在这章,我们讨论Java NIO -- 可伸缩的I/O -- 应用于TCP.

# 5.1 TCP的通道

可伸缩的I/O使用我们在4.2.1节所遇到的ServerSocketChannel和SocketChannel类通过TCP被执行.像所有的通道一样,这些通道一开始的时候以阻塞模式被创建.

## 5.1.1 导入语句

下面的Java导入语句假设在这章示例中至始至终都存在.

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.channels.spi.*;
import java.util.*;
```

## 5.1.2 ServerSocketChannel

ServerSocketChannel类为服务器端套接字提供了I/O通道:

```
class ServerSocketChannel {
    static ServerSocketChannel open()  throws IOException;
    ServerSocket socket() throws IOException;
    SocketChannel accept() throws IOException;
}
```

ServerSocketChannel被创建,关联的ServerSocket可以被获取,如下:

```
ServerSocketChannel channel  = ServerSocketChannel.open();
ServerSocket       serverSocket = channel.socket();
```

关联的服务器端套接字创建时是未绑定的,必须使用在3.3.7节描述的ServerSocket.bind方法进行绑定,在用来使用任何的网络操作之前.

accept相当于ServerSocket.accept方法,两个不同:

(a) 它返回一个SocketChannel而不是一个Socket

(b) 它可在非阻塞模式中执行,如果没有传入的连接可用可能返回null.

在下面的情况下,我们必须使用ServerSocketChannel.accept而不是ServerSocket.accept:

(a) 如果我们在非阻塞模式下执行accept操作,或者

(b) 如果我们打算在接收到的套接字上执行非阻塞的通道操作.

## 5.1.3 SocketChannel

类似的,SocketChannel类为socket提供了I/O通道:

```
class SocketChannel {
    static SocketChannel open() throws IOException;
    static SocketChannel open(SocketAddress address) throws IOException;
```

```
    Socket socket() throws IOException;
}
```

SocketChannel被创建,关联的Socket可以获取,如下:

```
SocketChannel channel= SocketChannel.open(…)
Socket        socket = channel.socket();
```

如果使用的是无参数形式的open方法,创建的管理的socket是未连接的,否则,socket被创建并连接到指定的远程SocketAddress.比如:

```
String host = "localhost";
int    port = 7; // echo port
SocketAddress address =    new InetSocketAddress(host, port);
SocketChannelchannel = SocketChannel.open(address);
```

# 5.2 TCP通道操作

如我们在5.1节看到的,SocketChannel以阻塞模式创建.它可以使用以下方法成为阻塞模式:

```
class SocketChannel {
    SelectableChannel configureBlocking(boolean block) throws IOException;
    boolean           isBlocking();
}
```

true为阻塞模式,false为非阻塞模式.

一个未连接的TCP通道可以使用SocketChannel.connect方法连接到一个目标.

```
class SocketChannel {
    boolean  connect(SocketAddress address) throws IOException;
    boolean  isConnected();
    boolean  isConnectionPending();
    boolean  finishConnect() throws IOException;
}
```

isConnected方法判定本地socket是否已经连接到目标上:它不会判定任何关于其它连接的端点,如3.4.10所讨论的.

isConnectionPending和finishConnect方法主要用在阻塞模式中,如5.2.2节所描述的.

## 5.2.1 阻塞的TCP连接

如果通道正在以阻塞模式连接:

(a) connect方法阻塞直到连接完成或者被拒绝,也就是说,它等同于调用有一个未指定的或者无限超时时间的Socket.connect.

(b) finishConnect方法不需要被调用:如果通道连接了,它立即返回或者通道未连接抛出NoConnectionPendingException(书中未写全).

(c) isConnectionPending方法从来不返回true.

### 5.2.2 非阻塞的TCP连接

如果通道正在以非阻塞模式连接：

(a) connect方法立即返回true如果连接立即可以成功,因为有时候发生在本地连接;其它情况下,它返回false,连接必须随后以finishConnect方法完成,同时连接协议持续异步.

(b) finishConnect方法返回true,如果连接已经成功连接;返回false,如果仍然在等待;或者抛出IOException,如果连接可能因为某些原因不能成功：大部分情况,连接被远程目标拒绝,因为在指定端口没有任何的监听,表现为ConnectException.

(c) isConnectionPending方法返回true,如果connect已经被调用但是finishConnect还没有被调用,其它情况下返回false.

将这个与TCP连接协议关联,finishConnect返回false,isConnectionPending返回true直到SYN/ACK分段已经从服务器接收到,如图5.1所展示的序列图.

一个简单的非阻塞TCP客户端连接操作在示例5.1中说明.

