<https://blog.csdn.net/luotuo44/article/details/38637671>

**转载请注明出处:**[**http://blog.csdn.net/luotuo44/article/details/38637671**](http://blog.csdn.net/luotuo44/article/details/38637671)

# 如何成为超时event：

         Libevent允许创建一个超时event，使用evtimer\_new宏。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38637671) [copy](https://blog.csdn.net/luotuo44/article/details/38637671)

1. //event.h文件
2. #define evtimer\_new(b, cb, arg)        event\_new((b), -1, 0, (cb), (arg))

        从宏的实现来看，它一样是用到了一般的event\_new，并且不使用任何的文件描述符。从超时event宏的实现来看，无论是evtimer创建的event还是一般event\_new创建的event，都能使得Libevent进行超时监听。其实，使得Libevent对一个event进行超时监听的原因是：在调用event\_add的时候，第二参数不能为NULL，要设置一个超时值。如果为NULL，那么Libevent将不会为这个event监听超时。下文统一称设置了超时值的event为超时event。

# 超时event的原理：

        Libevent对超时进行监听的原理不同于之前讲到的对信号的监听，Libevent对超时的监听的原理是，多路IO复用函数都是有一个超时值。如果用户需要Libevent同时监听多个超时event，那么Libevent就把超时值最小的那个作为多路IO复用函数的超时值。自然，当时间一到，就会从多路IO复用函数返回。此时对超时event进行处理即可。

        Libevent运行用户同时监听多个超时event，那么就必须要对这个超时值进行管理。Libevent提供了小根堆和通用超时(common timeout)这两种管理方式。下文为了叙述方便，就假定使用的是小根堆。

# 工作流程：

        下面来看一下超时event的工作流程。

## 设置超时值：

        首先调用event\_add时要设置一个超时值，这样才能成为一个超时event。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38637671) [copy](https://blog.csdn.net/luotuo44/article/details/38637671)

1. //event.c文件
2. //在event\_add中，会把第三个参数设为0.使得使用的是相对时间
3. **static** **inline** **int**
4. event\_add\_internal(**struct** event \*ev, **const** **struct** timeval \*tv,
5. **int** tv\_is\_absolute)
6. {
7. **struct** event\_base \*base = ev->ev\_base;
8. **int** res = 0;
9. **int** notify = 0;
11. //tv不为NULL,就说明是一个超时event,在小根堆中为其留一个位置
12. **if** (tv != NULL && !(ev->ev\_flags & EVLIST\_TIMEOUT)) {
13. **if** (min\_heap\_reserve(&base->timeheap,
14. 1 + min\_heap\_size(&base->timeheap)) == -1)
15. **return** (-1);  /\* ENOMEM == errno \*/
16. }
18. ...//将IO或者信号event插入到对应的队列中。

21. **if** (res != -1 && tv != NULL) {
22. **struct** timeval now;
24. //用户把这个event设置成EV\_PERSIST。即永久event.
25. //如果没有这样设置的话，那么只会超时一次。设置了，那么就
26. //可以超时多次。那么就要记录用户设置的超时值。
27. **if** (ev->ev\_closure == EV\_CLOSURE\_PERSIST && !tv\_is\_absolute)
28. ev->ev\_io\_timeout = \*tv;
30. //该event之前被加入到超时队列。用户可以对同一个event调用多次event\_add
31. //并且可以每次都用不同的超时值。
32. **if** (ev->ev\_flags & EVLIST\_TIMEOUT) {
33. /\* XXX I believe this is needless. \*/
34. //之前为该event设置的超时值是所有超时中最小的。
35. //从下面的删除可知，会删除这个最小的超时值。此时多路IO复用函数
36. //的超时值参数就已经改变了。
37. **if** (min\_heap\_elt\_is\_top(ev))
38. notify = 1; //要通知主线程。可能是次线程为这个event调用本函数
40. //从超时队列中删除这个event。因为下次会再次加入。
41. //多次对同一个超时event调用event\_add,那么只能保留最后的那个。
42. event\_queue\_remove(base, ev, EVLIST\_TIMEOUT);
43. }
45. //因为可以在次线程调用event\_add。而主线程刚好在执行event\_base\_dispatch
46. **if** ((ev->ev\_flags & EVLIST\_ACTIVE) &&
47. (ev->ev\_res & EV\_TIMEOUT)) {//该event被激活的原因是超时
49. ...
50. event\_queue\_remove(base, ev, EVLIST\_ACTIVE);
51. }
53. //获取现在的时间
54. gettime(base, &now);
56. //虽然用户在event\_add时只需用一个相对时间，但实际上在Libevent内部
57. //还是要把这个时间转换成绝对时间。从存储的角度来说，存绝对时间只需
58. //一个变量。而相对时间则需两个，一个存相对值，另一个存参照物。
59. **if** (tv\_is\_absolute) { //该参数指明时间是否为一个绝对时间
60. ev->ev\_timeout = \*tv;
61. } **else** {
62. //参照时间 + 相对时间  ev\_timeout存的是绝对时间
63. evutil\_timeradd(&now, tv, &ev->ev\_timeout);
64. }

