今まで知らなかった 5 つの事項: 日常的な Java ツール

構文解析、タイミング調整、サウンドなど、日常的な作業のための Java ツール

Ted Neward
Principal
Neward & Associates

Alex Theedom
Senior Java developer
Consultant

2017年 8月 31日
(初版 2010年 9月 14日)

Java ツールの中には、うまくカテゴリーに分類することができず、「役に立つ機能」といった注釈の下にまとめられがちなものがあります。今回の「今まで知らなかった 5 つの事項」では、(結局どこかにしまい込んでしまうことになっても) あると便利な一連のツールについて説明します。

このシリーズの他の記事を見る

かなり昔、小説家になろうと考えていた高校生の頃、私は『Writer's Digest』という雑誌を購読していました。その中の 1 つの記事に、「きちんとしまっておくにはあまりにも小さい小道具類」(キッチンの引き出しにあふれているどこにも分類しようのない小物に対するコラムニストなりの表現) に関するコラムがあったことを覚えています。そのコラムの標題は心に残っており、この連載に私が寄稿する最終回 (少なくとも、しばらくは休みます) のトピックを表す適切な表現のようです。

Java プラットフォームには、そうした「小道具類」に相当するもの、つまりほとんどの Java 開発者がその存在を知らず、ましてや使い方も知らない便利なコマンドライン・ツールやライブラリーが山のようにあります。それらの多くは、これまで「今まで知らなかった 5 つの事項」で説明してきたプログラミング・カテゴリーのどれにも当てはまりません。けれども使ってみると、なかにはさっと取り出して使えるようにしておきたいツールが見つかるかもしれません。

1. StAX

大部分の Java 開発者が初めて XML を目にした頃 (つまり 2000年頃)、XML ファイルを構文解析する方法は基本的に 2 通りありました。そのうちの 1 つの SAX パーサーは、基本的に一連のコールバック・メソッドによって起動されるイベントの巨大なステート・マシンです。もう 1 つの DOM

パーサーは XML 文書全体をメモリーにロードし、一連の独立したオブジェクトに分解します。オブジェクトは互いに関連付けられ、ツリーが形成されます。このツリーにより、その文書を XML Infoset で表現したもの全体を記述することができます。どちらのパーサーにも欠点がありました。SAX はあまりにも下位レベルのパーサーで使いにくかったのですが、DOM はリソースを消費しすぎ、大規模な XML ファイルの場合は特に、ツリー全体として消費するヒープは膨大になりました。

この連載について

皆さんは自分が Java プログラミングについて知っていると思うかもしれません。しかし実際には、ほとんどの開発者は Java プラットフォームの表面的な部分しか扱っておらず、当面の作業を完了するために十分なことしか学んでいません。この連載では、Ted Neward が Java プラットフォームのコア機能を深く掘り下げ、非常に厄介な Java プログラミングの難題の解決にも役立つ、ほとんど知られていない事実を紹介します。

幸いなことに、Java 開発者は XML ファイルを構文解析するための3 番目の方法を見つけました。この方法では、文書を一連の「ノード」としてモデリングし、これらのノードを 1 度に 1 つずつ文書ストリームから抽出して検証し、処理または破棄します。この「ノード」の「ストリーム」により、SAX と DOM の中間の方法が得られ、その方法には「Streaming API for XML (XML 用のストリーミング API)」、つまり StAX という名前が付けられました (StAX という頭字語は、元々同じ名前であった SAX パーサーと新しい API とを区別するために付けられました)。この StAX パーサーはその後 JDK に組み込まれ、現在は `javax.xml.stream` パッケージの中に組み込まれています。

StAX の使い方は非常に簡単です。`XMLEventReader` をインスタンス化して整形式の XML ファイルを指定し、(通常は `while` ループによって) 1 度に 1 つずつノードを抽出して検証します。例えば、Ant ビルド・スクリプトの中にあるすべてのターゲットをリストアップするためには、リスト 1 のようにします。

