Redis

基础

redis为什么这么快（高）

- 纯内存操作, 比磁盘要快
- 单线程操作, 避免了频繁的线程上下文切换
- 采用了非阻塞I/O多路复用机制
- Redis 内置了多种优化过后的数据结构实现. 如跳表

redis高性能怎么实现的?

性能一般基于两个方面, 一个是计算操作, 一个是读写操作.

- 计算操作, 由于redis是基于内存的, 所以计算操作(命令执行)很快, 然后单线程执行命令比较快. 所以redis的性能瓶颈不在计算操作, 而在于网络io
  - 单线程执行为啥快，那是因为单线程执行的话你就不需要加锁来控制了，锁是很重的很耗费资源.
  - 多线程的话, 也需要线程上下文切换, 这样的话性能也会降低.
- 读写操作, 无非就是网络io和磁盘io
  - 磁盘io优化: rdb持久化时, 会创建一个子进程来生成 RDB 文件，这样可以避免主线程的阻塞. 这也是一种写入时复制的思想
  - 网络io优化:
    - io多路复用: select epoll, 可以同时监听多个socket连接请求

    - 事件派发机制: 有很多不同性质的socket, redis有不同的handler来处理这些socket事件, redis6.0使用多线程来处理这些handler

Redis常用的数据结构有哪些？（高)

- 5种基础数据类型：String(字符串）、List(列表）、Set(集合）、Hash(散列）、Zset(有序集合)
- 4种特殊数据类型：HyperLogLogs(基数统计）、Bitmap(位存储）、geo(地理位置)、stream（消息队列)

redis每种数据类型的使用场景（高)

- String: 最常规的 set/get 操作，Value 可以是 String 也可以是数字。一般做一些复杂的计数功能的缓存。
- Hash: 这里 Value 存放的是结构化的对象，比较方便的就是操作其中的某个字段. 如果单纯做对象的存储, 那么直接使用string即可, 如果需要对对象中的字段做操作, 那么用hash.
- List:
    - list支持两端存取, 不能从中间取. 若从一侧存取, 则是栈. 若从异侧存取, 则是队列.
    - 使用 List 的数据结构，可以做简单的消息队列的功能。另外，可以利用 lrange 命令，做基于 Redis 的分页功能，性能极佳，用户体验好
- Set: 因为 Set 堆放的是一堆不重复值的集合。所以可以做全局去重的功能。我们的系统一般都是集群部署，使用 JVM 自带的 Set 比较麻烦。另外，就是利用交集、并集、差集等操作，交集就可以计算共同喜好，共同关注, 共同好友, 共同粉丝等功能. 差集就可以实现好友推荐, 音乐推荐功能.

- Sorted Set: Sorted Set 多了一个权重参数 Score，集合中的元素能够按 Score 进行排列。可以做排行榜应用， 取 TOP(N) 操作. 也可以做优先级任务队列.
- geo: 地理位置计算
- bitmap: 可以用来做布隆过滤器, 或者统计签到次数等
- HyperLogLog: 统计页面UV
- Stream: 做一个简单的消息队列

string底层结构（中)

Redis没有直接使用c语言的字符串，而是自己构建了一种名为简单动态字符串SDS。

- C语言的字符串并不记录自身的长度信息，所以为了获取一个c字符串的长度，程序必须要遍历整个字符串，整个操作的复杂度为O(N). 而SDS中存储了len属性，所以只需要通过len即可直到Redis字符串的长度，时间复杂度为O(1)
- C语言的字符串不记录自身长度带来的另外一个问题是会带来缓冲区溢出. 而SDS不会发生缓冲区溢出, 当需要对SDS进行修改时，先检查SDS的空间是否满足修改所需的要求，如果不满足的话，自动将SDS的空间拓展至所需的大小，然后再执行实际的修改操作。
- C的字符串未记录自身的长度，C的字符串底层是一个N+1的字符数组(1是'\0')。每次增长或者缩短一个C的字符串，都要对数组进行一次内存重分配. 在SDS中，数组里面可以包含未使用的字节，未使用字节的数量用SDS的free属性记录. 通过未使用空间，SDS实现了空间预分配和惰性空间释放两种优化策略。
    - 空间预分配: 对一个SDS进行修改，并且需要对其进行拓展时，程序不仅会对SDS分配修改所需要的空间，还会为SDS分配额外的未使用空间。

        - 如果修改之后的长度小于1MB，那么程序分配和len一样的未使用空间。比如修改之后的SDS实际长度为13，那么也会分配13字节的未使用空间，SDS的buf数组的实际长度会变成13+13+1=27字节
        - 如果修改时候大于1MB，那么程序会额外为SDS分配1MB的未使用空间
    - 惰性空间释放: 当SDS的API需要缩短SDS保存的字符串时，程序并不立即使用内存重分配来回收缩短后多出来的字节，而是使用free属性将这些字节的数量记录起来，并等待将来使用
    - SDS默认会为buf分配一字节的空间来保存空字符串结尾的'\0'，所以SDS可以重用部分C的字符串函数
- SDS 不光能存放文本数据，而且能保存图片、音频、视频、压缩文件这样的二进制数据

