1、听说过 Redis 吗？它是什么？

Redis 是一个**数据库**，不过与传统数据库不同的是 Redis 的数据库是存在**内存**中，所以**读写速度非常快**，因此 Redis 被广泛应用于**缓存**方向。

除此之外，Redis 也经常用来做分布式锁，Redis 提供了多种数据类型来支持不同的业务场景。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。

2、Redis 的五种数据结构整理

简单动态字符串(Simple Dynamic String，SDS)

Redis 没有直接使用 C 语言传统的字符串，而是自己构建了一种名为简单动态字符串（Simple dynamic string，SDS）的抽象类型，并将 SDS 用作 Redis 的默认字符串表示。

其实 SDS 等同于 C 语言中的 char * ，但它可以存储任意二进制数据，不能像 C 语言字符串那样以字符'\0'来标识字符串的结 束，因此它必然有个长度字段。

定义

```
struct sdshdr {
    // 记录 buf 数组中已使用字节的数量
    // 等于 sds 所保存字符串的长度
    int len;

    // 记录 buf 数组中未使用字节的数量
    int free;

    // 字节数组，用于保存字符串
    char buf[];
}
```

优点

- 获取字符串长度的复杂度为 O(1)。
- 杜绝缓冲区溢出。
- 减少修改字符串长度时所需要的内存重分配次数。
- 二进制安全。
- 兼容部分 C 字符串函数。

它具有很常规的 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。

链表

当有一个列表键包含了数量比较多的元素，又或者列表中包含的元素都是比较长的额字符串时，Redis 就会使用链表作为列表建的底层实现。

节点底层结构

```
typedef struct listNode {
    // 前置节点
    struct listNode *prev;
    // 后置节点
    struct listNode *next;
    // 节点的值
    void *value;
} listNode;
```

list 底层结构

```
typedef struct list {
    // 表头节点
    listNode *head;
    // 表尾节点
    listNode *tail;
    // 链表所包含的节点数量
    unsigned long len;
    // 节点值复制函数
    void *(*dup)(void *ptr);
    // 节点值是放过函数
    void (*free)(void *ptr);
    // 节点值对比函数
    int（*match）(void *ptr, void *key);
} list;
```

特性

- 链表被广泛用于实现 Redis 的各种功能，比如列表建、发布与订阅、慢查询、监视器等。
- 每个链表节点由一个 listNode 结构来表示，每个节点都有一个指向前置节点和后置节点的指针，所以 Redis 的链表实现是双端链表。
- 每个链表使用一个 list 结构表示，这个结构带有表头节点指针、表尾节点指针，以及链表长度等信息。
- 因为链表表头的前置节点和表尾节点的后置节点都指向 NULL，所以 Redis 的链表实现是无环链表。
- 通过为链表设置不同的类型特定函数，Redis 的链表可以用于保存各种不同类型的值。

字典

字典的底层是哈希表，类似 C++中的 map ，也就是键值对。

哈希表

```
typedef struct dictht {
    // 哈希表数组
    dictEntry **table;
    // 哈希表大小
    unsigned long size;
    // 哈希表大小掩码，用于计算索引值
    // 总是等于size-1
    unsigned long sizemark;
    // 该哈希表已有节点的数量
    unsigned long used;
} dichht;
```

哈希算法

当字典被用作数据库的底层实现，或者哈希键的底层实现时，Redis 使用 MurmurHash 算法。

这种算法的优点在于即使输入的键是规律的，算法仍能给出一个个很好的随机分布性，并且算法的计算速度非常快。

哈希冲突的解决方式

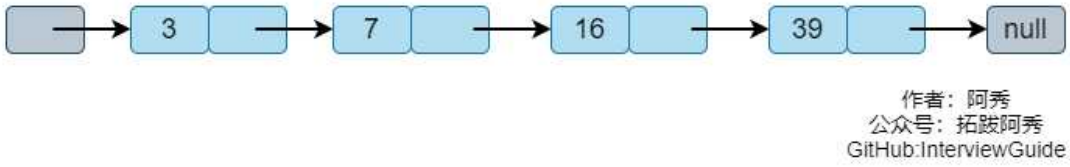
Redis 的哈希表使用链地址法来解决键冲突，每个哈希表节点都有一个 next 指针，多个哈希表节点可以用这个单向链表连接起来，这就解决了键冲突的问题。

特性

1. 字典被广泛用于实现 Redis 的各种功能，其中包括数据库和哈希键。
2. Redis 中的字典使用哈希表作为底层结构实现，每个字典带有两个哈希表，一个平时使用，另一个仅在进行 rehash 时使用。
3. Redis 使用 MurmurHash2 算法来计算键的哈希值。
4. 哈希表使用链地址法来解决键冲突。

跳跃表

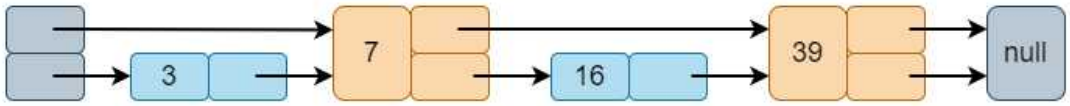
先看这样一张图：



作者：阿秀
公众号：拓跋阿秀
GitHub:InterviewGuide

如上图，我们要查找一个元素，就需要从头节点开始遍历，直到找到对应的节点或者是第一个大于要查找的元素的节点（没找到）。时间复杂度为 O(N)。

这个查找效率是比较低的，但如果我们把列表的某些节点拔高一层，如下图，例如把每两个节点中有一个节点变成两层。那么第二层的节点只有第一层的一半，查找效率也就会提高。



作者：阿秀
公众号：拓跋阿秀
GitHub:InterviewGuide

查找的步骤是从头节点的顶层开始，查到第一个大于指定元素的节点时，退回上一节点，在下一层继续查找。

比如我们要查找 16：

1. 从头节点的最顶层开始，先到节点 7。
2. 7 的下一个节点是 39，大于 16，因此我们退回到 7
3. 从 7 开始，在下一层继续查找，就可以找到 16。

这个例子中遍历的节点不比一维列表少，但是当节点更多，查找的数字更大时，这种做法的优势就体现出来了。还是上面的例子，如果我们要**查找的是 39**，那么只需要访问两个节点（7、39）就可以找到了。这比一维列表要减少一半的数量。

为了避免插入操作的时间复杂度是 O(N)，skiplist 每层的数量不会严格按照 2:1 的比例，而是对每个要插入的元素随机一个层数。

随机层数的计算过程如下：

1. 每个节点都有第一层
2. 那么它有第二层的概率是 p，有第三层的概率是 p*p
3. 不能超过最大层数

zskiplistNode

```
typedef struct zskiplistNode {
    // 后退指针
    struct zskiplistNode *backward;
    // 分值 权重
    double score;
    // 成员对象
    robj *obj;
    // 层
    struct zskiplistLevel {
        // 前进指针
        struct zskiplistNode *forward;
        // 跨度
        unsigned int span;
    } leval[];
} zskiplistNode;
```

一般来说，层的数量越多，访问其他节点的速度越快。

zskipList

```
typedef struct zskiplist {
    // 表头节点和表尾节点
    struct zskiplistNode *header, *tail;
    // 表中节点的数量
    unsigned long length;
    // 表中层数最大的节点的层数
    int leval;
} zskiplist;
```

特性

- 跳跃表是有序集合的底层实现之一
- Redis 的跳跃表实现由 zskiplist 和 zskiplistNode 两个结构组成，其中 zskiplist 用于保存跳跃表信息（比如表头节点、表尾节点、长度），而 zskiplistNode 则用于表示跳跃表节点
- 每个跳跃表节点的层高都是 1 至 32 之间的随机数
- 在同一个跳跃表中，多个节点可以包含相同的分值，但每个节点的成员对象必须是唯一的。
- 跳跃表中的节点按照分值大小进行排序，当分值相同时，节点按照成员对象的大小进行排序。
- 跳表是一种实现起来很简单，单层多指针的链表，它查找效率很高，堪比优化过的二叉平衡树，且比平衡树的实现简单。

压缩列表

> 压缩列表(ziplist)是列表键和哈希键的底层实现之一。当一个列表键只包含少量列表项，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键的底层实现。

特性

看他的名字就能看出来，是为了节省内存造的列表结构。

3、Redis 常见数据结构以及使用场景分别是什么？

String

String 数据结构是简单的 key-value 类型，value 其实不仅可以是 String，也可以是数字。 常规 key-value 缓存应用； 常规计数：微博数，粉丝数等。

Hash

Hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象，后续操作的时候，你可以直接仅 仅修改这个对象中的某个字段的值。 比如我们可以 Hash 数据结构来存储用户信息，商品信息等。

List

list 就是链表，Redis list 的应用场景非常多，也是 Redis 最重要的数据结构之一，比如微博的关注列表，粉丝列表， 消息列表等功能都可以用 Redis 的 list 结构来实现。

Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销。

另外可以通过 lrange 命令，就是从某个元素开始读取多少个元素，可以基于 list 实现分页查询，这个很棒的一个功 能，基于 Redis 实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西（一页一页的往下走），性能高。

Set

set 对外提供的功能与 list 类似是一个列表的功能，特殊之处在于 set 是可以自动排重的。

当你需要存储一个列表数据，又不希望出现重复数据时，set 是一个很好的选择，并且 set 提供了判断某个成员是否在 一个 set 集合内的重要接口，这个也是 list 所不能提供的。可以基于 set 轻易实现交集、并集、差集的操作。

比如：在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。

Redis 可以非常 方便的实现如共同关注、共同粉丝、共同喜好等功能。这个过程也就是求交集的过程，具体命令如下：`sinterstore key1 key2 key3` 将交集存在 key1 内。

Sorted Set

和 set 相比，sorted set 增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列。

举例： 在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维 度的消息排行榜）等信息，适合使用 Redis 中的 SortedSet 结构进行存储。

4、有 MySQL 不就够用了吗？为什么要用 Redis 这种新的数据库？

主要是因为 Redis 具备高性能和高并发两种特性。

1、**高性能**

假如用户第一次访问数据库中的某些数据。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存在缓存中，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。

操作缓存就是直接操作内存，所以速度相当快。如果数据库中的对应数据改变的之后，同步改变缓存中相应的数据即可！

2、**高并发**

直接操作缓存能够承受的请求是远远大于直接访问数据库的，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。

5、C++中的 Map 也是一种缓存型数据结构，为什么不用 Map，而选择 Redis 做缓存？

严格意义上来说缓存分为**本地缓存**和**分布式缓存**。

那以 C++ 语言为例，我们可以使用 STL 下自带的容器 map 来实现缓存，但只能实现本地缓存，它最主要的特点是轻量以及快速，但是其生命周期随着程序的销毁而结束，并且在多实例的情况下，每个实例都需要各自保存一份缓存，缓存不具有一致性。

使用 Redis 或 Memcached 之类的称为分布式缓存，在多实例的情况下，各实例共享一份缓存数据，缓存具有一致性。

这是 Redis 或者 Memcached 的优点所在，但它也有缺点，那就是需要保持 Redis 或 Memcached 服务的高可用，整个程序架构上较为复杂。

6、使用 Redis 的好处有哪些？

1、访问速度快，因为数据存在内存中，类似于 Java 中的 HashMap 或者 C++中的哈希表（如 unordered_map/unordered_set），这两者的优势就是查找和操作的时间复杂度都是 O(1)

2、数据类型丰富，支持 String，list，set，sorted set，hash 这五种数据结构

3、支持事务，Redis 中的操作都是原子性，换句话说就是对数据的更改要么全部执行，要么全部不执行，这就是原子性的定义

4、特性丰富：Redis 可用于缓存，消息，按 key 设置过期时间，过期后将会自动删除。

7、Memcached 与 Redis 的区别都有哪些？

1、存储方式

- Memecache 把数据全部存在内存之中，断电后会挂掉，没有持久化功能，数据不能超过内存大小。
- Redis 有部份存在硬盘上，这样能保证数据的持久性。

2、数据支持类型

- Memcache 对数据类型支持相对简单,只有 String 这一种类型
- Redis 有复杂的数据类型。Redis 不仅仅支持简单的 k/v 类型的数据，同时还提供 list，set，zset，hash 等数据结构的存储。

3、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。

4、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；但是 Redis 目前 是原生支持 cluster 模式的.

