**ECMP3.0平台开发指南**

**绵阳研发部**

**四川虹信软件股份有限公司**

**Sichuan Rainbow Consulting&Software Co., Ltd.**

**目录**

[1 开发环境准备 7](#_Toc528832436)

[1.1 环境 7](#_Toc528832437)

[1.1.1 操作系统 7](#_Toc528832438)

[1.1.2 数据库 7](#_Toc528832439)

[1.1.3 JDK 7](#_Toc528832440)

[1.1.4 应用服务器 7](#_Toc528832441)

[1.2 工具 7](#_Toc528832442)

[1.2.1 开发工具 7](#_Toc528832443)

[1.2.2 构建工具 7](#_Toc528832444)

[1.2.3 版本管理工具 9](#_Toc528832445)

[1.2.4 数据建模工具 10](#_Toc528832446)

[1.3 本地配置 10](#_Toc528832447)

[1.3.1 用户变量JAVA\_HOME 10](#_Toc528832448)

[1.3.2 用户变量GRADLE\_HOME 10](#_Toc528832449)

[1.3.3 用户变量MAVEN\_HOME 11](#_Toc528832450)

[1.3.4 系统变量Path 11](#_Toc528832451)

[1.3.5 用户变量Path 11](#_Toc528832452)

[2 项目准备 12](#_Toc528832453)

[2.1 项目名 12](#_Toc528832454)

[2.2 建数据库 12](#_Toc528832455)

[2.3 配置管理员反馈信息 12](#_Toc528832456)

[2.4 克隆项目 12](#_Toc528832457)

[2.4.1 HTTP方式克隆 12](#_Toc528832458)

[2.4.2 SSH方式克隆 15](#_Toc528832459)

[2.5 初始目录 19](#_Toc528832460)

[2.5.1 项目根目录下的配置文件 19](#_Toc528832461)

[2.5.2 模块创建 19](#_Toc528832462)

[3 模型设计 20](#_Toc528832463)

[3.1 概述 20](#_Toc528832464)

[3.2 设计模型 20](#_Toc528832465)

[3.3 导出脚本 21](#_Toc528832466)

[3.4 归档数据模型 23](#_Toc528832467)

[3.5 更新数据库并更新脚本 24](#_Toc528832468)

[3.6 Flyway 26](#_Toc528832469)

[3.6.1 通过flyway执行脚本 26](#_Toc528832470)

[3.6.2 Flyway命令介绍 28](#_Toc528832471)

[3.7 数据库表的设计 28](#_Toc528832472)

[3.7.1 数据库编码 28](#_Toc528832473)

[3.7.2 数据库设计命名规则 29](#_Toc528832474)

[3.7.3 索引 29](#_Toc528832475)

[4 Entity 30](#_Toc528832476)

[4.1 概述 30](#_Toc528832477)

[4.2 配置文件build.gradle 30](#_Toc528832478)

[4.2.1 编码 30](#_Toc528832479)

[4.2.2 插件 30](#_Toc528832480)

[4.2.3 添加依赖 30](#_Toc528832481)

[4.2.4 flyway配置 31](#_Toc528832482)

[4.2.5 上传Maven仓库 31](#_Toc528832483)

[4.2.6 上传到Git仓库操作 32](#_Toc528832484)

[4.3 命名规则 33](#_Toc528832485)

[4.4 类注解 34](#_Toc528832486)

[4.4.1 @Access(AccessType.FIELD) 34](#_Toc528832487)

[4.4.2 @DynamicInsert和@DynamicUpdate 34](#_Toc528832488)

[4.4.3 @Entity 34](#_Toc528832489)

[4.4.4 @Table(name = “user\_account”) 34](#_Toc528832490)

[4.4.5 @Cacheable 34](#_Toc528832491)

[4.5 继承基类和实现对应的业务实体类型 35](#_Toc528832492)

[4.5.1 继承BaseEntity 35](#_Toc528832493)

[4.5.2 继承BaseAuditableEntity 35](#_Toc528832494)

[4.5.3 继承AbstractEntity 35](#_Toc528832495)

[4.5.4 实现对应的业务实体类型接口 35](#_Toc528832496)

[4.6 属性上的注解 35](#_Toc528832497)

[4.6.1 @Column(name = "code", length = 20, unique = true, nullable = false) 35](#_Toc528832498)

[4.6.2 临时字段 36](#_Toc528832499)

[4.6.3 外键关联 36](#_Toc528832500)

[4.6.4 枚举 36](#_Toc528832501)

[4.6.5 大字段 37](#_Toc528832502)

[4.6.6 时间戳 37](#_Toc528832503)

[5 Dao 37](#_Toc528832504)

[5.1 概述 37](#_Toc528832505)

[5.2 配置文件build.gradle 38](#_Toc528832506)

[5.2.1 添加插件 38](#_Toc528832507)

[5.2.2 编码 38](#_Toc528832508)

[5.2.3 添加依赖 38](#_Toc528832509)

[5.2.4 打包 38](#_Toc528832510)

[5.3 命名规则 39](#_Toc528832511)

[5.4 接口注解 39](#_Toc528832512)

[5.5 接口继承 39](#_Toc528832513)

[5.6 接口扩展 39](#_Toc528832514)

[5.7 接口实现 40](#_Toc528832515)

[6 API 40](#_Toc528832516)

[6.1 概述 40](#_Toc528832517)

[6.2 配置文件build.gradle 40](#_Toc528832518)

[6.2.1 编码 40](#_Toc528832519)

[6.2.2 添加依赖 40](#_Toc528832520)

[6.2.3 上传Maven仓库 41](#_Toc528832521)

[6.3 接口命名 42](#_Toc528832522)

[6.4 包注解 42](#_Toc528832523)

[6.5 接口注解 42](#_Toc528832524)

[6.5.1 接口的相对路径 42](#_Toc528832525)

[6.5.2 接收的参数类型 43](#_Toc528832526)

[6.5.3 返回数据类型 43](#_Toc528832527)

[6.5.4 Api接口的介绍 43](#_Toc528832528)

[6.6 继承接口 43](#_Toc528832529)

[6.7 方法注解 43](#_Toc528832530)

[6.7.1 操作 43](#_Toc528832531)

[6.7.2 方法路径 44](#_Toc528832532)

[6.7.3 方法的介绍 45](#_Toc528832533)

[6.7.4 保存的返回类型 45](#_Toc528832534)

[6.7.5 删除的返回类型 45](#_Toc528832535)

[6.8 通过swagger查看API文档 45](#_Toc528832536)

[6.8.1 本地swagger地址 45](#_Toc528832537)

[7 Service 46](#_Toc528832538)

[7.1 概述 46](#_Toc528832539)

[7.2 配置文件build.gradle 46](#_Toc528832540)

[7.2.1 编码 46](#_Toc528832541)

[7.2.2 插件 46](#_Toc528832542)

[7.2.3 war包名称和路径 47](#_Toc528832544)

[7.2.4 添加依赖 47](#_Toc528832545)

[7.3 应用配置 47](#_Toc528832546)

[7.4 添加主类 48](#_Toc528832547)

[7.4.1 Jar包方式启动 48](#_Toc528832548)

[7.4.2 War包方式启动 48](#_Toc528832549)

[7.5 命名 49](#_Toc528832550)

[7.6 类注解 49](#_Toc528832551)

[7.7 继承类和实现接口 49](#_Toc528832552)

[7.7.1 继承类 49](#_Toc528832553)

[7.7.2 实现接口 49](#_Toc528832554)

[7.8 编写代码 49](#_Toc528832555)

[7.8.1 依赖注入 49](#_Toc528832556)

[7.8.2 构造函数 49](#_Toc528832557)

[7.8.3 事务控制 50](#_Toc528832558)

[7.8.4 给号器代码 50](#_Toc528832559)

[7.8.5 业务缓存 50](#_Toc528832560)

[7.8.6 清除缓存 51](#_Toc528832561)

[7.9 资源 51](#_Toc528832562)

[7.10 单元测试 51](#_Toc528832563)

[7.10.1 测试基类 51](#_Toc528832564)

[7.10.2 测试类 51](#_Toc528832565)

[8 Web 53](#_Toc528832566)

[8.1 概述 53](#_Toc528832567)

[8.2 配置文件build.gradle 53](#_Toc528832568)

[8.2.1 编码 53](#_Toc528832569)

[8.2.2 插件 54](#_Toc528832570)

[8.2.3 War包名称和路径 54](#_Toc528832572)

[8.2.4 添加依赖 54](#_Toc528832573)

[8.2.5 Jar包 54](#_Toc528832574)

[8.3 应用配置 55](#_Toc528832581)

[8.4 多语言 55](#_Toc528832594)

[8.5 控制器 55](#_Toc528832595)

[8.5.1 添加主类 55](#_Toc528832596)

[8.5.2 Jar包方式启动 55](#_Toc528832597)

[8.5.3 War包方式启动 56](#_Toc528832598)

[8.5.4 命名 56](#_Toc528832599)

[8.5.5 类注解 56](#_Toc528832600)

[8.5.6 继承类 56](#_Toc528832601)

[8.5.7 方法注解 56](#_Toc528832602)

[8.5.8 编码实现 57](#_Toc528832603)

[Html页面 57](#_Toc528832604)

