Azzolini Riccardo

Remote Method Invocation

1 Oggetti remoti

Un server RMI è un **oggetto remoto**.

Un oggetto remoto è descritto da un'**interfaccia remote**, cioè un'interfaccia:

- che estende `java.rmi.Remote`;
- i cui metodi devono dichiarare l'eccezione `java.rmi.RemoteException`.

Ad esempio:

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

public interface PowerService extends Remote {
    BigInteger square(int n) throws RemoteException;
    BigInteger power(int n1, int n2) throws RemoteException;
}
```

2 Passaggio di parametri

Come già anticipato, quando si invoca un metodo di un oggetto remoto, il passaggio dei parametri può avvenire in due modi diversi:

- Se si passa un oggetto locale o un valore di tipo primitivo, RMI fa automaticamente il marshalling e unmarshalling dei parametri, purché i tipi dei parametri (se non primitivi) siano *serializzabili*. Quindi, gli oggetti passati a (o restituiti da) un oggetto remoto devono implementare l'interfaccia `Serializable`.

  Ad esempio, considerando l'interfaccia remota

  ```
  interface MyInterface extends Remote {
      MyClass f(MyClass x) throws RemoteException;
  }
  ```

  se l'istanza di `MyClass` da passare come argomento a `f`, e/o l'istanza che `f` restituisce, è locale, allora è necessario che `MyClass` sia serializzabile:

```
class MyClass implements Serializable {
    private static final long serialVersionUID = 1;
    private int value;
    public MyClass(int value) { this.value = value; }
    public int getValue() { return value; }
}
```

- Se, invece, si passa un riferimento a un oggetto remoto, viene effettivamente trasmesso solo il riferimento, senza bisogno di serializzare l'oggetto.

3 Sviluppo di una semplice applicazione

I passi da seguire per lo sviluppo di una semplice applicazione RMI sono:

1. Definizione dell'interfaccia remota.
2. Definizione del codice dell'oggetto remoto, che deve:
   - implementare l'interfaccia remota;
   - estendere la classe `java.rmi.server.UnicastRemoteObject` e chiamare uno dei suoi costruttori, o, in alternativa, non estendere tale classe ma chiamare invece uno dei suoi metodi statici `exportObject`.
3. Definizione del codice del client, che deve:
   - richiedere al registry un riferimento all'oggetto remoto;
   - assegnare il riferimento a una variabile che ha come tipo l'interfaccia remota (la stessa utilizzata dal server).

4 Esempio: interfaccia remota e server

Come primo esempio, si realizza un'applicazione client–server nella quale il client invoca un metodo `sayHello` del server, che non ha parametri e restituisce una stringa. L'interfaccia remota è allora definita come segue:

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

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

Il codice del server, invece, è:

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

public class HelloImpl extends UnicastRemoteObject implements Hello {
    public HelloImpl() throws RemoteException {
        super();
    }

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

    public static void main(String[] args) throws Exception {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }

        HelloImpl obj = new HelloImpl();
        Naming.rebind("HelloServer", obj);
        System.out.println("Server ready");
    }
}
```

- La classe `HelloImpl` estende `UnicastRemoteObject` e implementa l'interfaccia remota `Hello`. Il fatto che il nome della classe sia il nome dell'interfaccia seguito da "`Impl`" è una convenzione spesso usata per gli oggetti remoti.
- Il costruttore chiama il costruttore di default della superclasse,[1] il che rende ogni istanza di `HelloImpl` un oggetto remoto.
- Nel main, per prima cosa si crea un `SecurityManager` (il cui scopo sarà spiegato in seguito). Successivamente, un'istanza di `HelloImpl` viene creata, e assegnata alla variabile `obj`, che è quindi un riferimento a un oggetto remoto. Infine, si usa il metodo `Naming.rebind` per pubblicare tale riferimento nel registry, assegnandogli l'etichetta `"HelloServer"`.
- Quando l'esecuzione del main finisce, la JVM rileva automaticamente che l'oggetto remoto deve rimanere in attesa di richieste dai client, e quindi il processo non termina.

[1] La chiamata `super()` si potrebbe anche omettere, poiché avvarrebbe implicitamente, ma è comunque obbligatorio definire esplicitamente il costruttore `HelloImpl()`, per dichiarare che esso solleva l'eccezione controllata `RemoteException`.

5 Sicurezza in RMI

Con RMI, è possibile caricare delle classi da remoto. Queste sono necessariamente considerate "non affidabili" (in quanto potrebbero contenere codice malevolo), quindi, di default, il caricamento delle classi remote è disabilitato (ogni tentativo di farlo genera un errore).

