#### [Libevent基本流程](http://blog.csdn.net/u012658346/article/details/44746313)

#### 以libevent的sample中的time\_test.c为例，在windows下单步调试，分析其基本流程。

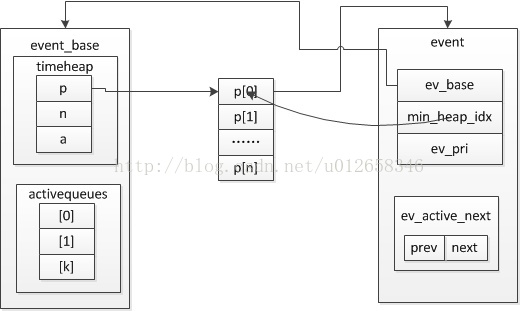
**[cpp]** [view plaincopy](http://blog.csdn.net/u012658346/article/details/44746313)[在CODE上查看代码片](https://code.csdn.net/snippets/632644)

1. //time\_test.c源码如下:
2. #include <sys/types.h>
3. #include <event2/event-config.h>
4. #include <sys/stat.h>
5. #include <time.h>
6. #include <fcntl.h>
7. #include <stdlib.h>
8. #include <stdio.h>
9. #include <string.h>
10. #include <errno.h>
12. #include <event2/event.h>
13. #include <event2/event\_struct.h>
14. #include <event2/util.h>
15. #include <winsock2.h>
17. **struct** timeval lasttime;
18. **int** event\_is\_persistent;
20. **static** **void**
21. timeout\_cb(evutil\_socket\_t fd, **short** event, **void** \*arg)
22. {
23. **struct** timeval newtime, difference;
24. **struct** event \*timeout = arg;
25. **double** elapsed;
27. evutil\_gettimeofday(&newtime, NULL);
28. evutil\_timersub(&newtime, &lasttime, &difference);
29. elapsed = difference.tv\_sec +
30. (difference.tv\_usec / 1.0e6);
32. printf("timeout\_cb called at %d: %.3f seconds elapsed.\n",
33. (**int**)newtime.tv\_sec, elapsed);
34. lasttime = newtime;
36. **if** (! event\_is\_persistent) {
37. **struct** timeval tv;
38. evutil\_timerclear(&tv);
39. tv.tv\_sec = 2;
40. event\_add(timeout, &tv);//对于非永久事件，则再次将其加入到事件处理框架中，否则事件只会被处理一次
41. }
42. }
44. **int**
45. main(**int** argc, **char** \*\*argv)
46. {
47. **struct** event timeout;
48. **struct** timeval tv;
49. **struct** event\_base \*base;
50. **int** flags;
52. **WORD** wVersionRequested;
53. WSADATA wsaData;
54. wVersionRequested = MAKEWORD(2, 2);
55. (**void**)WSAStartup(wVersionRequested, &wsaData);
57. event\_is\_persistent = 0;
58. flags = 0;
60. /\* Initalize the event library \*/
61. base = event\_base\_new();
63. /\* Initalize one event \*/
64. event\_assign(&timeout, base, -1, flags, timeout\_cb, (**void**\*) &timeout);
66. evutil\_timerclear(&tv);
67. tv.tv\_sec = 2;
68. event\_add(&timeout, &tv);
70. evutil\_gettimeofday(&lasttime, NULL);
72. event\_base\_dispatch(base);
74. **return** (0);
75. }

