Thread-Synchronisation in Java

Lass uns in die Thread-Synchronisation in Java eintauchen! Wenn mehrere Threads auf gemeinsame Ressourcen (wie Variablen, Sammlungen oder Dateien) zugreifen, müssen Sie Rennebedingungen verhindern –bei denen Threads sich gegenseitig stören und zu unvorhersehbaren Ergebnissen führen. Java bietet mehrere Werkzeuge dafür: `synchronized`, `Lock` und `Semaphore`. Ich werde jedes davon erklären, Beispiele zeigen und sie vergleichen, damit Sie das Richtige für Ihre Bedürfnisse auswählen können.

---

1. `synchronized` Schlüsselwort

Das `synchronized`-Schlüsselwort stellt sicher, dass nur ein Thread einen Codeblock oder eine Methode zur gleichen Zeit ausführen kann. Es ist das einfachste Synchronisationswerkzeug, das in den Kern von Java eingebaut ist.

So funktioniert es

- Sperrt den Monitor eines Objekts (jeder Java-Objekt hat einen).
- Andere Threads warten, bis die Sperre freigegeben wird.

Beispiel: Zähler mit `synchronized`

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
```

```
                counter.increment();
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Endgültiger Zählerstand: " + counter.getCount()); // Immer 2000
    }
}
```

- Ohne `synchronized` könnte `count++` (lesen-modifizieren-schreiben) überlappen und ein Ergebnis kleiner als 2000 liefern.
- `synchronized` sperrt die `Counter`-Instanz und stellt atomare Aktualisierungen sicher.

Synchronisierter Block Sie können auch spezifischen Code sperren, anstatt eine ganze Methode:

```
public void increment() {
    synchronized (this) {
        count++;
    }
}
```

- Verwenden Sie ein anderes Objekt als Sperre, wenn Sie eine feinere Kontrolle wünschen:

```
private final Object lock = new Object();
public void increment() {
    synchronized (lock) {
        count++;
    }
}
```

Vor- und Nachteile

- **Vorteile**: Einfach, eingebaut, gut für grundlegende Anwendungen.
- **Nachteile**: Keine Flexibilität (z. B. kann einen wartenden Thread nicht unterbrechen), blockiert bis zur Freigabe.

2. `Lock`-Schnittstelle (java.util.concurrent.locks)

Die `Lock`-Schnittstelle (z. B. `ReentrantLock`) bietet mehr Kontrolle als `synchronized`. Sie ist Teil des `java.util.concurrent`-Pakets und wurde eingeführt, um die Einschränkungen von `synchronized` zu beheben.

Hauptmerkmale

- Explizite `lock()`- und `unlock()`-Aufrufe.
- Unterstützt Try-Locks, Timeouts und unterbrechbare Sperren.
- Fairness-Option (Thread warten in Reihenfolge).

Beispiel: Zähler mit `ReentrantLock`

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

class Counter {
    private int count = 0;
    private final Lock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock(); // Immer in finally entsperren
        }
    }

    public int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
```

```
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Endgültiger Zählerstand: " + counter.getCount()); // Immer 2000
    }
}
```

- `try-finally` stellt sicher, dass die Sperre auch bei einem Ausnahmefall freigegeben wird.

Fortgeschrittene Funktionen

- **Try Lock**: Nicht blockierender Versuch, die Sperre zu erlangen:

```
if (lock.tryLock()) {
    try {
        count++;
    } finally {
        lock.unlock();
    }
} else {
    System.out.println("Konnte Sperre nicht erlangen");
}
```

- **Timeout**: Warten Sie eine begrenzte Zeit:

```
if (lock.tryLock(1, TimeUnit.SECONDS)) { ... }
```

- **Unterbrechbar**: Erlauben Sie wartenden Threads, unterbrochen zu werden:

```
lock.lockInterruptibly();
```

Vor- und Nachteile

- **Vorteile**: Flexibel, unterstützt fortschrittliche Funktionen, explizite Kontrolle.
- **Nachteile**: Ausführlicher, manuelles Entsperren erforderlich (Gefahr des Vergessens).