5、Memcached 是多线程，非阻塞 IO 复用的网络模型；Redis 使用单线程的多路 IO 复用模型。

6、Value 值大小不同：Redis 最大可以达到 512MB；Memcached 只有 1MB。

8、Redis 比 Memcached 的优势在哪里？

1、Memcached 所有的值均是简单字符串，Redis 作为其替代者，支持更为丰富的数据类型

2、Redis 的速度比 Memcached 快很多

3、Redis 可以做到持久化数据

9、缓存中常说的热点数据和冷数据是什么？

其实就是名字上的意思，热数据就是访问次数较多的数据，冷数据就是访问很少或者从不访问的数据。

需要注意的是只有热点数据，缓存才有价值。

对于冷数据而言，大部分数据可能还没有再次访问到就已经被挤出内存，不仅占用内存，而且价值不大。

数据更新前至少读取两次，缓存才有意义。这个是最基本的策略，如果缓存还没有起作用就失效了，那就没有太大价值了。

10、Redis 为什么是单线程的而不采用多线程方案？

这主要是基于一种客观原因来考虑的。因为 Redis 是基于内存的操作，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 不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）

11、单线程的 Redis 为什么这么快？

主要是有三个原因：

1、Redis 的全部操作都是纯内存的操作；

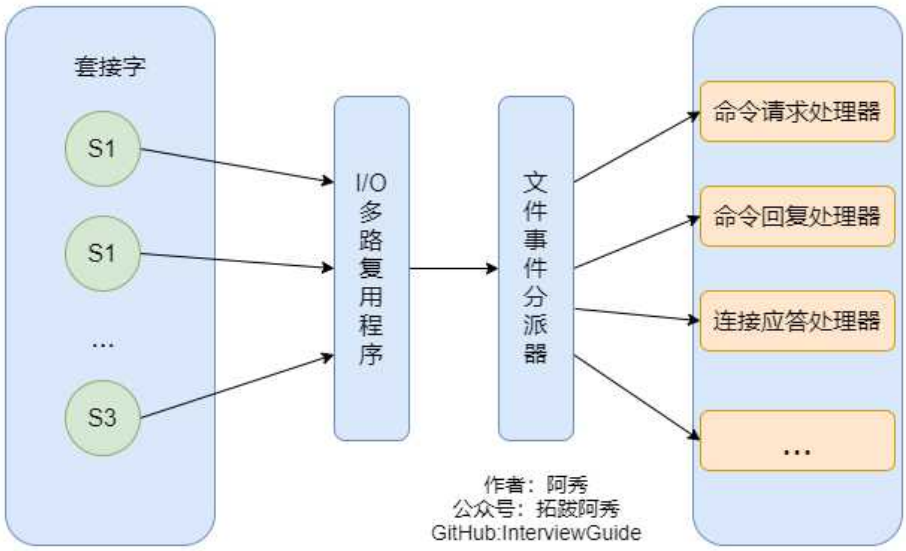
2、Redis 采用单线程，有效避免了频繁的上下文切换；

3，采用了非阻塞 I/O 多路复用机制。

12、了解 Redis 的线程模型吗？可以大致说说吗？

如果你打开看过 Redis 的源码就会发现 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：

- 多个 socket
- IO 多路复用程序
- 文件事件分派器
- 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）

使用 I/O 多路复用程序来同时监听多个套接字， 并根据套接字目前执行的任务来为套接字关联不同的事件处理器。

当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时， 与操作相对应的文件事件就会产生， 这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。

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。

一句话总结：“I/O 多路复用程序负责监听多个套接字， 并向文件事件分派器传送那些产生了事件的套接字。”

13、Redis 设置过期时间的两种方案是什么？

Redis 中有个设置时间过期的功能，即对存储在 Redis 数据库中的值可以设置一个过期时间。

作为一个缓存数据库， 这是非常实用的，比如一些 token 或者登录信息，尤其是短信验证码都是有时间限制的，按照传统的数据库处理方式，一般都是自己判断过期，这样无疑会严重影响项目性能。

我们 set key 的时候，都可以给一个 expire time，就是过期时间，通过过期时间我们可以指定这个 key 可以存活的时间，主要可采用**定期删除和惰性删除**两种方案。

1、定期删除

Redis 默认是每隔 100ms 就随机抽取一些设置了过期时间的 key，检查其是否过期，如果过期就删除。

注意这里是随机抽取的。为什么要随机呢？你想一想假如 Redis 存了几十万个 key ，每隔 100ms 就遍历所 有的设置过期时间的 key 的话，就会给 CPU 带来很大的负载！

2、惰性删除

定期删除可能会导致很多过期 key 到了时间并没有被删除掉，所以就有了惰性删除。

它是指某个键值过期后,此键值不会马上被删除,而是等到下次被使用的时候,才会被检查到过期，此时才能得到删除,惰性删除的缺点很明显是浪费内存。

除非你的系统去查一下那个 key，才会被 Redis 给删除掉。这就是所谓的惰性删除！

14、定期和惰性一定能保证删除数据吗？如果不能，Redis 会有什么应对措施？

并不能保证一定删除，Redsi 有一个 Redis 内存淘汰机制来确保数据一定会被删除。

首先介一下定期删除和惰性删除的工作流程：

1、定期删除，Redis 默认每个 100ms 检查，是否有过期的 key,有过期 key 则删除。需要说明的是，Redis 不是每个 100ms 将所有的 key 检查一次，而是随机抽取进行检查(如果每隔 100ms,全部 key 进行检查，Redis 岂不是卡死)。因此，如果只采用定期删除策略，会导致很多 key 到时间没有删除。

2、于是，惰性删除派上用场。也就是说在你获取某个 key 的时候，Redis 会检查一下，这个 key 如果设置了过期时间那么是否过期了？如果过期了此时就会删除。

3、采用定期删除+惰性删除就没其他问题了么？不是的，如果定期删除没删除 key。然后你也没即时去请求 key，也就是说惰性删除也没生效。这样，Redis

4、内存会越来越高。那么就应该采用内存淘汰机制。

在 Redis.conf 中有一行配置:maxmemory-policy volatile-lru

该配置就是配内存淘汰策略的，主要有以下六种方案：

1、**volatile-lru**：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

2、**volatile-ttl**：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

3、**volatile-random**：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

4、**allkeys-lru**：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

5、**allkeys-random**：从数据集（server.db[i].dict）中任意选择数据淘汰

6、**no-enviction**（驱逐）：禁止驱逐数据，新写入操作会报错

ps：如果没有设置 expire 的 key, 不满足先决条件(prerequisites); 那么 volatile-lru, volatile-random 和 volatile-ttl 策略的行为, 和 noeviction(不删除) 基本上一致。

15、Redis 对于大量的请求，是怎样处理的？

1、Redis 是一个单线程程序，也就说同一时刻它只能处理一个客户端请求；

2、Redis 是通过 IO 多路复用（select，epoll，kqueue，依据不同的平台，采取不同的实现）来处理多个客户端请求。

16、缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存击穿、缓存降级全搞定！

缓存雪崩

缓存雪崩指的是缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

我们可以简单的理解为：由于原有缓存失效，新缓存未到期间(例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库 CPU 和内存造成巨大压力，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。

解决办法

- 事前：尽量保证整个 Redis 集群的高可用性，发现机器宕机尽快补上，选择合适的内存淘汰策略。
- 事中：本地 ehcache 缓存 + hystrix 限流&降级，避免 MySQL 崩掉， 通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个 key 只允许一个线程查询数据和写缓存，其他线程等待。
- 事后：利用 Redis 持久化机制保存的数据尽快恢复缓存

缓存穿透

一般是黑客故意去请求缓存中不存在的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量 请求而崩掉。

这也看不懂？那我再换个说法好了。

缓存穿透是指查询一个一定不存在的数据，由于缓存不命中，接着查询数据库也无法查询出结果，因此也不会写入到缓存中，这将会导致每个查询都会去请求数据库，造成缓存穿透。

解决办法

1、布隆过滤器

这是最常见的一种解决方法了，它是将所有可能存在的数据哈希到一个足够大的 **bitmap** 中，一个一定不存在的数据会被 这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压 力。

对所有可能查询的参数以 hash 形式存储，在控制层先进行校验，不符合则丢弃，从而避免了对底层存储系统的查询压力；

这里稍微科普一下布隆过滤器。

> 布隆过滤器是引入了 k(k>1)k(k>1)个相互独立的哈希函数，保证在给定的空间、误判率下，完成元素判重的过程。 它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。
>
> 该算法的**核心思想**就是利用多个不同的 Hash 函数来解决“冲突”。Hash 存在一个冲突（碰撞）的问题，用同一个 Hash 得到的两个 URL 的值有可能相同。
>
> 为了减少冲突，我们可以多引入几个 Hash，如果通过其中的一个 Hash 值我们得出某元素不在集合中，那么该元素肯定不在集合中。只有在所有的 Hash 函数告诉我们该元素在集合中时，才能确定该元素存在于集合中。这便是布隆过滤器的基本思想，一般用于在大数据量的集合中判定某元素是否存在。