[8.6 57](#_Toc528832605)

[8.6.1 文件存放路径 57](#_Toc528832606)

[8.6.2 页面命名 58](#_Toc528832607)

[8.6.3 页面格式 58](#_Toc528832608)

[8.6.4 Js文件 59](#_Toc528832609)

[8.6.5 Css文件 59](#_Toc528832610)

[8.6.6 多语言lang\_zh\_CN.js 59](#_Toc528832611)

[9 签入和发布 59](#_Toc528832612)

[9.1 代码签入 59](#_Toc528832613)

[9.2 代码发布 61](#_Toc528832614)

[10 注册功能项和菜单 62](#_Toc528832615)

[10.1 概述 62](#_Toc528832616)

[10.2 配置功能项 62](#_Toc528832617)

[10.2.1 创建功能项组 62](#_Toc528832618)

[10.2.2 创建对应的功能项 63](#_Toc528832619)

[10.3 配置菜单 64](#_Toc528832620)

[10.3.1 创建根节点 64](#_Toc528832621)

[10.3.2 创建子节点 64](#_Toc528832622)

[11 异常日志 65](#_Toc528832623)

[11.1 服务端的异常日志 65](#_Toc528832624)

[11.2 需要特殊记录的异常日志 65](#_Toc528832626)

[11.3 查看异常日志 65](#_Toc528832627)

[12 附件 65](#_Toc528832628)

# 开发环境准备

软件及工具的下载地址见文档末尾附件

## 环境

### 操作系统

支持Windows和Linux等主流操作系统，推荐使用Linux Redhat 7版本，如要使用Windows，推荐用windows server 16版本，两种均选择64位的操作系统

### 数据库

支持MySQL 5.0及以上版本、PostgreSQL 9及以上版本、Oracle 10g及以上版本、SQL Server 2005及以上版本等数据库，推荐使用MySQL 5.6版本

### JDK

使用oracle jdk8及以上版本

### 应用服务器

支持Tomcat、JBoss(WildFly)、WebLogic、WebSphere，推荐使用Tomcat 8版本

## 工具

### 开发工具

IDEA、Eclipse JavaEE和MyEclipse，推荐使用IDEA2016版本

### 构建工具

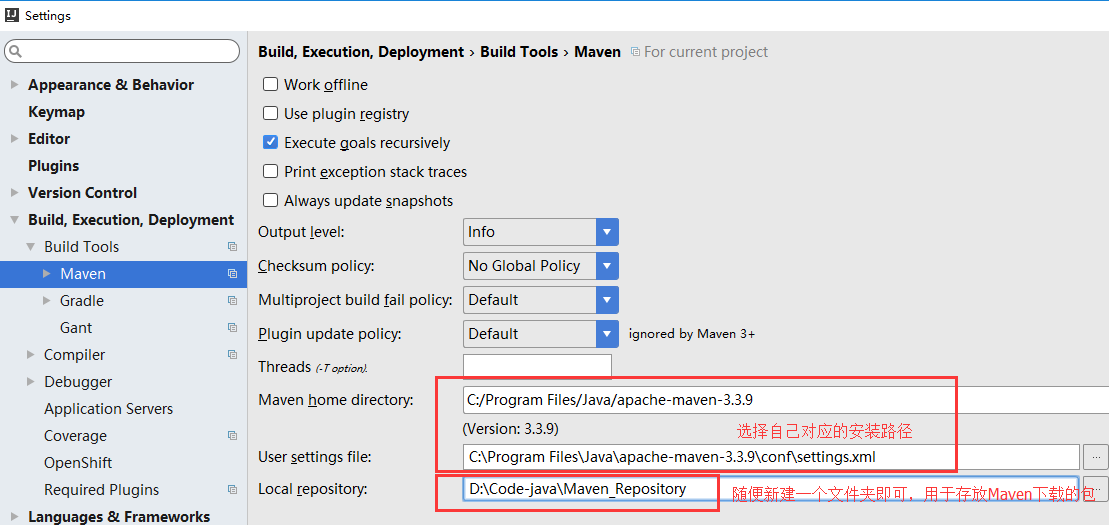
#### 工具

1. 使用Gradle作为构建工具，推荐使用3.4.1版本
2. Gradle需要使用Maven仓库，Maven推荐使用3.3.9版本

#### 在开发工具中添加插件

在IDEA启动页面点击右下角configure，选择Settings --> Build,Execution,Deloyment --> Build Tools --> Maven：

1. 在右侧栏的Maven\_home directory输入框中填入自己的Maven安装目录，例如：C:/Program Files/maven/apache-maven-3.3.9
2. 在右侧栏的User settings file输入框中填入Maven的setting.xml目录，例如：C:/Program Files/maven/apache-maven-3.3.9/conf/settings.xml
3. 在右侧栏的Local repository输入框中填入自己存放Maven下载的包的文件夹，此文件夹任意创建，例如：D:/Code-java/Maven\_Repository
4. 在左侧栏里点击Build Tools下的Gradle，在右侧栏里的Service directory path中填入自己存放Gradle下载的包的文件夹路径，例如：D:/Code-java/Gradle\_Repository





**如Build Tools里没有Gradle，在启动页面点击configure，选择Plugins，在里面搜索Gradle，勾选中后点OK，再进入Settings**

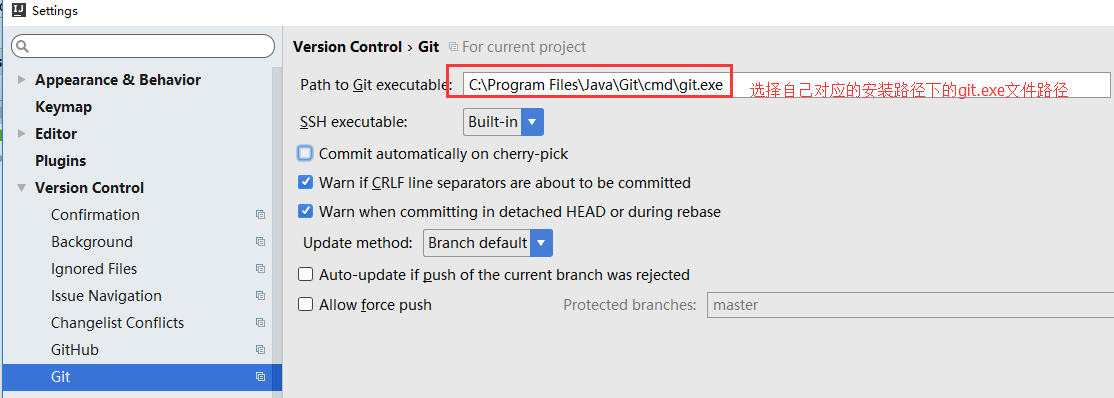
### 版本管理工具

#### 工具

可以使用Git、CVS和SVN，推荐使用Git作为版本控制器

#### 在开发工具中添加插件

在IDEA启动页面点击右下角configure，选择Settings --> Version Control --> Git,在右侧栏里的Path to Git executable里填入自己的git.exe文件路径



### 数据建模工具

使用PowerDesigner作为模型设计工具，推荐使用16.5版本

## 本地配置

### 用户变量JAVA\_HOME

选择自己的JDK的安装目录，例：C:/Program Files/Java/jdk1.8.0\_45

### 用户变量GRADLE\_HOME

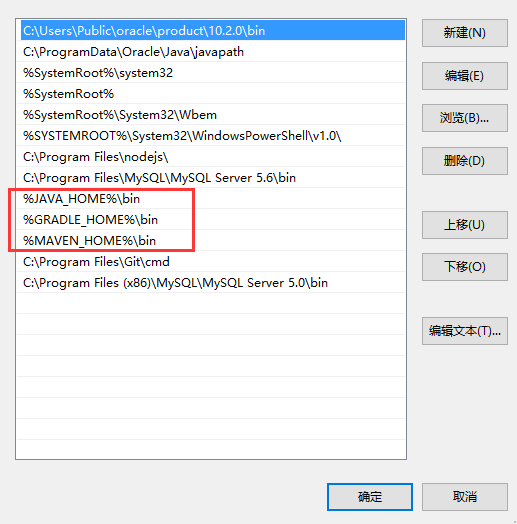
选择Gradle的安装目录，例：E:/Program Files/gradle-3.4.1

### 用户变量MAVEN\_HOME

选择自己的Maven的安装目录，例：E:/Program Files/maven/apache-maven-3.3.9

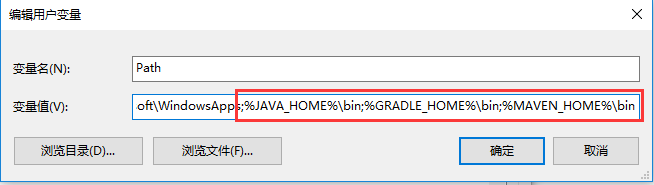
### 系统变量Path

在系统变量Path里添加三项，如下图



### 用户变量Path

在用户变量Path里把系统变量Path添加的三项加进去



# 项目准备

配置管理员：

姓名：冯华

QQ：1247170157

邮箱：[hua.feng@changhong.com](mailto:hua.feng@changhong.com)

