JDK のモバイル化と京都でのデモの話

星名大輔

1999 年 10 月 25 日

1　はじめに

JDK のモバイル化というのは、 JDK の全てのクラスのオブジェクトが migrate できるように改良することです。こうした改良を加えた JDK を Mobile JDK と呼んでいます。改良する際、問題となるのは、ファイルやポートなどの計算機のリソースを扱うクラスや、GUI に関係するクラスについて、単純にそれらのオブジェクトを移動させるだけでなく、それらのオブジェクトが扱っている「物」をどうするかということです。今回は、 `java.io` パッケージと `java.awt` パッケージについての話しをしたいと思います。

2　JavaGO についての予備知識

JavaGO は実行中のプログラムを実行の途中で停止し、スタックを含むプログラムの実行状態を別のコンピュータに送り、同じ状態で再実行させることができます。

JavaGO プログラムは Java を 3 つのキーワードで拡張した言語で記述され、そのプログラムは JavaGO のコンパイラ (jgoc) によって Java のプログラムに変換されます。

2.1　Java RMI のサンプルプログラム

JavaGO は RMI を利用しおり、 RMI を知らない人のために、少しだけ RMI の説明をします。用意するファイルは以下の 3 つです。 `ServerImpl` オブジェクトと `Client` オブジェクトは異なる計算機上にあるとします。 `Client` オブジェクトが同じ計算機上にない `ServerImpl` オブジェクトのメソッド `foo` を呼び、 `Agent` オブジェクト[1]が計算機間を移動する例です。ソースは以下の通り。

- `ServerImpl` クラス

```
public class ServerImpl extends UnicastRemoteObject implements Server{
     public ServerImpl(){/* コンストラクタ */
          Naming.bind("//venus/ServerImpl",this);
     }

     public void foo(Agent a) throws RemoteException{
```

[1] コードはここには載せていない。

```
 7:             a.sayHello();
 8:         }
 9:     }
```

- Server インターフェイス

```
10:     public interface Server extends Remote{
11:         public void foo(Agent a) throws RemoteException;
12:     }
```

- Client クラス

```
13:     public class Client{
14:         public static void main(String args[]){
15:             Server server=(Server) Naming.lookup("//venus/ServerImpl");
16:             server.foo(new Agent());
17:         }
18:     }
```

この例でわかることは、 Client オブジェクトは同じ計算機上にない ServerImpl オブジェクト[2]の foo メソッドを呼べる[3]ということと、 16 行目で引数として渡された Agent オブジェクトは ServerImpl オブジェクトの存在する計算機上へ移動する[4]ということです。 JavaGO は RMI のこれらの機能を利用して、 migration を行っています。また、 Agent オブジェクトのように計算機間を移動するオブジェクトは java.io.Serializale インターフェイスを implement していなければならないという約束事があります。

2.2 3 つのキーワード go, undock, migratory

- go
  go ステートメント を実行することによりマイグレーションを行うことができる。

```
System.out.println ("Good bye, harp.");
go("//venus/SimpleServer");
System.out.println ("Hello, venus.");
```

- undock
  migration を行うスタックの範囲を制限したい時には undock ステートメントを用いる。

```
undock{
   ...
}
```

[2]このようなオブジェクトをリモートオブジェクトと呼びます。
[3]このようなメソッドをリモートメソッドと呼びます。
[4]このオブジェクトはコピーされ、移動元と移動先の両方に存在するようになる。

- `migratory`
  migration を行うかもしれないメソッドはあらかじめ `migratory` と宣言しておかなければならない。これは JavaGO の約束事です。

  ```
  public migratory void foo();
  ```

3 java.io パッケージのモバイル化

3.1 やりたいこと

やりたいことは、単純で、

> migrate 後もそれまでに扱っていたリソース（ファイル、パイプ）を引き続き扱うことができる。

という機能を持つように `java.io` パッケージのクラスを改良をすることです。次のような例を見ると分かりやすいと思います。

```
 1:// 計算機 X 上でファイル a1 を操作するためのオブジェクトを作成
 2:      FileOutputStream f1=new FileOutputStream("a1");
 3://X 上のファイル a1 に文字列を書き込み。
 4:      f1.write('Hello World!');
 5:      go("Y");// 計算機 Y へ go
 6:// 計算機 Y 上でファイル a2 を操作するためのオブジェクトを作成
 7:      FileOutputStream f2=new FileOutputStream("a2");
 8://Y 上のファイル a2 に文字列を書き込み。
 9:      f2.write('Hello World!');
10:// さらに X 上のファイル a1 に文字列を書き込み。
11:      f1.write('Hello World!');
```

