Task Execution and Scheduling

1. Task execution
2. Task scheduling
3. Async and scheduled methods

1. Task Execution

- Overview
- The TaskExecutor interface
- How to use a TaskExecutor
- Synchronous execution
- Simple asynchronous execution
- Asynchronous execution with pooling
- Submitting callable tasks
- Additional task executor classes available

Demo package: `demo.taskexecution`

Overview

- Spring provides mechanisms that allow you to execute and schedule tasks asynchronously
  - Very similar to the executor/scheduler API in Java

- You should use the Spring APIs, rather than the analogous Java APIs
  - More idiomatic in Spring
  - Easier to configure bean properties (e.g. thread pooling, etc.)
  - Easier to inject as beans into other components in your app

The TaskExecutor Interface

- Spring defines a `TaskExecutor` interface, to execute a `Runnable` task
  - Identical to `java.util.concurrent.Executor`

```
import org.springframework.core.task.TaskExecutor;

public interface TaskExecutor {
  void execute(Runnable r);
}
```

- Various implementations of `TaskExecutor` are available, allowing tasks to be executed just how you like:
  - Synchronously in the calling thread
  - In a newly created thread
  - In a thread from the thread pool
  - In a customized implementation of `TaskExecutor`

How to use a TaskExecutor

- If you want multithreading in Spring, use `TaskExecutor` rather than creating threads yourself
  - The executor framework decouples the code that creates runnable tasks from the execution policy for those tasks

- For example, rather than doing this:

```
Thread thread = new Thread(new MyRunnableTask());
thread.start();
```

☹

- … you should do this:

```
TaskExecutor executor = … some TaskExecutor implementation …
executor.execute(new MyRunnableTask1());
executor.execute(new MyRunnableTask2());
```

😛

Synchronous Execution

- Spring provides a `SyncTaskExecutor` class
  - Executes task synchronously on the calling thread

- Example

```
@Configuration
public class MyConfig {

    @Bean
    public TaskExecutor syncTaskExecutor() {
        return new SyncTaskExecutor();
    }
}
```

MyConfig.java

```
public static void demoSync(ApplicationContext ctx) {
    TaskExecutor ex = ctx.getBean("syncTaskExecutor", TaskExecutor.class);
    Util.display("Before");
    ex.execute(new MyRunnableT
    Util.display("After");
}
```

Application.java

```
Fri Dec 30 11:06:26 GMT 2022, thread 01: Before
Fri Dec 30 11:06:26 GMT 2022, thread 01: My task 1
Fri Dec 30 11:06:27 GMT 2022, thread 01: My task 2
Fri Dec 30 11:06:28 GMT 2022, thread 01: My task 3
Fri Dec 30 11:06:29 GMT 2022, thread 01: My task 4
Fri Dec 30 11:06:30 GMT 2022, thread 01: My task 5
Fri Dec 30 11:06:31 GMT 2022, thread 01: After
```

Simple Asynchronous Execution (1 of 3)

- Spring provides a `SimpleAsyncTaskExecutor` class
  - Executes task asynchronously on a new thread
  - Creates a brand new thread for the task (i.e. no thread pooling)

- By default, the amount of concurrency is unlimited
  - i.e. you can execute as many concurrent tasks as you like

- You can limit concurrency, to prevent runaway processing
  - Set the `concurrencyLimit` bean property
  - Blocks new tasks from starting if already at the concurrency limit

- You can also influence threading via other bean properties
  - `daemon`, `threadPriority`, `threadGroup`, `threadFactory`
  - Etc.

Simple Asynchronous Execution (2 of 3)

- Example creating a `SimpleAsyncTaskExecutor`
  - Note we've set the concurrency limit to 3

```
@Bean
public TaskExecutor simpleAsyncTaskExecutor() {
    SimpleAsyncTaskExecutor executor = new SimpleAsyncTaskExecutor();
    executor.setConcurrencyLimit(3);
    return executor;
}                                                          MyConfig.java
```

