浅谈简单工作流设计——责任链模式配合策略与命令模式的实现

本文以项目中的一个工作流模块，演示责任链模式、策略模式、命令模式的组合实现！

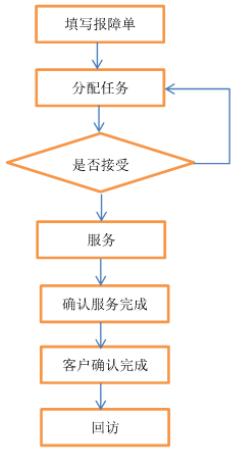
流程简介

最近在做的一个项目，涉及到的是一个流程性质的需求。关于工程机械行业的服务流程：服务任务流程和备件发运流程。

项目之初，需求不是很清晰，算是演化模型吧。先出一个简单版本，然后根据用户的使用情况，再进一步探测新需求。所以也就是说这两个流程中的每一步暂时都不是固定的，而应该是可配置、可增减的。

目前暂定的两个流程示意图如下：

暂定服务流程：



以上为两个流程的大致过程，当然实际过程中，可能还要走其他的流程。

但是，仔细分析，你会看到。不管有多少个中间步骤，它们始终都对应着它们在该流程中所处的状态：

```
[csharp] view plain copy print ?
01. /// <summary>
02.     /// 服务流程状态枚举
03.     /// </summary>
04.     public enum MaintanStateEnum
05.     {
06.         non_assign,         //已创建，待分配
07.         non_accept,         //已分配，待接收
08.         maintaining,        //已接收，服务中
09.         non_confirm,        //完成服务，待确认
10.         non_userConfirm,    //已确认，待客户确认
11.         non_feedback,       //客户已确认，待回访
12.         feedbacked,         //回访完成，流程结束
13.         goback              //退回分配，此为动作，为了方便编码，不对应服务状态
14.     }
```

你会看到non_后面跟的都是一个个动作。在这里分清状态和动作是很重要的，不然就很难理清了。还有有时一个动作对应着前后状态，不要出现重复的状态比如：created(创建完成)和non_assign(待分配)在这里就是所谓的重复状态。

这些状态其实就是贯穿着整个流程的主线，类似于一个城市的主干道一样。我们只要抓着这样一天线索来思考，就能够化繁为简。

每个步骤可配置，各个步骤不相耦合，实现调用端一致性——责任链模式

而责任链模式，正是为此而生的！

在这里，我采用了责任链模式来封装这种步骤的不确定带来的变化。

首先我们有必要先了解一下，什么是责任链模式：

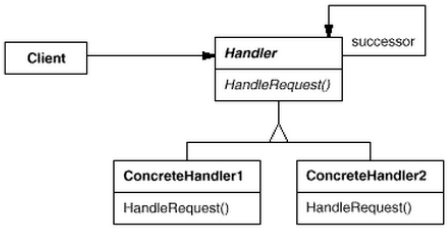


Figure 1. CoR class diagram

职责链模式（Chain of Responsibility）：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

适用场景：

1、有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定；

2、在不明确指定接收者的情况下，向多个对象中的一个提交一个请求；

3、处理一个请求的对象集合应被动态指定。

可以看到无论是上面哪种场景，都存在一个多对一的关系。在这个流程中，一很明显对应着服务流程（说白了就是数据库服务任务的一条记录）。大部分情况下，我们都在完成对一条服务中相关字段的修改，同时置不同的服务状态。而多，在这里应该对应着不同的步骤。

下面来看看实现：

```
[csharp] view plain copy print ?
/// <summary>
    /// 维修服务 处理器-抽象基类
    /// </summary>
    public abstract class ServiceHandler
    {
        protected ServiceHandler nextHandler;

        public virtual void Handle(MaintenanceForm maintenanceForm, object otherParams)
        {
            if (nextHandler!=null)
            {
                nextHandler.Handle(maintenanceForm, otherParams);
            }
        }

    }
```

