


MySQL

Contents

- Contents
-  概述
 - Redis 为什么快?
 - 为什么 Redis 是单线程?
 - Redis 为什么要引入多线程?
 - 为什么用 Redis 作为 MySQL 的缓存?
 - Redis 和 Memcached 的联系和区别?
 - 共同点
 - 区别
 - 如何理解 Redis 原子性操作原理?
-  数据结构
 - Redis 数据类型?
 - STRING
 - LIST
 - HASH
 - SET
 - Zset
 - BitMap
 - HyperLogLog
 - dGEO
 - Stream
 - Redis 底层数据结构?
 - SDS
 - 链表
 - 压缩列表###
 - 哈希
 - 跳表
 - 整数集合
 - quicklist
 - listpack

- 为什么用跳表而不用平衡树？
- 持久化
 - AOF 和 RDB?
 - AOF
 - RDB
 - AOF-RDB 混用
 - AOF 的三种写回策略？
 - AOF 的磁盘重写机制？
 - 为什么先执行 Redis 命令，再把数据写入 AOF 日志呢？
 - 好处
 - 缺陷
 - AOF 的重写的具体过程？
 - AOF 子进程的内存数据跟主进程的内存数据不一致怎么办？
 - RDB 在执行快照的时候，数据能修改吗？
 - Redis 过期机制？
 - Redis 的内存淘汰策略？
 - 不进行数据淘汰的策略
 - 进行数据淘汰的策略
 - Redis 持久化时对过期键会如何处理的？
 - RDB
 - AOF
 - Redis 主从模式中，对过期键会如何处理？
- 应用
 - 缓存雪崩、击穿、穿透和解决办法？
 - 缓存雪崩
 - 缓存击穿
 - 缓存穿透
 - 布隆过滤器是怎么工作的？
 - 如何保证数据库和缓存的一致性？
 - 如何保证删除缓存操作一定能成功？
 - 业务一致性要求高怎么办？
 - 如何避免缓存失效？
 - 如何实现延迟队列？
 - 如何设计一个缓存策略，可以动态缓存热点数据呢？
 - Redis 实现分布式锁？
 - 如何保证加锁和解锁过程的原子性？
 - 使用 Redis 实现分布式锁的优点和缺点？
 - 如何为分布式锁设置合理的超时时间？

- Redis 解决集群情况下分布式锁的可靠性?
- Redis 管道有什么用?
- Redis 如何处理大 key?
 - 定义
 - 影响
 - 处理
- Redis 支持事务回滚吗?
-  集群
 - Redis 集群模式有哪些?
 - Redis 切片集群的工作原理?
 - 哈希槽和 Redis 节点是如何对应的?
 - 主从模式的同步过程?
 - 第一次同步
 - 命令传播
 - 压力分摊
 - 增量复制
 - 主服务器如何知道要将哪些增量数据发送给从服务器?
 - 如何避免主从数据的不一致?
 - 主从架构中过期 key 如何处理?
 - 主从模式是同步复制还是异步复制?
 - 哨兵机制是什么?
 - 哨兵机制的工作原理?
 - 判断节点是否存活
 - 投票
 - 选出新主节点
 - 更换主节点
 - 通知客户的主节点已更换
 - 将旧主节点变为从节点
 - 什么是集群的脑裂?
 - 如何减少主从切换带来的数据丢失?
 - 异步复制同步丢失
 - 集群产生脑裂数据丢失

概述

Redis 为什么快？

- **基于内存操作**：Redis 的绝大部分操作在内存里就可以实现，数据也存在内存中，与传统的磁盘文件操作相比减少了 I/O，提高了操作的速度
- **高效的数据结构**：Redis 有专门设计了 STRING、LIST、HASH 等高效的数据结构，依赖各种数据结构提升了读写的效率
- **采用单线程**：单线程操作省去了上下文切换带来的开销和 CPU 的消耗，同时不存在资源竞争，避免了死锁现象的发生
- **I/O 多路复用**：采用 I/O 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

为什么 Redis 是单线程？

单线程指的是网络请求模块使用单线程进行处理，其他模块仍用多个线程。

官方答案是：因为 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。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。

redis 采用多线程的模块和原因

Redis 在启动的时候，会启动后台线程(BIO)：

- Redis 的早期版本会启动 2 个后台线程，来处理关闭文件、AOF 刷盘这两个任务
- Redis 的后续版本，新增了一个新的后台线程，用来异步释放 Redis 内存。执行 `unlink key / flushdb async / flushall async` 等命令，会把这些删除操作交给后台线程来执行

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很耗时，把这些任务都放在主线程来处理会导致主线程阻塞，导致无法处理后续的请求。后台线程相当于一个消费者，生产者把耗时任务丢到任务队列中，消费者不停轮询这个队列，拿出任务去执行对应的方法即可。

Redis 为什么要引入多线程？

因为 Redis 的瓶颈不在内存，而是在网络 I/O 模块带来 CPU 的耗时，所以 Redis6.0 的多线程是用来处理网络 I/O 这部分，充分利用 CPU 资源，减少网络 I/O 阻塞带来的性能损耗。Redis 引入的多线程 I/O 特性对性能提升至少是一倍以上。

