Laborskript Verteilte Systeme

Nebenläufigkeit in Java

Prof. Dr. Oliver Haase

1 Threads

Java bietet zwei verschiedene Arten an, Threads zu erzeugen und zu starten:

- *Entweder durch Erweitern der Klasse* `Thread`, oder
- *durch Implementierung der Schnittstelle* `Runnable`: Diese Variante funktioniert auch dann, wenn die eigene Threadklasse eine andere Superklasse als `Thread` benötigt.

1.1 Erweitern der Klasse `Thread`

1. Die neue Klasse `MyThread` muss als Subklasse von `Thread` deklariert werden:

```
public class MyThread extends Thread {
   ...
}
```

2. Die Klasse `MyThread` muss die Methode

```
public void run()
```

der Superklasse `Thread` überschreiben. Diese wird von der virtuellen Java-Maschine automatisch ausgeführt, sobald der entsprechende Thread gestartet wird und muss deshalb die wesentliche Logik der eigenen Threadklasse enthalten.

3. Thread starten:

```
Thread instance = new MyThread();
instance.start();
```

Merke: Methode `run()` programmieren, aber zum Starten `start()`–Methode aufrufen.

1.2 Implementieren der Schnittstelle `Runnable`

1. Die neue Klasse `MyRunnable` muss die Schnittstelle `Runnable` implementieren:

```
public class MyRunnable implements Runnable {
    ...
}
```

2. Die Schnittstelle `Runnable` enthält als einzige Methode die Methode

```
public void run()
```

   `MyRunnable` muss in dieser ihre eigentlichen Funktionalität implementieren.

3. Thread starten:

```
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Merke: Methode `run()` programmieren, aber zum Starten `start()`–Methode aufrufen.

2 Executor-Framework

Bei Verwendung des Executor-Framework, das integraler Bestandteil von Java ist, müssen Thread-Pools nicht mehr per Hand programmiert werden. Stattdessen gibt es einen sogenannten `ExecutorService`, der intern einen Thread-Pool verwaltet und dem man lediglich sogenannte `Tasks` zur Ausführung übergibt. Der `ExecutorService` reiht die Task selbständig in eine Task-Warteschlange ein, aus der sie von frei werdenden Threads des Thread-Pools abgearbeitet werden.

Ein Task muss die Schnittstelle `Runnable` (siehe Abschnitt 1.2) implementieren, ihre gewünschte Funktionalität wird in der `run`-Methode programmiert. (*Es gibt außerdem die Möglichkeit, dass die Tasks statt der `Runnable`-Schnittstelle die `Callable`-Schnittstelle implementieren, deren `call`-Methode es erlaubt, Ergebnisse zurückzuliefern und Exceptions zu werfen. Diese Möglichkeit wird hier aber nicht verwendet und deshalb nicht weiter betrachtet.*)

Einen `ExecutorService` besorgt man sich mit Hilfe einer der statischen Fabrikmethoden der Klasse `Executors`. Die in diesem Kontext wichtigsten Varianten sind die beiden folgenden Fabrikmethoden:

`public static ExecutorService newCachedThreadPool()`: erzeugt einen `ExecutorService`, der einen dynamisch wachsenden Thread-Pool verwaltet.

`public static ExecutorService newFixedThreadPool(int nThreads)`: erzeugt einen `ExecutorService`, der einen statischen Thread-Pool der Größe `nThreads` verwaltet.

Nach der Erzeugung können dem `ExecutorService` mit Hilfe wiederholter Aufrufe der Methode `public void execute(Runnable task)` beliebig viele Tasks zur Ausführung übergeben werden. Nach dem Übergeben der letzten Task muss der `ExecutorService` mit Hilfe der Methode `public void shutdown()` beendet werden, ansonsten verhindert das Laufen des `ExecutorService` das Beenden des Programms. Hier ein beispielhaftes Code-Fragment:

```
ExecutorService executor =
    Executors.newFixedThreadPool(NUM_THREADS);

while ( !done ) {
   executor.execute(new Runnable() {
      public void run() {
         // Implementierung der Task
      }
   });
}
executor.shutdown();
```

oder unter Verwendung eines Lambda-Ausdrucks:

```
ExecutorService executor =
    Executors.newFixedThreadPool(NUM_THREADS);

while ( !done ) {
   executor.execute(() -> {
         // Implementierung der Task
   });
}
executor.shutdown();
```

3 Synchronisierung nebenläufiger Zugriffe auf gemeinsame Daten

In Java müssen parallele, konkurrierende Zugriffe auf gemeinsame Daten von mehreren Threads aus zwei Gründen synchronisiert werden:

1. Um Änderungen eines Threads für andere Threads sichtbar zu machen
2. Um Inkonsistenzen (Parallelitätsanomalien) zu vermeiden

3.1 Sichtbarmachen von Änderungen

Das Java *Memory Model* ist auf single-threaded Applikationen ausgerichtet und kann im multi-threaded Fall unerwartete Effekte erzeugen. Als Beispiel dient das folgende einfache Programm:

```
public class StopThread {
   private static boolean stopRequested;
```

```
    public static void main(String[] args)
            throws InterruptedException {
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                System.out.println("start");
                int i = 0;
                while ( !stopRequested ) {
                    i++;
                }
                System.out.println("done");
            }
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

Man würde erwarten, dass das Programm zuerst `start` ausgibt, nach ca. einer Sekunde `done` und dann terminiert. Ausgeführt in einer Oracle-JVM gibt das Programm aber nur `start` aus und terminiert nicht. Das liegt daran, dass das Java *Memory Model* für nicht-volatile Variablen nicht garantiert, wann – bzw. ob überhaupt – die Änderungen durch einen Thread von anderen gesehen werden können! Zur Optimierung kann der Compiler deshalb die Schleife

```
while ( !stopRequested ) {
    i++;
}
```

abändern in

```
if ( !stopRequested ) {
    while ( true ) {
        i++;
    }
}
```

Man beachte, dass das im single-threaded Fall eine gültige Optimierungsmaßnahme ist: Da die Variable `stopRequested` im Schleifenrumpf nicht geändert wird, genügt es, sie vor dem ersten Schleifendurchlauf nur einmal zu lesen. Im multi-threaded Fall führt diese Änderung offensichtlich zu einem fehlerhaften Programm (jedenfalls fehlerhaft im Sinne des Programmierers).

Wenn man erzwingen möchte, dass eine Änderung der Variablen `stopRequested` für andere Threads sofort sichtbar wird (was u.a. dem Compiler die obige Optimierung verbietet), dann muss man `stopRequested` als `volatile` deklarieren, so wie in der folgenden modifizierten Version des Programms:

```
public class StopThread {
    private static volatile boolean stopRequested;

    public static void main(String[] args)
            throws InterruptedException {
        Thread backgroundThread = new Thread(new Runnable() {
```

```
            public void run() {
                System.out.println("start");
                int i = 0;
                while ( !stopRequested ) {
                    i++;
                }
                System.out.println("done");
            }
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

In einer verteilten Anwendung kommt die hier beschriebene Situation oft auf der Serverseite vor, wo das Entgegennehmen neuer Anfragen in einer Schleife passiert, die so lange durchlaufen wird, bis eine ein anderer Thread eine boolsche Variable setzt, die das Beenden des Servers anzeigt.

> *Merke*: Das Sichtbarmachen vom Änderungen mit `volatile` verhindert keine Inkonsistenzen durch konkurrierenden Zugriff. Es stellt nur sicher, dass schreibender Zugriff sofort in allen Threads sichtbar wird.

3.2 Synchronisierung konkurrierender Zugriffe

Wenn zwei oder mehr Threads gleichzeitig auf eine gemeinsame Ressource (ein gemeinsames Objekt) zugreifen, kann es zu *Parallelitätsanomalien* kommen. Dies kann passieren, wenn Modifikationen durch einen Thread nicht in einer atomaren Operation durchgeführt werden können. Beispiele dafür sind

- das Schreiben von 64-Bit-Variablen (`long` oder `double`);
- Der Inkrementoperator;
- Modifikationen an mehreren, voneinander abhängigen Variablen.

(*Das Schreiben einer 32-Bit-Variablen hingegen kann in einer atomaren Operation durchgeführt werden, siehe auch Abschnitt 3.1.*)

Die übliche Technik, solche Anomalien zu verhindern besteht darin, die konkurrierenden Zugriffe auf gemeinsame Daten zu serialisieren, d.h. den gleichzeitigen Zugriff zu verhindern. Damit wird der Parallelitätsgrad des nebenläufigen Programms gezielt verringert. Das technisches Mittel zur Serialisierung der Zugriffe in Java sind *Objektsperren*. In Java besitzt jedes Objekt eine *inhärente* ("innewohnende") Sperre (*Monitorsperre*, *Monitor*). Es gibt zwei Arten, diese Sperren einzusetzen, und zwar mit Hilfe von

1. *synchronisierten Methoden* und
2. *synchronisierten Anweisungen.*

Synchronisierte Methoden werden mit einem `synchronized`-Modifizierer markiert. Wenn ein Thread t_1 eine synchronisierte Instanzmethode m ausführt, hält er die inhärente Objektsperre des zu der Methode gehörigen Objektes. Solange t_1 diese Sperre hält, kann kein anderer Thread eine synchronisierte Methode (m oder eine andere synchronisierte Methode) am selben Objekt ausführen; dies geht erst, nachdem t_1 m wieder verlassen und die Sperre freigegeben hat. Nichtsynchronisierte Methoden am selben Objekt können weiterhin ohne Einschränkung parallel ausgeführt werden.

Um zu verhindern, dass lesende Operationen ungültige Zwischenzustände der geteilten Ressource sehen können, müssen *sowohl Lese- als auch Schreibmethoden synchronisiert werden.* Beispiel:

```
public class SyncSample {
   private int nextSerialNumber = 0;
   private boolean boolValue = false;

   public synchronized int generateSerialNumber() {
      return nextSerialNumber++;
   }

   public synchronized int getCurrentNumber() {
      return nextSerialNumber;
   }

   public synchronized boolean generateBoolean() {
      boolValue = !boolValue;
      return boolValue;
   }

   public synchronized boolean getCurrentBoolean() {
      return boolValue;
   }
}
```

Synchronisierte Anweisungen sind flexibler als synchronisierte Methoden, erlauben das feingranularere Setzen von Sperren, dafür sind sie etwas aufwendiger in der Verwendung. Bei synchronisierten Anweisungen spezifiziert man explizit das Objekt, dessen Sperre gehalten werden soll, sowie den Codeblock, (*kritischen Programmabschnitt*), der nach Erhalt der Sperre ausgeführt werden soll. Beispiel:

```
public class SyncSample2 {
   private Integer nextSerialNumber = 0;
   private Boolean boolValue = false;

   public int generateSerialNumber() {
      synchronized ( this.nextSerialNumber ) {
         return nextSerialNumber++;
      }
   }

   public int getCurrentNumber() {
      synchronized ( this.nextSerialNumber ) {
```

```
            return nextSerialNumber;
        }
    }

    public boolean generateBoolean() {
        synchronized ( this.boolValue ) {
            boolValue = !boolValue;
            return boolValue;
        }
    }

    public boolean getCurrentBoolean() {
        synchronized ( this. boolValue ) {
            return boolValue;
        }
    }
}
```

Im obigen Beispiel werden für die Instanzvariablen `nextSerialValue` und `boolValue` die Wrapperklassen `Integer` und `Boolean` statt der skalaren Typen `int` und `boolean` verwendet, weil die Instanzvariablen damit selbst Objekte mit inhärenten Sperren sind. In diesem Code-Beispiel wird, im Gegensatz zum vorherigen Beispiel, nur die jeweils betroffene Instanzvariable gesperrt; dadurch ist gleichzeitiges Arbeiten auf `nextSerialNumber` und `boolValue` möglich.

3.3 Lese-Schreib-Sperren für erhöhten Parallelitätsgrad

Häufig hat man die Situation, dass es auf gemeinsame Daten viele lesende und nur wenige schreibende Zugriffe gibt. Wenn man nun mit Hilfe inhärenter Sperren synchronisiert, dann schließen sich auch lesende Zugriffe gegenseitig aus, was eigentlich nicht nötig wäre. Um das zu vermeiden, kann man Lese-Schreib-Sperren einsetzen, die beliebige viele parallele lesende, aber nur einen gleichzeitigen schreibenden Zugriff erlauben. In Java gibt es hierfür die Schnittstelle `ReadWriteLock` und die implementierende Klasse `ReentrantReadWriteLock`. Die Benutzung ist im folgenden Code-Fragment beispielhaft skizziert:

```
Object sharedData = SomeData();
ReadWriteLock lock = new ReentrantReadWriteLock();

public void readAndUseData() {
    lock.readLock().lock();
    if ( sharedData.getSomeField() )
        ...
    lock.readLock().unlock();
}

public void writeData() {
    lock.writeLock().lock();
    sharedData.setSomeField(someValue);
    lock.writeLock().unlock();
}
```