<https://blog.csdn.net/luotuo44/article/details/38360525>

**转载请注明出处:**[**http://blog.csdn.net/luotuo44/article/details/38360525**](http://blog.csdn.net/luotuo44/article/details/38360525)

# Debug锁操作：

         Libevent还支持对锁操作的一些检测，进而捕抓一些典型的锁错误。Libevent检查：

* 解锁自己(线程)没有持有的锁
* 在未解锁前，自己(线程)再次锁定一个非递归锁。

        Libevent通过一些变量记录锁的使用情况，当检查到这些锁的错误使用时，就调用abort，退出运行。

## 开启调试功能：

        用户只需在调用evthread\_use\_pthreads或者evthread\_use\_windows\_threads之后，调用evthread\_enable\_lock\_debuging()函数即可开启调试锁的功能。该函数有一个拼写错误。在2.1.2-alpha版本中会改正为evthread\_enable\_lock\_debugging，为了后向兼容，两者都会支持的。

        现在看一下Libevent是锁调试功能。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38360525) [copy](https://blog.csdn.net/luotuo44/article/details/38360525)

1. //evthread.c文件
2. **void**
3. evthread\_enable\_lock\_debuging(**void**)
4. {
5. **struct** evthread\_lock\_callbacks cbs = {
6. EVTHREAD\_LOCK\_API\_VERSION,
7. EVTHREAD\_LOCKTYPE\_RECURSIVE,
8. debug\_lock\_alloc,
9. debug\_lock\_free,
10. debug\_lock\_lock,
11. debug\_lock\_unlock
12. };
13. **if** (\_evthread\_lock\_debugging\_enabled)
14. **return**;
16. //把当前用户定制的锁操作复制到\_original\_lock\_fns结构体变量中。
17. memcpy(&\_original\_lock\_fns, &\_evthread\_lock\_fns,
18. **sizeof**(**struct** evthread\_lock\_callbacks));
20. //将当前的锁操作设置成调试锁操作。但调试锁操作函数内部
21. //还是使用\_original\_lock\_fns的锁操作函数
22. memcpy(&\_evthread\_lock\_fns, &cbs,
23. **sizeof**(**struct** evthread\_lock\_callbacks));
25. memcpy(&\_original\_cond\_fns, &\_evthread\_cond\_fns,
26. **sizeof**(**struct** evthread\_condition\_callbacks));
27. \_evthread\_cond\_fns.wait\_condition = debug\_cond\_wait;
28. \_evthread\_lock\_debugging\_enabled = 1;
30. /\* XXX return value should get checked. \*/
31. event\_global\_setup\_locks\_(0);
32. }

        在上面代码的注释可以知道，虽然evthread\_lock\_fns的值被更新为debug\_lock\_alloc、debug\_lock\_lock和debug\_lock\_unlock。但实际上，使用的还是之前用户定制的线程锁操作函数，只是加多了一层抽象而已。如果看不懂这段话，可以看下面的代码，看完已经就会懂的了。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38360525) [copy](https://blog.csdn.net/luotuo44/article/details/38360525)

1. //evthread.c文件
2. **static** **void** \*
3. debug\_lock\_alloc(unsigned locktype)
4. {
5. **struct** debug\_lock \*result = mm\_malloc(**sizeof**(**struct** debug\_lock));
6. **if** (!result)
7. **return** NULL;
9. //用户设置过自己的线程锁函数
10. **if** (\_original\_lock\_fns.alloc) {
11. //用用户定制的线程锁函数分配一个线程锁
12. **if** (!(result->lock = \_original\_lock\_fns.alloc(
13. locktype|EVTHREAD\_LOCKTYPE\_RECURSIVE))) {
14. mm\_free(result);
15. **return** NULL;
16. }
17. } **else** {
18. result->lock = NULL;
19. }
20. result->locktype = locktype;
21. result->count = 0;
22. result->held\_by = 0;
23. **return** result;
24. }

