One-JARでアプリケーションの配布を単純化

カスタムのクラスローダーによるパワー・プログラミング

P. Simon Tuffs
Independent Consultant
Independent

2004年 11月 23日

Javaアプリケーションを単一のJavaアーカイブ・ファイル（JARファイル）として配布しようとしたことがある人であれば恐らく、最後のアーカイブをビルドする前に、関連のJARファイルを拡張する必要に迫られことがあるでしょう。これはやっかいなことであると同時に、ライセンス条件に反することにもなります。この記事ではSimon TuffsがOne-JARを紹介します。One-JARはカスタムのクラスローダーを使って、実行形式のJARファイル内にあるJARファイルから動的にクラスをロードするツールです。

歴史は繰り返すと言われています。最初は悲劇として、次には茶番として。私は最近、実行形式のアプリケーションを顧客に渡す時にこれを経験してしまいました。何度も経験があるのですが、必ず何かしら問題が起きるのです。アプリケーションのJARファイルを全て集め、DOSやUnix用（そしてCygwin用）の起動スクリプトを書き、そして顧客の環境変数が全て正しい場所を指すようにする時には、間違える恐れが充分あります。全てが完全にうまく行けば、アプリケーションはあるべき姿で顧客の手元に届き、そして実行します。ところがうまく行かない時には、そして普通はうまく行かないのですが、顧客側のサポートで何時間も費やす羽目になります。

私は最近、腐るほどの `ClassNotFound` 例外に困惑した顧客と話した後、もうたくさんだ、と思いました。アプリケーションを一つのJARファイルとしてパックする方法を見つけ、顧客に対してはそのアプリケーションを実行するための単純な機構（例えば `java -jar` のようなもの）を渡すようにしようと私は決心したのです。

その結果がOne-JARです。これはJavaのカスタム・クラスローダーを使った非常に単純なパッケージ化ソリューションであり、関連するJARファイルの構造を保持しつつ、単一のアーカイブ内部からアプリケーション・クラスの全てを動的にロードします。この記事では、One-JARを開発するために私がたどった過程を追いながら、実行可能アプリケーションを自己完結的なファイルとして顧客に引き渡すにはどうすべきかを解説します。

One-JARの概要

One-JARの詳細を説明する前にまず、これを構築する上での目標を議論しましょう。私はOne-JARアーカイブは下記であるべきだ、としました。

- `java -jar` 機構を使って実行できる。
- アプリケーションが必要とするすべてのファイルを含むことができる。つまりクラスもリソースも元々の（拡張していない）形式で含むことができる。
- `jar` ツールだけを使って組み立てられる、単純な内部構造を持つ。
- 元々のアプリケーションからは見えない。つまり元々のアプリケーションを、修正無しに一つのOne-JARアーカイブにパッケージできるようにする。

問題と解決策

One-JARを開発する過程で最も困難だったのは、別のJARファイルの中に含まれているJARファイルをどうやってロードするか、という問題です。 `java -jar` の起動と共に登場するJavaのクラスローダー `classloader sun.misc.Launcher$AppClassLoader` は、次の2つの仕方だけを知っています。

- JARファイルのルートにあるクラス/リソースをロードする
- `META-INF/MANIFEST.MF Class-Path` 属性によって指示されるコードベースにあるクラス/リソースをロードする

さらにこのクラスローダーは、 `CLASSPATH` に対する環境変数設定や、提供されるコマンドライン引き数 `-cp` は意図的に無視します。そして、別のJARファイルの中に含まれているJARファイルからクラスやリソースをどのようにロードするかは知りません。

One-JARが目指す目標を実現するためには、これを何とかする必要があるのは明らかです。

解決策1： 関連のJARファイルを拡張する

単一の実行形式JARファイルを作るために私が最初に試みたのは、当然なこと、つまり出荷可能なJARファイル（main.jarと呼ぶことにします）内にある関連のJARファイルを拡張することでした。 `com.main.Main` と呼ばれるアプリケーション・クラスがあり、これが `com.a.A` (a.jar内) と `com.b.B` (b.jar内) という2つのクラスに依存しているとすると、One-JARファイルは下記のように見えます。

```
main.jar
|  com/main/Main.class
|  com/a/A.class
|  com/b/B.class
```

