R7：缓冲 IO 实用功能——缓冲区

Libevent 的缓冲区（`evbuffer`）功能实现了一个字节队列，优化了向尾部添加数据以及从头部移除数据的功能。

缓冲区通常用作缓冲网络 IO 的 "缓冲" 部分，缓冲区不提供调度 IO 或在准备就绪时触发 IO：这是缓冲事件该做的。

除非有额外注明，本小节的函数均声明在 `<event2/buffer.h>` 中。

创建或释放缓冲区

接口

```
struct evbuffer *evbuffer_new(void);
void evbuffer_free(struct evbuffer *buf);
```

这些函数将对简明：`evbuffer_new()` 分配并返回一个空的缓冲区，`evbuffer_free()` 删除缓冲区及其所有组件。

这些函数自 Libevent 0.8 便存在。

缓冲区和线程安全

接口

```
int evbuffer_enable_locking(struct evbuffer *buf, void *lock);
void evbuffer_lock(struct evbuffer *buf);
void evbuffer_unlock(struct evbuffer *buf);
```

默认情况下，多线程同时访问缓冲区是不安全的。如果需要多线程访问，可以对缓冲区调用 `evbuffer_enable_locking()`。如果 `lock` 参数是 `NULL`，Libevent 调用先前通过 `evthread_set_lock_creation_callback` 设置的锁创建函数分配一个新锁，否则使用该参数作为锁。

`evbuffer_lock()` 和 `evbuffer_unlock()` 函数分别用于对缓冲区请求和释放锁，您可以通过这些函数来保证一些操作的原子性。如果缓冲区没有启用锁，这些函数不会有任何效果。

（注意，对于单个操作，不需要调用 `evbuffer_lock()` 和 `evbuffer_unlock()`：如果缓冲区启用了锁，则单个操作已经是原子的。只有在执行多个操作且需要排除其他线程介入时，才需要手动锁定缓冲区。）

这些函数于 Libevent 2.0.1-alpha 引入。

检查缓冲区

接口

```
size_t evbuffer_get_length(const struct evbuffer *buf);
```

该函数以字节为单位返回存储在缓冲区中的数据长度。

该函数由 Libevent 2.0.1-alpha 引入。

接口

```
size_t evbuffer_get_contiguous_space(const struct evbuffer *buf);
```

该函数返回在缓冲区头部连续存储的数据的字节数。缓冲区中的数据可能存储在好几个不连续的内存块，这个函数返回缓冲区当前第一个区块存储的字节数。

该函数于 Libevent 2.0.1-alpha 引入。

向缓冲区添加数据：基础

接口

```
int evbuffer_add(struct evbuffer *buf, const void *data, size_t datlen);
```

该函数把 `data` 中的 `datlen` 字节长度数据添加到 `buf` 的末端，成功返回 0，失败返回 -1。

接口

```
int evbuffer_add_printf(struct evbuffer *buf, const char *fmt, ...);
int evbuffer_add_vprintf(struct evbuffer *buf, const char *fmt, va_list ap);
```

这些函数向 `buf` 的尾部添加格式化数据，格式化参数和其他参数的处理分别和 C 库的 `printf` 以及 `vprintf` 一致。这些函数返回向缓冲区中添加的字节数。

接口

```
int evbuffer_expand(struct evbuffer *buf, size_t datlen);
```

该函数修改缓冲区的最后一个地址块，或新增一个地址块，这样缓冲区就足够包含 `datlen` 字节数据，而无需进一步分配。

示例

```
/* 有两种方法向缓冲区添加 "Hello world 2.0.1"。 */
/* 直接添加： */
evbuffer_add(buf, "Hello world 2.0.1", 17);

/* 通过 printf： */
evbuffer_add_printf(buf, "Hello %s %d.%d.%d", "world", 2, 0, 1);
```

`evbuffer_add()` 和 `evbuffer_add_printf()` 函数由 Libevent 0.8 引入，`evbuffer_expand()` 由 Libevent 0.9 引入，`evbuffer_add_vprintf()` 首次出现在 Libevent 1.1 中。

将数据转移到另一个缓冲区

为保证效率，Libevent 提供了将数据转移到另一个缓冲区的函数。

接口

```
int evbuffer_add_buffer(struct evbuffer *dst, struct evbuffer *src);
int evbuffer_remove_buffer(struct evbuffer *src, struct evbuffer *dst,
    size_t datlen);
```

`evbuffer_add_buffer()` 函数将 `src` 的所有数据移动到 `dst` 中，成功返回 0，失败返回 -1。

`evbuffer_remove_buffer()` 函数将 `src` 中 `datlen` 字节数据移动到 `dst` 的末端，并且尽可能避免拷贝。如果可移动的数据小于 `datlen`，则会移动所有数据。该函数返回移动的字节数。

`evbuffer_add_buffer()` 于 Libevent 0.8 引入，`evbuffer_remove_buffer()` 是 Libevent 2.0.1-alpha 新引入的。

向缓冲区头部添加数据

接口

```
int evbuffer_prepend(struct evbuffer *buf, const void *data, size_t size);
int evbuffer_prepend_buffer(struct evbuffer *dst, struct evbuffer* src);
```

这些函数行为与 `evbuffer_add()` 和 `evbuffer_add_buffer()` 相似，不同的是将数据添加进目标缓冲区的头部。

重排缓冲区内部布局

有时可能需要取出缓冲区的前 `N` 字节，并将其看作一个连续的数组。要达到此目的，您必须确保缓冲区的头部是连续的。

接口

```
unsigned char *evbuffer_pullup(struct evbuffer *buf, ev_ssize_t size);
```

