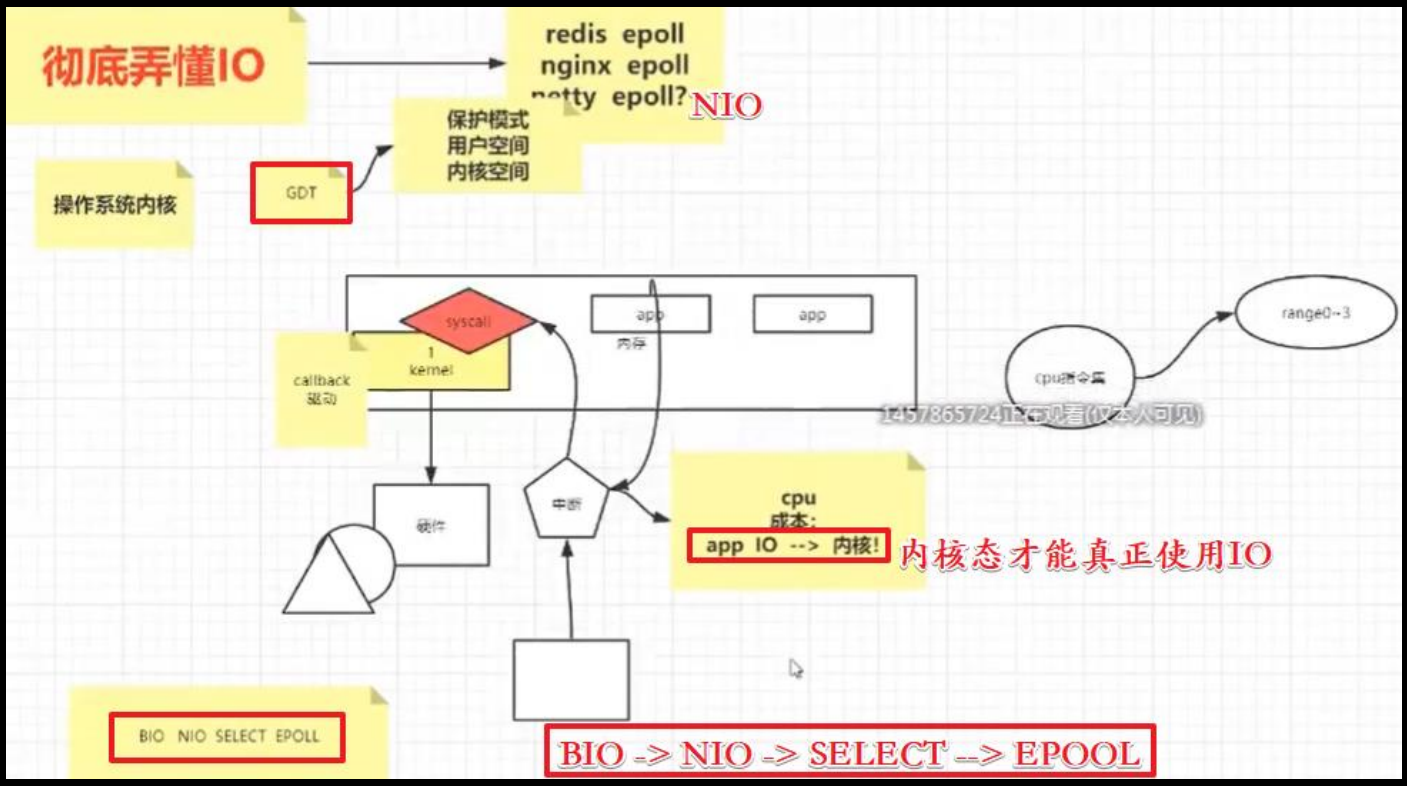
彻底弄懂 IO

IO 发展(BIO-NIO-SELECT-POOL-EPOOL)

https://www.bilibili.com/video/BV11z4y1Q7ns (值得反复观看)

