Java プログラミングのダイナミックス: 第 2 回 リフレクション入門

実行時のクラス情報を使用して柔軟なプログラミングを行う

Dennis Sosnoski
President
Sosnoski Software Solutions, Inc.

2003年 6月 03日

リフレクションを利用すれば、JVM にロードされたクラスの内部情報をコードからアクセスしたり、ソース・コードで選択したクラスではなく実行時に選択するクラスを扱うようにコードを記述できるようになります。そういう理由で、リフレクションは柔軟性の高いアプリケーションを構築するのに非常に有効なツールとなっています。しかし注意も必要です。使い方を誤るとコストも高くつきます。Java プラットフォームの内部についてのこのシリーズの第 2 回目でソフトウェア・コンサルタント Dennis Sosnoski 氏が紹介してくれるのは、リフレクションの使い方およびそのときのコストについてです。また実行時に Java のリフレクション API を利用してオブジェクトにフックする方法も示されます。

このシリーズの他の記事を見る

「Java プログラミングのダイナミックス 第 1 回」では、Java プログラミングでのクラスとクラスのロードについて紹介しました。そこでは Java バイナリー・クラスのフォーマットに含まれているさまざまな情報について説明しました。今回は Java リフレクション API を使ってそれらの情報を実行時にアクセスし利用する基礎的な方法を解説します。すでにリフレクションの基礎を理解している開発者にとっても興味のあるものとなるように、リフレクションと直接アクセスとで性能にどの程度差があるのかということも示します。

リフレクションを使用する場合、メタデータ (他のデータを記述するデータ) を扱う点が通常の Java プログラミングと異なってきます。Java 言語のリフレクションでアクセスされるのは、JVM 内でのクラスやオブジェクトについての記述という種類のメタデータです。リフレクションによって、さまざまなクラス情報を実行時アクセスすることができます。実行時に選択したクラスのフィールドの読み書きやメソッド呼び出しすら可能になります。

リフレクションは強力なツールです。リフレクションを利用することで、ソース・コードでコンポーネント同士をリンクしなくても実行時に組み立てることのできる柔軟なコードを構築することができます。しかしリフレクションには問題を起こしやすい側面もあります。本稿では、プログラムでリフレクションを使ったほうがよい場合の理由だけでなく、リフレクションを使わない

ほうがよい場合の理由も説明したいと思います。この二律背反 (trade-offs) が理解できれば、どんな場合に利点が欠点に優るのかを皆さん自身で判断できることになります。

初心者のクラス

このシリーズのその他の記事

リフレクションを利用するときの出発点は、いつでも `java.lang.Class` のインスタンスです。定義済みのクラスを扱いたい場合、Java 言語には以下のように `Class` のインスタンスを直接手に入れるための簡便な方法が用意されています。

```
Class clas = MyClass.class;
```

この方法を使用する場合、クラスをロードするための処理は、すべて舞台裏で行われます。ただし、どこか外部から実行時にクラスの名前を読み出す必要がある場合、この方法ではうまくいきません。クラスの情報を得るためにはクラス・ローダーを使用する必要があります。その場合の1つの方法を示すのが以下のコードです。

```
// "name" is the class name to load
Class clas = null;
try {
  clas = Class.forName(name);
} catch (ClassNotFoundException ex) {
  // handle exception case
}
// use the loaded class
```

クラスがロード済みの場合には、すでに存在する `Class` の情報が返されてきます。クラスがまだロードされていない場合には、クラス・ローダーがこの時点でそのクラスをロードし、新たに構築されたクラスのインスタンスを返してきます。

クラスのリフレクション

`Class` オブジェクトには、クラスのメタデータをリフレクション・アクセスするための基本的なフックがすべて用意されています。このメタデータには、クラスによって実装されているインターフェースだけでなく、クラスのパッケージやスーパークラスなどのクラス自体についての情報も含まれます。また、そのクラスで定義されているコンストラクターやフィールドやメソッドの詳細も含まれます。この後の方の情報はプログラミングで非常によく使用されますので、後でそれらの扱い方の例をいくつか示すことにします。

これら 3種類のクラス・コンポーネント (コンストラクター、フィールド、メソッド) それぞれについて、`java.lang.Class` は 4通りのリフレクション呼び出しを用意しており、情報をいろいろな方法でアクセスできるようにしています。これらの呼び出しは、すべて同じ標準的な形式に従い