`evbuffer_pullup()` 函数将缓冲区的前 `size` 字节 "线性化"，必要时进行复制或移动，以保证这些字节是连续的，占据相同的内存块。如果 `size` 是负的，该函数会线性化整个缓冲区。如果 `size` 比缓冲区的字节数更大，则返回 `NULL`，否则返回指向缓冲区第一个字节的指针。

调用 `evbuffer_pullup()` 时使用大的 `size` 会非常慢，因为可能需要拷贝整个缓冲区中的内容。

示例

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

#include <string.h>

int parse_socks4(struct evbuffer *buf, ev_uint16_t *port, ev_uint32_t *addr)
{
    /* 让我们来分析 SOCKS4 请求的头部！格式很简单：
     * 1 字节版本号，1 字节命令，2 字节目标端口，
     * 4 字节目标 ip。 */
    unsigned char *mem;

    mem = evbuffer_pullup(buf, 8);

    if (mem == NULL) {
        /* Not enough data in the buffer */
        return 0;
    } else if (mem[0] != 4 || mem[1] != 1) {
```

```
        /* Unrecognized protocol or command */
        return -1;
    } else {
        memcpy(port, mem+2, 2);
        memcpy(addr, mem+4, 4);
        *port = ntohs(*port);
        *addr = ntohl(*addr);
        /* Actually remove the data from the buffer now that we know we
           like it. */
        evbuffer_drain(buf, 8);
        return 1;
    }
}
```

注意：调用 `evbuffer_pullup()` 时使用的 `size` 如果与 `evbuffer_get_contiguous_space()` 的返回值一样，则不会有任何的拷贝或移动。

`evbuffer_pullup()` 函数于 Libevent 2.0.1-alhpa 引入，更早版本的 Libevent 总是保持数据连续，而无视开销。

从缓冲区移除数据

接口

```
int evbuffer_drain(struct evbuffer *buf, size_t len);
int evbuffer_remove(struct evbuffer *buf, void *data, size_t datlen);
```

`evbuffer_remove()` 函数拷贝并移除 `buf` 的前 `datlen` 字节数据到 `data` 中。如果缓冲区中的数据少于 `datlen` 字节，则会拷贝所有数据。该函数失败时返回 -1，否则返回拷贝的字节数。

`evbuffer_drain()` 函数行为与 `evbuffer_remove()` 函数类似，但是不会拷贝数据：该函数仅仅移除缓冲区首部数据，成功返回 0，失败返回 -1。

从缓冲区拷贝数据

有时可能需要拷贝缓冲区首部数据，但不希望移除这些数据。例如，您可能希望查看某种类型的完整记录是否已经到达，而不移除任何数据（如 `evbuffer_remove` 所做的），或者重排内部布局（如 `evbuffer_pullup()` 所做的）。

接口

```
ev_ssize_t evbuffer_copyout(struct evbuffer *buf, void *data, size_t datlen);
ev_ssize_t evbuffer_copyout_from(struct evbuffer *buf,
     const struct evbuffer_ptr *pos,
     void *data_out, size_t datlen);
```

`evbuffer_copyout()` 函数的行为与 `evbuffer_remove()` 类似，只不过不会将数据从缓冲区移除。即，拷贝 `buf` 的前 `datlen` 字节数据到到 `data`。如果数据量小于 `datlen` 字节，则会拷贝所有字节。该函数失败返回 -1，成功返回拷贝的字节数。

`evbuffer_copyout_from()` 函数行为与 `evbuffer_copyout()` 类似，只不过是从缓冲区指针 `pos` 开始拷贝，而非首部开始。有关 `evbuffer_ptr`，请参考 "在缓冲区内搜索" 中相关结构体。

如果从缓冲区拷贝数据太慢，请使用 `evbuffer_peek()` 替代。

示例

```
#include <event2/buffer.h>
#include <event2/util.h>
#include <stdlib.h>
#include <stdlib.h>

int get_record(struct evbuffer *buf, size_t *size_out, char **record_out)
{
    /* 假设我们讨论的是某种协议，其中包含一个按网络顺序排列的 4 字节大小的字段，
       后面跟着字节数。如果记录完整，我们返回 1 并设置 'out' 字段；如果记录不
       完整，则返回 0；发生错误则返回 -1。 */
    size_t buffer_len = evbuffer_get_length(buf);
    ev_uint32_t record_len;
    char *record;

    if (buffer_len < 4)
       return 0; /* size 字段还未到达。 */

   /* 使用 evbuffer_copyout，因此 size 字段还保留在缓冲区中。 */
    evbuffer_copyout(buf, &record_len, 4);
    /* 将 len_buf 转换成主机字节序 */
    record_len = ntohl(record_len);
    if (buffer_len < record_len + 4)
        return 0; /* 记录还不完整 */

    /* 好了，现在我们可以移除记录。 */
    record = malloc(record_len);
    if (record == NULL)
        return -1;
```

```
    evbuffer_drain(buf, 4);
    evbuffer_remove(buf, record, record_len);

    *record_out = record;
    *size_out = record_len;
    return 1;
}
```

`evbuffer_copyout()` 函数首次出现在 Libevent 2.0.5-alpha 中，`evbuffer_copyout_from()` 首次出现在 Libevent 2.1.1-alpha 中。

面向行的输入

接口

