

Snaker 用户手册

目录

- Snaker 用户手册 1
- 一、 常用操作..... 3
 - 1、 流程定义部署..... 3
 - deploy..... 3
 - redeploy 3
 - undeploy..... 3
 - 2、 启动流程实例..... 4
 - 根据 id 启动实例..... 4
 - 根据 name 启动实例..... 4
 - 3、 执行任务..... 4
 - 4、 转派任务..... 5
 - 5、 撤回任务..... 5
 - 6、 提取任务..... 5
 - 7、 任务驳回..... 5
 - 8、 自由流程..... 6
 - 9、 动态添加、减少参与者 7
 - 10、 编码设置参与者 7
 - 11、 节点拦截器 8
 - 12、 参与者使用组 9
 - 自定义访问策略..... 9
 - 获取待办任务..... 9
 - 执行任务..... 9
 - 13、 委托代理..... 10
 - 增加委托代理表..... 10
 - 配置拦截器..... 10
 - 管理委托代理..... 10

14、	子流程.....	11
二、	应用整合.....	12
1、	API 方式整合	12
➤	配置 snaker.xml	12
➤	编写帮助类.....	12
➤	调用流程引擎.....	14
2、	与 Spring 整合.....	14
➤	SnakerEngine 配置.....	14
➤	DBAccess 配置	15
➤	事务配置.....	16
三、	流程设计器.....	18
四、	API 说明	24

一、常用操作

1、流程定义部署

如何使用流程设计器定义流程请参考[三、流程设计器](#)。

部署的相关方法包含: **deploy**、**redeploy**、**undeploy**, 分别表示流程定义的部署、重新部署、卸载。部署时统一使用 **InputStream** 输入流作为流程定义的数据。可借助 **org.snaker.engine.helper.StreamHelper** 帮助类完成。方法如下定义:

InputStream 的方法名称	描述
getStreamFromString	根据字符串获取输入流
getStreamFromFile	根据文件对象获取输入流
getStreamFromClasspath	根据类路径下的资源文件名称获取输入流
getStreamFromUrl	根据 Url 远程资源获取输入流

➤ **deploy**

```
engine.process().deploy(StreamHelper.
```

```
getStreamFromClasspath("test/task/simple/process.snaker"));
```

部署相同的流程定义, 会产生版本号依次加1的新的流程定义数据。但是不会对同名的流程实例产生影响。

➤ **redeploy**

```
engine.process().redeploy(processId, StreamHelper.
```

```
getStreamFromClasspath("test/task/simple/process.snaker"));
```

重新部署流程会影响当前流程实例的执行

➤ **undeploy**

```
engine.process().undeploy(processId);
```

卸载流程只会更新状态 **state** 值, 不会物理删除数据。

2、启动流程实例

可根据流程定义的 **id** 或者名称启动流程实例。如果相同的流程名称存在不同的版本，并使用名称启动实例时，会按照最新的版本来启动，其它低版本运行中的流程实例不会受到影响，这样就允许流程的多个版本同时运行。

➤ 根据 id 启动实例

```
engine.startInstanceById(processId);  
engine.startInstanceById(processId, "admin");  
engine.startInstanceById(processId, "admin", args);
```

由 **id** 启动实例的参数为：流程定义 **id**、操作人 **operator**、参数列表 **args**

➤ 根据 name 启动实例

```
engine.startInstanceByName("simple");  
engine.startInstanceByName("simple", 0);  
engine.startInstanceByName("simple", 0, "admin");  
engine.startInstanceByName("simple", 0, "admin", args);
```

由 **name** 启动实例的参数为：流程定义 **name**、版本号 **version**、操作人 **operator**、参数列表 **args**

流程实例的启动会在以下的表中产生数据：

wf_order、**wf_hist_order**

3、执行任务

执行任务的处理逻辑包括两部分：

- 完成当前任务
- 按照流程定义产生新的任务

执行任务的 api 如下：

```
engine.executeTask(taskId);  
  
engine.executeTask(taskId, "admin");  
  
engine.executeTask(taskId, "admin", args);
```

执行任务的参数为：任务号 **taskId**、操作人 **operator**、参数列表 **args**

4、转派任务

任务转派的业务逻辑是结束当前任务，并创建新的任务给转派人。其调用的 api 为：

```
engine.task().createNewTask(task.getId(), 0, "test");  
  
engine.task().complete(task.getId());
```

`createNewTask` 方法中的第二个参数“任务类型”表示创建主办、协办任务。

5、撤回任务

根据历史任务 **id**，撤回由该历史任务派发的所有活动任务，如果无活动任务，则不允许撤回。抛出 `SnakerException` 异常

```
engine.task().withdrawTask(taskId, "admin");
```

6、提取任务

任务提取一般发生在参与者为部门、角色等组的情况下，该组的某位成员提取任务后，其它成员无法处理任务。

```
engine.task().take(taskId, "admin");
```

7、任务驳回

任务驳回有多种场景，常见的有：驳回上一步、驳回到任意节点

```
engine.executeAndJumpTask(String taskId, String operator, Map<String, Object> args,
```

String nodeName)

方法的参数 **nodeName** 决定驳回的方式：

- **nodeName** 为空（**null** 或空字符），则驳回至上一步（不允许驳回至 **fork**、**join**、**suprocess** 以及会签任务）
- **nodeName** 非空，则根据 **nodeName** 确定跳转的目标节点。该实现原理与其它流程引擎思路一致，通过动态创建连接完成跳转。

8、自由流程

Snaker 支持两种自由流：

- 已经定义流转节点，由用户随意在节点之间跳转
在 **engine.executeAndJumpTask** 方法已经支持，参考[任务驳回](#)
- 未定义流转节点，即流程定义没有流程模型的情况，由用户随意创建自定义任务（任务名称、任务参与者、任务关联的表单等）

```
TaskModel tm1 = new TaskModel();

tm1.setName("task1");

tm1.setDisplayName("任务1");

List<Task> tasks = null;

tasks = engine.createFreeTask(orderId, "1", args, tm1);

for(Task task : tasks) {

    engine.task().complete(task.getId(), "1", null);

}
```

