# 处理大并发之 libevent demo详细分析（对比epoll） - 记事本

libevent默认情况下是单线程，每个线程有且仅有一个event_base，对应一个struct event_base结构体，以及赋予其上的事件管理器，用来安排托管给它的一系列的事件。

当有一个事件发生的时候，event_base会在合适的时间去调用绑定在这个事件上的函数，直到这个函数执行完成，然后在返回安排其他事件。需要注意的是：合适的时间并不是立即。

例如：

```
1. struct event_base *base;
2. base = event_base_new();
```

event_base_new对比epoll，可以理解为epoll里的epoll_create。

event_base内部有一个循环，循环阻塞在epoll调用上，当有一个事件发生的时候，才会去处理这个事件。其中，这个事件是被绑定在event_base上面的，每一个事件就会对应一个struct event，可以是监听的fd。

其中struct event 使用event_new 来创建和绑定，使用event_add来启用，例如：

```
1. struct event *listener_event;
2. listener_event = event_new(base, listener, EV_READ|EV_PERSIST, do_accept, (void*)base);
```

参数说明：

base：event_base类型，event_base_new的返回值

listener：监听的fd，listen的fd

EV_READ|EV_PERSIST：事件的类型及属性

do_accept：绑定的回调函数

(void*)base：给回调函数的参数

event_add(listener_event, NULL);

对比epoll：

event_new相当于epoll中的epoll_wait，其中的epoll里的while循环，在libevent里使用event_base_dispatch。

event_add相当于epoll中的epoll_ctl，参数是EPOLL_CTL_ADD，添加事件。

注：libevent支持的事件及属性包括(使用bitfield实现，所以要用 | 来让它们合体)
EV_TIMEOUT: 超时
EV_READ: 只要网络缓冲中还有数据，回调函数就会被触发
EV_WRITE: 只要塞给网络缓冲的数据被写完，回调函数就会被触发
EV_SIGNAL: POSIX信号量
EV_PERSIST: 不指定这个属性的话，回调函数被触发后事件会被删除
EV_ET: Edge-Trigger边缘触发，相当于EPOLL的ET模式

事件创建添加之后，就可以处理发生的事件了，相当于epoll里的epoll_wait,在libevent里使用event_base_dispatch启动event_base循环，直到不再有需要关注的事件。

有了上面的分析，结合之前做的epoll服务端程序，对于一个服务器程序，流程基本是这样的：

1. 创建socket，bind，listen，设置为非阻塞模式

2. 创建一个event_base，即

```
1. struct event_base *  event_base_new(void)
```

3. 创建一个event，将该socket托管给event_base，指定要监听的事件类型，并绑定上相应的回调函数(及需要给它的参数)。即

```
1. struct event *  event_new(struct event_base *base, evutil_socket_t fd, short events, void (*cb)(evutil_socket_t, short, void *), void *arg)
```

4. 启用该事件，即

```
1. int  event_add(struct event *ev, const struct timeval *tv)
```

5. 进入事件循环，即

```
1. int  event_base_dispatch(struct event_base *event_base)
```

有了这些知识储备，来看下官网上的demo，网址：http://www.wangafu.net/~nickm/libevent-book/01_intro.html，这里引用的例子是Example: A low-level ROT13 server with Libevent

首先来翻译下例子上面的一段话：

对于select函数来说，不同的操作系统有不同的代替函数，它包括：poll,epoll,kqueue,evport和/dev/poll。这些函数的性能都比select要好，其中epoll在IO中添加，删除，通知socket准备好方面性能复杂度为O(1)。

不幸的是，没有一个有效的接口是一个普遍存在的标准，linux下有epoll，BSDS有kqueue，Solaris 有evport和/dev/poll，等等。没有任何一个操作系统有它们中所有的，所以如果你想做一个轻便的高性能的异步应用程序，你就需要把这些接口抽象的封装起来，并且无论哪一个系统使用它都是最高效的。

这对于你来说就是最低级的libevent API，它提供了统一的接口取代了select，当它在计算机上运行的时候，使用了最有效的版本。

这里是ROT13服务器的另外一个版本，这次，他使用了libevent代替了select。这意味着我们不再使用fd_sets，取而代之的使用event_base添加和删除事件，它可能在select，poll，epoll，kqueue等中执行。

代码分析：

这是一个服务端的程序，可以处理客户端大并发的连接，当收到客户端的连接后，将收到的数据做了一个变换，如果是 'a'-'m'之间的字符，将其增加13，如果是 'n'-'z'之间的字符，将其减少13，其他字符不变，然后将转换后的数据发送给客户端。

例如：客户端发送：Client 0 send  Message!

服务端会回复：Pyvrag 0 fraq  Zrffntr!

