

Redis 7 面试速成宝典

金毛败犬 著

通过精选 30 道面试题
带你速成 Redis 7 底层原理

没有出版社出版

前言

这是 Redis 面试题资料还是电子书？

你可以把它看成是一本电子书（因为整体结构像写书），也可以把它看成是面试题资料（因为和面试题有关），还可以把它看成是博客文章集合（因为我写起来像写博客）。但是我个人更愿意把它叫作“电子书”。

我为什么要写这本书？

这是 2023 年 4 月份创作的，今年参与过春招的人就知道，今年互联网行业参与招聘的人数极多，岗位的 hc 却极少。网络上还出现了一些小公司 Java 岗招聘一开，马上有千人投递的图片。在这样内卷的局面下，如果不选择躺平，我们不得不给自己添加一些亮点。我写这本书主要就是为了找工作的大家服务的，选择 Redis 是因为它应用广泛，同时也是后端面试常考的一门缓存数据库，最主要的是我对它比较熟悉一些，所以我觉得自己可以完成这本书的撰写。

这本书是什么内容？

这本书的背景是 Redis 7.0（目前最新版本），精选了 30 道面试题。我们通过面试题作为主题，讲解到 Redis 比较深的底层原理，但是又不会太深入源码，让大家能够易理解，并且找工作的大家也希望节省时间，同时也是让大家通过理解 30 道题的底层原理，能够做到对 Redis 7.0 整体的底层原理有一个基本的掌握。

受众群体是哪些？

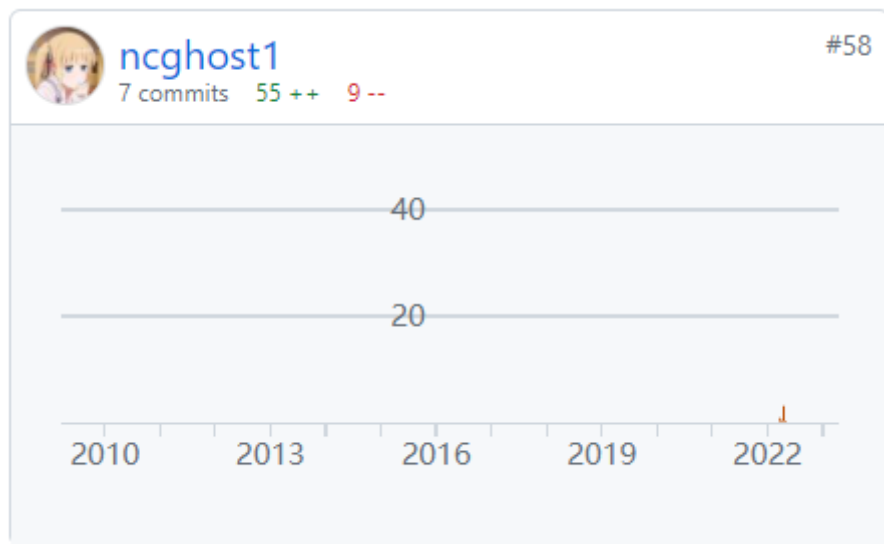
最主要的受众群体就是需要找工作的同学们，不管你是现在正在找工作，还是未来需要找工作的在校生的都可以看。我们想一想，同样的 Redis 面试题，如果我们能回答出较深的底层原理，甚至给面试官讲出 Redis 7 在这里做了什么改动，是不是比起其他人是个很大很大的亮点呢？

除此之外，如果你已经有了工作，但是想要快速了解 Redis 7 最新特性，或者说速成 Redis 底层也可以看。


如果你是 Redis 小白想知道可以看吗？我的回答是可以，但是我们是不讲 Redis 基本用法的，你需要先去学习 Redis 的常用数据类型的常用命令，知道怎么用再看比较好。当然你也可以一边看一边上 Redis 官方命令文档查询用法，如果看不懂英文，那么我的博客也有五种基本数据类型的命令介绍，不过是较早之前写的可能格式不太好看：<https://www.erirspace.cn/categories/Redis%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0/>

作者是何许人也？

作者混 IT 圈用的网名叫金毛败犬，是个卑微的 Redis 开源贡献者，虽然比较水，只有 +55 行的贡献，但是也是前一百名...你可以点开 redis 仓库的 contributor 看到。



作者还在 github 上创办了一个专门做热门开源项目源码中文注释的组织，**CN-Annotation-Team**，并且创建并开源了一个 Redis 7.0 版本的中文注释仓库，目前已经开源了一年，收获了 300+ stars，大家一起贡献了上万行的中文注释，希望读者也可以来贡献注释哦。仓库链接：<https://github.com/CN-annotation-team/redis7.0-chinese-annotated>



CN-Annotation-Team
我们是一个用爱发电的，制作各大热门开源项目的源码中文注释的组织！
73 followers · China

[Overview](#) [Repositories 4](#) [Projects](#) [Packages](#) [Teams 2](#) [People 18](#) [Settings](#)

Chinese-Annotation-Team介绍:

Hello!这里是一个用爱发电的对各热门开源项目进行中文注释的组织! 🌟🌟
我们正在筹备各种热门开源项目的中文注释仓库的上线哦! 请持续关注我们, 点击右上角follow吧! 🐼🐼
本组织的目的是帮助中文区的大家能够更轻松地去学习开源项目的源码, 我们希望你能够通过源码去理解你在使用的项目, 而不要再只是单纯背什么八股文啦! 八股文可能会骗你, 但源码不会! 🐼🐼
我们也希望更多人能参与到组织中来, 只要越多人用爱发电, 学习源码就能变得越来越容易了! 🌟🌟

加入Chinese-Annotation-Team

如果你有奉献精神, 同时是个喜欢分享知识的人, 我们非常希望你能加入组织。但是怎么让我们知道你有这样的品质呢? 这倒不是嘴上说说就行呢 🐼🐼, 所以我们想了个办法:

- 对我们的中文注释仓库贡献超过50行注释, 我们会主动邀请你加入组织 🐼🐼
- 如果你想要推动一个组织暂时没有的热门开源项目的中文注释项目, 你可以到github仓库下提issue让我们知道, 我们会和你进行讨论, 若你能满足一些要求后我们会邀请你加入组织, 并让你作为你所推动项目的负责人。 🐼🐼

Popular repositories

redis7.0-chinese-annotated Public
Redis 7.0.5 版本——中文注释, 持续更新! 欢迎参与本项目! 🐼🐼🐼
C 344 108

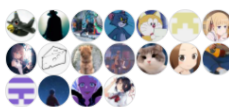
.github Public

redis Public
Forked from redis/redis
Redis is an in-memory database that persists on disk. The data model is key-value, but many different kind of values are supported: Strings, Lists, Sets, Sorted Sets, Hashes, Streams, HyperLogLogs,...

kafka3.3-chinese-annotated Public
kafka3.3 中文注释

View as: Public
You are viewing the README and pinned repositories as a public user.
You can [pin repositories](#) visible to anyone.
[Get started with tasks](#) that most successful organizations complete.


Discussions
Set up discussions to engage with your community!
[Turn on discussions](#)

People

[Invite someone](#)

Top languages
C Java

Most used topics Manage
[chinese-simplified](#) [redis](#) [redis-annotated](#)

最后附上我的 github 主页：<https://github.com/ncghost1>，欢迎大家光顾。



JJ Lu

ncghost1

Even if I can't be the best one, I'll try to be the one who works the hardest.

Edit profile

71 followers · 35 following


Magic School


China Nanning


eriri233@qq.com

www.eririspace.cn

Achievements

 x3


 x2

 x2

Beta

Send feedback

Organizations



Overview

Repositories 24

Projects

Packages

Stars 67

ncghost1 / README.md

Hi, I'm a Junior at a magic school and love open source and sharing.

JJ Lu's GitHub Stats

Total Stars Earned:	121
Total Commits (2023):	233
Total PRs:	32
Total Issues:	5
Contributed to (last year):	8

A+

Pinned

Customize your pins

redis/redis

Public

Redis is an in-memory database that persists on disk. The data model is key-value, but many different kind of values are supported: Strings, Lists, Sets, Sorted Sets, Hashes, Streams, HyperLogLogs,...

C

59.7k

22.5k

CN-annotation-team/redis7.0-chinese-annotated

Public

Redis 7.0.5 版本——中文注释, 持续更新! 欢迎参与本项目!

C

344

108

Redis-ReplicaLock

Public

A distributed lock idea implemented by redis and golang. It is more secure than a simple redis distributed lock.

Go

3

MassDataProcessingLab

Public

海量数据处理经典面试题的 Go 语言实现, 此外还提供 lab 来亲身实践~ Golang implementation of classical interview questions of mass data processing, also provide lab for you to practice.

Go

77

10

Mini-Tiktok

Public

这是一个来源于字节跳动后浪训练营的项目。A project from ByteDance backend youth training camp.

Go

12

1

HDT3213/godis

Public

A Golang Implemented Redis Server and Cluster. Go 语言实现的 Redis 服务器和分布式集群

Go

2.7k

466

发现内容有误, 该如何反馈?

如果你是从正版渠道购买来的, 那么我是提供了一个永久顾问服务的, 也就是会给你加我的微信, 以后有 Redis 相关问题都可以无偿问我, 同样也可以反馈错误内容, 更新之后我会再将最新版发送给加了微信的同学们。如果你是从网上看到的这份资料, 或者是不知道从谁手中买来的, 哎哟我滴乖乖, 赶紧进正版渠道购买啦:

官方淘宝店铺: 金毛败犬的小店



金毛败犬

刚刚来过

这家伙很神秘，没有写个人简介。

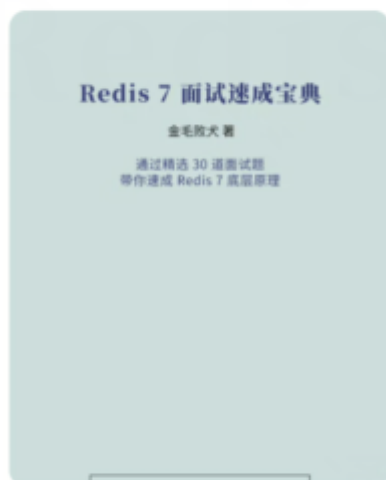
更多

芝麻信用未授权 > IP 福建

0 关注 0 粉丝

编辑资料

宝贝¹ 帖子 评价 动态³



上新优惠！Redis开源贡献者作品《Redis7面试

¥19.99 ¥29.99

+ 发布

没有更多了~

Redis member 推荐！Redis 7 面试速成宝典 PDF + 永久售后顾问

价格 **¥19.99** 0 月销量

配送 北京 至 北京西城区 快速 免运费 现货，付款后24小时内发货

付款方式 **全额支付**

数量 件

立即购买 **加入购物车**

金毛败犬家的小店

累计信用：0

联系： [和我联系](#)

资质： [资质](#)

该店铺尚未收到评价

[进入店铺](#) [收藏店铺](#)

官方淘宝店铺：金毛败犬的小店



目录

前言

目录

1. Redis 7 新特性

MP-AOF

listpack

全局共享复制积压缓冲区

2. Redis7 数据类型

3. 说说 String 数据类型

应用场景

扩容策略

内部编码

4. 说说 Hash 数据类型

应用场景

内部编码

哈希表扩容、缩容策略

5. 说说 List 数据类型

应用场景

内部编码

压缩节点

6. 说说 Set 数据类型

应用场景

内部编码

整数集合

7. 说说 Zset 数据类型

应用场景

内部编码

跳表

8. 说说 listpack 结构

listpack 结构设计

9. 说说 SDS 结构

为什么要有 SDS

SDS 底层结构

SDS 底层操作

SDS 预分配空间和惰性释放空闲空间策略

10. 说说 dict 结构

dict 结构设计

渐进式扩容与缩容策略

一直有持久化子进程存在是不是就不会扩容了？

11. 用 stream 做消息队列行不行？

pub/sub 能不能做消息队列

stream 特性

stream 做消息队列行不行：

简谈 Stream 底层结构

12. Redis 7 内存淘汰策略

近似 LRU 算法

近似 LFU 算法

13. Redis 7 懒惰删除

为什么需要懒惰删除

懒惰删除过程

过期 key 删除策略

为什么删除了很多 key 内存占用还是很高？

14. Redis 7 后台 IO

异步操作类型

异步处理过程

15. Redis 7 线程模型

事件循环器（IO 多路复用）

多线程与线程模型

16. Redis 7 事务

事务机制

17. Redis 7 缓存穿透，缓存击穿，缓存雪崩问题

缓存穿透

缓存击穿

缓存雪崩

个人看法

18. Redis 7 大 key 问题

什么是大 key 问题

解决方案

如何检测大 key

19. Redis 7 热点 key 问题

热点 key 产生原因：

热点 key 带来的问题

如何发现热点 key

解决方案

20. Redis 7 数据一致性问题

什么是数据一致性问题

一般更新策略

延迟双删和重试删除策略

读取 binlog 删除缓存：

21. Redis 7 持久化机制

RDB

AOF

MP-AOF

MP-AOF 和混合持久化

（附加题）同时开启 RDB 和 AOF，数据恢复的时候用哪个？

（附加题）AOF 开启了 always 策略能保证命令一定都落盘吗？

22. Redis 7 主从模式

主从架构

主从同步

23. Redis 7 哨兵模式

哨兵架构

哨兵机制

24. Redis 7 集群模式

集群模式简介

集群模式机制

25. Redis 7 实现分布式锁

为什么需要分布式锁？

单节点加锁

单节点加锁方案

多节点加锁

多节点加锁方案

26. Redis 7 实现延时队列

从简单到完善的延时队列实现

27. Redis 7 实现布隆过滤器

布隆过滤器

Redis 实现布隆过滤器

28. Redis 7 实现限流

限流方案

计数器固定窗口算法

计数器滑动窗口算法

漏斗限流算法

令牌桶算法

29. Redis 7 RESP 协议

传输-响应模型

RESP 协议

30. Redis 7 如何做内存优化

使用 32bit Redis

尽量使用 listpack

按位/字节操作

尽量使用 Hash 类型

Redis 内存分配

1. Redis 7 新特性

Redis 7.0 版本相比之前有不少改动，我们这里不挑命令，配置之类的改动来说，也不好记忆。对于 Redis 7.0 从我个人的角度来说，需要记住以下三个**关于结构**的重要改动：

MP-AOF

Redis 7.0 的 AOF 机制变成了 **MP-AOF (multipart-AOF)**，原本 AOF 功能是只使用单个 AOF 文件进行存储，现在分为了三类文件：**清单文件 (manifest)**，**基本文件 (base)**，**增量文件 (incr)**。

清单文件用于跟踪记录 base 和 incr 文件的信息，区分历史文件。基本文件为执行 AOF 重写时生成的文件，有 rdb 与 aof 两种格式，开启 rdb 功能时默认为 rdb 类型。增量文件记录每次重写后的写命令，为 aof 格式。数据恢复时根据清单文件查找最新的基本文件与它之后的增量文件进行恢复。

listpack

Redis 7.0 将 ziplist (压缩列表) 全部替换为了 listpack (紧凑列表)。紧凑列表与压缩列表相比是一个占用空间更小, 且没有“连锁更新”缺陷的数据结构。listpack 早在 5.0 版本就已经出现并作为 5.0 推出的 stream 数据类型的底层结构之一。

全局共享复制积压缓冲区

在 Redis 的部分同步中, 对于主从复制下的主节点, 主节点原本需要为每个从节点复制一份复制缓冲区。在这样的结构下, 从节点越多占用主节点内存也越多。在 Redis 7 中主节点只维护一份全局共享复制积压缓冲区, 为每个从节点维护一个复制偏移量, 通过偏移量在全局共享复制积压缓冲区中找到每个从节点开始同步的位置。全局共享复制积压缓冲区可以说很好地节省了主节点内存空间。

记忆要点: MP-AOF, 清单文件, 基本文件, 增量文件, listpack, 替换 ziplist, 占用空间更小, 没有连锁更新, 全局共享复制积压缓冲区

附赠一篇关于介绍 Redis 7 新特性比较详细的文章: https://blog.csdn.net/m0_51504545/article/details/124597713

2. Redis7 数据类型

Redis 7.0 支持的数据类型首先可以分为两大类: **Redis 内置数据类型**, **Redis Module**。

Redis 内置数据类型有五种类型是最常用的, 分别是: **String (字符串)**, **Hash (字典)**, **List (列表)**, **Set (集合)**, **Zset (又称 sortedset, 有序集合)**。

Redis 7.0 其它内置数据类型如下:

- Bitmap (位图, 可用于数据的压缩存储, 存在标记等)
- Hyperloglog (极致节省内存的集合结构, 用于海量数据下的基数统计)
- Pub/Sub (发布/订阅, 一种生产者/消费者模式, 用于消息转发, 但不负责存储)
- Geo (基于 zset 实现, 用于存储和操作地理位置信息)
- Stream (Redis 5.0 推出的消息队列)

Redis Module 是 4.0 增加的一种功能, 该功能可以**允许用户自定义扩展模块**, 在 redis 内部实现新的数据类型和功能, 使用统一的调用方式和传输协议格式扩展 redis 的能力。比较经典的 Redis Module 有: RedisBloom (布隆过滤器), RedisJson (支持 Json 结构)。

记忆要点: 两大类, 内置数据类型, 五种常用数据类型, String, Hash, List, Set, Zset, Redis Module, 用户自定义扩展模块

3. 说说 String 数据类型

String 即字符串类型, 它可以说是 Redis 最常用的数据类型, 且在 Redis 内部 key 都是用字符串类型存储的, 全局唯一。String 可以从三个方面来进行介绍:

应用场景

字符串类型的值能存储的内容种类很多，它的值可以是字符串，数字（整数和浮点数），还可以以二进制或特殊编码形式（如图片转 Base64 编码）存储图片，音频，视频等等。我们可以用 String 来**缓存**用户信息，同时也可以用于**计数**，例如播放量，点赞数等。

（ps：如果要存储的信息是 Json 结构的话，更好的选择是安装 RedisJson 模块使用 Json 结构进行存储，它的好处是可以对 Json 结构中的部分内容进行获取和修改。）

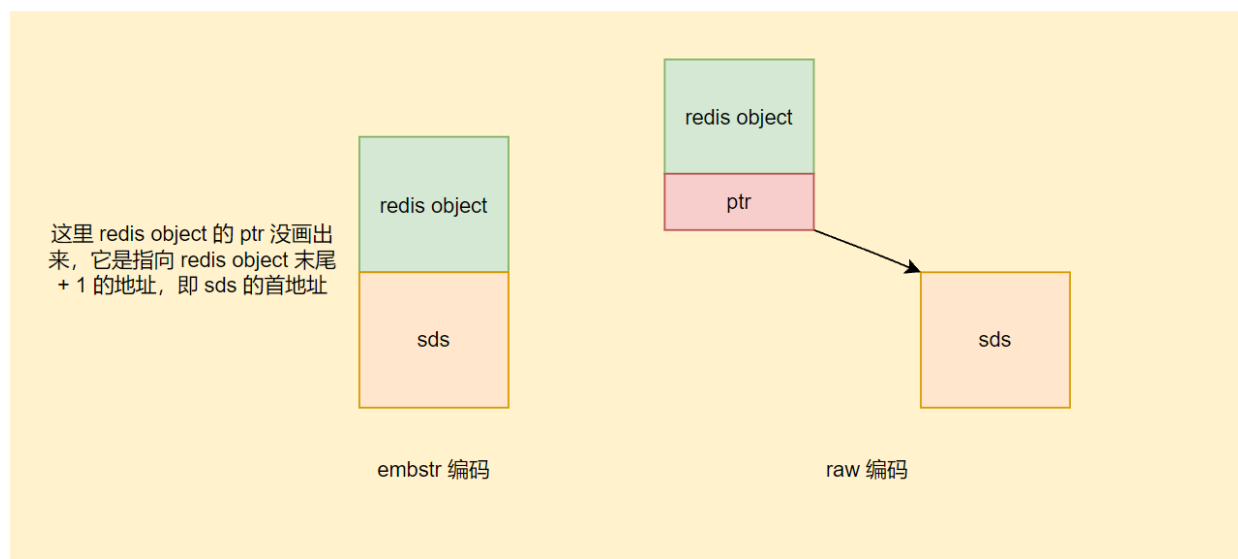
扩容策略

字符串类型是用预分配冗余空间的方式来减少内存的频繁分配。在**字符串值小于 1MB** 时，每次扩容进行**加倍扩容**，**大于等于 1MB** 以后每次扩容**增加 1MB**，最大不能超过 **512MB**。

内部编码

字符串类型有三种内部编码：**int**，**embstr**，**raw**。Redis 会根据当前值的类型和长度决定使用哪种内部编码实现。如 int 是 8 个字节的长整型（long long），embstr 是**小于等于 44 个字节**的字符串，raw 是**大于 44 个字节**的字符串。

embstr 和 raw 的区别在于：**embstr 编码将 redis object（redis 对象结构体，记录一些关于对象的元信息以及真实数据指针）和 sds（redis 源码的字符串结构体）分配在同一个 64B 内存块中，所以只会分配一次内存**，同时 embstr 编码字符串是不可修改的，如果修改则会转为 raw 编码字符串；而 **raw 编码 redis object 和 sds 在两个不同的内存块中，所以创建 raw 编码字符串需要分配两次内存。**



记忆要点：应用场景，缓存，计数，扩容策略，小于 1MB 加倍扩容，大于等于 1MB 每次扩容 1MB，最大 512MB，内部编码，int，embstr，raw，44 字节，redis object 和 sds，embstr 连续内存空间，raw 不连续内存空间，分配内存次数

4. 说说 Hash 数据类型

在 Redis 中，Hash 类型是指键值本身是一对键值对结构，即 `value = {field1, value1}, {field2, value2}, ...`，与编程语言中的 map 或 dict 类似。Hash 类型可以从以下三个方面进行介绍：

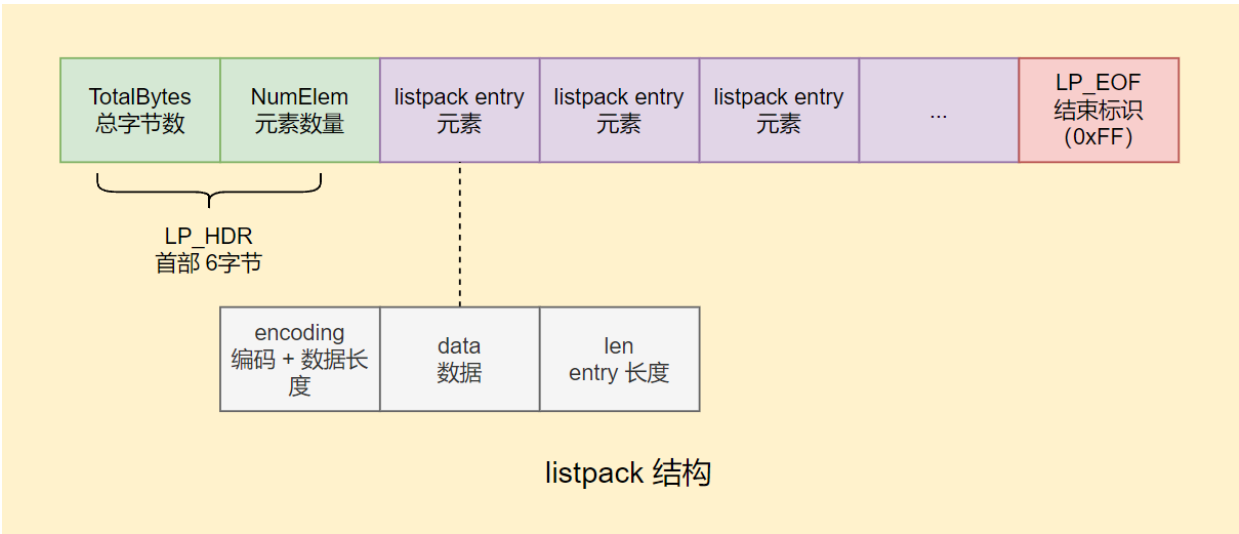
应用场景

我们一般将**有多个属性的结构**使用 Hash 类型进行存储，比如有 id，用户名，性别等属性的用户信息，如果使用 String 类型存储，可以每个属性用多个键分别存储，但占用键过多且信息内聚性较差；或者可以将全部属性进行序列化存储（如 Json 结构），这样只需要一个键进行存储，但序列化与反序列化也需要开销；而使用 Hash 类型可以每个用户属性使用一对 field-value 对进行存储，且只需要一个键，并且易于获取和修改部分属性内容。

内部编码

Redis 7.0 Hash 类型有两种内部编码：**listpack** 和 **hashtable**。

listpack（紧凑列表）：**当哈希类型元素个数小于等于 hash-max-listpack-entries 配置（默认 512 个）、同时所有值都小于等于 hash-max-listpack-value 配置（默认 64 字节）时**，Redis 会使用 listpack 作为 Hash 的内部实现，**listpack 使用连续的内存与紧凑的结构实现多个元素的连续存储，所以在节省内存方面比 hashtable 更加优秀。**



hashtable（哈希表）：**当哈希类型无法满足 listpack 的条件时**，Redis 会使用 **hashtable** 作为 Hash 的内部实现，**因为此时 listpack 的读写效率会下降**，而 hashtable 的读写时间复杂度为 $O(1)$ 。（hashtable 编码就不带结构图了，可以想象成链地址法的哈希表，关于底层结构 dict 具体可以看第 10 篇）

在 Redis 7.0 之前是使用 **ziplist** 和 hashtable 作为内部编码，而 ziplist 因为它的每个 entry 需要**存储上一个 entry 的长度**，所以会有触发“**连锁更新**”的风险，而 listpack 的每个 entry 则是**存储自己的长度**，发生更新时不会影响到其它 entry，而且 listpack 占用空间比 ziplist 要更小一些，所以在 **Redis 7.0 已经将 ziplist 全部替换成了 listpack**。（关于 listpack 具体可以看第 8 篇）。

哈希表扩容、缩容策略

一般说到 Hash 类型就会谈到 hashtable，这时候容易引出它的扩容和缩容的问题。hashtable 编码的底层是 dict 结构体，关于这个问题我们放到第十篇里再谈。

记忆要点：应用场景，有多个属性的结构，内部编码，listpack，连续内存，紧凑结构，替换 ziplist，解决连锁更新，hashtable，512 entries，64 字节

5. 说说 List 数据类型

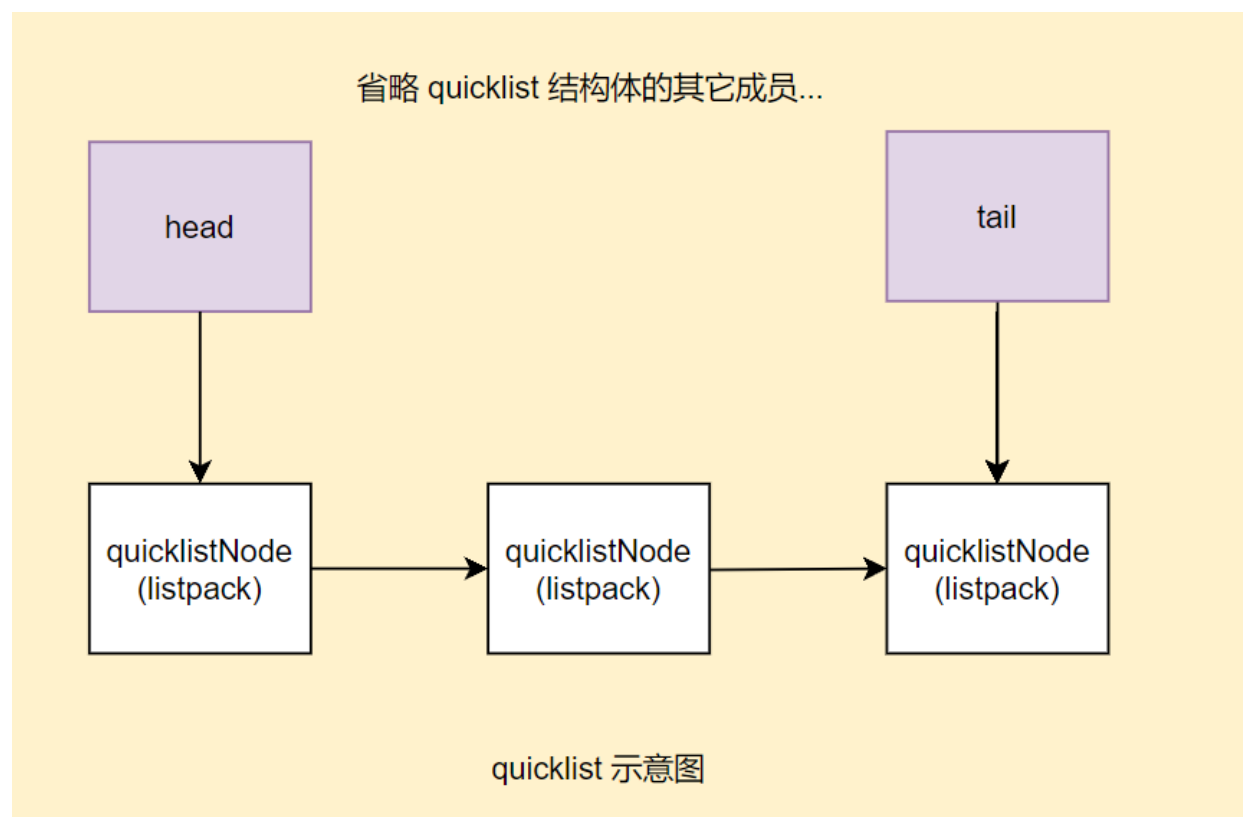
List 类型可以用来存储多个元素，一个列表最多可以存储 $2^{32} - 1$ 个元素。List 是一个双端列表，可以对两端进行插入和弹出操作，还支持获取指定范围，指定下标的元素等操作。它是一个比较灵活的数据结构，可以充当栈和队列的角色，并且它有两个主要的特点：**元素有序（按插入顺序排列）**，**列表元素可以重复**。List 类型可以从以下三个方面进行介绍：

应用场景

List 可以使用 `lpush + lpop` 实现一个后进先出的栈，也可以使用 `lpush + rpop` 实现一个先进先出的队列，还可以令生产者使用 `lpush`，消费者使用 `brpop` 实现一个简单的消息队列。List 很灵活，取决于怎么使用。

内部编码

Redis 3.2 之前，List 内部编码为 `ziplist`（压缩列表）和 `linkedlist`（链表）两种，而 3.2 至今为 **quicklist（快速列表）**。

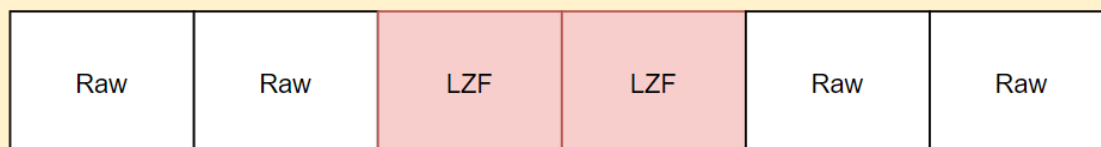


quicklist 实际上是将 **listpack（7.0 之前为 ziplist）** 和 **linkedlist** 结合起来，整体上看是 linkedlist，即链表结构，而链表中的每个节点是一个 listpack 结构（默认每个节点最多 8KB）。多个 listpack 结构通过前后指针连接在一起，它结合了紧凑列表和链表的优势，紧凑列表压缩了内存的使用量，而链表提高了查找效率（按索引查找时使用前后指针可以快速跳过节点，找到索引所在的紧凑列表后再从头或从尾部遍历查找获取指定元素）。

压缩节点

压缩节点是 List 的一种用于节省内存空间的技术，它的思想是 List 访问最频繁的是两端的节点，越靠内的节点越少被访问，所以我们可以将内部节点进行压缩存储，即局部性原理。这种方法会导致生成和访问压缩节点会拥有压缩与解压带来的开销，但节省了用于存储内部节点的内存空间，我们用压缩深度

(compress) 代表从两端起不会被压缩的节点数目，举个例子：compress = 2，List 有 6 个节点，那么从头部和尾部朝里的前 2 个节点不会被压缩（即有 4 个节点不会被压缩），而更里面的 2 个节点将会被压缩。



压缩节点示意图
(compress = 2)

压缩深度由 "list-compress-depth" 配置项指定，可以运行时客户端通过 `CONFIG SET list-compress-depth [value]` 指定，默认为 0（不进行压缩）。Redis 中使用的压缩技术名为 LZF，它的核心思想是对重复值进行压缩，通过哈希表来判断是否重复数据。

记忆要点：应用场景，栈，队列，消息队列，内部编码，quicklist，链表 + listpack，压缩内存，提高效率，压缩节点，局部性原理，保留外部压缩内部，节省内存

6. 说说 Set 数据类型

Set（集合）类型可以保存多个元素（member），Redis 支持集合内的增删改查，同时还支持多个集合取交集、并集、差集。Set 类型的特点是：**集合中不允许有重复元素，并且集合中的元素是无序的，不能通过索引下标获取元素，一个集合最多可以存储 $2^{32} - 1$ 个元素。** Set 类型可以通过以下三个方面进行介绍：

应用场景

Set 类型的应用场景很多，我们可以用 Set 来实现**标签功能**，比如记录一个文章的标签我们使用 Set 类型，向该集合内添加标签 id，之后获取该集合所有成员即可取出该文章关联的所有标签。除此之外，我们还可以使用 spop 和 srandmember 命令完成简单的**抽奖功能**。sinter 命令用于求交集，**我们可以使用求交集的功能完成“用户共同感兴趣的内容”之类的需求。** 并集和差集也可以用于完成一些其它的需求，集合功能很多，取决于用户怎么使用。

内部编码

intset（整数集合）：默认情况下，当集合中的元素全是 64 位范围内的整数，且元素数量 ≤ 512 时，采用整数集合作为内部编码。

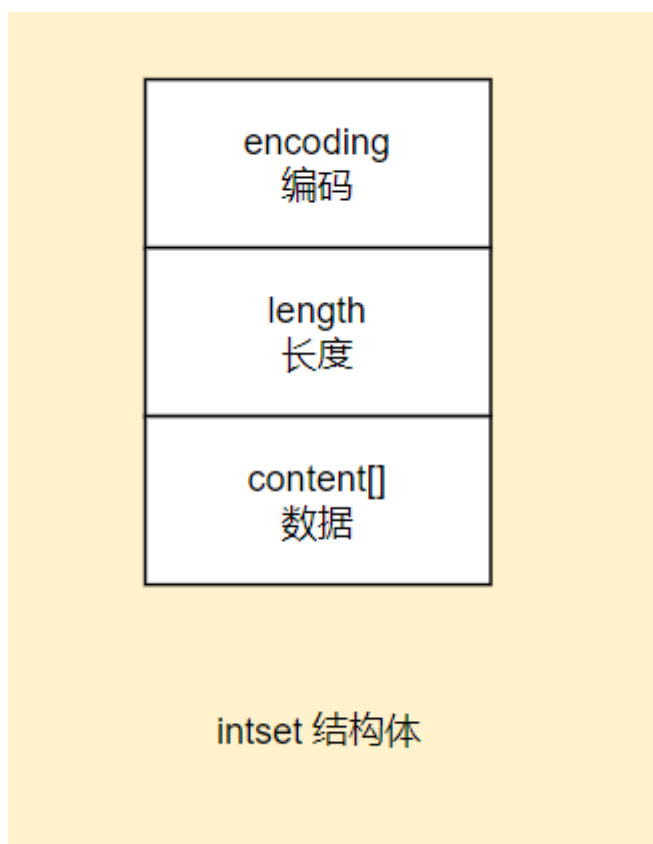
listpack (紧凑列表)：默认情况下，当集合中的元素数量 ≤ 128 ，每个元素大小 ≤ 64 字节，采用 listpack 作为内部编码。这个内部编码的 PR 是 **2022.11.10** 合并的，所以在较早的 Redis 7.0 版本 Set 类型是没有 listpack 的，若你只打算使用正式版本的 Redis，那么下一个正式版本 Redis 7.2 可以使用到 listpack 编码的 Set 类型。

hashtable (哈希表)：当不满足以上内部编码条件时，采用 hashtable 作为内部编码。

整数集合

intset (整数集合) 是 set 类型独有的数据结构，可以单独拿出来讲一讲。它其实是个简单的数据结构，我也尽量用最少的文字讲清楚它的基本原理。首先我们看一看 intset 结构体：

```
typedef struct intset {  
    /* 编码，记录整数集合底层数组(contents)的类型*/  
    uint32_t encoding;  
    /* 记录整数集合包含的元素个数 */  
    uint32_t length;  
    /* 整数集合的底层实现，虽声明为 int8_t 类型,但真正的类型取决于 encoding */  
    int8_t contents[];  
} intset;
```