关于 IO 发展的知识，先放在这里



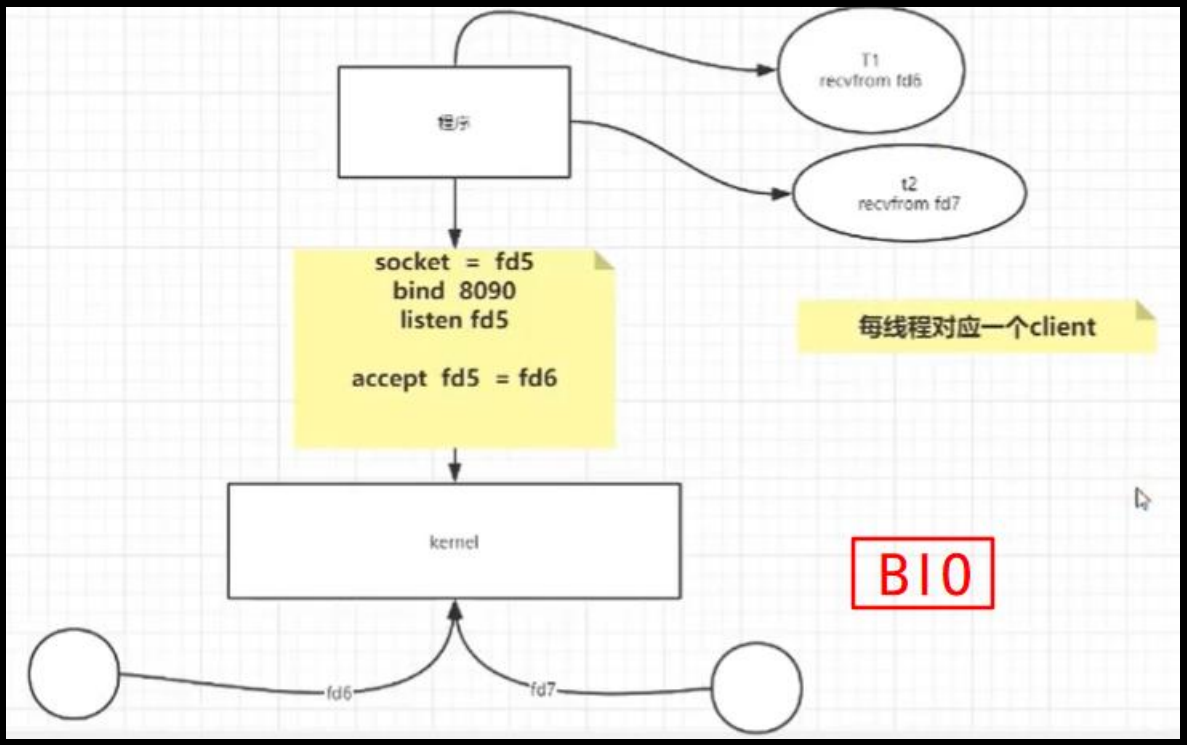
BIO → NIO → SELECT → EPOOL

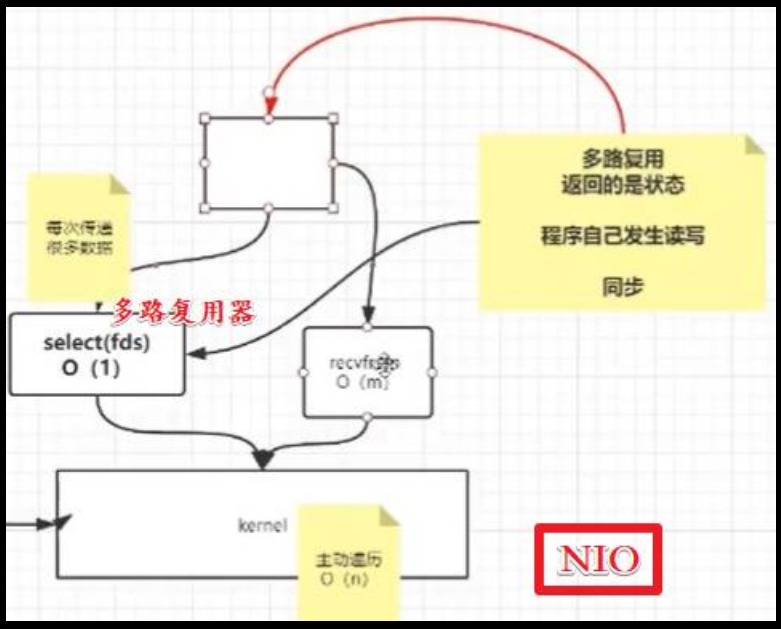
Linux 命令： **Strace**
抓取程序对系统有没有系统调用

```
strace -ff -o ./ooxx java TestSocket
```

