什么是Redis?

Redis 是一种基于内存的数据库，对数据的读写操作都是在内存中完成，因此**读写速度非常快**，常用于**缓存，消息队列、分布式锁等场景。**Redis 提供了多种数据类型来支持不同的业务场景，比如 String(字符串)、Hash(哈希)、 List (列表)、Set(集合)、Zset(有序集合)、Bitmaps（位图）、HyperLogLog（基数统计）、GEO（地理信息）、Stream（流），并且对数据类型的操作都是**原子性**的，因为执行命令由单线程负责的，不存在并发竞争的问题。

Redis和Memcached有什么区别?

- 共同点
  - 都是基于内存的数据库，一般都用来当作缓存使用。
  - 都有过期的策略。
  - 两者的性能都非常高。
- 区别：
  - Redis支持的数据类型更为丰富，而Memcached只支持简单的key-value数据类型。
  - Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用，而 Memcached 没有持久化功能，数据全部存在内存之中，Memcached 重启或者挂掉后，数据就没了；
  - Redis 原生支持集群模式，Memcached 没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据；
  - Redis 支持发布订阅模型、Lua 脚本、事务等功能，而 Memcached 不支持；

为什么用Redis作为MySQL的缓存?

- Redis具有高性能。用户第一次访问 MySQL 中的某些数据。这个过程会比较慢，因为是从硬盘上读取的。将该用户访问的数据缓存在 Redis 中，这样下一次再访问这些数据的时候就可以直接从缓存中获取了，操作 Redis 缓存就是直接操作内存，所以速度相当快。
- Redis具有高并发。直接访问 Redis 能够承受的请求是远远大于直接访问 MySQL 的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库。

Redis的数据类型以及使用场景是什么?

结构类型	结构存储的值	结构的读写能力
String 字符串	可以是字符串、整数或浮点数	对整个字符串或字符串的一部分进行操作；对整数或浮点数进行自增或自减操作；
List列表	一个链表，链表上的每个节点都包含一个字符串	对链表的两端进行push和pop操作，读取单个或多个元素；根据值查找或删除元素；
Set集合	包含字符串的无序集合	字符串的集合，包含基础的方法有看是否存在添加、获取、删除；还包含计算交集、并集、差集等
Hash散列	包含键值对的无序散列表	包含方法有添加、获取、删除单个元素
Zset有序集合	和散列一样，用于存储键值对	字符串成员与浮点数分数之间的有序映射；元素的排列顺序由分数的大小决定；包含方法有添加、获取、删除单个元素以及根据分值范围或成员来获取元素

- String 类型的应用场景：缓存对象、常规计数、分布式锁、共享 session 信息等。

- List 类型的应用场景：消息队列（但是有两个问题：1. 生产者需要自行实现全局唯一 ID；2. 不能以消费组形式消费数据）等。
- Hash 类型：缓存对象、购物车等。
- Set 类型：聚合计算（并集、交集、差集）场景，比如点赞、共同关注、抽奖活动等。
- Zset 类型：排序场景，比如排行榜、电话和姓名排序等。

五种常见的Redis数据类型是怎么实现的?

- String类型的内部实现
  - SDS 不仅可以保存文本数据，还可以保存二进制数据。
  - SDS 获取字符串长度的时间复杂度是 O(1)。
  - Redis 的 SDS API 是安全的，拼接字符串不会造成缓冲区溢出。
- List类型内部实现
  - List 数据类型底层数据结构由 quicklist 实现了，替代了双向链表和压缩列表。
- Hash 类型内部实现
  - 如果哈希类型元素个数小于 512 个，所有值小于 64 字节的话，Redis 会使用**压缩列表**作为 Hash 类型的底层数据结构；**压缩列表数据结构已经废弃了，交由 listpack 数据结构来实现了。**
  - 如果哈希类型元素不满足上面条件，Redis 会使用**哈希表**作为 Hash 类型的底层数据结构。
- Set 类型内部实现
  - 如果集合中的元素都是整数且元素个数小于 512 个，Redis 会使用**整数集合**作为 Set 类型的底层数据结构；
  - 如果集合中的元素不满足上面条件，则 Redis 使用**哈希表**作为 Set 类型的底层数据结构。
- ZSet 类型内部实现
  - 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 会使用**压缩列表**作为 Zset 类型的底层数据结构；**压缩列表数据结构已经废弃了，交由 listpack 数据结构来实现了。**
  - 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；

Redis是单线程吗?

- Redis 单线程指的是**「接收客户端请求->解析请求 ->进行数据读写等操作->发送数据给客户端」**这个过程是由一个线程（主线程）来完成的。
- 但是，**Redis 程序并不是单线程的**，Redis 在启动的时候，是会**启动后台线程**的，2 个后台线程，分别处理关闭文件、AOF 刷盘这两个任务；1个新的后台线程，用来异步释放 Redis 内存。之所以 Redis 为「关闭文件、AOF 刷盘、释放内存」这些任务创建单独的线程来处理，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采用单线程为什么还那么快?

- Redis 的大部分操作**都在内存中完成**，并且采用了高效的数据结构。因此 Redis 瓶颈可能是机器的内存或者网络带宽，而并非 CPU，既然 CPU 不是瓶颈，那么自然就采用单线程的解决方案了；
- Redis 采用单线程模型可以**避免了多线程之间的竞争**，省去了多线程切换带来的时间和性能上的开销，而且也不会导致死锁问题。
- Redis 采用了 **I/O 多路复用机制**处理大量的客户端 Socket 请求，简单来说，在 Redis 只运行单线程的情况下，该机制允许内核中，同时存在多个监听 Socket 和已连接 Socket。内核会一直监听这些 Socket 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，这就实现了一个 Redis 线程处理多个 IO 流的效果。

Redis 6.0之前为什么使用单线程?

- **CPU 并不是制约 Redis 性能表现的瓶颈所在**，更多情况下是受到内存大小和网络I/O的限制，所以 Redis 核心网络模型使用单线程并没有什么问题
- 使用了单线程后，可维护性高，多线程模型虽然在某些方面表现优异，但是它却引入了程序执行顺序的不确定性，带来了并发读写的一系列问题，**增加了系统复杂度、同时可能存在线程切换、甚至加锁解锁、死锁造成的性能损耗。**

Redis 6.0之后为什么引入了多线程?