11 行目では Y 上で X 上のファイル a1 に対して書き込みを行っていることに注意して下さい。

`javago.io` パッケージでは `java.io` パッケージにあるクラスを全て用意し、プログラマには `java.io` パッケージの代わりに `javago.io` パッケージを使用してもらう。そのためには `java.io` パッケージを `import` とする代わりに `javago.io` パッケージを `import` すればよい。

3.2 実現方法

前節で述べた機能を提供するための実現方法の概要を `File` クラスを例として説明します。

3.2.1 概要

`java.io.File` クラスに対して `javago.io` パッケージでは、

javago.io.File クラス
javago.io.WrapperFile インターフェイス
javago.io.WrapperFileImpl クラス

を用意します[5]。それぞれ役割は以下の通りです。

- javago.io.File クラス
  java.io.File クラスと同じインターフェイスを持つクラスで、プログラマには java.io.File クラスの代わりにこのクラスを使用してもらう。このクラスは WrapperFileImpl クラス（のオブジェクト）への参照を持っていて、このクラスのメソッドは実際にはファイルに対して直接操作をすることはなく、全て WrapperFileImpl クラスの同じ名前のメソッドに委託する。
  また、このクラスは WrapperFileImpl クラス（のオブジェクト）への参照を transient 変数として持っているため、このクラスのオブジェクトが migrate しても WrapperFileImpl オブジェクトは migrate しないことに注意。すなわち、ラッパーオブジェクトをオリジナルオブジェクトとそのリソースは、常に同一計算機上にある。

- javago.io.WrapperFile インターフェイス
  Remote インターフェイスのサブインターフェイスであり、 WrapperFileImpl クラスはこのインターフェイスで定義されているメソッドを implement することによって、それらのメソッドはリモートメソッドになることができる（つまり migrate 先から呼ぶことができる）。 JavaRMI の約束事である。

- javago.io.WrapperFileImpl クラス
  java.io.File クラスと同じインターフェイスを持つクラスで、 javago.io.File クラスからの委託を引き受け、今度はそれを java.io.File クラスに委託し、ファイルへの操作を行う（このクラスは java.io.File クラスへの参照を持っている）。
  つまり、実際のファイル操作はこの java.io.File クラスが行うことになる。また、このクラスは WrapperFile インターフェイスを implement しているため、 javago.io.File クラスは migrate した後も、 migrate 先から、このクラスのメソッドを呼ぶことができる。

3.2.2 オブジェクト生成時の処理

javago.io.File クラスのコンストラクタ内では以下の処理が行われる。

1. ラッパーオブジェクトを rmiregistry に登録するための名前を作成する。
2. ラッパーオブジェクトを作成し、 rmiregistry に登録する。
3. ラッパーオブジェクトを作成が作成される時、ラッパーオブジェクトのコンストラクタ内ではオリジナルオブジェクトが作成される。

[5] java.io.File クラス（オブジェクト）をオリジナルクラス（オブジェクト）、 javago.io.File クラス（オブジェクト）のことをモバイルクラス（オブジェクト）、 javago.io.WrapperFileImpl クラス（オブジェクト）のことをラッパークラス（オブジェクト）と（File クラスに限らず一般的に）呼ぶことにする

3.2.3 migrate 先へ到着直後の処理

モバイルオブジェクトが migrate した直後、このオブジェクトが持っているラッパーオブジェクトへの参照は `transient` 変数として保持していたため `null` になっている。よって、ラッパーオブジェクトへの参照を獲得し直さなければならないのだが、 `readObject` メソッド[6]内で、 Java RMI の `lookup` メソッドを使用することによって、これを行っている。

3.3 `java.io` パッケージと異なる点

いくつかセマンティクスが異なる点があるので、それを簡単に紹介します。これらの違いは、 migrate をするようなオブジェクト特有のものです。

3.3.1 `FileInputStream` クラスと `FileOutputStream` クラス

`FileOutputStream` クラスの次のコンストラクタについて、 1 つ目はこのコンストラクタが呼ばれた計算機上のファイルに対する `FileOutputStream` オブジェクトが作成されるが、 2 つ目は、引数の `File` オブジェクトが表すファイルに対する `FileOutputStream` オブジェクトが作成される。

```
FileOutputStream(String pathname)
FileOutputStream(File file)
```