很简单，整数集合结构体只有三个成员：**编码，长度，数据**。它的特点是可以减少不必要的内存分配和快速查找元素是否存在，缺点在于插入和删除性能较差。

如何减少不必要的内存分配：整数集合根据所存储的元素最大值确定编码，编码有三种：INTSET_ENC_INT16，INTSET_ENC_INT32，INTSET_ENC_INT64，它们分别对应 16 位，32 位，64 位整数，根据最大元素所占用来分配每个元素所占用的空间。比如当最大的元素只是 16 位范围内的整数时，那么整数集合为每个元素分配 2B 的空间来进行保存即可。同时在判断整数是否存在时，若要查询的整

数大于编码范围还可以快速判断元素不存在。

如何快速查找元素是否存在： 整数集合所保存的数据是**有序**的（从小到大），所以内部可以使用**二分查找**提升查询速度。

插入和删除性能较差： 插入元素时若该元素大于编码支持的最大值，则需要进行升级，选择一个可以容纳新元素的最小编码。升级时需要进行内存重分配并调整每个元素的偏移量，不过无论升不升级，插入和删除时都涉及 content 数组的内存分配（增加和减少），Redis 并没有为整数集合做预分配内存，原因应该是整数集合是元素数量较少时使用的内部编码，所以尽量节省内存空间。另外，为了保证有序，所以插入和删除时若操作的元素在中间，还需要调整之后元素的位置。所以插入和删除的**最差时间复杂度为 $O(n)$ 。

记忆要点： 应用场景，标签功能，抽奖功能，社交需求，内部编码，intset，64 位范围内整数，元素数量 ≤ 512 ，listpack，元素数量 ≤ 128 ，元素大小 ≤ 64 ，hashtable，整数集合，三个成员，编码，长度，数据，特点，省内存，查找快，插入删除慢

7. 说说 Zset 数据类型

Zset（有序集合）类型与 Set 类似，它保留了集合不能有重复元素的特点，不同在于通过给每个元素设置分数（score）可以将元素进行排序。Zset 提供了获取指定分数和元素范围查询、计算元素排名等功能。Zset 类型可以从以下三个方面进行介绍：

应用场景

Zset 最典型的一个应用是**排行榜**，比如我们使用 Zset 做一个视频点赞排行榜，score 负责存储点赞数，member 为视频 id，使用 `zrangebyscore` 命令来取出点赞数最高的部分视频。除了用 score 存储计数信息之外，我们一般还可以用来存储**时间戳**，以此返回最新的或者指定时间范围的信息。有的面试题就是怎么用 Redis 实现延时队列，关键就在于用分数存储时间戳，分数便代表了任务的执行时刻，然后让业务线程不断用 `zrangebyscore` 命令取出当前时间戳之前的所有元素，如果有元素则将它记录下来，并通过 `zrem` 命令从延时队列里删除。我们取出的元素便是已经可以执行的任务了，然后这个任务该干嘛就干嘛吧。

内部编码

listpack(紧凑列表)：默认情况下，当 Zset 的元素数量 ≤ 128 ，同时每个元素大小 ≤ 64 字节时，采用 listpack 作为 Zset 的内部编码。（7.0 之前是 ziplist）

skiplist(跳表)：不满足使用 listpack 的条件时，采用 skiplist 作为 Zset 的内部编码。

跳表

如果你被问到 Zset，那么很可能要被问一下跳表的实现了。不用太担心，跳表比起 B+ Tree，Radix-Tree 之类的树形存储结构算是简单的了，下面我会尽量用简单的方式与较少的文字说清楚跳表的基本实现原理。首先我们先来看看关于 skiplist 编码相关的结构体：

```
/* 跳表节点结构 */
typedef struct zskiplistNode {
    sds ele; /* 元素，字符串形式保存 */
    double score; /* 分数 */

```

```

struct zskiplistNode *backward; /* 后向指针，如果水平看链表则是左指针 */
struct zskiplistLevel {
    struct zskiplistNode *forward; /* 前向指针 */
    unsigned long span; /* 跨度 */
} level[]; /* 该节点拥有的跳表层级 */
} zskiplistNode;

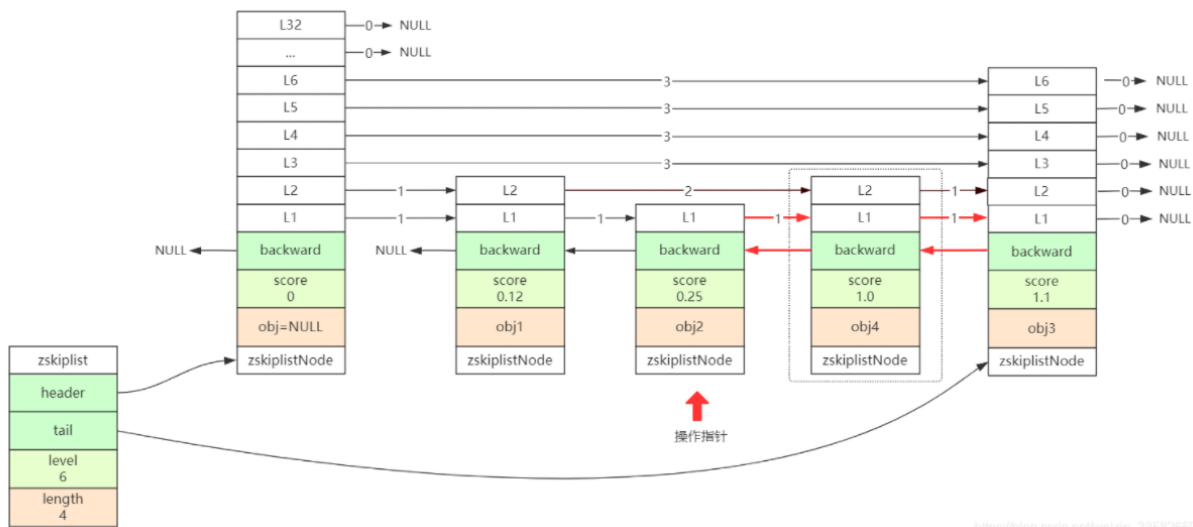
/* 跳表结构 */
typedef struct zskiplist {
    struct zskiplistNode *header, *tail; /* 头，尾节点 */
    unsigned long length; /* 长度，即元素数量 */
    int level; /* 最高层数 */
} zskiplist;

/* zset 结构，实际上是 skiplist 编码使用的结构 */
typedef struct zset {
    dict *dict; /* 字典 */
    zskiplist *zsl; /* 跳表，涉及分数查询和获取时使用 */
} zset;

```

可以看到实际上 skiplist 编码实际上是 **dict + skiplist**，而不是单纯的跳表，涉及分数的查询和获取时才使用跳表。不过我们主要讲跳表。接下来讲到原理时不会让大家盯着源码看，而是把源码翻译成文字和图片讲给大家。

跳表结构图有点难画，首先请允许我去网上找一张跳表结构图🤔：



这里每一块竖着的矩形都对应一个跳表节点，左下角则是 zskiplist 结构体。你会发现怎么有的节点这么多层，有的这么低呢？这就是跳表的精髓所在，接下来我们通过初始化，创建节点，查找节点三个部分来说明层级是怎么来的，以及跳表查找原理。

初始化：跳表初始化工作有两个：**初始化 zskiplist 结构体和头节点**，这里重要的是头节点不包含字符串数据（以后也不会），并且**拥有跳表所有层级 32 层**。

创建节点：创建跳表节点前，Redis 随机为该节点生成一个层数 level ($1 \leq \text{level} \leq 32$)，生成层数使用 while 循环，**每增加一层的概率为 25 %**，所以生成越高的层数概率也就越低。得到层数后即可进行节点初始化。

查找节点：查找时先从找到头节点并从最高层（第 32 层）开始，跳表查询是一个从上到下的过程。以上面的图查找 score 为 0.25 的第一个节点为例：

若当前节点层级的前向指针为 NULL 说明后面没有节点，跳到下一层，这里头结点 L32~L7 前向指针都是 NULL，则会一直下到 L6。

到了 L6 发现存在前向节点，但该节点 score(1.1) > 0.25，若前向节点的 score > 查询 score，跳到下一层，所以这里 L6 下降到 L5，而头结点的 L5 ~ L3 前向节点 score 均大于查询 score，则一直下到 L2。

到了 L2 后发现前向节点 score 为 0.12，若前向节点的 score <= 查询 score，跳转到前向节点，所以这里跳转到前向节点也就是第二个节点。这时第二个节点继续比较前向节点的 score，发现前向节点 score(1.0) > 0.25，则继续下到 L1。

到了 L1，继续比较前向节点 score，发现前向节点 score(0.25) <= 0.25，则跳到第三个节点。

到了第三个节点我们还是先走一遍和前向节点的比较流程，但是发现前向指针 score 更大，跳到下一层前其实会先判断一下当前节点 score 是不是等于查询 score，这时候我们发现当前节点 score(0.25) == 0.25，再判断 ele 是不是为 NULL，这是防止我们接下来返回的是头节点（因为 score 可以重复，若 score = 0 可能取到头结点）。发现不为 NULL，说明找到了对应元素，可以将它返回。（这个网上的图有点问题，元素数据不是 redis object 而是 sds 类型才对）

如果按 rank 查询，比如查询 score 从小到大的第 5 个节点是什么，则要记录当前节点已经跨过的节点数量 traversed，查找过程与指定 score 查询相同，但比较时变成 traversed + 前向节点 span 和 rank 相互比较。若大于 rank 向下，小于等于 rank 到前向节点。如果你不清楚什么是 span（跨度），span 是指在某层一个节点和它的后向节点之间实际上跨越了多少个元素，图中两个节点相互连接的直线中间有个数字，它就是 span。

看完查询过程，你应该明白了跳表层级的意义所在，就是**加速查询**。因为概率的原因，高于一层的节点不会太多，根据概率来算拥有二层的节点大约为 1/4。假如有 100 个节点，可能就有 25 个二层节点，理想情况下每 4 个节点能够拥有 1 个是 2 层节点，再往更高层看每层可能有零星几个节点，更高的层每个节点实际跨越过的节点数更多了，我们可以把每个节点到前向节点之间的跨度看作该节点所管理的范围，**所以跳表从上到下查询其实就是范围缩小这样一个加速查询的过程。**

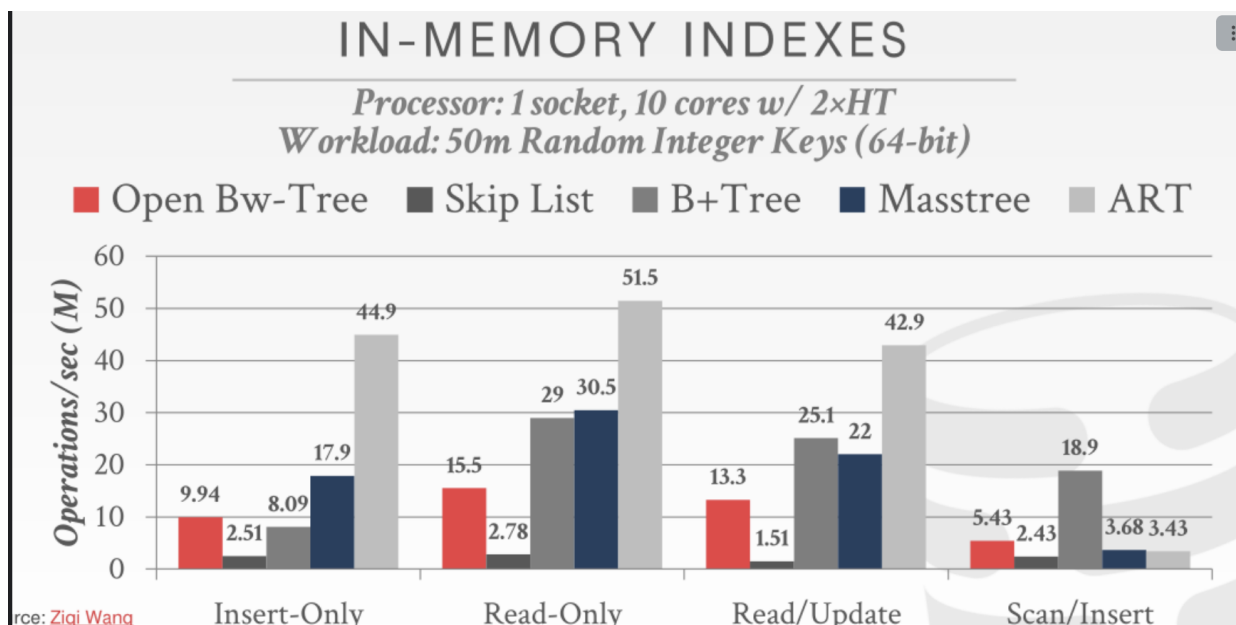
掌握了查询过程你就掌握了跳表的精髓，如果你还想要更加了解跳表，比如它的插入删除操作，那么你可以去找一篇详细的文章看看。但这是为了面试题准备的专题，要记的太多了反倒成了专门学习跳表技术的文章，个人认为掌握精髓足以，所以点到为止。

（如果面试官真的追问你插入删除了，其实也没什么难想的。类似单链表的插入删除，以插入来说，先查找到对应插入位置，然后创建节点后插入，插入就是遍历新节点的每一层，把插入位置前的节点的前向指针作为新节点前向指针，随后把插入位置前的节点的前向指针指向新节点，即成为新节点的后向节点，最后更新前向节点和新节点的跨度）

（附加题）为什么采用跳表而不用 B+ Tree 实现？

（对于这个问题我个人想法可能和网上不一样哈，大家自己动脑考虑，选择接受谁的说法）

网上结论可能会说在内存的话跳表写入比 B+ 树快。真的是这样吗？下面是一个在内存上的索引类数据结构的性能测试图（PPT 原链接：<https://15721.courses.cs.cmu.edu/spring2020/slides/07-oltpindexes2.pdf>）：



关注第二和第三条柱，即跳表和 B+ 树，无论是插入，还是读取，跳表都要比 B+ 树性能差。B+ 树查询比跳表快是显而易见的，B+ 树层数很少，定位一个数据跳转两三个节点一般就搞定了。但是跳表呢，它查询过程中需要跳转几次节点并不稳定，而且一般来说都要比 B+ 多，同时 cache miss（缓存未命中）也要比 B+ 树多得多。然后网上的跳表写入比 B+ 树快的结论是怎么来的呢？是因为跳表写入数据只需要修改少量节点（实际上是两个，当前节点和后向节点）就够了，而 B+ 树可能还会出现分裂，修改上层节点之类的现象。这些结论好像只考虑了到达了插入位置之后的操作，没考虑到寻找插入位置的过程。

个人认为最初采用跳表的原因是**跳表实现简单**，作者想写个性能尚可且容易实现的数据结构（大家不用在网上瞎猜作者有什么很高明的想法，Redis 在早期几乎就是它一个人捣鼓的项目）。而跳表为什么现在还不被换掉呢？这是因为跳表的性能其实已经足够了，而且 **Redis 是 IO 密集型的，它的瓶颈不在于 CPU，而是在于网络 IO**。所以跳表并没有成为 Redis 性能的瓶颈。

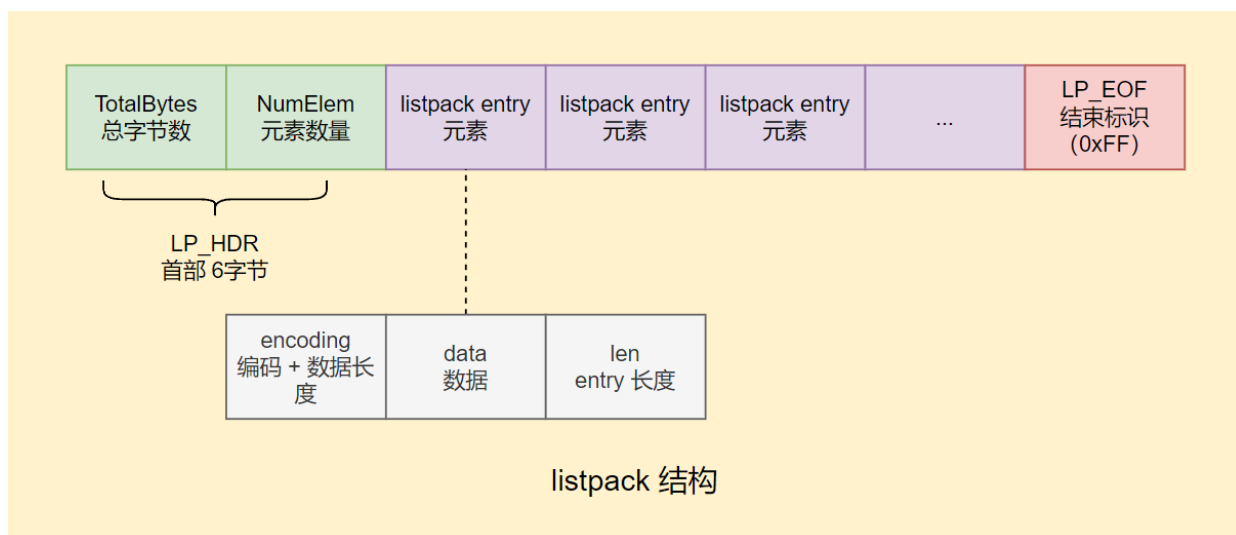
记忆要点：应用场景，排行榜，时间戳，内部编码，listpack，元素数量 ≤ 128 ，每个元素大小 $\leq 64B$ ，skiplist，dict + skiplist，跳表，初始化，头节点 32 层，创建节点，随机层数，新增一层概率 25%，查找节点，从头节点开始，从上到下缩小范围

8. 说说 listpack 结构

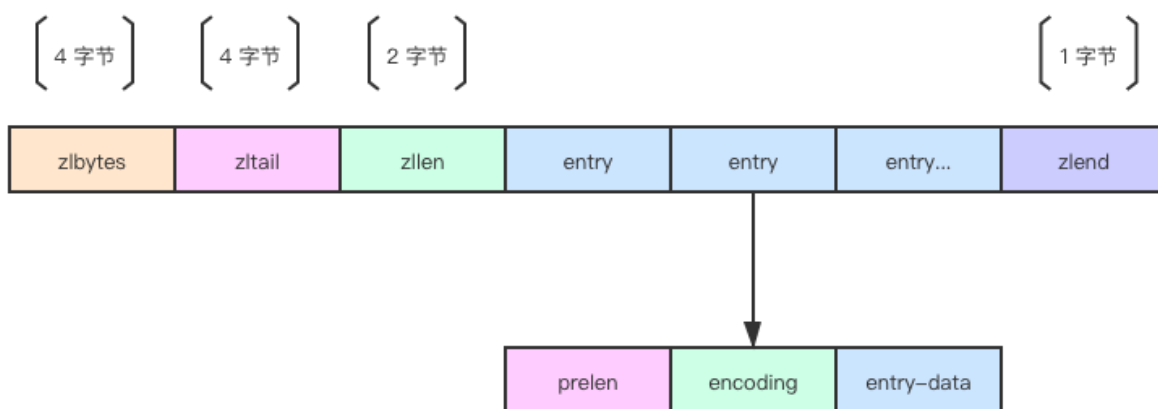
listpack 是 Redis 5.0 推出的数据结构，当时是作为新推出的 stream 数据类型的底层结构之一，到了 6.0 替换了 Hash 类型底层的 ziplist，之后在 7.0 版本中将 **ziplist 全部替换为 listpack**。因为它具有比 **ziplist 更少的内存以及没有“连锁更新”的缺陷**。接下来主要介绍 listpack 的**结构设计**，以及它和 ziplist 结构上的不同点。（注：我们不会讲太深的内容，不会涉及太多太难的源码，点到为止）

listpack 结构设计

首先看一看 listpack 结构图：



先来看首尾，listpack 首部有两个属性，记录的是总字节数与元素数量，尾部是使用 0xFF 作为结束标识符。接下来，让我从网上找个 ziplist 结构图下来对比一下它们俩的区别：



可以看到 ziplist 首部有三个属性，它们记录的是总字节数，尾节点（最后一个 entry）偏移量，元素数量。它的尾部其实仍然是 0xFF 作为结束标识符。可以发现 listpack 和 ziplist 首部存在区别，即 listpack 首部少了尾节点偏移量。listpack 不需要记录尾节点偏移量也能定位到尾节点，这是怎么做到的呢？

我们看一下 entry 的内部结构，答案就在它之中。listpack 的 entry 有三个属性：编码类型（+数据长度），数据，entry 长度。ziplist 的 entry 也有三个属性：上个 entry 长度，编码类型（+数据长度），数据，可以发现它俩 entry 的区别在于记录的长度，listpack entry 记录的是自己的长度（encoding + data，不包括 len），而 ziplist entry 记录的是上一个 entry 的长度。

listpack 如何定位尾节点：因为 listpack entry 将 len 排到了末尾，这就可以让 listpack 快速定位尾节点起始地址。获取尾节点首地址可以先通过 listpack 首地址 + TotalBytes - 1 得到结束标识符首地址，同时也是尾节点的下一个字节处。这时上一个字节即是尾节点长度数据，我们让指针每次倒退一个字节，直到找到字节内容 & 128 = 0 的位置（长度编码是令第一个字节的最高位为0，其余字节最高位为 1），就找到了尾节点长度的首地址，并且倒退在过程中我们记录每个字节的数据并做偏移计算，到达长度首地址时也就顺便获取了 entry 长度，此时再用指针（此时位置在长度的首地址）- len 即可得到 entry 的首地址。思想是这样，Redis 实现是指针 p 先到 len 最后一个字节的位置，然后调用计算长度的函数，计算得出 entry 长度 prevlen 后，再获取编码这个长度需要的字节数并累加到 prevlen 上，此时令 p -= prevlen - 1 即可得到 entry 首地址。除了定位尾节点之外，查找上一个节点的首地址也是同样的方法。

listpack 如何解决连锁更新：由于 ziplist entry 记录的是上一个 entry 的长度，所以会触发连锁更新。当 ziplist entry 上一个节点大小 < 254 字节时用 1 字节记录长度，>= 254 时用 5 字节记录长度。那么如果出现一种极端情况，连续好多个 entry 大小都为 253 字节，那么前面第一个 entry 因为修改超过了 254 字节时，prelen 也会扩大成 5 字节，随后后面的 entry 也会因为前面的大小超过 254 字节而扩大自己的 prelen，这就是“连锁更新”。而 listpack 因为只记录自己的长度，所以更新不会影响其它节点，也就解决了“连锁更新”问题。

有的人可能会点开 Redis 源码的 listpack.h 文件看到以下 listpackEntry 结构体认为这就是 entry 结构，有的网络文章也展出了这个结构体。这其实并不是 listpack 内部 entry 的样子，只是给外部用的哈，目前一般用在 hash 和 zset 获取随机元素的操作中。

```
typedef struct {
    unsigned char *sval;
    uint32_t slen;
    long long lval;
} listpackEntry;
```

因为我们不是专门讲 Redis 深层原理的文章，关于 listpack entry 的 encoding 类型就不在讨论范围之内了，况且这个类型多也不好记忆。我们只要知道 **encoding 存储数据编码以及数据长度** 即可。（有的网络文章说什么 ziplist 的 encoding 压缩效果更好更省内存，其实 listpack 和 ziplist 的 encoding 除了有些编码差异之外，压缩效果都是差不多的。不过我们不打算讲 ziplist，因为它已经被淘汰了，但面试可能还是会问，大家最好还是自己上网搜索相关文章进行学习）

简谈 listpack 的插入删除：listpack 只支持两端插入和删除，而插入和删除都可能涉及内存分配。插入新元素如果造成 listpack 大小 > 已经为 listpack 分配的内存大小，则会进行内存分配；而删除元素后如果造成 listpack 大小 < 原 listpack 大小，则会进行缩容。

记忆要点：Redis 5.0 出现，Redis 7.0 替换所有 ziplist，listpack 首部，总字节数，元素数量，0xFF 结束标识，listpack entry，编码类型，数据，当前节点长度（编码类型+数据所占字节数），如何定位尾节点或向前查找，指针倒退，entry 长度排在末尾，指针 - 长度，如何解决连锁更新，只保存当前 entry 长度，插入删除，只支持两端插入和删除，插入删除可能涉及内存分配

9. 说说 SDS 结构

Redis 使用 SDS（Simple Dynamic String，简单动态字符串）对字符串进行一个封装。接下来介绍为什么要有 SDS 和 SDS 的底层是什么样的。

为什么要有 SDS

SDS 的出现是为了解决 C 字符串在 Redis 三高 场景下的不足，分别有 **安全性、效率以及功能** 等方面。

Redis 是 C 语言，C 语言如果要存储字符串内容的形式是 char[] 或 char*。如果只存储字符串内容，那么如果要获取字符串长度的时候，需要遍历一遍字符串找到结束符，效率很低，不适合 Redis 这样用于高并发场景的数据库。同时，这样的字符串也不易于进行动态扩展，因为没有记录它被分配的内存是多少。此外，不记录长度和分配内存空间的话，扩展字符串也容易造成缓冲区溢出。而 SDS 通过空间预分配和惰性空间释放策略减少了修改字符串带来的内存重分配次数。

SDS 还是二进制安全的。二进制安全指的是 SDS API 都是以处理二进制的方式来处理字符串内容的，程序不会对其中的数据做任何限制、过滤、或者假设，数据在写入时是什么样的，它被读取时就是什么样。举例来说就是，如果字符串内容出现了和字符串结尾标识相同的“\0”字符，若以字符串形式处理则内容会被截断，这就出现了不安全的问题。通过使用二进制安全的 SDS，而不是 C 字符串，使得 Redis 不仅可以保存文本数据，还可以保存任意格式的二进制数据。

虽然 SDS 的 API 都是二进制安全的，但它们一样遵循 C 字符串以空字符结尾的惯例：这些 API 总会将 SDS 保存的数据的末尾设置为空字符，并且总会在为 buf 数组分配空间时多分配一个字节来容纳这个空字符，这是为了让那些保存文本数据的 SDS 可以**重用一部分 <string.h> 库定义的函数**。

总结 SDS 相比 C 字符串的特点如下：

- 常数复杂度获取字符串长度。
- 杜绝缓冲区溢出。
- 减少修改字符串长度时所需的内存重分配次数。
- 二进制安全。
- 兼容部分 C 字符串函数。

SDS 底层结构

SDS 是对 C 字符串的封装，重点在于提供了个**首部 (sdshdr)**，首部是用来记录字符串必要的元信息的。废话不多说，先上结构体：

```
/* sdshdr5 实际未被使用 */
struct __attribute__((__packed__)) sdshdr5 {
    unsigned char flags; /* 标志位，3 个最低有效位表示类型，同时 5 个最高有效位表示字符串长度 */
    char buf[];
};

struct __attribute__((__packed__)) sdshdr8 {
    uint8_t len; /* 已使用的长度 */
    uint8_t alloc; /* buf 分配内存空间，不包含空终止符号(\0) */
    unsigned char flags; /* 标志位，3位最低有效位表示类型，其余5个比特位未被使用 */
    char buf[];
};

struct __attribute__((__packed__)) sdshdr16 {
    uint16_t len; /* 已使用的长度 */
    uint16_t alloc; /* buf 分配内存空间，不包含空终止符号(\0) */
    unsigned char flags; /* 标志位，3位最低有效位表示类型，其余5个比特位未被使用 */
    char buf[];
};

struct __attribute__((__packed__)) sdshdr32 {
    uint32_t len; /* 已使用的长度 */
    uint32_t alloc; /* buf 分配内存空间，不包含空终止符号(\0) */
    unsigned char flags; /* 标志位，3位最低有效位表示类型，其余5个比特位未被使用 */
    char buf[];
};

struct __attribute__((__packed__)) sdshdr64 {
    uint64_t len; /* 已使用的长度 */
    uint64_t alloc; /* buf 分配内存空间，不包含空终止符号(\0) */
```



```
unsigned char flags; /* 标志位, 3位最低有效位表示类型, 其余5个比特位未被使用 */
char buf[];
};
```

可以看到有很多种类 sdshdr, sdshdr5 不使用就不进行讨论了, 看 **sdshdr8 ~ sdshdr64**, 它们都有四个结构体成员: **len (长度)**, **alloc (分配内存空间)**, **flags (标志位)**, **buf (字节数组)**。其中有三个元信息: 字符串长度, 字节数组分配空间 (不包括最后 1 字节的空终止符), sdshdr 类型。最后一个 buf[] 字节数组是用来存储字符串内容的。

我们再来关注下 **attribute ((packed))** 修饰, 这个修饰会取消编译器的优化对齐。一般情况下编译器会以所有结构体成员的最大公倍数进行对齐, 用 **packed 修饰后会变成按 1 字节对齐**, 那么这样做一个是**节省内存**, 第二个是可以**保证 buf[] 始终在 flags 的下一个字节位置**, 通过 **buf[-1]** 就可以取到首部的 flags, 进而可以通过 flags 判断首部类型, 知道了首部类型就能够知道首部其它成员大小, 从而获取其它结构体成员。第三个是因为 buf[-1] 就能找到首部, Redis 便可以将指向 buf[] 的指针暴露给上层函数。

为什么有那么多种 sdshdr: 我们可以发现每种结构体的区别在于 **len 和 alloc 的类型**, sdshdrxx 就用 xx 位的 len 和 alloc。这样做的目的是**节省首部空间**, 如果字符串很短, 8 位整数就能表示长度, 那么就用 sdshdr8 就够了, 这样 len 和 alloc 每个仅占 1 字节。单看节省几个字节没多大用处, 但是当 Redis 存储的字符串数据多了, 从首部节省下来的内存就很明显了, 毕竟在 Redis 中存储使用得最多的一般都是 SDS 类型。

SDS 底层操作

接下来我们简单谈一谈 SDS 的三个基本操作: 创建 (new), 释放 (free), 拼接 (cat)。不涉及源码, 只用文字讲原理。

创建: 创建 SDS 类型需要传入字符串以及字符串长度, 首先根据字符串长度确定首部类型 (不包括 sdshdr5), 计算并分配需要的内存空间 (首部长度 + 字符串长度 + 结束符), 最后就是为结构体成员变量赋值, 以及将字符串复制到 buf 数组。

释放: SDS 类型对外提供了两种释放字符串的方法, 一种是直接通过 free 释放内存; 另一种是重置 len = 0 以及让 buf[0] = '\0' (结束符), 不释放内存, 这样将来可以覆盖重写, 是一种优化性能的手段, 严格来说这个不算是释放字符串, 而是清空字符串, 但是我们放在这里一起讲。

拼接: 拼接指的是讲一个新的字符串拼接到一个 SDS 的字符串内容之后。首先需要检查 SDS 已经分配的内存空间是否能够容纳新字符串, 装不下则需要进行扩容, 如果拼接后的长度大于首部支持的最大长度还需要更换首部。最后就是将新字符串复制到原字符串之后 (复制起点为原字符串结束符处, 把原结束符覆盖掉), 然后新的末尾处设上 '\0' 结束标识符。

如果你想说还有更新操作啊~想知道字符串是怎么更新的, 比如 setrange 命令修改字符串的部分内容。很简单, 检查是否要扩容 + 内容复制, 与拼接原理类似。但是要注意 set 命令并不会在原字符串上修改内容, 而是替换了一个新的 SDS。

SDS 预分配空间和惰性释放空闲空间策略

预分配空间策略: SDS 预分配空间指的是为 buf[] 末端预先分配一些暂未使用的空间, 当将来字符串内容变长时可以减少内存分配。预分配空间策略为: **新增后长度 < 1MB**, 则按新长度的两倍分配; **新增后长度 >= 1MB**, 则按新长度 + 1MB 进行分配。

惰性释放空闲空间策略：惰性释放空闲空间这里指的是**释放 buf[] 尾部空闲空间**。惰性释放空闲空间指的是 `sdsRemoveFreeSpace` 函数，它会在 Redis 的一些 SDS 类型缓冲区空闲过久或过多时触发，也可以由客户端触发，由客户端触发主要指解析 String 类型命令时将输入的字符串参数空闲空间释放，这里的**释放条件是空闲空间大于总空间的 10 %**。在释放时，计算释放后的长度适配的 SDS 首部，原来的首部称为 `oldtype`，计算出来的适配首部称为 `type`，那么 `oldtype==type || type > SDS_TYPE_8` 条件满足的话，使用 `realloc` 重新分配已有的空间，否则使用 `malloc` 分配空间，并且替换成 `sdsHDR8` 首部的 SDS，释放旧 SDS。

(参考资料：redis之SDS字符串，到底高效在哪里？（全面分析）<https://zhuanlan.zhihu.com/p/594786067>)

记忆要点：为什么要有 SDS，常数复杂度获取字符串长度，杜绝缓冲区溢出，减少修改字符串长度时所需的内存重分配次数，二进制安全，兼容部分 C 字符串函数，SDS 底层结构，SDS 首部，四个结构体成员，`len`（长度），`alloc`（buf 分配空间），`flag`（首部类型标志），`buf[]`（存储字符串），有多种首部原因，多种首部 `len` 和 `alloc` 类型不同，节省首部占用内存，SDS 底层操作，创建，释放，拼接，预分配空间策略，新增后长度 < 1MB 双倍分配，新增后长度 >= 1MB 额外分配 1MB，惰性释放空闲空间策略