- 在 Redis 6.0 版本之后，也**采用了多个 I/O 线程来处理网络请求**，**这是因为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 I/O 的处理上。**但是对于命令的执行，Redis 仍然使用单线程来处理**

Redis 6.0 版本之后，Redis 在启动的时候，默认情况下会**额外创建 6 个线程**

- Redis-server： Redis的主线程，主要负责执行命令；
- bio_close_file、bio_aof_fsync、bio_lazy_free：三个后台线程，分别异步处理关闭文件任务、AOF刷盘任务、释放内存任务；
- io_thd_1、io_thd_2、io_thd_3：三个 I/O 线程，I/O 多线程，用来分担 Redis 网络 I/O 的压力。

Redis如何实现数据不丢失?

为了保证内存中的数据不会丢失，Redis 实现了数据持久化的机制，这个机制会把数据存储到磁盘，这样在 Redis 重启就能够从磁盘中恢复原有的数据。

三种数据持久化的方式：

- **AOF 日志**：每执行一条写操作命令，就把该命令以追加的方式写入到一个文件里；
- **RDB 快照**：将某一时刻的内存数据，以二进制的方式写入磁盘；
- **混合持久化方式**：Redis 4.0 新增的方式，集成了 AOF 和 RBD 的优点；

AOF日志是如何实现的?

Redis 在执行完一条写操作命令后，就会把该命令以追加的方式写入到一个文件里，然后 Redis 重启时，会读取该文件记录的命令，然后逐一执行命令的方式来进行数据恢复。

为什么先执行命令，再把数据写入日志呢?

- **避免额外的检查开销**：因为如果先将写操作命令记录到 AOF 日志里，再执行该命令的话，如果当前的命令语法有问题，那么如果不进行命令语法检查，该错误的命令记录到 AOF 日志里后，Redis 在使用日志恢复数据时，就可能会出错。
- **不会阻塞当前写操作命令的执行**：因为当写操作命令执行成功后，才会将命令记录到 AOF 日志。

AOF 写回策略有几种?

- **Always**，这个单词的意思是「总是」，所以它的意思是每次写操作命令执行完后，同步将 AOF 日志数据写回硬盘；
- **Everysec**，这个单词的意思是「每秒」，所以它的意思是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；
- **No**，意味着不由 Redis 控制写回硬盘的时机，转交给操作系统控制写回的时机，也就是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，再由操作系统决定何时将缓冲区内容写回硬盘。

AOF 日志过大，会触发什么机制?

AOF 日志是一个文件，随着执行的写操作命令越来越多，文件的大小会越来越大。 如果当 AOF 日志文件过大就会带来性能问题，比如重启 Redis 后，需要读 AOF 文件的内容以恢复数据，如果文件过大，整个恢复的过程就会很慢。

AOF 重写机制是在重写时，读取当前数据库中的所有键值对，然后将每一个键值对用一条命令记录到「新的 AOF 文件」，等到全部记录完后，就将新的 AOF 文件替换掉现有的 AOF 文件。

AOF重写期间，主进程修改了已存在的key-value对，出现的这种数据不一致问题如何解决?

Redis 设置了一个 **AOF 重写缓冲区**，这个缓冲区在创建 bgrewriteaof 子进程之后开始使用。

在重写 AOF 期间，当 Redis 执行完一个写命令之后，它会**同时将这个写命令写入到 「AOF 缓冲区」和「AOF 重写缓冲区」。**

当子进程完成 AOF 重写工作后，会向主进程发送一条信号，主进程收到该信号后，会调用一个信号处理函数，该函数主要做以下工作：

- 将 AOF 重写缓冲区中的所有内容追加到新的 AOF 的文件中，使得新旧两个 AOF 文件所保存的数据库状态一致；
- 新的 AOF 的文件进行改名，覆盖现有的 AOF 文件。

RDB快照是如何实现的呢?

RDB 快照就是记录某一个瞬间的内存数据，记录的是实际数据，而 AOF 文件记录的是命令操作的日志，而不是实际的数据。因此在 Redis 恢复数据时， RDB 恢复数据的效率会比 AOF 高些，因为直接将 RDB 文件读入内存就可以，不需要像 AOF 那样还需要额外执行操作命令的步骤才能恢复数据。

RDB做快照时会阻塞线程吗?

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**；

RDB在执行快照的时候，数据能修改吗?

可以的，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，关键的技术就在于**写时复制技术（Copy-On-Write, COW）。**执行 bgsave 命令的时候，会通过 fork() 创建子进程，此时子进程和父进程是共享同一片内存数据的，因为创建子进程的时候，会复制父进程的页表，但是页表指向的物理内存还是一个，此时如果主线程执行读操作，则主线程和 bgsave 子进程互相不影响。如果主线程执行写操作，则被修改的数据会复制一份副本，然后 bgsave 子进程会把该副本数据写入 RDB 文件，在这个过程中，主线程仍然可以直接修改原来的数据。

为什么会有混合持久化?

- RDB 优点是数据恢复速度快，但是快照的频率不好把握。频率太低，丢失的数据就会比较多，频率太高，就会影响性能。AOF 优点是丢失数据少，但是数据恢复不快。混合持久化既保证了 Redis 重启速度，又降低数据丢失风险。

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。**

优点

- 混合持久化结合了 RDB 和 AOF 持久化的优点，开头为 RDB 的格式，使得 Redis 可以更快的启动，同时结合 AOF 的优点，有减低了大量数据丢失的风险。

缺点

- AOF 文件中添加了 RDB 格式的内容，使得 AOF 文件的可读性变得很差；
- 兼容性差，如果开启混合持久化，那么此混合持久化 AOF 文件，就不能用在 Redis 4.0 之前版本了。

如何实现服务高可用?

要想设计一个高可用的 Redis 服务，一定要从 Redis 的多服务节点来考虑，比如 Redis 的主从复制、哨兵模式、切片集群。

解释一下脑裂现象，如何解决有其导致的数据丢失问题呢?

