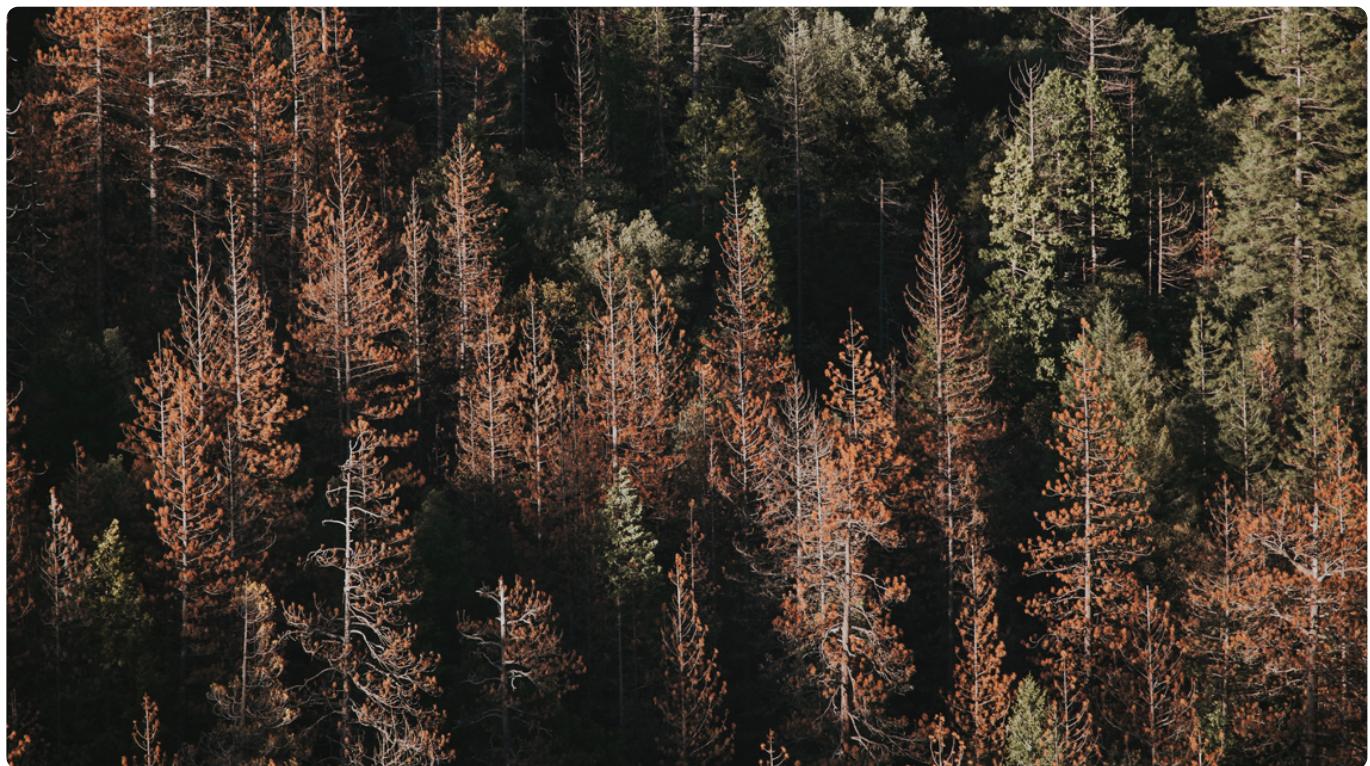


[下载APP](#)

35 | 答疑：编写高性能网络编程框架时，都需要注意哪些问题？

2019-10-28 盛延敏

网络编程实战

[进入课程 >](#)

讲述：冯永吉

时长 00:39 大小 630.12K



你好，我是盛延敏，这里是网络编程实战的第 35 讲，欢迎回来。

这一篇文章是实战篇的答疑部分，也是本系列的最后一篇文章。非常感谢你的积极评论与留言，让每一篇文章的留言区都成为学习互动的好地方。在今天的内容里，我将针对评论区的问题做一次集中回答，希望能帮助你解决前面碰到的一些问题。

