Java プログラミングのダイナミックス: 第 4 回 Javassist でのクラス変換

バイトコード内のメソッドを変換するために Javassist を使用する

Dennis Sosnoski
President
Sosnoski Software Solutions, Inc.

2003年 9月 16日

書かれたソースコードをそのとおり実行する Java クラスは退屈ではありませんか？それなら喜んでください。なぜならあなたはもうすぐ、コンパイラによってクラスを思いもよらない形に変形する知識を得るからです。この記事では、Java コンサルタントの Dennis Sosnoski が、Javassist (幅広く利用されている JBoss アプリケーション・サーバーに付属の、アスペクト指向プログラミング機能の基本となるバイトコード操作ライブラリー) に注目し、彼の「Java プログラミングのダイナミックス 」シリーズを本格的に取り扱います。あなたは Javassist で既存クラスの変換の基本知識を取得し、クラス操作に取り組むこのフレームワークのソースコードの威力と制限の両方を知ることでしょう。

このシリーズの他の記事を見る

Java クラスフォーマットの基本とリフレクションを通したランタイムアクセスについて取り扱った後、さらに高度なトピックへとこのシリーズを移しましょう。今月はこのシリーズの第 2 部に取り掛かります。ここでは Java クラス情報は、アプリケーションに操作される、データ構造の単なる別の形式となります。私は、このトピックの全範囲をクラス操作 と呼ぶことにします。

Javassist バイトコード操作ライブラリーを取り上げたクラス操作を始めましょう。Javassist は、バイトコードで動作する単なるライブラリーではなく、クラス操作の実験の大きな出発点となる特別な働きをします。あなたは、実際にバイトコードや Java 仮想マシン (JVM) アーキテクチャについて何も学ぶ必要なく、Javassist を Java クラスのバイトコードの変更に使用することができます。これはいくつかの点でいいことばかりではありません。(私は通常、相手がよく理解していない技術を安易に使用することは勧めません。) しかしそれは確かに、個別の命令のレベルでバイトコード操作をするフレームワークによるものよりはるかに便利です。

Javassist の基本

Javassist は、Java バイナリクラスを検査、編集、作成させます。検査の特徴は主にリフレクション API を通して Java で直接利用可能なものを複製しますが、実際にクラスを、単に実行するのではなく修正している場合、この情報にアクセスする代替方法を持っており有用です。これは、JVM

の設計は、JVM にロードされた後の生のクラスデータへのアクセス方法を何も用意していないからです。もしデータとしてクラスを取り扱うつもりなら、JVM の外部で行う必要があります。

このシリーズのその他の記事

Javassist は、操作しているクラスを検索し制御する `javassist.ClassPool` クラスを使用します。このクラスは、JVM クラスローダーによく似た動作をします。しかし、重要な違いとして、あなたのアプリケーションの一部として実行するためにロードされたクラスをリンクするのではなく、クラス・プールは Javassist API を通して、ロードしたクラスをデータとして使用可能にします。JVM サーチパスからロードするデフォルト・クラス・プールを使用することができ、またはご自分のパスリストを探索するか定義することができます。バイト配列あるいはストリームからバイナリクラスを直接ロードすることも、スクラッチから新しいクラスを作成することもできます。

クラス・プールの中でロードされたクラスは、`javassist.CtClass` インスタンスによって表わされます。Java 標準の `java.lang.Class` クラスのように、`CtClass` は、フィールドやメソッドのようなクラスデータを検査するためのメソッドを提供します。しかしながら、それは `CtClass` にとって単なるスタートであり、それはまた、クラスに新しいフィールド、メソッド、およびコンストラクタを追加するメソッド、ならびにクラス名、スーパークラスおよびインターフェースを変更するメソッドを定義します。おかしなことに、Javassist は、クラスからフィールド、メソッドあるいはコンストラクタを削除する方法はなにも提供しません。

フィールド、メソッドおよびコンストラクタは `javassist.CtField`、`javassist.CtMethod`、および `javassist.CtConstructor` インスタンスによってそれぞれ表わされます。これらのクラスは、クラスによって表わされるすべてのアイテムの様相を修正するメソッドを定義し、メソッドまたはコンストラクタの実際のバイトコード本文を含みます。