由于网络问题，集群节点之间失去联系。主从数据不同步；重新平衡选举，产生两个主服务。等网络恢复，旧主节点会降级为从节点，再与新主节点进行同步复制的时候，由于会从节点会清空自己的缓冲区，所以导致之前客户端写入的数据丢失了。

解决方案：当主节点发现从节点连接的总数量小于阈值或者通信超时，那么禁止主节点进行写数据，直接把错误返回给客户端。

过期删除策略有哪些?

- 定时删除：**在设置 key 的过期时间时，同时创建一个定时事件，当时间到达时，由事件处理器自动执行 key 的删除操作。**
  - 优点
    - 可以保证过期 key 会被尽快删除，也就是内存可以被尽快地释放。因此，定时删除对内存是最友好的。
  - 缺点
    - 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，无疑会对服务器的响应时间和吞吐量造成影响。所以，定时删除策略对 CPU 不友好。
- 惰性删除：**不主动删除过期键，每次从数据库访问 key 时，都检测 key 是否过期，如果过期则删除该 key。**
  - 优点
    - 因为每次访问时，才会检查 key 是否过期，所以此策略只会使用很少的系统资源，因此，惰性删除策略对 CPU 时间最友好。
  - 缺点
    - 如果一个 key 已经过期，而这个 key 又仍然保留在数据库中，那么只要这个过期 key 一直没有被访问，它所占用的内存就不会释放，造成了一定的内存空间浪费。所以，惰性删除策略对内存不友好。
- 定期删除：**每隔一段时间「随机」从数据库中取出一定数量的 key 进行检查，并删除其中的过期 key。**
  - 优点

        - 通过限制删除操作执行的时长和频率，来减少删除操作对 CPU 的影响，同时也能删除一部分过期的数据减少了过期键对空间的无效占用。
    - 缺点
        - 内存清理方面没有定时删除效果好，同时没有惰性删除使用的系统资源少。
        - 难以确定删除操作执行的时长和频率。如果执行的太频繁，定期删除策略变得和定时删除策略一样，对CPU不友好；如果执行的太少，那又和惰性删除一样了，过期 key 占用的内存不会及时得到释放。

Redis使用的过期删除策略是什么?

Redis 使用的过期删除策略是「**惰性删除+定期删除**」这两种策略配和使用。以求在合理使用 CPU 时间和避免内存浪费之间取得平衡。

Redis如何实现惰性删除的?

- Redis 在访问或者修改 key 之前，都会调用 expireIfNeeded 函数对其进行检查，检查 key 是否过期
- 如果过期，则删除该 key，然后返回 null 客户端；
- 如果没有过期，不做任何处理，然后返回正常的键值对给客户端；

Redis如何实现定期删除的?

- 从过期字典中随机抽取 20 个 key；
- 检查这 20 个 key 是否过期，并删除已过期的 key；
- 如果本轮检查的已过期 key 的数量，超过 5 个（20/4），也就是「已过期 key 的数量」占比「随机抽取 key 的数量」大于 25%，则继续重复步骤 1；如果已过期的 key 比例小于 25%，则停止继续删除过期 key，然后等待下一轮再检查。

Redis 为了保证定期删除不会出现循环过度，导致线程卡死现象，为此增加了定期删除循环流程的时间上限，默认不会超过 25ms。

Redis持久化时，对过期键会如何处理?

对于RDB文件来说，分为生成阶段和加载阶段

- **RDB 文件生成阶段**：从内存状态持久化成 RDB（文件）的时候，会对 key 进行过期检查，**过期的键「不会」被保存到新的 RDB 文件中**，因此 Redis 中的过期键不会对生成新 RDB 文件产生任何影响。
- **RDB 加载阶段**
    - **如果 Redis 是「主服务器」运行模式的话，在载入 RDB 文件时，程序会对文件中保存的键进行检查，过期键「不会」被载入到数据库中。**
    - **如果 Redis 是「从服务器」运行模式的话，在载入 RDB 文件时，不论键是否过期都会被载入到数据库中。**但由于主从服务器在进行数据同步时，从服务器的数据会被清空。

对于AOF 文件来说，分为两个阶段：AOF 文件写入阶段和 AOF 重写阶段。

- **AOF 文件写入阶段**：当 Redis 以 AOF 模式持久化时，**如果数据库某个过期键还没被删除，那么 AOF 文件会保留此过期键，当此过期键被删除后，Redis 会向 AOF 文件追加一条 DEL 命令来显式地删除该键值。**
- **AOF 重写阶段**：执行 AOF 重写时，会对 Redis 中的键值对进行检查，**已过期的键不会被保存到重写后的 AOF 文件中**，因此不会对 AOF 重写造成任何影响。

Redis主从模式中，会对过期键如何处理?

从库不会进行过期扫描，从库对过期的处理是被动的。也就是即使从库中的 key 过期了，如果有客户端访问从库时，依然可以得到 key 对应的值，像未过期的键值对一样返回。从库的过期键处理依靠主服务器控制，**主库在 key 到期时，会在 AOF 文件里增加一条 del 指令，同步到所有的从库**，从库通过执行这条 del 指令来删除过期的 key。

Redis内存满了，会发生什么?

在 Redis 的运行内存达到了某个阀值，就会触发**内存淘汰机制**，这个阀值就是我们设置的最大运行内存，

Redis内存淘汰策略有哪些?

- 不进行数据淘汰的策略
  - **noeviction**：它表示当运行内存超过最大设置内存时，不淘汰任何数据，而是不再提供服务，直接返回错误。
- 进行数据淘汰的策略
  - 在设置了过期时间的数据中进行淘汰
    - **volatile-random**：随机淘汰设置了过期时间的任意键值；
    - **volatile-ttl**：优先淘汰更早过期的键值。
    - **volatile-lru**：淘汰所有设置了过期时间的键值中，最久未使用的键值；
    - **volatile-lfu**：淘汰所有设置了过期时间的键值中，最少使用的键值；
  - 在所有数据范围内进行淘汰
    - **allkeys-random**：随机淘汰任意键值;
    - **allkeys-lru**：淘汰整个键值中最久未使用的键值；
    - **allkeys-lfu**：淘汰整个键值中最少使用的键值。

Redis如何实现LRU算法?