有关这部分内容，我将采用 Q&A 的形式来展开。

为什么在发送数据时，会先尝试通过 socket 直接发送，再由框架接管呢？

这个问题具体描述是下面这样的。

当应用程序需要发送数据时，比如下面这段，在完成数据读取和回应的编码之后，会调用 `tcp_connection_send_buffer` 方法发送数据。

 复制代码

```
1 // 数据读到 buffer 之后的 callback
2 int onMessage(struct buffer *input, struct tcp_connection *tcpConnection) {
3     printf("get message from tcp connection %s\n", tcpConnection->name);
4     printf("%s", input->data);
5
6     struct buffer *output = buffer_new();
7     int size = buffer_readable_size(input);
8     for (int i = 0; i < size; i++) {
9         buffer_append_char(output, rot13_char(buffer_read_char(input)));
10    }
11    tcp_connection_send_buffer(tcpConnection, output);
12    return 0;
13 }
```

而 `tcp_connection_send_buffer` 方法则会调用 `tcp_connection_send_data` 来发送数据：

 复制代码

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

在 `tcp_connection_send_data` 中，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。

 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 nwritten = 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_is_enabled(channel) && buffer_readable_size(output_
12 nwritten = write(channel->fd, data, size);
13 if (nwritten >= 0) {
14     nleft = nleft - nwritten;
15 } else {
16     nwritten = 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 + nwritten, nleft);
28     if (!channel_write_event_is_enabled(channel)) {
29         channel_write_event_enable(channel);
30     }
31 }
32
33 return nwritten;
34 }

```

这里有同学不是很理解，为啥不能做成无论有没有 WRITE 事件都统一往发送缓冲区写，再把 WRITE 事件注册到 event_loop 中呢？

这个问题问得非常好。我觉得有必要展开讲讲。

如果用一句话来总结的话，这是为了发送效率。

我们来分析一下，应用层读取数据，进行编码，之后的这个 buffer 对象是应用层创建的，数据也在应用层这个 buffer 对象上。你可以理解，tcp_connection_send_data 里面的 data 数据其实是应用层缓冲的，而不是我们 tcp_connection 这个对象里面的 buffer。

如果我们跳过直接往套接字发送这一段，而是把数据交给我们的 tcp_connection 对应的 output_buffer，这里有一个数据拷贝的过程，它发生在 buffer_append 里面。

[复制代码](#)

```
1 int buffer_append(struct buffer *buffer, void *data, int size) {  
2     if (data != NULL) {  
3         make_room(buffer, size);  
4         // 拷贝数据到可写空间中  
5         memcpy(buffer->data + buffer->writeIndex, data, size);  
6         buffer->writeIndex += size;  
7     }  
8 }
```

但是，如果增加了一段判断来直接往套接字发送，其实就跳过了这段拷贝，直接把数据发往到了套接字发生缓冲区。

[复制代码](#)

```
1 // 先往套接字尝试发送数据  
2 if (!channel_write_event_is_enabled(channel) && buffer_readable_size(output_bu  
3     nwrited = write(channel->fd, data, size)  
4     ...
```

在绝大部分场景下，这种处理方式已经满足数据发送的需要了，不再需要把数据拷贝到 `tcp_connection` 对象中的 `output_buffer` 中。

如果不满足直接往套接字发送的条件，比如已经注册了回调事件，或者 `output_buffer` 里面有数据需要发送，那么就把数据拷贝到 `output_buffer` 中，让 `event_loop` 的回调不断地驱动 `handle_write` 将数据从 `output_buffer` 发往套接字缓冲区中。

[复制代码](#)

```
1 // 发送缓冲区可以往外写  
2 // 把 channel 对应的 output_buffer 不断往外发送  
3 int handle_write(void *data) {  
4     struct tcp_connection *tcpConnection = (struct tcp_connection *) data;  
5     struct event_loop *eventLoop = tcpConnection->eventLoop;  
6     assertInSameThread(eventLoop);  
7  
8     struct buffer *output_buffer = tcpConnection->output_buffer;  
9     struct channel *channel = tcpConnection->channel;  
10  
11    ssize_t nwrited = write(channel->fd, output_buffer->data + output_buffer->  
12    if (nwrited > 0) {  
13        // 已读 nwrited 字节  
14        output_buffer->readIndex += nwrited;  
15        // 如果数据完全发送出去，就不需要继续了
```

