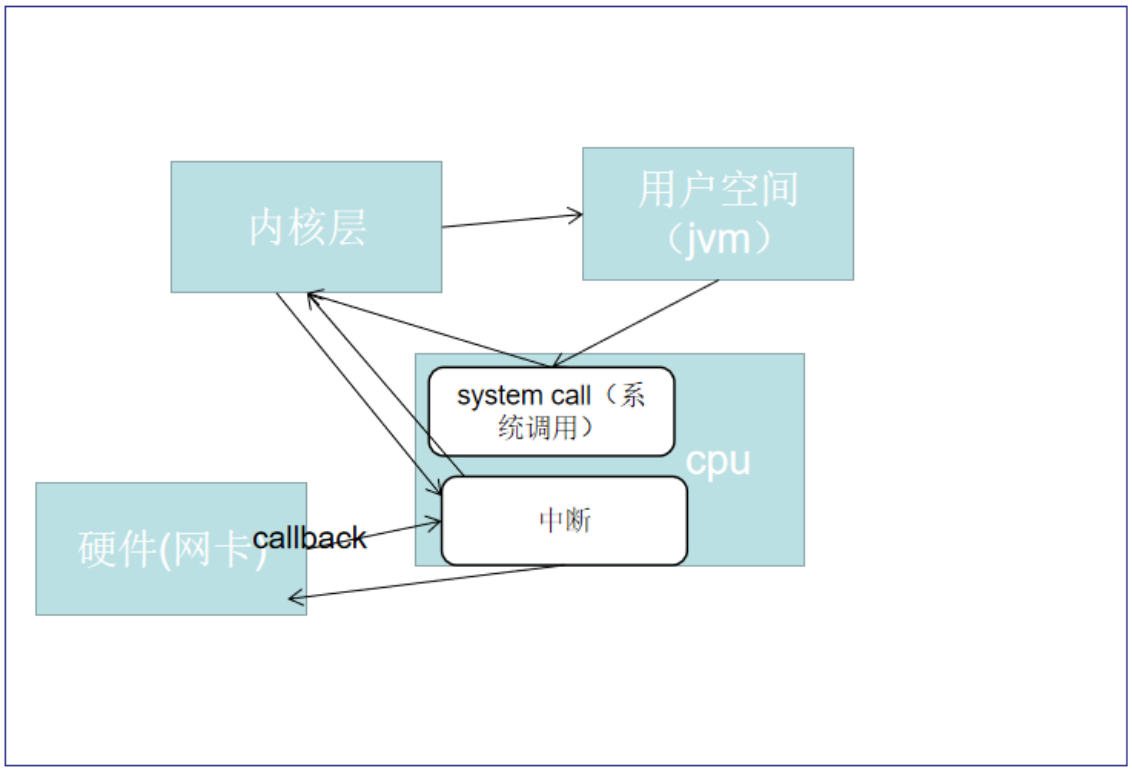
nio 和 epoll

nio是java中同步非阻塞io，epoll 是liunx 底层io 多路复用的一种技术，nio实现同步非阻塞是需要epoll底层支持，这篇文章我们就来搞懂epoll和nio的底层原理

