XNIO

Developer Guide

rev.1 (XNIO 2.0.x)

by David M. Lloyd

Part I. Using XNIO

Quick Start

1.1. TCP Client and Server

The first step when writing any XNIO application is usually to create a class which implements the `ChannelListener` interface. This is true regardless of protocol or whether it is a client or a server. In a TCP server, the `ChannelListener` instance is invoked every time a client connection is accepted. In the client, it is invoked only once when the connection to the server is established. Either way, the parameter to the listener is the newly created channel.

This primordial channel listener will generally do some connection setup work (such as checking the remote IP address against a blacklist or whitelist, and registering additional channel listeners for read and/or write events), but it should generally *not* perform any operation that can block for an extended period of time unless an `Executor` is in use (either configured when the client or server is set up [see *Section 4.2, “The XNIO Provider”*] or explicitly used by the handler implementation).

For most simple usages, there are two options for what to do next: send some data, or await the reception of some data. The simplest way to send is to use the `Channels.writeBlocking()` methods, which performs a blocking write. Since this operation may block, it should be done in an executor task.

Receiving data is generally best accomplished by way of registering a channel-readable listener via the `channel.getReadSetter().set()` method sequence, and then calling the `channel.resumeReads()` method (usually from `handleOpened()` and implementing the `handleReadable()` method. Once data is available on the channel, your read handler method will be invoked with the channel as the argument.

The read handler, like any channel listener method, should not indulge in any blocking or long-running operations as this can starve other consumers. If such an operation is required, then it should be spun off to another thread using an `Executor`. The general form for this method is something like the following code snippet:

```
public void handleEvent(final StreamChannel channel) {
    boolean ok = false;
    final ByteBuffer buffer = ByteBuffer.allocate(400);
    try {
        int c;
        while ((c = channel.read(buffer) != 0) {
            if (c == -1) {
                // Channel end-of-file
                log.info("Remote side closed the channel.");
                IoUtils.safeClose(channel);
                return;
            } else if (c == 0) {
                // Channel has no data available; indicate our further interest and return
                channel.resumeReads();
                ok = true;
                return;
            }
            buffer.flip();
            // XXX process buffer here
            // now clear the buffer for the next data
            buffer.clear();
        }
    } catch (IOException e) {
        log.error("I/O exception on read: %s", e);
        return;
    } finally {
        if (! ok) IoUtils.safeClose(channel);
    }
}
```

Following this general form is important - it is resilient against sporadic notifications which are possible on some platforms; it ensures that data is read in sequence, which is very important for stream channels; and it ensures that if the read fails, the channel is closed in an orderly fashion rather than just "hanging".

NIO Buffers

2.1. Buffers Overview

XNIO, like NIO, is based on the usage of buffers as implemented by the NIO buffer classes in the `java.nio` package. The NIO documentation defines a `java.nio.Buffer` as "a linear, finite sequence of elements of a specific primitive type". There are buffer types corresponding to every primitive type; however, as a practical matter, networking software will rarely use a buffer type other than `java.nio.ByteBuffer`.

Buffers are *mutable*, meaning that the data in the buffer is subject to alteration, as are the buffer's properties. Buffers are also *unsafe* for use in multiple threads without some type of external synchronization.

There are three primary properties of a `java.nio.Buffer`:

- the *position*, a zero-based mutable integer value representing the point in the buffer from which the next item will be read or written
- the *limit*, a zero-based mutable integer value which is used to mark the end of the buffer's data when reading or the point at which mo more data may be added when writing (this value is always greater than or equal to the position)
- the *capacity*, a zero-based fixed integer value which represents the total size of the buffer (this value is always greater than or equal to the limit)

In addition, there is one property which may be derived from these: the *remaining size*, which is equal to the difference between the position and the limit.

These properties are used to provide boundaries for data within a buffer; typically a buffer will have a larger capacity than limit (meaning that there is more space in the buffer than there is actual useful data). The position and limit properties allow the application to deal with data that is of a possibly smaller size than the buffer's total capacity.

