



下载APP



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

2019-10-23 盛延敏

网络编程实战

进入课程 >



讲述：冯永吉

时长 11:35 大小 10.62M



你好，我是盛延敏，这里是网络编程实战第 33 讲，欢迎回来。

这一讲，我们延续第 32 讲的话题，继续解析高性能网络编程框架的 I/O 模型和多线程模型设计部分。

多线程设计的几个考虑

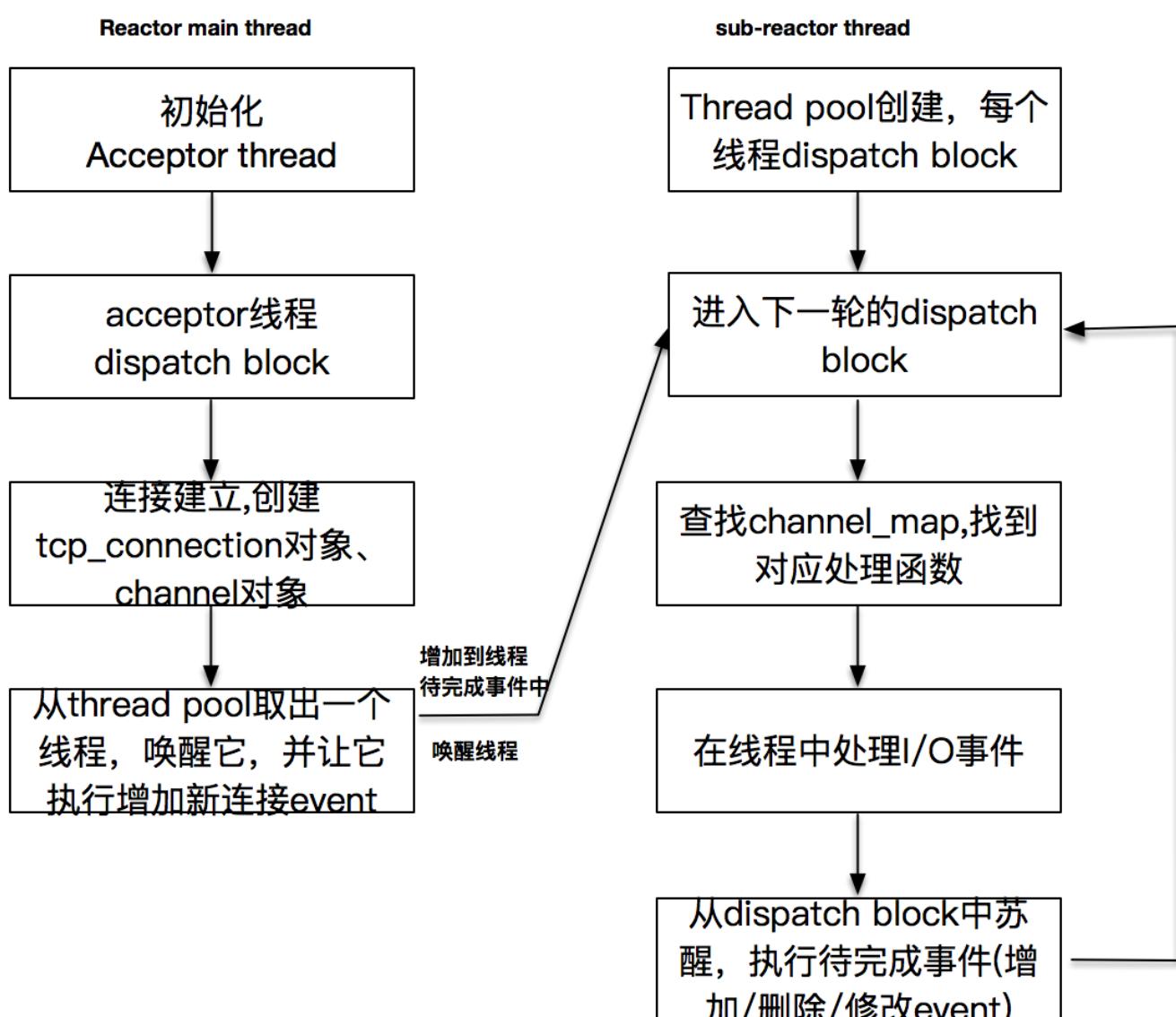
在我们的设计中，main reactor 线程是一个 acceptor 线程，这个线程一旦创建，会以 event_loop 形式阻塞在 event_dispatcher 的 dispatch 方法上，实际上，它在等待监听套接字上的事件发生，也就是已完成的连接，一旦有连接完成，就会创建出连接对象 tcp_connection，以及 channel 对象等。

