Redis网络编程精华全面揭秘

极客重生 2022-08-30 22:50 Posted on 广东

Editor's Note

从优秀开源软件redis学习网络编程和设计

The following article is from 远赴星辰 Author 妖道

远赴星辰

风雪兼程，赶赴一场星辰。

目录

导言

在目前的技术选型中，Redis 俨然已经成为了系统高性能缓存方案的事实标准，因此现在 Redis 也成为了后端开发的基本技能树之一，Redis 的底层原理也顺理成章地成为了必须学习的知识。

Redis 从本质上来讲是一个网络服务器，而对于一个网络服务器来说，网络模型是它的精华，搞懂了一个网络服务器的网络模型，你也就搞懂了它的本质。

本文通过层层递进的方式，介绍了 Redis 网络模型的版本变更历程，剖析了其从单线程进化到多线程的工作原理，此外，还一并分析并解答了 Redis 的网络模型的很多抉择背后的思考，帮助读者能更深刻地理解 Redis 网络模型的设计。

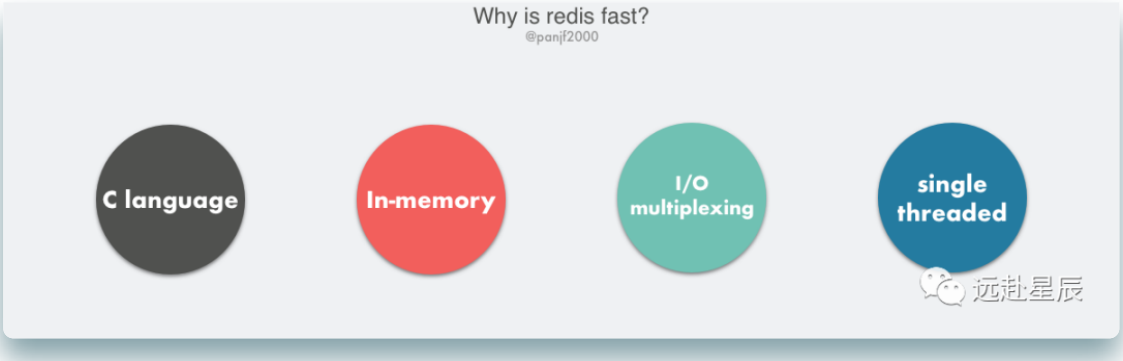
Redis 有多快?

根据官方的 benchmark，通常来说，在一台普通硬件配置的 Linux 机器上跑单个 Redis 实例，处理简单命令（时间复杂度 O(N) 或者 O(log(N))），QPS 可以达到 8w+，而如果使用 pipeline 批处理功能，则 QPS 至高能达到 100w。

仅从性能层面进行评判，Redis 完全可以被称之为高性能缓存方案。

Redis 为什么快?

Redis 的高性能得益于以下几个基础：



- **C 语言实现**，虽然 C 对 Redis 的性能有助力，但语言并不是最核心因素。
- **纯内存 I/O**，相较于其他基于磁盘的 DB，Redis 的纯内存操作有着天然的性能优势。
- **I/O 多路复用**，基于 epoll/select/kqueue 等 I/O 多路复用技术，实现高吞吐的网络 I/O。

- **单线程模型**，单线程无法利用多核，但是从另一个层面来说则避免了多线程频繁上下文切换，以及同步机制如锁带来的开销。

Redis 为何选择单线程?

Redis 的核心网络模型选择用单线程来实现，这在一开始就引起了很多人的不解，Redis 官方的对于此的回答是：

> It's not very frequent that CPU becomes your bottleneck with Redis, as usually Redis is either memory or network bound. For instance, using pipelining Redis running on an average Linux system can deliver even 1 million requests per second, so if your application mainly uses O(N) or O(log(N)) commands, it is hardly going to use too much CPU. ”

核心意思就是，对于一个 DB 来说，CPU 通常不会是瓶颈，因为大多数请求不会是 CPU 密集型的，而是 I/O 密集型。具体到 Redis 的话，如果不考虑 RDB/AOF 等持久化方案，Redis 是完全的纯内存操作，执行速度是非常快的，因此这部分操作通常不会是性能瓶颈，Redis 真正的性能瓶颈在于网络 I/O，也就是客户端和服务端之间的网络传输延迟，因此 Redis 选择了单线程的 I/O 多路复用来实现它的核心网络模型。

上面是比较笼统的官方答案，实际上更加具体的选择单线程的原因可以归纳如下：

避免过多的上下文切换开销

多线程调度过程中必然需要在 CPU 之间切换线程上下文 context，而上下文的切换又涉及程序计数器、堆栈指针和程序状态字等一系列的寄存器置换、程序堆栈重置甚至是高速缓存、TLB 快表的汰换，如果是进程内的多线程切换还好一些，因为单一进程内多线程共享进程地址空间，因此线程上下文比之进程上下文要小得多，如果是跨进程调度，则需要切换掉整个进程地址空间。

如果是单线程则可以规避进程内频繁的线程切换开销，因为程序始终运行在进程中单个线程内，没有多线程切换的场景。

避免同步机制的开销

如果 Redis 选择多线程模型，又因为 Redis 是一个数据库，那么势必涉及到底层数据同步的问题，则必然会引入某些同步机制，比如锁，而我们知道 Redis 不仅仅提供了简单的 key-value 数据结构，还有 list、set 和 hash 等等其他丰富的数据结构，而不同的数据结构对同步访问的加锁粒度又不尽相同，可能会导致在操作数据过程中带来很多加锁解锁的开销，增加程序复杂度的同时还会降低性能。

简单可维护

Redis 的作者 Salvatore Sanfilippo（别称 antirez）对 Redis 的设计和代码有着近乎偏执的简洁性理念，你可以在阅读 Redis 的源码或者给 Redis 提交 PR 的之时感受到这份偏执。因此代码的简单可维护性必然是 Redis 早期的核心准则之一，而引入多线程必然会导致代码的复杂度上升和可维护性下降。

事实上，多线程编程也不是那么尽善尽美，首先多线程的引入会使得程序不再保持代码逻辑上的串行性，代码执行的顺序将变成不可预测的，稍不注意就会导致程序出现各种并发编程的问题；其次，多线程模式也使得程序调试更加复杂和麻烦。网络上有一幅很有意思的图片，生动形象地描述了并发编程面临的窘境。

你期望的多线程编程 VS 实际上的多线程编程：

你期望的多线程VS实际上的多线程

前面我们提到引入多线程必须的同步机制，如果 Redis 使用多线程模式，那么所有的底层数据结构都必须实现成线程安全的，这无疑又使得 Redis 的实现变得更加复杂。

总而言之，Redis 选择单线程可以说是多方博弈之后的一种权衡：在保证足够的性能表现之下，使用单线程保持代码的简单和可维护性。

Redis 真的是单线程?

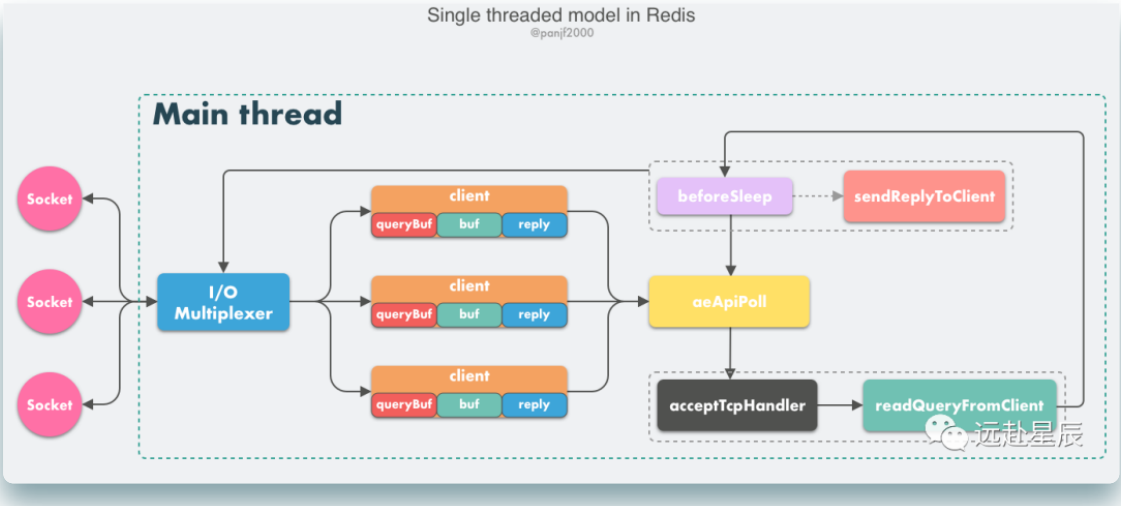
在讨论这个问题之前，我们要先明确『单线程』这个概念的边界：它的覆盖范围是核心网络模型，抑或是整个 Redis？如果是前者，那么答案是肯定的，在 Redis 的 v6.0 版本正式引入多线程之前，其网络模型一直是单线程模式的；如果是后者，那么答案则是否定的，Redis 早在 v4.0 就已经引入了多线程。

因此，当我们讨论 Redis 的多线程之时，有必要对 Redis 的版本划出两个重要的节点：

1. Redis v4.0 (引入多线程处理异步任务)
2. Redis v6.0 (正式在网络模型中实现 I/O 多线程)

单线程事件循环

我们首先来剖析一下 Redis 的核心网络模型，从 Redis 的 v1.0 到 v6.0 版本之前，Redis 的核心网络模型一直是一个典型的单 Reactor 模型：利用 epoll/select/kqueue 等多路复用技术，在单线程的事件循环中不断去处理事件（客户端请求），最后回写响应数据到客户端：



