Redis

Redis基本概念

为啥使用redis?

在日常的 Web 应用对数据库的访问中，**读操作的次数远超写操作**，比例大概在 **1:9** 到 **3:7**，所以需要读的可能性是比写的可能大得多的。当我们使用 SQL 语句去数据库进行读写操作时，数据库就会 **去磁盘把对应的数据索引取回来**，这是一个相对较慢的过程。
因为传统的关系型数据库如Mysql已经不能适用所有的场景了，比如热点数据的存量扣减，排班，考勤组的访问流量高峰等等，都很容易把数据库打崩，所以引入了缓存中间件，目前市面上比较常用的缓存中间件有Redis 和 Memcached 不过中和考虑了他们的优缺点，最后选择了Redis。阿里内部本身的缓存中间件是tair，本质上是redis。
如果我们把数据放在 Redis 中，也就是直接放在内存之中，让服务端直接去读取内存中的数据，那么这样 **速度** 明显就会快上不少 *(高性能)*，并且会 **极大减小数据库的压力***(特别是在高并发情况下)*。

Redis和Memcached的区别：

1与Memcached仅支持简单的key-value结构的数据记录不同，**Redis支持的数据类型要丰富得多**。Memcached基本只支持简单的key-value存储，不**支持枚举，不支持持久化和复制**等功能。
2 Redis只使用单核，而Memcached可以使用多核，所以平均每一个核上**Redis在存储小数据时比Memcached性能更高**。而在**100k**以上的数据中，Memcached性能要高于Redis。排班的信息基本上都不是很大，所以Redis的性能可以有更好的发挥。
3 现在的话，一般缓存中间件都需要集群管理，单机器一般都不太够用，作为基于内存的存储系统来说，机器物理内存的大小就是系统能够容纳的最大数据量。如果需要处理的数据量超过了单台机器的物理内存大小，就需要构建分布式集群来扩展存储能力。**Redis支持服务端进行集群管理，而MemCached不支持**。

redis简单介绍

Redis (**Re**mote **Di**ctionary **S**erver) 是一个使用 **C 语言** 编写的，开源的 *(BSD许可)* 高性能 **非关系型***(NoSQL)* 的 **键值对数据库**。
Redis 可以存储 **键** 和 **不同类型数据结构值** 之间的映射关系。键的类型只能是字符串，而值除了支持最 **基础的五种数据类型** 外，还支持一些 **高级数据类型**：bitMap，布隆过滤器，Geo，pub/sub
与传统数据库不同的是**Redis**的数据是**存在内存**中的，所以**读写速度**非常**快**，因此 Redis 被广泛应用于**缓存**方向，每秒可以处理超过10万次读写操作，是已知性能最快的 Key-Value 数据库。另外，Redis 也经常用来做**分布式锁**。
除此之外，Redis 支持事务 、持久化、LUA脚本、LRU驱动事件、多种集群方案。

缺点：
数据库 **容量受到物理内存的限制**，不能用作海量数据的高性能读写
Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂
Redis **不具备自动容错和恢复功能**，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的 IP 才能恢复。

为什么Redis单线程也这么快

- 因此Redis完全基于内存操作，CPU不是redis的瓶颈，redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会称为瓶颈，那么就顺理成章地使用单线程地方案了：

1. Redis完全基于内存，绝大部分请求是纯粹的内存操作，非常迅速，数据存在内存中，类似于HashMap, HashMap的优势就是查找和操作的时间复杂度是O(1)；
2. 数据结构简单，对数据的操作也简单；

3. 采用单线程，避免了不必要的上下文切换和竞争条件，不存在多线程导致的CPU切换，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有死锁问题导致的性能消耗；
4. 使用多路复用IO模型，非阻塞IO；
5. 使用的底层模型不同，它们之间底层实现方法以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM机制，因为一般的系统调用系统函数的时候，会浪费一定的事件去移动和请求。

什么是bigKey，有什么影响?

Redis 在写入数据时，需要为新的数据分配内存，相对应的，当从 Redis 中删除数据时，它会释放对应的内存空间。
如果一个 key 写入的 value 非常大，那么 Redis 在**分配内存时就会比较耗时**。同样的，当删除这个 key 时，**释放内存也会比较耗时**，这种类型的 key 我们一般称之为 bigkey。

Redis 提供了扫描 bigkey 的命令，执行以下命令就可以扫描出，一个实例中 bigkey 的分布情况，输出结果是以类型维度展示的：redis-cli -h 127.0.0.1 -p 6379 --bigkeys -i 0.01

Redis变慢了，怎么排查，会是哪些问题。

1. 使用的命令复杂度过高，或者包含的数据过多， 例如SORT, UNION这种
2. bigkey
3. 集中过期情况
    a. 同样是用lazy-free方式，然后的话，是进行random随机事件分配
4. 内存达到临界上线
    a. 虽然有淘汰机制，但是淘汰机制也是在主线程中进行，lru本身也挺费事额。
    b. 开启lazy-free，或者是将LRU更换为random这种更快的淘汰机制
5. RDB的fork耗时严重
    a. fork创建子线程会触发系统调用，拷贝内存页表时候会阻塞，影响redis运行
    b. 调整参数把
6. 开启AOF
    a. 主要是再进行aof的时候，write和fsync系统调用会发生阻塞。
    b. 升级和保障磁盘空间和速度，开启no-appendfsync-on-write
7. 再进行碎片整理
    a. 因为碎片整理是在主线程上的，所以会一定程度影响速度
    b. 关闭碎片整理或者合理配置参数。

Redis为什么会比mysql快 从几个角度分析

Redis采用的是基于内存的采用的是单进程单线程模型的 KV 数据库，由C语言编写，官方提供的数据是可以达到100000+的QPS（每秒内查询次数）。

1. 完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。它的数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)；
2. 数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的；
3. 采用单线程，**避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗**
4. 使用多路I/O复用模型，非阻塞IO；

使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

我们现在服务器都是多核的，Redis是单线程单核的，那不是很浪费?

是的他是单线程的，但是，我们可以通过在单机开多个**Redis实例**嘛。

单机会有瓶颈，那你们是怎么解决这个瓶颈的?

我们用到了集群的部署方式也就是Redis cluster，并且是主从同步读写分离，类似Mysql的主从同步，Redis cluster 支撑 N 个 Redis master node，每个master node都可以挂载多个 slave node。
这样整个 Redis 就可以横向扩容了。如果你要支撑更大数据量的缓存，那就横向扩容更多的 master 节点，每个 master 节点就能存放更多的数据了。

redis 的线程模型了解么?

Redis 内部使用**文件事件处理器 file event handler**，这个文件事件处理器是单线程的，所以 Redis 才叫做单线程的模型。它采用 **IO 多路复用机制同时监听多个 Socket**，根据 Socket 上的事件来选择对应的事件处理器进行处理。
文件事件处理器的结构包含 4 个部分：

1. 多个 Socket
2. IO 多路复用程序
3. 文件事件分派器
4. 事件处理器（连接应答处理器、命令请求处理器、命令回复处理器）

多个 Socket 可能会并发产生不同的操作，每个操作对应不同的文件事件，但是 IO 多路复用程序会监听多个 Socket，**会将 Socket 产生的事件放入队列中排队，事件分派器每次从队列中取出一个事件，把该事件交给对应的事件处理器进行处理**

Redis里有多线程的内容么?

我们所说的Redis单线程，指的是"其网络IO和键值对读写是由一个线程完成的"，也就是说，**Redis中只有网络请求模块和数据操作模块是单线程的。而其他的如持久化存储模块、集群支撑模块等是多线程的。**

首先了解下多线程适用场景

一个计算机程序在执行的过程中，主要需要进行两种操作分别是读写操作和计算操作。
其中读写操作主要是涉及到的就是I/O操作，其中包括网络I/O和磁盘I/O。计算操作主要涉及到CPU。
而多线程的目的，就是通过并发的方式来提升I/O的利用率和CPU的利用率。

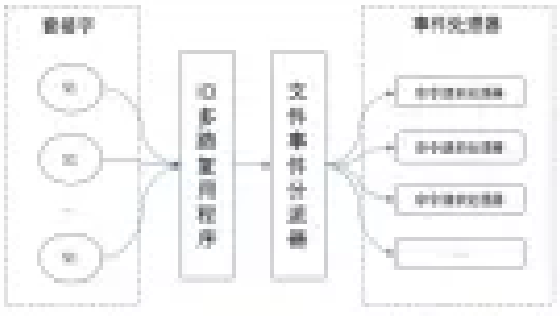
Redis不需要提升CPU利用率，因为**Redis的操作基本都是基于内存的，CPU资源根本就不是Redis的性能瓶颈。**

如果是用多线程来提高IO的利用率的话，会带来一个线程安全的问题。保障线程安全以及上下文切换其实是一个开销不小的工作。Redis本身是通过IO复用的方式来提高我们的IO利用率

IO复用

Linux多路复用技术，就是多个进程的IO可以注册到同一个管道上，这个管道会统一和内核进行交互。当管道中的某一个请求需要的数据准备好之后，进程再把对应的数据拷贝到用户空间中。IO多路复用在Linux下包括了三种，select、poll、epoll

在Redis 中，每当一个套接字准备好执行连接应答、写入、读取、关闭等操作时，就会产生一个文件事件。因为一个服务器通常会连接多个套接字，所以多个文件事件有可能会并发地出现。



