

云帆平台二次开发指南

由 | 左萌萌 | 最近更新于 2022年8月16日在GMT+8 09:51

内容

- 云帆平台开发指南
 - 1. 初始项目搭建
 - 1.1 昆仑微服务平台项目搭建
 - 前端项目搭建
 - 后端项目搭建
 - 1.2 莫邪单体平台项目搭建
 - 1.2.1 前端项目搭建
 - 1.2.1.1 node环境安装
 - 1.2.1.2 查看是否安装成功
 - 1.2.1.3 环境变量配置
 - 1.2.1.4 设置NPM私服
 - 1.2.1.5 下载平台前端模板工程
 - 1.2.1.6 安装npm依赖包及环境配置
 - 1.2.2 后端项目搭建
 - 1.2.2.1 通过archetype生成项目
 - 1.2.2.2 Java 示例工程包路径
 - 1.3 集成系统权限管理
 - 1.4 获取当前用户信息
 - 1.4 多数据源切换
 - 编程式动态数据源切换
 - 声明式动态数据源切换
 - 2. 平台表单视图内置变量
 - 2.1 表单变量使用
 - 可使用变量的地方:
 - 2.2 视图变量使用
 - 可使用变量的地方:
 - 2.3 视图SQL语法
 - 3. 视图自定义开发
 - 3.1 视图列按钮传递参数到表单
 - 3.1.1 列按钮配置
 - 3.1.2 扩展配置
 - 3.1.3 组件参数:
 - 3.2 获取URL查询参数
 - 3.2.1 查询条件URL传参配置说明
 - 3.2.2 URL参数传递到视图SQL查询配置
 - 3.2.3 URI传参控制行按钮显示
 - 3.3 自定义按钮调用Vue组件

- 3.3.1 列按钮配置
 - 3.3.2 扩展配置
 - 3.3.3 组件路径及参数
- 3.4 自定义按钮调用JS脚本
 - 3.4.1 列按钮配置
 - 3.4.2 扩展配置
 - 3.3.3 组件路径及参数
- 4. 表单自定义开发
 - 4.1 表单生命周期
 - 4.1.1 onLoad
 - 4.1.2 onRendered
 - 4.1.3 onValidate
 - 4.1.4 onPreAction
 - 4.1.5 onCustomAction
 - 4.1.6 onActionDone
 - 4.2 表单控件事件
 - 4.2.1 change事件 值改变
 - 4.2.2 visible-change事件 下拉框出现/隐藏时触发
 - 4.2.3 keyUp事件 按键松开
 - 4.2.4 keyDown事件 按键按下
 - 4.2.5 enter事件 回车
 - 4.2.6 onBlur事件 失去焦点
 - 4.2.7 onFocus事件 获取焦点
 - 4.2.8 upload事件 附件上传成功事件
 - 4.2.9 delete事件 附件删除成功事件
 - 4.3 表单自定义JS编写规范
 - 4.4 表单控件传递参数给弹出视图
 - 4.5 获取表单控件
 - 4.5.1 主表控件
 - 4.5.1 一对多从表控件
 - 4.5.1 一对一从表控件
 - 4.6 修改表单控件的值
 - 4.6.1 主表控件
 - 4.6.2 一对一从表控件
 - 4.6.3 一对多从表
 - 4.7 控制表单编辑/只读状态
 - 4.8 弹出自定义Vue组件
 - 4.8.1 表单弹出框自定义组件
 - 4.8.2 表单按钮自定义组件
 - 4.9 表单弹出框调控IFrame
- 5. 自定义Vue页面开发
 - 5.1 自定义页面打开视图
 - 5.2 自定义页面打开表单

- 5.2 自定义页面打开方式
- 5.3 自定义页面嵌入视图
- 5.4 自定义页面嵌入表单
- 6. 接口自定义开发规范
 - 6.1 常用方法规范
 - 6.2 传入参数
 - 后端代码示例
 - 前端代码示例
 - 6.3 响应参数
 - 6.4 异常类
 - 6.5 权限控制
 - 资源管理增加按钮
 - 后端接口增加权限判断
 - 前端接口增加权限判断
- 7. 平台安全配置参数说明
 - 微服务在nacos中配置的安全参数
 - 莫邪平台配置文件中安全参数
- 8. 多租户配置
 - 8.1 服务端多租户
 - 8.2 客户端多租户

云帆平台开发指南

1. 初始项目搭建

1.1 昆仑微服务平台项目搭建

前端项目搭建

1. 下载前端工程模板，并导入到VSCode
2. 安装依赖

```
npm install --registry=http://192.168.14.114:8081/repository/npm-group/
```

3. 打开.env文件，请确保使用的是cloud-gateway网关，修改VUE_APP_GATEWAY参数与网关ip和端口一致
4. 配置前端代理

打开配置文件vue.config.js，在节点devServer.proxy:参数配置后面增加一个配置；

```

'/demo-biz': {
  target: url,
  ws: true,
  pathRewrite: {
    '^/demo-biz': '/demo-biz'
  }
}

```

5. 前台工程结构如下:

- api目录, 前端与后台交互的接口均放置在此处
- assets目录, 主要是图标和一些静态文件
- components目录, 平台封装的部分组件
- config目录, 平台配置目录
- const目录, 平台表单设计相关目录
- page目录, 平台主框架目录
- router目录, 前端路由配置
- views目录, 项目组的vue文件可放置于此。

6. 运行项目

```
npm run dev
```

7. 登录: 打开Chrom浏览器, 输入地址 <http://localhost:端口号/> 打开登录页面, 输入验证码, 点登录;

后端项目搭建

1. 添加后端项目模板

在使用Maven模板创建项目之前, 请确保Maven的settings.xml中的仓库指向公司192.168.14.27的私服地址

以下1或2选择一点个进行安装

•

1. 直接通过命令生成
以新增微服务 demo-biz 为例

```

mvn archetype:generate \
  -DgroupId=com.tellhow.example \
  -DartifactId=demo-biz \
  -Dversion=1.0.0 \
  -Dpackage=com.tellhow.example.demo \
  -DarchetypeGroupId=com.tellhow.cloud \
  -DarchetypeArtifactId=cloud-biz-archetype \
  -DarchetypeVersion=1.5.1 \
  -DarchetypeCatalog=local

```

•

2. 以idea为例(Eclipse可参考[这里](#)), 点击File-New-Project..., 在左侧项目类别中选择Maven项目, 在右侧勾选Create from Archetype, 再点击Add Archetype, 在弹出的新增框中根据以下信息填入对应位置后点击OK新增模板

```
GroupId: com.tellhow.cloud
ArtifactId: cloud-biz-archetype
Version: 1.5.1
```

使用模板创建后端项目

新增完成模板之后选择刚添加的模板向导创建后端项目，填入项目的GroupId，ArtifactId及版本号信息

```
GroupId: com.tellhow.eomp
ArtifactId: demo-biz
Version: 1.0.0
```

输入完成后，next => Finish 完成，等待创建项目

2. 工程结构

使用模板创建后的工程结构如下：

- db 平台示例脚本（可以不用执行）
- demo-biz-api 接口项目
- demo-web web应用主项目
- naocs nacos配置文件以及路由配置文件目录

3. 微服务配置文件修改

平台使用阿里巴巴开源Nacos作为配置中心，所以遵循Nacos配置文件加载规则。加载规整如下：

- 1. 优先加载服务 resources 目录 bootstrap.yml
- 2. 根据 bootstrap.yml 配置加载 nacos 配置（规则为 `${spring.application.name} - ${spring.profiles.active}.yml`）这里就是 `demo-biz-dev.yml`（若没有请自行创建，注意格式为 yml）
- 3. 最后加载 nacos 的通用配置文件 `application-dev.yml`、

修改nacos/demo-biz-web-dev.yml，修改数据库连接信息，默认为mysql，如下图：

```

security:
  oauth2:
    client:
      client-id: system
      client-secret: 123456789
      scope: server

spring:
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    druid:
      driver-class-name: com.mysql.cj.jdbc.Driver
      username: ${MYSQL-USER:root}
      password: ${MYSQL-PWD:root}
      url: jdbc:mysql://${MYSQL-HOST:cloud-mysql}:${MYSQL-PORT:3306}/${MYSQL-DB:equip}?characterEr

mybatis-plus:
  tenant-enable: ture
  mapper-locations: classpath*:mapper/*Mapper.xml,classpath*:mapper/**/*.xml
  global-config:
    db-config:
      schema: equip
  configuration:
    variables:
      SCHEMA: equip

```

需要登录nacos，在nacos的配置管理=> 配置列表中点击导入配置，导入修改好的配置文件

4. 路由信息修改

- 4.1、登录干将管理界面
- 4.2、找到系统管理->动态路由菜单，选择text模式（默认为Tree模式）
- 4.3、将整个路由表（JSON格式）拷贝到文本文件中，打开工程结构，找到nacos文件夹下的route.json文件，将该路由信息加到整个的路由表中，更新路由

