这章讨论了JDK 1.4 UDP数据报可伸缩I/O的使用.

10.1 UDP的通道

在UDP之上的可伸缩I/O使用我们在之前4.2.1节所遇到的DatagramChannel类执行.

10.1.1 导入语句

下面的Java导入语句假设在这章的示例中至始至终存在.

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

10.1.2 创建一个DatagramChannel

DatagramChannel类有和我们之前已经遇到的SocketChannel一样的打开和关闭方法:

```
class DatagramChannel {
    static DatagramChannel open() throws IOException
    boolean isOpen();
    void close() throws IOException;
}
```

打开一个DatagramChannel返回一个阻塞模式的通道并可准备来使用.它的套接字可以绑定如果需要的话如9.3.5节所讨论的.通道必须被关闭当它结束的时候.警告(Caveat): 见5.2.5节的关于关闭注册的通道的说明,同样适用于DatagramChannel.

10.1.3 连接操作

DatagramChannel可以被直接连接和断开,和通过它的DatagramSocket一样:

```
class DatagramChannel {
    DatagramChannel connect(SocketAddress target) throws IOException;
    DatagramChannel disconnect() throws IOException;
    boolean isConnected();
}
```

这些方法操作等同于DatagramSocket一致的方法.不像SocketChannel的连接方法,connect方法不是一个UDP套接字连接操作的非阻塞版本:DatagramChannel.connect和DatagramSocket.connect语义上是相同的.

isConnected方法判断本地套接字是否已经连接上.

如我们在9.3.7看到的,连接一个数据报套接字仅仅是一个本地的操作而没有设计网络,也就是说,没有什么去阻塞.

10.2 DatagramChannel I/O

DatagramChannel有我们已经在4.2.2节看到的读和写操作,(FIXME: as required by the

interfaces which it implements):

```
class DatagramChannel {
    int read(ByteBuffer) throws IOException;
    int read(ByteBuffer[] buffers) throws IOException;
    int read(ByteBuffer[] buffers, int offset, int length) throws IOException;
    int write(ByteBuffer) throws IOException;
    int write(ByteBuffer[] buffers)throws IOException;
    int write(ByteBuffer[] buffers, int offset, int length) throws IOException;
}
```

现在,这个API完全不同于(FIXME: the DatagramPacket-orirnted API exported by the DatagramSocket class)在9.4.1节和9.4.2节描述的.特别是,没有地方去指定一个输出套接字的目的地址,或者指定一个传入的数据报的接收的源地址.因为这个原因,read和write方法在DatagramChannel受限于一个重要的语义约束:如果关联的套接字连接了如9.3.7节定义的,它们才能被使用.在这种情况下,一个接收或者发送的数据包的源或者目标地址只能是连接的套接字,所以上面的读/写API是适合的.

为了处理未连接的数据报套接字,DatagramChannel提供了两个新的方法:

```
class DatagramChannel {
    SocketAddress receive(ByteBuffer buffer) throws IOException;
    int send(ByteBuffer buffer, SocketAddress target) throws IOException;
}
```

这些分别对应于DatagramSocket.receive和DatagramSocket.send方法.DatagramChannel.send的target参数是传输的远程目标.DatagramChannel.receive的返回值为SocketAddress,如我们在2.2节看到的,代表了传输的远程源.

10.2.1 阻塞的UDP I/O

如我们在10.1节看到的,一个DatagramChannel以阻塞模式创建.在这种模式下:

(a) 一个读操作阻塞直到一个传入的数据报已经接收到套接字接收缓冲区,如果没有任何东西存在.

(b) 一个写操作阻塞直到空间可用使在套接字发送缓冲区中的输出数据报排队,也就是说,直到有足够的数据报预先传输到网络上排队:这将延迟,如果发生的话,它通常非常短暂.由于数据报以最大的速录传输.

10.2.2 非阻塞的UDP I/O

一个DatagramChannel可被放置于非阻塞模式:

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

在非阻塞模式中,write和send方法可以返回0,表明套接字发送缓冲区已经满了,没有传输可以被执行;类似的,read和receive方法可以分别返回0或者null,表明没有数据报可用.

一个简单的非阻塞的UDP I/O顺序再示例10.1中说明.

```
ByteBuffer buffer = ByteBuffer.allocate(8192);
DatagramChannel channel= DatagramChannel.open();
channel.configureBlocking(false);
SocketAddress address
```

```
= new InetSocketAddress("localhost", 7);
buffer.put(…);
buffer.flip();
while (channel.send(buffer, address) == 0)
; // do something useful …
buffer.clear();
while ((address = channel.receive(buffer)) == null)
; // do something useful …
```

Example 10.1 简单的非阻塞的UDP 客户端I/O

如注释所说,程序应该做一些有用的工作或者休眠而不是无知的自旋(FIXME: spinning mindlessly)当I/O传输返回0的时候.

