今まで知らなかった 5 つの事項: JAR

Java Archive は単にクラスを 1 つにまとめたものではありません

Ted Neward
Principal
Neward & Associates

Alex Theedom
Senior Java developer
Consultant

2017年 8月 31日
(初版 2010年 6月 15日)

多くの Java 開発者は JAR に関して基本事項以上のことは考えもせず、単にクラスをバンドルして本番サーバーに送るために JAR を使用しているにすぎません。しかし JAR は単に ZIP ファイルをリネームしたもの以上の機能を備えています。この記事では、Spring の依存関係や構成ファイルを JAR に含めるためのヒントを始めとし、Java Archive ファイルの機能を最大限に利用する方法について学びましょう。

このシリーズの他の記事を見る

大部分の Java 開発者にとって、JAR ファイルと、その特別な仲間である WAR と EAR は、長々と続く Ant プロセスや Maven プロセスの最終的な結果にすぎません。サーバー上の適切な場所に (あるいは稀なケースでは、ユーザーのマシンに) JAR をコピーすると、後は JAR のことを忘れてしまうのが普通です。

実は、JAR はソース・コードを保管しておく以上のことができます。ただし、他にどんなことができるのか、またそうした機能を要求する方法を理解する必要があります。この連載、「今まで知らなかった 5 つの事項」の今回のヒントでは、Java Archive ファイルを (そして場合によっては WAR と EAR も) 最大限に活用する方法について、特にデプロイメント時のヒントに重点を置いて説明します。

非常に多くの Java 開発者が Spring を使用しているので (そして Spring フレームワークでは JAR の従来の使い方に関する具体的な課題を提示するので)、いくつかのヒントは Spring アプリケーションの JAR を対象に説明します。

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。

まずは、Java Archive ファイルを作成する標準的な手順を示す簡単な例から始めます。この例は後で紹介するヒントの基礎になります。

JAR に入れる

通常、JAR ファイルを作成するのはソース・コードのコンパイルが終了した後であり、`jar` コマンドライン・ユーティリティーを使用するか、またはもっと一般的には Ant の `jar` タスクを使用することで、(パッケージごとに分類された) Java コードを集めて 1 つのまとまりにします。このプロセスは非常に単純なので、ここでは説明しませんが、この記事の後の方で JAR ファイルの作成方法の話題に戻ります。とりあえず、ここでは単に `Hello` をアーカイブする必要があります。`Hello` はスタンドアロンのコンソール・ユーティリティーであり、コンソールへのメッセージ出力という、信じられないほど便利なタスクを実行します (リスト 1)。

リスト 1. コンソール・ユーティリティーをアーカイブする

```
package com.tedneward.jars;

public class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Howdy!");
    }
}
```

`Hello` ユーティリティーは大したものではありませんが、JAR ファイルについて調べるための足掛かりとしては便利です。まずコードを実行することから始めましょう。

1. 実行可能ファイルとしての JAR

.NET や C++ などの言語は歴史的に、OS と相性がよいという利点があります。つまり、単純にコマンドラインでファイル名 (`helloWorld.exe`) を指定するだけで、あるいはファイルのアイコンを GUI シェル上でダブルクリックするだけで、そのアプリケーションを起動することができます。一方、`Java` プログラミングでは、ランチャー・アプリケーション (java) が JVM をプロセスにロードして起動するため、起動対象の `main()` メソッドが含まれるクラスを指定するコマンドライン引数 (`com.tedneward.Hello`) を渡さなければなりません。

こうした追加のステップがあるため、ユーザー・フレンドリーなアプリケーションを Java で作成するのは簡単ではありません。エンド・ユーザーは、こうした要素をすべてコマンドラインから入力する必要がある (この操作は多くのエンド・ユーザーから敬遠されます) だけでなく、なぜかエンド・ユーザーが入力操作を誤り、意味不明のエラーが返されることがよくあります。

この問題を解決するためには、JAR ファイルを「実行可能」にし、その JAR ファイルを実行する際にどのクラスを起動するのかを Java ランチャーが自動で認識できるようにします。それには、JAR ファイルのマニフェスト (JAR の `META-INF` サブディレクトリーにある `MANIFEST.MF`) へのエントリー・ポイントを指定するだけでよいのです。例えば次のようにします。

リスト 2. エントリー・ポイントを指定する

```
Main-Class: com.tedneward.jars.Hello
```

マニフェストは単に名前と値のペアにすぎません。マニフェストは場合によると改行や空白に微妙に影響されるため、マニフェストを生成する方法として最も簡単なのは、JAR をビルドする際に Ant を使用する方法です。リスト 3 では Ant の `jar` タスクの `manifest` 要素を使ってマニフェストを指定しています。