Zset底层的数据结构？（中）

Zset类型的底层数据结构是由压缩列表或跳表实现的：

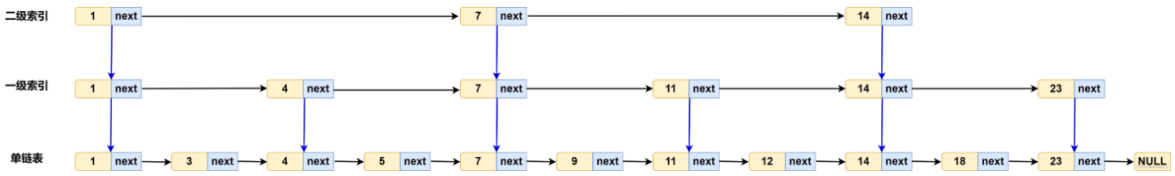
- 如果有序集合的元素个数小于128个，并且每个元素的值小于64字节时，Redis的Zset底层会使用压缩列表
- 如果有序集合的元素不满足上面的条件，Redis会使用跳表作为Zset类型的底层数据结构

介绍一下跳表（中）

链表在查找元素的时候，因为需要逐一查找，所以查询效率非常低，时间复杂度是O(N)

于是就出现了跳表。跳表是在链表基础上改进过来的，本质是一种多层的有序链表，可以快读定位数据

下图为一个二层跳表:



- 在链表中, 查找18需要遍历10个节点
- 跳表中, 只需要遍历6个节点（二级索引层遍历3个节点，一级索引层1个节点，原始链表2个节点）

本地缓存和Redis缓存的区别, 优缺点？（低）

本地缓存是指将数据存储在本地应用程序或服务器上，通常用于加速数据访问和提高响应速度。本地缓存
通常使用内存作为存储介质，利用内存的高速读写特性来提高数据访问速度。

优点：

- 访问速度快：本地缓存存储在本地内存中，访问速度非常快
- 减轻网络压力：本地缓存能够降低对远程服务器的访问次数，从而减轻网络压力
- 低延迟：本地缓存位于本地设备上，能够提供低延迟的访问速度，适用于对实时性要求较高的应用场景

缺点：

- 可扩展性有限：本地缓存的可扩展性受到硬件资源的限制，无法支持大规模的数据存储和访问

分布式缓存(Redis)是指将数据存储在多个分布式节点上，通过协同工作来提供高性能的数据访问服务。分布式缓存通常使用集群方式进行部署，利用多台服务器来分担数据存储和访问的压力

优点：

- 可扩展性强：分布式缓存的节点可以动态扩展，能够支持大规模的数据存储和访问需求。
- 易于维护：分布式缓存通常采用自动化管理方式，能够降低维护成本和管理的复杂性。

缺点：

- 访问速度相对较慢：相对于本地缓存，分布式缓存的访问速度相对较慢，因为数据需要从多个节点进行访问和协同. 但是这也比访问数据库要快的多
- 网络开销大：分布式缓存需要通过网络进行数据传输和协同操作，相对于本地缓存来说，网络开销较大

redis的常见应用场景/你平时会用redis做什么（中)

- 缓存：通过将热点数据存储在内存中，可以极大地提高访问速度，减轻数据库压力
- 排行榜：Redis的有序集合结构非常适合用于实现排行榜和排名系统，可以方便地进行数据排序和排名
- 分布式锁：Redis的特性可以用来实现分布式锁，确保多个进程或服务之间的数据操作的原子性和一致性
- 计数器: 由于Redis的原子操作和高性能，它非常适合用于实现计数器和统计数据的存储，如网站访问量统计、点赞数统计等
- 消息队列：Redis的发布订阅功能使其做为一个轻量级的消息队列，可以用来实现发布和订阅模式. (Redis高版本新增Stream数据结构, 可以直接作为一个轻量级MQ使用)

redis常见场景题

如何实现排行榜？(中)

Redis 的 `sorted set` 的可以用于各种排行榜的场景，比如直播间送礼物的排行榜、朋友圈的微信步数排行榜、王者荣耀中的段位排行榜、话题热度排行榜等等。

点赞排行榜的需求是按用户点赞顺序显示点赞前5个, 以blogId为key, 以当前时间戳为score, 以userId为val, 存入zset. 通过`zrange`获取点赞的前5个.

王者荣耀段位的排行榜, 就以rank为key, 以当前段位为score, 以userId为val, 存入zset. 通过 `zrevrange` 获取段位排行榜前几个, 通过 `zrevrank` 获取当前用户的段位排名. 段位升了就使用 `zadd` 重新添加score来修改score.

常用的一些 Redis 命令: `zrange` (从小到大排序) 、 `zrevrange` （从大到小排序）、 `zrank` (获取指定元素在集合中按score从小到大排名) `zrevrank` (获取指定元素在集合中按score从大到小的排名)

实现简单的抽奖系统怎么做？(中)

首先, 使用 `sadd k1 v1 v2 v3` 来添加元素

`SPOP key count`： 随机移除并获取指定集合中一个或多个元素，适合不允许重复中奖的场景。

`SRANDMEMBER key count`：随机获取指定集合中指定数量的元素，适合允许重复中奖的场景。

统计活跃用户怎么做? (中)

使用日期（精确到天）作为 key，然后用户 ID 为 offset，如果当日活跃过就设置为 1

```
# 2022年3月22活跃用户统计, id为111的用户活跃, id为222的用户活跃
setbit 20220322 111 1
setbit 20220322 222 1

# 2022年3月23活跃用户统计, id为111的用户活跃
setbit 20220323 111 1
```

```
# 统计2022年3月22的活跃用户
bitcount 20220322
```

```
# 统计2022年3月22和2022年3月23的总活跃人数 (两天都活跃)
bitop and desk1 20220322 20220323

# 因为bitop结果为10进制, 所以还需要bitcount一下才能获取具体活跃人数
bitcount desk1
```

```
# 统计2022年3月22和2022年3月23的在线活跃人数 (两天有任意一天活跃即可)
bitop or desk2 20220322 20220323

# 因为bitop结果为10进制, 所以还需要bitcount一下才能获取具体活跃人数
bitcount desk2
```

