NIO.2 入門: 第 2 回 ファイルシステム API

ファイルとディレクトリーに対する巡回、監視、属性の取得を可能にする新たな API を探る

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) を紹介する 2 回連載の後半となるこの記事では、新しい `java.nio.file` パッケージとそのサブパッケージに含まれる最も有用なクラスを取り上げます。NIO.2 の非同期チャネル API について説明した第 1 回と同様、この記事でもさまざまなサンプル・コードを用いて関連する概念を具体的に説明します。

このシリーズの他の記事を見る

この記事は、2 回にわたって Java 7 の More New I/O APIs for Java (NIO.2) を紹介する連載の最終回です。第 1 回で説明した非同期チャネル API と同じく、NIO.2 のファイルシステム API は、これまでの Java バージョンには欠けていた重要な I/O 処理方法を補います。NIO.2 Java Specification Request (JSR 203) では以下のように述べています。

> これまで長い間、Java プラットフォームには java.io.File クラスを超える優れたファイルシステム・インターフェースが必要でした。`java.io.File` クラスでは、各種のプラットフォームで一貫して使えるような方法でファイル名を扱うことはありません。その上、ファイル属性を効率的に取得する方法をサポートしていません。また、高度なアプケーションがファイルシステムに固有の機能 (シンボリック・リンクなど) を利用できるとしても、このクラスではそのようなアプリケーションに対応することができません。さらに、このクラスのメソッドの多くはエラーの発生時に、有用な情報を提供する例外をスローする代わりに、false を返すだけです。

そこで Java 7 ベータ版に新しく追加されたのが、以下の 3 つのファイル・システム・パッケージです。

- `java.nio.file`
- `java.nio.file.attribute`
- `java.nio.file.spi`

この記事では、これらのパッケージに含まれるクラスのうち、特に役立つ以下のクラスを抜粋して紹介します。

- `java.nio.file.Files` と `java.nio.file.FileVisitor`: ファイルシステムをウォークスルーして、特定の深さまでのファイルやディレクトリーに対して問い合わせを行い、検出された各ファイル/ディレクトリーに対してユーザー実装のコールバック・メソッドを実行することができます。
- `java.nio.file.Path` と `java.nio.file.WatchService`: 特定のディレクトリーを監視対象として登録するためのクラスです。ディレクトリーを監視するアプリケーションは、監視対象のディレクトリー内でファイルが作成、変更、あるいは削除されると通知を受け取ります。
- `java.nio.attribute.*AttributeView`: 今まで Java ユーザーが表示することのできなかったファイルおよびディレクトリーの属性を表示することができます。表示可能な属性には、ファイル所有者とグループのアクセス権限、アクセス制御リスト (ACL)、拡張ファイル属性などがあります。

ここからは、具体的な例を用いて上記のクラスの使い方を説明します。サンプル・コードはすぐに実行できる状態で用意されているので (「ダウンロード」を参照)、IBM® および Oracle から入手できる Java 7 ベータ版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、いずれの Java 7 も開発中の状態です。「参考文献」を参照してください)。

ファイル・ビジター

最初の例では、新たな `FileVisitor` API について説明します。

例えば、ディレクトリー・ツリーを再帰的にトラバースし、ツリーに含まれるファイルおよびディレクトリーごとに立ち止まって、検出された各エントリーに対して独自のコールバック・メソッドを呼び出させたいとします。Java 7 よりも前のバージョンでこのシナリオを実現するとなると、再帰的にディレクトリーをリストアップし、各ディレクトリーのエントリーを検査し、自分でコールバックを呼び出すという厄介なプロセスになっていました。Java 7 ではこのすべての処理を `FileVisitor` API を使って実行することができます。しかも、この API を使うのは、これ以上ないほどに簡単です。

まずは、独自の `FileVisitor` クラスを実装するところから始めます。このクラスには、ファイル・ビジター・エンジンがファイルシステムをトラバースするときに呼び出すコールバック・メソッドが含まれます。`FileVisitor` インターフェースを構成するメソッドは全部で 5 つあります。以下に、トラバース中に通常呼び出される順で、この 5 つのメソッドを記載します (メソッドのシグニチャーに含まれる `T` は、`java.nio.file.Path` またはスーパークラスのいずれかを表します)。

