Нишки

Понятието „нишки“ можете да разгледате като „разклонение на една програма на подпрограми, които работят едновременно“. Представете си например един HTTP сървър. Работата, която извършва е да прехвърля данните поискани от даден клиент. Когато обаче има повече от един клиент едновременно, то ако нямаме „многонишковост“ те трябва да се изчакват един друг на „опашка“. Ако имаме един клиент, който изтегля огромно количество информация, то всички останали би трябвало да го изчакват, а това определено не е уместно. Пускането на нишка за всеки един клиент означава, че ние ще обслужваме всички с равно количество процесорно време, независимо кой се е свързал с HTTP сървъра първи, кой втори и кой последен.

Първо трябва да правите разлика между „многонишковост“ и „многозадачност“. Когато говорим за „многозадачност“, то се има в предвид изпълнението на множество „процеси“ (програми) от операционната система едновременно. Тяхното „едновременно“ изпълнение не се контролира от тях, а именно от операционната система. При „многонишковост“ говорим за контрол над изпълнението на „подпрограми“ от самата програма. Затова стартирането на едно и също приложение два пъти НЕ създава „две нишки“, а създава „два процеса“.

По същество всяка една програма, която демонстрирахме досега се състоеше от една единствена нишка. Тя от своя страна има право да „пусне“ още една или повече нишки. Естествено никой не ограничава и те да пускат „под-нишки“ от своя страна. По този начин казваме, че се създава „дърво от нишки“. Важното в случая е, че една нишка не може да съществува ако нишката, която я е извикала престане да съществува.

Има два начина за „пускане на нишка“. Първият е чрез наследяване на системния клас „Thread“:

```
public class ThreadExample extends Thread{
   public void run(){
      ... оператори ...
   }
}
```

Другият е чрез имплементирането на интерфейс Runnable:

```
public class ThreadExample implements Runnable{
   public void run(){
      ... оператори ...
   }
}
```

При първият метод директно наследяваме функционалността на клас Thread и добавяме детайли в неговия „run()“ метод. Във втория ние само имплементираме интерфейсът Runnable, а в последствие предаваме инстанция на създадения клас като параметър за инстанция на клас Thread – така имаме доста по-голяма гъвкавост за промени и по-голям контрол. Затова

ще демонстрираме първият метод само като функционалност, но ще наблегнем на вторият.

Интерфейсът Runnable сам по себе си не е нещо интересно. Той съдържа само и единствено дефиниция за съществуване на метод „public void run();“. С други думи думите „implements Runnable“ не правят нищо друго освен да ви задължат да дефинирате нестатичен метод „run()“. Така все пак стигаме до заключението, че истинската работа с нишките се води от клас Thread. За да сме пределно ясни трябва да кажем, че самият клас Thread имплементира Runnable интерфейс!.

За стартиране на Thread се използва метод „start()“. Ето как би изглеждала нашата първа програма имплементираща нишки. В случая показваме двата метода за стартиране на нишка, които имплементират една и съща функционалност:

```
public class myfirstprogram{
 public static void main(String[] args){
   // Създаване на нишка с подаден обект имплементиращ Runnable:
   Thread T1 = new Thread(new RunnableExample());
   T1.start();

   // Създаване на нишка, чрез наследник на Thread:
   ThreadExtendExample T2 = new ThreadExtendExample();
   T2.start();
 }
}

class RunnableExample implements Runnable {
  public void run() {
    System.out.println("Аз съм нишка!");
  }
}

class ThreadExtendExample extends Thread {
    public void run() {
        System.out.println("Аз също съм нишка!");
    }
}
```

За да демонстрираме функционалността за „едновременно изпълнение“, нека напишем следната програма:

```
public class myfirstprogram{
 public static void main(String[] args){
   Thread T1 = new Thread(new RunnableExample("T1"));
   Thread T2 = new Thread(new RunnableExample("T2"));
   T1.start();
   T2.start();
 }
}
```

```
class RunnableExample implements Runnable {
  private String threadName;
  public RunnableExample(String threadName){
    this.threadName = threadName;
  }
  public void run() {
    for (int i=0; i<10; i++){
      System.out.print(this.threadName+":"+i+"; ");
    }
  }
}
```

Едно примерно изпълнение би било следното:

```
T1:0; T1:1; T1:2; T1:3; T1:4; T1:5; T2:0; T2:1; T2:2; T1:6;
T1:7; T1:8; T1:9; T2:3; T2:4; T2:5; T2:6; T2:7; T2:8; T2:9;
```

Понеже операциите са сравнително прости и изключително бързи (отпечатване на число на екрана), то първите няколко отпечатани числа на екрана са от T1, а последните от T2 (както по реда на изпълнение на методите start()). Все пак обаче в средата на изходните данни се вижда ясно, че двете нишки са се „засичали“ една друга, т.е. T2.start() не е изчакало T1.start() да завърши.

