Activiti工作流学习笔记(四)——工作流引擎中责任链模式的建立与应用原理

2021-04-12阅读 4020

原创/朱季谦

本文需要一定责任链模式的基础与Activiti工作流知识，主要分成三部分讲解：

一、简单理解责任链模式概念 二、Activiti工作流里责任链模式的建立 三、Activiti工作流里责任链模式的应用

一、简单理解责任链模式概念

网上关于责任链模式的介绍很多，菜鸟教程上是这样说的：责任链模式（Chain of Responsibility Pattern）为请求创建了一个接收者对象的链。在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

这个概念术语比较抽象。

我曾经在 深入理解Spring Security授权机制原理 一文中提到Spring Security在授权过程中有使用到过滤器的概念，**过滤器链就像一条铁链，中间的每个过滤器都包含对另一个过滤器的引用，从而把相关的过滤器链接起来，就像一条链的样子。这时请求线程如蚂蚁一样，会沿着这条链一直爬过去-----即，通过各过滤器调用另一个过滤器引用方法chain.doFilter(request，response)，实现一层嵌套一层地将请求传递下去，当该请求传递到能被处理的过滤器时，就会被处理，处理完成后转发返回。通过过滤器链，可实现在不同的过滤器当中对请求request做拦截增加，且过滤器之间彼此互不干扰。**

整个流程大致如下：

这个过滤器链的概念，其实就是**责任链设计模式**在Spring Security中的体现。

摘录一段网上关于职责链模式介绍，其主要包含以下角色：

1. 抽象处理者（Handler）角色：定义一个处理请求的接口，包含抽象处理方法和一个后继连接。
2. 具体处理者（Concrete Handler）角色：实现抽象处理者的处理方法，判断能否处理本次请求，如果可以处理请求则处理，否则将该请求转给它的后继者。
3. 客户类（Client）角色：创建处理链，并向链头的具体处理者对象提交请求，它不关心处理细节和请求的传递过程。

二、Activiti工作流里责任链模式的创建

最近在研究Activiti工作流框架，发现其所有实现都是采用命令模式实现，而命令模式当中的Invoker角色又是采用拦截器链式模式，即类似上面提到的过滤器链，即设计模式里的责任链模式。

这里的Activiti工作流版本是6.0。

CommandInterceptor是一个拦截器接口，包含三个方法：

- setNext()方法是在初始化时，设置每个拦截器对象中包含了下一个拦截器对象，最后形成一条拦截器链；
- getNext()可在每个拦截器对象中调用下一个拦截器对象；
- execute()是每个拦截器对请求的处理。若在上一个拦截器链式里不能处理该请求话，就会通过next.execute(CommandConfig var1, Command var2)将请求传递到下一个拦截器做处理，类似

上面过滤器里调用下一个过滤器的chain.doFilter(request，response)方法，将请求进行传递；

```
public interface CommandInterceptor {
    <T> T execute(CommandConfig var1, Command<T> var2);

    CommandInterceptor getNext();

    void setNext(CommandInterceptor var1);
}
```

抽象类AbstractCommandInterceptor实现了CommandInterceptor拦截器接口，在责任链模式当中充当抽象处理者（Handler）角色。该类最主要的属性是 protected CommandInterceptor next，在同一包下，直接通过next即可调用下一个拦截器对象。

```
public abstract class AbstractCommandInterceptor implements CommandInterceptor {
    protected CommandInterceptor next;
    public AbstractCommandInterceptor() {
    }
    public CommandInterceptor getNext() {
        return this.next;
    }
    public void setNext(CommandInterceptor next) {
        this.next = next;
    }
}
```

接下来，将会分析拦截器链是如何初始化与工作的。

SpringBoot集成Activiti配置如下：

```
  @Configuration
  public class SpringBootActivitiConfig {
      @Bean
      public ProcessEngine processEngine() {
          ProcessEngineConfiguration pro =
ProcessEngineConfiguration.createStandaloneProcessEngineConfiguration();
          pro.setJdbcDriver("com.mysql.jdbc.Driver");
          pro.setJdbcUrl("xxxx");
          pro.setJdbcUsername("xxxx");
          pro.setJdbcPassword("xxx");
          //避免发布的图片和xml中文出现乱码
          pro.setActivityFontName("宋体");
          pro.setLabelFontName("宋体");
          pro.setAnnotationFontName("宋体");
          //数据库更更新策略

 pro.setDatabaseSchemaUpdate(ProcessEngineConfiguration.DB_SCHEMA_UPDATE_TRUE);
          return pro.buildProcessEngine();
      }
  }
```