- `FileVisitResult preVisitDirectory(T dir)`: ディレクトリー内のエントリーを巡回する前に呼び出されます。このメソッドは `FileVisitResult` の列挙型の値のいずれかを返して、ファイル・ビジター API に次の処理を指示します。

- `FileVisitResult preVisitDirectoryFailed(T dir, IOException exception)`: 何らかの理由でディレクトリーを巡回できない場合に呼び出されるメソッドです。2 番目のパラメーターには、巡回に失敗する原因となった例外が指定されます。
- `FileVisitResult visitFile(T file, BasicFileAttributes attribs)`: カレント・ディレクトリー内のファイルにアクセスしている間に呼び出されます。2 番目のパラメーターには、アクセス中のファイルの属性が渡されます (ファイル属性については、この記事の「ファイル属性」セクションで詳しく説明します)。
- `FileVisitResult visitFileFailed(T file, IOException exception)`: ファイルへのアクセスに失敗すると呼び出されるメソッドです。2 番目のパラメーターには、アクセスが失敗する原因となった例外が指定されます。
- `FileVisitResult postVisitDirectory(T dir, IOException exception)`: ディレクトリーとそのすべてのサブディレクトリーの巡回が完了すると呼び出されます。exception パラメーターは、ディレクトリーの巡回が成功した場合には null となります。ディレクトリーの巡回が最後まで行かないうちに終了した場合には、その原因となった例外がこのパラメーターに指定されます。

開発者の時間を節約するため、NIO.2 には親切なことに `FileVisitor` インターフェースの実装、`java.nio.file.SimpleFileVisitor` が用意されています。このクラスは極めて基本的なもので、`*Failed()` メソッドに対しては単に例外を再スローし、それ以外のメソッドに対しては何もせずに続行します。このクラスの便利なところは、オーバーライドしたいメソッドだけを、匿名クラスを使ってオーバーライドできることです。それ以外のメソッドはデフォルトのまま実装されます。

リスト 1 に一例として、`FileVisitor` インスタンスを作成する方法を示します。

リスト 1. `FileVisitor` の実装

```
FileVisitor<Path> myFileVisitor = new SimpleFileVisitor<Path>() {

    @Override
    public FileVisitResult preVisitDirectory(Path dir) {
       System.out.println("I'm about to visit the "+dir+" directory");
       return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attribs) {

       System.out.println("I'm visiting file "+file+" which has size " +attribs.size());
       return FileVisitResult.CONTINUE;
    }

};
```

リスト 1 に記載した `FileVisitor` の実装は、巡回したディレクトリーとファイルごとにメッセージを出力するだけでなく、各ファイルの `BasicFileAttributes` から取得したファイルのサイズも通知してくれます。

次に、ファイルの巡回の出発点となる `Path` を作成します。それには以下のように、`java.nio.Paths` クラスを使用します。

```
Path headDir = Paths.get("headDir");
```

ツリーのトラバースを開始するには、`java.nio.Files` クラスの以下のいずれかのメソッドを使用することができます。

- `public static void walkFileTree(Path head, FileVisitor<? super Path> fileVisitor)`: 先頭ディレクトリーの下にあるファイル・ツリーを巡回し、`fileVisitor` に実装されたコールバック・メソッドを呼び出していきます。
- `public static void walkFileTree(Path head, Set<FileVisitOption> options, int depth, FileVisitor<? super Path> fileVisitor)`: 上記のメソッドと似ていますが、巡回オプションを指定するためのパラメーター、そしてファイル・ツリー内でトラバースする深さをディレクトリーの数で指定するためのパラメーターが追加されています。

ここでは、`walkFileTree()` メソッドを単純化した以下のバージョンを使用して、ファイル・ツリーの巡回プロセスを開始します。

```
Files.walkFileTree(headDir, myFileVisitor);
```

