<https://blog.csdn.net/drdairen/article/details/53694550>

**1、同步阻塞迭代模型**   
同步阻塞迭代模型是最简单的一种IO模型。

其核心代码如下：

bind(srvfd);

listen(srvfd);

for(;;)

{

clifd = accept(srvfd,...); //开始接受客户端来的连接

read(clifd,buf,...); //从客户端读取数据

dosomthingonbuf(buf);

write(clifd,buf)//发送数据到客户端

}

上面的程序存在如下一些弊端：

1）如果没有客户端的连接请求，进程会**阻塞在accept**系统调用处，程序不能执行其他任何操作。(**系统调用使得程序从用户态陷入内核态**，具体请参考：[程序员的自我修养](http://www.kuqin.com/book/327402.html)   
2）在与客户端建立好一条链路后，通过read系统调用从客户端接受数据，而客户端合适发送数据过来是不可控的。如果客户端迟迟不发生数据过来，则程序同样**会阻塞在read调用**，此时，如果另外的客户端来尝试连接时，都会失败。   
3）同样的道理，**write系统**调用也会使得程序出现阻塞(例如：客户端接受数据异常缓慢，导致写缓冲区满，数据迟迟发送不出)。

**2.多进程并发模型**   
同步阻塞迭代模型有诸多缺点。多进程并发模型在同步阻塞迭代模型的基础上进行了一些改进，以避免是程序阻塞在read系统调用上。

多进程模型核心代码如下：

bind(srvfd);

listen(srvfd);

for(;;){

clifd = accept(srvfd,...); //开始接受客户端来的连接

ret = fork();

switch( ret )

{

case -1 :

do\_err\_handler();

break;

case 0: // 子进程

client\_handler(clifd);

break ;

default : // 父进程

close(clifd);

continue ;

}

}

void client\_handler(clifd)

{

read(clifd,buf,...); //从客户端读取数据

dosomthingonbuf(buf);

write(clifd,buf)//发送数据到客户端

}

上述程序在accept系统调用时，如果没有客户端来建立连接，择**会阻塞在accept处**。一旦某个客户端连接建立起来，则立即开启一个新的进程来处理与这个客户的数据交互。避免程序阻塞在read调用，而影响其他客户端的连接。   
**3.多线程并发模型**   
在多进程并发模型中，每一个客户端连接开启fork一个进程，虽然linux中引入了写实拷贝机制，大大降低了fork一个子进程的消耗，但若客户端连接较大，则系统依然将不堪负重。**通过多线程(或线程池)并发模型，可以在一定程度上改善这一问题。**

在服务端的线程模型实现方式一般有三种：   
（1）按需生成(来一个连接生成一个线程)   
（2）线程池(预先生成很多线程)   
（3）Leader follower（LF）

为简单起见，以第一种为例，其核心代码如下：

void \*thread\_callback( void \*args ) //线程回调函数

{

int clifd = \*(int \*)args ;

client\_handler(clifd);

}

void client\_handler(clifd)

{

read(clifd,buf,...); //从客户端读取数据

dosomthingonbuf(buf);

write(clifd,buf)//发送数据到客户端

}

bind(srvfd);

listen(srvfd);

for(;;)

{

clifd = accept();

pthread\_create(...,thread\_callback,&clifd);

}

服务端分为主线程和工作线程，主线程负责accept()连接，而工作线程负责处理业务逻辑和流的读取等。   
因此，即使在工作线程阻塞的情况下，也只是阻塞在线程范围内，对继续接受新的客户端连接不会有影响。   
第二种实现方式，通过线程池的引入可以避免频繁的创建、销毁线程，能在很大程序上提升性能。   
但不管如何实现，多线程模型先天具有**如下缺点：**   
1）稳定性相对较差。一**个线程的崩溃会导致整个程序崩溃**。   
2）临界资源的访问控制，在加大程序复杂性的同时，**锁机制的引入会是严重降低程序的性能**。性能上可能会出现“辛辛苦苦好几年，一夜回到解放前”的情况。

**4.IO多路复用模型之select/poll**   
多进程模型和多线程(线程池)模型每个进程/线程只能处理一路IO，在服务器并发数较高的情况下，过多的进程/线程会使得服务器性能下降。而通过多路IO复用，能使得**一个进程同时处理多路IO**，提升服务器吞吐量。   
在Linux支持epoll模型之前，都使用select/poll模型来实现IO多路复用。   
以select为例，其核心代码如下：

bind(listenfd);

listen(listenfd);

FD\_ZERO(&allset);

FD\_SET(listenfd, &allset);

for(;;)

