**认识 Redis**

### 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。

除此之外，Redis 还支持**事务 、持久化、Lua 脚本、多种集群方案（主从复制模式、哨兵模式、切片机群模式）、发布/订阅模式，内存淘汰机制、过期删除机制**等等。

### Redis 和 Memcached 有什么区别？

* Redis **支持的数据类型更丰富**（String、Hash、List、Set、ZSet），而 Memcached 只支持最简单的 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？

***1、Redis 具备高性能***

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。

***2、 Redis 具备高并发***

单台设备的 Redis 的 QPS（Query Per Second，每秒钟处理完请求的次数） 是 MySQL 的 10 倍，Redis 单机的 QPS 能轻松破 10w，而 MySQL 单机的 QPS 很难破 1w。

## Redis 数据结构

### Redis 数据类型以及使用场景分别是什么？

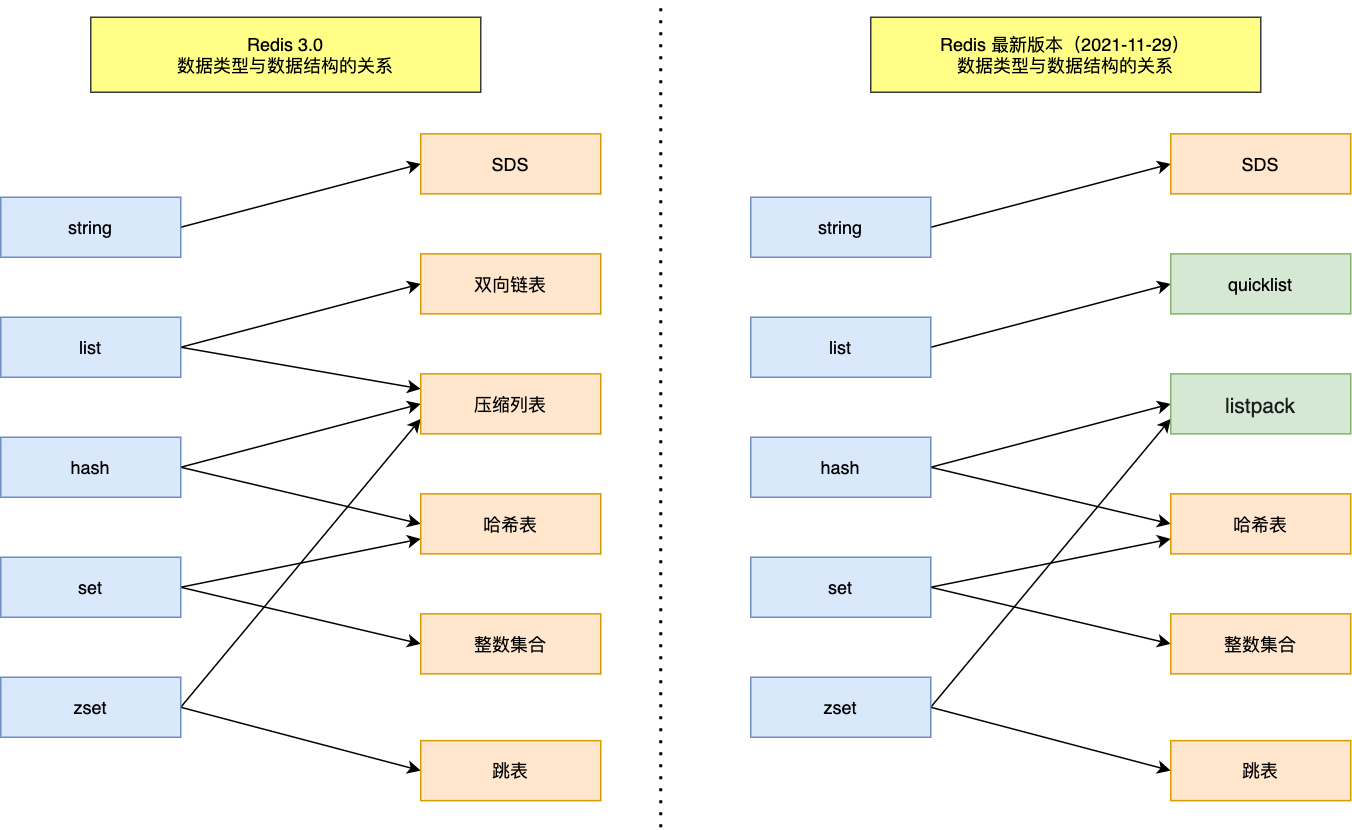
常规类型：

* String 类型：缓存对象、常规计数、分布式锁、共享 session 信息等。
* List 类型：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。

新增类型：

* BitMap（2.2 版新增）：二值状态统计的场景，比如签到、判断用户登陆状态、连续签到用户总数等；
* HyperLogLog（2.8 版新增）：海量数据基数统计的场景，比如百万级网页 UV 计数等；
* GEO（3.2 版新增）：存储地理位置信息的场景，比如滴滴叫车；
* Stream（5.0 版新增）：消息队列，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。

