**NIO**

This chapter introduces the main facilities of the “New I/O” packages. There are two important parts: the java.nio.channels package, which introduces the Selector and Channel abstractions, and the java.nio package, which introduces the Buffer abstraction.

**Why do we need this?**

Basic Java Sockets work well for small-scale systems. But when it comes to servers that have to deal with many thousands of clients simultaneously, certain issues arise.

If a programmer wants to ensure that certain connections get served before others, or impose a specific order of service, threads may make it harder to do that.

But the most important issue with threads is probably one we haven’t encountered yet. That’s because in our “echo service” examples, each client served is completely independent of all others; clients do not interact with each other or affect the state of the server.

The need to synchronize access to shared state makes it significantly harder to think about both correctness and performance of a threaded server.

Because of these complications, some programmers prefer to stick with a single-threaded approach, in which the server has only one thread, which deals with all clients—not sequentially, but all at once.

That is exactly the point of the Selector and Channel abstractions introduced in NIO. A Channel instance represents a “pollable” I/O target such as a socket (or a file, or a device). Channels can register an instance of class Selector. The select() method of Selector allows you to ask “Among the set of channels, which ones are currently ready to be serviced (i.e., accepted, read, or written)?” There are numerous details to be covered later, but that’s the basic motivation for Selector and Channel, both of which are part of the java.nio.channels package.

The other major feature introduced in NIO is the Buffer class. Just as selectors and channels give greater control and predictability of the overhead involved with handling many clients at once, Buffer enables more efficient, predictable I/O than is possible with the Stream abstraction.

That’s why channels are designed around the use of Buffer instances to pass data around. The Buffer abstraction represents a finite-capacity container for data—essentially, an array with associated pointers indicating where to put data in, and where to read data out.

**Using channel with Buffers**

As we said above, a Channel instance represents a connection to a device through which we can perform I/O.

Channels do not use streams; instead, they send/receive data from/to buffers.

Part of the power of NIO comes from the fact that channels can be made nonblocking. Recall that some socket operations can block indefinitely.

Calls to methods on a nonblocking channel always return immediately. The return value of such a call indicates the extent to which the requested operation was achieved. For example, a call to accept() on a nonblocking ServerSocketChannel returns the client SocketChannel if a connection is pending and null otherwise.

Let’s construct a nonblocking TCP echo client. The I/O operations that may block include connecting, reading, and writing. With a nonblocking channel, these operations return immediately. We must repeatedly call these operations until we have successfully completed all I/O.

**Selectors**

In technical terms, a selector is a multiplexor because a single selector can manage I/O on multiple channels.

To use a selector, create it (using the static factory method open()) and register it with the channels that you wish to monitor (note that this is done via a method of the channel, not the selector). Finally, call the selector’s select() method, which blocks until one or more channels are ready for I/O or a timeout expires. When select() returns, it tells you the number of channels ready for I/O. Now, in a single thread, we can check for ready I/O on several channels by calling select(). If no I/O becomes ready after a certain amount of time, select() returns 0 and allows us to continue on with other tasks.

**Buffers in Detail**

In NIO data is read into and written from buffers. Channels read data into buffers. We then access the data through the buffer. To write data, we first fill the buffer with data in the order we wish to send it.

**Buffer Indices**

A buffer goes beyond just storing a list of elements. It has internal state that keeps track of the current position when reading data from or writing data to the buffer, as well as the end of valid data for reading, etc. To do this, each buffer maintains four indices into its list of elements

**Buffer Creation**

Typically, we create buffers either by allocation or by wrapping an array of primitives.

It is a good idea to use direct buffers only if they provide a measurable increase in performance over nondirect buffers.

**Storing and Retrieving Data**

Once you have a buffer, it’s time to use it to hold data. As “containers” for data, buffers are used for both input and output; this is different from streams, which transfer data in only one direction. We place data into a buffer using put(), and retrieve data from a buffer using get(). A channel read() implicitly calls put(), and a channel write() implicitly calls get() on the given buffer.

**Prepare Buffers: clear(), flip(), and rewind()**

Before using a buffer for input or output, we need to make sure the buffer is correctly prepared with position and limit set to the proper values

**Compacting Data in a Buffer**

The compact() operation copies the elements between position and limit to the start of the buffer, to make room for subsequent put()/read() calls. The value of position is set to the length of the copied data, the value of limit is set to the capacity, and mark becomes undefined.

**Buffer Perspective: duplicate(), slice()**

NIO provides several ways of creating a new buffer that shares content with a given buffer, but differs on the processing of the elements. Basically, the new buffer has its own independent state variables (position, limit, capacity, and mark) but shares the backing storage with the original buffer. Any changes to the new buffer are shared with the original. Think of this as an alternate perspective on the same data. Table 5.4 lists the relevant methods.