ます。以下はコンストラクターを読み出すときに使用される 4通りのリフレクション呼び出しです。

- `Constructor getConstructor(Class[] params)` -- パラメーターの型を指定してパブリック・コンストラクターを取得する
- `Constructor[] getConstructors()` -- そのクラスのすべてのパブリック・コンストラクターを取得する
- `Constructor getDeclaredConstructor(Class[] params)` -- パラメーターの型を指定して (アクセス・レベルに関係なく) コンストラクターを取得する
- `Constructor[] getDeclaredConstructors()` -- そのクラスのすべてのコンストラクターを (アクセス・レベルに関係なく) 取得する

これらの呼び出しは、いずれも `java.lang.reflect.Constructor` のインスタンスを 1個以上返します。この `Constructor` クラスは、その唯一の引数にオブジェクトの配列をとり、元のクラスから新たに構築されたインスタンスを返す `newInstance` メソッドを定義しています。オブジェクトの配列はコンストラクター呼び出しに使用されるパラメーター値です。この動作を示す例として、リスト 1 に示すような `TwoString` というクラスがあり、そのコンストラクターが 1 対の `String` 引数をとるものとします。

リスト 1. 1 対の文字列を基に構築されるクラス

```
public class TwoString {
    private String m_s1, m_s2;
    public TwoString(String s1, String s2) {
        m_s1 = s1;
        m_s2 = s2;
    }
}
```

リスト 2 に示すコードは、このコンストラクターを取得し、それを使って `"a"` と `"b"` の 2個の`String` で`TwoString` クラスのインスタンスを作成しています。

リスト 2. コンストラクターのリフレクション呼び出し

```
    Class[] types = new Class[] { String.class, String.class };
    Constructor cons = TwoString.class.getConstructor(types);
    Object[] args = new Object[] { "a", "b" };
    TwoString ts = cons.newInstance(args);
```

リスト 2 のコードは、いろいろなリフレクション・メソッドからスローされチェックされる可能性のあるいく通りかの例外の種類を無視しています。これらの例外については Java API ドキュメントで詳しく説明されていますので、ここでは簡潔にするために、すべてのサンプル・コードでこれらの例外を省略しています。

コンストラクターに関して言えば、Java プログラミング言語では、引数のない (すなわちデフォルトの) コンストラクターを使ってクラスのインスタンスを簡便に作成するための特殊なメソッドも定義されています。この簡便な方法は、以下のように `Class` の定義自体に組み込まれています。

> `Object newInstance()` -- デフォルトのコンストラクターを使って新しいインスタンスを構築する

この方法では 1個のコンストラクターしか使用できませんが、それで充分な場合には非常に簡便です。この手法は、パブリックで引数のないコンストラクターを定義する必要のある JavaBeans を扱う場合に特に有効です。

リフレクションによるフィールド

フィールド情報にアクセスするための `Class` のリフレクション呼び出しは、コンストラクターにアクセスするための呼び出しと似ていますが、パラメーターの型の配列の代わりにフィールドの名前が使われます。

- `Field getField(String name)` -- 指定した名前のパブリック・フィールドを取得する
- `Field[] getFields()` -- そのクラスのすべてのパブリック・フィールドを取得する
- `Field getDeclaredField(String name)` -- そのクラスで宣言されているフィールドのうち指定した名前のものを取得する
- `Field[] getDeclaredFields()` -- そのクラスで宣言されているすべてのフィールドを取得する

コンストラクター呼び出しと似てはいますが、フィールドに関しては大きな違いが 1つあります。最初の 2つの呼び出しは、祖先クラスから継承されたフィールドであったとしても、そのクラスを通してアクセスされるパブリック・フィールドについての情報を返すという点です。これに対して後の 2つの呼び出しは、フィールドのアクセス・タイプには関係なく、そのクラスで直接宣言されたフィールドについての情報を返します。

これらの呼び出しで返されてくる `java.lang.reflect.Field` のインスタンスは、オブジェクトの参照を扱うための汎用的な `get/set` メソッドの他に、すべての基本型について `getXXX/setXXX` メソッドを定義しています。`getXXX` メソッドは自動的に型拡張変換を行うようになってはいますが (`###` 値を読み出すために `getInt` メソッドを使用するなど)、実際のフィールドの型に合わせてどのメソッドを使用するかはユーザーが決定する必要があります。