当用户期望使用多个 sub-reactor 子线程时，主线程会创建多个子线程，每个子线程在创建之后，按照主线程指定的启动函数立即运行，并进行初始化。随之而来的问题是，**主线程如何判断子线程已经完成初始化并启动，继续执行下去呢？这是一个需要解决的重点问题。**

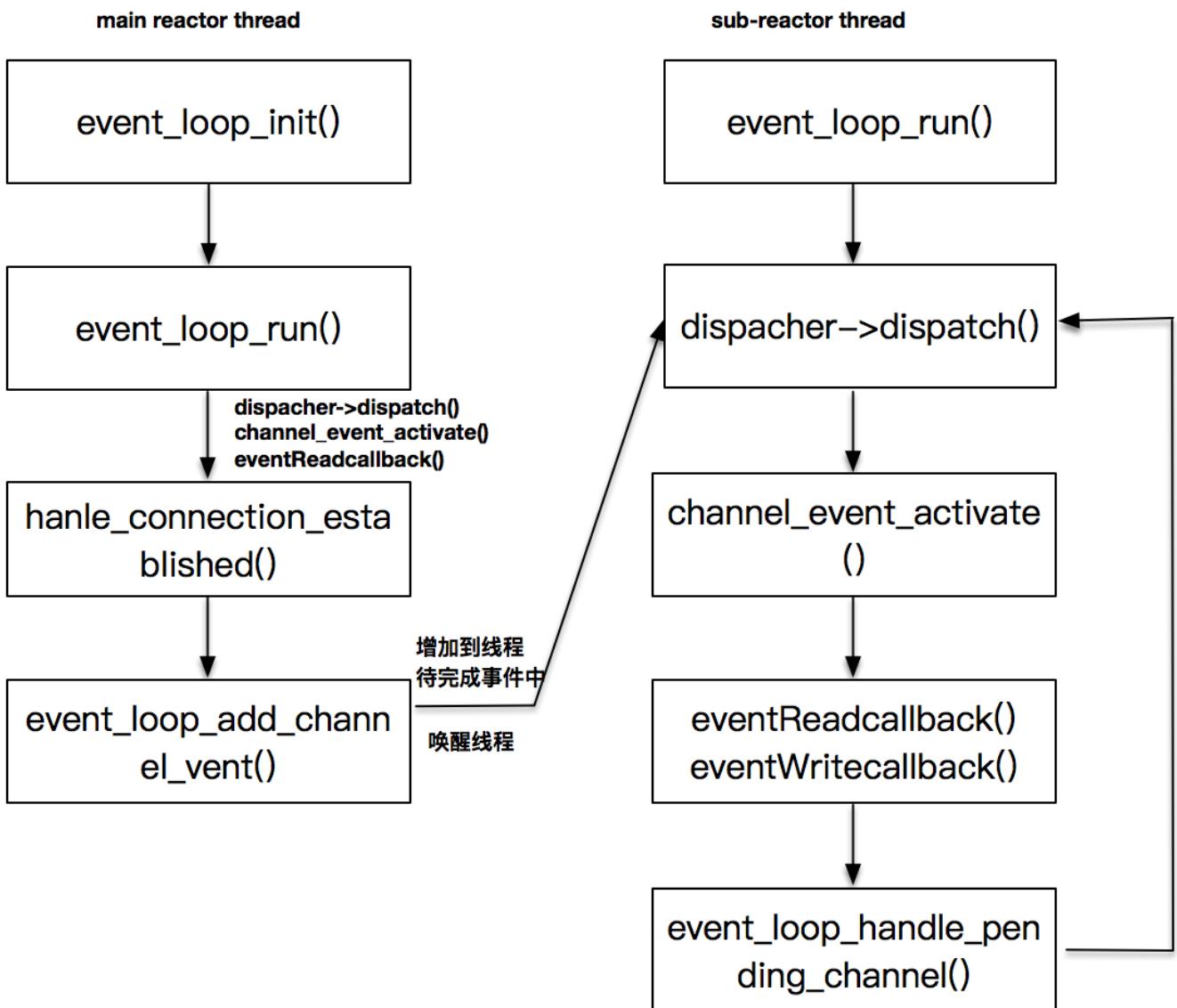
在设置了多个线程的情况下，需要将新创建的已连接套接字对应的读写事件交给一个 sub-reactor 线程处理。所以，这里从 thread_pool 中取出一个线程，**通知这个线程有新的事件加入。而这个线程很可能是处于事件分发的阻塞调用之中，如何协调主线程数据写入给子线程，这是另一个需要解决的重点问题。**

