Java NIO 哪些不容易注意到的事

by：黄金档 请关注我们！

java NIO 的实现中，有不少细节点非常有学习意义的，就好比下面的三个点：

1） Selector 的 wakeup 原理是什么？是如何实现的？

2） Channel 的 close 会做哪些事？

3） 会什么代码中经常出现 begin()和 end()这一对儿？

本文虽然针对这几个点做了点分析，不能算是非常深刻，要想达到通透的地步，看来还得经过实战的洗练。

1、 wakeup()

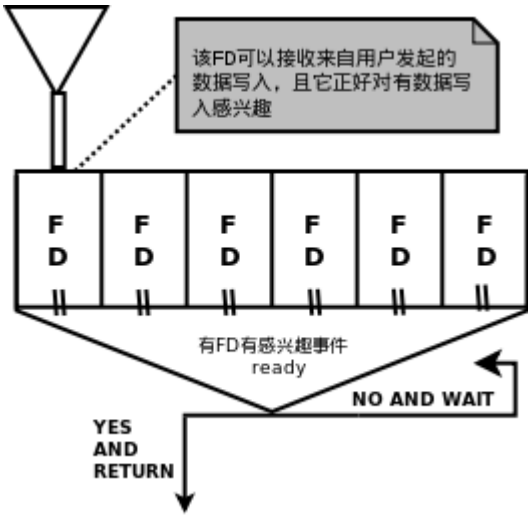
准确来说，应该是 Selector 的 wakeup()，即 Selector 的唤醒，为什么要有这个唤醒操作呢？那还得从 Selector 的选择方式来说明，前文已经总结过 Selector 的选择方式有三种：select()、select(timeout)、selectNow()。

selectNow 的选择过程是非阻塞的，与 wakeup 没有太大关系。

select(timeout)和 select()的选择过程是阻塞的，其他线程如果想终止这个过程，就可以调用 wakeup 来唤醒。

wakeup 的原理

既然 Selector 阻塞式选择因为找到感兴趣事件 ready 才会返回(排除超时、中断)，就给它构造一个感兴趣事件 ready 的场景即可。下图可以比较形象的形容 wakeup 原理：



Selector 管辖的 FD(文件描述符，linux 即为 fd，对应一个文件，windows 下对应一个句柄；每个可选择 Channel 在创建的时候，就生成了与其对应的 FD，Channel 与 FD 的联系见另一篇)中包含某一个 FD A， A 对数据可读事件感兴趣，当往图中漏斗端放入(写入)数据，数据会流进 A，于是 A 有感兴趣事件 ready，最终，select 得到结果而返回。

wakeup 在 Selector 中的定义如下：

```
public abstract Selector wakeup();
```

下面结合上图来追寻 wakeup 的实现：

linux 下 Selector 默认实现为 PollSelectorImpl，当内核版本大于 2.6 时，实现为 EPollSelectorImpl，仅看这两者的 wakeup 方法，代码似乎完全一样：

```
public Selector wakeup() {
    synchronized (interruptLock) {
        if (!interruptTriggered) {
            pollWrapper.interrupt();
            interruptTriggered = true;
        }
```

```
        }
        return this;
    }
```

window 下 Selector 的实现为 WindowsSelectorImpl，其 wakeup 实现如下：

```
    public Selector wakeup() {
        synchronized (interruptLock) {
            if (!interruptTriggered) {
                setWakeupSocket();
                interruptTriggered = true;
            }
        }
        return this;
    }
```

其中 interruptTriggered 为中断已触发标志，当 pollWrapper.interrupt()之后，该标志即为 true 了；得益于这个标志，连续两次 wakeup，只会有一次效果。

对比上图及上述代码，其实 pollWrapper.interrupt()及 setWakeupSocket()就是图中的往漏斗中倒水的过程，不管 windows 也好，linux 也好，它们 wakeup 的思想是完全一致的，不同的地方就在于实现的细节了，例如上图中漏斗与通道的链接部分，linux 下是采用管道 pipe 来实现的，而 windows 下是采用两个 socket 之间的通讯来实现的，它们都有这样的特性：1）都有两个端，一个是 read 端，一个是 write 端，windows 中两个 socket 也是一个扮演 read 的角色，一个扮演 write 的角色；2）当往 write 端写入数据，则 read 端即可以收到数据；从它们的特性可以看出，它们是能够胜任这份工作的。

