87_源码剖析：深入理解 Spring Task 底层原理

1、开篇

上节课作为本周的第一节课，专注于评价订单功能，我们首先描述了评价订单功能位于整个订单流程的位置，并且回顾了其具体的功能实现。然后通过业务流程图的方式展现整个订单流程以及评价订单功能模块。本节课引入 Spring Task 的源码剖析，在深入理解 Spring Task 底层原理的基础上再开始代码的编写。今天课程的内容包括以下几个部分：

- Spring Task 基本描述
- Spring Task 源代码分析

2、Spring Task 基本描述

Spring task 主要解决的事定时任务调度和执行的问题，其中有两个概念一个是任务，也就是需要执行的内容，另一个是调度，说的是任务如何执行。

如图 1 所示，task:scheduler 定义的是调度器，主要包括调度器线程池的大小。对应的 bean 定义的是具体执行任务的 bean，也就是要处理的业务逻辑。其中指定了 scheduler 而且执行了 task 中哪个方法（print）作为执行的任务，同时指定了 cron 表达式来说明任务执行方式（频率）。Cron 表达式的知识这里不展开说明。从配置文件来看 task 就是要执行的任务，scheduled-tasks 就描述了这个任务是如何执行的。

```
<task:scheduler id="scheduler" pool-size="3" />
<bean id="task" class="task.Task"/>
<task:scheduled-tasks scheduler="scheduler">
    <task:scheduled ref="task" method="print" cron="0/5 * * * * ?"/>
</task:scheduled-tasks>
```

图 1

如图 2 所示，Task 类内部的 print 方法就是简单打印文本，这也是我们所说的执行的具体业务逻辑，开发者只需要将主要精力放到这里就可以了，其他工作都交给 Spring task 来处理。

```
public class Task {
    public void print() {
        System.out.println("print执行");
    }
}
```

图 2

3、Spring Task 源代码分析

上面讲解了 Spring Task 的基本概念，在配置中讲到了 task:scheduler 和 task:scheduled-tasks 节点。这里将它们对应的源代码进行展开说明。

Scheduler 作为调度器需要对其基本线程池信息进行初始化，其中对于 task:scheduler 注释的解析由 TaskNamesapceHandler 完成。如图 3 所示，红框的部分就是对 Scheduler 注释的初始化。

```
@Override
public void init() {
    this.registerBeanDefinitionParser("annotation-driven", new AnnotationDrivenBeanDefinitionParser());
    this.registerBeanDefinitionParser("executor", new ExecutorBeanDefinitionParser());
    this.registerBeanDefinitionParser("scheduled-tasks", new ScheduledTasksBeanDefinitionParser());
    this.registerBeanDefinitionParser("scheduler", new SchedulerBeanDefinitionParser());
}
```

图 3

如图 4 所示，由于 SchedulerBeanDefinitionParser 是 AbstractSingleBeanDefinitionParser 的子类，所以 Spring 将 task:scheduler 标签解析为一个 BeanDefinition。其 beanClass 为 org.springframework.scheduling.concurrent.ThreadPoolTaskScheduler。

```
@Override
protected String getBeanClassName(Element element) {
    return "org.springframework.scheduling.concurrent.ThreadPoolTaskScheduler";
}

@Override
protected void doParse(Element element, BeanDefinitionBuilder builder) {
    String poolSize = element.getAttribute("pool-size");
    if (StringUtils.hasText(poolSize)) {
        builder.addPropertyValue("poolSize", poolSize);
    }
}
```

图 4

task:scheduled-tasks 和 task:scheduler 一样也有一个初始化和解析的过程，过程比较类似这里不赘述，就将解析的结果，也就是最终生成的 BeanDefinition 的结构展示给大家。

如 5 所示，最左边的 taskScheduler 属性指向 task:scheduler 标签，从左到右依次包含了 cronTasklist、fixedDelayTasklist、fixedRateTaskslist、triggerTasklist。