The duplicate() method creates a new buffer that shares the content of the original buffer.

Note that with buffer duplication, writing to the network and log could be done in parallel using different threads.

The slice() method creates a new buffer that shares some subsequence of the original buffer.

Channel reads and writes take only ByteBuffers; however, we may be interested in communicating using other primitive types.

The asReadOnlyBuffer() method works just like duplicate() except that all mutator methods on the new buffer will always throw a ReadOnlyBufferException. This includes all forms of put(), compact(), etc. Even calls to array() and arrayOffset() for a nondirect buffer throw this exception. Of course, changes to the non-read-only buffer that generated this read-only buffer will still be shared. Like a buffer created with duplicate(), read-only buffers have independent buffer state variables. You can use the isReadOnly() method to test if a buffer is read-only. If a buffer is already read-only, calling duplicate() or slice() will create a read-only buffer.

**Character Coding**

Recall from Chapter 3 that characters are encoded as sequences of bytes, and that there are various mappings (called charsets) between sets of characters and byte sequences. Another use of NIO buffers is to convert among various charsets. To use this facility, you need to know about two additional classes in the java.nio.charset package (we have already encountered Charset in Chapter 3): CharsetEncoder and CharsetDecoder.

**Stream (TCP) Channels in Detail**

Stream channels come in two varieties: SocketChannel and ServerSocketChannel. A SocketChannel is created by calling the open() factory method.

The most basic form of read takes a single ByteBuffer and reads up to the number of bytes remaining in the buffer. The other form of read takes an array of ByteBuffers and reads up to the number of bytes remaining in all of the buffers by filling each buffer in array order.

A ServerSocketChannel is created by calling the open() factory method.

Consider setting up a connection for a SocketChannel. If you give the open() factory method of SocketChannel a remote address, the call blocks until the connection completes. To avoid this, use the parameterless version of open(), configure the channel to be nonblocking, and call connect(), specifying the remote endpoint address. If the connection can be made without blocking, connect() returns true ; otherwise, you need some way to determine when the socket becomes connected.

With a nonblocking SocketChannel, once a connection has been initiated, the underlying socket may be neither connected nor disconnected; instead, a connection is “in progress.”

The isConnected() method allows you to determine whether the socket is connected so you can avoid having a NotYetConnectedException thrown (say, by read() or write()).

**Selectors in Detail**

**Registering Interest in Channels**

As we have seen, each selector has an associated set of channels which it monitors for specific I/O “operations of interest” to that channel. The association between a Selector and a Channel is represented by an instance of SelectionKey. (Note that a Channel instance can register more than one Selector instance, and so can have more than one associated instance of SelectionKey.) The SelectionKey maintains information about the kinds of operations that are of interest for a channel in a bitmap, which is just an int in which individual bits have assigned meanings.

**Selecting and Identifying Ready Channels**

The select() methods all return a count of how many registered channels are ready for I/O operations in their interest set to be performed. The three methods differ only in their blocking behavior. The parameterless method blocks until at least one registered channel has at least one operation in its interest set ready, or another thread invokes this selector’s wakeup() method (in which case it may return 0).

**Channel Attachments**

When a channel is ready for I/O, we often need additional information to process the request.

**Datagram (UDP) Channels**

Java NIO provides datagram (UDP) channels with the DatagramChannel class. As with the other forms of SelectableChannel we’ve seen, a DatagramChannel adds selection and nonblocking behavior and Buffer-based I/O to the capabilities of a DatagramSocket.

A DatagramChannel is created by calling the open() factory method, which creates an unbound DatagramChannel. The DatagramChannel is simply a wrapper around a basic DatagramSocket. You may directly access the particular DatagramSocket instance using the socket() method. This will allow you to call basic DatagramSocket methods to bind, set socket options, etc. When you are finished with a DatagramChannel, call the close() method.

The send() method constructs a datagram containing the data from the given ByteBuffer and transmits it to the SocketAddress specifying the destination. The receive() method prepares to accept a datagram into the specified buffer and return the address of the sender.

Another advantage of connect() is that a connected datagram channel may only receive datagrams from the specified endpoint so we don’t have to test for spurious reception.

The major difference between datagram channels and sockets is the ability of a channel to perform nonblocking I/O operations and use selectors. Selector creation, channel registration, selection, etc., work almost identically to the SocketChannel. One difference is that you cannot register for connect I/O operations, but you wouldn’t want to, since a DatagramChannel’s connect() never blocks anyway.