1.2 莫邪单体平台项目搭建

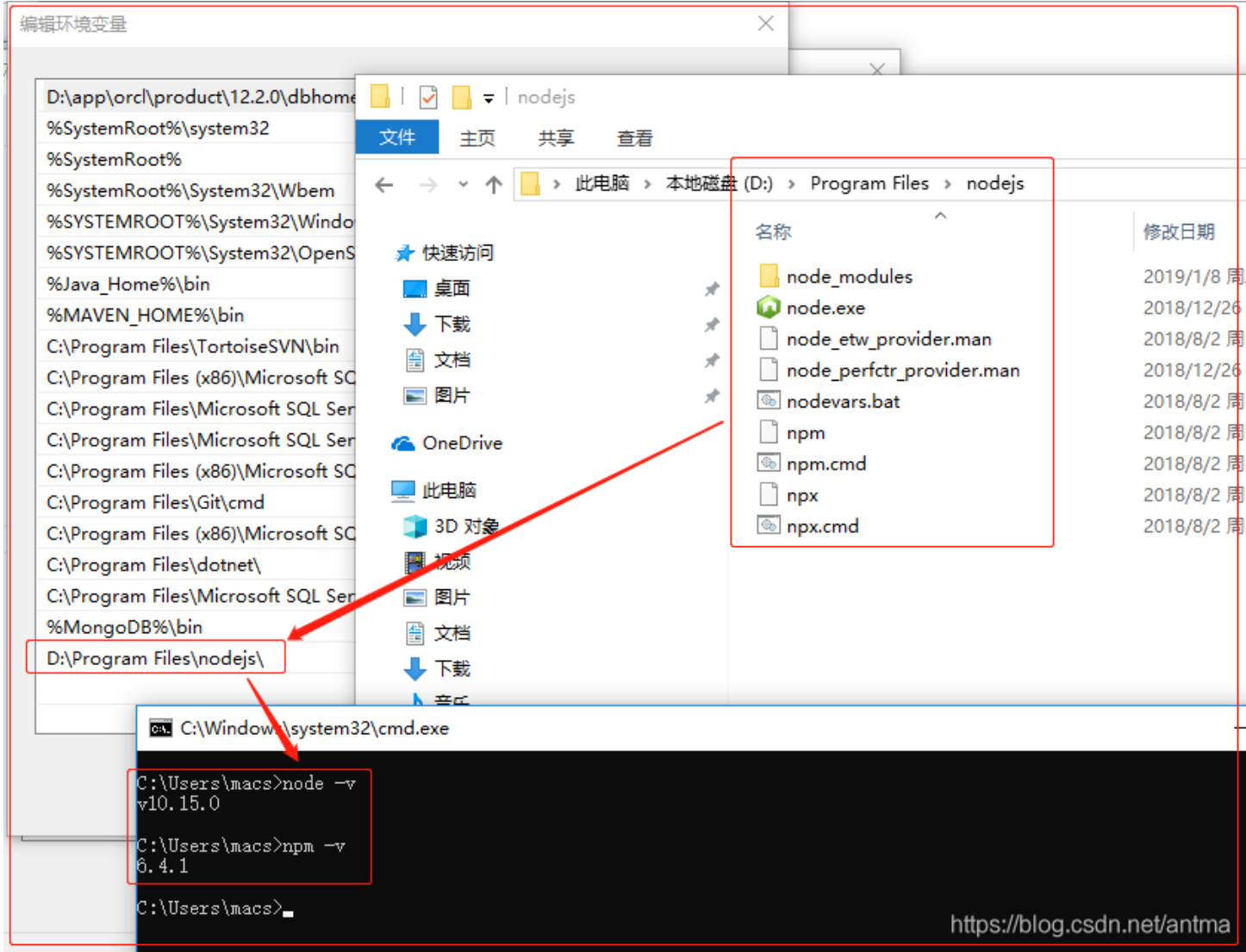
1.2.1前端项目搭建

1.2.1.1 node环境安装

Node.js 官方网站下载：<https://nodejs.org/en/> 

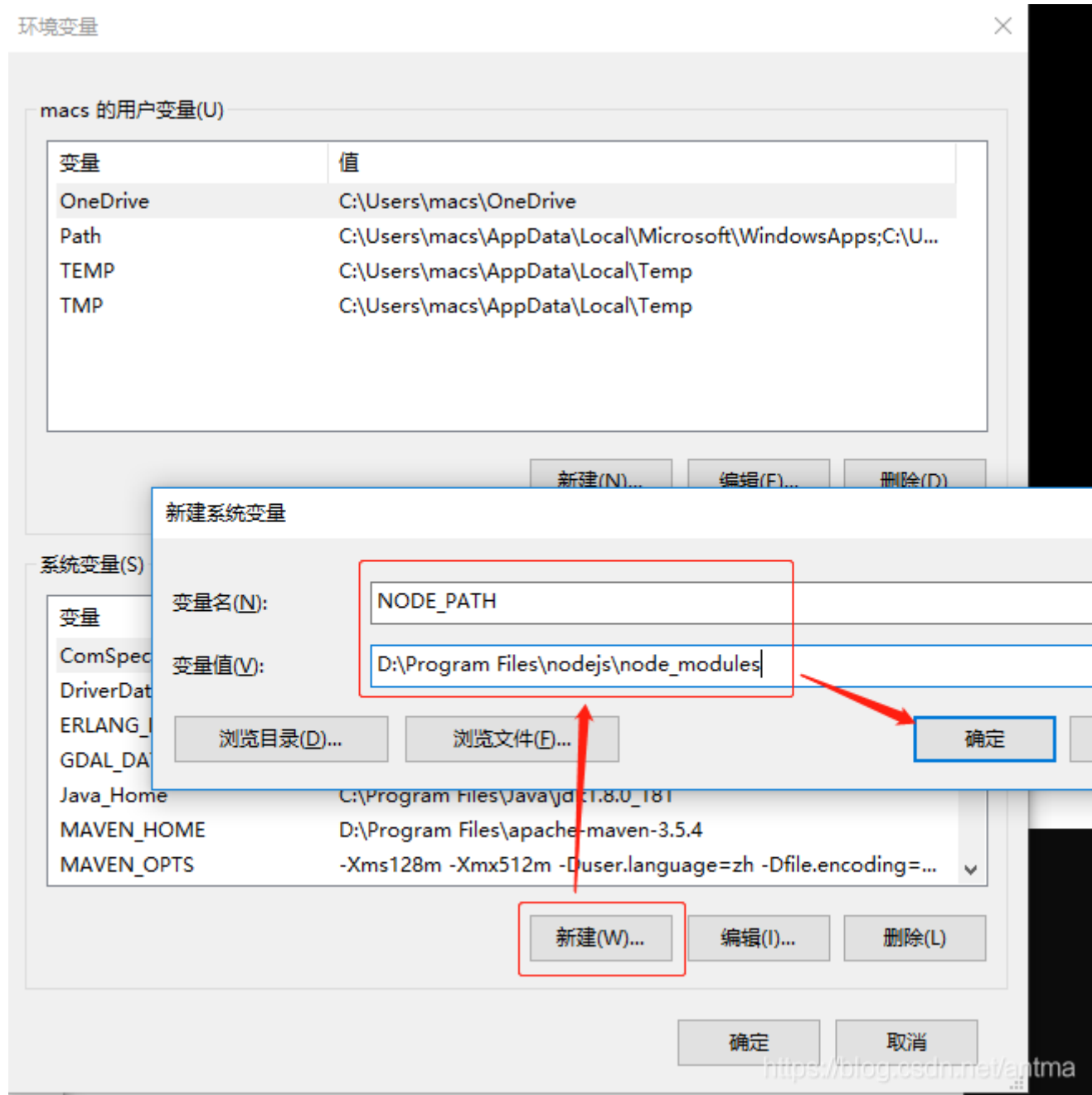
1.2.1.2 查看是否安装成功

安装成功，文件夹结构如下，并在上面安装过程中已自动配置了环境变量和安装好了npm包，此时可以执行node -v 和 npm -v 分别查看node和npm的版本号



1.2.1.3 环境变量配置

环境变量 -> 系统变量中新建一个变量名为 "NODE_PATH", 值为"D:\Program Files\nodejs\node_modules", 如下图:



1.2.1.4 设置NPM私服

```
npm install --registry=http://192.168.14.114:8081/repository/npm-group/
```

1.2.1.5 下载平台前端模板工程

打开终端命令行进入任意目录

设置公司npm地址

```
npm config set registry http://192.168.14.114:8081/repository/npm-group/
```

全局安装tidp-cli脚手架

```
npm install -g tidp-cli
```



```
C:\Users\andywen\Desktop>npm install -g tidp-cli
npm WARN deprecated axios@0.19.2: Critical security vulnerability fixed in v0.21.1. For more information, see https://github.com/axios/axios/pull/3410
C:\Users\andywen\AppData\Roaming\npm\tidp -> C:\Users\andywen\AppData\Roaming\npm\node_modules\tidp-cli\index.js
+ tidp-cli@1.0.0
added 192 packages from 138 contributors in 11.917s
```

查看前端工程模板列表

```
tidp list
```

初始化模板工程

```
tidp init tile test
```

其中项目名称、项目简介及作者名称根据业务需要自行填写（非必填，填写内容会填充到package.json对应字段中）

```
C:\Users\andywen\Desktop\websocket>tidp init tile test
✓ 模板下载成功!
? 请输入项目名称 myproject
? 请输入项目简介 这是一个XXX项目
? 请输入作者名称 myname
✓ 初始化模板成功
```

1.2.1.6安装npm依赖包及环境配置

```
npm install --registry=http://192.168.14.114:8081/repository/npm-group/
```

