Mitschrift Nebenläufige Programmierung

Günther Wutz

Einstieg in Streams (mit und ohne Parallelisierung)

Um aus einem Integer-Array einen Stream zu generieren, muss man über die **Arrays** Klasse gehen

```
1 int[] zahlen = {1, 2, 3, 4};
2 IntStream is = Arrays.stream(zahlen);
```

Aufgabe 1 - IntStream

Gegeben `int[] zahlen = {3, 5, 1, 3, 7, 29, 33, 49, 5, 1, 1, 2, 3};`

Summe IntStream

```
1 IntStream is = Arrays.stream(zahlen);
2 System.out.Println(is.sum());
```

```
1 Output:
2 142
```

Durschnitt IntStream

```
1 IntStream is = Arrays.stream(zahlen);
2 System.out.Println(is.average().getAsDouble());
```

```
1 Output:
2 10.923076923076923
```

Maximum IntStream

```
IntStream is = Arrays.stream(zahlen);
System.out.Println(is.max());
```

```
Output:
49
```

Histogramm IntStream

```
IntStream s = Arrays.stream(zahlen);
Map<Integer, Long> histogramm =
    s.boxed().collect(Collectors.groupingBy(
        Integer::intValue, Collectors.counting()));
System.out.println(histogramm);
```

```
Output:
{49=1, 33=1, 1=3, 2=1, 3=3, 5=2, 7=1, 29=1}
```

Ausgabe in der Form [3, 5, 3, . . . , 2, 3]

```
IntStream s = Arrays.stream(zahlen);
String str = "["
             + s.mapToObj(i -> String.valueOf(i)).collect(Collectors.joining(", "))
             + "]";
System.out.println(str);
```

```
Output:
[3, 5, 1, 3, 7, 29, 33, 49, 5, 1, 1, 2, 3]
```

Aufgabe 2 - Stream

Summe

```
List<Integer> zahlen =
    Arrays.asList(new Integer[] { 3, 5, 1, 3, 7, 29, 33, 49, 5, 1, 1, 2, 3 });
System.out.println(zahlen.stream()
                  .collect(Collectors.summingInt(Integer::intValue)));
```

Durchschnitt

```
List<Integer> zahlen =
    Arrays.asList(new Integer[] { 3, 5, 1, 3, 7, 29, 33, 49, 5, 1, 1, 2, 3 });
System.out.println(zahlen.stream()
                  .collect(Collectors.averagingInt(Integer::intValue)));
```

Maximum

```
List<Integer> zahlen =
    Arrays.asList(new Integer[] { 3, 5, 1, 3, 7, 29, 33, 49, 5, 1, 1, 2, 3 });
System.out.println(zahlen.stream()
                  .max(Integer::compareTo).get());
```

Achtung! `stream().max(...)` liefert ein `Optional<Integer>` weswegen man eigentlich eine Fallunterscheidung wie im folgenden Beispiel machen müsste (aus Einfachheitsgründen verzichte ich darauf)

```
List<Integer> zahlen =
    Arrays.asList(new Integer[] { 3, 5, 1, 3, 7, 29, 33, 49, 5, 1, 1, 2, 3 });
Optional<Integer> optInt = zahlen.stream().max(Integer::compareTo);
if(optInt.isPresent()) System.out.println(optInt.get());
```

Histogramm

```
List<Integer> zahlen =
    Arrays.asList(new Integer[] { 3, 5, 1, 3, 7, 29, 33, 49, 5, 1, 1, 2, 3 });
System.out.println(zahlen.stream()
                  .collect(Collectors
                  .groupingBy(Integer::intValue, Collectors.counting())));
```

Ausgabe

```
List<Integer> zahlen =
    Arrays.asList(new Integer[] { 3, 5, 1, 3, 7, 29, 33, 49, 5, 1, 1, 2, 3 });
System.out.println("["
                    + zahlen.stream()
                      .map(i -> i.toString())
                      .collect(Collectors.joining(", "))
                    + "]");
```