计算机网络io的底层

在了解epoll的工作原理前我们需要先了解在linux系统中，一次网络io是怎么去完成的。我们来看下面这张图：



图中有两个需要理解的要点：

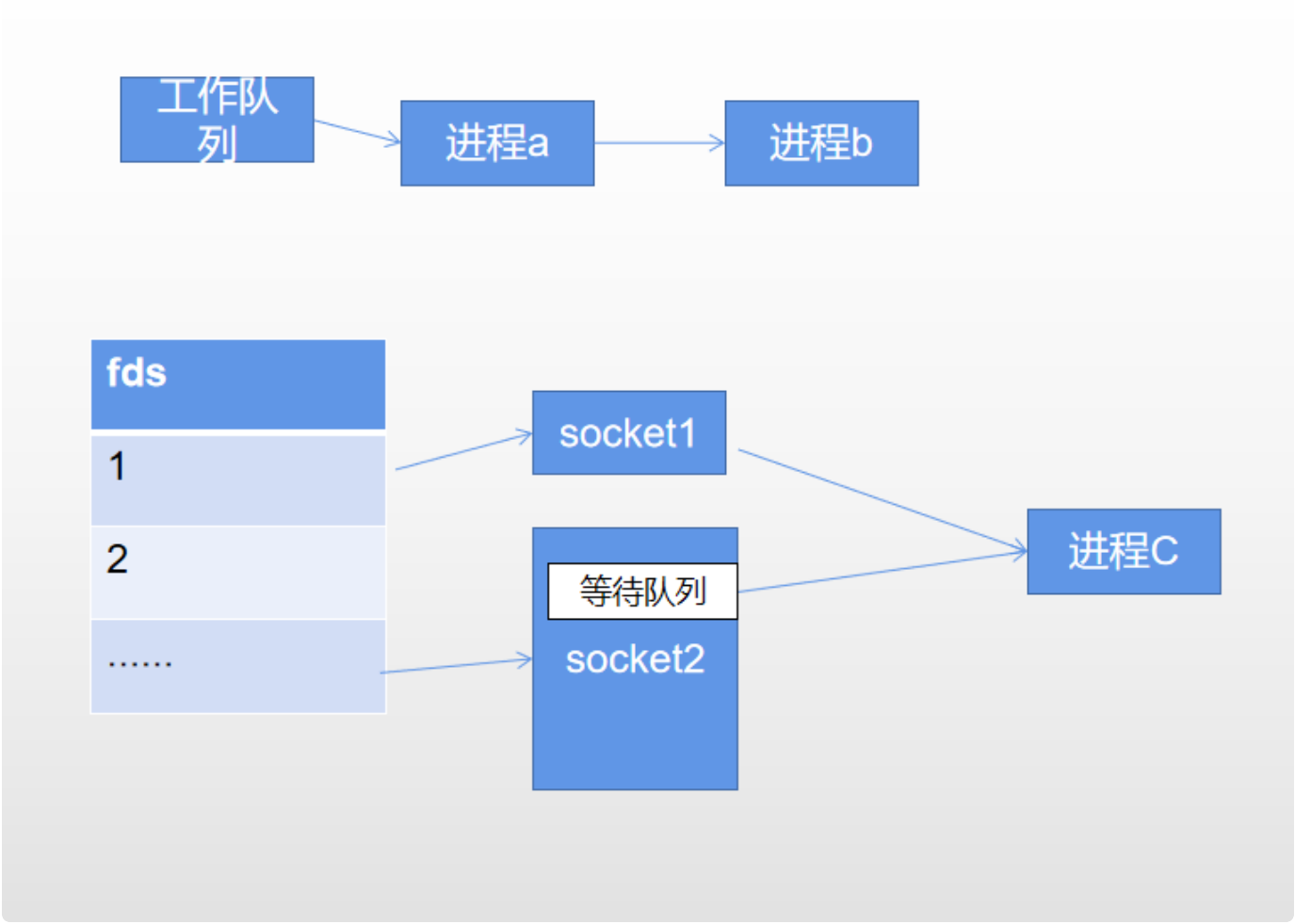
- 用户空间的程序（我们自己写的代码）是无法直接访问硬件层面的数据的，需要通过系统调用去将数据读取到内核空间再转到用户空间；
- 计算机执行程序时，会有优先级；一般而言，由硬件产生的信号需要cpu立马做出回应（不然数据可能就丢失），所以它的优先级很高。cpu理应中断掉正在执行的程序，去做出响应；当cpu完成对硬件的响应后，再重新执行用户程序。这种机制叫做中断；

当我们建立一个socket，用户程序通过系统调用向内核空间读取网卡的数据；网卡收到网络数据后，通过回调向cpu发出一个中断信号，操作系统便能得知有新数据到来，再通过中断程序去处理数据，然后将数据加载到内核空间，我们的程序再通过系统调用读取到网卡的数据。

