<https://blog.csdn.net/luotuo44/article/details/39290721>

**转载请注明出处：**<http://blog.csdn.net/luotuo44/article/details/39290721>

        对于非阻塞IO的网络库来说，buffer几乎是必须的。Libevent在1.0版本之前就提供了buffer功能。现在来看一下Libevent的buffer。

# ****buffer相关结构体：****

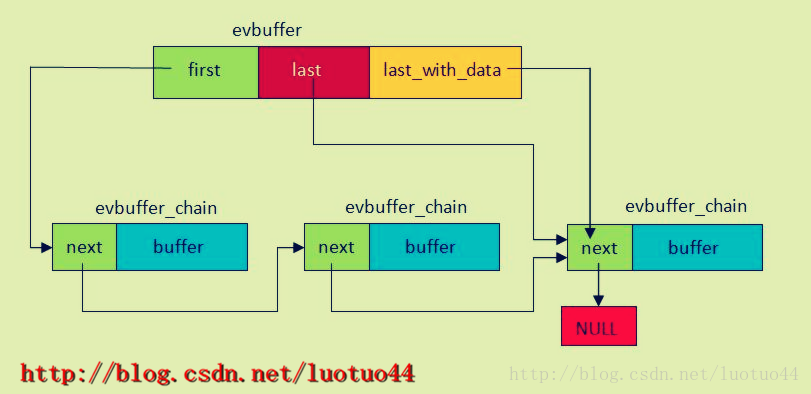
        Libevent为buffer定义了下面的结构体：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39290721) [copy](https://blog.csdn.net/luotuo44/article/details/39290721)

1. //evbuffer-internal.h文件
2. **struct** evbuffer\_chain;
3. **struct** evbuffer {
4. **struct** evbuffer\_chain \*first;
5. **struct** evbuffer\_chain \*last;
6. //这是一个二级指针。使用\*last\_with\_datap时，指向的是链表中最后一个有数据的evbuffer\_chain。
7. //所以last\_with\_datap存储的是倒数第二个evbuffer\_chain的next成员地址。
8. //一开始buffer->last\_with\_datap = &buffer->first;此时first为NULL。所以当链表没有节点时
9. //\*last\_with\_datap为NULL。当只有一个节点时\*last\_with\_datap就是first。
10. **struct** evbuffer\_chain \*\*last\_with\_datap;
12. **size\_t** total\_len;//链表中所有chain的总字节数
14. ...
15. };

18. **struct** evbuffer\_chain {
19. **struct** evbuffer\_chain \*next;
20. **size\_t** buffer\_len;//buffer的大小
22. //错开不使用的空间。该成员的值一般等于0
23. ev\_off\_t misalign;
25. //evbuffer\_chain已存数据的字节数
26. //所以要从buffer + misalign + off的位置开始写入数据
27. **size\_t** off;
29. ...
31. unsigned **char** \*buffer;
32. };

        这两个结构体配合工作得到下图所示的存储结构：



        因为last\_with\_datap成员比较特殊，上图只是展示了一种情况。后面还有一张图，展示另外一种情况。

        Libevent将缓冲数据都存放到buffer中。通过一个个的evbuffer\_chain连成的链表可以存放很多的缓冲数据。

        这是一个很常见的链表形式。但Libevent有一个很独特的地方，就是那个evbuffer\_chain结构体。

        首先，该结构体有misalign成员。该成员表示错开不用的buffer空间。也就是说buffer中真正的数据是从buffer + misalign开始。

        第二，evbuffer\_chain结构体buffer是一个指针，按道理来说，应该单独调用malloc分配一个堆内存并让buffer指向之。但实际上buffer指向的内存和evbuffer\_chain结构体本身的存储内存是一起分配的。下面代码展示了这一点：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39290721) [copy](https://blog.csdn.net/luotuo44/article/details/39290721)

1. //evbuffer-internal.h文件
2. #define EVBUFFER\_CHAIN\_SIZE sizeof(struct evbuffer\_chain)
4. #if \_EVENT\_SIZEOF\_VOID\_P < 8
5. #define MIN\_BUFFER\_SIZE 512
6. #else
7. #define MIN\_BUFFER\_SIZE 1024
8. #endif
10. //宏的作用就是返回，chain + sizeof(evbuffer\_chain) 的内存地址。
11. #define EVBUFFER\_CHAIN\_EXTRA(t, c) (t \*)((struct evbuffer\_chain \*)(c) + 1)

14. //buffer.c文件
15. **static** **struct** evbuffer\_chain \*
16. evbuffer\_chain\_new(**size\_t** size)//size是buffer所需的大小
17. {
18. **struct** evbuffer\_chain \*chain;
19. **size\_t** to\_alloc;
21. //所需的大小size 再 加上evbuffer\_chain结构体本身所需
22. //的内存大小。这样做的原因是，evbuffer\_chain本身是管理
23. //buffer的结构体。但buffer内存就分配在evbuffer\_chain结构体存储
24. //内存的后面。所以要申请多一些内存。
25. size += EVBUFFER\_CHAIN\_SIZE;//evbuffer\_chain结构体本身的大小

