<https://blog.csdn.net/luotuo44/article/details/38678333>

**转载请注明出处:**[**http://blog.csdn.net/luotuo44/article/details/38678333**](http://blog.csdn.net/luotuo44/article/details/38678333)

        前面的博文已经说到，如果要对多个超时event同时进行监听，就要对这些超时event进行集中管理，能够方便地(时间复杂度小)获取、加入、删除一个event。

        在之前的Libevent版本，Libevent使用小根堆管理这些超时event。小根堆的插入和删除时间复杂度都是O(logN)。在2.0.4-alpha版本时，Libevent引入了一个叫common-timeout的东西来管理超时event，要注意的是，它并不是替代小根堆，而是和小根堆配合使用的。事实上，common-timeout的实现要用到小根堆。

        Libevent的小根堆和数据结构教科书上的小根堆几乎是一样的。看一下数据结构和Libevent的小根堆源码，很容易就懂的。这样就不多讲了。

        本文主要讲一下common-timeout。从common的字面意思和它的实际使用来说，可以把它翻译成“公用超时”。

# common-timeout的用途：

        要讲解common-timeout，得先说明它的用途。前面说到它和小根堆是配合使用的。小根堆是用在：多个超时event的超时时长是随机的。而common-timeout则是用在：大量的超时event具有相同的超时时长。其中，超时时长是指event\_add参数的第二个参数。要注意的是，这些大量超时event虽然有相同的超时时长，但它们的超时时间是不同的。因为超时时间 = 超时时长+ 调用event\_add时间。

        毫无疑问，如果有相同超时时长的大量超时event都放到小根堆上，那么效率比较低的。虽然小根堆的插入和删除的时间复杂度都是O(logN)，但是如果有大量的N，效率也是会下降很多。

# common-timeout的原理：

        common-timeout的思想是，既然有大量的超时event具有相同的超时时长，那么就它们必定依次激活。如果把它们按照超时时间升序地放到一个队列中(在Libevent中就是这样做的)，那么每次只需检查队列的第一个超时event即可。因为其他超时event肯定在第一个超时之后才超时的。

        前面说到common-timeout和小根堆是配合使用的。从common-timeout中选出最早超时的那个event，将之插入到小根堆中。然后通过小根堆对这个event进行超时监控。超时后再从common-timeout中选出下一个最早超时的event。具体的超时监控处理过程可以参考《[超时event的处理](http://blog.csdn.net/luotuo44/article/details/38637671)》一文。通过这样处理后，就不用把大量的超时event都插入到小根堆中。

        下面看一下Libevent的具体实现吧。

# 相关结构体：

        首先看一下event\_base为common-timeout提供了什么成员变量。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38678333) [copy](https://blog.csdn.net/luotuo44/article/details/38678333)

1. //event-internal.h文件
2. **struct** event\_base {
3. //因为可以有多个不同时长的超时event组。故得是数组
4. //因为数组元素是common\_timeout\_list指针，所以得是二级指针
5. **struct** common\_timeout\_list \*\*common\_timeout\_queues;
6. //数组元素个数
7. **int** n\_common\_timeouts;
8. //已分配的数组元素个数
9. **int** n\_common\_timeouts\_allocated;
10. };
12. **struct** common\_timeout\_list {
13. //超时event队列。将所有具有相同超时时长的超时event放到一个队列里面
14. **struct** event\_list events;
16. **struct** timeval duration;//超时时长
17. **struct** event timeout\_event;//具有相同超时时长的超时event代表
18. **struct** event\_base \*base;
19. };

        在实际应用时，可能超时时长为10秒的有1k个超时event，时长为20秒的也有1k个，这就需要一个数组。数组的每一个元素是common\_timeout\_list结构体指针。每一个common\_timeout\_list结构体就会处理所有具有相同超时时长的超时event。

        common\_timeout\_list结构体里面有一个event结构体成员，所以并不是从多个具有相同超时时长的超时event中选择一个作为代表，而是在内部有一个event。

        common\_timeout\_list是使用struct  event\_list结构体队列来管理event，它是一种TAILQ\_QUEUE队列，可以参考博文《[TAILQ\_QUEUE队列](http://blog.csdn.net/luotuo44/article/details/38374009)》。