这里有几个核心的概念需要学习：

- **client**：客户端对象，Redis 是典型的 CS 架构（Client <---> Server），客户端通过 socket 与服务端建立网络通道然后发送请求命令，服务端执行请求的命令并回复。Redis 使用结构体 client 存储客户端的所有相关信息，包括但不限于 封装的套接字连接 -- *conn ， 当前选择的数据库指针 -- *db ， 读入缓冲区 -- querybuf ， 写出缓冲区 -- buf ， 写出数据链表 -- reply 等。
- **aeApiPoll**：I/O 多路复用 API，是基于 epoll_wait/select/kevent 等系统调用的封装，监听等待读写事件触发，然后处理，它是事件循环（Event Loop）中的核心函数，是事件驱动得以运行的基础。
- **acceptTcpHandler**：连接应答处理器，底层使用系统调用 accept 接受来自客户端的新连接，并为新连接注册绑定命令读取处理器，以备后续处理新的客户端 TCP 连接；除了这个处理器，还有对应的 acceptUnixHandler 负责处理 Unix Domain Socket 以及 acceptTLSHandler 负责处理 TLS 加密连接。
- **readQueryFromClient**：命令读取处理器，解析并执行客户端的请求命令。
- **beforeSleep**：事件循环中进入 aeApiPoll 等待事件到来之前会执行的函数，其中包含一些日常的任务，比如把 client->buf 或者 client->reply （后面会解释为什么这里需要两个缓冲区）中的响应写回到客户端，持久化 AOF 缓冲区的数据到磁盘等，相对应的还有一个 afterSleep 函数，在 aeApiPoll 之后执行。
- **sendReplyToClient**：命令回复处理器，当一次事件循环之后写出缓冲区中还有数据残留，则这个处理器会被注册绑定到相应的连接上，等连接触发写就绪事件时，它会将写出缓冲区剩余

的数据回写到客户端。

Redis 内部实现了一个高性能的事件库 --- AE，基于 epoll/select/kqueue/evport 四种事件驱动技术，实现 Linux/MacOS/FreeBSD/Solaris 多平台的高性能事件循环模型。Redis 的核心网络模型正式构筑在 AE 之上，包括 I/O 多路复用、各类处理器的注册绑定，都是基于此才得以运行。

至此，我们可以描绘出客户端向 Redis 发起请求命令的工作原理：

1. Redis 服务器启动，开启主线程事件循环（Event Loop），注册 `acceptTcpHandler` 连接应答处理器到用户配置的监听端口对应的文件描述符，等待新连接到来；
2. 客户端和服务端建立网络连接；
3. `acceptTcpHandler` 被调用，主线程使用 AE 的 API 将 `readQueryFromClient` 命令读取处理器绑定到新连接对应的文件描述符上，并初始化一个 `client` 绑定这个客户端连接；
4. 客户端发送请求命令，触发读就绪事件，主线程调用 `readQueryFromClient` 通过 socket 读取客户端发送过来的命令存入 `client->querybuf` 读入缓冲区；
5. 接着调用 `processInputBuffer`，在其中使用 `processInlineBuffer` 或者 `processMultibulkBuffer` 根据 Redis 协议解析命令，最后调用 `processCommand` 执行命令；
6. 根据请求命令的类型（SET, GET, DEL, EXEC 等），分配相应的命令执行器去执行，最后调用 `addReply` 函数族的一系列函数将响应数据写入到对应 `client` 的写出缓冲区：`client->buf` 或者 `client->reply`，`client->buf` 是首选的写出缓冲区，固定大小 16KB，一般来说可以缓冲足够多的响应数据，但是如果客户端在时间窗口内需要响应的数据非常大，那么则会自动切换到 `client->reply` 链表上去，使用链表理论上能够保存无限大的数据（受限于机器的物理内存），最后把 `client` 添加进一个 LIFO 队列 `clients_pending_write`；
7. 在事件循环（Event Loop）中，主线程执行 `beforeSleep` ⟶ `handleClientsWithPendingWrites`，遍历 `clients_pending_write` 队列，调用 `writeToClient` 把 `client` 的写出缓冲区里的数据回写到客户端，如果写出缓冲区还有数据遗留，则注册 `sendReplyToClient` 命令回复处理器到该连接的写就绪事件，等待客户端可写时在事件循环中再继续回写残余的响应数据。

对于那些想利用多核优势提升性能的用户来说，Redis 官方给出的解决方案也非常简单粗暴：在同一个机器上多跑几个 Redis 实例。事实上，为了保证高可用，线上业务一般不太可能会是单机模式，更加常见的是利用 Redis 分布式集群多节点和数据分片负载均衡来提升性能和保证高可用。

多线程异步任务

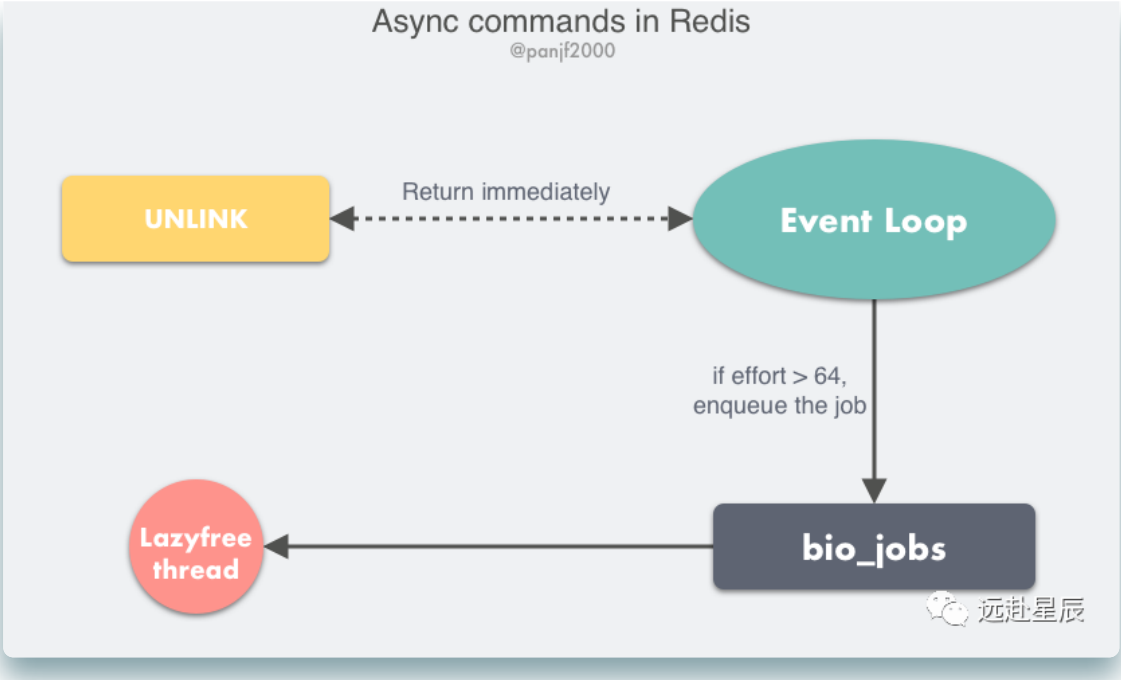
以上便是 Redis 的核心网络模型，这个单线程网络模型一直到 Redis v6.0 才改造成多线程模式，但这并不意味着整个 Redis 一直都只是单线程。

Redis 在 v4.0 版本的时候就已经引入了的多线程来做一些异步操作，此举主要针对的是那些非常耗时的命令，通过将这些命令的执行进行异步化，避免阻塞单线程的事件循环。

我们知道 Redis 的 DEL 命令是用来删除掉一个或多个 key 储存的值，它是一个阻塞的命令，大多数情况下你要删除的 key 里存的值不会特别多，最多也就几十上百个对象，所以可以很快执行完，但是如果你要删的是一个超大的键值对，里面有几百万个对象，那么这条命令可能会阻塞至少好几秒，又因为事件循环是单线程的，所以会阻塞后面的其他事件，导致吞吐量下降。

Redis 的作者 antirez 为了解决这个问题进行了很多思考，一开始他想的办法是一种渐进式的方案：利用定时器和数据游标，每次只删除一小部分的数据，比如 1000 个对象，最终清除掉所有的数据，但是这种方案有个致命的缺陷，如果同时还有其他客户端往某个正在被渐进式删除的 key 里继续写入数据，而且删除的速度跟不上写入的数据，那么将会无止境地消耗内存，虽然后来通过一个巧妙的办法解决了，但是这种实现使 Redis 变得更加复杂，而多线程看起来似乎是一个水到渠成的解决方案：简单、易理解。于是，最终 antirez 选择引入多线程来实现这一类非阻塞的命令。更多 antirez 在这方面的思考可以阅读一下他发表的博客：Lazy Redis is better Redis。

于是，在 Redis v4.0 之后增加了一些的非阻塞命令如 UNLINK 、 FLUSHALL ASYNC 、 FLUSHDB ASYNC 。



`UNLINK` 命令其实就是 `DEL` 的异步版本，它不会同步删除数据，而只是把 key 从 keyspace 中暂时移除掉，然后将任务添加到一个异步队列，最后由后台线程去删除，不过这里需要考虑一种情况是如果用 `UNLINK` 去删除一个很小的 key，用异步的方式去做反而开销更大，所以它会先计算一个开销的阀值，只有当这个值大于 64 才会使用异步的方式去删除 key，对于基本的数据类型如 List、Set、Hash 这些，阀值就是其中存储的对象数量。