Redis6.0以后对网络IO这块也还是采用了多线程的，但是的话，适用的范围**只包含了网络请求处理，不包含读写**
限制Redis的性能的主要瓶颈出现在网络IO的处理上，虽然之前采用了多路复用技术。但是我们前面也提到过，**多路复用的IO模型本质上仍然是同步阻塞型IO模型。**

在多路复用的IO模型中，在处理网络请求时，调用 select （其他函数同理）的过程是阻塞的，也就是说这个过程会阻塞线程，如果并发量很高，此处可能会成为瓶颈。

如果能采用多线程，使得网络处理的请求并发进行，就可以大大的提升性能。多线程除了可以减少由于网络 I/O 等待造成的影响，还可以充分利用 CPU 的多核优势。

Redis 6.0 只有在**网络请求的接收和解析，以及请求后的数据通过网络返回给时，使用了多线程**。而数据读写操作还是由单线程来完成的，所以，这样就不会出现并发问题了。

Redis通信协议Resp(Redis的文本协议)

编码

单行字符串以+开头
多行字符串以$开头
整数值以：开头
错误消息用–开头
数组用*开头
null用$ –1 \r\n
空字符串用$ 0 \r\n
结尾统一用\r\n, 多行字符串中间用\r

Reids数据类型

字符串String、字典Hash、列表List、集合Set、有序集合SortedSet。

字符串String

基础

字符串String是Redis最简单的数据结构。Redis所有的数据结构都是以唯一的key字符串作为名称，然后通过这个唯一Key值来获取相应的value数据(get set方法)。不同类型的数据结构的差异在于value结构不一样。
String的一个常见用途是缓存用户信息。我们将用户信息结构体使用JSON序列化为字符串，然后将序列化后的字符串塞进Redis来缓存。同样，取用户信息会经过一次反序列化的过程。

Redis的**字符串是动态字符串，是可以修改的字符串，内部结构实现上类似于Java的ArrayList,底层是SDS(**SDS（Simple Dynamic String ）来存储**)**，采用预分配冗余空间的方式来减少内存的频繁分配，内部的当前字符串时间分配的空间capacity一般要高于实际字符串长度len。**当字符串长度小于1M时，扩容都是加倍现有的空间，如果超过1M，扩容时一次只会多扩1M的空间。需要注意的是字符串最大长度为512M。**

String的实际应用场景比较广泛的有：

1. 缓存功能：String字符串是最常用的数据类型，不仅仅是Redis，各个语言都是最基本类型，因此，利用Redis作为缓存，配合其它数据库作为存储层，利用Redis支持高并发的特点，可以大大加快系统的读写速度、以及降低后端数据库的压力。
2. 计数器：许多系统都会使用Redis作为系统的实时计数器，可以快速实现计数和查询的功能。而且最终的数据结果可以按照特定的时间落地到数据库或者其它存储介质当中进行永久保存。
3. 共享用户Session：用户重新刷新一次界面，可能需要访问一下数据进行重新登录，或者访问页面缓存Cookie，但是可以利用Redis将用户的Session集中管理，在这种模式只需要保证Redis的高可用，每次用户Session的更新和获取都可以快速完成。大大提高效率。

常用指令

可以Set，Get，Del， Mset， Exist， Expire， mset， mget， incr， incrby

SDS动态字符串的优点是什么?

redis是基于C语言的，但是String没有直接采用c的string，而是一种叫做sds，simple dynamic string。sds有三个变量 int len;int free;char buf[];， sds其实是根据redis的需要对string做的一些优化。

好处有：

1. len变量，将获取动态长度的时间复杂度降到了O(1)
2. 杜绝缓冲区溢出

字符串拼接是我们经常做的操作，在C和Redis中一样，也是很常见的操作，但是问题就来了，C是不记录字符串长度的，一旦我们调用了拼接的函数，如果没有提前计算好内存，是会产生缓存区溢出的。而sds里面还有free变量，所以可以知道是否够拼接，不会导致盲目拼接造成的脏数据。

3. 减少修改字符串时带来的内存重分配次数

C语言字符串底层也是一个数组，每次创建的时候就创建一个N+1长度的字符，多的那个1，就是为了保存空字符的，这个空字符也是个坑，但是不是这个环节探讨的内容。

Redis是个高速缓存数据库，如果我们需要对字符串进行频繁的拼接和截断操作，如果我们写代码忘记了重新分配内存，就可能造成缓冲区溢出，以及内存泄露。

内存分配算法很耗时，且不说你会不会忘记重新分配内存，就算你全部记得，对于一个高速缓存数据库来说，这样的开销也是我们应该要避免的。

空间预分配：当我们对SDS进行扩展操作的时候，Redis会为SDS分配好内存，并且根据特定的公式，分配多余的free空间，还有多余的1byte空间（这1byte也是为了存空字符），这样就可以避免我们连续执行字符串添加所带来的内存分配消耗。

惰性空间释放：刚才提到了会预分配多余的空间，很多小伙伴会担心带来内存的泄露或者浪费，别担心，Redis大佬一样帮我们想到了，当我们执行完一个字符串缩减的操作，redis并不会马上收回我们的空间，因为可以预防你继续添加的操作，这样可以减少分配空间带来的消耗，但是**当你再次操作还是没用到多余空间的时候**，Redis也还是会收回对于的空间，防止内存的浪费的

4. 二进制安全

有很多数据结构经常会穿插空字符在中间，比如图片，音频，视频，压缩文件的二进制数据，里面往往就会有空字符\0， c中的字符串是根据空字符来判断结尾的， 而sds中加入了len变量，可以避免这种情况。

有序列表list

基础

Redis的列表相当于Java语言里面的**LinkedList**，注意它是链表而不是数组。这意味着list的插入和删除操作非常快，时间复杂度为O(1)，但是索引定位很慢，时间复杂度为O(n)。

当列表弹出了最后一个元素之后，该数据结构自动被删除，内存被回收。

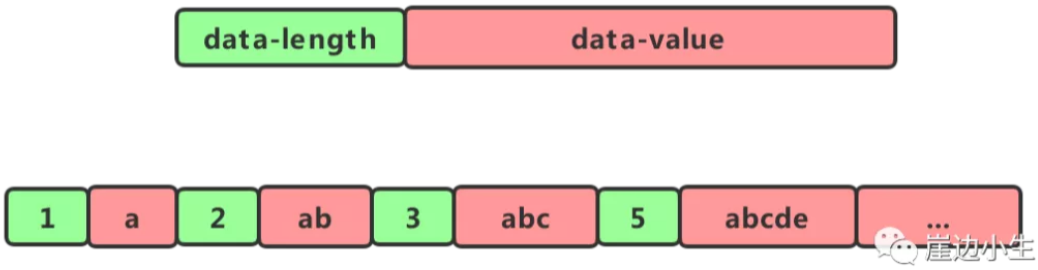
Redis的列表结构常用来保存异步队列使用。将需要延后处理的任务结构体序列化成字符串塞进Redis的列表，另一个线程从这个列表中轮询数据进行处理。

Redis底层存储的不是一个简单的**Linkedlist**，而是被称为快速链表**quicklist**的一个结构。

首先在**列表元素较少的情况下会使用一块连续的内存存储，这个结构是ziplist**，也即是压缩列表。它将所有的元素都紧挨着一起存储，分配的是一块连续的内存。当数据量比较多的时候才会改为**quicklist**。

因为普通的链表需要的附加指针空间太大，会比较浪费空间，而且会加重内存的碎片化。比如这个列表里存的只是**int**类型的数据，结构上还需要两个额外的指针**prev**和**next**。**所以Redis将链表和ziplist结合起来组成了quicklist。也就是多个ziplist使用双向指针串起来使用。这样既满足了快速的插入删除性能，又不会出现太大的空间冗余。**

zipList压缩列表



当一个列表只包含少量列表项,并且每个列表项要么就是小整数值,要么就是长度比较短的字符串,那么Redis就会使用压缩列表来做列表的底层实现。然后zipList，头头相连成为list
当一个哈希只包含少量键值对,比且每个键值对的键和值要么就是小整数值,要么就是长度比较短的字符串,那么Redis就会使用压缩列表来做哈希的底层实现。

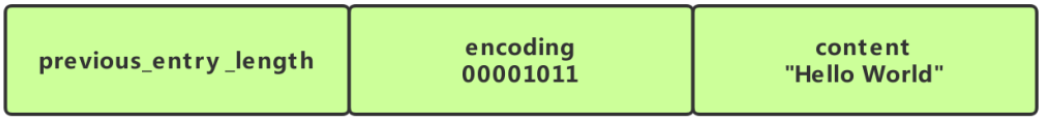
Redis压缩列表节点的构成

每个压缩列表节点可以保存一个字节数组或者一个整数值。其中，字节数组可以是以下三种长度中的一种。

节点的previous_entry_length属性以字节为单位,记录了压缩列表中前一个节点的长度。
previous_entry_length属性的长度可以是1字节或者5字节。

节点的encoding属性记录了节点的content属性所保存数据的类型以及长度。
节点的content属性负责保存节点的值,节点值可以是一个字节数组或者整数,值的类型和长度由节点的encoding属性决定。

保存字节数组



保存整数值

ZipList的一些注意点

- 压缩列表是Redis为节约内存自己设计的一种顺序型数据结构。
- 压缩列表被用作列表键和哈希键的底层实现之一。
- 压缩列表可以包含多个节点,每个节点可以保存一个字节数组或者整数值。
- 添加新节点到压缩列表,或者从压缩列表中删除节点,可能会引发连锁更新操作,但这种操作出现的几率并不高。

应用场景

List本身就是我们在开发过程中比较常用的数据结构了，热点数据更不用说了。

1. 消息队列：Redis的链表结构，可以轻松实现阻塞队列，可以使用左进右出的命令组成来完成队列的设计。比如：数据的生产者可以通过Lpush命令从左边插入数据，多个数据消费者，可以使用BRpop命令阻塞的“抢”列表尾部的数据。
2. 文章列表或者数据分页展示的应用。

比如，我们常用的博客网站的文章列表，当用户量越来越多时，而且每一个用户都有自己的文章列表，而且当文章多时，都需要分页展示，这时可以考虑使用Redis的列表，**列表不但有序同时还支持按照范围内获取元素，可以完美解决分页查询功能**。大大提高查询效率。

常用指令

可以LPUSH， RPUSH， LRANGE

使用过Redis做异步队列么，你是怎么用的?

