Интернет програмиране с Java – част 1

Автор: Светлин Наков,
Софийски Университет "Св. Климент Охридски"
Web-site: http://www.nakov.com

Последна промяна: 12.04.2002

Курсът "Интернет програмиране с Java" е предназначен за всички, които се интересуват от програмиране на Java и разработка на Интернет-ориентирани приложения и има за цел да запознае читателя със следните технологии:

- Socket програмиране – разработка на Java приложения, които комуникират по Интернет/Интранет по протоколите TCP/IP, например Chat клиент/сървъри, Web-сървъри, Mail клиент/сървъри и др.

- Java аплети – разработка на малки Java приложения, които могат да се вграждат във Web страници и да се изпълняват от Web-браузъра на клиента.

- Web-приложения – разработка на Web приложения с технологиите Servlets и Java Server Pages (JSP), създаване и deploy-ване на Web-приложения съгласно стандартите на Sun за J2EE, работа със сървъра Tomcat.

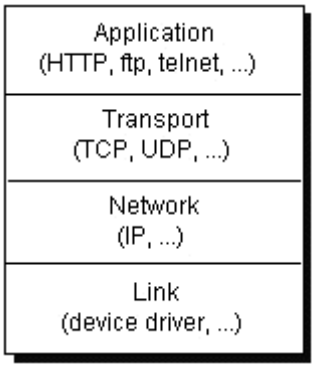
За да бъде разбран материала, е необходимо читателите да имат основни познания по организация на Интернет, програмиране, Java и HTML. Поради големият си обем, темата ще бъде разделена на няколко части (уроци), които ще бъдат публикувани в няколко последователни броя на списанието. За настоящия курс е създаден и сайт в Интернет на адрес: http://inetjava.sourceforge.net, където има много допълнителна информация по темата (на български и английски език), както и дискусионен форум.

Основи на Интернет – адреси, портове, протоколи, клиент/сървър приложения

Не можем да започнем един практически курс по разработка на Интернет приложения, без да засегнем, поне частично, основните принципи на които се основава пренасянето на данни в световната мрежа. В тази тема ще разгледаме накратко най-важните неща, които имат пряко отношение към мрежовото програмиране с Java. Предполагаме, че читателят има основни познания по организация на Интернет и затова няма да обясняваме подробно някои общоизвестни термини като например Web-сървър, "протокол" и т.н.

Основно понятие в Интернет е ***IP адрес***. Това е уникален 32-битов номер на компютър, който се записва като четири 8-битови числа, разделени по между си с точки. За улеснение на потребителите някои компютри в Интернет освен IP адрес могат да имат и име. Съответствията между IP адресите и имената на компютрите се поддържат от специални DNS сървъри, които могат да намират IP адрес по име на компютър и обратното.

Комуникацията между компютрите в Интернет е организирана на няколко нива, както показва следната диаграма:



Когато пишем Java програми които комуникират по мрежата, ние програмираме най-горния слой от диаграмата, така наречения Application слой. Преносът на данни, предизвикан от нашите Java

програми, обикновено се осъществява от транспортния слой посредством протоколите TCP или UDP. Транспортният слой използва по-долния мрежов слой за прехвърляне на малки количества информация, наречени IP пакети, от един компютър на друг, а тези пакети се прехвърлят чрез мрежови протоколи и връзки на още по-ниски нива. Като програмисти на Java, не е необходимо да знаем в детайли за всичко това, но все пак трябва да имаме представа поне от TCP и UDP протоколите дотолкова, доколкото е необходимо да преценим кога кой от тях да използваме и от IP протокола дотолкова, доколкото е необходимо да знаем, че всеки компютър в Интернет си има IP адрес, по който можем да се обръщаме към него.

