**Лабораторная работа №3**

**«Разработка клиент-серверных приложений на основе сокетов»**

|  |
| --- |
| **Цель работы:** изучить методы разработки клиент-серверных приложений, научиться создавать приложение-сервер и приложение-клиент, обменивающиеся данными через стандартный интерфейс.  **Задание на лабораторную работу**  Изучить классы пакета **java.net**, которые отвечают за различные аспекты сетевого взаимодействия, технологию работы с сокетами. Разработать приложение «клиент-сервер» согласно варианту задания.  **Краткие теоретические сведения**  Клиент-серверные приложения (англ. Client-server) — архитектура программного обеспечения, в которой программы являются либо клиентами, либо серверами. Клиентом (front end) является запрашивающая программа, сервером (back end) — программа, которая отвечает на запрос. Оба термина (клиент и сервер) могут применяться как к физическим устройствам, так и к программному обеспечению.  Стандартные клиент-серверные приложения по своей архитектуре являются двухуровневыми. Компоненты, отвечающие за презентационный сервис и прикладную логику, размещаются на клиентской машине и обращаются к общему источнику данных по сети. В таких приложениях пользовательский интерфейс и прикладная логика образуют один уровень на клиентском компьютере, а сервисы данных предоставляются отдельными приложениями. Двухуровневые приложения хорошо работают в масштабах подразделения при наличии умеренного количества пользователей (до 100) и быстродействующей защищенной сети. Отладку приложений клиент-сервер возможно производить на локальном компьютере.  Для организации взаимодействия между клиентом и сервером используются различные механизмы: обмен сообщениями, разделяемая память, совместные файлы, базы данных и т.п. В данной лабораторной работе используется механизм сокетов (socket, гнезд), обеспечивающих интерфейс между приложениями в рамках как локального компьютера, так и в локальной сети.  Сокеты Беркли — интерфейс программирования приложений (API), представляющий собой библиотеку для разработки приложений на языке Си с поддержкой межпроцессного взаимодействия (англ. *Inter-Process Communication,****IPC***), часто применяемого в компьютерных сетях. Сокеты Беркли (также известные как API сокетов BSD - Berkeley Software Distribution, иногда называемые **Berkeley Unix**), впервые и появились как API в операционой системе 4.2 BSD Unix (выпущенной в 1983 году). Тем не менее, только в 1989 году, Калифорнийский университет в Беркли смог начать выпускать версии операционной системы и сетевой библиотеки без лицензионных ограничений AT&T, действующих в защищенной авторским правом Unix. API сокетов Беркли сформировал de facto стандарт абстракции для сетевых сокетов.  Интерфейс сокета Беркли позволяет реализацию взаимодействия между компьютерами или между процессами на одном компьютере, используя концепцию Интернет-сокетов. Данная технология может работать со множеством различных устройств ввода/вывода и драйверов, несмотря на то, что их поддержка зависит от реализации операционной системы. Подобная реализация интерфейса лежит в основе TCP/IP, благодаря чему считается одной из фундаментальных технологий, на которых основывается Интернет. Технология сокетов впервые была разработана в Калифорнийском университете Беркли для применения на Юникс-системах. Все современные операционные системы имеют ту или иную реализацию интерфейса сокетов Беркли, так как это стало стандартным интерфейсов для подключения к сети Интернет.  В качестве примера можно привести исходные тексты двух приложений Java, работающих с потоковыми сокетами. Одно из этих приложений называется SocketServ и выполняет роль сервера, второе называется SocketClient и служит клиентом. Приложение SocketServ выводит на консоль строку «Socket Server Application» и затем переходит в состояние ожидания соединения с клиентским приложением SocketClient. Приложение SocketClient устанавливает соединение с сервером SocketServ, используя потоковый сокет с номером 9999 (этот номер выбран произвольно). Далее клиентское приложение выводит на свою консоль приглашение для ввода строк. Введенные строки отображаются на консоли и передаются серверному приложению. Сервер, получив строку, отображает ее в своем окне и посылает обратно клиенту. Клиент выводит полученную от сервера строку на консоли. Когда пользователь вводит строку «quit», цикл ввода и передачи строк завершается. Весь процесс показан на рисунке 1. |
| http://sergeev.sebastopol.ua/l01/pic3_1.jpg |
| Рисунок 1 - Пример работы клиент-серверного приложения |
| Здесь в окне клиентского приложения введено несколько строк, причем последняя строка была строкой «quit», завершившая работу приложений.  ***Исходный текст серверного приложения SocketServ***  import java.io.\*;  import java.net.\*;  import java.util.\*;  public class SocketServ  {  public static void main(String args[])  {  byte bKbdInput[] = new byte[256];  ServerSocket ss;  Socket s;  InputStream is;  OutputStream os;  try  {  System.out.println(  "Socket Server Application");  }  catch(Exception ioe)  {  System.out.println(ioe.toString());  }  try  {  ss = new ServerSocket(9999);  s = ss.accept();  is = s.getInputStream();  os = s.getOutputStream();  byte buf[] = new byte[512];  int lenght;  while(true)  {  lenght = is.read(buf);  if(lenght == -1)  break;  String str = new String(buf, 0);  StringTokenizer st;  st = new StringTokenizer(  str, "\r\n");  str = new String(  (String)st.nextElement());  System.out.println("> " + str);  os.write(buf, 0, lenght);  os.flush();  }  is.close();  os.close();  s.close();  ss.close();  }  catch(Exception ioe)  {  System.out.println(ioe.toString());  }  try  {  System.out.println("Press Enter to terminate application...");  System.in.read(bKbdInput);  }  catch(Exception ioe)  {  System.out.println(ioe.toString());  }  }  }  ***Описание исходного текста серверного приложения SocketServ***  В методе main, получающем управление сразу после запуска приложения, определено несколько переменных. Массив bKbdInput размером 256 байт предназначен для хранения строк, введенных при помощи клавиатуры. В переменную ss класса ServerSocket будет записана ссылка на объект, предназначенный для установления канала связи через потоковый сокет (но не ссылка на сам сокет):  ServerSocket ss;  Ссылка на сокет, с использованием которого будет происходить передача данных, хранится в переменной с именем s класса Socket:  Socket s;  Кроме