高并发libevent

现在高性能网络服务器基本都是异步I/O模式构建的，而Libevent就是对select、poll、epoll等各类异步模式接口的封装，通过设置回调函数的方式，在监听文件描述符和套接字读写事件的同时，还兼任定时器和信号接收的管理工作。

所以Libevent对高性能服务器后台开发、跨平台开发、网络开发都具有很大的参考学习价值。官方主页显示很多的项目都用到了Libevent库，而且还可作为主机内部进程间通信和数据交互。Libevent也考虑到pthread线程模型的同步问题，保证关键数据结构在多线程并行下的数据安全！

libevent 由下列组件构成：

1. evutil：用于抽象不同平台网络实现差异的通用功能。
2. event 和event_base：libevent 的核心，为各种平台特定的、基于事件的非阻塞IO 后端提供抽象API，让程序可以知道套接字何时已经准备好，可以读或者写，并且处理基本的超时功能，检测OS 信号。
3. bufferevent：为libevent 基于事件的核心提供使用更方便的封装。除了通知程序套接字已经准备好读写之外，还让程序可以请求缓冲的读写操作，可以知道何时IO 已经真正发生。（bufferevent 接口有多个后端，可以采用系统能够提供的更快的非阻塞IO 方式，如Windows 中的IOCP。）
4. evbuffer：在bufferevent 层之下实现了缓冲功能，并且提供了方便有效的访问函数。
5. evhttp：一个简单的HTTP 客户端/服务器实现。
6. evdns：一个简单的DNS 客户端/服务器实现。
7. evrpc：一个简单的RPC 实现

ubuntu上安装libevent只需输入以下命令：

```
sudo apt-get install libevent-dev
```

也可以直接到官网上下载源码包来安装。http://libevent.org/

```
#在当前目录下解压安装包：
tar -zxvf libevent-2.0.22-stable.tzr.gz
cd libevent-2.0.22-stable
#配置安装库的目标路径：
./configure --prefix=/usr
#编译安装libevent库：
make
make install
#查看libevent库是否安装成功：
ls -al /usr/lib | grep libevent
```

通过函数event_get_version()可以查看libevent的版本。

1 建立event_base

event_base算是Libevent最基础、最重要的对象，因为修改配置、添加事件等，基本都需要将它作为参数传递进去。

1.1创建默认的event_base

这个对象通过event_base_new创建:

```
#include <event2/event.h>
struct event_base *event_base_new(void);
```

event_base_new（）函数分配并且返回一个新的具有默认设置的event_base。函数会检测环境变量，返回一个到event_base的指针。如果发生错误，则返回NULL。选择各种方法时，函数会选择OS支持的最快方法。

使用完event_base之后，使用event_base_free（）进行释放。

```
void event_base_free(struct event_base *base);
```

注意：这个函数不会释放当前与event_base关联的任何事件，或者关闭他们的套接字，或者释放任何指针。

编译的时候需要加上`-levent`

1.2 检查后端的方法

有时候需要检查event_base支持哪些特征，或者当前使用哪种方法。

```
const char **event_get_supported_methods(void);
const char **event_get_version();
```

event_get_supported_methods（）函数返回一个指针，指向libevent支持的方法名字数组。这个数组的最后一个元素是NULL。

```
#include <event2/event.h>
#include <stdio.h>
int main(void)
{
    int i = 0;
    const char **methods = event_get_supported_methods();
    printf("Starting Libevent %s.  Available methods are:\n", event_get_version());
    while(methods[i] != NULL)
    {
        printf("%s\n", methods[i++]);
    }
}
```

1.3 创建复杂的event_base

要对取得什么类型的event_base有更多的控制，就需要使用event_config。event_config是一个容纳event_base配置信息的不透明结构体。需要event_base时，将event_config传递给event_base_new_with_config（）。

```
struct event_config *event_config_new(void);
struct event_base *event_base_new_with_config(const struct event_config *cfg);
void event_config_free(struct event_config *cfg);
```

要使用这些函数分配event_base，先调用event_config_new（）分配一个event_config。然后，对event_config调用其它函数，设置所需要的event_base特征。最后，调用event_base_new_with_config（）获取新的event_base。完成工作后，使用event_config_free（）释放event_config。

1.3.1 禁用特定后端

调用event_config_avoid_method（）可以通过名字让libevent避免使用特定的可用后端。成功时返回0，失败时返回-1。

```
int event_config_avoid_method(struct event_config *cfg, const char *method);
const char* method;//指定特定的后端，例如"select" "poll" "epoll"
```

1.3.2 设置特征

调用event_config_require_feature（）让libevent不使用不能提供所有指定特征的后端。成功时返回0，失败时返回-1。

```
int event_config_require_features(struct event_config *cfg,
                                  enum event_method_feature feature);
enum event_method_feature {
    EV_FEATURE_ET = 0x01, //要求支持边沿触发的后端
    EV_FEATURE_O1 = 0x02, //要求添加、删除单个事件，或者确定哪个事件激活的操作是O（1）复杂度的后端
    EV_FEATURE_FDS = 0x04, //要求支持任意文件描述符，而不仅仅是套接字的后端
};
```

注意:

设置event_config，请求OS不能提供的后端是很容易的。

比如说，对于libevent 2.0.1-alpha，在Windows中是没有O(1)后端的；

在Linux中也没有同时提供EV_FEATURE_FDS和EV_FEATURE_O1特征的后端。

如果创建了libevent不能满足的配置，event_base_new_with_config（）会返回NULL。

1.3.3 设置标志

调用event_config_set_flag（）让libevent在创建event_base时设置一个或者多个将在下面介绍的运行时标志。成功时返回0，失败时返回-1。

```
int event_config_set_flag(struct event_config *cfg,
    enum event_base_config_flag flag);

enum event_base_config_flag {
    EVENT_BASE_FLAG_NOLOCK = 0x01,
    EVENT_BASE_FLAG_IGNORE_ENV = 0x02,
    EVENT_BASE_FLAG_STARTUP_IOCP = 0x04,
    EVENT_BASE_FLAG_NO_CACHE_TIME = 0x08,
    EVENT_BASE_FLAG_EPOLL_USE_CHANGELIST = 0x10,
    EVENT_BASE_FLAG_PRECISE_TIMER = 0x20
};
```

```
EVENT_BASE_FLAG_NOLOCK: //不要为event_base分配锁。设置这个选项可以为event_base节省一点用于锁定和解锁的时间，但是让在多个线程中访问event_base成为不安全的。

EVENT_BASE_FLAG_IGNORE_ENV: //选择使用的后端时，不要检测EVENT_*环境变量。使用这个标志需要三思：这会让用户更难调试你的程序与libevent的交互。

EVENT_BASE_FLAG_STARTUP_IOCP: //仅用于Windows，让libevent在启动时就启用任何必需的IOCP分发逻辑，而不是按需启用。

EVENT_BASE_FLAG_NO_CACHE_TIME: //不是在事件循环每次准备执行超时回调时检测当前时间，而是在每次超时回调后进行检测。注意：这会消耗更多的CPU时间。

EVENT_BASE_FLAG_EPOLL_USE_CHANGELIST: //告诉libevent，如果决定使用epoll后端，可以安全地使用更快的基于changelist的后端。epoll-changelist后端可以在后端的分发函数调用之间，同样的fd多次修改其状态的情况下，避免不必要的系统调用。但是如果传递任何使用dup()或者其变体克隆的fd给libevent，epoll-changelist后端会触发一个内核bug，导致不正确的结果。在不使用epoll后端的情况下，这个标志是没有效果的。也可以通过设置EVENT_EPOLL_USE_CHANGELIST环境变量来打开epoll-changelist选项。
```

1.3.4 event_base优先级

libevent支持为事件设置多个优先级。然而，event_base默认只支持单个优先级。可以调用event_base_priority_init()设置event_base的优先级数目。

```
int event_base_priority_init(struct event_base *base, int n_priorities);
```