28. to\_alloc = MIN\_BUFFER\_SIZE; //内存块的最小值
29. **while** (to\_alloc < size)
30. to\_alloc <<= 1;
31. //从分配的内存大小可以知道，evbuffer\_chain结构体和buffer是一起分配的
32. //也就是说他们是存放在同一块内存中
33. **if** ((chain = mm\_malloc(to\_alloc)) == NULL)
34. **return** (NULL);
36. //只需初始化最前面的结构体部分即可
37. memset(chain, 0, EVBUFFER\_CHAIN\_SIZE);
39. //buffer\_len存储的是buffer的大小
40. chain->buffer\_len = to\_alloc - EVBUFFER\_CHAIN\_SIZE;
42. //宏的作用就是返回，chain + sizeof(evbuffer\_chain) 的内存地址。
43. //其效果就是buffer指向的内存刚好是在evbuffer\_chain的后面。
44. chain->buffer = EVBUFFER\_CHAIN\_EXTRA(u\_char, chain);
46. **return** (chain);
47. }

        前面的图中，buffer内存区域(蓝色区域)连在next的后面也是基于这一点的。在代码的while循环中也可以看到申请的空间大小是512的倍数，也就是说evbuffer\_chain申请的空间大小是512、1024、2048、4096……

        上面贴出了函数evbuffer\_chain\_new，该函数是用来创建一个evbuffer\_chain。现在贴出另外一个函数evbuffer\_new，它是用来创建一个evbuffer的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39290721) [copy](https://blog.csdn.net/luotuo44/article/details/39290721)

1. //buffer.c
2. **struct** evbuffer \*
3. evbuffer\_new(**void**)
4. {
5. **struct** evbuffer \*buffer;
7. buffer = mm\_calloc(1, **sizeof**(**struct** evbuffer));
8. **if** (buffer == NULL)
9. **return** (NULL);
11. buffer->refcnt = 1;
12. buffer->last\_with\_datap = &buffer->first;
14. **return** (buffer);
15. }

# ****Buffer的数据操作：****

## ****在链表尾添加数据：****

        Libevent提供给用户的添加数据接口是evbuffer\_add，现在就通过这个函数看一下是怎么将数据插入到buffer中的。该函数是在链表的尾部添加数据，如果想在链表的前面添加数据可以使用evbuffer\_prepend。在链表尾部插入数据，分下面几种情况：

1. 该链表为空，即这是第一次插入数据。这是最简单的，直接把新建的evbuffer\_chain插入到链表中，通过调用evbuffer\_chain\_insert。
2. 链表的最后一个节点(即evbuffer\_chain)还有一些空余的空间，放得下本次要插入的数据。此时直接把数据追加到最后一个节点即可。
3. 链表的最后一个节点并不能放得下本次要插入的数据，那么就需要把本次要插入的数据分开由两个evbuffer\_chain存放。

        具体的实现如下面所示：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39290721) [copy](https://blog.csdn.net/luotuo44/article/details/39290721)