TCP е протокол, който осигурява надежден двупосочен комуникационен канал между две приложения. Можем да сравним този канал с канала, по който се осъществява обикновен телефонен разговор. Например, ако искаме да се обадим на приятел, ние набираме неговия номер и когато той вдигне, се осъществява връзка между нас двамата. Използвайки тази връзка, ние можем да изпращаме и получаваме данни от нашия приятел, до момента, в който един от двамата затвори телефона и прекрати връзката. Подобно на телефонните линии, TCP протоколът гарантира, че данните, изпратени от едната страна на линията, ще се получат от другата страна на линията без изменение и то в същия ред, в който са изпратени. Ако това е невъзможно по някаква причина, ще възникне грешка и ние ще разберем, че има някакъв проблем с комуникационния канал. Именно заради тази своя надеждност, TCP е най-често използваният протокол за трансфер на информация по Интернет. Примери за приложения, които комуникират по TCP са Web-браузърите, Web-сървърите, FTP клиентите и сървърите, Mail клиентите и сървърите – приложения, за които редът на изпращане и пристигане на данните е много важен.

UDP е протокол, който позволява изпращане и получаване на малки независими един от друг пакети с данни, наречени дейтаграми, от един компютър на друг. За разлика от TCP, UDP не гарантира нито реда на пристигане на изпратените последователно дейтаграми, нито гарантира, че те ще пристигнат въобще. Изпращането на дейтаграма е като изпращане на обикновено писмо по пощата: редът на пристигане на писмата не е важен и всяко писмо е независимо от останалите. UDP се използва значително по-рядко от TCP заради това, че не осигурява комуникационен канал за данни, а позволява изпращане само на единични независими кратки пакети.

Както TCP, така и UDP протоколът позволява едновременно да се осъществяват няколко независими връзки между два компютъра. Например можем да зареждаме няколко различни Web-сайта чрез нашия Web-браузър и същевременно да теглим през FTP няколко различни файла от един или няколко FTP сървъра. Реално погледнато едно и също приложение (например нашият Web-браузър) отваря едновременно няколко независими комуникационни канала до един или няколко различни сървъра, като по всеки от тях прехвърля някаква информация. За да е възможно няколко приложения да комуникират по мрежата едновременно, е необходимо пакетите информация, предназначени за всяко от тях да бъдат обработени от него, а не от някой друг, за да може всяко приложение да получи своите данни. Именно за решаване на този конфликт се използват портовете в протоколите TCP и UDP. ***Портът*** е число между 0 и 65536 и задава идентификатор на връзката в рамките на машината. Всеки TCP или UDP пакет съдържа в себе си освен данните, които пренася, още 4 полета, описващи от кого до кого е изпратен пакета: source IP, source port, destination IP и destination port. По IP адресите се разпознават компютрите, отговорни за изпращане и получаване на съответните пакети, а по портовете се разпознават съответните приложения, работещи на тези компютри, които изпращат или трябва да получат информацията от тези пакети. Всяка TCP връзка в даден момент се определя еднозначно от порт и IP източник и порт и IP получател. Комуникационният канал, който предоставя една TCP връзка наричаме ***сокет***. Например нека нашият IP адрес е 212.50.1.81, и сме стартирали Internet Explorer и Outlook Express. С Internet Explorer ние браузваме някакъв сайт и за целта той е отворил няколко сокета към IP адрес 212.50.1.1 на порт 80 и тегли някакви Web-страници и картинки. В същото време с Outlook Express си теглим пощата, и за целта той е отворил сокет към 192.92.129.4 на порт 110. В този момент имаме няколко едновременно отворени TCP сокета (няколко независими една от друга комуникационни линии), чрез които нашият компютър комуникира с други два компютъра. Можем да ги представим схематично по следния начин:

Internet Explorer = 212.50.1.81:1033 ↔ 212.50.1.1:80 = Apache Web Server

Internet Explorer = 212.50.1.81:1037 ↔ 212.50.1.1:80 = Apache Web Server

Outlook Express = 212.50.1.81:1042 ↔192.92.129.4:110 = Microsoft Exchange POP3 Server