全てのバイトコードのソース

Javassist は、メソッドかコンストラクタのバイトコード本文を完全に交換させるか、あるいは選択的に、既存の本文 (コンストラクタ用に 2、3 の他の変更を加えたもの) の最初か最後にバイトコードを加えます。どちらにしても、新しいバイトコードは Java ライクなソース・コード・ステートメントあるいは文字列の中のブロックとして渡されます。Javassist のメソッドは、Java バイトコードの中へ提供するソースコード (それらはその後、対象となるメソッドかコンストラクタの本文に挿入する) を効率的にコンパイルします。

Javassist によって受け入れられたソースコードは、Java 言語に正確にマッチするものではありません。しかし主な違いは、挿入したコードの中で使用したいメソッドまたはコンストラクタのパラメーター、メソッドの返り値、および他のアイテムを表わすために使用される、いくつかの特殊な識別名の追加くらいです。これらの特殊な識別名はすべて `$` 記号で始まります。したがって、それらはコード中であなたが行なう以外では、どんな操作にも影響しないでしょう。

また、Javassist へ渡すソースコードの中でいくつかの制限があります。第 1 の制限は、フォーマットそのものです。(それは単一のステートメントかブロックでなくてはなりません) ブロック中にステートメントの任意のシーケンスを配置することができるので、これはほとんどの目的にとってたいした制限ではありません。ここに、どのように動作するか示すために、特殊な Javassist の識別名を最初の 2つのメソッドのパラメーターの値として使用した例があります。

```
{
  System.out.println("Argument 1: " + $1);
  System.out.println("Argument 2: " + $2);
}
```

ソースコードでのより本質的な制限は、追加されているステートメントまたはブロックの外部で宣言されたローカル変数を参照する方法がないということです。これは、例えば、メソッドの最初と最後の両方にコードを加えた場合、通常は最初に加えたコードから最後に加えたコードへ情報を渡すことができないということを意味します。この制限の逃げ道はありますが、回避方法は面倒なものです。(通常は、単一のブロックへ挿入する個別のコードをひとつに統合する方法を見つけなければなりません。)

Javassist のクラス操作

Javassist の適用例については、よく私がソースコード中で直接扱う操作方法を使用しましょう。メソッドの実行にかかる時間の測定です。ソースの中でこれを行うことはとても簡単です。単にメソッドの最初で現在の時間を記録し、次にメソッドの最後で再び現在の時間をチェックし、2つの値の差を計算します。あなたにソースコードがなければ、通常はこの種の時間の情報を得ることは非常に困難です。そこでクラス操作が役に立ちます--それはあなたに、ソースコードを必要とせずに任意のメソッドにこのような変更を加えさせます。

リスト 1 は、時間計測実験のためにモルモットとして使用する (悪い) サンプルメソッドを示します。`StringBuilder` クラスの `buildString` メソッドです。このメソッドは、どんな Java のパフォーマンスの達人も行うべきでない という方法で、任意の要求された長さの文字列を構築しています。--それは、より長い文字列を作成するために文字列の末尾へ繰り返し 1文字を付け足します。文字列が不変なので、この方法は、古い文字列および末尾に加えられた 1文字からコピーされたデータで、ループによって時間毎に新しい文字列が構築されることを意味します。この最終結果として、このメソッドがより長い文字列を作成するために使用されるにつれて、ますますオーバーヘッドに陥るでしょう。

リスト１．時間を計るメソッド

```
public class StringBuilder
{
    private String buildString(int length) {
        String result = "";
        for (int i = 0; i < length; i++) {
            result += (char)(i%26 + 'a');
        }
        return result;
    }

    public static void main(String[] argv) {
        StringBuilder inst = new StringBuilder();
        for (int i = 0; i < argv.length; i++) {
            String result = inst.buildString(Integer.parseInt(argv[i]));
            System.out.println("Constructed string of length " +
                result.length());
        }
    }
}
```

メソッドに時間計測機能を追加する

ソースコードをこのメソッドのために利用できるようにしてあるので、どのように直接、計測時間情報を追加するのか示しましょう。これは、私が Javassist を使用して行いたいことのモデルとしても役立つでしょう。リスト 2 は、時間計測機能を追加されただけの `buildString()` メソッドを示します。これは大した変更ではありません。追加コードは、単にローカル変数に開始時間を記録し、その後メソッドの最後に経過時間を算出し、それをコンソールに出力します。