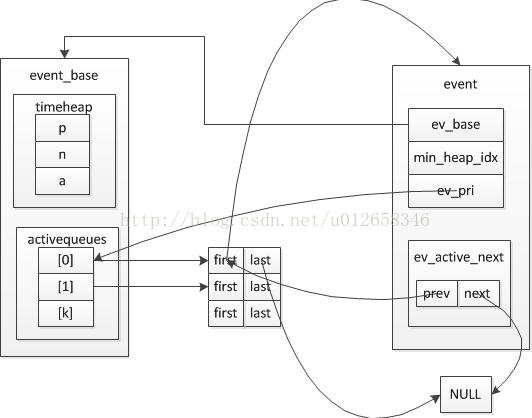
由上述代码可知，本例是一个简单的超时事件，但超时时间到达时，调用回调函数输出打印信息，同时由于该事件在main中设置为非永久事件，因此在回调函数中，会再次调用event\_add()将事件再次插入到事件处理框架中。

由于本例是一个简单的超时事件，同时通过查看event\_base和event的数据结构可知，事件只会出现在小根堆或者激活队列中，也即base的struct min\_heap timeheap;和struct event\_list \*activequeues;

在调用event\_add()后，事件被加入到小根堆中，如下所示：event中的min\_heap\_idx记录其在小根堆中的位置



在调用event\_dispatch()后的监听过程中，进入到event\_base\_loop()循环，然后调用timeout\_process()将超时事件从小根堆中移除，并加入到激活队列中，其中event中的ev\_pri记录了它的优先级，通过这个参数可以找到其所在的优先级队列



然后在循环中调用event\_process\_active()处理激活队列中的事件，将事件从激活队列中删除，并调用回调函数

#### 1、首先初始化事件处理框架event\_base，event\_init()、event\_base\_new()最终都相当于调用event\_base\_new\_with\_config()来创建一个base

**[cpp]** [view plaincopy](http://blog.csdn.net/u012658346/article/details/44746313)[在CODE上查看代码片](https://code.csdn.net/snippets/632644)

1. **struct** event\_base \*
2. event\_init(**void**)
3. {
4. **struct** event\_base \*base = event\_base\_new\_with\_config</strong>(NULL);
5. **return** (base);
6. }
8. **struct** event\_base \*
9. event\_base\_new(**void**)
10. {
11. **struct** event\_base \*base = NULL;
12. **struct** event\_config \*cfg = event\_config\_new();
13. **if** (cfg) {
14. base = event\_base\_new\_with\_config(cfg);
15. event\_config\_free(cfg);
16. }
17. **return** base;
18. }

**[cpp]** [view plaincopy](http://blog.csdn.net/u012658346/article/details/44746313)[在CODE上查看代码片](https://code.csdn.net/snippets/632644)

1. **struct** event\_base \*
2. event\_base\_new\_with\_config(**const** **struct** event\_config \*cfg)
3. {
4. **int** i;
5. **struct** event\_base \*base;
6. **int** should\_check\_environment;
7. **if** ((base = mm\_calloc(1, **sizeof**(**struct** event\_base))) == NULL) { //分配内存
8. }
9. detect\_monotonic();
10. gettime(base, &base->event\_tv);
12. min\_heap\_ctor(&base->timeheap);      //初始化小根堆、事件队列和一些成员变量
13. TAILQ\_INIT(&base->eventqueue);
14. base->sig.ev\_signal\_pair[0] = -1;
15. base->sig.ev\_signal\_pair[1] = -1;
16. base->th\_notify\_fd[0] = -1;
17. base->th\_notify\_fd[1] = -1;
19. event\_deferred\_cb\_queue\_init(&base->defer\_queue);    //初始化延迟队列及相应的通知函数和参数
20. base->defer\_queue.notify\_fn = notify\_base\_cbq\_callback;
21. base->defer\_queue.notify\_arg = base;
23. evmap\_io\_initmap(&base->io);
24. evmap\_signal\_initmap(&base->sigmap);
25. event\_changelist\_init(&base->changelist);
27. base->evbase = NULL;
29. should\_check\_environment =
30. !(cfg && (cfg->flags & EVENT\_BASE\_FLAG\_IGNORE\_ENV));
32. **for** (i = 0; eventops[i] && !base->evbase; i++) {     //eventops指向编译时选择的一个select/pool/epoll/kqueue接口封装对象
33. base->evsel = eventops[i];
34. base->evbase = base->evsel->init(base);
35. }   /\* allocate a single active event queue \*/
36. **if** (event\_base\_priority\_init(base, 1) < 0) {     //初始化激活事件队列，此时nactivequeues=1,只有一个优先级为1的队列
37. }
38. ......      //为多线程做的一些处理
39. **return** (base);
40. }