リスト 3 は、フィールド・リフレクション・メソッドの使い方の例です。あるオブジェクトの `int` フィールドを名前で指定して、それをインクリメントするというものです。

リスト 3. リフレクションによるフィールドのインクリメント

```
public int incrementField(String name, Object obj) throws... {
    Field field = obj.getClass().getDeclaredField(name);
    int value = field.getInt(obj) + 1;
    field.setInt(obj, value);
    return value;
}
```

このメソッドは、リフレクションによって柔軟なプログラミングができそうなことを示し始めています。`incrementField` は、特定のクラスを処理の対象とするのではなく、渡されてきたオブジェクトの `getClass` メソッドを使ってクラスの情報を入手し、そのクラスの中の名前で指定したフィールドを直接読み書きしています。

リフレクションによるメソッド

メソッド情報にアクセスするための `Class` のリフレクション呼び出しは、コンストラクターやフィールドのための呼び出しと非常によく似ています。

- `Method getMethod (String name, Class[] params)` -- 指定した名前のパブリック・メソッドをパラメーターの型を指定して取得する
- `Method[] getMethods()` -- そのクラスのすべてのパブリック・メソッドを取得する
- `Method getDeclaredMethod(String name, Class[] params)` -- そのクラスで宣言されているメソッドのうち指定した名前のものをパラメーターの型を指定して取得する
- `Method[] getDeclaredMethods()` -- そのクラスで宣言されているすべてのメソッドを取得する

フィールド呼び出しの場合と同様、最初の 2つの呼び出しは、祖先クラスから継承されたものであっても、そのクラスを通してアクセスされるパブリック・メソッドについての情報を返します。残りの 2つは、メソッドのアクセス・タイプには関係なく、そのクラスで直接宣言されたメソッドについての情報を返します。

これらの呼び出しで返されてくる `java.lang.reflect.Method` のインスタンスは、それを定義しているクラスのインスタンスに対するメソッドを呼び出すための `invoke` メソッドを定義しています。この `invoke` メソッドはクラス・インスタンスと呼び出しに使うパラメーター値の配列の 2個の引数をとります。

リスト 4 は、フィールドの例を一歩進めてメソッド・リフレクションの使い方を示した例です。このメソッドは、`get/set` メソッドを使って定義されている `int` 値の JavaBean プロパティーをインクリメントしています。たとえば、オブジェクトで `count` という整数値を扱うための `getCount/setCount` メソッドが定義されている場合、このメソッド呼び出しの `name` パラメーターに count を渡すことで、この値をインクリメントすることができます。

リスト 4. JavaBean プロパティーをリフレクションによってインクリメントする

```
public int incrementProperty(String name, Object obj) {
    String prop = Character.toUpperCase(name.charAt(0)) +
        name.substring(1);
    String mname = "get" + prop;
    Class[] types = new Class[] {};
    Method method = obj.getClass().getMethod(mname, types);
    Object result = method.invoke(obj, new Object[0]);
    int value = ((Integer)result).intValue() + 1;
    mname = "set" + prop;
    types = new Class[] { int.class };
    method = obj.getClass().getMethod(mname, types);
    method.invoke(obj, new Object[] { new Integer(value) });
    return value;
}
```

JavaBeans の約束ごとに従いプロパティー名の 1文字目を大文字にし、そのプロパティー名の前に `get` を付加することで読み出しメソッドの名前を作成し、同様に `set` を付加することで書き込みメソッドの名前を作成しています。JavaBeans の読み出しメソッドは値を返してくるだけであり、書き込みメソッドも値を指定するパラメーターを 1個とるだけですので、メソッドのパラメーターの型を指定することで型の一致を図っています。また、メソッドはパブリックでなければならない約束になっていますので、そのクラスで呼び出し可能なパブリック・メソッドを検索する形式を使用しています。

これは、リフレクションを使って基本型の値を渡している最初の例ですので、それがどんな動作をしているのかを確認しておくことにします。基本的な原則は簡単なことで、基本型の値を

渡す必要がある場合には必ずその基本型の代わりに対応するラッパー・クラスのインスタンス (`java.lang` パッケージで定義されている) を使用するようにします。呼び出しと戻り値の両方に対してこのようにします。したがって上の `get` メソッドの呼び出しでは、実際のプロパティーは `int` 値ですが、結果は`java.lang.Integer` ラッパーで得られるものとしています。

