R6：缓冲事件之概念和基础

大多数情况下，除了响应事件外，程序还要执行一定数量的数据缓冲。例如，当我们需要写数据，通常的模式：

- 决定要向连接写一些数据，将数据放到缓冲区内。
- 等待连接变成可写的。
- 尽可能多地写数据。
- 记录写了多少数据，如果还有更多的数据需要写，等待连接再次可写。

这种缓冲 IO 模式很常见，Libevent 为此提供了一种通用机制。缓冲事件（`bufferevent`）由底层传输接口（例如套接字），读缓冲以及写缓冲组成。与底层传输接口可读或可写时回调的普通事件不同，缓冲事件在写或读了足够数据后调用用户提供的回调函数。

由很多类型的缓冲事件共享一套公共的接口，编写此文档时存在以下类型：

socket-based bufferevents

通过底层套接字发送和接收数据的缓冲事件，使用 `event_*` 接口作为后端。

asynchronous-IO bufferevents

通过 Windows IOCP 接口提供的底层套接字发送和接收数据的缓冲事件。（Windows 限定，实验性的。）

filtering bufferevents

向底层缓冲事件对象传递数据之前处理输入和输出数据的缓冲事件，例如压缩或转换数据。

paired bufferevents

相互传输数据的两个缓冲事件。

注意

对于 Libevent 2.0.2-alpha，缓冲事件的接口在所有的缓冲事件类型间仍不是完全正交。换句话说，以下描述的接口并不是每一个都适用于所有缓冲事件类型。Libevent 开发者打算在未来版本中修复这个问题。

还需注意

缓冲事件目前只能用于像 TCP 这样的面向流的协议。未来可能会支持类似于 UDP 的面向数据报的协议。

本小节介绍的所有函数和类型声明在 `<event/bufferevent.h>` 中。与缓冲区（`evbuffer`）相关的函数定义在 `<event2/buffer.h>` 中，更多信息请参考下一章。

缓冲事件和缓冲区

每一个缓冲事件有一个输入缓冲区和一个输出缓冲区，都是缓冲区类型的。当需要向缓冲事件写数据时，会将其加入输出缓冲区；当需要向缓冲事件读数据时，从输入缓冲区中抽取数据。

缓冲区接口支持很多操作，我们将会放到下一章讨论。

回调和阈值

缓冲事件有两个数据相关回调：一个读回调和一个写回调。默认情况下，只要从底层传输接口读取了数据就会调用读回调，只要足够多的数据从输出缓冲区清空到底层传输接口就会调用写回调。然而您可以通过调整读写 "阈值（watermarks，或翻译成水位）" 来覆盖这些函数的行为。

每个缓冲事件有四个阈值：

读低阈

每当缓冲事件的输入缓冲区数据达到或超过该阈值，则发生读回调。默认是 0，因此一旦读缓冲区内有数据就会调用读回调。

读高阈

如果缓冲事件的输入缓冲区数据达到该阈值，则停止读取。直到有足够的数据被抽取，使得缓冲区内数据低阈该阈值，才会继续读取。默认是无穷，因此绝对不会因为输入缓冲区内有太多数据而停止读取。