ディレクトリー構造は以下のようになっているとします。

```
headDir
 |--- myFile1
 |--- mySubDirectory1
 |     \myFile2
 \--- mySubDirectory2
       |--- myFile3
       \--- mySubdirectory3
             \---myFile4
```

リスト 2 は、以上の前提でサンプル・コードを実行した結果の出力です。

リスト 2. `FileVisitor` の出力

```
I'm about to visit the headDir directory
I'm about to visit the headDir\mySubDirectory2 directory
I'm about to visit the headDir\mySubDirectory2\mySubDirectory3 directory
I'm visiting file headDir\mySubDirectory2\mySubDirectory3\myFile4 which has size 2
I'm visiting file headDir\mySubDirectory2\myFile3 which has size 2
I'm about to visit the headDir\mySubDirectory1 directory
I'm visiting file headDir\mySubDirectory1\myFile2 which has size 2
I'm visiting file headDir\myFile1 which has size 2
```

ご覧のように、ディレクトリー内のファイルのトラバースは深さ優先順で行われており、必ずしもアルファベット順で行われるわけではありません。コールバック・メソッドは正常に呼び出されていて、ツリー内のすべてのファイルについて記載されていることから、すべてのディレクトリーが巡回されたことがわかります。

以上のように、わずか 15 行のコードで、指定したファイル・ツリーを巡回して、そこに含まれるファイルを検査するファイル・ビジターを作成することができました。この例は基本的なものですが、必要であれば、どんなに複雑なコールバックでも実装することができます。

ディレクトリーの監視

次に紹介する例では、新しい `WatchService` API とその関連クラスがもたらす画期的な世界を説明します。

この例のシナリオは単純なもので、特定のディレクトリー (あるいは複数のディレクトリー) 内でファイルまたはディレクトリーが作成、変更、削除されているかどうかを追跡したいという設定です。この情報があれば、GUI 画面に表示されるファイルを更新することも、場合によっては構成ファイルの変更を検出して、構成ファイルをリロードすることもできます。このシナリオを以前の Java バージョンで実現するには、別個のスレッドで動作するエージェントを実装し、そのエージェントで監視対象とするディレクトリーの内容をすべて記録し、ファイルシステムを絶えずポーリングして該当するイベントが発生したかどうかを確認しなければなりません。Java 7では、`WatchService` API がディレクトリーの監視機能を提供します。この API は、独自のファイルシステムのポーリング・エージェントを作成する厄介さをすべて排除し、可能な場合には、既存のネイティブ・システム API をベースにすることによってパフォーマンスを改善します。

最初のステップは、`java.nio.file.FileSystems` クラスを使用して `WatchService` インスタンスを作成することです。この記事ではファイルシステムの詳細までは説明しないので、ここでは、ほとんどの場合デフォルト・ファイルシステムでこのクラスの `newWatchService()` メソッドを呼び出すことになるという点だけ覚えておいてください。

```
WatchService watchService = FileSystems.getDefault().newWatchService();
```

監視サービスのインスタンスが用意できたら、次は監視対象のパスを登録します。そこで監視対象とするディレクトリーの `Path` オブジェクトを作成しますが、ファイル・ビジターの例とは少し異なる方法で作成します。以下のように作成すれば、この `File` インスタンスを後でもう一度使用することができます。

```
File watchDirFile = new File("watchDir");
Path watchDirPath = watchDirFile.toPath();
```

この例では、上記の `Path` クラスが実装する `java.nio.file.Watchable` インターフェースに定義されている `register()` メソッドを使用します。このメソッドを呼び出す `Path` オブジェクトを、特定のイベントに対して指定した `watchService` に登録するには、`WatchKey register(WatchService watchService, WatchEvent.Kind<?>... events)` を使用します。イベントが通知をトリガーするのは、そのイベントが register の呼び出しで指定されている場合のみです。

デフォルトの `WatchService` 実装では、`java.nio.file.StandardWatchEventKind` クラスが以下の 3 つの静的 `watchEvent.Kind` 実装を定義しています。`register()` 呼び出しでは、これらの実装を使用することができます。