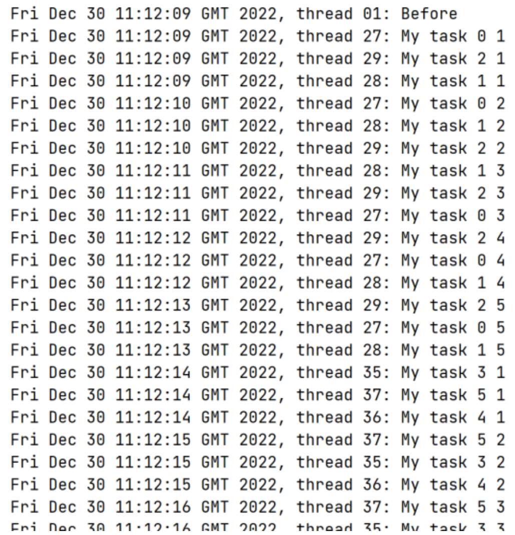
- The client code executes 10 tasks
  - But only 3 tasks will be executed concurrently, others will block

```
public static void demoSimpleAsync(ApplicationContext ctx) {
    TaskExecutor ex = ctx.getBean("simpleAsyncTaskExecutor", TaskExecutor.class);
    Util.display("Before");
    for (int i = 0; i < 10; i++)
        ex.execute(new MyRunnableTask("My task " + i, 5));
    Util.display("After");
}                                                      Application.java
```

olsen software

Simple Asynchronous Execution (3 of 3)

- Here's the console output… note the following points:
  - Up to 3 tasks are running concurrently at any given time
  - Other tasks are blocked from executing, due to concurrency limit
  - Each task runs in a new thread (i.e. no thread pooling)



```
Fri Dec 30 11:12:09 GMT 2022, thread 01: Before
Fri Dec 30 11:12:09 GMT 2022, thread 27: My task 0 1
Fri Dec 30 11:12:09 GMT 2022, thread 29: My task 2 1
Fri Dec 30 11:12:09 GMT 2022, thread 28: My task 1 1
Fri Dec 30 11:12:10 GMT 2022, thread 27: My task 0 2
Fri Dec 30 11:12:10 GMT 2022, thread 28: My task 1 2
Fri Dec 30 11:12:10 GMT 2022, thread 29: My task 2 2
Fri Dec 30 11:12:11 GMT 2022, thread 28: My task 1 3
Fri Dec 30 11:12:11 GMT 2022, thread 29: My task 2 3
Fri Dec 30 11:12:11 GMT 2022, thread 27: My task 0 3
Fri Dec 30 11:12:12 GMT 2022, thread 29: My task 2 4
Fri Dec 30 11:12:12 GMT 2022, thread 27: My task 0 4
Fri Dec 30 11:12:12 GMT 2022, thread 28: My task 1 4
Fri Dec 30 11:12:13 GMT 2022, thread 29: My task 2 5
Fri Dec 30 11:12:13 GMT 2022, thread 27: My task 0 5
Fri Dec 30 11:12:13 GMT 2022, thread 28: My task 1 5
Fri Dec 30 11:12:14 GMT 2022, thread 35: My task 3 1
Fri Dec 30 11:12:14 GMT 2022, thread 37: My task 5 1
Fri Dec 30 11:12:14 GMT 2022, thread 36: My task 4 1
Fri Dec 30 11:12:15 GMT 2022, thread 37: My task 5 2
Fri Dec 30 11:12:15 GMT 2022, thread 35: My task 3 2
Fri Dec 30 11:12:15 GMT 2022, thread 36: My task 4 2
Fri Dec 30 11:12:16 GMT 2022, thread 37: My task 5 3
Fri Dec 30 11:12:16 GMT 2022, thread 35: My task 3 3
```

olsen software

Asynchronous Execution with Pooling (1 of 3)

- Spring provides a `ThreadPoolTaskExecutor` class
  - This is the most commonly used task executor class ☺
  - (Under the covers, it wraps a Java `ThreadPoolExecutor`)
  - Executes task asynchronously, using a thread from a thread pool

- You can completely configure the thread pool
  - Core size of thread pool
  - Maximum size of thread pool
  - How long to keep unused threads in the thread pool
  - A blocking queue, to hold tasks while waiting for available thread
  - The policy about what to do if no threads/blocking queue space

Asynchronous Execution with Pooling (2 of 3)

- Example creating a `ThreadPoolTaskExecutor`
  - Note we've configured various aspects about thread pooling

