Java プログラミングのダイナミックス: 第 5 回 オンザフライでクラスを変換する

Javassist を使用してロード時にクラスを変更する方法を学ぶ

Dennis Sosnoski
President
Sosnoski Software Solutions, Inc.

2004年 2月 03日

しばらくご無沙汰していましたが、Java プログラミングのダイナミックスシリーズ第 5 回で DennisSosnoski が戻ってきました。コードの振る舞いを変更するために Java クラスファイルを変換するプログラムを書く方法を見てきましたが、この回では、Dennis はアスペクト指向の柔軟な「ジャストインタイム」機能を扱うために Javassist フレームワークを使用して、実際のクラスのロード処理と変換を組み合わせる方法を説明します。この手法は、実行時に変更したいことを決定し、プログラムを実行するたびに、異なる修正を行なわせることが可能です。途中、さらに JVM へのクラスのロード処理の一般的な問題についてより深く見て行きます。

このシリーズの他の記事を見る

第 4 回の「Javassist でのクラス変換」では、修正済のクラスファイルをバックアウトするよう記述し、コンパイラによって生成された Java クラスファイルを変換するために Javassist フレームワークを使用する方法を学習しました。この種のクラスファイルの変換処理は、永続的な変更を加えるという意味では優れていますが、アプリケーションを実行するたびに異なる変更を加えたい場合には、必ずしも有用ではありません。このような一時的な変更については、実際にアプリケーションの起動時に作動する方法が向いています。

JVM のアーキテクチャは、これをクラスローダインプリメンテーションを使用して行う便利な方法を提供してくれます。クラスローダのフックを使用すると、JVM にクラスをロードする処理をインターセプトし、それらが実際にロードされる前にクラス表現を変換することができます。これがどのように行われるのかを説明するために、最初にクラスロード処理を直接インターセプトする実例を示し、その後、どのように Javassist がアプリケーションで使用可能な便利で簡単な方法を提供するかを示します。この記事中では、このシリーズの以前の記事を引用します。

このシリーズのその他の記事

ロード処理の範囲

通常は、JVM へのパラメーターとしてメインクラスを指定することで Java アプリケーションを実行します。標準のオペレーションの場合はよいのですが、これでは多くのアプリケーションに役立つようにクラスのロード処理時にフックする手段がありません。第 1 回の「クラスとクラスのロード処理」で示したように、メインクラスの実行開始の前に多くのクラスがロードされます。これらのクラスロード処理をインターセプトするには、プログラムの実行を間接的に行うことが必要となります。

幸い、アプリケーションのメインクラスの実行の際に、JVM によって行われる処理をエミュレートすることは非常に簡単です。行うべき事は、指定されたクラスの static な `main()` メソッドを見つけ任意のコマンドライン引き数を渡して呼出すために、リフレクション(第 2 回で取り上げたような)を使用することです。リスト 1 は、これを行うサンプルコードを示します(コードを短くするためにインポートと例外処理は省きます)。

リスト 1. Java アプリケーション・ランナー

```
public class Run
{
    public static void main(String[] args) {
        if (args.length >= 1) {
            try {
                // load the target class to be run
                Class clas = Run.class.getClassLoader().
                    loadClass(args[0]);
                    // invoke "main" method of target class
                Class[] ptypes =
                    new Class[] { args.getClass() };
                Method main =
                    clas.getDeclaredMethod("main", ptypes);
                String[] pargs = new String[args.length-1];
                System.arraycopy(args, 1, pargs, 0, pargs.length);
                main.invoke(null, new Object[] { pargs });
                } catch ...
            }
            } else {
            System.out.println
                ("Usage: Run main-class args...");
        }
    }
}
```

このクラスを使用して Java アプリケーションを実行するには、`java` コマンド (アプリケーションのメインクラスとアプリケーションに渡したい任意の引数で続く) の対象として指定する必要があります。つまり、通常 Java アプリケーションを起動するために使用するコマンドなら

```
java test.Test arg1 arg2 arg3
```

代りに、そのコマンドと共に `Run` クラスを使用してアプリケーションを起動します。

```
java Run test.Test arg1 arg2 arg3
```

クラスロードをインターセプト

それのみについて言えば、リスト 1 の小さな `Run` クラスではあまり役に立ちません。クラスロード処理をインターセプトするという目的を達成するためには、アプリケーションのクラス用の独自のクラスローダを定義し使用することによって、もう一歩先に踏み込む必要があります。