Per consentirlo, è necessario usare un opportuno **security manager**, che è un'istanza della classe `java.lang.SecurityManager` (o di una sua sottoclasse[2]). L'istruzione per usare un security manager è

```
System.setSecurityManager(new SecurityManager());
```

e deve essere eseguita prima che RMI abbia bisogno di caricare classi da remoto (dunque, tipicamente, come prima operazione).

Nota: Se si eseguono client e server sulla stessa macchina, è facile fare in modo che abbiano entrambi accesso a tutte le classi necessarie, quindi un `SecurityManager` è superfluo.

5.1 File di policy

Un security manager richiede un **file di policy** (policy file), in cui si specifica quali di tipi di operazioni possono eseguire le classi caricate dai vari codebase (basi, "depositi" di codice) sulla rete. Tale file può essere collocato in una qualsiasi directory e può avere un nome qualsiasi: la sua posizione viene specificata attraverso la proprietà `java.security.policy`, che si imposta nel comando usato per avviare la JVM:

```
java -Djava.security.policy=<nomefile> <programma> <argomenti...>
```

Il file di policy più semplice è quello che definisce una "global permission", cioè permette a tutte le classi caricate di eseguire tutti i tipi di operazioni. Il contenuto di tale file (che può essere scritto con un normale text editor) è il seguente:

```
grant {
    permission java.security.AllPermission;
};
```

Ovviamente, non è opportuno usare una policy di questo tipo in un ambiente di produzione.

In generale, il formato di una entry in un file di policy è:

[2] Il pacchetto `java.rmi` fornisce la sottoclasse `RMISecurityManager`, ma essa è deprecata, in quanto identica a `SecurityManager`. Si consiglia quindi di usare direttamente quest'ultima.

```
grant [codeBase "URL"] {
    permission permissionClassName "target", "action";
    ...
};
```

Essa definisce in modo dettagliato i permessi attribuiti alle classi caricate dal codebase specificato; se la specifica del codebase è omessa, i permessi si applicano a tutte le classi, indipendentemente dalla loro provenienza.

Un esempio di policy file più dettagliato è:

```
grant {
    permission java.net.SocketPermission "*:1024-65535",
        "connect,accept";
    permission java.net.SocketPermission "*:80", "connect";
};
```

- la prima `permission` permette sia di connettersi (`connect`) a qualsiasi host (perché è indicato l'asterisco al posto di un indirizzo IP) che di accettare connessioni (`accept`) da qualsiasi host sulle porte da 1024 a 65535;
- la seconda `permission` consente di connettersi a qualsiasi host sulla porta 80 (HTTP).

6 Registrazione di un oggetto remoto presso un registry

Per poter chiamare un metodo di un oggetto remoto, il client deve prima procurarsi un riferimento (uno stub) di tale oggetto. A tale scopo, il client cerca il nome del servizio su un registry, e quest'ultimo restituisce il riferimento corrispondente.

Il registry è un processo che solitamente gira sulla stessa macchina del server (ma non necessariamente: potrebbe invece essere su una macchina terza), e, di default, è in ascolto sulla porta 1099.

Perché un client possa trovare un servizio, è necessario il server l'abbia registrato sul registry, associandolo a un'etichetta nota ai client (il nome del servizio). Ciò può essere fatto chiamando il metodo `Naming.rebind`, come nell'esempio di server visto prima:

```
Naming.rebind("HelloServer", obj);
```

6.1 Metodi dell'interfaccia `Registry`

L'interfaccia `Registry` definisce i seguenti metodi:

- `bind`: associa un riferimento remoto a un nome;

  ```
  void bind(String name, Remote obj)
      throws RemoteException, AlreadyBoundException, AccessException;
  ```

- `rebind`: come `bind`, ma sovrascrive un'eventuale associazione già esistente;

  ```
  void rebind(String name, Remote obj)
      throws RemoteException, AccessException;
  ```

- `unbind`: elimina l'associazione di un riferimento remoto a un nome;

  ```
  void unbind(String name)
      throws RemoteException, NotBoundException, AccessException;
  ```

- `list`: fornisce l'elenco dei servizi di cui il registry è a conoscenza;

  ```
  String[] list()
      throws RemoteException, AccessException;
  ```

- `lookup`: restituisce il riferimento remoto associato a un nome.

  ```
  Remote lookup(String name)
      throws RemoteException, NotBoundException, AccessException;
  ```

`bind`, `rebind` e `unbind` sono usati dal server, mentre `list` e `lookup` vengono impiegati dal client.