成功时这个函数返回0，失败时返回-1。base是要修改的event_base，n_priorities是要支持的优先级数目，这个数目至少是1。每个新的事件可用的优先级将从0（最高）到n_priorities-1（最低）。

常量EVENT_MAX_PRIORITIES表示n_priorities的上限。调用这个函数时为n_priorities给出更大的值是错误的。

注意：

必须在任何事件激活之前调用这个函数，最好在创建event_base后立刻调用。

1.3.5 定制event_base代码

```
struct event_config *cfg;
struct event_base *base;

cfg = event_config_new();
event_config_avoid_method(cfg, "select");   //避免使用低效率select
event_config_require_features(cfg, EV_FEATURE_ET);  //使用具有边沿触发类型的后端
event_config_set_flag(cfg, EVENT_BASE_FLAG_EPOLL_USE_CHANGELIST);
base = event_base_new_with_config(cfg);
event_config_free(cfg);

event_base_priority_init(base, 3); //设置优先级
event_base_free(base);
```

1.3.6 检查后端和特征

event_base_get_method（）返回event_base正在使用的方法。event_base_get_features（）返回event_base支持的特征的比特掩码。

```
const char *event_base_get_method(const struct event_base *base);
enum event_method_feature event_base_get_features(const struct event_base *base);
```

示例代码如下：

```
#include <stdio.h>
#include <event2/event.h>
int main(void)
{
    struct event_base *base;
    enum event_method_feature f;
    base = event_base_new();
    if (!base)
    {
        puts("Couldn't get an event_base!");
    }
    else
    {
        printf("Using Libevent with backend method %s.\n", event_base_get_method(base));
        f = event_base_get_features(base);
        if ((f & EV_FEATURE_ET))
            printf("  Edge-triggered events are supported.\n");
        if ((f & EV_FEATURE_O1))
            printf("  O(1) event notification is supported.\n");
        if ((f & EV_FEATURE_FDS))
            printf("  All FD types are supported.\n");
        puts("");
    }
}
```

2 与事件一起工作

libevent的基本操作单元是事件。每个事件代表一组条件的集合，这些条件包括：

(1)文件描述符已经就绪，可以读取或者写入

(2)文件描述符变为就绪状态，可以读取或者写入（仅对于边沿触发IO）

(3)超时事件

(4)发生某信号

(5)用户触发事件

所有事件具有相似的生命周期。调用libevent函数设置事件并且关联到event_base之后，事件进入“已初始化（**initialized**）”状态。此时可以将事件添加到event_base中，这使之进入“未决（**pending**）”状态。

在未决状态下，如果触发事件的条件发生（比如说，文件描述符的状态改变，或者超时时间到达），则事件进入“激活（**active**）”状态，（用户提供的）事件回调函数将被执行。

如果配置为“持久的（**persistent**）”，事件将保持为未决状态。否则，执行完回调后，事件不再是未决的。删除操作可以让未决事件成为非未决（已初始化）的；添加操作可以让非未决事件再次成为未决的。

2.1 创建事件

使用event_new（）接口创建事件。

```
struct event *event_new(struct event_base *base, evutil_socket_t fd,
    short what, event_callback_fn cb, void *arg);

/*这个标志表示某超时时间流逝后事件成为激活的。超时发生时，回调函数的what参数将带有这个标志。*/
#define EV_TIMEOUT      0x01

/*表示指定的文件描述符已经就绪，可以读取的时候，事件将成为激活的。*/
#define EV_READ         0x02

/*表示指定的文件描述符已经就绪，可以写入的时候，事件将成为激活的。*/
#define EV_WRITE        0x04

#define EV_SIGNAL       0x08 //用于实现信号检测
#define EV_PERSIST      0x10 //表示事件是“持久的”
#define EV_ET           0x20 //表示如果底层的event_base后端支持边沿触发事件，则事件应该是边沿触发的。这个标志影响EV_READ和EV_WRITE的语义。


typedef void (*event_callback_fn)(evutil_socket_t fd, short what, void * arg);

void event_free(struct event *event);
```

event_new()试图分配和构造一个用于base的新事件。what参数是上述标志的集合。如果fd非负，则它是将被观察其读写事件的文件。事件被激活时，libevent将调用cb函数，传递这些参数：文件描述符fd，表示所有被触发事件的位字段，以及构造事件时的arg参数。

发生内部错误，或者传入无效参数时，event_new（）将返回NULL。

要释放事件，调用event_free（）。

使用event_assign二次修改event的相关参数：

```
int event_assign(struct event *event, struct event_base *base,
    evutil_socket_t fd, short what,
    void (*callback)(evutil_socket_t, short, void *), void *arg);
```

除了event参数必须指向一个未初始化的事件之外，event_assign（）的参数与event_new（）的参数相同。成功时函数返回0，如果发生内部错误或者使用错误的参数，函数返回-1。

警告：

不要对已经在**event_base**中未决的事件调用**event_assign**（），这可能会导致难以诊断的错误。如果已经初始化和成为未决的，调用**event_assign**（）之前需要调用**event_del**（）。

2.2 让事件未决和非未决

2.2.1 让事件未决

所有新创建的事件都处于已初始化和非未决状态，调用event_add()可以使其成为未决的。

```
int event_add(struct event *ev, const struct timeval *tv);
```

在非未决的事件上调用event_add（）将使其在配置的event_base中成为未决的。成功时函数返回0，失败时返回-1。如果tv为NULL，添加的事件不会超时。否则，tv以秒和微秒指定超时值。

如果对已经未决的事件调用event_add（），事件将保持未决状态，并在指定的超时时间被重新调度。

2.2.2 让事件非未决

```
int event_del(struct event *ev);
```

对已经初始化的事件调用event_del（）将使其成为非未决和非激活的。如果事件不是未决的或者激活的，调用将没有效果。成功时函数返回0，失败时返回-1。

2.3 设置事件的优先级

多个事件同时触发时，libevent没有定义各个回调的执行次序。可以使用优先级来定义某些事件比其他事件更重要。

```
int event_priority_set(struct event *event, int priority);
```

示例代码：

```
#include <event2/event.h>

void read_cb(evutil_socket_t, short, void *);
void write_cb(evutil_socket_t, short, void *);

void main_loop(evutil_socket_t fd)
{
  struct event *important, *unimportant;
  struct event_base *base;
  base = event_base_new();
  event_base_priority_init(base, 2);

  important = event_new(base, fd, EV_WRITE|EV_PERSIST, write_cb, NULL);
  unimportant = event_new(base, fd, EV_READ|EV_PERSIST, read_cb, NULL);
  event_priority_set(important, 0);
  event_priority_set(unimportant, 1);
}
```

2.4 关于事件持久性

默认情况下，每当未决事件成为激活的（因为fd已经准备好读取或者写入，或者因为超时），事件将在其回调被执行前成为非未决的。

如果想让事件再次成为未决的，可以在回调函数中再次对其调用event_add（）。然而，如果设置了EV_PERSIST标志，事件就是持久的。这意味着即使其回调被激活，事件还是会保持为未决状态。如果想在回调中让事件成为非未决的，可以对其调用event_del（）。

每次执行事件回调的时候，持久事件的超时值会被复位。因此，如果具有EV_READ|EV_PERSIST标志，以及5秒的超时值，则事件将在以下情况下成为激活的：

1. 套接字已经准备好被读取的时候
2. 从最后一次成为激活的开始，已经逝去5秒

2.5 事件循环

一旦有了一个已经注册了某些事件的event_base，就需要让libevent等待事件并且通知事件的发生。

2.5.1 启动事件循环

```
#define EVLOOP_ONCE             0x01
#define EVLOOP_NONBLOCK         0x02
int event_base_loop(struct event_base *base, int flags);
int event_base_dispatch(struct event_base *base);
```

