Java RMI

2ª PARTE

Factory Pattern para RMI

Factory Pattern para RMI

- Concepto introducido en: Design Patterns, Elements of Reusable Object-Oriented Software.
- Objeto que controla la creación y/o el acceso a otros objetos.
- En el contexto de Java RMI: reducción en el número de objetos que es necesario registrar.

Factory Pattern para RMI

- Ejemplo práctico: la biblioteca.

1. Registro de nuevos usuarios para entrar en la biblioteca.
2. Identificación tarjeta de usuario.
3. Control de préstamo de libros.
4. Reserva de libros para cada usuario.

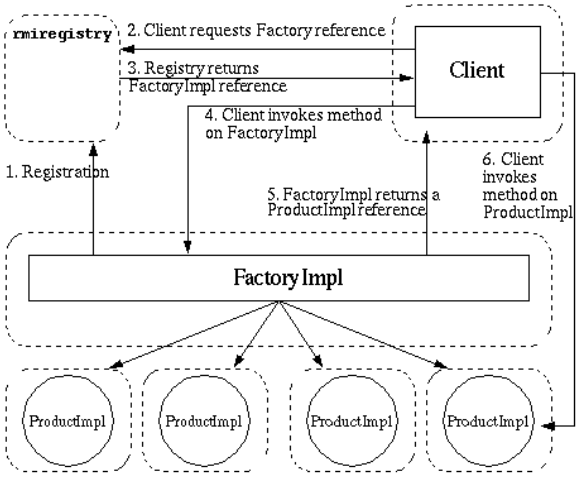
– *El bibliotecario representa una factoría dado que gestiona el acceso al contenido de la biblioteca.*

Factory Pattern para RMI

- Organización interna:
  - Hay dos únicas interfaces remotas que entiende el cliente: *Factory* y *Product*.
  - *FactoryImpl* implementa *Factory interface*.
  - *ProductImpl* implementa *Product interface.*

Factory Pattern para RMI

- Organización interna:



Factory Pattern para RMI

- Organización interna:
  - *FactoryImpl* se registra en el registro RMI.
  - El cliente solicita una referencia a *Factory*.
  - *Rmiregistry* devuelve una referencia remota a *FactoryImpl*.
  - El cliente invoca un método remoto de *FactoryImpl* para obtener una referencia remota de *ProductImpl*.
  - *FactoryImpl* devuelve una referencia remota de un objeto *ProductImpl* existente o bien uno que acaba de ser creado en función de la petición del cliente.
  - El cliente invoca un método remoto de *ProductImpl*.

Factory Pattern para RMI

- El bibliotecario representa un *interface* remoto con uno o más métodos que devuelven objetos que implementan el *interface LibraryCard.*
- El bibliotecario oferta métodos para acceder a los libros.
- La única implementación registrada en el registro RMI es la clase bibliotecario.

Estructura interna de RMI

Estructura interna de RMI

- Buscar respuesta a:
  - ¿Quién realmente crea el objeto de *stub*? ¿El servidor, el cliente o el registro?
  - ¿A qué puerto está escuchando el servidor?
  - ¿Acaso el servidor escucha en el puerto 1099? (puerto por defecto del registro RMI)
  - ¿Cómo sabe el cliente en qué puerto escucha el servidor?
  - ¿Es necesario el registro RMI para hacer funcionar el sistema?
  - ¿Se puede utilizar RMI sin el *rmiregistry*?

Estructura interna de RMI

- Planteamiento inicial: ignorar el registro RMI.
  - Tenemos un servidor y un cliente en máquinas distintas.
  - El servidor extiende *java.rmi.server.UnicastRemoteObject*.
  - El cliente quiere ejecutar un método remoto del servidor.
- Comunicaciones internas mediante *sockets*.
- *Stubs* y *Skeletons* contienen toda la información concerniente a las comunicaciones.

  Cliente ↔ *stub* ↔ [RED] ↔ *skeleton* ↔ Servidor

Estructura interna de RMI

- Funcionamiento a nivel de sockets:
  1. El servidor escucha un puerto.
  2. El cliente no sabe en qué maquina y puerto está escuchando el servidor. **Pero tiene un objeto *stub* que tiene esa información.**
  3. El cliente invoca la función del ***stub***.
  4. El *stub* se conecta al puerto del servido y envía los parámetros siguiendo los siguientes pasos:
     1. El cliente se conecta al puerto de escucha del servidor.
     2. El servidor acepta la conexión entrante y crea un nuevo *socket* para gestionar esta única conexión.
     3. El antiguo puerto de escucha permanece aguardando posteriores peticiones de otros clientes.