自由任务需要调用 **engine.task().complete** 方法结束任务。并且需要手动结束流程实例。

```
engine.order().complete(order.getId());
```

自由流程存在强制终止的情况，此时需要调用

```
void terminate(String orderId);
```

```
void terminate(String orderId, String operator);
```

强制终止实例先结束该实例所有活动的任务，最后结束流程实例。

9、动态添加、减少参与者

添加参与者需要判断所属的任务模型的参与类型（ANY、ALL）

ITaskService 提供两个添加参与者的方法：

➤ `addTaskActor(String taskId, String... actors);`

该方法根据 `taskId` 对应的任务模型判断是否属于会签任务，如果属于 ALL 的参与类型，则添加的每个 `actor` 都会产生新的任务。如果属于 ANY 的参与类型，则只是将 `actor` 添加到当前的任务中

➤ `addTaskActor(String taskId, Integer performType, String... actors);`

该方法根据 `performType` 类型确定是否产生新的任务

➤ `removeTaskActor(String taskId, String... actors);`

减少参与者仅仅是将所属任务的参与者删除（参考表：wf_task_actor）

10、 编码设置参与者

编码设置参与者主要用到 `AssignmentHandler` 接口，并在定义任务模型时，需要配置该接口的实现类：

General	
01.name	task1
02.displayName	task1
03.form	
04.assignee	
05.assignmentHandler	test.task.assignmenthandler.TaskAssign
06.performType	ANY
07.preInterceptors	
08.postInterceptors	

```
public class TaskAssign implements AssignmentHandler {  
  
    public Object assign(Execution execution) {  
  
        return "admin";  
  
    }  
  
}
```

此处方便测试，直接返回字符串。实际使用时，应该根据执行对象的 **args** 参数来判断执行人。

11、 节点拦截器

Snaker 支持对任意类型的节点提供前置、后置拦截器处理。

General	
01.name	start1
02.preInterceptors	
03.postInterceptors	

preInterceptors 为前置拦截器属性配置、**postInterceptors** 为后置拦截器属性配置。自定义拦截器需要实现接口：**SnakerInterceptor**。如下代码片段即实现对产生的任务拦截并输出日志信息：

```
public class LocalTaskInterceptor implements SnakerInterceptor {
    private static final Logger log =
        LoggerFactory.getLogger(LocalTaskInterceptor.class);
    public void intercept(Execution execution) {
        if(log.isInfoEnabled()) {
            log.info("LocalTaskInterceptor start...");
            for(Task task : execution.getTasks()) {
                StringBuffer buffer = new StringBuffer(100);
                buffer.append("创建任务[标识=").append(task.getId());
                buffer.append(", 名称=").append(task.getDisplayName());
                buffer.append(", 创建时间=").append(task.getCreateTime());
                buffer.append(", 参与者={");
                if(task.getActorIds() != null) {
                    for(String actor : task.getActorIds()) {
                        buffer.append(actor).append(";");
                    }
                }
                buffer.append("}");
                log.info(buffer.toString());
            }
            log.info("LocalTaskInterceptor finish...");
        }
    }
}
```


12、 参与者使用组

➤ 自定义访问策略

```
public class CustomAccessStrategy extends GeneralAccessStrategy {  
  
    protected List<String> ensureGroup(String operator) {  
  
        List<String> groups = new ArrayList<String>();  
  
        if(operator.equals("test")) {  
  
            groups.add("test");  
  
        } else {  
  
            groups.add("role1");  
  
        }  
  
        return groups;  
  
    }  
}
```

继承 **GeneralAccessStrategy** 类，实现 **ensureGroup** 方法，根据操作人获取该操作人对应的组（部门、角色等）

在 **snaker.xml** 中增加访问策略类的配置：

```
<bean class="test.task.group.CustomAccessStrategy"/>
```

➤ 获取待办任务

获取待办任务时，设置的参与者数组增加组的信息即可

```
String[] actorIds = new String[]{username, groups};  
snakerEngine.query().getWorkItems(page,  
  
    new QueryFilter().setOperators(actorIds));
```

➤ 执行任务


执行任务的方式没有改变，依然是调用

```
snakerEngine.executeTask(taskId, userName, args);
```

13、 委托代理

➤ 增加委托代理表

表名称为 **wf_surrogate**，其字段定义如下：

	Field	Type	Comment
	id	varchar(100)	主键ID
	process_Name	varchar(100)	流程名称
	operator	varchar(100)	授权人
	surrogate	varchar(100)	代理人
	odate	varchar(64)	操作时间
	sdate	varchar(64)	开始时间
	edate	varchar(64)	结束时间
	state	tinyint(1)	状态

➤ 配置拦截器

在类路径下的 **snaker.xml** 中增加拦截器配置，如下：

```
<bean class="org.snaker.engine.impl.SurrogateInterceptor"/>
```

由于在 snaker.xml 中配置的拦截器，属于全局任务拦截器，则当产生新的任务时，会执行该拦截器负责检查委托代理表，是否存在委托授权情况，如存在，则增加代理人的参与权限。

