Интернет програмиране с Java – част 2

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

Последна промяна: 19.05.2002

Синхронизация на нишки в Java

В предходната тема изяснихме какво е нишка (***thread***) и как се разработват многонишкови приложения с Java. В тази тема ще се запознаем с възможностите за синхронизация при многонишковото програмиране.

Има много ситуации, в които няколко нишки едновременно осъществяват достъп до общ ресурс. Например в една банка може едновременно двама клиенти да поискат да внесат пари по една и съща сметка. Да преположим, че сметките са обекти от класа Account, а операциите върху тях се извършват от класа Bank:

```
public class Account {
    private double mAmmount = 0;

    void setAmmount(double aAmmount) {
        mAmmount = aAmmount;
    }

    double getAmmount() {
        return mAmmount;
    }
}

public class Bank {
    public static void deposit(
        Account aAcc, double aSum)
    {
        double oldAmmount = aAcc.getAmmount();
        double newAmmount = oldAmmount + aSum;
        aAcc.setAmmount(newAmmount);
    }
}
```

Нека двамата клиенти се опитат едновременно да внесат съответно 100 и 500 лева в сметката `acc`, която е празна. Това би могло да стане по следния начин:

Клиент 1: `Bank.deposit(acc, 100);`
Клиент 2: `Bank.deposit(acc, 500);`

Както се вижда от кода, алгоритъмът за внасяне на пари към сметка работи така: Прочита сумата от сметката. Добавя сумата за внасяне към нея. Записвата новата сума в сметката. Ако заявките за внасяне на пари от двамата клиента се изпълняват едновременно, ще се получи следният неприятен ефект: Клиент 1 прочита сумата от сметката – 0 лева. Клиент 2 прочита сумата от сметката – също 0 лева. Клиент 1 прибавя към прочетената сума 100 лева и записва в сметката новата сума – 100 лева. Клиент 2 прибавя към прочетената сума 500 лева и записва в сметката новата сума – 500 лева. В резултат в сметката се получават 500 вместо 600 лева, а това за една банка това е абсолютно недопустимо. Натъкнахме се на класически синхронизационен проблем.

Когато няколко конкуретни нишки или няколко отделни програми се опитват едновременно да осъществят достъп до общ ресурс, се получават неприятни ефекти подобни на този с банката. Такива ефекти наричаме синхрознизационни проблеми, а техниката за решаването им наричаме синхронизация.

Синхронизацията решава проблемите с конкурентния достъп до общи ресурси, като прави достъпа до тях последователен. Тя предизвиква подреждане на заявките в последователност по такъв начин, така че когато една заявка се изпълнява, всички останали я чакат и се изпълняват едва след като тя приключи. Този процес съвсем не е автоматичен и се задава от програмиста чрез

средствата за синхронизация, които ни дава операционната система или платформата, за която разработваме софтуер.

В Java средствата за синхронизация са вградени. Запазената дума `synchronized` предизвиква синхронизирано изпълнение на програмен блок. Това означава, че две нишки не могат едновременно да изпълняват програмен код този блок. Ако едната е започнала изпълнение на код от блока, другата ще я изчака да завърши. Проблемът с банката можем да решим много просто, като заменим декларацията на метода `deposit()` от горната програма

```
public static void deposit(
    Account aAccount, double aSum)
```

с декларацията

```
synchronized public static void deposit(
    Account aAccount, double aSum).
```

Запазената дума `synchronized`, зададена при декларацията на метод предизвиква синхронизиране на изпълнението на този метод по обекта, на който той принадлежи, а при статични методи – по класа, на който той принадлежи. Синхронизацията на програмен код по някакъв обект предизвиква заключване на този обект при започване на изпълнението на синхронизирания код и отключване при завършване на изпълнението му. Когато някоя нишка се опита да изпълни синхронизиран код, чийто обект е заключен, тя принудително изчаква отключването на този обект. Така код, синхронизиран по един и същ обект, не може да се изпълнява от две нишки едновременно и заявките за изпълнението му се изпълняват една след друга. В Java синхронизацията може да става по всеки обект, защото е вградена в началния за цялата класова йерархия базов клас `java.lang.Object`. В горния пример чрез ключовата дума `synchronized` синхронизирахме достъпа до метода `deposit()` по банката, което означава, че двама клиенти не могат да бъдат едновременно обслужвани от нея. Въпреки, че това решава проблема, такъв подход не е правилен, защото заключва цялата банка, вместо само сметката, с която се работи. За да заключваме само сметката, до която методът `deposit()` осъществява достъп можем да използваме следния синхронизиран код:

```
public static void deposit(
    Account aAccount, double aSum)
{
    synchronized (aAccount) {
        double oldAmmount = aAccount.getAmmount();
        double newAmmount = oldAmmount + aSum;
        aAccount.setAmmount(newAmmount);
    }
}
```

