从代码重构出发建立责任链框架

国际技术部-配运研发组

林一天

1 前言

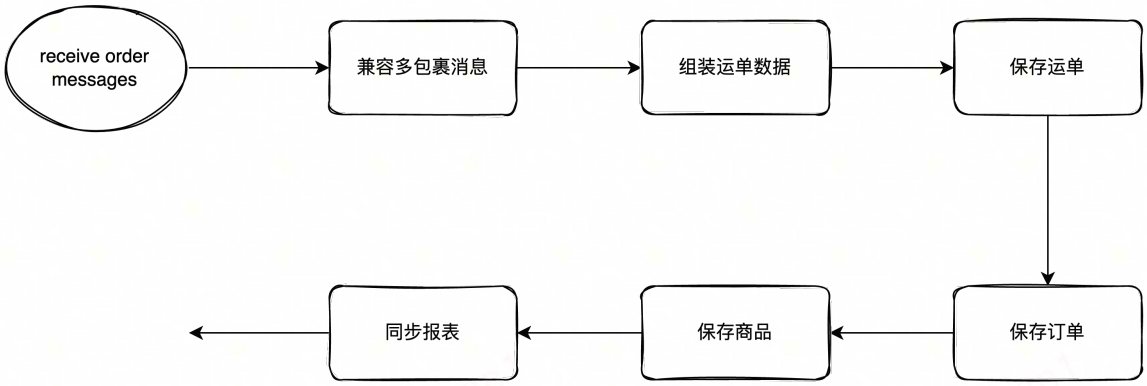
接单模块是DMS的核心功能, 从一开始就承担了小包和海外仓的接单业务, 后续的分拣机和国际快递业务依然使用了这个模块的代码. 业务的兼容压力迫使DMS必须开始重构接单的核心代码

2 一段代码的重构

原代码

```
@JProfiler(jKey = "LDMSEC-NEW.OmsOrderServiceImpl.saveOrder", mState = {JProEnum.TP, JProEnum.FunctionError})
public void saveOrder(OmsOrderDto omsOrderDto) {
    convertMultiPackage(omsOrderDto);
    logger.info("[OmsOrderServiceImpl][saveOrder] 从fop接单 -> waybillCode={}, omsOrderDto={}", omsOrderDto.getWaybillCode(), JSON.toJSONString(omsOrderDto));
    WaybillFlow waybillFlow = null;
    if (StringUtils.isNotBlank(omsOrderDto.getTransferCode())) {
        WaybillFlow queryDto = new WaybillFlow();
        queryDto.setTransferCode(omsOrderDto.getTransferCode());
        waybillFlow = dmsWaybillFlowService.getWaybillFlowByCondition(queryDto);
        if (!Objects.isNull(waybillFlow)) {

            //更新换单操作人所属仓库
            omsOrderDto.setEcCode(waybillFlow.getEcCode());
        }
    }
    List<CWaybillExtend> cWaybillExtendList = Oms2WaybillConvert.convert(omsOrderDto);
    //海外hub直接批量插入, 没有换单逻辑
    if (StringUtils.isNotEmpty(omsOrderDto.getWaybillSign())
            && EcWaybillSignUtils.isOverSeaHub(omsOrderDto.getWaybillSign())) {
        waybillService.saveOrUpdateByOutWaybillNo(cWaybillExtendList);
    } else {
        waybillService.saveOrUpdateToMainWaybill(cWaybillExtendList);
        //如果waybillflow不为空, 则自动称重
        try {
            //记录新老单号日志流水 注意啊, 不是麦哲伦的
            addWaybillFlowLog(waybillFlow, omsOrderDto);
        } catch (Exception e) {
            logger.error("记录新老单号流水发生异常,waybillCode:" + omsOrderDto.getWaybillCode(), e);
        }
    }

    So so = Oms2SoConvert.convert(omsOrderDto);
    soService.insertOrUpdate(so);

    List<Goods> goodsList = Oms2GoodsConvert.convert(omsOrderDto);
    if (CollectionUtils.isNotEmpty(goodsList)) {
        goodsService.batchInsert(goodsList);
    }

    // 同步报表
    this.doSendBaseReport(omsOrderDto);
}
```