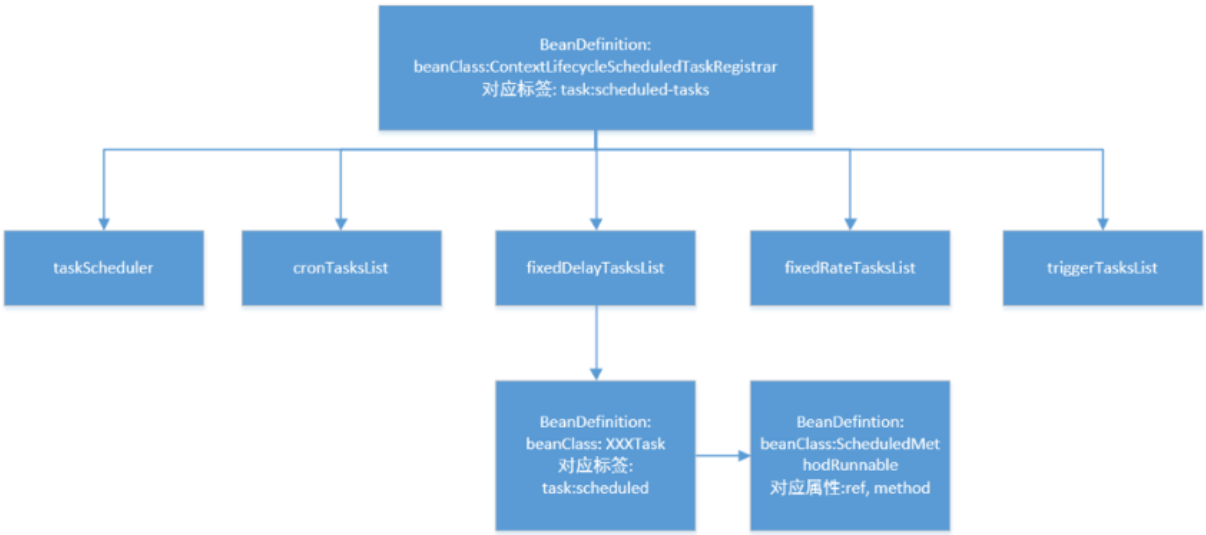

图 5

下面再来说说这些 task 是如何定义的。如图 6 所示，任务的类型是由 cron, fixed-delay, fixed-rate, trigger 四个属性决定的，fixed-delay 和 fixed-rate 为 IntervalTask。

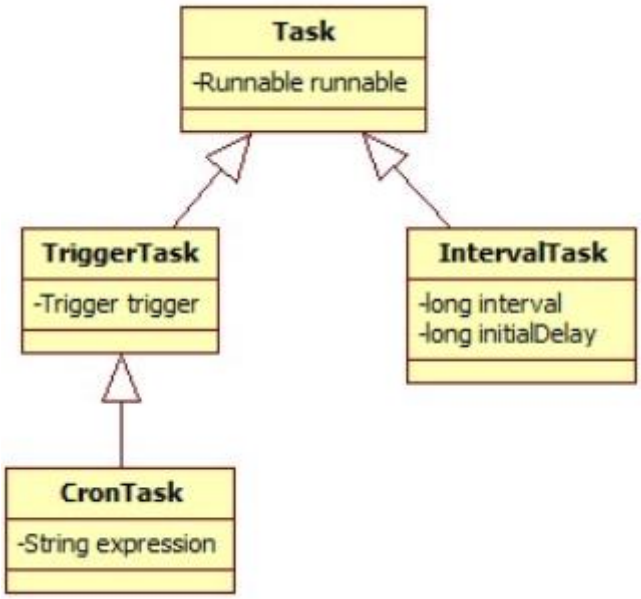

图 6

有了任务就来看看调度执行，如图 7 所示，其入口就是 ContextLifecycleScheduledTaskRegistrar，它继承了 ScheduledTaskRegistrar 类，实现了 SmartInitializingSingleton 接口。

