# 网络编程第15天-libevent

## 学习目标

* 描述什么是libevent并掌握如何安装
* 掌握event\_base的作用和使用方法
* 熟练掌握libevent库中的事件循环
* 掌握event事件的使用方法
* 掌握bufferevent的工作方式
* 掌握使用libevent实现tcp服务器端流程
* 掌握使用Libevent实现tcp客户端流程

## libevent介绍

1 事件驱动, 高性能, 轻量级, 专注于网络

2 源代码精炼, 易读

3 跨平台

4 支持多种I/O多路复用技术, 如epoll select poll等

5 支持I/O和信号等事件

## **libevent的安装**

登录官方网站: [http://libevent.org](http://libevent.org/), 查看相关信息

libevent源码下载主要分2个大版本：

1. 1.4.x 系列, 较为早期版本, 适合源码学习
2. 2.x系列, 较新的版本, 代码量比1.4版本多很多, 功能也更完善。

libevent的核心实现:

在linux上, 其实质就是epoll反应堆.

libevent是事件驱动, epoll反应堆也是事件驱动, 当要监测的事件发生的时候, 就会调用事件对应的回调函数, 执行相应操作. 特别提醒: 事件回调函数是由用户开发的, 但是不是由用户显示去调用的, 而是由libevent去调用的.

从官网[http://libevent.org](http://libevent.org/)上下载安装文件之后, 将安装文件上传到linux系统上;源码包的安装,以2.0.22版本为例,在官网可以下载到源码包libevent-2.0.22-stable.tar.gz, 安装步骤与第三方库源码包安装方式基本一致。

第一步: 解压libevent-2.0.22-stable.tar.gz

* 解压: tar -zxvf libevent-2.0.22-stable.tar.gz
* cd到libevent-2.0.22-stable目录下, 查看README文件, 该文件里描述了安装的详细步骤, 可参照这个文件进行安装.

第二步: 进入源码目录:

* 执行配置./configure, 检测安装环境, 生成makefile.
* 执行./configure的时候也可以指定路径, ./configure --prefix=/usr/xxxxx, 这样就可以安装到指定的目录下, 但是这样在进行源代码编译的时候需要指定用-I头文件的路径和用-L库文件的路径. 若默认安装不指定--prefix, 则会安装到系统默认的路径下, 编译的时候可以不指定头文件和库文件所在的路径.
* 执行make命令编译整个项目文件.
  + 通过执行make命令, 会生成一些库文件(动态库和静态库)和可执行文件.
* 执行sudo make install进行安装
* 安装需要root用户权限, 这一步需要输入当前用户的密码
* 执行这一步, 可以将刚刚编译成的库文件和可执行文件以及一些头文件拷贝到/usr/local目录下:

----头文件拷贝到了/usr/local/include目录下;

----库文件拷贝到了/usr/local/lib目录下.

## **libevent库的使用**

进入到libevent-2.0.22-stable/sample下, 可以查看一些示例源代码文件.

使用libevent库编写代码在编译程序的时候需要指定库名:-levent;

安装文件的libevent库文件所在路径:libevent-2.0.22-stable/.libs;

编写代码的时候用到event.h头文件, 或者直接参考sample目录下的源代码文件也可以.

#include <event2/event.h>

编译源代码文件(以hello-world.c文件为例)

gcc hello-world.c -levent

由于安装的时候已经将头文件和库文件拷贝到了系统头文件所在路径/usr/local/include和系统库文件所在路径/usr/local/lib, 所以这里编译的时候可以不用指定-I和-L.

编译示例代码hello-world.c程序:

gcc -o hello-world hello-world.c -levent

测试: 在另一个终端窗口进行测试, 输入: nc 127.1 9995, 然后回车立刻显示Hello, World!字符串.

## **libevent的使用**

### **libevent的地基-event\_base**

使用libevent 函数之前需要分配一个或者多个 event\_base 结构体, 每个event\_base结构体持有一个事件集合, 可以检测以确定哪个事件是激活的, event\_base结构相当于epoll红黑树的树根节点, 每个event\_base都有一种用于检测某种事件已经就绪的 “方法”(回调函数)

通常情况下可以通过event\_base\_new函数获得event\_base结构。

下面介绍一些常用函数:

相关函数说明:

1 struct event\_base \*event\_base\_new(void); //event.h的L:337

函数说明: 获得event\_base结构

参数说明: 无

返回值:

成功返回event\_base结构体指针;

失败返回NULL;

2 void event\_base\_free(struct event\_base \*); //event.h的L:561

函数说明: 释放event\_base指针

3 int event\_reinit(struct event\_base \*base); //event.h的L:349

函数说明: 如果有子进程, 且子进程也要使用base, 则子进程需要对event\_base重新初始化, 此时需要调用event\_reinit函数.