`A.class` が元々はa.jarから来たものであるという事実は失われてしまいます。 `B.class` の元々の位置も同じです。これは些細なことに見えるかも知れませんが、すぐ後で説明する通り、現実に問題を起こす可能性があるのです。

One-JARとFJEP

最近リリースされたFJEP (FatJar Eclipse Plugin) と呼ばれるツールは、フラットになったJARファイルを直接Eclipse内部で構築できるようになっています。One-JARはFatJarと統合され、JARファイルを拡張することなくJARファイルを埋め込めるようになっています。これについてさらに詳しくは、 参考文献 を見てください。

関連のJARファイルをファイルシステムに拡張してフラットな構造を作るのは非常に時間がかかります。また、関連のクラスを拡張して再度アーカイブするために、Antのようなビルド・ツールで作業することが必要になります。

この些細な問題とは別に、関連のJARファイルを拡張することに伴う深刻な問題にぶつかってしまいました。その問題とは下記の2つです。

- a.jarとb.jarが同じパス名（例えば `log4j.properties` ）を持つリソースを含んでいる場合には、どちらを選ぶのか？
- b.jarのライセンスが、無修正の形式で再配布することを明確に要求している場合にはどうするのか？ そのライセンスに違反しない限り、このように拡張することはできません。

私はこうした制限から、別の手法が必要だと判断しました。

解決策2： MANIFEST Class-Path

私はjava -jarローダー内の機構を調べることにしました。java -jarローダーは、アーカイブ内にあるMETA-INF/MANIFEST.MFという名前の、特別なファイル内部で規定されているクラスをロードします。私は `Class-Path` と呼ばれるプロパティを規定することで、ブートストラップ・クラスローダーに他のアーカイブを追加できるだろう、と思ったのです。そうしたOne-JARファイルは次のように見えます。

```
main.jar
|  META-INF/MANIFEST.MF
|  +  Class-Path: lib/a.jar lib/b.jar
|  com/main/Main.class
|  lib/a.jar
|  lib/b.jar
```

注意と手掛かり

URLClassloader は sun.misc.Launcher$AppClassLoader のベース・クラスです。sun.misc.Launcher$AppClassLoaderはやや意味深長なURL構文をサポートしており、この構文を使うとJARファイル内部のリソースを参照できるようになります。この構文は次のように使います。jar:file:/fullpath/main.jar!/a.resource

理論的にはJARファイル内部にあるJARファイル内へのエントリーを取得するには、例えば `jar:file:/fullpath/main.jar!/lib/a.jar!/a.resource` のようなものを使う必要がありますが、残念ながら、これはうまく行きません。JARファイルのプロトコル・ハンドラーは、最後の「！」セパレーターのみをJARファイルを指すものとして扱うのです。

これは動作するのでしょうか？ いや、まあmain.jarファイルを他の場所に移動して実行しようとするまでは、動作するように見えたのです。私はmain.jarをアセンブルするためにlibという名前のサブディレクトリーを作り、その中にa.jarとb.jarを押し込みました。残念ながらアプリケーション・クラスローダーは単純に、ファイルシステムから関連JARファイルを拾い出してしまい、埋め込まれたJARファイルからはクラスをロードしなかったのです。

これをなんとかするために、やや意味深長な `jar:!/` 構文（「注意と手掛かり」を見てください）で `Class-Path` を様々に変形させて使ってみたのですが、どれも動かすことはできませんでした。できたことといえば、a.jarとb.jarを別々に、main.jarと共にファイルシステムの中にばらまくことだけでした。しかし、これこそ正に避けたいと思っていたことなのです。

JarClassLoaderを入力する

この時点で私も苛立ってきました。どうしたらアプリケーションが、そのアプリケーション自体のJARファイル内にあるlibディレクトリーから、そのアプリケーションのクラスをロードするよう

