**Пакет java.net, ServerSocket**

На сегодняшний день использование клиентов служб мгновенного обмена сообщениями (instant messanger) стало незаменимым средством для всех пользователей Интернета. Существует множество клиентов (Skype, WhatsApp, Viber, ICQ и т. д.), о которых каждый слышал и которые мы ежедневно используем. Все они работают по определенным правилам, т.е. реализуют определенные протоколы взаимодействия.

Наиболее распространенный протокол HTTP (Hyper Text Transfert Protocol) определяет взаимодействие двух программ, клиента и сервера, которые могут быть запущены на разных и удаленных друг от друга машинах. Клиентом является приложение, которое пользуется каким-то сервисом, предоставляемым сервером, обычно размещенном на удаленном компьютере. Клиент должен подключиться к удаленному серверу, который постоянно находится в режиме ожидания соединения. После этого они могут обмениваться информацией.

HTTP использует протокол TCP/IP. В статье рассмотриваются возможности, предоставляемые Java для работы с этим протоколом.

**Распределение протоколов по уровням модели TCP/IP**

|  |  |  |
| --- | --- | --- |
| **№ п.п.** | **Название** | **Протоколы** |
| 5 | Прикладной | HTTP, RTP, FTP, DNS |
| 4 | Транспортный | TCP, UDP, SCTP, DCCP |
| 3 | Сетевой | IP. Вспомогательные протоколы, вроде ICMP и IGMP, работают поверх IP. |
| 2 | Канальный | Ethernet, IEEE 802.11 Wireless Ethernet, SLIP, Token Ring, ATM и MPLS |
| 1 | Физический | Физическая среда и принципы кодирования информации, T1, E1 |

Протокол HTTP располагается на прикладном уровне и использует для собственной реализации протоколы более низких уровней. Основой HTTP является протокол транспортного уровня TCP.

Согласно протоколу IP (Internet Packet), каждый узел (компьютер, switch и т.п.) в сети имеет свой IP-адрес. На данный момент интернет работает по протоколу IPv4, где IP адрес записывается 4 числами от 0 до 255 - например, 127.0.0.1. Существует и другой способ идентификации компьютеров в сети через доменное имя, которое более удобное и нагляднее идентифицирует компьютер, чем простой набор чисел (например, java-oline.ru). В Интернете существуют специальные сервера DNS (Domain Name System), которые осуществляют преобразование доменного имени в IP-адрес и наоборот.

TCP протокол базируется на IP для доставки пакетов, но добавляет два важных свойства :

* установление соединения между приложениями;
* использование портов, а не просто узлов.

Таким образом:

- для идентификации компьютера (host'a) - в сети используется IP-адрес;

- для идентификации приложения (*работающего на компьютере*) - TCP добавляет понятие порта.

Порт - это целое число от 1 до 65535 указывающее, какому приложению предназначается пакет.

Java для работы в сети имеет специальный пакет *java.net*, содержащий класс **Socket**, что в переводе означает «гнездо». Ключевыми классами для реализации взаимодействия программ по протоколу TCP являются :

* java.net.ServerSocket - класс реализует серверный сокет, который ожидает запросы, приходящие от клиентов по сети, и может отправлять ответ.
* java.net.Socket - класс реализует клиентский сокет.

**Серверный сокет ServerSocket**

Для создания серверного сокета **ServerSocket** можно использовать один из следующих конструкторов :

public ServerSocket() throws IOException;

public ServerSocket(int port) throws IOException;

public ServerSocket(int port, int backlog) throws IOException;

public ServerSocket(int port, int backlog, InetAddress bindAddr)

throws IOException;

Первым параметров в конструктор необходимо передать порт *port*, который будет привязан к серверному сокету. Если порт занят или запрещён к использованию политикой безопасности компьютера, то вызывается исключение IOException. Если значение передаваемого порта равно 0, то система сама выделит номер свободного порта. Значение полученного порта можно узнать через вызов функции *getLocalPort()*. Несвязанный серверный сокет ServerSocket() необходимо «связывать» с IP-адресом и портом (см. ниже метод *bind*).

Параметр *backlog* устанавливает максимальное количество клиентских подключений. Если количество подключений достигло предела, то следующему клиенту в подключении будет отказано. *По умолчанию, по-моему, установлено 50 подключений*.

Для работы с IP-адресами в библиотеке Java имеется класс **java.net.InetAddress**, который используется в третьем конструкторе *ServerSocket*. С помощью *InetAddress* можно определить адрес IP локального узла, а также адреса удаленного узла, заданного доменным именем. Наиболее распространенные методы класса *InetAddress* :

public static InetAddress getLocalHost();

public static InetAddress getByName(String host);

public static InetAddress[] getAllByName(String host);

public byte[] getAddress();

public String toString();

public String getHostName();

public boolean equals(Object obj);

При разработке сетевых приложений на начальном этапе, как правило, используют один компьютер (host). Для этого создатели протокола IP определили специальный адрес, называемый localhost - это IP-адрес "локальной заглушки (local loopback)" для работы приложений без использования сети. Общий порядок получения этого адреса в Java следующий :

InetAddress address = InetAddress.getByName(null);

address = InetAddress.getByName("localhost");

Если методу *getByName()* передать значение null, то по умолчанию будет использоваться *localhost*. Cодержимым InetAddress нельзя манипулировать. Для создания InetArddress можно использовать один из перегруженных статических методов класса getByName(), getAllByName() или getLocalHost().

