www.javaspektrum.de

Ausgabe 5/2023
D € 12,90 · A € 13,90 · CHF 20,20

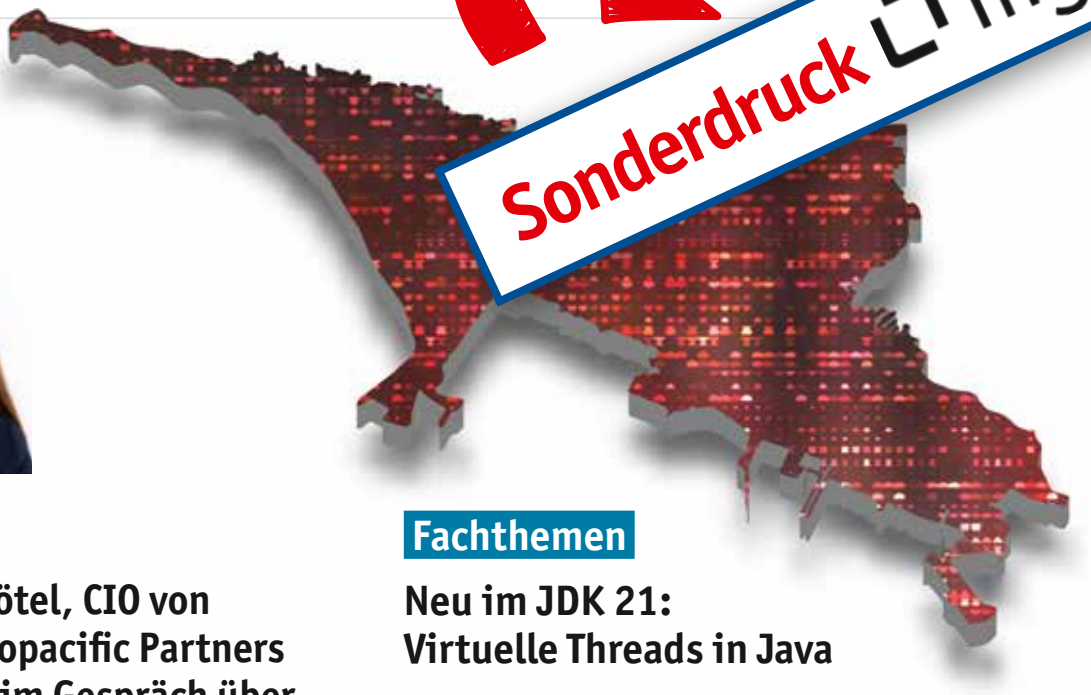
JavaSPEKTRUM

Magazin für professionelle Entwicklung und digitale Transformation

Der Weg nach Kotlin –
Konkurrenz für Java?

Monaden in Kotlin: Funktional für eine bessere Architektur

Java- und Kotlin-Lambdas im Vergleich



Sonderdruck mgm

Interview

Katharina Knötel, CIO von Coca-Cola Europacific Partners Deutschland, im Gespräch über Innovationsprozesse und digitale Transformation

