当一个单线程的服务器既想等待终端标准输入，也想等待若干个套接字连接请求，这个时候就需要使用I/O复用了。**select，poll，epoll都是IO多路复用的机制**。I/O多路复用就是通过一种机制，一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，poll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的；而epoll是异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

**1、select**

**1.1 函数原型**

#include <sys/select.h>

#include <sys/time.sys>

int **select** (int maxfdp1, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, ~~const~~ **struct** timeval \***timeout**);

//返回:若有就绪描述符为其数目，若超时则为0，出错则为-1

参数：

maxfdp1: 最大的描述符值+1

fd\_set \*readfds： 需要内核检测为可读的描述符

fd\_set \*writefds： 需要内核检测为可写的描述符

d\_set \*exceptfds： 需要内核检测为出错的描述符

**struct** timeval \***timeout**：超时时间

maxfdp1为轮询设定的边界，例如检测0,1,6描述符，maxfdp1为7，轮询实现for(i=0;i<7;i++)。也可以直接设置为当前系统支持的描述符最大值，但是这样会造成过多不必要的select轮询。

调用select函数时，需要指定需要检测的描述符对应位置置为1。当select返回时，描述符集合将被修改以指示哪些个描述符正处于可读、可写或有错误状态，使用FD\_ISSET宏对描述符进行测试找到状态变化的描述符，任何未就绪描述符对应位被清位0。如果select因为超时而返回的话，所有的描述符集合都将被清空。

（1）struct timeval

struct timeval{

\_\_time\_t tv\_sec; /\* Seconds. \*/

\_\_suseconds\_t tv\_usec; /\* Microseconds. \*/

};

select函数的timeval \*timeout参数三种情况

1. NULL select函数一直阻塞到有可读、可写或出错的描述符就绪
2. timeout的时间设置为0，select函数立即返回
3. timeout的时间设置不为零，select函数正常等待是否有描述符就绪，否则超时

（注意：对于linxu系统，参数无const限定，每次select调用后会修改timeout的值为剩余时间，unix系统则不会改变）

（2）fd\_set定义

当前库中相关定义

#define \_\_FD\_SETSIZE 1024

typedef long int \_\_fd\_mask;

#define \_\_NFDBITS (8 \* (int) sizeof (\_\_fd\_mask)) // 64

typedef struct{

\_\_fd\_mask fds\_bits[\_\_FD\_SETSIZE / \_\_NFDBITS]; // long int[16]

} fd\_set;

经计算，fd\_set是一个long int [16]的数组，实际可以将fd\_set看做是一个有1024 bit二进制位的对象，我们可以使用FD\_ZERO、FD\_SET、FD\_CLR和FD\_ISSET对fd\_set的二进制位进行处理。

（3）fd\_set函数

FD\_ZERO(fd\_set \*fdset); /\* 将fdset的所有位清空置为0 \*/

FD\_SET(int fd, fd\_set \*fdset); /\* 将fdset的fd位置为1 \*/

FD\_CLR(int fd, fd\_set \*fdset); /\* 将fdset的fd位置为0 \*/

int FD\_ISSET(int fd, fd\_set \*fdset); /\* 检测fdset的fd位是否为1 \*/

**1.2 select示例**

**调用前，需要将关心的描述符对应位置为1，调用之后使用FD\_ISSET判断所有描述符状态并作出不同处理。调用结束，不论是否超时还是有描述符未就绪，会导致传入的fd\_set中关心的描述符位有关位被清0，所以下一个循环使用时，必须重新置位所有感兴趣的描述符对应位。**

前面的文章中给出了使用多线程的TCP echo服务代码，这里给出基于select的单线程代码：

#include <stdio.h>

#include <sys/socket.h>

#include <arpa/inet.h> // sockaddr\_in, inet\_addr

#include <unistd.h> // close

#include <cstring> // memset, bezro

#include <errno.h>

#include <sys/time.h>

#define LOG(fmt, arg...) \

do { \

struct timeval tv; \

gettimeofday(&tv, NULL); \

printf("[%ld.%03ld] " fmt, tv.tv\_sec, tv.tv\_usec / 1000, ##arg); \

} while (0)

const char \*SRV\_ADDR = "127.0.0.1";

const int SRV\_PORT = 8080;

int main()

{

int socket\_fd = socket(AF\_INET, SOCK\_STREAM, 0); // tcp

if (socket\_fd == -1){

LOG("%s: create socket failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}else{