Jps :java 进程

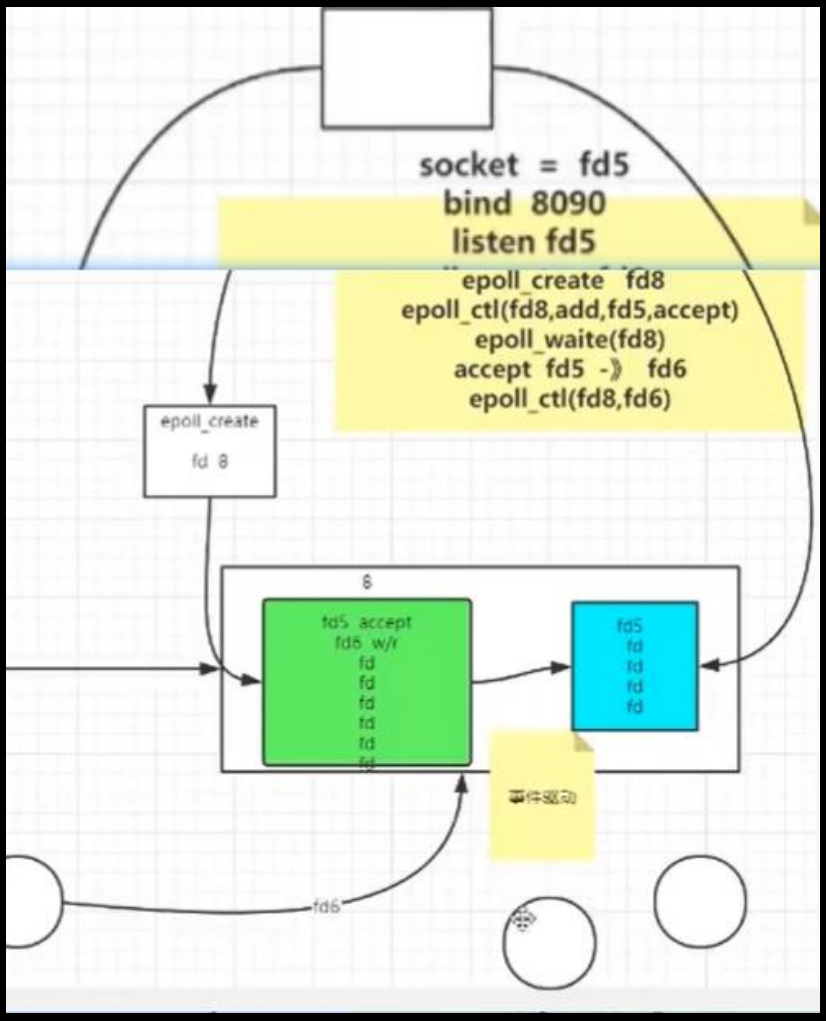
Nc 程序：作为客户端连接服务器， 类似 telnet



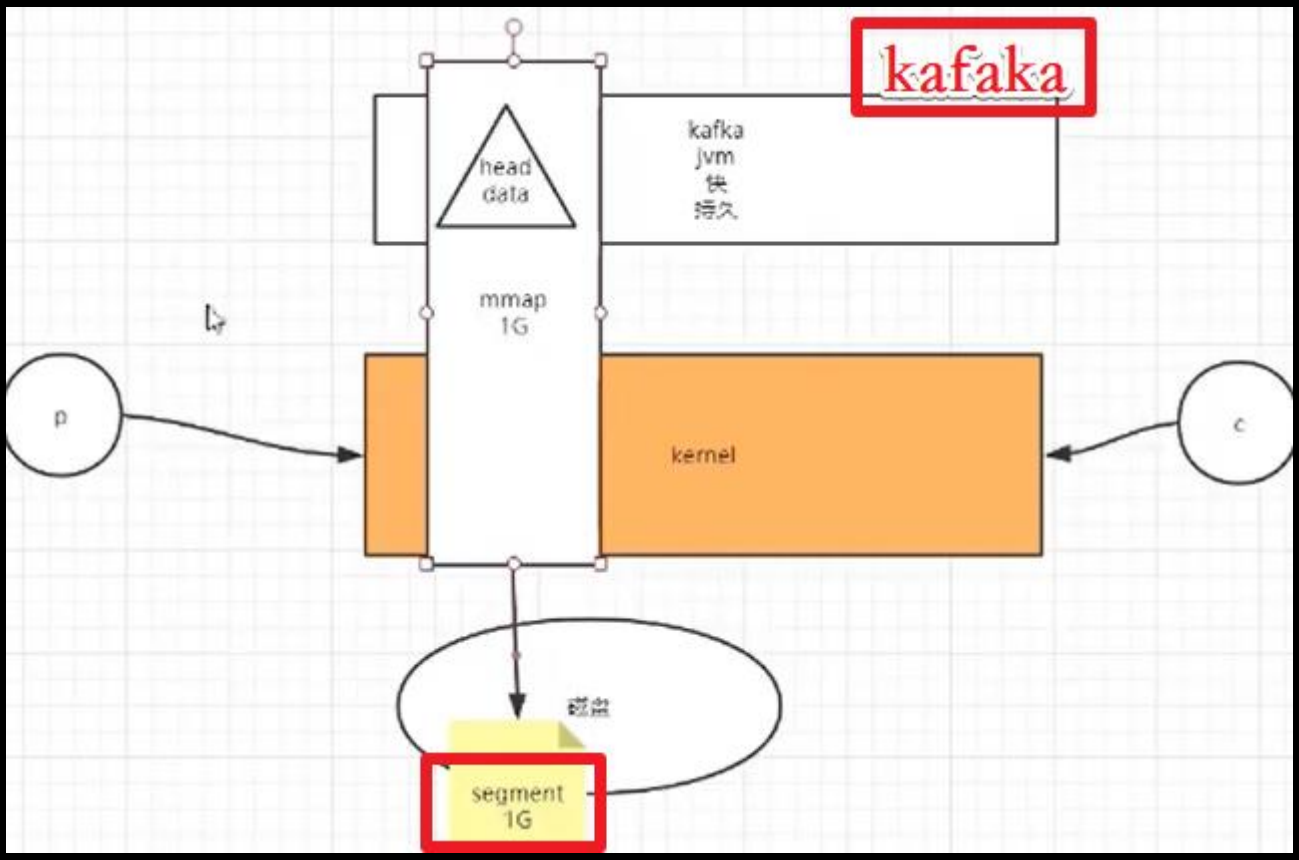


BIO 中 select 需要传入连接的客户端 fd，然后遍历，选择有读写事件的返回。
这个过程不好，于是产生了 epool

Epool (**event pool**):
开辟内存空间 A、B，用来存储 fds
来了一个连接(fd)，加入内存空间 A，当有读写事件产生时，产生硬件中断(**事件驱动**)，会将该连接 fd 加入内存空间 B，然后从 B 中全部取出，交给读写处理线程处理



通过 strace 工具可以查看到 Nginx\redis\kafka 就使用的 epool
Kafka 利用了 mmap



无论是 select、pool、epool，都叫做多路复用器

IO 是个细活
1. 零拷贝：前提是数据不需要加工
2. Mmap（Java 的 RandomAccessFile 会直接开辟 JVM 堆到磁盘文件的 buffer）
3. SendFile