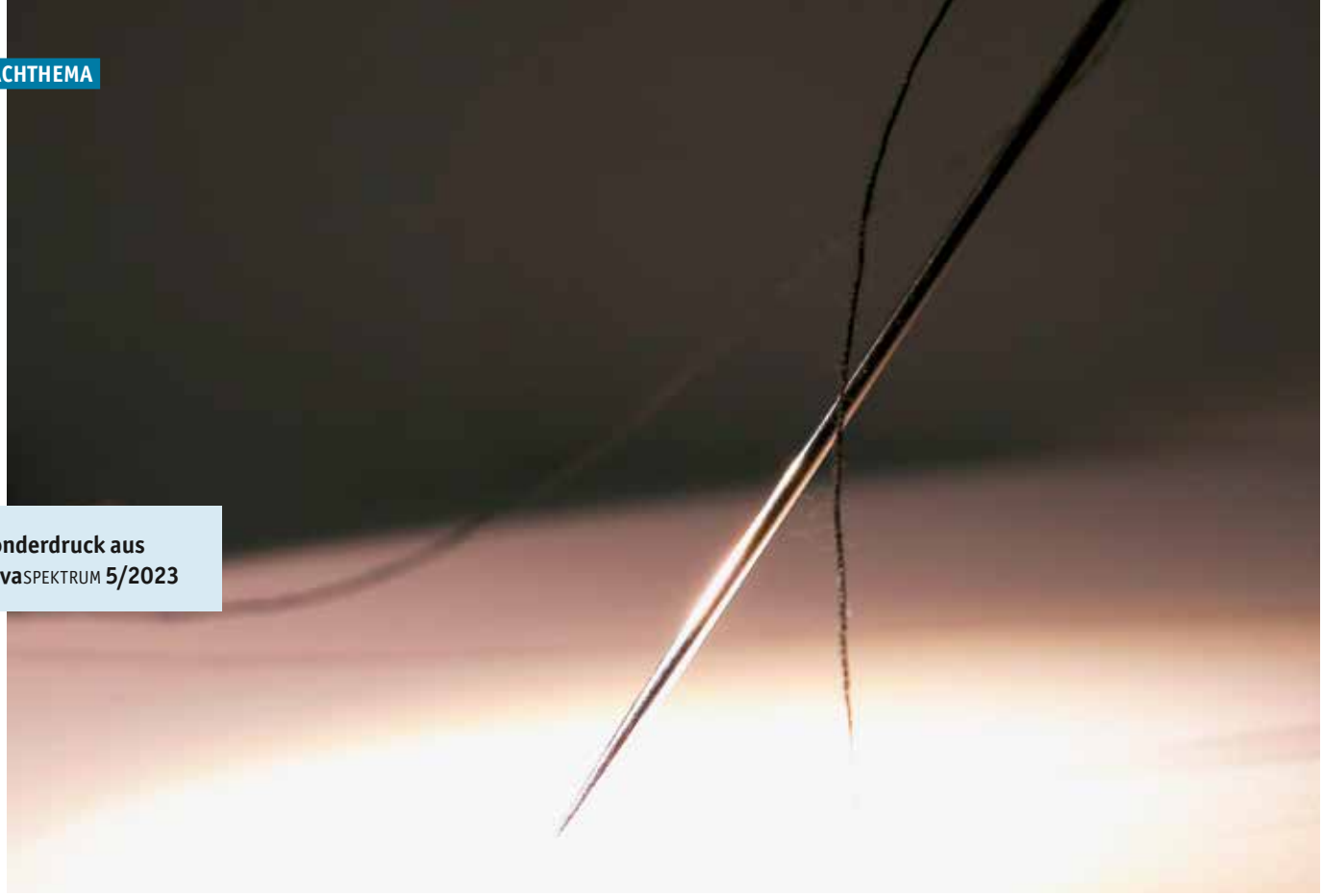
Fachthemen

Neu im JDK 21: Virtuelle Threads in Java

ChatGPT-4: Was kann es und wer sollte es nutzen?

Der Weg nach Kotlin – Konkurrenz für Java?

G30325



Sonderdruck aus
JavaSPEKTRUM 5/2023

Nur nicht den Faden verlieren!

Virtuelle Threads in Java

Christian Schuster

Threads sind überall: in Backends von Webanwendungen, Rich Clients, parallelen Pipelines zur Verarbeitung großer Datenmengen – um nur ein paar zu nennen. Auch Java bietet schon seit vielen Jahren brauchbare Unterstützung für Scheduling, Thread-Pools & Co., wirklich viel Neues gab es in der Richtung aber lange Zeit nicht. Mit JDK 21 wird sich das im September 2023 ändern: Parallelisierung in Java bekommt mit virtuellen Threads nicht nur einen neuen Anstrich, sondern ein komplett neues Fundament!

Ob zur Abarbeitung von Requests in Servlet-Containern, zur asynchronen Ausführung von Tasks in Desktop-Anwendungen oder für grundsätzliche JVM-Funktionen wie Garbage Collection: Threads sind in Java – und nicht nur dort – allgegenwärtig. In meinem Artikel werde ich virtuelle Threads vorstellen und die damit verbundenen Vorteile und neuen Herausforderungen näher beleuchten.

