一、redis基础

1. 基础数据类型

- string
- list
  - blpop/brpop，阻塞读在队列没有数据的时候，会立即进入休眠状态，一旦数据到来，则立刻醒过来。消息的延迟几乎为零
    - 阻塞度的问题：**空闲连接**的问题。如果线程一直阻塞在哪里，Redis 的客户端连接就成了闲置连接，闲置过久，服务器一般会主动断开连接，减少闲置资源占用。这个时候`blpop/brpop`会抛出异常来
- hash
- set
- zset

2. 扩展数据类型

HyperLogLog

用途
HyperLogLog 数据结构就是用来解决这种统计问题的。HyperLogLog 提供不精确的去重计数方案，虽然不精确但是也不是非常不精确，标准误差是 0.81%
指令

- 两个指令 pfadd 和 pfcount，根据字面意义很好理解，一个是增加计数，一个是获取计数
- 第三个指令 pfmerge，用于将多个 pf 计数值累加在一起形成一个新的 pf 值
  成本
- 需要占据 12k 的存储空间
  - 计数比较小时，它的存储空间采用稀疏矩阵存储
  - 仅仅在计数慢慢变大，稀疏矩阵占用空间渐渐超过了阈值时才会一次性转变成稠密矩阵，才会占用 12k 的空间

> HyperLogLog 实现中用到的是 16384 个桶，也就是 2^14，每个桶的 maxbits 需要 6 个 bits 来存储，最大可以表示 maxbits=63，于是总共占用内存就是2^14 * 6 / 8 = 12k字节

Bitmap

- 位图统计指令 bitcount 和位图查找指令 bitpos

GEO

- 位置信息服务（Location-Based Service，LBS）的应用
- GEO类型的底层数据结构就是用Sorted Set来实现的
- 对经度和纬度分别编码，然后再把经纬度各自的编码组合成一个最终编码
    - 对经度范围[-180,180]做N次的二分区操作
        - 经度范围[-180,180]被分成两个子区间：[-180,0)和[0,180]（左、右分区）。可以查看要编码的经度值，如果是落在左分区，就用0表示；如果落在右分区，就用1表示。
    - 对纬度的编码方式，和对经度的一样
    - 最终编码值的偶数位上依次是经度的编码值，奇数位上依次是纬度的编码值，其中，偶数位从0开始，奇数位从1开始
- 使用GeoHash编码后，我们相当于把整个地理空间划分成了一个个方格，每个方格对应了GeoHash中的一个分区
- 命令
    - GEOADD命令：用于把一组经纬度信息和相对应的一个ID记录到GEO类型集合中
    - GEORADIUS命令：会根据输入的经纬度位置，查找以这个经纬度为中心的一定范围内的其他元素

3. 特殊功能

pub-sub

> 消息多播允许生产者生产一次消息，中间件负责将消息复制到多个消息队列，每个消息队列由相应的消费组进行消费。

redis的list不支持消息多播

Redis单独使用了一个模块来支持消息多播，这个模块的名字叫着 PubSub，也就是发布者订阅者模型。

- 订阅
    - 基于名称订阅
    - 支持批量订阅
    - 支持模式订阅，订阅的主题使用正则表达式
- 缺点
    - 如果 Redis 停机重启，PubSub 的消息是不会持久化的，毕竟 Redis 宕机就相当于一个消费者都没有，所有的消息直接被丢弃
    - 如果消费者掉线，掉线后所有的消息都会丢失，无法找回了

布隆过滤器

布隆过滤器 (Bloom Filter) 专门用来解决去重问题。它在去重的同时，在空间上还能节省 90% 以上，只是有一定的误判概率。

当布隆过滤器说某个值存在时，这个值可能不存在；当它说不存在时，那就肯定不存在。
原理

- 每个布隆过滤器对应到 Redis 的数据结构里面就是一个大型的位数组和几个不一样的无偏 hash 函数。所谓无

偏就是能够把元素的 hash 值算得比较均匀。

- 向布隆过滤器中添加 key 时，会使用多个 hash 函数对 key 进行 hash 算得一个整数索引值然后对位数组长度进行取模运算得到一个位置，每个 hash 函数都会算得一个不同的位置。再把位数组的这几个位置都置为 1 就完成了 add 操作

空间占用
n表示预计元素的数量，f 表示错误率

```
k=0.7*(l/n)  # 约等于
f=0.6185^(l/n)  # ^ 表示次方计算，也就是 math.pow
```

hash 函数的数量也会直接影响到错误率，最佳的数量会有最低的错误率

其他的限流方案

1. 简单限流
    - 用zset实现一个滑动窗口，score是时间戳，缺点是每个用于都要维护一个，空间消耗大

stream

Streams是Redis 5.0专门针对消息队列场景设计的数据类型(Stream 狠狠地借鉴了 Kafka 的设计)

- XADD：插入消息，保证有序，可以自动生成全局唯一ID；
  XREAD：用于读取消息，可以按ID读取数据；（使用 `xread` 时，我们可以完全忽略消费组 (Consumer Group) 的存在，就好比 Stream 就是一个普通的列表 (list)）
  XREADGROUP：按消费组形式读取消息；
  XPENDING和XACK：XPENDING命令可以用来查询每个消费组内所有消费者已读取但尚未确认的消息，而XACK命令用于向消息队列确认消息处理已完成。
- 幂等性
    - 可以自动生成消息id，毫秒时间戳+自增id；也可以手动指定消息id
- 可靠性
    - Streams会自动使用内部队列（也称为PENDING List）留存消费组里每个消费者读取的消息，直到消费者使用XACK命令通知Streams“消息已经处理完成”。用来确保客户端至少消费了消息一次，而不会在网络传输的中途丢失了没处理。
    - Stream 的高可用是建立主从复制基础上的，它和其它数据结构的复制机制没有区别，也就是说在 Sentinel 和 Cluster 集群环境下 Stream 是可以支持高可用的。但也有可能会丢失部分数据。
- 总结

  Stream 的消费模型借鉴了 Kafka 的消费分组的概念，它弥补了 Redis Pub/Sub 不能持久化消息的缺陷。但是它又不同于 kafka，Kafka 的消息可以分 partition，而 Stream 不行。如果非要分 parition 的话，得在客户端做，提供不同的 Stream 名称，对消息进行 hash 取模来选择往哪个 Stream 里塞。

4. 缓存基础使用

特征

- 计算机三层存储
  - cpu，20~40ns
  - 内存，100ns
  - 硬盘，3~5ms
- 特征
  - 快速
  - 容量小于后端系统，无法存放所有数据
- Redis天然就具有高性能访问和数据淘汰机制，正好符合缓存的这两个特征的要求，所以非常适合用作缓存

使用方式

- 只读缓存
  - 特性
    - 读数据直接查redis
    - 写数据操作数据库，删除redis缓存
  - 优势：数据库和缓存一致
  - 劣势：读库重写缓存响应会慢
- 读写缓存
  - 特性
    - 读请求查redis
    - 写请求也会写缓存
  - 两种策略
    - 同步直写（写缓存同时写数据库）
    - 异步回写（缓存将要淘汰时回写数据库）
- 对比
  - 如果需要严格保证缓存和数据库的一致性，即保证两者操作的原子性，这就涉及到分布式事务问题了，常见的解决方案就是我们经常听到的两阶段提交（2PC）、三阶段提交（3PC）、TCC、消息队列等方式来保证了，方案也会比较复杂，一般用在对于一致性要求较高的业务场景中
  - 只读缓存是牺牲了一定的性能，优先保证数据库和缓存的一致性，它更适合对于一致性要求比较要高的业务场景。而如果对于数据库和缓存一致性要求不高，或者不存在并发修改同一个值的情况，那么使用读写缓存就比较合适，它可以保证更好的访问性能

缓存常见问题

- 数据一致性
  - 读写缓存
    - 要想保证缓存和数据库中的数据一致，就要采用同步直写策略。要在业务应用中使用事务机制，来保证缓存和数据库的更新具有原子性
  - 只读缓存
    - 删除缓存值或更新数据库失败而导致数据不一致，你可以使用重试机制确保删除或更新操作成功
    - 在删除缓存值、更新数据库的这两步操作中，有其他线程的并发读操作，导致其他线程读取到旧

值，应对方案是延迟双删

- 缓存雪崩
  大量的应用请求无法在Redis缓存中进行处理，大量请求被发送到数据库层，导致数据库层的压力激增
  - 原因
    - 大量key同时过期
    - 方案
      - 过期时间加随机值
      - 服务降级
        - 非核心数据返回预定义值，空值或错误
        - 核心数据继续读库
  - redis宕机
    - 方案
      - 业务中添加熔断/限流机制
        - 业务应用调用缓存接口时，缓存客户端并不把请求发给Redis缓存实例，而是直接返回
      - 事先预防：redis高可用集群
- 缓存击穿
  - 针对**某个热点数据**的请求，无法在缓存中处理，大量请求，都发送到了后端数据库，导致数据库压力激增
  - 经常发生在热点数据过期失效时
  - 方案
    - 热点数据就不设置过期时间了
- 缓存穿透
  要访问的数据既不在Redis缓存中，也不在数据库中
  - 场景
    - 业务误操作，数据被删除了
    - 恶意攻击
  - 方案
    - 缓存空值或默认值
    - 布隆过滤器先判断数据是否存在
    - 前端过滤
- 缓存污染
  有些数据被访问的次数非常少，甚至只会被访问一次。当这些数据服务完访问请求后，如果还继续留存在缓存中的话，就只会白白占用缓存空间。这种情况，就是缓存污染
  - 把冷数据逐步淘汰出缓存，这就会引入额外的操作时间开销，进而会影响应用的性能
  - 方案
    - 使用数据淘汰策略
      - volatile-random和allkeys-random，无效
      - volatile-ttl，除了在明确知道数据被再次访问的情况下，volatile-ttl可以有效避免缓存污染。在其他情况下效果都一般
      - lru，因为只看数据的访问时间，使用LRU策略在处理扫描式单次查询操作时，无法解决缓存