```
16     if (buffer_readable_size(output_buffer) == 0) {
17         channel_write_event_disable(channel);
18     }
19     // 回调 writeCompletedCallBack
20     if (tcpConnection->writeCompletedCallBack != NULL) {
21         tcpConnection->writeCompletedCallBack(tcpConnection);
22     }
23 } else {
24     yolanda_msgx("handle_write for tcp connection %s", tcpConnection->name);
25 }
26
27 }
```

你可以这样想象，在一个非常高效的处理条件下，你需要发送什么，都直接发送给了套接字缓冲区；而当网络条件变差，处理效率变慢，或者待发送的数据极大，一次发送不可能完成的时候，这部分数据被框架缓冲到 `tcp_connection` 的发送缓冲区对象 `output_buffer` 中，由事件分发机制来负责把这部分数据发送给套接字缓冲区。

关于回调函数的设计

在 `epoll-server-multithreads.c` 里面定义了很多回调函数，比如 `onMessage`, `onConnectionCompleted` 等，这些回调函数被用于创建一个 `TCP` Server，但是在 `tcp_connection` 对照中，又实现了 `handle_read` `handle_write` 等事件的回调，似乎有两层回调，为什么要这样封装两层回调呢？

这里如果说回调函数，确实有两个不同层次的回调函数。

第一个层次是框架定义的，对连接的生命周期管理的回调。包括连接建立完成后的回调、报文读取并接收到 `output` 缓冲区之后的回调、报文发送到套接字缓冲区之后的回调，以及连接关闭时的回调。分别是 `connectionCompletedCallBack`、`messageCallBack`、`writeCompletedCallBack`，以及 `connectionClosedCallBack`。

 复制代码

```
1 struct tcp_connection {
2     struct event_loop *eventLoop;
3     struct channel *channel;
4     char *name;
5     struct buffer *input_buffer;    // 接收缓冲区
6     struct buffer *output_buffer;  // 发送缓冲区
7
8     connection_completed_call_back connectionCompletedCallBack;
```

```
9     message_call_back messageCallBack;
10    write_completed_call_back writeCompletedCallBack;
11    connection_closed_call_back connectionClosedCallBack;
12
13    void * data; //for callback use: http_server
14    void * request; // for callback use
15    void * response; // for callback use
16 };
```

为什么要定义这四个回调函数呢？

因为框架需要提供给应用程序和框架的编程接口，我把它总结为编程连接点，或者叫做 program-hook-point。就像是设计了一个抽象类，这个抽象类代表了框架给你提供的一个编程入口，你可以继承这个抽象类，完成一些方法的填充，这些方法和框架类一起工作，就可以表现出一定符合逻辑的行为。

比如我们定义一个抽象类 People，这个类的其他属性，包括它的创建和管理都可以交给框架来完成，但是你需要完成两个函数，一个是 on_sad，这个人悲伤的时候干什么；另一个是 on_happy，这个人高兴的时候干什么。

 复制代码

```
1 abstract class People{
2     void on_sad();
3
4     void on_happy();
5 }
```

这样，我们可以试着把 tcp_connection 改成这样：

 复制代码

```
1 abstract class TCP_connection{
2     void on_connection_completed();
3
4     void on_message();
5
6     void on_write_completed();
7
8     void on_connectin_closed();
9 }
```

这个层次的回调，更像是一层框架和应用程序约定的接口，接口实现由应用程序来完成，框架负责在合适的时候调用这些预定义好的接口，回调的意思体现在“框架会调用预定好的接口实现”。

比如，当连接建立成功，一个新的 connection 创建出来，connectionCompletedCallBack 函数会被回调：

 复制代码

```
1 struct tcp_connection *
2 tcp_connection_new(int connected_fd, struct event_loop *eventLoop,
3 connection_completed_call_back connectionCompletedCallBack,
4 connection_closed_call_back connectionClosedCallBack,
5 message_call_back messageCallBack,
6 write_completed_call_back writeCompletedCallBack) {
7     ...
8     // add event read for the new connection
9     struct channel *channel1 = channel_new(connected_fd, EVENT_READ, handle_read,
10     tcpConnection->channel = channel1;
11
12     //connectionCompletedCallBack callback
13     if (tcpConnection->connectionCompletedCallBack != NULL) {
14         tcpConnection->connectionCompletedCallBack(tcpConnection);
15     }
16
17     ...
18 }
```

第二个层次的回调，是基于 epoll、poll 事件分发机制的回调。通过注册一定的读、写事件，在实际事件发生时，由事件分发机制保证对应的事件回调函数被及时调用，完成基于事件机制的网络 I/O 处理。

在每个连接建立之后，创建一个对应的 channel 对象，并为这个 channel 对象赋予了读、写回调函数：

 复制代码