1. //buffer.c文件
2. **int**
3. evbuffer\_add(**struct** evbuffer \*buf, **const** **void** \*data\_in, **size\_t** datlen)
4. {
5. **struct** evbuffer\_chain \*chain, \*tmp;
6. **const** unsigned **char** \*data = data\_in;
7. **size\_t** remain, to\_alloc;
8. **int** result = -1;
10. EVBUFFER\_LOCK(buf);//加锁,线程安全
12. //冻结缓冲区尾部，禁止追加数据
13. **if** (buf->freeze\_end) {
14. **goto** done;
15. }
17. //找到最后一个evbuffer\_chain。
18. chain = buf->last;
20. //第一次插入数据时，buf->last为NULL
21. **if** (chain == NULL) {
22. chain = evbuffer\_chain\_new(datlen);
23. **if** (!chain)
24. **goto** done;
25. evbuffer\_chain\_insert(buf, chain);
26. }
28. //EVBUFFER\_IMMUTABLE 是 read-only chain
29. **if** ((chain->flags & EVBUFFER\_IMMUTABLE) == 0) {//等于0说明是可以写的
30. //最后那个chain可以放的字节数
31. remain = (**size\_t**)(chain->buffer\_len - chain->misalign - chain->off);
32. **if** (remain >= datlen) {//最后那个chain可以放下本次要插入的数据
34. memcpy(chain->buffer + chain->misalign + chain->off,
35. data, datlen);
36. chain->off += datlen;//偏移量，方便下次插入数据
37. buf->total\_len += datlen;//buffer的总字节数
38. **goto** out;
39. } **else** **if** (!CHAIN\_PINNED(chain) &&//该evbuffer\_chain可以修改
40. evbuffer\_chain\_should\_realign(chain, datlen)) {
41. //通过调整后，也可以放得下本次要插入的数据
43. //通过使用chain->misalign这个错位空间而插入数据
44. evbuffer\_chain\_align(chain);
46. memcpy(chain->buffer + chain->off, data, datlen);
47. chain->off += datlen;
48. buf->total\_len += datlen;
49. **goto** out;
50. }
51. } **else** {
52. remain = 0; //最后一个节点是只写evbuffer\_chain
53. }
55. //当这个evbuffer\_chain是一个read-only buffer或者最后那个chain
56. //放不下本次要插入的数据时才会执行下面代码
57. //此时需要新建一个evbuffer\_chain
58. to\_alloc = chain->buffer\_len;
59. //当最后evbuffer\_chain的缓冲区小于等于2048时，那么新建的evbuffer\_chain的
60. //大小将是最后一个节点缓冲区的2倍。
61. **if** (to\_alloc <= EVBUFFER\_CHAIN\_MAX\_AUTO\_SIZE/2)//4096/2
62. to\_alloc <<= 1;
64. //最后的大小还是有要插入的数据决定。要注意的是虽然to\_alloc最后的值可能为
65. //datlen。但在evbuffer\_chain\_new中，实际分配的内存大小必然是512的倍数。
66. **if** (datlen > to\_alloc)
67. to\_alloc = datlen;
69. //此时需要new一个chain才能保存本次要插入的数据
70. tmp = evbuffer\_chain\_new(to\_alloc);
71. **if** (tmp == NULL)
72. **goto** done;
74. //链表最后那个节点还是可以放下一些数据的。那么就先填满链表最后那个节点
75. **if** (remain) {
76. memcpy(chain->buffer + chain->misalign + chain->off,
77. data, remain);
78. chain->off += remain;
79. buf->total\_len += remain;
80. buf->n\_add\_for\_cb += remain;
81. }
83. data += remain;//要插入的数据指针
84. datlen -= remain;
86. //把要插入的数据复制到新建一个chain中。
87. memcpy(tmp->buffer, data, datlen);
88. tmp->off = datlen;
89. //将这个chain插入到evbuffer中
90. evbuffer\_chain\_insert(buf, tmp);
91. buf->n\_add\_for\_cb += datlen;
93. out:
94. evbuffer\_invoke\_callbacks(buf);//调用回调函数
95. result = 0;
96. done:
97. EVBUFFER\_UNLOCK(buf);//解锁
98. **return** result;
99. }

        可以看到，evbuffer\_add函数是复制一份数据，保存在链表中。这样做的好处是，用户调用该函数后，就可以丢弃该数据。读者比较熟知的函数bufferevent\_write就是直接调用这个函数。当用户调用bufferevent\_write后，就可以马上把数据丢弃，无需等到Libevent把这份数据写到socket的缓存区中。

        前面的代码是把数据存放到evbuffer\_chain中，至于怎么把evbuffer\_chain插入到链表中，则是由函数evbuffer\_chain\_insert完成。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39290721) [copy](https://blog.csdn.net/luotuo44/article/details/39290721)

