Servizi RMI hot-swap.

RMI

RMI è un'astrazione della comunicazione di rete. Dai pacchetti di byte spediti attraverso i socket si passa ai metodi invocati su oggetti remoti Java. Dal punto di vista dell'utente-programmatore un oggetto remoto differisce da un oggetto non remoto in ciò che il remoto è creato non attraverso l'invocazione di un costruttore ma tramite un registro RMI. La definizione di un oggetto remoto richiede qualche sforzo in più e, precisamente:

1. la definizione di un'interfaccia remota
2. la concretizzazione di quell'interfaccia in una classe che estende UnicastRemoteObject[1]
3. la registrazione di un'istanza di quella classe in un registro RMI

L'interfaccia remota.

A conti fatti un oggetto remoto ha due contratti, vale a dire due gruppi di capacità dichiarate: un contratto locale ed uno remoto. Il contratto remoto è la visione che l'oggetto offre di sé agli utenti remoti. Il contratto remoto deve essere dichiarato in un'interfaccia figlia di java.rmi.Remote.

```
package server;

import java.rmi.*;

public interface Service extends Remote {

	public String execute(String command) throws RemoteException;
}
```

Per essere remota un'interfaccia, oltre ad estendere java.rmi.Remote, deve dichiarare solo metodi remoti[2]. Tra i metodi dichiarati nell'interfaccia che estende java.rmi.Remote sono remoti tutti quelli che dichiarino il rilascio di un'eccezione di tipo – o supertipo di – java.rmi.RemoteException.

L'oggetto remoto.

L'oggetto remoto è l'istanza di una classe che concretizza un'interfaccia remota.

```
package server;

import java.rmi.*;
import java.rmi.server.*;

public class ServiceImpl extends UnicastRemoteObject implements Service {

	public ServiceImpl() throws RemoteException {}

	public String execute(String command) {
		return command.toUpperCase();
	}

}
```

Un metodo remoto può fare tutto ciò che si desidera. Il solo limite sta nel tipo di argomenti che può ricevere e nel tipo di valori che può restituire: devono essere tipi serializzabili. Un tipo serializzabile è un tipo classe o interfaccia che estende java.io.Serializable o java.io.Externalizable. La necessità

1 O Activatable, categoria di cui non ci occuperemo.
2 Un grazie a lovaz per aver segnalato l'inesattezza del paragrafo nella sua versione precedente.

che i parametri e i valori restituiti siano serializzabili deriva dall'attività svolta dietro le quinte dell'invocazione remota di metodi. La parte del programma che invoca un metodo remoto è a tutti gli effetti un client che comunica certi dati attraverso un socket. I dati inviati sono i parametri dell'invocazione e devono essere convertiti in pacchetti di byte per l'invio attraverso la rete. La conversione subita dall'oggetto-parametro è la serializzazione Java. Lo stesso accade per il valore che l'oggetto remoto restituisce. Il valore restituito viene convertito in un pacchetto di byte e spedito attraverso la rete all'invocante e la conversione è realizzata adottando la serializzazione degli oggetti Java.

Il registro RMI.

L'accesso ai servizi offerti da un oggetto remoto è condizionato alla registrazione di quest'ultimo presso un registro RMI. Il registro RMI è una mappa che mantiene una coppia di riferimenti stringa-oggetto. I client accedono ad un oggetto remoto esistente in un registro RMI usando una stringa equivalente alla chiave con cui l'oggetto è stato registrato.

```
package server;

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

public class Main {
	public static void main(String[] args) throws Throwable {
		Registry registry = LocateRegistry.getRegistry();
		registry.rebind("service", new ServiceImpl());
	}
}
```

Il client.

