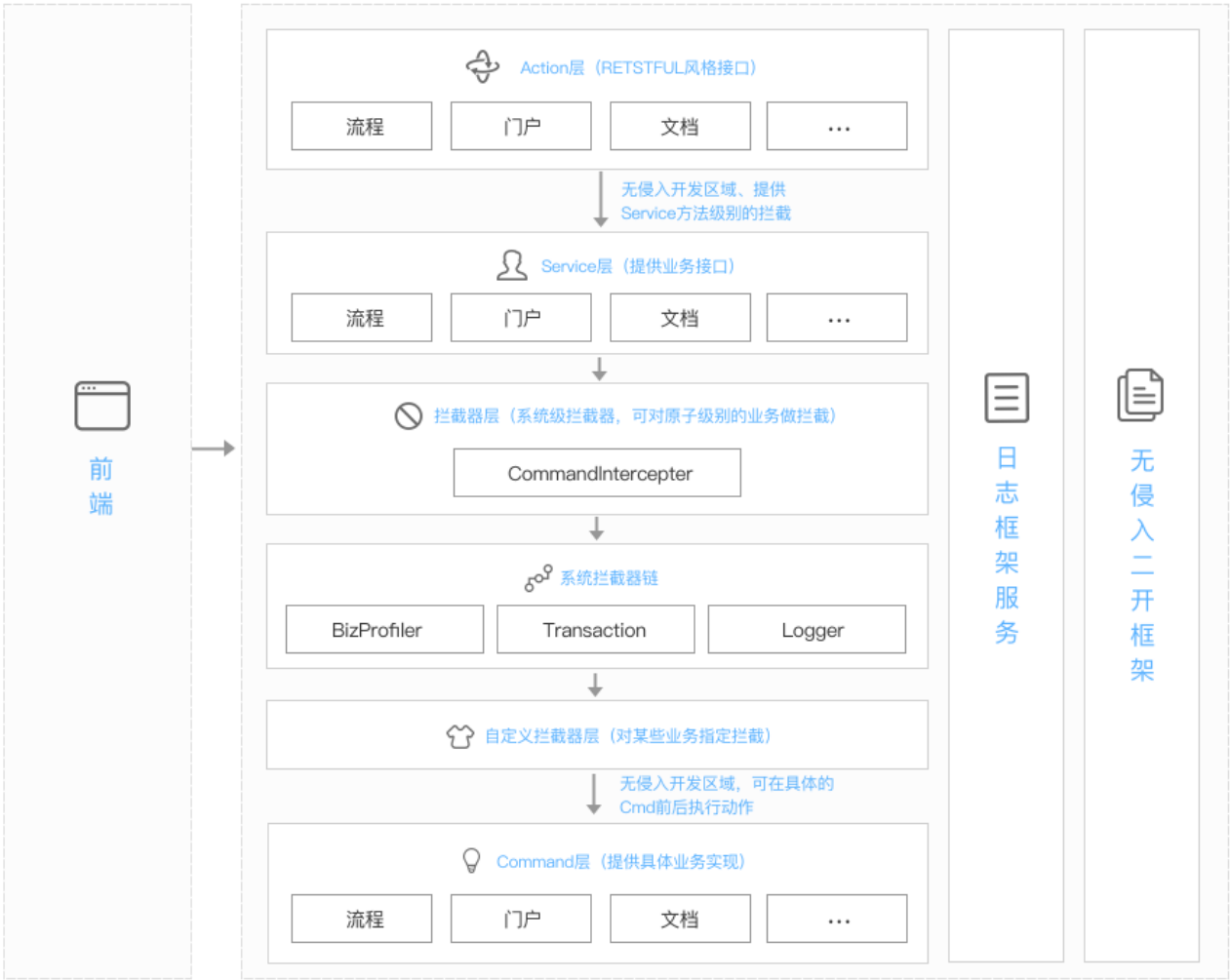
概述

新架构与现行的架构能够很好的结合，前后端分离的同时，对后端增加了分层、AOP、IOC、interceptor的支持。
新架构要求service和Command层必须面向接口编程， 同时通过IOC和命令委托方式进行各层的解耦（具体参加下方示例）；
另外，新架构还提供全局interceptor和局部interceptor、SERVICE-AOP、COMMAND-AOP的支持，可以进行比如日志记录、声明性事务、安全性，和缓存等等功能的实现和无侵入二开。

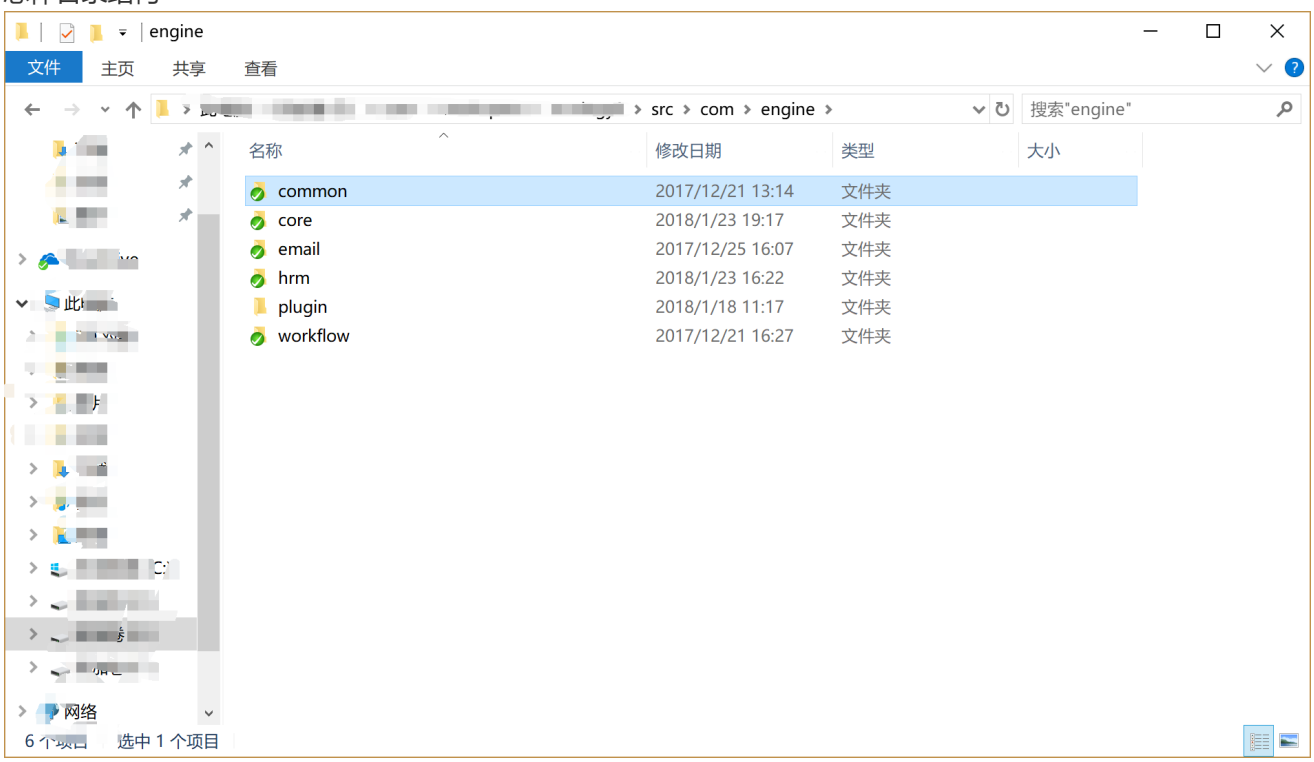
新架构采用命令模式和职责链模式作为基础开发模式，提供一系列的公共实现，用于规范开发过程。

