<https://blog.csdn.net/luotuo44/article/details/39325447>

**转载请注明出处：**[**http://blog.csdn.net/luotuo44/article/details/39325447**](http://blog.csdn.net/luotuo44/article/details/39325447)

# ****锁操作：****

        在[前一篇博文](http://blog.csdn.net/luotuo44/article/details/39290721)可以看到很多函数在操作前都需要对这个evbuffer进行加锁。同event\_base不同，如果evbuffer支持锁的话，要显式地调用函数evbuffer\_enable\_locking。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. //buffer.c文件
2. **int**//参数可以是一个锁变量也可以是NULL
3. evbuffer\_enable\_locking(**struct** evbuffer \*buf, **void** \*lock)
4. {
5. #ifdef \_EVENT\_DISABLE\_THREAD\_SUPPORT
6. **return** -1;
7. #else
8. **if** (buf->lock)
9. **return** -1;
11. **if** (!lock) {
12. //自己分配锁变量
13. EVTHREAD\_ALLOC\_LOCK(lock, EVTHREAD\_LOCKTYPE\_RECURSIVE);
14. **if** (!lock)
15. **return** -1;
16. buf->lock = lock;
17. //该evbuffer拥有锁，到时需要释放锁内存
18. buf->own\_lock = 1;
19. } **else** {
20. buf->lock = lock;//使用参数提供的锁
21. buf->own\_lock = 0;//自己没有拥有锁。不需要释放锁内存
22. }
24. **return** 0;
25. #endif
26. }

        可以看到，第二个参数可以为NULL。此时函数内部会申请一个锁。明显如果要让evbuffer能使用锁，就必须在一开始就调用evthread\_use\_windows\_threads()或者evthread\_use\_pthreads(),关于这个可以参考[一篇博文](http://blog.csdn.net/luotuo44/article/details/38350633)。

        因为用户操控这个evbuffer，所以Libevent提供了加锁和解锁接口给用户使用。

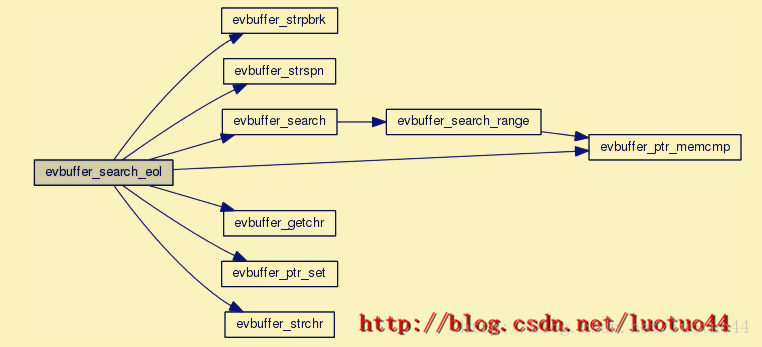
**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. //buffer.c文件
2. **void**
3. evbuffer\_lock(**struct** evbuffer \*buf)
4. {
5. EVBUFFER\_LOCK(buf);
6. }
8. **void**
9. evbuffer\_unlock(**struct** evbuffer \*buf)
10. {
11. EVBUFFER\_UNLOCK(buf);
12. }
14. //evbuffer-internal.h文件
15. #define EVBUFFER\_LOCK(buffer)                       \
16. **do** {                                \
17. EVLOCK\_LOCK((buffer)->lock, 0);              \
18. } **while** (0)
19. #define EVBUFFER\_UNLOCK(buffer)                     \
20. **do** {                                \
21. EVLOCK\_UNLOCK((buffer)->lock, 0);            \
22. } **while** (0)

        在Libevent内部，一般不会使用这个两个接口，而是直接使用EVBUFFER\_LOCK(buf)和EVBUFFER\_UNLOCK(buf)。

# ****查找操作：****

下图展示了evbuffer的一些查找操作以及调用关系。



## ****查找结构体：****

        对于一个数组或者一个文件，只需一个下标或者偏移量就可以定位查找了。但对于evbuffer来说，它的数据是由一个个的evbuffer\_chain用链表连在一起的。所以在evbuffer中定位，不仅仅要有一个偏移量，还要指明是哪个evbuffer\_chain，甚至是在evbuffer\_chain中的偏移量。因此Libevent定义了一个查找(定位)结构体：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. //buffer.h文件
2. **struct** evbuffer\_ptr {
3. ev\_ssize\_t pos;//总偏移量，相对于数据的开始位置
5. /\* Do not alter the values of fields. \*/
6. **struct** {
7. **void** \*chain;//指明是哪个evbuffer\_chain
8. **size\_t** pos\_in\_chain; //在evbuffer\_chain中的偏移量
9. } \_internal;
10. };

        有一点要注意，pos\_in\_chain是从misalign这个错开空间之后计算的，也就是说其实际偏移量为：chain->buffer+ chain->misalign + pos\_in\_chain。

        定位结构体有一个对应的操作函数evbuffer\_ptr\_set，该函数就像fseek函数那样，可以设置或者移动偏移量，并且可以绝对和相对地移动。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. //buffer.h文件
2. **enum** evbuffer\_ptr\_how {
3. EVBUFFER\_PTR\_SET, //偏移量是一个绝对位置
4. EVBUFFER\_PTR\_ADD //偏移量是一个相对位置
5. };

8. //buffer.c文件
9. //设置evbuffer\_ptr。evbuffer\_ptr\_set(buf, &pos, 0, EVBUFFER\_PTR\_SET)
10. //将这个pos指向链表的开头
11. //position指明移动的偏移量，how指明该偏移量是绝对偏移量还是相对当前位置的偏移量。
12. **int**//这个函数的作用就像C语言中的fseek，设置文件指针的偏移量
13. evbuffer\_ptr\_set(**struct** evbuffer \*buf, **struct** evbuffer\_ptr \*pos,
14. **size\_t** position, **enum** evbuffer\_ptr\_how how)
15. {
16. **size\_t** left = position;
17. **struct** evbuffer\_chain \*chain = NULL;
19. EVBUFFER\_LOCK(buf);
21. //这个switch的作用就是给pos设置新的总偏移量值。
22. **switch** (how) {
23. **case** EVBUFFER\_PTR\_SET://绝对位置
24. chain = buf->first;//从第一个evbuffer\_chain算起
25. pos->pos = position; //设置总偏移量
26. position = 0;
27. **break**;
28. **case** EVBUFFER\_PTR\_ADD://相对位置
29. chain = pos->\_internal.chain;//从当前evbuffer\_chain算起
30. pos->pos += position;//加上相对偏移量
31. position = pos->\_internal.pos\_in\_chain;
32. **break**;
33. }
35. //这个偏移量跨了evbuffer\_chain。可能不止跨一个chain。
36. **while** (chain && position + left >= chain->off) {
37. left -= chain->off - position;
38. chain = chain->next;
39. position = 0;
40. }