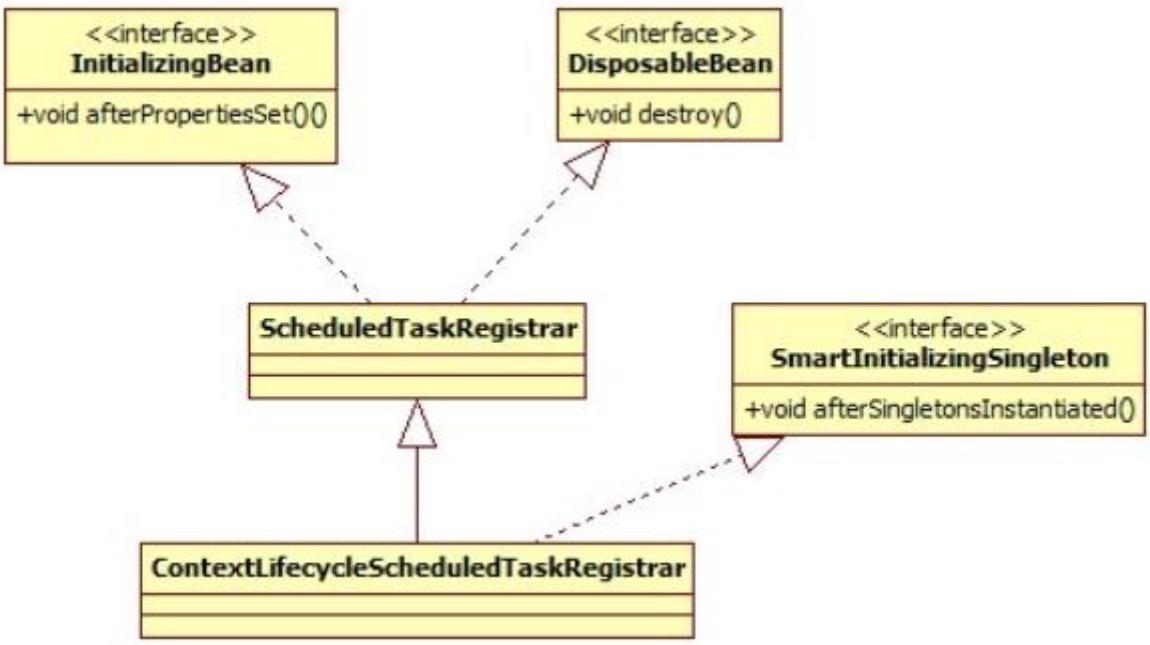

图 7

如图 8 所示，ContextLifecycleScheduledTaskRegistrar 中实现了 afterSingletonsInstantiated 方法，而这个方法会调用 scheduleTasks 方法。在这个方法中会对 schduler 进行初始化的操作，并且添加需要处理的 task。

```
@Override
public void afterSingletonsInstantiated() {
    scheduleTasks();
}
```

ScheduledTaskRegistrar.scheduleTasks:

```
protected void scheduleTasks() {
     // shcheduler初始化
    if (this.taskScheduler == null) {
        this.localExecutor = Executors.newSingleThreadScheduledExecutor();
        this.taskScheduler = new ConcurrentTaskScheduler(this.localExecutor);
    }
    if (this.triggerTasks != null) {
        for (TriggerTask task : this.triggerTasks) {
            addScheduledTask(scheduleTriggerTask(task));
        }
    }
    if (this.cronTasks != null) {
        for (CronTask task : this.cronTasks) {
            addScheduledTask(scheduleCronTask(task));
        }
    }
    if (this.fixedRateTasks != null) {
        for (IntervalTask task : this.fixedRateTasks) {
            addScheduledTask(scheduleFixedRateTask(task));
        }
    }
    if (this.fixedDelayTasks != null) {
        for (IntervalTask task : this.fixedDelayTasks) {
            addScheduledTask(scheduleFixedDelayTask(task));
        }
    }
}
```

