



Die verborgene Kunst der Thread-sicheren Programmierung: Erkundung von java.util.concurrent

Dr Heinz M. Kabutz

Last Updated 2025-11-02

Eine Geschichte von java.util.Vector

- **Eine der ersten Klassen in Java**
 - DIE Liste in Java 1.0
- **Ursprünglich entworfen als threadsichere Liste**
 - Die meisten Methoden sind mit „synchronized“ versehen
 - Sperren also auf „this“
 - Allerdings fehlte die Synchronisierung bei einigen nur lesenden Methoden
 - wie z.B. size()

Java 1.0 Vector

- **size() konnte veraltete Werte zurückliefern**

```
public class Vector1_0 {  
    protected int elementCount;  
  
    public final int size() {  
        return elementCount;  
    }  
  
    public final synchronized void addElement(Object obj) {  
        // ...  
    }  
}
```

Moving to Java 1.1

- **Führte eine mögliche Race-Condition ein**

```
public class Vector1_1 implements java.io.Serializable {  
    protected int elementCount;  
  
    public final int size() {  
        return elementCount;  
    }  
  
    public final synchronized void addElement(Object obj) {  
        // ...  
    }  
}
```

Moving to Java 1.4

- **Behebung von Sichtbarkeitsproblem von size()**
 - Plus Race-Condition bei Serialisierung ist auch behoben

```
public class Vector1_4 implements java.io.Serializable {  
    protected int elementCount;  
    public synchronized int size() {  
        return elementCount;  
    }  
    public synchronized void addElement(Object obj) {  
        // ...  
    }  
    private synchronized void writeObject(ObjectOutputStream s)  
        throws IOException {  
        s.defaultWriteObject();  
    }  
}
```

Aufgepasst, Java 1.4 kann zu Deadlock führen!

- **Häufig führt Behebung eines Fehlers zu neuen Problemen**

```
Vector v1 = new Vector();  
Vector v2 = new Vector();  
v1.addElement(v2);  
v2.addElement(v1);  
// serialize v1 and v2 from two different threads
```

- **Erwähnt im The Java Specialists' Newsletter Nr. 184**

- <https://www.javaspecialists.eu/archive/Issue184.html>

Wechsel zu Java 1.7

- **Deadlock wurde behoben, indem writeFields() außerhalb des „synchronized“ aufgerufen wurde**

```
public class Vector1_7 implements Serializable {  
    private void writeObject(java.io.ObjectOutputStream s)  
        throws java.io.IOException {  
        final java.io.ObjectOutputStream.PutField fields = s.putFields();  
        final Object[] data;  
        synchronized (this) {  
            fields.put("capacityIncrement", capacityIncrement);  
            fields.put("elementCount", elementCount);  
            data = elementData.clone();  
        }  
        fields.put("elementData", data);  
        s.writeFields();  
    }  
}
```

Neuer potenzieller Deadlock in Java 8 eingeführt

- „Externe Methoden“ wie `accept()` sollten nicht innerhalb eines Locks aufgerufen werden

```
public class Vector8<E> implements Serializable {  
    public synchronized void forEach(Consumer<? super E> action) {  
        Objects.requireNonNull(action);  
        final int expectedModCount = modCount;  
        final E[] elementData = (E[]) this.elementData;  
        final int elementCount = this.elementCount;  
        for (int i=0; modCount == expectedModCount && i < elementCount; i++) {  
            action.accept(elementData[i]);  
        }  
        if (modCount != expectedModCount) {  
            throw new ConcurrentModificationException();  
        }  
    }  
}
```

Erkenntnisse aus den Vector-Fehlern

- **Thread-Sicherheit ist subtil**
- **Tests decken nicht immer Nebenläufigkeitsfehler auf**
 - **Wir müssen wissen, wonach wir suchen**



Überblick über `java.util.concurrent`

Korrekt Thread-sicheren Code zu schreiben ist schwierig

- **Das Java Memory Model (JMM) ist unser Regelwerk**
 - *happens-before*, Reihenfolge, Zugriffssicherheit usw.
 - wir können nicht vollständig testen, ob eine Klasse dem JMM entspricht
- **Wir führen unseren Code aus und hoffen, dass er korrekt funktioniert**
 - Manche Fehler sind äußerst schwer zu entdecken