того, определены переменные is и os, соответственно, классов InputStream и OutputStream:  InputStream is;  OutputStream os;  В эти переменные будут записаны ссылки на входной и выходной поток данных, которые связаны с сокетом. После отображения на консоли строки названия приложения, метод main создает объект класса ServerSocket, указывая конструктору номер порта 9999:  ss = new ServerSocket(9999);  Конструктор возвращает ссылку на объект, с использованием которого можно установить канал передачи данных с клиентом. Канал устанавливается методом accept:  s = ss.accept();  Этот метод переводит приложение в состояние ожидания до тех пор, пока не будет установлен канал передачи данных. Метод accept в случае успешного создания канала передачи данных возвращает ссылку на сокет, с применением которого нужно принимать и передавать данные. На следующем этапе сервер создает входной и выходной потоки, вызывая для этого методы getInputStream и getOutputStream, соответственно:  is = s.getInputStream();  os = s.getOutputStream();  Далее приложение подготавливает буфер buf для приема данных и определяет переменную length, в которую будет записываться размер принятого блока данных:  byte buf[] = new byte[512];  int lenght;  Теперь все готово для запуска цикла приема и обработки строк от клиентского приложения. Для чтения строки мы вызываем метод read применительно ко входному потоку:  lenght = is.read(buf);  Этот метод возвращает управление только после того, как все данные будут прочитаны, блокируя приложение на время своей работы. Если такая блокировка нежелательна, следует выполнять обмен данными через сокет в отдельной задаче. Метод read возвращает размер принятого блока данных или -1, если поток исчерпан. Можно воспользоваться этим обстоятельством для завершения цикла приема данных:  if(lenght == -1) break;  После завершения приема блока данных массив преобразуется в текстовую строку str класса String, удаляя из нее символ перевода строки, и результат отображается на консоли сервера:  System.out.println("> " + str);  Затем полученная строка отправляется обратно клиентскому приложению, для чего вызывается метод write:  os.write(buf, 0, lenght);  Методу write передается ссылка на массив, смещение начала данных в этом массиве, равное нулю, и размер принятого блока данных. Для исключения задержек в передаче данных из-за накопления данных в буфере (при использовании буферизованных потоков) необходимо принудительно сбрасывать содержимое буфера метдом flush:  os.flush();  И хотя в данном случае не используются буферизованные потоки, можно включить вызов этого метода для примера. Теперь о завершающих действиях после прерывания цикла получения, отображения и передачи строк. Данное приложение явным образом закрывает входной и выходной потоки данных, сокет, а также объект класса ServerSocket, с использованием которого был создан канал передачи данных:  is.close();  os.close();  s.close();  ss.close();  ***Исходный текст клиентского приложения***  import java.io.\*;  import java.net.\*;  import java.util.\*;  public class SocketClient  {  public static void main(String args[])  {  byte bKbdInput[] = new byte[256];  Socket s;  InputStream is;  OutputStream os;  try  {  System.out.println(  "Socket Client Application" +  "\nEnter any string or" +  " 'quit' to exit...");  }  catch(Exception ioe)  {  System.out.println(ioe.toString());  }  try  {  s = new Socket("localhost",9999);  is = s.getInputStream();  os = s.getOutputStream();  byte buf[] = new byte[512];  int length;  String str;  while(true)  {  length = System.in.read(bKbdInput);  if(length != 1)  {  str = new String(bKbdInput, 0);  StringTokenizer st;  st = new StringTokenizer(str, "\r\n");  str = new String((String)st.nextElement());  System.out.println("> " + str);  os.write(bKbdInput, 0, length);  os.flush();  length = is.read(buf);  if(length == -1)  break;  str = new String(buf, 0);  st = new StringTokenizer(str, "\r\n");  str = new String((String)st.nextElement());  System.out.println(">> " + str);  if(str.equals("quit")) break;  }  }  is.close();  os.close();  s.close();  }  catch(Exception ioe)  {  System.out.println(ioe.toString());  }  try  {  System.out.println(  "Press <Enter> to " + "terminate application...");  System.in.read(bKbdInput);  }  catch(Exception ioe)  {  System.out.println(ioe.toString());  }  }  }  ***Описание исходного текста клиентского приложения SocketClient***  Внутри метода main клиентского приложения SocketClient определены переменные для ввода строки с клавиатуры (массив bKbdInput), сокет s класса Socket для работы с сервером SocketServ, входной поток is и выходной поток os, которые связаны с сокетом s. После вывода на консоль приглашающей строки клиентское приложение создает сокет, вызывая конструктор класса Socket:  s = new Socket("localhost",9999);  В процессе отладки сервер и клиент запускались на одном и том же узле, поэтому в качестве адреса сервера указана строка «localhost». Номер порта сервера SocketServ равен 9999, поэтому конструктору передали это значение. После создания сокета клиентское приложение создает входной и выходной потоки, связанные с этим сокетом:  is = s.getInputStream();  os = s.getOutputStream();  Теперь клиентское приложение готово обмениваться данными с сервером. Этот обмен выполняется в цикле, условием завершения которого является ввод пользователем строки «quit». Внутри цикла приложение читает строку с клавиатуры, записывая ее в массив bKbdInput:  length = System.in.read(bKbdInput);  Количество введенных символов сохраняется в переменной length. Далее если пользователь ввел строку, а не просто нажал на клавишу «Enter», эта строка отображается на консоли и передается серверу:  os.write(bKbdInput, 0, length);  os.flush();  Сразу после передачи сбрасывается буфер выходного потока. Далее приложение читает ответ, посылаемый сервером, в буфер buf:  length = is.read(buf);  Следует напомнить, что сервер посылает клиенту принятую строку в неизменном виде. Если сервер закрыл канал, то метод read возвращает значение -1. В этом случае прерывается цикл ввода и передачи строк:  if(length == -1) break;  Если же ответ сервера принят успешно, принятые данные записываются в строку str, которая отображается на консоли клиента:  System.out.println(">> " + str);  Перед завершением своей работы клиент закрывает входной и выходной потоки, а также сокет, на котором выполнялась передача данных:  is.close();  os.close();  s.close(); |

