# 13.业务与流程的绑定

UFLO是一套基于JAVA的流程引擎，它即可以用在基于J2EE的B/S系统之上，也可以用于基于J2EE的C/S系统之中。所以UFLO在设计之初就不与任何UI表现层绑定，它所要做的就是提供一系列的可供外部调用的API接口，对于我们的业务系统而言，可以调用UFLO中提供的API，实现业务流程的开启、任务的开始与完成等操作。

我们知道，UFLO中有两个子项目，分别是uflo-core与uflo-console。uflo-core提供了流程运行的核心环境，负责整个流程生命周期的管理工作；uflo-console则是一个基于WEB表现层的分支项目，这个项目中提供了基于网页的流程模版控制与测试中心、提供了用户待办页面及节假日配置等流程引擎运行的辅助模块；通过调用uflo-core模块中的ProcessService与其中的TaskService就可实现在我们的业务系统与流程引擎的交互。下面是开启一个流程实例操作的代码片段。

开启流程示例代码

```
import javax.annotation.Resource;
import com.bstek.bdf2.core.context.ContextHolder;
import com.bstek.uflo.service.ProcessService;
import com.bstek.uflo.service.StartProcessInfo;
public class BusinessTest {
 @Resource(name=ProcessService.BEAN_ID)
 private ProcessService processService;
 public void saveDataAndStartProcess(BusinessData data){
  String primaryKeyValue=saveBusinessData(data);
  StartProcessInfo info=new StartProcessInfo(ContextHolder.getLoginUserName());
  info.setBusinessId(primaryKeyValue);
  processService.startProcessById(101, info);
 }
}
```

从上面的代码中可以看到，在开始流程之前，我们首先将提交上业的BusinessData这个业务数据对象保存，保存后产生该对象的主键值primary KeyValue，接下来创建一个StartProcessInfo
对象，将业务对象主键值赋给这个对象，并设置流程开启人，最后调用ProcessClient接口的ProcessInstance startProcessById(long processId,StartProcessInfo startProcessInfo)方法开启一个ID为101的流程模版对应的流程实例。在上面的代码当中，ProcessService我们通过Spring注入到当前类中，ProcessService配置在Spring中的Bean的ID就是ProcessService.BEAN_ID的常量值，所以这里可以直接使用。

