<https://blog.csdn.net/luotuo44/article/details/38403241>

**转载请注明出处:**[**http://blog.csdn.net/luotuo44/article/details/38403241**](http://blog.csdn.net/luotuo44/article/details/38403241)

[上一篇博客](http://blog.csdn.net/luotuo44/article/details/38374009)说到了TAILQ\_QUEUE队列，它可以把多个event结构体连在一起。是一种归类方式。本文也将讲解一种将event归类、连在一起的结构：哈希结构。

# 哈希结构体：

        哈希结构由下面几个结构体一起配合工作：

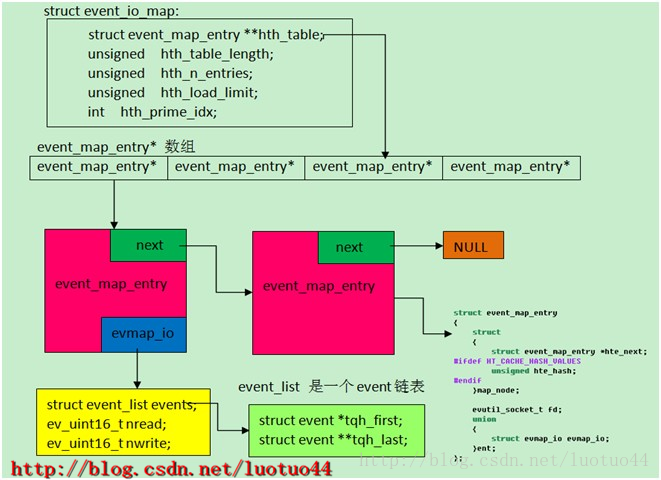
**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38403241) [copy](https://blog.csdn.net/luotuo44/article/details/38403241)

1. **struct** event\_list
2. {
3. **struct** event \*tqh\_first;
4. **struct** event \*\*tqh\_last;
5. };
7. **struct** evmap\_io {
8. //TAILQ\_HEAD (event\_list, event);
9. **struct** event\_list events;
10. ev\_uint16\_t nread;
11. ev\_uint16\_t nwrite;
12. };
14. **struct** event\_map\_entry {
15. HT\_ENTRY(event\_map\_entry) map\_node; //next指针
16. evutil\_socket\_t fd;
17. **union** { /\* This is a union in case we need to make more things that can
18. be in the hashtable. \*/
19. **struct** evmap\_io evmap\_io;
20. } ent;
21. };
23. **struct** event\_io\_map
24. {
25. //哈希表
26. **struct** event\_map\_entry \*\*hth\_table;
27. //哈希表的长度
28. unsigned hth\_table\_length;
29. //哈希的元素个数
30. unsigned hth\_n\_entries;
31. //resize 之前可以存多少个元素
32. //在event\_io\_map\_HT\_GROW函数中可以看到其值为hth\_table\_length的
33. //一半。但hth\_n\_entries>=hth\_load\_limit时，就会发生增长哈希表的长度
34. unsigned hth\_load\_limit;
35. //后面素数表中的下标值。主要是指明用到了哪个素数
36. **int** hth\_prime\_idx;
37. };

        结构体event\_io\_map指明了哈希表的存储位置、哈希表的长度、元素个数等数据。该哈希表是使用链地址法解决冲突问题的，这一点可以从hth\_talbe成员变量看到。它是一个二级指针，因为哈希表的元素是event\_map\_entry指针。

        除了怎么解决哈希冲突外，哈希表还有一个问题要解决，那就是哈希函数。这里的哈希函数就是模(%)。用event\_map\_entry结构体中的fd成员值模 event\_io\_map结构体中的hth\_table\_length。

        由上面那些结构体配合得到的哈希表结构如下图所示：



        从上图可以看到，两个发生了冲突的哈希表元素event\_map\_entry用一个next指向连在一起了(链地址解决冲突)。

        另外，从图或者从前面关于event\_map\_entry结构体的定义可以看到，它有一个evmap\_io结构体。而这个evmap\_io结构体又有一个struct event\_list 类型的成员，而struct event\_list类型就是一个TAILQ\_HEAD。这正是[前一篇博客](http://blog.csdn.net/luotuo44/article/details/38374009)说到的TAILQ\_QUEUE队列的队列头。从这一点可以看到，这个哈希结构还是比较复杂的。

        为什么在哈希表的元素里面，还会有一个TAILQ\_QUEUE队列呢？这得由Libevent的一个特征说起。Libevent允许用同一个文件描述符fd或者信号值，调用event\_new、event\_add多次。所以，同一个fd或者信号值就可以对应多个event结构体了。所以这个TAILQ\_QUEUE队列就是将这些具有相同fd或者信号值的多个event结构体连一起。

# 什么情况会使用哈希表：

        有一点需要说明，那就是Libevent中的哈希表只会用于Windows系统，像遵循POSIX标准的OS是不会用到哈希表的。从下面的定义可以看到这一点。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38403241) [copy](https://blog.csdn.net/luotuo44/article/details/38403241)