リスト 1. StAX を使ってターゲットをリストアップする

```
import java.io.*;
import javax.xml.namespace.QName;
import javax.xml.stream.*;
import javax.xml.stream.events.*;
import javax.xml.stream.util.*;

public class Targets
{
    public static void main(String[] args)
        throws Exception
    {
        for (String arg : args)
        {
            XMLEventReader xsr =
                XMLInputFactory.newInstance()
                    .createXMLEventReader(new FileReader(arg));
            while (xsr.hasNext())
            {
                XMLEvent evt = xsr.nextEvent();
                switch (evt.getEventType())
                {
                    case XMLEvent.START_ELEMENT:
                    {
                        StartElement se = evt.asStartElement();
                        if (se.getName().getLocalPart().equals("target"))
                        {
```

```
                            Attribute targetName =
                                se.getAttributeByName(new QName("name"));
                            // Found a target!
                            System.out.println(targetName.getValue());
                        }
                        break;
                    }
                    // Ignore everything else
                }
            }
        }
    }
}
```

StAX パーサーによって、すべての SAX コードと DOM コードを置き換えられるわけではありません。しかし StAX パーサーを使うことで、一部のタスクが容易になることは確かです。特に、XML 文書のツリー構造全体が必要なわけではない場合には、StAX パーサーが便利です。

StAX が提供するイベント・オブジェクトが上位レベルすぎる場合のために、StAX には `XMLStreamReader` の中に下位レベルの API も用意されています。また、おそらく XMLStreamReader ほど使い道はないかもしれませんが、StAX には `XMLEventWriter` もあり、同じく XML 出力のための `XMLStreamWriter` クラスもあります。

2. ServiceLoader

Java 開発者は、コンポーネントを使用する上で必要な知識と、コンポーネントを作成する上で必要な知識とを分離したいと思うことがよくあります。そのためには通常、そのコンポーネントが実行可能なアクションを記述するインターフェースを作成し、また何らかの中間的要素を使用して、そのコンポーネントのインスタンスを作成します。多くの開発者はそのために Spring フレームワークを使用しますが、Spring コンテナーよりもさらに軽量な別の方法があります。

`java.util` の `ServiceLoader` クラスは、JAR ファイルの中に隠れている構成ファイルを読み取ってインターフェースの実装を見つけ、選択可能な一連のオブジェクトとして、それらの実装を利用できるようにします。例えば、自分の召使い (Personal Servant) のようにタスクを実行するコンポーネントが必要な場合には、そうしたコンポーネントをリスト 2 のコードによって取得することができます。

リスト 2. IPersonalServant

```
public interface IPersonalServant
{
    // Process a file of commands to the servant
    public void process(java.io.File f)
        throws java.io.IOException;
    public boolean can(String command);
}
```

`can()` メソッドによって、自分の召使いとして提供される実装が要件を満たすかどうかを判断することができます。リスト 3 の `ServiceLoader` は基本的に要件を満たす一連の `IPersonalServant` です。

リスト 3. 要件を満たす IPersonalServant

```
import java.io.*;
```

```
import java.util.*;

public class Servant
{
    public static void main(String[] args)
        throws IOException
    {
        ServiceLoader<IPersonalServant> servantLoader =
            ServiceLoader.load(IPersonalServant.class);

        IPersonalServant i = null;
        for (IPersonalServant ii : servantLoader)
            if (ii.can("fetch tea"))
                i = ii;

        if (i == null)
            throw new IllegalArgumentException("No suitable servant found");

        for (String arg : args)
        {
            i.process(new File(arg));
        }
    }
}
```

このインターフェースの実装があるとすると、そのコードはリスト 4 のようになります。

リスト 4. Jeeves で IPersonalServant を実装する (訳注: Jeeves は英国の小説家が生み出した執事のキャラクター名)

```
import java.io.*;

public class Jeeves
    implements IPersonalServant
{
    public void process(File f)
    {
        System.out.println("Very good, sir.");
    }
    public boolean can(String cmd)
    {
        if (cmd.equals("fetch tea"))
            return true;
        else
            return false;
    }
}
```

