影响Redis性能的5大方面的潜在因素

一、Redis内部的阻塞式操作

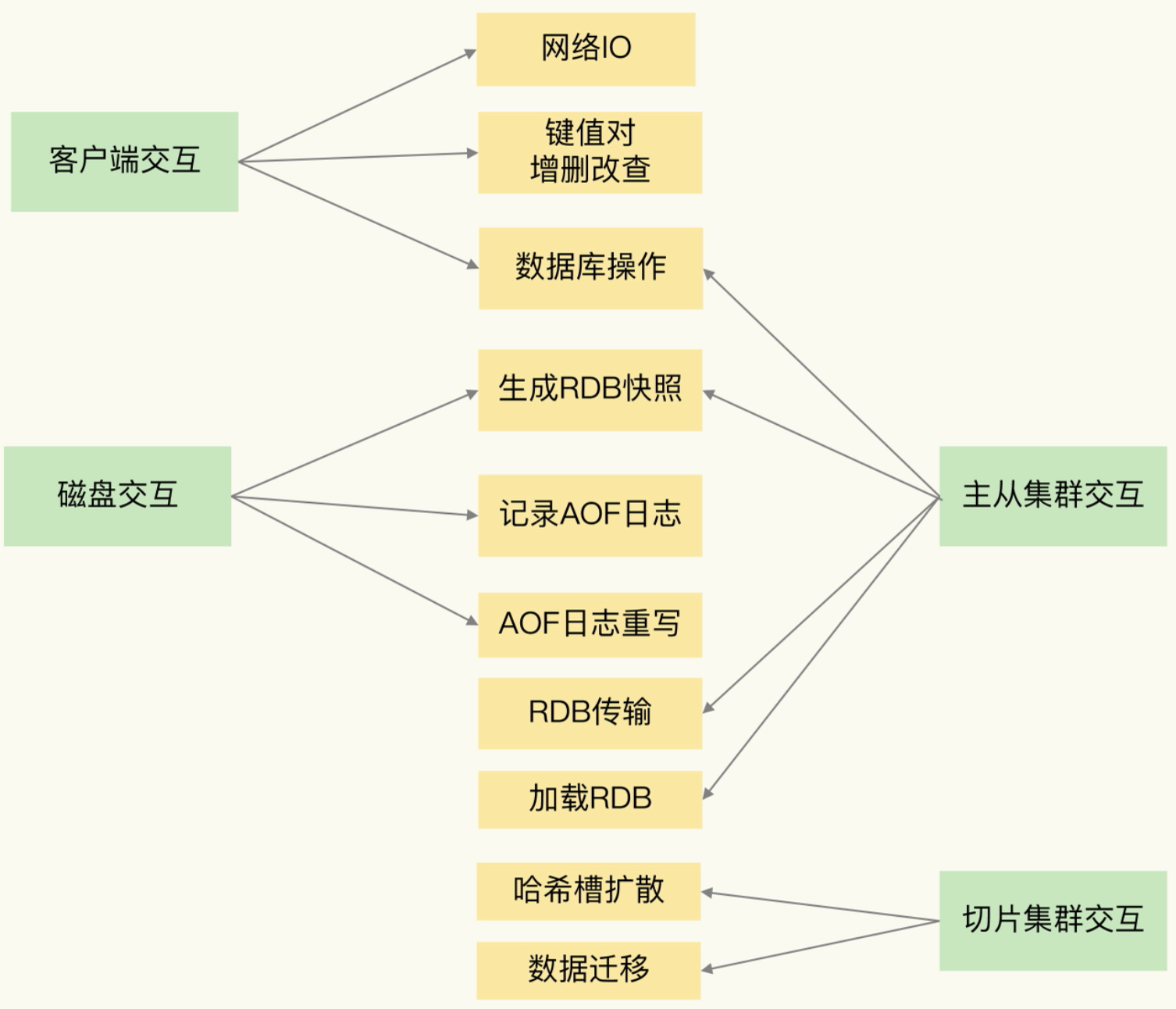
1.1 阻塞点

- [ ] 客户端：网络IO，键值对增删改查操作，数据库操作；

- [ ] 磁盘：生成RDB快照，记录AOF日志，AOF日志重写；

- [ ] 主从节点：主库生成、传输RDB文件，从库接收RDB文件、清空数据库、加载RDB文件；

- [ ] 切片集群实例：向其他实例传输哈希槽信息，数据迁移。



1.1.1 和客户端交互时的阻塞点

网络IO有时候会比较慢，但是Redis使用了IO多路复用机制，避免了主线程一直处在等待网络连接或请求到来的状态，所以，网络IO不是导致Redis阻塞的因素。

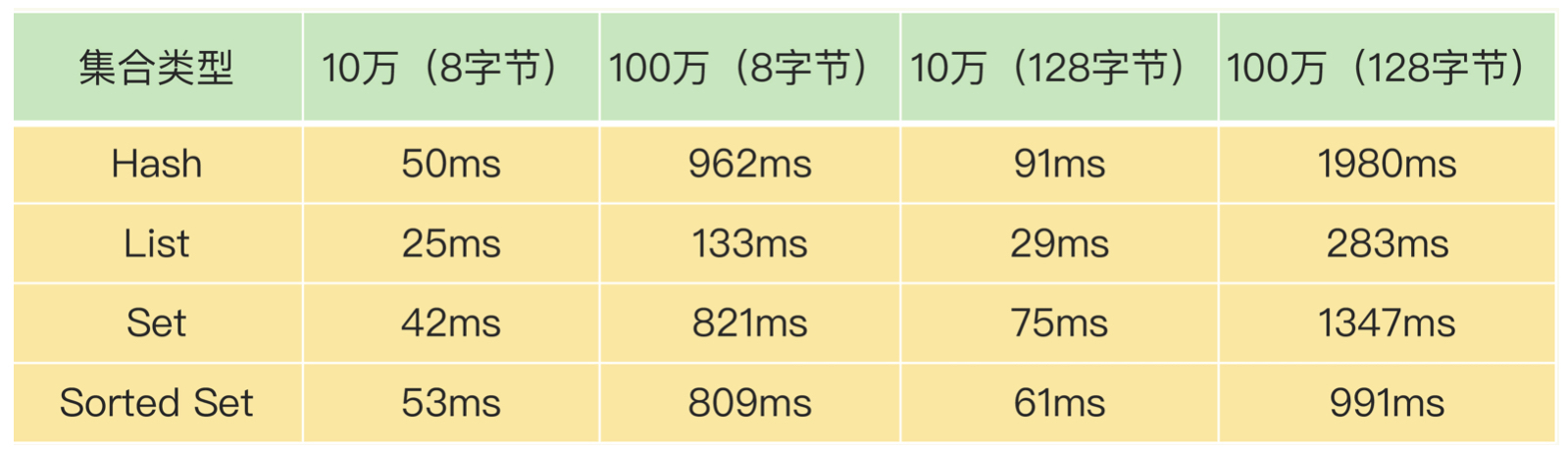
- [ ] 集合全量查询和聚合操作

键值对的增删改查操作是Redis和客户端交互的主要部分，也是Redis主线程执行的主要任务。所以，复杂度高的增删改查操作肯定会阻塞Redis。最基本的标准，就是看操作的复杂度是否为O(N)。

