

8. Applying Thread Pools

Motivation:

Richtiger Einsatz und Konfiguration eines Thread Pools

Inhaltsverzeichnis

8.1 Aufgaben und Ausführungsrichtlinien.....	3
8.2 Bestimmung der Größe von Thread Pools.....	9
8.3 Konfiguration von ThreadPoolExecutor.....	11
8.4 Erweiterung des ThreadPoolExecutors.....	23
8.5 Parallelisierung.....	25

8.1 Aufgaben und Ausführungsrichtlinien

- Nicht jeder Aufgabentyp ist mit jeder Ausführungsrichtlinie kompatibel.

Verschieden Aufgabentypen:

Unabhängige Aufgaben:

- für Thread Pools am einfachsten zu handhaben.
- können jederzeit gestartet und ausgeführt werden.
- die Größe und Konfiguration des Pools ist beliebig anpassbar; nur Einfluss auf die Performance.

Abhängige Aufgaben:

- wenn Aufgaben von den Seiteneffekten von anderen Aufgaben abhängen, muss dies bei der Konfiguration des Thread Pools beachtet werden.
- meistens hilft unbeschränkte Anzahl von Threads.

Verschieden Aufgabentypen (2):

Aufgaben, die sich auf „Thread confinement“ verlassen:

- Single-Threaded Executor stellt sicher, dass nur ein Thread gleichzeitig Aufgaben ausführt.
- Objekte, auf die in der Aufgabe zugegriffen wird, müssen nicht „Thread-safe“ sein.

Aufgaben die “ThreadLocal” verwenden:

- `ThreadLocal`: jeder Thread hat seine eigene lokale Variable
- Executor garantiert nicht, dass ein Thread eine Aufgabe ausführt; erzeugt oder beseitigt oft zu Gunsten der Performance neue Threads
- `ThreadLocal` Variablen nur sinnvoll, wenn sichergestellt ist, dass die Lebensdauer einer Variable an die Aufgabe gebunden ist; nicht für die Kommunikation zwischen Threads geeignet.

Verschieden Aufgabentypen (3):

Zeit-intensive Aufgaben:

- Thread Pool kann mit zeit-intensiven Aufgaben gefüllt sein;
=> kurze aber evtl. wichtigere Aufgaben können nicht ausgeführt werden
- Abhilfe: Aufgaben zeitlich begrenzen, z. B. mit Zeit-begrenzten Versionen blockierender Methoden, wie `Thread.join`, `BlockQueue.put`, `CountDownLatch.await` `Selector.select`
- nach Timeout: nicht abgeschlossene Aufgaben entweder abbrechen, oder wieder hinten in der Warteschlange einreihen.
- stellt kurze Wartezeiten für Aufgaben sicher.

Thread Stillstand durch Thread starvation deadlock

- alle Threads des Thread Pools sind blockiert.
- warten auf die Ausführung von anderen Aufgaben
- diese können nicht ausgeführt werden, weil keine Threads mehr verfügbar sind.
- **einfachstes Beispiel:** Single-Threaded Executor setzt in einer Aufgabe eine andere Aufgabe in die Warteschlange und wartet auf deren Ergebnis.
- Abhilfe: Anzahl der maximalen Threads, sehr hoch setzen oder nicht beschränken
- Berücksichtigung von anderen Einschränkungen für die maximale Anzahl an Threads, z. B. Datenbankverbindungen

```
ExecutorService exec = Executors.newSingleThreadExecutor();
```

```
...
```

```
public class RenderPageTask implements Callable<String> {  
    public String call() throws Exception {  
        Future<String> header, footer;  
        header = exec.submit(new LoadFileTask("header.html"));  
        footer = exec.submit(new LoadFileTask("footer.html"));  
        String page = renderBody();  
        // Will deadlock -- task waiting for result of subtask  
        return header.get() + page + footer.get();  
    }  
}
```

8.2 Bestimmung der Größe von Thread Pools

- Größe hängt von den Typen der Aufgaben und von der Hardware des Systems ab.
- generell nicht statisch festlegen, sondern zur Laufzeit dynamisch berechnen, z. B. mit Hilfe von `Runtime.availableProcessors`.
- keine diskrete korrekte Größe.
- Faktoren für die Größe eines Thread Pools:
 - Anzahl der Prozessoren
 - Größe des Arbeitsspeichers
 - Charakteristik der Aufgaben: Berechnungen, E/A, oder beides kombiniert.
 - andere einschränkende Größen, wie maximale Anzahl an Datenbankverbindungen

