<https://blog.csdn.net/luotuo44/article/details/38350633>

**转载请注明出处:**[**http://blog.csdn.net/luotuo44/article/details/38350633**](http://blog.csdn.net/luotuo44/article/details/38350633)

        Libevent提供给用户的可见多线程API都在thread.h文件中。在这个文件提供的API并不多。基本上都是一些定制函数，像前面几篇博文说到的，可以为Libevent定制用户自己的多线程函数。

# 开启多线程：

        Libevent默认是不开启多线程的，也没有锁、条件变量这些东西。这点和前面博客说到的"没有定制就用Libevent默认提供"，有所不同。只有当你调用了evthread\_use\_windows\_threads()或者evthread\_use\_pthreads()或者调用evthread\_set\_lock\_callbacks函数定制自己的多线程、锁、条件变量才会开启多线程功能。其实，前面的那两个函数其内部实现也是定制，在函数的内部，Libevent封装的一套Win32线程、pthreads线程。然后调用evthread\_set\_lock\_callbacks函数，进行定制。

        thread.h文件只提供了定制线程的接口，并没有提供使用线程接口。这点很像前面说到的Libevent日志和内存分配。其实这也很好理解。因为都是你提供定制的线程函数。你都能提供了，你肯定有办法使用，没必要要Libevent提供一些API给你使用。

        如果用户为libevent开启了多线程，那么libevent里面的函数就会变成线程安全的。此时主线程在使用event\_base\_dispatch，别的线程是可以线程安全地使用event\_add把一个event添加到主线程的event\_base中。具体的工作原理可以参考《[evthread\_notify\_base通知主线程](http://blog.csdn.net/luotuo44/article/details/38556059)》。

# 锁和条件变量结构体：

        Libevent允许用户定制自己的锁和条件变量。其实现原理和前面说到的日志和内存分配一样，都是内部有一个全局变量。定制自己的锁和条件变量，就是对这个全局变量进行赋值。

        锁结构：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38350633) [copy](https://blog.csdn.net/luotuo44/article/details/38350633)

1. //thread.h文件
2. **struct** evthread\_lock\_callbacks {
3. //版本号，设置为宏EVTHREAD\_LOCK\_API\_VERSION
4. **int** lock\_api\_version;
5. //支持的锁类型，有普通锁，递归锁，读写锁三种
6. unsigned supported\_locktypes;
8. //分配一个锁变量(指针类型)，因为不同的平台锁变量是不同的类型
9. //所以用这个通用的void\*类型
10. **void** \*(\*alloc)(unsigned locktype);
11. **void** (\*free)(**void** \*lock, unsigned locktype);
12. **int** (\*lock)(unsigned mode, **void** \*lock);
13. **int** (\*unlock)(unsigned mode, **void** \*lock);
14. };

        目前Libevent支持的locktype(锁类型)有三种：

* 普通锁， 值为0
* 递归锁， 值为EVTHREAD\_LOCKTYPE\_RECURSIVE
* 读写锁， 值为EVTHREAD\_LOCKTYPE\_READWRITE

        当用户定制了自己的线程锁后，就可以用alloc这个函数指针调用函数，获取一个锁变量指针（在支持pthreads的系统获得的是pthread\_mutex\_t类型指针）。参数locktype就是用户指定的锁类型。

        参数mode(锁模式)则取下面的值：

* EVTHREAD\_READ：仅用于读写锁：为读操作请求或者释放锁
* EVTHREAD\_WRITE：仅用于读写锁：为写操作请求或者释放锁
* EVTHREAD\_TRY：仅用于锁定：仅在可以立刻锁定的时候才请求锁定

        虽然Libevent提供了这些锁类型和mode类型，但实际上是否支持这些类型完全是由所定制的线程锁决定的。Libevent提供的pthreads线程锁和WIN32线程锁就只支持其中的一部分。具体是哪些下面会说到。

        条件变量结构：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38350633) [copy](https://blog.csdn.net/luotuo44/article/details/38350633)

1. //thread.h文件
2. **struct** evthread\_condition\_callbacks {
3. //版本号，设置为EVTHREAD\_CONDITION\_API\_VERSION宏
4. **int** condition\_api\_version;
6. **void** \*(\*alloc\_condition)(unsigned condtype);
8. **void** (\*free\_condition)(**void** \*cond);
9. **int** (\*signal\_condition)(**void** \*cond, **int** broadcast);
10. **int** (\*wait\_condition)(**void** \*cond, **void** \*lock,
11. **const** **struct** timeval \*timeout);
12. };

