<https://blog.csdn.net/luotuo44/article/details/38800363>

**转载请注明出处：**[**http://blog.csdn.net/luotuo44/article/details/38800363**](http://blog.csdn.net/luotuo44/article/details/38800363)

# 使用evconnlistener：

        基于event和event\_base已经可以写一个CS模型了。但是对于服务器端来说，仍然需要用户自行调用socket、bind、listen、accept等步骤。这个过程有点繁琐，为此在2.0.2-alpha版本的Libevent推出了一些对应的封装函数。

        用户只需初始化struct sockaddr\_in结构体变量，然后把它作为参数传给函数evconnlistener\_new\_bind即可。该函数会完成上面说到的那4个过程。下面的代码是一个使用例子。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38800363) [copy](https://blog.csdn.net/luotuo44/article/details/38800363)

1. #include<netinet/in.h>
2. #include<sys/socket.h>
3. #include<unistd.h>
5. #include<stdio.h>
6. #include<string.h>
8. #include<event.h>
9. #include<listener.h>
10. #include<bufferevent.h>
11. #include<thread.h>

14. **void** listener\_cb(evconnlistener \*listener, evutil\_socket\_t fd,
15. **struct** sockaddr \*sock, **int** socklen, **void** \*arg);
17. **void** socket\_read\_cb(bufferevent \*bev, **void** \*arg);
18. **void** socket\_error\_cb(bufferevent \*bev, **short** events, **void** \*arg);
20. **int** main()
21. {
22. evthread\_use\_pthreads();//enable threads
24. **struct** sockaddr\_in sin;
25. memset(&sin, 0, **sizeof**(**struct** sockaddr\_in));
26. sin.sin\_family = AF\_INET;
27. sin.sin\_port = htons(8989);
29. event\_base \*base = event\_base\_new();
30. evconnlistener \*listener
31. = evconnlistener\_new\_bind(base, listener\_cb, base,
32. LEV\_OPT\_REUSEABLE|LEV\_OPT\_CLOSE\_ON\_FREE | LEV\_OPT\_THREADSAFE,
33. 10, (**struct** sockaddr\*)&sin,
34. **sizeof**(**struct** sockaddr\_in));
36. event\_base\_dispatch(base);
38. evconnlistener\_free(listener);
39. event\_base\_free(base);
41. **return** 0;
42. }

45. //有新的客户端连接到服务器
46. //当此函数被调用时，libevent已经帮我们accept了这个客户端。该客户端的
47. //文件描述符为fd
48. **void** listener\_cb(evconnlistener \*listener, evutil\_socket\_t fd,
49. **struct** sockaddr \*sock, **int** socklen, **void** \*arg)
50. {
51. event\_base \*base = (event\_base\*)arg;
53. //下面代码是为这个fd创建一个bufferevent
54. bufferevent \*bev =  bufferevent\_socket\_new(base, fd,
55. BEV\_OPT\_CLOSE\_ON\_FREE);
57. bufferevent\_setcb(bev, socket\_read\_cb, NULL, socket\_error\_cb, NULL);
58. bufferevent\_enable(bev, EV\_READ | EV\_PERSIST);
59. }

62. **void** socket\_read\_cb(bufferevent \*bev, **void** \*arg)
63. {
64. **char** msg[4096];
66. **size\_t** len = bufferevent\_read(bev, msg, **sizeof**(msg)-1 );
68. msg[len] = '\0';
69. printf("server read the data %s\n", msg);
71. **char** reply[] = "I has read your data";
72. bufferevent\_write(bev, reply, strlen(reply) );
73. }

76. **void** socket\_error\_cb(bufferevent \*bev, **short** events, **void** \*arg)
77. {
78. **if** (events & BEV\_EVENT\_EOF)
79. printf("connection closed\n");
80. **else** **if** (events & BEV\_EVENT\_ERROR)
81. printf("some other error\n");
83. //这将自动close套接字和free读写缓冲区
84. bufferevent\_free(bev);
85. }

        上面的代码是一个服务器端的例子，客户端代码可以使用《[Libevent使用例子，从简单到复杂](http://blog.csdn.net/luotuo44/article/details/39670221)》博文中的客户端。这里就不贴客户端代码了。