Bestimmung der Größe von Thread Pools (2)

- bei reinen Rechenoperationen: $N_{threads} = N_{cpu} + 1$, +1: falls Fehler auftreten.
- generelle Formel für die Größe des Pools:

$$N_{threads} = N_{cpu} * U_{cpu} * (1 + \frac{W}{C})$$

N_{cpu} = Anzahl der Prozessoren

U_{cpu} = Gewünschte Prozessorauslastung , $0 \leq U_{cpu} \leq 1$

$\frac{W}{C}$ = Verhältnis zwischen Wartezeit und Rechenzeit

- für andere Ressourcen: Benötigte Größe der Ressource pro Thread durch gesamt verfügbare Ressource
- alternativ: Benchmarks mit verschiedenen Größen
- bei verschiedenen Kategorien von Aufgaben:
oft sinnvoll: jeder Kategorie einen eigenen Thread Pool zuweisen.

Konfiguration von ThreadPoolExecutor (2)

- `corePoolSize`: Zielgröße des Pools;
- `maximumPoolSize`: maximale Größe des Pools
- `keepAliveTime`: Zeit, nach der ein Thread beendet werden kann, wenn die momentane Größe des Pools die `corePoolSize` übersteigt.
- **Bsp:** `newFixedThreadPool`: setzt `corePoolSize` und `maximumPoolSize` fix => kein Timeout
- **Bsp:** `newCachedThreadPool`: `corePoolSize` auf 0, `maximumPoolSize` auf `Integer.MAX_VALUE`, `keepAliveTime` 1min => quasi unbeschränkter Thread Pool, der je nach Bedarf Threads erzeugt und beseitigt.
- Gefahr: ständige Thread Erzeugung und Beseitigung kann zu Lasten der Performance gehen.

Verwaltung von wartenden Aufgaben

- `BlockingQueue` verwaltet die wartenden Aufgaben eines Thread Pools.
- Gefahr: bei zu langsamer Ausführung: Warteschlange füllt sich bis Arbeitsspeicher voll.

Verwaltung von wartenden Aufgaben (2)

unbegrenzte BlockingQueues:

- z. B unbegrenzte `LinkedBlockingQueue`
- Standard für `newFixedThreadPool` und `newSingleThreadExecutor`
- Gefahr: unbegrenztes Wachstum.

begrenzte BlockingQueues:

- z. B begrenzte `LinkedBlockingQueue`, `ArrayBlockingQueue` oder `PriorityBlockingQueue`
- stabilere Warteschlange, aber führt zu der Frage: Wohin mit den blockierten Aufgaben? → Saturation Policies

Verwaltung von wartenden Aufgaben (3)

synchrones Weiterreichen (`SynchronousQueue`):

- wird in `newCachedThreadPool` verwendet
- es muss immer ein Arbeiter-Thread für die Übergabe einer Aufgabe bereit sein, oder ein neuer Thread wird ggf. erzeugt
- effizienteste Methode: umgeht Datenstruktur der Warteschlange
- sinnvolle Wahl, wenn Größe des Pools unbegrenzt, und es akzeptabel ist, dass Aufgaben zurückgewiesen werden.

Ausführungsreihenfolge:

- **FIFO:** `LinkedBlockingQueue` und `ArrayBlockingQueue`: die Aufgaben werden der Reihe nach, nach der sie angekommen sind ausgeführt.
- `PriorityBlockingQueue`: Aufgaben werden je nach Priorität ausgeführt.

Zusammenfassung:

- in den meisten Fällen beste Wahl: `newCachedThreadPool`
- in Server Applikationen oft `newFixedThreadPool` sinnvoll, denn bessere Kontrolle über Ressourcen
- Thread pool und Queue zu begrenzen, ist nur bei unabhängigen Aufgaben zu empfehlen.

