RMI

Tutorial

Outline

- Remote Method Invocation
- Program

RMI Overview

- RMI is a Java-implementation of RPC referred to as a distributed object application
- An RMI server typically creates some remote objects, makes references to those objects accessible, and waits for clients to invoke methods on those objects
- An RMI client obtains a remote reference to one or more remote objects on a server and invokes methods on them
  - RMI clients can locate remote objects through an RMI registry, assuming the RMI server has registered its remote objects with it
- The details of remote communication between server and client are handled by RMI
  - Remote communication looks like regular Java method invocations to the programmer
- The client can pass a class to a remote server and have it execute methods on that class

RMI Dynamic Code Loading

- RMI has the ability to download the definition of an object's class if the class is not defined in the client's JVM
  - All of the types and behaviors of an object can be transmitted to a remote JVM
  - New types and behaviors can be introduced into a remote JVM, thus dynamically extending the behavior of an application

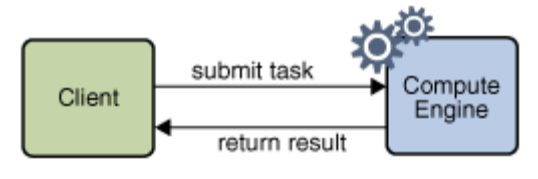
RMI Application Example

- Here are the steps for setting up an RMI application
  - This example is adapted from http://docs.oracle.com/javase/tutorial/rmi/overview.html

1. Write the code
    1.1 Write the remote interface
    1.2 Write the server code
    1.3 Write the client code
2. Compile the code
3. Make the classes network accessible
4. Start the RMI server and the application

RMI Application Example – Step 1.1

- Step 1.1 – Write the remote interface
  - The example we will see here allows a client to submit a task to a server program, the server program to run that task, and the results of that task returned to the client



  - `Compute` is the remote interface that allows tasks to be submitted to the engine
    - Since `Compute` inherits from java.rmi.Remote, its method `executeTask(Task<T>)` can be invoked from another JVM
  - `Task` is the client interface that defines how the compute engine executes a submitted task
    - The `Task` interface is the parameter to the `executeTask` method in the `Compute` interface
  - Objects are passed from client to server serialized, so the class implementing the `Task` interface and the parameterized return type `T` must both be `Serializable`

RMI Application Example – Step 1.1

Compute.java

```
1   package compute;
2
3   import java.rmi.Remote;
4   import java.rmi.RemoteException;
5
6   public interface Compute extends Remote {
7     <T> T executeTask(Task<T> t) throws RemoteException;
8   }
```

Task.java

```
1   package compute;
2
3   public interface Task<T> {
4     T execute();
5   }
```

RMI Application Example – Step 1.1 Explanation

- Since RMI can assume the `Task` objects are written in Java, implementations of the `Task` object that were previously unknown to the server are downloaded by RMI into the server's JVM
  - This means that clients are able to define new types of tasks to be run on the server without that code needing to be explicitly installed on the server
- The server code in the `ComputeEngine` class implements the `Compute` interface and enables different tasks to be submitted to it by calls to the `executeTask` method
  - This method just executes the task's `execute` method and returns the results to the remote client

RMI Application Example – Step 1.2

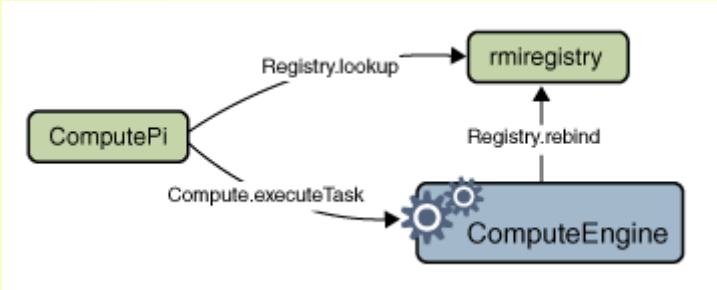
- Step 1.2 – Write the server code
  - A class that implements a remote interface needs to provide an implementation for each remote method in the interface
  - The server program needs to create the remote objects and export them to the RMI runtime, making them available to receive incoming remote invocations
  - A security manager must be created and installed so the RMI runtime knows what can be executed on the server
  - Remote objects are passed by reference from a client
  - Other parameters that are not remote objects are passed by value

RMI Application Example – Step 1.2

```
1   package engine;
2   import java.rmi.registry.LocateRegistry;
3   import java.rmi.registry.Registry;
4   import java.rmi.server.UnicastRemoteObject;
5   import compute.Compute;
6   import compute.Task;
7
8   public class ComputeEngine implements Compute {
9     public ComputeEngine() {
10      super();
11    }
12    public <T> T executeTask(Task<T> t) {
13      return t.execute();
14    }
15    public static void main(String[] args) {
16      if (System.getSecurityManager() == null) {
17        System.setSecurityManager(new SecurityManager());
18      }
19      try {
20        String name = "Compute";
21        Compute engine = new ComputeEngine();
22        Compute stub = (Compute) UnicastRemoteObject.exportObject(engine, 0);
23        Registry registry = LocateRegistry.getRegistry();
24        registry.rebind(name, stub);
25        System.out.println("ComputeEngine bound");
26      } catch (Exception e) {
27        System.err.println("ComputeEngine exception:");
28        e.printStackTrace();
29      }
30    }
31  }
```

RMI Application Example – Step 1.3

- Step 1.3 – Write the client code
  - The client for this program needs to define the task that it wants the server to perform (`Pi`)
    - This means the client needs to create a class that implements the `Task<T>` interface
  - The client has another program (`ComputePi`) that will obtain a reference to the newly-created `Task<T>` object and request it to be executed on the server
    - This means that it needs to contact the RMI registry and submit the `Task` to be executed by calling the `executeTask(Task<T>)` method on a `Compute` object