### 五种常见的 Redis 数据类型是怎么实现？



String 类型内部实现

SDS（简单动态字符串）

List 类型内部实现

**3.2版本及以前：双向链表或压缩列表**

**3.2版本后：quicklist**

Hash 类型内部实现

7.0版本前：**压缩列表或哈希表**

7.0版本及以后：listpack

Set 类型内部实现

**哈希表或整数集合**

ZSet 类型内部实现

7.0版本前：**压缩列表或跳表**

7.0版本及以后：**listpack**

## Redis 线程模型

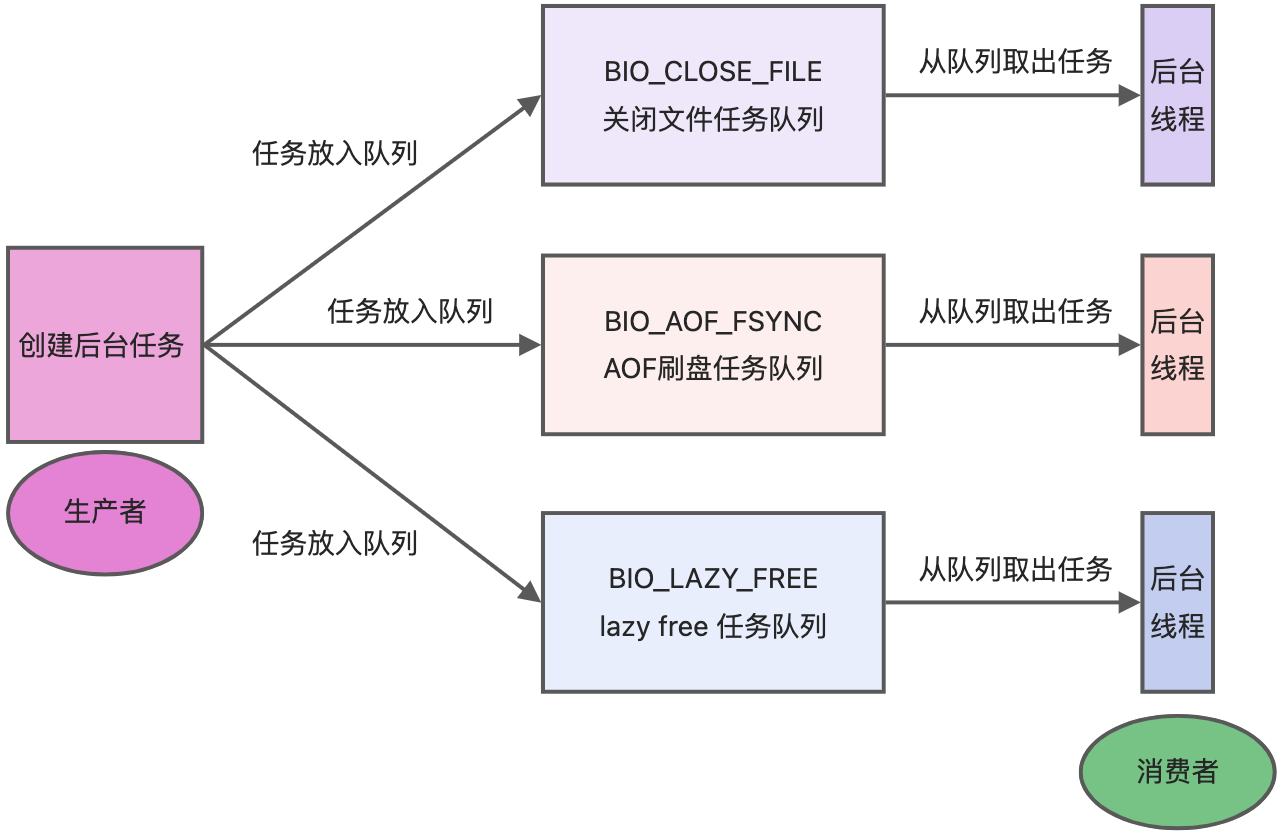
### Redis 是单线程吗？

**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**

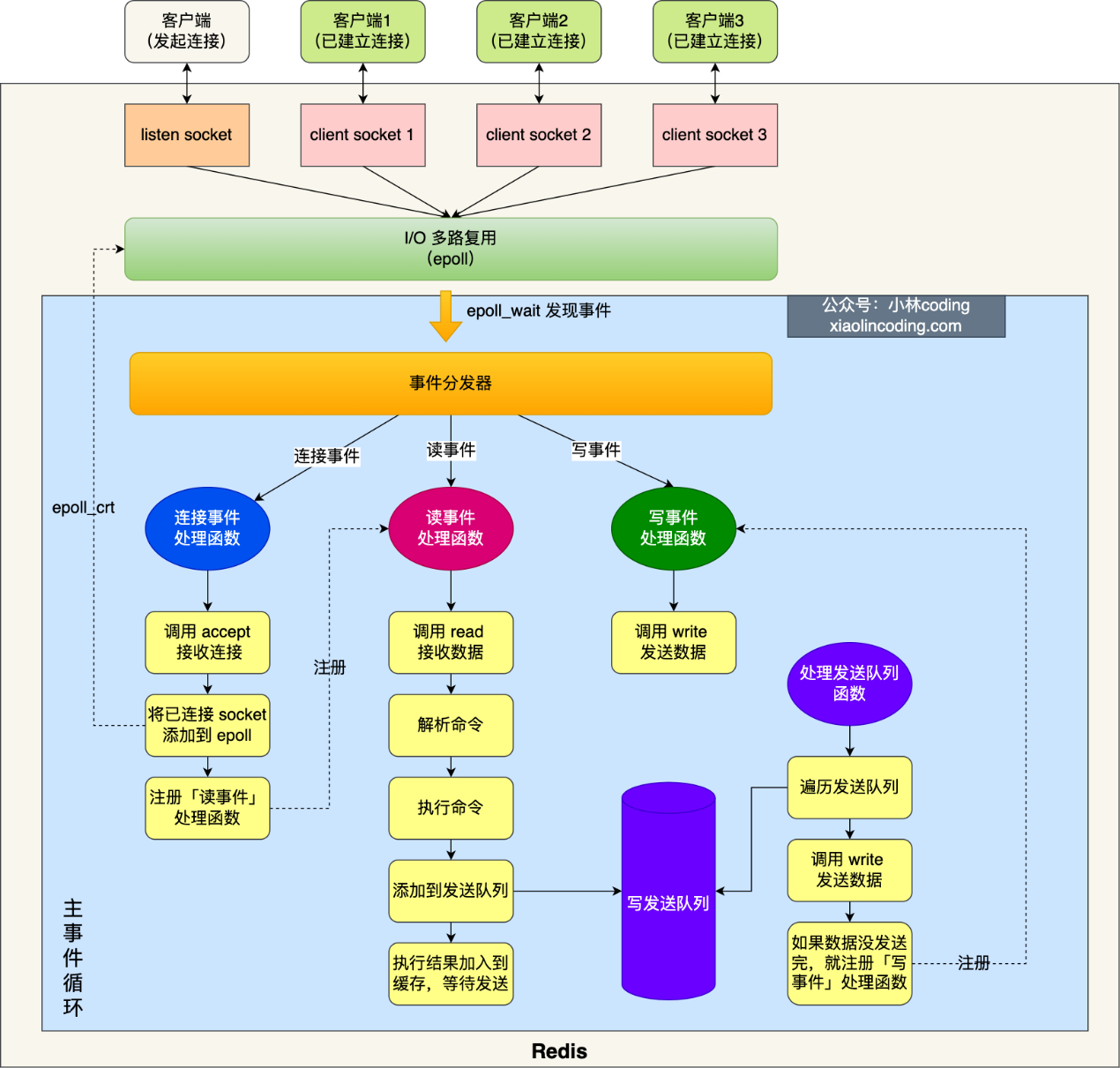
但是**Redis 程序并不是单线程的**

* **Redis 在 2.6 版本**，会启动 2 个后台线程，分别处理关闭文件、AOF 刷盘这两个任务；
* **Redis 在 4.0 版本之后**，新增了一个新的后台线程，用来异步释放 Redis 内存，也就是 lazyfree 线程。