Redis 实现的是一种**近似 LRU 算法**，目的是为了更好的节约内存，它的**实现方式是在 Redis 的对象结构体中添加一个额外的字段，用于记录此数据的最后一次访问时间。**

当 Redis 进行内存淘汰时，会使用**随机采样的方式来淘汰数据**，然后**淘汰最久没有使用的那个。**

Redis 实现的 LRU 算法的优点：

- 不用为所有的数据维护一个大链表，节省了空间占用；
- 不用在每次数据访问时都移动链表项，提升了缓存的性能；

缺点：**无法解决缓存污染问题**，比如应用一次读取了大量的数据，而这些数据只会被读取这一次，那么这些数据会留存在 Redis 缓存中很长一段时间，造成缓存污染。

Redis如何实现LFU算法?

LFU 算法相比于 LRU 算法的实现，多记录了「数据的访问频次」的信息

LFU 算法会记录每个数据的访问次数。当一个数据被再次访问时，就会增加该数据的访问次数。这样就解决了偶尔被访问一次之后，数据留存在缓存中很长一段时间的问题，相比于 LRU 算法也更合理一些。

缓存异常	产生原因	应对方案
缓存雪崩	大量数据同时过期	- 均匀设置过期时间，避免同一时间过期 - 互斥锁，保证同一时间只有一个应用在构建缓存 - 双 key 策略，主 key 设置过期时间，备 key 永久，主 key 过期时，返回备 key 的内容 - 后台更新缓存，定时更新、消息队列通知更新
	Redis 故障宕机	- 服务熔断 - 请求限流 - 构建 Redis 缓存高可靠集群
缓存击穿	频繁访问的热点数据过期	- 互斥锁 - 不给热点数据设置过期时间，由后台更新缓存
缓存穿透	访问的数据既不在缓存，也不在数据库	- 非法请求的限制； - 缓存空值或者默认值； - 使用布隆过滤器快速判断数据是否存在；

如何避免缓存雪崩？

当大量缓存数据在同一时间过期（失效）时，如果此时有大量的用户请求，都无法在 Redis 中处理，于是全部请求都直接访问数据库，从而导致数据库的压力骤增，严重的会造成数据库宕机，从而形成一系列连锁反应，造成整个系统崩溃，这就是**缓存雪崩**的问题。

- **将缓存失效时间随机打散**： 我们可以在原有的失效时间基础上增加一个随机值（比如 1 到 10 分钟）这样每个缓存的过期时间都不重复了，也就降低了缓存集体失效的概率。
- **设置缓存不过期**： 我们可以通过后台服务来更新缓存数据，从而避免因为缓存失效造成的缓存雪崩，也可以在一定程度上避免缓存并发问题。

如何避免缓存击穿？

如果缓存中的**某个热点数据过期**了，此时大量的请求访问了该热点数据，就无法从缓存中读取，直接访问数据库，数据库很容易就被高并发的请求冲垮，这就是**缓存击穿**的问题。

- 互斥锁方案：保证同一时间只有一个业务线程请求缓存，未能获取互斥锁的请求，要么等待锁释放后重新读取缓存，要么就返回空值或者默认值。
- 不给热点数据设置过期时间，由后台异步更新缓存，或者在热点数据准备要过期前，提前通知后台线程更新缓存以及重新设置过期时间；

如何避免缓存穿透？

当用户访问的数据，**既不在缓存中，也不在数据库中**，导致请求在访问缓存时，发现缓存缺失，再去访问数据库时，发现数据库中也没有要访问的数据，没办法构建缓存数据，来服务后续的请求。那么当有大量这样的请求到来时，数据库的压力骤增，这就是**缓存穿透**的问题。

- **非法请求的限制**：当有大量恶意请求访问不存在的数据的时候，也会发生缓存穿透，因此在 API 入口处我们要判断求请求参数是否合理，请求参数是否含有非法值、请求字段是否存在，如果判断出是恶意请求就直接返回错误，避免进一步访问缓存和数据库。
- **设置空值或者默认值**：当我们线上业务发现缓存穿透的现象时，可以针对查询的数据，在缓存中设置一个空值或者默认值，这样后续请求就可以从缓存中读取到空值或者默认值，返回给应用，而不会继续查询数据库。
- **使用布隆过滤器快速判断数据是否存在，避免通过查询数据库来判断数据是否存在**：我们可以在写入数据库数据时，使用布隆过滤器做个标记，然后在用户请求到来时，业务线程确认缓存失效后，

可以通过查询布隆过滤器快速判断数据是否存在，如果不存在，就不用通过查询数据库来判断数据是否存在，即使发生了缓存穿透，大量请求只会查询 Redis 和布隆过滤器，而不会查询数据库，保证了数据库能正常运行，Redis 自身也是支持布隆过滤器的。

如何设计一个缓存策略，可以动态缓存热点数据呢?

热点数据动态缓存的策略总体思路：**通过数据最新访问时间来做排名，并过滤掉不常访问的数据，只留下经常访问的数据。**

说说常见的缓存更新策略

实际开发中，Redis 和 MySQL 的更新策略用的是 Cache Aside，另外两种策略应用不了。

- Cache Aside（旁路缓存）策略：应用程序直接与「数据库、缓存」交互，并负责对缓存的维护，该策略又可以细分为「读策略」和「写策略」。
    - 写策略：先更新数据库中的数据，再删除缓存中的数据。
    - 读策略：如果读取的数据命中了缓存，则直接返回数据；如果读取的数据没有命中缓存，则从数据库中读取数据，然后将数据写入到缓存，并且返回给用户。

  写策略的步骤的顺序不能倒过来，即**不能先删除缓存再更新数据库**，原因是在「读+写」并发的时候，会出现缓存和数据库的数据不一致性的问题。
- 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
    - 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：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。

Redis如何实现延迟队列?

延迟队列是指把当前要做的事情，往后推迟一段时间再做。

- 在 Redis 可以使用有序集合（ZSet）的方式来实现延迟消息队列的，ZSet 有一个 Score 属性可以用来存储延迟执行的时间。
- 使用 zadd score1 value1 命令就可以一直往内存中生产消息。再利用 zrangebysocre 查询符合条件的所有待处理的任务， 通过循环执行队列任务即可。

Redis管道有什么用?