ProcessService是我们操作所有与流程相关动作的入口，该接口详细信息如下：

ProcessService接口源码

```
package com.bstek.uflo.service;
import java.io.InputStream;
import java.util.List;
import java.util.Map;
import java.util.zip.ZipInputStream;
import com.bstek.uflo.model.ProcessDefinition;
import com.bstek.uflo.model.ProcessInstance;
import com.bstek.uflo.model.variable.Variable;
import com.bstek.uflo.query.ProcessInstanceQuery;
import com.bstek.uflo.query.ProcessQuery;
import com.bstek.uflo.query.ProcessVariableQuery;
/**
 * @author Jacky.gao
 * @since 2013年7月29日
 */
public interface ProcessService {
```

```
public static final String BEAN_ID="uflo.processService";
/**
 * 根据流程模版ID，返回流程模版对象
 * @param processId 流程模版ID
 * @return 返回流程模版对象
 */
ProcessDefinition getProcessById(long processId);
/**
 * 根据流程模版Key，返回流程模版对象
 * @param key 流程模版Key
 * @return 返回流程模版对象
 */
ProcessDefinition getProcessByKey(String key);
/**
 * 根据流程模版的名称，返回与该名字匹配最新发布的流程模版对象
 * @param processName 流程模版名称
 * @return 返回流程模版对象
 */
ProcessDefinition getProcessByName(String processName);
/**
 * 根据流程模版的名称及分类ID，返回与该名字匹配最新发布的流程模版对象
 * @param processName 流程模版名称
 * @param categoryId 分类ID
 * @return 返回流程模版对象
 */
ProcessDefinition getProcessByName(String processName,String categoryId);
/**
 * 根据流程模版的名称与版本号，返回与该名字与版本号匹配最流程模版对象
 * @param processName 流程模版名称
 * @param version 版本号
 * @return 返回流程模版对象
 */
ProcessDefinition getProcessByName(String processName,int version);

/**
 * 根据流程模版ID，开启一个流程实例
 * @param processId 流程模版ID
 * @param startProcessInfo 开启流程实例时所需要的各种信息的包装对象
 * @return 返回开启成功的流程实例对象
 */
ProcessInstance startProcessById(long processId,StartProcessInfo startProcessInfo);

/**
 * 根据流程模版key，开启一个流程实例
 * @param key 流程模版key
 * @param startProcessInfo 开启流程实例时所需要的各种信息的包装对象
 * @return 返回开启成功的流程实例对象
 */
ProcessInstance startProcessByKey(String key,StartProcessInfo startProcessInfo);

/**
 * 根据流程模版的名称，根据该名称流程模版最新版本开启一个流程实例
 * @param processName 流程模版名称
 * @param startProcessInfo 开启流程实例时所需要的各种信息的包装对象
 * @return 返回开启成功的流程实例对象
 */
ProcessInstance startProcessByName(String processName,StartProcessInfo startProcessInfo);
/**
 * 根据流程模版的名称与版本号，开启一个流程实例
```

```
     * @param processName 流程模版名称
     * @param startProcessInfo 开启流程实例时所需要的各种信息的包装对象
     * @param version 版本号
     * @return 返回开启成功的流程实例对象
     */
    ProcessInstance startProcessByName(String processName,StartProcessInfo startProcessInfo,int
    version);

    /**
     * 删除一个指定的流程实例对象，与这个流程实例相关的人工任务也将会被删除
     * @param processInstance 流程实例对象
     */
    void deleteProcessInstance(ProcessInstance processInstance);
    /**
     * 删除指定流程实例ID对应的流程实例对象
     * @param processInstanceId 流程实例ID
     */
    void deleteProcessInstanceById(long processInstanceId);

    /**
     * 从一个压缩文件包中部署一个新的流程模版
     * @param zipInputStream 一个压缩文件输入流
     * @return 部署成功后的流程模版对象
     */
    ProcessDefinition deployProcess(ZipInputStream zipInputStream);

    /**
     * 从一个文件流中部署一个新的流程模版
     * @param inputStream 文件流
     * @return 部署成功后的流程模版对象
     */
    ProcessDefinition deployProcess(InputStream inputStream);

    /**
     * 更新一个流程模版，用指定InputStream中包含的流程模版对象来替换指定ID的流程模版对象
     * @param inputStream 新的流程模版流对象
     * @param processId 要替换的目标流程模版ID
     * @return 更新成功后的流程模版对象
     */
    ProcessDefinition deployProcess(InputStream inputStream,long processId);

    /**
     * 根据给定的流程实例ID，返回对应的流程实例对象
     * @param processInstanceId 流程实例ID
     * @return 返回流程实例对象
     */
    ProcessInstance getProcessInstanceById(long processInstanceId);
    /**
     * 根据流程实例ID，返回与该流程实例相关的所有的流程变量
     * @param processInsanceId 流程实例ID
     * @return 返回与该流程实例相关的所有的流程变量集合
     */
    List<Variable> getProcessVariables(long processInsanceId);
    /**
     * 根据流程实例对象，返回与该流程实例相关的所有的流程变量
     * @param processInsance 流程实例对象
     * @return 返回与该流程实例相关的所有的流程变量集合
     */
    List<Variable> getProcessVariables(ProcessInstance processInsance);
```

```
/**
 * 获取指定流程实例上的指定key的流程变量的值
 * @param key 流程变量的key
 * @param processInstance 流程实例对象
 * @return 流程变量值
 */
Object getProcessVariable(String key,ProcessInstance processInstance);
/**
 * 获取指定流程实例ID上对应的流程实例中指定key的流程变量的值
 * @param key 流程变量的key
 * @param processInsanceId 流程实例ID
 * @return 流程变量值
 */
Object getProcessVariable(String key,long processInsanceId);
/**
 * 删除指定流程实例ID中指定key的流程变量值
 * @param key 流程变量的key
 * @param processInsanceId 流程实例ID
 */
void deleteProcessVariable(String key,long processInstanceId);

/**
 * 向指定流程实例ID对应的流程实例中添加流程变量
 * @param processInstanceId 流程实例ID
 * @param key 流程变量的key
 * @param value 对应的流程变量的值
 */
void saveProcessVariable(long processInstanceId,String key,Object value);
/**
 * 向指定流程实例ID对应的流程实例中批量添加流程变量
 * @param processInstanceId 流程实例ID
 * @param variables 要添加的流程变量的Map
 */
void saveProcessVariables(long processInstanceId,Map<String,Object> variables);

/**
 * @return 返回创建成功的流程实例查询对象
 */
ProcessInstanceQuery createProcessInstanceQuery();
/**
 * @return 返回创建成功的流程变量查询对象
 */
ProcessVariableQuery createProcessVariableQuery();

/**
 * @return 创建创建成功的流程模版查询对象
 */
ProcessQuery createProcessQuery();

/**
 * 删除一个指定ID的流程模版对象，与该模版相关的所有实例及任务都将被删除
 * @param processId 流程模版ID
 */
void deleteProcess(long processId);
/**
 * 删除一个指定KEY的流程模版对象，与该模版相关的所有实例及任务都将被删除
 * @param processKey 流程模版KEY
 */
void deleteProcess(String processKey);
```

```
/**
 * 删除一个指定流程模版对象，与该模版相关的所有实例及任务都将被删除
 * @param processDefinition 流程模版对象
 */
void deleteProcess(ProcessDefinition processDefinition);

/**
 * 根据给定的流程模版ID，更新当前内存中保存的对应的流程模版对象
 * @param processId 流程模版ID
 */
void updateProcessForMemory(long processId);

/**
 * 从本地内存中移除指定的流程模版ID对应的流程模版对象
 * @param processId 流程模版ID
 */
void deleteProcessFromMemory(long processId);
/**
 * 从本地内存中移除指定的流程模版KEY对应的流程模版对象
 * @param processKey 流程模版KEY
 */
void deleteProcessFromMemory(String processKey);
/**
 * 从本地内存中移除指定的流程模版对象
 * @param processDefinition 流程模版对象
```

```
 */
 void deleteProcessFromMemory(ProcessDefinition processDefinition);
}
```

