<https://blog.csdn.net/luotuo44/article/details/39547391>

**转载请注明出处：**[**http://blog.csdn.net/luotuo44/article/details/39547391**](http://blog.csdn.net/luotuo44/article/details/39547391)

        前段时间阅读了libevent的源码。读毕，之前使用libevent时的一些疑问都已经豁然开朗了。对于libevent源码的分析，可以移步<http://blog.csdn.net/luotuo44/article/category/2435521>查看。如果是libevent的初学者，可以先阅读《[libevent使用例子，从简单到复杂](http://blog.csdn.net/luotuo44/article/details/39670221)》。

        本文通过自问自答的形式，希望能帮助其他人解答在使用libevent时的一些疑惑。

# ****一个文件描述符可以关联多个event吗？****

        同一个文件描述符(fd)是可以多次调用event\_new，产生不同的event的。这些具有相同fd的event，回调函数和回调参数是可以不同的。而且它们监听的事件也是可以不同(这可能是能关联多个event的原因吧)。

        如果多个event监听同一个fd的同一个事件，比如可读事件。那么当这个fd变成可读后，所有的监听该事件的event的回调函数都会被调用(即触发event)。没有监听该事件的event的回调函数不会被调用。如果两个event监听同一个fd的不同事件，那么它们的触发相互独立。

# ****一个超时event可以多次调用event\_add函数吗？****

        一个超时event是可以多次调用event\_add函数的。其实所有的event都可以多次调用event\_add函数。不过只有超时event多次调用才有实质的意义，其他event多次调用会被发现，然后被遣返(return)。

        如果每次调用event\_add时，超时值不同的话，那么以最后一次调用的为准。如果想取消超时，让这个event变成普通的event，直接把event\_add的第二个参数设为NULL即可。

# 怎么把一个超时event设置成永久触发(EV\_PERSIST)?

        通过libevent提供的evtimer\_xxx宏函数，是无法把一个超时event设置成永久的。于是有一些人就想在该超时event的超时回调函数中再次调用evtimer\_add函数。这就有点像对不可靠的信号再次设置信号处理函数。

        其实，不使用libevent提供的这些evtimer\_xx宏函数即可。一个event之所以是超时event，不是因为调用了evtimer\_xx这些宏函数，而是因为在调用event\_add函数时，第二个参数不为NULL。所以我们可以像下面代码那样设置一个永久的超时event

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39547391) [copy](https://blog.csdn.net/luotuo44/article/details/39547391)

1. **struct** event \*ev = event\_new(base, -1, EV\_PERSIST, cb, arg);
2. **struct** timeval timout = {2, 0}; //两秒的超时
3. event\_add(ev, &timeout);

# ****超时event的实现依赖于系统时间吗？****

        也可以这样问：使用了超时event，如果用户手动修改了系统时间，会有影响吗？

        如果所在的系统支持MONOTONIC时间的话，那么没有影响。如果不支持那么会有一些影响，即超时不那么准确。关于MONOTONIC时间，可以参考[这里](http://blog.csdn.net/luotuo44/article/details/39374759#t4)。可以用下面的代码测试你的系统是否支持MONOTONIC时间。

**[cpp]** [view plain](https://blog.csdn.net/luotuo44/article/details/39547391) [copy](https://blog.csdn.net/luotuo44/article/details/39547391)

1. **struct** timespec ts;
2. **if** (clock\_gettime(CLOCK\_MONOTONIC, &ts) == 0)
3. printf("支持\n");
4. **else**
5. printf("不支持\n");

        如果不是频繁修改，假如只修改一次，那么只会在修改后第一次的超时可能(仅仅是可能)会不那么准确，之后的超时又准确了(如果该event有EV\_PERSIST选项的话)。为什么说“仅仅”、“不那么”呢？因为这取决于你在Libevent处于什么状态下  修改的系统时间。细节可以参考<http://blog.csdn.net/luotuo44/article/details/38661787#t2>。

# ****对某个信号进行捕抓 和 对该信号使用信号event 能同时进行吗？****

        不可以！！

        因为Libevent内部实现信号event的原理就是对该信号设置一个信号捕抓函数(这也叫统一事件源)。 对信号捕抓熟悉的读者应该明白，信号捕抓函数只能有一个。你设置了另外一个，那么将覆盖之前设置的。