これで、あとはこれらの実装を含む JAR ファイルを構成し、`ServiceLoader` の認識できるものにするだけです。ただしその作業は厄介なものになる可能性があります。JDK では、JAR ファイルに `META-INF/services` ディレクトリーがなければならず、このディレクトリーにテキスト・ファイルを含め、そのテキスト・ファイルの名前はインターフェース・クラスの完全修飾名と一致する必要があります。この場合、インターフェース・クラスの名前は `META-INF/services/IPersonalServant` です。このインターフェース・クラスの名前が付けられたファイルには、このインターフェースの実装名を 1 行に 1 つずつ記述する必要があります (リスト 5)。

リスト 5. META-INF/services/IPersonalServant

```
Jeeves   # comments are OK
```

幸いなことに、Ant のビルド・システム (1.7.0 以降) には `jar` タスクに対するサービス・タグがあり、このタグを使うと作業が比較的容易になります (リスト 6)。

リスト 6. Ant のビルドによる IPersonalServant

```
    <target name="serviceloader" depends="build">
        <jar destfile="misc.jar" basedir="./classes">
            <service type="IPersonalServant">
                <provider classname="Jeeves" />
            </service>
        </jar>
    </target>
```

ここまで来ると、`IPersonalServant` を呼び出してコマンドの実行を指示するのは簡単です。ただし、それらのコマンドの構文解析と実行には注意が必要です。そこで、次の「小道具」が必要になります。

3. Scanner

パーサーの作成を支援する Java ユーティリティーは数え切れないほどあります。また、関数型言語は、パーサー関数ライブラリー (パーサー・コンビネーター) の作成にある程度の成功を収めています。しかし、構文解析の対象が、CSV ファイル、あるいは空白区切りの一連のテキスト・ファイルだけである、という場合はどうでしょう。そうした場合、ほとんどのユーティリティーは機能過剰ですが、だからといって `String.split()` では機能不足です。(正規表現を使おうとする人は、「あなたは問題を抱えており、正規表現を使ってその問題を解決しようとしました。すると今度は 2 つの問題を抱えたことになります。」という昔からの言い習わしを思い出してください。)

このような場合には、Java プラットフォームの `Scanner` クラスが最適です。`Scanner` は軽量のテキスト・パーサーとして使われることを想定しているため、構造化されたテキストを強く型付けされた部分に分解するための比較的単純な API を備えています。例えば、リスト 7 のような DSL 風の一連のコマンド (Terry Pratchett 著の小説、『Discworld』 から引用したもの) がテキスト・ファイルの中に並んでいる場合を想像してみてください。

リスト 7. Igor が実行すべきタスク・リスト (訳注: Igor は『Discworld』に登場する召使い)

```
fetch 1 head
fetch 3 eye
fetch 1 foot
attach foot to head
attach eye to head
admire
```

皆さんは (あるいは、この場合は `Igor` という召使いは)、`Scanner` を使うことで、この悪辣な一連の命令を容易に構文解析することができます (リスト 8)。