一般使用list结构作为队列，rpush生产消息，lpop消费消息。当lpop没有消息的时候，要适当sleep一会再重试。

如果对方追问可不可以不用sleep呢?

list还有个指令叫blpop，在没有消息的时候，它会阻塞住直到消息到来。

如果对方接着追问能不能生产一次消费多次呢?

使用pub/sub主题订阅者模式，可以实现 1:N 的消息队列。

如果对方继续追问 pub/sub有什么缺点?

在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如RocketMQ等。

如果对方究极TM追问Redis如何实现延时队列?

这一套连招下来，我估计现在你很想把面试官一棒打死（面试官自己都想打死自己了怎么问了这么多自己都不知道的），如果你手上有一根棒球棍的话，但是你很克制。平复一下激动的内心，然后神态自若的回答道：使用sortedset，拿时间戳作为score，消息内容作为key调用zadd来生产消息，消费者用zrangebyscore指令获取N秒之前的数据轮询进行处理。

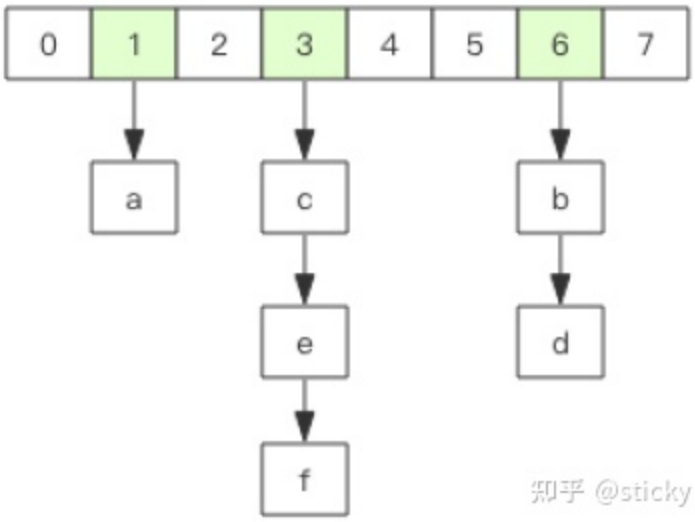
有没有什么能改进pub/sub的?

pub和sub的最大缺点是没有做到数据持久化，所以后续redis有推出了stream来提供数据可持久化的消息队列。

字典hash

基础

Redis的字典相当于Java语言里面的HashMap，它是无序字典。内部实现结构上同Java的HashMap也是一致的，**同样的数组+链表二维结构**。第一维hash的数组位置碰撞时，就会将碰撞的元素使用链表串接起来。



不同的是，**Redis的字典的值只能是字符串**，另外它们rehash的方式也不一样，因为Java的HahMap在字典很大时，rehash是一个耗时的操作，需要一次性全部rehash。Redis为了高性能，不能堵塞服务，所以采用了**渐进式rehash策略。**

渐进式rehash会在rehash的同时，**保留新旧两个hash结构，查询会同时查询两个hash结构**，然后在后续的定时任务中以及hash的子指令中，循序渐进地将旧hash的内容一点点迁移到新的hash结构中。

当hash移除了最后一个元素时，该数据结构自动被删除，内存被回收。

hash结构也可以用来存储用户信息，不同于字符串需要全部序列化整个对象，**hash可以对用户结构中的每个字段单独存储。**这样当我们需要**获取用户信息时可以进行部分获取。**而以整个字符串的形式来保存用户信息的话就只能一次性全部读取，这样会比较浪费网络流量。

hash的缺点在于其结构的存储消耗要高于单个字符串。

扩容和锁容的界限

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**。不过如果 Redis 正在做 **bgsave(持久化命令)**，为了减少内存也得过多分离，Redis 尽量不去扩容，但是如果 hash 表非常满了，**达到了第一维数组长度的 5 倍了**，这个时候就会 **强制扩容**。

常用指令

可以HSET, HGET, HGETALL

无序集合set

基础

Redis的集合相当于Java语言里面的HashSet，它内部的键值对是无序且唯一的。它的内部实现相当于一个特殊的字典，字典所有的value都是一个值NULL。

当集合中的最后一个元素移除之后，数据结构自动删除，内存被回收。set结构可以用来存储活动中奖的用户ID，因为有去重功能，可以保证同一用户不会中将两次。

可以**基于 Set 玩儿交集、并集、差集**的操作，比如交集吧，我们可以把两个人的**好友列表**整一个交集，看看俩人的共同好友是谁？对吧。

反正这些场景比较多，因为对比很快，操作也简单，两个查询一个Set搞定。

常用指令

可以SADD SMEMBERS查询 SISMEMBER是否存在 SCARD获取长度 SPOP

有序链表zset，最终要的一个部分

基础

zset类似于Java中的SortedSet和HashMap的结合体，一方面它是一个set，保证了内部value的唯一性，另一方面它可以给每个value赋予一个score，代表这个value的排序权重。**它的内部实现用的是一种叫做[跳跃列表]也就是跳表的数据结构。**

zset中最后一个value被移除后，数据结构自动删除，内存被回收。zset可以用来保存粉丝列表，value是粉丝的ID，score是关注时间。我们可以对粉丝列表按关注时间进行排序。

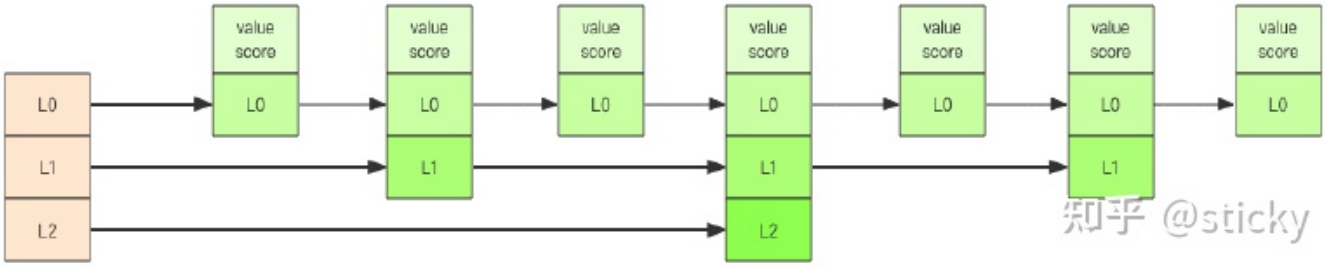
zset还刻意用来存储学生的成绩，value的值是学生的ID，score是他的考试成绩。我们可以对成绩按分数进行排序就可以得到他的名次。

跳跃列表

因为zset要支持随机的插入与删除，所以它不好使用数组来表示。我们需要这个链表按照score值进行排序。**这意味着每当有新元素需要插入时，要定位到特定位置的插入点，这样才可以继续保证链表是有序的**。通常我们会通过**二分查找来找到插入点**，但是二分查找的对象必须是数组，只有数组才可以支持快速位置定位。

跳跃表是一种可以与平衡树媲美的层次化链表结构——查找、删除、添加等操作都可以在对数期望时间下完成， 在mysql中我们其实也是用的b+树来实现的对数期望查询删除操作。

跳跃列表类似于公司的层级制，最下面一层所有的元素都会串起来。然后每隔几个元素挑选出一个代表来，再将这几个代表使用另外一级指针串起来。然后在这些代表里再跳出二级代表，再串起来。最终就形成了金字塔结构。类似于世界地图的位置：**亚洲-->中国->安徽省->安庆市->枞阳县->汤沟镇->田间村->xxxx号**。



[跳跃列表]之所以[跳跃]，是因为内部的元素可能[身兼数职]，比如上图中间的这个元素，同时处于L0、L1和L2层，可以快速在不同层次之间进行[跳跃]。

定位插入点时，现在顶层进行定位，然后下潜到下一级定位，一直下潜到最底层找到合适的位置，将新元素插入进去。相当于二分了，第一次是很小范围内找一个大概范围，然后向下升入。上面每一层链表的节点个数，是下面一层的节点个数的一半，这样查找过程就非常类似于一个二分查找，使得查找的时间复杂度可以降低到 *O(logn)*。当如果是严格要求上下层比例1：2的，插入元素也需要涉及到细分的调整，所以跳表的话，**优化为给每个插入的数分配层数，表示他应该在哪些层出现**，保持每层的比例大致在1:2左右。所以分配的比例大致也是按照50 25 12.5这样子一步步下去。

可以想象，当链表足够长，这样的多层链表结构可以帮助我们跳过很多下层节点，从而加快查找的效率。

为什么要用跳表，而不是平衡树/红黑树?