## 项目名

确定新建项目的名称并向配置管理员申请新建项目，例如要建一个演示项目，中文名称为：示例项目，英文名称为：demo(最大长度为10)

## 建数据库

确定数据库名称和数据库类型并向配置管理员申请新建数据库

## 配置管理员反馈信息

配置管理员根据开发人员提供的数据库名称、类型和项目名进行相关配置，然后向开发人员反馈Git仓库地址、应用模块代码、应用标识ID、数据库地址、数据库账号、密码等

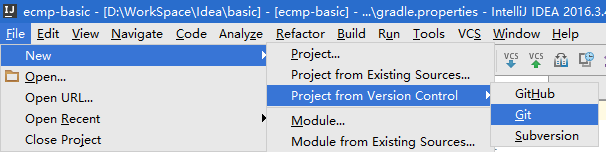
## 克隆项目

### HTTP方式克隆

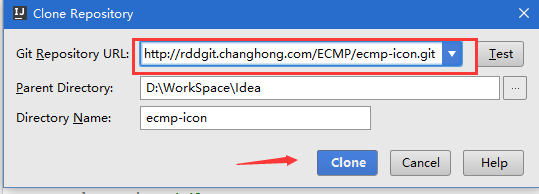
1. 在Git仓库里点击所创建的项目，选择http克隆方式，再点击复制地址按钮



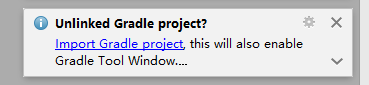
1. 在IDEA软件里点击File-->New-->Project from Version Control-->Git

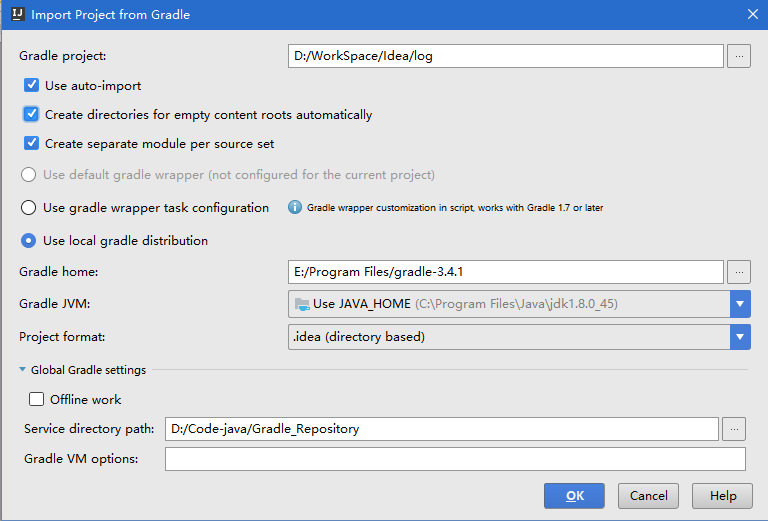


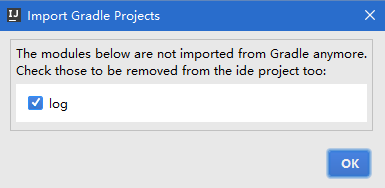
1. 在弹窗中粘贴项目的Git地址，点击Clone

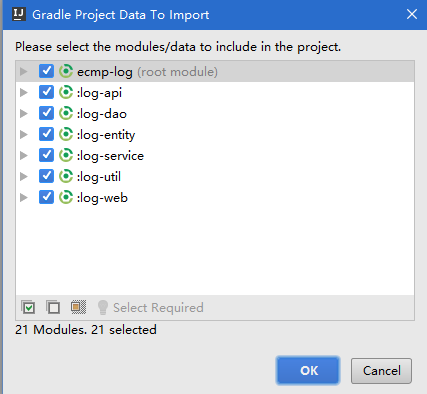


1. 克隆项目后，点击右下角弹窗的蓝色字体Import Gradle project，在新的弹窗里选中USe auto-import，点击OK，在之后的弹窗里均点击OK，Gradle项目就创建成功了









### SSH方式克隆

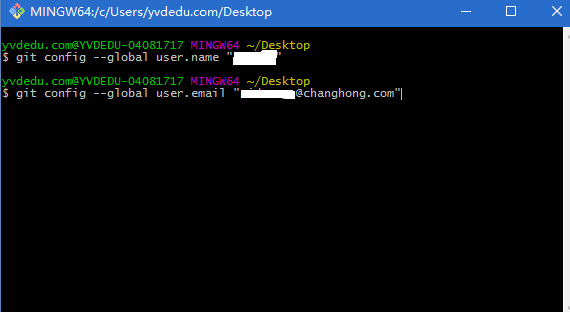
1. 在桌面空白处右键选择Git Bash Here，在窗口中执行以下代码：

//用户名

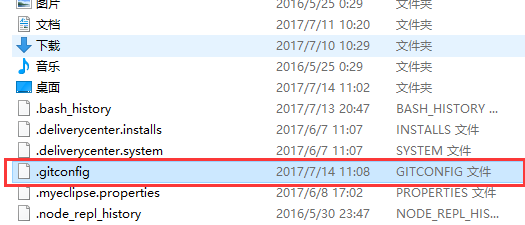
$ git config --global user.name "某某某"

//邮箱

$ git config --global user.email "xxx@changhong.com"



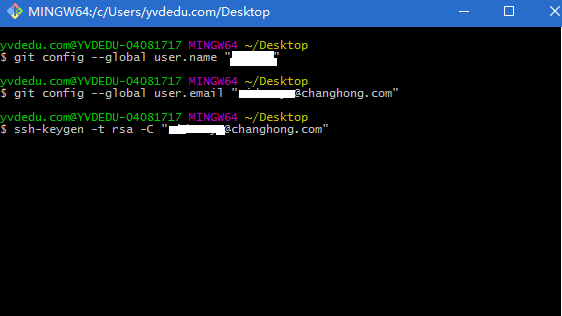
在C盘搜索.gitconfig文件，打开文件查看信息是否正确



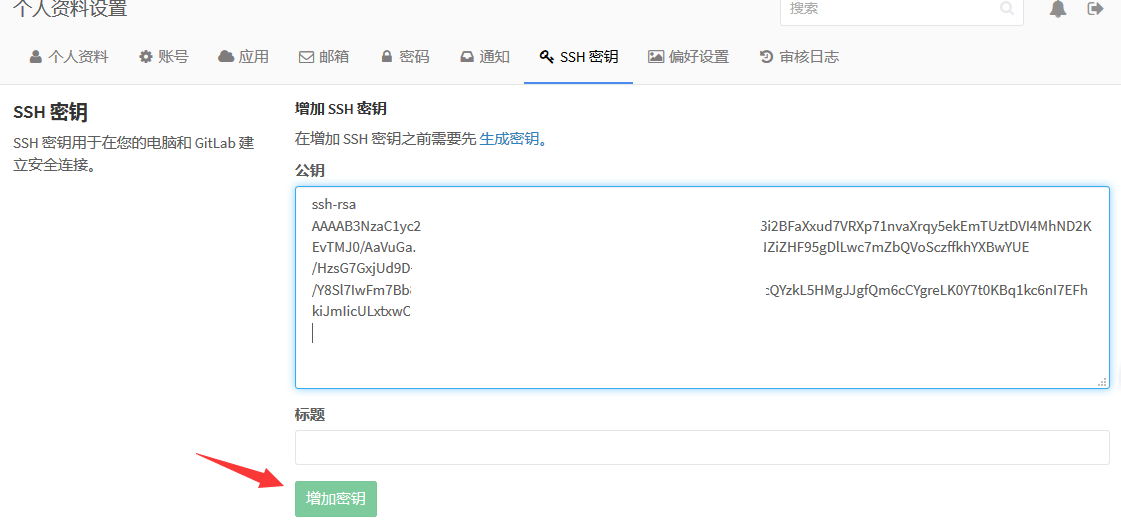
1. 继续执行以下代码：

$ ssh-keygen -t rsa -C “[xx.x@changhong.com](mailto:xx.x@changhong.com)”

按3个回车，密码为空



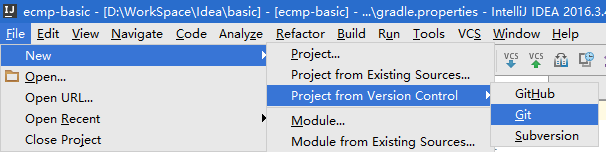
完成后在.gitconfig文件的同层目录会生成一个.ssh文件夹，打开其中的id\_rsa.pub文件，复制全部代码，最后在GitLab中打开个人资料设置，选择SSH密钥粘贴代码，标题会自动添加