リスト 8. Igor のタスクを Scanner で構文解析する

```
import java.io.*;
import java.util.*;

public class Igor
    implements IPersonalServant
```

```
{
    public boolean can(String cmd)
    {
        if (cmd.equals("fetch body parts"))
            return true;
        if (cmd.equals("attach body parts"))
            return true;
        else
            return false;
    }
    public void process(File commandFile)
        throws FileNotFoundException
    {
        Scanner scanner = new Scanner(commandFile);
        // Commands come in a verb/number/noun or verb form
        while (scanner.hasNext())
        {
            String verb = scanner.next();
            if (verb.equals("fetch"))
            {
                int num = scanner.nextInt();
                String type = scanner.next();
                fetch (num, type);
            }
            else if (verb.equals("attach"))
            {
                String item = scanner.next();
                String to = scanner.next();
                String target = scanner.next();
                attach(item, target);
            }
            else if (verb.equals("admire"))
            {
                admire();
            }
            else
            {
                System.out.println("I don't know how to "
                    + verb + ", marthter.");
            }
        }
    }

    public void fetch(int number, String type)
    {
        if (parts.get(type) == null)
        {
            System.out.println("Fetching " + number + " "
                + type + (number > 1 ? "s" : "") + ", marthter!");
            parts.put(type, number);
        }
        else
        {
            System.out.println("Fetching " + number + " more "
                + type + (number > 1 ? "s" : "") + ", marthter!");
            Integer currentTotal = parts.get(type);
            parts.put(type, currentTotal + number);
        }
        System.out.println("We now have " + parts.toString());
    }

    public void attach(String item, String target)
    {
        System.out.println("Attaching the " + item + " to the " +
            target + ", marthter!");
    }
```

```
    public void admire()
    {
        System.out.println("It'th quite the creathion, marthter");
    }

    private Map<String, Integer> parts = new HashMap<String, Integer>();
}
```

`Igor` が `ServantLoader` に登録されているとすると、`can()` 呼び出しをもっと適切なものに変更し、先ほどの `Servant` を再利用するのは簡単です (リスト 9)。

リスト 9. Igore が実行する内容

```
import java.io.*;
import java.util.*;

public class Servant
{
    public static void main(String[] args)
        throws IOException
    {
        ServiceLoader<IPersonalServant> servantLoader =
            ServiceLoader.load(IPersonalServant.class);

        IPersonalServant i = null;
        for (IPersonalServant ii : servantLoader)
            if (ii.can("fetch body parts"))
                i = ii;

        if (i == null)
            throw new IllegalArgumentException("No suitable servant found");

        for (String arg : args)
        {
            i.process(new File(arg));
        }
    }
}
```

当然ですが、実際の DSL の実装では、単純に標準出力ストリームに出力する以上のことをするはずです。どの部分をどの部分に追加するかの詳細は、読者に (そしてもちろん、忠実な `Igor` に) 任せることにします。

4. Timer

定期的にまたは 1 度だけ遅延させてタスクを実行したい場合、`java.util.Timer` クラスと `TimerTask` クラスが便利で比較的単純な手段です。

リスト 10. 遅延させて実行する

```
import java.util.*;

public class Later
{
    public static void main(String[] args)
    {
        Timer t = new Timer("TimerThread");
        t.schedule(new TimerTask() {
            public void run() {
                System.out.println("This is later");
                System.exit(0);
            }
        }, 1 * 1000);
        System.out.println("Exiting main()");
    }
}
```

`Timer` には `schedule()` メソッドのオーバーロードがいくつかあり、それによって指定のタスクを 1 回のみ実行するのか繰り返し実行するのかを指定します。また schedule() は `TimerTask` インスタンスを引数に取って起動します。`TimerTask` は本質的に `Runnable` です (TimerTask は実際に Runnable を実装しています) が、さらに 2 つのメソッドを持っており、そのうちの 1 つである `cancel()` はタスクを kill し、もう 1 つの `scheduledExecutionTime()` はタスクが起動される時刻の概略値を返します。

ただし、`Timer` がデーモン・スレッドではないスレッドを作成し、バックグラウンドでタスクを起動することに注意してください。そのためリスト 10 では、`System.exit()` を呼び出して VM を kill する必要があります。長期にわたって実行されるプログラムでは、デーモン・スレッドとして `Timer` を作成するのが最善の方法でしょう (ブール値の引数を取るコンストラクターを使ってデーモンのステータスを示します)。そうすれば VM が実行中のまま放置されることはありません。

Timer クラスには魔法のようなものは何もありませんが、Timer クラスを使うことで、バックグラウンドでタスクを起動するという意図をより明確にすることができます。また Timer によって数行の `Thread` コードが不要になるのに加え、Timer は (`java.util.concurrent` パッケージをまるごと使う準備がまだ完全には整っていない人にとって) 軽量の `ScheduledExecutorService` としても機能します。

