bufferevent可以理解为event的一个特例，bufferevent针对socket使用，bufferevent具有一个缓存区用于读写数据

**回调和水位**

bufferevent 有四个水位，其控制的功能如下：

读取低水位：

默认清空，当缓冲区有数据是，就会触发可读，当该水位被设置是，缓存区的数据需大于该水位，才会触发可读

读取高水位：

当缓存区数据大于该数值时，将不再读取数据，直到数据被取走，默认值是无限

写入低水位：

缓冲区的数据小于或等于该值时，才会触发可写，默认值是 0，所以只有输出缓冲区空的时候才会调用写入回调

写入高水位：bufferevent 没有直接使用这个水位，其具有特殊意义

**bufferevent 选项标志**

创建 bufferevent 时可以使用一个或者多个标志修改其行为

BEV\_OPT\_CLOSE\_ON\_FREE ：释放 bufferevent 时关闭底层传输端口

BEV\_OPT\_THREADSAFE ：

自动为 bufferevent 分配锁，这样就可以安全地在多个线程中使用 bufferevent

BEV\_OPT\_DEFER\_CALLBACKS ：设置这个标志时, bufferevent 延迟所有回调

BEV\_OPT\_UNLOCK\_CALLBACKS :

默认情况下,如果设置 bufferevent 为线程安全 的,则 bufferevent 会在调用用户提供的回调时进行锁定。设置这个选项会让 libevent 在执行回调的时候不进行锁定

**创建基于套接字的bufferevent**

struct bufferevent \*bufferevent\_socket\_new(

struct event\_base \*base,

evutil\_socket\_t fd,

enum bufferevent\_options options);

fd：套接字

options：选项标志

返回值：成功时 bufferevent，失败则返回 NULL

**释放bufferevent**

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

**设置bufferevent的回调函数**

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);

readcd：可读回调，当数据可读时调用，要禁用该回调，传递 NULL

writecd：可写回调，要禁用该回调，传递 NULL

eventcd：发生错误时回调，要禁用该回调，传递 NULL

cbarg：作为回调函数的ctx参数传入

获取bufferevent的信息使用如下函数

void bufferevent\_getcb(struct bufferevent \*bufev,

bufferevent\_data\_cb \*readcb\_ptr,

bufferevent\_data\_cb \*writecb\_ptr,

bufferevent\_event\_cb \*eventcb\_ptr,

void \*\*cbarg\_ptr);

**调整水位**

void bufferevent\_setwatermark(struct bufferevent \*bufev, short events,

size\_t lowmark, size\_t highmark);

events：EV\_READ | EV\_WRITE，标识“读取”或“写入”水位

lowmark：低水位

highmark：高水位

**启用禁用事件**

void bufferevent\_enable(struct bufferevent \*bufev, short events);

void bufferevent\_disable(struct bufferevent \*bufev, short events);

bufferevent默认启用了EV\_WRITE事件

events事件列表：与event使用的标志一样（如：EV\_READ | EV\_PERSIST等）

**在bufferevent上启动链接**

使用bufferevent\_socket\_connect连接远程机器

int bufferevent\_socket\_connect(struct bufferevent \*bev,

struct sockaddr \*address, int addrlen);

连接成功启函数返回 0，如果发生错误则返回 -1

**获取bufferevent的数据缓存区**

通过如下方法获取数据缓存区，通过数据缓存区可以进行很多操作，对数据缓存区的读写就是对服务器的读写，下一章将介绍evbuffer

struct evbuffer \*bufferevent\_get\_input(struct bufferevent \*bufev);

struct evbuffer \*bufferevent\_get\_output(struct bufferevent \*bufev);

**读取bufferevent数据**

bufferevent\_read：函数直接读取bufferevent

bufferevent\_read\_buffer：将数据读入指定的evbuffer中

size\_t bufferevent\_read(struct bufferevent \*bufev, void \*data, size\_t size);

int bufferevent\_read\_buffer(struct bufferevent \*bufev, struct evbuffer \*buf);

