Preferences APIを使用してオブジェクトを保存する

このAPIはデータを単純なオブジェクトとして表現可能な場合に役立つストレージです

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

2003年 10月 14日

Preferences API(軽量な、JDK 1.4に取入れられたクロスプラットフォームの永続性API)は、少量データ(文字列、単純なバイト配列など)の保存を目的とし、従来のデータベースへのインターフェースを目的としたものではありません。しかしながら、単純なオブジェクトとしてデータを表現可能な場合、それはストレージのような役割を果たします。この記事は、Preferences APIの入門を提示し、オブジェクトがどのように保存されるかを説明し、コードライブラリーを提供して実際の処理を明示します。

Preferences APIは軽量な、JDK 1.4に取り入れられたクロスプラットフォームの永続性APIです。それは従来のデータベースエンジンへのインターフェースではなく、実際の永続性のための適切なOS固有のバックエンドを使用するために作られています。このAPIは少量データの保存を目的としています。事実、その名前自体が目的を示していますが、主な使用目的はフォントサイズやウィンドウレイアウトのようなユーザ固有の設定情報(すなわちプリファレンス)を保存することです。(もちろんどんなものでもその中に保存することが可能です。)

Preferences APIは、文字列、数値、ブール値、単純なbyte配列などの保存を目的としています。この記事では、Preferences APIを使用してオブジェクトを保存する方法を示し、詳細を説明するワーキングライブラリを提供します。これは、データが文字列や数値のような個別の値としてではなく、単純なオブジェクトとして簡単に表現される場合に役に立ちます。

はじめに使い方の簡単なサンプルをいくつか含むAPIの簡潔な説明を示し、次にオブジェクトがAPIを使用してどのように保存されるかの詳細を示し、それを説明するコードを提示します。また、実際にこのAPIのいくつかのサンプルを示します。

なぜPreferences APIは作られたのか

もしPreferences APIが、主としてJavaプログラムのMicrosoft Windowsレジストリへのアクセスのために作成されたとしたら、意外に思うことでしょう。なぜこう言ったかというと、このAPIの設計はWindowsレジストリの設計に似ており、この記事の最初の3つの段落の内容は、レジストリにもほぼ同じように当てはまるのです。

しかしながら、Preferences APIはJava言語自体のようにクロスプラットフォームを意識しており、Windows以外のシステムでも少なくとも同様には動作します。(この記事のコードは、もちろんクロスプラットフォームです。)

Preferences APIの仕様は、このAPIがどのようにインプリメント可能かを明示せず、行う必要のあることしか明示しません。Java Runtime Environment (JRE)の各インプリメンテーションのこのAPIのインプリメント方法は異なります。多くの非レジストリインプリメンテーションは、APIデータをXML形式のファイルで、ユーザのホームディレクトリか共有ディレクトリーに保存します。

Windows レジストリのように、Preferences APIはデータの保存に階層的ツリーメタフォーを使用します。起点はルートノード です(ルートノードはツリーの基準であり、 他のすべてのノードはこのノードの子孫となります)。ノードは、他のノードと同様に名前つきの値も含むことができます。異なるプログラムはツリーの異なる箇所にデータを保存します。したがって互いに衝突することはありません。お分かりのように、Preferences APIは、そのような衝突を防ぐために特別な対策をとっています。

Preferences APIがどのように動作し使用されるのかを少し見てみましょう。

Preferences を使用する

Preferences APIを理解する最良の方法は実際に使用してみることです。まず最初に行う事は、ルートノードへのアクセスです。

```
Preferences root = Preferences.userRoot();
```

このコードは、データツリーのユーザルート を返します。システムのデータはすべてツリーに保存されると前述しましたが、これは厳密には正しくありません。実際には、そこにはユーザツリーとシステムツリーの2つの データツリーが存在します。2つのツリーは同じ振舞いをしますが、それらは異なる目的を持っています。ユーザツリーが各ユーザごとに異なる一方で、システムツリーはすべてのユーザが利用可能なデータを保存するために使用されます。

