You want to build a threaded, concurrent program with Spring but don’t know what approach to use since there’s no standard approach.

SOLUTION

Use Spring’s TaskExecutor abstraction. This abstraction provides numerous implementations for many environments, including basic Java SE Executor implementations, CommonJ WorkManager implementations, and custom implementations.

In Spring all the implementations are unified and can be cast to Java SE’s Executor interface, too.

HOW IT WORKS

Threading is a difficult issue that can be particularly tedious to implement using standard threading in the Java SE environment. Concurrency is another important aspect of server-side components but has little to no standardization in the enterprise Java space. In fact, some parts of the Java Enterprise Edition specifications forbid the explicit creation and manipulation of threads.

In the Java SE landscape, many options have been introduced over the years to deal with threading and concurrency. First, there was the standard java.lang.Thread support present since Java Development Kit (JDK) 1.0. Java 1.3 saw the introduction of java.util.TimerTask to support doing some sort of work periodically. Java 5 debuted the java.util.concurrent package, as well as a reworked hierarchy for building thread pools, oriented around java.util.concurrent.Executor.

The application programming interface (API ) for Executor is simple.

**package** java.util.concurrent;  
**public interface** Executor {  
    void execute(Runnable command);  
}

ExecutorService, a subinterface, provides more functionality for managing threads and provides support to raise events to threads, such as shutdown(). There are several implementations that have shipped with the JDK since Java SE 5.0. Many of them are available via static factory methods in the java.util.concurrent package. What follows are several examples using Java SE classes.

The ExecutorService class provides a submit() method, which returns a Future<T> object. An instance of Future<T> can be used to track the progress of a thread that’s usually executing asynchronously. You can call Future.isDone() or Future.isCancelled() to determine whether the job is finished or cancelled, respectively. When you use ExecutorService and submit() inside a Runnable instance whose run method has no return type, calling get() on the returned Future returns null, or the value specified on submission.

Runnable task = **new** Runnable(){  
    **public** void run(){  
        **try**{  
            Thread.sleep( 1000 \* 60 ) ;  
            System.out.println("Done sleeping for a minute, returning! " );  
        } **catch** (Exception ex) { */\* ... \*/* }  
    }  
};  
  
ExecutorService executorService  = Executors.newCachedThreadPool() ;  
  
**if**(executorService.submit(task, Boolean.TRUE).get().equals( Boolean.TRUE ))  
    System.out.println( "Job has finished!");

With this background information, you can explore some of the characteristics of the various implementations. For example, the following is a class designed to mark the passage of time using Runnable:

**package** com.apress.springrecipes.spring3.executors;  
  
**import** java.util.Date;  
  
**public class** DemonstrationRunnable **implements** Runnable {  
    **public** void run() {  
        **try** {  
            Thread.sleep(1000);  
            } **catch** (InterruptedException e) {  
                e.printStackTrace();  
            }  
            System.out.println(Thread.currentThread().getName());  
            System.out.printf("Hello at %s \n", **new** Date());  
    }  
}

You’ll use the same instance when you explore Java SE Executors and Spring’s TaskExecutor support.

**package** com.apress.springrecipes.spring3.executors;  
**import** java.util.Date;  
**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
**import** java.util.concurrent.ScheduledExecutorService;  
**import** java.util.concurrent.TimeUnit;  
  
**public class** ExecutorsDemo {  
  
    **public** static void main(String[] args) **throws** Throwable {  
        Runnable task = **new** DemonstrationRunnable();  
  
        ExecutorService cachedThreadPoolExecutorService =  
            Executors.newCachedThreadPool();  
        **if** (cachedThreadPoolExecutorService.submit(task).get() == **null**)  
            System.out.printf("The cachedThreadPoolExecutorService "  
                + "has succeeded at %s \n", **new** Date());  
  
        ExecutorService fixedThreadPool = Executors.newFixedThreadPool(100);  
        **if** (fixedThreadPool.submit(task).get() == **null**)  
            System.out.printf("The fixedThreadPool has " +  
                "succeeded at %s \n",  
                **new** Date());  
        ExecutorService singleThreadExecutorService =  
            Executors.newSingleThreadExecutor();  
        **if** (singleThreadExecutorService.submit(task).get() == **null**)  
            System.out.printf("The singleThreadExecutorService "  
                + "has succeeded at %s \n", **new** Date());  
  
        ExecutorService es = Executors.newCachedThreadPool();  
        **if** (es.submit(task, Boolean.TRUE).get().equals(Boolean.TRUE))  
            System.out.println("Job has finished!");  
  
        ScheduledExecutorService scheduledThreadExecutorService =  
            Executors.newScheduledThreadPool(10);  
        **if** (scheduledThreadExecutorService.schedule(  
            task, 30, TimeUnit.SECONDS).get() == **null**)  
            System.out.printf("The scheduledThreadExecutorService "  
                + "has succeeded at %s \n", **new** Date());  
  
        scheduledThreadExecutorService.scheduleAtFixedRate(task, 0, 5,  
            TimeUnit.SECONDS);  
  
    }  
}

If you use the submit() method version of the ExecutorService subinterface that accepts Callable<T>, then submit() returns whatever was returned from the main call() method in Callable. The following is the interface for Callable:

**package** java.util.concurrent;  
  
**public interface** Callable<V> {  
    V call() **throws** Exception;  
}

In the Java EE landscape, different approaches for solving these sorts of problems have been created, since Java EE by design restricts the handling of threads.

