Факультет «Информатика и системы управления»

Методические указания к лабораторной работе

по курсу «Распределенные системы обработки информации»

*"Удаленный вызов методов(RMI)"*

Москва 2004 г.

**Цель работы**

Ознакомиться с RMI. Написать клиент RMI и сервер RMI.

**Что такое RMI?**

Java RMI (Remote Method Invocation – удаленный вызов методов) представляет собой тип удаленного вызова процедур, независимый от сети, облегченный и полностью переносимый, так как написан на языке Java.

Основные шаги работы с RMI:

1. Определите (или найдите) удаленный интерфейс, согласованный с сервером.
2. Напишите код сервера.
3. Запустите программу rmic (Java RMI stub compiler – компилятор заглушек RMI) для генерации связующего кода.
4. Напишите код клиента.
5. Убедитесь, что на сервере запущен RMI реестр (программа rmiregistry).
6. Запустите сервер.
7. Запустите одного или нескольких клиентов.

Процедуры RMI определяют с помощью известного механизма Java – интерфейсов. Удаленные нтерфейсы долдны быть подклассами java.rmi. Remote, при этом и клиент и сервер должны находиться в одном пакете Java. Все параметры удаленных методов должны относиться или к примитивным типам (int, double и т.п.), либо реализовывать интерфейс java.io. Serializable.

Введение в распределенные вычисления с использованием RMI

Технология Remote Method Invocation (RMI), впервые представленная в JDK 1.1, продвинула сетевое программирование на более высокий уровень. Хотя RMI относительно проста в использовании, она является необыкновенно мощной технологией и раскрывает перед обычным Java‑программистом полностью новую парадигму – мир распределенных объектных вычислений.

Этот курс представляет собой углубленное введение в эту универсальную технологию. RMI получила значительное развитие в JDK 1.1 и во многом была улучшена в Java 2 SDK. При необходимости, различия между этими двумя версиями будут отмечены.

**Цели**

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.

**Сравнение распределенных и нераспределенных Java‑программ**

Разработчики RMI стремились сделать использование распределенных Java‑объектов таким же, как и использование локальных объектов. В следующей таблице перечислены некоторые важные отличия.

Не беспокойтесь о том, что не все отличия вам понятны. Все прояснится после рассмотрения архитектуры RMI. Вы можете использовать эту таблицу в качестве ссылки во время изучения RMI.

|  |  |  |
| --- | --- | --- |
| Локальный объект Удаленный объект | | |
| Определение объекта | Локальный объект определяется при помощи класса Java. | Экспортируемое поведение удаленного объекта определяется при помощи интерфейса, который должен быть расширен из интерфейса Remote. |
| Реализация объекта | Локальный объект реализуется своим классом Java. | Поведение удаленного объекта определяется классом Java, который реализует удаленный интерфейс. |
| Создание объекта | Новый экземпляр локального объекта создается оператором new. | Новый экземпляр удаленного объекта создается на компьютере хоста оператором new. Клиент не может непосредственно создать новый удаленный объект (если не использует технологию Java 2 Remote Object Activation). |
| Доступ к объекту | Доступ к локальному объекту осуществляется непосредственно через переменную-ссылку на объект. | Доступ к удаленному объекту осуществляется через переменную-ссылку на объект, указывающую на реализацию замещающей заглушки удаленного интерфейса. |
| Ссылки | В одной JVM, ссылка на объект указывает непосредственно на объект в динамической памяти. | «Удаленная ссылка» представляет собой указатель на замещающий объект («заглушку») в локальной динамической памяти. Заглушка содержит информацию, которая дает возможность соединиться с удаленным объектом, содержащим реализацию методов. |
| Активные ссылки | В одной JVM, объект считается «живым», если существует хотя бы одна ссылка на него. | В распределенной среде удаленная JVM может разрушиться, и сетевое соединение может быть потеряно. Считается, что удаленный объект имеет активную удаленную ссылку на него, если к нему производился доступ в течение определенного периода времени (срока аренды). Если все удаленные ссылки были удалены явно, или если у всех удаленных ссылок закончился срок аренды, тогда удаленный объект становится доступен для удаленной сборки мусора. |
| Финализация | Если объект реализует метод finalize(), он вызывается перед тем, как объект утилизируется сборщиком мусора. | Если удаленный объект реализует интерфейс Unreferenced, при удалении всех удаленных ссылок вызывается метод unreferenced этого интерфейса. |
| Сборка мусора | При удалении всех локальных ссылок на объект, он становится кандидатом на удаление сборщиком мусора. | Удаленный сборщик мусора работает совместно с локальным. Если нет удаленных ссылок и удалены все локальные ссылки на объект, он становится кандидатом для сборщика мусора в обычном значении этого понятия. |
| Исключительные ситуации | Исключительные ситуации являются либо исключительными ситуациями времени исполнения, либо классом Exceptions. Компилятор Java заставляет программу обрабатывать все Exceptions. | RMI заставляет программу иметь дело с любыми возможными объектами RemoteException, которые могут генерироваться. Это сделано для гарантии устойчивости распределенных приложений. |