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

应用场景

1. 排行榜：有序集合经典使用场景。例如视频网站需要对用户上传的视频做排行榜，榜单维护可能是多方面：按照时间、按照播放量、按照获得的赞数等。
2. 用Sorted Sets来做带权重的队列，比如普通消息的score为1，重要消息的score为2，然后工作线程可以选择按score的倒序来获取工作任务。让重要的任务优先执行。

微博热搜榜，就是有个后面的热度值，前面就是名称

常用指令

3. 可以ZADD ZRANGE排序 ZREVRANGE逆序 ZCARD获取长度 ZSCORE ZRANK获取排名

用跳表替换InnoDB中的B+树你觉得行不行?

B+树	平衡二叉树扩展而来	单key,范围，分页	O(Log(n)	除了数据，还多了左右指针，以及叶子节点指针	O(Log(n)，需要调整树的结构，算法比较复杂
跳表	有序链表扩展而来	单key，分页	O(Log(n)	除了数据，还多了指针，但是每个节点的指针小于<2,所以比B+树占用空间小	O(Log(n)，只用处理链表，算法比较简单

高级用法：

Bitmap：

位图是支持按 bit 位来存储信息，可以用来实现 **布隆过滤器（BloomFilter）**；
位图其实就是把存储单位粒度缩小化到bit上面，通过bit位置1还是0表示是否存在。
getbit和setbit是对位进行操作和读取
get和set也可以进行操作和读取不过是获得的ASCII码结果了， 例如he啊 hello之类的，然后bitmap的地位是左边，也就是从左边到右边，不断上升。

HyperLogLog:非精确统计去重Set

提供供不精确的去重计数功能，比较适合用来做大规模数据的去重统计，例如统计 UV；相较于Set，hyperLogLog占用空间小，虽然有误差但是误差也很小。

原理

HyperLogLog实际上不会存储每个元素的值，它使用的是概率算法，通过存储元素的hash值的第一个1的位置，来计算元素数量。是基于概率统计的算法，目前还没有完全理解到位。
redis里面会得到传入对象64位的hash值，其中前14位作为分桶设计思想， HLL里面有2^14大概是16k个桶，然后根据前14位去将当前元素分到对应的桶，然后后50位，去找到最后出现1的情况，最大的可能是第50位出现了1，桶的大小是6bit，所以可以记录63个数，所以50位对应的就是110010， 然后每个桶只存放一个数，如果后来的大，则替换，反之不替换，这种情况下，最终可以得到各个桶中出现1最高的那位，然后去估算出具体的数值大小。结合桶大小和桶内数据。

指令

pfadd， pfcount， pfmerge， 为什么前缀是pf，是因为这个算法是一个名字缩写为pf的教练想到的。

Geospatial: 打卡地点，计算离打卡要求地点的距离之类的

可以用来保存地理位置，并作位置距离计算或者根据半径计算位置等。有没有想过用Redis来实现附近的人？或者计算最优地图路径?
这三个其实也可以算作一种数据结构，不知道还有多少朋友记得，我在梦开始的地方，Redis基础中提到过，你如果只知道五种基础类型那只能拿60分，如果你能讲出高级用法，那就觉得你有点东西。

原理

坐标都是二维的，所以如果要计算距离或者最近单位都是需要进行一个计算， Geo的话，将二维的坐标转换到了一维空间中。类似的思想是对一个正方形不断的分为四份，然后编码，类似哈夫曼编码，最终可以得到一个无线小的区间以及一个编码值。

然后Redis将这个编码值使用52位的整数进行编码，这个时候这个最终编码结果越接近说明距离也越接近。

指令

geoadd 经度 维度 名称 增加元素
geodist key key1 key2 计算距离
geopos 根据key得到经纬度
geordiusbymember 根据半径来查找附近的人

Scan范围查找指令， 同样是快速但丢失精度的指令，替代keys

keys算法是遍历算法，复杂度是O(n), 且会阻塞线程，所以一般情况下谨慎使用。
Scan的特点：
虽然也是O(N)的时间复杂度，但是不会阻塞线程。
提供limit分页参数

原理

scan指令提供了三个参数，游标值cursor 匹配key limti值
然后其实也是去遍历存储所有key的hashMap，只不过是遍历完limit值就返回结果以及当前游标cursor执行到哪儿了。
遍历顺序是高位加法遍历，即从高位开始00 01 10 11这样子，好处是在hash发生扩容和缩容的时候，可以正常继续遍历，不需要担心会不会漏遍历以及重复便利的问题。

布隆过滤器bloom filter

位图+多哈希来判断，在判断不存在方面非常精准，在判断是否存在方面可能会出现一定的误差。
bf.add bf.madd bf.exists

pub/sub：

功能是订阅发布功能，可以用作简单的消息队列。

Pipeline：

可以批量执行一组指令，一次性返回全部结果，可以减少频繁的请求应答。(相当于请求打包，类似HTTP2中的一个TCP执行多个请求)

Lua：

Redis 支持提交 Lua 脚本来执行一系列的功能。利用他的原子性
我在前电商老东家的时候，秒杀场景经常使用这个东西，讲道理有点香，利用他的原子性。
话说你们想看秒杀的设计么？我记得我面试好像每次都问啊，想看的直接点赞后评论秒杀吧。

事务：

最后一个功能是事务，但 Redis 提供的不是严格的事务，**Redis 只保证串行执行命令，并且能保证全部执行，但是执行命令失败时并不会回滚，而是会继续执行下去。**

Redis缓存问题

redis 缓存雪崩，缓存穿透，缓存击穿

缓存雪崩：

目前电商首页以及热点数据都会去做缓存，一般缓存都是定时任务去刷新，或者查不到之后更新缓存的，定时任务刷新就有一个问题。举个例子：

如果所有key的失效时间都是12小时，中午12点刷新的，零点有一个大促活动大量用户涌入，假设每秒6000个请求，本来缓存可以抗住每秒5000个请求，但是缓存中所有Key都失效了。此时6000个/秒的请求全部落在了数据库上，数据库必然扛不住，真实情况可能DBA都没反应过来就直接挂了，此时，如果没有什么特别的方案来处理，DBA很着急，重启数据库，但是数据库立马又被新流量给打死了。这就是缓存雪崩。

同一时间大面积失效，瞬间Redis跟没有一样，那这个数量级别的请求直接达到数据库上几乎是灾难性的，如果挂的是一个用户服务的库，那其他依赖他的所有接口几乎都会报错，如果没有做熔断等策略基本就是瞬间挂一片的节奏。就算没有打崩数据库，但这么大流量达到数据库上，本身也是系统设计的一个问题。

处理方法：

在批量往Redis存数据的时候，把每个Key的失效时间都加个随机值就好了，这样可以保证数据不会在同一时间大面积失效。

setRedis（key,value,time+Math.random()*10000）；

如果Redis是集群部署，将热点数据均匀分布在不同的Redis库中也能避免全部失效。

或者设置热点数据永不过期，有更新操作就更新缓存就好了（比如运维更新了首页商品，那就刷新缓存就好了，不用设置过期时间），电商首页的数据也可以用这个操作。

缓存击穿跟：

是指某个Key非常热点，在不停地扛着大量请求，大并发集中地对这个点进行访问，当这个Key在失效地瞬间，持续地大并发直接落到了数据库上，就在这个Key的点上击穿了缓存。而缓存雪崩则是因为大面积地缓存失效。比如说微博的热搜第一，突然给下了，很有可能会出现缓存击穿的情况。

缓存穿透

是指缓存和数据库都没有的数据，而用户（黑客）不断发起请求(布隆过滤器)。举个例子：

我们数据库的id都是从1自增的，如果发起id=-1的数据或者id特别大不存在的数据，这样的不断攻击导致数据库压力很大，严重会击垮数据库；
所以针对缓存穿透的话，需要一个参数校验的过程，避免出现非故意性的不存在数据的高访问。在钉钉考勤里面的话，在实际进入打卡流程的时候，有一个简单的校验参数的步骤，主要校验的就是公司id，时间， 地点， 考勤组id这种字段。

解决方法

1. 缓存穿透可以在接口层增加校验，比如用户鉴权、参数校验、不合法地校验直接return，比如id做基础校验，id<=0直接拦截；
2. “布隆过滤器(Bloom Filter)”能够很好地预防预防患侧穿透地发生，其利用高效地数据结构和算法快速判断处你这个Key是否在数据库中存在，不存在return就好了，存在就去查DB刷新KV再return；
3. 缓存击穿的话，设置热点数据永不过期，再加上互斥锁就搞定了， 或者就是热点数据访问就更新失效时间。。

其实针对这种崩盘的情况，除了redis本身这些操作，一般都还有一些其他的措施的。比如限流(控死接入流量)+降级(return 简单值)，也有主从+哨兵不会都崩盘，然后持久化保证崩盘快速恢复。

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

Redis过期机制是怎么样的

Redis的过期策略，是有**定期删除+惰性删除**两种。
定期好理解，默认100ms就随机抽一些设置了过期时间的key，去检查是否过期，过期了就删了。

为啥不扫描全部设置了过期时间的key呢?

