16-异步机制：如何避免单线程模型的阻塞？

你好，我是蒋德钧。

Redis之所以被广泛应用，很重要的一个原因就是它支持高性能访问。也正因为这样，我们必须要重视所有可能影响Redis性能的因素（例如命令操作、系统配置、关键机制、硬件配置等），不仅要知道具体的机制，尽可能避免性能异常的情况出现，还要提前准备好应对异常的方案。

所以，从这节课开始，我会用6节课的时间介绍影响Redis性能的5大方面的潜在因素，分别是：

- Redis内部的阻塞式操作；
- CPU核和NUMA架构的影响；
- Redis关键系统配置；
- Redis内存碎片；
- Redis缓冲区。

这节课，我们就先学习了解下Redis内部的阻塞式操作以及应对的方法。

在第3讲中，我们学习过，Redis的网络IO和键值对读写是由主线程完成的。那么，如果在主线程上执行的操作消耗的时间太长，就会引起主线程阻塞。但是，Redis既有服务客户端请求的键值对增删改查操作，也有保证可靠性的持久化操作，还有进行主从复制时的数据同步操作，等等。操作这么多，究竟哪些会引起阻塞呢？

别着急，接下来，我就带你分门别类地梳理下这些操作，并且找出阻塞式操作。

Redis实例有哪些阻塞点？

Redis实例在运行时，要和许多对象进行交互，这些不同的交互就会涉及不同的操作，下面我们来看看和Redis实例交互的对象，以及交互时会发生的操作。

- **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。

为了帮助你理解，我再画一张图来展示下这4类交互对象和具体的操作之间的关系。

接下来，我们来逐个分析下在这些交互对象中，有哪些操作会引起阻塞。

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。

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。

那么，怎么判断操作复杂度是不是高呢？这里有一个最基本的标准，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)，我们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。这些操作可以作为Redis的**第一个阻塞点：集合全量查询和聚合操作**。

除此之外，集合自身的删除操作同样也有潜在的阻塞风险。你可能会认为，删除操作很简单，直接把数据删除就好了，为什么还会阻塞主线程呢？

其实，删除操作的本质是要释放键值对占用的内存空间。你可不要小瞧内存的释放过程。释放内存只是第一步，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，所以，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。

那么，什么时候会释放大量内存呢？其实就是在删除大量键值对数据的时候，最典型的就是删除包含了大量元素的集合，也称为bigkey删除。为了让你对bigkey的删除性能有一个直观的印象，我测试了不同元素数量的集合在进行删除操作时所消耗的时间，如下表所示：

集合类型	10万（8字节）	100万（8字节）	10万（128字节）	100万（128字节）
Hash	50ms	962ms	91ms	1980ms
List	25ms	133ms	29ms	283ms
Set	42ms	821ms	75ms	1347ms
Sorted Set	53ms	809ms	61ms	991ms

从这张表里，我们可以得出三个结论：

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。

经过刚刚的分析，很显然，**bigkey删除操作就是Redis的第二个阻塞点**。删除操作对Redis实例性能的负面影响很大，而且在实际业务开发时容易被忽略，所以一定要重视它。

既然频繁删除键值对都是潜在的阻塞点了，那么，在Redis的数据库级别操作中，清空数据库（例如FLUSHDB和FLUSHALL操作）必然也是一个潜在的阻塞风险，因为它涉及到删除和释放所有的键值对。所以，这就是**Redis的第三个阻塞点：清空数据库**。

2.和磁盘交互时的阻塞点

我之所以把Redis与磁盘的交互单独列为一类，主要是因为磁盘IO一般都是比较费时费力的，需要重点关注。

幸运的是，Redis开发者早已认识到磁盘IO会带来阻塞，所以就把Redis进一步设计为采用子进程的方式生成RDB快照文件，以及执行AOF日志重写操作。这样一来，这两个操作由子进程负责执行，慢速的磁盘IO就不会阻塞主线程了。

