**参考资料：**

**知乎作者：**[**自由技艺**](https://www.zhihu.com/people/wang-xiao-xiao-62-32-89)

**Linux 五种 IO 模式及 select、poll、epoll 详解（附样例代码）**

**0、前言**

对于每一个从事 Web 服务器开发的后端程序员，必然绕不开网络编程，而其中最基础也是最重要的部分就是 Linux IO 模式及 Socket 编程。什么是 Socket 呢？简单理解，就是 ip地址 + 端口号。当两个进程间需要通信时，首先要创建五元组（源ip地址、目的ip地址、源端口号、目的端口号、协议），建立 tcp 连接，建立好连接之后，两个进程各自有一个 Socket 来标识，这两个 socket 组成的 socket pair 也就唯一标识了一个连接。有了连接之后，应用程序得要从 tcp 流上获取数据，然后再处理数据。于是，诞生了三种高效的 Socket 编程方法：select、poll 和 epoll.

本文将首先介绍下 Linux IO 的几种模式以及一些前置知识，因为这是理解 select、poll 和 epoll 的前提；接下来重点介绍下 select、poll 和 epoll 的工作原理及优缺点；最后附上样例代码

注：本人并不是从事 web 后端开发的工作，平时也就用用 grpc/brpc（封装了 socket 编程模型），所以尽量从概念及实现原理上把这个知识地点讲清楚，如果有讲的不对的地方，欢迎专业人士批评指正。

**1、几个基础概念**

**1.1、用户空间和内核空间**  
对于32位操作系统而言，它的寻址空间是4G（2的32次方），注意这里的4G是虚拟内存空间大小。以 Linux 为例，它将最高的1G字节给内核使用，称为内核空间，剩下的3G给用户进程使用，称为用户空间。这样做的好处就是隔离，保证内核安全。

**1.2、进程切换**  
这是内核要干的事，字面意思很好理解，挂起正在运行的 A 进程，然后运行 B 进程，当然这其中的流程比较复杂，涉及到上下文切换，且非常消耗资源，感兴趣的同学可以去深入研究。

**1.3、进程的阻塞**

* 进程阻塞是本进程的行为，比如和其他进程通信时，等待请求的数据返回；
* 进程进入阻塞状态时不占用CPU资源的

**1.4、文件描述符**  
在 Linux 世界里，一切皆文件。怎么理解呢？当程序打开一个现有文件或创建新文件时，内核会向进程返回一个文件描述符，文件描述符在形式上是一个非负整数，其实就是一个索引值，指向该进程打开文件的记录表（它是由内核维护的）。

**1.5、缓存 I/O**  
和标准 IO 是一个概念，当应用程序需要从内核读数据时，数据先被拷贝到操作系统的内核缓冲区（page cache），然后再从该缓冲区拷贝到应用程序的地址空间。

**2、Linux IO 模式**

当应用程序发起一次 read 调用时，会经历以下两个阶段：

1. 等待数据准备 (Waiting for the data to be ready)
2. 将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)

**正式因为这两个阶段，Linux 系统产生了下述五种 IO 方式：**

* 阻塞 I/O（blocking IO）
* 非阻塞 I/O（nonblocking IO）
* 异步 I/O（asynchronous IO）
* I/O 多路复用（ IO multiplexing）
* 信号驱动 I/O（ signal driven IO）（很少见，可忽略）

它们具体怎么工作，这里做下总结：

**2.1、blocking 和 non-blocking的区别**  
blocking IO的特点就是在IO执行的两个阶段都被block了。

non-blocking IO 的特点是用户进程需要不断的主动询问内核 “ 数据好了吗？”

