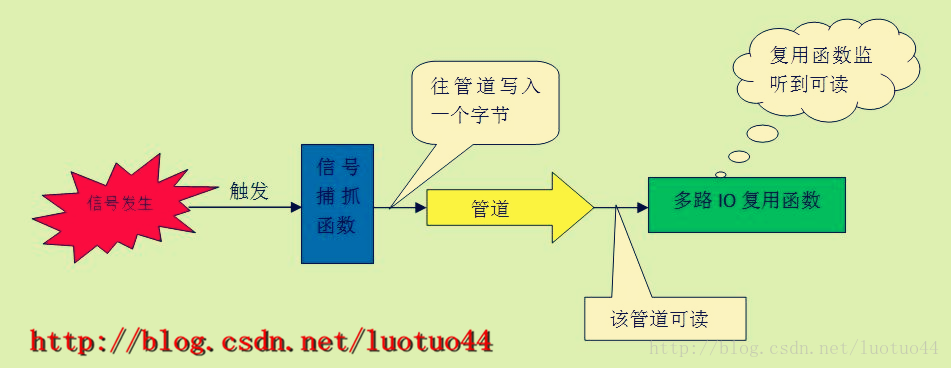
<https://blog.csdn.net/luotuo44/article/details/38538991>

**转载请注明出处：**[**http://blog.csdn.net/luotuo44/article/details/38538991**](http://blog.csdn.net/luotuo44/article/details/38538991)

# 信号event的工作原理：

        前面讲解了Libevent如何监听一个IO事件，现在来讲一下Libevent如何监听信号。Libevent对于信号的处理是采用统一事件源的方式。简单地说，就是把信号也转换成IO事件，集成到Libevent中。

        统一事件源的工作原理如下：假如用户要监听SIGINT信号，那么在实现的内部就对SIGINT这个信号设置捕抓函数。此外，在实现的内部还要建立一条管道(pipe)，并把这个管道加入到多路IO复用函数中。当SIGINT这个信号发生后，捕抓函数将会被调用。而这个捕抓函数的工作就是往管道写入一个字符(这个字符往往等于所捕抓到信号的信号值)。此时，这个管道就变成是可读的了，多路IO复用函数能检测到这个管道变成可读的了。换言之，多路IO复用函数检测到SIGINT信号的发生，也就完成了对信号的监听工作。这个过程如下图所示：



        了解完统一事件源的工作原理，现在来看一下Libevent具体的实现细节。按照上述的介绍，内部实现的工作有：

1. 创建一个管道(Libevent实际上使用的是socketpair)
2. 为这个socketpair的一个读端创建一个event，并将之加入到多路IO复用函数的监听之中
3. 设置信号捕抓函数
4. 有信号发生，就往socketpair写入一个字节

        统一事件源能够工作的一个原因是：多路IO复用函数都是可中断的。即处理完信号后，会从多路IO复用函数中退出，并将errno赋值为EINTR。有些OS的某些系统调用，比如Linux的read，即使被信号终端了，还是会自启动的。即不会从read函数中退出来。

# 用于信号event的结构体和变量：

        event\_base为信号监听提供了的成员如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

1. //event-internal.h文件
2. **struct** event\_base {
4. **const** **struct** eventop \*evsigsel;
5. **struct** evsig\_info sig;
7. ...
8. **struct** event\_signal\_map sigmap;
9. ...
10. };