第 1 回で説明したように、クラスローダはツリー形式の構造階層を使用しています。各クラスローダ (コア Java クラス用に JVM によって使用されるルートクラスローダを除く) は親のクラスローダを持ち、同じクラスが階層内の 1 つ以上のクラスローダによってロードされる場合の衝突の発生を防ぐために、クラスローダはそれら自身で、クラスをロードする前に親のクラスローダを利用してチェックします。この最初に親を利用してチェックするような処理をデリゲーション (委譲) といいます。クラスローダは、そのクラス情報にアクセスするルートに最も近いクラスローダに、クラスをロードする責務をデリゲートします。

リスト 1 の `Run` プログラムの実行を開始するときには既に、JVM のためのデフォルトのシステムクラスローダ (あなたが定義したクラスパスを除くもの) によってロードされています。クラスロード処理のデリゲーションの規則に従うために、システムクラスローダの代わりに私たち独自のクラスローダを使用する必要があり、全て同じクラスパス情報を使用し同じ親にデリゲートする必要があります。幸い、システムクラスローダインプリメンテーションのために現在の JVM によって使用される `java.net.URLClassLoader` クラスは、`getURLs()` メソッドを使用してクラスパス情報を検索する簡単な方法を提供します。クラスローダを記述するために、`java.net.URLClassLoader` をサブクラス化することができ、またメインクラスをロードするシステムクラスローダとして同じクラスパスおよび親クラスローダを使用するために、基底クラスを初期化します。リスト 2 は、この手法の実際のインプリメンテーションを示します。

リスト 2. 冗長なクラスローダ

```
public class VerboseLoader extends URLClassLoader
{
    protected VerboseLoader(URL[] urls, ClassLoader parent) {
        super(urls, parent);
    }
    public Class loadClass(String name)
        throws ClassNotFoundException {
        System.out.println("loadClass: " + name);
        return super.loadClass(name);
    }
    protected Class findClass(String name)
        throws ClassNotFoundException {
        Class clas = super.findClass(name);
        System.out.println("findclass: loaded " + name +
            " from this loader");
        return clas;
    }
    public static void main(String[] args) {
        if (args.length >= 1) {
            try {
                // get paths to be used for loading
                ClassLoader base =
                    ClassLoader.getSystemClassLoader();
                URL[] urls;
                if (base instanceof URLClassLoader) {
                    urls = ((URLClassLoader)base).getURLs();
                } else {
                    urls = new URL[]
                        { new File(".").toURI().toURL() };
                }
```

```
                // list the paths actually being used
                System.out.println("Loading from paths:");
                for (int i = 0; i < urls.length; i++) {
                    System.out.println(" " + urls[i]);
                }
                // load target class using custom class loader
                VerboseLoader loader =
                    new VerboseLoader(urls, base.getParent());
                Class clas = loader.loadClass(args[0]);
                    // invoke "main" method of target class
                Class[] ptypes =
                    new Class[] { args.getClass() };
                Method main =
                    clas.getDeclaredMethod("main", ptypes);
                String[] pargs = new String[args.length-1];
                System.arraycopy(args, 1, pargs, 0, pargs.length);
                Thread.currentThread().
                    setContextClassLoader(loader);
                main.invoke(null, new Object[] { pargs });
                } catch ...
            }
            } else {
            System.out.println
                ("Usage: VerboseLoader main-class args...");
        }
    }
}
```

私たちは `java.net.URLClassLoader` をサブクラス化し `VerboseLoader` クラスを作成しました。この loader インスタンスによってロードされるクラスに注目し (親クラスローダのデリゲーションよりも)、`VerboseLoader` クラスはロードされているクラスをすべてリストします。ここでも再び、コードを簡潔にするためにインポートおよび例外処理を省いています。

`VerboseLoader` クラスの最初の 2 つのメソッド、`loadClass()` と `findClass()` は、標準のクラスローダメソッドをオーバーライドしています。`loadClass()` メソッドはクラスローダに要求された各クラスのために呼ばれます。ここでは、単にコンソールにメッセージを出力した後に、実際の処理のために基底クラスを呼出します。基底クラスのメソッドは標準のクラスローダのデリゲーションの振る舞いをインプリメントし、最初に親クラスローダが要求されたクラスをロードできるかどうかチェックし、親クラスローダが失敗した場合にのみ、protected の `findClass()` メソッドを直接使用してクラスのロードを試みます。`VerboseLoader` の `findClass()` のインプリメンテーションについては、最初にオーバーライドした基底クラスのインプリメンテーションを呼び、その後呼出しが成功した場合にメッセージを出力します(例外をスローせずに返します)。

