NIO.2 入門: 第 1 回 非同期チャネル API

非同期入出力をサポートする新たなチャネルについて学ぶ

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

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

2010年 9月 21日

More New I/O APIs for the Java™ Platform (NIO.2) は、Java 7 の新しい主要な機能分野の 1 つとして、非同期チャネル機能と新たなファイルシステム API を Java 言語に加えます。これにより、開発者にはプラットフォームに依存しないファイル操作、非同期処理、そしてマルチキャスト・ソケット・チャネルのサポートがもたらされることになります。この 2 回連載記事の第 1 回目では、NIO.2 の非同期チャネル API に焦点を当て、第 2 回目で新しいファイルシステムの機能を取り上げます。

このシリーズの他の記事を見る

非同期チャネルとは、接続、読み取り、書き込みなどのノンブロッキング処理をサポートする接続のことであり、ノンブロッキング処理が開始された後にもこれらの処理を制御するためのメカニズムを提供します。Java 7 の More New I/O APIs for the Java Platform (NIO.2) は、Java 1.4 で導入された New I/O API (NIO) を強化し、`java.nio.channels` パッケージに以下の 4 つの非同期チャネルを追加しています。

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

これらのクラスはスタイルの点では NIO のチャネル API に似ており、どちらも使用するメソッドおよび引数構造は同じです。また、NIO のチャネル・クラスで使用できるほとんどの処理は新しい非同期バージョンでも使用することができますが、NIO.2 での大きな違いは、これらの新規チャネルでは一部の処理を非同期で実行できるという点です。

非同期チャネル API には、非同期処理が開始された後、非同期処理をモニターおよび制御するための 2 つのメカニズムがあります。一方のメカニズムは、`java.util.concurrent.Future`

オブジェクトを返し、保留中の処理をモデル化するこのオブジェクトを使用して、処理の状態を問い合わせて結果を取得するという仕組みです。もう一方のメカニズムでは、`java.nio.channels.CompletionHandler` という新しいクラスのオブジェクトを非同期処理に渡し、それによって非同期処理の完了後に実行されるハンドラー・メソッドを定義します。上記の 4 つの非同期チャネル・クラスがそれぞれの処理のために定義している API メソッドは重複するため、2 つのメカニズムのどちらでも使用することができます。

NIO.2 について説明するこの 2 回連載の第 1 回目では、非同期チャネルのそれぞれを紹介し、単純な例を用いて具体的にその使用方法を説明します。サンプル・コードはすぐに実行できるように用意されているので (「ダウンロード」を参照)、Oracle および IBM® から入手できる Java 7 ベータ版リリースで試してみてください (この記事を執筆している時点では、どちらも開発中の状態です。「参考文献」を参照してください)。連載の第 2 回では、NIO.2 のファイルシステム API について学びます。

非同期ソケットチャネルと future オブジェクト

初めに話題にするのは、`AsynchronousServerSocketChannel` クラスおよび `AsynchronousSocketChannel` クラスです。最初の例では、この 2 つの新しいクラスを使用して単純なクライアント/サーバーを実装する方法を説明します。まずはサーバーのセットアップから取り掛かります。

サーバーのセットアップ

`AsychronousServerSocketChannel` は、`ServerSocketChannel` と同じように以下のようにオープンしてアドレスにバインドすることができます。

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

`bind()` メソッドは引数としてソケットのアドレスを取ります。空きポートを探すのに便利な方法は、`null` アドレスを渡すことです。するとソケットは自動的にローカル・ホスト・アドレスにバインドされ、空いている一時ポートを使用することになります。

次に、チャネルに対して接続を受け付けるように指示します。

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

ここに、NIO.2 が NIO とは異なる最初の点があります。NIO.2 では、accept を呼び出すと必ずすぐに制御が返され、`ServerSocketChannel.accept()` が `SocketChannel` を返すのとは違って、`AsynchronousServerSocketChannel.accept()` は、後で `AsynchronousSocketChannel` を取得するために使用できる `Future<AsynchronousSocketChannel>` オブジェクトを返します。`Future` オブジェクトの総称型には実際の処理の結果が入るので、例えば、読み取りまたは書き込み処理の場合には、これらの処理によって読み取ったバイト数あるいは書き込んだバイト数が返されることになるため、`Future<Integer>` が返されます。