如果只想理解 wakeup 的原理，看到这里应该差不多了，不过，下面，想继续深入一下，满足更多人的好奇心。

先看看 linux 下 PollSelector 的具体 wakeup 实现，分阶段来介绍：

1）准备阶段

PollSelector 在构造的时候，就将管道 pipe，及 wakeup 专用 FD 给准备好，可以看一下它的实现：

```
    PollSelectorImpl(SelectorProvider sp) {
        super(sp, 1, 1);
        int[] fdes = new int[2];
        IOUtil.initPipe(fdes, false);
        fd0 = fdes[0];
        fd1 = fdes[1];
        pollWrapper = new PollArrayWrapper(INIT_CAP);
        pollWrapper.initInterrupt(fd0, fd1);
        channelArray = new SelectionKeyImpl[INIT_CAP];
    }
```

IOUtil.initPipe，采用系统调用 pipe(int fd[2])来创建管道，fd[0]即为 ready 端，fd[1]即为 write 端。

另一个需要关注的点就是 pollWrapper.initInterrupt(fd0, fd1)，先看一下它的实现：

```
    void initInterrupt(int fd0, int fd1) {
        interruptFD = fd1;
        putDescriptor(0, fd0);
        putEventOps(0, POLLIN);
        putReventOps(0, 0);
    }
```

可以看到，initInterrupt 在准备 wakeup 专用 FD，因为 fd0 是 read 端 fd，fd1 是 write 端 fd：

interruptFD 被初始化为 write 端 fd；

putDescriptor(0, fd0)初始化 pollfd 数组中的第一个 pollfd，即指 PollSelector 关注的第一个 fd，即为 fd0；

putEventOps(0, POLLIN)初始化 fd0 对应 pollfd 中的 events 为 POLLIN,即指 fd0 对可读事件感兴趣；
putReventOps(0, 0)只是初始化一下 fd0 对应的 pollfd 中的 revents;

2）执行阶段

有了前面的准备工作，就看 PollArrayWrapper 中的 interrupt()实现：

```
public void interrupt() {
    interrupt(interruptFD);
}
```

interrupt 是 native 方法，它的入参 interruptFD 即为准备阶段管道的 write 端 fd，对应于上图，其实就是漏斗端，因此，就是不看其实现，也知道它肯定扮演着倒水的这个动作，看其实现：

```
JNIEXPORT void JNICALL
Java_sun_nio_ch_PollArrayWrapper_interrupt(JNIEnv *env, jobject this, jint fd)
{
    int fakebuf[1];
    fakebuf[0] = 1;
    if (write(fd, fakebuf, 1) < 0) {
        JNU_ThrowIOExceptionWithLastError(env,
                                          "Write to interrupt fd failed");
    }
}
```

可以看出，interrupt(interruptFD)是往管道的 write 端 fd1 中写入一个字节(write(fd, fakebuf, 1))。
是的，只需要往 fd1 中写入一个字节，fd0 即满足了可读事件 ready，则 Selector 自然会因为有事件 ready 而中止阻塞返回。

EPollSelector 与 PollSelector 相比，其 wakeup 实现就只有 initInterrupt 不同，它的实现如下：

```
void initInterrupt(int fd0, int fd1) {
    outgoingInterruptFD = fd1;
    incomingInterruptFD = fd0;
    epollCtl(epfd, EPOLL_CTL_ADD, fd0, EPOLLIN);
}
```

epfd 之前的篇章里已经讲过，它是通过 epoll_create 创建出来的 epoll 文件 fd，epollCtl 调用内核 epoll_ctl 实现了往 epfd 上添加 fd0，且其感兴趣事件为可读(EPOLLIN)，

因此可以断定，EPollSelector 与 PollSelector 的 wakeup 实现是一致的。

因为之前一直专注与分析 linux 下的 Java NIO 实现，忽略了 windows 下的选择过程等，这里突然讲解其 wakeup 实现似乎很突兀，所以打算后面专门起一篇来介绍 windows 下的 NIO 实现，这里我们只需要理解 wakeup 原理，甚至自己去看看其 wakeup 实现，应该也没什么难度。

