<https://blog.csdn.net/luotuo44/article/details/39344743>

**转载请注明出处：**[**http://blog.csdn.net/luotuo44/article/details/39344743**](http://blog.csdn.net/luotuo44/article/details/39344743)

        和之前的《[Libevent工作流程探究](http://blog.csdn.net/luotuo44/article/details/38501341)》一样，这里也是用一个例子来探究bufferevent的工作流程。具体的例子可以参考《[Libevent使用例子，从简单到复杂](http://blog.csdn.net/luotuo44/article/details/34416429)》，这里就不列出了。其实要做的例子也就是bufferevent\_socket\_new、bufferevent\_setcb、bufferevent\_enable这几个函数。

        因为本文会用到《[Libevent工作流程探究](http://blog.csdn.net/luotuo44/article/details/38501341)》中提到的说法，比如将一个event插入到event\_base中。所以读者最好先读一下那篇博文。此外，因为bufferevent结构体本身会使用evbuffer结构体和还会调用相应的一些操作，所以读者还应该先阅读《[evbuffer结构与基本操作](http://blog.csdn.net/luotuo44/article/details/39290721)》和《[更多evbuffer操作函数](http://blog.csdn.net/luotuo44/article/details/39325447)》。

# ****bufferevent结构体：****

        bufferevent其实也就是在event\_base的基础上再进行一层封装，其本质还是离不开event和event\_base，从bufferevent的结构体就可以看到这一点。

        bufferevent结构体中有两个event，分别用来监听同一个fd的可读事件和可写事件。为什么不用一个event同时监听可读和可写呢？这是因为监听可写是困难的，[下面](http://blog.csdn.net/luotuo44/article/details/39344743#t7)会说到原因。读者也可以自问一下，自己之前有没有试过用最原始的event监听一个fd的可写。

        由于socket 是全双工的，所以在bufferevent结构体中，也有两个evbuffer成员，分别是读缓冲区和写缓冲区。 bufferevent结构体定义如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent\_struct.h文件
2. **struct** bufferevent {
3. **struct** event\_base \*ev\_base;
5. //操作结构体，成员有一些函数指针。类似struct eventop结构体
6. **const** **struct** bufferevent\_ops \*be\_ops;
8. **struct** event ev\_read;//读事件event
9. **struct** event ev\_write;//写事件event
11. **struct** evbuffer \*input;//读缓冲区
13. **struct** evbuffer \*output; //写缓冲区
15. **struct** event\_watermark wm\_read;//读水位
16. **struct** event\_watermark wm\_write;//写水位

19. bufferevent\_data\_cb readcb;//可读时的回调函数指针
20. bufferevent\_data\_cb writecb;//可写时的回调函数指针
21. bufferevent\_event\_cb errorcb;//错误发生时的回调函数指针
22. **void** \*cbarg;//回调函数的参数
24. **struct** timeval timeout\_read;//读事件event的超时值
25. **struct** timeval timeout\_write;//写事件event的超时值
27. /\*\* Events that are currently enabled: currently EV\_READ and EV\_WRITE
28. are supported. \*/
29. **short** enabled;
30. };

        如果看过Libevent的参考手册的话，应该还会知道bufferevent除了用于socket外，还可以用于socketpair 和 filter。如果用面向对象的思维，应从这个三个应用中抽出相同的部分作为父类，然后派生出三个子类。

        Libevent虽然是用C语言写的，不过它还是提取出一些公共部分，然后定义一个bufferevent\_private结构体，用于保存这些公共部分成员。从集合的角度来说，bufferevent\_private应该是bufferevent的一个子集，即一部分。但在Libevent中，bufferevent确实bufferevent\_private的一个成员。下面是bufferevent\_private结构体。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent-internal.h文件
2. **struct** bufferevent\_private {
3. **struct** bufferevent bev;
5. //设置input evbuffer的高水位时，需要一个evbuffer回调函数配合工作
6. **struct** evbuffer\_cb\_entry \*read\_watermarks\_cb;
8. /\*\* If set, we should free the lock when we free the bufferevent. \*/
9. //锁是Libevent自动分配的，还是用户分配的
10. unsigned own\_lock : 1;
12. ...
14. //这个socket是否处理正在连接服务器状态
15. unsigned connecting : 1;
16. //标志连接被拒绝
17. unsigned connection\_refused : 1;
19. //标志是什么原因把 读 挂起来
20. bufferevent\_suspend\_flags read\_suspended;
21. //标志是什么原因把 写 挂起来
22. bufferevent\_suspend\_flags write\_suspended;
24. **enum** bufferevent\_options options;
25. **int** refcnt;// bufferevent的引用计数
27. //锁变量
28. **void** \*lock;
29. };

# ****新建一个bufferevent：****

        函数bufferevent\_socket\_new可以完成这个工作。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent-internal.h文件
2. **struct** bufferevent\_ops {
3. **const** **char** \*type;//类型名称
5. off\_t mem\_offset;//成员bev的偏移量
7. //启动。将event加入到event\_base中
8. **int** (\*enable)(**struct** bufferevent \*, **short**);
10. //关闭。将event从event\_base中删除
11. **int** (\*disable)(**struct** bufferevent \*, **short**);
12. //销毁
13. **void** (\*destruct)(**struct** bufferevent \*);
14. //调整event的超时值
15. **int** (\*adj\_timeouts)(**struct** bufferevent \*);
16. /\*\* Called to flush data. \*/
17. **int** (\*flush)(**struct** bufferevent \*, **short**, **enum** bufferevent\_flush\_mode);
18. //获取成员的值。具体看实现
19. **int** (\*ctrl)(**struct** bufferevent \*, **enum** bufferevent\_ctrl\_op, **union** bufferevent\_ctrl\_data \*);
20. };

