Table of Contents

Libevent

本教程要求有一定的服务并发编程基础，了解select和epoll等多路I/O复用机制。

教程目的主要是快速建立libevent的认知，了解libevent的常用数据结构和编程方法。

达到可以使用libevent写出自己的高并发服务器处理模型。

1 Libevent官方

- 官方网站：http://libevent.org/

libevent版本一共有1.4系列和2.0系列两个稳定版本。

> 1.4系列比较古老，但是源码简单，适合源码的学习
>
> 2.0系列比较新，建议直接使用2.0

需要注意的是，1.4系列和2.0系列两个版本的接口并不兼容，就是2.0将一些接口的原型发>生了改变，所以将1.4升级到2.0需要重新编码。

1.1 libevent 特点

- 事件驱动，高性能；
- 轻量级，专注于网络；
- 跨平台，支持 Windows、Linux、Mac Os等；
- 支持多种 I/O多路复用技术， epoll、poll、dev/poll、select 和kqueue 等；
- 支持 I/O，定时器和信号等事件；

1.2 libevent下载与安装

在官网上找到 `libevent-2.0.22-stable.tar.gz` 下载地址。

```
tar -zxvf libevent-2.0.22-stable.tar.gz

cd libevent-2.0.22-stable/

./configure

make

sudo make install
```

1.3 libevent开源包

在 `.libs` 隐藏文件中包含全部libevent已经编译好的so文件。

其中core为libevent的核心文件，libevent.so为主链接文件，会关联到其他全部so文件。

在sample目录下会有已经编译好的服务器应用程序。

可以拿 `hello-world` 程序用来测试。

服务端:

```
./hello-world
```

客户端:

```
netcat 192.168.2.105 9995
```

如果客户端收到“hello world”字符串，表示libevent在本机可以正常使用。

2 EPOLL

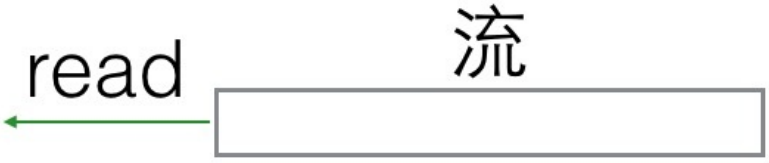
本章主要简述epoll的基础理论知识。方便理解libevent。

2.1 流 I\O操作 阻塞

2.1.1 流

- 可以进行I\O操作的内核对象
- 文件、管道、套接字……
- 流的入口：文件描述符(fd)

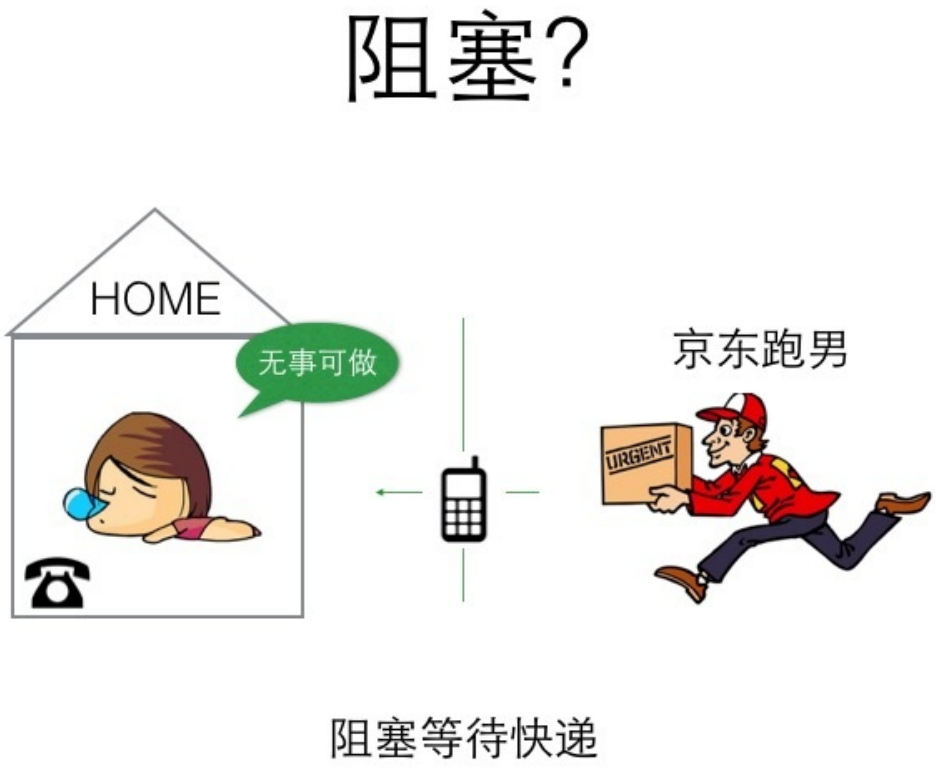
2.1.2 I\O操作



所有对流的读写操作，我们都可以称之为IO操作。

> 那么当一个流中再没有数据，**read**的时候，或者说在流中已经写满了数据，再**write**，我们的**IO**操作就会出现一种现象，就是阻塞现象。

2.1.2 阻塞



- 阻塞等待

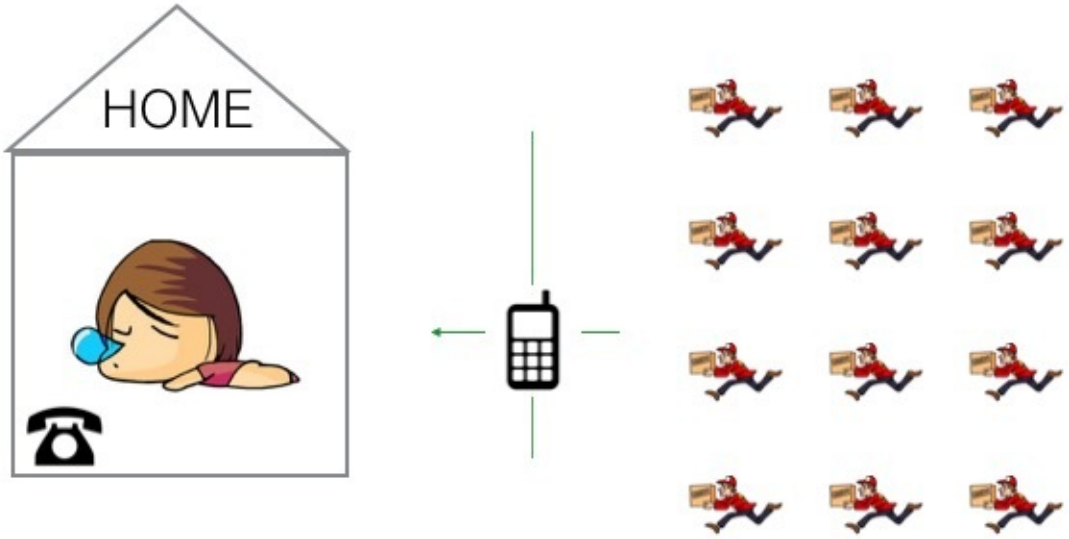
空出大脑可以安心睡觉。（不占用CPU宝贵的时间片）

- 非阻塞，忙轮询

浪费时间，浪费电话费，占用快递员时间（占用CPU，系统资源）

2.2 解决阻塞死等待的办法

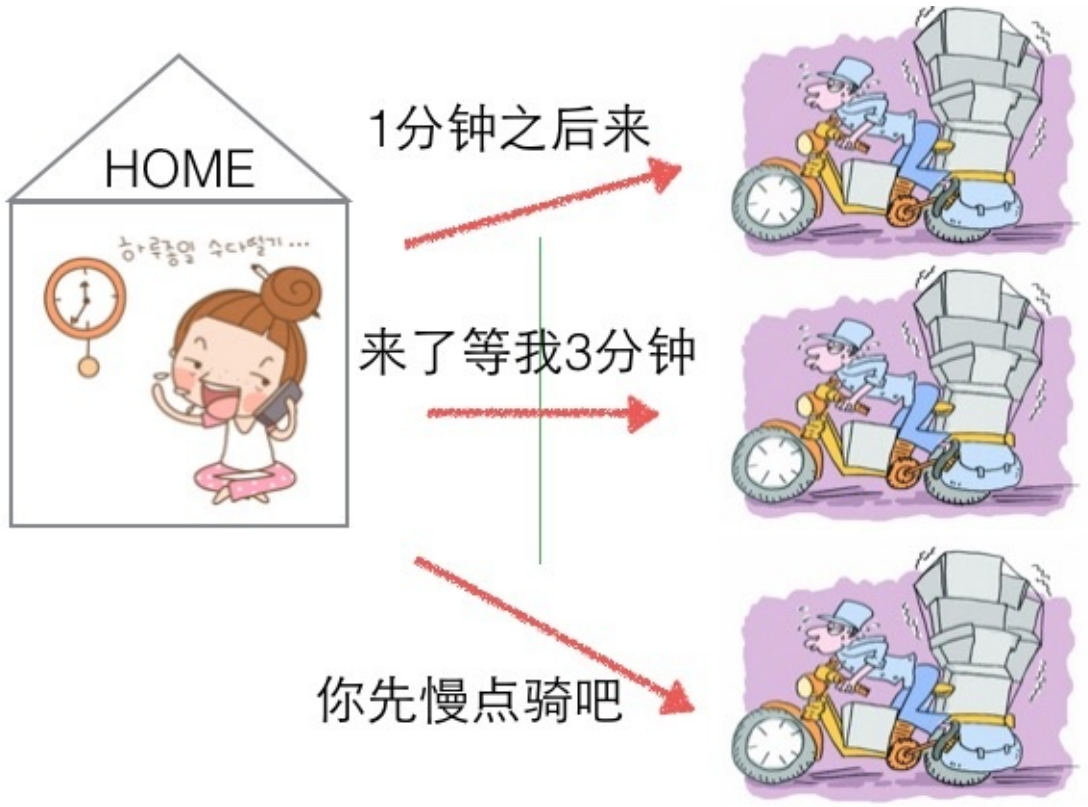
2.2.1 阻塞死等待的缺点



如果同一时刻到达，你同一时刻可能只签收并验货一份快递
你的电话是座机，在你签收的时候，便接不到其他快递员的电话。

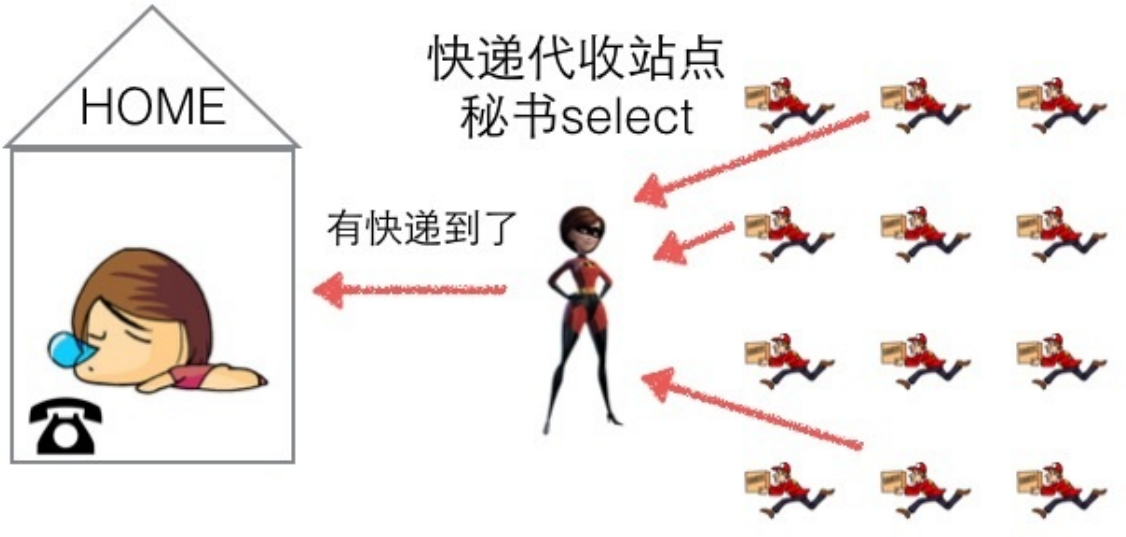
多线程 或 多进程

2.2.2 办法一：非阻塞、忙轮询




```
while true {
    for i in 流[] {
        if i has 数据 {
            读 或者 其他处理
        }
    }
}
```

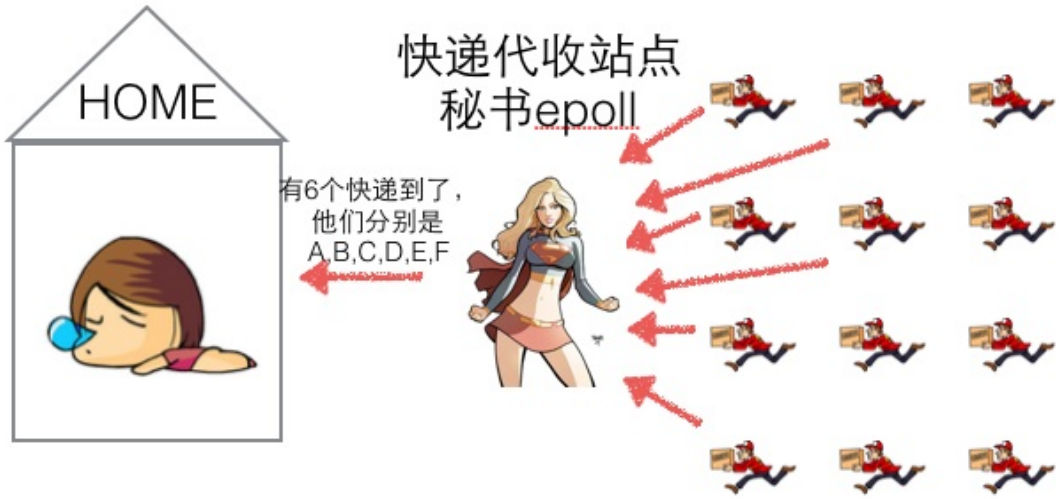
2.2.3 办法二：select



select 代收员 比较懒，她只会告诉你快递到了，但是是谁到的，你需要挨个快递员问一遍。

```
while true {
    select(流[]);  //阻塞

    for i in 流[] {
        if i has 数据 {
            读 或者 其他处理
        }
    }
}
```

2.2.3 办法三：epoll




```
while true {
    可处理的流[] = epoll_wait(epoll_fd);  //阻塞

    for i in 可处理的流[] {
        读 或者 其他处理
   }
}
```

2.3 什么是epoll

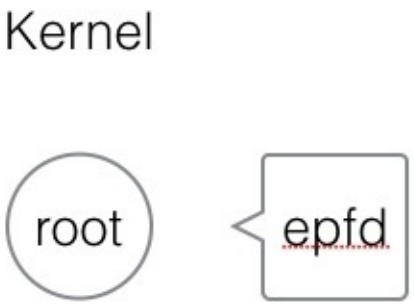
- 与**select**，**poll**一样，对**I\/O**多路复用的技术
- 只关心“活跃”的链接，无需遍历全部描述符集合
- 能够处理大量的链接请求(系统可以打开的文件数目)

2.4 epoll API

2.4.1 创建EPOLL

```
/**
 * @param size 告诉内核监听的数目
 *
 * @returns 返回一个epoll句柄（即一个文件描述符）
 */
int epoll_create(int size);
```