***Потоки ввода-вывода***

Для отправки и получения данных при организации сетевого взаимодействия использовались потоки данных InputStream, OutputStream.

*Поток данных (stream)* представляет из себя абстрактный объект, предназначенный для получения или передачи данных единым способом, независимо от связанного с потоком источника или приемника данных.

Потоки реализуются с помощью классов, входящях в пакет java.io. Потоки делятся на две больших группы — потоки ввода, и потоки вывода. Потоки ввода связаны с источниками данных, потоки вывода — с приемниками данных. Кроме того, потоки делятся на байтовые и символьные. Единицей обмена для байтовых потоков является байт, для символьных — символ Unicode.

|  |  |  |
| --- | --- | --- |
|  | Потоки ввода | Потоки вывода |
| Байтовые | InputStream | OutputStream |
| Символьные | Reader | Writer |

Класс InputStream представляет абстрактный входной поток байтов и является предком для всех входных байтовых потоков.

Создает входной байтовый поток конструктор:

InputStream()

Основные методы этого класса связаны с операцией чтения байтов:

abstract int read() throws IOException

читает очередной байт данных из входного потока. Значение должно быть от 0 до 255. При достижении конца потока возвращается -1. При ошибке ввода-вывода генерируется исключение. Подклассы должны обеспечить реализацию данного метода.

int read(byte[] buf)

читает данные в буфер и возвращает количество прочитанных байтов.

int read(byte[] buf, int offset, int len)

читает не более len байтов в буфер, заполняя его со смещением offset, и возвращает количество прочитанных байтов

void close()

закрывает поток.

Класс OutputStream представляет абстрактный выходной поток байтов и является предком для всех выходных байтовых потоков.

Создает выходной байтовый поток конструктор:

OutputStream()

Основные методы этого класса связаны с операцией чтения байтов:

abstract void write(int n) throws IOException

записывает очередной байт данных в выходной поток. Значащими являются 8 младших битов, старшие - игнорируются При ошибке ввода-вывода генерируется исключение. Подклассы должны обеспечить реализацию данного метода.

void write(byte[] buf)

записывает в поток данные из буфера.

void write(byte[] buf, int offset, int len)

записывает в поток len байтов из буфера, начиная со смещения offset

void close()

закрывает поток.

