<https://blog.csdn.net/luotuo44/article/details/38739549>

**转载请注明出处:**[**http://blog.csdn.net/luotuo44/article/details/38739549**](http://blog.csdn.net/luotuo44/article/details/38739549)

        Libevent提供了一些与event相关的操作函数和操作。本文就重点讲一下这方面的源代码。

        在Libevent中，无论是event还是event\_base，都是使用指针而不会使用变量。实际上，如果查看Libevent不同的版本，就可以发现event和event\_base这两个结构体的成员是不同的。对比libevent-2.0.21-stable和libevent-1.4.13-stable这两个版本，就可以发现其具有相当大的区别。

# ****event的参数:****

        一个event结构体和很多东西相关联，比如event\_base、文件描述符fd、回调函数、回调参数等等，下文把这些东西统一称为参数。这些参数都是在调用event\_new创建一个event时指定的。如果在后面需要再次获取这些参数时，可以通过一些函数来获取，而不应该直接访问event结构体的成员。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38739549) [copy](https://blog.csdn.net/luotuo44/article/details/38739549)

1. //event.c文件
2. evutil\_socket\_t //监听的文件描述符fd
3. event\_get\_fd(**const** **struct** event \*ev)
4. {
5. **return** ev->ev\_fd;
6. }
8. **struct** event\_base \* //获取event\_base
9. event\_get\_base(**const** **struct** event \*ev)
10. {
11. **return** ev->ev\_base;
12. }
14. **short** //获取该event监听的事件
15. event\_get\_events(**const** **struct** event \*ev)
16. {
17. **return** ev->ev\_events;
18. }
20. event\_callback\_fn //获取回调函数的函数指针
21. event\_get\_callback(**const** **struct** event \*ev)
22. {
23. **return** ev->ev\_callback;
24. }

27. **void** \* //获取回调函数参数
28. event\_get\_callback\_arg(**const** **struct** event \*ev)
29. {
30. **return** ev->ev\_arg;
31. }

34. **void** //一个函数获取所有
35. event\_get\_assignment(**const** **struct** event \*event, **struct** event\_base \*\*base\_out, evutil\_socket\_t \*fd\_out,
36. **short** \*events\_out, event\_callback\_fn \*callback\_out, **void** \*\*arg\_out)
37. {
38. **if** (base\_out)
39. \*base\_out = event->ev\_base;
40. **if** (fd\_out)
41. \*fd\_out = event->ev\_fd;
42. **if** (events\_out)
43. \*events\_out = event->ev\_events;
44. **if** (callback\_out)
45. \*callback\_out = event->ev\_callback;
46. **if** (arg\_out)
47. \*arg\_out = event->ev\_arg;
48. }

        前面的那些函数是获取单个参数的，最后那个函数可以同时获取多个参数。并且如果不想获取某个参数，可以对应地传入一个NULL。

# ****event的状态:****

        一个event是可以有多个状态的，比如已初始化状态(initialized)、未决状态(pending)、激活状态(active)。

        可以用event\_initialized函数检测一个event是否处于已初始化状态：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38739549) [copy](https://blog.csdn.net/luotuo44/article/details/38739549)

1. //event.c文件
2. **int**
3. event\_initialized(**const** **struct** event \*ev)
4. {
5. **if** (!(ev->ev\_flags & EVLIST\_INIT))
6. **return** 0;
8. **return** 1;
9. }

        可以看到event\_initialized只是检查event的ev\_flags是否有EVLIST\_INIT标志。从之前的博文可以知道，当用户调用event\_new后，就会为event加入该标志，所以用event\_new创建的even都是处于已初始化状态的。

        当用户调用event\_new创建一个event后，它还没处于未决状态(non-pending)，当用户调用event\_add函数，将一个event插入到event\_base队列后，就处于未决状态(pending)。

        如果event监听的事件发生了或者超时了，那么该event就会被激活，处于激活状态。当event的回调函数被调用后，它就不再是激活状态了，但还是处于未决状态。如果用户调用了event\_del或者event\_free(该函数内部调用event\_del)，那么该event就不再是未决状态了。

        可以调用event\_pending函数来检查event处于哪种事件的未决状态。但是该函数不仅仅会检查event的未决状态，还会检查event的激活状态。名不副实啊！！下面就看一下这个函数吧。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38739549) [copy](https://blog.csdn.net/luotuo44/article/details/38739549)

