Урок 6

Работа с сетью

Сокеты. Написание простого эхо-сервера и консольного клиента

[Основы работы в Сети](#)

[Написание эхо-сервера](#)

[Написание клиентской части](#)

[Дополнительная тема: Простейший HTTP сервер](#)

[Практическое задание](#)

[Дополнительные материалы](#)

Основы работы в Сети

В основу работы в сети, поддерживаемой в Java, положено понятие сокета, обозначающего конечную точку в Сети. Сокеты составляют основу современных способов работы в Сети, поскольку сокет позволяет отдельному компьютеру одновременно обслуживать много разных клиентов, предоставляя разные виды информации. Эта цель достигается благодаря применению порта — нумерованного сокета на отдельной машине.

Говорят, что серверный процесс «прослушивает» порт до тех пор, пока клиент не соединится с ним. Сервер в состоянии принять запросы от многих клиентов, подключаемых к порту с одним и тем же номером, хотя каждый сеанс связи индивидуален. Для управления соединениями со многими клиентами серверный процесс должен быть многопоточным или располагать какими-то другими средствами для мультиплексирования одновременного ввода-вывода.

Связь между сокетами устанавливается и поддерживается по определенному сетевому протоколу. Протокол Интернета (IP) является низкоуровневым маршрутизирующим сетевым протоколом, разбивающим данные на небольшие пакеты и посылающим их через Сеть по определенному адресу, что не гарантирует доставки всех этих пакетов по этому адресу. Протокол управления передачи (TCP) является сетевым протоколом более высокого уровня, обеспечивающим связывание, сортировку и повторную передачу пакетов, чтобы обеспечить надежную доставку данных. Еще одним сетевым протоколом является протокол пользовательских дейтаграмм (UDP). Этот сетевой протокол может быть использован непосредственно для поддержки быстрой, не требующей постоянного соединения и надежной транспортировки пакетов.

Как только соединение будет установлено, в действие вступает высокоуровневый протокол, тип которого зависит от используемого порта. Протокол TCP/IP резервирует первые 1 024 порта для отдельных протоколов. Например, порт 21 выделен для протокола FTP, порт 23 — для протокола Telnet, порт 25 — для электронной почты, порт 80 — для протокола HTTP и т.д. Каждый сетевой протокол определяет порядок взаимодействия клиента с портом.

Например, протокол HTTP[1] используется серверами и веб-браузерами для передачи гипертекста и графических изображений. Это довольно простой протокол для базового постраничного просмотра информации, предоставляемой веб-серверами. Рассмотрим принцип его действия. Когда клиент запрашивает файл у HTTP-сервера, это действие называется обращением. Оно состоит в том, чтобы отправить имя файла в специальном формате в предопределенный порт и затем прочитать содержимое этого файла. Сервер также сообщает код состояния, чтобы известить клиента, был ли запрос обслужен, а также причину, по которой он не может быть обслужен.

Главной составляющей Интернета является адрес, который есть у каждого компьютера в Сети. Изначально все адреса состояли из 32-разрядных значений, организованных по четыре 8-разрядных значения. Адрес такого типа определен в протоколе IPv4. Но в последнее время вступила в действие новая схема адресации, называемая IPv6 и предназначенная для поддержки намного большего адресного пространства. Правда, для сетевого программирования на Java обычно не приходится беспокоиться, какого типа адрес используется: IPv4 или IPv6, поскольку эта задача решается в Java автоматически.

Сокеты по протоколу TCP/IP служат для реализации надежных двунаправленных постоянных двухточечных потоковых соединений между хостами в Интернете. Сокет может служить для подключения системы ввода-вывода в Java к другим программам, которые могут находиться как на локальной машине, так и на любой другой в Интернете.

В Java поддерживаются две разновидности сокетов по протоколу TCP/IP: один — для серверов, другой — для клиентов. Класс ServerSocket предназначен для создания сервера, который будет

[1] Hyper Text Transfer Protocol - Протокол передачи гипертекста. Чаще всего под гипертекстом понимается HTML

обрабатывать клиентские подключения, тогда как класс Socket предназначен для обмена данными между сервером и клиентами по сетевому протоколу. При создании объекта типа Socket неявно устанавливается соединение клиента с сервером.