架构图


前端
Action层（RETSTFUL风格接口）
流程
门户
文档
...
无侵入开发区域、提供
Service方法级别的拦截
Service层（提供业务接口）
流程
门户
文档
...
拦截器层（系统级拦截器，可对原子级别的业务做拦截）
CommandIntercepter
系统拦截器链
BizProfiler
Transaction
Logger
自定义拦截器层（对某些业务指定拦截）
无侵入开发区域，可在具体的
Cmd前后执行动作
Command层（提供具体业务实现）
流程
门户
文档
...
日志框架服务
无侵入二开框架

目录结构

总体目录结构

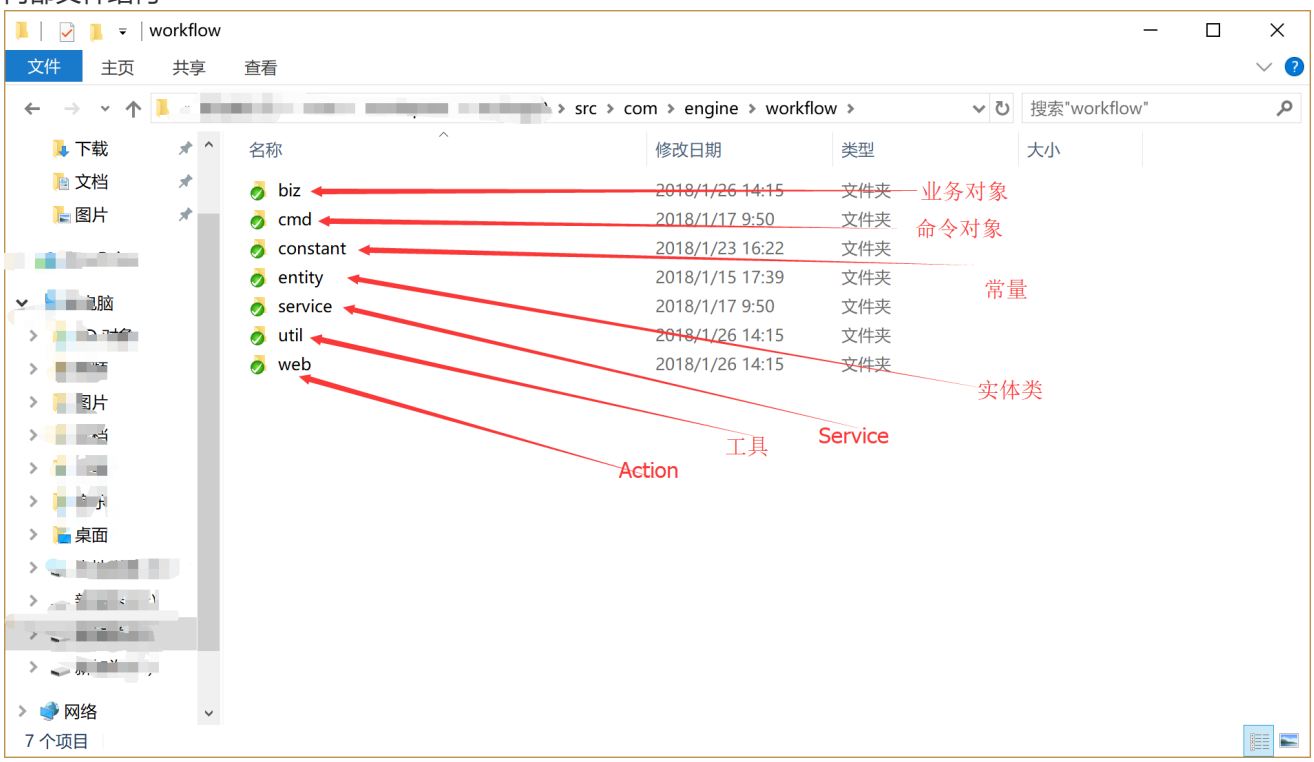

engine
文件
主页
共享
查看
src
com
engine
搜索"engine"
名称
修改日期
类型
大小
common 2017/12/21 13:14 文件夹
core 2018/1/23 19:17 文件夹
email 2017/12/25 16:07 文件夹
hrm 2018/1/23 16:22 文件夹
plugin 2018/1/18 11:17 文件夹
workflow 2017/12/21 16:27 文件夹
网络
选中 1 个项目

目录说明：

目录	说明
command	公共模块
core	核心框架
workflow	流程模块
hrm	人力资源模块
email	邮件模块
...	其他

注意：每一个模块在该目录下都应该有一个对应的目录

内部文件结构



目录说明：

目录	说明
biz	模块内公共业务类目录
constant	常量类目录
cmd	业务Command类目录
entity	实体类目录
service	业务Service服务类目录
util	工具类目录
web	Action类目录

