如何写一个Web服务器

最近两个月的业余时间在写一个私人项目，目的是在Linux下写一个高性能Web服务器，名字叫Zaver。主体框架和基本功能已完成，还有一些高级功能日后会逐渐增加，代码放在了github。Zaver的框架会在代码量尽量少的情况下接近工业水平，而不像一些教科书上的toy server为了教原理而舍弃了很多原本server应该有的东西。 在本篇文章中，我将一步步地阐明Zaver的设计方案和开发过程中遇到的困难以及相应的解决方法。

为什么要重复造轮子

几乎每个人每天都要或多或少和Web服务器打交道，比较著名的Web Server有Apache Httpd、Nginx、IIS。这些软件跑在成千上万台机器上为我们提供稳定的服务，当你打开浏览器输入网址，Web服务器就会把信息传给浏览器然后呈现在用户面前。那既然有那么多现成的、成熟稳定的web服务器，为什么还要重新造轮子，我认为理由有如下几点：

- 夯实基础。一个优秀的开发者必须有扎实的基础，造轮子是一个很好的途径。学编译器？边看教材变写一个。学操作系统？写一个原型出来。编程这个领域只有自己动手实现了才敢说真正会了。现在正在学网络编程，所以就打算写一个Server。
- 实现新功能。成熟的软件可能为了适应大众的需求导致不会太考虑你一个人的特殊需求，于是只能自己动手实现这个特殊需求。关于这一点Nginx做得相当得好了，它提供了让用户自定义的模块来定制自己需要的功能。
- 帮助初学者容易地掌握成熟软件的架构。比如Nginx，虽然代码写得很漂亮，但是想完全看懂它的架构，以及它自定义的一些数据结构，得查相当多的资料和参考书籍，而这些架构和数据结构是为了提高软件的可伸缩性和效率所设计的，无关高并发server的本质部分，初学者会迷糊。而Zaver用最少的代码展示了一个高并发server应有的样子，虽然没有Nginx性能高，得到的好处是没有Nginx那么复杂，server架构完全展露在用户面前。

教科书上的server

学网络编程，第一个例子可能会是Tcp echo服务器。 大概思路是server会listen在某个端口，调用accept等待客户的connect，等客户连接上时会返回一个fd(file descriptor)，从fd里read，之后write同样的数据到这个fd，然后重新accept，在网上找到一个非常好的代码实现，核心代码是这样的：

```
1  while ( 1 ) {
2
3      /*  Wait for a connection, then accept() it  */
4
5      if ( (conn_s = accept(list_s, NULL, NULL) ) < 0 ) {
6          fprintf(stderr, "ECHOSERV: Error calling accept()\n");
```

```
7           exit(EXIT_FAILURE);
8       }
9
10
11      /*  Retrieve an input line from the connected socket
12          then simply write it back to the same socket.     */
13
14      Readline(conn_s, buffer, MAX_LINE-1);
15      Writeline(conn_s, buffer, strlen(buffer));
16
17
18      /*  Close the connected socket  */
19
20      if ( close(conn_s) < 0 ) {
21          fprintf(stderr, "ECHOSERV: Error calling close()\n");
22          exit(EXIT_FAILURE);
23      }
24  }
```

完整实现在这里。如果你还不太懂这个程序，可以把它下载到本地编译运行一下，用telnet测试，你会发现在telnet里输入什么，马上就会显示什么。如果你之前还没有接触过网络编程，可能会突然领悟到，这和浏览器访问某个网址然后信息显示在屏幕上，整个原理是一模一样的！ 学会了这个echo服务器是如何工作的以后，在此基础上拓展成一个web server非常简单，因为HTTP是建立在TCP之上的，无非多一些协议的解析。client在建立TCP连接之后发的是HTTP协议头和（可选的）数据，server接受到数据后先解析HTTP协议头，根据协议头里的信息发回相应的数据，浏览器把信息展现给用户,一次请求就完成了。

这个方法是一些书籍教网络编程的标准例程，比如《深入理解计算机系统》（CSAPP）在讲网络编程的时候就用这个思路实现了一个最简单的server，代码结构和上面的echo服务器代码类似，完整实现在这里，非常短，值得一读，特别是这个server即实现了静态内容又实现了动态内容，虽然效率不高，但已达到教学的目的。之后这本书用事件驱动优化了这个server，关于事件驱动会在后面讲。

虽然这个程序能正常工作，但它完全不能投入到工业使用，原因是server在处理一个客户请求的时候无法接受别的客户，也就是说，这个程序无法同时满足两个想得到echo服务的用户，这是无法容忍的，试想一下你在用微信，然后告诉你有人在用，你必须等那个人走了以后才能用。

