浙江理工大学学报，2020，43(4)：520-526
Journal of Zhejiang Sci-Tech University
DOI:10.3969/j.issn.1673－3851(n).2020.04.013

基于 Reactor 与非阻塞 IO 的服务端框架设计与实现

包晓安[1]，聂凡杰[1]，徐璐[1]，张娜[1]，吴彪[2]
(1.浙江理工大学信息学院，杭州 310018；2.山口大学东亚研究科，日本山口 753-8514)

摘　要：吞吐量对服务端框架的处理效率有着重要的影响，为了进一步提升传统服务端框架的吞吐量，提出了一种基于 Reactor 模式与非阻塞 IO 的服务端框架。首先，对 Reactor 模式与非阻塞 IO 进行了优势分析，并阐述了 Reactor 线程池的分发逻辑；其次，通过设计自适应缓冲区结构降低了内存分配次数，提升了数据读入和写出的效率；最后，通过设计双缓冲结构优化了日志的写入操作，提升了日志写入效率。实验结果显示：在单线程测试环境下，对比 libevent，该服务端框架吞吐量平均提升了 9%；在多线程测试环境下，分别在 100 连接与 1000 连接时，对比 Boost.Asio，该服务端框架吞吐量分别平均提升了 28.66%与 20.76%。这表明该服务端框架吞吐量较高，可应用于较大数据量请求的场景。

关键词：Reactor；服务端；吞吐量；IO；线程池
中图分类号：TP311.5　　**文献标志码**：A　　**文章编号**：1673-3851 (2020) 07-0520-07

Design and implementation of server framework based on Reactor and non-blocking IO

BAO Xiaoan[1], *NIE Fanjie*[1], *XU Lu*[1], *ZHANG Na*[1], *WU Biao*[2]
(1. School of Information Science and Technology, Zhejiang Sci-Tech University, Hangzhou 310018, China;
2.Department of East Asian Studies, Yamaguchi University, Yamaguchi 753-8514, Japan)

Abstract: Throughput has an important impact on the processing efficiency of server-side framework. In order to further improve the throughput of traditional server-side frameworks, a server-side framework based on the Reactor pattern and non-blocking IO is proposed. Firstly, an analysis was made on the advantages of the Reactor pattern and non-blocking IO, and an exposition was made on the distribution logic of the Reactor thread pool. Secondly, an adaptive buffer structure was designed for purpose of reducing the number of memory allocations and improving the efficiency of data reading and writing. Last, the log write operation was optimized by designing a double-buffer structure, which improved the log write efficiency. The experimental results show that in the case of a single-threaded test environment, the throughput of the server-side framework is increased by 9% on average compared to libevent; in the case of a multi-threaded test environment, the throughput of the server-side framework, at 100 and 1000 connections respectively, is increased by 28.66% and 20.76% on average compared with Boost.Asio. This shows that the server-side framework has high throughput and can be applied to scenarios with large data volume requests.

Key words: Reactor; server side; throughput; IO; thread pool

收稿日期：2019－10－31　　**网络出版日期**：2020－03－05
基金项目：浙江省重点研发计划项目(2020C03094)；浙江省自然科学基金青年基金项目(LQ20F050010)；浙江省公益技术研究计划项目(GG20F010028)
作者简介：包晓安(1973－　)，男，浙江东阳人，教授，硕士，主要从事图像处理、机器学习等方面的研究。

0 引　言

吞吐量是单位时间内成功传送数据的数量[1]，是衡量服务端框架处理效率的重要参考因素，而服务端框架的 IO 处理能力对吞吐量又有着重要的影响。在 IO 处理方面，服务端程序要尽可能使数据在本端处理时间短，以便及时空出 CPU 资源处理后续请求[2-3]，避免请求在队列上堆积，造成服务端程序吞吐量急剧降低[4-5]，甚至宕机，这可能给企业造成无法估量的损失。

目前使用较多的服务端框架有 Boost.Asio、libevent、libev 等，这些服务端框架通过封装多路复用模型、日志、IPC 等给服务端程序开发带来了极大的便利[6-8]，但其各自的吞吐量仍有一定的提升空间。Gul 等[9]采用基于事件的响应机制，将每一个请求都视为一个事件，然后使用轮询的方式去处理事件，从而提升了并发响应能力，提升了响应速度。顾振德等[10]提出了事件驱动机制搭配线程池的框架，事件驱动模块负责监听请求，线程池负责耗时操作，从而避免了 IO 线程陷入耗时操作内，提升了服务端程序的响应速度。叶柏龙等[11]提出了基于 Proactor 与主线程池的方式，实现了一套 CPU 利用率低、吞吐量高的服务端框架。但是，以上服务端框架在 Reactor 反应堆配置、输入输出缓冲区结构以及日志处理流程上仍有一定的优化空间。