        从上面代码可以看到，当服务器端监听到一个客户端的连接请求后，就会调用listener\_cb这个回调函数。这个回调函数是在evconnlistener\_new\_bind函数中设置的。现在来看一下这个函数的参数有哪些，下面是其函数原型。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38800363) [copy](https://blog.csdn.net/luotuo44/article/details/38800363)

1. //listener.h文件
2. **typedef** **void** (\*evconnlistener\_cb)(**struct** evconnlistener \*, evutil\_socket\_t, **struct** sockaddr \*, **int** socklen, **void** \*);
4. **struct** evconnlistener \*evconnlistener\_new\_bind(**struct** event\_base \*base,
5. evconnlistener\_cb cb, **void** \*ptr, unsigned flags, **int** backlog,
6. **const** **struct** sockaddr \*sa, **int** socklen);

        第一个参数是很熟悉的event\_base，无论怎么样都是离不开event\_base这个发动机的。

        第二个参数是一个函数指针，该函数指针的格式如代码所示。当有新的客户端请求连接时，该函数就会调用。要注意的是：当这个回调函数被调用时，Libevent已经帮我们accept了这个客户端。所以，该回调函数有一个参数是文件描述符fd。我们直接使用这个fd即可。真是方便。这个参数是可以为NULL的，此时用户并不能接收到客户端。当用户调用evconnlistener\_set\_cb函数设置回调函数后，就可以了。

        第三个参数是传给回调函数的用户参数，作用就像event\_new函数的最后一个参数。

        参数flags是一些标志值，有下面这些：

* LEV\_OPT\_LEAVE\_SOCKETS\_BLOCKING：默认情况下，当连接监听器接收到新的客户端socket连接后，会把该socket设置为非阻塞的。如果设置该选项，那么就把之客户端socket保留为阻塞的
* LEV\_OPT\_CLOSE\_ON\_FREE：当连接监听器释放时，会自动关闭底层的socket
* LEV\_OPT\_CLOSE\_ON\_EXEC：为底层的socket设置close-on-exec标志
* LEV\_OPT\_REUSEABLE: 在某些平台，默认情况下当一个监听socket被关闭时，其他socket不能马上绑定到同一个端口，要等一会儿才行。设置该标志后，Libevent会把该socket设置成reuseable。这样，关闭该socket后，其他socket就能马上使用同一个端口
* LEV\_OPT\_THREADSAFE：为连接监听器分配锁。这样可以确保线程安全

        参数backlog是系统调用listen的第二个参数。最后两个参数就不多说了。

# evconnlistener的封装：

        接下来看一下Libevent是怎么封装evconnlistener的。

## 用到的结构体：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38800363) [copy](https://blog.csdn.net/luotuo44/article/details/38800363)

1. //listener.c文件
2. **struct** evconnlistener\_ops {//一系列的工作函数
3. **int** (\*enable)(**struct** evconnlistener \*);
4. **int** (\*disable)(**struct** evconnlistener \*);
5. **void** (\*destroy)(**struct** evconnlistener \*);
6. **void** (\*shutdown)(**struct** evconnlistener \*);
7. evutil\_socket\_t (\*getfd)(**struct** evconnlistener \*);
8. **struct** event\_base \*(\*getbase)(**struct** evconnlistener \*);
9. };
11. **struct** evconnlistener {
12. **const** **struct** evconnlistener\_ops \*ops;//操作函数
13. **void** \*lock; //锁变量，用于线程安全
14. evconnlistener\_cb cb;//用户的回调函数
15. evconnlistener\_errorcb errorcb;//发生错误时的回调函数
16. **void** \*user\_data;//回调函数的参数
17. unsigned flags;//属性标志
18. **short** refcnt;//引用计数
19. unsigned enabled : 1;//位域为1.即只需一个比特位来存储这个成员
20. };
22. **struct** evconnlistener\_event {
23. **struct** evconnlistener base;
24. **struct** event listener; //内部event,插入到event\_base
25. };

        在evconnlistener\_event结构体有一个event结构体。可以想象，在实现时必然是将服务器端的socket fd赋值给struct event 类型变量listener的fd成员。然后将listener加入到event\_base，这样就完成了自动监听工作。这也回归到之前学过的内容。

        下面看一下具体是怎么实现的。

## 初始化服务器socket：

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38800363) [copy](https://blog.csdn.net/luotuo44/article/details/38800363)