1. //event-internal.h文件
2. #ifdef WIN32
3. /\* If we're on win32, then file descriptors are not nice low densely packed
4. integers.  Instead, they are pointer-like windows handles, and we want to
5. use a hashtable instead of an array to map fds to events.
6. \*/
7. #define EVMAP\_USE\_HT
8. #endif

11. #ifdef EVMAP\_USE\_HT
12. #include "ht-internal.h"
13. **struct** event\_map\_entry;
14. HT\_HEAD(event\_io\_map, event\_map\_entry);
15. #else
16. #define event\_io\_map event\_signal\_map
17. #endif

        可以看到，如果是非Windows系统，那么event\_io\_map就会被定义成event\_signal\_map(这是一个很简单的结构)。而在Windows系统，那么就由HT\_HEAD这个宏定义event\_io\_map。最后得到的event\_io\_map就是本文最前面所示的那样。

        为什么只有在Windows系统才会使用这个哈希表呢？代码里面的注释给出了一些解释。因为在Windows系统里面，文件描述符是一个比较大的值，不适合放到event\_signal\_map结构中。而通过哈希(模上一个小的值)，就可以变得比较小，这样就可以放到哈希表的数组中了。而遵循POSIX标准的文件描述符是从0开始递增的，一般都不会太大，适用于event\_signal\_map。

# 哈希函数：

        前面说到哈希函数是 用文件描述符fd 模 哈希表的长度。实际上，并不是直接用fd，而是用一个叫hashsocket的函数将这个fd进行一些处理后，才去模 哈希表的长度。下面是hashsocket函数的实现：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38403241) [copy](https://blog.csdn.net/luotuo44/article/details/38403241)

1. //evmap.c文件
2. /\* Helper used by the event\_io\_map hashtable code; tries to return a good hash
3. \* of the fd in e->fd. \*/
4. **static** **inline** unsigned
5. hashsocket(**struct** event\_map\_entry \*e)
6. {
7. /\* On win32, in practice, the low 2-3 bits of a SOCKET seem not to
8. \* matter.  Our hashtable implementation really likes low-order bits,
9. \* though, so let's do the rotate-and-add trick. \*/
10. unsigned h = (unsigned) e->fd;
11. h += (h >> 2) | (h << 30);
12. **return** h;
13. }

        前面的event\_map\_entry结构体中，还有一个HT\_ENTRY的宏。从名字来看，它是一个哈希表的表项。它是一个条件宏，定义如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38403241) [copy](https://blog.csdn.net/luotuo44/article/details/38403241)

1. //ht-internal.h文件
2. #ifdef HT\_CACHE\_HASH\_VALUES
3. #define HT\_ENTRY(type)                          \
4. **struct** {                                      \
5. **struct** type \*hte\_next;                      \
6. unsigned hte\_hash;                          \
7. }
8. #else
9. #define HT\_ENTRY(type)                          \
10. **struct** {                                      \
11. **struct** type \*hte\_next;                      \
12. }
13. #endif

        可以看到，如果定义了HT\_CACHE\_HASH\_VALUES宏，那么就会多一个hte\_hash变量。从宏的名字来看，这是一个cache。不错，变量hte\_hash就是用来存储前面的hashsocket的返回值。当第一次计算得到后，就存放到hte\_hash变量中。以后需要用到(会经常用到)，就直接向这个变量要即可，无需再次计算hashsocket函数。如果没有这个变量，那么需要用到这个值，都要调用hashsocket函数计算一次。这一点从后面的代码可以看到。

# 哈希表操作函数：

        ht-internal.h文件里面定义了一系列的哈希函数的操作函数。下来就列出这些函数。如果打开ht-internal.h文件，就发现，它是宏的天下。该文件的函数都是由宏定义生成的。下面就贴出宏定义展开后的函数。同前一篇博文那样，是用gcc的-E选项展开的。下面的代码比较长，要有心理准备。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38403241) [copy](https://blog.csdn.net/luotuo44/article/details/38403241)

1. **struct** event\_list
2. {
3. **struct** event \*tqh\_first;
4. **struct** event \*\*tqh\_last;
5. };
7. **struct** evmap\_io
8. {
9. **struct** event\_list events;
10. ev\_uint16\_t nread;
11. ev\_uint16\_t nwrite;
12. };

15. **struct** event\_map\_entry
16. {
17. **struct**
18. {
19. **struct** event\_map\_entry \*hte\_next;
20. #ifdef HT\_CACHE\_HASH\_VALUES
21. unsigned hte\_hash;
22. #endif
23. }map\_node;
25. evutil\_socket\_t fd;
26. **union**
27. {
28. **struct** evmap\_io evmap\_io;
29. }ent;
30. };
32. **struct** event\_io\_map
33. {
34. //哈希表
35. **struct** event\_map\_entry \*\*hth\_table;
36. //哈希表的长度
37. unsigned hth\_table\_length;
38. //哈希的元素个数
39. unsigned hth\_n\_entries;
40. //resize 之前可以存多少个元素
41. //在event\_io\_map\_HT\_GROW函数中可以看到其值为hth\_table\_length的
42. //一半。但hth\_n\_entries>=hth\_load\_limit时，就会发生增长哈希表的长度
43. unsigned hth\_load\_limit;
44. //后面素数表中的下标值。主要是指明用到了哪个素数
45. **int** hth\_prime\_idx;
46. };