上述的流程只是网络io的一个整体流程，我们想要明白为什么会有io多路复用这种机制出现还要进一步了解服务器建立socket的过程。当服务器建立一个socket，操作系统会创建一个由文件系统管理的socket对象。这个socket对象包含了发送缓冲区、接收缓冲区、等待队列等成员，等待队列存放所有需要等待该socket事件的进程。其伪代码如下：

```c
//创建socket
int s = socket(AF_INET, SOCK_STREAM, 0);
//绑定
bind(s, ...)
//监听
listen(s, ...)
//接受客户端连接
int c = accept(s, ...)
//等待接收客户端数据
recv(c, ...);
//将数据打印出来
printf(...)
```

recv是个阻塞方法，当程序运行到recv时，它会一直等待，直到接收到数据才往下执行。当recv阻塞，是不会消耗cpu资源的，而我们网络io往往会有很多个socket，那么有没有什么办法，用一个进程去等待多个recv，也就是io多路复用？如下图所示，当进程c执行到recv方法等待网络数据时会被放入等待队列，cpu不会去执行进程c，也就是进程c不会占用cpu资源，当网卡有数据到来的时候网卡就会通过中断来将进程A唤醒，并挂到工作队列中让cpu运行它。



图中的fd是文件描述符，一个socket就是一个文件，socket句柄就是一个文件描述符。

那这个怎么和传统的BIO联系起来呢，我们看bio建立socket的代码：

```
{
 ExecutorService executor = Excutors.newFixedThreadPollExecutor(100);//线程池

 ServerSocket serverSocket = new ServerSocket();
 serverSocket.bind(8088);
 while(!Thread.currentThread.isInturrupted()){//主线程死循环等待新连接到来
     Socket socket = serverSocket.accept();
     executor.submit(new ConnectIOnHandler(socket));//为新的连接创建新的线程
 }

class ConnectIOnHandler extends Thread{
    private Socket socket;
    public ConnectIOnHandler(Socket socket){
       this.socket = socket;
    }
    public void run(){
      while(!Thread.currentThread.isInturrupted()&&!socket.isClosed()){死循环处理读写事件
          String someThing = socket.read()....//读取数据
          if(someThing!=null){
             ......//处理数据
             socket.write()....//写数据
          }

      }
    }
```

传统的BIO里面socket.read()，如果TCP RecvBuffer里没有数据，函数会一直阻塞，直到收到数据，返回读到的数据。也就是说图中的进程如果进入了等待队列（recv方法阻塞），socket.read()也会跟着阻塞，直到有数据过来，它相当于直接调用操作系统recv方法。