```
enum evbuffer_eol_style {
        EVBUFFER_EOL_ANY,
        EVBUFFER_EOL_CRLF,
        EVBUFFER_EOL_CRLF_STRICT,
        EVBUFFER_EOL_LF,
        EVBUFFER_EOL_NUL
};
char *evbuffer_readln(struct evbuffer *buffer, size_t *n_read_out,
    enum evbuffer_eol_style eol_style);
```

很多网络协议使用基于行的格式。`evbuffer_readln()` 函数从缓冲区头部提取一行并放置在新分配的，以 `NUL` 结尾的字符串中。如果 `n_read_out` 不是 `NULL`，则 `*n_read_out` 会被设置成字符串的字节数。如果无法读取一整行，该函数返回 `NULL`。行结束符不会被拷贝到字符串中。

`evbuffer_readln()` 函数可以识别 4 种行结束格式：

EVBUFFER_EOL_LF

行尾是一个换行符。（也就是 "\n"，ASCII 码是 0x0A。）

EVBUFFER_EOL_CRLF_STRICT

行尾是一个回车符接着一个换行符。（也就是 "\r\n"，ASCII 码是 0x0D 0x0A）。

EVBUFFER_EOL_CRLF

行尾是一个可选的回车符，接着一个换行符。（换句话说，既可以是 "\r\n"，也可以是 "\n"。）这种格式对分析基于文本的网络协议很有用，因为标准通常要求以 "\r\n" 结尾，而不遵循标准的的客户端有时只使用 "\n"。

EVBUFFER_EOL_ANY

行尾可以是任意数量的回车符和任意数量的换行符的任意序列。这种格式并不很有用，其存在只是为了向后兼容。

EVBUFFER_EOL_NUL

行尾是一个值为 0 的字节，即 ASCII NUL。

（注意如果使用 `event_set_mem_functions()` 覆盖默认 `malloc`，`evbuffer_readln` 返回的字符串会由指定的 `malloc` 函数分配。

示例

```
char *request_line;
size_t len;

request_line = evbuffer_readln(buf, &len, EVBUFFER_EOL_CRLF);
if (!request_line) {
    /* 第一行还没有到达。 */
} else {
    if (!strncmp(request_line, "HTTP/1.0 ", 9)) {
        /* 检测到了 HTTP 1.0 */
    }
    free(request_line);
}
```

`evbuffer_readln()` 接口自 Libevent 1.4.14-stable 可用。`EVBUFFER_EOL_NUL` 是 Libevent 2.1.1-alpha 新增的。

在缓冲区内搜索

`evbuffer_ptr` 结构体指向缓冲区内部位置，并包含可用于在缓冲区中迭代的数据。

接口

```
struct evbuffer_ptr {
        ev_ssize_t pos;
        struct {
                /* 内部字段 */
```

```
        } _internal;
};
```

`pos` 是仅有的公开字段，其他部分不应在用户代码中使用。该字段作为起始偏移指向缓冲区的某个位置。

接口

```
struct evbuffer_ptr evbuffer_search(struct evbuffer *buffer,
    const char *what, size_t len, const struct evbuffer_ptr *start);
struct evbuffer_ptr evbuffer_search_range(struct evbuffer *buffer,
    const char *what, size_t len, const struct evbuffer_ptr *start,
    const struct evbuffer_ptr *end);
struct evbuffer_ptr evbuffer_search_eol(struct evbuffer *buffer,
    struct evbuffer_ptr *start, size_t *eol_len_out,
    enum evbuffer_eol_style eol_style);
```

`evbuffer_search()` 函数扫描缓冲区以搜索 `len` 长字符串 `what`。该函数返回一个 `evbuffer_ptr` 指向该字符串，如果没有找到则返回 -1。如果提供了 `start` 参数，则从该位置开始搜索，否则从起始位置开始搜索。

`evbuffer_search_range()` 函数行为与 `evbuffer_search` 一致，不同的是该函数会在 `end` 之前搜索。

`evbuffer_search_eol()` 函数会以与 `evbuffer_readln()` 相同的方式检测行尾字符，但是不会拷贝内容，而是返回一个指向行尾字符的 `evbuffer_ptr`。如果 `eol_len_out` 不是 `NULL`，则会被设置成 `EOL` 字符串的长度。

接口

```
enum evbuffer_ptr_how {
        EVBUFFER_PTR_SET,
        EVBUFFER_PTR_ADD
};
int evbuffer_ptr_set(struct evbuffer *buffer, struct evbuffer_ptr *pos,
    size_t position, enum evbuffer_ptr_how how);
```

`evbuffer_ptr_set` 函数设置缓冲区的内部指针 `pos`。如果 `how` 是 `EVBUFFER_PTR_SET`，则指针会指向缓冲区的绝对位置 `position` 处，如果是 `EVBUFFER_PTR_ADD`，则指针向前移动 `position` 字节。该函数成功返回 0，失败返回 -1。

示例

```c
#include <event2/buffer.h>
#include <string.h>

/* 统计 buf 中 str 出现次数。 */
int count_instances(struct evbuffer *buf, const char *str)
{
    size_t len = strlen(str);
    int total = 0;
    struct evbuffer_ptr p;

    if (!len)
        /* 请不要尝试统计长度为 0 的字符串的出现次数。 */
        return -1;

    evbuffer_ptr_set(buf, &p, 0, EVBUFFER_PTR_SET);

    while (1) {
         p = evbuffer_search(buf, str, len, &p);
         if (p.pos < 0)
             break;
         total++;
         evbuffer_ptr_set(buf, &p, 1, EVBUFFER_PTR_ADD);
    }

    return total;
}
```

警告：任何修改缓冲区或其布局的调用都会使 `evbuffer_ptr` 失效，不能再安全地使用。

这些接口都是 Libevent 2.0.1-alpha 新增的。

无拷贝检查数据

有时可能需要在不拷贝的前提下读取缓冲区的数据（如 `evbuffer_copyout()`），并且不重排缓冲区内部内存（如 `evbuffer_pullup()`）。有时可能需要查看缓冲区中间部分的数据。

您可以这样做：

接口

