目录

[概要：Libevent服务器流程 2](#_Toc199784076)

[1.创建并绑定一个监听套接字 2](#_Toc199784077)

[2. event\_base的初始化与释放 4](#_Toc199784078)

[创建event\_base实例 4](#_Toc199784079)

[释放event\_base实例 4](#_Toc199784080)

[3. event事件处理 4](#_Toc199784081)

[创建event实例 4](#_Toc199784082)

[释放event实例 5](#_Toc199784083)

[将event添加到 event\_base 中进行监听 6](#_Toc199784084)

[删除event实例 6](#_Toc199784085)

[event\_free()与event\_del()的区别 6](#_Toc199784086)

[4. 事件循环 7](#_Toc199784087)

[进入事件循环 7](#_Toc199784088)

[退出事件循环 7](#_Toc199784089)

[6. 高级特性：缓冲区事件 (bufferevent) 8](#_Toc199784090)

[创建bufferevent结构体 8](#_Toc199784091)

[设置bufferevent回调函数 9](#_Toc199784092)

[启用bufferevent监听 10](#_Toc199784093)

[关闭bufferevent监听 10](#_Toc199784094)

[bufferevent读数据 11](#_Toc199784095)

[bufferevent写数据 11](#_Toc199784096)

[释放 bufferevent 实例 12](#_Toc199784097)

[客户端bufferevent TCP连接 12](#_Toc199784098)

[bufferevent进阶 12](#_Toc199784099)

[evbuffer 13](#_Toc199784100)

[evbuffer和bufferevent 17](#_Toc199784101)

## 概要：Libevent服务器流程

1. 创建event\_base
2. 创建event事件
3. 将event事件添加到event\_base
4. 进入事件循环
5. 释放event\_base

## 1.创建并绑定一个监听套接字

封装了 socket、bind 和 listen 的操作

#include <event2/listener.h>

struct evconnlistener \*evconnlistener\_new\_bind(

struct event\_base \*base,

evconnlistener\_cb cb,

void \*ptr,

unsigned flags,

int backlog,

const struct sockaddr \*sa,

int socklen);

base: event\_base 指针。

cb: 新连接到达时触发的回调函数（类型为 evconnlistener\_cb）。

ptr: 回调函数的用户自定义参数，会作为参数传入 cb。

flags: 标志位，控制 listener 的行为，常见取值如下：

LEV\_OPT\_CLOSE\_ON\_FREE: 当 listener 被释放时关闭底层 socket

LEV\_OPT\_REUSEABLE: 设置 SO\_REUSEADDR

LEV\_OPT\_THREADSAFE: 在多线程下安全使用

LEV\_OPT\_DISABLED: 初始化时不激活监听，需要手动启用

backlog: listen 时使用的队列长度，表示等待 accept 的最大连接数。通常设为 -1 表示默认值。

sa: 要绑定的地址结构体指针，如 sockaddr\_in 或 sockaddr\_in6。

socklen: 地址结构体的大小，如 sizeof(struct sockaddr\_in)。

**回调函数原型**

typedef void (\*evconnlistener\_cb)(

struct evconnlistener \*listener,

evutil\_socket\_t fd,

struct sockaddr \*sa,

int socklen,

void \*ptr);

listener: evconnlistener\_new\_bind 返回的listener

fd: 新建立的客户端连接的 socket 文件描述符

sa: 客户端地址信息。

socklen: 客户端地址结构体长度。

ptr: 用户传入的指针，通常是 server 上下文或状态信息。

**返回值**

成功返回一个 evconnlistener\* 指针。

失败返回 NULL。

## 2. event\_base的初始化与释放

### 创建event\_base实例

#include <event2/event.h>

struct event\_base \*event\_base\_new();

返回值

成功时，返回一个指向新分配并初始化的 struct event\_base 的指针。

如果发生错误（例如内存分配失败），则返回 NULL。

### 释放event\_base实例

#include <event2/event.h>

void event\_base\_free(struct event\_base \*base);

参数

base: 指向要被释放的事件基础结构的指针。

## 3. event事件处理

### 创建event实例

#include <event2/event.h>

struct event \*event\_new(

struct event\_base \*base,

evutil\_socket\_t fd,

short events,

void (\*callback)(evutil\_socket\_t, short, void \*),

void \*arg);