#### 2、event\_new()、event\_assign()最终都是调用event\_assign()来创建一个事件变量

**[cpp]** [view plaincopy](http://blog.csdn.net/u012658346/article/details/44746313)[在CODE上查看代码片](https://code.csdn.net/snippets/632644)

1. **struct** event \*
2. event\_new(**struct** event\_base \*base, evutil\_socket\_t fd, **short** events, **void** (\*cb)(evutil\_socket\_t, **short**, **void** \*), **void** \*arg)
3. {
4. **struct** event \*ev;
5. ev = mm\_malloc(**sizeof**(**struct** event));   //分配内存
6. **if** (event\_assign(ev, base, fd, events, cb, arg) < 0) {
7. }
8. **return** (ev);
9. }
10. **int**
11. event\_assign(**struct** event \*ev, **struct** event\_base \*base, evutil\_socket\_t fd, **short** events, **void** (\*callback)(evutil\_socket\_t, **short**, **void** \*), **void** \*arg)
12. {
13. **if** (!base)
14. base = current\_base;
15. \_event\_debug\_assert\_not\_added(ev);
16. ev->ev\_base = base;//将事件与事件处理框架联系起来
17. ev->ev\_callback = callback;//回调函数
18. ev->ev\_arg = arg;
19. ev->ev\_fd = fd;
20. ev->ev\_events = events;//事件，如EV\_READ、EV\_WRITE等
21. ev->ev\_res = 0;
22. ev->ev\_flags = EVLIST\_INIT;
23. ev->ev\_ncalls = 0;
24. ev->ev\_pncalls = NULL;
26. **if** (events & EV\_SIGNAL) {
27. **if** ((events & (EV\_READ|EV\_WRITE)) != 0) {
28. **return** -1;
29. }
30. ev->ev\_closure = EV\_CLOSURE\_SIGNAL;
31. } **else** {
32. **if** (events & EV\_PERSIST) {
33. evutil\_timerclear(&ev->ev\_io\_timeout);
34. ev->ev\_closure = EV\_CLOSURE\_PERSIST;
35. } **else** {
36. ev->ev\_closure = EV\_CLOSURE\_NONE;//在此例中，对应EV\_CLOSURE\_NONE
37. }
38. }
39. min\_heap\_elem\_init(ev);     //ev->ev\_timeout\_pos.min\_heap\_idx = -1;
40. **if** (base != NULL) {
41. ev->ev\_pri = base->nactivequeues / 2;/\* by default, we put new events into the middle priority \*/
42. }
43. **return** 0;
44. }

这样就创建了事件处理框架base和事件event，并通过ev->ev\_base将二者联系到了一起

#### 3、调用event\_add()将事件插入到base的相应队列中，对于本例，则是直接加入到小根堆中

**[cpp]** [view plaincopy](http://blog.csdn.net/u012658346/article/details/44746313)[在CODE上查看代码片](https://code.csdn.net/snippets/632644)