下面我们再来看一段完成任务的业务代码：

完成任务示例代码

```
import java.util.HashMap;
import java.util.Map;
import javax.annotation.Resource;
import com.bstek.bdf2.core.context.ContextHolder;
import com.bstek.uflo.service.TaskService;
import com.bstek.uflo.service.StartProcessInfo;
public class BusinessTest {
 @Resource(name=TaskService.BEAN_ID)
 private TaskService taskService;
 public void saveDataAndStartProcess(BusinessData data,long taskId){
  saveBusinessData(data);
  Map<String,Object> variables=new HashMap<String,Object>();
  variables.put("businessName", data.getName());
  variables.put("businessOwner", data.getBusinessOwner());
  taskService.start(taskId);
  taskService.complete(taskId,variables);
 }
}
```

比较开始流程的代码片段，我们会发现操作模式基本一样，都是先进行业务数据操作（保存、更新之类），再进行流程操作（开始流程实例或完成任务），在这段完成任务的示例代码当中，首先调用saveBusinessData方法来保存提交上来的业务数据，接下来创建一个Map集合，用于存放需要写入流程的流程变量的值，这里的我们从BusinessData中取了两个值放入这个Map，接下来调用TaskService的start方法开始指定的人工任务（UFLO中人工任务的完成必须要先开始，某些时候对于处理周期比较长的人工任务，可以先开始，再不断设置处理进度progress的值，最后再完成，这里就直接开始后完成）,最后调用TaskClient的complete方法完成指定人工任务并回将流程变量的Map写入到流程实现。