43. **if** (chain) {//设置evbuffer\_chain内的偏移量
44. pos->\_internal.chain = chain;
45. pos->\_internal.pos\_in\_chain = position + left;
46. } **else** {//跨过了所有的节点
47. pos->\_internal.chain = NULL;
48. pos->pos = -1;
49. }
51. EVBUFFER\_UNLOCK(buf);
53. **return** chain != NULL ? 0 : -1;
54. }

        可以看到，该函数只考虑了向后面的chain移动定位指针，不能向。当然如果参数position小于0，并且移动时并不会跨越当前的chain，还是可以的。不过最好不要这样做。如果确实想移回头，那么可以考虑下面的操作。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. pos.position -= 20;//移回头20个字节。
2. evbuffer\_ptr\_set(buf, &pos, 0, EVBUFFER\_PTR\_SET);

## ****查找一个字符：****

        有下面代码的前一个函数是可以用来查找一个字符的，第二个函数则是获取对于位置的字符。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. **static** **inline** ev\_ssize\_t
2. evbuffer\_strchr(**struct** evbuffer\_ptr \*it, **const** **char** chr);
4. **static** **inline** **char**//获取对应位置的字符
5. evbuffer\_getchr(**struct** evbuffer\_ptr \*it);
7. **static** **inline** **int**
8. evbuffer\_strspn(**struct** evbuffer\_ptr \*ptr, **const** **char** \*chrset);

        函数evbuffer\_strchr是从it指向的evbuffer\_chain开始查找，会往后面的链表查找。it是一个值-结果参数，如果查找到了，那么it将会指明被查找字符的位置，并返回相对于evbuffer的总偏移量(即it->pos)。如果没有找到，就返回-1。由于实现都是一般的字符比较，所以就不列出代码了。函数evbuffer\_getchr很容易理解也不列出代码了。

        第三个函数evbuffer\_strspn的参数chrset虽然是一个字符串，其实内部也是比较字符的。该函数所做的操作和C语言标准库里面的strspn函数是一样的。这里也不多说了。关于strspn函数的理解可以查看[这里](http://blog.csdn.net/luotuo44/article/details/24933673#t3)。

## ****查找一个字符串：****

        字符串的查找函数有evbuffer\_search\_range和evbuffer\_search，后者调用前者完成查找。

        在讲查找前，先看一个字符串比较函数evbuffer\_ptr\_memcmp。该函数是比较某一个字符串和从evbuffer中某个位置开始的字符是否相等。明显比较的时候需要考虑到跨evbuffer\_chain的问题。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. **static** **int** //匹配成功会返回0
2. evbuffer\_ptr\_memcmp(**const** **struct** evbuffer \*buf, **const** **struct** evbuffer\_ptr \*pos,
3. **const** **char** \*mem, **size\_t** len)
4. {
5. **struct** evbuffer\_chain \*chain;
6. **size\_t** position;
7. **int** r;
9. //链表数据不够
10. **if** (pos->pos + len > buf->total\_len)
11. **return** -1;
13. //需要考虑这个要匹配的字符串被分散在两个evbuffer\_chain中
14. chain = pos->\_internal.chain;
15. position = pos->\_internal.pos\_in\_chain;//从evbuffer\_chain中的这个位置开始
16. **while** (len && chain) {
17. **size\_t** n\_comparable;//该evbuffer\_chain中可以比较的字符数
18. **if** (len + position > chain->off)
19. n\_comparable = chain->off - position;
20. **else**
21. n\_comparable = len;
22. r = memcmp(chain->buffer + chain->misalign + position, mem,
23. n\_comparable);
24. **if** (r)//不匹配
25. **return** r;
27. //考虑跨evbuffer\_chain
28. mem += n\_comparable;
29. len -= n\_comparable;//还有这些是没有比较的
30. position = 0;
31. chain = chain->next;
32. }
34. **return** 0;//匹配成功
35. }

