

PSDDE

Programação em Sistemas Distribuídos

RMI

Remote Method Invocation

Daniel Cintra Cugler
danielcugler@ifsp.edu.br

O que veremos...

- RMI - Remote Method Invocation

O que é RMI?

- RMI: Remote Method Invocation
- É uma API Java que provê invocação remota de procedimentos
- RMI é uma tecnologia equivalente ao RPC (Remote Procedure Calls), mas voltada para objetos

RMI

- Suporta apenas chamadas entre JVMs (Java Virtual Machines). Portanto, roda apenas sobre a plataforma Java.
- Se desejado realizar a comunicação entre objetos que rodem em um contexto que não seja na JVM, é possível utilizar o CORBA.

RMI

- Uma aplicação RMI é frequentemente composta por dois programas diferentes, um servidor e um cliente. O servidor cria objetos remotos e faz referências a esses objetos disponíveis. Em seguida, ele é válido para clientes invocarem seus métodos sobre os objetos.
- O cliente executa referências remotas aos objetos remotos no servidor e invoca métodos nesses objetos remotos.
- O modelo de RMI fornece uma aplicação de objetos distribuídos para o programador. Ele é um mecanismo de comunicação entre o servidor e o cliente para se comunicarem e transmitirem informações entre si.

Exemplo

- Construiremos uma aplicação que permitirá a comunicação via objetos entre cliente/servidor.
- Embora tenhamos dois papéis definidos, os códigos referentes ao cliente e ao servidor ficarão no mesmo projeto.
- Criaremos uma interface que deverá ser comum ao cliente e ao servidor.

Exemplo








- O servidor disponibilizará dois métodos
 - `String helloWorld(String nome)`
 - `int soma(int n1, int n2)`

Atenção

- Garanta que você possui o JDK (Java Development Kit) instalado no seu computador.

Exemplo

- Em nosso exemplo, teremos 3 classes

- ▼  RMISimpleExample
 - ▶  JRE System Library [JavaSE-1.8]
 - ▼  src
 - ▼  com.example.rmi
 - ▶  Cliente.java
 - ▶  InterfaceRMI.java
 - ▶  Servidor.java

Exemplo

- Em nosso exemplo, teremos 3 classes

- ▼  RMISimpleExample

- ▶  JRE System Library [JavaSE-1.8]

- ▼  src

- ▼  com.example.rmi

- ▶  Cliente.java

- ▶  InterfaceRMI.java Começamos pela criação da ***interface***

- ▶  Servidor.java

Interface

InterfaceRMI

```
package com.example.rmi;

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

public interface InterfaceRMI extends Remote {

    public String helloWorld(String nome) throws RemoteException;

    public int soma(int n1, int n2) throws RemoteException;

}
```

Interface

InterfaceRMI

```
package com.example.rmi;

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

public interface InterfaceRMI extends Remote {








    public String helloWorld(String nome) throws RemoteException;

    public int soma(int n1, int n2) throws RemoteException;

}
```

Não esquecer!!!

Exemplo

- ▼  RMISimpleExample
 - ▶  JRE System Library [JavaSE-1.8]
 - ▼  src
 - ▼  com.example.rmi
 - ▶  Cliente.java
 - ▶  InterfaceRMI.java
 - ▶  Servidor.java ←

Classe Servidor – parte 1/2

```
package com.example.rmi;

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

public class Servidor extends UnicastRemoteObject implements InterfaceRMI{

    private static final long serialVersionUID = 1L;

    protected Servidor() throws RemoteException {
        super();
    }

    @Override
    public String helloWorld(String nome) throws RemoteException {
        System.err.println("Recebida requisição de " + nome);
        return "Hello World, " + nome + "!";
    }

    @Override
    public int soma(int n1, int n2) throws RemoteException {








        return n1 + n2;
    }
}
```

Classe Servidor – parte 2/2

(continuação)

```
public static void main(String[] args){  
    try {  
        Naming.rebind("MeuServidor", new Servidor());  
        System.err.println("Servidor está pronto!");  
    } catch (Exception e) {  
        System.err.println("Erro: " + e.toString());  
        e.printStackTrace();  
    }  
}
```