当我们要删除一个大 key 的时候，不要使用 del 命令删除，因为 del 是在主线程处理的，这样会导致 Redis 主线程卡顿，因此我们应该使用 unlink 命令来异步删除大key。



### Redis 单线程模式是怎样的？



edis 初始化的时候，会做下面这几件事情：

* 首先，调用 epoll\_create() 创建一个 epoll 对象和调用 socket() 创建一个服务端 socket
* 然后，调用 bind() 绑定端口和调用 listen() 监听该 socket；
* 最后，将调用 epoll\_ctl() 将 listen socket 加入到 epoll，同时注册「连接事件」处理函数。

初始化完后，主线程就进入到一个**事件循环函数**，主要会做以下事情：

* 首先，先调用**处理发送队列函数**，看是发送队列里是否有任务，如果有发送任务，则通过 write 函数将客户端发送缓存区里的数据发送出去，如果这一轮数据没有发送完，就会注册写事件处理函数，等待 epoll\_wait 发现可写后再处理 。
* 接着，调用 epoll\_wait 函数等待事件的到来：
  + 如果是**连接事件**到来，则会调用**连接事件处理函数**，该函数会做这些事情：调用 accpet 获取已连接的 socket -> 调用 epoll\_ctl 将已连接的 socket 加入到 epoll -> 注册「读事件」处理函数；
  + 如果是**读事件**到来，则会调用**读事件处理函数**，该函数会做这些事情：调用 read 获取客户端发送的数据 -> 解析命令 -> 处理命令 -> 将客户端对象添加到发送队列 -> 将执行结果写到发送缓存区等待发送；
  + 如果是**写事件**到来，则会调用**写事件处理函数**，该函数会做这些事情：通过 write 函数将客户端发送缓存区里的数据发送出去，如果这一轮数据没有发送完，就会继续注册写事件处理函数，等待 epoll\_wait 发现可写后再处理 。

### Redis 采用单线程为什么还这么快？

* Redis 的大部分操作**都在内存中完成**
* 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

### 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。此外，使用了单线程后可维护性高，避免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？

**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**

默认情况下 I/O 多线程只针对发送响应数据（write client socket），并不会以多线程的方式处理读请求（read client socket）。要想开启多线程处理客户端读请求，就需要把 Redis.conf 配置文件中的 io-threads-do-reads 配置项设为 yes。

因此， 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**（*这里的线程数不包括主线程*）：

* 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，io-threads 默认是 4 ，所以会启动 3（4-1）个 I/O 多线程，用来分担 Redis 网络 I/O 的压力。

## Redis 持久化

### Redis 如何实现数据不丢失？

Redis 共有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？

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。

当然，这样做也会带来风险：

* **数据可能会丢失：** 执行写操作命令和记录日志是两个过程，那当 Redis 在还没来得及将命令写入到硬盘时，服务器发生宕机了，这个数据就会有丢失的风险。
* **可能阻塞其他操作：** 由于写操作命令执行成功后才记录到 AOF 日志，所以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 写回策略有几种？

Redis 写入 AOF 日志的过程：

1. Redis 执行完写操作命令后，会将命令追加到 server.aof\_buf 缓冲区；
2. 然后通过 write() 系统调用，将 aof\_buf 缓冲区的数据写入到 AOF 文件，此时数据并没有写入到硬盘，而是拷贝到了内核缓冲区 page cache，等待内核将数据写入硬盘；
3. 具体内核缓冲区的数据什么时候写入到硬盘，由写回策略决定。

Redis 提供了 3 种写回硬盘的策略，控制是第3步的过程：

* **Always**：每次写操作命令执行完后，立即同步将 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**：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。