1. //listener.c文件
2. **struct** evconnlistener \*
3. evconnlistener\_new\_bind(**struct** event\_base \*base, evconnlistener\_cb cb,
4. **void** \*ptr, unsigned flags, **int** backlog, **const** **struct** sockaddr \*sa,
5. **int** socklen)
6. {
7. **struct** evconnlistener \*listener;
8. evutil\_socket\_t fd;
9. **int** on = 1;
10. **int** family = sa ? sa->sa\_family : AF\_UNSPEC;
12. //监听个数不能为0
13. **if** (backlog == 0)
14. **return** NULL;
16. fd = socket(family, SOCK\_STREAM, 0);
17. **if** (fd == -1)
18. **return** NULL;
20. //LEV\_OPT\_LEAVE\_SOCKETS\_BLOCKING选项是应用于accept到的客户端socket
21. //所以对于服务器端的socket，直接将之设置为非阻塞的
22. **if** (evutil\_make\_socket\_nonblocking(fd) < 0) {
23. evutil\_closesocket(fd);
24. **return** NULL;
25. }
27. **if** (flags & LEV\_OPT\_CLOSE\_ON\_EXEC) {
28. **if** (evutil\_make\_socket\_closeonexec(fd) < 0) {
29. evutil\_closesocket(fd);
30. **return** NULL;
31. }
32. }
34. **if** (setsockopt(fd, SOL\_SOCKET, SO\_KEEPALIVE, (**void**\*)&on, **sizeof**(on))<0) {
35. evutil\_closesocket(fd);
36. **return** NULL;
37. }
38. **if** (flags & LEV\_OPT\_REUSEABLE) {
39. **if** (evutil\_make\_listen\_socket\_reuseable(fd) < 0) {
40. evutil\_closesocket(fd);
41. **return** NULL;
42. }
43. }
45. **if** (sa) {
46. **if** (bind(fd, sa, socklen)<0) {//绑定
47. evutil\_closesocket(fd);
48. **return** NULL;
49. }
50. }
52. listener = evconnlistener\_new(base, cb, ptr, flags, backlog, fd);
53. **if** (!listener) {
54. evutil\_closesocket(fd);
55. **return** NULL;
56. }
58. **return** listener;
59. }

        evconnlistener\_new\_bind函数申请一个socket，然后对之进行一些有关非阻塞、重用、保持连接的处理、绑定到特定的IP和端口。最后把业务逻辑交给evconnlistener\_new处理。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38800363) [copy](https://blog.csdn.net/luotuo44/article/details/38800363)

1. //listener.c文件
2. **static** **const** **struct** evconnlistener\_ops evconnlistener\_event\_ops = {
3. event\_listener\_enable,
4. event\_listener\_disable,
5. event\_listener\_destroy,
6. NULL, /\* shutdown \*/
7. event\_listener\_getfd,
8. event\_listener\_getbase
9. };

12. **struct** evconnlistener \*
13. evconnlistener\_new(**struct** event\_base \*base,
14. evconnlistener\_cb cb, **void** \*ptr, unsigned flags, **int** backlog,
15. evutil\_socket\_t fd)
16. {
17. **struct** evconnlistener\_event \*lev;
19. **if** (backlog > 0) {
20. **if** (listen(fd, backlog) < 0)
21. **return** NULL;
22. } **else** **if** (backlog < 0) {
23. **if** (listen(fd, 128) < 0)
24. **return** NULL;
25. }
27. lev = mm\_calloc(1, **sizeof**(**struct** evconnlistener\_event));
28. **if** (!lev)
29. **return** NULL;
31. //赋值
32. lev->base.ops = &evconnlistener\_event\_ops;
33. lev->base.cb = cb;
34. lev->base.user\_data = ptr;
35. lev->base.flags = flags;
36. lev->base.refcnt = 1;
38. **if** (flags & LEV\_OPT\_THREADSAFE) {//线程安全就需要分配锁
39. EVTHREAD\_ALLOC\_LOCK(lev->base.lock, EVTHREAD\_LOCKTYPE\_RECURSIVE);
40. }
42. //在多路IO复用函数中，新客户端的连接请求也被当作读事件
43. event\_assign(&lev->listener, base, fd, EV\_READ|EV\_PERSIST,
44. listener\_read\_cb, lev);
46. //会调用event\_add，把event加入到event\_base中
47. evconnlistener\_enable(&lev->base);
49. **return** &lev->base;
50. }
52. **int**
53. evconnlistener\_enable(**struct** evconnlistener \*lev)
54. {
55. **int** r;
56. LOCK(lev);
57. lev->enabled = 1;
58. **if** (lev->cb)
59. r = lev->ops->enable(lev);//实际上是调用下面的event\_listener\_enable函数
60. **else**
61. r = 0;
62. UNLOCK(lev);
63. **return** r;
64. }
66. **static** **int**
67. event\_listener\_enable(**struct** evconnlistener \*lev)
68. {
69. **struct** evconnlistener\_event \*lev\_e =
70. EVUTIL\_UPCAST(lev, **struct** evconnlistener\_event, base);
72. //加入到event\_base，完成监听工作。
73. **return** event\_add(&lev\_e->listener, NULL);
74. }