为什么用 Redis 作为 MySQL 的缓存？

MySQL 是数据库系统，对于数据的操作需要访问磁盘，而将数据放在 Redis 中，需要访问就可以直接从内存获取，避免磁盘 I/O，提高操作的速度。

使用 Redis + MySQL 结合的方式可以有效提高系统 QPS。

Redis 和 Memcached 的联系和区别？

共同点

- 都是内存数据库
- 性能都非常高
- 都有过期策略

区别

- **线程模型**：Memcached 采用多线程模型，并且基于 I/O 多路复用技术，主线程接收到请求后分发给子线程处理，这样做好处是：当某个请求处理比较耗时，不会影响到其他请求的处理。缺点是 CPU 的多线程切换存在性能损耗，同时，多线程在访问共享资源时要加锁，也会在一定程度上降低性能；Redis 也采用 I/O 多路复用技术，但它处理请求采用是单线程模型，从接收请求到处理数据都在一个线程中完成。这意味着使用 Redis 一旦某个请求处理耗时比较长，那么整个 Redis 就会阻塞住，直到这个请求处理完成后返回，才能处理下一个请求，使用 Redis 时一定要避免复杂的耗时操作，单线程的好处是，少了 CPU 的上下文切换损耗，没有了多线程访问资源的锁竞争，但缺点是无法利用 CPU 多核的性能
- **数据结构**：Memcached 支持的数据结构很单一，仅支持 string 类型的操作。并且对于 value 的大小限制必须在 1MB 以下，过期时间不能超过 30 天；而 Redis 支持的数据结构非常丰富，除了常用的数据类型 string、list、hash、set、zset 之外，还可以使用 geo、hyperLogLog 数据类型；使用 Memcached 时，我们只能把数据序列化后写入到 Memcached 中。然后再从 Memcached 中读取数据，再反序列化为我们需要的格式，只能“整存整取”；Redis 提供的数据结构提升了操作的便利性
- **淘汰策略**：Memcached 必须设置整个实例的内存上限，数据达到上限后触发 LRU 淘汰机制，优先淘汰不常用使用的数据。它的数据淘汰机制存在一些问题：刚写入的数据可能会被优先淘汰掉，这个问题主要是它本身内存管理设计机制导致的；Redis 没有限制必须设置内存上限，如果内存足够使用，Redis 可以使用足够大的内存，同时 Redis 提供了多种内存淘汰策略
- **持久化**：Memcached 不支持数据的持久化，如果 Memcached 服务宕机，那么这个节点的数据将全部丢失。Redis 支持 AOF 和 RDB 两种持久化方式

- **集群**：Memcached 没有主从复制架构，只能单节点部署，如果节点宕机，那么该节点数据全部丢失，业务需要对这种情况做兼容处理，当某个节点不可用时，把数据写入到其他节点以降低对业务的影响；Redis 拥有主从复制架构，主节点可以实时同步主的数据，提高整个 Redis 服务的可用性

如何理解 Redis 原子性操作原理？

- **API**：Redis 提供的 API 都是单线程串行处理的
- **网络模型**：采用单线程的 epoll 的网络模型，用来处理多个 Socket 请求
- **请求处理**：Redis 会 fork 子进程来出来类似于 RDB 和 AOF 的操作，不影响主进程工作

数据结构

Redis 数据类型？

主要有 STRING，LIST，ZSET，SET，HASH。

STRING

String 类型的底层的数据结构实现主要是 SDS(简单动态字符串)。应用场景主要有：

- **缓存对象**：例如可以用 STRING 缓存整个对象的 JSON
- **计数**：Redis 处理命令是单线程，所以执行命令的过程是原子的，因此 String 数据类型适合计数场景，比如计算访问次数、点赞、转发、库存数量等等
- **分布式锁**：可以利用 SETNX 命令
- **共享 Session 信息**：服务器都会去同一个 Redis 获取相关的 Session 信息，解决了分布式系统下 Session 存储的问题

LIST

List 类型的底层数据结构是由**双向链表或压缩列表**实现的：

- 如果列表的元素个数小于 512 个，列表每个元素的值都小于 64 字节，Redis 会使用**压缩列表**作为 List 类型的底层数据结构；
- 如果列表的元素不满足上面的条件，Redis 会使用**双向链表**作为 List 类型的底层数据结构；

在 Redis 3.2 版本之后，List 数据类型底层数据结构只由 quicklist 实现，替代了双向链表和压缩列表。在 Redis 7.0 中，压缩列表数据结构被废弃，由 listpack 来实现。应用场景主要有：

- **微信朋友圈点赞**：要求按照点赞顺序显示点赞好友信息，如果取消点赞，移除对应好友信息
- **消息队列**：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

HASH

Hash 类型的底层数据结构是由压缩列表或哈希表实现的：

- 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；
- 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。

在Redis 7.0 中，压缩列表数据结构被废弃，交由 listpack 来实现。应用场景主要有：

- **缓存对象**：一般对象用 String + Json 存储，对象中某些频繁变化的属性可以考虑抽出来用 Hash 类型存储
- **购物车**：以用户 id 为 key，商品 id 为 field，商品数量为 value，恰好构成了购物车的 3 个要素

SET

Set 类型的底层数据结构是由**哈希表或整数集合**实现的：

- 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

应用场景主要有：

- **点赞**：key 是文章 id，value 是用户 id
- **共同关注**：Set 类型支持交集运算，所以可以用来计算共同关注的好友、公众号等。key 可以是用户 id，value 则是已关注的公众号的 id
- **抽奖活动**：存储某活动中中奖的用户名，Set 类型因为有去重功能，可以保证同一个用户不会中奖两次

提示

Set 的差集、并集和交集的计算复杂度较高，在数据量较大的情况下，如果直接执行这些计算，会导致 Redis 实例阻塞。在主从集群中，为了避免主库因为 Set 做聚合计算(交集、差集、并集)时导

致主库被阻塞，可以选择一个从库完成聚合统计，或者把数据返回给客户端，由客户端来完成聚合统计。

Zset

Zset 类型(Sorted Set，有序集合)可以根据元素的权重来排序，可以自己来决定每个元素的权重值。比如说，可以根据元素插入 Sorted Set 的时间确定权重值，先插入的元素权重小，后插入的元素权重大。应用场景主要有：

- 在面对需要展示最新列表、排行榜等场景时，如果数据更新频繁或者需要分页显示，可以优先考虑使用 Zset
- **排行榜**：有序集合比较典型的使用场景就是排行榜。例如学生成绩的排名榜、游戏积分排行榜、视频播放排名、电商系统中商品的销量排名等

