www.javaspektrum.de

Ausgabe 2/2021
D € 9,90 · A € 10,90 · CHF 18,20

4 194156 109901 02

JavaSPEKTRUM

Magazin für professionelle Entwicklung und digitale Transformation

Auf Knopfdruck –
Continuous Integration, Release & Delivery

Wie passen DevOps und SAP zusammen?

Continuous Delivery – eine Managerperspektive

Interview

Jürgen Schomakers, Managing Partner der Esri Deutschland GmbH, erklärt warum Geoinformationssysteme drängende Probleme unserer Zeit lösen können

Fachthemen

Wege durch den REST-API-Dschungel mit GraphQL

Programmierung mit Zukunft? – Future und CompletableFuture

zum Inhalt

Auf Knopfdruck – Continuous Integration, Release & Delivery

G30325

Nebenläufige Programmierung mit Zukunft?

Future und CompletableFuture

Christian Heitzmann

Ausführungsstränge innerhalb des Interface `Runnable` bieten keine einfache Möglichkeit zur Rückgabe von Werten. `Future` in Kombination mit Executoren füllt diese Lücke. Mehrere voneinander abhängige Verarbeitungsschritte lassen sich auch geschickt als Instanzen der Klasse `CompletableFuture` kombinieren, sodass der Flexibilität bei der nebenläufigen Implementierung kaum mehr Grenzen gesetzt sind.

Threads und damit Möglichkeiten zur nebenläufigen Programmierung standen in Java bereits vom ersten Tag an zur Verfügung. Bis und mit Version 1.4 diente allein das Interface `Runnable` mit seiner einzigen Methode `void run()` zur Kapselung eines auszuführenden Code-Strangs. Das änderte sich mit Java Version 5, als neu das typisierte Interface `Callable<T>` hinzukam. Seine ebenfalls einzige Methode `T call()` (man beachte das `T`) erlaubte neu auch die Rückgabe eines Werts, was mit `Runnable` nur über Umwege möglich war. Seitdem hat sich in Sachen Multithreading in Java auch anderweitig viel getan.

Zum einen hat die Einführung und stetige Erweiterung des Pakets `java.util.concurrent` ab Java 5 die Art der nebenläufigen Programmierung in Java grundlegend auf den Kopf gestellt. Der Low-Level-Umgang mit `Thread`, `wait` und `notify(All)`, `synchronized` und *synchronized Wrappern* sollte neu dem High-Level-Einsatz von *Executoren*, diversen *Queues*, ausgefeilten *Synchronisatoren*, *Locks* und *Concurrent Collections* weichen [Hei20]. „Sollte" deshalb, weil seine Konzepte in der Ausbildung und IT-Praxis meiner Meinung nach heute immer noch ein Schattendasein führen, obwohl sie bereits vor über 16 Jahren eingeführt wurden.

Zum anderen hat die Einführung von *Lambda-Ausdrücken* in Java 8 das explizite Erzeugen von `Runnable`- und `Callable`-Instanzen mehr oder weniger überflüssig gemacht. Ein Lambda-Ausdruck oder eine Methodenreferenz genügt, um einem `ExecutorService` einen auszuführenden Code-Strang beziehungsweise eine auszuführende Methode zu übergeben, wie Tabelle 1 zeigt [JAPIExecutorService].

Zugriff auf das Ergebnis asynchroner Tasks: Future

Tabelle 1 lässt sich unter anderem entnehmen:

- Auf die direkte Erzeugung von `Thread` wird zugunsten von `ExecutorService` verzichtet.
- `ExecutorService#submit` kann sowohl `Runnable`- als auch `Callable`-Instanzen annehmen.
- Dem `submit`-Aufruf sieht man es nicht direkt an, ob es sich um die `Runnable`- oder die `Callable`-Variante handelt. Die Methodensigna-

Interface	Methode	Exceptions	Lambda-Ausdruck-Beispiele	Rückgabe von `ExecutorService`
`Runnable`	`void run()`	*keine*	`ExecutorService#submit(() -> printSomething())` `ExecutorService#submit(this::printSomething)`	`Future<?>` `Future<?>`
`Callable<T>`	`T call()`	`throws Exception`	`ExecutorService#submit(() -> returnSomething())` `ExecutorService#submit(this::returnSomething)`	`Future<T>` `Future<T>`

Tabelle 1: Funktionale Interfaces Runnable und Callable

Christian Heitzmann ist Gründer und Geschäftsführer der Simplexa-Code AG in Luzern, die sich auf Softwareentwicklung, -schulung und -beratung mit Schwerpunkt technischer Implementierungsthemen und Java spezialisiert hat. Er entwickelt seit über 20 Jahren Software und unterrichtet seit vielen Jahren Algorithmen und Mathematik.
E-Mail: christian.heitzmann@simplexacode.ch

tur (hier: `printSomething` (*ohne* Rückgabewert) vs. `returnSomething` (*mit* Rückgabewert) entscheidet, welche Variante der Compiler im Hintergrund heranzieht.