LOG("%s: create socket (fd = %d) success.\n", \_\_func\_\_, socket\_fd);

}

/// 2、绑定到本地端口

sockaddr\_in servaddr;

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = inet\_addr(SRV\_ADDR);

servaddr.sin\_port = htons(SRV\_PORT);

int ret = ::bind(socket\_fd, (const sockaddr \*)&servaddr, sizeof(servaddr));

if (ret == -1){

LOG("%s: bind %s:%d failed. %s\n", \_\_func\_\_, SRV\_ADDR, SRV\_PORT, strerror(errno));

return 1;

}else{

LOG("%s: bind %s:%d success.\n", \_\_func\_\_, SRV\_ADDR, SRV\_PORT);

}

/// 3、监听

ret = ::listen(socket\_fd, 5);

if (ret == -1){

LOG("%s: listen failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}else{

LOG("%s: listening ...\n", \_\_func\_\_);

}

// select的参数

int max\_fd = socket\_fd;

fd\_set fd\_A; // 监听fd、客户fd

fd\_set work\_fd\_A; // 待检测的fd，用于select调用，每次调用都会被改变

FD\_ZERO(&fd\_A);

FD\_SET(socket\_fd, &fd\_A); // 添加监听fd

timeval timeout;

int len;

char buf[1024];

while (true)

{

FD\_ZERO(&work\_fd\_A);

memcpy(&work\_fd\_A, &fd\_A, sizeof(fd\_A));

// linux 为非const指针，调用之后会修改为剩下的时间，因此需要每次赋值

timeout.tv\_sec = 30;

timeout.tv\_usec = 0;

ret = select(max\_fd + 1, &work\_fd\_A, NULL, NULL, &timeout);

if (ret == -1){ // 出错

LOG("%s: select failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}else if (ret == 0){ // 超时

LOG("%s: select timeout.\n", \_\_func\_\_);

continue;

}

if (FD\_ISSET(socket\_fd, &work\_fd\_A)) // 新的客户端连接

{

sockaddr\_storage clientaddr;

socklen\_t sock\_len = sizeof(clientaddr);

int sock\_id = ::accept(socket\_fd, (sockaddr \*)&clientaddr, &sock\_len);

if (sock\_id < 0){

LOG("%s: accept error. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

char client\_ip[INET6\_ADDRSTRLEN];

inet\_ntop(AF\_INET, &clientaddr.sin\_addr, client\_ip, sock\_len);

int port = ntohs(clientaddr.sin\_port);

LOG("%s: new socket id = %d, clent %s:%d\n", \_\_func\_\_, sock\_id, client\_ip, port);

// 将新建立连接的客户端 fd加入fdset

FD\_SET(sock\_id, &fd\_A);

if (sock\_id > max\_fd) max\_fd = sock\_id;

}

else // 其他，描述符状态

{

// 遍历查找已就绪的套接字

// 0-stdin, 1-stdout, 2-stderr

for (int fd = 0; fd <= max\_fd; ++fd)

{

if (FD\_ISSET(fd, &work\_fd\_A))

{

len = read(fd, &buf, sizeof(buf));

if (len < 0){

LOG("%s: recv failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

else if (len == 0){ // 对端关闭连接

LOG("%s: client socket %d disconnet. recv len = 0. \n", \_\_func\_\_, fd);

// 从fdset中移除

::close(fd);

FD\_CLR(fd, &fd\_A);

// 例如开始有3,4,5,6,7 （max\_fd = 7 ）

// 后来5,6关闭，剩下 3,4,7 （max\_fd = 7 ）

// 当7关闭，剩下3,4 max\_fd应该减小为4

if (fd == max\_fd){

while (FD\_ISSET(max\_fd, &fd\_A) == false) --max\_fd;

}

}else{

buf[len] = 0;

LOG("%s: recv from socket %d (%d): %s", \_\_func\_\_, fd, len, buf);

// echo

len = ::write(fd, buf, strlen(buf));

if (len < 0){

printf("%s: send failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

}

}

}

}

}

::close(socket\_fd);