默认情况下，event_base_loop（）函数运行event_base直到其中没有已经注册的事件为止。

执行循环的时候，函数重复地检查是否有任何已经注册的事件被触发（比如说，读事件的文件描述符已经就绪，可以读取了；或者超时事件的超时时间即将到达）。如果有事件被触发，函数标记被触发的事件为“激活的”，并且执行这些事件。

在flags参数中设置一个或者多个标志就可以改变event_base_loop（）的行为。如果设置了EVLOOP_ONCE，循环将等待某些事件成为激活的，执行激活的事件直到没有更多的事件可以执行，然后返回。

如果设置了EVLOOP_NONBLOCK，循环不会等待事件被触发：循环将仅仅检测是否有事件已经就绪，可以立即触发，如果有，则执行事件的回调。

完成工作后，如果正常退出，event_base_loop（）返回0；如果因为后端中的某些未处理错误而退出，则返回-1。

event_base_dispatch（）等同于没有设置标志的event_base_loop（）。

示例代码：

```
#include <event2/event.h>

void cb_func(evutil_socket_t fd, short what, void *arg)
{
        const char *data = arg;
        printf("Got an event on socket %d:%s%s [%s]\n",
            (int) fd,
            (what&EV_TIMEOUT) ? " timeout" : "",
            (what&EV_READ)    ? " read" : "",
            data);
}

void main_loop(evutil_socket_t fd)
{
        struct event *ev;
        struct timeval five_seconds = {5,0};
        struct event_base *base = event_base_new();
        ev = event_new(base, fd, EV_TIMEOUT|EV_READ|EV_PERSIST, cb_func, (char*)"Reading event");
        event_add(ev, &five_seconds);
        event_base_dispatch(base);
}
```

2.5.2 停止事件循环

如果想在移除所有已注册的事件之前停止活动的事件循环，可以调用两个稍有不同的函数。

```
int event_base_loopexit(struct event_base *base, const struct timeval *tv);
int event_base_loopbreak(struct event_base *base);
```

event_base_loopexit（）让event_base在给定时间之后停止循环。如果tv参数为NULL，event_base会立即停止循环，没有延时。如果event_base当前正在执行任何激活事件的回调，则回调会继续运行，直到运行完所有激活事件的回调之后才退出。

event_base_loopbreak（）让event_base立即退出循环。它与event_base_loopexit（base,NULL）的不同在于，如果event_base当前正在执行激活事件的回调，它将在执行完当前正在处理的事件后立即退出。

注意event_base_loopexit(base,NULL)和event_base_loopbreak(base)在事件循环没有运行时的行为不同：前者安排下一次事件循环在下一轮回调完成后立即停止（就好像带EVLOOP_ONCE标志调用一样）；后者却仅仅停止当前正在运行的循环，如果事件循环没有运行，则没有任何效果。

这两个函数都在成功时返回0，失败时返回-1。

示例：立即关闭

```
#include <event2/event.h>

void cb(int sock, short what, void *arg)
{
    struct event_base *base = arg;
    event_base_loopbreak(base);
}

void main_loop(struct event_base *base, evutil_socket_t watchdog_fd)
{
    struct event *watchdog_event;
    watchdog_event = event_new(base, watchdog_fd, EV_READ, cb, base);
    event_add(watchdog_event, NULL);
    event_base_dispatch(base);
}
```

示例: 执行事件循环10秒，然后退出

```
#include <event2/event.h>

void run_base_with_ticks(struct event_base *base)
{
  struct timeval ten_sec;
  ten_sec.tv_sec = 10;
  ten_sec.tv_usec = 0;

  while (1) {
     event_base_loopexit(base, &ten_sec);
     event_base_dispatch(base);
     puts("Tick");
  }
}
```

2.5.3 检查循环是否退出

有时候需要知道对event_base_dispatch（）或者event_base_loop（）的调用是正常退出的，还是因为调用event_base_loopexit（）或者event_base_break（）而退出的。可以调用下述函数来确定是否调用了loopexit或者break函数。

```
int event_base_got_exit(struct event_base *base);
int event_base_got_break(struct event_base *base);
```

这两个函数分别会在循环是因为调用event_base_loopexit（）或者event_base_break（）而退出的时候返回true，否则返回false。下次启动事件循环的时候，这些值会被重设。

2.6 配置一次触发事件

如果不需要多次添加一个事件，或者要在添加后立即删除事件，而事件又不需要是持久的，则可以使用event_base_once（）。

```
int event_base_once(struct event_base *, evutil_socket_t, short,
  void (*)(evutil_socket_t, short, void *), void *, const struct timeval *);
```

除了不支持EV_SIGNAL或者EV_PERSIST之外，这个函数的接口与event_new（）相同。安排的事件将以默认的优先级加入到event_base并执行。回调被执行后，libevent内部将会释放event结构。成功时函数返回0，失败时返回-1。

2.7 检查事件状态

有时候需要了解事件是否已经添加，检查事件代表什么。

```
/*event_get_fd（）返回为事件配置的文件描述符*/
evutil_socket_t event_get_fd(const struct event *ev);



/*event_get_base（）返回为事件配置的event_base。*/
struct event_base *event_get_base(const struct event *ev);

/*event_get_events（）返回事件的标志（EV_READ、EV_WRITE等）。*/
short event_get_events(const struct event *ev);

/*event_get_callback（）和event_get_callback_arg（）返回事件的回调函数及其参数指针。*/
event_callback_fn event_get_callback(const struct event *ev);
void *event_get_callback_arg(const struct event *ev);

/*获取事件的优先级*/
int event_get_priority(const struct event *ev);
```

2.8 tcp示例代码

服务器端：tcp_server.c

```
/*************************************************************************
# File Name: tcp_server.c
# Author: wenong
# mail: huangwenlong@520it.com
# Created Time: 2016年09月03日 星期六 21时51分08秒
 ************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <ctype.h>
#include <arpa/inet.h>
#include <signal.h>
#include <errno.h>
#include <sys/wait.h>
#include <event2/event.h>
#define SERVERPORT 8888
#define MAXBYTES 1024

void  read_cb(evutil_socket_t clientfd, short what, void* arg)
{
    int i, recvlen;
    char buf[MAXBYTES];
    struct event* ev = (struct event*)arg;
    printf("read_cd clientfd %d\n", clientfd);
    recvlen = read(clientfd, buf, sizeof(buf));
    if(recvlen <= 0)
    {
        puts("the other size close or error occur");
        event_del(ev);
        event_free(ev);
        close(clientfd);
    }
    for(i = 0; i < recvlen; i++)
    {
        buf[i] = toupper(buf[i]);
    }
    write(clientfd, buf, recvlen);
}


void  accept_cb(evutil_socket_t serverfd, short what, void * arg)
{

    struct sockaddr_in clientaddr;
    struct event* ev;
    int clientaddrlen;
    int clientfd;
    puts("Accept client connect");

    clientaddrlen = sizeof(clientaddr);
```

```
    bzero((void*)&clientaddr, sizeof(clientaddr));
    clientfd  = accept(serverfd, (struct sockaddr*)&clientaddr, &clientaddrlen);
    printf("recv clientfd %d\n", clientfd);
    ev = event_new((struct event_base*)arg, clientfd, EV_READ  | EV_PERSIST, NULL, NULL);
    event_assign(ev, (struct event_base*)arg, clientfd, EV_READ | EV_PERSIST,
(event_callback_fn)read_cb, (void*)ev);

    event_add(ev, NULL);

}

void main_loop(evutil_socket_t fd)
{
    struct event_base * base;
    struct event* ev;
    base = event_base_new();
    ev = event_new(base, fd, EV_READ | EV_PERSIST, (event_callback_fn)accept_cb, (void*)base);
    event_add(ev, NULL);
    puts("server begin listenning...");
    event_base_dispatch(base);
    event_free(ev);
    event_base_free(base);
}


int main(int argc, char** argv)
{
    int serverfd;
    socklen_t serveraddrlen;
    struct sockaddr_in serveraddr;
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(SERVERPORT);
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serverfd = socket(AF_INET, SOCK_STREAM, 0);
    serveraddrlen = sizeof(serveraddr);
    bind(serverfd, (struct sockaddr*)&serveraddr, serveraddrlen);
    listen(serverfd, 128);
    main_loop(serverfd);
    close(serverfd);
    return 0;
}
```