2、缓存空对象

当存储层不命中后，即使返回的空对象也将其缓存起来，同时会设置一个过期时间，之后再访问这个数据将会从缓存中获取，保护了后端数据源；

如果一个**查询返回的数据为空**（不管是数据不存 在，还是系统故障），我们仍然把这个空结果进行缓存，但它的**过期时间会很短**，最长不超过五分钟。

但是这种方法会存在两个问题：

1、如果空值能够被缓存起来，这就意味着缓存需要更多的空间存储更多的键，因为这当中可能会有很多的空值的键；

2、即使对空值设置了过期时间，还是会存在缓存层和存储层的数据会有一段时间窗口的不一致，这对于需要保持一致性的业务会有影响。

我们可以从适用场景和维护成本两方面对这两种汇总方法进行一个**简单比较**：

适用场景：

缓存空对象适用于 1、数据命中不高 2、数据频繁变化且实时性较高 ；

布隆过滤器适用 1、数据命中不高 2、数据相对固定即实时性较低

维护成本：

缓存空对象的方法适合 1、代码维护简单 2、需要较多的缓存空间 3、数据会出现不一致的现象；

布隆过滤器适合 1、代码维护较复杂 2、缓存空间要少一些

缓存预热

缓存预热是指系统上线后，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，用户会直接查询事先被预热的缓存数据！

解决思路 1、直接写个缓存刷新页面，上线时手工操作下； 2、数据量不大，可以在项目启动的时候自动进行加载； 3、定时刷新缓存；

缓存更新

除了缓存服务器自带的缓存失效策略之外（Redis 默认的有 6 中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：**定时删除和惰性删除**，其中：

- 定时删除：定时去清理过期的缓存；
- 惰性删除：当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的 key 是比较麻烦的；第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

缓存击穿

缓存击穿，是指一个 key 非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个 key 在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。

比如常见的电商项目中，某些货物成为“爆款”了，可以对一些主打商品的缓存直接设置为永不过期。

即便某些商品自己发酵成了爆款，也是直接设为永不过期就好了。mutex key 互斥锁基本上是用不上的，有个词叫做大道至简。

缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。

系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

以参考日志级别设置预案： （1）一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级； （2）警告：有些服务在一段时间内成功率有波动（如在 95~100% 之间），可以自动降级或人工降级，并发送告警； （3）错误：比如可用率低于 90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级； （4）严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

服务降级的目的，是为了防止 Redis 服务故障，导致数据库跟着一起发生雪崩问题。

因此，对于不重要的缓存数据，可以采取服务降级策略，例如一个比较常见的做法就是，Redis 出现问题，不去数据库查询，而是直接返回默认值给用户。

17、假如 MySQL 有 1000 万数据，采用 Redis 作为中间缓存，取其中的 10 万，如何保证 Redis 中的数据都是热点数据？

可以使用 Redis 的**数据淘汰策略**，Redis 内存数据集大小上升到一定大小的时候，就会施行这种策略。具体说来，主要有 6 种内存淘汰策略：

> voltile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
>
> volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
>
> volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
>
> allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰
>
> allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
>
> no-enviction（驱逐）：禁止驱逐数据

18、Redis 持久化机制可以说一说吗？

Redis 是一个支持持久化的内存数据库，通过持久化机制把内存中的数据同步到硬盘文件来保证数据持久化。当 Redis 重启后通过把硬盘文件重新加载到内存，就能达到恢复数据的目的。

很多时候我们需要持久化数据也就是将内存中的数据写入到硬盘里面，大部分原因是为了之后重用数据（比如重启机 器、机器故障之后回复数据），或者是为了防止系统故障而将数据备份到一个远程位置。

实现：单独创建 fork()一个子进程，将当前父进程的数据库数据复制到子进程的内存中，然后由子进程写入到临时文件中，持久化的过程结束了，再用这个临时文件替换上次的快照文件，然后子进程退出，内存释放。

以下有两种持久化机制

快照（snapshotting）持久化（RDB 持久化）

Redis 可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。Redis 创建快照之后，可以对快照进行 备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis 主从结构，主要用来提高 Redis 性 能），还可以将快照留在原地以便重启服务器的时候使用。

快照持久化是 Redis 默认采用的持久化方式，在 Redis.conf 配置文件中默认有此下配置：

```
save 900 1 #在 900 秒(15 分钟)之后，如果至少有 1 个 key 发生变化，Redis 就会自动触发 BGSAVE 命令
创建快照。

save 300 10 #在 300 秒(5 分钟)之后，如果至少有 10 个 key 发生变化，Redis 就会自动触发 BGSAVE 命令创建快照。

save 60 10000 #在 60 秒(1 分钟)之后，如果至少有 10000 个 key 发生变化，Redis 就会自动触发 BGSAVE 命令创建快照。
```

AOF（append-only file）持久化

与快照持久化相比，AOF **持久化的实时性更好**，因此已成为主流的持久化方案。默认情况下 Redis 没有开启 AOF（append only file）方式的持久化，可以通过 appendonly 参数开启：`appendonly yes`

开启 AOF 持久化后每执行一条会更改 Redis 中的数据的命令，Redis 就会将该命令写入硬盘中的 AOF 文件。AOF 文件的 保存位置和 RDB 文件的位置相同，都是通过 dir 参数设置的，默认的文件名是 appendonly.aof。