修改配置文件，修改utils/constant.js 下面的配置：

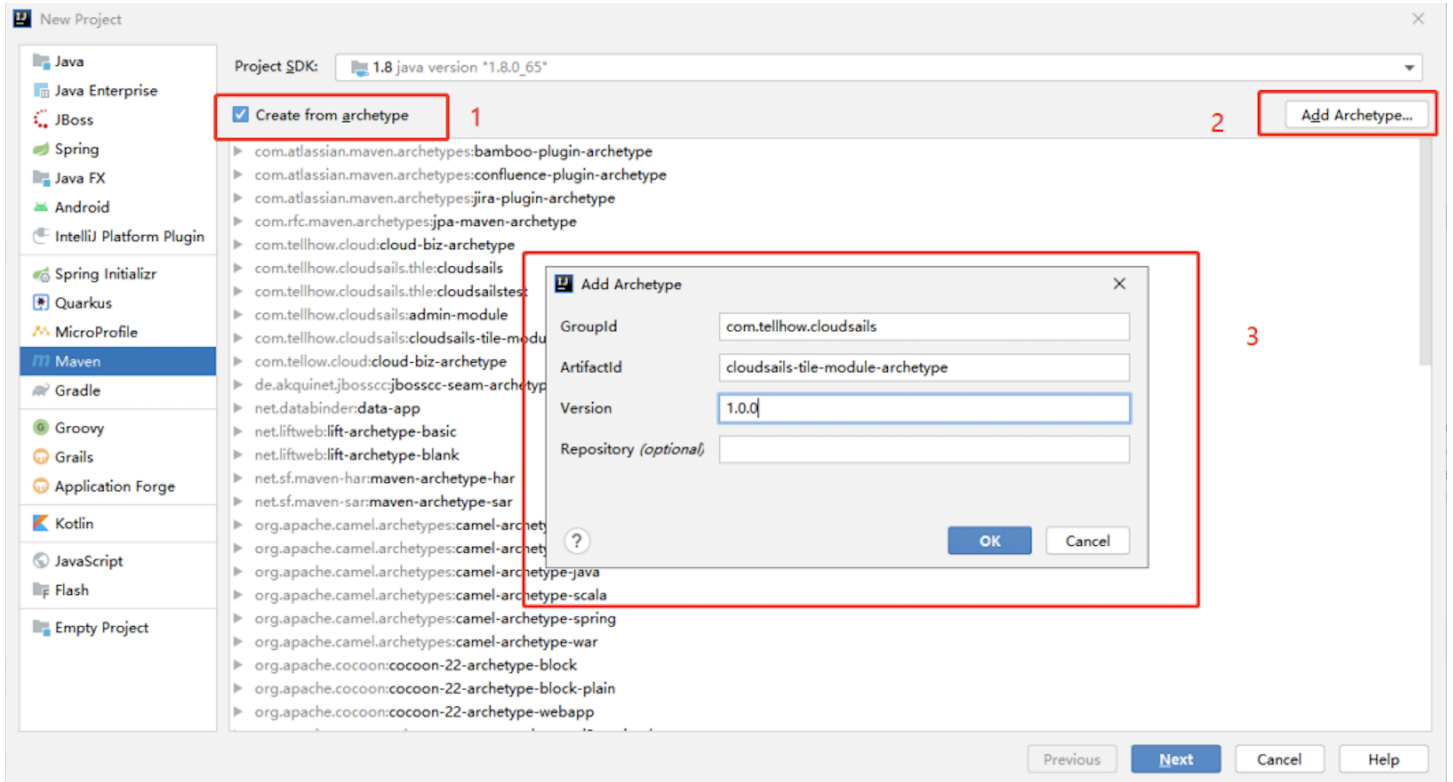
```
AppServerAddress: 'localhost',    服务端地址
AppServerPort: '9875',           服务端端口
AppServerRoot: '/admin',         服务端项目名称
```

启动前端工程，使用 shell npm run dev。

访问验证：<http://localhost:80> ，进入登录页面

1.2.2后端项目搭建

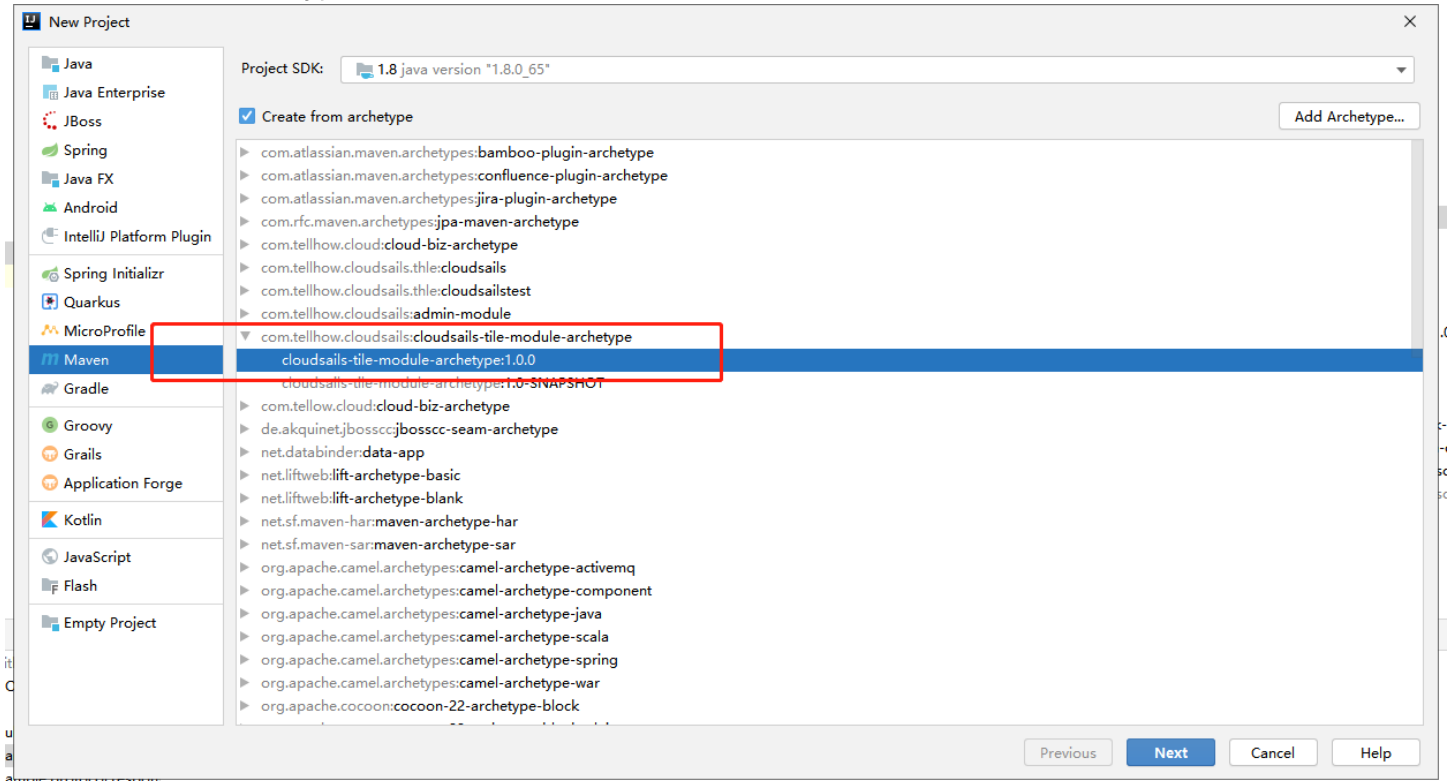
1.2.2.1 通过archetype生成项目

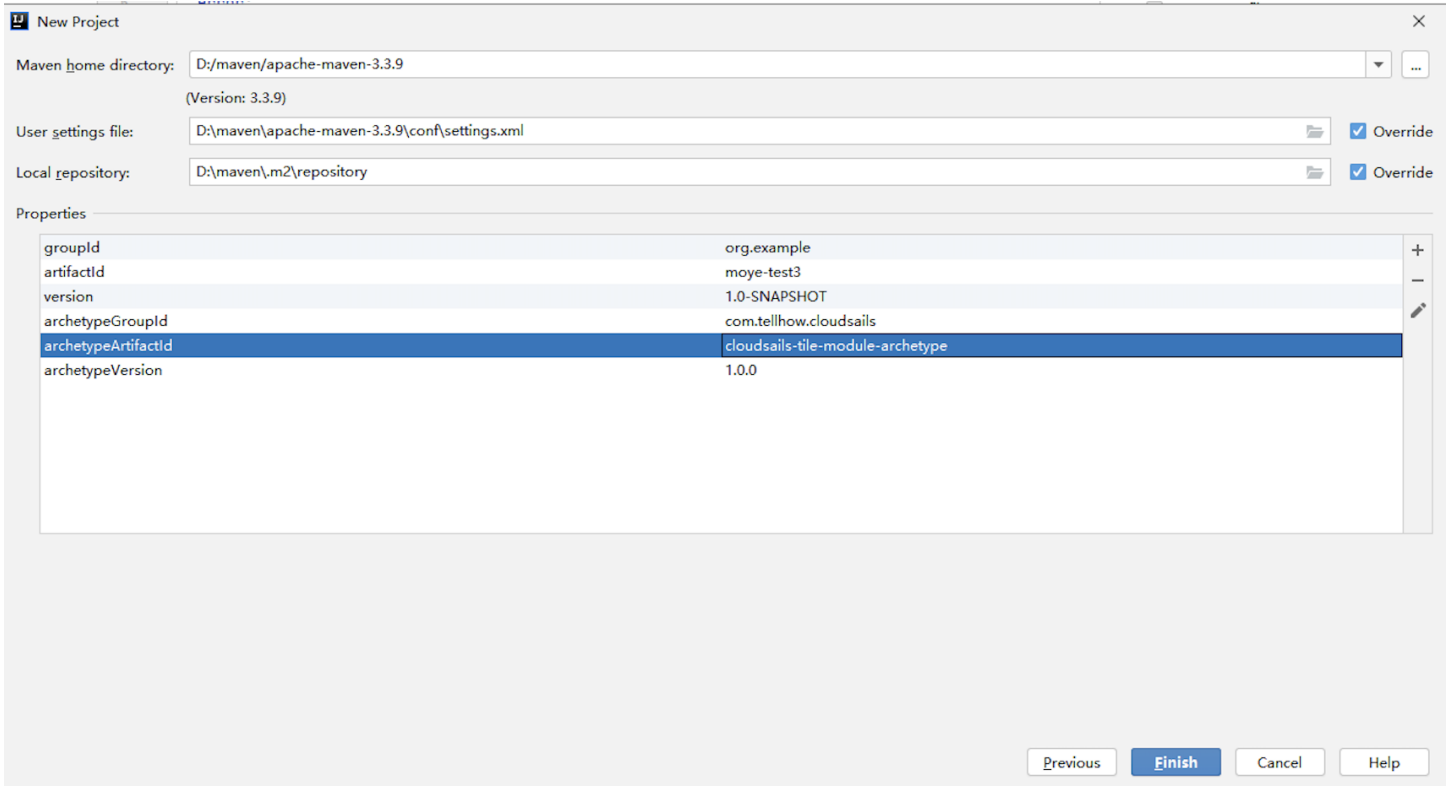


- 1.Create from archetype
- 2.Add archetype
- 3.填写archetype相关东西

```
com.tellhow.cloudsails  
cloudsails-tile-module-archetype  
1.0.0
```