13. //evsignal-internal.h文件
14. **struct** evsig\_info {
15. //用于监听socketpair读端的event. ev\_signal\_pair[1]为读端
16. **struct** event ev\_signal;
17. //socketpair
18. evutil\_socket\_t ev\_signal\_pair[2];
19. //用来标志是否已经将ev\_signal这个event加入到event\_base中了
20. **int** ev\_signal\_added;
21. //用户一共要监听多少个信号
22. **int** ev\_n\_signals\_added;
24. //数组。用户可能已经设置过某个信号的信号捕抓函数。但
25. //Libevent还是要为这个信号设置另外一个信号捕抓函数，
26. //此时，就要保存用户之前设置的信号捕抓函数。当用户不要
27. //监听这个信号时，就能够恢复用户之前的捕抓函数。
28. //因为是有多个信号，所以得用一个数组保存。
29. #ifdef \_EVENT\_HAVE\_SIGACTION
30. **struct** sigaction \*\*sh\_old;
31. #else//保存的是捕抓函数的函数指针，又因为是数组。所以是二级指针
32. ev\_sighandler\_t \*\*sh\_old;
33. #endif
34. /\* Size of sh\_old. \*/
35. **int** sh\_old\_max; //数组的长度
36. };

        在上面代码中，已经可以看到用于socketpair的ev\_signal\_pair变量，还有struct event结构体变量ev\_signal。那么Libevent是在何时创建socketpair以及如何将socketpair和ev\_signal相关联的呢？

# 初始化：

        在前面的博文《[跨平台Reactor接口的实现](http://blog.csdn.net/luotuo44/article/details/38458469#t3)》中，介绍了Libevent是如何选择一个多路IO复用函数的。在选定一个多路IO复用函数后，就会调用下面一行代码。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

1. base->evbase = base->evsel->init(base);

        这是初始化代码函数。下面给出poll的init函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

1. //poll.c文件
2. **static** **void** \*
3. poll\_init(**struct** event\_base \*base)
4. {
5. **struct** pollop \*pollop;
7. **if** (!(pollop = mm\_calloc(1, **sizeof**(**struct** pollop))))
8. **return** (NULL);
10. evsig\_init(base);
12. **return** (pollop);
13. }

        可以看到，其调用了evsig\_init函数。而正是这个evsig\_init函数完成了创建socketpair并将socketpair的一个读端与ev\_signal相关联。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

1. //signal.c文件
2. **int**
3. evsig\_init(**struct** event\_base \*base)
4. {
5. //创建一个socketpair
6. **if** (evutil\_socketpair(
7. AF\_UNIX, SOCK\_STREAM, 0, base->sig.ev\_signal\_pair) == -1) {
8. #ifdef WIN32
9. /\* Make this nonfatal on win32, where sometimes people
10. have localhost firewalled. \*/
11. event\_sock\_warn(-1, "%s: socketpair", \_\_func\_\_);
12. #else
13. event\_sock\_err(1, -1, "%s: socketpair", \_\_func\_\_);
14. #endif
15. **return** -1;
16. }
18. //子进程不能访问该socketpair
19. evutil\_make\_socket\_closeonexec(base->sig.ev\_signal\_pair[0]);
20. evutil\_make\_socket\_closeonexec(base->sig.ev\_signal\_pair[1]);
21. base->sig.sh\_old = NULL;
22. base->sig.sh\_old\_max = 0;

25. evutil\_make\_socket\_nonblocking(base->sig.ev\_signal\_pair[0]);
26. evutil\_make\_socket\_nonblocking(base->sig.ev\_signal\_pair[1]);
28. //将ev\_signal\_pair[1]与ev\_signal这个event相关联。ev\_signal\_pair[1]为读端
29. //该函数的作用等同于event\_new。实际上event\_new内部也是调用event\_assign函数完成工作的
30. event\_assign(&base->sig.ev\_signal, base, base->sig.ev\_signal\_pair[1],
31. EV\_READ | EV\_PERSIST, evsig\_cb, base);
33. //标明是内部使用的
34. base->sig.ev\_signal.ev\_flags |= EVLIST\_INTERNAL;
35. //Libevent中，event是有优先级的。前一篇博文已经说到这一点
36. event\_priority\_set(&base->sig.ev\_signal, 0); //最高优先级
38. base->evsigsel = &evsigops;
40. **return** 0;
41. }

        socketpair的两个端都调用evutil\_make\_socket\_closeonexec，因为不能让子进程可以访问的这个socketpair。因为子进程的访问可能会出现扰乱。比如，子进程往socketpair发送信息，使得父进程的多路IO复用函数误以为信号发生了；父进程确实发生了信号，也往socketpair发送了一个字节，但却被子进程接收了这个字节。父进程没有监听到可读。