但是，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。这就得到了Redis的**第四个阻塞点了：AOF日志同步写**。

3.主从节点交互时的阻塞点

在主从集群中，主库需要生成RDB文件，并传输给从库。主库在复制的过程中，创建和传输RDB文件都是由

子进程来完成的，不会阻塞主线程。但是，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，这就正好撞上了刚才我们分析的**第三个阻塞点。**

此外，从库在清空当前数据库后，还需要把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，所以，**加载RDB文件就成为了Redis的第五个阻塞点**。

4.切片集群实例交互时的阻塞点

最后，当我们部署Redis切片集群时，每个Redis实例上分配的哈希槽信息需要在不同实例间进行传递，同时，当需要进行负载均衡或者有实例增删时，数据会在不同的实例间进行迁移。不过，哈希槽的信息量不大，而数据迁移是渐进式执行的，所以，一般来说，这两类操作对Redis主线程的阻塞风险不大。

不过，如果你使用了Redis Cluster方案，而且同时正好迁移的是bigkey的话，就会造成主线程的阻塞，因为Redis Cluster使用了同步迁移。我将在第33讲中向你介绍不同切片集群方案对数据迁移造成的阻塞的解决方法，这里你只需要知道，当没有bigkey时，切片集群的各实例在进行交互时不会阻塞主线程，就可以了。

好了，你现在已经了解了Redis的各种关键操作，以及其中的阻塞式操作，我们来总结下刚刚找到的五个阻塞点：

- 集合全量查询和聚合操作；
- bigkey删除；
- 清空数据库；
- AOF日志同步写；
- 从库加载RDB文件。

如果在主线程中执行这些操作，必然会导致主线程长时间无法服务其他请求。为了避免阻塞式操作，Redis提供了异步线程机制。所谓的异步线程机制，就是指，Redis会启动一些子线程，然后把一些任务交给这些子线程，让它们在后台完成，而不再由主线程来执行这些任务。使用异步线程机制执行操作，可以避免阻塞主线程。