LockSupport: Seltener Verlust von unpark()

- **Bug 8074773**

- In JDK 7 konnte das Laden von Klassen den unpark()-Aufruf „verschlucken“
 - Äußerst schwer zu diagnostizieren; erforderte eine Woche CPU-Zeit zur Analyse
 - Empfohlene Umgehungslösung: LockSupport frühzeitig laden

```
static {  
    // Prevent rare disastrous classloading in first call to LockSupport.park.  
    // See: https://bugs.openjdk.java.net/browse/JDK-8074773  
    Class<?> ensureLoaded = LockSupport.class;  
}
```

- Seit JDK 9 sorgt ConcurrentHashMap dafür, dass LockSupport geladen wird
 - Oder?

Wieso wir die java.util.concurrent Klassen studieren

- **Brian Goetz, Java Concurrency in Practice:**
 - Wenn man eine zustandsabhängige Klasse implementieren muss, ist die beste Strategie meist, auf bestehenden Bibliotheksklassen wie Semaphore, BlockingQueue oder CountDownLatch aufzubauen.
- **Durch detaillierte Untersuchung von java.util.concurrent lernen wir:**
 - Welche Bausteine verfügbar sind
 - Wie man robuste, thread-sichere Klassen schreibt

Guter vs. schlechter Code

- **Wir alle machen Fehler**
 - Auf Deutsch sagen wir: „Vertrauen ist gut, Kontrolle ist besser!“
 - Test Driven Development
 - Sehr schwierig bei Multi-Thread-Code
 - Java Concurrency Stress kann helfen: <https://github.com/openjdk/jcstress>
- **Besser ist es, sich auf bewährte Synchronisierer zu verlassen**
 - Und jene zu nutzen, die am häufigsten eingesetzt werden
 - Bevorzuge ConcurrentHashMap gegenüber ConcurrentSkipListMap
 - Bevorzuge LinkedBlockingQueue gegenüber LinkedBlockingDeque

Fehlerberichte beitragen

- **Jeder kann Java-Fehler melden: <https://bugreport.java.com>**
 - Ich habe einige eingereicht: javaspecialists.eu/about/jdk-contributions/
 - Die meisten betrafen selten verwendete Klassen:
 - 1 in `LinkedTransferQueue` (behoben in Java 1.8.0+70)
 - 1 in `ThreadLocalRandom` (behoben in Java 21+9)
 - 1 in `ConcurrentSkipListMap` (behoben in Java 24)
 - 1 in `ArrayBlockingQueue` (behoben in Java 24)
 - 5 in `LinkedBlockingDeque` (alle behoben in Java 26)
- **Je weniger eine Klasse genutzt wird, desto höher ist die Fehlerwahrscheinlichkeit**

„Eat Your Own Dogfood“ Collections

- **Wie oft finden wir Instanzen der Collection Klassen im JDK:**
 - **213 × ConcurrentHashMap**
 - **11–24 × CopyOnWriteArrayList, ConcurrentLinkedQueue, ConcurrentLinkedDeque, FutureTask, LinkedBlockingQueue**
 - **2–6 × CountdownLatch, ArrayBlockingQueue, SynchronousQueue, ConcurrentSkipListSet**
 - **1 × ConcurrentSkipListMap, LinkedBlockingDeque, LinkedTransferQueue, Semaphore**
 - **0 × CopyOnWriteArraySet, CyclicBarrier, Exchanger, Phaser, PriorityBlockingQueue**

Wiederholen wir das noch einmal

- **Verwende möglichst häufig genutzte thread-sichere Klassen:**
 - **ConcurrentHashMap**
 - **LinkedBlockingQueue**
 - **ConcurrentLinkedQueue**
- **Ich fand Fehler ausschließlich in selten genutzten Klassen**

Bevor wir weitermachen ...

