# Java プログラミングのダイナミックス: 第 8 回 リフレクションに取って代わるコード生成

## ランタイム・コード生成は、最高のパフォーマンスを目指し直接アクセスでリフレクションに取って代わる方法を提供します

Dennis Sosnoski
President
Sosnoski Software Solutions, Inc.

2004年 6月 10日

この記事のシリーズの序盤では、直接アクセスと比較してどれくらいリフレクションのパフォーマンスが遅いかを学び、そして Javassist と Apache Byte Code Engineering Library (BCEL) との classworking について学びました。Java コンサルタントである Dennis Sosnoski は、(全速で走る生成コードにリフレクション・コードの座を奪わせる) ランタイム classworking をいかにして活用するかをデモンストレーションすることにより、彼の「Java プログラミングのダイナミックス」シリーズに終止符を打ちます。

このシリーズの他の記事を見る

Javassist と BCEL のフレームワークをどのようにして classworking に活用するかをすでにご存知と思いますので (シリーズの過去記事のリストをご覧ください。)、これから実用的な classworking アプリケーションを説明します。このアプリケーションはリフレクションの使用の代わりにランタイム時に生成され直後に JVM にロードされるクラスを採用します。それをまとめる過程で、最初の 2 本の記事そして Javassist と BCEL のカバレッジに戻り、長くなってしまった記事のシリーズを良い形で締めくくります。

## パフォーマンスのリフレクション

このシリーズの第 2 回で、リフレクションがフィールド・アクセスとメソッド呼び出しの直接コードに比べていかに遅いかを説明しました。多くのアプリケーションではこの『鈍重さ』は問題にはならないのですが、パフォーマンスが致命的に重大になる場合は常にあり得ます。そのようなケースでは、リフレクションはボトルネックとして全体の足を引っ張り得ます。しかし、静的にコンパイルされたコードにリフレクションの後を継がせるとかなりややこしくなり、時には (リフレクションにアクセスされるクラスやアイテムが (同一の構築プロセスの一部としてよりも) ランタイムに供給される、フレームワークの場合のように) アプリケーションそのものを作り直さなくてはならなくなります。

classworking は静的にコンパイルされたコードのパフォーマンスとリフレクションの柔軟性を合体させる選択肢を提供します。汎用のコードに使われるように (以前はリフレクションも届いた) ターゲット・クラスへのアクセスをラップするカスタム・クラスをランタイムにて構築するのが、ここでの基本的なアプローチです。JVM にカスタム・クラスをロードした後には、全速で走らせる事ができます。

## 舞台設定

リスト 1 はアプリケーションの出発点を提供します。ここでは単純な Bean クラス `HolderBean` そしてアクセス・クラス `ReflectAccess` を定義します。アクセス・クラスは、『`int` value』を持つ Bean クラスのプロパティー (`value1` か `value2`) のうちの 1 つの名前でなくてはならない 1 つのコマンド・ライン引き数を扱います。それは名前の付いたプロパティーの値を増分し、終了する前に両方のプロパティー値をプリントアウトします。

## リスト 1. Bean のリフレクション

```
public class HolderBean
{
    private int m_value1;
    private int m_value2;
    public int getValue1() {
        return m_value1;
    }
    public void setValue1(int value) {
        m_value1 = value;
    }
    public int getValue2() {
        return m_value2;
    }
    public void setValue2(int value) {
        m_value2 = value;
    }
}
public class ReflectAccess
{
    public void run(String[] args) throws Exception {
        if (args.length == 1 & args[0].length() > 0) {
            // create property name
            char lead = args[0].charAt(0);
            String pname = Character.toUpperCase(lead) +
                args[0].substring(1);
            // look up the get and set methods
            Method gmeth = HolderBean.class.getDeclaredMethod
                ("get" + pname, new Class[0]);
            Method smeth = HolderBean.class.getDeclaredMethod
                ("set" + pname, new Class[] { int.class });
            // increment value using reflection
            HolderBean bean = new HolderBean();
            Object start = gmeth.invoke(bean, null);
            int incr = ((Integer)start).intValue() + 1;
            smeth.invoke(bean, new Object[] {new Integer(incr)});
            // print the ending values
            System.out.println("Result values " +
                bean.getValue1() + ", " + bean.getValue2());
            } else {
            System.out.println("Usage: ReflectAccess value1|value2");
        }
    }
}
```