函数参数: 由event\_base\_new返回的执行event\_base结构的指针

返回值: 成功返回0, 失败返回-1

对于不同系统而言, event\_base就是调用不同的多路IO接口去判断事件是否已经被激活, 对于linux系统而言, 核心调用的就是epoll, 同时支持poll和select.

查看libevent支持的后端的方法有哪些:

const char \*\*event\_get\_supported\_methods(void);

函数说明: 获得当前系统(或者称为平台)支持的方法有哪些

参数: 无

返回值: 返回二维数组, 类似与main函数的第二个参数\*\*argv.

const char \* event\_base\_get\_method(const struct event\_base \*base);

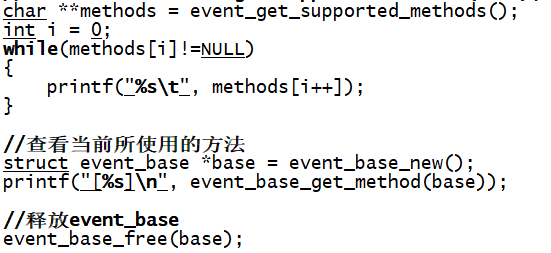
函数说明: 获得当前base节点使用的多路io方法

函数参数: event\_base结构的base指针.

返回值: 获得当前base节点使用的多路io方法的指针

编写代码获得当前系统支持的多路IO方法和当前所使用的方法:

相关的代码片段如下:



### **等待事件产生-循环等待event\_loop**

libevent在地基打好之后, 需要等待事件的产生, 也就是等待事件被激活, 所以程序不能退出, 对于epoll来说, 我们需要自己控制循环, 而在libevent中也给我们提供了API接口, 类似where(1)的功能.

函数如下：

int event\_base\_loop(struct event\_base \*base, int flags); //event.h的L:660

函数说明: 进入循环等待事件

参数说明:

* base: 由event\_base\_new函数返回的指向event\_base结构的指针
* flags的取值：
* #define EVLOOP\_ONCE 0x01

只触发一次, 如果事件没有被触发, 阻塞等待

* #define EVLOOP\_NONBLOCK 0x02

非阻塞方式检测事件是否被触发, 不管事件触发与否, 都会 立即返回.

这个函数一般不用, 而大多数都调用libevent给我们提供的另外一个API：

int event\_base\_dispatch(struct event\_base \*base); //event.h的L:364

函数说明: 进入循环等待事件

参数说明:由event\_base\_new函数返回的指向event\_base结构的指针

调用该函数, 相当于没有设置标志位的event\_base\_loop。程序将会一直运行, 直到没有需要检测的事件了, 或者被结束循环的API终止。

int event\_base\_loopexit(struct event\_base \*base, const struct timeval \*tv);

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

struct timeval {

long tv\_sec;

long tv\_usec;

};

两个函数的区别是如果正在执行激活事件的回调函数, 那么event\_base\_loopexit将在事件回调执行结束后终止循环（如果tv时间非NULL, 那么将等待tv设置的时间后立即结束循环）, 而event\_base\_loopbreak会立即终止循环。

