

27 | I/O多路复用遇上线程：使用poll单线程处理所有I/O事件

2019-10-09 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 08:37 大小 7.90M



你好，我是盛延敏，这里是网络编程实战第 27 讲，欢迎回来。

我在前面两讲里，分别使用了 fork 进程和 pthread 线程来处理多并发，这两种技术使用简单，但是性能却会随着并发数的上涨而快速下降，并不能满足极端高并发的需求。就像第 24 讲中讲到的一样，这个时候我们需要寻找更好的解决之道，这个解决之道基本的思想就是 I/O 事件分发。


关于文稿中的代码，你可以去[GitHub](#)上查看或下载完整代码。

重温事件驱动

基于事件的程序设计：GUI、Web

事件驱动的好处是占用资源少，效率高，可扩展性强，是支持高性能高并发的不二之选。

如果你熟悉 GUI 编程的话，你就会知道，GUI 设定了一系列的控件，如 Button、Label、文本框等，当我们设计基于控件的程序时，一般都会给 Button 的点击安排一个函数，类似这样：

 复制代码

```
1 // 按钮点击的事件处理
2 void onButtonClick(){
3
4 }
```

这个设计的思想是，一个无限循环的事件分发线程在后台运行，一旦用户在界面上产生了某种操作，例如点击了某个 Button，或者点击了某个文本框，一个事件会被产生并放置到事件队列中，这个事件会有一个类似前面的 onButtonClick 回调函数。事件分发线程的任务，就是为每个发生的事件找到对应的事件回调函数并执行它。这样，一个基于事件驱动的 GUI 程序就可以完美地工作了。

还有一个类似的例子是 Web 编程领域。同样的，Web 程序会在 Web 界面上放置各种界面元素，例如 Label、文本框、按钮等，和 GUI 程序类似，给感兴趣的界面元素设计 JavaScript 回调函数，当用户操作时，对应的 JavaScript 回调函数会被执行，完成某个计算或操作。这样，一个基于事件驱动的 Web 程序就可以在浏览器中完美地工作了。

在第 24 讲中，我们已经提到，通过使用 poll、epoll 等 I/O 分发技术，可以设计出基于套接字的事件驱动程序，从而满足高性能、高并发的需求。

事件驱动模型，也被叫做反应堆模型（reactor），或者是 Event loop 模型。这个模型的核心有两点。

第一，它存在一个无限循环的事件分发线程，或者叫做 reactor 线程、Event loop 线程。这个事件分发线程的背后，就是 poll、epoll 等 I/O 分发技术的使用。

第二，所有的 I/O 操作都可以抽象成事件，每个事件必须有回调函数来处理。acceptor 上有连接建立成功、已连接套接字上发送缓冲区空出可以写、通信管道 pipe 上有数据可以