Saturation policies

- werden gebraucht, wenn Aufgaben zurückgewiesen werden
- beim `ThreadPoolExecutor` `setRejectedExecutionHandler`
- verschiedene Implementierungen des `RejectedExecutionHandler`:
 - `AbortPolicy`: teilt dem Produzent der Aufgabe in Form einer `RejectedExecutionException` den Abbruch mit.
 - `DiscardPolicy`: verwirft die Aufgabe stillschweigend.
 - `DiscardOldestPolicy`: verwirft die Aufgabe, die als nächstes ausgeführt werden würde.
Achtung: sollte nicht bei einer `PriorityBlockingQueue` eingesetzt werden.
 - `CallerRunsPolicy`: versucht die Aufgabe dem Produzenten zurückzugeben.
- keine vorgefertigte Richtlinie, die nur blockiert.
Aber: Realisierung durch Semaphore, welcher die Erzeugungsrate von Aufgaben kontrolliert.

```

@ThreadSafe
public class BoundedExecutor {
    private final Executor exec;
    private final Semaphore semaphore;

    public BoundedExecutor(Executor exec, int bound) {
        this.exec = exec;
        this.semaphore = new Semaphore(bound);
    }

    public void submitTask(final Runnable command) throws
        InterruptedException {
        semaphore.acquire();
        try {
            exec.execute(new Runnable() {
                public void run() {
                    try {
                        command.run();
                    } finally {
                        semaphore.release();
                    }
                }
            });
        } catch (RejectedExecutionException e) {semaphore.release();}
    }
}

```

Thread Factories

- Thread Erzeugung im Thread Pool durch eine Thread Factory.
- Gründe für die Benutzung einer eigenen Thread Factory:
 - benutzen eines `UncaughtExceptionHandler`
 - benutzen einer eigenen Thread Klasse mit Event Logger
 - Anpassung der Priorität oder des “daemon” Status (i. A. nicht empfohlen)
 - eigene Benennung der Threads für Debugging Zwecke.
 - `privilegedThreadFactory`: erzeugt Threads, welche dieselben Rechte haben, wie der Thread der die Fabrik erzeugt.

```
public class MyThreadFactory implements ThreadFactory {
    private final String poolName;

    public MyThreadFactory(String poolName) {
        this.poolName = poolName;
    }

    public Thread newThread(Runnable runnable) {
        return new MyAppThread(runnable, poolName);
    }
}
```

```

public class MyAppThread extends Thread {
    public static final String DEFAULT_NAME = "MyAppThread";
    private static final AtomicInteger created = new AtomicInteger();
    private static final AtomicInteger alive = new AtomicInteger();
    private static final Logger log = Logger.getAnonymousLogger();

    public MyAppThread(Runnable r) { this(r, DEFAULT_NAME); }

    public MyAppThread(Runnable runnable, String name) {
        super(runnable, name + "-" + created.incrementAndGet());
        setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {
            public void uncaughtException(Thread t, Throwable e) {
                log.log(Level.SEVERE, "UNCAUGHT in thread " + t.getName(), e);
            }
        });
    }

    public void run() {
        log.log(Level.FINE, "Created " + getName());
        try {
            alive.incrementAndGet();
            super.run();
        } finally {
            alive.decrementAndGet();
            log.log(Level.FINE, "Exiting " + getName());
        }
    }
}

```

Anpassung des ThreadPoolExecutors nach Konstruktion

- alle Parameter außer der `BlockingQueue` sind nach Konstruktion veränderbar.
- `unconfigurableExecutorService`: stellt sicher, dass nur noch die Methoden der Schnittstelle `ExecutorService` sichtbar sind. Executor deswegen unveränderbar.
- `newSingleThreadExecutor` benutzt diese Methode

8.4 Erweiterung des ThreadPoolExecutors

- `ThreadPoolExecutor` bietet Methoden, die bei Bedarf überschrieben werden können.
- `beforeExecute`, `afterExecute` und `terminated`: erlauben die Ausführung von Code zu diesen Zeitpunkten.
- z. B für Logging, Überwachung oder Statistik.

```

public class TimingThreadPool extends ThreadPoolExecutor {

    private final ThreadLocal<Long> startTime = new ThreadLocal<Long>();
    private final Logger log = Logger.getLogger("TimingThreadPool");
    private final AtomicLong numTasks = new AtomicLong();
    private final AtomicLong totalTime = new AtomicLong();

    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        log.fine(String.format("Thread %s: start %s", t, r));
        startTime.set(System.nanoTime());
    }

    protected void afterExecute(Runnable r, Throwable t) {
        try {
            long endTime = System.nanoTime();
            long taskTime = endTime - startTime.get();
            numTasks.incrementAndGet();
            totalTime.addAndGet(taskTime);
            log.fine(String.format("Thread %s: end %s, time=%dns", t, r, taskTime));
        } finally { super.afterExecute(r, t); }
    }

    protected void terminated() {
        try { log.info(String.format("Terminated: avg time=
                                     %dns",totalTime.get()/numTasks.get()));
        } finally { super.terminated(); }
    }
}

