<https://blog.csdn.net/luotuo44/article/details/38556059>

**转载请注明出处:**[**http://blog.csdn.net/luotuo44/article/details/38556059**](http://blog.csdn.net/luotuo44/article/details/38556059)

        一般来说，是主线程执行event\_base\_dispatch函数。本文也是如此，如无特别说明，event\_base\_dispatch函数是由主线程执行的。

# notify的理由：

        本文要说明的问题是，当主线程在执行event\_base\_dispatch进入多路IO复用函数时，会处于休眠状态，休眠前解锁。此时，其他线程可能想往event\_base添加一个event，这个event可能是一般的IO event也可能是超时event。无论哪个，都需要及时告知主线程：有新的event要加进来。要实现这种功能就需要Libevent提供一种机制来提供唤醒主线程。

# 工作原理：

        Libevent提供的唤醒主线程机制也是挺简单的，其原理和《[信号event的处理](http://blog.csdn.net/luotuo44/article/details/38538991)》一文中提到的方法是一样的。提供一个内部的IO  event，专门用于唤醒主线程。当其他线程有event要add进来时，就往这个内部的IO event写入一个字节。此时，主线程在dispatch时，就能检测到可读，也就醒来了。这就完成了通知。这过程和Libevent处理信号event是一样的。

# 相关结构体：

        下面看一下Libevent的实现代码。同信号处理一样，先来看一下event\_base提供了什么成员。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38556059) [copy](https://blog.csdn.net/luotuo44/article/details/38556059)

1. **struct** event\_base {
2. …
3. //event\_base是否处于通知的未决状态。即次线程已经通知了，但主线程还没处理这个通知
4. **int** is\_notify\_pending;
5. evutil\_socket\_t th\_notify\_fd[2]; //通信管道
6. **struct** event th\_notify;//用于监听th\_notify\_fd的读端
7. //有两个可供选择的通知函数，指向其中一个通知函数
8. **int** (\*th\_notify\_fn)(**struct** event\_base \*base);
9. };

# 创建通知event并将之加入到event\_base：

        现在看Libevent怎么创建通信通道，以及怎么和event相关联。在event\_base\_new\_with\_config(event\_base\_new会调用该函数)里面会调用evthread\_make\_base\_notifiable函数，使得libevent变成可通知的。只有在已经支持多线程的情况下才会调用evthread\_make\_base\_notifiable函数的。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38556059) [copy](https://blog.csdn.net/luotuo44/article/details/38556059)

1. //event.c文件
2. **int**
3. evthread\_make\_base\_notifiable(**struct** event\_base \*base)
4. {
5. //默认event回调函数和默认的通知函数
6. **void** (\*cb)(evutil\_socket\_t, **short**, **void** \*) = evthread\_notify\_drain\_default;
7. **int** (\*notify)(**struct** event\_base \*) = evthread\_notify\_base\_default;
9. /\* XXXX grab the lock here? \*/
10. **if** (!base)
11. **return** -1;
13. //th\_notify\_fd[0]被初始化为-1,如果>=0,就说明已经被设置过了
14. **if** (base->th\_notify\_fd[0] >= 0)
15. **return** 0;
17. #if defined(\_EVENT\_HAVE\_EVENTFD) && defined(\_EVENT\_HAVE\_SYS\_EVENTFD\_H)
18. #ifndef EFD\_CLOEXEC
20. #define EFD\_CLOEXEC 0
21. #endif
22. //Libevent优先使用eventfd，但eventfd的通信机制和其他的不一样。所以
23. //要专门为eventfd创建通知函数和event回调函数
24. base->th\_notify\_fd[0] = eventfd(0, EFD\_CLOEXEC);
25. **if** (base->th\_notify\_fd[0] >= 0) {
26. evutil\_make\_socket\_closeonexec(base->th\_notify\_fd[0]);
27. notify = evthread\_notify\_base\_eventfd;
28. cb = evthread\_notify\_drain\_eventfd;
29. }
30. #endif
31. #if defined(\_EVENT\_HAVE\_PIPE)
32. //<0，说明之前的通知方式没有用上
33. **if** (base->th\_notify\_fd[0] < 0) {
34. //有些多路IO复用函数并不支持文件描述符。如果不支持，那么就不能使用这种
35. //通知方式。有关这个的讨论.查看http://blog.csdn.net/luotuo44/article/details/38443569
36. **if** ((base->evsel->features & EV\_FEATURE\_FDS)) {
37. **if** (pipe(base->th\_notify\_fd) < 0) {
38. event\_warn("%s: pipe", \_\_func\_\_);
39. } **else** {
40. evutil\_make\_socket\_closeonexec(base->th\_notify\_fd[0]);
41. evutil\_make\_socket\_closeonexec(base->th\_notify\_fd[1]);
42. }
43. }
44. }
45. #endif
47. #ifdef WIN32
48. #define LOCAL\_SOCKETPAIR\_AF AF\_INET
49. #else
50. #define LOCAL\_SOCKETPAIR\_AF AF\_UNIX
51. #endif
53. **if** (base->th\_notify\_fd[0] < 0) {
54. **if** (evutil\_socketpair(LOCAL\_SOCKETPAIR\_AF, SOCK\_STREAM, 0,
55. base->th\_notify\_fd) == -1) {
56. event\_sock\_warn(-1, "%s: socketpair", \_\_func\_\_);
57. **return** (-1);
58. } **else** {
59. evutil\_make\_socket\_closeonexec(base->th\_notify\_fd[0]);
60. evutil\_make\_socket\_closeonexec(base->th\_notify\_fd[1]);
61. }
62. }
64. //无论哪种通信机制，都要使得读端不能阻塞
65. evutil\_make\_socket\_nonblocking(base->th\_notify\_fd[0]);
67. //设置回调函数
68. base->th\_notify\_fn = notify;
70. //同样为了让写端不阻塞。虽然，如果同时出现大量需要notify的操作，会塞满通信通道。
71. //本次的notify会没有写入到通信通道中(已经变成非阻塞了)。但这无所谓，因为目的是
72. //唤醒主线程，通信通道有数据就肯定能唤醒。
73. **if** (base->th\_notify\_fd[1] > 0)
74. evutil\_make\_socket\_nonblocking(base->th\_notify\_fd[1]);
76. //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
77. //函数cb作为这个event的回调函数
78. event\_assign(&base->th\_notify, base, base->th\_notify\_fd[0],
79. EV\_READ|EV\_PERSIST, cb, base);
81. //标明是内部使用的
82. base->th\_notify.ev\_flags |= EVLIST\_INTERNAL;
83. event\_priority\_set(&base->th\_notify, 0); //最高优先级
85. **return** event\_add(&base->th\_notify, NULL);//加入到event\_base中。
86. }