```c
struct evbuffer_iovec {
        void *iov_base;
        size_t iov_len;
};
```

```
int evbuffer_peek(struct evbuffer *buffer, ev_ssize_t len,
    struct evbuffer_ptr *start_at,
    struct evbuffer_iovec *vec_out, int n_vec);
```

调用 `evbuffer_peek()` 时，需要提供一个 `evbuffer_iovec` 类型的数组 `vec_out`。该数组的长度是 `n_vec`。该函数会将数组中的元素设置成指向缓冲区内存区块的指针（`iov_base`），并将 `iov_len` 设置成对应区块的长度。

如果 `len` 小于 0，`evbuffer_peek()` 会尝试填充 `vec_out` 中的所有元素。否则，该函数会填充至 `vec_out` 所有元素均被使用，或缓冲区中至少 `len` 比特数据可见。如果该函数可以给出请求的所有数据，则返回 `evbuffer_iovec` 中实际使用的元素数量。否则，函数返回给出所有请求数据所需的元素个数。

如果 `ptr` 是 `NULL`，`evbuffer_peek()` 从缓冲区首部开始，否则从 `ptr` 开始。

示例

```
{
    /* 查看 buf 的前两个内存区块，并写进 stderr。 */
    int n, i;
    struct evbuffer_iovec v[2];
    n = evbuffer_peek(buf, -1, NULL, v, 2);
    for (i=0; i<n; ++i) { /* 可见的区块数量可能会少于 2 个。 */
        fwrite(v[i].iov_base, 1, v[i].iov_len, stderr);
    }
}

{
    /* 通过 write 将前 4096 字节数据发送到 stdout。 */
    int n, i, r;
    struct evbuffer_iovec *v;
    size_t written = 0;

    /* 检测需要多少个区块。 */
    n = evbuffer_peek(buf, 4096, NULL, NULL, 0);
    /* 为区块分配空间，如果 alloca() 可用，那么现在是使用这个函数的好时机。 */
    v = malloc(sizeof(struct evbuffer_iovec)*n);
    /* 实际填充 v。 */
    n = evbuffer_peek(buf, 4096, NULL, v, n);
    for (i=0; i<n; ++i) {
        size_t len = v[i].iov_len;
        if (written + len > 4096)
            len = 4096 - written;
        r = write(1 /* stdout */, v[i].iov_base, len);
        if (r<=0)
```

```
            break;
        /* 每次都要追踪写了多少数据，如果不这么做，写入最后一块数据的时候可能会
           导致写入数据超过 4096 字节。 */
        written += len;
    }
    free(v);
}

{
    /* 获取 "start\n" 出现后的前 16K 数据，并通过 consume() 函数传递。 */
    struct evbuffer_ptr ptr;
    struct evbuffer_iovec v[1];
    const char s[] = "start\n";
    int n_written;

    ptr = evbuffer_search(buf, s, strlen(s), NULL);
    if (ptr.pos == -1)
        return; /* 没有找到开始字符串。 */

    /* 将指针移动到开始字符串之后。 */
    if (evbuffer_ptr_set(buf, &ptr, strlen(s), EVBUFFER_PTR_ADD) < 0)
        return; /* 开始字符串后没有数据。 */

    while (n_written < 16*1024) {
        /* 获取单个内存区块。 */
        if (evbuffer_peek(buf, -1, &ptr, v, 1) < 1)
            break;
        /* 通过用户定义的 consume 函数传递数据。 */
        consume(v[0].iov_base, v[0].iov_len);
        n_written += v[0].iov_len;

        /* 将指针移动到下一个内存区块。 */
        if (evbuffer_ptr_set(buf, &ptr, v[0].iov_len, EVBUFFER_PTR_ADD)<0)
            break;
    }
}
```

注意

- 修改 `evbuffer_iovec` 指向的数据可能会导致未定义的行为。
- 如果调用了任何修改缓冲区的函数，`evbuffer_peek()` 所设置的指针可能会变得无效。
- 如果缓冲区是多线程使用的，调用 `evbuffer_peek()` 之前请确保使用 `evbuffer_lock()` 锁定，并且在使用完 `evbuffer_peek()` 给出的数据之后解锁。

该函数是 Libevent 2.0.2-alpha 新引入的。

直接向缓冲区添加数据

有时可能需要直接向缓冲区插入数据，而不需要先将数据写入字符数组然后再通过 `evbuffer_add()` 拷贝。您可以使用一对高级函数来达到此目的：`evbuffer_reserve_space()` 和 `evbuffer_commit_space()`。配合 `evbuffer_peek()`，这些函数可以通过 `evbuffer_iovec` 结构体提供缓冲区内部内存的直接访问。

接口