```

8.5 Parallelisierung

Parallelisierung von Sequentiellen Iterationen

- Durchläufe einer Iteration können als Aufgabe verpackt einem Executor zugewiesen werden.
- nur sinnvoll, wenn die einzelnen Durchläufe unabhängig, und die Dauer eines Durchlaufs es wert ist dafür eine Aufgabe anzulegen: komplexe Berechnungen oder E/A.

```
void processSequentially(List<Element> elements) {
    for (Element e : elements)
        process(e);
}

void processInParallel(Executor exec, List<Element> elements) {
    for (final Element e : elements)
        exec.execute(new Runnable() {
            public void run() {
                process(e);
            }
        });
}
```

Parallelisierung von rekursiven Algorithmen

- auch in rekursiven Algorithmen können sequentielle Schleifen auf dieselbe Weise parallelisiert werden.
- wichtig, dass ein Durchlauf nicht die Ergebnisse der rekursiven Durchläufe benötigt, welche er aufruft.

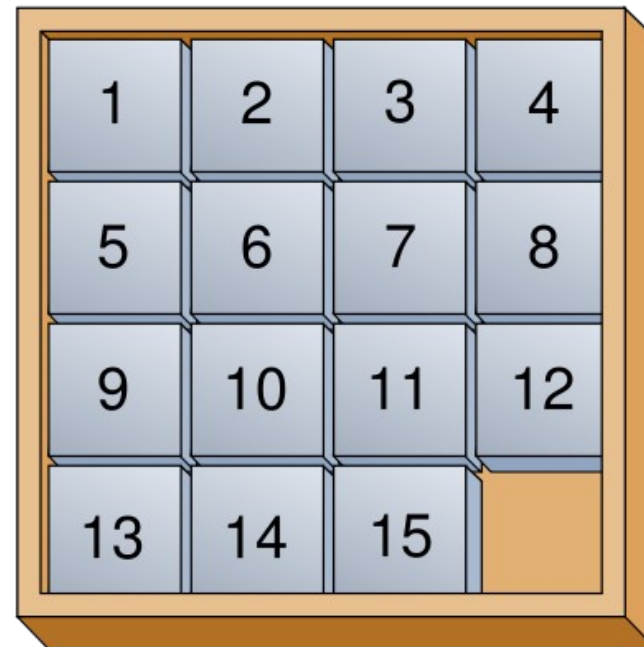
```
public <T> void sequentialRecursive(List<Node<T>> nodes, Collection<T> results)
{
    for (Node<T> n : nodes) {
        results.add(n.compute());
        sequentialRecursive(n.getChildren(), results);
    }
}
```

```
public <T> void parallelRecursive(final Executor exec, List<Node<T>> nodes,
                                final Collection<T> results) {
    for (final Node<T> n : nodes) {
        exec.execute(new Runnable() {
            public void run() {
                results.add(n.compute());
            }
        });
        parallelRecursive(exec, n.getChildren(), results);
    }
}
```

