<https://blog.csdn.net/luotuo44/article/details/38501341>

**转载请注明出处：**[**http://blog.csdn.net/luotuo44/article/details/38501341**](http://blog.csdn.net/luotuo44/article/details/38501341)

        之前的博文讲了很多Libevent的基础构件，现在以一个实际例子来初步探究Libevent的基本工作流程。由于还有很多Libevent的细节并没有讲所以，这里的探究还是比较简洁，例子也相当简单。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38501341) [copy](https://blog.csdn.net/luotuo44/article/details/38501341)

1. #include<unistd.h>
2. #include<stdio.h>
3. #include<event.h>
4. #include<thread.h>

7. **void** cmd\_cb(**int** fd, **short** events, **void** \*arg)
8. {
9. **char** buf[1024];
10. printf("in the cmd\_cb\n");
12. read(fd, buf, **sizeof**(buf));
13. }

16. **int** main()
17. {
18. evthread\_use\_pthreads();
20. //使用默认的event\_base配置
21. **struct** event\_base \*base = event\_base\_new();
23. **struct** event \*cmd\_ev = event\_new(base, STDIN\_FILENO,
24. EV\_READ | EV\_PERSIST, cmd\_cb, NULL);
26. event\_add(cmd\_ev, NULL); //没有超时
28. event\_base\_dispatch(base);
30. **return** 0;
31. }

        上面代码估计是不会比读者写的第一个Libevent程序复杂。但这已经包含了Libevent的基础工作流程。这里将进入这些函数的内部探究，并且只会讲解之前博文出现过的，没出现的，尽量不讲。在讲解之前，要先了解一下struct event这个结构体。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38501341) [copy](https://blog.csdn.net/luotuo44/article/details/38501341)

1. **struct** event {
2. TAILQ\_ENTRY(event) ev\_active\_next; //激活队列
3. TAILQ\_ENTRY(event) ev\_next; //注册事件队列
4. /\* for managing timeouts \*/
5. **union** {
6. TAILQ\_ENTRY(event) ev\_next\_with\_common\_timeout;
7. **int** min\_heap\_idx; //指明该event结构体在堆的位置
8. } ev\_timeout\_pos; //仅用于定时事件处理器(event).EV\_TIMEOUT类型
10. //对于I/O事件，是文件描述符；对于signal事件，是信号值
11. evutil\_socket\_t ev\_fd;
13. **struct** event\_base \*ev\_base; //所属的event\_base
15. //因为信号和I/O是不能同时设置的。所以可以使用共用体以省内存
16. //在低版本的Libevent，两者是分开的，不在共用体内。
17. **union** {
18. //无论是信号还是IO，都有一个TAILQ\_ENTRY的队列。它用于这样的情景:
19. //用户对同一个fd调用event\_new多次，并且都使用了不同的回调函数。
20. //每次调用event\_new都会产生一个event\*。这个xxx\_next成员就是把这些
21. //event连接起来的。
23. /\* used for io events \*/
24. //用于IO事件
25. **struct** {
26. TAILQ\_ENTRY(event) ev\_io\_next;
27. **struct** timeval ev\_timeout;
28. } ev\_io;
30. /\* used by signal events \*/
31. //用于信号事件
32. **struct** {
33. TAILQ\_ENTRY(event) ev\_signal\_next;
34. **short** ev\_ncalls; //事件就绪执行时，调用ev\_callback的次数         /\* Allows deletes in callback \*/
35. **short** \*ev\_pncalls; //指针，指向次数
36. } ev\_signal;
37. } \_ev;
39. **short** ev\_events;//记录监听的事件类型 EV\_READ EVTIMEOUT之类
40. **short** ev\_res;       /\* result passed to event callback \*///记录了当前激活事件的类型
41. //libevent用于标记event信息的字段，表明其当前的状态.
42. //可能值为前面的EVLIST\_XXX
43. **short** ev\_flags;
45. //本event的优先级。调用event\_priority\_set设置
46. ev\_uint8\_t ev\_pri;
47. ev\_uint8\_t ev\_closure;
48. **struct** timeval ev\_timeout;//用于定时器,指定定时器的超时值
50. /\* allows us to adopt for different types of events \*/
51. **void** (\*ev\_callback)(evutil\_socket\_t, **short**, **void** \*arg); //回调函数
52. **void** \*ev\_arg; //回调函数的参数
53. };

