Javaコードを守る方法 (あるいは他人のJavaコードを参照する方法)

Javaコードの逆コンパイルおよび曖昧化の完全ガイド

Greg Travis (mito@panix.com)
Freelance Java programmer

2001年 5月 01日

Web上のオープン・ソース・ライブラリーからコードにパッチを当てる場合や、一般的なオペレーティング・システムのルーチンへの呼び出しを行う場合など、自分で書いたソースではなく、ソース自体入手できないような得体の知れないコードの一部を使わざるを得ないことがあります。このようなコードをデバッグする際には、優れたJava逆コンパイラーとコンパイラーの正しい使用方法を知っている必要があります。逆に、自分で書いたコードを詮索好きな他人の目から守る方法や、それを実現するためのコードの曖昧化についても同時に知っている必要があります。 当記事は、Javaコードをオープンにしたりロックしたりするための初心者向けガイドですが、作者であるGreg Travisは、Mocha、HoseMocha、jmangle、およびJODEなどの人気の高いツールを例に取って、Javaコードの逆アセンブル、逆コンパイル、曖昧化について説明しています。

ソース・コードを自分が持っていないがために、解決することができないバグに遭遇することほどやり切れない思いをすることはありません。こうしたことがきっかけでJava逆コンパイラーが誕生しました。逆コンパイラーは、コンパイル済みのバイトコードをソース・コードに戻します。コードの逆コンパイルは、Java言語特有の概念ではないのですが、Java開発者以外では、広く認識されていませんでした。

逆コンパイルと表裏一体の概念が曖昧化です。逆コンパイラーによってコンパイル済みコードからソース・コードを簡単に取り出せるようになると、作成したコードや、コード内の機密情報の保護が難しくなります。Java逆コンパイラーが一般的に使用されるようになるにつれて、Java曖昧化ツールも一般に使用されるようになりました。曖昧化ツールは、コードを守る役目を果たします。逆コンパイルも曖昧化も、商用の開発サークルで論争を巻き起こしました。そうした論争の多くは、Java言語を巡るものです。

当記事では、コードの逆コンパイルと曖昧化のプロセスについてご紹介します。まずこれらの技法の背後にある理論的なアイデアを説明して、その後、商用プログラミングの世界で巻き起こった論争について、手短にお話します。また、代表的な (商用とオープン・ソースの両方の)逆コンパイラーと曖昧化ツールについても例を挙げていくつかご紹介します。

逆コンパイルとは?

逆コンパイルは、オブジェクト・コードをソース・コードに変換するプロセスです。コンパイルは、ソース・コードをオブジェクト・コードに変換するプロセスですから、その逆を行う逆コンパイルも理にかなっていると言えます。しかし、オブジェクト・コードとはいったい何でしょう? 簡単に言うと、オブジェクト・コードとは、実マシンまたは仮想マシンで直接実行される言語で表されたコードのことです。C言語などの場合、オブジェクト・コードは一般にハードウェアのCPUで実行されますが、Javaのオブジェクト・コードは一般に仮想マシンで実行されます。

逆コンパイルの複雑性

上述したように、逆コンパイルは一見単純そうですが、実際には非常に複雑です。と申しますのは、小規模で低レベルな振る舞いから大規模で高レベルな振る舞いを推論する必要があるからです。これを直観的に理解するために、コンピューター・プログラムを複雑な会社の構造として考えてみましょう。上層管理職は、部下に対して、「テクノロジー・スループットの最大化」などの形で目標を設定します。そして、部下たちは目標を達成するために、新規XMLデータベースのインストールなど、より具体的な行動を起こします。

読者がこの会社の新入社員であると仮定します。そして、部下の1人に、その人が何をしているかを尋ねたときに、「新しいXMLデータベースをインストールしています。」という答えが返ってきたとします。しかし、この答えからは、最終目標がテクノロジー・スループットの最大化であることを推測することはできません。結局、得られた断片的な情報を元にして、究極的な目標は、サプライ・チェーン構造の分別や消費者データの集約など、まったく別のものだと認識してしまう可能性があります。

しかし、好奇心旺盛な新入社員である場合には、さらにいくつかの質問を行ったり、社内のさまざまなレベルの人々に尋ねて回ったりすることが考えられます。そしていつかは、それらすべての答えを組み合わせることで、会社のより大きな目標がテクノロジー・スループットの最大化であると推測できるかもしれません。