`Future` オブジェクトを使用すると、カレント・スレッドは結果を待機する間ブロックされるようにすることができます。

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

以下は、タイムアウト時間を 10 秒に設定してカレント・スレッドをブロックする例です。

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

あるいは、処理の現状をポーリングすることも、また以下のように処理をキャンセルすることもできます。

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

`cancel()` メソッドでは、ブール値のフラグを引数に指定することによって、accept を実行しているスレッドに割り込んで処理をキャンセルできるかどうかを示します。以前の Java リリースでこのようなブロッキング I/O 処理を中止するには、ソケットをクローズするしか方法がなかったので、この機能強化は有益です。

クライアントのセットアップ

サーバーのセットアップができたので、今度はクライアントをセットアップするために `AsynchronousSocketChannel` をオープンしてサーバーに接続します。

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

クライアントがサーバーに接続された後は、バイト・バッファーを使ってチャネルを介した読み取り/書き込み処理を実行することができます (リスト 1 を参照)。

リスト 1. バイト・バッファーを使用した読み取り処理と書き込み処理

```
// 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()));
```

バイト・バッファーの配列を引数に取る、分散された読み取り/書き込み処理も非同期で行われます。

新しい非同期チャネルの API は、その下位にあるソケットを完全に抽象化します。これまでは、例えば `SocketChannel` で `socket()` を呼び出すことができましたが、この完全な抽象化により、ソケットを直接取得することはできません。そこで、非同期ネットワーク・チャネルにおけるソケットのオプションについて、問い合わせおよび設定を行うために、`getOption` および `setOption` という 2 つのメソッドが新たに導入されています。例えば、受信バッファーのサイズを調べる場合には、`channel.socket().getReceiveBufferSize()` の代わりに `channel.getOption(StandardSocketOption.SO_RCVBUF)` を使用します。

完了ハンドラー

`Future` オブジェクトを使用する代わりに、非同期処理にコールバックを登録するという方法を使用することもできます。`CompletionHandler` インターフェースには、以下の 2 つのメソッドがあります。

- `void completed(V result, A attachment)` — タスクが `V` タイプの結果で完了した場合に実行されます。
- `void failed(Throwable e, A attachment)` — `Throwable e` が原因でタスクが完了しなかった場合に実行されます。

両方のメソッドに含まれる attachment パラメーターは、非同期処理に渡されるオブジェクトです。複数の処理で同じ完了ハンドラー・オブジェクトが使用されている場合には、このパラメーターを使って、どの処理が完了したのかを追跡することができます。

オープン・コマンド

ここで、`AsynchronousFileChannel` クラスを使用した例を検討してみます。新しいチャネルを作成するため、以下のように `java.nio.file.Path` オブジェクトを静的 `open()` メソッドに渡します。

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

`FileChannel` の新しいオープン・コマンド

非同期チャネルのオープン・コマンドのフォーマットは、以前のバージョンの `FileChannel` クラスにも移植されました。NIO では `FileInputStream`、`FileOutputStream`、または `RandomAccessFile` で `getChannel()` を呼び出して `FileChannel` を取得しますが、NIO.2では、この記事の例でもわかるように `open()` メソッドを使って直接 `FileChannel` を作成することができます。

`Path` は、Java 7 で新しく導入されたクラスです (第 2 回で詳しく説明します)。上記の例では `Paths.get(String)` ユーティリティー・メソッドを使って、ファイル名を表す `String` から `Path` を作成しています。

デフォルトでは、ファイルは読み取り処理のために開かれますが、`open()` メソッドでは、ファイルを開く方法を指定する追加のオプションを引数として取ることができます。例えば以下の呼び出しでは、読み取り処理と書き込み処理を行うためにファイルを開いています。この呼び出しは、必要な場合にはファイルを作成し、チャネルがクローズされた時点、または JVM が終了した時点でファイルの削除を試みます。