これらの2つのツリーは必然的に異なる目的で使用されます。フォントプリファレンスは、ユーザ固有のものなのでユーザツリーに保存します。一方、プログラムの位置は、すべてのユーザにとって同じであり、すべてのユーザに使用される可能性があるので、システムツリーに保存します。

小さなプログラムはシステムツリーかユーザツリーのいずれかを使用し、大きなアプリケーションは両方を使用する可能性があります。この記事は、ユーザツリーとシステムツリーが同じ振舞いをすることを念頭に置きつつ、ユーザツリーのみについて取り扱います。

それでは、Preferences APIを使用して単純な値を読み書きする方法を見て行きましょう。

値を取得する

ルートノードを取得したら、値の読み書きに使用します。以下はフォントサイズの書き込み方です。

```
root.putInt( "fontsize", 10 );
```

また以下は、そのフォントサイズの読取り方です。

```
int fontSize = prefs.getInt( "fontsize", 12 );
```

`getInt()`は、デフォルト値（この場合12）を求めるものです。

もちろん、整数以外も読み書き可能です。多くの本来のJavaの型は読み書き可能です。以下のサンプルで示すように、ノードを他のノード内に保存することも可能です。

```
Preferences child = parent.node( "child" );
```

これがPreferences APIの全貌です。この有用性について残りは詳細にて示します。次のセクションにてそのうちの一つを示します。

パッケージごとにノードを取得する

2人のプログラマがそれぞれ「font size」という名前で異なるフォントサイズを保存したとしたら、問題が起こることは容易に想像がつきます。あるプログラムのプリファレンスは、別のプログラムにも影響を与えます。

その解決策は、以下のようにパッケージ独自の位置にそれを保存することです。

```
Preferences ourRoot = Preferences.userNodeForPackage( getClass() );
```

`userNodeForPackage()`メソッドはクラスオブジェクトを取得し、そのクラス固有のノードを返します。このように各アプリケーション(自身のパッケージ内にあると推測する)は独自のプリファレンスノードを持ちます。

Preferences APIの使い方についてよい考えがありますが、それにはオブジェクトを扱えるようにするために拡張する方法を知る必要があります。

オブジェクトを保存する

以下はPreferences ツリーにオブジェクトを書き込む理想的な方法です。

リスト1. Preferences ツリーにオブジェクトを書き込むための理想的方法

```
Font font = new Font( ... );
Preferences prefs = Preferences.userNodeForPackage( getClass() );
prefs.putObject( "font", font );
```

しかし残念な事に、Preferences オブジェクトは`putObject()`や`getObject()`メソッドを持ちません。これにできるだけ近い形をとるために、`PrefObj`と呼ばれるクラスにこれらのメソッドをインプリメントすることにします。以下に、それがどのように動作するかを示します。

リスト2. putObject()とgetObject()のインプリメント

```
Font font = new Font( ... );
Preferences prefs = Preferences.userNodeForPackage( getClass() );
PrefObj.putObject( prefs, "font", font );
```

これは`Preferences` クラスにメソッドを追加するのとほとんど同じです。

次のセクションでは、`getObject()`と`putObject()`がどのようにインプリメントされるかを示します。

オブジェクトをバイト配列に変換する

ここで使用する手法には2つの秘訣があります。１つめはオブジェクトのバイト配列への変換です。これを行う理由は簡単で、Preferences オブジェクトはオブジェクトを扱わずバイト配列を扱うためです。

幸いこれを最初から記述する必要はなくJava言語に組み込まれたものを利用します。オブジェクトをバイト配列に変換する方法はいくつかありますが、以下は`PrefObj`クラスで行う方法を示します。