```
1 // add event read for the new connection
2 struct channel *channel1 = channel_new(connected_fd, EVENT_READ, handle_read, |
```

`handle_read` 函数，对应用程序屏蔽了套接字的读操作，把数据缓冲到 `tcp_connection` 的 `input_buffer` 中，而且，它还起到了编程连接点和框架的耦合器的作用，这里分别调用了 `messageCallBack` 和 `connectionClosedCallBack` 函数，完成了应用程序编写部分代码在框架的“代入”。

 复制代码

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

`handle_write` 函数则负责把 `tcp_connection` 对象里的 `output_buffer` 源源不断地送往套接字发送缓冲区。

 复制代码

```
1 // 发送缓冲区可以往外写
2 // 把 channel 对应的 output_buffer 不断往外发送
3 int handle_write(void *data) {
4     struct tcp_connection *tcpConnection = (struct tcp_connection *) data;
5     struct event_loop *eventLoop = tcpConnection->eventLoop;
6     assertInSameThread(eventLoop);
7
8     struct buffer *output_buffer = tcpConnection->output_buffer;
9     struct channel *channel = tcpConnection->channel;
10
11    ssize_t nwritten = write(channel->fd, output_buffer->data + output_buffer->
12    if (nwritten > 0) {
13        // 已读 nwritten 字节
14        output_buffer->readIndex += nwritten;
15        // 如果数据完全发送出去，就不需要继续了
16        if (buffer_readable_size(output_buffer) == 0) {
17            channel_write_event_disable(channel);
18        }
19        // 回调 writeCompletedCallBack
20        if (tcpConnection->writeCompletedCallBack != NULL) {
```

```
21         tcpConnection->writeCompletedCallBack(tcpConnection);
22     }
23 } else {
24     yolanda_msgx("handle_write for tcp connection %s", tcpConnection->name);
25 }
26
27 }
```

tcp_connection 对象设计的想法是什么，和 channel 有什么联系和区别？

tcp_connection 对象似乎和 channel 对象有着非常紧密的联系，为什么要单独设计一个 tcp_connection 呢？

在文稿中，我也提到了，开始的时候我并不打算设计一个 tcp_connection 对象的，后来我才发现非常有必要存在一个 tcp_connection 对象。

第一，我需要在暴露给应用程序的 onMessage, onConnectionCompleted 等回调函数里，传递一个有用的数据结构，这个数据结构必须有一定的现实语义，可以携带一定的信息，比如套接字、缓冲区等，而 channel 对象过于单薄，和连接的语义相去甚远。

第二，这个 channel 对象是抽象的，比如 acceptor，比如文稿里提到的 socketpair 等，它们都是一个 channel，只要能引起事件的发生和传递，都是一个 channel，基于这一点，我也觉得最好把 channel 作为一个内部实现的细节，不要通过回调函数暴露给应用程序。

第三，在后面实现 HTTP 的过程中，我发现需要在上下文中保存 http_request 和 http_response 数据，而这个部分数据放在 channel 中是非常不合适的，所以才有了最后的 tcp_connection 对象。

 复制代码

```
1 struct tcp_connection {
2     struct event_loop *eventLoop;
3     struct channel *channel;
4     char *name;
5     struct buffer *input_buffer;    // 接收缓冲区
6     struct buffer *output_buffer;  // 发送缓冲区
7
8     connection_completed_call_back connectionCompletedCallBack;
```

```
9     message_call_back messageCallBack;
10    write_completed_call_back writeCompletedCallBack;
11    connection_closed_call_back connectionClosedCallBack;
12
13    void * data; //for callback use: http_server
14    void * request; // for callback use
15    void * response; // for callback use
16};
```

简单总结下来就是，每个 `tcp_connection` 对象一定包含了一个 `channel` 对象，而 `channel` 对象未必是一个 `tcp_connection` 对象。

主线程等待子线程完成的同步锁问题

有人在加锁这里有个疑问，如果加锁的目的是让主线程等待子线程初始化 `event_loop`，那不加锁不是也可以达到这个目的吗？主线程 `while` 循环里面不断判断子线程的 `event_loop` 是否不为 `null` 不就可以了？为什么一定要加一把锁呢？

 复制代码