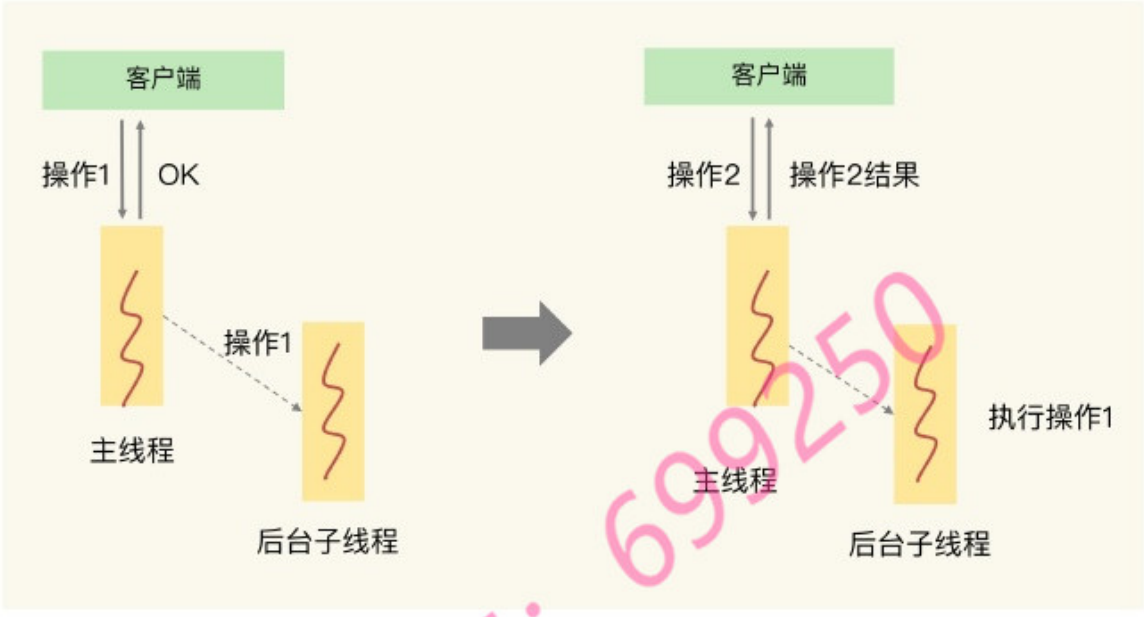
不过，这个时候，问题来了：这五大阻塞式操作都可以被异步执行吗？

哪些阻塞点可以异步执行？

在分析阻塞式操作的异步执行的可行性之前，我们先来了解下异步执行对操作的要求。

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。我再解释下关键路径上的操作是啥。这就是说，客户端把请求发送给Redis后，等着Redis返回数据结果的操作。

这么说可能有点抽象，我画一张图片来解释下。



主线程接收到操作1后，因为操作1并不用给客户端返回具体的数据，所以，主线程可以把它交给后台子线程来完成，同时只要给客户端返回一个“OK”结果就行。在子线程执行操作1的时候，客户端又向Redis实例发送了操作2，而此时，客户端是需要使用操作2返回的数据结果的，如果操作2不返回结果，那么，客户端将一直处于等待状态。

在这个例子中，操作1就不算关键路径上的操作，因为它不用给客户端返回具体数据，所以可以由后台子线程异步执行。而操作2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。而Redis的第一个阻塞点“集合全量查询和聚合操作”都涉及到了读操作，所以，它们是不能进行异步操作了。

我们再来看看删除操作。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。而我们刚才总结的第二个阻塞点“bigkey删除”，和第三个阻塞点“清空数据库”，都是对数据做删除，并不在关键路径上。因此，我们可以使用后台子线程来异步执行删除操作。

对于第四个阻塞点“AOF日志同步写”来说，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。所以，我们也可以启动一个子线程来执行AOF日志的同步写，而不用让主线程等待AOF日志的写完成。

最后，我们再来看下“从库加载RDB文件”这个阻塞点。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。所以，这个操作也属于关键路径上的操作，我们必须让从库的主线程来执行。

对于Redis的五大阻塞点来说，除了“集合全量查询和聚合操作”和“从库加载RDB文件”，其他三个阻塞点涉及的操作都不在关键路径上，所以，我们可以使用Redis的异步子线程机制来实现bigkey删除，清空数据库，以及AOF日志同步写。

那么，Redis实现的异步子线程机制具体是怎么执行呢？

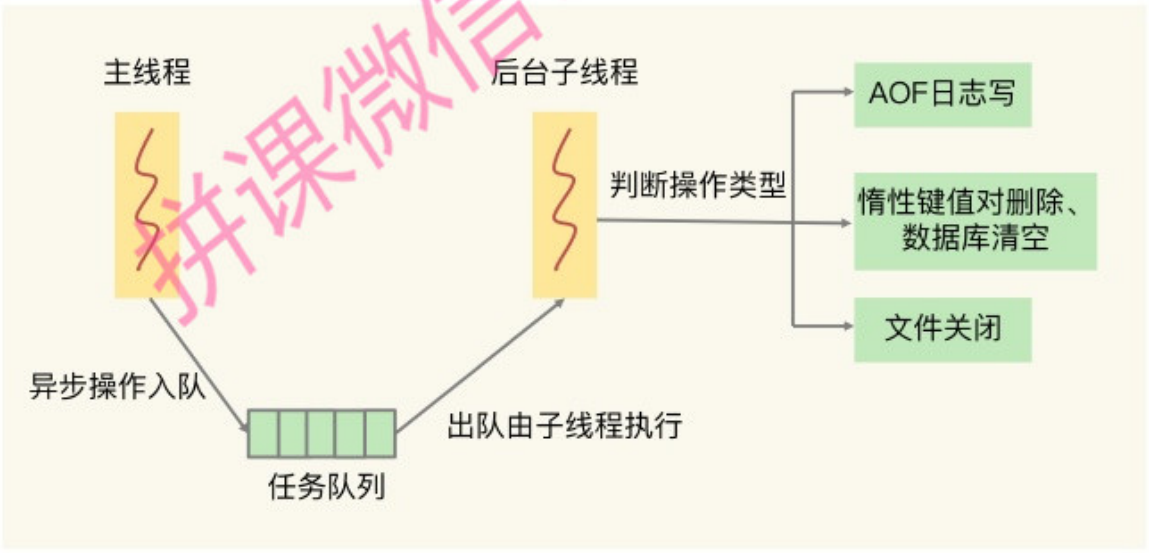
Redis主线程启动后，会使用操作系统提供的pthread_create函数创建3个子线程，分别由它们负责AOF日志写操作、键值对删除以及文件关闭的异步执行。