リスト3. オブジェクトをバイト配列に変換する

```
static private byte[] object2Bytes( Object o ) throws IOException {
  ByteArrayOutputStream baos = new ByteArrayOutputStream();
  ObjectOutputStream oos = new ObjectOutputStream( baos );
  oos.writeObject( o );
  return baos.toByteArray();
}
```

ここでは`ObjectOutputStream`クラスが不可欠です。それはオブジェクトを現実にバイトストリームに変換します。`ByteArrayOutputStream`を`ObjectOutputStream`でラッピングすることにより、バイトストリームをバイト配列に変換することができます。

以下に、反対のことを行うメソッドを示します。

リスト4. バイト配列をオブジェクトに変換する

```
static private Object bytes2Object( byte raw[] )
    throws IOException, ClassNotFoundException {
  ByteArrayInputStream bais = new ByteArrayInputStream( raw );
  ObjectInputStream ois = new ObjectInputStream( bais );
  Object o = ois.readObject();
  return o;
}
```

`ObjectOutputStream`はjava.io.Serializableインターフェースをインプリメントしたオブジェクトしか扱わないことを覚えておいてください。幸いにもそれは、Serializableのインプリメントを宣言したプログラムの任意のオブジェクトと、コアJavaライブラリのほぼすべてのオブジェクトを含みます。

前述のとおりPreferences APIはバイト配列を扱いますが、ここで構築したバイト配列がまだ完全ではないということが、次のセクションで分かるでしょう。

オブジェクトを分割する

Preferences APIでは保存可能なデータサイズに制限があります。特に、文字列はMAX_VALUE_LENGTHの値にて制限され、バイト配列は文字列にエンコードして保存されるためMAX_VALUE_LENGTHの75パーセント以内の長さに制限されます。

一方、オブジェクトは任意のサイズにすることが可能ですが、それを小さく分割する必要があります。当然ながら最も簡単な分割方法は、最初にバイト配列に変換した後、バイト配列を小さく分ける方法です。以下は、再び`PrefObj`から、バイト配列を分割するためのコードを示します。

リスト5. 保存可能な大きさにバイト配列を分割する

```
static private byte[][] breakIntoPieces( byte raw[] ) {
  int numPieces = (raw.length + pieceLength - 1) / pieceLength;
  byte pieces[][] = new byte[numPieces][];
  for (int i=0; i<numPieces; ++i) {
    int startByte = i * pieceLength;
    int endByte = startByte + pieceLength;
    if (endByte > raw.length) endByte = raw.length;
    int length = endByte - startByte;
    pieces[i] = new byte[length];
    System.arraycopy( raw, startByte, pieces[i], 0, length );
  }
  return pieces;
}
```

難しい事はなにもありません。単に配列の配列を作成するだけです。各配列の長さはほとんどがpieceLengthのバイト長と同じです。(pieceLength はMAX_VALUE_LENGTHの4分の3の長さです)。これに対応して、分割されたオブジェクトを再び組み立てるメソッドを示します。

リスト6. バイト配列へオブジェクトの断片を再統合する

```
static private byte[] combinePieces( byte pieces[][] ) {
  int length = 0;
  for (int i=0; i<pieces.length; ++i) {
    length += pieces[i].length;
  }
  byte raw[] = new byte[length];
  int cursor = 0;
  for (int i=0; i>pieces.length; ++i) {
    System.arraycopy( pieces[i], 0, raw, cursor, pieces[i].length );
    cursor += pieces[i].length;
  }
  return raw;
}
```

このルーチンは、オブジェクトの断片の合計長を算出し、その長さの単一の配列を作成します。その後、オブジェクトの断片を次々にその配列へコピーします。

オブジェクトの断片を読み書きする

ここでは2つめの秘訣を示します。値をノードに変換するという秘訣です。通常、Preferences APIを使用して値を保存する場合は、プリファレンスデータツリーのノードの１つのスロットにそれを配置します。

