



# Сетевое программирование и Веб-приложения

---

Обзор основных конструкций

# Основные понятия сетевого программирования

---

## Клиент -Сервер

Протокол – правила по которым формируют запросы(response) и ответы (request) во время сессии.

Порт – целое положительное число которое указывает клиент при обращении к серверу.

Порт эл.почты=25, порт ftp=21, ст.порт веб сервера =80(8080,8001).

Ст. номера:[0,1023] свободные порты[1024,65535]

Стек из 4 протоколов: TCP/IP:

1 уровень прикладной (application) протокол HTTP(80), SMTP(25), Telnet(23), FTP(21), POP3(100)

2 уровень –транспортный протокол (+ номера портов отправителя и получателя, контрольная сумма, длина сообщения) – TCP (transmission control protocol)  TCP пакет & UDP (user datagram protocol)  datagram

UDP- протокол состоит из дейтаграмм <1кб, и они могут прийти различными путями.

TCP  устанавливаем соединение  пересылаем TCP пакеты  пакеты проверяются (повторяются при неудаче)  поток байтов.

3 уровень сетевого протокола IP(internet protocol) (+ IP-адреса(доменные имена) и др.)

4 уровень канального протокола ENET, SLIP, PPP

В пакете java.net существует класс InetAddress экземпляр этого класса создается статическим методом

getByName(String host), host –доменное имя или IP-адрес.

## Работа в WWW (Пакет java.net)

---

**WWW основана на прикладном (application) протоколе HTTP.**

**Используется расширенная адресация URL (uniform resource locator)**

**Схемы адресации:**

**protocol://authority@host:port/path/file#ref**

**protocol://authority@host:port/path/file/extra\_path?info**

**Authority- имя:пароль –необязательно.**

**Класс URL € java.net**

**Объекты этого класса создаются 6 конструкторами типа URL(String url);**

**openConnection() –определяет связь с URL и возвращает объект класса URLConnection**

**openStream- открывает входной поток в виде объекта InputStream.**

# Получение веб-страницы

---

```
package simpleurl_app;
import java.net.*;
import java.io.*;
import java.net.MalformedURLException;
class simpleURL{
    public static void main(String[] args)
    {try{
        URL url1 =new URL("file:///nagent_log.txt");
        BufferedReader br=new BufferedReader(new
InputStreamReader(url1.openStream()));
        String s;
        while((s=br.readLine())!=null)
            System.out.print(s);
br.close();
    }catch(MalformedURLException e){e.printStackTrace();}
    catch(IOException ioe){System.err.println(ioe);}
    }
}
```

Сопутствующая информация о типе, архивных файлах, изображении, длине файла хранится в объектах классов `URLConnection` или `HttpURLConnection`.

# Пример

---

Программа при помощи `LineNumberReader` считывает первую страницу сайта `http://www.ru` и выводит ее на консоль.

```
import java.io.*;
import java.net.*;
public class Net {
public static void main(String args[]) {try {
URL url = new URL("http://www.ru");
LineNumberReader r = new LineNumberReader(new
InputStreamReader(url.openStream()));
String s = r.readLine();
while (s!=null) {
System.out.println(s);
s = r.readLine();}
System.out.println(r.getLineNumber());
r.close();} catch (MalformedURLException e) {
e.printStackTrace();
} catch (IOException e) {e.printStackTrace();}
```

// Ошибка `MalformedURLException` появляется в случае, если строка с URL содержит ошибки.

# Класс URLConnection

---

Методы- `getInputStream()` (именно с его помощью работает можно использовать для передачи данных на сервер, если он поддерживает такую операцию (многие публичные web-сервера закрыты для таких действий)).

Класс `URLConnection` является абстрактным. Виртуальная машина предоставляет реализации этого класса для каждого протокола, например, в том же пакете `java.net` определен класс `URLConnection`

# Получение свойств объектов URLConnection

---

- **setDoOutput(boolean out)** если out- true –передача от клиента на хост (по умолчанию false)
- **setDoInput(boolean in)** если in- true –передача от хоста клиенту (по умолчанию true)
- **setUseCaches (boolean cache)** если cache- false –передача без кэширования.
- **setRequestProperty(String name, string value)** –добавляет параметр name со значением value к заголовку посылаемого сообщения.

После задания параметров нужно установить соединение методом **connect()**.

Многие методы **getXxx()**, получающие значения с хоста, устанавливают соединения автоматически (без **connect()**).