BitMap

bit 是计算机中最小的单位，使用它进行储存将非常节省空间，特别适合一些数据量大且使用二值统计的场景。可以用于签到统计、判断用户登陆态等操作。

HyperLogLog

HyperLogLog 用于基数统计，统计规则是基于概率完成的，不准确，标准误差率是 0.81%。优点是，在输入元素的数量或者体积非常非常大时，所需的内存空间总是固定的、并且很小。比如百万级网页 UV 计数等；

dGEO

主要用于存储地理位置信息，并对存储的信息进行操作。底层是由 Zset 实现的，使用 GeoHash 编码方法实现了经纬度到 Zset 中元素权重分数的转换，这其中的两个关键机制就是「对二维地图做区间划分」和「对区间进行编码」。一组经纬度落在某个区间后，就用区间的编码值来表示，并把编码值作为 Zset 元素的权重分数。

Stream

Redis 专门为消息队列设计的数据类型。相比于基于 List 类型实现的消息队列，有这两个特有的特性：自动生成全局唯一消息 ID，支持以消费组形式消费数据。

之前方法缺陷：不能持久化，无法可靠的保存消息，并且对于离线重连的客户端不能读取历史消息。

Redis 底层数据结构？

SDS

SDS 不仅可以保存文本数据，还可以保存二进制数据。

$O(1)$ 复杂度获取字符串长度，因为有 Len 属性。

不会发生缓冲区溢出，因为 SDS 在拼接字符串之前会检查空间是否满足要求，如果空间不够会自动扩容，所以不会导致缓冲区溢出的问题。

链表

节点是一个双向链表，在双向链表基础上封装了 listNode 这个数据结构。包括链表节点数量 len、以及可以自定义实现的 dup、free、match 函数。

- listNode 链表节点的结构里带有 prev 和 next 指针，获取某个节点的前置节点或后置节点的时间复杂度只需 $O(1)$ ，而且这两个指针都可以指向 NULL，所以链表是无环链表；
- list 结构因为提供了表头指针 head 和表尾节点 tail，所以获取链表的表头节点和表尾节点的时间复杂度只需 $O(1)$ ；

缺陷：

- 链表每个节点之间的内存都是不连续的，无法很好利用 CPU 缓存。能很好利用 CPU 缓存的数据结构是数组，因为数组的内存是连续的
- 保存一个链表节点的值都需要一个链表节点结构头的分配，内存开销较大

压缩列表###

压缩列表是由连续内存块组成的顺序型数据结构，类似于数组。不仅可以利用 CPU 缓存，而且会针对不同长度的数据，进行相应编码，这种方法可以有效地节省内存开销。不能保存过多的元素，否则查询效率就会降低；新增或修改某个元素时，压缩列表占用的内存空间需要重新分配，甚至可能引发连锁更新的问题。

缺陷：

- 空间扩展操作也就是重新分配内存，因此连锁更新一旦发生，就会导致压缩列表占用的内存空间要多次重新分配，直接影响到压缩列表的访问性能
- 如果保存的元素数量增加了，或是元素变大了，会导致内存重新分配，会有连锁更新的问题
- 压缩列表只会用于保存的节点数量不多的场景，只要节点数量足够小，即使发生连锁更新也能接受

哈希

哈希表是一种保存键值对(key-value)的数据结构。优点在于能以 $O(1)$ 的复杂度快速查询数据。Redis 采用了拉链法来解决哈希冲突，在不扩容哈希表的前提下，将具有相同哈希值的数据串起来，形成链接。渐进式哈希的过程如下：

Redis 定义一个 dict 结构体，这个结构体里定义了**两个哈希表(ht[2])**。

- 给 ht2 分配空间
- 在 rehash 进行期间，每次哈希表元素进行新增、删除、查找或者更新操作时，Redis 除了会执行对应的操作之外，还会顺序将 ht1 中索引位置上的所有数据迁移到 ht2 上
- 随着处理客户端发起的哈希表操作请求数量越多，最终在某个时间点会把「哈希表 1」的所有 key-value 迁移到「哈希表 2」，从而完成 rehash 操作

tip 渐进式哈希的触发条件？

负载因子 = 哈希表已保存节点数/哈希表大小

当负载因子大于等于 1，没有执行 RDB 快照或没有进行 AOF 重写的时候，就会进行 rehash 操作。

当负载因子大于等于 5 时，此时说明哈希冲突非常严重了，不管有没有有在执行 RDB 快照或 AOF 重写，都会强制进行 rehash 操作。

跳表

一种多层的有序链表，能快速定位数据。当数据量很大时，跳表的查找复杂度就是 $O(\log N)$ 。

节点同时保存元素和元素的权重，每个跳表节点都有一个后向指针，指向前一个节点，目的是为了从跳表的尾节点开始访问节点，为了倒序查找时方便。跳表是一个带有层级关系的链表，而且每一层级可以包含多个节点，每一个节点通过指针连接起来。Zset 数组中一个属性是 level 数组，一个 level 数组就代表跳表的一层，定义了指向下一个节点的指针和跨度。

tip 跳表的查找过程？

查找一个跳表节点的过程时，跳表会从头节点的最高层开始，逐一遍历每一层。在遍历某一层的跳表节点时，会用跳表节点中的 SDS 类型的元素和元素的权重来进行判断：

- 如果当前节点的权重小于要查找的权重时，跳表就会访问该层上的下一个节点。
- 如果当前节点的权重等于要查找的权重时，并且当前节点的 SDS 类型数据小于要查找的数据时，跳表就会访问该层上的下一个节点。

如果上面两个条件都不满足，或者下一个节点为空时，跳表就会使用目前遍历到的节点的 level 数组里的下一层指针，然后沿着下一层指针继续查找。

跳表的相邻两层的节点数量的比例会影响跳表的查询性能。相邻两层的节点数量最理想的比例是 2:1，查找复杂度可以降低到 $O(\log N)$ 。为了防止插入删除时间消耗，跳表在创建节点的时候，随机生成每个节点的层数。具体的做法是，跳表在创建节点时候，会生成范围为 $[0-1]$ 的一个随机数，如果这个随机数小于 0.25(相当于概率 25%)，那么层数就增加 1 层，然后继续生成下一个随机数，直到随机数的结果大于 0.25 结束，最终确定该节点的层数。

