6 Threadpools

Zahlreiche Aufgaben, die nebenläufig ausgeführt werden sollen, sind oft nur von kurzer Dauer und treten nicht unbedingt regelmäßig auf. Würde man also für jede neue Aufgabe einen Thread erzeugen und starten, würde das Betriebssystem unnötig belastet werden. Es ist sinnvoller, Threads wiederzuverwenden.

Ein weiterer Punkt ist, dass sich eine große Anzahl von Threads negativ auf die Systemleistung auswirkt. Die maximale Anzahl von nebenläufigen Aktivitäten, die ein Prozess verwalten kann, ist nicht festgelegt und hängt von der Implementierung der JVM und dem zugrunde liegenden Betriebssystem ab. Es ist daher wichtig, die Menge der erzeugten Threads zu beschränken.

In der Praxis wird man deshalb weniger mit rudimentären Thread-Objekten arbeiten, sondern mit sogenannten *Threadpools*. Java stellt verschiedene Realisierungen in dem Paket `java.util.concurrency` zur Verfügung, die wir im Folgenden besprechen.

6.1 Das Poolkonzept und die Klasse Executors

Ein Threadpool verwaltet eine gewisse Anzahl von Threads. Soll eine Aufgabe nebenläufig durchgeführt werden, so übergibt man dem Pool ein entsprechendes `Runnable`-Objekt. Je nach Art des Pools wird es sofort einem Thread zugeteilt oder erst in eine Queue gestellt und später bearbeitet. Wenn der Thread das `Runnable` ausgeführt hat, wird er ohne zu terminieren zurück in den Pool gestellt und kann weitere noch wartende Aufgaben übernehmen. Es werden somit nicht dauernd neue Threads vom Betriebssystem angefordert und beendet.

Im Prinzip kann man mit den bisher vorgestellten Konzepten eigene Threadpools implementieren (siehe zum Beispiel [41]). In der Praxis sollten aber die von Java angebotenen Möglichkeiten verwendet werden.

Schauen wir uns zuerst die `Executors`-Klasse an. Sie enthält Fabrikmethoden zur Erzeugung von Threadpools folgender Typen:

1. `ThreadPoolExecutor`
2. `ScheduledThreadPoolExecutor`
3. `ForkJoinPool`

Die gebräuchlichsten Fabrikmethoden sind:

- `newCachedThreadPool()`
- `newFixedThreadPool(int nThreads)`
- `newScheduledThreadPool(int coreSize)`
- `newWorkStealingPool()`
- `newWorkStealingPool(int parallelism)`

Sie liefern alle eine `ExecutorService`- bzw. `ScheduledExecutorService`-Implementierung zurück.

Mit `newCachedThreadPool` erhalten wir einen Pool, der bei Bedarf neue Threads erzeugt. Unbenutzte Threads bleiben für 60 Sekunden erhalten und werden danach, falls sie zwischenzeitlich nicht benötigt wurden, terminiert. Pools dieser Art werden typischerweise zur Performance-Verbesserung von Programmen eingesetzt, die viele kurzlebige, asynchrone Tasks benötigen.

Mit `newFixedThreadPool(int nThreads)` wird ein Pool mit einer festen Anzahl von Threads erzeugt. Pools dieser Art haben eine unbeschränkte Warteschlange für übergebene Aufgaben. Zu jedem Zeitpunkt sind höchstens `nThreads` tätig. Ein eingestellter Task muss unter Umständen auf einen freien Thread warten. Stirbt ein Thread aufgrund eines Fehlers, wird er durch einen neuen ersetzt.

Die Fabrikmethode `newScheduledThreadPool(int coreSize)` liefert einen `ScheduledExecutorService`, mit dessen Hilfe Aufgaben nach einer gegebenen Verzögerung bzw. periodisch ausgeführt werden können. Mit Java 8 wurden noch zwei Fabrikmethoden für den `ForkJoinPool` eingeführt, der das sogenannte *Work-Stealing*-Verfahren unterstützt und dessen Arbeitsweise wir in Kapitel 13 besprechen werden. Abbildung 6-1 zeigt einen Auszug aus dem Klassendiagramm der beteiligten Klassen.

Alle Implementierungen des funktionalen Interface `Executor` stellen die `execute`-Methode bereit:

```
public interface Executor
{
  void execute(Runnable command);
}
```

Somit können allen Threadpools `Runnable`-Objekte bzw. entsprechende Lambda-Ausdrücke zur Ausführung übergeben werden:

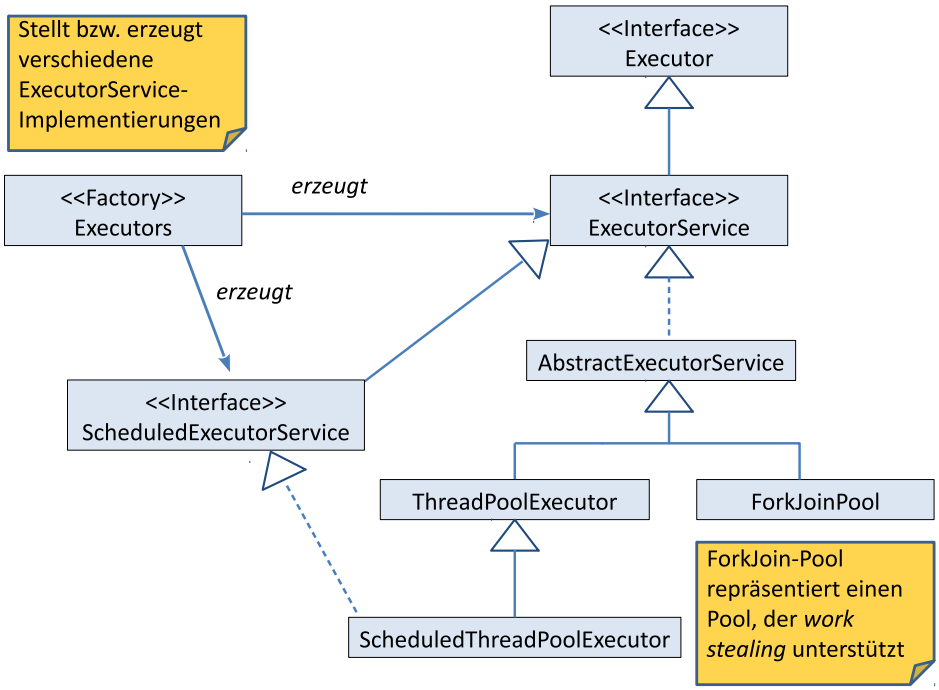

Abbildung 6-1: Klassenhierarchie des Poolkonzepts

```
ExecutorService executor = Executors.newFixedThreadPool(1);
executor.execute( () -> System.out.println("Hallo Welt") );
executor.shutdown();
```

Das Interface `ExecutorService` stellt noch weitere wichtige Methoden für den Einsatz von Threadpools zur Verfügung.

Die Threads von `ScheduledThreadPoolExecutor` bzw. `ThreadPoolExecutor` sind standardmäßig User-Threads. Da sie nicht von sich aus terminieren, muss der Threadpool mit `shutdown` kontrolliert beendet werden. Nach dem `shutdown` werden die zugewiesenen Aufgaben noch abgearbeitet, neue werden aber abgewiesen. Zu beachten ist, dass `shutdown` kein blockierender Aufruf ist. Mit `isShutdown` kann man den aktuellen Status abfragen und mit `isTerminated` erhält man die Auskunft, ob der Pool terminiert ist.

Das Herunterfahren eines Pools kann mit `shutdownNow` erzwungen werden. Hierbei werden alle aktiven Tasks des Pools mit `interrupt` zum Aufhören aufgefordert. Die in der Queue liegenden, wartenden Aufträge werden zurückgegeben. Diese Methode garantiert nicht, dass sich der Pool sofort beendet, da die Reaktion der Tasks auf `interrupt` implementierungsabhängig ist. Neben der nicht blockierenden Methode `shutdown` gibt es noch die blockierende `awaitTermination(long timeout, TimeUnit`

unit), die erst zurückkehrt, wenn alle Threads terminiert sind oder die angegebene Zeit abgelaufen ist.

6.1.1 Executors mit eigener ThreadFactory

Bei einigen der Fabrikmethoden kann eine ThreadFactory als Argument übergeben werden. Codebeispiel 6.1 zeigt die Erzeugung eines *CachedThreadPool*, dessen Threads alle die *Daemon*-Eigenschaft und eine geringe Priorität besitzen. Dieser Pool muss daher nicht mit shutdown beendet werden.

```
final ExecutorService executor = Executors.newCachedThreadPool(
   new ThreadFactory()
   {
     @Override
     public Thread newThread(Runnable r)
     {
       Thread th = new Thread(r, "MyFactoryThread");
       th.setPriority(Thread.MIN_PRIORITY);
       th.setDaemon(true);
       return th;
     }
   });
```

Codebeispiel 6.1: Ein Threadpool mit eigener Factory

Ein solcher Pool wird automatisch heruntergefahren, sobald der letzte User-Thread fertig ist.

6.1.2 Explizite ThreadPoolExecutor-Erzeugung

Man kann auch direkt einen Threadpool erzeugen, ohne die Fabrikmethoden von Executors zu benutzen. Für die direkte Erzeugung eines ThreadPoolExecutor steht unter anderem z. B. folgender Konstruktor zur Verfügung:

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          RejectedExecutionHandler handler)
```

Über die Parameter corePoolSize und maximumPoolSize kann die Anzahl der zur Verfügung gestellten Threads gesteuert werden. Es wird sichergestellt, dass mindestens corePoolSize Threads existieren. Bei Bedarf werden bis maximumPoolSize Threads erzeugt. Sind corePoolSize

und `maximumPoolSize` gleich, so handelt es sich um einen *fixed-size* Pool. Die beiden Parameter können bei den meisten Pools auch zur Laufzeit geändert werden. Mit `keepAliveTime` und `unit` wird die Zeit angegeben, wie lange unbenutzte Threads gehalten werden. Die `workQueue` wird benutzt, um übergebene Tasks ggf. zwischenzuspeichern. Es gibt dafür verschiedene Implementierungen, die wir in Kapitel 10 diskutieren. Mit `handler` wird angegeben, wie mit Tasks bei einer vollen Queue zu verfahren ist. Zur Verfügung stehen hier folgende Strategien:

- `AbortPolicy`: Entspricht dem Defaultverhalten. Es wird eine `RejectedExecutionException` geworfen.
- `CallerRunsPolicy`: Hier wird der Task abgelehnt. Er wird aber von dem `execute` aufrufenden Thread ausgeführt. Es findet in diesem Fall ein blockierender Aufruf statt.
- `DiscardOldestPolicy`: Der am längsten wartende Task wird zugunsten des neuen verdrängt.
- `DiscardPolicy`: Hier wird der übergebene Task ohne eine spezielle Meldung, z. B. über eine Exception, ignoriert.

6.1.3 Benutzerdefinierter ThreadPoolExecutor

Neben den obigen Konfigurationsmöglichkeiten kann ein Executor auch durch Ableitung angepasst werden. Codebeispiel 6.2 zeigt einen `Executor`, der die Anzahl und Ausführungsdauer der Tasks protokolliert.