リスト 2. 時間計測機能を備えたメソッド

```
    private String buildString(int length) {
        long start = System.currentTimeMillis();
        String result = "";
        for (int i = 0; i < length; i++) {
            result += (char)(i%26 + 'a');
        }
        System.out.println("Call to buildString took " +
            (System.currentTimeMillis()-start) + " ms.");
        return result;
    }
```

Javassist でやってみましょう

クラスバイトコードを操作するために Javassist を使用し、同じ結果を得ることは簡単であるように思えます。Javassist は、メソッドの最初と最後にコードを加える方法を提供しています。要するに、それは私がメソッドに計測時間情報を追加するために、ソースコード中で行ったことと同じです。

しかし、そこには障害があります。Javassist がどのようにあなたにコードを追加させるのか説明した時に、私は、追加コードがメソッドの他の箇所に定義されたローカル変数を参照することができないと言いました。この制限は、私がソースコード中で行なったのと同じ方法で、Javassist の中のタイミングコードのインプリメントを妨げます。その事例では、私が最初に追加したコードに新しいローカル変数を定義し、最後に追加した変数に参照を付けました。

それでは、同じ結果を得るために他の手段を使用することができるでしょうか？そうです、私はクラスに新しいメンバー・フィールドを加え、ローカル変数の代わりにそれを使用することがで

きました。しかしながら、それはくせのある解決策で、一般的な使用のためのいくつかの制限に苦しむことになります。例えば、再帰的なメソッドに何が起こるか考えてください。メソッドがそれ自体を呼ぶたびに、最後の呼び出しで記録された開始時間の値は上書きされ失われるでしょう。

幸運なことに、きれいな解決策があります。オリジナルのメソッドのコードを不変にしておきメソッド名だけを変更し、次に、オリジナル名を使用した新しいメソッドを追加することができます。このインターセプタ・メソッドは、同じ値を返すオリジナルのメソッドとして同じシグネチャーを使用することができます。リスト 3 は、この方法でソースコードを変更するとどのようになるかを示します。

リスト 3. ソースにインターセプタ・メソッドを加える

```
    private String buildString$impl(int length) {
        String result = "";
        for (int i = 0; i < length; i++) {
            result += (char)(i%26 + 'a');
        }
        return result;
    }
    private String buildString(int length) {
        long start = System.currentTimeMillis();
        String result = buildString$impl(length);
        System.out.println("Call to buildString took " +
            (System.currentTimeMillis()-start) + " ms.");
        return result;
    }
```

このインターセプタ・メソッドを使用する方法は、Javassist でうまく動作します。メソッド全体が単一のブロックなので、本文の中に何の問題もなくローカル変数を定義し使用することができます。また、インターセプションメソッドのためのソースコードの生成も簡単です--それは、任意の実行可能なメソッドのために動作する少しの代用品を必要とするだけです。

インターセプションの実行

メソッドに時間計測機能を追加するためのコードの実装は、Javassist basics に記述された Javassist API のうちのいくつかを使用します。リスト 4 はこのコードを、時間を計られるクラス名とメソッド名を与える、1 対のコマンドライン引数をとるアプリケーションの形式で示します。`main()` メソッドの本文はクラス情報を見つけるだけで、その後、実際に修正を行なう `addTiming()` メソッドにそれを渡します。`addTiming()` メソッドは、まず既存のメソッドを、名前の最後に「`$impl`」と付け加えてリネームし、次に、オリジナルの名前を使用してメソッドのコピーを作成します。その後、コピーされたメソッドの本文を、リネームされたオリジナル・メソッドの呼び出しをラッピングするタイミングコードに置き換えます。