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）。此时，删除或清空操作不会阻塞主线程，这就避免了对主线程的性能影响。

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。

下面这张图展示了Redis中的异步子线程执行机制，你可以再看下，加深印象。



这里有个地方需要你注意一下，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，如下所示：

```
FLUSHDB ASYNC
FLUSHALL AYSNC
```

小结

这节课，我们学习了Redis实例运行时的4大类交互对象：客户端、磁盘、主从库实例、切片集群实例。基于

这4大类交互对象，我们梳理了会导致Redis性能受损的5大阻塞点，包括集合全量查询和聚合操作、bigkey删除、清空数据库、AOF日志同步写，以及从库加载RDB文件。

在这5大阻塞点中，bigkey删除、清空数据库、AOF日志同步写不属于关键路径操作，可以使用异步子线程机制来完成。Redis在运行时会创建三个子线程，主线程会通过一个任务队列和三个子线程进行交互。子线程会根据任务的具体类型，来执行相应的异步操作。

不过，异步删除操作是Redis 4.0以后才有的功能，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。

例如，对于Hash类型的bigkey删除，你可以使用HSCAN命令，每次从Hash集合中获取一部分键值对（例如200个），再使用HDEL删除这些键值对，这样就可以把删除压力分摊到多次操作中，那么，每次删除操作的耗时就不会太长，也就不会阻塞主线程了。

最后，我想再提一下，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。对于这两个阻塞点，我也给你两个小建议。

- 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。

每课一问

按照惯例，我给你提一个小问题：我们今天学习了关键路径上的操作，你觉得，Redis的写操作（例如SET、HSET、SADD等）是在关键路径上吗？

欢迎在留言区写下你的思考和答案，我们一起交流讨论。如果你觉得今天的内容对你有所帮助，也欢迎你帮我分享给更多人，我们下节课见。

精选留言：

- Kaito 2020-09-14 00:09:57
  Redis的写操作（例如SET，HSET，SADD等）是在关键路径上吗？

  我觉得这需要客户端根据业务需要来区分：

  1、如果客户端依赖操作返回值的不同，进而需要处理不同的业务逻辑，那么HSET和SADD操作算关键路径，而SET操作不算关键路径。因为HSET和SADD操作，如果field或member不存在时，Redis结果会返回1，否则返回0。而SET操作返回的结果都是OK，客户端不需要关心结果有什么不同。

  2、如果客户端不关心返回值，只关心数据是否写入成功，那么SET/HSET/SADD不算关键路径，多次执行这些命令都是幂等的，这种情况下可以放到异步线程中执行。

  3、但是有种例外情况，如果Redis设置了maxmemory，但是却没有设置淘汰策略，这三个操作也都算关键路径。因为如果Redis内存超过了maxmemory，再写入数据时，Redis返回的结果是OOM error，这种情况下，客户端需要感知有错误发生才行。