# 使用common-timeout：

        现在来看看怎么使用common-timeout。从上面的代码可以想到，如果要使用common-timeout，就必须把超时event插入到common\_timeout\_list的events队列中。又因为其要求具有相同的超时时长，所以要插入的超时event要和某个common\_timeout\_list结构体有相同的超时时长。所以，我们还是来看一下怎么设置common\_timeout\_list结构体的超时时长。

        实际上，并不是设置。而是向event\_base申请一个具有特定时长的common\_timeout\_list。每申请一个，就会在common\_timeout\_queues数组中加入一个common\_timeout\_list元素。可以通过event\_base\_init\_common\_timeout申请。申请后，就可以直接调用event\_add把超时event插入到common-timeout中。但问题是，common-timeout和小根堆是共存的，event\_add又没有第三个参数作为说明，要插入到common-timeout还是小根堆。

## common-timeout标志：

        其实，event\_add是根据第二个参数，即超时时长值进行区分的。

        首先有一个基本事实，对一个struct timeval结构体,成员tv\_usec的单位是微秒，所以最大也就是999999,只需低20比特位就能存储了。但成员tv\_usec的类型是int或者long，肯定有32比特位。所以，就有高12比特位是空闲的。

        Libevent就是利用那空闲的12个比特位做文章的。这12比特位是高比特位。Libevent使用最高的4比特位作为标志位，标志它是一个专门用于common-timeout的时间，下文将这个标志称为common-timeout标志。次8比特位用来记录该超时时长在common\_timeout\_queues数组中的位置，即下标值。这也限制了common\_timeout\_queues数组的长度，最大为2的8次方，即256。

        为了方便地处理这些比特位，Libevent定义了下面这些宏定义和一个判断函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38678333) [copy](https://blog.csdn.net/luotuo44/article/details/38678333)

1. //event.c文件
2. #define COMMON\_TIMEOUT\_MICROSECONDS\_MASK       0x000fffff
3. #define MICROSECONDS\_MASK       COMMON\_TIMEOUT\_MICROSECONDS\_MASK
4. #define COMMON\_TIMEOUT\_IDX\_MASK 0x0ff00000
5. #define COMMON\_TIMEOUT\_IDX\_SHIFT 20
6. #define COMMON\_TIMEOUT\_MASK     0xf0000000
7. #define COMMON\_TIMEOUT\_MAGIC    0x50000000
9. #define COMMON\_TIMEOUT\_IDX(tv) \
10. (((tv)->tv\_usec & COMMON\_TIMEOUT\_IDX\_MASK)>>COMMON\_TIMEOUT\_IDX\_SHIFT)
12. #define MAX\_COMMON\_TIMEOUTS 256
14. **static** **inline** **int**
15. is\_common\_timeout(**const** **struct** timeval \*tv,
16. **const** **struct** event\_base \*base)
17. {
18. **int** idx;
19. //不具有common-timeout标志位，那么就肯定不是commont-timeout时间了
20. **if** ((tv->tv\_usec & COMMON\_TIMEOUT\_MASK) != COMMON\_TIMEOUT\_MAGIC)
21. **return** 0;
23. idx = COMMON\_TIMEOUT\_IDX(tv);//获取数组下标
24. **return** idx < base->n\_common\_timeouts;
25. }

        代码最后面的那个判断函数，是用来判断一个给定的struct timeval时间，是否为common-timeout时间。在event\_add\_internal函数中会用之作为判断，然后根据判断结果来决定是插入小根堆还是common-timeout，这也就完成了区分。

## 申请并得到特定时长的common-timeout：

        那么怎么得到一个具有common-timeout标志的时间呢？其实，还是通过前面说到的event\_base\_init\_common\_timeout函数。该函数将返回一个具有common-timeout标志的时间。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38678333) [copy](https://blog.csdn.net/luotuo44/article/details/38678333)