在这个代码中没有使用bufferevent这个强大的东西，在一个结构体中自己管理了一个缓冲区。结构体为：

```
1. struct fd_state {
2.     char buffer[MAX_LINE];
3.     size_t buffer_used;
4.
5.     size_t n_written;
6.     size_t write_upto;
7.
8.     struct event *read_event;
9.     struct event *write_event;
10. };
```

代码中自己管理了一个缓冲区，用于存放接收到的数据，发送的数据将其转换后也放入该缓冲区中，代码晦涩难懂，我也是经过打日志分析后，才明白点，这个缓冲区自己还得控制好。但是libevent 2已经提供了一个神器bufferevent，我们在使用的过程中最好不要自己管理这个缓冲区，之所以分析这个代码，是为了熟悉libevent 做服务端程序的流程及原理。

下面是代码，加有部分注释和日志：

代码：lowlevel_libevent_server.c

```
1.
2.
3.
4. #include <netinet/in.h>
5.
6. #include <sys/socket.h>
7.
8. #include <fcntl.h>
9.
10. #include <event2/event.h>
11.
12. #include <assert.h>
13. #include <unistd.h>
14. #include <string.h>
15. #include <stdlib.h>
16. #include <stdio.h>
17. #include <errno.h>
18.
19. #define MAX_LINE 80
20.
21. void do_read(evutil_socket_t fd, short events, void *arg);
22. void do_write(evutil_socket_t fd, short events, void *arg);
23.
24. char rot13_char(char c)
```