        污染
    - lfu

5. 其他知识点

事务

1. 事务的使用

- 第一步，MULTI 命令开启事务
- 第二步，客户端把事务中本身要执行的具体操作（例如增删改数据）发送给服务器端
- 第三步，客户端向服务器端发送提交事务的命令 EXEC

2. redis事务的acid

- 原子性
    - 事务操作入队时，命令和操作的数据类型不匹配，但Redis实例没有检查出错误
    - **错误的命令被跳过，其他命令正常执行，原子性无法保证**
    - redis没有事务回滚，DISCARD命令只是清空命令队列，主动放弃事务执行
    - EXEC命令执行时实例故障，如果开启了AOF日志，可以保证原子性，会过滤事务的命令
- 一致性
    - 在命令执行错误或Redis发生故障的情况下，Redis事务机制对一致性属性是有保证的
- 隔离性
    - 事务的EXEC命令还没有执行时，事务的命令操作是暂存在命令队列中的。此时，如果有其它的并发操作，我们就需要看事务是否使用了WATCH机制
    - WATCH机制的作用是，在事务执行前，监控一个或多个键的值变化情况，当事务调用EXEC命令执行时，WATCH机制会先检查监控的键是否被其它客户端修改了。如果修改了，就放弃事务执行，避免事务的隔离性被破坏。
- 持久性
    - 不管Redis采用什么持久化模式，事务的持久性属性是得不到保证的
    - redis常用于缓存，持久性相对不关注

内存碎片

形成原因

- 内因：内存分配器的分配策略
    - 分配方式有形成碎片的风险
- 外因：键值对大小不一样和删改操作
    - jemalloc的分配策略之一，是按照一系列固定的大小划分内存空间，有一定的浪费
    - 键值对会被修改和删除，这会导致空间的扩容和释放

判断是否有内存碎片

- info命令的mem_fragmentation_ratio的指标，它表示的就是Redis当前的内存碎片率
    - 操作系统实际分配给Redis的物理内存空间(里面就包含了碎片)/used_memory是Redis为了保存数据实际申请使用的空间
- 判断依据(经验阈值)

    - mem_fragmentation_ratio 大于1但小于1.5，相对合理
    - mem_fragmentation_ratio 大于 1.5，不合理，需要处理
    - 小于1，说明内存不够用，被swap了，对redis性能有很大影响，需要马上处理

方案

- 重启redis
- 4.0版本后支持自动清理
    - 数据转移，合并空间
    - 是有代价的
        - 数据拷贝，处理逻辑等都要占用主线程
    - 通过配置项设置
        - 启动清理的阈值
            - 占用空间
            - 碎片占用比例

缓冲区

1. 客户端缓冲区

- 服务器端给每个连接的客户端都设置了一个输入缓冲区和输出缓冲区，我们称之为客户端输入缓冲区和输出缓冲区
- 输入缓冲区会先把客户端发送过来的命令暂存起来，Redis主线程再从输入缓冲区中读取命令，进行处理。当Redis主线程处理完数据后，会把结果写入到输出缓冲区，再通过输出缓冲区返回给客户端

2. 输入缓冲区

- 溢出场景
    - 写入bigkey
    - 服务端处理过慢
- 判断
    - CLIENT LIST命令
- 后果
    - 客户端输入缓冲区溢出，Redis的处理办法就是把客户端连接关闭，结果就是业务程序无法进行数据存取了
    - 导致Redis内存占用过大，也会导致内存溢出（out-of-memory）问题，进而会引起Redis崩溃，给业务应用造成严重影响
- 方案
    - Redis并没有提供参数让我们调节客户端输入缓冲区的大小
    - 只能从数据命令的发送和处理速度入手，也就是避免客户端写入bigkey，以及避免Redis主线程阻塞

3. 输出缓冲区

- Redis为每个客户端设置的输出缓冲区也包括两部分：一部分，是一个大小为16KB的固定缓冲空间，用来暂存OK响应和出错信息；另一部分，是一个可以动态增加的缓冲空间，用来暂存大小可变的响应结果
- 溢出场景
    - 返回bigkey

    - 大小设置不合理
    - 执行了MONITOR命令（用来监测Redis执行的）
- 分类
    - 普通客户端
    - 订阅客户端
    - 从节点客户端
- 后果
    - 服务器端也会关闭客户端连接
- 方案
    - 避免大key
    - 不要再线上环境中持续使用MONITOR命令
    - 设置缓冲区上限，或者写入时间上限

4. 主从同步缓冲区

- 后果
    - 全量复制失败
- 方案
    - 控制主节点数据量大小
    - 设置合理的复制缓冲区大小
    - 控制从节点的数量

5. 增量同步缓冲区

- 设置合理的大小

并发

并发的两种方案

- 加锁
    - 降低并发
    - 引入分布式锁
- 原子操作
    - 单命令操作
        - 适用范围较小
    - 原子方式执行lua脚本
        - 在编写Lua脚本时，你要避免把不需要做并发控制的操作写入脚本中

安全

1. 指令安全
    - Redis 有一些非常危险的指令，这些指令会对 Redis 的稳定以及数据安全造成非常严重的影响。比如 `keys` 指令会导致 Redis 卡顿，`flushdb` 和 `flushall` 会让 Redis 的所有数据全部清空。
    - Redis 在配置文件中提供了 `rename-command` 指令用于将某些危险的指令修改成特别的名称，用来避免人为误操作。如果想完全封杀某条指令，可以将指令 `rename` 成空串。

2. 端口安全
    - 务必在 Redis 的配置文件中指定监听的 IP 地址
    - 增加 Redis 的密码访问限制
3. Lua 脚本安全
    - 禁止 Lua 脚本由用户输入的内容 (UGC) 生成
4. SSL 代理
    - Redis 并不支持 SSL 链接，不过 Redis 官方推荐SSL 代理使用 spiped 工具
        - spiped 会在客户端和服务器各启动一个 spiped 进程，客户端的spiped 进程 A 负责接受来自 Redis Client 发送过来的请求数据，加密后传送到服务端的 spiped 进程 B。spiped B 将接收到的数据解密后传递到 Redis Server。

6. redis常见使用场景

消息队列

- 3个需求
    - 消息保证有序
    - 重复消息处理：幂等性
    - 可靠性：宕机不丢数据
- list方案
    - lpush生产，rpop消费
    - brpop，阻塞读
        - 队列没有数据时自动阻塞
    - 幂等性
        - 生产者生成唯一id，消费者判断是否消费过
    - 可靠性
        - 消费者拿到数据还没消费挂掉了
        - Brpoplpush，数据保存到备份队列中
    - 不支持消费者组，消费慢时没办法
- streams方案
    - Streams是Redis 5.0专门针对消息队列场景设计的数据类型
    - XADD：插入消息，保证有序，可以自动生成全局唯一ID；
      XREAD：用于读取消息，可以按ID读取数据；
      XREADGROUP：按消费组形式读取消息；
      XPENDING和XACK：XPENDING命令可以用来查询每个消费组内所有消费者已读取但尚未确认的消息，而XACK命令用于向消息队列确认消息处理已完成。
    - 幂等性
        - 可以自动生成消息id，毫秒时间戳+自增id；也可以手动指定消息id
    - 可靠性
        - Streams会自动使用内部队列（也称为PENDING List）留存消费组里每个消费者读取的消息，直到消费者使用XACK命令通知Streams“消息已经处理完成”。
- 队列常见消息问题

    - 生产时
        - 发送失败，重试若干次后放弃
        - 网络抖动，没收到消息确认，重复发送数据
            - 要求幂等性
    - 消费时
        - 拿到消费还没消费就挂掉了
            - 消费成功发送ack给消息队列
    - 消息队列丢数据
        - redis可能丢数据
            - aof刷盘时间间隔内丢数据
            - 主从切换过程中丢数据
- redis可以用作轻量级的队列，且有一定概率丢数据，需要业务上确认是否接受

秒杀

- 特征
    - 瞬时并发访问量非常高
        - 使用Redis先拦截大部分请求，避免大量请求直接发送给数据库，把数据库压垮。
    - 读多写少，而且读操作是简单的查询操作

> 秒杀系统最重要的是，把大部分请求拦截在最前面，只让很少请求能够真正进入到后端系统，降低后端服务的压力，常见的方案包括：页面静态化（推送到CDN）、网关恶意请求拦截、请求分段放行、缓存校验和扣减库存、消息队列处理订单

- 流程
    - 活动前
        - 把商品详情页的页面元素静态化，然后使用CDN或是浏览器把这些静态化的元素缓存起来
    - 活动开始
        - 库存查验、库存扣减和订单处理
        - 订单处理可以在数据库中执行，库存扣减不能在数据库处理
            - 下单量超过实际库存量，出现超售
        - 需要直接在Redis中进行库存扣减，为了避免请求查询到旧的库存值，库存查验和库存扣减这两个操作需要保证原子性
- redis实现
    - 基于原子操作支撑秒杀场景
        - lua脚本
    - 基于分布式锁来支撑秒杀场景
        - 使用分布式锁来保证多个客户端能互斥执行这两个操作