23. //bufferevent\_sock.c文件
24. **const** **struct** bufferevent\_ops bufferevent\_ops\_socket = {
25. "socket",
26. evutil\_offsetof(**struct** bufferevent\_private, bev),
27. be\_socket\_enable,
28. be\_socket\_disable,
29. be\_socket\_destruct,
30. be\_socket\_adj\_timeouts,
31. be\_socket\_flush,
32. be\_socket\_ctrl,
33. };
35. //由于有几个不同类型的bufferevent，而且它们的enable、disable等操作是不同的。所以
36. //需要的一些函数指针指明某个类型的bufferevent应该使用哪些操作函数。结构体bufferevent\_ops\_socket
37. //就应运而生。对于socket，其操作函数如上。
39. //bufferevent\_sock.c文件
40. **struct** bufferevent \*
41. bufferevent\_socket\_new(**struct** event\_base \*base, evutil\_socket\_t fd,
42. **int** options)
43. {
44. **struct** bufferevent\_private \*bufev\_p;
45. **struct** bufferevent \*bufev;
47. ...//win32
49. //结构体内存清零，所有成员都为0
50. **if** ((bufev\_p = mm\_calloc(1, **sizeof**(**struct** bufferevent\_private)))== NULL)
51. **return** NULL;
53. //如果options中需要线程安全，那么就会申请锁
54. //会新建一个输入和输出缓存区
55. **if** (bufferevent\_init\_common(bufev\_p, base, &bufferevent\_ops\_socket,
56. options) < 0) {
57. mm\_free(bufev\_p);
58. **return** NULL;
59. }
60. bufev = &bufev\_p->bev;
61. //设置将evbuffer的数据向fd传
62. evbuffer\_set\_flags(bufev->output, EVBUFFER\_FLAG\_DRAINS\_TO\_FD);
64. //将fd与event相关联。同一个fd关联两个event
65. event\_assign(&bufev->ev\_read, bufev->ev\_base, fd,
66. EV\_READ|EV\_PERSIST, bufferevent\_readcb, bufev);
67. event\_assign(&bufev->ev\_write, bufev->ev\_base, fd,
68. EV\_WRITE|EV\_PERSIST, bufferevent\_writecb, bufev);
70. //设置evbuffer的回调函数，使得外界给写缓冲区添加数据时，能触发
71. //写操作，这个回调对于写事件的监听是很重要的
72. evbuffer\_add\_cb(bufev->output, bufferevent\_socket\_outbuf\_cb, bufev);
74. //冻结读缓冲区的尾部，未解冻之前不能往读缓冲区追加数据
75. //也就是说不能从socket fd中读取数据
76. evbuffer\_freeze(bufev->input, 0);
78. //冻结写缓冲区的头部，未解冻之前不能把写缓冲区的头部数据删除
79. //也就是说不能把数据写到socket fd
80. evbuffer\_freeze(bufev->output, 1);
82. **return** bufev;
83. }

        留意函数里面的evbuffer\_add\_cb调用，后面会说到。

        函数在最后面会冻结两个缓冲区。其实，虽然这里冻结了，但实际上Libevent在读数据或者写数据之前会解冻的读完或者写完数据后，又会马上冻结。这主要防止数据被意外修改。用户一般不会直接调用evbuffer\_freeze或者evbuffer\_unfreeze函数。一切的冻结和解冻操作都由Libevent内部完成。还有一点要注意，因为这里只是把写缓冲区的头部冻结了。所以还是可以往写缓冲区的尾部追加数据。同样，此时也是可以从读缓冲区读取数据。这个是必须的。因为在Libevent内部不解冻的时候，用户需要从读缓冲区中获取数据(这相当于从socket fd中读取数据)，用户也需要把数据写到写缓冲区中(这相当于把数据写入到socket fd中)。

        在bufferevent\_socket\_new函数里面会调用函数bufferevent\_init\_common完成公有部分的初始化。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent.c文件
2. **int**
3. bufferevent\_init\_common(**struct** bufferevent\_private \*bufev\_private,
4. **struct** event\_base \*base,
5. **const** **struct** bufferevent\_ops \*ops,
6. **enum** bufferevent\_options options)
7. {
8. **struct** bufferevent \*bufev = &bufev\_private->bev;
10. //分配输入缓冲区
11. **if** (!bufev->input) {
12. **if** ((bufev->input = evbuffer\_new()) == NULL)
13. **return** -1;
14. }
16. //分配输出缓冲区
17. **if** (!bufev->output) {
18. **if** ((bufev->output = evbuffer\_new()) == NULL) {
19. evbuffer\_free(bufev->input);
20. **return** -1;
21. }
22. }
24. bufev\_private->refcnt = 1;//引用次数为1
25. bufev->ev\_base = base;
27. /\* Disable timeouts. \*/
28. //默认情况下,读和写event都是不支持超时的
29. evutil\_timerclear(&bufev->timeout\_read);
30. evutil\_timerclear(&bufev->timeout\_write);
32. bufev->be\_ops = ops;
34. /\*
35. \* Set to EV\_WRITE so that using bufferevent\_write is going to
36. \* trigger a callback.  Reading needs to be explicitly enabled
37. \* because otherwise no data will be available.
38. \*/
39. //可写是默认支持的
40. bufev->enabled = EV\_WRITE;
42. #ifndef \_EVENT\_DISABLE\_THREAD\_SUPPORT
43. **if** (options & BEV\_OPT\_THREADSAFE) {
44. //申请锁。
45. **if** (bufferevent\_enable\_locking(bufev, NULL) < 0) {
46. /\* cleanup \*/
47. evbuffer\_free(bufev->input);
48. evbuffer\_free(bufev->output);
49. bufev->input = NULL;
50. bufev->output = NULL;
51. **return** -1;
52. }
53. }
54. #endif
55. ...//延迟调用的初始化，一般不需要用到
57. bufev\_private->options = options;
59. //将evbuffer和bufferevent相关联
60. evbuffer\_set\_parent(bufev->input, bufev);
61. evbuffer\_set\_parent(bufev->output, bufev);
63. **return** 0;
64. }

        代码中可以看到，默认是enable  EV\_WRITE的。

# 设置回调函数：

        函数bufferevent\_setcb完成这个工作。该函数相当简单，也就是进行一些赋值操作。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent.c文件
2. **void**
3. bufferevent\_setcb(**struct** bufferevent \*bufev,
4. bufferevent\_data\_cb readcb, bufferevent\_data\_cb writecb,
5. bufferevent\_event\_cb eventcb, **void** \*cbarg)
6. {
7. //bufferevent结构体内部有一个锁变量
8. BEV\_LOCK(bufev);
10. bufev->readcb = readcb;
11. bufev->writecb = writecb;
12. bufev->errorcb = eventcb;
14. bufev->cbarg = cbarg;
15. BEV\_UNLOCK(bufev);
16. }