读，这些都是一个个事件，通过事件分发，这些事件都可以一一被检测，并调用对应的回调函数加以处理。

几种 I/O 模型和线程模型设计

任何一个网络程序，所做的事情可以总结成下面几种：

read：从套接字收取数据；

decode：对收到的数据进行解析；

compute：根据解析之后的内容，进行计算和处理；

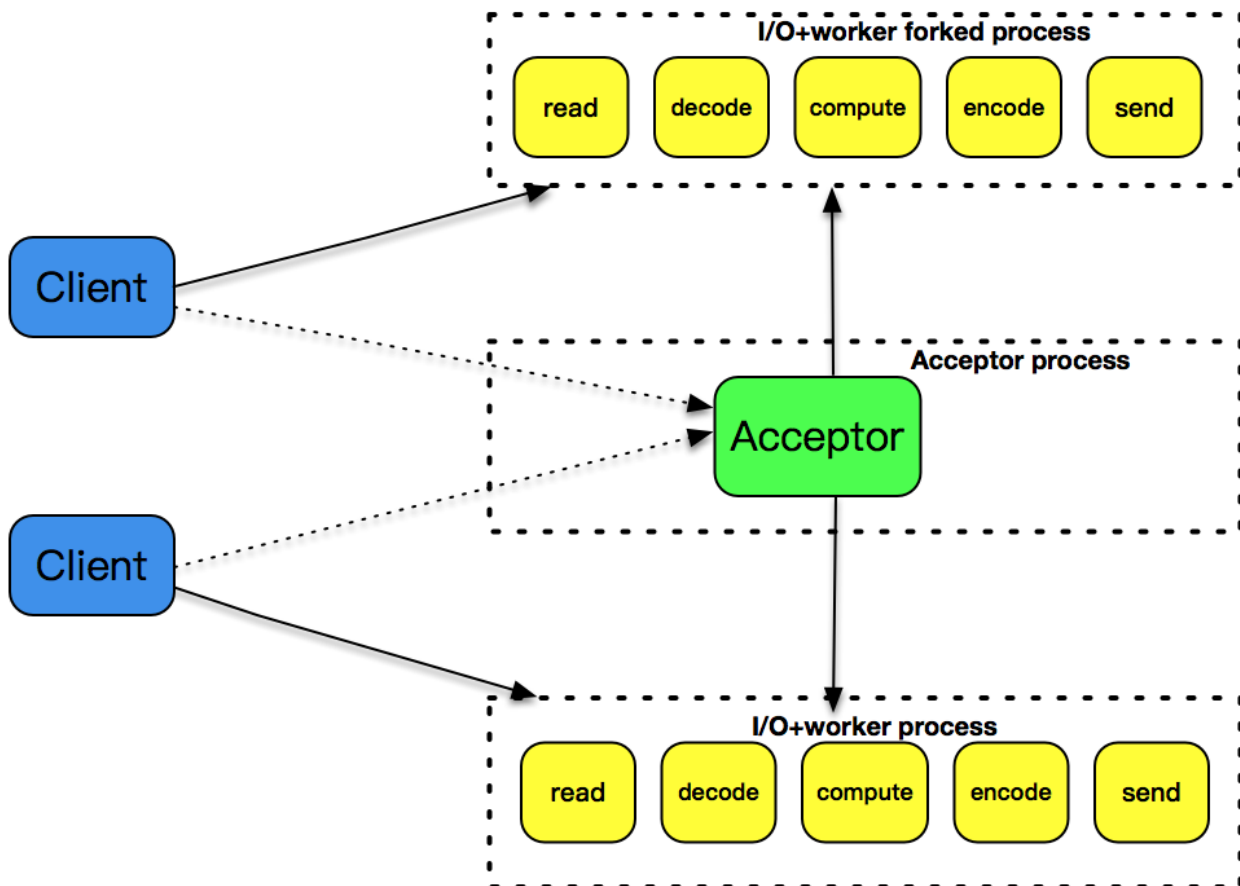
encode：将处理之后的结果，按照约定的格式进行编码；

send：最后，通过套接字把结果发送出去。

这几个过程和套接字最相关的是 read 和 send 这两种。接下来，我们总结一下已经学过的几种支持多并发的网络编程技术，引出我们今天的话题，使用 poll 单线程处理所有 I/O。