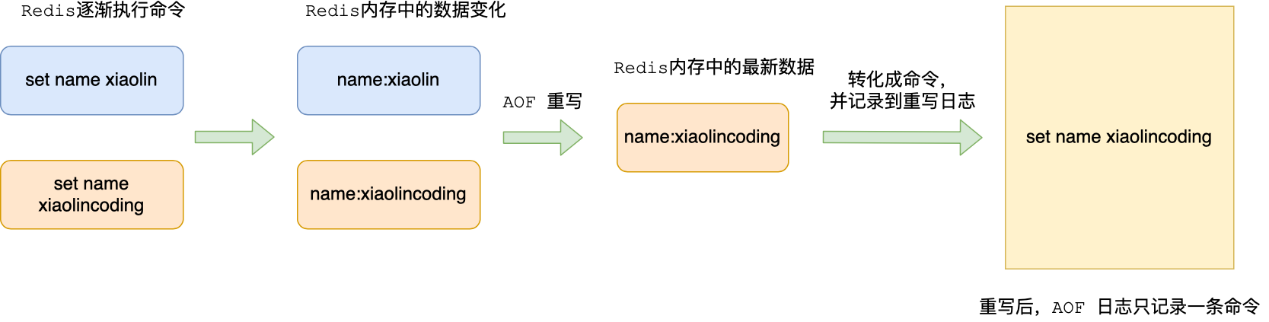


AOF 日志过大，会触发什么机制？

**AOF 重写机制**

当 AOF 文件的大小超过所设定的阈值后，Redis 就会启用 AOF 重写机制压缩 AOF 文件。

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。



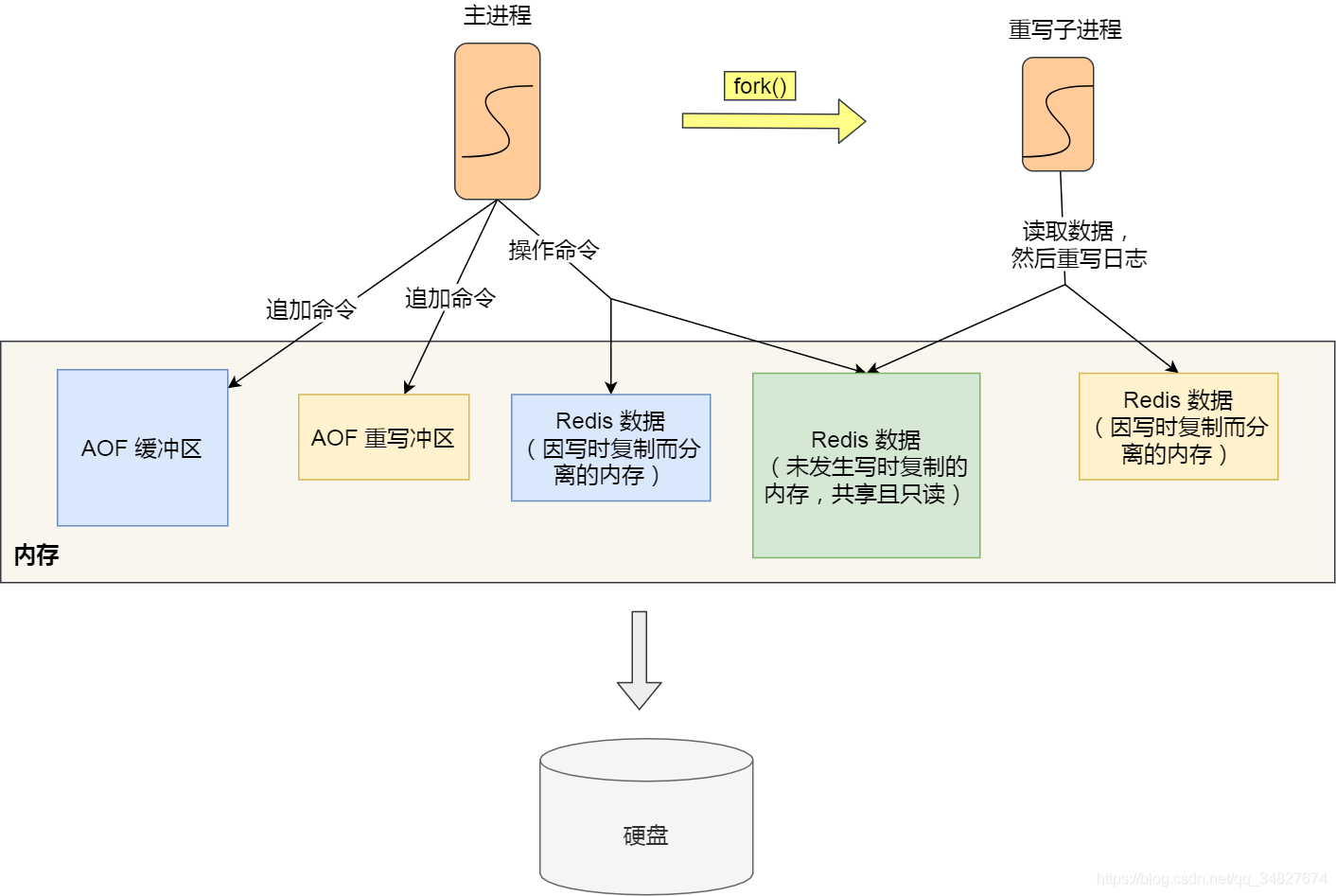
重写 AOF 日志的过程是怎样的？

**重写 AOF 过程是由后台子进程 bgrewriteaof 来完成的。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；
* 使用子进程而不是子线程：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，就会发生「写时复制」，于是父子进程就有了独立的数据副本，就不用加锁来保证数据安全。

**重写过程中，主进程依然可以正常处理命令。**

引入问题：重写 AOF 日志过程中，如果主进程修改了已经存在 key-value，那么会发生写时复制，此时这个 key-value 数据在子进程的内存数据就跟主进程的内存数据不一致了。为了解决这种数据不一致问题，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。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」**



在 bgrewriteaof 子进程执行 AOF 重写期间，主进程需要执行以下三个工作：

* 执行客户端发来的命令；
* 将执行后的写命令追加到 「AOF 缓冲区」；
* 将执行后的写命令追加到 「AOF 重写缓冲区」；

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。

### RDB 快照是如何实现的呢？

AOF 日志记录的是操作命令，故障恢复时，需要全量把日志都执行一遍，当 AOF 日志非常多，会造成 Redis 的恢复操作缓慢。

RDB 快照就是记录某一个瞬间的内存数据，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。

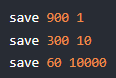
RDB 做快照时会阻塞线程吗？

Redis 提供了两个命令来生成 RDB 文件，分别是 save 和 bgsave：

* 执行了 save 命令，就会在主线程生成 RDB 文件，**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，可以**避免主线程的阻塞**；

Redis 还可以通过配置文件的选项来实现每隔一段时间自动执行一次 **bgsave** 命令。

默认配置：



选项名叫 save，但实际上执行的是 bgsave 命令。满足上面条件的任意一个，就会执行 bgsave：

* 900 秒之内，对数据库进行了至少 1 次修改；
* 300 秒之内，对数据库进行了至少 10 次修改；
* 60 秒之内，对数据库进行了至少 10000 次修改。

Redis 的快照是**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。如果频率太高，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。

RDB 在执行快照的时候，数据能修改吗？

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**

### 为什么会有混合持久化？

Redis 4.0 提出了**混合使用 AOF 日志和内存快照。**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**。

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全量内存数据以 RDB 方式写入到 AOF 文件，写入时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。

**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，可读性变得很差；
* 兼容性差，混合持久化不能用在 Redis 4.0 之前的版本。

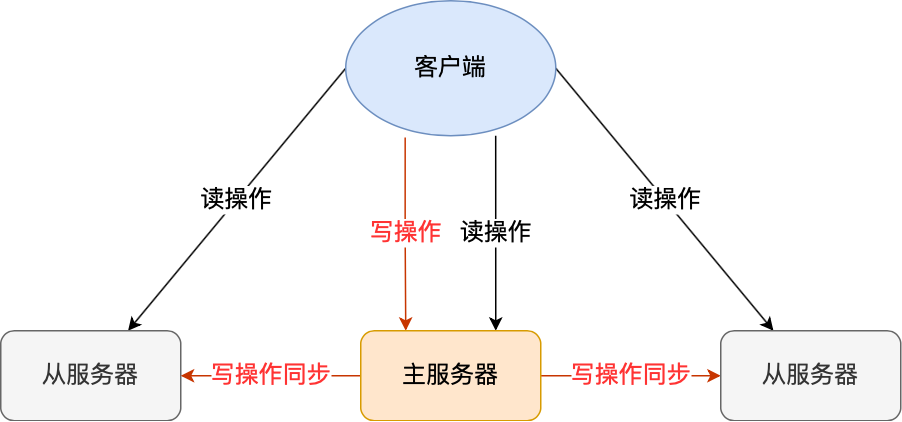
## Redis 集群

### Redis 如何实现服务高可用？

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。

主从复制

采用一主多从的模式，将数据从一台主 Redis 服务器同步数据到多台从 Redis 服务器上。主从服务器之间采用的是「读写分离」的方式。



主服务器：可以进行读写操作，当发生写操作时自动将写操作同步给从服务器

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，然后执行这条命令。

注意：主从服务器之间的命令复制是**异步**进行的，无法实现强一致性保证，数据不一致是难以避免的

主服务器自己在本地执行完命令后，就会向客户端返回结果了。如果从服务器还没有执行主服务器同步过来的命令，主从服务器间的数据就不一致了。

哨兵模式

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（**Redis Sentinel**），可以监控主从服务器，并且提供**主从节点故障转移的功能。**

