今まで知らなかった 5 つの事項: Java オブジェクトをシリアライズする場合

シリアライズされたデータは安全だと考えているなら、考え直す必要があります

Ted Neward
Principal
Neward & Associates

2010年 4月 08日
(初版 2010年 4月 06日)

Java オブジェクトのシリアライズは Java プログラミングではあまりにも基本的なため、当たり前のものとして軽く考えてしまいがちです。しかし Java プラットフォームの多くの側面と同様、シリアライズの成果を得るためにはシリアライズについて深く理解する必要があります。この新しい連載の第 1 回である今回は、Ted Neward が、Java オブジェクトのシリアライズ API を改めて見直す必要がある 5 つの理由を挙げながら、シリアライズされたデータのリファクタリング、暗号化、検証を行う上での秘訣 (そしてそのためのコード) を説明します。

このシリーズの他の記事を見る

数年前、私は Java 言語でアプリケーションを作成するソフトウェア・チームで働いていた際、平均的なプログラマーよりも少し詳しく Java オブジェクトのシリアライズに関して知っていると役立つという経験しました。

1 年ほど前、ユーザーごとのアプリケーション設定を管理していた開発者が、それらの設定を `Hashtable` に保存し、その `Hashtable` をシリアライズしてディスクに永続化することにしました。ユーザーが自分の設定を変更すると、その変更が反映された `Hashtable` が単純にディスクに書き戻されます。

これはスマートで制約のない設定システムでしたが、このチームが `Hashtable` から Java コレクション・ライブラリーの `HashMap` に移行することを決めると、このシステムは使いものにならなくなったのです。

`Hashtable` と `HashMap` とでは、ディスクに書き込まれる形式は異なり、互換性がありません。永続化されたユーザー設定のそれぞれに対して何らかのデータ変換ユーティリティーを実行しない限り (これは途方もない作業です)、そのアプリケーションを使い続ける間はストレージ・フォーマットを `Hashtable` のまま維持する必要があるように思えました。

この連載について

皆さんは自分が Java プログラミングについて知っていると思うかもしれません。しかし実際には、ほとんどの開発者は Java プラットフォームの表面的な部分しか扱っておらず、当面の作業を完了するために十分なことしか学んでいません。この連載では、Ted Neward が Java プラットフォームのコア機能を深く掘り下げ、非常に厄介な Java プログラミングの難題の解決にも役立つ、ほとんど知られていない事実を紹介します。

このチームは解決策がないと思っていましたが、それは彼らが Java のシリアライズに関する重要な (そしてあまり知られていない) 事実を知らなかったからにすぎません。その事実とは、Java のシリアライズは、あとになって型を変更できるように作られているということです。シリアライズを自動置換する方法を彼らに示すと、`HashMap` への移行は計画どおりに進みました。

この連載では、Java プラットフォームに関する有用な雑学的知識、つまりJava プログラミングでの難題を解決する上で役立つ、あまり知られていない事実を紹介します。今回はその第 1 回です。

Java オブジェクトのシリアライズ API は、初期の Java の時代 (JDK 1.1) から存在していたものであるため、最初に取り上げるのにふさわしい内容です。この記事でシリアライズに関する 5 つの事項を学ぶことで、標準的な Java API を扱う際にも新たな目で見る必要があることに皆さんは気付くはずです。

Java のシリアライズの基本

Java オブジェクトのシリアライズは JDK 1.1 を構成する画期的な機能セットの一部として導入されたものであり、Java のオブジェクト・グラフをストレージや送信用のバイト配列に変換するためのメカニズムとして機能します。そうしたバイト配列は、後で Java のオブジェクト・グラフに逆変換することができます。

要するにシリアライズの概念というのは、オブジェクト・グラフを「凍結」して (ディスクに、あるいはネットワーク経由で、あるいは他の手段で) 移動し、そして再度そのグラフを逆に「解凍」して有効な Java オブジェクトにする、ということです。これらはすべて、ほとんど魔法のように行われますが、それは `ObjectInputStream` クラスや `ObjectOutputStream` クラス、そして完全に忠実なメタデータのおかげであり、さらにはプログラマーがこのプロセスを選択してクラスを `Serializable` マーカー・インターフェースでタグ付けしてくれるおかげです。