```
public class MonitoredExecutor extends ThreadPoolExecutor          ❶
{
  private static final ThreadLocal<Long>
                       startTime = new ThreadLocal<>();           ❷
  private final AtomicLong counter = new AtomicLong();

  public MonitoredExecutor()
  {
    super( 4, Runtime.getRuntime().availableProcessors(),
          30, TimeUnit.SECONDS,
          new ArrayBlockingQueue<>(10),
          new ThreadPoolExecutor.AbortPolicy() );
  }

  @Override
  protected void beforeExecute(Thread t, Runnable r)               ❸
  {
    counter.incrementAndGet();
    startTime.set(System.nanoTime());
    super.beforeExecute(t, r);
  }
```

```
  @Override
  protected void afterExecute(Runnable r, Throwable t)          ❹
  {
    super.afterExecute(r, t);
    System.out.println("Dauer : "
              + (System.nanoTime() - startTime.get())
              + "(Thread : " + Thread.currentThread() + ")" );
  }

  @Override
  protected void terminated()                                   ❺
  {
    super.terminated();
    System.out.println("Anzahl Tasks " + counter.get() );
  }
}
```

Codebeispiel 6.2: Ein durch Ableitung angepasster Executor

Die Klasse `MonitoredExecutor` ist von `ThreadPoolExecutor` abgeleitet (❶) und konfiguriert den `ThreadPoolExecutor` entsprechend (❷). Es gibt verschiedene *Hook*-Methoden, die bei bestimmten Ereignissen aufgerufen werden. Zu Beginn der Ausführung eines Tasks wird die `beforeExecute`-Methode aufgerufen. In dem Beispiel wird die Startzeit protokolliert und der Task-Zähler um eins erhöht (❸). Ist ein Task zu Ende, wird die `afterExecute`-Methode aufgerufen. Hier werden die Verarbeitungszeit und der Träger-Thread auf die Konsole ausgegeben (❹). Beim `shutdown` des Pools wird durch `terminated` die Anzahl der bearbeiteten Tasks ausgegeben (❺).

6.2 Future- und Callable-Schnittstelle

Bisher haben alle nebenläufig auszuführenden Aufgaben das `Runnable`-Interface implementiert. Soll ein Task eine Rückgabe liefern, kann man das über die Verwendung eines speziellen *Rückgabe-Attributs* realisieren. Codebeispiel 6.3 zeigt eine mögliche Implementierung.

```
public class RunnableWithReturn<T> implements Runnable
{
  private T returnValue;
  private volatile Thread self;
  // ...

  public void run()
  {
    self = Thread.currentThread();
    // Berechnung und Ergebnis in returnValue abspeichern
  }
```

```
  // Blockierende Abfrage des Return-Werts
  public T get()
  {
    self.join();
    return returnValue;
  }
}
```

Codebeispiel 6.3: Aufbau eines `Runnable` mit Rückgabe

Die Lösung beinhaltet aber verschiedene Probleme: `self` kann `null` sein und der Task hat keine Möglichkeit, auftretende Ausnahmen während der Ausführung zurückzugeben. Eine einfache Lösung dafür ist die Verwendung eines sogenannten `FutureTask`-Objekts.

6.2.1 Callable, Future und FutureTask

Ein `FutureTask` ist ein Wrapper für ein `Callable`-Objekt. Das funktionale Interface `Callable` besteht aus einer Methode:

```
public interface Callable<V>
{
  V call() throws Exception;
}
```

Der `FutureTask` implementiert `Runnable` und kann somit einem Thread zur Ausführung übergeben werden. Über seine `get`-Methode erhält man Zugriff auf das Ergebnis. Das folgende Codebeispiel zeigt die Verwendung:

```
Callable<String> callable = () -> { return "Hallo"; };
FutureTask<String> futureTask = new FutureTask<>(callable);
new Thread(futureTask).start();
System.out.println("Ergebnis: " + futureTask.get() );
```

Das `Callable` ist hier sehr einfach gehalten, es liefert lediglich einen String. Nach der Verpackung über einen `FutureTask` wird es einem Thread zur Ausführung übergeben. Mit `futureTask.get` wird dann so lange gewartet, bis das Ergebnis der asynchronen Ausführung des `Callable` vorliegt. Ein `FutureTask` implementiert neben dem `Runnable`- auch das `Future`-Interface, das wir im nächsten Abschnitt besprechen.

6.2.2 Callable, Future und ExecutorService

Mit dem `ExecutorService` wird das Interface `Future` eingeführt, mit dessen Hilfe die Ergebnisrückgabe einer asynchronen Berechnung einfach und

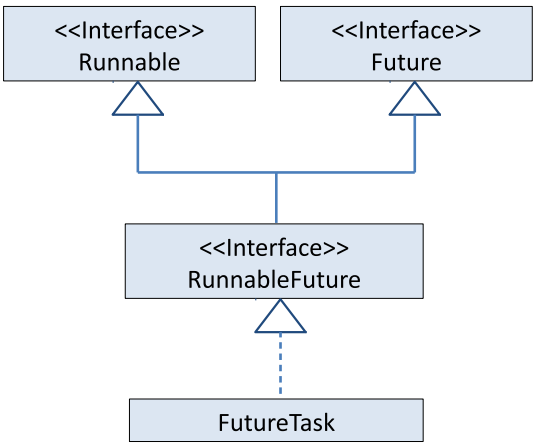

Abbildung 6-2: Vererbungshierarchie von `FutureTask`

einheitlich realisiert wird. Über das `Future`-Objekt kann neben dem Ergebnis auch der Status der Berechnung abgefragt werden.

Ein `Callable` wird einem `ServiceExecutor` über die Methode `submit` zur Ausführung übergeben, die ein `Future`-Objekt zurückliefert. Über dieses kann die Rückgabe erfragt werden. Das Codebeispiel 6.4 demonstriert eine Verwendung.

```
public static void main(String[] args)
{
  Callable<Integer> callable = new Callable<Integer>()                ❶
  {
    @Override
    public Integer call() throws Exception                            ❷
    {
      return 42;
    }
  };

  ExecutorService executor = Executors.newCachedThreadPool();
  Future<Integer> future = executor.submit( callable );               ❸

  try
  {
    Integer result = future.get();                                    ❹
    System.out.println( result );
  }
  catch (InterruptedException | ExecutionException e)                 ❺
  {
    // ...
  }
}
```

Codebeispiel 6.4: Ein Beispiel mit einem `Callable` und `Future`

Statt eines `Runnable`-Objekts wird jetzt ein parametrisiertes `Callable`-Objekt benutzt (❶). Die zu implementierende `call`-Methode hat eine typisierte Rückgabe (❷). Das `Callable` wird über `submit` dem Threadpool übergeben und man erhält ein `Future`-Objekt (❸). Da `Callable` auch ein funktionales Interface ist, kann man auch schreiben:

```
Future<Integer> future = executor.submit( () -> 42 );
```

Das Ergebnis der asynchronen Berechnung kann nun über die `get`-Methode des `Future`-Objekts erfragt werden (❹). Dabei bleibt `get` so lange blockiert, bis das Ergebnis vorliegt. Die `get`-Methode kann die beiden Ausnahmen `InterruptedException` und `ExecutionException` werfen (❺). Die Fehlerbehandlung im Zusammenhang mit `Callable` und `Future` besprechen wir später noch genauer. Abbildung 6-3 zeigt den Ablauf im Sequenzdiagramm.

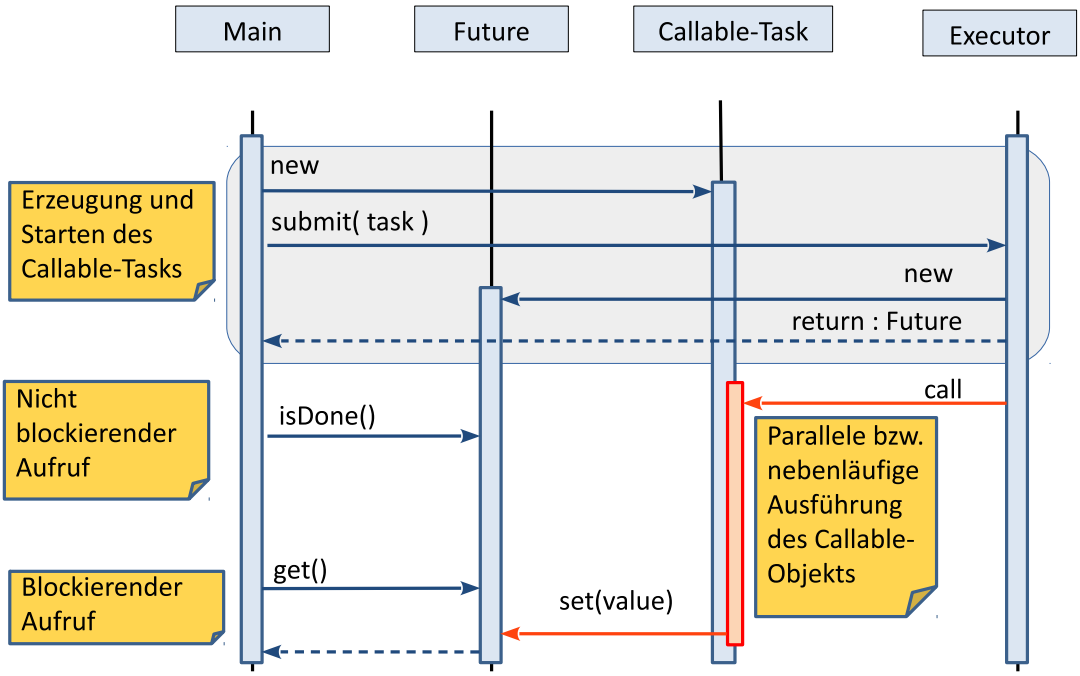

Abbildung 6-3: Funktionsweise des Future-Patterns

Ein `Future<V>` bietet neben `get` noch weitere Methoden an. Durch `get(long timeout, TimeUnit unit)` kann der Aufrufer eine maximale Wartezeit angeben. Ist das Ergebnis nach der vorgegebenen Zeit nicht verfügbar, wird eine `TimeOutException` geworfen. Mit `isDone` kann der Bearbeitungsstatus abgefragt werden. Zum Abbrechen kann `cancel(boolean mayInterruptIfRunning)` benutzt werden. Ist der Task noch nicht gestartet, wird er nicht ausgeführt. Befindet er sich mit-

ten in der Abarbeitung, wird in Abhängigkeit vom aufrufenden Parameter dem ausführenden Thread ggf. ein *Interrupt* gesendet. Der Task muss in dem Fall dann so implementiert sein, dass er den *Interrupt* berücksichtigt. Mit `isCancelled` kann geprüft werden, ob der Task abgebrochen wurde.

6.3 Callable und ThreadPoolExecutor

Mit `newCachedThreadPool` und `newFixedThreadPool` erhält man ein `ThreadPoolExecutor`-Objekt mit dem Interface `ExecutorService`. Bei Bedarf kann es auf `ThreadPoolExecutor` gecastet werden, um weitere Einstellungen des Pools vorzunehmen (siehe dazu das entsprechende API). Dagegen wird mit `newSingleThreadExecutor` ein Adapter zurückgeliefert, der keine weiteren Einstellungen erlaubt.

An einen `ExecutorService` kann man einen Task vom Typ `Runnable` oder `Callable` senden, der vom Pool möglichst bald ausgeführt wird. Alle der folgenden Methoden kehren nach dem Aufruf unmittelbar zurück, sodass der aufrufende Thread seine Tätigkeit nebenläufig ausführen kann:

- `Future<?> submit(Runnable task)`: Das zurückgegebene Objekt wird verwendet, um `isDone`, `cancel` und `isCancelled` aufzurufen. Der `get`-Aufruf liefert bei Fertigstellung nur den Wert `null`.
- `Future<T> submit(Runnable task, T result)`: Im Vergleich zum obigen `submit` liefert `get` das vorgegebene `result`-Objekt als Ergebnis zurück.
- `Future<T> submit(Callable<T> task)`: In dieser Version wird ein `Future`-Objekt zurückgeliefert, über das das Ergebnis der Berechnung abgeholt werden kann.

Schauen wir uns die Verwendung von `ExecutorService`, `Callable` und `Future` etwas näher an. Die Klasse `FindWordInFiles` aus Codebeispiel 6.5 realisiert ein `Callable` zur asynchronen Suche nach einem Wort in einer Datei.

```
class FindWordInFiles implements Callable<List<String>>          ❶
{
  private final Pattern searchPattern;
  private final Path path;   // Dateipfad

  public FindWordInFiles(Path path, String search)
  {
    this.path = path;
    this.searchPattern = Pattern.compile(".*\\b"+search+"\\b.*");
  }
```

```
  public List<String> call() throws IOException                    ❷
  {
    List<String> result = new ArrayList<>();
    List<String> lines = Files.readAllLines(path,
                                        StandardCharsets.UTF_8);

    int count = 0;
    for (String line : lines)
    {
      count++;
      if ( searchPattern.matcher(line).matches() )
      {
        result.add( path + " " + count + " : " + line);
      }
    }

    return result;
  }
}
```

Codebeispiel 6.5: Beispiel für eine Suche nach einem Wort in einer Datei

Die Rückgabe des asynchron ausgeführten Tasks ist `List<String>` (❶,❷), wobei jeder Eintrag der Liste die Zeile enthält, in der das Suchwort vorkommt.

Im Codebeispiel 6.6 wird das Wort »Haus« parallel in drei Textdateien gesucht.

```
public class FindWordBeispiel
{
  public static void main(String[] args)
  {
    ExecutorService pool = Executors.newCachedThreadPool();
    String search = "Haus";

    Callable<List<String>> task1 = new FindWordInFiles(            ❶
                           Paths.get("Text1.txt"), search);
    Callable<List<String>> task2 = new FindWordInFiles(
                           Paths.get("Text2.txt"), search);
    Callable<List<String>> task3 = new FindWordInFiles(
                           Paths.get("Text3.txt"), search);

    Future<List<String>> task1Future = pool.submit(task1);         ❷
    Future<List<String>> task2Future = pool.submit(task2);
    Future<List<String>> task3Future = pool.submit(task3);

    try
    {
      List<String> task1Liste = task1Future.get();                 ❸
      List<String> task2Liste = task2Future.get();
      List<String> task3Liste = task3Future.get();
```

```
      task1Liste.forEach(System.out::println);                    ❹
      task2Liste.forEach(System.out::println);
      task3Liste.forEach(System.out::println);
    }
    catch (InterruptedException | ExecutionException e)
    {
      e.printStackTrace();
    }
    pool.shutdown();
  }
}
```

Codebeispiel 6.6: Beispiel für eine parallele Suche

Zuerst werden die drei `Callable`-Objekte erzeugt (❶) und dann an den Threadpool mit `submit` einzeln übergeben (❷). Mit `get` wird anschließend auf das Ende des jeweiligen Tasks gewartet (❸), bevor die Ergebnisse auf die Konsole ausgegeben werden (❹).

Anstatt die Tasks einzeln an den Threadpool zu übergeben, können diese auch in eine `Collection` aufgenommen und dann mit `invokeAll` auf einmal übergeben werden (vgl. Codebeispiel 6.7). Hier ist zu beachten, dass `invokeAll` blockiert und erst zurückkommt, wenn alle Tasks beendet sind.

