CodeJava
Coding Your Passion

Learn Java Network:

- Java InetAddress Examples
- Java Socket Client Examples
- Java Socket Server Examples
- Java UDP Client Server Example
- Java Chat Console Application
- How to use URLConnection and HttpURLConnection
- URLConnection and HttpURLConnection Examples
- Download file using HttpURLConnection
- Upload files programmatically
- Utility class for GET/POST request

Java Socket Server Examples (TCP/IP)

Written by Nam Ha Minh
Last Updated on 18 July 2019 | Print Email

In this Java network programming tutorial, you will learn how to develop a socket server program to implement fully functional network client/server application. You will also learn how to create a multi-threaded server.

First, let's understand about the workflow and the API.

1. ServerSocket API

The **`ServerSocket`**class is used to implement a server program. Here are the typical steps involve in developing a server program:

1. Create a server socket and bind it to a specific port number
2. Listen for a connection from the client and accept it. This results in a client socket is created for the connection.
3. Read data from the client via an `InputStream` obtained from the client socket.
4. Send data to the client via the client socket's `OutputStream`.
5. Close the connection with the client.

The steps 3 and 4 can be repeated many times depending on the protocol agreed between the server and the client.

The steps 1 to 5 can be repeated for each new client. And each new connection should be handled by a separate thread.

Let's dive into each step in details.

Create a Server Socket:

Create a new object of the `ServerSocket` class by using one of the following constructors:

- `ServerSocket(int port)`: creates a server socket that is bound to the specified port number. The maximum number of queued incoming connections is set to 50 (when the queue is full, new connections are refused).

- `ServerSocket(int port, int backlog)`: creates a server socket that is bound to the specified port number and with the maximum number of queued connections is specified by the `backlog` parameter.

- `ServerSocket(int port, int backlog, InetAddress bindAddr)`: creates a server socket and binds it to the specified port number and a local IP address.

So when to use which?

Use the first constructor for a small number of queued connections (less than 50) and any local IP address available.

Use the second constructor if you want to explicitly specify the maximum number of queued requests.

And use the third constructor if you want to explicitly specify a local IP address to be bound (in case the computer has multiple IP addresses).

And of course, the first constructor is preferred for simple usage. For example, the following line of code creates a server socket and binds it to the port number 6868:

```
1  ServerSocket serverSocket = new ServerSocket(6868);
```

Note that these constructors can throw `IOException` if an I/O error occurs when opening the socket, so you have to catch or re-throw it.

Listen for a connection:

Once a `ServerSocket` instance is created, call `accept()` to start listening for incoming client requests:

```
1  Socket socket = serverSocket.accept();
```

Note that the `accept()` method blocks the current thread until a connection is made. And the connection is represented by the returned `Socket` object.

Read data from the client:

Once a `Socket` object is returned, you can use its `InputStream` to read data sent from the client like this:

```
1  InputStream input = socket.getInputStream();
```

The `InputStream` allows you to read data at low level: read to a byte array. So if you want to read the data at higher level, wrap it in an `InputStreamReader` to read data as characters:

```
1  InputStreamReader reader = new InputStreamReader(input);
2  int character = reader.read();  // reads a single character
```

You can also wrap the `InputStream` in a `BufferedReader` to read data as String, for more convenient:

```
1  BufferedReader reader = new BufferedReader(new InputStreamReader(input));
2  String line = reader.readLine();    // reads a line of text
```

Send data to the client:

Use the `OutputStream` associated with the `Socket` to send data to the client, for example:

```
1  OutputStream output = socket.getOutputStream();
```

As the `OutputStream` provides only low-level methods (writing data as a byte array), you can wrap it in a `PrintWriter` to send data in text format, for example:

```
PrintWriter writer = new PrintWriter(output, true);
writer.println("This is a message sent to the server");
```

The argument `true` indicates that the writer flushes the data after each method call (auto flush).

Close the client connection:

Invoke the `close()` method on the client `Socket` to terminate the connection with the client:

```
socket.close();
```

This method also closes the socket's `InputStream` and `OutputStream`, and it can throw `IOException` if an I/O error occurs when closing the socket.

We recommend you to use the try-with-resource structure so you don't have to write code to close the socket explicitly.

Of course the server is still running, for serving other clients.

Terminate the server:

A server should be always running, waiting for incoming requests from clients. In case the server must be stopped for some reasons, call the `close()` method on the ServerSocket instance:

```
serverSocket.close();
```

When the server is stopped, all currently connected clients will be disconnected.

The `ServerSocket` class also provides other methods which you can consult in its Javadoc here.

Implement a multi-threaded server:

So basically, the workflow of a server program is something like this:

```
ServerSocket serverSocket = new ServerSocket(port);

while (true) {
    Socket socket = serverSocket.accept();

    // read data from the client
    // send data to the client
}
```

The `while(true)` loop is used to allow the server to run forever, always waiting for connections from clients. However, there's a problem: Once the first client is connected, the server may not be able to handle subsequent clients if it is busily serving the first client.