2.2. Reading from and Writing to Buffers

Data can be read from or written to buffers in two ways: using *absolute* operations or *relative* operations. The absolute operations accept a parameter which represents the absolute position of the data to be read or written; the relative operations read or write at the current position, advancing the position by the size of the item being read or written.

When writing data to an empty buffer, either via the `putXXX()` operations or by reading from a channel into a buffer, the limit is generally set to be equal to the capacity, with the position advancing as the buffer is filled. For the sake of discussion, this state will be called the "filling" state.

Once the buffer is satisfactorily populated from the desired source, it may be *flipped* by invoking the `flip()` method on the buffer. This sets the limit to the position, and resets the position back to the start of the buffer, effectively allowing the data to be read out of the buffer again. This state will be referred to as the "flipped" state.

If a flipped buffer's data is not able to be fully consumed, the buffer may be restored to the filling state without losing any of its remaining data by way of the `compact()` method. This method effectively moves the remaining data to the beginning of the buffer, sets the position to be just after the end of this data, and resets the limit to be equal to the capacity.

A buffer may be cleared at any time by way of the `clear()` method. This method resets the position to zero, and sets the limit to be equal to the capacity. Thus the buffer is effectively emptied and restored to the filling state.

The `rewind()` method restarts the position back at zero. This allows a buffer in the flipped state which was read partially or completely to be reread in whole. A buffer in the filling state is effectively cleared by this method.

XNIO Channels

3.1. Channels Overview

In NIO terminology, a *channel* represents a connection to an entity capable of performing I/O operations. XNIO shares this definition, and in fact the channel API in XNIO has many similarities to the NIO channel API. Many operations are intercompatible between NIO and XNIO for this reason.

3.2. Channel Types

There are many types of channels; in fact, the entire `org.jboss.xnio.channels` package is dedicated to hosting the interface hierarchy therefor. The complete diagram of this hierarchy may be viewed *in the online API documentation* [http://docs.jboss.org/xnio/2.0/api/index.html?org/jboss/xnio/channels/package-summary.html]. While there are a multitude of interfaces, only a relatively small number of them are generally required to perform most tasks.

The key XNIO channel types for most network applications are:

- `StreamChannel` - a basic, bidirectional byte-oriented data channel. It extends from both of the two unidirectional parent types `StreamSourceChannel` and `StreamSinkChannel`. These types in turn extend the NIO `ScatteringByteChannel` and `GatheringByteChannel` interface types.
- `TcpChannel` - a subtype of `StreamChannel` which corresponds to a single TCP connection.
- `SslTcpChannel` - a subtype of `TcpChannel` which corresponds to a TCP connection encapsulated with SSL or TLS.
- `UdpChannel` - a message-oriented channel which represents a bound UDP socket.

In addition, the NIO `FileChannel` type may also be used and can interoperate with XNIO channel types.

3.3. Channels: Reading and Writing

Like NIO, reading and writing channels in XNIO amounts to using one of the `read` and `write` or `receive` and `send`, depending on whether the channel is a stream channel or a message (datagram) channel.

3.4. Channels: Cleaning Up and Shutting Down

The `close` method exists on all channel types. Its purpose is to release all resources associated with a channel, and ensure that the channel's close listener is invoked exactly one time. This method should *always* be called when a channel will no longer be used, in order to prevent resource starvation and possibly long-term leakage.

When a program is done sending data on a channel, its `shutdownOutput` method may be invoked to terminate output and send an end-of-file condition to the remote side. Since this amounts to a write operation on some channel types, calling this method may not be immediately successful, so the return value must be checked, like any non-blocking operation. If the method returns `true`, then the shutdown was successful; if `false`, then the shutdown cannot proceed until the channel is writable again. Be aware that once this method is called, no further writes may take place, even if the method call was not immediately successful. Even when this method returns `false`, some transmission may have occurred; the output side of the channel should be considered to be in a "shutting down" state.