**向bufferevent写入数据**

bufferevent\_write：

直接将数据发送给服务器，然后回调“可写函数”

bufferevent\_write\_buffer：

将指定的evbuffer的数据发送给服务器，然后回调“可写函数”

int bufferevent\_write(struct bufferevent \*bufev, const void \*data, size\_t size);

int bufferevent\_write\_buffer(struct bufferevent \*bufev, struct evbuffer \*buf);

**示例**

#include <event2/event.h>

#include <iostream>

#include <event2/bufferevent.h>

#include <sys/socket.h>

#include <memory>

#include <sys/types.h>

#include <unistd.h>

#include <signal.h>

#include <string.h>

// 如果缓存区有数据，将被调用

void readcb(struct bufferevent \*bev, void \*ptr)

{

    char text[100] = {0};

    bufferevent\_read(bev, text, sizeof(text));

    std::cout << text << std::endl;

}

// 如果向缓存区写入数据，且可以写入，则被调用

void wirtecb(struct bufferevent \*bev, void \*ptr)

{

    std::cout << "wirtecb 被调用" << std::endl;

}

// 发生事件时被调用

void eventcb(struct bufferevent \*bev, short events, void \*ptr)

{

    if (events & BEV\_EVENT\_CONNECTED)

    {

        // 连接完成后进入

        std::string hello = "Hello World";

        // 向缓存区写入数据 "Hello World"

        // 并将缓存区的数据发送到服务器

        bufferevent\_write(bev, hello.c\_str(), hello.size());

    }

    else if (events & BEV\_EVENT\_ERROR)

    {

        // 发生了其他错误

    }

}

int main(void)

{

signal(SIGCHLD, SIG\_IGN);

    // 设置Socket地址

    struct sockaddr\_in sin;

    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 \*/

    if (fork() > 0)

    {

        // 父进程

        auto socketFd = socket(AF\_INET, SOCK\_STREAM, 0);

        // 绑定

        if (bind(socketFd, (sockaddr \*)(&sin), sizeof(sin)) < 0)

        {

            std::cout << "bind 失败" << std::endl;

            return 0;

        }

        // 设置监听套接字

        if (listen(socketFd, 5) != 0)

        {

            std::cout << "listen 失败" << std::endl;

            return 0;

        }

        while (true)

        {

            sockaddr\_in cliaddr;

            socklen\_t cliaddr\_len = sizeof(cliaddr);

            // 接收连接

            auto confd = accept(socketFd, (sockaddr \*)&cliaddr, &cliaddr\_len);

            if (confd < 0)

            {

                std::cout << "accept 失败" << std::endl;

                continue;

            }

            else

            {

                std::cout << "accept 成功" << std::endl;

            }

            // 接收数据

            char recvText[100] = {0};

            recv(confd, recvText, sizeof(recvText), 0);

            std::cout << recvText << std::endl;

            // 发送数据

            std::string hello = "Hello Text Too";

            send(confd, hello.c\_str(), hello.size(), 0);

            close(socketFd);

            close(confd);

            return 0;

        }

    }

    else

    {

        // 子进程

        sleep(2);

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

        struct bufferevent \*bev = bufferevent\_socket\_new(base, -1, BEV\_OPT\_CLOSE\_ON\_FREE);

        // 设置回调函数

        bufferevent\_setcb(bev, readcb, wirtecb, eventcb, NULL);

        // 默认设置了 EV\_WRITE，设置 EV\_READ（读取） | EV\_PERSIST（持久）

        bufferevent\_enable(bev, EV\_READ | EV\_PERSIST);

        // 连接服务器，连接成功 eventcb 将被调用

        if (bufferevent\_socket\_connect(bev,

                                       (struct sockaddr \*)&sin, sizeof(sin)) < 0)

        {

            std::cout << "bufferevent\_socket\_connect 失败" << std::endl;

            bufferevent\_free(bev);

            return -1;

        }

        // 启动 event\_base

        event\_base\_dispatch(base);

        std::cout << "子进程退出" << std::endl;

}

    return 0;

}