1. //event.c文件
2. //申请一个时长为duration的common\_timeout\_list
3. **const** **struct** timeval \*
4. event\_base\_init\_common\_timeout(**struct** event\_base \*base,
5. **const** **struct** timeval \*duration)
6. {
7. **int** i;
8. **struct** timeval tv;
9. **const** **struct** timeval \*result=NULL;
10. **struct** common\_timeout\_list \*new\_ctl;
12. //这个时间的微秒位应该进位。用户没有将之进位。比如二进制的103，个位的3应该进位
13. **if** (duration->tv\_usec > 1000000) {
14. //将之进位，因为下面会用到高位
15. memcpy(&tv, duration, **sizeof**(**struct** timeval));
16. **if** (is\_common\_timeout(duration, base))
17. tv.tv\_usec &= MICROSECONDS\_MASK;//去除common-timeout标志
18. tv.tv\_sec += tv.tv\_usec / 1000000; //进位
19. tv.tv\_usec %= 1000000;
20. duration = &tv;
21. }
23. **for** (i = 0; i < base->n\_common\_timeouts; ++i) {
24. **const** **struct** common\_timeout\_list \*ctl =
25. base->common\_timeout\_queues[i];
26. //具有相同的duration， 即之前有申请过这个超时时长。那么就不用分配空间。
27. **if** (duration->tv\_sec == ctl->duration.tv\_sec &&
28. duration->tv\_usec ==
29. (ctl->duration.tv\_usec & MICROSECONDS\_MASK)) {//要&这个宏，才能是正确的时间
30. result = &ctl->duration;
31. **goto** done;
32. }
33. }
35. //达到了最大申请个数，不能再分配了
36. **if** (base->n\_common\_timeouts == MAX\_COMMON\_TIMEOUTS) {
37. **goto** done;
38. }
40. //新的超时时长，需要分配一个common\_timeout\_list结构体。
42. //之前分配的空间已经用完了，要重新申请空间
43. **if** (base->n\_common\_timeouts\_allocated == base->n\_common\_timeouts) {
44. **int** n = base->n\_common\_timeouts < 16 ? 16 :
45. base->n\_common\_timeouts\*2;
46. **struct** common\_timeout\_list \*\*newqueues =
47. mm\_realloc(base->common\_timeout\_queues,
48. n\***sizeof**(**struct** common\_timeout\_queue \*));
49. **if** (!newqueues) {
50. **goto** done;
51. }
52. base->n\_common\_timeouts\_allocated = n;
53. base->common\_timeout\_queues = newqueues;
54. }
56. //为该超时时长分配一个common\_timeout\_list结构体
57. new\_ctl = mm\_calloc(1, **sizeof**(**struct** common\_timeout\_list));
58. **if** (!new\_ctl) {
59. **goto** done;
60. }
62. //为这个结构体进行一些设置
63. TAILQ\_INIT(&new\_ctl->events);
64. new\_ctl->duration.tv\_sec = duration->tv\_sec;
65. new\_ctl->duration.tv\_usec =
66. duration->tv\_usec | COMMON\_TIMEOUT\_MAGIC | //为这个时间加入common-timeout标志
67. (base->n\_common\_timeouts << COMMON\_TIMEOUT\_IDX\_SHIFT);//加入下标值
69. //对timeout\_event这个内部event进行赋值。设置回调函数和回调参数。
70. evtimer\_assign(&new\_ctl->timeout\_event, base,
71. common\_timeout\_callback, new\_ctl);
73. new\_ctl->timeout\_event.ev\_flags |= EVLIST\_INTERNAL; //标志成内部event
74. event\_priority\_set(&new\_ctl->timeout\_event, 0); //优先级为最高级
75. new\_ctl->base = base;
76. //放到数组对应的位置上
77. base->common\_timeout\_queues[base->n\_common\_timeouts++] = new\_ctl;
78. result = &new\_ctl->duration;
80. done:
82. **return** result;
83. }

        该函数只是在event\_base的common\_timeout\_queues数组中申请一个特定超时时长的位置。同时该函数也会返回一个struct timeval结构体指针变量，该结构体已经被赋予了common-timeout标志。以后使用该变量作为event\_add的第二个参数，就可以把超时event插入到common-timeout中了。不应该也不能自己手动为struct timeval变量加入common-timeout标志。

        该函数中，也给内部的event进行了赋值,设置了回调函数和回调参数。要注意的是回调参数是这个common\_timeout\_list结构体变量指针。在回调函数中，有了这个指针，就可以访问events变量，即访问到该结构体上的所有超时event。于是就能手动激活这些超时event。

        在Libevent的官方例子中，得到event\_base\_init\_common\_timeout的返回值后，就把它存放到另外一个struct timeval结构体中。而不是直接使用返回值作为event\_add的参数。

## 将超时event存放到common-timeout中：

        现在已经向event\_base申请了一个特定的超时时长，并得到了具有common-timeout标志的时间。那么，就调用event\_add看看。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38678333) [copy](https://blog.csdn.net/luotuo44/article/details/38678333)