管道技术（Pipeline）是客户端提供的一种批处理技术，用于一次处理多个 Redis 命令，从而提高整个交互的性能。使用**管道技术可以解决多个命令执行时的网络等待**，它是把多个命令整合到一起发送给服务器端处理之后统一返回给客户端，这样就免去了每条命令执行后都要等待的情况，从而有效地提高了程序的执行效率。

Redis事务支持回滚吗?

Redis 中并没有提供回滚机制

如何用Redis实现分布式锁?

通过使用 SET 命令和 Lua 脚本在 Redis 单节点上完成了分布式锁的加锁和解锁。

优点

- 性能高效
- 实现方便
- 避免单点故障

缺点

- **超时时间不好设置。**如果锁的超时时间设置过长，会影响性能，如果设置的超时时间过短会保护不到共享资源。
    - 解决方案：写一个守护线程，然后去判断锁的情况，当锁快失效的时候，再次进行续约加锁，当主线程执行完成后，销毁续约锁即可，不过这种方式实现起来相对复杂。
- **Redis 主从复制模式中的数据是异步复制的，这样导致分布式锁的不可靠性。**

Redis 大 Key 对持久化有什么影响?

- 当 AOF 写回策略配置了 Always 策略，如果写入是一个大 Key，主线程在执行 fsync() 函数的时候，阻塞的时间会比较久，因为当写入的数据量很大的时候，数据同步到硬盘这个过程是很耗时的。
- AOF 重写机制和 RDB 快照（bgsave 命令）的过程，都会分别通过 `fork()` 函数创建一个子进程来处理任务。会有两个阶段会导致阻塞父进程（主线程）
    - 创建子进程的途中，由于要复制父进程的页表等数据结构，阻塞的时间跟页表的大小有关，页表越大，阻塞的时间也越长；
    - 创建完子进程后，如果父进程修改了共享数据中的大 Key，就会发生写时复制，这期间会拷贝物理内存，由于大 Key 占用的物理内存会很大，那么在复制物理内存这一过程，就会比较耗时，所以有可能会阻塞父进程。

大 Key 还有哪些影响?

- 客户端超时阻塞。由于 Redis 执行命令是单线程处理，然后在操作大 key 时会比较耗时，那么就会阻塞 Redis，从客户端这一视角看，就是很久很久都没有响应。
- 引发网络阻塞。每次获取大 key 产生的网络流量较大，如果一个 key 的大小是 1 MB，每秒访问量为 1000，那么每秒会产生 1000MB 的流量，这对于普通千兆网卡的服务器来说是灾难性的。
- 阻塞工作线程。如果使用 del 删除大 key 时，会阻塞工作线程，这样就没办法处理后续的命令。
- 内存分布不均。集群模型在 slot 分片均匀情况下，会出现数据和查询倾斜情况，部分有大 key 的 Redis 节点占用内存多，QPS 也会比较大。

如何避免大 Key?

最好在设计阶段，就把大 key 拆分成一个一个小 key。或者，定时检查 Redis 是否存在大 key ，如果该大 key 是可以删除的，不要使用 DEL 命令删除，因为该命令删除过程会阻塞主线程，而是用 unlink 命令删除大 key，因为该命令的删除过程是异步的，不会阻塞主线程。

如何判断key已过期?

Redis 首先检查该 key 是否存在于过期字典中，如果不在，则正常读取键值；如果存在，则会获取该 key 的过期时间，然后与当前系统时间进行比对，如果比系统时间大，那就没有过期，否则判定该 key 已过期。

Redis主从节点是长连接还是短链接?

长连接

如何判断 Redis 某个节点是否正常工作?

Redis 判断节点是否正常工作，基本都是通过互相的 ping-pong 心态检测机制，如果有一半以上的节点去 ping 一个节点的时候没有 pong 回应，集群就会认为这个节点挂掉了，会断开与这个节点的连接。

主从复制架构中，过期 key 如何处理?

主节点处理了一个key或者通过淘汰算法淘汰了一个key，这个时间主节点模拟一条del命令发送给从节点，从节点收到该命令后，就进行删除key的操作。

Redis 是同步复制还是异步复制?

Redis 主节点每次收到写命令之后，先写到内部的缓冲区，然后异步发送给从节点。

主从复制中两个 Buffer(replication buffer 、repl backlog buffer)有什么区别?

- 出现的阶段不一样
  - repl backlog buffer 是在增量复制阶段出现，**一个主节点只分配一个 repl backlog buffer**；
  - replication buffer 是在全量复制阶段和增量复制阶段都会出现，**主节点会给每个新连接的从节点，分配一个 replication buffer**；
- 两个Buffer都有大小限制，当缓冲区满了之后，发生事情不一样
  - 当 repl backlog buffer 满了，因为是环形结构，会直接**覆盖起始位置数据**;
  - 当 replication buffer 满了，会导致连接断开，删除缓存，从节点重新连接，**重新开始全量复制。**

为什么会出现主从数据不一致?

主从数据不一致，就是指客户端从从节点中读取到的值和主节点中的最新值并不一致。之所以会出现主从数据不一致的现象，是**因为主从节点间的命令复制是异步进行的**，所以无法实现强一致性保证。具体来说，在主从节点命令传播阶段，主节点收到新的写命令后，会发送给从节点。但是，主节点并不会等到从节点实际执行完命令后，再把结果返回给客户端，而是主节点自己在本地执行完命令后，就会向客户端返回结果了。如果从节点还没有执行主节点同步过来的命令，主从节点间的数据就不一致了。

如何应对主从数据不一致?

- 尽量保证主从节点间的网络连接状况良好，避免主从节点在不同的机房。
- 可以开发一个外部程序来监控主从节点间的复制进度。

为什么会有哨兵机制?

在 Redis 的主从架构中，由于主从模式是读写分离的，如果主节点（master）挂了，那么将没有主节点来服务客户端的写操作请求，也没有主节点给从节点（slave）进行数据同步了。它的作用是实现**主从节点故障转移。**它会监测主节点是否存活，如果发现主节点挂了，它就会选举一个从节点切换为主节点，并且把新主节点的相关信息通知给从节点和客户端。

哨兵机制主要负责三件事情：**监控、选主、通知**

如何判断主节点真的故障了?