リスト 1 は `Serializable` を実装した `Person` クラスを示しています。

リスト 1. Serializable を実装した Person クラス

```
package com.tedneward;

public class Person
    implements java.io.Serializable
{
    public Person(String fn, String ln, int a)
    {
        this.firstName = fn; this.lastName = ln; this.age = a;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
```

```
    public int getAge() { return age; }
    public Person getSpouse() { return spouse; }

    public void setFirstName(String value) { firstName = value; }
    public void setLastName(String value) { lastName = value; }
    public void setAge(int value) { age = value; }
    public void setSpouse(Person value) { spouse = value; }

    public String toString()
    {
        return "[Person: firstName=" + firstName +
            " lastName=" + lastName +
            " age=" + age +
            " spouse=" + spouse.getFirstName() +
            "]";
    }

    private String firstName;
    private String lastName;
    private int age;
    private Person spouse;

}
```

いったん `Person` クラスがシリアライズされると、ディスクにオブジェクト・グラフを書き込んだり、逆にディスクからオブジェクト・グラフを読み取ったりすることは非常に簡単です。これは下記の JUnit 4 によるユニット・テストを見るとわかります。

リスト 2. Person をデシリアライズする

```
public class SerTest
{
    @Test public void serializeToDisk()
    {
        try
        {
            com.tedneward.Person ted = new com.tedneward.Person("Ted", "Neward", 39);
            com.tedneward.Person charl = new com.tedneward.Person("Charlotte",
                "Neward", 38);

            ted.setSpouse(charl); charl.setSpouse(ted);

            FileOutputStream fos = new FileOutputStream("tempdata.ser");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(ted);
            oos.close();
        }
        catch (Exception ex)
        {
            fail("Exception thrown during test: " + ex.toString());
        }

        try
        {
            FileInputStream fis = new FileInputStream("tempdata.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            com.tedneward.Person ted = (com.tedneward.Person) ois.readObject();
            ois.close();

            assertEquals(ted.getFirstName(), "Ted");
            assertEquals(ted.getSpouse().getFirstName(), "Charlotte");

            // Clean up the file
            new File("tempdata.ser").delete();
        }
```

```
        catch (Exception ex)
        {
            fail("Exception thrown during test: " + ex.toString());
        }
    }
}
```

ここまでに説明したことは新しいことでも画期的なことでもなく、シリアライズの基本にすぎませんが、出発点としては適切です。ここでは `Person` を使用して、おそらく皆さんが Java オブジェクトのシリアライズに関して今まで知らなかった 5 つの事項を紹介します。

1. シリアライズによって生成された出力はリファクタリングすることが可能である

シリアライズによって生成された出力では、ある程度クラスを変更することができます。例えばクラスをリファクタリングした場合でも、そのクラスを `ObjectInputStream` によって読み取ることができます。

Java オブジェクトをシリアライズして生成された出力のなかで行える重要なこととしては、以下の操作が挙げられます。

- クラスへの新しいフィールドの追加
- static で修飾されたフィールドから static が付けられていないフィールドへの変更
- transient で修飾されたフィールドから transient が付けられていないフィールドへの変更

逆の場合 (static が付けられていないフィールドから static で修飾されたフィールドへの変更、あるいは transient が付けられていないフィールドからtransient で修飾されたフィールドへの変更) やフィールドを削除する場合には、どの程度の後方互換性を要求するかにより、追加の変更を行う必要があります。

シリアライズされたクラスをリファクタリングする

シリアライズによって生成された出力はリファクタリングすることが可能であるということがわかったので、`Person` クラスに新しいフィールドを追加した場合に何が起こるのかを調べてみましょう。