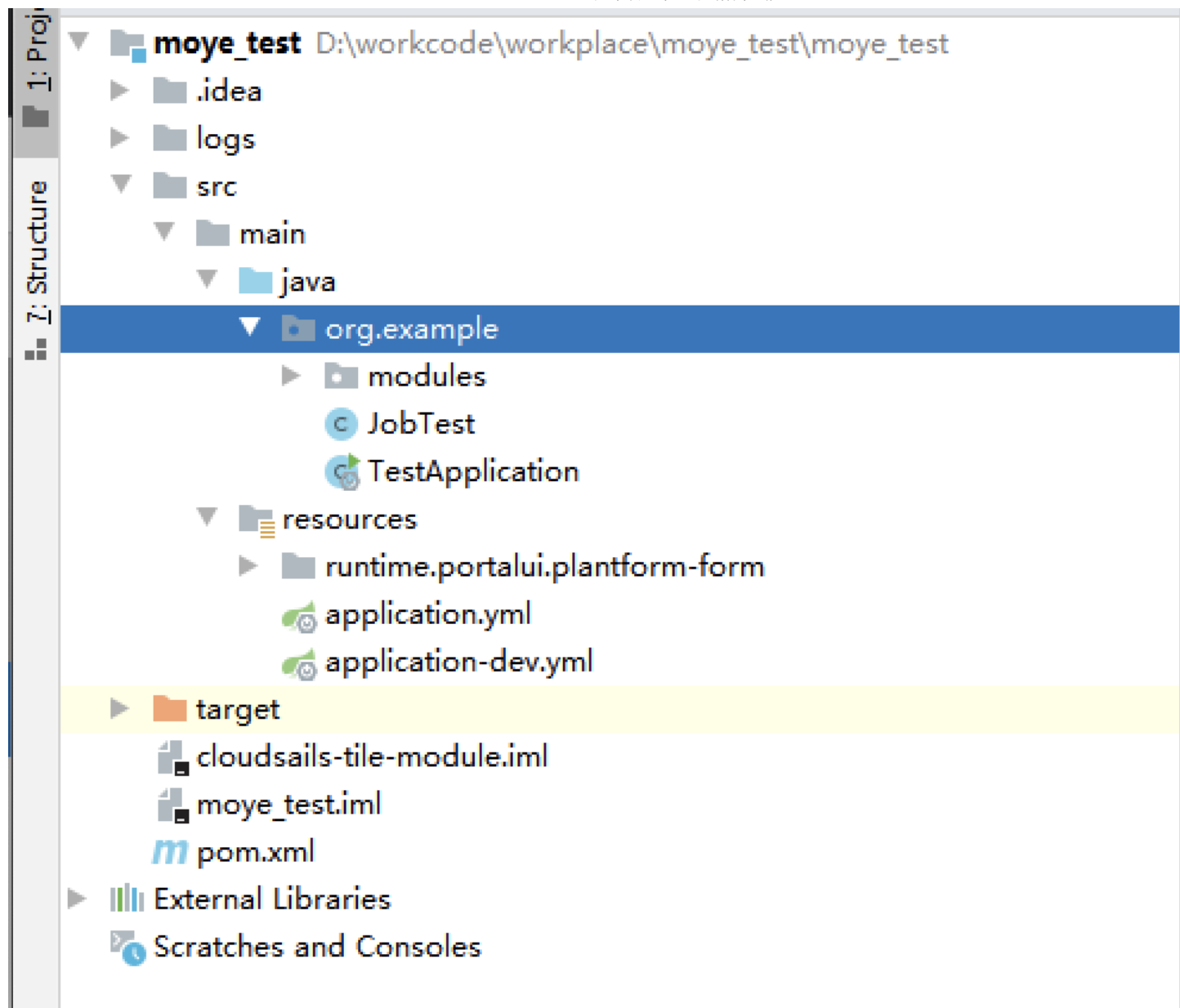
选择刚才新增的archetype，点击next





1.2.2.2 Java 示例工程包路径

- |——org.example.modules.one.controller 示例类control路径
- |——org.example.TestApplication 项目启动类启动
- Resource配置文件路径
- |——runtime.portalui 开发平台配置文件放置路径
- |——application.yml 项目配置文件
- |——application-dev.yml 项目配置文件



1.3 集成系统权限管理

1.4 获取当前用户信息

- 后端代码获取用户信息

```
UserInfo userInfo = UserUtils.getUser();  
String userId = userInfo.getUserId();
```

1.4 多数据源切换

编程式动态数据源切换

1. 使用动态数据源刷新

在代码中调用以下代码，实现数据源切换。

```
try {
    DynamicDataSourceContextHolder.setDataSourceType(DataSourceConstants.DEFAULT_DS_KEY);
    //业务逻辑
} finally {
    DynamicDataSourceContextHolder.clearDataSourceType();
}
```

2. 在事务开始前切换数据源

当使用@Transaction注解进行事务管理后，方法内部使用
DynamicDataSourceContextHolder.setDataSourceType("xxx")，切换多数据源，不生效。

解决方法:在使用@Transaction注解进行事务管理前，先切换动态数据源

```
FormDTO formDTO = getFormDTOByCode(code);
List<FormSubDTO> formSubDTOList = getListFormSubDTOByFormDTO(formDTO);
try {
    DynamicDataSourceContextHolder.setDataSourceType(formDTO.getFormBaseInfo().getDataSourceName());
    devFormConfigFieldService.batchDeleteFormDate(formDTO, formSubDTOList, ids);
} finally {
    DynamicDataSourceContextHolder.clearDataSourceType();
}

@Override
@Transactional(isStart = true)
@Transactional(rollbackFor = Exception.class)
public void batchDeleteFormDate(FormDTO formDTO, List<FormSubDTO> formSubDTOList, String ids) {
    String[] idsAll = ids.split(",");
    for (String id : idsAll) {
        deleteFormDate(formDTO, formSubDTOList, id);
    }
}
```

声明式动态数据源切换

声明式动态数据源切换基于Spring AOP动态代理实现，使用 **@DS** 切换数据源，注意：@DS注解建议写在
Service类上，MyBatis的Dao类上不生效（DAO类已经被MyBatis动态代理）。

@DS 可以注解在方法上或类上，**同时存在就近原则 方法上注解 优先于 类上注解**。

注解	结果
没有@DS	默认数据源
@DS("dsName")	dsName可以为组名也可以为具体某个库的名称

```

import com.tellhow.cloud.common.datasource.annotation.DS;

@Service
@DS("datasource1")
public class UserServiceImpl implements UserService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public List selectAll() {
        return jdbcTemplate.queryForList("select * from user");
    }

    @Override
    @DS("datasource2")
    public List selectByCondition() {
        return jdbcTemplate.queryForList("select * from user where age >10");
    }
}

```

2. 平台表单视图内置变量

2.1 表单变量使用

- 使用全局变量
格式：{变量名} 例子：{UUID}
- 使用主表字段的值
格式：F{字段名称} 例子：F{BILLNO}
- 使用一对一从表字段的值
格式：F{从表名称.字段名称} 例子：F{cb.BILLNO}
- 使用数据库序列
格式：S{序列名称} 例子：S{SYS_DICT_SEQ}

可使用变量的地方：

1. 表单设计->基本配置->表单标题
支持变量：全局变量，主表字段值，从表字段值
2. 表单设计->控件属性->默认值
支持变量：全局变量，序列
3. 表单设计->控件属性->选项(单选，复选，下拉，弹出框)->输入SQL
支持变量：全局变量，主表字段的值，从表字段的值
示例：
全局变量示例

```

SELECT id as value, label FROM CLOUDX.SYS_DICT_ITEM
<where>
    CREATE_TIME > '${NowDate}'
</where>

```

主表字段的值示例（级联表单字段为formField）：

```
SELECT id as value,label FROM CLOUDX.SYS_DICT_ITEM
<where>
  <if test="formField != null and formField != ''"> AND type = #{formField}</if>
</where>
```

从表字段的值示例（级联从表字段为formField，主表设置的子表字段名称为subtable）：

```
SELECT id as value,label FROM CLOUDX.SYS_DICT_ITEM
<where>
  <if test="subtable_FTF_formField != null and subtable_FTF_formField != ''"> AND type = #{subtable_FTF_formField}</if>
</where>
```

4. 表单设计->按钮配置->配置->显示条件

5. 表单设计->控件属性->从表按钮->配置->显示条件

支持变量：获取一对一子表单数据使用: \$F{从表名.字段名}

获取主表单数据使用: \$F{字段名}

获取全局参数数据使用: \${参数名}

2.2 视图变量使用

- 使用全局变量
格式：{变量名} 例子：{UUID}
- 查询条件参数
格式：#{变量名} 例子：#{createUser}
- 行参数
格式：Row.字段名 例子：Row.currentHandler
- url参数
格式：u{参数名} 例子：u{FlowId}

可使用变量的地方：

1. 视图设计->字段配置->视图SQL
2. 视图设计->字段配置->扩展配置(控件类型为转义)->输入SQL
3. 视图设计->查询条件配置->扩展配置(控件类型为下拉或弹出框)->输入SQL
支持变量：全局参数\${paramCode}, 视图查询条件#{字段名称}
示例：
全局变量

```
select organid,organname,parentid,organgrade from t_tbp_organ where organname = '${全局参数名称编码}'
```