epoll

epoll - I/O event notification facility

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：

```
typedef union epoll_data {
    void *ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
} epoll_data_t;

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。

关于 ET、LT 两种工作模式:
ET 模式仅当状态发生变化的时候才获得通知,这里所谓的状态的变化并不包括缓冲区中还有未处理的数据,也就是说,如果要采用 ET 模式,需要一直 read/write 直到出错为止,很多人反映为什么采用 ET 模式只接收了一部分数据就再也得不到通知了,大多因为这样;而 LT 模式是只要有数据没有处理就会一直通知下去的.

如何来使用 epoll 呢? 其实非常简单。
头文件#include <sys/epoll.h>
首先通过 create_epoll(int maxfds)来创建一个 epoll 的句柄
(在用完之后, 记得用 close()来关闭这个创建出来的 epoll 句柄)
之后在你的网络主循环里面, 每一帧的调用 epoll_wait(int epfd, epoll_event events, int max events, int timeout)来查询所有的网络接口, 看哪一个可以读, 哪一个可以写了。基本的语法为:
nfds = epoll_wait(kdpfd, events, maxevents, -1);
其中 kdpfd 为用 epoll_create 创建之后的句柄, events 是一个 epoll_event*的指针, 当 epoll_wait 这个函数操作成功之后, epoll_events 里面将储存所有的读写事件。max_events 是当前需要监听的所有 socket 句柄数。最后一个 timeout 是 epoll_wait 的超时, 为 0 的时候表示马上返回, 为-1 的时候表示一直等下去, 直到有事件范围, 为任意正整数的时候表示等这么长的时间, 如果一直没有事件, 则范围。一般如果网络主循环是单独的线程的话, 可以用-1 来等, 这样可以保证一些效率, 如果是和主逻辑在同一个线程的话, 则可以用 0 来保证主循环的效率。

epoll_wait 范围之后应该是一个循环, 遍利所有的事件。

几乎所有的 epoll 程序都使用下面的框架:

```
    for( ; ; )
    {
        nfds = epoll_wait(epfd,events,20,500);
        for(i=0;i<nfds;++i)
        {
            if(events[i].data.fd==listenfd) //有新的连接
            {
                connfd = accept(listenfd,(sockaddr *)&clientaddr, &clilen); //accept 这个连接
                ev.data.fd=connfd;
                ev.events=EPOLLIN|EPOLLET;
                epoll_ctl(epfd,EPOLL_CTL_ADD,connfd,&ev); //将新的 fd 添加到 epoll 的监听队列中
            }
            else if( events[i].events&EPOLLIN ) //接收到数据, 读 socket
            {
                n = read(sockfd, line, MAXLINE)) < 0    //读
                ev.data.ptr = md;     //md 为自定义类型, 添加数据
                ev.events=EPOLLOUT|EPOLLET;
                epoll_ctl(epfd,EPOLL_CTL_MOD,sockfd,&ev);//修改标识符, 等待下一个循环时发送数
据, 异步处理的精髓
            }
            else if(events[i].events&EPOLLOUT) //有数据待发送, 写 socket
            {
                struct myepoll_data* md = (myepoll_data*)events[i].data.ptr;    //取数据
                sockfd = md->fd;
                send( sockfd, md->ptr, strlen((char*)md->ptr), 0 );        //发送数据
                ev.data.fd=sockfd;
```

```
                ev.events=EPOLLIN|EPOLLET;
                epoll_ctl(epfd,EPOLL_CTL_MOD,sockfd,&ev); //修改标识符，等待下一个循环时接收数据
            }
            else
            {
                //其他的处理
            }
        }
    }
```

Select

https://www.cnblogs.com/Anker/archive/2013/08/14/3258674.html

IO 多路复用是指内核一旦发现进程指定的一个或者多个 IO 条件准备读取，它就通知该进程

select 函数

```
int select(int maxfdp1,fd_set *readset,fd_set *writeset,fd_set *exceptset,const struct timeval *timeout)
返回值：就绪描述符的数目，超时返回 0，出错返回-1
```

该函数准许进程指示内核等待多个事件中的任何一个发送，并只在有一个或多个事件发生或经历一段指定的时间后才唤醒。

1. 第一个参数 maxfdp1 指定待测试的描述字个数，它的值是待测试的最大描述字加 1（因此把该参数命名为 maxfdp1），描述字 0、1、2...maxfdp1-1 均将被测试。
2. 中间的三个参数 readset、writeset 和 exceptset 指定我们要让内核测试读、写和异常条件的描述字。如果对某一个的条件不感兴趣，就可以把它设为空指针。struct fd_set 可以理解为一个集合，这个集合中存放的是文件描述符，可通过以下四个宏进行设置：