Веб-сервер возвращает информацию. Запрошенную клиентом вместе с заголовком, сведения из которого можно получить методами типа **getXxx()**.

```
getContentType();  
getContentLength();  
getContent();  
getContentEncoding();
```

Потоки ввода-вывода для данного соединения создаются методами:

```
getInputStream(); getOutputStream();
```

# CGI программирование

Строка прересылаемая серверу обрабатывается программой расположенной в серверной директории cgi-bin

```
import java.net.*;
import java.io.*;
import java.net.URLConnection;
class SimpleURL{
public static void main(String[] args){
String req ="this text is posting to URL";
try{
//указываем URL CGI программы
URL url= new URL("http://localhost/cgi-bin/cgi.exe");
// создаем объект
URLConnection uc=url.openConnection();
//собираемся отправлять
uc.setDoOutput(true);
// и получать сообщения
uc.setDoInput(true);
// без кэширования
uc.setUseCaches(false);
// задаем тип
uc.setRequestProperty("content-type", "application/octet-stream");
// длину сообщения
uc.setRequestProperty("content-length","+req.length()");
//устанавливаем соединение
uc.connect();
// открываем выходной поток
DataOutputStream dos=new DataOutputStream(uc.getOutputStream());
//выводим сообщение на адрес URL
dos.writeBytes(req);
//закрываем выходной поток
dos.close();
```



# CGI

---

```
//открываем входной поток для ответа сервера
BufferedReader br = new BufferedReader (new
InputStreamReader(uc.getInputStream()));
String res=null;
while((res=br.readLine())!=null)
System.out.println(res);
br.close();
}catch (MalformedURLException me){
System.err.println(me);

}catch (IOException ioe){
System.err.println("Input Error:" +ioe);
}
}}
```



Упр.

1. Написать программу получающую заголовочную информацию и страницы с сайтов интернета
2. Написать CGI программу и программу для запуска CGI программы и получения результатов ее работы

# Пример CGI

---

```
<html>
<head>
  <title>GET & POST</title>
</head>
<body style="text-align: center;">
  <div style="margin: 5px; width: 200px; padding: 5px; border: 1px solid #E00000; background-color: #eee;
float: left;">
    <h2 style="font-family: Tahoma; font-size: large;">GET</h2>
    <form method="get" action="http://127.0.0.1/cgi_get.exe">
      <p>Введите два сомножителя:</p>
      <input name="fdig" size="7" />&nbsp;  
      <input name="sdig" size="7" />
      <br /><br />
      <input type="submit" value="Перемножить!" />
      <br /><br />
    </form>
  </div>
  <div style="margin: 5px; width: 400px; padding: 5px; border: 1px solid #E00000; background-color: #eee;
float: left;">
    <h2 style="font-family: Tahoma; font-size: large;">POST</h2>
    <form method="POST" action="http://127.0.0.1/cgi_post.exe">
      <p>Введите текст (не более 200 символов):</p>
      <input name="data" size="60" />
      <br /><br />
      <input type="submit" value="Отослать!" />
      <br /><br />
    </form>
  </div>
</body>
</html>
```

# Класс InetAddress

---

Пакет `java.net` также предоставляет доступ к протоколам более низкого уровня - TCP и UDP. Для этого сначала надо ознакомиться с классом `InetAddress`, который является интернет-адресом, или IP. Экземпляры этого класса создаются не с помощью конструкторов, а с помощью статических методов:

`InetAddress getLocalHost()`

`InetAddress getByName(String name)`

`InetAddress[] getAllByName(String name)`

Первый метод возвращает IP-адрес машины, на которой выполняется Java-программа.

Второй метод возвращает адрес сервера, чье имя передается в качестве параметра. Это может быть как DNS-имя, так и числовой IP, записанный в виде текста, например, "67.11.12.101". Наконец третий метод определяет все IP-адреса указанного сервера.

▪

# InetAddress

---

Следующая программа использует **InetAddress.getByName( )** для определения Вашего IP адреса. Чтобы использовать его, Вы должны знать имя своего компьютера. В Windows 95/98, зайдите в "Settings", "Control Panel", "Network", а затем выберите страничку "Identification". "Computer name" это имя, которое необходимо задать в командной строке.

```
import java.net.*;
public class WhoAmI {
    public static void main(String[] args)
        throws Exception {
        if(args.length != 1) {
            System.err.println(
                "Usage: WhoAmI MachineName");
            System.exit(1);
        }
        InetAddress a =
            InetAddress.getByName(args[0]);
        System.out.println(a);
    }
}
```

# localhost

---

Существует специальный адрес, называемый **localhost**, “локальная петля”, который является IP адресом для тестирования без наличия сети. Обычный способ получения этого адреса в Java это:

```
InetAddress addr = InetAddress.getBy_name(null);
```

Если Вы ставите параметр **null** в метод **getByName( )**, то, по умолчанию используется **localhost**.