```
int evbuffer_reserve_space(struct evbuffer *buf, ev_ssize_t size,
    struct evbuffer_iovec *vec, int n_vecs);
int evbuffer_commit_space(struct evbuffer *buf,
    struct evbuffer_iovec *vec, int n_vecs);
```

`evbuffer_reserve_space()` 函数提供指向缓冲区内部空间的指针。该函数根据需要扩展缓冲区以至少提供 `size` 字节空闲空间。这些指向空闲空间并存储空间长度的指针（`evbuffer_iovec`）会放置在您提供的数组 `vec` 中，`n_vecs` 是数组的长度。

`n_vecs` 的值至少为 1。如果提供的数组大小为 1，则 Libevent 会保证所请求的空间是连续的，即只在 1 个内存区块中。但是这么做可能会导致重新排列缓冲区内部布局或浪费内存。为了更高的性能，提供的数组大小至少为 2。该函数返回提供请求的空间所需要的内存区块数量。

写进内存空间的数据直到调用 `evbuffer_commit_space()` 才真正属于缓冲区。如果想要提交比请求的空间更少的数据，您可以减小任何 `evbuffer_iovec` 结构体的 `iov_len` 字段。您也可以（通过修改 `n_vecs`）传递更少的内存区块。`evbuffer_commit_space()` 函数成功返回 0，失败返回 -1。

提示和警告

- 调用任何重新排列缓冲区的函数或向其添加数据会导致通过 `evbuffer_reserve_space()` 获取的指针失效。
- 在当前的实现中，无论用户提供了多少 `evbuffer_iovec` 结构体，`evbuffer_reserve_space()` 都不会使用超过两个。这点可能会在将来版本中有所改变。
- 多次调用 `evbuffer_reserve_space()` 是安全的。
- 如果缓冲区需要多线程使用，请保证在调用 `evbuffer_reserve_space()` 之前使用 `evbuffer_lock()` 锁定，并在提交（commit）后立即释放。

示例

```
/* 假设我们需要使用 generate_data() 函数产生的 2048 字节输出填充缓冲区，
   并且不使用拷贝。 */
```

```
struct evbuffer_iovec v[2];
int n, i;
size_t n_to_add = 2048;

/* 申请 2048 字节。 */
n = evbuffer_reserve_space(buf, n_to_add, v, 2);
if (n<=0)
   return; /* 出于某些原因无法申请空间。 */

for (i=0; i<n && n_to_add > 0; ++i) {
   size_t len = v[i].iov_len;
   if (len > n_to_add) /* 不要写入超过 n_to_add 字节数据。 */
      len = n_to_add;
   if (generate_data(v[i].iov_base, len) < 0) {
      /* 如果产生数据时出现问题，直接停在这里即可，不会有数据提交到缓冲区。 */
      return;
   }
   /* 将 iov_len 设置成实际写入的字节数，保证不会提交太多空间。 */
   v[i].iov_len = len;
}

/* 将空间提交到缓冲区，注意我们传递 'i'（实际使用的数量）而非 'n'（可用数量）。 */
if (evbuffer_commit_space(buf, v, i) < 0)
   return; /* 提交错误 */
```

译者注：这段代码的 for 循环中应当少了 `n_to_add -= len;`。

不好的示例

```
/* 这是一些使用 evbuffer_reserve() 时可能犯的错误。
   不要模仿这段代码！ */
struct evbuffer_iovec v[2];

{
  /* 不要在调用修改缓冲区的函数之后使用通过 evbuffer_reserve_space()
     得到的指针。 */
  evbuffer_reserve_space(buf, 1024, v, 2);
  evbuffer_add(buf, "X", 1);
  /* 警告：如果 evbuffer_add 重新排列了缓冲区的内部布局，下一行不会有
     任何效果，甚至可能会使程序崩溃。取而代之，您需要在调用
     evbuffer_reserve_space 之前添加数据。 */
  memset(v[0].iov_base, 'Y', v[0].iov_len-1);
  evbuffer_commit_space(buf, v, 1);
}
```

```
{
  /* 不要修改 iov_base 指针。 */
  const char *data = "Here is some data";
  evbuffer_reserve_space(buf, strlen(data), v, 1);
  /* 警告：下面的代码并不会按照想象的那样工作。相反，您需要将数据拷贝至
     v[0].iov_base。 */
  v[0].iov_base = (char*) data;
  v[0].iov_len = strlen(data);
  /* 在这种情况下，幸运的话 evbuffer_commit_space 只会给出一个错误。 */
  evbuffer_commit_space(buf, v, 1);
}
```

这些函数及其接口自 Libevent 2.0.2-alpha 就存在。

使用缓冲区的网络 IO

Libevent 中缓冲区最常见的用法就是网络 IO。使用缓冲区的网络 IO 接口是：

接口

```
int evbuffer_write(struct evbuffer *buffer, evutil_socket_t fd);
int evbuffer_write_atmost(struct evbuffer *buffer, evutil_socket_t fd,
        ev_ssize_t howmuch);
int evbuffer_read(struct evbuffer *buffer, evutil_socket_t fd, int howmuch);
```

`evbuffer_read()` 函数从套接字 `fd` 读取最多 `howmuch` 字节数据到 `buffer` 中。该函数成功返回读取的字节数，遇到 `EOF` 返回 0，发生错误返回 -1。注意错误可能指示非阻塞操作不能立即成功，您应当检查错误码是否为 `EAGAIN`（在 Windows 上是 `WSAEWOULDBLOCK`）。如果 `howmuch` 为负数，则 `evbuffer_read()` 会尝试自己猜测大小。

`evbuffer_write_atmost()` 函数尝试向套接字 `fd` 至多写入 `buffer` 的前 `howmuch` 字节数据。该函数成功返回写入的字节数，失败返回 -1。与 `evbuffer_read()` 一样，需要检查错误码以判断是否真的发生了错误，或者仅仅是非阻塞 IO 不能立刻完成。如果提供一个负的 `howmuch`，该函数会尝试把缓冲区中所有内容写进套接字。

调用 `evbuffer_write()` 时提供负的 `howmuch` 参数与调用 `evbuffer_write_atmost()` 一样：程序会尝试尽可能多地刷新缓冲区。

在 Unix 中，这些函数对任何可读写的文件描述符均有效。在 Windows 中，则只支持套接字。

注意当您使用缓冲事件时（`bufferevent`），不需要调用这些 IO 函数：缓冲事件会自动完成这些操作。

`evbuffer_write_atmost()` 函数由 Libevent 2.0.1-alpha 引入。

缓冲区和回调

缓冲区用户可能频繁地需要知道是否有数据添加进缓冲区或从缓冲区移除，为了支持这些功能，Libevent 提供了一个通用的缓冲区回调机制。

接口