分布式锁

锁是保存在一个共享存储系统中的，可以被多个客户端共享访问和获取

- 单机上的锁
  - 检查锁变量值是否为0。如果是0，就把锁的变量值设置为1，表示获取到锁，如果不是0，就返回错误信息，表示加锁失败
- 分布式锁的两个要求
  - 分布式锁的加锁和释放锁的过程，涉及多个操作。所以，在实现分布式锁时，我们需要保证这些锁操作的原子性
  - 共享存储系统保存了锁变量，如果共享存储系统发生故障或宕机，那么客户端也就无法进行锁操作了。在实现分布式锁时，我们需要考虑保证共享存储系统的可靠性，进而保证锁的可靠性
- 单节点
  - 可以用SETNX和DEL命令组合来实现加锁和释放锁操作
    - 风险点
      - 没有释放锁，方案:加过期时间
      - 其他进程误删锁，方案：锁的value设置有意义的值
      - 删除锁操作的非原子性，方案：释放锁用lua脚本
- 多节点
  - 为了避免Redis实例故障而导致的锁无法工作的问题，Redis的开发者Antirez提出了分布式锁算法Redlock
  - 让客户端和多个独立的Redis实例依次请求加锁，如果客户端能够和半数以上的实例成功地完成加锁操作，那么我们就认为，客户端成功地获得分布式锁了，否则加锁失败
  - 步骤
    - 第一步是，客户端获取当前时间。
    - 第二步是，客户端按顺序依次向N个Redis实例执行加锁操作
    - 第三步是，一旦客户端完成了和所有Redis实例的加锁操作，客户端就要计算整个加锁过程的总耗时
  - 成功的两个条件
    - 客户端从超过半数（大于等于 N/2+1）的Redis实例上成功获取到了锁
    - 客户端获取锁的总耗时没有超过锁的有效时间
  - 在实际的业务应用中，如果你想要提升分布式锁的可靠性，就可以通过Redlock算法来实现
- 加锁失败
  - 直接抛出异常，通知用户稍后重试；
  - sleep 一会再重试；
  - 将请求转移至延时队列，过一会再试；

用redis保存时间序列数据

描述
周期性的统计数据，并提供多维度的聚合查询
特点
- 这种数据的写入特点很简单，就是插入数据快，这就要求我们选择的数据类型，在进行数据插入时，复杂度要低，尽量不要阻塞

- 在查询时间序列数据时，既有对单条记录的查询，也有对某个时间范围内的数据的查询，还有一些更复杂的查询，比如对某个时间范围内的数据做聚合计算。

方案1：Hash和Sorted Set

- 优势
  - hash可以实现对单键的快速查询
  - zset支持按时间戳范围的查询
- 问题
  1. 如何保证写入Hash和Sorted Set是一个原子性的操作呢
     - Redis用来实现简单的事务的MULTI和EXEC命令
  2. 怎么聚合计算
     - 只能先把时间范围内的数据取回到客户端，然后在客户端自行完成聚合计算
  3. 潜在风险，也就是大量数据在Redis实例和客户端间频繁传输，这会和其他操作命令竞争网络资源，导致其他操作变慢

方案二：RedisTimeSeries

RedisTimeSeries是Redis的一个扩展模块。它专门面向时间序列数据提供了数据类型和访问接口，并且支持在Redis实例上直接对数据进行按时间范围的聚合计算（不属于Redis的内建功能模块，在使用时，我们需要先把它的源码单独编译成动态链接库redistimeseries.so，再使用loadmodule命令进行加载）

- 优势
  - 支持直接在Redis实例上进行多种数据聚合计算，避免了大量数据在实例和客户端间传输
- 劣势
  - 范围查询的复杂度是O(N)级别的，同时，它的TS.GET查询只能返回最新的数据，没有办法像第一种方案的Hash类型一样，可以返回任一时间点的数据

7. redis新特性

1. 从单线程处理网络请求到多线程处理
   随着网络硬件的性能提升，Redis的性能瓶颈有时会出现在网络IO的处理上，也就是说，单个主线程处理网络请求的速度跟不上底层网络硬件的速度

- 两个方案
  - 用用户态网络协议栈（例如DPDK）取代内核网络协议栈。对redis而言修改多，风险大
  - 采用多个IO线程来处理网络请求。Redis的多IO线程只是用来处理网络请求的，对于读写命令，Redis仍然使用单线程来处理
- 步骤
  - 服务端和客户端建立Socket连接，并分配处理线程
  - IO线程读取并解析请求
  - 主线程执行请求操作
  - IO线程回写Socket和主线程清空全局队列

2. 实现服务端协助的客户端缓存
   存在一个问题，如果数据被修改了或是失效了，如何通知客户端对缓存的数据做失效处理?

- 两种模式
  - 普通模式

    - 广播模式

3. 从简单的基于密码访问到细粒度的权限控制
   除了设置某个命令或某类命令的访问控制权限，6.0版本还支持以key为粒度设置访问权限
4. 启用RESP 3协议
5. 下一步规划

- 基于NVM内存的实践
    - 新型非易失存储（Non-Volatile Memory，NVM）器件
    - NVM的特性
        - 最大的优势是可以直接持久化保存数据
        - 访问速度接近DRAM的速度
        - NVM内存的容量很大

二、redis的高性能、高稳定，可扩展

1. 高性能

高性能的原因

- 单线程

> 1. Redis的网络IO和键值对读写是由一个线程来完成的，这也是Redis对外提供键值存储服务的主要流程。但Redis的其他功能，比如持久化、异步删除、集群数据同步等，其实是由额外的线程执行的。
> 2. 多线程编程模式面临共享资源的并发访问控制问题，一般会引入同步原语来保护共享资源的并发访问，这也会降低系统代码的易调试性和可维护性

- 纯内存操作
- io多路复用

> 允许内核中，同时存在多个监听套接字和已连接套接字，内核会一直监听这些套接字上的连接请求或数据请求。一旦有请求到达，就会交给Redis线程处理
>
> select/epoll提供了基于事件的回调机制，即针对不同事件的发生，调用相应的处理函数

- 高效的数据结构

有哪些因素会影响到性能

- 大key删除
- key集中过期
- 清空数据库
- 从库加载rdb文件
- aof日志同步写
- 复杂命令，比如集合求交集、并集、差集等
- Redis实例运行机器上开启了透明内存大页机制
- 进程绑定CPU不合理
- Redis实例运行机器的内存不足，导致swap发生，Redis需要到swap分区读取数据

- 当Redis实例的数据量大时，无论是生成RDB，还是AOF重写，都会导致fork耗时严重
- 内存达到maxmemory
- 网卡压力过大(存疑)
- 客户端使用短连接来链接Redis(存疑)

cpu对redis的影响

- 主流的cpu架构
    - 物理核
        - 私有的一级缓存L1
        - 私有的二级缓存L2（10纳秒延迟，只有kb级别）
        - 多核公用的L3缓存（几M到几十M的级别）
        - 内存访问（百纳秒级别）
    - 逻辑核
        - 每个物理核通常都会运行两个超线程，也叫作逻辑核
        - 同一个物理核的逻辑核会共享使用L1、L2缓存
    - cpu处理器
        - 在主流的服务器上，一个CPU处理器会有10到20多个物理核。同时，为了提升服务器的处理能力，服务器上通常还会有多个CPU处理器（也称为多CPU Socket），每个处理器有自己的物理核（包括L1、L2缓存），L3缓存，以及连接的内存，同时，不同处理器间通过总线连接
        - 在多CPU架构上，应用程序可以在不同的处理器上运行。
        - 应用程序先在一个Socket上运行，并且把数据保存到了内存，然后被调度到另一个Socket上运行，此时，应用程序再进行内存访问时，就需要访问之前Socket上连接的内存，这种访问属于**远端内存访问**。和访问Socket直接连接的内存相比，远端内存访问会增加应用程序的延迟
        - 在多CPU架构下，一个应用程序访问所在Socket的本地内存和访问远端内存的延迟并不一致，所以，我们也把这个架构称为非统一内存访问架构（Non-Uniform Memory Access，NUMA架构）
        - 在CPU的NUMA架构下，对CPU核的编号规则，并不是先把一个CPU Socket中的所有逻辑核编完，再对下一个CPU Socket中的逻辑核编码，而是先给每个CPU Socket中每个物理核的第一个逻辑核依次编号，再给每个CPU Socket中的物理核的第二个逻辑核依次编号
- CPU架构对应用程序的影响
    - L1、L2缓存中的指令和数据的访问速度很快，所以，充分利用L1、L2缓存，可以有效缩短应用程序的执行时间
    - 在NUMA架构下，如果应用程序从一个Socket上调度到另一个Socket上，就可能会出现远端内存访问的情况，这会直接增加应用程序的执行时间
    - 应用程序被从一个cpu核调度到另一个cpu核的时候，每调度一次，一些请求就会受到运行时信息、指令和数据重新加载过程的影响，这就会导致某些请求的延迟明显高于其他请求。
    - 在多核CPU架构下，Redis如果在不同的核上运行，就需要频繁地进行上下文切换，这个过程会增加Redis的执行时间，客户端也会观察到较高的尾延迟了。所以，建议你在Redis运行时，把实例和某个核绑定，这样，就能重复利用核上的L1、L2缓存，可以降低响应延迟
- 如果网络中断处理程序和Redis实例各自所绑的CPU核不在同一个CPU Socket上，那么，Redis实例读取网络数据时，就需要跨CPU Socket访问内存，这个过程会花费较多时间
    - 最好把网络中断程序和Redis实例绑在同一个CPU Socket上，这样一来，Redis实例就可以直接从本地内存读取网络数据了
- redis绑定到cpu核
    - 问题