Jetzt stellt sich natürlich die Frage, wie auf das Ergebnis von eingereichten Tasks zugegriffen werden kann. `ExecutorService#submit` liefert jeweils ein typisiertes `Future`-Objekt zurück. Das Interface `Future<T>` enthält die fünf Methoden [JAPIFuture] aus Tabelle 2.

Rückgabe	Methode	blockierend
`boolean`	`cancel(boolean interrupt)`	nein
`boolean`	`isCancelled()`	nein
`boolean`	`isDone()`	nein
`T`	`get()`	ja
`T`	`get(long timeout, TimeUnit unit)`	ja, bis Time-out

Tabelle 2: Future

Die Tasks, die einem `ExecutorService` zur asynchronen Ausführung übergeben werden, lassen sich also „von außen" abbrechen (`cancel`), bezüglich ihres Status abfragen (`isCancelled`, `isDone`) und selbstverständlich in ihrem Ergebnis auslesen (`get` mit oder ohne `timeout`). Zu beachten ist, dass die `get`-Methode blockiert, das heißt, der Aufrufer von `get` wird so lange warten müssen, bis ein Ergebnis vorliegt oder eine Exception geworfen wird. Letzteres ist bereits dann der Fall, wenn der Task mittels `cancel` abgebrochen wurde oder wird. Möchte man das Risiko des Blockierens bei `get` nicht eingehen, dann empfiehlt es sich, zuvor mittels `isDone` unverbindlich „auf Tuchfühlung" zu gehen. Die Methode `isDone` wird `true` zurückliefern, wenn der Task normal beendet (`return`), abnormal beendet (Exception) oder abgebrochen (`cancel`) wurde.

Listing 1 zeigt das Gerüst eines kleinen Demoprogramms für die Anwendung von `Future`. Die Methode `numberIsPrime` testet (in einem bewusst naiven und ineffizienten Algorithmus, nämlich der *Probedivision* [WikiProbedivision]), ob ein übergebener `BigInteger` eine Primzahl ist oder nicht. Auf meiner Maschine benötigt ein solcher Test für eine 17-stellige Primzahl leicht über 15 Sekunden. Die Methode ist hier aus Platzgründen und mangels Relevanz nicht abgedruckt. Sie findet sich aber, zusammen mit allen anderen kompletten Code-Beispielen, als Download unter [Code].

Die „Hauptmethode" `execute` erzeugt zuerst einen *gecacheten Thread-Pool* als `ExecutorService` und übergibt ihm danach die Methode `boolean numberIsPrime(BigInteger number)` als auszuführenden Task. Die Methode wird einen `boolean` zurückgeben, auf den wir über das `Future<Boolean>`-Objekt Zugriff haben werden.

Listing 2 zeigt die Methode `waitBusilyAndPrintResult(Future<Boolean>)`, die das eben erwähnte `Future`-Objekt als Eingabeparameter erhält. In einem separaten Thread (ebenfalls vom gecacheten

```
import java.math.*;
import java.util.concurrent.*;

public class FutureDemo {

  private static final String NUMBER_STRING = "74747474747474747";
  private static final int TIMEOUT = 15;

  public static void main(String[] args) {
    FutureDemo demo = new FutureDemo();
    demo.execute();
  }

  private void execute() {
    BigInteger number = new BigInteger(NUMBER_STRING);
    ExecutorService service = Executors.newCachedThreadPool();
    Future<Boolean> primeFuture =
        service.submit(() -> numberIsPrime(number));
    service.submit(() -> waitBusilyAndPrintResult(primeFuture));
    service.submit(() -> waitBlockinglyAndPrintResult(primeFuture));
    service.shutdown();
  }
```

Listing 1: Gerüst von FutureDemo

Thread-Pool ausgeführt) wird mittels *aktiven Wartens* regelmäßig überprüft, ob für dieses `Future` schon das Ergebnis vorliegt (`isDone`). Falls nein, gibt er die Meldung *„Waiting busily ..."* aus und schläft für 1 Sekunde. Dieser zugegeben schlechte Stil soll stellvertretend für andere, sinnvolle Arbeit stehen, die in dieser Zeit im Thread verrichtet werden könnte.

