Remote Method Invocation (RMI)

Uma breve introdução

O QUE É RMI?

- É um mecanismo para permitir a invocação de métodos que residem em diferentes máquinas virtuais java (JVM).
- O JVM pode estar em diferentes máquinas ou podem estar na mesma máquina.
- O método pode ser executado em um endereço diferente do processo de chamada.
- É um mecanismo de chamada de procedimento remoto orientada a objetos.

Aplicação de objetos distribuídos (Sistemas distribuídos)

- Composto por dois ou mais programas, possivelmente um servidor e um ou vários clientes.
- O servidor cria objetos remotos e gera referencias aos mesmos.
- Os clientes invocam seus métodos.
- Este modelo de RMI implementa uma aplicação de objetos distribuídos.
- Esse modelo fornece mecanismos de comunicação entre o cliente e servidor.

Aplicação de objetos distribuídos (Continuação)

- A localização dos objetos remotos
  - Usando instalações de nomeação do RMI, através do registro RMI.
  - Passando e/ou retornando esses objetos remotos.
- Não é preciso lidar com a comunicação com os objetos remotos. (O sistema RMI trata isso para você)
- Não é preciso carregar os bytecodes das classes dos objetos passados como argumentos. (O sistema RMI trata isso para você)
- Todos os mecanismos para carregar um objeto e transmiti-lo é fornecido pelo RMI.

Interfaces e classes

- Todas as interfaces e classes do sistema RMI estão em java.rmi.
- O objeto remoto precisa obrigatoriamente implementar a interface Remote.
- As demais classes estendem RemoteObject.

A interface remota

- Estende a interface java.rmi.Remote
- Cada declaração de método deve propagar a exceção RemoteException

A classe RemoteObject

- Funções do servidor RMI são fornecidas pelo RemoteObject e suas sub-classes RemoteServer, Activatable e UnicastRemoteObject
- RemoteObject fornece implementações dos métodos toString, equals e hashCode na classes java.lang.Object
- As classes UnicastRemoteObject e Activatable cria objetos remotos e os exporta, ou seja, essas classes criam os objetos remotos utilizados pelos sistemas cliente

A classe RemoteException

- Super classe das exceções que o sistema RMI pode propagar
- Cada método remoto deve propagar RemoteException ou um das super classes para garantir robustez da aplicação
- A exceção será lançada a cada vez que tiver um erro
  - Falha de comunicação
  - Erro de protocolo
  - Erro de marshalling ou unmarshalling do(s) parâmetro(s) ou do objeto de retorno do método
- Aqueles que chamam esses métodos remotos devem tratar essas exceções. São exceções do tipo Checked

Implementação de um sistema simples usando RMI

- Vamos implementar o famoso AloMundo! :D \o/ o/
- Através dos seguintes passos
  - Definir a interface remota (estendendo a interface java.rmi.Remote)
  - Implementar o servidor
  - Implementar o cliente
  - Compilar os arquivos fontes
  - Colocar tudo para rodar! :D

Definir a interface remota (

```
public interface AloMundo extends Remote {

    String digaAloMundo() throws RemoteException;

}
```

Implementar o servidor

```
public class AloMundoServidor implementsAloMundo {
	public AloMundoServidor() {}

	public String digaAloMundo() {
		System.out.println("Chamada de aplicação Cliente recebida!");
		return "Agora não tem desculpa. Você já sabe RMI. Vambora programar!? Beleza!?";
	}

	public static void main(String args[]) {
		try {
			AloMundoServidor obj =new AloMundoServidor();
			AloMundo stub = (AloMundo) UnicastRemoteObject.exportObject(obj, 0);
			Registry registry =LocateRegistry.getRegistry();
			registry.bind("AloMundo", stub);
			System.out.println("Servidor pronto!");
		} catch (Exception e) {
			System.err.println("Capturando exceção no Servidor: " + e.toString());
			e.printStackTrace();
		}
	}
}
```