➤ 管理委托代理

保存或更新委托代理配置

```
engine.manager().saveOrUpdate(Surrogate surrogate);
```

删除委托代理配置

```
engine.manager().deleteSurrogate(String id);
```

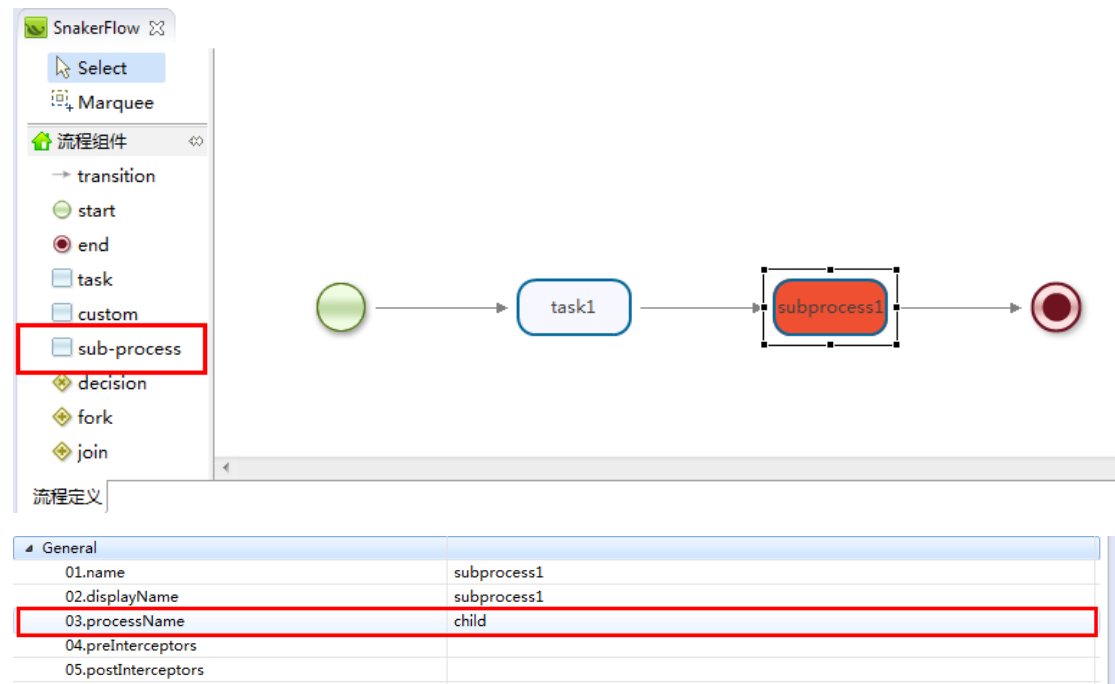
根据id获取委托代理对象

```
engine.manager().getSurrogate(String id);
```

根据当前操作人、流程名称获取该操作人的代理人，支持多级代理

```
engine.manager().getSurrogate(String operator, String processName)
```

14、 子流程



子流程定义，主要是设置 processName（子流程的名称 name 值）属性

二、应用整合

1、API 方式整合

API 整合方式适用于无业务容器托管的场景。

➤ 配置 snaker.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<config>

<!-- jdbc的数据库访问与事务拦截器-->

<bean class="org.snaker.engine.access.jdbc.JdbcAccess"/>

<bean

class="org.snaker.engine.access.transaction.DataSourceTransactionInterceptor"/>

<!--数据库方言配置-->

<bean class="org.snaker.engine.access.dialect.MySqlDialect"/>

<!-- 配置表达式引擎实现类 -->

<bean class="org.snaker.engine.impl.JuelExpression"/>

</config>
```

参数类型	访问对象	事务管理拦截器
DataSource	JdbcAccess	DataSourceTransactionInterceptor
SessionFactory	HibernateAccess	Hibernate3TransactionInterceptor
SqlSessionFactory	MybatisAccess	MybatisTransactionInterceptor

➤ 编写帮助类

```
import javax.sql.DataSource;
import org.snaker.engine.SnakerEngine;
import org.snaker.engine.access.jdbc.JdbcHelper;
import org.snaker.engine.cfg.Configuration;
```

```

/**
 * Snaker引擎帮助类
 */
public class SnakerHelper {
    private static final SnakerEngine engine;

    static {
        DataSource dataSource = JdbcHelper.getDataSource();
        engine = new Configuration()
            .initAccessDBObject(dataSource)
            .buildSnakerEngine();
    }

    public static SnakerEngine getEngine() {
        return engine;
    }
}

```

考虑到与现有应用整合，故提供 `initAccessDBObject` 方法，由应用系统提供具体的数据库访问对象，目前支持（jdbc 方式 `DataSource`、hibernate 方式 `SessionFactory`、mybatis 方式 `SqlSessionFactory`）。测试用例的基类中没有调用该方法。代码如下：

类路径：org.snaker.engine.test.TestSnakerBase

```

public class TestSnakerBase {
    protected String processId;
    protected SnakerEngine engine = getEngine();
    protected IProcessService processService = engine.process();
    protected IQueryService queryService = engine.query();

    private SnakerEngine getEngine() {
        return new Configuration().buildSnakerEngine();
    }
}

```

为了方便测试，Snaker 根据所使用的 DBAccess 实现类，从 `snaker.properties` 中获取初始化参数，如下：

```

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/snaker

jdbc.username=root

jdbc.password=root

```

➤ 调用流程引擎

```
SnakerHelper.getEngine().process().deploy(.....);
```

2、与 Spring 整合

Spring 集成 Snaker 时，需要配置流程引擎、事务管理、数据访问方式.具体可参考 snaker-demo。具体配置如下：

➤ SnakerEngine 配置

```
<bean class="org.snaker.engine.spring.SpringSnakerEngine">
    <property name="processService" ref="processService"/>
    <property name="orderService" ref="orderService"/>
    <property name="taskService" ref="taskService"/>
    <property name="queryService" ref="queryService"/>
    <property name="managerService" ref="managerService"/>
</bean>
<bean id="dbAccess"
class="org.snaker.engine.access.hibernate3.HibernateAccess">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="processService" class="org.snaker.engine.core.ProcessService">
    <property name="access" ref="dbAccess"/>
    <property name="cacheManager" ref="cacheManager"/>
</bean>
<bean id="orderService" class="org.snaker.engine.core.OrderService">
    <property name="access" ref="dbAccess"/>
</bean>
<bean id="taskService" class="org.snaker.engine.core.TaskService">
    <property name="access" ref="dbAccess"/>
</bean>
<bean id="managerService" class="org.snaker.engine.core.ManagerService">
    <property name="access" ref="dbAccess"/>
</bean>
<bean id="queryService" class="org.snaker.engine.core.QueryService">
    <property name="access" ref="dbAccess"/>
</bean>
<bean id="cacheManager"
class="org.snaker.engine.cache.memory.MemoryCacheManager"/>
```

➤ DBAccess 配置

🚦 Hibernate 方式

```
<!-- hibernate access -->
<bean id="dbAccess" class="org.snaker.engine.access.hibernate3.HibernateAccess">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