配列のリフレクション

配列は Java プログラミング言語ではオブジェクトとして扱われます。オブジェクトは、すべてクラスをもっています。配列を使う場合、他のオブジェクトの場合と同様、標準の `getClass` メソッドを使って配列のクラスを取得することができます。ただし既存のインスタンスを使わずに クラスを取得する方法は、他の種類のオブジェクトとは異なります。また、配列クラスを取得しても、それを使って直接行えることはあまりありません。通常のクラスでリフレクションによって可能になるコンストラクター・アクセスは配列では成り立ちませんし、配列にはアクセスできるフィールドもありません。配列オブジェクトに定義されているのは、ベースの `java.lang.Object` のメソッドだけです。

配列を特殊な方法で扱うには `java.lang.reflect.Array` クラスで提供されている静的メソッドのコレクションを使用します。このクラスのメソッドを使って、新しい配列の作成、配列オブジェクトの長さの取得、配列オブジェクトの添え字要素の値の読み書きを行うことができます。

リスト 5 は既存の配列のサイズを変更してしまう便利なメソッドの例です。このメソッドは、リフレクションを使って同じ型の新しい配列を作成し、古い配列のデータをすべてコピーした後、その新しい配列を返しています。

リスト 5. リフレクションによる配列の拡張

```
public Object growArray(Object array, int size) {
    Class type = array.getClass().getComponentType();
    Object grown = Array.newInstance(type, size);
    System.arraycopy(array, 0, grown, 0,
        Math.min(Array.getLength(array), size));
    return grown;
}
```

セキュリティーとリフレクション

セキュリティーは、リフレクションを扱う場合、複雑な問題を起こすことがあります。リフレクションはフレームワーク型のコードでよく使用されますが、その場合、通常のアクセス制限で行われるようなことは何も行わず、フレームワークがコードをまったく自由にアクセスできるようにしたいことがあります。しかし何ら制約のないアクセスを許すと、信頼できないコードと共有される環境でコードが実行される場合など、セキュリティーに大きなリスクをもたらす可能性があります。

このような相反するニーズがあるため、Java プログラミング言語はリフレクションのセキュリティーに対処するために多段階の手法を定義しています。基本モードは、ソース・コードのアクセスに適用されるのと同様の以下の制約をリフレクションにも強制するというものです。

- クラスのパブリック・コンポーネントはどこからでもアクセスできる
- プライベート・コンポーネントはそのクラス以外からはアクセスできない

- プロテクテッド・コンポーネントおよびパッケージ (デフォルト・アクセス) コンポーネントは制限付きでアクセスできる

しかしながら、これらの制約には簡単な逃げ道が存在します。少なくとも、そういう場合があります。これまでの例で使用してきた `Constructor`、`Field`、`Method` の各クラスは、すべて共通の基本クラスである `java.lang.reflect.AccessibleObject` クラスを拡張しています。このクラスは `setAccessible` メソッドを定義しており、このメソッドを使えば、これらのクラス (`Constructor`、`Field`、`Method`) のインスタンスのアクセス・チェックをオン/ オフすることができます。唯一捕捉できるとすればセキュリティー・マネージャーを稼働させている場合で、アクセス・チェックをオフにしようとするコードにそのための許可があるかどうかをチェックする場合です。許可がなければ、セキュリティー・マネージャーは例外をスローします。

リスト 6 は、リスト 1 の `TwoString` クラスのインスタンスにリフレクションを適用する場合のセキュリティー・マネージャーの働きを示すプログラム例です。

リスト 6. リフレクション・セキュリティーの動作

```
public class ReflectSecurity {
    public static void main(String[] args) {
        try {
            TwoString ts = new TwoString("a", "b");
            Field field = clas.getDeclaredField("m_s1");
//          field.setAccessible(true);
            System.out.println("Retrieved value is " +
                field.get(inst));
        } catch (Exception ex) {
            ex.printStackTrace(System.out);
        }
    }
}
```