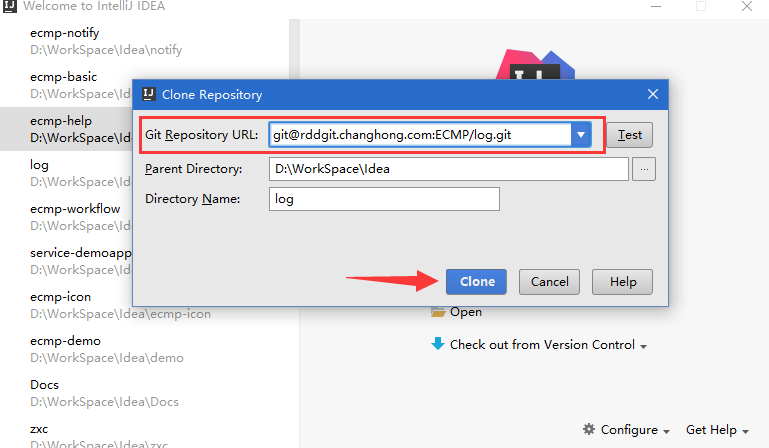
1. 在Git仓库里点击所创建的项目，选择SSH克隆方式，再点击复制地址按钮



1. 在IDEA软件里点击File-->New-->Project from Version Control-->Git



1. 在弹窗中粘贴复制的地址，点击Clone



后续步骤同2.4.1第4步

## 初始目录

### 项目根目录下的配置文件

#### .gitignore

用于配置不需要加入版本管理的文件

#### gradle.properties

该配置文件包含了组织、版本、组件版本、maven仓库和flyway的设置

1. 每次发版后需要将版本号升1，例如当前版本为1.0.1，发版后版本改为1.0.2

# 版本

VERSION = 1.0.1

1. 需要在原有代码的基础上增加以下flyway配置：

# ####################### flyway ##############################

flyway\_driver = com.mysql.jdbc.Driver

//将10.4.68.45:3306/service\_demoapp替换为自己的数据库地址、端口和名称

flyway\_url =jdbc:mysql://10.4.68.45:3306/service\_demoapp?characterEncoding=utf8

//数据库账号

flyway\_user = xx

//数据库密码

flyway\_password = xxxxxx

#### README.md

项目介绍的markdown文档

#### settings.gradle

设置项目结构，当在项目里新建一个模块时，该配置文件会自动生成代码

### 模块创建

根据项目的实际需要创建Entity、Dao、Service、Api、Web

# 模型设计

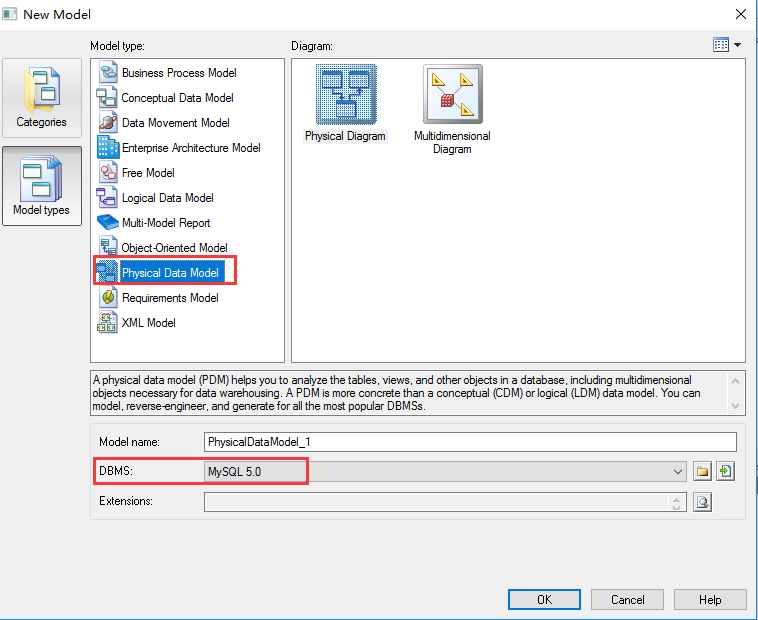
## 概述

根据数据库表进行数据库建模并生成数据库脚本

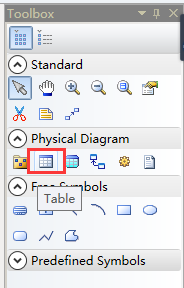
## 设计模型

利用PowerDesigner里的PDM创建数据库。以Mysql为例，点击【File】-->【New Model】

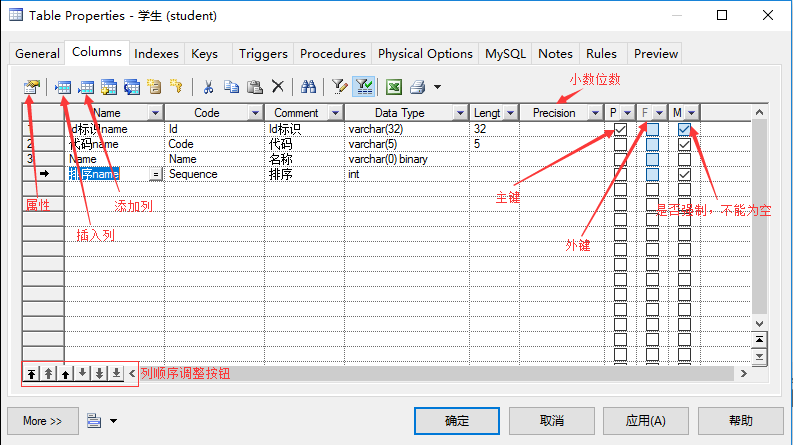
1. Model type选择Physical Data Model
2. Diagram选择Physical Diagram
3. 下面的DBMS下拉选项选择MySQL 5.0



在PDM的图形窗口中，单击工具选项版上的Table工具，再单击图形窗口的空白处，在单击的位置就出现一个空白的Table。点击Pointer工具或右击鼠标，释放Table工具

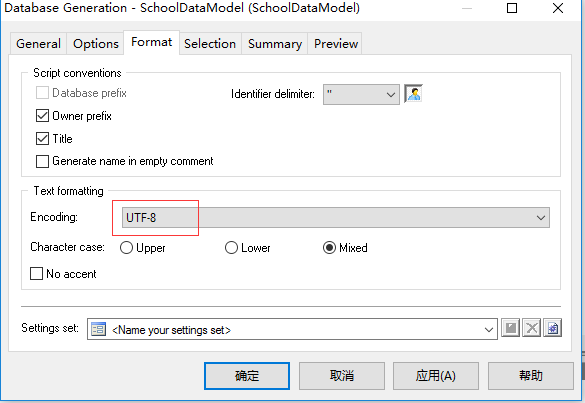
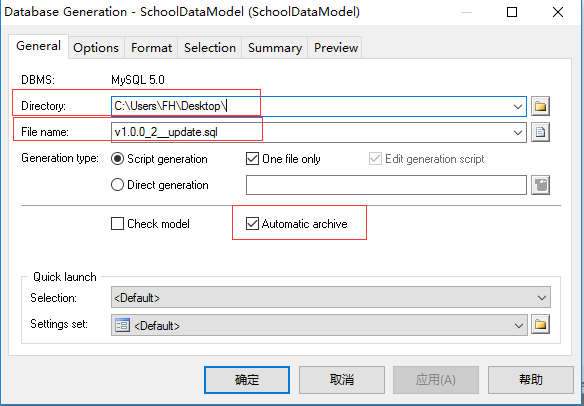
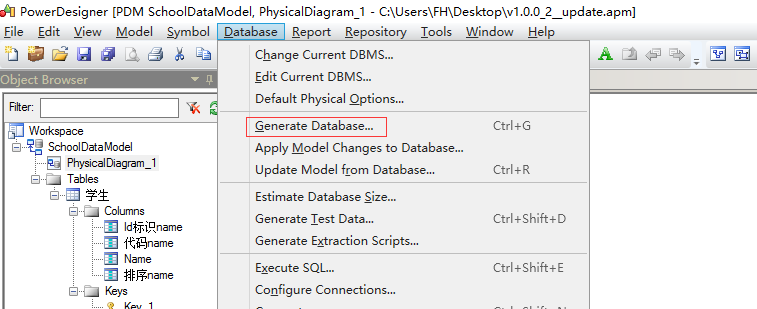


双击上一步添加的Table，添加列。



## 导出脚本

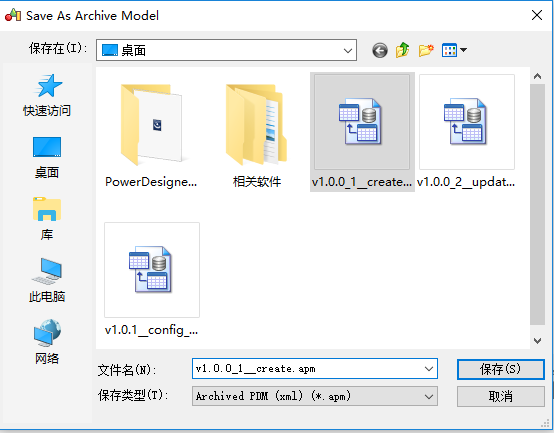
完成数据库设计之后，点击[Database]下的[Generate Database]生成数据库脚本,在弹出的窗口页面选中One file only和Authomatic archive,在Format里的Encoding选项选择编码UTF-8



1. **File name** ：升级脚本名称按“版本定义” 规范取名（首次创建的版本号必须大于1.0.0\_0，因为flywa创建的基线默认为1.0.0\_0，执行脚本时，flywa会重新扫描对应路径中特定的脚本文件，并且与元数据表(flyway\_version)进行校验，如果它们的版本号低于或等于元数据表的版本,它们将被忽略）；
2. **Directory** ：脚本存放目录，默认为项目entity模块下的/src/mian/resources/db/migration(如果不是的话，需要在entity模块下的build.gradle里面flyway下的locations配置路径)
3. **Automatic archive** ：自动归档当前的数据模型