写低阈

如果输出缓冲区数据达到或低于该阈值，则发生写回调。默认是 0，所以只有输出缓冲区是空的时候才会调用写回调。

写高阈

缓冲事件不直接使用该值，该阈值在缓冲事件被用作另一个缓冲事件的底层传输接口时具有特殊含义。请参考下方过滤型缓冲事件。

缓冲事件同样拥有 "错误" 或 "事件" 回调，以便在连接关闭或发生错误等情况下通知通知应用发生了非数据事件。以下是定义的事件标志：

BEV_EVENT_READING

读操作时发生某事件，具体是什么事件请参考其他标志。

BEV_EVENT_WRITING

写操作时发生某事件，具体是什么事件请参考其他标志。

BEV_EVENT_ERROR

缓冲事件操作发生错误，关于错误的更多信息，可以调用 `EVUTIL_SOCKET_ERROR()`。

BEV_EVENT_TIMEOUT

缓冲事件超时。

BEV_EVENT_EOF

得到文件结束指示。

BEV_EVENT_CONNECTED

连接请求已完成。

（以上事件名由 Libevent 2.0.2-alpha 引入。）

延迟回调

默认情况下，缓冲事件的回调函数会在对应条件满足时立即调用。（缓冲区也是这样的，我们会在后面讨论。）在依赖关系很复杂时，立即调用可能会导致一些问题。例如，假设一个回调函数在缓冲区 A 空时填入数据，另一个回调函数在缓冲区 A 满时移除数据。由于这些调用都是发生在栈上的，当依赖关系非常复杂时，可能会有栈溢出的风险。

为了解决这个问题，您可以通知缓冲事件（或 缓冲区）需要延迟回调。当条件满足时，延迟回调不会立即调用回调函数，而是在 `event_loop()` 中排队，在常规事件回调之后执行。

（延迟回调由 Libevent 2.0.1-alpha 引入。）

缓冲事件选项标志

在创建缓冲事件时可以通过一个或多个标志改变其行为，可识别的标志有：

BEV_OPT_CLOSE_ON_FREE

当缓冲事件释放时，关闭其底层传输。这会关闭底层套接字，底层缓冲事件等等。

BEV_OPT_THREADSAFE

自动为缓冲事件分配锁，因此是多线程安全的。

BEV_OPT_DEFER_CALLBACKS

设置该标志后，缓冲事件会延迟回调，如同上面描述的。

BEV_OPT_UNLOCK_CALLBACKS

默认情况下，当缓冲事件被设置成线程安全，任何用户提供的回调函数被调用时都会上锁。设置此标志后 Libevent 会在调用您的回调函数时释放锁。

（Libevent 2.0.5-beta 引入了 `BEV_OPT_UNLOCK_CALLBACKS`，其他标志都是 Libevent 2.0.1-alpha 引入的。）

使用基于套接字的缓冲事件

最易于使用的缓冲事件是基于套接字的。基于套接字的缓冲事件使用 Libevent 底层事件机制来检测网络套接字是否可读且/或可写，并且使用底层网络调用（如 `readv`，`writev`，`WSASend` 或 `WSARecv`）来传输和接收数据。

创建基于套接字的缓冲事件

可以通过 `bufferevent_socket_new()` 来创建一个基于套接字的缓冲事件：

接口

```
struct bufferevent *bufferevent_socket_new(
    struct event_base *base,
    evutil_socket_t fd,
    enum bufferevent_options options);
```

`base` 参数是一个 `event_base`，`options` 是缓冲事件选项（`BEV_OPT_CLOSE_ON_FREE` 等）组成的比特字段，`fd` 是一个可选的套接字文件描述符。如果您想稍后设置文件描述符，则可以把 `fd` 先设成 -1。

> 技巧：[确保提供给 `bufferevent_socket_new` 的套接字是非阻塞的，Libevent 为此提供了方便的 `evutil_make_socket_nonblocking`。]

在基于套接字的缓冲事件上建立连接

如果缓冲事件的套接字还没有连接，您可以创建一个新的连接。

接口

```
int bufferevent_socket_connect(struct bufferevent *bev,
    struct sockaddr *address, int addrlen);
```

`address` 和 `addrlen` 参数与标准函数 `connect()` 一致。如果缓冲事件还没有设置套接字，调用该函数会为其分配一个新的非套接字。

如果缓冲事件已经设置了套接字，调用 `bufferevent_socket_connect()` 会告知 Libevent 该套接字还没有建立连接，直到连接建立完成前都不会有读写操作。

