Lernzettel

Nebenläufigkeit und Parallelität in Scala:
Futures, ExecutionContext, Parallel Collections.

Universität:	Technische Universität Berlin
Kurs/Modul:	Programmieren II für Wirtschaftsinformatiker
Erstellungsdatum:	September 6, 2025

Lernzettel: Nebenläufigkeit und Parallelität in Scala: Futures, ExecutionContext, Parallel Collections.

(1) Grundlagen. Nebenläufigkeit bedeutet, dass mehrere Aufgaben gleichzeitig laufen können. Parallelität bedeutet, dass mehrere Aufgaben wirklich gleichzeitig ablaufen können, je nach verfügbarer Hardware. In Scala wird dies häufig durch Futures und Parallel Collections realisiert. Der Kurs behandelt diese Konzepte im Rahmen von funktionalem Stil und der Interoperabilität von Java und Scala.

(2) Futures in Scala. Ein Future[T] repräsentiert eine Berechnung, deren Ergebnis in Zukunft vorliegt. Sie starten üblicherweise sofort mit einem ExecutionContext. Beispiele:

```
import scala.concurrent.{Future, ExecutionContext}
import scala.concurrent.ExecutionContext.Implicits.global

val f: Future[Int] = Future { 1 + 2 }
f.map(_ * 2) // Future[Int]
```

(3) ExecutionContext. Der ExecutionContext steuert, auf welchen Threads Futures laufen. Häufig verwendest du den Global-Context:

```
import scala.concurrent.ExecutionContext.Implicits.global
```

Für erweiterte Anforderungen kannst du eigene Threadpools verwenden:

```
import java.util.concurrent.Executors
import scala.concurrent.ExecutionContext

val es = Executors.newFixedThreadPool(4)
implicit val ec: ExecutionContext = ExecutionContext.fromExecutor(es)
```

Beachte: Blockierende Aufrufe sollten innerhalb eines Blocks mit dem Konstrukt blocking markiert werden, damit der Pool entsprechend skalieren kann.

(4) Fortgeschrittene Muster mit Futures. Futures lassen sich mit map, flatMap und for-Komprehensionen zusammenführen, um asynchrone Workflows auszudrücken:

```
val a: Future[Int] = Future { computeA() }
val b: Future[Int] = Future { computeB() }

val sum: Future[Int] = for {
  va <- a
  vb <- b
} yield va + vb
```

Hinweis: Sei vorsichtig bei Nebenwirkungen und gemeinsamem Zustand.

(5) Parallel Collections. Parallele Sammlungen ermöglichen, Elemente unabhängig zu verarbeiten. Verwendung:

```
val data = (1 to 10000).par
val squares = data.map(x => x*x).toList
val total = squares.sum
```

Wichtige Hinweise: - Seit Scala 2.13 wurden parallele Sammlungen teilweise ausgelagert; nutze ggf. das Paket scala.collection.parallel.ParIterable bzw. die Bibliothek scala-parallel-collections. - Sei vorsichtig mit Nebenwirkungen; Operationen sollten rein funktional sein, um Race Conditions zu vermeiden.

(6) Best Practices und Hinweise. - Vermeide geteilten Mutable State; verwende immutable Strukturen. - Nutze Futures statt Threads direkt, um Ressourcen zu schonen. - Nutze for-Komprehension oder Sequenzen von Futures, um Fehler zu propagieren und zu handhaben. - Wenn du Akka oder Message-Passing (Actor-Modell) einsetzt, halte das Muster der isolationkonformen Kommunikation im Blick.

(7) Anwendungsbezug im Wirtschaftsinformatik-Kontext. In BI-Szenarien erleichtert asynchrones Laden und parallele Verarbeitung von Datenquellen das Reagieren auf Anfragen, z.B. beim gleichzeitigen Abrufen von Datenströmen oder beim parallelen Durchführen von ETL-Schritten. Akka kann als zusätzliche Lösung für verteilte Message-Passing-Concurrency dienen, wiederum unterstützt durch Futures und ExecutionContext.

(8) Hinweise zur Umsetzung und Performance. - Wähle sinnvolle Granularität der Aufgaben; zu feine Aufgabenteilung kann Overhead erzeugen. - Miss die Latenz vs. Throughput; manchmal ist eine einfache sequentielle Verarbeitung schneller, wenn der Overhead dominiert. - Berücksichtige Ressourcenlimits (Thread-Pool-Größe, CPU-Kerne) bei der Wahl des ExecutionContext.

(9) Beispiel-Architektur im BI-Umfeld (Kurzüberblick). - Datenquellen asynchron anfragen (Future + ExecutionContext). - Ergebnisse parallel aggregieren (Parallel Collections oder Futures mit for-Komprehension). - Ergebnisse sicher zusammenführen (immutable Datenstrukturen, Fehlerbehandlung).

(10) Weiterführende Ressourcen. - Scala-Dokumentation: Future, ExecutionContext. - Scala Parallel Collections (Versionabhängig).