

Практическая работа №2

Концепция удаленного вызова метода (RMI) в JAVA

Цель работы: Знакомство с механизмами удаленного вызова процедур в JAVA.

Теоретические сведения:

RMI - это объектно-ориентированный эквивалент RPC (удаленного вызова процедур). Это называется вызовом удаленного метода. Вызов удаленного метода (RMI) позволяет объекту Java вызывать метод объекта, запущенного на другом компьютере. RMI обеспечивает удаленную связь между Java-программой.

Главной целью разработчиков RMI было предоставление возможности программистам разрабатывать распределенные Java программы, используя такие же синтаксис и семантику, как и при разработке обычных нераспределенных программ. Для этого они должны были преобразовать модель работы классов и объектов в одной виртуальной машине Java (JVM) в новую модель работы классов и объектов в распределенной (несколько JVM) вычислительной среде.

Приложение RMI можно разделить на две части. Одним из них является программой клиентом и другая программа сервер. Сервер программа создает некоторые удаленный объект, сделать их ссылки доступны для клиента для вызова метода на нем. Клиент программа делает запрос для удаленных объектов на сервере и вызвать метод на них. Заглушка и скелет - два важных объекта, используемых для связи с удаленным объектом.

Таким образом, запустив открытый сервер RMI в системе, можно разрешить внешним субъектам взаимодействовать с ним и, возможно, выполнять методы на сервере RMI. Эти методы должны быть определены в реализации Сервера. Как только они вызываются клиентом, они будут выполняться на сервере, а возвращаемые значения будут возвращены клиенту. Еще одна интересная часть заключается в том, что собственный RMI (опять же, я НЕ говорю о JMXRMI) не поддерживает большую часть безопасности, кроме шифрования соединения с использованием SSL.

Интерфейсы: основа RMI

Архитектура RMI основана на одном важном принципе: определение поведения и реализация этого поведения считаются разными понятиями. RMI дает возможность разделить и выполнить на разных JVM код, определяющий поведение, и код, реализующий поведение.

Это прекрасно соответствует требованиям распределенных систем, в которых клиенты знают об определениях служб, а серверы предоставляют эти службы.

Конкретно в RMI определение удаленной службы кодируется при помощи интерфейса Java. Реализация удаленной службы кодируется в классе. Таким образом, ключ к пониманию RMI – помнить, что интерфейсы определяют поведение, а классы определяют реализацию.

На следующей диаграмме показана архитектура приложения RMI.

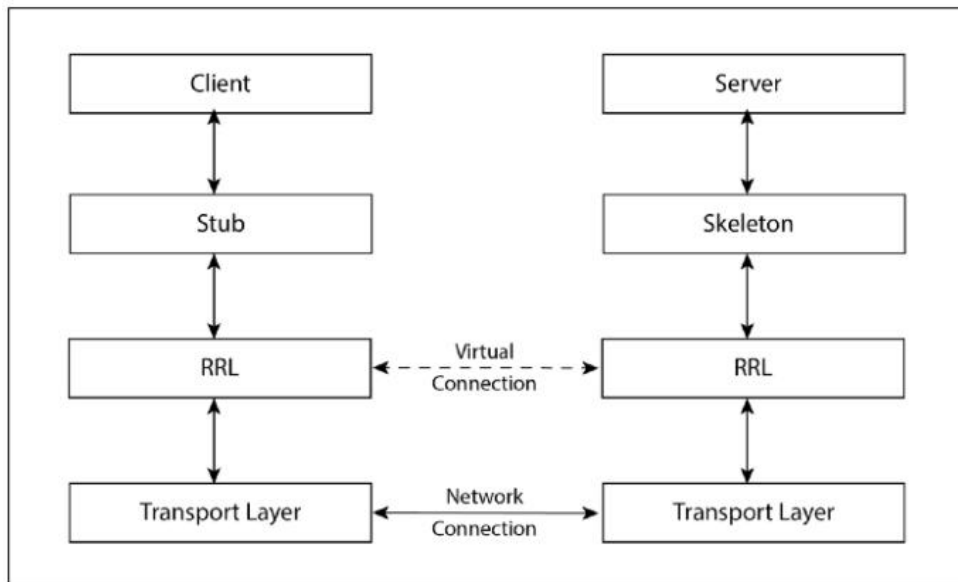


Рисунок 2.1 - Архитектура интерфейса RMI

Работа приложения RMI

Обобщённо, работу распределённого приложения RMI можно описать следующими шагами:

1. Когда клиент делает вызов удаленному объекту, он принимается заглушкой, которая в конечном итоге передает этот запрос в RRL.
2. Когда клиентский RRL получает запрос, он вызывает метод `invoke()` объекта `remoteRef`. Он передает запрос в RRL на стороне сервера.
3. RRL на стороне сервера передает запрос скелету (прокси на сервере), который, наконец, вызывает требуемый объект на сервере.
4. Результат полностью передается клиенту.

Уровни архитектуры RMI

Рассмотрев высокоуровневую архитектуру RMI, взглянем на ее реализацию. Реализация RMI, по существу, состоит из трех абстрактных уровней. Первый – это уровень заглушки и скелета, расположенный

непосредственно перед разработчиком. Этот уровень перехватывает вызовы методов, произведенные клиентом при помощи переменной-ссылки на интерфейс, и переадресует их в удаленную службу RMI.

Следующий уровень – уровень удаленной ссылки. Этот уровень понимает, как интерпретировать и управлять ссылками на удаленные объекты служб. В JDK 1.1 этот уровень соединяет клиентов с удаленными объектами служб, которые исполняются на сервере. Это соединение является связью типа один к одному (однонаправленное соединение). В Java 2 SDK этот уровень был расширен поддержкой активации пассивных удаленных объектов при помощи технологии Remote Object Activation.

Транспортный уровень основан на соединениях TCP/IP между сетевыми машинами. Он обеспечивает основные возможности соединения и некоторые стратегии защиты от несанкционированного доступа.

При использовании уровневой архитектуры каждый из уровней может быть изменен или заменен без воздействия на остальную систему. Например, транспортный уровень может быть заменен протоколом UDP/IP без изменения остальных уровней.

Заглушка и скелет

Имена «заглушка» и «скелет» могут сбивать с толку на первый взгляд, но это просто то, как называются «клиентская» и «серверная» части удаленного объекта.

Stub - это класс, реализующий удаленный интерфейс и служащий в качестве заполнителя на стороне клиента для удаленного объекта. С другой стороны, Skeleton - это серверная сущность, которая отправляет вызовы фактической реализации удаленного объекта.

Заглушка и скелетон - два важных объекта, используемых для связи с удаленным объектом.

Уровень заглушки и скелета RMI расположен непосредственно перед разработчиком Java. На этом уровне RMI использует прокси-модель проектирования, которая описана в книге Gamma, Helm, Johnson и Vlissides «Design Patterns». В прокси-модели объект одного контекста представляется другим (прокси-объектом) в отдельном контексте. Прокси-объект знает, как направлять вызовы методов между этими объектами.

В прокси-модели, используемой в RMI, роль прокси выполняет класс заглушки, а роль RealSub j ect выполняет класс, реализующий удаленную службу.

Скелет является вспомогательным классом, который создается для использования RMI. Скелет понимает, как взаимодействовать с заглушкой при RMI соединении. Скелет поддерживает общение с заглушкой; он читает параметры для вызова метода из соединения, производит вызов объекта, реализующего удаленную службу, принимает возвращаемое значение и записывает его обратно в заглушку.

В реализации RMI Java 2 SDK новый протокол связи сделал классы скелетов не нужными. RMI использует отражение для установления соединения с объектом удаленной службы. Вы должны использовать классы и объекты скелетов только в JDK 1.1 и совместимых с ним реализациях систем.

Заглушка

Заглушка, которая представляет собой объект, используемый в RMI в качестве шлюза для клиентской стороны. Все исходящие запросы проходят через него. Когда клиент вызывает метод,

1. Соединение устанавливается с помощью удаленной виртуальной машины.
2. Это параметры передачи для удаленной виртуальной машины. Это называют маршалами.
3. Затем дожидается вывода
4. Считывает возвращаемое значение или исключение. Это называют немаршалами.
5. Заключительный шаг - выполняется возврат значений вызывающей стороне.

