Нишки в Java

Ненко Табаков, Пламен Танов, Любомир Чорбаджиев

Технологично училище “Електронни системи”
Технически университет, София

10 юли 2009

Забележка: Тази лекция е адаптация на лекции от следните курсове:

- Prof. Judson Harward, Prof. Steven Lerman: *1.00 / 1.001 Introduction to Computers and Engineering Problem Solving Fall 2005* (MIT OpenCourseWare: Massachusetts Institute of Technology)
  Лиценз: Creative commons BY-NC-SA
- Dr. George Kocur: *1.00 Introduction to Computers and Engineering Problem Solving Spring 2005* (MIT OpenCourseWare: Massachusetts Institute of Technology)
  Лиценз: Creative commons BY-NC-SA

Тези материали са разработени в рамките на проекта "Софтуерна акадамеия "Електронни системи", съфинансиран от Европейския съюз и Европейския социален фонд.

Литература:

- The Java Tutorials. Trail: Essential Classes; Lesson: Concurrency.

Съдържание

Процеси и нишки

- Повечето операционни системи позволяват едновременно (конкурентно) изплънение на няколко процеса.
- Процесите са ресурсоемки.
- Процесите са толкова добре изолирани един от друг, че това прави комуникацията между тях трудна и сложна.
- За разлика от процесите нишките са значително по-леки и по-малко ресурсоемки. Нишките работят в рамките на едни процес, и между тях на практика няма изолация.

Нишки в Java

- Java е един от малкото езици за програмиране, които имат вградена в езика поддръжка за многонишково програмиране.
- Един от първите масови езици за програмиране, които разполагат с вградена поддръжка за многонишково програмиране е Ada.
- C# също притежва вградена поддръжка за многонишково програмиране.
- Езиците C и C++ нямат вградена поддръжка за многонишково програмиране. Вместо това те разчитат на външни библиотеки, които да предоставят такава поддръжка.
- Очакваният нов стандарт за C++ предвижда въвеждане в стандартната C++ библиотека на средства за многонишково програмиране (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2497.html).

Нишки в Java

- В Java сибирането на "боклука" от обекти, които не са нужни, се изпълнява от виртуалната машина в отделна нишка.
- Библиотеките за създаване на графични интерфейси използват отделна нишка за обработка и разпределение на събитията, генерирани от графичния интерфейс. Това позволява на графичния интерфейс на приложението да остава отзивчив дори когато приложението е заето с дълги пресмятания или входно/изходни операции.
- Всичко това показва, че на езика Java му е вътрешно присъща могонишковост. Дори когато дадена програма е написана така, че да не използва повече от една нишки, средата в която се изпълнява тази програма използва много нишки.
- Java предоставя средства, чрез които програмистите могат лесно да създават многонишкови приложения.

Класът Thread

- Класът Thread предоставя поддръжка за създаване на нишки в Java.
- Има два начина, по които можете да кажете на нишката какво да прави:
  - Като наследим класа Thread и заместим метода run().
  - Като предадем на конструктора на класа Thread обект, който реализира интерфейса Runnable.

Класът Thread: заместване на метода run()

- Наследяваме класа Thread и предефинираме метода **void** run():

```
1 class MyThread extends Thread {
2   ...
3   void run() {
4     // код, който се изпълнява от нишката
5   }
6 }
```

- За да създадем нишка трябва да създадем обект от класа MyThread и да извикаме метода start().

```
1 MyThread myThread=new MyThread();
2 myThread.start();
```

Класът Thread: използване на Runnable

- Друг възможен вариант за описание на поведението на нишка е да се използва отделен клас, който наследява интерфейсът Runnable:

```
1 interface Runnable {
2   public void run();
3 }
```

- За тази цел трябва де дефинирате клас, който реализира интерфейса Runnable:

```
1 class MyRunnable implements Runnable {
2   public void run{
3     // код, който се изпълнява в нишка
4   }
5 }
```

Класът Thread: използване на Runnable

- При създаването на нишка в конструктора на класа Thread се предава обект от класа MyRunnable:

```
1 Thread myThread=new Thread(new MyRunnable());
2 myThread.start();
```

- Една от причините да се използва този подход при дефиниране на поведението на нишки е, че в Java класовете могат да наследяват само един клас. При това положение, ако класът в който трябва да се опише поведение на нишка, вече е наследил друг клас, то използването на предходната стратегия става невъзможна.