```
1 // 由主线程调用，初始化一个子线程，并且让子线程开始运行 event_loop
2 struct event_loop *event_loop_thread_start(struct event_loop_thread *eventLoopThread)
3     pthread_create(&eventLoopThread->thread_tid, NULL, &event_loop_thread_run,
4
5     assert(pthread_mutex_lock(&eventLoopThread->mutex) == 0);
6
7     while (eventLoopThread->eventLoop == NULL) {
8         assert(pthread_cond_wait(&eventLoopThread->cond, &eventLoopThread->mutex) == 0);
9     }
10    assert(pthread_mutex_unlock(&eventLoopThread->mutex) == 0);
11
12    yolanda_msgx("event loop thread started, %s", eventLoopThread->thread_name);
13    return eventLoopThread->eventLoop;
14}
```

要回答这个问题，就要解释多线程下共享变量竞争的问题。我们知道，一个共享变量在多个线程下同时作用，如果没有锁的控制，就会引起变量的不同步。这里的共享变量就是每个 `eventLoopThread` 的 `eventLoop` 对象。

这里如果不加锁，一直循环判断每个 `eventLoopThread` 的状态，会对 CPU 增加很大的消耗，如果使用锁 - 信号量的方式来加以解决，就变得很优雅，而且不会对 CPU 造成过

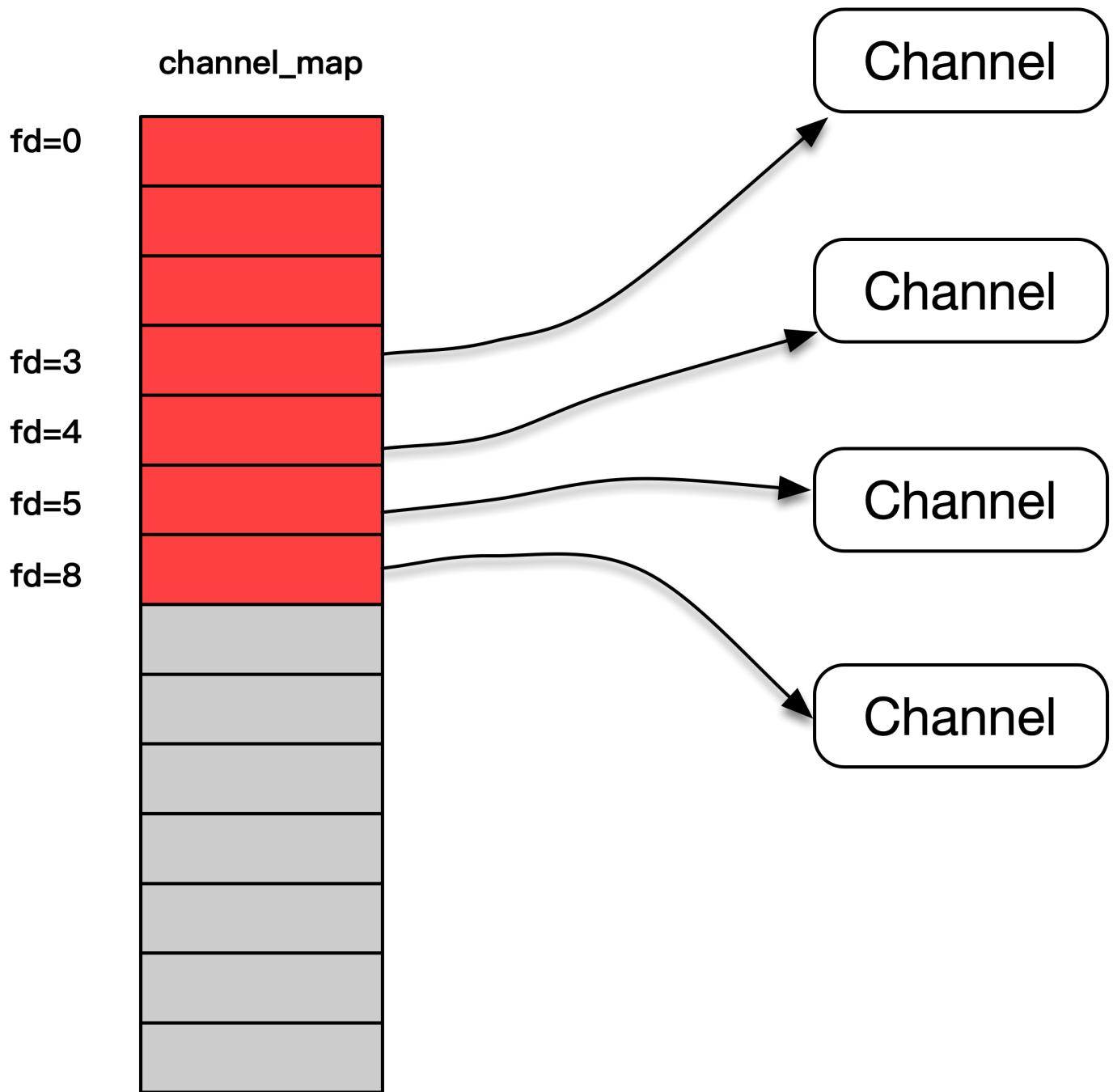
多的影响。

关于 channel_map 的设计，特别是内存方面的设计。

我们来详细介绍一下 channel_map。

channel_map 实际上是一个指针数组，这个数组里面的每个元素都是一个指针，指向了创建出的 channel 对象。如文稿中所说，我们用数据下标和套集字进行了映射，这样虽然有些元素是浪费了，比如 stdin, stdout, stderr 代表的套接字 0、1 和 2，但是总体效率是非常高的。