Для доступа к потокам ввода-вывода, связанным с классом Socket, можно воспользоваться методами getInputStream() и getOutputStream(). Каждый из этих методов может сгенерировать исключение типа IOException, если сокет оказался недействительным из-за потери соединения. Эти потоки ввода-вывода используются для передачи и приема данных.

Написание эхо-сервера

```
public class EchoServer {
    public static void main(String[] args) {
        Socket socket = null;
        try (ServerSocket serverSocket = new ServerSocket(8189)) {
            System.out.println("Сервер запущен, ожидаем подключения...");
            socket = serverSocket.accept();
            System.out.println("Клиент подключился");
            DataInputStream in = new DataInputStream(socket.getInputStream());
            DataOutputStream out = new DataOutputStream(socket.getOutputStream());
            while (true) {
                String str = in.readUTF();
                if (str.equals("/end")) {
                    break;
                }
                out.writeUTF("Эхо: " + str);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Для начала создается объект класса ServerSocket, представляющий собой сервер, который прослушивает порт 8189. Метод server.accept() переводит основной поток в режим ожидания, поэтому, пока никто не подключится, следующая строка кода выполнена не будет. Как только клиент подключился, информация о соединении с ним запишется в объект типа Socket. Для обмена сообщениями с клиентом необходимо создать обработчики входящего и исходящего потока, в данном случае это — DataInputStream и DataOutputStream.

Поскольку мы создаём эхо-сервер, обработка данных производится следующим образом: сервер считывает сообщение, переданное клиентом, добавляет к нему фразу «Эхо: » и отсылает обратно. Если клиент прислал сообщение «end», общение с ним прекращается, и сокет закрывается.

Блок finally предназначен для гарантированного закрытия всех сетевых соединений и освобождения ресурсов.

Написание клиентской части

Ниже представлен полный код клиентской части для эхо-сервера. Пока можете его просто бегло просмотреть, далее будем разбираться с каждым из блоков.

```
public class EchoClient extends JFrame {
    private final String SERVER_ADDR = "localhost";
    private final int SERVER_PORT = 8189;

    private JTextField msgInputField;
    private JTextArea chatArea;

    private Socket socket;
    private DataInputStream in;
    private DataOutputStream out;

    public EchoClient() {
        try {
            openConnection();
        } catch (IOException e) {
            e.printStackTrace();
        }
        prepareGUI();
    }

    public void openConnection() throws IOException {
        socket = new Socket(SERVER_ADDR, SERVER_PORT);
        in = new DataInputStream(socket.getInputStream());
        out = new DataOutputStream(socket.getOutputStream());
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    while (true) {
                        String strFromServer = in.readUTF();
                        if (strFromServer.equalsIgnoreCase("/end")) {
                            break;
                        }
                        chatArea.append(strFromServer);
                        chatArea.append("\n");
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }

    public void closeConnection() {
        try {
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            out.close();
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void sendMessage() {
        if (!msgInputField.getText().trim().isEmpty()) {
            try {
                out.writeUTF(msgInputField.getText());
                msgInputField.setText("");
                msgInputField.grabFocus();
            } catch (IOException e) {
                e.printStackTrace();
                JOptionPane.showMessageDialog(null, "Ошибка отправки сообщения");
            }
        }
    }

    public void prepareGUI() {
        // Параметры окна
        setBounds(600, 300, 500, 500);
        setTitle("Клиент");
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

        // Текстовое поле для вывода сообщений
        chatArea = new JTextArea();
        chatArea.setEditable(false);
        chatArea.setLineWrap(true);
        add(new JScrollPane(chatArea), BorderLayout.CENTER);

        // Нижняя панель с полем для ввода сообщений и кнопкой отправки
сообщений
        JPanel bottomPanel = new JPanel(new BorderLayout());
        JButton btnSendMsg = new JButton("Отправить");
        bottomPanel.add(btnSendMsg, BorderLayout.EAST);
        msgInputField = new JTextField();
        add(bottomPanel, BorderLayout.SOUTH);
        bottomPanel.add(msgInputField, BorderLayout.CENTER);
        btnSendMsg.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                sendMessage();
            }
        });
        msgInputField.addActionListener(new ActionListener() {
            @Override
```