Ist das Ergebnis bereit (`isDone == true`), dann ist auch garantiert, dass der darauffolgende `get`-Aufruf ohne Blockade sofort das Ergebnis zurückliefern wird. Allerdings werden dabei zumindest die zwei geprüften Exceptions `InterruptedException` und `ExceutionException` abgefangen und behandelt werden müssen. Darüber hinaus empfehle ich, auch gleich die ungeprüfte (Runtime-)Exception `CancellationException` mit zu behandeln, da ohne sie nicht klar erkannt werden kann, ob die Berechnung zuvor abgebrochen wurde oder nicht. Durch diese Ausnahmebehandlungen bläht der Code von ursprünglich zwei Zeilen (Ergebnis auslesen und ausgeben) leider sehr auf.

```
private void waitBusilyAndPrintResult(Future<Boolean> future) {
  while (!future.isDone()) {
    System.out.println("Waiting busily ...");
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.err.println("Wait was interrupted.");
      Thread.currentThread().interrupt();
    }
  }
  try {
    boolean result = future.get();
    System.out.format(
        "%s is %sa prime number.%n",
        NUMBER_STRING, result ? "" : "not ");
  } catch (InterruptedException e) {
    System.err.println("Thread was interrupted.");
    Thread.currentThread().interrupt();
  } catch (ExecutionException e) {
    System.err.println("Computation threw an exception:");
    e.printStackTrace();
  } catch (CancellationException e) { // RuntimeException
    System.err.println("Computation was canceled.");
  }
}
```

Listing 2: waitBusilyAndPrintResult

```
private void waitBlockinglyAndPrintResult(Future<Boolean> future) {
  System.out.println("Waiting blockingly ...");
  try {
    boolean result = future.get(TIMEOUT, TimeUnit.SECONDS);
    System.out.format(
        "%s is %sa prime number.%n",
        NUMBER_STRING, result ? "" : "not ");
  } catch (InterruptedException e) {
    System.err.println("Thread was interrupted.");
    Thread.currentThread().interrupt();
  } catch (ExecutionException e) {
    System.err.println("Computation threw an exception:");
    e.printStackTrace();
  } catch (TimeoutException e) {
    System.err.println("Wait for result timed out.");
    System.err.println("Computation will now be canceled.");
    future.cancel(true);
  } catch (CancellationException e) { // RuntimeException
    System.err.println("Computation was canceled.");
  }
}
```

Listing 3: waitBlockinglyAndPrintResult

Ein schönerer Stil zur Entgegennahme des Berechnungsergebnisses findet sich in Listing 3. Die Methode `waitBlockinglyAndPrintResult` wartet – wie der Name bereits impliziert – *blockierend* auf das Ergebnis, indem es `get` mit einem einstellbaren Time-out aufruft. Erneut müssen dabei wieder eine Reihe von Exceptions abgefangen und behandelt werden. Bei der Time-out-Variante von `get` kommt noch die geprüfte Exception `TimeoutException` hinzu. Deren Behandlung demonstriert im vorliegenden Code-Beispiel, wie der Task via seinem `Future`-Objekt hart abgebrochen werden kann (`cancel(true)`). Das ist dann sinnvoll, wenn nebst dem eigenen Thread noch andere Threads auf das Ergebnis warten und diese Warterei abgebrochen werden soll. Fühlen Sie sich – lieber Leser – frei, mit den Code-Beispielen zu experimentieren.

Verkettung asynchroner Tasks: CompletableFuture

In der Praxis kann es vorkommen, dass eine große Aufgabe nicht nur aus einer einzigen, sondern aus mehreren nacheinander auszuführenden Methoden besteht. Diese einzelnen Methoden lassen sich dann aber nicht einfach einem Thread-Pool übergeben, weil dann die zeitliche Aufeinanderfolge nicht mehr gewährleistet wäre. Außerdem müsste die Ausgabe einer vorherigen Methode jeweils manuell als Eingabe in die darauffolgende Methode transferiert werden. Genau für derartige Aufgaben wurde in Java 8 `CompletableFuture` eingeführt [JAPICompletableFuture].

