

More RMI (Material taken from Java Tutorial)

Distributed object applications need to do the following:

- **Locate remote objects:** Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of other remote invocations.
- **Communicate with remote objects:** Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
- **Load class definitions for objects that are passed around:** Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

Advantages of Dynamic Code Loading

- One of the central and unique features of RMI is its ability to download the definition of an object's class if the class is not defined in the receiver's Java virtual machine.
- All of the types and behavior of an object, previously available only in a single Java virtual machine, can be transmitted to another, possibly remote, Java virtual machine.
- RMI passes objects by their actual classes, so the behavior of the objects is not changed when they are sent to another Java virtual machine.
- This capability enables new types and behaviors to be introduced into a remote Java virtual machine, thus dynamically extending the behavior of an application.

Designing and Implementing the Application Components

- **Defining the remote interfaces :** A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods. If any of these interfaces or classes do not yet exist, you need to define them as well.
- **Implementing the remote objects :** Remote objects must implement one or more remote interfaces.
- **Implementing the clients :** Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

Building a Generic Compute Engine

- The compute engine is a remote object on the server that takes tasks from clients, runs the tasks, and returns any results.
- The tasks are run on the machine where the server is running.
- This type of distributed application can enable a number of client machines to make use of a particularly powerful machine or a machine that has specialized hardware.
- The novel aspect of the compute engine is that the tasks it runs do not need to be defined when the compute engine is written or started.
- New kinds of tasks can be created at any time and then given to the compute engine to be run.
- The only requirement of a task is that its class implement a particular interface.
- The code needed to accomplish the task can be downloaded by the RMI system to the compute engine. Then, the compute engine runs the task, using the resources on the machine on which the compute engine is running.

Building a Generic Compute Engine

- The ability to perform arbitrary tasks is enabled by the dynamic nature of the Java platform, which is extended to the network by RMI.
- RMI dynamically loads the task code into the compute engine's Java virtual machine and runs the task without prior knowledge of the class that implements the task.
- Such an application, which has the ability to download code dynamically, is often called a *behavior-based application*.
- Such applications usually require full agent-enabled infrastructures. With RMI, such applications are part of the basic mechanisms for distributed computing on the Java platform.

compute.Compute Interface

The `compute.Compute` interface defines the remotely accessible part, the compute engine itself. Here is the source code for the `Compute` interface:

```
package compute;

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

public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```

- By extending the interface `java.rmi.Remote`, the `Compute` interface identifies itself as an interface whose methods can be invoked from another Java virtual machine. Any object that implements this interface can be a remote object.
- As a member of a remote interface, the `executeTask` method is a remote method. Therefore, this method must be defined as being capable of throwing a `java.rmi.RemoteException`.
- The `Compute` interface's `executeTask` method returns the result of the execution of the `Task` instance passed

to it. Thus, the `executeTask` method has its own type parameter, `T`, that associates its own return type with the result type of the passed `Task` instance.

compute.Task Interface

- The second interface needed for the compute engine is the **Task** interface, which is the type of the parameter to the **executeTask** method in the **Compute** interface.
- The **compute.Task** interface defines the interface between the compute engine and the work that it needs to do, providing the way to start the work.

```
package compute;
```

```
public interface Task<T> {  
    T execute();  
}
```

- The **Task** interface defines a single method, **execute**, which has no parameters and throws no exceptions. Because the interface does not extend **Remote**, the method in this interface doesn't need to list **java.rmi.RemoteException** in its throws clause.
- The **Task** interface has a type parameter, **T**, which represents the result type of the task's computation. This interface's **execute** method returns the result of the computation and thus its return type is **T**.

Passing objects between different virtual machines

- RMI uses the Java object serialization mechanism to transport objects by value between Java virtual machines.
- For an object to be considered serializable, its class must implement the `java.io.Serializable` marker interface. Therefore, classes that implement the `Task` interface must also implement `Serializable`, as must the classes of objects used for task results.
- Different kinds of tasks can be run by a `Compute` object as long as they are implementations of the `Task` type. The classes that implement this interface can contain any data needed for the computation of the task and any other methods needed for the computation.

How RMI executes tasks

- RMI can assume that the **Task** objects are written in the Java programming language, implementations of the **Task** object that were previously unknown to the compute engine are downloaded by RMI into the compute engine's Java virtual machine as needed.
- This capability enables clients of the compute engine to define new kinds of tasks to be run on the server machine without needing the code to be explicitly installed on that machine.
- The compute engine, implemented by the **ComputeEngine** class, implements the **Compute** interface, enabling different tasks to be submitted to it by calls to its **executeTask** method. These tasks are run using the task's implementation of the **execute** method and the results, are returned to the remote client.

Server code