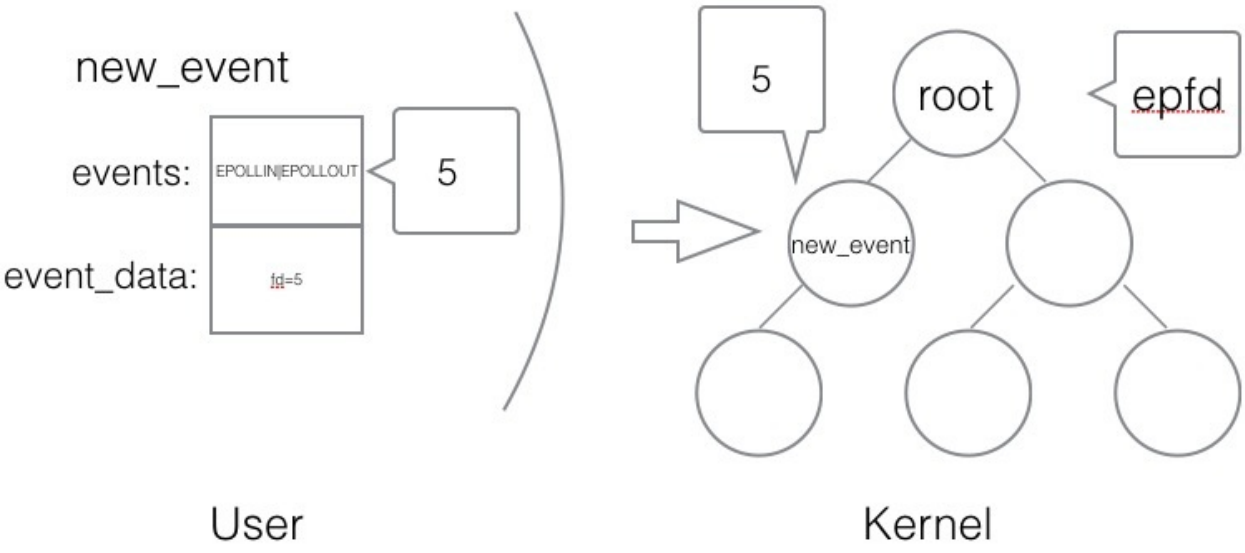
```
int epfd = epoll_create(1000);
```



2.4.2 控制EPOLL

```
/**
 * @param epfd 用epoll_create所创建的epoll句柄
 * @param op 表示对epoll监控描述符控制的动作
 *
 * EPOLL_CTL_ADD(注册新的fd到epfd)
 * EPOLL_CTL_MOD(修改已经注册的fd的监听事件)
 * EPOLL_CTL_DEL(epfd删除一个fd)
 *
 * @param fd 需要监听的文件描述符
 * @param event 告诉内核需要监听的事件
 *
 * @returns 成功返回0，失败返回-1, errno查看错误信息
 */
int epoll_ctl(int epfd, int op, int fd,
              struct epoll_event *event);


struct epoll_event {
 __uint32_t events; /* epoll 事件 */
 epoll_data_t data; /* 用户传递的数据 */
}

/*
 * events : {EPOLLIN, EPOLLOUT, EPOLLPRI,
             EPOLLHUP, EPOLLET, EPOLLONESHOT}
 */

typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;
```

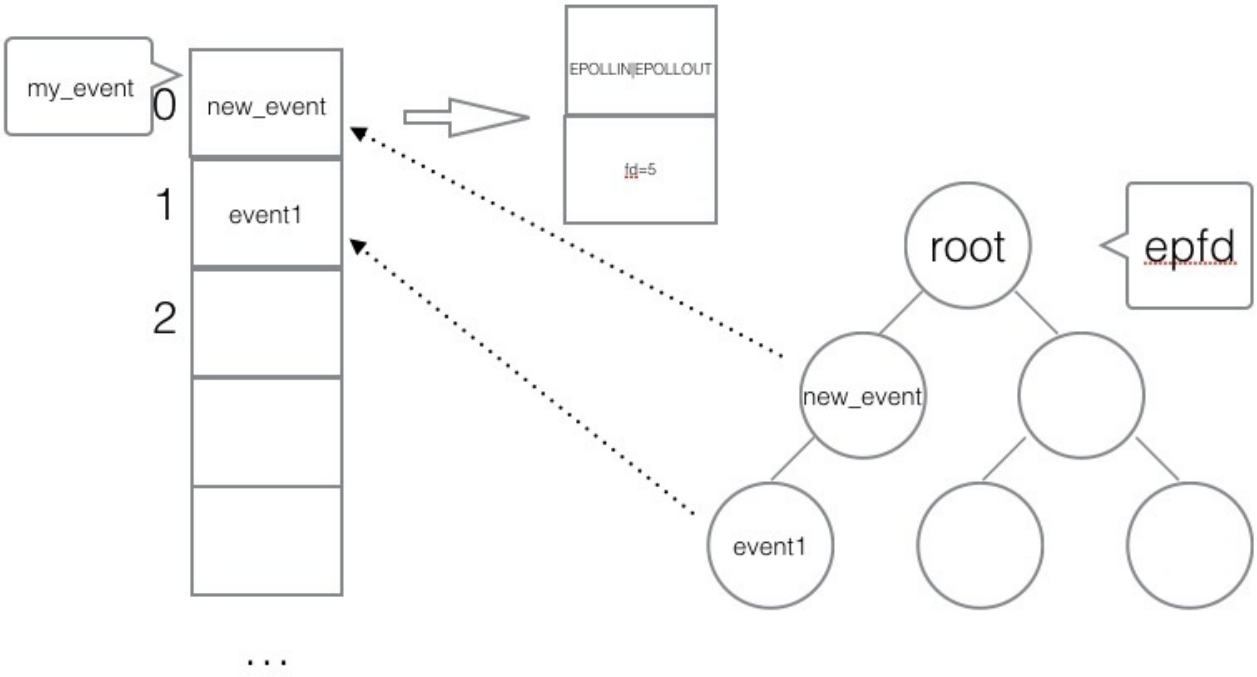
```
struct epoll_event new_event;

new_event.events = EPOLLIN | EPOLLOUT;
new_event.data.fd = 5;

epoll_ctl(epfd, EPOLL_CTL_ADD, 5, &new_event);
```



2.4.3 等待EPOLL

```
/**
 *
 * @param epfd 用epoll_create所创建的epoll句柄
 * @param event 从内核得到的事件集合
 * @param maxevents 告知内核这个events有多大,
 *              注意: 值 不能大于创建epoll_create()时的size.
 * @param timeout 超时时间
 *      -1: 永久阻塞
 *      0: 立即返回，非阻塞
 *      >0: 指定微秒
 *
 * @returns 成功: 有多少文件描述符就绪,时间到时返回0
 *          失败: -1, errno 查看错误
 */
int epoll_wait(int epfd, struct epoll_event *event,
            int maxevents, int timeout);
```

```
struct epoll_event my_event[1000];

int event_cnt = epoll_wait(epfd, my_event, 1000, -1);
```

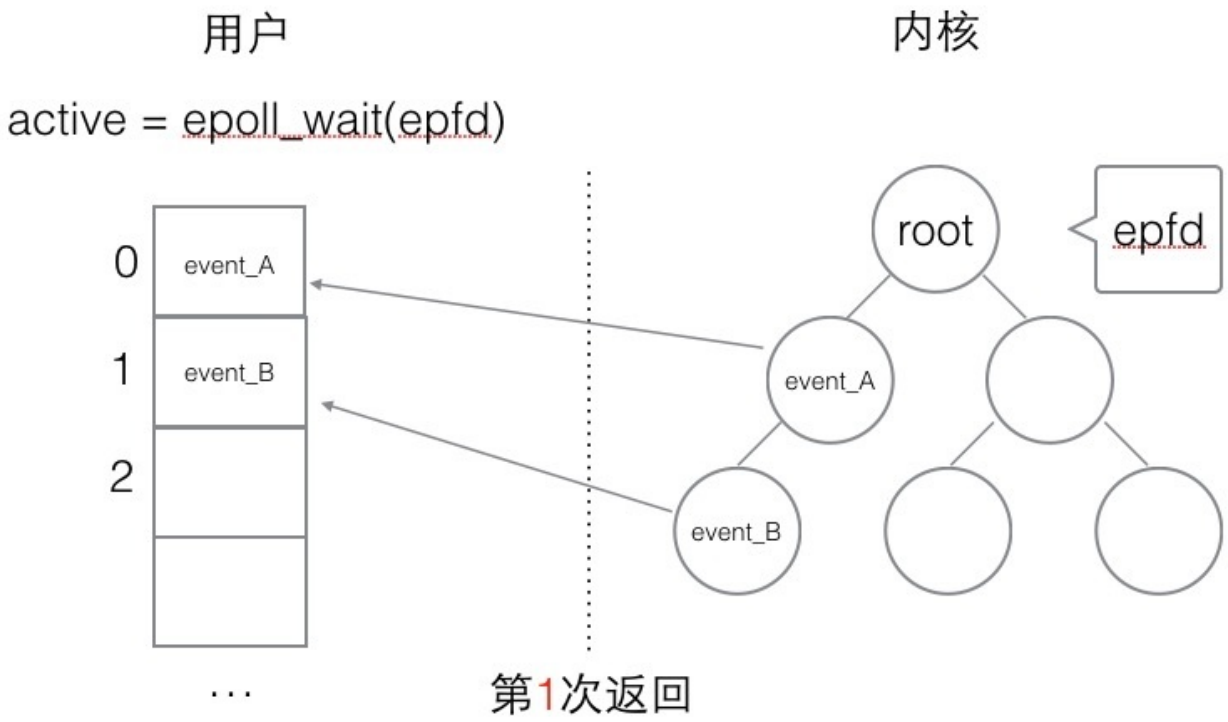


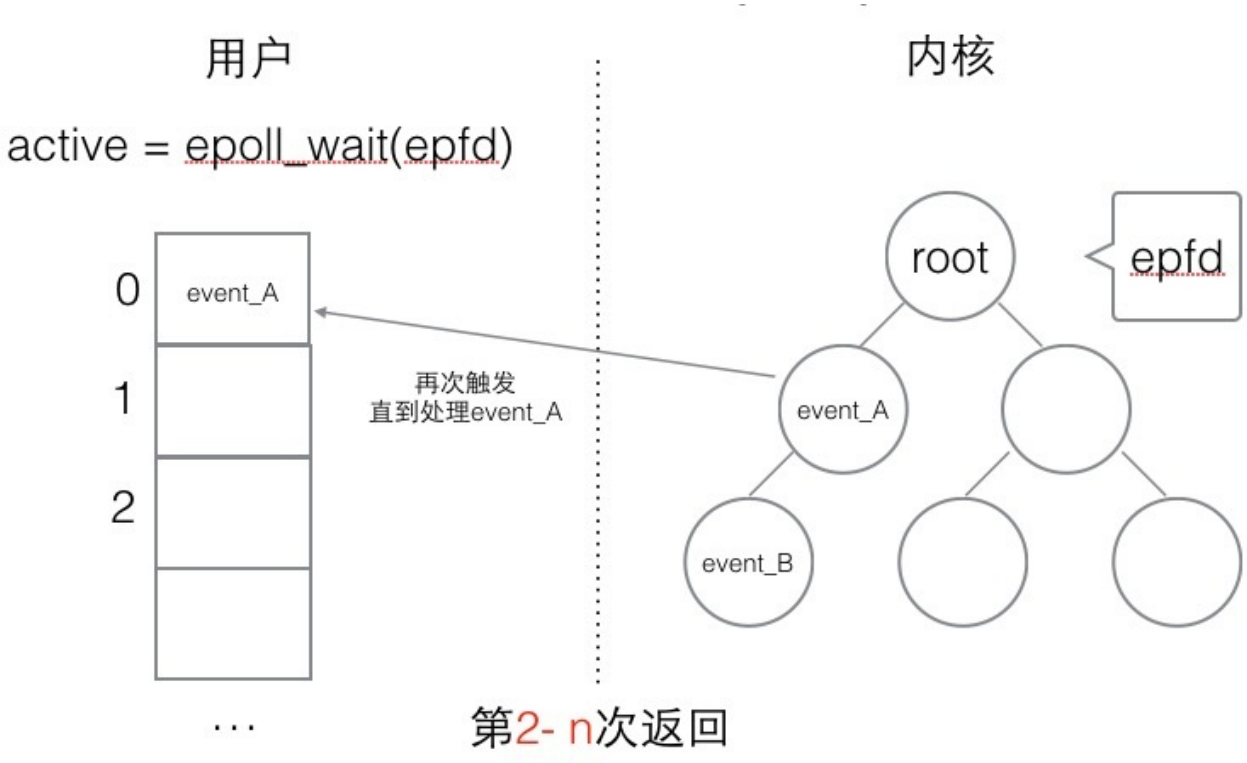
2.4.4 epoll编程框架

```
//创建 epoll
int epfd = epoll_crete(1000);

//将 listen_fd 添加进 epoll 中
epoll_ctl(epfd, EPOLL_CTL_ADD, listen_fd,&listen_event);

while (1) {
    //阻塞等待 epoll 中 的fd 触发
    int active_cnt = epoll_wait(epfd, events, 1000, -1);

    for (i = 0 ; i < active_cnt; i++) {
        if (evnets[i].data.fd == listen_fd) {
            //accept. 并且将新accept 的fd 加进epoll中.
        }
        else if (events[i].events & EPOLLIN) {
            //对此fd 进行读操作
        }
        else if (events[i].events & EPOLLOUT) {
            //对此fd 进行写操作
        }
    }
}
```