        - 把Redis实例绑到一个CPU逻辑核上时，就会导致子进程、后台线程和Redis主线程竞争CPU资源，一旦子进程或后台线程占用CPU时，主线程就会被阻塞，导致Redis请求延迟增加
    - 方案
        - 一个Redis实例对应绑一个物理核
            - 让主线程、子进程、后台线程共享逻辑核，可以在一定程度上缓解CPU资源竞争
        - 优化Redis源码
            - 把子进程和后台线程绑到不同的CPU核上

redis变慢怎么办

- 判断是否真的变慢了
    - 查看Redis的响应延迟
    - 基于当前环境下的Redis基线性能做判断
        - 从2.8.7版本开始，redis-cli命令提供了–intrinsic-latency选项，可以用来监测和统计测试期间内的最大延迟，这个延迟可以作为Redis的基线性能
        - 这个命令需要在服务器端直接运行，这也就是说，我们只考虑服务器端软硬件环境的影响
- 三大要素分析
    - redis自身特性
        - 慢查询命令
            - 根据工具或日志判断是否有慢查询命令
            - 方案
                - 换高效的命令
                - 在客户端进行数据处理
        - 过期key操作
            - 过期策略
                - 采样若干个key，删除过期key
                - 超过25%key过期，重复操作
            - 删除操作是阻塞的
                - Redis 4.0后可以用异步线程机制来减少阻塞影响
            - 常见就是key集中过期
    - 文件系统
        - redis持久化时，文件系统会将数据写回磁盘
        - aof3种模式
            - no，系统调用write
            - everysec，系统调用fsync，每秒调用一次
            - always，每个命令调用一次fsync
        - 风险点
            - aof重写时会对磁盘进行大量IO操作，可能会导致fsync被阻塞
            - 当主线程使用后台子线程执行了一次fsync，需要再次把新接收的操作记录写回磁盘时，如果主线程发现上一次的fsync还没有执行完，那么它就会阻塞。所以，如果后台子线程执行的fsync频繁阻塞的话（比如AOF重写占用了大量的磁盘IO带宽），主线程也会阻塞，导致Redis

性能变慢
        - 方案
            - 调整aof模式
            - 采用高速的固态硬盘作为AOF日志的写入设备
    - 操作系统
        - 内存swap
            - 内存swap是操作系统里将内存数据在内存和磁盘间来回换入和换出的机制，涉及到磁盘的读写，所以，一旦触发swap，无论是被换入数据的进程，还是被换出数据的进程，其性能都会受到慢速磁盘读写的影响
            - swap触发后影响的是Redis主IO线程，这会极大地增加Redis的响应时间
            - 通常，触发swap的原因主要是物理机器内存不足
                - Redis实例自身使用了大量的内存，超过物理内存限制
                - 和Redis实例在同一台机器上运行的其他进程，在进行大量的文件读写操作，占用较多内存
            - swap的大小是排查Redis性能变慢是否由swap引起的重要指标
            - 方案
                - 增加内存
                - 使用集群
        - 内存大页
            - 该机制支持2MB大小的内存页分配，而常规的内存页分配是按4KB的粒度来执行的
            - 问题
                - 写时复制涉及到内存大页，就会导致额外的大量复制
            - 方案
                - 关掉内存大页
        - 大量短连接

2. 高稳定

数据持久化+故障恢复

aof

要点

- AOF里记录的是Redis收到的每一条命令，这些命令是以文本形式保存的
- AOF日志是写后日志，“写后”的意思是Redis是先执行命令，把数据写入内存，然后才记录日志
    - 为了避免额外的检查开销，AOF日志的时候，并不会先去对这些命令进行语法检查。可以避免出现记录错误命令
    - 在命令执行后才记录日志，所以不会阻塞当前的写操作

风险

- 执行完命令未写日志宕机，日志丢失
- 主线程操作，可能阻塞下一个写操作

分析

风险都是和AOF写回磁盘的时机相关的

回写磁盘的策略

- 同步回写，命令执行完立刻同步
- 每秒回写
- 操作系统控制，redis只写到内存缓冲区

aof重写

- AOF文件过大会带来性能问题
    - 操作系统的文件大小限制
    - 大文件新增数据会慢
    - 故障恢复耗时长
- aof是追加写
    - 重写能把同一个key的多个日志记录合并成一条
- 重写过程是由后台线程bgrewriteaof来完成的
- 过程
    - 一个拷贝，两处日志
        - 每次执行重写时，主线程fork出后台的bgrewriteaof子进程。此时，fork会把主线程的内存拷贝一份给bgrewriteaof子进程，这里面就包含了数据库的最新数据。
            - fork瞬间会阻塞主线程
            - fork采用操作系统提供的写时复制(Copy On Write)机制，避免大量copy导致阻塞
            - 写操作时，会出现内存复制，可能阻塞主线程
        - 主线程未阻塞，仍然可以处理新来的操作。此时，如果有写操作，第一处日志就是指正在使用的AOF日志，Redis会把这个操作写到它的缓冲区。
            - 同时往老的aof日志追加，而且也会往缓冲区写一份
            - 高并发时缓冲区可能快速增长，Redis后来通过Linux管道技术让aof期间就能同时进行回放
        - 第二处日志，就是指新的AOF重写日志。这个操作也会被写到重写日志的缓冲区。
            - 不复用老的aof日志的原因：避免父子线程的写冲突；避免aof失败导致的数据污染
        - 用新的AOF文件替代旧文件
            - 修改文件名，保证切换的原子性

aof重放

- 命令操作只能一条一条按顺序执行，这个“重放”的过程就会很慢

rdb

- 内存快照
    - aof重放很慢，rdb很快
    - 把某一时刻的状态以文件的形式写到磁盘上
    - 数据恢复时，可以直接把RDB文件读入内存，很快地完成恢复
- 全量快照
    - save
        - 会阻塞主线程

- bgsave
  - 借助操作系统提供的写时复制技术（Copy-On-Write, COW）
    - 主线程要修改一块数据，这块数据就会被复制一份，生成副本。bgsave子进程会把这个副本数据写入RDB文件，主线程仍然可以直接修改原来的数据。
      - 内核需要创建用于管理子进程的相关数据结构，这些数据结构在操作系统中通常叫作进程控制块（Process Control Block，简称为PCB）。内核要把主线程的PCB内容拷贝给子线程。这个创建和拷贝过程由内核执行，是会阻塞主线程的。而且，在拷贝过程中，子进程要拷贝父进程的页表，这个过程的耗时和Redis实例的内存大小有关。如果Redis实例内存大，页表就会大，fork执行时间就会长，这就会给主线程带来阻塞风险。
      - 主线程fork出bgsave子进程后，bgsave子进程实际是复制了主线程的页表。这些页表中，就保存了在执行bgsave命令时，主线程的所有数据块在内存中的物理地址。这样一来，bgsave子进程生成RDB时，就可以根据页表读取这些数据，再写入磁盘中。如果此时，主线程接收到了新写或修改操作，那么，主线程会使用写时复制机制。具体来说，写时复制就是指，主线程在有写操作时，才会把这个新写或修改后的数据写入到一个新的物理地址中，并修改自己的页表映射。
  - 频率
    - 频繁执行的问题
      - 磁盘写压力大
      - fork子线程会阻塞主线程

混合使用

- Redis 4.0中提出了一个混合使用AOF日志和内存快照的方式
- 内存快照以一定的频率执行，在两次快照之间，使用AOF日志记录这期间的所有命令操作

数据淘汰

一般都是两步操作

- 第一，根据一定的策略，筛选出对应用访问来说“不重要”的数据
- 第二，将这些数据从缓存中删除，为新来的数据腾出空间
  容量设置
- 八二原理
  - 20%的数据贡献了80%的访问
  - “八二原理”是对大量实际应用的数据访问情况做了统计后，得出的一个统计学意义上的数据量和访问量的比例。
- 重尾效应
  - 20%的数据可能贡献不了80%的访问，而剩余的80%数据反而贡献了更多的访问量，我们称之为重尾效应

容量规划

- 容量规划不能一概而论，是需要结合应用数据实际访问特征和成本开销来综合考虑的
- 一般来说，我会建议把缓存容量设置为总数据量的15%到30%，兼顾访问性能和内存空间开销

淘汰策略

- 不淘汰
  - noeviction
    - 一旦缓存被写满了，再有写请求来时，Redis不再提供服务，而是直接返回错误
    - 一般不把它用在Redis缓存中
- 设置过期时间的数据中淘汰
  - volatile-random
  - volatile-ttl
- 越早过期的越先被删除
  - volatile-lru
  - volatile-lfu
- 所有数据中淘汰
  - allkeys-lru
  - allkeys-random
  - allkeys-lfu

lru

- 一般方案
  - 把所有的数据组织成一个链表，链表的头和尾分别表示MRU端和LRU端
  - 认为刚刚被访问的数据，肯定还会被再次访问，所以就把它放在MRU端；长久不访问的数据，肯定就不会再被访问了，所以就让它逐渐后移到LRU端，在缓存满时，就优先删除它
- 优化方案
  - 一般方案的问题：全量数据用链表空间开销大
  - redis方案
    - 每个数据记录最近一次访问的时间戳
    - 第一次淘汰
      - 随机挑选n个数据
    - 第n次淘汰
      - 挑选数据进入前一次的集合
      - 能进入候选集合的数据的lru字段值必须小于候选集合中最小的lru值
    - 每次都淘汰lru字段值最小的元素
- 淘汰数据处理
  - 干净数据直接删除
  - 脏数据回写到数据库（业务要自己实现，在数据改动时就写回到数据库，redis并不会操作）

lfu

会从两个维度来筛选并淘汰数据：一是，数据访问的时效性（访问时间离当前时间的远近）；二是，数据的被访问次数

- 在lru基础上实现
  - RedisObject中设置lru字段，记录数据访问时间
  - 没有维护全局链表，通过随机采样，选取一定数量的数据放入候选集合，根据lru值进行筛选