哨兵会每隔1s给所有主从节点发送PING命令，当主从节点收到PING命令后，会发送响应命令给哨兵，如果主从节点没在规定时间内响应命令，则认为其主观下线，但是对于判断主节点是否故障通常是部署多个哨兵节点，这样会减少误判，当其中一个哨兵认为主节点主观下线时，就会报告给其他哨兵，其他哨兵对该主节点来进行判断是否下线，最后根据总的赞同票数来判断是否客观下线。

由哪个哨兵进行主从故障转移?

首先由发现主节点主观下线的哨兵作为候选者，然后发起投票，当只有一个候选者时，它可以把票投给自己或者他人，然后征询其他哨兵的投票，要成功选举为leader，需要满足两个条件

- 拿到半数以上的赞成票；
- 拿到的票数同时还需要大于等于哨兵配置文件中的 quorum 值。

当有多个候选者时，候选者会先给自己投一票，然后向其他哨兵发起投票请求。如果投票者先收到「候选者 A」的投票请求，就会先投票给它，如果投票者用完投票机会后，收到「候选者 B」的投票请求后，就会拒绝投票。这时，候选者 A 先满足了上面的那两个条件，所以「候选者 A」就会被选举为 Leader。

为什么哨兵节点至少要有3个?

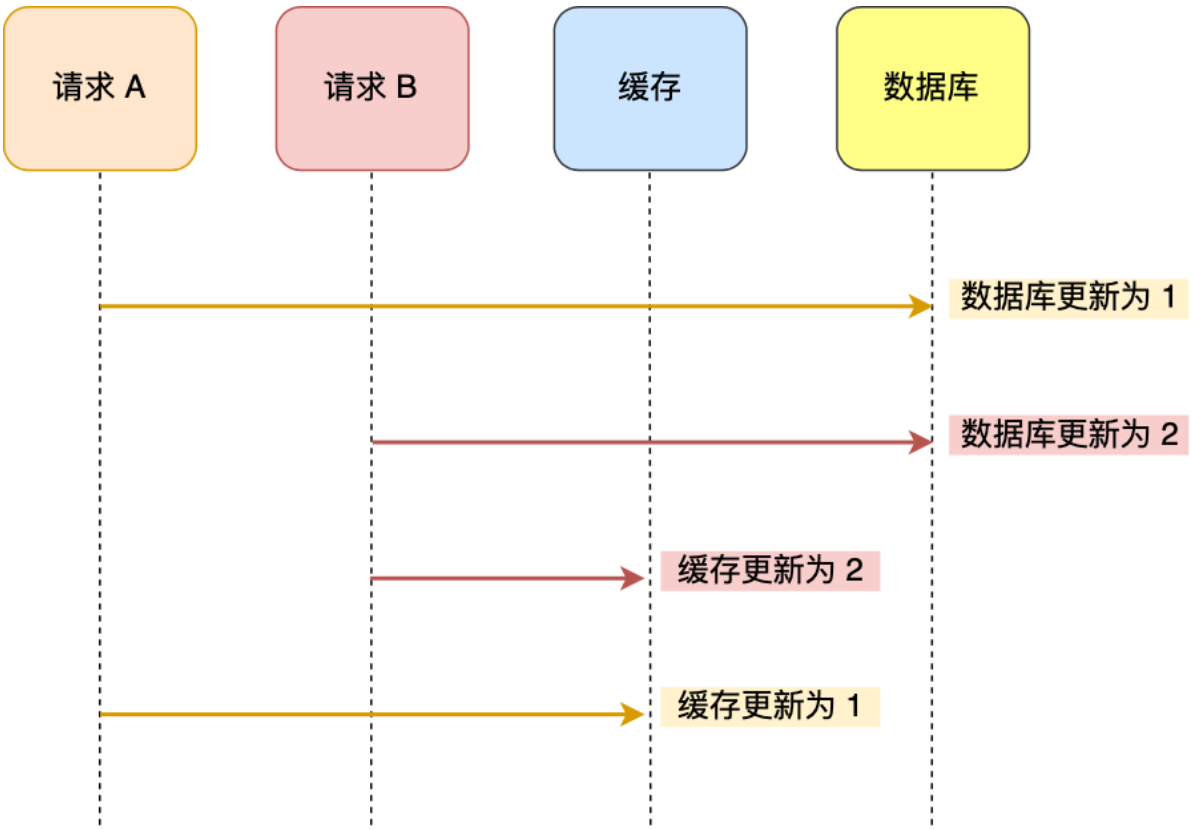
如果要是2个哨兵，其中一个哨兵要成为leader，必须获得2票，如果一个哨兵挂掉，则无法完成主从切换。

主从故障转移的过程是怎样的?

- 在已下线主节点（旧主节点）属下的所有「从节点」里面，挑选出一个从节点，并将其转换为主节点
    - 过滤掉已经离线的从节点；
    - 过滤掉历史网络连接状态不好的从节点；
    - 将剩下的从节点，进行三轮考察：优先级（越小越靠前）、复制进度（越大越靠前）、ID 号（选择小的）。在每一轮考察过程中，如果找到了一个胜出的从节点，就将其作为新主节点。
- 让已下线主节点属下的所有「从节点」修改复制目标，修改为复制「新主节点」；
- 将新主节点的 IP 地址和信息，通过「发布者/订阅者机制」通知给客户端；
- 继续监视旧主节点，当这个旧主节点重新上线时，将它设置为新主节点的从节点；

数据库和缓存如何保持一致性?