2.5 触发模式

2.5.1 水平触发

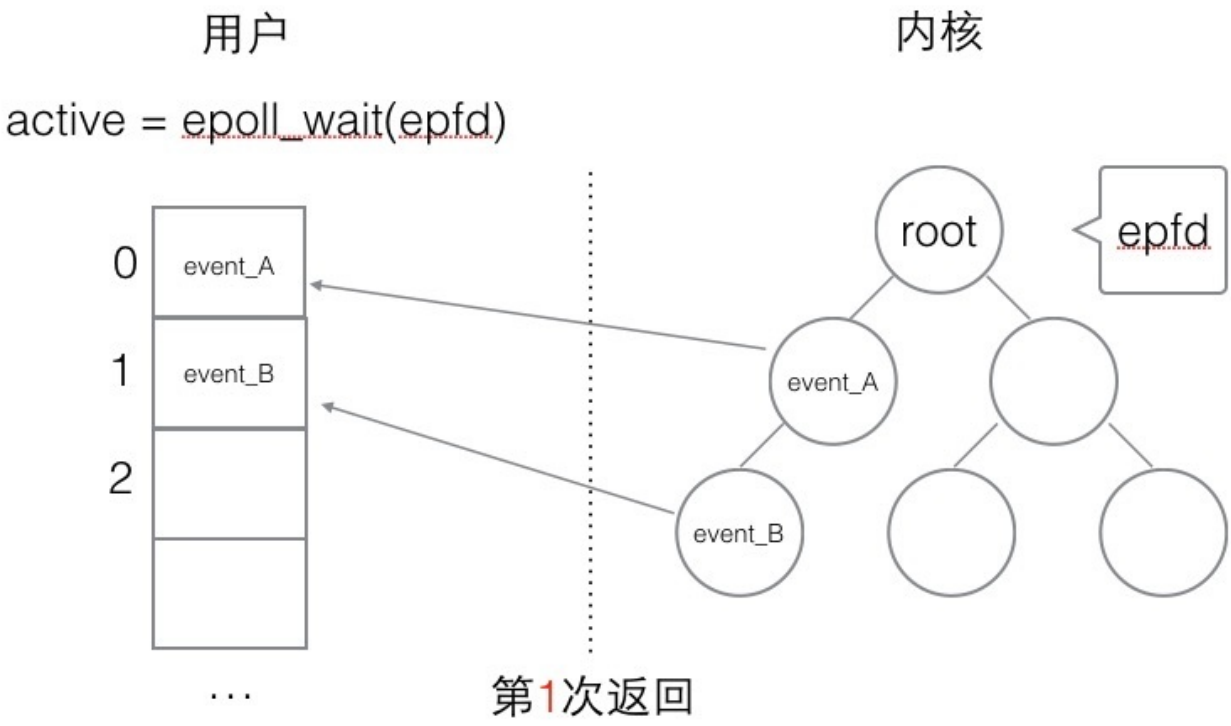


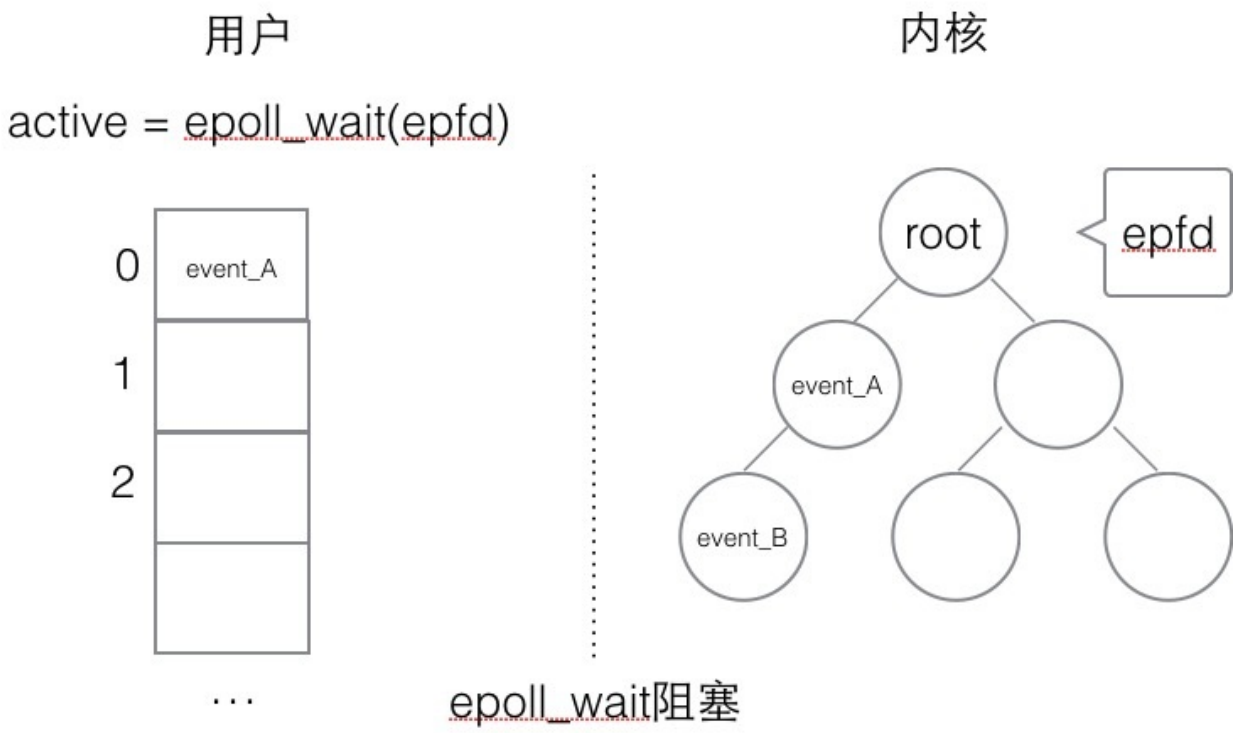


水平触发的主要特点是，如果用户在监听epoll事件，当内核有事件的时候，会拷贝给用户态事件，但是如果用户只处理了一次，那么剩下没有处理的会在下一次epoll_wait再次返回该事件。

这样如果用户永远不处理这个事件，就导致每次都会有该事件从内核到用户的拷贝，耗费性能，但是水平触发相对安全，最起码事件不会丢掉，除非用户处理完毕。

2.5.2 边缘触发





边缘触发，相对跟水平触发相反，当内核有事件到达， 只会通知用户一次，至于用户处理还是不处理，以后将不会再通知。这样减少了拷贝过程，增加了性能，但是相对来说，如果用户马虎忘记处理，将会产生事件丢的情况。

2.6 epoll服务器

2.6.1 服务端

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#include <sys/epoll.h>


#define SERVER_PORT         (7778)
#define EPOLL_MAX_NUM       (2048)
#define BUFFER_MAX_LEN      (4096)

char buffer[BUFFER_MAX_LEN];

void str_toupper(char *str)
{
    int i;
    for (i = 0; i < strlen(str); i ++) {
        str[i] = toupper(str[i]);
    }
}

int main(int argc, char **argv)
{
    int listen_fd = 0;
    int client_fd = 0;
    struct sockaddr_in server_addr;
    struct sockaddr_in client_addr;
```

```
    socklen_t           client_len;

    int epfd = 0;
    struct epoll_event event, *my_events;

    // socket
    listen_fd = socket(AF_INET, SOCK_STREAM, 0);

    // bind
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(SERVER_PORT);
    bind(listen_fd, (struct sockaddr*)&server_addr, sizeof(serve
r_addr));

    // listen
    listen(listen_fd, 10);

    // epoll create
    epfd = epoll_create(EPOLL_MAX_NUM);
    if (epfd < 0) {
        perror("epoll create");
        goto END;
    }

    // listen_fd -> epoll
    event.events = EPOLLIN;
    event.data.fd = listen_fd;
    if (epoll_ctl(epfd, EPOLL_CTL_ADD, listen_fd, &event) < 0) {
        perror("epoll ctl add listen_fd ");
        goto END;
    }

    my_events = malloc(sizeof(struct epoll_event) * EPOLL_MAX_NU
M);


    while (1) {
        // epoll wait
        int active_fds_cnt = epoll_wait(epfd, my_events, EPOLL_M
```

```
AX_NUM, -1);
        int i = 0;
        for (i = 0; i < active_fds_cnt; i++) {
            // if fd == listen_fd
            if (my_events[i].data.fd == listen_fd) {
                //accept
                client_fd = accept(listen_fd, (struct sockaddr*)
&client_addr, &client_len);
                if (client_fd < 0) {
                    perror("accept");
                    continue;
                }

                char ip[20];
                printf("new connection[%s:%d]\n", inet_ntop(AF_I
NET, &client_addr.sin_addr, ip, sizeof(ip)), ntohs(client_addr.s
in_port));

                event.events = EPOLLIN | EPOLLET;
                event.data.fd = client_fd;
                epoll_ctl(epfd, EPOLL_CTL_ADD, client_fd, &event
);
            }
            else if (my_events[i].events & EPOLLIN) {
                printf("EPOLLIN\n");
                client_fd = my_events[i].data.fd;

                // do read

                buffer[0] = '\0';
                int n = read(client_fd, buffer, 5);
                if (n < 0) {
                    perror("read");
                    continue;
                }
                else if (n == 0) {
                    epoll_ctl(epfd, EPOLL_CTL_DEL, client_fd, &e
vent);

                    close(client_fd);
                }
```

```
                else {
                    printf("[read]: %s\n", buffer);
                    buffer[n] = '\0';
#if 1
                    str_toupper(buffer);
                    write(client_fd, buffer, strlen(buffer));
                    printf("[write]: %s\n", buffer);
                    memset(buffer, 0, BUFFER_MAX_LEN);
#endif

/*
                    event.events = EPOLLOUT;
                    event.data.fd = client_fd;
                    epoll_ctl(epfd, EPOLL_CTL_MOD, client_fd, &e
vent);
*/
                }
            }
            else if (my_events[i].events & EPOLLOUT) {
                printf("EPOLLOUT\n");
/*
                client_fd = my_events[i].data.fd;
                str_toupper(buffer);
                write(client_fd, buffer, strlen(buffer));
                printf("[write]: %s\n", buffer);
                memset(buffer, 0, BUFFER_MAX_LEN);

                event.events = EPOLLIN;
                event.data.fd = client_fd;
                epoll_ctl(epfd, EPOLL_CTL_MOD, client_fd, &event
);
*/
            }
        }
    }



END:
    close(epfd);
```

```
    close(listen_fd);
    return 0;
}
```

2.6.2 客户端

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>

#define MAX_LINE (1024)
#define SERVER_PORT (7778)

void setnoblocking(int fd)
{
    int opts = 0;
    opts = fcntl(fd, F_GETFL);
    opts = opts | O_NONBLOCK;
    fcntl(fd, F_SETFL);
}

int main(int argc, char **argv)
{
    int  sockfd;
    char recvline[MAX_LINE + 1] = {0};

    struct sockaddr_in server_addr;

    if (argc != 2) {
        fprintf(stderr, "usage ./client <SERVER_IP>\n");
        exit(0);
```

```
    }


    // 创建socket
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "socket error");
        exit(0);
    }


    // server addr 赋值
    bzero(&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);

    if (inet_pton(AF_INET, argv[1], &server_addr.sin_addr) <= 0)
 {
        fprintf(stderr, "inet_pton error for %s", argv[1]);
        exit(0);
    }


    // 链接服务端
    if (connect(sockfd, (struct sockaddr*) &server_addr, sizeof(
server_addr)) < 0) {
        perror("connect");
        fprintf(stderr, "connect error\n");
        exit(0);
    }

    setnoblocking(sockfd);

    char input[100];
    int n = 0;
    int count = 0;



    // 不断的从标准输入字符串
    while (fgets(input, 100, stdin) != NULL)
```

```
    {
        printf("[send] %s\n", input);
        n = 0;
        // 把输入的字符串发送 到 服务器中去
        n = send(sockfd, input, strlen(input), 0);
        if (n < 0) {
            perror("send");
        }

        n = 0;
        count = 0;


        // 读取 服务器返回的数据
        while (1)
        {
            n = read(sockfd, recvline + count, MAX_LINE);
            if (n == MAX_LINE)
            {
                count += n;
                continue;
            }
            else if (n < 0){
                perror("recv");
                break;
            }
            else {
                count += n;
                recvline[count] = '\0';
                printf("[recv] %s\n", recvline);
                break;
            }
        }
    }

    return 0;
}
```

3.1 reactor反应堆模式

对每一个构架模式的分析，我们都使用参考文献的分析风格，着重分析意图、上下文、问题、解决方案、结构和实现6个方面的内容。

1. 意图

在事件驱动的应用中，将一个或多个客户的服务请求分离（demultiplex）和调度（dispatch）给应用程序。

2. 上下文

在事件驱动的应用中，同步地、有序地处理同时接收的多个服务请求。

3. 问题

在分布式系统尤其是服务器这一类事件驱动应用中，虽然这些请求最终会被序列化地处理，但是必须时刻准备着处理多个同时到来的服务请求。在实际应用 中，这些请求总是通过一个事件（如CONNECTOR、READ、WRITE等）来表示的。在有序地处理这些服务请求之前，应用程序必须先分离和调度这些 同时到达的事件。为了有效地解决这个问题，我们需要做到以下4方面：

为了提高系统的可测量性和反应时间，应用程序不能长时间阻塞在某个事件源上而停止对其他事件的处理，这样会严重降低对客户端的响应度。 为了提高吞吐量，任何没有必要的上下文切换、同步和CPU之间的数据移动都要避免。 引进新的服务或改良已有的服务都要对既有的事件分离和调度机制带来尽可能小的影响。 大量的应用程序代码需要隐藏在复杂的多线程和同步机制之后。

4. 解决方案

在一个或多个事件源上等待事件的到来，例如，一个已经连接的Socket描述符就是一个事件源。将事件的分离和调度整合到处理它的服务中，而将分离和调度机制从应用程序对特定事件的处理中分离开，也就是说分离和调度机制与特定的应用程序

无关。

具体来说，每个应用程序提供的每个服务都有一个独立的事件处理器与之对应。由事件处理器处理来自事件源的特定类型的事件。每个事件处理器都事先注册 到Reactor管理器中。Reactor管理器使用同步事件分离器在一个或多个事件源中等待事件的发生。当事件发生后，同步事件分离器通知 Reactor管理器，最后由Reactor管理器调度和该事件相关的事件处理器来完成请求的服务。

