|  |
| --- |
|  |

|  |
| --- |
| **Лабораторная работа №1:**  **Технология Java TM Remote Method Invocation (Java RMI)**  **Краткие теоретические сведения**  **Remote Procedure Call (RPC)**  Распределенные системы требуют, чтобы объекты вычислений, запущенные в различных адресных пространствах, потенциально на различных компьютерах, могли «общаться» между собой. Для базовой модели коммуникации Java TM  поддерживает сокеты, которые являются довольно гибким и удовлетворительным механизмом. Однако сокеты требуют наличия некоторых протоколов, которые бы кодировали/декодировали сообщения для обмена, и проектирование таких протоколов – подверженное ошибкам, тяжелое занятие.  Альтернатива сокетам – механизм Remote Procedure Call (RPC), что означает удаленный вызов процедур. Вместо работы напрямую с сокетами, создается иллюзия, что программист вызывает локальную процедуру, когда, на самом деле, аргументы вызова пакуются и отправляются на удаленную цель вызова. RPC-системы кодируют аргументы и возвращают значения, используя внешнее представление данных, такое как XDR (подробнее на http://www.ltg.ed.ac.uk/~ht/XMLData-Reduced.htm).  **Remote Method Call (RMI)**  RPC, однако, плохо вписывается в системы распределенных объектов, где необходима коммуникация между объектами уровня программы, которые располагаются в различных адресных пространствах. Для того чтобы соответствовать семантике вызовов объектов, системы распределенных объектов требуют вызовов удаленных методов (RMI). В таких системах локальный объект-суррогат (*stub*) управляет вызовом удаленного объекта. RMI на Java-платформе специально создан для работы в среде Java-приложений.  Приложения на основе RMI обычно состоят из двух отдельных программ–   сервера и клиента. Типичное серверное приложение создает несколько удаленных объектов, делает на них ссылки и ожидает клиентских вызовов этих удаленных объектов. Типичное клиентское приложение получает удаленную ссылку на один или  больше удаленных объектов на сервере и затем вызывает на нем методы. RMI обеспечивает механизм, по которому сервер и клиент сообщаются между собой и отправляют информацию туда и обратно. На такое приложение обычно ссылаются как на распределенный объект.  Приложениям, основанным на распределенных объектах необходимо  ·        *Разместить удаленные объекты*: приложения могут использовать один или два механизма, чтобы получить ссылки на удаленные объекты. Приложение может зарегистрировать свои удаленные объекты с помощью средства *rmiregistry*, либо приложение может передавать и возвращать ссылки на удаленные объекты как часть своей нормальной работы.  ·        *Сообщаться с удаленными объектами*: детали коммуникации между удаленными объектами обрабатываются с помощью RMI; для программиста удаленная коммуникация выглядит как стандартный вызов методов.  ·        *Загрузить байт-коды отправляемых классов объектов*: RMI обеспечивает необходимые механизмы для загрузки кода объектов.  На рис.1 показано распределенное приложение RMI, которое использует регистратор (*registry*) для получения ссылки на удаленный объект. Сервер вызывает регистратор для ассоциации (или для связывания) имени с удаленным объектом. Клиент ищет удаленный объект по его имени в регистраторе сервера, а затем вызывает на нем метод. На рисунке также видно, что система RMI использует существующий Web-сервер для загрузки байт-кода классов от  сервера на клиент и от клиента на сервер.  D:\sit\RMI_lab\RMI\image002.jpg  Рис. 1. Архитектура распределенного приложения.  **Пакет java.rmi**  Интерфейсы и классы, которые ответственные за работу с удаленными объектами в системе RMI, определены в иерархии пакета *java.rmi*. На рис.2 показаны отношения между некоторыми интерфейсами и классами.  D:\sit\RMI_lab\RMI\image004.jpg  Рис 2. Иерархия пакета *java.rmi*.  Интерфейс *java.rmi.Remote*  – это интерфейс, который объявляет набор методов, которые могут быть вызваны с удаленной машины. Удаленный интерфейс должен удовлетворять следующим требованиям:  1) удаленный интерфейс должен расширять интерфейс *java.rmi.Remote*.  2) каждое объявление метода в удаленном интерфейсе или его над-интерфейсах должно включать исключение *java.rmi.RemoteException* (или его суперклассы) в разделе throws вдобавок к остальным имеющимся исключениям.  3) в объявлении удаленного метода удаленный объект, объявленный как параметр или возвращаемое значение должен быть объявлен как удаленный интерфейс, а не реализация (implementation) класса того интерфейса.  Класс *RemoteException* является суперклассом исключений, выбрасываемых средой выполненя RMI во время удаленного вызова. Причины сбоя могут быть следующими:  ·        Ошибка коммуникации (удаленный сервер недоступен или не может установить соединение)  ·        Ошибка маршаллизации/демаршаллизации (упаковка/распаковка параметров)  ·        Ошибки протокола  Общие правила для класса, который реализует удаленный интерфейс:  ·        Класс обычно расширяет *java.rmi.server.UnicastRemoteObject*  ·        Класс может реализовывать любое  количество удаленных интерфейсов  ·        Класс может расширять другой удаленный класс  ·        Класс может определять методы, которые не находятся на удаленном интерфейсе, но использовать их следует только локально.  **Stub и skeleton**  RMI использует стандартный механизм (внедренный в RPC-системы) для сообщения между удаленными объектами: стабы (*stub*) и скелетоны (*skeleton*). Стаб для удаленного объекта действует как локальный представитель на клиенте или прокси (*proxy*) для удаленного объекта. Вызывающий объект вызывает метод на локальном стабе, который ответственен за доставку вызова метода на удаленный объект. В RMI стаб для удаленного объекта реализует тот же самый набор удаленных интерфейсов, что и удаленный объект.  Когда вызывается метод стаба, он делает следующее:  ·        Инициализирует соединение с удаленной JVM, содержащей удаленный объект  ·        Маршализует параметры на удаленную JVM  ·        Ожидает результат вызова метода  ·        Демаршализует (читает) возвращенное значение или исключение и возвращает это значение вызывающему объекту.  ·        Стаб прячет скрывает  сериализацию параметров и низкоуровневое сетевое взаимодействие, чтобы представить простой механизм вызовов.  На удаленной JVM каждый удаленный объект может иметь соответствующий скелетон. Однако начиная с Java 1.2, скелетоны не обязательны, поэтому их назначение здесь не рассматривается, ввиду непрактичности (подробнее см. в документации к J2SE 1.5).  Данную лабораторную работу рекомендуется выполнять, использую J2SE 1.5, поэтому рассмотрим некоторые отличия в реализации RMI между этой (пятой) и предыдущей (четвертой) версиями JDK.  Самая важная особенность – это динамическая генерация классов стаба (*stubclasses*) во время выполнения. Т.е. теперь **устраняется необходимость использовать *rmic–* компилятор стабов**. Генерация стабов вручную необходима, если известно что версия java-компилятора клиента ниже чем пятая, а также по ряду других причин.  Также изменилась версия по умолчанию протокола генерации стаба. Теперь она –v1.2. (-v – опция rmic).  **Методические указания**  **Задание**  *Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. Сервер отправляет строку «Hello, world!» клиенту, который выводит ее на экран.*  Для выполнения задания необходимо проделать следующие шаги:  1)    определить удаленный интерфейс  2)    реализовать сервер  3)    реализовать клиент  4)    скомпилировать исходные файлы  5)    запустить RMI регистратор, сервер и клиент.  **Определение удаленного интерфейса**  Он расширяет *java.rmi.Remote* и объявляет набор удаленных методов. Каждый удаленный метод должен выбрасывать *java.rmi.RemoteException* (или его суперкласс). Мы определяем только один метод – *sayHello()*, который возвращает строку клиенту:  **import java.rmi.Remote;**  **import java.rmi.RemoteException;**  **public interface Hello extends Remote {**  **String sayHello() throws RemoteException;}**  **Реализация сервера**  Класс сервера в данном контексте – это класс, который имеет метод *main()*, который создает экземпляр реализации удаленного объекта, экспортирует удаленный объект, а затем привязывает этот экземпляр имени в Java  RMI регистраторе.  **import java.rmi.registry.Registry;**  **import java.rmi.registry.LocateRegistry;**  **import java.rmi.RemoteException;**  **import java.rmi.server.UnicastRemoteObject;**  **public class Server implements Hello {**  **public Server() {}**  **//** реализация интерфейса  **public String sayHello()**{  **return "Hello, world!";**  }  **public static void main(String args[]) {**  **try {**  **//**создаем и экспортируем удаленный объект  **Server obj = new Server();**  **Hello stub =(Hello) UnicastRemoteObject.exportObject(obj, 0);**  **//**Регистрируем удаленный объект в RMI-регистраторе под именем  **Registry registry = LocateRegistry.createRegistry(1087);**  **registry.bind("Hello", stub);**  **System.err.println("Server ready");**  **} catch (Exception e) {**  **System.err.println("Server exception: " + e.toString());**  **e.printStackTrace();**  }  }  }  Статический метод *UnicastRemoteObject.exportObject()* экспортирует удаленный объект и возвращает стаб удаленного объекта для отправки клиенту. Возвращенный стаб реализует тот же набор удаленных интерфейсов, что и класс удаленного объекта и содержит имя хоста и порт, по которому можно связаться с удаленным объектом. Заметим, что в версии J2SE 1.5 отпадает необходимость предварительно генерировать стаб с помощью *rmic* (см. выше).  RMI-регистратор – это упрощенный сервис имен, который позволяет клиентам получить ссылку (стаб) на удаленный объект.  Как только удаленный объект зарегистрирован на сервере, программа, которая намеревается его вызвать, ищет объект по имени, получает ссылку, а затем вызывает удаленный метод.  Статический метод *LocateRegistry.getRegistry()*, который не принимает никаких аргументов, возвращает стаб, который реализует удаленный интерфейс  *java.rmi.registry.Registry* и отправляет вызовы на регистратор на хост сервера по порту по умолчанию 1099. При этом регистратор должен быть запущен (см. далее). Метод *bind()* затем вызывает стаб регистратора для связи стаба удаленного объекта с именем «Hello» в регистраторе.  **Реализация клиента**  Клиентская программа получает стаб для регистратора на серверной машине, ищет стаб удаленного объекта по имени в регистраторе, затем вызывает метод *sayHello()* на удаленном объекте, используя этот стаб.  **import java.rmi.registry.LocateRegistry;**  **import java.rmi.registry.Registry;**  **public class Client {**  **private Client() {}**  **public static void main(String[] args) {**  **String host = (args.length < 1) ? null : args[0];**  **try {**  **//**Получаем стаб регистратора с хоста, определенного в командной строке  // если в командной строке хост не указывается, то он считается как localhost  **Registry registry = LocateRegistry.getRegistry(host);**  // получаем стаб удаленного объекта от регистратора сервера  **Hello stub = (Hello) registry.lookup("Hello");**  **String response = stub.sayHello();**  **System.out.println("response: " + response);**  **} catch (Exception e) {**  **System.err.println("Client exception: " + e.toString());**  **e.printStackTrace();**  **}**  **}**  **}**  **Компиляция исходников**  javac - d destDir Hello.java Server.java Client.java,  где destDir – это директория, куда будут помещены файлы классов.  **Запуск**  Чтобы запустить регистратор, запускаем команду *rmiregistry* на хосте сервера (по умолчанию порт 1099):**start** **rmiregistry**  Запуск сервера:  **java -classpathclassDirServer**  где classDir – это корневая директория файлов классов.  Сервер должен вывести сообщение: **Serverready**  Запуск клиента:  **java  - classpath classDir Client**  где classDir – это корневая директория файлов классов.  Клиент должен вывести сообщение: **response: Hello, world!**    **Контрольные вопросы**  1.     В чем преимущество использование технологии RMI?  2.     Из каких компонентов состоят приложения на основе RMI?  3.     Какой пакет содержит интерфейсы и классы, ответственные за работу с удаленными объектами в системе RMI?  4.     Каким требованиям должен удовлетворять интерфейс удаленного класса?  5.     Какие ситуации могут быть причиной возбуждения исключительной ситуации *RemoteException*?  6.     Каким требованиям должен удовлетворять класс, реализующий удаленный интерфейс?  7.     Что такое стабы и скелетоны? Как работает стаб?  8.     Какие изменения RMI-технологии имеются в J2SE 1.5 по сравнению с предыдущими версиями ОВЛ,  9.     Для чего предназначен и как работает метод *LocateRegistry.getRegistry()*?  10.     Как запустить регистратор с портом по умолчанию?    **Варианты индивидуального задания**   1. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте водится строка.  Сервер возвращает эту строку без символов «а» клиенту, который выводит ее на экран. 2. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте вводятся два числа и знак арифметической операции.  Сервер возвращает результат клиенту, который выводит его на экран. 3. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте вводятся две буквы алфавита.  Сервер возвращает вид алфавита (кириллица/латиница) клиенту, который выводит результат на экран. 4. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте вводится строка через пробелы.  Сервер возвращает эту строку склеенной клиенту, который выводит ее на экран. 5. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте вводится строка.  Сервер возвращает эту строку, но с гласными, замененными символом  «\*» клиенту, который выводит ее на экран. 6. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте вводится число.  Сервер возвращает факториал этого числа клиенту, который выводит его на экран. 7. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте вводится дата рождения (например, 24.07).  Сервер возвращает знак зодиака, соответствующий этой дате клиенту, который выводит его на экран. 8. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте вводятся два числа.  Сервер возвращает их НОК клиенту, который выводит результат на экран. 9. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте вводится год (например, 1984).  Сервер возвращает високосный/невисокосный год клиенту, который выводит результат на экран. 10. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте вводится строка.  Сервер возвращает эту строку, в которой четные символы заменены на «ab», а нечетные – на «cd», клиенту, который выводит ее на экран. 11. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте вводится число.  Сервер возвращает это число в виде римского клиенту, который выводит его на экран. 12. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте вводится строка из слов через пробелы.  Сервер возвращает эту строку, в которой первые буквы каждого слова заменены на заглавные,  клиенту, который выводит ее на экран. 13. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте водится строка.  Сервер возвращает эту строку, в которой две подряд идущие гласные разделены пробелом, клиенту, который выводит ее на экран. 14. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте вводится число.  Сервер возвращает это число, разложенное на единицы, десятки, сотни и т.д., клиенту, который выводит результат на экран. 15. Клиент удаленно вызывает метод сервера, который может быть запущен на удаленном компьютере. На клиенте вводится строка, состоящая из символов и чисел.  Сервер возвращает сумму чисел этой сроки клиенту, который выводит ее на экран. |