An NIO.2 primer, Part 1: The asynchronous channel APIs

Learn about the new channels that support asynchronous I/O

Catherine Hope (catherine.hope@uk.ibm.com)
Apache Harmony Developer
IBM

Oliver Deakin (odeakin@uk.ibm.com)
Apache Harmony Developer
IBM

21 September 2010

The More New I/O APIs for the Java™ Platform (NIO.2) is one of the major new functional areas in Java 7, adding asynchronous channel functionality and a new file system API to the language. Developers will gain support for platform-independent file operations, asynchronous operations, and multicast socket channels. Part 1 of this two-part article focuses on the asynchronous channel APIs in NIO.2, and Part 2 covers the new file system functionality.

View more content in this series

An *asynchronous channel* represents a connection that supports nonblocking operations, such as connecting, reading, and writing, and provides mechanisms for controlling the operations after they've been initiated. The More New I/O APIs for the Java Platform (NIO.2) in Java 7 enhance the New I/O APIs (NIO) introduced in Java 1.4 by adding four asynchronous channels to the `java.nio.channels` package:

- `AsynchronousSocketChannel`
- `AsynchronousServerSocketChannel`
- `AsynchronousFileChannel`
- `AsynchronousDatagramChannel`

These classes are similar in style to the NIO channel APIs. They share the same method and argument structures, and most operations available to the NIO channel classes are also available in the new asynchronous versions. The main difference is that the new channels enable some operations to be executed asynchronously.

The asynchronous channel APIs provide two mechanisms for monitoring and controlling the initiated asynchronous operations. The first is by returning a `java.util.concurrent.Future` object, which models a pending operation and can be used to query its state and obtain the result. The second is by passing to the operation an object of a new class, `java.nio.channels.CompletionHandler`, which defines handler methods that are executed after the operation has completed. Each asynchronous channel class defines duplicate API methods for each operation so that either mechanism can be used.

This article, the first in a two-part series on NIO.2, introduces each of the channels and provides some simple examples to demonstrate their use. The examples are available in a runnable state (see Download), and you can try them out on the Java 7 beta releases available from Oracle and IBM® (both still under development at the time of this writing; see Resources). In Part 2, you'll learn about the NIO.2 file system API.

Asynchronous socket channels and futures

To start, we'll look at the `AsynchronousServerSocketChannel` and `AsynchronousSocketChannel` classes. Our first example demonstrates how a simple client/server can be implemented using these new classes. First we'll set up the server.

Server setup

An `AsychronousServerSocketChannel` can be opened and bound to an address similarly to a `ServerSocketChannel`:

```
AsynchronousServerSocketChannel server =
    AsynchronousServerSocketChannel.open().bind(null);
```

The `bind()` method takes a socket address as its argument. A convenient way to find a free port is to pass in a `null` address, which automatically binds the socket to the local host address and uses a free *ephemeral* port.

Next, we can tell the channel to accept a connection:

```
Future<AsynchronousSocketChannel> acceptFuture = server.accept();
```

This is the first difference from NIO. The accept call always returns immediately, and — unlike `ServerSocketChannel.accept()`, which returns a `SocketChannel` — it returns a `Future<AsynchronousSocketChannel>` object that can be used to retrieve an `AsynchronousSocketChannel` at a later time. The generic type of the `Future` object is the result of the actual operation. For example, a read or write returns a `Future<Integer>` because the operation returns the number of bytes read or written.

Using the `Future` object, the current thread can block to wait for the result:

```
AsynchronousSocketChannel worker = future.get();
```

Here it blocks with a timeout of 10 seconds:

```
AsynchronousSocketChannel worker = future.get(10, TimeUnit.SECONDS);
```

Or it can poll the current state of the operation, and also cancel the operation:

```
if (!future.isDone()) {
    future.cancel(true);
}
```

The `cancel()` method takes a boolean flag to indicate whether the thread performing the accept can be interrupted. This is a useful enhancement; in previous Java releases, blocking I/O operations like this could only be aborted by closing the socket.