Redis中涉及集合的操作复杂度通常为O(N)，要在使用时重视起来。例如集合元素全量查询操作HGETALL、SMEMBERS，以及集合的聚合统计操作，例如求交、并和差集。

- [ ] bigkey删除操作

删除操作的本质是要释放键值对占用的内存空间。为了更加高效地管理内存空间，在应用程序释放内存时，操作系统需要把释放掉的内存块插入一个空闲内存块的链表，以便后续进行管理和再分配。这个过程本身需要一定时间，而且会阻塞当前释放内存的应用程序。



①当元素数量从10万增加到100万时，4大集合类型的删除时间的增长幅度从5倍上升到了近20倍；

②集合元素越大，删除所花费的时间就越长；

③当删除有100万个元素的集合时，最大的删除时间绝对值已经达到了1.98s（Hash类型）。Redis的响应时间一般在微秒级别，所以，一个操作达到了近2s，不可避免地会阻塞主线程。

- [ ] 清空数据库

1.1.2 和磁盘交互时的阻塞点

- [ ] AOF日志同步写

Redis直接记录AOF日志时，会根据不同的写回策略对数据做落盘保存。一个同步写磁盘的操作的耗时大约是1～2ms，如果有大量的写操作需要记录在AOF日志中，并同步写回的话，就会阻塞主线程了。

1.1.3 主从节点交互时的阻塞点

- [ ] 对于从库来说，它在接收了RDB文件后，需要使用FLUSHDB命令清空当前数据库

- [ ] RDB文件加载到内存，这个过程的快慢和RDB文件的大小密切相关，RDB文件越大，加载过程越慢

1.1.4 切片集群实例交互时的阻塞点

- [ ] 使用了Redis Cluster方案（使用了同步迁移），而且同时迁移bigkey

1.2 阻塞点可以异步执行

如果一个操作能被异步执行，就意味着，它并不是Redis主线程的关键路径上的操作。需要把结果返回给客户端，它就是关键路径上的操作，所以主线程必须立即把这个操作执行完。

对于Redis来说，读操作是典型的关键路径操作，因为客户端发送了读操作之后，就会等待读取的数据返回，以便进行后续的数据处理。

- [ ] 集合全量查询和聚合操作

删除操作并不需要给客户端返回具体的数据结果，所以不算是关键路径操作。因此，可以使用后台子线程来异步执行删除操作。

- [x] bigkey删除

- [x] 清空数据库

为了保证数据可靠性，Redis实例需要保证AOF日志中的操作记录已经落盘，这个操作虽然需要实例等待，但它并不会返回具体的数据结果给实例。

- [x] AOF日志同步写

从库要想对客户端提供数据存取服务，就必须把RDB文件加载完成。

- [ ] 从库加载RDB文件

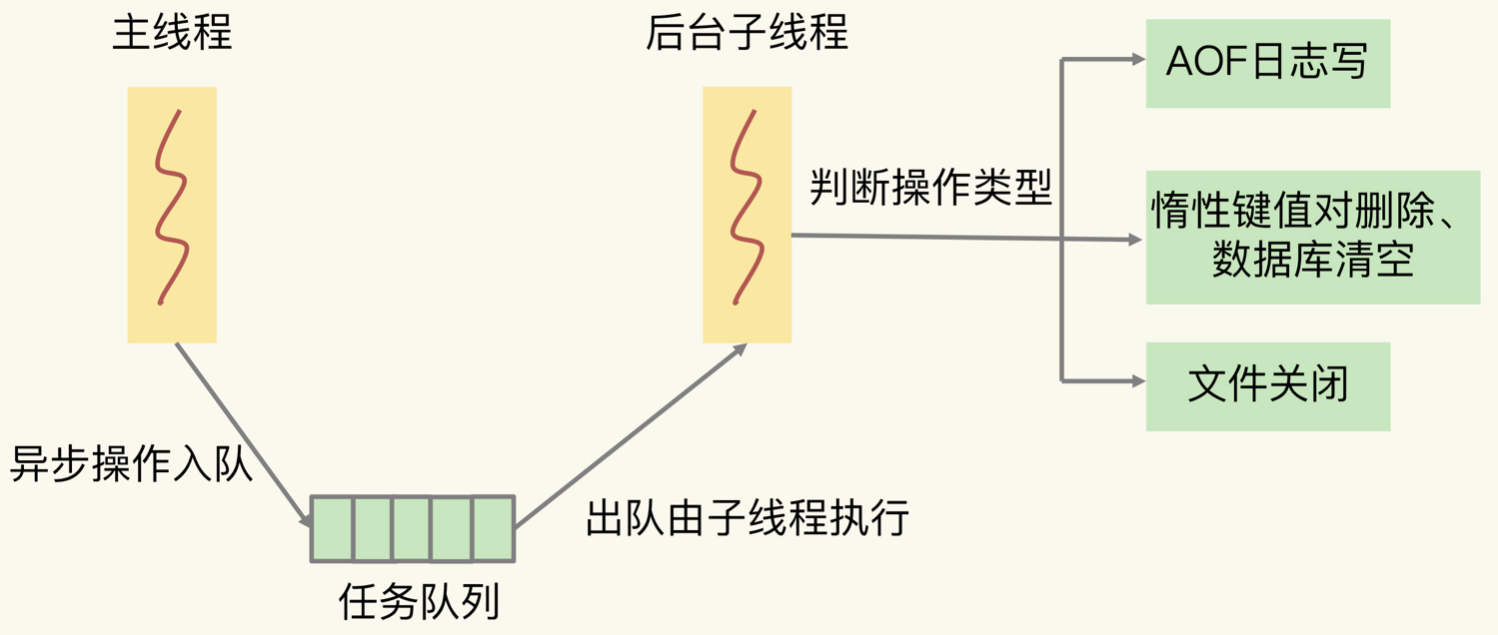
1.3 异步的子线程机制

Redis主线程启动后，会使用操作系统提供的pthread\_create函数创建3个子线程，分别由它们负责AOF日志写操作、键值对删除以及文件关闭的异步执行。

主线程通过一个链表形式的任务队列和子线程进行交互。当收到键值对删除和清空数据库的操作时，主线程会把这个操作封装成一个任务，放入到任务队列中，然后给客户端返回一个完成信息，表明删除已经完成。

但实际上，这个时候删除还没有执行，等到后台子线程从任务队列中读取任务后，才开始实际删除键值对，并释放相应的内存空间。因此，我们把这种异步删除也称为惰性删除（lazy free）。