        在Windows中，并没有直接的可以使用的socketpair API。此时，Libevent就自己实现了一个socketpair。具体可以参考《[通用类型和函数](http://blog.csdn.net/luotuo44/article/details/38780157#t9)》。

        在函数的最后可以看到event\_base的一个成员evsignal被赋值。evsignal是一个IO复用结构体，而evsigops是专门用于信号处理的[IO复用结构体](http://blog.csdn.net/luotuo44/article/details/38458469)变量。定义如下：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

1. //signal.c文件
2. **static** **const** **struct** eventop evsigops = {
3. "signal",
4. NULL,
5. evsig\_add,
6. evsig\_del,
7. NULL,
8. NULL,
9. 0, 0, 0
10. };

        该结构体只有evsig\_add和evsig\_del这两个函数指针。实际在工作时有这两个函数就足够了。

# 将信号event加入到event\_base:

        前面的代码已经完成了“创建socketpair并将socketpair的一个读端于ev\_signal相关联”。接下来看其他的工作。假如要对一个绑定了某个信号的event调用event\_add函数，那么在event\_add的内部会调用event\_add\_internal函数。而event\_add\_internal函数又会调用evmap\_signal\_add函数。如果看了之前的博文，应该对这个流程不陌生。下面看看evmap\_signal\_add函数：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

1. //evmap.c文件
2. **int**
3. evmap\_signal\_add(**struct** event\_base \*base, **int** sig, **struct** event \*ev)
4. {
5. //注意这里调用的是base的evsigsel变量。而不是evsel。
6. **const** **struct** eventop \*evsel = base->evsigsel;
7. **struct** event\_signal\_map \*map = &base->sigmap;
9. ...
11. **if** (TAILQ\_EMPTY(&ctx->events)) {
12. //实际调用的是evsig\_add函数
13. **if** (evsel->add(base, ev->ev\_fd, 0, EV\_SIGNAL, NULL)
14. == -1)
15. **return** (-1);
16. }
18. **return** (1);
19. }

        上面函数的内部调用了IO复用结构体的add函数指针，即调用了evsig\_add。现在我们深入evsig\_add函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

1. /signal.c文件
2. **static** **int**
3. evsig\_add(**struct** event\_base \*base, evutil\_socket\_t evsignal, **short** old, **short** events, **void** \*p)
4. {
5. **struct** evsig\_info \*sig = &base->sig;
6. (**void**)p;
8. //NSIG是信号的个数。定义在系统头文件中
9. EVUTIL\_ASSERT(evsignal >= 0 && evsignal < NSIG);
11. /\* catch signals if they happen quickly \*/
12. //加锁保护。但实际其锁变量为NULL。所以并没有保护。应该会在以后的版本有所改正
13. //在2.1.4-alpha版本中，就已经改进了这个问题。为锁变量分配了锁
14. EVSIGBASE\_LOCK();
15. //如果有多个event\_base，那么捕抓信号这个工作只能由其中一个完成。
16. **if** (evsig\_base != base && evsig\_base\_n\_signals\_added) {
17. event\_warnx("Added a signal to event base %p with signals "
18. "already added to event\_base %p.  Only one can have "
19. "signals at a time with the %s backend.  The base with "
20. "the most recently added signal or the most recent "
21. "event\_base\_loop() call gets preference; do "
22. "not rely on this behavior in future Libevent versions.",
23. base, evsig\_base, base->evsel->name);
24. }
25. evsig\_base = base;
26. evsig\_base\_n\_signals\_added = ++sig->ev\_n\_signals\_added;
27. evsig\_base\_fd = base->sig.ev\_signal\_pair[0]; //写端。0是写端，这确实与之前所接触到的有所不同
28. EVSIGBASE\_UNLOCK();
30. //设置Libevent的信号捕抓函数
31. **if** (\_evsig\_set\_handler(base, (**int**)evsignal, evsig\_handler) == -1) {
32. **goto** err;
33. }
35. //event\_base第一次监听信号事件。要添加ev\_signal到event\_base中
36. **if** (!sig->ev\_signal\_added) {
37. //注意，本函数的调用路径为event\_add->event\_add\_internal->evmap\_signal\_map->evsig\_add
38. //所以这里是递归调用event\_add函数。而event\_add函数是会加锁的。所以需要锁为递归锁
39. **if** (event\_add(&sig->ev\_signal, NULL))//添加一个内部的event
40. **goto** err;
41. sig->ev\_signal\_added = 1;
42. }
44. **return** (0);
46. err:
47. EVSIGBASE\_LOCK();
48. --evsig\_base\_n\_signals\_added;
49. --sig->ev\_n\_signals\_added;
50. EVSIGBASE\_UNLOCK();
51. **return** (-1);
52. }