整数集合

整数集合本质上是一块连续内存空间。

整数集合有一个升级规则，就是当将一个新元素加入到整数集合里面，如果新元素的类型(int32_t)比整数集合现有所有元素的类型(int16_t)都要长时，整数集合需要先进行升级，也就是按新元素的类型(int32_t)扩展 contents 数组的空间大小，然后才能将新元素加入到整数集合里，升级的过程中也要维持整数集合的有序性。

quicklist

其实 quicklist 就是双向链表 + 压缩列表组合，quicklist 就是一个链表，而链表中的每个元素又是一个压缩列表。quicklist 解决办法，通过控制每个链表节点中的压缩列表的大小或者元素个数，来规避连锁更新的问题。因为压缩列表元素越少或越小，连锁更新带来的影响就越小，从而提供了更好的访问性能。

listpack

listpack 没有压缩列表中记录前一个节点长度的字段了，listpack 只记录当前节点的长度，当向 listpack 加入一个新元素的时候，不会影响其他节点的长度字段的变化，从而避免了压缩列表的连锁更新问题。

为什么用跳表而不用平衡树？

- **从内存占用上来比较，跳表比平衡树更灵活一些：**平衡树每个节点包含 2 个指针(分别指向左右子树)，而跳表每个节点包含的指针数目平均为 $1/(1 - p)$ ，如果像 Redis 里的实现一样，取 $p = 1/4$ ，那么平均每个节点包含 1.33 个指针，比平衡树更有优势
- **在做范围查找的时候，跳表比平衡树操作要简单：**在平衡树上，找到指定范围的小值之后，还需要以中序遍历的顺序继续寻找其它不超过大值的节点。如果不对平衡树进行一定的改造，这里的中序遍历并不容易实现。而在跳表上进行范围查找就非常简单，只需要在找到小值之后，对第 1 层链表进行若干步的遍历就可以实现
- **从算法实现难度上来比较，跳表比平衡树要简单得多：**平衡树的插入和删除操作可能引发子树的调整，逻辑复杂，而跳表的插入和删除只需要修改相邻节点的指针，操作简单又快速

● 持久化

AOF 和 RDB?

AOF

每执行一条**写操作**命令，就将该命令以追加的方式写入到 AOF 文件，然后在恢复时，以逐一执行命令的方式来进行数据恢复。用 AOF 日志的方式来恢复数据很慢，因为 Redis 执行命令由单线程负责的，AOF 日志恢复数据的方式是顺序执行日志里的每一条命令，如果 AOF 日志很大，这个过程就会很慢了。

RDB

RDB 快照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 快照是全量快照，也就是说每次执行快照，都是把内存中的所有数据都记录到磁盘中。如果频率太频繁，可能会对 Redis 性能产生影响。如果频率太低，服务器故障时，丢失的数据会更多。通常可能设置至少 5 分钟才保存一次快照，这时如果 Redis 出现宕机等情况，意味着最多可能丢失 5 分钟数据。

AOF-RDB 混用

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 AOF 文件。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。

这样的好处在于，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**。缺点是 AOF 文件的可读性变差了。

AOF 的三种写回策略?

Always、Everysec 和 No，这三种策略在可靠性上是从高到低，而在性能上从低到高。

- **Always** 每次写操作命令执行完后，同步将 AOF 日志数据写回硬盘

- **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
- **No** 不控制写回硬盘的时机。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 的磁盘重写机制？

随着执行的命令越多，AOF 文件的体积自然也会越来越大，为了避免日志文件过大，Redis 提供了 AOF 重写机制，它会直接扫描数据中所有的键值对数据，然后为每一个键值对生成一条写操作命令，接着将该命令写入到新的 AOF 文件，重写完成后，就替换掉现有的 AOF 日志。重写的过程是由后台子进程完成的，这样可以使得主进程可以继续正常处理命令。

为什么先执行 Redis 命令，再把数据写入 AOF 日志呢？

好处

- **保证正确写入**：如果当前的命令语法有问题，错误的命令记录到 AOF 日志里后可能还会进行语法检查。先执行 Redis 命令，再把数据写入 AOF 日志可以保证写入的都是正确可执行的命令
- **不阻塞当前写操作**：因为当写操作命令执行成功后才会将命令记录到 AOF 日志，避免写入阻塞

缺陷

- **数据可能会丢失**：执行写操作命令和记录日志是两个过程，Redis 还没来得及将命令写入到硬盘时发生宕机，数据会有丢失的风险
- **阻塞其他操作**：不会阻塞当前命令的执行，但因为 AOF 日志也是在主线程中执行，所以当 Redis 把日志文件写入磁盘的时候，还是会阻塞后续的操作无法执行

AOF 的重写的具体过程？

触发重写机制后，主进程会创建重写 AOF 的子进程，此时父子进程共享物理内存，重写子进程只会对这个内存进行只读。重写 AOF 子进程读取数据库里的所有数据，并逐一把内存数据的键值对转换成一条命令，再将命令记录到重写日志。

在发生写操作的时候，操作系统才会去复制物理内存，这样是为了防止 fork 创建子进程时，由于物理内存数据的复制时间过长而导致父进程长时间阻塞的问题。

AOF 子进程的内存数据跟主进程的内存数据不一致怎么办？

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，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。

tip

Redis 的重写 AOF 过程是由后台子进程 bgrewriteaof 来完成的，这有两个好处：

- 子进程进行 AOF 重写期间，主进程可以继续处理命令请求，从而避免阻塞主进程
- 子进程带有主进程的数据副本，**使用子进程而不是线程**。因为如果是使用线程，多线程之间会共享内存，那么在修改共享内存数据的时候，需要通过加锁来保证数据的安全，而这样就会降低性能。创建子进程时，父子进程是共享内存数据的，不过这个共享的内存只能以只读的方式，而当父子进程任意一方修改了该共享内存，会发生写时复制，于是父子进程就有了独立的数据副本，不用加锁来保证数据安全