Клас Thread съдържа важни функции за контролиране на нишките. Ще разгледаме трите основни от тях поотделно:

1. Thread.sleep(милисекунди): „приспива“ текущата нишка и дава шанс на другите нишки да работят през това време:

```
public class myfirstprogram{
 public static void main(String[] args){
   Thread T1 = new Thread(new RunnableExample("T1"));
   Thread T2 = new Thread(new RunnableExample("T2"));
   T1.start();
   T2.start();
 }
}

class RunnableExample implements Runnable {
  private String threadName;
  public RunnableExample(String threadName){
    this.threadName = threadName;
  }
  public void run() {
    for (int i=0; i<5; i++){
      try{
        Thread.sleep(1000);
        System.out.print(this.threadName+":"+i+"; ");
      }
      catch (InterruptedException e){
        return;
```

```
      }
    }
  }
}
```

В този пример караме нишката да отпечатва съобщенията през интервали от 1 секунда. Реализирано по този начин ще видим, че изходът е много по-равномерен:

```
T1:0; T2:0; T1:1; T2:1; T1:2; T2:2; T1:3; T2:3; T2:4; T1:4;
```

„Приспиването на нишки“ се използва често, за да се дефинира приоритет на едни нишки спрямо други. Обикновено не бихме искали една по-маловажна подпрограма да може да отнеме прекалено много ресурси и по този начин да забавя работата на всички останали. Вземете за пример сайт за обработка на видео в който има два вида потребители – такива, които гледат видео клипове и такива, които ги качват и сървъра в последствие в отделна нишка преработва в удобен файлов формат. Определено „гледането на клипове“ са приоритетни операции спрямо преработката на клипове (тя може да се забави в зависимост от натовареността на системата). Затова можем да „приспиваме“ нишките за преработка на видео за малки периоди от време и по този начин да даваме „повече въздух“ на тези, която изпращат клиповете до потрибителите за гледане.

2. Thread.interrupt(): Изключението от тип InterruptedException вече подсказа, че нишките могат да се прекъсват. По принцип в старите версии на Java съществува метод Thread.stop(), но силно препоръчваме да го забравите веднага (ако работите с по-нова версия на Java, той вече не може да се използва тъй или иначе). Остарелия метод Thread.stop() прекъсваше нишката моментално. Досещате се, че по този начин можем лесно да загубим данни, да оставим файлове отворени и всякакви други поразии на софтуера. Вместо това по-добрият метод е „да кажем на нишката, че трябва да прекъсне“, тя да си довърши работата и да се спре сама. Можете веднага да си направите аналогия с разликата нормално затваряне на програма и „убиване на процеса“ чрез операционната система.

Когато извикаме метод „Thread.interrupt()“, то ние именно „съобщаваме“ на нишката, че трябва да спре. После в самата нишка ние трябва да имаме метод за прихващане на това събитие. Ако вътре в самата нишка използваме Object.wait(), Thread.join() или Thread.sleep() и ние извикаме прекъсване, то тези методи „виждат“ прекъсването и автоматично „хвърлят“ InterruptedException. Именно затова когато извиквахме „Thread.sleep()“ в миналия пример ние трябваше да прихванем това изключение. В него просто извикахме „return“ и приключихме изпълнението на нишката. Нека дадем един пример:

```
public class myfirstprogram{
 public static void main(String[] args){
   Thread T1 = new Thread(new RunnableExample("T1"));
   T1.start();

   try{
     Thread.sleep(500);
```

```
    }
    catch(InterruptedException e){}

    System.out.println("Interrupting thread...");
    T1.interrupt();
  }
}

class RunnableExample implements Runnable {
  private String threadName;
  public RunnableExample(String threadName){
    this.threadName = threadName;
  }
  public void run() {
    for (int i=0; i<50; i++){
      try{
        Thread.sleep(100);
        System.out.println(this.threadName+":"+i+"; ");
      }
      catch (InterruptedException e){
        System.out.println("I was interrupted!");
        return;
      }
    }
  }
}
```