        上面代码展示了，Libevent会从eventfd、pipe和socketpair中选择一种通信方式。由于有三种通信通道可供选择，下文为了方便叙述，就假定它选定的是pipe。

        上面代码的工作过程和普通的Libevent例子程序差不多，首先创建一个文件描述符fd，然后用这个fd创建一个event，最后添加到event\_base中。

# 唤醒流程：

        现在沿着这个内部的event的工作流程走一遍。

## 启动notify：

        首先往event写入一个字节，开启一切。由于这个event是内部的，用户是接触不到的。所以只能依靠Libevent提供的函数。当然这个函数也不会开放给用户，它只是供Libevent内部使用。

        现在来看Libevent内部的需要。在event\_add\_internal函数中需要通知主线程，在该函数的最后面会调用evthread\_notify\_base。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38556059) [copy](https://blog.csdn.net/luotuo44/article/details/38556059)

1. //event.c文件
2. **static** **int**
3. evthread\_notify\_base(**struct** event\_base \*base)
4. {
5. //确保已经加锁了
6. EVENT\_BASE\_ASSERT\_LOCKED(base);
7. **if** (!base->th\_notify\_fn)
8. **return** -1;
10. //写入一个字节，就能使event\_base被唤醒。
11. //如果处于未决状态，就没必要写多一个字节
12. **if** (base->is\_notify\_pending)
13. **return** 0;
15. //通知处于未决状态，当event\_base醒过来就变成已决的了。
16. base->is\_notify\_pending = 1;
17. **return** base->th\_notify\_fn(base);
18. }

        在evthread\_notify\_base中，会调用th\_notify\_fn函数指针。这个指针是在前面的evthread\_make\_base\_notifiable函数中被赋值的。这里以evthread\_notify\_base\_default作为例子。这个evthread\_notify\_base\_default完成实际的通知操作。

## 激活内部event：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38556059) [copy](https://blog.csdn.net/luotuo44/article/details/38556059)

1. //event.c文件。
2. **static** **int**
3. evthread\_notify\_base\_default(**struct** event\_base \*base)
4. {
5. **char** buf[1];
6. **int** r;
7. buf[0] = (**char**) 0;
8. //通知一下，用来唤醒。写一个字节足矣
9. #ifdef WIN32
10. r = send(base->th\_notify\_fd[1], buf, 1, 0);
11. #else
12. r = write(base->th\_notify\_fd[1], buf, 1);
13. #endif
14. //即使errno 等于 EAGAIN也无所谓，因为这是由于通信通道已经塞满了
15. //这已经能唤醒主线程了。没必要一定要再写入一个字节
16. **return** (r < 0 && errno != EAGAIN) ? -1 : 0;
17. }

        从上面两个函数看到，其实通知也是蛮简单的。只是往管道里面写入一个字节。当然这已经能使得event\_base检测到管道可读，从而实现唤醒event\_base。

        往管道写入一个字节，event\_base就会被唤醒，然后调用这个管道对应event的回调函数。当然，在event\_base醒来的时候，还能看到其他东西。这也是Libevent提供唤醒功能的原因。

        现在看一下这个唤醒event的回调函数，也是看默认的那个。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38556059) [copy](https://blog.csdn.net/luotuo44/article/details/38556059)