客户端: tcp_client.c

```
/*************************************************************************
# File Name: tcp_client.c
# Author: wenong
# mail: huangwenlong@520it.com
# Created Time: 2016年09月03日 星期六 22时10分11秒
 ************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <event2/event.h>
#define SERVERIP "127.0.0.1"
#define SERVERPORT 8888
#define MAXBYTES 1024
void read_cb(evutil_socket_t clientfd, short what, void* arg)
{
    puts("read clientfd");
    char buf[MAXBYTES];
    int ret;
    ret = read(clientfd, buf, sizeof(buf));
    if(ret <= 0)
    {
        close(clientfd);
        event_base_loopexit((struct event_base*)arg, NULL);
    }
    write(STDOUT_FILENO, buf, ret);
}

void cmd_msg_cb(evutil_socket_t stdinfd, short what, void* arg)
{
    int ret;
    int clientfd = (int)arg;
    char buf[MAXBYTES];
    puts("get msg from stdin");
    ret = read(stdinfd, buf, sizeof(buf));
    write(clientfd, buf, ret);
}

void main_loop(int clientfd)
{
    struct event_base* base;
    struct event* ev_socket, *ev_stdin;
    base = event_base_new();
    ev_stdin = event_new(base, STDIN_FILENO, EV_READ | EV_PERSIST
            , (event_callback_fn)cmd_msg_cb, (void*)clientfd);

    ev_socket = event_new(base, clientfd, EV_READ | EV_PERSIST
            , (event_callback_fn)read_cb, (void*)base);
```

```
    event_add(ev_socket, NULL);
    event_add(ev_stdin, NULL);
    event_base_dispatch(base);

    puts("event free and exit");
    event_free(ev_socket);
    event_free(ev_stdin);
    event_base_free(base);
}
int main(int argc, char** argv)
{
    int clientfd;
    struct sockaddr_in serveraddr;
    serveraddr.sin_family = AF_INET;
    inet_pton(AF_INET, SERVERIP, &serveraddr.sin_addr.s_addr);
    serveraddr.sin_port = htons(SERVERPORT);
    clientfd = socket(AF_INET, SOCK_STREAM, 0);
    connect(clientfd, (struct sockaddr*)&serveraddr, sizeof(serveraddr));

    main_loop(clientfd);
    return 0;
}
```

3 数据缓冲

很多时候，除了响应事件之外，应用还希望做一定的数据缓冲。比如说，写入数据的时候，通常的运行模式是：

（1）决定要向连接写入一些数据，把数据放入到缓冲区中

（2）等待连接可以写入

（3）写入尽量多的数据

（4）记住写入了多少数据，如果还有更多数据要写入，等待连接再次可以写入

这种缓冲IO模式很通用，libevent为此提供了一种通用机制，即bufferevent。bufferevent由一个底层的传输端口（如套接字），一个读取缓冲区和一个写入缓冲区组成。与通常的事件在底层传输端口已经就绪，可以读取或者写入的时候执行回调不同的是，bufferevent在读取或者写入了足够量的数据之后调用用户提供的回调。

3.1 bufferevent和evbuffer

每个bufferevent都有一个输入缓冲区和一个输出缓冲区，它们的类型都是“struct evbuffer”。有数据要写入到bufferevent时，添加数据到输出缓冲区；bufferevent中有数据供读取的时候，从输入缓冲区抽取（drain）数据。

3.2 回调水位

每个bufferevent有两个数据相关的回调：一个读取回调和一个写入回调。默认情况下，从底层传输端口读取了任意量的数据之后会调用读取回调；输出缓冲区中足够量的数据被清空到底层传输端口后写入回调会被调用。通过调整bufferevent的读取和写入“水位（watermarks）”可以覆盖这些函数的默认行为。

每个bufferevent有四个水位：

（1）读取低水位：读取操作使得输入缓冲区的数据量在此级别或者更高时，读取回调将被调用。默认值为0，所以每个读取操作都会导致读取回调被调用。

（2）读取高水位：输入缓冲区中的数据量达到此级别后，bufferevent将停止读取，直到输入缓冲区中足够量的数据被抽取，使得数据量低于此级别。默认值是无限，所以永远不会因为输入缓冲区的大小而停止读取。

（3）写入低水位：写入操作使得输出缓冲区的数据量达到或者低于此级别时，写入回调将被调用。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。

（4）写入高水位：bufferevent没有直接使用这个水位。它在bufferevent用作另外一个bufferevent的底层传输端口时有特殊意义。

bufferevent也有“错误”或者“事件”回调，用于向应用通知非面向数据的事件，如连接已经关闭或者发生错误。定义了下列事件标志：

```
BEV_EVENT_READING：#读取操作时发生某事件，具体是哪种事件请看其他标志。
BEV_EVENT_WRITING：#写入操作时发生某事件，具体是哪种事件请看其他标志。
BEV_EVENT_ERROR：#操作时发生错误。关于错误的更多信息，请调用EVUTIL_SOCKET_ERROR()，获取错误号。
BEV_EVENT_TIMEOUT：#发生超时。
BEV_EVENT_EOF：#遇到文件结束指示。
BEV_EVENT_CONNECTED：#请求的连接过程已经完成。
```

3.3 延迟回调

默认情况下，bufferevent的回调在相应的条件发生时立即被执行。（evbuffer的回调也是这样的）在依赖关系复杂的情况下，这种立即调用会制造麻烦。比如说，假如某个回调在evbuffer A空的时候向其中移入数据，而另一个回调在evbuffer A满的时候从中取出数据。这些调用都是在栈上发生的，在依赖关系足够复杂的时候，有栈溢出的风险。

要解决此问题，可以请求bufferevent（或者evbuffer）延迟其回调。条件满足时，延迟回调不会立即调用，而是在event_loop（）调用中被排队，然后在通常的事件回调之后执行。

3.4 bufferevent的选项标志

创建bufferevent时可以使用一个或者多个标志修改其行为。可识别的标志有：

```
BEV_OPT_CLOSE_ON_FREE：#释放bufferevent时关闭底层传输端口。这将关闭底层套接字，释放底层bufferevent等。
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，如上所述。
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。
```

3.5 与基于套接字的bufferevent一起工作

基于套接字的bufferevent是最简单的，它使用libevent的底层事件机制来检测底层网络套接字是否已经就绪，可以进行读写操作，并且使用底层网络调用（如readv、writev、WSASend、WSARecv）来发送和接收数据。

3.5.1 创建基于套接字的bufferevent

可以使用bufferevent_socket_new()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);
```

base是event_base，options是表示bufferevent选项（BEV_OPT_CLOSE_ON_FREE等）的位掩码，fd是一个可选的表示套接字的文件描述符。如果想以后设置文件描述符，可以设置fd为-1。

成功时函数返回一个bufferevent，失败则返回NULL。

3.5.2 在基于套接字的**bufferevent**上启动连接

如果bufferevent的套接字还没有连接上，可以启动新的连接。

```
int bufferevent_socket_connect(struct bufferevent *bev
                      , struct sockaddr *address, int addrlen);
```

address和addrlen参数跟标准调用connect()的参数相同。如果还没有为bufferevent设置套接字，调用函数将为其分配一个新的流套接字，并且设置为非阻塞的。

如果已经为bufferevent设置套接字，调用bufferevent_socket_connect()将告知libevent套接字还未连接，直到连接成功之前不应该对其进行读取或者写入操作。

连接完成之前可以向输出缓冲区添加数据。

如果连接成功启动，函数返回0；如果发生错误则返回-1。

注意：如果使用**bufferevent_socket_connect()**发起连接，将只会收到**BEV_EVENT_CONNECTED**事件。如果自己调用**connect()**，则连接上将被报告为写入事件。

3.6 通用**bufferevent**操作

3.6.1 释放**bufferevent**