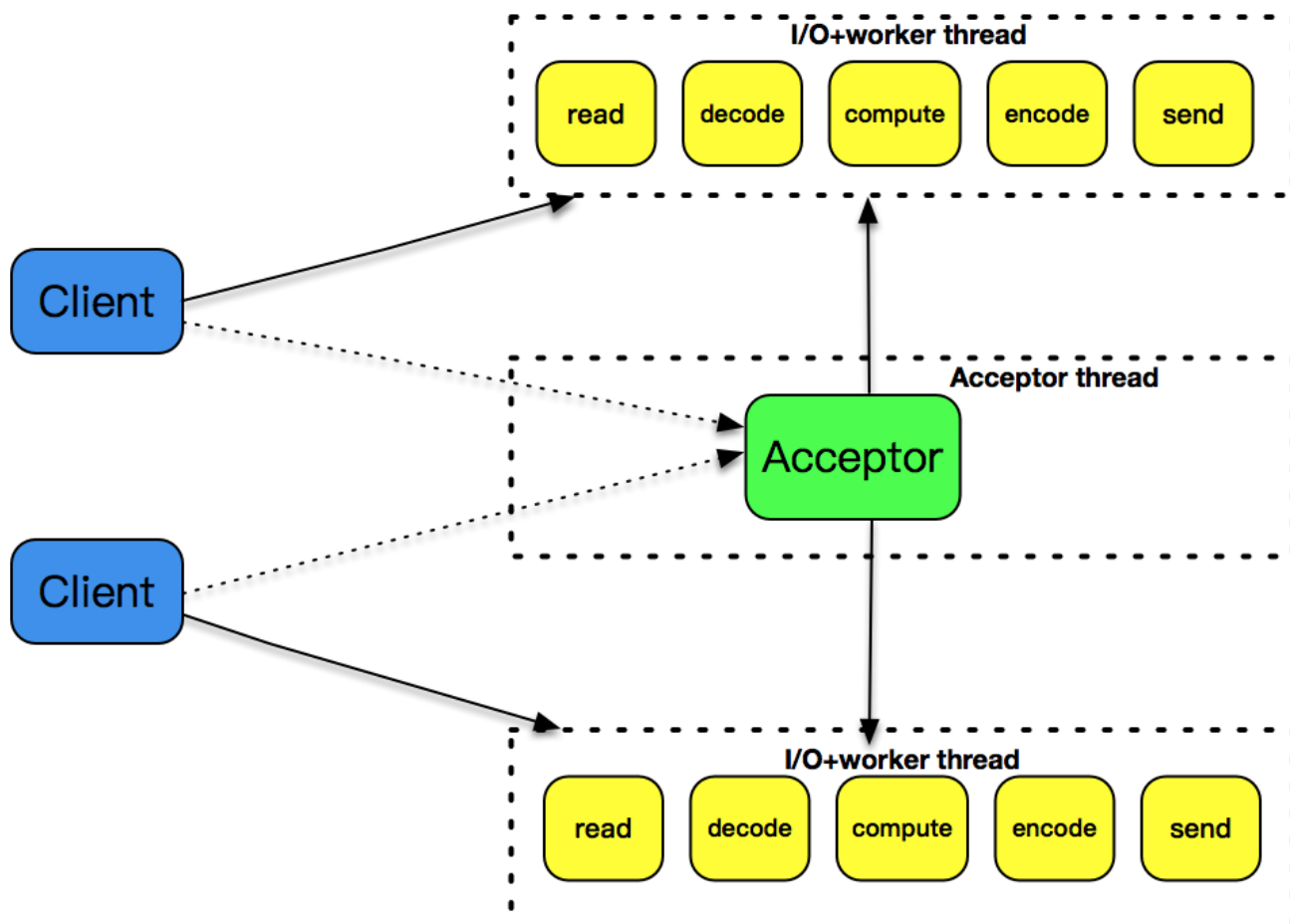
fork

第 25 讲中，我们使用 fork 来创建子进程，为每个到达的客户连接服务。文稿中的这张图很好地解释了这个设计模式，可想而知的是，随着客户数的变多，fork 的子进程也越来越多，即使客户和服务端之间的交互比较少，这样的子进程也不能被销毁，一直需要存在。使用 fork 的方式处理非常简单，它的缺点是处理效率不高，fork 子进程的开销太大。



pthread

第 26 讲中，我们使用了 `pthread_create` 创建子线程，因为线程是比进程更轻量级的执行单位，所以它的效率相比 `fork` 的方式，有一定的提高。但是，每次创建一个线程的开销仍然是不小的，因此，引入了线程池的概念，预先创建出一个线程池，在每次新连接达到时，从线程池挑选出一个线程为之服务，很好地解决了线程创建的开销。但是，这个模式还是没有解决空闲连接占用资源的问题，如果一个连接在一定时间内没有数据交互，这个连接还是要占用一定的线程资源，直到这个连接消亡为止。