リスト 4. Javassist にインターセプタ・メソッドを追加する

```
public class JassistTiming
{
    public static void main(String[] argv) {
        if (argv.length == 2) {
            try {

                // start by getting the class file and method
                CtClass clas = ClassPool.getDefault().get(argv[0]);
                if (clas == null) {
                    System.err.println("Class " + argv[0] + " not found");
```

```
            } else {

                // add timing interceptor to the class
                addTiming(clas, argv[1]);
                clas.writeFile();
                System.out.println("Added timing to method " +
                    argv[0] + "." + argv[1]);

            }

        } catch (CannotCompileException ex) {
            ex.printStackTrace();
        } catch (NotFoundException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        }

    } else {
        System.out.println("Usage: JassistTiming class method-name");
    }
}

private static void addTiming(CtClass clas, String mname)
    throws NotFoundException, CannotCompileException {

    //  get the method information (throws exception if method with
    //  given name is not declared directly by this class, returns
    //  arbitrary choice if more than one with the given name)
    CtMethod mold = clas.getDeclaredMethod(mname);

    //  rename old method to synthetic name, then duplicate the
    //  method with original name for use as interceptor
    String nname = mname+"$impl";
    mold.setName(nname);
    CtMethod mnew = CtNewMethod.copy(mold, mname, clas, null);

    //  start the body text generation by saving the start time
    //  to a local variable, then call the timed method; the
    //  actual code generated needs to depend on whether the
    //  timed method returns a value
    String type = mold.getReturnType().getName();
    StringBuffer body = new StringBuffer();
    body.append("{\nlong start = System.currentTimeMillis();\n");
    if (!"void".equals(type)) {
        body.append(type + " result = ");
    }
    body.append(nname + "($$);\n");

    //  finish body text generation with call to print the timing
    //  information, and return saved value (if not void)
    body.append("System.out.println(\"Call to method " + mname +
        " took \" +\n (System.currentTimeMillis()-start) + " +
        "\" ms.\");\n");
    if (!"void".equals(type)) {
        body.append("return result;\n");
    }
    body.append("}");

    //  replace the body of the interceptor method with generated
    //  code block and add it to class
    mnew.setBody(body.toString());
    clas.addMethod(mnew);

    //  print the generated code block just to show what was done
    System.out.println("Interceptor method body:");
    System.out.println(body.toString());
```

```
    }
}
```

インターセプタ・メソッドの本文の構築には、本文のテキストを蓄積するために `java.lang.StringBuffer` を使用します。(`StringBuilder` の中で使用している方法に対立するものとして、文字列の構築を行なう適切な方法を示します。) これは、オリジナル・メソッドが値を返すかどうかによって変わります。値を返す場合、インターセプタ・メソッドの最後に返すように、構築されたコードはローカル変数にその値を記録します。オリジナル・メソッドが無効な形式の場合、何も記録せず、またインターセプタ・メソッドからは何も返されません。

実際の本文のテキストは、(リネームされた) オリジナル・メソッドを呼び出すためを除いては、まるで標準の Java コードのように見えます。これは、コードの `body.append(nname + "($$); \n");` の行についてですが、`nname` はオリジナルのメソッドの変更後の名前です。この呼び出しの中で使用される `$$` 記号は、Javassist が、構築中のメソッドにパラメータのリストを示す様式です。この記号をオリジナルメソッドの呼び出しの中で使用することにより、インターセプタ・メソッドの呼び出しで提供された全ての引数が、オリジナルのメソッドに渡されます。

リスト 5 は、最初に未修正の形で `StringBuilder` プログラムを実行した結果を示し、その後に、計測時間情報を追加するために `JassistTiming` プログラムを実行し、それが修正された後にいよいよ、`StringBuilder` プログラムを実行します。 あなたは、修正が実行時間を記録した後にどのように `StringBuilder` が動作するか、またどのくらい時間が、非効率的な文字列構築コードのせいで作られた文字列の長さよりずっと速く増加するか知ることができます。

リスト 5. プログラムの実行

```
[dennis]$ java StringBuilder 1000 2000 4000 8000 16000
Constructed string of length 1000
Constructed string of length 2000
Constructed string of length 4000
Constructed string of length 8000
Constructed string of length 16000

[dennis]$ java -cp javassist.jar:. JassistTiming StringBuilder buildString
Interceptor method body:
{
long start = System.currentTimeMillis();
java.lang.String result = buildString$impl($$);
System.out.println("Call to method buildString took " +
 (System.currentTimeMillis()-start) + " ms.");
return result;
}
Added timing to method StringBuilder.buildString

[dennis]$ java StringBuilder 1000 2000 4000 8000 16000
Call to method buildString took 37 ms.
Constructed string of length 1000
Call to method buildString took 59 ms.
Constructed string of length 2000
Call to method buildString took 181 ms.
Constructed string of length 4000
Call to method buildString took 863 ms.
Constructed string of length 8000
Call to method buildString took 4154 ms.
Constructed string of length 16000
```