Това вече решава правилно проблема, защото заключва само сметката, която се променя, а не цялата банка. Важно е когато се използва заключване на обекти, да се заключва само този обект, който се променя и то само за времето, през което се променя, а не за по-дълго, за да може конкурентните нишки да чакат минимално при опит за достъп до него. Освен това, ако достъпът до някакъв обект трябва да е синхронизиран, той трябва да е синхронизиран навсякъде, където се работи с този обект. В противен случай полза от синхронизацията няма. Например ако в нашата банка внасянето на пари е синхронизирано, а тегленето не е синхронизирано, възможността за грешки при финансовите операции ще си остане съвсем реална.

Въпреки че синхронизацията чрез запазената дума `synchronized` върши работа в повечето случаи, тя съвсем не е достатъчна. За това в класа `java.lang.Object` съществуват още няколко важни метода свързани със синхронизацията – `wait()`, `notify()` и `notifyAll()`. Методът `wait()` приспива текущата нишка по даден обект докато друга нишка не извика `notify()` за същия обект за да я събуди. Методът `notify()` събужда една от заспалите по даден обект нишки, а `notifyAll()` събужда всичките. Ако по обекта няма заспали нишки, `notify()` и `notifyAll()` не правят нищо. Понякога за работата на една нишка е необходим ресурс, който се получава в резултат от работата на друга нишка. В този случай първата нишка трябва да изчака втората да свърши някаква работа и след това да продължи своето изпълнение. Ако първата нишка в един цикъл постоянно проверява дали втората е свършила очакваната работа, тя ще консумира по неразумен начин много процесорно време и ще намали производителността на цялата система. Много по-ефективно е

първата нишка да заспи, очаквайки събуждане от втората, а втората да свърши очакваната работа и веднага след това да събуди първата. Характерното за заспалите (или както още се наричат блокирали) нишки е, че не консумират процесорно време, което е причината приспиването на нишки да бъде предпочитан начин за чакане на ресурси.

Да разгледаме един класически проблем, известен като "производител – потребител". Един завод произвежда някаква продукция и разполага със складове в които може да побере някакво определено количество от нея. Когато складовете се напълнят заводът спира работа докато не продаде част от продукцията за да освободи място. Търговците от време на време идват в складовете и купуват част от произведената продукция. Когато търговец дойде и складът е празен, той чака докато заводът произведе продукция, за да му я продаде. Взаимодействието между производителя (завода) и потребителите (търговците) представлява постоянен процес, в който всеки върши своята работа, но същевременно зависи от другите и ги изчаква ако е необходимо. Проблемът "производител – потребител" се изразява в това да се организира коректно многонишковият процес на взаимодействие между производителя и потребителите без да се отнема излишно процесорно време когато някой чака някого за някакъв ресурс. Нека ресурсите, които производителят произвежда и потребителите консумират са текстови съобщения, а буферът, с който производителят разполага, е опашка с вместимост 5 съобщения. Следната е програма е примерно решение на проблема, реализирано на базата на средствата за синхронизация в Java:

```
import java.util.*;

class Producer extends Thread {
    static final int MAXQUEUE = 5;
    private Vector messages = new Vector();

    public void run() {
        try {
            while (true) {
                putMessage();
                sleep(1000);
            }
        } catch(InterruptedException e) { }
    }

    private synchronized void putMessage()
        throws InterruptedException
    {
        while (messages.size() == MAXQUEUE)
            wait();
        messages.addElement(
            new Date().toString() );
        notify();
    }

    // Called by Consumer
    public synchronized String getMessage()
        throws InterruptedException
    {
        notify();
        while (messages.size() == 0)
            wait();
        String message =
            (String)messages.firstElement();
        messages.removeElement( message );
        return message;
    }
}

class Consumer extends Thread {
    Producer producer;

    Consumer(Producer p) {
        producer = p;
    }

    public void run() {
        try {
            while (true) {
```

```
                String message =
                    producer.getMessage();
                System.out.println(message);
                sleep(1500);
            }
        } catch( InterruptedException e ) { }
     }
}

public class TestProducerConsumer {
    public static void main(String args[]) {
        Producer producer = new Producer();
        producer.start();
        Consumer c1 = new Consumer(producer);
        c1.start();
        Consumer c2 = new Consumer(producer);
        c2.start();
    }
}
```