Beispiel: Puzzle Framework

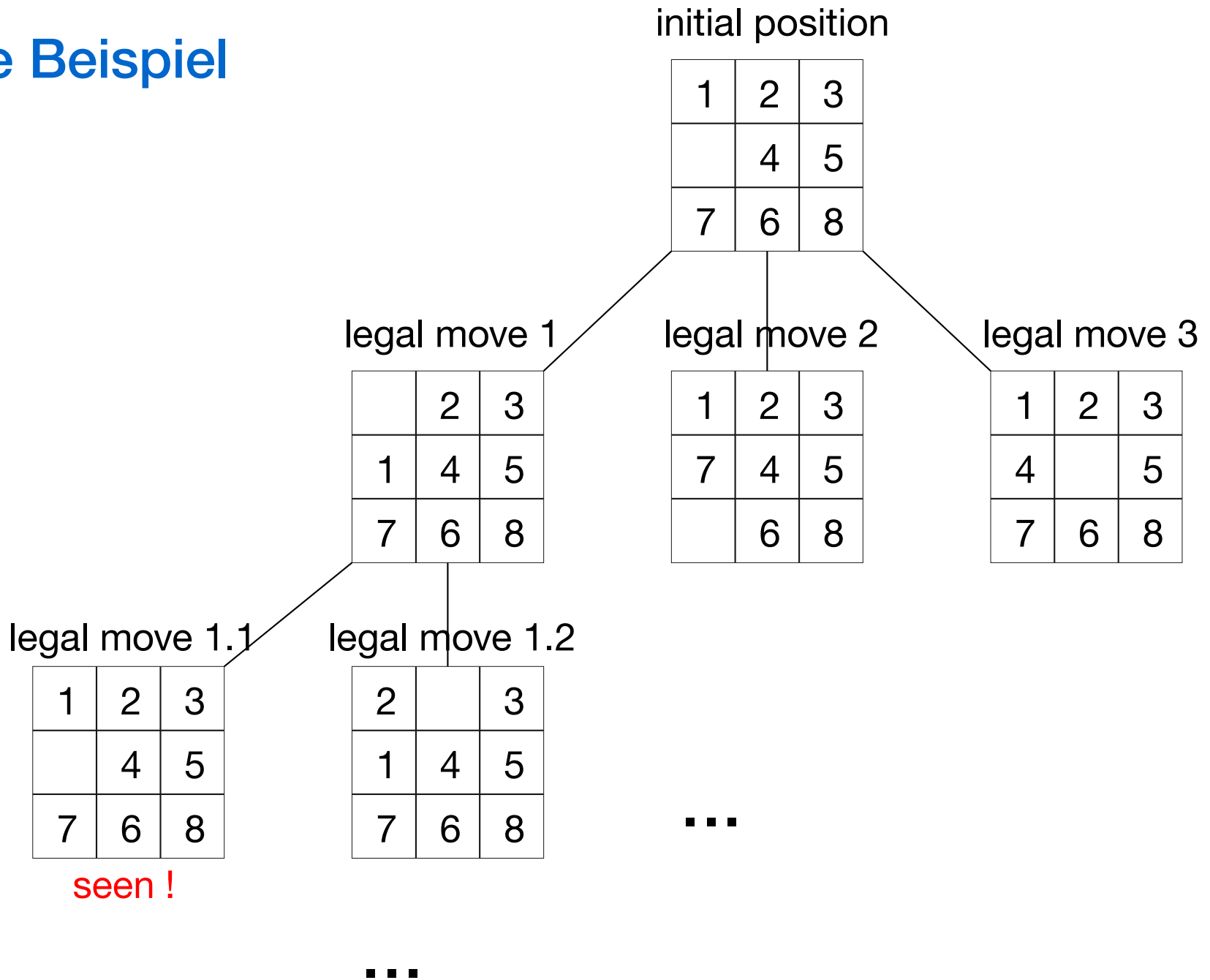
- **Definition:** Ein Puzzle hat ein Anfangszustand und einen Zielzustand und eine Menge von erlaubten Zügen zum Lösen des Puzzles.

```
public interface Puzzle <P, M> {  
    P initialPosition();  
    boolean isGoal(P position);  
    Set<M> legalMoves(P position);  
    P move(P position, M move);  
}
```



Sliding Block Puzzle

Puzzle Beispiel



SequentialPuzzleSolver:

- durchläuft Positionen durch erlaubte Züge und prüft ob der Zielzustand erreicht ist.
- hält eine Liste von gesehenen Positionen die nicht weiter überprüft werden müssen.
- PuzzleNode: repräsentiert eine Position, die durch eine Sequenz von erlaubten Zügen erreicht wurde.
- u. U. geringe Auslastung, da keine Nebenläufigkeit eingesetzt.

```

@Immutable
public class PuzzleNode <P, M> {
    final P pos;
    final M move;           // Zug, der die Position erzeugt hat
    final PuzzleNode<P, M> prev; // Node der vorherigen Position

    public PuzzleNode(P pos, M move, PuzzleNode<P, M> prev) {
        this.pos = pos;
        this.move = move;
        this.prev = prev;
    }

    List<M> asMoveList() { // Iteration der Züge, die zu dieser Node
                           geführt haben
        List<M> solution = new LinkedList<M>();
        for (PuzzleNode<P, M> n = this; n.move != null; n = n.prev)
            solution.add(0, n.move);
        return solution;
    }
}

```

```

public class SequentialPuzzleSolver <P, M> {
    private final Puzzle<P, M> puzzle;
    private final Set<P> seen = new HashSet<P>(); //Liste von gesehenen
                                                Positionen

    ...

    private List<M> search(PuzzleNode<P, M> node) {
        if (!seen.contains(node.pos)) {
            seen.add(node.pos);
            if (puzzle.isGoal(node.pos))
                return node.asMoveList();

            //Alle erlaubten Züge von der jetzigen Position ausgehend werden
            //iteriert
            for (M move : puzzle.legalMoves(node.pos)) {
                P pos = puzzle.move(node.pos, move);    //.
                PuzzleNode<P, M> child = new PuzzleNode<P, M>(pos, move, node);
                List<M> result = search(child); //rekursiver Aufruf der Suche
                if (result != null)
                    return result;
            }
        }
        return null;
    }
}