**InetAddress** это то, что Вы используете для ссылки на конкретную машину, и Вы должны предоставлять это, перед тем как продолжить дальнейшие действия. Вы не можете манипулировать содержанием **InetAddress**

Единственный способ создать **InetAddress** - это использовать один из перегруженных статических методов **getByName( )** (который Вы обычно используете), **getAllByName( )**, либо **getLocalHost( )**.

Вы можете создать адрес локальной петли, установкой строкового параметра **localhost**:

```
InetAddress.getBy_name("localhost");
```

(присваивание “localhost” конфигурируется в таблице “hosts” на Вашей машине), либо с помощью четырехточечной формы для именованного зарезервированного IP адреса для петли:

```
InetAddress.getBy_name("127.0.0.1");
```

# Классы Socket и ServerSocket

---

Для работы с TCP-протоколом используются классы Socket и ServerSocket.

Первым создается ServerSocket - сокет на стороне сервера. Его простейший конструктор имеет только один параметр - номер порта, на котором будут приниматься входящие запросы.

После создания вызывается метод accept(), который приостанавливает выполнение программы и ожидает, пока какой-нибудь клиент не инициализирует соединение

# Socket

---

Работа сервера возобновляется, а метод возвращает экземпляр класса Socket для взаимодействия с клиентом:

```
try {  
    ServerSocket ss = new ServerSocket(3456);  
    Socket client=ss.accept();  
    // Метод не возвращает управление, пока не подключится клиент  
} catch (IOException e) {e.printStackTrace();}
```

Клиент для подключения к серверу также используется класс Socket. Его простейший конструктор принимает два параметра - адрес сервера (в виде строки или экземпляра

InetAddress) и номер порта. Если сервер принял запрос, то сокет конструируется успешно, и далее можно воспользоваться методами getInputStream() или getOutputStream().

```
try {  
    Socket s = new Socket("localhost", 3456);  
    InputStream is = s.getInputStream();  
    is.read();} catch (UnknownHostException e) {  
    e.printStackTrace();} catch (IOException e) {  
    e.printStackTrace();  
}
```

# Пример

---

На стороне сервера класс `Socket` используется точно таким же образом - через методы `getInputStream()` и `getOutputStream()`.

Приведем более полный пример:

```
import java.io.*;
import java.net.*;
public class Server {
public static void main(String args[]) {
try {
ServerSocket ss = new ServerSocket(3456);
System.out.println("Waiting...");
Socket client=ss.accept();
System.out.println("Connected");
client.getOutputStream().write(10);
client.close();
ss.close();
} catch (IOException e) {
e.printStackTrace();}}}
```

Сервер по запросу клиента отправляет число 10 и завершает работу. При завершении вызываются методы `close()` для открытых сокетов



# Класс клиента:

---

```
import java.io.*;
import java.net.*;
public class Client {
public static void main(String args[]) {
try {
Socket s = new Socket("localhost", 3456);
InputStream is = s.getInputStream();
System.out.println("Read: "+is.read());
s.close();
} catch (UnknownHostException e) {
e.printStackTrace();
} catch (IOException e) {
e.printStackTrace();}}}}
```

После запуска сервера, а затем клиента, можно увидеть результат - полученное число 10, после чего обе программы закроются.

Классы `ServerSocket` `Socket` имеют расширенный конструктор для указания как локального адреса, с которого будет устанавливаться соединение, так и локального порта (иначе операционная система выделяет произвольный свободный порт).

# Работа с прокси - сервером

---

`Socket(proxy proxy);`

Этот конструктор использует ссылку на объект абстрактного класса `Proxy`.

Объект создается конструктором

`Proxy(Proxy.Type type, SocketAddress address);`

`Proxy.Type` :

- `DIRECT` – соединение без прокси-сервера;
- `HTTP` -соединение с прокси-сервером по протоколу `HTTP`, `FTP`;
- `SOCKS` -соединение с прокси-сервером по протоколу `SOCKS4` или `SOCKS5`.

`SocketAddress` -абстрактный класс, используют его расширение `InetSocketAddress`

`Socket sock = new Socket(new Proxy(Proxy.Type.SOCKS, new(InetSocketAddress("socks.domain.com", 1080))));`

# Работа с несколькими клиентами

---

Можно воспользоваться методом `setSoTimeout(int timeout)` класса `ServerSocket`, чтобы указать время в миллисекундах, на протяжении которого нужно ожидать подключение клиента. Это позволяет не "зависать" серверу, если никто не пытается начать с ним работать. Таймаут задается в миллисекундах, нулевое. После установления соединения с клиентом сервер выходит из

метода `accept()`, то есть перестает быть готов принимать новые запросы.