**Методы серверного сокета**

В таблице представлены наиболее часто используемые методы серверного сокета *ServerSocket*.

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| Socket accept() | Ожидание подключения клиента |
| void bind(SocketAddress endpoint) | Связывание ServerSocket c определенным адресом (IP-адрес и порт) |
| void close() | Закрытие сокета |
| ServerSocketChannel getChannel() | Получение объекта ServerSocketChannel, связанного с сокетом |
| InetAddress getInetAddress() | Получение локального адреса сокета сервера |
| int getLocalPort() | Получение номера порта, который серверный сокет слушает |
| SocketAddress getLocalSocketAddress() | Получение адреса серверного сокета в виде объекта SocketAddress |
| int getReceiveBufferSize() | Получение размера буфера серверного сокета |
| boolean isClosed() | Проверка, закрыт ли серверный сокет |
| void setReceiveBufferSize(int size) | Определение размера буфера серверного сокета |

После создания в приложении серверного сокета *ServerSocket* необходимо вызвать функцию **accept()**, которая переводит приложение в режим ожидания подключения клиента. Дальнейший код не выполняется, пока клиент не подключится. Как только клиент подключается функция возвращает объект класса **java.net.Socket**, который следует использовать для взаимодействия сервера с клиентом.

**Клиентский сокет Socket**

Коиентский сокет **Socket** можно создать с использованием одного из следующих конструкторов :

public Socket();

public Socket(String host, int port);

public Socket(InetAddress address, int port);

В строковой константе host можно указать как IP адрес сервера, так и его DNS имя. При этом программа автоматически выберет свободный порт на локальном компьютере и свяжет его с сокетом. При этом могут быть вызваны одно из двух видов исключений, связанного с неизвестным адресом хоста (в сети компьютер не будет найден) или отсутствием связи с этим сокетом.

Класс *Socket* имеет один интересный метод *setSoTimeout* :

public void setSoTimeout(int timeout) throws SocketException;

Метод *setSoTimeout* устанавливает время ожидания (timeout) для работы с сокетом. Если в течение этого времени никаких действий с сокетом не произведено (получение и отправка данных), то он самоликвидируется. Время задаётся в секундах, при установке timeout равным 0 сокет становится "вечным".

**Примеры использования SocketServer и Socket**

Ниже представлены примеры двух приложений : сервер и клиент. Серверное приложение стартует первым и ждет подключений клиентов. После этого стартует клиентское приложение и подключается к серверу. Из клиентского приложения можно отправлять сообщения, на которые сервер должен ответить.

**Листинг сервера**

Сервер создает сокет *ServerSocket*, открывает порт и ждет подключений клиента. После подключения клиента сервер формирует отдельный [поток Thread](http://java-online.ru/java-thread.xhtml), в который передает порядковый номер клиента и сокет *Socket* для обмена сообщениями с клиентом. Сам сервер продолжает ожидать подключение следующего клиента.

Здесь следует отметить немаловажную особенность серверного приложения : оно может обслуживать сразу несколько клиентов одновременно. Теоретически, количество одновременных подключений неограниченно, но практически всё упирается в мощность компьютеров. Эта проблема конечной мощности компьютеров используется в DOS атаках на серверы: их просто закидывают таким количеством подключений, что компьютеры не справляются с нагрузкой и «падают».

В этом простом примере демонстрируется использование серверного сокета *ServerSocket* для обслуживания нескольких одновременных подключений: сокет каждого нового подключения отправляется на обработку в отдельный вычислительный поток.

package com.common;

import java.io.\*;

import java.net.\*;

public class Server extends Thread

{

// открываемый порт сервера

private static final int port = 6666;

private String TEMPL\_MSG = "The client '%d' sent me message : \n\t";

private String TEMPL\_CONN = "The client '%d' closed the connection";

private Socket socket;

private int num;

public Server() {}

public void setSocket(int num, Socket socket)

{

// Определение значений

this.num = num;

this.socket = socket;

// Установка daemon-потока

setDaemon(true);

/\* Определение стандартного приоритета главного потока

\* int java.lang.Thread.NORM\_PRIORITY = 5-the default

/\* priority that is assigned to a thread.

setPriority(NORM\_PRIORITY);

// Старт потока

start();

}

public void run()

{

try {

// Определяем входной и выходной потоки сокета

// для обмена данными с клиентом

InputStream sin = socket.getInputStream();

OutputStream sout = socket.getOutputStream();

DataInputStream dis = new DataInputStream (sin );

DataOutputStream dos = new DataOutputStream(sout);

String line = null;

while(true) {

// Ожидание сообщения от клиента

line = dis.readUTF();