和惰性删除类似，当AOF日志配置成everysec选项后，主线程会把AOF写日志操作封装成一个任务，也放到任务队列中。后台子线程读取任务后，开始自行写入AOF日志，这样主线程就不用一直等待AOF日志写完了。



注：异步的键值对删除和数据库清空操作是Redis 4.0后提供的功能，Redis也提供了新的命令来执行这两个操作。

- [ ] 键值对删除：当你的集合类型中有大量元素（例如有百万级别或千万级别元素）需要删除时，我建议你使用UNLINK命令。

- [ ] 清空数据库：可以在FLUSHDB和FLUSHALL命令后加上ASYNC选项，这样就可以让后台子线程异步地清空数据库。

如果你使用的是4.0之前的版本，当你遇到bigkey删除时，我给你个小建议：先使用集合类型提供的SCAN命令读取数据，然后再进行删除。因为用SCAN命令可以每次只读取一部分数据并进行删除，这样可以避免一次性删除大量key给主线程带来的阻塞。

另，集合全量查询和聚合操作、从库加载RDB文件是在关键路径上，无法使用异步操作来完成。

- [ ] 集合全量查询和聚合操作：可以使用SCAN命令，分批读取数据，再在客户端进行聚合计算；

- [ ] 从库加载RDB文件：把主库的数据量大小控制在2~4GB左右，以保证RDB文件能以较快的速度加载。

二、CPU核和NUMA架构的影响

- [ ] 在多核CPU架构下，Redis如果在不同的核上运行，就需要频繁地进行上下文切换，这个过程会增加Redis的执行时间，客户端也会观察到较高的尾延迟了。所以在Redis运行时，把实例和某个核绑定，重复利用核上的L1、L2缓存，可以降低响应延迟。

- [ ] 如果服务器使用的是NUMA架构（非统一内存访问架构，Non-Uniform Memory Access），Redis实例一旦被调度到和中断处理程序不在同一个CPU Socket，就要跨CPU Socket访问网络数据，这就会降低Redis的性能。所以，把Redis实例和网络中断处理程序绑在同一个CPU Socket下的不同核上，这样可以提升Redis的运行性能。虽然绑核可以帮助Redis降低请求执行时间，但是，除了主线程，Redis还有用于RDB和AOF重写的子进程，以及4.0版本之后提供的用于惰性删除的后台线程。当Redis实例和一个逻辑核绑定后，这些子进程和后台线程会和主线程竞争CPU资源，也会对Redis性能造成影响。所以，两个建议：

• 如果不修改Redis代码，可以把按一个Redis实例一个物理核方式进行绑定，这样，Redis的主线程、子进程和后台线程可以共享使用一个物理核上的两个逻辑核。

• 如果很熟悉Redis的源码，就可以在源码中增加绑核操作，把子进程和后台线程绑到不同的核上，这样可以避免对主线程的CPU资源竞争。

• 如果你不熟悉Redis源码，Redis 6.0出来后，可以支持CPU核绑定的配置操作。

三、Redis关键系统配置

3.1 Redis真的变慢了吗

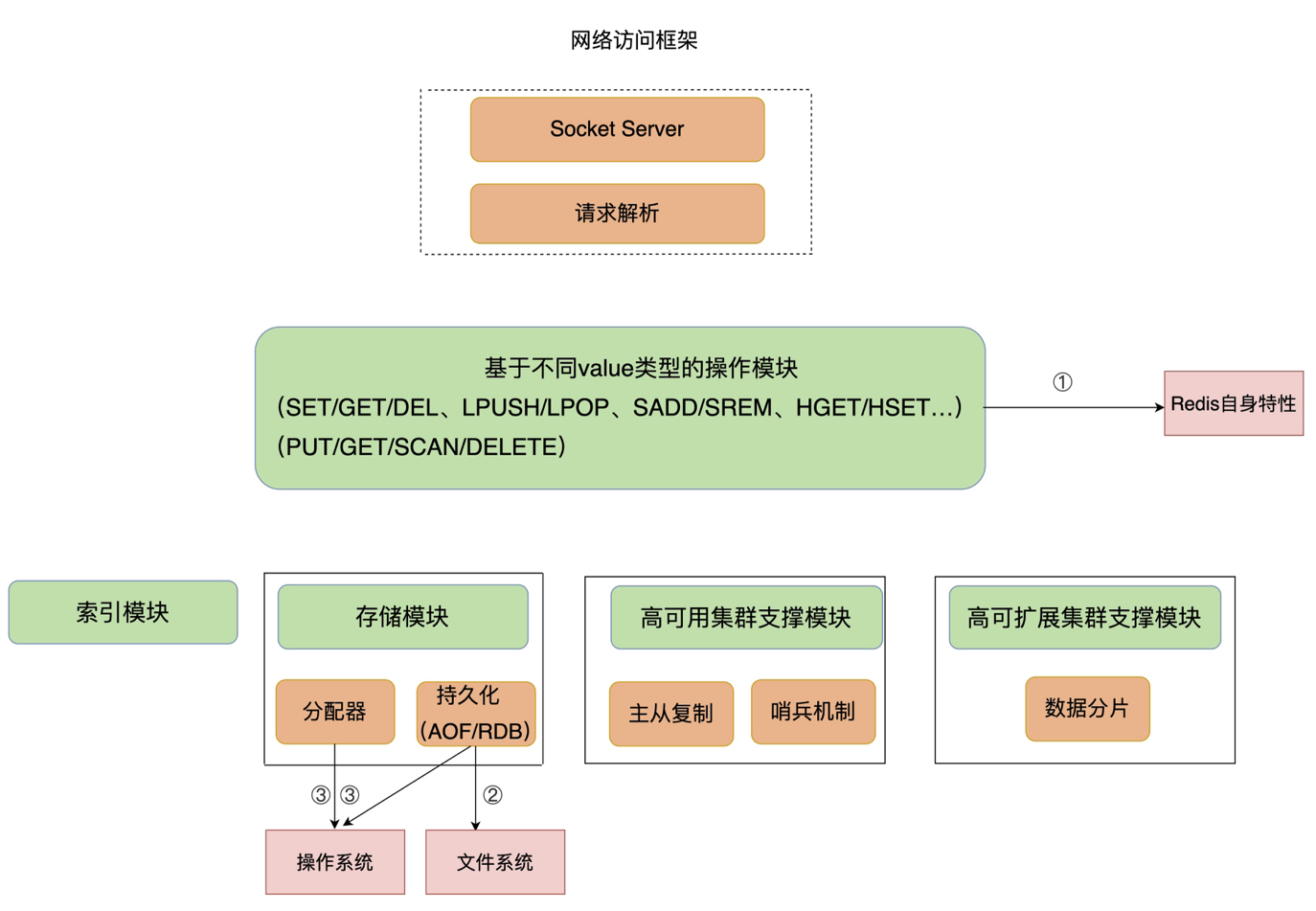
查看Redis的响应延迟，当前环境下的Redis基线性能。

从2.8.7版本开始，redis-cli命令提供了–intrinsic-latency选项，可以用来监测和统计测试期间内的最大延迟，这个延迟可以作为Redis的基线性能。其中，测试时长可以用–intrinsic-latency选项的参数来指定。