10. 说说 dict 结构

dict 结构其实是一个采用链地址法解决哈希冲突的哈希表，我们可以从它的**结构设计，渐进式扩容与缩容策略**进行介绍。

dict 结构设计

首先看一看 dict 结构体：

```
struct dict {
    dictType *type; /* 字典类型，8 bytes */
    dictEntry **ht_table[2]; /* 哈希表数组 */

    unsigned long ht_used[2]; /* 哈希表拥有键值对数量 */

    long rehashidx; /* rehash 进度/下标 */

    /* 将小尺寸的变量置于结构体的尾部，减少对齐产生的额外空间开销。 */

    int16_t pauserehash; /* rehash 是否暂停， > 0 表示暂停 */
    signed char ht_size_exp[2]; /* 哈希表大小的指数表示，即哈希表大小 = 2 ** exp */
};
```

dict 每个成员变量都做了简要注释，我们应该注意到哈希表与哈希表相关的统计信息变量 `ht_table[2]`，`ht_used[2]`，`ht_size_exp[2]`，它们都是长度为 2 的数组。这表示 Redis 其实维护了两个哈希表。平时使用的是下标为 0 的哈希表，当触发扩容或缩容而进行 rehash（重哈希）时，会将下标为 0 的哈希表作为旧哈希表，下标为 1 的哈希表作为新哈希表。旧哈希表的键值对会逐渐迁移至新哈希表，当旧哈希表的键值对数量为 0 时则说明迁移结束，此时将下标为 0 的哈希表释放，并把新哈希表变为下标 0。

接下来我们继续看 dictEntry 结构体：

```
typedef struct dictEntry {
    /* void * 类型的 key，可以指向任意类型的键 */
    void *key;
    /* 联合体 v 中包含了指向实际值的指针 *val、无符号的 64 位整数、有符号的 64 位整数，以及 double 双精度浮点数。
     * 这是一种节省内存的方式，因为当值为整数或者双精度浮点数时，由于它们本身就是 64 位的，void *val 指针也是占用 64 位（64 操作系统下），
     * 所以它们可以直接存在键值对的结构体中，避免再使用一个指针，从而节省内存开销（8 个字节）
     * 当然也可以是 void *，存储任何类型的数据，最早 redis1.0 版本就只是 void* */
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    } v;
    struct dictEntry *next; /* 同一个 hash 桶中的下一个条目 */
    void *metadata[]; /* 元信息 */
} dictEntry;
```

dictEntry 的成员变量有键和值，下一个 entry 的指针，元信息（目前只在集群有使用，存储双向指针用于将属于同个 slot 的 entry 链接起来）。可以看到存储值有个优化手段，它用了 union，**union 里的属性共用同一块 8 字节空间**。当值可以用整数或者双精度浮点数表示时，可以直接存在该结构体中，避免再使用指针寻址，即提高了性能也节省了内存。

另外，Redis 7.0 对 dict 结构体是有优化的，可以看看之前的 dict 结构：

```
// 7.0 版本之前的字典结构
typedef struct dictht {
    dictEntry **table; // 8 bytes
    unsigned long size; // 8 bytes
    unsigned long sizemask; // 8 bytes
    unsigned long used; // 8 bytes
} dictht;

typedef struct dict {
    dictType *type; // 8 bytes
    void *privdata; // 8 bytes
    dictht ht[2]; // 32 bytes * 2 = 64 bytes
    long rehashidx; // 8 bytes
    int16_t pauserehash; // 2 bytes
} dict;
```

优化如下：

1. 从字典结构里删除 privdata。
2. 将 dictht 融合进 dict 结构体里，相关元数据直接放到了 dict 中。
3. 去掉 sizemark 字段（计算 hash 值使用），这个值可以通过 size - 1 计算得到（&(size - 1)其实就是 % size），这样就可以少 8 字节。

4. 将 size 字段转变为 size_exp（就是 2 的 n 次方，指数），因为 size 目前是严格都是 2 的幂，size 内存占用从 8 字节降到了 1 字节。

内存方面优化：

默认情况下通过 sizeof 我们是可以看到新的 dict 是 **56 个字节**。

dict：一个指针（64 位，8 字节）+ 两个指针 + 两个 unsigned long + 一个 long + 一个 int16_t + 两个 char，总共实际上是 52 个字节，但是因为 jemalloc 内存分配机制，实际会分配 56 个字节，而实际上因为对齐，最后的 int16_t pauserehash 和 char ht_size_exp[2] 加起来是占用 8 个字节，代码注释也有说，将小变量放到最后来获得最小的填充。

渐进式扩容与缩容策略

扩容时机：触发扩容的前提是当前 Redis 没有子进程存在（需要持久化数据到硬盘时会创建子进程），因为写时复制的原因，父进程在此时执行扩容会对内存进行写操作，此时性能会很差（先在内存中复制一份该数据所在页后再写入）。扩容主要条件是当 Hash 的内部编码为 hashtable 且键值对数量大于负载因子（键值对数量与 hashtable 中桶数量的比值）时，需要进行扩容，哈希表的大小为 2 的幂，每次扩容把指数 +1 即加倍扩容，默认负载因子为 1。

hashtable 的扩容方式采用**渐进式扩容**。在 Redis 源码里 dict 为哈希表结构体，其中存放键值对的结构体成员为 `dictEntry **ht_table[2]`，它是两个数组，即两个哈希表，与哈希表相关的统计信息也是大小为 2 的数组。平时使用的是下标为 0 的哈希表，当触发扩容时，下标为 0 的哈希表作为旧哈希表，而下标为 1 的作为新哈希表，为新哈希表分配旧哈希表双倍大小的空间，dict 结构体的 rehashidx 置为 0，rehashidx 作为 rehash 时需要访问旧哈希表数组的下标。

Rehash 时机：首先想要触发 Rehash 前提是当前 Redis 没有子进程存在，原因已经在扩容时机中说过，满足这个前提条件后 Rehash 时机有两个，第一个是在对 Hash 进行更新和查找时若 `rehashidx != -1` 则表示需要 rehash，那么首先会进行 rehash，每次 rehash 访问旧哈希表的 rehashidx 位置，每次访问过下标元素后 `rehashidx + 1`，若没有找到键值对则最多连续访问 10 个空桶（数组下标元素为空），若找到存在的键值对则将它迁移到新哈希表，并把 `ht_used[0] - 1`（旧哈希表键值对数量 - 1），`ht_used[1] + 1`（新哈希表键值对数量 + 1）。将该下标对应链表的键值对全部访问完后令 `rehashidx + 1`，结束本轮 rehash，下次 rehash 从 rehashidx 位置继续；第二个是 Redis 在空闲时（没有子进程存在）大约每隔 100 ms 用 1ms 的时间来进行 rehash。

当某次 rehash 执行完毕，发现 `ht_used[0] == 0`（旧哈希表键值对数量为 0）时，表明 rehash 已完成，最后进行收尾工作：释放旧哈希表空间，将新哈希表以及新哈希表的统计信息赋值给同数组中下标为 0 的元素，重置 `ht_table[1]`（令 `ht_table[1] = NULL` 以及重置下标为 1 的统计信息），`rehashidx = -1`。

缩容时机：触发缩容的前提依然是当前 Redis 没有子进程存在，满足前提后当**负载因子为 0.1**时，哈希表进行缩容。缩容与扩容调用的是同一个函数，过程与扩容类似，不同在于新哈希表会是旧哈希表的 1/2 大小，同样也是采用渐进式 rehash 最终完成缩容。

一直有持久化子进程存在是不是就不会扩容了？

答案是不会，dict 结构有一个强制扩容负载因子 `static unsigned int dict_force_resize_ratio = 5`，即负载因子到达 5 后会强制进行扩容。

记忆要点：dict 结构，链地址法哈希表，维护两个哈希表，dictEntry 结构，union 优化，整数和双精度浮点数，节省内存和提高性能，Redis 7.0 dict 优化，dict 结构内存占用减少，渐进式扩容和缩容策略，扩容时机，负载因子为 1，渐进式 rehash，rehash 时机，查找和更新，空闲时 rehash 1ms，缩容时机，负载因子为 0.1，触发前提，不存在子进程

11. 用 stream 做消息队列行不行？

如果在介绍数据类型时，你讲出了 Redis 的 stream（消息队列）类型，估计会有面试官好奇问你一下，用这东西做消息队列行不行？靠不靠谱？这时候你自己挑出来的招得接住啊，为了能够让你成功装逼~ 我们接下来就分析一下用 stream 做消息队列行不行，它和专业消息队列有什么区别？

（我并没怎么使用过 stream 类型，以下内容大部分参考自：<https://blog.csdn.net/z2431435/article/details/124978166>，我负责为该篇介绍 stream 的文章做个提炼。如果作者看到了，可以联系商量下授权费，邮箱 eriri233@qq.com，请带上你是博主的证明发过来，否则不会回复，sorry~）

说到 stream 之前，我们先来讨论在它之前推出的 pub/sub 能否用来做消息队列。

pub/sub 能不能做消息队列

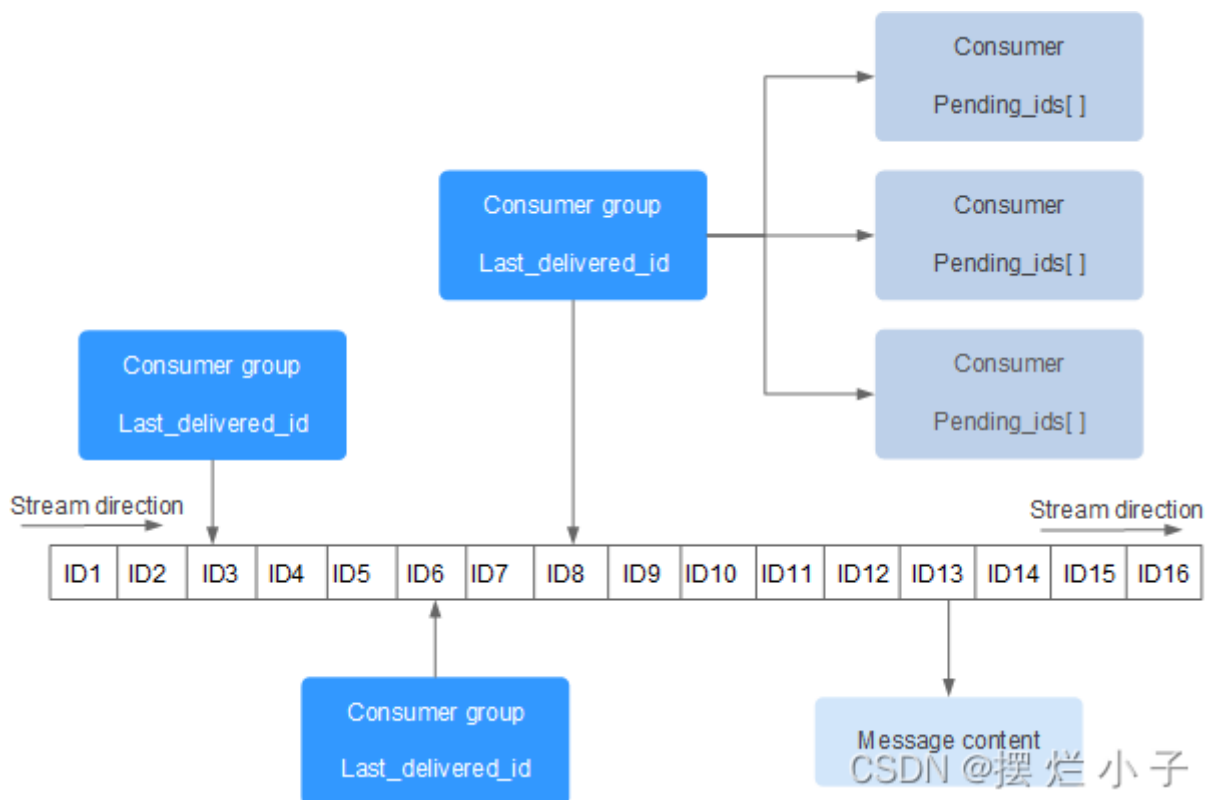
首先说答案：不能。原因有以下三点：

1. 消费者下线，数据会丢失。
2. 不支持数据持久化，Redis 宕机或重启，数据会丢失。
3. 若消息堆积过多，消费者未能及时消费，会造成缓冲区大小超过限制，消费者会被强制踢下线。

说白了，pub/sub 只是一个消息转发用的。它可以在实现 Redis 分布式锁时使用，比如让每个没抢到同把锁的线程监听同个 pub/sub，当占用锁的线程解锁后在 pub/sub 发出解锁消息，用来通知其他线程可以抢占锁了。关于 Redis 分布式锁的实现我们会在后面专门介绍。

stream 特性

在说到标题的问题前，我们还得先了解一下 stream 的特性。



Redis Stream 提供了**消息的持久化和主备复制功能**，可以让任何客户端访问任何时刻的数据，并且能记住**每一个客户端的访问位置**。

Redis Stream 有一个消息链表，将所有加入的消息都串起来，每个消息都有一个唯一的 ID 和对应的内容。每个 Stream 都有唯一的名称，它就是 Redis 的 key，在我们首次使用 xadd 指令追加消息时自动创建。

Consumer Group：消费组，使用 XGROUP CREATE 命令创建，一个消费组有多个消费者 (Consumer)。

last_delivered_id：游标，每个消费组会有个游标 last_delivered_id，任意一个消费者读取了消息都会使游标 last_delivered_id 往前移动。

pending_ids：消费者(Consumer)的状态变量，作用是维护消费者的未确认的 id。pending_ids 记录了当前已经被客户端读取的消息，但是还没有 ack (Acknowledge character：确认字符)。

stream 做消息队列行不行：

使用消息队列有两个需要考虑的问题：**消息丢失，消息可堆积**。

对于消息丢失问题，有三个方面：**生产者会不会丢消息，消息队列会不会丢消息，队列中间件会不会丢消息？**

生产者会不会丢消息？

无论是 Redis 还是专业的队列中间件，生产者都是可以保证消息不丢的。因为大不了重发。

消费者会不会丢消息？

消费者拿到消息后，还没处理完成，就异常宕机了，那消费者还能否重新消费失败的消息？

消费者在处理完消息后，必须「告知」队列中间件，队列中间件才会把标记已处理，否则仍旧把这些数据发给消费者。这也被称作“消费位移提交”，这种方案需要消费者和中间件互相配合，才能保证消费者这一侧的消息不丢。无论是 Redis 的 Stream，还是专业的队列中间件，例如 RabbitMQ、Kafka，其实都是这么做的。

队列中间件会不会丢消息？

Redis 在以下 2 个场景下，都会导致数据丢失：

1. AOF 持久化配置为每秒写盘，但这个写盘过程是异步的，Redis 宕机时会存在数据丢失的可能。
2. 主从复制也是异步的，主从切换时，也存在丢失数据的可能（从库还未同步完成主库发来的数据，就被提成主库）

基于以上原因我们可以看到，**Redis 本身无法保证严格的数据完整性。**

而专业的消息队列会怎样呢？其实面对第一点，在没将消息刷盘前就宕机任何消息队列都是无解的。那么对于第二点呢？像 RabbitMQ 或 Kafka 这类专业的队列中间件，在使用时，一般是部署一个集群，生产者在发布消息时，队列中间件通常会写「多个节点」，以此保证消息的完整性。这样一来，即便其中一个节点挂了，也能保证集群的数据不丢失。

那么消息能否堆积？

由于 Redis 是内存型的数据库，如果设置了内存淘汰策略会淘汰 key，这就意味着当消息堆积过多时消息可能会被删除，从而丢失消息。

而专业的消息队列比如 Kafka，RabbitMQ 它们都会将数据持久化到硬盘上，磁盘的成本要比内存小得多，当消息积压时，无非就是多占用一些磁盘空间，相比于内存，在面对积压时也会更加轻松。

从以上分析看来 stream 好像有不少问题，那么它真的不该用来当消息队列吗？也不能说它一定没法用，要根据应用场景来看：

Redis Stream 相比于 Kafka、RabbitMQ，部署和运维更加轻量。如果你的业务场景足够简单，对于数据丢失不敏感，而且消息积压概率比较小的情况下，把 Redis Stream 当作队列是完全可以的。

如果你的业务场景对于数据丢失非常敏感，而且写入量非常大，消息积压时会占用很多的机器资源，那么我建议你使用专业的消息队列中间件。

简谈 Stream 底层结构

可能当你说完以上内容之后，又被追问了个底层结构。。。那也是有点倒霉。我们来简单了解 Stream 底层结构保个底吧，不用担心会讲太多，100 字左右~

Stream 底层结构用到了两个数据结构，一个是 **listpack（紧凑列表）**，一个是 **rax 树（radix-Tree）**。**listpack 用来存储消息**，**rax 树是一个可以根据前缀来索引的树形索引结构（类似字典树，但是有压缩路径的优化）**，在 Stream 里不止用了一颗 rax 树，**为消息，消费者组，消费者，等待消费者确认的消息都建了 rax 树来索引。**

如果你对 rax 树好奇且有时间研究源码，可以看我们 Redis 7 中文注释仓库中的 rax.c 文件：<https://github.com/CN-annotation-team/redis7.0-chinese-annotated/blob/7.0-cn-annotated/src/rax.c>

记忆要点：pub/sub 不能做消息队列，消费者下线数据丢失，不能持久化，消息堆积超过限制踢消费者下线，Stream 特性，持久化，主从复制，消费任意时刻消息，记录消费位置，消费组，Stream 做消息队列行不行，考虑消息丢失和消息可堆积问题，Stream 丢消息场景，刷盘前宕机，主从同步前主节点宕机，内存淘汰导致无法堆积过多消息，适用场景，业务简单，数据丢失不敏感，消息积压概率较小，Stream 底层结构，listpack + rax 树

12. Redis 7 内存淘汰策略

虽然 Redis 内存淘汰策略在 7.0 没什么更新，但这也是一个常考问题，这里还是要提到一下：

Redis 中提供了 **8 种内存淘汰策略**：

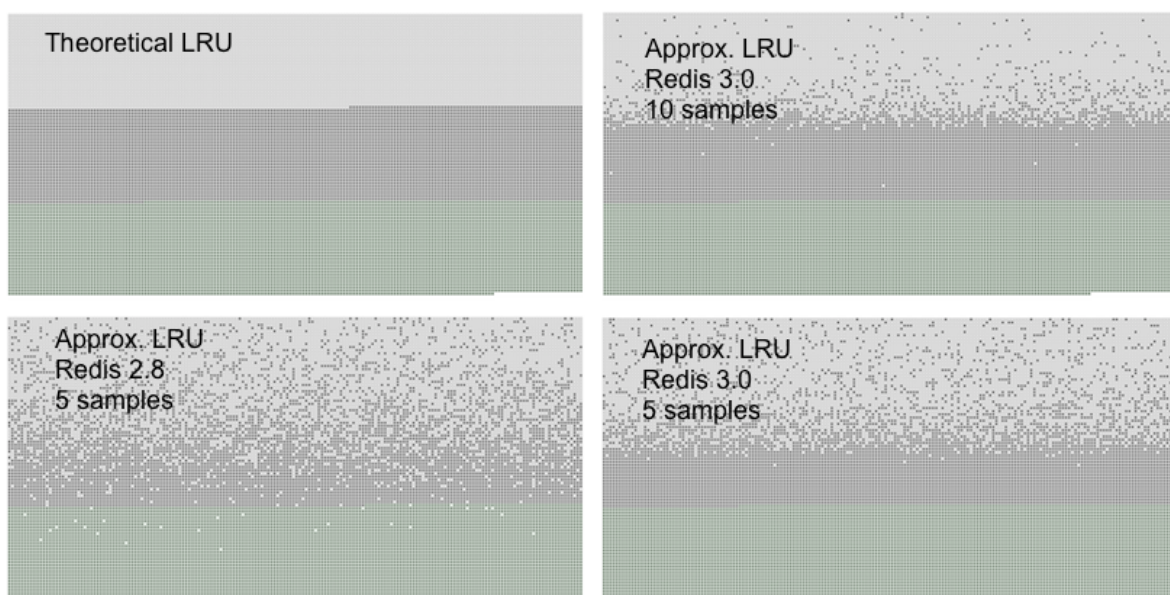
- volatile-lru：针对设置了过期时间的 key，使用 LRU 算法进行淘汰
- allkeys-lru：针对所有 key 使用 LRU 算法进行淘汰
- volatile-lfu：针对设置了过期时间的 key，使用 LFU 算法进行淘汰
- allkeys-lfu：针对所有 key 使用 LFU 算法进行淘汰
- volatile-random：从设置了过期时间的 key 中随机删除
- allkeys-random：从所有 key 中随机删除
- volatile-ttl：删除最近过期的 key
- noeviction（默认策略）：不删除 key，返回 OOM 错误，只能读取不能写入

近似 LRU 算法

Redis 使用的 LRU 算法是一个**近似的 LRU 算法**，这意味着 Redis 无法保证选择最久未访问的 key 进行淘汰。近似 LRU 算法是对少量 key 进行采样（默认为 5 个），将采样 key 放入一个淘汰池中，淘汰池默认最多存储 16 个 key，每轮淘汰循环淘汰采样 key 中访问时间最早的 key，直到释放足够的内存空间。Redis 不使用真正的 LRU 实现的原因是真正的 LRU 实现需要更多的内存。（终究还是内存太贵啊~）

自 Redis 3.0 以来，该算法得到了改进，提高了算法的性能，也使其能够更接近真实 LRU 算法。官方称它们在模拟中发现使用幂律分布的访问模式，真正的 LRU 和 Redis 近似 LRU 算法之间的差异很小甚至不存在。

下面是实际 LRU 和 Redis 3.0, Redis 2.8 使用近似 LRU 算法的效果图：



图中能够看到三种颜色的点：

- 浅灰色带是被逐出的 key。
- 灰色带是未被逐出的 key。
- 绿色带是新添加的 key。

可以看到在均每次采样 5 个 key 情况下，Redis 3.0 版本的近似 LRU 算法与 Redis 2.8 的近似 LRU 算法的效果相比更好。而 Redis 3.0 每次采样 10 个 key 又要比每次采样 5 个 key 效果更加接近实际的 LRU 算法。所以可以适当地提高采样数，牺牲一些额外的 CPU 时间来提升近似 LRU 的效果。

可以在运行时使用 `CONFIG SET maxmemory-samples [count]` 命令修改每次采样数。

近似 LFU 算法

从 Redis 4.0 开始，可以使用 LFU 相关的淘汰策略。在 LFU 模式下，Redis 会尝试跟踪记录 key 的访问频率，因此将很少使用的 key 淘汰。这意味着经常使用的 key 更有可能保留在内存中。

LFU 近似算法：它使用“Morris 计数器”近似计数算法来估计对象访问频率，核心思想就是令访问频率 +1 是有概率的，且访问频率越高时要 +1 概率越小。每个 key 仅用低 8 位记录访问频率（范围 0~255），并结合衰减周期，令访问频率随时间减少。因为如果过了很久都没怎么被访问后，我们不再希望将该 key 视为经常访问的，以便算法可以适应访问模式的转变。

对近似 LFU 的采样类似于上面提到的近似 LRU，就不再讲采样过程了。

然而，与近似 LRU 不同的是，近似 LFU 具有一些可调参数：

- lfu-log-factor：困难系数，该因子越大，LFU 提升访问频率概率越低，默认值为 10，默认效果为在大约一百万次访问时使访问频率达到最大（255）。
- lfu-decay-time：衰减系数，越大访问频率则衰减越慢，默认值为 1，默认效果为每分钟令访问频率 -1。如果设置为 0 则不会衰减。

关于 lfu-log-factor 对于访问频率计数值的影响，可以看看下面的官方测试结果：

factor	100 hits	1000 hits	100K hits	1M hits	10M hits
0	104	255	255	255	255
1	18	49	255	255	255
10	10	18	142	255	255
100	8	11	49	143	255

从上面的结果可以得出结论：**lfu-log-factor 越小，对于访问次数较少场景的访问频率区分就更好，越大则能够区分的范围越大，但不够精确。**所以这个因子该设置多大，我们需要根据应用的访问量来进行评估。

记忆要点：8 种内存淘汰策略，默认采用不淘汰策略，volatile 针对设置过期时间 4 种，allkeys 针对所有 key 3 种，lru, lfu, random, volatile-ttl, 近似 LRU，淘汰采样，默认采样 5 个 key，近似 LFU，概率增加访问频率，随时间衰减访问频率，困难系数，衰减系数

13. Redis 7 懒情删除

懒情删除 (lazy free) 是 Redis 4.0 推出的一个功能，在 4.0 版本推出了 **unlink** 指令，它可以对 key 进行**懒删除**，原理是将 key 在全局 dict 的哈希表原本的位置设为 NULL，然后提交一个回收该 key 内存的任务给后台线程。

为什么需要懒情删除

假如我们遇上一个占用空间很大的集合 key，对它使用 del 同步删除会引起主线程阻塞较长时间。为了解决这个问题，所以 Redis 引入了懒情删除功能，在使用 unlink 指令时将实际回收内存的工作交给后台线程异步回收。

(注：如果配置了 `lazyfree_lazy_user_del` 为 1，那么使用 del 指令也会尝试进行懒情删除)

懒情删除过程

首先将 key 在全局 dict 的哈希表原本的位置设为 NULL，即第一步为取消引用，随后将回收函数和回收对象包装成一个任务，将任务从尾部插入对应类型 (LAZY_FREE) 操作线程的任务链表。对应类型的操作线程每次取出链表头部的任务进行处理，处理时将传入对象作为传入函数的参数进行调用，等到提交的任务被取出和调用完成时，将该任务从链表中删除，便完成了一次懒情删除过程。

其实在包装与提交异步删除任务之前，还需要进行删除对象的成本计算。这个成本计算其实很简单，比如 string 类型成本为 1，而 hash, list, set, zset 之类的能存储多个元素的类型删除成本就等于元素个数。**删除成本 > 64 才会使用懒情删除，否则采用同步删除方式。**

过期 key 删除策略

当 key 过期的时候，Redis 需要将过期 key 删除来腾出内存，但是 Redis 并不会马上将过期的 key 删除，否则如果有大量 key 过期，删除它们会导致 Redis 阻塞很久。所以 Redis 也是采用了一种惰性的方法删除过期 key。

Redis 会定时执行一些任务，默认是每秒要执行 10 次，但是会根据客户端连接数量动态调整执行频率（客户端越多执行频率越高）。懒惰删除过期 key 的任务也是其中的一项定时任务。除此之外还有要提的一点是，Redis 会将过期时间作为 value 存在一个 expires 字典/哈希表里。默认情况下，Redis 每轮最多扫描 400 个 expires 字典中的桶，来采样最多 20 个 key，并将其中已经过期的 key 删除。如果本轮没有扫描到任何 key，或者采样出来的 key 里有超过 10% 是已经过期的，那么继续循环一轮。但是 Redis 也不会让该任务一直循环下去，每执行 16 轮会计算一下执行时间，如果超出了限制（默认 25 ms）就结束循环。

如果是主从架构，从节点是不会自我检查并删除过期 key 的，只能由主节点进行该任务。主节点将过期 key 的删除命令发送给从节点，令从节点能够删除过期 key。

除了定时扫描和删除过期 key 之外，还有在访问 key 时检查是否过期并删除，来保证我们不会读取到过期 key。

为什么删除了很多 key 内存占用还是很高？

了解了懒惰删除后，我们便可以来回答这个问题了。如果删除了很多 key 后内存占用还是很高，可能有以下原因：

1. 如果使用了 unlink 进行删除，那么很可能是因为**大部分 key 的内存还未来得及被异步回收**。
2. 如果使用了 del 进行删除，内存占用仍然还是很高，**那么可能配置项 lazyfree_lazy_user_del 不为 0，此时 del 也会使用懒惰删除的方式**，那么原因就可能和上面说的一样了。
3. 还有一个和**操作系统**有关的原因。如果真正删除了 key 内存变化依然不大，可能是因为这些被删除的 key 所在的页依然被其它对象引用，也就是还有其它的 key 存储在同一页，操作系统是以页为单位进行回收的，**若页还存在引用则不能被回收**。

记忆要点：unlink 懒删除，为什么需要懒惰删除，解决删除大 key 阻塞问题，懒惰删除过程，取消引用，计算成本，元素个数 = 成本，删除成本 > 64 采用懒惰删除，包装任务，将任务尾插至对应任务队列，后台线程从任务队列头部取任务，执行回调函数，为什么删除了很多 key 内存占用还是很高，三个原因，unlink 未来得及异步回收内存，配置问题导致 del 采用懒惰删除，删除 key 所在页仍被引用无法被操作系统回收

14. Redis 7 后台 IO

目前 Redis 7 后台 IO (bio) 线程有 3 个，这是由于目前有**三种异步操作**，Redis 为每种操作**单独分配一个线程**进行处理。

异步操作类型

BIO_AOF_FSYNC: AOF 会用后台线程做 AOF 文件 fsync 刷盘操作

BIO_LAZY_FREE: LazyFree (懒惰删除) 功能会先评估释放一个对象需要的代价 (可以认为释放常用数据类型代价 = 元素数量), 代价大于 64 就使用后台线程进行删除工作。这个 64 是一个经验值 (通过使用经验或实验测出来的值), 删除代价小于 64 的对象如果用异步删除的话, 线程切换和线程同步 (使用互斥锁) 带来的开销可能比同步删除要大。String 类型代价都是 1, 代价考虑的是一个类型的元素数量, 即使把释放 redisObject 结构体考虑进去, 我们给 String 分配内存空间顶多也就两块 (RAW 编码), 所以释放一个 String 开销并不大。

BIO_CLOSE_FILE: 异步关闭文件描述符, 目前只用来关闭 MP-AOF 的历史文件 (Redis 7.0 MP-AOF 的 AOF 文件类型, 具体看第 21 篇)。使用异步删除的原因主要是删除 MP-AOF 历史文件之前会调用 unlink(), 它先检查文件系统中此文件的连接数是否为 1, 如果不是 1 说明此文件还有其他链接对象, 因此只对此文件的连接数进行减 1 操作。若连接数为 1, 并且在此时没有任何进程打开该文件, 此内容才会真正地被删除掉。在有进程打开此文件的情况下, 则暂时不会删除, 直到所有打开该文件的进程都结束时文件就会被删除。对于持久化文件占用空间可能比较大, 那么在 redis 把旧的持久化文件的文件描述符关闭后, 会触发 unlink() 的删除, 如果是用主线程进行删除的话, 很可能导致主线程阻塞一段时间, 所以 Redis 要把关闭文件描述符的任务交给后台线程。**我们要记得并不是简单的 close, 还可能包括删除文件, 这样你就能想起来为什么需要异步 close 了。**

异步处理过程

创建任务: 用 bio.c 提供的函数**创建一个对应类型的异步操作任务**。

提交任务: 首先**获取对应任务链表的互斥锁**, 获取成功后把任务插入对应类型任务的链表的尾部, 将对任务的待处理计数 +1, 之后发出信号唤醒因没有任务而休眠的对应类型后台线程, 最后**释放互斥锁**。

获取任务: 对应操作类型的后台线程首先查看**对应任务待处理计数**, 如果待处理计数为 0 则释放得到的互斥锁并进入休眠状态。如果大于 0 表示有任务需要处理, 从链表头部取出任务后释放互斥锁。

执行任务: **根据线程自身对应的操作类型, 执行对应的具体操作** (LazyFree 线程执行回调函数, AOF Fsync 线程调用 fsync 系统调用函数, Close File 线程调用 Close 系统调用函数)。执行完毕后释放任务实例 (任务是一个包装的结构体, 需要手动释放), 获取对应任务链表互斥锁, 将任务从链表头部移除, 对应任务待处理计数 -1。 (如果你亲自去看源码会发现最后还调用了个唤醒其他等待后台任务执行完毕线程的函数, 而这个等待功能其实在 Redis 内部暂未用到, 所以我们不用去关心它)

记忆要点: 三个后台线程, 三种异步操作, 每种操作单独分配一个后台线程, 异步操作类型, AOF 持久化刷盘, 懒惰删除, 异步关闭文件, 异步处理过程, 创建任务, 提交任务, 获取任务, 执行任务

15. Redis 7 线程模型

当你说出了后台线程或者话题变得稍微深入下去时, 面试官可能会继续抛出关于 Redis 线程模型的问题。所以对于想要在面试 carry Redis 底层的人, 还是需要了解 Redis 线程模型的, 了解了 Redis 线程模型你就知道为什么 Redis 这么快, 在将来到自己写代码也可能有派上用场的一天。

事件循环器 (IO 多路复用)

在讲到线程模型之前，需要先讲一讲这个重要的**事件循环器**。

Redis 快的原因不仅除了是内存型，还有一个很大的原因是采用了**多路复用的 IO 机制**。在 Redis 里 ae.c 和 ae.h 源码中对多路复用进行了一个封装，我们就叫它**事件循环器**吧（也有的叫多路复用器，事件分派器的）。

事件循环器支持四种 IO 多路复用：evport, epoll, kqueue, select。事件循环器选择其中一种多路复用机制进行使用，前三种和操作系统有关，优先选择顺序从左到右，如果操作系统前三种都不支持，则会选择第四种 select（性能最差）。**可以认为在 linux 系统下选择的通常都是 epoll。**（如果你说了这个被追问 epoll 原理，我就不帮你接这招了，自己去找篇文章看吧🤔~当然也可以微信问我这个永久顾问~）

Redis 的事件循环器处理的事件有两种：**文件事件和时间事件**。

文件事件：指的一般是和 socket 有关的事件，文件事件可以设置四个事件状态：未设置（NONE），可读（WRITEABLE），可写（READABLE），逆序读写（BARRIER）。目前和客户端有关的事件回调处理函数主要有三种类型：连接处理函数 acceptTcpHandler（接受和建立与新客户端的连接），readQueryFromClient 请求处理函数（读取客户端命令并解析，命令存在则执行），回复处理函数 sendReplyToClient。（回复客户端）

时间事件：指的是按规定时间执行的事件，时间事件分为两类：**定时事件**（在指定的时间之后执行一次），**周期性事件**（每隔指定时间就执行一次）。

事件执行过程：首先计算调用多路复用获取事件的超时时间（通常等于下一个时间事件指定执行时间与现在的间隔），获取就绪事件前先执行前置处理函数（beforeSleep），调用具体的多路复用 api 在超时时间内阻塞获取就绪事件与就绪事件数量，之后执行后置处理函数（afterSleep）。之后遍历得到的就绪事件，通过事件状态掩码判断事件状态。

如果是**文件事件**，先判断是否是读写逆序状态，再判断可读，可写，可读可写则调用事件绑定的回调函数。如果设置了逆序读写，则转变为先写后读，否则正常情况下先读后写。

如果是**时间事件**，先判断是否被标记为了删除状态，若被标记为了删除状态，该事件没有其它引用了就将它删除；如果没有被标记删除状态，判断任务指定执行时间是否小于当前时间，小于则说明已经可以执行，接着令事件的引用计数 +1，执行实际的事件处理回调函数，执行完后再将引用计数 -1。最后判断事件处理回调函数返回的值，返回值表示下次是否还需不需要执行，如果不需要（返回值为 -1）则标记成删除状态，等待下一次循环将事件删除，如果需要则重新设置任务指定执行时间，回调函数返回值便是下一次指定的时间间隔。

