只要RDB文件存在，就可以用它来恢复还原数据库的状态。

手动触发分别对应save和bgsave命令:



- save命令：阻塞当前Redis服务器，直到RDB过程完成为止，对于内存比较大的实例会造成长时间阻塞，线上环境不建议使用。
- bgsave命令：Redis进程执行fork操作创建子进程，RDB持久化过程由子进程负责，完成后自动结束。阻塞只发生在fork阶段，一般时间很短。

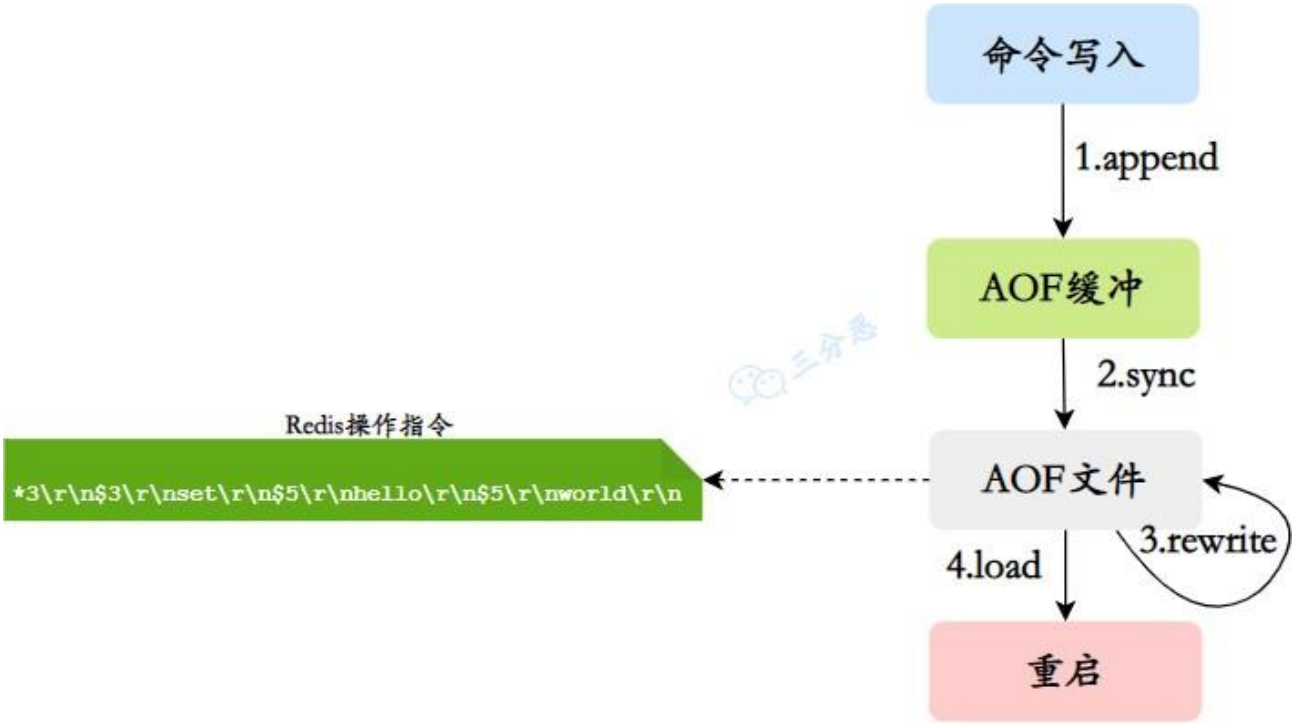
以下场景会自动触发RDB持久化：

- 使用save相关配置，如“save m n”。表示m秒内数据集存在n次修改时，自动触发bgsave。
- 如果从节点执行全量复制操作，主节点自动执行bgsave生成RDB文件并发送给从节点
- 执行debug reload命令重新加载Redis时，也会自动触发save操作
- 默认情况下执行shutdown命令时，如果没有开启AOF持久化功能则自动执行bgsave。

AOF

AOF（append only file）持久化：以独立日志的方式记录每次写命令， 重启时再重新执行AOF文件中的命令达到恢复数据的目的。AOF的主要作用是解决了数据持久化的实时性，目前已经是Redis持久化的主流方式。

AOF的工作流程操作：命令写入 （append）、文件同步（sync）、文件重写（rewrite）、重启加载 （load）



流程如下：

1）所有的写入命令会追加到aof_buf（缓冲区）中。

2）AOF缓冲区根据对应的策略向硬盘做同步操作。

3）随着AOF文件越来越大，需要定期对AOF文件进行重写，达到压缩 的目的。

4）当Redis服务器重启时，可以加载AOF文件进行数据恢复。

9.RDB 和 AOF 各自有什么优缺点？

RDB | **优点**

1. 只有一个紧凑的二进制文件 `dump.rdb`，非常适合备份、全量复制的场景。
2. **容灾性好**，可以把RDB文件拷贝道远程机器或者文件系统张，用于容灾恢复。
3. **恢复速度快**，RDB恢复数据的速度远远快于AOF的方式

RDB | **缺点**

1. **实时性低**，RDB 是间隔一段时间进行持久化，没法做到实时持久化/秒级持久化。如果在这一间隔事件发生故障，数据会丢失。
2. **存在兼容问题**，Redis演进过程存在多个格式的RDB版本，存在老版本Redis无法兼容新版本RDB的问题。

AOF | **优点**

1. **实时性好**，aof 持久化可以配置 `appendfsync` 属性，有 `always`，每进行一次命令操作就记录到 aof 文件中一次。
2. 通过 append 模式写文件，即使中途服务器宕机，可以通过 redis-check-aof 工具解决数据一致性问题。

AOF | **缺点**

1. AOF 文件比 RDB **文件大**，且 **恢复速度慢**。
2. **数据集大** 的时候，比 RDB **启动效率低**。

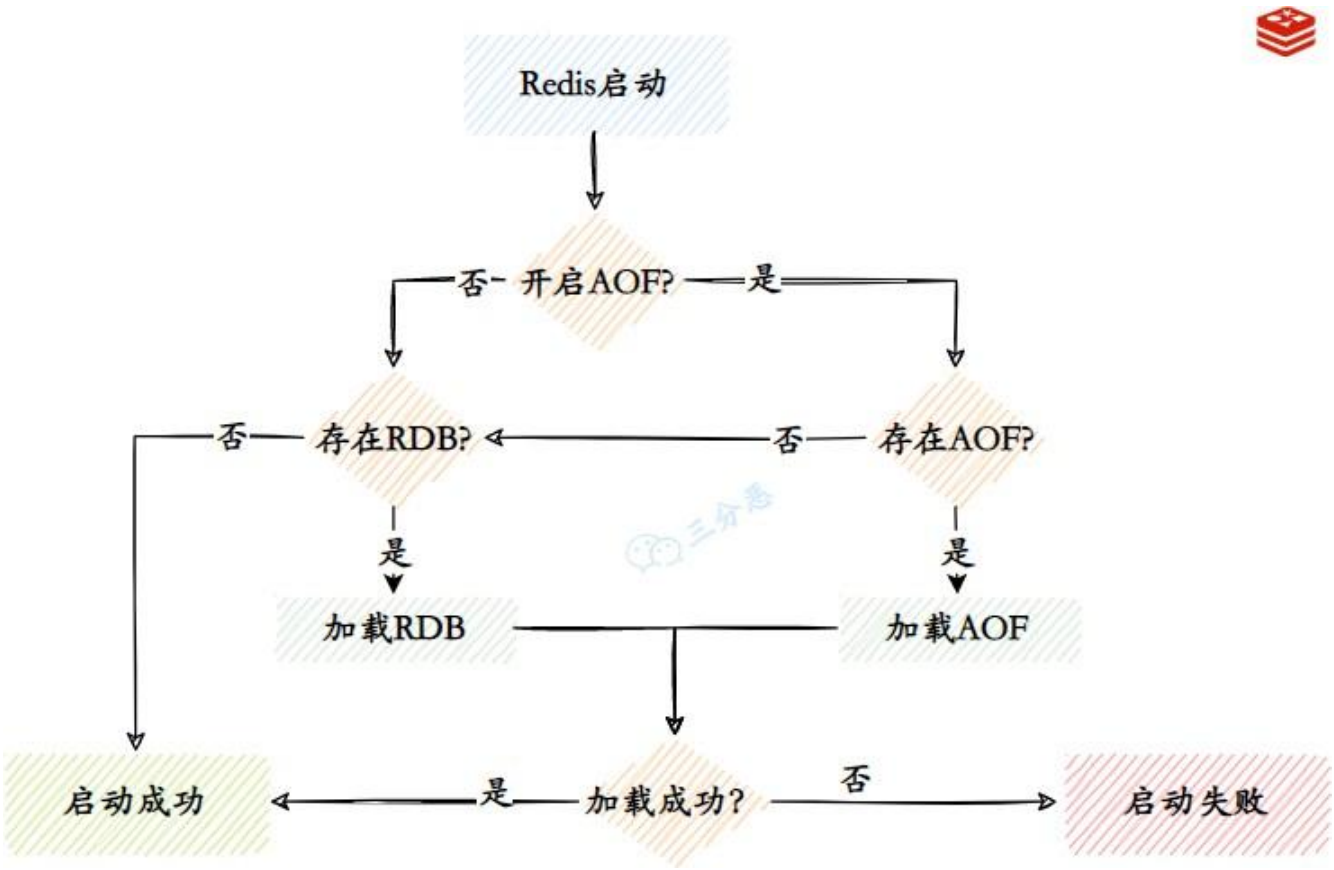
10.RDB和AOF如何选择？

- 一般来说， 如果想达到足以媲美数据库的 **数据安全性**，应该 **同时使用两种持久化功能**。在这种情况下，当 Redis 重启的时候会优先载入 AOF 文件来恢复原始的数据，因为在通常情况下 AOF 文件保存的数据集要比 RDB 文件保存的数据集要完整。
- 如果 **可以接受数分钟以内的数据丢失**，那么可以 **只使用** RDB **持久化**。
- 有很多用户都只使用 AOF 持久化，但并不推荐这种方式，因为定时生成 RDB 快照（snapshot）非常便于进行数据备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快，除此之外，使用 RDB 还可以避免 AOF 程序的 bug。
- 如果只需要数据在服务器运行的时候存在，也可以不使用任何持久化方式。

11.Redis的数据恢复？

当Redis发生了故障，可以从RDB或者AOF中恢复数据。

恢复的过程也很简单，把RDB或者AOF文件拷贝到Redis的数据目录下，如果使用AOF恢复，配置文件开启AOF，然后启动redis-server即可。



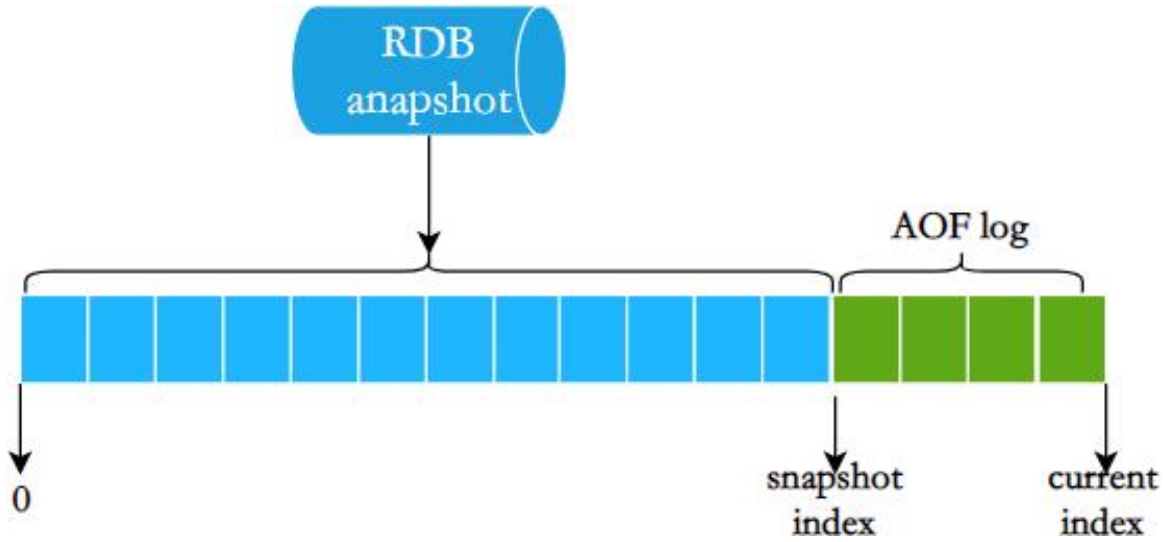
Redis 启动时加载数据的流程：

1. AOF持久化开启且存在AOF文件时，优先加载AOF文件。
2. AOF关闭或者AOF文件不存在时，加载RDB文件。
3. 加载AOF/RDB文件成功后，Redis启动成功。
4. AOF/RDB文件存在错误时，Redis启动失败并打印错误信息。

12. Redis 4.0 的混合持久化了解吗？

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**。将 `rdb` 文件的内容和增量的 AOF 日志文件存在一起。这里的 AOF 日志不再是全量的日志，而是 **自持久化开始到持久化结束** 的这段时间发生的增量 AOF 日志，通常这部分 AOF 日志很小：

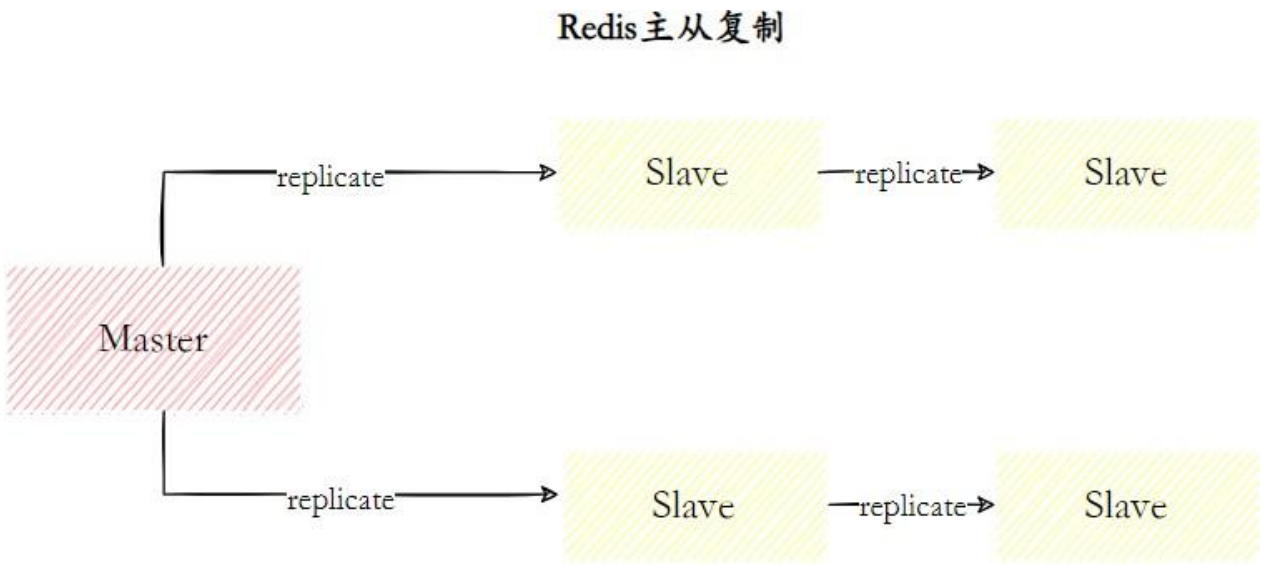


于是在 Redis 重启的时候，可以先加载 `rdb` 的内容，然后再重放增量 AOF 日志就可以完全替代之前的 AOF 全量文件重放，重启效率因此大幅得到提升。

高可用

Redis保证高可用主要有三种方式：主从、哨兵、集群。

13. 主从复制了解吗？



主从复制，是指将一台 Redis 服务器的数据，复制到其他的 Redis 服务器。前者称为 **主节点**(master)，后者称为**从节点**(slave)。且数据的复制是 **单向** 的，只能由主节点到从节点。Redis 主从复制支持 **主从同步** 和 **从从同步** 两种，后者是 Redis 后续版本新增的功能，以减轻主节点的同步负担。

> 主从复制主要的作用?

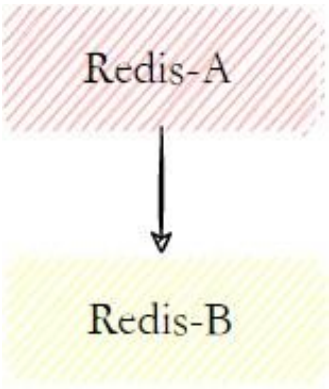
- **数据冗余：** 主从复制实现了数据的热备份，是持久化之外的一种数据冗余方式。
- **故障恢复：** 当主节点出现问题时，可以由从节点提供服务，实现快速的故障恢复 *(*实际上是一种服务的冗余*)*。
- **负载均衡：** 在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务 （即写 *Redis* 数据时应用连接主节点，读 *Redis* 数据时应用连接从节点），分担服务器负载。尤其是在写少读多的场景下，通过多个从节点分担读负载，可以大大提高 Redis 服务器的并发量。
- **高可用基石：** 除了上述作用以外，主从复制还是哨兵和集群能够实施的 **基础**，因此说主从复制是 Redis 高可用的基础。

14. Redis主从有几种常见的拓扑结构？

Redis的复制拓扑结构可以支持单层或多层复制关系，根据拓扑复杂性可以分为以下三种：一主一从、一主多从、树状主从结构。

1. 一主一从结构

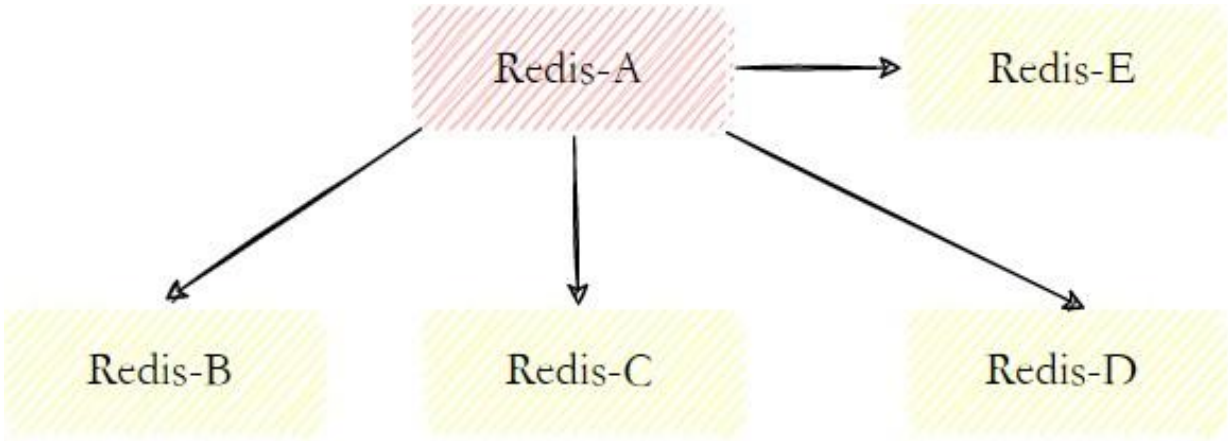
一主一从结构是最简单的复制拓扑结构，用于主节点出现宕机时从节点提供故障转移支持。



一主一从结构

2. 一主多从结构

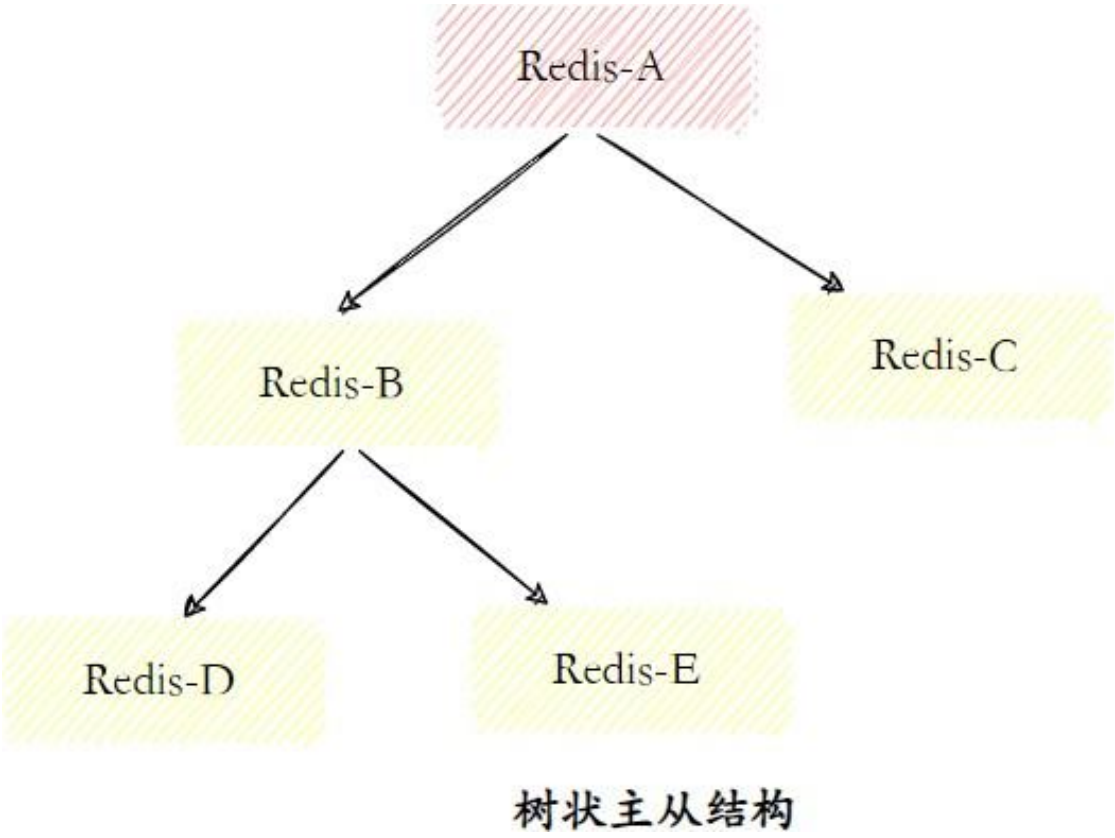
一主多从结构（又称为星形拓扑结构）使得应用端可以利用多个从节点实现读写分离（见图6-5）。对于读占比较大的场景，可以把读命令发送到从节点来分担主节点压力。



一主多从（星形）结构

3. 树状主从结构

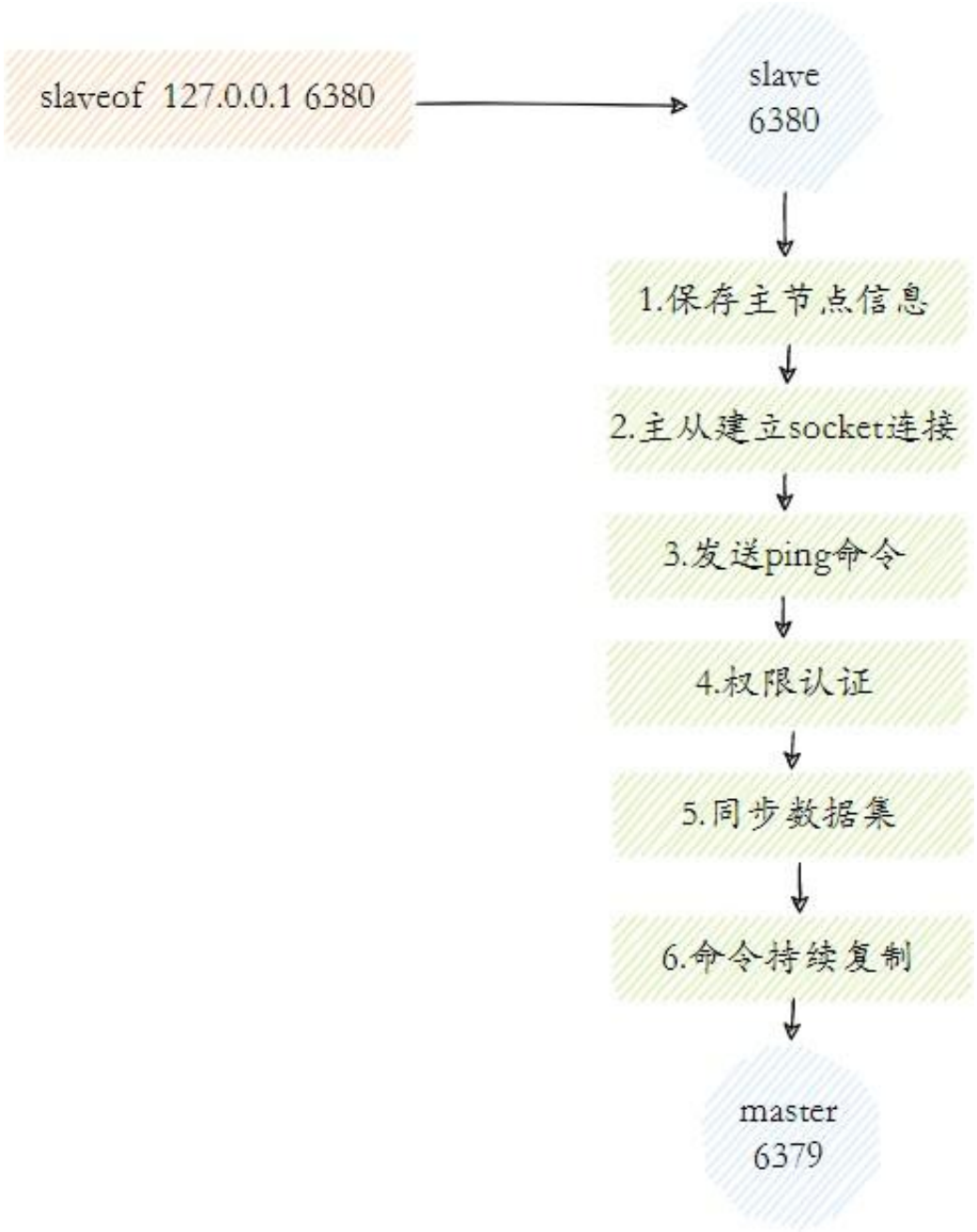
树状主从结构（又称为树状拓扑结构）使得从节点不但可以复制主节点数据，同时可以作为其他从节点的主节点继续向下层复制。通过引入复制中间层，可以有效降低主节点负载和需要传送给从节点的数据量。