1. **int**
2. event\_add(**struct** event \*ev, **const** **struct** timeval \*tv)
3. {
4. EVBASE\_ACQUIRE\_LOCK(ev->ev\_base, th\_base\_lock);
5. res = event\_add\_internal(ev, tv, 0);
6. EVBASE\_RELEASE\_LOCK(ev->ev\_base, th\_base\_lock);
7. **return** (res);
8. }
9. **static** **inline** **int**
10. event\_add\_internal(**struct** event \*ev, **const** **struct** timeval \*tv,
11. **int** tv\_is\_absolute)
12. {
13. **struct** event\_base \*base = ev->ev\_base;
14. **int** res = 0;
15. **int** notify = 0;
17. EVENT\_BASE\_ASSERT\_LOCKED(base);
18. \_event\_debug\_assert\_is\_setup(ev);
19. **if** (tv != NULL && !(ev->ev\_flags & EVLIST\_TIMEOUT)) {
20. **if** (min\_heap\_reserve(&base->timeheap,
21. 1 + min\_heap\_size(&base->timeheap)) == -1)       //为下面的超时事件的插入预留空间
22. **return** (-1);  /\* ENOMEM == errno \*/
23. }
24. //根据事件类型将其插入到不同的队列中
25. **if** ((ev->ev\_events & (EV\_READ|EV\_WRITE|EV\_SIGNAL)) &&
26. !(ev->ev\_flags & (EVLIST\_INSERTED|EVLIST\_ACTIVE))) {
27. **if** (ev->ev\_events & (EV\_READ|EV\_WRITE))
28. res = evmap\_io\_add(base, ev->ev\_fd, ev); //将其插入到IO事件队列中
29. **else** **if** (ev->ev\_events & EV\_SIGNAL)
30. res = evmap\_signal\_add(base, (**int**)ev->ev\_fd, ev);    //将其插入到signal事件队列中
31. **if** (res != -1)
32. event\_queue\_insert(base, ev, EVLIST\_INSERTED);  //将其插入到事件队列中
33. **if** (res == 1) {
34. /\* evmap says we need to notify the main thread. \*/
35. notify = 1;
36. res = 0;
37. }
38. }
39. //当上述三种插入队列均未完成时，进行如下操作
40. **if** (res != -1 && tv != NULL) {
41. **struct** timeval now;
42. **int** common\_timeout;
44. **if** (ev->ev\_closure == EV\_CLOSURE\_PERSIST && !tv\_is\_absolute) //对永久事件，记录其超时时间
45. ev->ev\_io\_timeout = \*tv;
47. **if** (ev->ev\_flags & EVLIST\_TIMEOUT) {
48. /\* XXX I believe this is needless. \*/
49. **if** (min\_heap\_elt\_is\_top(ev))
50. notify = 1;
51. event\_queue\_remove(base, ev, EVLIST\_TIMEOUT);
52. }
53. **if** ((ev->ev\_flags & EVLIST\_ACTIVE) &&        /\*检测事件是否已经由于超时而被激活，若是则将其从激活队列中删除，在执行回调函数之前对这个超时事件重新进行调度\*/
54. (ev->ev\_res & EV\_TIMEOUT)) {
55. **if** (ev->ev\_events & EV\_SIGNAL) {
56. **if** (ev->ev\_ncalls && ev->ev\_pncalls) {
57. /\* Abort loop \*/
58. \*ev->ev\_pncalls = 0;
59. }
60. }
61. event\_queue\_remove(base, ev, EVLIST\_ACTIVE);
62. }
63. gettime(base, &now);
64. /\*公共超时，主要用于大量具有相同超时时长的event，但超时时间不同(超时时间 = 超时时长+ 调用event\_add时间),很明显本例不是common\_timeout\*/
65. common\_timeout = is\_common\_timeout(tv, base);
66. **if** (tv\_is\_absolute) {
67. ev->ev\_timeout = \*tv;
68. } **else** **if** (common\_timeout) {
69. **struct** timeval tmp = \*tv;
70. tmp.tv\_usec &= MICROSECONDS\_MASK;
71. evutil\_timeradd(&now, &tmp, &ev->ev\_timeout);
72. ev->ev\_timeout.tv\_usec |=
73. (tv->tv\_usec & ~MICROSECONDS\_MASK);
74. } **else** {
75. evutil\_timeradd(&now, tv, &ev->ev\_timeout);//得到event的超时时间，ev->ev\_timeout=now+tmp
76. }
77. event\_queue\_insert(base, ev, EVLIST\_TIMEOUT);   //插入到超时队列中，由于不是common\_timeout，因此最终是将其插入到小根堆中
78. **if** (common\_timeout) {
79. **struct** common\_timeout\_list \*ctl =
80. get\_common\_timeout\_list(base, &ev->ev\_timeout);
81. **if** (ev == TAILQ\_FIRST(&ctl->events)) {
82. common\_timeout\_schedule(ctl, &now, ev);
83. }
84. } **else** {
85. **if** (min\_heap\_elt\_is\_top(ev))    //判断是否是堆顶
86. notify = 1;
87. }
88. }
89. /\* if we are not in the right thread, we need to wake up the loop \*/
90. **if** (res != -1 && notify && EVBASE\_NEED\_NOTIFY(base))
91. evthread\_notify\_base(base);
92. \_event\_debug\_note\_add(ev);
93. **return** (res);
94. }