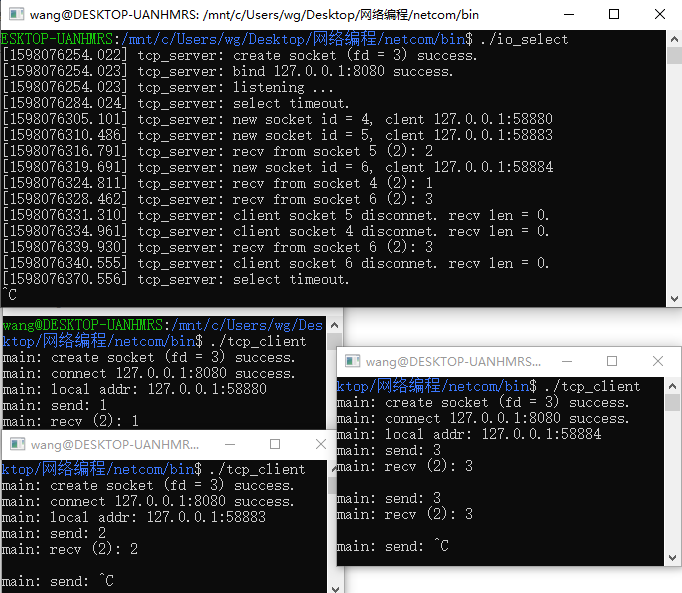
return 0;

}

**1.3 演示结果说明**

运行新的服务端，设置超时时间30s、关心的描述符，调用select函数()，等到30s中无任何关心的描述符就绪，重新设置进行超时时间、关心的描述符，再进行select调用。当关心的描述符就绪时，select函数会立即返回。

描述符集合最早只有服务端的监听描述符3，后来加入客户端对应的描述符4,5,6，此时描述符集合[3,4,5,6]，传递给select函数的maxfdp1=7；当前两个客户端关闭连接，描述符集合只有[3,6]，但是maxfdp1的值仍然为7。只要最大的描述符对应客户端保持连接，那么不论中间的客户端有多少，select进行轮询的次数不会发生改变。



**1.4 优缺点**

相对于阻塞式IO模型，使用select的优点是可以在单线程中同时处理多个描述符进行IO复用，占用资源少。

select的缺点：

**maxfd有最大值限制**\_\_FD\_SETSIZE，即最多能监听1024个描述符（可以修改该内核宏并重新编译内核，但一般不这么做）

**maxfdp1的值较大**而需要检测的描述符个数少，**select函数和用户执行FD\_ISSET函数会造成大量的无效轮询**。若并发使用时一段时间前面的连接断开，最后一个保持连接，例如描述符集合为[3,4,5,800]，max\_fd=801，实际只有4个关心的描述符，但是select函数会在内核中会轮询801次，并且用户也需要执行FD\_ISSET函数801次。

**2、poll**

poll和select同时期出现，目前主要是因为使用习惯、跨平台的限制原因，select要用的多。

poll要解决的问题和使用方式上与select保持一致，只不过**监听对象从3个fd\_set集合对象换成了一个pollfd结构数组，从而没有监听套接字的数量限制**(实际也是有限的，见poll函数说明)。

**2.1 函数原型**

#include <sys/poll.h>

int **poll**(struct pollfd \*fdarry, unsigned long int nfds, int timeout);

//返回:若有就绪描述符为其数目，若超时则为0，出错则为-1

参数介绍：

fds : 对应上述介绍的结构体指针

nfds : 标记数组中结构体元素的总个数。

timeout :超时时间(ms)，<0表示阻塞式等待，=0表示非阻塞式等待，>0表示等待时间

我们说poll模式监听的fd是没有限制的，实际也是有限的，例如64位系统nfds为8字节，最大值为2^64-1，远大于select模式监听最大值\_\_FD\_SETSIZE即1024/2048，因此说其无限制。

**(1) struct pollfd**

struct pollfd{

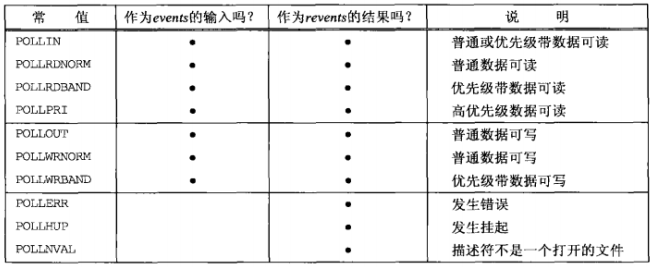
int fd; /\* 需要内核检测的描述符\*/

short int events; /\* 描述符上要监听的事件或事件集合\*/

short int revents; /\* 描述符上检测到的事件\*/

};