一般来说，要把运行时延迟和基线性能进行对比，如果观察到的Redis运行时延迟是其基线性能的2倍及以上，就可以认定Redis变慢了。

3.2 如何应对Redis变慢

Redis自身的操作特性、文件系统和操作系统，它们是影响Redis性能的三大要素。



3.3 Redis自身操作特性的影响

3.3.1 慢查询命令

是指在Redis中执行速度慢的命令。Redis提供的命令操作很多，并不是所有命令都慢，这和命令操作的复杂度有关。

处理方式：

- [ ] 用其他高效命令代替。比如说，如果你需要返回一个SET中的所有成员时，不要使用SMEMBERS命令，而是要使用SSCAN多次迭代返回，避免一次返回大量数据，造成线程阻塞。

- [ ] 当需要执行排序、交集、并集操作时，可以在客户端完成，而不要用SORT、SUNION、SINTER这些命令，以免拖慢Redis实例。

- [ ] 如果业务逻辑就是要求使用慢查询命令，考虑采用性能更好的CPU，更快地完成查询命令，避免慢查询的影响。

- [ ] KEYS用于返回和输入模式匹配的所有key，因为KEYS命令需要遍历存储的键值对，所以操作延时高。如果你不了解它的实现而使用了它，就会导致Redis性能变慢。所以，KEYS命令一般不被建议用于生产环境中。

3.3.2 过期key操作

过期key的自动删除机制是Redis用来回收内存空间的常用机制，应用广泛，本身就会引起Redis操作阻塞，导致性能变慢。

Redis键值对的key可以设置过期时间。默认情况下，Redis每100毫秒会删除一些过期key，具体的算法如下：

1. 采样ACTIVE\_EXPIRE\_CYCLE\_LOOKUPS\_PER\_LOOP个数的key，并将其中过期的key全部删除；

2. 如果超过25%的key过期了，则重复删除的过程，直到过期key的比例降至25%以下。

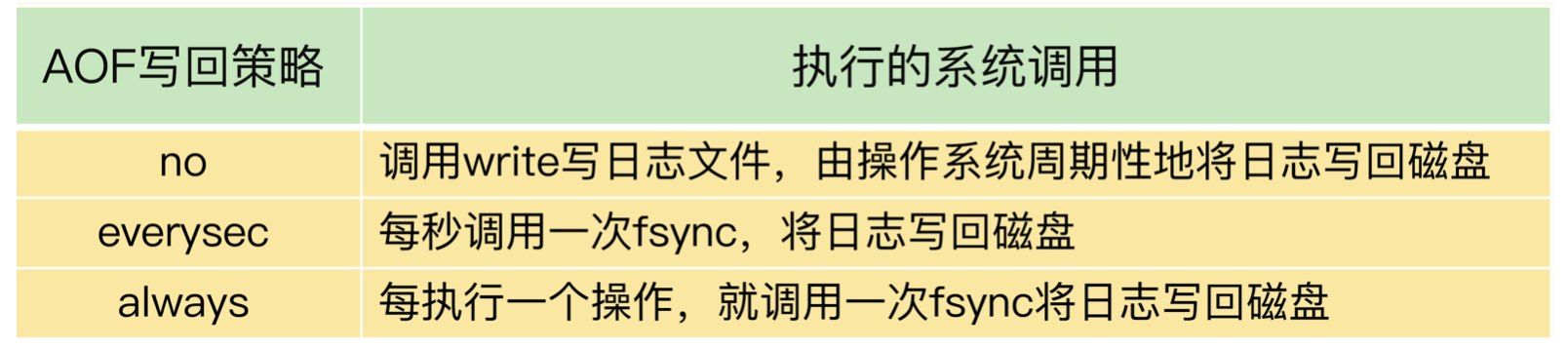
ACTIVE\_EXPIRE\_CYCLE\_LOOKUPS\_PER\_LOOP是Redis的一个参数，默认是20。如果触发了上面这个算法的第二条，Redis就会一直删除以释放内存空间。注意，删除操作是阻塞的（Redis 4.0后可以用异步线程机制来减少阻塞影响）。

频繁使用带有相同时间参数的EXPIREAT命令设置过期key，导致在同一秒内有大量的key同时过期。

3.4 文件系统：AOF模式

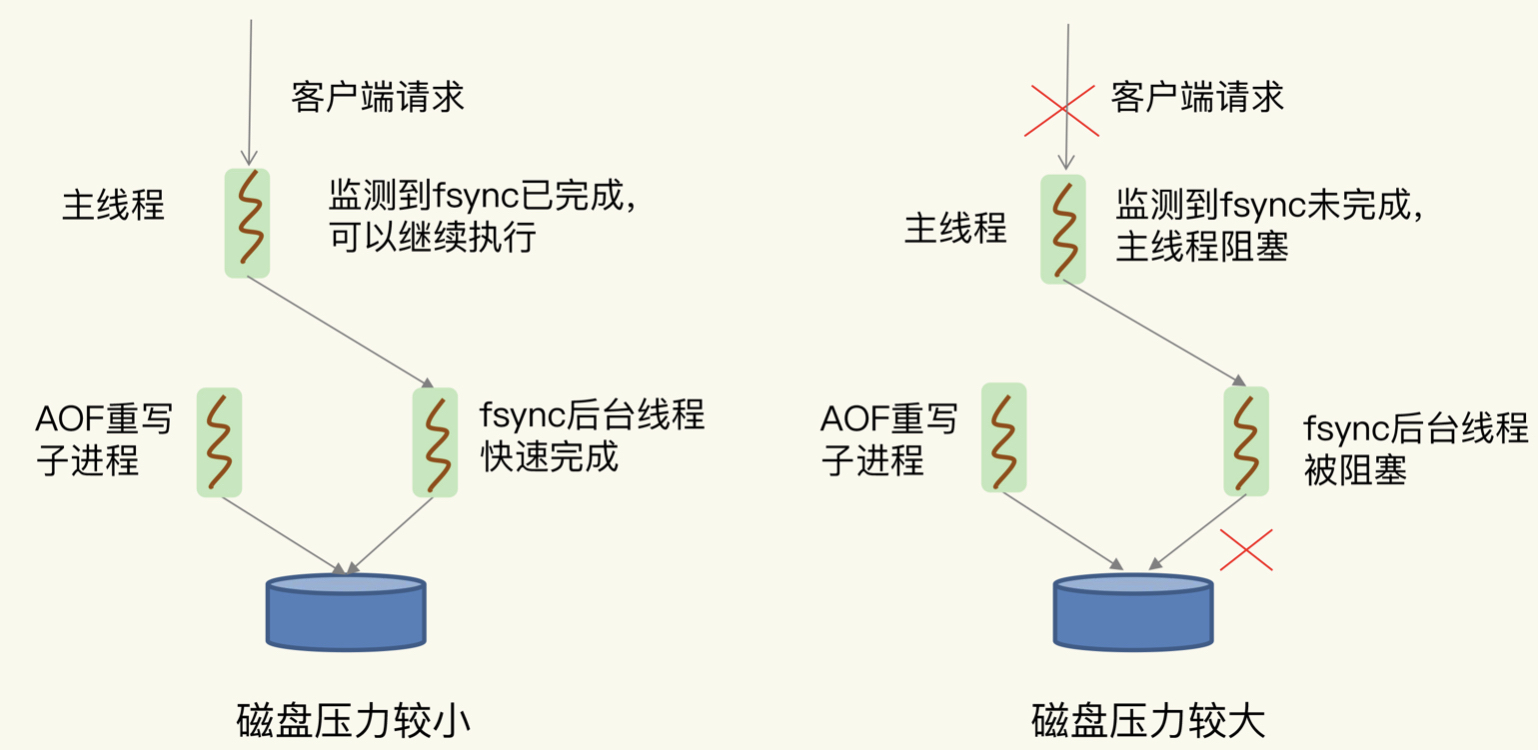
为了保证数据可靠性，Redis会采用AOF日志或RDB快照。其中，AOF日志提供了三种日志写回策略：no、everysec、always。这三种写回策略依赖文件系统的两个系统调用完成，也就是write和fsync。

