1. 环境搭建

1.1 Ecology安装和启动

- 准备 `ecology` | `resin` | `jdk1.8[可直接使用已安装的]`
- 修改resin配置 `resin/config/resin.conf`

```
// 修改jdk路径：相对路径或者绝对路径
<javac compiler="C:\Program Files\Java\jdk1.8.0_151\bin\javac" args="-encoding UTF-8"/>

// 修改ecology路径
<web-app id="/" root-directory="F:\WEAVER\ecology">
```

- 修改resin启动配置 `resin/resinstart.bat`

其中`java_home`为`jdk`的路径，可以是相对路径也可以是绝对路径

```
set java_home=C:\Program Files\Java\jdk1.8.0_151
```

- 启动`resin`点击`resinstart.bat`
- 初始验证码文件路径 `ecology/WEB-INF/code.key`

1.2 主要目录介绍

- `ecology/classbean`：存放编译后的class文件
- `ecology/log`：系统中日志存放目录
- `ecology/WEB-INF/prop`：`web`配置文件目录
- `ecology/WEB-INF/lib`：系统`jar`包路径

1.3 数据库配置

> 如果数据库与Ecology不在同一台服务器上，则可以修改数据库配置文件中的数据库配置。
>
> 数据库配置文件路径：`ecology/WEB-INF/prop/weaver.properties`

```
# sqlServer
DriverClasses = com.microsoft.sqlserver.jdbc.SQLServerDriver
ecology.url = jdbc:sqlserver://host:port;DatabaseName=dbname
ecology.user = username
ecology.password = password
```

```
# oracle
DriverClasses = oracle.jdbc.OracleDriver
ecology.url = jdbc:oracle:thin:@host:port:ecology
ecology.user = username
cology.password = password
```

2. E9常见表结构

2.1 流程相关数据存储表

数据库表名	中文说明	备注
workflow_base	流程基本信息	isbill=1
workflow_bill	流程表单信息	id > 0固定表名 id < 0 动态生成
workflow_billfield	表单字段信息	
workflow_billdetailtable	表单明细表	
workflow_nodebase	节点信息	
workflow_flownode	流程节点信息	
workflow_nodelink	流程出口信息	
workflow_nodegroup	节点操作人信息	
workflow_groupdetail	节点操作人详情	
workflow_requestbase	请求基本信息	
workflow_currentoperator	请求节点操作人	
workflow_requestlog	请求签字意见	
workflow_nownode	请求当前节点	
workflow_browserurl	系统浏览按钮信息	
workflow_selectitem	下拉框信息	

2.2 人力资源相关数据存储

表名	说明	备注
HrmResource	人力资源基本信息表	-
HrmResource_online	人员在线信息表	-
HrmResourceManager	系统管理员信息表	-
HrmDepartment	人力资源部门表	-
HrmDepartmentDefined	人力资源部门自定义字段信息表	-
HrmSubCompany	人力资源分部表	-
hrmroles	角色信息表	-
hrmrolemembers	角色人员	-
hrmjobtitles	岗位信息表	-

3. 前端开发

> `ecology9`前端上采用`react`+`antd`+`mobx`+`react-router`等框架实现的单页面应用

3.1 流程开发

> 流程表单前端接口：E9流程表单前端接口API

- 所有接口统一封装在全局对象`window.WfForm`中
- 表单字段相关操作，不推荐使用jQuery，禁止原生JS直接操作DOM结构！
- 在开发过程中，推荐都使用API接口操作，由产品统一运维；同时使用API才能完整的兼容移动终端

3.2 建模开发

3.2.1 布局代码块

> 建模布局代码块使用上与流程表单的代码块基本一致，区别在于接口的SDK不同，建模表单的所有接口统一封装在全局对象`window.ModeForm`中。

3.2.2 自定义按钮

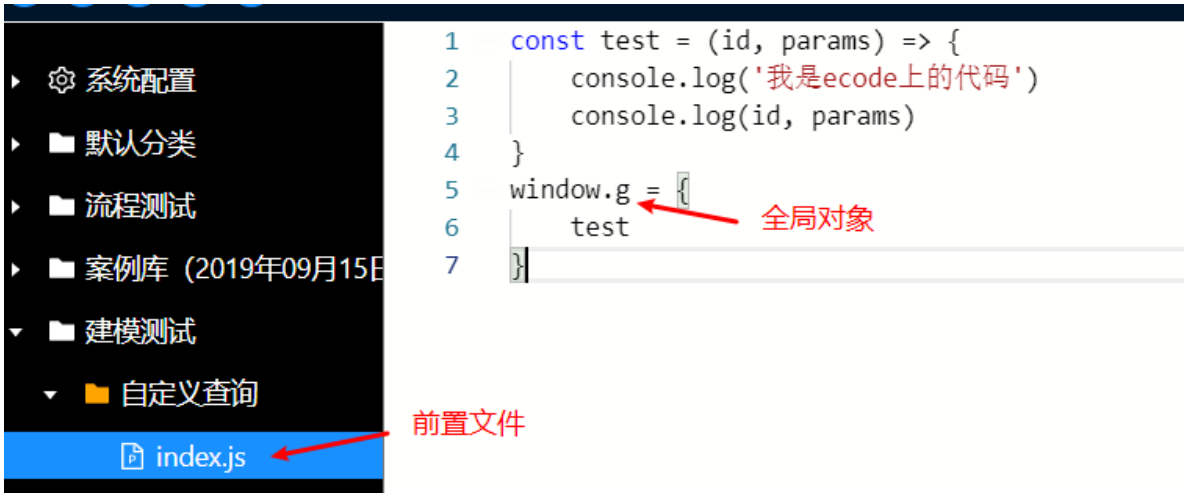
> 后端应用中心 -> 建模引擎 -> 查询
>
> 任选一个查询页面 -> 自定义按钮 -> 右键 -> 新建

- 方法体中存在多行代码时，每个语句必须以`;`结尾；否则会报错！
- `params`的值等于`'field1+field2+field3'`这个值是一个字符串
- `id`指的是数据`ID`

