<https://blog.csdn.net/nieyanshun_me/article/details/52397153>

**Nio与Epoll**

一直对nio和epoll没有系统的认识，最近看了下openjdk，简单的做个记录。

* **Linux2.6之后支持epoll**
* **windows支持select而不支持epoll**
* **不同系统下nio的实现是不一样的，包括Sunos linux 和windows**
* **select的复杂度为O(N)**
* **select有最大fd限制，默认为1024**
* **修改sys/select.h可以改变select的fd数量限制**
* **epoll的事件模型，无fd数量限制，复杂度O(1),不需要遍历fd**

个人对于Nio不算太熟，所以用参考《netty权威指南》，写了一个TimeServer，从这个代码入手分析nio的实现原理。

public class NioTimeServer {

public static void main(String[] args) {

int port = 8080;

MultiplexerTimeServer timeServer = new MultiplexerTimeServer(port);

new Thread(timeServer).start();

}

static final class MultiplexerTimeServer implements Runnable {

private Selector selector;

private ServerSocketChannel servChannel;

private volatile boolean stop;

public MultiplexerTimeServer(int port) {

try {

selector = Selector.open();

servChannel = ServerSocketChannel.open();

servChannel.configureBlocking(false);

servChannel.socket().bind(new InetSocketAddress(port), 1024);

servChannel.register(selector, SelectionKey.OP\_ACCEPT);

} catch (IOException e) {

e.printStackTrace();

System.exit(1);

}

}

public void stop() {

this.stop = true;

}

@Override

public void run() {

while (!stop) {

try {

selector.select(1000);

Set<SelectionKey> selectedKeys = selector.selectedKeys();

Iterator<SelectionKey> it = selectedKeys.iterator();

SelectionKey key = null;

while (it.hasNext()) {

key = it.next();

it.remove();

try {

handleInput(key);

} catch (Exception e) {

if (key != null) {

key.cancel();

if (key.channel() != null)

key.channel().close();

}

}

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

private void handleInput(SelectionKey key) throws IOException {

if (key.isValid()) {

if (key.isAcceptable()) {

ServerSocketChannel ssc = (ServerSocketChannel) key.channel();

SocketChannel sc = ssc.accept();

sc.configureBlocking(false);

sc.register(selector, SelectionKey.OP\_READ);

}

if (key.isReadable()) {

SocketChannel sc = (SocketChannel) key.channel();

ByteBuffer readBuf = ByteBuffer.allocate(1024);

int readBytes = sc.read(readBuf);

if (readBytes > 0) {

readBuf.flip();

byte[] bytes = new byte[readBuf.remaining()];

readBuf.get(bytes);

String body = new String(bytes, "UTF-8");

System.out.println("The time server receive order :" + body);

String currentTime = "QUERY TIME ORDER".equalsIgnoreCase(body)

? new Date(System.currentTimeMillis()).toString() : "BAD ORDER";

doWrite(sc, currentTime);

} else if (readBytes < 0) {

key.cancel();

sc.close();

}

}

}

}

/\*\*

\* @param sc

\* @param currentTime

\* @throws IOException

\*/

private void doWrite(SocketChannel sc, String response) throws IOException {

if (response != null && response.trim().length() > 0) {

byte[] bytes = response.getBytes();

ByteBuffer writeBuf = ByteBuffer.allocate(bytes.length);

writeBuf.put(bytes);

writeBuf.flip();

sc.write(writeBuf);

}

}

}

1.创建一个ServerSocketChannel，设置为非阻塞模式，同时绑定监听端口，并注册channel到选择器上（注册感兴趣的key），   
2.用一个线程去轮询选择器，调用选择器的select方法，获取所有就绪的key，key和channel是相关的，通过key的状态来决定进一步的处理。

**我们重点看的只有一个地方，那就是selector.select(1000);先看如何获取selector：**

public static Selector open() throws IOException {

return SelectorProvider.provider().openSelector();

}

这是使用了SelectorProvider去创建一个Selector，看下SelectorProvider的默认实例：

public static SelectorProvider provider() {

synchronized (lock) {

if (provider != null)

return provider;

return AccessController.doPrivileged(

new PrivilegedAction<SelectorProvider>() {

public SelectorProvider run() {

if (loadProviderFromProperty())

return provider;

if (loadProviderAsService())

return provider;

provider = sun.nio.ch.DefaultSelectorProvider.create();

return provider;

}

});

}

}

重点只看其中这一行：

provider = sun.nio.ch.DefaultSelectorProvider.create();

这里用到了DefaultSelectorProvider，看下create()方法:

public static SelectorProvider create() {

String osname = AccessController.doPrivileged(

new GetPropertyAction("os.name"));

if ("SunOS".equals(osname)) {

return new sun.nio.ch.DevPollSelectorProvider();