标准开发指南

开发说明

1、建立Action类

Action类需要在web目录下建立，web目录位于模块文件夹下； 每一个功能都应有一个与之对应的Action类，用于对外提供接口服务，Action中不建议包含业务逻辑的处理，业务逻辑请放到Command层（见后文）。 Action类作为边界类，对外提供接口服务， 对内做业务调用，并负责将内部返回的数据做JSON格式的转换，返回给接口的调用者，这里需要注意的是：数据格式的转换尽量的放到Action中， 不要放到业务层（Service、Command层），这样做的好处是有利于维护和二开。

> 示例 1 建立action类， 并配置方法的Path，以及返回数据的类型（注意：类并没有配置Path）

```
package com.engine.workflow.web.workflowPath;

/**
 * 标题设置action
 * */
public class TitleSetListAction {

    private TitleSetService getService(){
        //实例化Service类
        return ServiceUtil.getService(TitleSetServiceImpl.class);
    }

    /**
     * 标题设置
     * */
    @GET
    @Path("/getCondition")
    @Produces(MediaType.TEXT_PLAIN)
    public String getCondition(@Context HttpServletRequest request,@Context HttpServletResponse
response){
```

```
        Map<String,Object> apidatas = new HashMap<String,Object>();
        try{
            User user = HrmUserVarify.getUser(request, response);
            //实例化Service 并调用业务类处理
            apidatas = getService().getTitleSetCondition(ParamUtil.request2Map(request), user);
        }catch(Exception e){
            //异常处理
            e.printStackTrace();
            apidatas.put("api_status", false);
            apidatas.put("api_errormsg", "catch exception : " + e.getMessage());
        }
        //数据转换
        return JSONObject.toJSONString(apidatas);
    }
}
```

com.engine目录是核心业务逻辑类所在目录，不允许直接暴露对外服务接口，对外服务接口请暴露在com.api下（专门提供API服务的目录）。 具体操作是（见示例 1 和 2）： 在com.api.模块.web目录下建立对外接口类，然后通过extends（继承）的方式暴露RESTful服务接口。 示例 1中的Action建立后还不能被前端调用，因为类没有暴露出来，还差一步，见示例 2

> 示例 2 在api目录下暴露接口,直接extens之前写好的action

```
package com.api.workflow.web.workflowPath;

import javax.ws.rs.Path;

/**
 * 标题设置action
 * */
@Path("/workflow/nodeSet/titleSet")
public class TitleSetAction extends TitleSetListAction{
}
```

2、建立Service类

Service类需要在service目录下建立，service目录位于模块文件夹下；

每一个功能都应有一个与之对应的Service接口和impl实现类， 注意：**Service中不允许有具体的业务实现，仅作为服务的提供者，具体业务委托给具体的Command。**

Service接口不需要继承任何类，但需要将其中的服务接口描述清楚

> Service接口示例

```
/**
 * 后台流程监控service
 * @author luosy 2017/12/20
```

```
 * @version 1.0
 *
 */
public interface WorkflowMonitorSettingService {

    /**
     * 获取监控类型sessionkey 列表数据
     * @param params 参数列表
     * @param user 用户
     * @return sessionKey
     */
    public Map<String, Object> getMonitorTypeSessionkey(Map<String, Object> params);

}
```

Service实现类需要实现Service业务接口， 并继承框架中的Service类；

Service实现类需要在impl目录下建立，impl目录位于service文件夹下；

> Service实现类示例

```
/**
 * 后台流程监控service 实现类
 * @author luosy 2017/12/20
 * @version 1.0
 *
 */
public class WorkflowMonitorSettingServiceImpl extends Service implements
WorkflowMonitorSettingService {


    @Override
    public Map<String, Object> getMonitorTypeCondition(Map<String, Object> params,
            String method) {
        return commandExecutor.execute(new GetConditionCmd(params,user,method));
    }
}
```

3、Service对象实例化方式

Action中不能够通过new的形式实例化Service类，需要调用新架构提供的API来实例化

> 示例

```
LoadWorkflowTreeService lwtService = ServiceUtil.getService(LoadWorkflowTreeServiceImpl.class)
```

4、建立Command类

Command采用单一职责原则，一个类，只做一件事。
如果一个类承担的职责过多，就等于把这些职责耦合在一起了。
一个职责的变化可能会削弱或者抑制这个类完成其他职责的能力。
这种耦合会导致脆弱的设计，当发生变化时，设计会遭受到意想不到的破坏。而如果想要避免这种现象的发生，就要尽可能的遵守单一职责原则。此原则的核心就是解耦和增强内聚性，增加复用和可维护性；

Command需要在相应的CMD目录下建立XXXCmd（cmd目录位于模块文件夹下）， 实现Command接口即可(实现execute方法)， 如果需要记录日志，兼容一些公共处理，可以直接继承:

```
com.engine.common.biz.AbstractCommonCommand,
```

该抽象类默认包含user 、params和get set方法， 另外也包含了日志的。

> 示例

```
public class GetSearchConditionCmd extends AbstractCommonCommand<Map<String, Object>>{

    public GetSearchConditionCmd(Map<String, Object> params, User user) {
        this.user = user;
        this.params = params;
    }

    @Override
    public Map<String, Object> execute(CommandContext commandContext) {
        return result;
    }
}
```

5、规范

1. 入参最好使用Map或者Entity对象，尽可能的与Http对象解耦
2. 出参最好使用Map或者Entity对象，不要直接返回单一数据或者String类型的JSON格式数据；
3. 不同功能的服务封装在不同的Service中，用户可以非常清晰地使用特定的Service
4. Service中定义的各个方法都有相应的命令对象（XXCmd）
5. 日志记录尽可能的对本次修改的情况进行详细的记录，比如对一个人员的信息进行了修改， 日志中，要能够体现出修改的项目， 如果是重要功能，要在之前的基础上增加所修改项目的原值和新值，以便能够追溯到修改的内容。
6. Command按照功能进行分类，不要按照Command的类型（增删改查）进行分类，这样即使是新人， 也能够很快的定位到相关的类；
7. Command的设计要符合单一职责原则，不要做过多的事，如果不能把握住这个度， 请按照前端功能接口进行设计， 一个接口， 对应一个Command；
8. Command中尽可能不要直接调用Service， 如果需要是公共类，请放到Biz目录下；
9. Service必须有一个接口和一个实现类；
10. Service中不能包含具体的业务逻辑， 业务逻辑委托给具体的Command类；
11. Command中不允许直接调用Service方法， 如需要可以调用Biz包下的类；
12. Command中的参数必须要包含Getting和Setting方法，方便无侵入二开获取相关的参数；
13. 前端展示的项和内容不要写死， 一定要根据后端接口数据进行动态展现，这样方便二开，仅修改后端接口即可；