Redis 多线程网络模型

前面提到 Redis 最初选择单线程网络模型的理由是：CPU 通常不会成为性能瓶颈，瓶颈往往是**内存**和**网络**，因此单线程足够了。那么为什么现在 Redis 又要引入多线程呢？很简单，就是 Redis 的网络 I/O 瓶颈已经越来越明显了。

随着互联网的飞速发展，互联网业务系统所要处理的线上流量越来越大，Redis 的单线程模式会导致系统消耗很多 CPU 时间在网络 I/O 上从而降低吞吐量，要提升 Redis 的性能有两个方向：

- 优化网络 I/O 模块
- 提高机器内存读写的速度

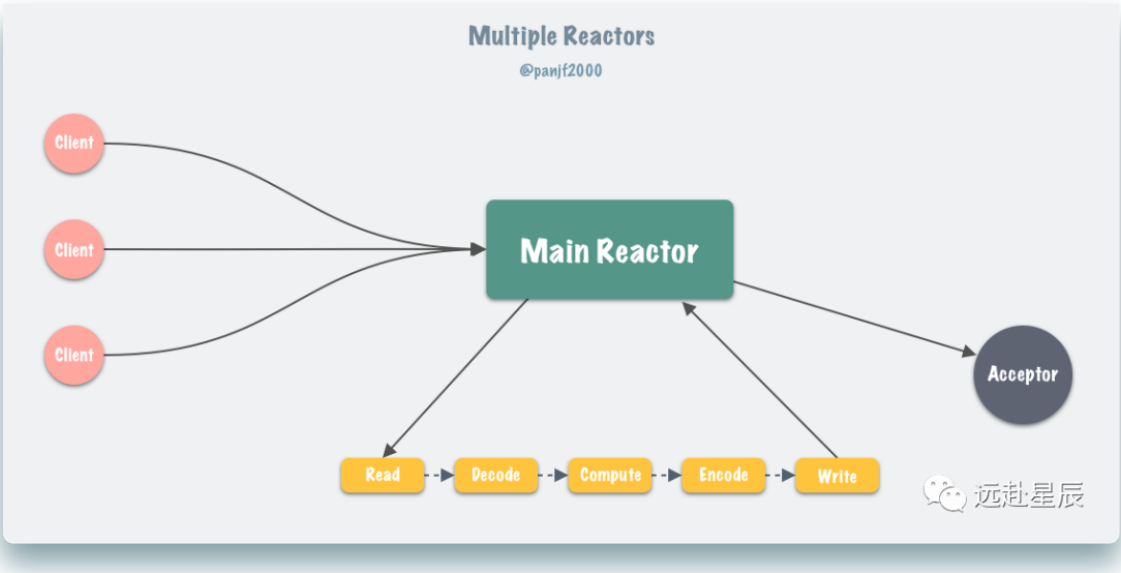
后者依赖于硬件的发展，暂时无解。所以只能从前者下手，网络 I/O 的优化又可以分为两个方向：

- 零拷贝技术或者 DPDK 技术
- 利用多核优势

零拷贝技术有其局限性，无法完全适配 Redis 这一类复杂的网络 I/O 场景，更多网络 I/O 对 CPU 时间的消耗和 Linux 零拷贝技术，可以阅读我的另一篇文章：Linux I/O 原理和 Zero-copy 技术全面揭秘。而 DPDK 技术通过旁路网卡 I/O 绕过内核协议栈的方式又太过于复杂以及需要内核甚至是硬件的支持。

因此，利用多核优势成为了优化网络 I/O 性价比最高的方案。

6.0 版本之后，Redis 正式在核心网络模型中引入了多线程，也就是所谓的 *I/O threading*，至此 Redis 真正拥有了多线程模型。前一小节，我们了解了 Redis 在 6.0 版本之前的单线程事件循环模型，实际上就是一个非常经典的 Reactor 模型：



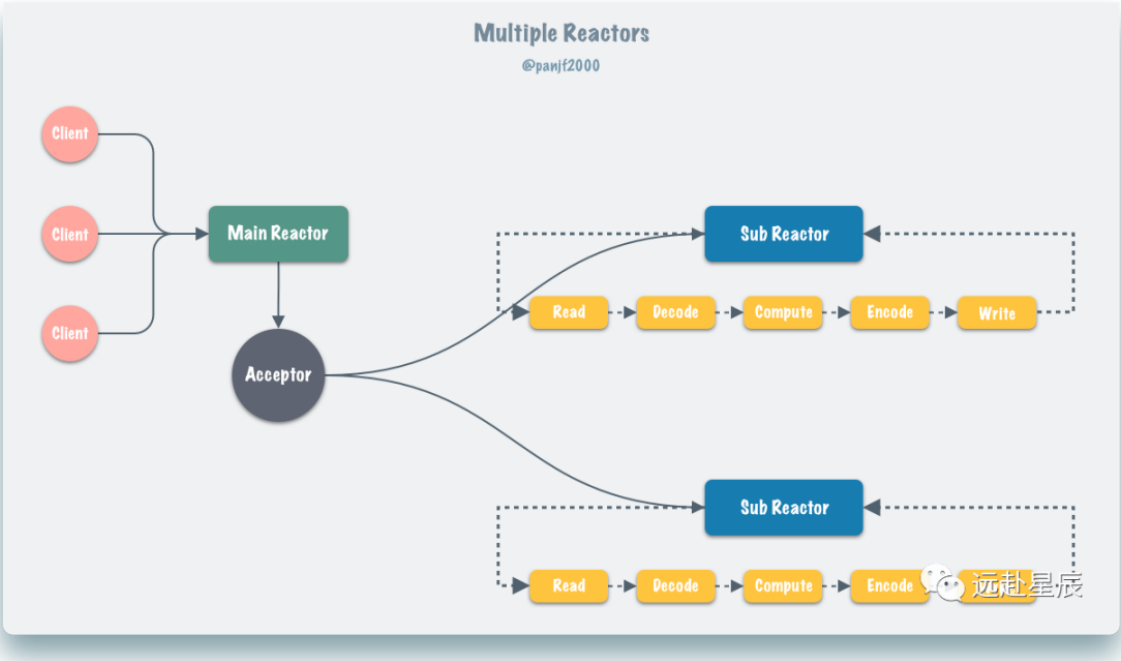
目前 Linux 平台上主流的高性能网络库/框架中，大都采用 Reactor 模式，比如 netty、libevent、libuv、POE(Perl)、Twisted(Python)等。

Reactor 模式本质上指的是使用 I/O 多路复用(I/O multiplexing) + 非阻塞 I/O(non-blocking I/O) 的模式。

更多关于 Reactor 模式的细节可以参考我之前的文章：Go netpoller 原生网络模型之源码全面揭秘，Reactor 网络模型那一小节，这里不再赘述。

Redis 的核心网络模型在 6.0 版本之前，一直是单 Reactor 模式：所有事件的处理都在单个线程内完成，虽然在 4.0 版本中引入了多线程，但是那个更像是针对特定场景（删除超大 key 值等）而打的补丁，并不能被视作核心网络模型的多线程。

通常来说，单 Reactor 模式，引入多线程之后会进化为 Multi-Reactors 模式，基本工作模式如下：

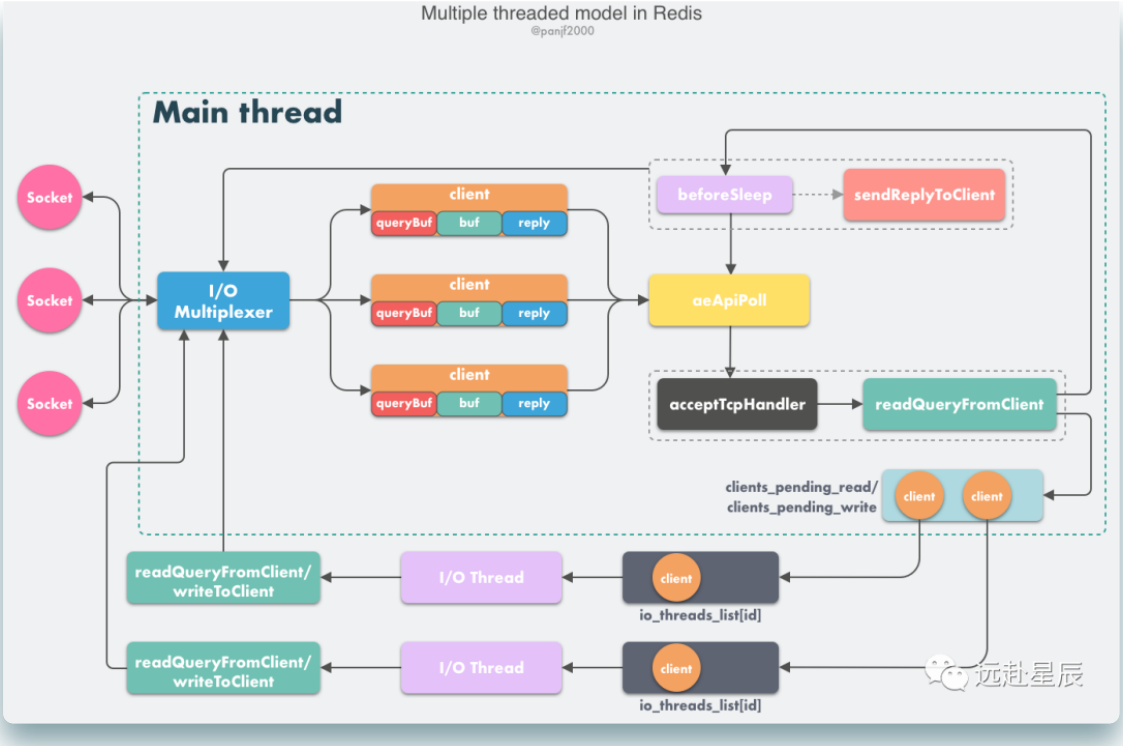


区别于单 Reactor 模式，这种模式不再是单线程的事件循环，而是有多个线程（Sub Reactors）各自维护一个独立的事件循环，由 Main Reactor 负责接收新连接并分发给 Sub Reactors 去独立处理，最后 Sub Reactors 回写响应给客户端。