然后一个改进的解决方案被提出来了：accept以后fork，父进程继续accept，子进程来处理这个fd。这也是一些教材上的标准示例，代码大概长这样：

```
1  /* Main loop */
2      while (1) {
3          struct sockaddr_in their_addr;
4          size_t size = sizeof(struct sockaddr_in);
5          int newsock = accept(listenfd, (struct sockaddr*)&their_addr, &size);
6          int pid;
7
8          if (newsock == -1) {
9              perror("accept");
10             return 0;
11         }
12
13         pid = fork();
14         if (pid == 0) {
15             /* In child process */
16             close(listenfd);
17             handle(newsock);
18             return 0;
19         }
20         else {
21             /* Parent process */
22             if (pid == -1) {
```

```
23                  perror("fork");
24                  return 1;
25              }
26              else {
27                  close(newsock);
28              }
29          }
30      }
```

完整代码在这里。表面上，这个程序解决了前面只能处理单客户的问题，但基于以下几点主要原因，还是无法投入工业的高并发使用。

- 每次来一个连接都fork，开销太大。任何讲Operating System的书都会写，线程可以理解为轻量级的进程，那进程到底重在什么地方？《Linux Kernel Development》有一节（Chapter3）专门讲了调用fork时，系统具体做了什么。地址空间是copy on write的，所以不造成overhead。但是其中有一个复制父进程页表的操作，这也是为什么在Linux下创建进程比创建线程开销大的原因，而所有线程都共享一个页表（关于为什么地址空间是COW但页表不是COW的原因，可以思考一下）。

- 进程调度器压力太大。当并发量上来了，系统里有成千上万进程，相当多的时间将花在决定哪个进程是下一个运行进程以及上下文切换，开销非常大，具体的分析请看《A Design Framework for Highly Concurrent Systems》。

- 在heavy load下多个进程消耗太多的内存，在进程下，每一个连接都对应一个独立的地址空间；即使在线程下，每一个连接也会占用独立。此外父子进程之间需要发生IPC，高并发下IPC带来的overhead不可忽略。

换用线程虽然能解决fork开销的问题，但是调度器和内存的问题还是无法解决。所以进程和线程在本质上是一样的，被称为process-per-connection model。因为无法处理高并发而不被业界使用。

一个非常显而易见的改进是用线程池，线程数量固定，就没上面提到的问题了。基本架构是有一个loop用来accept连接，之后把这个连接分配给线程池中的某个线程，处理完了以后这个线程又可以处理别的连接。看起来是个非常好的方案，但在实际情况中，很多连接都是长连接（在一个TCP连接上进行多次通信），一个线程在收到任务以后，处理完第一批来的数据，此时会再次调用read，天知道对方什么时候发来新的数据，于是这个线程就被这个read给阻塞住了（因为默认情况下fd是blocking的，即如果这个fd上没有数据，调用read会阻塞住进程），什么都不能干，假设有n个线程，第(n+1)个长连接来了，还是无法处理。

怎么办？我们发现问题是出在read阻塞住了线程，所以解决方案是把blocking I/O换成non-blocking I/O，这时候read的做法是如果有数据则返回数据，如果没有可读数据就返回-1并把errno设置为EAGAIN，表明下次有数据了我再来继续读(man 2 read)。

这里有个问题，进程怎么知道这个fd什么时候来数据又可以读了？这里要引出一个关键的概念，事件驱动/事件循环。

事件驱动(Event-driven)

如果有这么一个函数，在某个fd可以读的时候告诉我，而不是反复地去调用read，上面的问题不就解决了？这种方式叫做事件驱动，在linux下可以用select/poll/epoll这些I/O复用的函数来实现（man 7 epoll），因为要不断知道哪些fd是可读的，所以要把这个函数放到一个loop里，这个就叫事件循环（event loop）。示例代码如下：

```
while (!done)
{
  int timeout_ms = max(1000, getNextTimedCallback());
  int retval = epoll_wait(epds, events, maxevents, timeout_ms);

  if (retval < 0) {
    处理错误
  } else {
    处理到期的 timers

    if (retval > 0) {
      处理 IO 事件
    }
  }
}
```

在这个while里，调用epoll_wait会将进程阻塞住，直到在epoll里的fd发生了当时注册的事件。 这里有个非常好的例子来展示epoll是怎么用的。 需要注明的是，select/poll不具备伸缩性，复杂度是O(n)，而epoll的复杂度是O(1)，在Linux下工业程序都是用epoll（其它平台有各自的API，比如在Freebsd/MacOS下用kqueue）来通知进程哪些fd发生了事件，至于为什么epoll比前两者效率高，请参考这里。