Cifuentesが逆コンパイルのプロセスをどのように説明しているのかを見てみましょう。

> バイナリー・リエンジニアリング・プロジェクトはすべて、バイナリー・ファイルに保管されたコードの逆アセンブルを必要とする。理論的な視点からは、フォン・ノイマン型のマシンでデータとコードを分離することは、停止性問題と同等であり、完全な静的変換は不可能である。ただし、実際には、さまざまな技法を使用して静的に変換するコードの割合を高めたり、実行時に動的変換技法へのフックを利用したりすることが可能である。
> --"Binary Reengineering of Distributed Object Technology" (参考文献を参照)

オブジェクト・コードをソース・コードに変換すること以外にも、逆コンパイルを行う意義があります。Javaクラス・ファイルには、さまざまな種類の多くの情報が含まれている可能性があります。クラス・ファイルにどのような種類の情報が含まれているのかを知ることは、そうした情報を使用したり、そうした情報に対して、どのようなことができるのかを理解する上で重要です。Java逆アセンブラーによって、このような情報も知ることができるのです。

クラス・ファイルの逆アセンブル

Javaクラス・ファイルのバイナリー形式のコードは重要ではありません。重要なのは、データに含まれている情報を知ることです。そのために、多くのJDKに同梱されているツールjavapを使用することにします。javapはJavaコード逆アセンブラーであり、逆コンパイラーとは異なります。 逆アセンブラーは、(リスト1に示すような)機械が読める形式のオブジェクト・コードを、(リスト2に示すような) 人間が読める形式に変換します。

リスト1. 未加工のクラス・ファイルの内容

```
0000000 feca beba 0300 2d00 4200 0008 081f 3400
0000020 0008 073f 2c00 0007 0735 3600 0007 0737
0000040 3800 0007 0a39 0400 1500 000a 0007 0a15
0000060 0800 1600 000a 0008 0a17 0800 1800 0009
...
```

リスト2. javapの出力

```
Local variables for method void priv(int)
      Foo this  pc=0, length=35, slot=0
      int argument  pc=0, length=35, slot=1

Method void main(java.lang.String[])
      0 new #4
      3 invokespecial #10
      6 return
```

注意していただきたい点は、リスト2の出力がソース・コードと同じではないことに注意してください。リストの前半部分は、メソッドのローカル変数のリストで、後半部分はアセンブリー・コード、すなわち人間が読むことのできるオブジェクト・コードになっています。

クラス・ファイル中の要素

javapはクラス・ファイルを逆アセンブルしたり、アンパックするためなどに使用されます。ここで、Javaクラス・ファイルに含まれる情報を簡単に要約しておきます。この情報はjavapで逆アセンブルすることができます。

- **メンバー変数** : 各クラス・ファイルには、そのクラスの各データ・メンバーに関するすべての命名情報とタイプ情報が含まれます。
- **逆アセンブルされたメソッド** : クラスの各メソッドは、タイプ・シグネチャーとともに、仮想マシン命令列によって表現されます。
- **行番号** : メソッドの各セクションは、可能な限り、生成される元になったソース・コード上の行に対応付けされます。これによって、ランタイム・システムとデバッガーは、実行中のプログラムに関するスタック・トレースを行うことができるようになります。
- **ローカル変数名** : メソッドに対してローカルな変数は、メソッドがコンパイルされた後では、実際に名前を必要とすることはありませんが、javacコンパイラーの `-g` オプションを使用すると名前を含めることができます。これらの名前も、ランタイム・システムやデバッガーの情報として役立ちます。

Javaクラス・ファイルの内容が、少しご理解していただけたところで、これらの情報を各々の目的に合わせてどのように調整できるのか検討してみましょう。

逆コンパイラーの使用

概念的には、逆コンパイラーは、簡単に使用できます。逆コンパイラーはコンパイラーと逆のことをします。つまり、.classファイルを指定すると、ソース・コード・ファイルを返してくれます。

最近では、複雑なグラフィカル・インターフェースを備えた逆コンパイラーもありますが、まずここではMochaを使用することにします。Mochaは、一般に使用されるようになった最初の逆コンパイラーです。この記事の最後に、GPLのもとで利用可能なより新しい逆コンパイラーであるJODEについて説明します。(Mochaのダウンロードおよび Java逆コンパイラーのリストについては、 参考文献を参照してください。)

ディレクトリーにFoo.classというクラス・ファイルが入っていると仮定します。このクラス・ファイルをMochaで逆コンパイルするには、次のコマンドを入力するだけで簡単に行えます。

```
$ java mocha.Decompiler Foo.class
```

