Цель публикации показать начинающим Java программистам все этапы создания многопоточного сервера. Для полного понимания данной темы основная информация содержится в комментариях моего кода и в выводимых в консоли сообщениях для лучшего понимания что именно происходит и в какой именно последовательности.

В начале будет рассмотрено создание элементарного клиент-сервера, для усвоения базовых знаний, на основе которых будет строиться многопоточная архитектура.

<cut />

Понятия.

— Потоки: для того чтобы не перепутать что именно подразумевается под потоком я буду использовать существующий в профессиональной литературе синоним - нить, чтобы не путать Stream и Thread, всё-таки более профессионально выражаться - нить, говоря про Thread.

— Сокеты(Sockets): данное понятие тоже не однозначно, поскольку в какой-то момент сервер выполняет - клиентские действия, а клиент - серверные. Поэтому я разделил понятие серверного сокета - (ServerSocket) и сокета (Socket) через который практически осуществляется общение, его будем называть сокет общения, чтобы было понятно о чём речь.

Кроме того сокетов общения создаётся по одному на каждом из обменивающихся данными приложении, поэтому сокет приложения которое имеет у себя объект - ServerSocket и первоначально открывает порт в ожидании подключения будем называть сокет общения на стороне сервера, а сокет который создаёт подключающееся к порту по известному адресу второе приложение будем называть сокетом общения на стороне клиента.

Оглавление:

1) Однопоточный элементарный сервер.

2) Клиент.

3) Многопоточный сервер – сам по себе этот сервер не участвует в общении напрямую, а лишь является фабрикой однонитевых делегатов(делегированных для ведения диалога с клиентами серверов) для общения с вновь подключившимися клиентами, которые закрываются после окончания общения с клиентом.

4) Имитация множественного обращения клиентов к серверу.

Спасибо за подсказку про Thread.sleep();!

Конечно в реальном коде Thread.sleep(); устанавливать не нужно - это муветон ! В данной публикации я его использую только для того чтобы выполнение программы было нагляднее, что бы успевать разобраться в происходящем.

Так что тестируйте , изучайте и в своём коде никогда не используйте Thread.sleep();!

Итак, начнём с изучения структуры однопоточного сервер, который может принять только одного клиента для диалога. Код приводимый ниже необходимо запускать в своей IDE в этом идея всей статьи. Предлагаю все детали уяснить из подробно задокументированного кода ниже:

<ul>

<li>1) Однопоточный элементарный сервер. </li>

</ul>

<source lang="java">

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.IOException;

import java.net.ServerSocket;

import java.net.Socket;

public class TestAsServer {

/\*\*

\*

\* @param args

\* @throws InterruptedException

\*/

public static void main(String[] args) throws InterruptedException {

// стартуем сервер на порту 3345

try (ServerSocket server= new ServerSocket(3345)){

// становимся в ожидание подключения к сокету под именем - "client" на серверной стороне

Socket client = server.accept();

// после хэндшейкинга сервер ассоциирует подключающегося клиента с этим сокетом-соединением

System.out.print("Connection accepted.");

// инициируем каналы для общения в сокете, для сервера

// канал записи в сокет

DataOutputStream out = new DataOutputStream(client.getOutputStream());

System.out.println("DataOutputStream created");

// канал чтения из сокета

DataInputStream in = new DataInputStream(client.getInputStream());

System.out.println("DataInputStream created");

// начинаем диалог с подключенным клиентом в цикле, пока сокет не закрыт

while(!client.isClosed()){

System.out.println("Server reading from channel");

// сервер ждёт в канале чтения (inputstream) получения данных клиента

String entry = in.readUTF();

// после получения данных считывает их

System.out.println("READ from client message - "+entry);

// и выводит в консоль

System.out.println("Server try writing to channel");

// инициализация проверки условия продолжения работы с клиентом по этому сокету по кодовому слову - quit

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

System.out.println("Client initialize connections suicide ...");

out.writeUTF("Server reply - "+entry + " - OK");

Thread.sleep(3000);

break;

}

// если условие окончания работы не верно - продолжаем работу - отправляем эхо-ответ обратно клиенту

out.writeUTF("Server reply - "+entry + " - OK");