Multiple Reactors 模式通常也可以等同于 Master-Workers 模式，比如 Nginx 和 Memcached 等就是采用这种多线程模型，虽然不同的项目实现细节略有区别，但总体来说模式是一致的。

设计思路

Redis 虽然也实现了多线程，但是却不是标准的 Multi-Reactors/Master-Workers 模式，这其中的缘由我们后面会分析，现在我们先看一下 Redis 多线程网络模型的总体设计：



1. Redis 服务器启动，开启主线程事件循环（Event Loop），注册 acceptTcpHandler 连接应答处理器到用户配置的监听端口对应的文件描述符，等待新连接到来；
2. 客户端和服务端建立网络连接；
3. acceptTcpHandler 被调用，主线程使用 AE 的 API 将 readQueryFromClient 命令读取处理器绑定到新连接对应的文件描述符上，并初始化一个 client 绑定这个客户端连接；
4. 客户端发送请求命令，触发读就绪事件，服务端主线程不会通过 socket 去读取客户端的请求命令，而是先将 client 放入一个 LIFO 队列 clients_pending_read；
5. 在事件循环（Event Loop）中，主线程执行 beforeSleep ⟶ handleClientsWithPendingReadsUsingThreads，利用 Round-Robin 轮询负载均衡策略，把 clients_pending_read 队列中的连接均匀地分配给 I/O 线程各自的本地 FIFO 任务队列 io_threads_list[id] 和主线程自己，I/O 线程通过 socket 读取客户端的请求命令，存入 client->querybuf 并解析第一个命令，**但不执行命令**，主线程忙轮询，等待所有 I/O 线程完成读取任务；
6. 主线程和所有 I/O 线程都完成了读取任务，主线程结束忙轮询，遍历 clients_pending_read 队列，**执行所有客户端连接的请求命令**，先调用 processCommandAndResetClient 执行第一条已经解析好的命令，然后调用 processInputBuffer 解析并执行客户端连接的所有命令，在其中使用 processInlineBuffer 或者 processMultibulkBuffer 根据 Redis 协议解析命令，最后调用 processCommand 执行命令；
7. 根据请求命令的类型（SET, GET, DEL, EXEC 等），分配相应的命令执行器去执行，最后调用 addReply 函数族的一系列函数将响应数据写入到对应 client 的写出缓冲区：client->buf 或者 client->reply，client->buf 是首选的写出缓冲区，固定大小 16KB，

一般来说可以缓冲足够多的响应数据，但是如果客户端在时间窗口内需要响应的数据非常大，那么则会自动切换到 `client->reply` 链表上去，使用链表理论上能够保存无限大的数据（受限于机器的物理内存），最后把 `client` 添加进一个 **LIFO** 队列 `clients_pending_write`；

8. 在事件循环（**Event Loop**）中，主线程执行 `beforeSleep` ⟶ `handleClientsWithPendingWritesUsingThreads`，利用 **Round-Robin** 轮询负载均衡策略，把 `clients_pending_write` 队列中的连接均匀地分配给 **I/O** 线程各自的本地 **FIFO** 任务队列 `io_threads_list[id]` 和主线程自己，**I/O** 线程通过调用 `writeToClient` 把 `client` 的写出缓冲区里的数据回写到客户端，主线程忙轮询，等待所有 **I/O** 线程完成写出任务；
9. 主线程和所有 **I/O** 线程都完成了写出任务， 主线程结束忙轮询，遍历 `clients_pending_write` 队列，如果 `client` 的写出缓冲区还有数据遗留，则注册 `sendReplyToClient` 到该连接的写就绪事件，等待客户端可写时在事件循环中再继续回写残余的响应数据。

这里大部分逻辑和之前的单线程模型是一致的，变动的地方仅仅是把读取客户端请求命令和回写响应数据的逻辑异步化了，交给 **I/O** 线程去完成，这里需要特别注意的一点是：**I/O 线程仅仅是读取和解析客户端命令而不会真正去执行命令，客户端命令的执行最终还是要回到主线程上完成**。

源码剖析

> 以下所有代码基于目前最新的 Redis v6.0.10 版本。

多线程初始化

```
void initThreadedIO(void) {
    server.io_threads_active = 0; /* We start with threads not active. */

    // 如果用户只配置了一个 I/O 线程，则不会创建新线程（效率低），直接在主线程里处理
    if (server.io_threads_num == 1) return;

    if (server.io_threads_num > IO_THREADS_MAX_NUM) {
        serverLog(LL_WARNING,"Fatal: too many I/O threads configured. "
                             "The maximum number is %d.", IO_THREADS_MAX_NUM);
        exit(1);
```

```
    }

    // 根据用户配置的 I/O 线程数，启动线程。
    for (int i = 0; i < server.io_threads_num; i++) {
        // 初始化 I/O 线程的本地任务队列。
        io_threads_list[i] = listCreate();
        if (i == 0) continue; // 线程 0 是主线程。

        // 初始化 I/O 线程并启动。
        pthread_t tid;
        // 每个 I/O 线程会分配一个本地锁，用来休眠和唤醒线程。
        pthread_mutex_init(&io_threads_mutex[i],NULL);
        // 每个 I/O 线程分配一个原子计数器，用来记录当前遗留的任务数量。
        io_threads_pending[i] = 0;
        // 主线程在启动 I/O 线程的时候会默认先锁住它，直到有 I/O 任务才唤醒它。
        pthread_mutex_lock(&io_threads_mutex[i]);
        // 启动线程，进入 I/O 线程的主逻辑函数 IOThreadMain。
        if (pthread_create(&tid,NULL,IOThreadMain,(void*)(long)i) != 0) {
            serverLog(LL_WARNING,"Fatal: Can't initialize IO thread.");
            exit(1);
        }
        io_threads[i] = tid;
    }
}
```

`initThreadedIO` 会在 Redis 服务器启动时的初始化工作的末尾被调用，初始化 I/O 多线程并启动。

Redis 的多线程模式默认是关闭的，需要用户在 `redis.conf` 配置文件中开启：

```
io-threads 4
io-threads-do-reads yes
```

读取请求

当客户端发送请求命令之后，会触发 Redis 主线程的事件循环，命令处理器 readQueryFromClient 被回调，在以前的单线程模型下，这个方法会直接读取解析客户端命令并执行，但是多线程模式下，则会把 client 加入到 clients_pending_read 任务队列中去，后面主线程再分配到 I/O 线程去读取客户端请求命令：

```
void readQueryFromClient(connection *conn) {
    client *c = connGetPrivateData(conn);
    int nread, readlen;
    size_t qblen;

    // 检查是否开启了多线程，如果是则把 client 加入异步队列之后返回。
    if (postponeClientRead(c)) return;

    // 省略代码，下面的代码逻辑和单线程版本几乎是一样的。
    ...
}

int postponeClientRead(client *c) {
    // 当多线程 I/O 模式开启、主线程没有在处理阻塞任务时，将 client 加入异步队列。
    if (server.io_threads_active &&
        server.io_threads_do_reads &&
        !ProcessingEventsWhileBlocked &&
        !(c->flags & (CLIENT_MASTER|CLIENT_SLAVE|CLIENT_PENDING_READ)))
    {
        // 给 client 打上 CLIENT_PENDING_READ 标识，表示该 client 需要被多线程处理
        // 后续在 I/O 线程中会在读取和解析完客户端命令之后判断该标识并放弃执行命令，
        c->flags |= CLIENT_PENDING_READ;
        listAddNodeHead(server.clients_pending_read,c);
        return1;
    } else {
        return0;
    }
}
```

接着主线程会在事件循环的 beforeSleep() 方法中，调用 handleClientsWithPendingReadsUsingThreads：

```
int handleClientsWithPendingReadsUsingThreads(void) {
    if (!server.io_threads_active || !server.io_threads_do_reads) return0;
    int processed = listLength(server.clients_pending_read);
    if (processed == 0) return0;

    if (tio_debug) printf("%d TOTAL READ pending clients\n", processed);

    // 遍历待读取的 client 队列 clients_pending_read,
    // 通过 RR 轮询均匀地分配给 I/O 线程和主线程自己（编号 0）。
    listIter li;
    listNode *ln;
    listRewind(server.clients_pending_read,&li);
    int item_id = 0;
    while((ln = listNext(&li))) {
        client *c = listNodeValue(ln);
        int target_id = item_id % server.io_threads_num;
        listAddNodeTail(io_threads_list[target_id],c);
        item_id++;
    }

    // 设置当前 I/O 操作为读取操作，给每个 I/O 线程的计数器设置分配的任务数量，
    // 让 I/O 线程可以开始工作：只读取和解析命令，不执行。
    io_threads_op = IO_THREADS_OP_READ;
    for (int j = 1; j < server.io_threads_num; j++) {
        int count = listLength(io_threads_list[j]);
        io_threads_pending[j] = count;
    }

    // 主线程自己也会去执行读取客户端请求命令的任务，以达到最大限度利用 CPU。
    listRewind(io_threads_list[0],&li);
    while((ln = listNext(&li))) {
        client *c = listNodeValue(ln);
        readQueryFromClient(c->conn);
    }
    listEmpty(io_threads_list[0]);

    // 忙轮询，累加所有 I/O 线程的原子任务计数器，直到所有计数器的遗留任务数量都是 0
```

```
    // 表示所有任务都已经执行完成，结束轮询。
    while(1) {
        unsignedlong pending = 0;
        for (int j = 1; j < server.io_threads_num; j++)
            pending += io_threads_pending[j];
        if (pending == 0) break;
    }
    if (tio_debug) printf("I/O READ All threads finshed\n");

    // 遍历待读取的 client 队列，清除 CLIENT_PENDING_READ 和 CLIENT_PENDING_COMM
    // 然后解析并执行所有 client 的命令。
    while(listLength(server.clients_pending_read)) {
        ln = listFirst(server.clients_pending_read);
        client *c = listNodeValue(ln);
        c->flags &= ~CLIENT_PENDING_READ;
        listDelNode(server.clients_pending_read,ln);

        if (c->flags & CLIENT_PENDING_COMMAND) {
            c->flags &= ~CLIENT_PENDING_COMMAND;
            // client 的第一条命令已经被解析好了，直接尝试执行。
            if (processCommandAndResetClient(c) == C_ERR) {
                /* If the client is no longer valid, we avoid
                 * processing the client later. So we just go
                 * to the next. */
                continue;
            }
        }
        processInputBuffer(c); // 继续解析并执行 client 命令。

        // 命令执行完成之后，如果 client 中有响应数据需要回写到客户端，则将 client
        if (!(c->flags & CLIENT_PENDING_WRITE) && clientHasPendingReplies(c))
            clientInstallWriteHandler(c);
    }

    /* Update processed count on server */
    server.stat_io_reads_processed += processed;

    return processed;
}
```