Резултатът от изпълнението ще бъде:

```
T1:0;
T1:1;
T1:2;
T1:3;
Interrupting thread...
I was interrupted!
```

След малко практика ще забележите обаче, че прекъсването никак не настъпва моментално. Вземете например този пример:

```
public class myfirstprogram{
 public static void main(String[] args){
    Thread T1 = new Thread(new RunnableExample("T1"));
    T1.start();

    try{
      Thread.sleep(50);
    }
    catch(InterruptedException e){}

    System.out.println("Interrupting thread");
    T1.interrupt();
```

```
 }
}

class RunnableExample implements Runnable {
  private String threadName;
  public RunnableExample(String threadName){
    this.threadName = threadName;
  }
  public void run() {
    while(true){
      try{
        Thread.sleep(1);
        double time = System.currentTimeMillis();
        while(System.currentTimeMillis()-time < 5000){
          System.out.print(".");
        }
        System.out.println();
      }
      catch (InterruptedException e){
        System.out.println("I was interrupted!");
        return;
      }
    }
  }
}
```

Нишката изписва точки на екрана в продължение на 5 секунди, след това заспива за 1 милисекунда и продължава да изписва точки на екрана за нови 5 секунди (и така до безкрайност, защото всичко се намира в цикъл while(true)). В главната програма ние стартираме тази нишка, изчакваме 5 милисекунди и я прекъсваме, чрез T1.interrupt(). Ще забележите обаче, че нишката ще продължи да изписва точки на екрана още няколко секунди преди да спре. Това е защото най-вероятно прекъсването ще съвпадне с изпълнението на тялото на вложения while цикъл, който още не е завършил. Чак когато се достигне до Thread.sleep(1) ще бъде хвърлен InterruptedException, който ще бъде прихванат и нишката ще спре според инструкциите в секцията catch.

Какво обаче да правим ако не използваме Thread.sleep() или други готови методи, които следят за interrupt или пък сме започнали алгоритъм, който ще изиска много време преди да се „върнем обратно“ към някоя от тези методи? Естествено се досещаме, че сами можем да следим дали нишката е прекъсната. Ако използваме наследяване на клас Thread, то ни върши работа метод „interrupted()“ връщащ boolean. В нашата имплементация можем да я реализираме сами:

```
public class myfirstprogram{
 public static void main(String[] args){
   RunnableExample T1Obj = new RunnableExample("T1");
   Thread T1Thread = new Thread(T1Obj);
   T1Thread.start();
```

```
    try{
      Thread.sleep(50);
    }
    catch(InterruptedException e){}

    System.out.println("Interrupting thread");
    T1Obj.interrupt();
    T1Thread.interrupt();
  }
}

class RunnableExample implements Runnable {
  private boolean isInterrupted;
  String threadName;
  public RunnableExample(String threadName){
    this.threadName = threadName;
    this.isInterrupted = false;
  }
  public void interrupt(){
    this.isInterrupted = true;
  }
  private void stopMsg(){
    System.out.println("I was interrupted!");
  }
  public void run() {
    while(this.isInterrupted == false){
      try{
        Thread.sleep(1);
        double time = System.currentTimeMillis();
        while(System.currentTimeMillis()-time   <   5000   &&
this.isInterrupted == false){
          System.out.print(".");
        }
        System.out.println();
      }
      catch (InterruptedException e){
        this.stopMsg();
        return;
      }
    }
    stopMsg();
  }
}
```

Казано накратко ние създаваме една променлива, която ни съобщава дали нишката е прекъсната или не. После в самата програма ние трябва периодично да проверяваме дали случайно тя не е променена.