- **Hol dir unseren Kurs Data Structures in Java hier:**
 - **tinyurl.com/wjax25**
 - **Gutschein gilt nur heute bis 18:30 Uhr**
 - **Lebenslanger Zugriff auf den Kurs**
 - **Versuche ihn bis Ende Dezember abzuschließen**





Lektionen aus `Striped64`

LongAdder vs AtomicLong

- Kurzer Vergleich von 100 Millionen Inkrementierungen
 - Gegenüberstellung von AtomicLong und LongAdder (Striped64)

```
IntStream.range(0, 100_000_000)
    .parallel()
    .forEach(_ -> atomicLong.getAndIncrement());
```

tinyurl.com/wjax25

```
IntStream.range(0, 100_000_000)
    .parallel()
    .forEach(_ -> longAdder.increment());
```



Demo

- **Magie? Sehen wir uns an, wie LongAdder / Striped64 funktioniert.**

tinyurl.com/wjax25



Zentrale Erkenntnis

- **Der beste Umgang mit Contention ist, sie zu vermeiden.**

tinyurl.com/wjax25



Wandelnde Hardware- Landschaft

Wandelnde Hardware-Landschaft

- **Ich begann 1997 mit Java zu programmieren**
 - 64 MB RAM, ein Kern, 233 MHz, 32 Bit
 - Und das war einer der besseren Rechner im Unternehmen
 - Aktuelles Notebook hat 96 GB RAM, 12 Kerne, 38 GPU-Kerne, 3,7 GHz, 64 Bit
- **Speicher war knapp**
 - Man konnte sich Collections mit Milliarden von Einträgen nicht vorstellen
 - Es gab nur Platform Threads – begrenzt auf einige Tausend

Fehler an den Grenzen

- **Unmengen an Speicher und virtuellen Threads heutzutage**
 - Fehler in `LinkedBlockingDeque` ermöglichte zu viele Einträge
 - `size()` gab einen negativen Wert zurück
 - <https://www.javaspecialists.eu/archive/Issue328.html> – behoben in Java 26
 - Fehler in `ReentrantReadWriteLock` führte nach 65 536 Read-Locks zu einem Error
 - Hat einen Error geworfen
 - Ursache: Erschöpfung der Read-Locks
 - Behoben in Java 25
- **Demo: ManyReadLocks**



StartingGun-Synchronisierer

StartingGun-Synchronisierer

- **Angenommen, wir haben einen Dienst, der Zeit benötigt, zu starten**
 - Alle abhängigen Systemteile sollen warten, bis er bereit ist
 - Wir möchten jedoch nicht mit InterruptedException umgehen
 - Sobald Daten initialisiert sind, rufen wir ready() auf, wartende Threads zu wecken

```
public interface StartingGun {  
    void awaitUninterruptibly();  
    void ready();  
}
```

Verwendung von **synchronized** und **wait()/notifyAll()**

```
public class StartingGunMonitor implements StartingGun {  
    private boolean ready = false;  
    public synchronized void awaitUninterruptibly() {  
        boolean interrupted = Thread.interrupted();  
        while (!ready) {  
            try {  
                wait(); // not fully compatible with older Loom versions  
            } catch (InterruptedException e) {  
                interrupted = true;  
            }  
        }  
        if (interrupted) Thread.currentThread().interrupt();  
    }  
    public synchronized void ready() { ready = true; notifyAll(); }  
}
```

Aufbau von StartingGun auf CountDownLatch

```
public class StartingGunCountDownLatch implements StartingGun {  
    private final CountDownLatch latch = new CountDownLatch(1);  
    public void awaitUninterruptibly() {  
        var interrupted = Thread.interrupted();  
        while (true) {  
            try {  
                latch.await();  
                break;  
            } catch (InterruptedException e) {  
                interrupted = true;  
            }  
        }  
        if (interrupted) Thread.currentThread().interrupt();  
    }  
    public void ready() { latch.countDown(); }  
}
```

