# 异步，同步，阻塞，非阻塞

同步和异步关注的是消息通信机制，所谓同步就是调用者进行调用后，在没有得到结果之前，该调用一直不会返回，但是一旦调用返回，就得到了返回值，同步就是指调用者主动等待调用结果；而异步则相反，执行调用之后直接返回，所以可能没有返回值，等到有返回值时，由被调用者通过状态，通知来通知调用者．异步就是指被调用者来通知调用者调用结果就绪．所以，二者在消息通信机制上有所不同，一个是调用者检查调用结果是否就绪，一个是被调用者通知调用者结果就绪

阻塞和非阻塞关注的是程序在等待调用结果(消息，返回值)时的状态．阻塞调用是指在调用结果返回之前，当前线程会被挂起，调用线程只有在得到结果之后才会继续执行．非阻塞调用是指在不能立刻得到结构之前，调用线程不会被挂起，还是可以执行其他事情．

两组概念相互组合就有四种情况，分别是同步阻塞，同步非阻塞，异步阻塞，异步非阻塞．我们来举个例子来分别类比上诉四种情况．

比如你要从网上下载一个1G的文件，按下下载按钮之后，如果你一直在电脑旁边，等待下载结束，这种情况就是同步阻塞；如果你不需要一直呆在电脑旁边，你可以去看一会书，但是你还是隔一段时间来查看一下下载进度，这种情况就是同步非阻塞；如果你一直在电脑旁边，但是下载器在下载结束之后会响起音乐来提醒你，这就是异步阻塞；但是如果你不呆在电脑旁边，去看书，下载器下载结束后响起音乐来提醒你，那么这种情况就是异步非阻塞．

# Unix的I/O类型

知道上述两组概念之后，我们来看一下Unix下可用的5种I/O模型：

阻塞I/O(bloking IO)

非阻塞I/O(nonblocking IO)

多路复用I/O(IO multiplexing)

信号驱动I/O(signal driven IO)

异步I/O(asynchronous IO)

前４种都是同步，只有最后一种是异步I/O.需要注意的是Java NIO依赖于Unix系统的多路复用I/O,对于I/O操作来说，它是同步I/O，但是对于编程模型来说，它是异步网络调用.下面我们就以系统read的调用来介绍不同的I/O类型．

 当一个read发生时，它会经历两个阶段:

1 等待数据准备

2 将数据从内核内存空间拷贝到进程内存空间中

不同的I/O类型，在这两个阶段中有不同的行为．但是由于这块内容比较多，而且多为表述性的知识，所以这里我们只给出几张图片来解释，感觉兴趣的同学可以去具体了解一下。

阻塞I/O

阻塞I/O

非阻塞I/O

非阻塞I/O

多路复用I/O

多路复用I/O

信号驱动

信号驱动

异步I/O

异步I/O

# Java NIO的底层实现

 我们都知道Netty通过JNI的方式提供了Native Socket Transport，为什么Netty要提供自己的Native版本的NIO呢？明明Java NIO底层也是基于epoll调用(最新的版本)的．这里，我们先不明说，大家想一想可能的情况．下列的源码都来自于OpenJDK-8u40-b25版本．

## open方法

 如果我们顺着Selector.open()方法一个类一个类的找下去，很容易就发现Selector的初始化是由DefaultSelectorProvider根据不同操作系统平台生成的不同的SelectorProvider，对于Linux系统，它会生成EPollSelectorProvider实例，而这个实例会生成EPollSelectorImpl作为最终的Selector实现．

class EPollSelectorImpl extends SelectorImpl

{

.....

// The poll object

EPollArrayWrapper pollWrapper;

.....

EPollSelectorImpl(SelectorProvider sp) throws IOException {

.....

pollWrapper = new EPollArrayWrapper();

pollWrapper.initInterrupt(fd0, fd1);

.....

}

.....

}

 EpollArrayWapper将Linux的epoll相关系统调用封装成了native方法供EpollSelectorImpl使用．

private native int epollCreate();

private native void epollCtl(int epfd, int opcode, int fd, int events);

private native int epollWait(long pollAddress, int numfds, long timeout,

int epfd) throws IOException;

 上述三个native方法就对应Linux下epoll相关的三个系统调用

//创建一个epoll句柄，size是这个监听的数目的最大值．

int epoll\_create(int size);

//事件注册函数，告诉内核epoll监听什么类型的事件，参数是感兴趣的事件类型，回调和监听的fd

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

//等待事件的产生，类似于select调用，events参数用来从内核得到事件的集合

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

 所以，我们会发现在EpollArrayWapper的构造函数中调用了epollCreate方法，创建了一个epoll的句柄．这样，Selector对象就算创造完毕了．

## register方法

 与open类似，ServerSocketChannel的register函数底层是调用了SelectorImpl类的register方法，这个SelectorImpl就是EPollSelectorImpl的父类．

protected final SelectionKey register(AbstractSelectableChannel ch,

int ops,

Object attachment)

{

if (!(ch instanceof SelChImpl))

throw new IllegalSelectorException();

//生成SelectorKey来存储到hashmap中，一共之后获取

SelectionKeyImpl k = new SelectionKeyImpl((SelChImpl)ch, this);