Client setup

Next, we can set up the client by opening and connecting a `AsynchronousSocketChannel` to the server:

```
AsynchronousSocketChannel client = AsynchronousSocketChannel.open();
    client.connect(server.getLocalAddress()).get();
```

Once the client is connected to the server, reads and writes can be performed via the channels using byte buffers, as shown in Listing 1:

Listing 1. Using byte buffers for reads and writes

```
// send a message to the server
ByteBuffer message = ByteBuffer.wrap("ping".getBytes());
client.write(message).get();

// read a message from the client
worker.read(readBuffer).get(10, TimeUnit.SECONDS);
System.out.println("Message: " + new String(readBuffer.array()));
```

Scattering reads and writes, which take an array of byte buffers, are also supported asynchronously.

The APIs of the new asynchronous channels completely abstract away from the underlying sockets: there's no way to obtain the socket directly, whereas previously you could call `socket()` on, for example, a `SocketChannel`. Two new methods — `getOption` and `setOption` — have been introduced for querying and setting socket options in the asynchronous network channels. For example, the receive buffer size can be retrieved by `channel.getOption(StandardSocketOption.SO_RCVBUF)` instead of `channel.socket().getReceiveBufferSize();`.

Completion handlers

The alternative mechanism to using `Future` objects is to register a callback to the asynchronous operation. The `CompletionHandler` interface has two methods:

- `void completed(V result, A attachment)` executes if a task completes with a result of type `V`.

- `void failed(Throwable e, A attachment)` executes if the task fails to complete due to `Throwable e`.

The attachment parameter of both methods is an object that is passed in to the asynchronous operation. It can be used to track which operation finished if the same completion-handler object is used for multiple operations.

Open commands

Let's look at an example using the `AsynchronousFileChannel` class. We can create a new channel by passing in a `java.nio.file.Path` object to the static `open()` method:

```
AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open(Paths.get("myfile"));
```

New open commands for `FileChannel`

The format of the open commands for asynchronous channels has been backported to the `FileChannel` class. Under NIO, a `FileChannel` is obtained by calling `getChannel()` on a `FileInputStream`, `FileOutputStream`, or `RandomAccessFile`. With NIO.2, a `FileChannel` can be created directly using an `open()` method, as in the examples shown here.

`Path` is a new class in Java 7 that we look at in more detail in Part 2. We use the `Paths.get(String)` utility method to create a `Path` from a `String` representing the filename.

By default, the file is opened for reading. The `open()` method can take additional options to specify how the file is opened. For example, this call opens a file for reading and writing, creates it if necessary, and tries to delete it when the channel is closed or when the JVM terminates:

```
fileChannel = AsynchronousFileChannel.open(Paths.get("afile"),
    StandardOpenOption.READ, StandardOpenOption.WRITE,
    StandardOpenOption.CREATE, StandardOpenOption.DELETE_ON_CLOSE);
```

An alternative `open()` method provides finer control over the channel, allowing file attributes to be set.

Implementing a completion handler

Next, we want to write to the file and then, once the write has completed, execute something. We first construct a `CompletionHandler` that encapsulates the "something" as shown in Listing 2:

Listing 2. Creating a completion handler

```
CompletionHandler<Integer, Object> handler =
    new CompletionHandler<Integer, Object>() {
    @Override
    public void completed(Integer result, Object attachment) {
        System.out.println(attachment + " completed with " + result + " bytes written");
    }
    @Override
    public void failed(Throwable e, Object attachment) {
        System.err.println(attachment + " failed with:");
        e.printStackTrace();
    }
};
```

Now we can perform the write:

```
fileChannel.write(ByteBuffer.wrap(bytes), 0, "Write operation 1", handler);
```

The `write()` method takes:

- A `ByteBuffer` containing the contents to write
- An absolute position in the file
- An attachment object that is passed on to the completion handler methods
- A completion handler

Operations must give an absolute position in the file to read to or write from. It doesn't make sense for the file to have an internal position marker and for reads/writes to occur from there, because the operations can be initiated before previous ones are completed and the order they occur in is not guaranteed. For the same reason, there are no methods in the `AsynchronousFileChannel` API that set or query the position, as there are in `FileChannel`.

In addition to the read and write methods, an asynchronous lock method is also supported, so that a file can be locked for exclusive access without having to block in the current thread (or poll using `tryLock`) if another thread currently holds the lock.