これによってFoo.mochaという新しいファイルが作られます (Mochaは、オリジナル・ファイルのソース・コードが上書きされるのを避けるために、Foo.mochaという名前を作成します)。 この新しいファイルはJavaソース・ファイルです。すべてがうまく行っているとすると、正常にコンパイルすることができます。このファイルをFoo.javaという名前に変更して、先に進みます。

しかし、落とし穴があります。コードに対してMochaを実行しても、生成されたコードがソースと同一にならないということです。以下に例を示します。リスト3のオリジナル・ソースは、Foo.javaというテスト・プログラムから得られたものです。

リスト3. Foo.javaのオリジナル・ソースの抜粋

```
private int member = 10;

     public Foo() {
       int local = returnInteger();
       System.out.println( "foo constructor" );
       priv( local );
     }
```

そして、Mochaによって生成されたコードは次のとおりです。

リスト4. Mochaで生成されたFoo.javaのソース

```
       private int member;

       public Foo()
       {
           member = 10;
           int local = returnInteger();
           System.out.println("foo constructor");
           priv(local);
       }
```

上の2つの抜粋したコードを見ると、メンバー変数 `member` が10に初期化される部分が異なっています。オリジナル・ソースでは、宣言と同じ行で初期値として表現されていますが、逆コンパイ

ルされたソースでは、コンストラクターの内側の代入文として表現されています。逆コンパイルされたコードからは、オリジナル・ソースがどのようにコンパイルされたのかをある程度うかがい知ることができます。つまり、初期値は、コンストラクター内の代入としてコンパイルされているということです。このように、Javaコンパイラーの出力を逆コンパイルして調べることにより、Javaコンパイラーがとのように機能しているのかをかなり知ることができます。

逆コンパイルの複雑性: 繰り返し

Mochaはオブジェクト・コードの逆コンパイルに優れた効果を発揮しますが、常に成功するわけではありません。ソースを完全に復元できる逆コンパイラーはありません。問題が難しいためと、復元の際に抜けている部分をどう処理するかが、逆コンパイラーによって異なるためです。たとえば、Mochaは、ループ構成の正確な構造を割り出せないことがあります。その場合Mochaは、リスト5に示すように、最後の手段として、出力の中に `goto` ステートメントを挿入します。

リスト5. Mochaが正常に逆コンパイルできなかった例

```
            if (i1 == i3) goto 214 else 138;
            j3 = getSegment(i3).getZOrder();
            if (j1 != 1) goto 177 else 154;
            if (j3 > k2 && (!k1 || j3 < j2)) goto 203 else 173;
            expression 0
            if (j3 < k2 && (!k1 || j3 > j2)) goto 203 else 196;
            expression 0
            if == goto 201
            continue;
            i2 = i3;
```

Mochaに問題はありますが、一般的に逆コンパイラーはかなり正確にソースを復元します。逆コンパイラーの弱点さえ分かっていれば、逆コンパイルされたコードを手作業で分析し、微調整することにより、オリジナル・ソースをかなり正確に再現することができます。ただ、逆コンパイラーが常に改良され続けていることによって、新たに問題となっていることがあります。コードを他人に逆コンパイルされたくない場合には、どうしたらよいのでしょうか?

逆コンパイルとセキュリティーに対する脅威

多くの場合、コードの逆コンパイルが公正に行われている一方で、優れた逆コンパイラーはソフトウェアの海賊版を作るために欠かせないツールであるという事実もあります。 したがって、Javaコードのための安価な (または無料の) 逆コンパイル・ツールの存在は、特に商用のクローズド・ソースの舞台で仕事をしている開発者にとっては、深刻な問題です。

他の言語と比較すると、Javaコードは、例外的とも言えるほど逆コンパイルが容易です。これは、Java仮想マシンが (実マイクロプロセッサーと比較して) 相対的に単純であることや、Java言語のバイトコード・フォーマットに関するドキュメントが整っているためです。この逆コンパイルの容易さとWeb開発プラットフォームとしてのJava言語の圧倒的な人気から、商用開発分野で多くの議論が生じてきました。ソース・コードを保護することに熱心な会社や個人は、1996年に初めてMochaが紹介されたときから、Java逆コンパイラーを巡って大騒ぎしています。

事実、Mochaが最初にリリースされたときに、その作者であるHanpeter van Vlietは、数社から訴えられそうになりました (参考文献を参照)。最初彼は、自分のWebサイトからこの逆コンパイ