{

select(...);

if (FD\_ISSET(listenfd, &rset))

{/\*有新的客户端连接到来\*/

clifd = accept();

cliarray[] = clifd; /\*保存新的连接套接字\*/

FD\_SET(clifd, &allset);/\*将新的描述符加入监听数组中\*/

}

for(;;)

{/\*这个for循环用来检查所有已经连接的客户端是否由数据可读写\*/

fd = cliarray[i];

if (FD\_ISSET(fd , &rset))

dosomething();

}

}

select IO多路复用同样存在一些缺点，罗列如下：

1、**单个进程能够监视的文件描述符的数量存在最大限制，通常是1024，**当然可以更改数量，但由于select\*\*采用轮询的方式扫描文件描述符\*\*，文件描述符数量越多，性能越差；(在linux内核头文件中，有这样的定义：#define \_\_FD\_SETSIZE 1024)   
2、**内核 / 用户空间内存拷贝问题**，select需要复制大量的句柄数据结构，产生巨大的开销；   
select返回的是含有整个句柄的数组，应用程序需要遍历整个数组才能发现哪些句柄发生了事件；   
select的触发方式是水平触发，应用程序如果没有完成对一个已经就绪的文件描述符进行IO操作，那么之后每次select调用还是会将这些文件描述符通知进程。   
3、相比select模型，**poll使用链表保存文件描述符**，因此没有了监视文件数量的限制，但其他三个缺点依然存在。   
拿select模型为例，假设我们的服务器需要支持100万的并发连接，则在\_\_FD\_SETSIZE 为1024的情况下，则我们至少需要开辟1k个进程才能实现100万的并发连接。除了进程间上下文切换的时间消耗外，从内核/用户空间大量的无脑内存拷贝、数组轮询等，是系统难以承受的。因此，**基于select模型的服务器程序，要达到10万级别的并发访问，是一个很难完成的任务。**

int select(int maxfdp,fd\_set \*readfds,fd\_set \*writefds,fd\_set \*errorfds,struct timeval \*timeout);

另一个使用select的例子如下，从网络上接收数据存入一个文件：

main()

{

int sock;

FILE \*fp;

struct fd\_set fds;

struct timeval timeout={3,0}; //select等待3秒，3秒轮询，要非阻塞就置0

char buffer[256]={0}; //256字节的接收缓冲区

/\* 假定已经建立UDP连接，具体过程不写，简单，当然TCP也同理，主机ip和port都已经给定，要写的文件已经打开

sock=socket(...);

bind(...);

fp=fopen(...); \*/

while(1)

{

FD\_ZERO(&fds); //每次循环都要清空集合，否则不能检测描述符变化

FD\_SET(sock,&fds); //添加描述符

FD\_SET(fp,&fds); //同上

maxfdp=sock>fp?sock+1:fp+1; //描述符最大值加1

switch(select(maxfdp,&fds,&fds,NULL,&timeout)) //select使用

{

case -1: exit(-1);break; //select错误，退出程序

case 0:break; //再次轮询

default:

if(FD\_ISSET(sock,&fds)) //测试sock是否可读，即是否网络上有数据

{

recvfrom(sock,buffer,256,.....);//接受网络数据

if(FD\_ISSET(fp,&fds)) //测试文件是否可写

fwrite(fp,buffer...);//写入文件

//...

//buffer清空;

}// end if break;

}// end switch

}//end while

}//end main

select具体参数参见我的另一篇博客：[传送门](http://blog.csdn.net/drdairen/article/details/52577779)

**5.IO多路复用模型之epoll**   
epoll IO多路复用：一个看起来很美好的解决方案。参考文章：[高并发网络编程之epoll详解](http://www.open-open.com/lib/view/open1410403215664.html)

由于epoll的实现机制与select/poll机制完全不同，上面所说的**select的缺点在epoll上不复存在**。   
设想一下如下场景：有100万个客户端同时与一个服务器进程保持着TCP连接。而每一时刻，通常只有几百上千个TCP连接是活跃的(事实上大部分场景都是这种情况)。如何实现这样的高并发？

在select/poll时代，服务器进程每次都把这100万个连接告诉操作系统(从用户态复制句柄数据结构到内核态)，让操作系统内核去查询这些套接字上是否有事件发生，轮询完后，再将句柄数据复制到用户态，让服务器应用程序轮询处理已发生的网络事件，这一过程资源消耗较大，因此，**select/poll一般只能处理几千的并发连接。**

epoll的设计和实现与select完全不同。**epoll通过在Linux内核中申请一个简易的文件系统**(文件系统一般用什么数据结构实现？B+树)。把原先的select/poll调用分成了3个部分：   
1）调用epoll\_create()建立一个epoll对象(在epoll文件系统中为这个句柄对象分配资源)   
2）调用epoll\_ctl向epoll对象中添加这100万个连接的套接字   
3）调用epoll\_wait收集发生的事件的连接