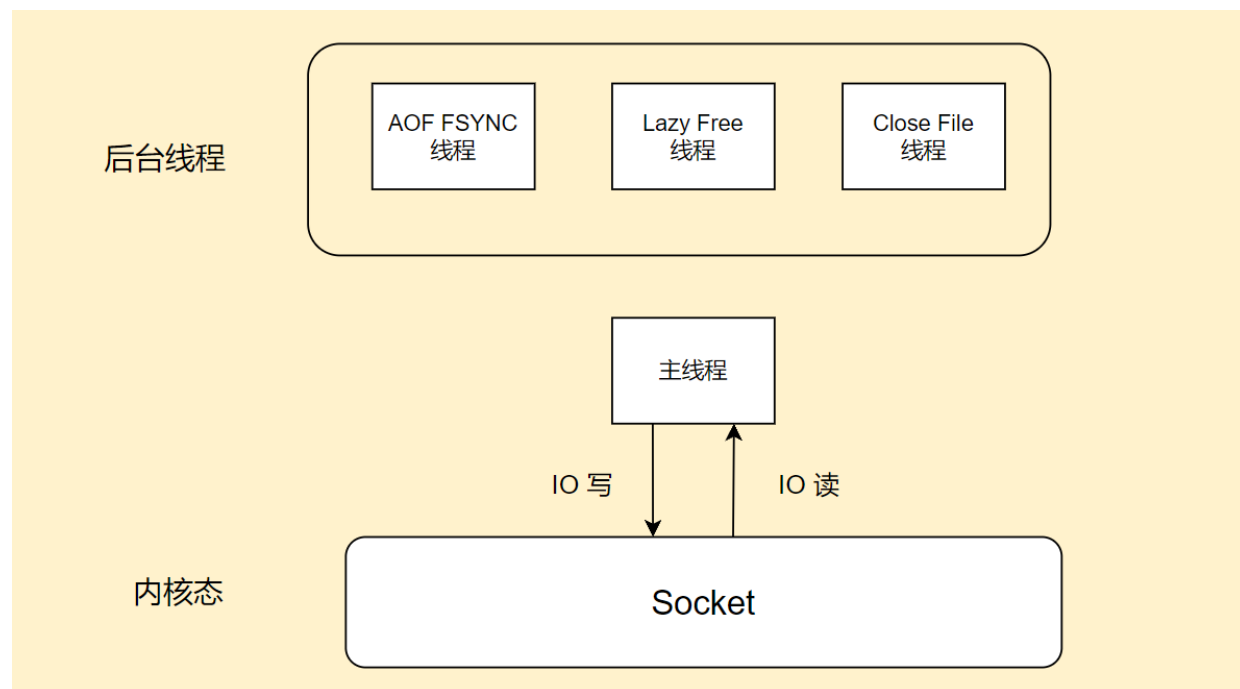
事件循环器真的很重要，因为 Redis 就是基于它来运行的，这也被叫作**事件驱动模型 (Reactor)**。比如 Redis 把监听客户端的 socket 的事件注册到操作系统的多路复用程序中，当客户端发送命令让读事件就绪时，事件循环器就可以通过多路复用获取到该事件，进而通过事件注册的回调处理函数来查找与执行命令。除了与客户端 socket 交互的事件之外，服务器定时任务（ServerCron，包括清理过期 key，触发持久化等）也是通过事件循环器来进行的。

（不过不要把一次过程当作事件驱动，事件驱动是死循环，事件来了就处理，否则就等着。也不要吧多路复用当作事件驱动的必要条件，不用多路复用也可以实现，事件驱动模型只是一种设计模式）

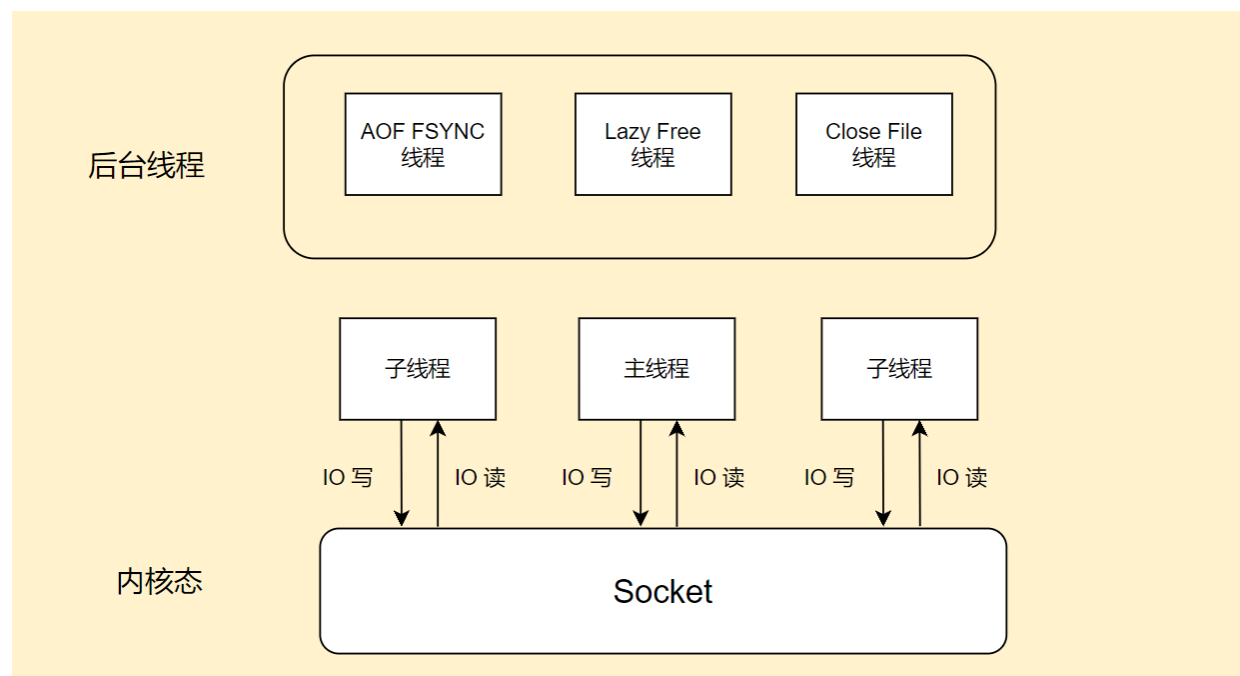
多线程与线程模型

还是老生常谈的一句话：**Redis 的性能瓶颈不在于 CPU，而在于内存和网络**。所以 Redis 6.0 加入了 IO 多线程机制，来**提高网络 IO 读写性能**。不过多线程默认是不开启的，需要修改配置文件自行开启。下面我们来看加入了多线程 Redis 的 IO 处理会有什么改变。

当我们没有开启多线程时，线程模型则是主线程 + 3 个后台线程：



这时主线程不仅自己要负责执行客户端发来的命令等事情，还得负责网络 IO 读写。大多数执行命令这种事情“啪”一下就做完了，读写 socket 这种还需要陷入内核态的操作与执行命令相比就要慢了很多，很明显会拖 Redis 大哥的后腿。所以在 Redis 6.0 就允许找线程小弟来帮忙**分担网络 IO 的工作**，自己则更专心于执行命令等别的工作。



一定要记住哦，虽然 Redis 引入了多线程，但是还是只有主线程负责处理命令，子线程只负责网络 IO。

接下来我们来了解一下，多线程在读写客户端 socket 方面会有什么变化：

读场景：在多线程+后台线程模型下，**用户**通过客户端把命令发送过来（这里强调了用户，因为主从节点关系的话服务器也属于一个服务器的客户端，但是并不会使用子线程 IO 来进行读操作），取出事件并回调 `readQueryFromClient`（请求处理函数），在这个函数实现里，**如果开启了多线程读机制，IO 线程在空闲状态且客户端未阻塞，则不会直接处理，而是延期处理，将该客户端添加到待读取的客户端链表中就返回，该客户端等待下次事件循环时，在前置处理函数（beforeSleep）中通过 round_robin 方式（各线程轮流分一个直至分完）将所有待读取的客户端分配给主线程和 IO 线程处理。**

写场景：在命令的具体实现函数中，会调用 `addReply` 函数将回复数据写入对应客户端的输出缓冲区（output buffer）。**无论是否开启多线程机制，都有一个待写入的客户端链表，在 addReply 里都会尝试将对应客户端加入链表，能加入链表的条件是：如果 IO 线程空闲（包括未开启多线程）且该客户端没有要回复的数据即可加入。在新一轮的事件循环的前置处理函数（beforeSleep）中，通过 round_robin 方式将所有待写入的客户端分配给主线程和 IO 线程，之后各线程将回复数据发送给客户端。**

即使开启了 Redis 多线程，子线程也不是一直在工作。如果待处理的客户端数量 $< \text{IO 线程数} * 2$ ，就停止 IO 线程，只使用主线程来处理 IO。停止 IO 线程并不是直接关闭 IO 线程，而是对所有 IO 线程的互斥锁（`pthread_mutex`）加锁令 IO 线程获取不到锁而阻塞。

（引申问题）为什么执行命令不用多线程

- Redis 的性能瓶颈不在于 CPU，而在于内存和网络。况且单线程 Redis 性能已经足够优秀，能够满足大多数高并发场景。
- 使用多线程会引入线程冲突之类的线程问题，会提升实现的复杂度。
- 可以在单台机器上部署多个实例。
- 可以把命令打包发送，让每次传输可以处理更多命令。

（附加题）客户端发送一条命令到最后响应的过程

其实这个题就是对以上提到的读写场景做的一个总结。Redis 客户端和服务端是使用 socket 通信的，Redis 客户端将一条命令发送给服务器，那么服务器收到了就会将它保存在 socket 输入缓冲区。由于服务器通过多路复用机制监听了客户端 socket，并在初始化客户端的连接的时候为可读事件绑定了一个（`readQueryFromClient`）回调函数，所以这里 socket 收到数据，服务器就能通过多路复用机制在新一轮事件循环中将 socket 取出，并调用可读事件的回调函数。在这个函数里，如果开启了多线程读机制，推迟读事件到下一轮事件循环的前置处理函数（beforeSleep）分配给主线程和 IO 线程处理，而如果是主线程则直接处理，把客户端 socket 的输入缓冲区数据先读到客户端的查询缓冲区（query buffer，客户端结构体的一个成员变量）。

之后会调用将查询缓冲区数据解析成命令的函数，如果开启了多线程，那么 IO 线程会参与解析命令，当所有 IO 线程处理完各自被分配的客户端后，主线程接着会在事件循环的前置处理函数（beforeSleep）中串行查找并执行所有客户端的命令。而如果没有启用多线程，就是主线程自己负责解析然后查找和执行命令了。命令具体实现的函数会将回复数据写入对应客户端的输出缓冲区（output buffer），并且尝试将客户端加入待写入客户端链表。最后，在事件循环的前置处理函数（beforeSleep）中，将待写入客户端分配给主线程和 IO 线程，各线程将输出缓冲区的数据写入 socket 输出缓冲区，最后发送给客户端。

记忆要点：Redis 快的原因，多路复用，事件循环器，支持四种多路复用，linux 为 `epoll`，两种事件类型，文件事件，文件事件状态，未设置，可读，可写，逆序读写，时间事件，两类，定时事件，周期性事件，事件执行过程，计算超时时间，前置处理，调用多路复用 api 获取就绪事件，后置处理，遍历就绪事件，文件事件处理，判断顺序：逆序读写->可读->可写，执行回调函数，时间事件处理，判断删除标记，判断执行时

间是否小于当前时间，小于则可执行，引用计数修改，执行回调函数，根据回调函数返回值判断类型，定时事件标记删除，周期性事件更新事件执行时间，事件驱动模型，多线程与线程模型，Redis 的性能瓶颈不在于 CPU 而在于内存和网络，Redis 6.0 引入多线程提高网络 IO 性能，多线程默认关闭，主线程 + 后台线程模型，多线程 + 后台线程模型，子线程只负责网络 IO，读场景，写场景，延期处理，加入待读取/写入客户端链表，处理时机在下次事件循环前置处理函数

16. Redis 7 事务

Redis 事务比较鸡肋，Redis 7 也没有什么更新。事务最新的命令 Watch 也是久远的 2.2 版本推出的。Redis 的事务一直被人诟病，因为它的事务竟然**没有回滚操作**。网络上一些介绍 Redis 事务的文章可能说事务具有原子性，隔离性，然后就开讲 Redis 事务。这会造成读者的误解！我们现在就来简单分析 **Redis 事务的功能机制以及它具有什么性质**。

事务机制

Redis 事务允许一次性执行一组命令，以命令 MULTI、EXEC 为主，还有两个重要的辅助命令：DISCARD 和 WATCH。

MULTI：开启事务，开启后输入的命令都不会直接执行，而是加入事务命令队列，如果检测到参数数量错误或命令不存在，则会中止当前事务并清空队列。而执行时检测出的错误并不会阻止其它命令的执行。

EXEC：执行事务，按执行命令顺序返回结果。

```
127.0.0.1:6379> MULTI
OK

# 不能嵌套事务
127.0.0.1:6379(TX)> MULTI
(error) ERR MULTI calls can not be nested

# 输入事务命令内容
127.0.0.1:6379(TX)> SET a 1
QUEUED
127.0.0.1:6379(TX)> GET a
QUEUED

# 执行事务
127.0.0.1:6379(TX)> EXEC
1) OK
2) "1"

# 开启新事务，测试执行出错
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379(TX)> SET a 666
QUEUED

# 语法错误命令
127.0.0.1:6379(TX)> SET a a b c
```

```
QUEUED
```

```
127.0.0.1:6379(TX)> GET a
```

```
QUEUED
```

```
# 执行出错不会阻止其它命令的执行
```

```
127.0.0.1:6379(TX)> EXEC
```

```
1) OK
```

```
2) (error) ERR syntax error
```

```
3) "666"
```

DISCARD：终止事务，清空命令队列并终止事务。

WATCH：监听 key，被监听的 key 如果在事务之外被修改，则事务不会执行（EXEC 时结果返回 nil）。

UNWATCH：取消监听 key。

```
# 监听 "a"
```

```
127.0.0.1:6379> WATCH a
```

```
OK
```

```
# 在开启事务前修改 a
```

```
127.0.0.1:6379> SET a 3
```

```
OK
```

```
# 事务
```

```
127.0.0.1:6379> MULTI
```

```
OK
```

```
127.0.0.1:6379(TX)> SET a 2
```

```
QUEUED
```

```
# 结果为 nil，未执行任何命令
```

```
127.0.0.1:6379(TX)> EXEC
```

```
(nil)
```

```
# 验证是否未执行 SET a 2
```

```
127.0.0.1:6379> GET a
```

```
"3"
```

以上 5 个命令就是 Redis 事务提供的全部功能。通过以上事务命令介绍和使用演示我们来解答 **Redis 事务是否满足 ACID 四大特性**：

Redis 事务具有原子性吗？

答：Redis 事务**不具有原子性**。原子性指事务中的操作要么全部成功，要么全部失败，如果中途出错需要回滚到事务之前的状态。而 Redis 事务执行出错不会阻止其它命令的执行，所以并不满足这个条件。

Redis 事务具有隔离性吗？

答：Redis 事务**具有隔离性**。隔离性原本指并发事务不会相互影响，但由于 Redis 是单线程不存在并发事务，所以我们考虑事务和事务之外的命令执行情况，Redis 事务执行时不会执行事务之外的命令，所以可以说它具有隔离性。

Redis 事务具有一致性吗？

答：一致性指事务执行前后要满足相同的约束条件（或者说相同的一致性状态），比如转账前后两个人余额总和要相同。一致性属于应用范畴，我们不讨论 Redis 事务是否具有一致性。

Redis 事务具有持久性吗？


答：Redis 事务**不具有持久性**。持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。然而 Redis 事务提交后也不会强制进行持久化，所以不满足该条件。

综上所述，Redis 只能说具有事务特性中的**隔离性**。

总结：Redis 事务特点是顺序性和隔离性。

记忆要点：事务特性，ACID，Redis 事务特点是顺序性和隔离性

17. Redis 7 缓存穿透，缓存击穿，缓存雪崩问题

关于缓存穿透，缓存击穿，缓存雪崩问题，它不属于 Redis 底层问题，而是应用问题，并且这个所指的缓存也不止局限于 Redis。这些问题网上文章很多，看起来是 Redis 最热门的面试题之一了，所以我也将这个问题拿进来讲一讲。如果你已经在网上看过这个面试题无数遍了，那就可以跳过这节，留下来看看我对这些问题的个人看法也可以。

缓存穿透

什么是缓存穿透： 用户查询不存在的数据。

缓存穿透会有什么问题： 如果出现大量查询不存在数据的请求，轻则增加数据库压力，重则将数据库打垮。

解决方案：

- 1. 校验参数：** 适用于能够界定一个合法查询范围的场景，直接屏蔽范围外的非法参数。
- 2. 缓存空值：** 这是一种通用的做法，把查询结果为空值的参数也缓存下来，下次查询时就可以直接从缓存取出空值返回。
- 3. 布隆过滤器：** 适用于空值查询参数种类太多的场景，因为当空值查询参数种类太多时，缓存太多空值也是很浪费内存的，如果缓存所有空值占用的内存空间比把所有真实数据加入布隆过滤器还要大，则可以采取布隆过滤器的方案。我们将所有真实数据加入布隆过滤器，布隆过滤器是一个**内存占用少**，能够判断数据是否**不存在**的数据结构。它的判断特性是**告知存在时数据不一定存在，但是告知不存在则一定不存在**。它的误判率并不高，合理配置可以保持在小数级别的概率，则可以过滤掉 99.9% 数据不存在的请求。关于布隆过滤器更详细的内容可以看第 27 题。

缓存击穿

什么是缓存击穿： 用户查询缓存已失效的数据。

缓存击穿会有什么问题： 如果出现大量查询缓存已失效的数据，轻则增加数据库压力，重则将数据库打垮。

解决方案：

1. 分布式锁： 当查询发现缓存不存在时，在缓存中设置关于该 key 的分布式锁，从数据库查询到结果并更新缓存后解锁。在第一个查询解锁之前，其余查询相同 key 的线程只能阻塞等待解锁，此时解锁后可以直接从缓存获取数据。

2. 热点缓存不过期： 如果能发现什么数据是热点的话，可以不设置过期时间。比如做社交平台的话，我们可以让百万粉丝以上大 v 的用户信息等缓存不会过期。

3. 续期： 缓存被访问时我们就为它进行续期，也可以访问次数到一个阈值后再进行续期，从而让热点缓存不要失效。（但是冷门数据过期后，突然有大量查询的场景这个方案是防不住的）

缓存雪崩

什么是缓存雪崩： 大量热点数据同时失效。

缓存雪崩会有什么问题： 如果出现大量热点数据同时失效，随后这些热点数据还在被大量查询，轻则增加数据库压力，重则将数据库打垮。

解决方案：

1. 分布式锁： 讲过啦！

2. 热点缓存不过期： 讲过啦！

3. 续期： 讲过啦！

4. 过期时间加随机数： 更新缓存并设置过期时间的时候，不要固定过期时间，选用一个固定值 + 随机值，让缓存不要在同一时间过期。

个人看法

对于缓存穿透，一般用缓存空值方案即可 carry。而对于缓存雪崩，缓存雪崩，面试当然你回答的方案多一些好。但是按实际应用来说的话，缓存续期肯定是做的，热点缓存不过期一般也做。但是过期时间加随机数我个人觉得就有点搞笑，让缓存不要在同一时间过期，有点针对预设场景出方案的感觉，而且也有点随缘，假如运气差的话可能在秒杀开始前 90 % 缓存过期。根本原因是大量请求打入数据库，而分布式锁我个人认为才是真正解决根本问题的方案。Facebook 也运用这种方案并把这种方案称为 lease 机制。我觉得使用缓存空值 + 续期 + 分布式锁就已经足够 carry 了，如果再能评估出热点令它不过期就更加好了。我觉得当你给人介绍完方案，也可以发表一下自己对这些方案的看法，表达一下自己的主见。

有的人可能看起来觉得分布式锁方案很麻烦，其实十行代码左右就可以搞定，以下给出如何使用的伪代码：

```
// 缓存查询失败后开始
lockName = "lock:" + key;

// 检测分布式锁是否存在
if(Redis.Exists(lockName) == true) {
    while(Redis.Exists(lockName) == true) {
        Sleep(100); // 大概每 100 ms 检测一次锁是否存在
    }
    value = Redis.Get(Key); // 此时可以直接从缓存获取了
    return value;
}
```



```
} else {  
    Redis.Set(lockName, 1); // 加锁操作  
}  
  
...  
// 从数据库获取到数据后  
Redis.Set(key, value); // 更新缓存  
Redis.Del(lockName); // 解锁
```

redis member 彬彬看过之后表示主要看场景，反正场景适合就看性价比，啥方案都能做。过期时间加随机数主要就是突发流量加大面积缓存失效才那样做，小范围的缓存穿透代价很大的场景就是分布式锁加续期来做的，定制化针对，其它没出问题的就没做啥特定的优化。加随机数来打散过期时间，他倒是觉得是在一些特殊场景性价比很高的方案，让突发流量平滑点，过期也平滑点，也不用每次 get 都被包装个 lua 脚本，然后还要想做续期。

彬彬说得有道理，我之前认为缓存续期肯定都要做，但是事实上好像并不是这样，看来还是得看场景啊。好吧，大家还是听大佬的话吧👂。

记忆要点：缓存穿透，数据不存在，三种解决方案，校验参数，缓存空值，布隆过滤器，缓存击穿，数据缓存失效，三种解决方案，分布式锁，热点缓存不过期，续期，过期时间加随机数

18. Redis 7 大 key 问题

大 key 问题也是一个热门的面试应用问题，所以也被归纳了进来，接下来我们聊聊这个大 key 问题是什么，以及解决方案。当然，如果你在网上看过无数遍了，那么可以跳过。

什么是大 key 问题

大 key 指的是某个键值对占用空间太大，通常都是 value 部分太大。

大 key 带来的问题

- 1. 慢查询：**大 key 影响命令执行速度和网络传输速度，会导致客户端等待较长时间。
- 2. 同步删除阻塞：**使用 del 同步删除大 key 可能导致 Redis 阻塞较久。
- 3. 主从同步慢：**大 key 过多影响主从节点之间的同步速度。若单 key 太大甚至可能无法使用全局共享复制缓冲区进行部分同步，导致所有从节点只能使用全量同步。
- 4. 集群内存分布不均：**在集群模式下，每个实例被分配部分 key，那么可能出现某个实例因为被分配较多的大 key 导致内存占用更高的情况，同时该实例的查询性能也会下降。

解决方案

大 key 属于设计失误问题，应该在设计阶段就将其排除。不过我们既然要解决问题，当然是等到问题出现了才入手，所以这里不讨论前期设计的问题，并且设计应该具体场景具体分析，我们只讨论出现大 key 后怎么办。

1. 使用 scan 类命令：如果是 Hash, Set, Zset 这类大 key，一次性遍历完毕会导致阻塞过久，我们可以用 hscan, sscan, zscan 命令来每次获取少量数据完成遍历。除了遍历操作，其它时间复杂度高的命令也要避免。

2. 删除大 key：当出现大 key 时，应该重新设计缓存并尽早删除大 key。删除大 key 的时候切记不要用同步删除的方式，否则删除大 key 可能会导致 Redis 阻塞过久。如果是 4.0 之后的版本应该使用 unlink 命令进行异步删除。

如何检测大 key

1. redis-cli --bigkeys：redis-cli --bigkeys 能够统计五种常用数据类型最大的 key，key 数量占总量百分比，平均占用内存空间的功能。最好不要在生产环境中的主节点使用，否则可能会导致阻塞，可以在从节点使用。如果你说没有从节点，那你也可以在后面加上参数 `-i [休眠秒数]`，可以做到每扫描 100 条休眠指定时间。

```
screamovo@USER-20220225QK:~$ redis-cli --bigkeys

# Scanning the entire keyspace to find biggest keys as well as
# average sizes per key type.  You can use -i 0.1 to sleep 0.1 sec
# per 100 SCAN commands (not usually needed).

[00.00%] Biggest zset found so far "ZA" with 1 members
[00.00%] Biggest string found so far "key:__rand_int__" with 3 bytes
[00.00%] Biggest hash found so far "myhash" with 1 fields
[00.00%] Biggest string found so far "counter:__rand_int__" with 6 bytes
[00.00%] Biggest list found so far "mylist" with 100000 items

----- summary -----

Sampled 7 keys in the keyspace!
Total key length in bytes is 52 (avg len 7.43)

Biggest list found "mylist" has 100000 items
Biggest hash found "myhash" has 1 fields
Biggest string found "counter:__rand_int__" has 6 bytes
Biggest zset found "ZA" has 1 members

1 lists with 100000 items (14.29% of keys, avg size 100000.00)
1 hashes with 1 fields (14.29% of keys, avg size 1.00)
4 strings with 13 bytes (57.14% of keys, avg size 3.25)
0 streams with 0 entries (00.00% of keys, avg size 0.00)
0 sets with 0 members (00.00% of keys, avg size 0.00)
1 zsets with 1 members (14.29% of keys, avg size 1.00)
```

2. rdbtools：rdbtools 是一款 python 编写的 rdb 文件分析工具，离线时可以使用 rdbtools 工具对 rdb 文件进行分析来找出大 key。（具体用法可以上网搜索）

3. scan + memory usage：若想要在生产环境上统计每个 key 的内存占用情况，可以使用 scan 命令每次遍历少量 key，之后将获取到的 key 使用 memory usage 命令查询占用内存字节数。少量多次的将所有 key 内存占用情况统计完毕。

记忆要点：什么是大 key，键值对占用内存大，大 key 带来的问题，慢查询，同步删除阻塞，主从同步慢，集群内存分布不均，解决方案，使用 scan 类命令，删除大 key，如何检测大 key，redis-cli --bigkeys，rdbtools，scan + memory usage

19. Redis 7 热点 key 问题

热点 key 是指某个 key 被访问得很频繁。热点 key 如果过于集中在某台服务器上可能会出现严重的问题，同时热点 key 问题也属于应用范畴。如果你已经在网上看过很多遍了，依然可以继续跳过~

热点 key 产生原因：

1. 双十一促销，秒杀，大 v 带货等场景容易出现热点 key。
2. 集群模式下，大量请求较频繁的 key 集中被分配到某个分片（实例）上。

热点 key 带来的问题

轻则增加网卡压力，重则将网卡填满。（响应速度变慢，还可能拒绝请求）

轻则增加 Redis 压力，重则将 Redis 打垮。

轻则增加数据库压力，重则将数据库打垮。（前提是 Redis 倒下了）

如何发现热点 key

1. **人工评估：** 比如商品要搞秒杀，大 v 要带什么货，这些都很容易成为热点 key，它们是可以被提前发现的。
2. **监控程序：** 可以在客户端或者 proxy 设置监控程序，统计每个 key 的访问次数，之后分析哪些是热点 key，最后将热点 key 反馈出来。分析热点 key 的方案有很多，简单的就是用哈希表统计 key 对应访问次数，并设置个时间范围和阈值，在规定时间内访问次数超过阈值的就标记为热点。

解决方案

1. **增加缓存：** 当 Redis 大哥绷不住时，你可以给它加个小弟分担一下。我们可以在业务代码上编写**本地缓存**，比如用一个哈希表缓存监控程序反馈过来的热点 key，这样下次查询热点 key 时可以直接返回，不用再继续请求 Redis。当然，不止可以在业务上加本地缓存，如果架构上有 proxy（代理服务器）的话，也可以在 proxy 上缓存热点 key。当然，多了一级缓存也会**带来数据一致性问题**，这下就要考虑本地缓存，Redis，数据库之间的数据一致性了。

2. **集群热点 Key Rehash：** 如果存在大量热点 key 集中在单台 Redis 上，我们也可以通过 rehash 把热点 key 打散到多台机器上。我们可以找出能够让热点 key 分布均匀的 hash tag 算法，然后**添加到 Redis 里使用 hash tag**，也就是在 key 后加 {}（如 user{1}），此时 {} 里输入的值才会被 Redis 用 crc16 算法来计算会被分配到的 slot（插槽），从而将热点 key 属于的 slot 能够分散在多台机器。（集群模式有 16384 个 slot，每个分片/实例 管理一个范围的 slot，key 按照 crc16 计算出的 slot 分配给对应的分片进行存储）

3. **读写分离：** 读写分离架构适用于**读多写少**的热点 key 场景，读写分离架构一般就是一主多从，主节点对外提供读写服务（也可以只写），从节点对外提供读服务，客户端读取缓存时随机选取一个节点进行读取，多个节点就分担了读压力。但是这种方案会有**主从数据不一致问题**。

网络上一般也提到了备份 key 这个方案，让多个分片去存储同个热点 key 从而分散压力。我觉得这个方案的数据不一致问题有点可怕，多个分片要同步数据可不是一件简单的事情，上面介绍的集群热点 Key Rehash 方案是我看了备份 key 方案自行设计的，不知道你们认为靠不靠谱~哈哈，大家自己思考吧~🤔

记忆要点：热点 key 产生原因，秒杀，双十一，大 v 带货，热点 key 带来的问题，网卡压力，Redis 压力，数据库压力，如何发现热点 key，人工评估，监控程序，解决方案，增加缓存，集群热点 Key Rehash，读写分离

20. Redis 7 数据一致性问题

数据一致性问题不是 Redis 的专属问题，而是任何缓存与数据库之间的问题，其它的还有主从数据一致性问题，不过这里一般问的是缓存与数据库之间。数据一致性问题也是 Redis 面试热门题目之一了，我们接下来就来讨论这个问题。同样的，如果你已经把这个问题刷烂了，那也可以跳过~

什么是数据一致性问题

这里说的数据一致性问题指的是**缓存与数据库之间数据不一致的问题**，因为数据写入存在先后顺序，一方的数据总是较另一方更旧，旧数据我们一般称它为**脏数据**。

数据一致性一般分为三种类型：

强一致性：这种一致性级别是最符合用户直觉的，它要求系统写入什么，读出来的也会是什么，用户体验好，但实现起来往往对系统的性能影响大。

弱一致性：这种一致性级别约束了系统在写入成功后，不承诺立即可以读到写入的值，也不承诺多久之后数据能够达到一致，但会尽可能地保证到某个时间级别（比如秒级别）后，数据能够达到一致状态。

最终一致性：最终一致性是弱一致性的一个特例，系统会保证在一定时间内，能够达到一个数据一致的状态。这里之所以将最终一致性单独提出来，是因为它是弱一致性中非常推崇的一种一致性模型，也是业界在大型分布式系统的数据一致性上比较推崇的模型。

一般更新策略

我们先来看看一般的数据更新策略：

先写数据库，后写缓存：假设有两个线程 A 和 B 要更新同个数据，但是要更新的值不一样。那么 A 写数据库 -> B 写数据库 -> B 写缓存 -> A 写缓存，这样就会出现数据不一致。

先写缓存，后写数据库：假设有两个线程 A 和 B 要更新同个数据，但是要更新的值不一样。那么 A 写缓存 -> B 写数据库 -> A 写数据库 -> B 写缓存，这样也会出现数据不一致。

先删缓存，后写数据库：在删缓存到写数据库之间，用户会因为缓存失效直接去访问数据库，此时查询到脏数据，随后会将缓存用脏数据进行更新，这会导致接下来的查询依然读取到脏数据，出现数据不一致。

先写数据库，后删缓存：在写完数据库到删除缓存之间，用户访问缓存会读取到脏数据。但是删除缓存后，新的访问从数据库获取数据，就可以拿到新数据更新缓存。理想情况下是这样，但是考虑多线程并发场景，还是线程 A 和 B，只不过这回 A 是读取，B 是写入。假如缓存不存在，A 查询数据库 -> B 写数据库 -> B 删缓存 -> A 写缓存。那么此时 A 用查询得到的旧数据更新缓存，仍然出现了数据不一致。

延迟双删和重试删除策略

延迟双删策略：先删缓存，更新数据库，延迟一段时间后再删除缓存。

延迟双删的第二次延迟删除就是为了把之前可能产生的缓存脏数据删除掉。延迟双删策略想要保证数据一致性，主要在于要把控好延迟时间。一般是估计一个业务处理所需的最大时间，再加上个几百毫秒或者 1 秒。而如果数据库是读写分离架构（从库提供读服务），主从节点之间还可能数据不一致，这就有可能读取到从库旧数据而用旧数据更新缓存，所以还需要把数据库完成主从同步的估计时间加上。

延迟双删的第二次删除需要延迟等待，可以起个线程异步删除，从而提高性能。

重试删除策略：延迟双删的第二次删除可能会由于某些原因执行失败了，为了保证数据一致性，还需要重试删除操作。删除失败时可以将 key 发到一个重试删除消息队列里（如果只是让同进程重试，本地使用链表或者队列结构即可），让另一个线程从队列中取出来重试删除操作，直到成功为止。

读取 binlog 删除缓存：

binlog 指的是 MySQL 的二进制日志文件，其实就是一个通过数据库日志让缓存与数据库保持同步的思想。可以使用阿里巴巴的 MySQL binlog 增量订阅&消费组件 canal 去订阅 binlog，将最新日志放入消息队列里，让一个服务专门取消息来删除缓存（当然也可以更新缓存）。

以上提到的方案都不是强一致性的，而是**最终一致性**。如果要实现强一致性，在分布式场景下，那么可能需要像 Raft, Paxos, 2pc 之类的分布式共识算法，让多数节点写入数据后才反馈客户端成功消息。然而我们使用 Redis 通常是为了**高性能缓存**，保证强一致性会导致性能有一定的损失，**大多数业务都是以最终一致性为目标即可**。如果要保证最终一致性，读取 binlog 删除缓存是比较稳定的，推荐使用的方案。

记忆要点：缓存与数据库数据一致性问题，三种类型，强一致性，弱一致性，最终一致性，一般更新策略，四种策略，删缓存还是更新缓存，缓存和数据库谁先谁后，延迟双删，评估延迟时间，删除重试，读取 binlog 删除缓存

21. Redis 7 持久化机制

Redis 的持久化机制也是面试常考题之一。而且 Redis 7 的 AOF 持久化变成了 **MP-AOF** 机制，如果你能讲出 Redis 7 的 MP-AOF，而不是大家都在背的旧 AOF 机制，那么面试官肯定对你刮目相看，并且觉得你这个人挖掘和学习新技术的本领还挺强的。哈哈，先暂停幻想🤔，我们一起来聊聊 Redis 7 的持久化机制。

RDB

RDB(Redis Database)是 Redis 的持久化机制之一，中文一般叫它**内存快照**。目前最新的 RDB 版本是 11，不过是 5 个月前更新的，7.0.6 之后的版本才可以使用到，Redis 7.0 正式版本（7.0.0）仍然是 10。RDB 版本存在兼容问题，新版本 Redis 可以使用旧版本 RDB 文件，但是旧版本 Redis 无法使用新版本 RDB 文件，**RDB 是向后兼容的**。

RDB 存储形式：RDB 是以二进制形式存储数据的，这个二进制形式最直观的理解就是像整数不用字符串形式写入文件，而是用它的二进制形式写入文件，所以你直接查看 RDB 文件是不能直接看出来原数据是什么的。

RDB 生成时机： RDB 可以执行 SAVE（同步保存）或者 BGSAVE（创建子进程后台保存）命令主动开始 RDB 的持久化过程。除此之外，RDB 还有被动持久化策略，被动持久化是由服务器定时任务（ServerCron）检查并执行的，该策略可以自己在 redis.conf 配置文件中配置，默认被动持久化时机如下：

- 900 秒内发生 1 次修改，距离上次保存超过 900 秒后则创建子进程进行 RDB 持久化。
- 300 秒内发生 10 次修改，距离上次保存超过 300 秒后则创建子进程进行 RDB 持久化。
- 60 秒内发生 10000 次修改，距离上次保存超过 60 秒后则创建子进程进行 RDB 持久化。