Скелет

Скелет, который представляет собой объект, используемый в RMI в качестве шлюза для серверной части. Все входящие запросы проходят через него. Когда сервер вызывает метод,

1. Считывает параметр для удаленного метода

2. Вызывает метод реального удаленного объекта.
3. Записывает и передает (маршалирует) результат вызывающей стороне.

Маршаллинг и демаршаллинг

Каждый раз, когда клиент вызывает метод, который принимает параметры удаленного объекта, параметры объединяются в сообщение перед отправкой по сети. Эти параметры могут быть примитивного типа или объектами. В случае примитивного типа параметры объединяются и к ним прикрепляется заголовок. Если параметры являются объектами, они сериализуются. Этот процесс известен как **маршаллинг**.

На стороне сервера упакованные параметры разделяются, а затем вызывается требуемый метод. Этот процесс называется **демаршалингом**.

Уровень удаленных ссылок (RRL)

Уровни удаленных ссылок определяют и поддерживают семантику вызовов соединения RMI. Этот уровень предоставляет объект `RemoteRef`, который обеспечивает соединение с объектами, реализующими удаленные службы.

Объекты заглушки используют метод `invoke()` в объекте `RemoteRef` для направления вызова метода. Объект `RemoteRef` понимает семантику вызова удаленных служб.

Реализация RMI в JDK 1.1 обеспечивает только один способ соединения клиентов с реализациями удаленных служб: однонаправленное соединение типа точка-точка. Перед тем, как клиент сможет использовать удаленную службу, экземпляр объекта, реализующего ее, должен быть создан на сервере и экспортирован в систему RMI. (Если это основная служба, она также должна быть поименована и зарегистрирована в реестре RMI).

Реализация RMI в Java 2 SDK добавляет новую семантику для соединения клиент-сервер. В этой версии RMI поддерживает способные к активизации удаленные объекты. Когда производится вызов метода прокси для такого объекта, RMI определяет, находится ли объект, реализующий удаленную службу, в пассивном состоянии. Если да, то RMI создаст экземпляр объекта и восстановит его состояние из дискового файла. Как только объект активизируется в памяти, он начинает вести себя так же, как и объект, реализующий удаленную службу JDK 1.1.

Доступны и другие типы семантики соединений. Например, в случае широковебчательного соединения, один прокси-объект может передать

запрос метода нескольким реализациям одновременно и принять первый ответ (это уменьшает время отклика и, возможно, повышает доступность объекта). В будущем Sun возможно добавит дополнительные типы семантики в RMI.

Транспортный уровень

Транспортный уровень осуществляет соединение между различными JVM. Все соединения представляют собой основанные на потоках сетевые соединения, использующие TCP/IP.

Даже если две JVM работают на одном и том же физическом компьютере, они соединяются через стек сетевых протоколов TCP/IP. (Вот почему вы должны иметь действующую конфигурацию TCP/IP на вашем Компьютере).

На следующей диаграмме показаны TCP/IP соединения между разными JVM. Транспортный уровень RMI был разработан для осуществления соединения между клиентами и сервером даже с учетом сетевых помех.

Хотя транспортный уровень предпочитает использовать несколько TCP/IP соединений, некоторые сетевые конфигурации разрешают только одно TCP/IP соединение между клиентом и сервером (некоторые браузеры ограничивают апплеты одним сетевым соединением с их сервером).

В этом случае, транспортный уровень распределяет несколько виртуальных соединений внутри одного TCP/IP соединения.

Реестр RMI

Реестр RMI - это пространство имен, в котором размещаются все объекты сервера. Каждый раз, когда сервер создает объект, он регистрирует этот объект в RMIregistry (используя методы **bind ()** или **reBind ()**). Они регистрируются с использованием уникального имени, известного как **имя привязки**.

Чтобы вызвать удаленный объект, клиенту нужна ссылка на этот объект. В это время клиент выбирает объект из реестра, используя его имя привязки (с помощью метода **lookup ()**).

На следующем рисунке показан данный процесс.