统计页面 UV 怎么做 (中)

将访问指定页面的每个用户 ID 添加到 `HyperLogLog` 中

```
# 以k1为键, 将v1, v2, v3插入HLL
pfadd k1 v1 v2 v3

pfadd page1:day1:uv userId1 userId2 userId3 ... userIdN
```

统计指定页面的 UV

```
# 统计k1对应的HLL中数据的数量, 不同统计重复插入的值, 因此它天然适合uv统计. (根据概率统计, 错误率在0.81%)
pfcount k1

pfcount page1:day1:uv
```

合并uv, 每天统计, 然后合并一个月的, 就能计算出一个月的uv总量

```
# 合并k1, k2, k3对应的HLL (每天统计, 然后合并一个月的, 就能计算出一个月的uv总量)
pfmerge k1 k2 k3

pfmerge page1:day1:uv page1:day2:uv page1:day3:uv
```

实现在线用户列表怎么做(按登录时间排序、可查询、踢人) (中)

zset首先可以满足排序问题，那元素（会话session）过期该如何维护呢?

由于zset的每个元素存在score，那么可以把登录时间戳作为score进行存储，顺序关系很自然就确定了.

> 如何维护过期元素呢?

既然score是登录时间戳，那么只要元素的score小于当前时间-会话有效时间，那么这些key就是过期的，删除这些元素来达到维护在线用户列表的目的

> 查询

不支持多端登录如何查询

- 不支持多端登录的方案相对简单，存储内容就按用户唯一标识存储即可，查询条件可以支持和用户唯一标识有关系的数据（即可以通过查询条件查询出对应的userId），这样将所有的查询转移到userId上，再从在线用户列表中获取是否存在，存在则返回，并获取登录用户的缓存（key为用户唯一标识），缓存不存在说明在线用户列表存在过期的脏数据，此时删除列表中的元素，返回即可，如果存在登录用户缓存，则封装用户列表需要的数据返回。

支持多端登录如何查询

- 支持多端登录，由于zset是不可重复的，那么在线用户列表的元素内容就不能只存一个用户唯一标识了，需要添加会话标识和用户唯一标识来一起存储，此时还需要再维护一个用户唯一标识和所有会话session的关系，这样就可以查询一个用户登录的所有在线列表信息了，该关系可以使用set存储，查询的时候，先查询到关系数据，在通过关系数据组装key，再去会话列表中查询，存在则返回，并获取登录用户的缓存（key为用户唯一标识），缓存不存在说明在线用户列表存在过期的脏数据，此时删

除列表中的元素和关系中的元素，返回即可，如果存在登录用户缓存，则封装用户列表需要的数据返回。

> 踢人

不支持多端登录如何踢人

- 不支多端登录的，踢除登录时，删除用户登录缓存和在线用户列表缓存元素即可

支持多端登录如何踢人

- 支持多端登录的，踢除登录时，除了删除用户登录缓存和在线用户列表缓存元素外，还需要删除用户唯一标识和session的关系集合中的元素。

线程模型

Redis 为什么设计成单线程的 (中)

- 多线程处理会涉及到锁，并且多线程处理会涉及到线程切换而消耗 CPU。采用单线程，避免了不必要的上下文切换和竞争条件
- 其次 CPU 不是 Redis 的瓶颈，Redis 的瓶颈最有可能是机器内存或者网络带宽。所以redis网络io操作采用了多线程.

Redis一定是单线程的吗 (中)

- Redis 单线程指的是执行命令是由一个主线程来完成的

- 但是Redis 在磁盘io(持久化操作)和网络io会使用多线程来处理，是因为这些任务的操作都是很耗时的，如果把这些任务都放在主线程来处理，那么 Redis 主线程就很容易发生阻塞，这样就无法处理后续的请求了

Redis 6.0 之后为什么引入了多线程？(中)

Redis 执行命令一直是单线程模型

因为Redis 的性能瓶颈有时会出现在网络 I/O 的处理上, 故在 Redis 6.0 版本之后，采用了多个 I/O 线程来处理网络请求，提高网络 I/O 的并行度

但是对于命令的执行，Redis 仍然使用单线程来处理

内存与持久化

redis的过期策略 (高)

每当我们对一个 key 设置了过期时间时，Redis 会把该 key 带上过期时间存储到一个**过期字典**（expires dict）中，也就是说「过期字典」保存了数据库中所有 key 的过期时间

过期策略: 定期删除+惰性删除

- 定期删除就是Redis默认每隔 100ms 就 随机抽取 一些设置了过期时间的key，检测这些key是否过期，如果过期了就将其删除
- 惰性删除是在你要获取某个key 的时候，redis会先去检测一下这个key是否已经过期，如果没有过期则返回给你，如果已经过期了，那么redis会删除这个key，不会返回给你

但是，仅仅通过给 key 设置过期时间还是有问题的。因为还是可能存在定期删除和惰性删除漏掉了很多过期 key 的情况。这样就导致大量过期 key 堆积在内存里，然后就 Out of memory 了。

怎么解决这个问题呢？Redis 内存淘汰机制

为什么Key过期不立即删除？（高）

在过期key比较多的情况下，删除过期key可能会占用相当一部分CPU时间，在内存不紧张但CPU时间
紧张的情况下，将CPU时间用于删除和当前任务无关的过期键上，会对服务器的响应时间和吞吐量造成影响