Parallelisieren mit Streams

Gegeben ist die folgende recht ineffiziente Primzahl-Testmethode für ganze Zahlen >= 2. Diese Methode ist ein Prädikat für ganze Zahlen im Sinne von Java 8.

```
static boolean isPrime(int zahl) {
    int teiler = 2;
    while (zahl % teiler != 0)
        teiler++;
    if (zahl == teiler)
        return true;
    return false;
}
```

Aufgabe 1

Geben Sie die Primzahlen im Bereich 1..1000 aus

```
System.out.println(IntStream
            .rangeClosed(1, 1000)
            .filter(MyIntStream::isPrime)
            .mapToObj(i -> String.valueOf(i))
            .collect(Collectors.joining(", ")));
```

Aufgabe 2

Geben Sie die Primzahlen im Bereich 1..100000 aus. Nützen Sie alle Kerne Ihres Rechners zur Berechnung!

```
System.out.println(IntStream
            .rangeClosed(1, 100000)
            .parallel()
            .filter(MyIntStream::isPrime)
            .mapToObj(i -> String.valueOf(i))
            .collect(Collectors.joining(", ")));
```

Aufgabe 3 (Optional)

Wenn Ihnen die obige Methode zur Ermittlung von Primzahlen zu primitiv ist, geben Sie eine effizientere Variante an.

Ein einfacher Spliterator

In einer der Aufgaben werden `Path`-Objekte (d. h. Objekte einer Klasse, die die `Path`-Schnittstelle implementieren) von einem Erzeuger in eine `BlockingQueue` gestellt.

Grundlagen

Ein Spliterator ist ein **Interface** der Java API

```
1  public interface Spliterator<T> {
2      public abstract boolean tryAdvance(Consumer<? super T>);
3      public void forEachRemaining(Consumer<? super T>);
4      public abstract Spliterator<T> trySplit();
5      public abstract long estimateSize();
6      public long getExactSizeIfKnown();
7      public Comparator<? super T> getComparator();
8  }
```

Aufgabe 1: einfache Erzeuger-Verbraucher-Kopplung über einen Stream

Eine `BlockingQueue` ist eine `Collection`. Damit erhalten wir einen Stream im Sinne von Java 8 zum Durchlaufen der Resultate des o. a. Erzeugers. Verwenden Sie diesen Stream in einem bzw. zwei Verbraucher(n) zum Auslesen aller `Path`-Objekte.

Starten Sie einen Erzeuger und 1-2 Verbraucher. Das Hauptprogramm soll sich erst beenden, wenn sich alle Erzeuger bzw. Verbraucher beendet haben.

Wie verhält sich ihr Programm?

Um nochmal zu verdeutlichen, dass Programm aus der vorherigen Aufgabe

```
1  public class Producer extends SimpleFileVisitor<Path> implements Runnable {
2      private final BlockingQueue<Path> queue;
3      private final int consumer;
4
5      public Producer(BlockingQueue<Path> queue, int consumer)
6      {
7          this.queue = queue;
8          this.consumer = consumer;
9      }
10
11     @Override
```

```
    public void run() {
        try {
            Path path = Paths.get("/home/gunibert/development");
            Files.walkFileTree(path, this);
            for(int i = 0; i < consumer; ++i) {
                Path poisonPill = Paths.get("wearedone");
                queue.put(poisonPill);
            }
        } catch(IOException | InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public FileVisitResult visitFile(
        Path file, BasicFileAttributes arg1) throws IOException {

        if(file.toString().matches(".*\\.(java|cpp|h)")) {
            try {
                queue.put(file);
            } catch(Exception e){
                e.printStackTrace();
            }
        }

        return FileVisitResult.CONTINUE;
    }
}
```

```
public class Consumer implements Runnable {
    private BlockingQueue<Path> queue;
    private boolean run = true;

    public Consumer(BlockingQueue<Path> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        while(run){

            try {
                Path p = (Path)queue.take();
                handlePath(p);
            } catch(Exception e){
                e.printStackTrace();
            }
        }
    }

    private void handlePath(Path p) {
```

```
        if(p.toString().equals("wearedone")) {
            run = false;
        } else {
            System.out.printf("File: %s\n", p.getFileName());
        }
    }
}
```

```
public static void main(String[] args) {
    BlockingQueue<Path> queue = new LinkedBlockingQueue<Path>();
    Producer p = new Producer(queue, 1);
    Consumer c = new Consumer(queue);

    new Thread(c).start();
    new Thread(p).start();
}
```