树状主从结构

15. Redis的主从复制原理了解吗？

Redis主从复制的工作流程大概可以分为如下几步：



1. 保存主节点（master）信息
   这一步只是保存主节点信息，保存主节点的ip和port。
2. 主从建立连接
   从节点（slave）发现新的主节点后，会尝试和主节点建立网络连接。
3. 发送ping命令
   连接建立成功后从节点发送ping请求进行首次通信，主要是检测主从之间网络套接字是否可用、主节点当前是否可接受处理命令。
4. 权限验证
   如果主节点要求密码验证，从节点必须正确的密码才能通过验证。
5. 同步数据集
   主从复制连接正常通信后，主节点会把持有的数据全部发送给从节点。
6. 命令持续复制

接下来主节点会持续地把写命令发送给从节点，保证主从数据一致性。

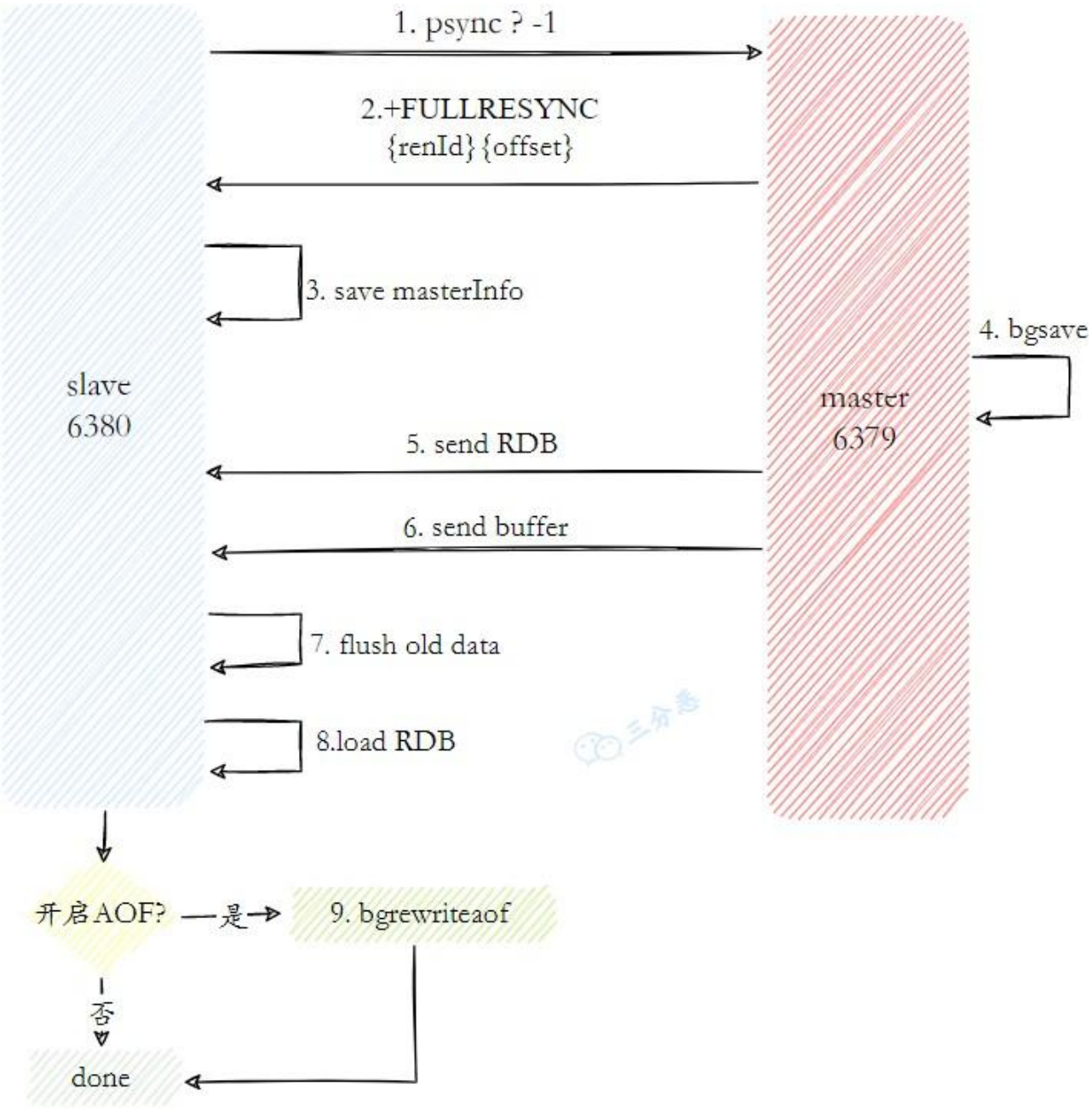
16. 说说主从数据同步的方式？

Redis在2.8及以上版本使用psync命令完成主从数据同步，同步过程分为：全量复制和部分复制。

全量复制

一般用于初次复制场景，Redis早期支持的复制功能只有全量复制，它会把主节点全部数据一次性发送给从节点，当数据量较大时，会对主从节点和网络造成很大的开销。

全量复制的完整运行流程如下：

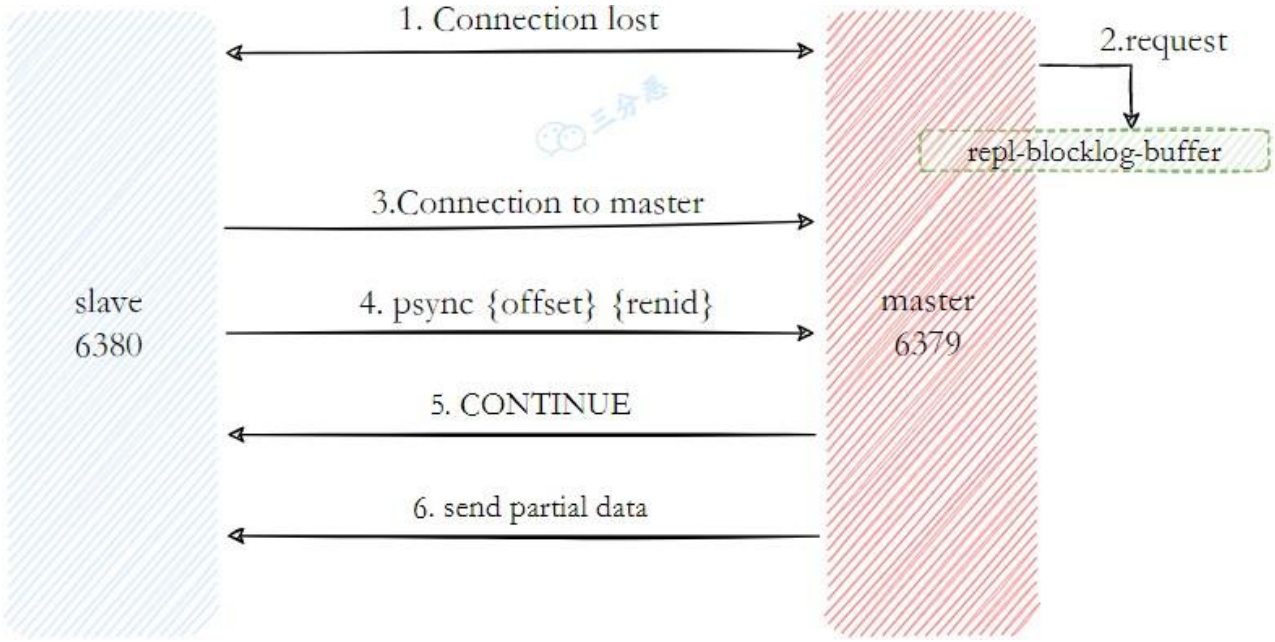


1. 发送psync命令进行数据同步，由于是第一次进行复制，从节点没有复制偏移量和主节点的运行ID，所以发送psync-1。
2. 主节点根据psync-1解析出当前为全量复制，回复+FULLRESYNC响应。
3. 从节点接收主节点的响应数据保存运行ID和偏移量offset
4. 主节点执行bgsave保存RDB文件到本地
5. 主节点发送RDB文件给从节点，从节点把接收的RDB文件保存在本地并直接作为从节点的数据文件
6. 对于从节点开始接收RDB快照到接收完成期间，主节点仍然响应读写命令，因此主节点会把这期间写命令数据保存在复制客户端缓冲区内，当从节点加载完RDB文件后，主节点再把缓冲区内的数据发送给从节点，保证主从之间数据一致性。
7. 从节点接收完主节点传送来的全部数据后会清空自身旧数据
8. 从节点清空数据后开始加载RDB文件
9. 从节点成功加载完RDB后，如果当前节点开启了AOF持久化功能， 它会立刻做bgrewriteaof操作，为了保证

全量复制后AOF持久化文件立刻可用。

部分复制

部分复制主要是Redis针对全量复制的过高开销做出的一种优化措施， 使用psync{runId}{offset}命令实现。当从 节点（slave）正在复制主节点 （master）时，如果出现网络闪断或者命令丢失等异常情况时，从节点会向 主节点要求补发丢失的命令数据，如果主节点的复制积压缓冲区内存在这部分数据则直接发送给从节点，这样就可以保持主从节点复制的一致性。



1. 当主从节点之间网络出现中断时，如果超过repl-timeout时间，主节点会认为从节点故障并中断复制连接
2. 主从连接中断期间主节点依然响应命令，但因复制连接中断命令无法发送给从节点，不过主节点内部存在的复制积压缓冲区，依然可以保存最近一段时间的写命令数据，默认最大缓存1MB。
3. 当主从节点网络恢复后，从节点会再次连上主节点
4. 当主从连接恢复后，由于从节点之前保存了自身已复制的偏移量和主节点的运行ID。因此会把它们当作psync参数发送给主节点，要求进行部分复制操作。
5. 主节点接到psync命令后首先核对参数runId是否与自身一致，如果一 致，说明之前复制的是当前主节点；之后根据参数offset在自身复制积压缓冲区查找，如果偏移量之后的数据存在缓冲区中，则对从节点发送+CONTINUE响应，表示可以进行部分复制。
6. 主节点根据偏移量把复制积压缓冲区里的数据发送给从节点，保证主从复制进入正常状态。

17.主从复制存在哪些问题呢？

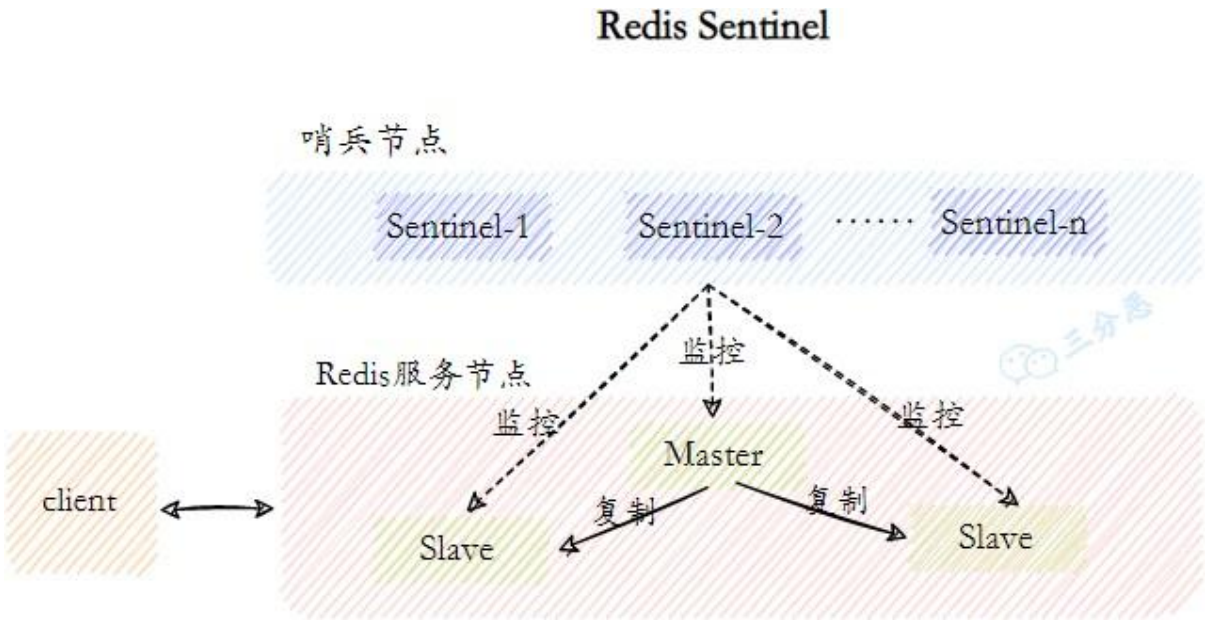
主从复制虽好，但也存在一些问题：

- 一旦主节点出现故障，需要手动将一个从节点晋升为主节点，同时需要修改应用方的主节点地址，还需要命令其他从节点去复制新的主节点，整个过程都需要人工干预。
- 主节点的写能力受到单机的限制。 主
- 节点的存储能力受到单机的限制。

第一个问题是Redis的高可用问题，第二、三个问题属于Redis的分布式问题。

18.Redis Sentinel（哨兵）了解吗？

主从复制存在一个问题，没法完成自动故障转移。所以我们需要一个方案来完成自动故障转移，它就是Redis Sentinel（哨兵）。



Redis Sentinel ，它由两部分组成，哨兵节点和数据节点：

- **哨兵节点：** 哨兵系统由一个或多个哨兵节点组成，哨兵节点是特殊的 Redis 节点，不存储数据，对数据节点进行监控。
- **数据节点：** 主节点和从节点都是数据节点；

在复制的基础上，哨兵实现了 **自动化的故障恢复** 功能，下面是官方对于哨兵功能的描述：

- **监控**（Monitoring）： 哨兵会不断地检查主节点和从节点是否运作正常。
- **自动故障转移**（Automatic failover）： 当 **主节点** 不能正常工作时，哨兵会开始 **自动故障转移操作**，它会将失效主节点的其中一个 **从节点升级为新的主节点**，并让其他从节点改为复制新的主节点。
- **配置提供者**（Configuration provider）： 客户端在初始化时，通过连接哨兵来获得当前 Redis 服务的主节点地址。
- **通知**（Notification）： 哨兵可以将故障转移的结果发送给客户端。

其中，监控和自动故障转移功能，使得哨兵可以及时发现主节点故障并完成转移。而配置提供者和通知功能，则需要在与客户端的交互中才能体现。

19.Redis Sentinel（哨兵）实现原理知道吗？

哨兵模式是通过哨兵节点完成对数据节点的监控、下线、故障转移。

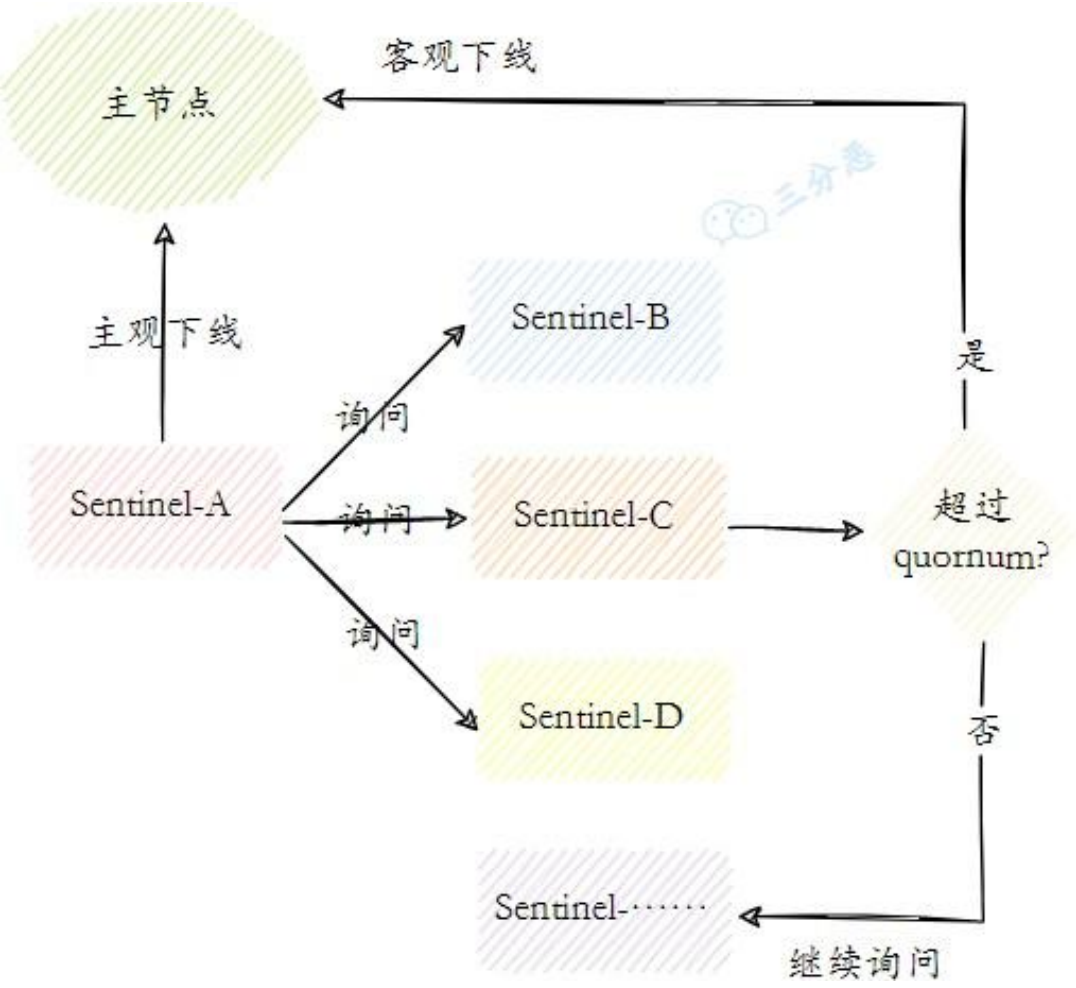
- **定时监控**

Redis Sentinel通过三个定时监控任务完成对各个节点发现和监控：

1. 每隔10秒，每个Sentinel节点会向主节点和从节点发送info命令获取最新的拓扑结构
2. 每隔2秒，每个Sentinel节点会向Redis数据节点的sentinel：hello 频道上发送该Sentinel节点对于主节点的判断以及当前Sentinel节点的信息
3. 每隔1秒，每个Sentinel节点会向主节点、从节点、其余Sentinel节点发送一条ping命令做一次心跳检测，来确认这些节点当前是否可达

- **主观下线和客观下线**

主观下线就是哨兵节点认为某个节点有问题，客观下线就是超过一定数量的哨兵节点认为主节点有问题。



1. 主观下线

每个Sentinel节点会每隔1秒对主节点、从节点、其他Sentinel节点发送ping命令做心跳检测，当这些节点超过down-after-milliseconds没有进行有效回复，Sentinel节点就会对该节点做失败判定，这个行为叫做主观下线。

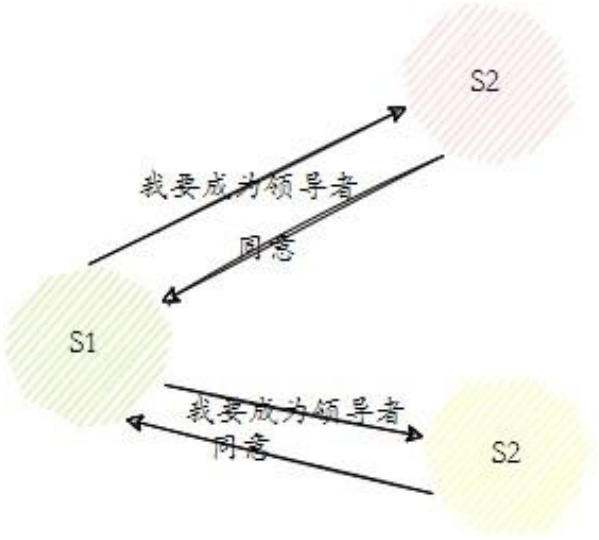
2. 客观下线
   当Sentinel主观下线的节点是主节点时，该Sentinel节点会通过sentinel is- master-down-by-addr命令向其他Sentinel节点询问对主节点的判断，当超过 <quorum>个数，Sentinel节点认为主节点确实有问题，这时该 Sentinel节点会做出客观下线的决定

- **领导者Sentinel节点选举**
  Sentinel节点之间会做一个领导者选举的工作，选出一个Sentinel节点作为领导者进行故障转移的工作。
  Redis使用了Raft算法实现领导者选举。
- **故障转移**
  领导者选举出的Sentinel节点负责故障转移，过程如下：

1. 在从节点列表中选出一个节点作为新的主节点，这一步是相对复杂一些的一步
2. Sentinel领导者节点会对第一步选出来的从节点执行slaveof no one命令让其成为主节点
3. Sentinel领导者节点会向剩余的从节点发送命令，让它们成为新主节点的从节点
4. Sentinel节点集合会将原来的主节点更新为从节点，并保持着对其关注，当其恢复后命令它去复制新的主节点