```
List<Callable<List<String>>> tasks = new ArrayList<>();
tasks.add(new FindWordInFiles(Paths.get("Text1.txt"), search));
tasks.add(new FindWordInFiles(Paths.get("Text2.txt"), search));
tasks.add(new FindWordInFiles(Paths.get("Text3.txt"), search));

try
{
  List<Future<List<String>>> tasksFuture=pool.invokeAll(tasks);

  for( Future<List<String>> future : tasksFuture )
  {
    future.get().forEach(System.out::println);
  }
}
catch (InterruptedException | ExecutionException e)
{
  e.printStackTrace();
}
```

Codebeispiel 6.7: Beispiel für die Verwendung von `invokeAll`

In den beiden Codebeispielen 6.6 und 6.7 musste mit der Ausgabe immer so lange gewartet werden, bis auch der langsamste Task fertig war. Das kann zu unnötigen Wartezeiten führen, da man mit der Veröffentlichung der Ergebnisse beginnen könnte, wenn der erste zu Ende ist. Um diese Limitierung zu umgehen, kann ein `CompletionService` eingesetzt werden.

Ein `CompletionService` verwaltet eine interne Queue, in die die `Future`-Objekte eingestellt werden, sobald die zugehörigen Tasks beendet sind. Codebeispiel 6.8 demonstriert dies.

```
ExecutorService pool = Executors.newCachedThreadPool();
String search = "Haus";
List<Callable<List<String>>> tasks = new ArrayList<>();
tasks.add(new FindWordInFiles(Paths.get("Text1.txt"), search));
tasks.add(new FindWordInFiles(Paths.get("Text2.txt"), search));
tasks.add(new FindWordInFiles(Paths.get("Text3.txt"), search));

CompletionService<List<String>> completionService =                ❶
                          new ExecutorCompletionService<>(pool);
tasks.forEach(completionService::submit);                          ❷

try
{
  for (int i = 0; i < tasks.size(); i++)
  {
    Future<List<String>> future = completionService.take();        ❸
    future.get().forEach(System.out::println);
  }
}
catch (InterruptedException | ExecutionException e)
{
  e.printStackTrace();
}
pool.shutdown();
```

Codebeispiel 6.8: Beispiel für die Verwendung von `CompletionService`

Bei der Erzeugung eines `ExecutorCompletionService`, einer Implementierung von `CompletionService`, wird der zu benutzende Threadpool angegeben (❶). Danach werden ihm die Tasks mit `submit` übergeben (❷). Sobald ein Task beendet ist, kann dessen `Future`-Objekt mit `take` aus der internen Queue des `CompletionService` entnommen werden (❸).

Hinweis

- Wird ein `ExecutorService` nicht mehr benötigt, sollte dessen `shutdown`-Methode aufgerufen werden, damit die belegten Ressourcen an das Betriebssystem zurückgegeben werden.
- Alle von einem Thread ausgeführten Tasks teilen sich dieselben Threadlokalen Daten. Um Probleme zu vermeiden, sollten Tasks am Anfang diese Daten korrekt initialisieren.

6.4 Callable und ScheduledThreadPoolExecutor

Für Tasks, die mehrfach bzw. periodisch ausgeführt werden sollen, steht die Klasse `ScheduledThreadPoolExecutor` mit dem Interface `ScheduledExecutorService` zur Verfügung. Instanzen können wie bei `ThreadPoolExecutor` am bequemsten über die Fabrikmethoden `newScheduledThreadPool` bzw. `newSingleThreadScheduledExecutor` der `Executors`-Klasse erhalten werden. Das `ScheduledExecutorService`-Interface ist von `ExecutorService` abgeleitet und stellt zusätzliche Methoden zur periodischen Ausführung von Tasks bereit.

Mit den `schedule`-Methoden kann ein `Runnable`- bzw. `Callable`-Task nach der angegebenen Zeit einmal ausgeführt werden. Für die periodische Ausführung kann `scheduleAtFixedRate` verwendet werden. Nach einer Anfangsverzögerung wird der Task periodisch gestartet. Wenn für die Ausführung einer Wiederholung länger als die angegebene Periode benötigt wird, werden die folgenden entsprechend später ablaufen. Es wird garantiert, dass sich Aktivitäten nie überlappen.

Das folgende Codebeispiel zeigt, wie der `ScheduledExecutorService` eingesetzt werden kann. Es wird ein Task gestartet, der jede Sekunde einen Signalton ausgibt (❶). Parallel dazu wird ein Task eingestellt, der nach 10 Sekunden den Signalton stoppt und den Pool herunterfährt (❷).