        现在看看Libevent是怎么调试（更准确来说，应该是检测）锁的。锁的检测，需要用到debug\_lock 结构体，它对锁的一些使用状态进行了记录。

## debug递归锁：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38360525) [copy](https://blog.csdn.net/luotuo44/article/details/38360525)

1. //evthread.c文件
2. **struct** debug\_lock {
3. unsigned locktype; //锁的类型
4. unsigned **long** held\_by; //这个锁是被哪个线程所拥有
5. /\* XXXX if we ever use read-write locks, we will need a separate
6. \* lock to protect count. \*/
7. **int** count; //这个锁的加锁次数
8. **void** \*lock; //锁类型，在pthreads下为pthread\_mutex\_t\*类型
9. };
11. **static** **int**
12. debug\_lock\_lock(unsigned mode, **void** \*lock\_)
13. {
14. **struct** debug\_lock \*lock = lock\_;
15. **int** res = 0;
16. **if** (lock->locktype & EVTHREAD\_LOCKTYPE\_READWRITE)
17. EVUTIL\_ASSERT(mode & (EVTHREAD\_READ|EVTHREAD\_WRITE));
18. **else**
19. EVUTIL\_ASSERT((mode & (EVTHREAD\_READ|EVTHREAD\_WRITE)) == 0);
20. **if** (\_original\_lock\_fns.lock)
21. res = \_original\_lock\_fns.lock(mode, lock->lock);
22. //lock 成功返回0，失败返回非0
23. **if** (!res) {
24. //记录这个锁的使用情况。
25. evthread\_debug\_lock\_mark\_locked(mode, lock);
26. }
27. **return** res;
28. }
30. **static** **void**
31. evthread\_debug\_lock\_mark\_locked(unsigned mode, **struct** debug\_lock \*lock)
32. {
33. ++lock->count; //增加锁的加锁次数.解锁时会减一
34. **if** (!(lock->locktype & EVTHREAD\_LOCKTYPE\_RECURSIVE))
35. EVUTIL\_ASSERT(lock->count == 1);
36. **if** (\_evthread\_id\_fn) {
37. unsigned **long** me;
38. me = \_evthread\_id\_fn(); //获取线程ID
39. **if** (lock->count > 1)
40. EVUTIL\_ASSERT(lock->held\_by == me);
41. lock->held\_by = me; //记录这个锁是被哪个线程所拥有
42. }
43. }

        这里主要是测试一个锁类型(如pthread\_mutex\_t)同时被加锁的次数。如果是一个非递归锁，那么将不允许多次锁定。对于锁的实现没有bug的话，如果是非递归锁，那么会在第二次锁住同一个锁时，卡死在debug\_lock\_lock 函数的original\_lock\_fns.lock上(即发生了死锁)。此时evthread\_debug\_lock\_mark\_locked是不会被调用的。但是，对于一个有bug的锁实现，那么就有可能发生这种情况。即对于非递归锁，其还是可以多次锁住同一个锁，并且不会发生死锁。此时，evthread\_debug\_lock\_mark\_locked函数将会被执行，在这个函数内部将会检测这种情况。Libevent的锁调试(检测)就是调试(检测)这种有bug的锁实现。

## debug解锁：

        现在看一下解锁时的检测。这主要是检测解锁一个自己没有锁定的锁，比如锁是由线程A锁定的，但线程B却去解锁。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38360525) [copy](https://blog.csdn.net/luotuo44/article/details/38360525)