同开始流程的代码一样，TaskService也是通过Spring注入，其在Spring中Bean的ID为TaskService.BEAN_ID。

同ProcessService接口作用类似，TaskService接口主要是操作所有与任务相关的动作的入口，TaskService接口详细信息如下所示：

TaskService接口源码

```
package com.bstek.uflo.service;
import java.util.List;
import java.util.Map;
import com.bstek.uflo.command.impl.jump.JumpNode;
import com.bstek.uflo.model.task.Task;
import com.bstek.uflo.model.task.TaskAppointor;
import com.bstek.uflo.model.task.TaskParticipator;
import com.bstek.uflo.model.task.TaskState;
import com.bstek.uflo.model.task.reminder.TaskReminder;
import com.bstek.uflo.query.TaskQuery;
public interface TaskService {
 public static final String BEAN_ID="uflo.taskService";
 public static final String TEMP_FLOW_NAME_PREFIX="__temp_flow_";

 /**
```

```
 * 设置处理处理进度值，正常情况下该值应该在0~100之间，同时当任务正常完成时，任务进度将自动设置为100
 * @param progress 任务进度值
 */
void setProgress(int progress);

/**
 * 根据给出的任务ID，获取当前任务节点下可指定任务处理人的任务节点名
 * @param taskId 任务ID
 * @return 可指定任务处理人的任务节点名列表
 */
List<String> getAvaliableAppointAssigneeTaskNodes(long taskId);

/**
 * 获取指定任务节点下配置的任务处理人列表
 * @param taskId 任务ID
 * @param taskNodeName 任务节点名称
 * @return 返回当前节点配置的任务处理人列表
 */
List<String> getTaskNodeAssignees(long taskId,String taskNodeName);
/**
 * 在某个任务中指定下一个指定任务节点上的任务处理人
 * @param taskId 具体任务对象ID
 * @param assignee 要指定的任务处理人
 * @param taskNodeName 指定任务处理人的任务节点名称
 */
void saveTaskAppointor(long taskId,String assignee,String taskNodeName);

/**
 * 在某个任务中指定下一个指定任务节点上的任务处理人,可以为多个处理人
 * @param taskId 具体任务对象ID
 * @param assignees 要指定的任务处理人集合
 * @param taskNodeName 指定任务处理人的任务节点名称
 */
void saveTaskAppointor(long taskId,String[] assignees,String taskNodeName);

/**
 * 向现有的会签任务中再添加一个新的会签任务
 * @param taskId 参考的任务ID
 * @param username 新的任务的处理人
 * @return 返回加签产生的新的任务对象
 */
Task addCountersign(long taskId,String username);

/**
 * 删除一个会签任务
 * @param taskId 要删除的会签任务的ID
 */
void deleteCountersign(long taskId);

/**
 * 获取当前任务可以跳转的任务节点名称
 * @param taskId 任务ID
 * @return 可跳转的目标任务节点集合
 */
List<JumpNode> getAvaliableForwardTaskNodes(long taskId);

/**
 * 完成指定ID的任务，同时设置下一步流向名称
 * @param taskId 要完成的任务ID
```

```
     * @param flowName 下一步流向名称
     */
    void complete(long taskId, String flowName);

    /**
     * 完成指定ID的任务，同时设置下一步流向名称
     * @param taskId 要完成的任务ID
     * @param flowName 下一步流向名称
     * @param opinion 任务处理意见
     */
    void complete(long taskId, String flowName,TaskOpinion opinion);

    /**
     * 批量完成指定ID的任务，并写入指定的流程变量
     * @param taskIds 要完成的任务的ID集合
     * @param variables 回写到要完成任务的变量集合
     */
    void batchComplete(List<Long> taskIds,Map<String,Object> variables);

    /**
     * 批量完成指定ID的任务，并写入指定的流程变量
     * @param taskIds 要完成的任务的ID集合
     * @param variables 回写到要完成任务的变量集合
     * @param opinion 任务处理意见
     */
    void batchComplete(List<Long> taskIds,Map<String,Object> variables,TaskOpinion opinion);

    /**
     * 完成指定ID的任务，同时设置下一步流向名称及回写到流程实例中的变量集合
     * @param taskId 任务ID
     * @param flowName 下一步流向名称
     * @param variables 回写的变量集合
     */
    void complete(long taskId, String flowName,Map<String,Object> variables);

    /**
     * 完成指定ID的任务，同时设置下一步流向名称及回写到流程实例中的变量集合
     * @param taskId 任务ID
     * @param flowName 下一步流向名称
     * @param variables 回写的变量集合
     * @param opinion 任务处理意见
     */
    void complete(long taskId, String flowName,Map<String,Object> variables,TaskOpinion opinion);

    /**
     * 完成指定ID的任务
     * @param taskId 要完成的任务ID
     */
    void complete(long taskId);

    /**
