

Concurrency in Java

Ein Blick in Vergangenheit, Gegenwart und Zukunft

Sebastian Henneberg

31. Januar 2011

Struktur

- Motivation
- Nebenläufigkeit bis Java 1.2
- Nebenläufigkeit in Java 1.4
- Nebenläufigkeit in Java 1.5
- externe Frameworks
- Nebenläufigkeit in Java 1.7
- Zusammenfassung

Motivation

Anwendungszwecke für Nebenläufigkeit

- ▶ GUI-Programmierung (Swing, Eclipse RCP)
- ▶ Web services
- ▶ Number crunching
- ▶ Datenbankprogrammierung
- ▶ intensive I/O-Operationen
- ▶ high-performance Computing
- ▶ Simulationen
- ▶ (eingebettete) Echtzeitsysteme

Motivation

Anwendungszwecke für Nebenläufigkeit

- ▶ GUI-Programmierung (Swing, Eclipse RCP)
- ▶ Web services
- ▶ Number crunching
- ▶ Datenbankprogrammierung
- ▶ intensive I/O-Operationen
- ▶ high-performance Computing
- ▶ Simulationen
- ▶ (eingebettete) Echtzeitsysteme

⇒ **Ausnutzung von Multicore Maschinen**

Motivation

Probleme bei nebenläufigen Anwendungen

▶ Liveness

- Deadlock
- Starvation / Fairness
- Livelock
- Race Condition

▶ Synchronisation

- Mutual exclusion
- inkonsistente Speicherzustände
- Sichtbarkeit von Modifikationen
- nicht-deterministisches Scheduling



Nebenläufigkeit bis Java 1.2

Klassen/Interfaces

- ▶ `java.lang.Object`
- ▶ `java.lang.Runnable`
- ▶ `java.lang.Thread`
- ▶ `java.lang.ThreadGroup`

Schlüsselwörter

- ▶ `synchronized`
- ▶ `volatile`
- ▶ `final`

Funktionalität eines Threads

- ▶ Java-Threads \Rightarrow native OS-Threads
- ▶ Daemon mode
- ▶ Priorisierung
- ▶ Gruppierung
- ▶ Steuerung
 - `stop()`
 - `suspend()`
 - `resume()`
 - `sleep()`
 - `join()`
 - `yield()`

Schlüsselwörter

- ▶ synchronized-Methoden

```
public synchronized void accessCriticalResource() {  
    // ...  
}
```

- ▶ synchronized-Statements

```
synchronized(someObject) {  
    someObject.doSomething();  
}
```

⇒ **Lock abgeben:** `Object.wait()` und `Object.notify[All]()`

- ▶ volatile

```
private volatile int counter;
```

- ▶ final

```
private final int someImmutable;
```

Nebenläufigkeit bis Java 1.2

Wars das schon? Résumé

- ▶ Threads stehen im Mittelpunkt
- ▶ keine zusätzlichen Werkzeuge
- ▶ Mutual exclusion
- ▶ immutable data
- ▶ blocking

“In those days, threads were used primarily for expressing *asynchrony*, not *concurrency*, and as a result, these mechanisms were generally adequate to the demands of the time.”

Brian Goetz (Senior Staff Engineer bei Sun)

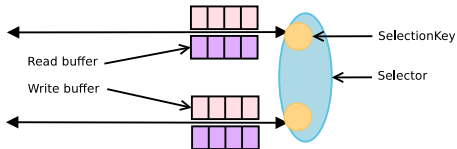
Nebenläufigkeit in Java 1.4

java.nio

- ▶ “New I/O” ermöglicht intensive I/O-Operationen
 - asynchron
 - multiplexed
 - non-blocking