L'accesso al servizio offerto da un oggetto remoto richiede una connessione al registro RMI che possiede quell'oggetto. Ottenuto il riferimento al registro, l'accesso all'oggetto remoto passa per una semplice richiesta nominale.

```
package client;

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

public class Main {
	public static void main(String[] args) throws Throwable {
		Registry registry = LocateRegistry.getRegistry("localhost", 1099);
		Service service = (Service)registry.lookup("service");
		System.out.println(service.execute("hello remote world!"));
	}
}
```

1099 è il valore della porta predefinita del registro RMI attivabile eseguendo il programma rmiregistry incluso nel JDK di Sun Microsystem.

Hot-Swap di classi.

Vogliamo sostituire una definizione di classe con un'altra avente lo stesso nome ma contenuto diverso mentre la JVM è in funzione. Definiamo un supertipo che contenga la dichiarazione dei metodi definiti nella nostra classe “hot swap”.

```
package hotswap;

public interface HotSwapService {
```

```
        Object execute(Object args);
}
```

Ora creiamo il programma principale che si occupa di creare un'istanza di questo HotSwapService andando a pescarne la definizione da una cartella predefinita.

```
package hotswap;

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

public class Main {
        private static URL[] serviceFolders = new URL[1];

        public static void main(String[] args) throws Throwable {
                String servicesAddress = "file:/x:/waste/hotswap/hotswapservices/";
                serviceFolders[0] = new URL(servicesAddress);
                while(!printRequest().equalsIgnoreCase("exit")) {
                        HotSwapService service = loadHotSwapService();
                        System.console().printf("Esecuzione: %s%n", service.execute("hello world"));
                }
        }

        private static String printRequest() {
                System.console().printf("Premere invio per eseguire il servizio. Digitare exit per terminare.%n");
                return System.console().readLine();
        }

        private static HotSwapService loadHotSwapService() throws Throwable {
                URLClassLoader classLoader = new URLClassLoader(serviceFolders);
                return (HotSwapService)classLoader.loadClass("services.Service").newInstance();
        }
}
```

Il metodo loadHotSwapService usa un'istanza di URLClassLoader per creare riflessivamente un'istanza di HotSwapService. URLClassLoader si distingue dal ClassLoader di sistema per il semplice fatto di poter essere puntato in direzione di una o più cartelle o file archivio jar. Il punto centrale del metodo loadHotSwapService è che ogni sua invocazione restituisce un'istanza che è sempre figlia di una classe diversa da ogni altra già presente nella JVM anche se il nome binario – services.Service – è sempre lo stesso. L'effetto è prodotto da una regola del linguaggio di programmazione Java secondo cui due classi sono diverse se pur avendo lo stesso nome pienamente qualificato esse siano state caricate da ClassLoader diversi. Se non adottassimo questo accorgimento e usassimo il ClassLoader di sistema per caricare la classe services.Service noi non otterremmo l'aggiornamento della definizione in memoria perchè la classe services.Service risulterebbe, dopo il primo caricamento, già presente nella JVM. Proseguiamo creando una prima versione di HotSwapService in una classe Service appartenente al package services compilata nella cartella [x:/waste/hotswap/hotswapservices/].

```
package services;

import hotswap.HotSwapService;

public class Service implements HotSwapService {

        public Object execute(Object args) {
                return String.valueOf(args).toUpperCase();
        }
}
```

Dopo aver compilato questa classe, l'esecuzione del programma principale genera un output simile a questo:

```
X:\waste\hotswap>java hotswap.Main
Premere invio per eseguire il servizio. Digitare exit per terminare.

Esecuzione: HELLO WORLD
Premere invio per eseguire il servizio. Digitare exit per terminare.
_
```

Lasciando in esecuzione il programma modifichiamo la definizione della classe Service.

```
package services;

import hotswap.HotSwapService;
```

```
public class Service implements HotSwapService {

	public Object execute(Object args) {
		String text = String.valueOf(args);
		int len = text.length();
		return "La lunghezza della stringa " + text + " è di " + len + " caratteri.";
	}
}
```

Dopo aver compilato la nuova definizione premiamo il tasto invio nella console di del programma principale. L'output rifletterà la mutazione appena introdotta.

```
X:\waste\hotswap>java hotswap.Main
Premere invio per eseguire il servizio. Digitare exit per terminare.

Esecuzione: HELLO WORLD
Premere invio per eseguire il servizio. Digitare exit per terminare.

Esecuzione: La lunghezza della stringa hello world è di 11 caratteri.
Premere invio per eseguire il servizio. Digitare exit per terminare.
```

L'applicazione per la fornitura di servizi remoti hot-swap[3].