异常处理

业务Command对象中不允许抛出非运行时异常，如需要对异常处理， 请先捕捉，然后转成ECException进行抛出，并对异常发生的情况添加大家可理解的说明。

> 示例：

```
try {
    //TODO
} catch (Exception e) {
    throw new ECException(command.getClass().getName() + "执行过程中异常", e);
}
```

日志处理

```
com.engine.common.biz.AbstractCommonCommand
```

已经包含了日志接口， 大家只需要实现对应方法， 作成日志对象返回即可， 该抽象类包含单日志记录和批量日志记录两个方法，大家需要根据自身情况进行选择性实现。

批量日志记录方法:

```
public List<BizLogContext> getLogContexts()
```

为了方便记录批量 和 更新操作，增加了一个日志处理类，仅需要少量的代码即可完成日志的记录，且可做到与业务代码解耦。 该类的实现方式是在业务更新前后去DB中查询，做数据做对比，区分出新建、更新、删除动作，其对性能可能会有稍微的影响（具体视功能SQL的执行效率而定），对于性能有严苛要求的功能，请酌情使用。

> 示例 1 使用SimpleBizLogger进行日志主从日志记录

```
package com.engine.workflow.cmd.workflowPath.node.operatorSetting;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import weaver.conn.RecordSetTrans;
import weaver.general.Util;
import weaver.hrm.HrmUserVarify;
import weaver.hrm.User;
import weaver.systeminfo.SystemEnv;
import weaver.workflow.workflow.WfRightManager;
import weaver.workflow.workflow.WorkflowComInfo;

import com.engine.common.biz.AbstractCommonCommand;
import com.engine.common.biz.SimpleBizLogger;
import com.engine.common.biz.SimpleBizLogger.SubLogInfo;
import com.engine.common.constant.BizLogSmallType4Workflow;
```

```
import com.engine.common.constant.BizLogType;
import com.engine.common.entity.BizLogContext;
import com.engine.core.interceptor.CommandContext;
import com.engine.workflow.biz.NodeOperatorBiz;
import com.engine.workflow.biz.nodeOperatorItem.AbstractItem;
import com.engine.workflow.constant.WfFunctionAuthority;
import com.engine.workflow.entity.node.OperatorTypeEntity;

public class DoSaveOperatorGroupInfoCmd extends AbstractCommonCommand<Map<String, Object>>{
    //简单日志记录对象
    private SimpleBizLogger logger;

    public DoSaveOperatorGroupInfoCmd(Map<String, Object> params,User user) {
        this.params = params;
        this.user = user;
        //初始化简单日志对象
        this.logger = new SimpleBizLogger();
    }

    @Override
    public BizLogContext getLogContext() {
        return null;
    }

    /**
     * 批量记录日志
     */
    @Override
    public List<BizLogContext> getLogContexts() {
        //计算修改记录并返回， 注意， 必须在业务代码执行完毕后调用，否则本次操作记录不会被记录
        return logger.getBizLogContexts();
    }

    @Override
    public Map<String, Object> execute(CommandContext commandContext) {
        //处理日志
        this.bofore();
        return doSave();
    }

    /**
     * 处理日志
     */
    public void bofore(){
        BizLogContext bizLogContext = new BizLogContext();
        bizLogContext.setLogType(BizLogType.WORKFLOW_ENGINE);//模块类型

bizLogContext.setBelongType(BizLogSmallType4Workflow.WORKFLOW_ENGINE_PATH_PATHSET_NODESET);//所
属大类型
        bizLogContext.setBelongTypeTargetId(Util.null2String(params.get("nodeid")));//所属大类型
id
        bizLogContext.setBelongTypeTargetName(Util.null2String(params.get("nodename")));//所属大
类型名称
```

```
        bizLogContext.setLogSmallType(BizLogSmallType4Workflow.WORKFLOW_ENGINE_OPERATORSET);//当前小类型
        logger.setUser(user);//当前操作人
        logger.setParams(params);//request请求参数(request2Map)
        String mainSql = "select id,groupname from workflow_nodegroup where nodeid = " + Util.getIntValue(Util.null2String(params.get("nodeid")));
        logger.setMainSql(mainSql);//主表sql
        logger.setMainPrimarykey("id");//主日志表唯一key
        logger.setMainTargetNameColumn("groupname");//当前targetName对应的列（对应日志中的对象名）

        SubLogInfo subLogInfo = logger.getNewSubLogInfo();
        String subSql = "select g.*,n.groupname from workflow_groupdetail g ,workflow_nodegroup n where g.groupid = n.id and g.groupid = " + Util.getIntValue(Util.null2String(params.get("groupid"))) + " order by g.id";
        subLogInfo.setSubSql(subSql,"id");
        subLogInfo.setSubTargetNameColumn("groupname");
        subLogInfo.setGroupId("0");  //所属分组， 按照groupid排序显示在详情中， 不设置默认按照add的顺序。
        subLogInfo.setSubGroupNameLabel(234212); //在详情中显示的分组名称，不设置默认显示明细x
        logger.addSubLogInfo(subLogInfo);
        //开始记录
        logger.before(bizLogContext);
    }

    public Map<String , Object> doSave(){
        //...业务代码
        return apidatas;
    }
}
```