2.1 总体逻辑梳理



2.2 初步重构

把散乱的代码按逻辑节点做内聚, 初步重构的结果如下

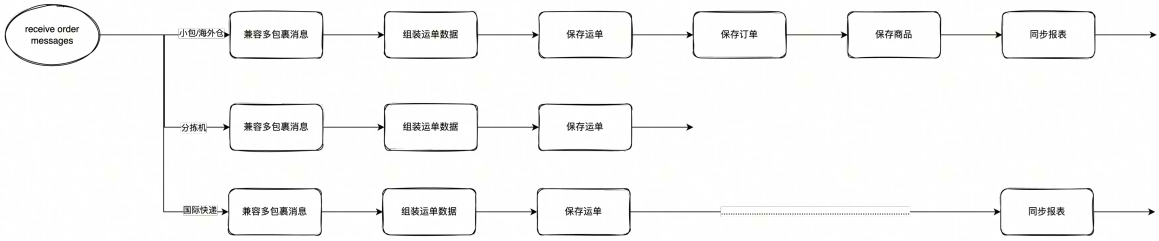
```
public void saveOrder(OmsOrderDto omsOrderDto) {
    // 兼容多包裹消息
    convertMultiPackage(omsOrderDto);
    // 组装运单数据
    List<CWaybillExtend> cWaybillExtendList = Oms2WaybillConvert.convert(omsOrderDto);
    // 保存运单
    convertAndSaveWaybill(cWaybillExtendList);
    // 保存订单
    convertAndSaveSo(omsOrderDto);
    // 保存商品
    convertAndSaveGoods(omsOrderDto);
    // 同步报表
    this.doSendBaseReport(omsOrderDto);
}
```

到这一步, 接单的代码有了条理, 再也不是一大坨杂乱的莫名其妙的小逻辑单元的滚热浓汤

2.3 总有业务要兼容

京东国际物流的业务发展迅猛, 在短时间内DMS团队又迎来了分拣机和国际快递的业务, 但原有的接单逻辑只能用来承接小包和海外仓的业务. 遇到新业务就开辟新的交互通道会破坏系统的功能内聚度, 让系统交互变得杂乱无章, 不符合基本的设计原则, 必须想办法让原有的接单逻辑可以兼容新的业务.

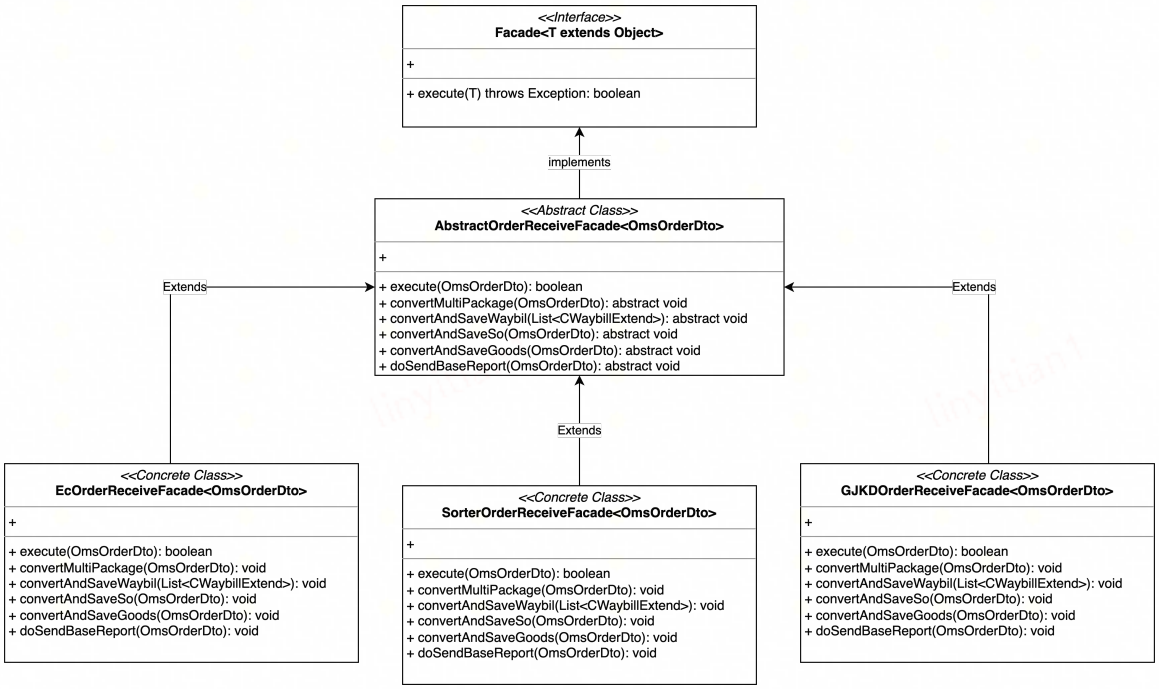
2.4 业务对比和兼容方案



三种业务的逻辑接近, 所以解决方案很简单. 在初步重构的步骤, 已经做了功能内聚, 同时, 接单的代码已经完成了模板化, 完全可以使用模板方法进行抽象, 具体步骤如下:

1. 对接单的模板代码抽象出Facade接口
2. 实现一个Facade接口的抽象类 AbstractFacade
3. 把接单的模板代码迁移到AbstractFacade的执行方法, 并增加模板代码引用的每个方法的抽象声明
4. 增加三个AbstractFacade的实现类, 分别实现 小包/海外仓,分拣机和 国际快递的 接单逻辑

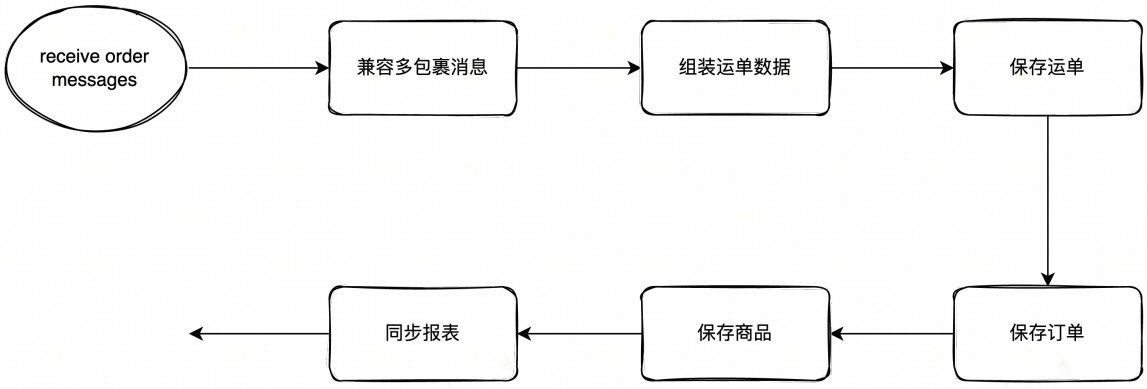
操作后的类图如下:



当然, 也可以在 AbstractOrderReceiveFacade 做点手脚, 把现在声明的抽象方法都用空方法去实现, 这样具体的实现类可以只override自己关注的领域的代码, 比如分拣机可以只实现个性化的保存运单的方法, 其他的直接使用默认的空方法, 国际快递同理

3 永远的拓展性准备

目前为止好像一切都很完美, DMS拥有了统一的接单入口, 新来的业务类型也和老业务流畅兼容, 似乎这是一次大获全胜的代码重构之旅. 但回看我们的代码实现, 在达成兼容的同时, 却没有完全打开代码的拓展潜力



回来再看一次接单的全局流程, 思考下面两个场景:

- 新业务需要在接单的同时保存包裹的称重信息(新增功能节点)
- 新/老业务需要调整接单流程的顺序: 先保存订单, 再保存运单(调整功能节点顺序)

对于第一种场景, 可以在Facade的执行方法增加 保存称重信息的 方法调用, 但对于第二种场景, 就必须新增Facade , 并让新/老业务改为继承新的Facade, 而且每一次细微的调整都可能诞生一个新的Facade. 很明显, 使用了模板方法的代码依然可能被迫破坏开闭原则, 为了在不更改代码的前提下, 做到支持调整顺序和新增删减, 必须在 模板方法的基础上再进一步

4 责任链的引进

4.1 什么是责任链模式(chain of responsibility)

> 顾名思义，责任链模式（Chain of Responsibility Pattern）为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。这种类型的设计模式属于行为型模式。
>
> 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。