RDB 在执行快照的时候，数据能修改吗？

可以。执行 bgsave 过程中，Redis 依然**可以继续处理操作命令**的，数据是能被修改的，采用的是写时复制技术(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，由于共享父进程的所有数据，可以直接读取主线程里的内存数据，并将数据写入到 RDB 文件。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。如果主线程要修改共享数据里的某一块数据，就会发生写时复制，数据的物理内存就会被复制一份，主线程在这个数据副本进行修改操作。与此同时，子进程可以继续把原来的数据写入到 RDB 文件。

Redis 过期机制？

三种过期删除策略：

- **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，由事件处理器执行 key 的删除操作
 - **优点**：内存可以被尽快地释放。定时删除对内存是最友好的

- **缺点**：定时删除策略对 CPU 不友好，删除过期 key 可能会占用相当一部分 CPU 时间，CPU 紧张的情况下将 CPU 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
- **惰性删除**：不主动删除过期键，每次从数据库访问 key 时检测 key 是否过期，如果过期则删除该 key
 - **优点**：只会使用很少的系统资源，对 CPU 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。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**：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

Redis 选择惰性删除+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 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**：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

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；已过期 key 的数量占比随机抽取 key 的数量大于 25%，则继续重复步骤直到比重小于 25%。

Redis 的内存淘汰策略？

不进行数据淘汰的策略

它表示当运行内存超过最大设置内存时，不淘汰任何数据，这时如果有新的数据写入，则会触发 OOM，只是单纯的查询或者删除操作的话还是可以正常工作。

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 持久化时对过期键会如何处理的？

RDB

RDB 分文生成阶段和加载阶段，生成阶段会对 key 进行过期检查，过期的 key 不会保存到 RDB 文件中；加载阶段看服务器是主服务器还是从服务器，如果是主服务器，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；如果从服务器，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。过期键对载入 RDB 文件的从服务器也不会造成影响。

AOF

AOF 文件写入阶段和 AOF 重写阶段。写入阶段如果数据库某个过期键还没被删除，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。重写阶段会对 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。

Redis 主从模式中，对过期键会如何处理？

从库不会进行过期扫描，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。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。

应用

缓存雪崩、击穿、穿透和解决办法？

缓存雪崩

当**大量缓存数据在同一时间过期或者 Redis 故障宕机**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增加，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。

解决方法

- **大量数据同时过期**
 - **均匀设置过期时间**：避免将大量的数据设置成同一个过期时间
 - **互斥锁**：当业务线程在处理用户请求时，如果发现访问的数据不在 Redis 里，就加个互斥锁，保证同一时间内只有一个请求来构建缓存。未能获取互斥锁的请求等待锁释放后重新读取缓存，或者返回空值或者默认值
 - **双 key 策略**：使用两个 key，一个是主 key，设置过期时间，一个是备 key，不会设置过期，key 不一样，但是 value 值是一样。当业务线程访问不到主 key 的缓存数据时，就直接返回备 key 的缓存数据，然后在更新缓存的时候，同时更新主 key 和备 key 的数据
 - **后台更新缓存**：业务线程不再负责更新缓存，缓存也不设置有效期，而是让缓存“永久有效”，并将更新缓存的工作交由后台线程定时更新
- **Redis 故障宕机**
 - **服务熔断或请求限流机制**：启动**服务熔断**机制，**暂停业务应用对缓存服务的访问，直接返回错误**，所以不用再继续访问数据库，保证数据库系统的正常运行，等到 Redis 恢复正常后，再允许业务应用访问缓存服务。服务熔断机制是保护数据库的正常允许，但是暂停了业务应用访问缓存服系统，全部业务都无法正常工作。也可以启用**请求限流**机制，**只将少部分请求发送到数据库进行处理，再多的请求就在入口直接拒绝服务**
 - **构建高可靠集群**：通过**主从节点的方式构建 Redis 缓存高可靠集群**。如果 Redis 缓存的主节点故障宕机，从节点可以切换成为主节点，继续提供缓存服务，避免了由于 Redis 故障宕机而导致的缓存雪崩问题

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。

解决方案：

- **互斥锁方案**：保证同一时间只有一个业务线程更新缓存，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

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。

解决方案

- **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会发生缓存穿透，可以在 API 入口处判断请求参数是否合理，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
- **缓存空值或者默认值**：当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**：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

布隆过滤器是怎么工作的？

布隆过滤器由**初始值都为 0 的位图数组**和 **N 个哈希函数**两部分组成。在写入数据库数据时，在布隆过滤器里做个标记，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。

- 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

缺陷

- 布隆过滤器由于是基于哈希函数实现查找的，会存在哈希冲突的可能性，数据可能落在相同位置，存在误判的情况。查询布隆过滤器说数据存在，并不一定证明数据库中存在这个数据，但是查询到数据不存在，数据库中一定就不存在这个数据
- 不支持一个关键字的删除，因为一个关键字的删除会牵连其他的关键字。改进方法就是 counting Bloom filter，用一个 counter 数组代替位数组，就可以支持删除了
- 对于输入的 n 个元素，要确定数组 m 大小和 hash 函数的个数，hash 函数个数 $k = (\ln 2) * (m/n)$ 时错误率最小。在错误率不大于 E 情况下，m 至少要等于 $n * \lg(1 / E)$ 才能表示 n 个元素的集合。