Exemplo

- ▼  RMISimpleExample
 - ▶  JRE System Library [JavaSE-1.8]
 - ▼  src
 - ▼  com.example.rmi
 - ▶  Cliente.java ←
 - ▶  InterfaceRMI.java
 - ▶  Servidor.java

Cliente

```
package com.example.rmi;

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

import javax.swing.JOptionPane;

public class Cliente {

    private static InterfaceRMI objetoRemoto;

    public static void main(String[] args)
        throws MalformedURLException, RemoteException, NotBoundException {

        objetoRemoto = (InterfaceRMI) Naming.lookup("//localhost/MeuServidor");

        String nome = JOptionPane.showInputDialog("Insira seu nome: ");

        String respostaServidor = objetoRemoto.helloWorld(nome);

        System.out.println("Soma 5 + 6 = " + objetoRemoto.soma(5, 6));

        JOptionPane.showMessageDialog(null, respostaServidor);
    }
}
```

Executando o projeto

- Nesta primeira execução, ambos cliente e servidor rodarão no mesmo computador. Faremos outros exemplos onde cliente e servidor rodarão em computadores separados.

Executando o projeto

Passo 1

- Podemos compilar o projeto de duas maneiras
 - (a) Se tiver utilizando o Eclipse, vá no menu “Project → build project” (ou “project → clean”, mas garanta que a opção “build automatically” esteja marcada).
 - (b) Compilar diretamente na linha de comando. Para isso, via prompt entre no diretório “src” do projeto e rode **(se Linux ou Windows)**:
 - javac com/example/rmi/InterfaceRMI.java
 - javac com/example/rmi/Servidor.java
 - javac com/example/rmi/Cliente.java

Executando o projeto

Passo 2

- Executar o `rmiregistry`
 - **Entrar no diretório src**
 - Executar, via prompt: “rmiregistry” (se linux)
- Este serviço faz um *link* entre objetos remotos e nomes. Isto permite que clientes locais ou remotos possam procurar por objetos disponíveis e realizar invocação de métodos remotos.

Executando o projeto

Passo 2

- Executar o `rmiregistry`
 - **Entrar no diretório src**
 - Executar, via prompt: “start rmiregistry” (**se windows**)
- Este serviço faz um *link* entre objetos remotos e nomes. Isto permite que clientes locais ou remotos possam procurar por objetos disponíveis e realizar invocação de métodos remotos.

Executando o projeto

Passo 3

- Executar a aplicação **servidor**
- Entrar no diretório `src` e executar, via prompt
(válido para windows e linux):
 - `java com.example.rmi.Servidor`

Executando o projeto

Passo 4

- Executar a aplicação **cliente**
- Entrar no diretório src e executar, via prompt
(válido para windows e linux):
 - `java com.example.rmi.Cliente`

Comportamento do projeto

- Ao executar o cliente é pedido para informar o nome do usuário.
- Em seguida, é impresso no console do cliente a soma entre 5 e 6. Esta soma é calculada no servidor.
- Além disso, é exibido na tela uma mensagem “Hello World <NOME>” – esta mensagem é enviada pelo servidor, após o cliente invocar o método remoto *helloWorld*.

Entendendo o código

Interface InterfaceRMI

- Possui interface para os métodos que serão disponibilizados para serem acessados remotamente.

```
package com.example.rmi;

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

public interface InterfaceRMI extends Remote {

    public String helloWorld(String nome) throws RemoteException;

    public int soma(int n1, int n2) throws RemoteException;

}
```

Entendendo o código

Classe Servidor (PARTE 1)

```
package com.example.rmi;

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

public class Servidor extends UnicastRemoteObject implements InterfaceRMI{

    private static final long serialVersionUID = 1L;

    protected Servidor() throws RemoteException {
        super();
    }

    @Override
    public String helloWorld(String nome) throws RemoteException {
        System.err.println("Recebida chamada de " + nome);
        return "Hello World, " + nome;
    }

    @Override
    public int soma(int n1, int n2) throws RemoteException {
        return n1 + n2;
    }
}
```