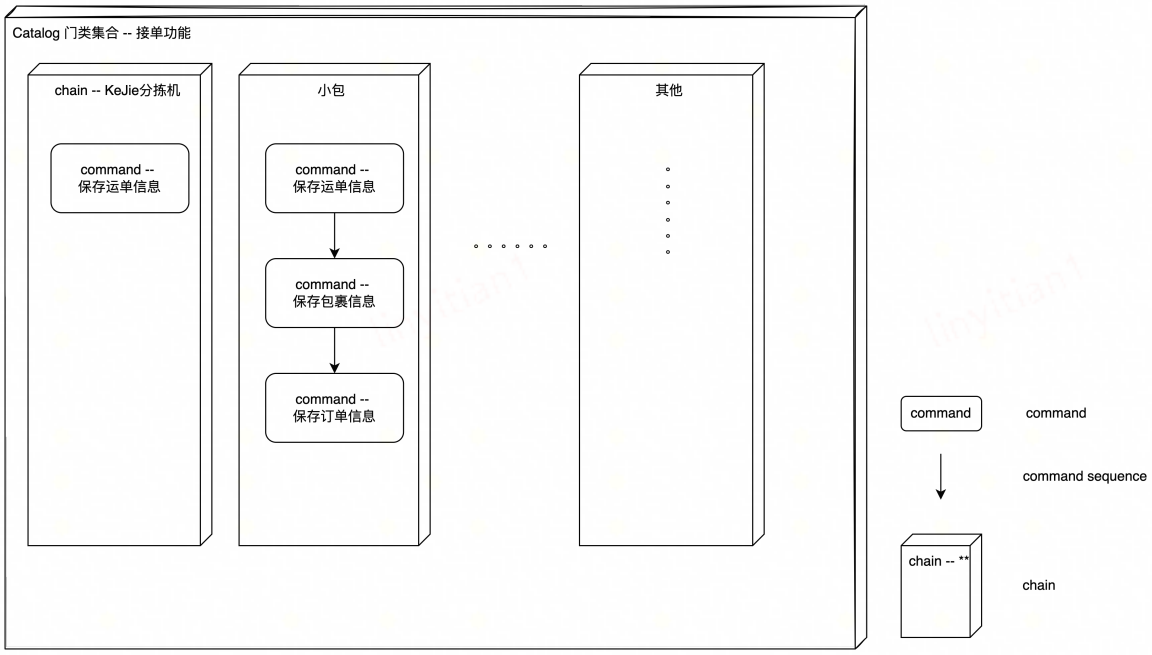
上面所说的每个接收者都包含对另一个接收者的引用是为了保证接收者的执行顺序, 用其他的有序结构保存接收者也是可以的, 类似的常见代码可以参考 servlet-api里的Filter 或 spring-mvc 里的Interceptor

5 责任链的实践

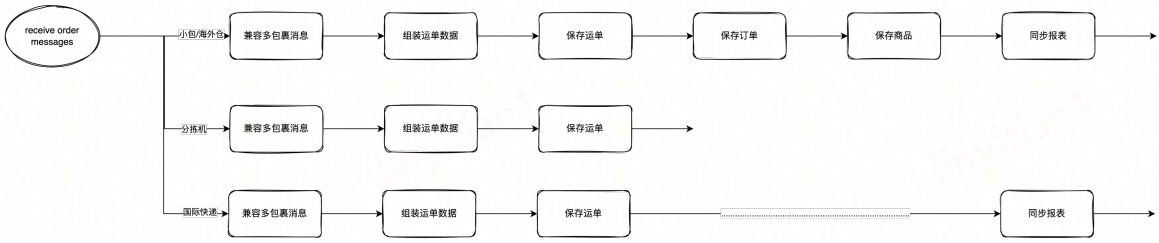
5.1 基本结构

现在可以开始设想用责任链的模式去实现接单逻辑

Catalog-Chain-Command的结构如下:



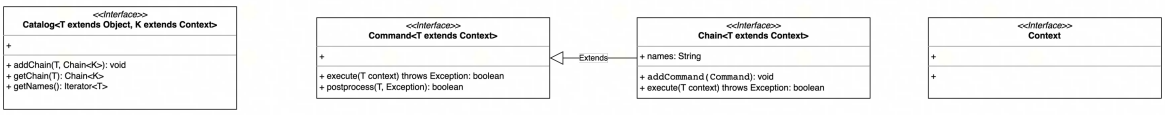
Command配置如下:



具体操作步骤如下:

> 1 链上所有接收者都必须接受相同的请求, 所以需要有一个请求的抽象, 毕竟没有任何一个开发者希望一个工具只能应用于一种业务
>
> 2 依托于 请求的抽象作为 入参, 需要一个可以描述逻辑单元(如上图的[保存运单]功能节点)的抽象定义
>
> 3 责任链的核心是维护逻辑单元的顺序, 需要一个可以持有逻辑元件顺序, 并且可以按照不同key 选择相应的执行链的 持有者抽象

基于以上三点, 第一级设计如下:

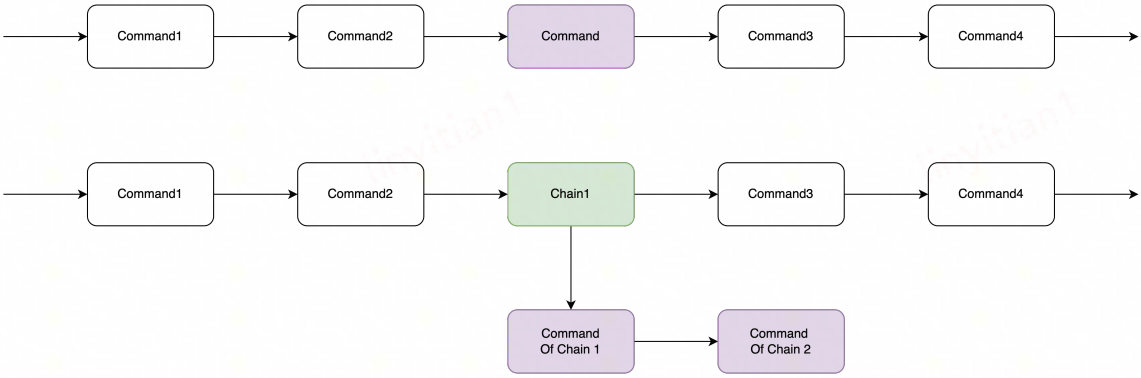


> Catalog： 按业务持有不同责任链配置
>
> Context： 不同于常见的其他设计模式, 责任链不会区分 出入参，而是用上下文的概念去维护整个责任链执行过程中产生的各种参数. 这种设计一般可以避免其他设计模式中可能出现的为了拼凑参数而导致性能下降的问题; 其缺点是 上下文的规模将随着业务复杂度上升而持续膨胀
>
> Command： 用来描述逻辑单元, 对于所在的责任链来说, Command是最小的执行单元, 其接收 上游的Context作为入参, 在对context进行了修饰后, 再传送给下游的 Command
>
> Chain： 责任链配置接口, addCommand方法在原有链条上追加一个尾部Command.

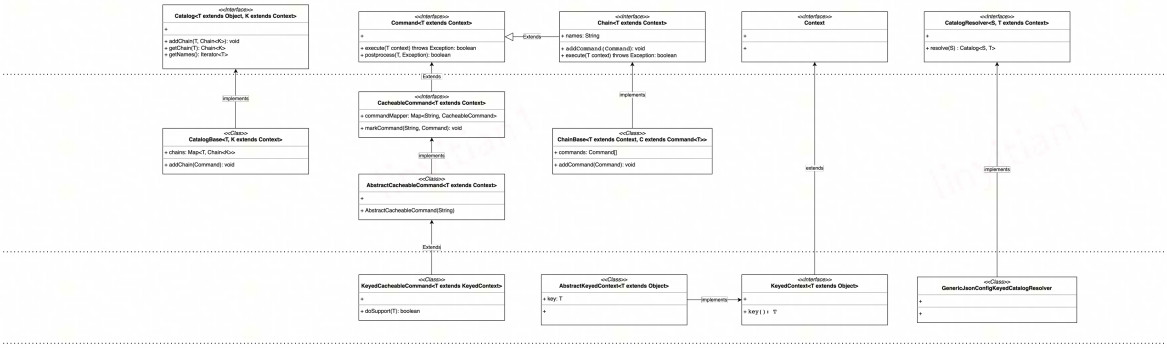
5.2 关键的Extends

Chain继承于Command, 这是一个非常重要的设计.

设想 Chain 和 Command 没有继承关系, 在调用 Chain::addCommand的时候, 参数的对象类型只能是Command类型, 无法接纳 Chain类型, 也就是Chain完全是线性结构, 而无法实现 二维及以上维度的拓展, 如下:



5.3 具体的框架设计



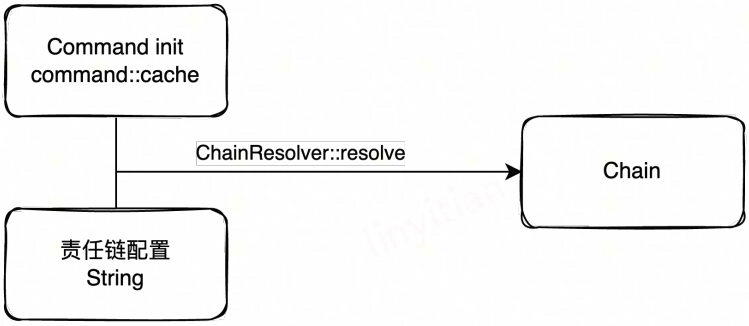
5.3.1 第一层:

> CatalogResolver是一个通过数据源解析成Catalog配置的解析器，我们默认的解析器是一个JSON字符串, 但对于其他场景, 可能我们需要支持更复杂的Catalog-Chain-Command类型, 此时需要根据场景自定义对应的CatalogResolver

5.3.2 第二层:

> CatalogBase 是Catalog的基本实现, 核心数据结构是 Map<T, Chain> , 通过 这个 Map 持有了各个业务对应的 责任链配置

> CacheableCommand 和 AbstractCacheableCommand 内部维护了一个 Map<String, Command> 的数据结构, 主要是为了给每个Command赋予一个exclusive的name
>
> ChainBase 是责任链执行的一种基本逻辑实现