#### 4、监听事件

event\_dispatch()循环当前事件所在的事件处理框架，event\_base\_dispatch()循环当前事件处理框架，二者最终都是调用event\_vase\_loop()对事件循环进行处理

**[cpp]** [view plaincopy](http://blog.csdn.net/u012658346/article/details/44746313)[在CODE上查看代码片](https://code.csdn.net/snippets/632644)

1. **int**
2. event\_dispatch(**void**)
3. {
4. **return** (event\_loop(0));
5. }
6. **int**
7. event\_loop(**int** flags)
8. {
9. **return** event\_base\_loop(current\_base, flags);
10. }
11. **int**
12. event\_base\_dispatch(**struct** event\_base \*event\_base)
13. {
14. **return** (event\_base\_loop(event\_base, 0));
15. }

##### 1.最终都是调用event\_base\_loop()对base中的事件循环处理

**[cpp]** [view plaincopy](http://blog.csdn.net/u012658346/article/details/44746313)[在CODE上查看代码片](https://code.csdn.net/snippets/632644)

1. **int**
2. event\_base\_loop(**struct** event\_base \*base, **int** flags)
3. {
4. **const** **struct** eventop \*evsel = base->evsel;
5. **struct** timeval tv;
6. **struct** timeval \*tv\_p;
7. **int** res, done, retval = 0;
9. EVBASE\_ACQUIRE\_LOCK(base, th\_base\_lock);
10. base->running\_loop = 1;
12. clear\_time\_cache(base);
14. **if** (base->sig.ev\_signal\_added && base->sig.ev\_n\_signals\_added)
15. evsig\_set\_base(base);
17. done = 0;
18. base->event\_gotterm = base->event\_break = 0;
19. **while** (!done) {     //循环处理事件
20. base->event\_continue = 0;
21. /\* Terminate the loop if we have been asked to \*/
22. **if** (base->event\_gotterm) {
23. **break**;
24. }
25. **if** (base->event\_break) {
26. **break**;
27. }
28. timeout\_correct(base, &tv);     //时间校对
29. tv\_p = &tv;
30. **if** (!N\_ACTIVE\_CALLBACKS(base) && !(flags & EVLOOP\_NONBLOCK)) {
31. timeout\_next(base, &tv\_p);
32. } **else** {
33. evutil\_timerclear(&tv);
34. }
36. /\* 没有事件，直接退出 \*/
37. **if** (!event\_haveevents(base) && !N\_ACTIVE\_CALLBACKS(base)) {
38. retval = 1;
39. **goto** done;
40. }
41. /\* update last old time \*/
42. gettime(base, &base->event\_tv);
43. clear\_time\_cache(base);
45. res = evsel->dispatch(base, tv\_p);
46. update\_time\_cache(base);
48. timeout\_process(base);//检测超时事件，若事件超时，则将其从超时队列中删除，并插入到激活队列中
50. **if** (N\_ACTIVE\_CALLBACKS(base)) {
51. **int** n = event\_process\_active(base);//处理激活队列中的事件
52. **if** ((flags & EVLOOP\_ONCE)
53. && N\_ACTIVE\_CALLBACKS(base) == 0
54. && n != 0)
55. done = 1;
56. } **else** **if** (flags & EVLOOP\_NONBLOCK)
57. done = 1;
58. }
59. done:
60. clear\_time\_cache(base);
61. base->running\_loop = 0;
62. EVBASE\_RELEASE\_LOCK(base, th\_base\_lock);
63. **return** (retval);
64. }