### 使用libevent库的步骤：

1 创建根节点--event\_base\_new

2 设置监听事件和数据可读可写的事件的回调函数

设置了事件对应的回调函数以后, 当事件产生的时候会自动调用回调函数

3 事件循环--event\_base\_dispatch

相当于while(1), 在循环内部等待事件的发生, 若有事件发生则会触发事件对应的回调函数。

4 释放根节点--event\_base\_free

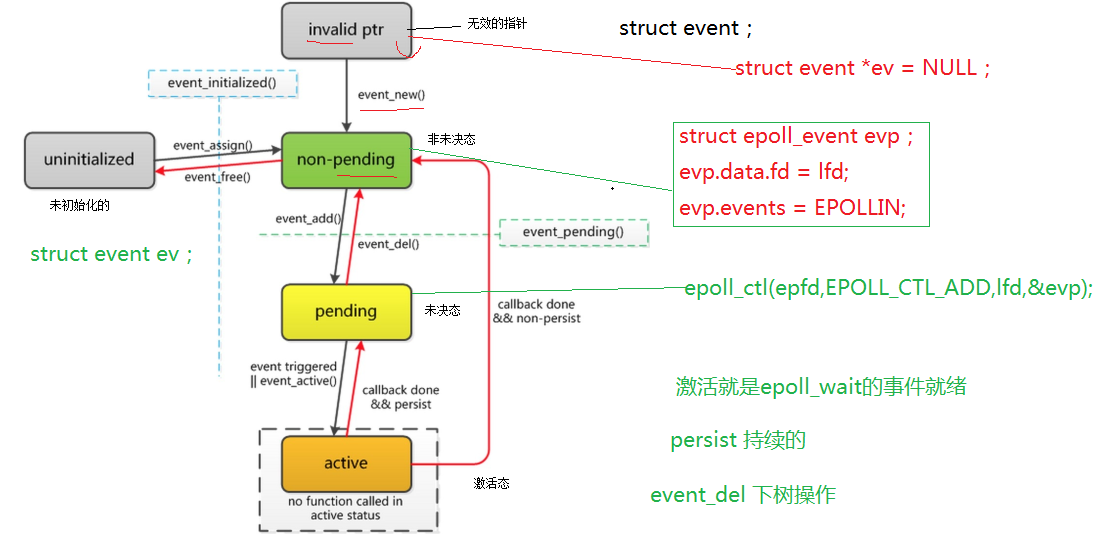
释放由event\_base\_new和event\_new创建的资源, 分别调用event\_base\_free

和event\_free函数.

### 事件驱动-event

事件驱动实际上是libevent的核心思想, 本小节主要介绍基本的事件event。

主要的状态转化：



主要几个状态：

无效的指针: 此时仅仅是定义了 struct event \*ptr；

非未决：相当于创建了事件, 但是事件还没有处于被监听状态, 类似于我们使用epoll的时候定义了struct epoll\_event ev并且对ev的两个字段进行了赋值, 但是此时尚未调用epoll\_ctl对事件上树.

未决：就是对事件开始监听, 暂时未有事件产生。相当于调用epoll\_ctl对要监听的事件上树, 但是没有事件产生.

激活：代表监听的事件已经产生, 这时需要处理, 相当于调用epoll\_wait函数有返回, 当事件被激活以后, libevent会调用该事件对应的回调函数.

libevent的事件驱动对应的结构体为struct event, 对应的函数在图上也比较清晰, 下面介绍一下主要的函数:

typedef void (\*event\_callback\_fn)(evutil\_socket\_t fd, short events, void \*arg);

struct event \*event\_new(struct event\_base \*base, evutil\_socket\_t fd, short events, event\_callback\_fn cb, void \*arg);

函数说明: event\_new负责创建event结构指针, 同时指定对应的地基base, 还有对应的文件描述符, 事件, 以及回调函数和回调函数的参数。

参数说明：

base: 对应的根节点--地基

fd: 要监听的文件描述符

events:要监听的事件

#define EV\_TIMEOUT 0x01 //超时事件

#define EV\_READ 0x02 //读事件

#define EV\_WRITE 0x04 //写事件

#define EV\_SIGNAL 0x08 //信号事件

#define EV\_PERSIST 0x10 //周期性触发

#define EV\_ET 0x20 //边缘触发, 如果底层模型支持设置 则有效, 若不支持则无效.

若要想设置持续的读事件则： EV\_READ | EV\_PERSIST

cb 回调函数, 原型如下：

typedef void (\*event\_callback\_fn)(evutil\_socket\_t fd, short events, void \*arg);

注意: 回调函数的参数就对应于event\_new函数的fd, event和arg

#define evsignal\_new(b, x, cb, arg) \

event\_new((b), (x), EV\_SIGNAL|EV\_PERSIST, (cb), (arg))

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

函数说明: 将非未决态事件转为未决态, 相当于调用epoll\_ctl函数(EPOLL\_CTL\_ADD), 开始监听事件是否产生, 相当于epoll的上树操作.