这是一个抽象处理器，流程中每个步骤的处理器，会覆写这个处理器的处理方法：

创建：

```
[csharp] view plain copy print ?
/// <summary>
   /// 创建服务单流程--应对接线员/服务接收员角色【或者其他具有创建权限的其他角色】
   /// </summary>
  public class CreateHanlder:ServiceHandler
   {
      public override void Handle(MaintenanceForm maintenanceForm, object otherParams)
      {
          //如果已创建(未分配)
          if (maintenanceForm.CurrentState==MaintanStateEnum.non_assign)
          {
              //创建该服务单

              return;
          }
          else    //已分配,则传递给下一个流程(假定为分配流程)
          {
              base.Handle(maintenanceForm,otherParams);
          }
      }

      //设置下一个处理流程
      public ServiceHandler NextHandler
      {
          set
          {
              base.nextHandler = value;
```

关闭

```
        }
    }
```

```
[csharp] view plain copy print ?
}
```

分配：

```
[csharp] view plain copy print ?
/// <summary>
    /// 服务流程之分配流程--应对服务处长等分配人员角色【或其他具有分配权限的人员】
    /// </summary>
    public class AssignHandler:ServiceHandler
    {
        public override void Handle(Model.MaintenanceForm maintenanceForm, object otherParams)
        {
            //如果当前处于已分配(未接收)状态
            if (maintenanceForm.CurrentState == MaintanStateEnum.non_accept)
            {
                //进行分配

                //记录日志
                Log(maintenanceForm);
                return;
            }
            else
            {
                //进入下一流程
                base.Handle(maintenanceForm,otherParams);
            }

        }

        public ServiceHandler NextHandler
        {
            set
            {
                base.nextHandler=value;
            }
        }
```

其他流程我就不一一贴出代码。下面来分析一下以上这几个处理器，有什么特别之处。首先，我们看出了，它是以状态为导向的（在Handle方法的判断中）。每个步骤都有一个NextHandler属性，用来配置下一个处理器，这就可以串联他们到一个流程中。大家可以看到每个Handle方法最后都有一个"return;"语句。没错，这里我使用了不完整的责任链模式。也就是一个流程不是一次走结束的，因为它们可能不是一个时间上的连贯，也可能不是一个人走完所有的流程，比如某人负责创建任务单，某人负责分配等。

下面看看，如何来串联他们到一个流程中：

```
[csharp] view plain copy print ?
//初始化服务流程链
        static ServiceHandler InitServiceChain()
        {
            //此处应用责任链模式，这样对维修服务单处理的入口只有一个
            //根据当前的服务所处的状态可以导向到特定的处理流程【流程为链式】

            CreateHanlder createHandler = new CreateHanlder();
            AssignHandler assignHandler=new AssignHandler();
            FinishHandler finishHandler=new FinishHandler();
            FeedbackHandler feedbackHandler = new FeedbackHandler();

            //显式指定流程链
            /*
             *注：此处体现流程可增减，可配置【例如写入配置文件】
             */
            createHandler.NextHandler=assignHandler;
            assignHandler.NextHandler=finishHandler;
            finishHandler.NextHandler = feedbackHandler;
            feedbackHandler.NextHandler = null;



            //返回流程启示，类似一个链表结构的头部指针
            return createHandler;
        }
```

上面中第二大代码块，即实现了所有处理器的拼装与整合（只是示例，并不完整）

下面我们来看看客户端调用：

```
[csharp] view plain copy print ?
static void Main(string[] args)
        {
            ServiceHandler serviceHandler = InitServiceChain();
            //应用场景：

            //场景一【演示创建服务单流程】
            //点击新增按钮，弹出窗口，创建维修服务表单
            MaintenanceForm newMaintenanceForm=new MaintenanceForm();
```

```
            newMaintenanceForm.Creator = "yh";
            newMaintenanceForm.CreatedTime = DateTime.Now;
            newMaintenanceForm.CurrentState = MaintanStateEnum.non_assign;

            serviceHandler.Handle(newMaintenanceForm,null);  //创建

            //场景二【演示分配流程】
            //在未分配维修单列表中选择一条维修单，点击分配
            MaintenanceForm maintenanceForm = new MaintenanceForm();
            maintenanceForm.LastModifiedTime = DateTime.Now;
            maintenanceForm.CurrentState = MaintanStateEnum.non_accept;

            serviceHandler.Handle(maintenanceForm,null);     //创建(跳过)->分配

            //场景三【演示已分配被维修人员退回,重新分配流程】
            //在被退回维修单列表中选择一条维修单，点击分配/重新分配
            maintenanceForm.LastModifiedTime = DateTime.Now;
            maintenanceForm.CurrentState = MaintanStateEnum.goback;

            serviceHandler.Handle(maintenanceForm,null);     //创建(跳过)->分配

            //场景四【演示接受流程】
            //在已分配待完成表单中，选择一条记录，点击接受任务按钮
            maintenanceForm.LastModifiedTime = DateTime.Now;
            maintenanceForm.CurrentState = MaintanStateEnum.maintaining;

            serviceHandler.Handle(maintenanceForm,null);     //创建(跳过)->分配(跳过)->接受

            //场景五【演示回访流程】
            //在已完成待回访表单中，选择一条记录，点击完成回访按钮
            maintenanceForm.LastModifiedTime = DateTime.Now;
            maintenanceForm.CurrentState = MaintanStateEnum.feedbacked;

            serviceHandler.Handle(maintenanceForm,null);     //创建(跳过)->分配(跳过)->接受(跳过).....->回访

            Console.Read();
        }
```