50. **int** event\_io\_map\_HT\_GROW(**struct** event\_io\_map \*ht, unsigned min\_capacity);
51. **void** event\_io\_map\_HT\_CLEAR(**struct** event\_io\_map \*ht);
53. **int** \_event\_io\_map\_HT\_REP\_IS\_BAD(**const** **struct** event\_io\_map \*ht);
55. //初始化event\_io\_map
56. **static** **inline** **void** event\_io\_map\_HT\_INIT(**struct** event\_io\_map \*head)
57. {
58. head->hth\_table\_length = 0;
59. head->hth\_table = NULL;
60. head->hth\_n\_entries = 0;
61. head->hth\_load\_limit = 0;
62. head->hth\_prime\_idx = -1;
63. }

66. //在event\_io\_map这个哈希表中，找个表项elm
67. //在下面还有一个相似的函数，函数名少了最后的\_P。那个函数的返回值
68. //是event\_map\_entry \*。从查找来说，后面那个函数更适合。之所以
69. //有这个函数，是因为哈希表还有replace、remove这些操作。对于
70. //A->B->C这样的链表。此时，要replace或者remove节点B。
71. //如果只有后面那个查找函数，那么只能查找并返回一个指向B的指针。
72. //此时将无法修改A的next指针了。所以本函数有存在的必要。
73. //在本文件中，很多函数都使用了event\_map\_entry \*\*。
74. //因为event\_map\_entry \*\*类型变量，既可以修改本元素的hte\_next变量
75. //也能指向下一个元素。
77. //该函数返回的是查找节点的前驱节点的hte\_next成员变量的地址。
78. //所以返回值肯定不会为NULL,但是对返回值取\*就可能为NULL
79. **static** **inline** **struct** event\_map\_entry \*\*
80. \_event\_io\_map\_HT\_FIND\_P(**struct** event\_io\_map \*head,
81. **struct** event\_map\_entry \*elm)
82. {
83. **struct** event\_map\_entry \*\*p;
84. **if** (!head->hth\_table)
85. **return** NULL;
87. #ifdef HT\_CACHE\_HASH\_VALUES
88. p = &((head)->hth\_table[((elm)->map\_node.hte\_hash)
89. % head->hth\_table\_length]);
90. #else
91. p = &((head)->hth\_table[(hashsocket(\*elm))%head->hth\_table\_length]);
92. #endif
94. //这里的哈希表是用链地址法解决哈希冲突的。
95. //上面的 % 只是找到了冲突链的头。现在是在冲突链中查找。
96. **while** (\*p)
97. {
98. //判断是否相等。在实现上，只是简单地根据fd来判断是否相等
99. **if** (eqsocket(\*p, elm))
100. **return** p;
102. //p存放的是hte\_next成员变量的地址
103. p = &(\*p)->map\_node.hte\_next;
104. }
106. **return** p;
107. }
109. /\* Return a pointer to the element in the table 'head' matching 'elm',
110. \* or NULL if no such element exists \*/
111. //在event\_io\_map这个哈希表中，找个表项elm
112. **static** **inline** **struct** event\_map\_entry \*
113. event\_io\_map\_HT\_FIND(**const** **struct** event\_io\_map \*head,
114. **struct** event\_map\_entry \*elm)
115. {
116. **struct** event\_map\_entry \*\*p;
117. **struct** event\_io\_map \*h = (**struct** event\_io\_map \*) head;
119. #ifdef HT\_CACHE\_HASH\_VALUES
120. **do**
121. {   //计算哈希值
122. (elm)->map\_node.hte\_hash = hashsocket(elm);
123. } **while**(0);
124. #endif
126. p = \_event\_io\_map\_HT\_FIND\_P(h, elm);
127. **return** p ? \*p : NULL;
128. }

131. /\* Insert the element 'elm' into the table 'head'.  Do not call this
132. \* function if the table might already contain a matching element. \*/
133. **static** **inline** **void**
134. event\_io\_map\_HT\_INSERT(**struct** event\_io\_map \*head,
135. **struct** event\_map\_entry \*elm)
136. {
137. **struct** event\_map\_entry \*\*p;
138. **if** (!head->hth\_table || head->hth\_n\_entries >= head->hth\_load\_limit)
139. event\_io\_map\_HT\_GROW(head, head->hth\_n\_entries+1);
141. ++head->hth\_n\_entries;
143. #ifdef HT\_CACHE\_HASH\_VALUES
144. **do**
145. {   //计算哈希值.此哈希不同于用%计算的简单哈希。
146. //存放到hte\_hash变量中，作为cache
147. (elm)->map\_node.hte\_hash = hashsocket(elm);
148. } **while** (0);
150. p = &((head)->hth\_table[((elm)->map\_node.hte\_hash)
151. % head->hth\_table\_length]);
152. #else
153. p = &((head)->hth\_table[(hashsocket(\*elm))%head->hth\_table\_length]);
154. #endif