**2.2、IO 多路复用**  
I/O 多路复用的特点是通过一种机制一个进程能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回。

**2.3、synchronous IO和asynchronous IO的区别**

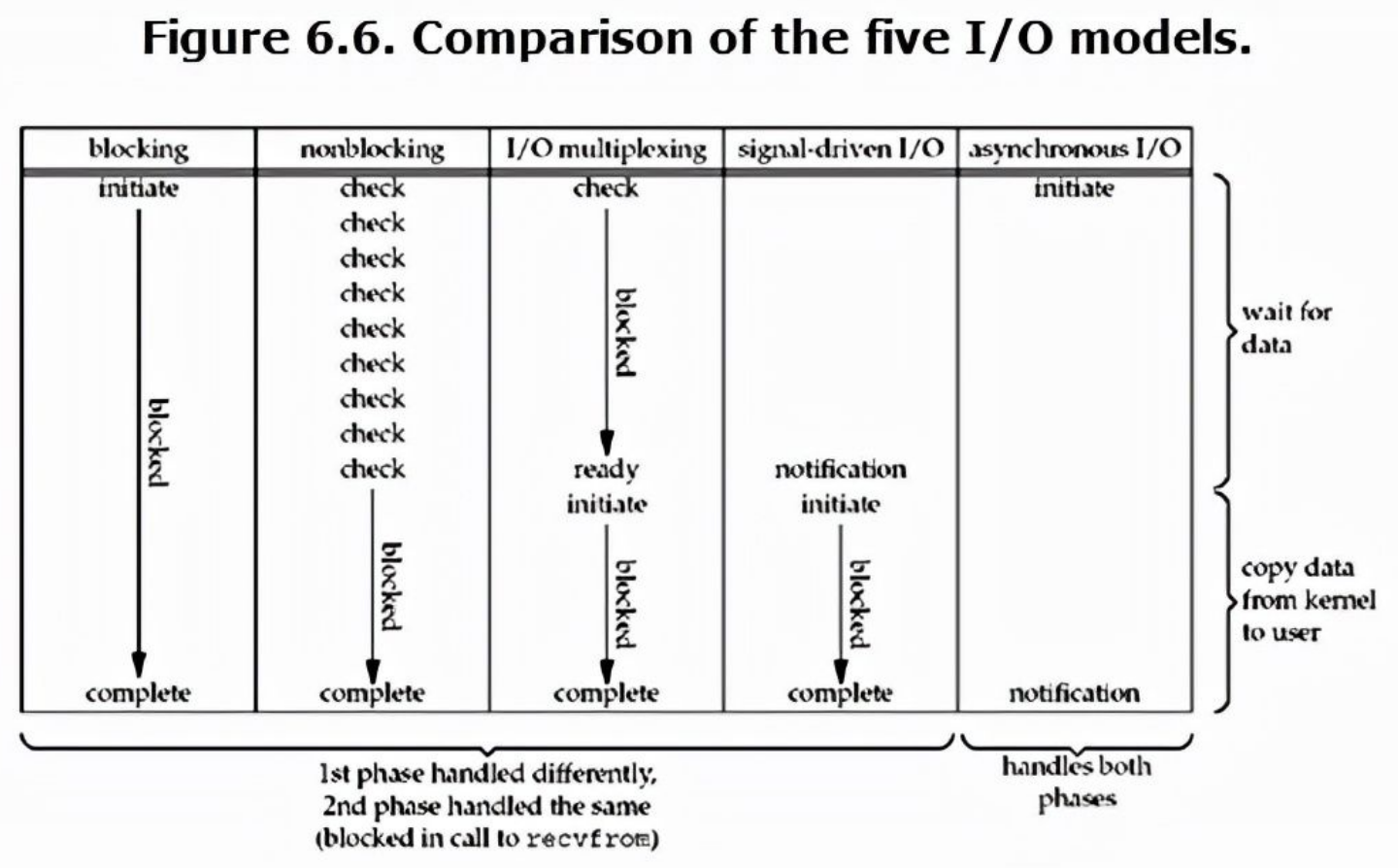
POSIX 中是这样定义的：

- A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes;

- An asynchronous I/O operation does not cause the requesting process to be blocked;

两者的区别就在于synchronous IO做”IO operation”的时候会将process阻塞。按照这个定义，之前所述的blocking IO，non-blocking IO，IO multiplexing 都属于 synchronous IO。