在建立连接之前将数据添加进输出缓冲区是合法的。

该函数在成功建立连接时返回 0，发生错误时返回 -1。

示例

```
#include <event2/event.h>
#include <event2/bufferevent.h>
#include <sys/socket.h>
#include <string.h>

void eventcb(struct bufferevent *bev, short events, void *ptr)
{
    if (events & BEV_EVENT_CONNECTED) {
         /* 连接到了 127.0.0.1:8080。通常我们会进行一些操作，
            例如开始读或写。 */
    } else if (events & BEV_EVENT_ERROR) {
         /* 连接时发生错误。（该标志其实并非表示连接时发生错误，
            而是缓冲事件进行某些操作时发生了错误，具体错误信息
            需要通过调用 EVUTIL_SOCKET_ERROR() 查看。） */
    }
}

int main_loop(void)
{
    struct event_base *base;
    struct bufferevent *bev;
    struct sockaddr_in sin;

    base = event_base_new();

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(0x7f000001); /* 127.0.0.1 */
    sin.sin_port = htons(8080); /* Port 8080 */

    bev = bufferevent_socket_new(base, -1, BEV_OPT_CLOSE_ON_FREE);

    bufferevent_setcb(bev, NULL, NULL, eventcb, NULL);

    if (bufferevent_socket_connect(bev,
        (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        /* 建立连接发生错误 */
        bufferevent_free(bev);
        return -1;
    }

    event_base_dispatch(base);
    return 0;
}
```

`bufferevent_socket_connect()` 函数由 Libevent 2.0.2-alpha 引入。在此之前，需要对套接字手动调用 `connect()`，完成连接后 Libevent 将报告可写。

注意如果使用 `bufferevent_socket_connect()` 建立连接，则只会收到 `BEV_EVENT_CONNECTED` 的事件；如果手动调用 `connect()`，则会收到可写事件。

如果您想自己调用 `connect()`，但在连接成功时仍然收到 `BEV_EVENT_CONNECTED` 事件，请在 `connect()` 返回 -1 且 `errno` 等于 `EAGAIN` 或 `EINPROGRESS` 后调用 `bufferevent_socket_connect(bev, NULL, 0)`。

`bufferevent_socket_connect` 函数于 Libevent 2.0.2-alpha 引入。

通过主机名连接

通常，您可以想将地址解析和建立连接合并成单个操作，为此 Libevent 提供了如下接口：

接口

```
int bufferevent_socket_connect_hostname(struct bufferevent *bev,
    struct evdns_base *dns_base, int family, const char *hostname,
    int port);
int bufferevent_socket_get_dns_error(struct bufferevent *bev);
```

这个函数解析主机 `hostname`，获取其 `family` 类型网络地址。（允许的地址族有 `AF_INET`，`AF_INET6` 和 `AF_UNSPEC`。）如果解析失败，则回调并报告错误，如果成功，则与 `bufferevent_socket_connect` 建立连接。

`dns_base` 参数是可选的。如果该参数是 `NULL`，则 Libevent 会在解析期间阻塞，通常您可能不希望出现这中情况。如果提供了 `dns_base`，则 Libevent 会用它来进行异步解析。关于 DNS 的更多内容，请参考第 9 章。

与 `bufferevent_socket_connect()` 一样，该函数通知 Libevent 目前缓冲事件还没有建立连接，在连接操作完成之前不要对套接字进行任何读写操作。

如果发生错误，可能是 DNS 解析错误。您可以通过调用 `bufferevent_socket_get_dns_error()` 获取最新的错误信息。如果返回的错误码是 0，则 DNS 没有发生错误。