内存淘汰机制 (高)

```
# noeviction -> 当内存不够时，不淘汰任何数据，只是抛出异常 (一般不使用)

# volatile-lru -> 从设置过期时间的数据中，淘汰最近最长时间没有被使用的数据
# allkeys-lru -> 从所有数据中，淘汰最近最长时间没有被使用的数据

# volatile-lfu -> 从设置过期时间的数据中，淘汰一段时间内使用次数最少数据
# allkeys-lfu -> 从所有数据中，淘汰一段时间内使用次数最少的数据

# volatile-random -> 从设置过期时间的数据中，随机淘汰一批数据
# allkeys-random -> 从所有数据中随机淘汰一批数据

# volatile-ttl -> 淘汰快超时的数据
```

名词解释

- LRU: 最近最久未使用；LRU是淘汰最长时间没有被使用的数据。
- LFU: 最近最少使用；LFU是淘汰一段时间内，使用次数最少的数据。
- TTL: time to live, 过期时间

Redis持久化机制 (高)

rdb:

RDB: 按照一定的时间周期策略把内存的数据以快照的形式保存到硬盘的二进制文件。即Snapshot快照存储，对应产生的数据文件为dump.rdb.

Redis 提供了两个命令来生成 RDB 文件，分别是 save 和 bgsave，他们的区别就在于是否在「主线程」里执行：

- 执行了 save 命令，就会在主线程生成 RDB 文件，由于和执行操作命令在同一个线程，所以如果写入 RDB 文件的时间太长，会阻塞主线程；
- 执行了 bgsave 命令，会创建一个子进程来生成 RDB 文件，这样可以避免主线程的阻塞；执行 bgsave 过程中，Redis 依然可以继续处理操作命令的，也就是数据是能被修改的。写时复制技术（Copy-On-Write）

aof:

AOF: Redis会将每一个收到的写命令追加到文件最后，类似于MySQL的binlog。当Redis重启是会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。

- AOF保存的数据更加完整, 但是性能相比RDB稍差

在 Redis 的配置文件中存在三种不同的 AOF 持久化方式

```
appendfsync always     #每次有数据修改发生时都会写入AOF文件,这样会严重降低Redis的速度
appendfsync everysec   #每次写操作命令执行完后，先将命令写入到 AOF 文件的内核缓冲区，然后每隔一秒将缓冲区里的内容写回到硬盘
appendfsync no         #让操作系统决定何时进行同步
```

- 为了兼顾数据和写入性能，可以用 `appendfsync everysec` ，让 Redis 每秒同步一次 AOF 文件，对Redis性能影响不大.
- 即使出现系统崩溃，用户最多只会丢失一秒之内产生的数据

混合持久化

Redis 4.0 提出混合使用 AOF 日志和内存快照，也叫混合持久化。

- 当开启了混合持久化时，在 AOF 重写日志时，fork 出来的重写子进程会先将与主线程共享的内存数据以 RDB 方式写入到 AOF 文件，然后主线程处理的操作命令会被记录在重写缓冲区里，重写缓冲区里的增量命令会以 AOF 方式写入到 AOF 文件，写入完成后通知主进程将新的含有 RDB 格式和 AOF 格式的 AOF 文件替换旧的的 AOF 文件。
- 使用了混合持久化，AOF 文件的前半部分是 RDB 格式的全量数据，后半部分是 AOF 格式的增量数据。
- 好处在于，重启 Redis 加载数据的时候，由于前半部分是 RDB 内容，这样加载的时候速度会很快
- 加载完 RDB 的内容后，才会加载后半部分的 AOF 内容，这里的内容是 Redis 后台子进程重写 AOF 期间，主线程处理的操作命令，可以使得数据更少的丢失。

故一般都会用混合持久化方式.

缓存问题（常考）

缓存击穿 (高)

一个被高并发访问并且缓存重建业务较复杂的key突然失效了，无数的请求访问会在瞬间给数据库带来巨大的冲击。

- 互斥锁方案:
  - 如果redis中没查到数据, 缓存未命中. 则获取互斥锁(setnx), 获取失败则休眠重试, 获取成功则再次查询缓存是否存在(双重检测), 没查到则查询数据库进行缓存重建

    - 也就是说能拿到锁的线程去做缓存重建, 没拿到锁的线程阻塞等待. 待缓存重建完成, 没拿到锁的线程直接查redis即可.
    - 互斥锁注重一致性, 但是性能较差, 需要数据库与缓存强一致性选择互斥锁.
- 逻辑过期方案:
    - 需要在将要缓存的数据中加上时间属性作为逻辑过期时间, 然后在redis中设置永不过期. 查询时从redis中获取数据, 对比逻辑过期时间看否过期, 未过期直接返回数据. 过期则获取互斥锁. 获取锁失败则说明已经有人进行缓存重建了, 那么直接返回旧数据. 获取锁成功(需要双重检测)则开启新线程进行缓存重建, 原线程直接返回保证效率.
    - 也就是说, 拿到锁的线程去做缓存重建, 没拿到锁的线程返回旧数据(因为redis中没设置过期时间, 所以总能拿到旧数据).
    - 逻辑过期性能好, 但是一致性差, 需要高性能选择逻辑过期.

无论是哪种方案, 其目标都是保证某个key过期, 大量请求访问它, 但是只有一个请求能访问数据库, 重建缓存.

缓存穿透 (高)

请求的数据在数据库和缓存中都不存在, 那么请求就会直接访问数据库.