Estructura interna de RMI

4. La comunicación cliente-servidor se realiza utilizando el nuevo *socket*.
5. Se realiza el intercambio de los parámetros del método siguiendo un protocolo preestablecido.
6. El protocolo puede ser JRMP (*Java Remote Method protocol*) o CORBA-compatible RMI-IIOP (*Internet Inter-ORB Protocol*).

5. El método es ejecutado en el servidor y el servidor es devuelto al cliente vía *stub*.
6. El *stub* devuelve el resultado al cliente como si hubiera ejecutado la función localmente.

Estructura interna de RMI

- Punto2: *El cliente no sabe en qué maquina y puerto está escuchando el servidor.* ***Pero tiene un objeto stub que tiene esa información.***
- Pero… **el puerto asociado a cada objeto remoto del servidor es dinámico!!!**
- Pregunta: ¿Cómo sabe el cliente en qué máquina y puerto está asociado el servidor si éste es dinámico?

Bootstrap Problem

Estructura interna de RMI

- Es necesario informar al cliente acerca del estado del servidor.
- Registro RMI:
  - *Hashmap* de dupletes {etiqueta, *Stub_object*}.
  - Clase *java.rmi.Naming*.
  - Puerto preestablecido (1099 por defecto).

Estructura interna de RMI

- Ejemplo estructura de un *stub*:

```
public final class CalcImpl_Stub
   extends java.rmi.server.RemoteStub
   implements Calc, java.rmi.Remote
{
   private static final long serialVersionUID = 2;
   private static java.lang.reflect.Method $method_add_0;
   static {
      try {
         $method_add_0 = Calc.class.getMethod("add",
            new java.lang.Class[] {int.class, int.class});
      } catch (java.lang.NoSuchMethodException e) {
         throw new java.lang.NoSuchMethodError(
            "stub class initialization failed");
      }
   }

   // constructors
   public CalcImpl_Stub(java.rmi.server.RemoteRef ref) {
      super(ref);
   }

   // methods from remote interfaces
```

Estructura interna de RMI

```
   // implementation of add(int, int)
   public int add(int $param_int_1, int $param_int_2)
      throws java.rmi.RemoteException

      try {
         Object $result = ref.invoke(this, $method_add_0,
                          new java.lang.Object[]
                          {new java.lang.Integer($param_int_1),
                          new java.lang.Integer($param_int_2)},
                          -7734458262622125146L);
      return ((java.lang.Integer) $result).intValue();
      } catch (java.lang.RuntimeException e) {
         throw e;
      } catch (java.rmi.RemoteException e) {
         throw e;
      } catch (java.lang.Exception e) {
         throw new java.rmi.UnexpectedException("undeclared checked
                                                 exception", e);
      }
   }
}
```


Estructura interna de RMI

1. *RMIRegistry* se ejecuta en el servidor y es un objeto remoto con un **puerto conocido**.

2. El servidor se exporta a un puerto anónimo de la máquina servidora. Este puerto es desconocido a los clientes.

3. Cuando se invoca *Naming.rebind()*, se le pasa la referencia de la implementación del objeto creado. La clase *Naming* construye un objeto *stub* y lo asocia al objeto remoto del registro. Los pasos de creación del *stub* son:

   1. Carga la clase *CalcImpl_Stub* en la JVM.

   2. Toma *RemoteRef obj* de *c*. *RemoteRef ref=c.getRef();*Esta referencia encapsula todos los detalles del servidor, como nombre, dirección y número de puerto asociado.

Estructura interna de RMI

3. Utiliza este objeto *RemoteRef* para construir el *stub*:
   CalcImpl_Stub stub=new CalcImpl_Stub(ref);
4. Para este objeto *stub* al *RMIRegistry* para asociarlo con la etiqueta.
5. *RMIRegistry* almacena internamente la etiqueta y el objeto *stub* en un *hashmap*.

4. Cuando el cliente ejecuta *Naming.lookup()*, para la etiqueta como parámetro. El *RMIRegistry* devuelve el *stub* de vuelta al cliente.
5. Ahora el cliente conoce el nombre del servidor y el puerto de escucha asociado al objeto remoto. El cliente puede invocar al método remoto de su *stub* para ejecutarlo.

Estructura interna de RMI

- El registro RMI únicamente se emplea para realizar *bootstrapping*.
- Alternativa: desarrollar un esquema de comunicación basado en *sockets* que serialicen el *stub* y lo envíe a los clientes.