Пускане и спиране на нишки

- За да се пусне нишка е необходимо да се създаде обект от класа `Thread` или негов наследник и да се извика метода `start()`.

```
1 Thread t1=new MyThread();
2 t1.start();
3 Thread t2=new Thread(new MyRunnable());
4 t2.start();
```

- Класът `Thread` по принцип притежва метод `stop()`. Целта на този метод е била той да се използва за спиране не нишки. Този метод обаче е несигурен от гледна точка на конкурентното програмиране. Поради това изпозването на този метод не се препоръчва и предстои махането на този метод от класа `Thread`.

Пускане и спиране на нишки

- Начинът за спиране на една нишка е нейният метод `run()` да завърши работа. Не използвайте метода `stop()` за да спрете нишка.
- Класът `Thread` притежава метод `isAlive()`, с който можете да проверите дали нишката все още работи или е завършила работа:

```
1 Thread t=new MyThread();
2 t.start();
3 ...
4 if(t.isAlive()) {
5   // нишката все още работи
6 } else {
7   // нишката е завършила
8 }
```

Пускане и спиране на нишки

- Класът `Thread` притежава метод `join()`, с чиято помощ можете да изчакате завършването на работата на дадена нишка:

```
1 Thread t=new MyThread();
2 t.start();
3 ...
4 t.join(); // спира и изчаква докато нишката t завърши
5 ...
```

Пример за използване на нишки

Пример за използване на нишки

```
1  package labs.threads;
2
3  public class URLCopyMain{
4    public static void main(String argv[]){
5      String[][] fileList={
6  {
7  "http://www.nps.gov/imr/pgallerycontent/p/l/20071129184415.
8  "Bryce.jpg"},
9  {
10 "http://microscopy.fsu.edu/micro/gallery/dinosaur/dino1.jpg
11 "dino1.jpg"},
12 {
13 "http://java.sun.com/docs/books/tutorial/index.html",
14 "tutorial.index.html"}};
```

Пример за използване на нишки

```
12      for(int i=0;i<fileList.length;i++){
13        Thread th;
14        String threadName="T"+i;
15        String fromURL=fileList[i][0];
16        String toFile=fileList[i][1];
17
18        th=new URLCopyThread(threadName,fromURL,toFile);
19        th.start();
20        System.err.println("Thread␣"+th.getName()
21            +"␣to␣copy␣from␣"+fileList[i][0]+"␣to␣"
22            +fileList[i][1]+"␣started");
23      }
24    }
25  }
```

Пример за използване на нишки

```
1  package labs.threads;
2
3  import java.io.BufferedInputStream;
4  import java.io.BufferedOutputStream;
5  import java.io.FileOutputStream;
6  import java.io.IOException;
7  import java.net.MalformedURLException;
8  import java.net.URL;
9
10 public class URLCopyThread extends Thread{
11   private URL fromURL;
12   private BufferedInputStream input;
13   private BufferedOutputStream output;
14   private String from,to;
```

```
12  public URLCopyThread(String n,String f,String t){
13    super(n);
14    from=f;
15    to=t;
16    try{
17      fromURL=new URL(from);
18      input=new BufferedInputStream(fromURL.openStream());
19      output=new BufferedOutputStream(
20          new FileOutputStream(to));
21    }catch(MalformedURLException m){
22      System.err.println("MalformedURLException␣"
23          +"␣creating␣URL␣"+from);
24    }catch(IOException io){
25      System.err.println("IOException␣"+io.toString());
26    }
27  }
```

```
26    public void run(){
27      byte[] buf=new byte[512];
28      int nread;
29      try{
30        while((nread=input.read(buf,0,512))>0){
31          output.write(buf,0,nread);
32          System.err
33              .println(getName()+": "+nread+" bytes");
34        }
35        input.close();
36        output.close();
37        System.err.println("Thread "+getName()+" copying "
38            +from+" to "+to+" finished");
39      }catch(IOException ioe){
40        System.out.println("IOException:"+ioe.toString());
41      }
42    }
43  }
```

