Leccion 4 - Patrones habituales en Java RMI

Puesta en marcha del minilaboratorio y laboratorio sobre temas interesantes adicionales sobre Java RMI.

Tabla de contenidos

1 Presentación del minilaboratorio.

Esta lección contiene algunos de los patrones básicos que se utilizan en el trabajo con invocación de objetos remotos, plasmados en Java RMI.

La potencia del modelo de Java RMI puede verse con la facilidad con la que se implementan algunos modelos que serían realmente un autentico calvario para el diseño en plataformas que no disfrutaran de este tipo de tecnología. Existen muchos patrones susceptibles de ser utilizados en RMI, aunque aquí sólo presentaré algunos de entre los más sencillos, que sirven para ilustrar la potencia del paradigma, y para desarrollar en el alumno algunas habilidades elementales.

En la fase preliminar de esta lección se presentan tres patrones sencillos:

- Paso de valores como parámetros de procedimientos remotos.
- Varias interfaces remotas con una misma implementación.
- Utilización de métodos remotos como factoría de objetos remotos.

2 Caja: metodos remotos con parametros de usuario.

En este ejemplo, veremos cómo una clase remota es capaz de admitir métodos a los que se les pasa (y devuelven) objetos preparados con clases diseñadas por el programador. El ejemplo es muy sencillo y consiste en que en el servidor existe un objeto remoto que presenta tres métodos:

```
package cajaserver;

import ...;

/**
 * Almacena un dato accesible vía RMI.
 * (Versión incompleta)
 *
 * El dato persiste en el servidor hasta que un cliente lo sobreescribe.
 * @author cllamas
 * @param <T> es el tipo de dato que guardamos en la Caja
 */
public interface Caja<T extends ...> extends ... {
    /**
     * Sobreescribe el dato en la caja.
     * @param a el dato pasa a ser el contenido actual de la caja.
     * @throws ...
     */
    ... void pon(T a) ...;
    /**
     * Retira el contenido de la caja y la vacía.
     * @return el contenido actual de la caja.
     * @throws ...
     */
    ... T    quita()  ...;
    /**
     * Consulta el contenido de la caja.
     * @return el contenido actual de la caja.
     * @throws ...
     */
    ... T    lee() ...;
}
```

Como puedes ver, me reservo los puntos suspensivos en este fragmento de código, y en los sucesivos para que pongas en práctica tus habilidades. Es muy, pero que muy fácil, y si utilizas un IDE, más todavía.

Esta interfaz nos sorprende porque está preparada para almacenar datos de algún tipo!! pero no definimos ese tipo hasta el momento en que creamos el objeto remoto!! con lo cual tenemos una definición de interfaz genérica. Interesante ¿verdad? Aquí te presento la implementación.

```
package cajaserver;

import ...

/**
 * Implementación de la interfaz remota Caja.
 * @author cllamas
 * @param <T> Tipo de dato a almacenar en la Caja.
 */
public ... CajaImpl<T extends ...> extends ... implements ... {
    private T contenido = null;

    public CajaImpl() ... {
        super();
    }

    @Override
    public void pon(T a) ... {
        this.contenido = a;
    }

    @Override
    public T quita() ... {
        T x = contenido;
        contenido = null;
        return x;
    }
<
    @Override
    public T lee() ... {
        return contenido;
    }
}
```

Y para que te resulta más sencillo, incluyo un caso concreto de tipo de dato que vamos a guardar en la caja, "Contador".

```java
package util;

import ...

/**
 * Parte de un valor y permite su incremento.
 * @author cllamas
 */
public class Contador ... {
    private int valor;
    /**
     * Inicializa Contador con valor 0.
     */
    public Contador() {
        valor = 0;
    }
    /**
     * Inicializa Contador con un valor inicial.
     * @param inicial valor inicial del contador que puede ser negativo.
     */
    public Contador(int inicial) {
        valor = inicial;
    }
    /**
     * Incrementa el contador en una unidad y devuelve su contenido.
     * @return valor actual del contador tras el incremento.
     */
    public int inc() {
        return ++valor;
    }
    /**
     * Lee el contenido del contador.
     * @return valor actual del contador.
     */
    public int lee() {
        return valor;
    }
}
```