20. 领导者Sentinel节点选举了解吗？

Redis使用了Raft算法实 现领导者选举，大致流程如下：



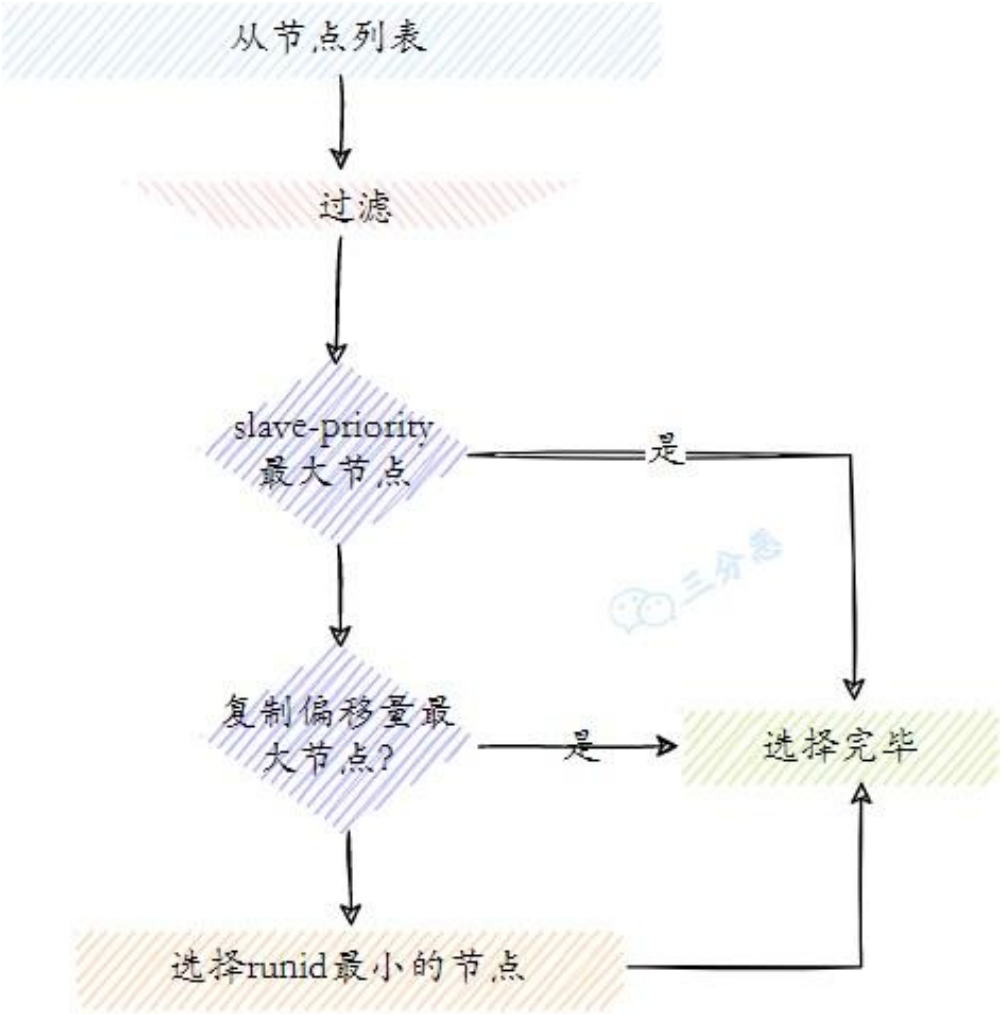
节点	发出的同意	接受的同意
S1		S2，S3
S2	S1	
S3	S1	

1. 每个在线的Sentinel节点都有资格成为领导者，当它确认主节点主观 下线时候，会向其他Sentinel节点发送sentinel is-master-down-by-addr命令， 要求将自己设置为领导者。
2. 收到命令的Sentinel节点，如果没有同意过其他Sentinel节点的sentinel is-master-down-by-addr命令，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大于等于max（quorum， num（sentinels）/2+1），那么它将成为领导者。
4. 如果此过程没有选举出领导者，将进入下一次选举。

21.新的主节点是怎样被挑选出来的？

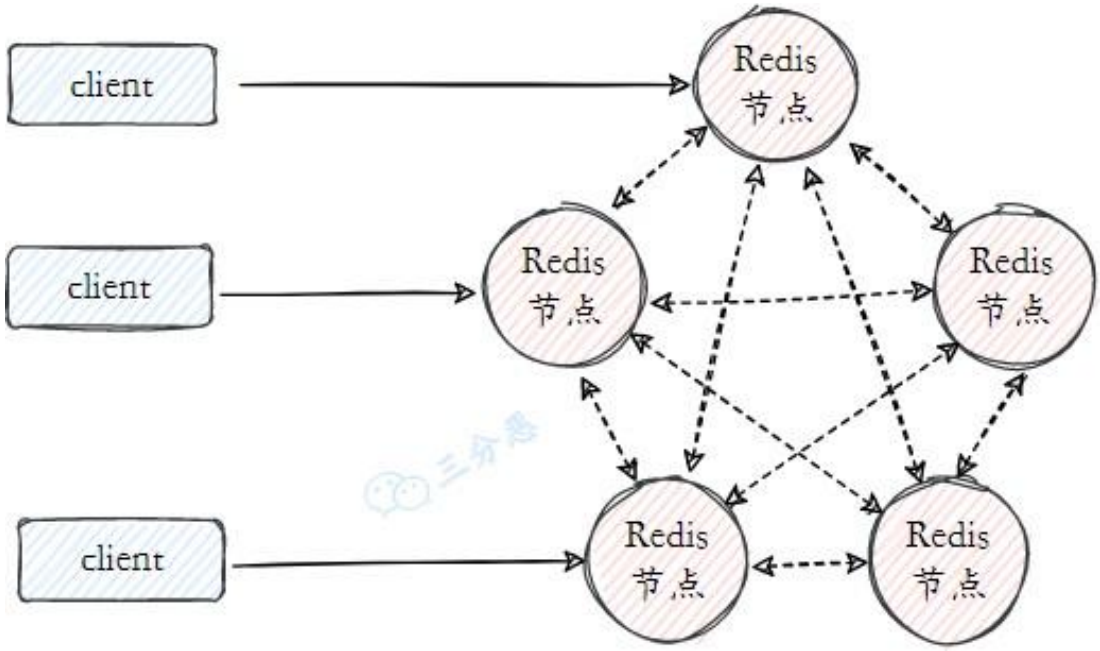
选出新的主节点，大概分为这么几步：



1. 过滤：“不健康”（主观下线、断线）、5秒内没有回复过Sentinel节 点ping响应、与主节点失联超过down-after-milliseconds*10秒。
2. 选择slave-priority（从节点优先级）最高的从节点列表，如果存在则返回，不存在则继续。
3. 选择复制偏移量最大的从节点（复制的最完整），如果存在则返 回，不存在则继续。
4. 选择runid最小的从节点。

22.Redis 集群了解吗？

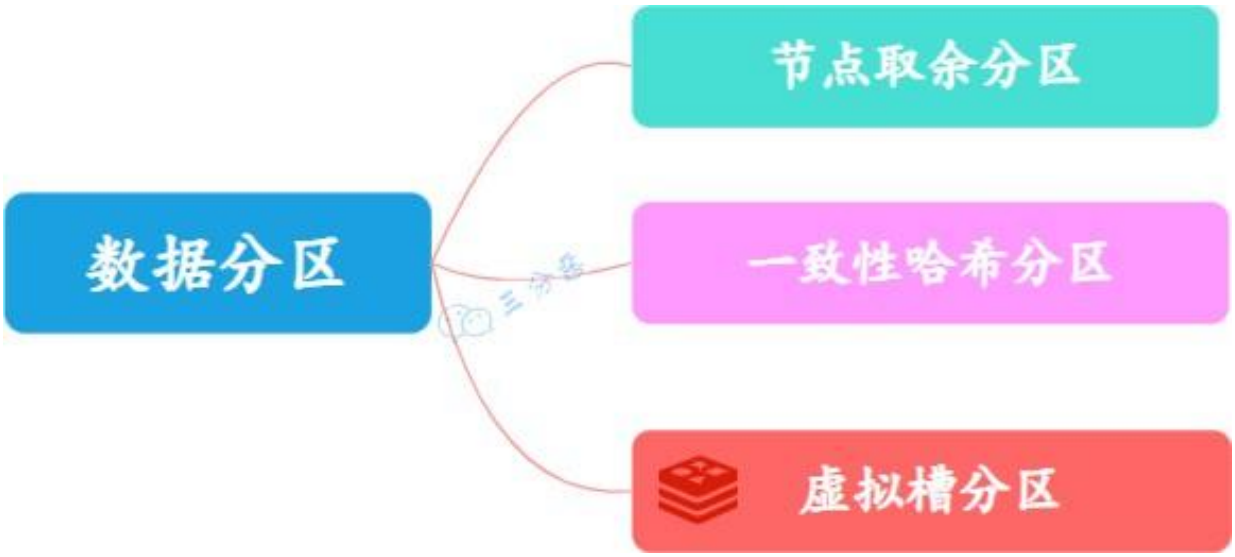
前面说到了主从存在高可用和分布式的问题，哨兵解决了高可用的问题，而集群就是终极方案，一举解决高可用和分布式问题。



1. **数据分区：** 数据分区 *(*或称数据分片*)* 是集群最核心的功能。集群将数据分散到多个节点，一方面 突破了 Redis 单机内存大小的限制，**存储容量大大增加**；**另一方面** 每个主节点都可以对外提供读服务和写服务，**大大提高了集群的响应能力**。
2. **高可用：** 集群支持主从复制和主节点的 **自动故障转移** （与哨兵类似），当任一节点发生故障时，集群仍然可以对外提供服务。

23.集群中数据如何分区？

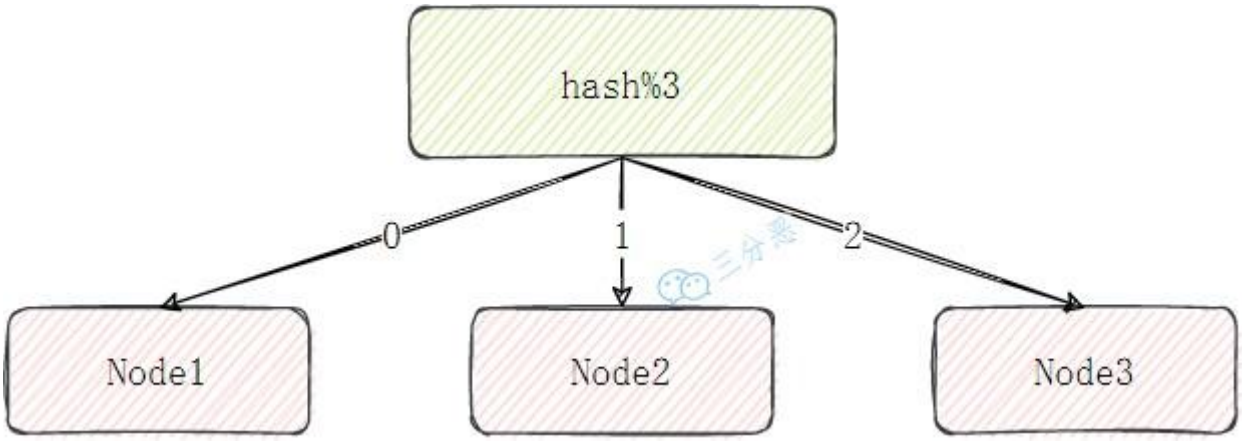
分布式的存储中，要把数据集按照分区规则映射到多个节点，常见的数据分区规则三种：



方案一：节点取余分区

节点取余分区，非常好理解，使用特定的数据，比如Redis的键，或者用户ID之类，对响应的hash值取余：
hash（key）%N，来确定数据映射到哪一个节点上。

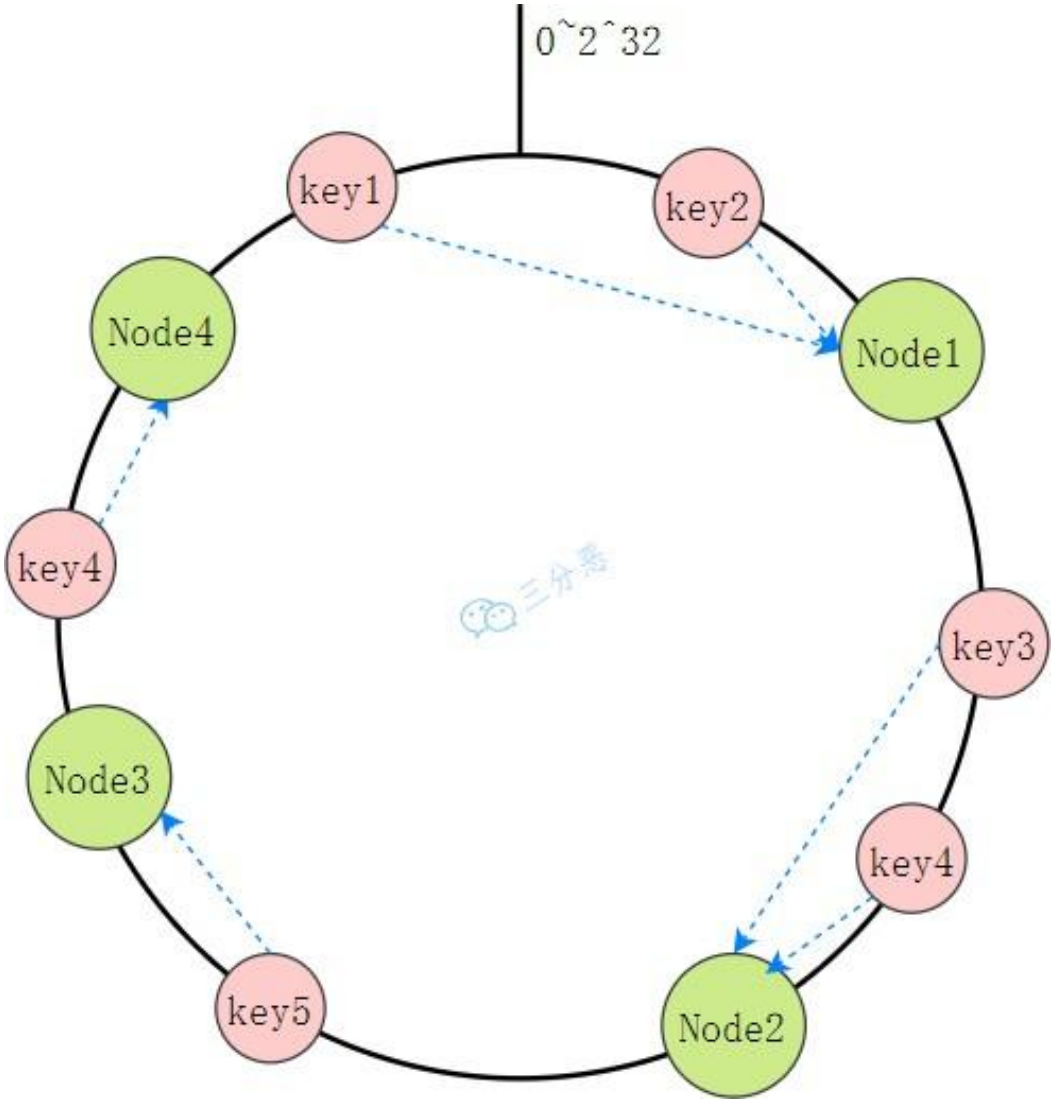
不过该方案最大的问题是，当节点数量变化时，如扩容或收缩节点，数据节点映射关 系需要重新计算，会导致数据的重新迁移。



方案二：一致性哈希分区

将整个 Hash 值空间组织成一个虚拟的圆环，然后将缓存节点的 IP 地址或者主机名做 Hash 取值后，放置在这个圆环上。当我们需要确定某一个 Key 需 要存取到哪个节点上的时候，先对这个 Key 做同样的 Hash 取值，确定在环上的位置，然后按照顺时针方向在环上“行走”，遇到的第一个缓存节点就是要访问的节点。

比如说下面 这张图里面，Key 1 和 Key 2 会落入到 Node 1 中，Key 3、Key 4 会落入到 Node 2 中，Key 5 落入到 Node 3 中，Key 6 落入到 Node 4 中。



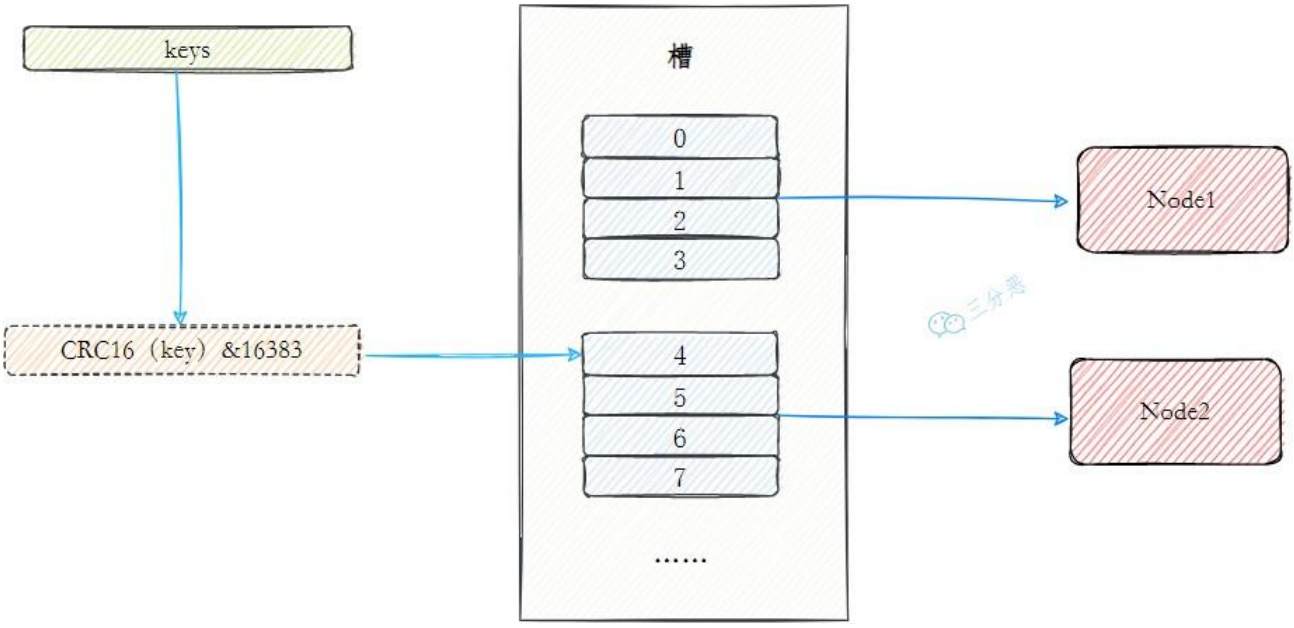
这种方式相比节点取余最大的好处在于加入和删除节点只影响哈希环中 相邻的节点，对其他节点无影响。

但它还是存在问题：

- 缓存节点在圆环上分布不平均，会造成部分缓存节点的压力较大
- 当某个节点故障时，这个节点所要承担的所有访问都会被顺移到另一个节点上，会对后面这个节点造成力。

方案三：虚拟槽分区

这个方案 一致性哈希分区的基础上，引入了**虚拟节点** 的概念。Redis 集群使用的便是该方案，其中的虚拟节点称为**槽**（slot）。槽是介于数据和实际节点之间的虚拟概念，每个实际节点包含一定数量的槽，每个槽包含哈希值在一定范围内的数据。



在使用了槽的一致性哈希分区中，槽是数据管理和迁移的基本单位。槽解耦了数据和实际节点 之间的关系，增加或删除节点对系统的影响很小。仍以上图为例，系统中有 `4` 个实际节点，假设为其分配 `16` 个槽(0-15)；

- 槽 0-3 位于 node1；4-7 位于 node2；以此类推....

如果此时删除 `node2`，只需要将槽 4-7 重新分配即可，例如槽 4-5 分配给 node1，`槽 6` 分配给 node3，`槽 7 分配给 node4`，数据在其他节点的分布仍然较为均衡。

24. 能说说Redis集群的原理吗？

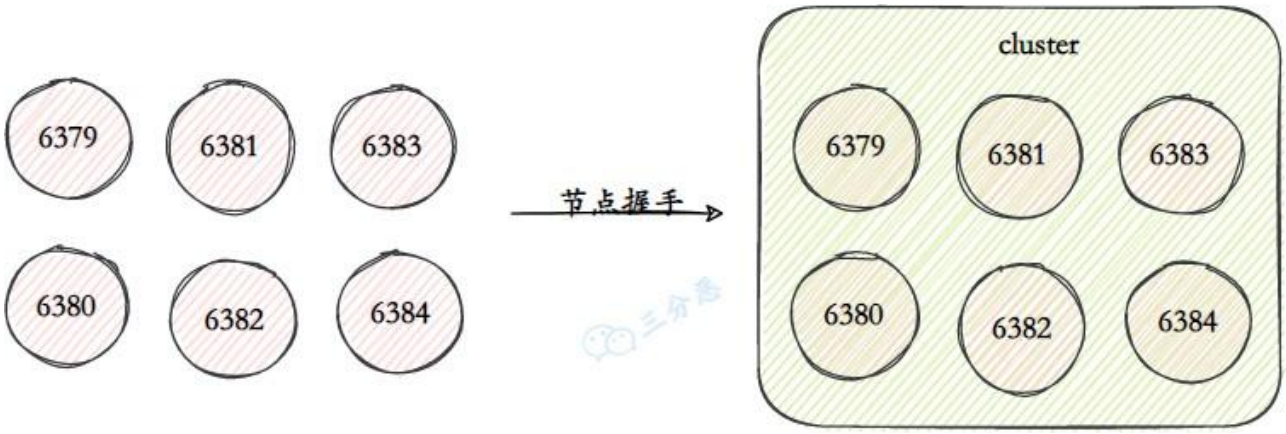
Redis集群通过数据分区来实现数据的分布式存储，通过自动故障转移实现高可用。

集群创建

数据分区是在集群创建的时候完成的。

设置节点

Redis集群一般由多个节点组成，节点数量至少为6个才能保证组成完整高可用的集群。每个节点需要开启配置cluster-enabled yes，让Redis运行在集群模式下。

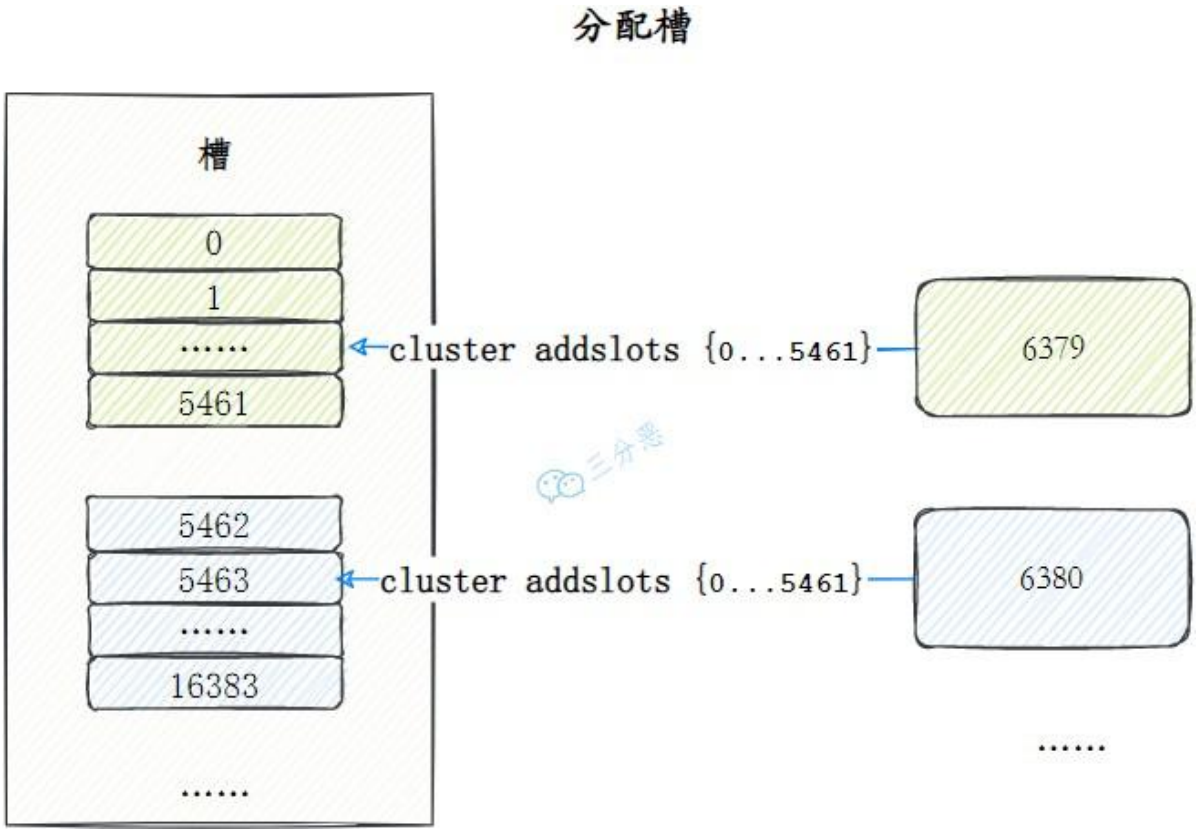


节点握手

节点握手是指一批运行在集群模式下的节点通过Gossip协议彼此通信， 达到感知对方的过程。节点握手是集群彼此通信的第一步，由客户端发起命 令：cluster meet{ip}{port}。完成节点握手之后，一个个的Redis节点就组成了一个多节点的集群。

分配槽（slot）

Redis集群把所有的数据映射到16384个槽中。每个节点对应若干个槽，只有当节点分配了槽，才能响应和这些槽关联的键命令。通过 cluster addslots命令为节点分配槽。