## 归档数据模型

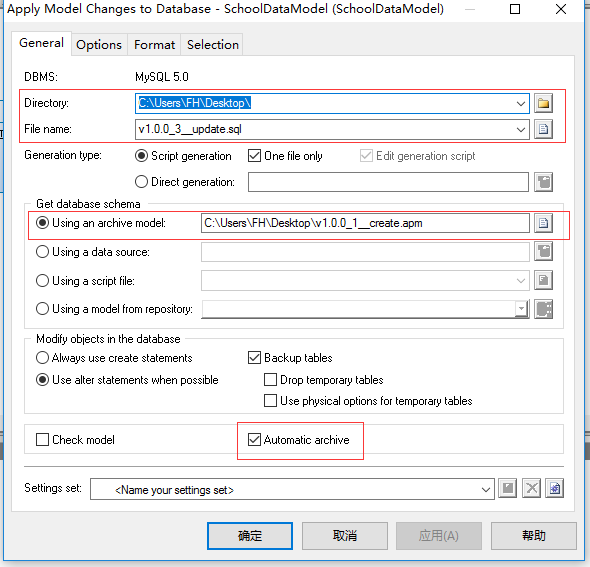
导出脚本完成后，会弹出保存数据模型的按钮，保存目录与名称与导出的脚本名称一致。



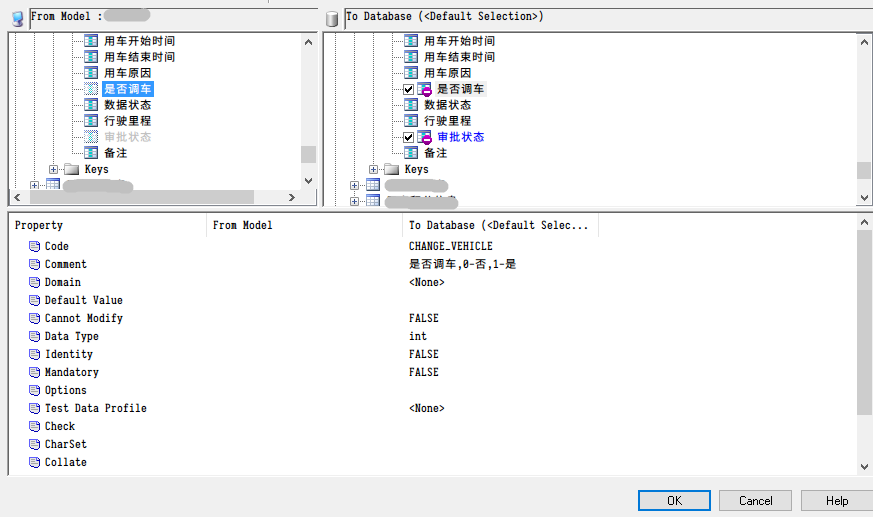
## 更新数据库并更新脚本

首次生成数据库脚本之后，修改数据库结构后需要使用如下步骤修改并生成更新脚本:

1. 获取Git上最新模型文件，利用PowerDesigne直接打开最新（版本号最大）的模型文件进行修改
2. 修改完成后点击[Database]下的[Apply Model Changes to Database]生成数据库更新脚本
3. 选择Using an archive model、Backup tables和Automatic archive



1. Using an archive model：已归档的最新模型文件（修改时打开的模型文件），主要作用是与修改后的数据模型进行对比，增量的生成数据库更新脚本
2. Backup tables：如果不选中的话会重新创建表，并且丢失原有数据
3. 其他同“3.3 导出脚本”



确定之后会出现对比差异页面，确认没有问题后点击OK生成最终的升级脚本，生成升级脚本之后参考“3.4 归档数据模型”对数据模型进行归档

**注意** ：

1. 用powerdesigner导出脚本的时候，存在WARNING的一定要处理掉，才能执行flywayMigrate
2. 生成的脚本包含rename table语句的，一定要检查当前数据库中对应的表（rename table后面跟的表名）是否存在外键，若存在，一定要在脚本最前面添加删除对应外键的语句（alter table 表名 drop FOREIGN KEY 外键名; ）

## Flyway

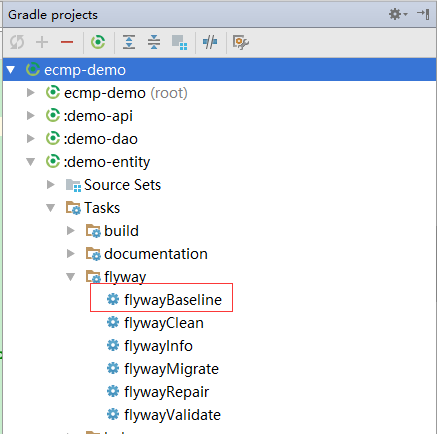
### 通过flyway执行脚本

修改gradle.properties中flyway相关配置。

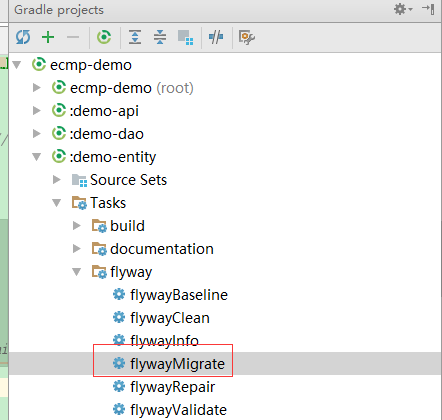
*# ####################### flyway ##############################***flyway\_driver**= **com.mysql.jdbc.Driver***#数据库连接地址与名称（ecmp\_demo）***flyway\_url**= **jdbc:mysql://地址:端口/ecmp\_demo?characterEncoding=utf8***#数据库用户、密码***flyway\_user**= **xx  
flyway\_password**= xxx

首次通过脚本更新数据库时，需要执行flywayBaseline创建基线。执行完成后，在数据库中会生成元数据表(flyway version)并插入一条基线记录（version为1，其实被flyway默认为v10 0 0版本了）

**注：基线的设置只需要第一次初始化数据库时执行一次就好了，以后都不需要再执行**



创建完基线之后，执行flywayMigrate将脚本文件中的更改更新到数据库服务器。



**注意：更新完成后，需要将生成的脚本和模型文件添加到Git管理，并马上push到服务器上，以便其它开发人员更改数据库时使用。**

### Flyway命令介绍

1. flywayMigrate：应用所有的迁移到最新版本，将数据库更新到最新，同时会在数据库中表flyway\_version增加相应升级的版本信息
2. flywayClean：清除掉所有的对象（数据、表以及存放版本信息的元数据表：flyway\_version ）， **禁止执行**
3. flywayInfo：打印所有的迁移的信息以及状态
4. flywayValidate：迁移之前进行验证
5. flywayBaseline：创建基线，初始化元数据表：flyway\_version，并插入一条原始verion=1
6. flywayRepair：移除所有失败的迁移（升级）、重置校验

## 数据库表的设计

### 数据库编码

数据库及数据库表的编码统一设置为utf-8编码。

### 数据库设计命名规则

1. 所有表名、字段名统一小写，多个单词或前后缀之间使用下划线连接。
2. 字段长度应该精确控制，即为最小的必需长度。例如：UUID为36位，如果某个字段是用UUID生成的，那么它的长度设计不应该超过36位。
3. 不允许写触发器和存储过程。
4. 所有表、视图、索引、函数、约束、主外键必须指定名称，规则如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 数据库对象 | 规则 | 最大长度 | 示例 |
| 表(实体名：UserAccount) | 英文名称，多个单词以‘\_ ’分隔 | 30 | user\_account |
| 字段名 | 英文名称，多个单词以‘\_ ’分隔 | 30 | code, name,short\_name |
| 主键 | pk\_ 表名 | 30 | pk\_user\_account |
| 外键 | fk\_ 表名\_ 字段名 | 30 | fk\_user\_account\_code |
| 普通索引 | idx\_ 表名\_ 字段名 | 30 | Idx\_user\_account\_code |
| 唯一索引 | uk\_ 表名\_ 字段名 | 30 | uk\_user\_account\_code |

**注：由于数据库对象之间的命名规则与最大长度限制可能造成命名存在冲突。如存在，可以通过缩减表名或自编序号。**

1. 禁止用SQL 保留字来命名表、视图、字段、索引。
2. 所有的外键、约束、索引、函数不允许重复。
3. 字段的命名规则，英文名称建议使用以下字母后缀:

|  |  |
| --- | --- |
| **类型** | **规则(字母后缀)** |
| 编码类 | code |
| 名称类 | name |
| 简称 | shortname |
| 标志类 | flag |
| 级次类 | lev |
| 金额类 | mny |
| 系统标识类 | id |
| 自定义项类 | def |
| 单价类 | price |
| 税类 | tax |
| 数量类 | qty |
| 比率类 | rate |
| 日期类 | date |
| 枚举 | enum |