67. //将该超时event插入到超时队列中
68. event\_queue\_insert(base, ev, EVLIST\_TIMEOUT);
70. //本次插入的超时值，是所有超时中最小的。那么此时就需要通知主线程。
71. **if** (min\_heap\_elt\_is\_top(ev))
72. notify = 1;
73. }
75. //如果代码逻辑中是需要通知的。并且本线程不是主线程。那么就通知主线程
76. **if** (res != -1 && notify && EVBASE\_NEED\_NOTIFY(base))
77. evthread\_notify\_base(base);
79. **return** (res);
80. }

        对于同一个event，如果是IO event或者信号event，那么将无法多次添加。但如果是一个超时event，那么是可以多次添加的。并且对应超时值会使用最后添加时指明的那个，之前的统统不要，即替换掉之前的超时值。

        代码中出现了多次使用了notify变量。这主要是用在：次线程在执行这个函数，而主线程在执行event\_base\_dispatch。前面说到Libevent能对超时event进行监听的原理是：多路IO复用函数有一个超时参数。在次线程添加的event的超时值更小，又或者替换了之前最小的超时值。在这种情况下，都是要通知主线程，告诉主线程，最小超时值已经变了。关于通知主线程evthread\_notify\_base，可以参考博文《[evthread\_notify\_base通知主线程](http://blog.csdn.net/luotuo44/article/details/38556059)》。

        代码中的第三个判断体中用到了ev->ev\_io\_timeout。但event结构体中并没有该变量。其实，ev\_io\_timeout是一个宏定义。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38637671) [copy](https://blog.csdn.net/luotuo44/article/details/38637671)

1. //event-internal.h文件
2. #define ev\_io\_timeout   \_ev.ev\_io.ev\_timeout

        要注意的一点是，在调用event\_add时设定的超时值是一个时间段(可以认为隔多长时间就触发一次)，相对于现在，即调用event\_add的时间，而不是调用event\_base\_dispatch的时间。

## 调用多路IO复用函数等待超时：

        现在来看一下event\_base\_loop函数，看其是怎么处理超时event的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38637671) [copy](https://blog.csdn.net/luotuo44/article/details/38637671)

1. //event.c文件
2. **int**
3. event\_base\_loop(**struct** event\_base \*base, **int** flags)
4. {
5. **const** **struct** eventop \*evsel = base->evsel;
6. **struct** timeval tv;
7. **struct** timeval \*tv\_p;
8. **int** res, done, retval = 0;
10. EVBASE\_ACQUIRE\_LOCK(base, th\_base\_lock);
12. base->running\_loop = 1;
14. done = 0;
15. **while** (!done) {
16. tv\_p = &tv;
17. **if** (!N\_ACTIVE\_CALLBACKS(base) && !(flags & EVLOOP\_NONBLOCK)) {
18. // 根据Timer事件计算evsel->dispatch的最大等待时间(超时值最小)
19. timeout\_next(base, &tv\_p);
20. } **else** { //不进行等待
21. //把等待时间置为0，即可不进行等待，马上触发事件
22. evutil\_timerclear(&tv);
23. }
25. res = evsel->dispatch(base, tv\_p);
27. //处理超时事件，将超时事件插入到激活链表中
28. timeout\_process(base);
30. **if** (N\_ACTIVE\_CALLBACKS(base)) {
31. **int** n = event\_process\_active(base);
32. }
33. }
35. done:
36. base->running\_loop = 0;
37. EVBASE\_RELEASE\_LOCK(base, th\_base\_lock);
39. **return** (retval);
40. }