```
        void FD_ZERO(fd_set *fdset);           //清空集合
        void FD_SET(int fd, fd_set *fdset);   //将一个给定的文件描述符加入集合之中
        void FD_CLR(int fd, fd_set *fdset);   //将一个给定的文件描述符从集合中删除
        int FD_ISSET(int fd, fd_set *fdset);   // 检查集合中指定的文件描述符是否可以读写
```

3. timeout 告知内核等待所指定描述字中的任何一个就绪可花多少时间。其 timeval 结构用于指定这段时间的秒数和微秒数。

```
    struct timeval{

            long tv_sec;   //seconds

            long tv_usec;  //microseconds

    };
```

这个参数有三种可能：

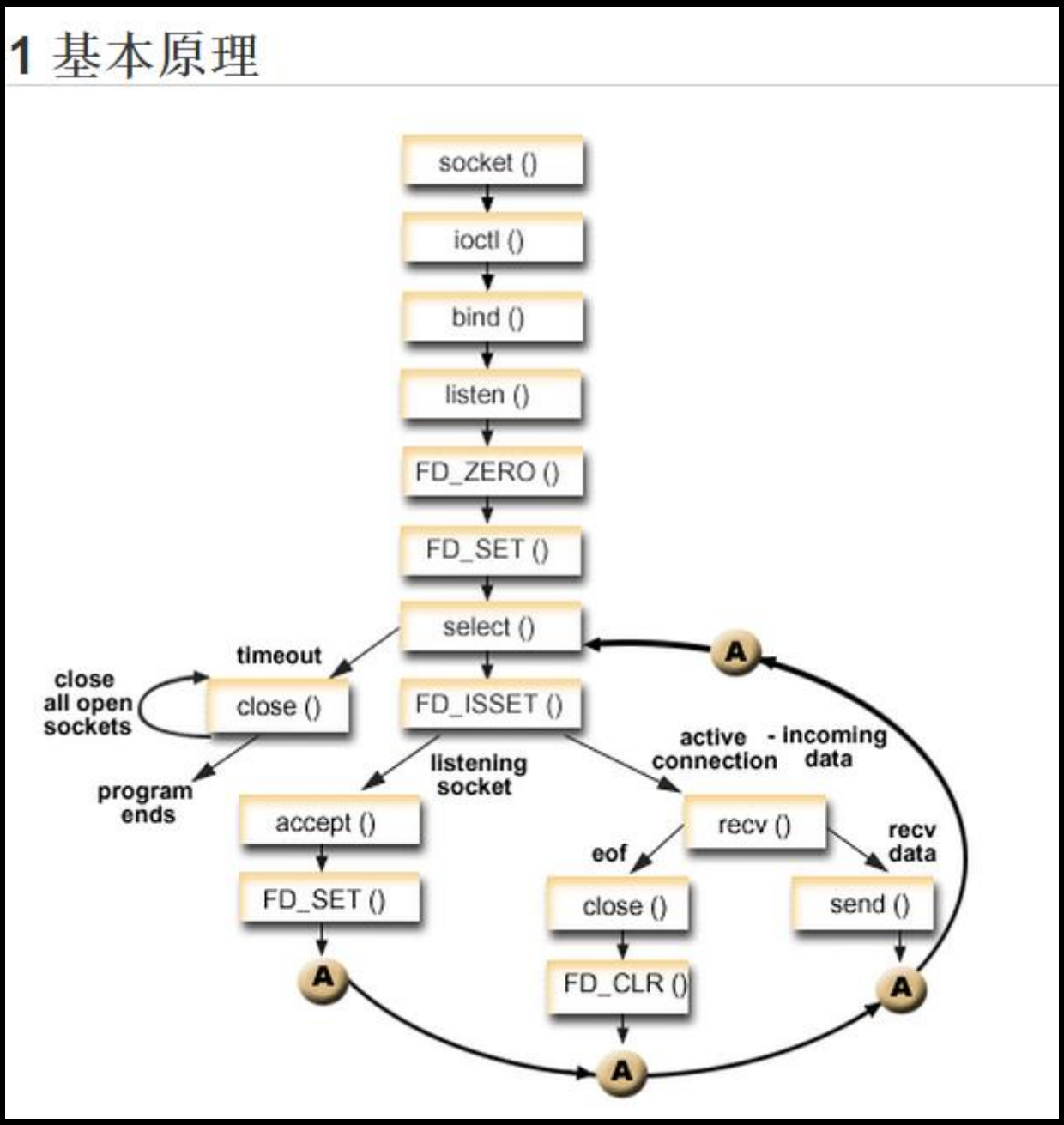
（1）永远等待下去：仅在有一个描述字准备好 I/O 时才返回。为此，把该参数设置为空指针 NULL。

（2）等待一段固定时间：在有一个描述字准备好 I/O 时返回，但是不超过由该参数所指向的 timeval 结构中指定的秒数和微秒数。

（3）根本不等待：检查描述字后立即返回，这称为轮询。为此，该参数必须指向一个 timeval 结构，而且其中的定时器值必须为 0。

原理图：

1 基本原理



poll

poll 的机制与 select 类似，与 select 在本质上没有多大差别，管理多个描述符也是进行轮询，根据描述符的状态进行处理，但是 poll 没有最大文件描述符数量的限制。poll 和 select 同样存在一个缺点就是，**包含大量文件描述符的数组被整体复制于用户态和内核的地址空间之间**，而不论这些文件描述符是否就绪，它的开销随着文件描述符数量的增加而线性增大。

poll 函数：

