-

一 libevent 简介

libevent 是一个支持 Windows、linux 和 bsd 等平台的网络事件驱动程序库。它支持多种 I/O 服用机制，按照优先级从高到低依次为：evport、kqueue、epoll、devpoll、rtsig、poll、select。它可根据操作系统，按照优先级从高到底自主选择驱动。

用户可以通过 http://www.monkey.org/~provos/libevent/来获取 libevent 的源码、libevent 出现的背景、以及其他一些详细资料。

二 libevent 的使用

1 初始化事件

我们首先完成对 libenvent 的事件初始化和事件驱动模型的选择。（在使用多线程的情况下，一般我们需获取所返回的事件根基）

最基本的事件是event，
最基本的队列/链表是event_base，
event可以挂在event_base上，
一个event_base可以挂几个event.

```
main_base = event_init();
```

event_init 函数返回的是一个 event_base 对象，该对象包括了事件处理过程中的一些全局变量，其结构为：

```
struct event_base {
  const struct eventop *evsel;
  void *evbase;
  int event_count;      /* counts number of total events */
  int event_count_active; /* counts number of active events */

  int event_gotterm;    /* Set to terminate loop */
  int event_break;      /* Set to terminate loop immediately */

  /* active event management */
  struct event_list **activequeues;
  int nactivequeues;

  /* signal handling info */
  struct evsignal_info sig;

  struct event_list eventqueue;
  struct timeval event_tv;

  struct min_heap timeheap;
```

```
    struct timeval tv_cache;
};
```

event_set作用有二
1设置event的事件敏感标志位
2将event挂接到event_base链表上

event_add作用
将event挂接到系统事件队列中等待被调用。

2 添加事件

在事件初始化完毕后，我们可以使用 **event_set** 设置事件，然后使用 **event_add** 将其加入。

这里我们首先完成 **socket** 的监听，然后将其加入的事件队列中（这里我们对所有的异常都不做考虑）。

（**1**）**socket** 监听

```
struct sockaddr_in listen_addr;

int port = 10000; //socket 监听端口

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

memset(&listen_addr, 0, sizeof(listen_addr));

listen_addr.sin_family = AF_INET;
listen_addr.sin_addr.s_addr = INADDR_ANY;
listen_addr.sin_port = htons(port)

reuseaddr_on = 1;

setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &reuseaddr_on, sizeof(reuseaddr_on)) #支持端口复用

bind(listen_fd, (struct sockaddr *) &listen_addr, sizeof(listen_addr))'

listen(listen_fd, 1024);

/*将描述符设置为非阻塞*/

int flags = fcntl(listen_fd,, F_GETFL);

flags |= O_NONBLOCK;

fcntl(listen_fd, F_SETFL, flags);
```

（**2**）事件设置

socket 服务建立后，就可以进行事件设置。我们使用 event_set 来设置事件对象，其传入参数包括事件根基(event_base 对象)，描述符，事件类型，事件发生时的回调函数，回调函数传入参数。其中事件类型包括 EV_READ、EV_WRITE、EV_PERSIST，EV_PERSIST 和前两者结合使用，表示该事件为持续事件。

```
struct event ev;

event_set(&ev, listen_fd, EV_READ | EV_PERSIST, accept_handle, (void *)&ev);
```

需要监控的fd及敏感事件

回调函数及传入回调函数的参数

所有的信息都被记录在struct event对象中。

（3）事件添加与删除

事件设置好后，就可以将其加入事件队列。event_add 用来将事件加入，它接受两个参数：要添加的事件和时间的超时值。如果需要将事件删除，可以使用 event_del 来完成。event_del 函数会取消所指定的事件。

```
event_add(&ev, NULL)
```

3 进入事件循环

事件成功添加后就是万事具备只欠东风了，libevent 提供了多种方式来进入事件循环，我个人常用的是 event_dispatch 和 event_base_loop，前者最后实际是使用当前事件根基来调用 event_base_loop。

```
event_base_loop(main_base, 0);
```