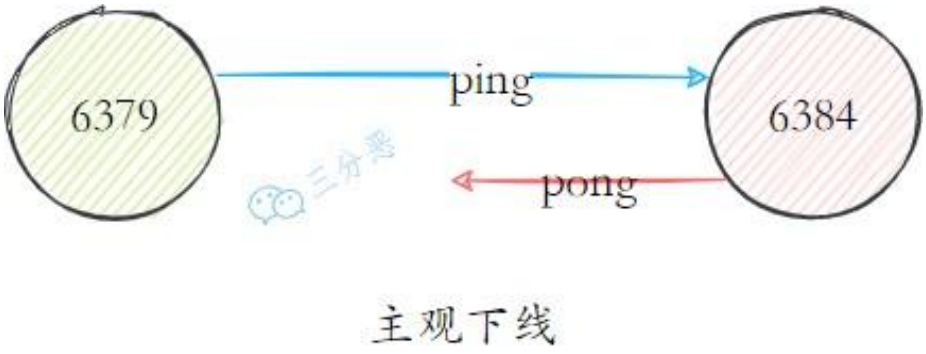


故障转移

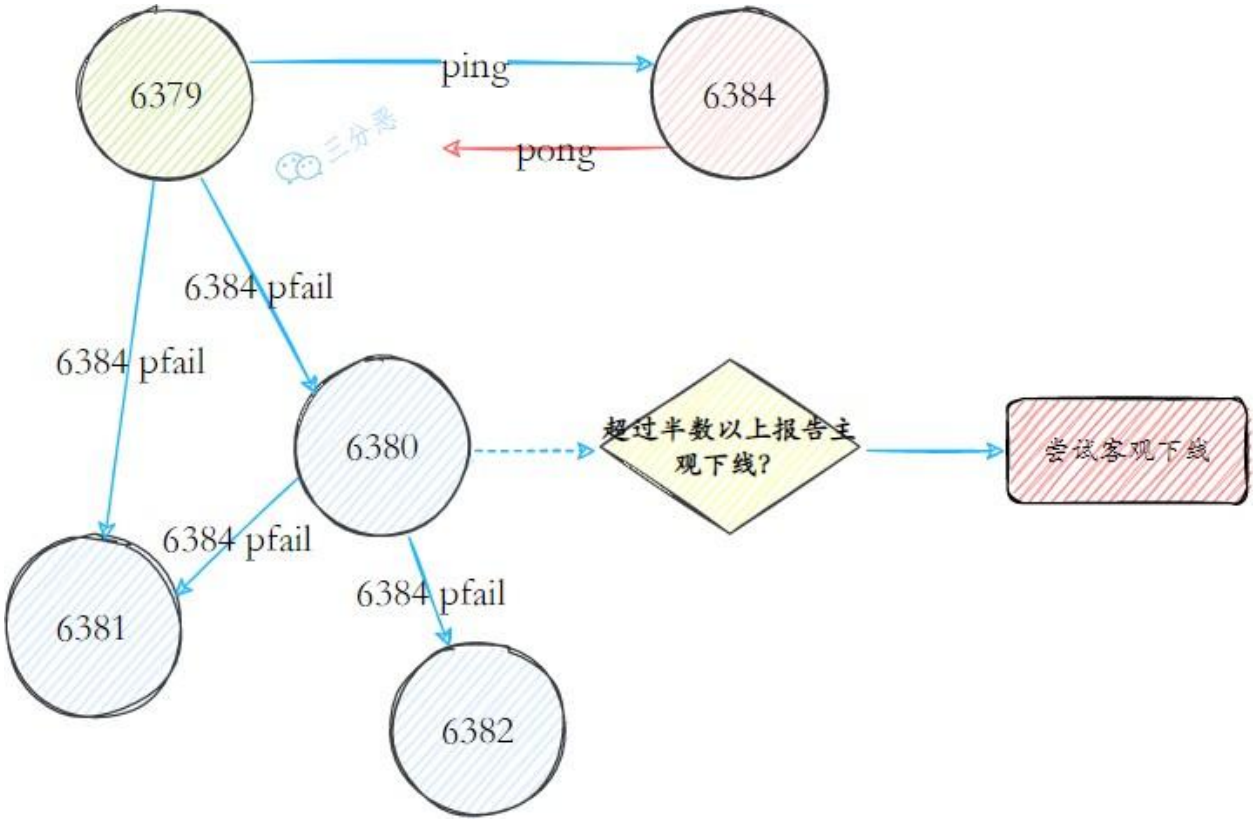
Redis集群的故障转移和哨兵的故障转移类似，但是Redis集群中所有的节点都要承担状态维护的任务。

故障发现

Redis集群内节点通过ping/pong消息实现节点通信，集群中每个节点都会定期向其他节点发送ping消息，接收节点回复pong 消息作为响应。如果在cluster-node-timeout时间内通信一直失败，则发送节 点会认为接收节点存在故障，把接收节点标记为主观下线（pfail）状态。



当某个节点判断另一个节点主观下线后，相应的节点状态会跟随消息在集群内传播。通过Gossip消息传播，集群内节点不断收集到故障节点的下线报告。当 半数以上持有槽的主节点都标记某个节点是主观下线时。触发客观下线流程。

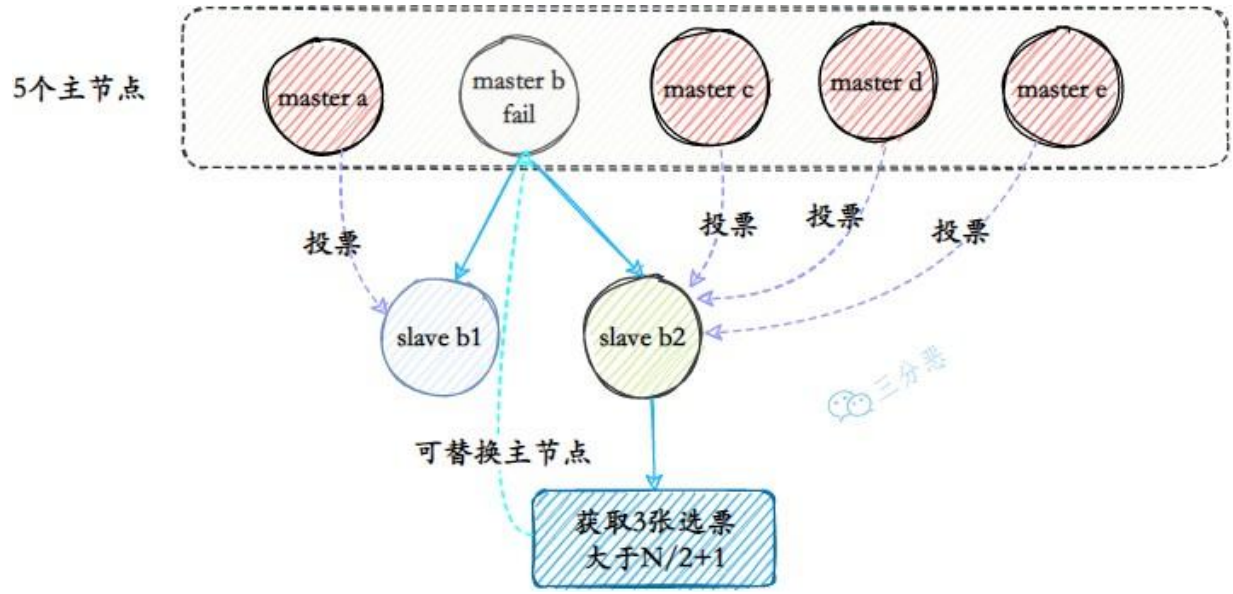


故障恢复

故障节点变为客观下线后，如果下线节点是持有槽的主节点则需要在它 的从节点中选出一个替换它，从而保证集群的高可用。

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。
2. 准备选举时间
   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。
3. 发起选举
   当从节点定时任务检测到达故障选举时间（failover_auth_time）到达后，发起选举流程。
4. 选举投票
   持有槽的主节点处理故障选举消息。投票过程其实是一个领导者选举的过程，如集群内有N个持有槽的主节点代表有N张选票。由于在每个配置纪元内持有槽的主节点只能投票给一个 从节点，因此只能有一个从节点获得N/2+1的选票，保证能够找出唯一的从节点。



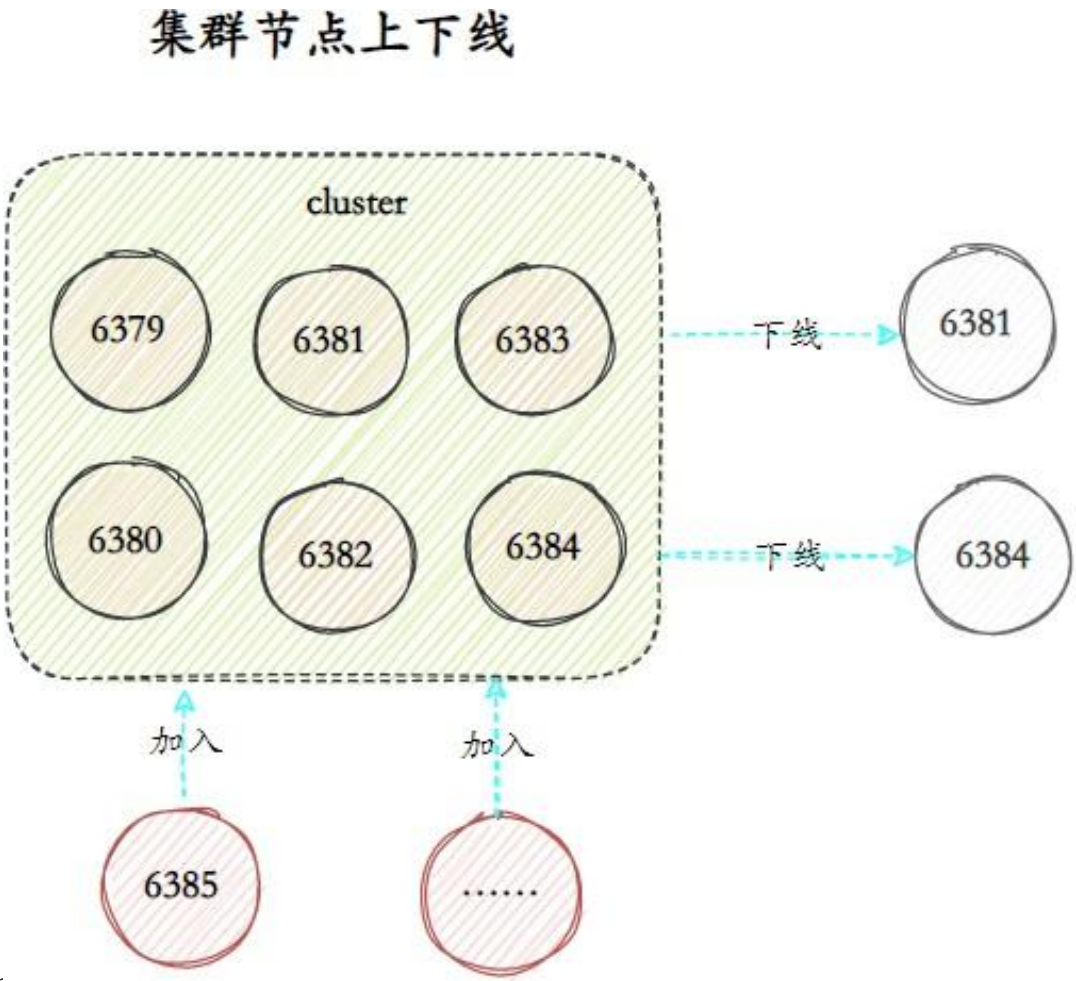
5. 替换主节点
   当从节点收集到足够的选票之后，触发替换主节点操作。

> 部署Redis集群至少需要几个物理节点？

在投票选举的环节，故障主节点也算在投票数内，假设集群内节点规模是3主3从，其中有2 个主节点部署在一台机器上，当这台机器宕机时，由于从节点无法收集到 3/2+1个主节点选票将导致故障转移失败。这个问题也适用于故障发现环节。因此部署集群时所有主节点最少需要部署在3台物理机上才能避免单点问题。

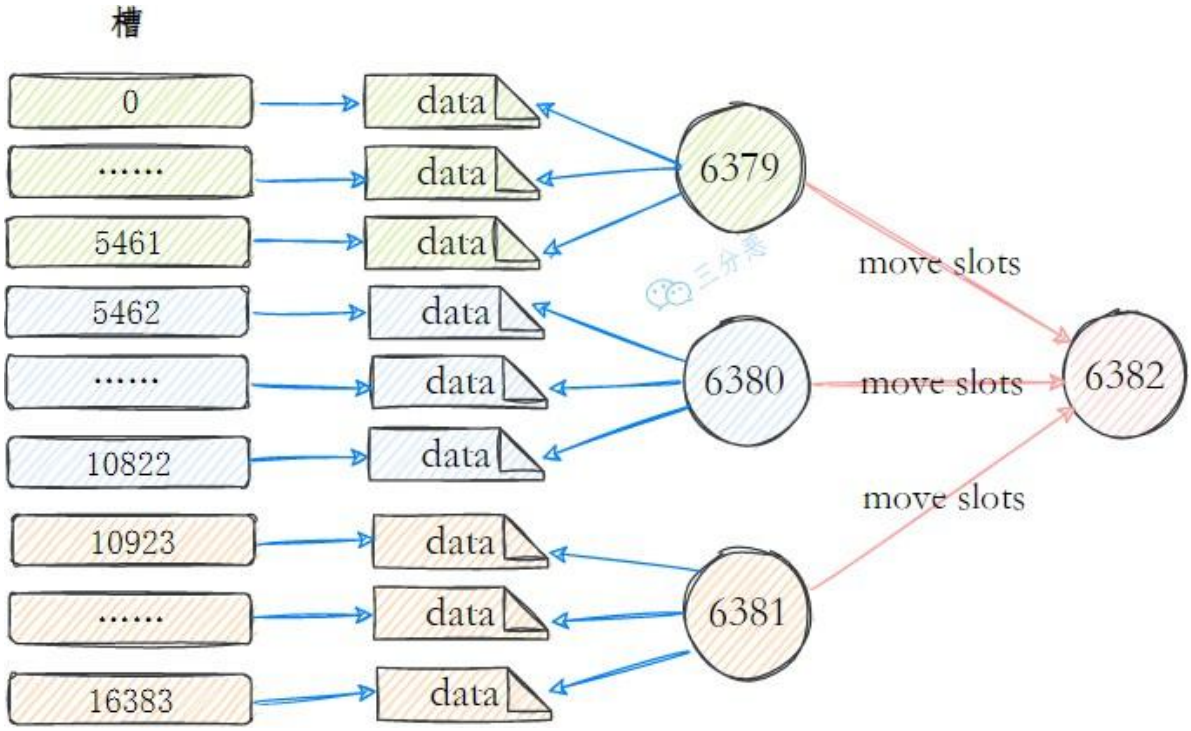
25. 说说集群的伸缩？

Redis集群提供了灵活的节点扩容和收缩方案，可以在不影响集群对外服务的情况下，为集群添加节点进行扩容也可以下线部分节点进行缩容。



其实，集群扩容和缩容的关键点，就在于槽和节点的对应关系，扩容和缩容就是将一部分槽 和 数据 迁移给新节点。

例如下面一个集群，每个节点对应若干个槽，每个槽对应一定的数据，如果希望加入1个节点希望实现集群扩容时，需要通过相关命令把一部分槽和内容迁移给新节点。



缩容也是类似，先把槽和数据迁移到其它节点，再把对应的节点下线。

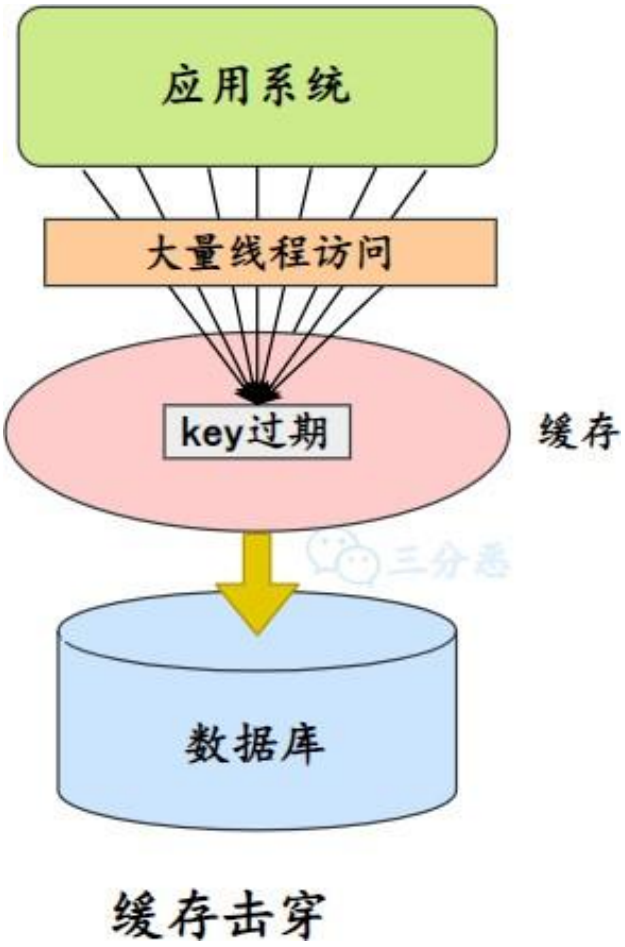
缓存设计

26.什么是缓存击穿、缓存穿透、缓存雪崩？

PS:这是多年黄历的老八股了，一定要理解清楚。

缓存击穿

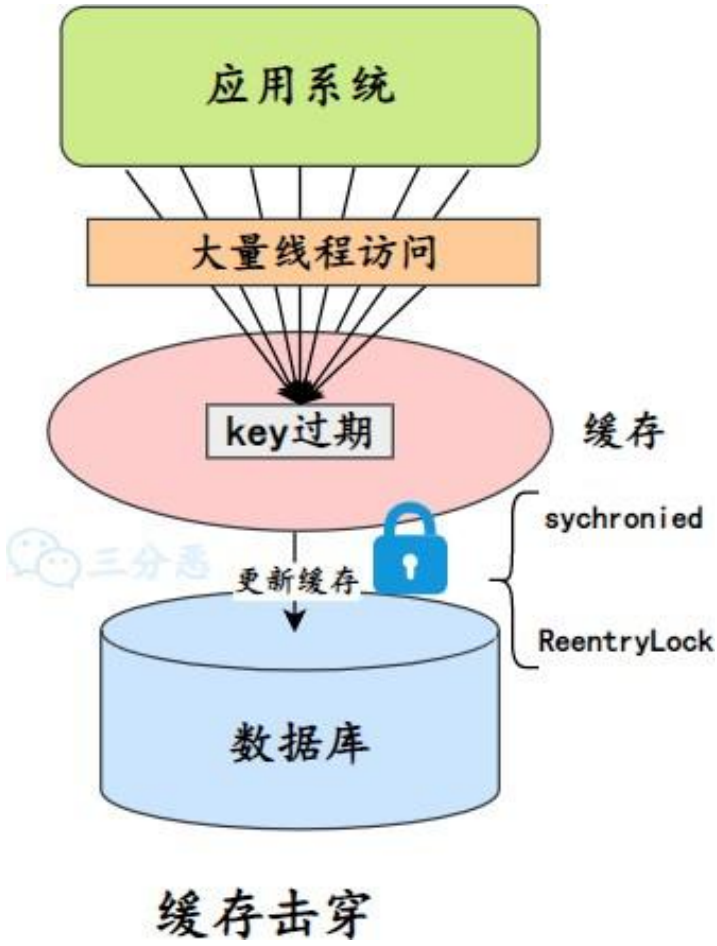
一个并发访问量比较大的key在某个时间过期，导致所有的请求直接打在DB上。



缓存击穿

解决方案：

1. 加锁更新，比如请求查询A，发现缓存中没有，对A这个key加锁，同时去数据库查询数据，写入缓存，再返回给用户，这样后面的请求就可以从缓存中拿到数据了。

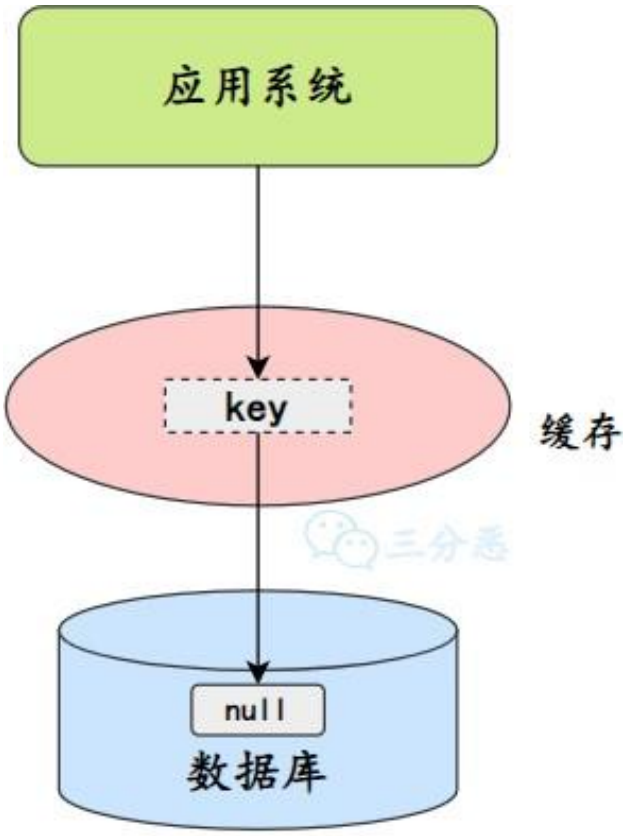


缓存击穿

2. 将过期时间组合写在value中，通过异步的方式不断的刷新过期时间，防止此类现象。

缓存穿透

缓存穿透指的查询缓存和数据库中都不存在的数据，这样每次请求直接打到数据库，就好像缓存不存在一样。



缓存穿透

缓存穿透将导致不存在的数据每次请求都要到存储层去查询，失去了缓存保护后端存储的意义。

缓存穿透可能会使后端存储负载加大，如果发现大量存储层空命中，可能就是出现了缓存穿透问题。缓

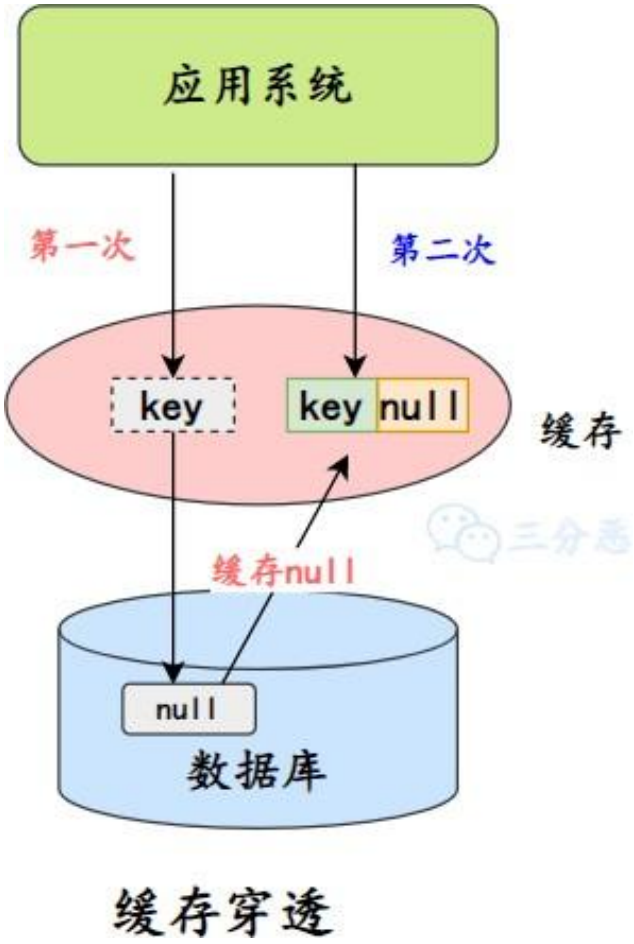
存穿透可能有两种原因：

1. 自身业务代码问题
2. 恶意攻击，爬虫造成空命中

它主要有两种解决办法：

- **缓存空值/默认值**

一种方式是在数据库不命中之后，把一个空对象或者默认值保存到缓存，之后再访问这个数据，就会从缓存中获取，这样就保护了数据库。

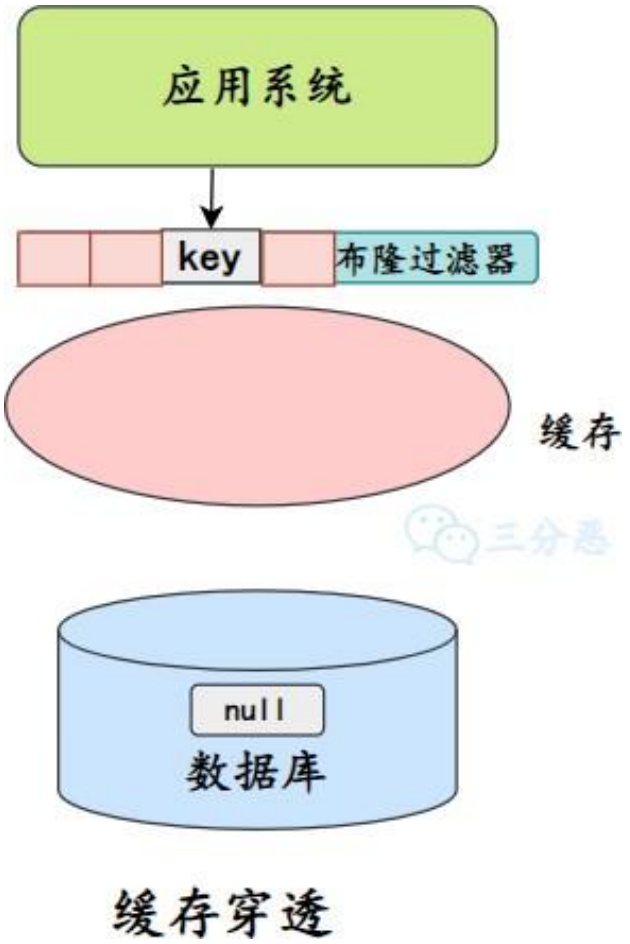


缓存空值有两大问题：

1. 空值做了缓存，意味着缓存层中存了更多的键，需要更多的内存空间（如果是攻击，问题更严重），比较有效的方法是针对这类数据设置一个较短的过期时间，让其自动剔除。
2. 缓存层和存储层的数据会有一段时间窗口的不一致，可能会对业务有一定影响。
   例如过期时间设置为5分钟，如果此时存储层添加了这个数据，那此段时间就会出现缓存层和存储层数据的不一致。
   这时候可以利用消息队列或者其它异步方式清理缓存中的空对象。