这里的核心工作是：

- 遍历待读取的 `client` 队列 `clients_pending_read`，通过 RR 策略把所有任务分配给 I/O 线程和主线程去读取和解析客户端命令。
- 忙轮询等待所有 I/O 线程完成任务。
- 最后再遍历 `clients_pending_read`，执行所有 `client` 的命令。

写回响应

完成命令的读取、解析以及执行之后，客户端命令的响应数据已经存入 `client->buf` 或者 `client->reply` 中了，接下来就需要把响应数据回写到客户端了，还是在 `beforeSleep` 中， 主线程调用 `handleClientsWithPendingWritesUsingThreads`：

```
int handleClientsWithPendingWritesUsingThreads(void) {
    int processed = listLength(server.clients_pending_write);
    if (processed == 0) return0; /* Return ASAP if there are no clients. */

    // 如果用户设置的 I/O 线程数等于 1 或者当前 clients_pending_write 队列中待写出
    // 数量不足 I/O 线程数的两倍，则不用多线程的逻辑，让所有 I/O 线程进入休眠，
    // 直接在主线程把所有 client 的相应数据回写到客户端。
    if (server.io_threads_num == 1 || stopThreadedIOIfNeeded()) {
        return handleClientsWithPendingWrites();
    }

    // 唤醒正在休眠的 I/O 线程（如果有的话）。
    if (!server.io_threads_active) startThreadedIO();

    if (tio_debug) printf("%d TOTAL WRITE pending clients\n", processed);

    // 遍历待写出的 client 队列 clients_pending_write，
    // 通过 RR 轮询均匀地分配给 I/O 线程和主线程自己（编号 0）。
    listIter li;
    listNode *ln;
    listRewind(server.clients_pending_write,&li);
```

```
int item_id = 0;
while((ln = listNext(&li))) {
    client *c = listNodeValue(ln);
    c->flags &= ~CLIENT_PENDING_WRITE;

    /* Remove clients from the list of pending writes since
     * they are going to be closed ASAP. */
    if (c->flags & CLIENT_CLOSE_ASAP) {
        listDelNode(server.clients_pending_write, ln);
        continue;
    }

    int target_id = item_id % server.io_threads_num;
    listAddNodeTail(io_threads_list[target_id],c);
    item_id++;
}

// 设置当前 I/O 操作为写出操作，给每个 I/O 线程的计数器设置分配的任务数量，
// 让 I/O 线程可以开始工作，把写出缓冲区（client->buf 或 c->reply）中的响应数据
io_threads_op = IO_THREADS_OP_WRITE;
for (int j = 1; j < server.io_threads_num; j++) {
    int count = listLength(io_threads_list[j]);
    io_threads_pending[j] = count;
}

// 主线程自己也会去执行读取客户端请求命令的任务，以达到最大限度利用 CPU。
listRewind(io_threads_list[0],&li);
while((ln = listNext(&li))) {
    client *c = listNodeValue(ln);
    writeToClient(c,0);
}
listEmpty(io_threads_list[0]);

// 忙轮询，累加所有 I/O 线程的原子任务计数器，直到所有计数器的遗留任务数量都是 0，
// 表示所有任务都已经执行完成，结束轮询。
while(1) {
    unsignedlong pending = 0;
    for (int j = 1; j < server.io_threads_num; j++)
        pending += io_threads_pending[j];
```

```
            if (pending == 0) break;
        }
        if (tio_debug) printf("I/O WRITE All threads finshed\n");

        // 最后再遍历一次 clients_pending_write 队列，检查是否还有 client 的中写出缓冲
        // 如果有，那就为 client 注册一个命令回复器 sendReplyToClient，等待客户端写就绪
        listRewind(server.clients_pending_write,&li);
        while((ln = listNext(&li))) {
            client *c = listNodeValue(ln);

            // 检查 client 的写出缓冲区是否还有遗留数据。
            if (clientHasPendingReplies(c) &&
                    connSetWriteHandler(c->conn, sendReplyToClient) == AE_ERR)
            {
                freeClientAsync(c);
            }
        }
        listEmpty(server.clients_pending_write);

        /* Update processed count on server */
        server.stat_io_writes_processed += processed;

        return processed;
    }
```

这里的核心工作是：

- 检查当前任务负载，如果当前的任务数量不足以用多线程模式处理的话，则休眠 I/O 线程并且直接同步将响应数据回写到客户端。
- 唤醒正在休眠的 I/O 线程（如果有的话）。
- 遍历待写出的 `client` 队列 `clients_pending_write`，通过 RR 策略把所有任务分配给 I/O 线程和主线程去将响应数据写回到客户端。
- 忙轮询等待所有 I/O 线程完成任务。
- 最后再遍历 `clients_pending_write`，为那些还残留有响应数据的 `client` 注册命令回复处理器 `sendReplyToClient`，等待客户端可写之后在事件循环中继续回写残余的响应数据。

I/O 线程主逻辑

```
void *IOThreadMain(void *myid) {
    /* The ID is the thread number (from 0 to server.iothreads_num-1), and is
     * used by the thread to just manipulate a single sub-array of clients. */
    long id = (unsignedlong)myid;
    char thdname[16];

    snprintf(thdname, sizeof(thdname), "io_thd_%ld", id);
    redis_set_thread_title(thdname);
    // 设置 I/O 线程的 CPU 亲和性，尽可能将 I/O 线程（以及主线程，不在这里设置）绑定
    // CPU 列表上。
    redisSetCpuAffinity(server.server_cpulist);
    makeThreadKillable();

    while(1) {
        // 忙轮询，100w 次循环，等待主线程分配 I/O 任务。
        for (int j = 0; j < 1000000; j++) {
            if (io_threads_pending[id] != 0) break;
        }

        // 如果 100w 次忙轮询之后如果还是没有任务分配给它，则通过尝试加锁进入休眠，
        // 等待主线程分配任务之后调用 startThreadedIO 解锁，唤醒 I/O 线程去执行。
        if (io_threads_pending[id] == 0) {
            pthread_mutex_lock(&io_threads_mutex[id]);
            pthread_mutex_unlock(&io_threads_mutex[id]);
            continue;
        }

        serverAssert(io_threads_pending[id] != 0);

        if (tio_debug) printf("[%ld] %d to handle\n", id, (int)listLength(io_t


        // 注意：主线程分配任务给 I/O 线程之时，
        // 会把任务加入每个线程的本地任务队列 io_threads_list[id]，
        // 但是当 I/O 线程开始执行任务之后，主线程就不会再去访问这些任务队列，避免数
        listIter li;
```

```
            listNode *ln;
            listRewind(io_threads_list[id],&li);
            while((ln = listNext(&li))) {
                client *c = listNodeValue(ln);
                // 如果当前是写出操作，则把 client 的写出缓冲区中的数据回写到客户端。
                if (io_threads_op == IO_THREADS_OP_WRITE) {
                    writeToClient(c,0);
                  // 如果当前是读取操作，则socket 读取客户端的请求命令并解析第一条命令。
                } elseif (io_threads_op == IO_THREADS_OP_READ) {
                    readQueryFromClient(c->conn);
                } else {
                    serverPanic("io_threads_op value is unknown");
                }
            }
            listEmpty(io_threads_list[id]);
            // 所有任务执行完之后把自己的计数器置 0，主线程通过累加所有 I/O 线程的计数器
            // 判断是否所有 I/O 线程都已经完成工作。
            io_threads_pending[id] = 0;

            if (tio_debug) printf("[%ld] Done\n", id);
        }
    }
```

I/O 线程启动之后，会先进入忙轮询，判断原子计数器中的任务数量，如果是非 0 则表示主线程已经给它分配了任务，开始执行任务，否则就一直忙轮询一百万次等待，忙轮询结束之后再查看计数器，如果还是 0，则尝试加本地锁，因为主线程在启动 I/O 线程之时就已经提前锁住了所有 I/O 线程的本地锁，因此 I/O 线程会进行休眠，等待主线程唤醒。

主线程会在每次事件循环中尝试调用 `startThreadedIO` 唤醒 I/O 线程去执行任务，如果接收到客户端请求命令，则 I/O 线程会被唤醒开始工作，根据主线程设置的 `io_threads_op` 标识去执行命令读取和解析或者回写响应数据的任务，I/O 线程在收到主线程通知之后，会遍历自己的本地任务队列 `io_threads_list[id]`，取出一个个 `client` 执行任务：

