

34 | 自己动手写高性能HTTP服务器（三）：TCP字节流处理和HTTP协议实现

2019-10-25 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 09:18 大小 8.52M



你好，我是盛延敏，这里是网络编程实战第 34 讲，欢迎回来。

这一讲，我们延续第 33 讲的话题，继续解析高性能网络编程框架的字节流处理部分，并为网络编程框架增加 HTTP 相关的功能，在此基础上完成 HTTP 高性能服务器的编写。

buffer 对象

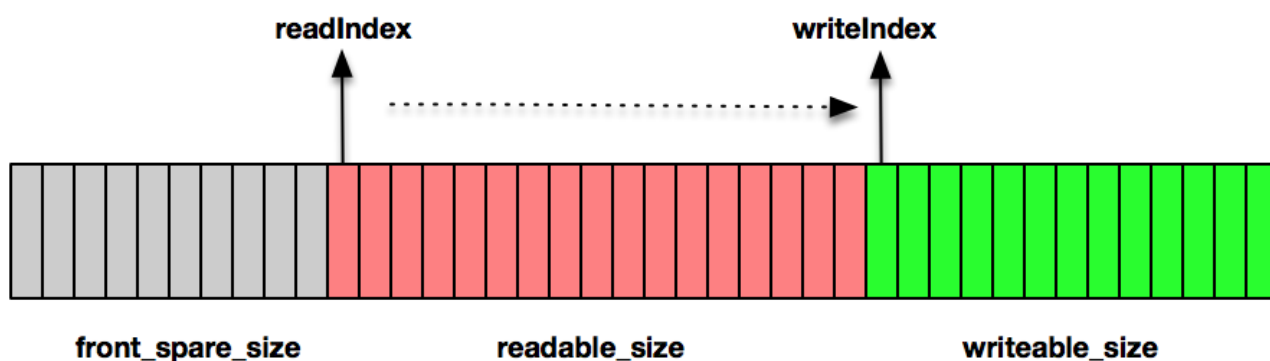
你肯定在各种语言、各种框架里面看到过不同的 buffer 对象，buffer，顾名思义，就是一个缓冲区对象，缓存了从套接字接收来的数据以及需要发往套接字的数据。

如果是从套接字接收来的数据，事件处理回调函数在不断地往 buffer 对象增加数据，同时，应用程序需要不断把 buffer 对象中的数据处理掉，这样，buffer 对象才可以空出新的位置容纳更多的数据。

如果是发往套接字的数据，应用程序不断地往 buffer 对象增加数据，同时，事件处理回调函数不断调用套接字上的发送函数将数据发送出去，减少 buffer 对象中的写入数据。


可见，buffer 对象是同时可以作为输入缓冲（input buffer）和输出缓冲（output buffer）两个方向使用的，只不过，在两种情形下，写入和读出的对象是有区别的。