切片集群模式

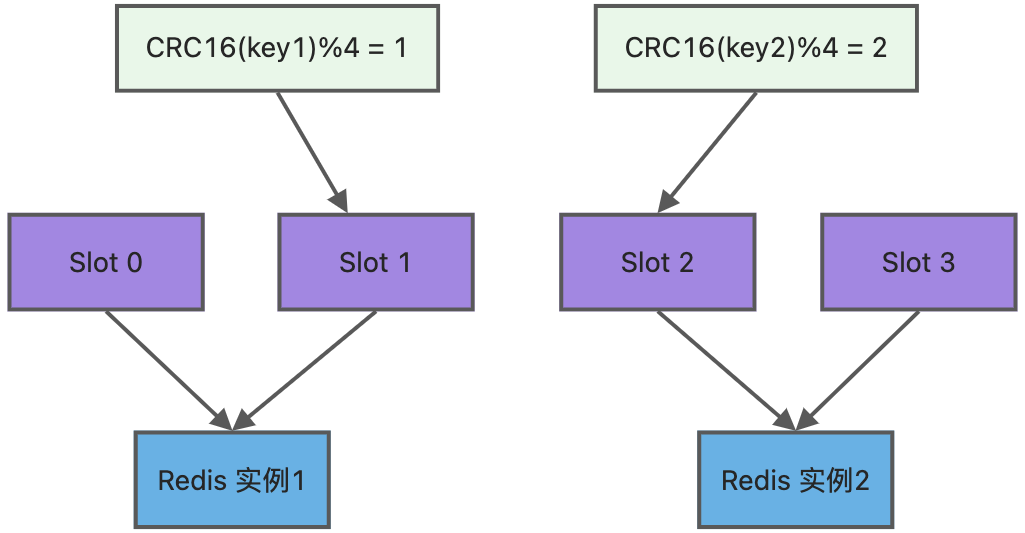
Redis 缓存数据量大到一台服务器无法缓存时，就需要使用 **Redis 切片集群**（Redis Cluster ）方案，它将数据分布在不同的服务器上

数据和哈希槽之间的映射关系：Redis Cluster 方案采用哈希槽（Hash Slot），Redis Cluster 方案中，**一个切片集群共有 16384 个哈希槽**：

* 根据键值对的 key，按照 CRC16 算法计算一个 16 bit 的值。
* 再用 16bit 值对 16384 取模，得到 0~16383 范围内的模数，每个模数代表一个相应编号的哈希槽。

哈希槽和节点之间的映射关系：两种方案：

* **平均分配：** 使用 cluster create 命令创建 Redis 集群时，Redis 会自动把所有哈希槽平均分布到集群节点上。比如集群中有 9 个节点，则每个节点上槽的个数为 16384/9 个。
* **手动分配：** 可以使用 cluster meet 命令手动建立节点间的连接，组成集群，再使用 cluster addslots 命令，手动指定每个节点上的哈希槽个数（**注意：在手动分配哈希槽时，需要把 16384 个槽都分配完，否则 Redis 集群无法正常工作**）。



### 集群脑裂导致数据丢失怎么办？

什么是脑裂？

主节点的网络发生了问题，与所有的从节点都失联了，唯独与客户端的连接正常。此时客户端向主节点写数据（称为**过程A**），无法同步到从节点。这时，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，这时集群就有两个主节点了，称为集群脑裂。

网络恢复后，把旧主节点降级为从节点，然后这个从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，此时的从节点会清空掉自己本地的数据，然后再做全量同步。所以，之前客户端在过程 A 写入的数据就会丢失了，也就是集群产生脑裂数据丢失的问题**。

解决方案

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，禁止主节点进行写数据，直接把错误返回给客户端。

相关的参数：

* min-slaves-to-write x：主节点必须要有至少 x 个从节点连接，如果小于这个数，主节点会禁止写数据。
* min-slaves-max-lag x：主从数据复制和同步的延迟不能超过 x 秒，如果超过，主节点会禁止写数据。

## Redis 过期删除与内存淘汰

### Redis 使用的过期删除策略是什么？

Redis 可以对 key 设置过期时间，Redis 会把该 key 带上过期时间存储到一个**过期字典**（expires dict）中。查询一个 key 时，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需要过期键值删除策略将已过期的键值对删除。Redis 使用的过期删除策略是「**惰性删除+定期删除**」两种策略配合使用，在合理使用 CPU 时间和避免内存浪费之间取得平衡。

惰性删除策略

**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。**

**优点：**访问时才会检查，占用比较少系统资源

**缺点：**只要过期 key 一直没有被访问，它所占用的内存就不会释放，造成空间浪费。

定期删除策略

**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：**能主动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：**执行的太频繁，对 CPU 不友好；如果执行的太少，过期 key 占用的内存不会及时得到释放

### Redis 持久化时，对过期键会如何处理的？

RDB文件

* **生成阶段**：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。
* **加载阶段**：RDB 加载阶段时：
  + **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**。所以过期键不会对载入 RDB 文件的主服务器造成影响；
  + **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。所以一般来说，过期键对载入 RDB 文件的从服务器也不会造成影响。

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**。
* **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，对过期键会如何处理？