La classe `Naming` fornisce delle versioni `static` di questi metodi, con la differenza che l'argomento `name` può opzionalmente essere un intero URL (invece del semplice nome del servizio richiesto), che permette di specificare l'indirizzo del registry.

7 Esempio: client ed esecuzione dell'applicazione

Il codice del client, che completa l'esempio iniziato prima, è il seguente:

```
import java.rmi.Naming;

public class HelloClient {
    public static void main(String[] args) throws Exception {
        System.setSecurityManager(new SecurityManager());
        Hello stub =
            (Hello) Naming.lookup("//" + args[0] + "/HelloServer");
        String response = stub.sayHello();
```

```
        System.out.println("response: " + response);
    }
}
```

1. Viene creato un security manager.
2. Si ottiene il riferimento all'oggetto server, chiamando l'operazione di `lookup`, alla quale viene passato come argomento un URL:
   - la specifica del protocollo (prima di `//`) deve essere omessa;
   - il nome dell'host / indirizzo IP è quello del registry, che qui viene letto come argomento a riga di comando;
   - la parte di path è il nome del servizio richiesto.
3. Si esegue l'invocazione remota del metodo `sayHello`, e si stampa la stringa che esso restituisce.

Adesso, avendo terminato la scrittura del codice, si può eseguire l'applicazione (dopo averla compilata, come al solito). Il procedimento per lanciarla richiede però alcuni passi in più del normale:

1. Si avvia il registry (sulla macchina server, se si stanno usando due macchine diverse), eseguendo il comando `rmiregistry`.
   - Se si vuole specificare una porta diversa da quella di default, la si può passare come argomento (ad esempio, `rmiregistry 2001`).
   - In genere, è comodo avviare il processo in background, mediante il comando `rmiregistry &` su Unix (e simili) oppure `start rmiregistry` su Windows, così da poter continuare a digitare altri comandi mentre il registry è in esecuzione.
2. Si avvia il server, indicando il percorso del file di policy. Ad esempio, se quest'ultimo è chiamato `policy` e si trova nella stessa cartella di `HelloImpl.class`:

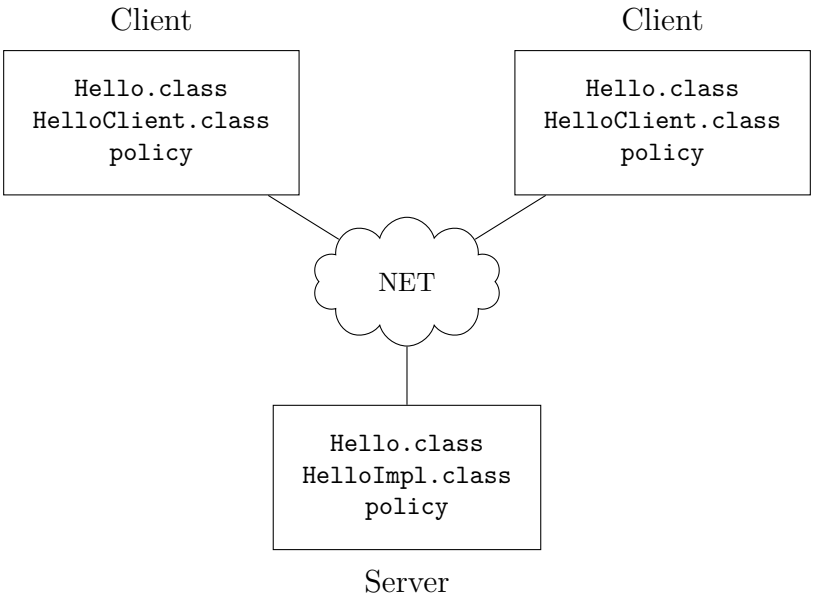
   ```
   java -Djava.security.policy=policy HelloImpl
   ```

3. Si avvia infine il client, indicando anche qui il percorso del file di policy, e passando come argomento l'indirizzo (o nome host) del server. Ad esempio:

   ```
   java -Djava.security.policy=policy HelloClient localhost
   ```

7.1 Deployment

Se si volessero eseguire server e client su macchine diverse, i vari file andrebbero distribuiti in questo modo:



- sulle macchine client si mette il bytecode dell'interfaccia remota `Hello`, quello della classe `HelloClient`, e il file di policy;
- sulla macchina server servono il bytecode dell'interfaccia remota, quello dell'oggetto remoto `HelloImpl`, e il file di policy.

Questo tipo di deployment è detto *statico*, perché si posizionano manualmente sulle varie macchine i file di cui esse hanno bisogno, quindi non è necessario il *dynamic class loading*.