无论走哪个流程，整个的调用方法只有一个：

```
[csharp] view plain copy print ?
serviceHandler.Handle(maintenanceForm,null);
```

第一个参数为任务的实体，第二个为附带参数（如果没有，可不必传递）。在任务实体会在流程链中传递，实体中当前状态会指引它交给哪个Handler处理。这样无论你流程中步骤如何变化，在调用端的调用方式都是唯一的。而且你增减步骤对其他流程都不产生任何影响，唯一需要改变的就是组装他们的地方。比如，你需要在创建完服务单之后，走一个审核流程，那么增加它就是一个非常简单的动作。

这里关于发运流程的做法我就不多举例子了，大同小异。

实现每个步骤不同的日志记录方式之——策略模式

这里可能会对日志的记录有多种需求，比如发送Email给远端工程师或者某些领导等，存入数据库或平面文件备份.....

策略模式的细节就不多做介绍了，看实现：

```
[csharp] view plain copy print ?
/// <summary>
    /// 日志记录策略抽象类
    /// </summary>
    public abstract class LogStrategy
    {
        public abstract void Log(Object obj);
    }
```

```
[csharp] view plain copy print ?
/// <summary>
    /// 数据库策略--服务日志记录实现类
    /// </summary>
    public class ServiceDatabaseLogStrategy:LogStrategy
    {
        public override void Log(Object obj)
        {
            if (obj == null)
            {
                throw new NullReferenceException("obj");
            }

            MaintenanceForm maintenanceForm = obj as MaintenanceForm;
            //插入数据库
            Console.WriteLine("数据库日志记录");
            Console.WriteLine();
        }
    }
```

```
[csharp] view plain copy print ?
/// <summary>
```

关闭

```
    /// E-mail策略--服务日志记录实现类
    /// 某些不关注细节的角色
    /// (比如领导，他们只关注结果，但他们需要宏观的把握，那么在完成服务步骤，日志可以通过email发送到领导邮箱)
    /// </summary>
    public class ServiceEMailDatabaseLogStrategy:LogStrategy
    {
        public override void Log(object obj)
        {
            if (obj==null)
            {
                throw new NullReferenceException("obj");
            }

            MaintenanceForm maintenanceForm = obj as MaintenanceForm;
            //发送到指定邮箱操作
            Console.WriteLine("email日志记录");
            Console.WriteLine();
        }
    }
```

```
[csharp] view plain copy print ?
/// <summary>
    /// 平面文件策略--服务日志记录实现类
    /// 可以支持文本文档或者XML文档，便于查看和交互
    /// </summary>
    public class ServiceFileDatabaseLogStrategy : LogStrategy
    {
        public override void Log(object obj)
        {
            if (obj==null)
            {
                throw new NullReferenceException("obj");
            }
            Console.WriteLine("文件日志记录");
            Console.WriteLine();
        }
    }
```

上面就是暂定的几个日志记录策略的实现，下面看看如何将它们组合到各个处理器中。首先，其实刚才处理器的抽象基类是有一个记录日志的虚方法的：

```
[csharp] view plain copy print ?
/// <summary>
    /// 维修服务 处理器-抽象基类
    /// </summary>
    public abstract class ServiceHandler
    {
        protected ServiceHandler nextHandler;

        public virtual void Handle(MaintenanceFormmaintenanceForm, object otherParams)
        {
            if (nextHandler != null)
            {
               nextHandler.Handle(maintenanceForm, otherParams);
            }
        }

        public virtual void Log(MaintenanceFormmaintenanceForm)
        {
            //记录入数据库

        }
    }
```