级联视图查询参数示例（级联表单字段为param）：

```
select organid,organname,parentid,organgrade from t_tbp_organ
<where>
  <if test="param != null and param != ''"> and organname = #{param}</if>
</where>
```

4. 视图设计->字段配置->行按钮配置/列按钮配置->扩展配置->显示条件

支持变量：行数据 \$Row.字段名(列按钮没有这个变量);url上的参数 u{参数名};全局参数{参数名}

示例：

设置当前用户是当前行记录的处理人时显示按钮: *Row.currentHandler* == '{CurrentUserName}'

设置当前流程号是1时显示按钮: 'u{FlowId}' == '1' 设置当前登录人是admin时显示按钮:

'{CurrentUserName}' == 'admin'

设置当前行处理人是admin时显示按钮: \$Row.currentHandler == 'admin'

多个条件连接通过 && 或者 ||,为时空默认显示

5. 视图设计->查询条件配置->默认值

支持变量：全局变量 \${参数名}

6. 视图设计->导航面板->树展示SQL

支持变量：全局参数\${paramCode}

2.3 视图SQL语法

支持mybatis的语法解析;

#{paramCode} 用于获取查询参数;

\${paramCode} 用于获取全局参数;

如果要传入参数，支持mybatis和全局标量的参数格式，参数为abc；（mybatis参数需添加if判断）

基本用法示例：

```
select * from table where id = '1'
```

传参用法示例：

```
<script> SELECT * FROM CLOUDX.T_TBP_NOTICE
<where>
  PUBLISHER = ${CurrentUserId}
  <if test="ISPUBLISH != null and ISPUBLISH != ''"> AND ISPUBLISH = #{ISPUBLISH}</if>
  <if test="PUBLISHCONTENT != null and PUBLISHCONTENT != ''"> AND PUBLISHCONTENT LIKE CONCAT('%'
</where> ORDER BY PUBLISHTIME DESC
</script>
```

3. 视图自定义开发

3.1 视图列按钮传递参数到表单

3.1.1 列按钮配置

选择视图 --> 列按钮配置 --> 添加按钮 --> 自定义按钮 --> 扩展配置

3.1.2 扩展配置

配置项	配置值
扩展类型	弹出框模式
打开方式	弹出框组件
内容来源	组件
组件类型	表单

3.1.3 组件参数:

- 获取行数据使用: \$Row.字段名
- 获取全局参数数据使用: \${参数名}
- 参数值为固定值: 25
- 参数值为全局变量: \${UUID}
- 参数值为行数据: \$Row.ID

例如:

传入表单组件对应的KEY	传入表单组件对应的VALUE
TITLE	233
PUBLISHER	\${CurrentUserName}
PUBLISHCONTENT	\$Row.ID
PUBLISTHTIME	\$Row.CREATETIME

3.2 获取URL查询参数

3.2.1 查询条件URL传参配置说明

第一步: 打开视图-> 基础信息配置-> 视图信息配置 -> URL 参数

PUBLISHER1

第二步: 切换到查询条件配置页-选择一个查询条件进行扩展配置SQL配置:

```
SELECT id as value,label FROM CLOUDX.SYS_DICT_ITEM <where><if test="PUBLISHER1 !='' and PUBLISHER1
```

特别说明:

选项配置-> SQL查询配置: 如果条件是从url 获取的请不要解析参数, 或者有级联的时候, 引用参数里排除下 URL传参。

第三步: 访问地址, 打开F12查看视图数据查询请求即可看到效果

http://192.168.63.56:8090/#/gen/view/portalui/m031603/v0429?PUBLISHER1=13

示例参考：192.168.63.56:9090 》云应用平台租户 》云应用平台空间 》云帆平台门户系统 》云帆测试分组 》xuxz 》弹窗遮罩测试

3.2.2 URL参数传递到视图SQL查询配置

第一步：打开视图-> 基础信息配置-> 视图信息配置 -> URL 参数

TITLE

第二步：打开视图-> 字段配置-> 视图SQL

```
<script>SELECT * FROM CLOUDX.T_TBP_NOTICE
  <where>
    <if test="TITLE != null and TITLE != ''"> AND TITLE = #{TITLE}</if>
  </where>
</script>
```

第三步：访问地址，打开F12查看视图数据查询请求即可看到效果

http://192.168.63.56:8090/#/gen/view/portalui/m031603/v0429?TITLE=zhangsan

3.2.3 URI传参控制行按钮显示

进入行按钮配置页面 -> 选择一个行按钮 -> 扩展配置 -> 显示条件

`$u{abc} == '123'`

3.3 自定义按钮调用Vue组件

3.3.1 列按钮配置

选择视图 --> 列按钮配置 --> 添加按钮 --> 自定义按钮 --> 扩展配置

3.3.2 扩展配置

配置项	配置值
扩展类型	自模式
自定义类型	组件

3.3.3 组件路径及参数

说明：组件放置在src/components目录下

```
@/components/test/test.vue
```

组件传参：

- 获取行数据使用: \$Row.字段名 (当前变量仅列按钮支持)
- 获取全局参数数据使用: \${参数名}

```
{"data":"viewtype","UUID":"${UUID}","ID":"$Row.ID","TITLE":"$Row.TITLE"}
```

行(列)按钮打开自定义组件引用表单组件: test.vue

```

<template>
  <div class="form-wrap" v-if="formVisible">
    <form-detail
      :config="{
        model: 'form',
        perms: 'add', //新增表单为 add,编辑表单为edit,查看表单为detail
        serviceName: 'default',
        moduleCode: 'test-tangzx-module',
        appCode: 'portalui',
        viewCode: 'tzx-notice-view',
        formCode: 'tzx-notice',
        openType: 1,
        // objectId: '123123', //查看和编辑表单需要此属性-值为表单id
        addedProps: {ID: '10000'}, //新增表单时,此参数默认带到表单字段上
        extraData: {test: '22222'} //表单自定义参数
      }"
      :btnCallBack="btnCallBack"
      :hasCloseBtn="true"
      :dialogVisibleFalse="dialogVisibleFalse"
    />
  </div>
</template>

<script>
  //视图自定义行按钮示例
  // 引入表单组件
  import FormDetail from "@/views/form/form-detail";

  export default {
    name: "Component",
    components: {
      FormDetail,
    },
    props: {
      //自定义参数
      customProps: {
        type: Object,
      },
    },
    data() {
      return {
        formVisible: true,
      };
    },
    mounted() {
      console.log(this.customProps);
    },
    methods: {
      // 表单按钮回调事件
      btnCallBack(ac) {
        // 通过ac的code属性判断按钮类型
        // 1.save 保存和更新
        // 2.custom 自定义
        // 通过ac的text属性判断按钮文字
        // 例如: ac.text === "自定义1"
        // 注意: 自定义按钮js如果触发了saveForm方法, 此表单按钮回调会被再次触发, 此时的code=== "save"
        if (ac.code === "save") {
          setTimeout(() => {
            this.$emit("refreshView");
          }, 1500);
        }
        // 可以通过code和text区分不同的自定义按钮
        if (ac.code === "custom" && ac.text === "自定义1") {
          // do something
        }
      },
      // 表单关闭回调事件 (点击表单关闭按钮或者在自定义按钮中调用closeForm方法都会触发此回调)
      dialogVisibleFalse() {
        this.formVisible = false;
      },
    },
  },

```

```
    },
  };
</script>

<style>
  .form-wrap {
    position: fixed;
    top: 0;
    left: 0;
    right: 0;
    bottom: 0;
    background: #fff;
    z-index: 99;
  }

  /* 关闭按钮样式覆盖 */
  .form-wrap .form-actions .dialog-close-btn {
    color: #09b9a2;
  }
</style>
```

3.4 自定义按钮调用JS脚本

3.4.1 列按钮配置

选择视图 --> 行(列)按钮配置 --> 添加按钮 --> 自定义按钮 --> 扩展配置

3.4.2 扩展配置

配置项	配置值
扩展类型	自定义模式
自定义类型	脚本

3.3.3 组件路径及参数

说明：脚本放置在src/components目录下
js路径：test.js

```
@/components/test/test.js
```

函数名称：test
参数：

- 获取行数据使用: \$Row.字段名 (当前变量仅列按钮支持)
- 获取全局参数数据使用: \${参数名}

```
{"data":"viewtype","UUID":"${UUID}","ID":"$Row.ID","TITLE":"$Row.TITLE"}
```

test.js

```
//视图自定义按钮示例
// data 即为传递过来的参数如上
function test(data, that, action) {
  action.refreshView();
  console.log(data)
}
export default {
  test
}
```

参数	说明
data	js参数
that	this对象
actions	按钮对象

4. 表单自定义开发

4.1 表单生命周期

4.1.1 onLoad

```
//数据加载后，页面渲染之前，this为window
form.on('onLoad',function(data, dataPermission, actions, subActions){},'cover');
```

参数	说明
data	表单数据
dataPermission	表单控件权限 e:可写 r:必填 v:是否显示
actions	主表按钮 disabled:可用 visible:可见 type:类型
subActions	从表按钮 disabled:可用 visible:可见 type:类型

4.1.2 onRendered

```
//页面渲染完成后
form.on('onRendered',function(data, prop, actions){});
```

参数	说明
data	表单数据
prop	表单属性 extraData:额外的数据 isFlow:是否关联流程 perms:表单状态
actions	主表按钮 disabled:可用 visible:可见 type:类型

4.1.3 onValidate

```
//内置校验完成后, 返回false阻止提交
form.on('onValidate',function(actions,data){});
```

参数	说明
data	表单数据
actions	主表按钮 disabled:可用 visible:可见 type:类型

4.1.4 onPreAction

```
//操作前 (包含自定义按钮) 执行, 返回false阻止按钮操作
form.on('onPreAction',function(actions,data,formRenderer){});
```