結果を明示するために、ここで `ReflectAccess` のサンプル実行を表記します。

```
[dennis]$ java -cp . ReflectAccess value1
Result values 1, 0
[dennis]$ java -cp . ReflectAccess value2
Result values 0, 1
```

## グルー・クラス(glue class)を構築

リフレクション版のコードを実例で説明しましたので、ここでどのようにして生成コードにリフレクションの用途の役割を継がせるかを示します。この置換を正しく機能させようとすると、(このシリーズ第 1 回の記事で論議されたクラスのロード処理に関与する) 微妙な問題がからんできます。アクセス・クラスの静的にコンパイルされたコードから、ランタイムの時点でアクセスしたいクラスを生成したいのですが、その生成されるクラスはコンパイラには存在しないのでそれに直接参照する手段が無いのが問題点です。

### このシリーズのその他の記事

- 第 1 回 クラスとクラスのロード処理
- 第 2 回 リフレクション入門
- 第 3 回 実用的なリフレクション
- 第 4 回 Javassist でのクラス変換
- 第 5 回 オンザフライでクラスを変換する
- 第 6 回 Javassist を使用したアスペクト指向の変更
- 第 7 回 Bytecode engineering with BCEL (英語)
- 第 8 回 リフレクションに取って代わるコード生成

それならば、どうすれば静的にコンパイルされたコードを生成されたクラスにリンクできるのでしょうか？基本的な解決法は、静的にコンパイルされたコードにアクセス可能なインターフェースまたは基本クラスを定義してから、生成されたクラスにそのインターフェースを実装するかその基本クラスを拡張する方法です。ランタイム直前までメソッドが実装されないにもかかわらず、静的にコンパイルされたコードはメソッドに直接呼び出しを行なうことができます。

リスト 2 では、生成コードへのリンクを供給するインターフェース、`Iaccess` を定義しました。インターフェースは 3 つのメソッドを含みます。最初のメソッドはアクセスされるターゲット・オブジェクトを設定するだけです。他の 2 つのメソッドはプロパティー値 `int` にアクセスするのに使用される get と set のメソッド用のプロキシです。

### リスト 2. グルー・クラスへのインターフェース

```
public interface IAccess
{
    public void setTarget(Object target);
    public int getValue();
    public void setValue(int value);
}
```

ここでの趣旨は、`Iaccess` インターフェースの生成された実装がターゲット・クラスの適切な get と set のメソッドを呼び出すコードを提供することです。ここではリスト 1 にある `HolderBean` クラスの `value1` プロパティーにアクセスしたいと想定した場合、このインターフェースがどのように実装されるかをリスト 3 にて示します。

## リスト 3. グルー・クラスのサンプル実装

```
public class AccessValue1 implements IAccess
{
    private HolderBean m_target;
    public void setTarget(Object target) {
        m_target = (HolderBean)target;
    }
    public int getValue() {
        return m_target.getValue1();
    }
    public void setValue(int value) {
        m_target.setValue1(value);
    }
}
```

特定のオブジェクト・タイプの特定のプロパティーに使われるように、リスト 2 のインターフェースは設計されています。このインターフェースは実装コードをシンプルにまとめます (バイトコード使用時には常に有利です) が、同時に実装クラスは非常に限定的です。リフレクションの汎用的な後釜としては、そのインターフェースは柔軟性に欠けます。(汎用的で柔軟性の高いアプローチに頼ることを抑制するそのインターフェースを通してのアクセスを可能にする) それぞれの種類のオブジェクトとプロパティーには、独立した実装クラスが必要となります。リフレクションのパフォーマンスがボトルネックとなる場合にその技術を選択的に利用するのであれば、その限界は問題にならないでしょう。

## Javassist で生成

Javassist を活用してリスト 2 の `IAccess` インターフェース用の実装クラスを生成するのは簡単です。インターフェースを実装する新規クラスを作成し、ターゲット・オブジェクト・リファレンスのメンバー変数を追加し、そして非引き数コンストラクターと単純な実装メソッドを追加して終了させるだけです。それらのステップを完成させるために、ターゲット・クラスと get/set メソッドの情報を入手しそして構成されたクラスのバイナリー表現を戻すメソッド呼び出しとして構造化された Javassist コードを、リスト 4 に表記します。