- lfu实现
  - 把原来24bit大小的lru字段，又进一步拆分成了两部分
    - ldt值：lru字段的前16bit，表示数据的访问时间戳
    - counter值：lru字段的后8bit，表示数据的访问次数
  - 执行
    - 在候选集合中，根据lru字段后8bit选择访问次数最少的数据淘汰。访问次数相同时，再根据lru字段的前16bit值大小，选择访问时间最久远的数据进行淘汰。
  - 8bit计数限制
    - 在实现LFU策略时，Redis并没有采用数据每被访问一次，就给对应的counter值加1的计数规则，而是采用了一个更优化的计数规则
    - 每当数据被访问一次时，首先，用计数器当前的值乘以配置项lfu_log_factor再加1，再取其倒数，得到一个p值；然后，把这个p值和一个取值范围在（0，1）间的随机数r值比大小，只有p值大于r值时，计数器才加1
    - 这是一种非线性递增的计数器方法

触发时机

- 惰性删除
  - 惰性策略就是在客户端访问这个 key 的时候，redis 对 key 的过期时间进行检查，如果过期了就立即删除。
- 定时删除
  - redis 会将每个设置了过期时间的 key 放入到一个独立的字典中，以后会定时遍历这个字典来删除到期的 key。
  - 处理步骤
    - 从过期字典中随机 N 个 key
    - 删除这 N 个 key 中已经过期的 key；
    - 如果过期的 key 比率超过 1/4，那就重复步骤 1；
  - 为了保证过期扫描不会出现循环过度，导致线程卡死现象，算法还增加了扫描时间的上限，默认不会超过 25ms。
- BIO
  - 为了支持懒惰删除，redis作者将对象共享机制彻底抛弃
  - 主线程需要将删除任务传递给异步线程，它是通过一个普通的双向链表来传递的。因为链表需要支持多线程并发操作，所以它需要有锁来保护。

主从模式

通过主从模式，可以实现读写分离，有两个特点

- 主从库都可以读
- 只有主库可以写，同步到从库

主从同步

1. 全量同步

- 三个阶段
  - 建立连接，协商同步
    - 从库传输主库id，复制进度2个参数
  - 同步数据给从库
    - 内存快照生成的rdb文件

      > 问什么全量同步不适用aof文件？两个原因：1. aof文件比rdb文件大，网络传输成本高；2. rdb文件加载快
      >
      > 无盘复制：

  - 同步新写命令给从库
    - 主库会在内存中用专门的replication buffer，记录RDB文件生成后收到的所有写操作
      - replication buffer是主从库在进行全量复制时，主库上用于和从库连接的客户端的buffer
      - replication buffer不是共享的，而是每个从库都有一个对应的客户端

      > 从库较多时，可以使用主从级联模式分担全量复制时的主库压力，通过“主-从-从”模式将主库生成RDB和传输RDB的压力，以级联的方式分散到从库上

  - 一旦主从库完成了全量复制，它们之间就会一直维护一个网络连接，主库会通过这个连接将后面陆续收到的命令操作再同步给从库，这个过程也称为基于长连接的命令传播，可以避免频繁建立连接的开销

2. 增量同步
   - repl_backlog_buffer缓冲区
     - repl_backlog_buffer是为了支持从库增量复制，主库上用于持续保存写操作的一块专用buffer

     > 这个缓冲区是所有从库共享的

     - repl_backlog_buffer是一个环形缓冲区，主库会记录自己写到的位置，从库则会记录自己已经读到的位置
     - 如果从库的读取速度比较慢，就有可能导致从库还未读取的操作被主库新写的操作覆盖了，这会导致主从库间的数据不一致。

     > 发现不一致后就会触发全量同步，导致额外的开销

     - 怎么避免
       - 调整repl_backlog_size这个参数的大小
       - 考虑使用切片集群来分担单个主库的请求压力
3. 一些问题

- 主从数据不一致
  - 原因
    - 主从库间的命令复制是异步进行的
    - 主库执行完写命令就直接返回，没有等从库同步完成
  - 场景
    - 同步信息网络同步延迟

    > 方案：尽量保证主从库间的网络连接状况良好

        - 从库执行其他命令没有及时处理同步信息

    > 方案：添加监控，不同步过多就自动摘除

- 读取过期数据
    - redis删除策略：惰性删除+过期删除

    > 思考：从库是否可以主动删除数据？ 不可以，可以考虑两个场景：1. 对于一个马上过期的key，master续活了，slave删掉的话就导致key丢失，主从不一致；2. slave机器时钟跳跃，提前删除了key，也导致不一致

    - 场景
        - 读从库数据过期仍返回（3.2版本之前会返回过期数据，后续版本不会）
    - 从库设置过期时间比主库晚
        - EXPIRE和PEXPIRE可能存在这个问题
        - EXPIREAT和PEXPIREAT 可以设置过期时间戳
- 配置项
    - protected-mode
        - 限定哨兵实例能否被其他服务器访问，yes时，哨兵实例只能在部署的服务器本地进行访问，会导致无法和其他服务器上的哨兵通信
    - cluster-node-timeout
        - Redis Cluster中实例响应心跳消息的超时时间
    - 主从库设置的 maxmemory 不同
        - 从库内存小，可能会先淘汰数据导致不一致
    - 主从同步的 client-output-buffer-limit 过小
        - 数据量很大情况下，buff溢出，同步失败并重试，产生复制风暴
- 脑裂

  脑裂，就是指在主从集群中，同时有两个主节点，它们都能接收写请求。而脑裂最直接的影响，就是客户端不知道应该往哪个主节点写入数据，结果就是不同的客户端会往不同的主节点上写入数据。而且，严重的话，脑裂会进一步导致数据丢失。
    - 分析
        - 第一步：确认是不是数据同步出现了问题(最常见的原因就是主库的数据还没有同步到从库，结果主库发生了故障，等从库升级为主库后，未同步的数据就丢失了,可以通过比对主从库上的复制进度差值来进行判断)
    - 第二步：排查客户端的操作日志，确认是否是脑裂现象
        - 在主从切换后的一段时间内，有一个客户端仍然在和原主库通信，并没有和升级的新主库进行交互
    - 第三步：发现是原主库假故障导致的脑裂
        - 在切换过程中，既然客户端仍然和原主库通信，这就表明，原主库并没有真的发生故障
        - 在被判断下线之后，原主库又重新开始处理请求了，而此时，哨兵还没有完成主从切换，客户端仍然可以和原主库通信，客户端发送的写操作就会在原主库上写入数据了
    - 为什么脑裂会导致数据丢失?
        - 主从切换后，从库一旦升级为新主库，哨兵就会让原主库执行slave of命令，和新主库重新进行全量同步。而在全量同步执行的最后阶段，原主库需要清空本地的数据，加载新主库发送的RDB文件
        - 在主从切换的过程中，如果原主库只是“假故障”，它会触发哨兵启动主从切换，一旦等它从假故障

中恢复后，又开始处理请求，这样一来，就会和新主库同时存在，形成脑裂。等到哨兵让原主库和新主库做全量同步后，原主库在切换期间保存的数据就丢失了。

- 方案
  - Redis已经提供了两个配置项来限制主库的请求处理，分别是min-slaves-to-write和min-slaves-max-lag
  - 这两个配置项组合后的要求是，主库连接的从库中至少有N个从库，和主库进行数据复制时的ACK消息延迟不能超过T秒，否则，主库就不会再接收客户端的请求了，这样的话，即使原主库是假故障，它在假故障期间也无法响应哨兵心跳，也不能和从库进行同步，自然也就无法和从库进行ACK确认了。
- 其他
  - 当脑裂发生时，两个配置还是无法严格保证数据不丢失，它只能是尽量减少数据的丢失
  - 脑裂产生问题的本质原因是，Redis 主从集群内部没有通过共识算法，来维护多个节点数据的强一致性。它不像 Zookeeper 那样，每次写请求必须大多数节点写成功后才认为成功。

故障转移-哨兵

哨兵机制是实现主从库自动切换的关键。

- 需要考虑三个问题
  1. 主库真的挂了吗?
  2. 该选择哪个从库作为主库?
  3. 怎么把新主库的相关信息通知给从库和客户端呢?
- 哨兵的三个任务
  1. 监控
     - 哨兵进程会使用PING命令检测它自己和主、从库的网络连接情况，用来判断实例的状态。
     - 为了避免误判，通常会采用多实例组成的集群模式进行部署，这也被称为哨兵集群
     - 在判断主库是否下线时，不能由一个哨兵说了算，只有大多数的哨兵实例，都判断主库已经“主观下线”了，主库才会被标记为“客观下线”
  2. 选择主库
     - 在多个从库中，先按照一定的筛选条件，把不符合条件的从库去掉。然后，我们再按照一定的规则，给剩下的从库逐个打分，将得分最高的从库选为新主库
     - 在选主时，除了要检查从库的当前在线状态，还要判断它之前的网络连接状态
     - 分别按照三个规则依次进行三轮打分，这三个规则分别是**从库优先级、从库复制进度以及从库ID号**。只要在某一轮中，有从库得分最高，那么它就是主库了，选主过程到此结束。如果没有出现得分最高的从库，那么就继续进行下一轮。
  3. 通知
- 主从切换
  - 谁来切换
    - 哨兵集群判断出主库“主观下线”后，会选出一个“哨兵领导者”，之后整个过程由它来完成主从切换。

> - 任何一个实例只要自身判断主库“主观下线”后，就会给其他实例发送is-master-down-by-addr命令。接着，其他实例会根据自己和主库的连接情况，做出Y或N的响应，Y相当于赞成票，N相当于反对票。
> - 此时，这个哨兵就可以再给其他哨兵发送命令，表明希望由自己来执行主从切换，并让所有

> 其他哨兵进行投票。这个投票过程称为“Leader选举”。因为最终执行主从切换的哨兵称为Leader，投票过程就是确定Leader（这个选举过程是分布式系统中的问题，就是我们经常听说的共识算法）。
>
> - 在投票过程中，任何一个想成为Leader的哨兵，要满足两个条件：第一，拿到半数以上的赞成票；第二，拿到的票数同时还需要大于等于哨兵配置文件中的quorum值。
> - 如果一次选举没有选出leader，哨兵集群会等待一段时间（也就是哨兵故障转移超时时间的2倍），再重新选举。

- 一些相关的问题
  - 哨兵挂了会影响监控和选择主库吗?