リスト 3. エントリー・ポイントをビルドする

```
    <target name="jar" depends="build">
        <jar destfile="outapp.jar" basedir="classes">
            <manifest>
                <attribute name="Main-Class" value="com.tedneward.jars.Hello" />
            </manifest>
        </jar>
    </target>
```

ユーザーがこの JAR ファイルを実行するために必要なのは、コマンドラインで `java -jar outapp.jar` を使用し、この JAR ファイルの名前を指定することだけです。一部の GUI シェルでは JAR ファイルをダブルクリックしても同じです。

2. JAR に依存関係情報を含めることができる

`Hello` ユーティリティーの噂が広まったようなので、実装を多様化する必要が出てきました。Spring や Guice などの DI (Dependency Injection: 依存性注入) コンテナーはさまざまな詳細事項の処理をしてくれますが、それでも少し問題があり、DI コンテナーを使用するようにコードを変更すると、リスト 4 のような結果になります。

リスト 4. Spring を使用する

```
package com.tedneward.jars;

import org.springframework.context.*;
import org.springframework.context.support.*;

public class Hello
{
    public static void main(String[] args)
    {
        ApplicationContext appContext =
            new FileSystemXmlApplicationContext("./app.xml");
        ISpeak speaker = (ISpeak) appContext.getBean("speaker");
        System.out.println(speaker.sayHello());
    }
}
```

Spring について

このヒントは読者が依存性注入と Spring フレームワークを理解しているものとしています。

`-classpath` コマンドライン・オプションの内容は、ランチャーの `-jar` オプションによって上書きされるため、このコードを実行する際に Spring は `CLASSPATH` 上にあると同時に環境変数の中で

もその場所が指定されていなければなりません。幸いなことに JAR では、他の JAR に対する依存関係の宣言がマニフェストに現れてもよいことになっています。これを利用すると、CLASSPATH を宣言しなくても暗黙的に CLASSPATH を作成することができます (リスト 5)。

リスト 5. Spring の CLASSPATH を作成する

```
<target name="jar" depends="build">
    <jar destfile="outapp.jar" basedir="classes">
        <manifest>
            <attribute name="Main-Class" value="com.tedneward.jars.Hello" />
            <attribute name="Class-Path"
                value="./lib/org.springframework.context-3.0.1.RELEASE-A.jar
                  ./lib/org.springframework.core-3.0.1.RELEASE-A.jar
                  ./lib/org.springframework.asm-3.0.1.RELEASE-A.jar
                  ./lib/org.springframework.beans-3.0.1.RELEASE-A.jar
                  ./lib/org.springframework.expression-3.0.1.RELEASE-A.jar
                  ./lib/commons-logging-1.0.4.jar" />
        </manifest>
    </jar>
</target>
```

`Class-Path` 属性の中に、アプリケーションが依存する JAR ファイルへの相対参照が含まれていることに注意してください。この相対参照を絶対参照として作成することも、あるいはまったく接頭辞を付けずに作成することもできます。接頭辞を付けない場合には、依存関係の JAR ファイルはアプリケーションの JAR と同じディレクトリーにあるものと見なされます。

残念ながら、Ant の `Class-Path` 属性に対する `value` 属性は 1 行の中になければなりません。これは JAR のマニフェストは複数の `Class-Path` 属性という概念に対応できないからです。そのため、これらの依存関係はすべてマニフェスト・ファイルの中の 1 行の中で記述されていなければなりません。これは確かに見苦しいのですが、`java -jar outapp.jar` のようにできることには価値があります。

3. JAR を暗黙的に参照することができる

Spring フレームワークを利用するコマンドライン・ユーティリティー (または他のアプリケーション) が何種類かある場合には、すべてのユーティリティーから参照できる共通の場所に Spring の JAR ファイルを置いた方が便利な場合があります。そうすることで、同じ JAR がファイルシステム上のあちこちに複数現れてしまう問題を避けることができます。Java ランタイムにとって JAR 用の共通の場所 (「拡張ディレクトリー」と呼ばれます) は、デフォルトで JRE がインストールされたディレクトリー配下にある `lib/ext` サブディレクトリーにあります。

JRE の場所はカスタマイズ可能ですが、指定された Java 環境の中で JRE の場所がカスタマイズされることは極めて稀です。このため、`lib/ext` は JAR を保存するには無難な場所であり、lib/ext に保存すれば、明示的な指定をしなくても Java 環境の `CLASSPATH` 上に JAR が存在している、と考えてもまったく問題ありません。

4. クラスパス・ワイルドカードを使用できる

`CLASSPATH` 環境変数 (Java 開発者が何年も前に捨て去るべきであったものです) やコマンドラインの `-classpath` パラメーターがとてつもなく長くなるのを避けるために、Java 6 ではクラスパス・ワイルドカードの概念が導入されています。引数に含まれるすべての JAR ファイルをそれぞれ明