Se, per fare dei test, si vuole simulare questo tipo di deployment su una sola macchina, è sufficiente mettere i file di client e server in due directory diverse.

Nota: Se si usano directory diverse, soprattutto per esempi più complessi di questo, il comando `rmiregistry` deve essere eseguito nella directory dove si sono messi i file del server, altrimenti potrebbero verificarsi degli errori.

8 Esempio di passaggio di un argomento

In questo secondo esempio, leggermente più complesso, il metodo dell'oggetto remoto che il client invoca ha un argomento di tipo non primitivo, che deve quindi essere serializzabile.

Per prima cosa, si definisce una semplice classe `Person`, che sarà quella passata come argomento, e viene perciò resa serializzabile nel solito modo:

```
import java.io.Serializable;

public class Person implements Serializable {
    private static final long serialVersionUID = 1;
    private String name;
    public Person(String name) { this.name = name; }
    public String getName() { return name; }
}
```

L'interfaccia remota viene modificata, aggiungendo una versione del metodo `sayHello` che riceve un argomento di tipo `Person`:

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

public interface HelloPerson extends Remote {
    String sayHello() throws RemoteException;
    String sayHello(Person p) throws RemoteException;
}
```

L'oggetto remoto deve implementare il nuovo metodo, nel quale riceve un oggetto di classe `Person` e accede (localmente) alle sue proprietà, che usa per costruire la stringa da rimandare al client. Per il resto, il codice rimane uguale a prima, ad eccezione del nome del servizio, che da `"HelloServer"` diventa `"HelloPersonServer"` (in modo che corrisponda ancora al nome dell'interfaccia remota):

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

public class HelloPersonImpl
        extends UnicastRemoteObject implements HelloPerson {
    public HelloPersonImpl() throws RemoteException {
        super();
    }

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

    public String sayHello(Person p) {
        return "Hello, " + p.getName();
```

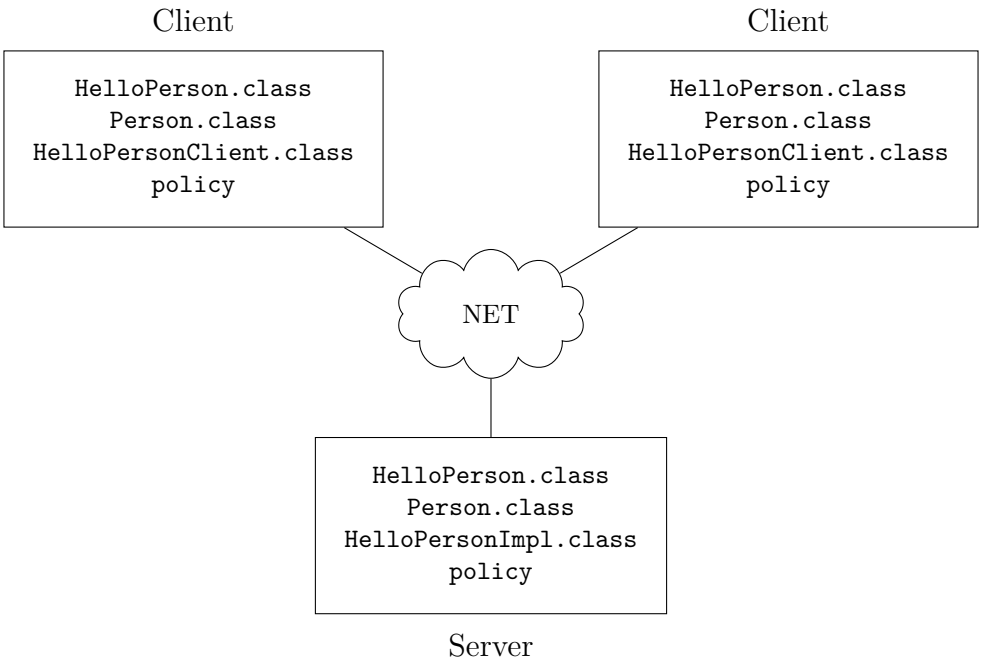
```
    }

    public static void main(String[] args) throws Exception {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }

        HelloPersonImpl obj = new HelloPersonImpl();
        Naming.rebind("HelloPersonServer", obj);
    }
}
```