157. //使用头插法，即后面才插入的链表，反而会在链表头。
158. elm->map\_node.hte\_next = \*p;
159. \*p = elm;
160. }

163. /\* Insert the element 'elm' into the table 'head'. If there already
164. \* a matching element in the table, replace that element and return
165. \* it. \*/
166. **static** **inline** **struct** event\_map\_entry \*
167. event\_io\_map\_HT\_REPLACE(**struct** event\_io\_map \*head,
168. **struct** event\_map\_entry \*elm)
169. {
170. **struct** event\_map\_entry \*\*p, \*r;
172. **if** (!head->hth\_table || head->hth\_n\_entries >= head->hth\_load\_limit)
173. event\_io\_map\_HT\_GROW(head, head->hth\_n\_entries+1);
175. #ifdef HT\_CACHE\_HASH\_VALUES
176. **do**
177. {
178. (elm)->map\_node.hte\_hash = hashsocket(elm);
179. } **while**(0);
180. #endif
182. p = \_event\_io\_map\_HT\_FIND\_P(head, elm);
184. //由前面的英文注释可知，这个函数是替换插入一起进行的。如果哈希表
185. //中有和elm相同的元素(指的是event\_map\_entry的fd成员变量值相等)
186. //那么就发生替代(其他成员变量值不同，所以不是完全相同，有替换意义)
187. //如果哈希表中没有和elm相同的元素，那么就进行插入操作
189. //指针p存放的是hte\_next成员变量的地址
190. //这里的p存放的是被替换元素的前驱元素的hte\_next变量地址
191. r = \*p; //r指向了要替换的元素。有可能为NULL
192. \*p = elm; //hte\_next变量被赋予新值elm
194. //找到了要被替换的元素r(不为NULL)
195. //而且要插入的元素地址不等于要被替换的元素地址
196. **if** (r && (r!=elm))
197. {
198. elm->map\_node.hte\_next = r->map\_node.hte\_next;
200. r->map\_node.hte\_next = NULL;
201. **return** r; //返回被替换掉的元素
202. }
203. **else** //进行插入操作
204. {
205. //这里貌似有一个bug。如果前一个判断中，r 不为 NULL，而且r == elm
206. //对于同一个元素，多次调用本函数。就会出现这样情况。
207. //此时，将会来到这个else里面
208. //实际上没有增加元素，但元素的个数却被++了。因为r 的地址等于 elm
209. //所以 r = \*p; \*p = elm; 等于什么也没有做。（r == elm)
210. //当然，因为这些函数都是Libevent内部使用的。如果它保证不会同于同
211. //一个元素调用本函数，那么就不会出现bug
212. ++head->hth\_n\_entries;
213. **return** NULL; //插入操作返回NULL，表示没有替换到元素
214. }
215. }

218. /\* Remove any element matching 'elm' from the table 'head'.  If such
219. \* an element is found, return it; otherwise return NULL. \*/
220. **static** **inline** **struct** event\_map\_entry \*
221. event\_io\_map\_HT\_REMOVE(**struct** event\_io\_map \*head,
222. **struct** event\_map\_entry \*elm)
223. {
224. **struct** event\_map\_entry \*\*p, \*r;
226. #ifdef HT\_CACHE\_HASH\_VALUES
227. **do**
228. {
229. (elm)->map\_node.hte\_hash = hashsocket(elm);
230. } **while** (0);
231. #endif
233. p = \_event\_io\_map\_HT\_FIND\_P(head,elm);
235. **if** (!p || !\*p)//没有找到
236. **return** NULL;
238. //指针p存放的是hte\_next成员变量的地址
239. //这里的p存放的是被替换元素的前驱元素的hte\_next变量地址
240. r = \*p; //r现在指向要被删除的元素
241. \*p = r->map\_node.hte\_next;
242. r->map\_node.hte\_next = NULL;
244. --head->hth\_n\_entries;
246. **return** r;
247. }

250. /\* Invoke the function 'fn' on every element of the table 'head',
251. using 'data' as its second argument.  If the function returns
252. nonzero, remove the most recently examined element before invoking
253. the function again. \*/
254. **static** **inline** **void**
255. event\_io\_map\_HT\_FOREACH\_FN(**struct** event\_io\_map \*head,
256. **int** (\*fn)(**struct** event\_map\_entry \*, **void** \*),
257. **void** \*data)
258. {
259. unsigned idx;
260. **struct** event\_map\_entry \*\*p, \*\*nextp, \*next;
262. **if** (!head->hth\_table)
263. **return**;
265. **for** (idx=0; idx < head->hth\_table\_length; ++idx)
266. {
267. p = &head->hth\_table[idx];
269. **while** (\*p)
270. {
271. //像A->B->C链表。p存放了A元素中hte\_next变量的地址
272. //\*p则指向B元素。nextp存放B的hte\_next变量的地址
273. //next指向C元素。
274. nextp = &(\*p)->map\_node.hte\_next;
275. next = \*nextp;
277. //对B元素进行检查
278. **if** (fn(\*p, data))
279. {
280. --head->hth\_n\_entries;
281. //p存放了A元素中hte\_next变量的地址
282. //所以\*p = next使得A元素的hte\_next变量值被赋值为
283. //next。此时链表变成A->C。即使抛弃了B元素。不知道
284. //调用方是否能释放B元素的内存。
285. \*p = next;
286. }
287. **else**
288. {
289. p = nextp;
290. }
291. }
292. }
293. }