# ****在Linux中，Libevent默认使用epoll吗？****

        如果你的系统支持epoll，那么它将优先使用epoll。事实上，Libevent总是优先选择高性能的多路IO复用函数(在Windows上却是一个例外，它并不优先使用IOCP)。

# ****怎么知道libevent具体是使用了哪个多路IO复用函数？****

        当你调用event\_base\_new得到一个event\_base后，就确定使用哪个多路IO复用函数了。此时调用event\_base\_get\_method函数就能得到该event\_base使用的是哪个多路IO复用函数。该函数返回一个字符串，字符串的内容就是”select”、”poll”、”epoll” 这类多路IO复用函数的名称。不过在Windows平台上，返回是的”win32”。一般情况下，在Windows平台上是选用select的。至于什么时候选用IOCP，可以参考下一条。

# ****在Windows中，怎么使用IOCP？****

        Windows中，Libevent对IOCP的支持比较少。只在连接监听器evconnlistener和bufferevent  socket中支持IOCP。此外，因为Libevent在Windows平台默认选择select，要通过设置EVENT\_BASE\_FLAG\_STARTUP\_IOCP宏，Libevent才会使用IOCP。

        可以通过event\_config\_set\_flag函数设置这个宏。具体的内容可以参考设置，可以参考<http://blog.csdn.net/luotuo44/article/details/38443569#t5>

# ****什么时候调用evthread\_use\_pthreads函数？****

        如果要使用多线程，需要线程安全，那么在调用event\_base\_new函数之前一定要调用该函数(对应的Windows版本为evthread\_use\_windows\_threads)。如果在event\_base\_new之后才调用evthread\_use\_pthreads，那么该event\_base就不会是线程安全的了。原理可以参考这里<http://blog.csdn.net/luotuo44/article/details/38501341#t0>。

        注意：该函数只是确保Libevent线程安全，多线程的使用还是要靠自己写代码。Libevent里面的代码也没有使用多线程，它仅仅用到了锁和条件变量。

# ****Libevent允许定制内存分配、日志、线程锁，这些定制有顺序要求吗？****

        有。首先，这三个东西的定制都应该放到程序的前面，确保放到其他任何libevent API调用前。其次，这个三者也是有顺序。它们的顺序应该为：内存分配、日志记录、线程锁。

# ****可以在次线程调用event\_add添加一个event吗？****

        可以。但为了安全，必须确保你的程序在一开始调用了evthread\_use\_pthreads函数。如果主线程不是在调用其他event的回调函数，那么该event将马上被主线程添加到监听队列中，和其他event一起等待事件的发生。至于主线程如何知晓次线程添加了event，可以参考《[evthread\_notify\_base通知主线程](http://blog.csdn.net/luotuo44/article/details/38556059)》。

# ****Libevent哪些函数是线程安全的？****

        你认为一个函数理应线程安全，那么Libevent的作者也会认为该函数得是线程安全的。

        调用evthread\_use\_pthreads函数后，就放心使用Libevent提供的函数吧。它总会在需要加锁的时候加锁，保证线程安全的。

# ****bufferevent线程安全吗？****

        你在调用bufferevent\_socket\_new的时候加入了BEV\_OPT\_THREADSAFE选项，那么就线程安全了。

# ****bufferevent\_write是非阻塞的吗？****

       是非阻塞的。调用后，不会被阻塞，能马上返回。

# ****既然bufferevent\_write马上返回，那么返回后用户的那份数据是否可以删除了？****

        如果读者试过仅仅用OS提供的系统网络API写非阻塞socket的发送代码的话，那么一定被非阻塞气死了。你得考虑要发送的数据并没有在一次write调用中发送完。此时，得找一个地方保存这些要发送的数据，等到下次调用write时还要使用。但找一个地方是一个烦人的事情。

        虽然bufferevent\_write是非阻塞的，但它很好人。当它返回后，他已经把用户要发送的数据都copy了一份，保存在内部的缓冲区中。所以从bufferevent\_write返回后，就可以丢弃要发送的数据了，无需伤脑筋找地方保存这些数据。