Y, como no podía ser menos, te paso también el lanzador del objeto remoto.

```java
package runserver;
```

```
...
...

/**
 * Lanza el servidor de Caja(s) de Contador(es)
 * @author cllamas
 */
public class RunCaja {
    public static void main(String[] args) {
        try {
            CajaImpl<Contador> cc = new CajaImpl<...>();

            Registry registro = LocateRegistry.createRegistry(...);

            registro.rebind(...);
            System.out.println("Objeto remoto enlazado");
        } catch (RemoteException re) {
            re.printStackTrace(System.err);
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Los clientes te los presento en la siguiente página, porque si no,corro el riesgo de indigestar la plataforma XD

2.1 Caja: los clientes.

Los clientes de esta aplicación son muy sencillos, y contiene cada uno de ellos una operación sencilla para que los ejecutes en el orden que desees para probar el comportamiento del servidor.

- El primero sirve para inicializar la Caja con un contador establecido a cierto valor. Primero crea un Contador, lo incrementa y lo guarda.

```
package cajaclientrmi;

import cajaserver.Caja;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import util.Contador;

/**
 * Introduce en Caja un elemento Contador con valor 1.
 * @author cllamas
 */
public class CajaClientRMIpon {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Contador c = new Contador();
        c.inc();
        System.out.println("Valor del nuevo contador: "+c.lee());
        try {
            Caja caja = (Caja)
Naming.lookup("rmi://localhost:1099/CajaRemota");
            caja.pon(c); /* !!!!! */

        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }

}
```

- El segundo sirve para ilustrar cómo se lee el contenido de la Caja de modo no destructivo (lee()). Como puede verse, lee el contador, lo incrementa y lo guarda en la caja.

```
package cajaclientrmi;

import cajaserver.Caja;
import java.rmi.Naming;
```

```
import java.rmi.Naming;
import java.rmi.NotBoundException;
import util.Contador;

/**
 * Lee el valor de la caja, lo incrementa y lo vuelve a volcar.
 * Si la caja está vacía se queja.
 * @author cllamas
 */
public class CajaClientRMIlee {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Contador c = null;
        try {
            Caja caja = (Caja)
Naming.lookup("rmi://localhost:1099/CajaRemota");
            c = (Contador) caja.lee();
            if (null != c) {
                c.inc();
                caja.pon(c);
                System.out.println("valor de la caja: "+
((Contador)caja.lee()).lee());
            } else
                System.out.println("La caja está vacía.");
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }

}
```

- y finalmente el tercero, sirve para consultar el elemento de la Caja de modo destructivo, mediante su mediante su extracción (quita()), con lo que la caja queda vacía.

```
package cajaclientrmi;

import cajaserver.Caja;
import java.rmi.Naming;
```

```
import java.rmi.NotBoundException;
import util.Contador;

/**
 * Quita el elemento de la caja.
 * @author cllamas
 */
public class CajaClientRMIquita {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Contador c = null;
        try {
            Caja caja = (Caja)
Naming.lookup("rmi://localhost:1099/CajaRemota");

            c = (Contador) caja.lee();
            if (null != c) {
                System.out.println("valor de la caja: "+c.lee());
                caja.quita();
                System.out.println("Caja vaciada.");
            } else
                System.out.println("La caja estaba vacía.");
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }

}
```

Adelante, pruébalo!

3 Hola: Dos interfaces sobre el mismo objeto.

Cada objeto representa un recurso del programa, una pequeña parte del estado de toda la aplicación. En el caso de las aplicaciones distribuidas (y no distribuidas) es muy común el que un mismo recurso

el caso de las aplicaciones distribuidas (y no distribuidas) es muy común el que un mismo recurso pueda presentarse de modo distinto frente a dos clientes con permisos o necesidades distintas, dando lugar a dos recursos distintos, soportados por una misma implementación.

En realidad, son *dos recursos distintos* (aunque relacionados), soportados por una misma implementación, pues aparecen como *dos servicios distintos*.

Veamos de qué forma tan sencilla puede hacerse esto en Java RMI. Para ello planteo un sistema donde cada cliente normalmente se presenta a la plataforma mediante la interfaz "Hola", anunciando su nombre. La plataforma hace una lista de todos esos nombres que guarda internamente.

Otro tipo de cliente, puede estar interesado en conocer la lista de usuarios registrados, y lo hace mediante la interfaz HolaRegistro. Se que es un ejemplo muy tonto, pero no lo es tanto, cuando pensamos que en Java se puede restringir el acceso a las interfaces en función de la política de seguridad vigente en el sistema, con lo cual tendríamos dos tipos de usuarios vigilados por AccessControl (una clase central del gestor de seguridad de Java).

Veamos la Interfaz Hola.

```
package holaServer;

import ...

public ... Hola ... {
    String saluda(String nombre) ... ;
}
```

Vemos que no tiene más que un sencillo método que sirve para registrar el nombre de un supuesto usuario, en el servidor.

HolaRegistro, sólo tiene un método, y cualquiera podría pensar que para ese viaje no se necesitan esas alforjas ;) pero piensa que de alguna forma podremos restringir el uso de esta interfaz mediante otros mecanismos. Si este método hubiera estado en la misma interfaz, hubiera sido más difícil, aunque, preveo que no imposible.

```
package holaServer;

import java.util.Map;
import ....;

public interface HolaRegistro ... {
    Map<String,String> lista() ...;
}
```

La implementación no puede ser más sencilla. En este caso se utiliza la tabla Hash para guardar un string con la IP del cliente como campo clave, y el nombre del cliente como el valor. Inspecciona el código con atención.

```
package holaServer;


public class HolaImpl ... implements Hola, HolaRegistro {
   private Map<String,String> listado=null;
```

```
    public HolaImpl() throws RemoteException {
        super();
        listado = new HashMap<String,String>();
    }

    public String saluda(String nombre) ... {
        String ip=null;
        try {
            ip = this.getClientHost();
        } catch (ServerNotActiveException snae) {
            snae.printStackTrace(System.err);
        }
        listado.put(ip, nombre);
        return "Hola, "+nombre+" !";
    }

    public Map<String,String> lista() {
        return listado;
    }
}
```

El programa lanzador es bien sencillo, sólo registra un objeto HelloImpl. Será en los clientes donde haremos la distinción.

```
package holaServer;


public class RunHola {

    public static void main(String [ ] args) {
       try {
          HolaImpl oRemoto = new HolaImpl();
          Registry registro = ....getRegistry("localhost");
          registro.rebind("ObjetoHola", oRemoto);

          System.err.println("Servidor preparado");
       } catch (Exception e) {
          System.err.println("Excepción del servidor: "+e.toString());
          e.printStackTrace();
       }
    }
}
```

Este cliente, es de prueba, y no hace más que invocar métodos según dos interfaces. Como la ip es la misma si utilizamos la misma máquina, prueba a correr el ejemplo con un compañero de laboratorio para ver que funciona, o lánzalo en diversas máquinas virtuales.

```
package holaCliente;

import java.util.Map;
import java.rmi.Naming;
import holaServer.Hola;
import holaServer.HolaRegistro;
```

```
public class Cliente {

   public static void main(String [ ] args) {
      String nombre = (args.length < 1) ? "anonimo" : args[0];
      try {
         Object obj = Naming.lookup("ObjetoHola");

         Hola oHola = (Hola) ...;
         HolaRegistro oHolaRegistro = (HolaRegistro) obj;

         String respuesta = oHola.saluda(nombre);
         System.out.println("[Respuesta: "+respuesta+"]");