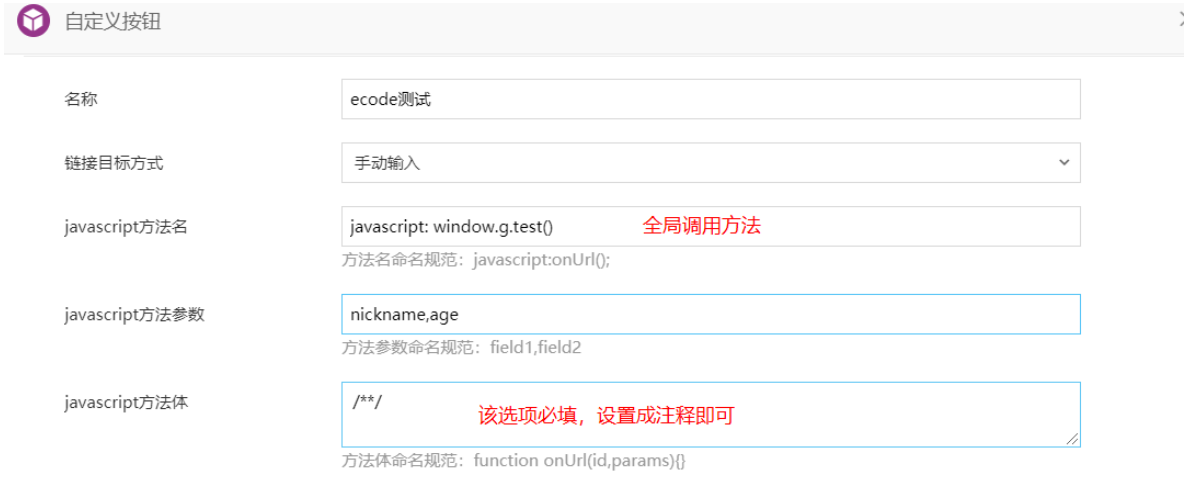
> 前端显示

- **配置`ecode`使用**

> 新建前置文件`index.js`并将方法挂到全局对象`window.g`下



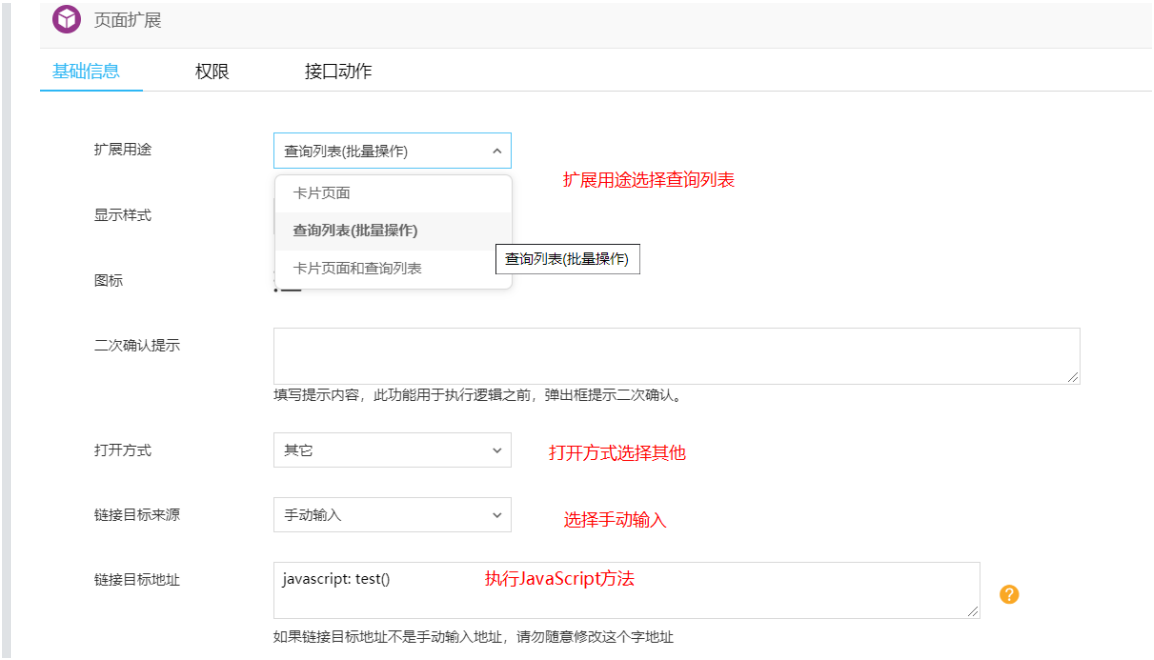
新建自定义按钮



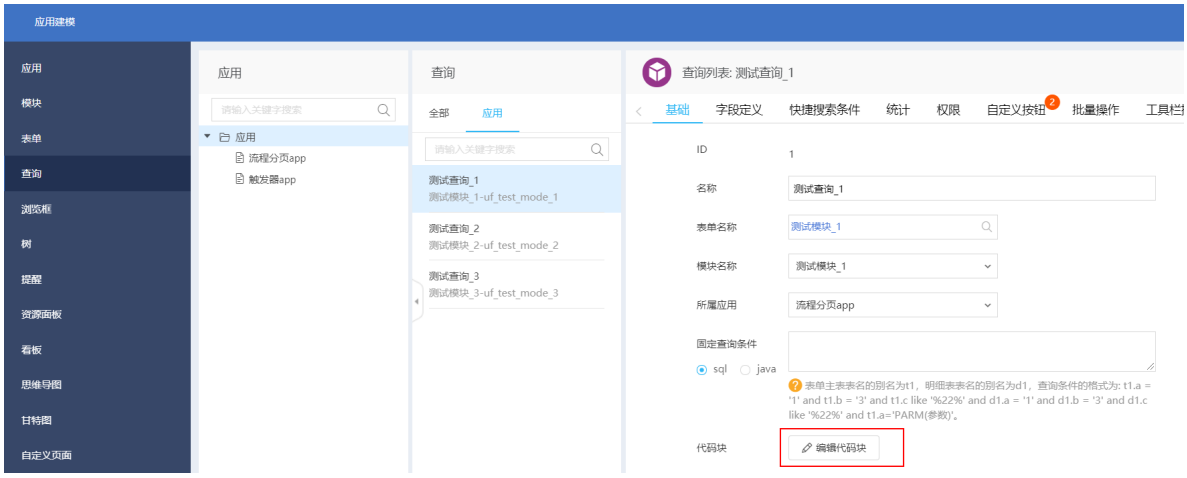
3.2.3 页面扩展

后端应用中心 -> 建模引擎 -> 模块

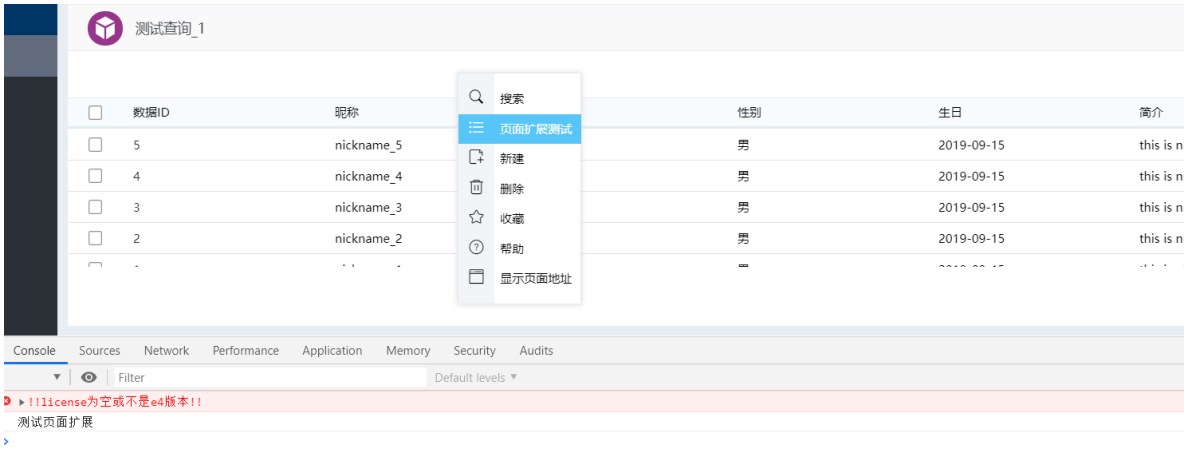
任选一个模块 -> 页面扩展 -> 右键 -> 新建



- 扩展用途：卡片页面、查询列表（批量操作）、卡片页面和查询列表
  - 卡片页面：可以设置页面扩展显示在卡片信息页面，可以选择在新建页面、编辑页面、查看页面显示页面扩展。
  - 查询列表（批量操作）：设置在查询列表时，则会在引用该模块的查询列表的批量操作中显示页面扩展项，在批量操作中勾选后会在前台列表中显示对应的页面扩展项。
  - 卡片页面和查询列表：可以设置页面扩展项既显示在对应的卡片页面又显示在查询列表（批量操作）中。
- `javascript:test()`：该方法可以在 `建模引擎 -> 查询 -> 该模块的查询列表 -> 编辑代码块` 中定义



前端按钮测试如下



- 页面扩展同样可以配置 `ecode` 使用，将**链接目标地址**改成: `javascript: window.g.test()` 即可，建议这样做，方便后续代码维护。

3.3 Ecode在线编辑

> 使用已经封装好的E9组件库进行页面开发或者页面改写会更加便捷迅速。

3.4 前端脚手架开发

---

4. 后端开发

> 代码案例：E9Demo

4.1 `Java`项目环境搭建

4.1.1 `web.xml` 部分配置

> `API` 接口的`xml`配置

```
<!-- ecology/WEB-INF/web.xml -->
<servlet>
    <servlet-name>restservlet</servlet-name>
    <servlet-
class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <init-param>
        <param-name>com.sun.jersey.config.property.packages</param-name>
        <!-- 设置jersey的扫包路径；可配置多个，以；隔开 -->
        <param-value>com.cloudstore;com.api</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>restservlet</servlet-name>
    <url-pattern>/api/*</url-pattern>
    <!-- 自定义接口前缀，绕开过滤器拦截 -->