Mit [JEP444] unterstützt Java ab JDK 21 – oder JDK 19 mit aktivierten Preview-Features – sogenannte *virtuelle Threads*. Diese virtuellen Threads sollen trotz vollständiger Kompatibilität deutlich leichtgewichtiger sein als „klassische" Java-Threads, die zur besseren Unterscheidbarkeit als *Plattform-Threads* bezeichnet werden. Da stellt sich natürlich die Frage: Sind Plattform-Threads wirklich so teuer, dass sich eine Optimierung lohnt? Die Antwort auf diese Frage ist ein klares Ja: Ein Plattform-Thread entspricht in Java immer genau einem Betriebssystem-Thread, der seinen eigenen Stack und diverse Verwaltungsstrukturen im Speicher benötigt. Die Bereitstellung dieser Strukturen durch das Betriebssystem ist mit verhältnismäßig viel Aufwand verbunden. So dauert das Anlegen eines Plattform-Threads in Java selbst auf aktueller Hardware einige zehn bis hundert Mikrosekunden. Zum Vergleich: Das Erzeugen und Initialisieren eines Java-Objekts liegt üblicherweise deutlich unter 100 Nanosekunden, ist also um etwa drei Größenordnungen schneller.

Aus diesem Grund wurden Thread-Pools entwickelt, durch die sich der beim Anlegen von Betriebssystem-Threads anfallende Overhead amortisiert. Java bringt zum Erzeugen verschiedener Thread-Pools schon seit Version 1.5 die Klasse `java.util.concurrent.Executors` mit. Thread-Pools reduzieren zwar den durchschnittlichen Aufwand für die Beschaffung eines Betriebssystem-Threads und beschränken den Ressourcenverbrauch, führen aber zu verschiedenen neuen Problemen: In Servlet-Containern müssen sie üblicherweise deutlich überdimensioniert werden, damit eintreffende Requests zügig einen verarbeitenden Thread zugeteilt bekommen. Einerseits bleiben dadurch bei geringer Systemlast viele wertvolle Betriebssystem-Threads ungenutzt, andererseits erzeugen die zahlreichen Kontextwechsel zwischen den Threads bei hoher Systemlast zusätzlichen Overhead. Außerdem müssen anwendungsspezifische thread-lokale Variablen bei Verwendung gepoolter Threads immer explizit entfernt werden, da sie ansonsten bei der Verarbeitung des nächsten Tasks auf dem gleichen Thread falsche, sensible oder sicherheitskritische Informationen enthalten können.

Asynchrone Programmiermodelle wie reaktive Programmierung ermöglichen es, die Anzahl benötigter Betriebssys-

tem-Threads gering zu halten und damit die Skalierbarkeit eines Systems zu verbessern. Allerdings ergeben sich daraus zumindest in Java ein paar gravierende Nachteile: Stacktraces sind wenig brauchbar, Exceptions können nicht generell mit `try` – `catch` – `finally` behandelt werden, und Bibliotheken müssen entweder bereits reaktiv implementiert sein oder dahingehend adaptiert werden.

Neu: Virtuelle Threads!

Im Jahr 2018 wurde mit [ProjectLoom] ein OpenJDK-Projekt ins Leben gerufen, das Erweiterungen für strukturierte Nebenläufigkeit und neue Programmiermodelle in Java einbringt. Virtuelle Threads, das bisher sichtbarste Ergebnis dieses Projekts, lösen die zuvor beschriebenen Probleme von Threads in Java, indem sie die bisher zwingende Eins-zu-eins-Zuordnung zwischen Java-Thread und Betriebssystem-Thread aufbrechen.

Ein virtueller Thread ist zunächst ein ganz normales Java-Objekt vom Typ `java.lang.VirtualThread`, das alle für die Ausführung und insbesondere für eine zeitweilige Unterbrechung der Ausführung benötigten Informationen kapselt. Der nach Art einer [Continuation] materialisierte Ausführungszustand in einem virtuellen Thread enthält vor allem den aktuellen Stack, der je nach Bedarf vergrößert und auch wieder verkleinert wird. Da virtuelle Threads nur innerhalb der JVM existieren, also keine Entsprechung im Betriebssystem haben, lassen sie sich mit ausreichend Java-Heap problemlos millionenfach erzeugen.