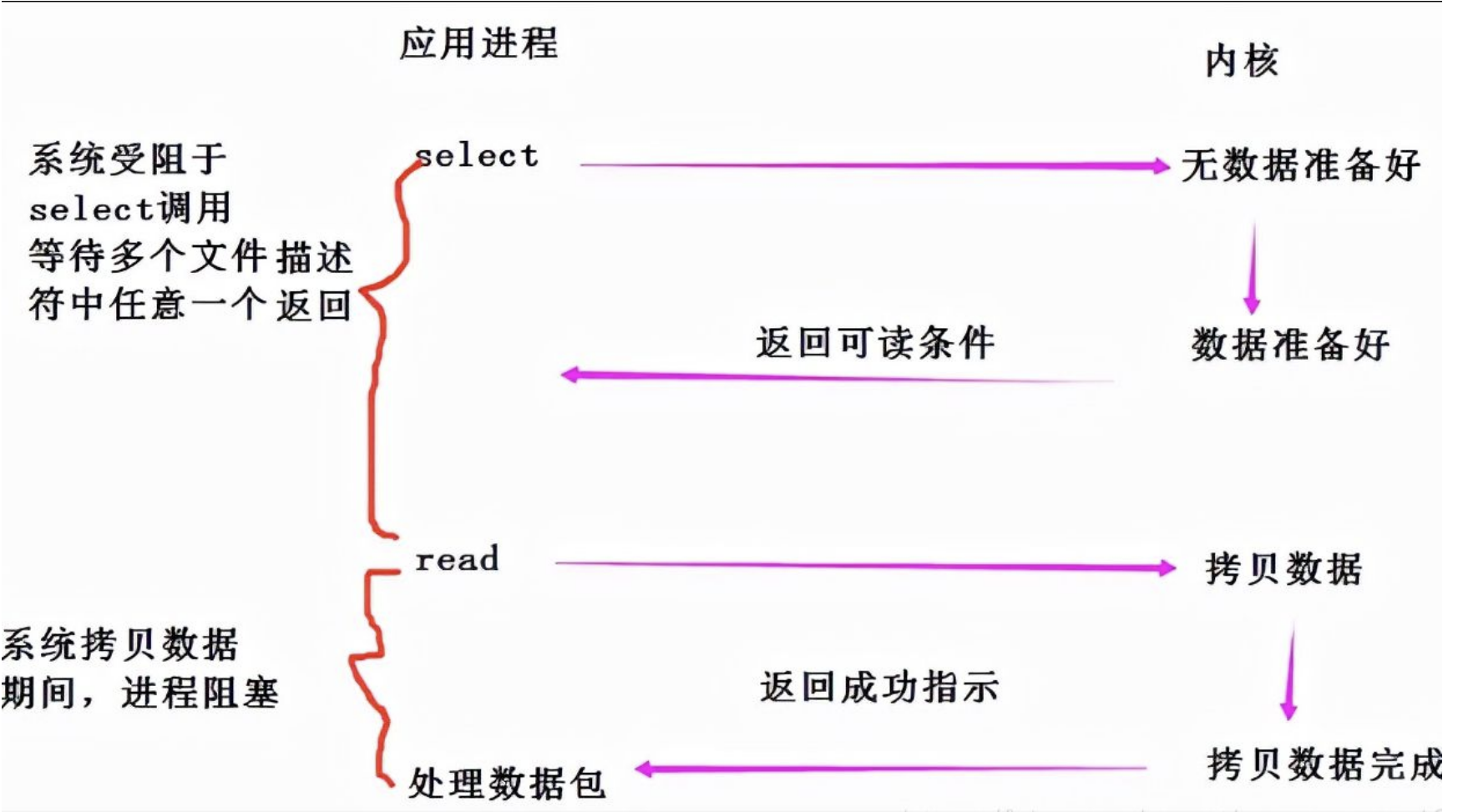
**2.4、各个IO 模式的比较**



可以发现：

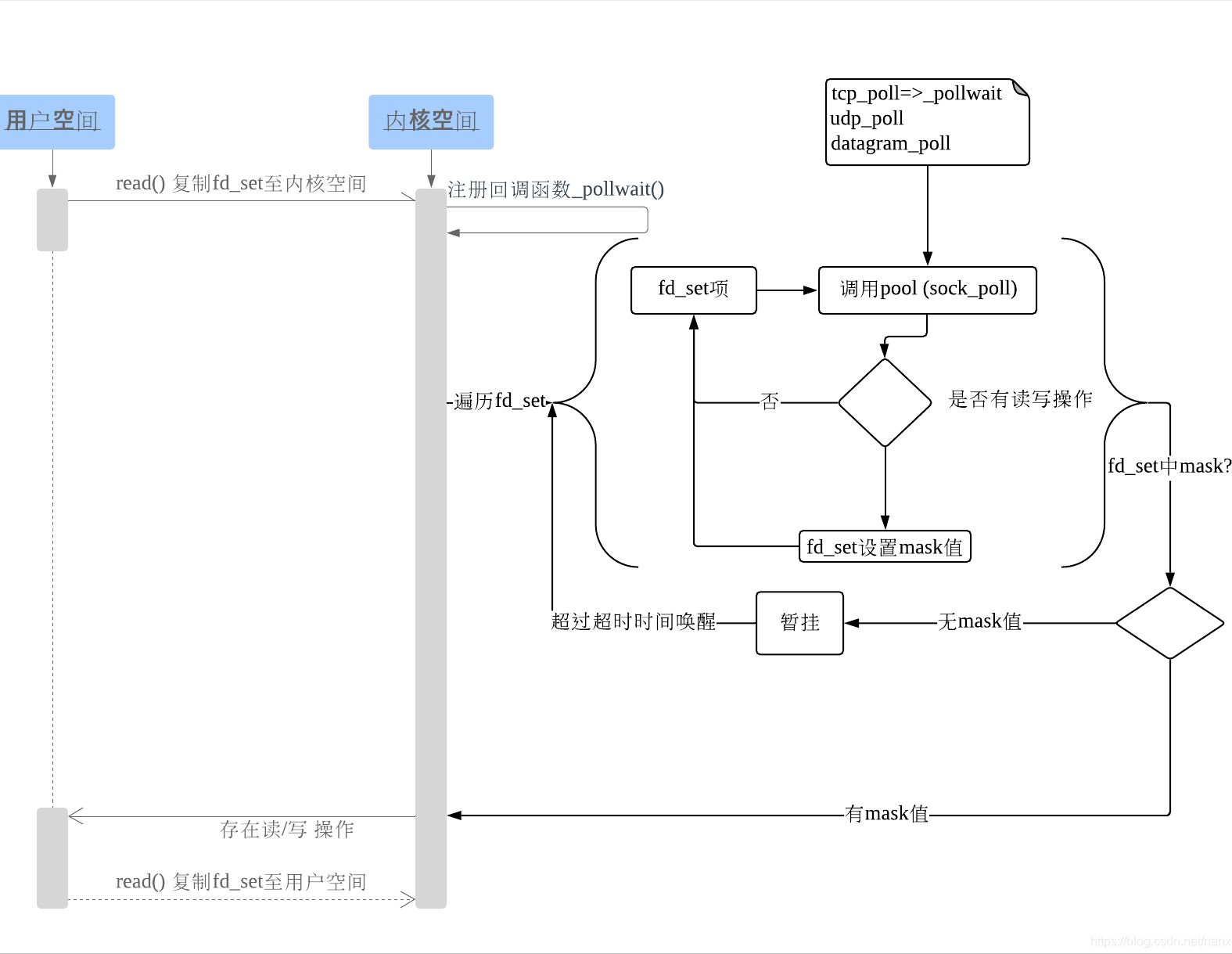
* 对于 non-blocking IO中，虽然进程大部分时间都不会被 block，但是它仍然要求进程去主动的 check，并且当数据准备完成以后，也需要进程主动的再次调用recvfrom来将数据拷贝到用户内存。
* asynchronous IO 完全不同于 no-blocking IO，它就像是用户进程将整个 IO 操作交给了内核完成，然后内核做完后发出信号通知。在此期间，用户进程不需要去检查 IO 操作的状态，也不需要主动的去拷贝数据。