⇒ beeinflusst Architektur von skalierbaren Servern

⇒ Thread-per-Socket → single-threaded



Nebenläufigkeit in Java 1.5

`java.util.concurrent`

- ▶ bisher umfangreichste API-Erweiterung für Nebenläufigkeit
- ▶ führt zahlreiche Neuerungen ein...
 - `Atomic`s
 - `Lock`s
 - `Synchronizers`
 - `Collections`
 - `Executors`

Nebenläufigkeit in Java 1.5

`java.util.concurrent`

- ▶ bisher umfangreichste API-Erweiterung für Nebenläufigkeit
- ▶ führt zahlreiche Neuerungen ein...
 - Atomics
 - Locks
 - Synchronizers
 - Collections
 - Executors

“In summary, using wait and notify directly is like programming in *concurrency assembly language*, as compared to the higher-level language provided by `java.util.concurrent`”

Joshua Bloch (Chief Java Architect bei Google)

Atomics (java.util.concurrent.atomic)

- ▶ lock-free
- ▶ thread-safe

Klassen

- ▶ Atomic(Boolean|Integer|Long|Reference)
- ▶ Atomic(Integer|Long|Reference)Array
- ▶ ...

```
1 class SerialNumber {
2     private AtomicLong serialNumber = new AtomicLong(0);
3     public long next() {
4         return serialNumber.getAndIncrement();
5     }
6 }
```

Locks (java.util.concurrent.locks)

Aufgabe

- ▶ locking und waiting
- ▶ unterscheidet sich von den Intrinsic Locks
- ▶ größere Flexibilität

Interfaces

- ▶ Condition
- ▶ Lock
- ▶ ReadWriteLock

```
1 Lock l = ...;
2 l.lock();
3 try {
4     // access locked resource
5 } finally {
6     l.unlock();
7 }
```

Synchronizers (`java.util.concurrent`)

Synchronisation von nebenläufigen Aufgaben

Klassen

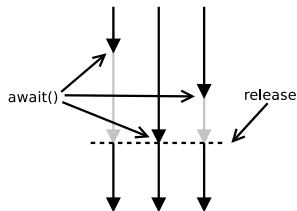
- ▶ Semaphore
- ▶ CountdownLatch
- ▶ CyclicBarrier
- ▶ Exchanger

Synchronizers (java.util.concurrent)

Synchronisation von nebenläufigen Aufgaben

Klassen

- ▶ Semaphore
- ▶ CountdownLatch
- ▶ CyclicBarrier
- ▶ Exchanger



```
1 CountdownLatch cdLatch = new CountdownLatch(3);
2 // ...
3 public void run() {
4     // ...
5     cdLatch.await();
6     // ...
7 }
```

Collections (`java.util.concurrent`)

Collections aus `java.util` für nebenläufige Anwendungen

Eigenschaften

- ▶ blockierende Operationen
- ▶ gleichzeitiges Schreiben/Ersetzen
- ▶ Konfiguration mittels “`concurrencyLevel`”
- ▶ keine `ConcurrentModificationException`
- ▶ Producer-Consumer-Prinzip

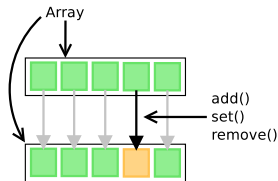
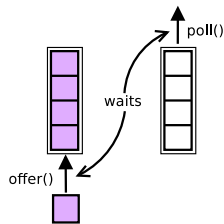
Collections (java.util.concurrent)

Interfaces

- ▶ `BlockingQueue<E>`
- ▶ `ConcurrentMap<K, V>`

Klassen

- ▶ `(Array|Linked|Priority)-BlockingQueue<E>`
- ▶ `ConcurrentHashMap<K, V>`
- ▶ `ConcurrentLinkedQueue<E>`
- ▶ `CopyOnWriteArray(List|Set)<E>`
- ▶ `SynchronousQueue<E>`



Executor (`java.util.concurrent`)