Кроме этих основных потоков, в пакет входят специализированные потоки, предназначенные для работы с различными источниками или приемниками данных (файл, массив, строка, конвейер), а также преобразующие потоки, предназначенные для преобразования информации, поступающей на вход потока, и выдачи ее на выход в преобразованном виде, например, преобразование байтовых потоков в символьные и наоборот. Для чтения/записи и преобразования набора байт в строку и обратно используются специальные классы из пакета java.io: InputStreamReader и OutputStreamReader. Эти два класса специально предназначены для преобразования потока байт в указанной кодировке (чем по умолчанию является кодировка системы) в Unicode строку Java. Чтобы избежать путаницы с кодировками русского языка в тексте программы (их достаточно много DOS/Win/UNIX), у компилятора javac есть специальный параметр encoding, который определяет кодировку исходного текста программы. Конструкторы таких классов в качестве аргумента принимают поток данных.

Например, для записи строкового массива в файл можно использовать метод со связкой BufferedWriter, OutputStreamWriter, FileOutputStream:

public boolean writeTextFile(String pathToFile, String[] allStrings, String enc)

{

try

{

BufferedWriter out;

out = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(pathToFile), enс));

for (int i = 0; i < allStrings.length; i++)

{

out.write(allStrings[i]);

out.write('\n');

}

out.close();

}

catch (IOException e)

{

return false;

}

return true;

}

Соответствующий метод для считывания содержимого файла в строковый массив может комбинировать BufferedReader, InputStreamReader и FileInputStream:

public String[] readTextFile(String pathToFile, String enc)

{

BufferedReader br = null;

try

{

br = new BufferedReader(new InputStreamReader(new FileInputStream(pathToFile),enc));

}

catch (UnsupportedEncodingException e)

{

return null;

}

catch (FileNotFoundException e)

{

return null;

}

String str = null;

ArrayList allStrings = new ArrayList();

try

{

while (!(str = br.readLine()).equals(null))

{

allStrings.add(str);

}

}

catch (IOException e)

{

return null;

}

catch (NullPointerException e)

{

}

try

{

br.close();

}

catch (IOException e)

{

return null;

}

String[] sar = (String [])allStrings.toArray(new String[allStrings.size()]);

return sar;

}

В процессах чтения и записи участвуют строковые переменные, содержащие путь к файлу и кодировку (для русских Windows это будет Cp1251), в которой производится считывание данных: pathToFile, enc.

В Java для преобразования потока байт (byte[]) в строку (String) и обратно, в классе String есть следующие возможности:

* Конструктор String(byte[] bytes, String enc) получает на вход поток байт с указанием их кодировки; если кодировку опустить, то она будет принята по умолчанию
* Метод getBytes(String enc) возвращает поток байт, записанных в указанной кодировке; кодировку также можно опустить.

**Порядок выполнения работы**

1. Изучить методические указания к работе
2. Разработать две программы для клиента и сервера в соответствии с вариантом задания. Во всех вариантах клиент отправляет серверу некоторый текст произвольной длины. Серверная часть выполняет его обработку и возвращает результат клиенту
3. Запустить программу сервера и программу клиента. Убедиться в наличии взаимодействия
4. Доработать клиентскую программу таким образом, чтобы исходные данные и результаты обработки считывались (сохранялись) в текстовом файле. Предусмотреть обработку исключительных ситуаций

 Таблица 1 – Варианты заданий

|  |  |
| --- | --- |
| Вариант | Задание |
| 1 | Вывод самого длинного слова |
| 2 | Замена всех квадратных скобок на круглые |
| 3 | Замена цепочки пробелов на один пробел |
| 4 | Вывод всех слов, представляющих собой числа в десятичной системе счисления |
| 5 | Вывод всех слов текста в алфавитном порядке |
| 6 | Вывод всех слов текста в обратном порядке |
| 7 | Замена всех слов, представляющих собой числа в шестнадцатиричной системе счисления, на их десятичный эквивалент |
| 8 | Исходный текст, в котором все числа десятичной системы счисления заключаются в двойные кавычки |

**Содержание отчета**

1. Постановка задачи
2. Описание метода решения задачи
3. Текст программы
4. Методика отладки и тестирования
5. Выводы

**Контрольные вопросы и задания**

1. Что означает понятие «клиент-серверное» приложение?
2. Какие методы обмена данными между клиентом и сервером вам известны?
3. Что такое сокета (socket)?
4. Какие методы имеются в классе Tsocket, для чего они предназначены?
5. Каким образом сервер узнает о том, что к нему обращается клиент?