**3、IO 多路复用之 select、poll、epoll 详解**  
select，poll，epoll 都是 IO 多路复用的机制，它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。  
**3.1、select**



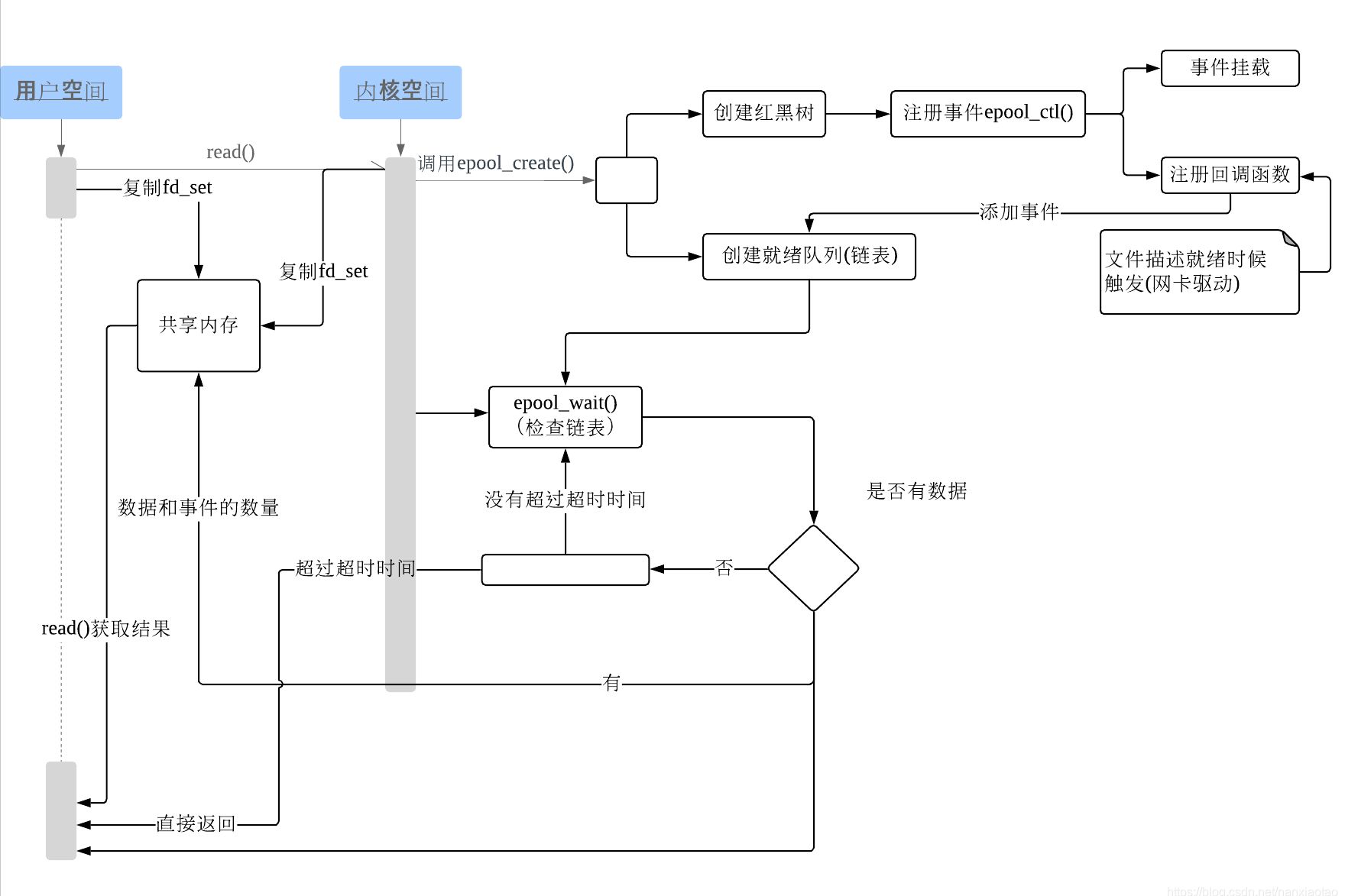
select 最多能同时监视 1024 个 socket（因为 fd\_set 结构体大小是 128 字节，每个 bit 表示一个文件描述符）。用户需要维护一个临时数组，存储文件描述符。当内核有事件发生时，内核将 fd\_set 中没发生的文件描述符清空，然后拷贝到用户区。select 返回的是整个数组，它需要遍历整个数组才知道谁发生了变化。