RDB 如何存储： RDB 存储时首先创建一个临时文件，先写入 REDIS + RDB 版本号，再写入一些关于服务器当前状态和 RDB 文件的辅助信息，之后遍历所有数据库（Redis 有 16 个库）里的键值对，为每个键值对选择合适的编码，能压缩存储的就压缩，将它们以二进制形式写入 RDB 临时文件。最后是写入 Redis Module 的一些元信息，EOF 结束符和 CRC64 校验和。写完临时文件后，用配置的 RDB 文件名重命名临时文件。

RDB 生成时写入数据怎么办： 这个场景的前提是后台生成 RDB。由于父子进程共享内存，所以我们担心子进程在读取数据库时，读取到父进程修改过的数据，于是子进程就将这个新数据写入 RDB 文件了，导致 RDB 文件不是创建时刻的内存快照。

这个问题实际上是操作系统解决的，操作系统用的是**写时复制（Copy-On-Write）**机制，父子进程共享的内存是只读的，如果哪一个进程要写入，则会将发生写入的页拷贝一份（谁写谁拷贝），修改自己的页表内容，之后在拷贝的页上修改数据。RDB 生成在 COW 机制下，父进程最多会复制一倍的内存。

RDB 数据恢复过程：

1. 读取 RDB 版本，若 RDB 版本小于等于当前 Redis 支持的版本则继续。
2. 读取 type，即读取下一个字节，type 占 1 字节。
3. 解析 type 获取接下来内容的类型，从而调用相应类型的读取函数去读取并加载它。
4. 重复第二第三步直到读取到结束符。
5. 如果 RDB 版本大于等于 5，则判断通过读取的 RDB 内容计算出的 CRC64 校验值与 RDB 文件末尾的校验值是否相同。相同则说明 RDB 文件完整，不相同则会退出进程。

RDB 的优点：

1. RDB 以二进制形式存储数据，并且尽量选择压缩数据存储，所以 RDB 文件一个优点是节省空间。
2. RDB 是数据库在某个时刻的内存快照，恢复速度快。

RDB 的缺点：

1. 由于 COW 机制的存在，如果在后台生成 RDB 时写入数据较多会比较消耗性能和内存空间。
2. RDB 被动持久化策略会有丢失部分新数据的风险。虽然可以自己配置策略，但一般也不会令 RDB 持久化得太频繁，生成越频繁性能越差。

AOF

Redis 7 推出了 MP-AOF，AOF 的本质没变，变的主要是 **AOF 文件结构**。我们先讲 AOF 持久化再讲 MP-AOF。

什么是 AOF： AOF (Append-Only-Files) 是可以看作是一种**增量日志文件**，日志的内容是以 RESP 协议格式（Redis 设计的网络传输格式）保存的写命令。默认不开启 AOF 持久化，需要修改配置文件开启。

AOF 写入时机： 每次执行完写操作的命令都会将该命令以 RESP 协议格式写入 AOF 缓冲区。

AOF 刷盘时机：

always： Redis 在每轮事件循环的前置处理函数中，将 AOF 缓冲区中的所有内容通过 write 系统调用函数追加写入 AOF 文件（此时在内核缓冲区），随后立即调用 fsync 系统调用函数将内核缓冲区的内容刷盘。该策略可靠性最好，但是性能最低。

everysec： Redis 在每轮事件循环的前置处理函数中，将 AOF 缓冲区中的内容通过 write 系统调用函数追加写入 AOF 文件（此时在内核缓冲区），如果距离上次刷盘超过 1s，就向 AOF FSYNC 类型的后台线程提交一个刷盘任务。everysec 是默认 AOF 刷盘策略，该策略在性能和可靠性都是适中的。

no： Redis 在每轮事件循环的前置处理函数中，将 AOF 缓冲区中的内容通过 write 系统调用函数追加写入 AOF 文件（此时在内核缓冲区），不主动调用 fsync，让刷盘时机由操作系统控制。性能最好，可靠性最低。

always 策略不能容忍 short write 情况，即无法一次性把 AOF 缓冲区的全部内容写进文件，如果出现 short write 则会直接退出进程。而其它策略都是可以容忍一次只写入部分 AOF 缓冲区内容的。

AOF 重写： 创建一个子进程，创建一个临时文件，遍历每个数据库，分析每个键值对的内容和状态，用对应的设置命令写入临时文件。临时文件写完后，用配置的 AOF 文件名将其重命名，最后关闭子进程。

AOF 需要重写的原因： AOF 文件记录了很多写命令，其中可能有很多是针对同个 key 的写命令，但是一般来说只有最后一次修改有效，所以我们记录了很多无效命令，这会导致占用很多的内存空间。所以 AOF 重写的好处就是能够节省 AOF 文件的占用空间，同时剔除了无效命令还能使恢复速度加快。

AOF 重写时机：

1. 使用 BGREWRITEAOF 命令主动执行后台重写。
2. AOF 文件占用空间到达上次重写的 2 倍，被动执行后台重写。

注：AOF 重写时如果父进程有数据写入，那么也是通过写时复制机制来保证数据安全的。还有，AOF 只有后台进程重写，没有在主进程重写的。

AOF 数据恢复过程：

1. 伪造一个 Redis 客户端。
2. 从 AOF 文件中解析一条命令
3. 使用伪客户端执行命令。
4. 重复第二，第三步直到 AOF 文件读取完毕。

AOF 的优点：

1. AOF 数据完整性较 RDB 更好。

AOF 的缺点:

1. AOF 文件记录写命令，并不使用二进制形式和压缩存储，占用空间较大。
2. AOF 机制下通过执行记录的命令来进行数据恢复，恢复速度较慢。

MP-AOF

MP-AOF (Multi Part-AOF) 是 Redis 7 推出的 AOF 机制。它将 AOF 分成了**四类文件**:

基本文件 (Base) : Base 文件是执行 AOF 重写生成的文件，有 RDB 和 AOF 两种格式，默认是生成 RDB 文件，清单文件中只会会有一个 Base 文件。Base 文件名格式为: `{配置 AOF 文件名}.{序列号}.base.aof`。(感觉基本文件叫起来很怪，所以直接叫 Base)

增量文件 (Incr) : Incr 文件是最近一次 AOF 重写后记录的所有写命令。清单文件中可能会有多个 Incr 文件，Incr 文件在 AOF 重写时创建，但如果 AOF 重写中途中止了，重试重写的时候又会创建一个 Incr 文件来记录新的写命令。Incr 文件名格式为: `{配置 AOF 文件名}.{序列号}.incr.aof`。(我们提到增量文件也不用中文名，而是说 Incr)

历史文件 (History) : 当 AOF 重写成功之后，之前的 base 和 incr 文件将成为历史文件。历史文件会通过后台线程异步删除。

清单文件 (Manifest) : 清单文件用于跟踪和管理其余类型的文件，记录格式为: `file {文件名} seq {序列号} type {文件类型首字母}`。

清单文件的内容示例如下:

```
file appendonly.aof.2.base.rdb seq 2 type b
file appendonly.aof.1.incr.aof seq 1 type h
file appendonly.aof.2.incr.aof seq 2 type h
file appendonly.aof.3.incr.aof seq 3 type h
file appendonly.aof.4.incr.aof seq 4 type i
file appendonly.aof.5.incr.aof seq 5 type i
```

MP-AOF 解决什么问题:

1. 减少 AOF 重写期间记录写命令带来的内存开销: 之前的 AOF 机制如果在 AOF 重写期间收到写命令，是将这些命令存储在内存中，待重写完成后再把这些命令追加写入 AOF 文件。那么，如果重写期间收到大量写命令的话，存储这些命令将是一笔不小的内存开销。而 MP-AOF 在 AOF 重写期间是创建了一个新的 Incr 文件，用来记录重写期间以及之后收到的写命令，于是就减少了内存开销。

2. 减少 AOF 重写时的硬盘 IO 次数: 之前的 AOF 机制为了尽量减少重写完成后，重写期间的写命令占用内存太多，所以会让执行 AOF 重写的子进程把一些重写期间的写命令写入文件，子进程完成重写任务后调用 fsync 刷盘并关闭文件，AOF 重写完成后再将剩余的命令写入重写后的文件，最后再调用一次 fsync 保证落盘。所以和 MP-AOF 在重写期间专门创建一个 Incr 文件来写入命令相比，旧机制硬盘 IO 次数更多了。

3. AOF 重写完成后不造成主进程阻塞: 之前的 AOF 机制由于将重写期间的写命令存到内存，那么重写完成后需要把内存中剩余的命令追加写入 AOF 文件并调用 fsync 刷盘，如果需要写的命令很多则可能会导致主进程阻塞。而 MP-AOF 在重写期间创建一个新的 Incr 文件来记录之后的写命令，和平时的 AOF 文件一样边写边刷盘（和配置的持久化策略有关），不会重写完成时突然要将大量命令写入文件中。

MP-AOF 和混合持久化

混合持久化：混合持久化是 Redis 4.0 推出的功能，它是在 AOF 文件里前面存储 RDB 格式数据，后面存储 AOF 格式数据。数据恢复的时候先根据 RDB 部分快速恢复，再根据 AOF 部分靠执行命令恢复数据。

MP-AOF 中的混合持久化：MP-AOF 也是可以 RDB + AOF 一起使用，但是和之前的混合持久化不同的是，RDB 和 AOF 数据分开了。RDB 作为 Base 文件，而 AOF 作为 Incr 文件。在数据恢复时，Redis 根据清单文件找出 Base 文件和 Incr 文件，先通过 Base 文件恢复数据，再通过 Incr 文件恢复数据。

(附加题) 同时开启 RDB 和 AOF，数据恢复的时候用哪个？

优先选择 AOF。如果 AOF 文件不存在，也不会通过 RDB 文件恢复。所以如果 AOF 文件丢了，想要通过 RDB 进行恢复得手动配置将 AOF 关闭。（注意 MP-AOF 重写生成的 RDB 文件是在 AOF 模式下用来恢复的，以上讲的 RDB 指的是 RDB 模式生成的 RDB 文件，默认文件名 dump.rdb）

这个问题我也问过 Redis 现任老大（Oran Agra），他的回复是不建议 RDB 和 AOF 一起开，而且 AOF 不存在却存在 RDB 文件的概率微乎其微。他们还打算把 BGSAVE 和 BGREWRITEAOF 合并，未来还可能禁用两个持久化机制同时开启，或者采用在 AOF 清单文件跟踪记录 RDB 模式生成的 RDB 快照的方案。

MP-AOF 的作者（chenyang8094，在阿里巴巴上班）也给了我答复，它个人认为原因是 RDB 恢复数据可能会有部分数据丢失（因为 RDB 持久化策略不够实时），所以没用 RDB 进行兜底。

(附加题) AOF 开启了 always 策略能保证命令一定都落盘吗？

很遗憾，并不能。首先，写 AOF 缓冲区，AOF 写文件 + 刷盘和执行命令这三样就不在同一个函数里，写 AOF 缓冲区是执行完命令之后，AOF 写文件 + 刷盘时机是每一轮事件循环的前置处理函数，虽然执行完命令到刷盘这个时间间隔很短很短，但是从理论上讲这个时候宕机还是会丢数据的。如果说已经调用 write 写入内核缓冲区了，操作系统没挂后面还能让操作系统自动 fsync 刷盘，但是调用 write 之前已经挂了就没救了。同时不止是考虑这个时间间隔，fsync 之后就事情交给硬件了，有没有成功落盘也要考虑硬件因素。

记忆要点：RDB，内存快照，向后兼容，存储形式，生成时机，如何存储，写时复制，RDB 数据恢复过程，优点缺点，AOF，写入时机，刷盘时机，AOF 重写，AOF 重写时机，AOF 优点，AOF 缺点，AOF 数据恢复过程，优点缺点，MP-AOF，四类文件，基本文件，增量文件，清单文件，历史文件，混合持久化

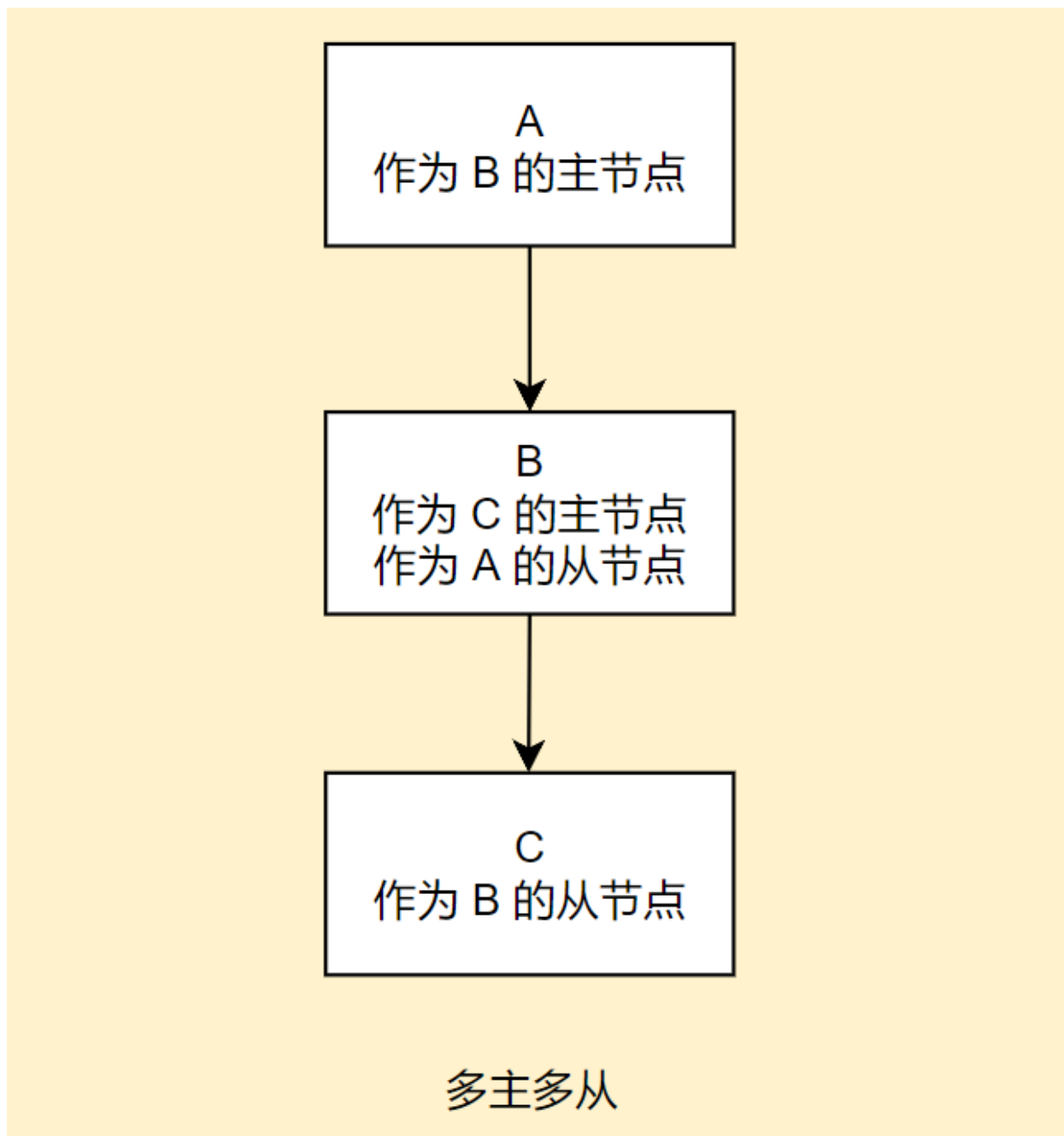
22. Redis 7 主从模式

关于 Redis 的主从模式，一般有两类问题：**主从架构和主从同步**。其中主从同步属于比较深的问题。本节重点放在 Redis 的主从同步上，而且我们会讲讲 Redis 7 的全局共享复制积压缓冲区。

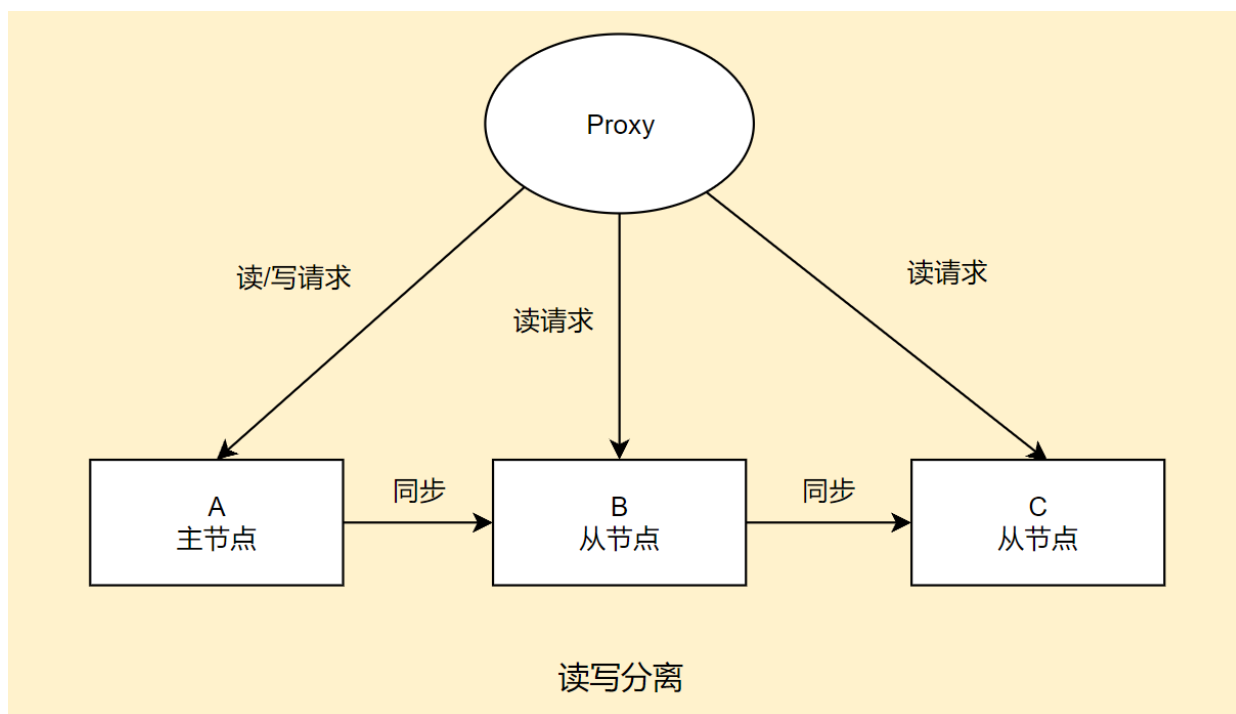
主从架构

主从架构个人认为可以分为三种：**一主一从，一主多从，多主多从**。

一主一从和一主多从很好理解。多主多从指的是：一个节点它可能不仅是主节点还是从节点。多主多从具体架构还可以分成多种方式，Redis 支持的方式如下图所示：



读写分离架构也是基于主从架构的，一个主节点负责对外提供读写服务，或者只提供写服务，其余从节点提供读服务。Proxy 负责根据各个节点负载情况将请求合理分配给各个节点。但是要注意读写分离架构下，存在数据不一致问题。



主从同步

Redis 的主从同步有两种方式：**全量同步**，**部分同步**。它们对应着使用 PSYNC 命令发起同步时的两种响应 **+FULLRESYNC** 和 **+CONTINUE**。

全量同步：全量同步指使用 **RDB + AOF 进行同步**。首先使用 RDB 同步，我们也可以叫它快照同步。主节点将数据库当前状态打包成 RDB 快照发送给从节点，从节点收到 RDB 文件后将 RDB 文件存入硬盘（有盘加载），之后清空数据库，再加载 RDB 从而完成一次快照同步。随后进行部分同步，把全量同步期间的命令全部加载上，这样一次全量同步就完成了。部分同步后面再进行说明。

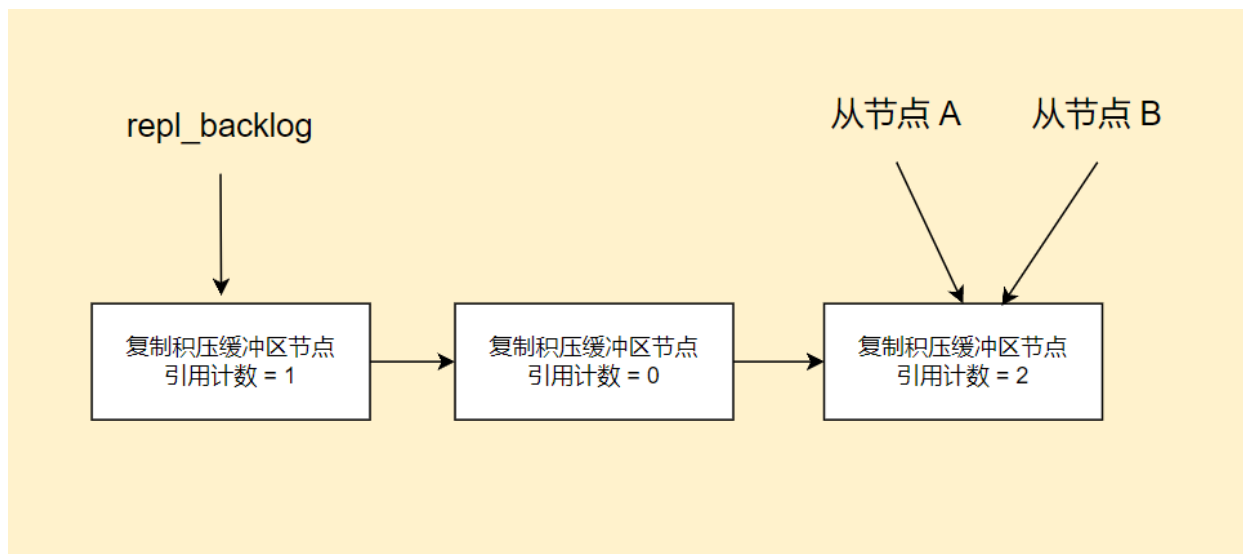
无盘传输：Redis 从 2.8 版本开始支持无盘传输。如果没有无盘传输进行快照同步的话，需要主节点先将 RDB 文件保存到本地硬盘，这样的全量同步对主节点负载可能很高，对主从同步效率和对外服务效率都有很大影响。无盘传输就是主节点一边遍历数据库，一边将数据序列化 RDB 格式发送给从节点，不需要先将 RDB 文件保存在本地。

无盘加载：与无盘复制思想类似，从节点在内存建立一个临时数据库，通过读取 socket 将接收到的 RDB 格式数据写入临时数据库，当读取到结束符时表明 RDB 文件已接收完毕。接着将临时数据库替换为正式的数据库，最后将原本的数据库释放掉。

部分同步：Redis 主节点维护了一个复制积压缓冲区，主节点将写指令记录在复制积压缓冲区中，并且异步地将缓冲区的指令同步到从节点。主节点会为每个从节点维护一个复制偏移量，同时也记录复制积压缓冲区的起始复制偏移量（offset）以及最大复制偏移量（histlen，源码里的相关字段名）。复制积压缓冲区的数据范围为起始复制偏移量 ~ 最大复制偏移量，如果从节点的复制偏移量在该范围内，则可以进行部分同步。部分同步是本篇的**重点**，接下来就讲讲 7.0 之前和 7.0 的复制积压缓冲区都是什么样的。

环形复制积压缓冲区：Redis 7.0 以前使用的复制积压缓冲区是一个定长环形数组，环形指的是如果内容写满了，则会回到头部，从头覆盖写入的新数据。

全局共享复制积压缓冲区：Redis 7.0 使用了一个全局共享复制积压缓冲区，它的结构是一个链表，节点是一个复制积压缓冲区块，如果你觉得全局共享复制积压缓冲区这个名字太长不好听，那么我觉得你也可以把它记作“链式复制积压缓冲区”，如下图所示：



和环形复制积压缓冲区不同，它将整个复制积压缓冲区分块，并且变成链表结构，在旧节点装不下要记录的新命令时，会创建新的节点来记录新命令，不会再覆盖之前的数据，节点规定最小创建 16 KB 的空间。

每个节点还需要记录**被引用的次数**，我们叫它**引用计数 (refcount)**。repl_backlog（之前是个 char*，表示环形复制积压缓冲区，现在是个记录缓冲区头节点和缓冲区元信息的结构体）一直引用第一个节点，所以第一个节点引用计数至少是 1，从节点会增加它需要复制的数据所在节点的引用计数。**这个引用计数在释放缓冲区空间时很重要**，当缓冲区空间超出限制时，会尝试从头开始释放最多 64 个节点。当检查到节点的引用计数为 1 时，则表示除了 repl_backlog 没有其它从节点在引用这个节点，于是就可以将它释放，腾出内存空间。

全局共享复制积压缓冲区还使用了 rax 树做节点稀疏索引，用来给从节点快速查找复制偏移量对应的缓冲区块。rax 树为每 64 个节点建立一个索引，索引为每 64 个节点中第一个节点的起始复制偏移量。

全局共享复制积压缓冲区的优势：

1. 节省大量内存空间： 环形复制积压缓冲区在给从节点发送数据前，会将数据复制到主节点为每个从节点维护的输出缓冲区（output buffer），然后再将输出缓冲区的内容写到 socket 缓冲区发送。而全局共享复制积压缓冲区不需要再复制一遍，它直接将缓冲区节点的数据写入 socket 缓冲区。相当于以前是每个从节点一个 buffer，现在是全局只需要一个 buffer。

2. 提升性能： 原理还是跟上面提到的一样，因为每个从节点不需要再复制一次缓冲区数据，所以也提升了主从同步的性能。

部分同步条件：

1. 主节点持有的复制 id 和从节点持有的复制 id 相同。（相同表示主节点和从节点拥有相同的数据集，只是从节点可能数据落后了一些）

2. 复制积压缓冲区存在。

3. 复制偏移量小于当前复制积压缓冲区的起始偏移量。

4. 复制偏移量大于当前复制积压缓冲区的总数据量。

当满足以上条件时，将复制积压缓冲区从复制偏移量开始的所有命令发送给从节点。如果不满足则进行全量同步。

记忆要点：主从架构，三种，一主一从，一主多从，多主多从，主从同步，两种，全量同步，部分同步，无盘传输，无盘加载，环形复制积压缓冲区（7.0 前），全局共享复制积压缓冲区（7.0），链表结构，节点引用计数，rax 树索引，全局共享复制积压缓冲区优势，节省大量内存空间，提升性能

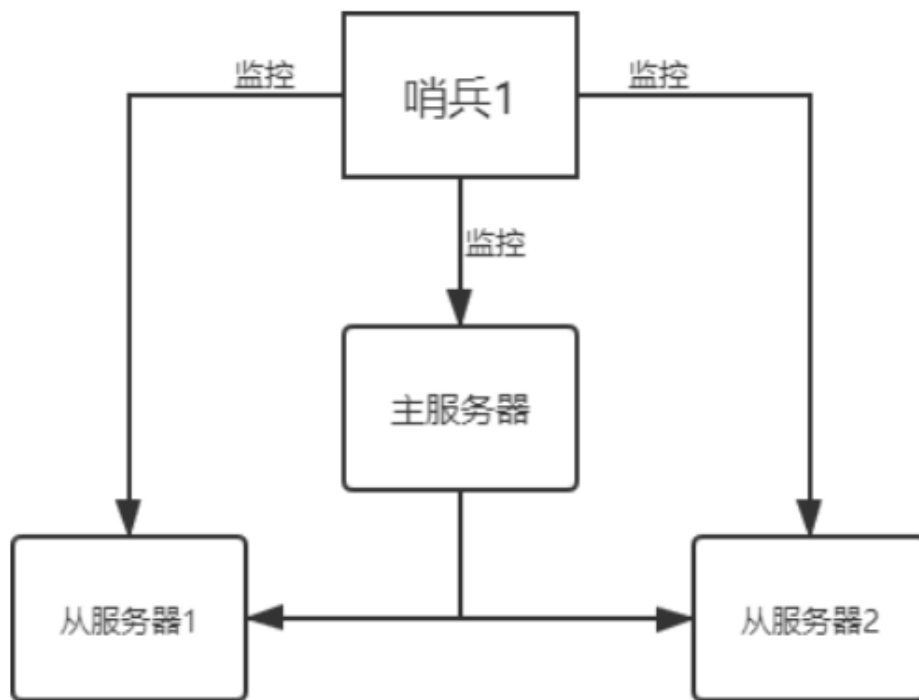
23. Redis 7 哨兵模式

哨兵模式（Sentinel）是在主从模式的基础上，为了保证 Redis 主从集群的**高可用性**而推出的功能。该功能可以让 Redis 服务器成为哨兵，哨兵的职责是**监控主从节点状态，当主节点宕机时进行故障转移**，也就是选举出一个从节点成为新的主节点。关于 Redis 的哨兵模式，一般有两类问题：**哨兵架构和哨兵机制**。其中哨兵机制属于比较深的问题。本节重点放在 Redis 的哨兵机制上。

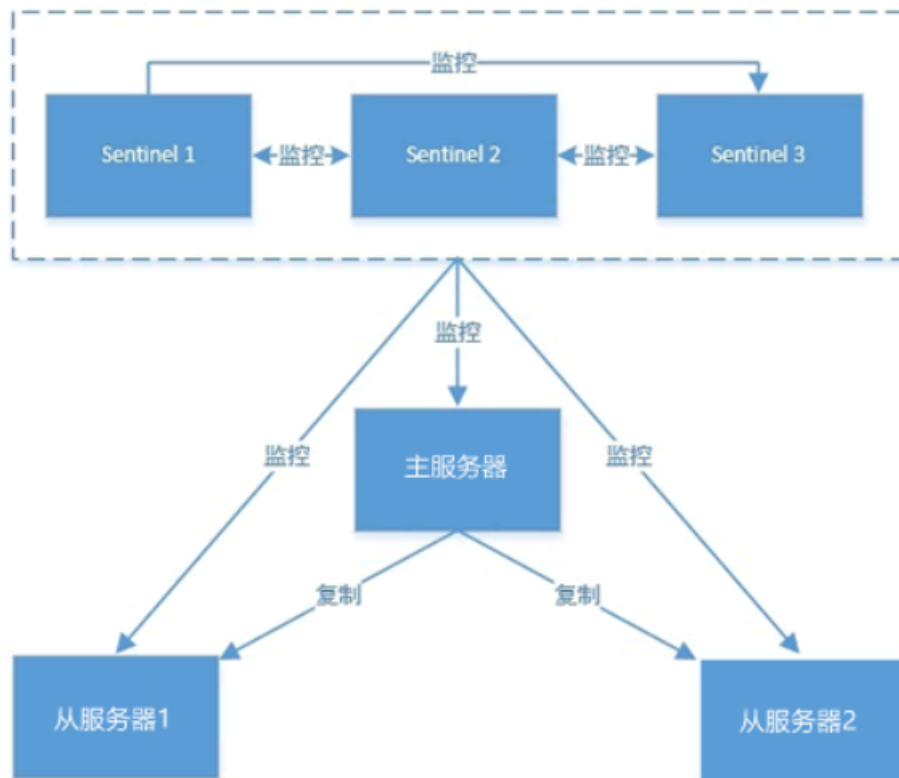
哨兵架构

哨兵架构有两种：**单哨兵架构和多哨兵架构**。

单哨兵架构：一个哨兵负责监控主从节点状态。



多哨兵架构： 多个哨兵负责监控主从节点和其它哨兵状态，多哨兵是为了保证哨兵集群的高可用性。



在哨兵架构下，客户端首先连接哨兵节点，哨兵节点负责将主节点信息告知客户端，随后客户端就与主节点建立连接开始通信。

(以上图片来自网络，出处：<http://c.biancheng.net/redis/sentinel-model.html>)

哨兵机制

角色：哨兵机制一共有三种角色：**哨兵，主节点，从节点。**

心跳检测：哨兵通过每秒向监控的节点发送 PING 命令的方式来检测节点的存活状态，这种方法也被称为心跳检测。

主观下线：哨兵主观认为节点下线，标记该节点状态为 SDOWN。

当满足以下三个条件之一时，哨兵主观认为节点下线：

1. 哨兵没有收到节点回复的时间超过了 `down_after_period`（默认 30s），则主观认为对方下线。
2. 哨兵认为某个节点是主节点，但是它报告自己是从节点，如果这种状态持续 `down_after_period`（默认 30s）+ 2 * `sentinel_info_period`（默认 10s）（默认总共 50s），则主观认为对方下线。
3. 节点在进行故障恢复，但是该过程超过了 `master_reboot_down_after_period`（默认 180s）时间限制，则主观认为对方下线。

客观下线：客观下线指有 `quorum` 数量的哨兵都认为主节点主观下线后，则把它标记为客观下线（**ODOWN**）。`quorum` 可以配置，但不能小于哨兵数 / 2 + 1，默认值为哨兵数 / 2 + 1，即超过半数哨兵认为主节点下线就成为客观下线。客观下线只适用于主节点，对于从节点和其它哨兵节点认为它们下线不需要进行协商。

故障转移流程：当主节点被认为客观下线时，哨兵需要选出一个从节点成为新的主节点，这就是故障转移。**故障转移共有五个步骤：**

1. 选举哨兵 Leader：首先需要监控同个主节点的哨兵中选举出一个哨兵 Leader 负责接下来的故障转移步骤。已经认为主节点客观下线的哨兵会作为候选者参与选举，候选者请求其它哨兵为自己投票，每个哨兵只有一票，所以只会为第一个请求自己的候选者投票。候选者请求完其他哨兵投票后，如果自己没有投过票，会投给得票数最多的候选者，如果还没有其它候选者得票则投给自己。投票结束后，**得票最多的候选者，且得票数 >= quorum 且超过半数（哨兵数 / 2 + 1），会成为哨兵 Leader**，将负责接下来的故障转移工作。

2. 选举主节点：哨兵 Leader 负责选举出一个从节点作为新的主节点。首先过滤掉已经被认为下线的，网络状况不好的（默认是超过 5s 没有收到回复的），优先级设置为 0 的从节点，剩下的从节点参与选举，选举过程如下：

第一轮：将优先级最大的从节点筛选出来。（优先级可以手动配置，默认值 100）

第二轮：选择复制偏移量最大的从节点。因为复制偏移量越大则表示数据集越新。

第三轮：选择复制 `run_id` 最大的从节点。（`run_id` 可以手动指定，未指定则随机生成）

最后，经过以上条件筛选出来的一个从节点，就是选举出来的新的主节点。

3. 将选举出来的从节点转变为主节点：令选举出来的从节点执行 `SLAVEOF NO ONE` 命令，将它转变为主节点。

4. 检测故障转移是否超时：如果故障转移用时超过了 `failover_timeout`（默认 180s），则中止本次故障转移。

5. 将新主节点设置成其余从节点的主节点：最后一步，让其余从节点调用 `SLAVEOF` 命令将选举出的主节点设置为自己的主节点。

记忆要点：哨兵架构，单哨兵，多哨兵，哨兵机制，三种角色，主节点，从节点，哨兵，心跳检测，主观下线，客观下线，故障转移流程，五个步骤，1. 选举哨兵 Leader，2. 选举主节点，3. 将选举出来的从节点转变为主节点，4. 检测故障转移是否超时，5. 将新主节点设置成其余从节点的主节点，重点在选举流程