```
struct evbuffer_cb_info {
        size_t orig_size;
        size_t n_added;
        size_t n_deleted;
};

typedef void (*evbuffer_cb_func)(struct evbuffer *buffer,
    const struct evbuffer_cb_info *info, void *arg);
```

缓冲区回调会在有数据添加进缓冲区或从缓冲区移除时调用。回调函数接收缓冲区，指向 `evbuffer_cb_info` 结构体的指针以及用户提供的 `arg` 作为参数。 `evbuffer_cb_info` 结构体的 `orig_size` 字段记录了缓冲区发生变化之前存储的数据的字节数； `n_added` 字段记录了添加进缓冲区的字节数； `n_deleted` 记录了从缓冲区移除的字节数。

接口

```
struct evbuffer_cb_entry;
struct evbuffer_cb_entry *evbuffer_add_cb(struct evbuffer *buffer,
    evbuffer_cb_func cb, void *cbarg);
```

`evbuffer_add_cb()` 函数为缓冲区注册一个回调函数，并且返回指向可以用于之后代码中标识特定回调实例的不透明指针。 `cb` 参数是将被调用的函数， `cbarg` 是用户向回调函数提供的参数。

可以为一个缓冲区添加多个回调函数，添加新的回调不会移除旧的回调。

示例

```
#include <event2/buffer.h>
#include <stdio.h>
#include <stdlib.h>

/* 这是一个记录总共从缓冲区移除了多少数据的回调函数，每移除 1M
   数据都会打印一个点。 */
struct total_processed {
```

```
    size_t n;
};
void count_megabytes_cb(struct evbuffer *buffer,
    const struct evbuffer_cb_info *info, void *arg)
{
    struct total_processed *tp = arg;
    size_t old_n = tp->n;
    int megabytes, i;
    tp->n += info->n_deleted;
    megabytes = ((tp->n) >> 20) - (old_n >> 20);
    for (i=0; i<megabytes; ++i)
        putc('.', stdout);
}

void operation_with_counted_bytes(void)
{
    struct total_processed *tp = malloc(sizeof(*tp));
    struct evbuffer *buf = evbuffer_new();
    tp->n = 0;
    evbuffer_add_cb(buf, count_megabytes_cb, tp);

    /* 使用缓冲区。结束之后： */
    evbuffer_free(buf);
    free(tp);
}
```

注意释放非空缓冲区不会等同于从缓冲区移除数据（移除计数不会增加），释放缓冲区也不会释放用户提供的回调数据指针。

如果您不想某个回调对缓冲区一直生效，可以移除（永远消失）或禁用（暂时关闭）该回调：

接口

```
int evbuffer_remove_cb_entry(struct evbuffer *buffer,
    struct evbuffer_cb_entry *ent);
int evbuffer_remove_cb(struct evbuffer *buffer, evbuffer_cb_func cb,
    void *cbarg);

#define EVBUFFER_CB_ENABLED 1
int evbuffer_cb_set_flags(struct evbuffer *buffer,
                          struct evbuffer_cb_entry *cb,
                          ev_uint32_t flags);
int evbuffer_cb_clear_flags(struct evbuffer *buffer,
                          struct evbuffer_cb_entry *cb,
                          ev_uint32_t flags);
```

您既可以通过添加回调时返回的 `evbuffer_cb_entry` 实例移除回调函数，也可以通过回调函数名及相应的参数来移除。`evbuffer_remove_cb*()` 函数成功返回 0，失败返回 -1。

`evbuffer_cb_set_flags()` 和 `evbuffer_cb_clear_flags()` 函数对回调实例设置或清除相应的标志。到目前为止，只有一个受支持的用户可见标志：`EVBUFFER_CB_ENABLED`。该标志位默认被置位。如果该标志被清除，则对缓冲区的修改不会造成该回调被调用。

接口

```
int evbuffer_defer_callbacks(struct evbuffer *buffer, struct event_base *base);
```

与缓冲事件回调类似，您可以设置缓冲区回调在缓冲区发生改变时不会立即调用，而是推迟并作为给定 `event_base` 事件循环的一部分执行。这在您有多个缓冲区，它们的回调函数可能会互相添加和删除数据，并且希望避免堆栈溢出的情况下，会非常有用。

如果缓冲区的回调被推迟，那么当它们最终被调用时，它们可能会汇总多个操作的结果。

与缓冲事件相似，缓冲区内部具有引用计数，因此在回调被推迟且还没有执行时释放缓冲区是安全的。

这整个回调机制于 Libevent 2.0.1-alpha 引入。`evbuffer_cb_(set|clear)_flags()` 函数自 Libevent 2.0.2-alpha 存在。

基于缓冲区的无拷贝 IO

真正高速的网络程序通常会尽可能少地拷贝数据。Libevent 提供了一些机制来达到此目的：

接口

