链滴

Redis 笔记四

作者：matthewhan

原文链接：https://ld246.com/article/1626342657094

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

<h2 id="Q-异步机制">Q：异步机制</h2>
<h3 id="Redis-实例有哪些阻塞点-">Redis 实例有哪些阻塞点？</h3>
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，下面我们看看和 Redis 实例交互的对象，以及交互时会发生的操作。
<ol start="0">
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。

<h4 id="0--和客户端交互时的阻塞点">0. 和客户端交互时的阻塞点</h4>
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。
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edis 的第一个阻塞点：集合全量查询和聚合操作。
bigkey 删除操作就是 Redis 的第二个阻塞点。删除操作对 Redis 实例性的负面影响很大，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 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 的三个阻塞点：清空数据库。
<h4 id="1--和磁盘交互时的阻塞点">1. 和磁盘交互时的阻塞点</h4>
Redis 的第四个阻塞点了：AOF 日志同步写。
<h4 id="2--主从节点交互时的阻塞点">2. 主从节点交互时的阻塞点</h4>
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 RD 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 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</strng>。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 第五个阻塞点。
<h4 id="3--切片集群实例交互时的阻塞点">3. 切片集群实例交互时的阻塞点</h4>
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，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。
除非迁移的是 bigkey。
五个阻塞点：
<ol start="0">
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。

<h3 id="哪些阻塞点可以异步执行-">哪些阻塞点可以异步执行？</h3>
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

6b0b61-a69c3449.webp?imageView2/2/interlace/1/format/jpg">
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 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，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。/p>
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 的异步子线程机来实现 bigkey 删除，清空数据库，以及 AOF 日志同步写。
<h3 id="异步的子线程机制">异步的子线程机制</h3>
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，分别由它负责 AOF 日志写操作、键值对删除以及文件关闭的异步执行。

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程异步地清空数据库，如下所示：
<pre><code class="language-bash highlight-chroma">FLUSHDB ASYNC
FLUSHALL AYSNC
</code></pre>

<h2 id="Q-CPU-与-Redis">Q：CPU 与 Redis</h2>
<h3 id="主流的-CPU-架构">主流的 CPU 架构</h3>
一个 CPU 处理器中一般有多个运行核心，我们把一个运行核心称为一个物理核，每个物理核都以运行应用程序。每个物理核都拥有私有的一级缓存（Level 1 cache，简称 L1 cache），包括一级令缓存和一级数据缓存，以及私有的二级缓存（Level 2 cache，简称 L2 cache）。不同的物理核还共享一个共同的三级缓存（Level 3 cache，简称为 L3 cache）。
另外，现在主流的 CPU 处理器中，每个物理核通常都会运行两个超线程，也叫作逻辑核。同一物理核的逻辑核会共享使用 L1、L2 缓存。

在主流的服务器上，一个 CPU 处理器会有 10 到 20 多个物理核。同时，为了提升服务器的处理力，服务器上通常还会有多个 CPU 处理器（也称为多 CPU Socket），每个处理器有自己的物理核包括 L1、L2 缓存），L3 缓存，以及连接的内存，同时，不同处理器间通过总线连接。

在多 CPU 架构上，应用程序可以在不同的处理器上运行。在刚才的图中，Redis 可以先在 Socke 1 上运行一段时间，然后再被调度到 Socket 2 上运行。

但是，有个地方需要你注意一下：如果应用程序先在一个 Socket 上运行，并且把数据保存到了存，然后被调度到另一个 Socket 上运行，此时，应用程序再进行内存访问时，就需要访问之前 Socke 上连接的内存，这种访问属于远端内存访问。和访问 Socket 直接连接的内存相比，远端内存访问会加应用程序的延迟。
在多 CPU 架构下，一个应用程序访问所在 Socket 的本地内存和访问远端内存的延迟并不一致所以，我们也把这个架构称为非统一内存访问架构（Non-Uniform Memory Access，NUMA 架构。
<h3 id="CPU-多核对-Redis-性能的影响">CPU 多核对 Redis 性能的影响</h3>
CPU 的 context switch 次数比较多会影响 Redis 的读写性能。
context switch 是指线程的上下文切换，这里的上下文就是线程的运行时信息。在 CPU 多核的境中，一个线程先在一个 CPU 核上运行，之后又切换到另一个 CPU 核上运行，这时就会发生 contex switch。
如果在 CPU 多核场景下，Redis 实例被频繁调度到不同 CPU 核上运行的话，那么，对 Redis 例的请求处理时间影响就更大了。每调度一次，一些请求就会受到运行时信息、指令和数据重新加载程的影响，这就会导致某些请求的延迟明显高于其他请求。
我们可以使用 taskset 命令把一个程序绑定在一个核上运行。
<pre><code class="language-bash highlight-chroma"><span c ass="highlight-cl">taskset -c 0 ./redis-server
</code></pre>
<h3 id="CPU-的-NUMA-架构对-Redis-性能的影响">CPU 的 NUMA 架构对 Redis 性能的影响</h >
如果网络中断处理程序和 Redis 实例各自所绑的 CPU 核不在同一个 CPU Socket 上，那么，Red s 实例读取网络数据时，就需要跨 CPU Socket 访问内存，这个过程会花费较多时间。
所以，为了避免 Redis 跨 CPU Socket 访问网络数据，我们最好把网络中断程序和 Redis 实例在同一个 CPU Socket 上，这样一来，Redis 实例就可以直接从本地内存读取网络数据了。
在 CPU 的 NUMA 架构下，对 CPU 核的编号规则，并不是先把一个 CPU Socket 中的所有逻辑编完，再对下一个 CPU Socket 中的逻辑核编码，而是先给每个 CPU Socket 中每个物理核的第一个辑核依次编号，再给每个 CPU Socket 中的物理核的第二个逻辑核依次编号。
<pre><code class="language-bash highlight-chroma"><span c ass="highlight-cl">lscpu

Architecture: x86_ 4
...
NUMA node0 CP (s): 0-5,12-17
NUMA node1 CP (s): 6-11,18-23
...
</code></pre>
<h3 id="绑核的风险和解决方案">绑核的风险和解决方案</h3>
把 Redis 实例和物理核绑定，可以让主线程、子进程、后台线程共享使用 2 个逻辑核，可以在定程度上缓解 CPU 资源竞争。但是，因为只用了 2 个逻辑核，它们相互之间的 CPU 竞争仍然还会在。如果你还想进一步减少 CPU 竞争，我再给你介绍一种方案。
另一种方案：修改 Redis 源码（略）