つまり、ある計算機上であるファイルの File オブジェクトを作成し、そのオブジェクトを持って移動すれば、どの計算機からでも、そのファイルへの読み書きができます。 `FileInputStream` クラスについても同様です。

3.3.2 `PipedInputStream` クラスと `PipedOutputStream` クラス

パイプを利用してスレッド同士が通信するために提供されているクラスです。
例えば、 `PipedInputStream` オブジェクト `in` と `PipedOutputStream` オブジェクト `out` が：

- 同一計算機上で作成された場合、
  `in` が別の計算機上へ migrate すると、 RMI では、それは `in` のコピーが移動する。オリジナルとコピーをそれぞれ i1 及び i2 と呼ぶと、 i1 への参照を持ったスレッドは同一計算機上で、 i2 への参照を持ったスレッドは migrate 先の計算機から `out` への参照を持つスレッドとパイプを使って通信できる。
- 異なる計算機上で作成された場合、
  これは、スレッド同士のネットワーク間の通信となる。ソケット通信と似ている。

[6] serialize されたオブジェクトを復元する時に（VM から）呼ばれるメソッド

4 java.awt パッケージのモバイル化

4.1 やりたいこと

java.awt パッケージのクラス使って作成した GUI アプリが migrate できるように、 java.awt パッケージのクラスを改良することです。

4.2 javago.awt パッケージを用意しなければならない理由

java.awt パッケージの Button クラスや Frame クラスは serializable なので本当は何もしなくても migrate できるはずですが、わざわざ javago.awt パッケージ用意しなければならない理由があります。それは、通常、プログラマが JavaGO でプログラムを移動させる場合、次のような状態のスタックを移動させればいいのですが、

```
        ↑
+---------+
|user code| (この中で'go')
+---------+
```

もし、 call-back メソッドの中で go を書いた場合、 JavaGO は

```
        ↑
+---------+
|user code| (この中で'go')
+---------+
| library | (java.awt パッケージのクラスのメソッドの中で user code を call-back)
+---------+
```

という状態のスタックを移動させなければなりません。 JavaGO は migratory と宣言されていないメソッドのスタックについては移動しようとしないので、「library」内のメソッドに migratory と宣言し、 JavaGO コンパイラで変換する必要があるのですが、「library」はあらかじめ class ファイルとして提供されているものなので、そうすることはできません。つまり、自前で java.awt パッケージに相当するものをつくり、それを JavaGO コンパイラで変換し javago.awt パッケージとして提供する必要があります。

4.3 java.awt のサンプルプログラム

```
    public static void main(String[] args) {
        Frame frame = new Frame();
        Button b=new Button();
        b.addActionListener(new MyActionListener());
        frame.add(b);
        frame.show();
    }

```

```
    public class MyActionListener implements ActionListener{
        public void actionPerformed(ActionEvent e){
                System.out.println("Hello World.");
        }
    }
```

4.4 実現方法

java.io パッケージの場合と同じようにラップするわけですが、さっきより単純で、 Button クラスを例に挙げると、

```
public class Button extends /*javago.awt.*/Component
        implements java.io.Serializable
{
    public Button() {
        wcomp=new SubButton(this);
    }

    public void foo(){
        ((java.awt.Button)wcomp).foo();
    }

    /* 以下省略 */
}
```

とするだけです[7]。ここで、 SubButton クラスというのは java.awt.Button クラスのサブクラスで、ラップしてしまってもイベントがきちんと伝わるためと、各コンポーネントが正しく（再）描画されるために、用意しています。従って、ラップされるのは、（正確には）java.awt.Button クラスでなく、 SubButton クラスということになります。 SubButton クラスは java.awt.Button クラスの全てのメソッドを継承またはオーバーライドしているので、ほぼ同等の機能を持っているわけですが、どう違うかは後で分かります。

その前に、先ほどのサンプルプログラムで使用しているメソッドについて、オリジナルオブジェクトへどのように delegate するかの例を挙げます。

- add メソッド:Frame クラス

```
public Component add(Component comp) {
    ((java.awt.Frame)wcomp).add(comp.getOriginal());
    return comp;
}
```