##### 2.timeout\_process(base)

**[cpp]** [view plaincopy](http://blog.csdn.net/u012658346/article/details/44746313)[在CODE上查看代码片](https://code.csdn.net/snippets/632644)

1. **static** **void**
2. timeout\_process(**struct** event\_base \*base)
3. {
4. /\* Caller must hold lock. \*/
5. **struct** timeval now;
6. **struct** event \*ev;
7. gettime(base, &now);
8. **while** ((ev = min\_heap\_top(&base->timeheap))) {
9. **if** (evutil\_timercmp(&ev->ev\_timeout, &now, >))
10. **break**;
11. /\* 若事件超时，则将其相应队列中删除\*/
12. event\_del\_internal(ev);
13. event\_active\_nolock(ev, EV\_TIMEOUT, 1);/\*<span style="font-family: Arial, Helvetica, sans-serif; font-size: 12px;">并插入到激活队列中\*/</span>
15. }
16. }

删除事件

**[cpp]** [view plaincopy](http://blog.csdn.net/u012658346/article/details/44746313)[在CODE上查看代码片](https://code.csdn.net/snippets/632644)

1. **static** **inline** **int**
2. event\_del\_internal(**struct** event \*ev)
3. {
4. **struct** event\_base \*base;
5. **int** res = 0, notify = 0;
7. EVENT\_BASE\_ASSERT\_LOCKED(ev->ev\_base);
8. base = ev->ev\_base;
9. EVUTIL\_ASSERT(!(ev->ev\_flags & ~EVLIST\_ALL));
10. /\* See if we are just active executing this event in a loop \*/
11. **if** (ev->ev\_events & EV\_SIGNAL) {
12. **if** (ev->ev\_ncalls && ev->ev\_pncalls) {
13. /\* Abort loop \*/
14. \*ev->ev\_pncalls = 0;
15. }
16. }
17. //将事件从相应的队列中移除
18. **if** (ev->ev\_flags & EVLIST\_TIMEOUT) {
19. event\_queue\_remove(base, ev, EVLIST\_TIMEOUT);//超时事件从超时队列中移除
20. }
21. **if** (ev->ev\_flags & EVLIST\_ACTIVE)
22. event\_queue\_remove(base, ev, EVLIST\_ACTIVE);
23. **if** (ev->ev\_flags & EVLIST\_INSERTED) {
24. event\_queue\_remove(base, ev, EVLIST\_INSERTED);
25. **if** (ev->ev\_events & (EV\_READ|EV\_WRITE))
26. res = evmap\_io\_del(base, ev->ev\_fd, ev);
27. **else**
28. res = evmap\_signal\_del(base, (**int**)ev->ev\_fd, ev);
29. **if** (res == 1) {
30. /\* evmap says we need to notify the main thread. \*/
31. notify = 1;
32. res = 0;
33. }
34. }
35. /\* if we are not in the right thread, we need to wake up the loop \*/
36. **if** (res != -1 && notify && EVBASE\_NEED\_NOTIFY(base))
37. evthread\_notify\_base(base);
38. \_event\_debug\_note\_del(ev);
39. **return** (res);
40. }
42. **static** **void**
43. event\_queue\_remove(**struct** event\_base \*base, **struct** event \*ev, **int** queue)
44. {
45. EVENT\_BASE\_ASSERT\_LOCKED(base);
46. **if** (~ev->ev\_flags & EVLIST\_INTERNAL)
47. base->event\_count--;
49. ev->ev\_flags &= ~queue;
50. **switch** (queue) {
51. **case** EVLIST\_INSERTED:
52. TAILQ\_REMOVE(&base->eventqueue, ev, ev\_next);
53. **break**;
54. **case** EVLIST\_ACTIVE:
55. base->event\_count\_active--;
56. TAILQ\_REMOVE(&base->activequeues[ev->ev\_pri],
57. ev, ev\_active\_next);
58. **break**;
59. **case** EVLIST\_TIMEOUT:
60. **if** (is\_common\_timeout(&ev->ev\_timeout, base)) {
61. **struct** common\_timeout\_list \*ctl =
62. get\_common\_timeout\_list(base, &ev->ev\_timeout);
63. TAILQ\_REMOVE(&ctl->events, ev,
64. ev\_timeout\_pos.ev\_next\_with\_common\_timeout);
65. } **else** {
66. min\_heap\_erase(&base->timeheap, ev);//不是公共超时事件，从小根堆中移除
67. }
68. **break**;
69. **default**:
70. event\_errx(1, "%s: unknown queue %x", \_\_func\_\_, queue);
71. }
72. }