**从库不会进行过期扫描，从库对过期的处理是被动的**。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

**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？

运行内存达到阀值，就会触发**内存淘汰机制**，这个阀值就是在 Redis 的配置文件中的配置项 maxmemory。

### Redis 内存淘汰策略有哪些？

① 不淘汰：

**noeviction**（Redis3.0之后，默认的内存淘汰策略） ：它表示当运行内存超过最大设置内存时，不淘汰任何数据，而是不再提供服务，直接返回错误。

② 在设置了过期时间的数据中淘汰：

**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**（Redis3.0 之前，默认的内存淘汰策略）：淘汰最久未使用的设置了过期时间的键值

**volatile-lfu**（Redis 4.0 后新增的内存淘汰策略）：淘汰最少使用的设置了过期时间的键值

③ 在所有数据范围内淘汰：

**allkeys-random**：随机淘汰任意键值

**allkeys-lru**：所有键值中淘汰最久未使用的键值

**allkeys-lfu**（Redis 4.0 后新增的内存淘汰策略）：所有键值中淘汰最少使用的键值

### Redis中的LRU 算法和 LFU 算法

LRU 算法

Least Recently Used 翻译为**最近最少使用**，会选择淘汰最近最少使用的数据。

链表中的元素按照操作顺序从前往后排列，最新操作的键会被移动到表头，当需要内存淘汰时，只需要删除链表尾部的元素即可

缺点：

* 需要用链表管理所有的缓存数据，这会带来额外的空间开销；
* 当有数据被访问时，需要在链表上把该数据移动到头端，如果有大量数据被访问，就会带来很多链表移动操作，会很耗时，进而会降低 Redis 缓存性能。

Redis 中的 LRU 算法

**Redis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**。内存淘汰时，会使用**随机采样的方式来淘汰数据**，它是随机取 5 个值（此值可配置），然后**淘汰最久没有使用的那个**。

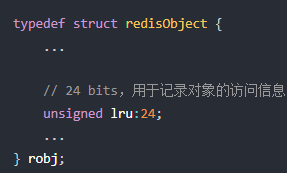
缺点：

**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。

LFU 算法

Least Frequently Used 翻译为**最近最不常用的**，会选择淘汰访问次数最少的数据

Redis 中的 LFU 算法



Redis中的LRU和LFU都是使用结构体中的lru字段记录访问信息：

**LRU 算法中**，lru 字段是用来记录 key 的访问时间戳

**LFU 算法中**，lru 字段高 16bit 记录 key 的访问时间戳；低 8bit记录 key 的访问频次

## Redis 缓存设计

### 如何避免缓存雪崩、缓存击穿、缓存穿透？

缓存雪崩

当设置了过期时间的缓存数据过期后，用户访问的数据如果不在缓存里，业务系统需要重新生成缓存，因此就会访问数据库，并将数据更新到 Redis 里

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**

解决方案：

* **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。

缓存击穿

缓存中的**某个热点数据（频繁被访问的数据）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。**缓存击穿实际上是缓存雪崩的一种

解决方案：

* 互斥锁方案（Redis 中使用 setNX 方法设置一个状态位，表示这是一种锁定状态），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；

缓存穿透

用户访问的数据，**既不在缓存中，也不在数据库中。**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**

发生场景：

* 业务误操作，缓存中的数据和数据库中的数据都被误删除了，所以导致缓存和数据库中都没有数据；
* 黑客恶意攻击，故意大量访问某些读取不存在数据的业务；

应对方案：

* **非法请求的限制**：在 API 入口处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**：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，而不用通过查询数据库来判断数据是否存在。

### 如何设计一个缓存策略，可以动态缓存热点数据呢？

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**。

具体细节如下：

* 先通过缓存系统做一个排序队列（比如存放 1000 个商品），系统会根据商品的访问时间，更新队列信息，越是最近访问的商品排名越靠前；
* 同时系统会定期过滤掉队列中排名最后的 200 个商品，然后再从数据库中随机读取出 200 个商品加入队列中；
* 这样当请求每次到达的时候，会先从队列中获取商品 ID，如果命中，就根据 ID 再从另一个缓存数据结构中读取实际的商品信息，并返回。

在 Redis 中可以用 zadd 方法和 zrange 方法来完成排序队列和获取 200 个商品的操作。

### 常见的缓存更新策略

Cache Aside（旁路缓存）策略

**Redis 和 MySQL 采用的更新策略**

应用程序直接与「数据库、缓存」交互，并负责对缓存的维护

**写策略：**

* 先更新数据库中的数据，再删除缓存中的数据。(**注意是删除不是更新！！**)