- `StandardWatchEventKind.ENTRY_CREATE`: 登録されている `Path` 内で、ファイルまたはディレクトリーが作成されたことを示します。`ENTRY_CREATE` イベントは、登録されている Path 内でファイルの名前が変更された場合、またはファイルがこの Path 内のディレクトリー内に移動された場合にもトリガーされます。
- `StandardWatchEventKind.ENTRY_MODIFY`: 登録されている `Path` 内にあるファイルまたはディレクトリーが変更されたことを示します。どのイベントが変更に該当するのかは、プラットフォームにある程度依存しますが、ファイルの内容を実際に変更すると、常に変更イベントがトリガーされるとだけ言っておけば十分でしょう。プラットフォームによっては、ファイルの属性が変更された場合にもこのイベントがトリガーされることがあります。

- `StandardWatchEventKind.ENTRY_DELETE`: 登録されている `Path` から、ファイルまたはディレクトリーが削除されたことを示します。`ENTRY_DELETE` イベントは、ファイルの名前が変更された場合、またはファイルがディレクトリーの外に移動された場合にもトリガーされます。

以下の例では、`ENTRY_CREATE` および `ENTRY_MODIFY` イベントを監視し、`ENTRY_DELETE` イベントは無視することにします。

```
WatchKey watchKey = watchDirPath.register(watchService,
       StandardWatchEventKind.ENTRY_CREATE, StandardWatchEventKind.ENTRY_MODIFY);
```

これで `Path` が監視対象として登録されたので、`WatchService` はバックグラウンドで機能し続け、ディレクトリーを内部で監視することになります。上記に示したのと同じように、`Path` を作成して `register()` を呼び出せば、同じ 1 つの `WatchService` インスタンスで複数のディレクトリーを監視することもできます。

読者のなかの観察力の鋭い方々は、`register()` メソッド呼び出しによって、今まで記事に登場していない `WatchKey` というクラスが返されることにお気付きのことでしょう。この `WatchKey` クラスは、register() メソッド呼び出しで指定した `WatchService` に登録された内容を表します。`WatchService` はイベントがトリガーされると、その イベントに関する `WatchKey` を返すので、この register() によって返される `WatchKey` を参照するかどうかは皆さん次第です。しかし、`WatchKey` がどのディレクトリーに登録されているかを調べるためのメソッド呼び出しはないため、複数のディレクトリーを監視しているとしたら、どの `WatchKey` がどの `Path` と関連付けられているのかを調べる必要が出てくるかもしれないことに注意してください。特定の `WatchKey` とその `WatchKey` で登録しているイベントが必要なくなったら、`Path` オブジェクトの `cancel()` メソッドを呼び出すだけで Path オブジェクトの `WatchService` への登録を解除することができます。

`Path` を登録した後は、いつでも都合の良い時に `WatchService` にチェックインすれば、対象のイベントが発生したかどうかを調べることができます。`WatchService` では 3 つのメソッドを使用して、何か興味深いことが起こっていないか調べられるようになっています。

- `WatchKey poll()`: WatchService に登録してあるイベントが発生した場合は、最初に発生したイベントに関する `WatchKey` を返し、イベントが発生しなかった場合には `null` を返します。
- `WatchKey poll(long timeout, TimeUnit unit)`: 引数としてタイムアウト時間と時間単位 (`java.util.concurrent.TimeUnit`) を取ります。指定された時間内にイベントが発生すると、このメソッドが終了し、発生したイベントに関する `WatchKey` を返します。タイムアウトするまでの間に、`WatchKey` を返す必要があるようなイベントが発生しなかった場合、`nu##` を返します。
- `WatchKey take()`: 上記のメソッドと同様ですが、`WatchKey` を返せるようになるまで無期限で待機します。

上記の 3 つのメソッドのいずれかによって `WatchKey` が返された後は、`reset()` メソッドが呼び出されるまで、登録されているイベントが発生したとしても `poll()` または `take()` の呼び出しによって WatchKey が返されることはありません。`WatchService` によって `WatchKey` が返されると、`WatchKey` の `pollEvents()` メソッドを呼び出すことで `WatchEvent` のリストが返されるので、トリガーされたイベントを調べることができます。