        如果不想设置某个操作的回调函数，直接设置为NULL即可。

# 令bufferevent可以工作:

        相信读者也知道，即使调用了bufferevent\_socket\_new和bufferevent\_setcb，这个bufferevent还是不能工作，必须调用bufferevent\_enable。为什么会这样的呢？

        如果看过之前的那些博文，相信读者知道，一个event能够工作，不仅仅需要new出来，还要调用event\_add函数，把这个event添加到event\_base中。在本文前面的代码中，并没有看到event\_add函数的调用。所以还需要调用一个函数，把event添加到event\_base中。函数bufferevent\_enable就是完成这个工作的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent.c文件
2. **int**
3. bufferevent\_enable(**struct** bufferevent \*bufev, **short** event)
4. {
5. **struct** bufferevent\_private \*bufev\_private =
6. EVUTIL\_UPCAST(bufev, **struct** bufferevent\_private, bev);
7. **short** impl\_events = event;
8. **int** r = 0;
10. //增加引用并加锁
11. //增加引用是为了防止其他线程调用bufferevent\_free，释放了bufferevent
12. \_bufferevent\_incref\_and\_lock(bufev);
14. //挂起了读，此时不能监听读事件
15. **if** (bufev\_private->read\_suspended)
16. impl\_events &= ~EV\_READ;
18. //挂起了写，此时不能监听写事情
19. **if** (bufev\_private->write\_suspended)
20. impl\_events &= ~EV\_WRITE;
22. bufev->enabled |= event;
24. //调用对应类型的enbale函数。因为不同类型的bufferevent有不同的enable函数
25. **if** (impl\_events && bufev->be\_ops->enable(bufev, impl\_events) < 0)
26. r = -1;
28. //减少引用并解锁
29. \_bufferevent\_decref\_and\_unlock(bufev);
30. **return** r;
31. }

        上面代码可以看到，最终会调用对应bufferevent类型的enable函数，对于socket bufferevent，其enable函数是be\_socket\_enable，代码如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent.c文件
2. **int**
3. \_bufferevent\_add\_event(**struct** event \*ev, **const** **struct** timeval \*tv)
4. {
5. **if** (tv->tv\_sec == 0 && tv->tv\_usec == 0)
6. **return** event\_add(ev, NULL);
7. **else**
8. **return** event\_add(ev, tv);
9. }

12. //bufferevent\_sock.c文件
13. #define be\_socket\_add(ev, t)            \
14. \_bufferevent\_add\_event((ev), (t))

17. **static** **int**
18. be\_socket\_enable(**struct** bufferevent \*bufev, **short** event)
19. {
20. **if** (event & EV\_READ) {
21. **if** (be\_socket\_add(&bufev->ev\_read,&bufev->timeout\_read) == -1)
22. **return** -1;
23. }
24. **if** (event & EV\_WRITE) {
25. **if** (be\_socket\_add(&bufev->ev\_write,&bufev->timeout\_write) == -1)
26. **return** -1;
27. }
28. **return** 0;
29. }

        如果读者熟悉Libevent的超时事件，那么可以知道Libevent是在event\_add函数里面确定一个event的超时的。上面代码也展示了这一点，如果读或者写event设置了超时(即其超时值不为0)，那么就会作为参数传给event\_add函数。如果读者不熟悉的Libevent的超时事件的话，可以参考《[超时event的处理](http://blog.csdn.net/luotuo44/article/details/38637671)》。

        用户可以调用函数bufferevent\_set\_timeouts，设置读或者写事件的超时。代码如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent.c文件
2. **int**
3. bufferevent\_set\_timeouts(**struct** bufferevent \*bufev,
4. **const** **struct** timeval \*tv\_read,
5. **const** **struct** timeval \*tv\_write)
6. {
7. **int** r = 0;
8. BEV\_LOCK(bufev);
9. **if** (tv\_read) {
10. bufev->timeout\_read = \*tv\_read;
11. } **else** {
12. evutil\_timerclear(&bufev->timeout\_read);
13. }
14. **if** (tv\_write) {
15. bufev->timeout\_write = \*tv\_write;
16. } **else** {
17. evutil\_timerclear(&bufev->timeout\_write);
18. }
20. **if** (bufev->be\_ops->adj\_timeouts)
21. r = bufev->be\_ops->adj\_timeouts(bufev);
22. BEV\_UNLOCK(bufev);
24. **return** r;
25. }

28. //bufferevent\_sock.c文件
29. **static** **int**
30. be\_socket\_adj\_timeouts(**struct** bufferevent \*bufev)
31. {
32. **int** r = 0;
33. //用户监听了读事件
34. **if** (event\_pending(&bufev->ev\_read, EV\_READ, NULL))
35. **if** (be\_socket\_add(&bufev->ev\_read, &bufev->timeout\_read) < 0)
36. r = -1;
38. //用户监听了写事件
39. **if** (event\_pending(&bufev->ev\_write, EV\_WRITE, NULL)) {
40. **if** (be\_socket\_add(&bufev->ev\_write, &bufev->timeout\_write) < 0)
41. r = -1;
42. }
43. **return** r;
44. }

        从上面代码可以看到：用户不仅仅可以设置超时值，还可以修改超时值，也是通过这个函数进行修的。当然也是可以删除超时的，直接把超时参数设置成NULL即可。

        至此，已经完成了bufferevent的初始化工作，只需调用event\_base\_dispatch函数，启动发动机就可以工作了。

# ****处理读事件：****

        接下来的任务：底层的socket fd接收数据后，bufferevent是怎么工作的。

## ****读事件的水位：****

        在讲解读事件之前，先来看一下水位问题，函数bufferevent\_setwatermark可以设置读和写的水位。这里只讲解读事件的水位。

        水位有两个：低水位和高水位。

        低水位比较容易懂，就是当可读的数据量到达这个低水位后，才会调用用户设置的回调函数。比如用户想每次读取100字节，那么就可以把低水位设置为100。当可读数据的字节数小于100时，即使有数据都不会打扰用户(即不会调用用户设置的回调函数)。可读数据大于等于100字节后，才会调用用户的回调函数。

        高水位是什么呢？其实，这和用户的回调函数没有关系。它的意义是：把读事件的evbuffer的数据量限制在高水位之下。比如，用户认为读缓冲区不能太大(太大的话，链表会很长)。那么用户就会设置读事件的高水位。当读缓冲区的数据量达到这个高水位后，即使socket fd还有数据没有读，也不会读进这个读缓冲区里面。一句话说，就是控制evbuffer的大小。