```
SocketAddress address = new InetSocketAddress("localhost",7);
SocketChannel channel = SocketChannel.open();
channel.configureBlocking(false);
if (!channel.connect(address)) {
    // connection was not instantaneous, still pending -- 连接不是及时的,仍然在等待
    while (!channel.finishConnect()) {
    // connection still incomplete: do something useful … 连接仍然未完成:做一些有用的事情
    }
}
```

Example 5.1 Simple non-blocking TCP client connection

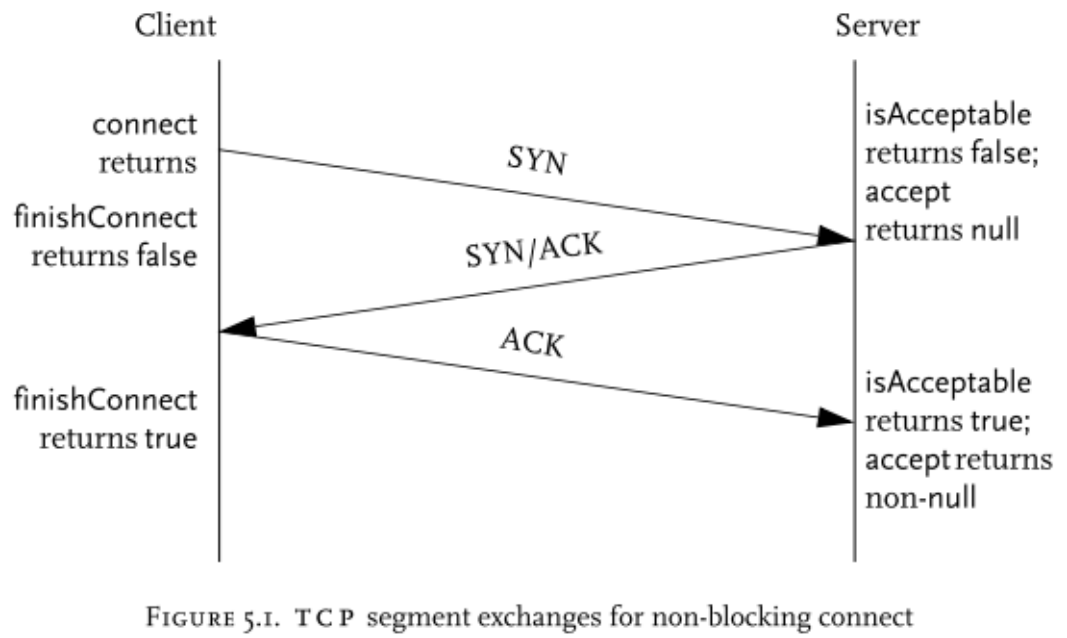


FIGURE 5.1. TCP segment exchanges for non-blocking connect

### 5.2.3 读操作

在阻塞模式中,一个TCP通道读取直到至少有一些数据可用,虽然不必要请求的数据的数量(FIXME: although not necessarily the amount of data requested).

在非阻塞模式中,一个TCP通道读永远不会阻塞.如果连接被另一个端点关闭了或者在这个端点上为读操作关闭了,将抛出异常.如果数据存在于套接字的接收缓冲区中,数据将返回,达到请求的数量或者数据自身的大小,无论哪一个更小.如果数据不存在,操作不做什么然后返回0.

如果没有异常抛出,返回值表明了多少数据被读取,可能为0.

### 5.2.4 写操作

在阻塞模式中,一个TCP通道写阻塞直到至少有一些数据在套接字发送缓冲区中可用,虽然不必要请求的空间量写入的数据(FIXME: although not necessarily the amount of space required by the data written).

---

多少空间?大部分TCP实现支持"low-water mark",默认是2KB:写只是传输数据如果至少有这样的空间在发送缓冲区可用. low-water mark,虽然被TCP实现,但是不能被Java如JDK 1.5控制.

---

在非阻塞模式中,一个TCP通道写永远不会阻塞.如果连接已经被另一端关闭或者在这个端点已经为写关闭,将抛出异常.如果在套接字的发送缓冲区空间是可用的,数据将传输达到指定的数量或者可用的空间,以较少者为准.如果空间不可用,操作什么也不做并返回0.

如果没有异常抛出,返回值表明写的多少数据,可能为0.

一个简单的非阻塞TCP I/O 顺序在示例5.2 说明.

