ProcEngine使用说明

一. 流程(Process), 节点(Node) 与 连接线(FlowLine)

流程(Process)

至少包含一个开始节点和一个结束节点的Node链条, 主要属性:

`processKey` 流程的全局唯一标识

`businessKey` 流程关联的对象ID

`name` 流程名称

`processGroup` 所在流程组, 如项目/合同等

`approveType` 为适应多流程版本而添加, 一般与 `ProcessKey` 相同即可

`approveTypeName` 为适应多流程版本而添加, 一般与 `name` 相同即可

`启动检查表达式` 流程启动前置检查表达式, 如果表达式执行结果为 `true` , 则允许流程启动, 否则启动失败.

下面例子表示:当ID为 `@BUSINESS_KEY@` 的应收款的状态为0时, 允许启动, 否则给出错误提示:"保存状态下才可启动此流程"

```
${trueIf(poStatusValid("Receivables", @BUSINESS_KEY@, 0), true,"保存状态下才可启动此流程")}
```

其中 `@BUSINESS_KEY@` 为全局变量, 值为启动流程时传入的 `businessKey` , 函数 `trueIf` 和 `poStatusValid` 具体用法参考函数手册.

节点(Node)

组成流程的单元. 节点可以独立于流程而存在, 一个节点可以应用到多条流程中(如果节点的persistence为true). 目前支持的节点类型包括: *开始节点(S)*, *结束节点(E)*, *用户任务(UT)*, *服务任务(ST)*, *互斥网关(EG)*,

主要属性:

`nodeKey` 节点的全局唯一标识

`role` 指定完成此任务需要的角色列表,如果为空, 则不控制角色, 即所有用户都可完成此任务.

`handleType` 节点处理类型, 包括:*普通节点(N)*, *提交审核(AR)*, *处理审核(HA)*

`persistence` 持久化, 表示此节点是否可以应用到多条流程中.

`前置表达式` 与 `NodeHandler` 的 `preOperation` 作用相同.

`后置表达式` 与 `NodeHandler` 的 `postOperation` 作用相同.

连接线(FlowLine)

连接流程各个节点,并决定流程走向, 主要属性:

`lineKey` 线条的全局唯一标识

`formNode` 线条起始节点l

`toNode` 线条目标节点

`condition` 条件表达式, 决定下一个节点,如:

```
${@approveResult@=="true"}
```

表示当变量 `@approveResult@` 的值为"true"时, 该连接线目标节点为流程的下一个节点

二. 节点动态属性(FormProperty)

每个节点可定义多个 `动态属性(FormProperty)` , 客户端会根据节点定义的动态属性, 正确地渲染出完成此任务时需要提交的页面表单元素.

包含以下属性:

`name` 变量名, 英文变量名, 如reason, approveResult等

`description` 变量描述

`type` 变量类型, 可选值有:

> - string - 字符串
> - text - 长文本
> - select - 下拉列表
> - radio - 单选
> - file - 文件
> - date - 日期
> - time - 时间
> - int - 整数
> - double - 小数
> - money - 金额
> - percent - 百分比
> - boolean - 布尔值

`minSelection` 下限, 对于字符串, 为字符串最小长度; 对于数字, 为最小值; 对于下拉列表, 为最小选择数.....

`maxSelection` 上限, 对于字符串, 为字符串最大长度; 对于数字, 为最大值; 对于下拉列表, 为最大选择数.....

`checkLogicExp` 取值校验表达式, 用于自动校验表单逻辑, 如下面例子用于校验客户名称是否存在, 调用了ClientService的方法existByName(), 当前变量的值用变量@this@表示

```
      ${clientService.existByName(@this@)}
```

`checkLogicFailedMsg` 与 `checkLogicExp` 同时使用, 用于给出逻辑校验失败后的错误信息

`valueExp` 取值表达式, 用于初始化表单数据. 如下面例子, 当表单类型为 `select` 时, 客户端初始下拉列表的可选项为 `通过` , `打回修改` , `无效` .

```
${radio("true", "通过", "false", "打回修改", "invalid", "无效")}
```

`required` 是否为必传变量

`readonly` 是否为只读变量

`multiple` 是否允许多选, 适用于下拉列表, 文件上传等情况

三. ProcessHandler

通常每条流程都需要定制一个ProcessHandler, 自定义的ProcessHandler需要实现接口 `com.shls.helper.proc.BasicProcessHandler` , 接口包含以下方法:

```
    /**
     * 创建流程实例前，如果此方法抛出异常，则会终止流程启动
     */
    void beforeStartProcess(ProcessContext context);

    /**
     * 创建流程实例后，执行开始节点之前. 如果此方法抛出异常，则会终止流程运行，并删除已生成的流程实例
     */
    void afterProcessStarted(ProcessContext context);

    /**
     * 流程完全启动完成后，即已创建流程实例，且成功执行开始节点之后. 如果此方法抛出异常，则会终止流程运行，并删除已生成的流程实例
     */
    void afterProcessStartedCompletely(ProcessContext context);

    /**
     * 创建启动失败监听.
     */
    void onProcessStartedFailed(ProcessContext context, Exception cause);

    /**
     * 指派任务处理人后
     *
     * @param assignTo 被指派人
     * @param manual   手动指派
     * @param userTYpe {@link com.shls.service.BasicService#USER_TYPE_HEXIN}, {@link com.shls.service.BasicService#USER_TYPE_CLIENT}
     */
    void onAssignedTaskHandler(TaskContext context, long assignTo, String
```

```
    userTYpe, boolean manual);

        /**
         * 转派处理人后
         * @param context
         * @param targetNodeKey 转派目标节点key
         * @param originalAssignTo 转派前处理人
         * @param targetAssignTo 转派后处理人
         */
        void onTransferAssign(ProcessContext context, String targetNodeKey, long
    originalAssignTo, long targetAssignTo);
```