しかし、ここではそれはまったく行いません。オブジェクトが単一の値でも、1セットの固定長バイト配列に変換します。もしバイト配列が1つだけなら、Preferences APIが直接バイト配列を扱うのでデータツリーのスロットへの書き込みは簡単です。しかし、通常は多数の配列を持つため、そのようにはいきません。

この秘訣は各オブジェクトごとにノードを割り当てるというものです。これがどういうことかを明確にしましょう。

通常は、１つのノード内の複数のスロット中の、１つのスロットに値を保存します。しかし、ここでは各オブジェクトごとに１つのノードを作成し、そのノードの複数のスロットに複数のバイト配列を保存します。これについてより具体的に扱いましょう。もし可能ならば、単一のスロットに１つのオブジェクトを保存するでしょう。

リスト7. 単一のスロットに１つのオブジェクトを保存する

```
Preferences parent = ....;
parent.putObject( "child", object );
```

しかし、Preferences は`putObject()`メソッドを持たないので、これを行う事はできません。その代わりに、以下に示すようにノードを１つ作成し、その中に複数のバイト配列を保存します。

リスト8. 代わりに、１つのノードにバイト配列を保存する

```
Preferences parent = ....;
Preferences child = parent.node( "child" );
for (int i=0; i<pieces.length; ++i) {
  child.putByteArray( ""+i, pieces[i] );
}
```

したがって、「child」という名のスロットに単一の値を保存する代わりに、私たちは「child」という名のノードに複数の値を保存しています。これらの値は数値キー（0、1、2など）を使用して保存されます。

数値キーの使用は、後のオブジェクト断片の読み取りを容易にします。

リスト9. 読み取りが容易なオブジェクト断片を読み取る

```
Preferences parent = ....;
Preferences child = parent.node( "child" );
for (int i=0; i<numPieces; ++i) {
  pieces[i] = child.getByteArray( ""+i, null );
}
```

次のセクションでは、これらのステップをすべて組み合わせたルーチンを示します。

すべてを統合する

すべてを統合する`PrefObjs`は前述のリスト3、5、8のメソッドを呼ぶ以下の`putObject()`という静的メソッドを持ちます。

リスト10. メソッドputObject()は、オブジェクト断片の書き込みに他のメソッドを使用します

```
static public void putObject( Preferences prefs, String key, Object o )
    throws IOException, BackingStoreException, ClassNotFoundException {
  byte raw[] = object2Bytes( o );
  byte pieces[][] = breakIntoPieces( raw );
  writePieces( prefs, key, pieces );
}
```

メソッド`putObject()`は全部の処理を3ステップに分けて行い、前述の3つのメソッドによって統合します。オブジェクトをバイト配列に変換し(リスト3)より小さな配列へ分割し(リスト5)、その後Preferences APIにオブジェクト断片を書き込みます。

以下は、読み取り用の同様のメソッドを示します。

リスト11. メソッドgetObject()は、オブジェクト断片の読み取りに同様のことを行います

```
static public Object getObject( Preferences prefs, String key )
    throws IOException, BackingStoreException, ClassNotFoundException {
  byte pieces[][] = readPieces( prefs, key );
  byte raw[] = combinePieces( pieces );
  Object o = bytes2Object( raw );
  return o;
}
```

このメソッドはPreferences APIからオブジェクト断片を読み取り、単一のバイト配列へ統合しオブジェクトに再変換します。

情報を別の場所に保存する

ご覧のように、これはPreferences APIが持つ機能は使用し、持たない 機能はインプリメントするという安全な方法であり、既存のライブラリを拡張するよい方法です。理論上、ライブラリを直接変更したりサブクラスを作成することは可能ですが、そのようなやり方はPreferences APIを使用する他のプログラムに影響を及ぼす可能性があります。この手法を使用すれば、APIを安全で便利な方法で拡張するとともに、オリジナルのAPIを原型のまま残しておくことが可能です。

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。