Implementar o Cliente

```
public class AloMundoCliente {
     private AloMundoCliente() {}

     public static void main(String[] args)  {
          String host = (args.length < 1) ? null : args[0];
          try  {
               Registry registry = LocateRegistry.getRegistry(host);
               AloMundo stub = (AloMundo) registry.lookup("AloMundo");
               String resposta = stub.digaAloMundo();
               System.out.println("resposta: " + resposta);
          } catch (Exception e)  {
               System.err.println("Capturando a exceção no Cliente: " + e.toString());
               e.printStackTrace();
          }
     }
}
```

Compilar os arquivos fonte

- No bat que eu criei, temos

```
"C:\Program Files\Java\jdk1.8.0_25\bin\javac" -d bin
exemplo/alomundo/AloMundo.java
exemplo/alomundo/AloMundoServidor.java
exemplo/alomundo/AloMundoCliente.java
```

- O parâmetro -d é apenas uma pasta destino
- Prestar a atenção no diretório HOME do java. Na sua máquina, favor, alterar!

Colocar tudo para rodar! (RMI Registry)

- No bat que eu criei, temos

```
"C:\Program Files\Java\jdk1.8.0_25\bin\rmiregistry.exe"
-J-Djava.rmi.server.useCodebaseOnly=false
```

- Sem o parâmetro -J-Djava.rmi.server.useCodebaseOnly=false o sistema só funciona se os .class estiverem no class-path da execução do rmiregistry.

Colocar tudo para rodar! (Cliente)

- No bat que eu criei, temos

```
"C:\Program Files\Java\jdk1.8.0_25\bin\java" -cp bin/
exemplo.alomundo.AloMundoCliente
```

- O parâmetro -cp bin/ determina o class-path da execução. Ou seja, as classes da aplicação Cliente

Colocar tudo para rodar! (Servidor)

- No bat que eu criei usei, temos

```
"C:\Program Files\Java\jdk1.8.0_25\bin\java" -cp bin/
-Djava.rmi.server.codebase=file:bin/
exemplo.alomundo.AloMundoServidor
```

- O parâmetro -Djava.rmi.server.codebase=file:bin/ serve para dizer onde encontrar os objetos remotos necessários

Compilar os arquivos fonte

- No SHELL SCRIPT que eu criei, temos

```
/usr/bin/javac -d bin exemplo/alomundo/AloMundo.java
exemplo/alomundo/AloMundoServidor.java
exemplo/alomundo/AloMundoCliente.java
```

- O parâmetro -d é apenas uma pasta destino
- Prestar a atenção no diretório HOME do java. Favor alterar na sua máquina!

Colocar tudo para rodar! (RMI Registry)

- No SHELL SCRIPT que eu criei, temos

```
/usr/bin/rmiregistry
-J-Djava.rmi.server.useCodebaseOnly=false
```

- Sem o parâmetro -J-Djava.rmi.server.useCodebaseOnly=false o sistema só funciona se os .class estiverem no class-path da execução do rmiregistry.

Colocar tudo para rodar! (Servidor)

- No SHELL SCRIPT que eu criei, temos

```
/usr/bin/java -cp bin/
-Djava.rmi.server.codebase=file:/home/rprado/rmi_introducao/
bin/ exemplo.alomundo.AloMundoServidor
```

- O parâmetro
  -Djava.rmi.server.codebase=file:/home/rprado/rmi_introducao/bin/ serve para dizer onde encontrar os objetos remotos necessários

Colocar tudo para rodar! (Cliente)

- No SHELL SCRIPT que eu criei, temos

```
/usr/bin/java -cp bin/ exemplo.alomundo.AloMundoCliente
```

- O parâmetro -cp bin/ determina o class-path da execução. Ou seja, as classes da aplicação Cliente

Referências

- http://www.devmedia.com.br/uma-introducao-ao-rmi-em-java/28681
- http://mrbool.com/rmi-remote-method-invocation-in-java/28575
- https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/hello/hello-world.html#start
- https://docs.oracle.com/javase/tutorial/rmi/overview.html

Conclusão

- Obrigado!!!

Para Contato!!!

GitHub

https://github.com/rodrigo-prado/rmi-introducao/

E-mail

rodrigo_prado@id.uff.br