```
typedef void (*evbuffer_ref_cleanup_cb)(const void *data,
    size_t datalen, void *extra);

int evbuffer_add_reference(struct evbuffer *outbuf,
    const void *data, size_t datlen,
    evbuffer_ref_cleanup_cb cleanupfn, void *extra);
```

这个函数通过引用将一段数据添加到缓冲区的末尾。没有拷贝操作，取而代之的是：缓冲区仅仅存储一个指向 `datlen` 字节的数据 `data`。因此，该指针必须在缓冲区使用时保持有效。当缓冲区不再需要这部分数据时会调用用户提供的 "cleanupfn" 函数并提供 "data" 指针，"datalen" 值以及 "extra" 指针作为参数。该函数成功返回 0，失败返回 -1。

示例

```c
#include <event2/buffer.h>
#include <stdlib.h>
#include <string.h>

/* 在本例中，我们有一些缓冲区并希望使用它们将 1m 数据输出到网络。
   我们在完成这件事的同时会尽量避免在内存中保留不必要的数据副本。 */

#define HUGE_RESOURCE_SIZE (1024*1024)
struct huge_resource {
    /* 我们记录了结构体中引用的数量，以便我们知道何时可以释放。 */
    int reference_count;
    char data[HUGE_RESOURCE_SIZE];
};

struct huge_resource *new_resource(void) {
    struct huge_resource *hr = malloc(sizeof(struct huge_resource));
    hr->reference_count = 1;
    /* 这里我们需要向 hr->data 填充数据。在实际使用中，
       我们可能会加载一些数据或做一些复杂的运算。这里，
       我们直接使用 0xEE 填充。 */
    memset(hr->data, 0xEE, sizeof(hr->data));
    return hr;
}

void free_resource(struct huge_resource *hr) {
    --hr->reference_count;
    if (hr->reference_count == 0)
        free(hr);
}

static void cleanup(const void *data, size_t len, void *arg) {
    free_resource(arg);
}

/* 这是真正将数据添加进缓冲区的函数。 */
void spool_resource_to_evbuffer(struct evbuffer *buf,
    struct huge_resource *hr)
{
    ++hr->reference_count;
    evbuffer_add_reference(buf, hr->data, HUGE_RESOURCE_SIZE,
        cleanup, hr);
}
```

`evbuffer_add_reference()` 函数及其接口自 2.0.2-alpha 存在。

向缓冲区添加文件

有些操作系统提供了将文件写入网络且不需要将数据拷贝至用户空间的方法。如果存在这种机制，可以通过以下接口访问：

接口

```
int evbuffer_add_file(struct evbuffer *output, int fd, ev_off_t offset,
    size_t length);
```

`evbuffer_add_file()` 函数假定有一个可读的文件描述符（不是套接字，仅此一次！） `fd` 。该函数会将该文件从偏移 `offset` 开始的 `length` 字节数据添加到 `output` 的尾部。成功返回 0，失败返回 -1。

> 警告：在 Libevent 2.0.x 中，对通过这种方法添加的数据的可靠操作只有：通过 `evbuffer_write*()` 将数据发送到网络，通过 `evbuffer_drain()` 移除数据，或通过 `evbuffer_*_buffer()` 将数据转移到另一个缓冲区中。不能使用 `evbuffer_remove()` 将数据移除，也不能使用 `evbuffer_pullup()` 线性化数据以及其他操作。Libevent 2.1.x 会尝试移除这些限制。

如果操作系统支持 `splice()` 或 `sendfile()` ，Libevent 会在调用 `evbuffer_write()` 时用这类函数将 `fd` 中的数据直接发送到网络，而不需要将数据拷贝到用户空间。如果 `splice/sendfile` 不存在，但是有 `mmap()` 函数，Libevent 会对文件进行映射，内核可能会发现这些数据不需要复制到用户空间。否则，Libevent 会直接把数据从硬盘读进内存。

文件描述符会在缓冲区刷新或释放后关闭。如果这不是您想要的，或者您想要对文件进行更细粒度的控制，请参考下方 `file_segment` 函数。

该函数于 Libevent 2.0.1-alpha 引入。

对文件的细粒度控制

`evbuffer_add_file()` 接口在多次添加同一个文件时并不高效，因为该函数占用了文件的所有权。

接口

```
struct evbuffer_file_segment;

struct evbuffer_file_segment *evbuffer_file_segment_new(
        int fd, ev_off_t offset, ev_off_t length, unsigned flags);
void evbuffer_file_segment_free(struct evbuffer_file_segment *seg);
```

```
int evbuffer_add_file_segment(struct evbuffer *buf,
    struct evbuffer_file_segment *seg, ev_off_t offset, ev_off_t length);
```

`evbuffer_file_segment_new()` 函数创建并返回一个新的 `evbuffer_file_segment` 对象以表示存储在底层 `fd` 中的一部分，从 `offset` 开始，长度为 `length` 字节。发生错误时该函数返回 `NULL`。

文件段（file segment）可以根据需要使用 `sendfile`，`splice`，`mmap`，`CreateFileMapping` 或者 `malloc()/read()` 实现。它们（文件段）使用最轻量级的支持机制创建，并根据需要转换成更重型的机制。（例如，如果操作系统支持 `sendfile` 和 `mmap`，则文件段仅使用 `sendfile` 实现，直到您需要实际检查其内容。这时候就需要 `mmap` 参与。）可以通过以下标志来控制一个文件段的细粒度行为：

EVBUF_FS_CLOSE_ON_FREE

如果设置了该标志，通过 `evbuffer_file_segment_free` 释放文件段会关闭底层文件。

EVBUF_FS_DISABLE_MMAP

如果设置了此标志，即使合适的情况下该文件段也不会使用 mapped-memory 式的后端（`CreateFileMapping/mmap`）。

EVBUF_FS_DISABLE_SENDFILE

如果设置了此标志，即使合适的情况下该文件段也不会使用 sendfile-stype 式的后端（`sendfile/splice`）。

EVBUF_FS_DISABLE_LOCKING

如果设置了此标志，不会为文件段分配锁：任何多线程使用该文件段的方式都是不安全的。

一旦拥有了一个 `evbuffer_file_segment`，您就可以通过 `evbuffer_add_file_segment()` 将文件段的一部分或全部数据添加进缓冲区。此处的 `offset` 参数指的是文件段内部的偏移，而非文件偏移。

当您不再需要使用文件段，可以通过 `evbuffer_file_segment_free()` 释放。实际存储空间不会立即释放，直到所有缓冲区都不再持有该文件段的引用。

接口