Първата връзка служи за изтегляне на някаква Web-страница. Тя има за източник приложението Internet Explorer и за нея е определен порт източник 1033 на нашия компютър (212.50.1.81). За получател е определен компютърът 212.50.1.1 и порт получател 80, който порт е свързан с приложението, което обслужва достъпа до Web-страниците на този компютър – Apache Web Server. Източник и получател не е съвсем точно казано, защото всички TCP връзки са двупосочни, т.е. предоставят два независими канала за данни за всяка от посоките, но все пак можем да приемем за източник това приложение, което е създало връзката (отворило сокета). Втората връзка служи за изтегляне на някаква картинка и прилича много на първата, но с една разлика – портът източник. Този порт източник е свързан също с приложението Internet Explorer на нашия компютър, но е друго число. Въпреки, че двете връзки са между едни и същи приложения, те са различни и независими, т.е. представляват два независими канала за данни. Единия служи за изтегляне на някакъв HTML документ, а другият за изтегляне на някаква картинка. Web-сървърът знае по кой от двата канала да изпрати HTML документа и по кой картинката. Internet Explorer също знае по кой от двата канала ще пристигне HTML документа и по кой картинката. Това се определя от порта източник, който е различен за двата канала. Портът източник се задава автоматично от операционната система при създаване на TCP сокет. Този порт е уникален в рамките на машината. Портът получател се задава от програмиста заедно с IP адреса получател за определяне на компютъра и приложението, към което програмата иска да се свърже. Програмистът трябва да знае предварително IP адреса и порта на приложението, с който иска да осъществи комуникация.

Съществуват два вида приложения, които комуникират по TCP протокола – клиентски и сървърски. ***Клиентските приложения*** се свързват към сървърските като отварят сокет към тях. За целта те предварително знаят техните IP адреси и портове. ***Сървърските приложения*** "слушат на определен порт" и чакат клиентско приложение да се свърже към тях. При пристигане на заявка за връзка от някой клиент на порта, на който сървърът слуша, се създава сокет за връзка между клиента на неговия порт източник и сървъра на неговия порт получател. Забележете, че клиентите отварят сокети към сървърите, а сървърите създават сокети само по клиентска заявка, т.е. те не отварят сокети. Едно клиент/сървър приложение можем да си представим като магазин с няколко щанда и клиенти, които пазаруват в него. Сървърът може да се сравни с магазин, а портът, на който слуша този сървър – с определен щанд вътре в магазина. Когато дойде клиентът, той се допуска, само ако иска да отиде на някой от щандовете, които работят (допуска се връзка само на отворен порт /порт на който слуша някое сървърско приложение/). Когато клиентът отиде на съответния щанд, той започва да си говори с продавача (осъществява комуникационна линия и прехвърля данни по нея в двете посоки) на определен език, който и двамата разбират (предварително известен протокол за комуникация). Както магазинът, така и щандът могат да обслужват няколко клиента едновременно, без да си пречат един на друг. След приключване на комуникацията клиентът си тръгва (и затваря сокета). Междувременно продавачът може да изгони клиента от магазина, ако той се държи невъзпитано или няма пари (сървърът може да затвори сокета по всяко време). За повечето операции със сокети имаме аналог с нашия пример с магазина и затова взаимодействието клиент/сървър лесно може да се интерпретира като взаимодействие потребител на услуга/извършител на услуга.

Третата връзка от показаните по-горе свърза приложението Outlook Express, което се идентифицира с порт 1042 на нашата машина (212.50.1.181) с приложението Microsoft Exchange POP3 Server, което се идентифицира с порт 110 на машината с IP адрес 192.92.129.4. Пристигналите TCP пакети на нашата машина ще бъдат разпознати от операционната система по четирите полета, които идентифицират един сокет – source IP, source port, destination IP и destination port и ако са валидни, информацията от тях ще се предаде на съответното приложение. Понеже едно приложение, както видяхме може да отвори повече от един сокет до някое друго приложение, е най-правилно да се каже, че портът източник и портът получател задават не само клиентското и сървърското приложение съответно, но и идентификатора на връзката в рамките на тези приложения (или по-точно в рамките на машината).

При UDP комуникацията концепцията с портовете е същата, само че не се осъществява комуникационен канал между приложенията, а се изпращат и получават отделни единични пакети.

Тези пакети носят в себе си същата допълнителна информация като TCP връзките – IP и порт на изпращач и IP и порт на получател. И при UDP протокола също има клиентски и сървърски приложения и по същият начин операционната система разпознава кой пакет за кое приложение е.