         Map<String,String> ... ;
         listado = oHolaRegistro.lista();

         for (String clave : listado.keySet()) {
             String valor = listado.get(clave);
             System.out.println("< "+clave+" , "+valor+" >");
         }
      } catch (Exception e) {
         System.err.println("<Cliente: Excepcion: "+e);
         e.printStackTrace();
      }
   }
}
```

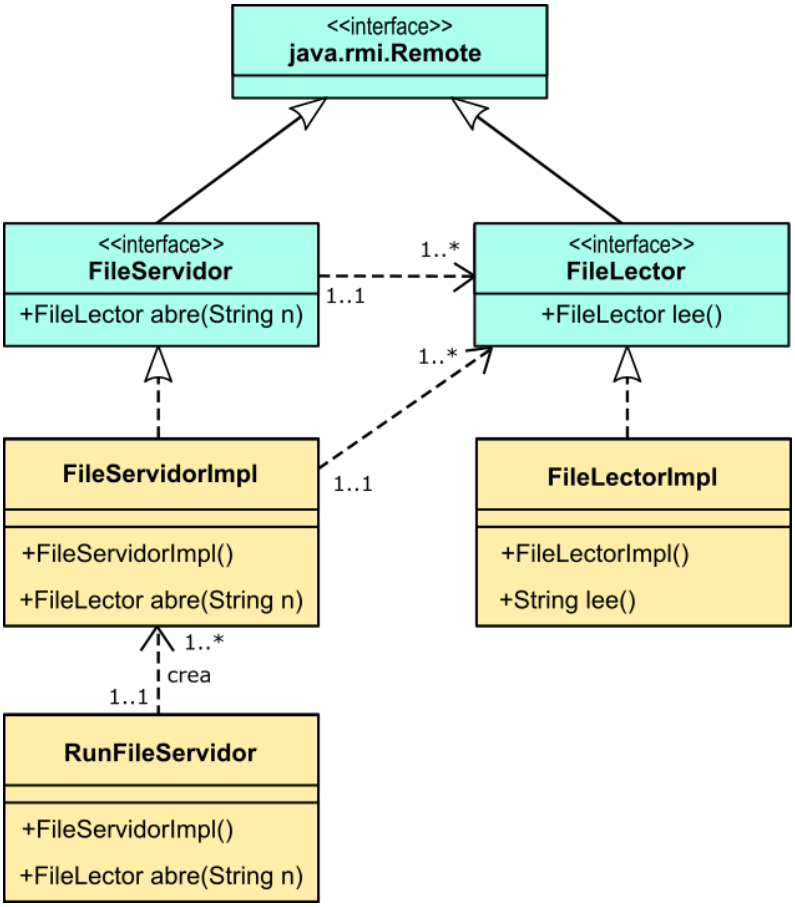
4 FileServer: objetos para administrar archivos.

En este caso, utilizaremos una construcción típica de métodos que fabrican objetos remotos. Consiste en un gestor de archivos preparado para que leamos archivos de un servidor remoto. Para cada petición de un cliente, crearemos un objeto remoto que alimentará al cliente con líneas a petición del cliente. Un

mismo archivo puede estar siendo servido para varios clientes mediante su respectivo objeto remoto, que conserva el estado de lectura en cada uno de ellos. Interesante, ¿verdad?

Pues el diseño y la implementación no pueden ser más sencillos.

Un diagrama de clases posible es el siguiente:



La interfaz principal es "FileServidor", que dispone de un método "abre()" al que se le proporciona el nombre del archivo que se desea leer. Si tiene éxito su apertura, se devuelve al cliente una referencia remota a un objeto que sirve para leer cada línea secuencialmente. Un aspecto interesante, es que tanto abre como lee pueden lanzar al cliente las excepciones típicas de la apertura de archivo y la lectura de archivos, con lo que se preserva la semántica de su uso normal sin RMI. Todas estas excepciones están en la plataforma cliente, con lo que no hay que tener en cuenta excepciones adicionales de usuario.

Como verás la interfaz FileServidor es muy sencilla.