になるでしょう？ 私は、これはカスタムのクラスローダーを書くしかない、と結論しました。カスタムのクラスローダーは、軽い考えで書けるようなものではありません。書くこと自体は決して複雑ではないのですが、クラスローダーはアプリケーションに対して重大な影響があるので、失敗が起きた時にそれを診断したり解釈したりすることが困難になるのです。クラスローダーに関して完全に説明するのはこの記事の範囲外ですが（ 参考文献 を見てください）、これから先の議論がよく理解できるように、基本的な概念の幾つかは説明しておきましょう。

クラスをロードする

クラスが未知であるようなオブジェクトにJVMが出会うと、JVMはクラスローダーを呼び出します。クラスローダーの仕事は、（クラス名に基づいて）そのクラスに対するバイトコードを見つけ、そうしたバイトをJVMに渡すことです。JVMはそのバイトをシステムの他の部分にリンクし、実行しているコードが新しいクラスを使えるようにします。JDKで致命的重要性を持つクラスは `java.lang.Classloader` であり、そして `loadClass` メソッドはおよそ次のようなものです。

```
    public abstract class ClassLoader {
        ...
        protected synchronized Class loadClass(String name, boolean resolve)
            throws ClassNotFoundException {...}
    }
```

`ClassLoader` クラスへのエントリー・ポイントの中心となるのが `loadClass()` メソッドです。`ClassLoader` は抽象クラスですが、何の抽象メソッドも宣言しないことに注意してください。そのため `loadClass()` こそが注目すべきメソッドである、という手掛かりが無いことになります。実は、これは注目すべき中心的なメソッドではないのです。古き良きJDK 1.1のクラスローダーの時代には、実質的にクラスローダーを拡張できる所は `loadClass()` だけでした。ところがJDK 1.2以降は、 `loadClass()` が既にしていること（下記がそのリストです）をそのままにしておくのが一番なのです。

- そのクラスが既にロードされているかどうかをチェックする
- 親のクラスローダーがそのクラスをロードできるかどうかをチェックする
- `findClass(String name)` を呼んで、派生クラスローダーにそのクラスをロードさせる

`ClassLoader.findClass()` の実装というのは新しい `ClassNotFoundException` を投げることであり、カスタムのクラスローダーを実装する時に最初に注目すべきメソッドがClassLoader.findClass() の実装なのです。

JARファイルがJARファイルでないのはいつなのか？

JARファイル内部にあるJARファイル内にクラスをロードする（先ほどの通り、致命的な問題です）ためにはまず、最上位レベル（上記のmain.jar）のJARファイルを開いて読み取れる必要があります。ところが私は `java -jar` 機構を使っていたので、 `java.class.path` システム・プロパティにある最初の（そして唯一の）要素がOne-JARファイルのフル・パス名であることが分かったのです！ そこに行き着くには次のようにします。

```
    jarName = System.getProperty("java.class.path");
```

次のステップは、アプリケーションのJARファイル・エントリー全てに対して繰り返しを行い、メモリーにロードしてくることです。これをリスト1に示します。

リスト1. 埋め込まれたJARファイルを見つけるために繰り返しを行う

```
    JarFile jarFile = new JarFile(jarName);
    Enumeration enum = jarFile.entries();
    while (enum.hasMoreElements()) {
        JarEntry entry = (JarEntry)enum.nextElement();
        if (entry.isDirectory()) continue;
        String jar = entry.getName();
        if (jar.startsWith(LIB_PREFIX) || jar.startsWith(MAIN_PREFIX)) {
            // Load it!
            InputStream is = jarFile.getInputStream(entry);
            if (is == null)
                throw new IOException("Unable to load resource /" + jar + " using " + this);
            loadByteCode(is, jar);
            ...
```

`LIB_PREFIX` は文字列lib/に対して評価を行い、そしてMAIN_PREFIXはmain/に対して評価を行うことに注意してください。私はループの中で、lib/またはmain/で始まるエントリーのバイトコードをメモリーにロードし、それをクラスローダーが使えるようにして、それ以外のJARファイル・エンントリーは無視しようと思ったのです。