- **布隆过滤器**
  除了缓存空对象，我们还可以在存储和缓存之前，加一个布隆过滤器，做一层过滤。

布隆过滤器里会保存数据是否存在，如果判断数据不不能再，就不会访问存储。

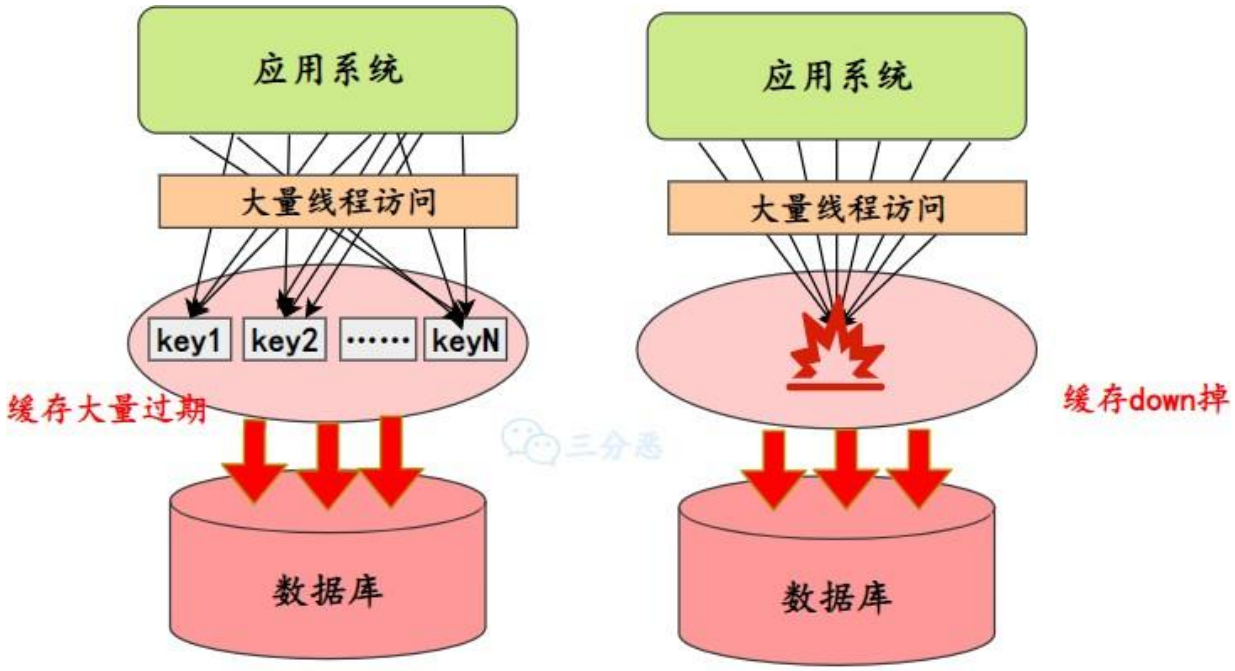


两种解决方案的对比：

解决缓存穿透	适用场景	维护成本
缓存空对象	• 数据命中不高 • 数据频繁实时性高	• 代码维护简单 • 需要较多的缓存空间 • 数据不一致
布隆过滤器	• 数据命中不高 • 数据相对固定实时性低	• 代码维护复杂 • 缓存空间占用少

缓存雪崩

某一时刻发生大规模的缓存失效的情况，例如缓存服务宕机、大量key在同一时间过期，这样的后果就是大量的请求进来直接打到DB上，可能导致整个系统的崩溃，称为雪崩。



缓存雪崩

缓存雪崩是三大缓存问题里最严重的一种，我们来看看怎么预防和处理。

- **提高缓存可用性**

1. 集群部署：通过集群来提升缓存的可用性，可以利用Redis本身的Redis Cluster或者第三方集群方案如Codis等。
2. 多级缓存：设置多级缓存，第一级缓存失效的基础上，访问二级缓存，每一级缓存的失效时间都不同。

- **过期时间**

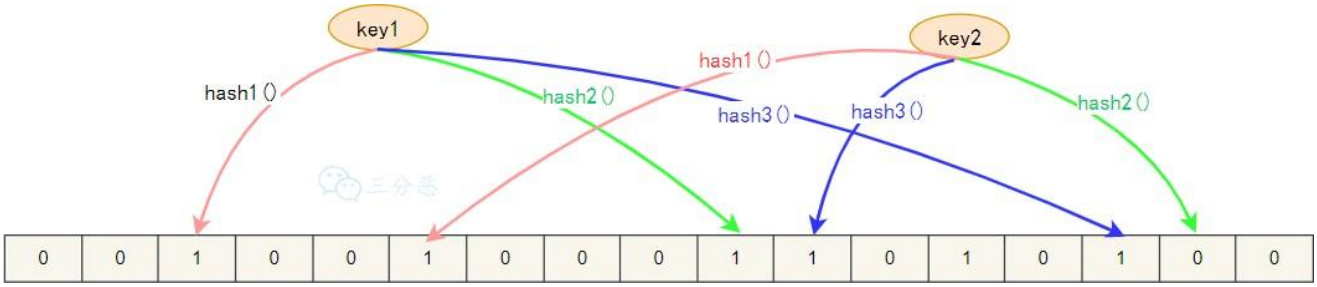
1. 均匀过期：为了避免大量的缓存在同一时间过期，可以把不同的 key 过期时间随机生成，避免过期时间太过集中。
2. 热点数据永不过期。

- **熔断降级**

1. 服务熔断：当缓存服务器宕机或超时响应时，为了防止整个系统出现雪崩，暂时停止业务服务访问缓存系统。
2. 服务降级：当出现大量缓存失效，而且处在高并发高负荷的情况下，在业务系统内部暂时舍弃对一些非核心的接口和数据的请求，而直接返回一个提前准备好的 fallback（退路）错误处理信息。

27.能说说布隆过滤器吗？

布隆过滤器，它是一个连续的数据结构，每个存储位存储都是一个bit，即 0或者 1，来标识数据是否存在。存储数据的时时候，使用K个不同的哈希函数将这个变量映射为bit列表的的K个点，把它们置为1。



我们判断缓存key是否存在，同样，K个哈希函数，映射到bit列表上的K个点，判断是不是1：

- 如果全不是1，那么key不存在；
- 如果都是1，也只是表示key可能存在。布

隆过滤器也有一些缺点：

1. 它在判断元素是否在集合中时是有一定错误几率，因为哈希算法有一定的碰撞的概率。
2. 不支持删除元素。

28. 如何保证缓存和数据库数据的一致性？

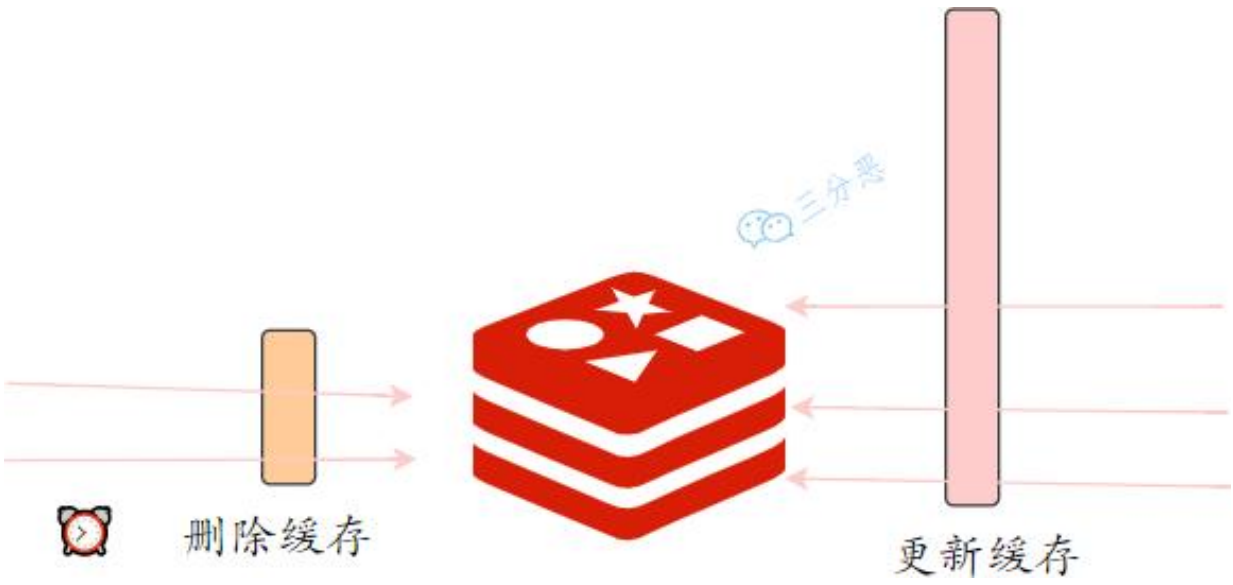
根据CAP理论，在保证可用性和分区容错性的前提下，无法保证一致性，所以缓存和数据库的绝对一致是不可能实现的，只能尽可能保存缓存和数据库的最终一致性。

选择合适的缓存更新策略

1. **删除缓存而不是更新缓存**

当一个线程对缓存的key进行写操作的时候，如果其它线程进来读数据库的时候，读到的就是脏数据，产生了数据不一致问题。

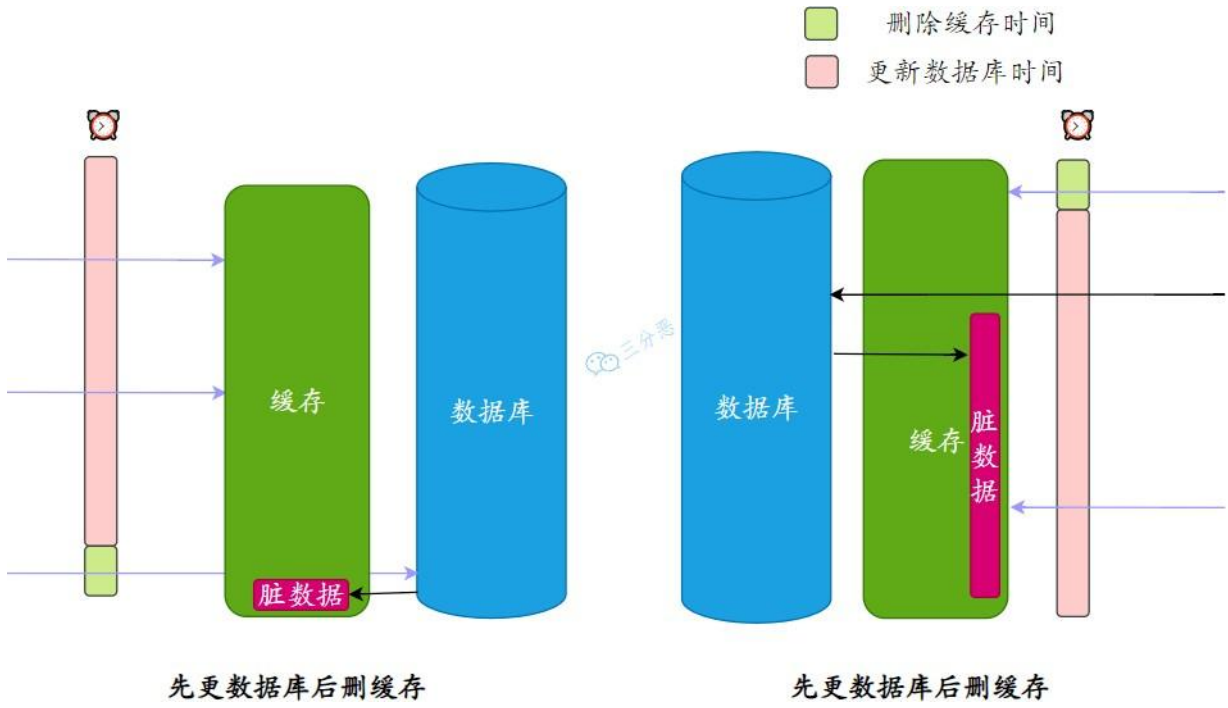
相比较而言，删除缓存的速度比更新缓存的速度快很多，所用时间相对也少很多，读脏数据的概率也小很多。



2. **先更数据，后删缓存**
   先更数据库还是先删缓存？这是一个问题。

更新数据，耗时可能在删除缓存的百倍以上。在缓存中不存在对应的key，数据库又没有完成更新的时候，如果有线程进来读取数据，并写入到缓存，那么在更新成功之后，这个key就是一个脏数据。

