在linux的网络编程中，很长的时间都在使用select来做事件触发。在linux新的内核中，有了一种替换它的机制，就是epoll。

相比于select，epoll最大的好处在于它不会随着监听fd数目的增长而降低效率。因为在内核中的select实现中，它是采用轮询来处理的，轮询的fd数目越多，自然耗时越多。并且，在linux/posix\_types.h头文件有这样的声明：

#define \_\_FD\_SETSIZE 1024

表示select最多同时监听1024个fd，当然，可以通过修改头文件再重编译内核来扩大这个数目，但这似乎并不治本。

**epoll的接口非常简单，一共就三个函数：**

**1. int epoll\_create(int size);**

创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大。这个参数不同于select()中的第一个参数，给出最大监听的fd+1的值。需要注意的是，当创建好epoll句柄后，它就是会占用一个fd值，在linux下如果查看/proc/进程id/fd/，是能够看到这个fd的，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

**2. int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);**

epoll的事件注册函数，它不同与select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。

**第一个**参数是epoll\_create()的返回值，

**第二个**参数表示动作，用三个宏来表示：

EPOLL\_CTL\_ADD：注册新的fd到epfd中；

EPOLL\_CTL\_MOD：修改已经注册的fd的监听事件；

EPOLL\_CTL\_DEL：从epfd中删除一个fd；

**第三个**参数是需要监听的fd，

**第四个**参数是告诉内核需要监听什么事，struct epoll\_event结构如下：

struct epoll\_event {

\_\_uint32\_t events; /\* Epoll events \*/

epoll\_data\_t data; /\* User data variable \*/

};

events可以是以下几个宏的集合：

EPOLLIN ： 表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；

EPOLLOUT： 表示对应的文件描述符可以写；

EPOLLPRI： 表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；

EPOLLERR： 表示对应的文件描述符发生错误；

EPOLLHUP： 表示对应的文件描述符被挂断；

EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。

EPOLLONESHOT： 只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

**3. int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);**

等待事件的产生，类似于select()调用。参数events用来从内核得到事件的集合，maxevents告之内核这个events有多大，这个maxevents的值不能大于创建epoll\_create()时的size，参数timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表示已超时。

从man手册中，得到ET和LT的具体描述如下

**EPOLL事件有两种模型：**

**Edge Triggered (ET) 边缘触发 只有数据到来，才触发，不管缓存区中是否还有数据。**

**Level Triggered (LT) 水平触发 只要有数据都会触发。**

假如有这样一个例子：

1. 我们已经把一个用来从管道中读取数据的文件句柄(RFD)添加到epoll描述符

2. 这个时候从管道的另一端被写入了2KB的数据

3. 调用epoll\_wait(2)，并且它会返回RFD，说明它已经准备好读取操作

4. 然后我们读取了1KB的数据

5. 调用epoll\_wait(2)......

**Edge Triggered 工作模式：**

如果我们在第1步将RFD添加到epoll描述符的时候使用了EPOLLET标志，那么在第5步调用epoll\_wait(2)之后将有可能会挂起，因为剩余的数据还存在于文件的输入缓冲区内，而且数据发出端还在等待一个针对已经发出数据的反馈信息。只有在监视的文件句柄上发生了某个事件的时候 ET 工作模式才会汇报事件。因此在第5步的时候，调用者可能会放弃等待仍在存在于文件输入缓冲区内的剩余数据。在上面的例子中，会有一个事件产生在RFD句柄上，因为在第2步执行了一个写操作，然后，事件将会在第3步被销毁。因为第4步的读取操作没有读空文件输入缓冲区内的数据，因此我们在第5步调用 epoll\_wait(2)完成后，是否挂起是不确定的。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。最好以下面的方式调用ET模式的epoll接口，在后面会介绍避免可能的缺陷。

i 基于非阻塞文件句柄

ii 只有当read(2)或者write(2)返回EAGAIN时才需要挂起，等待。但这并不是说每次read()时都需要循环读，直到读到产生一个EAGAIN才认为此次事件处理完成，当read()返回的读到的数据长度小于请求的数据长度时，就可以确定此时缓冲中已没有数据了，也就可以认为此事读事件已处理完成。

**Level Triggered 工作模式**

相反的，以LT方式调用epoll接口的时候，它就相当于一个速度比较快的poll(2)，并且无论后面的数据是否被使用，因此他们具有同样的职能。因为即使使用ET模式的epoll，在收到多个chunk的数据的时候仍然会产生多个事件。调用者可以设定EPOLLONESHOT标志，在 epoll\_wait(2)收到事件后epoll会与事件关联的文件句柄从epoll描述符中禁止掉。因此当EPOLLONESHOT设定后，使用带有 EPOLL\_CTL\_MOD标志的epoll\_ctl(2)处理文件句柄就成为调用者必须作的事情。