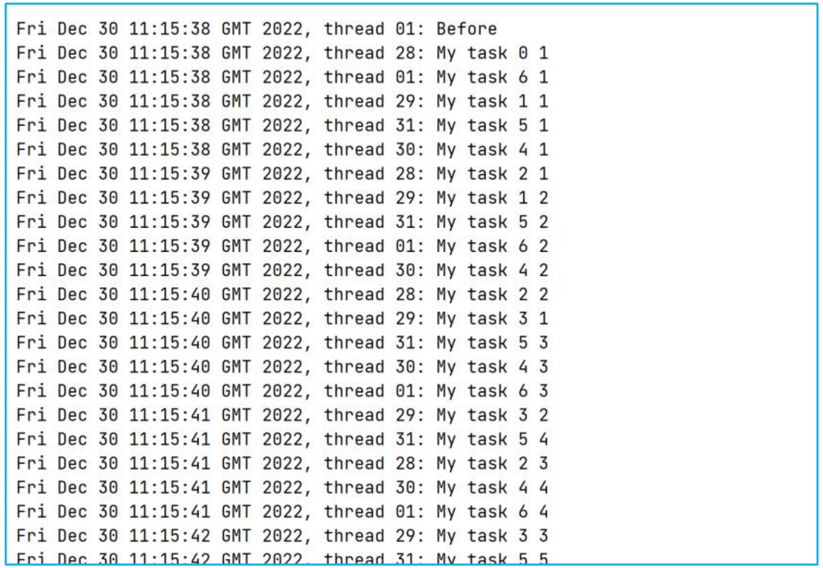
```
@Bean
public TaskExecutor threadPoolTaskExecutor() {
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
    executor.setCorePoolSize(2);
    executor.setMaxPoolSize(4);
    executor.setKeepAliveSeconds(10);
    executor.setQueueCapacity(2);
    executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
    return executor;
}                                                                   MyConfig.java
```

- The client code executes 10 tasks
  - They run as concurrently as possible, using thread-pool threads

```
public static void demoAsyncUsingThreadPool(ApplicationContext ctx) {
    TaskExecutor ex = ctx.getBean("threadPoolTaskExecutor", TaskExecutor.class);
    Util.display("Before");
    for (int i = 0; i < 10; i++)
        ex.execute(new MyRunnableTask("My task " + i, i + 1));
    Util.display("After");
}                                                                Application.java
```

Asynchronous Execution with Pooling (3 of 3)

- Here's the console output… note the following points:
  - Up to 4 tasks are running concurrently on threads from the thread pool (thread IDs 12, 14, 13, and 11 in this screenshot)
  - Surplus tasks are all executed on the calling thread (thread ID 01)



```
Fri Dec 30 11:15:38 GMT 2022, thread 01: Before
Fri Dec 30 11:15:38 GMT 2022, thread 28: My task 0 1
Fri Dec 30 11:15:38 GMT 2022, thread 01: My task 6 1
Fri Dec 30 11:15:38 GMT 2022, thread 29: My task 1 1
Fri Dec 30 11:15:38 GMT 2022, thread 31: My task 5 1
Fri Dec 30 11:15:38 GMT 2022, thread 30: My task 4 1
Fri Dec 30 11:15:39 GMT 2022, thread 28: My task 2 1
Fri Dec 30 11:15:39 GMT 2022, thread 29: My task 1 2
Fri Dec 30 11:15:39 GMT 2022, thread 31: My task 5 2
Fri Dec 30 11:15:39 GMT 2022, thread 01: My task 6 2
Fri Dec 30 11:15:39 GMT 2022, thread 30: My task 4 2
Fri Dec 30 11:15:40 GMT 2022, thread 28: My task 2 2
Fri Dec 30 11:15:40 GMT 2022, thread 29: My task 3 1
Fri Dec 30 11:15:40 GMT 2022, thread 31: My task 5 3
Fri Dec 30 11:15:40 GMT 2022, thread 30: My task 4 3
Fri Dec 30 11:15:40 GMT 2022, thread 01: My task 6 3
Fri Dec 30 11:15:41 GMT 2022, thread 29: My task 3 2
Fri Dec 30 11:15:41 GMT 2022, thread 31: My task 5 4
Fri Dec 30 11:15:41 GMT 2022, thread 28: My task 2 3
Fri Dec 30 11:15:41 GMT 2022, thread 30: My task 4 4
Fri Dec 30 11:15:41 GMT 2022, thread 01: My task 6 4
Fri Dec 30 11:15:42 GMT 2022, thread 29: My task 3 3
Fri Dec 30 11:15:42 GMT 2022, thread 31: My task 5 5
```

olsen software

Submitting Callable Tasks

- All the examples so far have executed `Runnable` tasks
  - i.e. no return value

