



下载APP

32 | 自己动手写高性能HTTP服务器（一）：设计和思路

2019-10-21 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 10:20 大小 9.48M



□你好，我是盛延敏，这里是网络编程实战第 32 讲，欢迎回来。

从这一讲开始，我们进入实战篇，开启一个高性能 HTTP 服务器的编写之旅。

在开始编写高性能 HTTP 服务器之前，我们先要构建一个支持 TCP 的高性能网络编程框架，完成这个 TCP 高性能网络框架之后，再增加 HTTP 特性的支持就比较容易了，这样就可以很快开发出一个高性能的 HTTP 服务器程序。

设计需求

在第三个模块性能篇中，我们已经使用这个网络编程框架完成了多个应用程序的开发，这也等于对这个网络编程框架提出了编程接口方面的需求，综合之前的使用经验，这个 TCP 高

性能网络框架需要满足的需求有以下三点。

第一，采用 reactor 模型，可以灵活使用 poll/epoll 作为事件分发实现。

第二，必须支持多线程，从而可以支持单线程单 reactor 模式，也可以支持多线程主 - 从 reactor 模式。可以将套接字上的 I/O 事件分离到多个线程上。

第三，封装读写操作到 Buffer 对象中。

按照这三个需求，正好可以把整体设计思路分成三块来讲解，分别包括反应堆模式设计、I/O 模型和多线程模型设计、数据读写封装和 buffer。今天我们主要讲一下主要的设计思路和数据结构，以及反应堆模式设计。

主要设计思路

反应堆模式设计

反应堆模式，按照性能篇的讲解，主要是设计一个基于事件分发和回调的反应堆框架。这个框架里面的主要对象包括：

event_loop

你可以把 event_loop 这个对象理解成和一个线程绑定的无限事件循环，你会在各种语言里看到 event_loop 这个抽象。这是什么意思呢？简单来说，它就是一个无限循环着的事件分发器，一旦有事件发生，它就会回调预先定义好的回调函数，完成事件的处理。

具体来说，event_loop 使用 poll 或者 epoll 方法将一个线程阻塞，等待各种 I/O 事件的发生。

channel

对各种注册到 event_loop 上的对象，我们抽象成 channel 来表示，例如注册到 event_loop 上的监听事件，注册到 event_loop 上的套接字读写事件等。在各种语言的 API 里，你都会看到 channel 这个对象，大体上它们表达的意思跟我们这里的设计思路是比较一致的。

acceptor

acceptor 对象表示的是服务器端监听器， acceptor 对象最终会作为一个 channel 对象，注册到 event_loop 上，以便进行连接完成的事件分发和检测。

event_dispatcher

event_dispatcher 是对事件分发机制的一种抽象，也就是说，可以实现一个基于 poll 的 poll_dispatcher，也可以实现一个基于 epoll 的 epoll_dispatcher。在这里，我们统一设计一个 event_dispatcher 结构体，来抽象这些行为。

channel_map

channel_map 保存了描述字到 channel 的映射，这样就可以在事件发生时，根据事件类型对应的套接字快速找到 channel 对象里的事件处理函数。

I/O 模型和多线程模型设计

I/O 线程和多线程模型，主要解决 event_loop 的线程运行问题，以及事件分发和回调的线程执行问题。

thread_pool

thread_pool 维护了一个 sub-reactor 的线程列表，它可以提供给主 reactor 线程使用，每次当有新的连接建立时，可以从 thread_pool 里获取一个线程，以便用它来完成对新连接套接字的 read/write 事件注册，将 I/O 线程和主 reactor 线程分离。

event_loop_thread

event_loop_thread 是 reactor 的线程实现，连接套接字的 read/write 事件检测都是在这个线程里完成的。

Buffer 和数据读写

buffer

buffer 对象屏蔽了对套接字进行的写和读的操作，如果没有 buffer 对象，连接套接字的 read/write 事件都需要和字节流直接打交道，这显然是不友好的。所以，我们也提供了一个基本的 buffer 对象，用来表示从连接套接字收取的数据，以及应用程序即将需要发送出去的数据。