1. //event.c文件
2. **static** **inline** **int**
3. event\_add\_internal(**struct** event \*ev, **const** **struct** timeval \*tv,
4. **int** tv\_is\_absolute)
5. {
6. **struct** event\_base \*base = ev->ev\_base;
7. **int** res = 0;
8. **int** notify = 0;
10. ...//加入到IO队列或者信号队列
12. **if** (res != -1 && tv != NULL) {
13. **struct** timeval now;
14. **int** common\_timeout;
16. gettime(base, &now);
18. //判断这个时间是否为common-timeout标志
19. common\_timeout = is\_common\_timeout(tv, base);
20. **if** (common\_timeout) {
21. **struct** timeval tmp = \*tv;
22. //只取真正的时间部分，common-timeout标志位和下标位不要
23. tmp.tv\_usec &= MICROSECONDS\_MASK;
24. //转换成绝对时间
25. evutil\_timeradd(&now, &tmp, &ev->ev\_timeout);
26. ev->ev\_timeout.tv\_usec |=
27. (tv->tv\_usec & ~MICROSECONDS\_MASK); //加入标志位
28. }
30. event\_queue\_insert(base, ev, EVLIST\_TIMEOUT);
32. **if** (common\_timeout) {
33. **struct** common\_timeout\_list \*ctl =
34. get\_common\_timeout\_list(base, &ev->ev\_timeout);
35. **if** (ev == TAILQ\_FIRST(&ctl->events)) {
36. common\_timeout\_schedule(ctl, &now, ev);
37. }
38. }
39. }
41. **return** (res);
42. }

        由于在《[超时event的处理](http://blog.csdn.net/luotuo44/article/details/38637671)》一文中已经对这个函数进行了一部分讲解，现在只讲有关common-timeout部分。

        虽然上面的代码省略了很多东西，但是有一点要说明，当超时event被加入common-timeout时并不会设置notify变量的，即不需要通知主线程。

# common-timeout与小根堆的配合：

        从上面的代码可以看到，首先是为超时event内部时间ev\_timeout加入common-timeout标志。然后调用event\_queue\_insert进行插入。但此时调用event\_queue\_insert插入，并不是插入到小根堆。它只是插入到event\_base的common\_timeout\_list数组的一个队列中。下面代码可以看到这一点。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38678333) [copy](https://blog.csdn.net/luotuo44/article/details/38678333)

1. //event.c文件
2. **static** **void**
3. event\_queue\_insert(**struct** event\_base \*base, **struct** event \*ev, **int** queue)
4. {
6. ev->ev\_flags |= queue;
7. **switch** (queue) {
8. **case** EVLIST\_TIMEOUT: {
9. **if** (is\_common\_timeout(&ev->ev\_timeout, base)) {
10. //根据时间向event\_base获取对应的common\_timeout\_list
11. **struct** common\_timeout\_list \*ctl =
12. get\_common\_timeout\_list(base, &ev->ev\_timeout);
13. insert\_common\_timeout\_inorder(ctl, ev);
14. }
15. **break**;
16. }
18. }
19. }
21. **static** **void** //in order说明是有序的。
22. insert\_common\_timeout\_inorder(**struct** common\_timeout\_list \*ctl,
23. **struct** event \*ev)
24. {
25. **struct** event \*e;
26. //虽然有相同超时时长，但超时时间却是 超时时长 + 调用event\_add的时间。
27. //所以是在不同的时间触发超时的。它们根据绝对超时时间，升序排在队列中。
28. //一般来说，直接插入队尾即可。因为后插入的，绝对超时时间肯定大。
29. //但由于线程抢占的原因，可能一个线程在evutil\_timeradd(&now, &tmp, &ev->ev\_timeout);
30. //执行完，还没来得及插入，就被另外一个线程抢占了。而这个线程也是要插入一个
31. //common-timeout的超时event。这样就会发生：超时时间小的反而后插入。
32. //所以要从后面开始遍历队列，寻找一个合适的地方。
33. TAILQ\_FOREACH\_REVERSE(e, &ctl->events,
34. event\_list, ev\_timeout\_pos.ev\_next\_with\_common\_timeout) {
35. **if** (evutil\_timercmp(&ev->ev\_timeout, &e->ev\_timeout, >=)) {
36. TAILQ\_INSERT\_AFTER(&ctl->events, e, ev, //从队列后面插入
37. ev\_timeout\_pos.ev\_next\_with\_common\_timeout);
38. **return**; //插入后就返回
39. }
40. }
42. //在队列头插入，只会发生在前面的寻找都没有寻找到的情况下
43. TAILQ\_INSERT\_HEAD(&ctl->events, ev,
44. ev\_timeout\_pos.ev\_next\_with\_common\_timeout);
45. }