参数说明：

ev: 调用event\_new创建的事件

timeout: 限时等待事件的产生, 也可以设置为NULL, 没有限时。

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

函数说明: 将事件从未决态变为非未决态, 相当于epoll的下树（epoll\_ctl调用 EPOLL\_CTL\_DEL操作）操作。

参数说明: ev指的是由event\_new创建的事件.

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

函数说明: 释放由event\_new申请的event节点。

## 编写一个基于event实现的tcp服务器：

总体步骤：

1 搭建服务器的固定三步：

* --创建socket
* --绑定bind
* --监听listen

2 调用event\_base\_new函数创建event\_base节点

3 创建要监听的事件event, 主要就是监听事件和读数据的事件

--设置好监听事件的回调函数,然后event\_add上树---->有新的连接, 则 调用accept接受新的连接---->将这个新的连接设置好回调函数(一般 是设置读事件), 然后继续event\_add上树, 若有客户端关闭连接则 从树上摘除该事件节点.

4 调用event\_base\_dispatch进入循环等待事件的发生

5 释放资源

调用event\_base\_free释放根节点和调用event\_free释放事件节点

测试代码:

程序名称: 02\_server.c

测试过程:

编译源代码: gcc -o 02\_server 02\_server.c -levent

启动程序: ./02\_server

在其他终端窗口上使用nc命令进行测试: nc 127.1 8888, 多开几个终端窗口使用nc命令进行测试.

发现问题:

当使用多个客户端(nc命令模拟客户端程序)进行测试的时候, 特别是当关闭所有客户端程序的时候, 若再次开启nc命令, 会发现异常.

分析原因:

在02\_server.c代码中,L:11处 struct event \*readev = NULL; 是一个全局变量, 当有多个客户端请求服务的时候, 如2个客户端请求服务的时候, 第二次readev的值会将第一次readev的值覆盖掉: 代码L:55处readev = event\_new(base, cfd, EV\_READ | EV\_PERSIST, readcb, base);可以进行如下的测试重现异常情况: 先后在终端A和B上执行nc 127.1 8888命令, 然后ctrl+c结束掉终端A上的nc命令, 再次进如到终端B上, 则会出现异常情况.(原因是由于readev是一个全局变量, 所以readev只能保留最后一次所赋的值, 当客户端退出后, 服务端会调用event\_del(readev);从根节点上摘除该事件, 此时其实从base节点上摘掉的是最后一个event事件节点, 所以最后一个客户端会出现异常, 其实只要是开启了多个客户端, 而且关闭客户端的时候只要不是关闭最后一个客户端, 都会出现这种异常情况)

若先结束终端B上的nc命令, 不会出现异常情况.

解决办法: 可以将对应事件的文件描述符和事件做一个映射, 说的通俗一点就是可以将fd和event定义在一个结构体当中, 然后定义一个结构体数组, 这样可以使fd和event形成一个一对一的映射关系, 通过fd就可以找到event.

struct event\_fd

{

evutil\_socket\_t fd;

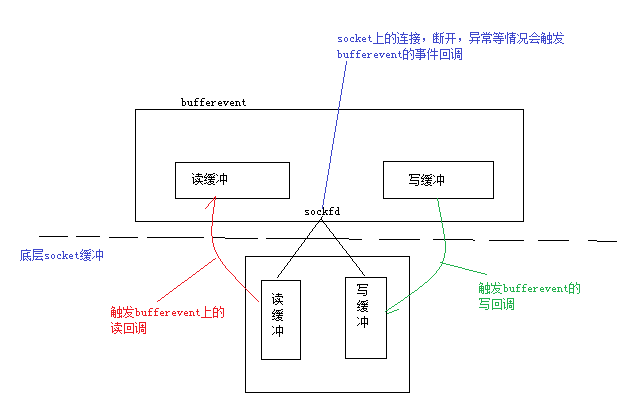
struct event \*ev;

}event[MAX];

详情见代码部分. 02\_server\_adv.c, 结合代码进行理解.