System.out.println("Server Wrote message to client.");

// освобождаем буфер сетевых сообщений (по умолчанию сообщение не сразу отправляется в сеть, а сначала накапливается в специальном буфере сообщений, размер которого определяется конкретными настройками в системе, а метод - flush() отправляет сообщение не дожидаясь наполнения буфера согласно настройкам системы

out.flush();

}

// если условие выхода - верно выключаем соединения

System.out.println("Client disconnected");

System.out.println("Closing connections & channels.");

// закрываем сначала каналы сокета !

in.close();

out.close();

// потом закрываем сам сокет общения на стороне сервера!

client.close();

// потом закрываем сокет сервера который создаёт сокеты общения

// хотя при многопоточном применении его закрывать не нужно

// для возможности поставить этот серверный сокет обратно в ожидание нового подключения

System.out.println("Closing connections & channels - DONE.");

} catch (IOException e) {

e.printStackTrace();

}

}

}

</source>

<ul>

<li> 2) Клиент.</li>

</ul>

<cut />

Сервер запущен и находится в блокирующем ожидании server.accept(); обращения к нему с запросом на подключение. Теперь можно подключаться клиенту, напишем код клиента и запустим его. Клиент работает когда пользователь вводит что-либо в его консоли (внимание ! в данном случае сервер и клиент запускаются на одном компьютере с локальным адресом - localhost, поэтому при вводе строк , которые должен отправлять клиент не забудьте убедиться , что вы переключились в рабочую консоль клиента!).

После ввода строки в консоль клиента и нажатия enter строка проверяется не ввёл ли клиент кодовое слово для окончания общения дальше отправляется серверу , где он читает её и то же проверяет на наличие кодового слова выхода. Оба и клиент и сервер получив кодовое слово закрывают ресурсы после предварительных приготовлений и завершают свою работу.

Посмотрим как это выглядит в коде:

<source lang="java">

import java.io.BufferedReader;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.IOException;

import java.io.InputStreamReader;

import java.net.Socket;

import java.net.UnknownHostException;

public class TestASClient {

/\*\*

\*

\* @param args

\* @throws InterruptedException

\*/

public static void main(String[] args) throws InterruptedException {

// запускаем подключение сокета по известным координатам и нициализируем приём сообщений с консоли клиента

try(Socket socket = new Socket("localhost", 3345);

BufferedReader br =new BufferedReader(new InputStreamReader(System.in));

DataOutputStream oos = new DataOutputStream(socket.getOutputStream());

DataInputStream ois = new DataInputStream(socket.getInputStream()); )

{

System.out.println("Client connected to socket.");

System.out.println();

System.out.println("Client writing channel = oos & reading channel = ois initialized.");

// проверяем живой ли канал и работаем если живой

while(!socket.isOutputShutdown()){

// ждём консоли клиента на предмет появления в ней данных

if(br.ready()){

// данные появились - работаем

System.out.println("Client start writing in channel...");

Thread.sleep(1000);

String clientCommand = br.readLine();

// пишем данные с консоли в канал сокета для сервера

oos.writeUTF(clientCommand);

oos.flush();

System.out.println("Clien sent message " + clientCommand + " to server.");

Thread.sleep(1000);

// ждём чтобы сервер успел прочесть сообщение из сокета и ответить

// проверяем условие выхода из соединения

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

// если условие выхода достигнуто разъединяемся

System.out.println("Client kill connections");

Thread.sleep(2000);

// смотрим что нам ответил сервер на последок перед закрытием ресурсов

if(ois.available()!=0) {

System.out.println("reading...");

String in = ois.readUTF();

System.out.println(in);

}

// после предварительных приготовлений выходим из цикла записи чтения

break;

}

// если условие разъединения не достигнуто продолжаем работу

System.out.println("Client sent message & start waiting for data from server...");

Thread.sleep(2000);

// проверяем, что нам ответит сервер на сообщение(за предоставленное ему время в паузе он должен был успеть ответить)

if(ois.available()!=0) {

// если успел забираем ответ из канала сервера в сокете и сохраняем её в ois переменную, печатаем на свою клиентскую консоль