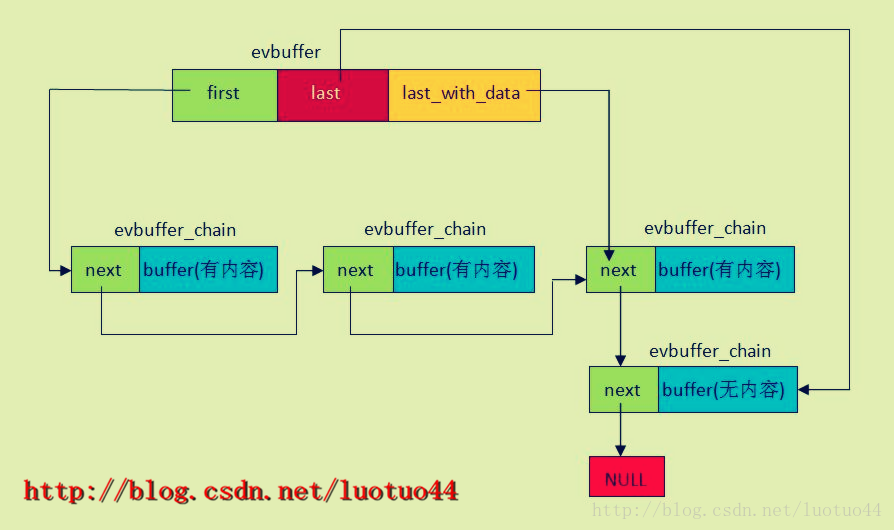
1. //buffer.c文件
2. **static** **void**
3. evbuffer\_chain\_insert(**struct** evbuffer \*buf,
4. **struct** evbuffer\_chain \*chain)
5. {
6. //新建evbuffer时是把整个evbuffer结构体都赋值0，
7. //并有buffer->last\_with\_datap = &buffer->first;
8. //所以\*buf->last\_with\_datap就是first的值，所以一开始为NULL
9. **if** (\*buf->last\_with\_datap == NULL) {
10. buf->first = buf->last = chain;
11. } **else** {
12. **struct** evbuffer\_chain \*\*ch = buf->last\_with\_datap;
13. /\* Find the first victim chain.  It might be \*last\_with\_datap \*/
14. //(\*ch)->off != 0表示该evbuffer\_chain有数据了
15. //CHAIN\_PINNED(\*ch)则表示该evbuffer\_chain不能被修改
16. //在链表中寻找到一个可以使用的evbuffer\_chain.
17. //可以使用是指该chain没有数据并且可以修改。
18. **while** ((\*ch) && ((\*ch)->off != 0 || CHAIN\_PINNED(\*ch)))
19. ch = &(\*ch)->next;//取的还是next地址。 这样看&((\*ch)->next)更清晰
21. //在已有的链表中找不到一个满足条件的evbuffer\_chain。一般都是这种情况
22. **if** (\*ch == NULL) {
23. /\* There is no victim; just append this new chain. \*/
24. //此时buf->last指向的chain不再是最后了。因为last->next被赋值了
25. buf->last->next = chain;
27. **if** (chain->off)//要插入的这个chain是有数据的
28. buf->last\_with\_datap = &buf->last->next;//last\_with\_datap指向的是倒数第二个有数据的chain的next
29. } **else** {//这种情况得到的链表可以参考下图
30. /\* Replace all victim chains with this chain. \*/
31. //断言，从这个节点开始，后面的说有节点都是没有数据的
32. EVUTIL\_ASSERT(evbuffer\_chains\_all\_empty(\*ch));
33. //释放从这个节点开始的余下链表节点
34. evbuffer\_free\_all\_chains(\*ch);
36. //把这个chain插入到最后
37. \*ch = chain;
38. }
39. buf->last = chain;//重新设置last指针，让它指向最后一个chain
40. }
41. buf->total\_len += chain->off;
42. }
44. **static** **void**
45. evbuffer\_free\_all\_chains(**struct** evbuffer\_chain \*chain)
46. {
47. **struct** evbuffer\_chain \*next;
48. **for** (; chain; chain = next) {//遍历余下的链表，删除之
49. next = chain->next;
50. evbuffer\_chain\_free(chain);
51. }
52. }

55. **static** **inline** **void**
56. evbuffer\_chain\_free(**struct** evbuffer\_chain \*chain)
57. {
58. ...//特殊buffer缓冲数据。一般的不用这些操作。直接释放内存即可
59. mm\_free(chain);
60. }

        可以看到，evbuffer\_chain\_insert的插入并不是已经一个简单的链表插入，还要检测链表里面是否有没有数据(off为0)的节点。但这个buffer链表里面会有这样的节点吗？其实是有这样节点，这种节点一般是用于预留空间的。预留空间这个概念在STL中是很常见的，它的主要作用是使得当下次添加数据时，无需额外申请空间就能保存数据。

## ****预留buffer空间：****

        其中一个扩大预留空间的函数是evbuffer\_expand。在讲evbuffer\_expand前，看一下如果存在没有数据(off为0)的节点，链表又会是怎么样的。这涉及到last\_with\_data指针的指向，如下图所示：



        好了，现在来说一下evbuffer\_expand。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39290721) [copy](https://blog.csdn.net/luotuo44/article/details/39290721)

1. //buffer.c文件
2. **int**
3. evbuffer\_expand(**struct** evbuffer \*buf, **size\_t** datlen)
4. {
5. **struct** evbuffer\_chain \*chain;
7. EVBUFFER\_LOCK(buf);//加锁
8. chain = evbuffer\_expand\_singlechain(buf, datlen);
9. EVBUFFER\_UNLOCK(buf);//解释
10. **return** chain ? 0 : -1;
11. }

        该函数的作用是扩大链表的buffer空间，使得下次add一个长度为datlen的数据时，无需动态申请内存。

        由于确保的是无需动态申请内存，所以假如这个链表本身还有大于datlen的空闲空间，那么这个evbuffer\_expand函数将不做任何操作。

        如果这个链表的所有buffer空间都被用完了，那么解决需要创建一个buffer为datlen的evbuffer\_chain，然后把这个evbuffer\_chain插入到链表最后面即可。此时这个evbuffer\_chain的off就等于0了，也就出现了前面说的的那个问题。

        如果链表的最后一个有数据chain还有一些空闲空间，但小于datlen。那么就有点麻烦。evbuffer\_expand 是调用evbuffer\_expand\_singlechain实现扩大空间的。而evbuffer\_expand\_singlechain函数有一个特点，预留空间datlen必须是在一个evbuffer\_chain中，不能跨chain。该函数的返回值就指明了哪个chain预留了datlen空间。不能跨chain也就导致了一些麻烦事。

        由于不能跨chain，但最后一个chain确实又还有一些空闲空间。前面的evbuffer\_add函数会把链表的所有节点的buffer都填得满满的。这说明所有节点的buffer还是用完的好，比较统一。要明确的是，此种情况下，肯定是要新建一个evbuffer\_chain插入到后面。