        虽然控制了evbuffer的大小，但socket fd可能还有数据。有数据就会触发可读事件，但处理可读的时候，又会发现设置了高水位，不能读取数据evbuffer。socket fd的数据没有被读完，又触发……。这个貌似是一个死循环。实际上是不会出现这个死循环的，因为Libevent发现evbuffer的数据量到达高水位后，就会把可读事件给挂起来，让它不能再触发了。Libevent使用函数bufferevent\_wm\_suspend\_read把监听读事件的event挂起来。下面看一下Libevent是怎么把一个event挂起来的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent-internal.h文件
2. #define bufferevent\_wm\_suspend\_read(b) \
3. bufferevent\_suspend\_read((b), BEV\_SUSPEND\_WM)

6. //bufferevent.c文件
7. **void**
8. bufferevent\_suspend\_read(**struct** bufferevent \*bufev, bufferevent\_suspend\_flags what)
9. {
10. **struct** bufferevent\_private \*bufev\_private =
11. EVUTIL\_UPCAST(bufev, **struct** bufferevent\_private, bev);
12. BEV\_LOCK(bufev);
13. **if** (!bufev\_private->read\_suspended)//不能挂多次
14. bufev->be\_ops->disable(bufev, EV\_READ);//实际调用be\_socket\_disable函数
15. bufev\_private->read\_suspended |= what;//因何而被挂起
16. BEV\_UNLOCK(bufev);
17. }

20. //bufferevent\_sock.c文件
21. **static** **int**
22. be\_socket\_disable(**struct** bufferevent \*bufev, **short** event)
23. {
24. **struct** bufferevent\_private \*bufev\_p =
25. EVUTIL\_UPCAST(bufev, **struct** bufferevent\_private, bev);
26. **if** (event & EV\_READ) {
27. **if** (event\_del(&bufev->ev\_read) == -1)
28. **return** -1;
29. }
30. /\* Don't actually disable the write if we are trying to connect. \*/
31. **if** ((event & EV\_WRITE) && ! bufev\_p->connecting) {
32. **if** (event\_del(&bufev->ev\_write) == -1)//删掉这个event
33. **return** -1;
34. }
35. **return** 0;
36. }

        居然是直接删除这个监听读事件的event，真的是挂了!!!

        看来不能随便设置高水位，因为它会暂停读。如果只想设置低水位而不想设置高水位，那么在调用bufferevent\_setwatermark函数时，高水位的参数设为0即可。

        那么什么时候取消挂起，让bufferevent可以继续读socket 数据呢？从高水位的意义来说，当然是当evbuffer里面的数据量小于高水位时，就能再次读取socket数据了。现在来看一下Libevent是怎么恢复读的。看一下设置水位的函数bufferevent\_setwatermark吧，它进行了一些为高水位埋下了一个回调函数。对，就是evbuffer的回调函数。[前一篇博文](http://blog.csdn.net/luotuo44/article/details/39325447#t6)说到，当evbuffer里面的数据添加或者删除时，是会触发一些回调函数的。当用户移除evbuffer的一些数据量时，Libevent就会检查这个evbuffer的数据量是否小于高水位，如果小于的话，那么就恢复 读事件。

        不说这么多了，上代码。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent.c文件
2. **void**
3. bufferevent\_setwatermark(**struct** bufferevent \*bufev, **short** events,
4. **size\_t** lowmark, **size\_t** highmark)
5. {
6. **struct** bufferevent\_private \*bufev\_private =
7. EVUTIL\_UPCAST(bufev, **struct** bufferevent\_private, bev);
9. BEV\_LOCK(bufev);
11. **if** (events & EV\_READ) {
12. bufev->wm\_read.low = lowmark;
13. bufev->wm\_read.high = highmark;
15. **if** (highmark) {//高水位
16. /\* There is now a new high-water mark for read.
17. enable the callback if needed, and see if we should
18. suspend/bufferevent\_wm\_unsuspend. \*/
20. //还没设置高水位的回调函数
21. **if** (bufev\_private->read\_watermarks\_cb == NULL) {
22. bufev\_private->read\_watermarks\_cb =
23. evbuffer\_add\_cb(bufev->input,
24. bufferevent\_inbuf\_wm\_cb,
25. bufev);//添加回调函数
26. }
27. evbuffer\_cb\_set\_flags(bufev->input,
28. bufev\_private->read\_watermarks\_cb,
29. EVBUFFER\_CB\_ENABLED|EVBUFFER\_CB\_NODEFER);
31. //设置(修改)高水位时，evbuffer的数据量已经超过了水位值
32. //可能是把之前的高水位调高或者调低
33. //挂起操作和取消挂起操作都是幂等的(即多次挂起的作用等同于挂起一次)
34. **if** (evbuffer\_get\_length(bufev->input) > highmark)
35. bufferevent\_wm\_suspend\_read(bufev);
36. **else** **if** (evbuffer\_get\_length(bufev->input) < highmark)//调低了
37. bufferevent\_wm\_unsuspend\_read(bufev);
38. } **else** {
39. //高水位值等于0，那么就要取消挂起 读事件
40. //取消挂起操作是幂等的
41. /\* There is now no high-water mark for read. \*/
42. **if** (bufev\_private->read\_watermarks\_cb)
43. evbuffer\_cb\_clear\_flags(bufev->input,
44. bufev\_private->read\_watermarks\_cb,
45. EVBUFFER\_CB\_ENABLED);
46. bufferevent\_wm\_unsuspend\_read(bufev);
47. }
48. }
49. BEV\_UNLOCK(bufev);
50. }

        这个函数，不仅仅为高水位设置回调函数，还会检查当前evbuffer的数据量是否超过了高水位。因为这个设置水位函数可能是在bufferevent工作一段时间后才添加的，所以evbuffer是有可能已经有数据的了，因此需要检查。如果超过了水位值，那么就需要挂起读。当然也存在另外一种可能：用户之前设置过了一个比较大的高水位，挂起了读。现在发现错了，就把高水位调低一点，此时就需要恢复读。