Quartz (a job scheduling framework) was among the first solutions to fill this thread feature gap with a solution that provided scheduling and concurrency. JCA 1.5 (or the J2EE Connector Architecture) is another specification that provides a primitive type of gateway for integration functionality and supports ad hoc concurrency. With JCA, components are notified about incoming messages and respond concurrently. JCA 1.5 provides a primitive, limited enterprise service bus—similar to integration features without nearly as much of the finesse of something like SpringSource’s Spring Integration framework.

The requirement for concurrency wasn’t lost on application server vendors, though. Many other initiatives came to the forefront. For example, in 2003, IBM and BEA jointly created the Timer and WorkManager APIs, which eventually became JSR-237 and was then merged with JSR-236 to focus on how to implement concurrency in a managed environment. The Service Data Object (SDO) specification, JSR-235, had a similar solution. In addition, open source implementations of the CommonJ API have sprung up in recent years to achieve the same solution.

The issue is that there’s no portable, standard, simple way of controlling threads and providing concurrency for components in a managed environment, similar to the case of Java SE solutions.

Spring provides a unified solution via the org.springframeworks.core.task.TaskExecutor interface. The TaskExecutor abstraction extends java.util.concurrent.Executor, which is part of Java 1.5.

In fact, the TaskExecutor interface is used quite a bit internally in the Spring Framework. For example, for Spring Quartz integration (which supports threading) and the message-driven POJO container support, there’s wide use of TaskExecutor.

**package** org.springframework.core.task;  
  
**import** java.util.concurrent.Executor;  
  
**public interface** TaskExecutor **extends** Executor {  
    void execute(Runnable task);  
}

In some places, the various solutions mirror the functionality provided by the core JDK options. In others, they’re quite unique and provide integrations with other frameworks such as with CommonJ WorkManager. These integrations usually take the form of a class that can exist in the target framework but that you can manipulate just like any other TaskExecutor abstraction.

Although there’s support for adapting an existing Java SE Executor or ExecutorService as a TaskExecutor, this isn’t so important in Spring because the base class for TaskExecutor is an Executor anyway. In this way, the TaskExecutor in Spring bridges the gap between various solutions on Java EE and Java SE.

Next, let’s see a simple example of the TaskExecutor, using the same Runnable defined previously. The client for the code is a simple Spring POJO, into which you’ve injected various instances of TaskExecutor with the sole aim of submitting Runnable.

**package** com.apress.springrecipes.executors;  
  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.context.ApplicationContext;  
**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;  
**import** org.springframework.core.task.SimpleAsyncTaskExecutor;  
**import** org.springframework.core.task.SyncTaskExecutor;  
**import** org.springframework.core.task.support.TaskExecutorAdapter;  
**import** org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;  
**import** org.springframework.stereotype.Component;  
  
**import** javax.annotation.PostConstruct;  
  
@Component  
**public class** SpringExecutorsDemo {  
  
    @Autowired  
    **private** SimpleAsyncTaskExecutor asyncTaskExecutor;  
    @Autowired  
    **private** SyncTaskExecutor syncTaskExecutor;  
    @Autowired  
    **private** TaskExecutorAdapter taskExecutorAdapter;  
    @Autowired  
    **private** ThreadPoolTaskExecutor threadPoolTaskExecutor;  
    @Autowired  
    **private** DemonstrationRunnable task;  
  
    @PostConstruct  
    **public** void submitJobs() {  
        syncTaskExecutor.execute(task);  
        taskExecutorAdapter.submit(task);  
        asyncTaskExecutor.submit(task);  
  
        **for** (int i = 0; i < 500; i++)  
            threadPoolTaskExecutor.submit(task);  
    }  
  
    **public** static void main(String[] args) {  
  
        **new** AnnotationConfigApplicationContext(ExecutorsConfiguration.class)  
            .registerShutdownHook();  
    }  
}

The application context demonstrates the creation of these various TaskExecutor implementations. Most are so simple that you could create them manually. Only in one case do you delegate to a factory bean to automatically trigger the execution , shown here:

**package** com.apress.springrecipes.executors;  
  
**import** org.springframework.context.annotation.Bean;  
**import** org.springframework.context.annotation.ComponentScan;  
**import** org.springframework.context.annotation.Configuration;  
**import** org.springframework.core.task.SimpleAsyncTaskExecutor;  
**import** org.springframework.core.task.SyncTaskExecutor;  
**import** org.springframework.core.task.support.TaskExecutorAdapter;  
**import** org.springframework.scheduling.concurrent.ScheduledExecutorFactoryBean;  
**import** org.springframework.scheduling.concurrent.ScheduledExecutorTask;  
**import** org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;  
  
**import** java.util.concurrent.Executors;  
  
@Configuration  
@ComponentScan  
**public class** ExecutorsConfiguration {  
  
    @Bean  
    **public** TaskExecutorAdapter taskExecutorAdapter() {  
        **return new** TaskExecutorAdapter(Executors.newCachedThreadPool());  
    }  
  
    @Bean  
    **public** SimpleAsyncTaskExecutor simpleAsyncTaskExecutor() {  
        **return new** SimpleAsyncTaskExecutor();  
    }  
  
    @Bean  
    **public** SyncTaskExecutor syncTaskExecutor() {  
        **return new** SyncTaskExecutor();  
    }  
   
    @Bean  
    **public** ScheduledExecutorFactoryBean scheduledExecutorFactoryBean(ScheduledExecutorTask scheduledExecutorTask) {  
        ScheduledExecutorFactoryBean scheduledExecutorFactoryBean = **new** ScheduledExecutorFactoryBean();  
        scheduledExecutorFactoryBean.setScheduledExecutorTasks(scheduledExecutorTask);  
        **return** scheduledExecutorFactoryBean;  
    }  
  
    @Bean  
    **public** ScheduledExecutorTask scheduledExecutorTask(Runnable runnable) {  
        ScheduledExecutorTask scheduledExecutorTask = **new** ScheduledExecutorTask();  
        scheduledExecutorTask.setPeriod(1000);  
        scheduledExecutorTask.setRunnable(runnable);  
        **return** scheduledExecutorTask;  
    }  
  
    @Bean  
    **public** ThreadPoolTaskExecutor threadPoolTaskExecutor() {  
        ThreadPoolTaskExecutor taskExecutor = **new** ThreadPoolTaskExecutor();  
        taskExecutor.setCorePoolSize(50);  
        taskExecutor.setMaxPoolSize(100);  
        taskExecutor.setAllowCoreThreadTimeOut(**true**);  
        taskExecutor.setWaitForTasksToCompleteOnShutdown(**true**);  
        **return** taskExecutor;  
    }  
}

The previous code shows different implementations of the TaskExecutor interface. The first bean, the TaskExecutorAdapter instance, is a simple wrapper around a java.util.concurrence.Executors instance so you can deal with it in terms of the Spring TaskExecutor interface. You use Spring here to configure an instance of an Executor and pass it in as the constructor argument.

SimpleAsyncTaskExecutor provides a new Thread for each submitted job. It does no thread pooling or reuse. Each job submitted runs asynchronously in a thread.

SyncTaskExecutor is the simplest of the implementations of TaskExecutor. Submission of a job is synchronous and tantamount to launching a Thread, running it, and then using join() to connect it immediately. It’s effectively the same as manually invoking the run() method in the calling thread, skipping threading altogether.

ScheduledExecutorFactoryBean automatically triggers jobs defined as ScheduledExecutorTask beans. You can specify a list of ScheduledExecutorTask instances to trigger multiple jobs simultaneously. A ScheduledExecutorTask instance can accept a period to space out the execution of tasks.

The last example is ThreadPoolTaskExecutor, which is a full-on thread pool implementation built on java.util.concurrent.ThreadPoolExecutor.

If you want to build applications using the CommonJ WorkManager/TimerManager support available in application servers like IBM WebSphere, you can use org.springframework.scheduling.commonj.WorkManagerTaskExecutor. This class delegates to a reference to the CommonJ Work Manager available inside of WebSphere. Usually, you’ll provide it with a JNDI reference to the appropriate resource.

In JEE 7, the javax.enterprise.concurrent package and specifically the ManagedExecutorService, was added. An instance of this ManagedExecutorService must be provided by JEE 7–compliant servers. If you want to use this mechanism with Spring TaskExecutor support, you can configure a DefaultManagedTaskExecutor, which will try to detect the default ManagedExecutorService (as mentioned by the specification), or you can explictly configure it.

The TaskExecutor support provides a powerful way to access scheduling services on an application server via a unified interface. If you’re looking for more robust (albeit much more heavyweight) support that can be deployed on any app server (e.g., Tomcat and Jetty), you might consider Spring’s Quartz support.