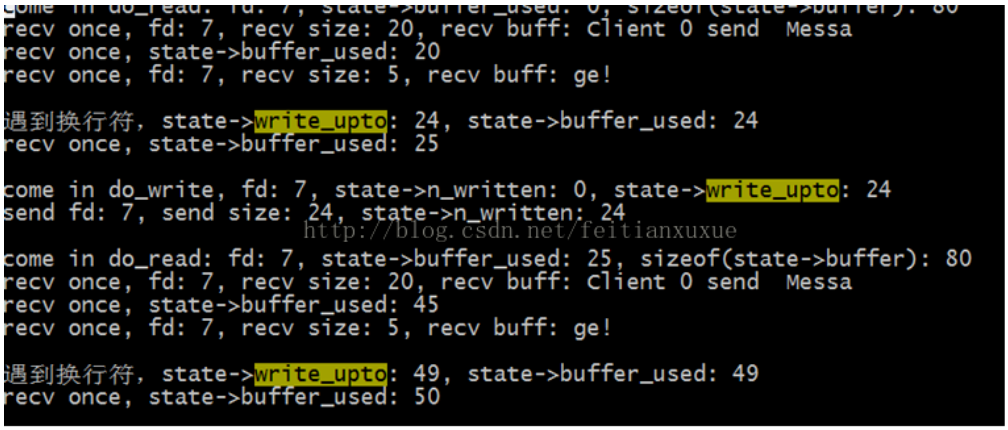
```
25. {
26.
27.
28.     if ((c >= 'a' && c <= 'm') || (c >= 'A' && c <= 'M'))
29.         return c + 13;
30.     else if ((c >= 'n' && c <= 'z') || (c >= 'N' && c <= 'Z'))
31.         return c - 13;
32.     else
33.         return c;
34. }
35.
36. struct fd_state {
37.     char buffer[MAX_LINE];
38.     size_t buffer_used;
39.
40.     size_t n_written;
41.     size_t write_upto;
42.
43.     struct event *read_event;
44.     struct event *write_event;
45. };
46.
47. struct fd_state * alloc_fd_state(struct event_base *base, evutil_socket_t fd)
48. {
49.     struct fd_state *state = malloc(sizeof(struct fd_state));
50.     if (!state)
51.         return NULL;
52.
53.     state->read_event = event_new(base, fd, EV_READ|EV_PERSIST, do_read, state);
54.     if (!state->read_event)
55.     {
56.         free(state);
57.         return NULL;
58.     }
59.
60.     state->write_event = event_new(base, fd, EV_WRITE|EV_PERSIST, do_write, state);
61.     if (!state->write_event)
62.     {
63.         event_free(state->read_event);
64.         free(state);
65.         return NULL;
66.     }
67.
68.     state->buffer_used = state->n_written = state->write_upto = 0;
69.
70.     assert(state->write_event);
71.     return state;
72. }
73.
74. void free_fd_state(struct fd_state *state)
75. {
76.     event_free(state->read_event);
77.     event_free(state->write_event);
78.     free(state);
79. }
80.
81. void do_read(evutil_socket_t fd, short events, void *arg)
82. {
83.     struct fd_state *state = arg;
84.     char buf[20];
85.     int i;
86.     ssize_t result;
87.     printf("\ncome in do_read: fd: %d, state->buffer_used: %d, sizeof(state->buffer): %d\n", fd, state->buffer_used, size
88. of(state->buffer));
89.     while (1)
90.     {
91.         assert(state->write_event);
92.         result = recv(fd, buf, sizeof(buf), 0);
93.         if (result <= 0)
94.             break;
95.         printf("recv once, fd: %d, recv size: %d, recv buff: %s\n", fd, result, buf);
96.
97.         for (i=0; i < result; ++i)
98.         {
99.             if (state->buffer_used < sizeof(state->buffer))
100.                 state->buffer[state->buffer_used++] = rot13_char(buf[i]);
101.
```

```
102.            if (buf[i] == '\n')
103.            {
104.                assert(state->write_event);
105.                event_add(state->write_event, NULL);
106.                state->write_upto = state->buffer_used;
107.                printf("遇到换行符，state->write_upto: %d, state->buffer_used: %d\n",state->write_upto, state->buffer_use
108. d);
109.            }
110.        }
111.        printf("recv once, state->buffer_used: %d\n", state->buffer_used);
112. }
113.
114.
115.    if (result == 0)
116.    {
117.        free_fd_state(state);
118.    }
119.    else if (result < 0)
120.    {
121.        if (errno == EAGAIN)
122.            return;
123.        perror("recv");
124.        free_fd_state(state);
125.    }
126. }
127.
128. void do_write(evutil_socket_t fd, short events, void *arg)
129. {
130.    struct fd_state *state = arg;
131.
132.    printf("\ncome in do_write, fd: %d, state->n_written: %d, state->write_upto: %d\n",fd, state->n_written, state->write
133. _upto);
134.    while (state->n_written < state->write_upto)
135.    {
136.        ssize_t result = send(fd, state->buffer + state->n_written, state->write_upto - state->n_written, 0);
137.        if (result < 0) {
138.            if (errno == EAGAIN)
139.                return;
140.            free_fd_state(state);
141.            return;
142.        }
143.        assert(result != 0);
144.
145.        state->n_written += result;
146.        printf("send fd: %d, send size: %d, state->n_written: %d\n", fd, result, state->n_written);
147.    }
148.
149.    if (state->n_written == state->buffer_used)
150.    {
151.        printf("state->n_written == state->buffer_used: %d\n", state->n_written);
152.        state->n_written = state->write_upto = state->buffer_used = 1;
153.        printf("state->n_written = state->write_upto = state->buffer_used = 1\n");
154.    }
155.
156.    event_del(state->write_event);
157. }
158.
159. void do_accept(evutil_socket_t listener, short event, void *arg)
160. {
161.    struct event_base *base = arg;
162.    struct sockaddr_storage ss;
163.    socklen_t slen = sizeof(ss);
164.    int fd = accept(listener, (struct sockaddr*)&ss, &slen);
165.    if (fd < 0)
166.    {
167.        perror("accept");
168.    }
169.    else if (fd > FD_SETSIZE)
170.    {
171.        close(fd);
172.    }
173.    else
174.    {
175.        struct fd_state *state;
176.        evutil_make_socket_nonblocking(fd);
177.        state = alloc_fd_state(base, fd);
```

```
178.        assert(state);
179.        assert(state->write_event);
180.        event_add(state->read_event, NULL);
181.    }
182. }
183.
184. void run(void)
185. {
186.    evutil_socket_t listener;
187.    struct sockaddr_in sin;
188.    struct event_base *base;
189.    struct event *listener_event;
190.
191.    base = event_base_new();
192.    if (!base)
193.        return;
194.
195.    sin.sin_family = AF_INET;
196.    sin.sin_addr.s_addr = 0;
197.    sin.sin_port = htons(8000);
198.
199.    listener = socket(AF_INET, SOCK_STREAM, 0);
200.    evutil_make_socket_nonblocking(listener);
201.
202. #ifndef WIN32
203.    {
204.        int one = 1;
205.        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));
206.    }
207. #endif
208.
209.    if (bind(listener, (struct sockaddr*)&sin, sizeof(sin)) < 0)
210.    {
211.        perror("bind");
212.        return;
213.        }
214.
215.
216.            if (listen(listener, 16)<0)
217.            {
218.                perror("listen");
219.                return;
220.            }
221.
222.            listener_event = event_new(base, listener, EV_READ|EV_PERSIST, do_accept, (void*)base);
223.
224.            event_add(listener_event, NULL);
225.
226.            event_base_dispatch(base);
227.        }
228.
229.        int main(int c, char **v)
230.        {
231.
232.
233.            run();
234.            return 0;
235.        }
```

编译：gcc -I/usr/include -o test lowlevel_libevent_server.c -L/usr/local/lib -levent

运行结果：


recv once, fd: 7, recv size: 20, recv buff: Client 0 send  Messa
recv once, state->buffer_used: 20
recv once, fd: 7, recv size: 5, recv buff: ge!
遇到换行符, state->write_upto: 24, state->buffer_used: 24
recv once, state->buffer_used: 25
come in do_write, fd: 7, state->n_written: 0, state->write_upto: 24
send fd: 7, send size: 24, state->n_written: 24
http://blog.csdn.net/feitianxuxue
come in do_read: fd: 7, state->buffer_used: 25, sizeof(state->buffer): 80
recv once, fd: 7, recv size: 20, recv buff: Client 0 send  Messa
recv once, state->buffer_used: 45
recv once, fd: 7, recv size: 5, recv buff: ge!
遇到换行符, state->write_upto: 49, state->buffer_used: 49
recv once, state->buffer_used: 50