- 如果当前是写出操作，则调用 `writeToClient`，通过 **socket** 把 `client->buf` 或者 `client->reply` 里的响应数据回写到客户端。

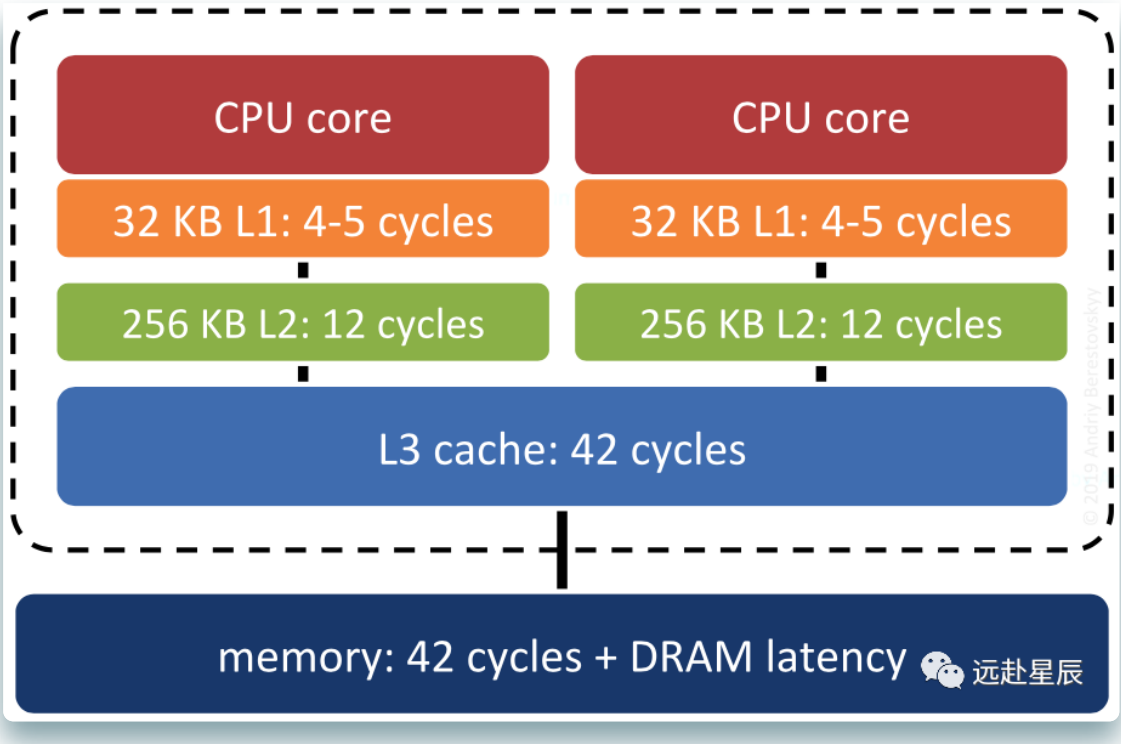
- 如果当前是读取操作，则调用 `readQueryFromClient`，通过 `socket` 读取客户端命令，存入 `client->querybuf`，然后调用 `processInputBuffer` 去解析命令，这里最终只会解析到第一条命令，然后就结束，不会去执行命令。
- 在全部任务执行完之后把自己的原子计数器置 `0`，以告知主线程自己已经完成了工作。

```
void processInputBuffer(client *c) {
// 省略代码
...

    while(c->qb_pos < sdslen(c->querybuf)) {
        /* Return if clients are paused. */
        if (!(c->flags & CLIENT_SLAVE) && clientsArePaused()) break;

        /* Immediately abort if the client is in the middle of something. */
        if (c->flags & CLIENT_BLOCKED) break;

        /* Don't process more buffers from clients that have already pending
         * commands to execute in c->argv. */
        if (c->flags & CLIENT_PENDING_COMMAND) break;
        /* Multibulk processing could see a <= 0 length. */
        if (c->argc == 0) {
            resetClient(c);
        } else {
            // 判断 client 是否具有 CLIENT_PENDING_READ 标识，如果是处于多线程 I/
            // 那么此前已经在 readQueryFromClient -> postponeClientRead 中为 cli
            // 则立刻跳出循环结束，此时第一条命令已经解析完成，但是不执行命令。
            if (c->flags & CLIENT_PENDING_READ) {
                c->flags |= CLIENT_PENDING_COMMAND;
                break;
            }

            // 执行客户端命令
            if (processCommandAndResetClient(c) == C_ERR) {
                /* If the client is no longer valid, we avoid exiting this
                 * loop and trimming the client buffer later. So we return
                 * ASAP in that case. */
                return;
            }
```

```
        }
    }

...
}
```

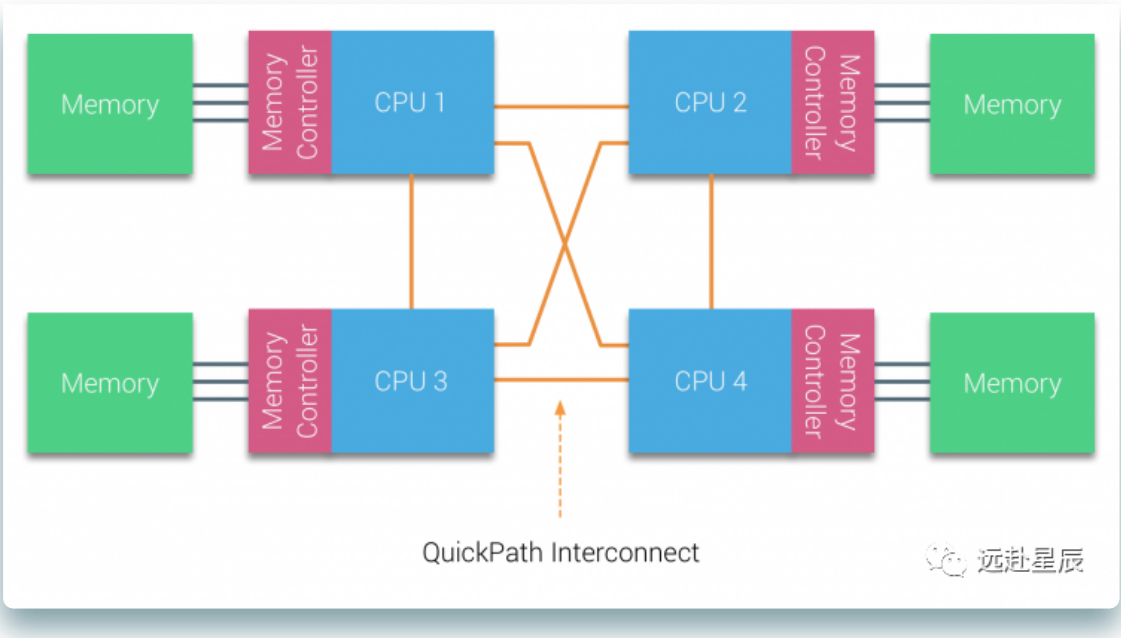
这里需要额外关注 I/O 线程初次启动时会设置当前线程的 CPU 亲和性，也就是绑定当前线程到用户配置的 CPU 上，在启动 Redis 服务器主线程的时候同样会设置 CPU 亲和性，Redis 的核心网络模型引入多线程之后，加上之前的多线程异步任务、多进程（BGSAVE、AOF、BIO、Sentinel 脚本任务等），Redis 现如今的系统并发度已经很大了，而 Redis 本身又是一个对吞吐量和延迟极度敏感的系统，所以用户需要 Redis 对 CPU 资源有更细粒度的控制，这里主要考虑的是两方面：CPU 高速缓存和 NUMA 架构。

首先是 CPU 高速缓存（这里讨论的是 L1 Cache 和 L2 Cache 都集成在 CPU 中的硬件架构），这里想象一种场景：Redis 主进程正在 CPU-1 上运行，给客户端提供数据服务，此时 Redis 启动了子进程进行数据持久化（BGSAVE 或者 AOF），系统调度之后子进程抢占了主进程的 CPU-1，主进程被调度到 CPU-2 上去运行，导致之前 CPU-1 的高速缓存里的相关指令和数据被汰换掉，CPU-2 需要重新加载指令和数据到自己的本地高速缓存里，浪费 CPU 资源，降低性能。



因此，Redis 通过设置 CPU 亲和性，可以将主进程/线程和子进程/线程绑定到不同的核隔离开来，使之互不干扰，能有效地提升系统性能。

其次是基于 NUMA 架构的考虑，在 NUMA 体系下，内存控制器芯片被集成到处理器内部，形成 CPU 本地内存，访问本地内存只需通过内存通道而无需经过系统总线，访问时延大大降低，而多个处理器之间通过 QPI 数据链路互联，跨 NUMA 节点的内存访问开销远大于本地内存的访问：