Комуникационните канали, наречени сокети не са достатъчни за осъществяване на комуникация. Подобно на ситуацията в магазина, клиентът трябва да комуникира със сървъра на някакъв предварително известен и за двамата език – ***протокол***. В Интернет съществува голямо количество стандартни протоколи за комуникация между приложенията, като всеки от тях е свързан с някаква услуга. Всяка услуга работи с някакъв протокол. Например услугата достъп за Web-ресурс работи по протокола HTTP, услугата за изпращане на e-mail работи по протокола SMTP, а услугата за достъп до файл от FTP сървър работи по протокола FTP. За всяка от тези стандартни Интернет услуги има и асоциирани стандартни номера на портове, на които тези услуги се предлагат. Стандартните портове са въведени за да се улесни създаването на клиентски приложения, понеже всяко клиентско приложение трябва да знае не само IP адреса или името на сървъра, на който се предлага услугата, до която то иска достъп, но също и порта, на който тази услуга е достъпна. Ето и няколко стандартни портове, протоколи и услуги:

порт	протокол	услуга
21	FTP	Услуга за достъп до отдалечени файлове. Използва се от FTP клиенти (например Internet Explorer, GetRight, CuteFTP)
25	SMTP	Услуга за изпращане на E-mail. Използва се от E-mail клиенти (например Outlook Express)
80	HTTP	Услуга за достъп до Web-ресурси. Използва се от Web-браузъри (например Internet Explorer)
110	POP3	Услуга за извличане на E-mail от пощенска кутия. Използва се от E-mail клиенти (например Outlook Express)

Java програмите могат да използват TCP и UDP протоколите за комуникация през Интернет чрез класовете от пакета `java.net`. По късно ще бъдат разгледани в детайли съответните класове, с който се могат да се създават TCP и UDP клиент/сървър приложения – `InetAddress`, `Socket`, `ServerSocket`, `DatagramSocket`, `DatagramPacket`, `URL` и др., но преди това ще направим кратък преглед на средствата за вход/изход и многонишково програмиране в Java, защото те са важна основа за разбиране на по-нататъшния материал.

Вход/изход в Java – кратък преглед

В тази тема ще направим съвсем кратък преглед на най-важните класове и методи за вход и изход в Java. Всичко останало може да се намери с документацията на JDK 1.3.

В езика Java входно-изходните операции са базирани на работа с потоци от данни. ***Потоците*** са канали за данни, при които достъпът се осъществява само последователно. Класовете, чрез които се осъществяват входно-изходните операции се намират в пакета `java.io`. Има два основни типа потоци – текстови и бинарни.

Текстовите потоци служат за четене и писане на текстова информация, а бинарните – за четене и писане на двоична информация. Базов за всички входни текстови потоци е интерфейсът `java.io.Reader`, а за всички изходни текстови потоци – `java.io.Writer`. Най-удобен за четене от текстови потоци е класът `java.io.BufferedReader`, който предлага метод за четене на цял текстов ред `readLine()`. За писане в текстови потоци е най-удобен класът `java.io.PrintWriter`, който има метод `println(…)`. Ето един прост пример за използване на текстови потоци – програма, която номерира редовете на текстов файл:

```
import java.io.*;
import java.lang.*;

public class TextFileLineNumberInserter {
   public static void main(String[] args) throws Exception {
       FileReader inFile = new FileReader("input.txt");
       BufferedReader in = new BufferedReader(inFile);

       FileWriter outFile = new FileWriter("output.txt");
```

```
        PrintWriter out = new PrintWriter(outFile);

        int counter = 0;
        String line;
        while ( (line=in.readLine()) != null ) {
            counter++;
            out.println(counter + ' ' + line);
        }

        in.close();
        out.close();
    }
}
```

Трябва да отбележим, че въпреки че Java работи вътрешно с Unicode стрингове, текстовите потоци четат и пишат символите не в Unicode, а като използват стандартните 8-бита за символ. При писане и четене информацията се преобразува от и към Unicode по текущо-активната кодова таблица, което създава известни проблеми. Това е една от причините, заради която не можем да обработваме бинарна информация с текстови потоци.

Базов за всички входни двоични потоци е интерфейсът `java.io.InputStream`, а за всички изходни двоични потоци е интерфейсът `java.io.OutputStream`. Ключов метод на `InputStream` е методът `read(byte[] b)`, който чете данни от входния поток и ги записва в масив, а ключови методи в OutputStream са `write(byte[] b, int off, int len)`, който изпраща данни от масив към изходния поток и `flush()`, който изпразва буферите и записва чакащата в тях информация.