针对以上问题，本文提出了一种基于 Reactor 模式与非阻塞 IO 的服务端框架，该框架通过 Reactor 线程池来应对突发型 IO；通过自适应的缓冲区来应对无法一次性处理收发消息的情况，避免了无意义地回调应用层函数而导致的忙循环，并从结构上避免了内存的重复分配；通过设计双缓冲日志，优化了日志写入操作；同时结合了线程池来应对耗时操作，以便及时空出 IO 线程去处理后续请求，从而提升整体框架的吞吐量。

1 Reactor 模式和非阻塞 IO

1.1 Reactor 模式

Reactor 模式是 Doug Schmidt 提出一种事件驱动的设计模式，其可以同时处理多个输入，并通过多路复用机制将输入请求分发给相应处理器做进一步处理。单线程 Reactor 模式的流程示意图如图 1 所示。Reactor 反应堆中的 EPOLL（或 SELECT、POLL）是 IO 多路复用模型[12]；应用层在读写事件处理器中绑定相应的回调函数，当事件在 Reactor 反应堆中被触发时进行回调操作；连接建立事件处理器负责绑定连接建立事件，当客户端发起请求时，Reactor 反应堆会将事件分发给连接建立事件处理器，然后再次向 Reactor 反应堆中注册回调事件[13]，即与图 1 中读写事件处理器进行绑定，当事件被触发时，进行回调操作，比如常见的读、写、编解码以及计算操作等。

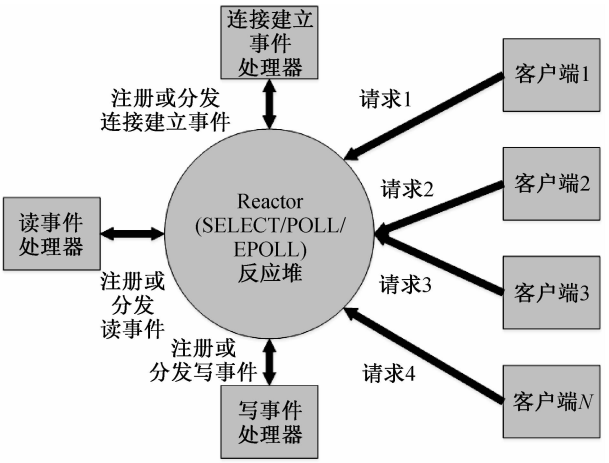

图 1　Reactor 模式流程示意图

1.2 非阻塞 IO

多路复用机制常搭配非阻塞 IO。服务端程序应当尽量避免耗时操作发生在 IO 线程，比如读写操作、与数据库或磁盘的交互等，所以当服务端程序执行耗时操作时，SOCKET 文件描述符的属性需提前被设置为 O_NONBLOCK，以便让线程在读写数据时能立刻返回，不会因所读数据不完整、对方缓冲区已满等情况而发生阻塞，进而影响程序对后续请求的处理。本文通过引入 Reactor 线程池与非阻塞 IO，使得服务端程序可以通过 Round robin 的方式将各个连接的文件描述符绑定到固定的 Reactor 反应堆中，这样某个连接的请求急剧增加也不会对服务端框架的吞吐量造成较大影响。

2 系统框架设计

本文服务端总体框架主要分为四个模块：Reactor 线程池、自适应缓冲区、双缓冲日志以及任务线程池。多个请求状态下的时序图如图 2 所示。其中 Epoll 为多路复用机制的一种；Connection 为发出请求的连接；Accept、Read 和 Write 为网络编程系统函数；Encode、Decode 和 Compute 为服务端开发过程中常见的编码、解码以及计算流程；Wait 表示线程池空闲时长。当连接 connection1 成为活动连接时，主反应堆 Base Reactor 监听到该事件后，将 Accept 函数返回的文件描述通过 Round robin