将超时事件插入到激活队列中

**[cpp]** [view plaincopy](http://blog.csdn.net/u012658346/article/details/44746313)[在CODE上查看代码片](https://code.csdn.net/snippets/632644)

1. **void**
2. event\_active\_nolock(**struct** event \*ev, **int** res, **short** ncalls)
3. {
4. **struct** event\_base \*base;
6. /\* We get different kinds of events, add them together \*/
7. **if** (ev->ev\_flags & EVLIST\_ACTIVE) {
8. ev->ev\_res |= res;
9. **return**;
10. }
11. base = ev->ev\_base;
12. EVENT\_BASE\_ASSERT\_LOCKED(base);
13. ev->ev\_res = res;
14. **if** (ev->ev\_pri < base->event\_running\_priority)
15. base->event\_continue = 1;
16. **if** (ev->ev\_events & EV\_SIGNAL) {
17. ev->ev\_ncalls = ncalls;
18. ev->ev\_pncalls = NULL;
19. }
20. event\_queue\_insert(base, ev, EVLIST\_ACTIVE);    //将事件插入到激活事件队列中，等待处理
21. **if** (EVBASE\_NEED\_NOTIFY(base))
22. evthread\_notify\_base(base);
23. }
24. **static** **void**
25. event\_queue\_insert(**struct** event\_base \*base, **struct** event \*ev, **int** queue)
26. {
27. EVENT\_BASE\_ASSERT\_LOCKED(base);
28. **if** (~ev->ev\_flags & EVLIST\_INTERNAL)
29. base->event\_count++;
30. ev->ev\_flags |= queue;
31. **switch** (queue) {
32. **case** EVLIST\_INSERTED:
33. TAILQ\_INSERT\_TAIL(&base->eventqueue, ev, ev\_next);    //将事件插入到事件队列中
34. **break**;
35. **case** EVLIST\_ACTIVE:
36. base->event\_count\_active++;
37. TAILQ\_INSERT\_TAIL(&base->activequeues[ev->ev\_pri], //根据事件的优先级<span style="font-family: Arial, Helvetica, sans-serif; font-size: 12px;">ev->ev\_pri</span><span style="font-size: 12px; font-family: Arial, Helvetica, sans-serif;">，将其插入到对应的激活队列中</span>
38. ev,ev\_active\_next);
39. **break**;
40. **case** EVLIST\_TIMEOUT: {
41. **if** (is\_common\_timeout(&ev->ev\_timeout, base)) {
42. **struct** common\_timeout\_list \*ctl =
43. get\_common\_timeout\_list(base, &ev->ev\_timeout);
44. insert\_common\_timeout\_inorder(ctl, ev);
45. } **else**
46. min\_heap\_push(&base->timeheap, ev);  //超时事件，且不是common\_timeout，将其插入到小根堆中
47. **break**;
48. }
49. **default**:
50. event\_errx(1, "%s: unknown queue %x", \_\_func\_\_, queue);
51. }
52. }