メイン・ディレクトリー

lib/ サブディレクトリーの役割についてはお話ししましたが、ではこのmain/ ディレクトリーは何のためにあるのでしょう？ 簡単に言うと、クラスローダーのデリゲーション・モードは、メイン・クラス `com.main.Main` が（自分が依存する）ライブラリー・クラスを見つけられるように、メイン・クラスをそれ自身のJARファイルの中に置くように要求するのです。そうすると新しいJARファイルは次のようになります。

```
one-jar.jar
 |  META-INF/MANIFEST.MF
 |  main/main.jar
 |  lib/a.jar
 |  lib/b.jar
```

上記のリスト1で、 `loadByteCode()` メソッドはJARファイル・エントリーのストリームとエントリー名を受け取り、そのエントリーのバイトをメモリーにロードし、そのエントリーがクラスを表すのかリソースを表すのかによって、2つまでの名前を割り当てます。これを説明する一番良い方法は、例を見ることでしょう。a.jarがクラス `A.class` とリソース `A.resource` を含んでいるとします。One-JARクラスローダーは、クラスに対しては一対のキーと値を持ち、リソースに対しては2つのキーを持つ、次のような `JarClassLoader.byteCode` という名前の `Map` 構造を構築します。

図1. メモリー内でのOne-JARの構造

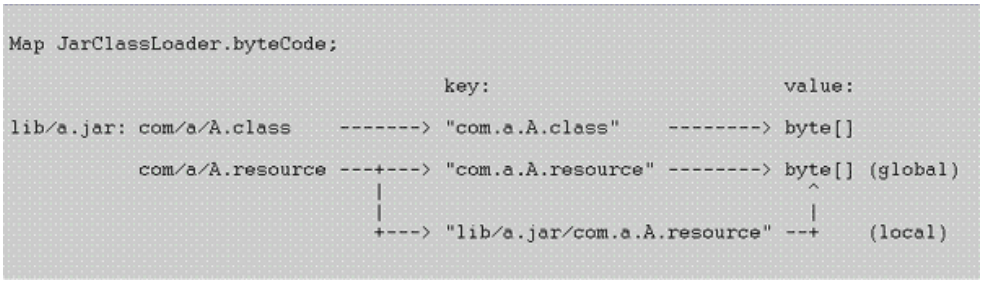

図1を穴の開くほど見ていれば、クラス・エントリーはそのクラス名に基づいてキー分けされ、リソースはグローバル名とローカル名という対でキー分けされていることが分かるでしょう。こ

の機構はリソースと名前との競合を解決するために使われています。つまり、もし2つのライブラリーJARファイルが同じグローバル名を持つリソースを定義している場合には、呼び出し側のスタック・フレームに基づいてローカル名が使われるのです。さらに詳しくは 参考文献 を見てください。

クラスを見つける

`findClass()` メソッドのところでクラスローダーの概要説明を中断したことを思い出してください。メソッド `findClass()` はクラスの名前を `String` として取り、その名前が表すバイトコードを見つけて定義する必要があります。 `loadByteCode` が親切にもクラス名とそのクラスのバイトコードとの間にMapを構築しているので、これを実装するのは非常に単純です。リスト2に示すように、クラス名に基づいて単純にバイトコードを検索し、 `defineClass()` を呼びます。

リスト2. findClass() の概要

```
    protected Class findClass(String name) throws ClassNotFoundException {
        ByteCode bytecode = (ByteCode)JarClassLoader.byteCode.get(name);
        if (bytecode != null) {
            ...
            byte bytes[] = bytecode.bytes;
            return defineClass(name, bytes, pd);
        }
        throw new ClassNotFoundException(name);
    }
```

リソースをロードする

One-JARの開発期間中、考え方の証明として実際に動作させることができたのは `findClass` が最初でした。ところが、より複雑なアプリケーションを展開し始めると、クラスのローディングだけではなく、リソースのローディングも処理する必要があることが分かりました。ここから話がややこしくなります。リソースを検索するために `ClassLoader` の中でオーバーライドする適切なメソッドを考えながら、私は自分にとって一番馴染みのあるものを選びました。これをリスト3に示します。