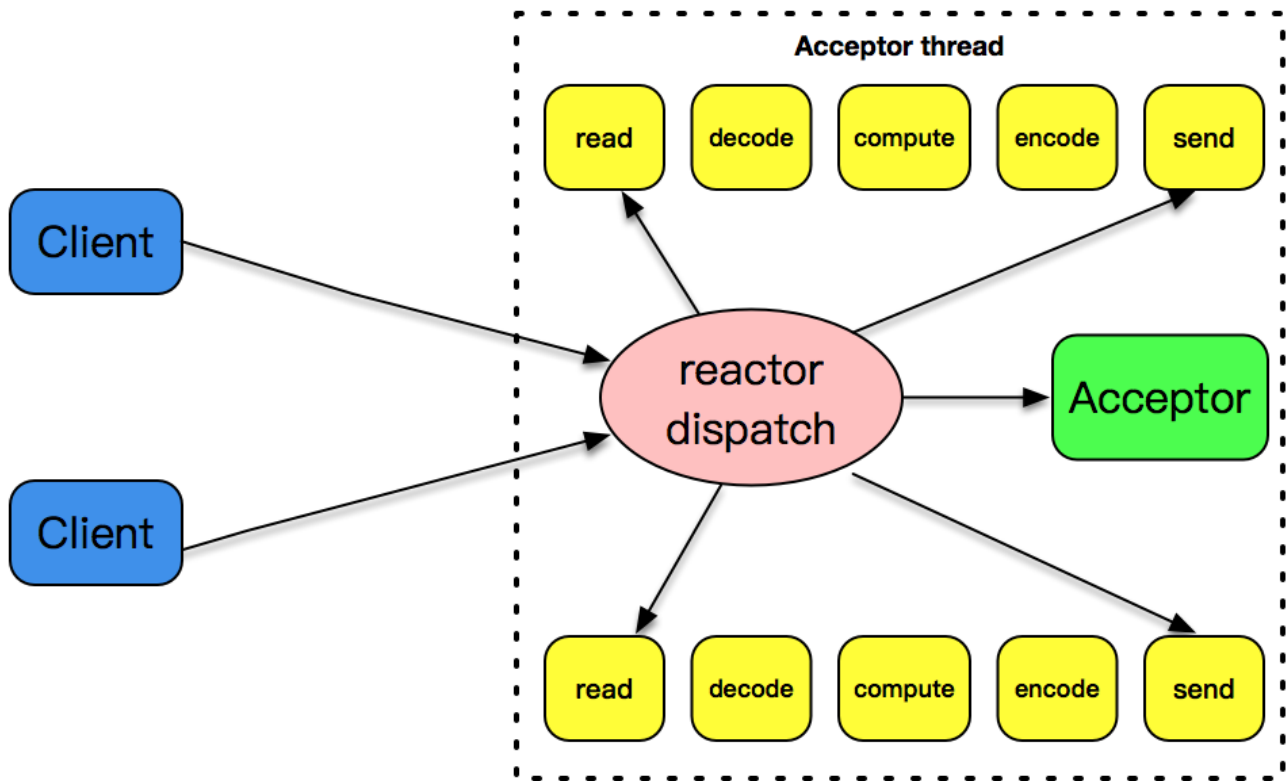


single reactor thread

前面讲到，事件驱动模式是解决高性能、高并发比较好的一种模式。为什么呢？

因为这种模式是符合大规模生产的需求的。我们的生活中遍地都是类似的模式。比如你去咖啡店喝咖啡，你点了一杯咖啡在一旁喝着，服务员也不会管你，等你有续杯需求的时候，再去和服务员提（触发事件），服务员满足了你的需求，你就继续可以喝着咖啡玩手机。整个柜台的服务方式就是一个事件驱动的方式。