Важно е да се съобразяваме с факта, че записването на данни, както в текстов, така и в двоичен изходен поток не винаги предизвиква тяхното изпращане. Ето защо по-нататък в нашата работа когато изпращаме двоични или текстови данни по сокет, винаги ще викаме метода `flush()`, за да сме сигурни, че данните са отпътували по сокета, а не чакат в някой буфер. Ето и фрагмент от програмен код, който копира двоични файлове:

```
FileInputStream inFile = new FileInputStream("input.bin");
FileOutputStream outFile = new FileOutputStream("output.bin");
byte buf[] = new byte[1024];
while (true) {
   int bytesRead = inFile.read(buf);
   if (bytesRead <= 0) break;
   outFile.write(buf, 0, bytesRead);
}
outFile.close();
inFile.close();
```

Многонишково програмиране в Java

В тази тема ще се запознаем с възможностите за многонишково програмиране с Java, тъй тези знания ще са ни крайно необходими в по-нататъшната ни работа.

Многонишковите (***multithreaded***) програми представляват програми, които могат да изпълняват едновременно няколко редици от програмни инструкции. Всяка такава редица от програмни инструкции наричаме ***thread*** (нишка). Изпълнението на многонишкова програма много прилича на изпълнение на няколко програми едновременно. Например в Microsoft Windows е възможно едновременно да слушаме музика, да теглим файлове от Интернет и да въвеждаме текст. Тези три действия се изпълняват от три различни програми (процеси), които работят едновременно. Когато няколко процеса в една операционна система работят едновременно, това се нарича многозадачност. Когато няколко отделни нишки в рамките на една програма работят едновременно, това се нарича ***multithreading*** (многонишковост). Например ако пишем програма, която работи като Web-сървър и Mail-сървър едновременно, то тази програма трябва да изпълнява едновременно поне 3 независими нишки – една за обслужване на Web заявките (по протокол HTTP), друга за изпращане на поща (по протокол SMTP) и трета за теглене на поща (по протокол POP3). Освен това е много вероятно за всеки потребител на тази програма да се създава по още една нишка, за да може този потребител да се обслужва независимо и да не бъде каран да чака, докато системата обслужва останалите.

С Java създаването на многонишкови програми е изключително лесно. Достатъчно е да наследим класа `java.lang.Thread` и да припокрием метода `run()`, в който да напишем програмния код на нашата нишка. След това можем да създаваме обекти от нашия клас и с извикване на метода им `start()` да започваме паралелно изпълнение на написания в тях програмен код. Ето един пример:

```
class MyThread extends Thread {
   private String name;
   private long timeInterval;

   public MyThread(String name, long timeInterval) {
       this.name = name;
       this.timeInterval = timeInterval;
   }

   public void run() {
        try {
          while (!isInterrupted()) {
              System.out.println(name);
              sleep(timeInterval);
          }
       } catch (InterruptedException intEx) {
       }
   }
}

public class ThreadTest
{
   public static void main(String[] args) {
       MyThread thread1 = new MyThread("thread 1", 1000);
       MyThread thread2 = new MyThread("thread 2", 2000);
       MyThread thread3 = new MyThread("thread 3", 1500);
       thread1.start();
       thread2.start();
       thread3.start();
   }
}
```

След стартиране на тази програма се създават и стартират 3 нишки от класа `MyThread`. Всяка от тях в безкраен цикъл печата на конзолата името си и изчаква някакво предварително зададено време между 1 и 2 секунди. Понеже трите нишки работят паралелно, се получава резултат подобен на този:

```
thread 1
thread 2
thread 3
thread 1
thread 3
thread 2
thread 1
thread 3
thread 1
...
```

Трябва да отбележим, че прекратяването на thread не трябва да става насилствено чрез метода `stop()`, а thread-ът трябва учтиво да бъде помолен да прекрати работата си чрез извикване на `interrupt()`. Затова всеки thread в програмния си код трябва да проверява, извиквайки метода `isInterrupted()`, дали не е помолен да прекрати работата си. Други интересни методи на класа `Thread` са `setPriority(int)`, `sleep()` и `setDaemon()` и за тях можем да прочетем в документацията. Средствата за синхронизация на thread-ове са изнесени в класа `java.lang.Object` в методите `wait()` и `notify()`, но за тях ще говорим в следващата тема.