Classe utilizada para exportar objetos remotos e obter o *stub* que se comunica com o objeto remoto

Entendendo o código

Classe Servidor (PARTE 1)

```
package com.example.rmi;

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

public class Servidor extends UnicastRemoteObject implements InterfaceRMI{

    private static final long serialVersionUID = 1L;

    protected Servidor() throws RemoteException {
        super();
    }

    @Override
    public String helloWorld(String nome) throws RemoteException {
        System.err.println("Recebida chamada de helloWorld de " + nome);
        return "Hello World, " + nome;
    }

    @Override
    public int soma(int n1, int n2) throws RemoteException {
        return n1 + n2;
    }
}
```

Implementaremos os métodos que disponibilizamos em nossa interface

Entendendo o código

Classe Servidor (PARTE 1)

```
package com.example.rmi;

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

public class Servidor extends UnicastRemoteObject implements InterfaceRMI{

    private static final long serialVersionUID = 1L;

    protected Servidor() throws RemoteException {
        super();
    }

    @Override
    public String helloWorld(String nome) throws RemoteException {
        System.err.println("Recebida chamada de helloWorld com nome: " + nome);
        return "Hello World, " + nome;
    }

    @Override
    public int soma(int n1, int n2) throws RemoteException {
        return n1 + n2;
    }
}
```

SerialVersionUID é um identificador único. A JVM utiliza-o para comparar objetos após o processo de serialização e deserialização.

Entendendo o código

Classe Servidor (PARTE 1)

```
package com.example.rmi;

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

public class Servidor extends UnicastRemoteObject implements InterfaceRMI{

    private static final long serialVersionUID = 1L;

    protected Servidor() throws RemoteException {
        super();
    }

    @Override
    public String helloWorld(String nome) throws RemoteException {
        System.err.println("Recebida chamada de helloWorld com nome " + nome);
        return "Hello World, " + nome;
    }

    @Override
    public int soma(int n1, int n2) throws RemoteException {
        return n1 + n2;
    }
}
```

O "L" é para definir que o número um é para ser tratado como *long* e não *int*.

Entendendo o código

Classe Servidor (PARTE 1)

```
package com.example.rmi;

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

public class Servidor extends UnicastRemoteObject implements RemoteServidor {

    private static final long serialVersionUID = 1L;

    protected Servidor() throws RemoteException {
        super();
    }

    @Override
    public String helloWorld(String nome) throws RemoteException {
        System.err.println("Recebida requisição de " + nome);
        return "Hello World, " + nome + "!";
    }

    @Override
    public int soma(int n1, int n2) throws RemoteException {
        return n1 + n2;
    }
}
```

Implementação dos dois métodos que queremos disponibilizar para acesso remoto.

MI{

Entendendo o código

Classe Servidor (PARTE 2)

```
public static void main(String[] args){  
    try {  
        Naming.rebind("MeuServidor", new Servidor());  
        System.err.println("Servidor está pronto!");  
    } catch (Exception e) {  
        System.err.println("Erro: " + e.toString());  
        e.printStackTrace();  
    }  
}
```

Associa a instância “new Servidor()” com o nome passado no primeiro parâmetro.

Este nome será a base para clientes localizarem o servidor, assim como uma URI.

