Javaの理論と実践: 動的プロキシーで装飾する

デコレーターやアダプターの構築には動的プロキシーが便利

Brian Goetz
Principal Consultant
Quiotix

2005年 8月 30日

動的プロキシー機能（dynamic proxy facility）はjava.lang.reflectパッケージの一部であり、JDKのバージョン1.3に追加されたものですが、この機能を利用すると、プログラムでプロキシー・オブジェクトを作ることができます。プロキシー・オブジェクトは、1つまたはそれ以上の既知のインターフェースを実装することができ、（組み込みの仮想メソッド・ディスパッチは使わず）リフレクションを使って、インターフェース・メソッドへのコールをプログラム的にディスパッチすることができます。このプロセスを利用すると、実装でメソッド・コールを「インターセプト（intercept）」して転送したり、動的に機能を追加したりすることができます。今回は、Brian Goetzが、動的プロキシーに関する幾つかのアプリケーションを調べます。

このシリーズの他の記事を見る

一般的な設計パターンには、ファサード（Facade）やブリッジ、インターセプター（Interceptor）、デコレーター（Decorator）、プロキシー（リモート・プロキシーや仮想プロキシーを含みます）、アダプターなどのパターンがありますが、動的プロキシーは、こうした一般的な設計パターンを実装するための、動的な代替機構を提供しています。動的なプロキシーを使わなくても、これらのパターンはすべて通常のクラスを使って実装することができますが、多くの場合、動的なプロキシーによる方法の方が便利で簡潔であり、大量のクラスを手書きしたり生成したりする必要がなくなります。

プロキシー・パターン

プロキシー・パターンでは、「スタブ（stub）」あるいは「代理（surrogate）」オブジェクトを作ります。これらのオブジェクトの目的は、リクエストを受け付け、実際の作業を行う別のオブジェクトにそのリクエストを転送することです。プロキシー・パターンが使われるのは、RMI（Remote Method Invocation）で別のJVMで実行するオブジェクトをローカル・オブジェクトのように見せる場合や、EJB（Enterprise JavaBeans）でリモート呼び出しやセキュリティー、トランザクション区分（transaction demarcation）を追加する場合、JAX-RPC Webサービスでリモート・サービスをローカル・オブジェクトのように見せる場合などです。それぞれの場合において、リモートである可能性のあるオブジェクトの振る舞いは、インターフェースが定義しますが、インターフェースはその性格上、複数の実装を許します。呼び出し側は（大部分の場合）、

自分が保持しているのはスタブへの参照のみであって実際のオブジェクトではない、ということが分かりません。これは、参照も実際のオブジェクトも同じインターフェースを実装しているためです。つまりスタブは本当のオブジェクトを見つけ、引数をマーシャリング（marshalling）して本当のオブジェクトに送り、戻り値をアンマーシャリング（unmarshalling）して呼び出し側に返します。プロキシーを使うのは、リモート化（remoting）を提供する場合（RMIやEJB、そしてJAX-RPCなど）や、セキュリティー・ポリシーでオブジェクトをラップする場合（EJB）、高価なオブジェクトを遅延ロードする場合（EJB Entity Beans）、あるいはロギングなどの実装を追加する場合などです。

5.0以前のJDKでは、RMIスタブは（そしてそれに対応するスケルトンは）、JDKツールセットの一部であるRMIコンパイラー（rmic）がコンパイル時に生成するクラスでした。リモート・インターフェースのそれぞれに対して、リモート・オブジェクトのふりをするスタブ（プロキシー）クラスが生成されますが、同時にスケルトン・オブジェクトも生成されます。スケルトン・オブジェクトは、リモートJVMではスタブと逆のことをし、引数をアンマーシャリングして本当のオブジェクトを呼び出します。同じように、Webサービス用のJAX-RPCツールは、ローカル・オブジェクトのように見える、リモートWebサービス用のプロキシー・クラスを生成します。

生成されるスタブ・クラスが、ソースコードとして生成されるにせよ、バイトコードとして生成されるにせよ、コンパイル・プロセスにとってコード生成は相変わらず追加的なステップであり、同じような名前のクラスがあちこちにあるため混乱の元になります。一方、動的プロキシーの機構では、コンパイル時にスタブ・クラスは生成せず、実行時にプロキシー・オブジェクトを生成します。JDK 5.0以上のRMI機能では、生成されるスタブではなく動的プロキシーを使うため、RMIが使いやすくなります。また多くのJ2EEコンテナーでも、EJBを実装するために動的プロキシーを使います。EJB技術はセキュリティーやトランザクション区切りの実装に関して、インターセプトに大きく依存していますが、動的プロキシーを利用すると、あるインターフェースに対して呼び出される全メソッドに対して中央制御のフロー・パスが提供されるため、インターセプトの実装が単純になるのです。