如何保证数据库和缓存的一致性？

Cache Aside

- **原理**：先从缓存中读取数据，如果没有就再去数据库里面读数据，然后把数据放回缓存中，如果缓存中可以找到数据就直接返回数据；更新数据的时候先把数据持久化到数据库，然后再让缓存失效
- **问题**：假如有两个操作一个更新一个查询，第一个操作先更新数据库，还没来得及删除数据库，查询操作可能拿到的就是旧的数据；更新操作马上让缓存失效了，所以后续的查询可以保证数据的一致性；还有的问题就是有一个是读操作没有命中缓存，然后就到数据库中取数据，此时来了一个写操作，写完数据库后，让缓存失效，然后，之前的那个读操作再把老的数据放进去，也会造成脏数据
- **可行性**：出现上述问题的概率其实非常低，需要同时达成读缓存时缓存失效并且有并发写的操作。数据库读写要比缓存慢得多，所以读操作在写操作之前进入数据库，并且在写操作之后更新，概率比较低

Read/Write Through

- **原理**：Read/Write Through 原理是把更新数据库(Repository)的操作由缓存代理，应用认为后端是一个单一的存储，而存储自己维护自己的缓存。
- **Read Through**：就是在查询操作中更新缓存，也就是说，当缓存失效的时候，Cache Aside策略是由调用方负责把数据加载入缓存，而 Read Through 则用缓存服务自己来加载，从而对调用方是透明的
- **Write Through**：当有数据更新的时候，如果没有命中缓存，直接更新数据库，然后返回。如果命中了缓存，则更新缓存，然后再由缓存自己更新数据库(这是一个同步操作)

Write Behind

- **原理**：在更新数据的时候，只更新缓存，不更新数据库，而缓存会异步地批量更新数据库。这个设计的好处就是让数据的 I/O 操作非常快，带来的问题是，数据不是强一致性的，而且可能会丢
- **第二步失效问题**：这种可能性极小，缓存删除只是标记一下无效的软删除，可以看作不耗时间。如果会出问题，一般程序在写数据库那里就没有完成：故意在写完数据库后，休眠很长时间再来删除缓存

2PC 或是 Paxos

- **2PC 或是 Paxos** 协议保证一致性，因为 2PC 太慢，而 Paxos 太复杂

如何保证删除缓存操作一定能成功？

- **重试机制**：引入消息队列，删除缓存的操作由消费者来做，删除失败的话重新去消息队列拉取相应的操作，超过一定次数没有删除成功就像业务层报错
- **订阅BINLog**：订阅 binlog 日志，拿到具体要操作的数据，然后再执行缓存删除。可以让删除服务模拟自己伪装成一个 MySQL 的从节点，向 MySQL 主节点发送 dump 请求，主节点收到请求后，就会开始推送 BINLog，删除服务解析 BINLog 字节流之后，转换为便于读取的结构化数据，再进行删除

业务一致性要求高怎么办？

- **先更新数据库再更新缓存**：可以先更新数据库再更新缓存，但是可能会有并发更新的缓存不一致的问题。解决办法是更新缓存前加一个分布式锁，保证同一时间只运行一个请求更新缓存，加锁后对于写入的性能就会带来影响；在更新完缓存时，给缓存加上较短的**过期时间**，出现缓存不一致的情况缓存的数据也会很快过期
- **延迟双删**：采用延迟双删，先删除缓存，然后更新数据库，等待一段时间再删除缓存。保证第一个操作再睡眠之后，第二个操作完成更新缓存操作。但是具体睡眠多久其实是个**玄学**，很难评估出来，这个方案也只是**尽可能**保证一致性而已，依然也会出现缓存不一致的现象。

如何避免缓存失效？

- **由后台线程频繁地检测缓存是否有效**：检测到缓存失效了马上从数据库读取数据，并更新到缓存
- **业务线程发现缓存数据失效后**：通过消息队列发送一条消息通知后台线程更新缓存，后台线程收到消息后，在更新缓存前可以判断缓存是否存在，存在就不执行更新缓存操作；不存在就读取数据库数据，并将数据加载到缓存
- **在业务刚上线的时候**：最好提前把数据缓起来，而不是等待用户访问才来触发缓存构建，这就是所谓的**缓存预热**，后台更新缓存的机制刚好也适合干这个事情

如何实现延迟队列？

使用 **ZSet**，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，再利用 `zrangebyscore` 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。

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**。用 `zadd` 方法和 `zrange` 方法来完成排序队列和获取前面商品。

Redis 实现分布式锁？

使用 `SETNX` 命令，只有插入的 key 不存在才插入，如果 `SETNX` 的 key 存在就插入失败，key 插入成功代表加锁成功，否则加锁失败；解锁的过程就是将 key 删除，**保证执行操作的客户端就是加锁的客户端**，加锁时候要设置 `unique_value`，解锁的时候，要先判断锁的 `unique_value` 是否为加锁客户端，是才将 `lock_key` 键删除。此外要给锁设置一个过期时间，以免客户端拿到锁后发生异常，导致锁一直无法释放，可以指定 `EX/PX` 参数设置过期时间。

```
SET lock_key unique_value NX PX 10000
```

如何保证加锁和解锁过程的原子性？

使用 Lua 脚本，因为 Redis 在执行 Lua 脚本时，可以以原子性的方式执行，保证了锁释放操作的原子性。

使用 Redis 实现分布式锁的优点和缺点？

- **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。Redis 主从复制模式中的数据是异步复制的，导致分布式锁的不可靠性。如果在 Redis 主节点获取到锁后，在没有同步到其他节点时，Redis 主节点宕机了，此时新的 Redis 主节点依然可以获取锁，所以多个应用服务就可以同时获取到锁

如何为分布式锁设置合理的超时时间？

可以基于续约的方式设置超时时间：先给锁设置一个超时时间，然后启动一个守护线程，让守护线程在一段时间后，重新设置这个锁的超时时间。实现方式就是：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 解决集群情况下分布式锁的可靠性？