- 方案一: 可以缓存空对象.
    - 优点是实现简单, 维护方便
    - 缺点是空对象会带来额外的内存消耗, 有可能出现短暂的数据不一致问题, 当每次的key都不一样时失效
- 方案二: 使用布隆过滤器.
    - 优点是不会有多余key, 内存占用少
    - 缺点是实现复杂, 且有误判的可能

缓存雪崩 (高)

同一时间大量的key都过期了, 或者redis宕机了, 导致大量请求访问数据库.

- 如果是大量key同一时间过期, 则给key添加固定的过期时间的基础上, 再增加一个随机的过期时间.
- 如果是redis宕机, 那么就使用主从或者集群保证可用性。
- 万一真的发生服务雪崩, 应该做好服务限流降级熔断的处理.

缓存预热 (中)

缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。

这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

缓存读写策略 (高)

> Cache Aside（旁路缓存）策略

核心思想是应用程序直接操作缓存

- 读优先读缓存, 缓存中没有数据, 读数据库, 在由应用程序写缓存.
- 写优先更新数据库, 确保数据持久化后, 删除缓存.

我们业务中常见的Redis缓存方案就是旁路缓存.

> Read/Write Through（读穿 / 写穿）策略

Read/Write Through（读穿 / 写穿）策略原则是应用程序只和缓存交互，不再和数据库交互，而是由缓存和数据库交互，相当于更新数据库的操作由缓存自己代理了

Read Through 策略:

- 先查询缓存中数据是否存在，如果存在则直接返回
- 如果不存在，则由缓存组件负责从数据库查询数据，并将结果写入到缓存组件，最后缓存组件将数据返回给应用

Write Through 策略:

- 当有数据更新的时候，先查询要写入的数据在缓存中是否已经存在：
- 如果缓存中数据已经存在，则更新缓存中的数据，并且由缓存组件同步更新到数据库中，然后缓存组件告知应用程序更新完成。
- 如果缓存中数据不存在，直接更新数据库，然后返回；

> Write Back（写回）策略

Write Back（写回）策略在更新数据的时候，只更新缓存，同时将缓存数据设置为脏的，然后立马返回，并不会更新数据库。对于数据库的更新，会通过批量异步更新的方式进行

Write Back 策略特别适合写多的场景, 但是带来的问题是，数据不是强一致性的，而且会有数据丢失的风险.

Write Back 是计算机体系结构中的设计，比如 CPU 的缓存、操作系统中文件系统的缓存都采用了 Write Back（写回）策略.

- 电脑在突然断电之后，之前写入的文件会有部分丢失，就是因为 Page Cache 还没有来得及刷盘造成的.

缓存与数据库一致性的问题 (高)

(这是一个常考且比较复杂的问题, 这里只给出解决方案, 具体可以查看网上博客文章)

先更新数据库，后更新缓存（不推荐）

1.操作失败的情况：

如果数据库更新成功，缓存更新失败，那么此时数据库中是新值，缓存中是旧值. 出现不一致

2.高并发的情况：

假设现在有线程A和线程B同时要进行更新操作，那么可能会这样：

（1）线程A更新了数据库；

（2）线程B更新了数据库；

（3）线程B更新了缓存；

（4）线程A更新了缓存；

此时数据库是线程B更新的值, 缓存中是线程A的旧值. 出现不一致

先删除缓存，后更新数据库 (不推荐)

高并发的情况下会出问题

假设同时有一个请求A进行更新操作，另一个请求B进行查询操作。那么可能会这样:

（1）请求A进行写操作，删除缓存；

（2）请求B查询发现缓存不存在；

（3）请求B去数据库查询得到旧值；

（4）请求B将旧值写入缓存；

（5）请求A将新值写入数据库；

缓存和数据库不一致

延迟双删(一般推荐)

先删除缓存，再更新数据库，等过一段时间后再对缓存进行一次删除，比如 5s 之后.

读写分离场景, 可能产生问题如下

一个请求A进行更新操作，另一个请求B进行查询操作。

（1）请求A进行写操作，删除缓存；

（2）请求A将数据写入数据库了；

（3）请求B查询缓存发现，缓存没有值；

（4）请求B去从库查询，这时还没有完成主从同步，因此查询到的是旧值；

（5）请求B将旧值写入缓存；

（6）数据库完成主从同步，从库变为新值；

出现缓存不一致

延迟删除缓存的时间到底设置要多久才合适呢?

- 问题1：延迟时间要大于「主从复制」的延迟时间
- 问题2：延迟时间要大于线程 B 读取数据库 + 写入缓存的时间

但是，**这个时间在分布式和高并发场景下，其实是很难评估的。**

很多时候，我们都是凭借经验大致估算这个延迟时间，例如延迟 1-5s，只能尽可能地降低不一致的概率。

采用这种方案，也只是尽可能保证一致性而已，极端情况下，还是有可能发生不一致。

先更新数据库，后删除缓存（配合mq/监听binlog重试才推荐）

1.操作失败的情况：

如果数据库更新成功，缓存删除失败。用户读取数据时会先从缓存中读取，而缓存中存储的是旧的数据。出现不一致

2.高并发的情况：

一个请求A做查询操作，一个请求B做更新操作。

（1）缓存刚好失效；

（2）请求A查询数据库，得一个旧值；

（3）请求B将新值写入数据库；

（4）请求B删除缓存；

（5）请求A将查到的旧值写入缓存；

这种情况下确实也会产生脏数据，不过这种情况发生的概率是很小的。为什么呢？

- 要出现以上情况，步骤（3）的写数据库操作比步骤（2）的读数据库操作耗时更短，才有可能使得步骤（4）先于步骤（5）。但是写操作基本不会快于读操作, 所以出现不一致的概率很低.