我在文稿中给出了一张图，描述了 buffer 对象的设计。



下面是 buffer 对象的数据结构。

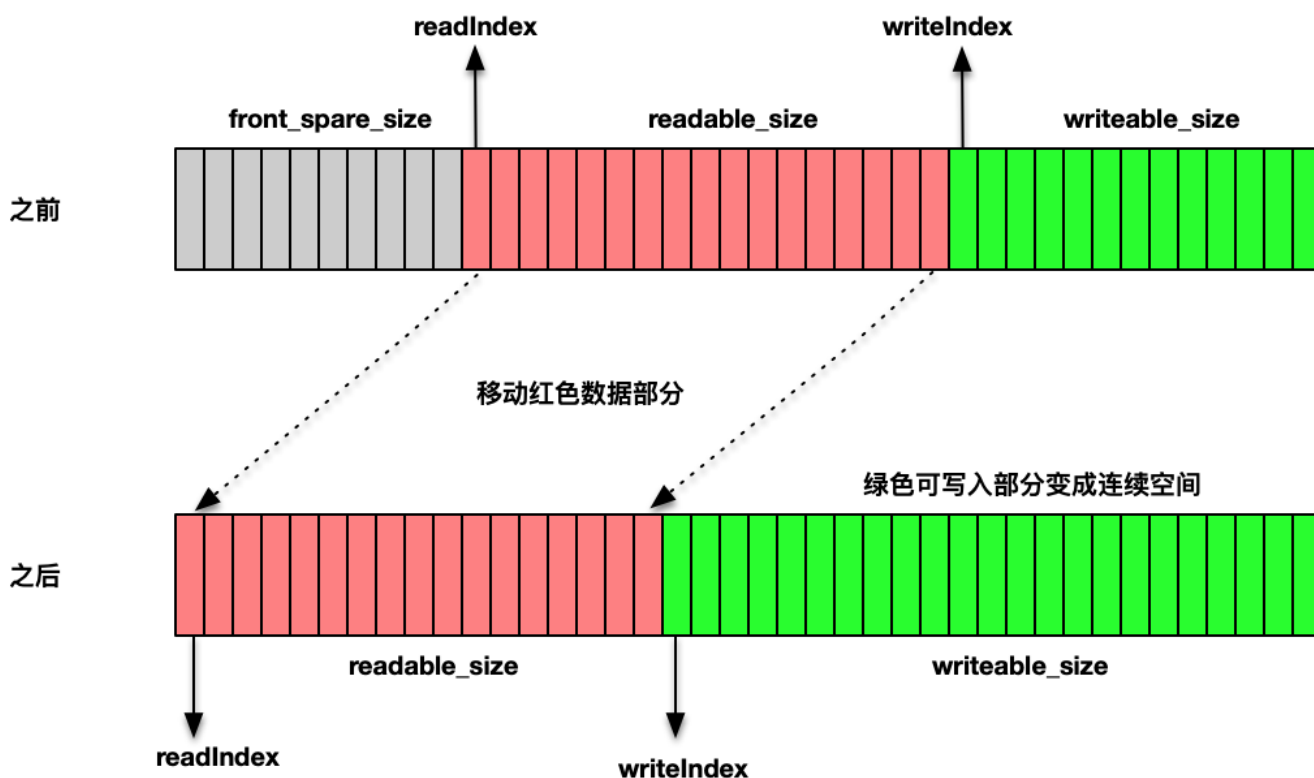
```
1 // 数据缓冲区
2 struct buffer {
3     char *data;           // 实际缓冲
4     int readIndex;        // 缓冲读取位置
5     int writeIndex;       // 缓冲写入位置
6     int total_size;       // 总大小
7 };
```

 复制代码

buffer 对象中的 writeIndex 标识了当前可以写入的位置；readIndex 标识了当前可以读出的数据位置，图中红色部分从 readIndex 到 writeIndex 的区域是需要读出数据的部分，而绿色部分从 writeIndex 到缓存的最尾端则是可以写出的部分。

随着时间的推移，当 readIndex 和 writeIndex 越来越靠近缓冲的尾端时，前面部分的 front_space_size 区域变得会很大，而这个区域的数据已经是旧数据，在这个时候，就需要调整一下整个 buffer 对象的结构，把红色部分往左侧移动，与此同时，绿色部分也会往左侧移动，整个缓冲区的可写部分就会变多了。

make_room 函数就是起这个作用的，如果右边绿色的连续空间不足以容纳新的数据，而最左边灰色部分加上右边绿色部分一起可以容纳下新数据，就会触发这样的移动拷贝，最终红色部分占据了最左边，绿色部分占据了右边，右边绿色的部分成为一个连续的可吸写入空间，就可以容纳下新的数据。下面的一张图解释了这个过程。



下面是 make_room 的具体实现。

复制代码

```
1 void make_room(struct buffer *buffer, int size) {
2     if (buffer_writeable_size(buffer) >= size) {
3         return;
4     }
5     // 如果 front_spare 和 writeable 的大小加起来可以容纳数据，则把可读数据往前面拷贝
6     if (buffer_front_spare_size(buffer) + buffer_writeable_size(buffer) >= size) {
7         int readable = buffer_readable_size(buffer);
8         int i;
9         for (i = 0; i < readable; i++) {
10             memcpy(buffer->data + i, buffer->data + buffer->readIndex + i, 1);
11         }
12     }
13 }
```