使用注解 `@ProcessHandler` 指定自定义的ProcessHandler关联的具体流程. 下面例子表示 `processKey` 为 `"receivablesApprove"` 的流程的ProcessHandler:

```
        @ProcessHandler("receivablesApprove")
        @Component
        public class ReceivablesApproveProcHandler implements BasicProcessHandler
        {
         //...
        }
```

四. NodeHandler

通常每条个点都需要定制一个NodeHandler, 赋予节点具体的业务逻辑处理能力

在方法上使用注解 `@NodeHandler` 与 `@AfterTask` 或 `@BeforeTask` , 表示该方法是一个NodeHandler. NodeHandler方法必须定义在实现了 `com.shls.helper.proc.BasicNodeHandler` 的类里面.

- @NodeHandler: 修饰方法为一个NodeHandler, 包含以下属性:

`value` : NodeHandler的全局唯一标识, 通常与NodeKey命名一致, 这样会自动与具体的节点对应起来.

`preOperation` 节点前置操作, 在当前NodeHandler执行之前被执行的NodeHandler, 值为另一个NodeHandler的value.

`postOperation` 节点后置操作, 在当前NodeHandler执行之后被执行的NodeHandler, 值为另一个NodeHandler的value.

- @BeforeTask: 指定该NodeHandler在任务开始之前执行. 通常用来判断任务是否具备开始的条件, 如果不具备开始的条件, 在这个方法中抛出任意异常,则可中止任务继续进行.
- @AfterTask: 指定该NodeHandler在当前任务结束之后, 且下个任务开始之前被执行. 通常用来执行具体的业务逻辑,如更新对象状态为通过/不通过等. 如果此方法抛出异常, 则当前任务会回滚到刚开始的状态.

下面例子, 定义了三个NodeHandler, 都使用注解 `@AfterTask` 表示在任务结束之后被执行. doService指定了 `preOperation` 为"checkStatus", `postOperation` 为"updateStatus", 所以最终的NodeHandler执行顺序为:

> checkStatus -> doService -> updateStatus

注意: 同一条NodeHandler链条上的各个NodeHandler必须有相同的@AfterTask/@BeforeTask

```
public class TestHandler implements BasicNodeHandler
{
    @NodeHandler(value = "checkStatus")
    @AfterTask
    public void checkStatus(TaskContext context)
    {
        //do checkStatus
    }

    @NodeHandler(value = "doService", preOperation = "checkStatus", postOperation
= "updateStatus")
    @AfterTask
    public void doService(TaskContext context, CustomForm form)
    {
        //do doService
    }

    @NodeHandler(value = "updateStatus")
    @AfterTask
    public void updateStatus(TaskContext context)
    {
        //do updateStatus
    }
}
```

上述例子中, NodeHandler "doService"方法第一个参数为TaskContext, 所有NodeHandler方法的第一个参数必须为TaskContext. 可以在方法后面追加任意参数, 流程引擎会自动注入变量到该参数中.

假设例子中的类CustomForm定义为:

```
public class CustomForm
{
    private String name;
    private int age;
    private List<String> roles;

    // 省略getter/setter
}
```

流程引擎会把从客户端传入的参数name, age和roles注入到方法参数 `form` 中.

五. ProcessContext与TaskContext

ProcessContext

用于获取当前流程上下文信息, 包含属性:

`processKey` 流程key

`businessKey` 流程启动时传入的businessKey

`currentUser` 当前登录用户

`instance` 当前流程实例

`formProperties` 流程自启动以来收集到的所有表单属性. 需要注意的是, 如果一个FormProperty被多次赋值, 只会保留最后一次更新的值

`procService` 当前系统中的 `ProcService` 实例

TaskContext

用于获取当前Task上下文信息, 继承自 `ProcessContext` , 除了包含 `ProcessContext` 中的所有属性外, 扩展了属性:

`task` 当前Task实例

`candidateRoles` 当前Task候选角色列表

六. 预分配与转派

预分配(PreAssign)

预分配是指在某个节点进行的过程中, 预先对后面一个或多个节点进行人员分配.

下面例子作用是, 分配NodeKey为 `createProject_qyfzr` 的处理人为ID为1000的用户

```
    @NodeHandler(value = "createProject_khjl")
    @AfterTask
    public void afterKhjl(TaskContext context)
    {
        context.getProcService().setPreAssign(context.getInstance().getId(),
"createProject_qyfzr", userId, null, 1000);
    }
```

转派(TransferAssign)

转派是指对于已进行了 预分配 处理人的节点, 节点处理人从A更换为B, 并生成一条转派记录.

下面例子作用是, 转派NodeKey为 `createProject_qyfzr` 的处理人为ID为1000的用户

```
    @NodeHandler(value = "createProject_khjl")
    @AfterTask
    public void afterKhjl(TaskContext context)
    {
        context.getProcService().transferAssign(context.getInstance().getId(),
"createProject_qyfzr", 1000, "CLIENT", "转派原因", currentUser());
    }
```