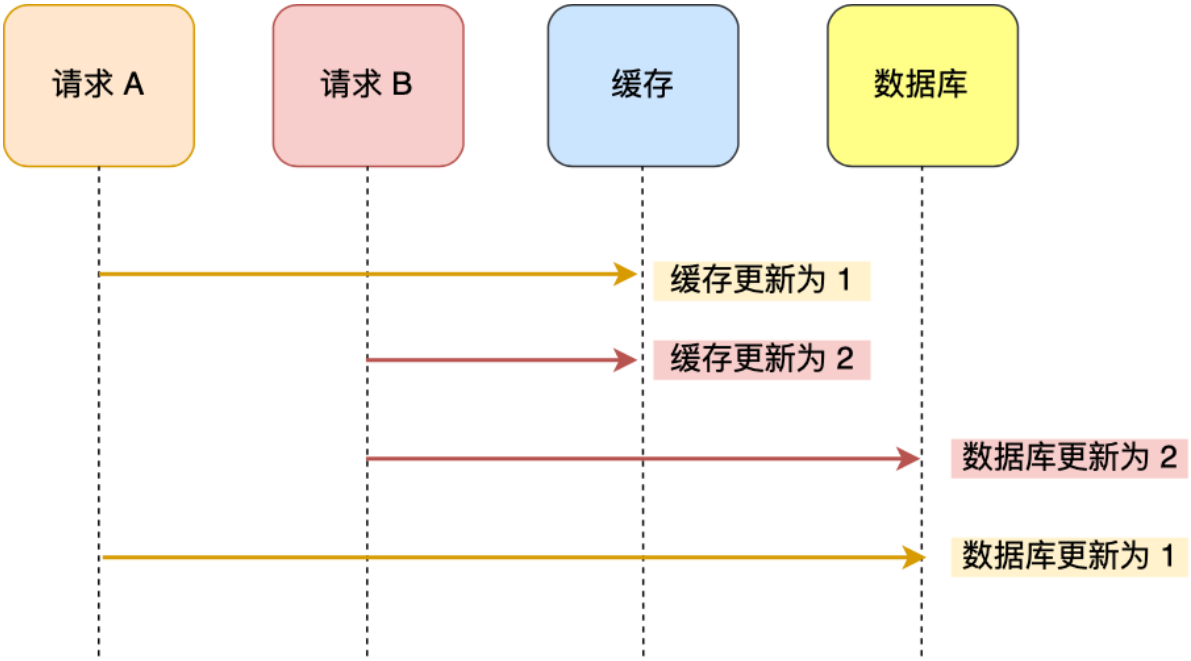
- 先更新数据库，再更新缓存



解决方案

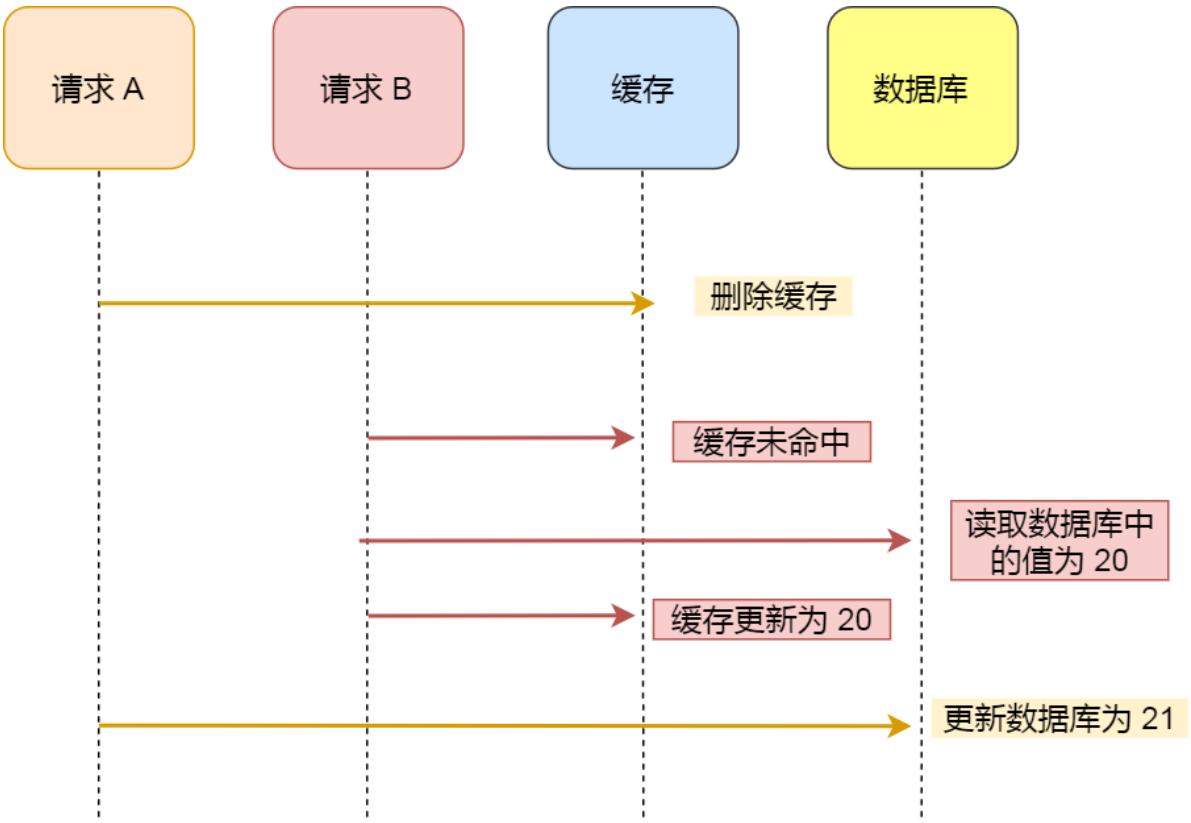
1. 在更新缓存前先加个**分布式锁**，保证同一时间只运行一个请求更新缓存，就会不会产生并发问题了，当然引入了锁后，对于写入的性能就会带来影响。
2. 在更新完缓存时，给缓存加上较短的**过期时间**，这样即时出现缓存不一致的情况，缓存的数据也会很快过期，对业务还是能接受的。

