# 指定超时事件：

struct event \*m\_evtimeout;

struct *timeval* m\_timeout;

*event\_base* \*m\_pBase;

m\_timeout.*tv\_sec* = s\_iTimeOut; //超时

m\_timeout.*tv\_usec* = 0;

m\_pBase = *event\_base\_new*();

if (!m\_pBase)

{

*printf*("Couldn't open event base!\n");

*exit*(1);

}

//设定超时

m\_evtimeout = *evtimer\_new*(m\_pBase, timeoutcb, m\_pBase);

*evtimer\_add*(m\_evtimeout, &m\_timeout);

\*\*@brief: 超时回调函数。

\*/

voidtimeoutcb(*evutil\_socket\_t* fd, short events, void \*arg)

{

struct *event\_base* \*base = (*event\_base*\*)arg;

*printf*("timeout...\n");

*event\_base\_loopexit*(base, *NULL*);

}

# 指定事件：

struct *event\_base* \* thread\_base; // 事件根基

struct event notify\_event;

*evutil\_socket\_t* notfiy\_recv\_fd; // socketpair 接收端fd（工作线程接收通知）

ret = *event\_assign*(notify\_event, thread\_base, notfiy\_recv\_fd, *EV\_READ* | *EV\_PERSIST*, notify\_cb, plt);

//添加事件

ret = *event\_add*(&plt->notify\_event, 0);

# 设定可互相通信的两个socket（管道）：

//创建一个socketpair， 可互相通信的两个socket，保存在fds里面

*evutil\_socket\_t* fds[2];

if (*evutil\_socketpair*(*AF\_INET*, *SOCK\_STREAM*, 0, fds) < 0)

{

*perror*( "创建socketpair失败\n");

return false;

}

//设置成无阻赛的socket

*evutil\_make\_socket\_nonblocking*(fds[0]);

*evutil\_make\_socket\_nonblocking*(fds[1]);

//激活读线程的读事件

*send*(notfiy\_send\_fd, "c", 1, 0);

# LibEvent多线程工作的设计方案：

## 主要设计思路：

**将连接请求及连接后的读写处理分布到多个线程中去。监听线程收到连接请求之后把这个socket丢给工作线程，工作线程内部创建bufferevent， 处理相关读和写事件。这个过程是通过每个线程的连接队列，和一个socketpair进行通知的。这样每个线程就很平均的处理所有的连接事件。**

1. 设定多个工作线程，每个工作线程都维护一个连接队列，处理多个用户连接。
2. 每个工作线程有自己的事件基*event\_base；*每个连接都有自己的bufferevent。每个线程启动时，就开启基于它的event\_base事件循环；每个连接的bufferevent用于注册它自己的事件处理回调函数。
3. 每个工作线程内部还要生成一对socketpair， 得到两个socket fd，用于监听线程激活工作线程的通知事件用。
4. 每个工作线程还有一个通知事件event，并绑定一个对该事件进行处理的通知回调函数，处理具体的连接后的逻辑，比如创建每个连接的连接socket，为每个连接绑定它的读写回调函数。

## 在初始化函数里

1. 创建线程池

for (int i = 0; i < thread\_numb; ++i)

{

LibeventThread\* plt = new LibeventThread();

//创建互相通信的两个socket

*evutil\_socket\_t* fds[2];

if (*evutil\_socketpair*(*AF\_INET*, *SOCK\_STREAM*, 0, fds) < 0)

{

*perror*( "创建socketpair失败\n");

return false;

}

//设置成无阻赛的socket

*evutil\_make\_socket\_nonblocking*(fds[0]);

*evutil\_make\_socket\_nonblocking*(fds[1]);

plt->notfiy\_recv\_fd = fds[0];

plt->notfiy\_send\_fd = fds[1];

m\_libevent\_threads[i] = plt;

}

1. 创建并启动工作线程

for (int i = 0; i < thread\_numb; ++i)

{

m\_libevent\_threads[i]->spThread.*reset*(new *std*::*thread*([]

(void\* arg)

{

auto me = (LibeventThread\*)arg;

*event\_base\_loop*(me->thread\_base, 0);

}, m\_libevent\_threads[i]));