> 示例 2 使用SimpleBizLogger进行主日志批量记录

```
package com.engine.workflow.cmd.workflowPath.node;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import weaver.conn.RecordSet;
import weaver.general.Util;
import weaver.hrm.User;
import weaver.rdeploy.workflow.WorkflowInitialization;
import weaver.systeminfo.SysMaintenanceLog;
import weaver.systeminfo.SystemEnv;
import weaver.workflow.workflow.WFNodeMainManager;
import weaver.workflow.workflow.WorkflowComInfo;
import weaver.workflow.workflow.WorkflowVersion;

import com.engine.common.biz.AbstractCommonCommand;
import com.engine.common.biz.SimpleBizLogger;
import com.engine.common.constant.BizLogSmallType4Workflow;
```

```
import com.engine.common.constant.BizLogType;
import com.engine.common.constant.ParamConstant;
import com.engine.common.entity.BizLogContext;
import com.engine.core.interceptor.CommandContext;

public class DoSaveNodeCmd extends AbstractCommonCommand<Map<String, Object>>{

    private SimpleBizLogger logger;


    public DoSaveNodeCmd(Map<String, Object> params,User user) {
        this.params = params;
        this.user = user;
        this.logger = new SimpleBizLogger();
    }
    public DoSaveNodeCmd() {
    }

    @Override
    public BizLogContext getLogContext() {
        return null;
    }
    @Override
    public List<BizLogContext> getLogContexts() {
        return logger.getBizLogContexts();
    }

    public void bofore(){
        WorkflowComInfo WorkflowComInfo = new WorkflowComInfo();
        BizLogContext bizLogContext = new BizLogContext();
        bizLogContext.setLogType(BizLogType.WORKFLOW_ENGINE);
        bizLogContext.setBelongType(BizLogSmallType4Workflow.WORKFLOW_ENGINE_PATH);//所属大类型
        bizLogContext.setBelongTypeTargetId(Util.null2String(params.get("workflowId")));//所属大
类型id

bizLogContext.setBelongTypeTargetName(WorkflowComInfo.getWorkflowname(Util.null2String(params.ge
t("workflowId"))));//所属大名称

bizLogContext.setLogSmallType(BizLogSmallType4Workflow.WORKFLOW_ENGINE_PATH_PATHSET_NODESET);//
当前小类型
        logger.setUser(user);//当前操作人
        logger.setParams(params);//request请求参数
        String mainSql  ="select f.nodeid, n.nodename,f.nodetype,n.nodeattribute,n.passnum from
workflow_flownode f inner join workflow_nodebase n on f.nodeid = n.id where f.workflowid = " +
Util.getIntValue(Util.null2String(params.get("workflowId")));
        logger.setMainSql(mainSql,"nodeid");
        logger.setMainTargetNameColumn("nodename");
        logger.setMainTargetNameMethod(this.getClass().getName() + ".getMethod",
"column:nodename+column:nodeattribute+" + user.getLanguage());
        logger.before(bizLogContext);
    }

    public String getMethod(String targetid,String para){
```

```
        return "1";
    }

    @Override
    public Map<String, Object> execute(CommandContext commandContext) {
        //日志记录
        this.bofore();
        return doSaveNodeInfo();
    }

    public Map<String, Object> doSaveNodeInfo(){
        //...业务代码
        return apidatas;
    }
}
```

> 示例 3 使用SimpleBizLogger进行单个主日志记录

```
package com.engine.workflow.cmd.workflowPath.node;

import java.util.HashMap;
import java.util.Map;

import weaver.conn.RecordSet;
import weaver.general.Util;
import weaver.hrm.HrmUserVarify;
import weaver.hrm.User;
import weaver.workflow.workflow.WfRightManager;
import weaver.workflow.workflow.WorkflowComInfo;

import com.engine.common.biz.AbstractCommonCommand;
import com.engine.common.biz.SimpleBizLogger;
import com.engine.common.constant.BizLogSmallType4Workflow;
import com.engine.common.constant.BizLogType;
import com.engine.common.entity.BizLogContext;
import com.engine.core.interceptor.CommandContext;
import com.engine.workflow.constant.WfFunctionAuthority;

public class DoUpdateNodeNameCmd extends AbstractCommonCommand<Map<String, Object>>{

    private SimpleBizLogger logger;

    public DoUpdateNodeNameCmd(Map<String, Object> params,User user) {
        this.params = params;
        this.user = user;
        this.logger = new SimpleBizLogger();
    }
    public DoUpdateNodeNameCmd() {

    }
```

```
    @Override
    public Map<String, Object> execute(CommandContext commandContext) {
        this.bofore();
        return updateNodeName();
    }

    public void bofore(){
        BizLogContext bizLogContext = new BizLogContext();
        WorkflowComInfo workflowComInfo = new WorkflowComInfo();
        bizLogContext.setLogType(BizLogType.WORKFLOW_ENGINE);
        bizLogContext.setBelongType(BizLogSmallType4Workflow.WORKFLOW_ENGINE_PATH);//所属大类型
        bizLogContext.setBelongTypeTargetId(Util.null2String(params.get("workflowId")));//所属大
类型id

bizLogContext.setBelongTypeTargetName(workflowComInfo.getWorkflowname(Util.null2String(params.ge
t("workflowId"))));//所属大名称

bizLogContext.setLogSmallType(BizLogSmallType4Workflow.WORKFLOW_ENGINE_PATH_PATHSET_NODESET);//
当前小类型
        logger.setUser(user);//当前操作人
        logger.setParams(params);//request请求参数
        String mainSql = "select b.id,b.nodename from workflow_flownode a left join
workflow_nodebase b on a.nodeid = b.id where a.workflowid =
"+Util.null2String(params.get("workflowId"))+" and a.nodeid = " +
Util.getIntValue(Util.null2String(params.get("nodeid")));
        logger.setMainSql(mainSql, "id");
        logger.setMainTargetNameColumn("nodename");
        logger.before(bizLogContext);
    }

    @Override
    public BizLogContext getLogContext() {
        return logger.getBizLogContext();
    }


    public  Map<String, Object> updateNodeName() {
        ...业务代码
    }
}
```

以下是针对有性能要求的功能记录日志的方式（不推荐，业务耦合比较严重）：

重要说明：**getLogContext()方法调用发生在command.execute方法之后，是系统自动调用执行，所以，在execute执行时候就考虑日志的内容，比如删除对象的名称和内容， 修改时修改项前后的值。**

具体操作步骤：请在update之前先将要变更的数据一次性查询出来，放入List<Map<String, Object>>中（批量操作，单个的可以不用List），然后在更新操作中记录需要更新的每一条的数据的键值对（Map<String, Object>）
待数据更新完成后， 调用日志工具类将本次更新中没有变更的字段过滤掉：
//m1 为老的数据键值对 ,m2为新的数据键值对
LogUtil.removeIntersectionEntry(map1, map2);