## リスト 4. Javassist グルー・クラス構成

```
/** Parameter types for call with no parameters. */
private static final CtClass[] NO_ARGS = {};
/** Parameter types for call with single int value. */
private static final CtClass[] INT_ARGS = { CtClass.intType };
protected byte[] createAccess(Class tclas, Method gmeth,
    Method smeth, String cname) throws Exception {
      // build generator for the new class
      String tname = tclas.getName();
      ClassPool pool = ClassPool.getDefault();
      CtClass clas = pool.makeClass(cname);
      clas.addInterface(pool.get("IAccess"));
      CtClass target = pool.get(tname);
      // add target object field to class
      CtField field = new CtField(target, "m_target", clas);
      clas.addField(field);
      // add public default constructor method to class
      CtConstructor cons = new CtConstructor(NO_ARGS, clas);
      cons.setBody(";");
      clas.addConstructor(cons);
      // add public setTarget method
      CtMethod meth = new CtMethod(CtClass.voidType, "setTarget",
          new CtClass[] { pool.get("java.lang.Object") }, clas);
      meth.setBody("m_target = (" + tclas.getName() + ")$1;");
```

```
        clas.addMethod(meth);
        // add public getValue method
        meth = new CtMethod(CtClass.intType, "getValue", NO_ARGS, clas);
        meth.setBody("return m_target." + gmeth.getName() + "();");
        clas.addMethod(meth);
        // add public setValue method
        meth = new CtMethod(CtClass.voidType, "setValue", INT_ARGS, clas);
        meth.setBody("m_target." + smeth.getName() + "($1);");
        clas.addMethod(meth);
        // return binary representation of completed class
        return clas.toBytecode();
}
```

仮にみなさんがこのシリーズをチェックしているのであれば、ここに表記されているオペレーションはすでにお馴染みのはずですので、あえてここで詳しく説明しません。(このシリーズをまだチェックしていないのであれば、シリーズ第 5 回の記事を読んで Javassist 関連の全体像を把握してください。)

## BCEL で生成

BCEL を利用してリスト 2 の `IAccess` 用の実装クラスを生成するのは Javassist 使用時ほどに簡単ではないのですが、それでも異常なほどに複雑と言うわけでもありません。この目的に使用されるコードをリスト 5 で示します。このコードはリスト 4 の Javassist コードと同様のオペレーションのシーケンスを使用しますが、BCEL の場合それぞれのバイトコード命令を詳細にわたって書く必要があるのでコードが長くなりがちです。Javassist バージョン同様、この実装の詳細にわたる説明を省略します。)不明な点がありましたら、このシリーズ第 7 回の記事で BCEL 関連の全体像を把握してください。)