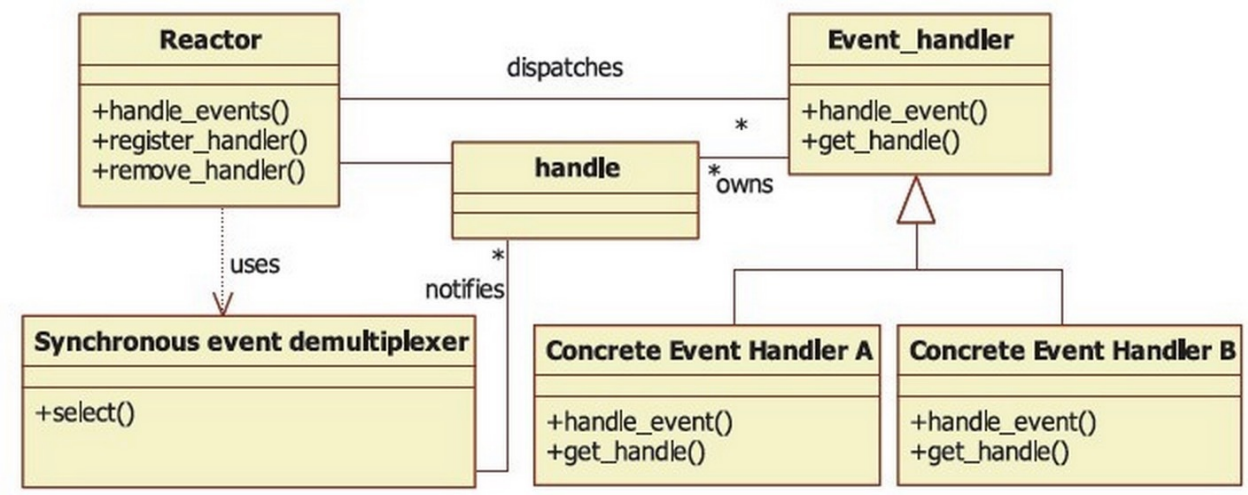
5. 结构

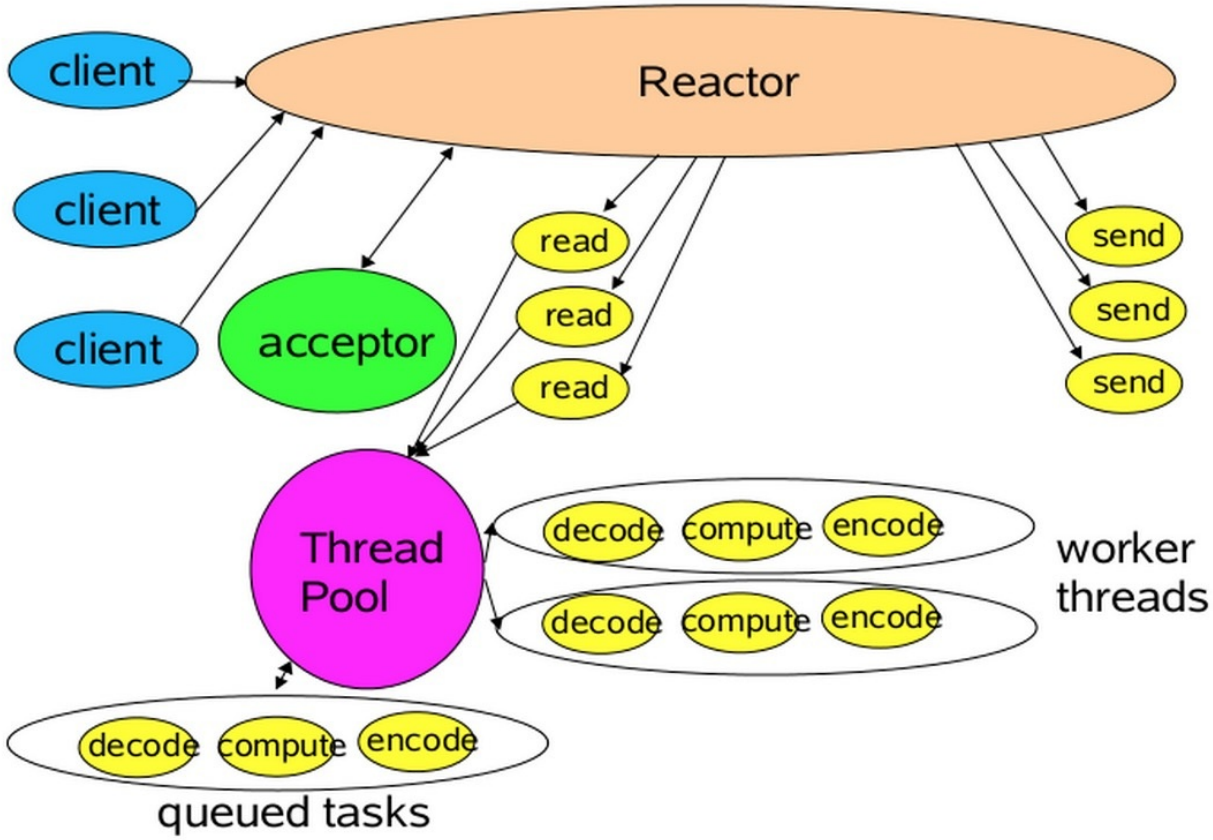
在Reactor模式中，有5个关键的参与者。

描述符（handle）：由操作系统提供，用于识别每一个事件，如Socket描述符、文件描述符等。在Linux中，它用一个整数来表示。事件可以来自外部，如来自客户端的连接请求、数据等。事件也可以来自内部，如定时器事件。 同步事件分离器（demultiplexer）：是一个函数，用来等待一个或多个事件的发生。调用者会被阻塞，直到分离器分离的描述符集上有事件发生。Linux的select函数是一个经常被使用的分离器。 事件处理器接口（event handler）：是由一个或多个模板函数组成的接口。这些模板函数描述了和应用程序相关的对某个事件的操作。 具体的事件处理器：是事件处理器接口的实现。它实现了应用程序提供的某个服务。每个具体的事件处理器总和一个描述符相关。它使用描述符来识别事件、识别应用程序提供的服务。 Reactor 管理器（reactor）：定义了一些接口，用于应用程序控制事件调度，以及应用程序注册、删除事件处理器和相关的描述符。它是事件处理器的调度核心。 Reactor管理器使用同步事件分离器来等待事件的发生。一旦事件发生，Reactor管理器先是分离每个事件，然后调度事件处理器，最后调用相关的模 板函数来处理这个事件。 通过上述分析，我们注意到，是Reactor管理器而不是应用程序负责等待事件、分离事件和调度事件。实际上，Reactor管理器并没有被具体的事件处理器调用，而是管理器调度具体的事件处理器，由事件处理器对发生的事件做出处理。这就是类似Hollywood原则的“反向控制”。应用程序要做的 仅仅是实现一个具体的事件处理器，然后把它注册到Reactor管理器中。接下来的工作由管理器来完成。这些参与者的相互关系如图2-1所示。

现在结合第1章分析的框架五元素来看一下Reactor构架模式的参与者与框架五元素之间的关系：Reactor构架模式的具体实现对应了元素1； 事件处理器接口对应元素2；具体的事件处理器对应元素3；Reactor管理器使用了Hollywood原则，可以认为和元素5对应；元素4的功能相对不 明显，没有明确的对应关系。

如果还是没有理解Reactor构架模式，没有关系，源代码会说明所有问题。此时可再分析一遍Reactor构架模式，然后继续以下内容。





3.2 epoll的反应堆模式实现

> epoll反应堆模式的实现-也就是libevent的实现原理。

```
#include <stdlib.h>
#include <stdio.h>
#include <stdio.h>
#include <sys/socket.h>
#include <sys/epoll.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <time.h>
#define MAX_EVENTS  1024
#define BUFLEN 128
#define SERV_PORT   8080


/*
 * status:1表示在监听事件中，0表示不在
 * last_active:记录最后一次响应时间,做超时处理
 */
struct myevent_s {
    int fd;                 //cfd listenfd
    int events;             //EPOLLIN  EPLLOUT
    void *arg;              //指向自己结构体指针
    void (*call_back)(int fd, int events, void *arg);
    int status;
    char buf[BUFLEN];
    int len;
    long last_active;
};

int g_efd;          /* epoll_create返回的句柄 */
struct myevent_s g_events[MAX_EVENTS+1];   /* +1 最后一个用于 list
en fd */
```

```
void eventset(struct myevent_s *ev, int fd, void (*call_back)(int
, int, void *), void *arg)
{
    ev->fd = fd;
    ev->call_back = call_back;
    ev->events = 0;
    ev->arg = arg;
    ev->status = 0;
    //memset(ev->buf, 0, sizeof(ev->buf));
    //ev->len = 0;
    ev->last_active = time(NULL);

    return;
}

void recvdata(int fd, int events, void *arg);
void senddata(int fd, int events, void *arg);

void eventadd(int efd, int events, struct myevent_s *ev)
{
    struct epoll_event epv = {0, {0}};
    int op;
    epv.data.ptr = ev;
    epv.events = ev->events = events;

    if (ev->status == 1) {
        op = EPOLL_CTL_MOD;
    }
    else {
        op = EPOLL_CTL_ADD;
        ev->status = 1;
    }

    if (epoll_ctl(efd, op, ev->fd, &epv) < 0)
        printf("event add failed [fd=%d], events[%d]\n", ev->fd,
 events);
    else
        printf("event add OK [fd=%d], op=%d, events[%0X]\n", ev-
>fd, op, events);
```

```
    return;
}

void eventdel(int efd, struct myevent_s *ev)
{
    struct epoll_event epv = {0, {0}};

    if (ev->status != 1)
        return;

    epv.data.ptr = ev;
    ev->status = 0;
    epoll_ctl(efd, EPOLL_CTL_DEL, ev->fd, &epv);

    return;
}


void acceptconn(int lfd, int events, void *arg)
{
    struct sockaddr_in cin;
    socklen_t len = sizeof(cin);
    int cfd, i;

    if ((cfd = accept(lfd, (struct sockaddr *)&cin, &len)) == -1
) {
        if (errno != EAGAIN && errno != EINTR) {
            /* 暂时不做出错处理 */
        }
        printf("%s: accept, %s\n", __func__, strerror(errno));
        return;
    }

    do {
        for (i = 0; i < MAX_EVENTS; i++) {
            if (g_events[i].status == 0)
                break;
        }
```

```
        if (i == MAX_EVENTS) {
            printf("%s: max connect limit[%d]\n", __func__, MAX_
EVENTS);
            break;
        }

        int flag = 0;
        if ((flag = fcntl(cfd, F_SETFL, O_NONBLOCK)) < 0)
        {
            printf("%s: fcntl nonblocking failed, %s\n", __func_
_, strerror(errno));
            break;
        }

        eventset(&g_events[i], cfd, recvdata, &g_events[i]);
        eventadd(g_efd, EPOLLIN, &g_events[i]);
    } while(0);

    printf("new connect [%s:%d][time:%ld], pos[%d]\n", inet_ntoa
(cin.sin_addr), ntohs(cin.sin_port), g_events[i].last_active, i)
;

    return;
}

void recvdata(int fd, int events, void *arg)
{
    struct myevent_s *ev = (struct myevent_s *)arg;
    int len;

    len = recv(fd, ev->buf, sizeof(ev->buf), 0);
    eventdel(g_efd, ev);

    if (len > 0) {
        ev->len = len;
        ev->buf[len] = '\0';
        printf("C[%d]:%s\n", fd, ev->buf);
        /* 转换为发送事件 */
        eventset(ev, fd, senddata, ev);
        eventadd(g_efd, EPOLLOUT, ev);
```

```
    }
    else if (len == 0) {
        close(ev->fd);
        /* ev-g_events 地址相减得到偏移元素位置 */
        printf("[fd=%d] pos[%d], closed\n", fd, (int)(ev - g_eve
nts));
    }
    else {
        close(ev->fd);
        printf("recv[fd=%d] error[%d]:%s\n", fd, errno, strerror
(errno));
    }

    return;
}

void senddata(int fd, int events, void *arg)
{
    struct myevent_s *ev = (struct myevent_s *)arg;
    int len;

    len = send(fd, ev->buf, ev->len, 0);
    //printf("fd=%d\tev->buf=%s\ttev->len=%d\n", fd, ev->buf, ev
->len);
    //printf("send len = %d\n", len);

    eventdel(g_efd, ev);
    if (len > 0) {
        printf("send[fd=%d], [%d]%s\n", fd, len, ev->buf);
        eventset(ev, fd, recvdata, ev);
        eventadd(g_efd, EPOLLIN, ev);
    }
    else {
        close(ev->fd);
        printf("send[fd=%d] error %s\n", fd, strerror(errno));
    }

    return;
}
```

```
void initlistensocket(int efd, short port)
{
    int lfd = socket(AF_INET, SOCK_STREAM, 0);
    fcntl(lfd, F_SETFL, O_NONBLOCK);
    eventset(&g_events[MAX_EVENTS], lfd, acceptconn, &g_events[M
AX_EVENTS]);
    eventadd(efd, EPOLLIN, &g_events[MAX_EVENTS]);

    struct sockaddr_in sin;

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons(port);

    bind(lfd, (struct sockaddr *)&sin, sizeof(sin));

    listen(lfd, 20);

    return;
}

int main(int argc, char *argv[])
{
    unsigned short port = SERV_PORT;

    if (argc == 2)
        port = atoi(argv[1]);

    g_efd = epoll_create(MAX_EVENTS+1);

    if (g_efd <= 0)
        printf("create efd in %s err %s\n", __func__, strerror(e
rrno));

    initlistensocket(g_efd, port);

    /* 事件循环 */
    struct epoll_event events[MAX_EVENTS+1];
```

```
    printf("server running:port[%d]\n", port);
    int checkpos = 0, i;
    while (1) {
        /* 超时验证，每次测试100个链接，不测试listenfd 当客户端60秒内没有和服务器通信，则关闭此客户端链接 */
        long now = time(NULL);
        for (i = 0; i < 100; i++, checkpos++) {
            if (checkpos == MAX_EVENTS)
                checkpos = 0;
            if (g_events[checkpos].status != 1)
                continue;
            long duration = now - g_events[checkpos].last_active;
            if (duration >= 60) {
                close(g_events[checkpos].fd);
                printf("[fd=%d] timeout\n", g_events[checkpos].fd);
                eventdel(g_efd, &g_events[checkpos]);
            }
        }
        /* 等待事件发生 */
        int nfd = epoll_wait(g_efd, events, MAX_EVENTS+1, 1000);
        if (nfd < 0) {
            printf("epoll_wait error, exit\n");
            break;
        }
        for (i = 0; i < nfd; i++) {
            struct myevent_s *ev = (struct myevent_s *)events[i].data.ptr;
            if ((events[i].events & EPOLLIN) && (ev->events & EPOLLIN)) {
                ev->call_back(ev->fd, events[i].events, ev->arg);
            }
            if ((events[i].events & EPOLLOUT) && (ev->events & EPOLLOUT)) {
                ev->call_back(ev->fd, events[i].events, ev->arg);
            }
        }
```

```
    }

    /* 退出前释放所有资源 */
    return 0;
}
```

4 event_base

> 本章主要来源《libevent参考手册（中文版）》。

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。

如果设置 event_base 使用锁,则可以安全地在多个线程中访问它 。然而,其事件循环只能 运行在一个线程中。如果需要用多个线程检测 IO,则需要为每个线程使用一个 event_base。

每个 event_base 都有一种用于检测哪种事件已经就绪的“方法”,或者说后端。可以识别的方法有:

- **select**
- poll
- **epoll**
- kqueue
- devpoll
- evport
- win32

4.1 创建event_base

4.1.1 创建默认的event_base

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。

```
struct event_base *event_base_new(void);
```

> 大多数程序使用这个函数就够了。

event_base_new()函数声明在中,首次出现在 libevent 1.4.3版。

4.1.2 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 ()**。

创建接口

```
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。

```
int event_config_avoid_method(struct event_config *cfg, const ch
ar *method);

enum event_method_feature {
    EV_FEATURE_ET = 0x01,
    EV_FEATURE_O1 = 0x02,
    EV_FEATURE_FDS = 0x04,
};
int event_config_require_features(struct event_config *cfg,
                                  enum event_method_feature feat
ure);

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

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 。 调用 event_config_require_feature ()让 libevent 不使用不能提供所有指定特征的后端。 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。

event_config_require_features()可识别的特征值有:

- EV_FEATURE_ET:要求支持边沿触发的后端
- EV_FEATURE_O1: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:要求支持任意文件描述符,而不仅仅是套接字的后端

event_config_set_flag()可识别的选项值有:

- 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。

上述操作 event_config 的函数都在成功时返回0,失败时返回-1。

> 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。

4.2 检查event_base后端

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。

接口1

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

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。

实例：

```
int i;
const char **methods = event_get_supported_methods();
printf("Starting Libevent %s.  Available methods are:\n",
    event_get_version());
for (i=0; methods[i] != NULL; ++i) {
    printf("    %s\n", methods[i]);
}
```

> 这个函数返回 libevent 被编译以支持的方法列表 。然而 libevent 运行的时候,操作系统可能 不能支持所有方法。比如说,可能 OS X 版本中的 kqueue 的 bug 太多,无法使用。

接口2

```
const char *
event_base_get_method(const struct event_base *base);

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

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。

实例

```
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.",
        event_base_get_method(base));
    f = event_base_get_features(base);
    if ((f & EV_FEATURE_ET))
        printf("  Edge-triggered events are supported.");
    if ((f & EV_FEATURE_O1))
        printf("  O(1) event notification is supported.");
    if ((f & EV_FEATURE_FDS))
        printf("  All FD types are supported.");
    puts("");
}
```

4.3 释放event_base

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。

event_base_free()定义在中,首次由 libevent 1.2实现。

4.4 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_prio
rities);
```