    <url-pattern>/port/*</url-pattern>
</servlet-mapping>
```

注意： /api/* 的请求必须在用户登录时才能进行访问。否则会被过滤器过滤。其 web.xml 配置 SessionFilter 实
现了过滤拦截。当然也可以自定义接口前缀，如上配置了 /port/*

```
<filter>
    <filter-name>SessionFilter</filter-name>
    <filter-class>com.cloudstore.dev.api.service.SessionFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>SessionFilter</filter-name>
    <!-- 拦截 /api/* 请求，做是否登录以及其他逻辑判断 -->
    <url-pattern>/api/*</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>SessionFilter</filter-name>
    <url-pattern>/page/interfaces/*.jsp</url-pattern>
</filter-mapping>
```

4.1.2 项目环境搭建

> 使用 `IDEA` 编辑器

- 创建 `Java` 项目
- 添加所需依赖

File -> Project Structure -> Project Settings -> Libraries

需要添加的`ecology`的依赖路径有: `ecology/WEB-INF/lib`; `resin/lib`; `ecology/classbean`;

其中`classbean`是必须要引入的, 其他两个按需引入

- 编译`Java`文件将编译后的`class`文件放入`ecology/classbean/`目录下即可

注意: `ecology/classbean` 最好备份, 因为`IDEA`在编译的时候可能会清除掉已有的`classbean`

后端项目结构以及开发案例详见手册: E9后端开发指南

4.1.3 Java项目结构

> 一般情况下遵循以下项目结构进行后端开发

其中，`com.api.wcode`和`com.engine.wcode`中的 `wcode`为自定义包名，不与已有的包重复即可。

```
com.api.wcode
        |-- web         接口定义层
com.engine.wcode
        |-- web         接口实现层
        |-- service     服务定义层
            |-- impl    服务实现层
        |-- domain      数据层
            |-- cmd     原子操作层定义
            |-- dao     Dao层
        |-- mapper      mapper接口定义层
```