//m1 m2此时 已经不存在交集， 保留下来都是有变更的字段，
//设置到日志对象中
logContext.setOldValues(m1);
logContext.setNewValues(m2);

注意：
1、新增的记录不需要setOldValues，但需要setNewValues
2、删除的记录不需要setNewValus，但需要setOldValues
3、如果当前功能属于二级功能，需要在外层查询到内层的日志记录， 请设置外层类型、外层类型对象的id和显示名： setBelongType(BizLogSmallType)、 setBelongTypeTargetId(String)和setBelongTypeTargetName(String)

批量记录日志时， 需要进行关联（列表显示）和明细日志进行关联，以便于前台查看(见下方示例3)

> 示例 1 execute中设置一部分必要的信息，防止execute执行完成后， 数据库中无法查询到修改前的数据。（仅设置execute方法执行后获取不到的信息） 其他部分信息放到getLogContext中设置。(建议使用示例4、5、6的方式)

```
public class GetSearchConditionCmd extends AbstractCommonCommand<Map<String, Object>>{
    protected BizLogContext bizLogContext;
    public GetSearchConditionCmd(Map<String, Object> params, User user) {
        this.user = user;
        this.params = params;
    }

    @Override
    public Map<String, Object> execute(CommandContext commandContext) {
        bizLogContext = new BizLogContext();
        bizLogContext.setTargetId(targetId);
        bizLogContext.setTargetName(targetName);
        //execute中设置一部分
        ...

        //
        return result;
    }

    @Override
    public BizLogContext getLogContext() {
        //这里set一部分
        bizLogContext.setDateObject(new Date());
        bizLogContext.setUserid(user.getUID());
        bizLogContext.setUsertype(user.getType()

        bizLogContext.setLogType(BizLogType.WORKFLOW_ENGINE);
        //bizLogContext.setTargetSmallType(BizLogTargetSmallType);
        bizLogContext.setOperateType(BizLogOperateType.DELETE);
        bizLogContext.setDesc(descStr);
        bizLogContext.setParams(params);
        //设置所属日志对象的id， 用于外层日志的查询
        bizLogContext.setBelongTypeTargetId(workflowid + "");
        //设置所属日志对象的类型， 用于外层日志的查询
        bizLogContext.setBelongType(BizLogSmallType4Workflow.WORKFLOW_ENGINE_PATH);
```

```
        //设置所属日志对象的名称，用于当无法根据名称查看时的显示
        bizLogContext.setBelongTypeTargetName(comInfo.getWorkflowname(workflowid + ""));
        return this.bizLogContext;
    }
}
```

> 示例 2(建议使用示例4、5、6的方式)

```
public class GetSearchConditionCmd extends AbstractCommonCommand<Map<String, Object>>{

    public GetSearchConditionCmd(Map<String, Object> params, User user) {
        this.user = user;
        this.params = params;
    }

    @Override
    public Map<String, Object> execute(CommandContext commandContext) {

        return result;
    }

    @Override
    public BizLogContext getLogContext() {
        bizLogContext = new BizLogContext();
        bizLogContext.setDateObject(new Date());
        bizLogContext.setUserid(user.getUID());
        bizLogContext.setUsertype(user.getType());
        bizLogContext.setTargetId(targetId);
        bizLogContext.setTargetName(targetName);
        bizLogContext.setLogType(BizLogType.WORKFLOW_ENGINE);
        //bizLogContext.setTargetSmallType(BizLogTargetSmallType);
        bizLogContext.setOperateType(BizLogOperateType.DELETE);
        bizLogContext.setDesc(descStr);
        bizLogContext.setParams(params);
        //设置所属日志对象的id， 用于外层日志的查询
        bizLogContext.setBelongTypeTargetId(workflowid + "");
        //设置所属日志对象的类型， 用于外层日志的查询
        bizLogContext.setBelongType(BizLogSmallType4Workflow.WORKFLOW_ENGINE_PATH);
        //设置所属日志对象的名称，用于当无法根据名称查看时的显示
        bizLogContext.setBelongTypeTargetName(comInfo.getWorkflowname(workflowid + ""));
        return bizLogContext;
    }
}
```

> 示例 3 批量记录日志(建议使用示例4、5、6的方式)

```
public class GetSearchConditionCmd extends AbstractCommonCommand<Map<String, Object>>{
```

```
    public GetSearchConditionCmd(Map<String, Object> params, User user) {
        this.user = user;
        this.params = params;
    }

    @Override
    public Map<String, Object> execute(CommandContext commandContext) {

        return result;
    }

    @Override
    public List<BizLogContext> getLogContexts() {
        List<BizLogContext> logs = new ArrayList<BizLogContext>();


        BizLogContext mainBizLogContext = new BizLogContext();
        //主日志记录id， 批量记录日志时仅显示主记录，不显示明细记录， 方便用户查看，当查看修改详情时，再显示具体的明细修改记录
        String mainLogid = mainBizLogContext.createMainid();

        bizLogContext.setDateObject(new Date());
        bizLogContext.setUserid(user.getUID());
        ...
        bizLogContext.setParams(params);
        //设置为主记录， 并为其生成唯一ID
        bizLogContext.setMainId(mainLogid);

        for (...) {
            BizLogContext detailBizLogContext = new BizLogContext();
            bizLogContext.setDateObject(new Date());
            bizLogContext.setUserid(user.getUID());
            ...

            //以下几个设置， 作为明细日志必须设置!

            //设置为明细记录， 不在日志列表中显示
            bizLogContext.setDetail(true);
            //当一个页面存在多个明细表时， 需要将将同一个明细表的groupid设置为同一个;
            bizLogContext.setGroupId(1);
            //设置当前所属明细表的名称label， 用于显示日志修改详情时的title
            bizLogContext.setGroupNameLabel(123);
            //设置当前明细所属的主日志记录id
            bizLogContext.setBelongMainId(mainLogid);
        }
        ....
        return logs;
    }
}
```