        几个函数的一路调用，思路还是挺清晰的。就是申请一个socket，进行一些处理，然后用之赋值给event。最后把之add到event\_base中。event\_base会对新客户端的请求连接进行监听。

        在evconnlistener\_enable函数里面，如果用户没有设置回调函数，那么就不会调用event\_listener\_enable。也就是说并不会add到event\_base中。

        event\_listener\_enable函数里面的宏EVUTIL\_UPCAST可以根据结构体成员变量的地址推算出结构体的起始地址。有关这个宏，可以查看”[结构体偏移量](http://blog.csdn.net/luotuo44/article/details/38780157#t10)”。

## 处理客户端的连接请求：

        现在来看一下event的回调函数listener\_read\_cb。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38800363) [copy](https://blog.csdn.net/luotuo44/article/details/38800363)

1. //listener.c文件
2. **static** **void**
3. listener\_read\_cb(evutil\_socket\_t fd, **short** what, **void** \*p)
4. {
5. **struct** evconnlistener \*lev = p;
6. **int** err;
7. evconnlistener\_cb cb;
8. evconnlistener\_errorcb errorcb;
9. **void** \*user\_data;
10. LOCK(lev);
11. **while** (1) { //可能有多个客户端同时请求连接
12. **struct** sockaddr\_storage ss;
13. #ifdef WIN32
14. **int** socklen = **sizeof**(ss);
15. #else
16. socklen\_t socklen = **sizeof**(ss);
17. #endif
18. evutil\_socket\_t new\_fd = accept(fd, (**struct** sockaddr\*)&ss, &socklen);
19. **if** (new\_fd < 0)
20. **break**;
21. **if** (socklen == 0) {
22. /\* This can happen with some older linux kernels in
23. \* response to nmap. \*/
24. evutil\_closesocket(new\_fd);
25. **continue**;
26. }
28. **if** (!(lev->flags & LEV\_OPT\_LEAVE\_SOCKETS\_BLOCKING))
29. evutil\_make\_socket\_nonblocking(new\_fd);
31. //用户还没设置连接监听器的回调函数
32. **if** (lev->cb == NULL) {
33. UNLOCK(lev);
34. **return**;
35. }
37. //由于refcnt被初始化为1.这里有++了，所以一般情况下并不会进入下面的
38. //if判断里面。但如果程在下面UNLOCK之后，第二个线调用evconnlistener\_free
39. //释放这个evconnlistener时，就有可能使得refcnt为1了。即进入那个判断体里
40. //执行listener\_decref\_and\_unlock。在下面会讨论这个问题。
41. ++lev->refcnt;
42. cb = lev->cb;
43. user\_data = lev->user\_data;
44. UNLOCK(lev);
45. cb(lev, new\_fd, (**struct** sockaddr\*)&ss, (**int**)socklen,
46. user\_data);//调用用户设置的回调函数，让用户处理这个fd
47. LOCK(lev);
48. **if** (lev->refcnt == 1) {
49. **int** freed = listener\_decref\_and\_unlock(lev);
50. EVUTIL\_ASSERT(freed);
51. **return**;
52. }
53. --lev->refcnt;
54. }
56. err = evutil\_socket\_geterror(fd);
57. **if** (EVUTIL\_ERR\_ACCEPT\_RETRIABLE(err)) {//还可以accept
58. UNLOCK(lev);
59. **return**;
60. }
62. //当有错误发生时才会运行到这里
63. **if** (lev->errorcb != NULL) {
64. ++lev->refcnt;
65. errorcb = lev->errorcb;
66. user\_data = lev->user\_data;
67. UNLOCK(lev);
68. errorcb(lev, user\_data);//调用用户设置的错误回调函数
69. LOCK(lev);
70. listener\_decref\_and\_unlock(lev);
71. }
72. }