        event结构体里面有几个TAILQ\_ENTRY队列节点类型。这里因为一个event是会同时处于多个队列之中。比如前几篇博文说到的同一个文件描述符或者信号值对应的多个event会被连在一起，所有的被加入到event\_base的event也会连在一起，所有被激活的event也会被连在一起。所以会有多个QAILQ\_ENTRY。

        event结构体只有一两个之前没有说到的概念，这不妨碍理解event结构体。而event\_base结构体则会太多之前没有说到的概念，所以这里就不贴出event\_base的代码了。

        在读这篇博文前，最好读一下前面几篇博文，因为会用到其他讲到的东西。如果之前有讲过的东西，这里也将一笔带过。

        好了，开始探究。

        最前面的evthread\_use\_pthreads();就不多说了，看《[多线程、锁、条件变量(一)](http://blog.csdn.net/luotuo44/article/details/38350633)》和《[多线程、锁、条件变量(二)](http://blog.csdn.net/luotuo44/article/details/38360525)》这两篇博文吧。

# 创建event\_base：

        下面看一下event\_base\_new函数。它是由event\_base\_new\_with\_config函数实现的。我们还是看后面那个函数吧。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38501341) [copy](https://blog.csdn.net/luotuo44/article/details/38501341)

1. //event.c文件
2. **struct** event\_base \*
3. event\_base\_new\_with\_config(**const** **struct** event\_config \*cfg)
4. {
5. **int** i;
6. **struct** event\_base \*base;
7. **int** should\_check\_environment;

10. //之所以不用mm\_malloc是因为mm\_malloc并不会清零该内存区域。
11. //而这个函数是会清零申请到的内存区域，这相当于被base初始化
12. **if** ((base = mm\_calloc(1, **sizeof**(**struct** event\_base))) == NULL) {
13. event\_warn("%s: calloc", \_\_func\_\_);
14. **return** NULL;
15. }
17. ...
19. TAILQ\_INIT(&base->eventqueue);
21. ...
23. **if** (cfg)
24. base->flags = cfg->flags;
26. evmap\_io\_initmap(&base->io);
27. evmap\_signal\_initmap(&base->sigmap);
29. base->evbase = NULL;
31. should\_check\_environment =
32. !(cfg && (cfg->flags & EVENT\_BASE\_FLAG\_IGNORE\_ENV));
34. //选择IO复用结构体
35. **for** (i = 0; eventops[i] && !base->evbase; i++) {
36. **if** (cfg != NULL) {
37. /\* determine if this backend should be avoided \*/
38. **if** (event\_config\_is\_avoided\_method(cfg,
39. eventops[i]->name))
40. **continue**;
41. **if** ((eventops[i]->features & cfg->require\_features)
42. != cfg->require\_features)
43. **continue**;
44. }
46. **if** (should\_check\_environment &&
47. event\_is\_method\_disabled(eventops[i]->name))
48. **continue**;
50. //找到一个满足条件的多路IO复用函数
51. base->evsel = eventops[i];
53. //初始化ev\_base。并且会对信号监听的处理也进行初始化
54. base->evbase = base->evsel->init(base);
55. }