3. Thread.join() – не-статичен метод на обект от тип Thread. Когато бъде извикан, то извикващата нишка трябва да изчака изпълнението на притежателя на метода. По този начин нишките се изчакват една друга. Нека се върнем към един по-стар пример, но този път го преработим така, че класът RunnableExample сам в себе си съдържа нишката си (всъщност много често използвана практика):

```
public class myfirstprogram{
 public static void main(String[] args){
   RunnableExample T1 = new RunnableExample("T1");
   RunnableExample T2 = new RunnableExample("T2");
   // Стартираме нишките като член-променливи на класовете
   T1.T.start();
   T2.T.start();
   // Караме главната програма да "заспи" докато свършат нишките
   try{
     T1.T.join();
     T2.T.join();
   }
   catch(InterruptedException e){
     System.out.println("What? Interrupt should NOT happen at all!");
   }
   System.out.println("Main finished!");
 }
}

class RunnableExample implements Runnable {
  private String threadName;
  // Вече нишката е член-променлива на класа
  Thread T;
  public RunnableExample(String threadName){
    this.threadName = threadName;
    // Инициализираме нишката вътре в конструктора
    this.T = new Thread(this, this.threadName);
  }
  public void run() {
    for (int i=0; i<10; i++){
      try{
        // Нарочно слагаме произволно време за "заспиване"
        Thread.sleep((int)Math.random()*1000);
        System.out.print(this.threadName+":"+i+"; ");
      }
      catch (InterruptedException e){
        return;
      }
    }
```

```
    System.out.println("\nThread              "+this.threadName+
"finished!");
  }
}
```

Ще видите, че main() метода ще изчака докато и двете нишки са свършили и чак тогава ще изпише съобщението „Main finished!“.

Синхронизация на нишки

Когато имаме многонишково приложение, то ние много често работим и със споделени ресурси. Нека демонстрираме с един пример – имаме масив с наредени числа. Имаме две нишки – такава, която променя числата с произволни и такава, която сортира числата по големина:

```
public class myfirstprogram{
 public static void main(String[] args){
   ArrayClass arr = new ArrayClass();
   ChangeThread c = new ChangeThread("Change Thread", arr);
   SortThread s = new SortThread("Sort Thread", arr);
   c.T.start();
   s.T.start();
   try{
     c.T.join();
     s.T.join();
   }
   catch(java.lang.InterruptedException e){}
   arr.showArray();
 }
}

class ArrayClass{
  int[] arr;
  public ArrayClass(){
    this.arr = new int[200];
    for(int i=this.arr.length-1; i>=0; i--){
      this.arr[i] = i;
    }
  }
  public void changeArray(){
    for(int i=0; i<this.arr.length; i++){
      this.arr[i] = (int)Math.round(Math.random()*100);
    }
    System.out.println("Change finished");
  }
  public void sortArray(){
    java.util.Arrays.sort(this.arr);
    System.out.println("Sort finished");
  }
```

```
  public void showArray(){
    for(int i: this.arr){
      System.out.print(i+" ");
    }
  }
}

class SortThread implements Runnable {
  private String threadName;
  ArrayClass arr;
  Thread T;
  public SortThread(String threadName, ArrayClass arr){
    this.threadName = threadName;
    this.T = new Thread(this, this.threadName);
    this.arr = arr;
  }
  public void run(){
    arr.sortArray();
  }
}

class ChangeThread implements Runnable {
  private String threadName;
  ArrayClass arr;
  Thread T;
  public ChangeThread(String threadName, ArrayClass arr){
    this.threadName = threadName;
    this.arr = arr;
    this.T = new Thread(this, this.threadName);
  }
  public void run() {
    arr.changeArray();
  }
}
```

При изпълнение на програмата ще забележите, че нито числата са подредени произволно, нито масива е сортиран напълно. Ще има няколко поредици от сортирани числа – нещо, което определено нито една от нишките не е пожелала.