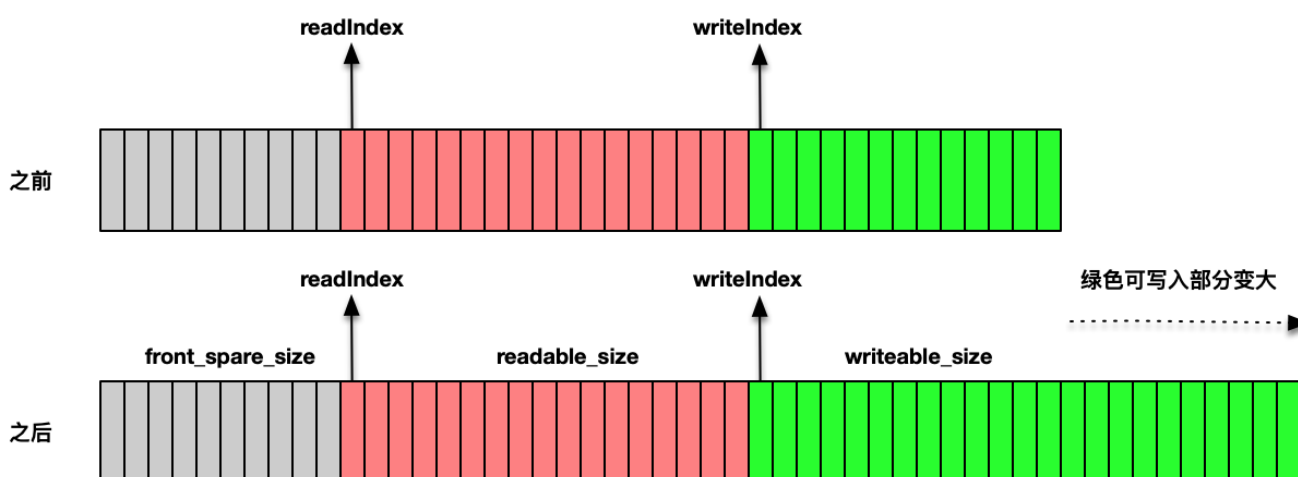
```

12     buffer->readIndex = 0;
13     buffer->writeIndex = readable;
14 } else {
15     // 扩大缓冲区
16     void *tmp = realloc(buffer->data, buffer->total_size + size);
17     if (tmp == NULL) {
18         return;
19     }
20     buffer->data = tmp;
21     buffer->total_size += size;
22 }
23 }

```

当然，如果红色部分占据过大，可写部分不够，会触发缓冲区的扩大操作。这里我通过调用 `realloc` 函数来完成缓冲区的扩容。

下面这张图对此做了解释。



套接字接收数据处理

套接字接收数据是在 `tcp_connection.c` 中的 `handle_read` 来完成的。在这个函数里，通过调用 `buffer_socket_read` 函数接收来自套接字的数据流，并将其缓冲到 `buffer` 对象中。之后你可以看到，我们将 `buffer` 对象和 `tcp_connection` 对象传递给应用程序真正的处理函数 `messageCallBack` 来进行报文的解析工作。这部分的样例在 HTTP 报文解析中会展开。

 复制代码

```

1 int handle_read(void *data) {
2

```

```

3     struct tcp_connection *tcpConnection = (struct tcp_connection *) data;
4     struct buffer *input_buffer = tcpConnection->input_buffer;
5     struct channel *channel = tcpConnection->channel;
6
7     if (buffer_socket_read(input_buffer, channel->fd) > 0) {
8         // 应用程序真正读取 Buffer 里的数据
9         if (tcpConnection->messageCallBack != NULL) {
10             tcpConnection->messageCallBack(input_buffer, tcpConnection);
11         }
12     } else {
13         handle_connection_closed(tcpConnection);
14     }
15 }

```

在 `buffer_socket_read` 函数里，调用 `readv` 往两个缓冲区写入数据，一个是 `buffer` 对象，另外一个是因为这里的 `additional_buffer`，之所以这样做，是担心 `buffer` 对象没办法容纳下来自套接字的数据流，而且也没有办法触发 `buffer` 对象的扩容操作。通过使用额外的缓冲，一旦判断出从套接字读取的数据超过了 `buffer` 对象里的实际最大可写大小，就可以触发 `buffer` 对象的扩容操作，这里 `buffer_append` 函数会调用前面介绍的 `make_room` 函数，完成 `buffer` 对象的扩容。

 复制代码


```

1 int buffer_socket_read(struct buffer *buffer, int fd) {
2     char additional_buffer[INIT_BUFFER_SIZE];
3     struct iovec vec[2];
4     int max_writable = buffer_writable_size(buffer);
5     vec[0].iov_base = buffer->data + buffer->writeIndex;
6     vec[0].iov_len = max_writable;
7     vec[1].iov_base = additional_buffer;
8     vec[1].iov_len = sizeof(additional_buffer);
9     int result = readv(fd, vec, 2);
10    if (result < 0) {
11        return -1;
12    } else if (result <= max_writable) {
13        buffer->writeIndex += result;
14    } else {
15        buffer->writeIndex = buffer->total_size;
16        buffer_append(buffer, additional_buffer, result - max_writable);
17    }
18    return result;
19 }