動的プロキシーの機構

動的プロキシーの機構の核心部分が、InvocationHandlerインターフェースです（リスト1）。呼び出しハンドラーの仕事は、要求されたメソッド呼び出しを、動的プロキシーに代わって実際に実行することです。呼び出しハンドラーには、（java.lang.reflectパッケージから）Methodオブジェクトと、そのメソッドに渡すべき引数のリストが渡されます。最も単純な場合では、呼び出しハンドラーはリフレクションのメソッドMethod.invoke() を呼び、その結果を返します。

リスト1. InvocationHandlerインターフェース

```
public interface InvocationHandler {
    Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

すべてのプロキシーには呼び出しハンドラーが関連付けられており、このハンドラーは、そのプロキシーのメソッドの1つが呼ばれる度に呼ばれます。一般的な設計の原則、つまりインターフェースはタイプを定義するためのもの、クラスは実装を定義するためのもの、という原則に従い、プロキシー・オブジェクトは1つ以上のインターフェースを実装できますが、クラスを実装することはできません。プロキシー・クラスにはアクセスするための名前がないため、コンストラ

クターを持つことができません。従って、プロキシー・クラスはファクトリーで作るしかありません。リスト2は、動的プロキシーの最も単純な実装です。このプロキシーはSetインターフェースを実装し、カプセル化されたSetインスタンスに対して、全てのSetメソッド (と、全てのObjectメソッド) をディスパッチします。

リスト2. Setをラップする単純な動的プロキシー

```
public class SetProxyFactory {

    public static Set getSetProxy(final Set s) {
        return (Set) Proxy.newProxyInstance
          (s.getClass().getClassLoader(),
                new Class[] { Set.class },
                new InvocationHandler() {
                    public Object invoke(Object proxy, Method method,
                      Object[] args) throws Throwable {
                        return method.invoke(s, args);
                    }
                });
    }
}
```

SetProxyFactoryクラスには、1つの静的ファクトリー・メソッド、getSetProxy() が含まれており、このメソッドが、Setを実装する動的プロキシーを返します。このプロキシー・オブジェクトは、実際にSetを実装しますが、呼び出し側は、 (リフレクションによる場合を除いて) 返ってきたオブジェクトが動的プロキシーであることを区別できません。SetProxyFactoryが返すプロキシーは、このファクトリー・メソッドに渡されたSetインスタンスにメソッドをディスパッチする以外、何もしません。リフレクション・コードは読みにくい場合が多いのですが、ここでは僅かなことしかしていないので、制御フローを追うのは難しくありません。あるメソッドがSetプロキシーに対して呼び出されると、そのメソッドは呼び出しハンドラーにディスパッチされます。呼び出しハンドラーは単純に、下にあるラップされたオブジェクトに対して、要求されたメソッドをリフレクションで呼び出すのです。当然のことですが、全く何もしないプロキシーなんて馬鹿げています。いや、本当にそうでしょうか。

何もしないアダプター

実際には、SetProxyFactoryのような、何もしないラッパーにも充分な使い道があります。つまり、オブジェクト参照を、安全に、特定なインターフェースだけに絞り込むために利用できるのです。そうすれば、呼び出し側は参照をアップ・キャスト(upcast)できず、プラグインやコールバックなど信頼できないコードに対して、オブジェクト参照を安全に渡せるようになります。リスト3には、典型的なコールバック・シナリオを実装するための、一連のクラス定義が含まれています。これを見ると、通常は手動で (あるいは、IDEが提供するコード生成ウィザードによって) 実装されるアダプター・パターンを、動的プロキシーで手軽に置き換えられることが分かると思います。

リスト3. 典型的なコールバック・シナリオ

```
public interface ServiceCallback {
    public void doCallback();
}

public interface Service {
    public void serviceMethod(ServiceCallback callback);
}

public class ServiceConsumer implements ServiceCallback {
    private Service service;