```
# include <poll.h>
int poll ( struct pollfd * fds, unsigned int nfds, int timeout);
```

```
struct pollfd {
int fd;         /* 文件描述符 */
short events;         /* 等待的事件 */
```

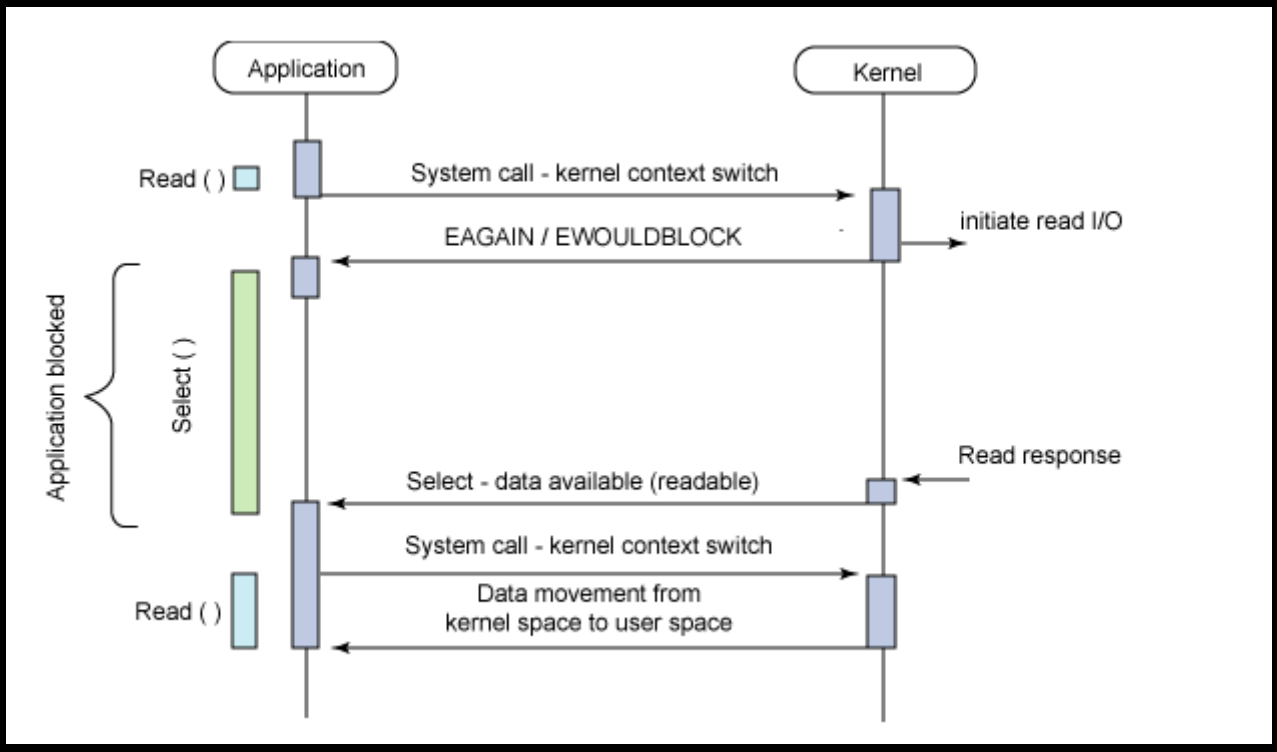
```
short revents;      /* 实际发生了的事件 */
} ;
```

select、poll、epoll

select，poll，epoll 都是 IO 多路复用的机制
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

select 实现

select 的调用过程如下所示：



（1）使用 copy_from_user 从用户空间拷贝 fd_set 到内核空间
（2）注册回调函数__pollwait
（3)遍历所有 fd，调用其对应的 poll 方法(对于 socket，这个 poll 方法是 sock_poll，sock_poll 根据情况会调用到 tcp_poll，udp_poll 或者 datagram_poll)
（4）以 tcp_poll 为例，其核心实现就是__pollwait，也就是上面注册的回调函数。
（5）__pollwait 的主要工作就是把 current（当前进程）挂到设备的等待队列中，不同的设备有不同的等待队列，对于 tcp_poll 来说，其等待队列是 sk->sk_sleep（注意把进程挂到等待队列中并不代表进程已经睡眠了）。在设备收到一条消息（网络设备）或填写完文件数据（磁盘设备）后，会唤醒设备等待队列上睡眠的进程，这时 current 便被唤醒了。
（6)poll 方法返回时会返回一个描述读写操作是否就绪的 mask 掩码，根据这个 mask 掩码给 fd_set 赋值。
（7）如果遍历完所有的 fd，还没有返回一个可读写的 mask 掩码，则会调用 schedule_timeout 是调用 select 的进程（也就是 current）进入睡眠。当设备驱动发生自身资源可读写后，会唤醒其等

待队列上睡眠的进程。如果超过一定的超时时间（schedule_timeout 指定），还是没人唤醒，则调用 select 的进程会重新被唤醒获得 CPU，进而重新遍历 fd，判断有没有就绪的 fd。