的方式分发给反应堆进行当读、写、编解码等操作，其中数据的收发都先经过自适应缓冲区的处理，待数据接收完整时再递交应用层做进一步处理。应用层处理过程中，如果其中某个操作较为耗时，为了不阻塞 IO 线程，则将耗时操作递交给线程池做进一步处理，以尽快空出 IO 线程，使线程得到最大程度的复用，从而提升服务端框架整体的吞吐量。同时子反应堆的数量以及线程池中线程的数量在服务端程序启动时就已经确定，当遇到突发请求时处理能力不会随着请求数量的急剧增多而陡然降低，也不会因为 IO 线程由于长时间被耗时操作所占据从而影吞吐量甚至失去响应[14]。

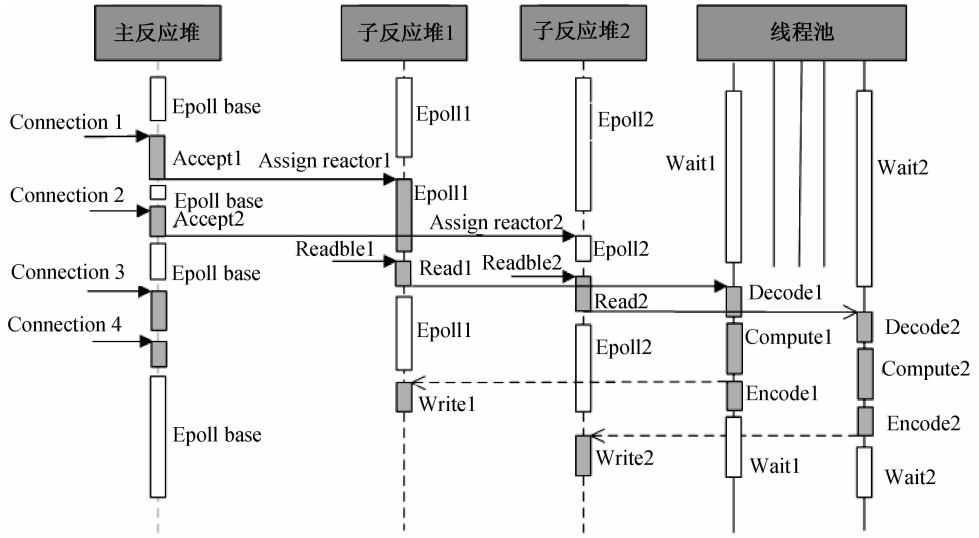

图 2 总体框架时序图

2.1 核心接口设计

框架核心接口的类图如图 3 所示，主要有 ReacotrLoop、ReactorAcceptor、ReactorChannel、TcpConnection 和 ThreadPool 等。

