evconnlistener是libevent提供的监听套接字，头文件event2/listener.h

**创建和释放evconnlistener**

struct evconnlistener \*evconnlistener\_new(struct event\_base \*base,

evconnlistener\_cb cb, void \*ptr, unsigned flags, int backlog,

evutil\_socket\_t fd);

struct evconnlistener \*evconnlistener\_new\_bind(struct event\_base \*base,

evconnlistener\_cb cb, void \*ptr, unsigned flags, int backlog,

const struct sockaddr \*sa, int socklen);

void evconnlistener\_free(struct evconnlistener \*lev);

参数：

base：event\_base

cb：有新连接时的回调函数

ptr：传给回调函数的参数

flags：用于控制回调函数的行为

LEV\_OPT\_LEAVE\_SOCKETS\_BLOCKING

默认情况下，连接监听器接收新套接字后，会将其设置为非阻塞的，此标志改为阻塞

LEV\_OPT\_CLOSE\_ON\_FREE

如果设置了这个选项，释放连接监听器会关闭底层套接字

LEV\_OPT\_CLOSE\_ON\_EXEC

如果设置了这个选项,连接监听器会为底层套接字设置 close-on-exec 标志

LEV\_OPT\_REUSEABLE

某些平台在默认情况下，关闭某监听套接字后，要过一会儿其他套接字才可以绑定到同一个 端口。设置这个标志会让，可以立即打开

LEV\_OPT\_THREADSAFE

为监听器分配锁,这样就可以在多个线程中安全地使用了

backlog：连接队列长度，与listen一样

fd：作为evconnlistener\_new的参数，是以进行了bind的套接字

sa，socklen：作为evconnlistener\_new\_bind的参数，不需要自己进行bind

**监听器回调函数**

typedef void (\*evconnlistener\_cb)(struct evconnlistener \*listener,

evutil\_socket\_t sock, struct sockaddr \*addr, int len, void \*ptr);

listener：是接收连接的连接监听器 。

sock：是 新接收的套接字。

add，len ：是接收连接的地址和地址长度

ptr：调用 evconnlistener\_new() 时用户提供的指针。

**启用和禁用 evconnlistener**

int evconnlistener\_disable(struct evconnlistener \*lev);

int evconnlistener\_enable(struct evconnlistener \*lev);

**获取监听器的信息**

evutil\_socket\_t evconnlistener\_get\_fd(struct evconnlistener \*lev);

struct event\_base \*evconnlistener\_get\_base(struct evconnlistener \*lev);

evconnlistener\_get\_fd：获取监听器的描述符

evconnlistener\_get\_base：获取event\_base

**设置accept()错误时回调函数**

typedef void (\*evconnlistener\_errorcb)(struct evconnlistener \*lis, void \*ptr);

void evconnlistener\_set\_error\_cb(struct evconnlistener \*lev, evconnlistener\_errorcb errorcb);

ptr：调用 evconnlistener\_new() 时传入的 ptr

**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>

#include <event2/buffer.h>

#include <event2/listener.h>

// 如果缓存区有数据，将被调用

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

{

char text[100] = {0};

// 获取 bufferevent 的 inputevbuffer

auto inputEvBuffer = bufferevent\_get\_input(bev);

// 获取缓存区的数据长度

std::cout << "缓存区数据长度：" << evbuffer\_get\_length(inputEvBuffer) << std::endl;

// 读取缓存区数据

if (evbuffer\_remove(inputEvBuffer, text, sizeof(text)) < 0)

{

std::cout << "读取数据错误" << std::endl;

return;

}

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";

// 获取 outputbuffer

auto outputEvBuffer = bufferevent\_get\_output(bev);

// 向缓存区添加数据

if (evbuffer\_add(outputEvBuffer, hello.c\_str(), hello.size()) == -1)

{

std::cout << "向 evbuffer 写入数据失败" << std::endl;

return;

}

}

else if (events & BEV\_EVENT\_ERROR)

{

// 发生了其他错误

}

}

// 服务器监听回调函数

void evconnlistenercb(struct evconnlistener \*listener,

evutil\_socket\_t sock, struct sockaddr \*addr, int len, void \*ptr)

{

// 接收数据

char recvText[100] = {0};

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

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

// 发送数据

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

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

close(sock);

// 释放 evconnlistener，这将会使 evconnlistener 从 event\_base 移除

// event\_base 的事件为 0 ，则会退出

evconnlistener\_free(listener);

}

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;

}

auto evBase = event\_base\_new();

auto evlistener = evconnlistener\_new(evBase, evconnlistenercb, nullptr, LEV\_OPT\_CLOSE\_ON\_FREE, 5, socketFd);

if (!evlistener)

{

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

return 0;

}

evconnlistener\_enable(evlistener);

// 启动 event\_base

event\_base\_dispatch(evBase);

std::cout << "父进程退出" << std::endl;

}

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;

}