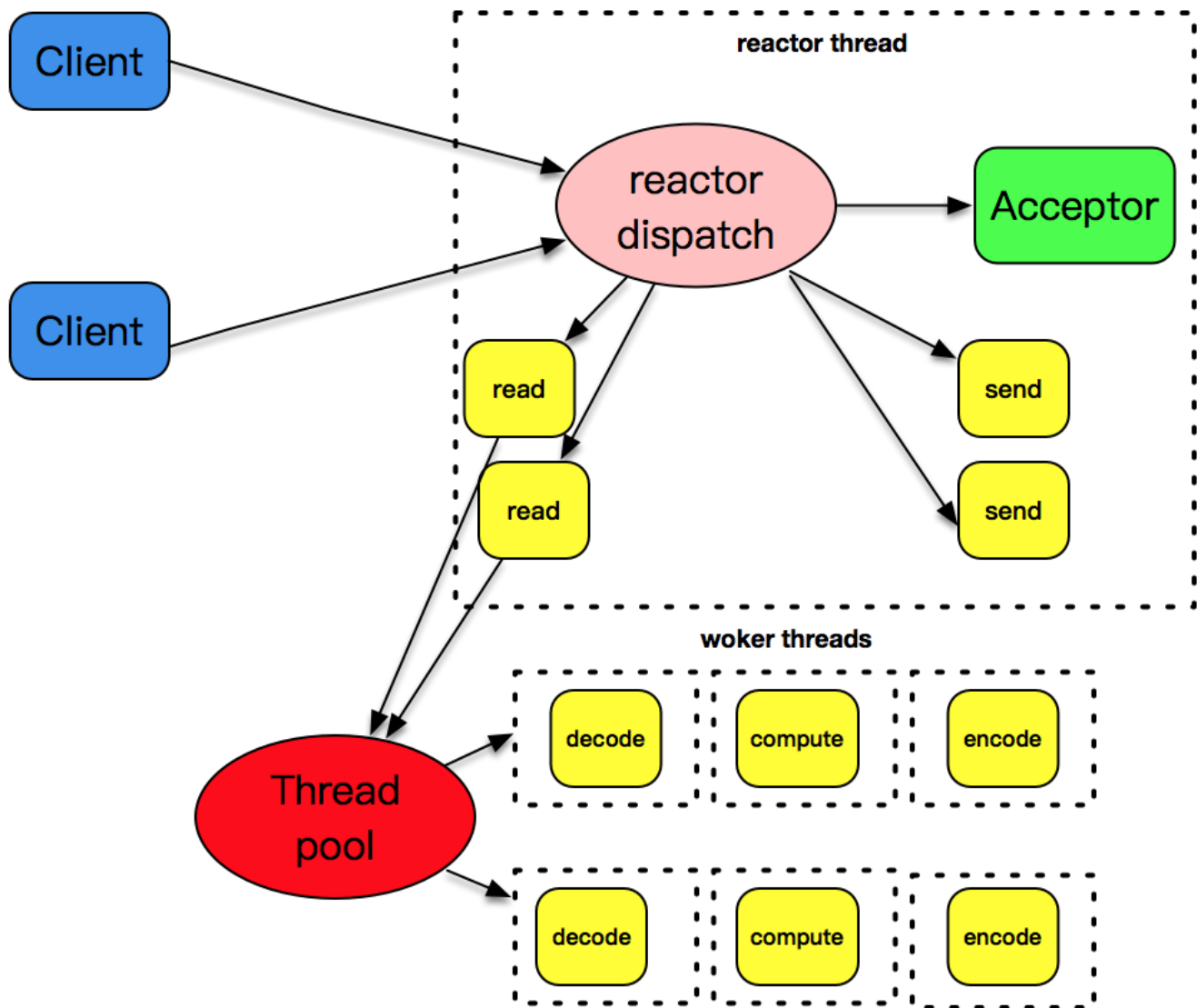
我在文稿中放了一张图解释了这一讲的设计模式。一个 reactor 线程上同时负责分发 acceptor 的事件、已连接套接字的 I/O 事件。