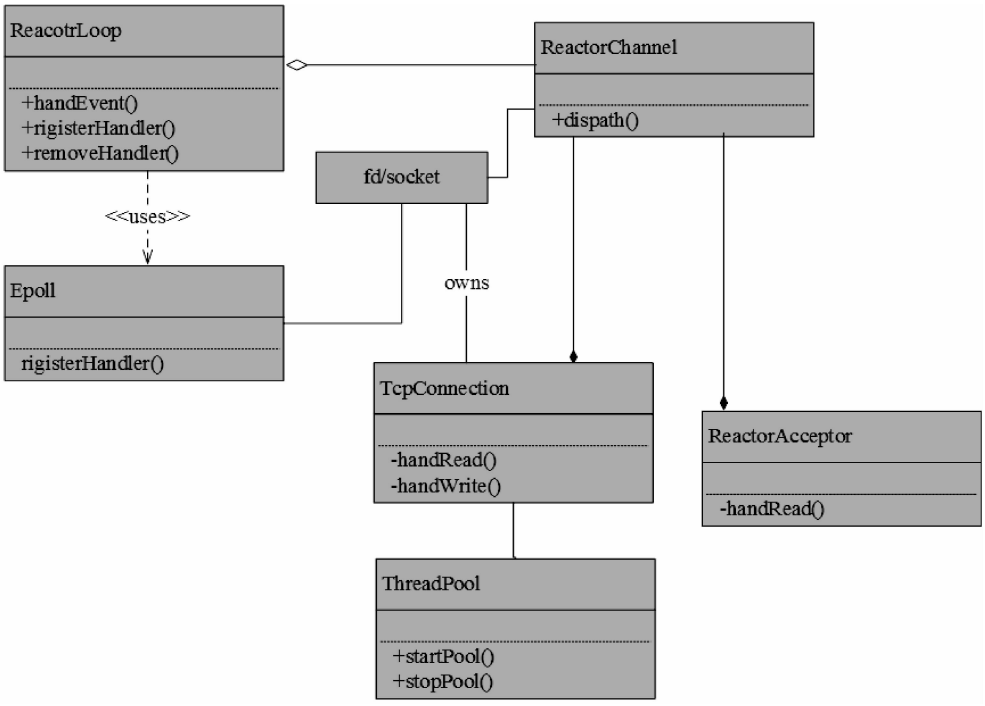

图 3 框架核心接口类图

a) ReacotrLoop 类。该类适用于多路复用机制，主要功能有：向 epoll 中注册服务端程序所关心的事件，如连接建立事件、可读可写事件等；通过多路复用机制，在内核对这些事件进行监听，当所监听的事件变为活动时，将该事件与子反应堆进行绑定，即派发给相应的 ReactorChannel 进行处理。

b) ReactorAcceptor 类。该类主要功能是处理用户发起的连接建立事件，然后将 Accept 函数返回的文件描述符交给子反应堆，并在其中注册所关注的事件，绑定相应的回调函数。

c) ReactorChannel 类。该类属于某个 IO 线程，每个 ReactorChannel 中包含了所监听的文件描述符，主要功能是负责对其所包含的文件描述符上所监听的事件进行分发。比如之前在多路复用机制中，监听的各类事件都是通过 ReactorChannel 类来决定选取哪一个函数来进行回调。

d) TcpConnection 类。该类是在文件描述符分发给子反应堆之后，为 Accept 函数返回的文件描述符绑定回调函数，如可读、可写等回调函数都是在该类中进行绑定。

e) ThreadPool 类。该类负责对耗时操作的处理，当读、写、编解码以及计算中涉及了较为耗时的操作时，将会把耗时任务递交给 ThreadPool 类中的线程进行处理，同时向 ThreadPool 类中传递回调函数，待任务完成时进行回调。

2.2　Reactor 线程池设计

Reactor 池设计图如图 4 所示。为了应对 IO 压力负载较大的场景，使用一个主反应堆负责处理客户端发起的连接建立请求，然后通过 Round robin[19] 的方式将该连接交付给 Reactor 线程池中的某一个子反应堆。在后续处理过程中，当该连接所绑定的文件描述符有事件发生时，都经由固定的子反应堆去处理。子反应堆数量在服务端程序运行时会根据所处环境的 CPU 核心数来确定，避免了突发型 IO 发生时吞吐量陡降的情况。同时当客户端连续多个请求发送到服务端时，由于采用了固定的子反应堆，使得收到回应的后客户端也无需对响应包进行排序处理[20]。此外多个客户端发起的连接被分发到了多个子反应堆中，即被分发到了多个线程中，充分利用了多 CPU 性能，提升了整体框架的吞吐量。