子线程是一个 event_loop 线程，它阻塞在 dispatch 上，一旦有事件发生，它就会查找 channel_map，找到对应的处理函数并执行它。之后它就会增加、删除或修改 pending 事件，再次进入下一轮的 dispatch。

文稿中放置了一张图，阐述了线程的运行关系。



为了方便你理解，我把对应的函数实现列在了另外一张图中。



主线程等待多个 sub-reactor 子线程初始化完

主线程需要等待子线程完成初始化，也就是需要获取子线程对应数据的反馈，而子线程初始化也是对这部分数据进行初始化，实际上这是一个多线程的通知问题。采用的做法在 [前面](#) 讲多线程的时候也提到过，使用 mutex 和 condition 两个主要武器。

下面这段代码是主线程发起的子线程创建，调用 `event_loop_thread_init` 对每个子线程初始化，之后调用 `event_loop_thread_start` 来启动子线程。注意，如果应用程序指定的线程池大小为 0，则直接返回，这样 acceptor 和 I/O 事件都会在同一个主线程里处理，就退化为单 reactor 模式。

[复制代码](#)

```
1 // 一定是 main thread 发起
2 void thread_pool_start(struct thread_pool *threadPool) {
3     assert(!threadPool->started);
4     assertInSameThread(threadPool->mainLoop);
5
6     threadPool->started = 1;
7     void *tmp;
8
9     if (threadPool->thread_number <= 0) {
10         return;
11     }
12
13     threadPool->eventLoopThreads = malloc(threadPool->thread_number * sizeof(s·
14     for (int i = 0; i < threadPool->thread_number; ++i) {
15         event_loop_thread_init(&threadPool->eventLoopThreads[i], i);
16         event_loop_thread_start(&threadPool->eventLoopThreads[i]);
17     }
18 }
```

我们再看一下 `event_loop_thread_start` 这个方法，这个方法一定是主线程运行的。这里我使用了 `pthread_create` 创建了子线程，子线程一旦创建，立即执行 `event_loop_thread_run`，我们稍后将看到，`event_loop_thread_run` 进行了子线程的初始化工作。这个函数最重要的部分是使用了 `pthread_mutex_lock` 和 `pthread_mutex_unlock` 进行了加锁和解锁，并使用了 `pthread_cond_wait` 来守候 `eventLoopThread` 中的 `eventLoop` 的变量。

[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 *eventLoop·
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->mut·
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 }
```

为什么要这么做呢？看一下子线程的代码你就会大致明白。子线程执行函数 `event_loop_thread_run` 一上来也是进行了加锁，之后初始化 `event_loop` 对象，当初始化完成之后，调用了 `pthread_cond_signal` 函数来通知此时阻塞在 `pthread_cond_wait` 上的主线程。这样，主线程就会从 `wait` 中苏醒，代码得以下行执行。子线程本身也通过调用 `event_loop_run` 进入了一个无限循环的事件分发执行体中，等待子线程 reactor 上注册过的事件发生。

 复制代码

```
1 void *event_loop_thread_run(void *arg) {
2     struct event_loop_thread *eventLoopThread = (struct event_loop_thread *) a
3
4     pthread_mutex_lock(&eventLoopThread->mutex);
5
6     // 初始化化 event loop, 之后通知主线程
7     eventLoopThread->eventLoop = event_loop_init();
8     yolanda_msgx("event loop thread init and signal, %s", eventLoopThread->thr
9     pthread_cond_signal(&eventLoopThread->cond);
10
11    pthread_mutex_unlock(&eventLoopThread->mutex);
12
13    // 子线程 event loop run
14    eventLoopThread->eventLoop->thread_name = eventLoopThread->thread_name;
15    event_loop_run(eventLoopThread->eventLoop);
16 }
```

可以看到，这里主线程和子线程共享的变量正是每个 `event_loop_thread` 的 `eventLoop` 对象，这个对象在初始化的时候为 `NULL`，只有当子线程完成了初始化，才变成一个非 `NULL` 的值，这个变化是子线程完成初始化的标志，也是信号量守护的变量。通过使用锁和信号量，解决了主线程和子线程同步的问题。当子线程完成初始化之后，主线程才会继续往下执行。

 复制代码