## 自带buffer的事件-bufferevent

bufferevent实际上也是一个event, 只不过比普通的event高级一些, 它的内部有两个缓冲区, 以及一个文件描述符（网络套接字）。一个网络套接字有读和写两个缓冲区, bufferevent同样也带有两个缓冲区, 还有就是libevent事件驱动的核心回调函数, 那么四个缓冲区以及触发回调的关系如下：



从图中可以得知, 一个bufferevent对应两个缓冲区, 三个回调函数, 分别是写回调, 读回调和事件回调.

bufferevent有三个回调函数：

读回调 – 当bufferevent将底层读缓冲区的数据读到自身的读缓冲区时触发读事件回调.

写回调 – 当bufferevent将自身写缓冲的数据写到底层写缓冲区的时候触发写事件回调, 由于数据最终是写入了内核的写缓冲区中, 应用程序已经无法控制, 这个事件对于应用程序来说基本没什么用, 只是通知功能.

事件回调 – 当bufferevent绑定的socket连接, 断开或者异常的时候触发事件回调.

主要使用的函数如下：

struct bufferevent \*bufferevent\_socket\_new(struct event\_base \*base, evutil\_socket\_t fd, int options);

函数说明: bufferevent\_socket\_new 对已经存在socket创建bufferevent事件, 可用于后面讲到的连接监听器的回调函数中.

参数说明：

base :对应根节点

fd :文件描述符

options : bufferevent的选项

BEV\_OPT\_CLOSE\_ON\_FREE -- 释放bufferevent自动关闭底层接口(当bufferevent被释放以后, 文件描述符也随之被close)

BEV\_OPT\_THREADSAFE -- 使bufferevent能够在多线程下是安全的

int bufferevent\_socket\_connect(struct bufferevent \*bev, struct sockaddr \*serv, int socklen);

函数说明: 该函数封装了底层的socket与connect接口, 通过调用此函数, 可以将bufferevent事件与通信的socket进行绑定, 参数如下：

bev – 需要提前初始化的bufferevent事件

serv – 对端(一般指服务端)的ip地址, 端口, 协议的结构指针

socklen – 描述serv的长度

说明: 调用此函数以后, 通信的socket与bufferevent缓冲区做了绑定, 后面调用了bufferevent\_setcb函数以后, 会对bufferevent缓冲区的读写操作的事件设置回调函数, 当往缓冲区中写数据的时候会触发写回调函数, 当数据从socket的内核缓冲区读到bufferevent读缓冲区中的时候会触发读回调函数.

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

函数说明: 释放bufferevent

void bufferevent\_setcb(struct bufferevent \*bufev,

bufferevent\_data\_cb readcb, bufferevent\_data\_cb writecb,

bufferevent\_event\_cb eventcb, void \*cbarg);

函数说明: bufferevent\_setcb用于设置bufferevent的回调函数, readcb, writecb, eventcb分别对应了读回调, 写回调, 事件回调, cbarg代表回调函数的 参数。

回调函数的原型：

typedef void (\*bufferevent\_data\_cb)(struct bufferevent \*bev, void \*ctx);

typedef void (\*bufferevent\_event\_cb)(struct bufferevent \*bev, short what, void \*ctx);

What 代表 对应的事件

BEV\_EVENT\_EOF--遇到文件结束指示

BEV\_EVENT\_ERROR--发生错误

BEV\_EVENT\_TIMEOUT--发生超时

BEV\_EVENT\_CONNECTED--请求的过程中连接已经完成

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

bufferevent\_write是将data的数据写到bufferevent的写缓冲区

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

bufferevent\_write\_buffer 是将数据写到写缓冲区另外一个写法, 实际上bufferevent的内部的两个缓冲区结构就是struct evbuffer。

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

bufferevent\_read 是将bufferevent的读缓冲区数据读到data中, 同时将读到的数据从bufferevent的读缓冲清除。

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

bufferevent\_read\_buffer 将bufferevent读缓冲数据读到buf中, 接口的另外一种。

int bufferevent\_enable(struct bufferevent \*bufev, short event);

int bufferevent\_disable(struct bufferevent \*bufev, short event);

bufferevent\_enable与bufferevent\_disable是设置事件是否生效, 如果设置为disable, 事件回调将不会被触发。

## 链接监听器-evconnlistener

链接监听器封装了底层的socket通信相关函数, 比如socket, bind, listen, accept这几个函数。链接监听器创建后实际上相当于调用了socket, bind, listen, 此时等待新的客户端连接到来, 如果有新的客户端连接, 那么内部先进行调用accept处理, 然后调用用户指定的回调函数。可以先看看函数原型, 了解一下它是怎么运作的：