`PersonV2` (リスト 3) では、最初の `Person` クラスに性別のフィールド (gender) を追加しています。

リスト 3. シリアライズされた Person クラスに新しいフィールドを追加する

```
enum Gender
{
    MALE, FEMALE
}

public class Person
    implements java.io.Serializable
{
    public Person(String fn, String ln, int a, Gender g)
    {
        this.firstName = fn; this.lastName = ln; this.age = a; this.gender = g;
    }

    public String getFirstName() { return firstName; }
```

```
    public String getLastName() { return lastName; }
    public Gender getGender() { return gender; }
    public int getAge() { return age; }
    public Person getSpouse() { return spouse; }

    public void setFirstName(String value) { firstName = value; }
    public void setLastName(String value) { lastName = value; }
    public void setGender(Gender value) { gender = value; }
    public void setAge(int value) { age = value; }
    public void setSpouse(Person value) { spouse = value; }

    public String toString()
    {
        return "[Person: firstName=" + firstName +
            " lastName=" + lastName +
            " gender=" + gender +
            " age=" + age +
            " spouse=" + spouse.getFirstName() +
            "]";
    }

    private String firstName;
    private String lastName;
    private int age;
    private Person spouse;
    private Gender gender;
}
```

シリアライズによって生成された出力では、指定されたソース・ファイルの中の多種多様な要素 (メソッド名、フィールド名、フィールドの型、アクセス修飾子、等々) に基づいて計算されたハッシュを使用して、そのハッシュ値をシリアライズ・ストリームのハッシュ値と比較します。

この 2 つの型が実際には同じものであると Java ランタイムが認識できるように、2 番目およびそれ以降のバージョンの `Person` は、最初にシリアライズしたバージョンのハッシュと同じく、(private static final `serialVersionUID` フィールドに保存された) ハッシュを持つ必要があります。従ってここで必要なものは serialVersionUID フィールドです。この `serialVersionUID` は、最初の (つまりバージョン 1 の) `Person` クラスに対して JDK の `serialver` コマンドを実行することで計算することができます。

`Person` の `serialVersionUID` が得られると、最初のオブジェクトのシリアライズ・データから `PersonV2` オブジェクトを作成できるだけでなく (このオブジェクトに新しいフィールドが追加されると、通常それらのフィールドにはデフォルト値として任意の値が設定されます (ほとんどの場合は「ヌル」) になります)、その逆も可能になります。つまり特別な苦労もなく `PersonV2` のデータから元の `Person` オブジェクトをデシリアライズすることができます。

2. シリアライズはセキュアではない

シリアライズのバイナリー・フォーマットは、人間が十分読めるようになっており、そのフォーマットから完全に元の内容を追跡できるという事実を知って Java 開発者が落胆することはよくあります。実際、シリアライズされたバイナリー・ストリームの内容をコンソールに出力するだけで、そのクラスがどんなものか、何が含まれているのかを十分に判断することができます。

これはセキュリティーに関して憂慮すべき問題が含まれていることを示しています。例えば RMI によってリモート・メソッドの呼び出しを行うと、通信路を介して送信されるオブジェクトのプライベート・フィールドはすべて、ソケット・ストリームの中でほとんど平文と同じように見え

てしまいます。これは明らかに、セキュリティーに関する最も基本的な注意事項にも違反しています。

幸いなことにシリアライズには、シリアライズ前とデシリアライズ後の両方で、シリアライズ・プロセスを「フック」してフィールド・データをセキュアに (つまりわかりにくく) する機能があります。そのためには、`Serializable` オブジェクトに `writeObject` メソッドを追加します。

シリアライズ・データをわかりにくくする

ここで例えば、`Person` クラスの機密データが age (年齢) フィールドだったとします。結局のところ、女性は決して年齢を明らかにせず、男性は決して年齢を語ろうとしないものです。このデータをわかりにくくするために、シリアライズ前にはデータを左ローテートし、デシリアライズ後に右ローテートして元に戻します。(これよりもセキュアなアルゴリズムの作成は皆さんに任せることにします。ここでは単に例として、このアルゴリズムを示してあります。)

シリアライズ・プロセスを「フック」するために `Person` クラスに `writeObject` メソッドを実装し、デシリアライズ・プロセスを「フック」するために `Person` クラスに `readObject` メソッドを実装します。この両方のメソッドを実装する際には、詳細部分を適切に行うことが重要です。もしアクセス修飾子、パラメーター、あるいは名前がリスト 4 に示すものと異なると、このコードを実行したときに警告することもなく失敗し、誰にでも `Person` の年齢が見られるようになってしまいます。