```
1 struct event_loop_thread {
2     struct event_loop *eventLoop;
3     pthread_t thread_tid;          /* thread ID */
4     pthread_mutex_t mutex;
5     pthread_cond_t cond;
6     char * thread_name;
7     long thread_count;      /* # connections handled */
8 };
```

你可能会问，主线程是循环在等待每个子线程完成初始化，如果进入第二个循环，等待第二个子线程完成初始化，而此时第二个子线程已经初始化完成了，该怎么办？

注意我们这里一上来是加锁的，只要取得了这把锁，同时发现 `event_loop_thread` 的 `eventLoop` 对象已经变成非 `NULL` 值，可以肯定第二个线程已经初始化，就直接释放锁往下执行了。

你可能还会问，在执行 `pthread_cond_wait` 的时候，需要持有那把锁么？这里，父线程在调用 `pthread_cond_wait` 函数之后，会立即进入睡眠，并释放持有的那把互斥锁。而当父线程再从 `pthread_cond_wait` 返回时（这是子线程通过 `pthread_cond_signal` 通知达成的），该线程再次持有那把锁。

增加已连接套接字事件到 **sub-reactor** 线程中

前面提到，主线程是一个 `main reactor` 线程，这个线程负责检测监听套接字上的事件，当有事件发生时，也就是一个连接已完成建立，如果我们有多个 `sub-reactor` 子线程，我们期望的结果是，把这个已连接套接字相关的 I/O 事件交给 `sub-reactor` 子线程负责检测。这样的好处是，`main reactor` 只负责连接套接字的建立，可以一直维持在一个非常高的处理效率，在多核的情况下，多个 `sub-reactor` 可以很好地利用上多核处理的优势。

不过，这里有一个令人苦恼的问题。

我们知道，`sub-reactor` 线程是一个无限循环的 `event loop` 执行体，在没有已注册事件发生的情况下，这个线程阻塞在 `event_dispatcher` 的 `dispatch` 上。你可以简单地认为阻塞在 `poll` 调用或者 `epoll_wait` 上，这种情况下，主线程如何能把已连接套接字交给 `sub-reactor` 子线程呢？

当然有办法。

如果我们能让 `sub-reactor` 线程从 `event_dispatcher` 的 `dispatch` 上返回，再让 `sub-reactor` 线程返回之后能够把新的已连接套接字事件注册上，这件事情就算完成了。

那如何让 `sub-reactor` 线程从 `event_dispatcher` 的 `dispatch` 上返回呢？答案是构建一个类似管道一样的描述字，让 `event_dispatcher` 注册该管道描述字，当我们想让 `sub-reactor` 线程苏醒时，往管道上发送一个字符就可以了。

在 event_loop_init 函数里，调用了 socketpair 函数创建了套接字对，这个套接字对的作用就是我刚刚说过的，往这个套接字的一端写时，另外一端就可以感知到读的事件。其实，这里也可以直接使用 UNIX 上的 pipe 管道，作用是一样的。

 复制代码

```
1 struct event_loop *event_loop_init() {  
2     ...  
3     //add the socketfd to event 这里创建的是套接字对，目的是为了唤醒子线程  
4     eventLoop->owner_thread_id = pthread_self();  
5     if (socketpair(AF_UNIX, SOCK_STREAM, 0, eventLoop->socketPair) < 0) {  
6         LOG_ERR("socketpair set failed");  
7     }  
8     eventLoop->is_handle_pending = 0;  
9     eventLoop->pending_head = NULL;  
10    eventLoop->pending_tail = NULL;  
11    eventLoop->thread_name = "main thread";  
12  
13    struct channel *channel = channel_new(eventLoop->socketPair[1], EVENT_READ);  
14    event_loop_add_channel_event(eventLoop, eventLoop->socketPair[0], channel);  
15  
16    return eventLoop;  
17 }
```

要特别注意的是文稿中的这句代码，这告诉 event_loop 的，是注册了 socketPair[1] 描述字上的 READ 事件，如果有 READ 事件发生，就调用 handleWakeup 函数来完成事件处理。

 复制代码