```
            public void actionPerformed(ActionEvent e) {
                sendMessage();
            }
        });

        // Настраиваем действие на закрытие окна
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                super.windowClosing(e);
                try {
                    out.writeUTF("/end");
                    closeConnection();
                } catch (IOException exc) {
                    exc.printStackTrace();
                }
            }
        });

        setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                new EchoClient();
            }
        });
    }
}
```

Самое простое это метод prepareGUI() и main(), первый занимается подготовкой интерфейса Swing к работе, а второй собственно запускает наше клиентское приложение. Конструктор тоже содержит не особо много кода и отвечает как раз за выполнение инициализации интерфейса через prepareGUI() и подключение к серверу с помощью метода openConnection().

Давайте же посмотрим что за поля, связанные с сетевой частью, есть в нашем клиенте.

```
private final String SERVER_ADDR = "localhost";
private final int SERVER_PORT = 8189;
private Socket socket;
private DataInputStream in;
private DataOutputStream out;
```

Константа SERVER_ADDR задаёт адрес эхо-сервера, к которому будет подключаться клиент, SERVER_PORT — номер порта. Для открытия соединения с сервером и обмена сообщениями используются объекты классов Socket, DataInputStream и DataOutputStream, по аналогии с серверной частью.

Далее смотрим на логику подключения к серверу.

```
public void openConnection() throws IOException {
    socket = new Socket(SERVER_ADDR, SERVER_PORT);
    in = new DataInputStream(socket.getInputStream());
    out = new DataOutputStream(socket.getOutputStream());
    new Thread(new Runnable() {
        @Override
        public void run() {
           try {
                while (true) {
                    String strFromServer = in.readUTF();
                    if (strFromServer.equalsIgnoreCase("/end")) {
                        break;
                    }
                    chatArea.append(strFromServer);
                    chatArea.append("\n");
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

Первым делом открываем сокет с указанием ip-адреса и порта сервера. Далее с помощью методов socket.getInputStream() и getOutputStream() запрашиваем у сокета доступ к исходящему(в сторону сервера), и входящему(направленному к нашему клиенту) потокам, но так как имея просто потоки мы ничего сделать не сможем(не сможем передавать данные), заворачиваем их в обработчики DataInputStream in и DataOutputStream out.

Сетевое соединение и обработчики проинициализированы, теперь запускаем отдельный поток, чтобы слушать что же нам пришлет сервер. Если попробуем это сделать в текущем потоке, то ничего не выйдет, так как зависнем в цикле while(), и построение нашего объекта (клиентского окна) так и не завершится. В этом отдельном потоке запускается цикл while() и начинает слушать входящие сообщения, операция readUTF() блокирующая, и поток будет периодически переходить в режим ожидания, пока сервер что-нибудь не пришлет.

Как только сообщение от сервера пришло, мы записываем его в строку strFromServer и выводим в основное текстовое поле нашего чата charArea. (название chatArea дано с учетом того, что в будущем мы получим чат).

С открытием соединения мы разобрались. Для закрытия же используется довольно простой метод closeConnection().

Ну и наконец метод для отправки сообщения в сторону сервера.

```
public void sendMessage() {
    if (!msgInputField.getText().trim().isEmpty()) {
        try {
            out.writeUTF(msgInputField.getText());
            msgInputField.setText("");
            msgInputField.grabFocus();
        } catch (IOException e) {
            e.printStackTrace();
```

```
            JOptionPane.showMessageDialog(null, "Ошибка отправки сообщения");
        }
    }
}
```

Клиент берет содержимое текстового поля для отправки и с помощью метода writeUTF() отправляет его серверу, после чего очищает текстовое поле и переводит на него фокус. Если вдруг не удалось отправить сообщение, то будет показано всплывающее окно с ошибкой.