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

別バージョンの `open()` メソッドでは、チャネルをさらに細かく制御して、ファイルの属性まで設定できるようになっています。

完了ハンドラーの実装

次は、ファイルに対して書き込み処理を行い、その処理が完了した時点で別の処理を実行するという例です。その場合はまず、「別の処理」をカプセル化する `CompletionHandler` を作成します (リスト 2 を参照)。

リスト 2. 完了ハンドラーの作成

```
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();
    }
};
```

この完了ハンドラーがあれば、以下のような書き込み処理を実行することができます。

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

`write()` メソッドが取る引数は以下のとおりです。

- 書き込む内容が含まれる `ByteBuffer`
- ファイル内での絶対位置
- 完了ハンドラー・メソッドに渡される attachment オブジェクト
- 完了ハンドラー

ファイルに対する読み取り/書き込み処理では、ファイル内で処理を行う絶対位置を指定する必要があります。ファイルに内部位置マーカーがあって、そのマーカーの位置から読み取り/書き込み処理を始めるのでは意味をなしません。その処理は前の処理が完了する前に開始される可能性があり、その順序は保証されていないからです。これと同じ理由から、`AsynchronousFileChannel` API には `FileChannel` にあるような、位置についての設定や問い合わせを行うメソッドはありません。

読み取り/書き込みメソッドだけでなく、ファイルを排他的アクセスのためにロックする場合の非同期ロック・メソッドもサポートされます。したがって、別のスレッドが現在ロックを保有しているとしても、カレント・スレッドでブロックする必要はありません (また、`tryLock` によるポーリングも不要です)。

非同期チャネル・グループ

作成された非同期チャネルは Java スレッドのプールを共有するチャネル・グループに組み込まれます。そして、開始された非同期 I/O 処理は、そのチャネル・グループで共有されているスレッド・プールにある Java スレッドで完了処理が行われます。

こう言うと、ちょっとだまされているかのように聞こえるかもしれません。というのも、皆さんはほとんどの非同期機能を自分で Java スレッド内に実装して、同じ振る舞いを実現することが

できるからです。そして皆さんは、より良いパフォーマンスを実現するために、オペレーティング・システムの非同期 I/O 機能だけを使用して NIO.2 を実行することができればよいのにと願っていることでしょう。しかし Java スレッドを使用せざるを得ない場合もあります。例えば、完了ハンドラー・メソッドは、プールのスレッドでの実行しか保証されていません。

デフォルトでは、`open()` メソッドによって作成されたチャネルはグローバル・チャネル・グループに組み込まれます。グローバル・チャネル・グループは、以下のシステム変数を使用して構成することができます。

- `java.nio.channels.DefaultThreadPoolthreadFactory`: デフォルトの代わりに使用する `java.util.concurrent.ThreadFactory` を定義します。
- `java.nio.channels.DefaultThreadPool.initialSize`: スレッド・プールの初期サイズを指定します。

新しいチャネル・グループを作成するには、`java.nio.channels.AsynchronousChannelGroup` の以下の 3 つのユーティリティー・メソッドを使用することができます。

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

上記のメソッドは、`java.util.concurrent.ExecutorService` または `java.util.concurrent.ThreadFactory` のいずれかとして指定されたスレッド・プールの定義を引数に取ります。例えば、以下の呼び出しが新規に作成するチャネル・グループは、スレッド数が 10 に固定されたプールを持ち、このプールのスレッドは `Executors` クラスのデフォルト・スレッド・ファクトリーによって作成されます。

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

3 つの非同期ネットワーク・チャネルには、引数に指定されたチャネル・グループをデフォルトのチャネル・グループの代わりに使用する、別バージョンの `open()` メソッドがあります。例えば以下の呼び出しで `channel` に指定している内容は、非同期処理にスレッドが必要な場合には、デフォルトのチャネル・グループの代わりに `tenThreadGroup` を使用してスレッドを取得することです。