このコードをコンパイルして何も特別なパラメーターを付けずにコマンド・ラインから直接実行すると、`field.get(inst)` 呼び出しで `IllegalAccessException` がスローされます。このコードから `field.setAccessible(true)` の行のコメントを外しコンパイルし直して実行すると、今度はうまく実行されます。最後に、セキュリティー・マネージャーを有効にするための JVM のパラメーター `-Djava.security.manager` をコマンド・ラインに付加すると、`ReflectSecurity` クラスの許可を定義しないかぎり、コードは再びうまく実行されなくなります。

リフレクションの性能

リフレクションは強力なツールなのですが、いくつか欠点もあります。主な欠点の 1つは性能に及ぼす影響です。リフレクションの処理は基本的にインタープリター型であり、何を行いたいのかを JVM に伝えると JVM がそれを行ってくれるという形式のものです。このような種類の処理は、同じ処理を直接行う場合よりも遅くなるのが常です。リフレクションを利用する場合の性能のコストを実証するために、本稿用に一連のベンチマーク・プログラムを用意しました (コード全体のリンクは参考文献参照)。

リスト 7 はフィールド・アクセスの性能テストの一部で、基本的なテスト・メソッドがいくつか含まれています。これらのメソッドはそれぞれフィールド・アクセスの形式を 1つずつテストします。`accessSame` は同じオブジェクトのメンバー・フィールドを、`accessOther` は直接アクセスされる別のオブジェクトのフィールドを、`accessReflection` はリフレクションでアクセスされる

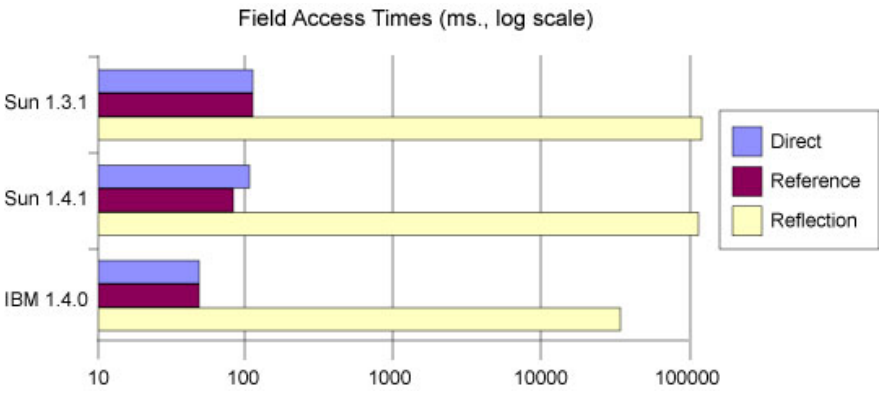
別のオブジェクトのフィールドをそれぞれ処理します。いずれの場合もメソッドは同じ計算を行い、ループの中で一連の簡単な加算/乗算を行います。

リスト 7. フィールド・アクセスの性能をテストするコード

```
public int accessSame(int loops) {
    m_value = 0;
    for (int index = 0; index < loops; index++) {
        m_value = (m_value + ADDITIVE_VALUE) *
            MULTIPLIER_VALUE;
    }
    return m_value;
}
public int accessReference(int loops) {
    TimingClass timing = new TimingClass();
    for (int index = 0; index < loops; index++) {
        timing.m_value = (timing.m_value + ADDITIVE_VALUE) *
            MULTIPLIER_VALUE;
    }
    return timing.m_value;
}
public int accessReflection(int loops) throws Exception {
    TimingClass timing = new TimingClass();
    try {
        Field field = TimingClass.class.
            getDeclaredField("m_value");
        for (int index = 0; index < loops; index++) {
            int value = (field.getInt(timing) +
                ADDITIVE_VALUE) * MULTIPLIER_VALUE;
            field.setInt(timing, value);
        }
        return timing.m_value;
    } catch (Exception ex) {
        System.out.println("Error using reflection");
        throw ex;
    }
}
```

このテスト・プログラムは、大きなループ・カウントを使ってそれぞれのメソッドを繰り返し呼び出し、何回かの呼び出しで測定された時間の平均を算出します。それぞれのメソッドを最初に呼び出すときの時間は平均には含まれませんので、初期化の時間は結果に影響を及ぼしません。本稿のテスト実行は 1GHz の PIIIm システムで行い、それぞれの呼び出しのループ・カウントは 1 千万回としました。Linux の 3種類の JVM についての時間結果を示したのが図 1 です。すべてのテストでそれぞれの JVM のデフォルト設定を使用しました。