**3.2、poll**



poll 就是把 select 中的 fd\_set 数组换成了链表，其他和 select 没什么不同。

**3.3、epoll**



epoll 是基于事件驱动的 IO 方式，它没有文件描述符个数限制，它将用户关心的文件描述符的事件存放到内核的一个事件表中（简单来说，就是由内核来负责存储（红黑树）有事件的 socket 句柄），这样在用户空间和内核空间的copy只需一次。优点如下：

1. 没有最大并发连接的限制，能打开的fd上限远大于1024（1G的内存能监听约10万个端口）
2. 采用回调的方式，效率提升。只有**活跃可用**的fd才会调用callback函数，也就是说 epoll 只管你“活跃”的连接，而跟连接总数无关；
3. 内存拷贝。使用mmap()文件映射内存来加速与内核空间的消息传递，减少复制开销。

epoll 有两种工作方式：

* LT模式（水平触发）：若就绪的事件一次没有处理完，就会一直去处理。也就是说，将没有处理完的事件继续放回到就绪队列之中（即那个内核中的链表），一直进行处理。
* ET模式（边缘触发）：就绪的事件只能处理一次，若没有处理完会在下次的其它事件就绪时再进行处理。而若以后再也没有就绪的事件，那么剩余的那部分数据也会随之而丢失。

由此可见：ET模式的效率比LT模式的效率要高很多。*简单点说就是，如果对于一个非阻塞 socket，如果使用 epoll 边缘模式去检测数据是否可读，触发可读事件以后，一定要一次性把 socket 上的数据收取干净才行，也就是说一定要循环调用 recv 函数直到 recv 出错，错误码是****EWOULDBLOCK****（****EAGAIN****一样）（此时表示 socket 上本次数据已经读完）；如果使用水平模式，则不用，你可以根据业务一次性收取固定的字节数，或者收完为止。*只是如果使用ET模式，就要保证每次进行数据处理时，要将其处理完，不能造成数据丢失，这样对编写代码的人要求就比较高。

**3.4、select、poll、epoll 三者区别**



**4、代码详解**  
**4.1 select**

**#include**<stdio.h>

**#include**<sys/types.h>

**#include**<sys/socket.h>

**#include**<unistd.h>

**#include**<netinet/in.h>

**#include**<arpa/inet.h>

**#include**<stdlib.h>

**#include**<string.h>

**#include**<sys/time.h>

**static** **void** **Usage**(**const** **char\*** proc)

{

printf("%s [local\_ip] [local\_port]\n",proc);

}

**int** array[4096];

**static** **int** **start\_up**(**const** **char\*** \_ip,**int** \_port)

{

**int** sock **=** socket(AF\_INET,SOCK\_STREAM,0);

**if**(sock **<** 0)

{

perror("socket");

exit(1);