4.2 后端API路径规范

模块	文件路径（含下级）	接口访问地址
流程	com.api.workflow	/api/workflow/...
门户	com.api.portal	/api/portal/...
文档	com.api.doc	/api/doc/...
建模	com.api.formmode	/api/formmode/...
移动建模	com.api.mobilemode	/api/mobilemode/...
会议	com.api.meeting	/api/meeting/...
人力	com.api.hrm	/api/hrm/...
财务	com.api.fna	/api/fna/...
项目	com.api.prj	/api/prj/...
公文	com.api.odoc	/api/odoc/...
集成	com.api.integration	/api/integration/...
微博	com.api.blog	/api/blog/...

4.3 自定义`Java`接口

4.3.1 流程节点前后附加操作

> 在节点前后附加操作中可设置接口动作，完成流程自定义附加操作
>
> 接口动作标识不能重复；接口动作类文件必须是类全名，该类必须实现接`weaver.interfaces.workflow.action`方法`public String execute(RequestInfo request)`

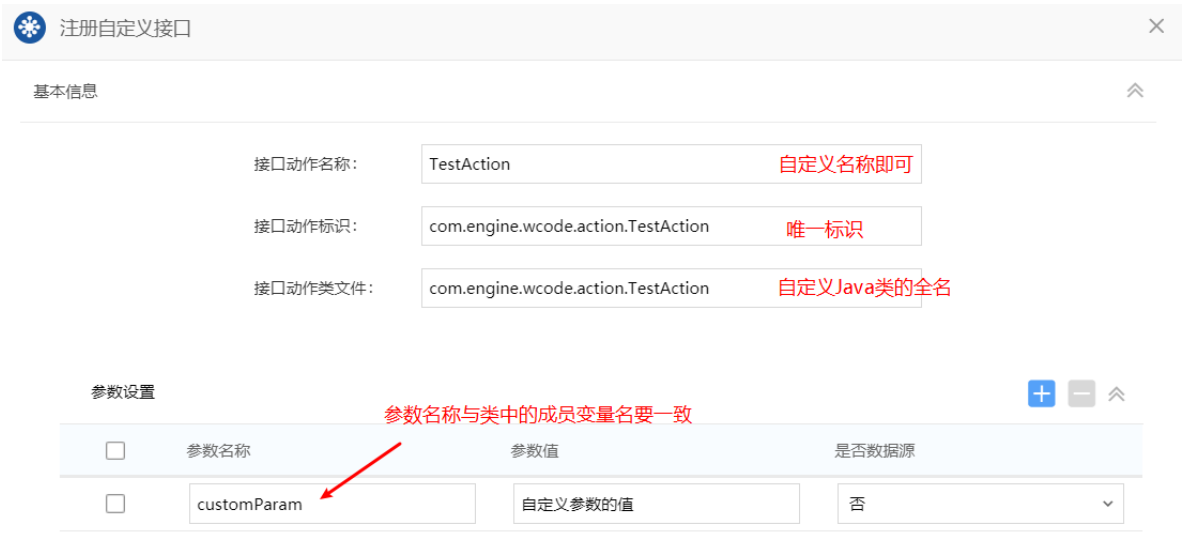
参考代码如下

```
1  package com.engine.wcode.action;
2
3  import com.weaver.general.Util;
4  import org.slf4j.Logger;
5  import org.slf4j.LoggerFactory;
6  import weaver.hrm.User;
7  import weaver.interfaces.workflow.action.Action;
8  import weaver.soa.workflow.request.*;
9
10 public class TestAction implements Action {
11
12     private String customParam; //自定义参数
13     private final Logger logger =
   LoggerFactory.getLogger(TestAction.class);
14
```

```
    @Override
    public String execute(RequestInfo requestInfo) {
        logger.debug("进入action requestid = {}",
requestInfo.getRequestid());
        showCurrentForm(requestInfo);

        showFormProperty(requestInfo);

        showDetailsTables(requestInfo);

        logger.debug("Action 执行完成，传入自定义参数：{}",
this.getCustomParam());
//        requestInfo.getRequestManager().setMessagecontent("返回自定义的错误
消息");
//        requestInfo.getRequestManager().setMessageid("自定义消息ID");
//        return FAILURE_AND_CONTINUE;  // 注释的三句话一起使用才有效果!
        return SUCCESS;
    }

    private void showCurrentForm(RequestInfo requestInfo) {
        String requestid = requestInfo.getRequestid(); // 请求ID
        String requestLevel = requestInfo.getRequestlevel(); // 请求紧急程度
        // 当前操作类型 submit:提交/reject:退回
        String src = requestInfo.getRequestManager().getSrc();
        // 流程ID
        String workFlowId = requestInfo.getWorkflowid();
        // 表单名称
        String tableName =
requestInfo.getRequestManager().getBillTableName();
        // 表单数据ID
        int bill_id = requestInfo.getRequestManager().getBillid();
        // 获取当前操作用户对象
        User user = requestInfo.getRequestManager().getUser();
        // 请求标题
        String requestName =
 requestInfo.getRequestManager().getRequestname();
        // 当前用户提交时的签字意见
        String remark = requestInfo.getRequestManager().getRemark();
        // 表单ID
        int form_id = requestInfo.getRequestManager().getFormid();
        // 是否是自定义表单
        int isbill = requestInfo.getRequestManager().getIsbill();

        logger.debug("requestid: {}", requestid);
        logger.debug("requestLevel: {}", requestLevel);
        logger.debug("src: {}", src);
        logger.debug("workFlowId: {}", workFlowId);
        logger.debug("tableName: {}", tableName);
        logger.debug("bill_id: {}", bill_id);
        logger.debug("user: {}", user);
        logger.debug("requestName: {}", requestName);
        logger.debug("remark: {}", remark);
        logger.debug("form_id: {}", form_id);
        logger.debug("isbill: {}", isbill);
    }
    /**
     * 获取主表数据
     */
```

```
68      private void showFormProperty(RequestInfo requestInfo) {
69          logger.debug("获取主表数据 ...");
70          // 获取表单主字段值
71          Property[] properties =
requestInfo.getMainTableInfo().getProperty();
72          for (Property property : properties) {
73              // 主字段名称
74              String name = property.getName();
75              // 主字段对应的值
76              String value = Util.null2String(property.getValue());
77              logger.debug("name: {}, value: {}", name, value);
78          }
79      }
80
81      /**
82       * 取明细数据
83       */
84      private void showDetailsTables(RequestInfo requestInfo) {
85          logger.debug("获取所有明细表数据 ...");
86          // 获取所有明细表
87          DetailTable[] detailTables =
requestInfo.getDetailTableInfo().getDetailTable();
88          if (detailTables.length > 0) {
89              for (DetailTable table: detailTables) {
90                  // 当前明细表的所有数据，按行存储
91                  Row[] rows = table.getRow();
92                  for (Row row: rows) {
93                      // 每行数据再按列存储
94                      Cell[] cells = row.getCell();
95                      for (Cell cell: cells) {
96                          // 明细字段名称
97                          String name = cell.getName();
98                          // 明细字段的值
99                          String value = cell.getValue();
100                         logger.debug("name: {}, value: {}", name, value);
101                     }
102                 }
103             }
104         }
105     }
106
107     public String getCustomParam() {
108         return customParam;
109     }
110
111     public void setCustomParam(String customParam) {
112         this.customParam = customParam;
113     }
114 }
```

