GitHub:

Bei eugenp für den Quellcode

1. Einführung

Java 8 hat eine neue Abstraktion eingeführt, die auf basiert Zukunft **asynchrone Aufgaben** – CompletableFuture class. Es kam im Grunde, um die Probleme des alten Future-API.

In diesem Tutorial werden wir untersuchen, wie mit Exceptions gearbeitet werden kann, wenn wir sie verwenden **CompletableFuture**.

2. CompletableFuture Recap

Zuerst müssen wir vielleicht ein wenig zusammenfassen, was die **CompletableFuture** ist. **CompletableFuture** ist ein **Future implementation**, mit der wir asynchrone Vorgänge ausführen und vor allem verketten können. Im Allgemeinen gibt es drei mögliche Ergebnisse für die **async operation to complete**

- normally (normaler weise)
- exceptionally (ausnahmweise)
- canceled from outside (von außerhalb abzubrechen)

CompletableFuture hat verschiedene API-Methoden, um all diese möglichen Ergebnisse zu erzielen.

Wie bei vielen anderen Methoden in **CompletableFuture**, gibt es spezifische **specific Executor variations**

- Non-Asynch Method Link
- Async Methods Link
- Async Methods With Custom Executor Link
- Extending CompletableFuture Link

Wir lernen Schritt für Schritt welche Variationen machbar sind.

3. handle()

Zuerst haben wir die Methode **handle()**. Mit dieser Methode, können wir auf das gesamte Ergebnis zugreifen unabhängig vom Ergebnis. Das heißt, die **handle()** Methode akzeptiert Zwei Eingänge und ist ein funktionales Interface. In der **handle()** Methode, Parameter sind das Ergebnis des vorherigen Fertigstellungsphase und die Ausnahme das ist passiert.

Das Wichtigste ist das Beide Parameter sind optional, was bedeutet, dass sie es sein können null. Dies ist in gewissem Sinne seit dem vorherigen offensichtlich Fertigstellungsphase wurde normal abgeschlossen. Dann das Ausnahme sollte sein null da gab es keine, ähnlich mit Fertigstellungsphase Nichtigkeit des Ergebniswerts.

Schauen wir uns nun ein Beispiel an Griff ( ) Methodengebrauch:

```
@ParameterizedTest
@MethodSource("parametersSource_handle")
void whenCompletableFutureIsScheduled_thenHandleStageIsAlwaysInvoked(int radius,
long expected)
  throws ExecutionException, InterruptedException {
    long actual = CompletableFuture
      .supplyAsync(() -> {
          if (radius <= 0) {
              throw new IllegalArgumentException("Supplied with non-positive
radius '%d'");
          }
          return Math.round(Math.pow(radius, 2) * Math.PI);
      })
      .handle((result, ex) -> {
          if (ex == null) {
              return result;
          } else {
              return -1L;
          }
      })
      .get();

    Assertions.assertThat(actual).isEqualTo(expected);
}

static Stream<Arguments> parameterSource_**handle()** {
    return Stream.of(Arguments.of(1, 3), Arguments.of(1, -1));
}
```

Hier ist zu beachten, dass das **handle()** Methode gibt eine neue zurück Fertigstellungsphase das wird immer ausgeführt, unabhängig vom vorherigen Fertigstellungsphase Ergebnis. Damit, **handle()** transformiert den Quellwert aus der vorherigen Stufe in einen Ausgabewert. Daher der Wert, den wir über die erhalten werden erhalten() Methode ist die von zurückgegebene der Griff() Methode.

4. ausnahmsweise()

Der Griff() Methode ist nicht immer praktisch, insbesondere wenn wir Exceptions nur verarbeiten möchten, wenn es eine gibt. Zum Glück haben wir eine Alternative – ausnahmsweise().

Mit dieser Methode können wir einen Rückruf bereitstellen, der ausgeführt werden soll nur wenn der vorherige Fertigstellungsphase endete mit einem Ausnahme. Wenn keine Exceptions gemacht wurden, wird der Rückruf weggelassen und die Ausführungskette wird mit dem Wert des vorherigen Rückrufs (falls vorhanden) zum nächsten Rückruf fortgesetzt.

Schauen wir uns ein konkretes Beispiel an:

```
@ParameterizedTest
@MethodSource("parametersSource_exceptionally")
```

```
void whenCompletableFutureIsScheduled_thenExceptionallyExecutedOnlyOnFailure(int
a, int b, int c, long expected)
  throws ExecutionException, InterruptedException {
    long actual = CompletableFuture
      .supplyAsync(() -> {
          if (a <= 0 || b <= 0 || c <= 0) {
              throw new IllegalArgumentException(String.format("Supplied with
incorrect edge length [%s]", List.of(a, b, c)));
          }
          return a * b * c;
      })
      .exceptionally((ex) -> -1)
      .get();

    Assertions.assertThat(actual).isEqualTo(expected);
}

static Stream<Arguments> parametersSource_exceptionally() {
    return Stream.of(
      Arguments.of(1, 5, 5, 25),
      Arguments.of(-1, 10, 15, -1)
    );
}
```

Hier funktioniert es also genauso wie **handle()**, aber wir haben eine Ausnahme Instanz als Parameter für unseren Rückruf. Dieser Parameter wird niemals sein null, Unser Code ist jetzt etwas einfacher.