tcp_connection

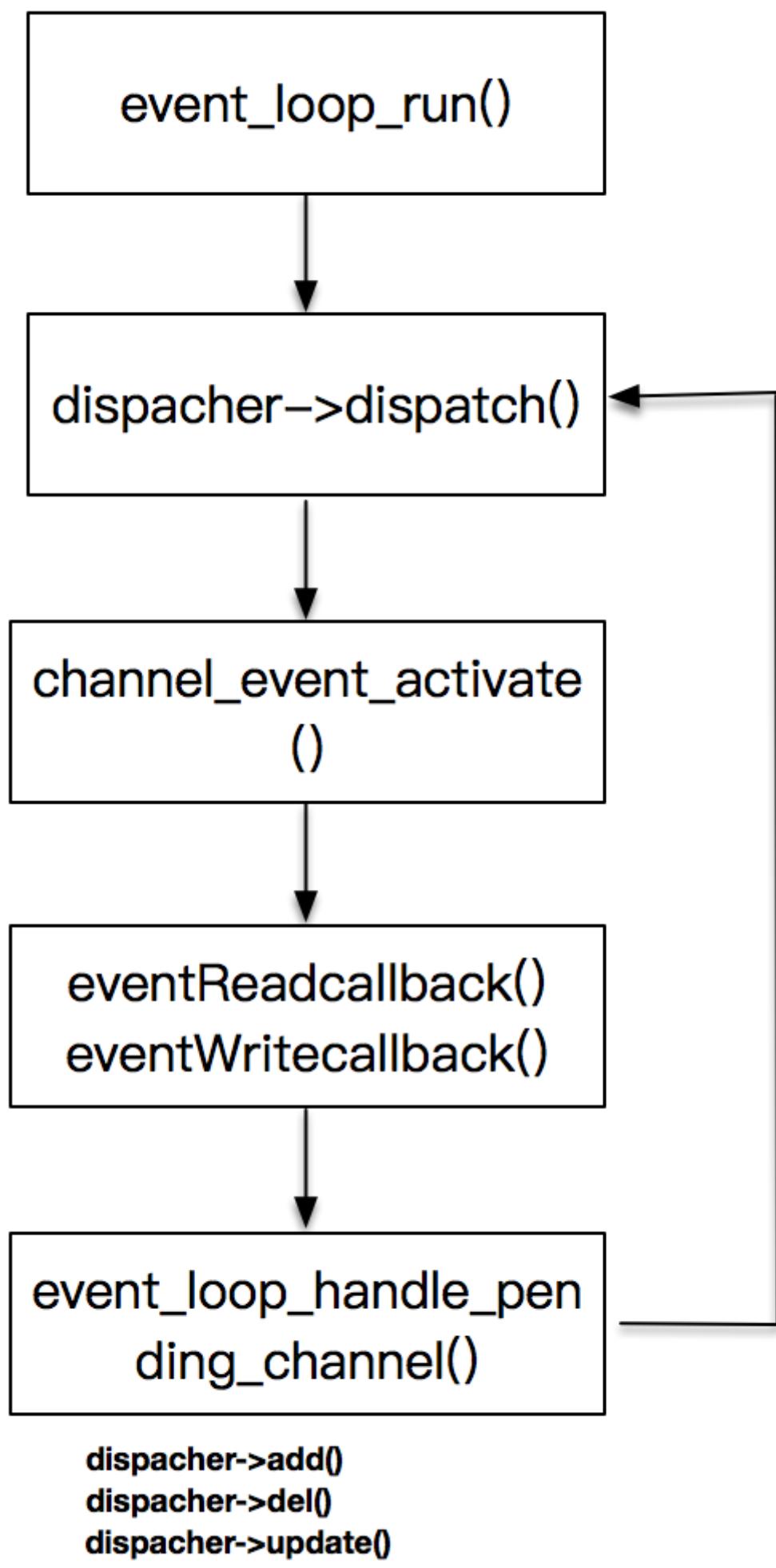
tcp_connection 这个对象描述的是已建立的 TCP 连接。它的属性包括接收缓冲区、发送缓冲区、channel 对象等。这些都是一个 TCP 连接的天然属性。

tcp_connection 是大部分应用程序和我们的高性能框架直接打交道的数据结构。我们不想把最下层的 channel 对象暴露给应用程序，因为抽象的 channel 对象不仅仅可以表示 tcp_connection，前面提到的监听套接字也是一个 channel 对象，后面提到的唤醒 socketpair 也是一个 channel 对象。所以，我们设计了 tcp_connection 这个对象，希望可以提供给用户比较清晰的编程入口。