Prägnante und überzeugende Anwendungsbeispiele liegen nicht auf der Hand und konnte ich auch in der Literatur nicht finden. In den populären Java-Büchern habe ich meist nur die bloße „Alibi-asynchrone" Aneinanderreihung von Methoden gefunden, die dazu noch so kurz waren (teilweise nur eine einzige `System.out.println`-Anweisung lang), dass ich deren Sinnhaftigkeit ernsthaft hinterfragen musste. Für derartige Aneinanderreihungen von Methoden braucht es nämlich gar keine `CompletableFuture`s, sondern ein Entwickler kann einfach eine neue, kapselnde Methode schreiben, die nacheinander die „kleinen" Methoden aufruft. Diese „große" Methode übergibt er dann einem `ExecutorService` – fertig.

Ich werde daher versuchen, in diesem Artikel die Lücke mit einem sinnvolleren, einleuchtenderen und flexibleren Beispiel zu füllen. Der Anwendungsfall, an dem ich den Einsatz von `CompletableFuture`s demonstrieren will, besteht aus neun Schritten:

9 Schritte des CompletableFuture-Anwendungsfalls

1. `createRandomNumberSet` erzeugt eine **zufällig große Menge großer BigInteger-Zufallszahlen.**
2. `removeEvenNumbers` löscht aus ihr daraufhin alle **geraden Zahlen.**
3. `removePrimeNumbers` löscht aus ihr daraufhin sämtliche **Primzahlen**. Als Algorithmus zur Detektion von Primzahlen kommt ein wesentlicher effizienterer Algorithmus als der im ersten Abschnitt dieses Artikels zum Einsatz.
4. `removeSquareNumbers` löscht aus der Menge daraufhin sämtliche **Quadratzahlen.**
5. `writeNumbersToFile` schreibt die Zahlenmenge dann in eine **Datei**.
6. `printRoughAverage` berechnet den ungefähren **Durchschnitt** der Menge (abgesehen von Rundungsfehlern) und gibt das Ergebnis auf der Konsole aus.
7. `printRoughMedian` berechnet den ungefähren **Median**, also den mittleren Wert der Menge (den Grenzfall bei einer geraden Anzahl von Elementen außer Acht gelassen), und gibt das Ergebnis auf der Konsole aus. Die Bestimmung des Medians beinhaltet auch eine Sortierung der Zahlenmenge.
8. `printMin` sucht den **Minimalwert** der Zahlenmenge und gibt ihn aus.
9. `printMax` sucht den **Maximalwert** der Zahlenmenge und gibt ihn aus.

Jetzt fragt sich der eine oder andere Leser bestimmt: *„Wie in aller Welt kommt der auf so ein Szenario?"* Meinen Überlegungen lagen folgende Kriterien zugrunde:

- Die Größenordnungen und damit die Laufzeiten der einzelnen Schritte sollen sich flexibel einstellen lassen.
- Die einzelnen Schritte sollen unterschiedlich „schwer" sein und somit auch unterschiedlich lange Laufzeiten haben.
- Einzelne Methoden sollen bewusst *nicht*, andere hingegen *ideal* parallelisierbar sein.
- Eine I/O-Operation soll bewusst einen Engpass bilden und seine Parallelisierung unattraktiv machen.
- Die Kombination von CPU-gebundenen und I/O-gebundenen Aufgaben soll den Einsatz mehrerer Thread-Pools motivieren.

Der gesamte Quellcode ist zu umfangreich, um hier abgedruckt zu werden. Stattdessen sei erneut auf [Code] verwiesen, wo die kompletten Beispiele heruntergeladen, direkt ausgeführt und für eigene Experimente mutiert werden können. Das Programm zeigt außerdem in einer Tabelle für jeden Verfahrensschritt an, welcher (Worker-) Thread aus welchem Thread-Pool sich ihm angenommen hat.

Listing 4 zeigt die bereits angesprochene rein sequenzielle Variante. Die Methode `executeSequentiallyOnMainThread` ruft die neun oben beschriebenen Methoden 10-mal hintereinander auf. Alle neun Methoden werden jeweils der Reihe nach auf dem `main`-Thread ausgeführt.