1. //evthread.c文件。
2. **static** **int**
3. debug\_lock\_unlock(unsigned mode, **void** \*lock\_)
4. {
5. **struct** debug\_lock \*lock = lock\_;
6. **int** res = 0;
7. //先检测
8. evthread\_debug\_lock\_mark\_unlocked(mode, lock);
9. **if** (\_original\_lock\_fns.unlock)
10. res = \_original\_lock\_fns.unlock(mode, lock->lock);
11. **return** res;
12. }
14. **static** **void**
15. evthread\_debug\_lock\_mark\_unlocked(unsigned mode, **struct** debug\_lock \*lock)
16. {
17. **if** (lock->locktype & EVTHREAD\_LOCKTYPE\_READWRITE)
18. EVUTIL\_ASSERT(mode & (EVTHREAD\_READ|EVTHREAD\_WRITE));
19. **else**
20. EVUTIL\_ASSERT((mode & (EVTHREAD\_READ|EVTHREAD\_WRITE)) == 0);
21. **if** (\_evthread\_id\_fn) {
22. //检测锁的拥有者是否为要解锁的线程
23. EVUTIL\_ASSERT(lock->held\_by == \_evthread\_id\_fn());
24. **if** (lock->count == 1)
25. lock->held\_by = 0;
26. }
27. --lock->count;//减少被加锁次数
28. EVUTIL\_ASSERT(lock->count >= 0);
29. }

        从代码中可以看到，这里主要是检测解锁的线程是否为锁的实际拥有者。即检测是否解锁一个自己不拥有的锁。这里不是为了检测锁的实现是否有bug，而是检测锁在使用的时候是否有bug。

        当然Libevent提供的检测能力还是很有限的。特别是对于前一个检测，如果是使用Windows线程锁或者pthreads线程锁，这个检测并没有什么用。毕竟这些锁的实现已经经过了千锤百炼。

# 定制线程锁、条件变量：

        现在来看一下线程锁定制函数evthread\_set\_lock\_callbacks。本来这个定制应该放在[前一篇博客](http://blog.csdn.net/luotuo44/article/details/38350633)讲的。但由于其实现用到了调试锁的一些内容，所以就放到这里讲。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38360525) [copy](https://blog.csdn.net/luotuo44/article/details/38360525)

1. //evthread.h文件
2. **int**
3. evthread\_set\_lock\_callbacks(**const** **struct** evthread\_lock\_callbacks \*cbs)
4. {
5. **struct** evthread\_lock\_callbacks \*target =
6. \_evthread\_lock\_debugging\_enabled //默认为0
7. ? &\_original\_lock\_fns : &\_evthread\_lock\_fns;
9. **if** (!cbs) {//参数为NULL，取消线程锁功能
10. **if** (target->alloc)
11. event\_warnx("Trying to disable lock functions after "
12. "they have been set up will probaby not work.");
13. memset(target, 0, **sizeof**(\_evthread\_lock\_fns));
14. **return** 0;
15. }
17. //一旦设置就不能修改
18. **if** (target->alloc) {
19. /\* Uh oh; we already had locking callbacks set up.\*/
20. **if** (target->lock\_api\_version == cbs->lock\_api\_version &&
21. target->supported\_locktypes == cbs->supported\_locktypes &&
22. target->alloc == cbs->alloc &&
23. target->free == cbs->free &&
24. target->lock == cbs->lock &&
25. target->unlock == cbs->unlock) {
26. /\* no change -- allow this. \*/
27. **return** 0;
28. }
29. event\_warnx("Can't change lock callbacks once they have been "
30. "initialized.");
31. **return** -1;
32. }
34. //这个四个函数指针都不为NULL时才能成功定制。因为这四个函数是配套使用的
35. **if** (cbs->alloc && cbs->free && cbs->lock && cbs->unlock) {
36. memcpy(target, cbs, **sizeof**(\_evthread\_lock\_fns));
37. **return** event\_global\_setup\_locks\_(1);
38. } **else** {
39. **return** -1;
40. }
41. }

        全局变量\_evthread\_lock\_debugging\_enabled的初始化值为0，当调用evthread\_enable\_lock\_debuging函数后其值为1。于是，无论是在调试锁还是非调试的情况下，target变量都能够修改实际使用的evthread\_lock\_callbacks结构体（线程锁操作函数指针结构体）。前面已经说到了，在非调试情况下，实际使用的是\_evthread\_lock\_fns变量的线程锁函数指针成员。在调试情况下实际使用的是\_original\_lock\_fns变量的。