    ...
    public void someMethod() {
        ...
        service.serviceMethod(this);
    }
}
```

ServiceConsumerクラスはServiceCallback（多くの場合、コールバックをサポートする便利な方法です）を実装し、コールバック参照としてthis参照をserviceMethod() に渡します。ところがこの方法では、Service実装がServiceCallbackを、ServiceConsumerや、ServiceConsumerがServiceに呼ばせようとは思っていない呼び出し側メソッドにアップ・キャストできてしまいます。この危険性を気にしない場合もあるかもしれません。しかし気にする場合には、コールバック・オブジェクトを内部クラスにするか、あるいは何もしないアダプター・クラスを書き（リスト4のServiceCallbackAdapterを見てください）、ServiceConsumerをServiceCallbackAdapterでラップします。ServiceCallbackAdapterは、ServiceがServiceConsumerに対してServiceCallbackをアップ・キャストするのを防ぎます。

リスト4. インターフェースに対するオブジェクトを安全に絞り込み、悪意のコードによるアップ・キャストを防ぐアダプター・クラス

```
public class ServiceCallbackAdapter implements ServiceCallback {
    private final ServiceCallback cb;

    public ServiceCallbackAdapter(ServiceCallback cb) {
        this.cb = cb;
    }

    public void doCallback() {
        cb.doCallback();
    }
}
```

ServiceCallbackAdapterのようなアダプター・クラスを書くのは単純ですが、退屈な作業です。ラップされたインターフェースの中のメソッドそれぞれに対して、転送メソッドを書かなければなりません。ServiceCallbackの場合、実装すべきメソッドは1つしかありませんでした。しかしCollectionsやJDBCインターフェースでは、何十ものメソッドがあります。最近のIDEでは、アダプターを書くために必要な作業量を減らすために、「委譲メソッド（Delegate Method）」ウィザードを提供していますが、ラップしたいインターフェースそれぞれに対して相変わらず1つのアダプター・クラスを書かなければならず、しかも、生成されたコードしか含まないクラスには、何となく不満が残るものです。「何もしない絞り込みアダプター・パターン」を、もっと簡潔に表現する方法があるはずです。

汎用アダプター・クラス

リスト2のSetProxyFactoryクラスは、これに等価な、Setインターフェース用のアダプター・クラスよりも確かに簡潔ですが、Setという、1つのインターフェースに対してしか使えないことは同じです。ところがジェネリックス（generics）を使うと、任意のインターフェースと同じことをする汎用プロキシー・ファクトリーを容易に作成できるのです。これをリスト5に示します。これはSetProxyFactoryとほとんど同じですが、どんなインターフェースでも動作します。これで、絞り込み用のアダプター・クラスを再度書く必要がなくなります。インターフェースTへのオブジェクトを安全に絞り込むプロキシー・オブジェクトを書きたい場合には、単純にgetProxy(T.class,object) を呼び出せば、それで終わりです。山のようなアダプター・クラスを別途用意する必要はありません。

リスト5. 汎用の、絞り込みアダプター・ファクトリー・クラス

```
public class GenericProxyFactory {

    public static<T> T getProxy(Class<T> intf,
      final T obj) {
        return (T)
          Proxy.newProxyInstance(obj.getClass().getClassLoader(),
                new Class[] { intf },
                new InvocationHandler() {
                    public Object invoke(Object proxy, Method method,
                      Object[] args) throws Throwable {
                        return method.invoke(obj, args);
                    }
                });
    }
}
```

デコレーター（Decorators）としての動的プロキシー

当然のことですが、動的プロキシー機能は、特定なインターフェースに対して単純にオブジェクト・タイプを絞り込む他にも、はるかに多くのことができます。リスト2やリスト5にある、単純な絞り込みアダプターとデコレーター・パターン（Decorator pattern）との距離は、ごく僅かです。デコレーター・パターンでは、プロキシーは、セキュリティー・チェックやロギングなどの追加機能で呼び出しをラップします。リスト6は、ログのInvocationHandlerを示しています。これは、対象となるターゲット・オブジェクトに対して単純にメソッドを呼び出すことに加えて、呼び出されたメソッドや渡された引数、戻り値などを示すログ・メッセージを書き出します。リフレクションのinvoke() コールを除いて、ここにあるコードは全て、単純にデバッグ・メッセージ生成の一部であり、他にはほとんど何もありません。プロキシー・ファクトリー・メソッド用のコードは、匿名の呼び出しハンドラーの代わりにLoggingInvocationHandlerを使うことを除くと、GenericProxyFactoryとほとんど同じです。

リスト6. 各メソッド・コールに対してデバッグ・ログを生成する、プロキシー・ベースのデコレーター