When a program does not wish to receive any more input, the `shutdownInput` method may be invoked. This will cause any future data received on the channel to be rejected or ignored. This method always returns immediately. Many applications will not need to call this method, however, as they will want to consume all input. When all input has been read, subsequent `read` or `receive` method invocations will return an EOF value such as -1.

3.5. Channel Listeners

The application program is notified of events on a channel by way of the `ChannelListener` interface. A class which implements this interface is known as a *channel listener*. The interface's single method, `handleEvent`, is invoked when a specific event occurs on a channel.

By default, XNIO uses only a small number of dedicated threads to handle events. This means that in general, channel listeners are expected to run only for a brief period of time (in other words, the listener should be *non-blocking*). If a channel listener runs for an extended period of time, other pending listeners will be starved while the XNIO provider waits for the listener to complete. If your design calls for long-running listeners, then the specific service, or alternately the entire provider instance, should be configured with a `java.util.concurrent.Executor` which runs handlers in a thread pool. Such a configuration can simplify your design, at the cost of a slight increase in latency.

Registering a channel listener involves accessing the *setter* for the corresponding listener type. The setter can accept a listener, which will be stored internally, replacing any previous value, to be invoked when the listener's condition is met. The new listener value will take effect immediately. Setting a listener to `null` will cause the corresponding event to be ignored. By default, unless explicitly specified otherwise, all listeners for a channel will default to `null` and have to be set in order to receive the corresponding notification.

The `ChannelListener` interface has a type parameter which specifies what channel type the listener expects to receive. Every channel listener setter will accept a channel listener for the channel type with which it is associated; however, they will additionally accept a channel listener for any *supertype* of that channel type as well. This allows general-purpose listeners to be applied to multiple channel types, while also allowing special-purpose listeners which take advantage of the features of a more specific channel type.

There are several types of events for which a channel listener may be registered. Though the circumstances for each type may differ, the same interface is used for all of them. The types are:

- *Channel Readable* - called when readable notifications are enabled and a call to the channel's `read` or `receive` method is expected to yield useful information. This notification type is enabled by invoking the `resumeReads` method on a readable channel, and is implicitly disabled once the corresponding channel listener, if any, is invoked. It can also be explicitly disabled by invoking the `suspendReads` method.
- *Channel Writable* - called when writable notifications are enabled and a call to the channel's `write` or `send` method is expected to accept at least some data; a call to the channel's `flush` or `shutdownWrites` is expected to make progress or complete. This notification type is enabled by invoking the `resumeWrites` method on a writable channel, and is implicitly disabled once the corresponding channel listener, if any, is invoked. It can also be explicitly disabled by invoking the `suspendWrites` method.
- *Channel Closed* - called when the channel is closed via the `close` method.
- *Channel Bound* - called when a channel was newly created and bound to a local address. The new channel is passed in as the argument to the listener. Such channels will typically extend the `BoundChannel` interface.
- *Channel Opened* - called when a channel was newly created and connected to, or accepted from, a remote peer. The new channel is passed in as the argument to the listener. Such channels will typically extend the `ConnectedChannel` interface.

Not all event types are relevant for all channels, however most channel types will support notification for channel close events, and most channel types will have a way to register a listener for channel binding or opening, though the mechanism may vary depending on whether the channel in question is a client or server, TCP or UDP, etc.

3.6. Channel Blocking Modes

When using NIO, a channel may operate in a *blocking* or *non-blocking* fashion. Non-blocking I/O is achieved in NIO by way of *selectors*, which are essentially coordination points between multiple channels. However, this API is combersome and difficult to use; as such, XNIO does not use this facility, preferring instead the callback-based *listener system*.

An XNIO channel is always non-blocking; however, blocking I/O may be simulated by way of the `awaitReadable()` and `awaitWritable()` methods, which will block until the channel is expected to be readable or writable without blocking, or until the current thread is interrupted. The `ChannelInputStream` and `ChannelOutputStream` classes use this facility by way of a wrapper around the stream channel types with a blocking `InputStream` or `OutputStream` for compatibility with APIs which rely on these types.