这时，启动项目后，pro.buildProcessEngine()这行代码会初始化Activiti框架，进入里面，会发现它有三种实现，默认是第二种，即ProcessEngineConfigurationImpl。

点进去，Activiti框架具体构建buildProcessEngine方法如下，其中 this.init()的作用是环境初始化，包括配置设置、JDBC连接、bean装载等的：

```
public ProcessEngine buildProcessEngine() {
    this.init();
    ProcessEngineImpl processEngine = new ProcessEngineImpl(this);
    if (this.isActiviti5CompatibilityEnabled &&
this.activiti5CompatibilityHandler != null) {

 Context.setProcessEngineConfiguration(processEngine.getProcessEngineConfigurati
on());
        this.activiti5CompatibilityHandler.getRawProcessEngine();
    }

    this.postProcessEngineInitialisation();
    return processEngine;
}
```

在this.init()方法里，涉及到责任链模式初始化的方法是this.initCommandExecutors()，里面详情如下：

```
public void initCommandExecutors() {
    this.initDefaultCommandConfig();
    this.initSchemaCommandConfig();
    //初始化命令调用器
    this.initCommandInvoker();
    //List存放进涉及到的拦截器
    this.initCommandInterceptors();
    //初始化命令执行器
    this.initCommandExecutor();
}
```

这里只需要关注最后三个方法——

该方法有两个作用，一个是生成Interceptor拦截器链，一个是创建命令执行器commandExecutor。

```
public void initCommandExecutor() {
    if (this.commandExecutor == null) {
        CommandInterceptor first =
this.initInterceptorChain(this.commandInterceptors);
        this.commandExecutor = new
CommandExecutorImpl(this.getDefaultCommandConfig(), first);
    }
}
```

this.initInterceptorChain(this.commandInterceptors)是将集合里的拦截器初始化生成一条拦截器链，先循环获取List集合里的拦截器对象chain.get(i)，然后通过setNext()方法在该拦截器对象chain.get(i)里设置下一个拦截器引用，这样，就可实现责任链里所谓每个接收者都包含对另一个接收者的引用的功能。

```
public CommandInterceptor initInterceptorChain(List<CommandInterceptor> chain) {
    if (chain != null && !chain.isEmpty()) {
        for(int i = 0; i < chain.size() - 1; ++i) {

((CommandInterceptor)chain.get(i)).setNext((CommandInterceptor)chain.get(i +
1));
        }
        return (CommandInterceptor)chain.get(0);
    } else {
        throw new ActivitiException("invalid command interceptor chain
configuration: " + chain);
    }
}
```

那么，这条拦截器链当中，都有哪些拦截器呢?

直接debug到这里，可以看到，总共有4个拦截器对象，按照顺序排，包括LogInterceptor，CommandContextInterceptor，TransactionContextInterceptor，CommandInvoker（在命令模式里，该类相当Invoker角色）。**这四个拦截器对象在责任链模式当中充当了具体处理者（Concrete Handler）角色。**

责任链模式里剩余客户类（Client）角色应该是命令执行器this.commandExecutor。

因此，工作流引擎当中的责任链模式结构图如下：

组成一条拦截器链如下图所示——

生成拦截器链后，会返回一个(CommandInterceptor)chain.get(0)，即拦截器LogInterceptor，为什么只返回第一个拦截器呢，这是一个很巧妙的地方，因为该拦截器里已经一层一层地嵌套进其他拦截器了，因此，只需要返回第一个拦截器，赋值给first即可。

接下来，就会创建命令执行器——

```
this.commandExecutor = new CommandExecutorImpl(this.getDefaultCommandConfig(),
first);
```

这个命令执行器是整个引擎的底层灵魂，通过它，可以实现责任链模式与命令模式——

拦截器链初始化介绍完成后，接下来开始介绍拦截器链在引擎里的应用方式。

三、Activiti工作流里责任链模式的应用

Activiti引擎的各操作方法其底层基本都是以命令模式来实现的，即调用上面创建的命令执行器this.commandExecutor的execute方法来实现的，例如自动生成28张数据库表的方法，就是通过命令模式去做具体实现的——

```
this.commandExecutor.execute(processEngineConfiguration.getSchemaCommandConfig()
, new SchemaOperationsProcessEngineBuild());
```