write只要把日志记录写到内核缓冲区，就可以返回了，并不需要等待日志实际写回到磁盘；而fsync需要把日志记录写回到磁盘后才能返回，时间较长。



AOF重写会对磁盘进行大量IO操作，同时，fsync又需要等到数据写到磁盘后才能返回，所以，当AOF重写的压力比较大时，就会导致fsync被阻塞。虽然fsync是由后台子线程负责执行的，但是，主线程会监控fsync的执行进度。

当主线程使用后台子线程执行了一次fsync，需要再次把新接收的操作记录写回磁盘时，如果主线程发现上一次的fsync还没有执行完，那么它就会阻塞。所以，如果后台子线程执行的fsync频繁阻塞的话（比如AOF重写占用了大量的磁盘IO带宽），主线程也会阻塞，导致Redis性能变慢。



- [ ] 如果业务应用对延迟非常敏感，但同时允许一定量的数据丢失，那么，可以把配置项no-appendfsync-on-rewrite设置为yes，表示在AOF重写时，不进行fsync操作。也就是说，Redis实例把写命令写到内存后，不调用后台线程进行fsync操作，就可以直接返回了。当然，如果此时实例发生宕机，就会导致数据丢失。反之，如果这个配置项设置为no（也是默认配置），在AOF重写时，Redis实例仍然会调用后台线程进行fsync操作，这就会给实例带来阻塞。

- [ ] 如果的确需要高性能，同时也需要高可靠数据保证，考虑采用高速的固态硬盘作为AOF日志的写入设备。

3.5 操作系统的内存swap

内存swap是操作系统里将内存数据在内存和磁盘间来回换入和换出的机制，涉及到磁盘的读写，所以，一旦触发swap，无论是被换入数据的进程，还是被换出数据的进程，其性能都会受到慢速磁盘读写的影响。

触发swap的原因主要是物理机器内存不足，对于Redis而言，有两种常见的情况：

- [ ] Redis实例自身使用了大量的内存，导致物理机器的可用内存不足；

- [ ] 和Redis实例在同一台机器上运行的其他进程，在进行大量的文件读写操作。文件读写本身会占用系统内存，这会导致分配给Redis实例的内存量变少，进而触发Redis发生swap。

当出现百MB，甚至GB级别的swap大小时，就表明，此时，Redis实例的内存压力很大，很有可能会变慢。所以，swap的大小是排查Redis性能变慢是否由swap引起的重要指标。

解决方案：

- [ ] 增加机器内存。

- [ ] 如果该实例在一个Redis切片集群中，可以增加Redis集群的实例个数，来分摊每个实例服务的数据量，进而减少每个实例所需的内存量。

- [ ] 如果Redis实例和其他操作大量文件的程序（例如数据分析程序）共享机器，可以将Redis实例迁移到单独的机器上运行，以满足它的内存需求量。

- [ ] 如果该实例正好是Redis主从集群中的主库，而从库的内存很大，也可以考虑进行主从切换，把大内存的从库变成主库，由它来处理客户端请求。

3.6 操作系统：内存大页机制（Transparent Huge Page, THP）

Linux内核从2.6.38开始支持内存大页机制，该机制支持2MB大小的内存页分配，而常规的内存页分配是按4KB的粒度来执行的。

Redis为了提供数据可靠性保证，需要将数据做持久化保存。这个写入过程由额外的线程执行，所以，此时Redis主线程仍然可以接收客户端写请求。客户端的写请求可能会修改正在进行持久化的数据。在这一过程中，Redis就会采用写时复制机制，一旦有数据要被修改，Redis并不会直接修改内存中的数据，而是将这些数据拷贝一份，然后再进行修改。

如果采用了内存大页，那么，即使客户端请求只修改100B的数据，Redis也需要拷贝2MB的大页。相反，如果是常规内存页机制，只用拷贝4KB。两者相比，当客户端请求修改或新写入数据较多时，内存大页机制将导致大量的拷贝，这就会影响Redis正常的访存操作，最终导致性能变慢。

解决方案：关闭内存大页

- [ ] 首先，排查下内存大页。方法是：在Redis实例运行的机器上执行如下命令:cat /sys/kernel/mm/transparent\_hugepage/enabled

如果执行结果是always，就表明内存大页机制被启动了；如果是never，就表示，内存大页机制被禁止。

- [ ] 在实际生产环境中部署时，不要使用内存大页机制，操作也很简单，只需要执行下面的命令就可以了：echo never /sys/kernel/mm/transparent\_hugepage/enabled

3.7 小结

- [ ] 获取Redis实例在当前环境下的基线性能。

- [ ] 是否用了慢查询命令？如果是的话，就使用其他命令替代慢查询命令，或者把聚合计算命令放在客户端做。

- [ ] 是否对过期key设置了相同的过期时间？对于批量删除的key，可以在每个key的过期时间上加一个随机数，避免同时删除。

- [ ] 是否存在bigkey？ 对于bigkey的删除操作，如果你的Redis是4.0及以上的版本，可以直接利用异步线程机制减少主线程阻塞；如果是Redis 4.0以前的版本，可以使用SCAN命令迭代删除；对于bigkey的集合查询和聚合操作，可以使用SCAN命令在客户端完成。