        该函数首先比较pos指向的当前evbuffer\_chain，如果字符mem还有一些字符没有参与比较，那么就需要用下一个evbuffer\_chain的数据。

        由于evbuffer的数据是由链表组成的，没办法直接用KMP查找算法或者直接调用strstr函数。有了evbuffer\_ptr\_memcmp函数，读者可能会想，一个字节一个字节地挪动evbuffer的数据，依次调用evbuffer\_ptr\_memcmp函数。但evbuffer\_search\_range函数也不是直接调用函数evbuffer\_ptr\_memcmp的。而是先用字符查找函数，找到要查找字符串中的第一个字符，然后才调用那个函数。下面是具体的代码。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. //buffer.c文件
2. **struct** evbuffer\_ptr
3. evbuffer\_search(**struct** evbuffer \*buffer, **const** **char** \*what, **size\_t** len, **const** **struct** evbuffer\_ptr \*start)
4. {
5. **return** evbuffer\_search\_range(buffer, what, len, start, NULL);
6. }
8. //参数start和end指明了查找的范围
9. **struct** evbuffer\_ptr
10. evbuffer\_search\_range(**struct** evbuffer \*buffer, **const** **char** \*what, **size\_t** len, **const** **struct** evbuffer\_ptr \*start, **const** **struct** evbuffer\_ptr \*end)
11. {
12. **struct** evbuffer\_ptr pos;
13. **struct** evbuffer\_chain \*chain, \*last\_chain = NULL;
14. **const** unsigned **char** \*p;
15. **char** first;
17. EVBUFFER\_LOCK(buffer);
19. //初始化pos
20. **if** (start) {
21. memcpy(&pos, start, **sizeof**(pos));
22. chain = pos.\_internal.chain;
23. } **else** {
24. pos.pos = 0;
25. chain = pos.\_internal.chain = buffer->first;
26. pos.\_internal.pos\_in\_chain = 0;
27. }
29. **if** (end)
30. last\_chain = end->\_internal.chain;
32. **if** (!len || len > EV\_SSIZE\_MAX)
33. **goto** done;
35. first = what[0];
37. //在本函数里面并不考虑到what的数据量比较链表的总数据量还多。
38. //但在evbuffer\_ptr\_memcmp函数中会考虑这个问题。此时该函数直接返回-1。
39. //本函数之所以没有考虑这样情况，可能是因为，在[start, end]之间有多少
40. //数据是不值得统计的，时间复杂度是O(n)。不是一个简单的buffer->total\_len
41. //就能获取到的
42. **while** (chain) {
43. **const** unsigned **char** \*start\_at =
44. chain->buffer + chain->misalign +
45. pos.\_internal.pos\_in\_chain;
46. //const void \* memchr ( const void \* ptr, int value, size\_t num );
47. //函数的作用是:在ptr指向的内存块中(长度为num个字节),需找字符value。
48. //如果找到就返回对应的位置，找不到返回NULL
49. p = memchr(start\_at, first,
50. chain->off - pos.\_internal.pos\_in\_chain);
51. **if** (p) {//找到了what[0]
52. pos.pos += p - start\_at;
53. pos.\_internal.pos\_in\_chain += p - start\_at;
54. //经过上面的两个 += 后，pos指向了这个chain中出现等于what[0]字符的位置
55. //但本函数是要匹配一个字符串，而非一个字符
57. //evbuffer\_ptr\_memcmp比较整个字符串。如果有需要的话，该函数会跨
58. //evbuffer\_chain进行比较，但不会修改pos。如果成功匹配，那么返回0
59. **if** (!evbuffer\_ptr\_memcmp(buffer, &pos, what, len)) {//匹配成功
60. //虽然匹配成功了，但可能是用到了end之后的链表数据。这也等于没有找到
61. **if** (end && pos.pos + (ev\_ssize\_t)len > end->pos)
62. **goto** not\_found;
63. **else**
64. **goto** done;
65. }
67. //跳过这个等于what[0]的字符
68. ++pos.pos;
69. ++pos.\_internal.pos\_in\_chain;
71. //这个evbuffer\_chain已经全部都对比过了。没有发现目标
72. **if** (pos.\_internal.pos\_in\_chain == chain->off) {
73. chain = pos.\_internal.chain = chain->next;
74. pos.\_internal.pos\_in\_chain = 0;//下一个chain从0开始
75. }
76. } **else** {//这个evbuffer\_chain都没有找到what[0]
77. **if** (chain == last\_chain)
78. **goto** not\_found;
80. //此时直接跳过这个evbuffer\_chain
81. pos.pos += chain->off - pos.\_internal.pos\_in\_chain;
82. chain = pos.\_internal.chain = chain->next;
83. pos.\_internal.pos\_in\_chain = 0;//下一个chain从0开始
84. }
85. }
87. not\_found:
88. pos.pos = -1;
89. pos.\_internal.chain = NULL;
90. done:
91. EVBUFFER\_UNLOCK(buffer);
92. **return** pos;
93. }

        evbuffer\_ptr\_memcmp函数和evbuffer\_search函数是有区别的。前者只会比较从pos指定位置开始的字符串，不会在另外的地方找一个字符串。而后者则会在后面另外找一个字符串进行比较。

## ****查找换行符：****

        换行符是一个比较重要的符号，例如http协议就基于行的。Libevent实现了一个简单的http服务器，因此在内部Libevent实现了一些读取一行数据函数以及与行相关的操作。

        有些系统行尾用\r\n有些则直接用\n，这些不统一给编程造成了一些麻烦。因此在Libevent中定义了一个枚举类型，专门来用表示eol(end of line)的。

* EVBUFFER\_EOL\_LF：行尾是’\n’字符
* EVBUFFER\_EOL\_CRLF\_STRICT：行尾是”\r\n”，一个回车符一个换行符
* EVBUFFER\_EOL\_CRLF：行尾是”\r\n”或者’\n’。这个是很有用的，因为可能标准的协议里面要求”\r\n”，但一些不遵循标准的用户可能使用’\n’
* EVBUFFER\_EOL\_ANY：行尾是任意次序或者任意数量的’\r’或者’\n’。这种格式不是很有用，只是用来向后兼容而已

         函数evbuffer\_readln是用来读取evbuffer中的一行数据(不会读取行尾符号)。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. **enum** evbuffer\_eol\_style {
2. EVBUFFER\_EOL\_ANY,
3. EVBUFFER\_EOL\_CRLF,
4. EVBUFFER\_EOL\_CRLF\_STRICT,
5. EVBUFFER\_EOL\_LF
6. };
8. //成功返回读取到的一行数据。否则返回NULL。该行数据会自动加上'\0'结尾
9. //如果n\_read\_out不为NULL，则被赋值为读取到的一行的字符数
10. **char** \*
11. evbuffer\_readln(**struct** evbuffer \*buffer, **size\_t** \*n\_read\_out,
12. **enum** evbuffer\_eol\_style eol\_style)
13. {
14. **struct** evbuffer\_ptr it;
15. **char** \*line;
16. **size\_t** n\_to\_copy=0, extra\_drain=0;
17. **char** \*result = NULL;
19. EVBUFFER\_LOCK(buffer);
21. **if** (buffer->freeze\_start) {
22. **goto** done;
23. }
25. //根据eol\_style行尾类型找到行尾。返回值的位置偏移量就指向那个行尾符号
26. //行尾符号前面的evbuffer数据就是一行的内容。extra\_drain指明这个行尾
27. //有多少个字符。后面需要把这个行尾符号删除，方便以后再次读取一行
28. it = evbuffer\_search\_eol(buffer, NULL, &extra\_drain, eol\_style);
29. **if** (it.pos < 0)
30. **goto** done;
31. n\_to\_copy = it.pos;//并不包括换行符
33. **if** ((line = mm\_malloc(n\_to\_copy+1)) == NULL) {
34. event\_warn("%s: out of memory", \_\_func\_\_);
35. **goto** done;
36. }
38. //复制并删除n\_to\_copy字节
39. evbuffer\_remove(buffer, line, n\_to\_copy);
40. line[n\_to\_copy] = '\0';
42. //extra\_drain指明是行尾符号占有的字节数。现在要删除之
43. evbuffer\_drain(buffer, extra\_drain);
44. result = line;
45. done:
46. EVBUFFER\_UNLOCK(buffer);
48. **if** (n\_read\_out)
49. \*n\_read\_out = result ? n\_to\_copy : 0;
51. **return** result;
52. }