反应堆模式设计

概述

下面，我们详细讲解一下以 event_loop 为核心的反应堆模式设计。我在文稿里放置了一张 event_loop 的运行详图，你可以对照这张图来理解。



当 event_loop_run 完成之后，线程进入循环，首先执行 dispatch 事件分发，一旦有事件发生，就会调用 channel_event_activate 函数，在这个函数中完成事件回调函数 eventReadcallback 和 eventWritecallback 的调用，最后再进行 event_loop_handle_pending_channel，用来修改当前监听的事件列表，完成这个部分之后，又进入了事件分发循环。

event_loop 分析

说 event_loop 是整个反应堆模式设计的核心，一点也不为过。先看一下 event_loop 的数据结构。

在这个数据结构中，最重要的莫过于 event_dispatcher 对象了。你可以简单地把 event_dispatcher 理解为 poll 或者 epoll，它可以让我们的线程挂起，等待事件的发生。

这里有一个小技巧，就是 event_dispatcher_data，它被定义为一个 void * 类型，可以按照我们的需求，任意放置一个我们需要的对象指针。这样，针对不同的实现，例如 poll 或者 epoll，都可以根据需求，放置不同的数据对象。

event_loop 中还保留了几个跟多线程有关的对象，如 owner_thread_id 是保留了每个 event loop 的线程 ID，mutex 和 con 是用来进行线程同步的。

socketPair 是父线程用来通知子线程有新的事件需要处理。pending_head 和 pending_tail 是保留在子线程内的需要处理的新的事件。

 复制代码

```
1 struct event_loop {
2     int quit;
3     const struct event_dispatcher *eventDispatcher;
4
5     /** 对应的 event_dispatcher 的数据. */
6     void *event_dispatcher_data;
7     struct channel_map *channelMap;
8
9     int is_handle_pending;
10    struct channel_element *pending_head;
11    struct channel_element *pending_tail;
12
13    pthread_t owner_thread_id;
14    pthread_mutex_t mutex;
```

```
15     pthread_cond_t cond;
16     int socketPair[2];
17     char *thread_name;
18 }
```

下面我们看一下 event_loop 最主要的方法 event_loop_run 方法，前面提到过，event_loop 就是一个无限 while 循环，不断地在分发事件。

 复制代码

```
1 /**
2 *
3 * 1. 参数验证
4 * 2. 调用 dispatcher 来进行事件分发，分发完回调事件处理函数
5 */
6 int event_loop_run(struct event_loop *eventLoop) {
7     assert(eventLoop != NULL);
8
9     struct event_dispatcher *dispatcher = eventLoop->eventDispatcher;
10
11    if (eventLoop->owner_thread_id != pthread_self()) {
12        exit(1);
13    }
14
15    yolanda_msgx("event loop run, %s", eventLoop->thread_name);
16    struct timeval timeval;
17    timeval.tv_sec = 1;
18
19    while (!eventLoop->quit) {
20        //block here to wait I/O event, and get active channels
21        dispatcher->dispatch(eventLoop, &timeval);
22
23        //handle the pending channel
24        event_loop_handle_pending_channel(eventLoop);
25    }
26
27    yolanda_msgx("event loop end, %s", eventLoop->thread_name);
28    return 0;
29 }
```

代码很明显地反映了这一点，这里我们在 event_loop 不退出的情况下，一直在循环，循环体中调用了 dispatcher 对象的 dispatch 方法来等待事件的发生。

event_dispatcher 分析

为了实现不同的事件分发机制，这里把 poll、epoll 等抽象成了一个 event_dispatcher 结构。event_dispatcher 的具体实现有 poll_dispatcher 和 epoll_dispatcher 两种，实现的方法和性能篇[21讲](#)和[22 讲](#)类似，这里就不再赘述，你如果有兴趣的话，可以直接研读代码。