リスト 4. シリアライズ・データをわかりにくくする

```
public class Person
    implements java.io.Serializable
{
    public Person(String fn, String ln, int a)
    {
        this.firstName = fn; this.lastName = ln; this.age = a;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getAge() { return age; }
    public Person getSpouse() { return spouse; }

    public void setFirstName(String value) { firstName = value; }
    public void setLastName(String value) { lastName = value; }
    public void setAge(int value) { age = value; }
    public void setSpouse(Person value) { spouse = value; }

    private void writeObject(java.io.ObjectOutputStream stream)
        throws java.io.IOException
    {
        // "Encrypt"/obscure the sensitive data
        age = age >> 2;
        stream.defaultWriteObject();
    }

    private void readObject(java.io.ObjectInputStream stream)
        throws java.io.IOException, ClassNotFoundException
    {
        stream.defaultReadObject();

        // "Decrypt"/de-obscure the sensitive data
        age = age << 2;
    }
```

```
    public String toString()
    {
        return "[Person: firstName=" + firstName +
            " lastName=" + lastName +
            " age=" + age +
            " spouse=" + (spouse!=null ? spouse.getFirstName() : "[null]") +
            "]";
    }

    private String firstName;
    private String lastName;
    private int age;
    private Person spouse;
}
```

わかりにくくしたデータを見る必要がある場合には、シリアライズされたデータ・ストリームやデータ・ファイルを見るだけでよいのです。しかもこのフォーマットは人間が十分読めるようになっているため、たとえクラスを入手できない場合でも、シリアライズされたストリームの内容を読み取ることができます。

3. シリアライズ・データには署名と封印が可能

この前のヒントではシリアライズ・データをわかりにくくすることが目的でしたが、シリアライズ・データの暗号化やデータが変更されていないかどうかについては考えませんでした。暗号化や署名管理を `writeObject` と `readObject` を使って行うことも可能ですが、もっと良い方法があります。

オブジェクト全体の暗号化と署名が必要な場合に最も単純な方法は、そのオブジェクトを `javax.crypto.SealedObject` や `java.security.SignedObject` によるラッパーの中に置く方法です。どちらもシリアライズ可能であるため、`SealedObject` の中にオブジェクトをラップすると、元のオブジェクトの周囲に一種の「ギフト・ボックス」が作られます。暗号化には対称キーが必要であり、そのキーを独立に管理する必要があります。同様に、`SignedObject` を使うとデータを検証することができ、この場合も対称キーを独立に管理する必要があります。

この 2 つのオブジェクトを使うことでシリアライズ・データの封印と署名ができ、デジタル署名の検証や暗号化の詳細に苦労しないで済みます。素晴らしいと思いませんか？

4. シリアライズではストリームにプロキシーを置くことが可能

場合によると、あるクラスがデータのコア要素を含み、その要素から、そのクラスの他のフィールドを派生させたり取得したりする場合があります。こうした場合、そのオブジェクト全体をシリアライズする必要はありません。それらのフィールドは transient で修飾することができますが、その場合でも、そのクラスはメソッドがフィールドにアクセスするたびに、そのフィールドが初期化されているかどうかをチェックするコードを明示的に生成する必要があります。

シリアライズすることが基本的な目的なので、そうした場合にはフライ級の選手、つまりプロキシーをストリームの中に配置した方が適切です。元の `Person` に `writeReplace` メソッドを提供すると、`Person` の代わりに異なる種類のオブジェクトをシリアライズすることができます。同様に、デシリアライズの際に `readResolve` メソッドが見つかると、このメソッドが呼び出され、代わりのオブジェクトが呼び出し側に返されます。

プロキシーのパッキングとアンパッキング

writeReplace メソッドと readResolve メソッドの 2 つを組み合わせて使用すると、Person クラスはそのデータすべて (あるいはデータのコア・サブセット) を PersonProxy にパッキングしてストリームの中に入れ、そのストリームをデシリアライズする際にパッキングを解除することができます。