    > 这也是分布式系统中的问题：即存在故障节点时，集群是否能正常提供服务。结论是 **存在故障节点时，只要集群中大多数节点状态正常，集群依旧可以对外提供服务**（可以了解下分布式系统容错问题：“拜占庭将军”问题）

  - 哨兵不是越多越好?
    - 在判定“主观下线”和选举“哨兵领导者”时，都需要和其他节点进行通信，交换信息，哨兵实例越多，通信的次数也就越多，而且部署多个哨兵时，会分布在不同机器上，节点越多带来的机器故障风险也会越大
  - 切换过程中写请求失败怎么办?
    - 客户端可以缓存一下写请求
  - 哨兵实例之间是怎么相互发现的?
    - 这个要归功于Redis提供的pub/sub机制

    > 从本质上说，哨兵就是一个运行在特定模式下的Redis实例，只不过它并不服务请求操作，只是完成监控、选主和通知的任务。所以，每个哨兵实例也提供pub/sub机制，客户端可以从哨兵订阅消息。哨兵提供的消息订阅频道有很多，不同频道包含了主从库切换过程中的不同关键事件。

    - 当多个哨兵实例都在主库上做了发布和订阅操作后，它们之间就能知道彼此的IP地址和端口。
  - 哨兵除了彼此之间建立起连接形成集群外，还需要和从库建立连接（哨兵是通过向主库发送INFO命令来获取从库的IP地址和端口的）。

3. 可扩展

Redis使用集群方案就是为了解决单个节点数据量大、写入量大产生的性能瓶颈的问题。

- 横向扩容
  - 横向增加redis实例个数
- 纵向扩容
  - 升级单个redis的资源配置，比如内存，硬盘，cpu
  - 优势
    - 简单直接
  - 劣势
    - 受到硬件和成本的限制
    - 在使用RDB进行持久化时，Redis会fork子进程来完成，fork操作的用时和Redis的数据量是正相关的，而fork在执行时会阻塞主线程。数据量越大，fork操作造成的主线程阻塞的时间越长。

- 横向扩容两个问题
  - 数据分到哪个实例
  - 客户端怎么知道数据在哪个实例

Redis Cluster

Redis Cluster是去中心化的

- 数据路由方案--**哈希槽**

  > 用哈希槽(Hash Slot)来处理数据和实例之间的映射关系

  - 一个集群共有16384个哈希槽，这些哈希槽类似于数据分区，每个键值对都会根据它的key，被映射到一个哈希槽中
  - Redis会自动把这些槽平均分布在集群实例上(也可以手动分配，不过在手动分配哈希槽时，需要把16384个槽都分配完，否则Redis集群无法正常工作)
- 客户端定位对应实例
  - Redis实例会把自己的哈希槽信息发给和它相连接的其它实例，来完成哈希槽分配信息的扩散
  - 客户端收到哈希槽信息后，会把哈希槽信息缓存在本地

  > 使用哈希槽的优势，可以把数据和节点解耦，哈希槽到实例的映射表很小；如果维护key到实例的映射表的方式的话，会有这几个缺点：1.大量key占用空间大；2.节点同步映射表成本高；3.数据迁移时修改成本大。

- 哈希槽改变
  - 场景
    - 实例新增，删除
    - 负载均衡，重新分配哈希槽
  - 重定向机制
    - 客户端给一个实例发送数据读写操作时，这个实例上并没有相应的数据，客户端要再给一个新实例发送操作命令
    - 如果被请求的实例上没有这个键值对映射的哈希槽，那么，这个实例就会给客户端返回MOVED响应结果，结果中就包含了新实例的访问地址
    - 数据迁移部分完成的情况下，客户端就会收到一条ASK报错信息
    - ASK命令并不会更新客户端缓存的哈希槽分配信息
- 数据迁移
  - 当集群节点不足以支撑业务需求时，就需要扩容节点，扩容就意味着节点之间的数据需要做迁移，而迁移过程中是否会影响到业务，这也是判定一个集群方案是否成熟的标准
  - Twemproxy不支持在线扩容，它只解决了请求路由的问题，扩容时需要停机做数据重新分配。
  - 而Redis Cluster和Codis都做到了在线扩容（不影响业务或对业务的影响非常小），重点就是在数据迁移过程中，客户端对于正在迁移的key进行操作时，集群如何处理？还要保证响应正确的结果

  > 迁移过程中，服务端针对正在迁移的key，需要让客户端或Proxy去新节点访问（重定向）

问题

1. 数据倾斜
    - 分类
        - 数据访问倾斜
        - 数据量倾斜
    - 数据量倾斜原因
        - bigkey，方案：大key拆分
    - slot分配不均
    - Hash Tag导致倾斜

    > Hash Tag是指加在键值对key中的一对花括号{}。这对括号会把key的一部分括起来，客户端在计算key的CRC16值时，只对Hash Tag花括号中的key内容进行计算。如果没用Hash Tag的话，客户端计算整个key的CRC16的值。好处是，如果不同key的Hash Tag内容都是一样的，那么，这些key对应的数据会被映射到同一个Slot中，同时会被分配到同一个实例上。主要是用在Redis Cluster和Codis中，支持事务操作和范围查询。使用hash tag时，需要在范围查询、事务执行的需求和数据倾斜带来的访问压力之间，进行取舍。

    - 数据访问倾斜
        - 热点数据
            - 热点数据多副本

            > 只能针对只读的热点数据，否则为了保证数据一致性，有额外的开销

2. 通信开销
   Redis官方给出了Redis Cluster的规模上限，就是一个集群运行1000个实例

   > 一个关键因素就是，实例间的通信开销会随着实例规模增加而增大，在集群超过一定规模时，集群吞吐量反而会下降

实例通信--Gossip协议
为了让集群中的每个实例都知道其它所有实例的状态信息，实例之间会按照一定的规则进行通信。这个规则就是Gossip协议
协议有两个操作：

- 一是，每个实例之间会按照一定的频率，从集群中随机挑选一些实例，把PING消息发送给挑选出来的实例，用来检测这些实例是否在线，并交换彼此的状态信息。
- 二是，一个实例在接收到PING消息后，会给发送PING消息的实例，发送一个PONG消息。PONG消息包含的内容和PING消息一样。

> Redis Cluster 把所有功能都集成在了 Redis 实例上，好处是，部署和使用非常简单；缺点是每个实例负责的工作比较重。与之相比，codis把这些功能拆分成多个组件，缺点是组件比较多，部署和维护比较复杂。

通信开销
通信开销受到通信消息大小和通信频率这两方面的影响

- 通信消息大小
    - clusterMsgDataGossip结构体(104字节)

    - 默认还会传递集群十分之一实例的状态信息
    - 还带有一个长度为 16384 bit 的 Bitmap表示本实例持有的slot
- 通信频率
    - 每个实例每1秒发送一条PING消息(从本地的实例列表中随机选出5个实例，再从这5个实例中找出一个最久没有通信的实例，把PING消息发送给该实例。这是实例周期性发送PING消息的基本做法)
    - 每个实例每100毫秒会做一次检测，给PONG消息接收超过cluster-node-timeout/2的节点发送PING消息

怎么降低通信开销

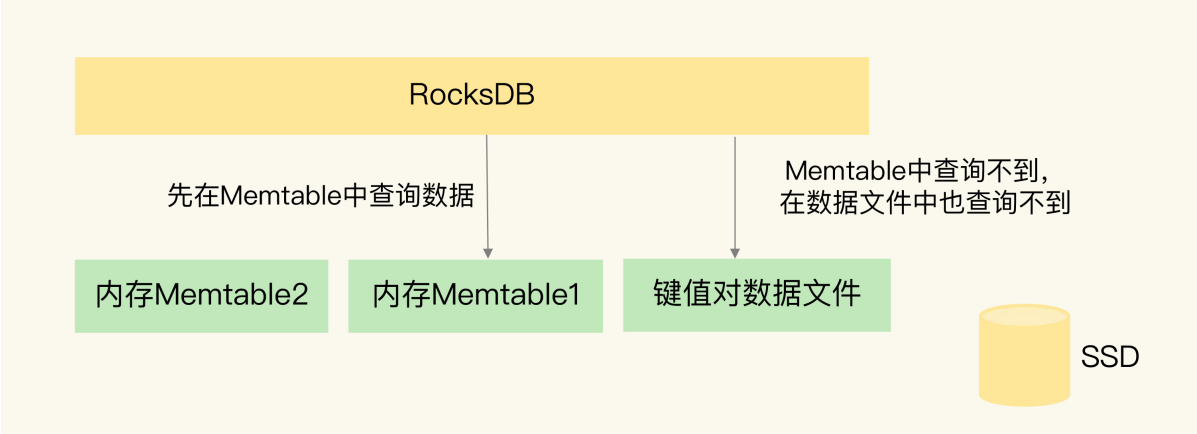
> 配置项cluster-node-timeout定义了集群实例被判断为故障的心跳超时时间，默认是15秒(适当调大，避免pong消息超时)