        Libevent还是想把所有节点的buffer都填满。如果最后一个chain的数据比较少，那么就直接不要那个chain。当然chain上的数据还是要的。Libevent新建一个比datlen更大的chain，把最后一个chain上的数据迁移到这个新建的chain上。这样就既能保证该chain节点也能填满，也保证了预留空间datlen必须在是一个chain的。如果最后一个chain的数据比较多，Libevent就认为迁移不划算，那么Libevent就让这个chain最后留有一些空间不使用。

        下面是该函数的代码展示了上面所说的：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39290721) [copy](https://blog.csdn.net/luotuo44/article/details/39290721)

1. //buffer.c文件
3. #define MAX\_TO\_COPY\_IN\_EXPAND 4096
4. //计算evbuffer\_chain的可用空间是多少
5. #define CHAIN\_SPACE\_LEN(ch) ((ch)->flags & EVBUFFER\_IMMUTABLE ? \
6. 0 : (ch)->buffer\_len - ((ch)->misalign + (ch)->off))
8. **static** **struct** evbuffer\_chain \*
9. evbuffer\_expand\_singlechain(**struct** evbuffer \*buf, **size\_t** datlen)
10. {
11. **struct** evbuffer\_chain \*chain, \*\*chainp;
12. **struct** evbuffer\_chain \*result = NULL;
13. ASSERT\_EVBUFFER\_LOCKED(buf);
15. chainp = buf->last\_with\_datap;
17. //\*chainp指向最后一个有数据的evbuffer\_chain或者为NULL
18. **if** (\*chainp && CHAIN\_SPACE\_LEN(\*chainp) == 0)//CHAIN\_SPACE\_LEN该chain可用空间的大小
19. chainp = &(\*chainp)->next;
21. //经过上面的那个if后，当最后一个有数据的evbuffer\_chain还有空闲空间时
22. //\*chainp就指向之。否则\*chainp指向最后一个有数据的evbuffer\_chain的next。
24. chain = \*chainp;
26. **if** (chain == NULL ||//这个chain是不可修改的，那么就只能插入一个新的chain了
27. (chain->flags & (EVBUFFER\_IMMUTABLE|EVBUFFER\_MEM\_PINNED\_ANY))) {
28. **goto** insert\_new;
29. }
31. **if** (CHAIN\_SPACE\_LEN(chain) >= datlen) {//这个chain的可用空间大于扩展空间
32. result = chain;
33. //这种情况，Libevent并不会扩大buffer空间.因为Libevent认为现在的可用空间可以用作用户提出的预留空间
34. **goto** ok;
35. }
37. **if** (chain->off == 0) {//当前一个chain存满了时，就会出现这种情况
38. **goto** insert\_new;//插入一个新的chain
39. }
41. //通过使用misalign错位空间，也能使得可用空间大于等于预留空间，那么也不用
42. //扩大buffer空间
43. **if** (evbuffer\_chain\_should\_realign(chain, datlen)) {
44. evbuffer\_chain\_align(chain);
45. result = chain;
46. **goto** ok;
47. }

50. //空闲空间小于总空间的1/8 或者 已有的数据量大于MAX\_TO\_COPY\_IN\_EXPAND(4096)
51. **if** (CHAIN\_SPACE\_LEN(chain) < chain->buffer\_len / 8 ||
52. chain->off > MAX\_TO\_COPY\_IN\_EXPAND) {//4096
54. //本chain有比较多的数据，将这些数据迁移到另外一个chain是不划算的
55. //此时，将不会改变这个chain。
57. //下一个chain是否可以有足够的空闲空间.有则直接用之
58. **if** (chain->next && CHAIN\_SPACE\_LEN(chain->next) >= datlen) {
59. result = chain->next;
60. **goto** ok;
61. } **else** {
62. **goto** insert\_new;
63. }
64. } **else** {
65. //由于本chain的数据量比较小，所以把这个chain的数据迁移到另外一个
66. //chain上是值得的。
67. **size\_t** length = chain->off + datlen;
68. **struct** evbuffer\_chain \*tmp = evbuffer\_chain\_new(length);
69. **if** (tmp == NULL)
70. **goto** err;
72. tmp->off = chain->off;
73. //进行数据迁移
74. memcpy(tmp->buffer, chain->buffer + chain->misalign,
75. chain->off);
76. EVUTIL\_ASSERT(\*chainp == chain);
77. result = \*chainp = tmp;
79. **if** (buf->last == chain)
80. buf->last = tmp;
82. tmp->next = chain->next;
83. evbuffer\_chain\_free(chain);
84. **goto** ok;
85. }
87. insert\_new:
88. result = evbuffer\_chain\_insert\_new(buf, datlen);
89. **if** (!result)
90. **goto** err;
91. ok:
92. EVUTIL\_ASSERT(result);
93. EVUTIL\_ASSERT(CHAIN\_SPACE\_LEN(result) >= datlen);
94. err:
95. **return** result;
96. }