Разглеждайки кода на програмата и документацията на методите `wait()` и `notify()`, вероятно ще ви направи впечатление, че тези методи могат да се викат само от код, който е синхронизиран по обекта, на който те принадлежат. Ако методът за заспиване и методът за събуждане се викат от различни нишки и са в блокове код, синхронизирани по един и същ обект, би трябвало след заспиването на първата нишка кодът, който я събужда никога да не се изпълни, защото ще чака завършването на заспалата нишка. Изглежда, че и двете нишки ще заспят за вечни времена, като едната ще чака другата да я събуди, а другата ще чака първата да излезе от синхронизирания код, но това няма да се случи никога понеже е заспала. Тези разсъждения, обаче са погрешни, защото извикването на `wait()` не само приспива текущата нишка, но и отключва обекта, по който тя е синхронизирана. Това позволява на блокът извикващ `notify()`, който е синхронизиран по същия обект, да се изпълни. Извикването на `notify()` събужда заспалата нишка, но не й разрешава веднага да продължи изпълнението си. Събудената нишка изчаква завършването на синхронизирания блок, от който е извикан `notify()`. След това продължава изпълнението си като заключва отново синхронизационния обект и го отключва едва след завършването на синхронизирания блок, в който е била заспала. Така заспиването за вечни времена, наричано още deadlock или мъртва хватка не настъпва. При неправилна употреба на средствата за синхронизация, обаче, настъпването на мъртва хватка съвсем не е изключено. Отговорност на програмиста е да предотврати възможността две или повече нишки в някой момент да започнат да се чакат взаимно.

Работа с TCP сокети

Както вече знаем от краткия преглед на Интернет протоколите, който направихме в началото, TCP сокетите представляват надежден двупосочен транспортен канал за данни между две приложения. Приложенията, които си комуникират през сокет, могат да се изпълняват на един и същ компютър или на различни компютри, свързани по между си чрез Интернет или друга TCP/IP мрежа. Тези приложения биват два вида – сървъри и клиенти. Клиентите се свързват към сървърите по IP адрес и номер на порт чрез класа `java.net.Socket`. Сървърите приемат клиенти чрез класа `java.net.ServerSocket`. Да разгледаме основната част от кода на едно просто сървърско приложение – DateServer:

```
ServerSocket srvSock = new ServerSocket(2002);
while (true) {
    Socket socket = srvSock.accept();
    OutputStreamWriter out =
        new OutputStreamWriter(
            socket.getOutputStream() );
    out.write(new Date()+ "\n");
    out.close();
    socket.close();
}
```

Този сървър отваря за слушане TCP порт 2002, след което в безкраен цикъл приема клиенти, изпраща им текущата дата и час и веднага след това затваря сокета с тях. Отварянето на сокет за слушане става като се създава обект от класа `ServerSocket`, в конструктура на който се задава номера на порта. Приемането на клиент се извършва от метода `accept()` на класа `ServerSocket`, при извикването на който текущата нишка блокира до пристигането на клиентска заявка, след което създава сокет между сървъра и пристигналия клиент. От създадения сокет сървърът взема изходния поток за изпращане на данни към клиента (чрез метода `getOutputStream()`) и изпраща в него текущата дата и час, записани на една текстова линия. Затварянето на изходния поток е важно. То предизвиква действителното изпращане на данните към клиента, понеже извиква метода `flush()` на изходния поток. Ако нито един от методите `close()` или `flush()` не бъде извикан, клиентът няма да получи нищо, защото изпратените данни ще останат в буфера на сокета и няма да отпътуват по него. Накрая, затварянето на сокета предизвиква прекъсване на комуникацията с клиента. Сървъра можем да изтестваме със стандартната програмка telnet, която е включена в повечето версии на Windows, Linux и Unix като напишем на конзолата следната команда:

```
telnet localhost 2002
```

Резултатът е получената от сървъра дата:

```
Sat May 18 22:46:55 EEST 2002
```

Нека сега напишем клиент за нашия сървър – програма, която се свързва към него, взема датата и часа, които той връща и ги отпечатва на конзолата. Основната част от кода е следната:

```
Socket socket = new Socket("localhost", 2002);
BufferedReader in = new BufferedReader(
    new InputStreamReader(
        socket.getInputStream() ) );
System.out.println(in.readLine());
socket.close();
```