Damit ein virtueller Thread seiner eigentlichen Aufgabe – der Ausführung von Anwendungscode – nachkommen kann, benötigt er letztlich natürlich trotzdem einen Betriebssystem-Thread. Das JDK erzeugt zu diesem Zweck einen von der gesamten JVM genutzten Pool von Plattform-Threads, den sogenannten *Carrier-Threads*, an die virtuelle Threads gebunden werden können (*Mount*). Bevor ein virtueller Thread blockiert, also auf ein Ereignis wartet und dafür gerade keine CPU benötigt, kann und sollte er sich aktiv von seinem Carrier-Thread lösen (*Unmount*).

Die Reaktivierung eines virtuellen Threads und die Zuweisung eines gegebenenfalls anderen Carrier-Threads übernimmt das JDK entweder bei Benachrichtigung durch das Betriebssystem (zum Beispiel für asynchrone I/O-Operationen) oder in einem dedizierten *Unparker-Thread* (für zeitbasierte Operationen). Der entscheidende Vorteil: Java-Anwendungen müssen zur optimalen Auslastung der CPU keine Thread-Pools mit übermäßig vielen Plattform-Threads mehr anlegen, die dann die meiste Zeit ungenutzt herumliegen, sondern erreichen das gleiche Ziel deutlich einfacher und besser mit einem Carrier-Thread pro CPU-Kern.

Da es für Anwendungscode nicht relevant sein sollte, welcher Carrier-Threads gerade für die Ausführung des aktuellen virtuellen Threads verantwortlich ist, stellt das JDK keine (öffentliche) API zum Zugriff auf den Carrier-Thread bereit. `Thread.currentThread()` liefert in diesem Fall immer den virtuellen Thread. Der Zugriff auf den Carrier-Thread ist den Klassen im Package `java.lang` und im Modul `jdk.internal.vm` vorbehalten.

Die Zusammenarbeit zwischen virtuellen Threads hat eine gewisse Ähnlichkeit mit kooperativem Multitasking, weil der Carrier-Thread aktiv freigegeben werden muss. Im Gegensatz zu den schon weitestgehend in Vergessenheit geratenen *Green Threads* aus Zeiten von Java 1.1 (das war zwischen 1997 und 2000) geschieht das bei Verwendung virtueller Threads allerdings zusätzlich zum präemptiven Multithreading auf Betriebssystemebene. Außerdem ist das Freigeben des Carrier-Thread so gut im JDK gekapselt, dass im Anwendungscode dafür nichts zu tun ist.

Virtuelle Threads in Anwendungen

Ob sich eine Umstellung auf virtuelle Threads in einem konkreten Anwendungsszenario lohnt, ist primär davon abhängig, was in den beteiligten Threads passiert. Jeder Thread hat zwei grundsätzliche Zustände:

- Entweder er möchte gerade etwas ausführen und braucht dafür einen CPU-Kern,
- oder er ist gerade blockiert (beispielsweise mit Warten auf eine I/O-Operation) und braucht deshalb gerade keinen CPU-Kern.

Die Unterschiede zwischen virtuellen Threads und Plattform-Threads fallen umso deutlicher aus, je länger und häufiger Threads blockiert sind: Wenn eine Anwendung oft auf Benutzerinteraktionen oder andere Systeme wartet, ist die Nutzung virtueller Threads wahrscheinlich vorteilhaft. Für langwierige CPU-lastige Operationen oder die Verarbeitung großer Datenmengen sind virtuelle Threads eher weniger geeignet, da ein gewisser Overhead für ihre Verwaltung anfällt.

Ein einfaches Rechenbeispiel mit nur einem Betriebssystem-Thread: Es sollen 10 Tasks ausgeführt werden, die jeweils erst eine Sekunde blockieren und anschließend noch eine Sekunde CPU-Zeit benötigen. Mit einem klassischen Plattform-Thread dauert die Ausführung dieser Tasks insgesamt 20 Sekunden, da alle zweisekündigen Tasks sequenziell ausgeführt werden müssen. Mit virtuellen Threads auf einem Carrier-Thread reduziert sich die Ausführungszeit auf 11 Sekunden: Während der ersten Sekunde benötigt kein virtueller Thread einen Carrier-Thread, und nach dieser Sekunde ist keiner der virtuellen Threads mehr blockiert. Anschließend müssen nur noch die 10 CPU-Sekunden abgearbeitet werden, wofür der Carrier-Thread 10 Sekunden benötigt. Mit virtuellen Threads wird der Betriebssystem-Thread also deutlich besser ausgelastet, da er nicht von blockierten Threads beansprucht wird. In der Praxis liegen die Ausführungszeiten wegen des Overheads für Kontextwechsel, Garbage Collector & Co. natürlich immer etwas über diesen idealen Werten.