```
ScheduledExecutorService scheduler =
               Executors.newScheduledThreadPool(1);
ScheduledFuture<?> beeperHandle =
         scheduler.scheduleAtFixedRate(                          ❶
                     Toolkit.getDefaultToolkit()::beep,
                     0, 1, TimeUnit.SECONDS);

scheduler.schedule( () -> { beeperHandle.cancel(true);
                            scheduler.shutdown();                ❷
                          },
                    10, TimeUnit.SECONDS);
```

Codebeispiel 6.9: Beispiel für geplante Ausführungen

6.5 Callable und ForkJoinPool

In Java 7 wurde zusammen mit dem ForkJoin-Framework (siehe Kapitel 13) der `ForkJoinPool` eingeführt, der in Java 8 noch mal überarbeitet wurde. Das ist der Standardpool, der für die Java-internen Parallelisierungen, wie z. B. bei den parallelen Array-Methoden und *parallel Streams* (siehe Kapitel 14), eingesetzt wird.

Der `ForkJoinPool` benutzt für seine interne Verwaltung ein *Work-Stealing*-Verfahren. Bei diesem Verfahren besitzt im Prinzip jeder Thread eine eigene Task-Queue, aus der er seine Aufträge holt bzw. Aufträge, die er generiert, hineinstellt [6]. Ist seine Queue leer, holt er sich vom Ende anderer Task-Queues Aufgaben und bearbeitet diese. Ein *Work-Stealing*-Pool kommt insbesondere mit einer vorher nicht abschätzbaren, hohen Anzahl von Tasks mit azyklischen Abhängigkeiten[1] zurecht, wie sie typischerweise in *Divide-and-Conquer*-Algorithmen auftreten. Sind die Tasks dagegen unabhängig voneinander, wie in unseren bisherigen Beispielen, besitzt er gegenüber einem `ThreadPoolExecutor` keine Vorteile, weil in diesem Fall insgesamt nur eine Queue benötigt wird.

Einen `ForkJoinPool` kann man sich entweder über die Fabrikmethoden der `Executors`-Klasse erzeugen oder durch den direkten Aufruf eines der folgenden Konstruktoren:

- `ForkJoinPool()`
- `ForkJoinPool(int parallelism)`
- ```
  ForkJoinPool(int parallelism,
  ForkJoinPool.ForkJoinWorkerThreadFactory factory,
  Thread.UncaughtExceptionHandler handler,
  boolean asyncMode)
  ```

Der Parameter `parallelism` gibt an, wie viele Threads benutzt werden sollen. Mit `asyncMode` kann man spezifizieren, dass die von einem Task neu generierten unabhängigen Aufgaben (*forked tasks*) nach dem FIFO-Prinzip abgearbeitet werden. Das Standardverhalten ist die LIFO-Abfertigung, weil in der Regel Sub-Tasks wie »Funktionsaufrufe« eingesetzt werden (siehe auch Abschnitt 13.3).

Der CommonPool

Um das ständige Erzeugen und Schließen von Pools zu vermeiden, benutzt Java einen globalen Threadpool, der bei der ersten Verwendung von Java-eigenen Parallelisierungskonzepten angelegt wird. Zugriff auf diesen Pool erhält man mit `ForkJoinPool.commonPool`. Möchte man den *CommonPool* konfigurieren, so kann man dies über das Setzen von System-Properties[2] bewerkstelligen:

- `java.util.concurrent.ForkJoinPool.common.parallelism`
- `java.util.concurrent.ForkJoinPool.common.threadFactory`

[1] Es gibt keine gegenseitige Abhängigkeit zwischen Tasks. Ihre Beziehungen können durch eine Baumstruktur beschrieben werden (siehe Kapitel 13).

[2] Aufrufparameter beim Starten der virtuellen Maschine.
  ```

- `java.util.concurrent.ForkJoinPool.common.exceptionHandler`

Der Defaultwert für `parallelism` ist in der Regel `Runtime.getRuntime().availableProcessors() - 1`, falls mehrere Kerne zur Verfügung stehen. Die Defaultkonfiguration kann auch innerhalb der Anwendung geändert werden. Hierbei muss beachtet werden, dass dies vor dem ersten Aufruf von `ForkJoinPool.commonPool` geschieht. Der folgende Code zeigt, wie man die Anzahl der verwendeten Threads setzen kann:

```
System.setProperty(
 "java.util.concurrent.ForkJoinPool.common.parallelism", "4");
ForkJoinPool commonPool = ForkJoinPool.commonPool();
```

Man beachte, dass Java selbst für die internen parallelen Konzepte `ForkJoinPool.commonPool` aufruft.