Asynchronous channel groups

Each asynchronous channel constructed belongs to a *channel group* that shares a pool of Java threads, which are used for handling the completion of initiated asynchronous I/O operations. This might sound like a bit of a cheat, because you could implement most of the asynchronous functionality yourself in Java threads to get the same behaviour, and you'd hope that NIO.2 could be implemented purely using the operating system's asynchronous I/O capabilities for better performance. However, in some cases, it's necessary to use Java threads: for instance, the completion-handler methods are guaranteed to be executed on threads from the pool.

By default, channels constructed with the `open()` methods belong to a global channel group that can be configured using the following system variables:

- `java.nio.channels.DefaultThreadPoolthreadFactory`, which defines a `java.util.concurrent.ThreadFactory` to use instead of the default one
- `java.nio.channels.DefaultThreadPool.initialSize`, which specifies the thread pool's initial size

Three utility methods in `java.nio.channels.AsynchronousChannelGroup` provide a way to create new channel groups:

- `withCachedThreadPool()`
- `withFixedThreadPool()`
- `withThreadPool()`

These methods take either the definition of the thread pool, given as a `java.util.concurrent.ExecutorService`, or a `java.util.concurrent.ThreadFactory`. For example, the following call creates a new channel group that has a fixed pool of 10 threads, each of which is constructed with the default thread factory from the `Executors` class:

```
AsynchronousChannelGroup tenThreadGroup =
    AsynchronousChannelGroup.withFixedThreadPool(10, Executors.defaultThreadFactory());
```

The three asynchronous network channels have an alternative version of the `open()` method that takes a given channel group to use instead of the default one. For example, this call tells `channel` to use the `tenThreadGroup` instead of the default channel group to obtain threads when required by the asynchronous operations:

```
AsynchronousServerSocketChannel channel =
    AsynchronousServerSocketChannel.open(tenThreadGroup);
```

Defining your own channel group allows finer control over the threads used to service the operations and also provides mechanisms for shutting down the threads and awaiting termination. Listing 3 shows an example:

Listing 3. Controlling thread shutdown with a channel group

```
// first initiate a call that won't be satisfied
channel.accept(null, completionHandler);
// once the operation has been set off, the channel group can
// be used to control the shutdown
if (!tenThreadGroup.isShutdown()) {
    // once the group is shut down no more channels can be created with it
    tenThreadGroup.shutdown();
}
if (!tenThreadGroup.isTerminated()) {
    // forcibly shutdown, the channel will be closed and the accept will abort
    tenThreadGroup.shutdownNow();
}
// the group should be able to terminate now, wait for a maximum of 10 seconds
tenThreadGroup.awaitTermination(10, TimeUnit.SECONDS);
```

The `AsynchronousFileChannel` differs from the other channels in that, in order to use a custom thread pool, the `open()` method takes an `ExecutorService` instead of an `AsynchronousChannelGroup`.

Asynchronous datagram channels and multicasting

The final new channel is the `AsynchronousDatagramChannel`. It's similar to the `AsynchronousSocketChannel` but worth mentioning separately because the NIO.2 API adds support for multicasting to the channel level, whereas in NIO it is only supported at the level of the `MulticastDatagramSocket`. The functionality is also available in `java.nio.channels.DatagramChannel` from Java 7.

An `AsynchronousDatagramChannel` to use as a server can be constructed as follows:

```
AsynchronousDatagramChannel server = AsynchronousDatagramChannel.open().bind(null);
```

Next, we set up a client to receive datagrams broadcast to a multicast address. First, we must choose an address in the multicast range (from 224.0.0.0 to and including 239.255.255.255), and also a port that all clients can bind to:

```
// specify an arbitrary port and address in the range
int port = 5239;
InetAddress group = InetAddress.getByName("226.18.84.25");
```

We also require a reference to which network interface to use:

```
// find a NetworkInterface that supports multicasting
NetworkInterface networkInterface = NetworkInterface.getByName("eth0");
```

Now, we open the datagram channel and set up the options for multicasting, as shown in Listing 4:

Listing 4. Opening a datagram channel and setting multicast options

```
// the channel should be opened with the appropriate protocol family,
// use the defined channel group or pass in null to use the default channel group
AsynchronousDatagramChannel client =
    AsynchronousDatagramChannel.open(StandardProtocolFamily.INET,  tenThreadGroup);
// enable binding multiple sockets to the same address
client.setOption(StandardSocketOption.SO_REUSEADDR, true);
// bind to the port
client.bind(new InetSocketAddress(port));
// set the interface for sending datagrams
client.setOption(StandardSocketOption.IP_MULTICAST_IF, networkInterface);
```

The client can join the multicast group in the following way:

```
MembershipKey key = client.join(group, networkInterface);
```

The `java.util.channels.MembershipKey` is a new class that provides control over the group membership. Using the key you can drop the membership, block and unblock datagrams from certain addresses, and return information about the group and channel.

The server can then send a datagram to the address and port for the client to receive, as shown in Listing 5:

Listing 5. Sending and receiving a datagram

```
// send message
ByteBuffer message = ByteBuffer.wrap("Hello to all listeners".getBytes());
server.send(message, new InetSocketAddress(group, port));

// receive message
final ByteBuffer buffer = ByteBuffer.allocate(100);
client.receive(buffer, null, new CompletionHandler<SocketAddress, Object>() {
    @Override
    public void completed(SocketAddress address, Object attachment) {
        System.out.println("Message from " + address + ": " +
            new String(buffer.array()));
    }

    @Override
    public void failed(Throwable e, Object attachment) {
        System.err.println("Error receiving datagram");
        e.printStackTrace();
    }
});
```

Multiple clients can also be created on the same port and joined to the multicast group to receive the datagrams sent from the server.

Conclusion

NIO.2's asynchronous channel APIs provide a convenient and standard way of performing asynchronous operations platform-independently. They allow application developers to write programs that use asynchronous I/O in a clear manner, without having to define their own Java threads and, in addition, may give performance improvements by using the asynchronous support on the underlying OS. As with many Java APIs, the amount that the API can exploit an OS's native asynchronous capabilities will depend on the support for that platform.

Downloads

Description	Name	Size
Sample Java code	j-nio2-1.zip	5KB

Resources

Learn

- "*Java technology, IBM style*: A new era in Java technology" (Chris Bailey, developerWorks, April 2010): The technical architect for the IBM Java service and support organisation provides an overview of some of the features in Java 7.
- JSR 203: More New I/O APIs for the Java Platform ("NIO.2"): The NIO.2 libraries implement this Java Specification Request.
- OpenJDK 7: The project is producing an open source implementation of the next major revision of the Java SE platform, available under the GPL2 license.
- "Merlin brings nonblocking I/O to the Java platform" (Aruna Kalagnanam and Balu G, developerWorks, March 2002): Read about the nonblocking features of the NIO package introduced in Java 1.4.
- "Getting started with new I/O (NIO)" (Greg Travis, developerWorks, July 2003): This hands-on tutorial covers the NIO library in detail, from the high-level concepts to under-the-hood programming.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Java Early Access downloads from Oracle: Download the latest preview version of JDK 7 from Oracle.
- IBM SDK, Java Technology Edition, Version 7 Early Access Release: Take the latest version out for a test drive.

Discuss

- IBM SDK for Java 7.0 forum: Help shape the direction of the IBM SDK for Java 7.
- IBM Java Runtimes and SDKs discussion forum: The IBM JTC team invites you to this discussion forum to share knowledge and ask questions about your experiences of using the various IBM runtimes and kits.
- Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the authors

Catherine Hope

Catherine Hope has worked in the Hursley Runtime Deliveries department of the IBM Java Technology Centre since starting as a graduate in 2006. After three years as a system tester, she has spent the past year working as a Java class library developer and contributor to the Apache Harmony project.

Oliver Deakin

Oliver Deakin joined IBM's Java Technology Centre in 2003. For the last five years, he has been a developer, committer, and PMC member for the Apache Harmony project, an open source implementation of the Java runtime. He has experience working in many areas of the class libraries in both Java and native code. In his spare time, he enjoys rock climbing.