За да поправим този проблем, когато една нишка „бърка“ в данните на друга, то използваме т.нар. „синхронизирани методи“. Въвежда се функционалност подобна на тази при трансакциите при бази от данни – достъпваните данни се „заключват“, променят се и се „отключват“ за достъп от други:

```
public class myfirstprogram{
 public static void main(String[] args){
   ArrayClass arr = new ArrayClass();
   ChangeThread c = new ChangeThread("Change Thread", arr);
   SortThread s = new SortThread("Sort Thread", arr);
```

```
   c.T.start();
   s.T.start();
   try{
     c.T.join();
     s.T.join();
   }
   catch(java.lang.InterruptedException e){}
   arr.showArray();
 }
}

class ArrayClass{
  int[] arr;
  public ArrayClass(){
    this.arr = new int[200];
    for(int i=this.arr.length-1; i>=0; i--){
      this.arr[i] = i;
    }
  }
 public synchronized void changeArray(){
    for(int i=0; i<this.arr.length; i++){
      this.arr[i] = (int)Math.round(Math.random()*100);
    }
    System.out.println("Change finished");
  }
 public synchronized void sortArray(){
    java.util.Arrays.sort(this.arr);
    System.out.println("Sort finished");
  }
  public void showArray(){
    for(int i: this.arr){
      System.out.print(i+" ");
    }
  }
}

class SortThread implements Runnable {
  private String threadName;
  ArrayClass arr;
  Thread T;
  public SortThread(String threadName, ArrayClass arr){
    this.threadName = threadName;
    this.T = new Thread(this, this.threadName);
    this.arr = arr;
  }
  public void run(){
    arr.sortArray();
  }
```

```
}

class ChangeThread implements Runnable {
  private String threadName;
  ArrayClass arr;
  Thread T;
  public ChangeThread(String threadName, ArrayClass arr){
    this.threadName = threadName;
    this.arr = arr;
    this.T = new Thread(this, this.threadName);
  }
  public void run() {
    arr.changeArray();
  }
}
```

Когато една нишка извика „синхронизиран“ метод от един обект, то всички други нишки, които в същия момент извикат същия или друг синхронизиран метод от същия обект „заспиват“ и изчакват изпълнението си в опашка. В този смисъл ако първо се стартира ChangeThread, то той ще извика метод changeArray(). Ако стартираме в същия момент SortThread, то той ще извика функцията sortArray(). Тъй като и двете извикани функции са синхронизирани (с ключова дума synchronized), то SortThread ще „заспи“ и ще изчака ChangeThread да си свърши работата. Едва след това метод sortArray() ще бъде стартиран.

Единствените методи, които не могат да бъдат синхронизирани са конструкторите. При тях естествено такава операция е напълно безсмислена – не е възможно две нишки да „създават“ един и същи обект едновременно.

В предишни статии ние споменахме за два обекта, които се различаваха по това как са реализирани техните методи. Това бяха StringBuilder (притежаващ не-синхронизирани методи) и StringBuffer (чийто методи са синхронизирани). Когато пишем подобни класове е хубаво винаги да преценяваме дали те ще бъдат използвани в многонишкови приложения или не. Синхронизирането трябва да се изпълнява много внимателно – в противен случай е напълно възможно при да се появят трудни за локализиране „бъгове“ в последствие при реална експлоатация на софтуера.

Понякога обаче ни се налага да не правим синхронизация на абсолютно целия метод, а само на фрагмент от него. Така на практика оптимизираме програмите, като заключваме нишките за по-кратко време. Синтаксисът е следният:

```
Оператори 1;
synchronized(<име на обект>){
   Оператори 2;
}
Оператори 3;
```

По този начин по време на изпълнението на групата „Оператори 2″ обектът дефиниран в блока ще бъде заключен за другите нишки. Чак след излизане от

блока те ще бъдат отключени. От предишния пример бихме могли да постигнем аналогична функционалност ако променим методите changeArray и sortArray по следния начин:

```
public void changeArray(){
 synchronized(this){
    for(int i=0; i<this.arr.length; i++){
      this.arr[i] = (int)Math.round(Math.random()*100);
    }
    System.out.println("Change finished");
 }
public void sortArray(){
 synchronized(this){
    java.util.Arrays.sort(this.arr);
 }
 System.out.println("Sort finished");
}
```

Виждаме, че вече сме оградили само и единствено блокът, където се прави промяна на данните. Извикването на System.out.println() не променя никакви данни и поради тази причина няма нужда да бъде изчакван от другите нишки.

Още повече – не е задължително блоковете да заключват всички обекти от текущия клас (в горния пример заключихме за синхронизация this). Можем да заключваме само и единствено обектът, с който работим, а именно – arr.this:

```
  public void changeArray(){
    synchronized(this.arr){
      for(int i=0; i<this.arr.length; i++){
        this.arr[i] = (int)Math.round(Math.random()*100);
      }
    }
    System.out.println("Change finished");
  }
  public void sortArray(){
    synchronized(this.arr){
      java.util.Arrays.sort(this.arr);
    }
    System.out.println("Sort finished");
  }
```

По този начин е съвсем възможно друга нишка да достъпва и променя други „член-обекти“ от текущия клас.