59. #ifndef \_EVENT\_DISABLE\_THREAD\_SUPPORT
60. //测试evthread\_lock\_callbacks结构中的lock指针函数是否为NULL
61. //即测试Libevent是否已经初始化为支持多线程模式。
62. //由于一开始是用mm\_calloc申请内存的，所以该内存区域的值为0
63. //对于th\_base\_lock变量，目前的值为NULL.
64. **if** (EVTHREAD\_LOCKING\_ENABLED() &&
65. (!cfg || !(cfg->flags & EVENT\_BASE\_FLAG\_NOLOCK))) { //配置是支持锁的
66. EVTHREAD\_ALLOC\_LOCK(base->th\_base\_lock,
67. EVTHREAD\_LOCKTYPE\_RECURSIVE); //申请一个锁
68. base->defer\_queue.lock = base->th\_base\_lock;
69. EVTHREAD\_ALLOC\_COND(base->current\_event\_cond);//申请一个条件变量
70. }
71. #endif
73. **return** (base);
74. }

        这里用到了event\_config结构体，关于这个结构体可以参考《[配置event\_base》](http://blog.csdn.net/luotuo44/article/details/38443569)一文。这个结构体主要是对event\_base进行一些配置。另外代码中还讲到了怎么使用选择一个多IO复用函数，这个可以参考《[跨平台Reactor接口的实现](http://blog.csdn.net/luotuo44/article/details/38458469)》一文。

        宏EVTHREAD\_LOCKING\_ENABLED主要是检测是否已经支持锁了。检测的方式也很简单，也就是检测\_evthread\_lock\_fns全局变量中的lock成员变量是否不为NULL。有关这个\_evthread\_lock\_fns全局变量可以查看《[多线程、锁、条件变量(一)](http://blog.csdn.net/luotuo44/article/details/38350633)》。

# 创建event：

        好了，现在event\_base已经新建出来了。下面看一下event\_new函数，它和前面的event\_base\_new一样，把主要是的初始化工作交给另一个函数。event\_new函数的工作只是创建一个struct event结构体，然后把它的参数原封不动地传给event\_assign，所以还是看event\_assign函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38501341) [copy](https://blog.csdn.net/luotuo44/article/details/38501341)

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. //进行一些赋值和初始化。
7. ev->ev\_base = base;
8. ev->ev\_callback = callback;
9. ev->ev\_arg = arg;
10. ev->ev\_fd = fd;
11. ev->ev\_events = events;
12. ev->ev\_res = 0;
13. ev->ev\_flags = EVLIST\_INIT; //初始化状态
14. ev->ev\_ncalls = 0;
15. ev->ev\_pncalls = NULL;
17. **if** (events & EV\_SIGNAL) {
18. **if** ((events & (EV\_READ|EV\_WRITE)) != 0) {
19. event\_warnx("%s: EV\_SIGNAL is not compatible with "
20. "EV\_READ or EV\_WRITE", \_\_func\_\_);
21. **return** -1;
22. }
23. }
25. ...
27. **return** 0;
28. }

        从event\_assign函数的名字可以得知它是进行赋值操作的。所以它能可以在event被初始化后再次调用。不过，初始化后再次调用的话，有些事情要注意。这个在后面的博客中会说到。

        从上面的代码可看到：如果这个event是用来监听一个信号的，那么就不能让这个event监听读或者写事件。原因是其与信号event的实现方法相抵触，具体可以参考《[信号event的处理](http://blog.csdn.net/luotuo44/article/details/38538991)》。

        注意，此时event结构体的变量ev\_flags的值是EVLIST\_INIT。对变量的追踪是很有帮助的。它指明了event结构体的状态。它通过以或运算的方式取下面的值：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38501341) [copy](https://blog.csdn.net/luotuo44/article/details/38501341)

1. //event\_struct.h文件
2. #define EVLIST\_TIMEOUT  0x01 //event从属于定时器队列或者时间堆
3. #define EVLIST\_INSERTED 0x02 //event从属于注册队列
4. #define EVLIST\_SIGNAL   0x04 //没有使用
5. #define EVLIST\_ACTIVE   0x08 //event从属于活动队列
6. #define EVLIST\_INTERNAL 0x10 //该event是内部使用的。信号处理时有用到
7. #define EVLIST\_INIT 0x80 //event已经被初始化了
9. /\* EVLIST\_X\_ Private space: 0x1000-0xf000 \*/
10. #define EVLIST\_ALL  (0xf000 | 0x9f) //所有标志。这个不能取