5. JavaSound

サーバー・サイド・アプリケーションではあまり登場しませんが、サウンドは管理者の聴覚に訴えることができる便利な手段です。またサウンドは愉快なことをするには適しています。JavaSound API が Java プラットフォームに登場したのは遅かったのですが、JavaSound API は最終的にコア・ランタイム・ライブラリーに組み込まれ、`javax.sound *` パッケージの中に隠されています。1 つのパッケージは MIDI ファイルであり、もう 1 つはサンプルの音声ファイルです (一般的な .WAV ファイル・フォーマットなど)。

JavaSound の「hello world」版として、リスト 11 では音声クリップを再生します。

リスト 11. 音声クリップを再生する

```
public static void playClip(String audioFile)
{
    try
```

```
    {
        AudioInputStream audioInputStream =
            AudioSystem.getAudioInputStream(
                this.getClass().getResourceAsStream(audioFile));
        DataLine.Info info =
            new DataLine.Info( Clip.class, audioInputStream.getFormat() );
        Clip clip = (Clip) AudioSystem.getLine(info);
        clip.addLineListener(new LineListener() {
                public void update(LineEvent e) {
                    if (e.getType() == LineEvent.Type.STOP) {
                        synchronized(clip) {
                            clip.notify();
                        }
                    }
                }
            });
        clip.open(audioInputStream);

        clip.setFramePosition(0);

        clip.start();
        synchronized (clip) {
            clip.wait();
        }
        clip.drain();
        clip.close();
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}
```

このコードの大部分は非常に単純です (少なくとも、JavaSound ではこれ以上単純にできないほど単純です)。最初のステップでは、再生ファイルを含む `AudioInputStream` を作成します。このメソッドができるだけコンテキストによる影響を受けないように、クラスをロードした `ClassLoader` から、`InputStream` として再生ファイルを取得します (音声ファイルへの正確なパスが事前にわかっている場合には、`AudioSystem` も `File` または `String` を引数に取ることができます)。再生ファイルを取得したら、`DataLine.Info` オブジェクトを `AudioSystem` に渡して `Clip` を取得します。これは音声クリップを再生するための最も容易な方法です。(他の方法を使うと (例えば `SourceDataLine` を取得するなど)、音声クリップを細かく制御することができますが、単に「再生する」だけであれば、そこまでする必要はありません。)

ここから先は、`AudioInputStream` に対して `open()` を呼び出すだけでよいはずです (「はず」と言う理由は、次のセクションで説明するバグに遭遇する場合があるためです)。`start()` を呼び出すと再生を開始し、`drain()` を呼び出すと再生が完了するまで待機し、`close()` を呼び出すと AudioInputStream 用の音声ラインを解放します。再生は別スレッドで行われるため、`stop()` を呼び出すと再生が停止し、その後で `start()` を呼び出すと再生を停止した場所から再生を開始します。`setFramePosition(0)` を使うと、リセットされて最初に戻ります。

まとめ

これで雑多なツールの説明が終わりました。ここで説明したツールを多くの読者がよく知っていることは私も承知しています。しかし私のこれまでの仕事上の経験から、こうしたツールの紹介が役立つと思える人や、長い間忘れたままになっていたこうしたツールを思い出すことでメリットが得られる人が数多くいることも確かです。

私がこの連載へ寄稿するのは少し休むことにし、他の寄稿者がその人の専門分野を説明するのに任せたいと思います。でも心配しないでください。この連載であれ、あるいは他の興味深い領域を探る新たな「5 つの事項」であれ、私は戻ってくるつもりです。それまで、ぜひ Java プラットフォームの調査を続け、さらに生産性の高いプログラマーになるために役立つ、隠れた珠玉のツールを見つけてください。

著者について

Ted Neward

Ted Neward has written over 250 articles and a dozen books across many different technologies, including .NET, iOS, Java, Android, and JavaScript. He resides in Seattle with his wife, two kids, nine laptops, fourteen mobile devices, and two cats. Email him if you're interested in having him or his company work with you.

Alex Theedom

May 2017