24. Redis 7 集群模式

集群模式 (Cluster) 是高并发场景下常用的一种模式，同时集群模式也是主从，哨兵，集群三种模式里最难的，不仅细节多，代码量也最多。当然，我不会带着大家啃一堆源码，而是把关于集群模式的基本概念和重点内容尽量用简要的文字提炼给大家，不过本篇内容还是很长，有一万字左右，如果你没有耐心看完，那么起码要了解一下集群的一些概念，关于机制中讲过程的部分可以略过。本篇内容将分成**集群模式简介（包括架构）**和**集群模式机制**两个主题对集群模式进行介绍。

集群模式简介

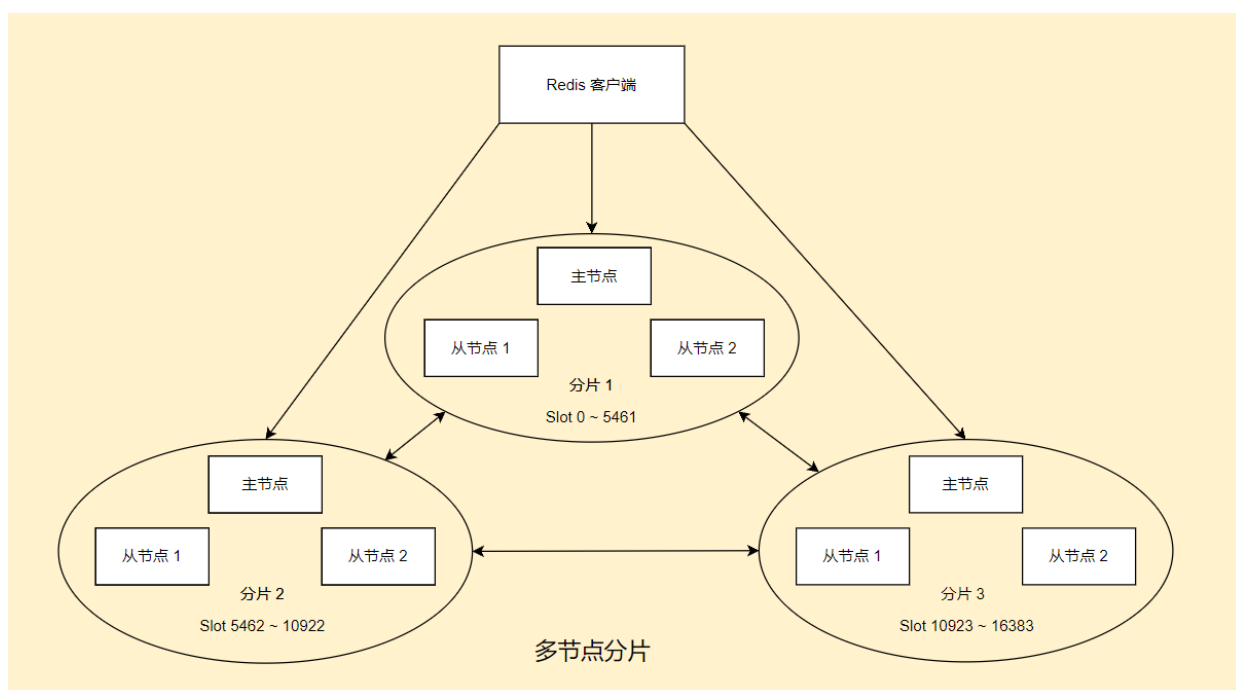
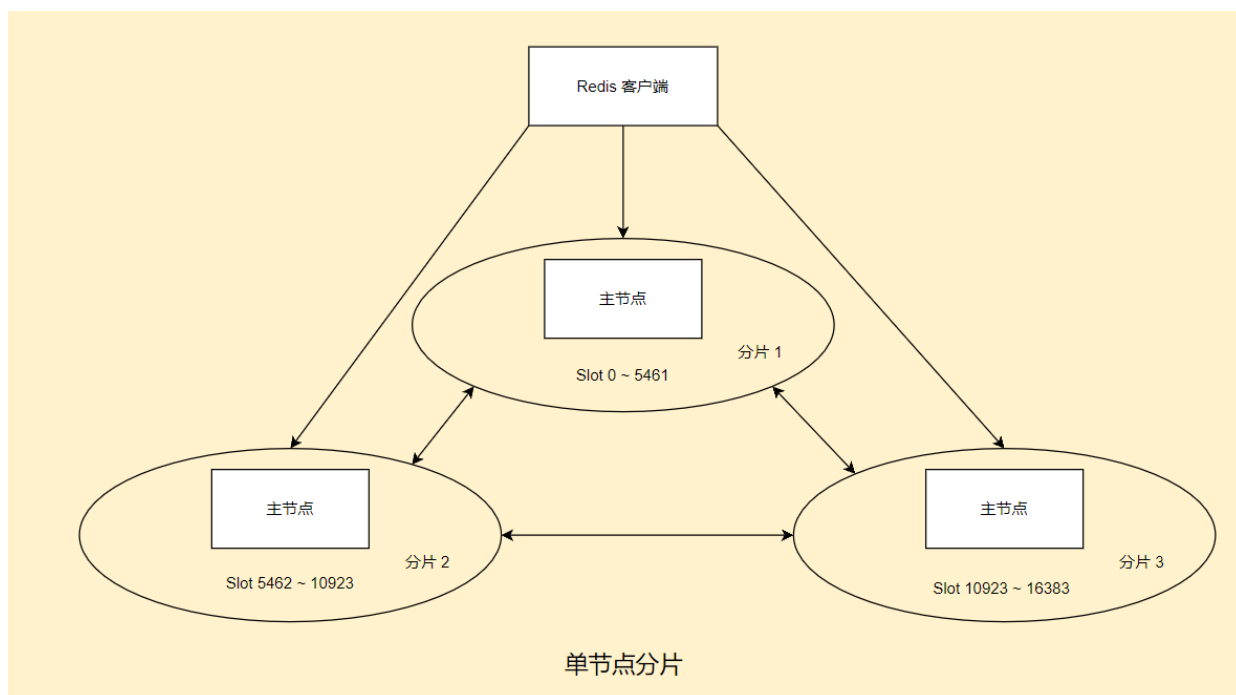
集群模式： 集群模式指的是 Redis Cluster 模式，集群模式可以让多个节点提供读写服务，并且是去中心化的，不需要选举出 Leader 来协调集群工作。集群模式有两个重要概念：**分片，哈希槽**。

分片(shard)： 集群中提供读写服务的节点。

哈希槽(slot)： 集群模式共有 **16384** 个 slot，集群中的 key 会通过 CRC16 算法计算出一个值并和 16384 取模 ($\text{slot} = \text{CRC16}(\text{key}) \& 16383$)，结果即为 key 所属的 slot 编号。每个分片负责管理部分 slot，slot 和分片是一对一分配的，分片提供的是自己管理的 slot 的读写服务。（不用中文而用英文 slot 是因为这个术语还挺常用的，希望大家也记住这个单词）

为什么需要集群模式： 集群模式和之前提过的架构相比，之前的都是只有单节点对外提供写服务，如果想要提升性能和存储能力只能垂直扩展，也就是升级机器硬件，但是一般来说成本昂贵且提升上限不高。而且每个节点需要存储所有的 key，导致每个节点存储压力很大，容易内存不足。**而集群模式每个分片都能对外提供读写服务，并且管理不同的 key，不仅提升了 Redis 分布式集群的性能，也降低了每个节点的存储压力。**集群模式提升性能和存储能力的方法一般是添加分片，即**水平扩展**。水平扩展方式在集群节点数量不多的情况下，集群节点之间的通信开销不大，提升性能是比垂直扩展成本要低而且效果更好的。

集群架构： 集群架构一般都是指多分片，只有单分片是算不上“集群”的。可以将**分片是单节点还是多节点（主从关系）**分成两种集群架构，如下图所示：



给多节点分片做成读写分离虽然理论上也可以，但一般不需要这么做。因为在集群模式下，我们可以通过添加主节点来提升整个集群的读写能力。就算你真的想做读写分离，Redis 自带的集群模式也不会去把槽位对应到多个节点呀~只能找找有没有支持的第三方集群或者自己实现了。不过事实上，确实没听说过哪家在用 Redis 原生 Cluster 的。。。

集群模式机制

Redis 集群是一个全网状结构，其中每个节点都使用 TCP 连接与其他每个节点相连。在一个由 N 个节点组成的集群中，每个节点都有 N-1 个传出连接和 N-1 个传入连接。

请求重定向： 集群模式下，客户端可以将命令发送给任意的分片（主节点），如果对应的 key 不属于该分片，会将 key 对应的 slot 所属分片的地址告知客户端，客户端接着就可以向正确的分片发送请求。这就是请求重定向。

请求重定向过程：

(Redis 的客户端有很多种，每种客户端可能实现都有差异，以下举例的是有缓存槽位和分片关系的映射表的客户端)

1. 客户端将 key 用 CRC16 算法计算并将结果和 16384 取模，得到 key 所属的 slot，然后在槽位映射表中找到该槽位对应分片的地址，之后向该分片发送命令。

2. 分片在收到命令后，先解析命令中的 key，将 key 经过 CRC16 计算和 16384 取模得出 slot，通过分片自己的 slots 缓存找出管理该槽的分片。如果发现自己并不是管理该槽的分片，那么就会回复客户端 `-MOVED` 开头的消息，`-MOVED` 后边跟着 slot 以及管理该槽分片的地址 (IP + Port / Hostname + Port)。

3. 客户端收到 `-MOVED` 开头的回复后，解析出分片地址，用回复的分片地址更新槽位映射表，最后向该分片地址发送命令。

数据迁移： Redis 集群支持数据迁移，数据迁移的基本单位是 slot，分片可以将自己的槽位迁移到其它分片。处在数据迁移过程中的源分片 slot 状态为 **migrating**，而目标分片 slot 状态为 **importing**。

对正在数据迁移的分片请求重定向过程：

对正在进行数据迁移的分片发送命令，可能 key 未被迁移，也可能已经迁移到其他节点了，所以这是一个特殊情况，以下是对正在数据迁移的分片的请求重定向过程：

1. 客户端将 key 用 CRC16 算法计算并将结果和 16384 取模，得到 key 所属的 slot，然后在槽位映射表中找到该槽位对应分片的地址，之后向该分片发送命令。

2. 分片在收到命令后，先解析命令中的 key，将 key 经过 CRC16 计算和 16384 取模得出 slot，通过分片自己的 slots 缓存找出管理该槽的分片。如果槽位是自己的，但是发现槽位处于正在迁移状态 (migrating)，且无法查找到 key，那么此时并不能清楚是 key 本身就不存在，还是 key 已经迁移到目标分片了，所以需要让客户端到目标分片去问问看。于是这回回复客户端 `-ASK` 开头的消息，`-ASK` 后面跟着 slot 以及**目标分片的地址 (IP + Port / Hostname + Port)**。

3. 客户端收到 `-ASK` 开头的回复后，解析出分片地址，然后先向目标分片发送一个 `ASKING` 命令，再发送原先的命令。这里先发送 `ASKING` 是告诉目标节点要接受下一条命令关于 importing 状态槽位的查询。如果没有 `ASKING` 通知分片，那么对未迁移完毕的槽位进行查询只会回复 `MOVED`，并告知客户端去源分片查询。这是因为未迁移完毕的槽位仍属于源分片，那么这样一来就会出现重定向循环。(注意这里并不会更新本地槽位映射表)

gossip 协议： Redis 集群模式使用的是去中心化，弱一致性的 gossip 通信协议。gossip 的中文意思是八卦，流言蜚语。gossip 协议也和流言一样，一传十，十传百。在需要传递消息到整个集群时，每个节点将消息传播给自己已知的节点，通过这种像流言一样的扩散方式，最后消息会传播给所有网络正常的节点。正常情况下，每个节点都能认知到其余所有节点 (其余所有节点都能成为自己的已知节点)。

心跳检测： Redis 集群节点不断地交换 ping 和 pong 数据包。这两种数据包具有相同的结构，都携带重要的配置信息，唯一的区别是消息类型字段。我们将 ping + pong 数据包称为一组心跳数据包。但是 pong 包也能主动发送，并且不需要回复，比如能够用来主动通知集群配置变更。每个节点都会尝试在 `NODE_TIMEOUT` (认为节点下线的超时时间) 的一半时间内对所有已知节点发送 ping，例如在超时时间设置为 60 秒的 100 节点集群中，每个节点将尝试每 30 秒发送 99 个 ping，总计每秒 3.3 个 ping。乘以 100 个节点，就是整个集群中每秒 330 次 ping。所以，超时时间和 ping 频率正相关，需要根据业务背景进行合理配置。

心跳数据包内容： 集群的 ping 和 pong 包都具有集群消息均通用的首部，首部包含的信息如下：

- 节点 ID，一个 160 位的伪随机字符串，在节点第一次创建时分配，并在集群节点的整个生命周期中保持不变。
- 发送节点的 currentEpoch 和 configEpoch 字段，这些字段后面会介绍。如果该节点是一个副本，configEpoch 是其主节点的最后已知 configEpoch。
- 节点标志，表明该节点是从节点，还是主节点，或者其它关于节点状态的信息。
- 发送节点提供的 slot 位图，如果该节点是从节点，则为其主节点的 slot 位图。（slot 位图指用位图结构记录节点所管理的插槽，有 16384 个比特位（2 KB），比特位为 1 则表示插槽由该节点管理）
- 基本端口，即发送节点用于监听客户端命令的端口。
- 集群端口，即发送节点监听集群消息的端口。
- 从发送节点的角度看集群的状态，比如是故障还是正常。
- 主节点 ID（如果发送节点是从节点会携带该数据）

除了通用首部之外，ping 和 pong 包还具有一个特殊的 **gossip 字段**，该部分表示的是发送节点对集群中其它节点的主观看法，具体指的是：发送节点会从已知节点集合中随机选出部分节点，并在 gossip 字段附上它们的信息，包括（但不仅限于）节点 ID，节点 IP 和端口号，节点标志信息。而这个随机选出的部分节点数量与集群节点总数正相关，最大数量是总数的 1/10。

以上内容不需要记得太详细，我们只要知道 ping 和 pong 包**不仅携带了自身信息，还携带了部分其他节点信息**就好了。

集群消息类型： 集群消息类型目前有 12 种，我们挑 **meet, ping, pong, fail, update** 五种类型来对它们进行简要说明。

1. meet: 新节点加入集群时使用。当执行 `cluster meet ip port [cluster-bus-port]` 命令之后，执行端会向 ip:port 指定的地址发送 meet 包，相当于邀请目标节点加入集群。当目标节点收到 meet 消息时，目标节点尚未认识发送命令的节点。所以为了让目标节点强制接受它作为可信节点，它发送一个 meet 包而不是一个 ping 包。这两个数据包具有完全相同的格式，但 meet 强制接收方确认发送方是可信的。

2. ping: 用于心跳检测，集群节点握手等场景。主要作用是传播节点的状态信息，让接收者能够知道其他节点的状态并进行缓存。

3. pong: 用于心跳检测，回复 meet 包，主从角色切换时主动通知其它节点等场景。主要作用和 ping 包一样，仍然是传播节点的状态信息。

4. fail: fail 包用来通知集群某个节点处于故障状态，当一个节点被大多数节点标记为 pfail（疑似下线，可以理解为哨兵机制的主观下线）状态时，会进入 fail 状态。当一个主节点进入 fail 状态后，将进行故障转移，选出一个从节点成为新的主节点。

5. update: 更新节点配置信息时，通知其它节点使用的消息类型。包含配置纪元（Config Epoch），节点 ID 和 节点所管理 slot 的位图信息。

新节点加入过程：

1. 首先需要集群中的某个节点对新节点发出邀请，所以要在集群中的任意一个节点执行 `cluster meet ip port [cluster-bus-port]`，ip port 填入新节点的 ip 和端口号。执行后，该节点会为新节点创建一个 clusterNode 结构，并加入到节点信息哈希表中（用于记录其它集群节点信息），最后向新节点发送 meet 包，**开始节点握手阶段**。

2. 新节点收到 meet 包，为发送节点创建一个 clusterNode 结构，并加入到记录其他节点信息的哈希表中。此时除了 IP 和端口信息之外，并不会用消息头内容更新发送节点的其它信息，对于记录的发送节点 ID 也是暂时为它随机生成的，直到将来收到该节点的 pong 消息才会更新节点信息。之后给发送节点返回 pong 消息，消息头携带着关于自己的元信息。

3. 发起邀请的节点收到 pong 包，确认了新节点已经收到 meet 包，于是用消息头的元信息更新该节点信息，接着再发送一个 ping 包给新节点。

4. 新节点收到 ping 包，确认了对方收到了 pong 包，至此**节点握手阶段完成**。

5. 发起邀请的节点之后就会在 ping 其它节点的时候，有概率在 gossip 字段携带新节点信息。当其它节点拆包发现有未知的节点信息，但是是已知节点发送过来的，已知节点的消息是可信的，所以便会向新节点发起握手。最后，新节点会被其它所有节点认知。

删除节点：删除节点可以分为三种：**删除主节点，删除从节点，删除分片**。

删除主节点：我们要删除一个具有从节点的主节点时，一般情况下是通过客户端连接它的从节点，然后执行手动故障转移命令 `cluster failover`。手动故障转移的好处是能够避免数据丢失，因为从节点会先告诉主节点暂停服务，然后主节点告知从节点此时的复制偏移量，从节点等待复制数据直至复制偏移量与主节点的复制偏移量相等，然后从节点才开始进行故障转移。要注意的是，如果想要让从节点成功切换为主节点，从节点需要已经被超过半数的节点认知，这样才能通过故障转移选举的最低票数限制。故障转移的具体过程我们放到后边再讲。

删除从节点：一般通过客户端执行 `del-node ip:port <node-id>` 命令删除从节点，`ip:port` 可以为集群中任意节点的 IP 和端口号，`<node-id>` 为被删除节点 ID。客户端会向除了被删除节点之外的节点发送 `cluster forget <node-id>` 命令，令它们删除关于被删除节点的记录。`del-node` 也可以删除主节点，但前提是主节点是空的，不被分配任何 slot。

`cluster forget` 命令具体会让执行节点将被删除节点加入黑名单，持续时间为 1 分钟，并将节点从自己的节点信息字典中删除。该节点在处理来自其他节点的心跳包中的 gossip 部分时，将跳过所有在黑名单中的节点 ID。删除节点会在黑名单存活 1 分钟时间，这样就有了 1 分钟的时间窗口来让客户端通知其它节点删除指定节点。

删除分片：在我们正常删除分片之前，需要将分片管理的哈希槽迁移到其它分片，然后使用 `del-node` 将空的主节点和从节点删除。重新分片可以使用客户端提供的 `reshard` 命令，具体用法可以查看官方文档：<https://redis.io/docs/management/scaling/#learn-more>。

故障转移：默认情况下，集群节点发现某个节点超过 15s 未交换过数据，就会将它标志为 **pfail（疑似下线）** 状态。当**超过半数节点**标记同个节点 pfail 状态时，就会将它标志为 **fail（确定下线）** 状态。如果该节点是主节点，那么就会触发故障转移。除此之外，还可以通过向从节点发送 `cluster failover` 命令进行手动故障转移。在讲故障转移具体流程前，我们先来了解两个重要的东西：**集群当前纪元和配置纪元**。

集群当前纪元（Current Epoch）：Redis 中的 epoch 相当于 Raft 中的 term（让了解 Raft 算法的人联想一下哈，不了解也没关系），它为事件提供增量版本控制。当多个节点提供相互冲突的信息时，另一个节点可以了解哪个状态是最新的。在创建节点时，每个 Redis 集群节点都将自己的 current epoch（因为纪元很绕口所以用英文）初始化为 0，每次从其它节点接收到数据包时，如果发送方的 current epoch 大于自己的 current epoch，则将自己的 current epoch 更新为发送方的 current epoch。集群中所有节点的 current epoch 最终会达成一致，相当于对集群状态的认知达成了一致。当集群的状态发生变化并且寻求其它节点同意以执行某些操作时，将使用 current epoch，以确保集群变化的状态是最新的，目前仅在故障转移选举中使用。总的来说，**集群当前纪元相当于集群的版本号，用于保证节点之间的数据同步和故障转移的正确性**

配置纪元 (Config Epoch) : config epoch 记录的是 slot 的最新配置时刻, 保证集群能够正确记录 slot 当前属于哪个分片。集群消息首部会携带节点的 config epoch 以及管理的 slot 信息, 接收方会通过消息更新自己记录的发送方的 config epoch 和管理的 slot 信息。接收方会记录每个 slot 对应的节点信息, 然后通过节点信息就能间接得到 slot 对应的 config epoch。Config epoch 在从节点故障转移选举胜出时, 将值更新为最新的 current epoch。config epoch 是保证递增的, 而且每个分片的 config epoch 在集群里是唯一的, 但是由于有些手动操作会在不需要集群达成共识情况下增加 config epoch, 网络分区导致形成多个集群等原因, 可能两个分片的 config epoch 会相同, 这时就产生了 config epoch 冲突。Redis 对 config epoch 冲突的解决方案是: 比较两个冲突节点的节点 ID, 令 ID 更小的节点把 current epoch + 1, 并且 config epoch 更新为最新的 current epoch。(选 ID 最小的没什么说法, 只是为了方便解决冲突)

注: 虽然我们说上面两个 epoch 代表最新状态, 但是因为 gossip 协议不是强一致性的, 只能保证最终一致性, 所以每个节点所记录的 epoch 只是它所知的最新状态, 不一定是集群里最新的状态。

故障转移流程:

1. 当集群节点把故障节点标记为 fail 状态时, 立即向已知的其他节点广播故障节点 ID, 强制其他节点将故障节点标记为 fail 状态。

2. 集群从节点会在每 100 ms 一次的集群定时任务中进行检查, 如果发现主节点为 fail 状态且集群没有配置不允许自动故障转移, 或者从节点需要执行手动故障转移, 主节点有管理的哈希槽, 那么可以进行故障转移。首先从节点检查自己是否满足故障转移的条件, 如果是自动故障转移, 检查上次与主节点通信(与主节点链接未断开)或者与主节点断开链接的时刻是否过早, 如果过早就说明从节点数据集可能过旧, 不能切换为主节点。不过只会是与主节点断开链接的时刻出现过早情况, 因为如果和主节点链接未断开说明还未超时呢, 所以在都是默认配置的情况下, 可以大概说是主从节点断开链接时间超过 175 秒, 则无法参加故障转移。

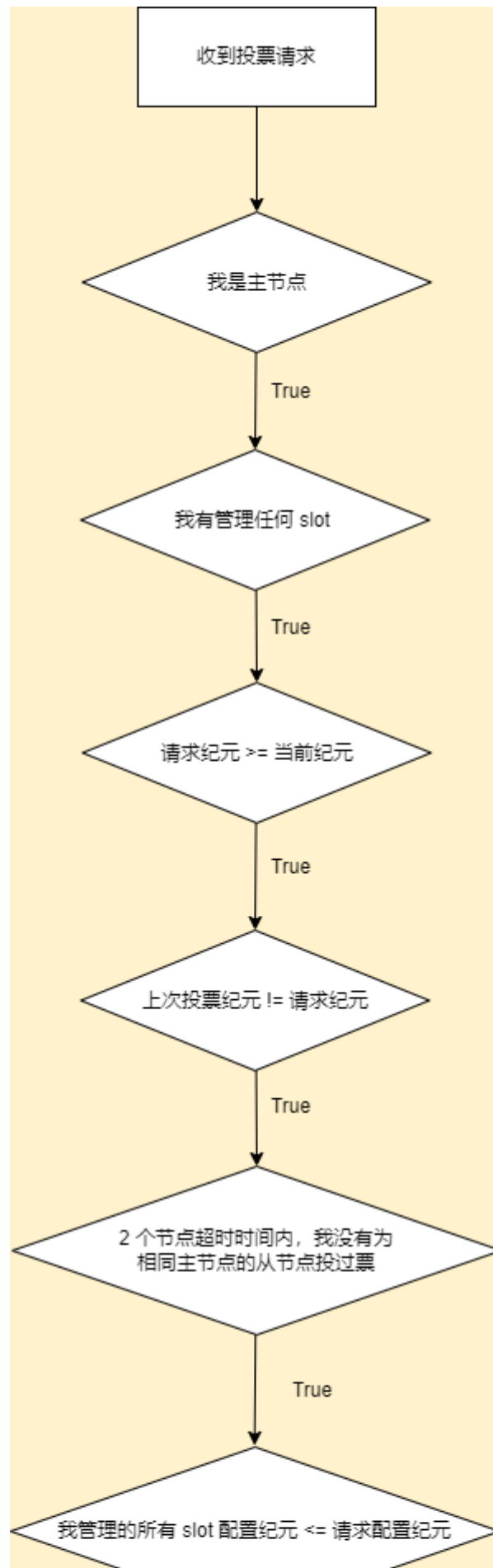
(计算方法: 断开链接时间 - 超时时间 (默认 15s) > 主节点定期 ping 从节点时间 (默认 10s) + 超时时间 (默认 15s) * 有效因子 (默认 10))

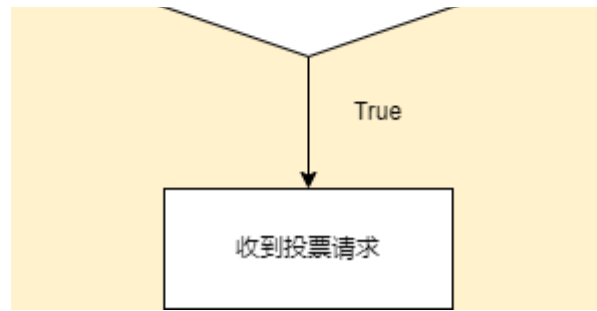
3. 在发起选举前, 如果是自动故障转移, 那么从节点要先获得一个排名, 这个排名是故障节点的所有从节点的复制偏移量排名。如果有 N 个从节点, 那么排名范围就是 0 ~ N, 复制偏移量越大的则数据集越新, 排名就会越靠前。根据排名来决定谁先发起投票, 如果网络延迟相同, 那么最先发起投票的从节点最有可能获得最多票。当然有可能所有节点复制偏移量都相同, 那么排名都是 0, 同时发起选举有可能会出现平票, 导致本次投票作废。为了减少出现平票的概率, 所以发起选举的时间还会加上随机延迟, 随机延迟的范围在 0 ~ 500 ms。发起选举时间的公式为: `mstime() + 500 + random() % 500 + rank * 1000`, 单位为毫秒, mstime() 为当前毫秒时间戳, 后面的 + 500 是故意延迟一会儿让 fail 消息传播。因为有可能记录的其它从节点信息不是最新的, 这一轮计算的排名就不够准确, 所以在计算完排名后节点会向其它从节点广播自己的最新信息, 在节点发起选举之前, 如果收到其它从节点的最新复制偏移量, 则会计算出一个自己的最新排名, 并重新计算选举发起时间。等到当前时间大于等于选举发起时间时, 就可以发起选举了。如果是手动故障转移, 那么不需要排名, 直接指定执行手动故障转移的从节点发起选举。

4. 接下来终于到**选举阶段**了。首先从节点将 current epoch + 1, 再让当前选举纪元 (failover_auth_epoch) = current epoch, 随后广播投票请求给其他节点。

5. 当集群中的节点收到请求时, **先检查自己是不是主节点或者没有管理任何槽位, 否则没有投票资格。**接着检查请求纪元 (请求方的当前纪元) 是否小于自己的 current epoch, 小于则说明不是最新的投票, 直接返回, 否则继续。然后检查上次投票纪元是否和请求纪元相同, 如果相同则表示已经给请求方投过票了, 直接返回, 否则继续。接着我们判断 2 个节点超时时间内, 是否已经为相同主节点的从节点投过票了, 已经投过就返回, 否则继续。接下来是最后一个判断, 判断当前节点所管理槽位的 config epoch 是不是都小于等于请求方 config epoch, 如果是, 就可以给请求方投上我们宝贵的一票了。

emm...是不是判断太多了，文字看得有点晕？那我画个流程图吧：





6. 从节点检查自己的得票数，如果超过半数（集群节点数 / 2 + 1），则认为自己胜出。可以将自己切换为主节点了。

7. 从节点将自己设置为主节点，并将原来的主节点管理的 slot 分配给自己，然后更新集群状态并将最新集群配置持久化到硬盘，接着向所有节点发送一个 pong 包，让其他节点更新集群状态，最后清除手动故障转移信息（对自动故障转移无效果）。

副本迁移： Redis 为了提高集群模式的**可用性**，实现了自动的副本迁移。当集群出现孤儿主节点时（没有从节点的主节点），会从具有最多从节点的主节点旗下迁移一个从节点给它。当然，选中的主节点不能迁移出从节点后，自己就变成孤儿主节点了，所以至少有 2 个从节点才能将副本迁移出去。

记忆要点： 集群模式简介，分片 (shard)，哈希槽 (slot)，为什么需要集群模式，各分片提供读写和存储能力，水平扩展，集群架构，单节点分片，多节点分片，集群模式机制，请求重定向，请求重定向过程，-MOVED，数据迁移，对正在数据迁移的分片请求重定向过程，-ASK，gossip 协议，去中心化，弱一致性，像流言一样传播，心跳检测，心跳数据包内容，携带自身和随机其它节点信息，集群消息类型，ping，pong，meet，fail，update，新节点加入过程，节点握手阶段 meet -> pong -> ping，通过 gossip 字段让其它节点认识新节点，删除主节点，删除从节点，删除分片，故障转移，自动和手动故障转移，一个节点被超过半数节点标记 pfail 时标记 fail 状态，发现主节点 fail 时触发自动故障转移，集群当前纪元 (current epoch)，配置纪元 (config epoch)，故障转移流程，故障转移选举流程，副本迁移

25. Redis 7 实现分布式锁

（注意，接下来开始连续四篇都是讲 Redis 应用实现，但是我们不详细讲命令用法，如果遇到不认识的命令，请自己上 Redis 官网查询：<https://redis.io/commands/>，也可以上我的个人网站查看相关文章：<https://www.eririspace.cn/categories/Redis%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0/>，就是太早之前写的，格式都写得比较丑🤔）

用 Redis 实现分布式锁是关于 Redis 应用的面试题之一，而且 Redis 做分布式锁也是很常见的。知道如何用 Redis 实现一个分布式锁，不仅是为了面试，对自己写项目也是有用的。Redis 分布式锁可以分为两大类：**单节点加锁和多节点加锁**，接下来我们会分别对它们进行介绍，不过重点在于**单节点加锁**，我们会递进式的让大家从学会最简单的单节点分布式锁实现开始，一步一步的提高可靠性，最后学会实现一个可靠性较好的单节点分布式锁。

为什么需要分布式锁？

估计在被问到实现分布式锁之前，会被问到为什么需要分布式锁？那我们首先来回答一下这个问题。

答：分布式锁是用来解决分布式场景下的数据冲突的。在分布式场景下，多个实例可能会对同个数据进行访问和修改，这样就有可能出现冲突场景。比如有一个事务，是获取数据将它 + 1 之后将结果放回。现在有两个节点，节点 A 和节点 B 获取同一数据后，要对数据做 + 1 操作，令数据 + 1 后再将相同的数据放回。那么本来我们期望的是 A 和 B 各执行一次，最终数据能是原来的数据 + 2。但是 A 和 B 如果同时获取到数据，然后自己将它 + 1 后放回，那么最终数据只是原来的数据 + 1。所以我们需要在这个事务开始前加一个锁，执行完毕后解锁，获取到锁的节点才能执行事务，其余节点只能等待该节点解锁。这个锁用编程语言提供的锁肯定是不行的，因为它的作用范围只是实例本身，所以我们需要分布式锁。分布式锁一般通过所有节点都能访问的中间件实现，那么常用来做分布式锁的有 Redis，Zookeeper，Etcd 等中间件。

单节点加锁

单节点加锁指的是加锁过程只涉及单个 Redis 节点，性能高，但是可靠性较差。不过如果用 Redis 做分布式锁的话，一般业务都是要求性能优先，单节点加锁是最常用的一类 Redis 分布式锁。

单节点加锁方案

1. SETNX 命令

最简单的方法，我们可以通过一条 SETNX 命令来作为分布式锁的加锁操作，SETNX 命令在 key 不存在时才会进行设置。解锁时通过 DEL 命令将 key 删除。伪代码如下（C++ 风格，因为 C++ 是多数人都能看懂的语言）：

```
string key = data_id;

// 执行 SETNX key 1（值随便设置即可），如果不成功则进入内部
// 其实需要先对 Redis.Do 返回值做类型转换，
// 但我们是伪代码哈，怎么简单怎么来
// 我们假设这个函数能自动类型转换
// 以后的伪代码用上这个函数都视为自动类型转换~
// 你问我为什么不把命令名写成函数 Redis.Setnx() 这样？
// 那是因为我写完这几个应用篇后才想到... 不想改啦
if(Redis.Do("SETNX", key, 1) == 0) {

    // 在循环中不断尝试获取锁
    while(true) {
        if (Redis.Do("SETNX", key, 1) == 1)
            break; // 获取锁成功，退出循环

        Sleep(100); // 大约 100 ms 循环一次
    }
}

...
// 执行业务逻辑完毕后解锁
Redis.Do("DEL", key);
```


这样简单的加锁会有什么问题呢？我们可以想想，获取锁的节点如果崩溃了，那么就没人能解锁啦！所以我们还可以为分布式锁设置一个期限，也就是过期时间，这样分布式锁过期了就会被 Redis 删除，以上提到的问题也就解决啦。那么接下来我们就来为分布式锁加上这个过期时间。

2. SET 命令 + EX/PX/EXAT/PXAT 参数

SET 命令目前支持很多的扩展参数，EX / PX / EXAT / PXAT 是四个与过期时间有关的参数。过期时间参数后边需要跟着一个数字，不同参数跟着的数字意义如下：

EX: 秒级过期时间，如果输入 1 则是 1s 后过期。

PX: 毫秒级过期时间，如果输入 1000 则是 1000ms (1s) 后过期。

EXAT: 秒级时间戳，作为过期时刻，如果输入 1682577137 则是北京时间 2023-04-27 14:32:17 过期。

PXAT: 毫秒级时间戳，作为过期时刻，如果输入 1682577137000 则是北京时间 2023-04-27 14:32:17.000 过期。

接下来是一个超时时间为 30s 的分布式锁伪代码示例：

```
string key = data_id;
int expire = 30;

// 执行 SET key 1 NX EX 30, 如果不成功则进入内部
if(Redis.Do("SET", key, 1, "NX", "EX", expire) == 0) {

    // 在循环中不断尝试获取锁
    while(true) {
        if (Redis.Do("SET", key, 1, "NX", "EX", expire) == 1)
            break; // 获取锁成功，退出循环

        Sleep(100); // 大约 100 ms 循环一次
    }
}

...
// 执行业务逻辑完毕后解锁
Redis.Do("DEL", key);
```

这下我们的分布式锁加上过期时间了，因为业务节点崩溃导致无法释放锁的问题解决了。我们想想还有什么问题吗？比如，业务节点因为某些原因执行业务较慢，而这时分布式锁已经过期了，别的节点就可以获取到锁了。那这下就是两个节点都持有同个锁，它们都可以一起操作数据了，这下又可能出现冲突了，分布式锁白做了呀！怎么办呢？解决办法请看下一个加锁方案。

(网上流传的 SETEX + EXPIRE 或者为了隔离性执行把它俩写在 lua 脚本里的方案已经过时了，那是早期 Redis 版本的做法。Redis 早已支持在 SET 中使用与过期时间有关的扩展参数，我们也是以 Redis 7 实现分布式锁来说的，所以这些旧方法我们就不再讲述，具体可以上网搜索)

