

Asynchrone Anwendungen mit CompletableFuture

Vorstellung

- Software-Architekt, Entwickler und Trainer
- Fachliche Schwerpunkte
 - Enterprise Java
 - APIs und Services
 - Kommunikation und Systemintegration
- Kundenspezifische Inhouse-Schulungen
- (Co-)Autor mehrerer Bücher & zahlreicher Artikel
- Sprecher auf internationalen Konferenzen

Feedback



Gründe für den Einsatz von Asynchronität

- Synchrone Abläufe skalieren schlechter
- Verbindungen zu vielen unterschiedlichen APIs
- Technische Anforderungen
 - Microservice-Architekturen
 - Anbindung anderer Systeme
- Fachliche Anforderungen
 - Einholen von Preisen oder Verfügbarkeiten bei mehreren Anbietern
 - Aggregation von Daten
- Unwägbarkeiten verteilter Systeme

Seit Java 5: Future<T>

```
ExecutorService executor = ...;  
QuoteService quoteService = ...;  
  
Future<Integer> futureQuoteFromA =  
    executor.submit(quoteService::getQuoteFromSupplierA);  
  
Future<Integer> futureQuoteFromB =  
    executor.submit(quoteService::getQuoteFromSupplierB);  
  
// do other tasks for some time...
```

Blockierende API für Abruf des Ergebnisses (Pull)

```
// do even more tasks for some time...  
  
Integer quoteFromA = futureQuoteFromA.get();  
Integer quoteFromB = futureQuoteFromB.get();  
  
quoteService.processQuotes(quoteFromA, quoteFromB);
```

Blockierende API für Abruf des Ergebnisses (Poll)

```
Map<String, Integer> allQuotes = new HashMap<>();
while(quotes.size() < 2) {
    try {
        allQuotes.put("A", futureQuoteFromA.get(500, MILLISECONDS));
    } catch (TimeoutException e) {
        log.info("Quote not yet available from supplier A.");
    }

    try {
        allQuotes.put("B", futureQuoteFromB.get(500, MILLISECONDS));
    } catch (TimeoutException e) {
        log.info("Quote not yet available from supplier B.");
    }

    // do other tasks for some time...
}
quoteService.processQuotes(allQuotes.get("A"), allQuotes.get("B"));
```

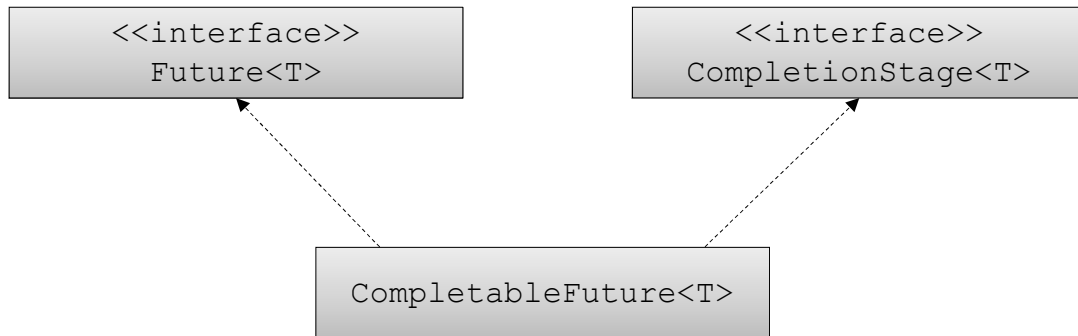
Java 8: CompletableFuture<T>

- *Fluent Programming* für das Behandeln asynchroner Ergebnisse
- Mehrstufige Abläufe („Stages“) deutlich eleganter ausdrückbar
- Push-Ansatz: Bindet weniger Ressourcen als Pull und Poll
- Ein erstes Beispiel:

```
CompletableFuture<Integer> futureQuoteFromA =
    CompletableFuture.supplyAsync(quoteService::getQuoteFromSupplierA);

futureQuoteFromA.thenAcceptAsync(quoteService::displayQuote);
```

CompletionStage und CompletableFuture



CompletableFuture – Factory-Methoden

- Nebenläufige Ausführung einer Task

- mit Ergebnis

```
<U> CompletableFuture<U> supplyAsync (Supplier<U> task)
<U> CompletableFuture<U> supplyAsync (Supplier<U> task, Executor e)
```

- ohne Ergebnis

```
CompletableFuture<Void> runAsync (Runnable task)
CompletableFuture<Void> runAsync (Runnable task, Executor executor)
```

Consumer, Function, Runnable

- Consumer<T>

```
void accept(T t);
```

- Function<T, U>

```
R apply(T t);
```

- Runnable

```
void run();
```

- BiConsumer<T, U>

```
void accept(T t, U u);
```

- BiFunction<T, U, R>

```
R apply(T t, U u);
```

Basisfunktionalität von CompletableFuture

- Consumer<T>

```
CompletableFuture<Void> thenAccept(Consumer<T> task)  
CompletableFuture<Void> thenAcceptAsync(Consumer<T> task)  
CompletableFuture<Void> thenAcceptAsync(Consumer<T> task, Executor e)
```