        在函数内部会申请空间，并且从evbuffer中提取出一行数据。下面看看函数evbuffer\_search\_eol是怎么实现的。该函数执行查找行尾的工作，它在内部区分4种不同的行尾类型。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. **struct** evbuffer\_ptr
2. evbuffer\_search\_eol(**struct** evbuffer \*buffer,
3. **struct** evbuffer\_ptr \*start, **size\_t** \*eol\_len\_out,
4. **enum** evbuffer\_eol\_style eol\_style)
5. {
6. **struct** evbuffer\_ptr it, it2;
7. **size\_t** extra\_drain = 0;
8. **int** ok = 0;
10. EVBUFFER\_LOCK(buffer);//加锁
12. **if** (start) {
13. memcpy(&it, start, **sizeof**(it));
14. } **else** {//从头开始找
15. it.pos = 0;
16. it.\_internal.chain = buffer->first;
17. it.\_internal.pos\_in\_chain = 0;
18. }
20. **switch** (eol\_style) {
21. **case** EVBUFFER\_EOL\_ANY:
22. **if** (evbuffer\_find\_eol\_char(&it) < 0)
23. **goto** done;
24. memcpy(&it2, &it, **sizeof**(it));
25. //该case的就是寻找在最前面的任意数量的\r和\n。
26. //evbuffer\_strspn返回第一个不是\r或者\n的下标。
27. //此时extra\_drain前面的都是\r或者\n。直接删除即可
28. extra\_drain = evbuffer\_strspn(&it2, "\r\n");
29. **break**;
30. **case** EVBUFFER\_EOL\_CRLF\_STRICT: {//\r\n
31. it = evbuffer\_search(buffer, "\r\n", 2, &it);
32. **if** (it.pos < 0)//没有找到
33. **goto** done;
34. extra\_drain = 2;
35. **break**;
36. }
37. **case** EVBUFFER\_EOL\_CRLF://\n或者\r\n
38. **while** (1) {//这个循环的一个chain一个chain地检查的
39. //从it指向的evbuffer\_chain开始往链表后面查找\n和\r。
40. //找到两个中的一个即可。两个都有就优先查找\n。
41. //会修改it的内容，使得it指向查找到的位置。查找失败返回-1。
42. //如果一个evbuffer\_chain有\n或者\r就马上返回。
43. **if** (evbuffer\_find\_eol\_char(&it) < 0)
44. **goto** done;
45. **if** (evbuffer\_getchr(&it) == '\n') {//获取对应位置的字符
46. extra\_drain = 1;
47. **break**;
48. } **else** **if** (!evbuffer\_ptr\_memcmp(
49. buffer, &it, "\r\n", 2)) {//如果刚才找到的是\r就再测测是不是\r\n
50. extra\_drain = 2;
51. **break**;
52. } **else** {
53. //\r的后面不是\n，此时跳过\r，继续查找
54. **if** (evbuffer\_ptr\_set(buffer, &it, 1,
55. EVBUFFER\_PTR\_ADD)<0)
56. **goto** done;
57. }
58. }
59. **break**;
60. **case** EVBUFFER\_EOL\_LF://\n
61. **if** (evbuffer\_strchr(&it, '\n') < 0)//没有找到
62. **goto** done;
63. extra\_drain = 1;
64. **break**;
65. **default**:
66. **goto** done;
67. }
69. ok = 1;
70. done:
71. EVBUFFER\_UNLOCK(buffer);
73. **if** (!ok) {
74. it.pos = -1;
75. }
76. **if** (eol\_len\_out)
77. \*eol\_len\_out = extra\_drain;
79. **return** it;
80. }

# ****回调函数：****

        evbuffer有一个回调函数队列成员callbacks，向evbuffer删除或者添加数据时，就会调用这些回调函数。之所以是回调函数队列，是因为一个evbuffer是可以添加多个回调函数的，而且同一个回调函数可以被添加多次。

        使用回调函数时有一点要注意：因为当evbuffer被添加或者删除数据时，就会调用这些回调函数，所以在回调函数里面不要添加或者删除数据，不然将导致递归，死循环。

        evbuffer的回调函数对bufferevent来说是非常重要的，bufferevent的一些重要功能都是基于evbuffer的回调函数完成的。

## ****回调相关结构体：****

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. //buffer.h文件
2. **struct** evbuffer\_cb\_info {
3. //添加或者删除数据之前的evbuffer有多少字节的数据
4. **size\_t** orig\_size;
5. **size\_t** n\_added;//添加了多少数据
6. **size\_t** n\_deleted;//删除了多少数据
8. //因为每次删除或者添加数据都会调用回调函数，所以上面的三个成员只能记录从上一次
9. //回调函数被调用后，到本次回调函数被调用这段时间的情况。
10. };

13. //两个回调函数类型
14. **typedef** **void** (\*evbuffer\_cb\_func)(**struct** evbuffer \*buffer, **const** **struct** evbuffer\_cb\_info \*info, **void** \*arg);
16. //buffer\_compat.h文件。这个类型的回调函数已经不被推荐使用了
17. **typedef** **void** (\*evbuffer\_cb)(**struct** evbuffer \*buffer, **size\_t** old\_len, **size\_t** new\_len, **void** \*arg);
19. //evbuffer-internal.h文件
20. //内部结构体，结构体成员对用户透明
21. **struct** evbuffer\_cb\_entry {
22. /\*\* Structures to implement a doubly-linked queue of callbacks \*/
23. TAILQ\_ENTRY(evbuffer\_cb\_entry) next;
24. /\*\* The callback function to invoke when this callback is called.
25. If EVBUFFER\_CB\_OBSOLETE is set in flags, the cb\_obsolete field is
26. valid; otherwise, cb\_func is valid. \*/
27. **union** {//哪个回调类型。一般都是evbuffer\_cb\_func
28. evbuffer\_cb\_func cb\_func;
29. evbuffer\_cb cb\_obsolete;
30. } cb;
31. **void** \*cbarg;//回调函数的参数
32. ev\_uint32\_t flags;//该回调的标志
33. };
35. **struct** evbuffer {
36. ...
37. //可以添加多个回调函数。所以需要一个队列存储
38. TAILQ\_HEAD(evbuffer\_cb\_queue, evbuffer\_cb\_entry) callbacks;
39. };