System.out.println(String.format(TEMPL\_MSG, num) + line);

System.out.println("I'm sending it back...");

// Отсылаем клиенту обратно эту самую

// строку текста

dos.writeUTF("Server receive text : " + line);

// Завершаем передачу данных

dos.flush();

System.out.println();

if (line.equalsIgnoreCase("quit")) {

// завершаем соединение

socket.close();

System.out.println(String.format(TEMPL\_CONN, num));

break;

}

}

} catch(Exception e) {

System.out.println("Exception : " + e);

}

}

//---------------------------------------------------------

public static void main(String[] ar)

{

ServerSocket srvSocket = null;

try {

try {

int i = 0; // Счётчик подключений

// Подключение сокета к localhost

InetAddress ia;

ia = InetAddress.getByName("localhost");

srvSocket = new ServerSocket(port, 0, ia);

System.out.println("Server started\n\n");

while(true) {

// ожидание подключения

Socket socket = srvSocket.accept();

System.err.println("Client accepted");

// Стартуем обработку клиента в отдельном потоке

new Server().setSocket(i++, socket);

}

} catch(Exception e) {

System.out.println("Exception : " + e);

}

} finally {

try {

if (srvSocket != null)

srvSocket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

System.exit(0);

}

}

**Листинг клиента**

Клиентский пример использования класса java.net.Socket для подключения к серверу и обмена с ним сообщениями. Если серверу отправить сообщение «quit», то цикл обмена сообщениями будет прекращен, сервер закроет подключение (свой сокет) и клиентский сокет также следует закрыть.

package com.common;

import java.net.\*;

import java.io.\*;

public class Client

{

private static final int serverPort = 6666;

private static final String localhost = "127.0.0.1";

public static void main(String[] ar)

{

Socket socket = null;

try{

try {

System.out.println("Welcome to Client side\n" +

"Connecting to the server\n\t" +

"(IP address " + localhost +

", port " + serverPort + ")");

InetAddress ipAddress;

ipAddress = InetAddress.getByName(localhost);

socket = new Socket(ipAddress, serverPort);

System.out.println("The connection is established.");

System.out.println(

"\tLocalPort = " +

socket.getLocalPort() +

"\n\tInetAddress.HostAddress = " +

socket.getInetAddress().getHostAddress() +

"\n\tReceiveBufferSize (SO\_RCVBUF) = " +

socket.getReceiveBufferSize());

// Получаем входной и выходной потоки

// сокета для обмена сообщениями с сервером

InputStream sin = socket.getInputStream();

OutputStream sout = socket.getOutputStream();

DataInputStream in ;

DataOutputStream out;

in = new DataInputStream (sin );

out = new DataOutputStream(sout);

// Создаем поток для чтения с клавиатуры.

InputStreamReader isr;

isr = new InputStreamReader(System.in);

BufferedReader keyboard;

keyboard = new BufferedReader(isr);

String line = null;

System.out.println(

"Type in something and press enter");

System.out.println();

while (true) {

// Пользователь должен ввести строку

// и нажать Enter

line = keyboard.readLine();

// Отсылаем строку серверу

out.writeUTF(line);

// Завершаем поток

out.flush();

// Ждем ответа от сервера

line = in.readUTF();

if (line.endsWith("quit"))

break;

else {

System.out.println(

"The server sent me this line :\n\t"

+ line);

}

}

} catch (Exception e) {

e.printStackTrace();

}

} finally {

try {

if (socket != null)

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

**Тестирование приложений**

Результаты тестирования серверного и клиентского приложений будут выводиться в соответствующие «консоли». После подключения клиента к серверу он может отправлять сообщения. Сервер возвращает изменное сообщение клиента. Одновременно к серверу может подключиться несколько клиентов.

**Сообщения сервера**

Сервер после старта выводит соответствующее сообщение и ждет подключений клиентов. После подкючения клиента и получения от него сообщения сервер возвращает клиенту измененную строку. При получении сообщения «quit» сеанс завершается.

Server started

Client accepted

The client '0' sent me message :

Привет

I'm sending it back...

The client '0' sent me message :

quit

I'm sending it back...

The client '0' closed the connection

**Сообщения клиента**

После подключения клиента к серверу выводится информация о сокете. Далее серверу отправляется сообщение «Привет». Сервер информирует клиента, что получил это сообщение. При отправлении текста «quit» сеанс обмена сообщениями с сервером завершается.

Welcome to Client side

Connecting to the server

(IP address 127.0.0.1, port 6666)

The connection is established.

LocalPort = 57975

InetAddress.HostAddress = 127.0.0.1

ReceiveBufferSize (SO\_RCVBUF) = 65536

Type in something and press enter

Привет

The server sent me this line :

Server receive text : Привет

quit

**Скачать примеры**

Рассмотренные на странице примеры использования ServerSocket, Socket для создания сетвых приложений в виде проекта Eclipse можно скачать [здесь](http://java-online.ru/sources/java-socket.zip) (24.8Кб).