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。

> 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。

4.5 event_base和fork

不是所有事件后端都在调用 fork()之后可以正确工作。所以,如果在使用 fork()或者其 他相关系统调用启动新进程之后,希望在新进程中继续使用 event_base,就需要进行重新初始化。

```
int event_reinit(struct event_base *base);
```

成功时这个函数返回 0,失败时返回 -1。

实例

```
struct event_base *base = event_base_new();

/* ... add some events to the event_base ... */

if (fork()) {
    /* In parent */
    continue_running_parent(base); /*...*/
} else {
    /* In child */
    event_reinit(base);
    continue_running_child(base); /*...*/
}
```

event_reinit()定义在中,在 libevent 1.4.3-alpha 版中首次可用。

5 事件循环event_loop

5.1 运行循环

一旦有了一个已经注册了某些事件的 event_base(关于如何创建和注册事件请看下一节), 就需要让 libevent 等待事件并且通知事件的发生。

```
#define EVLOOP_ONCE             0x01
#define EVLOOP_NONBLOCK         0x02
#define EVLOOP_NO_EXIT_ON_EMPTY 0x04

int event_base_loop(struct event_base *base, int flags);
```

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,然会返回。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。

为帮助理解,这里给出 event_base_loop()的算法概要:

```
while (any events are registered with the loop,
        or EVLOOP_NO_EXIT_ON_EMPTY was set) {

    if (EVLOOP_NONBLOCK was set, or any events are already activ
e)
        If any registered events have triggered, mark them activ
e.
    else
        Wait until at least one event has triggered, and mark it
 active.

    for (p = 0; p < n_priorities; ++p) {
       if (any event with priority of p is active) {
          Run all active events with priority of p.
          break; /* Do not run any events of a less important pr
iority */
       }
    }

    if (EVLOOP_ONCE was set or EVLOOP_NONBLOCK was set)
       break;
}
```

为方便起见,也可以调用

```
int event_base_dispatch(struct event_base *base);
```

event_base_dispatch ()等同于没有设置标志的 event_base_loop ( )。所以, event_base_dispatch ()将一直运行,直到没有已经注册的事件了,或者调用 了 event_base_loopbreak()或者 event_base_loopexit()为止。

这些函数定义在中,从 libevent 1.0版就存在了。

5.2 停止循环

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。

实例：

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

/* Here's a callback function that calls loopbreak */
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;

    /* Construct a new event to trigger whenever there are any b
ytes to
       read from a watchdog socket.  When that happens, we'll ca
ll the
       cb function, which will make the loop exit immediately wi
thout
       running any other active events at all.
     */
    watchdog_event = event_new(base, watchdog_fd, EV_READ, cb, b
ase);

    event_add(watchdog_event, NULL);

    event_base_dispatch(base);
}
```

实例2：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;

  /* Now we run the event_base for a series of 10-second interva
ls, printing
     "Tick" after each.  For a much better way to implement a 10
-second
     timer, see the section below about persistent timer events.
 */
  while (1) {
     /* This schedules an exit ten seconds from now. */
     event_base_loopexit(base, &ten_sec);

     event_base_dispatch(base);
     puts("Tick");
  }
}
```

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。

这些函数声明在中。

event_break_loopexit()函数首次在 libevent 1.0c 版本 中实现;
event_break_loopbreak()首次在 libevent 1.4.3版本中实现。

5.3 转储event_base的状态

为帮助调试程序(或者调试 libevent),有时候可能需要加入到 event_base 的事件及其状态 的完整列表。调用 event_base_dump_events()可以将这个列表输出到指定的文件中。

```
void event_base_dump_events(struct event_base *base, FILE *f);
```

这个列表是人可读的,未来版本的 libevent 将会改变其格式。

6 事件event

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:

- 文件描述符已经就绪,可以读取或者写入
- 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)
- 超时事件
- 发生某信号
- 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。

6.1 创建事件

6.1.1 生成新事件

使用 event_new()接口创建事件。

```
#define EV_TIMEOUT      0x01
#define EV_READ         0x02
#define EV_WRITE        0x04
#define EV_SIGNAL       0x08
#define EV_PERSIST      0x10
#define EV_ET           0x20

typedef void (*event_callback_fn)(evutil_socket_t, short, void *)
;

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

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

event_new()试图分配和构造一个用于 base 的新的事件。 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。

> 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。

要释放事件,调用 event_free()。对未决或者激活状态的事件调用 event_free()是安全 的:在释放事件之前,函数将会使事件成为非激活和非未决的。

实例：

```
#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%s [%s]",
            (int) fd,
            (what&EV_TIMEOUT) ? " timeout" : "",
            (what&EV_READ)    ? " read" : "",
            (what&EV_WRITE)   ? " write" : "",
            (what&EV_SIGNAL)  ? " signal" : "",
            data);
}

void main_loop(evutil_socket_t fd1, evutil_socket_t fd2)
{
        struct event *ev1, *ev2;
        struct timeval five_seconds = {5,0};
        struct event_base *base = event_base_new();

        /* The caller has already set up fd1, fd2 somehow, and m
ake them
           nonblocking. */

        ev1 = event_new(base, fd1, EV_TIMEOUT|EV_READ|EV_PERSIST
, cb_func,
           (char*)"Reading event");
        ev2 = event_new(base, fd2, EV_WRITE|EV_PERSIST, cb_func,
           (char*)"Writing event");

        event_add(ev1, &five_seconds);
        event_add(ev2, NULL);
        event_base_dispatch(base);
}
```

上述函数定义在 中,首次出现在 libevent 2.0.1-alpha 版本中。 event_callback_fn 类型首次在2.0.4-alpha 版本中作为 typedef 出现。

6.1.2 事件标志

- EV_TIMEOUT

这个标志表示某超时时间流逝后事件成为激活的。构造事件的时候,EV_TIMEOUT 标志是 被忽略的:可以在添加事件的时候设置超时 ,也可以不设置。超时发生时,回调函数的 what 参数将带有这个标志。

- EV_READ

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。

- EV_WRITE

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。

- EV_SIGNAL 用于实现信号检测,请看下面的 “构造信号事件”节。
- EV_PERSIST 表示事件是“持久的”,请看下面的“关于事件持久性”节。
- EV_ET

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。

从2.0.1-alpha 版本开始,可以有任意多个事件因为同样的条件而未决。比如说,可以有两 个事件因为某个给定的 fd 已经就绪,可以读取而成为激活的。这种情况下,多个事件回调 被执行的次序是不确定的。

> 这些标志定义在中。除了 EV_ET 在2.0.1-alpha 版本中引入外,所有标志 从1.0 版本开始就存在了。

6.1.3 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:

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

6.1.4 信号事件

libevent 也可以监测 POSIX 风格的信号。要构造信号处理器,使用:

```
#define evsignal_new(base, signum, cb, arg) \
    event_new(base, signum, EV_SIGNAL|EV_PERSIST, cb, arg)
```

除了提供一个信号编号代替文件描述符之外,各个参数与 event_new()相同。

实例

```
struct event *hup_event;
struct event_base *base = event_base_new();

/* call sighup_function on a HUP signal */
hup_event = evsignal_new(base, SIGHUP, sighup_function, NULL);
```

> 注意 :信号回调是信号发生后在事件循环中被执行的,所以可以安全地调用通常不能 在 POSIX 风格信号处理器中使用的函数。

警告 :不要在信号事件上设置超时,这可能是不被支持的。 **[待修正:眞是这样的吗?]**

libevent 也提供了一组方便使用的宏用于处理信号事件:

```
#define evsignal_add(ev, tv) \
    event_add((ev),(tv))
#define evsignal_del(ev) \
    event_del(ev)
#define evsignal_pending(ev, what, tv_out) \
    event_pending((ev), (what), (tv_out))
```

evsignal_*宏从2.0.1-alpha 版本开始存在。先前版本中这些宏叫做 signal_add()、signal_del ()等等。

关于信号的警告

在当前版本的 libevent 和大多数后端中,每个进程任何时刻只能有一个 event_base 可以监 听信号。如果同时向两个 event_base 添加信号事件,即使是不同的信号,也只有一 个 event_base 可以取得信号。 kqueue 后端没有这个限制。

6.2 事件的未决和非未决

构造事件之后,在将其添加到 event_base 之前实际上是不能对其做任何操作的。使用event_add()将事件添加到 event_base。

6.2.1 设置未决事件

```
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。

注意 :不要设置 tv 为希望超时事件执行的时间。如果在 2010 年 1 月 1 日设置 “tv->tv_sec=time(NULL)+10;”,超时事件将会等待40年,而不是10秒。

6.2.2 设置非未决事件

```
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。

注意 :如果在事件激活后,其回调被执行前删除事件,回调将不会执行。

这些函数定义在中,从0.1版本就存在了。

6.3 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。

在前一章讨论过,每个 event_base 有与之相关的一个或者多个优先级。在初始化事件之后, 但是在添加到 event_base 之前,可以为其设置优先级。

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

事件的优先级是一个在 0和 event_base 的优先级减去1之间的数值。成功时函数返回 0,失 败时返回-1。

多个不同优先级的事件同时成为激活的时候 ,低优先级的事件不会运行 。libevent 会执行高优先级的事件,然后重新检查各个事件。只有在没有高优先级的事件是激活的时候 ,低优先级的事件才会运行。

实例：

```
#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);
  /* Now base has priority 0, and priority 1 */
  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);

  /* Now, whenever the fd is ready for writing, the write callba
ck will
     happen before the read callback.  The read callback won't h
appen at
     all until the write callback is no longer active. */
}
```

如果不为事件设置优先级,则默认的优先级将会是 event_base 的优先级数目除以 2。

6.4 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。

```
int event_pending(const struct event *ev, short what, struct tim
eval *tv_out);

#define event_get_signal(ev) /* ... */
evutil_socket_t event_get_fd(const struct event *ev);
struct event_base *event_get_base(const struct event *ev);
short event_get_events(const struct event *ev);
event_callback_fn event_get_callback(const struct event *ev);
void *event_get_callback_arg(const struct event *ev);
int event_get_priority(const struct event *ev);

void event_get_assignment(const struct event *event,
        struct event_base **base_out,
        evutil_socket_t *fd_out,
        short *events_out,
        event_callback_fn *callback_out,
        void **arg_out);
```

event_pending()函数确定给定的事件是否是未决的或者激活的。如果是,而且 what 参 数设置了 EV_READ、EV_WRITE、EV_SIGNAL 或者 EV_TIMEOUT 等标志,则函数会返回事件当前为之未决或者激活的所有标志 。如果提供了 tv_out 参数,并且 what 参数中设置了 EV_TIMEOUT 标志,而事件当前正因超时事件而未决或者激活,则 tv_out 会返回事件 的超时值。

event_get_fd()和 event_get_signal()返回为事件配置的文件描述符或者信号值。event_get_base()返回为事件配置的 event_base。event_get_events()返回事件的标志(EV_READ、EV_WRITE 等)。event_get_callback()和 event_get_callback_arg() 返回事件的回调函数及其参数指针。

event_get_assignment()复制所有为事件分配的字段到提供的指针中。任何为 NULL 的参数会被忽略。

实例

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

/* Change the callback and callback_arg of 'ev', which must not
be
 * pending. */
int replace_callback(struct event *ev, event_callback_fn new_cal
lback,
    void *new_callback_arg)
{
    struct event_base *base;
    evutil_socket_t fd;
    short events;

    int pending;

    pending = event_pending(ev, EV_READ|EV_WRITE|EV_SIGNAL|EV_TI
MEOUT,
                            NULL);
    if (pending) {
        /* We want to catch this here so that we do not re-assig
n a
         * pending event.  That would be very very bad. */
        fprintf(stderr,
                "Error! replace_callback called on a pending eve
nt!\n");
        return -1;
    }

    event_get_assignment(ev, &base, &fd, &events,
                         NULL /* ignore old callback */ ,
                         NULL /* ignore old callback argument */
);

    event_assign(ev, base, fd, events, new_callback, new_callbac
k_arg);
    return 0;
}
```

6.5 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。

不能删除或者手动激活使用 event_base_once ()插入的事件:如果希望能够取消事件, 应该使用 event_new()或者 event_assign()。

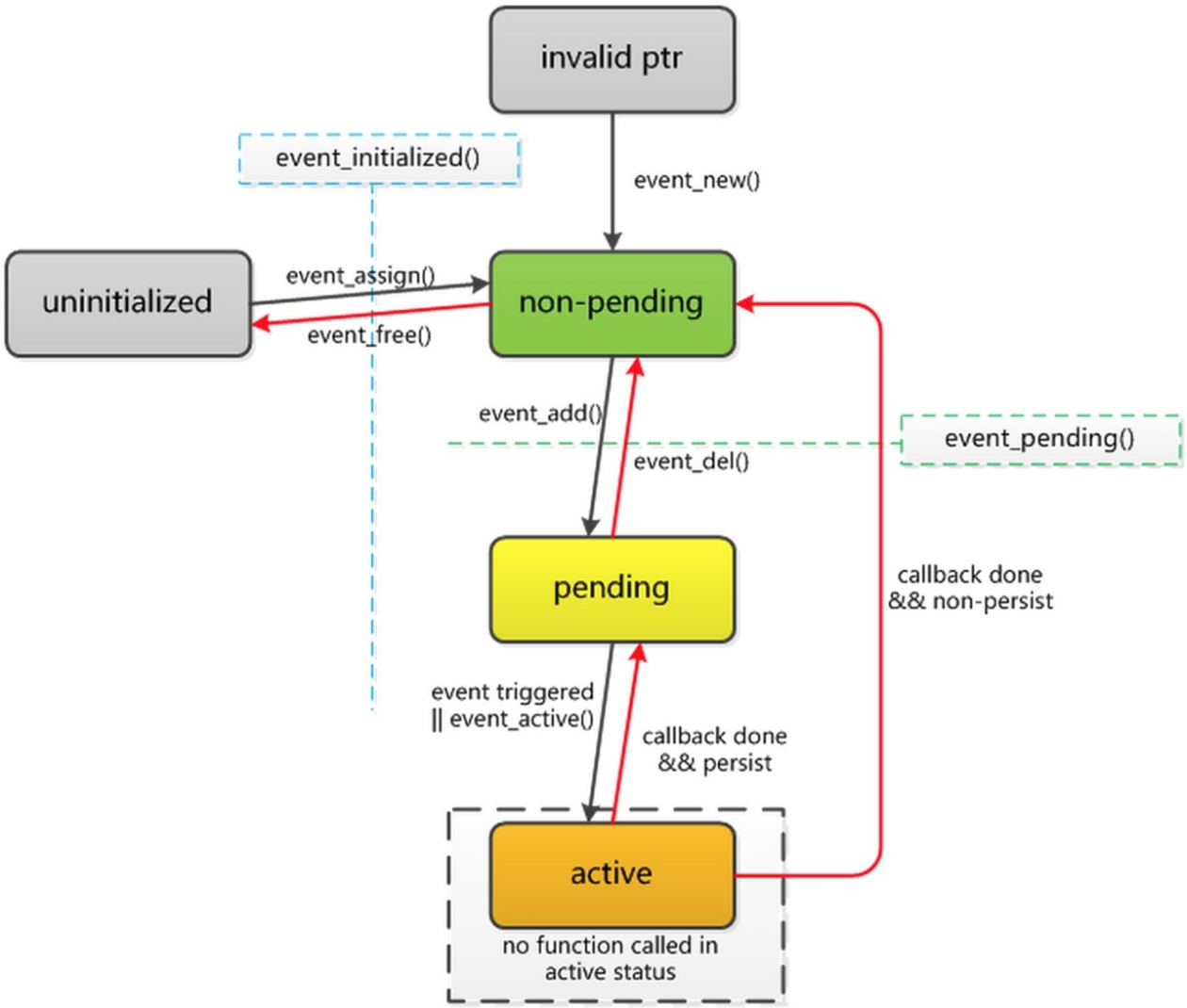
6.6 手动激活事件

极少数情况下,需要在事件的条件没有触发的时候让事件成为激活的。

```
void event_active(struct event *ev, int what, short ncalls);
```