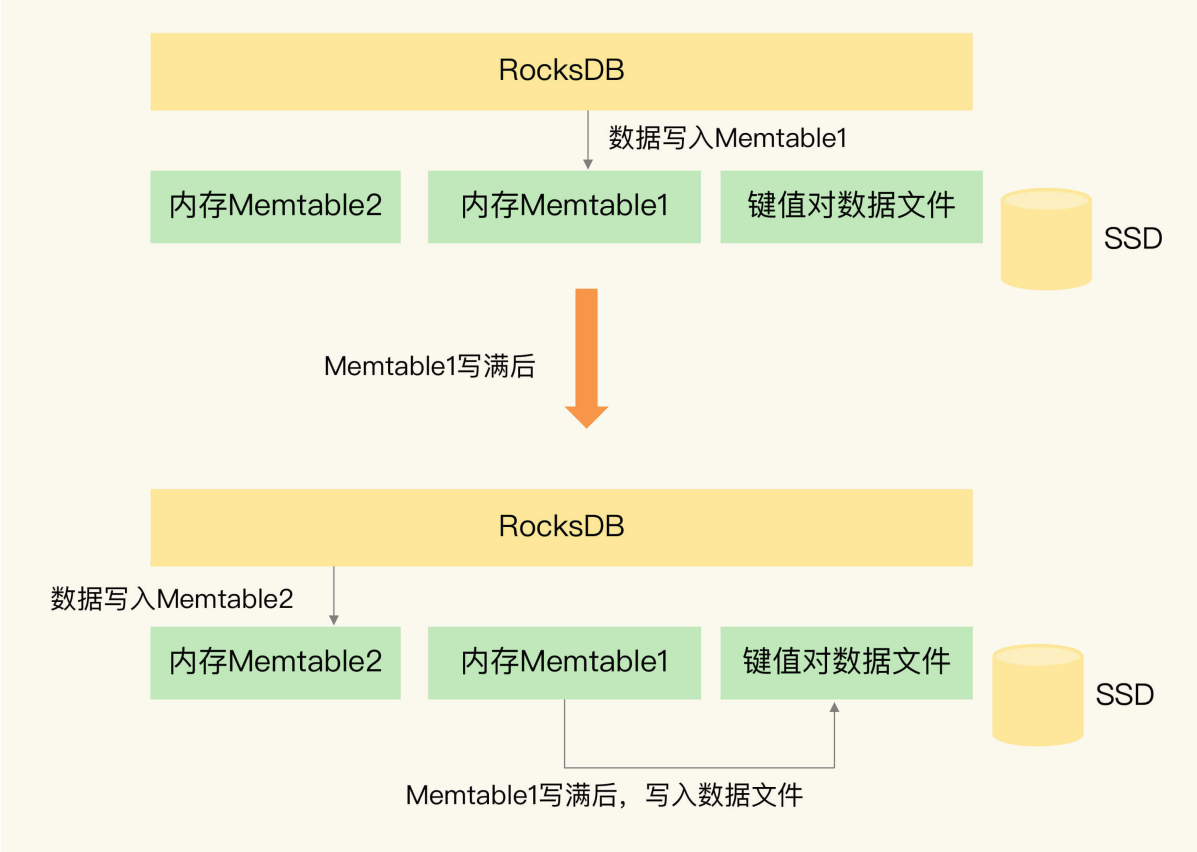
pika

Pika在刚开始设计的时候，就有两个目标：一是，单实例可以保存大容量数据，同时避免了实例恢复和主从同步时的潜在问题；二是，和Redis数据类型保持兼容，可以支持使用Redis的应用平滑地迁移到Pika上

- 大实例redis的问题
    - 内存快照RDB生成和恢复效率低
    - 主从节点全量同步时长增加
    - 缓冲区易溢出（一旦缓冲区溢出了，主从节点间就会又开始全量同步）
- pika方案
    - 读文件就可恢复，不需要rdb
    - binlog同步增量，使用的文件，不需要缓冲区
- 整体架构
    - 

    - 网络框架

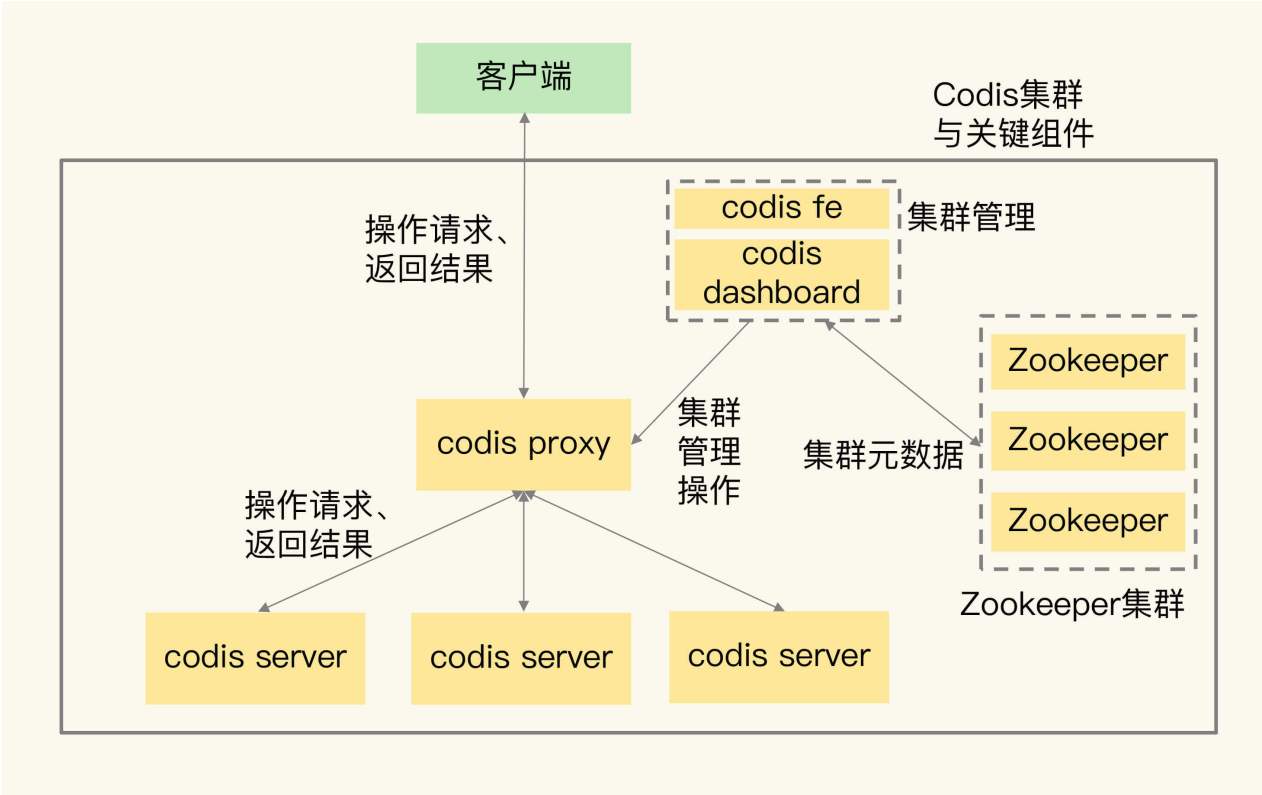
        - 负责底层网络请求的接收和发送
    - Pika线程模块
        - 采用了多线程模型来具体处理客户端请求，包括一个请求分发线程（DispatchThread）、一组工作线程（WorkerThread）以及一个线程池（ThreadPool）
        - 请求分发线程专门监听网络端口，一旦接收到客户端的连接请求后，就和客户端建立连接，并把连接交由工作线程处理。工作线程负责接收客户端连接上发送的具体命令请求，并把命令请求封装成Task，再交给线程池中的线程，由这些线程进行实际的数据存取处理
    - Nemo模块
        - 实现了Pika和Redis的数据类型兼
        - 方案
            - RocksDB只提供了单值的键值对类型
            - list：List集合的key当成key用，List集合的元素值，则被嵌入到单值键值对的值当中
            - set：key和元素member值，都被嵌入到了Pika单值键值对的键当中
            - hash
            - zset
- RocksDB
    - 业界广泛应用的持久化键值数据库，提供了基于SSD保存数据的功能
    - 大概原理
        - 