---

3. `Semaphore`

Ein `Semaphore` steuert den Zugriff auf eine Ressource, indem er einen Satz von Genehmigungen verwaltet. Er ist ideal, um die Gleichzeitigkeit zu begrenzen (z. B. maximal 5 Threads können auf eine Ressource zugreifen).

So funktioniert es

- Threads erlangen Genehmigungen mit `acquire()`.
- Genehmigungen werden mit `release()` freigegeben.
- Wenn keine Genehmigungen verfügbar sind, warten die Threads.

Beispiel: Begrenzung der Datenbankverbindungen

```
import java.util.concurrent.Semaphore;

class ConnectionPool {
    private final Semaphore semaphore = new Semaphore(3); // Max 3 Verbindungen

    public void connect() {
        try {
            semaphore.acquire();
            System.out.println(Thread.currentThread().getName() + " verbunden");
            Thread.sleep(1000); // Arbeit simulieren
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            semaphore.release();
            System.out.println(Thread.currentThread().getName() + " getrennt");
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        ConnectionPool pool = new ConnectionPool();
        Runnable task = () -> pool.connect();

        Thread[] threads = new Thread[10];
        for (int i = 0; i < 10; i++) {
            threads[i] = new Thread(task, "Thread-" + i);
            threads[i].start();
        }
    }
}
```

- Nur 3 Threads können sich gleichzeitig „verbinden“; andere warten, bis eine Genehmigung freigegeben wird.

Fortgeschrittene Funktionen

- **Fairness**: `new Semaphore(3, true)` stellt FIFO-Reihenfolge sicher.
- **Try Acquire**: `semaphore.tryAcquire()` überprüft ohne Blockierung.

Vor- und Nachteile

- **Vorteile**: Ideal für Ressourcenpools, flexibles Genehmigungssystem.
- **Nachteile**: Komplexer als `synchronized` oder `Lock`, nicht für einfache gegenseitige Ausschlüsse.

Kombination mit `ExecutorService`

Hier ist, wie diese Werkzeuge mit einem Thread-Pool funktionieren:

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

class SharedResource {
    private int value = 0;
    private final Lock lock = new ReentrantLock();

    public void increment() {
```

```
        lock.lock();
        try {
            value++;
        } finally {
            lock.unlock();
        }
    }

    public int getValue() {
        return value;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        SharedResource resource = new SharedResource();
        ExecutorService executor = Executors.newFixedThreadPool(4);

        for (int i = 0; i < 10; i++) {
            executor.submit(() -> {
                for (int j = 0; j < 100; j++) {
                    resource.increment();
                }
            });
        }

        executor.shutdown();
        executor.awaitTermination(5, TimeUnit.SECONDS);
        System.out.println("Endgültiger Wert: " + resource.getValue()); // Immer 1000
    }
}
```

Vergleich

Merkmal	`synchronized`	`Lock` (ReentrantLock)	`Semaphore`
Einfachheit	Hoch	Mittel	Niedrig
Flexibilität	Niedrig	Hoch	Mittel

Merkmal	`synchronized`	`Lock` (ReentrantLock)	`Semaphore`
Granularität	Methode/Block	Explizit	Genehmigungsbasiert
Unterbrechbar	Nein	Ja	Ja
Fairness-Option	Nein	Ja	Ja
Anwendungsfall	Einfache gegenseitige Ausschlüsse	Fortgeschrittene Kontrolle	Begrenzung des Ressourcenzugangs

---

Wann was verwenden?

- `synchronized`: Schnell, einfache gegenseitige Ausschlüsse (z. B. Schutz einer einzelnen Variablen).
- `Lock`: Benötigen Try-Locks, Timeouts oder Unterbrechbarkeit (z. B. komplexe Arbeitsabläufe).
- `Semaphore`: Begrenzung des gleichzeitigen Zugriffs auf eine Ressource (z. B. Verbindungs-Pools, Rate-Limiting).