- Function<T, U>

```
<U> CompletableFuture<U> thenApply(Function<T,U> task)  
<U> CompletableFuture<U> thenApplyAsync(Function<T,U> task)  
<U> CompletableFuture<U> thenApplyAsync(Function<T,U> task, Executor e)
```

- Runnable

```
CompletableFuture<Void> thenRun(Runnable task)  
CompletableFuture<Void> thenRunAsync(Runnable task)  
CompletableFuture<Void> thenRunAsync(Runnable task, Executor e)
```

Beispiel: Verknüpfung mehrerer Stages

```
CompletableFuture<Void> futureAllStagesDone =  
    CompletableFuture.supplyAsync(quoteService::getQuoteFromSupplierA)  
        .thenApply(quoteService::computeSalePrice)  
        .thenAccept(quoteService::displaySalePrice)  
        .thenRun(() -> sendEmailNotification());  
  
// do other tasks while the above chain of actions is executed  
...
```

Fehlerbehandlung

- Unchecked Exception in einer Stage führt zu Abbruch
 - folgende Stages werden nicht mehr ausgeführt
 - Abbruch wird nicht nach außen sichtbar
 - aufgetretene Exception geht verloren
- Spezielle Methoden für Fehlerbehandlung und Fortsetzung

```
CompletableFuture<T> exceptionally (Function<Throwable, T> task)  
<U> CompletableFuture<U> handle (BiFunction<T, Throwable, U> task)  
CompletableFuture<T> whenComplete (BiConsumer<T, Throwable> task)
```

Fehlerbehandlung

```
CompletableFuture.supplyAsync (quoteService::getQuoteFromSupplierA)  
    .exceptionally (this::handleQuoteRetrievalIssue)  
    .thenAccept (quoteService::displaySalePrice);
```

```
protected Integer handleQuoteRetrievalIssue(Throwable t) {  
    // handle error and return default quote  
    return -1;  
}
```

- Behandlung der Exception in nächster Stage mit handle

```
CompletableFuture.supplyAsync (quoteService::getQuoteFromSupplierA)  
    .handle ((quote, t) -> {  
        quoteService.displaySalePriceOrError (quote, t);  
        return null;  
    });
```

Kombination mehrerer Ergebnisse

- Neue Stage auf Basis der Ergebnisse mehrerer vorheriger Stages

```
acceptEither (CompletionStage<T> other, Consumer<T> task)  
applyToEither (CompletionStage<T> other, Function<T, U> task)  
runAfterEither (CompletionStage<T> other, Runnable task)
```

```
thenAcceptBoth (CompletionStage <U> other, BiConsumer<T, U> task)  
thenCombine (CompletionStage <U> other, BiFunction<T, U, V> task)  
runAfterBoth (CompletionStage <T> other, Runnable task)
```


Kombination mehrerer Ergebnisse

```
CompletableFuture<Integer> futureQuoteFromA =  
    CompletableFuture.supplyAsync(quoteService::getQuoteFromSupplierA)  
        .exceptionally(this::handleQuoteRetrievalIssue);  
  
CompletableFuture<Integer> futureQuoteFromB =  
    CompletableFuture.supplyAsync(quoteService::getQuoteFromSupplierB)  
        .exceptionally(this::handleQuoteRetrievalIssue);  
  
CompletableFuture<Void> bothProcessed =  
    futureQuoteFromA.thenCombine(futureQuoteFromB,  
        quoteService::calculateAveragePrice)  
        .thenApply(quoteService::computeSalePrice)  
        .thenAccept(quoteService::displaySalePrice)  
        .thenRun(() -> sendEmailNotification());  
  
// do some other tasks meanwhile...
```

Kleinere API-Erweiterungen in Java 9

- Delays

```
Executor exe =  
    CompletableFuture  
        .delayedExecutor(50L, TimeUnit.SECONDS);
```

- Timeouts

```
new CompletableFuture()  
    .orTimeout(1, TimeUnit.SECONDS)  
  
new CompletableFuture()  
    .completeOnTimeout(value, 1, TimeUnit.SECONDS)
```

Zusammenfassung

- `CompletableFuture<T>` ermöglicht elegante Implementierung mehrstufiger asynchrone Abläufe
- Weniger Code notwendig als mit `Future<T>`
- Algorithmus besser lesbar und leichter verständlich
- Gute Unterstützung für Fehlerbehandlung
- Abhängigkeiten von mehreren Ergebnissen darstellbar

- Let's start coding! 😊



Thilo Frotscher

Entwicklung, Beratung und kundenspezifisches Training

Enterprise Java,
Services & Systemintegration

thilo@frotscher.com

 [@thfro](https://twitter.com/thfro)