1. //event.c文件
2. **int**
3. event\_pending(**const** **struct** event \*ev, **short** event, **struct** timeval \*tv)
4. {
5. **int** flags = 0;
7. **if** (EVUTIL\_FAILURE\_CHECK(ev->ev\_base == NULL)) {
8. event\_warnx("%s: event has no event\_base set.", \_\_func\_\_);
9. **return** 0;
10. }
12. EVBASE\_ACQUIRE\_LOCK(ev->ev\_base, th\_base\_lock);
14. //flags记录用户监听了哪些事件
15. **if** (ev->ev\_flags & EVLIST\_INSERTED)
16. flags |= (ev->ev\_events & (EV\_READ|EV\_WRITE|EV\_SIGNAL));
18. //flags记录event被什么事件激活了.用户可以调用event\_active
19. //手动激活event，并且可以使用之前用户没有监听的事件作为激活原因
20. **if** (ev->ev\_flags & EVLIST\_ACTIVE)
21. flags |= ev->ev\_res;
23. //记录该event是否还有超时属性
24. **if** (ev->ev\_flags & EVLIST\_TIMEOUT)
25. flags |= EV\_TIMEOUT;
27. //event可以被用户乱设值，然后作为参数。这里为了保证
28. //其值只能是下面的事件。
29. event &= (EV\_TIMEOUT|EV\_READ|EV\_WRITE|EV\_SIGNAL);
31. /\* See if there is a timeout that we should report \*/
32. **if** (tv != NULL && (flags & event & EV\_TIMEOUT)) {
33. **struct** timeval tmp = ev->ev\_timeout;
34. tmp.tv\_usec &= MICROSECONDS\_MASK;
35. #if defined(\_EVENT\_HAVE\_CLOCK\_GETTIME) && defined(CLOCK\_MONOTONIC)
36. /\* correctly remamp to real time \*/
37. evutil\_timeradd(&ev->ev\_base->tv\_clock\_diff, &tmp, tv);
38. #else
39. \*tv = tmp;
40. #endif
41. }
43. EVBASE\_RELEASE\_LOCK(ev->ev\_base, th\_base\_lock);
45. **return** (flags & event);
46. }

        该函数的作用是检查某个事件(由第二个参数指定)是否处于未决或者激活状态。

        由flags的几个 |= 操作可知，它会把event监听的事件种类都记录下来。并且还会把event被激活的原因(也是一个事件)记录下来。下面会讲到手动激活一个event。所以event可能会被一个没有监听的事件锁激活。

        如果该函数的第三个参数不为NULL，并且用户之前也让这个event监听了超时事件，而且用户在第二个参数中指明了要检查超时事件，那么将第三个参数将被赋值为该event的下次超时时间(绝对时间)。

        event\_pending函数的一个作用是可以判断一个event是否已经从event\_base中删除了。比如说，某个event监听写事件而加入了event\_base，但可能在某个时刻被删除。那么可以用下面的代码判断这个event是否已经被删除了。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38739549) [copy](https://blog.csdn.net/luotuo44/article/details/38739549)

1. **if**( event\_pending(ev, EV\_WRITE, NULL) == 0 )
2. printf("delete\n");
3. **else**
4. printf("no delete\n");

# ****手动激活event：****

        除了运行event\_base\_dispatch死等外界条件把event激活外，Libevent还提供了一个API函数event\_active，可以手动地把一个event激活。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38739549) [copy](https://blog.csdn.net/luotuo44/article/details/38739549)

1. //event.c文件
2. //res是激活的原因，是诸如EV\_READ EV\_TIMEOUT之类的宏.
3. //ncalls只对EV\_SIGNAL信号有用，表示信号的次数
4. //因为IO事件不讲究次数，信号才讲究次数
5. **void**
6. event\_active(**struct** event \*ev, **int** res, **short** ncalls)
7. {
8. //加锁，可以线程安全地手动激活一个event
9. EVBASE\_ACQUIRE\_LOCK(ev->ev\_base, th\_base\_lock);
10. event\_active\_nolock(ev, res, ncalls);
11. EVBASE\_RELEASE\_LOCK(ev->ev\_base, th\_base\_lock);
12. }

15. **void**
16. event\_active\_nolock(**struct** event \*ev, **int** res, **short** ncalls)
17. {
18. **struct** event\_base \*base;
20. //该event已经是激活状态
21. **if** (ev->ev\_flags & EVLIST\_ACTIVE) {
22. ev->ev\_res |= res;
23. **return**;
24. }
26. base = ev->ev\_base;
27. ev->ev\_res = res;//记录被激活原因。以后会用到
29. ...
30. **if** (ev->ev\_events & EV\_SIGNAL) {
31. #ifndef \_EVENT\_DISABLE\_THREAD\_SUPPORT
32. **if** (base->current\_event == ev && !EVBASE\_IN\_THREAD(base)) {
33. ++base->current\_event\_waiters;
34. EVTHREAD\_COND\_WAIT(base->current\_event\_cond, base->th\_base\_lock);
35. }
36. #endif
37. ev->ev\_ncalls = ncalls;
38. ev->ev\_pncalls = NULL;
39. }
41. //将event插入到激活队列
42. event\_queue\_insert(base, ev, EVLIST\_ACTIVE);
44. //调用本函数的线程不是主线程的话，就会通知主线程。使得主线程能赶快处理激活event
45. **if** (EVBASE\_NEED\_NOTIFY(base))
46. evthread\_notify\_base(base);
47. }