- [ ] Redis AOF配置级别是什么？业务层面是否的确需要这一可靠性级别？如果我们需要高性能，同时也允许数据丢失，可以将配置项no-appendfsync-on-rewrite设置为yes，避免AOF重写和fsync竞争磁盘IO资源，导致Redis延迟增加。当然， 如果既需要高性能又需要高可靠性，最好使用高速固态盘作为AOF日志的写入盘。

- [ ] Redis实例的内存使用是否过大？发生swap了吗？如果是的话，就增加机器内存，或者是使用Redis集群，分摊单机Redis的键值对数量和内存压力。同时，要避免出现Redis和其他内存需求大的应用共享机器的情况。

- [ ] 在Redis实例的运行环境中，是否启用了透明大页机制？如果是的话，直接关闭内存大页机制就行了。

- [ ] 是否运行了Redis主从集群？如果是的话，把主库实例的数据量大小控制在2~4GB，以免主从复制时，从库因加载大的RDB文件而阻塞。

- [ ] 是否使用了多核CPU或NUMA架构的机器运行Redis实例？使用多核CPU时，可以给Redis实例绑定物理核；使用NUMA架构时，注意把Redis实例和网络中断处理程序运行在同一个CPU Socket上。

四、Redis内存碎片

当数据删除后，Redis释放的内存空间会由内存分配器管理，并不会立即返回给操作系统。所以，操作系统仍然会记录着给Redis分配了大量内存。

Redis释放的内存空间可能并不是连续的，这些不连续的内存空间很有可能处于一种闲置的状态。这就会导致虽然有空闲空间，Redis却无法用来保存数据，不仅会减少Redis能够实际保存的数据量，还会降低Redis运行机器的成本回报率。

4.1 什么是内存碎片

虽然操作系统的剩余内存空间总量足够，但是，应用申请的是一块连续地址空间的N字节，但在剩余的内存空间中，没有大小为N字节的连续空间了，那么，这些剩余空间就是内存碎片。

4.2 内存碎片是如何形成的

4.2.1 内因：内存分配器的分配策略

内存分配器的分配策略就决定了操作系统无法做到“按需分配”。这是因为，内存分配器一般是按固定大小来分配内存，而不是完全按照应用程序申请的内存空间大小给程序分配。

Redis可以使用libc、jemalloc、tcmalloc多种内存分配器来分配内存，默认使用jemalloc。jemalloc的分配策略之一，是按照一系列固定的大小划分内存空间，例如8字节、16字节、32字节、48字节，…, 2KB、4KB、8KB等。当程序申请的内存最接近某个固定值时，jemalloc会给它分配相应大小的空间。这样的分配方式本身是为了减少分配次数。

但是，如果Redis每次向分配器申请的内存空间大小不一样，这种分配方式就会有形成碎片的风险，而这正好来源于Redis的外因了。

4.2.2 外因：键值对大小不一样和删改操作

- [ ] Redis通常作为共用的缓存系统或键值数据库对外提供服务，不同业务应用的数据都可能保存在Redis中，这就会带来不同大小的键值对，本身就会有大小不一的空间需求。内存分配器只能按固定大小分配内存，所以，分配的内存空间一般都会比申请的空间大一些，不会完全一致，这本身就会造成一定的碎片，降低内存空间存储效率。

- [ ] 这些键值对会被修改和删除，这会导致空间的扩容和4.3释放。具体来说，一方面，如果修改后的键值对变大或变小了，就需要占用额外的空间或者释放不用的空间。另一方面，删除的键值对就不再需要内存空间了，此时，就会把空间释放出来，形成空闲空间。

4.3 如何判断是否有内存碎片

Redis是内存数据库，内存利用率的高低直接关系到Redis运行效率的高低。为了让用户能监控到实时的内存使用情况，Redis自身提供了INFO命令，可以用来查询内存使用的详细信息，命令如下：

INFO memory

其中mem\_fragmentation\_ratio的指标，它表示的就是Redis当前的内存碎片率，即两个指标used\_memory\_rss和used\_memory相除的结果。used\_memory\_rss是操作系统实际分配给Redis的物理内存空间，里面就包含了碎片；而used\_memory是Redis为了保存数据实际申请使用的空间。

经验阈值：

- [ ] mem\_fragmentation\_ratio 大于1但小于1.5。这种情况是合理的。这是因为，刚才我介绍的那些因素是难以避免的。毕竟，内因的内存分配器是一定要使用的，分配策略都是通用的，不会轻易修改；而外因由Redis负载决定，也无法限制。所以，存在内存碎片也是正常的。

- [ ] mem\_fragmentation\_ratio 大于 1.5 。这表明内存碎片率已经超过了50%。一般情况下，这个时候，我们就需要采取一些措施来降低内存碎片率了。

4.4 如何清理内存碎片

4.4.1 重启Redis实例

后果：

- [ ] 如果Redis中的数据没有持久化，数据就会丢失；

- [ ] 即使Redis数据持久化了，还需要通过AOF或RDB进行恢复，恢复时长取决于AOF或RDB的大小，如果只有一个Redis实例，恢复阶段无法提供服务。

4.4.2 内存碎片清理（从4.0-RC3版本以后）

当有数据把一块连续的内存空间分割成好几块不连续的空间时，操作系统就会把数据拷贝到别处。此时，数据拷贝需要能把这些数据原来占用的空间都空出来，把原本不连续的内存空间变成连续的空间。

注：碎片清理是有代价的，操作系统需要把多份数据拷贝到新位置，把原有空间释放出来，这会带来时间开销。因为Redis是单线程，在数据拷贝时，Redis只能等着，这就导致Redis无法及时处理请求，性能就会降低。而且，有的时候，数据拷贝还需要注意顺序，会进一步增加Redis的等待时间，导致性能降低。

Redis专门为自动内存碎片清理功机制设置的参数了。通过设置参数，来控制碎片清理的开始和结束时机，以及占用的CPU比例，从而减少碎片清理对Redis本身请求处理的性能影响。

- [ ] 启用自动内存碎片清理，可以把activedefrag配置项设置为yes，命令如下：config set activedefrag yes

- [ ] 分别设置了触发内存清理的一个条件，如果同时满足这两个条件，就开始清理：

active-defrag-ignore-bytes 100mb：表示内存碎片的字节数达到100MB时，开始清理；

active-defrag-threshold-lower 10：表示内存碎片空间占操作系统分配给Redis的总空间比例达到10%时，开始清理。

- [ ] 为了尽可能减少碎片清理对Redis正常请求处理的影响，自动内存碎片清理功能在执行时，还会监控清理操作占用的CPU时间，而且还设置了两个参数，分别用于控制清理操作占用的CPU时间比例的上、下限，既保证清理工作能正常进行，又避免了降低Redis性能。这两个参数具体如下：