事件驱动是实现高性能服务器的关键，像Nginx，lighttpd，Tornado，NodeJs都是基于事件驱动实现的。

Zaver

结合上面的讨论，我们得出了一个事件循环+ non-blocking I/O + 线程池的解决方案，这也是Zaver的主题架构（同步的事件循环+non-blocking I/O又被称为Reactor模型）。 事件循环用作事件通知，如果listenfd上可读，则调用accept，把新建的fd加入epoll中；是普通的连接fd，将其加入到一个生产者-消费者队列里面，等工作线程来拿。 线程池用来做计算，从一个生产者-消费者队列里拿一个fd作为计算输入，直到读到EAGAIN为止，保存现在的处理状态（状态机），等待事件循环对这个fd读写事件的下一次通知。

开发中遇到的问题

Zaver的运行架构在上文介绍完毕，下面将总结一下我在开发时遇到的一些困难以及一些解决方案。把开发中遇到的困难记录下来是个非常好的习惯，如果遇到问题查google找到个解决方案直接照搬过去，不做任何记录，也没有思考，那么下次你遇到同样的问题，还是会重复一遍搜索的过程。有时我们要做代码的创造者，不是代码的“搬运工”。做记录定期回顾遇到的问题会使自己成长更快。

- 如果将fd放入生产者-消费者队列中后，拿到这个任务的工作线程还没有读完这个fd，因为没读完数据，所以这个fd可读，那么下一次事件循环又返回这个fd，又分给别的线程，怎么处理?

答：这里涉及到了epoll的两种工作模式，一种叫边缘触发（Edge Triggered），另一种叫水平触发（Level Triggered）。ET和LT的命名是非常形象的，ET是表示在状态改变时才通知（eg，在边缘上从低电平到高电平），而LT表示在这个状态才通知（eg，只要处于低电平就通知），对应的，在epoll里，ET表示只要有新数据了就通知（状态的改变）和“只要有新数据”就一直会通知。

举个具体的例子：如果某fd上有2kb的数据，应用程序只读了1kb，ET就不会在下一次epoll_wait的时候返回，读完以后又有新数据才返回。而LT每次都会返回这个fd，只要这个fd有数据可读。所以

在Zaver里我们需要用epoll的ET，用法的模式是固定的，把fd设为nonblocking，如果返回某fd可读，循环read直到EAGAIN（如果read返回0，则远端关闭了连接）。

- 如果某个线程在处理fd的同时，又有新的一批数据发来，该fd可读（注意和上面那个问题的区别，一个是处理同一批数据时，一个是处理新来的一批数据），那么该fd会被分给另一个线程，这样两个线程处理同一个fd肯定就不对了。

答：用EPOLLONESHOT解决这个问题。在fd返回可读后，需要显式地设置一下才能让epoll重新返回这个fd。

- 当server和浏览器保持着一个长连接的时候，1)浏览器突然被关闭了，2)断电了/被拔网线了，那么server端怎么处理这个socket?

答：对于1)，此时该fd在事件循环里会返回一个可读事件，然后就被分配给了某个线程，该线程read会返回0，代表对方已关闭这个fd，于是server端也调用close即可。对于2)，协议栈无法感知，SO_KEEPALIVE超时时间太长不适用，所以只能通过应用层timer超时事件解决。

- 如何设计和实现timer?

答：Nginx把timer实现成了rbtree，这就很奇怪，timer模块需要频繁找最小的key（最早超时的事件）然后处理后删除，这个场景下难道不是最小化堆是最好的数据结构么？然后通过搜索得知阿里的Tengine将timer的实现了4-heap（四叉最小堆）。四叉堆是二叉堆的变种，比二叉堆有更浅的深度和更好的CPU Cache命中率。Tengine团队声称用最小堆性能提升比较明显。在Zaver中为了简化实现，使用了二叉堆来实现timer的功能。

- 把fd设置为non-blocking后，while循环需要不断地read直到返回-1，errno为EAGAIN，然后等待下次epoll_wait返回这个fd再继续处理。这时就遇到了一个blocking read不曾遇到的问题：数据可能分包到达，于是在协议解析到一半的时候read就返回-1，所以我们必须将已读到的数据保存下来，并维护一个状态，以表示是否还需要数据，比如解析HTTP Request Header的时候, 读到`GET /index.html HTT`就结束了，在blocking I/O里只要继续read就可以，但在nonblocking I/O，我们必须维护这个状态，下一次必须读到'P'，否则HTTP协议解析错误。

答：解决方案是维护一个状态机，在解析Request Header的时候对应一个状态机，解析Header Body的时候也维护一个状态机，Zaver状态机的时候参考了Nginx在解析header时的实现，我做了一些精简和设计上的改动。