}

**struct** **sockaddr\_in** local;

local.sin\_family **=** AF\_INET;

local.sin\_port **=** htons(\_port);

local.sin\_addr.s\_addr **=** inet\_addr(\_ip);

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

{

perror("bind");

exit(2);

}

**if**(listen(sock,10) **<** 0)

{

perror("listen");

exit(3);

}

**return** sock;

}

**int** **main**(**int** argc,**char\*** argv[])

{

**if**(argc **!=** 3)

{

Usage(argv[0]);

**return** **-**1;

}

**int** listensock **=** start\_up(argv[1],atoi(argv[2]));

**int** maxfd **=** 0;

fd\_set rfds;

fd\_set wfds;

array[0] **=** listensock;

**int** i **=** 1;

**int** array\_size **=** **sizeof**(array)**/sizeof**(array[0]);

**for**(; i **<** array\_size;i**++**)

{

array[i] **=** **-**1;

}

**while**(1)

{

FD\_ZERO(**&**rfds);

FD\_ZERO(**&**wfds);

**for**(i **=** 0;i **<** array\_size;**++**i)

{

**if**(array[i] **>** 0)

{

FD\_SET(array[i],**&**rfds);

FD\_SET(array[i],**&**wfds);

**if**(array[i] **>** maxfd)

{

maxfd **=** array[i];

}

}

}

**switch**(select(maxfd **+** 1,**&**rfds,**&**wfds,NULL,NULL))

{

**case** 0**:**

{

printf("timeout\n");

**break**;

}

**case** **-**1**:**

{

perror("select");

**break**;

}

**default:**

{

**int** j **=** 0;

**for**(; j **<** array\_size; **++**j)

{

**if**(j **==** 0 **&&** FD\_ISSET(array[j],**&**rfds))

{

*//listensock happened read events*

**struct** **sockaddr\_in** client;

socklen\_t len **=** **sizeof**(client);

**int** new\_sock **=** accept(listensock,(**struct** **sockaddr\***)**&**client,**&**len);

**if**(new\_sock **<** 0)*//accept failed*

{

perror("accept");

**continue**;

}

**else***//accept success*

{

printf("get a new client%s\n",inet\_ntoa(client.sin\_addr));

fflush(stdout);

**int** k **=** 1;

**for**(; k **<** array\_size;**++**k)

{

**if**(array[k] **<** 0)

{

array[k] **=** new\_sock;

**if**(new\_sock **>** maxfd)

maxfd **=** new\_sock;

**break**;

}

}

**if**(k **==** array\_size)

{

close(new\_sock);

}

}

}*//j == 0*

**else** **if**(j **!=** 0 **&&** FD\_ISSET(array[j], **&**rfds))

{

*//new\_sock happend read events*

**char** buf[1024];

ssize\_t s **=** read(array[j],buf,**sizeof**(buf) **-** 1);

**if**(s **>** 0)*//read success*

{

buf[s] **=** 0;

printf("clientsay#%s\n",buf);

**if**(FD\_ISSET(array[j],**&**wfds))

{

**char** **\***msg **=** "HTTP/1.0 200 OK <\r\n\r\n<html><h1>yingying beautiful</h1></html>\r\n";

write(array[j],msg,strlen(msg));

}

}

**else** **if**(0 **==** s)

{

printf("client quit!\n");

close(array[j]);

array[j] **=** **-**1;

}

**else**

{

perror("read");

close(array[j]);

array[j] **=** **-**1;

}

}*//else j != 0*

}

**break**;

}

}

}

**return** 0;

}

**4.2、Poll**

**#include**<stdio.h>

**#include**<stdlib.h>

**#include**<string.h>

**#include**<sys/types.h>

**#include**<sys/socket.h>

**#include**<netinet/in.h>

**#include**<arpa/inet.h>

**#include**<poll.h>

**static** **void** **usage**(**const** **char** **\***proc)

{

printf("%s [local\_ip] [local\_port]\n",proc);

}

**int** **start\_up**(**const** **char\***\_ip,**int** \_port)

{

**int** sock **=** socket(AF\_INET,SOCK\_STREAM,0);

**if**(sock **<** 0)

{

perror("socket");

**return** 2;

}

**int** opt **=** 1;

setsockopt(sock,SOL\_SOCKET,SO\_REUSEADDR,**&**opt,**sizeof**(opt));

**struct** **sockaddr\_in** local;

local.sin\_family **=** AF\_INET;

local.sin\_port **=** htons(\_port);

local.sin\_addr.s\_addr **=** inet\_addr(\_ip);

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

{

perror("bind");

**return** 3;

}

**if**(listen(sock,10) **<** 0)

{

perror("listen");