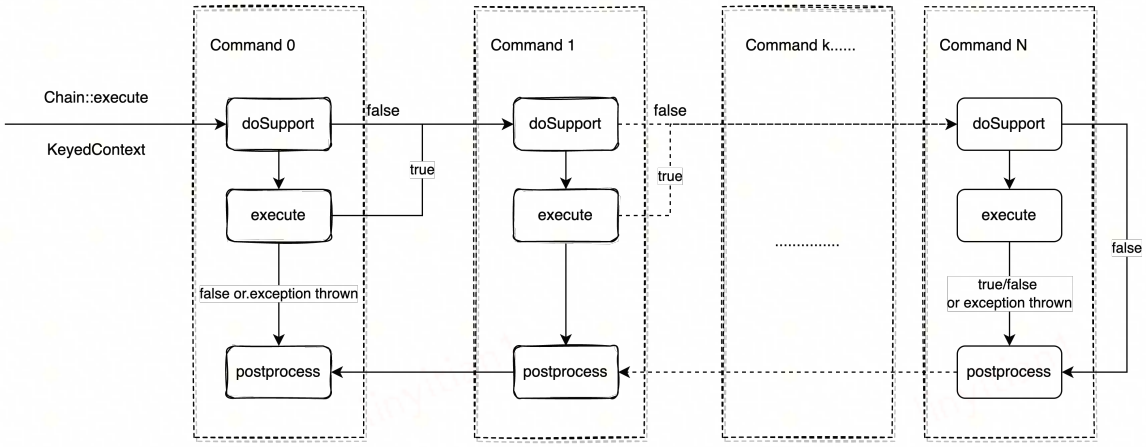


用Command构建Chain的流程大致如上

这里重点看一下ChainBase的实现, 先看一下代码:

```java
public boolean execute(T context) throws Exception {
    if (context == null) {
        throw new IllegalArgumentException();
    }
    boolean saveResult = false;
    Exception saveException = null;
    int i = 0;
    int n = commands.length;
    for (i = 0; i < n; i++) {
        try {
            saveResult = commands[i].execute(context);
            if (!saveResult) {
                break;
            }
        } catch (Exception e) {
            saveException = e;
            break;
        }
    }
    if (i >= n) {
        i--;
    }
    boolean handled = false;
    boolean result = false;
    for (int j = i; j >= 0; j--) {
        try {
            result = commands[j].postprocess(context, saveException);
            if (!result) {
                handled = true;
            }
        } catch (Exception e) {
            // Silently ignore
        }
    }
    // Return the exception or result state from the last execute()
    if ((saveException != null) && !handled) {
        throw saveException;
    } else {
        return (saveResult);
    }
}
```

> 责任链的执行, 是从第一个 Command的 execute开始，每执行完成一个 Command::execute , 就把传入的Context 递给下一个Command , 除非 上一个Command 执行的结果是 false 或者是 Exception
>
> thrown.



同时还可以发现，在执行完最后一个Command 或者 Command返回了结束标记的时候，Chain还会从最后一个被执行的Command开始倒序调用 Command 的 postprocess 方法.

5.3.3 第三层:

前面已经有了基本概念的抽象和 基础功能实现, 这一层就可以开始实现业务真正需要的具体逻辑

```
public abstract class KeyedCacheableCommand<T extends KeyedContext> extends AbstractCacheableCommand<T> {
    2 usages  linyitian1
    public KeyedCacheableCommand(String name) { super(name); }
    1 usage  linyitian1
    @Override
    public boolean postprocess(T context, Exception saveException) { return true; }
    2 usages  linyitian1
    @Override
    public boolean execute(T context) throws Exception {
        if (!doSupport(context)) {
            return false;
        }
        return doExecute(context);
    }
    1 usage  linyitian1
    protected boolean doSupport(T context) { return true; }
    1 usage  2 implementations  linyitian1
    protected abstract boolean doExecute(T context) throws Exception;
}
```