System.out.println("reading...");

String in = ois.readUTF();

System.out.println(in);

}

}

}

// на выходе из цикла общения закрываем свои ресурсы

System.out.println("Closing connections & channels on clentSide - DONE.");

} catch (UnknownHostException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

</source>

<ul>

<li>

3) Многопоточный сервер

</li>

</ul>

<cut />

А что если к серверу хочет подключиться ещё один клиент!? Ведь описанный выше сервер либо находится в ожидании подключения одного клиента, либо общается с ним до завершения соединения, что делать остальным клиентам? Для такого случая нужно создать фабрику которая будет создавать описанных выше серверов при подключении к сокету новых клиентов и не дожидаясь пока делегированный подсервер закончит диалог с клиентом откроет accept() в ожидании следующего клиента. Но чтобы на серверной машине хватило ресурсов для общения со множеством клиентов нужно ограничить количество возможных подключений. Фабрика будет выдавать немного модифицированный вариант предыдущего сервера(модификация будет касаться того что класс сервера для фабрики будет имплементировать интерфейс - Runnable? для возможности его использования в пуле нитей - ExecuteServices). Давайте создадим такую серверную фабрику и ознакомимся с подробным описанием её работы в коде:

<ul>

<li> Фабрика: </li>

</ul>

<source lang="java">

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

/\*\*

\* @author mercenery

\*

\*/

public class MultiThreadServer {

static ExecutorService executeIt = Executors.newFixedThreadPool(2);

/\*\*

\* @param args

\*/

public static void main(String[] args) {

// стартуем сервер на порту 3345 и инициализируем переменную для обработки консольных команд с самого сервера

try (ServerSocket server = new ServerSocket(3345);

BufferedReader br = new BufferedReader(new InputStreamReader(System.in))) {

System.out.println("Server socket created, command console reader for listen to server commands");

// стартуем цикл при условии что серверный сокет не закрыт

while (!server.isClosed()) {

// проверяем поступившие комманды из консоли сервера если такие

// были

if (br.ready()) {

System.out.println("Main Server found any messages in channel, let's look at them.");

// если команда - quit то инициализируем закрытие сервера и

// выход из цикла раздачии нитей монопоточных серверов

String serverCommand = br.readLine();

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

System.out.println("Main Server initiate exiting...");

server.close();

break;

}

}

// если комманд от сервера нет то становимся в ожидание

// подключения к сокету общения под именем - "clientDialog" на

// серверной стороне

Socket client = server.accept();

// после получения запроса на подключение сервер создаёт сокет

// для общения с клиентом и отправляет его в отдельную нить

// в Runnable(при необходимости можно создать Callable)

// монопоточную нить = сервер - MonoThreadClientHandler и тот

// продолжает общение от лица сервера

executeIt.execute(new MonoThreadClientHandler(client));

System.out.print("Connection accepted.");

}

// закрытие пула нитей после завершения работы всех нитей

executeIt.shutdown();

} catch (IOException e) {

e.printStackTrace();

}

}

}

</source>

<ul>

<li> Модифицированный Runnable сервер для запуска из предыдущего кода: </li>

</ul>

<source lang="java">

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.IOException;

import java.net.Socket;

public class MonoThreadClientHandler implements Runnable {

private static Socket clientDialog;

public MonoThreadClientHandler(Socket client) {

MonoThreadClientHandler.clientDialog = client;

}

@Override

public void run() {

try {

// инициируем каналы общения в сокете, для сервера

// канал записи в сокет следует инициализировать сначала канал чтения для избежания блокировки выполнения программы на ожидании заголовка в сокете

DataOutputStream out = new DataOutputStream(clientDialog.getOutputStream());

// канал чтения из сокета

DataInputStream in = new DataInputStream(clientDialog.getInputStream());

System.out.println("DataInputStream created");

System.out.println("DataOutputStream created");

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// основная рабочая часть //

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// начинаем диалог с подключенным клиентом в цикле, пока сокет не

// закрыт клиентом

while (!clientDialog.isClosed()) {

System.out.println("Server reading from channel");