```
package fileserver;


/**
 * Interfaz que construye un objeto remoto FileLector para leer archivos de te
xto.
 * @author cllamas
 */
... FileServidor extends Remote {
    public FileLector abre(String nombre) throws ..., FileNotFoundException;
}
```

Y la interfaz del lector de archivos también. (y los nombres de las clases XD)

```
package fileserver;

import java.io.IOException;

/**
```

```
 * Interfaz para leer secuencialmente las líneas de un archivo de texto.
 * @author cllamas
 */
public ... FileLector ... {
    /**
     * Utiliza BufferedReader.readline() para leer líneas del archivo.
     * @return String con la línea actual del archivo.
     * @throws ...
     * @throws IOException del mismo modo que BufferedReader->readLine()
     */
    String leeLinea() ... ..., IOException;
}
```

Las implementaciones no pueden ser más sencillas. FileServidorImpl es como sigue...

```
package fileserver;

/*
 * To change this license header, choose License Headers in Project Propertie
s.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
/**
 * Implementación de FileServidor
 * @author cllamas
 */
public class FileServidorImpl ... implements FileServidor {

    public FileServidorImpl() ... {
        super();
    }

    @Override
    public FileLector abre(String nombre) ... , FileNotFoundException {
        return new FileLectorImpl(nombre);
    }

}
```

Y FileLectorImpl es también sencillita...

```
package fileserver;


/**
 *
 * @author cllamas
 */
public class FileLectorImpl ... {
```

```
public class FileLectorImpl ... {
    private final BufferedReader br;
    public FileLectorImpl(String nombre) throws ..., FileNotFoundException {
        super();
        this.br = new BufferedReader(new FileReader(nombre));
    }

    @Override
    public String leeLinea() throws ..., IOException {
        return br.readLine();
    }
}
```

El lanzador sorprende por su sencillez, pues sólo es necesario registrar un objeto, cuando en el sistema aparecerán muchos más según se vayan abriendo archivos a petición de los clientes.

```
package runfileserver;

/**
 * Lanza el objeto remoto ObjetoFileServidor de tipo FileServidor.
 * @author cllamas
 */
public class RunFileServidor {
    public static void main(String[] args) {
        try {
        FileServidor cc = new FileServidorImpl();

        Registry registro = LocateRegistry....(1099);

        registro.rebind("...", cc);
        System.out.println("Objeto remoto 'ObjetoFileServidor' enlazado");
        } catch (RemoteException re) {
            re.printStackTrace(System.err);
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Para finalizar la foto, veamos en la siguiente página el cliente.