使用 hibernate 时, 需要在 sessionFactory 配置中增加 snaker 映射文件的 mappingResources 属性:

```
    <property name="mappingResources">
        <list>
            <value>hbm/snaker.task.hbm.xml</value>
            <value>hbm/snaker.order.hbm.xml</value>
            <value>hbm/snaker.cccorder.hbm.xml</value>
            <value>hbm/snaker.process.hbm.xml</value>
            <value>hbm/snaker.taskactor.hbm.xml</value>
            <value>hbm/snaker.workitem.hbm.xml</value>
            <value>hbm/snaker.surrogate.hbm.xml</value>
        </list>
    </property>
```

🚦 SpringJdbc 方式

```
<!-- springjdbc access -->
<bean id="dbAccess" class="org.snaker.engine.access.spring.SpringJdbcAccess">
    <property name="dataSource" ref="dataSource"/>
    <property name="lobHandler" ref="lobHandler"/>
</bean>
<bean id="lobHandler" class="org.springframework.jdbc.support.Lob.DefaultLobHandler"
lazy-init="true" />
```

🚦 Mybatis 方式

```
<!-- mybatis access -->
<bean id="dbAccess" class="org.snaker.engine.access.mybatis.MybatisAccess">
    <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
</bean>
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="configLocation" value="classpath:mybatis.cfg.xml"></property>
    <property name="dataSource" ref="dataSource" />
    <property name="typeAliasesPackage" value="org.snaker.engine.entity" />
</bean>
```

使用mybatis, 需要在mybatis.cfg.xml配置文件中增加snaker的类型及映射文件

```

<typeAliases>
  <package name="org.snaker.engine.entity"/>
</typeAliases>
<mappers>
  <mapper resource="mapper/process.xml"/>
  <mapper resource="mapper/order.xml"/>
  <mapper resource="mapper/task.xml"/>
  <mapper resource="mapper/task-actor.xml"/>
  <mapper resource="mapper/hist-order.xml"/>
  <mapper resource="mapper/hist-task.xml"/>
  <mapper resource="mapper/hist-task-actor.xml"/>
  <mapper resource="mapper/query.xml"/>
  <mapper resource="mapper/hist-query.xml"/>
  <mapper resource="mapper/surrogate.xml"/>
</mappers>

```

➤ 事务配置

基于 Spring 的项目在使用 Snaker 时，完全将事务交给 Spring 托管，其事务配置如下：

```

<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="start*" propagation="REQUIRED"/>
    <tx:method name="execute*" propagation="REQUIRED"/>
    <tx:method name="save*" propagation="REQUIRED"/>
    <tx:method name="delete*" propagation="REQUIRED" />
    <tx:method name="update*" propagation="REQUIRED" />
    <tx:method name="remove*" propagation="REQUIRED" />
    <tx:method name="assign*" propagation="REQUIRED" />
    <tx:method name="create*" propagation="REQUIRED" />
    <tx:method name="complete*" propagation="REQUIRED" />
    <tx:method name="finish*" propagation="REQUIRED" />
    <tx:method name="terminate*" propagation="REQUIRED" />
    <tx:method name="take*" propagation="REQUIRED" />
    <tx:method name="deploy*" propagation="REQUIRED" />
    <tx:method name="undeploy*" propagation="REQUIRED" />

    <tx:method name="get*" propagation="REQUIRED" read-only="true" />
    <tx:method name="find*" propagation="REQUIRED" read-only="true" />
    <tx:method name="query*" propagation="REQUIRED" read-only="true" />
    <tx:method name="search*" propagation="REQUIRED" read-only="true" />
    <tx:method name="is*" propagation="REQUIRED" read-only="true" />
    <tx:method name="*" propagation="REQUIRED" />
  </tx:attributes>

```



```
</tx:advice>
<aop:config>
    <aop:advisor advice-ref="txAdvice" pointcut="execution(*
org.snaker.engine.core.*.*(..)) or execution(*
org.snaker.framework.*.service..*.*(..)) or execution(*
org.snaker.modules.flow.service..*.*(..))"/>
</aop:config>
<aop:aspectj-autoproxy proxy-target-class="true" />
```

三、流程设计器

1. 安装插件

- 获取插件位置:

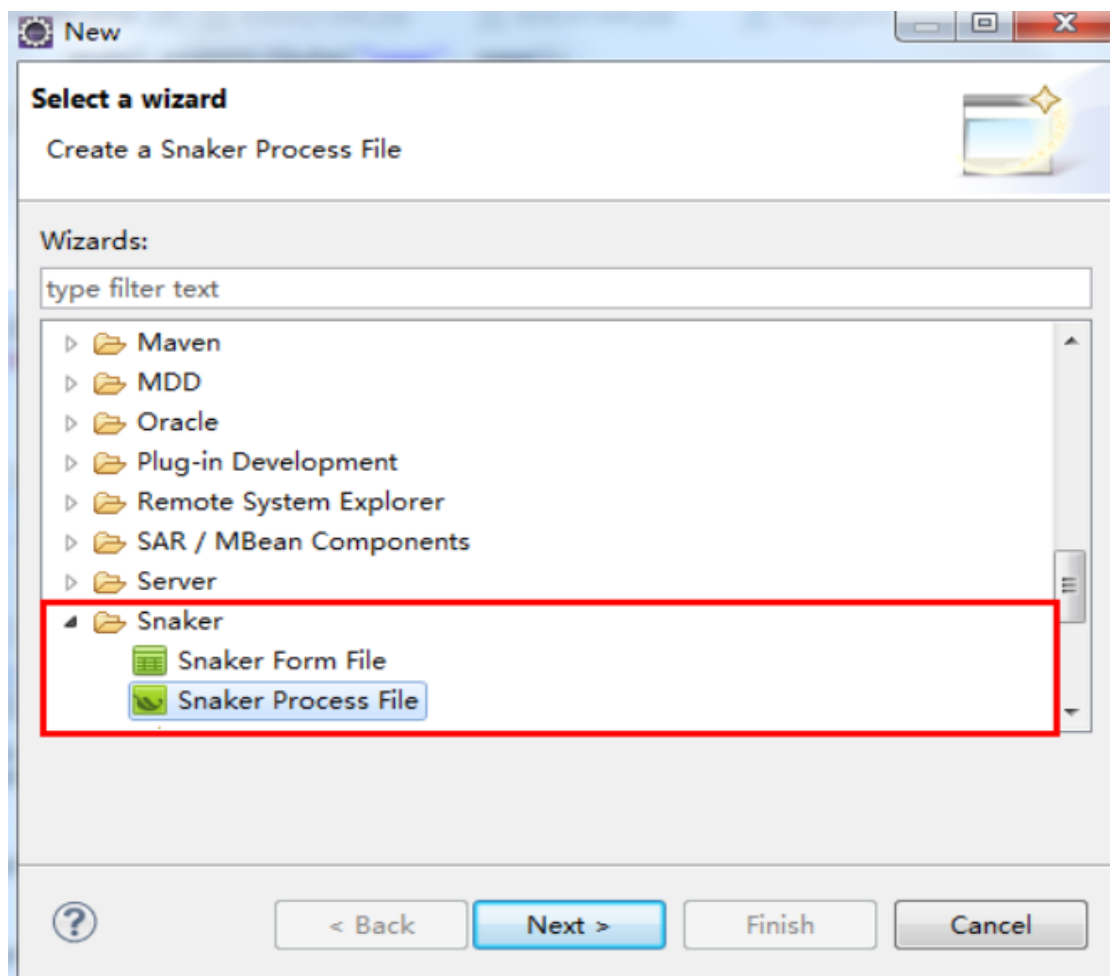
snaker-core-master/dist/plugins/snaker-designer_*.*.jar