`VerboseLoader` の `main()` メソッドは、収容クラスのために使われるローダーからクラスパスの URL リストを取得するか、`URLClassLoader` のインスタンスでないローダーと共に使用される場合に唯一のクラスパスエントリーとしてカレントディレクトリを使用するかのどちらかです。別の方法としては、実際に使用されているパスをリストし、次に `VerboseLoader` クラスのインスタンスを作成し、コマンドライン上で指定されたターゲットクラスをロードするために使用します。ロジックの残りの、ターゲットクラスの `main()` メソッドを見つけて呼出す部分は、リスト 1 の `Run` コードと同じです。

リスト 3 は、リスト 1 の `Run` アプリケーションを呼出す `VerboseLoader` コマンドラインと出力のサンプルを示します。

リスト 3. リスト 2 のプログラムのサンプル出力

```
[dennis]$ java VerboseLoader Run
Loading from paths:
 file:/home/dennis/writing/articles/devworks/dynamic/code5/
loadClass: Run
loadClass: java.lang.Object
findclass: loaded Run from this loader
loadClass: java.lang.Throwable
loadClass: java.lang.reflect.InvocationTargetException
loadClass: java.lang.IllegalAccessException
loadClass: java.lang.IllegalArgumentException
loadClass: java.lang.NoSuchMethodException
loadClass: java.lang.ClassNotFoundException
loadClass: java.lang.NoClassDefFoundError
loadClass: java.lang.Class
loadClass: java.lang.String
loadClass: java.lang.System
loadClass: java.io.PrintStream
Usage: Run main-class args...
```

この場合、`VerboseLoader` によって直接ロードされるクラスは `Run` クラスのみです。`Run` クラスによって使用される他のすべてのクラスは Java クラスのコアであるのです (それらは親のクラスローダを介してデリゲーションによってロードされます)。すべてではありませんが、これらのほとんどのコア Java クラスは、実際には `VerboseLoader` アプリケーション自体の起動時にロードされています。したがって、親のクラスローダは、以前に作成された `java.lang.Class` インスタンスへの参照を返すだけとなります。

Javassist のインターセプト

リスト 2 の `VerboseClassloader` は、クラスロード処理のインターセプトの基本を示します。クラスをロード時に修正するために、リソースとなるバイナリクラスファイルにアクセスする `findClass()` メソッドにコードを追加し、その後にバイナリデータを操作することができます。Javassist は、この種のインターセプションを直接行うためのコードを実際に含んでいます。したがって、このサンプルをさらに取り上げる代わりに Javassist インプリメンテーションを使用するメソッドを見てゆきたいと思います。

Javassist でのクラスロード処理のインターセプトは、第 4 回で示した `javassist.ClassPool` クラスに基づいています。その記事では、`javassist.CtClass` インスタンスの形でクラスの Javassist 表現を戻し、`ClassPool` から直接クラスを名前で指定して要求しました。しかしながら、`ClassPool` を使用する方法はそれだけではありません。Javassist は、さらにクラスデータのソースとして `ClassPool` を使用するクラスローダを `javassist.Loader` クラスの形で提供します。

クラスをロード時に操作するために、`ClassPool` は Observer パターンを使用します。`ClassPool` のコンストラクタへ、期待される observer インターフェースのインスタンスである `javassist.Translator` を渡すことができます。新しいクラスが `ClassPool` から要求されるたびに、それは observer の `onWrite()` メソッドを呼出します (それは、`ClassPool` によって行われる前に、クラス表現を修正することができます)。

`javassist.Loader` クラスは便利な `run()` メソッドを持ちます。この `run()` メソッドは、ターゲットクラスをロードし、引数として配列を与えてそのクラスの `main()` メソッドを呼出します (リスト 1 のコードのように)。リスト 4 では、Javassist のクラスとこのメソッドを使用して、ターゲットのアプリケーションのクラスをロードし実行します。この単純な `javassist.Translator`