4.1 FileServer: el cliente

El programa cliente obtiene un stub del registro del host del servidor, busca la referencia del objeto remoto por su nombre concreto en el registro, y después invoca el método sayHello sobre el objeto, usando el stub. He aquí un código fuente de cliente modernizado:

```
package fileservercliente;


/**
 * Cliente de prueba de FileServer
 * @author cllamas
 */
public class FileServerCliente {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        String nombreArchivo = "Hola.txt";
        FileLector fl;
        FileServidor fs;
        String linea;

        try {
            fs = (FileServidor) Naming.lookup("rmi://localhost:.../ObjetoFileS
ervidor");

            try {
                fl = fs.abre(nombreArchivo);
                try {
                    while (null != (linea = fl.leeLinea())) {
                        System.out.println(nombreArchivo+ ": "+linea);
                    }
                } catch (EOFException ioe) {
                    System.err.println("Archivo finalizado con EOFException");
                } catch (Exception ex) {
                    ex.printStackTrace(System.err);
                }
            } catch (FileNotFoundException fnfe) {
                System.err.println("El archivo " + nombreArchivo + " no exist
e.");
            } catch (Exception ex) {
                ex.printStackTrace(System.err);
            }
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Si quieres además un par de archivos para probar, puedes hacerlo con cualquier archivo de texto que contenga unas pocas líneas. Y si lo deseas, puedes complicar el cliente para que lea líneas del servidor remoto poco a poco según pulsas en lugar de leerlas todas a la vez.

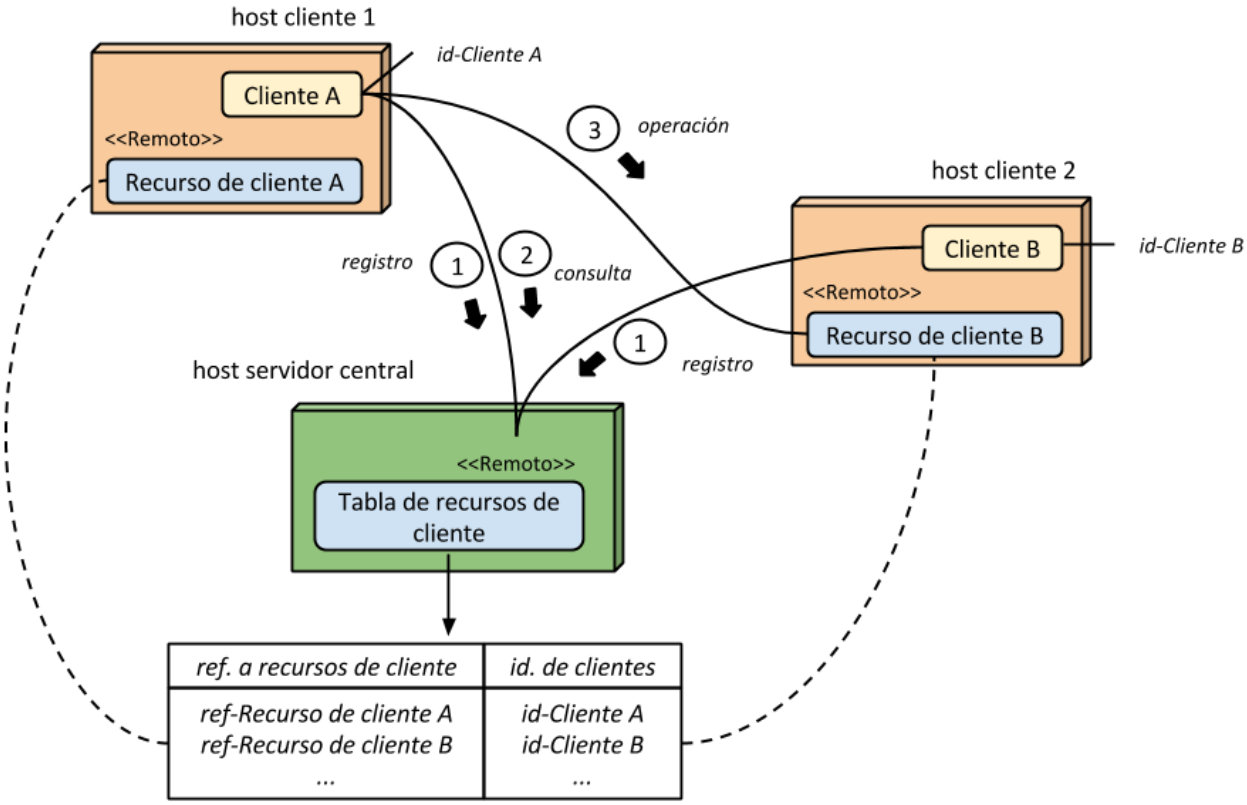
Espero que los ejemplos no te hayan resultado muy complicados, y procura que no te quede ninguno sin funcionar. Nos vemos en la lección 4. ;)

5 Sugerencias para el laboratorio.

Existen patrones mucho más elaborados que los que acabo de presentarte, y gran parte de ellos comienzan planteando un escenario en el que los clientes presentan objetos remotos para que el resto de los procesos del sistema realicen operaciones con ellos, entre los que tenemos:

- un servidor central, que guarda la lista de objetos remotos de los clientes, sirviendo de directorio, y

- los clientes del sistema que pueden efectuar invocaciones a los objetos presentes en los clientes para obtener acceso a algún tipo de recurso deslocalizado.



La figura refleja este modo de operación. ¿Eres capaz de imaginar aplicaciones para un patrón de este tipo? Te sugiero algunos:

- Compartición de archivos entre clientes.
- Chatrooms alojados en clientes.
- Notificación de eventos por multidifusión.

Espera algo muy sencillo, pero en esta línea para la lección 4. ;)