Il programma dimostrativo che esporremo combina RMI e caricamento dinamico di classi per ottenere un server in grado di aggiornare la definizione dei suoi sotto-servizi offerti senza interruzioni nella fornitura. Il programma è piuttosto semplice. Abbiamo un oggetto in possesso di un metodo remoto che restituisce un servizio. Il servizio è a sua volta un oggetto dotato di un metodo remoto. Quando il primo oggetto, chiamiamolo server, intercetta la richiesta di fornitura di un servizio esso determina se il servizio sia già stato caricato e, in caso affermativo, se la versione caricata sia anche la più aggiornata disponibile. Nel caso in cui il servizio non sia presente il server cerca di caricarlo. Se il servizio è presente ed è aggiornato il server si limita a restituirlo. Se il servizio è presente ma non è aggiornato il server usa il caricamento dinamico per rimpiazzare la definizione in memoria prima di restituirla al richiedente.

Gli oggetti remoti.

Gli oggetti remoti del programma sono due: il server e ogni servizio da questi offerto. Il server dispone di un unico metodo remoto, quello che restituisce un servizio remoto dato il suo nome.

```
package hsrmi;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Collection;


public interface HSRMIServer extends Remote {

    HSRMIService getService(String serviceName) throws RemoteException;
}
```

HSRMIService è il tipo – interfaccia remota dei servizi ed è fatto come segue.

```
package hsrmi;

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

public interface HSRMIService extends Remote {

    Object execute(Object args) throws RemoteException;
}
```

La classe HSRMIServerImpl concretizza l'interfaccia HSRMIServer in un oggetto remoto e sfrutta

3 Che suona un po' meno criptico di “provider di servizi remoti hot-swap”.

un terzo oggetto, ServiceFinder, per la fornitura vera e propria del servizio.

```
package hsrmi;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Collection;

public class HSRMIServerImpl extends UnicastRemoteObject implements HSRMIServer {
    private ServiceFinder serviceFinder = new ServiceFinder();

    public HSRMIServerImpl() throws RemoteException {}

    public HSRMIService getService(String serviceName) throws RemoteException {
        return serviceFinder.getService(serviceName);
    }
}
```

ServiceFinder è il cuore del programma ed è una versione un po' più raffinata del metodo loadHotSwapService visto in precedenza. I servizi hot-swap sono caricati a partire da una cartella predefinita, x:\rmiservices. ServiceFinder dispone di un unico metodo pubblico, getService(String).

```
    /*...parte di ServiceFinder.java... */
    public HSRMIService getService(String serviceName) {
        ServiceData data = serviceMap.get(serviceName);
        if(data == null) {
            return loadService(serviceName);
        } else if(data.isUpdated()) {
            return data.getService();
        } else {
                return updateService(data);
        }
    }
```