- 安装插件:

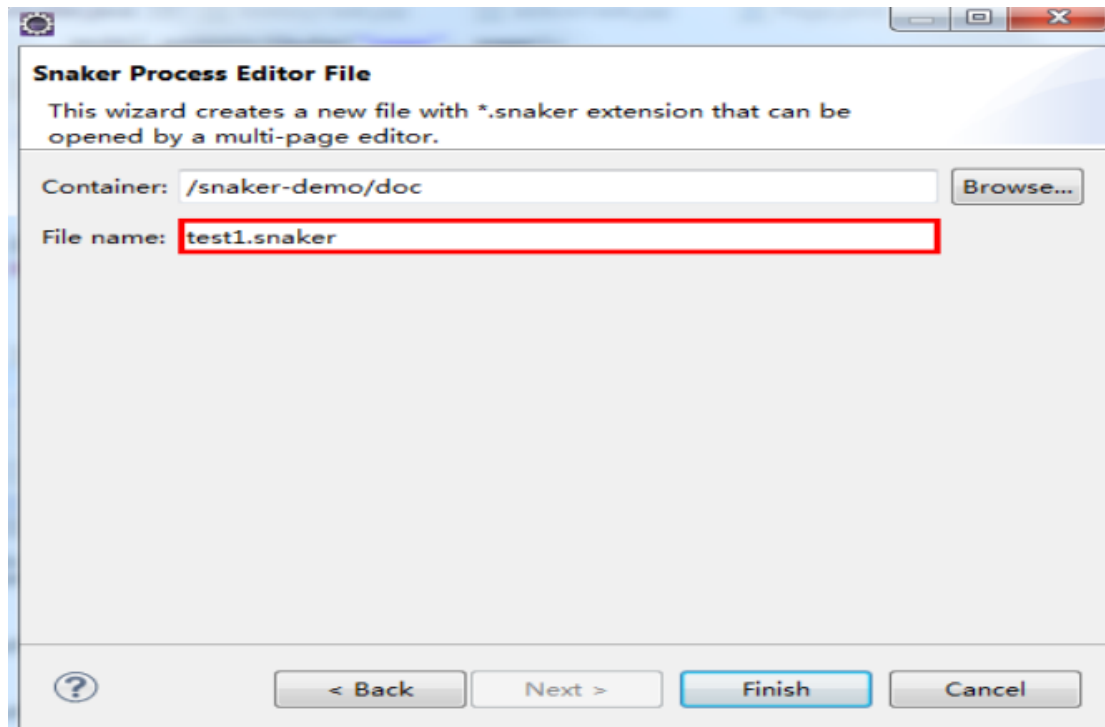
复制 snaker-designer_*.*.jar 到 eclipse 的 plugins 目录下, 重新启动 eclipse 即可(经过测试的版本有 eclipse4.2/4.3)

2. 定义流程

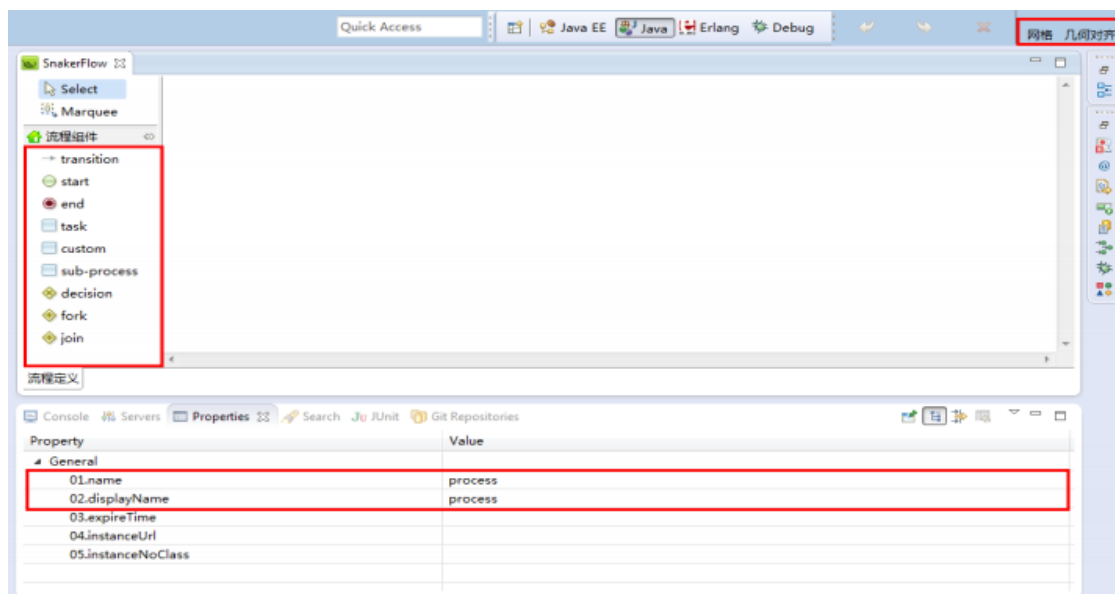
依次选择 File->New->Other->Snaker, 如果安装成功, 如下图所示:



选择 Snaker Process File 并输入文件名称，如下图所示：



点击 **Finish**，则打开流程设计器主界面，其中包括两大部分：流程组件、属性 **Properties** 视图，如下图所示：



3. 流程组件:

目前节点组件包括: **start**、**end**、**task**、**custom**、**sub-process**、**decision**、**fork**、**join**, 分别对应开始、结束、任务、自定义、子流程、决策、分支、合并组件模型.

4. 属性视图:

对组件模型设置属性, 包括常用的 **name**、**displayName** 等

5. 布局工具:

右上角的网格、几何对齐用于图形布局.



6. 属性说明

组件模型	属性	描述
通用属性	name	组件名称，模型内名称唯一
	displayName	组件中文显示名称，方便阅读
	preInterceptors	前置拦截器(节点模型)
	postInterceptors	后置拦截器(节点模型)
Process	instanceUrl	流程定义列表页面直接启动流程实例的 URL
	instanceNoClass	流程实例编号生成类
Transition	expr	决策选择 Decision 节点的输出变迁表达式
Task	form	用户参与的表单任务对应的 URL
	assignee	任务参与者变量
	assignmentHandler	任务参与者处理类
	taskType	任务类型（Main：主办；Aidant：协办）
	performType	任务参与类型（针对多个参与者） ANY 为其中一个参与者完成即往下流转； ALL 为所有参与者完成才往下流转
	expireTime	期望完成时间，设置表达式变量由参数传递
	autoExecute	是否自动执行：Y 自动；N 非自动
	callback	执行后的回调类
	reminderTime	提醒时间
	reminderRepeat	提醒次数
Custom	clazz	自定义节点的 Java 类路径，两种方式： 1.实现 IHandler 接口 2.无接口实现的普通 java 类，需要设置下面方法名称、参数属性
	methodName	定义需要执行的 java 类的方法名称
	args	定义传递的参数表达式
	var	定义返回值变量名称
SubProcess	processName	子流程名称（对应 process 的 name 属性）
Decision	expr	决策选择表达式

	handleClass	决策选择的处理类，实现 DecisionHandler 接口
--	-------------	--------------------------------

7. 自定义模型

Snaker 流程设计器中的组件模型是通过 xml 的配置加载的。配置文件请参考：

<http://git.oschina.net/yuqs/snaker-designer/blob/master/src/model-process.xml>

如 Decision 节点的配置如下：

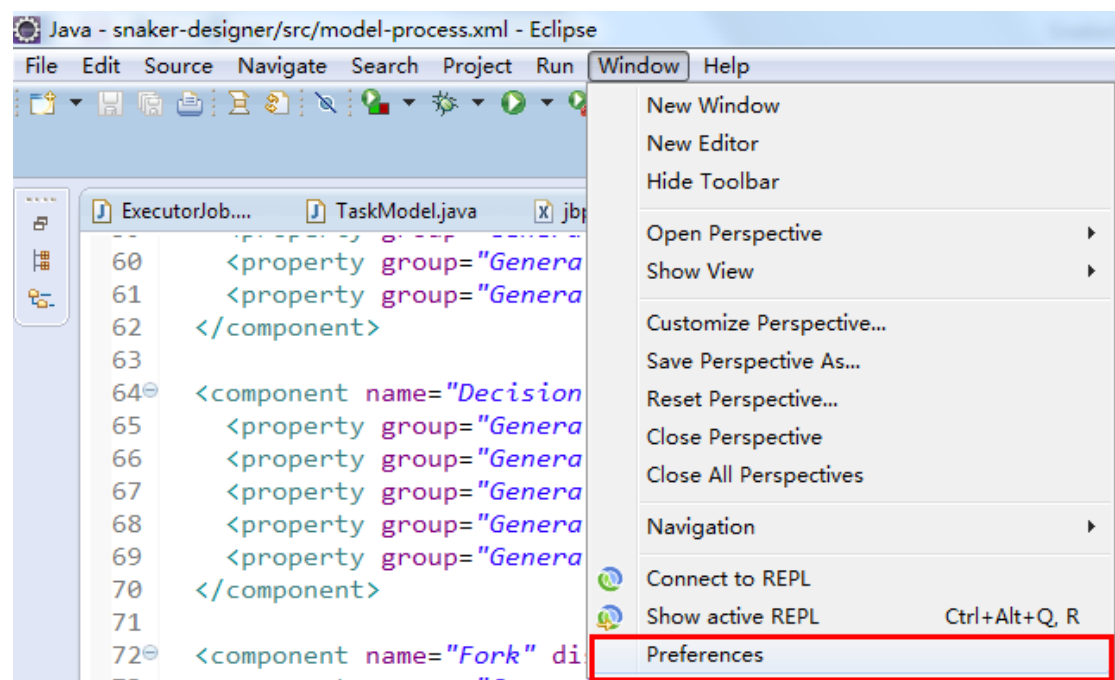
```
<component name="Decision" displayName="decision" group="process" describe="创建决策节点" iconSmall="icons/16/gateway_exclusive.png">
  <property group="General" name="name" defaultValue="decision"/>
  <property group="General" name="expr" defaultValue=""/>
  <property group="General" name="handleClass" defaultValue=""/>
  <property group="General" name="preInterceptors"/>
  <property group="General" name="postInterceptors"/>
</component>
```