关于 wakeup，这里还有两个疑问：
为什么 wakeup 方法返回 Selector？
windows 下也是有 pipe 的，为什么使用 socket 而不是使用 pipe 来实现 wakeup 的？

2、close()

close()操作限于通道，而且还是实现了 InterruptibleChannel 接口的通道，例如 FileChannel 就没有 close 操作。
在分析 close()具体实现之前，我们先得理解为什么要有 close()这个操作：
一个可选择的通道，在创建之初会生成一个 FileDescriptor，linux 下即为 fd，windows 下即为句柄，这些都是系统资源，不能无限占用，当在不使用的时候，就应该将其释放，close 即是完成这个工作的。
抽象类 AbstractInterruptibleChannel 实现了 InterruptibleChannel 接口，而 SelectableChannel 继承自 AbstractInterruptibleChannel，因此，可选择的通道同时也是可以 close 的。
AbstractInterruptibleChannel 的 close 实现如下：

```
public final void close() throws IOException {
synchronized (closeLock) {
    if (!open)
    return;
    open = false;
    implCloseChannel();
}
}
```

看来具体关闭逻辑就在 implCloseChannel()中了，于是再看 AbstractSelectableChannel：

```
protected final void implCloseChannel() throws IOException {
implCloseSelectableChannel();
synchronized (keyLock) {
       int count = (keys == null) ? 0 : keys.length;
    for (int i = 0; i < count; i++) {
    SelectionKey k = keys[i];
    if (k != null)
       k.cancel();
    }
}
}
```

先看 synchronized 同步块，它将当前通道保存的 SelectionKey 全部 cancel，意思就是说，当前通关闭了，与它相关的所有 SelectionKey 都没有意义了，所以要全部取消掉，之前讲解 cancel 过程已经说明了，cancel 操作只是将 SelectionKey 加入对应选择器的 cancelKeys 集合中，在下次正式选择开始的时候再一一清除；

这么看来，还是应该追究一下 implCloseSelectableChannel()的实现了，下面分别从 ServerSocketChannel 和 SocketChannel 实现出发：

先看 ServerSocketChannelImpl，

```
protected void implCloseSelectableChannel() throws IOException {
synchronized (stateLock) {
    nd.preClose(fd);
    long th = thread;
    if (th != 0)
    NativeThread.signal(th);
    if (!isRegistered())
    kill();
}
}
```

出现了两个很奇怪的东西，看来要完全弄懂这段代码，是得好好分析一下它们了，它们是：NativeDispatcher nd 和 NativeThread；

如果已经对 linux 信号机制非常熟悉，应该很容易猜测到 NativeThread.signal(th)在做什么，

是的，它在唤醒阻塞的线程 th，下面我们来看看它是如何做到的：

NativeThread 类非常简单，几乎全是 native 方法：

```
 class NativeThread {
    static native long current();
    static native void signal(long nt);
    static native void init();
    static {
       Util.load();
       init();
    }
```

```
}
```