この仕組みを説明するために、リスト 3 の単純な例で、前に登録した `WatchKey` の続きを記載します。

リスト 3. `pollEvents()` を使用する

```
// Create a file inside our watched directory
File tempFile = new File(watchDirFile, "tempFile");
tempFile.createNewFile();

// Now call take() and see if the event has been registered
WatchKey watchKey = watchService.take();
for (WatchEvent<?> event : watchKey.pollEvents()) {
   System.out.println(
   "An event was found after file creation of kind " + event.kind()
   + ". The event occurred on file " + event.context() + ".");
}
```

リスト 3 のコードを実行すると、以下の内容が出力されます。

```
An event was found after file creation of kind ENTRY_CREATE. The event occurred
 on file tempFile.
An event was found after file creation of kind ENTRY_MODIFY. The event occurred
 on file tempFile.
```

上記を見るとわかるように、新しく作成された `tempFile` に対しては期待通り `ENTRY_CREATE` イベントが出力されています。しかしそれと同時に、別のイベントも出力されています。オペレーティング・システムによっては、このようにファイルの作成または削除によって `ENTRY_MODIFY` イベントが生成されることもあります。変更イベントを受け取るように登録していなかったとしたら、OS の種類とは関係なく、`ENTRY_CREATE` イベントだけを受け取ることになります。

拡張版のサンプル・コード (このセクションの例で登録されている `WatchKey` に対して、ファイルの変更および削除を実際に行うサンプル・コード) は、サンプル・コードのダウンロードに含まれています。

ファイル属性

最後となる 3 番目の例では、`java.nio.file.attribute` パッケージに含まれるクラスを使用してファイル属性を取得および設定するための新しい API を紹介します。

これらの新規 API では、皆さんが想像するより多くのファイル属性にアクセスすることができます。これまでの Java リリースで取得できるのは基本的なファイル属性 (サイズ、最終更新時刻、隠しファイルであるかどうか、ファイルまたはディレクトリーのどちらであるか、などを示す属性) だけでした。これ以外のファイル属性を取得、あるいは変更するには、こうした処理を実行するプラットフォームに固有のネイティブ・コードを使用して実装する必要がありますが、それは簡単な作業とは言えません。素晴らしいことに、Java 7 ではこうした処理のうち、プラットフォームに固有の部分を完全に抽象化する `java.nio.file.attribute` クラスを使用して、さらに多くの属性を簡単に読み取ることや、場合によっては変更することも可能です。

新しい API には、オペレーティング・システムに固有のものも含め、7 つの属性ビューが用意されています。これらの「ビュー」クラスを使用して、それぞれのクラスに関連付けられた任意の属性を取得、設定することができます。また、各クラスには対応する属性クラスがあり、そのク

ラスに実際の属性情報が含まれます。ここからは、これらのクラスについて順に説明していきます。

`AclFileAttributeView` と `AclEntry`

`AclFileAttributeView` では、特定のファイルの ACL およびファイル所有者属性を取得、設定することができます。このクラスの `getAcl()` メソッドは、そのファイルにアクセス権が設定されている `AclEntry` オブジェクトの `List` を返します。このアクセス・リストを変更するには、`setAcl(List<AclEntry>)` メソッドを使用することができます。この属性ビューは Microsoft® Windows® システムでしか使用することができません。

`BasicFileAttributeView` と`BasicFileAttributes`

このビュー・クラスでは (その名前が示すとおり)、これまでの Java バージョンでも使用可能な属性を基にした一連の基本的なファイル属性を取得することができます。このクラスの `readAttributes()` メソッドが返す `BasicFileAttributes` インスタンスに、ファイルの最終更新時刻、最終アクセス時刻、作成時刻、サイズ、タイプ (通常のファイル、ディレクトリー、シンボリック・リンク、またはその他) といった詳細情報が含まれます。この属性ビューはすべてのプラットフォームで使用することができます。