        手动激活一个event的原理是：把event插入到激活队列。如果执行激活动作的线程不是主线程，那么还要唤醒主线程，让主线程及时处理激活event，不再睡眠在多路IO复用函数中。

        由于手动激活一个event是直接把这个event插入到激活队列的，所以event的被激活原因(由res参数所指定)可以不是该event监听的事件。比如说该event只监听了EV\_READ事件，那么可以调用event\_active(ev,EV\_SIGNAL, 1);用信号事件激活该event。

# ****删除event：****

        之前的博文都只是讲怎么创建event和将之add到event\_base中。现在来讲一下怎么删除一个event。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38739549) [copy](https://blog.csdn.net/luotuo44/article/details/38739549)

1. **void**
2. event\_free(**struct** event \*ev)
3. {
4. event\_del(ev);
5. mm\_free(ev);//释放内存
6. }

9. **int**
10. event\_del(**struct** event \*ev)
11. {
12. **int** res;
13. //加锁保证线程安全
14. EVBASE\_ACQUIRE\_LOCK(ev->ev\_base, th\_base\_lock);
15. res = event\_del\_internal(ev);
16. EVBASE\_RELEASE\_LOCK(ev->ev\_base, th\_base\_lock);
18. **return** (res);
19. }


23. **static** **inline** **int**
24. event\_del\_internal(**struct** event \*ev)
25. {
26. **struct** event\_base \*base;
27. **int** res = 0, notify = 0;
29. base = ev->ev\_base;
31. /\* See if we are just active executing this event in a loop \*/
32. **if** (ev->ev\_events & EV\_SIGNAL) {
33. **if** (ev->ev\_ncalls && ev->ev\_pncalls) {
34. /\* Abort loop \*///终止循环
35. \*ev->ev\_pncalls = 0;
36. }
37. }
39. //从超时集合中删除.超时集合可能是小根堆也可能是common-timeout
40. **if** (ev->ev\_flags & EVLIST\_TIMEOUT) {
41. //删除超时event并不需要通知主线程。如果该event不是最早超时的，
42. //那肯定不用通知了。如果是的话，那么主线程会醒来。醒来后，
43. //主线程还是会再次检查超时集合中有哪些超时event超时了。这个被
44. //删除的超时event自然也检查不出来。主线程只会空手而回。
45. event\_queue\_remove(base, ev, EVLIST\_TIMEOUT);
46. }
48. //该event已经在active队列中了。那么需要在active队列中删除之
49. **if** (ev->ev\_flags & EVLIST\_ACTIVE)
50. event\_queue\_remove(base, ev, EVLIST\_ACTIVE);
52. //该event已经在注册队列(eventqueue)中了，那么需要在注册队列中删除之
53. **if** (ev->ev\_flags & EVLIST\_INSERTED) {
54. event\_queue\_remove(base, ev, EVLIST\_INSERTED);
56. //此外还要在该fd或者sig队列中删除之。同一个fd可以有多个event。
57. //所以这里还有一个队列
58. **if** (ev->ev\_events & (EV\_READ|EV\_WRITE))
59. res = evmap\_io\_del(base, ev->ev\_fd, ev);
60. **else**
61. res = evmap\_signal\_del(base, (**int**)ev->ev\_fd, ev);
62. **if** (res == 1) {
63. /\* evmap says we need to notify the main thread. \*/
64. notify = 1;
65. res = 0;
66. }
67. }
69. //可能需要通知主线程
70. **if** (res != -1 && notify && EVBASE\_NEED\_NOTIFY(base))
71. evthread\_notify\_base(base);
73. **return** (res);
74. }

        虽然要调用三个函数才能删除一个event，不过思路还是挺清晰的。删除的时候要加锁，删除完后要释放内存。从之前的博文也可以知道，一个event是会被加入到各种队列中的。所以将一个event删除，所做的工作主要是：将这个event从各种队列中删除掉。

        删除一个event这个操作可能不是主线程调用的，这时就可能需要通知主线程。关于通知主线程的原理可以参考博文《[evthread\_notify\_base通知主线程](http://blog.csdn.net/luotuo44/article/details/38556059)》。

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/38739549