这个函数让事件 ev 带有标志 what(EV_READ、EV_WRITE 和 EV_TIMEOUT 的组合)成 为激活的。事件不需要已经处于未决状态,激活事件也不会让它成为未决的。

6.7 事件状态之间的转换



7 数据缓冲Bufferevent

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:

- 决定要向连接写入一些数据,把数据放入到缓冲区中
- 等待连接可以写入
- 写入尽量多的数据
- 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。

有多种共享公用接口的 bufferevent 类型,编写本文时已存在以下类型:

- `基于套接字的 bufferevent` :使用 event_*接口作为后端,通过底层流式套接字发送或者接收数据的 bufferevent
- `异步 IO bufferevent` :使用 Windows IOCP 接口,通过底层流式套接字发送或者接收数据的 bufferevent(仅用于 Windows,试验中)
- `过滤型 bufferevent` :将数据传输到底层 bufferevent 对象之前,处理输入或者输出数据的 bufferevent:比如说,为了压缩或者转换数据。
- `成对的 bufferevent` :相互传输数据的两个 bufferevent。

> `注意` :截止2.0.2-alpha 版,这里列出的 bufferevent 接口还没有完全正交于所有的 bufferevent 类型。也就是说,下面将要介绍的接口不是都能用于所有 bufferevent 类型。libevent 开发 者在未来版本中将修正这个问题。
>
> `也请注意` :当前 bufferevent 只能用于像 TCP 这样的面向流的协议,将来才可能会支持 像 UDP 这样的面向数据报的协议。

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。 evbuffer 接口支持很多种操作,后面的章节将讨论这些操作。

7.1 回调和水位

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:

- `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。
- `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。
- `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。
- `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。请看后面关于过滤型 bufferevent 的介绍。

7.2 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,随后会介绍)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 。

(延迟回调由 libevent 2.0.1-alpha 版引入)

7.3 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。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。

(BEV_OPT_UNLOCK_CALLBACKS 由2.0.5-beta 版引入,其他选项由 2.0.1-alpha 版引 入)

7.4 使用bufferevent

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。

7.4.1 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。

7.4.2 在bufferevent上启动链接

```
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。

```
#include <event2/event.h>
#include <event2/bufferevent.h>
#include <sys/socket.h>
#include <string.h>

void eventcb(struct bufferevent *bev, short events, void *ptr)
{
    if (events & BEV_EVENT_CONNECTED) {
         /* We're connected to 127.0.0.1:8080.   Ordinarily we'd
 do
            something here, like start reading or writing. */
    } else if (events & BEV_EVENT_ERROR) {
         /* An error occured while connecting. */
    }
}

int main_loop(void)
{
    struct event_base *base;
    struct bufferevent *bev;
    struct sockaddr_in sin;

    base = event_base_new();

    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 */

    bev = bufferevent_socket_new(base, -1, BEV_OPT_CLOSE_ON_FREE
);

    bufferevent_setcb(bev, NULL, NULL, eventcb, NULL);

    if (bufferevent_socket_connect(bev,
        (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        /* Error starting connection */
        bufferevent_free(bev);
```

```
        return -1;
    }

    event_base_dispatch(base);
    return 0;
}
```

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。

7.5.1 释放bufferevent操作

```
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 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。

7.5.2 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。

接口

```
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。

接口

```
void bufferevent_setwatermark(struct bufferevent *bufev, short e
vents,
    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”。

示例

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

```
#include <event2/buffer.h>
#include <event2/util.h>

#include <stdlib.h>
#include <errno.h>
#include <string.h>

struct info {
    const char *name;
    size_t total_drained;
};

void read_callback(struct bufferevent *bev, void *ctx)
{
    struct info *inf = ctx;
    struct evbuffer *input = bufferevent_get_input(bev);
    size_t len = evbuffer_get_length(input);
    if (len) {
        inf->total_drained += len;
        evbuffer_drain(input, len);
        printf("Drained %lu bytes from %s\n",
             (unsigned long) len, inf->name);
    }
}

void event_callback(struct bufferevent *bev, short events, void
*ctx)
{
    struct info *inf = ctx;
    struct evbuffer *input = bufferevent_get_input(bev);
    int finished = 0;

    if (events & BEV_EVENT_EOF) {
        size_t len = evbuffer_get_length(input);
        printf("Got a close from %s.  We drained %lu bytes from
it, "
            "and have %lu left.\n", inf->name,
            (unsigned long)inf->total_drained, (unsigned long)le
n);
        finished = 1;
```

```
    }
    if (events & BEV_EVENT_ERROR) {
        printf("Got an error from %s: %s\n",
            inf->name, evutil_socket_error_to_string(EVUTIL_SOCK
ET_ERROR()));
        finished = 1;
    }
    if (finished) {
        free(ctx);
        bufferevent_free(bev);
    }
}

struct bufferevent *setup_bufferevent(void)
{
    struct bufferevent *b1 = NULL;
    struct info *info1;

    info1 = malloc(sizeof(struct info));
    info1->name = "buffer 1";
    info1->total_drained = 0;

    /* ... Here we should set up the bufferevent and make sure i
t gets
       connected... */

    /* Trigger the read callback only whenever there is at least
 128 bytes
       of data in the buffer. */
    bufferevent_setwatermark(b1, EV_READ, 128, 0);

    bufferevent_setcb(b1, read_callback, NULL, event_callback, i
nfo1);

    bufferevent_enable(b1, EV_READ); /* Start reading. */
    return b1;
}
```

7.5.3 操作bufferevent中的数据

通过bufferevent得到evbuffer

如果只是通过网络读取或者写入数据 ,而不能观察操作过程,是没什么好处的。bufferevent 提供了下列函数用于观察要写入或者读取的数据。

```
struct evbuffer *bufferevent_get_input(struct bufferevent *bufev)
;
struct evbuffer *bufferevent_get_output(struct bufferevent *bufe
v);
```

这两个函数提供了非常强大的基础 :它们分别返回输入和输出缓冲区 。关于可以对 evbuffer 类型进行的所有操作的完整信息,请看下一章。

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。

向bufferevent的输出缓冲区添加数据

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

这些函数向 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 。bufferevent_write_buffer()移除 buf 的所有内 容,将其放置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。

从bufferevent的输入缓冲区移除数据

```
size_t bufferevent_read(struct bufferevent *bufev, void *data, s
ize_t size);
int bufferevent_read_buffer(struct bufferevent *bufev,
    struct evbuffer *buf);
```

这些函数从 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。函数返回实际移除的字节数。 bufferevent_read_buffer()函数抽空输入缓冲区的所有内容,将其放置到 buf 中,成功时返 回0,失败时返回 -1。

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。

示例

```
#include <event2/bufferevent.h>
#include <event2/buffer.h>

#include <ctype.h>

void
read_callback_uppercase(struct bufferevent *bev, void *ctx)
{
        /* This callback removes the data from bev's input buffe
r 128
           bytes at a time, uppercases it, and starts sending it
           back.

           (Watch out!  In practice, you shouldn't use toupper t
o implement
           a network protocol, unless you know for a fact that t
he current
           locale is the one you want to be using.)
         */

        char tmp[128];
        size_t n;
        int i;
```

```
        while (1) {
                n = bufferevent_read(bev, tmp, sizeof(tmp));
                if (n <= 0)
                        break; /* No more data. */
                for (i=0; i<n; ++i)
                        tmp[i] = toupper(tmp[i]);
                bufferevent_write(bev, tmp, n);
        }
}

struct proxy_info {
        struct bufferevent *other_bev;
};
void
read_callback_proxy(struct bufferevent *bev, void *ctx)
{
        /* You might use a function like this if you're implemen
ting
           a simple proxy: it will take data from one connection
 (on
           bev), and write it to another, copying as little as
           possible. */
        struct proxy_info *inf = ctx;

        bufferevent_read_buffer(bev,
            bufferevent_get_output(inf->other_bev));
}

struct count {
        unsigned long last_fib[2];
};

void
write_callback_fibonacci(struct bufferevent *bev, void *ctx)
{
        /* Here's a callback that adds some Fibonacci numbers to
 the
           output buffer of bev.  It stops once we have added 1k
 of
           data; once this data is drained, we'll add more. */
```

```
        struct count *c = ctx;

        struct evbuffer *tmp = evbuffer_new();
        while (evbuffer_get_length(tmp) < 1024) {
                unsigned long next = c->last_fib[0] + c->last_f
ib[1];
                c->last_fib[0] = c->last_fib[1];
                c->last_fib[1] = next;

                evbuffer_add_printf(tmp, "%lu", next);
        }

        /* Now we add the whole contents of tmp to bev. */
        bufferevent_write_buffer(bev, tmp);

        /* We don't need tmp any longer. */
        evbuffer_free(tmp);
}
```

7.5.4 bufferevent的清空操作

```
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。

otype 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。

8 数据封装evBuffer

libevent 的 evbuffer 实现了为向后面添加数据和从前面移除数据而优化的字节队列。

evbuffer 用于处理缓冲网络 IO 的“缓冲”部分。它不提供调度 IO 或者当 IO 就绪时触发 IO 的 功能:这是 bufferevent 的工作。

除非特别说明,本章描述的函数都在 event2/buffer.h中声明。

8.1 创建和释放evbuffer

```
struct evbuffer *evbuffer_new(void);
void evbuffer_free(struct evbuffer *buf);
```

这两个函数的功能很简明: evbuffer_new() 分配和返回一个新的空 evbuffer ; 而 evbuffer_free()释放 evbuffer 和其内容。

8.2 evbuffer与线程安全

```
int evbuffer_enable_locking(struct evbuffer *buf, void *lock);
void evbuffer_lock(struct evbuffer *buf);
void evbuffer_unlock(struct evbuffer *buf);
```

默认情况下,在多个线程中同时访问 evbuffer 是不安全的。如果需要这样的访问,可以调 用 evbuffer_enable_locking() 。如果 lock 参数为 NULL , libevent 会使用 evthread_set_lock_creation_callback 提供的锁创建函数创建一个锁 。否则,libevent 将 lock 参数用作锁。

evbuffer_lock()和 evbuffer_unlock()函数分别请求和释放 evbuffer 上的锁。可以使用这两个 函数让一系列操作是原子的。如果 evbuffer 没有启用锁,这两个函数不做任何操作。

(注意:对于单个操作,不需要调用 evbuffer_lock()和evbuffer_unlock(): 如果 evbuffer 启用了锁,单个操作就已经是原子的 。只有在需要多个操作连续执行 ,不让其他线程介入的 时候,才需要手动锁定 evbuffer)

8.3 检查evbuffer

```
size_t evbuffer_get_length(const struct evbuffer *buf);
//这个函数返回 evbuffer 存储的字节数,它在2.0.1-alpha 版本中引入。
```

```
int evbuffer_add(struct evbuffer *buf,
                 const void *data, size_t datlen);
//这个函数返回连续地存储在 evbuffer前面的字节数。
//evbuffer中的数据可能存储在多个分隔开的内存块中,
//这个函数返回当前第一个块中的字节数。
```

8.4 向evbuffer添加数据

```
int evbuffer_add(struct evbuffer *buf, const void *data, size_t
datlen);
//这个函数添加 data 处的 datalen 字节到 buf 的末尾,
//成功时返回0,失败时返回-1。
```

```
int evbuffer_add_printf(struct evbuffer *buf, const char *fmt, .
..)
int evbuffer_add_vprintf(struct evbuffer *buf, const char *fmt,
va_list ap);
//这些函数添加格式化的数据到 buf 末尾。
//格式参数和其他参数的处理分别与 C 库函数 printf 和 vprintf 相同。函数返
回添加的字节数。
```

```
int evbuffer_expand(struct evbuffer *buf, size_t datlen);
//这个函数修改缓冲区的最后一块,或者添加一个新的块,
//使得缓冲区足以容纳 datlen 字节, 而不需要更多的内存分配。
```

示例

```
/* Here are two ways to add "Hello world 2.0.1" to a buffer. */
/* Directly: */
evbuffer_add(buf, "Hello world 2.0.1", 17);

/* Via printf: */
evbuffer_add_printf(buf, "Hello %s %d.%d.%d", "world", 2, 0, 1);
```

8.5 evbuffer数据移动

为提高效率,libevent 具有将数据从一个 evbuffer 移动到另一个的优化函数。

```
int evbuffer_add_buffer(struct evbuffer *dst, struct evbuffer *s
rc);

int evbuffer_remove_buffer(struct evbuffer *src,
                    struct evbuffer *dst,
                    size_t datlen);
```

evbuffer_add_buffer()将 src 中的所有数据移动到 dst 末尾,成功时返回0,失败时返回-1。

evbuffer_remove_buffer()函数从 src 中移动 datlen 字节到 dst 末尾,尽量少进行复制。如果字节数小于 datlen,所有字节被移动。函数返回移动的字节数。

evbuffer_add_buffer()在0.8版本引入; evbuffer_remove_buffer()是2.0.1-alpha 版本新增加的。

8.6 添加数据到evbuffer前

8 链接监听器evconnlistener

evconnlistener 机制提供了监听和接受 TCP 连接的方法。

本章的所有函数和类型都在 event2/listener.h 中声明,除非特别说明。

它们都在2.0.2-alpha 版本中首次出现。

8.1 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);
```

两个 evconnlistener_new*()函数都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。

更多细节请查看系统的 listen()函数文档。

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。

要释放连接监听器,调用 evconnlistener_free()。

可识别的标志

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。更多信息请查 看 fcntl 和 FD_CLOEXEC 的平台文档。

- 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 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。这是 2.0.8-rc 的新功能。

链接监听器回调

```
typedef void (*evconnlistener_cb)(struct evconnlistener *listene
r,
    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。

8.2 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。

8.3 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。它是 2.0.9-rc 版本引入的。

8.4 检测 evconnlistener

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

这些函数分别返回监听器关联的套接字和 event_base。

8.5 侦测错误

可以设置一个一旦监听器上的 accept()调用失败就被调用的错误回调函数 。对于一个不解决就会锁定进程的错误条件,这很重要。

```
typedef void (*evconnlistener_errorcb)(struct evconnlistener *li
s, void *ptr);
void evconnlistener_set_error_cb(struct evconnlistener *lev,
    evconnlistener_errorcb errorcb);
```

如果使用 evconnlistener_set_error_cb() 为监听器设置了错误回调函数,则监听器发生错误 时回调函数就会被调用。

第一个参数是监听器,

第二个参数是调用 evconnlistener_new() 时传入的 ptr。

9 libevent常用设置

libevent 有一些被整个进程共享的、影响整个库的全局设置。

> 必须在调用**libevent** 库的任何其他部分之前修改这些设置，否则，**libevent** 会进入不一致的状态。

提示：

> 本章节部分内容转自《libevent中文手册（中文版）》

9.1 日志消息回调设置

libevent 可以记录内部错误和警告。如果编译进日志支持，还会记录调试信息。默认配置下 这些信息被写到stderr。通过提供定制的日志函数可以覆盖默认行为。