```
void bufferevent_free(struct bufferevent *bev);
```

这个函数释放bufferevent。bufferevent内部具有引用计数，所以，如果释放bufferevent时还有未决的延迟回调，则在回调完成之前bufferevent不会被删除。

如果设置了BEV_OPT_CLOSE_ON_FREE标志，并且bufferevent有一个套接字或者底层bufferevent作为其传输端口，则释放bufferevent将关闭这个传输端口。

3.6.2 操作回调、水位和启用/禁用

```
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);

void bufferevent_getcb(struct bufferevent *bufev,
    bufferevent_data_cb *readcb_ptr,
    bufferevent_data_cb *writecb_ptr,
    bufferevent_event_cb *eventcb_ptr,
    void **cbarg_ptr);
```

bufferevent_setcb()函数修改bufferevent的一个或者多个回调。readcb、writecb和eventcb函数将分别在已经读取足够的数据、已经写入足够的数据，或者发生错误时被调用。每个回调函数的第一个参数都是发生了事件的bufferevent，最后一个参数都是调用bufferevent_setcb()时用户提供的cbarg参数：可以通过它向回调传递数据。事件回调的events参数是一个表示事件标志的位掩码：请看前面的“回调和水位”节。

要禁用回调，传递NULL而不是回调函数。

注意：**bufferevent**的所有回调函数共享单个**cbarg**，所以修改它将影响所有回调函数。

```
void bufferevent_enable(struct bufferevent *bufev, short events);
void bufferevent_disable(struct bufferevent *bufev, short events);
short bufferevent_get_enabled(struct bufferevent *bufev);
```

可以启用或者禁用bufferevent上的EV_READ、EV_WRITE或者EV_READ | EV_WRITE事件。没有启用读取或者写入事件时，bufferevent将不会试图进行数据读取或者写入。

没有必要在输出缓冲区空时禁用写入事件：bufferevent将自动停止写入，然后在有数据等待写入时重新开始。

类似地，没有必要在输入缓冲区高于高水位时禁用读取事件：bufferevent将自动停止读取，然后在有空间用于读取时重新开始读取。

默认情况下，新创建的bufferevent的写入是启用的，但是读取没有启用。

可以调用bufferevent_get_enabled()确定bufferevent上当前启用的事件。

```
void bufferevent_setwatermark(struct bufferevent *bufev, short events,
    size_t lowmark, size_t highmark);
```

bufferevent_setwatermark()函数调整单个bufferevent的读取水位、写入水位，或者同时调整二者。（如果events参数设置了EV_READ，调整读取水位。如果events设置了EV_WRITE标志，调整写入水位）对于高水位，0表示“无限”。

3.6.3 操作**bufferevent**中的数据

- 写数据

  如果写入操作因为数据量太少而停止（或者读取操作因为太多数据而停止），则向输出缓冲区添加数据（或者从输入缓冲区移除数据）将自动重启操作。

```
int bufferevent_write(struct bufferevent *bufev,
    const void *data, size_t size);
```

这个函数向bufferevent的输出缓冲区添加数据。bufferevent_write()将内存中从data处开始的size字节数据添加到输出缓冲区的末尾。成功时这些函数都返回0，发生错误时则返回-1。

注意：即使没有调用**bufferevent_enable**使能写事件，调用**bufferevent_write**时，内部会添加写事件的监控，触发写回调函数后，再把写事件清除掉，写回调函数返回时，这个时候检查一下输出缓冲区是否低于写低水位，低了就调用一下写**bufferevent**的回调函数。

- 读数据

```
size_t bufferevent_read(struct bufferevent *bufev, void *data, size_t size);
```

这个函数从bufferevent的输入缓冲区移除数据。bufferevent_read()至多从输入缓冲区移除size字节的数据，将其存储到内存中data处。

注意：对于**bufferevent_read()**，**data**处的内存块必须有足够的空间容纳**size**字节数据。

3.6.4 读写超时

跟其他事件一样，可以要求在一定量的时间已经流逝，而没有成功写入或者读取数据的时候调用一个超时回调。

```
void bufferevent_set_timeouts(struct bufferevent *bufev,
    const struct timeval *timeout_read, const struct timeval *timeout_write);
```

设置超时为NULL会移除超时回调。

试图读取数据的时候，如果至少等待了timeout_read秒，则读取超时事件将被触发。试图写入数据的时候，如果至少等待了timeout_write秒，则写入超时事件将被触发。

注意，只有在读取或者写入的时候才会计算超时。也就是说，如果bufferevent的读取被禁止，或者输入缓冲区满（达到其高水位），则读取超时被禁止。类似的，如果写入被禁止，或者没有数据待写入，则写入超时被禁止。

读取或者写入超时发生时，相应的读取或者写入操作被禁止，然后超时事件回调被调用，带有标志BEV_EVENT_TIMEOUT | BEV_EVENT_READING或者BEV_EVENT_TIMEOUT | BEV_EVENT_WRITING。

3.6.5 对**bufferevent**发起清空操作

```
int bufferevent_flush(struct bufferevent *bufev,
    short iotype, enum bufferevent_flush_mode state)
```

清空bufferevent要求bufferevent强制从底层传输端口读取或者写入尽可能多的数据，而忽略其他可能保持数据不被写入的限制条件。函数的细节功能依赖于bufferevent的具体类型。

iotype参数应该是EV_READ、EV_WRITE或者EV_READ | EV_WRITE，用于指示应该处理读取、写入，还是二者都处理。

state参数可以是BEV_NORMAL、BEV_FLUSH或者BEV_FINISHED。

BEV_FINISHED指示应该告知另一端，没有更多数据需要发送了；

而BEV_NORMAL和BEV_FLUSH的区别依赖于具体的bufferevent类型。

失败时bufferevent_flush()返回-1，如果没有数据被清空则返回0，有数据被清空则返回1。

3.7 类型特定的bufferevent函数

这些bufferevent函数不能支持所有bufferevent类型。

```
int bufferevent_priority_set(struct bufferevent *bufev, int pri);
int bufferevent_get_priority(struct bufferevent *bufev);
```

这个函数调整bufev的优先级为pri。关于优先级的更多信息请看event_priority_set()。

成功时函数返回0，失败时返回-1。这个函数仅能用于基于套接字的bufferevent。

```
int bufferevent_setfd(struct bufferevent *bufev, evutil_socket_t fd);
evutil_socket_t bufferevent_getfd(struct bufferevent *bufev);
```

这些函数设置或者返回基于fd的事件的文件描述符。只有基于套接字的bufferevent支持setfd()。两个函数都在失败时返回-1；setfd()成功时返回0。

```
struct event_base *bufferevent_get_base(struct bufferevent *bev);
```

这个函数返回bufferevent的event_base。

3.8 手动锁定和解锁

有时候需要确保对bufferevent的一些操作是原子地执行的。为此，libevent提供了手动锁定和解锁bufferevent的函数。

```
void bufferevent_lock(struct bufferevent *bufev);
void bufferevent_unlock(struct bufferevent *bufev);
```

注意：如果创建**bufferevent**时没有指定**BEV_OPT_THREADSAFE**标志，或者没有激活**libevent**的线程支持，则锁定操作是没有效果的。

用这个函数锁定bufferevent将自动同时锁定相关联的evbuffer。这些函数是递归的：锁定已经持有锁的bufferevent是安全的。当然，对于每次锁定都必须进行一次解锁。

3.9 示例代码

服务器tcp_server.c：