     * 完成指定ID的任务
     * @param taskId 要完成的任务ID
     * @param opinion 任务处理意见
     */
    void complete(long taskId,TaskOpinion opinion);

    /**
     * 完成指定ID的任务，同时设置回写到流程实例中的变量集合
```

```
     * @param taskId 任务ID
     * @param variables 变量集合
     */
    void complete(long taskId,Map<String,Object> variables);


    /**
     * 完成指定ID的任务，同时设置回写到流程实例中的变量集合
     * @param taskId 任务ID
     * @param variables 变量集合
     * @param opinion 任务处理意见
     */
    void complete(long taskId,Map<String,Object> variables,TaskOpinion opinion);

    /**
     * 完成指定ID的任务，跳转到指定的目标节点
     * @param taskId 任务ID
     * @param targetNodeName 指定的目标节点名称
     */
    void forward(long taskId,String targetNodeName);

    /**
     * 完成指定ID的任务，跳转到指定的目标节点
     * @param taskId 任务ID
     * @param targetNodeName 指定的目标节点名称
     * @param opinion 任务处理意见
     */
    void forward(long taskId,String targetNodeName,TaskOpinion opinion);

    /**
     * 完成指定ID的任务，跳转到指定的目标节点，同时设置回写到流程实例中的变量集合
     * @param taskId 任务ID
     * @param targetNodeName 指定的目标节点名称
     * @param variables 变量集合
     */
    void forward(long taskId,String targetNodeName,Map<String,Object> variables);

    /**
     * 完成指定ID的任务，跳转到指定的目标节点，同时设置回写到流程实例中的变量集合
     * @param taskId 任务ID
     * @param targetNodeName 指定的目标节点名称
     * @param variables 变量集合
     * @param opinion 任务处理意见
     */
    void forward(long taskId,String targetNodeName,Map<String,Object> variables,TaskOpinion opinion);

    /**
     * 完成指定ID的任务，跳转到指定的目标节点，同时设置回写到流程实例中的变量集合
     * @param task 任务对象
     * @param targetNodeName 指定的目标节点名称
     * @param variables 变量集合
     * @param opinion 任务处理意见
     */
    void forward(Task task,String targetNodeName,Map<String,Object> variables,TaskOpinion opinion);

    /**
     * 完成指定ID的任务，跳转到指定的目标节点，同时设置回写到流程实例中的变量集合，并指定任务状态
     * @param task 任务对象
     * @param targetNodeName 指定的目标节点名称
```

```
     * @param variables 变量集合
     * @param opinion 任务处理意见
     * @param state 任务状态
     */
    void forward(Task task,String targetNodeName,Map<String,Object> variables,TaskOpinion
opinion,TaskState state);
    /**
     * 完成指定ID的任务，回退到指定的目标节点，同时设置回写到流程实例中的变量集合
     * @param taskId 任务ID
     * @param targetNodeName 指定的目标节点名称
     * @param variables 变量集合
     * @param opinion 任务处理意见
     */
    void rollback(long taskId,String targetNodeName,Map<String,Object> variables,TaskOpinion opinion);

    /**
     * 完成指定ID的任务，回退到指定的目标节点，同时设置回写到流程实例中的变量集合
     * @param task 任务对象
     * @param targetNodeName 指定的目标节点名称
     * @param variables 变量集合
     * @param opinion 任务处理意见
     */
    void rollback(Task task,String targetNodeName,Map<String,Object> variables,TaskOpinion opinion);

    /**
     * 完成指定ID的任务，回退到指定的目标节点，同时设置回写到流程实例中的变量集合
     * @param taskId 任务ID
     * @param targetNodeName 指定的目标节点名称
     * @param variables 变量集合
     */
    void rollback(long taskId,String targetNodeName,Map<String,Object> variables);

    /**
     * 完成指定ID的任务，回退到指定的目标节点
     * @param taskId 任务ID
     * @param targetNodeName 指定的目标节点名称
     * @param variables 变量集合
     */
    void rollback(long taskId,String targetNodeName);

    /**
     * 获取指定任务ID对应的可回退的目标任务节点名列表
     * @param taskId 要回退的任务ID
     * @return 返回可回退的目标任务节点名列表
     */
    List<JumpNode> getAvaliableRollbackTaskNodes(long taskId);

    /**
     * 获取指定任务ID对应的可回退的目标任务节点名列表
     * @param task 要回退的任务
     * @return 返回可回退的目标任务节点名列表
     */
    List<JumpNode> getAvaliableRollbackTaskNodes(Task task);

    /**