> 日志对象

```
package com.engine.common.entity;

import com.alibaba.fastjson.JSONObject;
import com.engine.common.constant.BizLogSmallType;
import com.engine.common.constant.BizLogOperateType;
import com.engine.common.constant.BizLogType;

import java.io.Serializable;
import java.util.Date;
import java.util.Map;
import java.util.UUID;

/**
 * Created by wcc on 2017/12/11.
 */
public class BizLogContext implements Serializable {
    /**
     * 操作日期
     */
    protected String date;

    /**
     * 时间
     */
    protected String time;

    /**
     * 日期事件对象
     */
    protected Date dateObject;

    /**
     * 操作人
     */
    protected int userid;

    /**
     * 操作人类型
     */
    protected int usertype;

    /**
     * 目标对象id
     */
    protected String targetId;

    /**
     * 目标对象名称（用于显示）
     */
    protected String targetName;

    /**
     * 所属类型（用于查询，以及显示）
```

```
     */
    protected BizLogSmallType belongType;

    /**
     * 所属大类型对象的id
     */
    protected String belongTypeTargetId;

    /**
     * 所属大类型对象的显示名
     */
    protected String belongTypeTargetName;

    /**
     * 目标对象类型（大分类）
     */
    protected BizLogType logType;

    /**
     * 目标对象类型（小分类）
     */
    protected BizLogSmallType logSmallType;

    /**
     * 操作类型（增删改查）
     */
    protected BizLogOperateType operateType;

    /**
     * 操作IP
     */
    protected String clientIp;

    /**
     * 修改前的值
     */
    protected Map<String, Object> oldValues;

    /**
     * 修改后的值
     */
    protected Map<String, Object> newValues;

    /**
     * 操作详细说明
     */
    protected String desc;

    /**
     * 涉及的相关参数
     */
    protected Map<String, Object> params;
```

```
    /**
     * 主日志
     */
    protected String mainId;

    /**
     * 从表日志
     */
    protected String belongMainId;

    /**
     * 分组
     */
    protected String groupId;

    /**
     * 分组
     */
    protected boolean isDetail;

    /**
     * 分组标题
     */
    protected int groupNameLabel;


    public BizLogContext() {
    }
}
```

日志类型分为大类型和小类型， 大类型为模块，其定义在com.engine.common.constant.BizLogType中, 如果其中没有你模块的信息， 请自己添加一下：

```
public enum BizLogType {

    WORKFLOW(1, 2118),
    WORKFLOW_ENGINE(2, 33636),
    HRM(3, 179),
    HRM_ENGINE(4, 179),
    PORTAL(5, 582),
    PORTAL_ENGINE(6, 33637),
    DOC(7, 2115),
    DOC_ENGINE(8, 33638),
    MEETING(9, 34076),
    MEETING_ENGINE(10, 34076),
    WKP(11, 83995),
    WKP_ENGINE(12, 83995),
    LANG(13,  16066),
    LANG_ENGINE(14, 84641),
    INTEGRATION_ENGINE(15, 32269),
```

```
    FULLSEARCH(16, 31953),
    ODOC(16, 27618),
    ODOC_ENGINE(17, 27618),
    SYSTEM(18,16686),
    SYSTEM_ENGINE(19,16686),
    ;

    /**
     * code
     */
    protected int code;

    protected int labelId;

    public int getCode() {
        return code;
    }

    public int getLableId() {
        return labelId;
    }

    BizLogType(int code, int labelId) {
        this.code = code;
        this.labelId = labelId;
    }

}
```

小类型所在类是一个接口， 所有的模块具体的小类型都需要继承这个接口，然后将类型定义在这个类中

> 小类型接口类

```
package com.engine.common.constant;

public interface BizLogSmallType {

    int getCode();

    int getLableId();

    BizLogType getBizLogType();
}
```

> 示例， 流程小类型， 请各模块参考

```
package com.engine.common.constant

public enum BizLogSmallType4Workflow implements BizLogSmallType {
    WORKFLOW_ENGINE_TYPE(1, 33806),
    WORKFLOW_ENGINE_PATH(2, 33657),
```

```
    WORKFLOW_ENGINE_MONITORSET(3, 17989),
    WORKFLOW_ENGINE_MONITORSET_TYPE(4, 2239),
    WORKFLOW_ENGINE_PATH_RULE(5, 32481),
    WORKFLOW_ENGINE_REPORTSET(6, 33665),
    WORKFLOW_ENGINE_PATH_TRANSFER(7, 33660),
    WORKFLOW_ENGINE_CUSTOMQUERYSET(8, 20785),
    WORKFLOW_ENGINE_CUSTOMQUERYSET_TYPE(9, 23799),
    WORKFLOW_ENGINE_PATHIMPORT(10, 33659),
    WORKFLOW_ENGINE_CODEMAINTENANCE_STARTCODE(11, 20578),
    WORKFLOW_ENGINE_CODEMAINTENANCE_RESERVECODE(12, 22779),
    WORKFLOW_ENGINE_REPORTSET_REPORTTYPESET(13, 33664),
    WORKFLOW_ENGINE_PATH_PATHSET_NODESET(14, 126552),
    WORKFLOW_ENGINE_REPORTSET_REPORTSHARE(15,33666),
    WORKFLOW_ENGINE_PATH_PATHSET_FUNCTIONMANAGER(16, 18361),
    WORKFLOW_ENGINE_NODELINK(17, 126553),
    WORKFLOW_ENGINE_PATH_PATHSET_WORKFLOWPLAN(18, 18812),
    WORKFLOW_ENGINE_PATH_PATHSET_NODEFIELD(19, 15615),
    WORKFLOW_ENGINE_REPORTSET_COMPETENCESET(20,382890),
    WORKFLOW_ENGINE_FORMSET_FORM(21, 33655),
    WORKFLOW_ENGINE_PATH_PATHSET_OPERATIONMENU(22, 16380),
    WORKFLOW_ENGINE_OPERATORSET(23, 124954),
    WORKFLOW_ENGINE_SUPERVISESET(24, 21220),
    WORKFLOW_ENGINE_WORKFLOW_TO_DOC(25, 22231),
    WORKFLOW_ENGINE_FIELD(26, 382028),
    WORKFLOW_ENGINE_SUBWORKFLOWSET(27, 21584),
    WORKFLOW_ENGINE_WORKFLOW_TO_WORKPLAN(28, 24086),
    WORKFLOW_ENGINE_ROW_RULE(29, 18368),
    WORKFLOW_ENGINE_COL_RULE(30, 18369),
    WORKFLOW_ENGINE_PATH_PATHSET_BASESET(31, 82751),
    WORKFLOW_ENGINE_PATH_PATHSET_LINKAGEVIEWATTR(32, 21684),
    WORKFLOW_ENGINE_DATAINPUT(33,21848), //字段联动
    WORKFLOW_ENGINE_PATH_BROWSERDATADEFINITION(34,32752),//浏览数据定义
    WORKFLOW_ENGINE_PATH_WORKFLOWMAINTAINRIGHT(35,33805),//路径维护权限
    WORKFLOW_ENGINE_PATH_PREADDINOPERATE(36,18009),//节点前附加操作
    WORKFLOW_ENGINE_PATH_ADDINOPERATE(37,18010),//节点前附加操作
    WORKFLOW_ENGINE_PATH_LINKOPERATE(38,15610),//出口附加规则
    ;

    protected int code;

    protected int labelId;

    private BizLogType bizLogType = BizLogType.WORKFLOW_ENGINE;

    BizLogSmallType4Workflow(int code, int labelId) {
        this.code = code;
        this.labelId = labelId;
    }

    @Override
    public int getCode() {
        return this.code;
    }
```

```
    @Override
    public int getLableId() {
        return this.labelId;
    }

    @Override
    public BizLogType getBizLogType() {
        return bizLogType;
    }
}
```