##### 3.event\_process\_active()对激活事件进行处理

**[cpp]** [view plaincopy](http://blog.csdn.net/u012658346/article/details/44746313)[在CODE上查看代码片](https://code.csdn.net/snippets/632644)

1. **static** **int**
2. event\_process\_active(**struct** event\_base \*base)
3. {
4. /\* Caller must hold th\_base\_lock \*/
5. **struct** event\_list \*activeq = NULL;
6. **int** i, c = 0;
7. **for** (i = 0; i < base->nactivequeues; ++i) {
8. **if** (TAILQ\_FIRST(&base->activequeues[i]) != NULL) {
9. base->event\_running\_priority = i;
10. activeq = &base->activequeues[i];
11. c = event\_process\_active\_single\_queue(base, activeq);//依次处理每个优先级对应的激活事件队列
12. **if** (c < 0) {
13. base->event\_running\_priority = -1;
14. **return** -1;
15. } **else** **if** (c > 0)
16. **break**;
17. }
18. }
19. event\_process\_deferred\_callbacks(&base->defer\_queue,&base->event\_break);
20. base->event\_running\_priority = -1;
21. **return** c;
22. }
23. **static** **int**
24. event\_process\_active\_single\_queue(**struct** event\_base \*base,
25. **struct** event\_list \*activeq)
26. {
27. **struct** event \*ev;
28. **int** count = 0;
29. EVUTIL\_ASSERT(activeq != NULL);
30. **for** (ev = TAILQ\_FIRST(activeq); ev; ev = TAILQ\_FIRST(activeq)) {
31. **if** (ev->ev\_events & EV\_PERSIST)
32. event\_queue\_remove(base, ev, EVLIST\_ACTIVE);    //若是永久事件，将其从激活队列中移除
33. **else**
34. event\_del\_internal(ev);     //否则直接删除该事件
35. **if** (!(ev->ev\_flags & EVLIST\_INTERNAL))
36. ++count;
37. **switch** (ev->ev\_closure) {
38. **case** EV\_CLOSURE\_SIGNAL:
39. event\_signal\_closure(base, ev);
40. **break**;
41. **case** EV\_CLOSURE\_PERSIST:
42. event\_persist\_closure(base, ev);
43. **break**;
44. **default**:
45. **case** EV\_CLOSURE\_NONE:
46. EVBASE\_RELEASE\_LOCK(base, th\_base\_lock);
47. (\*ev->ev\_callback)(//调用回调函数，输出打印信息，同时调用event\_add()将事件再次加入到base中
48. ev->ev\_fd, ev->ev\_res, ev->ev\_arg);
49. **break**;
50. }
51. EVBASE\_ACQUIRE\_LOCK(base, th\_base\_lock);
52. **if** (base->event\_break)
53. **return** -1;
54. **if** (base->event\_continue)
55. **break**;
56. }
57. **return** count;
58. }

然后继续循环

#### 总结

1)首先调用event\_init()或event\_base\_new()创建一个事件处理框架

2)调用event\_new()或event\_assign()创建一个事件

3)调用event\_add()将事件插入到base中的相应队列中

4)调用event\_base\_dispatch()循环监听事件

在event\_base\_dispatch中，则主要是调用timeout\_process()和event\_process\_active()：

其中timeout\_process()检测超时事件，若事件超时，则将其从超时队列中删除，并插入到激活队列中

event\_process\_active()则是对激活队列中的事件进行处理