- 优势和不足
  - 优势
    - 实例重启快
    - 主从库重新执行全量同步的风险低
  - 劣势
    - 用了ssd，数据访问性能差
    - 使用了binglog，会影响写入性能
  - 优化方案
    - 使用多个线程并发读写
    - 使用高配的ssd

codis

- 架构



    - codis server：这是进行了二次开发的Redis实例，支持数据迁移操作，负责处理数据读写请求
    - codis proxy：接收客户端请求，并把请求转发给codis server
    - Zookeeper集群：保存集群元数据，例如数据位置信息和codis proxy信息
    - codis dashboard和codis fe：共同组成了集群管理工具。其中，codis dashboard负责执行集群管理工作，增删server、proxy、数据迁移。codis fe提供dashboard的Web操作界面
- 关键点
    - 数据路由
        - Codis集群一共有1024个Slot，把这些Slot分配给codis server(手动或自动)，每个server上包含一部分Slot。
        - 客户端要读写数据时，使用CRC32计算哈希值，对1024取模，得到对应Slot编号，找到server
        - 数据路由表，在codis dashboard上分配好路由表后，dashboard会把路由表发送给codis proxy，并保存在Zookeeper中。
        - 在Redis Cluster中，数据路由表是通过每个实例相互间的通信传递的，最后会在每个实例上保存一份。当数据路由信息发生变化时，就需要在所有实例间通过网络消息进行传递。
- 扩容+数据迁移
    - 扩容
        - 增加server
        - 增加proxy
            - 启动proxy，通过dashboard把proxy加入集群
            - 写入到Zookeeper，客户端可以从Zookeeper上读取proxy访问列表
    - 数据迁移
        - 步骤
            - 从要迁移的Slot中随机选择一个数据，发送给目的server
            - 目的server收到数据后，给源server返回确认消息。源server将刚才迁移的数据删除
            - 不断重复这个迁移过程，直到全部迁移完成
        - 同步迁移

        - 在数据从源server发送给目的server的过程中，源server是阻塞的，无法处理新的请求操作
    - 异步迁移
        - 两个特点
            - 当源server把数据发送给目的server后，就可以处理其他请求操作
            - 过程中，迁移的数据会被设置为只读
        - 对于bigkey，异步迁移采用了拆分指令的方式进行迁移
            - 对bigkey中每个元素，用一条指令进行迁移，而不是把整个bigkey进行序列化后再整体传输
            - 为了避免迁移过程中出问题，在目标server上，给bigkey的元素设置一个临时过期时间
- 兼容性
    - codis proxy是和单实例客户端兼容
- 可靠性
    - codis server
        - 使用主从集群来保证codis server的可靠性
        - 给每个server配置从库，并使用哨兵机制进行监控
    - Zookeeper
        - Zookeeper集群使用多个实例来保存数据，只要有超过半数的Zookeeper实例可以正常工作，Zookeeper集群就可以提供服务
    - codis proxy
        - 利用Zookeeper的高可靠性保证来确保codis proxy的可靠性
- 其他问题
    - 不支持redis的事务（命令操作的key可能分布在不通的节点上）
    - 不支持redis的一些命令，比如rename命令
    - 因为增加了proxy层，性能要差一些
    - 引入了zookeeper解决集群分布式问题，增加了运维的复杂度

三、底层实现原理

string

- 简单动态字符串sds
    - 底层组成
        - buf 字节数组
            - Redis会自动在数组最后加一个“\0”，这就会额外占用1个字节的开销，是为了兼容c的字符串处理函数
        - len 数组已用长度
        - alloc 实际分配长度
        - flags 标记位

    - 扩容策略
        - 长度小于 1M 之前，加倍扩容
        - 当长度超过 1M 之后，每次扩容只会多分配 1M
- redisObject
    - 8字节的元数据和一个8字节指针
    - 3种编码方式
        - int
            - Long类型整数时，RedisObject中的指针就直接赋值为整数数据
        - embstr
            - 对字符串数据，当长处<=44字节时，RedisObject中的元数据、指针和SDS是一块连续的内存区域，可以避免内存碎片
            - 44=64-16(redis object)-3(sds的3个字段)-1(sds最后一个'\0')
        - raw
            - 字符串，长度>44字节，Redis就会给SDS分配独立的空间，并用指针指向SDS结构
- Redis会使用一个全局哈希表保存所有键值对
    - 哈希表的每一项是一个dictEntry的结构体，用来指向一个键值对。dictEntry结构中有三个8字节的指针，分别指向key、value以及下一个dictEntry，三个指针共24字节
    - Redis使用的内存分配库jemalloc。在分配内存时，会根据我们申请的字节数N，找一个比N大，但是最接近N的2的幂次数作为分配的空间，这样可以减少频繁分配的次数。
- 用string保存大量的小值kv的时候，额外的空间浪费需要注意

ziplist

- 表头
    - zlbytes、zltail和zllen，分别表示列表长度、列表尾的偏移量，以及列表中的entry个数
- 表尾
    - zlend，表示列表结束
- 是用一系列连续的entry保存数据，节省了指针的空间
- entry结构
    - prev_len，表示前一个entry的长度
        - 压缩列表中zlend的取值默认是255，因此，就默认用255表示整个压缩列表的结束，其他表示长度的地方就不能再用255这个值了
        - 当上一个entry长度小于254字节时，prev_len取值为1字节，否则，就取值为5字节
    - len：表示自身长度，4字节；
    - encoding：表示编码方式，1字节；Redis 为了节约存储空间，对 encoding 字段进行了相当复杂的设计。
    - content：保存实际数据
- 级联更新
    - 每个 entry 都会有一个 `prevlen` 字段存储前一个 entry 的长度。如果内容小于 254 字节，`prevlen` 用 1 字节存储，否则就是 5 字节。这意味着如果某个 entry 经过了修改操作从 253 字节变成了 254 字节，那么它的下一个 entry 的 `prevlen` 字段就要更新，从 1 个字节扩展到 5 个字节；如果这个 entry 的长

度本来也是 253 字节，那么后面 entry 的 `prevlen` 字段还得继续更新

- Redis基于压缩列表实现了List、Hash和Sorted Set这样的集合类型，这样做的最大好处就是节省了dictEntry的开销
- 怎么用hash存储大量的单值kv结构
  - 对k进行hash，高位指定hash表名，低位作为key
- Redis Hash类型的两种底层实现结构
  - 压缩列表和哈希表
  - Hash类型设置了用压缩列表保存数据时的两个阈值，一旦超过阈值，就会用哈希表来保存数据
    - 用压缩列表保存时哈希集合中的最大元素个数
    - 用压缩列表保存时哈希集合中单个元素的最大长度
- 为了能充分使用压缩列表的精简内存布局，我们一般要控制保存在Hash集合中的元素个数

hashTable

```
struct RedisDb {
    dict* dict; // all keys  key=>value
    dict* expires; // all expired keys key=>long(timestamp)
    ...
}

struct zset {
    dict *dict; // all values  value=>score
    zskiplist *zsl;
}

struct dict {
    ...
    dictht ht[2];
}

struct dictEntry {
    void* key;
    void* val;
    dictEntry* next; // 链接下一个 entry
}
struct dictht {
    dictEntry** table; // 二维
    long size; // 第一维数组的长度
    long used; // hash 表中的元素个数
    ...
}
```

- 扩容
  - 正常情况下，当 hash 表中元素的个数等于第一维数组的长度时，就会开始扩容，扩容的新数组是原数组大小的 2 倍。如果 Redis 正在做 bgsave，为了减少内存页的过多分离 (Copy On Write)，Redis 尽量不去扩容 (`dict_can_resize`)。

    - 如果 hash 表已经非常满了，元素的个数已经达到了第一维数组长度的 5 倍（`dict_force_resize_ratio`），说明 hash 表已经过于拥挤了，这个时候就会强制扩容
- 缩容
    - 缩容的条件是元素个数低于数组长度的 10%。缩容不会考虑 Redis 是否正在做 bgsave
- 渐进式rehash
    - 搬迁操作埋伏在当前字典的后续指令中(来自客户端的hset/hdel指令等)
    - Redis还会在定时任务中对字典进行主动搬迁

quicklist

- quicklist 是 ziplist 和 linkedlist 的混合体，它将 linkedlist 按段切分，每一段使用 ziplist 来紧凑存储，多个 ziplist 之间使用双向指针串接起来
- quicklist 内部默认单个 ziplist 长度为 8k 字节，超出了这个字节数，就会新起一个 ziplist。ziplist 的长度由配置参数`list-max-ziplist-size`决定

skiplist

```
struct zslnode {
  string value;
  double score;
  zslnode*[] forwards;  // 多层连接指针
  zslnode* backward;  // 回溯指针
}

struct zsl {
  zslnode* header; // 跳跃列表头指针
  int maxLevel; // 跳跃列表当前的最高层
  map<string, zslnode*> ht; // hash 结构的所有键值对
}
```

基数树(字典树)

- Rax 被用在 Redis Stream 结构里面用于存储消息队列，在 Stream 里面消息 ID 的前缀是时间戳 + 序号，这样的消息可以理解为时间序列消息。使用 Rax 结构进行存储就可以快速地根据消息 ID 定位到具体的消息，然后继续遍历指定消息之后的所有消息
- Rax 被用在 Redis Cluster 中用来记录槽位和key的对应关系

一些命令

scan

- Redis 中所有的 key 都存储在一个很大的字典中，这个字典的结构是一维数组 + 二维链表结构，第一维数组的大小总是 2^n(n>=0)，扩容一次数组大小空间加倍，也就是 n++
- scan 指令返回的游标就是第一维数组的位置索引
- 之所以scan返回的结果可能多可能少，是因为不是所有的槽位上都会挂接链表，有些槽位可能是空的，还有些槽位上挂接的链表上的元素可能会有多个
- scan 的遍历顺序非常特别。它不是从第一维数组的第 0 位一直遍历到末尾，而是采用了高位进位加法来遍历

(高位进位法从左边加，进位往右边移动，同普通加法正好相反)。之所以使用这样特殊的方式进行遍历，是考虑到字典的扩容和缩容时避免槽位的遍历重复和遗漏。