        从后面的那个if语句可以得知，当sig->ev\_signal\_added变量为0时(即用户第一次监听一个信号)，就会将ev\_signal这个event加入到event\_base中。从前面的“统一事件源”可以得知，这个ev\_signal的作用就是通知event\_base，有信号发生了。只需一个event即可完成工作，即使用户要监听多个不同的信号，因为这个event已经和socketpair的读端相关联了。如果要监听多个信号，那么就在信号处理函数中往这个socketpair写入不同的值即可。event\_base能监听到可读，并可以从读到的内容可以判断是哪个信号发生了。

       从代码中也可得知，Libevent并不会为每一个信号监听创建一个event。它只会创建一个全局的专门用于监听信号的event。这个也是“统一事件源”的工作原理。

# 设置信号捕抓函数：

       evsig\_add函数还调用了\_evsig\_set\_handler函数完成设置Libevent内部的信号捕抓函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

1. //signal.c文件
2. **typedef** **void** (\*ev\_sighandler\_t)(**int**);
4. //evsignal是信号值，handler是信号捕抓函数
5. **int**
6. \_evsig\_set\_handler(**struct** event\_base \*base,
7. **int** evsignal, **void** (\_\_cdecl \*handler)(**int**))
8. {
9. //如果有sigaction就优先使用之
10. #ifdef \_EVENT\_HAVE\_SIGACTION
11. **struct** sigaction sa;
12. #else
13. ev\_sighandler\_t sh;
14. #endif
15. **struct** evsig\_info \*sig = &base->sig;
16. **void** \*p;

19. //数组的一个元素就存放一个信号。信号值等于其下标
20. **if** (evsignal >= sig->sh\_old\_max) { //不够内存。重新分配
21. **int** new\_max = evsignal + 1;
22. event\_debug(("%s: evsignal (%d) >= sh\_old\_max (%d), resizing",
23. \_\_func\_\_, evsignal, sig->sh\_old\_max));
24. p = mm\_realloc(sig->sh\_old, new\_max \* **sizeof**(\*sig->sh\_old));
25. **if** (p == NULL) {
26. event\_warn("realloc");
27. **return** (-1);
28. }
30. memset((**char** \*)p + sig->sh\_old\_max \* **sizeof**(\*sig->sh\_old),
31. 0, (new\_max - sig->sh\_old\_max) \* **sizeof**(\*sig->sh\_old));
33. sig->sh\_old\_max = new\_max;
34. sig->sh\_old = p;
35. }
37. //注意sh\_old是一个二级指针。元素是一个一级指针。为这个一级指针分配内存
38. /\* allocate space for previous handler out of dynamic array \*/
39. sig->sh\_old[evsignal] = mm\_malloc(**sizeof** \*sig->sh\_old[evsignal]);
40. **if** (sig->sh\_old[evsignal] == NULL) {
41. event\_warn("malloc");
42. **return** (-1);
43. }
45. /\* save previous handler and setup new handler \*/
46. #ifdef \_EVENT\_HAVE\_SIGACTION
47. memset(&sa, 0, **sizeof**(sa));
48. sa.sa\_handler = handler;
49. sa.sa\_flags |= SA\_RESTART;
50. sigfillset(&sa.sa\_mask);
52. //设置信号处理函数
53. **if** (sigaction(evsignal, &sa, sig->sh\_old[evsignal]) == -1) {
54. event\_warn("sigaction");
55. mm\_free(sig->sh\_old[evsignal]);
56. sig->sh\_old[evsignal] = NULL;
57. **return** (-1);
58. }
59. #else
60. //设置信号处理函数
61. **if** ((sh = signal(evsignal, handler)) == SIG\_ERR) {
62. event\_warn("signal");
63. mm\_free(sig->sh\_old[evsignal]);
64. sig->sh\_old[evsignal] = NULL;
65. **return** (-1);
66. }
67. //保存之前的信号捕抓函数。当用户event\_del这个信号监听后，就可以恢复了
68. \*sig->sh\_old[evsignal] = sh;
69. #endif
71. **return** (0);
72. }