Nel codice serviceMap è il nome di un riferimento di tipo HashMap<String, ServiceData>. ServiceData è una struttura dati usata per conservare alcune informazioni utili al meccanismo di sostituzione dei servizi.

```
package hsrmi;

import java.io.*;
import java.net.*;
import java.util.Calendar;

public class ServiceData {
    private String serviceName;
    private HSRMIService service;
    private ClassLoader serviceClassLoader;
    private Calendar birthTime;
    private File classFile;

    public ServiceData(String serviceName, URL codebase) {
            try {
                File basePath = new File(codebase.toURI());
                String classFileName = serviceName.replace('.', File.separatorChar) + ".class";
                classFile = new File(basePath.getCanonicalPath() + File.separator + classFileName);
        } catch(Exception ex) {
                throw new RuntimeException(ex);
        }
        this.serviceName = serviceName;
    }

    public String getServiceName() {
        return serviceName;
    }

    public HSRMIService getService() {
        return service;
    }

    public void setService(HSRMIService service) {
        this.service = service;
    }

    public ClassLoader getServiceClassLoader() {
        return serviceClassLoader;
    }

    public void setServiceClassLoader(ClassLoader serviceClassLoader) {
        this.serviceClassLoader = serviceClassLoader;
    }

    public Calendar getBirthTime() {
        return birthTime;
    }

    public boolean isUpdated() {
            Calendar fileTime = Calendar.getInstance();
            fileTime.setTimeInMillis(classFile.lastModified());
            return fileTime.compareTo(getBirthTime()) <= 0;
    }
```

```
    public void setBirthTime(Calendar birthTime) {
        this.birthTime = birthTime;
    }

}
```

In particolare ServiceData, tramite il suo metodo isUpdated, è in grado di determinare se il servizio a cui fa riferimento sia o no aggiornato rispetto al file da cui ha avuto origine. Tornando al metodo getService di ServiceFinder:

```
    public HSRMIService getService(String serviceName) {
        ServiceData data = serviceMap.get(serviceName);
        if(data == null) {
            return loadService(serviceName);
        } else if(data.isUpdated()) {
            return data.getService();
        } else {
            return updateService(data);
        }
    }
```

si dovrebbe notare la logica di funzionamento. Se non esiste un ServiceData per la chiave serviceName, ServiceFinder tenta di caricare il servizio. Se il ServiceData esiste ed è aggiornato – rispetto al file di origine – allora getService restituirà il servizio già caricato. Se il ServiceData esiste ma non è aggiornato ServiceFinder tenta l'aggiornamento. Il caricamento del servizio crea un URLClassLoader usando la cartella predefinita già citata come punto di origine, ottiene da quel classloader un'istanza di HSRMIService e la immagazzina insieme alla data di creazione del file class corrispondente in un oggetto ServiceData. Prima di restituire il servizio, l'oggetto ServiceData che lo contiene viene registrato nella mappa serviceMap.

```
    public HSRMIService getService(String serviceName) {
        ServiceData data = serviceMap.get(serviceName);
        if(data == null) {
            return loadService(serviceName);
        } else if(data.isUpdated()) {
            return data.getService();
        } else {
                return updateService(data);
        }
    }

    private ServiceData loadServiceData(String serviceName) {
        ServiceData serviceData = new ServiceData(serviceName, SERVICES_DIRECTORIES[0]);
        try {
            URLClassLoader classLoader = new URLClassLoader(SERVICES_DIRECTORIES);
            serviceData.setServiceClassLoader(classLoader);
        } catch(Exception ex) {
            System.err.println("Cannot create URLClassLoader");
            ex.printStackTrace();
        }
        try {
                String classUrl = serviceName.replace('.', File.separatorChar) + ".class";
            URL classAddress = serviceData.getServiceClassLoader().getResource(classUrl);
            if(classAddress != null) {
                    File classFile = new File(classAddress.toURI());
                    Class<?> serviceClass = serviceData.getServiceClassLoader().loadClass(serviceName);
                    HSRMIService service = (HSRMIService)serviceClass.newInstance();
                    Calendar serviceBirthTime = Calendar.getInstance();
                    serviceBirthTime.setTimeInMillis(classFile.lastModified());
                    serviceData.setBirthTime(serviceBirthTime);
                    serviceData.setService(service);
            } else {
                    System.out.println("Class address not found");
            }
        } catch(Exception ex) {
            System.err.println("Cannot find class for service " + serviceName);
            ex.printStackTrace();
        }
        return serviceData.getService() == null ? null : serviceData;
    }
```