43. //选出超时值最小的那个
44. **static** **int**
45. timeout\_next(**struct** event\_base \*base, **struct** timeval \*\*tv\_p)
46. {
47. /\* Caller must hold th\_base\_lock \*/
48. **struct** timeval now;
49. **struct** event \*ev;
50. **struct** timeval \*tv = \*tv\_p;
51. **int** res = 0;
53. // 堆的首元素具有最小的超时值，这个是小根堆的性质。
54. ev = min\_heap\_top(&base->timeheap);
56. //堆中没有元素
57. **if** (ev == NULL) {
58. \*tv\_p = NULL;
59. **goto** out;
60. }
62. //获取当然时间
63. **if** (gettime(base, &now) == -1) {
64. res = -1;
65. **goto** out;
66. }
68. // 如果超时时间<=当前时间，不能等待，需要立即返回
69. // 因为ev\_timeout这个时间是由event\_add调用时的绝对时间 + 相对时间。所以ev\_timeout是
70. // 绝对时间。可能在调用event\_add之后，过了一段时间才调用event\_base\_diapatch,所以
71. // 现在可能都过了用户设置的超时时间。
72. **if** (evutil\_timercmp(&ev->ev\_timeout, &now, <=)) {
73. evutil\_timerclear(tv); //清零，这样可以让dispatcht不会等待，马上返回
74. **goto** out;
75. }
77. // 计算等待的时间=当前时间-最小的超时时间
78. evutil\_timersub(&ev->ev\_timeout, &now, tv);
80. out:
81. **return** (res);
82. }

        上面代码的流程是：计算出本次调用多路IO复用函数的等待时间，然后调用多路IO复用函数中等待超时。

## 激活超了时的event：

        上面代码中的timeout\_process函数就是处理超了时的event。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38637671) [copy](https://blog.csdn.net/luotuo44/article/details/38637671)

1. //event.c文件
2. //把超时了的event，放到激活队列中。并且，其激活原因设置为EV\_TIMEOUT
3. **static** **void**
4. timeout\_process(**struct** event\_base \*base)
5. {
6. /\* Caller must hold lock. \*/
7. **struct** timeval now;
8. **struct** event \*ev;
10. **if** (min\_heap\_empty(&base->timeheap)) {
11. **return**;
12. }
14. gettime(base, &now);
16. //遍历小根堆的元素。之所以不是只取堆顶那一个元素，是因为当主线程调用多路IO复用函数
17. //进入等待时，次线程可能添加了多个超时值更小的event
18. **while** ((ev = min\_heap\_top(&base->timeheap))) {
19. //ev->ev\_timeout存的是绝对时间
20. //超时时间比此刻时间大，说明该event还没超时。那么余下的小根堆元素更不用检查了。
21. **if** (evutil\_timercmp(&ev->ev\_timeout, &now, >))
22. **break**;

25. //下面说到的del是等同于调用event\_del.把event从这个event\_base中(所有的队列都)
26. //删除。event\_base不再监听之。
27. //这里是timeout\_process函数。所以对于有超时的event，才会被del掉。
28. //对于有EV\_PERSIST选项的event，在处理激活event的时候，会再次添加进event\_base的。
29. //这样做的一个好处就是，再次添加的时候，又可以重新计算该event的超时时间(绝对时间)。
30. event\_del\_internal(ev);
32. //把这个event加入到event\_base的激活队列中。
33. //event\_base的激活队列又有该event了。所以如果该event是EV\_PERSIST的，是可以
34. //再次添加进该event\_base的
35. event\_active\_nolock(ev, EV\_TIMEOUT, 1);
36. }
37. }

        当从多路IO复用函数返回时，就检查时间小根堆，看有多少个event已经超时了。如果超时了，那就把这个event加入到event\_base的激活队列中。并且把这个超时del(删除)掉，这主要是用于非PERSIST 超时event的。删除一个event的具体操作可以查看[这里](http://blog.csdn.net/luotuo44/article/details/38739549#t3)。

        把一个event添加进激活队列后的工作流程可以参考《[Libevent工作流程探究](http://blog.csdn.net/luotuo44/article/details/38501341)》一文。

# 处理永久超时event：

        现在来看一下如果该超时event有EV\_PERSIST选项，在后面是怎么再次添加进event\_base，因为前面的代码注释中已经说了，在选出超时event时，会把超时的event从event\_base中delete掉。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38637671) [copy](https://blog.csdn.net/luotuo44/article/details/38637671)

1. //event.c文件
2. **int**
3. event\_assign(**struct** event \*ev, **struct** event\_base \*base, evutil\_socket\_t fd,
4. **short** events, **void** (\*callback)(evutil\_socket\_t, **short**, **void** \*), **void** \*arg)
5. {
6. ...
7. **if** (events & EV\_PERSIST) {
8. ev->ev\_closure = EV\_CLOSURE\_PERSIST;
9. } **else** {
10. ev->ev\_closure = EV\_CLOSURE\_NONE;
11. }
13. **return** 0;
14. }