select和epoll

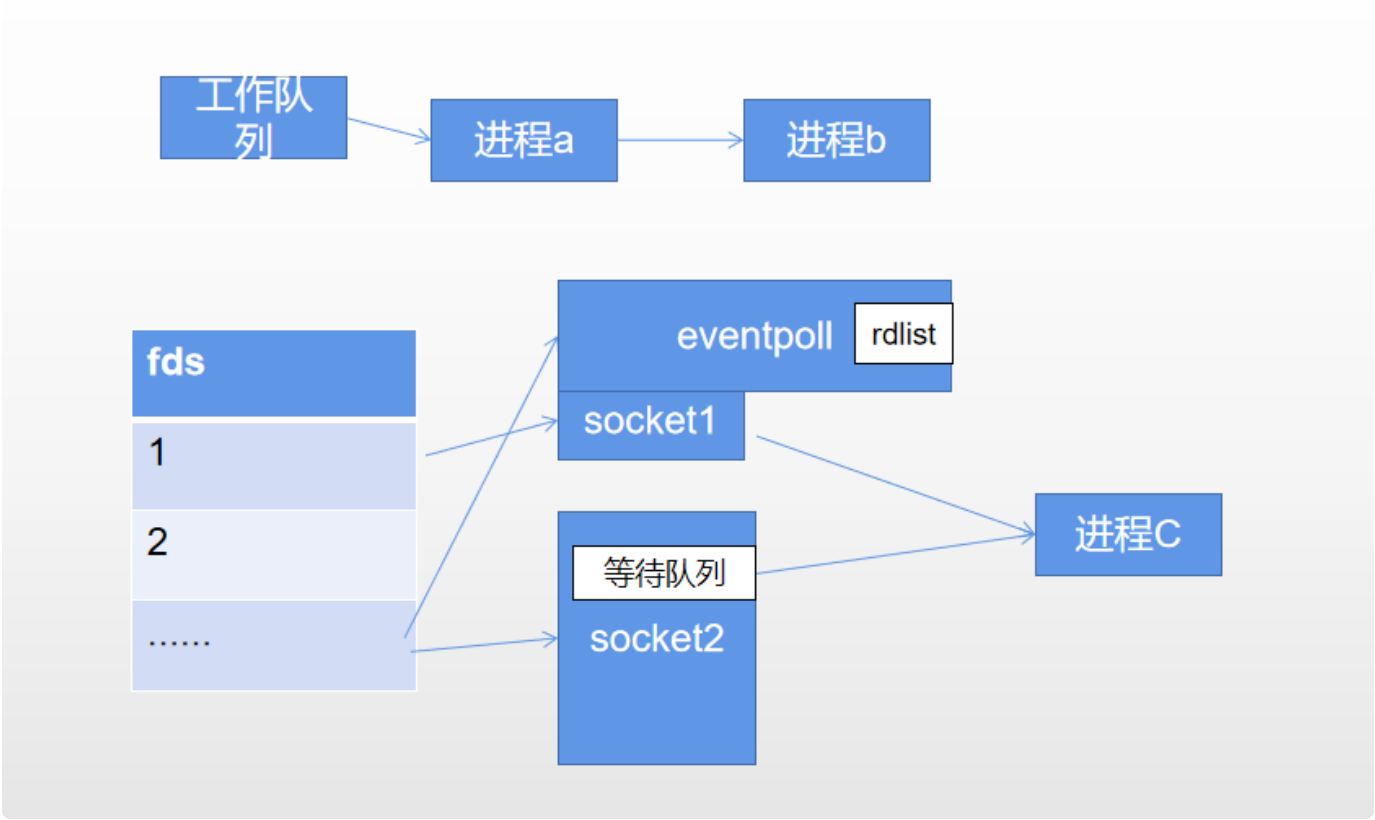
图中的机制还存在一个问题，前文提到过如果socket中有数据过来，网卡会给一个中断给到cpu，那么中断程序又是怎么去找到对应的fd的，这里面的工作流程是怎么样的？我们来看select是怎么做的：

数组fds存放着所有需要监视的socket，然后调用select，如果fds中的所有socket都没有数据，select会阻塞；当有一个socket接收到数据，发起一个中断，select返回，唤醒进程，将进程从所

有socket的等待队列中移除，遍历fds，通过FD_ISSET判断具体哪个socket收到数据，然后做出处理。该方法实现了一个进程监听多个socket，而不是傻傻的等recv返回（就像bio那样），但是这种方式有致命的缺点：

- 每次调用select都需要将进程加入到所有监视socket的等待队列，每次唤醒都需要从每个队列中移除。这里涉及了两次遍历，而且每次都要将整个fds列表传递给内核，有一定的开销。
- 进程被唤醒后，程序并不知道哪些socket收到数据，还需要遍历一次

为了解决上述缺点，于是有了epoll。在epoll中，内核维护一个“就绪列表”，引用收到数据的socket，避免了进程被唤醒后需要遍历才能找到socket，于是原来的模型就被改成了这样：



evenpoll 未 epoll 创建的对象，它内部维护了“就绪列表”；内核会将eventpoll添加到socket的等待队列中，当socket收到数据后，中断程序会操作eventpoll对象，而不是直接操作进程。当有socket就绪，发起中断，中断会将对应的socket引用存入rdlist中，并唤醒进程，进程只需要找rdlist就能快速处理就绪的socket；另：evenpoll通过红黑树保存监视的socket，而rdlist也并非直接引用socket，而是通过epitem间接引用，红黑树的节点也是epitem对象，图中简化了模型。

nio 多路复用

既然操作系统提供给我们这么好的机制，自然不能白白浪费，我们应用程序也想一个线程监听多个socket，谁有数据就起一个线程让它接收数据干活，干完活就扔到线程池里面去；为了更好的设计一套