> 配置：后端应用中心 -> 流程引擎 -> 路径管理 -> 路径设置
>
> 任选一个流程 -> 流程设置 -> 节点信息
>
> 任选一个节点 -> 节点前 / 节点后附加操作



4.3.2 页面扩展接口

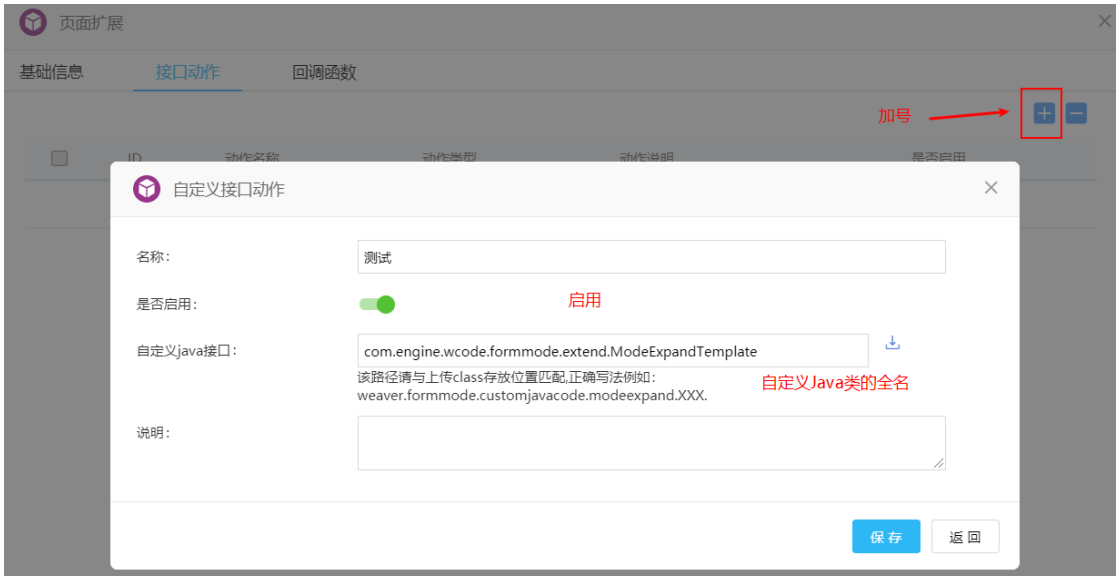
> 页面扩展 -> 接口动作 -> 自定义接口动作
>
> 执行页面扩展的后续操作，通过配置自定义 `Java` 接口动作类实现。
>
> 接口动作类文件必须是类全名。该类必须继承 `weaver.formmode.customjavacode.AbstractModeExpandJavaCode` 方法 `public void doModeExpand(Map<String, Object> param)`

参考代码

```
package com.engine.wcode.formmode.extend;

import weaver.conn.RecordSet;
import weaver.general.Util;
import weaver.hrm.User;
import weaver.soa.workflow.request.RequestInfo;
import weaver.formmode.customjavacode.AbstractModeExpandJavaCode;

import java.util.Map;

public class ModeExpandTemplate extends AbstractModeExpandJavaCode {

    @Override
    public void doModeExpand(Map<String, Object> param) throws Exception {
        // 当前用户
        User user = (User) param.get("user");
        int billid = -1; // 数据id
        int modeid = -1; // 模块id
        RequestInfo requestInfo = (RequestInfo) param.get("RequestInfo");
        if (requestInfo != null) {
```

```
            billid = Util.getIntValue(requestInfo.getRequestid());
            modeid = Util.getIntValue(requestInfo.getWorkflowid());
            if (billid > 0 && modeid > 0) {
                RecordSet rs = new RecordSet();
                //------请在下面编写业务逻辑代码------
            }
        }
    }
}
```

> 配置：后端应用中心 -> 建模引擎 -> 模块
>
> 任选一个模块 -> 页面扩展 -> 任选一个扩展名称 -> 接口动作 -> 点击 `+` 号 -> 自定义接口动作