10.3 多路复用

10.3.1 UDP中的可伸缩I/O操作

UDP中的可伸缩I/O操作都适用于DatagramChannel对象.它们的'准备'状态的含义在表10.1展示.

表10.1 UDP中的可伸缩I/O操作

操作	含义
OP_READ	数据存在于套接字接收缓冲区中或者异常挂起.
OP_WRITE	在套接字发送缓冲区中存在空间或者异常挂起.在UDP中,OP_WRITE几乎总是准备好除了在套接字发送缓冲区中没有空间可用的期间.最后只注册OP_WRITE一旦这个缓冲区满的条件被探测到,也就是说,当一个通道写返回少于请求的写的长度,注销OP_WRITE一旦它被清理了,也就是说,一个通道写完全成功了.

10.3.2 多路复用I/O

一个DatagramChannel可以使用一个选择器注册,调用Selector.select方法去等待通道的可读或者可写,如10.3.1节讨论的:

```
DatagramChannelchannel = DatagramChannel.open();
// bind to port 1100
channel.bind(new InetSocketAddress(1100));
Selector selector = Selector.open();
// register for OP_READ
channel.register(selector, SelectionKey.OP_READ);
// Select
selector.select();
```

10.3.3 示例

一个简单的多路复用UDP echo服务器在示例10.2中展示(书中写的是List<Datagram>,正确的应该是List<DatagramPacket>).

```
public class NIOUDPEchoServer implements Runnable {
    static final int TIMEOUT = 5000;// 5s
    private ByteBuffer buffer = ByteBuffer.allocate(8192);
    private DatagramChannel channel;
    private List<DatagramPacket> outputQueue = new LinkedList<DatagramPacket>();
```

```
    // Create new NIOUDPEchoServer
    public NIOUDPEchoServer(int port) throws IOException {
        this.channel = DatagramChannel.open();
        channel.socket().bind(new InetSocketAddress(port));
        channel.configureBlocking(false);
    }
    // Runnable. run method
    @Override
    public void run() {
        try {
            Selector selector = Selector.open();
            channel.register(selector, SelectionKey.OP_READ);
            // loop while there are any registered channels
            while (!selector.keys().isEmpty()) {
                int keysAdded = selector.select(TIMEOUT);
                // Standard post-select processing …
                Set selectedKeys = selector.selectedKeys();
                synchronized (selectedKeys) {
                    Iterator it = selectedKeys.iterator();
                    while (it.hasNext()) {
                        SelectionKey key = (SelectionKey) it.next();
                        it.remove();
                        if (!key.isValid())
                            continue;
                        if (key.isReadable())
                            handleReadable(key);
                        if (key.isWritable())
                            handleWritable(key);
                    } // while
                } // synchronized
            } // while
        } // try
        catch (IOException e) {
            // …
        }
    }
    // handle readable key
    void handleReadable(SelectionKey key) {
        DatagramChannel channel = (DatagramChannel) key.channel();
        try {
            buffer.clear();
            SocketAddress address = channel.receive(buffer);
            if (address == null)
                return; // no data
            buffer.flip();
            channel.send(buffer, address);
            int count = buffer.remaining();
```

```
            if (count > 0) {
                // Write failure: queue the write request
                // as a DatagramPacket, as this nicely holds
                // the data and reply address
                byte[] bytes = new byte[count];
                buffer.get(bytes);
                outputQueue.add(new DatagramPacket(bytes, count, address));
                // Register for OP_WRITE
                key.interestOps(SelectionKey.OP_READ | SelectionKey.OP_WRITE);
            }
        } catch (IOException e) {
            // …
        }
    } // handleReadable()
    // handle writable key
    void handleWritable(SelectionKey key) {
        DatagramChannel channel = (DatagramChannel) key.channel();
        try {
            while (!outputQueue.isEmpty()) {
                DatagramPacket packet = outputQueue.get(0);
                buffer.clear();
                buffer.put(packet.getData());
                buffer.flip();
                channel.send(buffer, packet.getSocketAddress());
                if (buffer.hasRemaining()) // write failed, retry
                    return;
                outputQueue.remove(0);
            }
            // All writes succeeded & queue empty, so
            // deregister for OP_WRITE
            key.interestOps(SelectionKey.OP_READ);
        } catch (IOException e) {
            // …
        }
    } // handleWritable()
} // end of NIOUDPEchoServer
```

这个服务器永远不会在I/O操作中阻塞,只有在Selector.select中.如我们在9.2.1节看到的,它比TCP服务器简单点,不必去管理客户端连接.在多路复用I/O,这个意味着它不必去管理SelectionKey.isAcceptable的情况,或者处理其它的套接字的连接.它需要去管理输出队列包含了数据和目标地址.幸运地是,java.net.DatagramPacket类恰好包含了这些数据项目已经被传递.