        如果看过《UNIX环境高级编程》信号那章的话，上面这段代码很容易看懂。这里就不讲了。

        这里我们做一个猜测：当我们对某个信号进行event\_new和event\_add后，就不应该再次设置该信号的信号捕抓函数。否则event\_base将无法监听到信号的发生。下面代码验证这猜测。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

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

7. **void** sig\_cb(**int** fd, **short** events, **void** \*arg)
8. {
9. printf("in the sig\_cb\n");
10. }
12. **void** signal\_handle(**int** sig)
13. {
14. printf("catch the sig %d\n", sig);
15. }
17. **int** main()
18. {
20. **struct** event\_base \*base = event\_base\_new();
22. **struct** event \*ev = evsignal\_new(base, SIGUSR1, sig\_cb, NULL);
23. event\_add(ev, NULL);
25. signal(SIGUSR1, signal\_handle);
27. printf("pid = %d\n", getpid());
29. printf("begin\n");
30. event\_base\_dispatch(base);
31. printf("end\n");
33. **return** 0;
34. }

        运行上面代码, 通过在外部给这个进程发生信号的方式。可以看到，event\_base确实无法监听到信号了。所有信号都被signal\_handle捕抓了。

# 捕抓信号：

        前面的代码中有两个函数并没有讲，分别是信号捕抓函数evsig\_handler和调用event\_assign时的信号回调函数evsig\_cb。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

1. //signal.c文件
2. **static** **void** \_\_cdecl
3. evsig\_handler(**int** sig)
4. {
5. ...
6. ev\_uint8\_t msg;
8. **if** (evsig\_base == NULL) {
9. event\_warnx(
10. "%s: received signal %d, but have no base configured",
11. \_\_func\_\_, sig);
12. **return**;
13. }
15. #ifndef \_EVENT\_HAVE\_SIGACTION
16. //这主要是为了应对旧时代的信号不可靠
17. //现在的OS并不会出现这个问题
18. signal(sig, evsig\_handler);
19. #endif
21. /\* Wake up our notification mechanism \*/
22. msg = sig;
23. send(evsig\_base\_fd, (**char**\*)&msg, 1, 0); //向socketpair写入一个字节
25. ...
26. }

        从evsig\_handler函数的实现可以看到，实现得相当简单。只是将信号对应的值写入到socketpair中。evsig\_base\_fd是socketpair的写端，这是一个全局变量，在evsig\_add函数中被赋值的。

        从“统一事件源”的工作原理来看，现在已经完成了对信号的捕抓，已经将该信号的当作IO事件写入到socketpair中了。现在event\_base应该已经监听到socketpair可读了，并且会为调用回调函数evsig\_cb了。下面看看evsig\_cb函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