Свързването към сървър става чрез създаването на обект от класа `java.net.Socket`, като в конструктора му се задават IP адреса или името на сървъра и номера на порта. От свързания успешно сокет се взема входния поток и се прочита това, което сървърът изпраща. След приключване на работа сокетът се затваря. При работа със сокети и входно-изходни потоци понякога възникват грешки, в резултат на което се хвърлят изключения (exceptions). Затова е задължително и в двете програми, които дадохме за пример, кодът, който комуникира по сокет да бъде поставен в `try ... catch` блок, за да бъдат обработени евентуално възникналите грешки. Грешки възникват в най-разнообразни ситуации. Например ако сървърът не е пуснат и клиентът се опита да се свърже с него, ако връзката между клиента и сървъра се прекъсне, ако сървърът се опита да слуша на зает вече порт и в много други случаи. Също от важно значение е, че ако клиентът се опита да прочете данни от сървъра, а той не му изпрати нищо, клиентът ще блокира до затваряне на сокета. Затова сървърът и клиентът трябва да комуникират по предварително известен и за двамата протокол и да го спазват стриктно.

Даденият по-горе пример за сървър обслужва клиентите последователно. Ако двама клиенти едновременно дадат заявка, първият ще бъде обслужен веднага, а вторият едва след приключване на обслужването на първия. Тази стратегия работи, но само за прости сървъри, в които обслужването на клиент отнема много малко време. В повечето случаи обслужването на един клиент отнема известно време и останалите клиенти не могат да бъдат карани да го чакат. Затова се налага сървърът да се обслужва клиентите едновременно и независимо един от друг. За реализация на тази стратегия в средата на Java се използва многонишковия подход, при който за всеки клиент се създава отделна нишка. Това е препоръчвания начин за разработка на сървъри, предназначени да работят с повече от един клиент. Ще демонстрираме силата на многонишковия подход с един стандартен пример – сървър за разговори (chat server):

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

public class NakovChatServer {
    public static void main(String[] args)
```

```
    throws IOException {
        ServerSocket serverSocket =
            new ServerSocket(5555);
        System.out.println("Nakov Chat Server"
            + " started on port " +
            serverSocket.getLocalPort());

        ServerMsgDispatcher dispatcher =
            new ServerMsgDispatcher();
        dispatcher.start();

        while (true) {
            Socket clientSock =
                serverSocket.accept();
            ClientListener clientListener =
                new ClientListener(clientSock,
                    dispatcher);
            dispatcher.addClient(clientSock);
            clientListener.start();
        }
    }
}

class ClientListener extends Thread {
    private Socket mSocket;
    private ServerMsgDispatcher mDispatcher;
    private BufferedReader mIn;

    public ClientListener(Socket aSocket,
    ServerMsgDispatcher aServerMsgDispatcher)
    throws IOException {
        mSocket = aSocket;
        mIn = new BufferedReader(
            new InputStreamReader(
                mSocket.getInputStream()));
        mDispatcher = aServerMsgDispatcher;
    }

    public void run() {
        try {
            while (!isInterrupted()) {
                String msg = mIn.readLine();
                if (msg == null) break;
                mDispatcher.dispatchMsg(
                    mSocket, msg);
            }
        } catch (IOException ioex) { }
        mDispatcher.deleteClient(mSocket);
    }
}

class ServerMsgDispatcher extends Thread {
    private Vector mClients = new Vector();
    private Vector mMsgQueue = new Vector();

    public synchronized void addClient(
    Socket aClientSocket) {
        mClients.add(aClientSocket);
    }

    public synchronized void deleteClient(
    Socket aClientSock) {
        int i = mClients.indexOf(aClientSock);
        if (i != -1) {
            mClients.removeElementAt(i);
            try {
                aClientSock.close();
            } catch (IOException ioe) {}
        }
    }

    public synchronized void dispatchMsg(
    Socket aSocket, String aMsg) {
        String IP = aSocket.getInetAddress().
```