分布式锁算法 Redlock(红锁)。基于**多个 Redis 节点**的分布式锁，即使有节点发生了故障，锁变量仍然是存在的，客户端还是可以完成锁操作。官方推荐是至少部署 5 个 Redis 节点，而且都是主节点，它们之间没有任何关系，都是一个个孤立的节点。

基本思路：是让客户端和多个独立的 Redis 节点依次请求申请加锁，如果客户端能够和半数以上的节点成功地完成加锁操作，那么就认为，客户端成功地获得分布式锁，否则加锁失败。即使有某个 Redis 节点发生故障，锁的数据在其他节点上也有保存，客户端仍然可以正常地进行锁操作，锁的数据也不会丢失。

Redlock 算法加锁三个过程：

- **第一步：**客户端获取当前时间(t1)

- **第二步**：客户端按顺序依次向 N 个 Redis 节点执行加锁操作：加锁操作使用 SET NX，EX/PX 选项，以及带上客户端的唯一标识。如果某个 Redis 节点发生故障了，为了保证在这种情况下，Redlock 算法能够继续运行，需要给「加锁操作」设置一个超时时间，加锁操作的超时时间需要远远地小于锁的过期时间
- **第三步**：一旦客户端从超过半数(大于等于 $N/2+1$)的 Redis 节点上成功获取到了锁，就再次获取当前时间(t_2)，然后计算整个加锁过程的总耗时($t_2 - t_1$)。如果 $t_2 - t_1 < \text{锁的过期时间}$ 认为客户端加锁成功，否则认为加锁失败

加锁成功要同时满足两个条件：有超过半数的 Redis 节点成功的获取到了锁，并且总耗时没有超过锁的有效时间，那么就是加锁成功。

加锁成功后，客户端需要重新计算这把锁的有效时间，计算的结果是「锁最初设置的过期时间」减去「客户端从大多数节点获取锁的总耗时($t_2 - t_1$)」。如果计算的结果已经来不及完成共享数据的操作了，可以释放锁，以免出现还没完成数据操作，锁就过期了的情况。加锁失败后，客户端向**所有 Redis 节点发起释放锁的操作**，执行释放锁的 Lua 脚本就可以。

Redis 管道有什么用？

管道技术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。

但使用管道技术也要注意避免发送的命令过大，或管道内的数据太多而导致的网络阻塞。管道技术本质上是客户端提供的功能，而非 Redis 服务器端的功能。

Redis 如何处理大 key？

定义

String 类型的值大于 10 KB；Hash、List、Set、ZSet 类型的元素的个数超过 5000 个

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。客户端认为很久没有响应
- **引发网络阻塞**：每次获取大 key 产生的网络流量较大
- **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

处理

- 当 value 是 string，比较难拆分，则使用序列化、压缩算法将 key 的大小控制在合理范围内，但是序列化和反序列化都会带来更多时间上的消耗
- 当 value 是 string，压缩之后仍然是大 key，则需要进行拆分，一个大 key 分为不同的部分，记录每个部分的 key，使用 multiget 等操作实现事务读取
- 分拆成几个 key-value，存储在一个 hash 中，每个 field 代表一个具体的属性，使用 hget，hmget 来获取部分的 value，使用 hset，hmset 来更新部分属性
- 当 value 是 list/set 等集合类型时，根据预估的数据规模来进行分片，不同的元素计算后分到不同的片

Redis 支持事务回滚吗？

不支持，Redis 提供的 DISCARD 命令只能用来主动放弃事务执行，把暂存的命令队列清空，起不到回滚的效果。因为 Redis 事务的执行时，错误通常都是编程错误造成的，这种错误通常只会出现在开发环境中，而很少会在实际的生产环境中出现，所以官方认为没有必要为 Redis 开发事务回滚功能。

集群

Redis 集群模式有哪些？

- **主从**：选择一台作为主服务器，将数据到多台从服务器上，构建一主多从的模式，主从之间读写分离。主服务器可读可写，发生写操作会同步给从服务器，而从服务器一般是只读，并接受主服务器同步过来写操作命令。主从服务器之间的命令复制是**异步**进行的，所以无法实现强一致性保证(主从数据时时刻刻保持一致)
- **哨兵**：当 Redis 的主从服务器出现故障宕机时，需要手动进行恢复，为了解决这个问题，Redis 增加了哨兵模式，哨兵监控主从服务器，并且提供**主从节点故障转移的功能**
- **切片集群**：当数据量大到一台服务器无法承载，需要使用 Redis 切片集群(Redis Cluster)方案，它将数据分布在不同的服务器上，以此来降低系统对单主节点的依赖，提高 Redis 服务的读写性能

Redis 切片集群的工作原理？

切片集群会采用哈希槽来进行数据和节点的映射，一个切片集群一共有 16384 个槽位，每个存储数据的 key 会经过运算映射到 16384 个槽位中，映射关系如下：

- 由 key 通过 CRC16 算法计算出一个 16bit 的数字
- 根据上面计算得到的数字对 16384 取模来确定对应的哈希槽

哈希槽和 Redis 节点是如何对应的？

主要有**平均分配**和**手动分配**两种方式。平均分配是集群创建时，Redis 自动将哈希槽平均分配到集群节点上；手动分配是使用命令指定每个节点上面的哈希槽数目，使用手动分配时要把 16384 个槽位给分完，否则集群不会正常工作。

主从模式的同步过程？

第一次同步

主要分为建立**连接协商**、**主从数据同步**、**发送新操作**三个步骤：

- **连接协商**：从服务器先发送命令给主服务器表示要进行数据同步，命令内容包括**主服务器的 runID** 和**复制进度**两个参数，主服务器收到命令之后会给从服务响应命令，响应包括**主服务器的 runID** 和**复制进度**。从服务器收到响应之后会记录这两个值
- **主从数据同步**：主服务器生成 RDB 文件并发送给从服务器，从服务器收到 RDB 之后先清空自己的数据，再载入 RDB 文件。为了主从数据的一致性，这个期间主服务器后续的写操作会记录到 replication buffer 缓冲区里
- **发送新操作**：主服务器发送 replication buffer 里面的写操作给从服务器，从服务器执行这些操作。
第一次同步完成