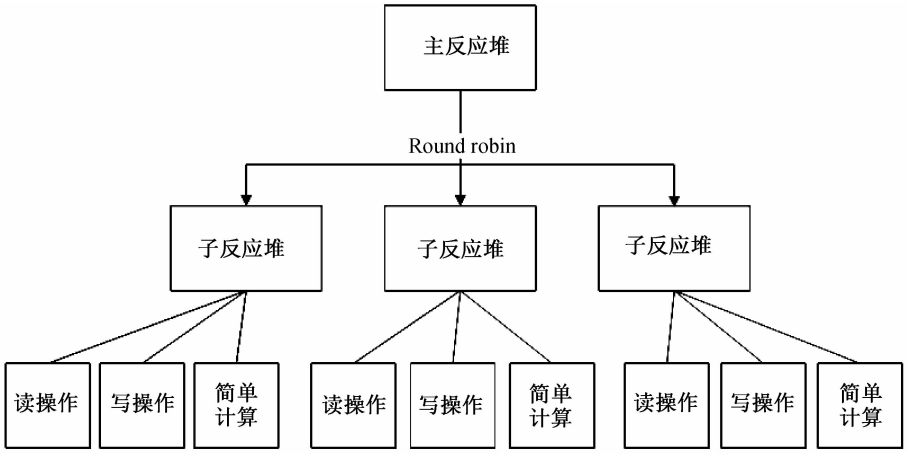

图 4　Reactor 池设计图

2.3　缓冲 Buffer 设计

读写缓冲区是服务端框架中必然存在的模块。TCP 通信中数据的传递流程示意图如图 5 所示。当客户端与服务端建立连接后，客户端准备向服务端发送 10 KB 数据，该数据会先进入客户端的 TCP 缓冲区，但由于 TCP 拥塞控制机制的存在[15]，如果服务端前窗口只能容纳 3 KB 数据，此时 Write 函数会发生阻塞，影响线程复用，从而阻碍了线程处理文件描述符上后续的活动事件，所以需要缓冲区将这部分还未发送的数据存起来，然后在多路复用机制中注册 EPOLLOUT 事件[16-17]，待窗口滑动有空间继续接纳数据后，再去完成后续发送工作，使得 IO 线程能及时空出来去处理新的请求，这也是服务端框架中缓冲区存在的必要性。

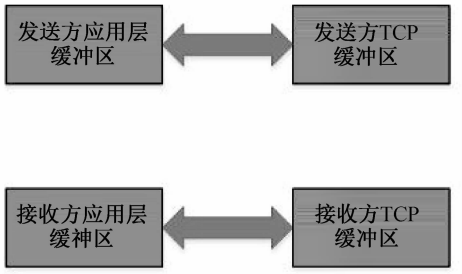

图 5　TCP 数据传递流程示意图

自适应缓冲区的结构设计是基于 Vector 数据结构来实现的，因为 Vector 容器具有自动扩容的机制，当某一次读取或写入数据量较大时，Vector 会根据其自动扩容的性质重新分配一块较大的内存，实现尽可能地一次性读取或写入的全部数据。同时如果服务端接收的数据长度长时间处在大于缓冲区

长度时,由于 Vector 的自动扩容,避免了重复的内存分配工作,提升了效率[18]。自适应缓冲区的结构设计图如图 6 所示,其中写索引与读索引的差值为缓冲区中可读区域的大小,尾部索引的大小为 Vector 容器中的容量,尾部索引与写索引的差值为缓冲区中可写区域的大小。传统解决方案中,当某个数据包接收完成后,通过将接收的数据向后腾挪的方式为数据包的长度值留出空间,所以每次数据的接收都将导致一次内存的移动。本文设计的内存缓冲区中头部区的大小会提前预留,当向缓冲区中接收完所有数据后,可直接将已接收数据的长度值写入提前预留的头部区域,避免了频繁的内存移动。

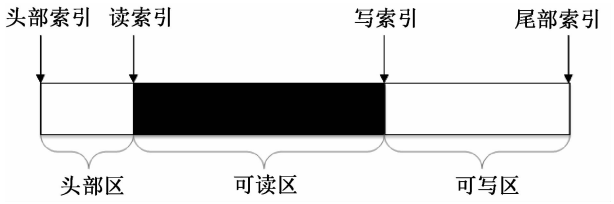

图 6 自适应缓冲区的结构设计图

2.4 双缓冲日志设计

系统日志是服务端程序开发过程中必须面对的模块,因为系统日志是故障诊断的重要手段,它可以记录程序运行时的动态信息,帮助维护人员分析和重现错误,进而更正系统错误,提高系统运行的可靠性[21],所以经常需要对日志写入磁盘。但是,对数据库或磁盘的读写效率相比与 CPU 运算效率相差较大[22],这就造成了当 IO 线程执行到日志写入时,会增大线程对事件的响应时间,降低服务端程序的吞吐量,所以日志写入过程的优化对提升服务端程序吞吐量显得尤为重要。基于此,本文设计了双缓冲日志模块,双缓冲日志处理流程如图 7 所示。该模块具备两个线程 Thread1、Thread2。Thread1 是产生日志的线程,即前台线程;Thread2 是负责将日志写入磁盘或数据库的线程,即后台线程。有两种情况会唤醒后台线程对前台已缓存的日志进行写入,一是当后台线程超时,通过条件变量的方式唤醒后台线程,以免日志丢失;二是前台线程所持有的日志缓冲区消耗完后,唤醒后台线程递补前台线程的日志缓冲区,随即后台线程开始向磁盘或数据库写入日志。后台线程在对日志向磁盘或数据库写入的过程中,此刻前台线程已经持有新的日志缓冲区应对日志的产生。此外,为了防止短时间内日志量的急剧增多,为 Thread1 与 Thread2 各自配备了两块缓冲区,即图 7 中的 FrontBuffer1、FrontBuffer2、BackBuffer1、BackBuffer2。当前台线程短时间内日志量激增,使得当前台线程一块缓冲区写满时,不必立即唤醒后台线程递补缓冲区 BackBuffer1,而是使用同一线程的备用缓冲区 FrontBuffer2 进行递补,避免了频繁地线程切换带来的开销,进一步降低了前台线程的等待时间,提升了整体框架的吞吐量。