Il client deve creare un oggetto `Person` e passarlo al nuovo metodo `sayHello` dell'oggetto remoto, mentre nel resto del codice cambiano solo i nomi dell'interfaccia remota e del servizio richiesto al registry:

```
import java.rmi.Naming;

public class HelloPersonClient {
    public static void main(String[] args) throws Exception {
        System.setSecurityManager(new SecurityManager());
        HelloPerson stub = (HelloPerson) Naming.lookup(
            "//" + args[0] + "/HelloPersonServer"
        );

        String response = stub.sayHello();
        System.out.println("response: " + response);

        Person someone = new Person("Caio Sempronio");
        response = stub.sayHello(someone);
        System.out.println("response: " + response);
    }
}
```

8.1 Deployment

Quando si esegue il deployment dell'applicazione, la nuova classe `Person` deve essere distribuita sia ai client che al server; per il resto, la dislocazione dei file rimane la stessa di prima:



9 Gestione alternativa del registry

Invece della classe `Naming`, per gestire il registry si può usare la classe `LocateRegistry`, che consente di ottenere un oggetto di tipo `Registry` (l'interfaccia già presentata prima), sul quale è possibile invocare metodi che RMI trasformerà in chiamate al processo registry. I metodi principali offerti da questa classe sono:

- diverse varianti di `getRegistry`,

  ```
  public static Registry getRegistry()
      throws RemoteException;
  public static Registry getRegistry(int port)
      throws RemoteException;
  public static Registry getRegistry(String host)
      throws RemoteException;
  public static Registry getRegistry(String host, int port)
      throws RemoteException;
  ```

  che permettono di ottenere un riferimento a un registry esistente, specificando opzionalmente l'host su cui esso si trova (che di default è `localhost`) e la porta sulla quale è in attesa (quella di default è la 1099);

- `createRegistry`, che permette di creare direttamente un registry sull'host corrente e sulla porta specificata,

```
public static Registry createRegistry(int port)
    throws RemoteException;
```

eliminando così la necessità di avviare separatamente un processo registry.

9.1 Esempio con `getRegistry`

Usando `LocateRegistry` al posto di `Naming` nell'esempio precedente, il codice dell'oggetto remoto diventa:

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

public class HelloPersonImpl
        extends UnicastRemoteObject implements HelloPerson {
    public HelloPersonImpl() throws RemoteException {
        super();
    }

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

    public String sayHello(Person p) {
        return "Hello, " + p.getName();
    }

    public static void main(String[] args) throws RemoteException {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }

        HelloPersonImpl obj = new HelloPersonImpl();
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind("HelloPersonServer", obj);
    }
}
```

Analogamente, il client diventa:

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

```
public class HelloPersonClient {
    public static void main(String[] args) throws Exception {
        System.setSecurityManager(new SecurityManager());

        Registry registry = LocateRegistry.getRegistry(args[0], 1099);
        HelloPerson stub =
            (HelloPerson) registry.lookup("HelloPersonServer");

        String response = stub.sayHello();
        System.out.println("response: " + response);

        Person someone = new Person("Caio Sempronio");
        response = stub.sayHello(someone);
        System.out.println("response: " + response);
    }
}
```

Una differenza importante è che, siccome l'indirizzo del registry è già stato specificato nella chiamata al metodo `getRegistry`, per il lookup si usa una semplice etichetta (il nome del servizio), non un URL.

9.2 Esempio con `createRegistry`

Se si vuole che il registry venga avviato automaticamente dal server, è sufficiente sostituire, nel codice di quest'ultimo, la chiamata a `getRegistry` con una a `createRegistry`:

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

public class HelloPersonImpl
        extends UnicastRemoteObject implements HelloPerson {
    // ...uguale a prima

    public static void main(String[] args) throws RemoteException {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }

        HelloPersonImpl obj = new HelloPersonImpl();
        Registry registry = LocateRegistry.createRegistry(1099);
```

```
        registry.rebind("HelloPersonServer", obj);
    }
}
```

In questo modo, lo stesso processo funge sia da server che da registry.

10 Deployment in ambiente distribuito

Quando si esegue il deployment di un'applicazione RMI in un ambiente distribuito, come già detto si hanno tipicamente server e registry su una macchina, e il client su un'altra.

Potrebbe capitare che il server invii al client degli oggetti di classi che quest'ultimo non conosce. Per permettere al client di procurarsi dinamicamente il codice di tali classi, si possono installare i file `.class` su un web server. Poi, quando si lancia il server RMI, bisogna indicare, con la proprietà `java.rmi.server.codebase`, l'URL dove si trovano le classi; ad esempio, se tale URL è `https://serverhost/stubsdir/`:

```
java -Djava.security.policy=policy
     -Djava.rmi.server.codebase=https://serverhost/stubsdir/
     ServerMain
```

Nota: In questo caso, il security manager è obbligatorio, mentre per l'uso in locale è superfluo.