 复制代码

```
1 /** 抽象的 event_dispatcher 结构体，对应的实现如 select,poll,epoll 等 I/O 复用. */
2 struct event_dispatcher {
3     /** 对应实现 */
4     const char *name;
5
6     /** 初始化函数 */
7     void *(*init)(struct event_loop * eventLoop);
8
9     /** 通知 dispatcher 新增一个 channel 事件 */
10    int (*add)(struct event_loop * eventLoop, struct channel * channel);
11
12    /** 通知 dispatcher 删除一个 channel 事件 */
13    int (*del)(struct event_loop * eventLoop, struct channel * channel);
14
15    /** 通知 dispatcher 更新 channel 对应的事件 */
16    int (*update)(struct event_loop * eventLoop, struct channel * channel);
17
18    /** 实现事件分发，然后调用 event_loop 的 event_activate 方法执行 callback*/
19    int (*dispatch)(struct event_loop * eventLoop, struct timeval *);
20
21    /** 清除数据 */
22    void (*clear)(struct event_loop * eventLoop);
23};
```

channel 对象分析

channel 对象是用来和 event_dispatcher 进行交互的最主要的结构体，它抽象了事件分发。一个 channel 对应一个描述字，描述字上可以有 READ 可读事件，也可以有 WRITE 可写事件。channel 对象绑定了事件处理函数 event_read_callback 和 event_write_callback。

 复制代码

```
1 typedef int (*event_read_callback)(void *data);
2
```

```
3 typedef int (*event_write_callback)(void *data);
4
5 struct channel {
6     int fd;
7     int events; // 表示 event 类型
8
9     event_read_callback eventReadCallback;
10    event_write_callback eventWriteCallback;
11    void *data; //callback data, 可能是 event_loop, 也可能是 tcp_server 或者 tcp_connecti
12 };
```

◀ ▶

channel_map 对象分析

event_dispatcher 在获得活动事件列表之后，需要通过文件描述字找到对应的 channel，从而回调 channel 上的事件处理函数 event_read_callback 和 event_write_callback，为此，设计了 channel_map 对象。

复制代码

```
1 /**
2  * channel 映射表, key 为对应的 socket 描述字
3  */
4 struct channel_map {
5     void **entries;
6
7     /* The number of entries available in entries */
8     int nentries;
9 };
```

◀ ▶

channel_map 对象是一个数组，数组的下标即为描述字，数组的元素为 channel 对象的地址。

比如描述字 3 对应的 channel，就可以这样直接得到。

复制代码

```
1 struct chanenl * channel = map->entries[3];
```

◀ ▶

这样，当 event_dispatcher 需要回调 channel 上的读、写函数时，调用 channel_event_activate 就可以，下面是 channel_event_activate 的实现，在找到了对应的 channel 对象之后，根据事件类型，回调了读函数或者写函数。注意，这里使用了 EVENT_READ 和 EVENT_WRITE 来抽象了 poll 和 epoll 的所有读写事件类型。

 复制代码

```
1 int channel_event_activate(struct event_loop *eventLoop, int fd, int revents) {
2     struct channel_map *map = eventLoop->channelMap;
3     yolanda_msgx("activate channel fd == %d, revents=%d, %s", fd, revents, eventLoop->t
4
5     if (fd < 0)
6         return 0;
7
8     if (fd >= map->nentries) return (-1);
9
10    struct channel *channel = map->entries[fd];
11    assert(fd == channel->fd);
12
13    if (revents & (EVENT_READ)) {
14        if (channel->eventReadCallback) channel->eventReadCallback(channel->data);
15    }
16    if (revents & (EVENT_WRITE)) {
17        if (channel->eventWriteCallback) channel->eventWriteCallback(channel->data);
18    }
19
20    return 0;
21 }
```