}

// use EPollSelectorProvider for Linux kernels >= 2.6

if ("Linux".equals(osname)) {

String osversion = AccessController.doPrivileged(

new GetPropertyAction("os.version"));

String[] vers = osversion.split("\\.", 0);

if (vers.length >= 2) {

try {

int major = Integer.parseInt(vers[0]);

int minor = Integer.parseInt(vers[1]);

if (major > 2 || (major == 2 && minor >= 6)) {

return new sun.nio.ch.EPollSelectorProvider();

}

} catch (NumberFormatException x) {

// format not recognized

}

}

}

return new sun.nio.ch.PollSelectorProvider();

}

重点到了，我们看到create方法中是通过区分操作系统来返回不同的Provider的。其中SunOs就是Solaris返回的是DevPollSelectorProvider，对于Linux，返回的Provder是EPollSelectorProvider，其余操作系统，返回的是PollSelectorProvider（比如Windows，是不支持epoll的，见注释）   
继续看下EPollSelectorProvider

public class EPollSelectorProvider

extends SelectorProviderImpl

{

public AbstractSelector openSelector() throws IOException {

return new EPollSelectorImpl(this);

}

public Channel inheritedChannel() throws IOException {

return InheritedChannel.getChannel();

}

}

这里用到的是EPollSelectorImpl，由此可知，epoll在nio的实现就在这里了。   
EPollSelectorImpl 中select的实现如下：

protected int doSelect(long timeout)

throws IOException

{

if (closed)

throw new ClosedSelectorException();

processDeregisterQueue();

try {

begin();

pollWrapper.poll(timeout);

} finally {

end();

}

processDeregisterQueue();

int numKeysUpdated = updateSelectedKeys();

if (pollWrapper.interrupted()) {

// Clear the wakeup pipe

pollWrapper.putEventOps(pollWrapper.interruptedIndex(), 0);

synchronized (interruptLock) {

pollWrapper.clearInterrupted();

IOUtil.drain(fd0);

interruptTriggered = false;

}

}

return numKeysUpdated;

}

只看这一句

pollWrapper.poll(timeout);

其中，pollWrapper：

// The poll object

EPollArrayWrapper pollWrapper;

关于EPollArrayWrapper：

/\*\*

\* Manipulates a native array of epoll\_event structs on Linux:

\*

\* typedef union epoll\_data {

\* void \*ptr;

\* int fd;

\* \_\_uint32\_t u32;

\* \_\_uint64\_t u64;

\* } epoll\_data\_t;

\*

\* struct epoll\_event {

\* \_\_uint32\_t events;

\* epoll\_data\_t data;

\* };

\*

\* The system call to wait for I/O events is epoll\_wait(2). It populates an

\* array of epoll\_event structures that are passed to the call. The data

\* member of the epoll\_event structure contains the same data as was set

\* when the file descriptor was registered to epoll via epoll\_ctl(2). In

\* this implementation we set data.fd to be the file descriptor that we

\* register. That way, we have the file descriptor available when we

\* process the events.

\*

\* All file descriptors registered with epoll have the POLLHUP and POLLERR

\* events enabled even when registered with an event set of 0. To ensure

\* that epoll\_wait doesn't poll an idle file descriptor when the underlying

\* connection is closed or reset then its registration is deleted from

\* epoll (it will be re-added again if the event set is changed)

\*/

这是类注释，说明了epoll的数据结构等   
此类是epoll在openjdk中的实现类，肯定有epoll相关的jni：

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;

private static native int sizeofEPollEvent();

private static native int offsetofData();

private static native int fdLimit();

private static native void interrupt(int fd);

private static native void init();

重点在poll方法：

int poll(long timeout) throws IOException {

updateRegistrations();

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

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

if (getDescriptor(i) == incomingInterruptFD) {

interruptedIndex = i;

interrupted = true;

break;

}

}

return updated;

}

首先调用epollCtl系统调用，更新fd到epoll实例，然后调用epollWait系统调用，线程在此处阻塞，超时或有fd就绪时会被唤醒，返回值是一个fd的集合，0表示无就绪时间，-1表示report error and abort，否则遍历并处理fd。   
关于epoll可以参考此文 <http://www.ulduzsoft.com/2014/01/select-poll-epoll-practical-difference-for-system-architects/> 。

**脚注**

The syscall select is available in Windows but select processing is O(n) in the number of file descriptors unlike the modern constant-time multiplexers like epoll which makes select unacceptable for high-concurrency servers. This document will describe how high-concurrency programs are designed in Windows.

Instead of epoll or kqueue, Windows has its own I/O multiplexer called I/O completion ports (IOCPs). IOCPs are the objects used to poll overlapped I/O for completion. IOCP polling is constant time (REF?).

Windows支持select系统调用，（时间复杂度O(N)），但是不支持Epoll，Windows自身的 multiplexer是IOCPs