# ****bufferevent的可读事件是水平触发还是边沿触发？****

        bufferevent的可读事件是边沿触发的。也就是说，如果客户端往服务器发了100字节的数据，而且客户端仅仅发送一次数据。那么服务器触发可读事件后，就应该把这100字节都读出来(假设这100字节是一起到达的)。不应该想着，这次回调只读4字节(比如是长度信息)，然后等到下次回调再读取其他数据。因为客户端只发送一次数据，所以不会再有下次回调了，即使bufferevent的缓冲区里面还有数据。

        当然，如果客户端再次发送数据，那么bufferevent的可读回调函数又会被调用。

        具体的原理可以参考<http://blog.csdn.net/luotuo44/article/details/39344743#t6>。

# ****bufferevent读事件的高水位是什么意思？****

        读事件的低水位比较容易理解，当bufferevent读缓冲区的数据到达这个低水位后，用户设置的可读回调函数才会被调用。比如说，用户设置了4字节的低水位，因为用户认为少于4字节都是不值得去处理的。当bufferevent的读缓冲区的数据量小于4字节时，并不会调用用户的可读回调函数。当数据量大于等于4字节时，就会调用用户的可读回调函数。

        读事件的高水位又是什么呢？在默认情况下（即没有设置高水位），一旦socket fd有数据可读了，那么libevent就会把数据从该socket fd的内核缓冲区 读取到bufferevent的读缓冲区中。客户端往服务器发送大量数据，服务器会不断地把数据copy到bufferevent缓冲区中。此时TCP的滑动窗口协议就没有用了。

        读事件的高水位此时就应运而生了，当bufferevent读缓冲区的数据量达到这个高水位后，就不再从socket fd中读取数据了。此时，socket fd的内核缓冲区会堆积大量数据，滑动窗口协议就起作用了。当bufferevent的读缓冲区的数量少于高水位后，libevent又可以从socket fd的缓冲区读取数据。停止读取、恢复读取这一系列操作都是由libevent负责完成，用户完全不知情。有的读者可能会想：既然已经监听了可读事件，那怎么做到socket 内核缓冲区既要保留数据，又能避免无休止地触发可读事件。Libevent的解决方法可以参考<http://blog.csdn.net/luotuo44/article/details/39344743#t5>。

# ****次线程调用bufferevent\_write，为什么不能发送数据？****

        此种情况，一般是主线程在event\_base\_dispatch中运行。用户想在次线程中调用bufferevent\_write发送数据。

        首先，确保你已经调用了evthread\_use\_pthreads函数(Windows平台为evthread\_use\_windows\_threads函数)。

        其次，确保你在是event\_base\_new函数之前调用的。

# 使用bufferevent时，为什么每次最多只能读取4096字节？

        如果客户端往服务器发送了大量的数据，并且服务器是使用bufferevent的。那么在bufferevent的读事件回调函数中，一般最多只能接收到4096个字节。这个是libevent这个库本身代码所限制的。

        libevent监听到一个socket fd可读后，就会去把数据从socket fd的内核缓冲区的数据copy到bufferevent内部的一个缓冲区里面。但libevent每次最多只copy 4096字节，即使socket fd的缓冲区里面有再多的数据。用户在读事件回调函数中读取数据，是从bufferevent内部的缓冲区读取的。所以最多只能读取4096字节。当然如果用户故意没有把这个4096字节读完，那么下次可以读取超过4096字节。

        对于某些情景，只copy 4096字节，性能是不够的。此时，只能修改libevent的源代码，然后重新编译。在后面的链接可以看到libevent为什么每次只从socket fd中读取4096字节，也在那里能修改之。<http://blog.csdn.net/luotuo44/article/details/39325447#t11>

# 在Linux中编译Libevent生成的四个静态库各有什么区别？

        在Linux中，编译Libevent，会产生下面这个静态库libevent.a、libevent\_core.a、libevent\_extra.a、libevent\_pthreads.a

        这四个静态库的区别是：

* event\_core.a： 包含Libevent的核心内容。比如event、buffer、bufferevent、log、epoll、evthread
* event\_extra.a： 包含Libevent额外提供的四大功能，为：event\_tagging、http、dns、rpc
* event\_pthreads.a： 包含了pthreads线程的具体实现
* event.a： event.a = event\_core + event\_extra

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/39547391