        从上面的代码中也可以看到：当参数为NULL时，就等于取消了线程锁功能。此后，Libevent的代码将运行在没有线程锁的无线程安全状态下。

        上面的第二个if语句则说明，在已经定制了线程锁之后，是无法再次定制的。我觉得这主要是怕：这个修改线程锁的动作刚好发生在另外一个线程获取获取锁的之后，即调用lock函数之后。并且是在另外的线程释放锁之前，即调用unlock函数之前。如果允许修改锁定制的线程锁，那么将可能发生，加锁和解锁操作是完全不同的两套线程锁。

        前面说到参数cbs可以为NULL，其实这就给了我们一个修改定制线程锁的方法。我们可以先用NULL作为参数调用一次evthread\_set\_lock\_callbacks函数，然后用真正的线程锁方案作为参数，再次调用evthread\_set\_lock\_callbacks函数。当然这相当容易发生bug。后面也会给出一个例子。

# 锁的使用：

## 加锁和解锁：

        Libevent中，一些函数支持多线程。一般都是使用锁进行线程同步。在Libevent的代码中，一般是使用EVTHREAD\_ALLOC\_LOCK宏获取一个锁变量，EVBASE\_ACQUIRE\_LOCK宏进行加锁，EVBASE\_RELEASE\_LOCK宏进行解锁。在阅读Libevent源代码中，一般都只会看到EVBASE\_ACQUIRE\_LOCK和EVBASE\_RELEASE\_LOCK。锁的内部实现是看不见的。

        现在对EVBASE\_ACQUIRE\_LOCK进行深究，看其是怎么一层层地封装的。先看event\_add函数的实现：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38360525) [copy](https://blog.csdn.net/luotuo44/article/details/38360525)

1. //event.c文件
2. **int**
3. event\_add(**struct** event \*ev, **const** **struct** timeval \*tv)
4. {
5. **int** res;
7. **if** (EVUTIL\_FAILURE\_CHECK(!ev->ev\_base)) {
8. event\_warnx("%s: event has no event\_base set.", \_\_func\_\_);
9. **return** -1;
10. }
11. //加锁
12. EVBASE\_ACQUIRE\_LOCK(ev->ev\_base, th\_base\_lock);
14. res = event\_add\_internal(ev, tv, 0);
15. //解锁
16. EVBASE\_RELEASE\_LOCK(ev->ev\_base, th\_base\_lock);
18. **return** (res);
19. }

        其中，EVBASE\_ACQUIRE\_LOCK是一个条件宏。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38360525) [copy](https://blog.csdn.net/luotuo44/article/details/38360525)

1. //evthread-internal.h文件
2. #ifndef WIN32
3. #define EVTHREAD\_EXPOSE\_STRUCTS
4. #endif

7. #if ! defined(\_EVENT\_DISABLE\_THREAD\_SUPPORT) && defined(EVTHREAD\_EXPOSE\_STRUCTS)
9. #define EVLOCK\_LOCK(lockvar,mode)                   \
10. **do** {                                \
11. **if** (lockvar)                        \
12. \_evthread\_lock\_fns.lock(mode, lockvar);     \
13. } **while** (0)
15. #define EVBASE\_ACQUIRE\_LOCK(base, lockvar) do {             \
16. EVLOCK\_LOCK((base)->lockvar, 0);         \
17. } **while** (0)

20. #elif ! defined(\_EVENT\_DISABLE\_THREAD\_SUPPORT)
22. **int** \_evthreadimpl\_lock\_lock(unsigned mode, **void** \*lock);

25. #define EVLOCK\_LOCK(lockvar,mode)                   \
26. **do** {                                \
27. **if** (lockvar)                        \
28. \_evthreadimpl\_lock\_lock(mode, lockvar);     \
29. } **while** (0)
31. #define EVBASE\_ACQUIRE\_LOCK(base, lockvar) do {             \
32. EVLOCK\_LOCK((base)->lockvar, 0);         \
33. } **while** (0)