[7]javago.awt.Component クラスに以下のような宣言がある。

```
java.awt.Component wcomp;
```

自分がラップしている SubFrame オブジェクトに comp がラップしているオブジェクトを add します。 getOriginal メソッドはラップしてるオブジェクトを返します。

- show メソッド:Frame クラス

```
public void show() {
  ((java.awt.Frame)wcomp).show();
}
```

自分がラップしている SubFrame オブジェクトを show するだけです。

結局、 show することにより、ディスプレイ上に表示されるのは、 javago.awt.Frame オブジェクトでなく、 SubFrame オブジェクトということになり、もし、その SubFrame に Button が貼り付いている場合、それは javago.awt.Button クラスがラップしている SubButton クラスのオブジェクトです。

4.5 イベントの伝達

通常、 java.awt パッケージの場合、 java.awt.Button が押されると、 java.awt.Button クラスの processEvent メソッドが呼ばれ、以下の順序で java.awt.event.ActionListener インターフェイスの actionPerformed メソッドが呼ばれます。

1. Button クラスの processEvent メソッド
2. Button クラスの processActionEvent メソッド
3. ActionListener インターフェイスの actionPerformed メソッド

javago.awt.Button クラスでは、この流れが以下のように変ります。イベントを受け取るのは SubButton クラスであることに注意して、

1. SubButton クラスの processEvent メソッド
   java.awt.Button クラスの processEvent メソッドをオーバーライドし、単に、自分をラップしている javago.awt.Button クラスの processEvent メソッドを呼ぶ。
2. Button クラスの processEvent メソッド
   本来、 java.awt.Button クラスの processEvent メソッドと同じことをし、 Button クラスの processActionEvent メソッドを呼ぶ。
3. Button クラスの processActionEvent メソッド
4. ActionListener インターフェイスの actionPerformed メソッド

結局、 SubButton が押されると、初めに SubButton クラスの processEvent メソッドが呼ばれ、最終的にプログラマが implement した javago.awt.event.ActionListener インターフェイス actionPerformed メソッドが呼ばれ、イベントが適切に起ります。

4.6 コンポーネントの再描画

プログラマが、例えば `Frame` 内にある描画をしたいならば、 `paint` メソッドをオーバーライドしなければならない。また、（直接ではないですが）VM から `paint` メソッド呼ばれることにより、適切に再描画される。ここで問題なのは、プログラマが `javago.awt.Frame` クラスの `paint` メソッドをオーバーライドしても、 VM から呼ばれるのは `SubFrame` クラスの `paint` メソッドなので、再描画されないということです。そこで、 `SubFrame` クラスの `paint` メソッドでは、自分をラップしているオブジェクトの `paint` メソッドを呼ぶようにしてやれば、プログラマがオーバーライドした `paint` メソッドが呼ばれる。

5 京都でのデモの話

10 月 13 日に AMO Project の話しをするため、米澤先生、関口さん、橋本さん、大岩君と京都へ行きました。そこで、 emacs 風のエディタを計算機間で migrate させるというデモをやりました。単に GUI アプリが migrate できるというだけでなく、

> JavaGO と Mobile JDK を使えば、 migrate させることを前提とせずに作ったアプリを少し手を加えるだけで migrate させることができる

というのが売りです。お茶大の佐藤先生の AgentSpace も migrate できる GUI アプリをつくることができますが、 AgentSpace の場合は、最初から migrate する GUI アプリを作るつもりでコードを書かなくてはなりません。

実際に、大岩君が「JavaGO のことを何も知らないプログラマ」役として、 emacs 風のエディタを作り、それを僕が migrate できるように手を加えました。どのくらい、手を加えればいいかというと、

- `java.io`/`java.awt` パッケージを `import` している部分を `javago.io`/ `javago.awt` パッケージに書き換える。
- migration を行うメソッドを `migratory` で宣言するように書き加える。

の 2 点です。もちろん、例えば、 `Ctrl-x g` で移動先の入力を受け付けて、その後、 `go` ステートメントを含むメソッドを呼ぶ、というようなコードは加える必要はあります。

今回の大岩君の emacs 風エディタの場合、上の 2 つについての変更点は 10 箇所ぐらいでした。そして、 JavaGO でソース変換して、 javac でコンパイルすれば、 migrate できるようになります。

今後の Mobile JDK の予定としては、 4 年生の山内君（？）がクラスライブラリの半自動モバイル化ツールを作るそうです。

おしまい