  另外，我查阅了lazy-free相关的源码，发现有很多细节需要补充下：

1、lazy-free是4.0新增的功能，但是默认是关闭的，需要手动开启。

2、手动开启lazy-free时，有4个选项可以控制，分别对应不同场景下，要不要开启异步释放内存机制：
a) lazyfree-lazy-expire：key在过期删除时尝试异步释放内存
b) lazyfree-lazy-eviction：内存达到maxmemory并设置了淘汰策略时尝试异步释放内存
c) lazyfree-lazy-server-del：执行RENAME/MOVE等命令或需要覆盖一个key时，删除旧key尝试异步释放内存
d) replica-lazy-flush：主从全量同步，从库清空数据库时异步释放内存

3、即使开启了lazy-free，如果直接使用DEL命令还是会同步删除key，只有使用UNLINK命令才会可能异步删除key。

4、这也是最关键的一点，上面提到开启lazy-free的场景，除了replica-lazy-flush之外，其他情况都只是*可能*去异步释放key的内存，并不是每次必定异步释放内存的。

开启lazy-free后，Redis在释放一个key的内存时，首先会评估代价，如果释放内存的代价很小，那么就直接在主线程中操作了，没必要放到异步线程中执行（不同线程传递数据也会有性能消耗）。

什么情况才会真正异步释放内存？这和key的类型、编码方式、元素数量都有关系（详细可参考源码中的lazyfreeGetFreeEffort函数）：

a) 当Hash/Set底层采用哈希表存储（非ziplist/int编码存储）时，并且元素数量超过64个
b) 当ZSet底层采用跳表存储（非ziplist编码存储）时，并且元素数量超过64个
c) 当List链表节点数量超过64个（注意，不是元素数量，而是链表节点的数量，List的实现是在每个节点包含了若干个元素的数据，这些元素采用ziplist存储）

只有以上这些情况，在删除key释放内存时，才会真正放到异步线程中执行，其他情况一律还是在主线程操作。

也就是说String（不管内存占用多大）、List（少量元素）、Set（int编码存储）、Hash/ZSet（ziplist编码存储）这些情况下的key在释放内存时，依旧在主线程中操作。

可见，即使开启了lazy-free，String类型的bigkey，在删除时依旧有阻塞主线程的风险。所以，即便Redis提供了lazy-free，我建议还是尽量不要在Redis中存储bigkey。

个人理解Redis在设计评估释放内存的代价时，不是看key的内存占用有多少，而是关注释放内存时的工作量有多大。从上面分析基本能看出，如果需要释放的内存是连续的，Redis作者认为释放内存的代价比较低，就放在主线程做。如果释放的内存不连续（大量指针类型的数据），这个代价就比较高，所以才会放在异步线程中去执行。

如果我的理解有偏差，还请老师和大家指出！ [29赞]

- Spring4J 2020-09-14 14:44:09
Redis的异步子线程机制就跟java里面的线程池原理差不多，都是主线程封装任务到队列中，子线程到队列中拉取任务异步执行，运用了生产者消费者的模型 [1赞]

- test 2020-09-14 11:36:58
写操作是否在关键路径，需要看使用方是否需要确认写入已经完成。 [1赞]

- 漫步oo0云端 2020-09-14 07:04:34
  今天学习了5种阻塞点，请问老师，后面会学习，当redis发生阻塞时如何分析是什么操作导致的这个技能吗？ [1赞]

- 小袁 2020-09-14 19:41:12
  我听说网络部分已经有多线程实现，这里已经不是问题了吧？

- 那时刻 2020-09-14 10:16:12
  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，然后写入AOF日志。请问老师，如果写入操作比较频繁，是否也会引起redis延迟增大呢？

- MClink 2020-09-14 09:40:19
  想问一个问题，Redis 自己会不会维护一个可用内存区域呢，我看过一些工具的设计，有些都是向 os 申请到内存空间后，都会通过标记作用（即标记为可复用，而不是真正的释放内存）来管理以往申请过的内存。文中提到的空闲内存块的链表，应该是 os 层级的把，那 Redis 有没有相关的处理呢？