36. #else //不支持多线程
37. #define EVBASE\_ACQUIRE\_LOCK(base, lock)  ((void)0)
39. #endif

42. //evthread.c文件
43. **int**
44. \_evthreadimpl\_lock\_lock(unsigned mode, **void** \*lock)
45. {
46. **if** (\_evthread\_lock\_fns.lock)
47. **return** \_evthread\_lock\_fns.lock(mode, lock);
48. **else**
49. **return** 0;
50. }

        虽然是条件宏，但最终都是调用了\_evthread\_lock\_fns结构体中的lock指针指向的函数，即调用了定制锁的锁函数，进行了锁定。但不同的是，在第一种宏中，并没有对\_evthread\_lock\_fns.lock这个指针作是否为NULL判断，而第二种宏，会在\_evthreadimpl\_lock\_lock对这个指针进行判断，当这个指针不为NULL时才进行函数调用。

        在非Windows系统上会把EVBASE\_ACQUIRE\_LOCK宏定义成第一种情况。但在Linux上调用event\_add时，即使\_evthread\_lock\_fns.lock为NULL也没有出现段错误。

        实际上，虽然第一种情况没有对\_evthread\_lock\_fns.lock进行判断，但它对lockvar进行了判断。但Lockvar为何物？顺藤摸瓜，lockvar为event\_base结构体中的th\_base\_lock成员，类型为viod\*。实际上，lockvar就是申请得到的锁变量。下面代码将看到如何申请。如果th\_base\_lock为NULL，那么就不会对\_evthread\_lock\_fns.lock这个函数指针进行函数调用了。

        在event\_base\_new\_with\_config函数可以看到th\_base\_lock成员的赋值情况。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38360525) [copy](https://blog.csdn.net/luotuo44/article/details/38360525)

1. **struct** event\_base \*
2. event\_base\_new\_with\_config(**const** **struct** event\_config \*cfg)
3. {
4. **struct** event\_base \*base;
6. //之所以不用mm\_malloc是因为mm\_malloc并不会清零该内存区域。
7. //而这个函数是会清零申请到的内存区域。这相当于给base初始化
8. **if** ((base = mm\_calloc(1, **sizeof**(**struct** event\_base))) == NULL) {
9. event\_warn("%s: calloc", \_\_func\_\_);
10. **return** NULL;
11. }
13. …….//其他成员的初始化
15. #ifndef \_EVENT\_DISABLE\_THREAD\_SUPPORT
17. //对于th\_base\_lock变量，目前的值为NULL.
18. //EVTHREAD\_LOCKING\_ENABLED宏是测试\_evthread\_lock\_fns.lock
19. //是否不为NULL
20. **if** (EVTHREAD\_LOCKING\_ENABLED() &&
21. (!cfg || !(cfg->flags & EVENT\_BASE\_FLAG\_NOLOCK))) {
22. **int** r;
23. EVTHREAD\_ALLOC\_LOCK(base->th\_base\_lock, //申请锁变量
24. EVTHREAD\_LOCKTYPE\_RECURSIVE);
25. }
26. #endif
28. …..
29. **return** (base);
30. }

        从这里可以看到，如果\_evthread\_lock\_fns.lock为NULL，那么th\_base\_lock成员肯定为NULL，那么后面就不会调用\_evthread\_lock\_fns.lock()函数。从而避过段错误。

        会不会th\_base\_lock不为NULL，而\_evthread\_lock\_fns.lock为NULL呢？

        th\_base\_lock是由要\_evthread\_lock\_fns.lock非NULL，才会被赋值为非NULL。如果\_evthread\_lock\_fns.lock为NULL，那么th\_base\_lock就肯定为NULL了。此外，结构体event\_base是定义是event\_internal.h文件的。所以，正常情况下，该结构体的成员是不可见的。所以你是无法直接访问并修改其成员。

        其实，有一种可能达到目标。就是先把\_evthread\_lock\_fns.lock赋值成非NULL，然后用来把th\_base\_lock赋值成非NULL，之后把\_evthread\_lock\_fns.lock修改为NULL。下面是Libevent提供的定制线程锁的函数evthread\_set\_lock\_callbacks。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38360525) [copy](https://blog.csdn.net/luotuo44/article/details/38360525)