//attach用户想要存储的对象

k.attach(attachment);

//调用子类的implRegister方法

synchronized (publicKeys) {

implRegister(k);

}

//设置关注的option

k.interestOps(ops);

return k;

}

 EpollSelectorImpl的相应的方法实现如下，它调用了EPollArrayWrapper的add方法，记录下Channel所对应的fd值,然后将ski添加到keys变量中．在EPollArrayWrapper中有一个byte数组eventLow记录所有的channel的fd值.

protected void implRegister(SelectionKeyImpl ski) {

if (closed)

throw new ClosedSelectorException();

SelChImpl ch = ski.channel;

//获取Channel所对应的fd,因为在linux下socket会被当作一个文件，也会有fd

int fd = Integer.valueOf(ch.getFDVal());

fdToKey.put(fd, ski);

//调用pollWrapper的add方法,将channel的fd添加到监控列表中

pollWrapper.add(fd);

//保存到HashSet中，keys是SelectorImpl的成员变量

keys.add(ski);

}

 我们会发现,调用register方法并没有涉及到EpollArrayWrapper中的native方法epollCtl的调用,这是因为他们将这个方法的调用推迟到Select方法中去了.

## Select方法

 和register方法类似,SelectorImpl中的select方法最终调用了其子类EpollSelectorImpl的doSelect方法

protected int doSelect(long timeout) throws IOException {

.....

try {

....

//调用了poll方法,底层调用了native的epollCtl和epollWait方法

pollWrapper.poll(timeout);

} finally {

....

}

....

//更新selectedKeys,为之后的selectedKeys函数做准备

int numKeysUpdated = updateSelectedKeys();

....

return numKeysUpdated;

}

 由上述的代码，可以看到，EPollSelectorImpl先调用EPollArrayWapper的poll方法,然后在更新SelectedKeys．其中poll方法会先调用epollCtl来注册先前在register方法中保存的Channel的fd和感兴趣的事件类型，然后epollWait方法等待感兴趣事件的生成,导致线程阻塞.

int poll(long timeout) throws IOException {

updateRegistrations(); ////先调用epollCtl,更新关注的事件类型

////导致阻塞，等待事件产生

updated = epollWait(pollArrayAddress, NUM\_EPOLLEVENTS, timeout, epfd);

.....

return updated;

}

 等待关注的事件产生之后(或在等待时间超过预先设置的最大时间),epollWait函数就会返回.select函数从阻塞状态恢复.

## selectedKeys方法

 我们先来看SelectorImpl中的selectedKeys方法.

//是通过Util.ungrowableSet生成的,不能添加,只能减少

private Set<SelectionKey> publicSelectedKeys;

public Set<SelectionKey> selectedKeys() {

....

return publicSelectedKeys;

}

 很奇怪啊,怎麽直接就返回publicSelectedKeys了,难道在select函数的执行过程中有修改过这个变量吗?

 publicSelectedKeys这个对象其实是selectedKeys变量的一份副本,你可以在SelectorImpl的构造函数中找到它们俩的关系,我们再回头看一下select中updateSelectedKeys方法.

private int updateSelectedKeys() {

//更新了的keys的个数,或在说是产生的事件的个数

int entries = pollWrapper.updated;

int numKeysUpdated = 0;

for (int i=0; i<entries; i++) {

//对应的channel的fd

int nextFD = pollWrapper.getDescriptor(i);

//通过fd找到对应的SelectionKey

SelectionKeyImpl ski = fdToKey.get(Integer.valueOf(nextFD));

if (ski != null) {

int rOps = pollWrapper.getEventOps(i);

//更新selectedKey变量,并通知响应的channel来做响应的处理

if (selectedKeys.contains(ski)) {

if (ski.channel.translateAndSetReadyOps(rOps, ski)) {

numKeysUpdated++;

}

} else {

ski.channel.translateAndSetReadyOps(rOps, ski);

if ((ski.nioReadyOps() & ski.nioInterestOps()) != 0) {

selectedKeys.add(ski);

numKeysUpdated++;

}

}

}

}

return numKeysUpdated;

}

# 后记

 看到这里,详细大家都已经了解到了NIO的底层实现了吧.这里我想在说两个问题.

 一是为什么Netty自己又从新实现了一边native相关的NIO底层方法? 听听Netty的创始人是怎麽说的吧(链接在末尾)。因为Java的版本使用的epoll的level-triggered模式，而Netty则希望使用edge-triggered模式，而且Java版本没有将epoll的部分配置项暴露出来，比如说TCP\_CORK和SO\_REUSEPORT。

 二是看这么多源码,花费这么多时间有什么作用呢?我感觉如果从非功利的角度来看,那么就是纯粹的希望了解的更多,有时候看完源码或在理解了底层原理之后,都会用一种恍然大悟的感觉,比如说AQS的原理.如果从目的性的角度来看,那么就是你知道底层原理之后,你的把握性就更强了,如果出了问题,你可以更快的找出来,并且解决.除此之外,你还可以按照具体的现实情况,以源码为模板在自己造轮子,实现一个更加符合你当前需求的版本.