```


套接字发送数据处理

当应用程序需要往套接字发送数据时，即完成了 read-decode-compute-encode 过程后，通过往 buffer 对象里写入 encode 以后的数据，调用 tcp_connection_send_buffer，将 buffer 里的数据通过套接字缓冲区发送出去。

 复制代码

```
1 int tcp_connection_send_buffer(struct tcp_connection *tcpConnection, struct bu
2     int size = buffer_readable_size(buffer);
3     int result = tcp_connection_send_data(tcpConnection, buffer->data + buffer
4     buffer->readIndex += size;
5     return result;
6 }
```

如果发现当前 channel 没有注册 WRITE 事件，并且当前 tcp_connection 对应的发送缓冲无数据需要发送，就直接调用 write 函数将数据发送出去。如果这一次发送不完，就将剩余需要发送的数据拷贝到当前 tcp_connection 对应的发送缓冲区中，并向 event_loop 注册 WRITE 事件。这样数据就由框架接管，应用程序释放这部分数据。

 复制代码

```
1 // 应用层调用入口
2 int tcp_connection_send_data(struct tcp_connection *tcpConnection, void *data,
3     size_t nwrited = 0;
4     size_t nleft = size;
5     int fault = 0;
6
7     struct channel *channel = tcpConnection->channel;
8     struct buffer *output_buffer = tcpConnection->output_buffer;
9
10    // 先往套接字尝试发送数据
11    if (!channel_write_event_registered(channel) && buffer_readable_size(output
12        nwrited = write(channel->fd, data, size);
13        if (nwrited >= 0) {
14            nleft = nleft - nwrited;
15        } else {
16            nwrited = 0;
17            if (errno != EWOULDBLOCK) {
18                if (errno == EPIPE || errno == ECONNRESET) {
19                    fault = 1;
20                }
21            }
22        }
23    }
24
25    if (!fault && nleft > 0) {
26        // 拷贝到 Buffer 中, Buffer 的数据由框架接管
```

```


27     buffer_append(output_buffer, data + nwrited, nleft);
28     if (!channel_write_event_registered(channel)) {
29         channel_write_event_add(channel);
30     }
31 }
32
33 return nwrited;
34 }

```

HTTP 协议实现

下面，我们在 TCP 的基础上，加入 HTTP 的功能。

为此，我们首先定义了一个 http_server 结构，这个 http_server 本质上就是一个 TCPserver，只不过暴露给应用程序的回调函数更为简单，只需要看到 http_request 和 http_response 结构。

 复制代码

```

1 typedef int (*request_callback)(struct http_request *httpRequest, struct http_
2
3 struct http_server {
4     struct TCPserver *tcpServer;
5     request_callback requestCallback;
6 };

```

在 http_server 里面，重点是需要完成报文的解析，将解析的报文转化为 http_request 对象，这件事情是通过 http_onMessage 回调函数来完成的。在 http_onMessage 函数里，调用的是 parse_http_request 完成报文解析。

 复制代码

```

1 // buffer 是框架构建好的，并且已经收到部分数据的情况下
2 // 注意这里可能没有收到全部数据，所以要处理数据不够的情形
3 int http_onMessage(struct buffer *input, struct tcp_connection *tcpConnection)
4     yolanda_msgx("get message from tcp connection %s", tcpConnection->name);
5
6     struct http_request *httpRequest = (struct http_request *) tcpConnection->
7     struct http_server *httpServer = (struct http_server *) tcpConnection->dat
8
9     if (parse_http_request(input, httpRequest) == 0) {
10         char *error_response = "HTTP/1.1 400 Bad Request\r\n\r\n";
11         tcp_connection_send_data(tcpConnection, error_response, sizeof(error_r
12         tcp_connection_shutdown(tcpConnection);