毫无疑问，先删缓存，再更数据库，缓存中key不存在的时间的时间更长，有更大的概率会产生脏数据。



目前最流行的缓存读写策略cache-aside-pattern就是采用先更数据库，再删缓存的方式。

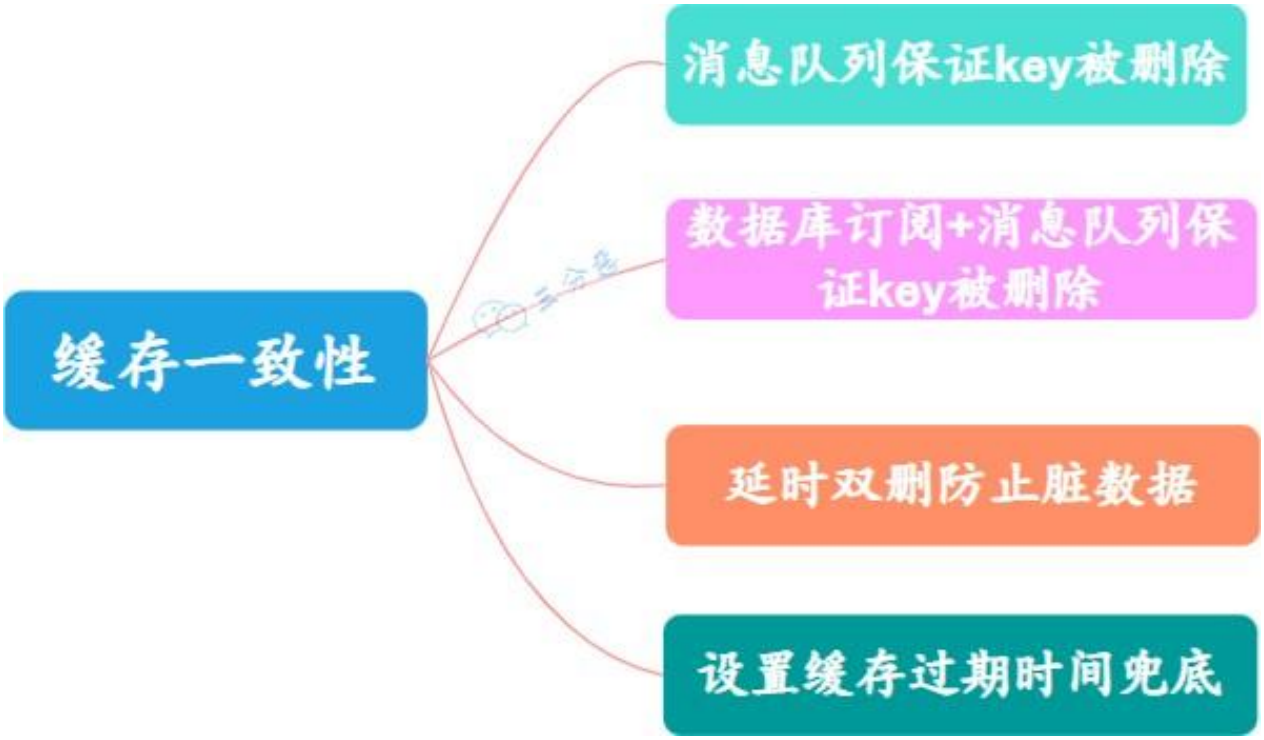
缓存不一致处理

如果不是并发特别高，对缓存依赖性很强，其实一定程序的不一致是可以接受的。但

是如果对一致性要求比较高，那就得想办法保证缓存和数据库中数据一致。

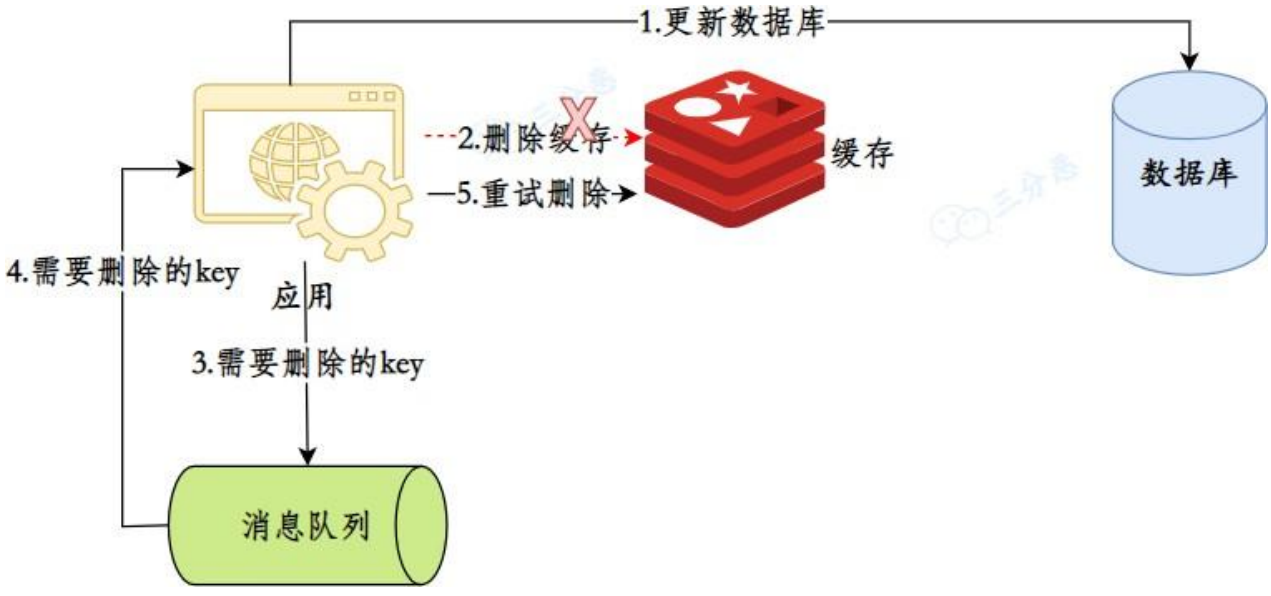
缓存和数据库数据不一致常见的两种原因：

- 缓存key删除失败
- 并发导致写入了脏数据



消息队列保证key被删除

可以引入消息队列，把要删除的key或者删除失败的key丢尽消息队列，利用消息队列的重试机制，重试删除对应的key。

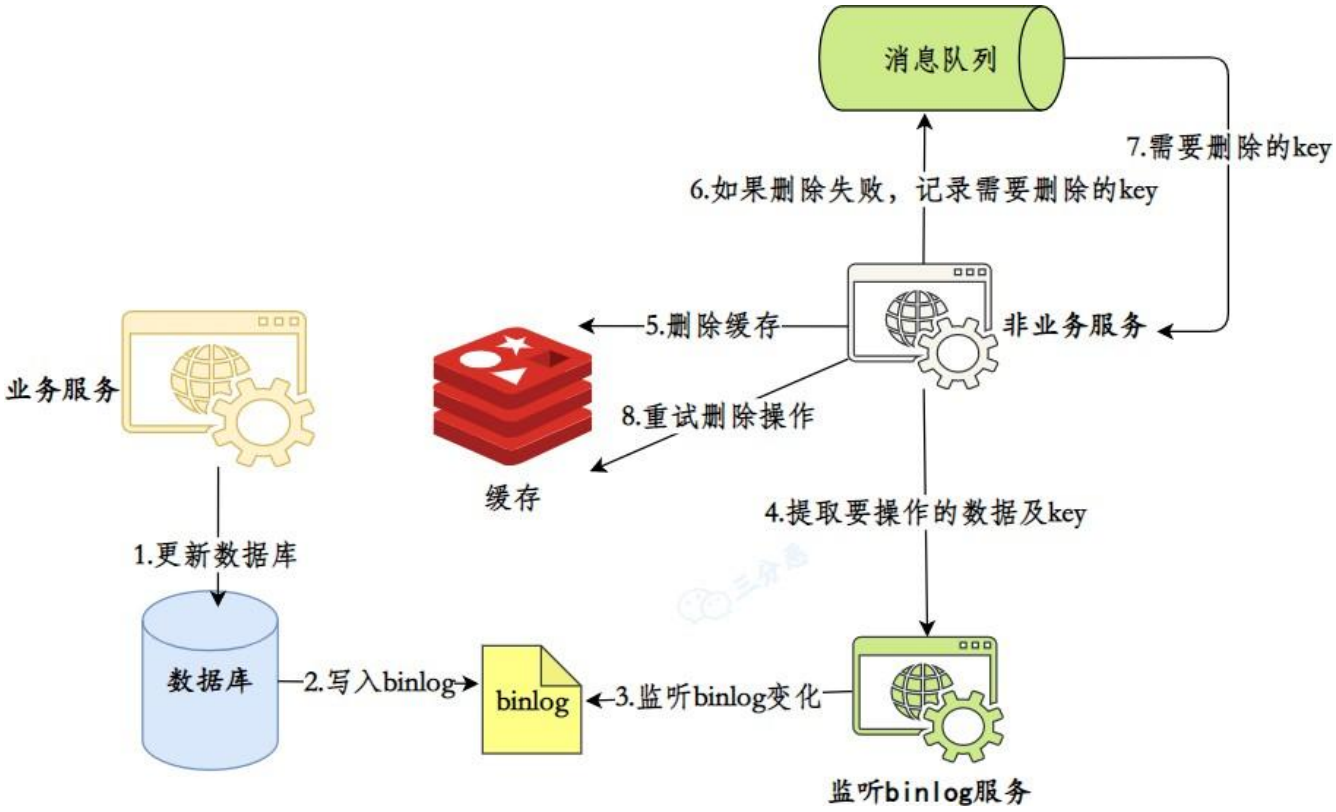


这种方案看起来不错，缺点是对业务代码有一定的侵入性。

数据库订阅+消息队列保证key被删除

可以用一个服务（比如阿里的 canal）去监听数据库的binlog，获取需要操作的数据。

然后用一个公共的服务获取订阅程序传来的信息，进行缓存删除操作。

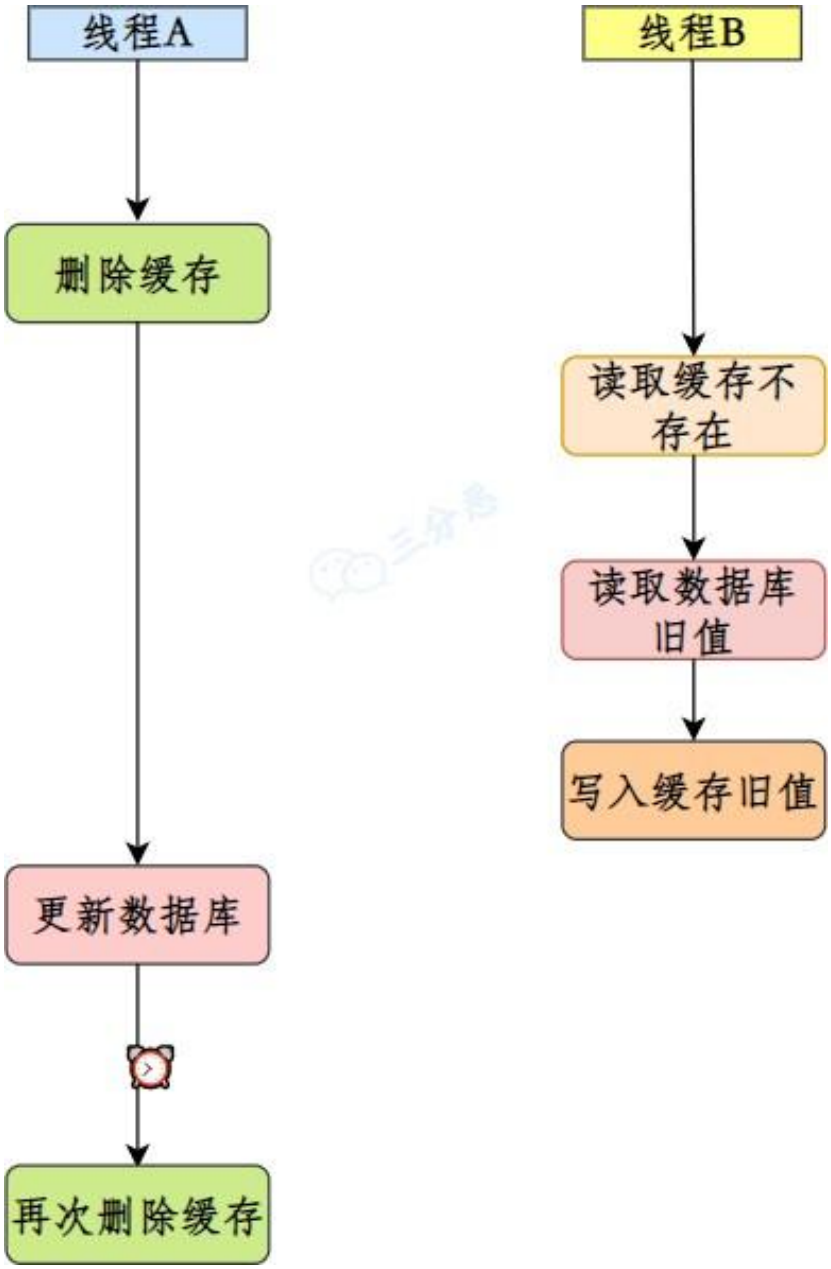


这种方式降低了对业务的侵入，但其实整个系统的复杂度是提升的，适合基建完善的大厂。

延时双删防止脏数据

还有一种情况，是在缓存不存在的时候，写入了脏数据，这种情况在先删缓存，再更数据库的缓存更新策略下发生的比较多，解决方案是延时双删。

简单说，就是在第一次删除缓存之后，过了一段时间之后，再次删除缓存。



这种方式的延时时间设置需要仔细考量和测试。

设置缓存过期时间兜底

这是一个朴素但是有用的办法，给缓存设置一个合理的过期时间，即使发生了缓存数据不一致的问题，它也不会永远不一致下去，缓存过期的时候，自然又会恢复一致。

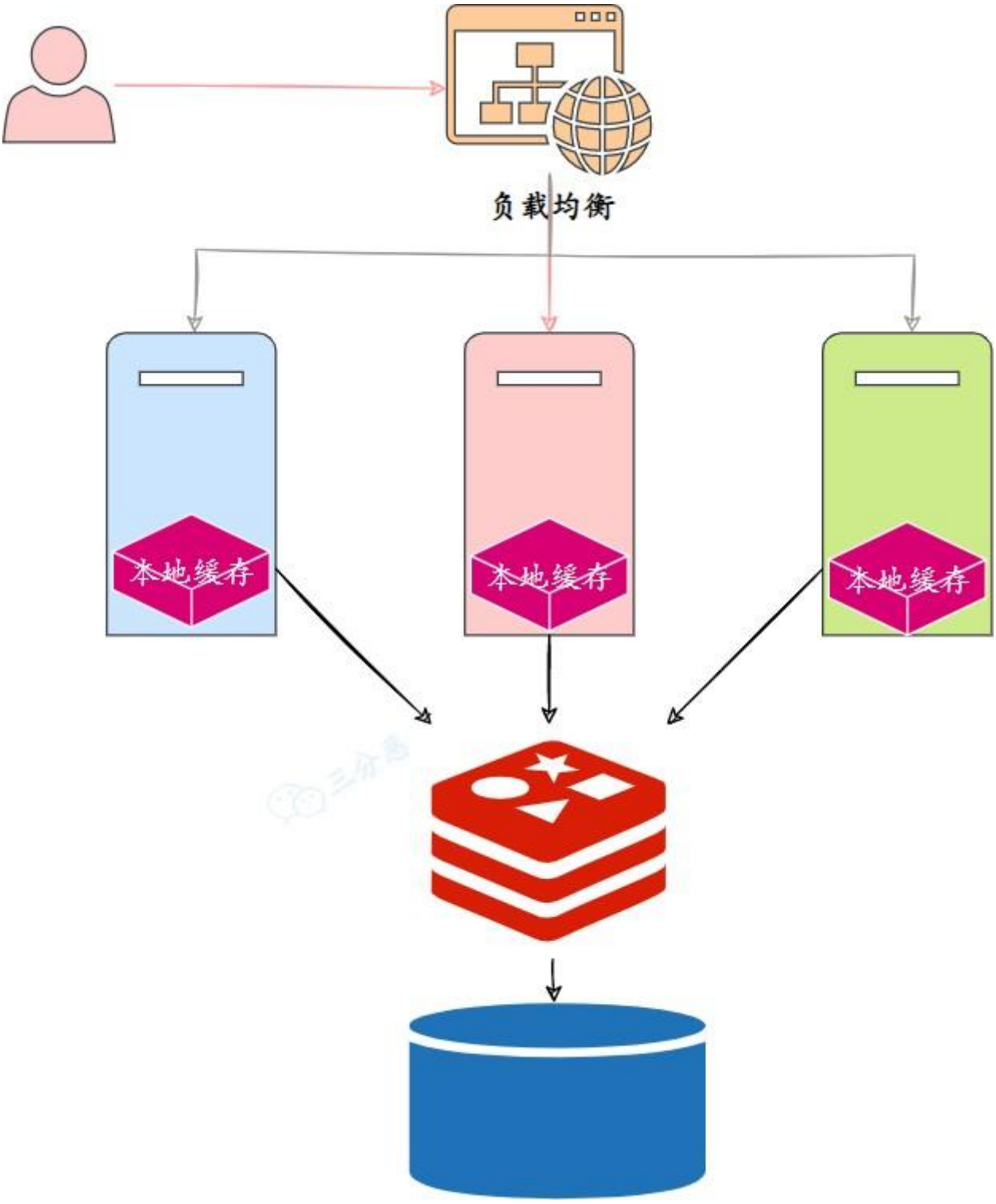
29. 如何保证本地缓存和分布式缓存的一致？

PS:这道题面试很少问，但实际工作中很常见。

在日常的开发中，我们常常采用两级缓存：本地缓存+分布式缓存。

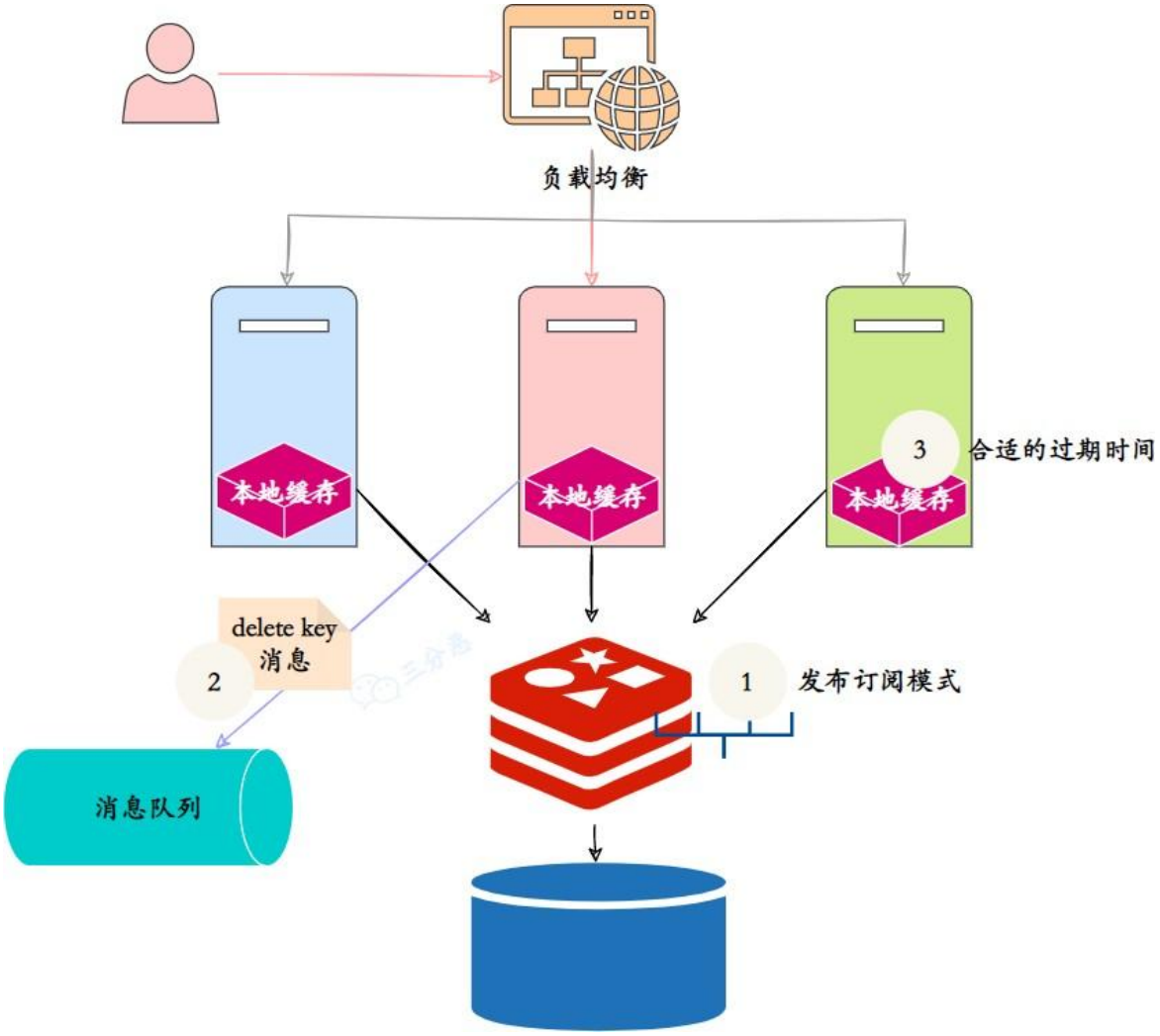
所谓本地缓存，就是对应服务器的内存缓存，比如Caffeine，分布式缓存基本就是采用Redis。

那么问题来了，本地缓存和分布式缓存怎么保持数据一致？



Redis缓存，数据库发生更新，直接删除缓存的key即可，因为对于应用系统而言，它是一种中心化的缓存。

但是本地缓存，它是非中心化的，散落在分布式服务的各个节点上，没法通过客户端的请求删除本地缓存的key，所以得想办法通知集群所有节点，删除对应的本地缓存key。



可以采用消息队列的方式：

1. 采用Redis本身的Pub/Sub机制，分布式集群的所有节点订阅删除本地缓存频道，删除Redis缓存的节点，同事发布删除本地缓存消息，订阅者们订阅到消息后，删除对应的本地key。
   但是Redis的发布订阅不是可靠的，不能保证一定删除成功。
2. 引入专业的消息队列，比如RocketMQ，保证消息的可靠性，但是增加了系统的复杂度。
3. 设置适当的过期时间兜底，本地缓存可以设置相对短一些的过期时间。

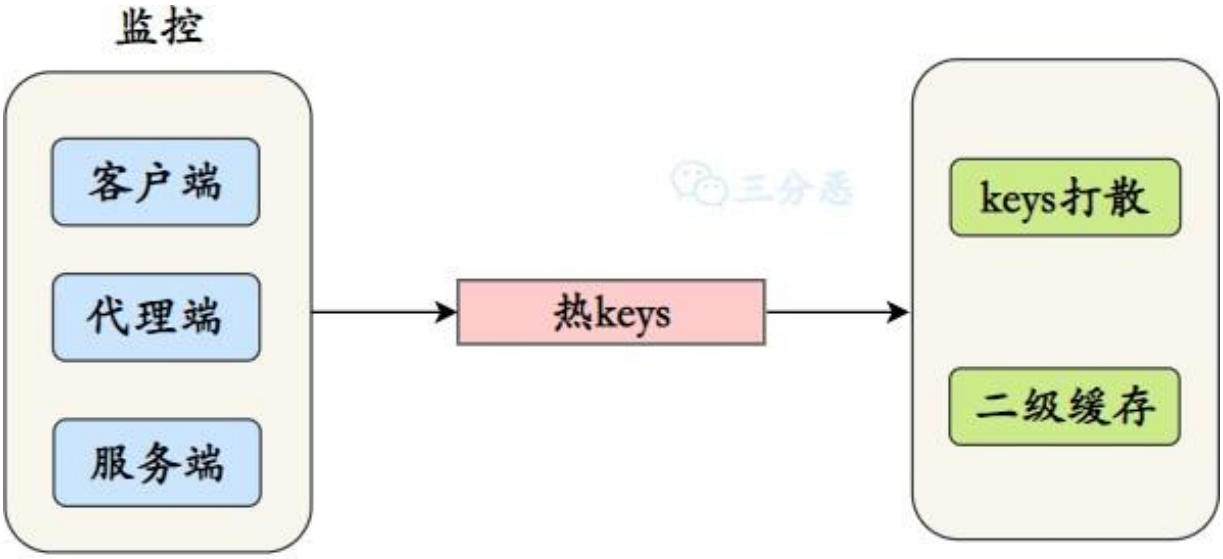
30. 怎么处理热key？

> **什么是热Key？**
> 所谓的热key，就是访问频率比较的key。

比如，热门新闻事件或商品，这类key通常有大流量的访问，对存储这类信息的 Redis来说，是不小的压力。

假如Redis集群部署，热key可能会造成整体流量的不均衡，个别节点出现OPS过大的情况，极端情况下热点key甚至会超过 Redis本身能够承受的OPS。

> **怎么处理热key？**



热key处理

对热key的处理，最关键的是对热点key的监控，可以从这些端来监控热点key:

1. 客户端
   客户端其实是距离key“最近”的地方，因为Redis命令就是从客户端发出的，例如在客户端设置全局字典（key和调用次数），每次调用Redis命令时，使用这个字典进行记录。
2. 代理端
   像Twemproxy、Codis这些基于代理的Redis分布式架构，所有客户端的请求都是通过代理端完成的，可以在代理端进行收集统计。
3. Redis服务端
   使用monitor命令统计热点key是很多开发和运维人员首先想到，monitor命令可以监控到Redis执行的所有命令。

只要监控到了热key，对热key的处理就简单了：

1. 把热key打散到不同的服务器，降低压力
2. 加入二级缓存，提前加载热key数据到内存中，如果redis宕机，走内存查询

31.缓存预热怎么做呢？

所谓缓存预热，就是提前把数据库里的数据刷到缓存里，通常有这些方法：

1、直接写个缓存刷新页面或者接口，上线时手动操作

2、数据量不大，可以在项目启动的时候自动进行加载

3、定时任务刷新缓存.

32. 热点key重建？问题？解决？

开发的时候一般使用“缓存+过期时间”的策略，既可以加速数据读写，又保证数据的定期更新，这种模式基本能够满足绝大部分需求。

但是有两个问题如果同时出现，可能就会出现比较大的问题：

- 当前key是一个热点key（例如一个热门的娱乐新闻），并发量非常大。
- 重建缓存不能在短时间完成，可能是一个复杂计算，例如复杂的 SQL、多次IO、多个依赖等。 在缓存失效的瞬间，有大量线程来重建缓存，造成后端负载加大，甚至可能会让应用崩溃。

> 怎么处理呢？

要解决这个问题也不是很复杂，解决问题的要点在于：

- 减少重建缓存的次数。
- 数据尽可能一致。
- 较少的潜在危险。

所以一般采用如下方式：

1. 互斥锁（mutex key）
   这种方法只允许一个线程重建缓存，其他线程等待重建缓存的线程执行完，重新从缓存获取数据即可。
2. 永远不过期
   “永远不过期”包含两层意思：

- 从缓存层面来看，确实没有设置过期时间，所以不会出现热点key过期后产生的问题，也就是“物理”不过期。
- 从功能层面来看，为每个value设置一个逻辑过期时间，当发现超过逻辑过期时间后，会使用单独的线程去构建缓存。

33. 无底洞问题吗？如何解决？

> 什么是无底洞问题？

2010年，Facebook的Memcache节点已经达到了3000个，承载着TB级别的缓存数据。但开发和运维人员发现了一个问题，为了满足业务要求添加了大量新Memcache节点，但是发现性能不但没有好转反而下降了，当时将这种现象称为缓存的“**无底洞**”现象。

那么为什么会产生这种现象呢?

通常来说添加节点使得Memcache集群 性能应该更强了，但事实并非如此。键值数据库由于通常采用哈希函数将 key 映射到各个节点上，造成key的分布与业务无关，但是由于数据量和访问量的持续增长，造成需要添加大量节点做水平扩容，导致键值分布到更多的 节点上，所以无论是Memcache还是Redis的分布式，批量操作通常需要从不同节点上获取，相比于单机批量操作只涉及一次网络操作，分布式批量操作会涉及多次网络时间。

> 无底洞问题如何优化呢？

先分析一下无底洞问题：

- 客户端一次批量操作会涉及多次网络操作，也就意味着批量操作会随着节点的增多，耗时会不断增大。网
- 络连接数变多，对节点的性能也有一定影响。

常见的优化思路如下：

- 命令本身的优化，例如优化操作语句等。

- 减少网络通信次数。
- 降低接入成本，例如客户端使用长连/连接池、NIO等。

Redis运维

34. Redis报内存不足怎么处理？

Redis 内存不足有这么几种处理方式：

- 修改配置文件 redis.conf 的 maxmemory 参数，增加 Redis 可用内存也可
- 以通过命令set maxmemory动态设置内存上限
- 修改内存淘汰策略，及时释放内存空间
- 使用 Redis 集群模式，进行横向扩容。

35. Redis的过期数据回收策略有哪些？

Redis主要有2种过期数据回收策略：

惰性删除

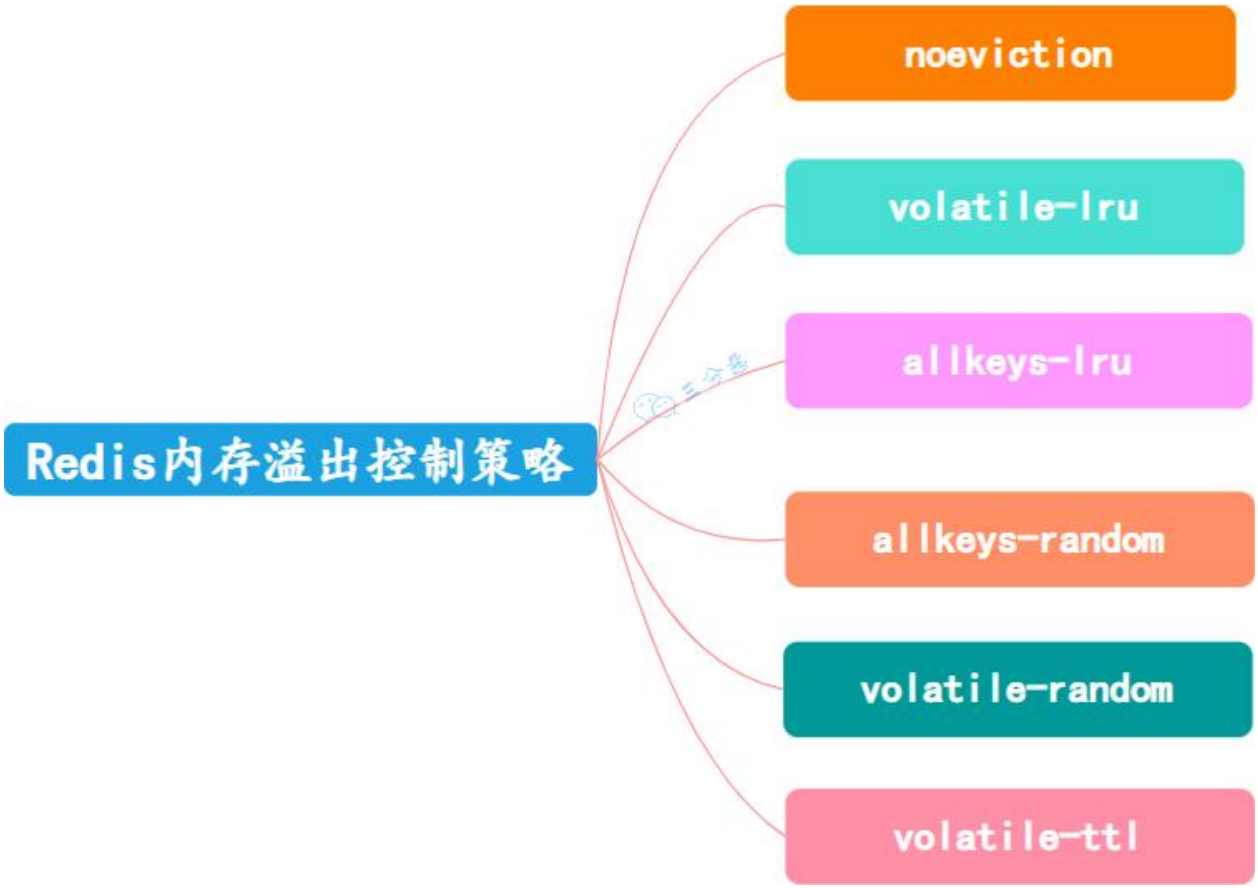
惰性删除指的是当我们查询key的时候才对key进行检测，如果已经达到过期时间，则删除。显然，他有一个缺点就是如果这些过期的key没有被访问，那么他就一直无法被删除，而且一直占用内存。

定期删除

定期删除指的是Redis每隔一段时间对数据库做一次检查，删除里面的过期key。由于不可能对所有key去做轮询来删除，所以Redis会每次随机取一些key去做检查和删除。

