Java Remote Method Invocation

Reference book:

[1] Kishori Sharan. *Java APIs, Extensions and Libraries: With JavaFX, JDBC, Jmod, Jlink, Networking, and the Process API*. Second Edition. USA; Apress. April 7, 2018. 813p. Chapter 6, Java Remote Method Invocation; p.489 – 513.

In this chapter, you will learn:

- What Java Remote Method Invocation (RMI) is and the RMI architecture
- How to develop and package RMI server and client applications
- How to start the rmiregistry, RMI server, and client applications

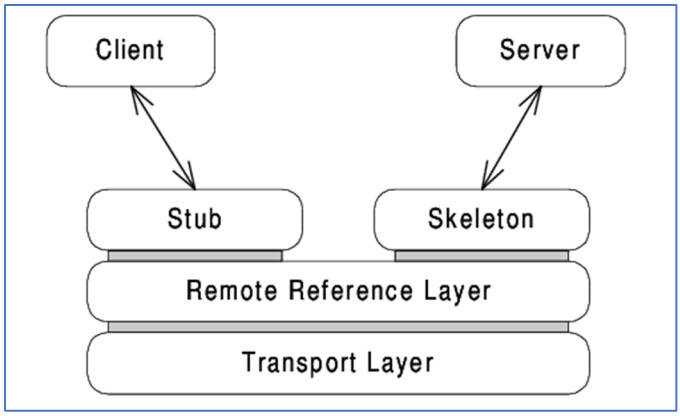
An RMI application contains classes and interfaces that fall into three parts:

- Server part
- Client part
- Common part, which is present in both the client and server

What Is Java Remote Method Invocation?

- Java RMI enables a Java application to invoke a method on a Java object in a remote JVM
- RMI lets you treat the remote object as if it is a local object. Internally, it uses sockets to handle access to the remote object and to invoke its methods.

RMI Architecture Java



- The RMI Architecture is very simple involving a client program, a server program, a stub and skeleton.
- In RMI, the client and server do not communicate directly; instead communicates through stub and skeleton. They are nothing but special programs generated by RMI compiler. They can be treated as proxies for the client and server. Stub program resides on client side and skeleton program resides on server.
- That is, the client sends a method call with appropriate parameters to stub. The stub in turn calls the skeleton on the server. The skeleton passes the stub request to the server to execute the remote method. The return value of the method is sent to the skeleton by the server. The

skeleton, as you expect, sends back to the stub. Finally the return value reaches client through stub.

How Java RMI Works?

- An RMI application consists of two programs, a client and a server, that run in two different JVMs.
- The server program creates Java objects and makes them accessible to the remote client programs to invoke methods on those objects.
- The client program needs to know the location of the remote objects on the server, so it can invoke methods on them.
- The server program creates a remote object and registers (or binds) its reference to an RMI registry.
- An RMI registry is a name service that is used to bind a remote object reference to a name, so a client can get the reference of the remote object using a name-based lookup in the registry.
- An RMI registry runs in a separate process from the server program. It is supplied as a tool called rmiregistry.

Developing an RMI Application?

- Writing a remote interface
- Implementing the remote interface in a class. An object of this class serves as the

remote object.

- Writing a server program. It creates an object of the class that implements the remote interface and registers it with the RMI registry.
- Writing a client program that accesses the remote object on the server.

Writing the Remote Interface

- It must extend the marker Remote interface.
- All methods in a remote interface must throw a RemoteException or an exception, which is its superclass such as IOException or Exception.

Implementing the Remote Interface

- There are two ways to write your implementation class for a remote interface.
- One way is to inherit it from the java.rmi.server.UnicastRemoteObject class.
- Another way is to inherit it from no class or any lass other than the UnicastRemoteObject class.
- One thing you need to note is that the constructors for the UnicastRemoteObject class throw a RemoteException. If you inherit the remote object implementation class from the UnicastRemoteObject class, the implementation class's constructor must throw a RemoteException in its declaration.
- You can have any number of other methods in this class.

Writing the RMI Server Program

- Installs the security manager.

- Creates and exports the remote object.
- Registers (or binds) the remote object with the RMI registry application

Installing the Security Manager

- You need to make sure that the server code is running under a security manager. When you run a Java program under a security manager, you must also control access to the privileged resources through a Java policy file.
- A security manager controls the access to privileged resources through a policy file. You will need to set appropriate permissions to access the resources used in a Java RMI application.

```
//Install the security manager

SecurityManager securityManager = System.getSecurityManager();
if(securityManager == null) {
        System.setProperty("java.security.policy", "mypolicy\\policy.policy");
        System.setSecurityManager(new SecurityManager());
}
```

```
//The sample Java policy file
grant {
        permission java.security.AllPermission;
};
```

Creating and Exporting the Remote Object

- You need to create remote object
- You need to export the remote object, so remote clients can invoke its remote methods. If your remote object class inherits from the UnicastRemoteObject class, you do not need to export it. It is exported automatically when you create it.
- If your remote object's class does not inherit from the UnicastRemoteObject class, you need to export it explicitly using one of the exportObject() static methods of the UnicastRemoteObject class. The exportObject() method returns the reference of the exported remote object, which is also called a stub or a remote reference.