进入到commandExecutor方法里，会发现前边new CommandExecutorImpl(this.getDefaultCommandConfig(), first)建立命令执行器时，已将配置对象和嵌套其他拦截器的LogInterceptor拦截器对象，通过构造器CommandExecutorImpl(CommandConfig defaultConfig, CommandInterceptor first)生成对象时，传参赋值给了相应的对象属性，其中first引用指向LogInterceptor，即拦截器链上的第一个拦截器——

```
public class CommandExecutorImpl implements CommandExecutor {
    protected CommandConfig defaultConfig;
    protected CommandInterceptor first;

    public CommandExecutorImpl(CommandConfig defaultConfig, CommandInterceptor
first) {
        this.defaultConfig = defaultConfig;
        this.first = first;
    }

    public CommandInterceptor getFirst() {
        return this.first;
    }

    public void setFirst(CommandInterceptor commandInterceptor) {
        this.first = commandInterceptor;
    }

    public CommandConfig getDefaultConfig() {
        return this.defaultConfig;
    }

    public <T> T execute(Command<T> command) {
        return this.execute(this.defaultConfig, command);
    }

    public <T> T execute(CommandConfig config, Command<T> command) {
        return this.first.execute(config, command);
    }
}
```

当引擎执行this.commandExecutor.execute(xxx，xxx))类似方法时，其实是执行了this.first.execute(config, command)方法，这里的this.first在构建命令执行器时是通过LogInterceptor传进来的，因此，执行代码其实是调用了LogInterceptor内部的execute()方法，也就是说，开始拦截器链上的第一个LogInterceptor拦截器传递方法execute()请求——

进入到拦截器链上的第一个拦截器LogInterceptor。

根据其内部代码可以看出，这是一个跟日志有关的拦截器，内部并没有多少增强功能，只是做了一个判断是否需要debug日志打印。若需要，则进行debug打印，若不需要，直接进入到 if (!log.isDebugEnabled()) 为true的作用域内部，进而执行this.next.execute(config, command)用以将请求传递给下一个拦截器做处理。

```
public class LogInterceptor extends AbstractCommandInterceptor {
    private static Logger log = LoggerFactory.getLogger(LogInterceptor.class);
    public LogInterceptor() {
    }
    public <T> T execute(CommandConfig config, Command<T> command) {
```

```
        if (!log.isDebugEnabled()) {
            return this.next.execute(config, command);
        } else {
            log.debug("\n");
            log.debug("--- starting {} ----------------------------------------
---------------", command.getClass().getSimpleName());
            Object var3;
            try {
                var3 = this.next.execute(config, command);
            } finally {
                log.debug("--- {} finished ------------------------------------
------------------", command.getClass().getSimpleName());
                log.debug("\n");
            }

            return var3;
        }
    }
}
```

这里有一个小地方值得稍微打断说下，就这个 if (!log.isDebugEnabled())判断。众生周知，若集成第三方日志插件如logback之类，若其配置里去除debug的打印，即时代码里 存在log.debug("xxxxx")也不会打印到控制台，那么，这里增加一个判断 if (!log.isDebugEnabled())是否多次一举呢?

事实上，这里并非多此一举，增加这个判断，是可以提升代码执行效率的。因为log.debug("xxxxx")里的字符串拼接早于log.debug("xxxxx")方法执行的，也就是说，即使该log.debug("xxxxx")不会打印，但其内部的字符串仍然会进行拼接，而拼接，是需要时间的，虽然很细微，但同样属于影响性能范畴内的。因此，增加一个if判断，若无需要打印debug日志时，那么就无需让其内部的字符串进行自动拼接。

这是一个很小的知识点，但面试过程中其实是有可能会遇到这类与日志相关的面试题的。

接下来，让我们继续回到拦截器链的传递上来。

LogInterceptor拦截器调用this.next.execute(config, command)，意味着将请求传递到下一个拦截器上进行处理，根据前边分析，可知下一个拦截器是CommandContextInterceptor，根据代码大概可知，这个拦截器内主要是获取上下文配置对象和信息相关的，这些都是在工作流引擎初始化时生成的，它们被保存在Stack栈里，具体都保存了哪些信息暂不展开分析——

```
public class CommandContextInterceptor extends AbstractCommandInterceptor {
    ......
public <T> T execute(CommandConfig config, Command<T> command) {
    CommandContext context = Context.getCommandContext();
    boolean contextReused = false;
    if (config.isContextReusePossible() && context != null &&
context.getException() == null) {
        contextReused = true;
        context.setReused(true);
    } else {
        context = this.commandContextFactory.createCommandContext(command);
    }

    try {
        Context.setCommandContext(context);
        Context.setProcessEngineConfiguration(this.processEngineConfiguration);
        if (this.processEngineConfiguration.getActiviti5CompatibilityHandler()
!= null) {
```

```
 Context.setActiviti5CompatibilityHandler(this.processEngineConfiguration.getActiviti5CompatibilityHandler());
        }
        //继续将命令请求传递到下一个拦截器
        Object var5 = this.next.execute(config, command);
        return var5;
    } catch (Exception var31) {
        context.exception(var31);
    } finally {
     ......
    }

    return null;
}
}
```