在看其本地实现：

```
//自定义中断信号，kill -l
#define INTERRUPT_SIGNAL (__SIGRTMAX - 2)
//自定义的信号处理函数，当前函数什么都不做
static void
nullHandler(int sig)
{
}
#endif
//NativeThread.init()的本地实现，可以看到它用到了 sigaction
//sigaction 用来 install 一个信号
JNIEXPORT void JNICALL
Java_sun_nio_ch_NativeThread_init(JNIEnv *env, jclass cl)
{
#ifdef __linux__
    sigset_t ss;
    // 以下这段代码是常见的信号安装过程
    // 讲解这段代码的目的只是为了让大家理解 NativeThread.signal
    // 的工作原理，故很多细节就简单带过了
    struct sigaction sa, osa;
    // sa 用于定制信号 INTERRUPT_SIGNAL 的处理方式的
    // 如 sa_handler = nullHandler 即用来指定信号处理函数的
    // 即线程收到信号时，为执行这个函数，nullHandler 是个空壳
    // 函数，所以它什么都不做
    // 不用理解 sa_flags 各个标识代表什么
    // sigemptyset 顾名思义，它是初始化 sigaction 的 sa_mask 位
    // sigaction(INTERRUPT_SIGNAL, &sa, &osa)执行后
    // 如果成功，则表示 INTERRUPT_SIGNAL 这个信号安装成功了
    // 为什么要有这个 init 呢，其实不用这不操作也许不会有问题
    // 但因为不能确保 INTERRUPT_SIGNAL 没有被其他线程 install
    // 过，如果 sa_handler 对应函数不是空操作，则在使用这个信号
    // 时会对当前线程有影响
    sa.sa_handler = nullHandler;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    if (sigaction(INTERRUPT_SIGNAL, &sa, &osa) < 0)
    JNU_ThrowIOExceptionWithLastError(env, "sigaction");
#endif
}

JNIEXPORT jlong JNICALL
Java_sun_nio_ch_NativeThread_current(JNIEnv *env, jclass cl)
{
#ifdef __linux__
    // pthread_self()即是获取当前线程 ID，它与 getpid()是不同的
    // 具体细节没有研究
    return (long)pthread_self();
#else
```

```
    return -1;
#endif
}

JNIEXPORT void JNICALL
Java_sun_nio_ch_NativeThread_signal(JNIEnv *env, jclass cl, jlong thread)
{
#ifdef __linux__
    //这个就是最关键的 signal 实现了，可以看到，它调用了 pthread 库的 pthread_kill
    //像 thread 线程发送一个 INTERRUPT_SIGNAL 信号，这个信号就是在 init 中 install
    //的，对应的处理函数是空函数，也就是说，往 thread 线程发送一个信号，如果该线程处于
    //阻塞状态，则会因为受到信号而终止阻塞，而如果处于非阻塞，则无影响
    if (pthread_kill((pthread_t)thread, INTERRUPT_SIGNAL))
    JNU_ThrowIOExceptionWithLastError(env, "Thread signal failed");
#endif
}
```

Java 的 NativeThread 做静态初始化时已经执行了 init，也就是说 INTERRUPT_SIGNAL 信号已经被安装，而 ServerSocketChannelImpl 中的 thread 有两种可能值，见代码段：

```
    try {
      begin();
      if (!isOpen())
        return null;
      thread = NativeThread.current();
      for (;;) {
          n = accept0(this.fd, newfd, isaa);
          if ((n == IOStatus.INTERRUPTED) && isOpen())
          continue;
          break;
      }
      } finally {
        thread = 0;
        end(n > 0);
        assert IOStatus.check(n);
    }
```

try 的内部，for 循环之前，thread 被复制为 NativeThread.current() 即为当前线程 id；finally 时 thread 又被修改回 0，因此在 implCloseSelectableChannel 才有这样一段：

```
   if (th != 0)
       NativeThread.signal(th);
```

NativeThread.signal(th) 通过像当前线程发送 INTERRUPT_SIGNAL 信号而确保 th 线程没有被阻塞，即如果阻塞就停止阻塞。

为了让大家更好的理解信号的安装和使用，下面写了一个小程序来说明：

```
#include <pthread.h>
#include <stdio.h>
#include <sys/signal.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
```

```c
#define NUMTHREADS 3
#define INTERRUPT_SIGNAL (SIGRTMAX - 2)

void *threadfunc(void *parm)
{
    pthread_t           self = pthread_self();
    int                rc;
    printf("Thread 0x%.8x %.8x entered\n", self);
    errno = 0;
    rc = sleep(30);
    if (rc != 0 && errno == EINTR) {
        printf("Thread 0x%.8x %.8x got a signal delivered to it\n",
               self);
        return NULL;
    }
    printf("Thread 0x%.8x %.8x did not get expected results! rc=%d, errno=%d\n",
           self, rc, errno);
    return NULL;
}

void sigroutine(int dunno) {
    printf("\nI'm doing nothing here : %d\n", dunno);
    return;
}

int main () {
    int                 i;
    int                 rc;
    struct sigaction      actions;
    pthread_t            threads[NUMTHREADS];
    memset(&actions, 0, sizeof(actions));
    sigemptyset(&actions.sa_mask);
    actions.sa_flags = 0;
    actions.sa_handler = sigroutine;

    rc = sigaction(INTERRUPT_SIGNAL,&actions,NULL);
    if(rc){
        printf("sigaction error!\n");
        exit(-1);
    }

    for(i = 0; i < NUMTHREADS; ++i) {
        rc = pthread_create(&threads[i], NULL, threadfunc,(void*)i);
        if(rc){
            printf("pthread_create error!\n");
            exit(-1);
        }
    }
```

```
    sleep(3);
    rc = pthread_kill(threads[0], INTERRUPT_SIGNAL);
    if(rc){
        printf("pthread_kill error!\n");
        exit(-1);
    }
    for(;;);
    return 1;
}
```