Entendendo o código

Classe Cliente

```
package com.example.rmi;

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

import javax.swing.JOptionPane;

public class Cliente {

    private static InterfaceRMI objetoRemoto;

    public static void main(String[] args)
        throws MalformedURLException, RemoteException, NotBoundException {

        objetoRemoto = (InterfaceRMI) Naming.lookup("//localhost/MeuServidor");

        String nome = JOptionPane.showInputDialog("Insira seu nome: ");

        String respostaServidor = objetoRemoto.helloWorld(nome);

        System.out.println("Soma 5 + 6 = " + objetoRemoto.soma(5, 6));

        JOptionPane.showMessageDialog(null, respostaServidor);
    }
}
```


Entendendo o código

Classe Cliente

```
package com.example.rmi;

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

import javax.swing.JOptionPane;

public class Cliente {

    private static InterfaceRMI objetoRemoto;

    public static void main(String[] args)
        throws MalformedURLException, RemoteException, NotBoundException {

        objetoRemoto = (InterfaceRMI) Naming.lookup("//localhost/MeuServidor");

        String nome = JOptionPane.showInputDialog("Insira seu nome: ");

        String respostaServidor = objetoRemoto.helloWorld(nome);

        System.out.println("Soma 5 + 6 = " + objetoRemoto.soma(5, 6));

        JOptionPane.showMessageDialog(null, respostaServidor);
    }
}
```

Utilizaremos esta variável para receber o objeto remoto

Entendendo o código

Classe Cliente

```
package com.example.rmi;
```

```
import java.net.MalformedURLException;
```

```
import java.rmi.Naming;
```

```
import java.rmi.NotBoundException;
```

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

```
import javax.swing.JOptionPane;
```

```
public class Cliente {
```

```
    private static InterfaceRMI objetoRemoto;
```

```
    public static void main(String[] args)
```

```
        throws MalformedURLException, RemoteException, NotBoundException {
```

```
        objetoRemoto = (InterfaceRMI) Naming.lookup("//localhost/MeuServidor");
```

```
        String nome = JOptionPane.showInputDialog("Insira seu nome: ");
```

```
        String respostaServidor = objetoRemoto.helloWorld(nome);
```

```
        System.out.println("Soma 5 + 6 = " + objetoRemoto.soma(5, 6));
```

```
        JOptionPane.showMessageDialog(null, respostaServidor);
```

```
    }
```

Retorna objeto remoto,
disponibilizado no servidor apontado
pela string passada como parâmetro

Entendendo o código

Classe Cliente

```
package com.example.rmi;

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

import javax.swing.JOptionPane;

public class Cliente {

    private static InterfaceRMI objetoRemoto;

    public static void main(String[] args)
        throws MalformedURLException, RemoteException, NotBoundException {

        objetoRemoto = (InterfaceRMI) Naming.lookup("//localhost/MeuServidor");

        String nome = JOptionPane.showInputDialog("Insira seu nome: ");

        String respostaServidor = objetoRemoto.helloWorld(nome);

        System.out.println("Soma 5 + 6 = " + objetoRemoto.soma(5, 6));

        JOptionPane.showMessageDialog(null, respostaServidor);
    }
}
```

Procedimentos remotos podem ser invocados como se estivessem rodando localmente

Exemplo 2

Exemplo 2

- Agora, vamos criar dois projetos – um para o cliente e outro para o servidor
 - Projeto *RMIExemplo2Cliente*
 - Projeto *RMIExemplo2Servidor*
- O servidor deverá disponibilizar o método ***long fatorialRecursivo(int n)*** que deverá calcular o fatorial recursivamente.

Exemplo 2 - Projeto *RMIExemplo2Servidor* Interface *MetodosRMI*

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface MetodosRMI extends Remote{  
    public long fatorialRecursivo(int n) throws RemoteException;  
}
```

Exemplo 2 - Projeto *RMIExemplo2Servidor*

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

Classe Servidor

public class Servidor extends UnicastRemoteObject implements MetodosRMI{
    private static final long serialVersionUID = 1L;

    protected Servidor() throws RemoteException {
        super();
    }

    @Override
    public long fatorialRecursivo(int n) throws RemoteException {
        if(n == 0)
            return 1;
        else
            return (n * fatorialRecursivo(n-1));
    }

    public static void main(String[] args) {
        try {
            Naming.rebind("ServidorFuncoesMatematicas", new Servidor());
            System.err.println("Servidor está pronto!");
        } catch (Exception e) {
            System.err.println("Erro: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Exemplo 2 - Projeto *RMIExemplo2***Cliente**

Interface ***MetodosRMI***

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface MetodosRMI extends Remote{  
    public long fatorialRecursivo(int n) throws RemoteException;  
}
```