### 索引

1. 为了保证索引具有提高效率的作用，估计记录数不大于50的表一律不允许建索引。
2. 只用于DISTINCT或GROUP BY子句引用的列，不用建立索引。
3. 对于数据值变化不大的列，比如只有yes和no，不要建索引，否则效率降低。
4. 建立索引index（part1,part2,part3）,相当于建立了index(part1),index(part1,part2)和index（part1,part2,part3）三个索引。
5. 索引应该建在小字段上，对于大的文本字段甚至超长字段，不要建索引。
6. 频繁进行数据操作的表，不要建立太多的索引。
7. 删除无用的索引，避免对执行计划造成负面影响。

# Entity

## 概述

对对象实体的封装,orm实体用于数据库映射,vo实体用于表示层和逻辑层的数据交换即展示对象

## 配置文件build.gradle

### 编码

//java编译的时候缺省状态下会因为中文字符而失败

[compileJava, compileTestJava, javadoc]\*.options\*.encoding = 'UTF-8'

### 插件

Flyway固定版本：

plugins {  
 id **"org.flywaydb.flyway"** version **"4.1.2"** }

### 添加依赖

**dependencies** {

**compile**(

**"com.ecmp:core-entity:**$ecmp\_version**"**,

**"com.oracle:ojdbc:**$oracle\_version**"**

)

}

### flyway配置

flyway {

driver = "$flyway\_driver"

url = "$flyway\_url"

user = "$flyway\_user"

password = "$flyway\_password"

//flyway发布版本记录的表名称

table = 'flyway\_version'

// 默认脚本存放路径：/src/mian/resources/db/migration

//locations = ["filesystem:${projectDir}/db-migrate/db/migration"]

// 脚本文件前缀，默认V

//sqlMigrationPrefix = 'v'

// 禁止flywayClean，在生产环境中非常重要

cleanDisabled = true

}

### 上传Maven仓库

#### 打成jar包上传

jar {  
 manifest {  
 attributes(**"Manifest-Version"**: **"**$project.version**"**)  
 attributes(**"Extension-Name"**: **"**$project.name**"**)  
 attributes(**"Build-Time"**: **new** Date().format(**"yyyy-MM-dd HH:mm:ss"**))  
 }  
}

*//打包源代码*task sourcesJar(**type**: Jar, **dependsOn**: classes) {  
 classifier = **'sources'** from sourceSets.**main**.allSource  
}  
  
artifacts {  
 archives **sourcesJar**}

#### 若希望安装到.m2本地仓库

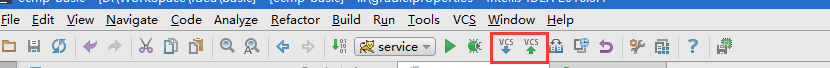
install {  
 repositories.**mavenInstaller** {  
 pom.version = **"**$project.version**"** pom.artifactId = **"**$project.name**"** pom.groupId = **"**$project.group**"** }  
 }

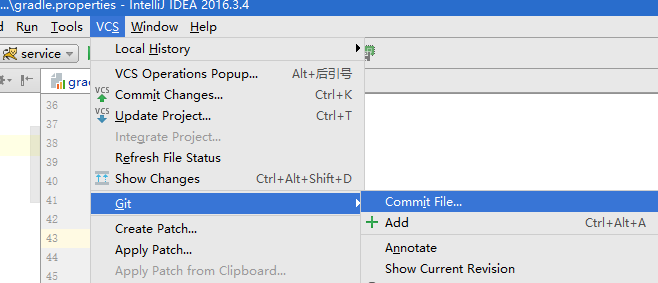
#### 上传到nexus

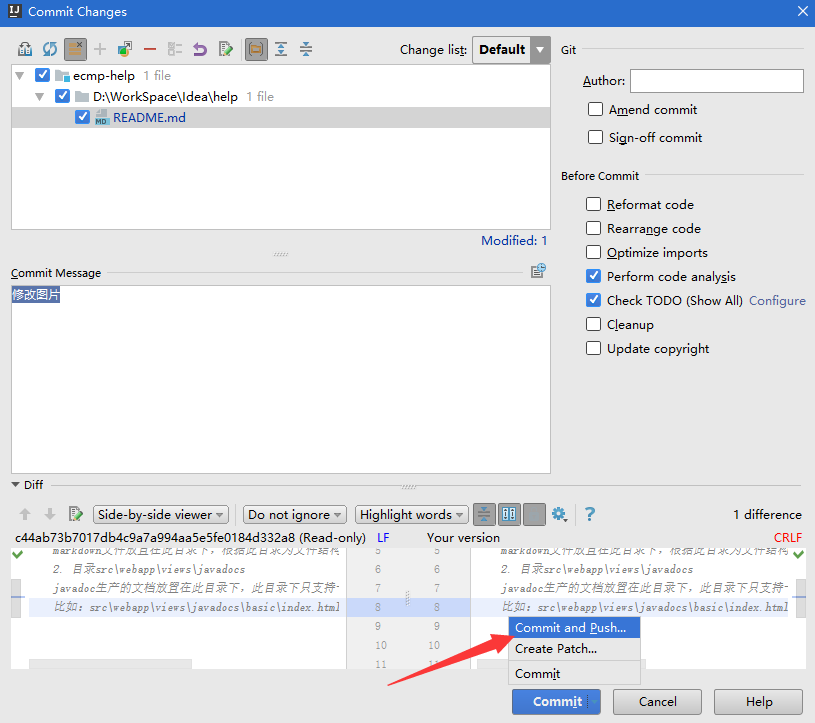
uploadArchives {  
 repositories {  
 mavenDeployer {  
 repository(**url**: MAVEN\_REPO\_URL) {  
 authentication(**userName**: NEXUS\_USERNAME, **password**:NEXUS\_PASSWORD)  
 }  
 pom.version = **"**$project.version**"** pom.artifactId = **"**$project.name**"** pom.groupId = **"**$project.group**"** }  
 }  
 **return 'Success'** }

### 上传到Git仓库操作

上传时点击上传箭头按钮或者点击VCS-->Git-->Commit File,再确认自己需要上传的文件，最后选中Commit按钮里的Commit and Push







## 命名规则

实体类命名均与数据库表名称一致，采用驼峰命名，例如数据库名为user\_account，实体名为UserAccount

## 类注解

@Access(AccessType.FIELD)

@Entity

@Table(name = "user\_account")

@DynamicInsert

@DynamicUpdate

@Cacheable

public class UserAccount extends BaseAuditableEntity implements ITenant,ICodeUnique,IRank,IFrozen,IDataAuthEntity {}

### @Access(AccessType.FIELD)

1. AccessType.FIELD：按字段访问，Jpa字段注解添加在属性上
2. AccessType.PROPERTY 按属性访问，注解添加在get方法上

### @DynamicInsert和@DynamicUpdate

在插入和修改数据的时候,语句中只包括要插入和修改的字段

### @Entity

@Entity表示这是一个实体Bean

### @Table(name = “user\_account”)

1. 这里的@Table命名空间是import javax.persistence.Table，是jpa的注解，注意和hibernate的@Table注解分别开来
2. @Table配合@Entity使用，如果要在dao的实现类里自己写JPQL语句的话，必须要@Table标签

### @Cacheable

该注解是jpa的注解，表示该实体加入二级缓存（sessionFactory），需要进行缓存的才使用该注解

## 继承基类和实现对应的业务实体类型

### 继承BaseEntity

这个类里面包含了id以及id的get()和set()方法，id是由UUID产生的字符串类型，继承这个类就不用再在实体里面写id和生成get()和set()方法

### 继承BaseAuditableEntity

这个类本身继承了BaseEntity并实现了IAuditable接口，这个类主要是在生成id的基础上增加了创建人id、账号、名称、创建时间和修改人id、账号、名称、修改时间

### 继承AbstractEntity

该类为JPA实体抽象类，该类判断了对象是否新建状态，BaseEntity继承了该类

### 实现对应的业务实体类型接口

1. 业务实体类型的基本接口
   1. RelationEntity：分配关系业务实体接口
   2. TreeEntity：树形结构业务实体接口
2. 业务实体类型的特征接口
   1. IFrozen：冻结属性
   2. IRank：排序属性
   3. ITenant：带租户类型
   4. ICodeUnique：代码唯一
   5. IDataAuthEntity：数据权限管理的业务实体
   6. IDataAuthTreeEntity：权限管理的树形业务实体

各实体如果包含上述类型的话，要实现对应的接口，如果包含多种类型的话，所包含的类型的接口方法均要实现

## 属性上的注解

### @Column(name = "code", length = 20, unique = true, nullable = false)

@Column(name = "code", length = 20, unique = true, nullable = false)

private String code;

1. name代表该属性在数据库中的名称
2. length是数据库中限制的最大长度
3. unique代表唯一性
4. nullable代表是否允许为空，值为false表示不能为空

### 临时字段

@Transient

@Transient

private String temp;

表示是临时字段，不会持久化

### 外键关联

@ManyToOne(cascade = CascadeType.REFRESH)

@JoinColumn(name = "feature\_group\_id")

private FeatureGroup featureGroup;

//@OneToOne(cascade = {CascadeType.REFRESH})

//@JoinColumn(name = "id", referencedColumnName = "id", insertable = false, updatable = false)

//private User user;