single reactor thread + worker threads

但是上述的设计模式有一个问题，和 I/O 事件处理相比，应用程序的业务逻辑处理是比较耗时的，比如 XML 文件的解析、数据库记录的查找、文件资料的读取和传输、计算型工作的处理等，这些工作相对而言比较独立，它们会拖慢整个反应堆模式的执行效率。

所以，将这些 decode、compute、encode 型工作放置到另外的线程池中，和反应堆线程解耦，是一个比较明智的选择。我在文稿中放置了这样的一张图。反应堆线程只负责处理 I/O 相关的工作，业务逻辑相关的工作都被裁剪成一个一个小任务，放到线程池里由空闲的线程来执行。当结果完成后，再交给反应堆线程，由反应堆线程通过套接字将结果发送出去。



样例程序

从今天开始，我们会接触到为本课程量身定制的网络编程框架。使用这个网络编程框架的样例程序已经放到文稿中：

复制代码

```

1 #include <lib/acceptor.h>
2 #include "lib/common.h"
3 #include "lib/event_loop.h"
4 #include "lib/tcp_server.h"
5
6 char rot13_char(char c) {
7     if ((c >= 'a' && c <= 'm') || (c >= 'A' && c <= 'M'))
8         return c + 13;
9     else if ((c >= 'n' && c <= 'z') || (c >= 'N' && c <= 'Z'))
10         return c - 13;
11     else
12         return c;
13 }

```

```

14
15 // 连接建立之后的 callback
16 int onConnectionCompleted(struct tcp_connection *tcpConnection) {
17     printf("connection completed\n");
18     return 0;
19 }
20
21 // 数据读到 buffer 之后的 callback
22 int onMessage(struct buffer *input, struct tcp_connection *tcpConnection) {
23     printf("get message from tcp connection %s\n", tcpConnection->name);
24     printf("%s", input->data);
25
26     struct buffer *output = buffer_new();
27     int size = buffer_readable_size(input);
28     for (int i = 0; i < size; i++) {
29         buffer_append_char(output, rot13_char(buffer_read_char(input)));
30     }
31     tcp_connection_send_buffer(tcpConnection, output);
32     return 0;
33 }
34
35 // 数据通过 buffer 写完之后的 callback
36 int onWriteCompleted(struct tcp_connection *tcpConnection) {
37     printf("write completed\n");
38     return 0;
39 }
40
41 // 连接关闭之后的 callback
42 int onConnectionClosed(struct tcp_connection *tcpConnection) {
43     printf("connection closed\n");
44     return 0;
45 }
46
47 int main(int c, char **v) {
48     // 主线程 event_loop
49     struct event_loop *eventLoop = event_loop_init();
50
51     // 初始化 acceptor
52     struct acceptor *acceptor = acceptor_init(SERV_PORT);
53
54     // 初始 tcp_server, 可以指定线程数目, 如果线程是 0, 就只有一个线程, 既负责 acceptor, 也负责
55     struct TCPserver *tcpServer = tcp_server_init(eventLoop, acceptor, onConnectionCompleted,
56                                                     onWriteCompleted, onConnectionClosed,
57     tcp_server_start(tcpServer);
58
59     // main thread for acceptor
60     event_loop_run(eventLoop);
61 }

```


这个程序的 main 函数部分只有几行, 因为是第一次接触到, 稍微展开介绍一下。

第 49 行创建了一个 event_loop, 即 reactor 对象, 这个 event_loop 和线程相关联, 每个 event_loop 在线程里执行的是一个无限循环, 以便完成事件的分发。