296. /\* Return a pointer to the first element in the table 'head', under
297. \* an arbitrary order.  This order is stable under remove operations,
298. \* but not under others. If the table is empty, **return** NULL. \*/
299. //获取第一条冲突链的第一个元素
300. **static** **inline** **struct** event\_map\_entry \*\*
301. event\_io\_map\_HT\_START(**struct** event\_io\_map \*head)
302. {
303. unsigned b = 0;
305. **while** (b < head->hth\_table\_length)
306. {
307. //返回哈希表中，第一个不为NULL的节点
308. //即有event\_map\_entry元素的节点。
309. //找到链。因为是哈希。所以不一定哈希表中的每一个节点都存有元素
310. **if** (head->hth\_table[b])
311. **return** &head->hth\_table[b];
313. ++b;
314. }
316. **return** NULL;
317. }


321. /\* Return the next element in 'head' after 'elm', under the arbitrary
322. \* order used by HT\_START.  If there are no more elements, return
323. \* NULL.  If 'elm' is to be removed from the table, you must call
324. \* this function for the next value before you remove it.
325. \*/
326. **static** **inline** **struct** event\_map\_entry \*\*
327. event\_io\_map\_HT\_NEXT(**struct** event\_io\_map \*head,
328. **struct** event\_map\_entry \*\*elm)
329. {
330. //本哈希解决冲突的方式是链地址
331. //如果参数elm所在的链地址中，elm还有下一个节点，就直接返回下一个节点
332. **if** ((\*elm)->map\_node.hte\_next)
333. {
334. **return** &(\*elm)->map\_node.hte\_next;
335. }
336. **else** //否则取哈希表中的下一条链中第一个元素
337. {
338. #ifdef HT\_CACHE\_HASH\_VALUES
339. unsigned b = (((\*elm)->map\_node.hte\_hash)
340. % head->hth\_table\_length) + 1;
341. #else
342. unsigned b = ( (hashsocket(\*elm)) % head->hth\_table\_length) + 1;
343. #endif
345. **while** (b < head->hth\_table\_length)
346. {
347. //找到链。因为是哈希。所以不一定哈希表中的每一个节点都存有元素
348. **if** (head->hth\_table[b])
349. **return** &head->hth\_table[b];
350. ++b;
351. }
353. **return** NULL;
354. }
355. }


359. //功能同上一个函数。只是参数不同，另外本函数还会使得--hth\_n\_entries
360. //该函数主要是返回elm的下一个元素。并且哈希表的总元素个数减一。
361. //主调函数会负责释放\*elm指向的元素。无需在这里动手
362. //在evmap\_io\_clear函数中，会调用本函数。
363. **static** **inline** **struct** event\_map\_entry \*\*
364. event\_io\_map\_HT\_NEXT\_RMV(**struct** event\_io\_map \*head,
365. **struct** event\_map\_entry \*\*elm)
366. {
367. #ifdef HT\_CACHE\_HASH\_VALUES
368. unsigned h = ((\*elm)->map\_node.hte\_hash);
369. #else
370. unsigned h = (hashsocket(\*elm));
371. #endif
373. //elm变量变成存放下一个元素的hte\_next的地址
374. \*elm = (\*elm)->map\_node.hte\_next;
376. --head->hth\_n\_entries;
378. **if** (\*elm)
379. {
380. **return** elm;
381. }
382. **else**
383. {
384. unsigned b = (h % head->hth\_table\_length)+1;
386. **while** (b < head->hth\_table\_length)
387. {
388. **if** (head->hth\_table[b])
389. **return** &head->hth\_table[b];
391. ++b;
392. }
394. **return** NULL;
395. }
396. }


400. //素数表。之所以去素数，是因为在取模的时候，素数比合数更有优势。
401. //听说是更散，更均匀
402. **static** unsigned event\_io\_map\_PRIMES[] =
403. {
404. //素数表的元素具有差不多2倍的关系。
405. //这使得扩容操作不会经常发生。每次扩容都预留比较大的空间
406. 53, 97, 193, 389, 769, 1543, 3079,
407. 6151, 12289, 24593, 49157, 98317,
408. 196613, 393241, 786433, 1572869, 3145739,
409. 6291469, 12582917, 25165843, 50331653, 100663319,
410. 201326611, 402653189, 805306457, 1610612741
411. };