参数fd为负值（如-1）时，poll函数会忽略该pollfd，不进行对应evenets的检测。参数events和revents的标志取值和说明如下



注意最后一组标志，poll需要检测错误异常不需要像select显示请求指定，poll会在合适的时候从revents中返回。

events取值可以上述读写标志中的单个值，也可以是多个值的组合。POLLIN | POLLPRI等价于select()的读事件，POLLOUT | POLLWRBAND等价于select()的写事件。POLLIN等价于POLLRDNORM | POLLRDBAND，而POLLOUT则等价于POLLWRNORM。

这里，说明一下**poll返回revents的值的的定义情况**：

正规的TCP、UDP数据都认为是普通数据

TCP的带外数据是优先级带数据

TCP连接的读半关闭（shutdown读，收到对端FIN字节），是普通数据，读操作返回0

TCP连接错误可以认为是普通数据，也可以是错误（POLLERR）。随后的读操作返回-1，并把errno设置为合适的值。用于处理诸如接收到RST或发生超时等条件。

监听套接字有新的连接，多数实现认为是普通数据，也可以认为是优先级数据。

非阻塞connect的完成认为是使响应套接字可写

**2.2 poll示例**

创建一个struct pollfd数组，将期望检测的描述符pollfd.fd及其事件pollfd.events保存在结构中，设置超时时间调用poll()，返回后检测数组中就绪描述符检测到的事件并做对应处理。

仍然以TCP echo为例，将select示例代码main函数修改如下

int main()

{

// 1、创建socket

int socket\_fd = socket(AF\_INET, SOCK\_STREAM, 0); // tcp

if (socket\_fd == -1){

LOG("%s: create socket failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}else{

LOG("%s: create socket (fd = %d) success.\n", \_\_func\_\_, socket\_fd);

}

/// 2、绑定到本地端口

sockaddr\_in servaddr;

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = inet\_addr(SRV\_ADDR);

servaddr.sin\_port = htons(SRV\_PORT);

int ret = ::bind(socket\_fd, (const sockaddr \*)&servaddr, sizeof(servaddr));

if (ret == -1){

LOG("%s: bind %s:%d failed. %s\n", \_\_func\_\_, SRV\_ADDR, SRV\_PORT, strerror(errno));

return 1;

}else{

LOG("%s: bind %s:%d success.\n", \_\_func\_\_, SRV\_ADDR, SRV\_PORT);

}

/// 3、监听

ret = ::listen(socket\_fd, 2048);

if (ret == -1){

LOG("%s: listen failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}else{

LOG("%s: listening ...\n", \_\_func\_\_);

}

/// poll的参数

struct pollfd fdarray[2048];

// 添加监听

fdarray[0].fd = socket\_fd;

fdarray[0].events = POLLIN|POLLPRI; // 同select读

// 初始化其他结构对象

for(int i=1; i<2048; ++i)

fdarray[i].fd = -1; // fd为负值的pollfd结构会被poll忽略

int maxi = 0; // fdarray数组中正在使用最大下标值

int timeout = 30000; // ms

int nready; // 就绪的描述符，poll返回值

int len;

char buf[1024];

while (true)

{

nready = ::poll(fdarray, maxi + 1, timeout);

if (nready == -1){ // 出错

LOG("%s: poll failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

else if (nready == 0){ // 超时

LOG("%s: poll timeout.\n", \_\_func\_\_);

continue;

}

if (fdarray[0].revents & (POLLIN|POLLPRI)) // 新的客户端连接

{

sockaddr\_in clientaddr;

socklen\_t sock\_len = sizeof(clientaddr);

int sock\_id = ::accept(socket\_fd, (sockaddr \*)&clientaddr, &sock\_len);

if (sock\_id < 0){

LOG("%s: accept error. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

char client\_ip[INET6\_ADDRSTRLEN];

inet\_ntop(AF\_INET, &clientaddr.sin\_addr, client\_ip, sock\_len);

int port = ntohs(clientaddr.sin\_port);

LOG("%s: new socket id = %d, clent %s:%d\n", \_\_func\_\_, sock\_id, client\_ip, port);

// 将新建立连接的客户端加入fdarray

int i;

for( i = 1; i < 2048; i++)

if(fdarray[i].fd < 0){

fdarray[i].fd = sock\_id;

fdarray[i].events = POLLIN|POLLPRI;

if( i > maxi)

maxi = i;

break;