RMI Application Example – Step 1.3

```
1   package client;
2
3   import compute.Task;
4   import java.io.Serializable;
5   import java.math.BigDecimal;
6
7   public class Pi implements Task<BigDecimal>, Serializable {
8
9     private static final long serialVersionUID = 227L;
10
11    /** constants used in pi computation */
12    private static final BigDecimal FOUR =
13      BigDecimal.valueOf(4);
14
15    /** rounding mode to use during pi computation */
16    private static final int roundingMode =
17        BigDecimal.ROUND_HALF_EVEN;
18
19    /** digits of precision after the decimal point */
20    private final int digits;
21
22    /**
23     * Construct a task to calculate pi to the specified
24     * precision.
25     */
26    public Pi(int digits) {
27      this.digits = digits;
28    }
29
30    /**
31     * Calculate pi.
32     */
33    public BigDecimal execute() {
34      return computePi(digits);
35    }
36    /**
37     * Compute the value of pi to the specified number of
38     * digits after the decimal point using Machin's formula.
39     *          pi/4 = 4*arctan(1/5) - arctan(1/239)
40     */
41    public static BigDecimal computePi(int digits) {
42      int scale = digits + 5;
43      BigDecimal arctan1_5 = arctan(5, scale);
44      BigDecimal arctan1_239 = arctan(239, scale);
45      BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
46                                  arctan1_239).multiply(FOUR);
47      return pi.setScale(digits, BigDecimal.ROUND_HALF_UP);
48    }
49    /**
50     * Compute the value, in radians, of the arctangent of
51     * the inverse of the supplied integer to the specified
52     * number of digits after the decimal point.  The value
53     * is computed using the power series expansion
54     * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 + ...
55     */
56    public static BigDecimal arctan(int inverseX, int scale) {
57      BigDecimal result, numer, term;
58      BigDecimal invX = BigDecimal.valueOf(inverseX);
59      BigDecimal invX2 = BigDecimal.valueOf(inverseX * inverseX);
60      numer = BigDecimal.ONE.divide(invX, scale, roundingMode);
61      result = numer;
62      int i = 1;
63      do {
64        numer = numer.divide(invX2, scale, roundingMode);
65        int denom = 2 * i + 1;
66        term = numer.divide(BigDecimal.valueOf(denom),
67                            scale, roundingMode);
68        if ((i % 2) != 0) {
69          result = result.subtract(term);
70        } else {
71          result = result.add(term);
72        }
73        i++;
74      } while (term.compareTo(BigDecimal.ZERO) != 0);
75      return result;
76    }
77  }
```

RMI Application Example – Step 1.3

```
1   package client;
2
3   import java.rmi.registry.LocateRegistry;
4   import java.rmi.registry.Registry;
5   import java.math.BigDecimal;
6   import compute.Compute;
7
8   public class ComputePi {
9     public static void main(String args[]) {
10      if (System.getSecurityManager() == null) {
11        System.setSecurityManager(new SecurityManager());
12      }
13      try {
14        String name = "Compute";
15        Registry registry = LocateRegistry.getRegistry(args[0]);
16        Compute comp = (Compute) registry.lookup(name);
17        Pi task = new Pi(Integer.parseInt(args[1]));
18        BigDecimal pi = comp.executeTask(task); // makes remote procedure call
19        System.out.println(pi);
20      } catch (Exception e) {
21        System.err.println("ComputePi exception:");
22        e.printStackTrace();
23      }
24    }
25  }
```

RMI Application Example – Step 2

- Step 2 – Compile the code
  - Place Compute.java and Task.java in the compute directory
  - Place ComputeEngine.java in the engine directory
  - Place ComputePi.java and Pi.java in the client directory
  - Build a jar file of the interfaces that will be used on the client and the server

    Windows -> `javac compute\Compute.java compute\Task.java`

    Windows -> `jar cvf compute.jar compute\*.class`

    Mac -> `javac compute/Compute.java compute/Task.java`

    Mac -> `jar cvf compute.jar compute/*.class`
  - Compile the server

    Windows -> `javac -classpath .;compute.jar engine\ComputeEngine.java`

    Mac -> `javac –classpath .:compute.jar engine/ComputeEngine.java`
  - Compile the client

    Windows -> `javac –classpath .;compute.jar client\ComputePi.java client\Pi.java`

    Mac -> `javac –classpath .:compute.jar client/ComputePi.java client/Pi.java`

RMI Application Example – Step 3

- Step 3 – Make the classes network accessible
  - The compute.jar file needs to be accessible to clients when compiling
    - This is usually accomplished by making it available through the file system or a web server
  - Pi.class needs to be available to be downloaded by the server upon remote invocation (if it doesn't have that file in its classpath)
    - This is usually accomplished by making it available through the file system or a web server
  - The locations of these files will be specified in the java.rmi.server.codebase environment variable that is set when the server and client are executed in Step 4

RMI Application Example – Step 4

- You need to create a server.policy file to specify the security a client will have running a program on the server
- You also need to create a client.policy file to specify the security a server will have in an object it returns to the client

server.policy

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

client.policy

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

RMI Application Example – Step 4

- Step 4 – Start the RMI server and the application
  - Start the RMI registry from the same directory that contains the server code

```
rmiregistry
```

  - Run the server (all on one line)

```
java -classpath compute.jar;.
     -Djava.security.policy=server.policy engine.ComputeEngine
```

  - Run the client (all on one line)
    - The two parameters passed along the command line are the server's hostname (or IP address) and the number of digits of Pi to retrieve

```
java -classpath .
     -Djava.security.policy=client.policy client.ComputePi localhost 55
```