Дополнительная тема: Простейший HTTP сервер[2]

Умения работать с классом ServerSocket вполне достаточно для того чтобы написать простейший веб сервер с которым сможет взаимодействовать любой веб браузер.

```
public class HttpServer {

   public static void main(String[] args) {
       try (ServerSocket serverSocket = new ServerSocket(8080)) {
           System.out.println("Server started!");

           while (true) {
               // ожидаем подключения
               Socket socket = serverSocket.accept();
               System.out.println("Client connected!");

               // для подключившегося клиента открываем потоки
               // чтения и записи
               try (BufferedReader input = new BufferedReader(new
InputStreamReader(socket.getInputStream(), StandardCharsets.UTF_8));
                    PrintWriter output = new PrintWriter(socket.getOutputStream())) {

                   // ждем первой строки запроса
                   while (!input.ready()) ;

                   // считываем и печатаем все что было отправлено клиентом
                   System.out.println();
                   while (input.ready()) {
                       System.out.println(input.readLine());
                   }

                   // отправляем ответ
                   output.println("HTTP/1.1 200 OK");
                   output.println("Content-Type: text/html; charset=utf-8");
                   output.println();
                   output.println("<p>Привет всем!</p>");
                   output.flush();

                   // по окончанию выполнения блока try-with-resources потоки,
                   // а вместе с ними и соединение будут закрыты
                   System.out.println("Client disconnected!");
               }
           }
       } catch (IOException ex) {
```

[2] Материал (Простейший HTTP-сервер) добавлен автором (Алексей Ушаровский), ссылка на оригинал: https://habr.com/ru/post/441150/

```
            ex.printStackTrace();
        }
    }
}
```

Если вы запустите данный код и введёте в окне любого браузера http://localhost:8080, то увидите строку “Привет всем!” которая была отправлена в ответ веб сервером. При этом сервер напечатает в консоли HTTP запрос, который был ему отправлен браузером. Выглядеть он будет примерно так

```
GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apn
g,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language:
ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7,he;q=0.6,de;q=0.5,cs;q=0.4
Cookie: _ga=GA1.1.1849608036.1549463927;
portainer.pagination_containers=100; _gid=GA1.1.80775985.1550669456;
If-Modified-Since: Sat, 05 Jan 2019 12:10:16 GMT
```

Следует обратить особое внимание на то, что взаимодействие браузера с веб сервером по протоколу HTTP сводится прежде всего к пересылке текста в определенном формате. Стоит обратить внимание на первую строку которая начинается со слова GET. Это так называемый HTTP метод. Фактически - команда серверу передать в качестве ответа документ, который содержится по указанному адресу. Адрес находится через пробел после метода. Если вы напишете в окне браузера http://localhost:8080/something, то вместо косой черты будет /something. Через пробел после адреса идет версия протокола. В следующих строках идут т.н. HTTP заголовки, которые нужны для передачи серверу различных параметров клиента. В этом курсе мы не будем на них останавливаться подробно.

В коде вы можете видеть ответ на данный HTTP запрос. Он имеет похожую структуру. Первая строка содержит версию протокола HTTP и код ответа 200, который означает, что запрошенный документ был найден и будет передан в этом ответном запросе. Далее, как и в запросе следуют заголовки (только один, в котором передается кодировка). После заголовков следует пустая строка, которая отделяет заголовок HTTP запроса от его содержания. В данном случае это строка “Привет всем!!!” которую вы увидели в своем браузере.

Практическое задание

1. Написать консольный вариант клиент\серверного приложения, в котором пользователь может писать сообщения как на клиентской стороне, так и на серверной. Т.е. если на клиентской стороне написать «Привет», нажать Enter, то сообщение должно передаться на сервер и там отпечататься в консоли. Если сделать то же самое на серверной стороне, сообщение,

соответственно, передаётся клиенту и печатается у него в консоли. Есть одна особенность, которую нужно учитывать: клиент или сервер может написать несколько сообщений подряд. Такую ситуацию необходимо корректно обработать.

Разобраться с кодом с занятия — он является фундаментом проекта-чата.

ВАЖНО! *Сервер общается только с одним клиентом, т.е. не нужно запускать цикл, который будет ожидать второго/третьего/n-го клиента.*

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл. Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. — М.: Вильямс, 2014. — 864 с.
2. Брюс Эккель. Философия Java // 4-е изд.: Пер. с англ. — СПб.: Питер, 2016. — 1 168 с.
3. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.
4. Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 720 с.