链滴

如何实现一个 Web Server

作者：tinylcy

原文链接：https://ld246.com/article/1512892730490

来源网站：链滴

最近重构了去年造的一个轮子 Vino。Vino 旨在实现一个轻量并且能够保证性能的 Web Server，仅注 Web Server 的本质部分。在重构过程中，Vino 借鉴了许多优秀开源项目的思想，如 Nginx、Mongoose 和 Webbench。因此，对比上一个版本的 Vino，现在的 Vino 不仅性能得到提升，而且设计更为优雅、健壮 :D。

本文将会对 Vino 目前所具备的关键特性进行阐述，并总结开发过程中的一点心得。

单线程 + Non-Blocking

Vino 整体采用了基于事件驱动的单线程 + Non-Blocking 模型。采用单线程模型，避免了系统分配线程及线程之间通信的开销，同时降低了内存的耗用。由于采用了单线程模型，为了更好的提高线程用率，Vino 将默认 Blocking 的 I/O 设置为 Non-Blocking I/O，即在线程读/写数据的过程中，如缓冲区为空/缓冲区满，线程不会阻塞，而是立即返回，并设置 errno。

Vino 最初的灵感来源于 Computer Systems: A Programmer's Perspective 一书讲述网络编程时实的一个简单的 Web Server，每到来一个请求，Web Server 都会 fork 一个进程去处理。显然，在高发的场景下，这种模型是不合理的。每次 fork 进程会带来巨大的开销，并且系统中进程的数量是有的。同时，伴随多进程带来的进程调度的开销也不可小觑，CPU 会花费大量的时间用于决定调用哪一进程。进程调度引发的进程上下文之间的切换，也需要耗费相当大的资源。

很容易联想到采用多线程模型来替代多进程模型，相比于多进程模型，多线程模型占用的系统资源会大降低，但是本质上并没有减小线程调度带来的开销。为了减小由线程调度导致的开销，我们可以采线程池模型，即固定线程的数量，但是问题依旧存在：因为 Linux 默认 I/O 是阻塞（Blocking）的，果线程池中所有的线程同时阻塞于正在处理的请求，那么新到来的请求就没有线程去处理了。因此，果我们用 Non-Blocking 的 I/O 替换默认的 Blocking I/O，线程将不会阻塞于数据的读写，问题便得到解决。

HTTP Keep-Alive

Vino 支持 HTTP 长连接（Persistent Connections），即多个请求可以复用同一个 TCP 连接，以此少由 TCP 建立/断开连接所带来的性能开销。每到来一个请求，Vino 会对请求进行解析，判断请求头是否存在 Connection: keep-alive 请求头。如果存在，在处理完一个请求后会保持连接，并对数据冲区（用于保存请求内容，响应内容）及状态标记进行重置，否则，关闭连接。

关于 HTTP Keep-Alive 的优势，RFC 2616 有着更完善的总结，引用如下。

> ● By opening and closing fewer TCP connections, CPU time is saved in routers and hosts (clients, servers, proxies, gateways, tunnels, or caches), and memory used for TCP protocol control blocks can be saved in hosts.
>
> ● HTTP requests and responses can be pipelined on a connection. Pipelining allows a client to make multiple requests without waiting for each response, allowing a single TCP connection to be used much more efficiently, with much lower elapsed time.
>
> ● Network congestion is reduced by reducing the number of packets caused by TCP opens and by allowing TCP sufficient time to determine the congestion state of the network.
>
> ● Latency on subsequent requests is reduced since there is no time spent in TCP's connection opening handshake.
>
> ● HTTP can evolve more gracefully, since errors can be reported without the penalty of closing the TCP connection. Clients using future versions of HTTP might optimistically try a new feature, but if communicating with an older server, retry with old semantics after an error is reported.