示例：普通 HTTP 客户端 v0

```
/* 不要拷贝这份代码：这是一个低级的 HTTP 客户端实现。
   请查看 evhttp 相关内容。
*/
#include <event2/dns.h>
```

```c
#include <event2/bufferevent.h>
#include <event2/buffer.h>
#include <event2/util.h>
#include <event2/event.h>

#include <stdio.h>

void readcb(struct bufferevent *bev, void *ptr)
{
    char buf[1024];
    int n;
    struct evbuffer *input = bufferevent_get_input(bev);
    while ((n = evbuffer_remove(input, buf, sizeof(buf))) > 0) {
        fwrite(buf, 1, n, stdout);
    }
}

void eventcb(struct bufferevent *bev, short events, void *ptr)
{
    if (events & BEV_EVENT_CONNECTED) {
         printf("Connect okay.\n");
    } else if (events & (BEV_EVENT_ERROR|BEV_EVENT_EOF)) {
         struct event_base *base = ptr;
         if (events & BEV_EVENT_ERROR) {
                int err = bufferevent_socket_get_dns_error(bev);
                if (err)
                        printf("DNS error: %s\n", evutil_gai_strerror(err));
         }
         printf("Closing\n");
         bufferevent_free(bev);
         event_base_loopexit(base, NULL);
    }
}

int main(int argc, char **argv)
{
    struct event_base *base;
    struct evdns_base *dns_base;
    struct bufferevent *bev;

    if (argc != 3) {
        printf("Trivial HTTP 0.x client\n"
               "Syntax: %s [hostname] [resource]\n"
               "Example: %s www.google.com /\n",argv[0],argv[0]);
        return 1;
    }

    base = event_base_new();
```

```
    dns_base = evdns_base_new(base, 1);

    bev = bufferevent_socket_new(base, -1, BEV_OPT_CLOSE_ON_FREE);
    bufferevent_setcb(bev, readcb, NULL, eventcb, base);
    bufferevent_enable(bev, EV_READ|EV_WRITE);
    evbuffer_add_printf(bufferevent_get_output(bev), "GET %s\r\n", argv[2]);
    bufferevent_socket_connect_hostname(
        bev, dns_base, AF_UNSPEC, argv[1], 80);
    event_base_dispatch(base);
    return 0;
}
```

`bufferevent_socket_connect_hostname()` 在 Libevent 2.0.3-alpha 中首次出现；`bufferevent_socket_get_dns_error()` 在 2.0.5-beta 中首次出现。

通用 bufferevent 操作

释放 bufferevent

接口

```
void bufferevent_free(struct bufferevent *bev);
```

该函数用于释放缓冲事件。缓冲事件内部有索引计数，因此如果释放时还有等待中的延迟回调，会先回调再释放。

然而，`bufferevent_free()` 函数会尝试尽快释放缓冲事件。如果此时缓冲事件中有数据正在等待写（如写入文件或套接字），则在缓冲事件释放之前，缓冲区可能不会刷新（即待写数据丢失）。

如果设置了 `BEV_OPT_CLOSE_ON_FREE` 标志，并且该缓冲事件带有一个套接字或底层缓冲事件作为其传输接口，则传输接口会在缓冲事件释放时一并关闭。

该函数由 Libevent 0.8 引入。

回调、阈值和启用开关

接口

```
typedef void (*bufferevent_data_cb)(struct bufferevent *bev, void *ctx);
typedef void (*bufferevent_event_cb)(struct bufferevent *bev,
    short events, void *ctx);

void bufferevent_setcb(struct bufferevent *bufev,
```

```
    bufferevent_data_cb readcb, bufferevent_data_cb writecb,
    bufferevent_event_cb eventcb, void *cbarg);

void bufferevent_getcb(struct bufferevent *bufev,
    bufferevent_data_cb *readcb_ptr,
    bufferevent_data_cb *writecb_ptr,
    bufferevent_event_cb *eventcb_ptr,
    void **cbarg_ptr);
```

`bufferevent_setcb()` 函数改变缓冲事件的一个或多个回调函数。`readcb`，`writecb` 和 `eventcb` 分别在（缓冲事件）已经读取足够的数据（输入缓冲数据高于阈值），已经写入足够的数据（输出缓冲数据低于阈值）以及发生某些事件时回调。回调函数的第一个参数是触发了事件的缓冲事件，最后一个参数是调用 `bufferevent_setcb()` 时设置的 `cbarg` 参数：您可以通过这个参数向回调函数传递数据。事件回调（`bufferevent_event_cb`）的参数 `events` 是事件标志的比特掩码，参考上方的回调和阈值。

可以通过传递 `NULL` 来禁用某个回调。注意所有的回调函数都共享同一个 `cbarg` 值，改变这个值会影响所有回调函数。

您可以通过向 `bufferevent_getcb()` 传递指针以获取缓冲事件当前设置的回调函数，该函数会将 `*readcb_ptr` 设置成当前读回调，`*writecb_ptr` 设置成当前写回调，`*eventcb_ptr` 设置成当前事件回调，`*cbarg_ptr` 设置成当前回调的参数。（`bufferevent_getcb()` 中）设置成 `NULL` 的指针会被忽略。