36.Redis有哪些内存溢出控制/内存淘汰策略？

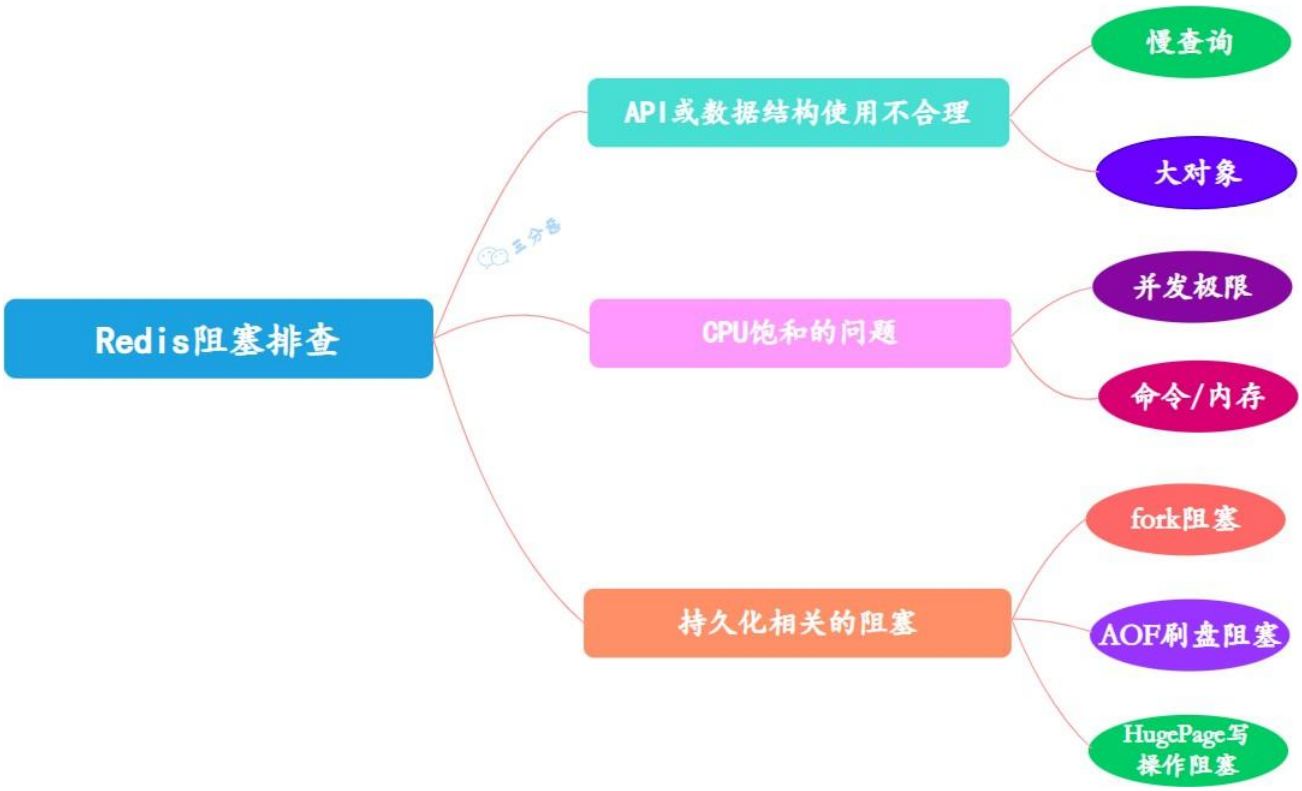
Redis所用内存达到maxmemory上限时会触发相应的溢出控制策略，Redis支持六种策略：



1. noeviction：默认策略，不会删除任何数据，拒绝所有写入操作并返 回客户端错误信息，此 时Redis只响应读操作。
2. volatile-lru：根据LRU算法删除设置了超时属性（expire）的键，直 到腾出足够空间为止。如果没有可删除的键对象，回退到noeviction策略。
3. allkeys-lru：根据LRU算法删除键，不管数据有没有设置超时属性， 直到腾出足够空间为止。
4. allkeys-random：随机删除所有键，直到腾出足够空间为止。
5. volatile-random：随机删除过期键，直到腾出足够空间为止。
6. volatile-ttl：根据键值对象的ttl属性，删除最近将要过期数据。如果 没有，回退到noeviction策略。

37.Redis阻塞？怎么解决？

Redis发生阻塞，可以从以下几个方面排查：



- API**或数据结构使用不合理**

  通常Redis执行命令速度非常快，但是不合理地使用命令，可能会导致执行速度很慢，导致阻塞，对于高并发的场景，应该尽量避免在大对象上执行算法复杂 度超过O（n）的命令。

  对慢查询的处理分为两步：

  1. 发现慢查询： slowlog get{n}命令可以获取最近 的n条慢查询命令；
  2. 发现慢查询后，可以从两个方向去优化慢查询：
     1） 修改为低算法复杂度的命令，如hgetall改为hmget等，禁用keys、sort等命 令
     2） 调整大对象：缩减大对象数据或把大对象拆分为多个小对象，防止一次命令操作过多的数据。

- CPU**饱和的问题**

  单线程的Redis处理命令时只能使用一个CPU。而CPU饱和是指Redis单核CPU使用率跑到接近100%。针对这种情况，处理步骤一般如下：

  1. 判断当前Redis并发量是否已经达到极限，可以使用统计命令redis-cli-h{ip}-p{port}--stat获取当前 Redis使用情况
  2. 如果Redis的请求几万+，那么大概就是Redis的OPS已经到了极限，应该做集群化水品扩展来分摊OPS压力
  3. 如果只有几百几千，那么就得排查命令和内存的使用

- **持久化相关的阻塞**

  对于开启了持久化功能的Redis节点，需要排查是否是持久化导致的阻塞。

  1. fork阻塞
     fork操作发生在RDB和AOF重写时，Redis主线程调用fork操作产生共享 内存的子进程，由子进程完成持

久化文件重写工作。如果fork操作本身耗时过长，必然会导致主线程的阻塞。

2. AOF刷盘阻塞
   当我们开启AOF持久化功能时，文件刷盘的方式一般采用每秒一次，后台线程每秒对AOF文件做fsync操作。当硬盘压力过大时，fsync操作需要等 待，直到写入完成。如果主线程发现距离上一次的fsync成功超过2秒，为了数据安全性它会阻塞直到后台线程执行sync操作完成。
3. HugePage写操作阻塞
   对于开启Transparent HugePages的 操作系统，每次写命令引起的复制内存页单位由4K变为2MB，放大了512 倍，会拖慢写操作的执行时间，导致大量写操作慢查询。

38. 大key问题了解吗？

Redis使用过程中，有时候会出现大key的情况， 比如：单

- 个简单的key存储的value很大，size超过10KB
- hash， set，zset，list 中存储过多的元素（以万为单位）

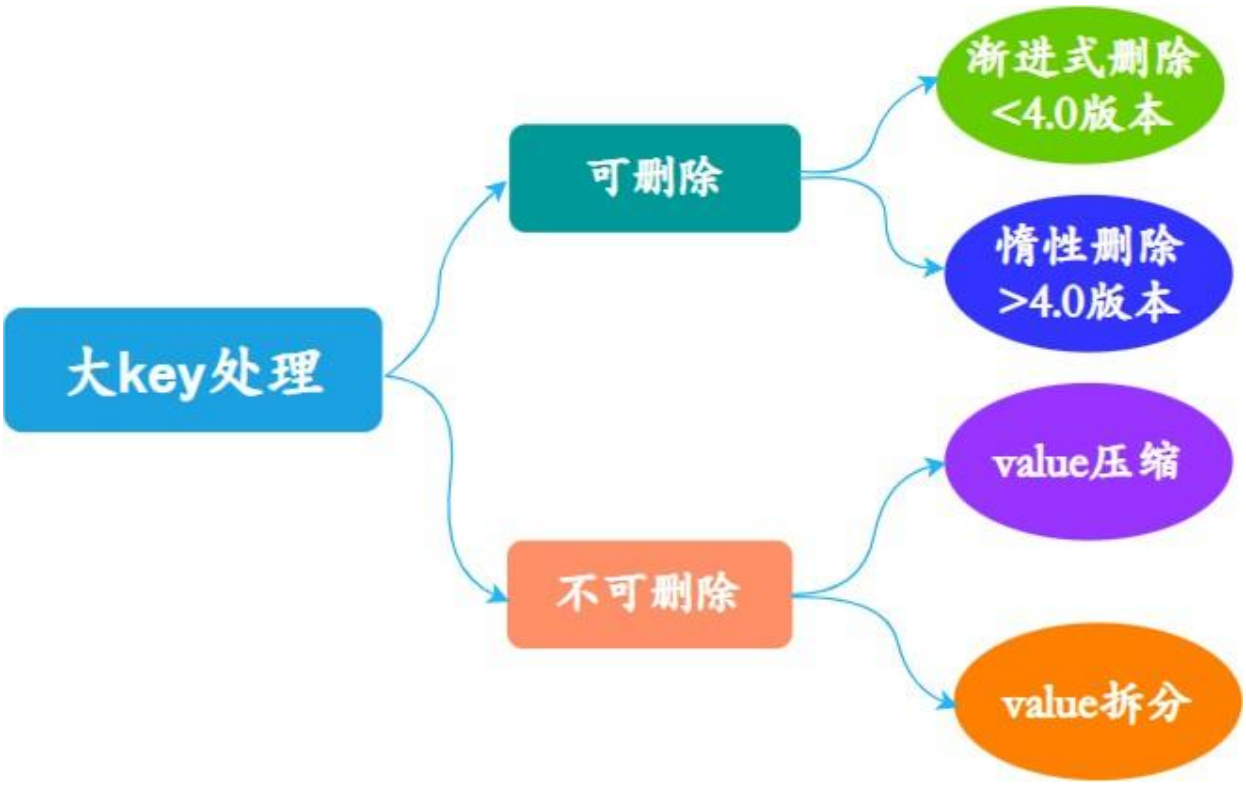
> **大key会造成什么问题呢？**

- 客户端耗时增加，甚至超时
- 对大key进行IO操作时，会严重占用带宽和CPU造
- 成Redis集群中数据倾斜
- 主动删除、被动删等，可能会导致阻塞

> **如何找到大key?**

- bigkeys命令：使用bigkeys命令以遍历的方式分析Redis实例中的所有Key，并返回整体统计信息与每个数据类型中Top1的大Key
- redis-rdb-tools：redis-rdb-tools是由Python写的用来分析Redis的rdb快照文件用的工具，它可以把rdb快照文件生成json文件或者生成报表用来分析Redis的使用详情。

> **如何处理大key?**



- **删除大**key
  - 当Redis版本大于4.0时，可使用UNLINK命令安全地删除大Key，该命令能够以非阻塞的方式，逐步地清理传入的Key。
  - 当Redis版本小于4.0时，避免使用阻塞式命令KEYS，而是建议通过SCAN命令执行增量迭代扫描key，然后判断进行删除。
- **压缩和拆分**key
  - 当vaule是string时，比较难拆分，则使用序列化、压缩算法将key的大小控制在合理范围内，但是序列化和反序列化都会带来更多时间上的消耗。
  - 当value是string，压缩之后仍然是大key，则需要进行拆分，一个大key分为不同的部分，记录每个部分的key，使用multiget等操作实现事务读取。
  - 当value是list/set等集合类型时，根据预估的数据规模来进行分片，不同的元素计算后分到不同的片。

39. Redis常见性能问题和解决方案？

1. Master 最好不要做任何持久化工作，包括内存快照和 AOF 日志文件，特别是不要启用内存快照做持久化。
2. 如果数据比较关键，某个 Slave 开启 AOF 备份数据，策略为每秒同步一次。
3. 为了主从复制的速度和连接的稳定性，Slave 和 Master 最好在同一个局域网内。
4. 尽量避免在压力较大的主库上增加从库。
5. Master 调用 BGREWRITEAOF 重写 AOF 文件，AOF 在重写的时候会占大量的 CPU 和内存资源，导致服务 load 过高，出现短暂服务暂停现象。
6. 为了 Master 的稳定性，主从复制不要用图状结构，用单向链表结构更稳定，即主从关为：Master<－ Slave1<－Slave2<－Slave3…，这样的结构也方便解决单点故障问题，实现 Slave 对 Master 的替换，也即，如果 Master 挂了，可以立马启用 Slave1 做 Master，其他不变。

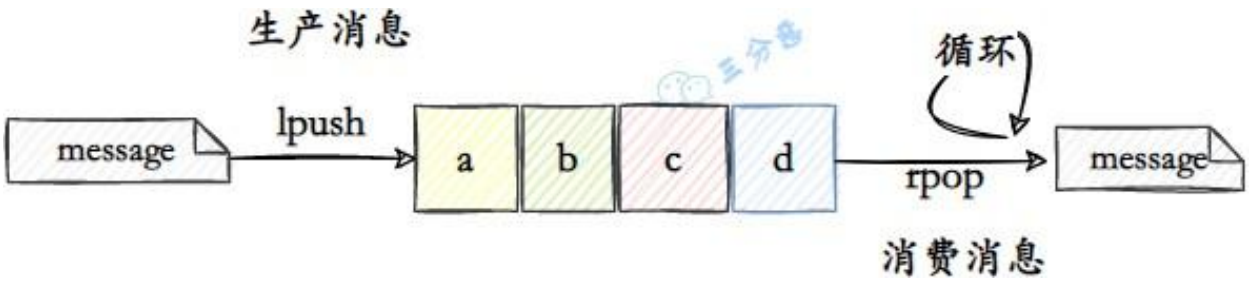
Redis应用

40. 使用Redis 如何实现异步队列？

我们知道redis支持很多种结构的数据，那么如何使用redis作为异步队列使用呢？一般有以下几种方式：

- **使用list作为队列，lpush生产消息，rpop消费消息**

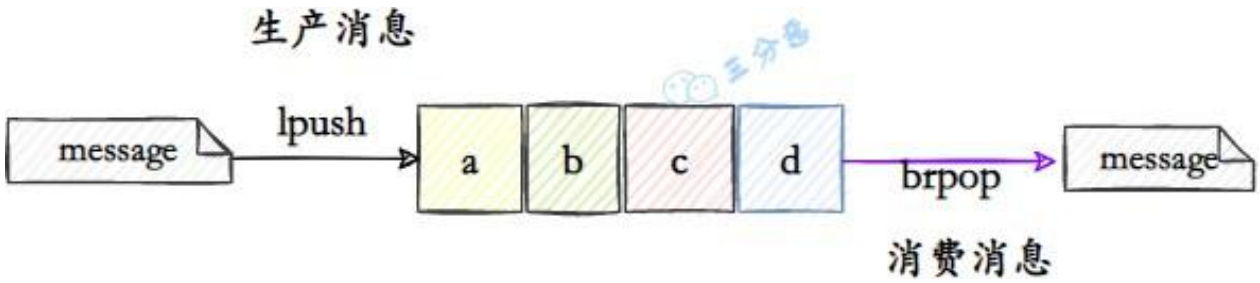
这种方式，消费者死循环rpop从队列中消费消息。但是这样，即使队列里没有消息，也会进行rpop，会导致Redis CPU的消耗。



可以通过让消费者休眠的方式的方式来处理，但是这样又会又消息的延迟问题。

-**使用list作为队列，lpush生产消息，brpop消费消息**

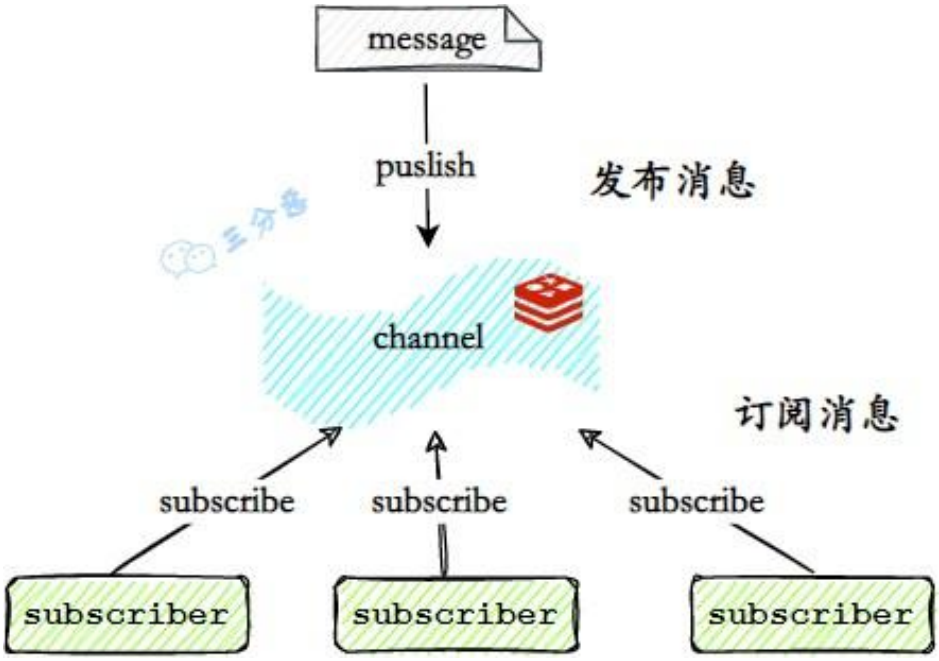
brpop是rpop的阻塞版本，list为空的时候，它会一直阻塞，直到list中有值或者超时。



这种方式只能实现一对一的消息队列。

- **使用Redis的pub/sub来进行消息的发布/订阅**

发布/订阅模式可以1：N的消息发布/订阅。发布者将消息发布到指定的频道频道（channel），订阅相应频道的客户端都能收到消息。

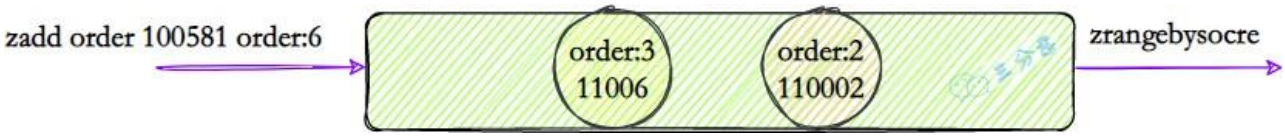


但是这种方式不是可靠的，它不保证订阅者一定能收到消息，也不进行消息的存储。所以，一般的异步队列的实现还是交给专业的消息队列。

41. Redis 如何实现延时队列?

- **使用zset，利用排序实现**

可以使用 zset这个结构，用设置好的时间戳作为score进行排序，使用 zadd score1 value1..... 命令就可以一直往内存中生产消息。再利用 zrangebysocre 查询符合条件的所有待处理的任务，通过循环执行队列任务即可。



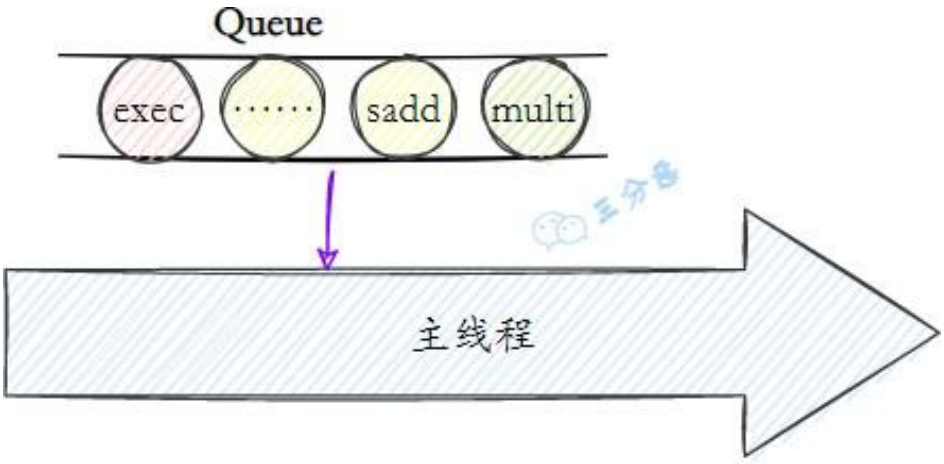
42. Redis 支持事务吗？

Redis提供了简单的事务，但它对事务ACID的支持并不完备。

multi命令代表事务开始，exec命令代表事务结束，它们之间的命令是原子顺序执行的：

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> sadd user:a:follow user:b
QUEUED
127.0.0.1:6379> sadd user:b:fans user:a
QUEUED
127.0.0.1:6379> sismember user:a:follow user:b
(integer) 0
127.0.0.1:6379> exec 1) (integer) 1
2) (integer) 1
```

Redis事务的原理，是所有的指令在 exec 之前不执行，而是缓存在
服务器的一个事务队列中，服务器一旦收到 exec 指令，才开执行整个事务队列，执行完毕后一次性返回所有指令的运行结果。



因为Redis执行命令是单线程的，所以这组命令顺序执行，而且不会被其它线程打断。

Redis**事务的注意点有哪些？**

需要注意的点有：

- Redis 事务是不支持回滚的，不像 MySQL 的事务一样，要么都执行要么都不执行；
- Redis 服务端在执行事务的过程中，不会被其他客户端发送来的命令请求打断。直到事务命令全部执行完毕才会执行其他客户端的命令。

Redis **事务为什么不支持回滚？**

Redis 的事务不支持回滚。

如果执行的命令有语法错误，Redis 会执行失败，这些问题可以从程序层面捕获并解决。但是如果出现其他问题，则依然会继续执行余下的命令。

这样做的原因是因为回滚需要增加很多工作，而不支持回滚则可以**保持简单、快速的特性**。

43. Redis和Lua脚本的使用了解吗？

Redis的事务功能比较简单，平时的开发中，可以利用Lua脚本来增强Redis的命令。 Lua

脚本能给开发人员带来这些好处：

- Lua脚本在Redis中是原子执行的，执行过程中间不会插入其他命令。
- Lua脚本可以帮助开发和运维人员创造出自己定制的命令，并可以将这 些命令常驻在Redis内存中，实现复用的效果。
- Lua脚本可以将多条命令一次性打包，有效地减少网络开销。

比如这一段很（烂）经（大）典（街）的秒杀系统利用lua扣减Redis库存的脚本：