Socket Factories con Java RMI

Socket Factories con Java RMI

1. Control a nivel de red a cerca de cómo se realiza la invocación de métodos remotos.
2. *Socket factory*:
   - Especifica el formato de comunicación a través de *sockets*.
   - Controla la creación de *sockets*.
   - Por parte del cliente: inicio y establecimiento de comunicaciones.
   - Por parte del servidor: mecanismo de escucha y aceptación de peticiones.

Socket Factories con Java RMI

- Un *socket factory* cliente controla la creación de *sockets* usados para realizar la invocación remota.
- Un *socket factory* servidor controla la creación de *sockets* servidores para recibir invocaciones remotas.
- El *stub* remoto para un objeto remoto concreto contiene el *socket factory* cliente asociado al objeto.
- Métodos:
  - `LocateRegistry.createRegistry`
  - `LocateRegistry.getRegistry`

Socket Factories con Java RMI

- PASO1: Implementar un ServerSocket y Socket personalizado.
  - Encriptación y desencriptación XOR.

XorInputStream.java

```
package examples.rmisocfac;
import java.io.*;
class XorInputStream extends FilterInputStream {

   private final byte pattern;

   public XorInputStream(InputStream in, byte pattern) {
        super(in);
       this.pattern = pattern;
    }

    public int read() throws IOException {
      int b = in.read();
     if (b != -1)
          b = (b ^ pattern) & 0xFF;
     return b;
    }
```

Socket Factories con Java RMI

XorInputStream.java

```
public int read(byte b[], int off, int len) throws
    IOException {
         int numBytes = in.read(b, off, len);

    if (numBytes <= 0)
         return numBytes;

        for(int i = 0; i < numBytes; i++) {
        b[off + i] = (byte)((b[off + i] ^ pattern) & 0xFF);
         }

    return numBytes;
     }
}
```

Socket Factories con Java RMI

XorInputStream.java

```
public int read(byte b[], int off, int len) throws
    IOException {
         int numBytes = in.read(b, off, len);

    if (numBytes <= 0)
         return numBytes;

        for(int i = 0; i < numBytes; i++) {
        b[off + i] = (byte)((b[off + i] ^ pattern) & 0xFF);
         }

    return numBytes;
     }
}
```

Socket Factories con Java RMI

- **XorOutputStream.java**

```
package examples.rmisocfac;

import java.io.*;

class XorOutputStream extends FilterOutputStream {

    private final byte pattern;

    public XorOutputStream(OutputStream out, byte pattern) {
        super(out);
        this.pattern = pattern;
    }

     public void write(int b) throws IOException {
       out.write((b ^ pattern) & 0xFF);
    }
}
```

Socket Factories con Java RMI

- **XorServerSocket.java**

```
package examples.rmisocfac;

import java.io.*;
import java.net.*;
class XorServerSocket extends ServerSocket {

    private final byte pattern;

    public XorServerSocket(int port, byte pattern) throws IOException
    {
        super(port);
        this.pattern = pattern;
    }

    public Socket accept() throws IOException {
        Socket s = new XorSocket(pattern);
        implAccept(s);
        return s;
    }
}
```

Socket Factories con Java RMI

- **XorSocket.java**

```
package examples.rmisocfac;

import java.io.*;
import java.net.*;

class XorSocket extends Socket {

    private final byte pattern;

    private InputStream in = null;

    private OutputStream out = null;

    public XorSocket(byte pattern)
        throws IOException
    {
        super();
        this.pattern = pattern;
    }
```


Socket Factories con Java RMI

XorSocket.java

```
public XorSocket(String host, int port, byte pattern)
        throws IOException
    {
        super(host, port);
        this.pattern = pattern;
    }

    public synchronized InputStream getInputStream() throws
    IOException {
        if (in == null) {
            in = new XorInputStream(super.getInputStream(), pattern);
        }
        return in;
    }

    public synchronized OutputStream getOutputStream() throws
    IOException {
        if (out == null) {
            out = new XorOutputStream(super.getOutputStream(),
    pattern);
        }
        return out;    }}
```

Socket Factories con Java RMI

- PASO2: Implementar un ***RMIClientSocketFactory*** personalizado.

```
package examples.rmisocfac;

import java.io.*;
import java.net.*;
import java.rmi.server.*;

public class XorClientSocketFactory
    implements RMIClientSocketFactory, Serializable {
    private byte pattern;
    public XorClientSocketFactory(byte pattern) {
     this.pattern = pattern;
    }

    public Socket createSocket(String host, int port)
     throws IOException
    {
     return new XorSocket(host, port, pattern);
    }
    public int hashCode() {
     return (int) pattern;
    }
    public boolean equals(Object obj) {
     return (getClass() == obj.getClass() &&
          pattern == ((XorClientSocketFactory) obj).pattern);
    }
}
```