CommandContextInterceptor拦截器没有对命令请求做处理，它继续将请求传递到下一个拦截器TransactionContextInterceptor，根据名字就大概可以猜到，这个拦截器主要是增加与事务有关的功能——

```
public <T> T execute(CommandConfig config, Command<T> command) {
    CommandContext commandContext = Context.getCommandContext();
    boolean isReused = commandContext.isReused();
    Object var9;
    try {
        if (this.transactionContextFactory != null && !isReused) {
            TransactionContext transactionContext =
this.transactionContextFactory.openTransactionContext(commandContext);
            Context.setTransactionContext(transactionContext);
            commandContext.addCloseListener(new
TransactionCommandContextCloseListener(transactionContext));
        }
        var9 = this.next.execute(config, command);
    } finally {
    ......
    }
    return var9;
}
```

TransactionContextInterceptor拦截器同样没有对命令请求做处理，而是继续传递到下一个拦截器，也就是最后一个拦截器CommandInvoker，根据名字可以大概得知，这是一个与命令请求有关的拦截器，传递过来的请求将会在这个拦截器里处理——

```
public class CommandInvoker extends AbstractCommandInterceptor {
    ......
    public <T> T execute(CommandConfig config, final Command<T> command) {
        final CommandContext commandContext = Context.getCommandContext();
        commandContext.getAgenda().planOperation(new Runnable() {
            public void run() {
                commandContext.setResult(command.execute(commandContext));
            }
        });
        this.executeOperations(commandContext);
        if (commandContext.hasInvolvedExecutions()) {
            Context.getAgenda().planExecuteInactiveBehaviorsOperation();
```

```
            this.executeOperations(commandContext);
        }
        return commandContext.getResult();
    }
}
```

进入到其内部，可以发现，这里没有再继续调用this.next.execute(config, command)这样的请求进行传递，而是直接执行command.execute(commandContext)，然后将返回值进行返回，其中，command是请求参数当中的第二个参数，让我们回过头看下该请求案例最开始的调用——

```
this.commandExecutor.execute(processEngineConfiguration.getSchemaCommandConfig()
, new SchemaOperationsProcessEngineBuild());
```

这里的第二个参数是new SchemaOperationsProcessEngineBuild()，不妨进入到SchemaOperationsProcessEngineBuild类中，是吧，其内部同样有一个execute方法——

```
public final class SchemaOperationsProcessEngineBuild implements Command<Object>
{
    public SchemaOperationsProcessEngineBuild() {
    }

    public Object execute(CommandContext commandContext) {
        DbSqlSession dbSqlSession = commandContext.getDbSqlSession();
        if (dbSqlSession != null) {
            dbSqlSession.performSchemaOperationsProcessEngineBuild();
        }

        return null;
    }
}
```

可见，CommandInvoker拦截器内部执行command.execute(commandContext)，就相当于执行了new SchemaOperationsProcessEngineBuild().execute(commandContext)，也就是——

```
 public Object execute(CommandContext commandContext) {
        DbSqlSession dbSqlSession = commandContext.getDbSqlSession();
        if (dbSqlSession != null) {
            dbSqlSession.performSchemaOperationsProcessEngineBuild();
        }
        return null;
    }
```

这是一种命令模式的实现。

本文主要是分析责任链模式在Activiti框架中的实践，故暂不展开分析框架中的其他设计模式，有兴趣的童鞋可以自行深入研究，在Activiti框架当中，其操作功能底层基本都是以命令模式来实现的。

至此，就大概分析完了责任链模式在Activiti框架的创建和应用，学习完这块内容，我对责任链模式有了更好理解，相对于看网上那些简单以小例子来介绍设计模式的方法，我更喜欢去深入框架当中学习其设计模式，这更能让我明白，这种设计模式在什么场景下适合应用，同时，能潜移默化地影响到我，让我在设计系统架构时，能明白各设计模式的落地场景具体都是怎样的。