巧妙利用 SpringBoot 应用责任连模式，让编程事半功倍！

原创 鸭血粉丝Tang Java极客技术 2022-10-08 07:30 发表于湖北

每天早上七点三十推送干货

一、什么是责任链模式？

责任链模式（Chain of Responsibility Pattern），顾名思义，为请求者和接受者之间创建一条对象处理链路，避免请求发送者与接收者耦合在一起！

责任链模式，是一种实用性非常强的设计模式，比较典型的应用场景有：

- Apache Tomcat 对 Encoding 编码处理的处理
- SpringBoot 里面的拦截器、过滤器链
- netty 中的处理链
- 支付风控的机制
- 日志处理级别

尤其是当程序的处理流程很长的时候，采用责任链设计模式，不仅实现优雅，而且易复用可扩展！

今天我们就一起来了解一下在 SpringBoot 中如何应用责任链模式！

二、代码实践

在 SpringBoot 中，责任链模式的实践方式有好几种，今天我们主要抽三种实践方式给大家介绍。

我们以某下单流程为例，将其切成多个独立检查逻辑，可能会经过的数据验证处理流程如下：

采用责任链设计模式来编程，代码实践如下！

2.1、方式一

首先，我们定义一个简易版的下单对象 `OrderContext` 。

```
public class OrderContext {

    /**
     * 请求唯一序列ID
     */
    private String seqId;

    /**
     * 用户ID
     */
    private String userId;
```

```
    /**
     * 产品skuId
     */
    private Long skuId;

    /**
     * 下单数量
     */
    private Integer amount;

    /**
     * 用户收货地址ID
     */
    private String userAddressId;

 //..set、get
}
```

然后，我们定义一个数据处理接口 `OrderHandleIntercept` ，用于标准化执行！

```
public interface OrderHandleIntercept {

    /**
     * 指定执行顺序
     * @return
     */
    int sort();

    /**
     * 对参数进行处理
     * @param context
     * @return
     */
    OrderAddContext handle(OrderAddContext context);
}
```

接着，我们分别创建三个不同的接口实现类，并指定执行顺序，内容如下：

- `RepeatOrderHandleInterceptService` ：用于重复下单的逻辑验证
- `ValidOrderHandleInterceptService` ：用于验证请求参数是否合法
- `BankOrderHandleInterceptService` ：用于检查客户账户余额是否充足

```
@Component
public class RepeatOrderHandleInterceptService implements OrderHandleIntercept {

    @Override
    public int sort() {
        //用于重复下单的逻辑验证，在执行顺序为1
        return 1;
    }

    @Override
    public OrderAddContext handle(OrderAddContext context) {
        System.out.println("通过seqId，检查客户是否重复下单");
        return context;
    }
}
```

```
@Component
public class ValidOrderHandleInterceptService implements OrderHandleIntercept {


    @Override
    public int sort() {
        //用于验证请求参数是否合法，执行顺序为2
        return 2;
    }

    @Override
    public OrderAddContext handle(OrderAddContext context) {
        System.out.println("检查请求参数，是否合法，并且获取客户的银行账户");
        return context;
    }
}
```

```
@Component
```

```
public class BankOrderHandleInterceptService implements OrderHandleIntercept {


    @Override
    public int sort() {
        //用于检查客户账户余额是否充足，在执行顺序为3
        return 3;
    }

    @Override
    public OrderAddContext handle(OrderAddContext context) {
        System.out.println("检查银行账户是否合法，调用银行系统检查银行账户余额是否满足下单金额");
        return context;
    }
}
```

再然后，我们还需要创建一个订单数据验证管理器 `OrderHandleChainService` ，用于管理这些实现类。

```
@Component
public class OrderHandleChainService implements ApplicationContextAware {

    private List<OrderHandleIntercept> handleList = new ArrayList<>();


    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansE
        //获取指定的接口实现类，并按照sort进行排序，放入List中
        Map<String, OrderHandleIntercept> serviceMap = applicationContext.getBeansOfType(O
        handleList = serviceMap.values().stream()
                .sorted(Comparator.comparing(OrderHandleIntercept::sort))
                .collect(Collectors.toList());
    }

    /**
     * 执行处理
     * @param context
     * @return
     */
    public OrderAddContext execute(OrderAddContext context){
        for (OrderHandleIntercept handleIntercept : handleList) {
```

```
            context =handleIntercept.handle(context);
        }
        return context;
    }
}
```

最后，我们编写单元测试来看看效果如何！

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class CalculatorServiceTest {

    @Autowired
    private OrderHandleChainService orderHandleChainService;

    @Test
    public void test(){
        orderHandleChainService.execute(new OrderAddContext());
    }
}
```

执行结果如下：

```
通过seqId，检查客户是否重复下单
检查请求参数，是否合法，并且获取客户的银行账户
检查银行账户是否合法，调用银行系统检查银行账户余额是否满足下单金额
```

如果需要继续加验证流程或者处理流程，只需要重新实现 `OrderHandleIntercept` 接口就行，其他的代码无需改动！

当然，有的同学可能觉得这种方法用的不习惯，不喜欢通过 `sort()` 来指定顺序，也可以通过如下方式进行手动 `add` 排序。

```
@Component
public class OrderHandleChainService {

    private List<OrderHandleIntercept> handleList = new ArrayList<>();
```

```
    private List<OrderHandleIntercept> handleList = new ArrayList<>();

    @Autowired
    private ValidOrderHandleInterceptService validOrderHandleInterceptService;

    @Autowired
    private RepeatOrderHandleInterceptService repeatOrderHandleInterceptService;