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

独自のチャネル・グループを定義すると、非同期処理に対応するためのスレッドを詳細に制御できるだけでなく、スレッドをシャットダウンして、終了を待機するメカニズムを提供することもできます。リスト 3 に、一例を記載します。

リスト 3. チャネル・グループによるスレッドのシャットダウンの制御

```
// 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);
```

`AsynchronousFileChannel` は、`open()` メソッドがカスタム・スレッド・プールを使用するために `AsynchronousChannelGroup` ではなく `ExecutorService` を引数に取るという点で、他のチャネルとは異なります。

非同期データグラム・チャネルとマルチキャスト

最後に紹介する新しいチャネルは `AsynchronousDatagramChannel` です。このチャネルは `AsynchronousSocketChannel` に似ていますが、それとは別に説明するだけの価値があります。なぜなら、マルチキャストは NIO では `MulticastDatagramSocket` のレベルでしかサポートされませんが、NIO.2 API ではチャネル・レベルでもサポートされるようになっているからです。さらに、Java 7 からは `java.nio.channels.DatagramChannel` でもマルチキャスト機能を使用することができます。

サーバーとして使用する `AsynchronousDatagramChannel` を作成する方法は以下のとおりです。

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

次に、マルチキャスト・アドレスへのデータグラム・ブロードキャストを受信するクライアントをセットアップします。それにはまず、マルチキャスト範囲内 (224.0.0.0 から 239.255.255.255 まで) にあるアドレスを選択するとともに、すべてのクライアントがバインドできるポートを指定する必要があります。

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

さらに、使用するネットワーク・インターフェースへの参照も必要です。

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

ここで、データグラム・チャネルをオープンしてマルチキャストのオプションを設定します (リスト 4 を参照)。

リスト 4. データグラム・チャネルのオープンおよびマルチキャスト・オプションの設定

```
// 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);
```

クライアントは以下のようにしてマルチキャスト・グループに参加することができます。

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

`java.util.channels.MembershipKey` は、グループのメンバーシップを制御するための新しいクラスです。この key を使用して、メンバーシップの破棄、特定のアドレスに対するデータグラムのブロックおよびブロック解除、そしてグループとチャネルに関する情報のリターンを実行することができます。

以上の作業が完了すると、サーバーが、クライアントの受信アドレスおよびポートにデータグラムを送信できるようになります (リスト 5 を参照)。

リスト 5. データグラムの送受信

```
// 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();
    }
});
```

同じポート上に複数のクライアントを作成して、これらのクライアントをサーバーから送信されるデータグラムを受信するマルチキャスト・グループに参加させることも可能です。

まとめ

NIO.2 の非同期チャネル API は、非同期処理をプラットフォームの種類とは関係なく同じように実行できる、便利で標準的な方法です。これらの非同期チャネル API を使用することで、開発者は

独自の Java スレッドを定義することなく、非同期 I/O 処理を使用するプログラムを簡潔に作成することができ、さらに OS での非同期サポートを使用することにより、パフォーマンスも改善される可能性があります。多くの Java API と同様、NIO.2 の非同期チャネル API が OS のネイティブな非同期機能をどれだけ利用するかは、そのプラットフォームがどれだけサポートされているかによって決まります。

ダウンロード

内容	ファイル名	サイズ
Sample Java code	j-nio2-1.zip	5KB

著者について

Catherine Hope

Catherine Hope は、2006年に新卒として IBM Java Technology Centre に入社して以来、Hursley Runtime Deliveries 部門に勤務しています。システム・テスターとして 3 年間の経験を積んだ後、この 1 年は Java クラス・ライブラリーの開発者、Apache Harmony プロジェクトのコントリビューターとして活躍しています。

Oliver Deakin

Oliver Deakin は 2003年に IBM Java Technology Centre に入社して以来、Java ランタイムのオープンソース実装である Apache Harmony プロジェクトの開発者、コミッター、PMC メンバーとして働いています。Java とネイティブ・コードの両方で、さまざまなクラス・ライブラリー分野に取り組んできました。余暇は、ロック・クライミングを楽しんでいます。