在 Redis 的配置文件中存在三种不同的 AOF 持久化方式，它们分别是：

```
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  #每秒钟同步一次，显示地将多个写命令同步到硬盘
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，用户可以考虑 appendfsync everysec 选项 ，让 Redis 每秒同步一次 AOF 文件，Redis 性能 几乎没受到任何影响。

而且这样即使出现系统崩溃，用户最多只会丢失一秒之内产生的数据。当硬盘忙于执行写入操作的时候，Redis 还会优雅的放慢自己的速度以便适应硬盘的最大写入速度。

Redis 4.0 对于持久化机制的优化

Redis 4.0 开始支持 RDB 和 AOF 的混合持久化（默认关闭，可以通过配置项 aof-use-rdb-preamble 开启）。

如果把混合持久化打开，AOF 重写的时候就直接把 RDB 的内容写到 AOF 文件开头。这样做的好处是可以结合 RDB 和 AOF 的优点, 快速加载同时避免丢失过多的数据。

当然缺点也是有的， AOF 里面的 RDB 部分是压缩格式不再是 AOF 格式，可读性较差。

19、AOF 重写了解吗？可以简单说说吗？

AOF 重写可以产生一个新的 AOF 文件，这个新的 AOF 文件和原有的 AOF 文件所保存的数据库状态一样，**但体积更小**。

AOF 重写是一个有歧义的名字，该功能是通过读取数据库中的**键值**对来实现的，程序无须对现有 AOF 文件进行任何读 入、分析或者写入操作。

在执行 BGREWRITEAOF 命令时，Redis 服务器会维护一个 AOF **重写缓冲区**，该缓冲区会在子进程创建新 AOF 文件期间，记录服务器执行的所有写命令。

当子进程完成创建新 AOF 文件的工作之后，服务器会将重写缓冲区中的所有内容 追加到新 AOF 文件的末尾，使得新旧两个 AOF 文件所保存的数据库状态一致。最后，服务器用新的 AOF 文件替换旧的 AOF 文件，以此来完成 AOF 文件重写操作。

20、是否使用过 Redis 集群？集群的原理是什么？

Redis Sentinel（哨兵）着眼于高可用，在 master 宕机时会自动将 slave 提升为 master，继续提供服务。

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，自动从服务器中选举出新的主服务器。

Redis Cluster（集群）着眼于扩展性，在单个 Redis 内存不足时，使用 Cluster 进行分片存储。

21、如何解决 Redis 的并发竞争 Key 问题

所谓 Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作，但是最后执行的顺序和我们期望的顺序不同，这样也就导致了结果的不同！

推荐一种方案：分布式锁（zookeeper 和 Redis 都可以实现分布式锁）。（如果不存在 Redis 的并发竞争 Key 问 题，不要使用分布式锁，这样会影响性能）

基于 zookeeper 临时有序节点可以实现的分布式锁。大致思想为：每个客户端对某个方法加锁时，在 zookeeper 上的 与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。

判断是否获取锁的方式很简单，只需要判断有 序节点中序号最小的一个。当释放锁的时候，只需将这个瞬时节点删除即可。

同时，其可以避免服务宕机导致的锁 无法释放，而产生的死锁问题。完成业务流程后，删除对应的子节点释放锁。

在实践中，当然是从以可靠性为主。所以首推 Zookeeper。

22、如何保证缓存与数据库双写时的数据一致性

首先说一句，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？

一般来说，就是如果你的系统不是严格要求缓存+数据库必须一致性的话，缓存可以稍微的跟数据库偶尔有不一致的 情况，最好不要做这个方案，最好将**读请求和写请求串行化**，串到一个内存队列里去，这样就可以保证一定不会出现不一致的情况。

串行化之后，就会导致系统的吞吐量会大幅度的降低，用比正常情况下多几倍的机器去支撑线上的一个请求。

最经典的缓存+数据库读写的模式，就是 **预留缓存模式** 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，**先删除缓存，然后再更新数据库，这样读的时候就会发现缓存中没有数据而直接去数据库中拿数据了**。（因为要删除，狗日的编辑器可能会背着你做一些优化，要彻底将缓存中的数据进行删除才行）

互联网公司非常喜欢问这道面试题，因为缓存在互联网公司使用非常频繁。

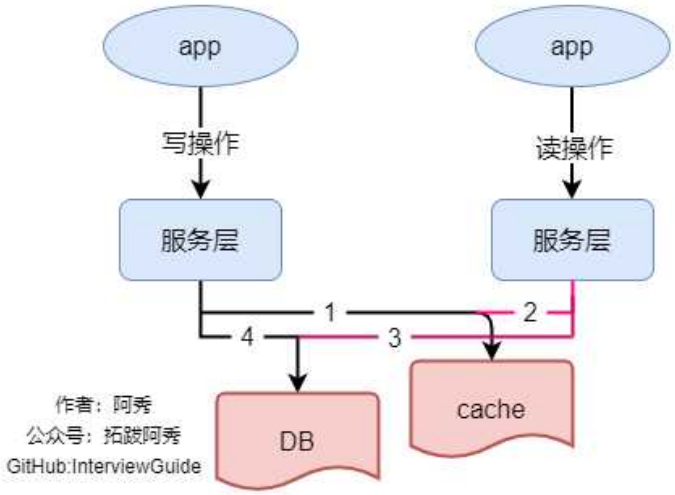
在高并发的业务场景下，数据库的性能瓶颈往往都是用户并发访问过大。所以，一般都使用 Redis 做一个缓冲操作，让请求先访问到 Redis，而不是直接去访问 MySQL 等数据库，从而减少网络请求的延迟响应。

23、数据为什么会出现不一致的情况？

这样的问题主要是在并发读写访问的时候，缓存和数据相互交叉执行。

一、单库情况下

同一时刻发生了并发读写请求，**例如为 A(写) B (读)2 个请求**

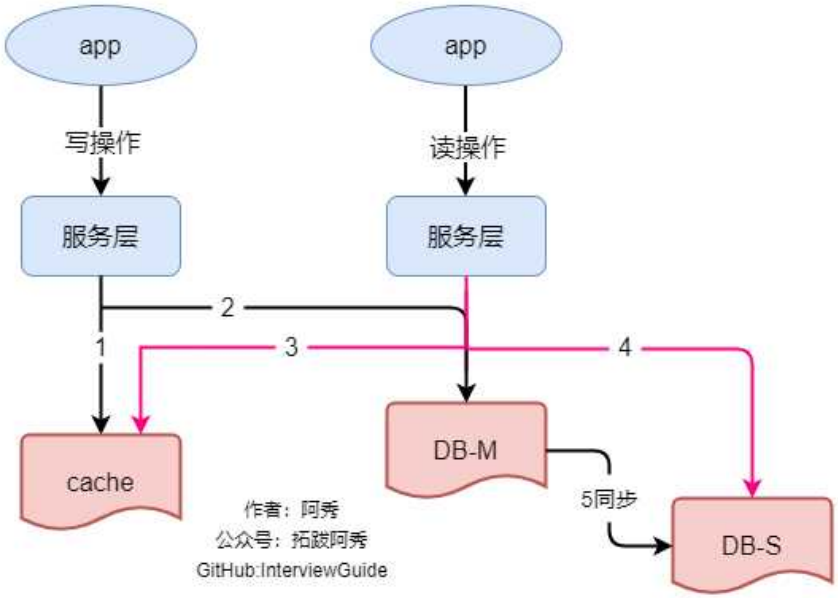


1. A 请求发送一个写操作到服务端，第一步会淘汰 cache，然后因为各种原因卡主了，不在执行后面业务(例：大量的业务操作、调用其他服务处理消耗了 1s）。
2. B 请求发送一个读操作，读 cache，因为 cache 淘汰，所以为空
3. B 请求继续读 DB，读出一个脏数据，并写入 cache
4. A 请求终于执行完全，在写入数据到 DB

总结：因最后才把写操作数据入 DB，并没同步。cache 里面一直保持脏数据

脏数据是指源系统中的数据不在给定的范围内或对于实际业务毫无意义，或是数据格式非法，以及在源系统中存在不规范的编码和含糊的业务逻辑。

二、主从同步，读写分离的情况下，读从库而产生脏数据



1. A 请求发送一个写操作到服务端，第一步会淘汰 cache
2. A 请求写主数据库，写了最新的数据。
3. B 请求发送一个读操作，读 cache，因为 cache 淘汰，所以为空
4. B 请求继续读 DB，读的是从库，此时主从同步还没同步成功。读出脏数据，然后脏数据入 cache
5. 最后数据库主从同步完成

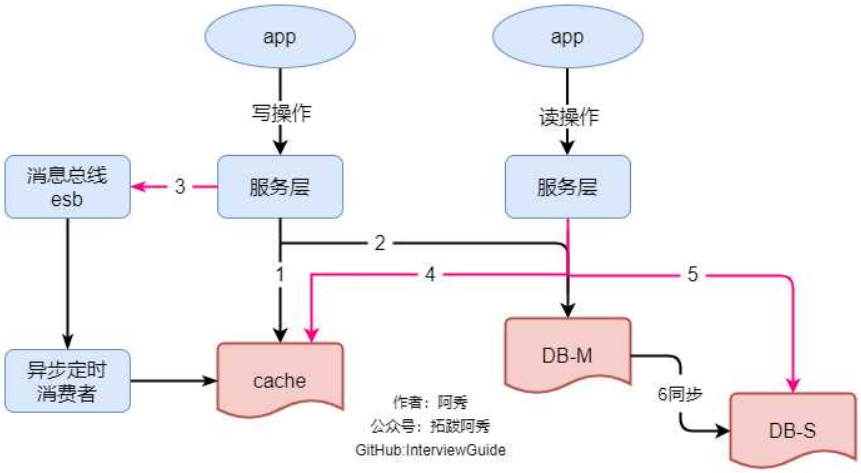
总结：这种情况下请求 A 和请求 B 操作时序没问题，是主从同步的时延问题(假设 1s)，导致读请求读取从库读到脏数据导致的数据不一致

根本原因:

单库下，逻辑处理中消耗 1s，可能读到旧数据入缓存

主从+读写分离，在 1s 的主从同步时延中，到从库的旧数据入缓存

24、常见的数据优化方案你了解吗？

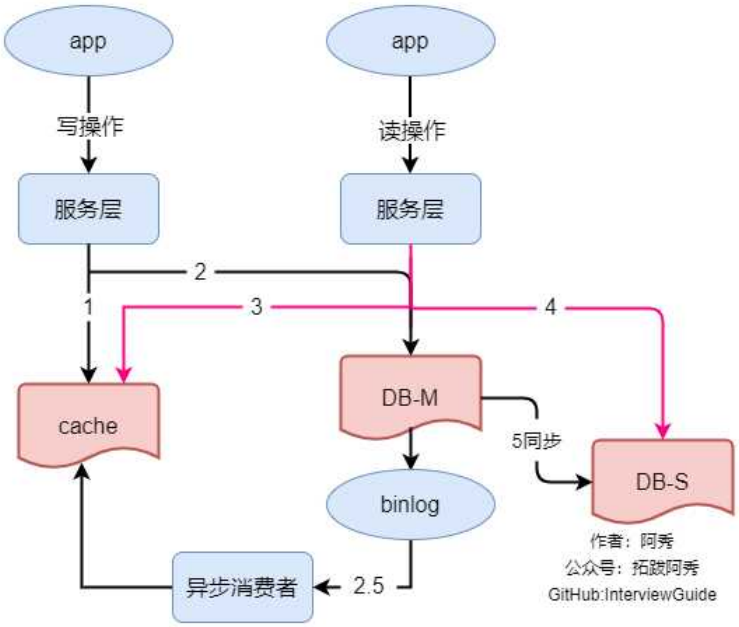


一、缓存双淘汰法

1. 先淘汰缓存
2. 再写数据库
3. 往消息总线 esb 发送一个淘汰消息，发送立即返回。写请求的处理时间几乎没有增加，这个方法淘汰了缓存两次。因此被称为“缓存双淘汰法”，而在消息总线下游，有一个异步淘汰缓存的消费者，在拿到淘汰消息在 1s 后淘汰缓存，这样，即使在一秒内有脏数据入缓存，也能够被淘汰掉。

二、异步淘汰缓存

上述的步骤，都是在业务线里面执行，新增一个线下的读取 binlog 异步淘汰缓存模块，读取 binlog 总的数据，然后进行异步淘汰。



这里简单提供一个思路

1.思路：

MySQL binlog 增量发布订阅消费+消息队列+增量数据更新到 Redis

1）读请求走 Redis：热数据基本都在 Redis

2）写请求走 MySQL: 增删改都操作 MySQL

3）更新 Redis 数据：MySQ 的数据操作 binlog，来更新到 Redis

2.Redis 更新

1）数据操作主要分为两块：

- 一个是全量(将全部数据一次写入到 Redis)
- 一个是增量（实时更新）

这里说的是增量,指的是 mysql 的 update、insert、delate 变更数据。这样一旦 MySQL 中产生了新的写入、更新、删除等操作，就可以把 binlog 相关的消息推送至 Redis，Redis 再根据 binlog 中的记录，对 Redis 进行更新，就无需在从业务线去操作缓存内容。

25、Redis 的高并发和高可用是如何保证的？

这样的问题主要是在并发读写访问的时候，缓存和数据相互交叉执行。

Redis 的主从架构模式

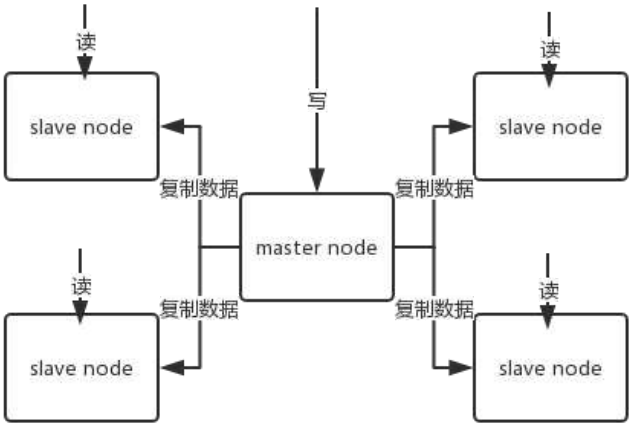
Rdis 的主从架构模式是实现高并发的主要依赖，一般很多项目只需要一主多从就可以实现其所需要的功能。通常使用单主用来写入数据，单机几万 QPS；多从一般是查询数据，多个从实例可以提供每秒 10w 的 QPS。

一些项目需要在实现高并发的同时，尽可能多的容纳大量的数据，这时需要使用 Redis 集群，使用 Redis 集群之后，可以提供每秒几十万的读写并发。

Redis 高可用，如果是做主从架构部署，那么加上哨兵就可以实现，任何一个实例宕机，可以进行主备切换。

单机的 Redis，能够承载的 QPS 大概就在上万到几万不等。对于缓存来说，一般都是用来支撑**读高并发**的。

因此架构做成主从(master-slave)架构，一主多从，主负责写，并且将数据复制到其它的 slave 节点，从节点负责读。所有的**读请求全部走从节点**。这样也可以很轻松实现水平扩容，**支撑读高并发**。



Redis replication -> 主从架构 -> 读写分离 -> 水平扩容支撑读高并发

Redis replication 的核心机制

- Redis 采用**异步方式**复制数据到 slave 节点，不过 Redis2.8 开始，slave node 会周期性地确认自己每次复制的数据量；
- 一个 master node 是可以配置多个 slave node 的；
- slave node 也可以连接其他的 slave node；
- slave node 做复制的时候，不会 block master node 的正常工作；
- slave node 在做复制的时候，也不会 block 对自己的查询操作，它会用旧的数据集来提供服务；但是复制完成的时候，需要删除旧数据集，加载新数据集，这个时候就会暂停对外服务了；
- slave node 主要用来进行横向扩容，做读写分离，扩容的 slave node 可以提高读的吞吐量。

注意，如果采用了主从架构，那么建议必须**开启** master node 的持久化，不建议用 slave node 作为 master node 的数据热备，因为那样的话，如果你关掉 master 的持久化，可能在 master 宕机重启的时候数据是空的，然后可能一经过复制， slave node 的数据也丢了。

另外，master 的各种备份方案，也需要做。万一本地的所有文件丢失了，从备份中挑选一份 rdb 去恢复 master，这样才能**确保启动的时候，是有数据的**，即使采用了后续讲解的**高可用机制**，slave node 可以自动接管 master node，但也可能 sentinel 还没检测到 master failure，master node 就自动重启了，还是可能导致上面所有的 slave node 数据被清空。

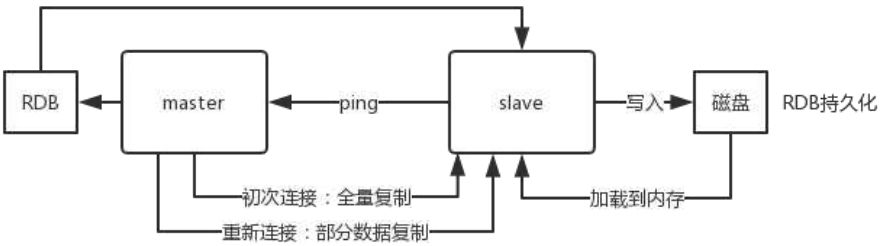
Redis 主从复制的核心原理

当启动一个 slave node 的时候，它会发送一个 PSYNC 命令给 master node。

如果这是 slave node 初次连接到 master node，那么会触发一次 `full resynchronization` 全量复制。此时 master 会启动一个后台线程，开始生成一份 RDB 快照文件，同时还会将从客户端 client 新收到的所有写命令缓存在内存中。

RDB 文件生成完毕后， master 会将这个 RDB 发送给 slave，slave 会先**写入本地磁盘，然后再从本地磁盘加载到内存**中，接着 master 会将内存中缓存的写命令发送到 slave，slave 也会同步这些数据。

slave node 如果跟 master node 有网络故障，断开了连接，会自动重连，连接之后 master node 仅会复制给 slave 部分缺少的数据。



无磁盘化复制

master 在内存中直接创建 RDB ，然后发送给 slave，不会在自己本地落地磁盘了。只需要在配置文件中开启 `repl-diskless-sync yes` 即可。