```
private static class LoggingInvocationHandler<T>
      implements InvocationHandler {
        final T underlying;

        public LoggingHandler(T underlying) {
            this.underlying = underlying;
        }
```

```
        public Object invoke(Object proxy, Method method,
          Object[] args) throws Throwable {
            StringBuffer sb = new StringBuffer();
            sb.append(method.getName()); sb.append("(");
            for (int i=0; args != null && i<args.length; i++) {
                if (i != 0)
                    sb.append(", ");
                sb.append(args[i]);
            }
            sb.append(")");
            Object ret = method.invoke(underlying, args);
            if (ret != null) {
                sb.append(" -> "); sb.append(ret);
            }
            System.out.println(sb);
            return ret;
        }
    }
```

HashSetをロギング・プロキシーでラップし、次のような単純なテスト・プログラムを実行してみます。

```
    Set s = newLoggingProxy(Set.class, new HashSet());
    s.add("three");
    if (!s.contains("four"))
        s.add("four");
    System.out.println(s);
```

そうすると、次のような出力が得られます。

```
  add(three) -> true
  contains(four) -> false
  add(four) -> true
  toString() -> [four, three]
  [four, three]
```

この方法は、オブジェクトの周りにデバッグ・ラッパーを追加する方法としては、なかなか良く、また容易です。こちらの方が、代行クラスを生成し、手動で大量のprintln() ステートメントを作るよりも、ずっと容易（しかも汎用的）です。これを、さらに押し進めることができます。デバッグ出力を無条件に生成するのではなく、動的コンフィギュレーション・ストアー（コンフィギュレーション・ファイルで初期化され、JMX MBeanで動的に変更されます）を調べ、実際にデバッグ・ステートメントを生成すべきかどうかを、クラス毎やインスタンス毎に判断するようにもできるのです。

この時点で、AOPのファンの読者は、「そのためにAOPがあるではないか！」と吹き出すかも知れません。確かにその通りですが、与えられた問題を解決する方法は1つではありません。ある技術が問題を解決できるからといって、それが最良の解決法とは限りません。いずれにせよ、動的プロキシーによる方法は、完全に「純粋なJava」の範囲内で動作する、という利点があります。AOPはどこでも使われているわけではなく、どこでも使うべきものでもありません。

アダプターとしての動的プロキシー

プロキシーは、本当のアダプターとして使うこともでき、下にあるオブジェクトが実装するものとは異なるインターフェースをエクスポートする、オブジェクトのビューを提供します。呼び出しハンドラーは、下にある同じオブジェクトに対して、あらゆるメソッド・コールをディスパッ

チする必要はありません。名前を調べ、異なるメソッドには異なるオブジェクトをディスパッチすればよいのです。例えば、プロパティーに対してセッターとゲッターを規定した永続的実体（PersonやCompany、PurchaseOrderなど）を表すJavaBeansインターフェース・セットがあるとしましょう。そして、こうしたインターフェースを実装するオブジェクトに対してデータベース・レコードをマップする、永続性レイヤー（persistence layer）を書くとします。各インターフェースに対してクラスを書いたり生成したりする代わりに、汎用のJavaBeans風プロキシー・クラスを1つ使い、プロパティーをMapに保存する方法があるのです。

リスト7は、動的プロキシーの例です。このプロキシーは、呼ばれたメソッドの名前を調べ、プロパティー・マップを調べたり修正したりすることによって、ゲッター・メソッドとセッター・メソッドを直接実装します。この1つのプロキシー・クラスによって、複数のJavaBeans風インターフェースのオブジェクトを実装できるのです。

リスト7. Mapにゲッターとセッターをディスパッチする動的プロキシー・クラス

```
public class JavaBeanProxyFactory {
    private static class JavaBeanProxy implements InvocationHandler {
        Map<String, Object> properties = new HashMap<String,
          Object>();

        public JavaBeanProxy(Map<String, Object> properties) {
            this.properties.putAll(properties);
        }

        public Object invoke(Object proxy, Method method,
          Object[] args)
          throws Throwable {
            String meth = method.getName();
            if (meth.startsWith("get")) {
                String prop = meth.substring(3);
                Object o = properties.get(prop);
                if (o != null && !method.getReturnType().isInstance(o))
                    throw new ClassCastException(o.getClass().getName() +
                      " is not a " + method.getReturnType().getName());
                return o;
            }
            else if (meth.startsWith("set")) {
                // Dispatch setters similarly
            }
            else if (meth.startsWith("is")) {
                // Alternate version of get for boolean properties
            }
            else {
                // Can dispatch non get/set/is methods as desired
            }
        }
    }