Anstelle des `Consumers` sollen wir hier einen Stream zum Auslesen der `Path`-Objekte nutzen.

Ich hab das jetzt einmal mit diesem Codeschnippsel getestet und bin nicht sicher, ob ich damit das gewünschte Ergebnis bekomme.

```
queue.stream().map(path -> path.toString()).forEach(System.out::println);
```

Die Krux an der Sache ist, dass man zum Zeitpunkt t als Stream lediglich den Inhalt bekommt, der da ist. Nachdem dieser ausgelesen wurde, beendet sich das Programm (was nicht der eigentliche Nutzen sein sollte).

Aufgabe 2: ein selbst erstellter Spliterator

Geben Sie einen `Spliterator` für die o. a. `BlockingQueue` an. Sie können hierzu die Methode `stream` der Klasse `java.util.stream.StreamSupport` benützen. Stellen Sie sicher, dass alle vom Erzeuger gelieferten Objekte auch vom Verbraucher gelesen werden.

Aufgaben aus Java Magazine von Oracle

```
import java.util.Random;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class JavaMagazine2014Sept {
    static final int MAXSEEDVALUE = 200_000;
```

```
7       static final int SEEDVALUE = new Random().nextInt(MAXSEEDVALUE);
8       static final int COUNT = 10;
9
10      public static void main(String[] args) {
11          System.out.println(IntStream.rangeClosed(SEEDVALUE+1,MAXSEEDVALUE)
12          .parallel()
13          .filter(i -> IntStream. range (2, i)
14          .filter(j-> i%j == 0)
15          .count() == 0)
16          .limit(COUNT)
17          .mapToObj(String::valueOf )
18          .collect(Collectors.joining("")));
19      }
20  }
```

This program finds the `COUNT` number of prime numbers that are greater than some random starting value. It runs slowly because its not making effective use of the Stream API. What change could be made that would result in a considerable speedup?

1. Replace the lambda expression `(j -> i % j == 0)` by an anonymous class implementing the `IntPredicate` interface
2. Move the `limit()` function before the outer `filter()`
3. Use `noneMatch()` instead of the inner `filter()`
4. Replace both `filter()` functions using the iteration of the Java Collections Framework.

Desktop-Suche

Gegeben ist ein GUI Rahmenprogramm (siehe con-ws14-blatt07.zip im eLearning)

Aufgabe 1

Auf Drückn der `Start`-Schaltfläche soll eine nebenläufige Suche gestartet werden. Teilergebnisse und das Endergebnis der Suche sollen angezeigt werden (im Hauptfenster). Benützen Sie dazu eine Ihrer Lösungen der Aufgabe 4.

Aufgabe 2

Nach dem Drücken des `Stop`-Schalters sollen alle gestarteten Aktivitäten so bald wie möglich beendet werden. Die laufende GUI soll nicht abgebrochen werden. `System.exit()` löst das Problem nicht.