编译命令：gcc <program_name> -lpthread -o signal_test.out

输出样本：

```
Thread 0xb77bcb70 00016088 entered
Thread 0xb6fbbb70 00000000 entered
Thread 0xb67bab70 00000000 entered

I'm doing nothing here : 62
Thread 0xb77bcb70 00016088 got a signal delivered to it
Thread 0xb6fbbb70 00000000 did not get expected results! rc=0, errno=0
Thread 0xb67bab70 00000000 did not get expected results! rc=0, errno=0
```

其实该小程序的意图很简单：创建了 3 个线程，每个线程内部会 sleep 30 秒，安装了一个信号 INTERRUPT_SIGNAL，然后往第一个线程发送 INTERRUPT_SIGNAL 信号；可想而知，第一个线程会因为收到信号而终止 sleep，后面两个线程就只能等 30 秒了。

现在理解了 NativeThread 了，我们再看 NativeDispatcher

首先我们得知道在 ServerSocketChannelImpl 中，nd 被初始化为 SocketDispatcher，见：

```
static {
    Util.load();
    initIDs();
    nd = new SocketDispatcher();
}
```

又因为 linux 下一切皆文件的思想(现实虽然不绝对)，SocketDispatcher 其实就是用 FileDispatcher 实现的，最终 FileDispatcher 也只是封装了一大堆 native 方法，一波三折，

关于 FileDispatcher，这里先不详细讲解了，先针对 nd.preClose(fd)和 kill 将 implCloseSelectableChannel 的过程说明白吧：

首先，我们要明白这样一个道理：在多线程环境下，总是很难知道什么时候可安全的关闭或释放资源(如 fd)，当一个线程 A 使用 fd 来读写，而另一个线程 B 关闭或释放了 fd，则 A 线程就会读写一个错误的文件或 socket；为了防止这种情况出现，于是 NIO 就采用了经典的 two-step 处理方案：

第一步：创建一个 socket pair，假设 FDs 为 sp[2]，先 close 掉 sp[1]，这样，该 socket pair 就成为了一个半关闭的链接；复制(dup2)sp[0]到 fd(即为我们想关闭或释放的 fd)，这个时候，其他线程如果正在读写立即会获得 EOF 或者 Pipe Error，read 或 write 方法里会检测这些状态做相应处理；

第二步：最后一个会使用到 fd 的线程负责释放

nd.preClose(fd)即为两步曲中的第一步，我们先来看其实现，最终定位到 FileDispatcher.c，相关代码如下：

```
static int preCloseFD = -1;

JNIEXPORT void JNICALL
Java_sun_nio_ch_FileDispatcher_init(JNIEnv *env, jclass cl)
```

```
{
    int sp[2];
    if (socketpair(PF_UNIX, SOCK_STREAM, 0, sp) < 0) {
    JNU_ThrowIOExceptionWithLastError(env, "socketpair failed");
        return;
    }
    preCloseFD = sp[0];
    close(sp[1]);
}
JNIEXPORT void JNICALL
Java_sun_nio_ch_FileDispatcher_preClose0(JNIEnv *env, jclass clazz, jobject fdo)
{
    jint fd = fdval(env, fdo);
    if (preCloseFD >= 0) {
    if (dup2(preCloseFD, fd) < 0)
        JNU_ThrowIOExceptionWithLastError(env, "dup2 failed");
    }
}
```