## ****设置回调函数：****

        下面看一下怎么设置回调函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. **struct** evbuffer\_cb\_entry \*
2. evbuffer\_add\_cb(**struct** evbuffer \*buffer, evbuffer\_cb\_func cb, **void** \*cbarg)
3. {
4. **struct** evbuffer\_cb\_entry \*e;
5. **if** (! (e = mm\_calloc(1, **sizeof**(**struct** evbuffer\_cb\_entry))))
6. **return** NULL;
7. EVBUFFER\_LOCK(buffer);//加锁
8. e->cb.cb\_func = cb;
9. e->cbarg = cbarg;
10. e->flags = EVBUFFER\_CB\_ENABLED;//标志位, 允许回调
11. TAILQ\_INSERT\_HEAD(&buffer->callbacks, e, next);
12. EVBUFFER\_UNLOCK(buffer);//解锁
13. **return** e;
14. }

        参数cbarg就是回调函数被调用时的那个arg参数，这点对于熟悉Libevent的读者应该不难理解。上面这个函数是被一个evbuffer\_cb\_entry结构体指针插入到callbacks队列的前面，有关TAILQ\_HEAD队列和相关的插入操作可以参考博文《[TAILQ\_QUEUE队列](http://blog.csdn.net/luotuo44/article/details/38374009)》。

        上面函数返回一个evbuffer\_cb\_entry结构体指针。用户可以利用这个返回的结构体作一些处理，因为这个结构体已经和添加的回调函数绑定了。比如可以设置这个回调函数的标志值。或者利用这个结构体指针作为标识，从队列中找到这个回调函数并删除之。如下面代码所示:

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. **int** //设置标志
2. evbuffer\_cb\_set\_flags(**struct** evbuffer \*buffer,
3. **struct** evbuffer\_cb\_entry \*cb, ev\_uint32\_t flags)
4. {
5. /\* the user isn't allowed to mess with these. \*/
6. flags &= ~EVBUFFER\_CB\_INTERNAL\_FLAGS;
7. EVBUFFER\_LOCK(buffer);
8. cb->flags |= flags;
9. EVBUFFER\_UNLOCK(buffer);
10. **return** 0;
11. }
13. **int** //清除某个标志
14. evbuffer\_cb\_clear\_flags(**struct** evbuffer \*buffer,
15. **struct** evbuffer\_cb\_entry \*cb, ev\_uint32\_t flags)
16. {
17. /\* the user isn't allowed to mess with these. \*/
18. flags &= ~EVBUFFER\_CB\_INTERNAL\_FLAGS;
19. EVBUFFER\_LOCK(buffer);
20. cb->flags &= ~flags;
21. EVBUFFER\_UNLOCK(buffer);
22. **return** 0;
23. }
25. **int** //从队列中删除这个回调函数
26. evbuffer\_remove\_cb\_entry(**struct** evbuffer \*buffer,
27. **struct** evbuffer\_cb\_entry \*ent)
28. {
29. EVBUFFER\_LOCK(buffer);
30. TAILQ\_REMOVE(&buffer->callbacks, ent, next);
31. EVBUFFER\_UNLOCK(buffer);
32. mm\_free(ent);
33. **return** 0;
34. }

37. **int** //根据用户设置的回调函数和回调参数这两个量 从队列中删除
38. evbuffer\_remove\_cb(**struct** evbuffer \*buffer, evbuffer\_cb\_func cb, **void** \*cbarg)
39. {
40. **struct** evbuffer\_cb\_entry \*cbent;
41. **int** result = -1;
42. EVBUFFER\_LOCK(buffer);
43. TAILQ\_FOREACH(cbent, &buffer->callbacks, next) {
44. **if** (cb == cbent->cb.cb\_func && cbarg == cbent->cbarg) {
45. result = evbuffer\_remove\_cb\_entry(buffer, cbent);
46. **goto** done;
47. }
48. }
49. done:
50. EVBUFFER\_UNLOCK(buffer);
51. **return** result;
52. }
54. //Libevent还是提供了一个删除所有回调函数的接口
55. **static** **void**
56. evbuffer\_remove\_all\_callbacks(**struct** evbuffer \*buffer)
57. {
58. **struct** evbuffer\_cb\_entry \*cbent;
60. **while** ((cbent = TAILQ\_FIRST(&buffer->callbacks))) {
61. TAILQ\_REMOVE(&buffer->callbacks, cbent, next);
62. mm\_free(cbent);
63. }
64. }

        前面的代码展示了两个回调函数的类型，分别是evbuffer\_cb\_func和evbuffer\_cb。后者在回调的时候可以得知删除或者添加数据之前的数据量和之后的数据量。但这两个数值都可以通过evbuffer\_cb\_info获取。所以evbuffer\_cb回调类型的优势没有了。此外，还有一个问题。一般是通过evbuffer\_setcb函数设置evbuffer\_cb类型的回调函数。而这个函数会先删除之前添加的所有回调函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. **void**
2. evbuffer\_setcb(**struct** evbuffer \*buffer, evbuffer\_cb cb, **void** \*cbarg)
3. {
4. EVBUFFER\_LOCK(buffer);
6. **if** (!TAILQ\_EMPTY(&buffer->callbacks))
7. evbuffer\_remove\_all\_callbacks(buffer);//清空之前的回调函数
9. **if** (cb) {
10. **struct** evbuffer\_cb\_entry \*ent =
11. evbuffer\_add\_cb(buffer, NULL, cbarg);
12. ent->cb.cb\_obsolete = cb;
13. ent->flags |= EVBUFFER\_CB\_OBSOLETE;
14. }
15. EVBUFFER\_UNLOCK(buffer);
16. }

        可以看到，evbuffer\_setcb为标志位flags加上了EVBUFFER\_CB\_OBSOLETE属性。从名字可以看到这是一个已经过时的属性。其实evbuffer\_setcb已经被推荐使用了。