```
ByteBuffer buffer = ByteBuffer.allocate(8192);
SocketChannel channel;// initialization as in Example 5.1
channel.configureBlocking(false);
buffer.put(…);
buffer.flip();
while (channel.write(buffer) == 0)
; // do something useful …
buffer.clear();
while (channel.read(buffer) == 0)
; // do something useful …

Example 5.2 Simple non-blocking TCP client I/O
```

如注释所说的,程序应该做一些有用的工作或者至少休眠,而不是盲目的自旋(FIXME: spinning mindlessly)当连接未完成的时候如示例5.1或者I/O传输返回0如示例5.2.

### 5.2.5 关闭操作

一个SocketChannel通过close方法关闭:

```
class SocketChannel {
    void close() throws IOException;
}
```

然而,Selectable,SocketChannel和ServerSocketChannel(和DatagramChannel)的基类的文档,是相当晦涩(obscurely)的:

---

一旦向Selector注册,一个通道保持注册状态直到它被注销.注销涉及释放(deallocating)选择器已分配给该通道的所有资源.

一个通道不能直接注销;相反,必须取消代表注册的key.取消一个key请求通道在selector的下一次selection操作期间注销通道.

这意味着关闭一个当前注册的SelectableChanel在内部实现中在两个阶段：

1. 一个操作去阻止在通道上的进一步操作,在Selectable.close方法中执行,也会取消与通道关联的任意的key,和
2. 一个延迟的操作将真正关闭套接字:这个将发生在Selector.selectXXX方法的下一次执行.

这是按预期的,但是在TCP客户端使用非阻塞的通道I/O会引起问题.通常,一个客户端将退出它的select循环在关闭一个通道之后,这样阶段2永远不会发生.这个行为的特征(symptom)是本地的套接字保持CLOSE-WAIT(*)状态.最简单的解决方案就是在关闭通道后立即调用Selector.selectNow.

```
Selector sel;
SocketChannel sch;
// …
sch.close();
sel.selectNow();
```

(FIXME: Generally, there isn't a great deal of point in using non-blocking I/O in clients: it saves threads, but clients rarely deal with enough different servers (or connections) for it to be worth the trouble.)

* CLOSE-WAIT意味着TCP等待本地端点去关闭套接字在接收到远程关闭之后.见附录：TCP端口状态.

## 5.3 TCP中多路复用

除了OP_READ和OP_WRITE操作,在TCP中多路复用处理准备就绪(readiness)的OP_ACCEPT和OP_CONNECT操作.被TCP支持的可选值的I/O操作展示在表格5.1中.

TABLE 5.1 TCP中的可选择操作

| 操作 | 含义 |
| --- | --- |
| OP_ACCEPT (ServerSocketChannel) | ServerSocketChannel.accept不会返回null：要么一个传入的连接已经存在或者一个异常被挂起 |
| OP_CONNECT (正在连接等待的SocketChannel) | SocketChannnel.finishConnect不会返回false:要么连接已经完成除了finishConnect阶段(FIXME: apart from the step )或者一个异常被挂起,通常为ConnectException. |
| OP_READ (连接的SocketChannel) | read不会返回0:那么数据存在套接字接收缓冲区中,达到了流的末尾或者一个异常被挂起.流的末尾出现在如果远程端关闭了连接或者为停止了输出流,或者本地端关闭了输入流. |
| OP_WRITE (连接的SocketChannel) | write不会能返回0：要么在套接字发送缓冲区中存在空间,或者连接已经关闭或者在远程端关闭了输入流,或者一个异常被挂起. |

a. 除了流的末尾,java.nio.channels.SelectionKey.OP_READ的javadoc到JDK 1.4.2多余地(redundantly)规定了"被远程关闭进一步的阅读"：这个引用Socket.shurdownOutput的远程执行,已经被流的末尾条件覆盖,是远程关闭和本地关闭为输入流的省略情况.见3.7.1和3.7.2节.
b.

实际上,在底层的本地select()API中,没有这四种而只有两种事件:
(a) 'readable'事件:标志着在接收缓冲区中存储数据和一个可用的传入的连接到达accept方法(FIXME: availability of an incoming connection to the method).换句话说,在幕后,OP_ACCEPT和OP_READ是同一个事件.
(b) 'writeable'事件,标志着在发送缓冲区中存在空间和一个客户端套接字连接的完成.换句话说,在幕后,OP_CONNECT和OP_WRITE也是同一个事件,更多的,由于两者简单的意味着在发送缓冲区中空间是可用的-一个首先会出现的必要就是当连接完成的时候(FIXME: a condition which of course first occurs when the connection is complete).(这里翻译的有点晕+_+)

在Java中OP_ACCEPT和OP_READ等同(identity)是没有问题的,OP_ACCEPT只在ServerSocket中有效,OP_READ是无效的.
然而,OP_CONNECT和OP_WRITE等同会引起问题.OP_CONNECT和OP_WRITE客户端套接字都是有效的,这样在这些事件之间存在着模棱两可:确实JDK 1.4的某些实现会胡作非为如果你尝试同时使用OP_CONNECT和OP_WRITE.查看Bug 485073,496079I,4919127C的示例.因此你必须如下处理:
(a) 只在未连接的套接字上使用OP_CONNECT:OP_CONNECT必须尽快从interestOps中移除当它为一个通道准备好了.
(b) 只在连接的套接字上使用OP_WRITE.

这种混乱本来是可以避免的,如果Sun没有试图区分事件,在其规范是不清楚的.
而且,你应该只使用OP_WRITE当你有一些东西要去写入,(FIXME: you've already failed to write it completely (by getting a short or zero-length result from a write method)) .一旦连接完成,OP_WRITE几乎总是准备好的,除了空间在套接字发送缓冲区中不可用的期间.(这个时间可能延长(protracted)), 如果远程端在读操作上慢于本地端的写操作).
它只在选择OP_WRITE的时非常有用,如果你有一些数据准备好发送,只推荐你如果你你只遇到了一个短事件的写(a short write)(也就是说,之前的写操作没有写入需要的全部的数量).
换句话说,你应该假设一个连接的通道准备好写直到你实际发现它不是.无论何时你没有什么要写入或者无论何时一个写操作成功完成,你应该立即停止选择OP_WRITE.

### 5.3.1 示例

一个简单的多路复用的TCp echo 服务器展示在下面的示例中.这个服务器永远不会在I/O中阻塞,只有在Selector.select才会阻塞.这个服务器使用一个关键的附件去维护每一个接收的连接的一个单独的byteBuffer.这是一个一般的附件连接上下文技术的简单例子(FIXME: This is a simple example of a more general attachment technique for connection contexts),将会在第12章展示完全.