Архитектура Java RMI

Целью разработки архитектуры RMI было создание распределенной объектной модели Java, которая свободно интегрируется в язык программирования Java и локальную объектную модель. Разработчики RMI достигли этой цели; была создана система, которая переносит безопасность и устойчивость архитектуры Java в мир распределенных вычислений.

**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.*

Следующий рисунок иллюстрирует это разделение:

Помните, что интерфейсы Java не содержат исполняемого кода. RMI поддерживает два класса, реализующих один и тот же интерфейс. Первый класс является реализацией поведения и исполняется на сервере. Второй класс работает как промежуточный интерфейс для удаленной службы и исполняется на клиентской машине. Это показано на следующей диаграмме. Клиентская программа вызывает методы прокси-объекта, RMI передает запрос на удаленную JVM и направляет его в реализацию объекта. Любые возвращаемые из реализации значения передаются назад в прокси-объект и затем в клиентскую программу.

**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 ТСР/IР 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.

**Уровень заглушки и скелета**

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. На следующей диаграмме классов показана прокси-модель.

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.

**Уровень удаленных ссылок**

Уровни удаленных ссылок определяют и поддерживают семантику вызовов соединения RMI. Этот урове 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 широковещательного 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 для выполнения упражнений этого курса). 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 ТСР/IР соединений, некоторые сетевые конфигурации разрешают только одно TCP/IP‑соединение между клиентом и сервером (некоторые броузеры ограничивают апплеты одним сетевым соединением с их сервером).

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

Именование удаленных объектов

При рассмотрении архитектуры RMI постоянно откладывался один вопрос: «Как клиент находит удаленную службу RMI?» Сейчас вы получите ответ на этот вопрос. Клиенты находят удаленные службы, используя службу имен или каталогов. Это может показаться хождением по кругу. Как клиент может найти службу, используя службу? И это действительно так. Служба имен или каталогов исполняется на хорошо известном хосте и имеет известный номер порта. *(Хорошо известный* означает, что все в организации знают об этом).

RMI может использовать много различных служб каталогов, включая Java Naming and Directory Interface (JNDI). RMI и сама включает в себя простую службу, называемую реестром RMI, rmiregistry. Реестр RMI работает на каждой машине, содержащей объекты удаленных служб и принимающей запросы на обслуживание, по умолчанию используя порт 1099.

На хосте программа сервера создает удаленную службу, предварительно создавая локальный объект, реализующий эту службу. Затем она экспортирует этот объект в RMI. Как только объект экспортирован, RMI создает службу прослушивания, ожидающую соединения с клиентом и запроса службы. После экспорта, сервер регистрирует объект в реестре RMI, используя общедоступное имя.

На стороне клиента к реестру RMI доступ обеспечивается через статический класс Naming. Он предоставляет метод lookup(), который клиент использует для запросов к реестру. Метод lookup() принимает URL, указывающий на имя хоста и имя требуемой службы. Метод возвращает удаленную ссылку на обслуживающий объект. URL принимает следующий вид:

rmi:// <host\_name>

[:<name\_service\_port>] /<service\_name>

где host\_name – это имя, распознаваемое в локальной сети (LAN), или DNS‑имя в сети Internet. Необходимо только указать name\_service\_port, если служба имен исполняется на порте, отличном от принимаемого по умолчанию 1099.

Использование RMI

Сейчас наступило время создать рабочую RMI‑систему и получить практический опыт. Вы создадите простую удаленную службу, реализующую калькулятор, и попробуете использовать ее из клиентской программы.

Рабочая RMI‑система состоит из нескольких частей.