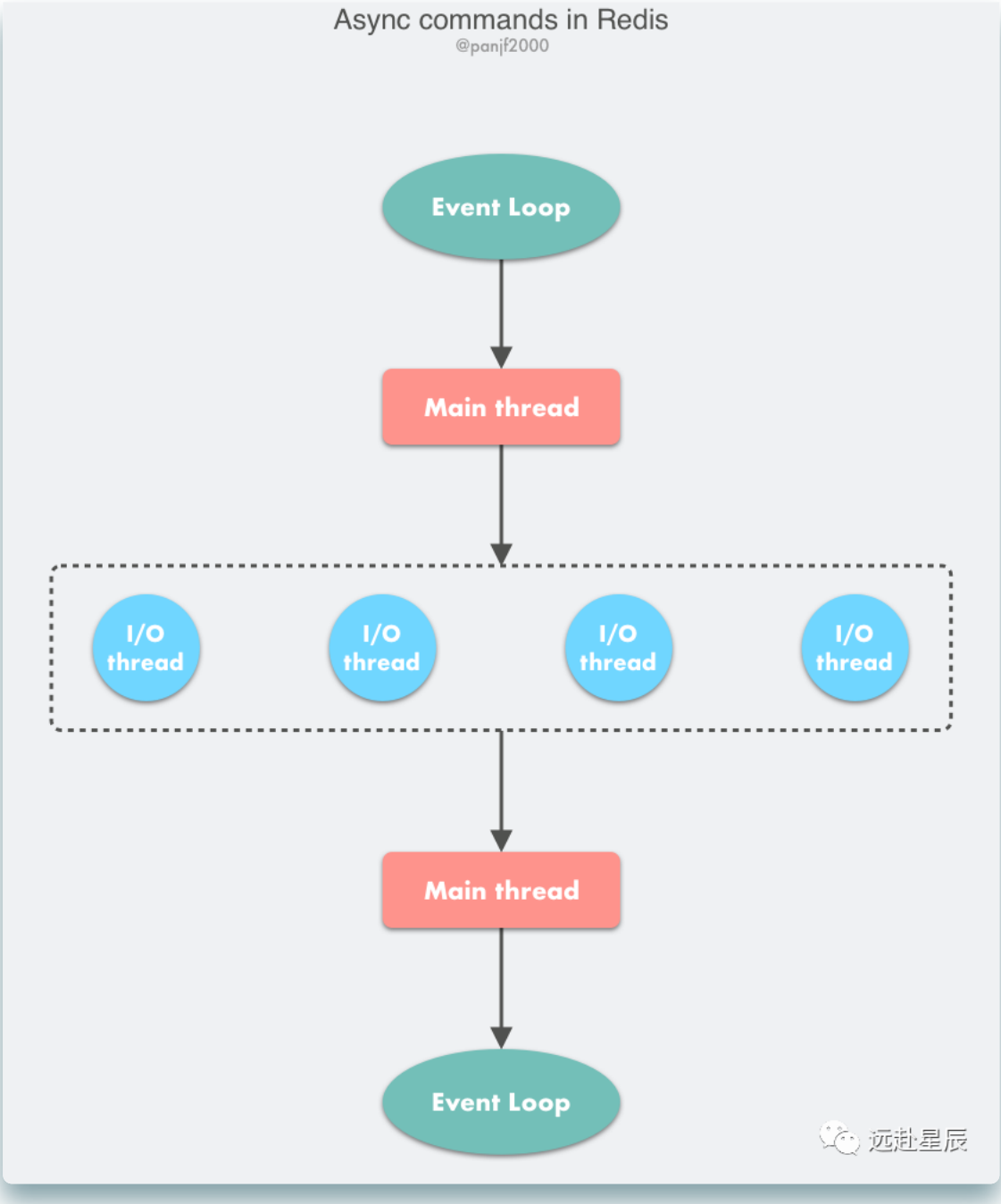
因此，Redis 通过设置 CPU 亲和性，让主进程/线程尽可能在固定的 NUMA 节点上的 CPU 上运行，更多地使用本地内存而不需要跨节点访问数据，同样也能大大地提升性能。

关于 NUMA 相关知识请读者自行查阅，篇幅所限这里就不再展开，以后有时间我再单独写一篇文章介绍。

最后还有一点，阅读过源码的读者可能会有疑问，Redis 的多线程模式下，似乎并没有对数据进行锁保护，事实上 Redis 的多线程模型是全程无锁（Lock-free）的，这是通过原子操作+交错访问来实现的，主线程和 I/O 线程之间共享的变量有三个：`io_threads_pending` 计数器、`io_threads_op` I/O 标识符和 `io_threads_list` 线程本地任务队列。

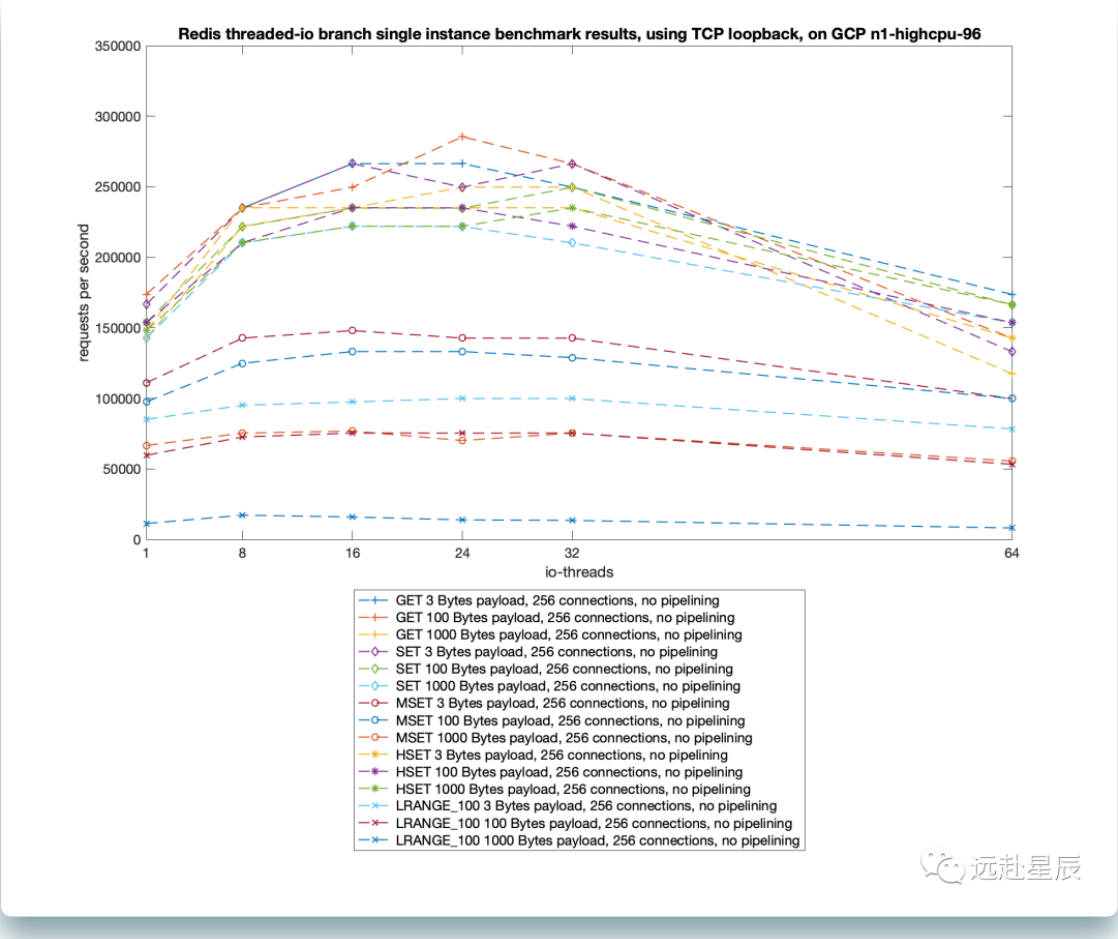
`io_threads_pending` 是原子变量，不需要加锁保护，`io_threads_op` 和 `io_threads_list` 这两个变量则是通过控制主线程和 I/O 线程交错访问来规避共享数据竞争问题：I/O 线程启动之后会通过忙轮询和锁休眠等待主

线程的信号，在这之前它不会去访问自己的本地任务队列 `io_threads_list[id]`，而主线程会在分配完所有任务到各个 I/O 线程的本地队列之后才去唤醒 I/O 线程开始工作，并且主线程之后在 I/O 线程运行期间只会访问自己的本地任务队列 `io_threads_list[0]` 而不会再去访问 I/O 线程的本地队列，这也就保证了主线程永远会在 I/O 线程之前访问 `io_threads_list` 并且之后不再访问，保证了交错访问。`io_threads_op` 同理，主线程会在唤醒 I/O 线程之前先设置好 `io_threads_op` 的值，并且在 I/O 线程运行期间不会再去访问这个变量。