99. **static** **inline** **struct** evbuffer\_chain \*
100. evbuffer\_chain\_insert\_new(**struct** evbuffer \*buf, **size\_t** datlen)
101. {
102. **struct** evbuffer\_chain \*chain;
103. **if** ((chain = evbuffer\_chain\_new(datlen)) == NULL)
104. **return** NULL;
105. evbuffer\_chain\_insert(buf, chain);
106. **return** chain;
107. }

        上面代码中evbuffer\_expand\_singlechain函数的第一个if语句，可以联合前面的两张图一起看，更容易看懂。

        evbuffer\_expand\_singlechain函数是要求一个节点就能提供大小为datlen的可用空间。其实Libevent还提供了\_evbuffer\_expand\_fast函数，该函数还有一个整型的参数n，用来表示使用不超过n个节点的前提下，提供datlen的可用空间。不过这个函数只留给Libevent内部使用，用户不能使用之。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39290721) [copy](https://blog.csdn.net/luotuo44/article/details/39290721)

1. //buffer.c文件
2. **int**//用最多不超过n个节点就提供datlen大小的空闲空间。链表过长是不好的
3. \_evbuffer\_expand\_fast(**struct** evbuffer \*buf, **size\_t** datlen, **int** n)
4. {
5. **struct** evbuffer\_chain \*chain = buf->last, \*tmp, \*next;
6. **size\_t** avail;
7. **int** used;
9. EVUTIL\_ASSERT(n >= 2); //n必须大于等于2
11. //最后一个节点是不可用的
12. **if** (chain == NULL || (chain->flags & EVBUFFER\_IMMUTABLE)) {
13. //这种情况下，直接新建一个足够大的evbuffer\_chain即可
14. chain = evbuffer\_chain\_new(datlen);
15. **if** (chain == NULL)
16. **return** (-1);
18. evbuffer\_chain\_insert(buf, chain);
19. **return** (0);
20. }
22. used = 0; /\* number of chains we're using space in. \*/
23. avail = 0; /\* how much space they have. \*/
24. **for** (chain = \*buf->last\_with\_datap; chain; chain = chain->next) {
25. **if** (chain->off) {//最后一个有数据的节点的可用空间也是要被使用
26. **size\_t** space = (**size\_t**) CHAIN\_SPACE\_LEN(chain);
27. EVUTIL\_ASSERT(chain == \*buf->last\_with\_datap);
28. **if** (space) {
29. avail += space;
30. ++used;
31. }
32. } **else** {//链表中off为0的空buffer统统使用
33. /\* No data in chain; realign it. \*/
34. chain->misalign = 0;
35. avail += chain->buffer\_len;
36. ++used;
37. }
38. **if** (avail >= datlen) {//链表中的节点的可用空间已经足够了
39. **return** (0);
40. }
41. **if** (used == n)//到达了最大可以忍受的链表长度
42. **break**;
43. }
45. //前面的for循环，如果找够了空闲空间，那么是直接return。所以
46. //运行到这里时，就说明还没找到空闲空间。一般是因为链表后面的off等于0
47. //的节点已经被用完了都还不能满足datlen
48. **if** (used < n) {
49. EVUTIL\_ASSERT(chain == NULL);
51. //申请一个足够大的evbuffer\_chain，把空间补足
52. tmp = evbuffer\_chain\_new(datlen - avail);
53. **if** (tmp == NULL)
54. **return** (-1);
56. buf->last->next = tmp;
57. buf->last = tmp;
58. **return** (0);
59. } **else** { //used == n。把后面的n个节点都用了还是不够datlen空间
60. //链表后面的n个节点都用上了，这个n个节点中，至少有n-1个节点的off等于
61. //0。n个节点都不够，Libevent就认为这些节点都是饭桶，Libevent会统统删除
62. //然后新建一个足够大的evbuffer\_chain。
64. //用来标志该链表的所有节点都是off为0的。在这种情况下，将删除所有的节点
65. **int** rmv\_all = 0; /\* True iff we removed last\_with\_data. \*/
66. chain = \*buf->last\_with\_datap;
67. **if** (!chain->off) {
68. //这说明链表中的节点都是没有数据的evbuffer\_chain
69. EVUTIL\_ASSERT(chain == buf->first);
70. rmv\_all = 1;//标志之
71. avail = 0;
72. } **else** {
73. //最后一个有数据的chain的可用空间的大小。这个空间是可以用上的
74. avail = (**size\_t**) CHAIN\_SPACE\_LEN(chain);
75. chain = chain->next;
76. }

