Mobilność kodu

Ireneusz Szcześniak

17 kwietnia 2012

1 Wprowadzenie

Mobilność kodu (ang. code mobility) jest cechą jezyka Java, która pozwala maszynie wirtualnej Javy na pobranie plików klas (*.class), a następnie wykonywaniu takiego kodu. Maszyna wirtualna może pobrać pliki klas z serwera HTTP, serwera FTP bądź systemu plików.

Częścią informacji na temat klasy, którą klient RMI otrzymał jest adnotacja mówiąca, skąd plik klasy klient może pobrać. Adnotację taką dodaje maszyna wirtualna Javy, na której jest uruchomiony serwer RMI, na podstawie zdefiniowanej bazy kodu. Baza kodu (ang. codebase) to lista lokalizacji, z których mogą być pobrane pliki klas.

2 Pierwszy przykład

Przykład pokazuje pobranie przez klienta RMI obiektu klasy, której definicji nie zna. Klient sam pobiera plik klasy z serwera HTTP, aby mógł używać tego obiektu.

Interfejs serwera deklaruje funkcję `get`, która zwraca obiekty klasy implementującej interfejs `Runnable`:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ServerIntf extends Remote
{
    Runnable get() throws RemoteException;
}
```

Serwer zwraca metodą `get` obiekty klasy `OurClass`, która implementuje interfejs `Runnable`:

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

public class Server
    extends UnicastRemoteObject
    implements ServerIntf
{
    public Server() throws RemoteException
    {
    }

    public OurClass get()
    {
        return new OurClass();
    }

    public static void main(String[] args) throws Exception
    {
        LocateRegistry.createRegistry(1099);
        ServerIntf server = new Server();
        Naming.rebind("server", server);
    }
}
```

Klient pobiera obiekt zwrócony przez serwer i wykonuje funkcję `run`. Problem w tym, że klient nie zna klasy zwróconego obiektu i musi pobrać plik OurClass.class, aby tę funkcję wykonać.

```
import java.rmi.Naming;
import java.rmi.RMISecurityManager;

public class Client
{
    public static void main(String[] args) throws Exception
    {
        System.setProperty("java.security.policy", "client.policy");
        System.setSecurityManager(new SecurityManager());
        ServerIntf server = (ServerIntf) Naming.lookup("server");
        Runnable o = server.get();
        o.run();
    }
}
```

2.1 Uruchomienie przykładu

- `git clone https://github.com/iszczesniak/RMI.git`
- W pierwszym terminalu uruchom serwer HTTP:
  - `cd RMI/mobility/sample1/server`
  - `python -m SimpleHTTPServer`
  - Sprawdź, np. przez `firefox localhost:8000`, czy serwer działa. W przeglądarce powinna wyświetlić się zawartość katalogu `RMI/mobility/sample1/server`.
- W drugim terminalu uruchom serwer RMI:
  - `cd RMI/mobility/sample1/server`
  - `javac Server.java`
  - `java -Djava.rmi.server.codebase="http://localhost:8000/" Server`
  - Ważne jest "/" na końcu bazy kodu!
- W trzecim terminalu uruchom klienta RMI:
  - `cd RMI/mobility/sample1/client`
  - `javac Client.java`
  - `java Client`

2.2 Dyskusja

Adres lokalizacji, z których klient RMI może pobrać pliki klas używane przez serwer, wskazujemy przy uruchomieniu serwera przez ustawienie zmiennej systemowej `java.rmi.server.codebase`. Robimy to na przykład w komendzie uruchamiającej serwer:

```
-Djava.rmi.server.codebase="http://localhost:8000/"
```

Klient RMI będzie potrzebował definicji klasy OurClass i dlatego JVM będzie szukać tej definicji w katalogu bieżącym, a następnie w lokalizacjach wskazanych w CLASSPATH. Jeżeli tam nie znajdzie definicji, to sprawdzi, czy została dołączona do klasy adnotacja mówiąca o lokalizacji definicji klasy. Dopiero wtedy JVM pobierze definicję, co zostanie wypisane przez serwer HTTP w terminalu.

Aby korzystać z kodu mobilnego, klient *musi* uruchomić Security Managera:

```
System.setProperty("java.security.policy", "client.policy");
System.setSecurityManager(new SecurityManager());
```

W pliku `client.policy` powinna się znajdować zasada bezpieczeństwa pozwalająca na tworzenie połączenia:

```
grant
{
      permission java.net.SocketPermission "*:*", "connect";
};
```

Warto zauważyć, że klasa `OurClass` musi implementować interfejs `Serializable`, ponieważ jej obiekty są poddawane serializacji przy zwracaniu jako wartości metody zdalnej. Jednak klasy mobilne *nie muszą* być serializowalne.

Klasa `NonserializableMobileClass` jest przykładem klasy mobilnej, która nie jest serializowalna. Funkcja `run` klasy `OurClass` tworzy obiekt tej klasy już po stronie klienta RMI. Serwer HTTP odnotuje pobranie nie tylko klasy `OurClass`, ale też klasy `NonserializableMobileClass`.

3 Zadania

1. Zmodyfikować pierwszy przykład tak, aby funkcja `get` zwracała obiekty klasy `OurClass2` wyprowadzonej z klasy `OurClass`. Funkcja `run` klasy `OurClass2` powinna wypisać "Hello from OurClass2!", a następnie wywołać funkcję `run` z klasy `OurClass`.

2. Zaimplementować serwer RMI, który będzie wykonywał obliczenia dla klienta RMI. Serwer RMI będzie implementował metodę zdalną `compute`, która będzie przyjmować obiekt implementujący interfejs `Runnable`. Serwer będzie wykonywał metodę `run`, a potem uprzednio otrzymany obiekt będzie zwracał jako wynik wywołania funkcji `compute`. Serwer RMI powinien pobierać definicje klas dostarczanych przez klienta RMI, a klient RMI wskazywać lokalizację, gdzie pliki klas są dostępne.