}

if( i == 2048 ){

LOG("%s: too many clients\n", \_\_func\_\_);

break; // 退出

}

if(--nready <= 0)

continue; // 若当前poll返回为nready为1，避免执行后续的轮询操作

}

// 遍历其他pollfd结构

for (int i = 1; i <= maxi; ++i)

{

int fd = fdarray[i].fd;

if ( fdarray[i].revents & (POLLIN|POLLPRI|POLLERR))

{

len = read(fd, &buf, sizeof(buf));

if (len < 0){

LOG("%s: recv failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

else if (len == 0){ // 对端关闭连接

LOG("%s: client socket %d disconnet. recv len = 0. \n", \_\_func\_\_, fd);

// 从fdarray中移除

::close(fd);

fdarray[i].fd = -1;

if( i == maxi )

while( fdarray[maxi].fd < 0) --maxi;

}else{

buf[len] = 0;

LOG("%s: recv from socket %d (%d): %s", \_\_func\_\_, fd, len, buf);

// echo

len = ::write(fd, buf, strlen(buf));

if (len < 0){

printf("%s: send failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

}

if(--nready <= 0)

break; // 根据poll返回值和已处理描述符次数降低轮询次数

}

}

}

::close(socket\_fd);

return 0;

}

演示结果完全同2.3中select实现的TCP echo服务端。

**2.3 select、poll对比总结**

上述通过select和poll实现的Tcp echo服务端使用方式类似：

（1）都需先创建一个指定感兴趣的描述符和检测状态的对象

select 使用的是 fe\_set；poll使用的是struct pollfd数组

（2）之后都需要调用select、poll阻塞或者非阻塞形式等待就绪描述符

（3）select、poll返回后，需要经过轮询获取就绪的socket并作出相应处理

相对于select受限于最多只能处理最大的描述符个数为\_\_FD\_SETSIZE，poll是没有最大的描述符个数的限制。

**2.4 select、poll使用优化**

select使用了max\_fd记录描述符集合中的标记为检测状态描述符的最大值，poll中记录了struct pollfd数组中标记为检测状态描述符的数组最大下标maxi。为了减少轮询获得就绪描述符，有如下处理：

当有新的客户端连接、客户端断开连接是，都会更新max\_fd、maxi，以减少轮询次数。在poll中使用if(--nready <= 0) continue ; 和 if(--nready <= 0) break ;，进一步减少轮询次数。

进一步地，select内核轮询是固定是fd\_set对象，不能进一步优化，但poll内核轮询对象是struct pollfd数组，在每次调用poll时我们传递数组对象可以仅仅包含需要的描述符，而不是大量无用的描述符为-1的结构对象。

我们可以设计另外一个数据结构，例如 std::set<struct pollfd>或std::unored\_set<struct pollfd>保存仅需要检测的对象，在调用poll时转换成c风格数组（可能有空间消耗和性能的损耗），减少poll内核轮询次数。

**3、epoll**

epoll是linux内核中的一种可扩展的异步IO时间处理机制，用于替代select和poll系统调用，且在具有大量应用程序请求时能够获得更好的性能（select和poll系统调用完成操作的时间复杂度为O(n)，epoll能在O(1)时间内完成），epoll与freeBSD的kqueue类似，都向用户空间提供了描述符文件管理来进行操作

**3.1 函数原型**

epoll使用需要有三个函数调用，都声明在头文件**sys/epoll.h**，先调用**epoll\_create()**创建一个epoll的句柄/描述符并告知内核需要监听描述符的个数，再调用**epoll\_ctl()**管理感描述符（增加、删除、修改），最后循环使用**epoll\_wait()**进行不同事件的派发处理。最后，必须使用**close()**关闭epoll描述符。

（1）**epoll\_create**

int **epoll\_create**(int size); // 返回: 失败返回-1，成功返回epoll描述符

创建一个epoll的句柄，告诉内核需要监听size个数目描述符。当创建好epoll句柄后，它就是会占用一个fd值，在linux下通过/proc/进程id/fd/查看。**在使用完epoll后，必须调用close() 关闭，否则可能导致fd被耗尽**。