Listing 5 verwendet erstmals `CompletableFuture`s. Startpunkt für die Verknüpfung einzelner Tasks ist die Klasse `CompletableFuture` selbst. Deren statische Methode `supplyAsync` nimmt einen `Supplier` (erstaunlicherweise *kein* `Runnable` oder `Callable`) entgegen. Das Resultat dieses `Supplier`s (hier: die erzeugte Zahlenmenge aus `createRandomNumberSet`) dient als Eingabe für die darauffolgende `thenApplyAsync`-Methode. `thenApplyAsync`-Methoden werden jeweils mit `Function`s parametrisiert. Das letzte Glied der Kette, `thenAcceptAsync`, erwartet als Terminator folgerichtig einen `Consumer`. Die Analogie zu Streams

```
import java.math.*;
import java.util.*;
import java.util.concurrent.*;

/* Remaining imports omitted. */

public class CompletableFutureDemo {

  private static final int RUN_COUNT = 10;
  private static final int NUMBER_COUNT_MAX = 10_000_000;
  private static final int NUMBER_BIT_COUNT = 128;
  private static final int CPU_POOL_THREAD_COUNT = 8;

  /* [...] Main method omitted. */

  private void executeSequentiallyOnMainThread() {
    for (int i = 1; i <= RUN_COUNT; i++) {
      Set<BigInteger> numberSet = createRandomNumberSet();
      removeEvenNumbers(numberSet);
      removePrimeNumbers(numberSet);
      removeSquareNumbers(numberSet);
      writeNumbersToFile(numberSet);
      printRoughAverage(numberSet);
      printRoughMedian(numberSet);
      printMin(numberSet);
      printMax(numberSet);
    }
  }
```

Listing 4: executeSequentiallyOnMainThread

ist augenscheinlich, wie Tabelle 3 noch mal aufzeigt (`CF` steht für `CompletableFuture`).

Rückgabe	Methode	Stream
`CF<U>`	`CF.supplyAsync(Supplier<U>)`	create
`CF<U>`	`CF.thenApplyAsync(Function<↑T, ↓U>)`	intermediate
`CF<Void>`	`CF.thenAcceptAsync(Consumer<↑T>)`	terminal

Tabelle 3: Teile der Push-API von CompletableFuture

Ausführung in eigenen Thread-Pools

Ausgeführt werden die einzelnen Tasks auf Threads des *Common-Pools* [JAPICommonPool]. Dabei handelt es sich um Javas globalen Thread-Pool, auf dem zum Beispiel auch das Fork-Join-Framework [Hei19] oder parallele Streams ausgeführt werden. Die Konfiguration des Common-Pools erfolgt über System-Properties und ist daher alles andere als komfortabel oder flexibel. Dabei

```
  private void executeCompletableFuturesOnCommonPool() {
    @SuppressWarnings("unchecked")
    CompletableFuture<Void>[] completableFutureArray =
        new CompletableFuture[RUN_COUNT];
    for (int runNumber = 0; runNumber < RUN_COUNT; runNumber++) {
      CompletableFuture<Void> completableFuture =
          CompletableFuture.supplyAsync(this::createRandomNumberSet)
              .thenApplyAsync(this::removeEvenNumbers)
              .thenApplyAsync(this::removePrimeNumbers)
              .thenApplyAsync(this::removeSquareNumbers)
              .thenApplyAsync(this::writeNumbersToFile)
              .thenApplyAsync(this::printRoughAverage)
              .thenApplyAsync(this::printRoughMedian)
              .thenApplyAsync(this::printMin)
              .thenAcceptAsync(this::printMax);
      completableFutureArray[runNumber] = completableFuture;
    }
    CompletableFuture.allOf(completableFutureArray).join();
  }
```

Listing 5: executeCompletableFuturesOnCommonPool

kann es auch zu kritischen Seiteneffekten auf die anderen soeben erwähnten Java-internen Parallelisierungskonzepte kommen, sodass sich der Einsatz eigener Thread-Pools geradezu aufdrängt.