Aufgrund ihrer Leichtgewichtigkeit ist es weder empfehlenswert noch sinnvoll, virtuelle Threads in Thread-Pools zu verwalten. Die Erzeugung neuer virtueller Threads ist deutlich effizienter und robuster als jedes Pooling. Da Thread-Pools manchmal aber auch dazu verwendet werden, den Zugriff auf andere Ressourcen (zum Beispiel Datenbankverbindungen) implizit zu begrenzen, kann der Wegfall des Thread-Pools zur Überlastung dieser Ressourcen führen. Bei Verwendung virtueller Threads müssen solche Begrenzungen daher immer explizit mit Connections-Pools, Semaphoren oder vergleichbaren Konstrukten erfolgen.

Neue APIs für virtuelle Threads

Da virtuelle Threads Instanzen der von `java.lang.Thread` abgeleiteten Klasse `java.lang.VirtualThread` sind, lassen sich virtuelle Threads und Plattform-Threads im Anwendungscode praktisch identisch

```
// erzeugt und startet einen Plattform-Thread
Thread platformThread = Thread.ofPlatform().start(() -> {
  // im Plattform-Thread auszuführender Code
});

// erzeugt und startet einen virtuellen Thread
Thread virtualThread = Thread.ofVirtual().start(() -> {
  // im virtuellen Thread auszuführender Code
});
```

Listing 1: Starten von Plattform- und virtuellen Threads

Christian Schuster ist Softwareentwickler und Teamleiter bei mgm technology partners GmbH. Sein Schwerpunkt ist die Umsetzung komplexer Web-Anwendungen auf Basis von Java und Spring – immer mit einem Auge über dem Tellerrand. E-Mail: christian.schuster@mgm-tp.com

verwenden. Um die Erzeugung von Plattform-Threads und virtuellen Threads weitestgehend zu vereinheitlichen, enthält das JDK auch ein paar praktische API-Erweiterungen. Der Code zur Erzeugung der beiden Thread-Typen unterscheidet sich damit nur in der aufgerufenen statischen Methode (s. Listing 1).

Auch ein neuer `ExecutorService`, der für jeden auszuführenden Task einen neuen virtuellen Thread erzeugt, ist im JDK verfügbar (s. Listing 2). Im Gegensatz zu den bisherigen Implementierungen legt er keinen einzigen Betriebssystem-Thread an, sondern überlässt die Verarbeitung den virtuellen Threads und damit effektiv dem JVM-weit genutzten Pool von Carrier-Threads.

Virtual Thread Pinning

Der Zustand, dass sich ein virtueller Thread während einer blockierenden Operation nicht von seinem Carrier-Thread lösen kann, wird als *Virtual Thread Pinning* bezeichnet, weil der virtuelle Thread an seinen Carrier-Thread „gepinnt" ist. Dieser Zustand ist hinsichtlich Performance und Skalierbarkeit einer Anwendung ungünstig, da ein solcher Carrier-Thread nicht für andere virtuelle Threads zur Verfügung steht. Blockierende Operationen in virtuellen Threads können also problematisch sein.

```
// ein ExecutorService, der für jeden Task einen
// neuen virtuellen Thread erzeugt
ExecutorService executor =
  Executors.newVirtualThreadPerTaskExecutor();
executor.submit(() -> {
  // in einem neuen virtuellen Thread auszuführender Code
});
```

Listing 2: ExecutorService mit virtuellen Threads

```
class SynchronizedVirtualThreadPinningTask {

  public synchronized void execute() {
    // Code mit blockierender Operation
  }
}
```