（2）**epoll\_ctl**

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

//返回:若成功为0，出错则为-1

/\* 有效的op操作符 ( "op" parameter )\*/

#define EPOLL\_CTL\_ADD 1 /\* 注册新的描述符fd和对应监听事件 \*/

#define EPOLL\_CTL\_DEL 2 /\* 删除一个已注册的fd的监听事件 \*/

#define EPOLL\_CTL\_MOD 3 /\* 修改已经注册的fd的监听事件 \*/

/\* epoll事件枚举值 \*/

enum EPOLL\_EVENTS

{

EPOLLIN = 0x001,

EPOLLPRI = 0x002,

EPOLLOUT = 0x004,

EPOLLRDNORM = 0x040,

EPOLLRDBAND = 0x080,

EPOLLWRNORM = 0x100,

EPOLLWRBAND = 0x200,

EPOLLMSG = 0x400,

EPOLLERR = 0x008,

EPOLLHUP = 0x010,

EPOLLRDHUP = 0x2000,

EPOLLEXCLUSIVE = 1u << 28, //解决惊群现象，保证一个事件只有一个线程处理

EPOLLWAKEUP = 1u << 29,

EPOLLONESHOT = 1u << 30, // 仅监听一次；多线程处理；若还需要再次监听需要重新注册

EPOLLET = 1u << 31 // 边缘触发(Edge Triggered)模式

};

/\* epoll事件数据结构 \*/

typedef union epoll\_data{

void \*ptr;

int fd;

\_\_uint32\_t u32;

\_\_uint64\_t u64;

} epoll\_data\_t;

struct epoll\_event {

\_\_uint32\_t events; /\* epoll事件 \*/

epoll\_data\_t data; /\* 用户数据 \*/

};

参数：

int epfd： epoll\_create调用返回的epoll描述符

int op： 管理动作类型

int fd: 要监听检测的描述符

struct epoll\_event \*event: 监听描述符上的事件

（3）**epoll\_wait**

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

//返回:若有就绪描述符为其数目，若超时则为0，出错则为-1

等待描述符epfd上触发就绪的事件，参数\*events为接收就绪事件数据的结构数组指针，参数maxevents 为当前调用允许接收的最多就绪事件（必须小于**epoll\_create**时传递的size)个数，timeout与poll参数意义一致（小于零的-1阻塞，0不阻塞，正数为阻塞的时间ms）。

注意： epoll\_waite()中参数maxevents的值要小于等于参数\*event指向的数组长度。

**3.2 epoll示例**

epoll使用相对简单，先用**epoll\_create()**创建监听maxevents个事件的epoll描述符，再将服务端监听套集字监听事件通过**epoll\_ctl()**注册到epoll队列。之后循环调用指定参数（返回最大监听事件数量参数、接收一定数量epoll事件的pEvents对象）的**epoll\_wait()**就绪事件，再从就绪符事件pEvents对象中遍历出说有事件，根据描述符和事件类型作出不同处理。

以水平触发为例，修改poll实现是TCP echo服务，代码如下

#include <stdio.h>

#include <sys/socket.h>

#include <arpa/inet.h> // sockaddr\_in, inet\_addr

#include <unistd.h> // close

#include <cstring> // memset, bezro

#include <errno.h>

#include <sys/epoll.h>

#include <sys/time.h>

#define LOG(fmt, arg...) \

do \

{ \

struct timeval tv; \

gettimeofday(&tv, NULL); \

printf("[%ld.%03ld] " fmt, tv.tv\_sec, tv.tv\_usec / 1000, ##arg); \

} while (0)

const char \*SRV\_ADDR = "127.0.0.1";

const int SRV\_PORT = 8080;

int main()

{

// 1、创建socket

int socket\_fd = socket(AF\_INET, SOCK\_STREAM, 0); // tcp

if (socket\_fd == -1){

LOG("%s: create socket failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}else{

LOG("%s: create socket (fd = %d) success.\n", \_\_func\_\_, socket\_fd);

}

/// 2、绑定到本地端口

sockaddr\_in servaddr;

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = inet\_addr(SRV\_ADDR);

servaddr.sin\_port = htons(SRV\_PORT);

int ret = ::bind(socket\_fd, (const sockaddr \*)&servaddr, sizeof(servaddr));

if (ret == -1){

LOG("%s: bind %s:%d failed. %s\n", \_\_func\_\_, SRV\_ADDR, SRV\_PORT, strerror(errno));

return 1;

}else{

LOG("%s: bind %s:%d success.\n", \_\_func\_\_, SRV\_ADDR, SRV\_PORT);

}

/// 3、监听

ret = ::listen(socket\_fd, 2048);

if (ret == -1){

LOG("%s: listen failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}else{

LOG("%s: listening ...\n", \_\_func\_\_);

}

/// epoll操作

// 创建epoll

const int maxevents = 2048;

int epollftd = ::epoll\_create(maxevents);

if(epollftd == -1){

LOG("%s: epoll create faild. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}else{