```
repl-diskless-sync yes
# 等待 5s 后再开始复制，因为要等更多 slave 重新连接过来
repl-diskless-sync-delay 5Copy to clipboardErrorCopied
```

过期 key 处理

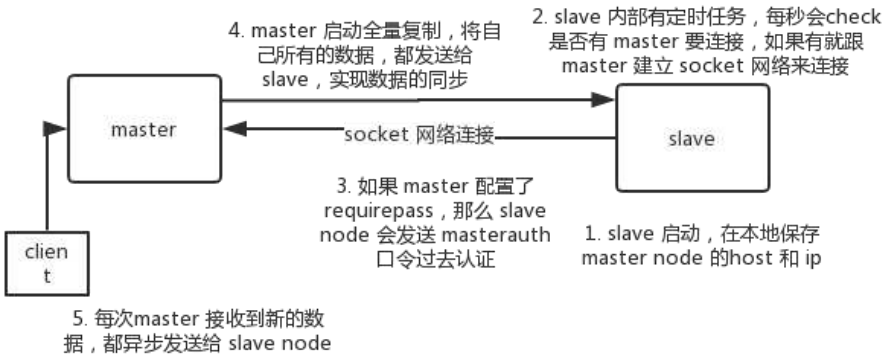
slave 不会过期 key，只会等待 master 过期 key。如果 master 过期了一个 key，或者通过 LRU 淘汰了一个 key，那么会模拟一条 del 命令发送给 slave。

复制的完整流程

slave node 启动时，会在自己本地保存 master node 的信息，包括 master node 的 `host` 和 `ip` ，但是复制流程没开始。

slave node 内部有个定时任务，每秒检查是否有新的 master node 要连接和复制，如果发现，就跟 master node 建立 socket 网络连接。然后 slave node 发送 `ping` 命令给 master node。

如果 master 设置了 requirepass，那么 slave node 必须发送 masterauth 的口令过去进行认证。master node **第一次执行全量复制**，将所有数据发给 slave node。而在后续，master node 持续将写命令，异步复制给 slave node。



全量复制

- master 执行 bgsave ，在本地生成一份 rdb 快照文件。
- master node 将 rdb 快照文件发送给 slave node，如果 rdb 复制时间超过 60 秒（repl-timeout），那么 slave node 就会认为复制失败，可以适当调大这个参数(对于千兆网卡的机器，一般每秒传输 100MB，6G 文件，很可能超过 60s)
- master node 在生成 rdb 时，会将所有新的写命令缓存在内存中，在 slave node 保存了 rdb 之后，再将新的写命令复制给 slave node。
- 如果在复制期间，内存缓冲区持续消耗超过 64MB，或者一次性超过 256MB，那么停止复制，复制失败。

```
client-output-buffer-limit slave 256MB 64MB 60Copy to clipboardErrorCopied
```

- slave node 接收到 rdb 之后，清空自己的旧数据，然后重新加载 rdb 到自己的内存中，同时**基于旧的数据版本**对外提供服务。
- 如果 slave node 开启了 AOF，那么会立即执行 BGREWRITEAOF，重写 AOF。

增量复制

- 如果全量复制过程中，master-slave 网络连接断掉，那么 slave 重新连接 master 时，会触发增量复制。
- master 直接从自己的 backlog 中获取部分丢失的数据，发送给 slave node，默认 backlog 就是 1MB。
- master 就是根据 slave 发送的 psync 中的 offset 来从 backlog 中获取数据的。

heartbeat

主从节点互相都会发送 heartbeat 信息。

master 默认每隔 10 秒发送一次 heartbeat，slave node 每隔 1 秒发送一个 heartbeat。

异步复制

master 每次接收到写命令之后，先在内部写入数据，然后异步发送给 slave node。

26、Redis 如何才能做到高可用？

如果系统在 365 天内，有 99.99% 的时间，都是可以哗哗对外提供服务的，那么就说系统是高可用的。

一个 slave 挂掉了，是不会影响可用性的，还有其它的 slave 在提供相同数据下的相同的对外的查询服务。

但是，如果 master node 死掉了，会怎么样？没法写数据了，写缓存的时候，全部失效了。slave node 还有什么用呢，没有 master 给它们复制数据了，系统相当于不可用了。

Redis 的高可用架构，叫做 `failover` **故障转移**，也可以叫做主备切换。

master node 在故障时，自动检测，并且将某个 slave node 自动切换为 master node 的过程，叫做主备切换。这个过程，实现了 Redis 的主从架构下的高可用。

27、Redis 集于哨兵集群实现高可用？

哨兵的介绍

sentinel，中文名是哨兵。哨兵是 Redis 集群架构中非常重要的一个组件，主要有以下功能：

- 集群监控：负责监控 Redis master 和 slave 进程是否正常工作。
- 消息通知：如果某个 Redis 实例有故障，那么哨兵负责发送消息作为报警通知给管理员。
- 故障转移：如果 master node 挂掉了，会自动转移到 slave node 上。
- 配置中心：如果故障转移发生了，通知 client 客户端新的 master 地址。

哨兵用于实现 Redis 集群的高可用，本身也是分布式的，作为一个哨兵集群去运行，互相协同工作。

- 故障转移时，判断一个 master node 是否宕机了，需要大部分的哨兵都同意才行，涉及到了分布式选举的问题。
- 即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的，因为如果一个作为高可用机制重要组成部分的故障转移系统本身是单点的，那就很坑爹了。

哨兵的核心知识

- 哨兵至少需要 3 个实例，来保证自己的健壮性。
- 哨兵 + Redis 主从的部署架构，是**不保证数据零丢失**的，只能保证 Redis 集群的高可用性。
- 对于哨兵 + Redis 主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练。

哨兵集群必须部署 2 个以上节点，如果哨兵集群仅仅部署了 2 个哨兵实例，quorum = 1。

```
+----+         +----+
| M1 |---------| R1 |
| S1 |         | S2 |
+----+         +----+Copy to clipboardErrorCopied
```

配置 quorum=1 ，如果 master 宕机， s1 和 s2 中只要有 1 个哨兵认为 master 宕机了，就可以进行切换，同时 s1 和 s2 会选举出一个哨兵来执行故障转移。但是同时这个时候，需要 majority，也就是大多数哨兵都是运行的。

```
2 个哨兵，majority=2
3 个哨兵，majority=2
4 个哨兵，majority=2
5 个哨兵，majority=3
...Copy to clipboardErrorCopied
```

如果此时仅仅是 M1 进程宕机了，哨兵 s1 正常运行，那么故障转移是 OK 的。但是如果是整个 M1 和 S1 运行的机器宕机了，那么哨兵只有 1 个，此时就没有 majority 来允许执行故障转移，虽然另外一台机器上还有一个 R1，但是故障转移不会执行。

经典的 3 节点哨兵集群是这样的：

```
      +----+
      | M1 |
      | S1 |
      +----+
         |
+----+   |   +----+
| R2 |----+----| R3 |
| S2 |        | S3 |
+----+        +----+Copy to clipboardErrorCopied
```

配置 quorum=2，如果 M1 所在机器宕机了，那么三个哨兵还剩下 2 个，S2 和 S3 可以一致认为 master 宕机了，然后选举出一个来执行故障转移，同时 3 个哨兵的 majority 是 2，所以还剩下的 2 个哨兵运行着，就可以允许执行故障转移。

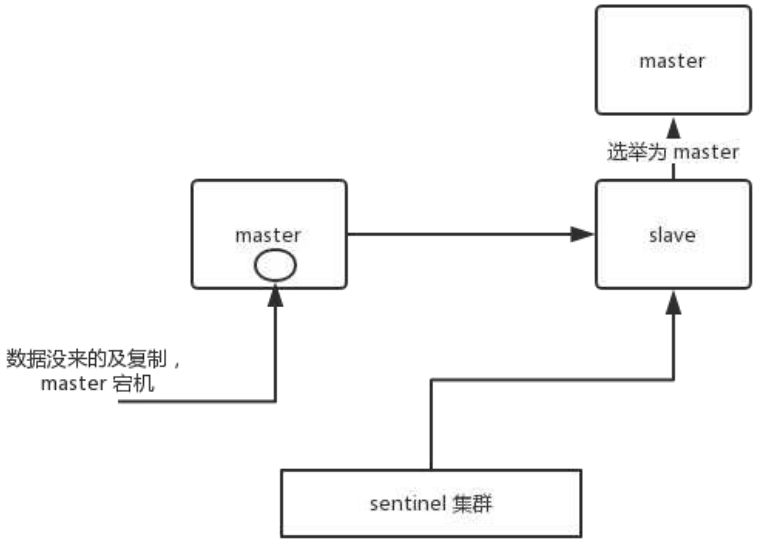
28、Redis 哨兵主备切换的数据丢失问题

导致数据丢失的两种情况

主备切换的过程，可能会导致数据丢失：

- 异步复制导致的数据丢失

因为 master->slave 的复制是异步的，所以可能有部分数据还没复制到 slave，master 就宕机了，此时这部分数据就丢失了。

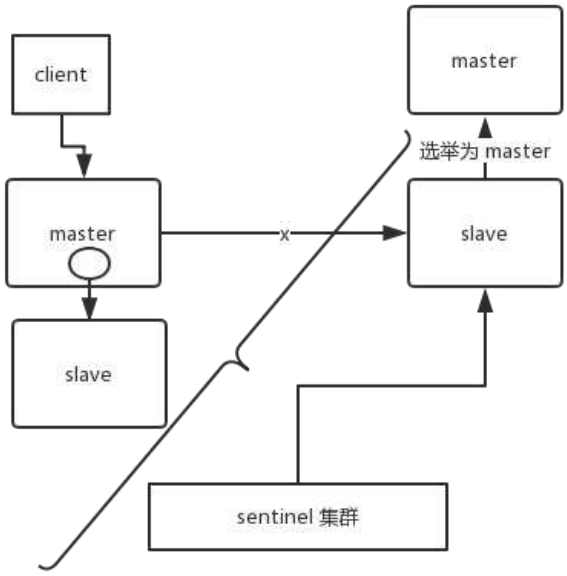


- 脑裂导致的数据丢失

脑裂，也就是说，某个 master 所在机器突然**脱离了正常的网络**，跟其他 slave 机器不能连接，但是实际上 master 还运行着。此时哨兵可能就会**认为** master 宕机了，然后开启选举，将其他 slave 切换成了 master。这个时候，集群里就会有两个 master ，也就是所谓的**脑裂**。

此时虽然某个 slave 被切换成了 master，但是可能 client 还没来得及切换到新的 master，还继续向旧 master 写数据。

因此旧 master 再次恢复的时候，会被作为一个 slave 挂到新的 master 上去，自己的数据会清空，重新从新的 master 复制数据。而新的 master 并没有后来 client 写入的数据，因此，这部分数据也就丢失了。



数据丢失问题的解决方案

进行如下配置：