}

## 在主线程中：

1. 监听所有的用户连接，

当一个连接来到时，会得到一个连接fd：

通过下面代码中的监听回调取得连接fd:

m\_listener = *evconnlistener\_new\_bind*(m\_base, listener\_cb, (void\*)this,

*LEV\_OPT\_REUSEABLE* | *LEV\_OPT\_CLOSE\_ON\_FREE*, -1, (struct *sockaddr*\*)&sin, sizeof(sin));

1. 启动主线程事件循环

*printf*("Server running.\n");

*event\_base\_dispatch*(m\_base);

if (*WSAENOTSOCK* == *WSAGetLastError*())

{

*fprintf*(*stderr*, "操作无效套接字啊！");

}

*fprintf*(*stderr*, "Libevent派发线程退出！");

*event\_base\_free*(m\_base);

m\_base = *NULL*;

*evconnlistener\_free*(m\_listener);

*printf*("Server shutdown.\n");

## 在监听回调listener\_cb中：

1. 轮询线程池，分配一个线程给这个连接。
2. 将当前的连接fd 放入连接队列中
3. 向当前线程用于发送的socket fd发送一个字节的通知消息，用于激活当前线程的通知事件，从而启动当前线程的通知回调函数。

在通知回调函数中：

1. 首先将socketpair的1个字节通知信号读出
2. 从连接队列中取出一个连接fd。
3. 创建每个连接的bufferevent

它是通过调用下面的代码获得的。

//创建每个socket的bufferevent

auto bev = *bufferevent\_socket\_new*(plt->thread\_base, item.fd, *BEV\_OPT\_THREADSAFE*);

1. 设定当前连接的读写回调函数， 以及传给回调的参数

*bufferevent\_setcb*(bev, conn\_readcb, *NULL*, conn\_eventcb, c2);

*bufferevent\_enable*(bev, *EV\_READ* | *EV\_WRITE*);

## 读写回调函数中处理具体的收发逻辑

# Libevent函数使用

struct *evbuffer* \*input = *bufferevent\_get\_input*(bev);

sz=*evbuffer\_get\_length*(input);

*bufferevent\_read*(bev,buf,sz);

struct *evbuffer* \*output= *bufferevent\_get\_output*(bev);

*bufferevent\_free*(bev);

struct *event\_config* \*cfg = *event\_config\_new*();

if (cfg)

{

if(-1==*event\_config\_set\_num\_cpus\_hint*(cfg,4))return -1;

base = *event\_base\_new\_with\_config*(cfg);

*event\_config\_free*(cfg);

}

struct bufferq {

*u\_char* \*buf;

int len;

int offset;

*TAILQ\_ENTRY*(bufferq) entries;

};

struct client {

struct event ev\_read;

struct event ev\_write;

*TAILQ\_HEAD*(, bufferq) writeq;

};

len = *read*(fd, buf, BUFLEN);

if (len == 0)

{

/\* 客户端断开连接，在这里移除读事件并释放客户数据结构。 \*/

*printf*("Client disconnected./n");

close(fd);

*event\_del*(&client->ev\_read);

*free*(client);

return;

}

# Bufferevent锁

*bufferevent\_lock*(m\_bev);

*bufferevent\_unlock*(m\_bev);

# Bufferevent设置超时

struct *timeval* tv = { 1,0 };

struct *timeval* tv1 = { 1,0 };

*bufferevent\_set\_timeouts*(bev, &tv, &tv1);

# Bufferevent支持多线程

int ret = *bufferevent\_enable*(bev, *BEV\_OPT\_THREADSAFE*);

if (ret < 0) { *printf*("----------------------------\n"); }

[说明]：因为是异步读写，bufferevent\_write将写入事件丢进工作线程的事件列表中可能存在此主线程在调用bufferevent\_free的时候，将套接字关闭（即：主线程立马关闭套接字），而下一瞬间工作线程从激活队列中取出此处的写入事件发给客户端时，这个时候，发现此套接字无效，select的时候发现对无效套接字进行操作，立刻报错！所以需要保证操作的套接字有效

# 标准库锁的使用

1. *std*::*lock\_guard*<*std*::*mutex*> lock(channel\_mtx);

*2． std*::*unique\_lock*<*std*::*mutex*> lock(channel\_mtx);

*3.超时锁*

*std*::*timed\_mutex* timer\_mutex;

timer\_mutex.*try\_lock\_for*(*std*::*chrono*::*milliseconds*(10));

… … code … ..

timer\_mutex.*unlock*();