无侵入开发指南

E9已经支持COMMAND类级别和SERVICE-METHOD方法级别的动态代理，以支撑在不修改标准代码的前提下完成个性化的开发。 两种方式都有各自适用的场景，大家可根据自身需求进行选择。

动态代理类必须位于src目录下的：

```
com.customization.个性化功能模块名称_编号
```

目录下， 其中编号后续会提供申请生成页面， 在该目录下可进行service、cmd目录的划分

```
//存放Service代理类
plugin.模块名称_个性化功能的简称.service
//存放command代理类
plugin.模块名称_个性化功能的简称.cmd
```

类级别（Command）动态代理

该机制提供对COMMAND类级的动态代理，以支撑在不修改标准代码的前提下完成个性化的开发。此种方式可以在标准COMMAND执行前做参数的预处理加工、持久化等， 在COMMAND执行后做返回值的二次处理、加工，持久化等。

代理类需要继承类：

```
com.engine.core.interceptor.AbstractCommandProxy
```

并增加注解，指定对哪一个command做动态代理，并对本代理类做一个功能说明

```
@CommandDynamicProxy(target = 被代理CMD类对象, desc="功能描述，必须要有")
```

在内部execute方法中，必须显示的调用nextExecute方法，使代理链能够顺序执行，得到标准业务类返回的结果集。

```
Map<String, Object> result = nextExecute(targetCommand);
```

> 示例代码

```
@CommandDynamicProxy(target = DoSaveCmd.class, desc="附加在类型保存上的示例代理程序")
public class CustomDoSaveCmd extends AbstractCommandProxy<Map<String,Object>> {
    @Override
    public Map<String, Object> execute(Command<Map<String, Object>> targetCommand) {
        System.out.println(getClass().getName() + "command 执行之前做一些事");

        //获取到被代理对象
        DoSaveCmd doSaveCmd = (DoSaveCmd)targetCommand;
        //获取被代理对象的参数
        Map<String, Object> params = doSaveCmd.getParams();
        //对参数做预处理
        //TODO
        //参数回写
        doSaveCmd.setParams(params);
        //执行标准的业务处理
        Map<String, Object> result = nextExecute(targetCommand);

        //对返回值做加工处理
        result.put("a", "b");

        System.out.println(getClass().getName() + "command 执行之后做一些事");

        return result;
    }
}
```

方法级（Service-Method）动态代理

在E9中，同一个功能的多个接口共存于一个Service类中，使用Service方法级别代理可以使一个代理类完成对多个接口的代理， 相比于类级别代理，可大大减少了类的数量，降低开发难度，提高可维护性。

代理类需要继承类：

```
com.engine.core.impl.aop.AbstractServiceProxy
```

并 实现需要拦截的Service接口， 比如 要对WorkflowTypeService做代理拦截， 则必须要实现该接口：

```
com.engine.workflow.service.WorkflowTypeService
```

增加类注解，指定对哪一个Service做动态代理，并对本代理类做一个功能说明

```
@ServiceDynamicProxy(target = WorkflowTypeServiceImpl.class, desc="为流程类型增加一个图标字段")
```

对需要代理的Service方法增加注解，并增加说明，此处不需要指定代理的方法， 代理框架会自动抓取你所实现的方法

```
@ServiceMethodDynamicProxy(desc="保存时， 增加一些日志输入")
```

在内部方法中，必须显示的调用executeMethod方法，并将当前方法接收的参数按照顺序传入，使代理链能够顺序执行，得到标准业务类返回的结果集。

```
Map<String, Object> result = (Map<String, Object>)executeMethod(params, user);
```

此处请一定注意：executeMethod的方法的参数类型与个数与被代理方法完全一致， 必须按照顺序全部传入，否则将导致程序异常！

> 示例代码

```
@ServiceDynamicProxy(target = WorkflowTypeServiceImpl.class, desc="为流程类型增加一个图标字段")
public class CustomWorkflowTypeService extends AbstractServiceProxy implements
WorkflowTypeService  {

    /**
     * 重写保存方法， 在保存完成之后保存自定义信息
     * @param params
     * @param user
     * @return
     */
    @Override
    @ServiceMethodDynamicProxy(desc="保存时， 增加一些日志输入")
    public Map<String, Object> doSaveOperation(Map<String, Object> params, User user) {
        System.out.println(getClass().getName() + " 在数据保存做一些事111。。。。");

        //对参数做预处理
        //TODO

        //调用被代理类方法
        Map<String, Object> result = (Map<String, Object>)executeMethod(params, user);

        //对结果做二次处理加工
        //TODO
        System.out.println(getClass().getName() + " 在数据保存之后做一些事1111。。。。");
        return result;
    }

    @Override
    public Map<String, Object> doDeleteOperation(Map<String, Object> params, User user) {
        return null;
    }

    @Override
    public Map<String, Object> getConditionInfo(Map<String, Object> params, User user) {
        return null;
    }

    @Override
    public Map<String, Object> getSessionKey(Map<String, Object> params, User user) {
```

```
        return null;
    }
}
```

不需要代理的方法，为其提供空实现即可，不会影响标准业务逻辑。（不要增加代理注解）