        现在假设用户移除了一些evbuffer的数据，进而触发了evbuffer的回调函数，当然也就调用了函数bufferevent\_inbuf\_wm\_cb。下面看一下这个函数是怎么恢复读的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent.c文件
2. **static** **void**
3. bufferevent\_inbuf\_wm\_cb(**struct** evbuffer \*buf,
4. **const** **struct** evbuffer\_cb\_info \*cbinfo,
5. **void** \*arg)
6. {
7. **struct** bufferevent \*bufev = arg;
8. **size\_t** size;
10. size = evbuffer\_get\_length(buf);
12. **if** (size >= bufev->wm\_read.high)
13. bufferevent\_wm\_suspend\_read(bufev);
14. **else**
15. bufferevent\_wm\_unsuspend\_read(bufev);
16. }
18. //bufferevent-internal.h文件
19. #define bufferevent\_wm\_unsuspend\_read(b) \
20. bufferevent\_unsuspend\_read((b), BEV\_SUSPEND\_WM)
22. //bufferevent.c文件
23. **void**
24. bufferevent\_unsuspend\_read(**struct** bufferevent \*bufev, bufferevent\_suspend\_flags what)
25. {
26. **struct** bufferevent\_private \*bufev\_private =
27. EVUTIL\_UPCAST(bufev, **struct** bufferevent\_private, bev);
29. BEV\_LOCK(bufev);
30. bufev\_private->read\_suspended &= ~what;
31. **if** (!bufev\_private->read\_suspended && (bufev->enabled & EV\_READ))
32. bufev->be\_ops->enable(bufev, EV\_READ);//重新把event插入到event\_base中
33. BEV\_UNLOCK(bufev);
34. }

        因为用户可以手动为这个evbuffer添加数据，此时也会调用bufferevent\_inbuf\_wm\_cb函数。此时就要检查evbuffer的数据量是否已经超过高水位了，而不能仅仅检查是否低于高水位。

        高水位导致读的挂起和之后读的恢复，一切工作都是由Libevent内部完成的，用户不用做任何工作。

## ****从socket中读取数据：****

        从前面的一系列博文可以知道，如果一个socket可读了，那么监听可读事件的event的回调函数就会被调用。这个回调函数是在bufferevent\_socket\_new函数中被Libevent内部设置的，设置为bufferevent\_readcb函数，用户并不知情。

        当socket有数据可读时，Libevent就会监听到，然后调用bufferevent\_readcb函数处理。该函数会调用evbuffer\_read函数，把数据从socket fd中读取到evbuffer中。然后再调用用户在bufferevent\_setcb函数中设置的读事件回调函数。所以，当用户的读事件回调函数被调用时，数据已经在evbuffer中了，用户拿来就用，无需调用read这类会阻塞的函数。

        下面看一下bufferevent\_readcb函数的具体实现。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. **static** **void**
2. bufferevent\_readcb(evutil\_socket\_t fd, **short** event, **void** \*arg)
3. {
4. **struct** bufferevent \*bufev = arg;
5. **struct** bufferevent\_private \*bufev\_p =
6. EVUTIL\_UPCAST(bufev, **struct** bufferevent\_private, bev);
7. **struct** evbuffer \*input;
8. **int** res = 0;
9. **short** what = BEV\_EVENT\_READING;
10. ev\_ssize\_t howmuch = -1, readmax=-1;
12. \_bufferevent\_incref\_and\_lock(bufev);
14. **if** (event == EV\_TIMEOUT) {
15. /\* Note that we only check for event==EV\_TIMEOUT. If
16. \* event==EV\_TIMEOUT|EV\_READ, we can safely ignore the
17. \* timeout, since a read has occurred \*/
18. what |= BEV\_EVENT\_TIMEOUT;
19. **goto** error;
20. }
22. input = bufev->input;

25. //用户设置了高水位
26. **if** (bufev->wm\_read.high != 0) {
27. howmuch = bufev->wm\_read.high - evbuffer\_get\_length(input);
28. /\* we somehow lowered the watermark, stop reading \*/
29. **if** (howmuch <= 0) {
30. bufferevent\_wm\_suspend\_read(bufev);
31. **goto** done;
32. }
33. }
35. //因为用户可以限速，所以这么要检测最大的可读大小。
36. //如果没有限速的话，那么将返回16384字节，即16K
37. //默认情况下是没有限速的。
38. readmax = \_bufferevent\_get\_read\_max(bufev\_p);
39. **if** (howmuch < 0 || howmuch > readmax) /\* The use of -1 for "unlimited"
40. \* uglifies this code. XXXX \*/
41. howmuch = readmax;
43. //一些原因导致读 被挂起，比如加锁了。
44. **if** (bufev\_p->read\_suspended)
45. **goto** done;
47. //解冻，使得可以在input的后面追加数据
48. evbuffer\_unfreeze(input, 0);
49. res = evbuffer\_read(input, fd, (**int**)howmuch); //从socket fd中读取数据
50. evbuffer\_freeze(input, 0);//冻结
52. **if** (res == -1) {
53. **int** err = evutil\_socket\_geterror(fd);
54. **if** (EVUTIL\_ERR\_RW\_RETRIABLE(err))//EINTER or EAGAIN
55. **goto** reschedule;
57. //不是 EINTER or EAGAIN 这两个可以重试的错误，那么就应该是其他致命的错误
58. //此时，应该报告给用户
59. what |= BEV\_EVENT\_ERROR;/\*\*< unrecoverable error encountered \*/
60. } **else** **if** (res == 0) {//断开了连接
61. what |= BEV\_EVENT\_EOF;
62. }
64. **if** (res <= 0)
65. **goto** error;
67. //速率相关的操作
68. \_bufferevent\_decrement\_read\_buckets(bufev\_p, res);

71. //evbuffer的数据量大于低水位值。
72. **if** (evbuffer\_get\_length(input) >= bufev->wm\_read.low)
73. \_bufferevent\_run\_readcb(bufev);//调用用户设置的回调函数
75. **goto** done;
77. reschedule:
78. **goto** done;
80. error:
81. //把监听可读事件的event从event\_base的事件队列中删除掉.event\_del
82. bufferevent\_disable(bufev, EV\_READ);//会调用be\_socket\_disable函数
83. \_bufferevent\_run\_eventcb(bufev, what);//会调用用户设置的错误处理函数
85. done:
86. \_bufferevent\_decref\_and\_unlock(bufev);
87. }