1. //signal.c文件
2. **static** **void**
3. evsig\_cb(evutil\_socket\_t fd, **short** what, **void** \*arg)
4. {
5. **static** **char** signals[1024];
6. ev\_ssize\_t n;
7. **int** i;
9. //NSIG是信号的个数
10. **int** ncaught[NSIG];
11. **struct** event\_base \*base;
13. base = arg;
15. memset(&ncaught, 0, **sizeof**(ncaught));
17. **while** (1) {
18. //读取socketpair中的数据。从中可以知道有哪些信号发生了
19. //已经socketpair的读端已经设置为非阻塞的。所以不会被阻塞在
20. //recv函数中。这个循环要把socketpair的所有数据都读取出来
21. n = recv(fd, signals, **sizeof**(signals), 0);
22. **if** (n == -1) {
23. **int** err = evutil\_socket\_geterror(fd);
24. **if** (! EVUTIL\_ERR\_RW\_RETRIABLE(err))
25. event\_sock\_err(1, fd, "%s: recv", \_\_func\_\_);
26. **break**;
27. } **else** **if** (n == 0) {
28. /\* XXX warn? \*/
29. **break**;
30. }
32. //遍历数据数组，把每一个字节当作一个信号
33. **for** (i = 0; i < n; ++i) {
34. ev\_uint8\_t sig = signals[i];
35. **if** (sig < NSIG)
36. ncaught[sig]++; //该信号发生的次数
37. }
38. }
40. EVBASE\_ACQUIRE\_LOCK(base, th\_base\_lock);
41. **for** (i = 0; i < NSIG; ++i) {
42. **if** (ncaught[i]) //有信号发生就为之调用evmap\_signal\_active
43. evmap\_signal\_active(base, i, ncaught[i]);
44. }
45. EVBASE\_RELEASE\_LOCK(base, th\_base\_lock);
46. }

        该回调函数的作用是读取socketpair的所有数据，并将数据当作信号，再根据信号值调用evmap\_signal\_active。

        有一点要注意，evsig\_cb这个回调函数并不是用户为监听一个信号调用event\_new时设置的用户回调函数，而是Libevent内部为了处理信号而设置的内部回调函数。累！！

# 激活信号event:

        虽然如此，但是现在的情况是：当有信号发生时，就会调用evmap\_signal\_active函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

1. //event-internal.h文件
2. #define ev\_signal\_next  \_ev.ev\_signal.ev\_signal\_next
4. #define ev\_ncalls   \_ev.ev\_signal.ev\_ncalls
5. #define ev\_pncalls  \_ev.ev\_signal.ev\_pncalls
7. //evmap.c文件
8. **void** //后两个参数分别是信号值和发生的次数。
9. evmap\_signal\_active(**struct** event\_base \*base, evutil\_socket\_t sig, **int** ncalls)
10. {
11. **struct** event\_signal\_map \*map = &base->sigmap;
12. **struct** evmap\_signal \*ctx;
13. **struct** event \*ev;
15. //通过这个fd找到对应的TAILQ\_HEAD
16. GET\_SIGNAL\_SLOT(ctx, map, sig, evmap\_signal);
18. //遍历该fd的队列
19. TAILQ\_FOREACH(ev, &ctx->events, ev\_signal\_next)
20. event\_active\_nolock(ev, EV\_SIGNAL, ncalls);
21. }

24. //event.c文件
25. **void**
26. event\_active\_nolock(**struct** event \*ev, **int** res, **short** ncalls)
27. {
28. **struct** event\_base \*base;

31. base = ev->ev\_base;
32. ev->ev\_res = res;
34. //这将停止处理低优先级的event。一路回退到event\_base\_loop中。
35. **if** (ev->ev\_pri < base->event\_running\_priority)
36. base->event\_continue = 1;
38. **if** (ev->ev\_events & EV\_SIGNAL) {
39. #ifndef \_EVENT\_DISABLE\_THREAD\_SUPPORT
40. **if** (base->current\_event == ev && !EVBASE\_IN\_THREAD(base)) {
41. ++base->current\_event\_waiters;
42. //由于此时是主线程执行，所以并不会进行这个判断里面
43. EVTHREAD\_COND\_WAIT(base->current\_event\_cond, base->th\_base\_lock);
44. }
45. #endif
46. ev->ev\_ncalls = ncalls;
47. ev->ev\_pncalls = NULL;
48. }
50. //插入到激活队列中.插入到队尾
51. event\_queue\_insert(base, ev, EVLIST\_ACTIVE);
53. }