那么如何解决由操作失败导致的不一致性呢？重试!

- **消息队列重试(推荐)**: 将删除缓存的操作放入mq中.
  - 更新数据库, 发mq消息, mq接受到消息后, 异步删除缓存. 删除失败可以重试. mq可以保证消息可靠性.
    - 消息队列保证可靠性：写到队列中的消息，成功消费之前不会丢失（重启项目也不担心）
    - 消息队列保证消息成功投递：下游从队列拉取消息，成功消费后才会删除消息，否则还会继续投递消息给消费者（符合重试的需求）
- **订阅binlog日志重试(推荐)**: 监听binlog来删除缓存.
  - 使用类似于阿里的 canal的中间件, 监听mysql binlog. 更新数据库后, 监听到binlog变化, 删除缓存.
    - 无需考虑写消息队列失败情况：只要写 MySQL 成功，Binlog 肯定会有
    - binlog订阅者（消费者）直接获取变更的数据，然后删除缓存。

总结: 想要保证数据库和缓存一致性，**推荐采用「先更新数据库，再删除缓存」方案，并配合「消息队列重试」或「监听binlog重试」的方式来做**, 但是即便是这样也不能完全保证一致性, 所以完全保证缓存和数据库一致性是很难的.

聊聊布隆过滤器原理 (低)

布隆过滤器由**初始值都为0的位图数组**和**N个哈希函数**组成.

当我们在写入数据库数据时，在布隆过滤器里的位图中做个标记，这样下次查询数据是否在数据库时，只需要查询布隆过滤器，如果查询到数据没有被标记，说明不在数据库中.

布隆过滤器会通过3个操作完成标记:

- 第一步，使用N个哈希函数分别对数据做哈希计算，得到N个哈希值;
- 第二步，将第一步得到的N个哈希值对位图数组的长度取模，得到每个哈希值在位图数组的对应位置。
- 第三步，将每个哈希值在位图数组的对应位置的值设置为1;

简单来说就是使用n个hash函数对数据做hash计算得到n个值, 然后将这些值映射到位图中. 当数据查询时, 只需要查询布隆过滤器看看位图中的数据是否为1.

注意:布隆过滤器有误判的可能.

分布式锁 (高)

基本的分布式锁实现

基于Redis的分布式锁实现思路:

- 利用setnx获取锁，并设置过期时间，保存线程标识
- 释放锁时先判断线程标识是否与自己一致，一致则删除锁
    - 判断锁是否为自己的然后再释放, 这个过程必须是原子性的否则有可能释放别人的锁, 这就需要lua脚本.

特性:

- 利用set nx满足互斥性
- 利用set ex保证故障时锁依然能释放，避免死锁，提高安全性

- 利用Redis集群保证高可用和高并发特性

释放错锁的问题

- 线程1获取锁后阻塞, 锁超时时间一到, 锁自动释放
- 线程2获取锁, 开始执行线程2的业务. 此时线程1执行完业务释放锁, 就会释放掉线程2的锁.
- 线程3又获取了锁, 执行业务. 此时线程2执行完毕又释放了线程3的锁.

因此, 释放锁之前需要判断锁是否是自己的.

`获取锁的值，判断锁是否是自己的，释放锁`, 这三步不具有原子性. 所以会产生如下问题

- 线程1执行完业务准备释放锁, 从redis中查到了锁的值, 判断锁是自己的. 此时, 线程1阻塞. 然后锁过期了.
- 线程2获取锁, 开始执行业务. 线程1又醒来开始释放锁, 此时就释放了线程2的锁.

所以释放锁的逻辑必须写在lua脚本中, 保证原子性.

如何实现可重入锁/redisson可重入锁原理

由于自己写的锁使用了string类型的 setnx, 只要方法1上锁, 方法1调用的方法二需要再上锁是无法做到的, 会直接返回false

而redisson的可重入锁, 使用了hash类型, key为lock, field为线程名, value为数字.

- 加锁
  - 方法1加锁后, value=1, 方法1调用方法2

    - 方法2加锁, redisson判断锁标识是否是自己(field字段是否为同一线程), 若是则value++, value为2
    - 同理, 方法3加锁, value=3
- 释放锁
    - 方法3执行完毕, 释放锁, value--, value=2
    - 方法2释放锁, value=1
    - 方法1释放锁, value=0, 此时删除redis的该数据, 锁完全释放.

所以使用hash结构, hsetnx就可以实现可重入锁

如何实现锁的可重试

redis的发布订阅机制实现等待、唤醒，获取锁失败的重试机制

- 先直接获取锁, 如果获取失败, 并不是直接重试, 因为现在立即重试大概率其他线程正在执行业务. 获取锁失败会先订阅一下, 然后等待.
- 获取锁成功的线程在释放锁时会发布一条消息.
- 当其他线程得到该消息时, 就会重新获取锁, 如果再次获取锁失败, 就会再次等待.
- 但是不是无限制的等待, 因为他会有一个等待时间, 超过该时间则不重试直接返回false

业务线程没执行完, 锁超时释放了怎么办/如何实现超时续约

超时续约：利用watchDog看门狗机制,每隔一段时间, 重置锁的超时时间

- 看门狗机制会创建一个守护线程, 当锁快到期但是业务线程没执行完时为锁增加时间 (续命).
- 当然看门狗也不会无限地增加超时时间, redisson有一个参数用来设置加锁的时间, 超过这个时间后锁便自动解开了，不会延长锁的有效期

锁的主从一致性问题

redis为了保证高可用, 使用主从架构. 一主多从

- 获取锁时, 往redis主节点setnx. 主节点加锁成功后宕机, 此时数据未同步到从节点.
- 某个从节点成为新主节点. 但是新主节点没有锁信息. 此时其他线程还可以加锁.
- 这就产生了主从架构锁丢失问题.