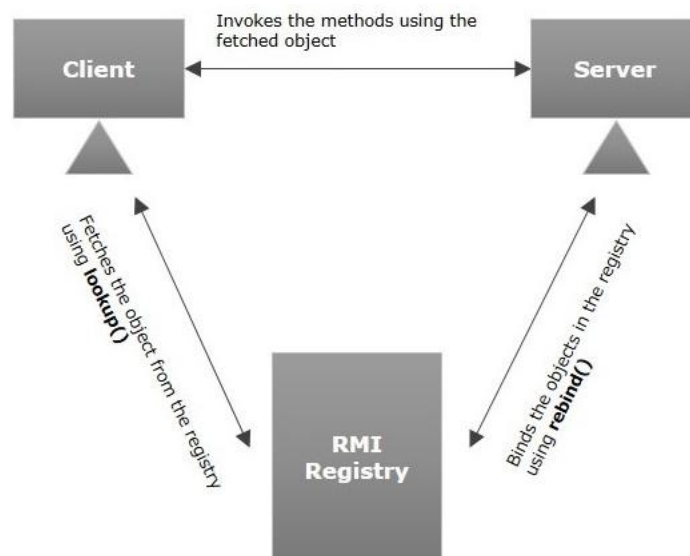


Рисунок 2.2 - Процесс размещения объектов сервера в реестре RMI

Основные шаги работы с RMI:

1. Определить удаленный интерфейс, согласованный с сервером.
2. Написать код сервера.
3. Запустить программу `rmic` (Java RMI stub compiler – компилятор заглушек RMI) для генерации связующего кода.
4. Написать код клиента.
5. Убедиться, что на сервере запущен RMI реестр (программа `rmiregistry`).
6. Запустить сервер.
7. Запустить один или несколько клиентов.

Пример реализации RMI программы

В качестве примера, рассмотрим создание каркаса RMI приложения.

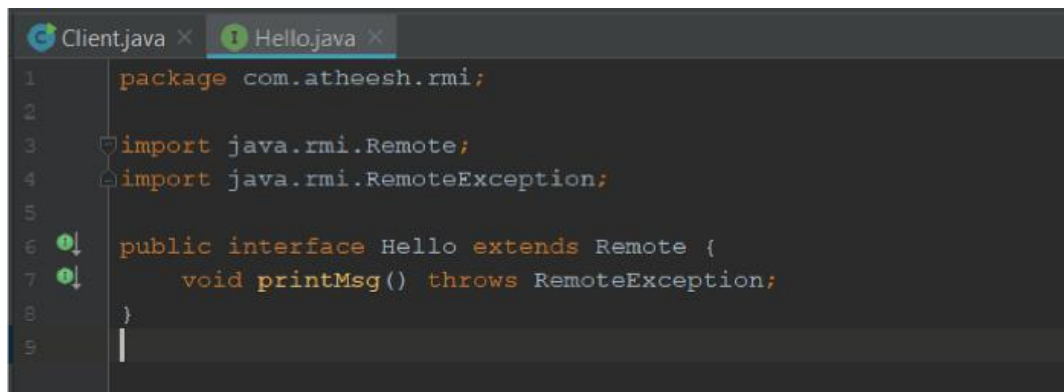
Мы можем использовать `java`, встроенный в пакет `java.rmi`, для создания приложения с технологией RMI. Будем придерживаться следующего порядка реализации:

1. Создать удаленный интерфейс.
2. Реализовать удаленного интерфейса.
3. Скомпилировать заглушку и скелет (компилятор(>`rmic`) запускается с указанием файла класса, реализующего удаленную службу).
4. Запуск реестр(если используете `Naming.rebind()`, то нужно отдельным процессом запустить сервер регистрации – `rmiregistry`).
5. Создание и запуск сервера.
6. Создание и запуск клиента.

Первым шагом является написание и компилирование Java кода для интерфейсов служб.