LOG("%s: epoll create success. (epollfd = %d) \n", \_\_func\_\_, epollftd);

}

// 注册服务端监听

epoll\_event listen\_event;

listen\_event.data.fd = socket\_fd;

listen\_event.events = EPOLLIN|EPOLLPRI; // 同select读, 默认水平触发

ret = epoll\_ctl(epollftd, EPOLL\_CTL\_ADD, listen\_event.data.fd, &listen\_event);

if(ret == -1){

LOG("%s: epoll\_ctrl EPOLL\_CTL\_ADD failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

// 创建接收epoll触发事件的epoll\_event的数组

epoll\_event pEvents[maxevents];

int nready;

int len;

char buf[1024];

while (true)

{

nready = ::epoll\_wait(epollftd, pEvents, maxevents, 30000);

//LOG("%s: epoll\_wait ==================\n", \_\_func\_\_); // 测试maxevents取值不同的情况

if (nready == -1){ // 出错

LOG("%s: epoll\_wait failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

else if (nready == 0){ // 超时

LOG("%s: epoll\_wait timeout.\n", \_\_func\_\_);

continue;

}

for (int i = 0; i < nready; i++)

{

const epoll\_event& eventRef = pEvents[i];

if(eventRef.data.fd == socket\_fd) // 新的客户端连接

{

sockaddr\_in clientaddr;

socklen\_t sock\_len = sizeof(clientaddr);

int sock\_id = ::accept(socket\_fd, (sockaddr \*)&clientaddr, &sock\_len);

if (sock\_id < 0){

LOG("%s: accept error. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

char client\_ip[INET6\_ADDRSTRLEN];

inet\_ntop(AF\_INET, &clientaddr.sin\_addr, client\_ip, sock\_len);

int port = ntohs(clientaddr.sin\_port);

LOG("%s: new socket id = %d, clent %s:%d\n", \_\_func\_\_, sock\_id, client\_ip, port);

// 将新建立连接的客户端加入epoll队列

epoll\_event ev;

ev.events = EPOLLIN | EPOLLPRI;

ev.data.fd = sock\_id;

::epoll\_ctl(epollftd, EPOLL\_CTL\_ADD, ev.data.fd, &ev);

}

else if(eventRef.events & (EPOLLIN|EPOLLPRI|EPOLLERR)) // 读事件

{

len = read(eventRef.data.fd, &buf, sizeof(buf));

if (len < 0){

LOG("%s: recv failed. %s\n", \_\_func\_\_, strerror(errno)); // 出错

return 1;

}

else if (len == 0){ // 对端关闭连接

LOG("%s: client socket %d disconnet. recv len = 0. \n", \_\_func\_\_, eventRef.data.fd);

// 从epoll队列移除

epoll\_event ev;

ev.data.fd = eventRef.data.fd;

ev.events = EPOLLIN | EPOLLPRI;

// 水平触发时，若不处理该事件，会无限接收客户断开连接的事件

::epoll\_ctl(epollftd, EPOLL\_CTL\_DEL, pEvents[i].data.fd, &ev);

::close(eventRef.data.fd);

}else{

buf[len] = 0;

LOG("%s: recv from socket %d (%d): %s", \_\_func\_\_, eventRef.data.fd, len, buf);

// echo

len = ::write(eventRef.data.fd, buf, strlen(buf));

if (len < 0){

printf("%s: send failed. %s\n", \_\_func\_\_, strerror(errno));

return 1;

}

}

}

}

}

::close(socket\_fd); // 注意，这里实际是执行不到的

return 0;

}

演示效果同前面的poll示例截图一样。

在前面提到

epoll\_waite()中参数maxevents的值要小于等于参数\*event指向的数组长度。

代码中指定size = 2048, \*event指向数组长度2048。如果maxevents值小于当前实际事件数的数量，epll\_wait()将先返回maxevents个事件，之后进入再次进入epll\_wait()不断返回剩下的事件，但是每次最多返回maxevents个事件。例如当前事件有5个，

如果设置maxevents为2，那么运行过程是：(1)epll\_wait()返回2个事件，处理事件；(2) epll\_wait()返回2个事件，处理事件；(2) epll\_wait()返回1个事件，处理事件；