解决方案: redisson联锁机制, 使用多个独立的Redis主节点, 多主, 或者多主多从

- 获取锁时, 往每一个redis主节点都写入key. 即便其中一台redis宕机, 其他redis依旧有锁信息.
- 并且必须在所有节点都获取到锁, 才算获取锁成功.
- Redisson 分布式联锁 RedissonMultiLock 对象可以将多个 RLock 锁对象关联为一个联锁, 可以把一组锁当作一个锁来加锁和释放。

redlock红锁

联锁机制配合redis多主节点还存在问题:

- 联锁必须在所有节点都获取到锁, 才算获取锁成功. 那么当某个redis主节点网络较慢, 就导致加锁时间会很长, 很容易出现加锁超时失败的情况.
- 如果某个redis主节点宕机, 也会加锁失败.

而redlock规定, 只需要半数以上节点加锁成功, 就算这次加锁成功. 并且规定了严格的加锁时间, 一定时间内无法加锁成功, 则直接返回.

- 避免了某个redis主节点网络较慢, 就导致加锁时间会很长的问题
- 只需要半数以上节点加锁成功就可以, 避免了频繁加锁失败问题.

但是redlock对redis部署的要求很高, 需要多主部署, 或者部署多个独立的redis集群. 并且redlock复杂度和成本都很高, 所以其实redlock并不推荐使用. 哈哈哈哈看了这么久以为redlock是最牛逼的最终解决方案, 结果并不推荐使用. 别生气, 虽然它不推荐使用, 但是可以用来给面试官吹牛逼, 毕竟很多面试官对分布式锁的了解都没有深入到redlock, 懂了把.

高可用

（高可用并不是常考点，了解即可）

如何保证Redis服务高可用？ (中)

主从或者集群.

什么是主从复制 (中)

- 主从复制就是将一台Redis主节点的数据复制到其他的Redis从节点中，尽最大可能保证Redis主节点和从节点的数据是一致的。
- 主从复制是Redis高可用的基石，Redis Sentinel以及RedisCluster 都依赖于主从复制。

- 主从复制这种方案不仅保障了Redis服务的高可用，还实现了读写分离，提高了系统的并发量，尤其是读并发量。

讲一下Redis主从同步中的完全同步 (低)

完全同步发生在以下几种情况：

- 初次同步：当一个从服务器（slave)首次连接到主服务器（master)时，会进行一次完全同步。
- 从服务器数据丢失：如果从服务器数据由于某种原因（如断电）丢失，它会请求进行完全同步。
- 主服务器数据发生变化：如果从服务器长时间未与主服务器同步，导致数据差异太大，也可能触发完全同步

主从服务器间的第一次同步的过程可分为三个阶段：

- 第一阶段是建立链接、协商同步；
- 第二阶段是主服务器同步数据给从服务器；
- 第三阶段是主服务器发送新写操作命令给从服务器。

具体过程：

1. 从服务器发送SYNC命令：从服务器向主服务器发送SYNC命令，请求开始同步。
2. 主服务器生成RDB快照：接收到SYNC命令后，主服务器会保存当前数据集的状态到一个临时文件，这个过程称为RDB(Redis Database)快照。
3. 传输RDB文件：主服务器将生成的RDB文件发送给从服务器。
4. 从服务器接收并应用RDB文件：从服务器接收RDB文件后，会清空当前的数据集，并载入RDB文件中的数据

5. 主服务器记录写命令：在RDB文件生成和传输期间，主服务器会记录所有接收到的写命令到replication backlog buffer。
6. 传输写命令：一旦RDB文件传输完成，主服务器会将replication backlog buffer中的命令发送给从服务器，从服务器会执行这些命令，以保证数据的一致性。

讲一下Redis主从同步中的增量同步 (低)

增量同步允许从服务器从断点处继续同步，而不是每次都进行完全同步。它基于PSYNC命令，使用了运行ID(runID)和复制偏移量（offset)的概念

主要有三个步骤：

- 从服务器在恢复网络后，会发送psync命令给主服务器
- 主服务器收到该命令后，然后用CONTINUE响应命令告诉从服务器接下来采用增量复制的方式同步数据
- 然后主服务将主从服务器断线期间，所执行的写命令发送给从服务器，然后从服务器执行这些命令。

主服务器怎么知道要将哪些增量数据发送给从服务器呢？主要依赖两个字段

- repl_backlog_buffer: 一个环形缓冲区，用于主从服务器断连后，从中找到差异的数据
- replication offset: 标记上面那个缓冲区的同步进度，主从服务器都有各自的偏移量，主服务器使用master_repl_offset来记录自己写到的位置，从服务器使用slave_repl_offset来记录自己读到的位置