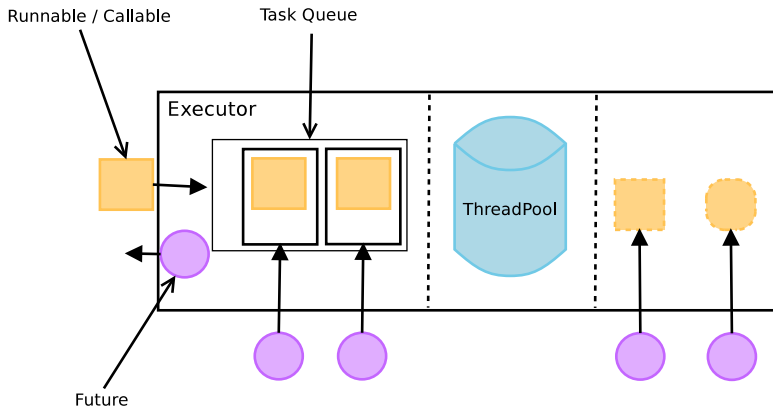
Executor

- ▶ asynchrone und nebenläufige Ausführung vieler Aufgaben
- ▶ Verwaltung der Ressourcen zur Laufzeit
- ▶ erhöhte Performanz
- ▶ Separation of concerns

wichtige Interfaces

- ▶ `Callable<V>`
- ▶ `Executor`
- ▶ `ExecutorService`
- ▶ `Future<V>`
- ▶ `ScheduledExecutorService`

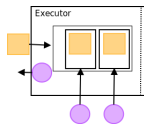
Executor, Runnable, Callable, Future, ...



Variationen der Executor

▶ Queuing

- Ausführung: direkt / verzögert
- beschränkte Queue
- priorisiert Queue
- abbrechbare Aufgaben



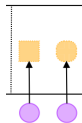
▶ Pool

- fixe / variable Größe
- ThreadFactory
- Wiederverwendung von Threads



▶ Results

- Status abfragbar
- Ergebnis extrahierbar
- Aufgaben zurückweisbar



externe Frameworks

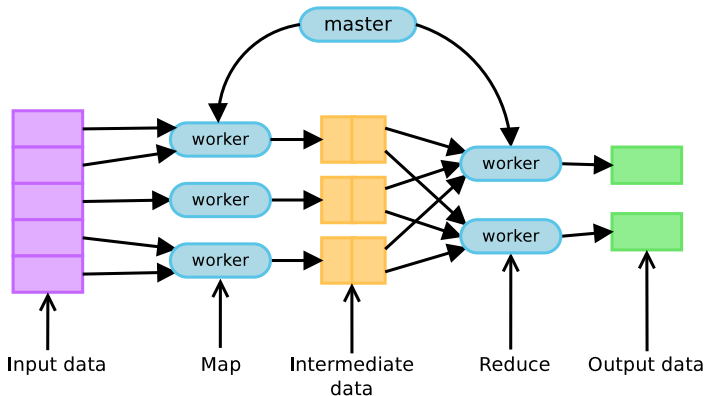
MapReduce

- ▶ Nebenläufigkeit auf ein Cluster ausweiten
- ▶ Verarbeitung riesiger Datenmengen
- ▶ z.B. für Data Mining
- ▶ wird bei Google genutzt

Actors

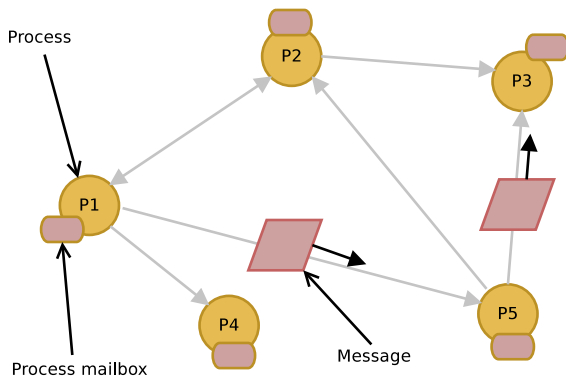
- ▶ leichgewichtige Prozesse
- ▶ Kommunikation über message passing
- ▶ kein gemeinsamer Speicher