ラーを削除しましたが、後になってCremaという、より良い解決策を考え出しました。CremaはMochaの完全な対抗手段、すなわちJava曖昧化ツールです。

Cremaに続いて、多くのJava曖昧化ツールが登場しました。それらの中には、商用のものも、オープン・ソースのものもあります。後で示すように、優れた曖昧化ツールを使用すると、Javaコードの保護を大幅に強化することができます。

救済策としての曖昧化

コードの曖昧化は、文字通り、コードをあいまいにする操作です。Java曖昧化ツールは、巧みな方法でプログラムを変更します。JVMでの 振る舞い は変更前と同じであっても、人間には分かりづらくしてしまうのです。

曖昧化されたコードに対して逆コンパイラーを実行したときにどのようなことが起こるのか、サンプルを使って見てみましょう。リスト6は、jmangleというツールで曖昧化されたJavaコードをMochaで逆コンパイルしようとした場合の結果を示しています。一見気付かないかもしれませんが、下の抜粋は、前のいくつかのリストで使用したものと同じです。

リスト6. jmangleによって曖昧化されたコード

```
public Foo()
{
    jm2 = 10;
    int i = jm0();
    System.out.println("foo constructor");
    jm1(i);
}
```

jmangleなどの曖昧化ツールは、変数やメソッドの名前(場合によってはクラスやパッケージの名前なども) を無意味なストリングに変更します。これによって、人間には読みにくく、しかもJVMで実行する結果は、本質的にオリジナルと同じになります。

巧妙な手段

すべての曖昧化ツールはシンボルを無意味にしますが、それだけではありません。Cremaは、逆コンパイルを妨げる多くの意地悪な手段を提供していることで有名ですが、その後作成された多くの曖昧化ツールがそれに追随しています。

ソースをぼかすための一般的な方法の1つとして、無意味なストリングに書き換えてしまう仕掛けをさらに次のレベルにまで進めて、クラス・ファイルに含まれているシンボルを正しくないストリングで置き換えるやり方があります。 `private` などのキーワードに置き換えてしまったり、さらに徹底して、`***` などまったく意味のないシンボルに置き換えてしまったりするなどの例があります。仮想マシンの中には、特にブラウザーなどの場合、このような仕掛けに対応しないものがあります。たとえば、技術的には、`=` などの変数はJava仕様と相いれません。このような変数でも受け入れてしまう仮想マシンと、そうでない仮想マシンがあります。

Cremaの爆弾

Cremaは、別の仕掛けも持っていますが、この仕掛けは、文字通り爆弾と言えます。Cremaには、Mochaを完全にシャットダウンする機能が備わっていました。これは、コンパイルされた

コードに小さな「爆弾」を追加して、Mochaがコードを逆コンパイルしようとしたときにMochaをクラッシュさせるというものでした。

現在では、もうCremaは入手できなくなっていますが、Cremaと同じように特にMochaをシャットダウンするために、HoseMochaというツールが設計されました。HoseMochaがどのように機能するのかを示すために、ここでもまた逆アセンブラーのjavapを使用してみましょう。リスト7は、HoseMochaが爆弾を仕掛ける前のコードを示しています。

リスト7. 爆弾が仕掛けられる前のコード

```
Method void main(java.lang.String[])
      0 new #4
      3 invokespecial #10
      6 return
```

次に、HoseMochaによって処理された後のコードを示します。

リスト8. 爆弾が仕掛けられた後のコード

```
Method void main(java.lang.String[])
      0 new #4
      3 invokespecial #10
      6 return
      7 pop
```

爆弾に気が付きましたか? ルーチンが戻った後でpop命令が出されていますね。しかし、戻った後で関数が何か処理を行うなどということがあり得るのでしょうか? 明らかに不可能です。これが仕掛けなのです。戻りステートメントの後に命令を配置することでその命令が決して実行されないことが保証されます。つまり、このコードは、本質的に逆コンパイルできないことになります。と言うのは、このコードは、Javaソース・コードに対応していないためです。

しかし、なぜこのようなささいな欠陥が原因でMochaがクラッシュするのでしょうか? このような欠陥を無視したり、警告を発して先に進むようなことなど簡単にできそうなものです。Mochaがこの種の爆弾に弱いことはバグと考えることもできますが、van VlietがMochaに関する懸念を踏まえて故意にこのように作ったと考えるほうがよさそうです。