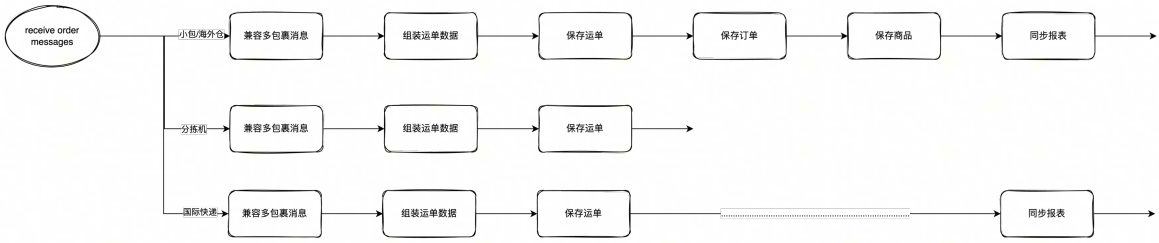
基于简单的模板方法的Command实现

万事俱备, 现在可以开始用责任链的思想重构我们的代码了

6 实践

先回顾一下上一阶段的重构结果

```java
public void saveOrder(OmsOrderDto omsOrderDto) {
    // 兼容多包裹消息
    convertMultiPackage(omsOrderDto);
    // 组装运单数据
    List<CWaybillExtend> cWaybillExtendList = Oms2WaybillConvert.convert(omsOrderDto);
    // 保存运单
    convertAndSaveWaybill(cWaybillExtendList);
    // 保存订单
    convertAndSaveSo(omsOrderDto);
    // 保存商品
    convertAndSaveGoods(omsOrderDto);
    // 同步报表
    this.doSendBaseReport(omsOrderDto);
}
```



在已经有了一套代码模板之后, 可以很轻易的按照 Catalog-> Chain -> Command 的顺序把代码重构为责任链

6.1 Catalog的实现

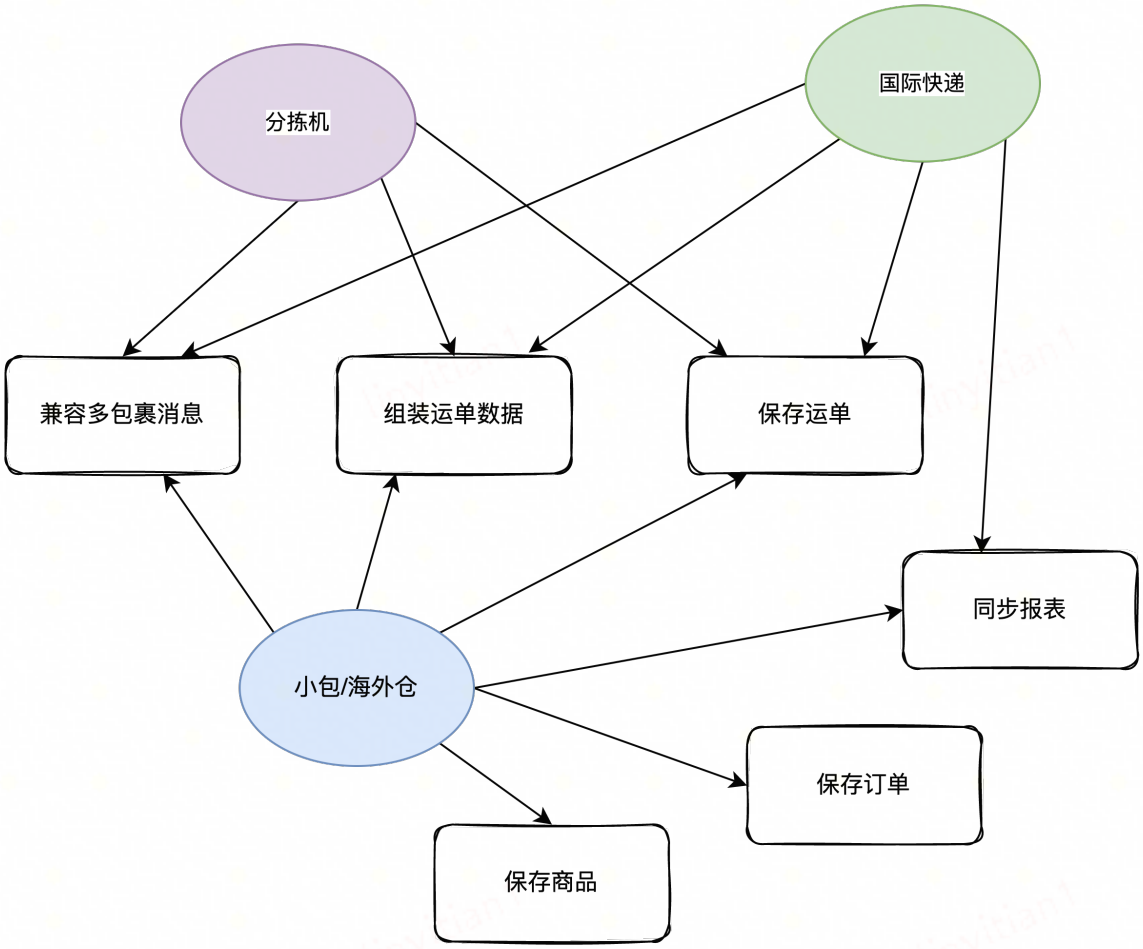
通过公司的DUCC工具可以把静态配置升级为动态配置, 依托 设计的CatalogResolver 和DUCC同样可以实现责任链配置的动态化