- 先更新缓存，再更新数据库

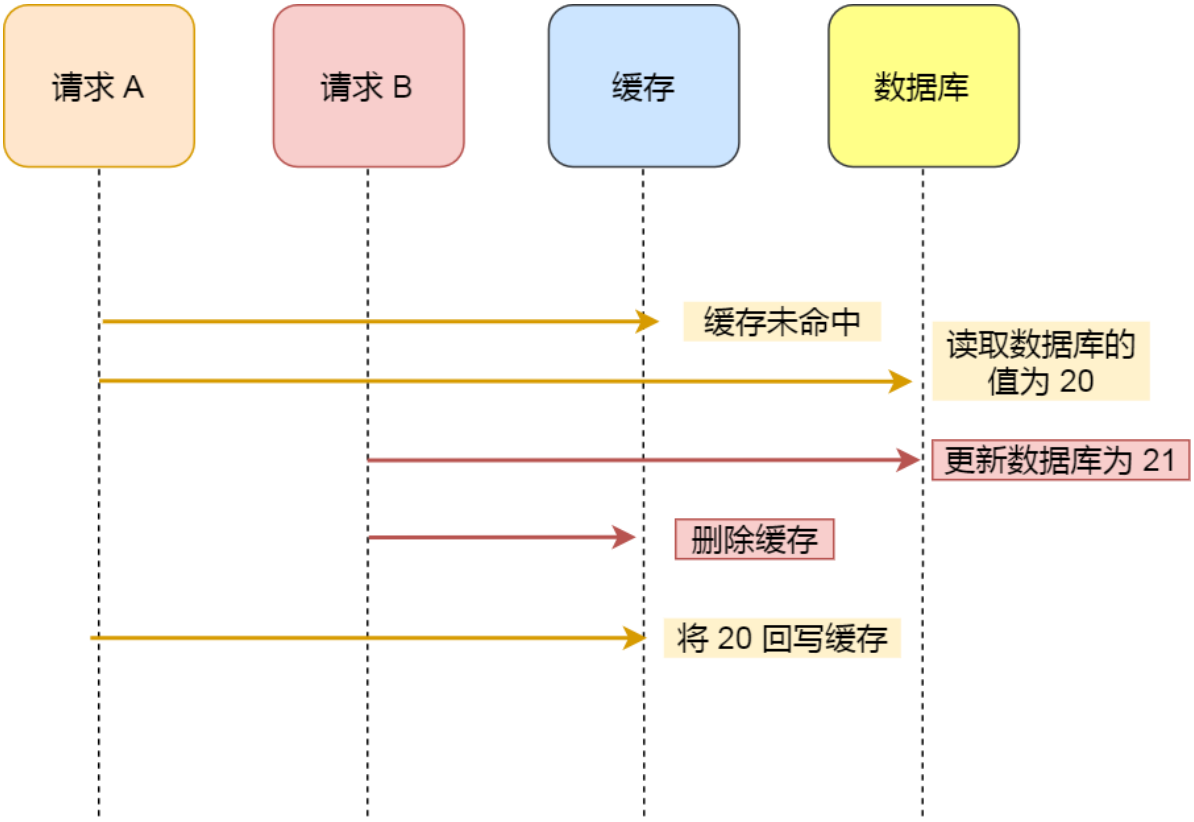


以上两种方法都会出现并发请求导致的数据不一致问题！

- 先删除缓存，再更新数据库

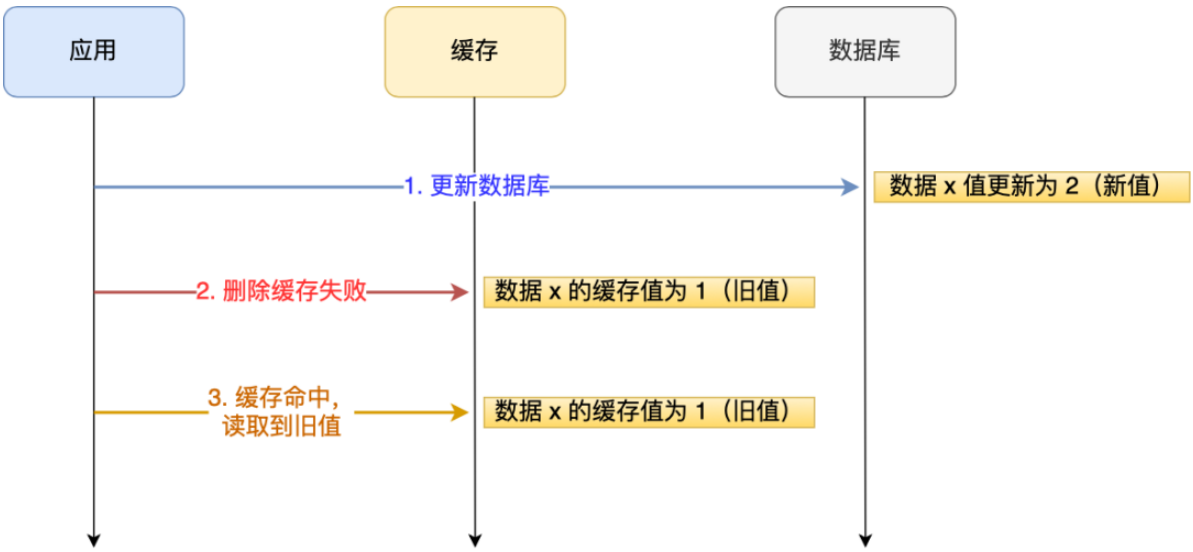


- 「先更新数据库 + 再删除缓存」（**Cache Aside 策略**）的方案，是可以保证数据一致性的。



理论上分析，先更新数据库，再删除缓存也是会出现数据不一致性的问题，**但是在实际中，这个问题出现的概率并不高。因为缓存的写入通常要远远快于数据库的写入**，所以在实际中很难出现请求 B 已经更新了数据库并且删除了缓存，请求 A 才更新完缓存的情况。

如何解决先更新数据库再删除缓存情况下，删除缓存失败导致数据不一致的问题？



- 重试机制：可以引入**消息队列**，将第二个操作（删除缓存）要操作的数据加入到消息队列，由消费者来操作数据。
  - 如果应用**删除缓存失败**，可以从消息队列中重新读取数据，然后再次删除缓存，这个就是**重试机制。**当然，如果重试超过的一定次数，还是没有成功，我们就需要向业务层发送报错信息了。
  - 如果**删除缓存成功**，就要把数据从消息队列中移除，避免重复操作，否则就继续重试。
- 订阅 MySQL binlog，再操作缓存：「**先更新数据库，再删缓存**」的策略的第一步是更新数据库，那么更新数据库成功，就会产生一条变更日志，记录在 binlog 里。于是我们就可以通过订阅 binlog 日志，拿到具体要操作的数据，然后再执行缓存删除。