```
            getHostAddress();
        String port = "" + aSocket.getPort();
        aMsg = IP + ":" + port + " : " +
            aMsg + "\n\r";
        mMsgQueue.add(aMsg);
        notify();
    }

    private synchronized String
    getNextMsgFromQueue()
    throws InterruptedException {
        while (mMsgQueue.size() == 0)
            wait();
        String msg = (String) mMsgQueue.get(0);
        mMsgQueue.removeElementAt(0);
        return msg;
    }

    private synchronized void
    sendMsgToAllClients(String aMsg) {
        for (int i=0; i<mClients.size(); i++) {
            Socket socket =
                (Socket) mClients.get(i);
            try {
                OutputStream out =
                    socket.getOutputStream();
                out.write(aMsg.getBytes());
                out.flush();
            } catch (IOException ioe) {
                deleteClient(socket);
            }
        }
    }

    public void run() {
        try {
            while (true) {
                String msg =
                    getNextMsgFromQueue();
                sendMsgToAllClients(msg);
            }
        } catch (InterruptedException ie) { }
    }
}
```

Като функционалност сървърът не е сложен. Единственото, което прави той е да приема съобщения от клиентите си и да изпраща всяко прието съобщение до всеки клиент, като отбелязва в него от кого го е получил. Можем да го изтестваме като отворим няколко telnet-сесии към порт 5555 по същия начин, както в предния пример с Date-сървъра.

Сървърът има две основни нишки. Едната е главната програма (`NakovChatServer`), която слуша на порт 5555 и приема нови клиенти, а другата е нишката-диспечер (`ServerMsgDispatcher`), която разпраща получените от клиентите съобщения до всички свързани към сървъра. За всеки клиент в сървъра се създава още една нишка (обект от класа `ClientListener`), която служи за получаване на съобщения от него. При стартирането си сървърът отваря за слушане порт 5555, създава диспечера за съобщения и го стартира. След това в безкраен цикъл започва да приема клиенти, да ги добавя в списъка на диспечера, да създава по една нишка за получаване на съобщенията от тях и да я стартира. Нишката за получаване на съобщения от клиент в основния си цикъл (метода `run()`) чете съобщения от клиента, добавя ги в опашката на диспечера (извиквайки метода `DispatchMsg()`), след което го събужда ако е заспал (като му `notify()` метода). Четенето на съобщение става с метода `readLine()` и е операция, която блокира нишката докато не пристигне съобщение или не настъпи грешка. При настъпване на грешка, клиентът се премахва от списъка на диспечера (чрез извикване на `deleteClient()`). Нишката `ServerMsgDispatcher` е добър пример за използване на проблема "производител – потребител" в практиката. В основния си цикъл (в метода `run()`) нишката взема от опашката си поредното съобщение и го разпраща до всички клиенти. В този цикъл тя се явява консуматор на съобщения. Ако опашката е празна, нишката чака (извиквайки `wait()`) докато пристигне ново съобщение. Съобщенията пристигат асинхронно чрез

извиквания от нишките за обслужване на клиент. Клиентите играят ролята на производител на съобщения. Диспечерът пази всички активни клиенти в един списък. За да поддържа списъка актуален, сървърът добавя в него всеки нов клиент при пристигането му и го премахва от там при първия неуспешен опит за изпращане или получаване на съобщение от него. Така например, ако клиентът затвори сокета, той ще бъде премахнат от списъка, защото четенето на съобщение от него ще се провали. Макар и сървърът да е способен да обслужва много клиенти едновременно, не може все още да твърди, че е добре написан. Проблемът е, че нишката-диспечер, която изпраща получените съобщения, работи последователно. Тя не преминава към изпращането на следващо съобщение от опашката докато не изпрати текущото на всички клиенти. Ако връзката с един от клиентите е много бавна, заради него всички ще се наложи да чакат преди да получат следващото изпратено съобщение. Ето защо е необходимо диспечерът да работи паралелно. Единият вариант е да се създава по една нишка за всяко изпращане на съобщение до някой клиент. Това обаче означава, че ако сървърът има 1000 клиента и получи почти едновременно 100 съобщения, ще трябва да създаде 100000 нишки, за да разпрати съобщенията до клиентите. Създаването и изпълнението на толкова много нишки обаче, изисква огромно количество процесорно време и памет (особено при програмиране на Java), така че ще ни е необходим доста мощен компютър за да може така модифицираният chat-сървър да работи със задоволителна скорост. Ето защо е по-разумно за всеки клиент да се създаде по една нишка, която служи за разпращане на съобщенията, предназначени за него. Тази нишка трябва да поддържа опашка от съобщения, защото ако съобщенията пристигат по-бързо отколкото се могат да се изпратят, ще възникне проблем. Тя трябва да заспива, когато опашкатае празна и да се събужда, когато в нея постъпи съобщение. Тази нишка може да се реализира по същия начин като класа `ServerMsgDispatcher`, защото има почти същата функционалност. Примерна реализация на този подход има на сайта на курса “Интернет програмиране с Java” – http://inetjava.sourceforge.net.