（8）把 fd_set 从内核空间拷贝到用户空间。

select 的几大缺点：
（1）每次调用 select，都需要把 fd 集合从用户态拷贝到内核态，这个开销在 fd 很多时会很大
（2）同时每次调用 select 都需要在内核遍历传递进来的所有 fd，这个开销在 fd 很多时也很大
（3）select 支持的文件描述符数量太小了，默认是 1024

poll 实现

poll 的实现和 select 非常相似，**只是描述 fd 集合的方式不同，poll 使用 pollfd 结构而不是 select 的 fd_set 结构**，其他的都差不多。

Epoll 实现

epoll 既然是对 select 和 poll 的改进，就应该能避免上述的三个缺点
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**。

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。

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，和 select 实现中的第 7 步是类似的）。

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。

总结

（1）select，poll 实现需要自己不断轮询所有 fd 集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而 epoll 其实也需要调用 epoll_wait 不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪 fd 放入就绪链表中，并唤醒在 epoll_wait 中进入睡眠的进程。虽然都要睡眠和交替，但是 select 和 poll 在“醒着”的时候要遍历整个 fd 集合，而 epoll 在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的 CPU 时间。这就是回调机制带来的性能提升。

（2）select，poll 每次调用都要把 fd 集合从用户态往内核态拷贝一次，并且要把 current 往设

备等待队列中挂一次，而 epoll 只要一次拷贝，而且把 current 往等待队列上挂也只挂一次（在 epoll_wait 的开始，注意这里的等待队列并不是设备等待队列，只是一个 epoll 内部定义的等待队列）。这也能节省不少的开销。