多路复用的java框架，我们需要知道epoll的api:

- epoll_create：当某个进程调用epoll_create方法时，内核会创建一个eventpoll对象
- epoll_ctl：epoll 注册并监听事件的函数；
- epoll_wait：等待文件描述符epfd上的事件。

我们来看NIO是如何进行IO的：

Plain Text | 复制代码

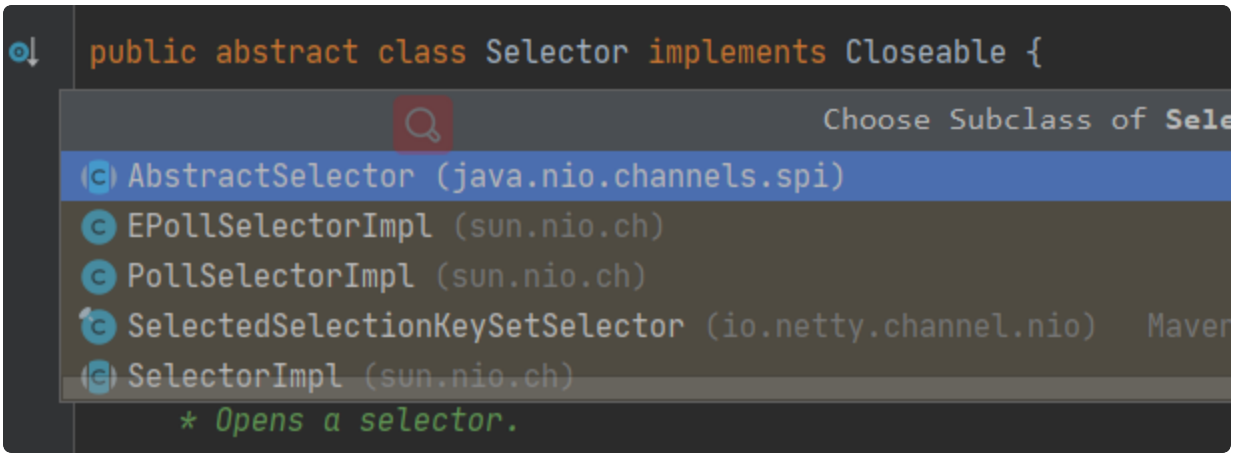
```
 public static void main(String[] args) throws IOException, InterruptedException {

        ServerSocketChannel serverSocket = ServerSocketChannel.open();
        serverSocket.socket().bind(new InetSocketAddress(8080));
        serverSocket.configureBlocking(false);
        Selector selector = Selector.open();
        serverSocket.register(selector, SelectionKey.OP_ACCEPT);
        System.out.println("服务启动成功");

        while (true) {
            selector.select();
            Set<SelectionKey> selectionKeys = selector.selectedKeys();
            Iterator<SelectionKey> iterator = selectionKeys.iterator();
            // 遍历SelectionKey对事件进行处理
            while (iterator.hasNext()) {
                SelectionKey key = iterator.next();
                iterator.remove();
                // 如果是OP_ACCEPT事件，则进行连接获取和事件注册
                if (key.isAcceptable()) {
                    ServerSocketChannel server = (ServerSocketChannel) key.channel();
                    SocketChannel socketChannel = server.accept();
                    socketChannel.configureBlocking(false);
                    // 这里只注册了读事件，如果需要给客户端发送数据可以注册写事件
                    socketChannel.register(selector, SelectionKey.OP_READ);
                    System.out.println("客户端连接成功");
                }
                // 如果是OP_READ事件，则进行读取和打印
                if (key.isReadable()) {
                    SocketChannel socketChannel = (SocketChannel) key.channel();
                    ByteBuffer byteBuffer = ByteBuffer.allocate(128);
                    int read = socketChannel.read(byteBuffer);
                    // 如果有数据，把数据打印出来
                    if (read > 0) {
                        System.out.println("接收到消息：" + new String(byteBuffer.array()));
                    } else if (read == -1) {
                        // 如果客户端断开连接，关闭Socket
                        System.out.println("客户端断开连接");
                        socketChannel.close();
                    }
                }
```

```
41                 }
42
43             }
44
45         }
```