```
min-slaves-to-write 1
min-slaves-max-lag 10Copy to clipboardErrorCopied
```

表示，要求至少有 1 个 slave，数据复制和同步的延迟不能超过 10 秒。

如果说一旦所有的 slave，数据复制和同步的延迟都超过了 10 秒钟，那么这个时候，master 就不会再接收任何请求了。

- 减少异步复制数据的丢失

有了 `min-slaves-max-lag` 这个配置，就可以确保说，一旦 slave 复制数据和 ack 延时太长，就认为可能 master 宕机后损失的数据太多了，那么就拒绝写请求，这样可以把 master 宕机时由于部分数据未同步到 slave 导致的数据丢失降低的可控范围内。

- 减少脑裂的数据丢失

如果一个 master 出现了脑裂，跟其他 slave 丢了连接，那么上面两个配置可以确保说，如果不能继续给指定数量的 slave 发送数据，而且 slave 超过 10 秒没有给自己 ack 消息，那么就直接拒绝客户端的写请求。因此在脑裂场景下，最多就丢失 10 秒的数据。

29、sdown 和 odown 转换机制

- sdown 是主观宕机，就一个哨兵如果自己觉得一个 master 宕机了，那么就是主观宕机
- odown 是客观宕机，如果 quorum 数量的哨兵都觉得一个 master 宕机了，那么就是客观宕机

sdown 达成的条件很简单，如果一个哨兵 ping 一个 master，超过了 `is-master-down-after-milliseconds` 指定的毫秒数之后，就主观认为 master 宕机了；

如果一个哨兵在指定时间内，收到了 quorum 数量的其它哨兵也认为那个 master 是 sdown 的，那么就认为是 odown 了。

30、哨兵集群的自动发现机制

哨兵互相之间的发现，是通过 Redis 的 pub/sub 系统实现的，每个哨兵都会往 `__sentinel__:hello` 这个 channel 里发送一个消息，这时候所有其他哨兵都可以消费到这个消息，并感知到其他的哨兵的存在。

每隔两秒钟，每个哨兵都会往自己监控的某个 master+slaves 对应的 `__sentinel__:hello` channel 里**发送一个消息**，内容是自己的 host、ip 和 runid 还有对这个 master 的监控配置。

每个哨兵也会去**监听**自己监控的每个 master+slaves 对应的 `__sentinel__:hello` channel，然后去感知到同样在监听这个 master+slaves 的其他哨兵的存在。

每个哨兵还会跟其他哨兵交换对 `master` 的监控配置，互相进行监控配置的同步。

slave 配置的自动纠正

哨兵会负责自动纠正 slave 的一些配置，比如 slave 如果要成为潜在的 master 候选人，哨兵会确保 slave 复制现有 master 的数据；如果 slave 连接到了一个错误的 master 上，比如故障转移之后，那么哨兵会确保它们连接到正确的 master 上。

slave->master 选举算法

如果一个 master 被认为 odown 了，而且 majority 数量的哨兵都允许主备切换，那么某个哨兵就会执行主备切换操作，此时首先要选举一个 slave 来，会考虑 slave 的一些信息：

- 跟 master 断开连接的时长
- slave 优先级
- 复制 offset
- run id

如果一个 slave 跟 master 断开连接的时间已经超过了 down-after-milliseconds 的 10 倍，外加 master 宕机的时长，那么 slave 就被认为不适合选举为 master。

```
(down-after-milliseconds * 10) +
milliseconds_since_master_is_in_SDOWN_stateCopy to clipboardErrorCopied
```

接下来会对 slave 进行排序：

- 按照 slave 优先级进行排序，slave priority 越低，优先级就越高。
- 如果 slave priority 相同，那么看 replica offset，哪个 slave 复制了越多的数据，offset 越靠后，优先级就越高。
- 如果上面两个条件都相同，那么选择一个 run id 比较小的那个 slave。

quorum 和 majority

每次一个哨兵要做主备切换，首先需要 quorum 数量的哨兵认为 odown，然后选举出一个哨兵来做切换，这个哨兵还需要得到 majority 哨兵的授权，才能正式执行切换。

如果 quorum < majority，比如 5 个哨兵，majority 就是 3，quorum 设置为 2，那么就 3 个哨兵授权就可以执行切换。

但是如果 quorum >= majority，那么必须 quorum 数量的哨兵都授权，比如 5 个哨兵，quorum 是 5，那么必须 5 个哨兵都同意授权，才能执行切换。

configuration epoch

哨兵会对一套 Redis master+slaves 进行监控，有相应的监控的配置。

执行切换的那个哨兵，会从要切换到的新 master（salve->master）那里得到一个 configuration epoch，这就是一个 version 号，每次切换的 version 号都必须是唯一的。

如果第一个选举出的哨兵切换失败了，那么其他哨兵，会等待 failover-timeout 时间，然后接替继续执行切换，此时会重新获取一个新的 configuration epoch，作为新的 version 号。

configuration 传播

哨兵完成切换之后，会在自己本地更新生成最新的 master 配置，然后同步给其他的哨兵，就是通过之前说的 pub/sub 消息机制。

这里之前的 version 号就很重要了，因为各种消息都是通过一个 channel 去发布和监听的，所以一个哨兵完成一次新的切换之后，新的 master 配置是跟着新的 version 号的。其他的哨兵都是根据版本号的大小来更新自己的 master 配置的。

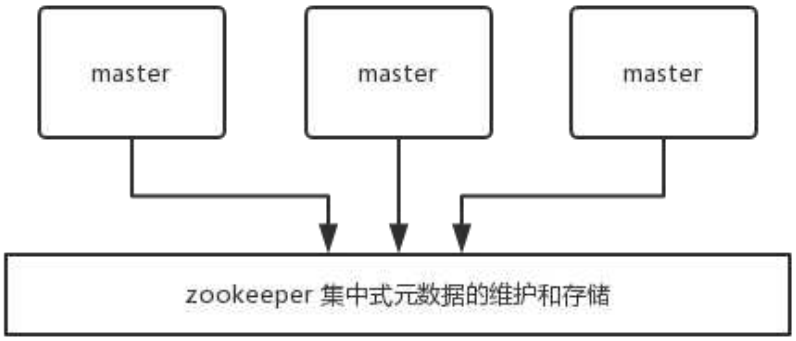
27、Redis 集群模式的工作原理是什么？

基本通信原理

集群元数据的维护有两种方式：集中式、Gossip 协议。Redis cluster 节点间采用 gossip 协议进行通信。

集中式是将集群元数据（节点信息、故障等等）集中存储在某个节点上。集中式元数据集中存储的一个典型代表，就是大数据领域的 storm 。

它是分布式的大数据实时计算引擎，是集中式的元数据存储的结构，底层基于 zookeeper（分布式协调的中间件）对所有元数据进行存储维护。



集中式的**好处**在于，元数据的读取和更新，时效性非常好，一旦元数据出现了变更，就立即更新到集中式的存储中，其它节点读取的时候就可以感知到；**不好**在于，所有的元数据的更新压力全部集中在一个地方，可能会导致元数据的存储有压力。

gossip 好处在于，元数据的更新比较分散，不是集中在一个地方，更新请求会陆陆续续打到所有节点上去更新，降低了压力；不好在于，元数据的更新有延时，可能导致集群中的一些操作会有一些滞后。

- 10000 端口：每个节点都有一个专门用于节点间通信的端口，就是自己提供服务的端口号+10000，比如 7001，那么用于节点间通信的就是 17001 端口。每个节点每隔一段时间都会往另外几个节点发送 `ping` 消息，同时其它几个节点接收到 `ping` 之后返回 `pong` 。
- 交换的信息：信息包括故障信息，节点的增加和删除，hash slot 信息等等。

gossip 协议

gossip 协议包含多种消息，包含 `ping` , `pong` , `meet` , `fail` 等等。

- meet：某个节点发送 meet 给新加入的节点，让新节点加入集群中，然后新节点就会开始与其它节点进行通信。