参数	说明
data	表单数据
actions	主表按钮 disabled:可用 visible:可见 type:类型
formRenderer	表单控件 controls:所有控件 dataItems:数据项 formPermission:表单权限

4.1.5 onCustomAction

```
//自定义按钮执行
form.on('onCustomAction',function(actions,data){});
```

参数	说明
data	表单数据
actions	主表按钮 disabled:可用 visible:可见 type:类型

4.1.6 onActionDone

```
//操作完成后执行
form.on('onActionDone',function(actions,data,httpRes){});
```

参数	说明
data	表单数据
actions	主表按钮 disabled:可用 visible:可见 type:类型
httpRes	接口响应结果

4.2 表单控件事件

4.2.1 change事件 值改变

控件添加属性 data-on-change="changeLevel"

```
<a-text key="BZR_NAME" data-name="编制人姓名" data-on-change="changeLevel"></a-text>
```

添加script

```
<script>
    //示例 value:改变的值; oldValue:旧值; form:表单数据; $this:当前控件对象
    function changeLevel(value,oldValue,form,$this){
        //获取下拉框的选中的值
        let select = $this.options.find(item => item.value == value);
        //将选中的label赋值给表单的编号字段
        form.BILLNO.value = select.label;
    }
</script>
```

4.2.2 visible-change事件 下拉框出现/隐藏时触发

控件添加属性 data-on-visible-change="visibleChange"

```
<a-dropdown key="BZR_NAME" data-name="编制人姓名" data-on-visible-change="visibleChange"></a-dropdo
```

添加script

```
<script>
    //示例 form:表单控件; $this:当前控件对象;visible 下拉框出现/隐藏时触发 出现则为 true, 隐藏则为 false
    function visibleChange(form,$this,visible){
        if(visible){
            $this.$message.error("下拉框出现时触发,给出提示,下拉项数据量: " + $this.optSet.length)
        }
    }
</script>
```


4.2.3 keyUp事件 按键松开

控件添加属性 data-on-key-up="keyUp"

```
<a-text key="BZR_NAME" data-name="编制人姓名" data-on-key-up="keyUp"></a-text>
```

添加script

```
<script>
  //示例 form:表单控件; $this:当前控件对象;
  function keyUp(form,$this){
    console.log(form)
    console.log($this)
  }
</script>
```

4.2.4 keyDown事件 按键按下

控件添加属性 data-on-key-down="keyDown"

```
<a-text key="BZR_NAME" data-name="编制人姓名" data-on-key-up="keyDown"></a-text>
```

添加script

```
<script>
  //示例 form:表单控件; $this:当前控件对象;
  function keyDown(form,$this){
    console.log(form)
    console.log($this)
  }
</script>
```

4.2.5 enter事件 回车

控件添加属性 data-on-enter-key-down="enterKeyDown"

```
<a-text key="BZR_NAME" data-name="编制人姓名" data-on-enter-key-down="enterKeyDown"></a-text>
```

添加script

```
<script>
  //示例 form:表单控件; $this:当前控件对象;
  function enterKeyDown(form,$this){
    console.log(form)
    console.log($this)
  }
</script>
```

4.2.6 onBlur事件 失去焦点

控件添加属性 data-on-blur="onBlur"

```
<a-text key="BZR_NAME" data-name="编制人姓名" data-on-blur="onBlur"></a-text>
```

添加script

```
<script>
  //示例 form:表单控件; $this:当前控件对象;
  function onBlur(form,$this){
    console.log(form)
    console.log($this)
  }
</script>
```

4.2.7 onFocus事件 获取焦点

控件添加属性 data-on-focus="onFocus"

```
<a-text key="BZR_NAME" data-name="编制人姓名" data-on-focus="onFocus"></a-text>
```

添加script

```
<script>
  //示例 form:表单控件; $this:当前控件对象;
  function onFocus(form,$this){
    console.log(form)
    console.log($this)
  }
</script>
```

4.2.8 upload事件 附件上传成功事件

控件添加属性 data-on-upload="onUpload"

```
<a-image key="CANCEL_MANNAME" drag-attribute="false" data-name="附件" data-limit="100M" data-on-up
```

添加script

```
<script>
  //示例 form:表单控件; $this:当前控件对象;files 文件列表
  function onUpload(form,$this,files){
    console.log(form)
    console.log($this)
  }
</script>
```

4.2.9 delete事件 附件删除成功事件

控件添加属性 data-on-delete="onDelete"

```
<a-image key="CANCEL_MANNAME" drag-attribute="false" data-name="附件" data-limit="100M" data-on-de
```

添加script

```
<script>
  //示例 form:表单控件; $this:当前控件对象;files 文件列表
  function onDelete(form,$this,files){
    console.log(form)
    console.log($this)
  }
</script>
```

4.3 表单自定义JS编写规范

- 编写表单控件事件的时候,需要新增一个script标签,写入。表单生命周期方法默认读不到控件事件方法
- 若希望在表单生命周期里调用控件事件方法, 可将控件事件方法绑定到window上。
- 例子:window.changeLevel=changeLevel;

4.4 表单控件传递参数给弹出视图

- 支持在对应的弹框视图的SQL、默认值及查询条件下拉框弹出框中使用此表单参数。
- 默认值使用表单参数: \$u{引用的表单字段名}, SQL上使用表单参数: #{引用的表单字段名}
- 使用从表字段 变量为子表字段名称_FTF_字段名称

4.5 获取表单控件

4.5.1 主表控件

this.字段名.value 例子: this.BILLNO.value

4.5.1 一对多从表控件

this.从表名.value 例子: this.CONGBIAO.value

4.5.1 一对一从表控件

this['从表名.字段名'].value 例子: this['CB1.ID'].value

4.6 修改表单控件的值

4.6.1 主表控件

this.字段名.value 例子: this.BILLNO.value = 1

4.6.2 一对一从表控件

this['从表名.字段名'].value 例子: this['CB1.ID'].value = 1

4.6.3 一对多从表

```
//'cb_test1' 为从表名称
this['cb_test1'].value = (this['cb_test1'].value).map((item,index) => {
  if(index === 0){
    // 'BIG_CATEGORY_EN' 为从表字段名
    item['BIG_CATEGORY_EN'] = 'test'
  }
  return item;
})
```

4.7 控制表单编辑/只读状态

```
form.on('onRendered',function(data, prop, actions,form){  
    //设置控件只读  
    this.BILLNO.options.readonlyFormula = true;  
  
    //设置控件必填  
    this.STATION_NAME.options.requiredFormula = true;  
    this.STATION_NAME.required = true;  
  
    //设置控件隐藏  
    this.ShortText1657610340204.display = false;  
});
```

4.8 弹出自定义Vue组件

4.8.1 表单弹出框自定义组件

1. 表单控件类型选择弹出框

```
{  
    "弹出方式":"组件",  
    "组件路径" : "demo/index",//默认加载前端项目components目录下的文件 如:components/demo/index.vue组件,  
    "回调函数": "callback(backData, form)",//关闭弹框后需执行的回调函数,backData:页面返回的值;form:表单对象  
}
```

2. 编写回调函数

```
<script>  
    function callback(backData, form){  
        console.log(backData)  
        console.log(form)  
        //修改表单其他控件的值  
        form.BILLNO.value = backData.name;  
    }  
    window.callback=callback;  
</script>
```

3. 组件示例

```

<template>
  <div>
    <h3>自定义组件</h3>
    <el-button type="primary" @click="handleOk">确 定</el-button>
  </div>
</template>

<script>
export default {
  name: 'Component',
  props: {
    data: {
      type: Object
    }
  },

  mounted() {
    console.log(this.data)
  },
  methods: {
    handleOk(){
      let data = {
        id: '1',
        text: '1',
        name: '张三',
        age: '18'
      }
      //回显数据,同时执行回调方法
      this.$emit("complete",data);
      //关闭
      this.$emit("handlerCancel")
    }
  }
}
</script>

<style scoped>
</style>

```

4.8.2 表单按钮自定义组件

1. 表单按钮类型选择自定义

```

{
  "自定义类型": "组件",
  "组件路径" : "demo/index", //默认加载前端项目components目录下的文件 如: components/demo/index.vue组件,
  "参数": {"data": "viewtype", "name": "张三", "sex": "BBB"}, //传递给组件的参数
}

```

2. 组件示例

```

<template>
  <div>
    <h3>自定义组件</h3>
    <el-button type="primary" @click="handleClose">确 定</el-button>
  </div>
</template>

<script>
export default {
  name: 'Component',
  props: {
    data: {
      type: Object
    },
    //自定义参数
    customProps: {
      type: Object
    }
  },

  mounted() {
    console.log(this.customProps)
  },
  methods: {
    handleClose(done) {
      this.$confirm('确认关闭? ')
        .then(_ => {
          done()
        })
        .catch(_ => {})
    }
  }
}
</script>

<style scoped>
</style>

```

4.9 表单弹出框调控IFrame

1. 表单控件类型选择弹出框

```

{
  "弹出方式": "IFRAME",
  "链接地址" : "https://www.baidu.com"
  "回调函数": "callback(backData, form)",//关闭弹框后需执行的回调函数,backData:页面返回的值;form:表单对象
}

```

2. 编写回调函数

```

<script>
function callback(backData, form){
  console.log(backData)
  console.log(form)
  //修改表单其他控件的值
  form.BILLNO.value = backData.name;
}
window.callback=callback;
</script>

```

3. iframe示例页面

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Iframe弹框测试</title>
  </head>
  <body>
    <button onclick="ok()">确定</button>
    <button onclick="cancel()">取消</button>
  </body>
  <script>
    //打开弹框接收表单带过来的参数
    window.addEventListener('message',function(e){
      console.log(e.data);
    },false);