## ****Libevent如何调用回调函数：****

        下面看一下是怎么调用回调函数的。其实现也简单，直接遍历回调队列，然后依次调用回调函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. **static** **void** //在evbuffer\_add中调用的该函数,running\_deferred为0
2. evbuffer\_run\_callbacks(**struct** evbuffer \*buffer, **int** running\_deferred)
3. {
4. **struct** evbuffer\_cb\_entry \*cbent, \*next;
5. **struct** evbuffer\_cb\_info info;
6. **size\_t** new\_size;
7. ev\_uint32\_t mask, masked\_val;
8. **int** clear = 1;
10. **if** (running\_deferred) {
11. mask = EVBUFFER\_CB\_NODEFER|EVBUFFER\_CB\_ENABLED;
12. masked\_val = EVBUFFER\_CB\_ENABLED;
13. } **else** **if** (buffer->deferred\_cbs) {
14. mask = EVBUFFER\_CB\_NODEFER|EVBUFFER\_CB\_ENABLED;
15. masked\_val = EVBUFFER\_CB\_NODEFER|EVBUFFER\_CB\_ENABLED;
16. /\* Don't zero-out n\_add/n\_del, since the deferred callbacks
17. will want to see them. \*/
18. clear = 0;
19. } **else** { //一般都是这种情况
20. mask = EVBUFFER\_CB\_ENABLED;
21. masked\_val = EVBUFFER\_CB\_ENABLED;
22. }
24. **if** (TAILQ\_EMPTY(&buffer->callbacks)) {//用户没有设置回调函数
25. //清零
26. buffer->n\_add\_for\_cb = buffer->n\_del\_for\_cb = 0;
27. **return**;
28. }
30. //没有添加或者删除数据
31. **if** (buffer->n\_add\_for\_cb == 0 && buffer->n\_del\_for\_cb == 0)
32. **return**;
34. new\_size = buffer->total\_len;
35. info.orig\_size = new\_size + buffer->n\_del\_for\_cb - buffer->n\_add\_for\_cb;
36. info.n\_added = buffer->n\_add\_for\_cb;
37. info.n\_deleted = buffer->n\_del\_for\_cb;
38. **if** (clear) {//清零，为下次计算做准备
39. buffer->n\_add\_for\_cb = 0;
40. buffer->n\_del\_for\_cb = 0;
41. }
43. //遍历回调函数队列，调用回调函数
44. **for** (cbent = TAILQ\_FIRST(&buffer->callbacks);
45. cbent != TAILQ\_END(&buffer->callbacks);
46. cbent = next) {
48. next = TAILQ\_NEXT(cbent, next);
50. //该回调函数没有enable
51. **if** ((cbent->flags & mask) != masked\_val)
52. **continue**;
54. **if** ((cbent->flags & EVBUFFER\_CB\_OBSOLETE))//已经不被推荐使用了
55. cbent->cb.cb\_obsolete(buffer,
56. info.orig\_size, new\_size, cbent->cbarg);
57. **else**
58. cbent->cb.cb\_func(buffer, &info, cbent->cbarg);//调用用户设置的回调函数
59. }
60. }

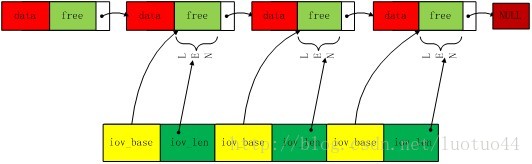
        无论是删除数据还是添加数据的函数，例如evbuffer\_add和evbuffer\_drain函数，都是会调用evbuffer\_invoke\_callbacks函数的。而这个函数会调用evbuffer\_run\_callbacks函数。

# evbuffer与网络IO：

## ****从Socket中读取数据：****

        Libevent通过evbuffer\_read函数从一个socket中读取数据到evbuffer中。在读取socket数据之前，Libevent会调用ioctl函数来获取这个socket的读缓冲区中有多少字节，进而确定本次要读多少字节到evbuffer中。Libevent会根据要读取的字节数，在真正read之前会先把evbuffer扩容，免得在read的时候缓冲区不够。

        在扩容的时候，如果所在的系统是支持类似readv这样的可以把数据读取到像一个iovec结构体函数，那么Libevent就会选择在n个evbuffer\_chain中找到足够的空闲空间(往往通过申请堆空间)，因为这样可以使用类似Linux的iovec结构体。把链表的各个evbuffer\_chain的空闲空间的地址赋值给iovec数组(如下图所示)，然后调用readv函数直接读取，readv会把数据读取到相应的chain中。



        上图中，有一点是和Libevent的实际情况不符合的。在evbuffer\_read中，最后的那个几个evbuffer\_chain一般是没有数据的，只有空闲的区域。上图为了好看，就加上了data这区域。

        将链表的各个evbuffer\_chain的空闲空间的地址赋值给iovec数组，这个操作是由函数\_evbuffer\_read\_setup\_vecs完成的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. //让vecs数组的指针指向evbuffer中的可用chain.标明哪个chain可用并且从chain的哪里开始，以及可用的字节数
2. //howmuch是要扩容的大小。vecs、n\_vecs\_avail分别是iovec数组和数组的大小
3. //chainp是值-结果参数，它最后指向第一个有可用空间的chain
4. **int**
5. \_evbuffer\_read\_setup\_vecs(**struct** evbuffer \*buf, ev\_ssize\_t howmuch,
6. **struct** evbuffer\_iovec \*vecs, **int** n\_vecs\_avail,
7. **struct** evbuffer\_chain \*\*\*chainp, **int** exact)
8. {
9. **struct** evbuffer\_chain \*chain;
10. **struct** evbuffer\_chain \*\*firstchainp;
11. **size\_t** so\_far;
12. **int** i;
14. **if** (howmuch < 0)
15. **return** -1;
17. so\_far = 0;
19. //因为找的是evbuffer链表中的空闲空间，所以从最后一个有数据的chain中开始找
20. firstchainp = buf->last\_with\_datap;
21. **if** (CHAIN\_SPACE\_LEN(\*firstchainp) == 0) {//这个chain已经没有空间了
22. firstchainp = &(\*firstchainp)->next;//那么只能下一个chain了
23. }
25. //因为Libevent在调用本函数之前，一般会调用\_evbuffer\_expand\_fast来扩大
26. //evbuffer的可用空间。所以下面的循环中并没有判断chain是否为NULL，就直接
27. //chain->next
28. chain = \*firstchainp;
29. **for** (i = 0; i < n\_vecs\_avail && so\_far < (**size\_t**)howmuch; ++i) {
30. **size\_t** avail = (**size\_t**) CHAIN\_SPACE\_LEN(chain);
31. //如果exact为真，那么即使这个chain有更多的可用空间，也不会使用。只会
32. //要自己正需要的空间
33. **if** (avail > (howmuch - so\_far) && exact)
34. avail = howmuch - so\_far;
35. vecs[i].iov\_base = CHAIN\_SPACE\_PTR(chain);//这个chain的可用空间的开始位置
36. vecs[i].iov\_len = avail;//可用长度
37. so\_far += avail;
38. chain = chain->next;
39. }
41. \*chainp = firstchainp; //指向第一个有可用空间的chain
42. **return** i;//返回需要多少个chain才能有howmuch这么多的空闲空间
43. }