MapReduce



Implementierung:
Apache Hadoop

Actor model



zahlreiche Implementierungen:

Akka, Ateji PX, Korus, Kilim, ActorFoundry, Jetlang

Nebenläufigkeit in Java 1.7

Fork/Join

- ▶ inspiriert durch divide & conquer
- ▶ teile (rekursiv) großes Problem in mehrere kleinere Probleme
⇒ Partitionierung muss möglich sein
- ▶ wenn Aufgabe klein genug, verarbeite sequentiell

ParallelArray

- ▶ Verarbeitung großer homogener Datenmengen
- ▶ mit DB-ähnlichen Operationen wie
 - Filter
 - Mapping
 - Aggregation

Fork/Join

RecursiveAction

```
1 Result compute(Problem problem) {
2     if (problem.size < THRESHOLD)
3         return solveSequentially(problem);
4     else {
5         Result left, right;
6         invokeAll {
7             left = compute(extractLeftHalf(problem));
8             right = compute(extractRightHalf(problem));
9         }
10        return combine(left, right);
11    }
12 }
```

Fork/Join

RecursiveAction

```
1 Result compute(Problem problem) {
2     if (problem.size < THRESHOLD)
3         return solveSequentially(problem);
4     else {
5         Result left, right;
6         invokeAll {
7             left = compute(extractLeftHalf(problem));
8             right = compute(extractRightHalf(problem));
9         }
10        return combine(left, right);
11    }
12 }
```

Aufruf

```
1 ForkJoinPool pool = new ForkJoinPool();
2 pool.invoke(new Problem(...));
```

ParallelArray

```
1 ParallelArray<Student> students =
2     new ParallelArray<Student>(fjPool, data);
3
4 int youngster = students.withMapping(selectAge)
5     .min();
6
7 public class Student {
8     String name;
9     int age;
10 }
11
12 static final Ops.ObjectToInt<Student> selectAge =
13     new Ops.ObjectToInt<Student>() {
14     public int op(Student student) {
15         return student.age;
16     }
17 };
```

ParallelArray

```
1 ParallelArray<Student> students =
2     new ParallelArray<Student>(fjPool, data);
3
4 int youngster = students.withMapping(selectAge)
5                     .min();
6
7 public class Student {
8     String name;
9     int age;
10 }
11
12 static final Ops.ObjectToInt<Student> selectAge =
13     new Ops.ObjectToInt<Student>() {
14     public int op(Student student) {
15         return student.age;
16     }
17 };

addAll(), allUniqueElements(), sort(), with(Filter|Mapping)(), ...
```

Zusammenfassung

- ▶ Entwicklung der Hardware beeinflusst Java API

Zusammenfassung

- ▶ Entwicklung der Hardware beeinflusst Java API
- ▶ Thread verschwindet aus dem Zentrum ...
- ▶ ... wird ersetzt durch Runnable und Callable
- ▶ ... sowie durch umfangreiche API

Zusammenfassung

- ▶ Entwicklung der Hardware beeinflusst Java API
- ▶ Thread verschwindet aus dem Zentrum ...
- ▶ ... wird ersetzt durch `Runnable` und `Callable`
- ▶ ... sowie durch umfangreiche API
- ▶ Sprachfeatures werden durch Frameworks ersetzt
- ▶ ... diese unterscheiden sich stark
- ▶ ... Wahl hängt sehr von Problem/Domäne ab

Anhang

Thread stoppen

```
1 public class StopThread {
2
3     private static boolean stopRequested;
4
5     public static void main(String[] args) throws InterruptedException
6
7         Thread backgroundThread = new Thread(new Runnable() {
8             public void run() {
9                 int i = 0;
10                while (!stopRequested)
11                    i++;
12            }
13        });
14
15        backgroundThread.start();
16        Thread.sleep(1000);
17        stopRequested = true;
18    }
19 }
```