如果设置maxevents为3，运行过程是：(1)epll\_wait()返回3个事件，处理事件；(2) epll\_wait()返回2个事件，处理事件；

再如果设置maxevents大于等5，运行过程是：(1)epll\_wait()返回5个事件，处理事件。

**3.2 epoll两种事件模型**

EPOLL事件有两种模型 **Level Triggered (LT)** 和 **Edge Triggered (ET)：**

**LT(level triggered，水平触发模式)**是缺省的工作方式，并且同时支持 block 和 non-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。

**ET(edge-triggered，边缘触发模式)**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，等到下次有新的数据进来的时候才会再次触发就绪事件。

例如，某时间段客户端只发送一段数据，当前套接字读缓冲区有n个字符，read执行一次仅读取1个字符时。在TL模式下，不断触发读事件知道当前缓冲区，一共执行n次；而在EL模式下，内核仅触发一次，那么读取数据的完整性就要求程序保证实现，只要缓冲区有数据就一直读。同样，对于写操作，ET模式下需要程序保证写数据完整性，只要缓冲区还有空间，就一直写。

Epoll的另外两个补充：

ET模式下accept的问题：在某一时刻，有多个连接同时到达，服务器的 TCP 就绪队列瞬间积累多个就绪连接，由于是边缘触发模式，epoll 只会通知一次，accept 只处理一个连接，导致 TCP 就绪队列中剩下的连接都得不到处理。在这种情形下，我们应该如何有效的处理呢？用 while 循环包住 accept 调用，当accept返回-1且 errno 设置为 EAGAIN s就表示所有连接都处理完，再推出while循环。

ET模式设置在非阻塞模式下工作：ET模式下的读写需要一直读或写直到出错（对于读，当读到的实际字节数小于请求字节数时就可以停止），而如果你的文件描述符如果不是非阻塞的，那这个一直读或一直写势必会在最后一次阻塞。这样就不能在阻塞在epoll\_wait上了，造成其他文件描述符的任务饥饿。

小结

LT：水平触发，效率会低于ET触发，尤其在大并发，大流量的情况下。但是LT对代码编写要求比较低，不容易出现问题。LT模式服务编写上的表现是：只要有数据没有被获取，内核就不断通知你，因此不用担心事件丢失的情况。

ET：边缘触发，效率非常高，在并发，大流量的情况下，会比LT少很多epoll的系统调用，因此效率高。但是对编程要求高，需要细致的处理每个请求，否则容易发生丢失事

从本质上讲：与LT相比，ET模型是通过减少系统调用（epoll\_ctl调用）来达到提高并行效率的。

**3.3 select、poll、epoll对比**

select有最大描述符个数限制，select和poll在拥有一个大量描述符连接的情况下且在多数情况下只有部分连接符就绪的时候，每次系统调用都会进行线性轮询所有描述符，会导致性能线性下降。而epoll不存在上述两种缺点，没有描述符上限，且程序仅需要对就绪的描述符进行处理。但是，当所有的描述符基本都是活跃时，例如一个WLAN高速环境，epoll并不比select、poll效率高，反而会因为过多使用epoll\_ctl导致效率有所下降。换句话说，epoll特别适用于连接数量多，但活动连接较少的情况。另外，epoll内核实现使用了链表、mmap和红黑树，使得内存拷贝，在内存拷贝和效率上要比其他突出。

|  |  |  |  |
| --- | --- | --- | --- |
| 系统调用 | select | poll | epoll |
| 事件集合 | 用户通过3个参数分别传入感兴趣的可读，可写及异常等事件  内核通过对这些参数的在线修改来反馈其中的就绪事件  这使得用户每次调用select都要重置这3个参数 | 统一处理所有事件类型，因此只需要一个事件集参数。  用户通过pollfd.events传入感兴趣的事件，内核通过修改pollfd.revents反馈其中就绪的事件 | 内核通过一个事件表直接管理用户感兴趣的所有事件。  因此每次调用epoll\_wait时，无需反复传入用户感兴趣的事件。  epoll\_wait系统调用的参数events仅用来反馈就绪的事件 |
| 应用程序索引就绪文件  描述符的时间复杂度 | O(n) | O(n) | O(1) |
| 最大支持文件描述符数 | 一般有最大值限制 | 无限制 | 无限制 |
| 工作模式 | LT | LT | 支持ET高效模式 |
| 内核实现和工作效率 | 采用轮询方式检测就绪事件，时间复杂度：O(n) | 采用轮询方式检测就绪事件，时间复杂度：O(n) | 采用回调方式检测就绪事件，时间复杂度：O(1) |