RMI introduktion

Denne artikel beskriver Java RMI (Remtote Method Invocation).

Den beskriver teorien bag RMI, viser et simpelt kode eksempel og forklarer hvordan det virker.

Den forudsætter kendskab til Java og generel programmering men ikke til RMI.

Skrevet den **16. Feb 2010** af **arne_v** I kategorien **Programmering / Java** | ★★☆☆☆

Historie:
V1.0 - 12/01/2004 - original
V1.1 - 31/01/2004 - forbedret formatering
V1.2 - 09/02/2004 - flere ændringer af formatering
V1.3 - 16/02/2010 - smårettelser

Teori

RMI bygger på det såkaldte Proxy Pattern (et af de originale patterns fra Design Patterns af Erich Gamma m.fl.).

Ideen er at en client kan kalde remote server kode på samme måde som lokal kode.

Selve kaldet foregår som:

```
client kode--stub kode--(netværk)--skeleton kode--server kode
```

Stub koden har samme interface som server koden. Stub koden pakker alle argumenter ned (serialiserer) og sender dem over en socket til skeleton koden.

Skeleton koden udpakker alle argumenter (deserialiserer) og kalder server koden.

Retur værdien sende tilbage på samme vis bare modsat.

Både stub og skeleton koden genereres automatisk af et værktøj der kommer med Java SDK.

Etableringen af forbindlesen mellem client og server er speciel og involverer noget der hedder RMIRegistry.

Serveren connecter til RMIRegistry og registerer sig under et navn.

Client connecter til RMIRegistry og slår navnet op og får returneret et stub objekt.

Client connecter så via stub objekt til server skeleton objekt.

Eksempel

Her kommer nu et simpelt eksempel som illusterer hvordan det virker.

Calc.java

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

// interfacet som er fælles for stub og server
// skal extende Remote
public interface Calc extends Remote {
   // metoder i interfacet
   // skal throwe RemoteException
   public int add(int a, int b) throws RemoteException;
   public int mul(int a, int b) throws RemoteException;
}
```

CalcImpl.java

```
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

// server klassen
// skal extende UnicastRemoteObject og implementere interfacet
public class CalcImpl extends UnicastRemoteObject implements Calc {
   // constructor
   // skal erklæres eksplicit
   // skal throwe RemoteException
   public CalcImpl() throws RemoteException {
   }
   // metoder fra interfacet
   // må godt men behøver ikke throwe RemoteException
   public int add(int a, int b) {
       // lidt debug så vi kan at det virker
       System.out.println("Vi er blevet bedt om at lave en add");
       return (a + b);
   }
   public int mul(int a, int b) {
       // lidt debug så vi kan at det virker
       System.out.println("Vi er blevet bedt om at lave en mul");
       return (a * b);
   }
   // simpelt hoved program til at teste med
   public static void main(String[] args) {
      try {
```

```
         // binde server objekt til RMIRegistry under navnet "Calc"
         Naming.rebind("Calc", new CalcImpl());
      } catch (RemoteException e) {
         e.printStackTrace();
      } catch (MalformedURLException e) {
         e.printStackTrace();
      }
   }
}
```

TestCalc.java

```
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

// test client klasse
public class TestCalc {
   // hoved program
   public static void main(String[] args) {
      try {
         // slå service op i RMIregistry og hent stub objekt
         Calc c = (Calc)Naming.lookup("Calc");
         // check at service virker
         System.out.println(c.add(2,3));
         System.out.println(c.mul(2,3));
      } catch (MalformedURLException e) {
         e.printStackTrace();
      } catch (RemoteException e) {
         e.printStackTrace();
      } catch (NotBoundException e) {
         e.printStackTrace();
      }
   }
}
```

Compile .java filer til .class filer:

```
javac *.java
```

Generere stub og skeleton udfra server implementations class fil:

```
rmic CalcImpl
```

Starte RMIRegistry:

```
rmiregistry
```

Starte server:

```
java CalcImpl
```

Starte client:

```
java TestCalc
```

RMIRegistry skal have stub i classpath.

Server skal have interface + implementation + skeleton i classpath.

Client skal have stub i classpath.

Hvad er RMI godt til og hvad er det ikke godt til

RMI er meget godt til client/server løsninger på LAN. Det er ikke så velegnet til løsninger hvor netværket er internettet. Det giver ofte problemer med firewalls og porte der skal åbnes. Man kan godt få det til at virke, men til brug over internet er SOAP RPC over HTTP (Web Services) bedre.

Som hovedregel gælder det at fordelene ved RMI er store, når man har mange funktioner med mange argumenter som ikke fylder meget, mens det ikke er velegnet til en funktion med et argument som kan være meget stort (som f.eks. fil upload).

Udviklingen i Java

I nyere Java versioner behøver man ikke generere stub og skeleton på compile time, da de kan genereres on the fly på runtime.

Men koden ovenfor virker stadig.

Videre

For avancerede emner indenfor RMI se:
http://www.eksperten.dk/guide/225

Kommentar af simonvalter d. 12. Jan 2004 | 1

fint beskrevet :) ser frem til den avancerede.

Kommentar af bagnavnet d. 08. Dec 2004 | 2

Lige hvad jeg som nybegynder havde brug for!

Kommentar af iwatter d. 25. May 2004 | 3

Kommentar af ttn- d. 30. Jul 2004 | 4

Igen, som ved stortset alle dine artikler, er der alt for meget kode i forhold til tekst. Forklar og analyser hver der sker. Ellers kan man ikke rigtig kalde det for en artikel. Dog er denne sat til middel, fordi RMI ikke er så avanceret igen.

Kommentar af HannnaBanana d. 18. Oct 2011 | 5

Hmm: Hvordan skal man forstå dette stykke:

Compile .java filer til .class filer:

```
javac *.java
```

Generere stub og skeleton udfra server implementations class fil:

```
rmic CalcImpl
```

Starte RMIRegistry:

```
rmiregistry
```

Starte server:

```
java CalcImpl
```

Starte client:

```
java TestCalc
```

RMIRegistry skal have stub i classpath.

Server skal have interface + implementation + skeleton i classpath.

Client skal have stub i classpath.