从上面两个函数实现，我们可以看到，在 `init` 函数中，创建了一个半关闭的 `socket pair`，`preCloseFD` 即为未关闭的一端，`init` 在静态初始化时就会被执行；再来看关键的 `preClose0`，它的确是采用 `dup2` 来复制 `preCloseFD`，这样一来，`fd` 就被替换成了 `preCloseFD`，这正是 `socket pair` 中未被关闭的一端。

既然 `nd.preClose(fd)` 只是预关闭，则真正执行关闭的逻辑肯定在这个 `kill` 中了，从代码逻辑上还是比较好懂的，`if (!isRegistered())` 即表示该通道没有被注册，表示所有 `Selector` 都没有意愿关心这个通道了，则自然可以放心的关闭 `fd`，通道与 `fd` 的联系请看另一篇。

果断猜测 `kill` 中有 `nd.close(fd)` 这样的代码，不信请看：

```
public void kill() throws IOException {
synchronized (stateLock) {
    if (state == ST_KILLED)
    return;
    if (state == ST_UNINITIALIZED) {
           state = ST_KILLED;
    return;
       }
    assert !isOpen() && !isRegistered();
    nd.close(fd);
    state = ST_KILLED;
}
}
```

果然如此，这样一来，关闭二步曲就能够较安全的释放我们的 `fd` 资源了，至于 `nd.close(fd)` 的本地实现，这里就不讲了，肯定是采用了 `close(fd)` 的系统调用。

总的来说，通道的 `close` 就是为了断开它与内核 `fd` 的那点联系。

3、begin() & end()

`begin()` 和 `end()` 总是配对使用的，`Channel` 和 `Selector` 均有自己的实现，所完成的功能也是有所区别的：

`Selector` 的 `begin()` 和 `end()` 是这样使用的：

```
try {
    begin();
    pollWrapper.poll(timeout);
} finally {
```

```
    end();
}
```

我们先试想这样一个场景，poll 阻塞过程中，Selector 所在线程被中断了，会发生什么事？具体发生什么事这里就不深究了，至少，我们要通知一下辛苦 poll 的内核吧，不管是发信号也好，还是其他方式。

Selector 不是有个天然的 wakeup 吗？似乎还挺优雅，为何不直接使用呢？是的，它们的确使用了，请看 AbstractSelector 中的实现：

```
protected final void begin() {
if (interruptor == null) {
    interruptor = new Interruptible() {
        public void interrupt() {
        AbstractSelector.this.wakeup();
        }};
}
AbstractInterruptibleChannel.blockedOn(interruptor);
if (Thread.currentThread().isInterrupted())
    interruptor.interrupt();
}
protected final void end() {
AbstractInterruptibleChannel.blockedOn(null);
}
```

我们看到，begin 中出现了 wakeup()，不过要理解 begin 和 end，似乎我们先得弄明白 AbstractInterruptibleChannel.blockedOn 究竟在干什么：

AbstractInterruptibleChannel 是这样写的：

```
static void blockedOn(Interruptible intr) {
     sun.misc.SharedSecrets.getJavaLangAccess()
            .blockedOn(Thread.currentThread(),intr);
}
```

其中 JavaLangAccess 接口在 java.lang.System 中被实例化，它是这样写的：

```
private static void setJavaLangAccess() {
    // Allow privileged classes outside of java.lang
    sun.misc.SharedSecrets.setJavaLangAccess(new sun.misc.JavaLangAccess(){
        public sun.reflect.ConstantPool getConstantPool(Class klass) {
            return klass.getConstantPool();
        }
        public void setAnnotationType(Class klass, AnnotationType type) {
            klass.setAnnotationType(type);
        }
        public AnnotationType getAnnotationType(Class klass) {
            return klass.getAnnotationType();
        }
        public <E extends Enum<E>>
        E[] getEnumConstantsShared(Class<E> klass) {
            return klass.getEnumConstantsShared();
        }
        public void blockedOn(Thread t, Interruptible b) {
            t.blockedOn(b);
        }
        public void registerShutdownHook(int slot, Runnable r) {
            Shutdown.add(slot, r);
```

```
        }
    });
}
```

现在我们发现，JavaLangAccess 的 blockedOn 实现，居然只有这么一句 t.blockedOn(b)，那么 AbstractInterruptibleChannel.blockedOn 实现就可以翻译成这样：

