**libevent多线程使用bufferevent的那些事**

置顶 2016年11月09日 17:42:20 [linux\_c\_coding\_man](https://me.csdn.net/peng314899581) 阅读数：3528 标签： [libevent](http://so.csdn.net/so/search/s.do?q=libevent&t=blog) [buffereven](http://so.csdn.net/so/search/s.do?q=buffereven&t=blog) 更多

个人分类： [libev/libevent](https://blog.csdn.net/peng314899581/article/category/6504167) [网络编程](https://blog.csdn.net/peng314899581/article/category/6508056) [多线程](https://blog.csdn.net/peng314899581/article/category/6337633)

版权声明：本文为博主原创文章，如需转载请标明出处链接！否则追究法律责任！ https://blog.csdn.net/peng314899581/article/details/53102720

void do\_accept(struct evconnlistener \*listener, evutil\_socket\_t fd, struct sockaddr \*sa, int socklen, void \*ctx)

{

//不使用Nagle算法，选择立即发送数据而不是等待产生更多的数据然后再一次发送

int optval = 1;

setsockopt(fd, SOL\_SOCKET, TCP\_NODELAY, &optval, sizeof(optval));

struct bufferevent \* bev = bufferevent\_socket\_new(base, fd, BEV\_OPT\_CLOSE\_ON\_FREE);

if (NULL == bev)

{

return 1;

}

bufferevent\_setcb(bev, do\_read, NULL, NULL, NULL);

bufferevent\_enable(bev, EV\_READ);

}

在之前的《libevent使用event事件触发服务端实例》中，do\_accept使用的event，这里使用了bufferevent，它们的区别：   
1.bufferevent有自己的一套错误触发处理机制，可以在bufferevent\_setcb设置错误回调函数，event没有   
2.void do\_read(struct bufferevent \*bev, void \*ctx)；这是bufferevent的读事件回调函数格式   
void do\_read(evutil\_socket\_t fd, short ev, void \*ctx)；这是event的读事件回调函数格式   
对于event回调传过去的参数是套接字，也就是说我们自己要在do\_read中做recv操作，并存进数据缓冲中，对于bufferevent回调传过去的是bufferevent事件，看一下bufferevent的结构

struct bufferevent

{

struct event\_base \*ev\_base;

struct event ev\_read;

struct event ev\_write;

struct evbuffer \*input;//接收数据缓冲

struct evbuffer \*output;//发送数据缓冲

struct event\_watermark wm\_read;

struct event\_watermark wm\_write;

evbuffercb readcb;//读回调函数指针

evbuffercb writecb;//写回调函数指针

everrorcb errorcb;//错误回调函数指针

void \*cbarg;

int timeout\_read; /\* in seconds \*/

int timeout\_write; /\* in seconds \*/

short enabled; /\* events that are currently enabled \*/

};

从结构可以看出来，bufferevent已经帮我们做了接收操作，并存在了缓冲input中，也就是说bufferevent相比较event是多了数据缓冲绑定，我们不需要考虑数据缓冲的事情了。当读事件触发，我们就调用bufferevent\_read从input拿数据，一切都是那么轻松。既然bufferevent做了接收处理，当然也会通过recv返回值给我们做了错误回调。但是bufferevent不是线程安全的，具体表现：   
1. 当你使用bufferevent发送或者接收数据的时候，同时发生断开情况(如客户端强行关闭)，事件回调会收到这个消息， 你当然要bufferevent\_free,两个线程对同一个bufferevent出现的抢占会造成段错误，当然你可以打开libevent的锁机制解决这个问题，那也只是解决这个问题，又会带来另一个问题。现在发送或者接收和释放不冲突了，但是释放断开连接之后还发送数据的话，会造成死锁，锁死在bufferevent的锁里面（bufferevent\_write(m\_bev, data, len)这个函数里）**（!!!本人碰到的正是这个问题!!!**）。   
2. 当你为新的连接创建bufferevent加入base里面的时候，另外有一个断开连接要从base里面拿出去，那么会造成对base的抢占段错误。当然event也会存在这个问题，打开libevent的锁是可以解决这个问题的。

对于第一种情况，解决的办法是有的，可以给bufferevent在包装一层，标记一个是否可用选项，释放的时候把标识设为不可用，发送和接收的时候先判断是否可用再操作来避免死锁，也就是说先打开了libevent的锁，然后包装一层之后为了这个标识又要再加一次锁。后来看了memcache源码，发现它没有使用bufferevent，只是使用了event来做触发，后面的所有操作都是围绕着套接字进行的，于是我也乖乖的使用基本套接字来进行所有的操作，毕竟关闭套接字的时候使用套接字发送接收不会崩溃，只是返回一个错误码给我们判断，毕竟关闭套接字之后再发送数据也不会死锁也是返回错误码给我们处理。