Listing 3: Virtual Thread Pinning mit synchronized-Block

```
import java.util.concurrent.locks.ReentrantLock;

class ReentrantLockNoVirtualThreadPinningTask {
  private final ReentrantLock lock = new ReentrantLock();

  public void execute() {
    lock.lock();
    try {
      // Code mit blockierender Operation
    } finally {
      lock.unlock();
    }
  }
}
```

Listing 4: ReentrantLock als Ersatz für einen synchronized-Block

Die meisten blockierenden Operationen im JDK wurden für die Nutzung mit virtuellen Threads bereits so angepasst, dass sie sich bei Ausführung in einem virtuellen Thread von ihrem Carrier-Thread lösen und die angeforderte Operation asynchron ausführen. Für ein paar blockierende Operationen, zum Beispiel `FileInputStream.read()` oder `Object.wait()`, ist das allerdings aufgrund von Einschränkungen der damit verbundenen Betriebssystem-Aufrufe oder des JDK nicht möglich, der virtuelle Thread bleibt während einer solchen blockierenden Operation an seinen Carrier-Thread gebunden. Der zeitweilig nicht zur Abarbeitung anderer virtueller Threads verfügbare Carrier-Thread wird in diesem Fall – und auch *nur* in diesem Fall – vom JDK selbst durch das temporäre Hinzufügen eines zusätzlichen Carrier-Threads kompensiert.

Virtual Thread Pinning tritt außerdem auf, wenn eine blockierende Operation innerhalb eines `synchronized`-Blocks aufgerufen wird (s. Listing 3). Diese Einschränkung könnte in einer späteren JDK-Version noch aufgelöst werden, bis dahin lassen sich betroffene `synchronized`-Blöcke aber relativ einfach durch ein `ReentrantLock` ersetzen (s. Listing 4).

Nativer Code ist, unabhängig davon, ob er über das Java Native Interface (JNI) oder die mit JDK 21 noch im Preview befindlichen *Foreign Functions* aufgerufen wird, ebenfalls anfällig für Virtual Thread Pinning: Zur Ausführung von nativem Code wird zwangsläufig ein Betriebssystem-Thread benötigt. Erfolgt der native Aufruf in einem virtuellen Thread, wird dieser an seinen Carrier-Thread gebunden. Der Carrier-Thread kann während des nativen Aufrufs nicht für andere virtuelle Threads verwendet werden, auch wenn der darunter liegende Betriebssystem-Thread blockiert wird.

Blockierende Aufrufe in `synchronized`-Blöcken oder nativem Code sollten wegen der negativen Auswirkungen von Virtual Thread Pinning also grundsätzlich vermieden werden. Um problematische Aufrufe zu identifizieren, kann die JVM mit der Option `-Djdk.tracePinnedThreads=full` oder `-Djdk.tracePinnedThreads=short` gestartet werden. Beide führen dazu, dass blockierende Aufrufe in gepinnten virtuellen Threads einen Stacktrace loggen. Während mit dem Wert `full` der gesamte Stacktrace zum Zeitpunkt des blockierenden Aufrufs ausgegeben wird, sind es mit `short` nur die Stackframes, die einem nativen Aufruf entsprechen oder sich gerade in einem `synchronized`-Block befinden – also den Verursachern des Virtual Thread Pinning.

Fazit

Virtuelle Threads sind ein wichtiges neues JDK-Feature, mit dem sich Performance und Skalierbarkeit von Java-Anwendungen in Zukunft noch einmal deutlich verbessern lassen, ohne dass dafür ein Paradigmenwechsel im Anwendungscode erforderlich wäre. Viele Frameworks und Laufzeitumgebungen (zum Beispiel Spring, Apache Tomcat und Eclipse Jetty) sind bereits auf den Einsatz mit virtuellen Threads vorbereitet. Wie die Auswirkungen speziell auf konkurrierende Ansätze wie reaktive Frameworks aussehen werden, steht derzeit allerdings noch in den Sternen. In jedem Fall bringen virtuelle Threads frischen Wind in die Entwicklung nebenläufiger Java-Anwendungen.

Links

[Continuation] https://de.wikipedia.org/wiki/Continuation
[JEP444] https://openjdk.org/jeps/444
[ProjectLoom] https://wiki.openjdk.org/display/loom/Main