Socket Factories con Java RMI

- PASO3: Implementar un ***RMIServerSocketFactory*** personalizado.

```
package examples.rmisocfac;
import java.io.*;
import java.net.*;
import java.rmi.server.*;

public class XorServerSocketFactory
    implements RMIServerSocketFactory {

    private byte pattern;

    public XorServerSocketFactory(byte pattern) {
     this.pattern = pattern;
    }
    public ServerSocket createServerSocket(int port)
     throws IOException
    {
     return new XorServerSocket(port, pattern);
    }
    public int hashCode() {
     return (int) pattern;
    }
    public boolean equals(Object obj) {
     return (getClass() == obj.getClass() &&
          pattern == ((XorServerSocketFactory) obj).pattern);
    }
}
```

Socket Factories con Java RMI

Pasos adicionales:

4. Escribir un servidor que cree un objeto remoto y lo exporte empleando *RMIClientSocketFactory* and *RMIServerSocketFactory*.
5. Escribir un cliente que invoque el método remoto.

- PASO4: Implementar un servidor.

Interface

```
package examples.rmisocfac;

public interface Hello extends java.rmi.Remote {
    String sayHello() throws java.rmi.RemoteException;
}
```


Socket Factories con Java RMI

Servidor

```
package examples.rmisocfac;

import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class HelloImpl implements Hello {

    public HelloImpl() {}

    public String sayHello() {
        return  "Hello World!";
    }

    public static void main(String args[]) {

    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }

        byte pattern = (byte) 0xAC;
```

Socket Factories con Java RMI

```
try {

        HelloImpl obj = new HelloImpl();
        RMIClientSocketFactory csf = new XorClientSocketFactory(pattern);
        RMIServerSocketFactory ssf = new XorServerSocketFactory(pattern);
        Hello stub =
        (Hello) UnicastRemoteObject.exportObject(obj, 0, csf, ssf);


        LocateRegistry.createRegistry(2002);
        Registry registry = LocateRegistry.getRegistry(2002);
        registry.rebind("Hello", stub);
        System.out.println("HelloImpl bound in registry");

    } catch (Exception e) {
        System.out.println("HelloImpl exception: " + e.getMessage());
        e.printStackTrace();
    }
    }
}

-------------------------------------------------
static RemoteexportObject(Remote obj, int port, RMIClientSocketFactory csf,
    RMIServerSocketFactory ssf)
                Exports the remote object to make it available to receive
    incoming calls, using a transport specified by the given socket factory.
```


Socket Factories con Java RMI

Cliente

```
package examples.rmisocfac;

import java.rmi.*;
import java.rmi.registry.*;

public class HelloClient {

    public static void main(String args[]) {

     if (System.getSecurityManager() == null) {
         System.setSecurityManager(new SecurityManager());
     }

        try {
         Registry registry = LocateRegistry.getRegistry(2002);
            Hello obj = (Hello) registry.lookup("Hello");
            String message = obj.sayHello();
            System.out.println(message);

        } catch (Exception e) {
         System.out.println("HelloClient exception: " +
                            e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Socket Factories con Java RMI

PASO 1:

Compilar las clases remotas interface, client y server

```
javac -d . XorInputStream.java
javac -d . XorOutputStream.java
javac -d . XorSocket.java
javac -d . XorServerSocket.java
javac -d . XorServerSocketFactory.java
javac -d . XorClientSocketFactory.java
javac -d . Hello.java
javac -d . HelloClient.java
javac -d . HelloImpl.java
```

PASO 2:

Ejecutar rmic para generar stubs y skeletons.

```
rmic -d . examples.rmisocfac.HelloImpl
```


Socket Factories con Java RMI

PASO 3:

Lanzar el servidor

```
java -Djava.security.policy=policy
   examples.rmisocfac.HelloImpl
```

PASO 4:

Lanzar el cliente

```
java -Djava.security.policy=policy examples.rmisocfac.HelloClient
```

Comentarios-dudas frecuentes

Socket Factories con Java RMI

Diferencias entre:

```
registry.rebind("rmi://localhost/FlightServices:1099",new RemoteConnection()");
Y
Naming.rebind("rmi://localhost/FlightServices:1099",new RemoteConnection()");

------
Estructura de Naming.rebind:

    public static void rebind(String name, Remote obj)
        throws RemoteException, java.net.MalformedURLException
        {
        ParsedNamingURL parsed = parseURL(name);
        Registry registry = getRegistry(parsed);if (obj == null)
        throw new NullPointerException("cannot bind to null");
    registry.rebind(parsed.name, obj);
        }
```