定时器 Timer

如果一个请求在建立连接后迟迟没有发送数据，或者对方突然断电，应该如何处理？我们需要实现定器来处理超时的请求。Vino 定时器的实现参考了 Nginx 的设计，Nginx 使用一颗红黑树来存储各个时事件，每次事件循环时从红黑树中不断找出最小（早）的事件，如果超时则触发超时处理。为了简实现，在 Vino 中，我实现了一个小顶堆来存储定时事件，如果被处理的定时事件同时支持长连接，么在该请求处理完毕后会更新该请求对应的定时器，也就是重新计时。定时器相关代码见 vn_event_timer.h 和 vn_event_timer.c。

HTTP Parser

由于网络的不确定性，我们并不能保证一次就能读取所有的请求数据。因此，对于每一个请求，我们会开辟一段缓冲区用于保存已经读取到的数据。同时，我们需要同时对读取到的数据进行解析，以保读取到的数据都是合理的数据，例如，假设目前缓冲区内的数据为 GET /index.html HTT，那么下一读取到的字符必须为 P，否则，应立即检测出当前请求是一个异常的请求，并主动关闭当前的连接。

基于以上分析，我们需要实现一个 HTTP 状态机（Parser）来维持当前的解析状态，Vino 状态机的现参考了 Nginx 的设计，并对 Nginx 的实现做了简化。HTTP Parser 相关代码见 vn_http_parse.h 和 n_http_parse.c。

Memory Pool

我们一般使用 malloc/calloc/free 来分配/释放内存，但是这些函数对于一些需要长时间运行的程序说会有一些弊端。频繁使用这些函数分配和释放内存，会导致内存碎片，不容易让系统直接回收内存典型的例子就是大并发频繁分配和回收内存，会导致进程的内存产生碎片，并且不会立马被系统回收。

使用内存池分配内存，可以在一定程度上提升内存分配的效率，不需要每次都调用 malloc/calloc 函。同时，使用内存池使得内存管理更加简单。在 Vino 中，针对每一个请求，Vino 都会为其分配一或个内存池（各个内存池形成一个单链表），在请求处理完毕后，一并释放所有的内存。

Vino 内存池的实现依旧参考了 Nginx 的实现，并做了简化，Memory Pool 相关代码见 vn_palloc.h 和 vn_palloc.c。

其他

在开发 Vino 的过程中，还有许多需要考虑和权衡的地方。响应请求时，如果用户请求的是一个很大文件，导致写缓冲区满，我们如何更好的设计响应缓冲区？如何更高效的设计底层数据结构（如字符、链表、小顶堆等）？如何更优雅的解析命令行参数？如何对特定信号进行处理？如何更健壮的处理误信息？当代码的数量达到一定程度后，如何更快的定位异常代码？

Vino 的开发 & 重构暂时告一段落，源码放在了 GitHub 上。当然，Vino 还有许多不足之处，以及实现的特性。

- 仅支持 HTTP GET 方法，暂不支持其他 HTTP method。
- 暂不支持动态请求的处理。
- 支持的 HTTP/1.1 特性有限。
- ...

写这篇文章，希望对初学者有所帮助。

参考

[1] Vino, https://github.com/tinylcy/vino .

[2] Computer Systems: A Programmer's Perspective, http://csapp.cs.cmu.edu/ .

[3] Advanced Programming in the UNIX Environment (3rd Edition), https://www.amazon.ca/Advanced-Programming-UNIX-Environment-3rd/dp/0321637739 .

[4] Unix Network Programming, Volume 1, https://www.amazon.ca/Unix-Network-Programming-Sockets-Networking/dp/0131411551 .

[5] Nginx, https://github.com/nginx/nginx .

[6] Mongoose, https://github.com/cesanta/mongoose .

[7] Web Bench, http://home.tiscali.cz/~cz210552/webbench.html .

[8] Zaver, https://github.com/zyearn/zaver .

[9] RFC 2616, https://tools.ietf.org/html/rfc2616 .

[10] How to use epoll? A complete example in C, https://banu.com/blog/2/how-to-use-epoll--complete-example-in-c/ .