1. **int**
2. evthread\_set\_lock\_callbacks(**const** **struct** evthread\_lock\_callbacks \*cbs)
3. {
4. …..
5. **if** (!cbs) {//参数为NULL，取消线程锁功能
6. **if** (target->alloc)
7. event\_warnx("Trying to disable lock functions after "
8. "they have been set up will probaby not work.");
9. memset(target, 0, **sizeof**(\_evthread\_lock\_fns));
10. **return** 0;
11. }
12. …..
13. }

        从代码中可以看到，当参数cbs为NULL时，是可以取消线程锁功能的。可以尝试编译运行下面的代码。代码一运行就可以看到段错误了。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38360525) [copy](https://blog.csdn.net/luotuo44/article/details/38360525)

1. #include  <event.h>
2. #include  <thread.h>
3. #include  <unistd.h>
5. **void** cmd\_cb(**int** fd, **short** event, **void** \*arg)
6. {
8. }
10. **int** main()
11. {
12. evthread\_use\_pthreads();
14. event\_base \*base = event\_base\_new();
16. evthread\_set\_lock\_callbacks(NULL);
18. event \*cmd\_event = event\_new(base, STDIN\_FILENO, EV\_READ | EV\_PERSIST,
19. cmd\_cb, base);
20. event\_add(cmd\_event, NULL);
22. event\_base\_dispatch(base);
24. **return** 0;
25. }

        从这里可以看到，一旦设置了线程、锁函数，那么就不应该对其进行修改。  
  
        值得注意的是，在Libevent中，像EVBASE\_ACQUIRE\_LOCK这个宏是专门给event\_base用的。

## 断言已加锁：

        在Libevent中，很多线程安全的函数都会调用一个已加锁断言。确保在进入这函数的时候，已经获得了一个锁。一般是调用EVENT\_BASE\_ASSERT\_LOCKED(base);完成这个断言。要注意的是：这个已锁断言要在开启了调试锁的前提下，才能使用的。

        下面代码可以看到断言锁是怎么实现的：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38360525) [copy](https://blog.csdn.net/luotuo44/article/details/38360525)

1. EVENT\_BASE\_ASSERT\_LOCKED(base);
3. #define EVENT\_BASE\_ASSERT\_LOCKED(base)      \
4. EVLOCK\_ASSERT\_LOCKED((base)->th\_base\_lock)

7. #define EVLOCK\_ASSERT\_LOCKED(lock)                  \
8. **do** {                                \
9. **if** ((lock) && \_evthread\_lock\_debugging\_enabled) {   \
10. EVUTIL\_ASSERT(\_evthread\_is\_debug\_lock\_held(lock)); \
11. }                           \
12. } **while** (0)

15. **int**
16. \_evthread\_is\_debug\_lock\_held(**void** \*lock\_)
17. {
18. **struct** debug\_lock \*lock = lock\_;
19. **if** (! lock->count)
20. **return** 0;
21. **if** (\_evthread\_id\_fn) {
22. unsigned **long** me = \_evthread\_id\_fn();
23. **if** (lock->held\_by != me)
24. **return** 0;
25. }
26. **return** 1;
27. }

        从EVLOCK\_ASSERT\_LOCKED宏的判断可以知道，\_evthread\_lock\_debugging\_enabled要不为0。而它的赋值是由evthread\_enable\_lock\_debuging()完成的，这个函数的作用就是开启锁调试功能。

        前面在讲调试锁的时候，有说到evthread\_debug\_lock\_mark\_locked函数，这个函数在加锁的时候会被调用。该函数会记录锁是由哪个线程加的，具体实现是通过记录线程ID。面的\_evthread\_is\_debug\_lock\_held函数的功能就是测试本线程ID是否等于之前加锁的线程ID。这样就完成了已加锁断言。

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/38360525