上面的抽象类实现的是，默认记录入数据库

其实，每个步骤的处理器中也都存在日志记录的处理(只是为了不干扰讲解责任链，而省略了)

```
[csharp] view plain copy print ?
public classCreateHanlder:ServiceHandler
    {
       public override voidHandle(MaintenanceForm maintenanceForm, object otherParams)
       {
           //如果未分配
           if(maintenanceForm.CurrentState==MaintanStateEnum.non_assign)
           {

               //记录日志
               Log(maintenanceForm);
               return;
           }
           else    //已分配,则传递给下一个流程(假定为分配流程)
           {
              base.Handle(maintenanceForm,otherParams);
           }
       }

       //设置下一个处理流程
```

关闭

```
        public ServiceHandler NextHandler
        {
            set
            {
                base.nextHandler = value;
            }
        }

        public LogStrategy LogStrategy { get;set; }

        public override void Log(MaintenanceFormmaintenanceForm)
        {
            //如果没有显式指定日志记录策略，则调用基类记录方法【存入DB】
            if (LogStrategy == null)
            {
                base.Log(maintenanceForm);
            }
            else
            {
               LogStrategy.Log(maintenanceForm);
            }
        }
    }
```

这里有一个LogStrategy（日志策略基类）类型的属性，用来对外提供配置接口。每个处理器类都覆写了处理器基类的Log方法，逻辑为：如果有日志记录策略，则以日志记录策略来记录，否则用基类的记录方式(DB方式)。

上面在Handle方法返回之前，调用了被覆写的Log方法。

下面看看，外面是如何组装日志记录策略的：

```
[csharp] view plain copy print ?
static ServiceHandlerInitServiceChain()
        {
            //此处应用责任链模式，这样对维修服务单处理的入口只有一个
            //根据当前的服务所处的状态可以导向到特定的处理流程【流程为链式】

            CreateHanlder createHandler = newCreateHanlder();
            AssignHandler assignHandler=newAssignHandler();
            FinishHandler finishHandler=newFinishHandler();
            FeedbackHandler feedbackHandler =new FeedbackHandler();

            //显式指定流程链
            /*
             *注：此处体现流程可增减，可配置【例如写入配置文件】
             */
           createHandler.NextHandler=assignHandler;
           assignHandler.NextHandler=finishHandler;
            finishHandler.NextHandler =feedbackHandler;
            feedbackHandler.NextHandler = null;

            //显式指定日志的记录策略
            /*
             * 支持多种日志记录策略(包括DB、Email、File)
             * 如果不配置，缺省记录方式为DB
             *
             * 此处体现可配置性--某些高层只关注结果
             * (也就是说可能他们只关注‘服务完成’流程，那么该流程的日志记录策略就可以设置为email)
             *
             * 配置同样可以写入配置文件,也可支持一个流程多种日志记录方式
             *
             */

            //日志记录策略示例:
            createHandler.LogStrategy = newServiceDatabaseLogStrategy();          //DB
            assignHandler.LogStrategy = newServiceFileDatabaseLogStrategy();      //File
            finishHandler.LogStrategy = newServiceEMailDatabaseLogStrategy();     //email
            feedbackHandler.LogStrategy = newServiceEMailDatabaseLogStrategy();    //email

            //返回流程启示，类似一个链表结构的头部指针
            return createHandler;
        }
```

可以看到在组装流程的时候我们同时组装了日志记录的策略。事实上，这里每个流程只对应了一种策略，当然可以为一个流程配置几个日志记录策略啦(修改为List<LogStrategy>，然后在处理器的Log方法中依次调用)。

~~处理数据库处理逻辑调用端的一致性——命令模式~~

关闭

不知道大家平时是否都习惯了用三层架构来应对一般项目。在我们的项目中，BLL层是一个傀儡这是一个事实（不对此进行辩论，无论你的是不是，反正我们的项目是了，至于对与错，大家心里都明白）。这里，我一改往日数据库操作调用端采用的各种繁杂的XXXBLL.XXX()的不一致性。改为采用了命令模式来实现对数据库的增改删查。至于理由——业务都由各个处理器实现，没有采用BLL形式的必要。同时我的数据库操作方法的调用端一直，修改就封装在内部。