配置内容包括属性、显示图标、节点名称等。

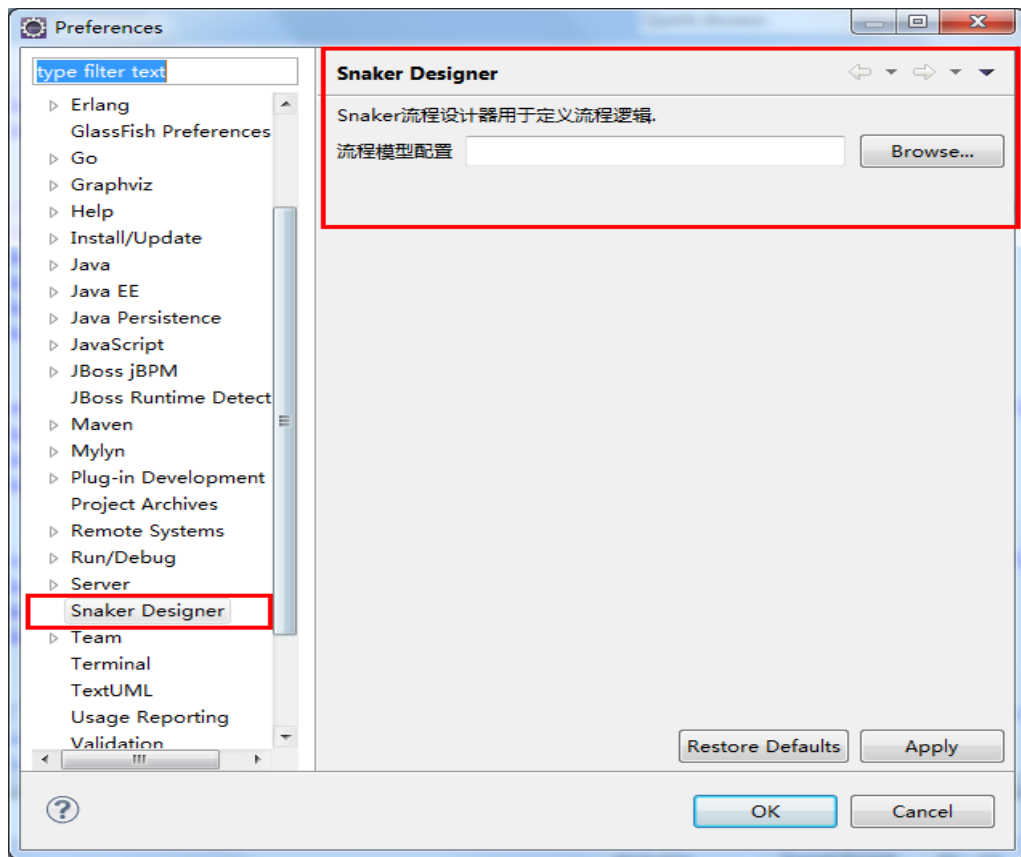
如果需要增加节点模型，只需要参考 component 的属性定义即可。model-process.xml 文件注释了 Sql 节点的定义，仅供参考。

自定义配置完成后，需要更新配置文件，步骤如下：

- 选择 Window->Preferences 打开首选项



- 选择配置面板：



- 在配置面板中，点击按钮“Browse...”选择自定义模型配置的 XML 文件即可。

四、API 说明

1. SnakerEngine:

```
//获取process服务
IProcessService process();

//获取查询服务
IQueryService query();

//获取实例服务
IOrderService order();

//获取任务服务
ITaskService task();

//获取管理服务
IManagerService manager();

//根据流程定义ID启动流程实例
Order startInstanceById(String id);

//根据流程定义ID，操作人ID启动流程实例
Order startInstanceById(String id, String operator);

//根据流程定义ID，操作人ID，参数列表启动流程实例
Order startInstanceById(String id, String operator, Map<String,
Object> args);

//根据流程名称启动流程实例
Order startInstanceByName(String name);

//根据流程名称、版本号启动流程实例
Order startInstanceByName(String name, Integer version);

//根据流程名称、版本号、操作人启动流程实例
Order startInstanceByName(String name, Integer version, String
operator);

//根据流程名称、版本号、操作人、参数列表启动流程实例
Order startInstanceByName(String name, Integer version, String
operator, Map<String, Object> args);
```



```

//根据父执行对象启动子流程实例
Order startInstanceByExecution(Execution execution);

//根据任务主键ID执行任务
List<Task> executeTask(String taskId);

//根据任务主键ID，操作人ID执行任务
List<Task> executeTask(String taskId, String operator);

//根据任务主键ID，操作人ID，参数列表执行任务
List<Task> executeTask(String taskId, String operator, Map<String,
Object> args);

//根据任务主键ID，操作人ID，参数列表执行任务，并且根据nodeName跳转到任意
节点
List<Task> executeAndJumpTask(String taskId, String operator,
Map<String, Object> args, String nodeName);

//根据流程实例ID，操作人ID，参数列表按照节点模型model创建新的自由任务
List<Task> createFreeTask(String orderId, String operator, Map<String,
Object> args, WorkModel model);

```

2. IProcessService:

```

//保存流程定义
void saveProcess(Process process);

//根据主键ID获取流程定义对象
Process getProcessById(String id);

//根据流程name获取流程定义对象
Process getProcessByName(String name);

//根据流程name、version获取流程定义对象
Process getProcessByVersion(String name, Integer version);

//根据给定的参数列表args查询process
List<Process> getProcesss(QueryFilter filter);

//根据给定的参数列表args分页查询process
List<Process> getProcesss(Page<Process> page, QueryFilter filter);

//根据InputStream输入流，部署流程定义
String deploy(InputStream input);

```

```
//根據InputStream輸入流，部署流程定义
void redeploy(String id, InputStream input);

//卸载指定的流程定义，只更新状态
void undeploy(String id);
```

3. IOrderService:

```
/**
 * 流程实例正常完成
 * @param orderId 流程实例id
 */
void complete(String orderId);

/**
 * 创建抄送实例
 * @param orderId 流程实例id
 * @param actorIds 参与者id
 * @since 1.5
 */
void createCCOrder(String orderId, String... actorIds);

/**
 * 流程实例强制终止
 * @param orderId 流程实例id
 */
void terminate(String orderId);

/**
 * 流程实例强制终止
 * @param orderId 流程实例id
 * @param operator 处理人员
 */
void terminate(String orderId, String operator);

/**
 * 更新抄送记录为已阅
 * @param orderId 流程实例id
 * @param actorIds 参与者id
 */
void updateCCStatus(String orderId, String... actorIds);

/**
 * 删除抄送记录
 * @param orderId 流程实例id
 * @param actorId 参与者id
 */
void deleteCCOrder(String orderId, String actorId);
```