リスト3. getResourceAsStream()メソッド

```
    public InputStream getResourceAsStream(String name) {
        URL url = getResource(name);
        try {
            return url != null ? url.openStream() : null;
        } catch (IOException e) {
            return null;
        }
    }
```

この時点で警告が鳴っていたはずです。しかし私はとにかく、リソースを検索するのになぜURLが使われているのか理解することができませんでした。ですから私はこの実装を無視し、私独自のものを挿入したのです。これをリスト4に示します。

リスト4. One-JARでのgetResourceAsStream() の実装

```
    public InputStream getResourceAsStream(String resource) {
        byte bytes[] = null;
        ByteCode bytecode = (ByteCode)byteCode.get(resource);
        if (bytecode != null) {
            bytes = bytecode.bytes; }
        ...
        if (bytes != null) {
            return new ByteArrayInputStream(bytes);
        }
        ...
        return null;
    }
```

最後の難関

私が新しく実装した `getResourceAsStream()` メソッドはうまく動作するように見えました。ところがURL `url = object.getClass().getClassLoader().getResource()` パターンを使ったリソースをロードするアプリケーションをOne-JARしようとするとうまく行かず、おかしくなってしまうのです。なぜなのでしょう？ `ClassLoader` のデフォルト実装が返すURLはヌルなので、呼び出し側のコードを壊してしまうのです。

この時点で、すべてが本当に混乱してきました。lib/ ディレクトリーにあるJARファイル内部にあるリソースを参照するために、どんなURLを使うべきかを考えなくてはならなくなったのです。それは例えば `jar:file:main.jar!lib/a.jar!com.a.A.resource` のようなものなのでしょうか？

私は考えつく限りの組み合わせを試してみましたが、どれもうまく行きません。とにかく `jar:` 構文は、ネストしたJARファイルをサポートしません。そのためOne-JARの進め方そのものが、明らかに行き止まりにぶつかってしまったのです。大部分のアプリケーションは `ClassLoader.getResource` を使わないようですが、一部は明らかに使っています。私としては「アプリケーションが `ClassLoader.getResource` を使っている場合にはOne-JARは使えません」という例外を作りたくありません。

そして最後に解決策が・・・！

jar:構文と格闘しているうちに、私はJava Runtime EnvironmentがURLプレフィックスをハンドラーにマップする機構に行き当たりました。これが `findResource` 問題を解決するための手掛かりになりました。単純に、 `onejar:` と呼ばれる独自のプロトコル・プレフィックスを作り出せば良いのです。そうすれば新しいプレフィックスをプロトコル・ハンドラーにマップすることができ、リソースに対するバイト・ストリームを返せるようになります。これをリスト5に示します。リスト5はJarClassLoaderというファイルと、新しいcom/simontuffs/onejar/Handler.javaというファイルの2つを表していることに注意してください。

リスト5. findResourceとonejar: プロトコル

```
com/simontuffs/onejar/JarClassLoader.java

    protected URL findResource(String $resource) {
        try {
            // resolve($resource) returns the name of a resource in the
            // byteCode Map if it is known to this classloader.
            String resource = resolve($resource);
            if (resource != null) {
                // We know how to handle it.
```

```
                return new URL(Handler.PROTOCOL + ":" + resource);
            }
            return null;
        } catch (MalformedURLException mux) {
            WARNING("unable to locate " + $resource + " due to " + mux);
        }
        return null;
    }

com/simontuffs/onejar/Handler.java

    package com.simontuffs.onejar;
    ...
    public class Handler extends URLStreamHandler {
        /**
         * This protocol name must match the name of the package in which this class
         * lives.
         */
        public static String PROTOCOL = "onejar";
        protected int len = PROTOCOL.length()+1;

        protected URLConnection openConnection(URL u) throws IOException {
            final String resource = u.toString().substring(len);
            return new URLConnection(u) {
                public void connect() {
                }
                public InputStream getInputStream() {

                    // Use the Boot classloader to get the resource.  There
                    // is only one per one-jar.
                    JarClassLoader cl = Boot.getClassLoader();
                    return cl.getByteStream(resource);
                }
            };
        }
    }
```