`bufferevent_setcb()` 由 Libevent 1.4.4 引入，`bufferevent_data_cb` 和 `bufferevent_event_cb` 由 Libevent 2.0.2 引入，`bufferevent_getcb()` 由 Libevent 2.1.1-alpha 引入。

接口

```
void bufferevent_enable(struct bufferevent *bufev, short events);
void bufferevent_disable(struct bufferevent *bufev, short events);

short bufferevent_get_enabled(struct bufferevent *bufev);
```

您可以开启或关闭缓冲事件的 `EV_READ`，`EV_WRITE` 或 `EV_READ|EV_WRITE` 事件。如果没有开启读和写，缓冲事件不会尝试读写数据。

输出缓冲为空时没有必要关闭写事件：缓冲事件会自动停止写数据，并在有数据可写时重新开始写。

同样，输入缓冲区到达读高阈时无需关闭读事件：缓冲事件会自动停止读取，并在有足够空间时继续读取。

默认情况下，刚创建的缓冲事件开启了写事件，但没有开启读事件。

可以通过调用 `bufferevent_get_enabled()` 获取缓冲事件当前开启的事件。

这些函数由 Libevent 0.8 引入，除 `bufferevent_get_enabled()` 外，该函数是 2.0.3-alpha 引入的。

接口

```
void bufferevent_setwatermark(struct bufferevent *bufev, short events,
    size_t lowmark, size_t highmark);
```

`bufferevent_setwatermark()` 函数会调整一个缓冲事件的读阈值或写阈值，或两者都调整。（如果 `events` 设置了 `EV_READ`，则会调整读阈值，如果 `events` 设置了 `EV_WRITE`，则会调整写阈值。）

高阈设置成 0 意味着 "无穷"。

该函数由 Libevent 1.4.4 首次引入。

示例

```
#include <event2/event.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>
#include <event2/util.h>

#include <stdlib.h>
#include <errno.h>
#include <string.h>

struct info {
    const char *name;
    size_t total_drained;
};

void read_callback(struct bufferevent *bev, void *ctx)
{
    struct info *inf = ctx;
    struct evbuffer *input = bufferevent_get_input(bev);
    size_t len = evbuffer_get_length(input);
    if (len) {
        inf->total_drained += len;
        evbuffer_drain(input, len);
        printf("Drained %lu bytes from %s\n",
```

```
            (unsigned long) len, inf->name);
    }
}

void event_callback(struct bufferevent *bev, short events, void *ctx)
{
    struct info *inf = ctx;
    struct evbuffer *input = bufferevent_get_input(bev);
    int finished = 0;

    if (events & BEV_EVENT_EOF) {
        size_t len = evbuffer_get_length(input);
        printf("Got a close from %s.  We drained %lu bytes from it, "
            "and have %lu left.\n", inf->name,
            (unsigned long)inf->total_drained, (unsigned long)len);
        finished = 1;
    }
    if (events & BEV_EVENT_ERROR) {
        printf("Got an error from %s: %s\n",
            inf->name, evutil_socket_error_to_string(EVUTIL_SOCKET_ERROR()));
        finished = 1;
    }
    if (finished) {
        free(ctx);
        bufferevent_free(bev);
    }
}

struct bufferevent *setup_bufferevent(void)
{
    struct bufferevent *b1 = NULL;
    struct info *info1;

    info1 = malloc(sizeof(struct info));
    info1->name = "buffer 1";
    info1->total_drained = 0;

    /* ... 这里我们应当设置缓冲事件并确保建立连接 ... */

    /* 输入缓冲区至少有 128 字节数据时才会触发读回调 */
    bufferevent_setwatermark(b1, EV_READ, 128, 0);

    bufferevent_setcb(b1, read_callback, NULL, event_callback, info1);

    bufferevent_enable(b1, EV_READ); /* Start reading. */
    return b1;
}
```