図 1. フィールド・アクセス時間



グラフは対数目盛りにして時間の範囲全体をカバーするようにしたため、グラフでは差異が目立たなくなっています。図の最初の 2つ (いずれも Sun の JVM) では、リフレクションでの実行時間は直接アクセスによる場合の千倍以上になっています。それに比べ IBM の JVM では少し良い結果になっていますが、それでもリフレクション・メソッドは他のメソッドの 700倍以上の時間を要しています。どの JVM でも他の 2つのメソッドの間に大した時間的差はありませんが、IBM の JVM はこれらのメソッドを Sun の 2つの JVM の 2倍近い速度で実行しています。おそらく、この差は Sun の Hot Spot JVM が特殊な最適化を行っている影響なのではないかと思います。この最適化は簡単なベンチマークで性能を低下させる傾向があります。

フィールド・アクセス時間のテストの他に、メソッド呼び出しについても同様の所要時間のテストを行いました。メソッド呼び出しの場合、フィールド・アクセスの場合と同様、3つのアクセス形式を試し、それぞれ引数なしのメソッドを使う場合とメソッド呼び出しで値の渡しと戻しを行う場合をテストしました。リスト 8 は値の渡し/戻しの形式の呼び出しをテストするための 3つのメソッドを示すコードです。

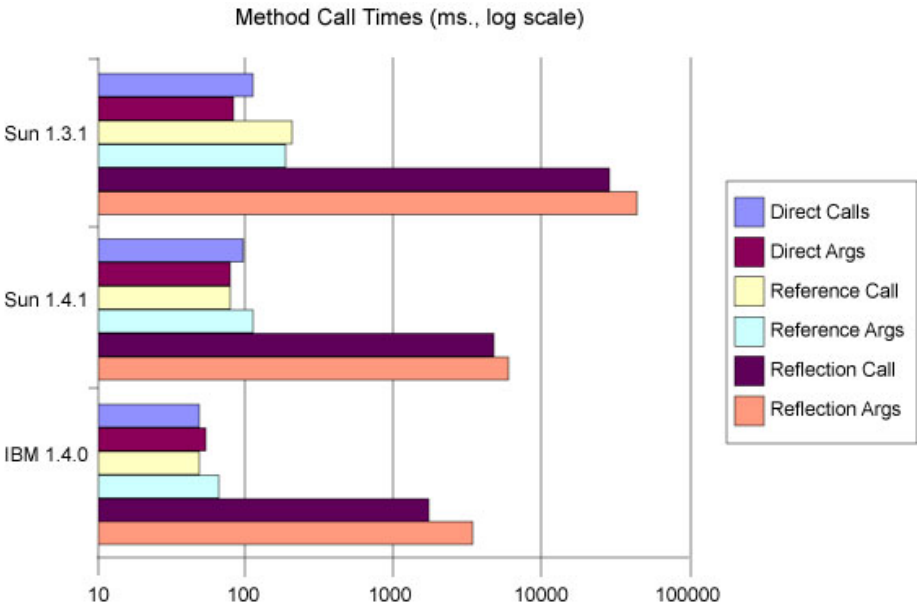
リスト 8. メソッド・アクセスの性能をテストするコード

```
public int callDirectArgs(int loops) {
    int value = 0;
    for (int index = 0; index < loops; index++) {
        value = step(value);
    }
    return value;
}
public int callReferenceArgs(int loops) {
    TimingClass timing = new TimingClass();
    int value = 0;
    for (int index = 0; index < loops; index++) {
        value = timing.step(value);
    }
    return value;
}
public int callReflectArgs(int loops) throws Exception {
    TimingClass timing = new TimingClass();
    try {
        Method method = TimingClass.class.getMethod
            ("step", new Class [] { int.class });
        Object[] args = new Object[1];
        Object value = new Integer(0);
        for (int index = 0; index < loops; index++) {
            args[0] = value;
            value = method.invoke(timing, args);
        }
        return ((Integer)value).intValue();
    } catch (Exception ex) {
        System.out.println("Error using reflection");
        throw ex;
    }
}
```