假如Redis里面所有的key都有过期时间，都扫描一遍？那太恐怖了，而且我们线上基本上也都是会设置一定的过期时间的。全扫描跟你去查数据库不带where条件不走索引全表扫描一样，100s一次，Redis累都累死了。。

如果一直没随机到很多key，里面不就存在大量的无效key了?

好问题，**惰性删除**，见名知意，惰性嘛，我不主动删，我懒，我等你来查询了我看看你过期没，过期就删了还不给你返回，没过期该怎么样就怎么样。

最后就是如果的如果，定期没删，我也没查询，那可咋整？---内存满了。被迫淘汰。

Redis内存淘汰机制是怎么样的(4类2种情况)

官网上给到的内存淘汰机制是以下几个：

1. **noeviction:直接返回错误**，当内存限制达到并且客户端尝试执行会让更多内存被使用的命令（大部分的写入指令，但DEL和几个例外）
2. **allkeys-lru: 尝试回收最少使用的键（LRU），**使得新添加的数据有空间存放。
3. **volatile-lru: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。**
4. allkeys-random: 回收随机的键使得新添加的数据有空间存放。
5. **volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。**
6. volatile-ttl: **回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,**使得新添加的数据有空间存放。

如果没有键满足回收的前提条件的话，策略volatile-lru, volatile-random以及volatile-ttl就和 noeviction 差不多了

redis实现分布式锁

并发去操作Redis数据的话，会有什么问题呢?

某个时刻，多个系统实例都去更新某个 key。可以**基于 Zookeeper 实现分布式锁(Tair好像不是基于Zookeeper的)**。每个系统通过 Zookeeper 获取分布式锁，确保同一时间，只能有一个系统实例在操作某个 Key，别人都不允许读和写。然后redis里面其实也有和分布式锁相关的操作

你要写入缓存的数据，都是从 MySQL 里查出来的，都得写入 MySQL 中，写入 MySQL 中的时候必须保存一个时间戳，从 MySQL 查出来的时候，时间戳也查出来。

每次要写之前，先判断一下当前这个 Value 的时间戳是否比缓存里的 Value 的时间戳要新。如果是的话，那么可以写，否则，就不能用旧的数据覆盖新的数据。

Setnx(Set if not exist）+ expire

setnx是去获取锁，然后expire是去设置锁失效，其实也就是防止一个线程在setnx后出问题了，导致锁一直无法释放造成的死锁。

然后setnx + expire是两步操作，所以的话，存在一定的问题， 然后官方推出了set xxx ex 时间 nx true

RedLock

上面的setnx有一个问题，就是如果向主节点获取锁成功，而这个消息还没传到从节点时，主节点挂了，从节点升级为主节点，此时另一个线程又向从节点要了同样对象的锁。这个时候出现系统中同一把锁被两个客户端同时持有了。

redlock需要有多个redis实例，redis之间是相互独立的不能是主从关系的。采用的是**大部分机制**

流程

加锁时，会向过半节点发送set(key value nx=true, ex=xxx)指令，只要过半节点执行成功，杰认为加锁成功，del的时候也是同理，需要过半节点完成删除操作。
这种设计需要多个节点进行读写，所以效率会有一定的下降。

Redis 高可用性

Redis集群之间是怎么进行数据交互的？以及Redis是怎么进行持久化的?

持久化的话是Redis高可用中比较重要的一个环节，因为Redis数据在内存的特性，持久化必须得有，我了解到的持久化是有两种方式的。

RDB(Redis DataBase)：RDB 持久化机制，是对 Redis 中的数据执行周期性的持久化。
对方追问**RDB的原理**是什么？你给出两个词汇就可以了，fork和cow。fork是指redis通过创建子进程来进行RDB操作，cow指的是copy on write，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写脏的页面数据会逐渐和子进程分离开来。

注：回答这个问题的时候，如果你还能说出AOF和RDB的优缺点，我觉得我是面试官在这个问题上我会给你点赞，两者其实区别还是很大的，而且涉及到Redis集群的数据同步问题等等。想了解的伙伴也可以留言，我会专门写一篇来介绍的。
AOF(Append Only File)：AOF 机制对每条写入命令作为日志，以 append-only 的模式写入一个日志文件中，因为这个模式是只追加的方式，所以没有任何磁盘寻址的开销，所以很快，有点像Mysql中的binlog。
两种方式都可以把Redis内存中的数据持久化到磁盘上，然后再将这些数据备份到别的地方去，RDB更适合做冷备，AOF更适合做热备，比如我杭州的某电商公司有这两个数据，我备份一份到我杭州的节点，再备份一个到上海的，就算发生无法避免的自然灾害，也不会两个地方都一起挂吧，这灾备也就是异地容灾，地球毁灭他没办法。

tip：两种机制全部开启的时候，Redis在重启的时候会默认使用AOF去重新构建数据，因为AOF的数据是比RDB更完整的。

AOF和RDB两种机制各自优缺点是啥

RDB：全量持久化

RDB是按照一个较长时间间隔， 将数据集生成一个快照。但我们知道，Redis 是一个 **单线程** 的程序，这意味着，我们不仅仅要响应用户的请求，**还需要进行内存快照。而后者要求 Redis 必须进行 IO 操作**，这会严重拖累服务器的性能。

还有一个**重要的问题是，我们在 持久化的同时，内存数据结构 还可能在 变化**，比如一个大型的 hash 字典正在持久化，结果一个请求过来把它删除了，可是这才刚持久化结束，咋办?

RDB底层原理： 使用系统多进程 COW(Copy On Write) 机制 | fork 函数

Redis 在持久化时会调用 glibc 的函数 **fork 产生一个子进程**，简单理解也就是基于当前进程 复制 了一个进程，**主进程和子进程会共享内存里面的代码块和数据段。**

快照持久化 可以完全交给 子进程 来处理，父进程 则继续 处理客户端请求。**子进程 做数据持久化，它不会修改现有的内存数据结构**，它只是对数据结构进行遍历读取，然后序列化写到磁盘中。但是 **父进程不一样，它必须持续服务客户端请求，然后对 内存数据结构进行不间断的修改。**

使用操作系统的 COW (copy on write)机制来进行 **数据段页面** 的分离。数据段是由很多操作系统的页面组合而成，当父进程对其中一个页面的数据进行修改时，会将被共享的页面复 制一份分离出来，然后 **对这个复制的页面进行修改。**

然后下一个时刻，RDB再把这段中的delete给快照掉

优点：他会**生成多个数据文件**，每个数据文件分别都代表了某一时刻Redis里面的数据，这种方式，有没有觉得很适合做冷备，完整的数据运维设置定时任务，定时同步到远端的服务器，比如阿里的云服务，这样一旦线上挂了，你想**恢复多少分钟之前的数据，就去远端拷贝一份之前的数据就好了。**

RDB对Redis的性能影响非常小，是因为在同步数据的时候他只是fork了一个子进程去做持久化的，而且他在数据恢复的时候速度比AOF来的快。

缺点：

RDB都是快照文件，都是默认五分钟甚至更久的时间才会生成一次，这意味着你这次同步到下次同步这中间五分钟的数据都很可能全部丢失掉。**AOF则最多丢一秒的数据**，数据完整性上高下立判。

还有就是**RDB在生成数据快照的时候，如果文件很大，客户端可能会暂停几毫秒甚至几秒**，你公司在做秒杀的时候他刚好在这个时候fork了一个子进程去生成一个大快照，哦豁，出大问题。

AOF：增量持久化

AOF在对日志文件进行操作的时候是以**append-only的方式去写**的，他只是**追加的方式写数据**，自然就少了很多磁盘寻址的开销了，写入性能惊人，文件也不容易破损。**类似binlog只记录修改。**

也因为是增量日志，所以AOF有时候文件会很大，就要采用**AOF重写/瘦身原则。**

其 **原理** 就是 **开辟一个子进程** 对内存进行 **遍历** 转换成一系列 Redis 的操作指令，**序列化到一个新的 AOF 日志文件** 中。序列化完毕后再将操作期间发生的 **增量 AOF 日志** 追加到这个新的 AOF 日志文件中，

优点：

上面提到了，RDB五分钟一次生成快照，但是**AOF是一秒一次去通过一个后台的线程fsync操作**，那最多丢这一秒的数据。

AOF的日志是通过一个叫非常可读的方式记录的，这样的特性就适合做**灾难性数据误删除的紧急恢复**了，比如公司的实习生通过flushall清空了所有的数据，只要这个时候后台重写还没发生，你马上拷贝一份AOF日志文件，把最后一条flushall命令删了就完事了。

tip：我说的命令你们别真去线上系统操作啊，想试去自己买的服务器上装个Redis试，别到时候来说，敖丙真是个渣男，害我把服务器搞崩了，Redis官网上的命令都去看看，不要乱试！！！