Because of this mechanism, blocking and non-blocking operations can be intermixed freely and easily. One common pattern, for example, revolves around using blocking operations for write and non-blocking operations for read. Another pattern is to use blocking I/O in both directions, but only for the duration of a request.

3.7. Zero-Copy I/O

NIO supports so-called "zero-copy" I/O by way of two methods on the `FileChannel` class: `transferTo` and `transferFrom`. These methods accept either a readable or writable channel as an argument. Since XNIO extends the NIO channels for its stream types, you can pass XNIO stream channel types directly in to these methods.

However, most JDKs will, in reality, probe the given channel type and perform special optimizations for certain implementations. Because of this, passing in an XNIO channel type may yield poorer performance than the equivalent NIO channel would. To address this issue, the XNIO stream channel types also have `transferFrom` and `transferTo` methods which accept a `FileChannel` as an argument, and in turn can take advantage of the same optimizations.

XNIO Network Clients and Servers

4.1. XNIO Options

Most of the services provided by XNIO and its subprojects are configured with *options*. An option is identified by an instance of the `org.jboss.xnio.Option` class. Instances of this class are also associated with a specific argument type parameter; in other words, option values are strongly typed.

Options and option values are serializable, singleton constants (not unlike `enum` values), and support identity comparison (in other words, one can use the == operator to determine option equality). By convention, options always have an immutable serializable type as the argument.

There are two kinds of options: simple and sequence. A *simple option* has only a single value, typically of a primitive type such as `Integer` or `Boolean`, or an `enum` type. A *sequence option* is an option whose argument is a sequence of simple values. A special immutable `List` implementation, `org.jboss.xnio.Sequence`, is provided for this purpose.

The initial configuration of an XNIO service is usually specified using an `org.jboss.xnio.OptionMap`. This is an immutable, serializable map of options and values. Instances are constructed using a builder pattern, as follows:

```
OptionMap map = OptionMap.builder()
        .set(Options.TCP_KEEPALIVE, true)
        .set(Options.REUSE_ADDRESSES, true)
        .setSequence(Options.SSL_ENABLED_PROTOCOLS, "SSLv2", "TLSv1")
        .set(Options.RECEIVE_BUFFER, 2048)
        .getMap();
```

4.2. The XNIO Provider

To use XNIO services, one must have an XNIO provider instance (an object which implements the `org.jboss.xnio.Xnio` class). The `Xnio` class contains static methods to locate providers and create provider instances. A single provider instance can support many distinct services concurrently.

Here is an example illustrating the creation of a new provider:

```
final XnioConfiguration conf = new XnioConfiguration();
conf.setOptionMap(OptionMap.builder()
        .set(Options.READ_THREADS, 3)
        .set(Options.WRITE_THREADS, 1)
        .set(Options.CONNECT_THREADS, 1)
        .getMap());
Xnio xnio = Xnio.create(conf);
```

The returned provider will use three threads for read operations, one thread for write operations, and one thread for connect and accept operations.

A provider may be configured to run all channel listeners in a separate `java.util.concurrent.Executor` by way of the `executor` property of `XnioConfiguration`.

4.3. XNIO TCP Server

The simplest form of an XNIO TCP server is as follows:

```
TcpServer server = xnio.createTcpServer(openListener, OptionMap.EMPTY);
server.bind(new InetSocketAddress(12345));
server.bind(new InetSocketAddress(23456));
```

This creates a basic server listening on ports 12345 and 23456, which will cause the given `openListener` the be called every time an incoming connection is accepted on either port. The resulting TCP channel can then be queried to determine the actual local address.

If one wishes to control each individial binding, the `TcpServer.bind()` method's return value may be utilized. When this method is called, it returns an `IoFuture` which will return the result of the bind operation as a `BoundChannel`. This returned channel can be closed at a later point to cause only the corresponding binding to be undone.

4.4. XNIO TCP Client

4.5. XNIO TCP Acceptor

4.6. XNIO UDP Server

Chapter 5.

SSL and XNIO

XNIO Application Design Strategies

fully blocking
non-blocking read, blocking write
fully nonblokcing