函数声明所在的头文件: 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);

函数说明:

是在当前没有套接字的情况下对链接监听器进行初始化, 看最后2个参数实际上就是bind使用的关键参数, backlog是listen函数的关键参数（略有不同的是, 如果backlog是-1, 那么监听器会自动选择一个合适的值, 如果填0, 那么监听器会认为listen函数已经被调用过了）, ptr是回调函数的参数, cb是有新连接之后的回调函数, 但是注意这个回调函数触发的时候, 链接器已经处理好新连接了, 并将与新连接通信的描述符交给回调函数。flags 需要参考几个值：

LEV\_OPT\_LEAVE\_SOCKETS\_BLOCKING 文件描述符为阻塞的

LEV\_OPT\_CLOSE\_ON\_FREE 关闭时自动释放

LEV\_OPT\_REUSEABLE 端口复用

LEV\_OPT\_THREADSAFE 分配锁, 线程安全

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

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

evutil\_socket\_t fd);

evconnlistener\_new函数与前一个函数不同的地方在与后2个参数, 使用本函数时, 认为socket已经初始化好, 并且bind完成, 甚至也可以做完listen, 所以大多数时候, 我们都可以使用第一个函数。

两个函数的回调函数

typedef void (\*evconnlistener\_cb)(struct evconnlistener \*evl, evutil\_socket\_t fd, struct sockaddr \*cliaddr, int socklen, void \*ptr);

回调函数fd参数是与客户端通信的描述符, 并非是等待连接的监听的那个描述符, 所以cliaddr对应的也是新连接的对端地址信息, 已经是accept处理好的。

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

函数说明: 释放链接监听器

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

函数说明: 使链接监听器生效

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

函数说明: 使链接监听器失效

如果上述函数都较为了解了, 可以尝试去看懂hello-world.c的代码, 在安装包的sample目录下, 其中有涉及到信号的函数, 看看自己能否找到函数的原型在哪？实际上就是一个宏定义, 也是我们之前介绍的event\_new函数, 只是对应一个信号事件而已, 处理机制略有不同。

#define evsignal\_new(b, x, cb, arg) \

event\_new((b), (x), EV\_SIGNAL|EV\_PERSIST, (cb), (arg))

代码测试:

思路hello-world.c代码中当使用nc客户端测试的时候为什么nc收到hello world之后就立刻关闭了.

答案: 当服务器收到新的连接请求的时候, 会自动触发listener\_cb回调函数, 该函数中有往bufferevent缓冲区中写入的操作(调用bufferevent\_write), 接着又会触发写回调函数conn\_writecb的执行, 这个回调函数中调用了bufferevent\_free, 该函数能够释放bufferevent, 同时会关闭socket连接.

若是按下crtl+c会将程序终止, 此时会触发异常事件的退出函数(conn\_eventcb)

hello-world.c代码没有读事件触发, 可以将代码进行修改, 将bufferevent的读事件添加上.

思考: 如何修改hello-world.c添加读回调.

libevent客户端代码阅读和分析.

画hello-world.c代码的改进版的流程图.

总结的话:

对于bufferevent来说, 一个文件描述符, 2个缓冲区, 3个回调函数

文件描述符是用于和客户端进行通信的通信文件描述符, 并不是监听的文件描述符

2个缓冲区是指: 一个bufferevent包括读缓冲区和写缓冲区

3个回调函数指: 读回调函数 写回调函数 和事件回调函数(客户端关闭连接或者是被信号终止进程会触发事件回调函数)

其中写回调基本上没什么用, 事件回调指的是socket上的连接和断开,异常等情况会触发bufferevent的事件回调.

读回调函数的触发时机:

当socket的内核socket读缓冲区中有数据的时候, bufferevent会将内核缓冲区中的数据读到自身的读缓冲区, 会触发bufferevent的读操作, 此时会调用bufferevent的读回调函数.

写回调函数的触发时机:

当往bufferevent的写缓冲区写入数据的时候, bufferevent底层会把缓冲区中的数据写入到内核的socket的写缓冲区中, 此时会触发bufferevent的写回调函数, 最后由内核的驱动程序将数据发送出去.