        通过evmap\_signal\_active、event\_active\_nolock和event\_queue\_insert这三个函数的调用后，就可以把一个event插入到激活队列了。

        由于这些函数的执行本身就是在Libevent处理event的回调函数之中的(Libevent正在处理内部的信号处理event)。所以并不需要从event\_base\_loop里的while循环里面再次执行一次evsel->dispatch()，才能执行到这次信号event。即无需等到下一次处理激活队列，就可以执行该信号event了。分析如下：

        首先要明确，现在执行上面三个函数相当于在执行event的回调函数。所以其是运行在event\_process\_active函数之中的。为什么是在这里，可以参考《[Libevent工作流程探究](http://blog.csdn.net/luotuo44/article/details/38501341)》一文。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

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. base->event\_running\_priority = i;
13. activeq = &base->activequeues[i];
14. c = event\_process\_active\_single\_queue(base, activeq);
15. }
16. }
18. **return** c;
19. }
21. **static** **int**
22. event\_process\_active\_single\_queue(**struct** event\_base \*base,
23. **struct** event\_list \*activeq)
24. {
25. **struct** event \*ev;
27. //遍历该优先级的所有event，并处理之
28. **for** (ev = TAILQ\_FIRST(activeq); ev; ev = TAILQ\_FIRST(activeq)) {
30. ...//开始处理这个event。会调用event的回调函数
31. }
33. }

        从上面的代码可以看到，Libevent在处理内部的那个信号处理event的回调函数时，其实是在event\_process\_active\_single\_queue的一个循环里面。因为Libevent内部的信号处理event的优先级最高优先级，并且在前面的将用户信号event插入到队列(即event\_queue\_insert)，在插入到队列的尾部。所以无论用户的这个信号event的优先级是多少，都是在Libevent的内部信号处理event的后面。所以在遍历上面两个函数的里外两个循环时，肯定会执行到用户的信号event。

# 执行已激活信号event：

        现在看看Libevent是怎么处理已激活的信号event的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38538991) [copy](https://blog.csdn.net/luotuo44/article/details/38538991)

1. //event.c文件
2. **static** **inline** **void**
3. event\_signal\_closure(**struct** event\_base \*base, **struct** event \*ev)
4. {
5. **short** ncalls;
6. **int** should\_break;
8. /\* Allows deletes to work \*/
9. ncalls = ev->ev\_ncalls;
10. **if** (ncalls != 0)
11. ev->ev\_pncalls = &ncalls;
13. //while循环里面会调用用户设置的回调函数。该回调函数可能会执行很久
14. //所以要解锁先.
15. EVBASE\_RELEASE\_LOCK(base, th\_base\_lock);
16. //如果该信号发生了多次，那么就需要多次执行回调函数
17. **while** (ncalls) {
18. ncalls--;
19. ev->ev\_ncalls = ncalls;
20. **if** (ncalls == 0)
21. ev->ev\_pncalls = NULL;
22. (\*ev->ev\_callback)(ev->ev\_fd, ev->ev\_res, ev->ev\_arg);
24. EVBASE\_ACQUIRE\_LOCK(base, th\_base\_lock);
25. //其他线程调用event\_base\_loopbreak函数中断之
26. should\_break = base->event\_break;
27. EVBASE\_RELEASE\_LOCK(base, th\_base\_lock);
29. **if** (should\_break) {
30. **if** (ncalls != 0)
31. ev->ev\_pncalls = NULL;
32. **return**;
33. }
34. }
35. }

        可以看到，如果对应的信号发生了多次，那么该信号event的回调函数将被执行多次。

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/38538991