```
public class NIOEchoServer implements Runnable {
    public static final int TIMEOUT = 5 * 1000;// 5s
    public static final int BUFFERSIZE = 8192;
    private ServerSocketChannel serverChannel;
    // Constructor
    public NIOEchoServer(int port) throws IOException {
        this.serverChannel = ServerSocketChannel.open();
        serverChannel.configureBlocking(false);
        serverChannel.socket().bind(new InetSocketAddress(port));
    }
    // Runnable.run method
    @Override
```

```
    public void run() {
        try {
            Selector selector = Selector.open();
            serverChannel.register(selector, serverChannel.validOps());
            // loop while there are any registered channels
            while (selector.keys().size() > 0) {
                int keyCount = selector.select(TIMEOUT);
                Iterator selectedKeysIterator = selector.selectedKeys().iterator();
                // loop over selected keys
                while (selectedKeysIterator.hasNext()) {
                    SelectionKey key = (SelectionKey) selectedKeysIterator.next();
                    // Remove from selected set and test validity
                    selectedKeysIterator.remove();
                    if (!key.isValid())
                        continue;
                    // dispatch:
                    if (key.isAcceptable())
                        handleAcceptable(key);
                    if (key.isReadable())
                        handleReadable(key);
                    if (key.isWritable())
                        handleWritable(key);
                } // end iteration
            } // end while selector.keys().size() > 0
        } catch (IOException e) { /* … */
        }
    } // end run()

    // handle acceptable key
    void handleAcceptable(SelectionKey key) {
        try {
            ServerSocketChannel srvCh = (ServerSocketChannel) key.channel();
            SocketChannel channel = srvCh.accept();
            channel.configureBlocking(false);
            // allocate a buffer to the conversation
            ByteBuffer buffer = ByteBuffer.allocateDirect(BUFFERSIZE);
            // register the accepted channel for read,
            // with the buffer as the attachment
            channel.register(key.selector(), SelectionKey.OP_READ, buffer);
        } catch (IOException e) {
            /* … */
        }
    } // end handleAcceptable()

    // handle readable key
    void handleReadable(SelectionKey key) {
        try {
```

```
            SocketChannel channel = (SocketChannel) key.channel();
            ByteBuffer buffer = (ByteBuffer) key.attachment();
            int count = channel.read(buffer);
            // Echo input to output, assuming writability
            // (see Table 5.1)
            handleWritable(key);
            if (count < 0) {
                // EOF - flush remaining output and close.
                while (buffer.position() > 0) {
                    buffer.flip();
                    channel.write(buffer);
                    buffer.compact();
                }
                key.cancel();
                channel.close();
            }
        } catch (IOException e) {
            /* … */
        }
    } // end handleReadable()

    // handle writable key
    void handleWritable(SelectionKey key) {
        try {
            SocketChannel channel = (SocketChannel) key.channel();
            ByteBuffer buffer = (ByteBuffer) key.attachment();
            buffer.flip();
            int count = channel.write(buffer);
            buffer.compact();
            // Register or deregister for OP_WRITE depending on
            // success of write operation (see Table 5.1).
            int ops = key.interestOps();
            if (buffer.hasRemaining())
                ops |= SelectionKey.OP_WRITE;
            else
                ops &= ~SelectionKey.OP_WRITE;
            key.interestOps(ops);
        } catch (IOException e) {
            /* … */
        }
    } // end handleWritable()
}// end of NIOEchoServer
```