        细心的读者可能会发现：对用户的读事件回调函数的触发是边缘触发的。这也就要求，在回调函数中，用户应该尽可能地把evbuffer的所有数据都读出来。如果想等到下一次回调时再读，那么需要等到下一次socketfd接收到数据才会触发用户的回调函数。如果之后socket fd一直收不到任何数据，那么即使evbuffer还有数据，用户的回调函数也不会被调用了。

# ****处理写事件：****

        对一个可读事件进行监听是比较容易的，但对于一个可写事件进行监听则比较困难。为什么呢？因为可读监听是监听fd的读缓冲区是否有数据了，如果没有数据那么就一直等待。对于可写，首先要明白“什么是可写”，可写就是fd的写缓冲区(这个缓冲区在内核)还没满，可以往里面放数据。这就有一个问题，如果写缓冲区没有满，那么就一直是可写状态。如果一个event监听了可写事件，那么这个event就会一直被触发（死循环）。因为一般情况下，如果不是发大量的数据这个写缓冲区是不会满的。

        也就是说，不能监听可写事件。但我们确实要往fd中写数据，那怎么办？Libevent的做法是：当我们确实要写入数据时，才监听可写事件。也就是说我们调用bufferevent\_write写入数据时，Libevent才会把监听可写事件的那个event注册到event\_base中。当Libevent把数据都写入到fd的缓冲区后，Libevent又会把这个event从event\_base中删除。比较烦琐。

        bufferevent\_writecb函数不仅仅要处理上面说到的那个问题，还要处理另外一个坑爹的问题。那就是：判断socket fd是不是已经连接上服务器了。这是因为这个socket fd是非阻塞的，所以它调用connect时，可能还没连接上就返回了。对于非阻塞socket fd，一般是通过判断这个socket是否可写，从而得知这个socket是否已经连接上服务器。如果可写，那么它就已经成功连接上服务器了。这个问题，这里先提一下，[后面](http://blog.csdn.net/luotuo44/article/details/39344743#t8)会详细讲。

        同前面的监听可读一样，Libevent是在bufferevent\_socket\_new函数设置可写的回调函数，为bufferevent\_writecb。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent\_sock.c文件
2. **static** **void**
3. bufferevent\_writecb(evutil\_socket\_t fd, **short** event, **void** \*arg)
4. {
5. **struct** bufferevent \*bufev = arg;
6. **struct** bufferevent\_private \*bufev\_p =
7. EVUTIL\_UPCAST(bufev, **struct** bufferevent\_private, bev);
8. **int** res = 0;
9. **short** what = BEV\_EVENT\_WRITING;
10. **int** connected = 0;
11. ev\_ssize\_t atmost = -1;
13. \_bufferevent\_incref\_and\_lock(bufev);
15. **if** (event == EV\_TIMEOUT) {
16. /\* Note that we only check for event==EV\_TIMEOUT. If
17. \* event==EV\_TIMEOUT|EV\_WRITE, we can safely ignore the
18. \* timeout, since a read has occurred \*/
19. what |= BEV\_EVENT\_TIMEOUT;
20. **goto** error;
21. }
23. ...//判断这个socket是否已经连接上服务器了

26. //用户可能设置了限速，如果没有限速，那么atmost将返回16384(16K)
27. atmost = \_bufferevent\_get\_write\_max(bufev\_p);
29. //一些原因导致写被挂起来了
30. **if** (bufev\_p->write\_suspended)
31. **goto** done;
33. //如果evbuffer有数据可以写到sockfd中
34. **if** (evbuffer\_get\_length(bufev->output)) {
35. //解冻链表头
36. evbuffer\_unfreeze(bufev->output, 1);
37. //将output这个evbuffer的数据写到socket fd 的缓冲区中
38. //会把已经写到socket fd缓冲区的数据，从evbuffer中删除
39. res = evbuffer\_write\_atmost(bufev->output, fd, atmost);
40. evbuffer\_freeze(bufev->output, 1);
42. **if** (res == -1) {
43. **int** err = evutil\_socket\_geterror(fd);
44. **if** (EVUTIL\_ERR\_RW\_RETRIABLE(err))//可以恢复的错误。一般是EINTR或者EAGAIN
45. **goto** reschedule;
46. what |= BEV\_EVENT\_ERROR;
47. } **else** **if** (res == 0) {//该socket已经断开连接了
48. what |= BEV\_EVENT\_EOF;
49. }
50. **if** (res <= 0)
51. **goto** error;
52. }
54. //如果把写缓冲区的数据都写完成了。为了防止event\_base不断地触发可写
55. //事件，此时要把这个监听可写的event删除。
56. //前面的atmost限制了一次最大的可写数据。如果还没写所有的数据
57. //那么就不能delete这个event，而是要继续监听可写事情，知道把所有的
58. //数据都写到socket fd中。
59. **if** (evbuffer\_get\_length(bufev->output) == 0) {
60. event\_del(&bufev->ev\_write);
61. }

64. //如果evbuffer里面的数据量已经写得七七八八了，小于设置的低水位值，那么
65. //就会调用用户设置的写事件回调函数
66. **if** ((res || !connected) &&
67. evbuffer\_get\_length(bufev->output) <= bufev->wm\_write.low) {
68. \_bufferevent\_run\_writecb(bufev);
69. }
71. **goto** done;
73. reschedule:
74. **if** (evbuffer\_get\_length(bufev->output) == 0) {
75. event\_del(&bufev->ev\_write);
76. }
77. **goto** done;
79. error:
80. bufferevent\_disable(bufev, EV\_WRITE);//有错误。把这个写event删除
81. \_bufferevent\_run\_eventcb(bufev, what);
83. done:
84. \_bufferevent\_decref\_and\_unlock(bufev);
85. }

        上面代码的逻辑比较清晰，调用evbuffer\_write\_atmost函数把数据从evbuffer中写到evbuffer缓冲区中，此时要注意函数的返回值，因为可能写的时候发生错误。如果发生了错误，就要调用用户设置的event回调函数(网上也有人称其为错误处理函数)。

        之后，还要判断evbuffer的数据是否已经全部写到socket 的缓冲区了。如果已经全部写了，那么就要把监听写事件的event从event\_base的插入队列中删除。如果还没写完，那么就不能删除，因为还要继续监听可写事件，下次接着写。