```
#define EVENT_LOG_DEBUG 0
#define EVENT_LOG_MSG   1
#define EVENT_LOG_WARN  2
#define EVENT_LOG_ERR   3

/* Deprecated; see note at the end of this section */
#define _EVENT_LOG_DEBUG EVENT_LOG_DEBUG
#define _EVENT_LOG_MSG   EVENT_LOG_MSG
#define _EVENT_LOG_WARN  EVENT_LOG_WARN
#define _EVENT_LOG_ERR   EVENT_LOG_ERR


typedef void (*event_log_cb)(int severity, const char *msg);
void event_set_log_callback(event_log_cb cb);
```

要覆盖libevent 的日志行为，编写匹配event_log_cb 签名的定制函数，将其作为参数传递 给event_set_log_callback（）。

随后libevent 在日志信息的时候，将会把信息传递给你提供的函数。再次调用event_set_log_callback（），传递参数NULL，就可以恢复默认行为。

实例

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

static void discard_cb(int severity, const char *msg)
{
    /* This callback does nothing. */
}

static FILE *logfile = NULL;
static void write_to_file_cb(int severity, const char *msg)
{
    const char *s;
    if (!logfile)
        return;
    switch (severity) {
        case _EVENT_LOG_DEBUG: s = "debug"; break;
        case _EVENT_LOG_MSG:   s = "msg";   break;
        case _EVENT_LOG_WARN:  s = "warn";  break;
        case _EVENT_LOG_ERR:   s = "error"; break;
        default:               s = "?";     break; /* never reac
hed */
    }
    fprintf(logfile, "[%s] %s\n", s, msg);
}

/* Turn off all logging from Libevent. */
void suppress_logging(void)
{
    event_set_log_callback(discard_cb);
}

/* Redirect all Libevent log messages to the C stdio file 'f'. */

void set_logfile(FILE *f)
{
    logfile = f;
    event_set_log_callback(write_to_file_cb);
}
```

在用户提供的event_log_cb 回调函数中调用libevent 函数是不安全的。

比如说，如果试图编写一个使用bufferevent 将警告信息发送给某个套接字的日志回调函数，可能会遇到奇怪 而难以诊断的bug。未来版本libevent 的某些函数可能会移除这个限制。

这个函数在中声明，在libevent 1.0c 版本中首次出现。

9.2 致命错误回调设置

libevent 在检测到不可恢复的内部错误时的默认行为是调用exit（）或者abort（），退出正在运行的进程。这类错误通常意味着某处有bug：要么在你的代码中，要么在libevent 中。

如果希望更优雅地处理致命错误，可以为libevent 提供在退出时应该调用的函数，覆盖默认 行为。

```
typedef void (*event_fatal_cb)(int err);
void event_set_fatal_callback(event_fatal_cb cb);
```

要使用这些函数，首先定义libevent 在遇到致命错误时应该调用的函数，将其传递给 event_set_fatal_callback（）。

随后libevent 在遇到致命错误时将调用你提供的函数。 你的函数不应该将控制返回到libevent：这样做可能导致不确定的行为。

为了避免崩溃，libevent 还是会退出。你的函数被不应该调用其它libevent 函数。这些函数声明在中，在libevent 2.0.3-alpha 版本中首次出现。

9.3 内存管理回调设置

默认情况下，libevent 使用C 库的内存管理函数在堆上分配内存。

通过提供malloc、realloc和free 的替代函数，可以让libevent 使用其他的内存管理器。

希望libevent 使用一个更高效的分配器时；或者希望libevent 使用一个工具分配器，以便检查内存泄漏时，可能需要这样做。

```
void event_set_mem_functions(void *(*malloc_fn)(size_t sz),
                             void *(*realloc_fn)(void *ptr, size
_t sz),
                             void (*free_fn)(void *ptr));
```

这里有个替换libevent 分配器函数的示例，它可以计算已经分配的字节数。

实际应用中可能 需要添加锁，以避免运行在多个线程中时发生错误。

实例

```

#include <event2/event.h>
#include <sys/types.h>
#include <stdlib.h>

/* This union's purpose is to be as big as the largest of all th
e
 * types it contains. */
union alignment {
    size_t sz;
    void *ptr;
    double dbl;
};
/* We need to make sure that everything we return is on the righ
t
   alignment to hold anything, including a double. */
```

```
#define ALIGNMENT sizeof(union alignment)

/* We need to do this cast-to-char* trick on our pointers to adj
ust
   them; doing arithmetic on a void* is not standard. */
#define OUTPTR(ptr) (((char*)ptr)+ALIGNMENT)
#define INPTR(ptr) (((char*)ptr)-ALIGNMENT)

static size_t total_allocated = 0;
static void *replacement_malloc(size_t sz)
{
    void *chunk = malloc(sz + ALIGNMENT);
    if (!chunk) return chunk;
    total_allocated += sz;
    *(size_t*)chunk = sz;
    return OUTPTR(chunk);
}
static void *replacement_realloc(void *ptr, size_t sz)
{
    size_t old_size = 0;
    if (ptr) {
        ptr = INPTR(ptr);
        old_size = *(size_t*)ptr;
    }
    ptr = realloc(ptr, sz + ALIGNMENT);
    if (!ptr)
        return NULL;
    *(size_t*)ptr = sz;
    total_allocated = total_allocated - old_size + sz;
    return OUTPTR(ptr);
}
static void replacement_free(void *ptr)
{
    ptr = INPTR(ptr);
    total_allocated -= *(size_t*)ptr;
    free(ptr);
}
void start_counting_bytes(void)
{
    event_set_mem_functions(replacement_malloc,
```

```
                                replacement_realloc,
                                replacement_free);
}
```

注意

- 替换内存管理函数影响libevent 随后的所有分配、调整大小和释放内存操作。所以，必 须保证在调用任何其他libevent 函数之前进行替换。否则，libevent 可能用你的free 函 数释放用C 库的malloc 分配的内存。
- 你的malloc 和realloc 函数返回的内存块应该具有和C 库返回的内存块一样的地址对 齐。
- 你的realloc 函数应该正确处理realloc(NULL,sz)（也就是当作malloc(sz)处理）
- 你的realloc 函数应该正确处理realloc(ptr,0)（也就是当作free(ptr)处理）
- 你的free 函数不必处理free(NULL)
- 你的malloc 函数不必处理malloc(0)
- 如果在多个线程中使用libevent，替代的内存管理函数需要是线程安全的。
- libevent 将使用这些函数分配返回给你的内存。所以，如果要释放由libevent 函数分配 和返回的内存，而你已经替换malloc 和realloc 函数，那么应该使用替代的free 函数。

event_set_mem_functions 函数声明在中，在libevent 2.0.1-alpha 版本中 首次出现。

可以在禁止event_set_mem_functions 函数的配置下编译libevent 。这时候使用 event_set_mem_functions 将不会编译或者链接。

在2.0.2-alpha 及以后版本中，可以通过检查是否定义了 EVENT_SET_MEM_FUNCTIONS_IMPLEMENTED 宏来确定 event_set_mem_functions 函数是否存在。

9.4 锁和线程的设置

编写多线程程序的时候,在多个线程中同时访问同样的数据并不总是安全的。

libevent 的结构体在多线程下通常有三种工作方式:

- 某些结构体内在地是单线程的:同时在多个线程中使用它们总是不安全的。
- 某些结构体具有可选的锁:可以告知 libevent 是否需要在多个线程中使用每个对象。
- 某些结构体总是锁定的 :如果 libevent 在支持锁的配置下运行 ,在多个线程中使用它们 总是安全的。

为获取锁,在调用分配需要在多个线程间共享的结构体的 libevent 函数之前,必须告知 libevent 使用哪个锁函数。

如果使用 pthreads 库,或者使用 Windows 本地线程代码,那么你是幸运的:已经有设置 libevent 使用正确的 pthreads 或者 Windows 函数的预定义函数。

接口

```
#ifdef WIN32
  int evthread_use_windows_threads(void);
#define EVTHREAD_USE_WINDOWS_THREADS_IMPLEMENTED
#endif

#ifdef _EVENT_HAVE_PTHREADS
  int evthread_use_pthreads(void);
#define EVTHREAD_USE_PTHREADS_IMPLEMENTED

#endif
```

这些函数在成功时都返回 0,失败时返回 -1。

如果使用不同的线程库,则需要一些额外的工作,必须使用你的线程库来定义函数去实现:

- 锁
- 锁定
- 解锁
- 分配锁
- 析构锁
- 条件变量
- 创建条件变量
- 析构条件变量
- 等待条件变量
- 触发/广播某条件变量
- 线程
- 线程ID检测

使用 evthread_set_lock_callbacks 和 evthread_set_id_callback 接口告知 libevent 这些函数。

接口

```
#define EVTHREAD_WRITE  0x04
#define EVTHREAD_READ   0x08
#define EVTHREAD_TRY    0x10

#define EVTHREAD_LOCKTYPE_RECURSIVE 1
#define EVTHREAD_LOCKTYPE_READWRITE 2

#define EVTHREAD_LOCK_API_VERSION 1

struct evthread_lock_callbacks {
       int lock_api_version;
       unsigned supported_locktypes;
       void *(*alloc)(unsigned locktype);
       void (*free)(void *lock, unsigned locktype);
       int (*lock)(unsigned mode, void *lock);
       int (*unlock)(unsigned mode, void *lock);
};

int evthread_set_lock_callbacks(const struct evthread_lock_callb
acks *);

void evthread_set_id_callback(unsigned long (*id_fn)(void));

struct evthread_condition_callbacks {
        int condition_api_version;
        void *(*alloc_condition)(unsigned condtype);
        void (*free_condition)(void *cond);
        int (*signal_condition)(void *cond, int broadcast);
        int (*wait_condition)(void *cond, void *lock,
            const struct timeval *timeout);
};

int evthread_set_condition_callbacks(
        const struct evthread_condition_callbacks *);
```

evthread_lock_callbacks 结构体描述的锁回调函数及其能力。对于上述版本,

lock*api_version* 字段必须设置为 *EVTHREAD_LOCK_API_VERSION* 。必须设置 *supported_locktypes* 字段为 *EVTHREAD_LOCKTYPE** 常量的组合以描述支持的锁类型 (在 2.0.4-alpha 版本中),

EVTHREAD_LOCK_RECURSIVE 是必须的,

EVTHREAD_LOCK_READWRITE 则没有使用)。

alloc 函数必须返回指定类型的新锁 ;

free 函数必须释放指定类型锁持有的所有资源 ;

lock 函数必须试图以指定模式请求锁定 ,如果成 功则返回0,失败则返回非零;

unlock 函数必须试图解锁,成功则返回 0,否则返回非零。

可识别的锁类型有:

- 0:通常的,不必递归的锁。
- EVTHREAD_LOCKTYPE_RECURSIVE :不会阻塞已经持有它的线程的锁 。一旦持有它的线程进行原来锁定次数的解锁,其他线程立刻就可以请求它了。
- EVTHREAD_LOCKTYPE_READWRITE :可以让多个线程同时因为读而持有它 ,但是 任何时刻只有一个线程因为写而持有它。写操作排斥所有读操作。

可识别的锁模式有:

- EVTHREAD_READ :仅用于读写锁:为读操作请求或者释放锁
- EVTHREAD_WRITE :仅用于读写锁:为写操作请求或者释放锁
- EVTHREAD_TRY :仅用于锁定:仅在可以立刻锁定的时候才请求锁定

id_fn 参数必须是一个函数,它返回一个无符号长整数,标识调用此函数的线程。对于相同 线程,这个函数应该总是返回同样的值 ;而对于同时调用该函数的不同线程 ,必须返回不同 的值。

vthread_condition_callbacks 结构体描述了与条件变量相关的回调函数。对于上述版本 , condition_api_version 字段必须设置为 EVTHREAD_CONDITION_API_VERSION 。 alloc_condition 函数必须返回到新条件变量的指针 。它接受0作为其参数。free_condition 函 数必须释放条件变量持有的存储器和资源。 wait_condition 函数要求三个参数:一个 由 alloc_condition 分配

的条件变量 ,一个由你提供的 evthread_lock_callbacks.alloc 函数分配 的锁,以及一个可选的超时值 。调用本函数时 ,必须已经持有参数指定的锁 ;本函数应该释 放指定的锁,等待条件变量成为授信状态,或者直到指定的超时时间已经流逝(可选)。wait_condition 应该在错误时返回-1,条件变量授信时返回0,超时时返回1。返回之前,函 数应该确定其再次持有锁。最后, signal_condition 函数应该唤醒等待该条件变量的某个线 程(broadcast 参数为 false 时),或者唤醒等待条件变量的所有线程(broadcast 参数为 true 时)。只有在持有与条件变量相关的锁的时候,才能够进行这些操作。

> 关于条件变量的更多信息,请查看 pthreads 的 pthread*cond*函数文档,或者 Windows 的 CONDITION_VARIABLE(Windows Vista 新引入的)函数文档。

实例：

```
关于使用这些函数的示例,

请查看 Libevent 源代码发布版本中的
evthread_pthread.c 和 evthread_win32.c 文件。
```

这些函数在 中声明,其中大多数在 2.0.4-alpha 版本中首次出现。 2.0.1-alpha 到 2.0.3-alpha 使用较老版本的锁函数 。event_use_pthreads 函数要求程序链接 到 event_pthreads 库。

条件变量函数是2.0.7-rc 版本新引入的,用于解决某些棘手的死锁问题。

可以创建禁止锁支持的libevent。这时候已创建的使用上述线程相关函数的程序将不能运行。

调试做的使用

为帮助调试锁的使用,libevent 有一个可选的“锁调试”特征。这个特征包装了锁调用,以便捕获典型的锁错误,包括:

- 解锁并没有持有的锁
- 重新锁定一个非递归锁

如果发生这些错误中的某一个, libevent 将给出断言失败并且退出。

```
void event_enable_debug_mode(void);
```

必须在创建或者使用任何锁之前调用这个函数。为安全起见,请在设置完线程函数后立即调用这个函数。

9.5 调试事件的使用

libevent 可以检测使用事件时的一些常见错误并且进行报告。这些错误包括:

- 将未初始化的 event 结构体当作已经初始化的
- 试图重新初始化未决的 event 结构体

跟踪哪些事件已经初始化需要使用额外的内存和处理器时间 ,所以只应该在眞正调试程序的 时候才启用调试模式。

```
void event_enable_debug_mode(void);
```

必须在创建任何 event_base 之前调用这个函数。

如果在调试模式下使用大量由 event_assign(而不是 event_new)创建的事件,程序可能 会耗尽内存,这是因为没有方式可以告知 libevent 由 event_assign 创建的事件不会再被使 用了(可以调用 event_free 告知由 event_new 创建的事件已经无效了)。如果想在调试时 避免耗尽内存,可以显式告知 libevent 这些事件不再被当作已分配的了:

```
void event_debug_unassign(struct event *ev);
```