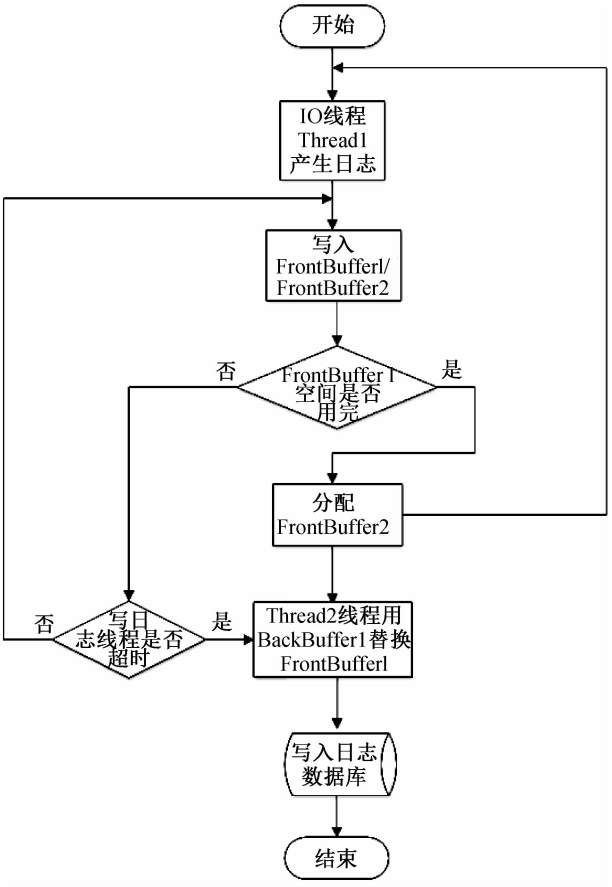

图 7 双缓冲日志处理流程

3 实验设计与实现

为了验证本文所实现的服务端框架的吞吐量,分别在单线程与多线程下与主流服务端框架做了对比实验,测试方法参考了 Boost.Asio 的性能测试[23],采用 ping pong 协议搭配日志写入过程来测试吞吐量,具体流程为:客户端发送 4096 byte 数据给服务端,服务端把收到的数据写入本地磁盘后(参与对比的框架搭配 log4cpp 日志库),再将收到的数据回传给客户端,直到某一方断开连接或超时,每个测试时间默认最长持续 15 s。同时,千兆网带宽理论传输能力为 125 MiB/s,除去包头、MAC、preamble 等数据后,吞吐量约为 112 MiB/s,这远低于当前 CPU 处理能力,所以将客户端与服务端程序都运行在同一台服务器,避免被千兆网带宽限制[24]。实验环境中,操作系统为 centos-release-7-

7. 1908. 0. el7. centos. x86 _64，CPU 为 Intel（R）Xeon（R）Silver 4110 CPU @ 2. 10 GHz，内存为 16 G DDR4。

单线程下测试并发连接数分别为 1、10、100、1000 时，与 libevent2 对比，其中服务端与客户端程序均为单线程。表 1 为本文所实现框架与 libevent2 吞吐量的对比结果，两者在 1 个连接上升至 10 个连接的阶段中，吞吐量增长速度最快。两者处于 100 连接时，吞吐量均达到峰值。图 8 为该对比结果的曲线图，从图 8 中可以看出两者吞吐量随着连接的增加较为接近，但本文服务端框架吞吐量整体高于 libevent2。

表 1　单线程、不同连接数时本文框架与 libevent2 的吞吐量

连接数/个	吞吐量/(MiB·s^{-1})	
	本文框架	libevent2
1	58. 23	57. 41
10	508. 87	469. 05
100	564. 22	492. 63
1000	404. 19	361. 08