• active-defrag-cycle-min 25： 表示自动清理过程所用CPU时间的比例不低于25%，保证清理能正常开展；

• active-defrag-cycle-max 75：表示自动清理过程所用CPU时间的比例不高于75%，一旦超过，就停止清理，从而避免在清理时，大量的内存拷贝阻塞Redis，导致响应延迟升高。

五、Redis缓冲区

缓冲区的功能主要就是用一块内存空间来暂时存放命令数据，以免出现因为数据和命令的处理速度慢于发送速度而导致的数据丢失和性能问题。但因为缓冲区的内存空间有限，如果往里面写入数据的速度持续地大于从里面读取数据的速度，就会导致缓冲区需要越来越多的内存来暂存数据。当缓冲区占用的内存超出了设定的上限阈值时，就会出现缓冲区溢出。

如果发生了溢出，就会丢数据了。随着累积的数据越来越多，缓冲区占用内存空间越来越大，一旦耗尽了Redis实例所在机器的可用内存，就会导致Redis实例崩溃。

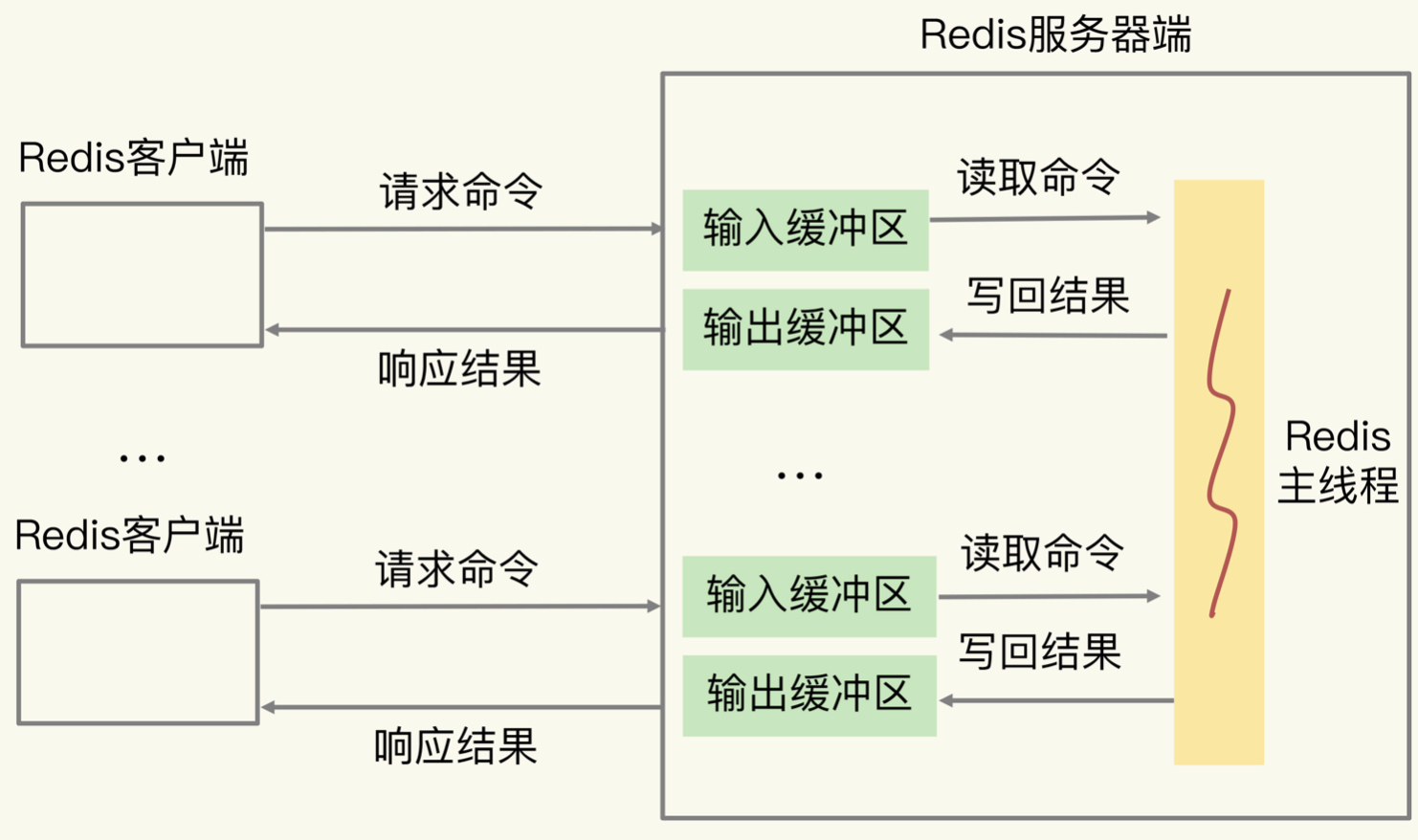
所以毫不夸张地说，缓冲区是用来避免请求或数据丢失的惨案的，但也只有用对了，才能真正起到“避免”的作用。

Redis是典型的client-server架构，所有的操作命令都需要通过客户端发送给服务器端。所以，缓冲区在Redis中的一个主要应用场景，就是在客户端和服务器端之间进行通信时，用来暂存客户端发送的命令数据，或者是服务器端返回给客户端的数据结果。此外，缓冲区的另一个主要应用场景，是在主从节点间进行数据同步时，用来暂存主节点接收的写命令和数据。

5.1 客户端输入和输出缓冲区

为了避免客户端和服务器端的请求发送和处理速度不匹配，服务器端给每个连接的客户端都设置了一个输入缓冲区和输出缓冲区。

输入缓冲区会先把客户端发送过来的命令暂存起来，Redis主线程再从输入缓冲区中读取命令，进行处理。当Redis主线程处理完数据后，会把结果写入到输出缓冲区，再通过输出缓冲区返回给客户端，如下图所示：



5.1.1 如何应对输入缓冲区溢出？

输入缓冲区就是用来暂存客户端发送的请求命令的，所以可能导致溢出的情况主要是下面两种：

- [ ] 写入了bigkey，比如一下子写入了多个百万级别的集合类型数据；

- [ ] 服务器端处理请求的速度过慢，例如，Redis主线程出现了间歇性阻塞，无法及时处理正常发送的请求，导致客户端发送的请求在缓冲区越积越多。

要查看和服务器端相连的每个客户端对输入缓冲区的使用情况，可以使用CLIENT LIST命令：

CLIENT LIST

- [ ] 与服务器端连接的客户端的信息。如果有多个客户端，输出结果中的addr会显示不同客户端的IP和端口号。

- [ ] 与输入缓冲区相关的三个参数：

• cmd，表示客户端最新执行的命令。

• qbuf，表示输入缓冲区已经使用的大小。

• qbuf-free，表示输入缓冲区尚未使用的大小。

qbuf和qbuf-free的总和就是，Redis服务器端当前为已连接的这个客户端分配的缓冲区总大小。如果qbuf很大，而同时qbuf-free很小，这时候输入缓冲区已经占用了很多内存，而且没有什么空闲空间了。此时，客户端再写入大量命令的话，就会引起客户端输入缓冲区溢出，Redis的处理办法就是把客户端连接关闭，结果就是业务程序无法进行数据存取了。