第 52 行初始化了 acceptor, 用来监听在某个端口上。


第 55 行创建了一个 TCPServer, 创建的时候可以指定线程数目, 这里线程是 0, 就只有一个线程, 既负责 acceptor 的连接处理, 也负责已连接套接字的 I/O 处理。这里比较重要的是传入了几个回调函数, 分别对应了连接建立完成、数据读取完成、数据发送完成、连接关闭完成几种操作, 通过回调函数, 让业务程序可以聚焦在业务层开发。

第 57 行开启监听。

第 60 行运行 event_loop 无限循环, 等待 acceptor 上有连接建立、新连接上有数据可读等。

样例程序结果

运行这个服务器程序, 开启两个 telnet 客户端, 我们看到服务器端的输出如下:

 复制代码

```
1  ./poll-server-onethread
2  [msg] set poll as dispatcher
3  [msg] add channel fd == 4, main thread
4  [msg] poll added channel fd==4
5  [msg] add channel fd == 5, main thread
6  [msg] poll added channel fd==5
7  [msg] event loop run, main thread
8  [msg] get message channel i==1, fd==5
9  [msg] activate channel fd == 5, revents=2, main thread
10 [msg] new connection established, socket == 6
11 connection completed
12 [msg] add channel fd == 6, main thread
13 [msg] poll added channel fd==6
14 [msg] get message channel i==2, fd==6
15 [msg] activate channel fd == 6, revents=2, main thread
16 get message from tcp connection connection-6
17 afadsfaf
18 [msg] get message channel i==2, fd==6
19 [msg] activate channel fd == 6, revents=2, main thread
20 get message from tcp connection connection-6
```

```
21 afadsfaf
22 fdafasf
23 [msg] get message channel i==1, fd==5
24 [msg] activate channel fd == 5, revents=2, main thread
25 [msg] new connection established, socket == 7
26 connection completed
27 [msg] add channel fd == 7, main thread
28 [msg] poll added channel fd==7
29 [msg] get message channel i==3, fd==7
30 [msg] activate channel fd == 7, revents=2, main thread
31 get message from tcp connection connection-7
32 sfasggwqe
33 [msg] get message channel i==3, fd==7
34 [msg] activate channel fd == 7, revents=2, main thread
35 [msg] poll delete channel fd==7
36 connection closed
37 [msg] get message channel i==2, fd==6
38 [msg] activate channel fd == 6, revents=2, main thread
39 [msg] poll delete channel fd==6
40 connection closed
```

这里自始至终都只有一个 main thread 在工作，可见，单线程的 reactor 处理多个连接时也可以表现良好。

总结

这一讲我们总结了几种不同的 I/O 模型和线程模型设计，并比较了各自不同的优缺点。从这一讲开始，我们将使用自己编写的编程框架来完成业务开发，这一讲使用了 poll 来处理所有的 I/O 事件，在下一讲里，我们将会看到如何把 acceptor 的连接事件和已连接套接字的 I/O 事件交由不同的线程处理，而这个分离，不过是在应用程序层简单的参数配置而已。

思考题

和往常一样，给大家留两道思考题：

1. 你可以试着修改一下 onMessage 方法，把它变为期中作业中提到的 cd、ls 等 command 实现。
2. 文章里服务器端的 decode-compute-encode 是在哪里实现的？你有什么办法来解决业务逻辑和 I/O 逻辑混在一起么？

欢迎你在评论区写下你的思考，或者在 GitHub 上上传你的代码，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 26 | 使用阻塞I/O和线程模型：换一种轻量化的方式

下一篇 28 | I/O多路复用进阶：子线程使用poll处理连接I/O事件

精选留言 (7)

写留言



fxzhang

2019-10-09

老师可否讲解linux下如何开发的，最近想换工作，但是之前都在windows下面开发，想自学一下linux下是如何开发的，但是有一种找不到开头不知道该怎么学习的感觉，很无力