6.6 Exception-Handling

Eine wichtige Frage ist, wie mit Fehlern umgegangen wird, die in nebenläufig ausgeführten Tasks auftreten. Betrachten wir ein Beispiel, in dem wir einen Task mit einer Division durch null an einen Pool senden:

```
ExecutorService executor = Executors.newCachedThreadPool();
executor.execute( () -> System.out.println(1 / 0) );
```

Wir erhalten daraufhin die Meldung

```
Exception in thread "pool-1-thread-1" java.lang.ArithmeticException: /
    by zero
  at kap6.threadpool.ExceptionBeispiel1.lambda$0(ExceptionBeispiel1.
      java:12)
    ...
```

Der Pool-Thread wird durch die Ausnahme terminiert und der Default-Handler wird in diesem Fall aufgerufen. Wenn anstatt `execute` nun `submit` verwendet wird, also

```
ExecutorService executor = Executors.newCachedThreadPool();
executor.submit( () -> System.out.println(1/0) );
```

erscheint keine Ausgabe auf der Konsole. Der Grund ist, dass bei der Verwendung von `submit` jede nicht behandelte Ausnahme von `Runnable` oder `Callable` abgefangen wird:

```
public void run()
{
  Throwable thrown = null;
  try
  {
    while (!isInterrupted())
    {
      runTask(getTaskFromWorkQueue());
    }
  }
  catch (Throwable e)
  {
    thrown = e;
  }
  finally
  {
    threadExited(this, thrown);
  }
}
```

und erst bei einem Zugriff mit `get` auf das zurückgegebene `Future`-Objekt die Ausnahme auslöst. Mit

```
ExecutorService executor = Executors.newCachedThreadPool();
Future<?> future = executor.submit(
                          () -> System.out.println(1 / 0) );
try
{
  future.get();
}
catch (InterruptedException | ExecutionException e)
{
  e.printStackTrace();
}
```

erhält man auf der Konsole

```
java.util.concurrent.ExecutionException: java.lang.ArithmeticException:
     / by zero
   at java.util.concurrent.FutureTask.report(Unknown Source)
   ...
```

Alternativ kann man die Exception im Task abfangen und loggen. Damit der Aufrufende über die Ausnahme in Kenntnis gesetzt wird, sollte die Exception weitergegeben werden:

```
future = executor.submit(() ->
   {
     try
     {
       System.out.println(1 / 0);
     }
     catch (Exception ex)
     {
        // Eigene Fehlerbehandlung etwa Loggen
        System.out.println("Ausführungsfehler = " + ex);
        throw ex; // Damit man über get die Ausnahme noch sieht
     }
   });
```

6.7 Tipps für das Arbeiten mit Threadpools

Im Folgenden sind einige nützliche Tipps zusammengestellt, die sich in der Praxis bewährt haben.

Temporäre Änderung des Thread-Namens

Beim Debugging ist es äußerst hilfreich, wenn man Threads über sinnvolle Namen identifizieren kann. Das Defaultschema für die Pool-Thread-Benennung ist `pool-N-thread-M`, wobei `N` die Poolnummer und `M` die Thread-Nummer ist.

Eine einfache Lösung, mit der Thread-Namen temporär geändert werden können, zeigt die folgende Hilfsmethode. Dabei wird das übergebene `Callable`-Objekt in einem Wrapper gekapselt. Der aktuelle Thread-Name wird vor der Ausführung der Aktivität abgespeichert, geändert (❶) und am Ende wieder hergestellt (❷).

```
public static <T> Future<T> submit(ExecutorService service,
    Callable<T> task, String name)
{
  return service.submit(() ->
   {
     Thread current = Thread.currentThread();
     String oldname = current.getName();
     current.setName(name);                                    ❶
     try
     {
       return task.call();
     } finally
     {
       current.setName(oldname);                               ❷
     }
   });
}
```

Anzahl der Pool-Threads

Neben der Frage, ob man Daemon-Threads nutzen möchte oder nicht, sollte man sich auch Gedanken über die Poolgröße machen. Eine angemessene Poolgröße hängt einerseits von der Anzahl und andererseits von der Art der zu bearbeitenden Tasks ab. Die Frage ist also, ob sie z. B. eher IO-intensiv oder rechenintensiv sind. Goetz et al. [16] geben folgende Faustregel für die Anzahl der Pool-Threads an

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

mit

$$\begin{aligned} N_{cpu} &= \text{Anzahl der zur Verfügung stehenden Kerne} \\ U_{cpu} &= \text{Auslastung der CPU, } 0 \leq U_{cpu} \leq 1 \\ \frac{W}{C} &= \text{Verhältnis zwischen Warte- und Rechenzeit} \end{aligned}$$

Für rechenintensive Tasks und Vollauslastung sollte

$$\begin{aligned} N_{threads} &= N_{cpu} + 1 \\ &= \texttt{Runtime.getRuntime().availableProcessors()} + 1 \end{aligned}$$

gewählt werden, da selbst bei rechenintensiven Aufgaben es gelegentlich *page faults* und somit Unterbrechungen bzw. Wartezeiten gibt.

6.8 Zusammenfassung

In Java werden viele praktische Konzepte für den Umgang mit Threads angeboten, die auch in der Praxis angewendet werden sollten. Durch die Klassenmethoden von `Executors` können bequem verschiedene Threadpools erzeugt werden, die alle das `ExecutorService`-Interface implementieren. Die Pools können sowohl Tasks vom Typ `Runnable` (ohne explizite Wertrückgabe durch Aufruf der `execute`-Methode) als auch vom Typ `Callable` (mit Wertrückgabe durch die `submit`-Methode) ausführen. Das durch `submit` zurückgegebene `Future`-Objekt dient dazu, den Stand asynchroner Verarbeitung abzufragen. Es gibt außerdem mehrere Möglichkeiten zur Konfiguration eines Threadpools.