414. //素数表中，元素的个数。
415. **static** unsigned event\_io\_map\_N\_PRIMES =
416. (unsigned)(**sizeof**(event\_io\_map\_PRIMES)
417. /**sizeof**(event\_io\_map\_PRIMES[0]));

420. /\* Expand the internal table of 'head' until it is large enough to
421. \* hold 'size' elements.  Return 0 on success, -1 on allocation
422. \* failure. \*/
423. **int** event\_io\_map\_HT\_GROW(**struct** event\_io\_map \*head, unsigned size)
424. {
425. unsigned new\_len, new\_load\_limit;
426. **int** prime\_idx;
428. **struct** event\_map\_entry \*\*new\_table;
429. //已经用到了素数表中最后一个素数，不能再扩容了。
430. **if** (head->hth\_prime\_idx == (**int**)event\_io\_map\_N\_PRIMES - 1)
431. **return** 0;
433. //哈希表中还够容量，无需扩容
434. **if** (head->hth\_load\_limit > size)
435. **return** 0;
437. prime\_idx = head->hth\_prime\_idx;
439. **do** {
440. new\_len = event\_io\_map\_PRIMES[++prime\_idx];
442. //从素数表中的数值可以看到，后一个差不多是前一个的2倍。
443. //从0.5和后的new\_load\_limit <= size，可以得知此次扩容
444. //至少得是所需大小(size)的2倍以上。免得经常要进行扩容
445. new\_load\_limit = (unsigned)(0.5\*new\_len);
446. } **while** (new\_load\_limit <= size
447. && prime\_idx < (**int**)event\_io\_map\_N\_PRIMES);
449. **if** ((new\_table = mm\_malloc(new\_len\***sizeof**(**struct** event\_map\_entry\*))))
450. {
451. unsigned b;
452. memset(new\_table, 0, new\_len\***sizeof**(**struct** event\_map\_entry\*));
454. **for** (b = 0; b < head->hth\_table\_length; ++b)
455. {
456. **struct** event\_map\_entry \*elm, \*next;
457. unsigned b2;
459. elm = head->hth\_table[b];
460. **while** (elm) //该节点有冲突链。遍历冲突链中所有的元素
461. {
462. next = elm->map\_node.hte\_next;
464. //冲突链中的元素，相对于前一个素数同余(即模素数后，结果相当)
465. //但对于现在的新素数就不一定同余了，即它们不一定还会冲突
466. //所以要对冲突链中的所有元素都再次哈希，并放到它们应该在的地方
467. //b2存放了再次哈希后，元素应该存放的节点下标。
468. #ifdef HT\_CACHE\_HASH\_VALUES
469. b2 = (elm)->map\_node.hte\_hash % new\_len;
470. #else
471. b2 = (hashsocket(\*elm)) % new\_len;
472. #endif
473. //用头插法插入数据
474. elm->map\_node.hte\_next = new\_table[b2];
475. new\_table[b2] = elm;
477. elm = next;
478. }
479. }
481. **if** (head->hth\_table)
482. mm\_free(head->hth\_table);
484. head->hth\_table = new\_table;
485. }
486. **else**
487. {
488. unsigned b, b2;
490. //刚才mm\_malloc失败，可能是内存不够。现在用更省内存的
491. //mm\_realloc方式。当然其代价就是更耗时(下面的代码可以看到)。
492. //前面的mm\_malloc会同时有hth\_table和new\_table两个数组。
493. //而mm\_realloc则只有一个数组，所以省内存，省了一个hth\_table数组
494. //的内存。此时，new\_table数组的前head->hth\_table\_length个
495. //元素存放了原来的冲突链的头部。也正是这个原因导致后面代码更耗时。
496. //其实，只有在很特殊的情况下，这个函数才会比mm\_malloc省内存.
497. //就是堆内存head->hth\_table区域的后面刚好有一段可以用的内存。
498. //具体的，可以搜一下realloc这个函数。
499. new\_table = mm\_realloc(head->hth\_table,
500. new\_len\***sizeof**(**struct** event\_map\_entry\*));
502. **if** (!new\_table)
503. **return** -1;
505. memset(new\_table + head->hth\_table\_length, 0,
506. (new\_len - head->hth\_table\_length)\***sizeof**(**struct** event\_map\_entry\*)
507. );
509. **for** (b=0; b < head->hth\_table\_length; ++b)
510. {
511. **struct** event\_map\_entry \*e, \*\*pE;
513. **for** (pE = &new\_table[b], e = \*pE; e != NULL; e = \*pE)
514. {
516. #ifdef HT\_CACHE\_HASH\_VALUES
517. b2 = (e)->map\_node.hte\_hash % new\_len;
518. #else
519. b2 = (hashsocket(\*elm)) % new\_len;
520. #endif
521. //对于冲突链A->B->C.
522. //pE是二级指针，存放的是A元素的hte\_next指针的地址值
523. //e指向B元素。
525. //对新的素数进行哈希，刚好又在原来的位置
526. **if** (b2 == b)
527. {
528. //此时，无需修改。接着处理冲突链中的下一个元素即可
529. //pE向前移动，存放B元素的hte\_next指针的地址值
530. pE = &e->map\_node.hte\_next;
531. }
532. **else**//这个元素会去到其他位置上。
533. {
534. //此时冲突链修改成A->C。
535. //所以pE无需修改，还是存放A元素的hte\_next指针的地址值
536. //但A元素的hte\_next指针要指向C元素。用\*pE去修改即可
537. \*pE = e->map\_node.hte\_next;
539. //将这个元素放到正确的位置上。
540. e->map\_node.hte\_next = new\_table[b2];
541. new\_table[b2] = e;
542. }
544. //这种再次哈希的方式，很有可能会对某些元素操作两次。
545. //当某个元素第一次在else中处理，那么它就会被哈希到正确的节点
546. //的冲突链上。随着外循环的进行，处理到正确的节点时。在遍历该节点
547. //的冲突链时，又会再次处理该元素。此时，就会在if中处理。而不会
548. //进入到else中。
549. }
550. }
552. head->hth\_table = new\_table;
553. }