Jeder `supplyAsync`-, `thenApplyAsync`- und `thenAcceptAsync`-Methode lässt sich auch ein `Executor` beziehungsweise `ExcecutorService` als zusätzlichen Parameter mitgeben. In `executeCompletableFuturesOnDifferentThreadPools` in Listing 6 wird dies demonstriert. Um den Einsatz eigener Thread-Pools noch überzeugender zu machen, werden zwei verschiedene Thread-Pools erzeugt: ein `cpuService` als *Fixed Thread Pool* mit einer einstellbaren Anzahl Worker-Threads und ein `ioService` als *Single Thread Executor*, der jeweils für das Schreiben der Zahlenmenge in eine Datei zuständig ist. Die Idee ist, bei Festplattenzugriffen bewusst auf parallele Zugriffe zu verzichten, sondern stattdessen eine serielle Einzelabarbeitung zu forcieren.

Verzweigen und Vereinen

Dem aufmerksamen Leser wird nicht entgangen sein, dass die bisher gezeigten Code-Beispiele – trotz `CompletableFuture` – noch immer nicht wirklich sinnvoll parallelisiert sind. Zwar laufen `RUN_COUNT` komplette Ausführungsstränge (jeweils bestehend aus den neun erwähnten Methoden) bis auf das Schreiben in die Datei parallel, *innerhalb* eines Ausführungsstranges ließe sich aber noch effizienter parallelisieren. Insbesondere können *während* der I/O-gebun-

ARS

Ihr Partner für die erfolgreiche Umsetzung Ihrer IT-Modernisierung

Wir beraten unsere Kunden zu Modernisierungsszenarien und -strategien und unterstützen sie umfassend - der ganzheitliche Ansatz ist unsere Stärke.
Kreativ. Innovativ. Qualitativ.

Profitieren Sie von unserer Erfahrung!

www.ars.de

zum Inhalt

```
private void executeCompletableFuturesOnDifferentThreadPools() {
  ExecutorService cpuService =
      Executors.newFixedThreadPool(CPU_POOL_THREAD_COUNT);
  ExecutorService ioService = Executors.newSingleThreadExecutor();
  @SuppressWarnings("unchecked")
  CompletableFuture<Void>[] completableFutureArray =
      new CompletableFuture[RUN_COUNT];
  for (int runNumber = 0; runNumber < RUN_COUNT; runNumber++) {
    CompletableFuture<Void> completableFuture =
        CompletableFuture.supplyAsync(
              this::createRandomNumberSet, cpuService)
            .thenApplyAsync(this::removeEvenNumbers, cpuService)
            .thenApplyAsync(this::removePrimeNumbers, cpuService)
            .thenApplyAsync(this::removeSquareNumbers, cpuService)
            .thenApplyAsync(this::writeNumbersToFile, ioService)
            .thenApplyAsync(this::printRoughAverage, cpuService)
            .thenApplyAsync(this::printRoughMedian, cpuService)
            .thenApplyAsync(this::printMin, cpuService)
            .thenAcceptAsync(this::printMax, cpuService);
    completableFutureArray[runNumber] = completableFuture;
  }
  CompletableFuture.allOf(completableFutureArray).join();
  cpuService.shutdown();
  ioService.shutdown();
}
```

Listing 6: executeCompletableFuturesOnDifferentThreadPools

denen Abarbeitung von `writeNumbersToFile` *parallel dazu* bereits der Durchschnitt, der Median, das Minimum und das Maximum ermittelt und auf der Konsole ausgegeben werden. Listing 7 zeigt, wie sich derartige *Verzweigungen* einfach realisieren lassen.

Das *Vereinen* von `CompletableFuture`s ist für zwei Stränge über `thenCombineAsync` noch machbar, für drei und mehr Stränge aber keine Freude mehr. Das hervorragende Buch [HetTra16] behandelt nicht nur diesen Teil, sondern auch die gesamte Thematik der `Future`s und `CompletableFuture`s ausführlicher und detaillierter als jedes andere im Rahmen meiner Recherchen herangezogene Java-Buch.

Fazit

Als regelmäßiger JavaSPEKTRUM-Autor habe ich selten so lange recherchiert, selten einen so großen API-Umfang durchforstet und selten so lange an überzeugenden Beispielen getüftelt wie für den vorliegenden Artikel. Ich weiß nicht, ob Oracle mit der Einführung von `CompletableFuture` und seinem Interface `CompletionStage` der Java-Gemeinde nicht eher einen Bärendienst erwiesen hat, ist der schier unüberschaubare Umfang dieser neuen APIs doch geradezu erschlagend. Das Angebot neuer Concurrency-High-Level-APIs sollte meiner Meinung nach die Arbeit mit nebenläufiger Programmierung vereinfachen und nicht verkomplizieren. Eine Vielzahl kleingranularer Konfigurationsmöglichkeiten macht die Performanz von Multithreading-Anwendungen nämlich erfahrungsgemäß nicht besser.