    @Autowired
    private BankOrderHandleInterceptService bankOrderHandleInterceptService;

    @PostConstruct
    public void init(){
     //依次手动add对象
        handleList.add(repeatOrderHandleInterceptService);
        handleList.add(validOrderHandleInterceptService);
        handleList.add(bankOrderHandleInterceptService);
    }

    /**
     * 执行处理
     * @param context
     * @return
     */
    public OrderAddContext execute(OrderAddContext context){
        for (OrderHandleIntercept handleIntercept : handleList) {
            context =handleIntercept.handle(context);
        }
        return context;
    }
}
```

2.2、方式二

第二种实现方式，就更简单了，我们通过注解 `@Order` 来指定排序，代替手动方法排序 `sort()` ，操作方式如下：

```
/**
 * 指定注入顺序为1
 *
 */
```

```
@Order(1)
@Component
public class RepeatOrderHandleInterceptService implements OrderHandleIntercept {


    //...省略
}
```

```
/**
 * 指定注入顺序为2
 *
 */
@Order(2)
@Component
public class ValidOrderHandleInterceptService implements OrderHandleIntercept {


    //...省略
}
```

```
/**
 * 指定注入顺序为3
 *
 */
@Order(3)
@Component
public class BankOrderHandleInterceptService implements OrderHandleIntercept {


    //...省略
}
```

```
@Component
public class OrderHandleChainService  {

 @Autowired
    private List<OrderHandleIntercept> handleList;
```

```
    /**
     * 执行处理
     * @param context
     * @return
     */
    public OrderAddContext execute(OrderAddContext context){
        for (OrderHandleIntercept handleIntercept : handleList) {
            context =handleIntercept.handle(context);
        }
        return context;
    }
}
```

运行单元测试，你会发现结果与上面运行的结果一致，原因 `Spring` 的 `ioc` 容器，支持通过 `Map` 或者 `List` 来直接注入对象，省去自己排序。

2.3、方式三

通过定义抽象类来实现责任链设计模式，还是以上面的案例为例，我们需要先定义一个抽象类，比如 `AbstractOrderHandle` 。

```
public abstract class AbstractOrderHandle {

    /**
     * 责任链，下一个链接节点
     */
    private AbstractOrderHandle next;

    /**
     * 执行入口
     * @param context
     * @return
     */
    public OrderAddContext execute(OrderAddContext context){
        context = handle(context);
        // 判断是否还有下个责任链节点，没有的话，说明已经是最后一个节点
        if(getNext() != null){
            getNext().execute(context);
```

```
        }
        return context;
    }

    /**
     * 对参数进行处理
     * @param context
     * @return
     */
    public abstract OrderAddContext handle(OrderAddContext context);


    public AbstractOrderHandle getNext() {
        return next;
    }

    public void setNext(AbstractOrderHandle next) {
        this.next = next;
    }
}
```

然后，分别创建三个处理类，并排好序号。

```
@Order(1)
@Component
public class RepeatOrderHandle extends AbstractOrderHandle {

    @Override
    public OrderAddContext handle(OrderAddContext context) {
        System.out.println("通过seqId，检查客户是否重复下单");
        return context;
    }
}
```

```
@Order(2)
@Component
public class ValidOrderHandle extends AbstractOrderHandle {

    @Override
    public OrderAddContext handle(OrderAddContext context) {
```

```
        System.out.println("检查请求参数，是否合法，并且获取客户的银行账户");
        return context;
    }
}
```

```
@Order(3)
@Component
public class BankOrderHandle extends AbstractOrderHandle {

    @Override
    public OrderAddContext handle(OrderAddContext context) {
        System.out.println("检查银行账户是否合法，调用银行系统检查银行账户余额是否满足下单金额");
        return context;
    }
}
```

接着，创建一个责任链管理器，比如 `OrderHandleManager` 。

```
@Component
public class OrderHandleManager {

    @Autowired
    private List<AbstractOrderHandle> orderHandleList;


    @PostConstruct
    public void init(){
        //如果List没有按照@Order注解方式排序，可以通过如下方式手动排序
        Collections.sort(orderHandleList, AnnotationAwareOrderComparator.INSTANCE);
        int size = orderHandleList.size();
        for (int i = 0; i < size; i++) {
            if(i == size -1){
                orderHandleList.get(i).setNext(null);
            } else {
                orderHandleList.get(i).setNext(orderHandleList.get(i + 1));
            }
        }

    }
```

```
    /**
     * 执行处理
     * @param context
     * @return
     */
    public OrderAddContext execute(OrderAddContext context){
        context = orderHandleList.get(0).execute(context);
        return context;
    }
}
```

最后，我们编写单元测试，来看看效果。

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class CalculatorServiceTest {

    @Autowired
    private OrderHandleManager orderHandleManager;

    @Test
    public void test(){
        orderHandleManager.execute(new OrderAddContext());
    }

}
```

运行结果与预期一致！

```
通过seqId，检查客户是否重复下单
检查请求参数，是否合法，并且获取客户的银行账户
检查银行账户是否合法，调用银行系统检查银行账户余额是否满足下单金额
```

三、小结

本文主要围绕在 SpringBoot 中如何引入责任链设计模式，介绍了三种玩法，其中第二种用法最多，其次就是第一种，第三种用的比较少，第三种本质是一种链式写法，可能理解上不如第一种直观，但是效果是一样的。

有效的使用责任链设计模式，可以显著降低业务代码的复杂度，可读性更好，更容易扩展，希望对大家有帮助！

编程学习部

大公司为什么禁止在SpringBoot项目中使用@Autowired注解?

黄进广寒

python创意编程展（二）

小桥和流水还有人家