3. 看门狗 (Watch dog)

为了解决业务尚未执行完，锁却自动过期的问题，我们可以引入一个叫**看门狗 (Watch dog)** 的机制。这个机制很好理解，就是加锁成功之后，创建一个子线程定时给锁续期，这样只要业务仍在执行，锁就不会过期。下面我们给上边的 SET + 扩展参数的示例加上看门狗：

```

// 看门狗具体逻辑函数
void watchDog(string key, string uuid, int time) {

    while(true) {

        // key 续期
        // 首先判断 value 是不是等于生成的 uuid
        // 如果是则说明是自己持有的锁
        if (Redis.Do("GET", key) != uuid)
            break; // 如果 val 不是生成的 uuid，则是已经解锁了，退出循环

        Redis.Do("EXPIRE", key, time) // 续期
        sleep(time/3 * 1000); // 每 1/3 过期时间循环一次
    }
}

string key = data_id;
string uuid = UUID.Generate() // 生成 uuid 作为 value
int expire = 30;

// 执行 SET key uuid NX EX 30，如果不成功则进入内部
if(Redis.Do("SET", key, uuid, "NX", "EX", expire) == 0) {

    // 在循环中不断尝试获取锁
    while(true) {
        if (Redis.Do("SET", key, uuid, "NX", "EX", expire) == 1)
            break; // 获取锁成功，退出循环

        sleep(100); // 大约 100 ms 循环一次
    }
}

// 创建并运行看门狗线程
thread t(watchDog, key, uuid, expire);
t.Run() // 开始运行看门狗线程，C++ thread 不是用 Run 方法别被误导哦

...
// 执行业务逻辑完毕后解锁
Redis.Do("DEL", key);

```

示例中的看门狗函数里的 GET 和 EXPIRE 逻辑其实最好写在 lua 脚本让 Redis 隔离性执行。否则就算 GET 发现 value 仍然是 生成的 uuid，但到了 EXPIRE 时，也可能已经解锁了，就会执行无效续期，或者为当前持有锁的其它节点续期。用 lua 脚本封装多个 Redis 命令也能节省命令的 RTT（Round-Trip Time，往返时间），所以学会写 lua 脚本对使用 Redis 来说还是很重要的，lua 是一门简单的语言，看看菜鸟教程学基本语法什么的就够用了，推荐大家到自己要在业务中使用 Redis 时，也能去学 lua 并用 lua 脚本封装能够在一次传输内完成的 Redis 命令逻辑，能够提高性能并且防止冲突。而节省传输时间还有 pipeline 方法，也就是将命令打包传输，需要看使用的 Redis 客户端框架是否支持该功能。

此时我们增加了看门狗，现在已经有了自动锁过期和续期机制了，又解决掉了一个问题。现在已经算是个比较可靠的单节点分布式锁了，Java 的著名 Redis 框架 Redisson 实现的分布式锁也是这样，区别在于它使用 hash 结构和 hincrby 命令来设置 key，以实现可重入锁，以及使用了 pub/sub 来向等待解锁的节点通知解锁消息。那么现在还有什么问题吗？可能你会想到业务节点要是全局卡住了，看门狗也没法续期，虽然能卡住这么久的情况估计概率小到可以忽略不计，但是想要进行更可靠的处理也是可以的。一个办法就是减少看门狗的续期间隔，让看门狗续期更频繁。还有一个办法是在代码里设置业务超时时间，让它小于等于分布式锁期限，如果执行超时了自动中止本次业务。让我们先放下这个极小概率事件，想一个单节点加锁无法解决的故障。当我们加锁的节点具有从节点时，如果它宕机了接着触发故障转移，将从节点提升为了主节点，但是宕机前并没有完成主从同步，从节点没有复制这个分布式锁，那么现在锁就丢失了。为了解决这个问题，就得使用多节点加锁方案了。

多节点加锁

多节点加锁顾名思义，需要在多个节点上完成加锁操作，才会认为获取到了锁。说到 Redis 的多节点加锁方案，那么一般问的就是 Redis 作者 antirez 提出的 **Redlock** 了。

多节点加锁方案

Redlock

Redlock 的思想其实很简单，就是让超过半数的节点都设置好了锁，就认为加锁成功。下面帮大家精简的翻译一下官方文档中关于 Redlock 的加锁流程（官方文档：<https://redis.io/docs/manual/patterns/distributed-locks/#the-redlock-algorithm>）：

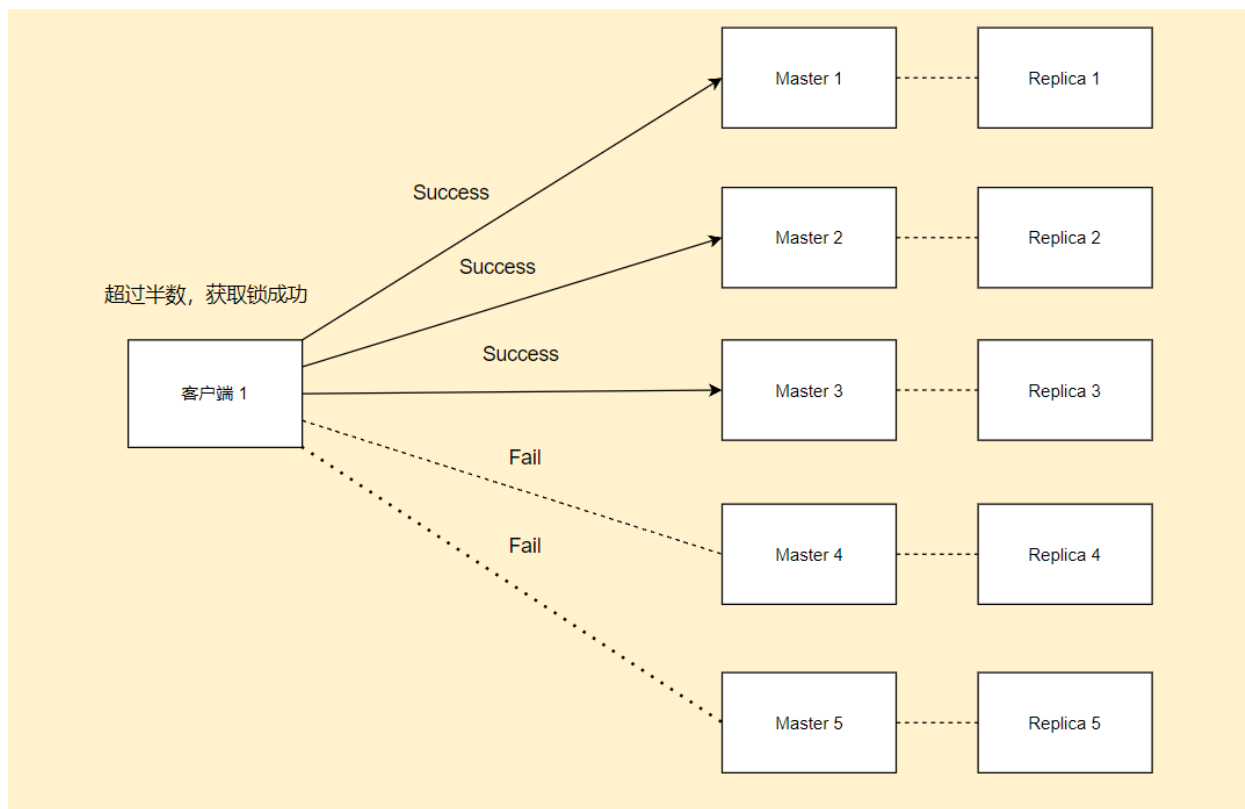
1. 首先获取当前毫秒时间戳，作为获取锁的开始时间。
2. 在所有 N 个节点中按顺序进行加锁。在每个 Redis 节点加锁时，客户端要设置一个的等待时间，并且这个等待时间要小于锁过期时间。如果节点超时未响应，则忽略该节点，向下一个节点加锁。例如，如果锁过期时间为 10 秒，则等待时间可能设置在 5 ~ 50 毫秒范围内。这可以防止客户端在尝试与已关闭的 Redis 节点通信而长时间处于阻塞状态：如果一个节点不可用，我们应该尽快尝试与下一个节点通信。
3. 客户端通过当前时间减去步骤 1 中获得的时间戳，来计算加锁所用的时间。当且仅当客户端能够在超过半数的节点中完成加锁时，并且加锁的总时间小于锁有效期，则认为获得了锁。
4. 如果获取锁成功，则锁的有效时间 = 锁过期时间 - 加锁经过的时间。
5. 如果客户端由于某种原因未能获得锁（要么无法在 $N/2 + 1$ 个节点中完成加锁，要么有效时间为负），它将对所有节点发出解锁命令（所有节点，包括未加锁成功的节点）。

Redlock 的解锁很简单，对所有节点发出解锁命令（就是删除锁）即可。

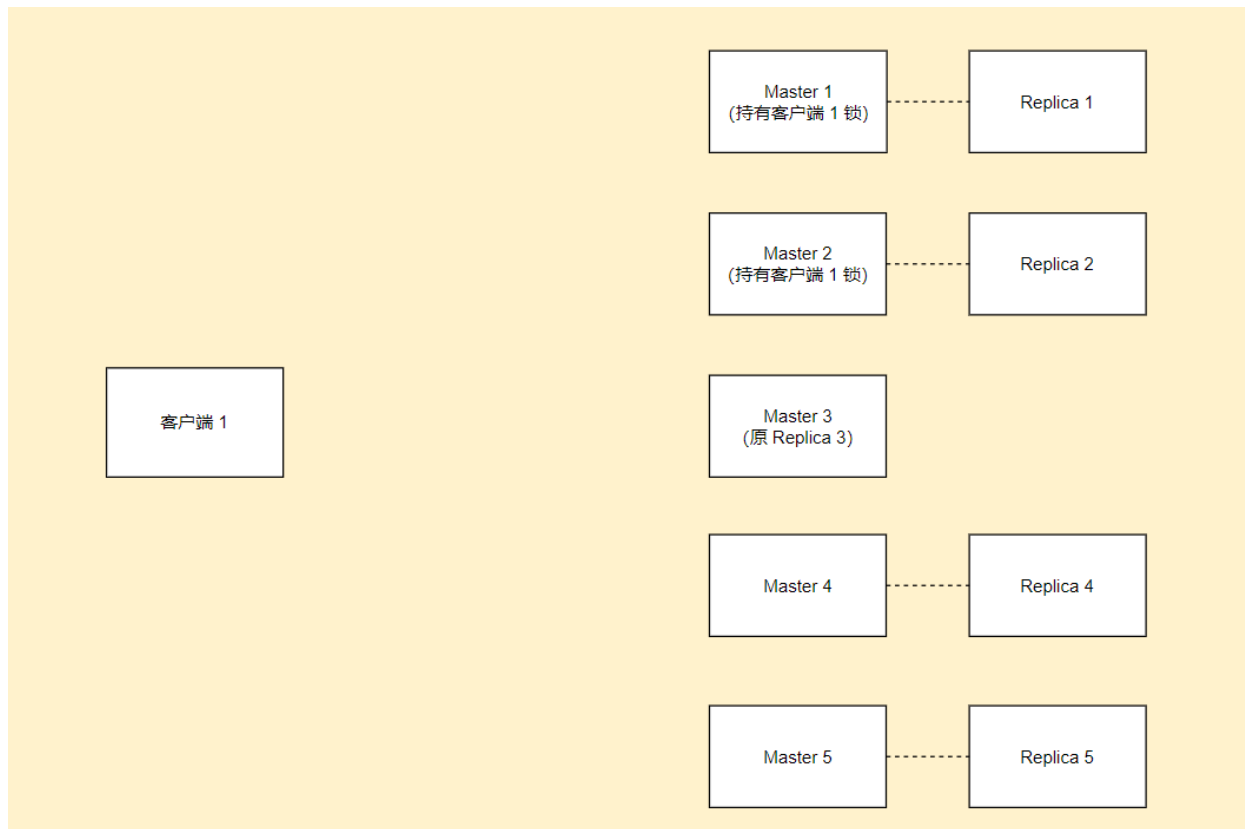
Redlock 解决问题了吗？

现在我们就来唠唠这个 Redlock，它真的解决主从未同步导致锁丢失的问题了吗？

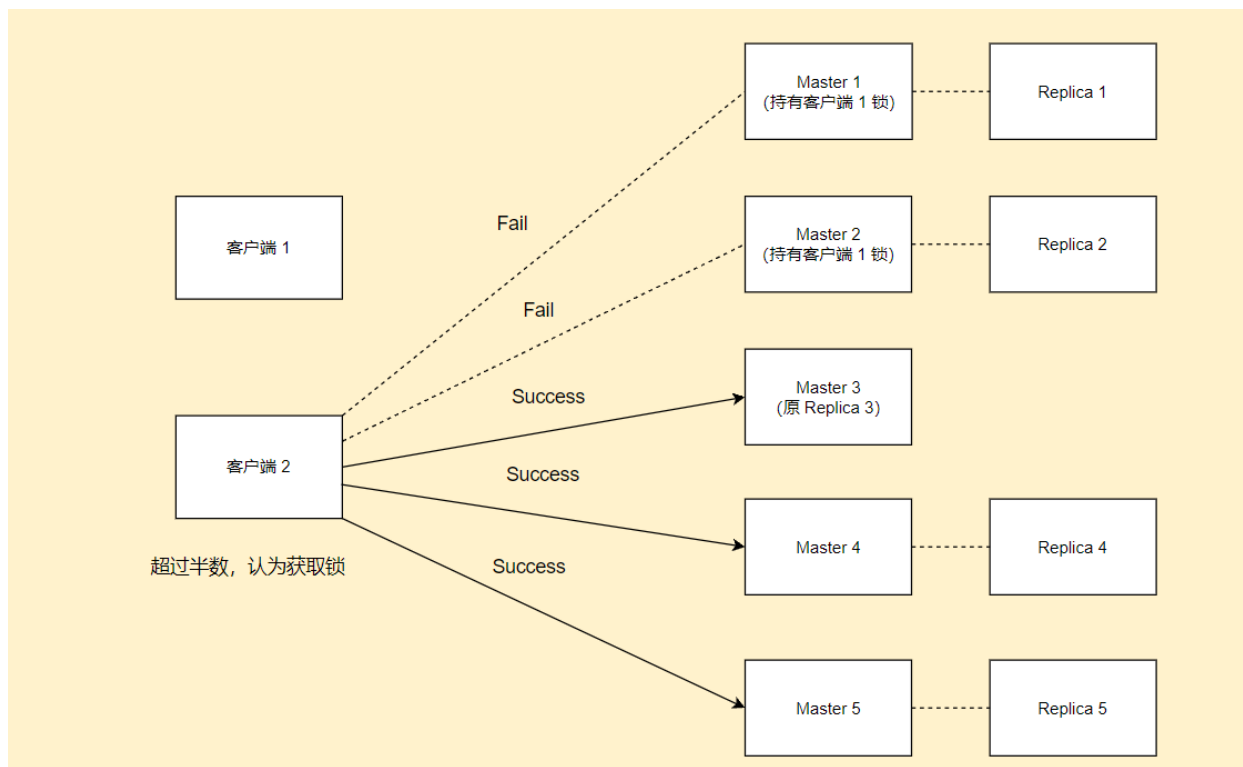
以 5 个主节点为例，它们各自有 1 个从节点。首先客户端 1 通过 Redlock 算法加锁，在主节点 1, 2, 3 加锁成功，但在主节点 4 和主节点 5 由于网络问题加锁失败了。此时加锁节点数量为 3，已经超过了半数，所以客户端 1 认为自己获取到了锁。



此时主节点 3 宕机了，锁还尚未同步到从节点 3，从节点此时完成故障转移，切换为主节点。



现在客户端 2 要进行加锁，它到主节点 1, 2 加锁失败了，但是在主节点 3, 4, 5 均加锁成功。此时加锁节点数量为 3，已经超过了半数，所以客户端 2 认为自己获取到了锁。



哦豁，此时此刻，客户端 1 和客户端 2 都认为自己持有了锁，这下又玩完了。所以吧，Redlock 还是有漏洞的。都不用像 DDIA（《数据密集型应用系统设计》，著名的野猪书）作者 Martin Kleppmann 去分析 Redlock 关于时钟漂移之类的更深层的漏洞，就算给超过半数节点加锁了，主从未同步的宕机导致锁丢失的问题还是存在。那么 Redlock 就完全不能用了吗？也不是，如果你让刚刚完成主从切换的新主节点等待一个锁过期时间之后，再开始对外提供服务，那么也是可以规避掉这个问题的。

吾有一计——ReplicaLock:

（现在是夹带私货时间~🔍🔍）

早在去年，我就思考过关于主从未同步而宕机导致锁丢失的问题，然后我动手用 Go 语言实现并开源了一个多节点加锁方案，称为 ReplicaLock。（github 仓库地址：<https://github.com/ncghost1/Redis-ReplicaLock>）

ReplicaLock 中文翻译可以叫 副本锁 / 从节点锁，这是因为 ReplicaLock 加锁成功的因素就和从节点有关。ReplicaLock 的原理真的简单简单很简单，比 Redlock 还要简单。就是等待所有从节点同步完毕后，客户端才认为自己获取到了锁。整个流程只用对一个主节点进行加锁，然后设置一个等待时间，通过 wait 命令来等待从节点完成同步，在等待时间内所有从节点完成同步了，客户端就认为自己获取到了锁。这样一来，主节点宕机后，切换成主节点的从节点也已经保证了拥有锁信息，就不会出现锁丢失情况了。

所以，我不认为解决宕机锁丢失问题有必要让客户端跑去尝试给全部主节点加锁。一般一个主节点配两台从节点就已经足够可靠了，所以一般情况下 ReplicaLock 只会在三个节点加锁成功便能认为获取锁成功，而整个 Redis 集群主节点可能远比三个节点多得多，如果要超过半数的节点加锁成功，可能耗时很长，这样性能会很差。ReplicaLock 可能会因为要加锁的主节点写流量很大，导致需要复制的数据变多，这样就会导致加锁慢，甚至超时。对于这个问题，我建议如果有空闲的机器，可以让它们专门作为分布式锁使用的节点，分布式锁加锁请求流量几乎不可能超过 Redis 性能（什么业务这么顶啊？原神还是星穹铁道？🤔）。除此之外，还有可能因为网络不良导致加锁慢或者超时，不过这是所有加锁方式都无法避免的问题，不能当作 ReplicaLock 的缺点。

记忆要点：单节点加锁方案，SETNX，SET + 超时参数，看门狗机制，多节点加锁方案，Redlock

26. Redis 7 实现延时队列

在之前关于 Zset 类型的文章里，我提到了可以用 Zset 实现延时队列。使用 Redis 实现延时队列实际上也是面试官爱问的一个应用题，而且实现起来其实还是有很多细节的，如果你不单单能讲出实现延时队列的基本思路，还能讲出实现的一些细节，甚至给延时队列添加更多可靠性保障功能就更好了。所以我们这回就来好好研究一下，如何用 Redis 实现延时队列。

从简单到完善的延时队列实现

如果你比较了解 Redis 的用法，那么你可能会想到启用过期监听功能，然后通过 pub/sub 订阅 Redis 推送的过期 key 消息。但是这个方案有两个问题，一个是 pub/sub 不能持久化消息，而且 Redis 服务器只会推送一次过期通知。另一个问题，也就是 key 过期了也是惰性删除的，Redis 并不会马上发现过期 key，如果忘了就回咱们的第 17 篇看看。

所以还是先不要搞其他花里胡哨的，我们用 Zset 实现就好。基本思路是：利用 score 存定时事件下次执行的时间戳，而 member（元素）就是事件的相关信息，然后通过定期的用 `ZRANGEBYSCORE` 命令获取当前已经可以执行的事件，最后用 `ZREMRANGEBYSCORE` 命令将它们从 Zset 删除。以下是 C++ 风格伪代码示例：

```
// Redis 延时队列类
class DelayQueue {
    string events_key; // 存储所有事件的有序集合 key

    DelayQueue(string events_key) {
        this->events_key = events_key;
    }

    // 添加事件
    void AddEvent(string event, long long time) {
        Redis.Do("ZADD", events_key, time, event);
    }

    // 获取到期事件
    string[] GetEvent() {
        string [] eventList; // 到期事件数组
        long long now = GetNowUnixMillTime(); // 获取当前毫秒时间戳

        // 从事件有序集合中获取已经到期事件
        // 假设返回值自动类型转换为 string 数组
        eventList = Redis.Do("ZRANGEBYSCORE", events_key, 0, now);

        // 将已经到期的事件删除
        Redis.Do("ZREMRANGEBYSCORE", events_key, 0, now);
        return eventList; // 返回到期事件
    }
}
```




以上就是一个基本的 Redis 延时队列了。但是这样简单的实现是有一些问题的。比如消费者调用 GetEvent() 从延时队列中获取定时事件，然后自己按顺序处理，如果消费者突然宕机了，那这些事件就丢失了。所以我们需要让到期的事件能够持久化，这个持久化还是直接交给 Redis 的持久化机制，我们可以用一个 List 类型作为一个 FIFO（先进先出）队列，它负责存储到期事件。当然你也可以用 Stream 类型，而且它可能更好，但是 Stream 类型一般比较冷门，毕竟有更专业的消息队列，面试官也可能没用过，所以我们还是老老实实说 List 的实现吧。现在我们改造一下逻辑，添加一个到期事件 List，MoveReadyEvent 函数，它的功能是将到期事件从 Zset 移动至 List，然后我们将 GetEvent() 变为从 List 获取到期事件。改造后的伪代码如下：

```
// Redis 延时队列类 第2版
class DelayQueue {
    string events_key; // 存储所有事件的有序集合 key
    string ready_events_key; // 到期/就绪事件列表 key

    DelayQueue(string events_key, string ready_events_key) {
        this->events_key = events_key;
        this->ready_events_key = events_key;
    }

    // 添加事件
    void AddEvent(string event, long long time) {
        Redis.Do("ZADD", events_key, time, event);
    }

    // 将到期事件从 事件有序集合 移动至 到期事件列表
    void MoveReadyEvent() {
        string [] eventList; // 到期事件数组
        long long now = GetNowUnixMillTime(); // 获取当前毫秒时间戳

        // 从事件有序集合中获取已经到期事件
        // 假设返回值自动类型转换为 string 数组
        eventList = Redis.Do("ZRANGEBYSCORE", events_key, 0, now);

        // 将到期事件加入列表，用 RPUSH 相当于从队列尾部加入
        for(int i = 0; i < eventList.size(); i++) {
```

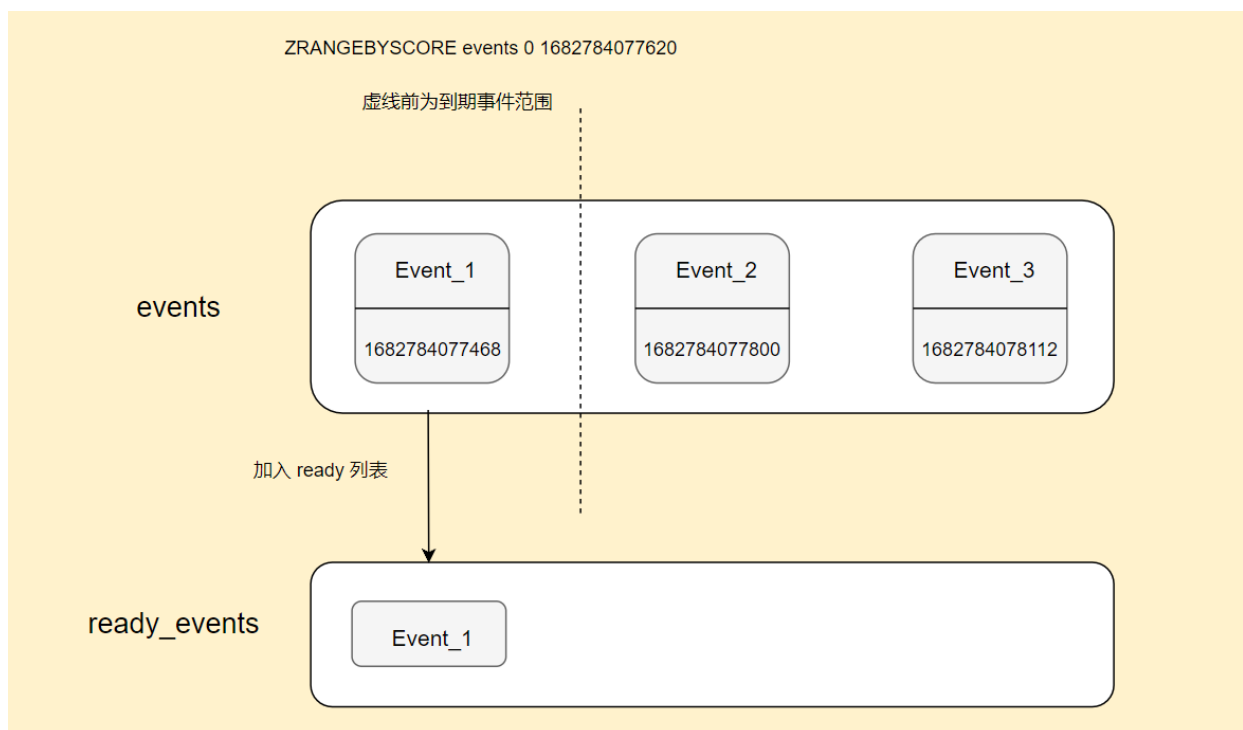
```

        Redis.Do("RPUSH", ready_events_key, eventList[i]);
    }
}

// 获取到期事件
// 我们加个整型参数 n
// 可以指定一次拉取的到期事件数量
string[] GetEvent(int n) {
    string [] eventList; // 到期事件数组

    // 从队列头部弹出 n 个到期事件
    eventList = Redis.Do("LPOP", ready_events_key, n);
    return eventList; // 返回到期事件
}
}

```



现在我们已经将到期事件存储到了 List 里，具备持久化功能了。但是吧，现在还是存在消费者丢失消息的可能性。消费者拉取 n 个事件，那我们就该从 List 弹出 n 个事件，如果消费者消费中途宕机了，或者传输给消费者超时了，还是丢掉了未消费的事件。现在只是和以前相比丢得少了，那么我们这时候该怎么保证消费失败，事件也不丢失呢？

我们可以将从 List 弹出的事件再存到一个 Zset 里，这个 Zset 作为重试队列，score 存储的是下次重试时刻的毫秒时间戳。我们再引入一个消费确认机制，消费者消费完成后发送一个 Ack 消息，确认已经消费完毕，我们就可以将事件从重试队列中删掉了。之后可以用一个线程定期扫描这个重试队列，将已经到达重试时间的事件放回 List，等待下次重新消费。而这个重试时间我们可以让第一次重试是 1 分钟后，之后每次重试翻倍，限制一个最大值是 1 小时后，所以我们还需要一个 hash 类型来记录事件对应的重试次数。同时，我们不能让一直消费失败的事件永远重试下去，所以我们还要限制最大重试次数，事件到达最大重试次数后，我们将它放到一个 Set 里，等待人工后续处理，类似于死信队列。以下是改造后的伪代码：

```

// Redis 延时队列类 最终版
class DelayQueue {

```

```

string events_key; // 存储所有事件的有序集合 key
string ready_events_key; // 到期/就绪事件列表 key
string retry_events_key; // 重试事件的有序集合 key
string retry_times_key; // 记录事件重试次数的哈希表 key
string dead_events_key; // 达到重试上限的事件集合 key
int max_retry_times; // 最大重试次数
long long retry_after; // 重试时间间隔, 单位毫秒
long long max_retry_after; // 最大重试时间间隔
int max_retry_after_times; // 到达最大重试时间间隔的重试次数

DelayQueue(string events_key, string ready_events_key, string retry_events_key,
string retry_times_key, int max_retry_times, long long retry_after, long long
max_retry_after) {
    this->events_key = events_key;
    this->ready_events_key = events_key;
    this->retry_events_key = retry_events_key;
    this->retry_times_key = retry_times_key;
    this->dead_events_key = dead_events_key;
    this->max_retry_times = max_retry_times;
    this->retry_after = retry_after;
    this->max_retry_after = max_retry_after;

    // 计算到达最大重试时间间隔的重试次数, 之后用次数判断可以防溢出
    int count = 0;
    while(max_retry_after > 0) {
        max_retry_after -= (1<<count) * retry_after;
        ++count;
    }
    this->max_retry_after_times = count;
}

// 添加事件
void AddEvent(string event, long long time) {
    Redis.Do("ZADD", events_key, time, event);
}

// 添加重试达到上限的事件到对应集合
void AddDeadEvent(string key) {
    Redis.Do("SADD", dead_events_key, event);
}

// 将到期事件从 事件有序集合 移动至 到期事件列表
void MoveReadyEvent() {
    string [] eventList; // 到期事件数组
    long long now = GetNowUnixMillTime(); // 获取当前毫秒时间戳

    // 从延时队列中获取已经到期事件
    // 假设返回值自动类型转换为 string 数组
    eventList = Redis.Do("ZRANGEBYSCORE", events_key, 0, now);

    // 将到期事件加入列表, 用 RPUSH 相当于从队列尾部加入
    for(int i = 0; i < eventList.size(); i++) {

```

```

        Redis.Do("RPUSH", ready_events_key, eventList[i]);
    }
}

// 将重试事件从 重试事件有序集合 移动至 到期事件列表
void MoveRetryEvent() {
    string [] eventList; // 可重试事件数组
    long long now = GetNowUnixMillTime(); // 获取当前毫秒时间戳

    // 从重试队列中获取可重试事件
    // 假设返回值自动类型转换为 string 数组
    eventList = Redis.Do("ZRANGEBYSCORE", retry_events_key, 0, now);
    for(int i = 0; i < eventList.size(); i++) {
        string event = eventList[i];

        // 检查重试次数是否超出限制
        if(Redis.Do("HGET", retry_times_key, event) > max_retry_times) {
            Redis.Do("HDEL", retry_times_key, event); // 删除计数
            AddDeadEvent(event); // 加入重试次数达到上限的集合
            continue;
        }

        Redis.Do("RPUSH", ready_events_key, event); // 将事件重新加入列表
        Redis.Do("ZREM", retry_events_key, event); // 从重试队列中删除
    }
}

// 收到消费确认消息后，删除重试事件
void DelRetryEvent(string key) {
    Redis.Do("ZREM", retry_events_key, event);
}

// 获取到期事件
string[] GetEvent(int n) {
    string [] eventList; // 到期事件数组

    // 从队列头部弹出 n 个到期事件
    eventList = Redis.Do("LMPOP", ready_events_key, n);

    for(int i = 0; i < eventList.size(); i++) {
        string event = eventList[i];

        // 自增重试次数，并获取 HINCRBY 命令返回值，返回自增后的重试次数
        int count = Redis.Do("HINCRBY", retry_times_key, event, 1);
        long long now = GetNowUnixMillTime(); // 获取当前毫秒时间戳
        long long retry_time; // 重试时间

        // 计算重试时间，这里要判断有没有到达最大时间限制
        if(count >= max_retry_after_times) {
            retry_time = max_retry_times;
        } else {
            retry_time = now + (1 << (count-1)) * retry_after;
        }
    }
}

```

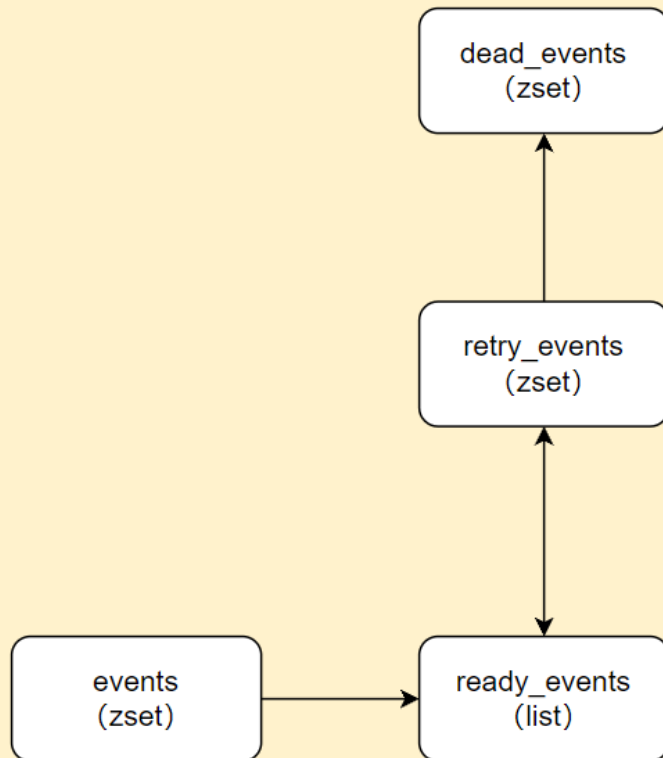
```

    }

    // 将事件加入重试事件有序集合（重试队列）
    Redis.Do("ZADD", retry_events_key, retry_time, event);
}

return eventList; // 返回到期事件
}
}

```



以上就是我们 Redis 延时队列的最终版本了。但是以上实现其实还有些问题，比如没有使用 lua 脚本封装多个 Redis 命令，那么就可能因为执行不具有隔离性而翻车（我暂时还没想到以上逻辑不用 lua 会在哪里冲突，就先这么说吧），而且传输次数太多导致性能并不算好；我们接收到消费确认消息后，从重试队列删除重试事件时，可能已经被负责定时扫描的线程加回列表了，所以也可能导致重复消费；还有一个问题是，如果我们的事件信息有可能完全相同，那就还需要为每个事件生成一个 UUID（唯一 id），并再使用一个 hash 类型存储事件和 UUID 的映射关系，其它数据类型中只存储它的 UUID，最后要返回事件给消费者时再从 hash 类型中取出事件信息；另外，你也可以为重试事件单独放入一个专门的列表，而不是放回到期事件列表，将它们分开处理。

可能你看到一大串的代码，根本不想去看。没事，我们总结一下用 Redis 实现一个较为可靠的延时队列至少需要哪些数据结构，你只要记住并理解它们就 OK 了：

1. 存储所有事件的有序集合（zset）
2. 存储到期/就绪事件的列表（list）
3. 记录事件重试次数的哈希表（hash）

4. 达到重试上限的事件集合 (set)

参考文章: <https://www.cnblogs.com/Finley/p/16400287.html> , 已获得作者授权 (倒不如说作者指示我写这个的🐶)

记忆要点: Redis 实现一个较为可靠的延时队列需要哪些数据结构, 存储所有事件的有序集合 (zset) , 存储到期/就绪事件的列表 (list) , 记录事件重试次数的哈希表 (hash) , 达到重试上限的事件集合 (set)