你在这里可以看到图中描绘了 channel_map 的设计。



而且，我们的 `channel_map` 还不会太占用内存，在最开始的时候，整个 `channel_map` 的指针数组大小为 0，当这个 `channel_map` 投入使用时，会根据实际使用的套接字的增长，按照 32、64、128 这样的速度成倍增长，这样既保证了实际的需求，也不会一下子占用太多的内存。

此外，当指针数组增长时，我们不会销毁原来的部分，而是使用 `realloc()` 把旧的内容搬过去，再使用 `memset()` 用来给新申请的内存初始化为 0 值，这样既高效也节省内存。

总结

以上就是实战篇中一些同学的疑问。

在这篇文章之后，我们的专栏就告一段落了，我希望这个专栏可以帮你梳理清楚高性能网络编程的方方面面，如果你能从中有所领悟，或者帮助你在面试中拿到好的结果，我会深感欣慰。

如果你觉得今天的答疑内容对你有所帮助，欢迎把它转发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 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

精选留言 (4)





CCC

2019-10-28

真的非常谢谢老师，这个专栏我大多数文章都看了两遍以上，很多操作系统的细节关联的都搜了不少，很多以前只是了解的东西做到了真的理解了，再次谢谢老师！

展开 ▼





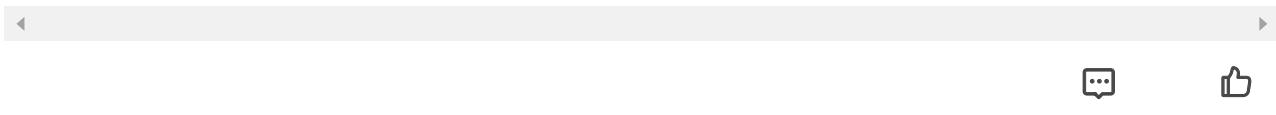
tt

2019-10-28

自己是做后端开发的，平常的开发工作也时常需要深入到TCP的底层去排除问题。学习完这个课程，真的是大大丰富了自己的网络知识细节。

尤其是最后的实战部分。最近在研究PYTHON和JAVASCRIPT中的异步编程模型和事件循环，但对它们的底层实现细节不清楚，看了老师的实战代码，里面也有事件循环，也有C...
展开 ▾

作者回复: 感谢支持，有收获是对我工作最大的肯定



传说中的成大大

2019-10-28

这是老师给我吗收拾的细软,让我们下山了吗? o(╥﹏╥)o

作者回复: 江湖还在，继续交流哦



MoonGod

2019-10-28

谢谢老师的解答，整个系列受益匪浅。

展开 ▾

作者回复: 非常欣慰，感谢支持