556. //一般是当hth\_n\_entries >= hth\_load\_limit时，就会调用
557. //本函数。hth\_n\_entries表示的是哈希表中节点的个数。而hth\_load\_limit
558. //是hth\_table\_length的一半。hth\_table\_length则是哈希表中
559. //哈希函数被模的数字。所以，当哈希表中的节点个数到达哈希表长度的一半时
560. //就会发生增长，本函数被调用。这样的结果是：平均来说，哈希表应该比较少发生
561. //冲突。即使有，冲突链也不会太长。这样就能有比较快的查找速度。
562. head->hth\_table\_length = new\_len;
563. head->hth\_prime\_idx = prime\_idx;
564. head->hth\_load\_limit = new\_load\_limit;
566. **return** 0;
567. }

570. /\* Free all storage held by 'head'.  Does not free 'head' itself,
571. \* or individual elements. 并不需要释放独立的元素\*/
572. //在evmap\_io\_clear函数会调用该函数。其是在删除所有哈希表中的元素后
573. //才调用该函数的。
574. **void** event\_io\_map\_HT\_CLEAR(**struct** event\_io\_map \*head)
575. {
576. **if** (head->hth\_table)
577. mm\_free(head->hth\_table);
579. head->hth\_table\_length = 0;
581. event\_io\_map\_HT\_INIT(head);
582. }

585. /\* Debugging helper: return false iff the representation of 'head' is
586. \* internally consistent. \*/
587. **int** \_event\_io\_map\_HT\_REP\_IS\_BAD(**const** **struct** event\_io\_map \*head)
588. {
589. unsigned n, i;
590. **struct** event\_map\_entry \*elm;
592. **if** (!head->hth\_table\_length)
593. {
594. //刚被初始化，还没申请任何空间
595. **if** (!head->hth\_table && !head->hth\_n\_entries
596. && !head->hth\_load\_limit && head->hth\_prime\_idx == -1
597. )
598. **return** 0;
599. **else**
600. **return** 1;
601. }
603. **if** (!head->hth\_table || head->hth\_prime\_idx < 0
604. || !head->hth\_load\_limit
605. )
606. **return** 2;
608. **if** (head->hth\_n\_entries > head->hth\_load\_limit)
609. **return** 3;
611. **if** (head->hth\_table\_length != event\_io\_map\_PRIMES[head->hth\_prime\_idx])
612. **return** 4;
614. **if** (head->hth\_load\_limit != (unsigned)(0.5\*head->hth\_table\_length))
615. **return** 5;
617. **for** (n = i = 0; i < head->hth\_table\_length; ++i)
618. {
619. **for** (elm = head->hth\_table[i]; elm; elm = elm->map\_node.hte\_next)
620. {
622. #ifdef HT\_CACHE\_HASH\_VALUES
624. **if** (elm->map\_node.hte\_hash != hashsocket(elm))
625. **return** 1000 + i;
627. **if**( (elm->map\_node.hte\_hash % head->hth\_table\_length) != i)
628. **return** 10000 + i;
630. #else
631. **if** ( (hashsocket(\*elm)) != hashsocket(elm))
632. **return** 1000 + i;
634. **if**( ( (hashsocket(\*elm)) % head->hth\_table\_length) != i)
635. **return** 10000 + i;
636. #endif
637. ++n;
638. }
639. }
641. **if** (n != head->hth\_n\_entries)
642. **return** 6;
644. **return** 0;
645. }

        代码中的注释已经对这个哈希表的一些特征进行了描述，这里就不多说了。

# 哈希表在Libevent的使用：

        现在来讲一下event\_io\_map的应用。

        在event\_base这个结构体中有一个event\_io\_map类型的成员变量io。它就是一个哈希表。当一个监听读或者写操作的event，调用event\_add函数插入到event\_base中时，就会调用evmap\_io\_add函数。evmap\_io\_add函数应用到这个event\_io\_map结构体。该函数的定义如下，其中使用到了一个宏定义，我已经展开了。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38403241) [copy](https://blog.csdn.net/luotuo44/article/details/38403241)