注：event和event\_base的绑定在该函数中已经完成了

参数

base: 想要关联此事件的 event\_base 结构体指针

fd: 要监听的文件描述符

events: 一个位掩码，表示感兴趣的事件类型。常用标志如下：

EV\_READ（文件描述符准备好读）

EV\_WRITE（文件描述符准备好写）

EV\_PERSIST（持久监听事件）

callback: 当事件触发时将被调用的回调函数

arg: 作为参数传递给回调函数的用户数据指针

返回值

成功时，返回指向新分配并初始化的 struct event 的指针。

如果发生错误（如内存分配失败），则返回 NULL。

### 释放event实例

#include <event2/event.h>

void event\_free(struct event \*event);

参数

event: 想要释放的event的指针

### 将event添加到 event\_base 中进行监听

#include <event2/event.h>

int event\_add(struct event \*ev, const struct timeval \*timeout);

注：event和event\_base的绑定在event\_new()中完成，此处无需event\_base参数

参数

ev: 想要添加的event的指针

timeout: 可选的时间值，指定事件的最大等待时间。如果为 NULL则表示没有超时限制

返回值

成功时返回 0。

如果发生错误，则返回 -1 并设置全局变量 errno 以指示错误的原因。

### 删除event实例

#include <event2/event.h>

int event\_del(struct event \*ev);

参数

ev: 想要删除的event的指针

返回值

成功时返回 0。

如果发生错误，则返回 -1 并设置全局变量 errno 来指示错误的原因。

### event\_free()与event\_del()的区别

event\_del() 只是简单地从事件循环中移除事件，使其不再响应相关事件的发生

event\_free() 则进一步释放了该事件实例所占用的所有资源，包括内存。

event\_del()和event\_add()相对应；event\_free()和event\_new()相对应

如果只是想暂停对某个事件的关注而不打算立刻释放它的资源，应该使用 event\_del()。

如果确定不再需要某个事件并希望释放其占用的所有资源，那么应该使用 event\_free()。

## 4. 事件循环

### 进入事件循环

#include <event2/event.h>

int event\_base\_dispatch(struct event\_base \*base);

参数

base: 指向你想要运行事件循环的 struct event\_base 实例的指针。

返回值

成功时返回 0。

如果发生错误，则返回 -1 并设置全局变量 errno 来指示错误的原因。

如果因为调用了 event\_base\_loopexit() 而退出，则返回 1。

### 退出事件循环

#include <event2/event.h>

int event\_base\_loopexit(

struct event\_base \*event\_base,

const struct timeval \*tv);

event\_base: 希望事件循环退出的 event\_base 结构体

tv: 指定事件循环应该在多久之后退出。如果为 NULL，则事件循环会在处理完当前事件后立即尝试退出。如果不为NULL，则会在指定时间后尝试退出

返回值

成功时返回 0。

如果发生错误，则返回 -1。

## 高级特性：缓冲区事件 (bufferevent)

bufferevent 是 libevent 提供的一种封装好的、基于缓冲的异步 I/O 结构，用于处理 socket、文件描述符等读写事件。

它管理两个方向的数据流：

输入缓冲区（read buffer）

输出缓冲区（write buffer）

这两个缓冲区是 evbuffer 类型

### 创建bufferevent结构体

#include <event2/bufferevent.h>

struct bufferevent\* bufferevent\_socket\_new(

struct event\_base\* base,

evutil\_socket\_t fd,

int options);

base: 要关联的 event\_base 的指针

fd: 套接字描述符

options: 位掩码，指定创建 bufferevent 时的选项。常用的选项包括：

BEV\_OPT\_CLOSE\_ON\_FREE: 当 bufferevent 被释放时，自动关闭相关的套接字。

BEV\_OPT\_THREADSAFE: 使 bufferevent 在多线程环境中安全使用。

BEV\_OPT\_DEFER\_CALLBACKS: 推迟回调执行直到事件循环的末尾。

BEV\_OPT\_UNLOCK\_CALLBACKS: 如果设置了 BEV\_OPT\_THREADSAFE，此选项确保在回调开始之前解锁锁。

### 设置bufferevent回调函数

void bufferevent\_setcb(

struct bufferevent \*bufev,

bufferevent\_data\_cb readcb,

bufferevent\_data\_cb writecb,

bufferevent\_event\_cb eventcb,

void \*cbarg);

bufev: 要设置回调函数的 bufferevent 实例

readcb: 当有数据可从输入缓冲区读取时调用的回调函数。不需要处理读事件则传入 NULL。

void read\_callback(struct bufferevent \*bev, void \*ctx);