**return** 4;

}

**return** sock;

}

**int** **main**(**int** argc, **char\***argv[])

{

**if**(argc **!=** 3)

{

usage(argv[0]);

**return** 1;

}

**int** sock **=** start\_up(argv[1],atoi(argv[2]));

**struct** **pollfd** peerfd[1024];

peerfd[0].fd **=** sock;

peerfd[0].events **=** POLLIN;

**int** nfds **=** 1;

**int** ret;

**int** maxsize **=** **sizeof**(peerfd)**/sizeof**(peerfd[0]);

**int** i **=** 1;

**int** timeout **=** **-**1;

**for**(; i **<** maxsize; **++**i)

{

peerfd[i].fd **=** **-**1;

}

**while**(1)

{

**switch**(ret **=** poll(peerfd,nfds,timeout))

{

**case** 0**:**

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

**break**;

**case** **-**1**:**

perror("poll");

**break**;

**default:**

{

**if**(peerfd[0].revents **&** POLLIN)

{

**struct** **sockaddr\_in** client;

socklen\_t len **=** **sizeof**(client);

**int** new\_sock **=** accept(sock,\

(**struct** **sockaddr\***)**&**client,**&**len);

printf("accept finish %d\n",new\_sock);

**if**(new\_sock **<** 0)

{

perror("accept");

**continue**;

}

printf("get a new client\n");

**int** j **=** 1;

**for**(; j **<** maxsize; **++**j)

{

**if**(peerfd[j].fd **<** 0)

{

peerfd[j].fd **=** new\_sock;

**break**;

}

}

**if**(j **==** maxsize)

{

printf("to many clients...\n");

close(new\_sock);

}

peerfd[j].events **=** POLLIN;

**if**(j **+** 1 **>** nfds)

nfds **=** j **+** 1;

}

**for**(i **=** 1;i **<** nfds;**++**i)

{

**if**(peerfd[i].revents **&** POLLIN)

{

printf("read ready\n");

**char** buf[1024];

ssize\_t s **=** read(peerfd[i].fd,buf, \

**sizeof**(buf) **-** 1);

**if**(s **>** 0)

{

buf[s] **=** 0;

printf("client say#%s",buf);

fflush(stdout);

peerfd[i].events **=** POLLOUT;

}

**else** **if**(s **<=** 0)

{

close(peerfd[i].fd);

peerfd[i].fd **=** **-**1;

}

**else**

{

}

}*//i != 0*

**else** **if**(peerfd[i].revents **&** POLLOUT)

{

**char** **\***msg **=** "HTTP/1.0 200 OK \

<\r\n\r\n<html><h1> \

yingying beautiful \

</h1></html>\r\n";

write(peerfd[i].fd,msg,strlen(msg));

close(peerfd[i].fd);

peerfd[i].fd **=** **-**1;

}

**else**

{

}

}*//for*

}*//default*

**break**;

}

}

**return** 0;

}

具体流程如下：

1. poll()函数返回fds集合中就绪的读、写，或出错的描述符数量，返回0表示超时，返回-1表示出错；
2. fds是一个struct pollfd类型的数组，用于存放需要检测其状态的socket描述符，并且调用poll函数之后fds数组不会被清空；
3. nfds记录数组fds中描述符的总数量；
4. timeout是调用poll函数阻塞的超时时间，单位毫秒；
5. 一个pollfd结构体表示一个被监视的文件描述符，通过传递fds[]指示 poll() 监视多个文件描述符。其中，结构体的events域是监视该文件描述符的事件掩码，由用户来设置这个域，结构体的revents域是文件描述符的操作结果事件掩码，内核在调用返回时设置这个域。events域中请求的任何事件都可能在revents域中返回。

**4.3、epoll**

**#include**<stdio.h>

**#include**<sys/types.h>

**#include**<sys/socket.h>

**#include**<netinet/in.h>

**#include**<arpa/inet.h>

**#include**<stdlib.h>

**#include**<string.h>

**#include**<sys/epoll.h>

**static** **Usage**(**const** **char\*** proc)

{

printf("%s [local\_ip] [local\_port]\n",proc);

}

**int** **start\_up**(**const** **char\***\_ip,**int** \_port)

{

**int** sock **=** socket(AF\_INET,SOCK\_STREAM,0);

**if**(sock **<** 0)

{

perror("socket");

exit(2);

}

**struct** **sockaddr\_in** local;

local.sin\_family **=** AF\_INET;

local.sin\_port **=** htons(\_port);

local.sin\_addr.s\_addr **=** inet\_addr(\_ip);

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

{

perror("bind");

exit(3);

}

**if**(listen(sock,10)**<** 0)

{

perror("listen");

exit(4);