1. //event.c
2. **static** **void**
3. evthread\_notify\_drain\_default(evutil\_socket\_t fd, **short** what, **void** \*arg)
4. {
5. unsigned **char** buf[1024];
6. **struct** event\_base \*base = arg;
8. //读完fd的所有数据，免得再次被唤醒
9. #ifdef WIN32
10. **while** (recv(fd, (**char**\*)buf, **sizeof**(buf), 0) > 0)
11. ;
12. #else
13. **while** (read(fd, (**char**\*)buf, **sizeof**(buf)) > 0)
14. ;
15. #endif
17. EVBASE\_ACQUIRE\_LOCK(base, th\_base\_lock);
18. //修改之，使得其不再是未决的了。当然这也能让其他线程可以再次唤醒值。参看evthread\_notify\_base函数
19. base->is\_notify\_pending = 0;
20. EVBASE\_RELEASE\_LOCK(base, th\_base\_lock);
21. }

        这个函数也比较简单，也就是读取完管道里的所有数据，免得被多路IO复用函数检测到管道可读，而再次被唤醒。

        上面的流程就完成了Libevent的通知唤醒主线程的功能，思路还是蛮清晰的。实现起来也是很简单。

# 注意事项：

        有一点要注意：要让Libevent支持这种可通知机制，就必须让Libevent使用多线程，即在代码的一开始调用evthread\_use\_pthreads()或者evthread\_use\_windows\_threads()。虽然用户可以手动调用函数evthread\_make\_base\_notifiable。但实际上是不能实现通知功能的。分析如下：

        Libevent代码中是通过调用函数evthread\_notify\_base来通知的。但这个函数都是在一个if语句中调用的。判断的条件为是否需要通知。If成立的条件中肯定会&&上一个EVBASE\_NEED\_NOTIFY(base)。比如在event\_add\_internal函数中的为：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38556059) [copy](https://blog.csdn.net/luotuo44/article/details/38556059)

1. **if** (res != -1 && notify && EVBASE\_NEED\_NOTIFY(base))
2. evthread\_notify\_base(base);

        notify是根据函数中的判断而来，而EVBASE\_NEED\_NOTIFY这个宏定义的作用是判断当前的线程是否不等于主线程(即为event\_base执行event\_base\_dispatch函数的线程)。它是一个条件宏。其中一个实现为：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38556059) [copy](https://blog.csdn.net/luotuo44/article/details/38556059)

1. #define EVBASE\_NEED\_NOTIFY(base)             \
2. (\_evthread\_id\_fn != NULL &&          \
3. (base)->running\_loop &&           \
4. (base)->th\_owner\_id != \_evthread\_id\_fn())
6. #define EVTHREAD\_GET\_ID() \
7. (\_evthread\_id\_fn ? \_evthread\_id\_fn() : 1)

        event\_base结构体中th\_owner\_id变量指明当前为event\_base执行event\_base\_dispatch函数的是哪个线程。在event\_base\_loop函数中用宏EVTHREAD\_GET\_ID()赋值。

        如果一开始没有调用evthread\_use\_pthreads或者evthread\_use\_windows\_threads，那么全局变量evthread\_id\_fn就为NULL。也就不能获取线程的ID了。EVBASE\_NEED\_NOTIFY宏也只会返回0，使得不能调用evthread\_notify\_base函数。关于线程这部分的分析，可以参考《[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)》。

        下面的用一个例子验证。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38556059) [copy](https://blog.csdn.net/luotuo44/article/details/38556059)

1. #include<event.h>
2. #include<stdio.h>
3. #include<unistd.h>
4. #include<thread.h>
6. #include<pthread.h> //Linux thread
8. **struct** event\_base \*base = NULL;

11. **void** pipe\_cb(**int** fd, **short** events, **void** \*arg)
12. {
13. printf("in the cmd\_cb\n");
14. }
16. **void** timeout\_cb(**int** fd, **short** events, **void** \*arg)
17. {
18. printf("in the timeout\_cb\n");
19. }
21. **void**\* thread\_fn(**void** \*arg)
22. {
23. **char** ch;
24. scanf("%c", &ch); //just for wait
26. **struct** event \*ev = event\_new(base, -1, EV\_TIMEOUT | EV\_PERSIST,
27. timeout\_cb, NULL);
29. **struct** timeval tv = {2, 0};
30. event\_add(ev, &tv);
31. }
33. **int** main(**int** argc, **char** \*\* argv)
34. {
35. **if**( argc >= 2 && argv[1][0] == 'y')
36. evthread\_use\_pthreads();
38. base = event\_base\_new();
39. evthread\_make\_base\_notifiable(base);
41. **int** pipe\_fd[2];
42. pipe(pipe\_fd);
44. **struct** event \*ev = event\_new(base, pipe\_fd[0],
45. EV\_READ | EV\_PERSIST, pipe\_cb, NULL);
47. event\_add(ev, NULL);
49. pthread\_t **thread**;
50. pthread\_create(&**thread**, NULL, thread\_fn, NULL);

53. event\_base\_dispatch(base);
55. **return** 0;
56. }

        如果次线程的event被add到event\_base中，那么每2秒timeout\_cb函数就会被调用一次。如果没有被add的话，就永远等待下去，没有任何输出。

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/38556059