4.3.3 计划任务

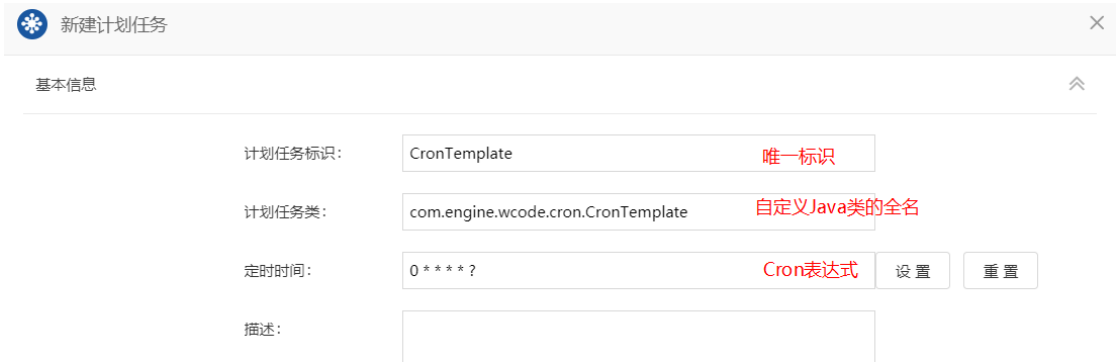
> 通过配置自定义 `Java` 接口的实现类，定时执行相应的代码

- 按照设定的时间定时执行任务，计划任务标识不能重复
- 计划任务类必须是类的全名，该类必须继承 `weaver.interfaces.schedule.BaseCronJob` 类,重写方法 `public void execute() {}`
- 时间格式按 `Cron` 表达式的定义

参考代码

```
package com.engine.wcode.cron;

import weaver.interfaces.schedule.BaseCronJob;

public class CronTemplate extends BaseCronJob {

    @Override
    public void execute() {
        //------请在下面编写业务逻辑代码------
    }
}
```

> 配置：后端应用中心 -> 集成中心 -> 计划任务 -> 任务列表 -> 新建



通过计划任务列表的每个计划任务的自定义按钮，可以对每个任务进行状态操作，具体使用如下所示

状态详解：

1. 启用: 计划任务将根据Cron表达式执行;
2. 禁用: 计划任务将不再执行，重启服务也不会再次执行;
3. 暂停: 针对计划任务进行停止，重启服务将恢复正常状态;
4. 恢复: 针对暂停状态的计划任务进行恢复，恢复后计划任务将继续执行;
5. 执行: 单次执行计划任务，不影响Cron表达式周期执行;
6. 测试: 检查填写的计划任务类是否符合规范（继承weaver.interfaces.schedule.BaseCronJob类,重写方法public void execute() {}）

4.3.4 自定义按钮接口

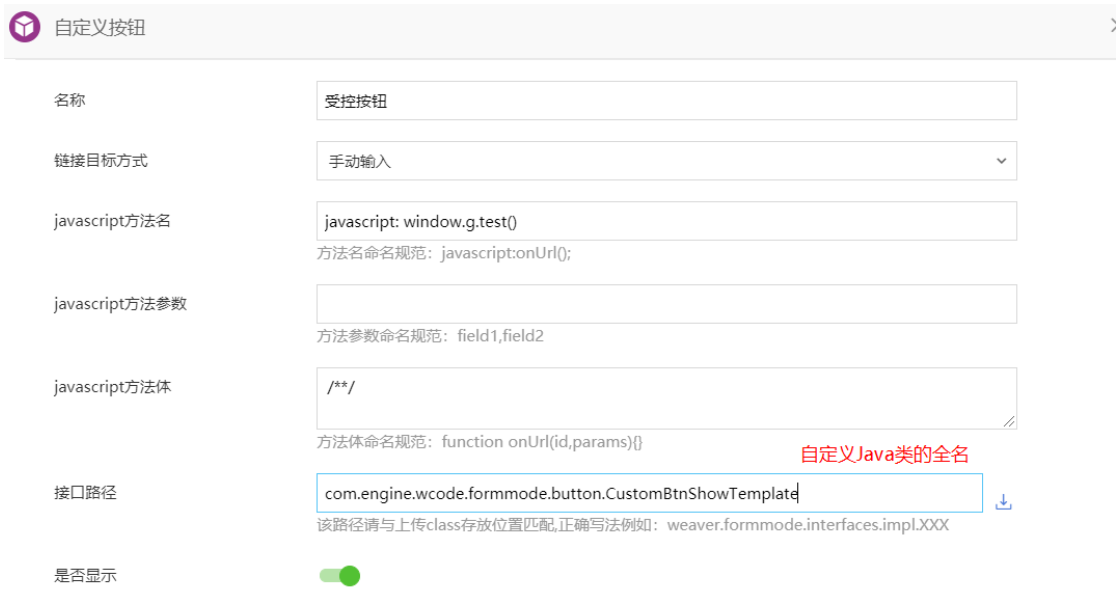
> 通过配置自定义`Java`类，判断自定义按钮在查询列表中是否显示

参考代码

```
package com.engine.wcode.formmode.button;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import weaver.formmode.interfaces.PopedomCommonAction;

public class CustomBtnShowTemplate implements PopedomCommonAction {

    private Logger logger =
LoggerFactory.getLogger(CustomBtnShowTemplate.class);

    /**
     * 得到是否显示操作项
     * @param modeid 模块id
     * @param customid 查询列表id
     * @param uid 当前用户id
```

```
     * @param billid 表单数据id
     * @param buttonname 按钮名称
     * @retrun "true"或者"false"true显示/false不显示
     */
    @Override
    public String getIsDisplayOperation(String modeid, String
customid,String uid, String billid, String buttonname) {
        logger.debug("modeId: {}", modeid);
        logger.debug("customId: {}", customid);
        logger.debug("uid: {}", uid);
        logger.debug("billId: {}", billid);
        logger.debug("buttonname: {}", buttonname);
        return "false";
    }
}
```