通常情况下，Redis服务器端不止服务一个客户端，当多个客户端连接占用的内存总量，超过了Redis的maxmemory配置项时（例如4GB），就会触发Redis进行数据淘汰。一旦数据被淘汰出Redis，再要访问这部分数据，就需要去后端数据库读取，这就降低了业务应用的访问性能。此外，更糟糕的是，如果使用多个客户端，导致Redis内存占用过大，也会导致内存溢出（out-of-memory）问题，进而会引起Redis崩溃，给业务应用造成严重影响。

避免输入缓冲区溢出。

- [ ] 把缓冲区调大（不可行）

Redis的客户端输入缓冲区大小的上限阈值，在代码中就设定为了1GB。一方面，这个大小对于处理绝大部分客户端的请求已经够用了；另一方面，如果再大的话，Redis就有可能因为客户端占用了过多的内存资源而崩溃。

- [ ] 从数据命令的发送和处理速度入手，也就是前面提到的避免客户端写入bigkey，以及避免Redis主线程阻塞。

5.1.2 如何应对输出缓冲区溢出

Redis为每个客户端设置的输出缓冲区也包括两部分：

- [ ] 一个大小为16KB的固定缓冲空间，用来暂存OK响应和出错信息；

- [ ] 一个可以动态增加的缓冲空间，用来暂存大小可变的响应结果。

发生输出缓冲区溢出的情况：

- [ ] 服务器端返回bigkey的大量结果；

- [ ] 执行了MONITOR命令；

- [ ] 缓冲区大小设置得不合理。使用client-output-buffer-limit设置合理的缓冲区大小上限，或是缓冲区连续写入时间和写入量上限。具体设置的内容包括两方面：

• 设置缓冲区大小的上限阈值；

• 设置输出缓冲区持续写入数据的数量上限阈值，和持续写入数据的时间的上限阈值。

当给普通客户端设置缓冲区大小时，通常可以在Redis配置文件中进行这样的设置：

client-output-buffer-limit normal 0 0 0

其中，normal表示当前设置的是普通客户端，第1个0设置的是缓冲区大小限制，第2个0和第3个0分别表示缓冲区持续写入量限制和持续写入时间限制。

对于普通客户端来说，它每发送完一个请求，会等到请求结果返回后，再发送下一个请求，这种发送方式称为阻塞式发送。在这种情况下，如果不是读取体量特别大的bigkey，服务器端的输出缓冲区一般不会被阻塞的。

所以，通常把普通客户端的缓冲区大小限制，以及持续写入量限制、持续写入时间限制都设置为0，也就是不做限制。

对于订阅客户端来说，一旦订阅的Redis频道有消息了，服务器端都会通过输出缓冲区把消息发给客户端。所以，订阅客户端和服务器间的消息发送方式，不属于阻塞式发送。不过，如果频道消息较多的话，也会占用较多的输出缓冲区空间。

因此，会给订阅客户端设置缓冲区大小限制、缓冲区持续写入量限制，以及持续写入时间限制，可以在Redis配置文件中这样设置：

client-output-buffer-limit pubsub 8mb 2mb 60

其中，pubsub参数表示当前是对订阅客户端进行设置；8mb表示输出缓冲区的大小上限为8MB，一旦实际占用的缓冲区大小要超过8MB，服务器端就会直接关闭客户端的连接；2mb和60表示，如果连续60秒内对输出缓冲区的写入量超过2MB的话，服务器端也会关闭客户端连接。

5.2 主从集群中的缓冲区

5.2.1 复制缓冲区的溢出问题

在全量复制过程中，主节点在向从节点传输RDB文件的同时，会继续接收客户端发送的写命令请求。这些写命令就会先保存在复制缓冲区中，等RDB文件传输完成后，再发送给从节点去执行。主节点上会为每个从节点都维护一个复制缓冲区，来保证主从节点间的数据同步。

所以，如果在全量复制时，从节点接收和加载RDB较慢，同时主节点接收到了大量的写命令，写命令在复制缓冲区中就会越积越多，最终导致溢出。

其实，主节点上的复制缓冲区，本质上也是一个用于和从节点连接的客户端，使用的输出缓冲区。复制缓冲区一旦发生溢出，主节点也会直接关闭和从节点进行复制操作的连接，导致全量复制失败。

如何避免复制缓冲区发生溢出：

- [ ] 控制主节点保存的数据量大小。按通常的使用经验，我们会把主节点的数据量控制在2~4GB，这样可以让全量同步执行得更快些，避免复制缓冲区累积过多命令。

- [ ] 使用client-output-buffer-limit配置项，来设置合理的复制缓冲区大小。设置的依据，就是主节点的数据量大小、主节点的写负载压力和主节点本身的内存大小。

在主节点执行如下命令：

config set client-output-buffer-limit slave 512mb 128mb 60

其中，slave参数表明该配置项是针对复制缓冲区的。512mb代表将缓冲区大小的上限设置为512MB；128mb和60代表的设置是，如果连续60秒内的写入量超过128MB的话，也会触发缓冲区溢出。

在实际应用中设置复制缓冲区的大小时，可以根据写命令数据的大小和应用的实际负载情况（也就是写命令速率），来粗略估计缓冲区中会累积的写命令数据量；然后，再和所设置的复制缓冲区大小进行比较，判断设置的缓冲区大小是否足够支撑累积的写命令数据量。

主节点上复制缓冲区的内存开销，会是每个从节点客户端输出缓冲区占用内存的总和。如果集群中的从节点数非常多的话，主节点的内存开销就会非常大。所以，必须得控制和主节点连接的从节点个数，不要使用大规模的主从集群。

5.2.2 复制积压缓冲区repl\_backlog\_buffer的溢出问题

主节点在把接收到的写命令同步给从节点时，同时会把这些写命令写入复制积压缓冲区。一旦从节点发生网络闪断，再次和主节点恢复连接后，从节点就会从复制积压缓冲区中，读取断连期间主节点接收到的写命令，进而进行增量同步。

- [ ] 复制积压缓冲区是一个大小有限的环形缓冲区。当主节点把复制积压缓冲区写满后，会覆盖缓冲区中的旧命令数据。如果从节点还没有同步这些旧命令数据，就会造成主从节点间重新开始执行全量复制。

- [ ] 为了应对复制积压缓冲区的溢出问题，可以调整复制积压缓冲区的大小，也就是设置repl\_backlog\_size这个参数的值。