79. //chain指向第一个off等于0的evbuffer\_chain 或者等于NULL
81. //将这些off等于0的evbuffer\_chain统统free掉，不要了。
82. //然后new一个足够大的evbuffer\_chain即可。这能降低链表的长度
83. **for** (; chain; chain = next) {
84. next = chain->next;
85. EVUTIL\_ASSERT(chain->off == 0);
86. evbuffer\_chain\_free(chain);
87. }
89. //new一个足够大的evbuffer\_chain
90. tmp = evbuffer\_chain\_new(datlen - avail);
91. **if** (tmp == NULL) {//new失败
92. **if** (rmv\_all) {//这种情况下，该链表就根本没有节点了
93. ZERO\_CHAIN(buf);//相当于初始化evbuffer的链表
94. } **else** {
95. buf->last = \*buf->last\_with\_datap;
96. (\*buf->last\_with\_datap)->next = NULL;
97. }
98. **return** (-1);
99. }
101. **if** (rmv\_all) {//这种情况下，该链表就只有一个节点了
102. buf->first = buf->last = tmp;
103. buf->last\_with\_datap = &buf->first;
104. } **else** {
105. (\*buf->last\_with\_datap)->next = tmp;
106. buf->last = tmp;
107. }
108. **return** (0);
109. }
110. }

## ****在链表头添加数据：****

        前面的evbuffer\_add是在链表尾部追加数据，Libevent提供了另外一个函数evbuffer\_prepend可以在链表头部添加数据。在这个函数里面可以看到evbuffer\_chain结构体成员misalign的一些使用，也能知道为什么会有这个成员。

        evbuffer\_prepend函数并不复杂，只需弄懂misalign的作用就很容易明白该函数的实现。考虑这种情况：要在链表头插入数据，那么应该new一个新的evbuffer\_chain，然后把要插入的数据放到这个新建个的evbuffer\_chain中。但evbuffer\_chain\_new申请到的buffer空间可能会大于要插入的数据长度。插入数据后，buffer就必然会剩下一些空闲空间。那么这个空闲空间放在buffer的前面好还是后面好呢？Libevent认为放在前面会好些，此时misalign就有用了。它表示错开不用的空间，也就是空闲空间。如果再次在链表头插入数据，就可以使用到这些空闲空间了。所以，misalign也可以认为是空闲空间，可以随时使用。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39290721) [copy](https://blog.csdn.net/luotuo44/article/details/39290721)

1. //buffer.c文件
2. **int**
3. evbuffer\_prepend(**struct** evbuffer \*buf, **const** **void** \*data, **size\_t** datlen)
4. {
5. **struct** evbuffer\_chain \*chain, \*tmp;
6. **int** result = -1;
8. EVBUFFER\_LOCK(buf);
10. //冻结缓冲区头部，禁止在头部添加数据
11. **if** (buf->freeze\_start) {
12. **goto** done;
13. }
15. chain = buf->first;
17. //该链表暂时还没有节点
18. **if** (chain == NULL) {
19. chain = evbuffer\_chain\_new(datlen);
20. **if** (!chain)
21. **goto** done;
22. evbuffer\_chain\_insert(buf, chain);
23. }
25. **if** ((chain->flags & EVBUFFER\_IMMUTABLE) == 0) {//该chain可以修改
26. /\* If this chain is empty, we can treat it as
27. \* 'empty at the beginning' rather than 'empty at the end' \*/
28. **if** (chain->off == 0)
29. chain->misalign = chain->buffer\_len;
31. //考虑这种情况:一开始chain->off等于0，之后调用evbuffer\_prepend插入
32. //一些数据(还没填满这个chain),之后再次调用evbuffer\_prepend插入一些
33. //数据。这样就能分别进入下面的if else了
35. **if** ((**size\_t**)chain->misalign >= datlen) {//空闲空间足够大
36. memcpy(chain->buffer + chain->misalign - datlen,
37. data, datlen);
38. chain->off += datlen;
39. chain->misalign -= datlen;
40. buf->total\_len += datlen;
41. buf->n\_add\_for\_cb += datlen;
42. **goto** out;
43. } **else** **if** (chain->misalign) {//不够大，但也要用
44. memcpy(chain->buffer,//用完这个chain,所以从头开始
45. (**char**\*)data + datlen - chain->misalign,
46. (**size\_t**)chain->misalign);
47. chain->off += (**size\_t**)chain->misalign;
48. buf->total\_len += (**size\_t**)chain->misalign;
49. buf->n\_add\_for\_cb += (**size\_t**)chain->misalign;
50. datlen -= (**size\_t**)chain->misalign;
51. chain->misalign = 0;
52. }
53. }

56. //为datlen申请一个evbuffer\_chain。把datlen长的数据放到这个新建的chain
57. **if** ((tmp = evbuffer\_chain\_new(datlen)) == NULL)
58. **goto** done;
59. buf->first = tmp;
60. **if** (buf->last\_with\_datap == &buf->first)
61. buf->last\_with\_datap = &tmp->next;
63. tmp->next = chain;
65. tmp->off = datlen;
66. tmp->misalign = tmp->buffer\_len - datlen;
68. memcpy(tmp->buffer + tmp->misalign, data, datlen);
69. buf->total\_len += datlen;
70. buf->n\_add\_for\_cb += (**size\_t**)chain->misalign;
72. out:
73. evbuffer\_invoke\_callbacks(buf);//调用回调函数
74. result = 0;
75. done:
76. EVBUFFER\_UNLOCK(buf);
77. **return** result;
78. }