```
1 struct channel *channel = channel_new(eventLoop->socketPair[1], EVENT_READ, han
```

我们来看看这个 handleWakeup 函数：

事实上，这个函数就是简单的从 socketPair[1] 描述字上读取了一个字符而已，除此之外，它什么也没干。它的主要作用就是让子线程从 dispatch 的阻塞中苏醒。

 复制代码

```
1 int handleWakeup(void * data) {  
2     struct event_loop *eventLoop = (struct event_loop *) data;  
3     char one;  
4     ssize_t n = read(eventLoop->socketPair[1], &one, sizeof one);
```

```
5     if (n != sizeof one) {
6         LOG_ERR("handleWakeup failed");
7     }
8     yolanda_msgx("wakeup, %s", eventLoop->thread_name);
9 }
```

现在，我们再回过头看看，如果有新的连接产生，主线程是怎么操作的？在 `handle_connection_established` 中，通过 `accept` 调用获取了已连接套接字，将其设置为非阻塞套接字（标记），接下来调用 `thread_pool_get_loop` 获取一个 `event_loop`。`thread_pool_get_loop` 的逻辑非常简单，从 `thread_pool` 线程池中按照顺序挑选出一个线程来服务。接下来是创建了 `tcp_connection` 对象。

 复制代码

```
1 // 处理连接已建立的回调函数
2 int handle_connection_established(void *data) {
3     struct TCPserver *tcpServer = (struct TCPserver *) data;
4     struct acceptor *acceptor = tcpServer->acceptor;
5     int listenfd = acceptor->listen_fd;
6
7     struct sockaddr_in client_addr;
8     socklen_t client_len = sizeof(client_addr);
9     // 获取这个已建立的套接字，设置为非阻塞套接字
10    int connected_fd = accept(listenfd, (struct sockaddr *) &client_addr, &client_len);
11    make_nonblocking(connected_fd);
12
13    yolanda_msgx("new connection established, socket == %d", connected_fd);
14
15    // 从线程池里选择一个 eventloop 来服务这个新的连接套接字
16    struct event_loop *eventLoop = thread_pool_get_loop(tcpServer->threadPool)
17
18    // 为这个新建立套接字创建一个 tcp_connection 对象，并把应用程序的 callback 函数设置
19    struct tcp_connection *tcpConnection = tcp_connection_new(connected_fd, eventLoop,
20    //callback 内部使用
21    if (tcpServer->data != NULL) {
22        tcpConnection->data = tcpServer->data;
23    }
24    return 0;
25 }
```

在调用 `tcp_connection_new` 创建 `tcp_connection` 对象的代码里，可以看到先是创建了一个 `channel` 对象，并注册了 `READ` 事件，之后调用 `event_loop_add_channel_event` 方法往子线程中增加 `channel` 对象。

[复制代码](#)

```
1 tcp_connection_new(int connected_fd, struct event_loop *eventLoop,
2                     connection_completed_call_back connectionCompletedCallBack,
3                     connection_closed_call_back connectionClosedCallBack,
4                     message_call_back messageCallBack, write_completed_call_back
5                     ...
6 // 为新的连接对象创建可读事件
7 struct channel *channel1 = channel_new(connected_fd, EVENT_READ, handle_read,
8 tcpConnection->channel = channel1;
9
10 // 完成对 connectionCompleted 的函数回调
11 if (tcpConnection->connectionCompletedCallBack != NULL) {
12     tcpConnection->connectionCompletedCallBack(tcpConnection);
13 }
14
15 // 把该套集字对应的 channel 对象注册到 event_loop 事件分发器上
16 event_loop_add_channel_event(tcpConnection->eventLoop, connected_fd, tcpConnection->
17 return tcpConnection;
18 }
```

请注意，到现在为止的操作都是在主线程里执行的。下面的 `event_loop_do_channel_event` 也不例外，接下来的行为我期望你是熟悉的，那就是加解锁。

如果能够获取锁，主线程就会调用 `event_loop_channel_buffer_nolock` 往子线程的数据中增加需要处理的 channel event 对象。所有增加的 channel 对象以列表的形式维护在子线程的数据结构中。

接下来的部分是重点，如果当前增加 channel event 的不是当前 event loop 线程自己，就会调用 `event_loop_wakeup` 函数把 event_loop 子线程唤醒。唤醒的方法很简单，就是往刚刚的 `socketPair[0]` 上写一个字节，别忘了，`event_loop` 已经注册了 `socketPair[1]` 的可读事件。如果当前增加 channel event 的是当前 event loop 线程自己，则直接调用 `event_loop_handle_pending_channel` 处理新增加的 channel event 事件列表。

[复制代码](#)