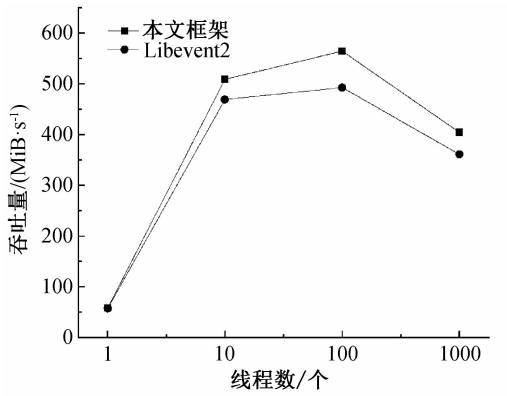

图 8　单线程下本文框架与 libevent2 吞吐量随线程数变化的曲线

在多线程下分别测试并发连接数为 100 或 1000，线程数分别为 2、4、6、8 时，对本文所实现框架与 Boost. Asio 进行对比。并发连接数为 100 时本文所实现框架与 Boost. Asio 的吞吐量见表 2，并发连接数为 1000 时的吞吐量见表 3。

表 2　100 连接、不同线程数时本文框架与 Boost. Asio 的吞吐量

线程数/个	吞吐量/(MiB·s^{-1})	
	本文框架	Boost. Asio
2	858. 17	649. 03
4	1402. 73	1141. 29
6	1879. 44	1326. 52
8	1743. 92	1479. 87

表 3　1000 连接、不同线程数时本文框架与 Boost. Asio 的吞吐量

线程数/个	吞吐量/(MiB·s^{-1})	
	本文框架	Boost. Asio
2	640. 24	611. 79
4	1287. 96	1065. 32
6	1598. 33	1307. 91
8	1689. 31	1248. 44

当并发连接数为 100 或 1000，线程数分别为 2、4、6、8 时，本文框架与 Boost. Asio 的吞吐量随线程数变化曲线如图 9 所示，从图中可以明显看出本文所实现服务端框架与 Boost. Asio 吞吐量变化的趋势。

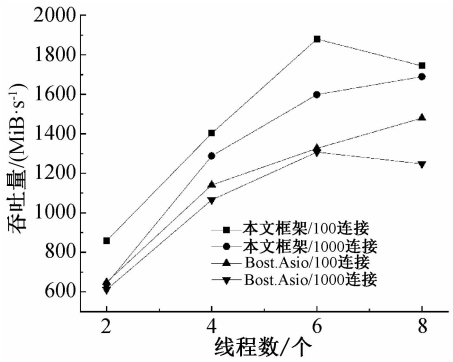

图 9　本文框架与 Boost. Asio 的吞吐量随线程数变化的曲线

通过在 ping pong 协议中搭配日志写入的测试方式，对比了本文服务端框架与传统服务端框架，从实验结果可以看出，当测试数据包大小为 4096 byte 时，本文实现的服务端框架数据吞吐量整体上好于传统服务端框架，在多线程环境中具有一定的优势。从图 9 中可以看出，在多线程环境下，本文服务端框架与 Boost. Asio 都启用两个线程时，吞吐量较为接近，但随着线程数的增加，尤其当各自启用 8 个线程时本文服务端框架的吞吐量明显高于 Boost. Asio。并且在两者都处于 8 线程环境中，从 100 连接转变到 1000 连接时，两种框架下的吞吐量的增长速度均呈下降趋势，Boost. Asio 吞吐量降低了 15. 6%，但本文所实现的框架仅降低了 3. 1%，可看出在本文所实现的服务端框架整体吞吐量的优势较为明显。这是因为本文实现框架通过采用固定数目的 Reactor 线程池，使得服务端程序的总体处理能力不会随着连接数目的上升而陡降。自适应缓冲区的大小也会依据其所传送数据包的大小做自适应调整，后续接收相同大小数据包时则不会再重复分配内存，也实现尽可能地一次性读取或写入全部数据。

同时日志结构也通过双缓冲技术优化了写入操作，从而提升了整体服务端框架的吞吐量。

4 结 语

本文在对现有服务端框架 Boost.Asio 和 libevent 进行分析后，在 Reactor 设计模式与非阻塞 IO 的环境下，通过设计 Reactor 线程池、自适应缓冲区以及双缓冲日志等，在一定条件下提升了服务端的吞吐量。本文实现的服务端框架已经在浙江理工大学智慧校园项目中投入使用，整体运行正常。今后工作将准备在已实现的服务端框架基础上，从负载均衡、分布式集群入手，以应对更大数据量请求的场景。

参考文献：

[1] 杨志军，孙洋洋. 区分站点状态的两级轮询控制系统[J]. 计算机应用，2019，39(5)：1416-1420.