看看实现：

```
[csharp] view plain copy print ?
/// <summary>
    /// 命令接口--可以支持：撤消/重做 操作
```

```
    /// </summary>
    public interface ICommand
    {
        //object Undo();

        object Execute();

        //object Redo();
    }
```

命令实现：

```
[csharp] view plain copy print ?
/// <summary>
    /// 实现创建服务任务单命令--Command模式
    /// </summary>
    public classCreateMaintenanceFormCommand:ICommand
    {
        private MaintenanceFormmaintenanceForm;
        publicCreateMaintenanceFormCommand(MaintenanceForm _maintenanceForm)
        {
            maintenanceForm = _maintenanceForm;
        }

        public object Execute()
        {
            //string str="insertinto....";
            //DB.ExecuteNonQuery(....);
            return null;
        }
    }
```

```
[csharp] view plain copy print ?
/// <summary>
    /// 分配维修任务命令实现
    /// </summary>
    public classAssignMaintenanceFormCommand:ICommand
    {
        private MaintenanceFormmaintenanceForm;
        publicAssignMaintenanceFormCommand(MaintenanceForm _maintenanceForm)
        {
            maintenanceForm = _maintenanceForm;
        }

        public object Execute()
        {
            //throw new NotImplementedException();
            return null;
        }
    }
```

且看创建服务单处理器中，如何调用：

```
[csharp] view plain copy print ?
public classCreateHanlder:ServiceHandler
    {
       public override voidHandle(MaintenanceForm maintenanceForm, object otherParams)
       {
           //如果未分配
           if(maintenanceForm.CurrentState==MaintanStateEnum.non_assign)
           {

               //构造创建服务单命令，并执行
               ICommand createCommand = newCreateMaintenanceFormCommand(maintenanceForm);
               createCommand.Execute();

               //记录日志
               Log(maintenanceForm);
               return;
           }
           else    //已分配,则传递给下一个流程(假定为分配流程)
           {
              base.Handle(maintenanceForm,otherParams);
           }
       }
```

关闭

参数都是，各个命令的构造方法传入，所以有灵活性。额外的参数，来自Handle方法的otherParams，所以参数传递没有产生限制。

当然，这里没有实现命令的撤销与重做。

总结

以上就是该服务流程所采用的三种设计模式——责任链模式、策略模式、命令模式。对设计不精，但是很有兴趣。这是我的一些想法，我觉得在大家设计一些简单工作流或者流程性质很强的需求的时候，能够有一定的指导意义！

顶 踩
8 2

- 上一篇 从零开始学.net多线程系列(三)——同步
- 下一篇 浅谈跨网站脚本攻击(XSS)的手段与防范(简析新浪微博XSS攻击事件)

我的同类文章

【杂七杂八】（12）

- 用Proxmox创建虚拟机教程 2012-05-18 阅读 3030
- OAuth工作原理随想——让你的系统提... 2012-03-04 阅读 5843
- 记住密码"功能的正确设计 2011-12-25 阅读 3134
- 浅谈人人网以及淘宝网的IM即时通信以... 2011-09-04 阅读 3506
- android 中关于 activity 的一些理解 2011-07-16 阅读 1205
- 浅谈公司核心业务数据表的重构——结... 2012-03-11 阅读 7162
- 辞旧迎新——年度web开发合辑，新年... 2012-01-22 阅读 9248
- 实时系统解决方案 TIBCO Rendezvous ... 2011-09-24 阅读 2719
- 智力题：关于进入房间一次，判断哪个... 2011-08-13 阅读 5036

更多文章

主题推荐 workflow 设计 微软 需求

猜你在找

数据库简明教程
NoSQL与Mongo DB数据库入门
MySQL数据库管理
OA工作流设计制作
Winform数据库编程:ADO.NET入门
和我一起学Windows Workflow Fo…
和我一起学Windows Workflow Fo…
SharePoint Workflow Visualiza…
SharePoint 2010 自定义状态机…
java设计模式二21责任链简单一…

查看评论

charmfeel 2楼 2014-11-12 17:57发表
挺好的，如果能有一个例子的源码来测试就更直观明了了。

Farley1005 1楼 2012-11-02 10:29发表
顶

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack VPN Spark ERP
IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5 Spring Apache .NET API
HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE
Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr Angular
Cloud Foundry Redis Scala Django Bootstrap

关闭