* Определение интерфейсов для удаленных служб
* Реализация удаленных служб
* Файлы заглушки и скелета
* Сервер, предоставляющий удаленные службы
* Служба имен RMI, дающая возможность клиентам найти удаленные службы
* Поставщик файла классов (HTTP или FTP‑сервер)
* Клиентская программа, которая нуждается в удаленных службах

Для упрощения задачи вы будете использовать один и тот же каталог для кода как клиента, так и сервера. При запуске клиента и сервера из одного и того же каталога вам не придется настраивать HTTP или FTP серверы для доступа к файлам классов. (Использование серверов HTTP и FTP в качестве поставщиков файлов классов детально рассматривается в разделе «Распространение и установка программного обеспечения RMI»)

Если предположить, что RMI‑система уже спроектирована, для ее создания необходимо выполнить следующие шаги:

1. Написать и откомпилировать Java‑код для интерфейсов
2. Написать и откомпилировать Java‑код для классов реализации
3. Создать файлы классов заглушки и скелета из классов реализации
4. Написать Java‑код программы хоста для удаленного обслуживания
5. Разработать Java‑код для клиентской программы RMI
6. Установить и запустить RMI‑систему

**1. Интерфейсы**

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 к части локального объекта), должен быть объявлен как удаленный интерфейс, а не реализация класса.

Интерфейс Calculator определяет все удаленные возможности, предлагаемые службой:

public interface Calculator

extends java.rmi. Remote {public long add (long a, long b)

throws j ava.rmi. RemoteException;

public long sub (long a, long b)

throws j ava.rmi. RemoteException;

public long mul (long a, long b)

throws j ava.rmi. RemoteException;

public long div (long a, long b)

throws j ava.rmi. RemoteException;

Скопируйте этот файл в ваш каталог и откомпилируйте его при помощи компилятора Java:

>javac Calculator.java

**2. Реализация**

Теперь вы пишете реализацию удаленной службы. Ниже приведен класс CalculatorImpl:

public class CalculatorImpl extends

java.rmi.server. UnicastRemoteObj ect implements Calculator {

 // Реализации должны иметь

 // явный конструктор для

 // того, чтобы объявить

 // исключительную ситуацию RemoteException

public CalculatorImpl()

throws java.rmi. RemoteException {

super ();

public long add (long a, long b)

throws java.rmi. RemoteException {return a + b;

public long sub (long a, long b)

throws java.rmi. RemoteException {return a – b;

public long mul (long a, long b)

throws java.rmi. RemoteException {return a \* b;

public long div (long a, long b)

throws java.rmi. RemoteException {

return a / b;

}

И опять, скопируйте этот код в ваш каталог и откомпилируйте его.

Класс реализации использует UnicastRemoteOb j ect для присоединения к системе RMI. В данном примере класс реализации непосредственно расширяет UnicastRemoteOb j ect. Это не является обязательным требованием. Класс, не расширяющий UnicastRemoteObj ect, может использовать свой метод exportOb j ect () для присоединения к RMI.

Если класс расширяет UnicastRemoteObj ect, он должен обеспечить конструктор, объявляющий, что он может сгенерировать объект RemoteException. Если этот конструктор вызывает метод super (), он активизирует код в UnicastRemoteObj ect, который выполняет RMI‑соединение и инициализацию удаленного объекта.

**3. Заглушки и скелеты**

Дальше вы используете компилятор RMI, rmic, для генерации файлов заглушки и скелета. Компилятор запускается с указанием файла класса, реализующего удаленную службу.

>rmic CalculatorImpl

Попробуйте выполнить это в вашем каталоге. После запуска rmic вы должны найти файл Calculator\_Stub. class.

**4. Хост-сервер**

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

import java.rmi. Naming;

public class CalculatorServer {

public CalculatorServer () {try {

Calculator c = new CalculatorImpl();

Naming.rebind (»

rmi://localhost:1099/

CalculatorService», c);} catch (Exception e) {

System.out.println («Trouble:» + e);

public static void main (String args[]) {new CalculatorServer();

В этом примере вы видите вызов статического метода **Naming.rebind()**. Однако этот вызов требует, чтобы регистрация была запущена отдельным процессом на вашем компьютере. Имя сервера регистрации – это **rmiregistry**, и под 32‑битной Windows вы пишете:

start rmiregistry

для запуска в фоновом режиме.

Как и многие другие сетевые программы, **rmiregistry** обращается по IP адресу машины, на которой она установлена, но она также слушает порт. Если вы вызовите **rmiregistry** как показано выше, без аргументов, будет использован порт по умолчанию 1099. Если вы хотите использовать другой порт, вы добавляете аргумент в командную строку, указывающий порт. Следующий пример устанавливает порт 2005, так что **rmiregistry** под управлением 32-битной Windows должна запускаться так:

start rmiregistry 2005

Информация о порте также должна передаваться в команде **bind()**, наряду с IP адресом машины, где располагается регистрация. Но это может выявить проблему, если вы хотите проверять RMI программы локально. В выпуске JDK 1.1.1, есть целая связка проблем:

1) **localhost** не работает с RMI. Поэтому для экспериментов с RMI на одной машине вы должны использовать имя машины. Чтобы найти имя вашей машины под управлением 32‑битной Windows, перейдите в панель управления и выберите «Network». Выберите закладку «Identification», и посмотрите имя вашего компьютера. Регистр в имени игнорируется. (Пример имени: «peppy»)

2) RMI не работает, пока ваш компьютер имеет активные TCP/IP соединения, даже если все ваши компоненты просто общаются друг с другом на локальной машине. Это значит, что вы должны соединятся с вашим провайдером Internet до того, как попробуете запустить программу или будете огорчены неким сообщением об ошибке.