```
Redis-trib.rb add-nodeCopy to clipboardErrorCopied
```

其实内部就是发送了一个 gossip meet 消息给新加入的节点，通知那个节点去加入我们的集群。

- ping：每个节点都会频繁给其它节点发送 ping，其中包含自己的状态还有自己维护的集群元数据，互相通过 ping 交换元数据。
- pong：返回 ping 和 meet，包含自己的状态和其它信息，也用于信息广播和更新。
- fail：某个节点判断另一个节点 fail 之后，就发送 fail 给其它节点，通知其它节点说，某个节点宕机啦。

ping 消息深入

ping 时要携带一些元数据，如果很频繁，可能会加重网络负担。

每个节点每秒会执行 10 次 ping，每次会选择 5 个最久没有通信的其它节点。当然如果发现某个节点通信延时达到了 `cluster_node_timeout / 2` ，那么立即发送 ping，避免数据交换延时过长，落后的时间太长了。

比如说，两个节点之间都 10 分钟没有交换数据了，那么整个集群处于严重的元数据不一致的情况，就会有问题。所以 `cluster_node_timeout` 可以调节，如果调得比较大，那么会降低 ping 的频率。

每次 ping，会带上自己节点的信息，还有就是带上 1/10 其它节点的信息，发送出去，进行交换。至少包含 3 个其它节点的信息，最多包含 总节点数减 2 个其它节点的信息。

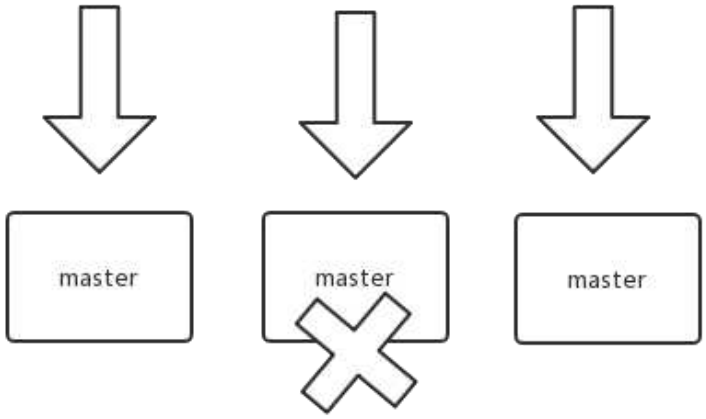
分布式寻址算法

- hash 算法（大量缓存重建）
- 一致性 hash 算法（自动缓存迁移）+ 虚拟节点（自动负载均衡）
- Redis cluster 的 hash slot 算法

hash 算法

来了一个 key，首先计算 hash 值，然后对节点数取模。然后打在不同的 master 节点上。一旦某一个 master 节点宕机，所有请求过来，都会基于最新的剩余 master 节点数去取模，尝试去取数据。

这会导致**大部分的请求过来，全部无法拿到有效的缓存**，导致大量的流量涌入数据库。



一致性 hash 算法

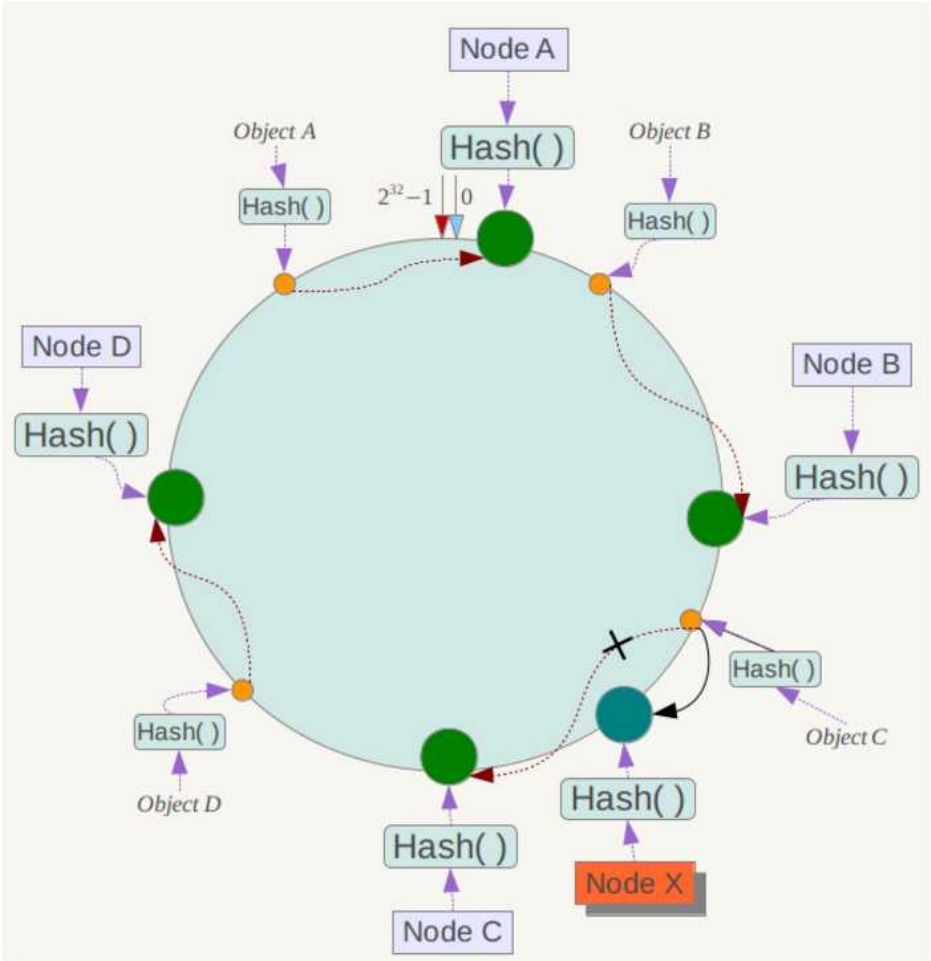
一致性 hash 算法将整个 hash 值空间组织成一个虚拟的圆环，整个空间按顺时针方向组织，下一步将各个 master 节点（使用服务器的 ip 或主机名）进行 hash。这样就能确定每个节点在其哈希环上的位置。

来了一个 key，首先计算 hash 值，并确定此数据在环上的位置，从此位置沿环**顺时针"行走"**，遇到的第一个 master 节点就是 key 所在位置。

在一致性哈希算法中，如果一个节点挂了，受影响的数据仅仅是此节点到环空间前一个节点（沿着逆时针方向行走遇到的第一个节点）之间的数据，其它不受影响。增加一个节点也同理。

燃鹅，一致性哈希算法在节点太少时，容易因为节点分布不均匀而造成**缓存热点**的问题。

为了解决这种热点问题，一致性 hash 算法引入了虚拟节点机制，即对每一个节点计算多个 hash，每个计算结果位置都放置一个虚拟节点。这样就实现了数据的均匀分布，负载均衡。



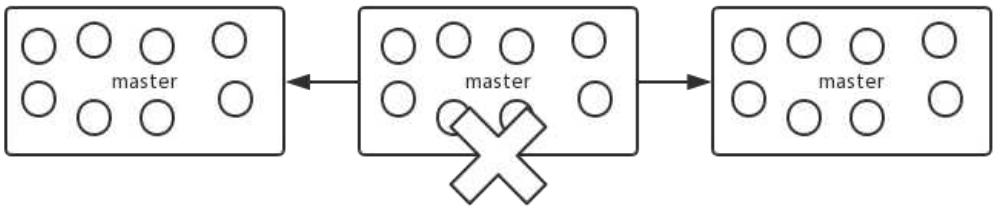
Redis cluster 的 hash slot 算法

Redis cluster 有固定的 16384 个 hash slot，对每个 key 计算 CRC16 值，然后对 16384 取模，可以获取 key 对应的 hash slot。

Redis cluster 中每个 master 都会持有部分 slot，比如有 3 个 master，那么可能每个 master 持有 5000 多个 hash slot。

hash slot 让 node 的增加和移除很简单，增加一个 master，就将其他 master 的 hash slot 移动部分过去，减少一个 master，就将它的 hash slot 移动到其他 master 上去。移动 hash slot 的成本是非常低的。客户端的 api，可以对指定的数据，让他们走同一个 hash slot，通过 hash tag 来实现。

任何一台机器宕机，另外两个节点，不影响的。因为 key 找的是 hash slot，不是机器。



Redis cluster 的高可用与主备切换原理

Redis cluster 的高可用的原理，几乎跟哨兵是类似的。

判断节点宕机

如果一个节点认为另外一个节点宕机，那么就是 `pfail` ，**主观宕机**。如果多个节点都认为另外一个节点宕机了，那么就是 `fail` ，**客观宕机**，跟哨兵的原理几乎一样，sdown，odown。

在 `cluster-node-timeout` 内，某个节点一直没有返回 pong ，那么就被认为 `pfail` 。

如果一个节点认为某个节点 `pfail` 了，那么会在 `gossip ping` 消息中， `ping` 给其他节点，如果**超过半数**的节点都认为 `pfail` 了，那么就会变成 `fail` 。

从节点过滤

对宕机的 master node，从其所有的 slave node 中，选择一个切换成 master node。

检查每个 slave node 与 master node 断开连接的时间，如果超过了 `cluster-node-timeout * cluster-slave-validity-factor` ，那么就**没有资格**切换成 `master` 。

从节点选举

每个从节点，都根据自己对 master 复制数据的 offset，来设置一个选举时间，offset 越大（复制数据越多）的从节点，选举时间越靠前，优先进行选举。

所有的 master node 开始 slave 选举投票，给要进行选举的 slave 进行投票，如果大部分 master node （`N/2 + 1`） 都投票给了某个从节点，那么选举通过，那个从节点可以切换成 master。

从节点执行主备切换，从节点切换为主节点。

与哨兵比较

整个流程跟哨兵相比，非常类似，所以说，Redis cluster 功能强大，直接集成了 replication 和 sentinel 的功能。

28、如何保证缓存与数据库的双写一致性？

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？

Cache Aside Pattern

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**。

为什么是删除缓存，而不是更新缓存？

原因很简单，很多时候，在复杂点的缓存场景，缓存不单单是数据库中直接取出来的值。

比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据并进行运算，才能计算出缓存最新的值的。

另外更新缓存的代价有时候是很高的。是不是说，每次修改数据库的时候，都一定要将其对应的缓存更新一份？也许有的场景是这样，但是对于**比较复杂的缓存数据计算的场景**，就不是这样了。如果你频繁修改一个缓存涉及的多个表，缓存也频繁更新。但是问题在于，**这个缓存到底会不会被频繁访问到？**

举个栗子，一个缓存涉及的表的字段，在 1 分钟内就修改了 20 次，或者是 100 次，那么缓存更新 20 次、100 次；但是这个缓存在 1 分钟内只被读取了 1 次，有**大量的冷数据**。实际上，如果你只是删除缓存的话，那么在 1 分钟内，这个缓存不过就重新计算一次而已，开销大幅度降低，**用到缓存才去算缓存。**

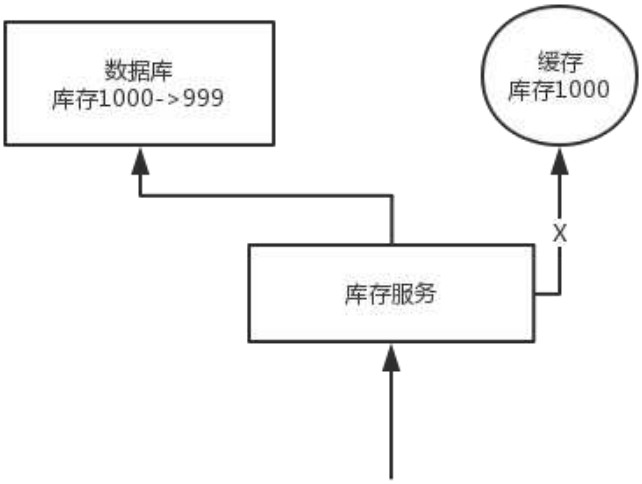
其实删除缓存，而不是更新缓存，就是一个 lazy 计算的思想，不要每次都重新做复杂的计算，不管它会不会用到，而是让它到需要被使用的时候再重新计算。

像 mybatis，hibernate，都有懒加载思想。查询一个部门，部门带了一个员工的 list，没有必要说每次查询部门，都把里面的 1000 个员工的数据也同时查出来啊。

因为 80% 的情况，查这个部门，就只是要访问这个部门的信息就可以了。先查部门，同时要访问里面的员工，那么这个时候只有在你要访问里面的员工的时候，才会去数据库里面查询 1000 个员工。

最初级的缓存不一致问题及解决方案

问题：先更新数据库，再删除缓存。如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据就出现了不一致。



解决思路 1：先删除缓存，再更新数据库。如果数据库更新失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致。因为读的时候缓存没有，所以去读了数据库中的旧数据，然后更新到缓存中。

解决思路 2：延时双删。依旧是先更新数据库，再删除缓存，唯一不同的是，我们把这个删除的动作，在不久之后再执行一次，比如 5s 之后。