        现在来看一下，把监听写事件的event从event\_base的插入队列中删除后，如果下次用户有数据要写的时候，怎么把这个event添加到event\_base的插入队列。

        用户一般是通过bufferevent\_write函数把数据写入到evbuffer(写入evbuffer后，接着就会被写入socket，所以调用bufferevent\_write就相当于把数据写入到socket。)。而这个bufferevent\_write函数是直接调用evbuffer\_add函数的。函数evbuffer\_add没有调用什么可疑的函数，能够把监听可写的event添加到event\_base中。唯一的可能就是那个回调函数。对就是evbuffer的回调函数。关于evbuffer的回调函数，可以参考[这里](http://blog.csdn.net/luotuo44/article/details/39325447#t6)。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent.c文件
2. **int**
3. bufferevent\_write(**struct** bufferevent \*bufev, **const** **void** \*data, **size\_t** size)
4. {
5. **if** (evbuffer\_add(bufev->output, data, size) == -1)
6. **return** (-1);
8. **return** 0;
9. }

12. //buffer.c文件
13. **int**
14. evbuffer\_add(**struct** evbuffer \*buf, **const** **void** \*data\_in, **size\_t** datlen)
15. {
16. ...
18. out:
19. evbuffer\_invoke\_callbacks(buf);//调用回调函数
20. result = 0;
21. done:
22. **return** result;
23. }

        还记得本文前面的bufferevent\_socket\_new函数吗？该函数里面会有

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. evbuffer\_add\_cb(bufev->output,bufferevent\_socket\_outbuf\_cb, bufev);

        当bufferevent的写缓冲区output的数据发生变化时，函数bufferevent\_socket\_outbuf\_cb就会被调用。现在马上飞到这个函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent\_sock.c文件
2. **static** **void**
3. bufferevent\_socket\_outbuf\_cb(**struct** evbuffer \*buf,
4. **const** **struct** evbuffer\_cb\_info \*cbinfo,
5. **void** \*arg)
6. {
7. **struct** bufferevent \*bufev = arg;
8. **struct** bufferevent\_private \*bufev\_p =
9. EVUTIL\_UPCAST(bufev, **struct** bufferevent\_private, bev);
11. **if** (cbinfo->n\_added && //evbuffer添加了数据
12. (bufev->enabled & EV\_WRITE) && //默认情况下是enable EV\_WRITE的
13. !event\_pending(&bufev->ev\_write, EV\_WRITE, NULL) &&//这个event已经被踢出event\_base了
14. !bufev\_p->write\_suspended) {//这个bufferevent的写并没有被挂起
16. //把这个event添加到event\_base中
17. **if** (be\_socket\_add(&bufev->ev\_write, &bufev->timeout\_write) == -1) {
18. /\* Should we log this? \*/
19. }
20. }
21. }

        这个函数首先进行一些判断，满足条件后就会把这个监听写事件的event添加到event\_base中。其中event\_pending函数就是判断这个bufev->ev\_write是否已经被event\_base删除了。关于event\_pending，可以参考[这里](http://blog.csdn.net/luotuo44/article/details/38739549#t1)。

        对于bufferevent\_write，初次使用该函数的读者可能会有疑问：调用该函数后，参数data指向的内存空间能不能马上释放，还是要等到Libevent把data指向的数据都写到socket 缓存区才能删除？其实，从[前一篇博文](http://blog.csdn.net/luotuo44/article/details/39290721#t2)可以看到，evbuffer\_add是直接复制一份用户要发送的数据到evbuffer缓存区的。所以，调用完bufferevent\_write，就可以马上释放参数data指向的内存空间。

        网上的关于Libevent的一些使用例子，包括我写的《 [Libevent使用例子，从简单到复杂](http://blog.csdn.net/luotuo44/article/details/39670221)》，都是在主线程中调用bufferevent\_write函数写入数据的。从上面的分析可以得知，是可以马上把监听可写事件的event添加到event\_base中。如果是在次线程调用该函数写入数据呢？此时，主线程可能还睡眠在poll、epoll这类的多路IO复用函数上。这种情况下能不能及时唤醒主线程呢？其实是可以的，只要你的Libevent在一开始使用了线程功能。具体的分析过程可以参考《[evthread\_notify\_base通知主线程](http://blog.csdn.net/luotuo44/article/details/38556059)》。上面代码中的be\_socket\_add会调用event\_add，而在次线程调用event\_add就会调用evthread\_notify\_base通知主线程。

# ****bufferevent\_socket\_connect：****

        用户可以在调用bufferevent\_socket\_new函数时，传一个-1作为socket的文件描述符，然后调用bufferevent\_socket\_connect函数连接服务器，无需自己写代码调用connect函数连接服务器。

        bufferevent\_socket\_connect函数会调用socket函数申请一个套接字fd，然后把这个fd设置成非阻塞的(这就导致了一些坑爹的事情)。接着就connect服务器，因为该socket fd是非阻塞的，所以不会等待，而是马上返回，连接这工作交给内核来完成。所以，返回后这个socket还没有真正连接上服务器。那么什么时候连接上呢？内核又是怎么通知通知用户呢？

        一般来说，当可以往socket fd写东西了，那就说明已经连接上了。也就是说这个socket fd变成可写状态，就连接上了。

        所以，对于“非阻塞connect”比较流行的做法是：用select或者poll这类多路IO复用函数监听该socket的可写事件。当这个socket触发了可写事件，然后再对这个socket调用getsockopt函数，做进一步的判断。

        Libevent也是这样实现的，下面来看一下bufferevent\_socket\_connect函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent\_sock.c文件
2. **int**
3. bufferevent\_socket\_connect(**struct** bufferevent \*bev,
4. **struct** sockaddr \*sa, **int** socklen)
5. {
6. **struct** bufferevent\_private \*bufev\_p =
7. EVUTIL\_UPCAST(bev, **struct** bufferevent\_private, bev);
9. evutil\_socket\_t fd;
10. **int** r = 0;
11. **int** result=-1;
12. **int** ownfd = 0;
14. \_bufferevent\_incref\_and\_lock(bev);
16. **if** (!bufev\_p)
17. **goto** done;