```



```

13     }
14
15     // 处理完了所有的 request 数据，接下来进行编码和发送
16     if (http_request_current_state(httpRequest) == REQUEST_DONE) {
17         struct http_response *httpResponse = http_response_new();
18
19         //httpServer 暴露的 requestCallback 回调
20         if (httpServer->requestCallback != NULL) {
21             httpServer->requestCallback(httpRequest, httpResponse);
22         }
23
24         // 将 httpResponse 发送到套接字发送缓冲区中
25         struct buffer *buffer = buffer_new();
26         http_response_encode_buffer(httpResponse, buffer);
27         tcp_connection_send_buffer(tcpConnection, buffer);
28
29         if (http_request_close_connection(httpRequest)) {
30             tcp_connection_shutdown(tcpConnection);
31             http_request_reset(httpRequest);
32         }
33     }
34 }

```

还记得🔗第 16 讲中讲到的 HTTP 协议吗？我们从 16 讲得知，HTTP 通过设置回车符、换行符做为 HTTP 报文协议的边界。

请求方法	空格	URL	空格	协议版本	回车	换行
头部字段名	:	值	回车	换行		
头部字段名	:	值	回车	换行		
回车	换行					
请求正文						


parse_http_request 的思路就是寻找报文的边界，同时记录下当前解析工作所处的状态。根据解析工作的前后顺序，把报文解析的工作分成 REQUEST_STATUS、REQUEST_HEADERS、REQUEST_BODY 和 REQUEST_DONE 四个阶段，每个阶段解析的方法各有不同。

在解析状态行时，先通过定位 CRLF 回车换行符的位置来圈定状态行，进入状态行解析时，再次通过查找空格字符来作为分隔边界。

在解析头部设置时，也是先通过定位 CRLF 回车换行符的位置来圈定一组 key-value 对，再通过查找冒号字符来作为分隔边界。

最后，如果没有找到冒号字符，说明解析头部的工作完成。

parse_http_request 函数完成了 HTTP 报文解析的四个阶段：

 复制代码

```
1 int parse_http_request(struct buffer *input, struct http_request *httpRequest)
2     int ok = 1;
3     while (httpRequest->current_state != REQUEST_DONE) {
4         if (httpRequest->current_state == REQUEST_STATUS) {
5             char *crlf = buffer_find_CRLF(input);
6             if (crlf) {
7                 int request_line_size = process_status_line(input->data + input->readIndex, crlf);
8                 if (request_line_size) {
9                     input->readIndex += request_line_size; // request line size
10                    input->readIndex += 2; //CRLF size
11                    httpRequest->current_state = REQUEST_HEADERS;
12                }
13            }
14        } else if (httpRequest->current_state == REQUEST_HEADERS) {
15            char *crlf = buffer_find_CRLF(input);
16            if (crlf) {
17                /**
18                 *    <start>-----<colon>:-----<crlf>
19                 */
20                char *start = input->data + input->readIndex;
21                int request_line_size = crlf - start;
22                char *colon = memmem(start, request_line_size, ":", 2);
23                if (colon != NULL) {
24                    char *key = malloc(colon - start + 1);
25                    strncpy(key, start, colon - start);
26                    key[colon - start] = '\0';
27                    char *value = malloc(crlf - colon - 2 + 1);
28                    strncpy(value, colon + 1, crlf - colon - 2);
29                    value[crlf - colon - 2] = '\0';
30
31                    http_request_add_header(httpRequest, key, value);
32
33                    input->readIndex += request_line_size; //request line size
34                    input->readIndex += 2; //CRLF size
35                } else {
```


```

36         // 读到这里说明：没找到，就说明这个是最后一行
37         input->readIndex += 2; //CRLF size
38         httpRequest->current_state = REQUEST_DONE;
39     }
40 }
41 }
42 }
43 return ok;
44 }

```

处理完了所有的 request 数据，接下来进行编码和发送的工作。为此，创建了一个 http_response 对象，并调用了应用程序提供的编码函数 requestCallback，接下来，创建了一个 buffer 对象，函数 http_response_encode_buffer 用来将 http_response 中的数据，根据 HTTP 协议转换为对应的字节流。

可以看到，http_response_encode_buffer 设置了如 Content-Length 等 http_response 头部，以及 http_response 的 body 部分数据。