Interface idêntica à criada no projeto Servidor

Exemplo 2 - Projeto *RMIExemplo2***Cliente**

Classe *Cliente*

```
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import javax.swing.JOptionPane;

public class Cliente {

    private static MetodosRMI objetoRemoto;

    public static void main(String[] args) throws RemoteException,
        MalformedURLException, NotBoundException{

        String ip_servidor = JOptionPane.showInputDialog("Qual o IP do servidor? ");

        objetoRemoto = (MetodosRMI) Naming.lookup("//" + ip_servidor +
                                                    "/ServidorFuncoesMatematicas");

        int n = Integer.valueOf(JOptionPane.showInputDialog("Digite um número: "));

        long fatorial = objetoRemoto.fatorialRecursivo(n);

        JOptionPane.showMessageDialog(null, "Fatorial de " + n + " = " + fatorial);
    }
}
```

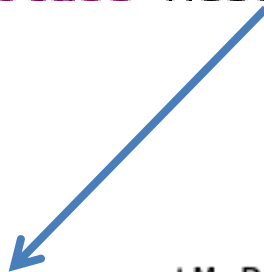
Executando o projeto

- Siga os mesmos procedimentos utilizados no primeiro exemplo:
 - Compilar códigos do projeto Servidor (como não utilizamos um pacote específico, entrar no diretório src e compilar seguindo o modelo *“javac Servidor.java”*)
 - Compilar códigos do projeto Cliente
 - Executar o rmiregistry no servidor
 - Executar o código da classe Servidor
 - Gerar um arquivo JAR executável para o projeto cliente: *“File → export → runnable jar file”*
 - Copiar o arquivo jar em outro computador e executá-lo.

Importante

- Quando desejar que um método remoto retorne um objeto de uma classe que você criou, certifique-se de que esta classe implemente a interface *Serializable*. Caso contrário, sua aplicação não conseguirá serializar o objeto e retornará erro.

```
public class Message implements Serializable{
```



```
public ArrayList<Message> getMyPendingMessages(User user) throws RemoteException;
```

O que foi visto...

- RMI - Remote Method Invocation

Exercício 1

Utilizando RMI, crie um sistema que provenha um ambiente para chat entre usuários. Deverão ser criados dois projetos: (a) um para cuidar da parte do cliente e (b) outro para o servidor. O servidor será responsável por receber as mensagens dos clientes e armazená-las em um ArrayList para que clientes possam consultá-las sob demanda.

Exercício 1

(continuação)

O servidor deverá gerenciar uma lista de usuários que estão online. Portanto, deverá fornecer métodos para login e logoff de usuários. O servidor deverá também disponibilizar um método para envio de mensagens.

Exercício 1

(continuação)

O cliente deverá implementar uma Thread, que a cada 2,5 segundos fará uma requisição ao servidor a fim de verificar se há mensagens para o cliente.

A tela a seguir exhibe os métodos que ***deverão ser disponibilizados*** para acesso remoto.

Exercício 1

(continuação)

- Enviar mensagem

```
public boolean sendMessage(Message msg) throws RemoteException;
```

- Retornar lista de usuários online

```
public ArrayList<User> getOnlineUserList() throws RemoteException;
```

- Método para realizar login

```
public boolean login(User user) throws RemoteException;
```

- Método para realizar logoff

```
public boolean logoff(User user) throws RemoteException;
```

- Método para retornar mensagens enviadas para o usuário

```
public ArrayList<Message> getMyPendingMessages(User user) throws RemoteException;
```


Exercício 1

(continuação)

- *Recomenda-se* que as classes *Message* e *User* devam possuir os seguintes atributos (se desejado, pode-se mudar os atributos utilizados):
- Classe ***Message***
 - User fromUser;
 - int toProntuario;
 - String message;
- Classe ***User***
 - int prontuario;
 - String userName;