Однако как правило желательно, чтобы сервер мог работать с несколькими клиентами одновременно.

Для этого необходимо при подключении очередного пользователя создавать новый поток исполнения, который будет обслуживать его, а основной поток снова войдет в метод `accept()`.

**Пример такого решения:**

```
import java.io.*;
import java.net.*;
public class NetServer {
public static final int PORT = 2500;
private static final int TIME_SEND_SLEEP = 100;
private static final int COUNT_TO_SEND = 10;
private ServerSocket servSocket;
public static void main(String[] args) {NetServer server = new NetServer();
server.go();}public NetServer() {
```

# Пример Работа с несколькими клиентами

---

```
try{servSocket = new ServerSocket(PORT);
}catch(IOException e){
System.err.println("Unable to open Server Socket : " +
e.toString());}}
public void go() {
// Класс-поток для работы с подключившимся клиентом
class Listener implements Runnable{Socket socket;
public Listener(Socket aSocket){
socket = aSocket;}
public void run(){
try{
System.out.println("Listener started");
int count = 0;
OutputStream out = socket.getOutputStream();
OutputStreamWriter writer = new OutputStreamWriter(out);
PrintWriter pWriter = new PrintWriter(writer);
```

# Пример Работа с несколькими клиентами

---

```
while(count<COUNT_TO_SEND){count++;
pWriter.print(((count>1)?",":"")+ "Say" + count);
sleeps(TIME_SEND_SLEEP);}
pWriter.close();}catch(IOException e){
System.err.println("Exception : " + e.toString());}}}}
// Основной поток, циклически выполняющий метод accept()
System.out.println("Server started");
while(true){
try{Socket socket = servSocket.accept();
Listener listener = new Listener(socket);
Thread thread = new Thread(listener);
thread.start();
}catch(IOException e){
System.err.println("IOException : " + e.toString());}}}}
public void sleeps(long time) {
try{Thread.sleep(time);}catch(InterruptedException e){}}}}
```

# Пример клиента

---

Эта программа будет запускать несколько потоков, каждый из которых независимо подключается к серверу, считывает его ответ и выводит на консоль.

```
import java.io.*;
import java.net.*;
public class NetClient implements Runnable{
public static final int PORT = 2500;
public static final String HOST = "localhost";
public static final int CLIENTS_COUNT = 5;
public static final int READ_BUFFER_SIZE = 10;
private String name = null;
public static void main(String[] args) {
String name = "name";
for(int i=1; i<=CLIENTS_COUNT; i++){
NetClient client = new NetClient(name+i);
Thread thread = new Thread(client);
thread.start();}}
```

# Клиент пример

---

```
public NetClient(String name) {this.name = name;}
public void run() {
char[] readed = new char[READ_BUFFER_SIZE];
StringBuffer strBuff = new StringBuffer();
try{
Socket socket = new Socket(HOST, PORT);
InputStream in = socket.getInputStream();
InputStreamReader reader = new InputStreamReader(in);
while(true){
int count = reader.read(readed, 0, READ_BUFFER_SIZE);
if(count==-1)break;
strBuff.append(readed, 0, count);
Thread.yield();}
} catch (UnknownHostException e) {e.printStackTrace();}
} catch (IOException e) {e.printStackTrace();}
System.out.println("client " + name + " read : " +
    strBuff.toString());}}
```

# Работа по протоколу UDP

---

Для дейтаграмм используются сокеты дейтаграммного типа. Существуют 3 конструктора:

1. DatagramSocket()-сокет к любому свободному порту.
2. DatagramSocket(int port)- на локальной машине к порту port.
3. DatagramSocket(int port, InetAddress addr)

Методы отправки и приема дейтаграмм

Send(DatagramPacket pack)-посылает дейтаграмму упакованную в пакет.  
receive(DatagramPacket pack)-получает дейтаграмму и упаковывает в пакет.

При обмене дейтаграммами соединения обычно не устанавливается, но можно установить соединение методом connect(InetAddress addr,int port);

disconnect()- разрыв соединения.

```
String mes="message";
```

```
byte[] data=mes.getBytes();
```

```
InetAddress addr = InetAddress.getByName(host);
```

```
DatagramPacket pack= new DatagramPacket (data, data.length,addr, port);
```

```
DatagramSocket ds= new DatagramSocket ();
```

```
ds.send(pack);
```

```
ds.close();
```