```
1 int event_loop_do_channel_event(struct event_loop *eventLoop, int fd, struct cl
2 //get the lock
3 pthread_mutex_lock(&eventLoop->mutex);
4 assert(eventLoop->is_handle_pending == 0);
5 // 往该线程的 channel 列表里增加新的 channel
6 event_loop_channel_buffer_nolock(eventLoop, fd, channel1, type);
7 //release the lock
```

```
8     pthread_mutex_unlock(&eventLoop->mutex);
9     // 如果是主线程发起操作，则调用 event_loop_wakeup 唤醒子线程
10    if (!isInSameThread(eventLoop)) {
11        event_loop_wakeup(eventLoop);
12    } else {
13        // 如果是子线程自己，则直接可以操作
14        event_loop_handle_pending_channel(eventLoop);
15    }
16
17    return 0;
18 }
```

如果是 event_loop 被唤醒之后，接下来也会执行 event_loop_handle_pending_channel 函数。你可以看到在循环体内从 dispatch 退出之后，也调用了 event_loop_handle_pending_channel 函数。

复制代码

```
1 int event_loop_run(struct event_loop *eventLoop) {
2     assert(eventLoop != NULL);
3
4     struct event_dispatcher *dispatcher = eventLoop->eventDispatcher;
5
6     if (eventLoop->owner_thread_id != pthread_self()) {
7         exit(1);
8     }
9
10    yolanda_msgx("event loop run, %s", eventLoop->thread_name);
11    struct timeval timeval;
12    timeval.tv_sec = 1;
13
14    while (!eventLoop->quit) {
15        //block here to wait I/O event, and get active channels
16        dispatcher->dispatch(eventLoop, &timeval);
17
18        // 这里处理 pending channel, 如果是子线程被唤醒, 这个部分也会立即执行到
19        event_loop_handle_pending_channel(eventLoop);
20    }
21
22    yolanda_msgx("event loop end, %s", eventLoop->thread_name);
23    return 0;
24 }
```

event_loop_handle_pending_channel 函数的作用是遍历当前 event loop 里 pending 的 channel event 列表，将它们和 event_dispatcher 关联起来，从而修改感兴趣的事件集

合。

这里有一个点值得注意，因为 event loop 线程得到活动事件之后，会回调事件处理函数，这样像 onMessage 等应用程序代码也会在 event loop 线程执行，如果这里的业务逻辑过于复杂，就会导致 event_loop_handle_pending_channel 执行的时间偏后，从而影响 I/O 的检测。所以，将 I/O 线程和业务逻辑线程隔离，让 I/O 线程只负责处理 I/O 交互，让业务线程处理业务，是一个比较常见的做法。

总结

在这一讲里，我们重点讲解了框架中涉及多线程的两个重要问题，第一是主线程如何等待多个子线程完成初始化，第二是如何通知处于事件分发中的子线程有新的事件加入、删除、修改。第一个问题通过使用锁和信号量加以解决；第二个问题通过使用 socketpair，并将 socketpair 作为 channel 注册到 event loop 中来解决。

思考题

和往常一样，给你布置两道思考题：

第一道，你可以修改一下代码，让 sub-reactor 默认的线程个数为 cpu^*2 。

第二道，当前选择线程的算法是 round-robin 的算法，你觉得有没有改进的空间？如果改进的话，你可能会怎么做？

欢迎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，一起交流进步一下。



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 32 | 自己动手写高性能HTTP服务器（一）：设计和思路

下一篇 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 (3)

 写留言



MoonGod

2019-10-23

老师关于加锁这里有个疑问，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？为啥一定要加一把锁呢？

展开 ▼

作者回复: 好问题，我答疑统一回答吧。

 1



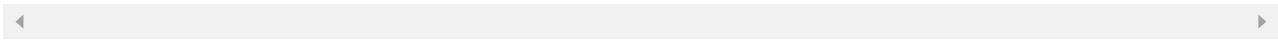
鱼向北游

2019-10-23

netty选子线程是两种算法，都是有个原子自增计数，如果线程数不是2的幂用取模，如果是就是按位与线程数减一

展开 ▼

作者回复: �恩，涨知识了，代码贴一个？



程序水果宝

2019-10-23

求完整的代码链接

展开 ▼

编辑回复: 代码链接请去详情页查看。