增加、删除、修改 channel event

那么如何增加新的 channel event 事件呢？这几个函数是用来增加、删除和修改 channel event 事件的。

 复制代码

```
1 int event_loop_add_channel_event(struct event_loop *eventLoop, int fd, struct channel *c
2
3 int event_loop_remove_channel_event(struct event_loop *eventLoop, int fd, struct channel *
4
5 int event_loop_update_channel_event(struct event_loop *eventLoop, int fd, struct channel *
```



前面三个函数提供了入口能力，而真正的实现则落在这三个函数上：

 复制代码

```
1 int event_loop_handle_pending_add(struct event_loop *eventLoop, int fd, struct channel *channel);
2
3 int event_loop_handle_pending_remove(struct event_loop *eventLoop, int fd, struct channel *channel);
4
5 int event_loop_handle_pending_update(struct event_loop *eventLoop, int fd, struct channel *channel);
```



我们看一下其中的一个实现，`event_loop_handle_pendign_add` 在当前 `event_loop` 的 `channel_map` 里增加一个新的 key-value 对，key 是文件描述字，value 是 `channel` 对象的地址。之后调用 `event_dispatcher` 对象的 `add` 方法增加 `channel event` 事件。注意这个方法总在当前的 I/O 线程中执行。

 复制代码

```
1 // in the i/o thread
2 int event_loop_handle_pending_add(struct event_loop *eventLoop, int fd, struct channel *channel) {
3     yolanda_msgx("add channel fd == %d, %s", fd, eventLoop->thread_name);
4     struct channel_map *map = eventLoop->channelMap;
5
6     if (fd < 0)
7         return 0;
8
9     if (fd >= map->nentries) {
10         if (map_make_space(map, fd, sizeof(struct channel *)) == -1)
11             return (-1);
12     }
13
14     // 第一次创建，增加
15     if ((map)->entries[fd] == NULL) {
16         map->entries[fd] = calloc(1, sizeof(struct channel *));
17         map->entries[fd] = channel;
18         //add channel
19         struct event_dispatcher *eventDispatcher = eventLoop->eventDispatcher;
20         eventDispatcher->add(eventLoop, channel);
21         return 1;
22     }
23
24     return 0;
25 }
```



总结

在这一讲里，我们介绍了高性能网络编程框架的主要设计思路和基本数据结构，以及反应堆设计相关的具体做法。在接下来的章节中，我们将继续编写高性能网络编程框架的线程模型以及读写 Buffer 部分。

思考题

和往常一样，给你留两道思考题：

第一道，如果你有兴趣，不妨实现一个 select_dispatcher 对象，用 select 方法实现定义好的 event_dispatcher 接口；

第二道，仔细研读 channel_map 实现中的 map_make_space 部分，说说你的理解。

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。

The image shows the cover of a book titled '网络编程实战' (Network Programming Practical) by Sheng Yanmin. The cover features a portrait of the author, Sheng Yanmin, a man with dark hair wearing a grey blazer over a light blue shirt. To his left, the title '网络编程实战' is displayed in large, bold, dark letters. Below it, the subtitle '从底层到实战，深度解析网络编程' (From the bottom layer to practical combat, deeply parse network programming) is written in a smaller, lighter font. In the top left corner, there is a logo consisting of a stylized orange 'Q' shape followed by the text '极客时间'. At the bottom of the cover, there is a dark banner with white text that reads: '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。' (New version upgrade: Click 'Send to friends to read', 20 friends can read for free, invite to subscribe and get cash reward).

© 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。

精选留言 (3)

 写留言



鱼向北游

2019-10-21

老师的程序读了一遍，c版的netty，果然高手们的思路都是相通的



传说中的成大大

2019-10-21

第二道题 就是一个扩容啊 类似std的vector自动扩容 而且每次成倍的增长



刘系

2019-10-21

第二课后题：当描述字大于channel_map的容量时，map_make_space会被调用。在map初始化时，容量为0，往map里写描述字时先给容量为32，如果描述字仍然大于等于32将会使容量右移一位，也就是描述字容量增加两倍再与要写入的描述字进行比较，直至容量大于要写入的描述字。然后使用realloc进行空间开辟，保留原有空间，扩展新空间。将新空间内存置0。最后更新map

展开 ▼