处理缓冲事件的数据

Reading and writing data from the network does you no good if you can't look at it.（噢，我的老天爷，这句话要怎么翻译才显得不那么蹩脚哩？）缓冲事件提供了以下方法来获取要写和要读的数据：

接口

```
struct evbuffer *bufferevent_get_input(struct bufferevent *bufev);
struct evbuffer *bufferevent_get_output(struct bufferevent *bufev);
```

这两个函数是非常强大的基础接口：它们分别返回输入和输出缓冲区。关于可以对缓冲区做的所有操作，请参考下一章。

注意程序只能移除输入缓冲区中的数据（而非添加数据），也只能向输出缓冲区中添加数据（而非移除数据）。

如果写操作因数据量太少而停止（或读操作因数据量太多而停止），向输出缓冲区添加数据（或移除输入缓冲区中的数据）会自动重启操作。（这里指的应当是输出缓冲区内的数据量太少，需要等待应用向缓冲区写入一定量的数据再一起写入底层传输接口，或输入缓冲区数据太多，需等待应用读取并移除一定量的数据再从底层传输接口读取数据。）

这些函数由 Libevent 2.0.1-alpha 引入。

接口

```
int bufferevent_write(struct bufferevent *bufev,
    const void *data, size_t size);
int bufferevent_write_buffer(struct bufferevent *bufev,
    struct evbuffer *buf);
```

这些函数向缓冲事件的输出缓冲区写入数据。调用 `bufferevent_write()` 将 `data` 起的 `size` 字节数据添加至输出缓冲区的尾部。调用 `bufferevent_write_buffer()` 将 `buf` 的所有数据添加至输出缓冲区的尾部。两者都在成功时返回 0，发生错误时返回 -1。

这些函数自 Libevent 0.8 就存在。

接口

```
size_t bufferevent_read(struct bufferevent *bufev, void *data, size_t size);
int bufferevent_read_buffer(struct bufferevent *bufev,
```

```
    struct evbuffer *buf);
```

这些函数从缓冲事件的输入缓冲区移除数据。`bufferevent_read()` 函数从输入缓冲区移除 `size` 字节数据，存入 `data`，返回实际移动的字节数。`bufferevent_read_buffer()` 函数将输入缓冲区的数据全部移入 `buf`，成功返回 0，失败返回 -1。

注意对于 `bufferevent_read()`，内存块 `data` 必须有足够的空间存储 `size` 字节数据。

`bufferevent_read()` 函数自 Libevent 0.8 便存在，`bufferevent_read_buffer()` 由 Libevent 2.0.1-alpha 引入。

示例

```
#include <event2/bufferevent.h>
#include <event2/buffer.h>

#include <ctype.h>

void
read_callback_uppercase(struct bufferevent *bev, void *ctx)
{
        /* 这个回调函数每次从 bev 的输入缓冲区移除 128 字节数据，转成大写，然后发送回去。

           （当心！实际上不应该在网络协议中使用 toupper，除非您知道当前语言环境就是您想
            使用的语言环境。）
         */

        char tmp[128];
        size_t n;
        int i;
        while (1) {
                n = bufferevent_read(bev, tmp, sizeof(tmp));
                if (n <= 0)
                        break; /* No more data. */
                for (i=0; i<n; ++i)
                        tmp[i] = toupper(tmp[i]);
                bufferevent_write(bev, tmp, n);
        }
}

struct proxy_info {
        struct bufferevent *other_bev;
};
void
read_callback_proxy(struct bufferevent *bev, void *ctx)
{
```

```
        /* 如果您正在实现一个简单的代理：从一个连接中获取数据（bev），写入另一个连接，
           要求尽可能少的拷贝。那么您可能需要一个这样的函数。 */
        struct proxy_info *inf = ctx;

        bufferevent_read_buffer(bev,
            bufferevent_get_output(inf->other_bev));
}

struct count {
        unsigned long last_fib[2];
};

void
write_callback_fibonacci(struct bufferevent *bev, void *ctx)
{
        /* 这是一个将一些斐波那契数列写进 bev 输出缓冲区的回调函数。这个函数会在写入 1k
           数据时停止，如果这些数据传输完了，则添加更多数据。 */
        struct count *c = ctx;

        struct evbuffer *tmp = evbuffer_new();
        while (evbuffer_get_length(tmp) < 1024) {
                unsigned long next = c->last_fib[0] + c->last_fib[1];
                c->last_fib[0] = c->last_fib[1];
                c->last_fib[1] = next;

                evbuffer_add_printf(tmp, "%lu", next);
        }

        /* 现在我们将 tmp 的所有内容写进 bev。 */
        bufferevent_write_buffer(bev, tmp);

        /* 我们不再需要 tmp 了。 */
        evbuffer_free(tmp);
}
```