Когда вы создаете удаленный интерфейс, вы должны следовать следующим правилам:

1. Удаленный интерфейс должен быть публичным – **public** (он не может иметь «доступ на уровне пакета», так же он не может быть «дружественным»). В противном случае клиенты будут получать ошибку при попытке загрузки объекта, реализующего удаленный интерфейс.
2. Удаленный интерфейс должен расширять интерфейс `java.rmi.Remote`.
3. Каждый метод удаленного интерфейса должен объявлять `java.rmi.RemoteException` в своем предложении `throws` в добавок к любым исключениям, специфичным для приложения.
4. Удаленный объект, передаваемый как аргумент или возвращаемое значение (либо напрямую, либо как часть локального объекта), должен быть объявлен как удаленный интерфейс, а не реализация класса.



```
1 package com.atheesh.rmi;
2
3 import java.rmi.Remote;
4 import java.rmi.RemoteException;
5
6 public interface Hello extends Remote {
7     void printMsg() throws RemoteException;
8 }
9
```

5. Теперь необходимо написать реализацию удаленной службы. Ниже приведен соответствующий класс.



```
1 package com.atheesh.rmi;
2
3 import java.rmi.RemoteException;
4
5 public class HelloImpl implements Hello {
6     @Override
7     public void printMsg() throws RemoteException {
8         System.out.println("This is an example RMI program");
9     }
10 }
11
```

6. Далее создаётся клиентская программа.


```

Client.java
1  package com.atheesh.rmi;
2
3  import java.rmi.registry.LocateRegistry;
4  import java.rmi.registry.Registry;
5
6  public class Client {
7      private Client() {}
8      public static void main(String[] args) {
9          try {
10             // Getting the registry
11             Registry registry = LocateRegistry.getRegistry(host: null);
12
13             // Looking up the registry for the remote object
14             Hello stub = (Hello) registry.lookup(name: "Hello");
15
16             // Calling the remote method using the obtained object
17             stub.printMsg();
18
19             // System.out.println("Remote method invoked");
20         } catch (Exception e) {
21             System.err.println("Client exception: " + e.toString());
22             e.printStackTrace();
23         }
24     }
25 }
26

```

7. Удаленные службы RMI должны быть помещены в процесс сервера. Класс Server является очень простым сервером, предоставляющим простые элементы для размещения.

```

Server.java
1  package com.atheesh.rmi;
2
3  import java.rmi.registry.LocateRegistry;
4  import java.rmi.registry.Registry;
5  import java.rmi.server.UnicastRemoteObject;
6
7  public class Server extends HelloImpl {
8      public Server() {
9      }
10
11     public static void main(String args[]) {
12         try {
13             // Instantiating the implementation class
14             HelloImpl obj = new HelloImpl();
15
16             // Exporting the object of implementation class
17             // (here we are exporting the remote object to the stub)
18             Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, port: 0);
19
20             // Binding the remote object (stub) in the registry
21             Registry registry = LocateRegistry.getRegistry();
22
23             registry.bind(name: "Hello", stub);
24             System.err.println("Server ready");
25         } catch (Exception e) {
26             System.err.println("Server exception: " + e.toString());
27             e.printStackTrace();
28         }
29     }
30 }
31

```

8. Запуск RMI системы

Вы должны запустить три консоли, одну для сервера, одну для клиента и одну для реестра RMI.

Перейдите в папку, в которой хранятся все программы, с помощью `cmd` и скомпилируйте файлы.

```
Javac *.java
```

Далее запустим RMI - реестр. Вы должны находиться в каталоге, в котором находятся написанные вами классы.

```
start rmiregistry
```

Во второй консоли запустите сервер, содержащий Server, и наберите следующее:

```
Java Server
```

Программа запустится, загрузит реализацию в память и будет ожидать соединения клиента.

В последней консоли запустите клиентскую программу.

```
java Client
```

Если все пройдет хорошо, то увидите в консоли надпись "Hello":

Даже если вы запустили три консоли на одном и том же компьютере, RMI использует стек протоколов TCP/IP вашей сети для взаимодействия между тремя отдельными JVM. Это вполне законченная RMI система.

Задание на практическую работу

Используя информацию из описания данной практической работы, необходимо реализовать удалённый метод решения квадратных уравнений общего вида $ax^2 + bx + c = 0$. При этом, условие уравнения передавать на сервер, а клиентская часть должна получать результат в виде объектов пользовательского класса. Клиент и сервер должны работать на одном хосте.