如此一来，要实现上面说是的场景，只需要在**进程启动时建立一个epoll对象**，然后在需要的时候向这个epoll对象中添加或者删除连接。同时，epoll\_wait的效率也非常高，因为调用epoll\_wait时，并没有一股脑的向操作系统复制这100万个连接的句柄数据，内核也不需要去遍历全部的连接。   
select poll epoll总结参考我的另一篇博客：[传送门](http://blog.csdn.net/drdairen/article/details/52946773)   
epoll是在2.6内核中提出的，是之前的select和poll的增强版本。相对于select和poll来说，epoll更加灵活，没有描述符限制。epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。

epoll操作过程需要三个接口，分别如下：

#include <sys/epoll.h>

int epoll\_create(int size);

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

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

　　epoll既然是对select和poll的改进，就应该能避免上述的三个缺点。那epoll都是怎么解决的呢？在此之前，我们先看一下epoll和select和poll的调用接口上的不同，select和poll都只提供了一个函数——select或者poll函数。而epoll提供了三个函数，epoll\_create,epoll\_ctl和epoll\_wait，epoll\_create是创建一个epoll句柄；epoll\_ctl是注册要监听的事件类型；epoll\_wait则是等待事件的产生。

　　1、对于第一个缺点，epoll的解决方案在epoll\_ctl函数中。每次注册新的事件到epoll句柄中时（在epoll\_ctl中指定EPOLL\_CTL\_ADD），会把所有的fd拷贝进内核，而不是在epoll\_wait的时候重复拷贝。**epoll保证了每个fd在整个过程中只会拷贝一次。**

　2、对于第二个缺点，epoll的解决方案不像select或poll一样每次都把current轮流加入fd对应的设备等待队列中，而只在epoll\_ctl时把current挂一遍（这一遍必不可少）并为每个fd指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调函数，而这个回调函数会把就绪的fd加入一个就绪链表）。**epoll\_wait的工作实际上就是在这个就绪链表中查看有没有就绪的fd**（利用schedule\_timeout()实现睡一会，判断一会的效果）。   
　3、对于第三个缺点，epoll没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,**在1GB内存的机器上大约是10万左右**，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

当某一进程调用epoll\_create方法时，Linux内核会创建一个eventpoll结构体，这个结构体中有两个成员与epoll的使用方式密切相关。eventpoll结构体如下所示：

struct eventpoll{

....

/\*红黑树的根节点，这颗树中存储着所有添加到epoll中的需要监控的事件\*/

struct rb\_root rbr;

/\*双链表中则存放着将要通过epoll\_wait返回给用户的满足条件的事件\*/

struct list\_head rdlist;

....

};

每一个epoll对象都有一个独立的eventpoll结构体，用于存放通过epoll\_ctl方法向epoll对象中添加进来的事件。这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来(红黑树的插入时间效率是lgn，其中n为树的高度)。

而所有添加到epoll中的事件都会与设备(网卡)驱动程序建立回调关系，也就是说，当相应的事件发生时会调用这个回调方法。这个回调方法在内核中叫ep\_poll\_callback,它会将发生的事件添加到rdlist双链表中。

在epoll中，对于每一个事件，都会建立一个epitem结构体，如下所示：

struct epitem{

struct rb\_node rbn;//红黑树节点

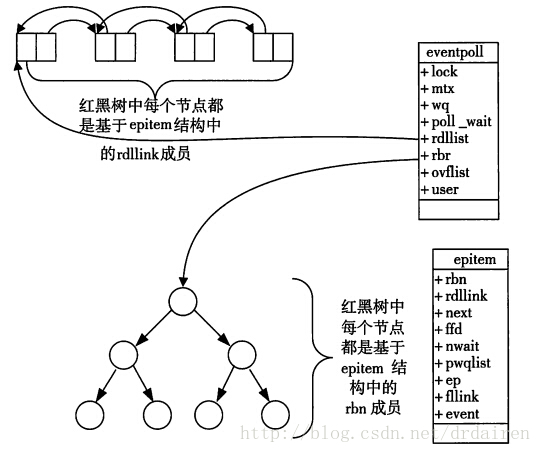
struct list\_head rdllink;//双向链表节点

struct epoll\_filefd ffd; //事件句柄信息

struct eventpoll \*ep; //指向其所属的eventpoll对象

struct epoll\_event event; //期待发生的事件类型

}

图示如下：   
   
当调用epoll\_wait检查是否有事件发生时，只需要检查eventpoll对象中的rdlist双链表中是否有epitem元素即可。如果rdlist不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户。