1. 表示是多对一关联和一对一关联
2. CascadeType.REFRESH表示级联刷新，即会重新查询数据库里的最新数据
3. 外键为xxx
4. 一对一默认是关联注解，id是主键，就不需要写referencedColumnName = "id"，如果关联其他字段，则需要单独写出来
5. **禁止使用一对多关联或者多对多关联**

### 枚举

@Enumerated(EnumType.ORDINAL)

@Enumerated(EnumType.ORDINAL)

@Column(name = "type\_enum", length = 20)

private TypeEnum typeEnum;

表示是枚举，EnumType.ORDINA枚举位置，EnumType.STRING枚举名称，建议使用枚举位置(下标)

### 大字段

@Lob，@Basic(fetch = FetchType.LAZY)

@Lob

@Basic(fetch = FetchType.LAZY)

@Column(name = "big\_text")

private String bigText;

1. @Lob表示是大字段
2. @Basic表示加载方式，fetch = FetchType.LAZY表示是懒加载

### 时间戳

@Temporal(TemporalType.TIMESTAMP)

@Temporal(TemporalType.TIMESTAMP)

@Column(name = "add\_test\_time", nullable = false)

private Date addTestTime;

1. TemporalType.DATE是日期
2. TemporalType.TIME是时间
3. TemporalType.TIMESTAMP是日期和时间

# Dao

## 概述

Dao层用于数据访问

## 配置文件build.gradle

### 添加插件

apply **plugin**: **'java'**apply **plugin**: **'org.springframework.boot'**apply **plugin**: **'io.spring.dependency-management'**

### 编码

//java编译的时候缺省状态下会因为中文字符而失败

[compileJava, compileTestJava, javadoc]\*.options\*.encoding = 'UTF-8'

### 添加依赖

依赖core的dao

**dependencies** {

*//括号里写‘:项目名-entity’*

**compile** **project**(':demoapp-entity')

compile(  
 **"org.springframework.boot:spring-boot-starter"**,  
 **"com.ecmp:jpa-ecmp-spring-boot-starter:**$ecmp\_version**"**)

}

### 打包

jar {  
 manifest {  
 attributes(**"Manifest-Version"**: **"**$project.version**"**)  
 attributes(**"Extension-Name"**: **"**$project.name**"**)  
 attributes(**"Build-Time"**: **new** Date().format(**"yyyy-MM-dd HH:mm:ss"**))  
 }  
 enabled = **true**}

## 命名规则

接口命名按照“实体名”+“Dao”的方式

## 接口注解

@Repository用于标注数据访问组件

@Repository

public interface DemoGeneralDao extends BaseEntityDao<DemoGeneral> {}

## 接口继承

BaseTreeDao：树形结构继承此接口

BaseRelationDao: 分配关系继承此接口

BaseEntityDao：其他类型均继承此接口

如果存在dao数据访问扩展的情况，还要再继承对应的扩展接口

@Repository

public interface DemoDao extends BaseEntityDao<Demo>, DemoExtDao

## 接口扩展

如果自己还想要扩展一些功能，可以写一个对应的扩展接口，命名为“实体名”+“ExtDao”的方式，即DemoExtDao，扩展的接口不需要继承其他的接口，也不需要加接口注解

public interface DemoExtDao {

/\*\*

\* 保存主数据

\*

\* @param entity 主数据实体

\* @param isNew 是否是创建

\* @return 保存结果

\*/

Demo save(Demo entity, boolean isNew);

}

## 接口实现

1. 如果有扩展的接口，则需要写实现类。实现类需要继承对应的基类的实现类，比如DemoDao继承了BaseEntityDao，则实现类需要继承BaseEntityDaoImpl，同时还需要实现扩展的接口DemoExtDao
2. 实现类不需要加类注解

public class DemoDaoImpl extends BaseEntityDaoImpl<Demo> implements DemoExtDao {

public DemoDaoImpl(EntityManager entityManager) {

super(Demo.class, entityManager);

}

@Override

public Demo save(Demo entity, boolean isNew) {

//具体代码

}

}

# API

## 概述

对实体的API服务定义

## 配置文件build.gradle

### 编码

//java编译的时候缺省状态下会因为中文字符而失败

[compileJava, compileTestJava, javadoc]\*.options\*.encoding = 'UTF-8'

### 添加依赖

dependencies {

//依赖本项目的实体

compile project(':demo-entity')

//依赖core的api

compile (

"com.ecmp:core-api:$ecmp\_version"

)

}

### 上传Maven仓库

#### 打成jar包上传

jar {  
 manifest {  
 attributes(**"Manifest-Version"**: **"**$project.version**"**)  
 attributes(**"Extension-Name"**: **"**$project.name**"**)  
 attributes(**"Build-Time"**: **new** Date().format(**"yyyy-MM-dd HH:mm:ss"**))  
 }  
}

task sourcesJar(**type**: Jar, **dependsOn**: classes) {  
 classifier = **'sources'** from **sourceSets**.**main**.allSource  
}  
  
artifacts {  
 archives **sourcesJar**}

#### 若希望安装到Maven本地仓库

install {  
 repositories.**mavenInstaller** {  
 pom.version = **"**$project.version**"** pom.artifactId = **"**$project.name**"** pom.groupId = **"**$project.group**"** }  
}

#### 上传到nexus

uploadArchives {  
 repositories {  
 mavenDeployer {  
 repository(**url**: MAVEN\_REPO\_URL) {  
 authentication(**userName**: NEXUS\_USERNAME, **password**: NEXUS\_PASSWORD)  
 }  
 pom.version = **"**$project.version**"** pom.artifactId = **"**$project.name**"** pom.groupId = **"**$project.group**"** }  
 }  
 **return 'Success'**}

## 接口命名

命名按照“I”+“实体名”+“Service”的方式

## 包注解

api包下面每层要有一个package-info，对该层api包进行说明，指定api的所属应用模块：

//表明所属模块，这里演示的模块为DEMO

@AppModule("DEMO")

package com.ecmp.demo.api;

import com.ecmp.annotation.AppModule;

## 接口注解

@Path("demo")

@Consumes(MediaType.APPLICATION\_JSON)

@Produces(MediaType.APPLICATION\_JSON)

@Api(value = "IDemoService演示实体的服务接口")

public interface IDemoService

extends IBaseEntityService<Demo>,IFindAllService<Demo>,IDataAuthEntityService<Demo> {

}

### 接口的相对路径

@Path(“demo”)表示接口的相对路径是：/demo

### 接收的参数类型

@Consumes(MediaType.APPLICATION\_JSON)表示接收的参数类型是Json类型

### 返回数据类型

@Produces(MediaType.APPLICATION\_JSON)表示返回数据类型是Json类型

### Api接口的介绍

@Api(value = “这是一个demo”)

## 继承接口

1. IBaseEntityService：实体继承BaseEntity的继承此接口

2. IBaseRelationService：存在分配关系的继承此接口

3. IBaseTreeService：实体继承TreeEntity的继承此接口

4. IDataAuthEntityService：非树形结构的权限管理的继承此接口

5. IDataAuthTreeEntityService：树形结构的权限管理的继承此接口

6. IFindAllService：需要获取所有业务实体或者获取所有未冻结的业务实体的继承此接口

7. IFindByPageService：分页查询业务实体的继承此接口

## 方法注解

### 操作

/\*\*

\* 查询所有业务实体

\* @return 业务实体清单

\*/

@GET

@Path("findAll")

@ApiOperation(value = "查询所有数据", notes = "查询所有数据")

List<Demo> findAll();

/\*\*

\* 通过Id获取一个业务实体

\* @param id 业务实体Id

\* @return 业务实体

\*/

@GET

@Path("findOne")

@ApiOperation(value = "获取一个数据角色组",notes = "通过Id获取一个数据角色组")

T findOne(@QueryParam("id") String id);

/\*\*

\* 保存业务实体

\* @param entity 业务实体

\* @return 操作结果

\*/

@POST

@Path("save")

@ApiOperation(value = "保存业务实体",notes = "保存一个业务实体")

OperateResultWithData<T> save(T entity);

/\*\*

\* 删除业务实体

\* @param id 业务实体Id

\* @return 操作结果

\*/

@DELETE

@Path("delete")

@ApiOperation(value = "删除业务实体",notes = "删除一个业务实体")

OperateResult delete(String id);

1. @GET代表是获取或者查询，若有参数则需要加@QueryParam注释
2. @POST代表是保存或者更新，若有多个参数则需要加@QueryParam注释
3. @DELETE代表是删除，若有多个参数则需要加@QueryParam注释

### 方法路径

@path(“findAll”)表示查询所有的数据的方法路径（自定义）

### 方法的介绍

@ApiOperation(value = "查询所有主数据", notes = "查询所有主数据") 表示对方法的介绍

### 保存的返回类型

OperateResultWithData<Demo> save(Demo demo);

返回该类型包含了信息、状态和数据

### 删除的返回类型

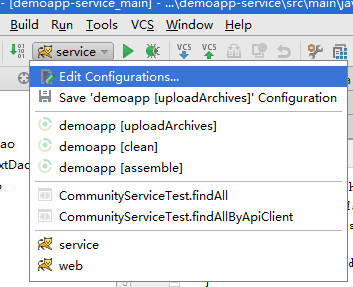
OperateResult delete(String id);