Das Wichtigste, was Sie hier bemerken sollten, ist das ausnahmsweise() Der Rückruf der Methode wird nur ausgeführt, wenn die vorherige Stufe mit einer abgeschlossen wurde Ausnahme. Es bedeutet im Grunde, dass wenn die Ausnahme ereignete sich irgendwo in der Hinrichtungskette, und es gab bereits eine **handle()** Methode, die es gefangen hat – die **expetional ()** Der Rückruf wird danach nicht mehr ausgeführt:

```
@ParameterizedTest
@MethodSource("parametersSource_exceptionally")
void
givenCompletableFutureIsScheduled_whenHandleIsAlreadyPresent_thenExceptionallyIsNo
tExecuted(int a, int b, int c, long expected)
  throws ExecutionException, InterruptedException {
    long actual = CompletableFuture
      .supplyAsync(() -> {
          if (a <= 0 || b <= 0 || c <= 0) {
              throw new IllegalArgumentException(String.format("Supplied with
incorrect edge length [%s]", List.of(a, b, c)));
          }
          return a * b * c;
      })
      .handle((result, throwable) -> {
          if (throwable != null) {
              return -1;
          }
```

```
            return result;
        })
        .exceptionally((ex) -> {
            System.exit(1);
            return 0;
        })
        .get();

    Assertions.assertThat(actual).isEqualTo(expected);
}
```

Hier, ausnahmsweise() wird nicht angerufen, da die **handle()** Methode fängt schon die Ausnahme, falls vorhanden. Daher, es sei denn, die Ausnahme tritt innerhalb der **handle()** Methode, die ausnahmsweise() Methode hier wird nie ausgeführt.

5. wenn**CompletableFuture** ()

Wir haben auch eine wenn**CompletableFuture** () Methode in der API. Es akzeptiert die BiConsumer mit zwei Parametern: das Ergebnis und gegebenenfalls die Ausnahme von der vorherigen Stufe. Diese Methode unterscheidet sich jedoch erheblich von der oben genannten.

Der Unterschied ist das wenn**CompletableFuture** () wird keine außergewöhnlichen Ergebnisse aus den vorherigen Phasen übersetzen. Also, auch wenn man das bedenkt wenn**CompletableFuture** ()Der Rückruf von ‘ wird immer ausgeführt. Die Ausnahme von der vorherigen Phase, falls vorhanden, breitet sich weiter aus:

```
@ParameterizedTest
@MethodSource("parametersSource_whenComplete")
void whenCompletableFutureIsScheduled_thenWhenCompletedExecutedAlways(Double a,
long expected) {
    try {
        CountDownLatch countDownLatch = new CountDownLatch(1);
        long actual = CompletableFuture
          .supplyAsync(() -> {
              if (a.isNaN()) {
                  throw new IllegalArgumentException("Supplied value is NaN");
              }
              return Math.round(Math.pow(a, 2));
          })
          .whenComplete((result, exception) -> countDownLatch.countDown())
          .get();
        Assertions.assertThat(countDownLatch.await(20L,
java.util.concurrent.TimeUnit.SECONDS));
        Assertions.assertThat(actual).isEqualTo(expected);
    } catch (Exception e) {
        Assertions.assertThat(e.getClass()).isSameAs(ExecutionException.class);

Assertions.assertThat(e.getCause().getClass()).isSameAs(IllegalArgumentException.c
lass);
```

```
    }
}

static Stream<Arguments> parametersSource_whenComplete() {
    return Stream.of(
      Arguments.of(2d, 4),
      Arguments.of(Double.NaN, 1)
    );
}
```

Wie wir hier sehen können, Rückruf drinnen wenn abgeschlossen () läuft in beiden Testaufrufen. Im zweiten Aufruf haben wir jedoch mit dem abgeschlossen Ausführungsausnahme, die die Ursache unserer hat IllegalArgumentException. Wie wir sehen können, breitet sich die Ausnahme vom Rückruf auf die Freude aus. Wir werden die Gründe dafür im nächsten Abschnitt behandeln.

6. Unbehandelte Exceptions

Schließlich müssen wir uns ein wenig mit unbehandelten Exceptions befassen. Im Allgemeinen, wenn eine Ausnahme nicht erfasst wird, dann die **CompletableFuture** ver**CompletableFuture**t mit einem Ausnahme das breitet sich nicht zur Freude aus. In unserem obigen Fall bekommen wir die Ausführungsausnahme von der erhalten() Methodenaufruf. Das liegt also daran, dass wir versucht haben, auf das Ergebnis zuzugreifen, als **CompletableFuture** endete mit einem Ausnahme.

Daher müssen wir das Ergebnis des überprüfen **CompletableFuture** vor dem erhalten() Anrufung. Es gibt verschiedene Möglichkeiten, dies zu tun. Der erste und wahrscheinlich bekannteste Ansatz ist via **isCompletedExceptionally ()** / **isCancelled ()** / **isDone ()** Methoden. Diese Methoden geben a zurück boolescher in dem Fall CompletableFutre abgeschlossen mit der Ausnahme, wird von außen storniert oder erfolgreich abgeschlossen.

Es ist jedoch erwähnenswert, dass es auch eine gibt Zustand() Methode, die eine zurückgibt Staat enum Instanz. Diese Instanz repräsentiert den Status der **CompletableFuture**, wie LAUFEN, ERFOLG, usw. Dies ist also eine andere Möglichkeit, auf das Ergebnis des **CompletableFuture**.