図 2 はメソッド呼び出しの時間結果を示したものです。ここでもリフレクションは直接的な方法よりも随分遅くなっています。ただし、差はフィールド・アクセスの場合ほど大きくはなく、Sun 1.3.1 の JVM での数百倍から IBM の JVM での引数なしの場合の 30倍以下までの範囲に収まっています。引数を使う場合のリフレクションによるメソッド呼び出しのテスト性能は、すべての JVM で引数なしの呼び出しの場合に比べかなり遅くなっています。これは多分 `int` 値の渡しと戻しに `java.lang.Integer` ラッパーを必要としていることも理由の 1つに挙げられるでしょう。`Integer`

は変更できないため、メソッドからのリターンがあるたびに新しい `Integer` を生成する必要があり、これが大きなオーバーヘッドになっています。

図 2. メソッド呼び出しの時間



リフレクションの性能は Sun が 1.4 JVM の開発で主眼とした目標の 1つであり、それがリフレクションによるメソッド呼び出しの結果に表れています。Sun の 1.4.1 JVM は、この種の処理に関してバージョン 1.3.1 よりもかなり性能を向上させており、私のテストでは約 7倍高速になっています。といってもこの単純なテストでは、ここでも IBM の 1.4.0 JVM のほうが高い性能を示しており、Sun の 1.4.1 JVM よりも 2、3倍高速に処理を行っています。

さらに、リフレクションを使ってオブジェクトを作成する場合の所要時間を計る同様のテスト・プログラムも記述しました。しかし、このテストの場合、フィールド呼び出しやメソッド呼び出しほどには大きな違いは出ていません。`newInstance()` 呼び出しを使って単純な `java.lang.Object` のインスタンスを作成する場合、`new Object()` による方法に比べ Sun 1.3.1 JVM で 12倍、IBM 1.4.0 JVM で約 4倍長い時間がかかり、Sun 1.4.1 JVM では約 2倍しか違いません。`Array.newInstance(type, size)` で配列を作成する場合、`new type[size]` による方法に比べ、どの JVM でも最大で約 2倍長い時間がかかり、配列のサイズが大きくなるにつれその差は小さくなります。

リフレクションのまとめ

Java 言語のリフレクションは、プログラム・コンポーネントを動的にリンクしたい場合にいろいろと便利な方法を提供してくれます。リフレクションを利用することで、前もって対象となるクラスをハードコードしなくても (セキュリティーに関する制約に従いつつ) プログラムによって任意のクラスのオブジェクトを作成したり操作できるようになります。こうした特徴からリフレクションは、オブジェクトを非常に一般化した方法で扱うためのライブラリーの作成に特に有効です。たとえばリフレクションはオブジェクトをデータベースや XML などの外部フォーマットに永続保存するフレームワークによく使用されます。

リフレクションには欠点もいくつかあります。1つは性能の問題です。リフレクションは、フィールド・アクセスやメソッド・アクセスに使用される場合、直接的なコードよりも格段に遅くなります。それがどの程度の問題となるかは、プログラムでのリフレクションの使い方によります。プログラムの処理の中でも実行される機会の比較的少ない部分に使用されるのであれば、その性能の遅さが問題となることはありません。私の行ったテストでは、最悪の場合でもリフレクション処理に数マイクロ秒しかかかっていないことが示されました。性能の問題は、性能を命とするアプリケーションの中核のロジックにリフレクションが使用される場合にしか重大な意味をもちません。

多くのアプリケーションにとってそれより重大な欠点は、リフレクションを利用することで何を行っているコードなのかがわかり難くなることです。プログラマーは、プログラムのロジックをソース・コードで理解できるものと考えますので、リフレクションなどソース・コードを無視する手法は保守上の問題を引き起こす可能性があります。またリフレクション・コードは、性能比較のためのサンプル・コードに見られるように、同等の直接的なコードよりも複雑になります。これらの問題に対処するための最善の方法は、リフレクションをいたずらに使用せず (本当に柔軟性を得るのに役立つ場面でのみ使用する)、対象とするクラスの中にその使い方を記述することです。

次回は、リフレクションの使い方の例をさらに詳しく紹介したいと思います。その例では Java アプリケーションに対するコマンド・ライン引数を処理するための API が用意されます。皆さん自身のアプリケーションでも役立ちそうなツールです。またリフレクションの長所を活用しつつ弱点は回避する例ともなっています。リフレクションはコマンド・ライン処理の簡略化に役立つのでしょうか。Java プログラミングのダイナミックス の第 3 回にご期待ください。

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。