- 怎么较好的实现header的解析

答：HTTP header有很多，必然有很多个解析函数，比如解析`If-modified-since`头和解析`Connection`头是分别调用两个不同的函数，所以这里的设计必须是一种模块化的、易拓展的设计，可以使开发者很容易地修改和定义针对不同header的解析。Zaver的实现方式参考了Nginx的做法，定义了一个struct数组，其中每一个struct存的是key，和对应的函数指针hock，如果解析到的headerKey == key，就调hock。定义代码如下

```
zv_http_header_handle_t zv_http_headers_in[] = {
    {"Host", zv_http_process_ignore},
    {"Connection", zv_http_process_connection},
    {"If-Modified-Since", zv_http_process_if_modified_since},
    ...
    {"", zv_http_process_ignore}
};
```

- 怎样存储header

答：Zaver将所有header用链表连接了起来，链表的实现参考了Linux内核的双链表实现（list_head），它提供了一种通用的双链表数据结构，代码非常值得一读，我做了简化和改动，代码在这里。

- 压力测试

答：压力测试为了测量网站对高并发的承受程度，在哪个并发度会使网站挂掉。这个有很多成熟的方案了，比如http_load, webbench, ab等等。我最终选择了webbench，理由是简单，用fork来模拟client，代码只有几百行，出问题可以马上根据webbench源码定位到底是哪个操作使Server挂了，这就说到了我在做压力测试时遇到一个问题，然后看了一下Webbench的源码，就很快找到了问题所在（并且非常推荐C初学者看一看它的源码，只有几百行，但是涉及了命令行参数解析、fork子进程、父子进程用pipe通信、信号handler的注册、构建HTTP协议头的技巧等一些编程上的技巧）。

之前提到的那个问题是：用Webbech测试，Server在测试结束时挂了。

百思不得其解，不管时间跑多久，并发量开多少，都是在最后webbench结束的时刻，server挂了，所以我猜想肯定是这一刻发生了什么“事情”。 开始调试定位错误代码，我用的是打log的方式，后面的事实证明在这里这不是很好的方法，在多线程环境下要通过看log的方式定位错误是一件比较困难的事。最后log输出把错误定位在向socket里write对方请求的文件，也就是系统调用挂了，write挂了难道不是返回-1的吗？于是唯一的解释就是进程接受到了某signal，这个signal使进程挂了。于是用strace重新进行测试，在strace的输出log里发现了问题，系统在write的时候接受到了SIGPIPE，默认的signal handler是终止进程。SIGPIPE产生的原因为，对方已经关闭了这个socket，但进程还往里面写。所以我猜想webbench在测试时间到了之后不会等待server数据的返回直接close掉所有的socket。抱着这样的怀疑去看webbench的源码，果然是这样的，webbench设置了一个定时器，在正常测试时间会读取server返回的数据，并正常close；而当测试时间一到就直接close掉所有socket，不会读server返回的数据，这就导致了zaver往一个已被对方关闭的socket里写数据，系统发送了SIGPIPE。

解决方案也非常简单，把SIGPIPE的信号handler设置为SIG_IGN，忽略该信号即可。

不足

目前Zaver还有很多改进的地方，比如：

- 现在新分配内存都是通过malloc的方式，之后会改成内存池的方式
- 还不支持动态内容，后期开始考虑增加php的支持
- HTTP/1.1较复杂，目前只实现了几个主要的（keep-alive, browser cache）的header解析
- 不活动连接的超时过期还没有做
- ...

总结

本文介绍了Zaver，一个结构简单，支持高并发的http服务器。 基本架构是事件循环 + non-blocking I/O + 线程池。 Zaver的代码风格参考了Nginx的风格，所以在可读性上非常高。另外，Zaver提供了配置文件和命令行参数解析，以及完善的Makefile和源代码结构，也可以帮助任何一个C初学者入门一个项目是怎么构建的。

参考资料

[1] https://github.com/zyearn/zaver
[2] http://nginx.org/en/
[3] 《linux多线程服务端编程》
[4] http://www.martinbroadhurst.com/server-examples.html
[5] http://berb.github.io/diploma-thesis/original/index.html
[6] rfc2616
[7] https://banu.com/blog/2/how-to-use-epoll-a-complete-example-in-c/
[8] Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition)

Posted by Jiashun Zhu program

« 当执行kill -9 PID时系统发生了什么 如何用C/C++写一个基于事件的爬虫 »

Categories

- algorithms (2)
- brpc (1)
- computersystem (7)
- life (3)
- program (17)
- reading (3)
- thoughts (7)