**epoll的用法分为三步：**   
\*\*第一步：epoll\_create()系统调用。此调用返回一个句柄，之后所有的使用都依靠这个句柄来标识。   
第二步：epoll\_ctl()系统调用。通过此调用向epoll对象中添加、删除、修改感兴趣的事件，返回0标识成功，返回-1表示失败。   
第三步：epoll\_wait()系统调用。通过此调用收集收集在epoll监控中已经发生的事件。\*\*

下面是一个简单使用了epoll和pipe管道的例子，子进程写，父进程使用epoll监听管道写端。

#include <iostream>

#include <unistd.h>

#include <errno.h>

#include <sys/epoll.h>

#include <cstdio>

#include <cstdlib>

#include <cstring>

using namespace std;

int main()

{

struct epoll\_event ev; //事件临时变量

const int MAXEVENTS = 1024; //最大事件数

struct epoll\_event events[MAXEVENTS]; //监听事件数组

int ret,pid;

int pipe\_fd[2];

if((ret=pipe(pipe\_fd))<0)//创建管道的两端，0读 1写

{

cout<<"create pipe fail:"<<ret<< ",errno:" << errno <<endl;

return -1;

}

ev.data.fd = pipe\_fd[0]; //设置监听文件描述符

/\*EPOLLLT 是默认行为，基本上就是说，只要一个文件描述符处于就绪状态，epoll 就会不停的通知你有事件发生。传统的 select/poll 都是这样的

EPOLLET 是新的方式，只在一个文件描述符新处于就绪的时候通知一次，之后不管数据有没有读完，都不会再通知，当然，有新数据到还是会通知的。

所以，用 EPOLLET 的时候，一定要把文件描述符设置为 non-blocking，而且最好是一直读数据，读到返回 EAGAIN 才停下\*/

ev.events = EPOLLIN|EPOLLET; //设置要处理的事件类型

int epfd=epoll\_create(MAXEVENTS);

ret=epoll\_ctl(epfd,EPOLL\_CTL\_ADD,pipe\_fd[0],&ev);

if (ret != 0)

{

cout << "epoll\_ctl fail:" << ret << ",errno:" << errno << endl;

close(pipe\_fd[0]);

close(pipe\_fd[1]);

close(epfd);

return -1;

}

if((pid=fork())>0)//父进程

{

int count=epoll\_wait(epfd,events,MAXEVENTS,5000);

char r\_buf[100];

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

{

if((events[i].data.fd==pipe\_fd[0])&&(events[0].events&EPOLLIN))

{

int r\_num=read(pipe\_fd[0],r\_buf,100);

printf("read num is %d bytes data from the pipe,value is %d \n",r\_num,atoi(r\_buf));

}

}

close(pipe\_fd[1]);

close(pipe\_fd[0]);

close(epfd);

cout<<"parent close read fd[0],wirte fd[1] and epfd over"<<endl;

}//子进程

else if(pid==0)

{

close(pipe\_fd[0]);//关读端

cout<<"sub close read fd[0] over,sub does't read"<<endl;

char w\_buf[100];

strcpy(w\_buf,"1234");

if(write(pipe\_fd[1],w\_buf,5)!=-1)//子进程往管道里写

printf("sub write over \n");

close(pipe\_fd[1]);//关写端

printf("sub close write fd[1] over \n");

}

return 0;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69

注意这一句ev.events = EPOLLIN|EPOLLET;   
这里要说   
**IO模型的两种触发方式**，如下：   
**LT（Level\_triggered(水平触发)）**:是epoll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，直至变为未就绪状态，也就是epoll\_wait()会通知处理程序去读写。如果这次没有把数据一次性全部读写完(如读写缓冲区太小)，那么下次调用 epoll\_wait()时，它还会通知你在上次没读写完的文件描述符上继续读写所以，这种模式编程出错误可能性要小一点。传统的select/poll都是只有这种触发方式。   
**ET （Edge\_triggered(边缘触发)）**: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\_wait()通知处理程序去读写,如果这次没有把数据全部读写完(如读写缓冲区太小)，那么下次调用epoll\_wait()时，它不会通知你，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，也就是它只会通知你一次，直到该文件描述符上出现第二次可读写事件才会通知你。   
下面是一个epoll在网络编程上的典型用法，ET non-blocking模式下的服务器代码：