Если учесть все это, команда **bind()** принимает вид:

Naming.bind (« //peppy:2005/CalculatorService», с);

Если вы используете порт по умолчанию 1099, вам не нужно указывать порт, так что вы можете просто сказать:

Naming.bind (« //peppy/CalculatorService», с);

Вы можете выполнить локальную проверку, оставив в покое IP адрес, а использовать только идентификатор:

Naming.bind («CalculatorService», с);

Имя сервиса здесь произвольно. В данном случае CalculatorService выбрано просто как имя класса, но вы можете назвать так, как захотите. Важно, чтобы это было уникальное имя регистрации, чтобы клиент знал, когда будет искать, что производит удаленные объекты. Если имя уже зарегистрировано, вы получите **AlreadyBoundException**. Чтобы предотвратить это, вы всегда можете использовать **rebind()** вместо **bind()**, так как **rebind()** либо добавляет новый элемент, либо заменяет уже существующий.

Даже после завершения работы **main()**, ваш объект будет оставаться созданным и зарегистрированным, ожидая, что придет клиент и выполнит запрос. Пока **rmiregistry** остается запущенным, и вы не вызовите **Naming.unbind()** на вашей машине, объект будет оставаться там. По этой причине, когда вы разрабатываете ваш код, вам необходимо выгружать **rmiregistry** и перезапускать его, когда скомпилируете новую версию вашего удаленного объекта.

Вам не обязательно запускать **rmiregistry** как внешний процесс. Если вы знаете, что только ваше приложение использует регистрацию, вы можете загрузить ее внутри вашей программы с помощью строки:

LocateRegistry.createRegistry(2005);

Как и раньше, 2005 – это номер порта, который мы использовали в этом примере. Это эквивалентно запуску **rmiregistry 2005** из командной строки, но часто этот способ является более подходящим при разработке RMI кода, так как это снжает число необходимых действий при запуске и остановке регистрации После того, как вы выполните этот код, вы можете вызвать **bind()**, используя **Naming**, как и ранее.

**5. Клиент**

Исходный код клиента следующий:

import java.rmi. Naming;•

import java.rmi. RemoteException;

import java.net. MalformedURLException;

import java.rmi. NotBoundException;

public class CalculatorClient {

public static void main (String[] args) {try {

Calculator c = (Calculator)

Naming.lookup («rmi://remotehost

/CalculatorService»); System.out.println (c.sub (4, 3)); System.out.println (c.add (4, 5)); System.out.println (c.mul (3, 6)); System.out.println (c.div (9, 3));

}

catch (MalformedURLException murle) {

System.out.println();

System.out.println (

«MalformedURLException»);

System.out.println(murle);

}

catch (RemoteException re) {

System.out.println();

System.out.println (

«RemoteException»);

System.out.println(re);} catch (NotBoundException nbe) {

System.out.println();

System.out.println (

«NotBoundException»); System.out.println(nbe);

} catch (

java.lang. ArithmeticException

ae) {

System.out.println(); System.out.println (

«java.lang. ArithmeticException»); System.out.println(ae);

**6. Запуск RMI‑системы**

Теперь вы готовы к запуску системы! Вы должны запустить три консоли, одну для сервера, одну для клиента и одну для реестра RMI.

Начните с реестра. Вы должны находиться в каталоге, в котором находятся написанные вами классы. Напишите следующее:

Rmiregistry

Если не сработает, то в текущем каталоге наберите полный путь к файлу rmiregistry.exe, Он находится в каталоге JAVA\_HOME/bin/

Если все пройдет хорошо, реестр начнет работать, и вы можете перейти к следующей консоли.

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

>java CalculatorServer

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

В последней консоли запустите клиентскую программу.

>java CalculatorClient

Если все пройдет хорошо, вы увидите следующую информацию:

1

9

18

3

Вот и все: вы создали работающую систему RMI. 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.

# Литература

1. Серия «Библиотека профессионала» К. Хорстманн Г. Корнелл «Java 2. том 2 «Тонкости программирования»» «Издательство Вильямс» 2002 г.

2.Методичка с сайта JGURU.ru

3. Sun Java 2 SE API