缺点：

一样的数据，**AOF文件比RDB还要大**。

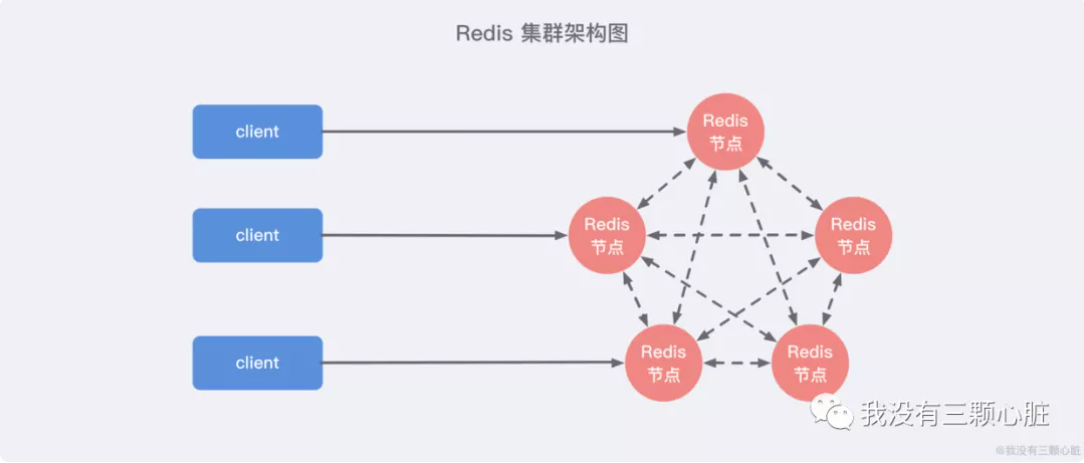
AOF开启后，Redis支持写的QPS会比RDB支持写的要低，他不是每秒都要去异步刷新一次日志嘛fsync，当然即使这样性能还是很高，我记得ElasticSearch也是这样的，异步刷新缓存区的数据去持久化，为啥这么做呢，不直接来一条怼一条呢，那我会告诉你这样性能可能低到没办法用的，大家可以思考下为啥哟。

两者之间如何选择:

小孩子才做选择，**我全都要**，你单独用RDB你会丢失很多数据，你单独用AOF，你数据恢复没RDB来的快，真出什么时候第一时间用RDB恢复，然后AOF做数据补全，真香！冷备热备一起上，才是互联网时代一个高健壮性系统的王道。

Redis 4.0 为了解决这个问题，带来了一个新的持久化选项——**混合持久化**。将 rdb 文件的内容和增量的 AOF 日志文件存在一起。这里的 AOF 日志不再是全量的日志，而是 **自持久化开始到持久化结束** 的这段时间发生的增量 AOF 日志，通常这部分 AOF 日志很小：既解决了RDB丢失大量数据的同时也解决了AOF文件过大的问题。

Reids cluster集群以及分片方案



上图 展示了 **Redis Cluster** 典型的架构图，集群中的每一个 Redis 节点都 **互相两两相连**，客户端任意 **直连** 到集群中的 **任意一台**，就可以对其他 Redis 节点进行 **读写** 的操作。

Redis 集群中内置了16384个哈希槽。当客户端连接到 Redis 集群之后，会同时得到一份关于这个**集群的配置信息**，当客户端具体对某一个key值进行操作时，会计算出它的一个 Hash 值，然后把结果对16384**求余数**，这样每个key都会对应一个编号在0-16383之间的哈希槽，Redis 会根据节点数量**大致均等**的将哈希槽映射到不同的节点。

再结合集群的配置信息就能够知道这个 key 值应该存储在哪一个具体的 Redis 节点中，如果不属于自己管，那么就会使用一个特殊的 MOVED 命令来进行一个跳转，告诉客户端去连接这个节点以获取数据：

集群的主要作用：

1. **数据分区：** 数据分区*(或称数据分片)*是集群最核心的功能。集群将数据分散到多个节点，**一方面**突破了 Redis 单机内存大小的限制，**存储容量大大增加**；**另一方面**每个主节点都可以对外提供读服务和写服务，**大大提高了集群的响应能力**。Redis 单机内存大小受限问题，在介绍持久化和主从复制时都有提及，例如，如果单机内存太大，bgsave和bgrewriteaof的fork操作可能导致主进程阻塞，主从环境下主机切换时可能导致从节点长时间无法提供服务，全量复制阶段主节点的复制缓冲区可能溢出……
2. **高可用：** 集群支持主从复制和主节点的 **自动故障转移** *（与哨兵类似）*，当任一节点发生故障时，集群仍然可以对外提供服务。

集群中数据如何分区?

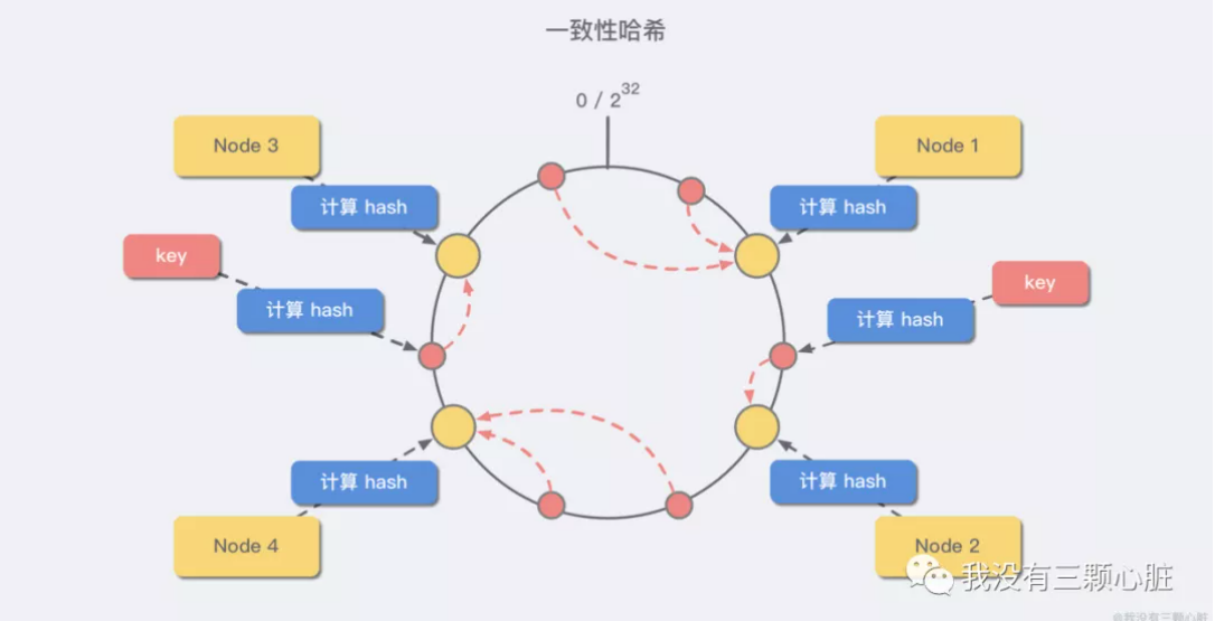
方案一：哈希值 % 节点数(最简单的实现方式)

哈希取余分区思路非常简单：计算key的 hash 值，然后对节点数量进行取余，从而决定数据映射到哪个节点上。

不过该方案最大的问题是，**当新增或删减节点时**，节点数量发生变化，系统中所有的数据都需要 **重新计算映射关系**，引发大规模数据迁移。

方案二：一致性哈希分区(分布式中常用的一致性，改进了新增和删除节点时，所有节点都要涉及调整)

一致性哈希算法将 **整个哈希值空间** 组织成一个虚拟的圆环，范围是 *[0 - 232 - 1]*，对于每一个数据，根据 key 计算 hash 值，确数据在环上的位置，然后从此位置沿顺时针行走，找到的第一台服务器就是其应该映射到的服务器：



与哈希取余分区相比，一致性哈希分区将**增减节点的影响限制在相邻节点**。以上图为例，如果在node1和node2之间增加node5，则只有node2中的一部分数据会迁移到node5；如果去掉node2，则原node2中的数据只会迁移到node4中，只有node4会受影响。

一致性哈希分区的主要问题在于，当 **节点数量较少** 时，增加或删减节点，**对单个节点的影响可能很大**，造成数据的严重不平衡。还是以上图为例，如果去掉 node2，node4 中的数据由总数据的 1/4 左右变为 1/2 左右，与其他节点相比负载过高。

方案三：带有虚拟节点的一致性哈希分区(改进了节点数量较少，影响较大的情况)

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

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

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

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

集群间通信

在**集群**中，没有数据节点与非数据节点之分：**所有的节点都存储数据，也都参与集群状态的维护**。为此，集群中的每个节点，都提供了两个 TCP 端口：

- **普通端口：**即我们在前面指定的端口*(7000等)*。普通端口主要用于为客户端提供服务 *(与单机节点类似)*；但在节点间数据迁移时也会使用。

- **集群端口：** 端口号是普通端口 + 10000 *（10000是固定值，无法改变）*，如 7000 节点的集群端口为 17000。**集群端口只用于节点之间的通信**，如搭建集群、增减节点、故障转移等操作时节点间的通信；不要使用客户端连接集群接口。为了保证集群可以正常工作，在配置防火墙时，要同时开启普通端口和集群端口。

通信协议有哪些?

节点间通信，按照通信协议可以分为几种类型：单对单、广播、Gossip 协议等。重点是广播和 Gossip 的对比。

- 广播是指向集群内所有节点发送消息。**优点**是集群的收敛速度快(集群收敛是指集群内所有节点获得的集群信息是一致的)，**缺点**是每条消息都要发送给所有节点，CPU、带宽等消耗较大。
- Gossip 协议的特点是：在节点数量有限的网络中，**每个节点都“随机”的与部分节点通信** *（并不是真正的随机，而是根据特定的规则选择通信的节点）*，经过一番杂乱无章的通信，每个节点的状态很快会达到一致。Gossip 协议的 **优点** 有负载 *（比广播）* 低、去中心化、容错性高 *（因为通信有冗余）* 等；**缺点** 主要是集群的收敛速度慢。

消息类型有哪些?