     * 将指定ID的任务撤回到上一个任务节点
     * @param taskId 任务的ID
     */
    void withdraw(long taskId);
```

```
/**
 * 将指定ID的任务撤回到上一个任务节点
 * @param taskId 任务的ID
 * @param opinion 任务处理意见
 */
void withdraw(long taskId,TaskOpinion opinion);

/**
 * 将指定ID的任务撤回到上一个任务节点，并填充变量
 * @param taskId 任务的ID
 * @param variables 变量集合
 */
void withdraw(long taskId,Map<String,Object> variables);

/**
 * 将指定ID的任务撤回到上一个任务节点，并填充变量
 * @param taskId 任务的ID
 * @param variables 变量集合
 * @param opinion 任务处理意见
 */
void withdraw(long taskId,Map<String,Object> variables,TaskOpinion opinion);

/**
 * 判断当前任务是否可被撤回到上一任务节点
 * @param taskId 任务的ID
 * @return
 */
boolean canWithdraw(long taskId);

/**
 * 判断当前任务是否可回退到上一任务节点
 * @param task 任务对象
 * @return
 */
boolean canWithdraw(Task task);

/**
 * 根据ID获取一个任务对象
 * @param taskId 任务ID
 * @return 返回任务对象
 */
Task getTask(long taskId);

/**
 * 认领一个任务
 * @param taskId 要认领任务的ID
 * @param username 认领任务的人的用户名
 */
void claim(long taskId, String username);
/**
 * 对认领后的任务进行释放，从而允许其它人认领
 * @param taskId 要释放任务的ID
 * @param username 任务的释放人
 */
void release(long taskId);
/**
 * 开始处理一个任务
 * @param taskId 任务的ID
```

```
 */
void start(long taskId);

/**
 * 批量开始一批任务
 * @param taskIds 要开始的任务的ID集合
 */
void batchStart(List<Long> taskIds);

/**
 * 批量开始并完成一批指定的任务，并可写入指定的流程变量
 * @param taskIds 要完成的任务的ID集合
 * @param variables 要回写的流程变量
 */
void batchStartAndComplete(List<Long> taskIds,Map<String,Object> variables);

/**
 * 批量开始并完成一批指定的任务，并可写入指定的流程变量
 * @param taskIds 要完成的任务的ID集合
 * @param variables 要回写的流程变量
 * @param opinion 任务处理意见
 */
void batchStartAndComplete(List<Long> taskIds,Map<String,Object> variables,TaskOpinion opinion);

/**
 * 将一个任务挂起
 * @param taskId 要挂起的任务的ID
 * @param username 扶起任务的人的用户名
 */
void suspend(long taskId);
/**
 * 让处于挂起状态的任务恢复正常
 * @param taskId 要操作的任务的ID
 * @param username
 */
void resume(long taskId);

/**
 * 删除一个任务实例，要求这个任务对应的实例实例未结束，否则将不能删除
 * @param taskId 任务实例ID
 */
void deleteTask(long taskId);

/**
 * 删除某个任务节点上产生的所有任务实例，要求这里的processInstanceId对应的流程实例未结束
 * @param processInstanceId 流程实例ID
 * @param nodeName 任务节点名称
 */
void deleteTaskByNode(long processInstanceId,String nodeName);

/**
 * 取消指定的任务，任务状态将会标记上Canceled标记，同时任务也会被插入到历史表中，以备查询
 * @param taskId 任务ID
 */
void cancelTask(long taskId);

/**
 * 根据任务ID取得当前任务潜在处理人列表
 * @param taskId 任务ID
```

```
     * @return 处理人列表
     */
    List<TaskParticipator> getTaskParticipators(long taskId);

    /**
     * 获取指定流程实例下任务节点的通过指派的任务处理人信息
     * @param taskNodeName 任务节点名称
     * @param processInstanceId 流程实例ID
     * @return TaskAppointor集合
     */
    List<TaskAppointor> getTaskAppointors(String taskNodeName,long processInstanceId);
    /**
     * 更改任务处理人
     * @param taskId 任务ID
     * @param username 新的处理人用户名
     */
    void changeTaskAssignee(long taskId,String username);

    /**
     * 查找指定任务节点上指定key对象的UserData的值
     * @param task 任务对象
     * @param key UserData的key值
     * @return UserData对应的value值
     */
    String getUserData(Task task,String key);

    /**
     * 查找指定任务节点上指定key对象的UserData的值
     * @param processId 流程模版ID
     * @param taskNodeName 任务节点名称
     * @param key UserData的key值
     * @return UserData对应的value值
     */
    String getUserData(long processId,String taskNodeName,String key);

    TaskQuery createTaskQuery();

    List<TaskReminder> getTaskReminders(long taskId);

    void deleteTaskReminder(long taskReminderId);
```

```
 List<TaskReminder> getAllTaskReminders();
}
```