 复制代码

```

1 void http_response_encode_buffer(struct http_response *httpResponse, struct bu
2     char buf[32];
3     snprintf(buf, sizeof buf, "HTTP/1.1 %d ", httpResponse->statusCode);
4     buffer_append_string(output, buf);
5     buffer_append_string(output, httpResponse->statusMessage);
6     buffer_append_string(output, "\r\n");
7
8     if (httpResponse->keep_connected) {
9         buffer_append_string(output, "Connection: close\r\n");
10    } else {
11        snprintf(buf, sizeof buf, "Content-Length: %zd\r\n", strlen(httpRespon:
12        buffer_append_string(output, buf);
13        buffer_append_string(output, "Connection: Keep-Alive\r\n");
14    }
15
16    if (httpResponse->response_headers != NULL && httpResponse->response_heade
17        for (int i = 0; i < httpResponse->response_headers_number; i++) {
18            buffer_append_string(output, httpResponse->response_headers[i].key
19            buffer_append_string(output, ": ");
20            buffer_append_string(output, httpResponse->response_headers[i].valu
21            buffer_append_string(output, "\r\n");
22        }
23    }
24
25    buffer_append_string(output, "\r\n");
26    buffer_append_string(output, httpResponse->body);

```

完整的 HTTP 服务器例子

现在，编写一个 HTTP 服务器例子就变得非常简单。你可以在文稿中，以及 GitHub 仓库中看到这个例子。

在这个例子中，最主要的部分是 onRequest callback 函数，这里，onRequest 方法已经在 parse_http_request 之后，可以根据不同的 http_request 的信息，进行计算和处理。例子程序里的逻辑非常简单，根据 http request 的 URL path，返回了不同的 http_response 类型。比如，当请求为根目录时，返回的是 200 和 HTML 格式。

[复制代码](#)

```
1 #include <lib/acceptor.h>
2 #include <lib/http_server.h>
3 #include "lib/common.h"
4 #include "lib/event_loop.h"
5
6 // 数据读到 buffer 之后的 callback
7 int onRequest(struct http_request *httpRequest, struct http_response *httpResponse) {
8     char *url = httpRequest->url;
9     char *question = memmem(url, strlen(url), "?", 1);
10    char *path = NULL;
11    if (question != NULL) {
12        path = malloc(question - url);
13        strncpy(path, url, question - url);
14    } else {
15        path = malloc(strlen(url));
16        strncpy(path, url, strlen(url));
17    }
18
19    if (strcmp(path, "/") == 0) {
20        httpResponse->statusCode = OK;
21        httpResponse->statusMessage = "OK";
22        httpResponse->contentType = "text/html";
23        httpResponse->body = "<html><head><title>This is network programming</";
24    } else if (strcmp(path, "/network") == 0) {
25        httpResponse->statusCode = OK;
26        httpResponse->statusMessage = "OK";
27        httpResponse->contentType = "text/plain";
28        httpResponse->body = "hello, network programming";
29    } else {
30        httpResponse->statusCode = NotFound;
31        httpResponse->statusMessage = "Not Found";
32        httpResponse->keep_connected = 1;
```

```

33     }
34
35     return 0;
36 }
37
38
39 int main(int c, char **v) {
40     // 主线程 event_loop
41     struct event_loop *eventLoop = event_loop_init();
42
43     // 初始 tcp_server, 可以指定线程数目, 如果线程是 0, 就是在这个线程里 acceptor+i/o; !
44     //tcp_server 自己带一个 event_loop
45     struct http_server *httpServer = http_server_new(eventLoop, SERV_PORT, onR
46     http_server_start(httpServer);
47
48     // main thread for acceptor
49     event_loop_run(eventLoop);
50 }

```

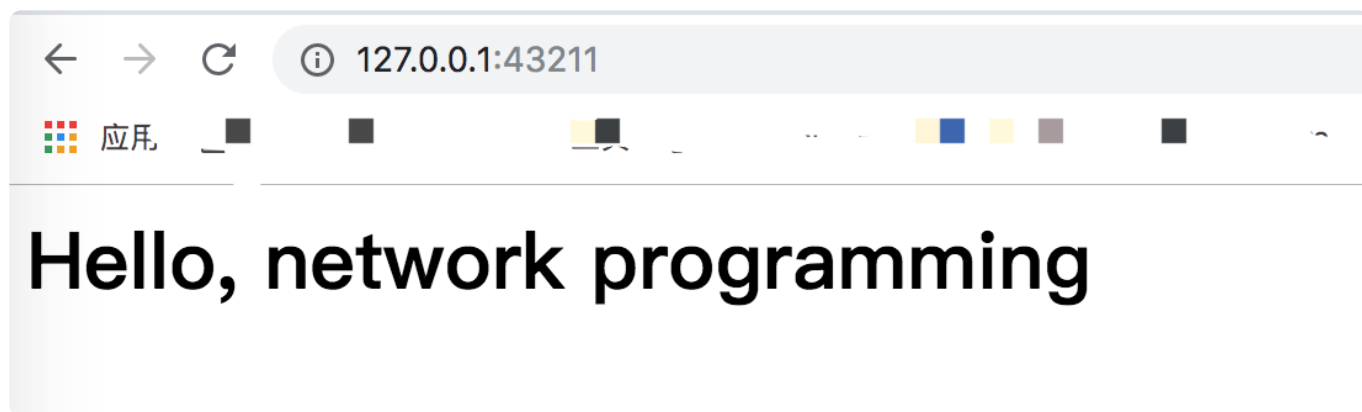
运行这个程序之后，我们可以通过浏览器和 curl 命令来访问它。你可以同时开启多个浏览器和 curl 命令，这也证明了我们的程序是可以满足高并发需求的。