Thread stoppen

```
10         while (!stopRequested)
11             i++;
12     }
```

Thread stoppen

```
10         while (!stopRequested)
11             i++;
12     }
```

⇓ “hoisting”

```
10         if (!stopRequested)
11             while (true)
12                 i++;
```

Thread stoppen

```
10         while (!stopRequested)
11             i++;
12     }
```

⇓ “hoisting”

```
10         if (!stopRequested)
11             while (true)
12                 i++;
```

- ▶ Optimierung des HotSpot compiler (C2)
- ▶ “Loop Invariant Code Motion”
- ▶ Lösung?

synchronized vs. volatile

```
1 public class StopThread {
2     // ...
3     private static synchronized void requestStop() {
4         stopRequested = true;
5     }
6     private static synchronized boolean stopRequested() {
7         return stopRequested;
8     }
9
10    // ...
11    while (!stopRequested())
12        i++;
13    }
14    // ...
15    requestStop();
16 }
17 }

3 private static volatile boolean stopRequested;
```

wait(), notify() und synchronized

```
1  public class ConnectionPool {
2
3      private List<Connection> connections = createConnections();
4
5      public Connection getConnection() throws InterruptedException
6          synchronized (connections) {
7          while (connections.isEmpty()) {
8              connections.wait();
9          }
10         return connections.remove(0);
11     }
12 }
13
14 public void returnConnection(Connection conn) {
15     synchronized (connections) {
16         connections.add(conn);
17         connections.notify();
18     }
19 }
20 }
```

praktischeres Beispiel für Fork/Join

```
1  public class MaxWithFJ extends RecursiveAction {
2      private final int threshold;
3      private final SelectMaxProblem problem;
4      public int result;
5
6      public MaxWithFJ(SelectMaxProblem problem, int threshold) {
7          this.problem = problem;
8          this.threshold = threshold;
9      }
10
11     protected void compute() {
12         if (problem.size < threshold)
13             result = problem.solveSequentially();
14         else {
15             int midpoint = problem.size / 2;
16             MaxWithFJ left = new MaxWithFJ(problem.subproblem(
17                 0, midpoint), threshold);
18             MaxWithFJ right = new MaxWithFJ(problem.subproblem(
19                 midpoint + 1, problem.size), threshold);
20             coInvoke(left, right);
21             result = Math.max(left.result, right.result);
22         }
```

praktischeres Beispiel für Fork/Join

```
24
25     public static void main(String[] args) {
26         SelectMaxProblem problem = ...
27         int threshold = ...
28         int nThreads = ...
29         MaxWithFJ mfj = new MaxWithFJ(problem, threshold);
30         ForkJoinExecutor fjPool = new ForkJoinPool(nThreads);
31
32         fjPool.invoke(mfj);
33         int result = mfj.result;
34     }
35 }
```

Performanz von Fork/Join

Table 1. Results of Running select-max on 500k-element Arrays on various systems

	Threshold=500k	Threshold=50k	Threshold=5k	Threshold=500	Threshold=-50
Pentium-4 HT (2 threads)	1.0	1.07	1.02	.82	.2
Dual-Xeon HT (4 threads)	.88	3.02	3.2	2.22	.43
8-way Opteron (8 threads)	1.0	5.29	5.73	4.53	2.03
8-core Niagara (32 threads)	.98	10.46	17.21	15.34	6.49

Performanz von ParallelArray

Table 1. Performance measurement for the max-GPA query

		Threads			
		1	2	4	8
Students	1000	1.00	0.30	0.35	1.20
	10000	2.11	2.31	1.02	1.62
	100000	9.99	5.28	3.63	5.53
	1000000	39.34	24.67	20.94	35.11
	10000000	340.25	180.28	160.21	190.41

“Core 2 Quad system running Windows”