observer のインプリメンテーションは、ここでは要求されたクラスに関するメッセージの出力のみを行います。

リスト 4. Javassist アプリケーション・ランナー

```
public class JavassistRun
{
    public static void main(String[] args) {
        if (args.length >= 1) {
            try {
                // set up class loader with translator
                Translator xlat = new VerboseTranslator();
                ClassPool pool = ClassPool.getDefault(xlat);
                Loader loader = new Loader(pool);
                    // invoke "main" method of target class
                String[] pargs = new String[args.length-1];
                System.arraycopy(args, 1, pargs, 0, pargs.length);
                loader.run(args[0], pargs);
                } catch ...
            }
            } else {
            System.out.println
                ("Usage: JavassistRun main-class args...");
        }
    }
    public static class VerboseTranslator implements Translator
    {
        public void start(ClassPool pool) {}
        public void onWrite(ClassPool pool, String cname) {
            System.out.println("onWrite called for " + cname);
        }
    }
}
```

以下は、リスト 1 の `Run` アプリケーションを呼出すために使用される、`JavassistRun` コマンドラインと出力のサンプルです。

```
[dennis]$java -cp .:javassist.jar JavassistRun Run
onWrite called for Run
Usage: Run main-class args...
```

実行時間の計測

第 4 回で考察したメソッドの実行時間計測の修正方法は、パフォーマンスの問題を切り離すために有用なツールとなりえますが、それは実際にはより柔軟なインターフェースを必要とします。その記事では、プログラムへのコマンドラインパラメーターとしてクラスとメソッド名を渡すだけでした。バイナリクラスファイルをロードし時間計測コードを追加した後に、クラスをバックアウトするよう書きました。本記事については、ロード時に修正する方法を使用し、クラスと時間を計測したいメソッドの指定においてパターンマッチングをサポートするようにコードを変更します。

クラスがロードされると同時に修正を行うようにコードを変更することは簡単です。リスト 4 の `javassist.Translator` コードを構築すると、書かれているクラス名がターゲットクラス名とマッチする場合にのみ、`onWrite()` から計測時間情報を追加するメソッドを呼出すことができます。リスト 5 はこれについて 示します (`addTiming ()` の詳細は省きます。これについては第 4 回を参照してください)。

リスト 5. ロード時に時間計測コードを追加する

```
public class TranslateTiming
{
    private static void addTiming(CtClass clas, String mname)
        throws NotFoundException, CannotCompileException {
        ...
    }
    public static void main(String[] args) {
        if (args.length >= 3) {
            try {
                // set up class loader with translator
                Translator xlat =
                    new SimpleTranslator(args[0], args[1]);
                ClassPool pool = ClassPool.getDefault(xlat);
                Loader loader = new Loader(pool);
                    // invoke "main" method of target class
                String[] pargs = new String[args.length-3];
                System.arraycopy(args, 3, pargs, 0, pargs.length);
                loader.run(args[2], pargs);
                } catch (Throwable ex) {
                ex.printStackTrace();
            }
            } else {
            System.out.println("Usage: TranslateTiming" +
                " class-name method-mname main-class args...");
        }
    }
    public static class SimpleTranslator implements Translator
    {
        private String m_className;
        private String m_methodName;
        public SimpleTranslator(String cname, String mname) {
            m_className = cname;
            m_methodName = mname;
        }
        public void start(ClassPool pool) {}
        public void onWrite(ClassPool pool, String cname)
            throws NotFoundException, CannotCompileException {
            if (cname.equals(m_className)) {
                CtClass clas = pool.get(cname);
                addTiming(clas, m_methodName);
            }
        }
    }
}
```

パターンメソッド

ロード時にメソッド時間計測コードを動作させるほかに、リスト 5 に示されるように、時間計測メソッドの指定において柔軟性をもたせると良いと思います。最初は Java 1.4 `java.util.regex` パッケージでサポートされている正規表現を使用して柔軟性をインプリメントしましたが、その後それでは必要な柔軟性をもたないことがわかりました。問題は、修正するクラスとメソッドを選択するために私が使用したかったパターンが、正規表現モデルにうまく適合しないということでした。

それでは、どのようなパターンがクラスとメソッドを選択するために必要なのでしょうか？必要だったのは、実際のクラスとメソッド名、リターンタイプ、およびコールパラメータータイプを含む、パターンにおけるクラスとメソッドの任意の複数の特性を使用する性能でした。その他方では、実際に名前や型の比較にはそれほど柔軟性は必要ではありませんでした。私が必要だった

比較形式のほとんどは、単純な等しい文字列同士の比較であり、比較に基本的なワイルドカードを使用することで残りの文字を表しました。これに対応する最も簡単な方法は、少々の拡張機能を使用して、標準の Java メソッドの宣言のようなパターンを作成することでした。