17. **static** **int**
18. event\_process\_active\_single\_queue(**struct** event\_base \*base,
19. **struct** event\_list \*activeq)
20. {
21. **struct** event \*ev;
23. //遍历同一优先级的所有event
24. **for** (ev = TAILQ\_FIRST(activeq); ev; ev = TAILQ\_FIRST(activeq)) {
26. //下面这个if else 是用于IO event的。这里贴出，是为了了解一些非超时event是
27. //怎么处理永久事件(EV\_PERSIST)的。
28. //如果是永久事件，那么只需从active队列中删除。
29. **if** (ev->ev\_events & EV\_PERSIST)
30. event\_queue\_remove(base, ev, EVLIST\_ACTIVE);
31. **else** //不是的话，那么就要把这个event删除掉。
32. event\_del\_internal(ev);

35. **switch** (ev->ev\_closure) {
36. //这个case只对超时event的EV\_PERSIST才有用。IO的没有用
37. **case** EV\_CLOSURE\_PERSIST:
38. event\_persist\_closure(base, ev);
39. **break**;
41. **default**: //默认是EV\_CLOSURE\_NONE
42. **case** EV\_CLOSURE\_NONE:
43. //没有设置EV\_PERSIST的超时event，就只有一次的监听机会
44. (\*ev->ev\_callback)(
45. ev->ev\_fd, ev->ev\_res, ev->ev\_arg);
46. **break**;
47. }
48. }
49. }

52. **static** **inline** **void**
53. event\_persist\_closure(**struct** event\_base \*base, **struct** event \*ev)
54. {
55. //在event\_add\_internal函数中，如果是超时event并且有EV\_PERSIST，那么就会把ev\_io\_timeout设置成
56. //用户设置的超时时间(相对时间)。否则为0。即不进入判断体中。
57. //说明这个if只用于处理具有EV\_PERSIST属性的超时event
58. **if** (ev->ev\_io\_timeout.tv\_sec || ev->ev\_io\_timeout.tv\_usec) {
59. **struct** timeval run\_at, relative\_to, delay, now;
60. ev\_uint32\_t usec\_mask = 0;
62. gettime(base, &now);
64. //delay是用户设置的超时时间。event\_add的第二个参数
65. delay = ev->ev\_io\_timeout;
66. //是因为超时才执行到这里，event可以同时监听多种事件。如果是由于可读而执行
67. //到这里，那么就说明还没超时。
68. **if** (ev->ev\_res & EV\_TIMEOUT) { //如果是因为超时而激活，那么下次超时就是本次超时的
69. relative\_to = ev->ev\_timeout; // 加上 delay 时间。
70. } **else** {
71. relative\_to = now; //重新计算超时值
72. }
74. evutil\_timeradd(&relative\_to, &delay, &run\_at);
75. //无论relative是哪个时间，run\_at都不应该小于now。
76. //如果小于，则说明是用户手动修改了系统时间，使得gettime()函数获取了一个
77. //之前的时间。比如现在是9点，用户手动调回到7点。
78. **if** (evutil\_timercmp(&run\_at, &now, <)) {
79. //那么就以新的系统时间为准
80. evutil\_timeradd(&now, &delay, &run\_at);
81. }
83. //把这个event再次添加到event\_base中。注意，此时第三个参数为1，说明是一个绝对时间
84. event\_add\_internal(ev, &run\_at, 1);
85. }
86. EVBASE\_RELEASE\_LOCK(base, th\_base\_lock);
87. (\*ev->ev\_callback)(ev->ev\_fd, ev->ev\_res, ev->ev\_arg);//执行回调函数
88. }

        这段代码的处理流程是：如果用户指定了EV\_PERSIST，那么在event\_assign中就记录下来。在event\_process\_active\_single\_queue函数中会针对永久event进行调用event\_persist\_closure函数对之进行处理。在event\_persist\_closure函数中，如果是一般的永久event，那么就直接调用该event的回调函数。如果是超时永久event，那么就需要再次计算新的超时时间，并将这个event再次插入到event\_base中。

        这段代码也指明了，如果一个event因可读而被激活，那么其超时时间就要重新计算。而不是之前的那个了。也就是说，如果一个event设置了3秒的超时，但1秒后就可读了，那么下一个超时值，就要重新计算设置，而不是2秒后。

        从前面的源码分析也可以得到：如果一个event监听可读的同时也设置了超时值，并且一直没有数据可读，最后超时了，那么这个event将会被删除掉，不会再等。

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/38637671