## ****读取数据：****

        现在来看一下怎么从evbuffer中复制一些数据。Libevent提供了函数evbuffer\_copyout用来复制evbuffer的数据。当然是从链表的前面开始复制。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39290721) [copy](https://blog.csdn.net/luotuo44/article/details/39290721)

1. //buffer.c文件
2. ev\_ssize\_t
3. evbuffer\_copyout(**struct** evbuffer \*buf, **void** \*data\_out, **size\_t** datlen)
4. {
5. **struct** evbuffer\_chain \*chain;
6. **char** \*data = data\_out;
7. **size\_t** nread;
8. ev\_ssize\_t result = 0;
10. EVBUFFER\_LOCK(buf);
12. chain = buf->first;
14. **if** (datlen >= buf->total\_len)
15. datlen = buf->total\_len;//最大能提供的数据
17. **if** (datlen == 0)
18. **goto** done;
20. //冻结缓冲区头部，禁止读取缓冲区的数据
21. **if** (buf->freeze\_start) {
22. result = -1;
23. **goto** done;
24. }
26. nread = datlen;
27. **while** (datlen && datlen >= chain->off) {
28. memcpy(data, chain->buffer + chain->misalign, chain->off);
29. data += chain->off;
30. datlen -= chain->off;
32. chain = chain->next;
33. }
35. **if** (datlen) {
36. memcpy(data, chain->buffer + chain->misalign, datlen);
37. }
39. result = nread;
40. done:
41. EVBUFFER\_UNLOCK(buf);
42. **return** result;
43. }

        这个函数逻辑比较简单，这里就不多讲了。

        有时我们不仅仅想复制数据，还想删除数据，或者是复制后就删除数据。这些操作在socket编程中还是很常见的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39290721) [copy](https://blog.csdn.net/luotuo44/article/details/39290721)

1. //buffer.c文件
2. **int**
3. evbuffer\_drain(**struct** evbuffer \*buf, **size\_t** len)
4. {
5. **struct** evbuffer\_chain \*chain, \*next;
6. **size\_t** remaining, old\_len;
7. **int** result = 0;
9. EVBUFFER\_LOCK(buf);
10. old\_len = buf->total\_len;
12. **if** (old\_len == 0)
13. **goto** done;
15. //冻结缓冲区头部，禁止删除头部数据
16. **if** (buf->freeze\_start) {
17. result = -1;
18. **goto** done;
19. }
21. //要删除的数据量大于等于已有的数据量。并且这个evbuffer是可以删除的
22. **if** (len >= old\_len && !HAS\_PINNED\_R(buf)) {
23. len = old\_len;
24. **for** (chain = buf->first; chain != NULL; chain = next) {
25. next = chain->next;
26. evbuffer\_chain\_free(chain);
27. }
29. ZERO\_CHAIN(buf);//相当于初试化evbuffer的链表
30. } **else** {
31. **if** (len >= old\_len)
32. len = old\_len;
34. buf->total\_len -= len;
35. remaining = len;
36. **for** (chain = buf->first;
37. remaining >= chain->off;
38. chain = next) {
39. next = chain->next;
40. remaining -= chain->off;
42. //已经删除到最后一个有数据的evbuffer\_chain了
43. **if** (chain == \*buf->last\_with\_datap) {
44. buf->last\_with\_datap = &buf->first;
45. }
47. //删除到倒数第二个有数据的evbuffer\_chain
48. **if** (&chain->next == buf->last\_with\_datap)
49. buf->last\_with\_datap = &buf->first;
51. //这个chain被固定了，不能删除
52. **if** (CHAIN\_PINNED\_R(chain)) {
53. EVUTIL\_ASSERT(remaining == 0);
54. chain->misalign += chain->off;
55. chain->off = 0;
56. **break**;//后面的evbuffer\_chain也是固定的
57. } **else**
58. evbuffer\_chain\_free(chain);
59. }
61. buf->first = chain;
62. **if** (chain) {
63. chain->misalign += remaining;
64. chain->off -= remaining;
65. }
66. }
68. evbuffer\_invoke\_callbacks(buf);//因为删除数据，所以也要调用回调函数
69. done:
70. EVBUFFER\_UNLOCK(buf);
71. **return** result;
72. }

75. **int**
76. evbuffer\_remove(**struct** evbuffer \*buf, **void** \*data\_out, **size\_t** datlen)
77. {
78. ev\_ssize\_t n;
79. EVBUFFER\_LOCK(buf);
80. n = evbuffer\_copyout(buf, data\_out, datlen);
81. **if** (n > 0) {
82. **if** (evbuffer\_drain(buf, n)<0)
83. n = -1;
84. }
85. EVBUFFER\_UNLOCK(buf);
86. **return** (**int**)n;
87. }

        可以看到evbuffer\_remove是先复制数据，然后才删除evbuffer的数据。而evbuffer\_drain则直接删除evbuffer的数据，而不会复制。

        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/39290721