 复制代码

```

1 $curl -v http://127.0.0.1:43211/
2 * Trying 127.0.0.1...
3 * TCP_NODELAY set
4 * Connected to 127.0.0.1 (127.0.0.1) port 43211 (#0)
5 > GET / HTTP/1.1
6 > Host: 127.0.0.1:43211
7 > User-Agent: curl/7.54.0
8 > Accept: */*
9 >
10 < HTTP/1.1 200 OK
11 < Content-Length: 116
12 < Connection: Keep-Alive
13 <
14 * Connection #0 to host 127.0.0.1 left intact
15 <html><head><title>This is network programming</title></head><body><h1>Hello, I

```



总结

这一讲我们主要讲述了整个编程框架的字节流处理能力，引入了 buffer 对象，并在此基础上通过增加 HTTP 的特性，包括 `http_server`、`http_request`、`http_response`，完成了 HTTP 高性能服务器的编写。实例程序利用框架提供的能力，编写了一个简单的 HTTP 服务器程序。

思考题

和往常一样，给大家布置两道思考题：

第一道，你可以试着在 HTTP 服务器中增加 MIME 的处理能力，当用户请求 `/photo` 路径时，返回一张图片。

第二道，在我们的开发中，已经有很多面向对象的设计，你可以仔细研读代码，说说你对这部分的理解。

欢迎你在评论区写下你的思考，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

网络编程实战

从底层到实战，深度解析网络编程

盛延敏

前大众点评云平台首席架构师



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 33 | 自己动手写高性能HTTP服务器（二）：I/O模型和多线程模型实现

下一篇 35 | 答疑：编写高性能网络编程框架时，都需要注意哪些问题？

精选留言 (5)

写留言



tt

2019-10-28

嗯，对于第二个问题，因为我是从C++语言开始进入编程的，老师的C代码确实很多都是面向对象的。

很多模块，比如tcp_connection，对应的头文件中声明的函数，第一个参数都是tcp_connection指针，这就相当于this指针。而相应的以"_new"结尾的函数就相当于C++中的构...

展开

作者回复: 是的，面向对象写起来简洁许多





MoonGod
2019-10-26

如果发现当前 channel 没有注册 WRITE 事件，并且当前 tcp_connection 对应的发送缓冲无数据需要发送，就直接调用 write 函数将数据发送出去。

老师好，这里没有理解，为啥不能做成无论有没有write事件都统一往发送缓冲区写呢，之后如果没有write事件，就再注册一个就好了不是？

展开 ▾

作者回复: 非常好的问题，我发到统一答疑部分了。感谢~



Steiner
2019-10-25

实在学不动了，我想抄袭老师的创意自己写一个框架，然后慢慢该吧
(◦~'◦)

展开 ▾

作者回复: 很好奇学不动的意思是啥



我来也
2019-10-25

老师的c代码看上去是一种享受。
逻辑很清晰，很佩服函数命名。

以前我们缓冲区是用的循环，避免频繁的挪动数据，不过要处理好溢出的情况。

...

展开 ▾

编辑回复: 您好，文章已进行改正，谢谢反馈。



传说中的成大大
2019-10-25

第二个问题 才是把我考到了 我感觉现在我对设计模式的理解并不深,但是我现在感受特深的一点就是单一职责原理 buffer类才套接字的处理 tcpconnect应用层面的处理,而且最近

在工作中我也是尝试着画流程图 把每个功能进行细分 分到一个流程分支里面只处理一个逻辑

展开 ✓