        这个函数所做的工作也比较简单，就是accept客户端，然后调用用户设置的回调函数。所以，用户回调函数的参数fd是一个已经连接好了的socket。

        上面函数说到了错误回调函数，可以通过下面的函数设置连接监听器的错误监听函数。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38800363) [copy](https://blog.csdn.net/luotuo44/article/details/38800363)

1. //listener.h文件
2. **typedef** **void** (\*evconnlistener\_errorcb)(**struct** evconnlistener \*, **void** \*);
4. //listener.c文件
5. **void**
6. evconnlistener\_set\_error\_cb(**struct** evconnlistener \*lev,
7. evconnlistener\_errorcb errorcb)
8. {
9. LOCK(lev);
10. lev->errorcb = errorcb;
11. UNLOCK(lev);
12. }

# 释放evconnlistener：

        调用evconnlistener\_free可以释放一个evconnlistener。由于evconnlistener拥有一些系统资源，在释放evconnlistener\_free的时候会释放这些系统资源。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/38800363) [copy](https://blog.csdn.net/luotuo44/article/details/38800363)

1. //listener.c文件
2. **void**
3. evconnlistener\_free(**struct** evconnlistener \*lev)
4. {
5. LOCK(lev);
6. lev->cb = NULL;
7. lev->errorcb = NULL;
8. **if** (lev->ops->shutdown)//这里的shutdown为NULL
9. lev->ops->shutdown(lev);
11. //引用次数减一，并解锁
12. listener\_decref\_and\_unlock(lev);
13. }
15. **static** **int**
16. listener\_decref\_and\_unlock(**struct** evconnlistener \*listener)
17. {
18. **int** refcnt = --listener->refcnt;
19. **if** (refcnt == 0) {
20. //实际调用event\_listener\_destroy
21. listener->ops->destroy(listener);
22. UNLOCK(listener);
23. //释放锁
24. EVTHREAD\_FREE\_LOCK(listener->lock, EVTHREAD\_LOCKTYPE\_RECURSIVE);
25. mm\_free(listener);
26. **return** 1;
27. } **else** {
28. UNLOCK(listener);
29. **return** 0;
30. }
31. }
33. **static** **void**
34. event\_listener\_destroy(**struct** evconnlistener \*lev)
35. {
36. **struct** evconnlistener\_event \*lev\_e =
37. EVUTIL\_UPCAST(lev, **struct** evconnlistener\_event, base);
39. //把event从event\_base中删除
40. event\_del(&lev\_e->listener);
41. **if** (lev->flags & LEV\_OPT\_CLOSE\_ON\_FREE)//如果用户设置了这个选项，那么要关闭socket
42. evutil\_closesocket(event\_get\_fd(&lev\_e->listener));
43. }

        要注意一点，LEV\_OPT\_CLOSE\_ON\_FREE选项关闭的是服务器端的监听socket，而非那些连接客户端的socket。

        现在来说一下那个listener\_decref\_and\_unlock。前面注释说到，在函数listener\_read\_cb中，一般情况下是不会调用listener\_decref\_and\_unlock，但在多线程的时候可能会调用。这种特殊情况是：当主线程accept到一个新客户端时，会解锁，并调用用户设置的回调函数。此时，引用计数等于2。就在这个时候，第二个线程执行evconnlistener\_free函数。该函数会执行listener\_decref\_and\_unlock。明显主线程还在用这个evconnlistener，肯定不能删除。此时引用计数也等于2也不会删除。但用户已经调用了evconnlistener\_free。Libevent必须要响应。当第二个线程执行完后，主线程抢到CPU，此时引用计数就变成1了，也就进入到if判断里面了。在判断体里面执行函数listener\_decref\_and\_unlock，并且完成删除工作。

        总得来说，Libevent封装的这个evconnlistener和一系列操作函数，还是比较简单的。思路也比较清晰。

参考：

        http://www.wangafu.net/~nickm/libevent-book/Ref8\_listener.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/38800363