那repl_backlog_buffer缓冲区是什么时候写入的呢？

- 在主服务器进行命令传播时，不仅会将写命令发送给从服务器，还会将写命令写入到repl_backlog_buffer
  缓冲区里，因此这个缓冲区里会保存着最近传播的写命令。
- 网络断开后，当从服务器重新连上主服务器时，从服务器会通过psync命令将自己的复制偏移量slave_repl_offset发送给主服务器，主服务器根据自己的master_repl_offset和
  slave_repl_offset之间的差
  距，然后来决定对从服务器执行哪种同步操作：
  - 如果判断出从服务器要读取的数据还在repl_backlog_buffer缓冲区里，那么主服务器将采用增量同步的方式；
  - 相反，如果判断出从服务器要读取的数据已经不存在repl_backlog_buffer缓冲区里，那么主服务器将采用全量同步的方式。

当主服务器在repl_backlog_buffer中找到主从服务器差异（增量）的数据后，就会将增量的数据写入到replication buffer 缓冲区，这个缓冲区就是缓存将要传播给从节点的命令。

主从复制与Sentinel哨兵机制（中)

- 主从复制方案下，master发生启机的话可以手动将某一台slave升级为master, Redis服务可用性提高。
- slave可以分担读请求，读吞吐量大幅提高。
- 但其缺陷也很明显，一旦master容机，我们需要从slave中手动选择一个新的master, 同时需要修改应用方的主节点地址，还需要命令所有从节点去复制新的主节点，整个过程需要手动处理, 很麻烦, 很容易出问题.
- redis的Sentinel哨兵机制可以自动帮选主.

Sentinel(哨兵）有什么作用？（中)

- 监控Redis节点的运行状态并自动实现故障转移。
- 当master节点出现故障的时候，Sentinel会自动根据一定的规则选出一个slave升级为master,确保整个Redis系统的可用性。整个过程完全自动，不需要人工介入

聊聊哨兵机制的选主的算法 (低)

当redis集群的主节点故障时，Sentinel集群将从剩余的从节点中选举一个新的主节点，步骤如下：

1. **故障节点主观下线**
    - Sentinel集群的每一个节点会定时对redis集群的所有节点发心跳包检测节点是否正常
    - 如果一个节点在一定时间内没有回复心跳包，则该redis节点被该Sentinel节点认为主观下线
2. **故障节点客观下线**
    - 当节点被一个Sentinel节点记为主观下线时, 并不意味着该节点肯定故障了, 可能是该Sentinel节点自身出现了网络分区, 故还需要Sentine集群的其他Sentinel节点共同判断为主观下线才行
    - 该Sentinel节点会询问其他Sentinel节点，如果Sentine集群中超过一定数量的Sentinel节点认为该redis
      节点主观下线，则该redis客观下线
    - 如果客观下线的redis节点是从节点或者是Sentinel节点，则操作到此为止，没有后续的操作了
    - 如果客观下线的redis节点为主节点，则开始故障转移，从从节点中选举一个节点升级为主节点

3. **Sentinel集群选举Leader**
    - 如果需要从redis集群选举一个节点为主节点，首先需要从Sentine集群中选举一个Sentinel节点作为Leader . 这会使用到Raft算法.
    - 简单讲一下Raft算法
        - 每一个Sentinel节点都可以成为Leader, 当一个Sentinel节点确认redis集群的主节点主观下线后, 会请求其他Sentinel节点要求将自己选举为Leader
        - 被请求的Sentinel节点如果没有同意过其他Sentinel节点的选举请求，则同意该请求（选举票数+1),否则不同意
        - 如果一个Sentinel节点获得半数以上的选举票数, 则该Sentinel节点选举为Leader;否则重新进行选举
4. **Sentinel Leader决定新主节点**
    - 当Sentine集群选举出Leader后，由Sentinel Leader从redis从节点中选择一个redis节点作主节点
    - 选举依据如下
        - 过滤故障的节点
        - 选择优先级slave-priority最大的从节点作为主节点，如不存在则继续
        - 选择复制偏移量（数据写入量的字节，记录写了多少数据。主服务器会把偏移量同步给从服务器，当主从的偏移量一致，则数据是完全同步）最大的从节点作为主节点，如不存在则继续
        - 选择runid(redis每次启动的时候生成随机的runid作为redis的标识)最小的从节点作为主节点

Redis缓存的数据量太大怎么办？(中)

分片集群，数据被分散到集群不同的redis节点上，避免了单个redis缓存的数据量过大问题。

Redis Cluster的作用? (中)

高并发场景下，使用Redis主要会遇到的两个问题：

- 缓存的数据量太大：实际缓存的数据量可以达到几十G,甚至是成百上千G;
- 并发量要求太大：Redis号称单机可以支持10w并发, 但是并发超过10w怎么办?

主从复制和Redis Sentinel这两种方案本质都是通过增加slave数量的方式来提高Redis服务的整体可用性和读吞吐量，不支持横向扩展来缓解写压力以及解决缓存数据量过大的问题。

这时候, redis分片集群就可以解决这个问题. 通过部署多台Redis主节点（master), 然后将key分散到不同的主节点上, 客户端的请求通过路由规则转发到目标master上, 这就是分片集群.

注意, 这些节点之间平等，并没有主从之说，同时对外提供读/写服务。

分片集群很方便拓展, 只需要增加redis主节点即可.

Redis Cluster 是如何分片的?/Redis Cluster中的数据是如何分布的?/ 怎么知道key应该在哪个哈希槽中?(中)

- Redis Cluster并没有使用一致性哈希，采用的是哈希槽分区，每一个键值对都属于一个哈希槽
- Redis Cluster通常有16384个哈希槽，要计算给定key应该分布到哪个哈希槽中，我们只需要先对每个key计算CRC-16(XMODEM)校验码，然后再对这个校验码对16384(哈希槽的总数）取模，得到的值即是key对应的哈希槽