```
-- 库存未预热
if (redis.call('exists', KEYS[2]) == 1) then
    return -9;
end;
-- 秒杀商品库存存在
if (redis.call('exists', KEYS[1]) == 1) then
    local stock = tonumber(redis.call('get', KEYS[1]));
    local num = tonumber(ARGV[1]);
    --  余库存少  请求数量
    if (stock < num) then
        return -3
    end;
    --  减库存
    if (stock >= num) then
        redis.call('incrby', KEYS[1], 0 - num);
        --  减成功
        return 1
    end;
    return -2;
end;
-- 秒杀商品库存不存在
return -1;
```

44.Redis的管道了解吗？

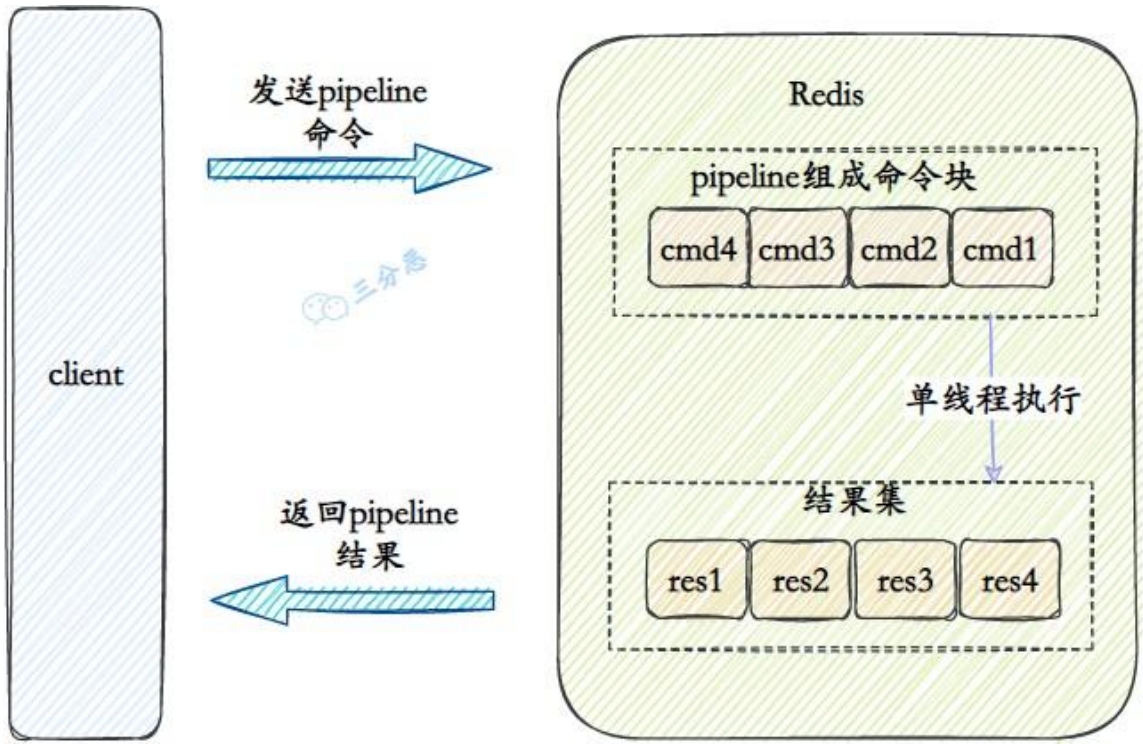
Redis 提供三种将客户端多条命令打包发送给服务端执行的方式：

Pipelining(管道) 、 Transactions(事务) 和 Lua Scripts(Lua 脚本) 。

Pipelining（管道）

Redis 管道是三者之中最简单的，当客户端需要执行多条 redis 命令时，可以通过管道一次性将要执行的多条命令发送给服务端，其作用是为了降低 RTT(Round Trip Time) 对性能的影响，比如我们使用 nc 命令将两条指令发送给 redis 服务端。

Redis 服务端接收到管道发送过来的多条命令后，会一直执命令，并将命令的执行结果进行缓存，直到最后一条命令执行完成，再所有命令的执行结果一次性返回给客户端 。



Pipelining**的优势**

在性能方面， Pipelining 有下面两个优势：

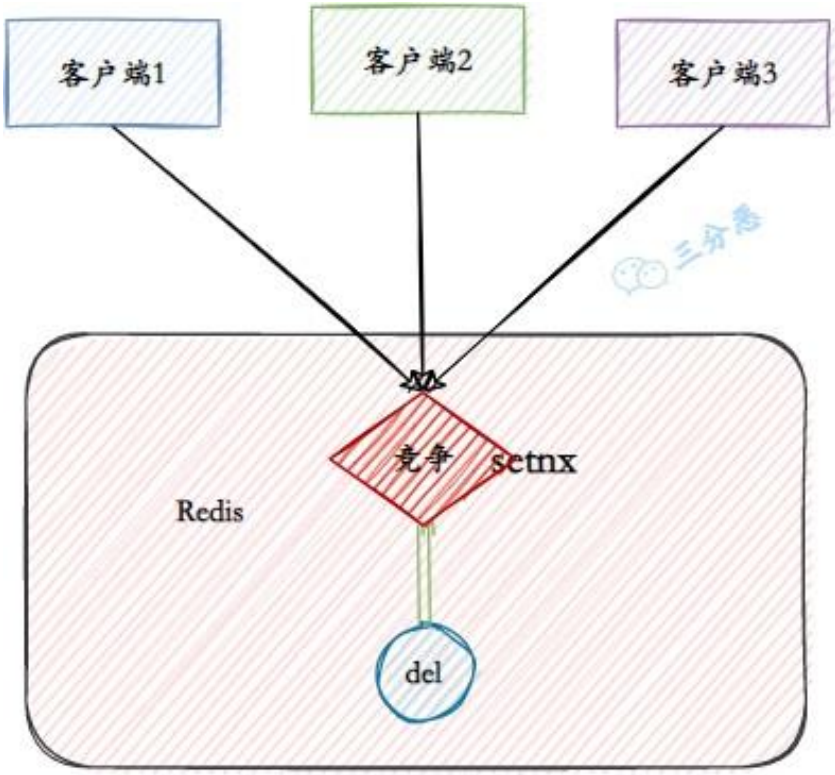
- **节省了**RTT：将多条命令打包一次性发送给服务端，减少了客户端与服务端之间的网络调用次数
- **减少了上下文切换**：当客户端/服务端需要从网络中读写数据时，都会产生一次系统调用，系统调用是非常耗时的操作，其中设计到程序由用户态切换到内核态，再从内核态切换回用户态的过程。当我们执行10 条 redis 命令的时候，就会发生 10 次用户态到内核态的上下文切换，但如果我们使用 Pipeining 将多条命令打包成一条一次性发送给服务端，就只会产生一次上下文切换。

45. Redis实现分布式锁了解吗？

Redis是分布式锁本质上要实现的目标就是在 Redis 里面占一个“茅坑”，当别的进程也要来占时，发现已经有人蹲在那里了，就只好放弃或者稍后再试。

- V1：setnx**命令**

占坑一般是使用 setnx(set if not exists) 指令，只允许被一个客户端占坑。先来先占， 用完了，再调用 del 指令释放茅坑。

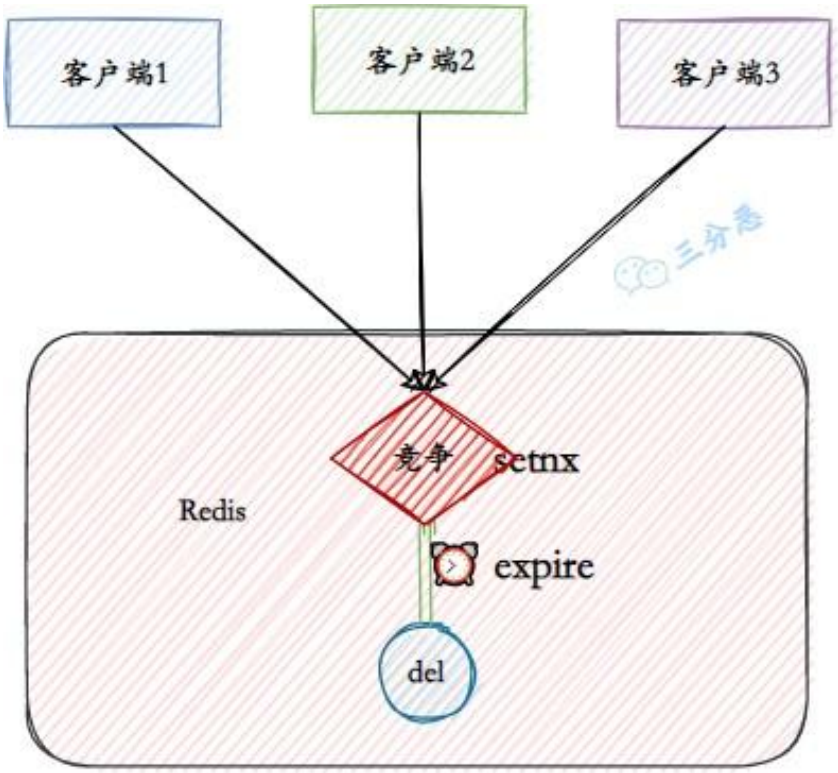



```
> setnx lock:fighter true
OK
... do something critical ...
> del lock:fighter
(integer) 1
```

但是有个问题，如果逻辑执行到中间出现异常了，可能会导致 del 指令没有被调用，这样就会陷入死锁，锁永远得不到释放。

- V2:**锁超时释放**

所以在拿到锁之后，再给锁加上一个过期时间，比如 5s，这样即使中间出现异常也可以保证 5 秒之后锁会自动释放。



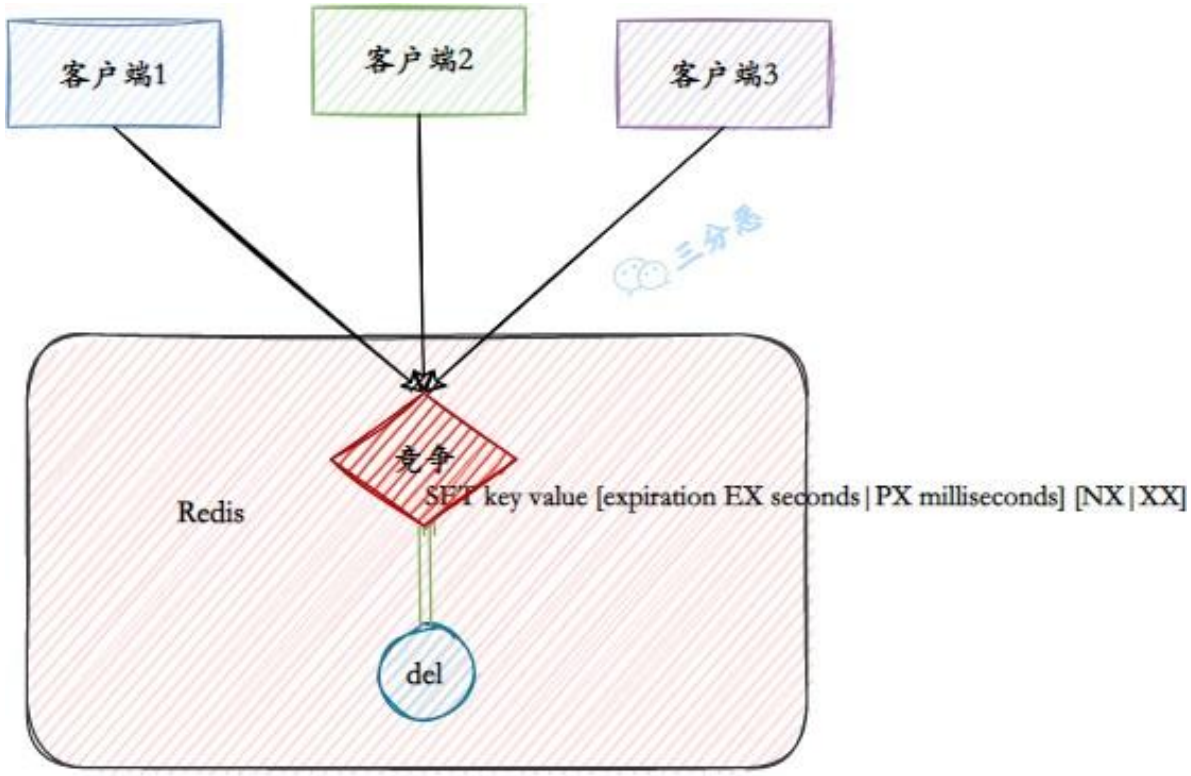

```
> setnx lock:fighter true
OK
> expire lock:fighter 5
... do something critical ...
> del lock:fighter
(integer) 1
```

但是以上逻辑还有问题。如果在 setnx 和 expire 之间服务器进程突然挂掉了，可能是因为机器掉电或者是被人为杀掉的，就会导致 expire 得不到执行，也会造成死锁。

这种问题的根源就在于 setnx 和 expire 是两条指令而不是原子指令。如果这两条指令可以一起执行就不会出现问题。

- V3:set指令

这个问题在Redis 2.8 版本中得到了解决，这个版本加入了set 指令的扩展参数，使得 setnx 和expire 指令可以一起执行。




```
set lock:fighter3 true ex 5 nx OK ... do something critical ... > del lock:codehole
```

上面这个指令就是 setnx 和 expire 组合在一起的原子指令，这个就算是比较完善的分布式锁了。当然实际的开发，没人会去自己写分布式锁的命令，因为有专业的轮子——Redisson。

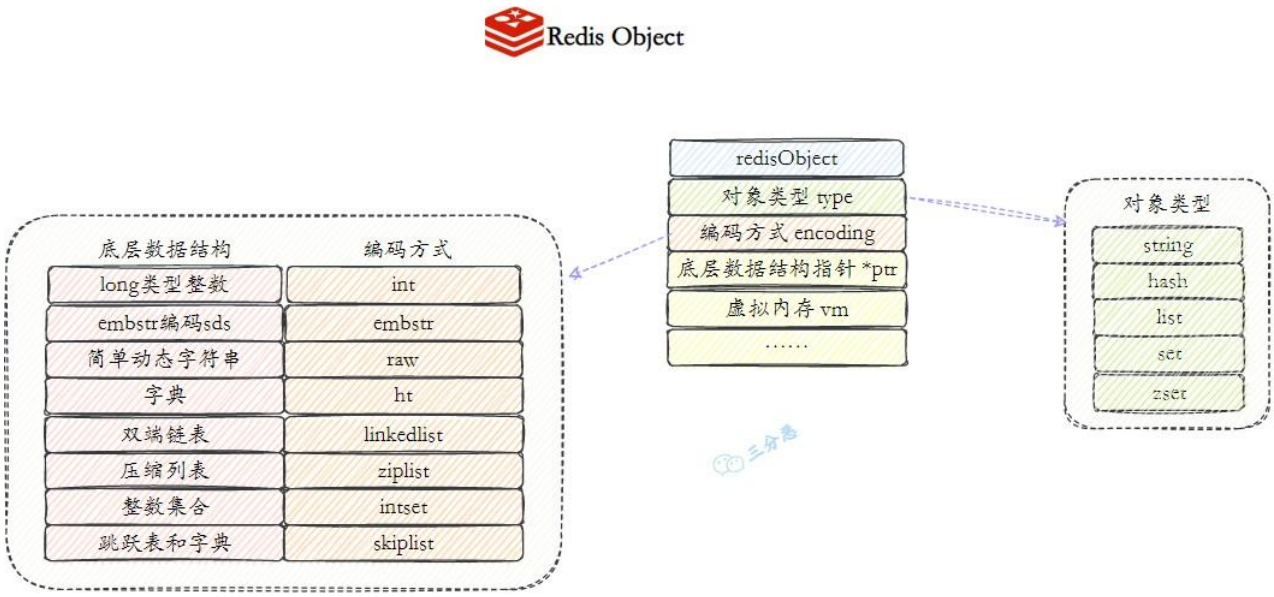
底层结构

这一部分就比较深了，如果不是简历上写了精通Redis，应该不会怎么问。

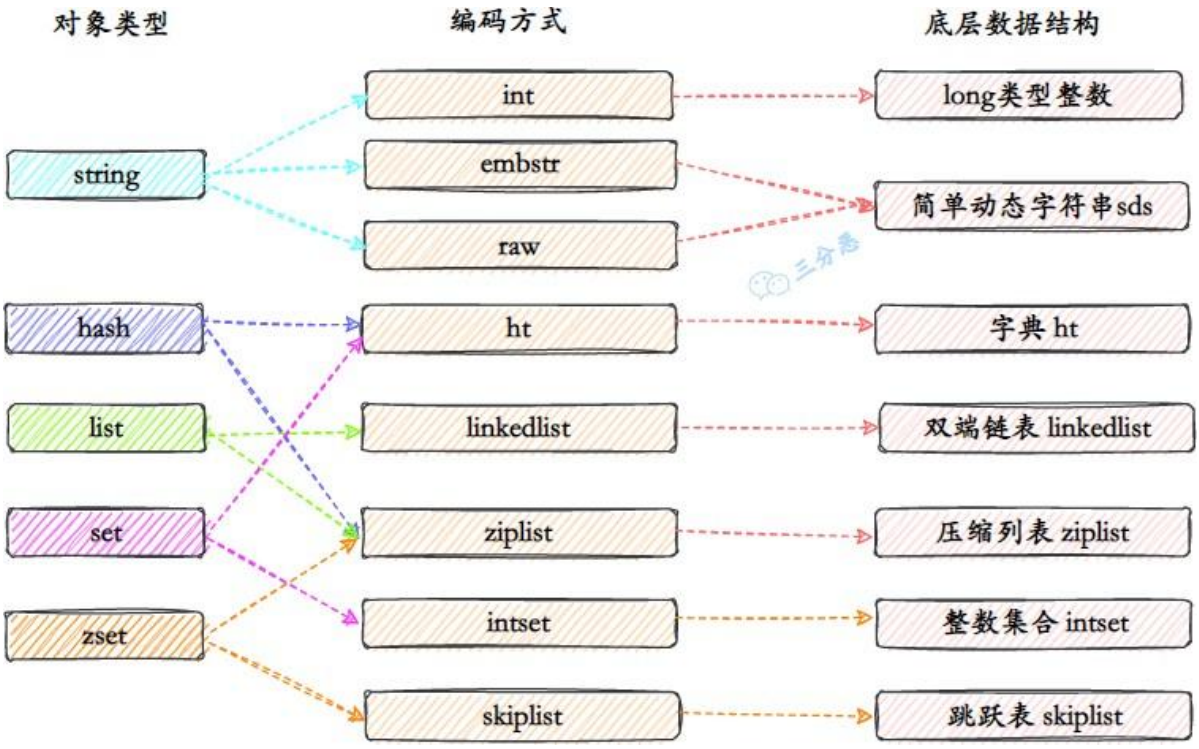
46. 说说Redis底层数据结构？

Redis有**动态字符串**(sds)、**链表**(list)、**字典**(ht)、**跳跃表**(skiplist)、**整数集合**(intset)、**压缩列表**(ziplist) 等底层数据结构。

Redis并没有使用这些数据结构来直接实现键值对数据库，而是基于这些数据结构创建了一个对象系统，来表示所有的key-value。



我们常用的数据类型和编码对应的映射关系：



简单看一下底层数据结构，如果对数据结构掌握不错的话，理解这些结构应该不是特别难：

1. **字符串**：redis没有直接使用C语言传统的字符串表示，而是自己实现的叫做简单动态字符串SDS的抽象类型。 C语言的字符串不记录自身的长度信息，而SDS则保存了长度信息，这样将获取字符串长度的时间由O(N)降低到了O(1)，同时可以避免缓冲区溢出和减少修改字符串长度时所需的内存重分配次数。

2. **链表**linkedlist：redis链表是一个双向无环链表结构，很多发布订阅、慢查询、监视器功能都是使用到了链表来实现，每个链表的节点由一个listNode结构来表示，每个节点都有指向前置节点和后置节点的指针，同时表头节点的前置和后置节点都指向NULL。

3. **字典**dict：用于保存键值对的抽象数据结构。Redis使用hash表作为底层实现，一个哈希表里可以有多个哈希表节点，而每个哈希表节点就保存了字典里中的一个键值对。
   每个字典带有两个hash表，供平时使用和rehash时使用，hash表使用链地址法来解决键冲突，被分配到同一个索引位置的多个键值对会形成一个单向链表，在对hash表进行扩容或者缩容的时候，为了服务的可用性， rehash的过程不是一次性完成的，而是渐进式的。

4. **跳跃表**skiplist：跳跃表是有序集合的底层实现之一，Redis中在实现有序集合键和集群节点的内部结构中都是用到了跳跃表。Redis跳跃表由zskiplist和zskiplistNode组成，zskiplist用于保存跳跃表信息（表头、表尾节点、长度等），zskiplistNode用于表示表跳跃节点，每个跳跃表节点的层高都是1-32的随机数，在同一个跳跃表中，多个节点可以包含相同的分值，但是每个节点的成员对象必须是唯一的，节点按照分值大小排 序，如果分值相同，则按照成员对象的大小排序。

5. **整数集合**intset：用于保存整数值的集合抽象数据结构，不会出现重复元素，底层实现为数组。

6. **压缩列表**ziplist：压缩列表是为节约内存而开发的顺序性数据结构，它可以包含任意多个节点，每个节点可以保存一个字节数组或者整数值。

47. Redis 的 SDS 和 C 中字符串相比有什么优势？

C 语言使用了一个长度为 `N+1` 的字符数组来表示长度为 `N` 的字符串，并且字符数组最后一个元素总是 `\0`，这种简单的字符串表示方式 不符合 Redis 对字符串在安全性、效率以及功能方面的要求。

> C语言的字符串可能有什么问题？

这样简单的数据结构可能会造成以下一些问题：

- **获取字符串长度复杂度高** ： 因为 C 不保存数组的长度，每次都需要遍历一遍整个数组，时间复杂度为O(n)；不
- 能杜绝 **缓冲区溢出/内存泄漏** 的问题 : C字符串不记录自身长度带来的另外一个问题是容易造成缓存区溢出（buffer overflow），例如在字符串拼接的时候，新的
- C 字符串 **只能保存文本数据** → 因为 C 语言中的字符串必须符合某种编码（比如 ASCII），例如中间出现的 `'\0'` 可能会被判定为提前结束的字符串而识别不了；

> Redis如何解决？优势？

简单来说一下 Redis 如何解决的：

1. **多增加 len 表示当前字符串的长度**：这样就可以直接获取长度了，复杂度 O(1)；
2. **自动扩展空间**：当 SDS 需要对字符串进行修改时，首先借助于 `len` 和 `alloc` 检查空间是否满足修改所需的要求，如果空间不够的话，SDS 会自动扩展空间，避免了像 C 字符串操作中的溢出情况；
3. **有效降低内存分配次数**：C 字符串在涉及增加或者清除操作时会改变底层数组的大小造成重新分配，SDS 使

用了 **空间预分配** 和 **惰性空间释放** 机制，简单理解就是每次在扩展时是成倍的多分配的，在缩容是也是先留着并不正式归还给 OS；

4. **二进制安全**：C 语言字符串只能保存 `ascii` 码，对于图片、音频等信息无法保存，SDS 是二进制安全的，写入什么读取就是什么，不做任何过滤和限制；

48.字典是如何实现的？Rehash 了解吗？

字典是 Redis 服务器中出现最为频繁的复合型数据结构。除了 hash 结构的数据会用到字典外，整个 Redis 数据库的所有 `key` 和 `value` 也组成了一个 **全局字典**，还有带过期时间的 `key` 也是一个字典。*(存储在 RedisDb 数据结构中)*

> 字典结构是什么样的呢？

Redis 中的字典相当于 Java 中的 HashMap，内部实现也差不多类似，采用哈希与运算计算下标位置；通过 "**数组 + 链表**" 的**链地址法** 来解决哈希冲突，同时这样的结构也吸收了两种不同数据结构的优点。

> 字典是怎么扩容的？

字典结构内部包含 **两个** hashtable，通常情况下只有一个哈希表 ht[0] 有值，在扩容的时候，把ht[0]里的值 rehash到ht[1]，然后进行**渐进式**rehash ——所谓渐进式rehash，指的是这个rehash的动作并不是一次性、集中式地完成的，而是分多次、渐进式地完成的。

待搬迁结束后，h[1]就取代h[0]存储字典的元素。

49.跳跃表是如何实现的？原理？

PS:跳跃表是比较常问的一种结构。

跳跃表（skiplist）是一种有序数据结构，它通过在每个节点中维持多个指向其它节点的指针，从而达到快速访问节点的目的。

> 为什么使用跳跃表?

首先，因为 zset 要支持随机的插入和删除，所以它 **不宜使用数组来实现**，关于排序问题，我们也很容易就想到 **红黑树/ 平衡树** 这样的树形结构，为什么 Redis 不使用这样一些结构呢？

1. **性能考虑：** 在高并发的情况下，树形结构需要执行一些类似于 rebalance 这样的可能涉及整棵树的操作，相对来说跳跃表的变化只涉及局部；
2. **实现考虑：** 在复杂度与红黑树相同的情况下，跳跃表实现起来更简单，看起来也更加直观；

基于以上的一些考虑，Redis 基于 William Pugh 的论文做出一些改进后采用了 **跳跃表** 这样的结构。本质是解决查找问题。

> 跳跃表是怎么实现的？

跳跃表的节点里有这些元素：

- **层**
  跳跃表节点的level数组可以包含多个元素，每个元素都包含一个指向其它节点的指针，程序可以通过这些层来加快访问其它节点的速度，一般来说，层的数量月多，访问其它节点的速度就越快。

每次创建一个新的跳跃表节点的时候，程序都根据幂次定律，随机生成一个介于1和32之间的值作为level数组的大小，这个大小就是层的“高度”

- **前进指针**
  每个层都有一个指向表尾的前进指针（level[i].forward属性），用于从表头向表尾方向访问节点。我们看一下跳跃表从表头到表尾，遍历所有节点的路径：

- **跨度**
  层的跨度用于记录两个节点之间的距离。跨度是用来计算排位（rank）的：在查找某个节点的过程中，将沿途访问过的所有层的跨度累计起来，得到的结果就是目标节点在跳跃表中的排位。

  例如查找，分值为3.0、成员对象为o3的节点时，沿途经历的层：查找的过程只经过了一个层，并且层的跨度为3，所以目标节点在跳跃表中的排位为3。

- **分值和成员**
  节点的分值（score属性）是一个double类型的浮点数，跳跃表中所有的节点都按分值从小到大来排序。节点的成员对象（obj属性）是一个指针，它指向一个字符串对象，而字符串对象则保存这一个SDS值。

50. 压缩列表了解吗？

压缩列表是 Redis **为了节约内存** 而使用的一种数据结构，是由一系列特殊编码的连续内存快组成的顺序型数据结构。

一个压缩列表可以包含任意多个节点（entry），每个节点可以保存一个字节数组或者一个整数值。压

缩列表由这么几部分组成：

- zlbyttes:记录整个压缩列表占用的内存字节数
- zltail:记录压缩列表表尾节点距离压缩列表的起始地址有多少字节
- zllen:记录压缩列表包含的节点数量
- entryX:列表节点
- zlend:用于标记压缩列表的末端

51. 快速列表 quicklist 了解吗？

Redis 早期版本存储 list 列表数据结构使用的是压缩列表 ziplist 和普通的双向链表 linkedlist，也就是说当元素少时使用 ziplist，当元素多时用 linkedlist。

但考虑到链表的附加空间相对较高，`prev` 和 `next` 指针就要占去 `16` 个字节（64 位操作系统占用 `8` 个字节），另外每个节点的内存都是单独分配，会家具内存的碎片化，影响内存管理效率。

后来 Redis 新版本（3.2）对列表数据结构进行了改造，使用 `quicklist` 代替了 `ziplist` 和

quicklist是综合考虑了时间效率与空间效率引入的新型数据结构。

quicklist由list和ziplist结合而成，它是一个由ziplist充当节点的双向链表。

其他问题

52. 假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如何将它们全部找出来？

使用 `keys` 指令可以扫出指定模式的 key 列表。但是要注意 keys 指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用 `scan` 指令，`scan` 指令可以无阻塞的提取出指定模式的 `key` 列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用指 `keys` 令长。