- `ThreadPoolTaskExecutor` also allows you to submit `Callable<T>` tasks, which eventually return a result
  - Call `submit()`
  - Returns a `Future<T>` object, representing the pending result

- For a complete example, see:
  - `MyCallableTask` class, implements `Callable<Integer>`
  - `MyConfig` class, see `threadPoolTaskExecutor()`
  - `Main` class, see `demoAsyncUsingThreadPoolWithResults()`

Additional Task Executor Classes Available

- Spring provides several additional implementations of `TaskExecutor`, if you need even more control
    - `ConcurrentTaskExecutor`
    - `SimpleThreadPoolTaskExecutor`
    - `WorkManagerTaskExecutor`

- See Spring docs for full details

2. Task Scheduling

- Overview
- The TaskScheduler interface
- Creating a TaskScheduler bean
- Scheduling tasks with a fixed delay
- Scheduling tasks at a fixed rate
- Scheduling tasks based on a trigger

Demo package: `demo.taskscheduling`

Overview

- The previous section discussed `TaskExecutor`
  - Executes a task immediately (according to thread pool constraints, if you specified any)

- Spring also provides a `TaskScheduler` interface
  - Schedules a task for execution at some time in the future
  - E.g. with a fixed delay
  - E.g. at a fixed rate
  - E.g. based on trigger info

The TaskScheduler Interface

- Here's the definition of the `TaskScheduler` interface
  - Similar to `java.util.concurrent.ScheduledThreadPoolExecutor`

```
import org.springframework.scheduling.TaskScheduler;

public interface TaskScheduler {

    ScheduledFuture scheduleWithFixedDelay(Runnable r, long delay);
    ScheduledFuture scheduleWithFixedDelay(Runnable r, Date startTime, long delay);

    ScheduledFuture scheduleAtFixedRate(Runnable r, long period);
    ScheduledFuture scheduleAtFixedRate(Runnable r, Date startTime, long period);

    ScheduledFuture schedule(Runnable r, Date startTime);
    ScheduledFuture schedule(Runnable r, Trigger trigger);
}
```

Creating a TaskScheduler Bean

- Spring provides the `ThreadPoolTaskScheduler` class, which is a simple implementation of `TaskScheduler`
  - This is perfectly adequate in most scenarios
  - Has various bean properties, e.g. pool size, thread priority, etc.

- All the examples in this section will use the following bean:

```java
@Configuration
public class MyConfig {

    @Bean
    public TaskScheduler taskScheduler() {
        ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler();
        scheduler.setPoolSize(3);
        return scheduler;
    }
}
```

MyConfig.java

Scheduling Tasks with a Fixed Delay

- `scheduleWithFixedDelay()` schedules tasks with a fixed delay after completion of the previous task

```
public static void demoFixedDelayExecution(ApplicationContext ctx) {

    ThreadPoolTaskScheduler sch = ctx.getBean(ThreadPoolTaskScheduler.class);

    Util.display("Scheduling a task, which prints 1,2,3 (this takes 4s). " +
                 "Fixed delay of 5s between completion of task and start of next.");

    sch.scheduleWithFixedDelay(new MyRunnableTask("MyRunnableTask1", 3), 5_000);

    try { Thread.sleep(20_000); }  catch (InterruptedException e) {}

    sch.shutdown();
}                                                                    Application.java
```