    public static<T> T getProxy(Class<T> intf,
      Map<String, Object> values) {
        return (T) Proxy.newProxyInstance
          (JavaBeanProxyFactory.class.getClassLoader(),
                new Class[] { intf }, new JavaBeanProxy(values));
    }
}
```

リフレクションはObjectに関して動作するため、タイプセーフが失われる危険性が少しありますが、JavaBeanProxyFactoryでのゲッター処理は、ここで私がゲッターに対してisInstance() チェックをしているのと同じように、必要な追加タイプ・チェックを「焼き込んで」います。

パフォーマンス・コスト

ご覧の通り、動的プロキシーを利用すると、多くのコードを単純化できる可能性があります。生成される大量のコードを動的プロキシーによって置き換えられるだけではなく、1つのプロキシー・クラスによって、手書きあるいは生成されたコードによる複数のクラスを置き換えられるのです。ではコストはどうなのでしょう。動的プロキシーでは、組み込みの仮想メソッド・ディスパッチを使わずにリフレクションでメソッドをディスパッチするため、恐らく多少のパフォーマンス・コストが発生します。ごく初期のJDKでは、リフレクションのパフォーマンスは貧弱でした（初期のJDKでは、それ以外のパフォーマンスも、ほとんど全て貧弱でしたが）。しかし、ここ10年ほどの間に、リフレクションは大幅に高速化されています。

私はベンチマーク構築にまで入り込まずに、非科学的ですが単純なテスト・プログラムを書いてみました。Setにデータを詰め込み、Setに要素をランダムに挿入し、参照し、削除するというループです。これを、3つのSet実装で実行しました。飾り気のないHashSet、全てのメソッドを下にあるHashSetに単純に転送する手書きのSetアダプター、そして、これも全てのメソッドを下にあるHashSetに単純に転送する、プロキシー・ベースのSetアダプター、とう3つです。それぞれのループ繰り返しは、幾つかのランダムな数を生成し、1つ以上のSet操作を行いました。手書きのアダプターは、生のHashSetに比べて、ほんの数パーセントのパフォーマンス・オーバーヘッドが生じただけです（これは恐らく、JVMレベルでの効果的なインライン・キャッシングと、ハードウェア・レベルでの分岐予測によるものです）。プロキシー・アダプターは生のHashSetよりも大分遅いのですが、オーバーヘッドは2倍にまではなりませんでした。

この実験から私が得た結論は、ほとんどの場合において、たとえ軽量のメソッドに対する場合であっても、プロキシーによる方法で十分うまく行く、ということです。そして、プロキシーとして代行する操作が重量級になるにつれ（つまり、リモート・メソッド・コールやシリアル化を使用するメソッド、IOを行う、あるいはデータベースからデータをフェッチするなど）、プロキシーによるオーバーヘッドは、実質的にゼロに近づきます。もちろん、プロキシーによる方法を使用した場合、受け入れがたいオーバーヘッドを生ずる場合もあり得ますが、そうした場合はごく稀でしょう。

まとめ

プロキシーやデコレーター、アダプターなど、多くの設計パターンを実装する上で、動的プロキシーは強力なツールですが、まだ充分に利用されていません。こうした設計パターンをプロキシー・ベースで実装すると、書きやすい上に変になりにくく、しかも汎用性が高まります。多くの場合、1つの動的プロキシー・クラスで全インターフェースに対するデコレーターやプロキシーの役割を果たすことができ、それぞれのインターフェースに対して静的クラスを書く必要が無くなります。非常にパフォーマンスを重視するアプリケーションを除いて、手書きや機械生成のスタブによる方法よりも、動的プロキシーによる方法が好ましいと言えるでしょう。

著者について

Brian Goetz

Brian Goetz は18 年間以上に渡って、専門的ソフトウェア開発者として働いています。彼はカリフォルニア州ロスアルトスにあるソフトウェア開発コンサルティング会社、Quiotixの主席コンサルタントであり、またいくつかのJCP Expert Groupの一員でもあります。2005年の末にはAddison-Wesleyから、Brianによる著、Java Concurrency In Practiceが出版される予定です。Brian著による有力業界紙に掲載済みおよび掲載予定の記事のリストを参照してください。