        在Windows系统，虽然没有readv函数，但它有WSARecv函数，可以把数据读取到一个类似iovec的结构体中，所以在Windows系统中，Libevent还是选择在n个evbuffer\_chain中找到足够的空闲空间。所以在Libevent中有下面的宏定义：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. //buffer.c文件
2. //sys/uio.h文件定义了readv函数
3. #if defined(\_EVENT\_HAVE\_SYS\_UIO\_H) || defined(WIN32)
4. #define USE\_IOVEC\_IMPL //该宏标志所在的系统支持类似readv的函数
5. #endif
7. //Windows系统定义了下面结构体
8. **typedef** **struct** \_\_WSABUF {
9. u\_long   len;
10. **char** FAR \*buf;
11. } WSABUF, \*LPWSABUF;

        如果所在的系统不支持类似readv这样的函数，那么Libevent就只能在一个evbuffer\_chain申请一个足够大的空间，然后直接调用read函数了。

        前面说到的扩容，分别是由函数\_evbuffer\_expand\_fast和函数evbuffer\_expand\_singlechain完成的。在《[evbuffer结构与基本操作](http://blog.csdn.net/luotuo44/article/details/39290721#t3)》一文中已经有对这两个函数的介绍，这里就不多说了。

        由于存在是否支持类似readv函数 这两种情况，所以evbuffer\_read在实现上也出现了两种实现。

        上面说了这么多，还是来看一下evbuffer\_read的具体实现吧。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. //buffer.c文件
2. //返回读取到的字节数。错误返回-1，断开了连接返回0
3. **int** //howmuch指出此时evbuffer可以使用的空间大小
4. evbuffer\_read(**struct** evbuffer \*buf, evutil\_socket\_t fd, **int** howmuch)
5. {
6. **struct** evbuffer\_chain \*\*chainp;
7. **int** n;
8. **int** result;
10. #ifdef USE\_IOVEC\_IMPL //所在的系统支持iovec或者是Windows操作系统
11. **int** nvecs, i, remaining;
12. #else
13. **struct** evbuffer\_chain \*chain;
14. unsigned **char** \*p;
15. #endif
17. EVBUFFER\_LOCK(buf);//加锁
19. //冻结缓冲区尾部，禁止在尾部追加数据
20. **if** (buf->freeze\_end) {
21. result = -1;
22. **goto** done;
23. }
25. //获取这个socket接收缓冲区里面有多少字节可读.通过ioctl实现
26. n = get\_n\_bytes\_readable\_on\_socket(fd);
27. **if** (n <= 0 || n > EVBUFFER\_MAX\_READ)//每次只读EVBUFFER\_MAX\_READ(4096)个字符
28. n = EVBUFFER\_MAX\_READ;
29. **if** (howmuch < 0 || howmuch > n)
30. howmuch = n;
32. #ifdef USE\_IOVEC\_IMPL //所在的系统支持iovec或者是Windows操作系统
33. //NUM\_READ\_IOVEC等于4
34. //扩大evbuffer,使得其有howmuch字节的空闲空间
35. //在NUM\_READ\_IOVEC个evbuffer\_chain中扩容足够的空闲空间
36. **if** (\_evbuffer\_expand\_fast(buf, howmuch, NUM\_READ\_IOVEC) == -1) {
37. result = -1;
38. **goto** done;
39. } **else** {
40. //在posix中IOV\_TYPE为iovec，在Windows中为WSABUF
41. IOV\_TYPE vecs[NUM\_READ\_IOVEC];
43. #ifdef \_EVBUFFER\_IOVEC\_IS\_NATIVE //所在的系统支持iovec结构体
44. nvecs = \_evbuffer\_read\_setup\_vecs(buf, howmuch, vecs,
45. NUM\_READ\_IOVEC, &chainp, 1);
46. #else //Windows系统。因为没有native的 iovec
48. //在Windows系统中，evbuffer\_iovec定义得和posix中的iovec一样.
49. //因为\_evbuffer\_read\_setup\_vecs函数只接受像iovec那样结构体的参数
50. **struct** evbuffer\_iovec ev\_vecs[NUM\_READ\_IOVEC];
51. nvecs = \_evbuffer\_read\_setup\_vecs(buf, howmuch, ev\_vecs, 2,
52. &chainp, 1);
54. //因为在Windows中，需要使用WSABUF结构体读取数据，所以要从evbuffer\_iovec
55. //中将一些值提出出来，放到vecs中
56. **for** (i=0; i < nvecs; ++i)
57. WSABUF\_FROM\_EVBUFFER\_IOV(&vecs[i], &ev\_vecs[i]);
58. #endif
60. //完成把数据从socket fd中读取出来
61. #ifdef WIN32
62. {
63. **DWORD** bytesRead;
64. **DWORD** flags=0;
65. //虽然Windows支持类似readv的函数，但Windows没有readv函数，只有下面的函数
66. **if** (WSARecv(fd, vecs, nvecs, &bytesRead, &flags, NULL, NULL)) {
67. **if** (WSAGetLastError() == WSAECONNABORTED)
68. n = 0;
69. **else**
70. n = -1;
71. } **else**
72. n = bytesRead;
73. }
74. #else
75. n = readv(fd, vecs, nvecs);//POSIX
76. #endif
77. }
79. //如果所在的系统不支持 iovec并且不是Windows系统。也就是说不支持类似
80. //readv这样的函数。那么只能把所有的数据都读到一个chain中
81. #else /\*!USE\_IOVEC\_IMPL\*/
83. //把一个chain扩大得可以有howmuch字节的空闲空间
84. **if** ((chain = evbuffer\_expand\_singlechain(buf, howmuch)) == NULL) {
85. result = -1;
86. **goto** done;
87. }
89. p = chain->buffer + chain->misalign + chain->off;
91. //读取数据
92. #ifndef WIN32
93. n = read(fd, p, howmuch);
94. #else
95. n = recv(fd, p, howmuch, 0);
96. #endif
98. #endif /\* USE\_IOVEC\_IMPL \*///终止前面的if宏