Nel caso in cui il servizio già caricato e richiesto non sia aggiornato, ServiceFinder tenta l'aggiornamento con il metodo updateService.

```
    private HSRMIService updateService(ServiceData serviceData) {
        ServiceData updatedData = loadServiceData(serviceData.getServiceName());
        if(updatedData != null) {
                serviceMap.put(updatedData.getServiceName(), updatedData);
        } else {
                updatedData = serviceMap.get(serviceData.getServiceName());
        }
        return updatedData.getService();
    }
```

L'aggiornamento differisce dal caricamento in ciò che l'eventuale fallimento si risolve nella persistenza del servizio precedente. Riproniamo la classe ServiceFinder nella sua interezza.

```
package hsrmi;

import java.io.File;
import java.net.*;
import java.util.*;

public class ServiceFinder {
    private final URL[] SERVICES_DIRECTORIES = new URL[1];
    private Map<String, ServiceData> serviceMap = new HashMap<String, ServiceData>();

    public ServiceFinder() {
            try {
                    SERVICES_DIRECTORIES[0] = new URL("file:/x:/rmiservices/");
            } catch(Exception ex) {
                    throw new RuntimeException(ex);
            }
    }

    public HSRMIService getService(String serviceName) {
        ServiceData data = serviceMap.get(serviceName);
        if(data == null) {
            return loadService(serviceName);
        } else if(data.isUpdated()) {
            return data.getService();
        } else {
                return updateService(data);
        }
    }

    private HSRMIService loadService(String serviceName) {
        ServiceData serviceData = loadServiceData(serviceName);
        if(serviceData != null) {
                serviceMap.put(serviceName, serviceData);
                return serviceData.getService();
        } else {
                return null;
        }
    }

    private ServiceData loadServiceData(String serviceName) {
        ServiceData serviceData = new ServiceData(serviceName, SERVICES_DIRECTORIES[0]);
        try {
            URLClassLoader classLoader = new URLClassLoader(SERVICES_DIRECTORIES);
            serviceData.setServiceClassLoader(classLoader);
        } catch(Exception ex) {
            System.err.println("Cannot create URLClassLoader");
            ex.printStackTrace();
        }
        try {
            String classUrl = serviceName.replace('.', File.separatorChar) + ".class";
            URL classAddress = serviceData.getServiceClassLoader().getResource(classUrl);
            if(classAddress != null) {
                    File classFile = new File(classAddress.toURI());
                    Class<?> serviceClass = serviceData.getServiceClassLoader().loadClass(serviceName);
                    HSRMIService service = (HSRMIService)serviceClass.newInstance();
                    Calendar serviceBirthTime = Calendar.getInstance();
                    serviceBirthTime.setTimeInMillis(classFile.lastModified());
                    serviceData.setBirthTime(serviceBirthTime);
                    serviceData.setService(service);
            } else {
                    System.out.println("Class address not found");
            }
        } catch(Exception ex) {
            System.err.println("Cannot find class for service " + serviceName);
            ex.printStackTrace();
        }
        return serviceData.getService() == null ? null : serviceData;
    }

    private HSRMIService updateService(ServiceData serviceData) {
        ServiceData updatedData = loadServiceData(serviceData.getServiceName());
        if(updatedData != null) {
                serviceMap.put(updatedData.getServiceName(), updatedData);
        } else {
                updatedData = serviceMap.get(serviceData.getServiceName());
        }
        return updatedData.getService();
    }
}
```

Il lato server del programma termina con una breve classe principale il cui unico scopo è quello di registrare un'istanza di HSRMIServerImpl presso il registro RMI predefinito, avviato esternamente.

```
package hsrmi;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Main {

    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry();
```

```
            registry.rebind("hsrmiserver", new HSRMIServerImpl());
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

}
```

Il client.