图 8

还是接着上面代码，接着说 scheduler 的初始化，它的初始化是借助 ConcurrentTaskScheduler 完成的。

如图 9 所示，ConcurrentTaskScheduler 实现了 TaskScheduler 接口，继承了 ConcurrentTaskExecutor 类。

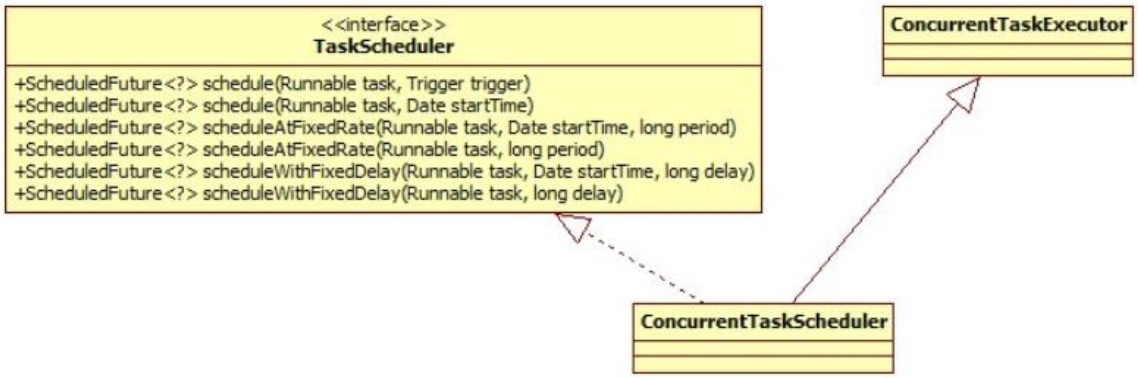

图 9

有了 task 和 scheduler 然后就来看任务调度了，这里以 CronTask 的调度为例为大家讲解。如图 10 所示，在 ScheduledTaskRegistrar 中执行的 scheduleCronTask 方法，会使用 taskScheduler 中的 schedule 创建 task，其中会传入 trigger 作为参数。

```
public ScheduledTask scheduleCronTask(CronTask task) {
    ScheduledTask scheduledTask = this.unresolvedTasks.remove(task);
    if (this.taskScheduler != null) {
        scheduledTask.future = this.taskScheduler.schedule(task.getRunnable(), task.getTrigger());
    }
    return (newTask ? scheduledTask : null);
}
```

图 10

如图 11 所示，cron 是通过 triiger 实现的，CronTrigger 类是实现了 Trigger 接口。

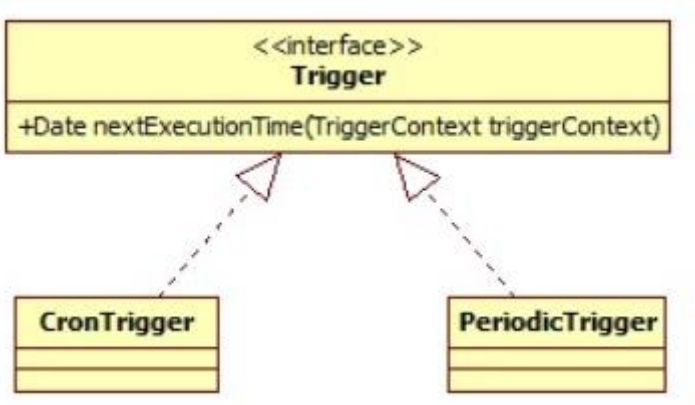

图 11

如图 12 所示，CronTrigger 构造器使用了 CronSequenceGenerator。

```
public CronTrigger(String expression) {
    this.sequenceGenerator = new CronSequenceGenerator(expression);
}
```

图 12

如图 13 所示 CronSequenceGenerator 中定义了表达和时区的信息。

```
public CronSequenceGenerator(String expression) {
    this(expression, TimeZone.getDefault());
}

public CronSequenceGenerator(String expression, TimeZone timeZone) {
    this.expression = expression;
    this.timeZone = timeZone;
    parse(expression);
}
```