## リスト 5. BCEL グルー・クラス構成

```
/** Parameter types for call with single int value. */
    private static final Type[] INT_ARGS = { Type.INT };
/** Utility method for adding constructed method to class. */
private static void addMethod(MethodGen mgen, ClassGen cgen) {
    mgen.setMaxStack();
    mgen.setMaxLocals();
    InstructionList ilist = mgen.getInstructionList();
    Method method = mgen.getMethod();
    ilist.dispose();
    cgen.addMethod(method);
}
protected byte[] createAccess(Class tclas,
    java.lang.reflect.Method gmeth, java.lang.reflect.Method smeth,
    String cname) {
    // build generators for the new class
    String tname = tclas.getName();
    ClassGen cgen = new ClassGen(cname, "java.lang.Object",
        cname + ".java", Constants.ACC_PUBLIC,
        new String[] { "IAccess" });
    InstructionFactory ifact = new InstructionFactory(cgen);
    ConstantPoolGen pgen = cgen.getConstantPool();
    //. add target object field to class
    FieldGen fgen = new FieldGen(Constants.ACC_PRIVATE,
        new ObjectType(tname), "m_target", pgen);
    cgen.addField(fgen.getField());
    int findex = pgen.addFieldref(cname, "m_target",
        Utility.getSignature(tname));
    // create instruction list for default constructor
    InstructionList ilist = new InstructionList();
    ilist.append(InstructionConstants.ALOAD_0);
    ilist.append(ifact.createInvoke("java.lang.Object", "<init>",
```

```
        Type.VOID, Type.NO_ARGS, Constants.INVOKESPECIAL));
    ilist.append(InstructionFactory.createReturn(Type.VOID));
    // add public default constructor method to class
    MethodGen mgen = new MethodGen(Constants.ACC_PUBLIC, Type.VOID,
        Type.NO_ARGS, null, "<init>", cname, ilist, pgen);
    addMethod(mgen, cgen);
    // create instruction list for setTarget method
    ilist = new InstructionList();
    ilist.append(InstructionConstants.ALOAD_0);
    ilist.append(InstructionConstants.ALOAD_1);
    ilist.append(new CHECKCAST(pgen.addClass(tname)));
    ilist.append(new PUTFIELD(findex));
    ilist.append(InstructionConstants.RETURN);
    // add public setTarget method
    mgen = new MethodGen(Constants.ACC_PUBLIC, Type.VOID,
        new Type[] { Type.OBJECT }, null, "setTarget", cname,
        ilist, pgen);
    addMethod(mgen, cgen);
    // create instruction list for getValue method
    ilist = new InstructionList();
    ilist.append(InstructionConstants.ALOAD_0);
    ilist.append(new GETFIELD(findex));
    ilist.append(ifact.createInvoke(tname, gmeth.getName(),
        Type.INT, Type.NO_ARGS, Constants.INVOKEVIRTUAL));
    ilist.append(InstructionConstants.IRETURN);
    // add public getValue method
    mgen = new MethodGen(Constants.ACC_PUBLIC, Type.INT,
        Type.NO_ARGS, null, "getValue", cname, ilist, pgen);
    addMethod(mgen, cgen);
    // create instruction list for setValue method
    ilist = new InstructionList();
    ilist.append(InstructionConstants.ALOAD_0);
    ilist.append(new GETFIELD(findex));
    ilist.append(InstructionConstants.ILOAD_1);
    ilist.append(ifact.createInvoke(tname, smeth.getName(),
        Type.VOID, INT_ARGS, Constants.INVOKEVIRTUAL));
    ilist.append(InstructionConstants.RETURN);
    // add public setValue method
    mgen = new MethodGen(Constants.ACC_PUBLIC, Type.VOID,
        INT_ARGS, null, "setValue", cname, ilist, pgen);
    addMethod(mgen, cgen);
    // return bytecode of completed class
    return cgen.getJavaClass().getBytes();
}
```

## パフォーマンスのチェック

ここで Javassist バージョンと BCEL バージョンのメソッド構造がどれほどに機能するかを試す事ができます。ランタイムでコードを生成する当初の趣旨はリフレクションの (より速い) 代わりを探すことでしたので、ここで成功したかどうかをパフォーマンス比較を通して確認するのが得策です。さらに、それぞれのフレームワークにてグルー・クラスがどれだけの時間を費やすかを観察します。

パフォーマンスをチェックするためのテスト・コードの主たる部分をリスト 6 に示します。`runReflection()` メソッドはテストのリフレクション部分を担当し、`runAccess()` はダイレクト・アクセスの部分を担い、そして `run()` は (プリンティングとタイミングの結果を含む) 全体のプロセスを制御します。`runReflection()` と `runAccess()` は両方とも、(このリストには含まれないがダウンロードに含まれるコードを使用して) コマンド・ラインから受け渡されるパラメーターとして実行されるループ数を取り入れます。(リスト 6 の最後にある) `DirectLoader` クラスは生成されたクラスをロードする簡単な方法を提供します。