// серверная нить ждёт в канале чтения (inputstream) получения

// данных клиента после получения данных считывает их

String entry = in.readUTF();

// и выводит в консоль

System.out.println("READ from clientDialog message - " + entry);

// инициализация проверки условия продолжения работы с клиентом

// по этому сокету по кодовому слову - quit в любом регистре

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

// если кодовое слово получено то инициализируется закрытие

// серверной нити

System.out.println("Client initialize connections suicide ...");

out.writeUTF("Server reply - " + entry + " - OK");

Thread.sleep(3000);

break;

}

// если условие окончания работы не верно - продолжаем работу -

// отправляем эхо обратно клиенту

System.out.println("Server try writing to channel");

out.writeUTF("Server reply - " + entry + " - OK");

System.out.println("Server Wrote message to clientDialog.");

// освобождаем буфер сетевых сообщений

out.flush();

// возвращаемся в началло для считывания нового сообщения

}

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// основная рабочая часть //

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// если условие выхода - верно выключаем соединения

System.out.println("Client disconnected");

System.out.println("Closing connections & channels.");

// закрываем сначала каналы сокета !

in.close();

out.close();

// потом закрываем сокет общения с клиентом в нити моносервера

clientDialog.close();

System.out.println("Closing connections & channels - DONE.");

} catch (IOException e) {

e.printStackTrace();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

</source>

<cut />

Для имитации множественного обращения клиентов к серверу, создадим и запустим (после запуска серверной части) фабрику Runnable клиентов которые будут подключаться серверу и писать сообщения в цикле:

<ul>

<li> 4) Имитация множественного обращения клиентов к серверу. </li>

</ul>

<source lang="java">

import java.io.IOException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class Main {

// private static ServerSocket server;

public static void main(String[] args) throws IOException, InterruptedException {

// запустим пул нитей в которых колличество возможных нитей ограничено -

// 10-ю.

ExecutorService exec = Executors.newFixedThreadPool(10);

int j = 0;

// стартуем цикл в котором с паузой в 10 милисекунд стартуем Runnable

// клиентов,

// которые пишут какое-то количество сообщений

while (j < 10) {

j++;

exec.execute(new TestRunnableClientTester());

Thread.sleep(10);

}

// закрываем фабрику

exec.shutdown();

}

}

</source>

<cut />

Как видно из предыдущего кода фабрика запускает - TestRunnableClientTester() клиентов, напишем для них код и после этого запустим саму фабрику, чтобы ей было кого исполнять в своём пуле:

<source lang="java">

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.IOException;

import java.net.Socket;

public class TestRunnableClientTester implements Runnable {

static Socket socket;

public TestRunnableClientTester() {

try {

// создаём сокет общения на стороне клиента в конструкторе объекта

socket = new Socket("localhost", 3345);

System.out.println("Client connected to socket");

Thread.sleep(2000);

} catch (Exception e) {

e.printStackTrace();

}

}

@Override

public void run() {

try (

// создаём объект для записи строк в созданный скокет, для

// чтения строк из сокета

// в try-with-resources стиле

DataOutputStream oos = new DataOutputStream(socket.getOutputStream());

DataInputStream ois = new DataInputStream(socket.getInputStream())) {

System.out.println("Client oos & ois initialized");

int i = 0;

// создаём рабочий цикл

while (i < 5) {

// пишем сообщение автогенерируемое циклом клиента в канал

// сокета для сервера

oos.writeUTF("clientCommand " + i);

// проталкиваем сообщение из буфера сетевых сообщений в канал

oos.flush();

// ждём чтобы сервер успел прочесть сообщение из сокета и

// ответить

Thread.sleep(10);

System.out.println("Client wrote & start waiting for data from server...");

// забираем ответ из канала сервера в сокете

// клиента и сохраняем её в ois переменную, печатаем на

// консоль

System.out.println("reading...");

String in = ois.readUTF();

System.out.println(in);

i++;

Thread.sleep(5000);

}

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

</source>

<cut />

Запускайте, вносите изменения в код, только так на самом деле можно понять работу этой структуры.

Спасибо за внимание.