# 将event加入到event\_base中：

        创建完一个event结构体后，现在看一下event\_add。它同前面的函数一样，内部也是调用其他函数完成工作。因为它用到了锁，所以给出它的代码

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38501341) [copy](https://blog.csdn.net/luotuo44/article/details/38501341)

1. //event.c文件
2. **int**
3. event\_add(**struct** event \*ev, **const** **struct** timeval \*tv)
4. {
5. **int** res;
7. //加锁
8. EVBASE\_ACQUIRE\_LOCK(ev->ev\_base, th\_base\_lock);
9. res = event\_add\_internal(ev, tv, 0);
10. //解锁
11. EVBASE\_RELEASE\_LOCK(ev->ev\_base, th\_base\_lock);
12. **return** (res);
13. }
15. **static** **inline** **int**
16. event\_add\_internal(**struct** event \*ev, **const** **struct** timeval \*tv,
17. **int** tv\_is\_absolute)
18. {
19. **struct** event\_base \*base = ev->ev\_base;
20. **int** res = 0;
21. **int** notify = 0;
22. ...
23. **if** ((ev->ev\_events & (EV\_READ|EV\_WRITE|EV\_SIGNAL)) &&
24. !(ev->ev\_flags & (EVLIST\_INSERTED|EVLIST\_ACTIVE))) {
25. **if** (ev->ev\_events & (EV\_READ|EV\_WRITE))
26. res = evmap\_io\_add(base, ev->ev\_fd, ev); //加入io队列
27. **else** **if** (ev->ev\_events & EV\_SIGNAL)
28. res = evmap\_signal\_add(base, (**int**)ev->ev\_fd, ev);//加入信号队列
29. **if** (res != -1)
30. event\_queue\_insert(base, ev, EVLIST\_INSERTED);//向event\_base注册事件
31. }
32. ...
33. **return** (res);
34. }

        event\_add函数只是对event\_base加了锁，然后调用event\_add\_internal函数完成工作。所以函数event\_add是线程安全的。

        event\_add\_internal函数会调用前几篇博文讲到的evmap\_io\_add和evmap\_signal\_add，把有相同文件描述符fd和信号值sig的event连在一个队列里面。成功之后，就会调用event\_queue\_insert，向event\_base注册事件。

        前面博文的evmap\_io\_add和evmap\_signal\_add函数内部还有一些地方并没有说到。那就是把要监听的fd或者sig添加到多路IO复用函数中，使得其是可以监听的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38501341) [copy](https://blog.csdn.net/luotuo44/article/details/38501341)

1. //evmap.c文件
2. **int**
3. evmap\_io\_add(**struct** event\_base \*base, evutil\_socket\_t fd, **struct** event \*ev)
4. {
5. **const** **struct** eventop \*evsel = base->evsel;
6. **struct** event\_io\_map \*io = &base->io;
7. **struct** evmap\_io \*ctx = NULL;
8. **int** nread, nwrite, retval = 0;
9. **short** res = 0, old = 0;
10. **struct** event \*old\_ev;
12. ...
14. //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
15. //宏的展开，可以到http://blog.csdn.net/luotuo44/article/details/38403241查看
16. GET\_IO\_SLOT\_AND\_CTOR(ctx, io, fd, evmap\_io, evmap\_io\_init,
17. evsel->fdinfo\_len);
19. //同一个fd可以调用event\_new,event\_add
20. //多次。nread、nwrite就是记录有多少次。如果每次event\_new的回调函数
21. //都不一样，那么当fd有可读或者可写时，这些回调函数都是会触发的。
22. //对一个fd不能event\_new、event\_add太多次的。后面会进行判断
23. nread = ctx->nread;
24. nwrite = ctx->nwrite;
26. **if** (nread)
27. old |= EV\_READ;
28. **if** (nwrite)
29. old |= EV\_WRITE;
31. **if** (ev->ev\_events & EV\_READ) {
32. //记录是不是第一次。如果是第一次，那么就说明该fd还没被
33. //加入到多路IO复用中。即还没被加入到像select、epoll这些
34. //函数中。那么就要加入。这个在后面可以看到。
35. **if** (++nread == 1)
36. res |= EV\_READ;
37. }
38. **if** (ev->ev\_events & EV\_WRITE) {
39. **if** (++nwrite == 1)
40. res |= EV\_WRITE;
41. }
42. **if** (EVUTIL\_UNLIKELY(nread > 0xffff || nwrite > 0xffff)) {
43. event\_warnx("Too many events reading or writing on fd %d",
44. (**int**)fd);
45. **return** -1;
46. }