```
/*************************************************************************
# File Name: tcp_server.c
# Author: wenong
# mail: huangwenlong@520it.com
# Created Time: 2016年09月03日 星期六 21时51分08秒
 ************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <ctype.h>
#include <arpa/inet.h>
#include <signal.h>
#include <errno.h>
#include <sys/wait.h>
#include <event2/event.h>
#include <event2/bufferevent.h>
#define SERVERPORT 8888
#define MAXBYTES 1024


void read_buf_cb(struct bufferevent* bev, void* cbarg)
{
    int ret, i;
    char buf[MAXBYTES];
    ret = bufferevent_read(bev, buf, sizeof(buf));
    printf("read_buf_cd length %d\n", ret);
    for(i = 0; i < ret; i++)
    {
        buf[i] = toupper(buf[i]);
    }
    bufferevent_write(bev, buf, ret);

}


void event_cb(struct bufferevent* bev, short event, void* cbarg)
{
    struct event_base* base = (struct event_base*)cbarg;
    if(BEV_EVENT_READING & event)
        puts("BEV_EVENT_READING");

    if(BEV_EVENT_WRITING & event)
        puts("BEV_EVENT_WRITING");

    if(BEV_EVENT_ERROR & event)
        puts("BEV_EVENT_ERROR");


    if(BEV_EVENT_EOF & event)
```

```
    {
        puts("BEV_EVENT_EOF");
        bufferevent_free(bev);
    }

}


void  accept_cb(evutil_socket_t serverfd, short what, void * arg)
{

    struct sockaddr_in clientaddr;
    struct event* ev;
    struct bufferevent* bev;
    struct event_base* base = (struct event_base*)arg;
    int clientaddrlen;
    int clientfd;
    puts("Accept client connect\n");
    clientaddrlen = sizeof(clientaddr);
    bzero((void*)&clientaddr, sizeof(clientaddr));
    clientfd  = accept(serverfd, (struct sockaddr*)&clientaddr, &clientaddrlen);
    printf("recv clientfd %d\n", clientfd);
    evutil_make_socket_nonblocking(clientfd);
    bev = bufferevent_socket_new(base, clientfd, BEV_OPT_CLOSE_ON_FREE
            | BEV_OPT_DEFER_CALLBACKS);
    bufferevent_setcb(bev, (bufferevent_data_cb)read_buf_cb
            , NULL, (bufferevent_event_cb)event_cb, (void*)base);
    bufferevent_enable(bev, EV_READ);
    bufferevent_setwatermark(bev, EV_READ, 10, 0);

}

void main_loop(evutil_socket_t fd)
{
    struct event_base * base;
    struct event* ev;
    base = event_base_new();
    ev = event_new(base, fd, EV_READ | EV_PERSIST, (event_callback_fn)accept_cb, (void*)base);
    event_add(ev, NULL);
    puts("server begin listenning\n");
    event_base_dispatch(base);
    event_free(ev);
    event_base_free(base);
}


int main(int argc, char** argv)
{
    int serverfd;
    socklen_t serveraddrlen;
    struct sockaddr_in serveraddr;
    serveraddr.sin_family = AF_INET;

    serveraddr.sin_port = htons(SERVERPORT);
```

```
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serverfd = socket(AF_INET, SOCK_STREAM, 0);
    serveraddrlen = sizeof(serveraddr);
    bind(serverfd, (struct sockaddr*)&serveraddr, serveraddrlen);
    listen(serverfd, 128);
    main_loop(serverfd);
    close(serverfd);
    return 0;
}
```

客户端tcp_client.c

```
/*************************************************************************
# File Name: tcp_client.c
# Author: wenong
# mail: huangwenlong@520it.com
# Created Time: 2016年09月03日 星期六 22时10分11秒
 ************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <event2/event.h>
#include <event2/bufferevent.h>
#define SERVERIP "127.0.0.1"
#define SERVERPORT 8888
#define MAXBYTES 1024

void* cmd_msg_cb(evutil_socket_t stdinfd, short what, void* arg)
{
    int ret;
    struct bufferevent* bev = (struct bufferevent*)arg;
    char buf[MAXBYTES];
    puts("get msg from stdin:");
    ret = read(stdinfd, buf, sizeof(buf));
    bufferevent_write(bev, buf, ret);
}


void read_buf_cb(struct bufferevent* bev, void* cbarg)
{
    int ret;
    char buf[MAXBYTES];
    ret = bufferevent_read(bev, buf, sizeof(buf));
    write(STDOUT_FILENO, buf, ret);
}

void event_cb(struct bufferevent* bev, short event, void* cbarg)
{
    struct event_base* base = (struct event_base*)cbarg;
    if(BEV_EVENT_READING & event)
        puts("BEV_EVENT_READING");

    if(BEV_EVENT_WRITING & event)
        puts("BEV_EVENT_WRITING");

    if(BEV_EVENT_ERROR & event)
        puts("BEV_EVENT_ERROR");


    if(BEV_EVENT_EOF & event)
```

```
    {
        puts("BEV_EVENT_EOF");
        event_base_loopexit(base, NULL);
       //event_base_loopexit(bufferevent_get_base(bev), NULL);
    }

}
void main_loop(int clientfd)
{
    struct event_base* base;
    struct bufferevent* bev;
    struct event*ev_stdin;
    base = event_base_new();


    bev = bufferevent_socket_new(base, clientfd, BEV_OPT_CLOSE_ON_FREE
            | BEV_OPT_DEFER_CALLBACKS);
    bufferevent_setcb(bev, (bufferevent_data_cb)read_buf_cb
            , NULL, (bufferevent_event_cb)event_cb, (void*)base);
    bufferevent_enable(bev, EV_READ);

    ev_stdin = event_new(base, STDIN_FILENO, EV_READ | EV_PERSIST
            , (event_callback_fn)cmd_msg_cb, (void*)bev);

    event_add(ev_stdin, NULL);

    event_base_dispatch(base);
    bufferevent_free(bev);
    event_free(ev_stdin);
    event_base_free(base);
    puts("exit now...");
}

int main(int argc, char** argv)
{
    int clientfd;
    struct sockaddr_in serveraddr;
    serveraddr.sin_family = AF_INET;
    inet_pton(AF_INET, SERVERIP, &serveraddr.sin_addr.s_addr);
    serveraddr.sin_port = htons(SERVERPORT);
    clientfd = socket(AF_INET, SOCK_STREAM, 0);
    connect(clientfd, (struct sockaddr*)&serveraddr, sizeof(serveraddr));
    main_loop(clientfd);
    return 0;
}
```

4 辅助类型和函数

4.1 基本类型

4.1.1 evutil_socket_t

在除Windows之外的大多数地方，套接字是个整数，操作系统按照数值次序进行处理。然而，使用Windows套接字API时，socket具有类型SOCKET，它实际上是个类似指针的句柄，收到这个句柄的次序是未定义的。在Windows中，libevent定义evutil_socket_t类型为整型指针，可以处理socket()或者accept()的输出，而没有指针截断的风险。

4.1.2 标准整数类型

Type	Width	Signed	Maximum	Minimum
ev_uint64_t	64	No	EV_UINT64_MAX	0
ev_int64_t	64	Yes	EV_INT64_MAX	EV_INT64_MIN
ev_uint32_t	32	No	EV_UINT32_MAX	0
ev_int32_t	32	Yes	EV_INT32_MAX	EV_INT32_MIN
ev_uint16_t	16	No	EV_UINT16_MAX	0
ev_int16_t	16	Yes	EV_INT16_MAX	EV_INT16_MIN
ev_uint8_t	8	No	EV_UINT8_MAX	0
ev_int8_t	8	Yes	EV_INT8_MAX	EV_INT8_MIN

4.2 套接字API

```
int evutil_closesocket(evutil_socket_t s);
#define EVUTIL_CLOSESOCKET(s) evutil_closesocket(s)
```

这个接口用于关闭套接字。在Unix中，它是close()的别名；

```
int evutil_make_socket_nonblocking(evutil_socket_t sock);
```

evutil_make_socket_nonblocking()函数要求一个套接字（来自socket()或者accept()）作为参数，将其设置为非阻塞的。（设置Unix中的O_NONBLOCK标志和Windows中的FIONBIO标志）