Therefore, to solve this problem, threads are used: each client socket is handled by a new thread. The server's main thread is only responsible for listening and accepting new

connections. Hence the workflow is updated to implement a multi-threaded server like this:

```
while (true) {
    Socket socket = serverSocket.accept();

    // create a new thread to handle client socket
}
```

You will understand clearly in the examples below.

2. Java Server Socket Example #1: Time Server

The following program demonstrates how to implement a simple server that returns the current date time for every new client. Here's the code:

```
import java.io.*;
import java.net.*;
import java.util.Date;

/**
 * This program demonstrates a simple TCP/IP socket server.
 *
 * @author www.codejava.net
 */
public class TimeServer {

    public static void main(String[] args) {
        if (args.length < 1) return;

        int port = Integer.parseInt(args[0]);

        try (ServerSocket serverSocket = new ServerSocket(port)) {

            System.out.println("Server is listening on port " + port);

            while (true) {
                Socket socket = serverSocket.accept();

                System.out.println("New client connected");

                OutputStream output = socket.getOutputStream();
                PrintWriter writer = new PrintWriter(output, true);

                writer.println(new Date().toString());
            }

        } catch (IOException ex) {
            System.out.println("Server exception: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

You need to specify a port number when running this server program, for example:

```
java TimeServer 6868
```

This makes the server listens for client requests on the port number 6868. You would see the server's output:

```
Server is listening on port 6868
```

And the following code is for a client program that simply connects to the server and prints the data received, and then terminates:

```java
import java.net.*;
import java.io.*;

/**
 * This program demonstrates a simple TCP/IP socket client.
 *
 * @author www.codejava.net
 */
public class TimeClient {

    public static void main(String[] args) {
        if (args.length < 2) return;

        String hostname = args[0];
        int port = Integer.parseInt(args[1]);

        try (Socket socket = new Socket(hostname, port)) {

            InputStream input = socket.getInputStream();
            BufferedReader reader = new BufferedReader(new InputStreamReader(ir

            String time = reader.readLine();

            System.out.println(time);


        } catch (UnknownHostException ex) {

            System.out.println("Server not found: " + ex.getMessage());

        } catch (IOException ex) {

            System.out.println("I/O error: " + ex.getMessage());
        }
    }
}
```

To run this client program, you have to specify the hostname/IP address and port number of the server. If the client is on the same computer with the server, type the following command to run it:

```
java TimeClient localhost 6868
```

Then you see a new output in the server program indicating that the client is connected:

```
New client connected
```

And you should see the client's output:

```
Mon Nov 13 11:00:31 ICT 2017
```

This is the date time information returned from the server. Then the client terminates and the server is still running, waiting for new connections. It's that simple.

3. Java Socket Server Example #2: Reverse Server (single-threaded)

Next, let's see a more complex socket server example. The following server program echoes anything sent from the client in reversed form (hence the name `ReverseServer`). Here's the code:

```
import java.io.*;
import java.net.*;

/**
 * This program demonstrates a simple TCP/IP socket server that echoes every
 * message from the client in reversed form.
 * This server is single-threaded.
 *
 * @author www.codejava.net
 */
public class ReverseServer {

    public static void main(String[] args) {
        if (args.length < 1) return;

        int port = Integer.parseInt(args[0]);

        try (ServerSocket serverSocket = new ServerSocket(port)) {

            System.out.println("Server is listening on port " + port);

            while (true) {
                Socket socket = serverSocket.accept();
                System.out.println("New client connected");

                InputStream input = socket.getInputStream();
                BufferedReader reader = new BufferedReader(new InputStreamReade

                OutputStream output = socket.getOutputStream();
                PrintWriter writer = new PrintWriter(output, true);


                String text;

                do {
                    text = reader.readLine();
                    String reverseText = new StringBuilder(text).reverse().toSt
                    writer.println("Server: " + reverseText);

                } while (!text.equals("bye"));

                socket.close();
            }

        } catch (IOException ex) {
            System.out.println("Server exception: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

As you can see, the server continues serving the client until it says ‘bye’. Run this server program using the following command:

```
java ReverseServer 9090
```

The server is up and running, waiting for incoming requests from clients:

```
Server is listening on port 9090
```

Now, let’s create a client program. The following program connects to the server, reads input from the user and prints the response from the server. Here’s the code:

```
import java.net.*;
import java.io.*;

/**
 * This program demonstrates a simple TCP/IP socket client that reads input
 * from the user and prints echoed message from the server.
 *
 * @author www.codejava.net
 */
public class ReverseClient {

    public static void main(String[] args) {
        if (args.length < 2) return;

        String hostname = args[0];
        int port = Integer.parseInt(args[1]);

        try (Socket socket = new Socket(hostname, port)) {

            OutputStream output = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(output, true);

            Console console = System.console();
            String text;

            do {
                text = console.readLine("Enter text: ");

                writer.println(text);

                InputStream input = socket.getInputStream();
                BufferedReader reader = new BufferedReader(new InputStreamReade

                String time = reader.readLine();

                System.out.println(time);

            } while (!text.equals("bye"));

            socket.close();

        } catch (UnknownHostException ex) {

            System.out.println("Server not found: " + ex.getMessage());

        } catch (IOException ex) {

            System.out.println("I/O error: " + ex.getMessage());
        }
    }
}
```