节点间发送的消息主要分为 5 种：meet 消息、ping 消息、pong 消息、fail 消息、publish 消息。不同的消息类型，通信协议、发送的频率和时机、接收节点的选择等是不同的

- **MEET 消息：** 在节点握手阶段，当节点收到客户端的CLUSTER MEET命令时，会向新加入的节点发送MEET消息，请求新节点加入到当前集群；新节点收到 MEET 消息后会回复一个PONG消息。
- **PING 消息：** 集群里每个节点每秒钟会选择部分节点发送PING消息，接收者收到消息后会回复一个PONG消息。**PING 消息的内容是自身节点和部分其他节点的状态信息**，作用是彼此交换信息，以及检测节点是否在线。PING消息使用 Gossip 协议发送，接收节点的选择兼顾了收敛速度和带宽成本，**具体规则如下**：(1)随机找 5 个节点，在其中选择最久没有通信的 1 个节点；(2)扫描节点列表，选择最近一次收到PONG消息时间大于cluster_node_timeout / 2的所有节点，防止这些节点长时间未更新。
- **PONG消息：** PONG消息封装了自身状态数据。可以分为两种：**第一种**是在接到MEET/PING消息后回复的PONG消息；**第二种**是指节点向集群广播PONG消息，这样其他节点可以获知该节点的最新信息，例如故障恢复后新的主节点会广播PONG消息。
- **FAIL 消息：** 当一个主节点判断另一个主节点进入FAIL状态时，会向集群广播这一FAIL消息；接收节点会将这一FAIL消息保存起来，便于后续的判断。
- **PUBLISH 消息：** 节点收到 PUBLISH 命令后，会先执行该命令，然后向集群广播这一消息，接收节点也会执行该 PUBLISH 命令。

集群数据如何存储的有了解吗?

节点为了存储集群状态而提供的数据结构中，最关键的是 clusterNode 和 clusterState 结构：前者记录了一个节点的状态，后者记录了集群作为一个整体的状态。

clusterNode 结构

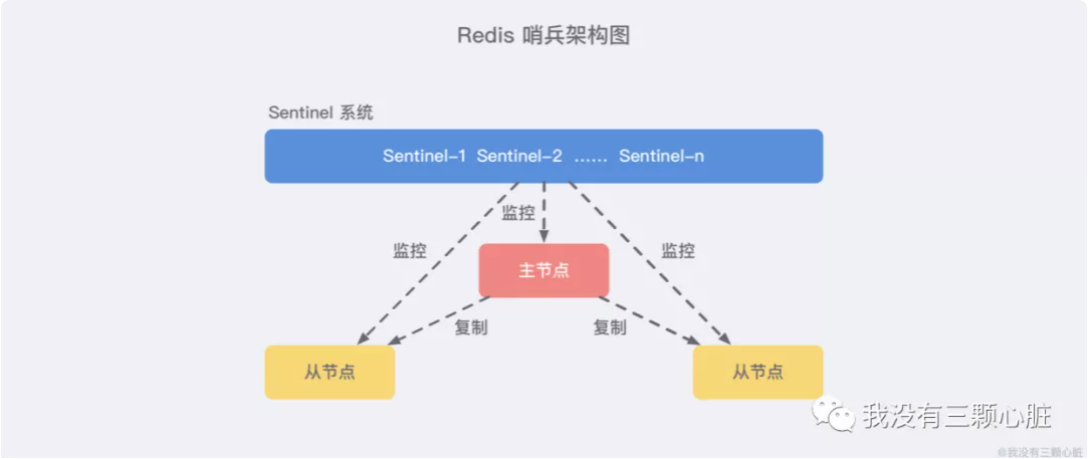
clusterNode 结构保存了 **一个节点的当前状态**，包括创建时间、节点 id、ip 和端口号等。每个节点都会用一个 clusterNode 结构记录自己的状态，并为集群内所有其他节点都创建一个 clusterNode 结构来记录节点状态。

节点ID， 创建时间，节点端口号，槽的数量

clusterState 结构

clusterState 结构保存了在当前节点视角下，集群所处的状态。
集群状态，数量 哈希表

Redis哨兵原理简单介绍一下



*上图*展示了一个典型的哨兵架构图，它由两部分组成，哨兵节点和数据节点：

- **哨兵节点：**哨兵系统由一个或多个哨兵节点组成，哨兵节点是特殊的 Redis 节点，不存储数据；
- **数据节点：** 主节点和从节点都是数据节点；被哨兵节点监听，有个概念叫心跳。

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

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

- **通知（Notification）：** 哨兵可以将故障转移的结果发送给客户端。

Master选举

故障转移操作的第一步 要做的就是在已下线主服务器属下的所有从服务器中，挑选出一个状态良好、数据完整的从服务器，然后向这个从服务器发送 slaveof no one 命令，将这个从服务器转换为主服务器。但是这个从服务器是怎么样被挑选出来的呢?

1. 在失效主服务器属下的从服务器当中， 那些被标记为主观下线、已断线、或者最后一次回复 PING 命令的时间大于五秒钟的从服务器都会被**淘汰**。
2. 在失效主服务器属下的从服务器当中， 那些与失效主服务器连接断开的时长超过 down-after 选项指定的时长十倍的从服务器都会被**淘汰**。
3. 在 **经历了以上两轮淘汰之后** 剩下来的从服务器中， 我们选出 **复制偏移量（replication offset）最大** 的那个 **从服务器** 作为新的主服务器；如果复制偏移量不可用，或者从服务器的复制偏移量相同，那么 **带有最小运行 ID** 的那个从服务器成为新的主服务器

Redis 主从复制, 完整重同步，部分重同步

5.1 连接过程

1. 主服务器创建快照文件，发送给从服务器，并在发送期间使用缓冲区记录执行的写命令。快照文件发送完毕之后，开始从服务器发送存储在缓存区中的写命令；
2. 从服务器丢失所有旧数据，载入主服务器发送的快照文件，之后从服务器开始接受主服务器发送的写命令；
3. 主服务器每执行一次写命令，就向从服务器发送相同的写命令。

5.2 主从链

随着负载不断上升，逐服务器可能无法很快地更新所有从服务器，或者重新连接和重新同步从服务器将导致系统超载。为了解决这个问题，可以创建一个中间层来分担主服务器的复制工作。中间层的服务器是最上层服务器的从服务器，又是最下层服务器的主服务器。

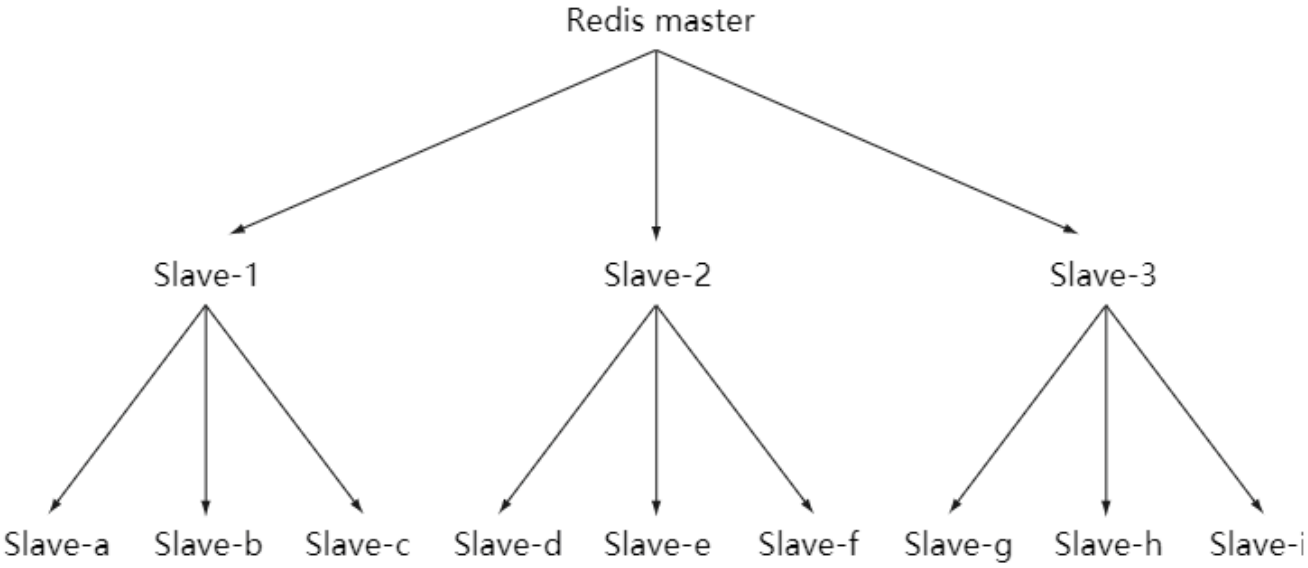


Figure 4.1An example Redis master/slave replica tree with nine lowest-level slaves and three intermediate replication helper servers

Sentinel（哨兵）可以监听集群中的服务器，并在主服务器进入下线状态时，自动从从服务器选举出新的主服务器。

5.3 分片

分片是将数据划分为多个部分的方法，可以将数据存储到多台机器里面，这种方法在解决某些问题时可以获得线程级别的性能提升。

例如有4个Redis实例R0，R1，R2，R3，还有很多表示用户的键user:1，user:2，...，有不同的方式来选择一个指定的键存储在哪个实例中。

最简单的方式是范围分片，例如用户id从0–1000的存储到实例R0中，用户id从1001–2000的存储到实例R1中，等待。但是这样需要维护一张映射范围表，维护操作代价很高。

还有一种方式是哈希分片，使用CRC32哈希函数将键转换为一个数字，再对实例数量求模就能指定应该存储的实例。

根据执行分片的位置，可以分成三种分片方式：

1. 客户端分片：客户端使用一致性哈希等算法决定键应当分布到哪个节点；
2. 代理分片：将客户端请求发送到代理上，由代理转发请求到正确的节点上；
3. 服务器分片：Redis Cluster

Redis 数据一致性

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的问题，那么你如何解决一致性问题?