4 处理连接

到这里为止，大家已经完成了事件的设置、事件的添加并进入到了事件循环。但是当事件发生时（这里就是连接建立）如何处理呢？聪明的用户会想到前面我们在事件设置时指定的回调函数 accept_handle。没错，当连接建立时回调函数 accept_handle 会自动的得到调用。

在回调函数中如果依然使用read/write来读写的时，又就变成阻塞的了，为了避免这种情况，libevent实现了读写缓冲区，数据先被读到输入缓冲区，再通过回调提供给上层，数据先写入输出缓冲区，再适机通过write()写出，一切都是异步的。

对于缓冲区的读写在非阻塞式网络编程中是一个难以处理的问题，幸运的是 libevent 提供了 bufferevent 和 evbuf 来替我们完成该项工作。这里我们采用 bufferevent 来处理。

(1)生成 bufferevent 对象

使用 bufferevent_new 对象来生成 bufferevent 对象，并分别指定读、写、连接错误时的处理函数和函数传入参数。

(2) 设置读取量

bufferevent 的读事件激活以后，即使用户没有读取完 bufferevent 缓冲区中的数据，bufferevent 读事件也不会再次被激活。因为 bufferevent 的读事件是由其所监控的描述符的读事件激活的，只有描述符可读，读事件才会被激活。可通过设置 wm_read.high 来控制 bufferevent 从描述符缓冲区中读取的数据量。

(3) 将事件加入事件队列

和前面一样，在事件设置好后，需将事件加入到事件队列中，不过 **bufferevent** 的有自己专门的加入函数 **bufferevent_base_set** 和激活函数 **bufferevent_enable**。

bufferevent 接收两个参数事件根基个事件对象，前者用来指定事件将加入到哪个事件根基中，后者说明需将那个 **bufferevnet** 事件加入。(在多线程的情况下，每个线程可能有自己单独的事件根基)

在 **bufferevent** 初始化完毕后，可以使用 **bufferevent_enable** 和 **bufferevent_disable** 反复的激活与禁止事件，其接收参数为事件对象和事件标志。其中标志参数为 **EV_READ** 和 **EV_WRITE**。

```
void accept_handle(const int sfd, const short event, void *arg)
{

  struct sockaddr_in addr;

  socklen_t addrlen = sizeof(addr);

  int fd = accept(sfd, (struct sockaddr *) &addr, &addrlen); //处理连接

  buf_ev = bufferevent_new(fd, buffered_on_read, NULL, NULL, fd)
                                  读回调
  buf_ev->wm_read.high = 4096

  bufferevent_base_set(main_base, buf_ev);  将buffer event加入到事件队列中

  bufferevent_enable(buf_ev, EV_READ);

}
```

此处为了防止read/write调用的阻塞特性，使用了buffer event机制，将新的fd加入到缓冲区队列中，由libevent接管，epoll检测到可读时，先由libevent的单独线程将数据读入缓冲区，然后再回调用户的读函数。写过程也是一样的，先写入libevent提供的缓冲区，再由epoll检测到fd可写时，在线程中调用write阻塞写入。

5 读取缓冲区

当缓冲区读就绪时会自动激活前面注册的缓冲区读函数，我们可以使用 **bufferevent_read** 函数来读取缓冲区

bufferevent_read 函数参数分别为:所需读取的事件缓冲区，读入数据的存放地，希望读取的字节数。函数返回实际读取的字节数。

注意：及时缓冲区未读完，事件也不会再次被激活（除非再次有数据）。因此此处需反复读取直到全部读取完毕。 反复读取，直到缓冲区无数据！

6 写回客户端

bufferevent 系列函数不但支持读取缓冲区，而且支持写缓冲区(即将结果返回给客户端)。

```
void buffered_on_read(struct bufferevent *bev, void * arg){

   char buffer[4096]

    ret = bufferevent_read(bev, &buffer, 4096);

    bufferevent_write(bef, (void *)&buffer, 4096);

}
```

三结束语

至此我们已经可以使用 **libevent** 编写非阻塞的事件驱动服务器，它支持连接建立、**socket** 可读等事件的处理。

但在实际的使用事件驱动的服务器中，通常是使用一个线程处理连接，然后使用多个线程来处理请求。后面我将继续介绍如何使用 **libevent** 来编写多线程的服务器。