**然后详细解释ET, LT:**

**LT(level triggered)是缺省的工作方式**，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表．

**ET(edge-triggered)是高速工作方式**，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了(比如，你在发送，接收或者接收请求，或者发送接收的数据少于一定量时导致了一个EWOULDBLOCK 错误）。但是请注意，如果一直不对这个fd作IO操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once),不过在TCP协议中，ET模式的加速效用仍需要更多的benchmark确认（这句话不理解）。

在许多测试中我们会看到如果没有大量的idle -connection或者dead-connection，epoll的效率并不会比select/poll高很多，但是当我们遇到大量的idle- connection(例如WAN环境中存在大量的慢速连接)，就会发现epoll的效率大大高于select/poll。（未测试）

另外，当使用epoll的ET模型来工作时，当产生了一个EPOLLIN事件后，

读数据的时候需要考虑的是当recv()返回的大小如果等于请求的大小，那么很有可能是缓冲区还有数据未读完，也意味着该次事件还没有处理完，所以还需要再次读取：

while(rs)

{

buflen = recv(activeevents[i].data.fd, buf, sizeof(buf), 0);

if(buflen < 0)

{

// 由于是非阻塞的模式,所以当errno为EAGAIN时,表示当前缓冲区已无数据可读

// 在这里就当作是该次事件已处理处.

if(errno == EAGAIN)

break;

else

return;

}

else if(buflen == 0)

{

// 这里表示对端的socket已正常关闭.

}

if(buflen == sizeof(buf)

rs = 1; // 需要再次读取

else

rs = 0;

}

还有，假如发送端流量大于接收端的流量(意思是epoll所在的程序读比转发的socket要快),由于是非阻塞的socket,那么send()函数虽然返回,但实际缓冲区的数据并未真正发给接收端,这样不断的读和发，当缓冲区满后会产生EAGAIN错误(参考man send),同时,不理会这次请求发送的数据.所以,需要封装socket\_send()的函数用来处理这种情况,该函数会尽量将数据写完再返回，返回-1表示出错。在socket\_send()内部,当写缓冲已满(send()返回-1,且errno为EAGAIN),那么会等待后再重试.这种方式并不很完美,在理论上可能会长时间的阻塞在socket\_send()内部,但暂没有更好的办法.

ssize\_t socket\_send(int sockfd, const char\* buffer, size\_t buflen)

{

ssize\_t tmp;

size\_t total = buflen;

const char \*p = buffer;

while(1)

{

tmp = send(sockfd, p, total, 0);

if(tmp < 0)

{

// 当send收到信号时,可以继续写,但这里返回-1.

if(errno == EINTR)

return -1;

// 当socket是非阻塞时,如返回此错误,表示写缓冲队列已满,

// 在这里做延时后再重试.

if(errno == EAGAIN)

{

usleep(1000);

continue;

}

return -1;

}

if((size\_t)tmp == total)

return buflen;

total -= tmp;

p += tmp;

}

return tmp;

}

**代码：**

#include <iostream>

#include <sys/socket.h>

#include <sys/epoll.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

#include <pthread.h>

#include <errno.h>

#define MAXLINE 10

#define OPEN\_MAX 100

#define LISTENQ 20

#define SERV\_PORT 5555

#define INFTIM 1000

//线程池任务队列结构体

struct task

{

int fd; //需要读写的文件描述符

struct task \*next; //下一个任务

};

//用于读写两个的两个方面传递参数

struct user\_data

{

int fd;

unsigned int n\_size;

char line[MAXLINE];

};

//线程的任务函数

void \* readtask(void \*args);

void \* writetask(void \*args);

//声明epoll\_event结构体的变量,ev用于注册事件,数组用于回传要处理的事件

struct epoll\_event ev, events[20];

int epfd;

pthread\_mutex\_t mutex;

pthread\_cond\_t cond1;

struct task \*readhead = NULL, \*readtail = NULL, \*writehead = NULL;

void setnonblocking(int sock)

{

int opts;

opts = fcntl(sock, F\_GETFL);

if (opts < 0)

{

perror("fcntl(sock,GETFL)");

exit(1);

}

opts = opts | O\_NONBLOCK;

if (fcntl(sock, F\_SETFL, opts) < 0)

{

perror("fcntl(sock,SETFL,opts)");

exit(1);

}

}

int main()

{

int i, maxi, listenfd, connfd, sockfd, nfds;

pthread\_t tid1, tid2;

struct task \*new\_task = NULL;

struct user\_data \*rdata = NULL;

socklen\_t clilen;

pthread\_mutex\_init(&mutex, NULL);

pthread\_cond\_init(&cond1, NULL);