> 配置：后端应用中心 -> 建模引擎 -> 查询
>
> 任选一个查询列表 -> 自定义按钮 -> 右键 -> 新建



前端查询列表中，由于接口中返回false，则 受控按钮 不显示

4.4 数据库操作

4.4.1 CURD

> 使用 `weaver.conn.RecordSet` 可以对数据库进行 `CURD` 等数据库操作

参考代码：

```
RecordSet rs = new RecordSet();
String sql = "select loginid, lastname from hrmresource where id=?";
// 防止sql注入, objects 为动态参数
rs.executeQuery(sql, 2);
if (rs.next()) {
    String loginid = rs.getString("loginid");
    String lastname = rs.getString("lastname");
}
String updateSql = "update hrmresource lastname=? where id=?";
// 返回是否更新成功
boolean bool = rs.executeUpdate(sql, "孙悟空", 2);
```

4.4.2 使用事务

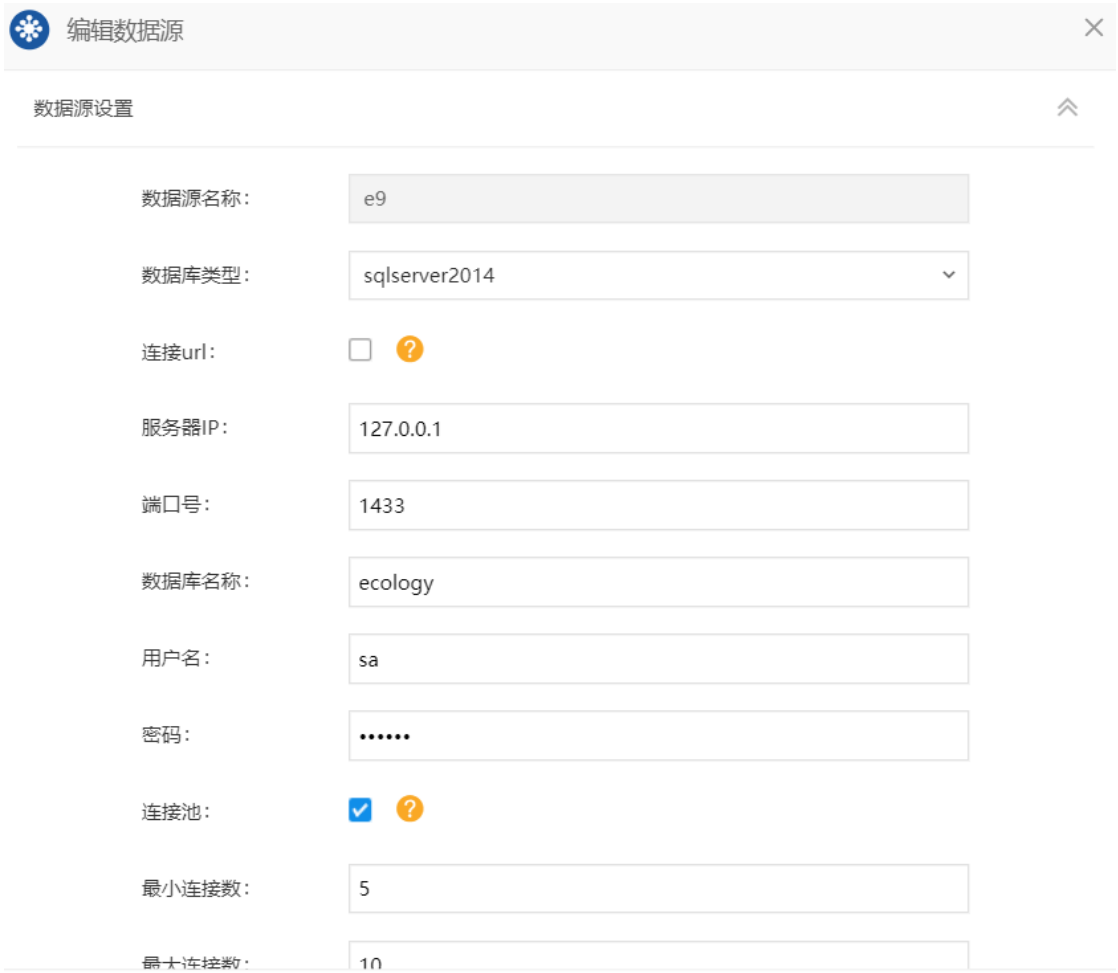
> 使用`weaver.conn.RecordSetTrans`可以对数据库进行事务操作

参考代码

```
RecordSetTrans rst = new RecordSetTrans();
// 开启事务
rst.setAutoCommit(false);
String sql = "update hrmresource lastname=? where id=?";
try {
    int a = 1/0;
    rst.executeUpdate(sql, "猪八戒", 2);
    // 提交事务
    rst.commit();
} catch (Exception e) {
    e.printStackTrace();
    // 事务回滚
    rst.rollback();
}
```

4.4.3 Mybatis的使用

> 目前E9没有成熟的使用Mybatis的方案，以下是通过自己的研究给出一个不成熟的方案，有`Bug`概不负责！
>
> 支持多数据源，数据源配置如下：
>
> 后端应用中心 -> 集成中心-> 数据源设置 -> 新建



> `xml` 方式开发

创建 `mapper` 接口

```java
package com.engine.wcode.mapper;

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

public interface TestMapper {
    List<Map<String, String>> selectAll();
}
```

在 `../ecology/WEB-INF/config/mapper/` 路径下创建 `test.xml`

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.engine.wcode.mapper.TestMapper">
  <select id="selectAll" resultType="java.util.HashMap" >
    select loginid, password, lastname from hrmresource
  </select>
</mapper>
```

> **注解方式开发**
>
> 与`xml`方式相比,使用注解开发的方便之处在于不用写`mapper.xml`文件

创建 `mapper` 接口

```java
package com.engine.wcode.mapper;

import org.apache.ibatis.annotations.Select;

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

public interface HrmMapper {
    @Select("select loginid, password, lastname from hrmresource")
    List<Map<String, String>> selectHrm();
}
```

> 使用 `Mybatis` 操作数据库

- 获取 `Mapper`