Thread.currentThread().blockedOn(intr) ,只因为该方法是包级私有的 ,并且 Interruptible 也是对我们不可见的，我们无法直接调用。

最后只用看 java.lang.Thread 中 blockedOn 的实现了：

```
private volatile Interruptible blocker;
void blockedOn(Interruptible b) {
    synchronized (blockerLock) {
        blocker = b;
    }
}
```

原来 Thread 类中包含 Interruptible 的私有成员 blocker，blockedOn 只是给它赋值而已。

到这里 ,要理解 blockedOn 究竟要做什么 ,就剩下理解这个 blocker 究竟有什么用 ,其实找到我们最常用的方法 interrupt()：

```
public void interrupt() {
if (this != Thread.currentThread())
    checkAccess();
    synchronized (blockerLock) {
        Interruptible b = blocker;
        if (b != null) {
            interrupt0();       // Just to set the interrupt flag
            b.interrupt();
            return;
        }
   }
   interrupt0();
}
```

看到了吧，b.interrupt()，java 线程执行 interrupt 时，如果 blocker 有被赋值，则会执行它的 interrupt。

最终回归到 begin()和 end()，豁然开朗：

begin()中的 Interruptible 实现的 interrupt 中就调用了 wakeup()，这样一来，当内核 poll 阻塞中，java 线程执行 interrupt()，就会触发 wakeup()，从而使得内核优雅的终止阻塞；

至于 end()，就更好理解了，poll()结束后，就没有必要再 wakeup 了，所以就 blockOn(null)了。

blockOn 我们可以理解为，如果线程被中断，就附带执行我的这个 interrupt 方法吧。

以上讲解了 Selector 对 begin()、end()的运用，下面就来看 Channel 是如何运用它们，实现在 AbstractInterruptibleChannel(blockOn 的提供者)中：

```
protected final void begin() {
    if (interruptor == null) {
        interruptor = new Interruptible() {
            public void interrupt() {
              synchronized (closeLock) {
                if (!open)
                return;
                interrupted = true;
                open = false;
                try {
                   AbstractInterruptibleChannel.this.implCloseChannel();
```

```
                } catch (IOException x) { }
            }
        }};
    }
    blockedOn(interruptor);
    if (Thread.currentThread().isInterrupted())
        interruptor.interrupt();
}

protected final void end(boolean completed)
throws AsynchronousCloseException
{
    blockedOn(null);
    if (completed) {
        interrupted = false;
        return;
    }
    if (interrupted) throw new ClosedByInterruptException();
    if (!open) throw new AsynchronousCloseException();
}
```

理解了 Selector 的 begin()、end()实现，再来看这个，基本没什么难度，其实也可以猜想到，Selector 既然在 begin()和 end()作用域内挂上 wakeup，则 Channel 肯定会在 begin()和 end()作用域内挂上 close 之类的。

的确如此，它在 begin()中挂上的是 implCloseChannel()来实现关闭 Channel。

Channel 的使用地方非常多，在涉及到与内核交互(体现在那些 native 方法上)时，都会在头尾加上这个 begin()、end()。

另外，似乎 Channel 的 end 有所不同，它还包含一个参数 completed，用于表示 begin()和 end()之间的操作是否完成，意图也很明显，begin()的 interrupt()中已经设置，如果线程中断时，interrupted 会被更改为 true，这样在 end()被执行的时候，如果未完成，则会跑出 ClosedByInterruptException 异常，当然，如果操作确实没有被打断，则会将其平反。

见 ServerSocketChannel#accept 实现的代码端：

```
    try {
    begin();
    if (!isOpen())
        return null;
    thread = NativeThread.current();
    for (;;) {
        n = accept0(this.fd, newfd, isaa);
        if ((n == IOStatus.INTERRUPTED) && isOpen())
        continue;
        break;
    }
    } finally {
    thread = 0;
    end(n > 0);
    assert IOStatus.check(n);
    }
```

n 为 accept0 native 方法的返回值，当且仅当 n>0 时属于正常返回，所以才有了这个 end(n > 0)，从上述代码我们可以看到，当前这个 begin()和 end()就是防止在 accept0 时被中断所做的措施。