```
int evutil_make_listen_socket_reuseable(evutil_socket_t sock);
```

这个函数确保关闭监听套接字后，它使用的地址可以立即被另一个套接字使用。

5 连接监听器，接收tcp连接

5.1 创建和释放evconnlistener

```
#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);

void evconnlistener_free(struct evconnlistener *lev);
```

两个evconnlistener_new_bind()函数分配和返回一个新的连接监听器对象。连接监听器使用event_base来得知什么时候在给定的监听套接字上有新的TCP连接。新连接到达时，监听器调用你给出的回调函数。

两个函数中，base参数都是监听器用于监听连接的event_base。cb是收到新连接时要调用的回调函数；如果cb为NULL，则监听器是禁用的，直到设置了回调函数为止。ptr指针将传递给回调函数。flags参数控制回调函数的行为，下面会更详细论述。backlog是任何时刻网络栈允许处于还未接受状态的最大未决连接数。如果backlog是负的，libevent会试图挑选一个较好的值；如果为0，libevent认为已经对提供的套接字调用了listen()。

两个函数的不同在于如何建立监听套接字。evconnlistener_new()函数假定已经将套接字绑定到要监听的端口，然后通过fd传入这个套接字。如果要libevent分配和绑定套接字，可以调用evconnlistener_new_bind()，传输要绑定到的地址和地址长度。

可以给evconnlistener_new()函数的flags参数传入一些标志。可以用或(OR)运算任意连接下述标志：

```
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，以便将其用于libevent。如果不想要这种行为，可以设置这个标志。*/

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
/*设置这个标志会让libevent标记套接字是可重用的，这样一旦关闭，可以立即打开其他套接字，在相同端口进行监听。*/

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。*/
```

连接监听器回调函数:

```
typedef void (*evconnlistener_cb)(struct evconnlistener *listener,
    evutil_socket_t sock, struct sockaddr *addr, int len, void *ptr)
```

接收到新连接会调用提供的回调函数。listener参数是接收连接的连接监听器。sock参数是新接收的套接字。addr和len参数是接收连接的地址和地址长度。ptr是调用evconnlistener_new()时用户提供的指针。

要释放连接监听器，调用evconnlistener_free()。

5.2 启用和禁用evconnlistener

```
int evconnlistener_disable(struct evconnlistener *lev);
int evconnlistener_enable(struct evconnlistener *lev);
```

这两个函数暂时禁止或者重新允许监听新连接。

5.3 调整evconnlistener的回调函数

```
void evconnlistener_set_cb(struct evconnlistener *lev,
    evconnlistener_cb cb, void *arg);
```

函数调整evconnlistener的回调函数和其参数。

5.4 获取event_base

```
struct event_base *evconnlistener_get_base(struct evconnlistener *lev);
```

5.5示例代码

服务器端：tcp_server.c

```c
/*************************************************************************
# File Name: tcp_server.c
# Author: wenong
# mail: huangwenlong@520it.com
# Created Time: 2016年09月03日 星期六 21时51分08秒
 ************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <ctype.h>
#include <arpa/inet.h>
#include <signal.h>
#include <errno.h>
#include <sys/wait.h>
#include <event2/event.h>
#include <event2/bufferevent.h>
#include <event2/listener.h>
#define SERVERPORT 8888
#define MAXBYTES 1024

static struct event_base * base;

void write_buf_cb(struct bufferevent* bev, void* cbarg)
{
    printf("%s\n", __FUNCTION__);
}

void read_buf_cb(struct bufferevent* bev, void* cbarg)
{
    int ret, i;
    char buf[MAXBYTES];
    ret = bufferevent_read(bev, buf, sizeof(buf));
    printf("read_buf_cd length %d\n", ret);
    for(i = 0; i < ret; i++)
    {
        buf[i] = toupper(buf[i]);
    }
    bufferevent_write(bev, buf, ret);

}


void event_cb(struct bufferevent* bev, short event, void* cbarg)
{
    if(BEV_EVENT_READING & event)
        puts("BEV_EVENT_READING");


    if(BEV_EVENT_WRITING & event)
```

```
        puts("BEV_EVENT_WRITING");

    if(BEV_EVENT_ERROR & event)
        puts("BEV_EVENT_ERROR");

    if(BEV_EVENT_EOF & event)
    {
        puts("BEV_EVENT_EOF");
        bufferevent_free(bev);
    }
    if(BEV_EVENT_TIMEOUT & event)
    {
       puts("BEV_EVENT_TIMEOUT");
        bufferevent_free(bev);
    }
}


void  accept_cb(struct evconnlistener *listener,
            evutil_socket_t clientfd, struct sockaddr *addr
            , int len, void *arg)
{

    struct bufferevent* bev;
    struct event_base* base = (struct event_base*)arg;
    puts("Accept client connect");

    evutil_make_socket_nonblocking(clientfd);
    bev = bufferevent_socket_new(base, clientfd, BEV_OPT_CLOSE_ON_FREE
            | BEV_OPT_DEFER_CALLBACKS);
    bufferevent_setcb(bev, (bufferevent_data_cb)read_buf_cb
            , (bufferevent_data_cb)write_buf_cb, (bufferevent_event_cb)event_cb, NULL);

    struct timeval timeout_read = {10, 0};
    bufferevent_set_timeouts(bev, &timeout_read, NULL);
    bufferevent_setwatermark(bev, EV_READ, 10, 0);
    bufferevent_setwatermark(bev, EV_WRITE, 10, 0);

    bufferevent_enable(bev, EV_READ | EV_WRITE);
}

void main_loop(struct sockaddr_in * addr)
{

    struct evconnlistener *evcon;
    base = event_base_new();
    evcon = evconnlistener_new_bind(base, (evconnlistener_cb)accept_cb
            , (void*)base, LEV_OPT_CLOSE_ON_FREE | LEV_OPT_REUSEABLE
            , 128, (struct sockaddr*)addr, sizeof(struct sockaddr_in));

    puts("server begin listenning...");
```

```
    event_base_dispatch(base);
    evconnlistener_free(evcon);
    event_base_free(base);

}


int main(int argc, char** argv)
{
    int serverfd;
    socklen_t serveraddrlen;
    struct sockaddr_in serveraddr;
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(SERVERPORT);
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serverfd = socket(AF_INET, SOCK_STREAM, 0);
    serveraddrlen = sizeof(serveraddr);
    main_loop( &serveraddr);

    return 0;
}
```

客户端代码：tcp_client.c

```c
/*************************************************************************
# File Name: tcp_client.c
# Author: wenong
# mail: huangwenlong@520it.com
# Created Time: 2016年09月03日 星期六 22时10分11秒
 ************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <errno.h>
#include <event2/event.h>
#include <event2/bufferevent.h>
#define SERVERIP "127.0.0.1"
#define SERVERPORT 8888
#define MAXBYTES 1024
static struct sockaddr_in serveraddr;
static struct event_base* base;
static struct bufferevent* bev;

void* cmd_msg_cb(evutil_socket_t stdinfd, short what, void* arg)
{
    int ret;
    char buf[MAXBYTES];
    ret = read(stdinfd, buf, sizeof(buf));
    bufferevent_write(bev, buf, ret);
}

void read_buf_cb(struct bufferevent* bev, void* cbarg)
{
    int ret;
    char buf[MAXBYTES];
    ret = bufferevent_read(bev, buf, sizeof(buf));
    write(STDOUT_FILENO, buf, ret);
}

void event_cb(struct bufferevent* bev, short event, void* cbarg)
{
    if(BEV_EVENT_READING & event)
        puts("BEV_EVENT_READING");

    if(BEV_EVENT_WRITING & event)
        puts("BEV_EVENT_WRITING");

    if(BEV_EVENT_ERROR & event)
        printf("BEV_EVENT_ERROR %s\n", strerror(EVUTIL_SOCKET_ERROR())); //获取错误信息


    if(BEV_EVENT_CONNECTED & event)
```

```
        puts("BEV_EVENT_CONNECTED");

    if(BEV_EVENT_EOF & event)
    {
        puts("BEV_EVENT_EOF");
        event_base_loopexit(base, NULL);
    }
    if(BEV_EVENT_TIMEOUT & event)
    {
        puts("BEV_EVENT_TIMEOUT");
        event_base_loopexit(base, NULL);
    }