writecb: 当成功将数据写入到输出缓冲区时调用的回调函数。不需要处理写事件则传入 NULL。

void write\_callback(struct bufferevent \*bev, void \*ctx);

eventcb: 当发生特定事件（如错误、连接关闭等）时调用的回调函数。这些事件包括但不限于超时、错误、EOF 等。

void event\_callback(struct bufferevent \*bev, short events, void \*ptr);

cbarg: 传递给每个回调函数的一个参数。可以是任何类型的数据，通常用来传递上下文信息。

### 启用bufferevent监听

int bufferevent\_enable(

struct bufferevent \*bufev,

short event);

bufev: 要启用事件监听的 bufferevent 实例。

event: 位掩码，指定要启用的事件类型。常见的事件类型包括：

EV\_READ: 启用读事件监听。

EV\_WRITE: 启用写事件监听。

EV\_READ | EV\_WRITE: 同时启用读和写事件监听。

### 关闭bufferevent监听

int bufferevent\_disable(

struct bufferevent \*bufev,

short event);

bufev: 指向你希望禁用事件监听的 bufferevent 实例。

event: 一个位掩码，指定你希望禁用的事件类型。常用的事件包括：

EV\_READ: 禁用读事件监听。

EV\_WRITE: 禁用写事件监听。

EV\_READ | EV\_WRITE: 同时禁用读和写事件监听。

### bufferevent读数据

size\_t bufferevent\_read(

struct bufferevent \*bufev,

void \*data,

size\_t size);

bufev: 指向你希望从中读取数据的 bufferevent 实例。

data: 一个指向足够大的内存区域的指针，用于存放从缓冲区读取的数据。

size: 要读取的最大字节数。

返回值

返回实际读取的字节数。如果没有任何数据可读，则返回值可能小于请求的 size。

### bufferevent写数据

int bufferevent\_write(

struct bufferevent \*bufev,

const void \*data,

size\_t size);

bufev: 指向你希望写入数据的 bufferevent 实例。

data: 要写入的数据指针。

size: 数据的字节大小。

返回值

成功时返回 0。

如果发生错误，则返回 -1。

### 释放 bufferevent 实例

void bufferevent\_free(struct bufferevent \*bufev);

### 客户端bufferevent TCP连接

int bufferevent\_socket\_connect(

struct bufferevent \*bev,

struct sockaddr \*address,

int socklen);

bev: bufferevent 实例

address: 指向描述目标服务器地址的 sockaddr 结构

socklen: address 结构的大小。

返回值

成功时返回 0。

如果发生错误，则返回 -1 并设置相应的错误码。

### bufferevent进阶

#### 设置读写上下限

void bufferevent\_setwatermark(struct bufferevent \*bufev,

short events,

size\_t lowmark,

size\_t highmark);

参数说明

bufev 要设置水位的 bufferevent 对象。

events 指定设置的是 EV\_READ、EV\_WRITE，或两者组合。

lowmark 低水位（低于此值则不会调用回调）。

highmark 高水位（高于此值将会暂停读写，防止缓存溢出）。

工作原理（以 EV\_READ 为例）

高水位 (highmark)：

如果输入缓冲区中的数据达到 highmark 字节，

libevent 会暂停底层 socket 的读取，防止缓冲区爆满。

低水位 (lowmark)：

只有当输入缓冲区的可读数据达到或超过 lowmark 字节时，才触发 read\_cb 回调。

如果数据还不足，它会暂时不调用回调，等更多数据进来。

写缓冲区（EV\_WRITE）逻辑类似，控制的是向 socket 写数据的节奏。

## evbuffer

evbuffer 是一个动态的、链表式的字节缓冲区，用于存放数据，可以高效地支持：

* 动态增长和缩减
* 零拷贝数据拼接
* 分段写入
* 文件或 socket I/O

evbuffer 是 bufferevent 的内部成员，但也可以独立使用

由于存放数据方式是链式，所以**数据并不连续**

**从bufferevent中获取**

struct evbuffer \*input = bufferevent\_get\_input(bev); // 获取读缓冲

struct evbuffer \*output = bufferevent\_get\_output(bev); // 获取写缓冲

**创建与释放**

struct evbuffer \*evbuffer\_new(void); 创建一个新的 evbuffer

void evbuffer\_free(struct evbuffer \*buf); 释放 evbuffer

void evbuffer\_enable\_locking(struct evbuffer \*buf, void \*lock); 启用线程安全锁支持（多线程使用）