49. //把fd加入到多路IO复用中。
50. **if** (res) {
51. **void** \*extra = ((**char**\*)ctx) + **sizeof**(**struct** evmap\_io);
52. **if** (evsel->add(base, ev->ev\_fd,
53. old, (ev->ev\_events & EV\_ET) | res, extra) == -1)
54. **return** (-1);
55. retval = 1;
56. }
58. //nread进行了++。把次数记录下来。下次对于同一个fd，这个次数就有用了
59. ctx->nread = (ev\_uint16\_t) nread;
60. ctx->nwrite = (ev\_uint16\_t) nwrite;
62. TAILQ\_INSERT\_TAIL(&ctx->events, ev, ev\_io\_next);
64. **return** (retval);
65. }

        代码中有两个计数nread和nwrite，当其值为1时，就说明是第一次监听对应的事件。此时，就要把这个fd添加到多路IO复用函数中。这就完成fd与select、poll、epoll之类的多路IO复用函数的相关联。这完成对fd监听的第一步。

        下面再看event\_queue\_insert函数的实现。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38501341) [copy](https://blog.csdn.net/luotuo44/article/details/38501341)

1. //event.c文件
2. **static** **void**
3. event\_queue\_insert(**struct** event\_base \*base, **struct** event \*ev, **int** queue)
4. {
5. ...
7. ev->ev\_flags |= queue;
8. **switch** (queue) {
9. **case** EVLIST\_INSERTED:
10. TAILQ\_INSERT\_TAIL(&base->eventqueue, ev, ev\_next);
11. **break**;
12. ...
13. }
14. }

        这个函数的主要作为是把event加入到对应的队列中。在这里，是为了把event加入到eventqueue这个已注册队列中，即将event向event\_base注册。注意，此时event结构体的ev\_flags变量为EVLIST\_INIT | EVLIST\_INSERTED了。

# 进入主循环，开始监听event：

        现在事件已经添加完毕，开始进入主循环event\_base\_dispatch函数。还是同样，该函数内部调用event\_base\_loop完成工作。

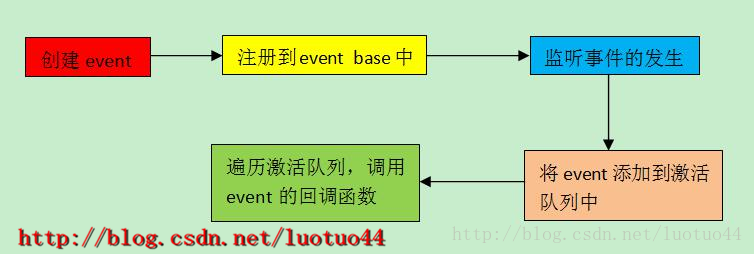
**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38501341) [copy](https://blog.csdn.net/luotuo44/article/details/38501341)