Mein Artikel konnte natürlich nur an der Oberfläche kratzen. Wer sich auf die hier gezeigten „Standardfunktionen" beschränkt und einen klaren Plan seines Ausführungsstranges hat, der kann mit `CompletableFuture`s mit wenig Aufwand eine übersichtliche, performante und flexible Parallelisierung seiner Aufgaben erreichen. Insbesondere erlaubt der Einsatz mehrerer unabhängiger Thread-Pools eine effiziente Lastverteilung beziehungsweise -bündelung, die ansonsten nur mit viel manuellem Synchronisationsaufwand zu realisieren wäre.

Literatur und Links

[BlogBigDecimal] C. Heitzmann, Using BigDecimal as an Accurate Replacement for Floating-Point Numbers, SimplexaCode Software Blog, https://link.simplexacode.ch/ewvs

```
private void executeCompletableFuturesWithSplitting() {
  ExecutorService cpuService =
      Executors.newFixedThreadPool(CPU_POOL_THREAD_COUNT);
  ExecutorService ioService = Executors.newSingleThreadExecutor();
  @SuppressWarnings("unchecked")
  CompletableFuture<Void>[] completableFutureArray =
      new CompletableFuture[RUN_COUNT];
  for (int runNumber = 0; runNumber < RUN_COUNT; runNumber++) {
    CompletableFuture<Set<BigInteger>> part1 =
        CompletableFuture.supplyAsync(
              this::createRandomNumberSet, cpuService)
            .thenApplyAsync(this::removeEvenNumbers, cpuService)
            .thenApplyAsync(this::removePrimeNumbers, cpuService)
            .thenApplyAsync(this::removeSquareNumbers, cpuService);
    CompletableFuture<Void> part2a =
        part1.thenAcceptAsync(this::writeNumbersToFile, ioService);
    CompletableFuture<Void> part2b =
        part1.thenAcceptAsync(this::printRoughAverage, cpuService);
    CompletableFuture<Void> part2c =
        part1.thenAcceptAsync(this::printRoughMedian, cpuService);
    CompletableFuture<Void> part2d =
        part1.thenAcceptAsync(this::printMin, cpuService);
    CompletableFuture<Void> part2e =
        part1.thenAcceptAsync(this::printMax, cpuService);
    CompletableFuture<Void> part2 =
        CompletableFuture.allOf(
            part2a, part2b, part2c, part2d, part2e);
    completableFutureArray[runNumber] = part2;
  }
  CompletableFuture.allOf(completableFutureArray).join();
  cpuService.shutdown();
  ioService.shutdown();
}
```

Listing 7: executeCompletableFuturesWithSplitting

[Code] Quellcode zum Herunterladen, https://link.simplexacode.ch/va9s
[Hei19] C. Heitzmann, Das Fork-Join-Framework, in: JavaSPEKTRUM, 5/2019
[Hei20] C. Heitzmann, Locking in Java, in: JavaSPEKTRUM, 2/2020
[HetTra16] J. Hettel, M. T. Tran, Nebenläufige Programmierung in Java, dpunkt.verlag, 2016
[Ind21] M. Inden, Der Weg zum Java-Profi, 5. Auflage, dpunkt.verlag, 2021
[JAPICommonPool] Java Platform, Standard Edition & Java Development Kit Version 15 API Specification, Class ForkJoinPool, Method commonPool, https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/concurrent/ForkJoinPool.html#commonPool()
[JAPICompletableFuture], Java Platform, Standard Edition & Java Development Kit Version 15 API Specification, Class CompletableFuture<T>, https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/concurrent/CompletableFuture.html
[JAPIExecutorService] Java Platform, Standard Edition & Java Development Kit Version 15 API Specification, Interface ExecutorService, https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/concurrent/ExecutorService.html
[JAPIFuture] Java Platform, Standard Edition & Java Development Kit Version 15 API Specification, Interface Future<V>, https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/concurrent/Future.html
[Ull18] C. Ullenboom, Java SE 9 Standard-Bibliothek, Rheinwerk Verlag, 2018
[WikiProbedivision] Probedivision, https://de.wikipedia.org/wiki/Probedivision