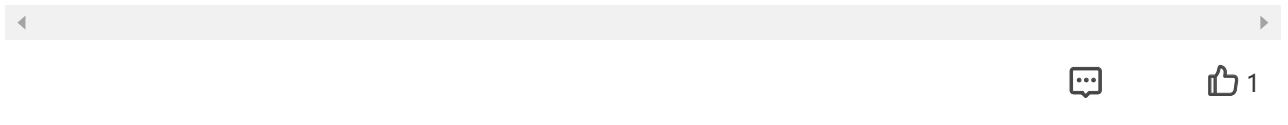
作者回复: 先学习一下Linux下的安装、配置、管理，把工作环境放到Linux下面，让Linux成为你的工作效率机器；

其次，慢慢学习Bash，感受一下Linux的能力；

接下来就是学习一些 Linux下的程序设计，如I/O、网络等。

如果你能把这篇系列的所有代码都改一遍，运行一遍，就是一个良好的开头。

加油~



Berry Wang

2019-10-12

“文稿中的这张图很好地解释了这个设计模式，可想而知的是，随着客户数的变多，fork的子进程也越来越多，即使客户和服务端之间的交互比较少，这样的子进程也不能被销毁，一直需要存在。” 老师这里的子进程需要一直存在是为什么呢？

展开 ▾

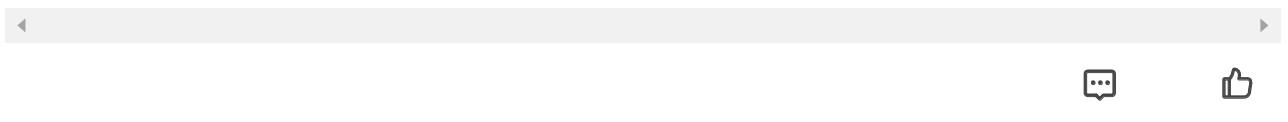


向东

2019-10-10

老师能否对事件分发调用event_loop的event_activate方法执行callback的部分涉及回调部分讲详细点呢？

作者回复: 第四篇会详细进行分解讲述。



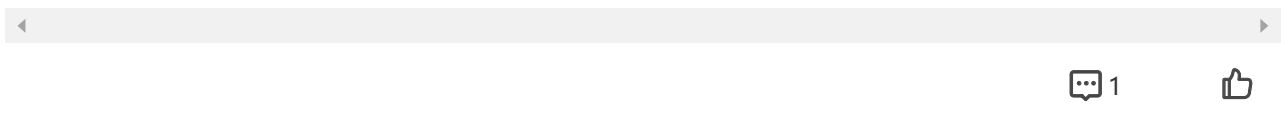
卡卡

2019-10-09

老师的代码 github上有 地址有同学已经发出来啦

展开 ▾

作者回复: 📖

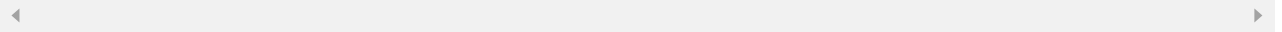


沉淀的梦想

2019-10-09

onWriteCompleted是在什么情况被回调的呢？在整个测试中似乎都没有被回调

作者回复: 写完成之后, 你可以打印一段话看看是否被回调到。



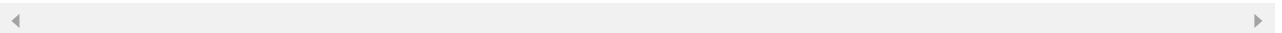
刘丹

2019-10-09

请问 lib 目录下的代码能贴出来给大家学习吗?

展开 ∨

作者回复: <https://github.com/froghui/yolanda>



hello world

2019-10-09

老师完整的代码可以贴出来吗

展开 ∨

作者回复: <https://github.com/froghui/yolanda>