Необходимост от синхронизация на нишки

```
1  package labs.threads;
2
3  public class UnsynchornizedExample{
4    final static int THREADS_COUNT=50;
5    final static int CYCLES=1000000;
6
7    public long value=0;
8
9    public static class Incrementor implements Runnable{
10     private UnsynchornizedExample ute;
11     public Incrementor(UnsynchornizedExample ute){
12       this.ute=ute;
13     }
14     public void run(){
15       for(int i=0;i<CYCLES;i++){
16         ute.value++;
17       }
18     }
19   }
```

```
17   public static void main(String[] args){
18     UnsynchornizedExample ute
19         =new UnsynchornizedExample();
20     Thread[] threads=new Thread[THREADS_COUNT];
21     for(int i=0;i<THREADS_COUNT;i++){
22       threads[i]=new Thread(new Incrementor(ute));
23     }
24     for(int i=0;i<THREADS_COUNT;i++){
25       threads[i].start();
26     }
27     try{
28       for(int i=0;i<THREADS_COUNT;i++){
29         threads[i].join();
30       }
31     }catch(InterruptedException e){
32       e.printStackTrace();
33     }
34     System.out.println("value="+ute.value);
35   }
36 }
```

Необходимост от синхронизация на нишки

- Дадени са 50 нишки, които инкрементират една и съща променлива 1000000 пъти. Очакването е след като нишките прюключат работа, стойността на променливата `value` да бъде равна на `50*1000000=50000000`.
- Резултата от няколко последователни пускания на програмата е следния:

```
value=47643339
value=47822482
value=50000000
value=48498127
```

- Проблемът е липсата на синхронизация на нишките при едновременен достъп до общ ресурс — променливата `value`.

Атомарност

- За да бъде коректно предходната програма `UnsynchronizedExample.java`, е необходимо операцията:

```
value++;
```

  да бъде атомарна.
- *Атомарна* се нарича операция, която се изпълнява изцяло, без да бъде прекъсвана от виртуалната машина.

Атомарност

- Операцията

```
value++;
```

типично не е атомарна.

- Изпълнението на операцията `value++` може да се представи по следния начин:

```
1 local1=value;
2 local1=local1+1;
3 value=local1;
```

Състояние на надпревара

- Когато две нишки или повече нишки се опитат конкурентно да променят състоянието на общ ресурс, инструкциите им могат да се преплетат.
- Преплитането на инструкциите на две или повече нишки зависи от начина, по който нишките се планират върху процесора.

Състояние на надпревара

- Да предположим, че първоначално `value=5`. Един вариант за преплитане на инструкциите на две нишки е следния:

```
1 thread[0]: local0=value        /*local1 = 5*/
2 thread[0]: local0=local0+1     /*local1 = 6*/
3 thread[1]: local1=value        /*local2 = 5*/
4 thread[1]: local1=local1+1     /*local2 = 6*/
5 thread[1]: value=local1        /*value = 6 */
6 thread[0]: value=local0        /*value = 6 */
```

- Стойността на `value` става 6, докато правилния резултат трябва да бъде 7.

Състояние на надпревара

- *Състояние на надпревара (race condition)* се нарича ситуацията, при която няколко процеса или нишки конкурентно манипулират данни, които са общи, поделени между конкурентните процеси (нишки).
- Крайната стойност на поделените (общите) данни зависи от начина, по който процесите се планират върху процесора.
- За да се предотврати състоянието на надпревара (race condition), манипулацията на общите ресурси от конкурентните процеси (нишки) трябва да бъде синхронизирана.

Критичен участък (critical section)

- Няколко процеса или нишки се състезават за достъп до общи (поделени) данни.
- Всеки процес или нишка има участък от кода, в който работи с общите данни. Такъв участък от кода се нарича *критичен участък*.
- Трябва да се изгради механизъм, чрез който да се гарантира, че когато един процес се намира в критична секция, никой друг процес не може да влезе в своя критичен участък.

Критичен участък в Java: sychronized

- За дефиниране на критични участъци от кода Java предоставя ключовата дума **synchronized**.
- Ключовата дума **synchronized** може да се използва по няколко начина. Една от възможните употреби е методи да се дефинират като **synchronized**.

```
1 class Foo {
2 ...
3   public synchronized void bar() {
4     ...
5   }
6 ...
7 }
```

Критичен участък в Java: sychronized

- Дефинирането на метода `bar()` като **synchronized** означава, че този метод не може да бъде прекъснат (не може да се преплете) с други **synchronized** методи, работещи върху същия обект.
- Ако друга нишка се опита да изпълни **synchronized** метод върху същия обект, то тя ще бъде блокирана, докато първия **synchronized** метод не завърши работа.
- Обърнете внимание, че **synchronized** методи се блокират само от други **synchronized** методи. Методите, които не са синхронизирани, работят независимо.
- Синхронизират се методите, работещи върху един и същи обект. Докато работи **synchronized** метод върху даден обект, друга нишка може да изпълнява **synchronized** метод на друг обект.

Критичен участък в Java: sychronized