27. Redis 7 实现布隆过滤器

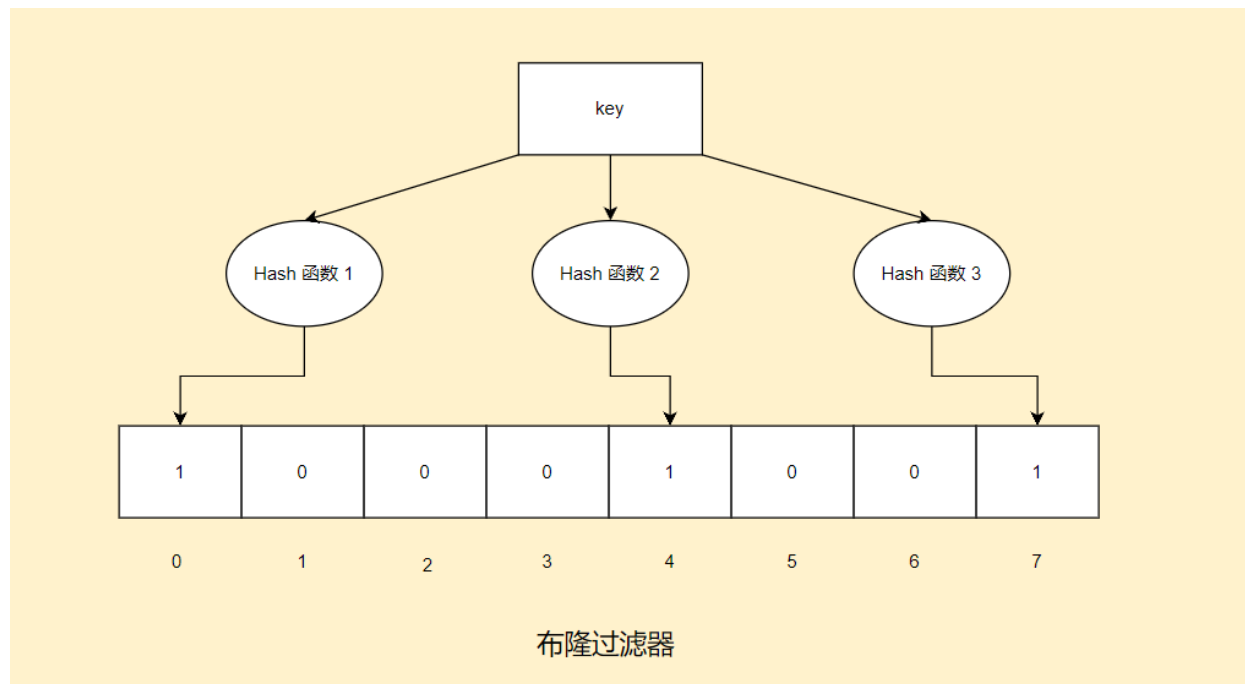
布隆过滤器是海量数据处理面试题常问的解决方案, 一般情况下是了解原理和应用场景, 并把它讲出来即可。但是, 万一有个面试官出点新颖的问题, 问你怎么用 Redis 实现布隆过滤器呢? 那么本篇我们就来未雨绸缪一下, 首先带大家了解布隆过滤器, 之后我们用 Redis 实现一个布隆过滤器。

<https://hur.st/bloomfilter/>

布隆过滤器

布隆过滤器是一种快速检索数据是否存在于集合中的概率型数据结构。它在 1970 年由布隆提出, 基于二进制和一系列随机哈希函数构建而成。其主要优点是占用空间小、查询速度快, 适合用于大规模数据处理, 如网络爬虫中的 URL 判重、拼写检查、黑名单过滤等场景。

布隆过滤器的基本思想是利用位数组来表示集合, 并使用 k 个独立的哈希函数将元素映射到位数组中的 k 个位置上, 将这些位置标记为 1。当检查一个元素是否在集合中时, 将该元素进行 k 次哈希映射, 并检查对应的 k 个位置是否都为 1, 若存在任一位置为 0, 则可确定该元素不在集合中。若所有位置都为 1, 则该元素可能存在于集合中, 也可能不在 (因为别的 key 也可能哈希冲突), 存在一定的误判率。



需要注意的是，布隆过滤器无法删除已添加的元素，因为删除操作可能会影响其他元素的检索结果。同时，误判率随着元素数量的增加而逐渐增加，但可以通过增大位数组长度或使用更多的哈希函数来降低误判率。

我们来做 <https://hur.st/bloomfilter/> 这个网站，计算在 1G 数据的场景下，使用 5 个函数，让误判率达到 0.1% 需要多大的位数组。

n

Number of items in the filter (optionally with SI units: k, M, G, T, P, E, Z, Y)

1G

p

Probability of false positives, fraction between 0 and 1 or a number indicating 1-in-p

1.0E-3

m

Number of bits in the filter (or a size with KB, KiB, MB, Mb, GiB, etc)

k

Number of hash functions

5

提交

n = 1,000,000,000

p = 0.001 (1 in 1,000)

m = **17,284,997,872 (2.01GiB)**

k = 5

可以看到需要大约 2.01 GB 的位数组即可让误判率达到 0.1%，也就是数据量的 2 倍。也许你会觉得空间还是太多了，而且你也不需要那么低的误判率，你认为 1% 就够用了，那么我们看看 1% 误判率需要多大的数组：

n

Number of items in the filter (optionally with SI units: k, M, G, T, P, E, Z, Y)

1G

p

Probability of false positives, fraction between 0 and 1 or a number indicating 1-in-p

1.0E-2

m

Number of bits in the filter (or a size with KB, KiB, MB, Mb, GiB, etc)

k

Number of hash functions

5

提交

n = 1,000,000,000

p = 0.01 (1 in 100)

m = **9,848,803,655 (1.15GiB)**

k = 5

只需要 1% 误判率的话，需要开的空间就降低到 1.15 GB 了，少了一个小数点降低还是蛮多的。如果你觉得需要多开一倍多一些的数据空间还是太多了，咳咳...这个布隆过滤器啊，就是在有足够空闲空间的场景下我们才去使用的，如果你连两倍的数据空间都剩不下来，那咱还是别折腾了。

Redis 实现布隆过滤器

Redis 里提供了 bitmap（位图），可以按位进行操作，所以我们的布隆过滤器就是在 bitmap 基础上实现的，不过要注意 bitmap 最大只能分配 512 MB 空间，所以我们实现的布隆过滤器要限制在 512 MB 以内。以下我们用 C++ 风格的伪代码分步骤来实现布隆过滤器。

首先我们定义 3 个简单的 hash 函数：

```
long long hashfunc_1(string key, long long bloom_size) {
    long long hash = 0;
    long long sum = 0;
    for(int i = 0; i < key.length(); i++) {
        sum += (int)key[i];
    }

    // 简单点，用每个字符的总和 % 布隆过滤器位数即可
    hash = sum % bloom_size;
    return hash;
}

long long hashfunc_2(string key, long long bloom_size) {
    long long hash = 0;
    long long sum = 0;
    for(int i = 0; i < key.length(); i++) {
        sum += (int)key[i];
    }

    // （每个字符的总和 + 20） % 布隆过滤器位数
    hash = (sum + 20) % bloom_size;
    return hash;
}

long long hashfunc_3(string key, long long bloom_size) {
    long long hash = 0;
    long long sum = 0;
    for(int i = 0; i < key.length(); i++) {
        sum += (int)key[i];
    }

    // （每个字符的总和 + 80） % 布隆过滤器位数
    hash = (sum + 80) % bloom_size;
    return hash;
}

// 全局函数数组保存 hash 函数
vector<function<long long(string, long long)>> hashFuncArr;

void initHashFuncArray() {
    hashFuncArr.push_back(hashfunc_1);
    hashFuncArr.push_back(hashfunc_2);
    hashFuncArr.push_back(hashfunc_3);
}
```

接下来我们再实现布隆过滤器的添加 key 功能：

```
// 布隆过滤器位图的 key
string BloomKey = "BloomFilter";

void bloomAdd(string key, long long bloom_size) {
    for(int i = 0; i < hashFuncArr.size(); i++) {
        long long hash = hashFuncArr[i](key, bloom_size);

        // 将位图第 hash 位设为 1
        Redis.Do("SETBIT", BloomKey, hash, 1);
    }
}
```

最后实现布隆过滤器的查询功能：

```
bool bloomExists(string key, long long bloom_size) {
    for(int i = 0; i < hashFuncArr.size(); i++) {
        long long hash = hashFuncArr[i](key, bloom_size);

        // 如果位图第 hash 位为 0，则 key 不存在
        if (Redis.Do("GETBIT", BloomKey, hash) == 0)
            return false;
    }
    return true; // 返回 true，但也不代表 key 一定存在
}
```

这样，我们一个简单的布隆过滤器就实现完了。是不是感觉很简单呢？

最后给大家附赠 Redis 官方推荐的布隆过滤器 Module 仓库（除了布隆过滤器，它还有其它功能）：<https://github.com/RedisBloom/RedisBloom>

记忆要点：布隆过滤器，哈希函数，位数组，查询存在不一定存在，但是不存在一定不存在，Redis 实现布隆过滤器，bitmap

28. Redis 7 实现限流

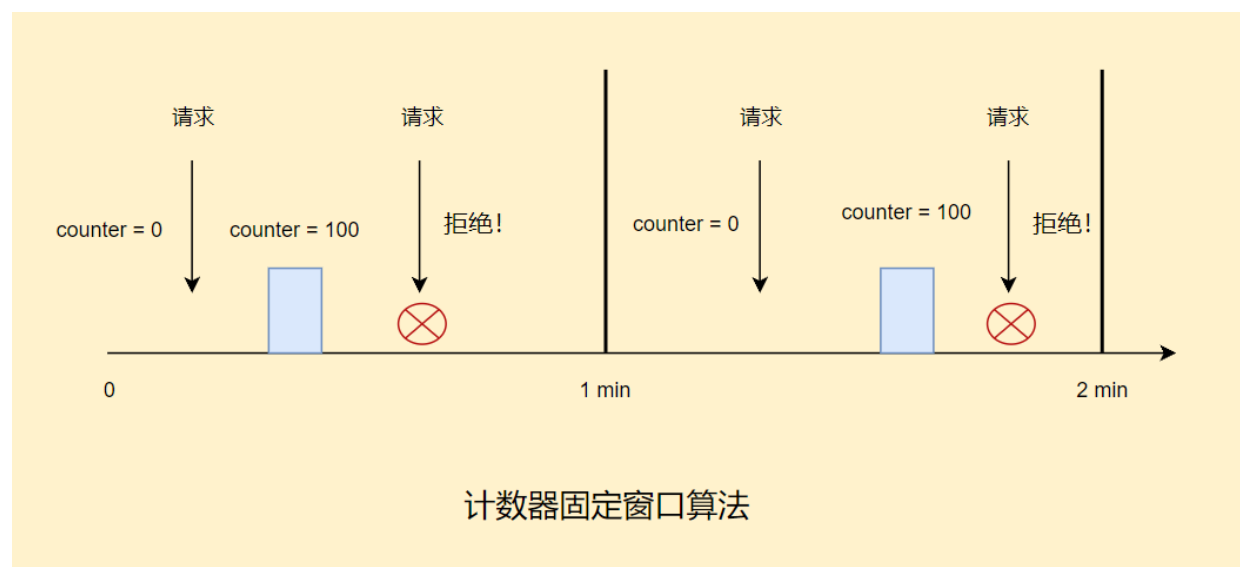
Redis 能用来做限流功能，虽然使用率可能没分布式锁，延时队列和布隆过滤器那么高，但学会怎么用 Redis 实现限流还是值得的。因为根据你在前边所学到的东西，你的面试 Redis 能答得很不错，并且能讲到较深的底层原理了，这时就有可能面试官突发奇想，问你怎么用 Redis 实现限流功能。那么我们不仅要接得住招，还得给它出几种招式，让面试官心服口服。那么接下来，我们就用 Redis 来实现三种经典的限流算法：**计数器算法，漏斗限流算法，令牌桶算法**。还没有学过限流算法的同学，看我这一篇就够啦！

限流方案

你现在要完成一个关于限流的需求，某接口一分钟之内最多只能请求 100 次，并且用 Redis 实现。那么你会想到用什么方法实现呢？

计数器固定窗口算法

计数器算法是一种简单的限流算法，它又分为固定窗口和滑动窗口两种算法，我们先讲更为简单的固定窗口算法。固定窗口算法是指通过维护一个单位时间内的计数值 counter，每个请求进来了就让 counter + 1。当前时间窗口内 counter 如果超过限制了就拒绝请求，到了下一个时间窗口就将 counter 重置为 0。用我们的需求例子来说，就是维护 1 分钟的时间窗口，counter > 100 就拒绝请求，每过去 1 分钟就将 counter 清零。这个方法其实是有问题的，因为它的时间不是动态的 1 分钟窗口，所以如果在一个时间窗口的最后时刻突发大量请求，迅速打满了限制，然后马上到了下一分钟窗口，将 counter 重置之后又马上打满了限制，那可就是 1 分钟内请求数量达到限制的两倍了。这个问题也有人将它称为临界值问题。



接下来我们讨论实现细节。我们主动去检查什么时候到达新的时间窗口，或者用定时器重置 counter，都是不需要的。我们在设置 counter 时加上一个过期时间，过期时间设置为时间窗口大小就好。

在 Redis 中，我们可以用一个 String 类型作为 counter，以下是 C++ 风格的伪代码示例：

```
// 定义变量
string counter_key = "limited_counter"; // 计数器 field
int period = 60; // 时间窗口大小, 60 秒
int limited = 100; // 请求限制, 100 次
...

bool isActionAllowed(string counter_key, int period, int limited) {

    // 对 counter 用 INCR 命令 + 1, 如果返回值 > 100 那么就拒绝请求
    if(Redis.Do("INCR", counter_key) > limited) {
        return false;
    }

    // 可以请求, 设置过期时间后返回 true
    Redis.Do("EXPIRE", counter_key, period);
    return true;
}
```

```
}
```

计数器滑动窗口算法

计数器滑动窗口算法与固定窗口算法相比实现较难，但是对于限流需求来说更加贴近。滑动窗口算法的实现方法也有很多种，比如有将时间窗口划分成多格的，每格是一个很小的时间窗口。假设把 60 秒划分为 5 格，每格占 12 秒，把请求限制平均分到每一格，每一格的请求限制就是 $100/5 = 20$ ，每过 10 秒将时间窗口向前移动一格。这样的方法我个人觉得不行，这样做本质上还是固定窗口，只是将大化小了，限流还是做得不够平滑。

我们用 Redis 的 Zset 可以实现一个比上面提到的方案更好的滑动窗口。我们用请求的时间戳作为 score，将当前时间戳 - 60 秒便可以得到我们的窗口，窗口之前的元素全部删除掉，获取 Zset 元素数量即是我们的 counter。这个窗口范围会跟着时间而滑动，就是我们的滑动窗口啦。以下是用 Zset 实现的滑动窗口伪代码：

```
// 定义变量
string key = "slidingwindow"; // 固定窗口 key
int period = 60; // 时间窗口大小, 60 秒
int limited = 100; // 请求限制, 100 次
...

bool isActionAllowed(string key, int period, int limited) {
    // 首先判断开始时间
    int now = GetNowUnixTime(); // 获取当前秒级时间戳

    // 删除早于时间窗口的元素
    Redis.Do("ZREMBYSCORE", key, 0, now - period);

    int counter = Redis.Do("ZCARD", key);

    // 窗口内的元素数量达到限制了, 返回 false
    if (counter >= limited) return false;

    // 加入请求, score 设为当前时间戳
    // value 随便设置即可, 这里也用当前时间戳
    Redis.Do("ZADD", key, now, now);

    // 请求成功后我们设置一个过期时间, 过期时间为一个时间窗口大小
    // 这是为了自动释放冷门的限流 key
    Redis.Do("EXPIRE", key, period);
    return true;
}
```

漏斗限流算法

漏斗限流算法也是一个常用的限流算法，一看就知道“漏斗”是用来比喻算法过程的。让我们想想，漏斗是有固定的容量的，并且它能进水和流水，如果水满了，外面的水想要进来，就要等着水从漏嘴流出去。如果漏斗流水速率 < 灌水速率，那么漏斗最后将被灌满，漏斗满了就需要等待水流出去，腾出空间让水能灌进来。在漏斗限流算法中，除了漏斗容量之外，还需要漏斗剩余空间，流水速率，上次流水时间。漏斗剩余空

间代表当前还可以继续请求的数量，流水速率就代表请求频率限制（比如我们的需求就是 100/min），上次流水时间用于计算和当前时间的差值，然后用这个时间差计算这段时间内流水腾出了多少空间。

如果要用 Redis 实现，我们可以用 hash 类型作为漏斗容器，并将漏斗容量，流水速率，剩余空间，上次流水时间作为 field-value 添加进去。具体如何实现，还是直接看伪代码里的注释吧：

```
// 由《Redis深度历险 核心原理与应用实践》中的单机漏斗算法代码示例改编为 Redis 实现版本

class Funnel {
public:

    // 按理来说这些变量最好也在构造函数再初始化
    // 但是我们就写参数很多的构造函数来影响观感了~
    string funnel_key = "Funnel"; // 漏斗 key
    string capacity_field = "capacity"; // 容量
    string leaking_rate_field = "leaking_rate"; // 流水速率
    string left_quota_field = "left_quota"; // 剩余空间
    string leaking_ts_field = "leaking_ts_field"; // 上次流水时间
    long long period; // 时间窗口大小，用于设置过期时间，单位秒

    // 构造函数，到 Redis 里初始化漏斗数据
    Funnel(int capacity, long long period) {
        this->period = period;
        double leaking_rate = (double)capacity / period * 1000; // 计算流水速率
        Redis.Do("HSET", funnel_key, capacity_field, capacity);
        Redis.Do("HSET", funnel_key, leaking_rate_field, leaking_rate);
        Redis.Do("HSET", funnel_key, left_quota_field, capacity);
        Redis.Do("HSET", funnel_key, leaking_ts_field, GetNowUnixMillTime());
    }

    // 增加漏斗剩余空间函数
    void makeRoom() {
        int now = GetNowUnixMillTime(); // 获取当前毫秒时间戳
        long long leaking_ts = Redis.Do("HGET", funnel_key, leaking_ts_field);
        long long delta_ts = now - leaking_ts; // 计算上次流水到现在的时间差(毫秒)

        // 关键：计算上次流水到现在这段时间内流水腾出的空间
        int delta_quota = int(delta_ts * leaking_rate);

        // 负数情况是流速大且间隔时间过长，发生上溢了
        // 直接将剩余空间设置为漏斗容量
        if(delta_quota < 0) {
            int capacity = Redis.Do("HGET", funnel_key, capacity_field);
            Redis.Do("HSET", funnel_key, left_quota_field, capacity);
            Redis.Do("HSET", funnel_key, leaking_ts_field, GetNowUnixMillTime());
            return;
        }

        // 能腾出的空间太少，下次再来
        if(delta_quota < 1) return;

        int capacity = Redis.Do("HGET", funnel_key, capacity_field);
```

```

    int left_quota = Redis.Do("HGET", funnel_key, left_quota_field);

    // 更新上次流水时间为当前时间
    Redis.Do("HSET", funnel_key, leaking_ts_field, GetNowUnixMillTime());

    // 流水增加空间后，不能超过最大容量
    if(left_quota + delta_quota > capacity) {
        Redis.Do("HSET", funnel_key, left_quota_field, capacity);
        return;
    }

    // 增加空间
    Redis.Do("HINCRBY", funnel_key, left_quota_field, delta_quota);
}

// 向漏斗“灌水”函数
// 参数 quota 为灌水所需的空间
// 返回 true 表示漏斗剩余空间足够，可以“灌水”，false 则相反
bool watering(int quota) {
    makeRoom(); // 首先尝试腾出空间
    int left_quota = Redis.Do("HGET", funnel_key, left_quota_field);

    // 剩余空间足够的情况
    if(left_quota > quota) {

        // 令剩余空间减去灌水所需空间
        Redis.Do("HINCRBY", funnel_key, left_quota_field, -quota);

        // 请求成功后我们设置一个过期时间，过期时间为一个时间窗口大小
        // 这是为了自动释放冷门的限流 key
        Redis.Do("EXPIRE", funnel_key, period);
        return true;
    }
    return false;
}

};

void initFunnel(int capacity, double leaking_rate) {
    Funnel funnel = new Funnel(capacity, leaking_rate);
}

// 创建漏斗，全局变量
// 根据我们的需求，60s 内 100 次请求，容量是 100，
// 第二个参数是时间窗口大小，60s
Funnel funnel = initFunnel(100, 60);

// 模拟请求函数
void sendRequest(string query, string host) {

    // 每次请求尝试向漏斗倒入空间为 1 的水
    // 返回 true 则可以发起请求
    if(funnel.watering(1)) {

```

```
        sendMessage(query, host);
    }
}
```

咳咳，好像有点多，看起来也比计数器算法复杂。没事，你只需要知道漏斗算法可以用 hash 类型存储，还有漏斗算法两个核心（灌水和倒水）就 OK 了。到时候你再想想给你限制时间和最大请求数，你怎么计算出流水能腾出多少空间就好。

不过以上示例代码是有问题的，问题在哪呢？我们需要从 Redis 获取数据，处理后再放回去，这个过程不是具有隔离性的（可能其他人说原子性，但我比较严谨，原子性不仅操作无法分割，而且要么全部成功要么全部失败，前面事务分析已经说过 Redis 是无法做到的）。如果想要让这套操作具有隔离性，我们要么将整套逻辑和命令写成 lua 脚本交给 Redis 执行，要么写一个漏斗限流的 Redis Module 嵌入 Redis 使用。而前者相比后者简单，这下知道学一点 lua 语言的重要性了吧~

令牌桶算法

令牌桶算法和漏斗限流实现起来大同小异，它以固定速度往一个桶内添加令牌，当桶内装满令牌后就停止。这个令牌即是我们的流量单位，按我们的需求来说就是拿到一个令牌就能发起一次请求。同时，我们的令牌桶结构还是能用 hash 类型存储。那么咱们废话少说，直接来看示例：

```
class TokenBucket {
public:

    // 按理来说这些变量最好也在构造函数再初始化
    // 但是我们就写参数很多的构造函数来影响观感了~
    string key = "TokenBucket"; // 令牌桶 key
    string capacity_field = "capacity"; // 令牌容量 field
    string tokens_field = "tokens"; // 桶中令牌数量 field
    string lastAddTs_field = "lastAddTs"; // 上次增加令牌时间 field
    string rate_field = "rate"; // 放入令牌速率，单位 ms
    long long period; // 时间窗口大小，用于设置过期时间，单位秒

    // 构造函数
    TokenBucket(long long tokens, long long capacity, long long period) {
        this->period = period;
        double rate = (double)capacity / period * 1000; // 计算放入令牌速率
        Redis.Do("HSET", key, capacity_field, capacity);
        Redis.Do("HSET", key, tokens_field, tokens);
        Redis.Do("HSET", key, lastAddTs_field, GetNowUnixMillTime());
        Redis.Do("HSET", key, rate_field, rate);
    }

    // 获取令牌
    bool GetToken() {

        // 获取当前毫秒时间戳
        long long now = GetNowUnixMillTime();

        // 获取上次增加令牌时间
        long long lastAdd_ts = Redis.Do("HGET", key, lastAddTs_field);
```



```

// 计算与上次增加令牌的毫秒时间差
long long delta_ts = now - lastAdd_ts;

long long capacity = Redis.Do("HGET", key, capacity_field);
long long tokens = Redis.Do("HGET", key, tokens_field);
double rate = Redis.Do("HGET", key, rate_field);

long long delta_tokens = (long long)(delta_ts * rate)

// 负数则说明发生上溢
if(delta_tokens < 0) {
    Redis.Do("HSET", key, tokens_field, capacity);
    Redis.Do("HSET", key, lastAddTs_field, GetNowUnixMillTime());
    Redis.Do("HINCRBY", key, tokens_field, -1);
    return true;
}

// 没有令牌可用, 返回 false 表示获取失败
if(tokens + delta_tokens == 0) {
    return false;
}

// 更新上次增加令牌时间
Redis.Do("HSET", key, lastAddTs_field, GetNowUnixMillTime());

// 增加令牌后, 令牌数不能超过令牌桶容量
if(tokens + delta_tokens > capacity) {

    // 这里设置 capacity - 1, 因为获取了 1 个令牌
    Redis.Do("HSET", key, tokens_field, capacity - 1);

    // 请求成功后我们设置一个过期时间, 过期时间为一个时间窗口大小
    // 这是为了自动释放冷门的限流 key
    Redis.Do("EXPIRE", key, period);
    return true;
}

// 增加令牌数, 但是减去 1, 因为获取了 1 个令牌
Redis.Do("HINCRBY", key, tokens_field, delta_tokens - 1);

// 请求成功后我们设置一个过期时间, 过期时间为一个时间窗口大小
// 这是为了自动释放冷门的限流 key
Redis.Do("EXPIRE", key, period);
return true;
}
};

long long capacity = 100;
long long period = 60;
TokenBucket tokenBucket = TokenBucket(capacity, capacity, period);
// 模拟请求函数
void SendRequest(){

```

```
// 每次请求尝试从令牌桶中取出一块令牌
// 返回 true 则可以发起请求
if(tokenBucket.GetToken()) {
    sendMessage(query, host);
}
}
```

至此，三种常用限流算法都介绍完了。不过我要强调一下，以上限流算法逻辑最好还是封装成 lua 脚本，这样才能保证隔离性而且减少了传输次数，封装成 lua 脚本才是真正能用的 Redis 限流应用。不会写 lua，你就玩不了 Redis 高端应用的呢。（哎？你问我为什么没写成 lua，那不是为了给你们看清楚逻辑吗！学了 lua 后尝试把我这些逻辑翻译成 lua 脚本，那么你的 Redis lua 应用就基本合格了~）

最后还是用个简单的例子给大家看看 redis 的 lua 脚本是什么样的吧：

```
// Redis 执行 lua 脚本的命令：EVAL script numkey [key [key...]] [arg [arg...]]
// 包装 EXPIRE mykey 60 + INCR mykey 命令
// 返回 INCR 的返回值
// numkey 表示 key 数量，Redis 会根据 numkey 将之后 numkey 数量的参数当作 key
// lua 脚本用双引号包裹，使用 redis.call() 调用 redis 命令，命令用单引号包裹
// 脚本里 KEYS[1] 表示第一个 key，ARGV[1] 表示第一个 arg（参数）
EVAL "redis.call('EXPIRE', KEYS[1], ARGV[1]); return redis.call('INCR', KEYS[1]);" 1
mykey 60
```

记忆要点：限流算法，计数器固定窗口算法，计数器滑动窗口算法，漏斗限流算法，令牌桶算法

29. Redis 7 RESP 协议

前面我们已经将 Redis 底层原理介绍得差不多了，但需要较为全面了解 Redis 的话，还需要了解它的传输协议。Redis 使用的传输协议是 RESP (**RE**dis **S**erialization **P**rotocol)，虽然该协议是专门为 Redis 设计的，但它也可以用于其他 C-S 架构软件项目。

传输-响应模型

Redis 采用的是传输-响应模型，通过接收客户端的命令，将结果响应给客户端，这也是最简单的模型。但是也有两个例外情况：

1. Redis 支持流水线（pipeline）。也就是客户端可以一次发送多个命令并等待稍后的回复。
2. 当 Redis 客户端订阅 Pub/Sub 频道时，会转变为推送协议。客户端不再需要主动发送命令，因为服务器会在收到新消息后，会将消息推送到客户端订阅的频道。

RESP 协议

RESP 协议在 Redis 1.2 中引入，在 Redis 2.0 中成为与 Redis 服务器通信的标准方式。

Redis 通过以下方式使用 RESP 作为请求-响应协议：

- 客户端将命令作为 RESP 批量字符串数组发送到 Redis 服务器。
- 服务器根据命令实现使用其中一种 RESP 类型进行回复。

RESP 是一种序列化协议，支持五种数据类型：Simple Strings、Errors、Integers、Bulk Strings 和 Arrays，并且在 RESP 消息第一个字节标识了数据类型：

- 对于 Simple Strings（简单字符串），消息的第一个字节是 "+"
- 对于 Errors（错误消息），消息的第一个字节是 "-"
- 对于 Integers（整数），消息的第一个字节是 ":"
- 对于 Bulk Strings（批量字符串），消息的第一个字节是 "\$"
- 对于 Arrays（数组），消息的第一个字节是 "*"

RESP 消息结束符为 '\r\n' (CRLF)。

Simple Strings 示例：

简单字符串编码如下："+" 开头，后跟一个不能包含 '\r' 或 '\n' 字符的字符串（不允许有换行），并以 "\r\n" (CRLF) 结尾。

```
" +I Love Redis\r\n"
```

简单字符串因其编码较简单，所以传输开销较小，但是非二进制安全（字符串内容不能出现换行符）。如果要发送二进制安全的字符串，需要使用 Bulk Strings。

Bulk Strings 示例：

批量字符串用于表示长度最大为 512 MB 的单个字符串，并且是二进制安全的。

批量字符串格式如下：

- 以 "\$" 开头，后面跟着字符串长度/字节数，然后再以 "\r\n" 作为结尾。
- 实际的字符串。
- 最后以 "\r\n" 结束。

```
"$12\r\nI Love Redis\r\n"
```

空字符串编码为：

```
"$0\r\n\r\n"
```

NULL 值是长度为 -1 的特殊编码：

```
"$-1\r\n"
```

Integers 示例:

整数类型很简单, 以 ":" 开头, 后面跟着整数, 最后以 "\r\n" 结尾即可。

```
":23333\r\n"
```

Errors 示例:

错误类型与简单字符串相似, 只是将 "+" 换成了 "-" 开头。

```
"-Error message\r\n"
```

Arrays 示例:

客户端使用 RESP 数组向 Redis 服务器发送命令。对于某些将元素集合返回给客户端的 Redis 命令, Redis 服务器也会使用 RESP 数组作为它们的回复类型。

RESP 数据格式如下:

- 以 "*" 开头, 后面跟数组中的元素数量 (十进制), 然后是 "\r\n"。
- 数组的每个元素都是一种 RESP 类型。

```
*5\r\n:1\r\n:2\r\n:3\r\n:4\r\n$5\r\nhello\r\n
```

表示 [1,2,3,4,hello]

空数组:

```
"*0\r\n"
```

客户端发送命令给服务器使用的是仅包含 Bulk Strings 类型的 RESP 数组, 以 "GET mykey" 为例:

```
*2\r\n$3\r\nGET\r\n$5\r\nmykey\r\n
```

RESP 的特点:

RESP 协议的最主要特点就是它的**设计简单**, 所以**实现起来也简单**, 不容易出 bug, 而且它是**人类可读**的一种传输编码。此外, RESP **性能较好**, Redis 官方说的是在性能上能与二进制协议相当。

记忆要点：传输-响应模型，RESP 协议五种数据类型，RESP 特点，设计简单，实现简单，人类可读，性能较好

30. Redis 7 如何做内存优化

Redis 是内存型的数据库，所以存储成本是非常高的，我们在用 Redis 的时候当然希望能够优化内存占用，从而能够存储更多的数据。那么我们的最后一篇，就来讲讲如何做 Redis 的内存优化。

使用 32bit Redis

如果能够确定我们所要存储的数据不会超过 4G（32bit 最大寻址能力），那么我们可以编译并使用 32bit 的 Redis，能够比 64bit 的 Redis 要节省许多空间（32bit 指针是 4 字节，64bit 指针是 8 字节）。同时，RDB 和 AOF 文件在 32bit 和 64bit 下都是兼容的，所以我们可以放心在 32bit 和 64bit 之间切换。

尽量使用 listpack

我们在前面讲过，hash，list，set，zset 类型在存储较少的小元素时，会使用节约内存的 listpack 作为内部实现。所以为了节省内存，我们可以尽量使用这四种类型存储数据，并且分成多个小 key 存储。

当然，你也可以调整 listpack 的相关配置。比如使用 `CONFIG SET hash-max-listpack-entries 1024` 将 hash 类型 listpack 存储元素数量上限调高至 1024，`CONFIG SET hash-max-listpack-value 128` 将 hash 类型 listpack 存储单个元素最大长度调高至 128 字节。但是要注意，listpack 是顺序查找的，也就是查询一个元素时间复杂度是 $O(n)$ ，如果存储了太多元素性能会很差，所以我们也不要将上限调得太高。不过也不用担心元素较少的情况下，如果有大量查询会因为 $O(n)$ 将 Redis 打垮。别忘了还有个叫 CPU 缓存的东西，连续内存访问是能够好好利用它的，而它的速度比内存要高出至少一个数量级。

如果是 Redis 7 之前的版本，就是尽量使用 ziplist。

按位/字节操作

Redis 支持按位和按字节操作的命令：GETBIT、SETBIT、GETRANGE、SETRANGE。我们知道 QQ 群公告有个要群成员确认收到的功能，那么我们群管理也需要查看哪些人没有确认收到。假如我们为每个群成员安排一个递增 id，就可以使用 Redis 的位操作命令了。比如 id = 1 的成员确认收到了，就使用 `SETBIT Announcement_Confirm_List 1 1` 将第一位设置为 1。如果我们想知道哪些成员已经确认收到了，就遍历位图找出 bit 为 1 的位置就可以了，而如果是想知道哪些成员没确认收到，那么就是找 bit 为 0 的位置。只用 1 bit 大小就能存储一个信息，这已经是很极致利用内存的优化了。

尽量使用 Hash 类型

对于一些具有几个字段的结构体数据，比如 JSON 数据，我们尽量将它使用 hash 类型进行存储，这远比将每个字段单独用 string 类型存储节省内存要多得多。它能节省内存的原因还是上面提到过的 listpack 结构，Redis 官方做过的测试表明，小对象使用 hash 类型存储要比将字段分开用 string 存储节省了一个数量级的空间。

Redis 内存分配

这是最后的部分了，我们先来说说 Redis 内存管理。

Redis 内存管理并不是自己实现的，为了省事直接使用了现成的第三方内存管理库。Redis 默认使用了效果较好的 jemalloc，除此之外还可以使用 libc，但是效果可能不如 jemalloc。如果你了解 Redis 底层的内存管理，那么你可以去搜索一篇讲 jemalloc 的文章看一看。

然后我们讲讲该如何为 Redis 分配内存比较好。

Redis 可以通过设置 maxmemory 来限制最大可分配内存（但是可能会有少量额外分配）。当删除 key 时，Redis 不一定能释放内存给操作系统。例如，如果我们有 5GB 的数据，然后删除 2GB 的数据，RSS（Resident Set Size，进程消耗的内存页数）可能仍然大约为 5GB。这不是 Redis 的特殊之处，这其实是大多数内存管理器的工作方式。原因在之前那篇内存淘汰策略里我也提过了，也就是被删除 key 所在的页面还有其它的 key，只要还在被引用就没法被回收。

上面所说的内容意味着我们应该以**内存最大峰值**配置 maxmemory，如果我们预估内存最大存储峰值是 10G，即使平时只用上 5GB，那么也该配置为 10GB。（这个道理好像讲得跟废话一样，因为大家都知道，嘻嘻）

还要提到的一点是，内存分配器能够重用空闲内存块，因此在删除 5GB 数据集中的 2GB 之后，当我们开始添加更多 key 时，我们会看到 RSS 保持稳定，不会增长更多，因此分配器基本上是在尝试重用之前释放（逻辑上释放，没有真正交给操作系统）的 2GB 内存。

如果 maxmemory 未设置，Redis 将继续分配它认为合适的内存，直到它耗尽所有可用内存，所以通常建议配置一些限制。除此之外我们还可以设置 maxmemory-policy，也就是内存淘汰策略。

记忆要点：内存优化方案，使用 32bit Redis，尽量使用 listpack，按位/字节操作，尽量使用 hash 类型存储多字段结构数据