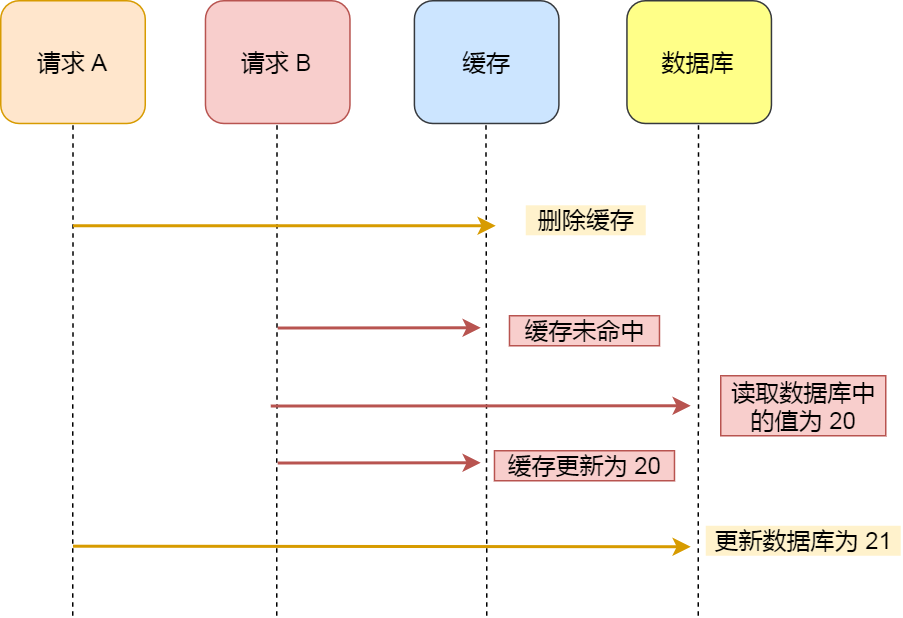
**读策略：**

* 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。

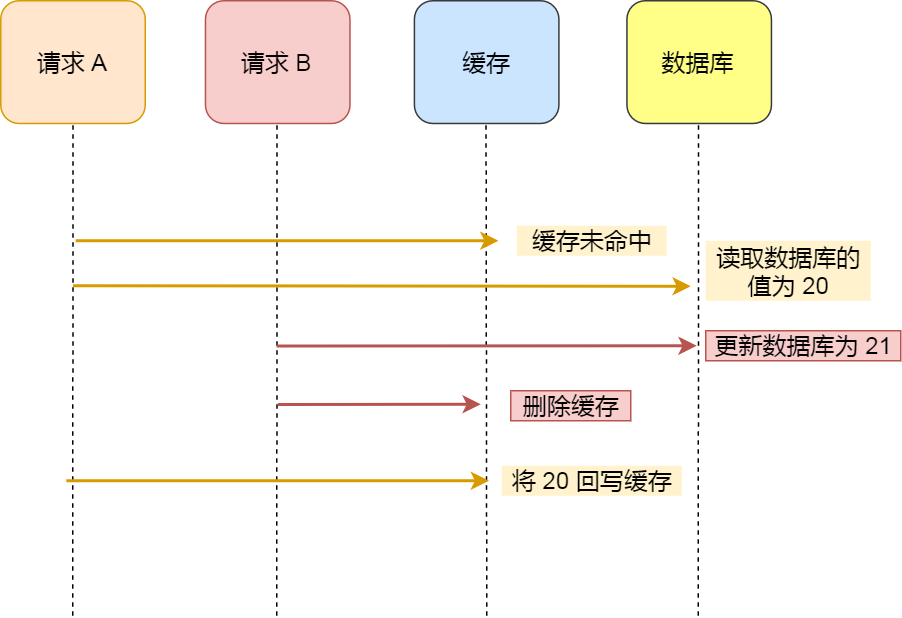
为什么写策略先更新再删除：

读写并发时，容易导致缓存和数据库的数据不一致的问题：

① 先删缓存再更新数据库



② 先更新数据库再删缓存



**但是在实际中，缓存的写入通常要远远快于数据库的写入，因此请求 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基本不会出现**

**Cache Aside 策略适合读多写少的场景，**因为当写入比较频繁时，缓存中的数据会被频繁地清理，这样会对缓存的命中率有一些影响

Read/Write Through（读穿 / 写穿）策略

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代理

**读穿（读直达）：**

* 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。

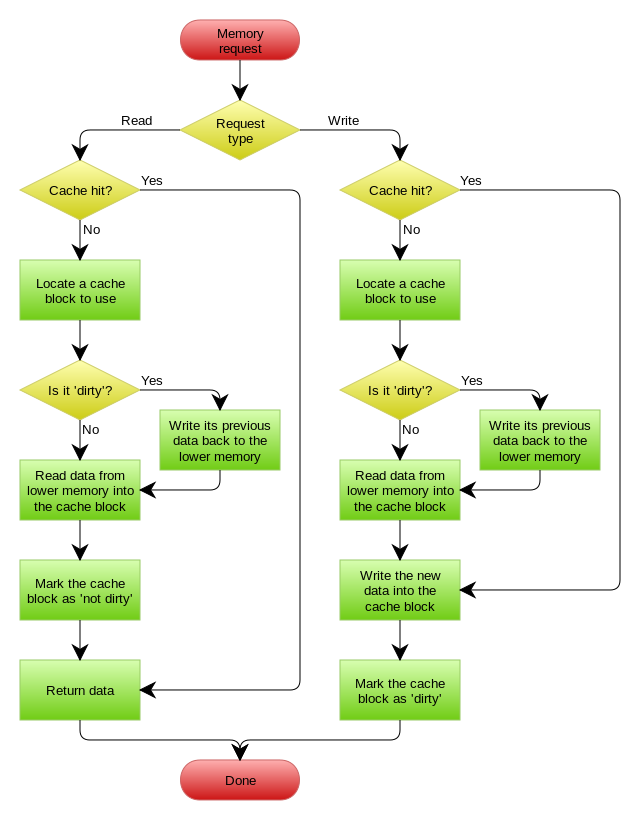
**写穿（写直达）：**

* 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

不用的原因：Redis 不提供写入数据库和自动加载数据库中的数据的功能

Write Back（写回）策略

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，但数据不是强一致性的，而且会有数据丢失的风险（因为**缓存一般使用内存**）**

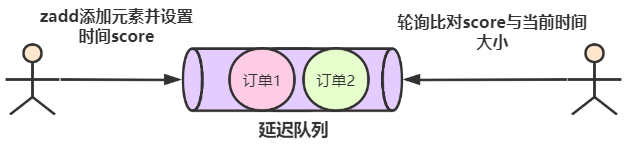
## Redis 实战

### 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。常见使用场景：

* 在淘宝、京东等购物平台上下单，超过一定时间未付款，订单会自动取消；
* 打车的时候，在规定时间没有车主接单，平台会取消你的单并提醒你暂时没有车主接单；
* 点外卖的时候，如果商家在10分钟还没接单，就会自动取消订单；

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。



### Redis 的大 key 如何处理？

大key指的是**key 对应的 value 很大**。一般而言，下面这两种情况被称为大 key：

* 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；

大 key 造成的问题

* **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。

如何找到大 key

***1、redis-cli –bigkeys命令***



注意事项：

* 最好选择在从节点上执行该命令。因为主节点上执行时，会阻塞主节点；
* 如果没有从节点，那么可以选择在 Redis 实例业务压力的低峰阶段进行扫描查询，以免影响到实例的正常运行；或者可以使用 -i 参数控制扫描间隔，避免长时间扫描降低 Redis 实例的性能。

缺点：

* 只能返回每种类型中最大的那个 bigkey，无法得到大小排在前 N 位的 bigkey；
* 对于集合类型来说，这个方法只统计集合元素个数的多少，而不是实际占用的内存量。但是，一个集合中的元素个数多，并不一定占用的内存就多。因为，有可能每个元素占用的内存很小，这样的话，即使元素个数有很多，总内存开销也不大；

***2、SCAN 命令***

使用 SCAN 命令对数据库扫描，然后用 TYPE 命令获取返回的每一个 key 的类型

String 类型：直接使用 STRLEN 命令获取字符串的长度，也就是占用的内存空间字节数。

集合类型：

* 如果能够预先从业务层知道集合元素的平均大小，那么，可以使用下面的命令获取集合元素的个数，然后乘以集合元素的平均大小，这样就能获得集合占用的内存大小了。List 类型：LLEN 命令；Hash 类型：HLEN 命令；Set 类型：SCARD 命令；Sorted Set 类型：ZCARD 命令；
* 如果不能提前知道写入集合的元素大小，可以使用 MEMORY USAGE 命令（需要 Redis 4.0 及以上版本），查询一个键值对占用的内存空间。