//初始化用于读线程池的线程

pthread\_create(&tid1, NULL, readtask, NULL);

pthread\_create(&tid2, NULL, readtask, NULL);

//生成用于处理accept的epoll专用的文件描述符

epfd = epoll\_create(256);

struct sockaddr\_in clientaddr;

struct sockaddr\_in serveraddr;

listenfd = socket(AF\_INET, SOCK\_STREAM, 0);

//把socket设置为非阻塞方式

setnonblocking(listenfd);

//设置与要处理的事件相关的文件描述符

ev.data.fd = listenfd;

//设置要处理的事件类型

ev.events = EPOLLIN | EPOLLET;

//注册epoll事件

epoll\_ctl(epfd, EPOLL\_CTL\_ADD, listenfd, &ev);

bzero(&serveraddr, sizeof(serveraddr));

serveraddr.sin\_family = AF\_INET;

char \*local\_addr = "200.200.200.222";

inet\_aton(local\_addr, &(serveraddr.sin\_addr));//htons(SERV\_PORT);

serveraddr.sin\_port = htons(SERV\_PORT);

bind(listenfd, (sockaddr \*) &serveraddr, sizeof(serveraddr));

listen(listenfd, LISTENQ);

maxi = 0;

for (;;)

{

//等待epoll事件的发生

nfds = epoll\_wait(epfd, events, 20, 500);

//处理所发生的所有事件

for (i = 0; i < nfds; ++i)

{

if (events[i].data.fd == listenfd)

{

connfd = accept(listenfd, (sockaddr \*) &clientaddr, &clilen);

if (connfd < 0)

{

perror("connfd<0");

exit(1);

}

setnonblocking(connfd);

char \*str = inet\_ntoa(clientaddr.sin\_addr);

std::cout << "connec\_ from >>" << str << std::endl;

//设置用于读操作的文件描述符

ev.data.fd = connfd;

//设置用于注测的读操作事件

ev.events = EPOLLIN | EPOLLET;

//注册ev

epoll\_ctl(epfd, EPOLL\_CTL\_ADD, connfd, &ev);

} else

if (events[i].events & EPOLLIN)

{

printf("reading!\n");

if ((sockfd = events[i].data.fd) < 0) continue;

new\_task = new task();

new\_task->fd = sockfd;

new\_task->next = NULL;

//添加新的读任务

pthread\_mutex\_lock(&mutex);

if (readhead == NULL)

{

readhead = new\_task;

readtail = new\_task;

} else

{

readtail->next = new\_task;

readtail = new\_task;

}

//唤醒所有等待cond1条件的线程

pthread\_cond\_broadcast(&cond1);

pthread\_mutex\_unlock(&mutex);

} else

if (events[i].events & EPOLLOUT)

{

rdata = (struct user\_data \*) events[i].data.ptr;

sockfd = rdata->fd;

write(sockfd, rdata->line, rdata->n\_size);

delete rdata;

//设置用于读操作的文件描述符

ev.data.fd = sockfd;

//设置用于注测的读操作事件

ev.events = EPOLLIN | EPOLLET;

//修改sockfd上要处理的事件为EPOLIN

epoll\_ctl(epfd, EPOLL\_CTL\_MOD, sockfd, &ev);

}

}

}

}

void \* readtask(void \*args)

{

int fd = -1;

unsigned int n;

//用于把读出来的数据传递出去

struct user\_data \*data = NULL;

while (1)

{

pthread\_mutex\_lock(&mutex);

//等待到任务队列不为空

while (readhead == NULL)

pthread\_cond\_wait(&cond1, &mutex);

fd = readhead->fd;

//从任务队列取出一个读任务

struct task \*tmp = readhead;

readhead = readhead->next;

delete tmp;

pthread\_mutex\_unlock(&mutex);

data = new user\_data();

data->fd = fd;

if ((n = read(fd, data->line, MAXLINE)) < 0)

{

if (errno == ECONNRESET)

{

close(fd);

} else

std::cout << "readline error" << std::endl;

if (data != NULL) delete data;

} else

if (n == 0)

{

close(fd);

printf("Client close connect!\n");

if (data != NULL) delete data;

} else

{

data->n\_size = n;

//设置需要传递出去的数据

ev.data.ptr = data;

//设置用于注测的写操作事件

ev.events = EPOLLOUT | EPOLLET;

//修改sockfd上要处理的事件为EPOLLOUT

epoll\_ctl(epfd, EPOLL\_CTL\_MOD, fd, &ev);

}

}

}