As you can see, this client program is running until the user types ‘bye’. Run it using the following command:

```
java ReverseClient localhost 9090
```

Then it asks you to enter some text:

```
Enter text:_
```

Type something, say ‘Hello’ and you should see the server’s response like this:

```
Enter text: Hello
Server: olleH
Enter text:_
```

You see the server responds ‘Server: olleH’ in which ‘olledH’ is the reversed form of ‘Hello’. The text ‘Server:’ is added to clearly separate client’s message and server’s message. The

client program is still running, asking input and printing server's response until you type 'bye' to terminate it.

Keep this first client program running, and start a new one. In the second client program, you will see it asks for input and then hangs forever. Why?

It's because the server is single-threaded, and while it is busily serving the first client, subsequent clients are block.

Let's see how to solve this problem in the next example.

4. Java Socket Server Example #3: Reverse Server (multi-threaded)

Modify the server's code to handle each socket client in a new thread like this:

```
import java.io.*;
import java.net.*;

/**
 * This program demonstrates a simple TCP/IP socket server that echoes every
 * message from the client in reversed form.
 * This server is multi-threaded.
 *
 * @author www.codejava.net
 */
public class ReverseServer {

    public static void main(String[] args) {
        if (args.length < 1) return;

        int port = Integer.parseInt(args[0]);

        try (ServerSocket serverSocket = new ServerSocket(port)) {

            System.out.println("Server is listening on port " + port);

            while (true) {
                Socket socket = serverSocket.accept();
                System.out.println("New client connected");

                new ServerThread(socket).start();
            }

        } catch (IOException ex) {
            System.out.println("Server exception: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

You see, the server creates a new `ServerThread` for every socket client:

```
while (true) {
    Socket socket = serverSocket.accept();
    System.out.println("New client connected");

    new ServerThread(socket).start();
}
```

The `ServerThread` class is implemented as follows:

```
import java.io.*;
import java.net.*;

/**
 * This thread is responsible to handle client connection.
 *
 * @author www.codejava.net
 */
public class ServerThread extends Thread {
    private Socket socket;

    public ServerThread(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        try {
            InputStream input = socket.getInputStream();
            BufferedReader reader = new BufferedReader(new InputStreamReader(ir

            OutputStream output = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(output, true);


            String text;

            do {
                text = reader.readLine();
                String reverseText = new StringBuilder(text).reverse().toStrin
                writer.println("Server: " + reverseText);

            } while (!text.equals("bye"));

            socket.close();
        } catch (IOException ex) {
            System.out.println("Server exception: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

As you can see, we just move the processing code to be executed into a separate thread, implemented in the `run()` method.

Now let run this new server program and run several client programs, you will see the problem above has solved. All clients are running smoothly.

Let experiment the examples in this lesson in different ways: run multiple clients, test on local computer, and test on different computers (the server runs on a machine and the client runs on another).

API Reference:

Socket Class Javadoc

ServerSocket Class Javadoc

Related Java Network Tutorials:

- Java InetAddress Examples
- Java Socket Client Examples (TCP/IP)
- Java UDP Client Server Program Example

Other Java network tutorials:

- How to use Java URLConnection and HttpURLConnection

- Java URLConnection and HttpURLConnection Examples
- Java HttpURLConnection to download file from an HTTP URL
- Java HTTP utility class to send GET/POST request
- How to Create a Chat Console Application in Java using Socket
- Java upload files by sending multipart request programmatically

About the Author:

Nam Ha Minh is certified Java programmer (SCJP and SCWCD). He started programming with Java in the time of Java 1.4 and has been falling in love with Java since then. Make friend with him on Facebook and watch his Java videos you YouTube.

Add comment

Name E-mail

comment

Notify me of follow-up comments

I'm not a robot reCAPTCHA Privacy - Terms

Send

Comments

1 2 3

#11**ILBOUD W ANDRE** 2022-07-22 09:56
Hello Mr Nam Ha Minh, I hope you are well.
I learn to program the sockets but with difficulties to recover the messages (it is communicated with a biochemistry automaton).
can you help me. I am ILBOUDO W ANDRE I am in BURKINA FASO in West Africa
Quote

#10**Lenny** 2021-10-13 09:44
Thankyou so much brooo, now I understand what was taught at college today!
tysmmmm

Quote

#9**Sivann San** 2021-07-24 04:10
Thank you very much!!

Quote

#8**RandomBoi** 2021-01-02 03:40
Thank you so much!! It was incredibly helpful!!!

Quote

#7**Drjones** 2020-12-30 10:45
Thank you so much, very clear and useful! :)

Quote

1 2 3

Refresh comments list

See All Java Tutorials