101. **if** (n == -1) {//错误
102. result = -1;
103. **goto** done;
104. }
105. **if** (n == 0) {//断开了连接
106. result = 0;
107. **goto** done;
108. }

111. #ifdef USE\_IOVEC\_IMPL//使用了iovec结构体读取数据。需要做一些额外的处理
113. //chainp是由\_evbuffer\_read\_setup\_vecs函数调用得到。它指向未从fd读取数据时
114. //第一个有空闲空间位置的chain
116. remaining = n;//n等于读取到的字节数
117. //使用iovec读取数据时，只是把数据往chain中填充，并没有修改evbuffer\_chain
118. //的成员，比如off偏移量成员。此时就需要把这个off修改到正确值
119. **for** (i=0; i < nvecs; ++i) {
120. //CHAIN\_SPACE\_LEN(\*chainp)返回的是填充数据前的空闲空间。
121. //除了最后那个chain外，其他的chain都会被填满的。所以对于非last
122. //chain，直接把off加上这个space即可。
123. ev\_ssize\_t space = (ev\_ssize\_t) CHAIN\_SPACE\_LEN(\*chainp);
124. **if** (space < remaining) {//前面的chain
125. (\*chainp)->off += space;
126. remaining -= (**int**)space;
127. } **else** {//最后那个chain
128. (\*chainp)->off += remaining;
129. buf->last\_with\_datap = chainp;//指向最后一个有数据的chain
130. **break**;
131. }
132. chainp = &(\*chainp)->next;
133. }
134. #else
135. chain->off += n;
136. //调整last\_with\_datap，使得\*last\_with\_datap指向最后一个有数据的chain
137. advance\_last\_with\_data(buf);
138. #endif
140. buf->total\_len += n;
141. buf->n\_add\_for\_cb += n;//添加了n字节
143. /\* Tell someone about changes in this buffer \*/
144. evbuffer\_invoke\_callbacks(buf);//evbuffer添加了数据，就需要调用回调函数
145. result = n;
146. done:
147. EVBUFFER\_UNLOCK(buf);
148. **return** result;
149. }

## ****往socket写入数据：****

        因为evbuffer是用链表的形式存放数据，所以要把这些链表上的数据写入socket，那么使用writev这个函数是十分有效的。同前面一样，使用iovec结构体数组，就需要设置数组元素的指针。这个工作由evbuffer\_write\_iovec函数完成。

        正如前面的从socket读出数据，可能所在的系统并不支持writev这样的函数。此时就只能使用一般的write函数了，但这个函数要求数据放在一个连续的空间。所以Libevent有一个函数evbuffer\_pullup，用来把链表内存拉直，即把一定数量的数据从链表中copy到一个连续的内存空间。这个连续的空间也是由某个evbuffer\_chain的buffer指针指向，并且这个evbuffer\_chain会被插入到链表中。这个时候就可以直接使用write或者send函数发送这特定数量的数据了。

        不同于读，写操作还有第三种可能。那就是[sendfile](http://man7.org/linux/man-pages/man2/sendfile.2.html)。如果所在的系统支持sendfile，并且用户是通过evbuffer\_add\_file添加数据的，那么此时Libevent就是所在系统的sendfile函数发送数据。

        Libevent内部一般通过evbuffer\_write函数把数据写入到socket fd中。下面是具体的实现。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39325447) [copy](https://blog.csdn.net/luotuo44/article/details/39325447)

1. //buffer.c文件
2. **int**
3. evbuffer\_write(**struct** evbuffer \*buffer, evutil\_socket\_t fd)
4. {
5. //把evbuffer的所有数据都写入到fd中
6. **return** evbuffer\_write\_atmost(buffer, fd, -1);
7. }

10. **int**//howmuch是要写的字节数。如果小于0，那么就把buffer里的所有数据都写入fd
11. evbuffer\_write\_atmost(**struct** evbuffer \*buffer, evutil\_socket\_t fd,
12. ev\_ssize\_t howmuch)
13. {
14. **int** n = -1;
16. EVBUFFER\_LOCK(buffer);
18. //冻结了链表头，无法往fd写数据。因为写之后，还要把数据从evbuffer中删除
19. **if** (buffer->freeze\_start) {
20. **goto** done;
21. }
23. **if** (howmuch < 0 || (**size\_t**)howmuch > buffer->total\_len)
24. howmuch = buffer->total\_len;
26. **if** (howmuch > 0) {
27. #ifdef USE\_SENDFILE //所在的系统支持sendfile
28. **struct** evbuffer\_chain \*chain = buffer->first;
29. //需通过evbuffer\_add\_file添加数据，才会使用sendfile
30. **if** (chain != NULL && (chain->flags & EVBUFFER\_SENDFILE)) //并且要求使用sendfile
31. n = evbuffer\_write\_sendfile(buffer, fd, howmuch);
32. **else** {
33. #endif
34. #ifdef USE\_IOVEC\_IMPL //所在的系统支持writev这类函数
35. //函数内部会设置数组元素的成员指针，以及长度成员
36. n = evbuffer\_write\_iovec(buffer, fd, howmuch);
37. #elif defined(WIN32)
38. /\* XXX(nickm) Don't disable this code until we know if
39. \* the WSARecv code above works. \*/
40. //把evbuffer前面的howmuch字节拉直。使得这howmuch字节都放在一个chain里面
41. //也就是放在一个连续的空间，不再是之前的多个链表节点。这样就能直接用
42. //send函数发送了。
43. **void** \*p = evbuffer\_pullup(buffer, howmuch);
44. n = send(fd, p, howmuch, 0);
45. #else
46. **void** \*p = evbuffer\_pullup(buffer, howmuch);
47. n = write(fd, p, howmuch);
48. #endif
49. #ifdef USE\_SENDFILE
50. }
51. #endif
52. }
54. **if** (n > 0)
55. evbuffer\_drain(buffer, n);//从链表中删除已经写入到socket的n个字节
57. done:
58. EVBUFFER\_UNLOCK(buffer);
59. **return** (n);
60. }

参考：

<http://www.wangafu.net/~nickm/libevent-book/Ref7_evbuffer.html>

    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/39325447