#include <sys/socket.h>

#include <sys/wait.h>

#include <netinet/in.h>

#include <netinet/tcp.h>

#include <sys/epoll.h>

#include <sys/sendfile.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <strings.h>

#include <fcntl.h>

#include <errno.h>

#define MAX\_EVENTS 10

#define PORT 8080

//设置socket连接为非阻塞模式

void setnonblocking(int sockfd) {

int opts;

opts = fcntl(sockfd, F\_GETFL);

if(opts < 0) {

perror("fcntl(F\_GETFL)\n");

exit(1);

}

opts = (opts | O\_NONBLOCK);

if(fcntl(sockfd, F\_SETFL, opts) < 0) {

perror("fcntl(F\_SETFL)\n");

exit(1);

}

}

int main(){

struct epoll\_event ev, events[MAX\_EVENTS];

int addrlen, listenfd, conn\_sock, nfds, epfd, fd, i, nread, n;

struct sockaddr\_in local, remote;

char buf[BUFSIZ];

//创建listen socket

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

{

perror("sockfd\n");

exit(1);

}

setnonblocking(listenfd);

bzero(&local, sizeof(local));

local.sin\_family = AF\_INET;

local.sin\_addr.s\_addr = htonl(INADDR\_ANY);;

local.sin\_port = htons(PORT);

if( bind(listenfd, (struct sockaddr \*) &local, sizeof(local)) < 0)

{

perror("bind\n");

exit(1);

}

listen(listenfd, 20);//设置为监听描述符

epfd = epoll\_create(MAX\_EVENTS);

if (epfd == -1)

{

perror("epoll\_create");

exit(EXIT\_FAILURE);

}

ev.events = EPOLLIN;

ev.data.fd = listenfd;

if (epoll\_ctl(epfd, EPOLL\_CTL\_ADD, listenfd, &ev) == -1)

{

perror("epoll\_ctl: listen\_sock");

exit(EXIT\_FAILURE);

}

for (;;)

{

nfds = epoll\_wait(epfd, events, MAX\_EVENTS, -1);//超时时间-1，永久阻塞直到有事件发生

if (nfds == -1)

{

perror("epoll\_pwait");

exit(EXIT\_FAILURE);

}

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

{

fd = events[i].data.fd;

if (fd == listenfd) //如果是监听的listenfd，那就是连接来了，保存来的所有连接

{

//每次处理一个连接，while循环直到处理完所有的连接

while ((conn\_sock = accept(listenfd,(struct sockaddr \*) &remote,

(size\_t \*)&addrlen)) > 0)

{

setnonblocking(conn\_sock);

ev.events = EPOLLIN | EPOLLET;//边沿触发非阻塞模式

ev.data.fd = conn\_sock;

//把连接socket加入监听结构体

if (epoll\_ctl(epfd, EPOLL\_CTL\_ADD, conn\_sock,

&ev) == -1) {

perror("epoll\_ctl: add");

exit(EXIT\_FAILURE);

}

}

//已经处理完所有的连：accept返回-1，errno为EAGAIN

//出错：返回-1，errno另有其值

if (conn\_sock == -1)

{

if (errno != EAGAIN && errno != ECONNABORTED

&& errno != EPROTO && errno != EINTR)

perror("accept");

}

continue;//继续循环，但是不执行该循环后面的部分

}

if (events[i].events & EPOLLIN) //可读事件

{

n = 0;

while ((nread = read(fd, buf + n, BUFSIZ-1)) > 0)

{

n += nread;

}

if (nread == -1 && errno != EAGAIN)

{

perror("read error");

}

ev.data.fd = fd;

ev.events = events[i].events | EPOLLOUT;

//修改该fd监听事件类型，监测是否可写

if (epoll\_ctl(epfd, EPOLL\_CTL\_MOD, fd, &ev) == -1)

{

perror("epoll\_ctl: mod");

}

}

if (events[i].events & EPOLLOUT) //可写事件

{

sprintf(buf, "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\nHello World", 11);

int nwrite, data\_size = strlen(buf);

n = data\_size;

while (n > 0)

{

nwrite = write(fd, buf + data\_size - n, n);

if (nwrite < n)

{

if (nwrite == -1 && errno != EAGAIN)

{

perror("write error");

}

break;

}

n -= nwrite;

}

//写完就关闭该连接socket

close(fd);

}

}

}

return 0;

}

必要的讲解已经在注释里了，我这里就不再赘述。

版权声明：本文为博主原创文章，未经博主允许不得转载，否则博主会很生气，后果会很严重。 https://blog.csdn.net/drdairen/article/details/53694550