## リスト 6. パフォーマンスのテスト・コード

```
/** Run timed loop using reflection for access to value. */
private int runReflection(int num, Method gmeth, Method smeth,
    Object obj) {
    int value = 0;
    try {
        Object[] gargs = new Object[0];
        Object[] sargs = new Object[1];
        for (int i = 0; i < num; i++) {
            // messy usage of Integer values required in loop
            Object result = gmeth.invoke(obj, gargs);
            value = ((Integer)result).intValue() + 1;
            sargs[0] = new Integer(value);
            smeth.invoke(obj, sargs);
            }
    } catch (Exception ex) {
        ex.printStackTrace(System.err);
        System.exit(1);
    }
    return value;
}
/** Run timed loop using generated class for access to value. */
private int runAccess(int num, IAccess access, Object obj) {
    access.setTarget(obj);
    int value = 0;
    for (int i = 0; i < num; i++) {
        value = access.getValue() + 1;
        access.setValue(value);
    }
    return value;
}
public void run(String name, int count) throws Exception {
    // get instance and access methods
    HolderBean bean = new HolderBean();
    String pname = name;
    char lead = pname.charAt(0);
    pname = Character.toUpperCase(lead) + pname.substring(1);
    Method gmeth = null;
    Method smeth = null;
    try {
        gmeth = HolderBean.class.getDeclaredMethod("get" + pname,
            new Class[0]);
        smeth = HolderBean.class.getDeclaredMethod("set" + pname,
            new Class[] { int.class });
    } catch (Exception ex) {
        System.err.println("No methods found for property " + pname);
        ex.printStackTrace(System.err);
        return;
    }
    // create the access class as a byte array
    long base = System.currentTimeMillis();
    String cname = "IAccess$impl_HolderBean_" + gmeth.getName() +
        "_" + smeth.getName();
    byte[] bytes = createAccess(HolderBean.class, gmeth, smeth, cname);
    // load and construct an instance of the class
    Class clas = s_classLoader.load(cname, bytes);
    IAccess access = null;
    try {
        access = (IAccess)clas.newInstance();
    } catch (IllegalAccessException ex) {
        ex.printStackTrace(System.err);
        System.exit(1);
    } catch (InstantiationException ex) {
        ex.printStackTrace(System.err);
        System.exit(1);
    }
```

```
    System.out.println("Generate and load time of " +
        (System.currentTimeMillis()-base) + " ms.");
    // run the timing comparison
    long start = System.currentTimeMillis();
    int result = runReflection(count, gmeth, smeth, bean);
    long time = System.currentTimeMillis() - start;
    System.out.println("Reflection took " + time +
        " ms. with result " + result + " (" + bean.getValue1() +
        ", " + bean.getValue2() + ")");
    bean.setValue1(0);
    bean.setValue2(0);
    start = System.currentTimeMillis();
    result = runAccess(count, access, bean);
    time = System.currentTimeMillis() - start;
    System.out.println("Generated took " + time +
        " ms. with result " + result + " (" + bean.getValue1() +
        ", " + bean.getValue2() + ")");
}
/** Simple-minded loader for constructed classes. */
protected static class DirectLoader extends SecureClassLoader
{
    protected DirectLoader() {
        super(TimeCalls.class.getClassLoader());
    }
    protected Class load(String name, byte[] data) {
        return super.defineClass(name, data, 0, data.length);
    }
}
```

単純なタイミングのテストを実行するには、`run()` メソッドを 2 度 (リスト 1 の `HolderBean` クラスにあるそれぞれのプロパティーにつき 1 度ずつ) 呼び出します。2 つのテスト受け渡しを実行することは、道理に合って公平なテストには重要です。コードを通る最初の受け渡しは、Javassist と BCEL のクラス生成プロセスに多くのオーバーヘッドを追加する全ての必要なクラスをロードします。このオーバーヘッドは 2 つ目の受け渡しには必要とされていませんので、実際のシステム内で使用される場合にクラス生成がどれだけの時間を必要とするかのより正確な予測を提供します。テストが実行されるときに生成される出力のサンプルを以下に示します。

```
[dennis]$$ java -cp .:bcel.jar BCELCalls 2000
Generate and load time of 409 ms.
Reflection took 61 ms. with result 2000 (2000, 0)
Generated took 2 ms. with result 2000 (2000, 0)
Generate and load time of 1 ms.
Reflection took 13 ms. with result 2000 (0, 2000)
Generated took 2 ms. with result 2000 (0, 2000)
```

2K から 512K にわたるループ・カウント (回数) で呼び出された場合のタイミング・テストの結果を、図 1 に示します (このテスト環境では、Athlon 2200+ XP システム上で、Sun 1.4.2 JVM を使用し、Mandrake Linux 9.1 を実行します)。それぞれのテスト実行において (オーバーヘッドに邪魔されない) 2 つ目のプロパティーのリフレクション時間と生成コードの時間を両方グラフにまとめました。図 1 のグラフでは、Javassist コード生成使用時の時間の組み合わせ (青の Reflection と茶色の Javassist) が最初の 2 段に表記され、BCEL コード生成の同様の組み合わせ (黄色の Reflection と水色の BCEL) が下 2 段に続きます。Javassist か BCEL のどちらを使用してグルー・クラスを生成するかに関係なく、実行タイムはほぼ同一です。これは予測された結果ですが、予想通りの結果を実際に観察すれば安心できるものです。