Per provare il sistema è necessario e sufficiente un client che richieda al server lo stesso servizio per un certo periodo di tempo. Ad esempio:

```
package client;

import hsrmi.HSRMIService;
import hsrmi.HSRMIServer;
import java.rmi.registry.*;

public class Main {

        public static void main(String[] args) throws Throwable {
                Registry registry = LocateRegistry.getRegistry("localhost", 1099);
                HSRMIServer server = (HSRMIServer)registry.lookup("hsrmiserver");
                for(int i = 0; i < 100; i++) {
                        HSRMIService service = server.getService("it.tukano.CharCounter");
                        if(service != null) {
                                System.out.println(service.execute("hello world"));
                        } else {
                                System.out.println("Service not available...");
                        }
                        Thread.sleep(25000);
                }
                System.out.println("End of test");
        }
}
```

Esecuzione.

Avviamo il registro rmi predefinito eseguendo rmiregistry da linea di comando.

```
X:\rmitut\server>start rmiregistry

X:\rmitut\server>
```

In seguito avviamo il server.

```
X:\rmitut\server>java hsrmi.Main
_
```

Dopodichè avviamo il client.

```
X:\rmitut\client>java -cp .;HSRMI.jar client.Main
Service not available...
```

Qui HSRMI.jar è un archivio jar contenente le interfacce remote HSRMIServer e HSRMIService, necessarie sia alla compilazione che all'esecuzione di client.Main. Ora che il server e il client sono in comunicazione possiamo introdurre una prima definizione del servizio it.tukano.CharCounter. Ad esempio:

```
package it.tukano;

import java.rmi.*;
import java.rmi.server.*;
import hsrmi.HSRMIService;

public class CharCounter extends UnicastRemoteObject implements hsrmi.HSRMIService {
        public CharCounter() throws RemoteException {}

        public Object execute(Object arg) throws RemoteException {
```

```
                String value = String.valueOf(arg);
                return "La lunghezza di \""+value+"\" e': " + value.length();
        }
}
```

Compilando questo file il client riceverà dal server il servizio desiderato.

```
X:\rmitut\client>java -cp .;HSRMI.jar client.Main
Service not available...
La lunghezza di "hello world" e': 11
```

Eliminando il file CharCounter.class dalla cartella dei servizi noteremo come il client continuerà a ricevere il servizio in forza dell'istanza conservata dal server in memoria. Cambiando la definizione di CharCounter e ricompilando...

```
package it.tukano;

import java.rmi.*;
import java.rmi.server.*;
import hsrmi.HSRMIService;

public class CharCounter extends UnicastRemoteObject implements hsrmi.HSRMIService {
        public CharCounter() throws RemoteException {}

        public Object execute(Object arg) throws RemoteException {
                return "Tecnologia fa rima con magia!";
        }
}
```

il client otterrà delle risposte che riflettono il mutamento intervenuto nel servizio.

```
X:\rmitut\client>java -cp .;HSRMI.jar client.Main
Service not available...
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
Tecnologia fa rima con magia!
Tecnologia fa rima con magia!
```

Lo stesso vale nel caso in cui il servizio cambi senza essere preventivamente eliminato.

```
package it.tukano;

import java.rmi.*;
import java.rmi.server.*;
import hsrmi.HSRMIService;

public class CharCounter extends UnicastRemoteObject implements hsrmi.HSRMIService {
        public CharCounter() throws RemoteException {}

        public Object execute(Object arg) throws RemoteException {
                return "David Copperfield ci fa un baffo ci fa!!!";
        }
}
```

```
X:\rmitut\client>java -cp .;HSRMI.jar client.Main
Service not available...
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
La lunghezza di "hello world" e': 11
Tecnologia fa rima con magia!
Tecnologia fa rima con magia!
Tecnologia fa rima con magia!
Tecnologia fa rima con magia!
Tecnologia fa rima con magia!
David Copperfield ci fa un baffo ci fa!!!
```