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. **int** res, done, retval = 0;
8. //加锁
9. EVBASE\_ACQUIRE\_LOCK(base, th\_base\_lock);
11. done = 0;
13. **while** (!done) {
14. //该函数的内部会解锁，然后调用OS提供的的多路IO复用函数。
15. //这个函数退出后，又会立即加锁。这有点像条件变量。
16. res = evsel->dispatch(base, tv\_p);
18. **if** (N\_ACTIVE\_CALLBACKS(base)) {
19. **int** n = event\_process\_active(base);
20. }
21. }
23. done:
24. //解锁
25. EVBASE\_RELEASE\_LOCK(base, th\_base\_lock);
26. **return** (retval);
27. }

        在event\_base\_loop函数内部会进行加锁，这是因为这里要对event\_base里面的多个队列进行一些数据操作(增删操作)，此时要用锁来保护队列不被另外一个线程所破坏。

        上面代码中有两个函数evsel->dispatch和event\_process\_active。前一个将调用多路IO复用函数，对event进行监听，并且把满足条件的event放到event\_base的激活队列中。后一个则遍历这个激活队列的所有event，逐个调用对应的回调函数。

        可以看到整个流程如下图所示：



# 将已激活event插入到激活列表：

        我们还是深入看看Libevent是怎么把event添加到激活队列的。dispatch是一个函数指针，它的实现都包含是一个多路IO复用函数。这里选择poll这个多路IO复用函数来作分析。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38501341) [copy](https://blog.csdn.net/luotuo44/article/details/38501341)

1. //poll.c文件
2. **static** **int**
3. poll\_dispatch(**struct** event\_base \*base, **struct** timeval \*tv)
4. {
5. **int** res, i, j, nfds;
6. **long** msec = -1;
7. **struct** pollop \*pop = base->evbase;
8. **struct** pollfd \*event\_set;
10. nfds = pop->nfds;
12. event\_set = pop->event\_set;
14. //解锁
15. EVBASE\_RELEASE\_LOCK(base, th\_base\_lock);
16. res = poll(event\_set, nfds, msec);
17. //再次加锁
18. EVBASE\_ACQUIRE\_LOCK(base, th\_base\_lock);
20. ...
22. i = random() % nfds;
23. **for** (j = 0; j < nfds; j++) {
24. **int** what;
25. **if** (++i == nfds)
26. i = 0;
27. what = event\_set[i].revents;
28. **if** (!what)
29. **continue**;
31. res = 0;
33. //如果fd发生错误，就把之当作读和写事件。之后调用read
34. //或者write时，就能得知具体是什么错误了。这里的作用是
35. //通知到上层。
36. **if** (what & (POLLHUP|POLLERR))
37. what |= POLLIN|POLLOUT;
39. **if** (what & POLLIN)
40. res |= EV\_READ;
41. **if** (what & POLLOUT)
42. res |= EV\_WRITE;
43. **if** (res == 0)
44. **continue**;
46. //把这个ev放到激活队列中。
47. evmap\_io\_active(base, event\_set[i].fd, res);
48. }
50. **return** (0);
51. }

        pollfd数组的数据是在evmap\_io\_add函数中添加的，在evmap\_io\_add函数里面，有一个evsel->add调用，它会把数据(fd和对应的监听类型)放到pollfd数组中。

        当主线程从poll返回时，没有错误的话，就说明有些监听的事件发生了。在函数的后面，它会遍历这个pollfd数组，查看哪个fd是有事件发生。如果事件发生，就调用evmap\_io\_active(base, event\_set[i].fd, res);在这个函数里面会把这个fd对应的event放到event\_base的激活event队列中。下面是evmap\_io\_active的代码。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38501341) [copy](https://blog.csdn.net/luotuo44/article/details/38501341)