```

ConcurrentPuzzleSolver:

- Beschleunigung durch die gleichzeitige Auswertung von mehreren Zügen.
- weitgehend unabhängige Aufgaben.
- benutzt eine Klasse `SolverTask`, die `PuzzleNode` erweitert und `Runnable` implementiert.

```

protected class SolverTask extends PuzzleNode<P, M> implements Runnable {
    SolverTask(P pos, M move, PuzzleNode<P, M> prev) {
        super(pos, move, prev);
    }

    public void run() {
        // Puzzle schon gelöst oder Position schon gesehen
        if (solution.isSet() || seen.putIfAbsent(pos, true) != null)
            return;
        if (puzzle.isGoal(pos))
            solution.setValue(this);
        else
            // falls noch keine Lösung gefunden wurde, werden die
            // erlaubten Züge als neue Aufgaben dem executor aufgetragen
            for (M m : puzzle.legalMoves(pos))
                exec.execute(newTask(puzzle.move(pos, m), m, this));
    }
}

protected Runnable newTask(P p, M m, PuzzleNode<P,M> n) {
    return new SolverTask(p, m, n);
}

```

```

public class ConcurrentPuzzleSolver <P, M> {
    ...
    private final ConcurrentMap<P, Boolean> seen = new ConcurrentHashMap<P,
                                                                    Boolean>();
    protected final ValueLatch<PuzzleNode<P,M>> solution = new
                                                                    ValueLatch<PuzzleNode<P,M>>();

    public ConcurrentPuzzleSolver(Puzzle<P, M> puzzle) { ...
        exec.setRejectedExecutionHandler(new ThreadPoolExecutor.DiscardPolicy());
    }

    private ExecutorService initThreadPool(){return
        Executors.newCachedThreadPool();}

    public List<M> solve() throws InterruptedException {
        try { P p = puzzle.initialPosition();
            // bringt eine neue Aufgabe zur Ausführung, die wiederum neue
            // Aufgaben erstellt.
            exec.execute(newTask(p, null, null));
            // blockiert bis Lösung gefunden wurde
            PuzzleNode<P,M> solnPuzzleNode = solution.getValue();
            return /* Lösung*/
        } finally {
            exec.shutdown();
        }
    }
}

```

```
@ThreadSafe
public class ValueLatch <T> {
    @GuardedBy("this") private T value = null;
    //nur eine Lösung wird akzeptiert
    private final CountDownLatch done = new CountDownLatch(1);

    public boolean isSet() {
        return (done.getCount() == 0);
    }

    public synchronized void setValue(T newValue) {
        if (!isSet()) {
            value = newValue;
            done.countDown();
        }
    }

    public T getValue() throws InterruptedException {
        done.await();
        synchronized (this) {
            return value;
        }
    }
}
```

ConcurrentPuzzleSolver (2):

- sequentieller Ansatz terminiert, wenn alle Positionen überprüft wurden.
- `ConcurrentPuzzleSolver` terminiert nicht, wenn es keine Lösung gibt.
- Lösung: Zahl der `SolverTasks` überwachen und die `solution` auf `null` setzen, wenn diese Zahl 0 erreicht hat.

```

public class PuzzleSolver <P,M> extends ConcurrentPuzzleSolver<P, M> {
    PuzzleSolver(Puzzle<P, M> puzzle) { super(puzzle); }

    private final AtomicInteger taskCount = new AtomicInteger(0);

    protected Runnable newTask(P p, M m, PuzzleNode<P, M> n) {
        return new CountingSolverTask(p, m, n);
    }

    class CountingSolverTask extends SolverTask {
        CountingSolverTask(P pos, M move, PuzzleNode<P, M> prev) {
            super(pos, move, prev);
            taskCount.incrementAndGet();
        }

        public void run() {
            try {
                super.run();
            } finally {
                if (taskCount.decrementAndGet() == 0)
                    solution.setValue(null);
            }
        }
    }
}

```

Zusammenfassung:

Das Executor Framework nimmt dem Programmierer viel Arbeit ab und ist zudem eine leistungsfähige und flexible Lösung um Aufgaben nebenläufig auszuführen.

Zu beachten ist:

- es gibt Kombinationen von Einstellungen die sich nicht vertragen
- manche Aufgabentypen erfordern bestimmte Ausführungsrichtlinien