先删缓存，再更新数据库：

先删除缓存，数据库还没有更新成功，此时如果读取缓存，缓存不存在，去数据库中读取到的是旧值，缓存不一致发生。

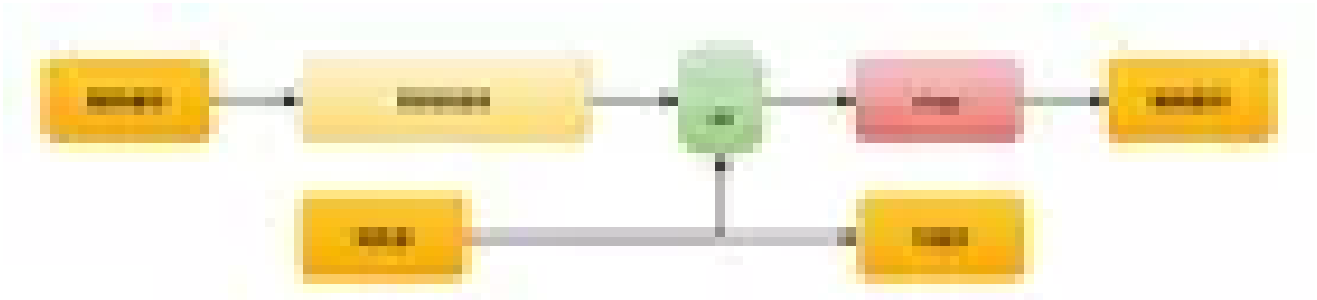
解决方案：

延时双删的方案的思路是，为了避免更新数据库的时候，其他线程从缓存中读取不到数据，就在更新完数据库之后，再sleep一段时间，然后再次删除缓存。

sleep的时间要对业务读写缓存的时间做出评估，sleep时间大于读写缓存的时间即可。

流程如下：

1. 线程1删除缓存，然后去更新数据库
2. 线程2来读缓存，发现缓存已经被删除，所以直接从数据库中读取，这时候由于线程1还没有更新完成，所以读到的是旧值，然后把旧值写入缓存
3. 线程1，根据估算的时间，sleep，由于sleep的时间大于线程2读数据+写缓存的时间，所以缓存被再次删除
4. 如果还有其他线程来读取缓存的话，就会再次从数据库中读取到最新值



先更新数据库，再删缓存

这个就更明显的问题了，更新数据库成功，如果删除缓存失败或者还没有来得及删除，那么，其他线程从缓存中读取到的就是旧值，还是会发生不一致。

解决方案：

先更新数据库，成功后往消息队列发消息，消费到消息后再删除缓存，借助消息队列的重试机制来实现，达到最终一致性的效果。

统一的思路其实可以是不对缓存进行操作

每次放入缓存的时候，设置一个过期时间，比如5分钟，以后的操作只修改数据库，不操作缓存，等待缓存超时后从数据库重新读取。

最经典的缓存+数据库读写的模式，就是 Cache Aside Pattern

- 读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。
- 更新的时候，**先更新数据库，然后再删除缓存**。

为啥是删除缓存而不是更新缓存

一方面，更新成本比较大， 另一方面，更新的数据不代表是高平被访问的数据，只有高倍访问数据才适合放缓存里面。

Redis 缓存更新问题

缓存更新方式这是决定在使用缓存时就该考虑的问题。

缓存的数据在数据源发生变更时需要对缓存进行更新，数据源可能是 DB，也可能是远程服务。更新的方式可以是**主动更新**。数据源是 DB 时，可以在更新完 DB 后就直接更新缓存。

当数据源不是 DB 而是其他远程服务，可能无法及时主动感知数据变更，这种情况下一般会选择对缓存数据设置失效期，也就是数据不一致的最大容忍时间。

这种场景下，可以选择**失效更新，key 不存在或失效时先请求数据源获取最新数据，然后再次缓存，并更新失效期。**

但这样做有个问题，如果依赖的远程服务在更新时出现异常，则会导致数据不可用。改进的办法是**异步更新，就是当失效时先不清除数据，继续使用旧的数据，然后由异步线程去执行更新任务。这样就避免了失效瞬间的空窗期**。另外还有一种**纯异步更新方式，定时对数据进行分批更新。**实际使用时可以根据业务场景选择更新方式。

双写一致性(数据库的数据跟缓存的数据不一致,发生在更新时期)

一般来说，执行更新操作时，我们会有两种选择：

- 先操作数据库，再操作缓存
- 先操作缓存，再操作数据库

首先，要明确的是，无论我们选择哪个，我们都希望这**两个操作要么同时成功，要么同时失败**。所以，这会演变成一个**分布式事务**的问题。

所以，**如果原子性被破坏了**，可能会有以下的情况：

- **操作数据库成功了，操作缓存失败了**。
- **操作缓存成功了，操作数据库失败了**

操作缓存

1. 更新缓存
2. 删除缓存

一般我们都是采取**删除缓存**缓存策略的，原因如下：

1. 高并发环境下，无论是先操作数据库还是后操作数据库而言，如果加上更新缓存，那就**更加容易**导致数据库与缓存数据不一致问题。(删除缓存**直接和简单**很多)
2. 如果每次更新了数据库，都要更新缓存【这里指的是频繁更新的场景，这会耗费一定的性能】，倒不如直接删除掉。等再次读取时，缓存里没有，那我到数据库找，在数据库找到再写到缓存里边(体现**懒加载**)

基于这两点，对于缓存在更新时而言，都是建议执行**删除**操作！

方案一 先更新数据库， 再删除缓存Cache Aside Pattern

正常的情况是这样的：

- 先操作数据库，成功；但这个时候可能会有线程来读缓存。
- 再删除缓存，也成功；

删除缓存失败的解决思路：

- 将需要删除的key发送到**消息队列**中
- 自己消费消息，获得需要删除的key
- **不断重试删除操作，直到成功**

方案二 先删除缓存，再更新数据库

正常情况是这样的：

- 先删除缓存，成功；
- 再更新数据库，也成功；

看起来是很美好，但是我们在并发场景下分析一下，就知道还是有问题的了：

- 线程A删除了缓存
- 线程B查询，发现缓存已不存在
- 线程B去数据库查询得到旧值
- 线程B将旧值写入缓存
- 线程A将新值写入数据库

所以也会导致数据库和缓存不一致的问题。

并发下解决数据库与缓存不一致的思路：

- 将删除缓存、修改数据库、读取缓存等的操作积压到**队列**里边，实现**串行化**
- **延时双删**的方案的思路是，为了避免更新数据库的时候，其他线程从缓存中读取不到数据，就在更新完数据库之后，再sleep一段时间，然后再次删除缓存。

sleep的时间要对业务读写缓存的时间做出评估，sleep时间大于读写缓存的时间即可。

流程如下：

1. 线程1删除缓存，然后去更新数据库
2. 线程2来读缓存，发现缓存已经被删除，所以直接从数据库中读取，这时候由于线程1还没有更新完成，所以读到的是旧值，然后把旧值写入缓存
3. 线程1，根据估算的时间，sleep，由于sleep的时间大于线程2读数据+写缓存的时间，所以缓存被再次删除
4. 如果还有其他线程来读取缓存的话，就会再次从数据库中读取到最新值

Redis事务

redis事务的操作指令有mulit exec rollback，具体的使用流程是

先mulit

然后进行一系列的操作

然后exec，此时开始执行从mulit到exec之间的一系列操作。

Redis场景问题

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

使用keys指令可以扫出指定模式的key列表。类似于 key 然后正则表达式

对方接着追问：如果这个redis正在给线上的业务提供服务，那使用keys指令会有什么问题?

这个时候你要回答redis关键的一个特性：redis的单线程的。keys指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用scan指令，scan指令可以无阻塞的提取出指定模式的key列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用keys指令长。

不过，增量式迭代命令也不是没有缺点的： 举个例子， 使用 SMEMBERS 命令可以返回集合键当前包含的所有元素， 但是对于 SCAN 这类增量式迭代命令来说， 因为在对键进行增量式迭代的过程中， 键可能会被修改， 所以增量式迭代命令只能对被返回的元素提供有限的保证 。

rdb 结束或者 aof 重写结束的时候，子进程如何通知父进程；

子进程返回0，父进程返回子进程pid表示rdb完成了，如果返回的pid=-1，说明rdb的过程出现问题。

如果redis上三分钟有效期的临时数据在申请过程中过期了怎么办（看门狗）

redis 如果一个key特别大怎么办

redis –cli –h –p ––bigkeys 找出这个key ， 然后干掉他