**写入数据**（向缓冲区追加）

int evbuffer\_add(struct evbuffer \*buf, const void \*data, size\_t len); 添加原始数据

int evbuffer\_add\_buffer(struct evbuffer \*dst, struct evbuffer \*src); 把一个缓冲区的数据追加到另一个缓冲区

int evbuffer\_add\_printf(struct evbuffer \*buf, const char \*fmt, ...); 类似 printf，格式化添加数据

int evbuffer\_add\_vprintf(struct evbuffer \*buf, const char \*fmt, va\_list ap); va\_list 版本

int evbuffer\_add\_reference(struct evbuffer \*buf, const void \*data, size\_t len, evbuffer\_ref\_cleanup\_cb cb, void \*cbarg); 引用外部内存，零拷贝（高性能场景）

**读取数据**（从缓冲区取出）

int evbuffer\_remove(struct evbuffer \*buf, void \*data\_out, size\_t datalen); 复制并移除前面的数据

int evbuffer\_drain(struct evbuffer \*buf, size\_t len); 仅移除前面 len 个字节

unsigned char \*evbuffer\_pullup(struct evbuffer \*buf, ssize\_t len); 把一段可能不连续的数据整理成一块连续内存，返回可读指针（只读）

int evbuffer\_remove\_buffer(struct evbuffer \*src, struct evbuffer \*dst, size\_t len); 把数据从 src 移到 dst

**读取/查找**辅助函数

size\_t evbuffer\_get\_length(struct evbuffer \*buf); 获取缓冲区当前长度

char \*evbuffer\_readln(struct evbuffer \*buf, size\_t \*n\_read\_out, enum evbuffer\_eol\_style eol\_style); 读取一行文本，按行结束符分割

int evbuffer\_search(struct evbuffer \*buf, const char \*data, size\_t len, struct evbuffer\_ptr \*start); 查找子串

int evbuffer\_search\_eol(struct evbuffer \*buf, struct evbuffer\_ptr \*start, size\_t \*eol\_len\_out, enum evbuffer\_eol\_style eol\_style); 查找行尾位置

**数据传输**相关

int evbuffer\_write(struct evbuffer \*buf, evutil\_socket\_t fd); 把数据写入 socket

int evbuffer\_read(struct evbuffer \*buf, evutil\_socket\_t fd, int howmuch); 从 socket 读取数据到 buffer

int evbuffer\_write\_atmost(struct evbuffer \*buf, evutil\_socket\_t fd, ssize\_t howmuch); 写最多 howmuch 字节

int evbuffer\_readln\_buffer(struct evbuffer \*buf, struct evbuffer \*dst, enum evbuffer\_eol\_style eol\_style);

**高效发送文件内容**

int **evbuffer\_add\_file**(struct evbuffer \*output,

int fd,

ev\_off\_t offset,

ev\_off\_t length);

**参数**

output struct evbuffer \* 要写入的输出缓冲区，通常用bufferevent\_get\_output(bev) 获取

fd int 已打开的文件描述符（以只读方式）

offset ev\_off\_t 从文件的哪个位置开始读（一般是 0）

length ev\_off\_t 要读取的字节数，传 st.st\_size（可用 fstat 获取文件大小）

**返回值**

0 添加成功

-1 失败（比如 fd 不合法或系统调用失败）

**注意事项：**

evbuffer\_add\_file() 会自动关闭 fd，无需手动 close(fd)。

这个函数只是将文件加入缓冲区队列，真正的发送是由 libevent 的事件循环完成的（异步！）。

**底层机制（Linux）**

内部调用 sendfile()：直接将数据从内核页缓存传输到 socket，不经过用户态内存。

零拷贝，节省 CPU 和内存带宽。

避免了用户态内存分配、复制、释放，尤其适合大文件。

## evbuffer和bufferevent

**推荐的使用模式**

在回调函数中：

|  |  |
| --- | --- |
| 用途 | 推荐使用 |
| 控制 socket 读写行为 | bufferevent |
| 拼装和发送数据 | evbuffer（通过 bufferevent\_get\_output()） |
| 分包、解析协议 | evbuffer（通过 bufferevent\_get\_input()） |
| 发送数据（少量/立即） | bufferevent\_write() |
| 发送大数据或文件映射 | evbuffer\_add\_file() 等 |
| 限制读写速度/容量 | bufferevent\_setwatermark()、超时等 |