没有启用调试的时候调用 event_debug_unassign 没有效果。

实例

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

#include <stdlib.h>

void cb(evutil_socket_t fd, short what, void *ptr)
{
    /* We pass 'NULL' as the callback pointer for the heap alloc
ated
```

```
     * event, and we pass the event itself as the callback point
er
     * for the stack-allocated event. */
    struct event *ev = ptr;

    if (ev)
        event_debug_unassign(ev);
}

/* Here's a simple mainloop that waits until fd1 and fd2 are bot
h
 * ready to read. */
void mainloop(evutil_socket_t fd1, evutil_socket_t fd2, int debu
g_mode)
{
    struct event_base *base;
    struct event event_on_stack, *event_on_heap;

    if (debug_mode)
       event_enable_debug_mode();

    base = event_base_new();

    event_on_heap = event_new(base, fd1, EV_READ, cb, NULL);
    event_assign(&event_on_stack, base, fd2, EV_READ, cb, &event
_on_stack);

    event_add(event_on_heap, NULL);
    event_add(&event_on_stack, NULL);

    event_base_dispatch(base);

    event_free(event_on_heap);
    event_base_free(base);
}
```

10 基于libevent服务器

10.1 Hello_World服务器

```
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <signal.h>
#ifndef WIN32
#include <netinet/in.h>
# ifdef _XOPEN_SOURCE_EXTENDED
#  include <arpa/inet.h>
# endif
#include <sys/socket.h>
#endif

#include <event2/bufferevent.h>
#include <event2/buffer.h>
#include <event2/listener.h>
#include <event2/util.h>
#include <event2/event.h>

static const char MESSAGE[] = "Hello, World!\n";

static const int PORT = 9995;

static void listener_cb(struct evconnlistener *, evutil_socket_t
,
    struct sockaddr *, int socklen, void *);
static void conn_writecb(struct bufferevent *, void *);
static void conn_eventcb(struct bufferevent *, short, void *);
static void signal_cb(evutil_socket_t, short, void *);

int
main(int argc, char **argv)
{
        struct event_base *base;
        struct evconnlistener *listener;
        struct event *signal_event;
```

```
        struct sockaddr_in sin;
#ifdef WIN32
        WSADATA wsa_data;
        WSAStartup(0x0201, &wsa_data);
#endif

        base = event_base_new();
        if (!base) {
                fprintf(stderr, "Could not initialize libevent!\
n");
                return 1;
        }

        memset(&sin, 0, sizeof(sin));
        sin.sin_family = AF_INET;
        sin.sin_port = htons(PORT);

        listener = evconnlistener_new_bind(base, listener_cb, (v
oid *)base,
            LEV_OPT_REUSEABLE|LEV_OPT_CLOSE_ON_FREE, -1,
            (struct sockaddr*)&sin,
            sizeof(sin));

        if (!listener) {
                fprintf(stderr, "Could not create a listener!\n"
);
                return 1;
        }

        signal_event = evsignal_new(base, SIGINT, signal_cb, (vo
id *)base);

        if (!signal_event || event_add(signal_event, NULL)<0) {
                fprintf(stderr, "Could not create/add a signal e
vent!\n");
                return 1;
        }

        event_base_dispatch(base);
```

```
        evconnlistener_free(listener);
        event_free(signal_event);
        event_base_free(base);

        printf("done\n");
        return 0;
}

static void
listener_cb(struct evconnlistener *listener, evutil_socket_t fd,
    struct sockaddr *sa, int socklen, void *user_data)
{
        struct event_base *base = user_data;
        struct bufferevent *bev;

        bev = bufferevent_socket_new(base, fd, BEV_OPT_CLOSE_ON_
FREE);
        if (!bev) {
                fprintf(stderr, "Error constructing bufferevent!"
);
                event_base_loopbreak(base);
                return;
        }
        bufferevent_setcb(bev, NULL, conn_writecb, conn_eventcb,
NULL);
        bufferevent_enable(bev, EV_WRITE);
        bufferevent_disable(bev, EV_READ);

        bufferevent_write(bev, MESSAGE, strlen(MESSAGE));
}

static void
conn_writecb(struct bufferevent *bev, void *user_data)
{
        struct evbuffer *output = bufferevent_get_output(bev);
        if (evbuffer_get_length(output) == 0) {
                printf("flushed answer\n");
                bufferevent_free(bev);
        }
```

```
}

static void
conn_eventcb(struct bufferevent *bev, short events, void *user_d
ata)
{
        if (events & BEV_EVENT_EOF) {
                printf("Connection closed.\n");
        } else if (events & BEV_EVENT_ERROR) {
                printf("Got an error on the connection: %s\n",
                    strerror(errno));/*XXX win32*/
        }
        /* None of the other events can happen here, since we ha
ven't enabled
         * timeouts */
        bufferevent_free(bev);
}

static void
signal_cb(evutil_socket_t sig, short events, void *user_data)
{
        struct event_base *base = user_data;
        struct timeval delay = { 2, 0 };

        printf("Caught an interrupt signal; exiting cleanly in t
wo seconds.\n");

        event_base_loopexit(base, &delay);
}
```

10.2 基于事件服务器

服务端

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

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/queue.h>
#include <unistd.h>
#include <sys/time.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

#include <event.h>

static void
fifo_read(evutil_socket_t fd, short event, void *arg)
{
        char buf[255];
        int len;
        struct event *ev = arg;

        /* Reschedule this event */
        event_add(ev, NULL);

        fprintf(stderr, "fifo_read called with fd: %d, event: %d
, arg: %p\n",
            (int)fd, event, arg);
        len = read(fd, buf, sizeof(buf) - 1);

        if (len == -1) {
                perror("read");
                return;
```

```
        } else if (len == 0) {
                fprintf(stderr, "Connection closed\n");
                return;
        }

        buf[len] = '\0';

        fprintf(stdout, "Read: %s\n", buf);
}

int
main(int argc, char **argv)
{
        struct event evfifo;

        struct stat st;
        const char *fifo = "event.fifo";
        int socket;

        if (lstat(fifo, &st) == 0) {
                if ((st.st_mode & S_IFMT) == S_IFREG) {
                        errno = EEXIST;
                        perror("lstat");
                        exit(1);
                }
        }

        unlink(fifo);
        if (mkfifo(fifo, 0600) == -1) {
                perror("mkfifo");
                exit(1);
        }

        /* Linux pipes are broken, we need O_RDWR instead of O_R
DONLY */
        socket = open(fifo, O_RDONLY | O_NONBLOCK, 0);

        if (socket == -1) {
                perror("open");
                exit(1);
```

```
        }

        fprintf(stderr, "Write data to %s\n", fifo);

        /* Initalize the event library */
        event_init();

        /* Initalize one event */
        event_set(&evfifo, socket, EV_READ, fifo_read, &evfifo);

        /* Add it to the active events, without a timeout */
        event_add(&evfifo, NULL);

        event_dispatch();

        return (0);
}
```

客户端

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>

int main(int argc, char *argv[])
{
    int fd = 0;
    char *str = "hello libevent!";

    fd = open("event.fifo", O_RDWR);
    if (fd < 0) {
        perror("open error");
        exit(1);
    }

    while (1) {
        write(fd, str, strlen(str));
        sleep(1);
    }

    close(fd);

        return 0;
}
```

10.3 回显服务器

```
#include <event2/listener.h>
#include <event2/bufferevent.h>
#include <event2/buffer.h>

#include <arpa/inet.h>

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

static void
echo_read_cb(struct bufferevent *bev, void *ctx)
{
        /* This callback is invoked when there is data to read o
n bev. */
        struct evbuffer *input = bufferevent_get_input(bev);
        struct evbuffer *output = bufferevent_get_output(bev);

        /* Copy all the data from the input buffer to the output
 buffer. */
        evbuffer_add_buffer(output, input);
}

static void
echo_event_cb(struct bufferevent *bev, short events, void *ctx)
{
        if (events & BEV_EVENT_ERROR)
                perror("Error from bufferevent");
        if (events & (BEV_EVENT_EOF | BEV_EVENT_ERROR)) {
                bufferevent_free(bev);
        }
}

static void
```

```
accept_conn_cb(struct evconnlistener *listener,
    evutil_socket_t fd, struct sockaddr *address, int socklen,
    void *ctx)
{
        /* We got a new connection! Set up a bufferevent for it.
 */
        struct event_base *base = evconnlistener_get_base(listen
er);
        struct bufferevent *bev = bufferevent_socket_new(
                base, fd, BEV_OPT_CLOSE_ON_FREE);

        bufferevent_setcb(bev, echo_read_cb, NULL, echo_event_cb
, NULL);

        bufferevent_enable(bev, EV_READ|EV_WRITE);
}

static void
accept_error_cb(struct evconnlistener *listener, void *ctx)
{
        struct event_base *base = evconnlistener_get_base(listen
er);
        int err = EVUTIL_SOCKET_ERROR();
        fprintf(stderr, "Got an error %d (%s) on the listener. "
                "Shutting down.\n", err, evutil_socket_error_to_
string(err));

        event_base_loopexit(base, NULL);
}

int
main(int argc, char **argv)
{
        struct event_base *base;
        struct evconnlistener *listener;
        struct sockaddr_in sin;

        int port = 9876;

        if (argc > 1) {
```

```
                port = atoi(argv[1]);
        }
        if (port<=0 || port>65535) {
                puts("Invalid port");
                return 1;
        }

        base = event_base_new();
        if (!base) {
                puts("Couldn't open event base");
                return 1;
        }

        /* Clear the sockaddr before using it, in case there are
 extra
         * platform-specific fields that can mess us up. */
        memset(&sin, 0, sizeof(sin));
        /* This is an INET address */
        sin.sin_family = AF_INET;
        /* Listen on 0.0.0.0 */
        sin.sin_addr.s_addr = htonl(0);
        /* Listen on the given port. */
        sin.sin_port = htons(port);

        listener = evconnlistener_new_bind(base, accept_conn_cb,
NULL,
            LEV_OPT_CLOSE_ON_FREE|LEV_OPT_REUSEABLE, -1,
            (struct sockaddr*)&sin, sizeof(sin));
        if (!listener) {
                perror("Couldn't create listener");
                return 1;
        }
        evconnlistener_set_error_cb(listener, accept_error_cb);

        event_base_dispatch(base);
        return 0;
}
```

10.3 libevent实现http服务器

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>     //for getopt, fork
#include <string.h>     //for strcat
//for struct evkeyvalq
#include <sys/queue.h>
#include <event.h>
//for http
//#include <evhttp.h>
#include <event2/http.h>
#include <event2/http_struct.h>
#include <event2/http_compat.h>
#include <event2/util.h>
#include <signal.h>

#define MYHTTPD_SIGNATURE   "myhttpd v 0.0.1"

//处理模块
void httpd_handler(struct evhttp_request *req, void *arg) {
    char output[2048] = "\0";
    char tmp[1024];

    //获取客户端请求的URI(使用evhttp_request_uri或直接req->uri)
    const char *uri;
    uri = evhttp_request_uri(req);
    sprintf(tmp, "uri=%s\n", uri);
    strcat(output, tmp);

    sprintf(tmp, "uri=%s\n", req->uri);
    strcat(output, tmp);
    //decoded uri
    char *decoded_uri;
    decoded_uri = evhttp_decode_uri(uri);
    sprintf(tmp, "decoded_uri=%s\n", decoded_uri);
    strcat(output, tmp);
```

```
    //解析URI的参数(即GET方法的参数)
    struct evkeyvalq params;
    //将URL数据封装成key-value格式,q=value1, s=value2
    evhttp_parse_query(decoded_uri, &params);
    //得到q所对应的value
    sprintf(tmp, "q=%s\n", evhttp_find_header(&params, "q"));
    strcat(output, tmp);
    //得到s所对应的value
    sprintf(tmp, "s=%s\n", evhttp_find_header(&params, "s"));
    strcat(output, tmp);

    free(decoded_uri);

    //获取POST方法的数据
    char *post_data = (char *) EVBUFFER_DATA(req->input_buffer);
    sprintf(tmp, "post_data=%s\n", post_data);
    strcat(output, tmp);

    /*
       具体的：可以根据GET/POST的参数执行相应操作，然后将结果输出
       ...
     */

    /* 输出到客户端 */

    //HTTP header
    evhttp_add_header(req->output_headers, "Server", MYHTTPD_SIG
NATURE);
    evhttp_add_header(req->output_headers, "Content-Type", "text
/plain; charset=UTF-8");
    evhttp_add_header(req->output_headers, "Connection", "close"
);
    //输出的内容
    struct evbuffer *buf;
    buf = evbuffer_new();
    evbuffer_add_printf(buf, "It works!\n%s\n", output);
    evhttp_send_reply(req, HTTP_OK, "OK", buf);
    evbuffer_free(buf);
```

```
}
void show_help() {
    char *help = "http://localhost:8080\n"
        "-l <ip_addr> interface to listen on, default is 0.0.0.0
\n"
        "-p <num>     port number to listen on, default is 1984\
n"
        "-d           run as a deamon\n"
        "-t <second>  timeout for a http request, default is 120
 seconds\n"
        "-h           print this help and exit\n"
        "\n";
    fprintf(stderr,"%s",help);
}
//当向进程发出SIGTERM/SIGHUP/SIGINT/SIGQUIT的时候，终止event的事件侦听
循环
void signal_handler(int sig) {
    switch (sig) {
        case SIGTERM:
        case SIGHUP:
        case SIGQUIT:
        case SIGINT:
            event_loopbreak();  //终止侦听event_dispatch()的事件侦
听循环，执行之后的代码
            break;
    }
}

int main(int argc, char *argv[]) {
    //自定义信号处理函数
    signal(SIGHUP, signal_handler);
    signal(SIGTERM, signal_handler);
    signal(SIGINT, signal_handler);
    signal(SIGQUIT, signal_handler);

    //默认参数
    char *httpd_option_listen = "0.0.0.0";
    int httpd_option_port = 8080;
    int httpd_option_daemon = 0;
    int httpd_option_timeout = 120; //in seconds
```

```
    //获取参数
    int c;
    while ((c = getopt(argc, argv, "l:p:dt:h")) != -1) {
        switch (c) {
            case 'l' :
                httpd_option_listen = optarg;
                break;
            case 'p' :
                httpd_option_port = atoi(optarg);
                break;
            case 'd' :
                httpd_option_daemon = 1;
                break;
            case 't' :
                httpd_option_timeout = atoi(optarg);
                break;
            case 'h' :
            default :
                show_help();
                exit(EXIT_SUCCESS);
        }
    }

    //判断是否设置了-d，以daemon运行
    if (httpd_option_daemon) {
        pid_t pid;
        pid = fork();
        if (pid < 0) {
            perror("fork failed");
            exit(EXIT_FAILURE);
        }
        if (pid > 0) {
            //生成子进程成功，退出父进程
            exit(EXIT_SUCCESS);
        }
    }

    /* 使用libevent创建HTTP Server */
```

```
    //初始化event API
    event_init();

    //创建一个http server
    struct evhttp *httpd;
    httpd = evhttp_start(httpd_option_listen, httpd_option_port)
;
    evhttp_set_timeout(httpd, httpd_option_timeout);

    //指定generic callback
    evhttp_set_gencb(httpd, httpd_handler, NULL);
    //也可以为特定的URI指定callback
    //evhttp_set_cb(httpd, "/", specific_handler, NULL);

    //循环处理events
    event_dispatch();

    evhttp_free(httpd);
    return 0;
}
```