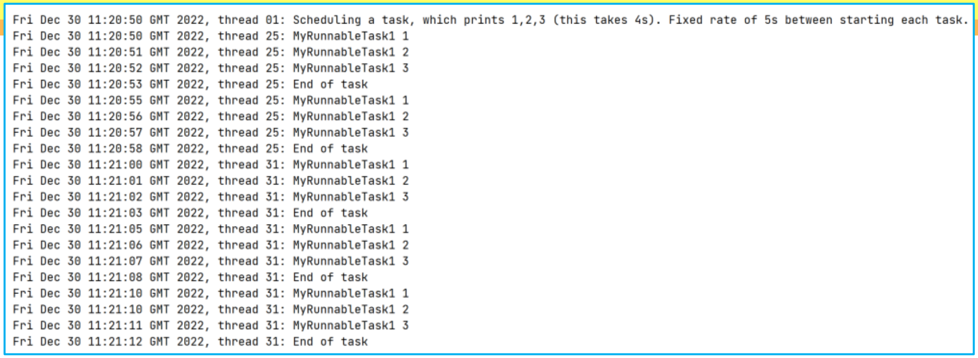
```
Fri Dec 30 11:18:18 GMT 2022, thread 01: Scheduling a task, which prints 1,2,3 (this takes 4s). Fixed delay of 5s between completion of task and start of next.
Fri Dec 30 11:18:18 GMT 2022, thread 25: MyRunnableTask1 1
Fri Dec 30 11:18:19 GMT 2022, thread 25: MyRunnableTask1 2
Fri Dec 30 11:18:20 GMT 2022, thread 25: MyRunnableTask1 3
Fri Dec 30 11:18:21 GMT 2022, thread 25: End of task
Fri Dec 30 11:18:26 GMT 2022, thread 25: MyRunnableTask1 1
Fri Dec 30 11:18:27 GMT 2022, thread 25: MyRunnableTask1 2
Fri Dec 30 11:18:28 GMT 2022, thread 25: MyRunnableTask1 3
Fri Dec 30 11:18:29 GMT 2022, thread 25: End of task
Fri Dec 30 11:18:34 GMT 2022, thread 30: MyRunnableTask1 1
Fri Dec 30 11:18:35 GMT 2022, thread 30: MyRunnableTask1 2
Fri Dec 30 11:18:36 GMT 2022, thread 30: MyRunnableTask1 3
Fri Dec 30 11:18:37 GMT 2022, thread 30: End of task
```

Scheduling Tasks at a Fixed Rate

- `scheduleAtFixedRate()` schedules tasks at a fixed rate between tasks (regardless of when they complete)

```java
public static void demoFixedRateExecution (ApplicationContext ctx) {

    ThreadPoolTaskScheduler sch = ctx.getBean(ThreadPoolTaskScheduler.class);

    Util.display("Scheduling a task, which prints 1,2,3 (this takes 4s). " +
                 "Fixed rate of 5s between starting each task.");

    sch.scheduleAtFixedRate(new MyRunnableTask("MyRunnableTask1", 3), 5_000);

    try { Thread.sleep(20_000); } catch (InterruptedException e) {}

    sch.shutdown();
}
```

Application.java



```
Fri Dec 30 11:20:50 GMT 2022, thread 01: Scheduling a task, which prints 1,2,3 (this takes 4s). Fixed rate of 5s between starting each task.
Fri Dec 30 11:20:50 GMT 2022, thread 25: MyRunnableTask1 1
Fri Dec 30 11:20:51 GMT 2022, thread 25: MyRunnableTask1 2
Fri Dec 30 11:20:52 GMT 2022, thread 25: MyRunnableTask1 3
Fri Dec 30 11:20:53 GMT 2022, thread 25: End of task
Fri Dec 30 11:20:55 GMT 2022, thread 25: MyRunnableTask1 1
Fri Dec 30 11:20:56 GMT 2022, thread 25: MyRunnableTask1 2
Fri Dec 30 11:20:57 GMT 2022, thread 25: MyRunnableTask1 3
Fri Dec 30 11:20:58 GMT 2022, thread 25: End of task
Fri Dec 30 11:21:00 GMT 2022, thread 31: MyRunnableTask1 1
Fri Dec 30 11:21:01 GMT 2022, thread 31: MyRunnableTask1 2
Fri Dec 30 11:21:02 GMT 2022, thread 31: MyRunnableTask1 3
Fri Dec 30 11:21:03 GMT 2022, thread 31: End of task
Fri Dec 30 11:21:05 GMT 2022, thread 31: MyRunnableTask1 1
Fri Dec 30 11:21:06 GMT 2022, thread 31: MyRunnableTask1 2
Fri Dec 30 11:21:07 GMT 2022, thread 31: MyRunnableTask1 3
Fri Dec 30 11:21:08 GMT 2022, thread 31: End of task
Fri Dec 30 11:21:10 GMT 2022, thread 31: MyRunnableTask1 1
Fri Dec 30 11:21:10 GMT 2022, thread 31: MyRunnableTask1 2
Fri Dec 30 11:21:11 GMT 2022, thread 31: MyRunnableTask1 3
Fri Dec 30 11:21:12 GMT 2022, thread 31: End of task
```