然后查看一下`Selector`类linux系统下的的实现



它包括了 `PollSelectorImpl` ， `EPollSelectorImpl` , `SelectedSelectionKeySetSelector` 实现类，我们知道io多路复用在linux 中有Select,Poll Epoll 三种实现，自然就对应了三个类，我们重点关注`EPollSelectorImpl`的源码，在NIO使用示例中 `Selector` 是通过 `Selector.open()` 创建的，我们重点看看如何创建的这个 `Selector` :

```
public class EPollSelectorProvider
    extends SelectorProviderImpl
{
    public AbstractSelector openSelector() throws IOException {
        return new EPollSelectorImpl(this);
    }

    public Channel inheritedChannel() throws IOException {
        return InheritedChannel.getChannel();
    }
}


EPollSelectorImpl(SelectorProvider sp) throws IOException {
        super(sp);

        this.epfd = EPoll.create();
        this.pollArrayAddress = EPoll.allocatePollArray(NUM_EPOLLEVENTS);

        try {
            this.eventfd = new EventFD();
            IOUtil.configureBlocking(IOUtil.newFD(eventfd.efd()), false);
        } catch (IOException ioe) {
            EPoll.freePollArray(pollArrayAddress);
            FileDispatcherImpl.closeIntFD(epfd);
            throw ioe;
        }

        // register the eventfd object for wakeups
        EPoll.ctl(epfd, EPOLL_CTL_ADD, eventfd.efd(), EPOLLIN);
    }
```

我们看到在 `EPollSelectorImpl` 构造器中调用了 `EPoll.create()` 这个便是epoll的api。我们也可以顺便看看`EPoll`类的源码：

```
class EPoll {
    private EPoll() { }
    static native int create() throws IOException;

    static native int ctl(int epfd, int opcode, int fd, int events);

    static native int wait(int epfd, long pollAddress, int numfds, int timeout)
        throws IOException;
}
```