```
package engine;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }
}
```

Server code

```
public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
        String name = "Compute";
        Compute engine = new ComputeEngine();
        Compute stub =
            (Compute) UnicastRemoteObject.exportObject(engine, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind(name, stub);
        System.out.println("ComputeEngine bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine exception:");
        e.printStackTrace();
    }
}
```

- The main method's first task is to create and install a security manager, which protects access to system resources from untrusted downloaded code running within the Java virtual machine. A security manager determines whether downloaded code has access to the local file system or can perform any other privileged operations.

Server code

```
Compute engine = new ComputeEngine();  
Compute stub =  
    (Compute) UnicastRemoteObject.exportObject(engine, 0);
```

- The static `UnicastRemoteObject.exportObject` method exports the supplied remote object so that it can receive invocations of its remote methods from remote clients. The static `UnicastRemoteObject.exportObject` method exports the supplied remote object so that it can receive invocations of its remote methods from remote clients.
- The second argument, an `int`, specifies which TCP port to use to listen for incoming remote invocation requests for the object. It is common to use the value zero, which specifies the use of an anonymous port. The actual port will then be chosen at runtime by RMI or the underlying operating system.
- The `exportObject` method returns a stub for the exported remote object. Note that the type of the variable `stub` must be `Compute`, not `ComputeEngine`, because the stub for a remote object only implements the remote interfaces that the exported remote object implements.

Client code

- A client needs to call the compute engine, but it also has to define the task to be performed by the compute engine.
- Two separate classes make up the client in our example. The first class, **ComputePi**, looks up and invokes a **Compute** object.
- The second class, **Pi**, implements the **Task** interface and defines the work to be done by the compute engine. The job of the **Pi** class is to compute the value of π to some number of decimal places.

```
package compute;

public interface Task<T> {
    T execute();
}
```

This is the non-remote **Task** interface.

Client code : client.ComputePi

```
package client;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}
```

Client code : `client.ComputePi`

```
String name = "Compute";
```

```
Registry registry = LocateRegistry.getRegistry(args[0]);
```

- After installing a security manager, the client constructs a name to use to look up a `Compute` remote object, using the same name used by `ComputeEngine` to bind its remote object.
- Also, the client uses the `LocateRegistry.getRegistry` API to synthesize a remote reference to the registry on the server's host.
- The value of the first command-line argument, `args[0]`, is the name of the remote host on which the `Compute` object runs.
- The client then invokes the lookup method on the registry to look up the remote object by name in the server host's registry.

Client code : `client.ComputePi`

```
Pi task = new Pi(Integer.parseInt(args[1]));  
    BigDecimal pi = comp.executeTask(task);  
    System.out.println(pi);
```

- Next, the client creates a new `Pi` object, passing to the `Pi` constructor the value of the second command-line argument, `args[1]`, parsed as an integer. This argument indicates the number of decimal places to use in the calculation.
- Finally, the client invokes the `executeTask` method of the `Compute` remote object. The object passed into the `executeTask` invocation returns an object of type `BigDecimal`, which the program stores in the variable `result`.

Client code : client.Pi

```
package client;

import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;

public class Pi implements Task<BigDecimal>, Serializable {
    ...

    public BigDecimal execute() {
        return computePi(digits);
    }
}
```

- You can lookup the rest of the code from the Java Tutorial.