リフレクション対生成コードのスピード対決 (単位はミリ秒)

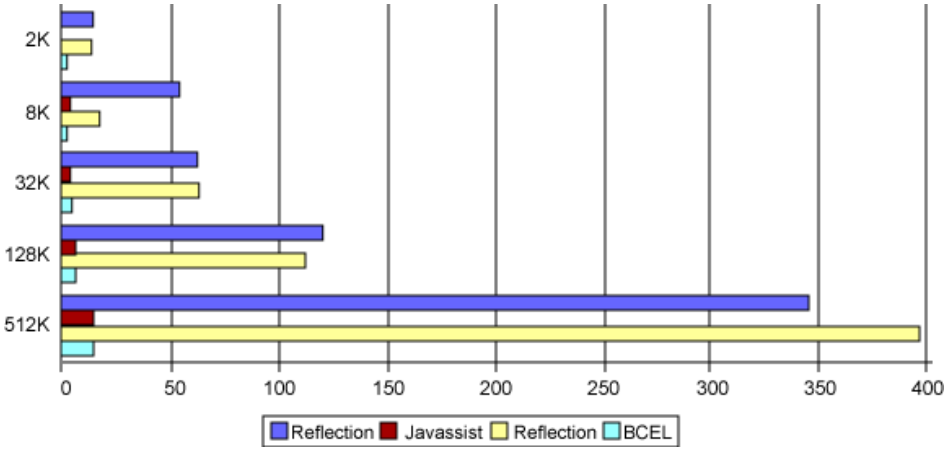


図 1 で一目瞭然のとおり、どのケースでもリフレクションよりも遥かに速く生成コードが実行されます。生成コードのスピード面での優位性は、ループ回数が増えるほどに増していきます。(生成コードのスピードとリフレクションのスピードの割合は) 2K ループではだいたい 5:1 の割合で、512K のループではだいたい 24:1 の割合まで増加します。最初のグルー・クラスの構成とロードは、Javassist では 320ms (ミリ秒) そして BCEL では 370ms (ミリ秒) を費やし、2 度目のグルー・クラスの構成は Javassist ではたったの 4ms を費やし、BCEL では 2ms でした (クロックの精度は 1ms ですので、測定時間の精度はかなり大雑把になります。) これらの時間を組み合わせれば、2K ループの場合でも、リフレクションよりもクラスを生成するケースの方が全体的にはより良いパフォーマンスを示します (リフレクションの合計実行時間が 14ms なのに対し、コード生成時の合計実行時間は 4ms から 6ms です。)。

実際上、図が示すよりも生成コードの優位性は顕著です。25 ループまで下げて実験を行なった場合、リフレクションはまだ 6ms から 7ms の時間を要しますが生成コードは測定不能なほどに高速でした。少ないループでのリフレクションが費やす時間は、しきい値に達した時の JVM 内での最適化を反映している模様です。ループ回数を約 20 以下まで下げると、リフレクションのコードも測定不能なほどに速くなりました。

## 更なる加速に向けて

ランタイム classworking がアプリケーションに与えるパフォーマンスがどのようなものかを示しました。今度パフォーマンスの最適化に関する難題に直面した場合には、この手法を思い出してください。この特効薬のおかげで、大掛かりな作り直しをせずに済むかも知れません。しかも、classworking はパフォーマンスだけに救いの手を差し伸べるわけではありません。それは、ランタイムの必須条件に合わせてアプリケーションを仕立てる独特なほどに柔軟なアプローチでもあるのです。たとえ自分のコードにそれを採用する理由が無いとしても、それはプログラミングを楽しくそして興味深いものにする Java の特性だと言えるでしょう。

classworking の実社会でのアプリケーションの記事で、この「Java プログラミングのダイナミックス」のシリーズを締めくくります。これは悲観すべき事ではありません。私が Java バイトコード操作を中心に構築されたツールの一部を developerWorks の会合にて公開するときに、別の classworking アプリケーションの見本を目にする機会を得られます。まず最初に、Mother Goose から直接取り入れた一対のテスト・ツールに関する記事から始める予定です。

# 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。