通过idea 可以看到 `sun.nio.ch.EPollSelectorImpl#doSelect` 调用了 `EPoll.wait` ：

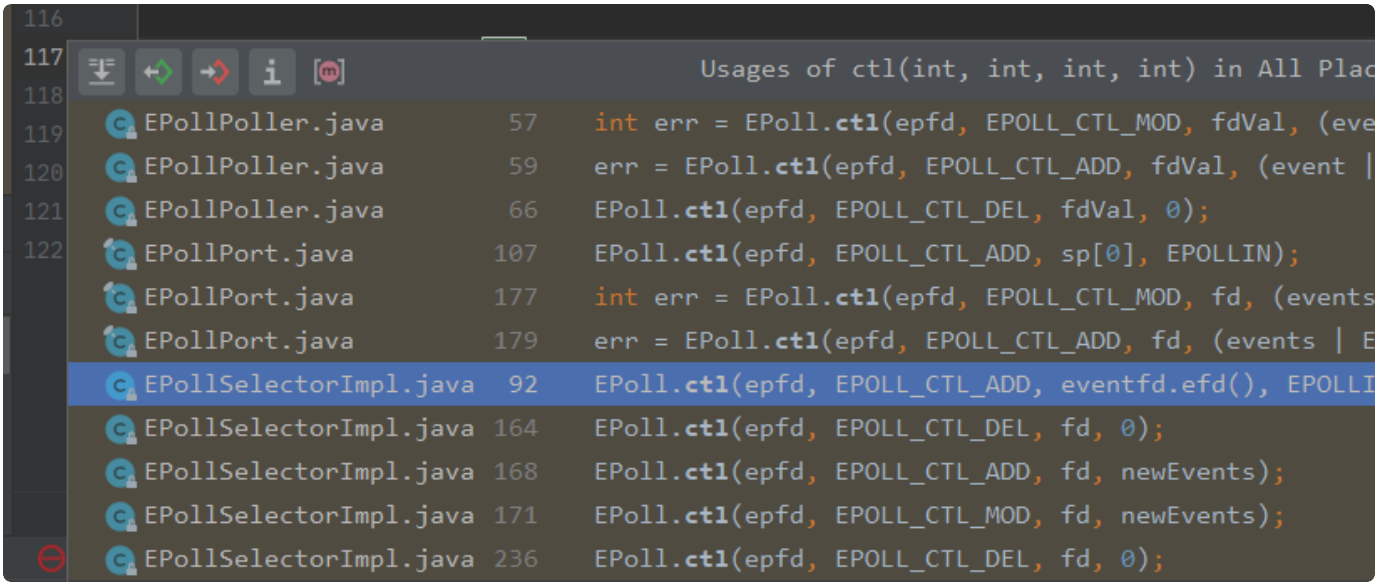
```
@Override
protected int doSelect(Consumer<SelectionKey> action, long timeout)
    throws IOException
{
    assert Thread.holdsLock( obj: this);

    // epoll_wait timeout is int
    int to = (int) Math.min(timeout, Integer.MAX_VALUE);
    boolean blocking = (to != 0);
    boolean timedPoll = (to > 0);

    int numEntries;
    processUpdateQueue();
    processDeregisterQueue();
    try {
        begin(blocking);

        do {
            long startTime = timedPoll ? System.nanoTime() : 0;
            long comp = B  File is read-only  g);
            try {
                numEntries = EPoll.wait(epfd, pollArrayAddress, NUM_EPOLLEVENTS, to);
            } finally {
                Blocker.end(comp);
            }
```

`EPoll.ctl` 在多处调用：



主要调用方法是 `sun.nio.ch.EPollSelectorImpl#processUpdateQueue`：

```
/**
 * Process changes to the interest ops.
 */
private void processUpdateQueue() {
    assert Thread.hol
                    File is read-only

    synchronized (updateLock) {
        SelectionKeyImpl ski;
        while ((ski = updateKeys.pollFirst()) != null) {
            if (ski.isValid()) {
                int fd = ski.getFDVal();
                // add to fdToKey if needed
                SelectionKeyImpl previous = fdToKey.putIfAbsent(fd, ski);
                assert (previous == null) || (previous == ski);

                int newEvents = ski.translateInterestOps();
                int registeredEvents = ski.registeredEvents();
                if (newEvents != registeredEvents) {
                    if (newEvents == 0) {
                        // remove from epoll
                        EPoll.ctl(epfd, EPOLL_CTL_DEL, fd,  events: 0);
                    } else {
                        if (registeredEvents == 0) {
                            // add to epoll
                            EPoll.ctl(epfd, EPOLL_CTL_ADD, fd, newEvents);
                        } else {
                            // modify events
                            EPoll.ctl(epfd, EPOLL_CTL_MOD, fd, newEvents);
                        }
```

这个方法在 `Selector.Select` 中被调用

```java
/**
 * Selects a set of keys whose corresponding channels are ready for I/O
 * operations.
 *
 * <p> This method performs a blocking <a href="#selop">selection
 * operation</a>.  It returns only after at least one channel is selected,
 * this selector's {@link #wakeup wakeup} method is invoked, or the current
 * thread is interrupted, whichever comes first.  </p>
 *
 * @return  The number of keys, possibly zero, whose ready-operation sets
 *          now indicate readiness for at least one category of operations
 *          for which the channel was not previously detected to be ready
 *
 * @throws  IOException
 *          If an I/O error occurs
 *
 * @throws  ClosedSelectorException
 *          If this selector is closed
 */
public abstract int select() throws IOException;
```

大概意思就是选择一个准备好的通道。

通过上述的源码，我们发现NIO其实就是对linux多路复用的一个封装。离上一次写公众号已经很久了，懒了懒了。