```
typedef void (*evbuffer_file_segment_cleanup_cb)(
    struct evbuffer_file_segment const *seg, int flags, void *arg);

void evbuffer_file_segment_add_cleanup_cb(struct evbuffer_file_segment *seg,
        evbuffer_file_segment_cleanup_cb cb, void *arg);
```

您可以为文件段注册一个回调，当文件段的最后引用被释放，文件段本身即将被释放时，这个回调将会被调用。不要在这个回调函数中尝试恢复文件段，将其添加到任何缓冲区，等等。

这些文件段函数在 Libevent 2.1.1-alpha 中首次出现，`evbuffer_file_segment_add_cleanup_cb()` 是 2.1.2-alpha 新添加的。

通过引用将缓冲区添加到另一个缓冲区

可以通过引用将缓冲区添加到另一个缓冲区：并非将缓冲区的内容移动到另一个缓冲区中，而是将缓冲区的引用提供给另一个，看起来就像将数据拷贝过去了一样。

接口

```
int evbuffer_add_buffer_reference(struct evbuffer *outbuf,
    struct evbuffer *inbuf);
```

`evbuffer_add_buffer_reference()` 函数的行为与将 `outbuf` 所有内容拷贝至 `inbuf` 类似，但是不会进行任何不必要的拷贝。该函数成功返回 0，失败返回 -1。

注意，对 `inbuf` 内容的后续更改不会反映到 `outbuf` 中：该函数引用的是缓冲区的当前内容，而不是缓冲区本身。

同样需要注意不能嵌套使用缓冲区引用：`evbuffer_add_buffer_reference` 的 `outbuf` 不可作为另一次调用的 `inbuf`。

该函数于 Libevent 2.1.1-alpha 引入。

限定缓冲区只能添加或删除数据

接口

```
int evbuffer_freeze(struct evbuffer *buf, int at_front);
int evbuffer_unfreeze(struct evbuffer *buf, int at_front);
```

可以通过这些函数暂时禁止缓冲区头部或尾部的改动。缓冲事件的代码内部使用了这些函数以避免输出缓冲区头部或输入缓冲区尾部的意外改动。

`evbuffer_freeze()` 函数于 Libevent 2.0.1-alpha 引入。

废弃的缓冲区函数

缓冲区接口在 Libevent 2.0 中改变了很多。在此之前，所有缓冲区都是使用连续内存实现的，这使得访问效率非常低。

`<event.h>` 曾经暴露过缓冲区结构体的内部细节，之后都不再可见。这些细节在 1.4 到 2.0 之间发生了太多变化，任何依赖这些细节的代码都无法正常工作。

曾经有个 `EVBUFFER_LENGTH()` 宏来获取缓冲区的字节长度，实际数据可以通过 `EVBUFFER_DATA()` 获得。这些在 `<event2/buffer_compat.h>` 中仍然可用，不过需要当心：`EVBUFFER_DATA(b)` 是 `evbuffer_pullup(b, -1)` 的别名，该操作代价非常昂贵。

其他废弃的接口还有：

废弃接口

```
char *evbuffer_readline(struct evbuffer *buffer);
unsigned char *evbuffer_find(struct evbuffer *buffer,
    const unsigned char *what, size_t len);
```

`evbuffer_readline()` 函数于当前 `evbuffer_readln(buffer, NULL, EVBUFFER_EOL_ANY)` 行为一致。

`evbuffer_find()` 函数会寻找字符串在缓冲区中第一次出现的位置，并返回对应指针。与 `evbuffer_search()` 不同的是，该函数只需找第一个出现的字符串。为了兼容旧代码，现在这个函数会将找到的第一个字符串之前的数据线性化（线性化部分为缓冲区首部到第一个找到的字符串的结尾处）。

缓冲区的回调接口也不一致：

废弃接口

```
typedef void (*evbuffer_cb)(struct evbuffer *buffer,
    size_t old_len, size_t new_len, void *arg);
void evbuffer_setcb(struct evbuffer *buffer, evbuffer_cb cb, void *cbarg);
```

缓冲区同时只能有一个回调函数，因此设置新的回调函数会禁用之前的回调，设置 `NULL` 则为禁用回调。

回调时传递的参数是缓冲区的旧长度和新长度，而非 `evbuffer_cb_info` 结构体。因此，如果 `old_len` 大于 `new_len`，则意味着数据被移除了；如果 `new_len` 大于 `old_len`，则意味着有数据被添加进缓冲区。不存在延迟回调，因此永远不会在回调中添加或删除数据。

废弃的函数目前在 `<event2/buffer_compat.h>` 中仍然可用。