命令传播

第一次同步完成之后双方会维护一个 TCP 连接，后续主服务器的写命令通过 TCP 连接发送给从服务器，保证主从一致

压力分摊

为了分摊服务器的压力，生成和传输 RDB 的工作可以分摊到经理从服务器上。

增量复制

如果服务器网路断开，在恢复之后，会把网络断开期间主服务器接收到的写操作命令同步给从服务器。

主服务器如何知道要将哪些增量数据发送给从服务器？

网络断开从服务器重新上线之后，会发送自己的复制偏移量到主服务器，主服务器根据偏移量之间的差距判断要执行的操作：如果从服务器要读的数据在 `repl_backlog_buffer` 中，则采用增量复制；如果不在，采用全量复制。

`repl_backlog_buffer`

`repl_backlog_buffer` 是一个**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` 标记缓冲区的同步进度。

如何避免主从数据的不一致？

让主从节点处于同一机房，降低网络延迟；或者由外部程序监控主从复制进度：先计算得出主从服务之间的复制进度差，如果复制进度差大于程序设定的阈值，让客户端不再在此节点读取数据，减小数据不一致的情况对业务的影响。

为了避免出现客户端和所有从节点都不能连接的情况，需要把复制进度差值的阈值设置得大一些。

主从架构中过期 key 如何处理？

主节点处理一个过期的 key 之后就会发送一条删除命令给从服务器，从节点收到命令后进行删除。

主从模式是同步复制还是异步复制？

异步。因为主节点收到写命令之后，先写到内部的缓冲区，然后再异步发送给从节点。

哨兵机制是什么？

因为在主从架构中读写是分离的，如果主节点挂了，将没有主节点来响应客户端的写操作请求，也无法进行数据同步。**哨兵作用是实现主从节点故障转移**。哨兵会监测主节点是否存活，如果发现主节点挂了，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。

哨兵机制的工作原理？

判断节点是否存活

哨兵会周期性给所有主节点发送 PING 命令来判断其他节点是否正常运行。如果 PING 命令响应失败哨兵会将节点标记为**主观下线**，然后该哨兵会向其他节点发出投票命令，当票数达到设定的阈值之后这个主节点就被标记为**客观下线**。然后哨兵会从从节点中选择一个作为主节点。

投票

哨兵集群中会选择一个 leader 来负责主从切换，选举是一个投票过程：判断主节点为**客观下线**的是候选者，候选者向其他哨兵发送命令表示要成为 leader，其他哨兵会进行投票，每个哨兵只有一票，可以投给自己或投给别人，但是只有候选者才能把票投给自己。候选者之后拿到半数以上的赞成票并且票数大于设置的阈值，就会成为候选者。

选出新主节点

把网络状态不好的从节点给排除：先把已经下线的从节点过滤掉，然后把以往网络连接状态不好的从节点排除掉。接下来要对所有从节点进行三轮考察：**优先级、复制进度、ID 号**。在进行每一轮考察的时候，哪个从节点优先胜出，就选择其作为新主节点：

- **第一轮考察**：哨兵首先会根据从节点的优先级来进行排序，优先级越小排名越靠前。
- **第二轮考察**：如果优先级相同，则查看复制的下标，哪个接收的复制数据多哪个就靠前。
- **第三轮考察**：如果优先级和下标都相同，选择 ID 较小的那个。

更换主节点

选出新主节点之后，哨兵 leader 让已下线主节点属下的所有从节点指向新主节点。

通知客户的主节点已更换

客户端和哨兵建立连接后，客户端会订阅哨兵提供的频道。主从切换完成后，哨兵就会向 `+switch-master` 频道发布新主节点的 IP 地址和端口的消息，这个时候客户端就可以收到这条信息，然后用这里面的新主节点的 IP 地址和端口进行通信了。

将旧主节点变为从节点

继续监视旧主节点，当旧主节点重新上线时，哨兵集群就会向它发送 SLAVEOF 命令，让它成为新主节点的从节点。

什么是集群的脑裂？

如果主节点的网络突然发生了问题与所有的从节点都失联了，但此时的主节点和客户端的网络是正常的，客户端不知道集群内部已经出现了问题，还在向这个失联的主节点写数据，此时这些数据被主节点缓存到了缓冲区里。哨兵也发现主节点失联了，就会在从节点中选举出一个 leader 作为主节点，会导致集群有两个主节点。

网络恢复后哨兵因为之前已经选举出一个新主节点了，它就会把旧主节点降级，然后从旧主节点会向新主节点请求数据同步，**因为第一次同步是全量同步的方式，旧主节点会清空掉自己本地的数据。客户端在过程之前写入的数据就会丢失了。**所以脑裂会导致集群数据的丢失。

如何减少主从切换带来的数据丢失？

异步复制同步丢失

配置一个阈值，一旦所有的从节点数据复制和同步的延迟都超过了阈值，主节点就会拒绝接收任何请求。对于客户端发现主节点不可写后，可以采取降级措施。将数据暂时写入本地缓存和磁盘中，在一段时间后重新写入主节点来保证数据不丢失，也可以将数据写入消息队列，等主节点恢复正常，再隔一段时间去消费消息队列中的数据，让将数据重新写入主节点。

集群产生脑裂数据丢失

当主节点发现从节点下线或者通信超时的总数量小于阈值时，那么禁止主节点进行写数据，直接把错误返回给客户端。**设置主节点连接的从节点中至少有 N 个从节点，并且主节点进行数据复制时的 ACK 消息延迟不能超过 T 秒**，否则，主节点就不会再接收客户端的写请求了。等到新主节点上线时，就只有新主节点能接收和处理客户端请求，此时，新写的数据会被直接写到新主节点中。而原主节点会被哨兵降为从节点，即使它的数据被清空了，也不会有新数据丢失。