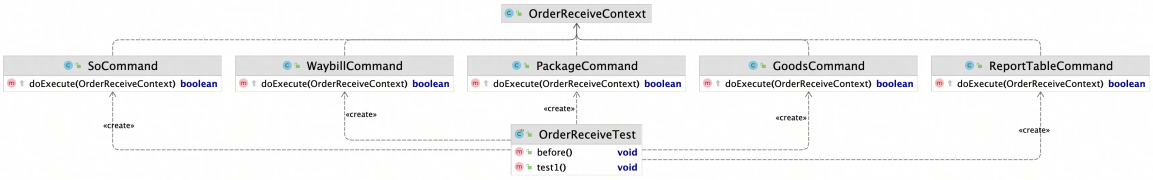
```java
public class GenericJsonConfigKeyedCatalogResolver implements CatalogResolver<String, KeyedContext> {
    1 usage  linyitian1 *
    @Override
    public Catalog<String, KeyedContext> resolve(String config) {
        Map<String, List<String>> catalogMap
                linyitian1
                = JSONObject.parseObject(config, new TypeReference<Map<String, List<String>>>() {});
        CatalogBase<String, KeyedContext> catalogBase = new CatalogBase();
        catalogMap.entrySet().stream().forEach(stringListEntry -> {
            List<CacheableCommand> collect
                    = stringListEntry.getValue().stream().map(CacheableCommand.commandMapper::get).collect(Collectors.toList());
            catalogBase.addChain(stringListEntry.getKey()
                    , new ChainBase<>(collect));
        });
        return catalogBase;
    }
}
```

这里只是适配DMS接单场景的解析实现, 其他业务应该因地制宜制定自己的解析逻辑

6.2 Command的实现

归总三个业务的接单逻辑涉及的逻辑单元, 为每一个逻辑单元都 创建一个Command , 基本逻辑如下:





类图如上

下面展示的是重构后, 接单逻辑的具体调用实现:

```java
public class OrderReceiveTest {

    1 usage
    final GenericJsonConfigKeyedCatalogResolver genericJsonConfigKeyedCatalogResolver = new GenericJsonConfigKeyedCatalogResolver();

    @Before
    public void before() {
        new GoodsCommand( name: "goods");
        new PackageCommand( name: "package");
        new ReportTableCommand( name: "reportTable");
        new SoCommand( name: "so");
        new WaybillCommand( name: "waybill");
    }

    创建一个Command 并cache到一致的map中
    @Test
    public void test1() throws Exception {
        //
        String jsonConfig = "{\"kejie\":[\"package\", \"waybill\"], \"boardCross\":[\"package\", \"waybill\", \"reportTable\"]}";
        Catalog<String, KeyedContext> catalogByWaybillSign = genericJsonConfigKeyedCatalogResolver.resolve(jsonConfig);

        WaybillSignKeyedContext waybillSignKeyedContext = new WaybillSignKeyedContext(WaybillSign.FBA);

        Chain<KeyedContext> chain = catalogByWaybillSign.getChain(waybillSignKeyedContext.key().getMark());
        boolean execute = chain.execute(waybillSignKeyedContext);

        System.out.println(execute);
    }

}
```

通过动态配置，后面接单代码需要新增节点和调整节点顺序都可以支持了

7 其他

7.1 postprocess方法的作用是什么?

责任链上的各个节点往往存在下游节点依赖上游节点产生的临时资源, 而更上游的节点不允许这个临时资源露出的场景: 比如 一长串的节点都需要依托第一个节点开启的文件流对象进行流处理，这种场景下, 文件流不能露出超过 第一个节点, 同时交托给下游节点可能导致提前关闭. 最好的方法就是在 doExecute方法中开启 文件流并集成到 context，再在回头调用到自己的 postprocess 方法时把文件流关闭掉. 这里体现的是一种系统资源管理的思想

7.2 为什么不直接用Map代替Context?

使用Map作为共同的请求介质也是一种常见的设计, 比如 Apache commons-chain 就是使用的Map

DMS抛弃这种做法, 主要是因为使用Map可能导致几个严重的代码和系统隐患

1 缺少严格的CR可能导致运行时错误

K-V结构的读写依赖传入传出的Key的字面量, 编写代码的过程中RD如果没有发现, 很可能导致运行时才暴露出来

2 无法避免同一个概念重复出现在上下文的情况

8 总结

大多数设计模式都诞生于代码重构/设计, 本文从一个代码重构的case出发，讲解了原始代码先后使用模板方法和责任链模式进行重构的过程, 希望能让读者理解责任链模式的本质

参考:

Wiki pedia : Template method pattern: https://en.wikipedia.org/wiki/Template_method_pattern

Wiki pedia : Strategy pattern: https://en.wikipedia.org/wiki/Strategy_pattern