}
void main_loop()
{
    struct event*ev_stdin;

    base = event_base_new();
    bev = bufferevent_socket_new(base, -1, BEV_OPT_CLOSE_ON_FREE
            | BEV_OPT_DEFER_CALLBACKS);

    bufferevent_setcb(bev, (bufferevent_data_cb)read_buf_cb
            , NULL, (bufferevent_event_cb)event_cb, NULL);

    bufferevent_socket_connect(bev, (struct sockaddr*)&serveraddr, sizeof(struct sockaddr_in));

    struct timeval timeout_read;
    timeout_read.tv_sec = 10;
    timeout_read.tv_usec = 0;
    bufferevent_set_timeouts(bev, &timeout_read, NULL);

    bufferevent_enable(bev, EV_READ);


    ev_stdin = event_new(base, STDIN_FILENO, EV_READ | EV_PERSIST
            , (event_callback_fn)cmd_msg_cb, NULL);
    event_add(ev_stdin, NULL);

    event_base_dispatch(base);
    bufferevent_free(bev);
    event_free(ev_stdin);
    event_base_free(base);
    puts("exit now...");
}

int main(int argc, char** argv)
{
    serveraddr.sin_family = AF_INET;
    inet_pton(AF_INET, SERVERIP, &serveraddr.sin_addr.s_addr);
    serveraddr.sin_port = htons(SERVERPORT);

    main_loop();
```

```
    return 0;
}
```

6 从bufferevent中取出evbuffer

```
struct evbuffer* bufferevent_get_input(struct bufferevent *bufev); //取出输入缓冲区
struct evbuffer* bufferevent_get_output(struct bufferevent *bufev); //取出输出缓冲区

/*对evbuffer的操作*/

//读到的一行内容
char *evbuffer_readln(struct evbuffer*buffer, size_t *n_read_out,enum evbuffer_eol_style
eol_style);
enum evbuffer_eol_style {
    EVBUFFER_EOL_ANY,           // 任意数量的\r和\n
    EVBUFFER_EOL_CRLF,          // \r或者\r\n
    EVBUFFER_EOL_CRLF_STRICT,   // \r\n
    EVBUFFER_EOL_LF,            // \n
    EVBUFFER_EOL_NUL            // \0
};

//将数据添加到evbuffer的结尾
int evbuffer_add(struct evbuffer *buf,const void *data, size_t datlen);

//从evbuffer读取数据到data
int evbuffer_remove(struct evbuffer*buf, void *data, size_t datlen);

//丢掉len字节的数据
int evbuffer_drain (struct evbuffer *buf, size_t len);

//返回evbuffer中存储的字节长度
size_t evbuffer_get_length(const structevbuffer *buf);

/*在buffer中搜索指定字符串
第一个参数是搜索在evbuffer中，
第二个参数what是要搜索的字符串。
第三个参数len为what的字符串长度，
第四个参数，如果start不为空，则会从start中所指定的位置开始搜索，为NULL则从头开始找。
返回值：如果找到struct evbuffer_ptr的成员pos返回对应的索引，没有找到则pos返回-1*/
struct evbuffer_ptr evbuffer_search (struct evbuffer *buffer,
                                     const char *what,
                                     size_t len,
                                     const struct evbuffer_ptr *start);
//第二个函数的不同是指定了一个搜索范围；
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 {
    ev_ssize_t pos;  //位置
    struct {/* internal fields */} _internal;
};
```

服务器综合代码：

```
/*************************************************************************
# File Name: tcp_server.c
# Author: wenong
# mail: huangwenlong@520it.com
# Created Time: 2016年09月03日 星期六 21时51分08秒
 ************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <ctype.h>
#include <arpa/inet.h>
#include <signal.h>
#include <errno.h>
#include <sys/wait.h>
#include <event2/event.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>
#include <event2/listener.h>
#define SERVERPORT 8888
#define MAXBYTES 1024

static struct event_base * base;

void write_buf_cb(struct bufferevent* bev, void* cbarg)
{
    printf("%s\n", __FUNCTION__);
}

void read_buf_cb(struct bufferevent* bev, void* cbarg)
{
    int ret, i;
    char buf[MAXBYTES];
    struct evbuffer* input_evbuffer = bufferevent_get_input(bev);

    struct evbuffer_ptr begin, end;
    while(1)
    {
        begin = evbuffer_search(input_evbuffer, "a", 1, NULL);
        printf("find a index %ld\n", begin.pos);
        if(begin.pos >= 0)
        {
            evbuffer_drain(input_evbuffer, begin.pos);
            end = evbuffer_search(input_evbuffer, "b", 1, NULL);
            if(end.pos > 0)
            {
                ret = evbuffer_remove(input_evbuffer, buf, end.pos - begin.pos + 1);
                buf[ret] = '\0';

                printf("read_buf_cd length %d\n", ret);
```

```
                for(i = 0; i < ret; i++)
                    buf[i] = toupper(buf[i]);
                bufferevent_write(bev, buf, ret);
            }
            else
                break;
        }
        else
        {
            bufferevent_flush(bev, EV_READ, BEV_NORMAL);
            break;
        }

    }

}
void event_cb(struct bufferevent* bev, short event, void* cbarg)
{
    if(BEV_EVENT_READING & event)
        puts("BEV_EVENT_READING");

    if(BEV_EVENT_WRITING & event)
        puts("BEV_EVENT_WRITING");

    if(BEV_EVENT_ERROR & event)
        puts("BEV_EVENT_ERROR");

    if(BEV_EVENT_EOF & event)
    {
        puts("BEV_EVENT_EOF");
        bufferevent_free(bev);
    }
    if(BEV_EVENT_TIMEOUT & event)
    {
        puts("BEV_EVENT_TIMEOUT");
        bufferevent_free(bev);
    }
}


void  accept_cb(struct evconnlistener *listener,
        evutil_socket_t clientfd, struct sockaddr *addr
        , int len, void *arg)
{

    struct bufferevent* bev;
    struct event_base* base = (struct event_base*)arg;
    puts("Accept client connect");
    evutil_make_socket_nonblocking(clientfd);
    bev = bufferevent_socket_new(base, clientfd, BEV_OPT_CLOSE_ON_FREE
            | BEV_OPT_DEFER_CALLBACKS);
    bufferevent_setcb(bev, (bufferevent_data_cb)read_buf_cb

            , (bufferevent_data_cb)write_buf_cb, (bufferevent_event_cb)event_cb, NULL);
```

```
    struct timeval timeout_read;
    timeout_read.tv_sec = 60;
    timeout_read.tv_usec = 0;
    bufferevent_set_timeouts(bev, &timeout_read, NULL);
    bufferevent_setwatermark(bev, EV_READ, 10, 0);

    bufferevent_enable(bev, EV_READ);
}

void main_loop(struct sockaddr_in * addr)
{

    struct evconnlistener *evcon;
    base = event_base_new();
    evcon = evconnlistener_new_bind(base, (evconnlistener_cb)accept_cb
            , (void*)base, LEV_OPT_CLOSE_ON_FREE | LEV_OPT_REUSEABLE
            , 128, (struct sockaddr*)addr, sizeof(struct sockaddr_in));

    puts("server begin listenning...");


    event_base_dispatch(base);
    evconnlistener_free(evcon);
    event_base_free(base);

}


int main(int argc, char** argv)
{
    int serverfd;
    socklen_t serveraddrlen;
    struct sockaddr_in serveraddr;
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(SERVERPORT);
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serverfd = socket(AF_INET, SOCK_STREAM, 0);
    serveraddrlen = sizeof(serveraddr);
    main_loop( &serveraddr);

    return 0;
}
```