ProcessService对于我们程序员来说，主要是实现业务流程的开启、流程模版与实例的删除、流程模版与变量的获取等操作；而TaskService则提供了大量针对人工任务的操作。在这两个接口当中，我们为每个方法都添加了详细的注释，使用时查看注释应该就可以明确具体用法。

关于UFLO2的任务状态
在UFLO2当中，对于已完成的任务或跳转、退回的任务，不会被立即删除，而是还会存在运行的任务表中，只是他们的状态（TaskState）发生的相应的变化，在流程实例运行到结束节点时，才会将这些任务从运行的任务表中删除。所以如果我们通过TaskService的createTaskQuery方法创建一个TaskQuery对象，要对任务进行查询时需要注意任务的状态，也就是需要TaskQuery添加限制任务状态的方法（addTaskState），多个状态采“或”关系来连接，比如如果我们要自己定义待办任务列表页面，在取待办任务列表时就需要为TaskQuery添加下面这些任务状态限制：

查询待办时，添加任务状态限制

```
TaskQuery query=taskService.createTaskQuery();
query.addTaskState(TaskState.Created);
query.addTaskState(TaskState.InProgress);
query.addTaskState(TaskState.Ready);
query.addTaskState(TaskState.Suspended);
query.addTaskState(TaskState.Reserved);
```