图 13

再接着看 ConcurrentTaskScheduler 中的 schedule 方法。如图 14 所示，方法体中使用了 ReschedulingRunnable 实现调度。

```
@Override
public ScheduledFuture<?> schedule(Runnable task, Trigger trigger) {
    ErrorHandler errorHandler = (this.errorHandler != null ? this.errorHandler :
                                        TaskUtils.getDefaultErrorHandler(true));
    return new ReschedulingRunnable(task, trigger, this.scheduledExecutor, errorHandler).schedule();
}
```

图 14

如图 15 所示，ReschedulingRunnable 实现了 ScheduledFuture 接口。

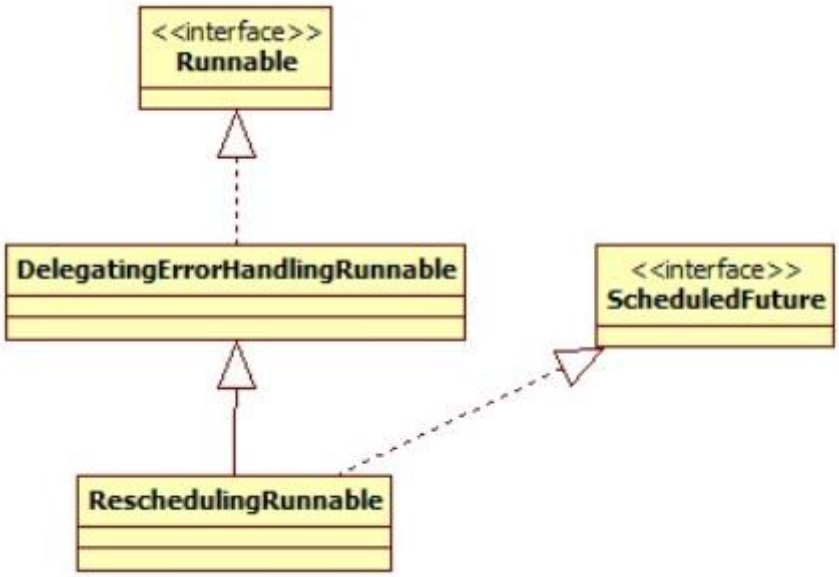

图 15

ReschedulingRunnable 中的 schdule 方法挑起了调度的大梁

```
public ScheduledFuture<?> schedule() {
    synchronized (this.triggerContextMonitor) {
        this.scheduledExecutionTime = this.trigger.nextExecutionTime(this.triggerContext);
        if (this.scheduledExecutionTime == null) {
            return null;
        }
        long initialDelay = this.scheduledExecutionTime.getTime() - System.currentTimeMillis();
        this.currentFuture = this.executor.schedule(this, initialDelay, TimeUnit.MILLISECONDS);
        return this;
    }
}
```

图 16

从图 15 的类图可以看出，ReschedulingRunnable 本身实现了 Runnable 接口，其 run 方法如图 17 所示，方法体中会更新 triggerContext 的上下文信息，如果 currentFuture 为取消状态会再次调用 schedule 方法。

```
@Override
public void run() {
    Date actualExecutionTime = new Date();
    super.run();
    Date completionTime = new Date();
    synchronized (this.triggerContextMonitor) {
        this.triggerContext.update(this.scheduledExecutionTime, actualExecutionTime, completionTime);
        if (!this.currentFuture.isCancelled()) {
             //下次调用
            schedule();
        }
    }
}
```

图 17

3、总结

本节课主要对 Spring Task 的源码进行解析让大家对其运行原理有深刻的认识。首先我们通过配置的例子了解 Spring Task 由 任务和调度组成，在 XML 配置中由 task、task:scheduler 和 task:scheduled-tasks 三个节点完成功能。接下来将注意力集中到 scheduler 和 scheduled-tasks 的源代码解释上，从初始化、解析到执行逐一解释代码运行流程。

下节课会开始代码实战的部分，我们会基于 Spring 任务调度实现教师教学和评分信息的更新。下期见，拜拜。