        既然event\_queue\_insert函数并没有完成插入到小根堆。那么就看event\_add\_internal的最后面的那个if判断。读者可能会问，为什么要插入到小根堆。其实，前面已经说到了。common-timeout是采用一个代表的方式进行工作的。所以肯定要有一个代表被插入小根堆中，这也是common-timeout和小根堆的相互配合。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38678333) [copy](https://blog.csdn.net/luotuo44/article/details/38678333)

1. //event.c文件
2. **if** (common\_timeout) {
3. **struct** common\_timeout\_list \*ctl =
4. get\_common\_timeout\_list(base, &ev->ev\_timeout);
5. **if** (ev == TAILQ\_FIRST(&ctl->events)) {
6. common\_timeout\_schedule(ctl, &now, ev);
7. }
8. }
10. **static** **void**
11. common\_timeout\_schedule(**struct** common\_timeout\_list \*ctl,
12. **const** **struct** timeval \*now, **struct** event \*head)
13. {
14. **struct** timeval timeout = head->ev\_timeout;
15. timeout.tv\_usec &= MICROSECONDS\_MASK; //清除common-timeout标志
16. //用common\_timeout\_list结构体的一个event成员作为超时event调用event\_add\_internal
17. //由于已经清除了common-timeout标志，所以这次将插入到小根堆中。
18. event\_add\_internal(&ctl->timeout\_event, &timeout, 1);
19. }

        从判断可以看到，它判断要插入的这个超时event是否为这个队列的第一个元素。如果是的话，就说这个特定超时时长队列第一次有超时event要插入。这就要进行一些处理。

        在common\_timeout\_schedule函数中，我们可以看到，它将一个event插入到小根堆中了。并且也可以看到，代表者不是用户给出的超时event中的一个，而是common\_timeout\_list结构体的一个event成员。

# 将common-timeout event激活：

        现在来看一下当common\_timeout\_list的内部event成员被激活时怎么处理。它的回调函数为common\_timeout\_callback。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38678333) [copy](https://blog.csdn.net/luotuo44/article/details/38678333)

1. //event.c文件
2. **static** **void**
3. common\_timeout\_callback(evutil\_socket\_t fd, **short** what, **void** \*arg)
4. {
5. **struct** timeval now;
6. **struct** common\_timeout\_list \*ctl = arg;
7. **struct** event\_base \*base = ctl->base;
8. **struct** event \*ev = NULL;
9. EVBASE\_ACQUIRE\_LOCK(base, th\_base\_lock);
10. gettime(base, &now);
11. **while** (1) {
12. ev = TAILQ\_FIRST(&ctl->events);
14. //该超时event还没到超时时间。不要检查其他了。因为是升序的
15. **if** (!ev || ev->ev\_timeout.tv\_sec > now.tv\_sec ||
16. (ev->ev\_timeout.tv\_sec == now.tv\_sec &&
17. (ev->ev\_timeout.tv\_usec&MICROSECONDS\_MASK) > now.tv\_usec))
18. **break**;
20. //一系列的删除操作。包括从这个超时event队列中删除
21. event\_del\_internal(ev);
22. //手动激活超时event。注意，这个ev是用户的超时event
23. event\_active\_nolock(ev, EV\_TIMEOUT, 1);
24. }
26. //不是NULL，说明该队列还有超时event。那么需要再次common\_timeout\_schedule，进行监听
27. **if** (ev)
28. common\_timeout\_schedule(ctl, &now, ev);
29. EVBASE\_RELEASE\_LOCK(base, th\_base\_lock);
30. }

        在回调函数中，会手动把用户的超时event激活。于是，用户的超时event就能被处理了。

        由于Libevent这个内部超时event的优先级是最高的，所以在接下来就会处理用户的超时event，而无需等到下一轮多路IO复用函数调用返回后。这一点同信号event是一样的，在《[信号event的处理](http://blog.csdn.net/luotuo44/article/details/38538991)》博文的最后有一些论证。

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/38678333