读写超时

与其他事件一样，若缓冲事件在指定时间内没有完成任何读或写操作，则可以调用一个超时回调。

接口

```
void bufferevent_set_timeouts(struct bufferevent *bufev,
    const struct timeval *timeout_read, const struct timeval *timeout_write);
```

将超时设成 `NULL` 即移除超时，然而在 Libevent 2.1.2-alpha 之前这对任何事件类型都不起作用。（作为旧版本的解决方案，如果您不需要超时，可以尝试将超时设置成好几天并/或让回调函数忽略 `BEV_TIMEOUT` 事件。）

缓冲事件在尝试读数据时等待超过 `timeout_read` 秒后将会触发读超时，在尝试写数据时等待超过 `timeout_write` 秒后将会触发写超时。

注意只有在缓冲事件需要读或写时才会计时。换句话说，如果读操作没有被开启，或者输入缓冲区满了（超过读高阈），读超时是不会开启的。相似的，写超时在未开启写操作或没有数据需要写时不会起作用。

当读或写超时触发时，缓冲事件会关闭相应的读或写操。随后事件回调将会被调用，`BEV_EVENT_TIMEOUT|BEV_EVENT_READING` 或 `BEV_EVENT_TIMEOUT|BEV_EVENT_WRITING` 会被置位。

该函数自 Libevent 2.0.1-alpha 便存在，但直到 2.0.4-alpha 版本才对各种各种缓冲事件表现出一致的行为。

刷新缓冲事件

接口

```
int bufferevent_flush(struct bufferevent *bufev,
    short iotype, enum bufferevent_flush_mode state);
```

刷新缓冲事件会告知缓冲事件忽略其他限制，强制从底层传输接口读取或写入尽可能多的数据。该函数的功能细节与缓冲事件的类型有关。

`iotype` 参数应当是 `EV_READ`，`EV_WRITE` 或 `EV_READ|EV_WRITE`，以指明应当处理读还是写，或者二者均处理。`state` 参数可能是 `BEV_NORMAL`，`BEV_FLUSH` 或 `BEV_FINISHED` 中的一个。`BEV_FINISHED` 表示应当告知另一端没有更多数据需要发送了，`BEV_NORMAL` 和 `BEV_FLUSH` 之间的区别则与缓冲事件的类型有关。

`bufferevent_flush()` 函数失败时返回 -1，没有数据需要刷新时返回 0，刷新了数据则返回 1。

当前（自 2.0.5-beta）仅有一些缓冲事件类型实现了 `bufferevent_flush()`。特别是，基于套接字的缓冲事件还没有实现。

类型特定的缓冲事件函

这些函数并不支持所有类型的缓冲事件。

接口

```
int bufferevent_priority_set(struct bufferevent *bufev, int pri);
int bufferevent_get_priority(struct bufferevent *bufev);
```

该函数（`bufferevent_priority_set`）将 `bufev` 的优先级调整为 `pri`，关于优先级的更多信息请参考 `event_priority_set()`。

该函数成功返回 0，失败返回 -1，且只对套接字类型的缓冲事件有效。

`bufferevent_priority_set()` 函数于 Libevent 1.0 引入，`bufferevent_get_priority()` 于 Libevent 2.1.2-alpha 引入。

接口

```
int bufferevent_setfd(struct bufferevent *bufev, evutil_socket_t fd);
evutil_socket_t bufferevent_getfd(struct bufferevent *bufev);
```

这些函数为文件类型缓冲事件设置或返回文件描述符。只有套接字类型的缓冲事件支持 `setfd()`。这两个函数都在失败时返回 -1，`setfd()` 成功时返回 0.