ここまで、古い逆コンパイルと曖昧化ツール (古いけれども気の利いたもの) を見てきました。これらのツールは、特にグラフィカル・インターフェースの点で、年々洗練の度合いを増しています。最後に、最近の動向を理解していただくために、より新しい逆コンパイラーに注目してみましょう。

最近のツール

過去5年の間に複雑さを増したのは逆コンパイルや曖昧化の技法だけではありません。それらの技法へのインターフェースもますます巧妙になってきています。最近の逆コンパイラーでは、.classファイルのディレクトリーをブラウズして、.classファイルを1回クリックするだけで逆コンパイルすることができるものもあります。

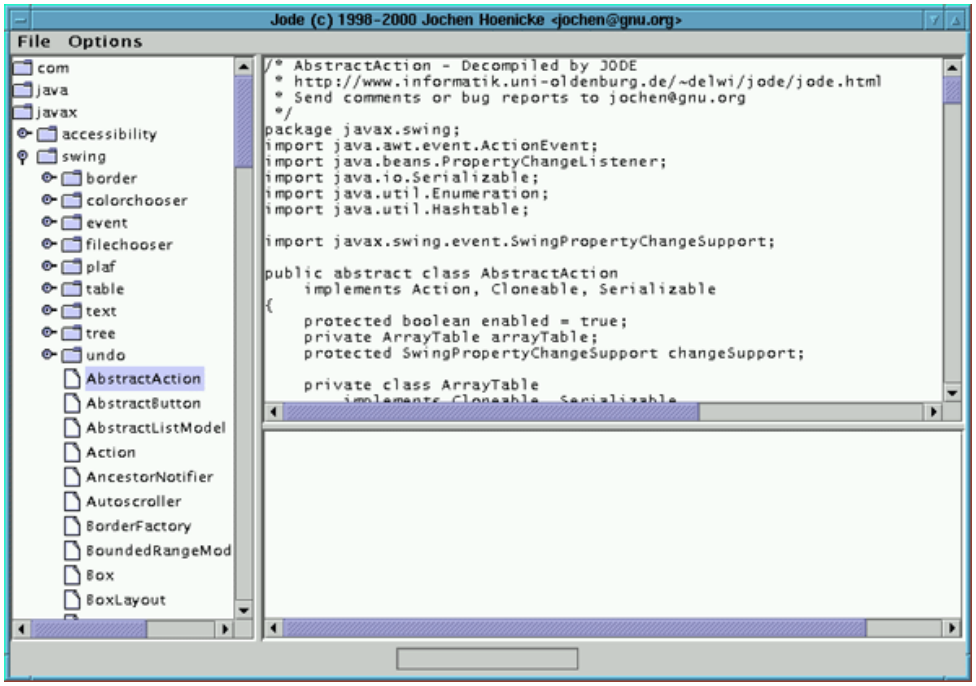
JODE (Java Optimize and Decompile Environment) はこうしたプログラムの1つです。コマンド行に .jarファイルの名前を入力すると、クラスをグラフィカルにブラウズできます。各クラスは自動

的に逆コンパイルされますが、その様子を確認することもできるのです。これは特に、Java SDKとともに提供されたライブラリーまでソース・コードをたどる場合などに便利です。使い方は、非常に簡単です。次のコマンドを入力してください。

```
$ java jode.swingui.Main --classpath [path to your Java SDK]/jre/lib/rt.jar
```

これにより、図1に示すように、ファイルがきれいに表示されます。

図1. JODE: 逆コンパイラー



その他の便利なツールのリストについては、参考文献を参照してください。

結論

MochaやHoseMochaなど昔からある保守的なツールを選択するか、より新しいツールを追究するか、いずれにしても、Java逆コンパイラーと曖昧化ツールについて学習するための出発点としてこの記事を役立てていただければと思います。ここを起点に、参考文献セクションで提供されたリンクをサーフィンし、ご自分でツールをいくつかお試しになることで、スキルを上げる手掛かりにしてください。いろいろと議論はありますが、逆コンパイルおよび曖昧化の技法はすでに普及していますし、今後数年でさらに洗練されることは間違いありません。

著者について

Greg Travis

Greg Travisはニューヨーク市在住のフリーランス・プログラマーです。コンピューターに関する彼の興味は、「Bionic Woman」の中でジェイミーが、スピーカーを使って彼女をあざける邪悪な人工知能によって照明とドアが制御されているビルから逃げ出そうとした話にまで、さかのぼることができます。Gregは、コンピューター・プログラムがちゃんと機能するのは、まったく偶然であると、固く信じています。Gregの連絡先は mito@panix.com です。