1. **int**
2. evmap\_io\_add(**struct** event\_base \*base, evutil\_socket\_t fd, **struct** event \*ev)
3. {
4. **const** **struct** eventop \*evsel = base->evsel;
5. **struct** event\_io\_map \*io = &base->io;
6. **struct** evmap\_io \*ctx = NULL;
7. **int** nread, nwrite, retval = 0;
8. **short** res = 0, old = 0;
9. **struct** event \*old\_ev;
11. EVUTIL\_ASSERT(fd == ev->ev\_fd);
13. **if** (fd < 0)
14. **return** 0;
16. //GET\_IO\_SLOT\_AND\_CTOR(ctx, io, fd, evmap\_io, evmap\_io\_init,
17. //                   evsel->fdinfo\_len);SLOT指的是fd
18. //GET\_IO\_SLOT\_AND\_CTOR宏将展开成下面这个do{}while(0);
19. **do**
20. {
21. **struct** event\_map\_entry \_key, \*\_ent;
22. \_key.fd = fd;
24. **struct** event\_io\_map \*\_ptr\_head = io;
25. **struct** event\_map\_entry \*\*ptr;
27. //哈希表扩容，减少冲突的可能性
28. **if** (!\_ptr\_head->hth\_table
29. || \_ptr\_head->hth\_n\_entries >= \_ptr\_head->hth\_load\_limit)
30. {
31. event\_io\_map\_HT\_GROW(\_ptr\_head,
32. \_ptr\_head->hth\_n\_entries + 1);
33. }
35. #ifdef HT\_CACHE\_HASH\_VALUES
36. **do**{
37. (&\_key)->map\_node.hte\_hash = hashsocket((&\_key));
38. } **while**(0);
39. #endif
41. //返回值ptr,是要查找节点的前驱节点的hte\_next成员变量的地址.
42. //所以返回值肯定不会为NULL,而\*ptr就可能为NULL。说明hte\_next
43. //不指向任何节点。也正由于这个原因，所以即使\*ptr 为NULL,但是可以
44. //给\*ptr赋值。此时，是修改前驱节点的hte\_next成员变量的值，使之
45. //指向另外一个节点。
46. //这里调用\_event\_io\_map\_HT\_FIND\_P原因有二：1.查看该fd是否已经
47. //插入过这个哈希表中。2.得到这个fd计算哈希位置。
48. ptr = \_event\_io\_map\_HT\_FIND\_P(\_ptr\_head, (&\_key));
50. //在event\_io\_map这个哈希表中查找是否已经存在该fd的event\_map\_entry了
51. //因为同一个fd可以调用event\_new多次，然后event\_add多次的。
52. **if** (\*ptr)
53. {
54. \_ent = \*ptr;
55. }
56. **else**
57. {
58. \_ent = mm\_calloc(1, **sizeof**(**struct** event\_map\_entry) + evsel->fdinfo\_len);
59. **if** (EVUTIL\_UNLIKELY(\_ent == NULL))
60. **return** (-1);
62. \_ent->fd = fd;
63. //调用初始化函数初始化这个evmap\_io
64. (evmap\_io\_init)(&\_ent->ent.evmap\_io);
66. #ifdef HT\_CACHE\_HASH\_VALUES
67. **do**
68. {
69. ent->map\_node.hte\_hash = (&\_key)->map\_node.hte\_hash;
70. }**while**(0);
71. #endif
72. \_ent->map\_node.hte\_next = NULL;
74. //把这个新建的节点插入到哈希表中。ptr已经包含了哈希位置
75. \*ptr = \_ent;
76. ++(io->hth\_n\_entries);
77. }

80. //这里是获取该event\_map\_entry的next和prev指针。因为
81. //evmap\_io含有next、prev变量。这样在之后就可以把这个
82. //event\_map\_entry连起来。这个外do{}while(0)的功能是
83. //为这个fd分配一个event\_map\_entry,并且插入到现有的哈希
84. //表中。同时，这个fd还是结构体event的一部分。而event必须
85. //插入到event队列中。
86. (ctx) = &\_ent->ent.evmap\_io;
88. } **while** (0);

91. ....
93. //ctx->events是一个TAILQ\_HEAD。结合之前讲到的TAILQ\_QUEUE队列，
94. //就可以知道：同一个fd，可能有多个event结构体。这里就把这些结构体连
95. //起来。依靠的链表是，event结构体中的ev\_io\_next。ev\_io\_next是
96. //一个TAILQ\_ENTRY,具有前驱和后驱指针。队列头部为event\_map\_entry
97. //结构体中的evmap\_io成员的events成员。
98. TAILQ\_INSERT\_TAIL(&ctx->events, ev, ev\_io\_next);
100. **return** (retval);
101. }

        GET\_IO\_SLOT\_AND\_CTOR宏的作用就是让ctx指向struct event\_map\_entry结构体中的TAILQ\_HEAD。这样就可以使用TAILQ\_INSERT\_TAIL宏，把ev变量插入到队列中。如果有现成的event\_map\_entry就直接使用，没有的话就新建一个。

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/38403241