JarClassLoaderをブートストラップする

ここで恐らく皆さんは最後の疑問を抱いているでしょう。 `JarClassLoader` がOne-JARファイルからクラスのロードを開始できるように、そもそもどうやって `JarClassLoader` を起動シーケンスの中に挿入したのでしょうか？ 厳密な説明はこの記事の範囲外ですが、基本的には、 `META-INF/MANIFEST.MF/Main-Class` 属性としてメイン・クラス `com.main.Main` を使う代わりに、 `Main-Class` 属性として規定されている新しいブートストラップ・メイン・クラス `com.simontuffs.onejar.Boot` を作ったのです。この新しいクラスは次のことをします。

- 新しい `JarClassLoader` を作る。
- 新しいローダーを使ってmain/main.jarから（ main.jarにある `META-INF/MANIFEST.MF Main-Class` エントリーに基づいて ） `com.main.Main` をロードする。
- クラスをロードし、反映を使って `main()` を呼び出すことにより `com.main.Main.main(String[])` （ あるいは `main.jar/MANIFEST.MF` ファイルにある `Main-Class` の名前であればどんな名前でも ）を呼ぶ。One-JARコマンドラインで渡される引き数は、変更されることなく、アプリケーションのメイン・メソッドに渡される。

まとめ

これまでの説明が難しすぎると思えても心配する必要はありません。One-JARがどのように動作するかを理解するよりも、One-JARを使う方がずっと簡単なのです。FatJar Eclipse Plugin（ 参考文献

のFJEPを見てください）の登場によって、今やEclipseのユーザーであれば、Wizardのチェックボックスを選択するだけでOne-JARアプリケーションを作ることができるのです。依存するライブラリーはlib/ ディレクトリーに置かれ、メイン・プログラムとクラスはmain/main.jarに置かれ、そしてMETA-INF/MANIFEST.MFファイルは自動的に書かれます。JarPlug（これも 参考文献 を見てください）を使えば、ビルドしたJARファイルの内部を見ることができ、IDE内部からJARファイルを起動することができるのです。

全体的に見ればOne-JARは、出荷用にアプリケーションをパッケージする上での問題に対する、単純かつ強力なソリューションです。ただし、あらゆるアプリケーション・シナリオに対して有効なわけではありません。例えば、アプリケーションが、その親に対して権限委任しない古い形式のJDK 1.1クラスローダーを使っている場合には、クラスローダーはネストしたJARファイル内にあるクラスの検索に失敗します。これは、そのやっかいなクラスローダーを修正するために「ラップする」クラスローダーをビルドして展開することで解決できますが、それはJavassistやByte Code Engineering Library (BCEL) などのツールでバイトコード操作の技術を使うことを意味します。

また、埋め込みのアプリケーションやWebサーバーで使われている、特定なタイプのクラスローダーでも問題にぶつかるかも知れません。具体的に言うと、最初に親のクラスローダーに権限委任しないクラスローダーや、ファイルシステム中のコードベースを探すようなクラスローダーでは、問題が起きる可能性があります。そうした場合は、One-JARにはJARファイル・エントリーをファイルシステムに拡張する機構があることが役に立つはずです。この機構はMETA-INF/MANIFEST.MFファイルにある `One-JAR-Expand` 属性が制御します。あるいは、バイトコード操作を使えば、関連JARファイルの整合性を崩さずにクラスローダーをオンザフライで修正することもできます。この手法を選ぶのであれば、恐らくそれぞれの場合で、カスタム化したラッピング・クラスローダーが必要になるでしょう。

FatJar Eclipse PluginとJarPlugをダウンロードするには、そしてOne-JARについてさらに学ぶには、参考文献 を見てください。

著者について

P. Simon Tuffs

P. Simon Tuffs博士は独立のコンサルタントで、現在はJavaのWebサービスのスケーラビリティーを専門にしています。また、暇を見つけてはOne-JARなどのオープン・ソースのプロジェクトを構築したりリリースしたりしています。Tuffs博士とその成果について詳しくはwww.simontuffs.comを見てください。