```java
package com.engine.wcode.db;

import org.apache.ibatis.session.Configuration;
import org.apache.ibatis.session.ExecutorType;
import org.apache.ibatis.session.SqlSession;
import weaver.conn.ConnectionPool;
import weaver.conn.WeaverConnection;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import static weaver.conn.mybatis.MyBatisFactory.sqlSessionFactory;

/**
* @author Y-Aron
* @create 2019/5/8
*/
public class MapperUtil {

    private static final Map<Class, Object> cacheMapper = new
ConcurrentHashMap<>(1);

    private static final Map<String, SqlSession> cacheSqlSession = new
ConcurrentHashMap<>(1);

    public static <T> T getMapper(Class<T> clazz) {
        return MapperUtil.getMapper(clazz, false);
    }

    public static <T> T getMapper(Class<T> clazz, boolean enableCache) {
        return MapperUtil.getMapper(clazz, null, ExecutorType.SIMPLE,
enableCache);
    }

    public static <T> T getMapper(Class<T> clazz, String dataSource) {
        return MapperUtil.getMapper(clazz, dataSource, false);
    }

    public static <T> T getMapper(Class<T> clazz, String dataSource,
boolean enableCache) {
        return MapperUtil.getMapper(clazz, dataSource, ExecutorType.SIMPLE,
enableCache);
    }
```

```
37
38      public static <T> T getMapper(Class<T> clazz, String dataSource,
    ExecutorType executorType, boolean enableCache) {
39          String threadName = Thread.currentThread().getName();
40          SqlSession sqlSession = cacheSqlSession.get(threadName);
41          if (sqlSession == null) {
42              ConnectionPool pool = ConnectionPool.getInstance();
43              WeaverConnection connection = pool.getConnection(dataSource);
44              Configuration config = sqlSessionFactory.getConfiguration();
45              if (executorType == null) {
46                  executorType = config.getDefaultExecutorType();
47              }
48              if (enableCache) {
49                  config.setLocalCacheScope(LocalCacheScope.STATEMENT);
50              }
51              sqlSession = sqlSessionFactory.openSession(executorType,
    connection);
52          }
53          cacheSqlSession.put(threadName, sqlSession);
54          return MapperUtil.getMapper(clazz, sqlSession);
55      }
56
57      public static <T> T getMapper(Class<T> clazz, SqlSession sqlSession) {
58          if (cacheMapper.containsKey(clazz)) {
59              //noinspection unchecked
60              return (T) cacheMapper.get(clazz);
61          }
62          Configuration config = sqlSession.getConfiguration();
63          if (!config.hasMapper(clazz)) {
64              config.addMapper(clazz);
65          }
66          T mapper = sqlSession.getMapper(clazz);
67          cacheMapper.put(clazz, mapper);
68          return mapper;
69      }
70
71      private static SqlSession getCurrentSqlSession() {
72          if (cacheSqlSession.size() == 0) return null;
73          return cacheSqlSession.get(Thread.currentThread().getName());
74      }
75  }
```

- 数据库操作

```
1  HrmMapper mapper = MapperUtil.getMapper(HrmMapper.class);
2  mapper.selectHrm();
```

4.5 缓存SDK

> 缓存基类：`Util_DataCache`

方法名称	方法作用
getObjVal(String name)	从所有缓存获取 缓存数据 (主要函数)
setObjVal(String name, Object value)	设置所有缓存数据 (主要函数)
setObjVal(String name, Object value,int seconds)	设置所有缓存数据 支持 超时自动消失 (主要函数)
containsKey(String name)	判断该键名的所有缓存是否存在
clearVal(String name)	清除该键名的所有缓存
setObjValWithEh(String name,Object value)	设置本地缓存 (特定情况下使用)
getObjValWithEh(String name)	获取本地缓存 （特定情况下使用）
setObjValWithRedis(String name,Object value)	设置Redis缓存 需要自己释放数据(特定情况下使用)
setObjValWithRedis(String name,Object value,int seconds)	单独设置Redis缓存 超时时间(s)后释放数据(特定情况下使用)
getObjValWithRedis(String name)	单独获取Redis缓存(特定情况下使用)
containsKeylWithEh(String name)	判断本地缓存是否存在该键名(特定情况下使用)
clearValWithEh(String name)	清除本地缓存(特定情况下使用)
containsKeyWithRedis(String name)	判断Redis上是否存在该键名(特定情况下使用)
clearValWithRedis(String name)	清除Redis缓存

> 检查页面

`chechRedis.jsp` 检查 `Redis` 环境的状态

`getRedis.jsp` 检查 `DataKey` 的数据

注意数据变更后必须再次执行 `setObjVal` 把数据推送到 `Redis`

```
import com.cloudstore.dev.api.util.Util_DataCache;
public Map<String,String> refreshDataFormDB() {
    Map<String,String> map = new HashMap<String, String>();
    Map<String,String> mapdb = getSystemIfo("y");
    map.putAll(mapdb);
    if(mapdb.size()>0) {
        Util_DataCache.setObjVal(em_url, mapdb.get(em_url));
        Util_DataCache.setObjVal(em_corpid, mapdb.get(em_corpid));
        Util_DataCache.setObjVal(accesstoken,mapdb.get(accesstoken));
        Util_DataCache.setObjVal(ec_id,mapdb.get(ec_id));
        Util_DataCache.setObjVal(ec_url, mapdb.get(ec_url));
        Util_DataCache.setObjVal(ec_name, mapdb.get(ec_name));
        Util_DataCache.setObjVal(rsa_pub, mapdb.get(rsa_pub));
        Util_DataCache.setObjVal(ec_version, mapdb.get(ec_version));
        Util_DataCache.setObjVal(ec_iscluster, mapdb.get(ec_iscluster));
        Util_DataCache.setObjVal(em_url, mapdb.get(em_url));
```

```
        Util_DataCache.setObjVal(em_url_open, mapdb.get(em_url_open));
    }
    return map;
}
```

5. 对接异构系统

5.1 接口白名单配置

5.2 Token认证

6. 开发技巧