20. fd = bufferevent\_getfd(bev);
21. **if** (fd < 0) {//该bufferevent还没有设置fd
22. **if** (!sa)
23. **goto** done;
24. fd = socket(sa->sa\_family, SOCK\_STREAM, 0);
25. **if** (fd < 0)
26. **goto** done;
27. **if** (evutil\_make\_socket\_nonblocking(fd)<0)//设置为非阻塞
28. **goto** done;
29. ownfd = 1;
30. }
31. **if** (sa) {
32. r = evutil\_socket\_connect(&fd, sa, socklen);//非阻塞connect
33. **if** (r < 0)
34. **goto** freesock;
35. }
36. ...
38. //为bufferevent里面的两个event设置监听的fd
39. //后面会调用bufferevent\_enable
40. bufferevent\_setfd(bev, fd);
42. **if** (r == 0) {//暂时还没连接上，因为fd是非阻塞的
43. //此时需要监听可写事件，当可写了，并且没有错误的话，就成功连接上了
44. **if** (! be\_socket\_enable(bev, EV\_WRITE)) {
45. bufev\_p->connecting = 1;//标志这个sockfd正在连接
46. result = 0;
47. **goto** done;
48. }
49. } **else** **if** (r == 1) {//已经连接上了
50. /\* The connect succeeded already. How very BSD of it. \*/
51. result = 0;
52. bufev\_p->connecting = 1;
53. event\_active(&bev->ev\_write, EV\_WRITE, 1);//手动激活这个event
54. } **else** {// connection refused
55. /\* The connect failed already.  How very BSD of it. \*/
56. bufev\_p->connection\_refused = 1;
57. bufev\_p->connecting = 1;
58. result = 0;
59. event\_active(&bev->ev\_write, EV\_WRITE, 1);//手动激活这个event
60. }
62. **goto** done;
64. freesock:
65. \_bufferevent\_run\_eventcb(bev, BEV\_EVENT\_ERROR);//出现错误
66. **if** (ownfd)
67. evutil\_closesocket(fd);
68. done:
69. \_bufferevent\_decref\_and\_unlock(bev);
70. **return** result;
71. }

        这个函数比较多错误处理的代码，大致看一下就行了。有几个地方要注意，即使connect的时候被拒绝，或者已经连接上了，都会手动激活这个event。一个event即使没有加入event\_base，也是可以手动激活的。具体原理参考[这里](http://blog.csdn.net/luotuo44/article/details/38739549#t2)。

        无论是手动激活event，或者监听到这个event可写了，都是会调用bufferevent\_writecb函数。现在再次看一下该函数，只看connect部分。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //bufferevent\_sock.c文件
2. **static** **void**
3. bufferevent\_writecb(evutil\_socket\_t fd, **short** event, **void** \*arg)
4. {
5. **struct** bufferevent\_private \*bufev\_p =
6. EVUTIL\_UPCAST(bufev, **struct** bufferevent\_private, bev);
7. **int** connected = 0;
9. \_bufferevent\_incref\_and\_lock(bufev);
11. ...
12. //正在连接。因为这个sockfd可能是非阻塞的，所以可能之前的connect还没
13. //连接上。而判断该sockfd是否成功连接上了的一个方法是判断这个sockfd是否可写
14. **if** (bufev\_p->connecting) {
15. //c等于1，说明已经连接成功
16. //c等于0，说明还没连接上
17. //c等于-1，说明发生错误
18. **int** c = evutil\_socket\_finished\_connecting(fd);

21. **if** (bufev\_p->connection\_refused) {//在bufferevent\_socket\_connect中被设置
22. bufev\_p->connection\_refused = 0;
23. c = -1;
24. }
26. **if** (c == 0)//还没连接上，继续监听可写吧
27. **goto** done;

30. //错误，或者已经连接上了
31. bufev\_p->connecting = 0;//修改标志值
33. **if** (c < 0) {//错误
34. event\_del(&bufev->ev\_write);
35. event\_del(&bufev->ev\_read);
36. \_bufferevent\_run\_eventcb(bufev, BEV\_EVENT\_ERROR);
37. **goto** done;
39. } **else** {//连接上了。
40. connected = 1;
41. ...//win32
43. //居然会调用用户设置的错误处理函数。太神奇了
44. \_bufferevent\_run\_eventcb(bufev,
45. BEV\_EVENT\_CONNECTED);
46. **if** (!(bufev->enabled & EV\_WRITE) || //默认都是enable EV\_WRITE的
47. bufev\_p->write\_suspended) {
48. event\_del(&bufev->ev\_write);//不再需要监听可写。因为已经连接上了
49. **goto** done;
50. }
51. }
52. }
54. ...

57. done:
58. \_bufferevent\_decref\_and\_unlock(bufev);
59. }

        可以看到无论是connect被拒绝、发生错误或者连接上了，都在这里做统一的处理。

        如果已经连接上了，那么会调用用户设置event回调函数(网上也称之为错误处理函数)，通知用户已经连接上了。并且，还会把监听可写事件的event从event\_base中删除，其理由在前面已经说过了。

        函数evutil\_socket\_finished\_connecting会检查这个socket，从而得知这个socket是处于什么状态。在bufferevent\_socket\_connect函数中，出现的一些错误，比如被拒绝，也是能通过这个函数检查出来的。所以可以在这里做统一的处理。该函数的内部是使用。贴一下这个函数的代码吧。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39344743) [copy](https://blog.csdn.net/luotuo44/article/details/39344743)

1. //evutil.c文件
2. //Return 1 for connected, 0 for not yet, -1 for error.
3. **int**
4. evutil\_socket\_finished\_connecting(evutil\_socket\_t fd)
5. {
6. **int** e;
7. ev\_socklen\_t elen = **sizeof**(e);
9. //用来检测这个fd是否已经连接上了，这个fd是非阻塞的
10. //如果e的值被设为0，那么就说明连接上了。
11. //否则e被设置为对应的错误值。
12. **if** (getsockopt(fd, SOL\_SOCKET, SO\_ERROR, (**void**\*)&e, &elen) < 0)
13. **return** -1;
15. **if** (e) {
16. **if** (EVUTIL\_ERR\_CONNECT\_RETRIABLE(e))//还没连接上
17. **return** 0;
18. EVUTIL\_SET\_SOCKET\_ERROR(e);
19. **return** -1;
20. }
22. **return** 1;
23. }

        好长啊！终于写完了。

    谢绝推酷、第七城市的转载！！！

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/luotuo44/article/details/39344743