4. ITaskService:

```
//完成指定的任务，删除活动任务记录，创建历史任务
Task complete(String taskId);

//完成指定的任务，删除活动任务记录，创建历史任务
Task complete(String taskId, String operator);

//根据任务主键ID，操作人ID完成任务
Task complete(String taskId, String operator, Map<String, Object>
args);

//根据任务主键ID，操作人ID提取任务
Task take(String taskId, String operator);

//向指定的任务id添加参与者
void addTaskActor(String taskId, String... actors);

//向指定的任务id添加参与者
void addTaskActor(String taskId, Integer performType, String...
actors);

//对指定的任务id删除参与者
void removeTaskActor(String taskId, String... actors);

//根据任务主键id、操作人撤回任务
Task withdrawTask(String taskId, String operator);
/**
 * 根据已有任务id、任务类型、参与者创建新的任务
 */
List<Task> createNewTask(String taskId, int taskType, String...
actors);
```

5. IQueryService:

```
/**
 * 根据流程实例ID获取流程实例对象
 * @param orderId 流程实例id
 * @return Order 流程实例对象
 */
Order getOrder(String orderId);
/**
 * 根据流程实例ID获取历史流程实例对象
 * @param orderId 历史流程实例id
```

```

    * @return HistoryOrder 历史流程实例对象
    */
HistoryOrder getHistOrder(String orderId);
/**
    * 根据任务ID获取任务对象
    * @param taskId 任务id
    * @return Task 任务对象
    */
Task getTask(String taskId);
/**
    * 根据任务ID获取历史任务对象
    * @param taskId 历史任务id
    * @return HistoryTask 历史任务对象
    */
HistoryTask getHistTask(String taskId);
/**
    * 根据任务ID获取活动任务参与者数组
    * @param taskId 任务id
    * @return String[] 参与者id数组
    */
String[] getTaskActorsByTaskId(String taskId);
/**
    * 根据任务ID获取历史任务参与者数组
    * @param taskId 历史任务id
    * @return String[] 历史参与者id数组
    */
String[] getHistoryTaskActorsByTaskId(String taskId);

/**
    * 根据filter查询活动任务
    * @param filter 查询过滤器
    * @return List<Task> 活动任务集合
    */
List<Task> getActiveTasks(QueryFilter filter);

/**
    * 根据filter分页查询活动任务
    * @param page 分页对象
    * @param filter 查询过滤器
    * @return List<Task> 活动任务集合
    */
List<Task> getActiveTasks(Page<Task> page, QueryFilter filter);

/**

```

```

    * 根据filter查询流程实例列表
    * @param filter 查询过滤器
    * @return List<Order> 活动实例集合
    */
List<Order> getActiveOrders(QueryFilter filter);

/**
 * 根据filter分页查询流程实例列表
 * @param page 分页对象
 * @param filter 查询过滤器
 * @return List<Order> 活动实例集合
 */
List<Order> getActiveOrders(Page<Order> page, QueryFilter filter);

/**
 * 根据filter查询历史流程实例
 * @param filter 查询过滤器
 * @return List<HistoryOrder> 历史实例集合
 */
List<HistoryOrder> getHistoryOrders(QueryFilter filter);

/**
 * 根据filter分页查询历史流程实例
 * @param page 分页对象
 * @param filter 查询过滤器
 * @return List<HistoryOrder> 历史实例集合
 */
List<HistoryOrder> getHistoryOrders(Page<HistoryOrder> page,
QueryFilter filter);

/**
 * 根据filter查询所有已完成的任务
 * @param filter 查询过滤器
 * @return List<HistoryTask> 历史任务集合
 */
List<HistoryTask> getHistoryTasks(QueryFilter filter);

/**
 * 根据filter分页查询已完成的历史任务
 * @param page 分页对象
 * @param filter 查询过滤器
 * @return List<HistoryTask> 历史任务集合
 */
List<HistoryTask> getHistoryTasks(Page<HistoryTask> page,

```

```
QueryFilter filter);
```

```
/**
 * 根据filter分页查询工作项（包含process、order、task三个实体的字段集合）
 * @param page 分页对象
 * @param filter 查询过滤器
 * @return List<WorkItem> 活动工作项集合
 */
List<WorkItem> getWorkItems(Page<WorkItem> page, QueryFilter filter);
```

```
/**
 * 根据filter分页查询抄送工作项（包含process、order）
 * @param page 分页对象
 * @param filter 查询过滤器
 * @return List<WorkItem> 抄送工作项集合
 */
List<HistoryOrder> getCCWorks(Page<HistoryOrder> page, QueryFilter filter);
```

```
/**
 * 根据filter分页查询已完成的历史任务项
 * @param page 分页对象
 * @param filter 查询过滤器
 * @return List<WorkItem> 历史工作项集合
 */
List<WorkItem> getHistoryWorkItems(Page<WorkItem> page, QueryFilter filter);
```

```
/**
 * 根据类型T、Sql语句、参数查询单个对象
 * @param T 类型
 * @param sql sql语句
 * @param args 参数列表
 * @return
 */
public <T> T nativeQueryObject(Class<T> T, String sql, Object... args);
```

```
/**
 * 根据类型T、Sql语句、参数查询列表对象
 * @param T 类型
 * @param sql sql语句
 * @param args 参数列表
```

```
    * @return
    */
    public <T> List<T> nativeQueryList(Class<T> T, String sql, Object...
args);
```

```
/**
 * 根据类型T、Sql语句、参数分页查询列表对象
 * @param page 分页对象
 * @param T 类型
 * @param sql sql语句
 * @param args 参数列表
 * @return
 */
    public <T> List<T> nativeQueryList(Page<T> page, Class<T> T, String
sql, Object... args);
```