Registering (binding) the Remote Object

- LocateRegistry.createRegistry(17676); //port : 17676
- Naming.rebind("rmi://host:port/nameRemoteObject", anRemoteObject);

Writing the RMI Client Program

- It makes sure that it is running under a security manager.
- The RMI client program calls the methods on remote objects, which exist on the remote server. The first thing that a client program must do is to know the location of the remote object → It performs the lookup in the registry using the lookup() method of the Registry interface.

```
XObject obj = (XObject) Naming.lookup("rmi://host:port/nameRemoteObject");
```

Separating the Server and Client Code

It is important that you separate the code for the server and client programs in an RMI application.

- The server program needs to have the following three components:
  - The remote interface
  - The implementation class for the remote interface
  - The server program
- The client program needs to have the following two components.
  - The remote interface
  - The client program

QUESTIONS AND EXERCISES

1. What is Java Remote Method Invocation?
2. What is the fully qualified name of the interface that every remote interface must extend?
3. What steps do you need to perform in your RMI server program after you create a remote object, so the remote object is available for a client to use?
4. What is RMI registry and where is it located?
5. In an RMI application, can an RMI registry and RMI server be deployed to two different machines? If your answer is no, explain why.
6. Describe the typical sequence of steps an RMI client program needs to perform to call a method on a remote object.
7. An RMI application involves three layers of applications: client, RMI registry, and server. In what order must these applications be run?

Remote Method Invocation (advance)

https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/index.html

https://www.javacamp.org/moreclasses/rmi/rmi23.html

- The server seems running forever. If no clients use the remote objects, the server wastes system resources.
- If the RMI server keeps running without any connection, it seems wasting resources. It would be better if a remote object is delivered, shuts down itself when necessary and is activated on demand, rather than running all the time. Actually, Java RMI activation daemon, rmid is designed to do such job. The activation daemon will listen and handle the creation of activatable object on demand.
- Based on the RMI specification, you should create an activatable object first, then make a setup program to deliver the activatable object.

What is an activatable remote object?

An activatable remote object is a remote object that starts executing when its remote methods are invoked and shuts itself down when necessary.

How to create an activatable object?

If a class *extends java.rmi.activation.Activatable class* and has *a constructor to accept ActivationID and MarshalledObject parameters*, that class is an activatable object.

Implementing the Remote Interface (changed)

- Let it extend Activatable class
- Declared a two-argument constructor passing ActivationID to register the object with the activation system and a MarshalledObject. This constructor is required and should throw RemoteException. The super(id, 0) method calls Activatable constructor to pass an activation ID and a port number. In this case, the port number is default 1099.

The following classes are involved with the activation process:

1. ActivationGroup class -- responsible for creating new instances of activatable objects in its group.
2. ActivationGroupDesc class -- contains the information necessary to create or re-create an activation group in which to activate objects in the same JVM.
3. ActivationGroupDesc.CommandEnvironment class -- allows overriding default system properties and specifying implementation-defined options for an ActivationGroup.
4. ActivationGroupID class -- identifies the group uniquely within the activation system and contains a reference to the group's activation system.
5. getSystem()-- returns an ActivationSystem interface implementation class.
6. MarshalledObject class -- a container for an object that allows that object to be passed as a paramter in an RMI call.

How to create a set-up program

The purpose of the set-up program is to register information about activable object with rmid and the rmiregistry. It takes the following steps:

1. Install security manager for the ActivationGroup VM.
2. Set security policy
3. Create an instance of ActivationGroupDesc class
4. Register the instance and get an ActivationGroupID.
5. Create an instance of ActivationDesc.
6. Register the instance with rmid.
7. Bind or rebind the remote object instance with its name
8. Exit the system.

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

```
    public static void main(String[] args) throws RemoteException, ActivationException,
MalformedURLException {
//              Install security manager for the ActivationGroup VM
                SecurityManager securityManager = System.getSecurityManager();
                if(securityManager == null)
                        System.setSecurityManager(new SecurityManager());

//              Set security policy
                Properties props = new Properties();
                props.setProperty("java.security.policy", "javapolicy.policy");

//              Create an instance of ActivationGroupDesc class
                ActivationGroupDesc.CommandEnvironment aec = null;
                ActivationGroupDesc groupDesc = new ActivationGroupDesc(props, aec);

//              Register the instance and get an ActivationGroupID.
                ActivationGroupID groupID =
ActivationGroup.getSystem().registerGroup(groupDesc);

//              Create an instance of ActivationDesc.
                ActivationDesc desc = new ActivationDesc(groupID,
CalActivatableImpl.class.getName(), null, null);

//              Register the instance with rmid.
                Remote obj = Activatable.register(desc);

//              Bind or rebind the remote object instance with its name
                Naming.rebind("rmi://localhost:1099/myobj", obj);
```

```
            System.out.println("Exported!");

//          Exit the system
            System.exit(0);
        }
```

How to run program

1. Compile java classes

   Using ***javac*** tool

   javac -d . *.java

2. Start the rmiregistry

   Using command: ***start rmiregistry***

3. Start the activation daemon, rmid

   Using command: ***start rmid -J-Djava.security.policy=rmi.policy***

```
grant{
        permission com.sun.rmi.rmid.ExecPermission "${java.home}${/}bin${/}java";
        permission com.sun.rmi.rmid.ExecOptionPermission "-Djava.security.policy=*";
};
```

4. Run the server program

   Using tool: ***java -Djava.security.policy=javapolicy.policy*** **YourServer**

5. Run the client program

   Using tool: ***java -Djava.security.policy=javapolicy.policy YourClient***