Probleme bei diesen Ansätzen

- **synchronized/wait()** nicht vollständig virtuellen Threads kompatibel
 - Behoben in Java 24
- In beiden Fällen führte eine Unterbrechung zu **InterruptedException**
 - Wir fangen sie ab, zahlen aber dennoch den Preis der Exception-Erzeugung
- Eine weitere Möglichkeit besteht darin, das Verhalten von **CountDownLatch** direkt zu imitieren
 - Kurze Demo folgt

Lock-Splitting am Beispiel von `LinkedBlockingQueue`

Aufbau von `LinkedBlockingQueue`

- **Single lock would cause `put()/take()` contention**
- **Has separate `putLock` and `takeLock` `ReentrantLock`**
 - We can `put()` and `take()` from a single queue at the same time
 - Has higher throughput for the SPSC case
 - And surprises for the SPMC case
 - Subtleties regarding visibility due to two locks
 - Use `AtomicInteger` count as a volatile synchronizer
- **Demo `LockSplittingDemo`**

Aufbau von `LinkedBlockingQueue`

- Ein einzelner Lock würde bei `put()/take()` zu Contention führen
- Es gibt getrennte `putLock`- und `takeLock`-Instanzen (`ReentrantLock`)
 - Dadurch kann gleichzeitig `put()` und `take()` auf derselben Warteschlange erfolgen
 - Höherer Durchsatz im SPSC-Fall (Single-Producer/Single-Consumer)
 - Überraschungen im SPMC-Fall (Single-Producer/Multi-Consumer)
- Feinheiten der Sichtbarkeit wegen zweier Locks
 - `AtomicInteger count` dient als volatile-Synchronisierer
- Demo: `LockSplittingDemo`

Schwach konsistente Iteratoren – `ArrayBlockingQueue`

Zirkuläre array-basierte Warteschlange ArrayBlockingQueue

● Schwach konsistente Iteration

- ArrayDeque würde hier eine ConcurrentModificationException auslösen

```
var queue = new ArrayBlockingQueue<Integer>(10);  
Collections.addAll(queue, 1, 2, 3, 4, 5);  
var iterator = queue.iterator();  
for (var i = 0; i < 3; i++) System.out.println(iterator.next()); // 1, 2, 3  
Collections.addAll(queue, 6, 7, 8, 9, 10);  
iterator.forEachRemaining(System.out::println); // 4, 5, 6, 7, 8, 9, 10
```

- Was passiert jedoch, wenn wir den gesamten Array-Zyklus durchlaufen?

- ArrayBlockingQueue muss ihre aktuellen Iteratoren informieren

- Aber wie geschieht das?

- Demo: WeaklyConsistentViaWeakReferences



Double-Checked-Locking – CopyOnWriteArrayList

CopyOnWriteArrayList – Double-Checked-Locking

- In `remove(Object)` werden Prüfungen vor dem Sperren durchgeführt

```
public boolean remove(Object o) {  
    Object[] snapshot = getArray();  
    int index = indexOfRange(o, snapshot, 0, snapshot.length);  
    return index >= 0 && remove(o, snapshot, index);  
}  
// also addIfAbsent(E e),
```

- Demo: DCLOnSteroidsCOWDemo

Schlussfolgerung

The Java Specialists' Newsletter

- **Werde Teil unserer Java Specialists-Community**
 - www.javaspecialists.eu/archive/subscribe/
- **Leser in über 150 Ländern**
- **25 Jahre Newsletter über fortgeschrittene Java-Themen**
 - Alle bisherigen Ausgaben unter <https://www.javaspecialists.eu>
 - Längst laufender Java-Newsletter der Welt



Nicht vergessen ...

- **Hol dir unseren Kurs Data Structures in Java**
 - **`tinyurl.com/wjax25`**
 - **Gutschein gilt nur heute bis 18:30 Uhr**
 - **Lebenslanger Zugriff auf den Kurs**
- **Für alle, die die Aufzeichnung ansehen:**
 - **Melde dich beim Java Specialists' Newsletter an**
 - **<https://www.javaspecialists.eu>**
 - **Antworte auf die Willkommens-E-Mail, dass du den Kurs erhalten möchtest**