Scheduling Tasks based on a Trigger

- `schedule()` schedules tasks based on a `Trigger`, which you can pass as the 2nd argument

- `Trigger` is an interface, Spring has 2 implementations:
  - `PeriodicTrigger` – Similar capabilities to previous 2 slides
  - `CronTrigger` – Schedules tasks based on cron expression

```
public static void demoCronTriggeredExecution(ApplicationContext ctx) {

    ThreadPoolTaskScheduler sch = ctx.getBean(ThreadPoolTaskScheduler.class);

    Util.display("Scheduling a task to run 15 minutes past each hour, " +
                 "but only for 9am-5pm, Monday to Friday.");

    sch.schedule(new MyRunnableTask("MyRunnableTask1", 3),
                 new CronTrigger("0 15 9-17 * * MON-FRI"));
}
```

Application.java

3. Async and Scheduled Methods

- Overview
- Enabling async and scheduled methods
- Defining async methods
- Defining scheduled methods

Demo package: `demo.asyncAndScheduledMethods`

Overview

- Spring has some special annotations that make it even easier to execute and schedule methods asynchronously…
  - You just provide a suitable `TaskExecutor` bean, and Spring will use it implicitly to achieve asynchrony

- The `@Async` annotation…
  - You can annotate a bean method with `@Async`
  - When you invoke the method, Spring will automatically execute the method asynchronously via the `TaskExecutor`

- The `@Scheduled` annotation…
  - You can annotate a bean method with `@Scheduled`
  - Spring will automatically schedule the method for execution on a suitable thread, according to the specified delay/rate/trigger

Enabling Async and Scheduled Methods

- On any configuration class in your app, add one or both of the following annotations
  - `@EnableAsync`
  - `@EnableScheduling`
- You must also provide Spring with a `TaskExecutor` bean
  - E.g. `SimpleAsyncTaskExecutor`

```
@Configuration
@EnableAsync
@EnableScheduling
public class MyConfig {

    @Bean
    public TaskExecutor taskExecutor() {
        return new SimpleAsyncTaskExecutor();
    }
}
```

MyConfig.java

olsen software

Defining Async Methods

- Here's an example of how to define async methods
  - The 2nd method shows how to return a result, via

See Application.java for how to call async methods

```
@Component
public class MyBean {

    @Async
    public void myAsyncMethod() {
        Util.display("Start of myAsyncMethod");
        try {
            Thread.sleep(5_000);
        }
        catch (InterruptedException e) {}
        Util.display("End of myAsyncMethod");
    }

    @Async
    public Future<Integer> myAsyncMethodWithResult() {
        Util.display("Start of myAsyncMethodWithResult");
        try {
            Thread.sleep(5_000);
        }
        catch (InterruptedException e) {}
        Util.display("End of myAsyncMethodWithResult");
        return new AsyncResult<Integer>(42);
    }
}
```

MyBean.java

Defining Scheduled Methods

- Here's an example of how to define scheduled methods
  - Scheduled methods can't return a result (because you don't invoke them yourself - Spring schedules execution automatically)

```
@Component
public class MyBean {

    @Scheduled(fixedDelay=5000)
    public void myScheduledMethodWithFixedDelay5Seconds() {
        // Do some stuff…
    }

    @Scheduled(fixedRate=5000)
    public void myScheduledMethodAtFixedRate5Seconds() {
        // Do some stuff…
    }

    @Scheduled(cron="*/5 * * * * MON-FRI")
    public void myScheduledMethodBasedOnCronTrigger() {
        // Do some stuff…
    }
}
```

MyBean.java

Summary

- Task execution
- Task scheduling
- Async and scheduled methods