ソースを信じますか？

Javassist は、あなたにバイトコードの命令リストそのものではなく、ソースコード上で作業させることにより、クラス操作を容易にする偉大な仕事を行ないます。しかし、この使いやすさにはいくつかの不都合が付属しています。前記の「全てのバイトコードのソース」の中で述べたように、Javassist によって使用されるソースコードは正確な Java 言語ではありません。コードの中の特殊な記号の認識に加えて、Javassist はコンパイル時のコードのチェックを、Java 言語仕様によって要求されているよりも、ルーズに実装します。このため、あなたが注意しなければ、それは予期しない結果をもたらすような方法でソースからバイトコードを生成するでしょう。

例として、リスト 6 は、インターセプタ・コードの中でメソッドの開始時間用に使用されるローカル変数の型を `long` から `int` に変更するとき、何が起こるか示します。Javassist はソースコードを受け取りそれを有効なバイトコードに変換しますが、結果として得られる時間はごみです。Java プログラムで直接この代入をコンパイルした場合、それは Java 言語の規則の一つを破るため、コンパイルエラーとなります。縮小代入はキャストを要求します。

リスト 6. int に long を格納する

```
[dennis]$ java -cp javassist.jar:. JassistTiming StringBuilder buildString
Interceptor method body:
{
int start = System.currentTimeMillis();
java.lang.String result = buildString$impl($$);
System.out.println("Call to method buildString took " +
 (System.currentTimeMillis()-start) + " ms.");
return result;
}
Added timing to method StringBuilder.buildString
[dennis]$ java StringBuilder 1000 2000 4000 8000 16000
Call to method buildString took 1060856922184 ms.
Constructed string of length 1000
Call to method buildString took 1060856922172 ms.
Constructed string of length 2000
Call to method buildString took 1060856922382 ms.
Constructed string of length 4000
Call to method buildString took 1060856922809 ms.
Constructed string of length 8000
Call to method buildString took 1060856926253 ms.
Constructed string of length 16000
```

ソースコードの中で何を行なうかによっては、Javassist に無効なバイトコードを生成させてしまいます。リスト 7 は、この例を示します。ここでは、`int` の値を返すものとして計時メソッドを常に扱うために `JassistTiming` コードをパッチしました。Javassist は再び不具合のないソースコードを受け取りますが、結果として得られるバイトコードは、それを実行しようとしたとき検証に失敗します。

リスト 7. int に文字列を格納する

```
[dennis]$ java -cp javassist.jar:. JassistTiming StringBuilder buildString
Interceptor method body:
{
long start = System.currentTimeMillis();
int result = buildString$impl($$);
System.out.println("Call to method buildString took " +
 (System.currentTimeMillis()-start) + " ms.");
return result;
}
Added timing to method StringBuilder.buildString
[dennis]$ java StringBuilder 1000 2000 4000 8000 16000
Exception in thread "main" java.lang.VerifyError:
 (class: StringBuilder, method: buildString signature:
 (I)Ljava/lang/String;) Expecting to find integer on stack
```

Javassist に与えるソースコードに注意している限り、この種の問題はたいした問題ではありません。しかしながら、Javassist が必ずしもコード内のエラーを発見するわけではないということや、エラーの結果を予測することは難しいということを自覚することは大切です。

次回予告

この記事で取り扱った以上の Javassist の情報があります。次回は、Javassist がクラスのバルク・モディフィケーションと、実行時にロードされるクラスとしてのオンザフライ・モディフィケーションのために提供するいくつかの特別なフックに焦点を当て、もう少し深く研究しましょう。これらの機能は、Javassist をあなたのアプリケーションに様相をインプリメントするための偉大なツールにします。この強力なツールの詳細を知るために必ず続編を読みましょう。

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。