[2] 刘珂男，童薇，冯丹，等. 一种灵活高效的虚拟CPU调度算法[J]. 软件学报，2017，28(2)：398-410.

[3] Rosa L，Rodrigues L，Lopes A，et al. Self management of adaptable component-based applications[J]. IEEE Transactions on Software Engineering，2013，39(3)：403-421.

[4] 杨迪. 基于容器云的微服务系统[J]. 电信科学，2018，34(9)：169-178.

[5] Zhao F，Wei L N，Chen H B. Optimal time allocation for wireless information and power transfer in wireless powered communication systems[J]. IEEE Transactions on Vehicular Technology，2016，65(3)：1830-1835.

[6] Jiang W，Feng G，Qin S. Optimal cooperative content caching and delivery policy for heterogeneous cellular networks[J]. IEEE Transactions on Mobile Computing，2017，16(5)：1382-1393.

[7] Vaarandi R，Pihelgas M. LogCluster. A data clustering and pattern mining algorithm for event logs. Internationl Conference on Network and Service Management. IEEE，2016：1-7.

[8] 罗剑锋. Boost程序库完全开发指南[M]. 4版. 北京：电子工业出版社，2017：491-530.

[9] Gul K S Q，王鹏，罗森林，潘丽敏，等. 一种高并发网络Web应用技术研究[J]. 信息网络安全，2017(12)：29-35.

[10] 顾振德，刘子辰，龙隆，等. 基于Netty的IoT终端通信服务系统设计[J]. 计算机应用与软件，2019，36(4)：135-139.

[11] 叶柏龙，刘蓬. Proactor模式的NIO框架的设计与实现[J]. 计算机应用与软件，2014，31(9)：110-113.

[12] 曹文彬，谭新明，刘备，等. 基于事件驱动的高性能WebSocket服务器的设计与实现[J]. 计算机应用与软件，2018，35(1)：21-27.

[13] Guan H B，Ma R H，Li J. Workload-Aware credit scheduler for improving network IO performance in virtualization environment[J]. IEEE Transactions on Cloud Computing，2014，2(2)：130-142.

[14] 胡晓喻，陈庆奎. 智能家居接入服务器策略的设计与实现[J]. 计算机工程与设计，2017，38(2)：544-549.

[15] Wu D P，Zhang P N，Wang H G，et al. Node service ability aware packet forwarding mechanism in intermittently connected wireless networks[J]. IEEE Transactions on Wireless Communications，2016，15(12)：8169-8181.

[16] 邱杰，朱晓姝，孙小雁. 基于Epoll模型的消息推送研究与实现[J]. 合肥工业大学学报(自然科学版)，2016，39(4)：476-480.

[17] 林祥辉，张瑾，黄康平，等. 一种基于内存的高效在线数据处理服务框架[J].中文信息学报，2014，28(1)：80-86.

[18] Zou L，Wang Z D，Han Q L，et al. Ultimate boundedness control for networked systems with try-once-discard protocol and uniform quantization effects[J]. IEEE Transactions on Automatic Control，2017，62(12)：6582-6588.

[19] Thakur A，Goraya M S. A taxonomic survey on load balancing in cloud[J]. Journal of Network and Computer Applications，2017，98：43-57.

[20] 陈俊，郑梦娜，高金凤. 一类网络控制系统数据包乱序的补偿控制[J]. 浙江理工大学学报(自然科学版)，2017，37(5)：691-698.

[21] 廖湘科，李姗姗，董威，等. 大规模软件系统日志研究综述[J]. 软件学报，2016，27(8)：1934-1947.

[22] 林祥辉，张瑾，黄康平，等. 一种基于内存的高效在线数据处理服务框架[J].中文信息学报，2014，28(1)：80-86.

[23] 王建辉. 基于高性能网络库Asio的测控服务器设计与实现[D]. 合肥：安徽大学，2015：5-39.

[24] 丁尚，童鑫，陈艳，等. 基于简单再生码的带宽感知的分布式存储节点修复优化[J]. 软件学报，2017，28(8)：1940-1951.

(责任编辑：康 锋)