`bufferevent_setfd()` 函数由 Libevent 1.4.4 引入，`bufferevent_getfd()` 函数由 Libevent 2.0.2-alpha 引入。

接口

```
struct event_base *bufferevent_get_base(struct bufferevent *bev);
```

该函数返回缓冲事件的 `event_base`，于 Libevent 2.0.9-rc 引入。

接口

```
struct bufferevent *bufferevent_get_underlying(struct bufferevent *bufev);
```

该函数返回底层作为传输接口的缓冲事件。关于何时会发生这种情况的信息，请参考关于过滤型缓冲事件的说明。

该函数于 Libevent 2.0.2-alpha 引入。

手动锁定和解锁

与缓冲区一样，有时您希望确保对缓冲事件的许多操作是原子的。Libevent 提供了一些可以手动锁定和解锁缓冲事件的函数。

接口

```
void bufferevent_lock(struct bufferevent *bufev);
void bufferevent_unlock(struct bufferevent *bufev);
```

注意如果创建缓冲事件时没有设置 `BEV_OPT_THREADSAFE` 或没有启用 Libeven 对多线程的支持，则锁定缓冲事件是没有任何效果的。

使用该函数锁定缓冲事件会一并锁定其关联的缓冲区。这些函数是可递归的：持有锁时再次对缓冲事件上锁是安全的。当然，您必须要对每一次上锁操作调用解锁。

这些函数于 Libevent 2.0.6-rc 引入。

过时的缓冲事件函数

从 Libevent 1.4 到 Libevent 2.0，缓冲事件的后端代码一致在修改。在旧的接口中，有时候访问缓冲事件结构体的内部是很正常的，并且还会使用依赖于这种访问的宏。

更复杂的是，旧代码有时会使用以 `evbuffer` 开头的缓冲事件函数。

以下是一个在 2.0 版本之前使用过的函数的概要：

目前名称	旧名称
bufferevent_data_cb	evbuffercb
bufferevent_event_cb	everrorcb
BEV_EVENT_READING	EVBUFFER_READ
BEV_EVENT_WRITE	EVBUFFER_WRITE
BEV_EVENT_EOF	EVBUFFER_EOF
BEV_EVENT_ERROR	EVBUFFER_ERROR
BEV_EVENT_TIMEOUT	EVBUFFER_TIMEOUT
bufferevent_get_input(b)	EVBUFFER_INPUT(b)
bufferevent_get_output(b)	EVBUFFER_OUTPUT(b)

旧函数定义在 `<event.h>` 而不是 `<event2/bufferevent.h>` 中。

如果您仍需要访问缓冲事件结构体的公共部分，您可以包含 `<event2/bufferevent_struct.h>` 。我们不推荐这么做：不同版本的 Libevent 中缓冲事件结构体的内容可能有变化。本小节的宏和名称只有在包含 `<event2/bufferevent_compat.h>` 时才能使用。

较老版本的设置缓冲事件的接口有些不同：

接口

```
struct bufferevent *bufferevent_new(evutil_socket_t fd,
    evbuffercb readcb, evbuffercb writecb, everrorcb errorcb, void *cbarg);
int bufferevent_base_set(struct event_base *base, struct bufferevent *bufev);
```

`bufferevent_new()` 只在废弃的 "默认（当前）" `event_base` 上创建套接字型缓冲事件。调用 `bufferevent_base_set` 只会调整套接字型缓冲事件的 `event_base` 。

不同于用 `timeval` 结构体设置超时，这些函数只使用一个代表秒的数字：

接口

```
void bufferevent_settimeout(struct bufferevent *bufev,
    int timeout_read, int timeout_write);
```

最后，注意 Libevent 2.0 之前的底层缓冲区的实现效率是非常低的，这对将缓冲时间用于高性能应用来说是一个问题。