1. **void** //evmap.c文件
2. evmap\_io\_active(**struct** event\_base \*base, evutil\_socket\_t fd, **short** events)
3. {
4. **struct** event\_io\_map \*io = &base->io;
5. **struct** evmap\_io \*ctx;
6. **struct** event \*ev;
8. //由这个fd找到对应event\_map\_entry的TAILQ\_HEAD.
9. GET\_IO\_SLOT(ctx, io, fd, evmap\_io);
11. //遍历这个队列。将所有与fd相关联的event结构体都处理一遍
12. TAILQ\_FOREACH(ev, &ctx->events, ev\_io\_next) {
13. **if** (ev->ev\_events & events)
14. event\_active\_nolock(ev, ev->ev\_events & events, 1);
15. }
16. }
18. **void** //event.c文件
19. event\_active\_nolock(**struct** event \*ev, **int** res, **short** ncalls)
20. {
21. **struct** event\_base \*base;
22. base = ev->ev\_base;
24. ...
25. //将ev插入到激活队列
26. event\_queue\_insert(base, ev, EVLIST\_ACTIVE);
28. ...
29. }

32. //将event 插入到event\_base的对应(由queue指定)的队列里面
33. **static** **void** //event.c文件
34. event\_queue\_insert(**struct** event\_base \*base, **struct** event \*ev, **int** queue)
35. {
36. ...
38. ev->ev\_flags |= queue;
39. **switch** (queue) {
40. **case** EVLIST\_ACTIVE:
41. base->event\_count\_active++;
42. //将event插入到对应对应优先级的激活队列中
43. TAILQ\_INSERT\_TAIL(&base->activequeues[ev->ev\_pri],
44. ev,ev\_active\_next);
45. **break**;
46. }
47. }

        经过上面三个函数的调用，就可以把一个fd对应的所有符合条件的event插入到激活队列中。因为Libevent还对事件处理设有优先级，所以有一个激活数组队列，而不是只有一个激活队列。

        注意，此时event结构体的ev\_flags变量为EVLIST\_INIT | EVLIST\_INSERTED | EVLIST\_ACTIVE了。

# 处理激活列表中的event：

        现在已经完成了将event插入到激活队列中。接下来就是遍历激活数组队列，把所有激活的event都处理即可。

        现在来追踪event\_process\_active函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38501341) [copy](https://blog.csdn.net/luotuo44/article/details/38501341)

1. //event.c文件
2. **static** **int**
3. event\_process\_active(**struct** event\_base \*base)
4. {
5. **struct** event\_list \*activeq = NULL;
6. **int** i, c = 0;
8. //从高优先级到低优先级遍历优先级数组
9. **for** (i = 0; i < base->nactivequeues; ++i) {
10. //对于特定的优先级，遍历该优先级的所有激活event
11. **if** (TAILQ\_FIRST(&base->activequeues[i]) != NULL) {
12. activeq = &base->activequeues[i];
13. c = event\_process\_active\_single\_queue(base, activeq);
14. ...
15. }
16. }
17. **return** c;
18. }
20. **static** **int**
21. event\_process\_active\_single\_queue(**struct** event\_base \*base,
22. **struct** event\_list \*activeq)
23. {
24. **struct** event \*ev;
25. **int** count = 0;
27. **for** (ev = TAILQ\_FIRST(activeq); ev; ev = TAILQ\_FIRST(activeq)) {
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);
33. **if** (!(ev->ev\_flags & EVLIST\_INTERNAL))
34. ++count;
36. //下面开始处理这个event
37. **switch** (ev->ev\_closure) {
38. ...
39. **case** EV\_CLOSURE\_NONE:
40. //调用用户设置的回调函数。
41. (\*ev->ev\_callback)(ev->ev\_fd, ev->ev\_res, ev->ev\_arg);
42. **break**;
43. }
45. EVBASE\_ACQUIRE\_LOCK(base, th\_base\_lock);
47. }
48. **return** count;
49. }

        上面的代码，从高到低优先级遍历激活event优先级数组。对于激活的event，要调用event\_queue\_remove将之从激活队列中删除掉。然后再对这个event调用其回调函数。

        event\_queue\_remove函数的调用会改变event结构体的ev\_flags变量的值。调用后， ev\_flags变量为EVLIST\_INIT | EVLIST\_INSERTED。现在又可以等待下一次事件的到来了。

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/38501341