    //action等于ok表单关闭弹框，data即要回填到表单的值，id和text属性固定，可添加其它属性，在配置的回调函数使
    function ok(){
      var stringData = {
        action: 'ok', //操作类型 ok|确定 cancel|取消
        data:[ { id:'360001', text:'南昌', parent: '江西'},{ id:'380001', text:'福州', p
      }
      window.parent.postMessage(stringData, "*");
    }

    //action等于cancel关闭弹框，此时不会做任何操作，data可不传，不会对表单进行回填，仅关闭弹框
    function cancel(){
      var stringData = {
        action: 'cancel', //操作类型 ok|确定 cancel|取消
        data:[ { id:'370001', text:'武汉', parent: '湖北'},{ id:'360004', text:'萍乡', p
      }
      window.parent.postMessage(stringData, "*");
    }
  </script>
</html>

```

5. 自定义Vue页面开发

- 项目安装 th-render-component 包

```
npm i th-render-component --registry=http://192.168.14.114:8081/repository/npm-group/
```

- 在项目入口文件 main.js 引入 th-render-component 包

```

import formRender from "th-render-component"
import 'th-render-component/examples/assets/theme/render.scss'
Vue.use(formRender);

```

- 在项目中添加表单页面 form-detail.vue

```

<template>
  <pc-form-render
    :config-data="configData"
    :hasCloseBtn="hasCloseBtn"
    @changeCustomComponent="changeCustomComponent"
    @changeJS="changeJS"
    @dialogVisibleFalse="dialogVisibleFalse"
  >
  </pc-form-render>
</template>

<script>
export default {
  data() {
    return {
      PopupSon: null, //传递组件,
      configData: {},
      query: {},
    };
  },
  props: {
    // 表单配置
    config: Object,
    // 表单按钮回调
    btnCallBack: Function,
    // 是否有关闭按钮
    hasCloseBtn: {
      type: Boolean,
      default: false,
    },
    // 关闭按钮事件
    dialogVisibleFalse: {
      type: Function,
      default: () => {},
    },
  },
  created() {
    if (this.config) {
      this.query = JSON.parse(JSON.stringify(this.config));
    }
    let query = this.query;
    this.configData = {
      formCode: query.formCode || "",
      moduleCode: query.moduleCode || "",
      ownerService: query.ownerService || "",
      relevantModel: query.relevantModel || "",
      replaceVal: { openType: query.openType || "" },
      serviceName: query.serviceName || "",
      type: query.perms || "",
      model: query.model || "",
      objectId: query.objectId || "",
      appCode: query.appCode || "",
      addedProps: query.addedProps || {},
      extraData: query.extraData || {},
    };
  },
  provide() {
    return {
      configData: this.configData,
      btnCallBack: this.btnCallBack,
    };
  },
  methods: {
    changeCustomComponent(val, callback) {
      const component = (resolve) => {
        require(["@/components/" + val], resolve);
      };
      callback(component);
    },
    // 根据路径查找自定义JS
    changeJS(val, callback) {
      callback(require(`@/components/${val}`));
    }
  }
}

```



```

    },
  },
};
</script>

<style lang="scss" scoped>
</style>

```

- 在项目中添加视图页面 th-table-view.vue

```

<template>
  <th-table-view
    :config-data="configData"
    :single-mode="true"
    :style="{ height: 'calc(100%)' }"
    @changeCustomComponent="changeCustomComponent"
    @dialogCustomComponent="dialogCustomComponent"
    @changeJS="changeJS"
    @action="handleAction"
  ></th-table-view>
</template>

<script>
export default {
  props: {
    config: Object,
  },
  data() {
    return {
      configData: {
        appCode: this.config.appCode,
        moduleCode: this.config.moduleCode,
        serviceName: this.config.serviceName,
        viewCode: this.config.viewCode,
        extraData: this.config.extraData
      },
    };
  },
  methods: {
    changeJS(val, callback) {
      callback(require(`@/components/${val}`));
    },
    changeCustomComponent(val, callback) {
      const component = (resolve) => {
        require(["@/components/" + val], resolve);
      };
      callback(component);
    },
    // 自定义弹出框
    dialogCustomComponent(val, callback) {
      const component = (resolve) => {
        require(["@/views/" + val], resolve);
      };
      callback(component);
    }
  },
};
</script>

```

5.1 自定义页面打开视图

- 在自定义页面中添加以下代码

```

<template>
  <el-dialog
    title="弹出框"
    :visible.sync="dialogVisible"
    width="600px"
    :close-on-click-modal="false"
    append-to-body
  >
    <th-table-view
      :config="{
        appCode: '', // 项目编码
        moduleCode: '', // 模块编码
        serviceName: '', // 服务名称
        viewCode: '', // 视图编码
        extraData: {} // 视图查询参数
      }"
    />
    <span slot="footer" class="dialog-footer">
      <el-button type="primary" @click="dialogVisible = false">确 定</el-button>
    </span>
  </el-dialog>
</template>

<script>
// 引入视图组件，文件地址为th-table-view.vue放入项目的路径地址
import ThTableView from "@views/view/th-table-view.vue";

export default {
  name: "Component",
  components: {
    ThTableView,
  },
  data() {
    return {
      dialogVisible: true
    }
  }
};
</script>
<style>
</style>

```

- 通过控制 dialogVisible 变量的值实现视图的打开和关闭

5.2 自定义页面打开表单

- 在自定义页面中添加以下代码

```

<template>
  <div class="form-detail-wrap" v-if="formVisible">
    <form-detail
      :config="{
        model: 'flow', // flow为流程表单 form为普通表单
        perms: 'edit', // add为新增 edit为编辑 detail为查看
        serviceName: 'default', // 服务名
        moduleCode: 'lichao20210707', // 模块名称
        appCode: 'test-app', // 项目编码
        viewCode: 'lichao20210707-view', // 视图编码
        formCode: 'lichao20210707', // 表单编码
        openType: 0,
        objectId: '12331231234555', // 编辑和查看表单需要此属性-值为表单主键
        ownerService: 'default', // 流程表单使用
        taskId: 'e62d237f-0e46-11ec-9f93-5254009bc984', // 流程表单使用
        processInsId: '49d55b6c-00e6-11ec-9159-5254009bc984', // 流程表单使用
      }"
      :btnCallBack="btnCallBack"
      :hasCloseBtn="true"
      :dialogVisibleFalse="dialogVisibleFalse"
      :persistSuccess="persistSuccess"
    />
  </div>
</template>

<script>
// 引入表单组件, 文件地址为form-detail.vue 放入项目的路径地址
import FormDetail from "@views/form/form-detail";

export default {
  name: "Component",
  components: {
    FormDetail,
  },
  data() {
    return {
      formVisible: true,
    };
  },
  methods: {
    // 普通表单按钮回调事件
    btnCallBack(ac) {
      // 通过ac的code属性判断按钮类型
      // 1.save 保存和更新
      // 2.custom 自定义
      // 通过ac的text属性判断按钮文字
      // 例如: ac.text === "自定义1"
      // 注意: 自定义按钮js如果触发了saveForm方法, 此表单按钮回调会被再次触发, 此时的code=== "save"
      if (ac.code === "save") {
        // do something
      }
      // 可以通过code和text区分不同的自定义按钮
      if (ac.code === "custom" && ac.text === "自定义1") {
        // do something
      }
    },
    // 表单关闭回调事件 (点击表单关闭按钮或者在自定义按钮中调用closeForm方法都会触发此回调) (普通表单和流程表单)
    dialogVisibleFalse() {
      this.formVisible = false;
    },
    // 流程按钮触发回调
    persistSuccess() {
      this.formVisible = false;
    }
  },
};
</script>

```

```

<style>
.form-detail-wrap {
  position: fixed;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background: #fff;
  z-index: 99;
}
/* 关闭按钮样式覆盖 */
.form-wrap .form-actions .dialog-close-btn {
  color: #09b9a2;
}
</style>

```

- 通过控制 formVisible 变量值实现表单的打开和关闭

5.3 自定义页面嵌入视图

- 在自定义页面中添加以下代码

```

<template>
  <th-table-view
    :config="{
      appCode: '', // 项目编码
      moduleCode: '', // 模块编码
      serviceName: '', // 服务名称
      viewCode: '', // 视图编码
      extraData: {} // 视图查询参数
    }"
  />
</template>

<script>
// 引入视图组件，文件地址为th-table-view.vue放入项目的路径地址
import ThTableView from "@views/view/th-table-view.vue";

export default {
  name: "Component",
  components: {
    ThTableView,
  }
};
</script>
<style>
</style>

```

5.4 自定义页面嵌入表单

- 在自定义页面中添加以下代码

```

<template>
  <form-detail
    :config="{
      model: 'flow', // flow为流程表单 form为普通表单
      perms: 'edit', // add为新增 edit为编辑 detail为查看
      serviceName: 'default', // 服务名
      moduleCode: 'lichao20210707', // 模块名称
      appCode: 'test-app', // 项目编码
      viewCode: 'lichao20210707-view', // 视图编码
      formCode: 'lichao20210707', // 表单编码
      openType: 0,
      objectId: '12331231234555', // 编辑和查看表单需要此属性-值为表单主键
      ownerService: 'default', // 流程表单使用
      taskId: 'e62d237f-0e46-11ec-9f93-5254009bc984', // 流程表单使用
      processInsId: '49d55b6c-00e6-11ec-9159-5254009bc984', // 流程表单使用
    }"
    :btnCallBack="btnCallBack"
    :hasCloseBtn="false"
    :persistSuccess="persistSuccess"
  />
</template>