この方法のサンプルとして、以下に、`test.StringBuilder` クラスの `buildString(int)` メソッドとマッチするいくつかのパターンを示します。

```
java.lang.String test.StringBuilder.buildString(int)
test.StringBuilder.buildString(int)
*buildString(int)
*buildString
```

これらのパターンの一般的なものは、まずオプションのリターンタイプ(正確なテキストのもの)、次にクラスとメソッド名を組み合わせたパターン (ワイルドカード文字"*"を使用したもの)、そして最後に、パラメータータイプのリスト(正確なテキストのもの)です。リターンタイプが存在する場合は、スペースによって、マッチするメソッド名と分けられている必要があります(パラメーターのリストはメソッド名のマッチに続いていますが)。パラメーターを柔軟にマッチさせるために、2 種類の動作を取り入れることにしました。パラメーターが小カッコで囲まれたリストとして与えられる場合、それらは正確にメソッドのパラメーターと一致する必要があります。代りに、大カッコ (「[]」) によって囲まれている場合は、リストされた型はマッチするメソッドのパラメーターとしてすべて存在している必要があります。しかし、メソッドは任意の順でそれらを使用する可能性があり、さらに追加のパラメーターを使用する可能性もあります。したがって、`*buildString(int, java.lang.String)` は「buildString」で終わる名前の、`int`、`String` の順で正確に 2 つのパラメーターをとる任意のメソッドとマッチします。`*buildString[int,java.lang.String]` は同じ名前のメソッドとマッチしますが、どれか 1 つが `int` で他のどれかが `java.lang.String` の 2 つ以上のパラメーターをとります。

リスト 6 は、これらのパターンを扱う為に記述した `javassist.Translator` サブクラスを簡略化したものです。実際のマッチングコードはこの記事に関連するものではありませんが、それを見たいか、使用したい場合は、ダウンロードファイル (参考文献を参照) に含まれています。この `TimingTranslator` を使用するメインプログラムクラスは `BatchTiming` であり、これもまたダウンロードファイルに含まれています。

リスト 6. パターンマッチング変換プログラム

```
public class TimingTranslator implements Translator
{
    public TimingTranslator(String pattern) {
        // build matching structures for supplied pattern
        ...
    }
    private boolean matchType(CtMethod meth) {
        ...
    }
    private boolean matchParameters(CtMethod meth) {
        ...
    }
    private boolean matchName(CtMethod meth) {
        ...
    }
    private void addTiming(CtMethod meth) {
        ...
    }
```

```
    public void start(ClassPool pool) {}
    public void onWrite(ClassPool pool, String cname)
        throws NotFoundException, CannotCompileException {
        // loop through all methods declared in class
        CtClass clas = pool.get(cname);
        CtMethod[] meths = clas.getDeclaredMethods();
        for (int i = 0; i < meths.length; i++) {
            // check if method matches full pattern
            CtMethod meth = meths[i];
            if (matchType(meth) &&
                matchParameters(meth) && matchName(meth)) {
                // handle the actual timing modification
                addTiming(meth);
            }
        }
    }
}
```

次回予告

前回の 2 つの記事では、基本的な変換を行うための Javassist の使用方法を見ました。次回の記事では、bytecode を編集するために検索と交換を行う手法を提供する、このフレームワークの高度な機能について取り扱います。これらの機能は、すべてのメソッドコールやフィールドアクセスをインターセプトするといった変更を含むプログラムの振舞いの変更を、簡単に行います。それらは、なぜ Javassist が Java プログラムにおけるアスペクト指向のサポートのための優れたフレームワークなのかを理解する鍵となります。アプリケーションの様相を解き明かすために、どのように Javassist を使用できるのかを知るため、次回もご覧になってください。

著者について

Dennis Sosnoski

Dennis Sosnoski はシアトル地域にある Java 技術のコンサルティング会社、Sosnoski Software Solutions, Inc. の創立者で、主席コンサルタントでもあり、また XML や Web サービスに関するトレーニングやコンサルティングの専門家でもあります。彼のプロとしてのソフトウェア開発経験は 30年以上に渡り、ここ数年はサーバー側の XML 技術や Java 技術に注力しています。Dennis は、全米各地で行われる会議で頻繁に講演を行っています。また、Java クラスワーキング技術を基に構築された、オープンソースの JiBX XML Data Binding フレームワークの中心開発者でもあります。