```
public void set(key, value) {
    putToDb(key, value);
    deleteFromRedis(key);

    // ... a few seconds later
    deleteFromRedis(key);
}Copy to clipboardErrorCopied
```

删除的动作，可以有多种选择，比如：

- 使用 DelayQueue，会随着 JVM 进程的死亡，丢失更新的风险；
- 放在 MQ，但编码复杂度为增加。

总之，我们需要综合各种因素去做设计，选择一个最合理的解决方案。

比较复杂的数据不一致问题分析

数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改。一个请求过来，去读缓存，发现缓存空了，去查询数据库，**查到了修改前的旧数据**，放到了缓存中。

随后数据变更的程序完成了数据库的修改。完了，数据库和缓存中的数据不一样了…

为什么上亿流量高并发场景下，缓存会出现这个问题？

只有在对一个数据在并发的进行读写的时候，才可能会出现这种问题。其实如果说你的并发量很低的话，特别是读并发很低，每天访问量就 1 万次，那么很少的情况下，会出现刚才描述的那种不一致的场景。

但是问题是，如果每天的是上亿的流量，每秒并发读是几万，每秒只要有数据更新的请求，就**可能会出现上述的数据库+缓存不一致的情况**。

解决方案如下：

更新数据的时候，根据**数据的唯一标识**，将操作路由之后，发送到一个 jvm 内部队列中。

读取数据的时候，如果发现数据不在缓存中，那么将重新执行"读取数据+更新缓存"的操作，根据唯一标识路由之后，也发送到同一个 jvm 内部队列中。

一个队列对应一个工作线程，每个工作线程**串行**拿到对应的操作，然后一条一条的执行。

这样的话，一个数据变更的操作，先删除缓存，然后再去更新数据库，但是还没完成更新。此时如果一个读请求过来，没有读到缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，然后同步等待缓存更新完成。

这里有一个**优化点**，一个队列中，其实**多个更新缓存请求串在一起是没意义的**，因此可以做过滤，如果发现队列中已经有一个更新缓存的请求了，那么就不用再放个更新请求操作进去了，直接等待前面的更新操作请求完成即可。

待那个队列对应的工作线程完成了上一个操作的数据库的修改之后，才会去执行下一个操作，也就是缓存更新的操作，此时会从数据库中读取最新的值，然后写入缓存中。

如果请求还在等待时间范围内，不断轮询发现可以取到值了，那么就直接返回；如果请求等待的时间超过一定时长，那么这一次直接从数据库中读取当前的旧值。

高并发的场景下，该解决方案要注意的问题：

- 读请求长时阻塞

由于读请求进行了非常轻度的异步化，所以一定要注意读超时的问题，每个读请求必须在超时时间范围内返回。

该解决方案，最大的风险点在于说，**可能数据更新很频繁**，导致队列中积压了大量更新操作在里面，然后**读请求会发生大量的超时**，最后导致大量的请求直接走数据库。务必通过一些模拟真实的测试，看看更新数据的频率是怎样的。

另外一点，因为一个队列中，可能会积压针对多个数据项的更新操作，因此需要根据自己的业务情况进行测试，可能需要**部署多个服务**，每个服务分摊一些数据的更新操作。

如果一个内存队列里居然会挤压 100 个商品的库存修改操作，每个库存修改操作要耗费 10ms 去完成，那么最后一个商品的读请求，可能等待 10 * 100 = 1000ms = 1s 后，才能得到数据，这个时候就导致**读请求的长时阻塞**。

一定要做根据实际业务系统的运行情况，去进行一些压力测试，和模拟线上环境，去看看最繁忙的时候，内存队列可能会挤压多少更新操作，可能会导致最后一个更新操作对应的读请求，会 hang 多少时间，如果读请求在 200ms 返回，如果你计算过后，哪怕是最繁忙的时候，积压 10 个更新操作，最多等待 200ms，那还可以的。

如果一个内存队列中可能积压的更新操作特别多，那么你就要**加机器**，让每个机器上部署的服务实例处理更少的数据，那么每个内存队列中积压的更新操作就会越少。

其实根据之前的项目经验，一般来说，数据的写频率是很低的，因此实际上正常来说，在队列中积压的更新操作应该是很少的。像这种针对读高并发、读缓存架构的项目，一般来说写请求是非常少的，每秒的 QPS 能到几百就不错了。

我们来**实际粗略测算一下**。

如果一秒有 500 的写操作，如果分成 5 个时间片，每 200ms 就 100 个写操作，放到 20 个内存队列中，每个内存队列，可能就积压 5 个写操作。每个写操作性能测试后，一般是在 20ms 左右就完成，那么针对每个内存队列的数据的读请求，也就最多 hang 一会儿，200ms 以内肯定能返回了。

经过刚才简单的测算，我们知道，单机支撑的写 QPS 在几百是没问题的，如果写 QPS 扩大了 10 倍，那么就扩容机器，扩容 10 倍的机器，每个机器 20 个队列。

- 读请求并发量过高

这里还必须做好压力测试，确保恰巧碰上上述情况的时候，还有一个风险，就是突然间大量读请求会在几十毫秒的延时 hang 在服务上，看服务能不能扛的住，需要多少机器才能扛住最大的极限情况的峰值。

但是因为并不是所有的数据都在同一时间更新，缓存也不会同一时间失效，所以每次可能也就是少数数据的缓存失效了，然后那些数据对应的读请求过来，并发量应该也不会特别大。

- 多服务实例部署的请求路由

可能这个服务部署了多个实例，那么必须**保证**说，执行数据更新操作，以及执行缓存更新操作的请求，都通过 Nginx 服务器**路由到相同的服务实例上**。

比如说，对同一个商品的读写请求，全部路由到同一台机器上。可以自己去做服务间的按照某个请求参数的 hash 路由，也可以用 Nginx 的 hash 路由功能等等。

- 热点商品的路由问题，导致请求的倾斜

万一某个商品的读写请求特别高，全部打到相同的机器的相同的队列里面去了，可能会造成某台机器的压力过大。

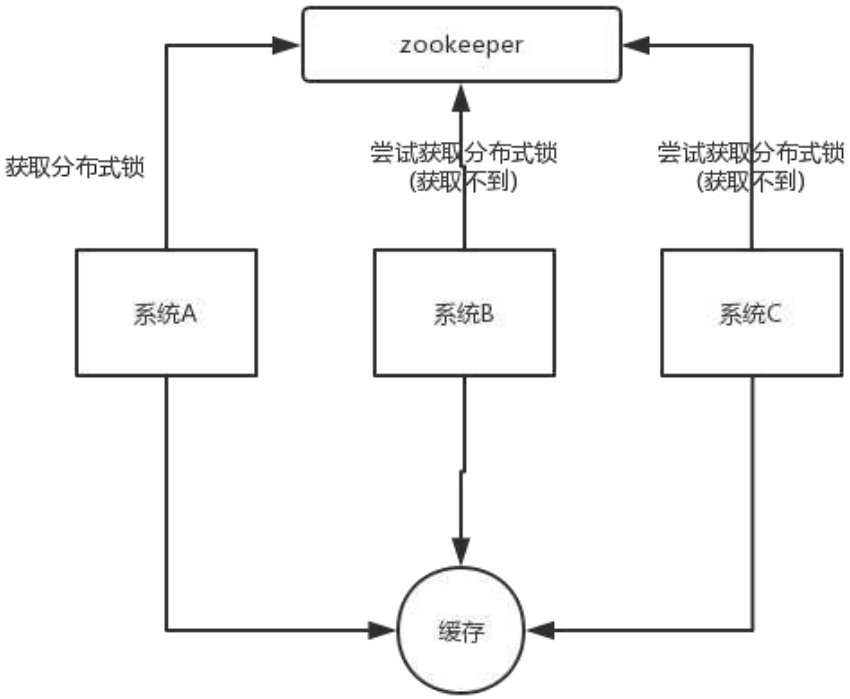
就是说，因为只有在商品数据更新的时候才会清空缓存，然后才会导致读写并发，所以其实要根据业务系统去看，如果更新频率不是太高的话，这个问题的影响并不是特别大，但是的确可能某些机器的负载会高一些。

29、Redis 的并发竞争问题是什么？如何解决这个问题？了解 Redis 事务的 CAS 方案吗？

这个也是线上非常常见的一个问题，就是**多客户端同时并发写**一个 key，可能本来应该先到的数据后到了，导致数据版本错了；或者是多客户端同时获取一个 key，修改值之后再写回去，只要顺序错了，数据就错了。

而且 Redis 自己就有天然解决这个问题的 CAS 类的乐观锁方案。

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。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。



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。

30、生产环境中的 Redis 是怎么部署的？

这个问题主要是想看看你了解不了解你们公司的 Redis 生产集群的部署架构。

如果你不了解，那么确实你就很失职了，你的 Redis 是主从架构？集群架构？用了哪种集群方案？有没有做高可用保证？有没有开启持久化机制确保可以进行数据恢复？线上 Redis 给几个 G 的内存？设置了哪些参数？压测后你们 Redis 集群承载多少 QPS？

Redis cluster，10 台机器，5 台机器部署了 Redis 主实例，另外 5 台机器部署了 Redis 的从实例，每个主实例挂了一个从实例，5 个节点对外提供读写服务，每个节点的读写高峰 QPS 可能可以达到每秒 5 万，5 台机器最多是 25 万读写请求每秒。

机器是什么配置？32G 内存+ 8 核 CPU + 1T 磁盘，但是分配给 Redis 进程的是 10g 内存，一般线上生产环境，Redis 的内存尽量不要超过 10g，超过 10g 可能会有问题。

5 台机器对外提供读写，一共有 50g 内存。

因为每个主实例都挂了一个从实例，所以是高可用的，任何一个主实例宕机，都会自动故障迁移，Redis 从实例会自动变成主实例继续提供读写服务。

你往内存里写的是什么数据？每条数据的大小是多少？商品数据，每条数据是 10kb。100 条数据是 1mb，10 万条数据是 1g。常驻内存的是 200 万条商品数据，占用内存是 20g，仅仅不到总内存的 50%。目前高峰期每秒就是 3500 左右的请求量。

其实大型的公司，会有基础架构的 team 负责缓存集群的运维。