Optimistisches Sperren

Aufgabe 1 - Sieht leicht aus, ist aber nicht ganz leicht

Versuchen Sie, einen Zähler in einem System mit Nebenläufigkeit für ganze Zahlen zu implementieren. `next()` soll beginnend ab 0 fortlaufend ganze Zahlen liefern. *Die Performance sollte besser sein, als bei einer Lösung mit einer* ***`synchronized next()`****-Funktion.*

```
1  class Counter {
2      private ... int counter;
3      public int next() { return counter++; } // Liefern und schalten
4  }
```

! Prinzipiell wird man eine Lösung bei den sog. optimistischen Sperren suchen. **Aber**: man kann wohl nur dann Lösungen mit kürzeren Antwortzeiten finden, wenn man die Phasen der Sperrung reduziert. Einen möglichen Ansatz zeigt D. Lea mit seinem Programm 'Spinlock' (siehe unten).

Optimistische Updates: [Lea99, 2.4.4.2]

Ein optimistisches Update hat 3 Phasen

1. Innerhalb einer Sperre: Kopieren des aktuellen Zustands
2. Außerhalb der Sperre: Kopie ändern
3. Innerhalb einer Sperre: Zurückschreiben, aber nur, falls sich der Zustand nicht geändert hat

Optimistische Update-Techniken Sperren den Zugriff auf Objekte nur kurz, insbesondere bei Multicore-Prozessoren.

```
1  class Optimistic { // Generic code sketch
2      private State state;
3
4      private synchronized State getState() {
5          return state;
6      }
7
8      private synchronized boolean commit(State assumed, State next) {
9          if (state == assumed) {
10             state = next;
11             return true;
12         } else {
13             return false;
14         }
```

```
15      }
16  }
```

Was tun, wenn der commit nicht klappt?

Ansatz: wir probieren es solange, bis es klappt. Dazu wollen wir die Sperre möglichst kurz halten, wie im folgenden Beispiel für einen Spinlock aus [Lea99, 3.2.6.5]. Wenn Sie das Beispiel selbst implementieren, werden Sie sehen, dass die Laufzeiten steigen, wenn man nach den Fehlschlägen beim `commit` sofort wieder probiert, ob das `commit` erfolgreich ist. Die Schwierigkeit besteht darin, die *richtigen* Pausen für die Wiederholung zu finden.

Aktives Warten: Busy Wait nach [Lea99, 3.2.6.5]

! Busy Wait ist nur für extreme Sonderfälle sinnvoll. Meist funktioniert die einfache Lösung mit `wait()` und `notifyAll()` besser. Zuverlässiger ist sie in jedem Fall.

Das folgende Listing zeigt die einfachste Form eines sog. “Spinlock”.

```
1  protected void busyWaitUntilCond() {
2      while(!busy) {
3          Thread.yield();
4      }
5  }
```

1. Ein Spinlock kann beliebig viel CPU-Zeit verbrauchen
2. `yield` ist nur ein Hinweis an die JVM, es muss kein anderer Thread aktiviert werden
3. Einzig sinnvoller Anwendungsfalls: extrem “kurze” kritische Abschnitte

```
1  class SpinLock { // Avoid needing to use this
2      private volatile boolean busy = false;
3
4      synchronized void release() {
5          busy = false;
6      }
7
8      void acquire() throws InterruptedException {
9          int itersBeforeYield = 100; // 100 is arbitrary
10         int itersBeforeSleep = 200; // 200 is arbitrary
11         long sleepTime = 1; // 1msec is arbitrary
12         int iters = 0;
13         for(;;) {
14             if(!busy) { // test-and-test-and-set
15                 synchronized(this) {
```

```
16                      if(!busy) {
17                          busy = true;
18                          return;
19                      }
20                  }
21              }
22
23              if(iters < itersBeforeYield) { // spin phase
24                  ++iters;
25              } else if(iters < itersBeforeSleep) { // yield phase
26                  ++iters;
27                  Thread.yield();
28              } else { // back-off phase
29                  Thread.sleep(sleepTime);
30                  sleepTime = 3 * sleepTime / 2 + 1; // 50 percent is arbitrary
31              }
32          }
33      }
34  }
```