このビューの一例を見てみましょう。特定のファイルのファイル属性ビューを取得する場合は、常に、対象とするファイルの `Path` オブジェクトを作成するところから始めます。

```
File attribFile = new File("attribFile");
Path attribPath = attribFile.toPath();
```

目的のファイル属性ビューを取得するには、`Path` の `getFileAttributeView(Class viewClass)` メソッドを使用します。例えば `attribPath` の `BasicFileAttributeView` を取得するとしたら、ただ単に、このメソッドを以下のように呼び出します。

```
BasicFileAttributeView basicView
    = attribPath.getFileAttributeView(BasicFileAttributeView.class);
```

前述のとおり、`BasicFileAttributeView` から `BasicFileAttributes` を取得するには `readAttributes()` メソッドを呼び出せばよいのです。

```
BasicFileAttributes basicAttribs = basicView.readAttributes();
```

このとおり、至って簡単に、対象とするファイルのすべての基本ファイル属性を取得できたので、このファイルを使って必要なことを行えます。`BasicFileAttributes` で変更できるのは、作成時刻、最終更新時刻、最終アクセス時刻だけです (ファイルのサイズまたはタイプを変更しても意味がありません)。時刻の属性を変更するには、`java.nio.file.attribute.FileTime` クラスを使用して新しい時刻を作成した後、`BasicFileAttributeView` の `setTimes()` メソッドを呼び出します。例えば、ファイルの最終更新時刻を 1 分前に進めるには以下のようにします。

```
FileTime newModTime
    = FileTime.fromMillis(basicAttribs.lastModifiedTime().toMillis() + 60000);
basicView.setTimes(newModTime, null, null);
```

2 つの `null` は、このファイルの最終アクセス時刻と作成時刻については変更しないという意味です。上記と同じ方法で基本属性を再確認すると、最終更新時刻は変更されている一方、作成時刻と最終アクセス時刻は変わっていないことがわかります。

`DosFileAttributeView` と `DosFileAttributes`

このビュー・クラスは、DOS に固有の属性を取得する場合に使用します (ご想像のとおり、このビューは Windows システム専用です)。このクラスの `readAttributes()` メソッドが返す `DosFileAttributes` インスタンスに、対象とするファイルが読み取り専用であるか、隠しファイルであるか、システム・ファイルであるか、アーカイブであるかの詳細情報が含まれます。このビュー・クラスには、これらのプロパティーのそれぞれを設定する `set*(boolean)` メソッドもあります。

`FileOwnerAttributeView` と `UserPrincipal`

このビュー・クラスでは、特定のファイルの所有者を取得および設定することができます。このクラスの `getOwner()` メソッドが返す `UserPrincipal` (これも同じく `java.nio.file.attribute` パッケージに含まれています) には `getName()` メソッドがあり、このメソッドが所有者の名前が含まれる `String` を返します。このビュー・クラスはまた、ファイルの所有者を変更するための `setOwner(UserPrincipal)` メソッドも提供します。このビューはすべてのプラットフォームで使用することができます。

`FileStoreSpaceAttributeView` と `FileStoreSpaceAttributes`

この覚えやすい名前のクラスは、特定のファイル・ストアに関する情報を取得する場合に使用するクラスです。このクラスの `readAttributes()` メソッドは、ファイル・ストアの合計スペース、割り当てられていないスペース、および使用可能なスペースに関する詳細情報が含まれる `FileStoreSpaceAttributes` インスタンスを返します。このビューも、すべてのプラットフォームで使用することができます。

`PosixFileAttributeView` と `PosixFileAttributes`

この UNIX® システム専用のビュー・クラスは、POSIX (Portable Operating System Interface) に固有の属性を取得および設定するために使用します。このクラスの `readAttributes()` メソッドが返す `PosixFileAttributes` インスタンスには、指定のファイルの所有者、グループ所有者、およびファイル・アクセス権 (UNIX の `chmod` コマンドで通常設定するような属性) に関する詳細情報が含まれます。また、これらの属性を変更するための `setOwner(UserPrincipal)`、`setGroup(GroupPrincipal)`、および `setPermissions(Set<PosixFilePermission>)` メソッドも用意されています。