        条件变量的版本为EVTHREAD\_CONDITION\_API\_VERSION时，alloc\_condition的参数取0。奇怪的是，Libevent并没有提供其他的版本号。前面的线程锁也是只提供给了一个版本号。

        signal\_condition的第一个参数为alloc\_condition的返回值，第二个参数指明唤醒多少个等待的线程。当broadcast取1时，唤醒所有的线程。取其他值时，只唤醒其中一个线程。

        wait\_condition的第二个参数为前面线程锁evthread\_lock\_callbacks结构中的alloc指针函数的返回值。熟悉条件变量的读者，这点还是比较容易懂的。对于第三个参数，和pthread\_cond\_timedwait有所不同。pthread\_cond\_timedwait的时间是绝对时间，这里的timeout则是等待的时间，所以千万不要用一个绝对时间作为参数值，不然等到老都等不到超时。如果该参数为NULL，那么就没有超时，将死等下去，直到另外的线程调用了signal\_condition。

# Libevent封装的多线程：

        说了这么多，其实对于用户来说，如果想让Libevent支持多线程。Windows用户直接调用evthread\_use\_windows\_threads()，遵循pthreads线程的系统直接调用evthread\_use\_pthreads()就可以了。其他什么东西都不需要做了。还有一点要注意的是，这两个函数要在代码的一开始就调用， 必须在event\_base\_new函数之前调用。好了，现在还是研究代码吧。

        下面看一下evthread\_use\_pthreads()函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38350633) [copy](https://blog.csdn.net/luotuo44/article/details/38350633)

1. //evthread\_pthreads.c文件
2. **int**
3. evthread\_use\_pthreads(**void**)
4. {
6. //结构体中做一些函数指针作为参数。这些函数都是定义在evthread\_pthread.c文件中
7. **struct** evthread\_lock\_callbacks cbs = {
8. EVTHREAD\_LOCK\_API\_VERSION,
9. EVTHREAD\_LOCKTYPE\_RECURSIVE,
10. evthread\_posix\_lock\_alloc,//函数指针
11. evthread\_posix\_lock\_free,//函数指针
12. evthread\_posix\_lock,//函数指针
13. evthread\_posix\_unlock//函数指针
14. };
15. **struct** evthread\_condition\_callbacks cond\_cbs = {
16. EVTHREAD\_CONDITION\_API\_VERSION,
17. evthread\_posix\_cond\_alloc,
18. evthread\_posix\_cond\_free,
19. evthread\_posix\_cond\_signal,
20. evthread\_posix\_cond\_wait
21. };
22. /\* Set ourselves up to get recursive locks. \*/
23. **if** (pthread\_mutexattr\_init(&attr\_recursive))
24. **return** -1;
25. **if** (pthread\_mutexattr\_settype(&attr\_recursive, PTHREAD\_MUTEX\_RECURSIVE))
26. **return** -1;
28. evthread\_set\_lock\_callbacks(&cbs); //定制锁操作
29. evthread\_set\_condition\_callbacks(&cond\_cbs); //定制条件变量操作
30. evthread\_set\_id\_callback(evthread\_posix\_get\_id); //设置可以获取线程ID的回调函数
31. **return** 0;
32. }

        函数一开始就定义并初始化了一个evthread\_lock\_callbacks结构和一个evthread\_condition\_callbacks结构。然后用之去进行定制。

        代码中的attr\_recursive是一个锁属性pthread\_mutexattr\_t类型的全局变量。在这个函数中，它被设置成具有递归属性。在申请锁时，可以看到其作用。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38350633) [copy](https://blog.csdn.net/luotuo44/article/details/38350633)

1. **static** **void** \*
2. evthread\_posix\_lock\_alloc(unsigned locktype)
3. {
4. pthread\_mutexattr\_t \*attr = NULL;
5. pthread\_mutex\_t \*lock = mm\_malloc(**sizeof**(pthread\_mutex\_t));
6. **if** (!lock)
7. **return** NULL;
8. **if** (locktype & EVTHREAD\_LOCKTYPE\_RECURSIVE)
9. attr = &attr\_recursive;
10. **if** (pthread\_mutex\_init(lock, attr)) {
11. mm\_free(lock);
12. **return** NULL;
13. }
14. **return** lock;
15. }

        可以看到这个全局变量是用于设置递归锁的。从这个函数可以看到，Libevent提供的pthreads版本锁只支持递归锁和普通非递归锁，并不支持读写锁。当然你可以提供一套支持读写锁的锁操作。阅读Libevent提供的WIN32版本锁代码，也可以看到并不支持读写锁。值得注意的是，WIN32的锁默认是具有递归功能的，无需用EVTHREAD\_LOCKTYPE\_RECURSIVE作参数值。