***3、RdbTools 工具***

RdbTools 第三方开源工具，可以用来解析 Redis 快照（RDB）文件，找到其中的大 key。

比如，下面这条命令，将大于 10 kb 的  key  输出到一个表格文件：



如何删除大 key

如果一下子释放了大量内存，空闲内存块链表操作时间就会增加，相应地就会造成 Redis 主线程的阻塞，如果主线程发生了阻塞，其他所有请求可能都会超时，超时越来越多，会造成 Redis 连接耗尽，产生各种异常。

***1、分批次删除***

**大 Hash：**使用 hscan 命令，每次获取 100 个字段，再用 hdel 命令，每次删除 1 个字段

**大 List：**通过 ltrim 命令，每次删除少量元素

**大 Set：**使用 sscan 命令，每次扫描集合中 100 个元素，再用 srem 命令每次删除一个键

**大 ZSet：**使用 zremrangebyrank 命令，每次删除 top 100个元素

***2、异步删除***

Redis 4.0 版本开始，可以采用**异步删除**法，**用 unlink 命令代替 del 来删除**

### Redis 管道有什么用？

**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。

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

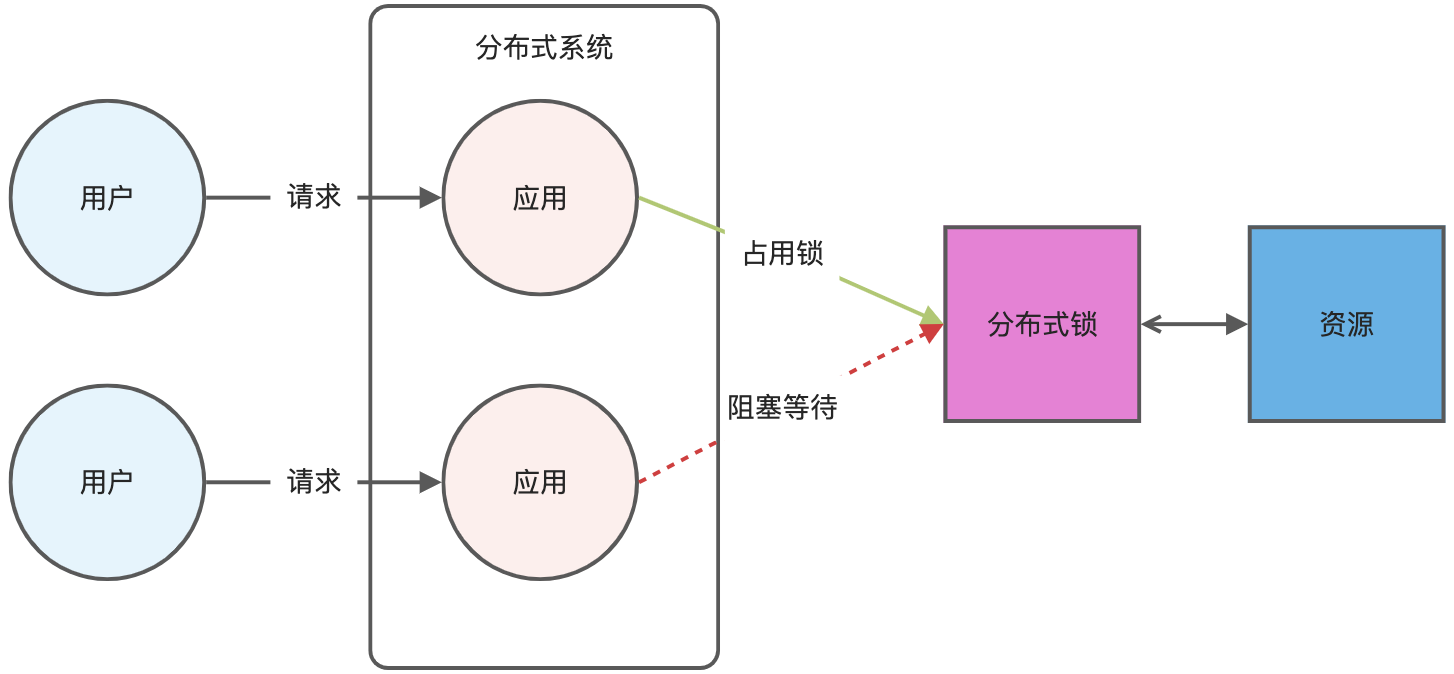
### Redis 事务支持回滚吗？

**Redis 中并没有提供回滚机制**，虽然 Redis 提供了 DISCARD 命令，但是这个命令只能用来主动放弃事务执行，在事务提交前把暂存的命令队列清空，起不到回滚的效果。

而事务提交后，实际执行时报错了，正确的命令依然可以正常执行，所以这可以看出 **Redis 并不一定保证原子性**

### 如何用 Redis 实现分布式锁

分布式锁是用于分布式环境下并发控制的一种机制，用于控制某个资源在同一时刻只能被一个应用所使用。



实现加分布式锁的分布式命令：



* lock\_key ： key 键；
* unique\_value ：客户端生成的唯一的标识，区分来自不同客户端的锁操作；
* NX ：代表只在 lock\_key 不存在时，才对 lock\_key 进行设置操作；
* PX 10000 ：设置 lock\_key 的过期时间为 10000ms，避免客户端发生异常无法释放锁。

解锁时，要先判断锁的 unique\_value 是否为加锁客户端，是的话才将 lock\_key 键删除。

基于 Redis 实现分布式锁的优缺点

优点：

1. 性能高效（这是选择缓存实现分布式锁最核心的出发点）。
2. 实现方便。Redis 提供了 setnx 方法，实现分布式锁很方便。
3. 避免单点故障（Redis 是跨集群部署的，避免了单点故障）。

缺点：

1. **超时时间难以控制**：一个线程获取到了锁之后，业务代码执行时间超过了锁的超时时间，锁自动失效，被另一个线程意外地持有了锁，导致共享资源失去保护

解决方法：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

1. **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：**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。

Redis 如何解决集群情况下分布式锁的可靠性？

Redis 官方设计了一个分布式锁算法 Redlock（红锁）

Redlock 算法的基本思路，**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，那么我们就认为，客户端成功地获得分布式锁，否则加锁失败**。

加锁成功的具体条件：

* 条件一：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；
* 条件二：客户端从大多数节点获取锁的总耗时（t2-t1）小于锁设置的过期时间。

若加锁失败，客户端向**所有 Redis 节点发起释放锁的操作**