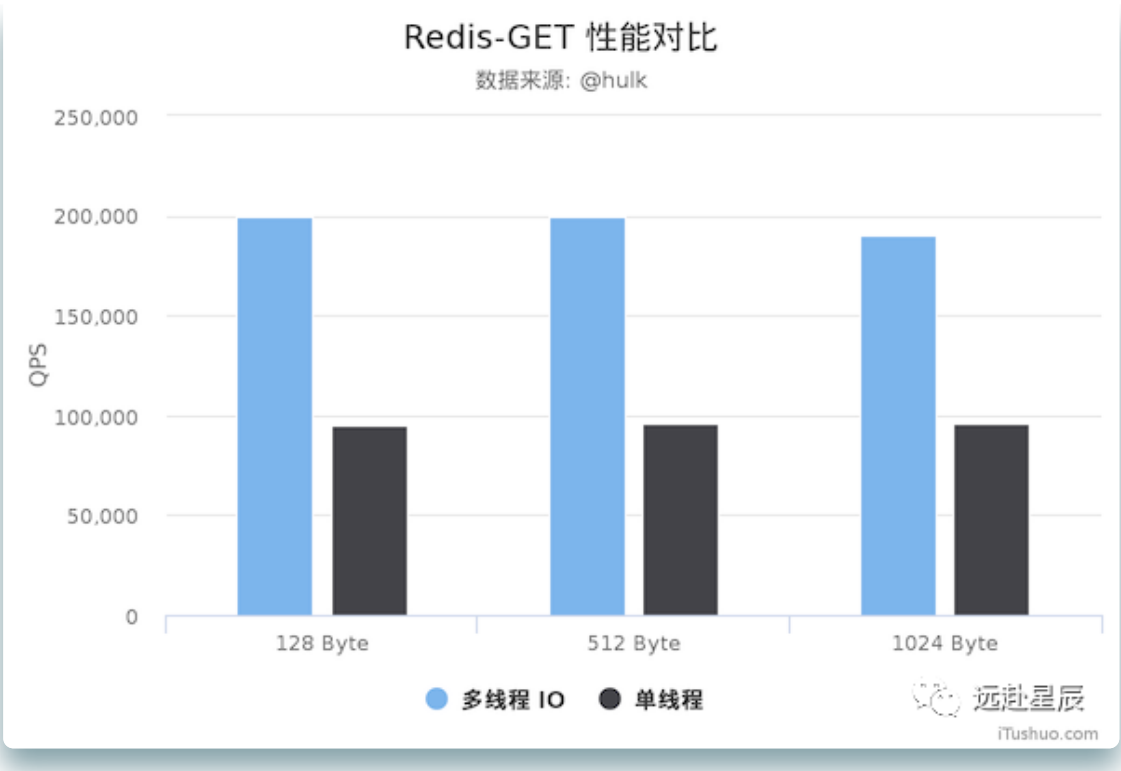
性能提升

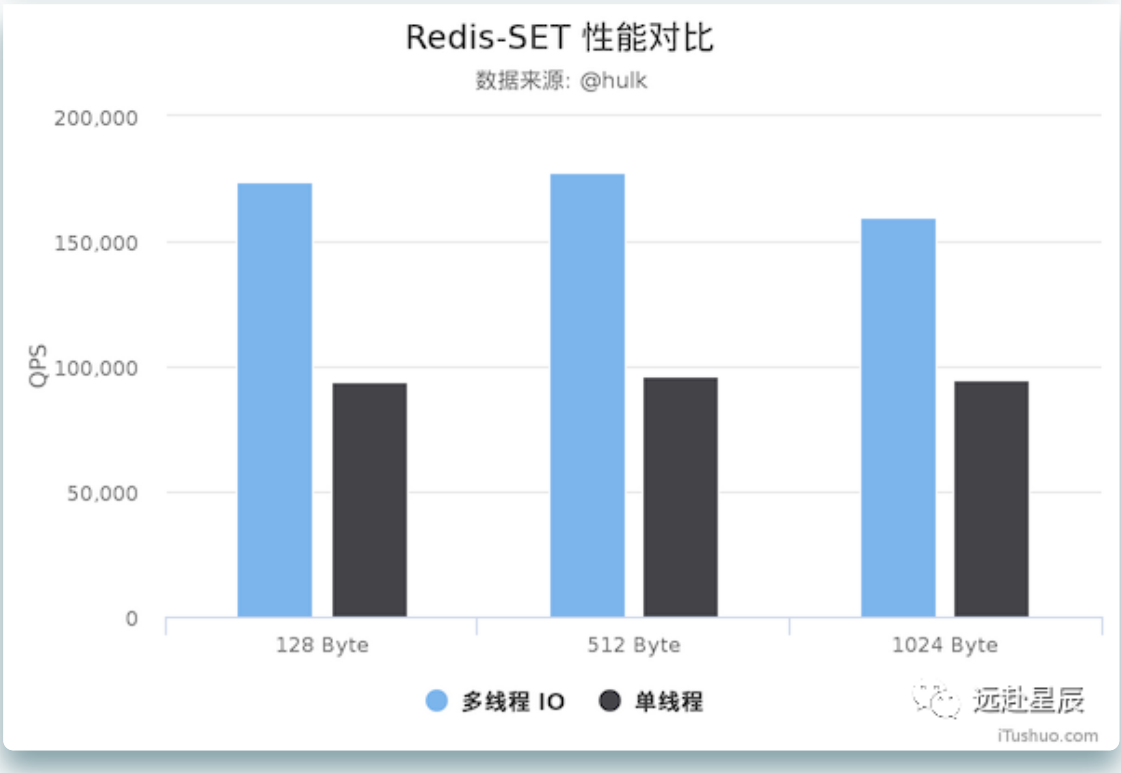
Redis 将核心网络模型改造成多线程模式追求的当然是最终性能上的提升，所以最终还是要以 benchmark 数据见真章：



测试数据表明，Redis 在使用多线程模式之后性能大幅提升，达到了一倍。更详细的性能压测数据可以参阅这篇文章：Benchmarking the experimental Redis Multi-Threaded I/O。

以下是美图技术团队实测的新旧 Redis 版本性能对比图，仅供参考：





模型缺陷

首先第一个就是我前面提到过的，Redis 的多线程网络模型实际上并不是一个标准的 Multi-Reactors/Master-Workers 模型，和其他主流的开源网络服务器的模式有所区别，最大的不同就是在标准的 Multi-Reactors/Master-Workers 模式下，Sub Reactors/Workers 会

完成 网络读 -> 数据解析 -> 命令执行 -> 网络写 整套流程，Main Reactor/Master 只负责分派任务，而在 Redis 的多线程方案中，I/O 线程任务仅仅是通过 socket 读取客户端请求命令并解析，却没有真正去执行命令，所有客户端命令最后还需要回到主线程去执行，因此对多核的利用率并不算高，而且每次主线程都必须在分配完任务之后忙轮询等待所有 I/O 线程完成任务之后才能继续执行其他逻辑。

Redis 之所以如此设计它的多线程网络模型，我认为主要的原因是为了保持兼容性，因为以前 Redis 是单线程的，所有的客户端命令都是在单线程的事件循环里执行的，也因此 Redis 里所有的数据结构都是非线程安全的，现在引入多线程，如果按照标准的 Multi-Reactors/Master-Workers 模式来实现，则所有内置的数据结构都必须重构成线程安全的，这个工作量无疑是巨大且麻烦的。

所以，在我看来，Redis 目前的多线程方案更像是一个折中的选择：既保持了原系统的兼容性，又能利用多核提升 I/O 性能。

其次，目前 Redis 的多线程模型中，主线程和 I/O 线程的通信过于简单粗暴：忙轮询和锁，因为通过自旋忙轮询进行等待，导致 Redis 在启动的时候以及运行期间偶尔会有短暂的 CPU 空转引起的高占用率，而且这个通信机制的最终实现看起来非常不直观和不简洁，希望后面 Redis 能对目前的方案加以改进。

总结

Redis 作为缓存系统的事实标准，它的底层原理值得开发者去深入学习，Redis 自 2009 年发布第一版之后，其单线程网络模型的选择在社区中从未停止过讨论，多年来一直有呼声希望 Redis 能引入多线程从而利用多核优势，但是作者 antirez 是一个追求大道至简的开发者，对 Redis 加入任何新功能都异常谨慎，所以在 Redis 初版发布的十年后才最终将 Redis 的核心网络模型改造成多线程模式，这期间甚至诞生了一些 Redis 多线程的替代项目。虽然 antirez 一直在推迟多线程的方案，但却从未停止思考多线程的可行性，Redis 多线程网络模型的改造不是一朝一夕的事情，这其中牵扯到项目的方方面面，所以我们可以看到 Redis 的最终方案也并不完美，没有采用主流的多线程模式设计。

让我们来回顾一下 Redis 多线程网络模型的设计方案：

- 使用 I/O 线程实现网络 I/O 多线程化，I/O 线程只负责网络 I/O 和命令解析，不执行客户端命令。
- 利用原子操作+交错访问实现无锁的多线程模型。
- 通过设置 CPU 亲和性，隔离主进程和其他子进程，让多线程网络模型能发挥最大的性能。

通读本文之后，相信读者们应该能够了解到一个优秀的网络系统的实现所涉及到的计算机领域的各种技术：设计模式、网络 I/O、并发编程、操作系统底层，甚至是计算机硬件。另外还需要对项目迭代和重构的谨慎，对技术方案的深入思考，绝不仅仅是写好代码这一个难点。

参考&延伸阅读

- Redis v5.0.10
- Redis v6.0.10
- Lazy Redis is better Redis
- An update about Redis developments in 2019
- How fast is Redis?
- Go netpoller 原生网络模型之源码全面揭秘
- Linux I/O 原理和 Zero-copy 技术全面揭秘
- Benchmarking the experimental Redis Multi-Threaded I/O
- NUMA DEEP DIVE PART 1: FROM UMA TO NUMA

References

[1] Lazy Redis is better Redis: *http://antirez.com/news/93*
[2] Linux I/O 原理和 Zero-copy 技术全面揭秘: *https://strikefreedom.top/linux-io-and-zero-copy*
[3] Go netpoller 原生网络模型之源码全面揭秘: *https://strikefreedom.top/go-netpoll-io-multiplexing-reactor*
[4] Redis v6.0.10: *https://github.com/redis/redis/tree/6.0.10*
[5] Benchmarking the experimental Redis Multi-Threaded I/O: *https://itnext.io/benchmarking-the-experimental-redis-multi-threaded-i-o-1bb28b69a314*
[6] Redis v5.0.10: *https://github.com/redis/redis/tree/5.0.10*
[7] Redis v6.0.10: *https://github.com/redis/redis/tree/6.0.10*
[8] Lazy Redis is better Redis: *http://antirez.com/news/93*
[9] An update about Redis developments in 2019: *http://antirez.com/news/126*
[10] How fast is Redis?: *https://redis.io/topics/benchmarks*
[11] Go netpoller 原生网络模型之源码全面揭秘: *https://strikefreedom.top/go-netpoll-io-*