<script>
// 引入表单组件，文件地址为form-detail.vue 放入项目的路径地址
import FormDetail from "@views/form/form-detail";

export default {
  name: "Component",
  components: {
    FormDetail,
  },
  methods: {
    // 普通表单按钮回调事件
    btnCallBack(ac) {
      // 通过ac的code属性判断按钮类型
      // 1.save 保存和更新
      // 2.custom 自定义
      // 通过ac的text属性判断按钮文字
      // 例如: ac.text === "自定义1"
      // 注意: 自定义按钮js如果触发了saveForm方法，此表单按钮回调会被再次触发，此时的code=== "save"
      if (ac.code === "save") {
        // do something
      }
      // 可以通过code和text区分不同的自定义按钮
      if (ac.code === "custom" && ac.text === "自定义1") {
        // do something
      }
    },
    // 流程按钮触发回调
    persistSuccess() {}
  },
};
</script>

<style>
</style>

```

6. 接口自定义开发规范

6.1 常用方法规范

- GET: SELECT 获取(查询)资源
- POST: CREATE 创建(添加)资源

- PUT: UPDATE 更新(修改)资源
- DELETE: DELETE 删除资源

6.2 传入参数

后端代码示例

```
@PostMapping(value = "{appId}/{moduleId}/update")
public R<String> editAll(@PathVariable(name = "appId") String appId,
                        @PathVariable(name = "moduleId") String moduleId,
                        @RequestBody @Valid ViewModelV0 viewModelV0) {
    return R.ok("修改成功");
}
```

方法解析:

1. @PostMapping: value为请求路径, 此为post请求, 等同于@RequestMapping(value = "{appId}/{moduleId}/update", method = RequestMethod.POST); 若get请求, 可使用@GetMapping
2. @PathVariable: 映射URL绑定的占位符, 绑定到操作方法的入参中
3. @RequestBody: 将前端传来的json格式的数据转为自己定义好的javabean对象

前端代码示例

```
//在form.js中定义方法
export function editFormConfig(appId, moduleId, data) {
    return request({
        url: '/gen/design/form/' + appId + '/' + moduleId + '/update',
        method: 'post',
        data: data
    })
}

//从form.js中导入方法
import { editFormConfig } from '@api/gen/online/form'
//使用方法
editFormConfig(this.appId, this.moduleId, cloneData).then(result => {
    //处理方法返回值
});
```

6.3 响应参数

所有方法返回**R**对象, 返回值放在**R**对象中, 代码示例如下:

```
public R<List<PopupTree>> getTreeTableData() {
    List<PopupTree> treeTableTData = new ArrayList<>();
    //正常返回
    //return R.ok(treeTableTData);
    //发生错误时返回, code为1
    return R.failed("异常");
}
```

其中, **R**对象中参数如下:

```

@ApiModel("响应信息主体")
public class R<T> implements Serializable {
    @ApiModelProperty("返回标记: 成功标记=0, 失败标记=1, 自定义错误编码=xxx")
    private int code;
    @ApiModelProperty("业务异常编码")
    private String bizCode;
    @ApiModelProperty("返回信息")
    private String msg;
    @ApiModelProperty("数据")
    private T data;
}

```

前端处理返回参数，代码示例如下

```

editFormConfig()
    .then(result => {
        if (result.data.code == 0) {
            //正常返回
        } else {
            //失败发挥
        }
    })
    .catch(() => {
        //发生异常
    })

```

6.4 异常类

抛出异常请使用**BizException**类，若自定义异常类请继承**BizException**类；异常枚举类需实现**ThAbstractErrorEnum**；代码示例如下

```

throw new BizException(FileStorageErrorEnum.UPLOAD_ERROR);

```

6.5 权限控制

资源管理增加按钮

在系统管理-资源管理，找到对应资源，找到对应资源下新增按钮，例如新增按钮资源标识为**test_api_right**，可给此按钮增加角色实现权限控制。

后端接口增加权限判断

需要api接口增加注解 **PreAuthorize**，代码示例如下：

```

@PostMapping(value =("/{appId}/{moduleId}/update"))
@PreAuthorize("@pms.hasPermission('test_api_right')")
public R<String> editAll(@PathVariable(name = "appId") String appId,
                        @PathVariable(name = "moduleId") String moduleId,
                        @RequestBody @Valid ViewModelV0 viewModelV0) {

    return R.ok("修改成功");
}

```

前端接口增加权限判断

可在前端增加权限判断按钮是否显示，代码如下：

```
//引入权限判断js
import { checkPerm } from '../../../util/permission'
//定义对象
data() {
  return {
    permissions: {
      test_api_right: checkPerm.call(this, ['test_api_right'])
    }
  }
}
//按钮显示增加权限判断
<el-button v-if="permissions.test_api_right" icon="el-icon-plus" size="mini" type="primary" @click
```





7. 平台安全配置参数说明

安全配置表为：CLOUDX.T_TBP_SECURITY_CONFIG, 配置信息缓存在redis中
(tidp:1:security_config::props),直接从数据库中修改安全配置表相关配置时不会生效

参数名	参数值
audit.log	on
audit.event.type	系统事件,业务事件
security.log.warnsize	1024
audit.time.range	00:00-23:59
audit.user.role	sysadmin,bizadmin,auditadmin,approveadmin,ordinar
login.ip.ChangeNum	3
login.lockSecondsAfterFailed	1200

参数名	参数值
login.maxFailureTimes	3
login.maxOnlineUser	50000
login.password.charMatch	4
login.password.dynamic.token	off

参数名	参数值
login.password.forceChangeAtFirstLogin	on
login.password.minLength	8
login.password.maxLength	20
login.user.onlyOneComputerPerUser	on
login.user.onlyOneUserPerComputer	on
login.password.remindCycle	60
login.password.repeat.num	1
login.session.timeout	30

参数名	参数值
security.shield.hostWhiteList	127.0.0.1:8080
security.shield.csrfWhiteList	https://127.0.0.1  , http://127.0.0.1 
security.shield.serviceUrlList	https://127.0.0.1  , http://127.0.0.1 
security.shield.resourceWhiteList	.js .css .ico .jpg .png .svg

参数名	参数值
security.shield.sql	on
security.shield.whiteUrls	
security.shield.xss	on
security.shield.ip	off
security.shield.ip.range	192.168.14.1~192.168.14.255

参数名	参数值
security.user.checkuserstatus	on
security.user.lastlogintime.days	90
security.user.sleep.alert.days	30
security.user.sleep.days	90

微服务在nacos中配置的安全参数

cloud-gateway-dev.yml

`security.open-code:true
cloud.security.filter.isLimitIPInterceptor:true
cloud.security.filter.ipLimitCount: 5
cloud.security.filter.csrfWhiteList: http://127.0.0.1
cloud.security.filter.hostWhiteList:127.0.0.1:8080
cloud.security.filter.whileUrl: /test/user`

#验证码开关
#是否开启IP非法请求次数限制
#IP非法请求过多次数限制
#referer白名单配置，为空不校验，多个分号隔开
#host白名单配置，为空不校验，多个分号隔开
#url白名单配置，多个逗号隔开

application-dev.yml

`security.separationOfFourPowers:false
security.viewButtonPerm:false`

#四权开关
#表单视图接口权限校验开关

cloud-upms-biz-dev.yml

`cloud.encodingId:bcrypt
security.not-permit-users: admin,sys,test,guest,anonymous`

#存储密码加密类型，默认bcrypt，国密sm，md5，m
#不允许新建账号白名单

莫邪平台配置文件中安全参数

我们要确认莫邪平台使用的权限管理模式：四权分立权限管理模式\基于超级管理权限管理模式；

前端项目查看config.js配置文件查看配置参数SYSMGR_SOFP（V1.3.1版本之前是SEPARATION_OF_FOUR_POWERS）,true表示采用的四权分立管理模式；

```
//四权分立
SYSMGR_SOFP: true
```

后台项目查看application-dev.yml配置参数:

```
security.separationOfFourPowers:true
```

前后端对是否开启四权分立的配置必须保持一致，即不可前端配置true后端配置false;

存储密码加密类型配置：

```
#存储密码加密类型，默认bcrypt，国密sm，md5，md5x hash算法
cloud.encodingId:bcrypt
```

验证码开关配置：

查看前端config.js配置文件中配置的登录认证方式：

```
//认证方式 oauth or jwt
AUTH_TYPE: 'oauth',
//true 莫邪登录 false 微服务登录
useTile: true,
//认证服务
AuthAddress: "http://127.0.0.1:9875/admin",
```

AUTH_TYPE配置的是oauth认证方式，认证服务地址配置的是微服务网关时，验证码开关在nacos的cloud-gateway-dev.yml文件中设置：

```
//true:开启验证码，false:关闭验证码校验
security.open-code: true
```

AUTH_TYPE配置的是jwt认证时验证码校验配置在application-dev.yml中：

```
//true:开启验证码，false:关闭验证码校验
security.open-code: true
```

8. 多租户配置

多租户为各地区部、产品线在同一套环境上提供数据隔离技术，每个租户拥有独立的数据库。通常只有平台的cloud-auth、cloud-upms-biz、cloud-code-gen、cloud-file-web、cloud-information-web、cloud-default-web等服务需要开启服务端多租户模式，其他业务服务都开启客户端多租户模式

8.1 服务端多租户

服务端多租户模式：通过请求头中的租户ID参数，动态切换至对应的租户数据源

默认服务端多租户模式是关闭状态，开启服务端多租户模式：

1. 启动类上添加了@EnableTenantServer
2. 配置文件中添加服务端多租户配置

```
cloud:
  tenant-server:
    enable: true
```

8.2 客户端多租户

客户端多租户模式：通过Filter拦截请求头中的租户ID参数，内部进行远程调用或者发送请求时，会在请求头中带上租户ID。例如业务微服务需要远程调用平台的cloud-upms-biz服务，则需要开启客户端多租户模式

默认客户端多租户模式是关闭状态，开启客户端多租户模式：

1. 启动类上添加了@EnableTenantClient
2. 配置文件中添加客户端多租户配置

```
cloud:
  tenant-client:
    enable: true
```