示的に起動する代わりに、クラスパスでクラスパス・ワイルドカードを使って `lib/*` と指定することで、そのディレクトリー上の JAR ファイルを (非再帰的に) 指定することができます。

残念ながら、クラスパス・ワイルドカードは先ほど説明した `Class-Path` 属性のマニフェストのエントリーには対応していません。ただし、開発者がコード生成ツールや分析ツールなどを使って行う作業の場合には、クラスパス・ワイルドカードによって確かに (サーバーを含む) Java アプリケーションの起動が容易になります。

5. JAR はコード以外のものを保持できる

Java エコシステムの構成要素の中には、環境をどのように設定するかを記述する構成ファイルに依存しているものがあまりにも多い一方で、開発者が JAR ファイルと一緒に構成ファイルをコピーするのを忘れることもよくあります。

構成ファイルの中にはシステム管理者が編集できるものもありますが、大半の構成ファイルはシステム管理者の専門領域とは程遠いため、構成ファイルの編集はデプロイメントでバグを生じさせる原因となります。合理的な解決策は構成ファイルをコードと一緒にパッケージ化することですが、JAR ファイルは基本的に ZIP ファイルの変形であるため、これは可能です。そのためには、JAR ファイルをビルドする際に単純に Ant タスクまたは `jar` コマンドラインに構成ファイルを含めればよいのです。

また、JAR には単に構成ファイルだけではなく、他のタイプのファイルを含めることもできます。例えば、`SpeakEnglish` コンポーネントから properties ファイルにアクセスする必要がある場合、それをリスト 6 のように設定することができます。

リスト 6. ランダムに応答する

```
package com.tedneward.jars;

import java.util.*;

public class SpeakEnglish
    implements ISpeak
{
    Properties responses = new Properties();
    Random random = new Random();

    public String sayHello()
    {
        // Pick a response at random
        int which = random.nextInt(5);

        return responses.getProperty("response." + which);
    }
}
```

JAR ファイルの中に `responses.properties` を入れるということは、JAR ファイルと同時にデプロイが必要なファイルが 1 つ少なくてすむということです。それを実現するには、JAR のステップの中で responses.properties ファイルを含めるだけでよいのです。

ただし、プロパティーを JAR ファイルの中に保存した後、どのようにしてプロパティーを取り出すのか、と思う人がいるかもしれません。必要なデータが同じ JAR ファイルの中にある場合には

(例えば上の例の場合など)、JAR のファイルの場所を知ろうとして `JarFile` オブジェクトを使って JAR を開いたりする必要はありません。そんなことをせず、そのクラスの `ClassLoader` を使用して、その JAR ファイルの中の「リソース」としてプロパティーを見つければよいのです。そのためにはリスト 7 のように `ClassLoader getResourceAsStream()` を使います。

リスト 7. ClassLoader によってリソースを見つける

```
package com.tedneward.jars;

import java.util.*;

public class SpeakEnglish
    implements ISpeak
{
    Properties responses = new Properties();
    // ...

    public SpeakEnglish()
    {
        try
        {
            ClassLoader myCL = SpeakEnglish.class.getClassLoader();
            responses.load(
                myCL.getResourceAsStream(
                    "com/tedneward/jars/responses.properties"));
        }
        catch (Exception x)
        {
            x.printStackTrace();
        }
    }

    // ...
}
```

この手順はあらゆる種類のリソースに適用することができます (構成ファイル、音声ファイル、グラフィックス・ファイル、等々)。この手順に従えば、ほとんどすべてのタイプのファイルを JAR にバンドルし、そのファイルを (`ClassLoader` を使って) `InputStream` として取得し、お望みの方法でそのファイルを使用することができます。

まとめ

この記事では JAR に関し、ほとんどの Java 開発者が知らない (少なくとも、これまでの JAR の歴史や事例証拠に基づけば、知らないと思われる) 事項の上位 5 つについて説明しました。こうした JAR 関連のヒントは WAR にも同じように適用できることに注意してください。ただし、一部のヒント (特に `Class-Path` 属性と `Main-Class` 属性に関するヒント) は WAR に適用してもあまり意味がありません。WAR の場合、ディレクトリーの内容はすべてサーブレット環境で取得され、またサーブレット環境には定義済みのエントリー・ポイントがあるからです。とはいえ、これらのヒントを総合すれば、「よし、まずこのディレクトリーの中のものをすべてコピーしよう」といったような態度が改まるはずです。それと同時に、Java アプリケーションのデプロイがはるかに容易になるはずです。

ダウンロード

内容	ファイル名	サイズ
Sample code for this article	j-5things6-src.zip	10KB

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