`UserDefinedFileAttributeView` と `String`

これは Windows 専用のビュー・クラスで、ファイルの拡張属性を取得および設定する場合に使用します。拡張属性は、これが単なる名前と値のペアであり、どんな名前や値にすることもできるという点で、他の属性とは異なります。例えば、ファイルの内容を変更せずに隠しメタデータをファイルに追加したいといった場合には、拡張属性が役立ちます。このビューの `list()` メソッドは、該当するファイルの拡張属性の `String` 名の `List` を返します。

特定の属性の名前を取得した後、その属性の内容を取得するためのメソッドとして、このビュー・クラスには `size(String name)` および `read(String name, ByteBuffer dest)` がありま

す。前者は属性値のサイズを返すメソッド、後者は属性値を `ByteBuffer` に読み込むメソッドです。さらに、`write(String name, ByteBuffer source)` メソッドを使用して属性を作成、変更することも、`delete(String name)` メソッドで既存の属性を完全に削除することもできます。

このビューはおそらく、新しい属性ビューのなかでも最も興味深いものでしょう。その理由は、このビューによって、ファイルに任意の `String` 名と `ByteBuffer` 値を持つ属性を追加することができるからです。お察しのとおり、属性の値は `ByteBuffer` であるため、そこにはお望みのバイナリー・データを何でも保管することができます。

まず、以下のように属性ビューを取得します。

```
UserDefinedFileAttributeView userView
    = attribPath.getFileAttributeView(UserDefinedFileAttributeView.class);
```

次にこのビューで `list()` メソッドを呼び出して、このファイルのユーザー定義属性名のリストを取得します。

```
List<String> attribList = userView.list();
```

取得対象とする値に関連付けられた特定の属性名を取得したら、その値に適切なサイズの `ByteBuffer` を割り当て、それからこのビューの `read(String, ByteBuffer)` メソッドを呼び出します。

```
ByteBuffer attribValue = ByteBuffer.allocate(userView.size(attribName));
userView.read(attribName, attribValue);
```

これで `attribValue` には、その特定の属性に保管されていたデータが含まれることになります。独自の属性を設定するには、`ByteBuffer` を作成して、そこに任意のデータを取り込んでからこのビューの `write(String, ByteBuffer)` メソッドを呼び出せばよいのです。

```
userView.write(attribName, attribValue);
```

上記のメソッドを呼び出して属性の書き込みを行うと、該当する属性が新規に作成されるか、あるいは同じ名前の属性が既に存在する場合にはその既存の属性が上書きされます。

この記事で最後に説明する 3 番目の例は、以上のとおりです。4 つの属性ビュー (`BasicFileAttributeView`、`FileOwnerAttributeView`、`FileStoreSpaceAttributeView`、`UserDefinedAttri` を具体的に説明する完全なサンプル・コードは、サンプル・コードのダウンロードに含まれています。

まとめ

この記事で紹介した API だけでなく、NIO.2 には他にも多数のファイル関連の API があります。Java 7 では、シンボリック・リンクを作成、検査、および変更することも可能です。さらに、ファイルシステムに関する下位レベルの情報にアクセスするためのクラスや、プロバイダー (`FileSystem` および `FileStore` と呼ばれます) があらゆるタイプのファイルシステムにアクセスできるようにさせるためのクラスも新しく提供されています。

要約すると、NIO.2 によって、Java 開発者は単純で一貫性のある強力な API でファイルシステムを操作できるようになるということです。NIO.2 の目的は、ファイルおよびディレクトリーを扱う上での複雑なプラットフォーム固有の詳細を抽象化し、プログラマーに強力さと柔軟性をもたらすことです。そして NIO.2 はこの目的を見事に達成しています。

ダウンロード

内容	ファイル名	サイズ
Sample Java code for the examples in this article	j-nio2-2.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。