リスト 5. プロキシーを使う

```
class PersonProxy
    implements java.io.Serializable
{
    public PersonProxy(Person orig)
    {
        data = orig.getFirstName() + "," + orig.getLastName() + "," + orig.getAge();
        if (orig.getSpouse() != null)
        {
            Person spouse = orig.getSpouse();
            data = data + "," + spouse.getFirstName() + "," + spouse.getLastName() + ","
              + spouse.getAge();
        }
    }

    public String data;
    private Object readResolve()
        throws java.io.ObjectStreamException
    {
        String[] pieces = data.split(",");
        Person result = new Person(pieces[0], pieces[1], Integer.parseInt(pieces[2]));
        if (pieces.length > 3)
        {
            result.setSpouse(new Person(pieces[3], pieces[4], Integer.parseInt
              (pieces[5])));
            result.getSpouse().setSpouse(result);
        }
        return result;
    }
}

public class Person
    implements java.io.Serializable
{
    public Person(String fn, String ln, int a)
    {
        this.firstName = fn; this.lastName = ln; this.age = a;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getAge() { return age; }
    public Person getSpouse() { return spouse; }

    private Object writeReplace()
        throws java.io.ObjectStreamException
    {
        return new PersonProxy(this);
    }

    public void setFirstName(String value) { firstName = value; }
    public void setLastName(String value) { lastName = value; }
    public void setAge(int value) { age = value; }
    public void setSpouse(Person value) { spouse = value; }

    public String toString()
```

```
    {
        return "[Person: firstName=" + firstName +
            " lastName=" + lastName +
            " age=" + age +
            " spouse=" + spouse.getFirstName() +
            "]";
    }

    private String firstName;
    private String lastName;
    private int age;
    private Person spouse;
}
```

PersonProxy は Person のすべてのデータを追跡する必要があることに注意してください。これは多くの場合、プロキシーが Person の内部クラスであってプライベート・フィールドにアクセスできる必要がある、ということです。また PersonProxy は場合によると他のオブジェクト参照を追跡し、例えば Person の配偶者 (spouse) を手動でシリアライズしなければならない場合もあります。

この手法は、読み書きをバランスさせる必要のない数少ない事例の 1 つです。例えば、別の型にリファクタリングされたバージョンのクラスに readResolve メソッドを持たせることで、シリアライズされたオブジェクトを新しい型に暗黙的に変換することができます。同様に、そのクラスで writeReplace メソッドを使用することで古いクラスを新しいバージョンにシリアライズすることもできます。

5. 信頼し、ただし検証する

シリアライズされたストリームは元々ストリームに書き込まれたデータと常に同じデータである、と想定できるならば素晴らしいことです。しかし米国の前大統領がかつて指摘したように、「信頼し、ただし検証する」という方針の方が安全です。

シリアライズされたオブジェクトの場合、「信頼し、ただし検証する」ということは、デシリアライズの後、各フィールドが確実に適切な値になるように「念のために」検証する、ということです。そのためには、ObjectInputValidation インターフェースを実装し、validateObject() メソッドをオーバーライドします。そして、この validateObject() メソッドが呼び出された場合に何か問題があるような場合には、InvalidObjectException がスローされます。

まとめ

Java オブジェクトのシリアライズは大半の Java 開発者が考えている以上に柔軟であり、それを活用することで厄介な状況を切り抜けられることがよくあります。

幸いなことに、こうしたコーディングのヒントは JVM のあちこちにあります。重要なことはそれらのヒントについて知り、困難に直面した際にそれらを活用できるようにしておくことです。

この連載「今まで知らなかった 5 つの事項」の次回では、Java のコレクションを取り上げます。それまで、シリアライズを楽しんでみてください。

ダウンロード

内容	ファイル名	サイズ
Sample code for this article	5things1-src.zip	10KB

著者について

Ted Neward

Ted Neward has written over 250 articles and a dozen books across many different technologies, including .NET, iOS, Java, Android, and JavaScript. He resides in Seattle with his wife, two kids, nine laptops, fourteen mobile devices, and two cats. Email him if you're interested in having him or his company work with you.