}

**return** sock;

}

**int** **main**(**int** argc, **char\***argv[])

{

**if**(argc **!=** 3)

{

Usage(argv[0]);

**return** 1;

}

**int** sock **=** start\_up(argv[1],atoi(argv[2]));

**int** epollfd **=** epoll\_create(256);

**if**(epollfd **<** 0)

{

perror("epoll\_create");

**return** 5;

}

**struct** **epoll\_event** ev;

ev.events **=** EPOLLIN;

ev.data.fd **=** sock;

**if**(epoll\_ctl(epollfd,EPOLL\_CTL\_ADD,sock,**&**ev) **<** 0)

{

perror("epoll\_ctl");

**return** 6;

}

**int** evnums **=** 0;*//epoll\_wait return val*

**struct** **epoll\_event** evs[64];

**int** timeout **=** **-**1;

**while**(1)

{

**switch**(evnums **=** epoll\_wait(epollfd,evs,64,timeout))

{

**case** 0**:**

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

**break**;

**case** **-**1**:**

perror("epoll\_wait");

**break**;

**default:**

{

**int** i **=** 0;

**for**(; i **<** evnums; **++**i)

{

**struct** **sockaddr\_in** client;

socklen\_t len **=** **sizeof**(client);

**if**(evs[i].data.fd **==** sock \

**&&** evs[i].events **&** EPOLLIN)

{

**int** new\_sock **=** accept(sock, \

(**struct** **sockaddr\***)**&**client,**&**len);

**if**(new\_sock **<** 0)

{

perror("accept");

**continue**;

}*//if accept failed*

**else**

{

printf("Get a new client[%s]\n", \

inet\_ntoa(client.sin\_addr));

ev.data.fd **=** new\_sock;

ev.events **=** EPOLLIN;

epoll\_ctl(epollfd,EPOLL\_CTL\_ADD,\

new\_sock,**&**ev);

}*//accept success*

}*//if fd == sock*

**else** **if**(evs[i].data.fd **!=** sock **&&** \

evs[i].events **&** EPOLLIN)

{

**char** buf[1024];

ssize\_t s **=** read(evs[i].data.fd,buf,**sizeof**(buf) **-** 1);

**if**(s **>** 0)

{

buf[s] **=** 0;

printf("client say#%s",buf);

ev.data.fd **=** evs[i].data.fd;

ev.events **=** EPOLLOUT;

epoll\_ctl(epollfd,EPOLL\_CTL\_MOD, \

evs[i].data.fd,**&**ev);

}*//s > 0*

**else**

{

close(evs[i].data.fd);

epoll\_ctl(epollfd,EPOLL\_CTL\_DEL, \

evs[i].data.fd,NULL);

}

}*//fd != sock*

**else** **if**(evs[i].data.fd **!=** sock \

**&&** evs[i].events **&** EPOLLOUT)

{

**char** **\***msg **=** "HTTP/1.0 200 OK <\r\n\r\n<html><h1>yingying beautiful </h1></html>\r\n";

write(evs[i].data.fd,msg,strlen(msg));

close(evs[i].data.fd);

epoll\_ctl(epollfd,EPOLL\_CTL\_DEL, \

evs[i].data.fd,NULL);

}*//EPOLLOUT*

**else**

{

}

}*//for*

}*//default*

**break**;

}*//switch*

}*//while*

**return** 0;

}

1. **epoll\_create**函数创建一个epoll句柄，参数size表明内核要监听的描述符数量。调用成功时返回一个epoll句柄描述符，失败时返回-1。
2. **epoll\_ct**l函数注册要监听的事件类型。四个参数解释如下： epfd表示epoll句柄； op表示fd操作类型：**EPOLL\_CTL\_ADD**（注册新的fd到epfd中），**EPOLL\_CTL\_MOD**（修改已注册的fd的监听事件），**EPOLL\_CTL\_DEL**（从epfd中删除一个fd）；fd是要监听的描述符； event表示要监听的事件，epoll\_event结构体定义如下：

**struct** **epoll\_event** {

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

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

};

**typedef** **union** **epoll\_data** {

**void** **\***ptr;

**int** fd;

\_\_uint32\_t u32;

\_\_uint64\_t u64;

} epoll\_data\_t;

3**. epoll\_wait** 函数等待事件的就绪，成功时返回就绪的事件数目，调用失败时返回 -1，等待超时返回 0。maxevents告诉内核events的大小，timeout表示等待的超时事件。

**5、总结**

epoll是 Linux 目前大规模网络并发程序开发的首选模型。在绝大多数情况下性能远超 select和poll。目前流行的高性能web服务器Nginx正式依赖于epoll提供的高效网络套接字轮询服务。但是，在并发连接不高的情况下，多线程+阻塞 IO 方式可能性能更好。