这个TaskQuery只会帮我们取到上述五种状态的任务，这五种状态的任务才是待办任务。

在项目当中，如果我们需要查询流程流转相关历史信息，那么可以通过HistoryService来实现。在HistoryService当中提供了大量的与流程相关的历史信息查询功能，比如根据流程实例的ID查询当前实例流转过程当中经过了哪些流程节点，或者根据流程实例查询当前实例流转时产生了哪些人工任务，这些人工任务在处理时处理人是谁，所有者是谁，花费多长时间等。同样HistoryService也配置在Spring当中，其bean的ID为“uflo.historyService”，我们的业务系统可根据需要调用这个HistoryService查询所需要的历史流程信息。HistoryService接口详细描述如下：

HistoryService接口源码

```
package com.bstek.uflo.service;
import java.util.List;
import com.bstek.uflo.model.HistoryActivity;
import com.bstek.uflo.model.HistoryProcessInstance;
import com.bstek.uflo.model.HistoryTask;
import com.bstek.uflo.model.HistoryVariable;
import com.bstek.uflo.query.HistoryProcessInstanceQuery;
import com.bstek.uflo.query.HistoryProcessVariableQuery;
import com.bstek.uflo.query.HistoryTaskQuery;
/**
 * @author Jacky.gao
 * @since 2013年8月15日
 */
public interface HistoryService {
 public static final String BEAN_ID="uflo.historyService";
 /**
  * 根据流程实例ID，返回当前实例产生的所有历史节点的集合
  * @param processInstanceId 流程实例ID
  * @return 返回HistoryActivity集合
  */
```

```
List<HistoryActivity> getHistoryActivitysByProcesssInstanceId(long processInstanceId);
/**
 * 根据历史流程实例ID，返回当前历史流程实例产生的所有历史节点的集合
 * @param historyProcessInstanceId 历史流程实例ID
 * @return 返回HistoryActivity集合
 */
List<HistoryActivity> getHistoryActivitysByHistoryProcesssInstanceId(long historyProcessInstanceId);
/**
 * 根据流程模版ID，返回所有的历史流程实例集合
 * @param processId 流程模版ID
 * @return 返回所有的历史流程实例集合
 */
List<HistoryProcessInstance> getHistoryProcessInstances(long processId);
/**
 * 根据流程实例ID，返回的对应的历史流程实例
 * @param processInstanceId 流程实例ID
 * @return 返回的对应的历史流程实例
 */
HistoryProcessInstance getHistoryProcessInstance(long processInstanceId);
/**
 * 根据流程实例ID，返回对应的历史任务集合
 * @param processInstanceId 流程实例ID
 * @return 返回历史任务集合
 */
List<HistoryTask> getHistoryTasks(long processInstanceId);
/**
 * 根据任务ID，返回对应的历史任务
 * @param taskId 任务ID
 * @return 返回历史任务
 */
HistoryTask getHistoryTask(long taskId);
/**
 * @return 返回创建的历史任务查询对象
 */
HistoryTaskQuery createHistoryTaskQuery();
/**
 * @return 返回创建创建的历史流程实例查询对象
 */
HistoryProcessInstanceQuery createHistoryProcessInstanceQuery();
/**
 * @return 返回创建的历史流程实例查询对象
 */
HistoryProcessVariableQuery createHistoryProcessVariableQuery();
/**
 * 根据历史流程实例ID，返回所有的历史流程变量
 * @param historyProcessInstanceId 历史流程实例ID
 * @return 返回所有的历史流程变量
 */
List<HistoryVariable> getHistoryVariables(long historyProcessInstanceId);
/**
 * 根据历史流程实例ID和流程变量名，返回对应的历史流程变量对象
 * @param historyProcessInstanceId 历史流程实例ID
 * @param key 流程变量名字
 * @return 返回对应的历史流程变量对象
```

```
  */
 HistoryVariable getHistoryVariable(long historyProcessInstanceId,String key);
}
```