        前面还说到Libevent提供了EVTHREAD\_READ、EVTHREAD\_READ、EVTHREAD\_TRY三种锁模式(mode)。但在Libevent提供的pthreads版本锁中，只在evthread\_posix\_lock函数中使用到这些宏。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38350633) [copy](https://blog.csdn.net/luotuo44/article/details/38350633)

1. **static** **int**
2. evthread\_posix\_lock(unsigned mode, **void** \*\_lock)
3. {
4. pthread\_mutex\_t \*lock = \_lock;
5. **if** (mode & EVTHREAD\_TRY)
6. **return** pthread\_mutex\_trylock(lock);
7. **else**
8. **return** pthread\_mutex\_lock(lock);
9. }

        可以看到，它仅仅支持EVTHREAD\_TRY这个锁模式。WIN32版本也是如此。

        条件变量也简单地对系统native的条件进行一些简单的封装。这里就不多说了。在Windows中，因为在Windows Vista之前的Windows 操作系统并不支持提供条件变量，此时Libevent就使用Windows提供的EVENT进行一些封装来实现条件变量的功能。如果所在的Windows系统支持条件变量，Libevent将优先使用Windows本身提供的条件变量。这点可以在evthread\_use\_windows\_threads函数看到。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38350633) [copy](https://blog.csdn.net/luotuo44/article/details/38350633)

1. //evthread\_win32.c文件
2. **int**
3. evthread\_use\_windows\_threads(**void**)
4. {
5. **struct** evthread\_lock\_callbacks cbs = {
6. EVTHREAD\_LOCK\_API\_VERSION,
7. EVTHREAD\_LOCKTYPE\_RECURSIVE,
8. evthread\_win32\_lock\_create,
9. evthread\_win32\_lock\_free,
10. evthread\_win32\_lock,
11. evthread\_win32\_unlock
12. };

15. **struct** evthread\_condition\_callbacks cond\_cbs = {
16. EVTHREAD\_CONDITION\_API\_VERSION,
17. evthread\_win32\_cond\_alloc,
18. evthread\_win32\_cond\_free,
19. evthread\_win32\_cond\_signal,
20. evthread\_win32\_cond\_wait
21. };
22. #ifdef WIN32\_HAVE\_CONDITION\_VARIABLES //有内置的条件变量功能
23. **struct** evthread\_condition\_callbacks condvar\_cbs = {
24. EVTHREAD\_CONDITION\_API\_VERSION,
25. evthread\_win32\_condvar\_alloc,
26. evthread\_win32\_condvar\_free,
27. evthread\_win32\_condvar\_signal,
28. evthread\_win32\_condvar\_wait
29. };
30. #endif
32. evthread\_set\_lock\_callbacks(&cbs);
33. evthread\_set\_id\_callback(evthread\_win32\_get\_id);
35. //优先使用Windows自身提供的条件变量
36. #ifdef WIN32\_HAVE\_CONDITION\_VARIABLES
37. **if** (evthread\_win32\_condvar\_init()) {
38. evthread\_set\_condition\_callbacks(&condvar\_cbs);
39. **return** 0;
40. }
41. #endif
42. evthread\_set\_condition\_callbacks(&cond\_cbs);
44. **return** 0;
45. }

        一旦用户调用evthread\_use\_windows\_threads()或者evthread\_use\_pthreads()函数，那么用户就为Libevent定制了自己的线程锁操作。Libevent的其他代码中，如果需要用到锁，就会去调用这些线程锁操作。在实现上，当调用evthread\_use\_windows\_threads()或者evthread\_use\_pthreads()函数时，两个函数的内部都会调用evthread\_set\_lock\_callbacks函数。而这个设置函数会把前面两个evthread\_use\_xxx函数中定义的cbs变量值复制到一个evthread\_lock\_callbacks类型的\_evthread\_lock\_fns全局变量保存起来。以后，Libevent需要用到多线程锁操作，直接访问这个\_evthread\_lock\_fn变量即可。对于条件变量，也是用这样方式实现的。

# 定制的顺序：

        前面的一些博文和这篇都说到了用户可以定制自己的操作，比如内存分配、日志记录、线程锁。这些定制都应该放在代码的最前面，即不能在使用Libevent的event、event\_base这些结构体之后。因为这些结构体会使用到内存分配、日志记录、线程锁的。而这三者的定制顺序应该是：内存分配->日志记录->线程锁。

参考：

<http://www.wangafu.net/~nickm/libevent-book/Ref1_libsetup.html>

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/38350633