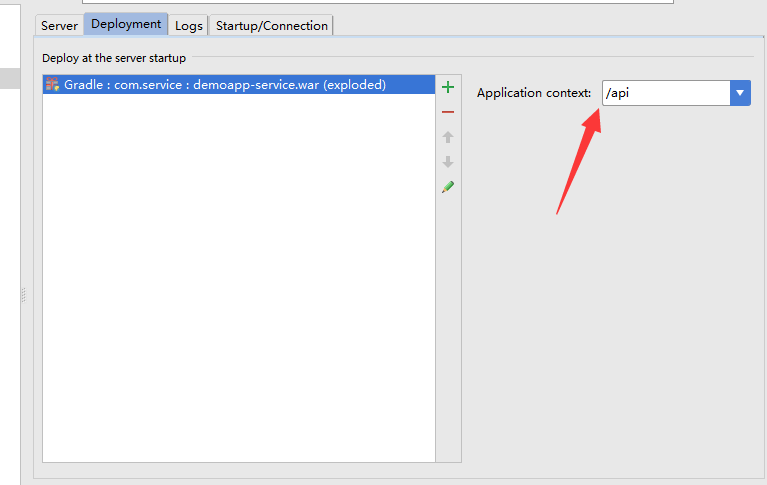
返回类型包含信息和状态

## 通过swagger查看API文档

### 本地swagger地址

在Tomcat的Deployment-->Application context配置 /api，若没有配置/api 则去掉下面地址中的两个/api





<http://127.0.0.1:8080/api/api-docs?/url=/api/swagger.json>

# Service

## 概述

该层是业务逻辑层同时也是Api的实现层

## 配置文件build.gradle

### 编码

//java编译的时候缺省状态下会因为中文字符而失败

[**compileJava**, **compileTestJava**, **javadoc**]\*.options\*.encoding = **'UTF-8'**

### 插件

apply **plugin**: **'war'**apply **plugin**: **'org.springframework.boot'**apply **plugin**: **'io.spring.dependency-management'**

### war包名称和路径

war{

enabled = **true**  
 *//包名称* archiveName **"brm-**${baseName}**.**${extension}**"** *//包存放路径* destinationDir = file(**"**${rootProject.projectDir}**/build/war"**)  
}

### 添加依赖

dependencies {

compile project(':demo-dao')

compile project(':demo-api')

compile(

"com.ecmp:core-service:$ecmp\_version",

**"org.springframework.boot:spring-boot-starter-web"**,  
**"com.ecmp:cxf-service-ecmp-spring-boot-starter:**$ecmp\_version**"**,  
**"com.ecmp:edm-ecmp-spring-boot-starter:**$ecmp\_version**"**

)

// 测试依赖

testCompile(

**"org.springframework.boot:spring-boot-starter-test"**

)

}

## 应用配置

在resources下的application.properties里面写入自己的应用配置：



## 添加主类

注意：请将主类的这两个文件放在com.ecmp这个包路径下。

### Jar包方式启动

@SpringBootApplication

//类名自定义  
**public class** ApiApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(ApiApplication.**class**, args);  
 }  
}

### War包方式启动

**//类名自定义**

**public class** ApiServlet **extends** SpringBootServletInitializer {@Override  
 **protected** SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  
 **return** builder.sources(ApiApplication.**class**);  
 }  
}

## 命名

接口命名按照“实体名”+“Service”的方式

## 类注解

@Service用于标注业务层组件:

@Service

public class DemoService extends BaseEntityService<Demo> implements IDemoService{}

## 继承类和实现接口

### 继承类

BaseTreeService：实体类型是树形结构的继承此类

BaseRelationService：实体存在分配关系的继承此类

BaseEntityService：实体类型非树形结构的继承此类

### 实现接口

实现对应的接口

public **class** **DemoService** **extends** **BaseEntityService<Demo>** **implements** **IDemoService**

## 编写代码

### 依赖注入

1. 使用@Autowired标签注入此Service要操作的实体的Dao接口
2. 使用@Autowired标签注入此Service用到的其他的Service或dao

### 构造函数

构造函数返回该service对应的dao

@Override

protected BaseEntityDao<Demo> getDao() {

return demoDao;

}

### 事务控制

通过在方法上添加注解@Transactional(readOnly = true)来进行控制，一般查询类的方法都应该添加此注解

@Transactional(readOnly = true)

@Override

public List<DemoGeneralTree> getDemoGeneralTree() {

}

### 给号器代码

/\*\*

\* 通过给号器进行给号后再保存

\*

\* @param entity 要保存的实体

\* @return 操作结果

\*/

@Override

public OperateResultWithData<Demo> save(Demo entity) {

if (StringUtils.isBlank(entity.getCode())) {

entity.setCode(NumberGenerator.getNumber(Demo.class));

}

return super.save(entity);

}

### 业务缓存

@Transactional(readOnly = true)

@Cacheable(value = "test",key = "’test\_’+#userId")

public List<Demo> test(String userId){}

1. **这里的@Cacheable是spring的注解**
2. value指明了缓存将被存到什么地方
3. 任何存储在缓存中的数据为了高速访问都需要一个key，Spring默认使用被@Cacheable注解的方法的签名来作为key，当然你可以重写key，自定义key可以使用SpEL表达式，"#userId"是一个SpEL表达式，他将使用test()方法中的userId参数作为key的一部分

### 清除缓存

1. 清除当前实体二级缓存：dao.evict()
2. 清除实体指定Id的二级缓存: dao.evict(ID id)
3. 清除所有实体二级缓存:dao.evictAll()

## 资源

在messages.properties里面写入自己需要的提示信息，例如：

00001 = 欢迎进入ECMP平台！

## 单元测试

### 测试基类

为了简化测试代码，可以写一个测试基类UnitTestBase：

@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public abstract class** UnitTestBase **extends** AbstractJUnit4SpringContextTests {

@Before

public void initTest() throws Exception {

ContextUtil.mockUser();

System.out.println(ContextUtil.getTenantCode() + "-" + ContextUtil.getUserAccount());

}

}

### 测试类

#### 继承测试基类

**public class** DemoServiceTest **extends** UnitTestBase {}

#### 本地Service测试

##### 注入本地Service

@Autowired

private DemoService demoService;

##### 测试

@Test

public void testSave() {

Demo demo = new Demo();

demo.setName("test3" );

demo.setCode("code3" );

OperateResultWithData<Demo> resultWithData = demoService.save(demo);

Assert.assertTrue(resultWithData.successful());

}

#### ApiClient调用本地接口测试

1. ApiClient调用服务器接口测试需要开启Service里的applicationConfig.properties本地api测试地址：

# ####################### 本地测试配置 #######################

# api本地测试地址

API\_BASE\_ADDRESS = http://127.0.0.1:8080/api/

# 模拟的租户代码

MOCK\_TENANT\_CODE=10011

# 模拟的用户账号

MOCK\_ACCOUNT=admin

1. 测试代码：

@Test

public void testFindAll() {

IDemoService proxy = ApiClient.createProxy(IDemoService.class);

OperateResultWithData<Demo> resultWithData = proxy.findAll()；

Assert.assertTrue(resultWithData.successful());

}

**测试通过后签入代码并发布**

#### ApiClient调用服务器上已经部署的接口测试

ApiClient调用服务器接口测试需要注释掉Service里的applicationConfig.properties本地api测试地址

# ####################### 本地测试配置 #######################

# api本地测试地址

#API\_BASE\_ADDRESS = http://127.0.0.1:8080/api/

# 模拟的租户代码

MOCK\_TENANT\_CODE=10011

# 模拟的用户账号

MOCK\_ACCOUNT=admin

@Test

public void testFindAll() {

IDemoService proxy = ApiClient.createProxy(IDemoService.class);

OperateResultWithData<Demo> resultWithData = proxy.findAll()；

Assert.assertTrue(resultWithData.successful());

}

# Web

## 概述

后台与页面的交互

## 配置文件build.gradle

### 编码

*//java编译的时候缺省状态下会因为中文字符而失败*

[compileJava, compileTestJava, javadoc]\*.options\*.encoding = 'UTF-8'

### 插件

apply **plugin**: **'war'**apply **plugin**: **'org.springframework.boot'**apply **plugin**: **'io.spring.dependency-management'**

### War包名称和路径

war{

enabled = **true**

*//包名称*

archiveName "${baseName}.${extension}"

*//包存放路径*

destinationDir = file("${rootProject.projectDir}/build/war")

}

### 添加依赖

**dependencies** {

*//括号里是‘：项目名-api’*

**compile** **project**(':demoapp-api')

**compile**(

**"org.springframework.boot:spring-boot-starter-web"**,  
**"com.ecmp:web-ecmp-spring-boot-starter:**$ecmp\_version**"**

)

}

### Jar包

jar {  
 manifest {  
 attributes(**"Manifest-Version"**: **"**$project.version**"**)  
 attributes(**"Extension-Name"**: **"CH-**$project.name**"**)  
 attributes(**"Build-Time"**: **new** Date().format(**"yyyy-MM-dd HH:mm:ss"**))  
 }  
 enabled = **true**}  
  
*//跳过上传*uploadArchives {  
 enabled = **false**}

## 应用配置

在resources下的application.properties里面写入应用配置



## 多语言

在resources/ lang/messages.properties里面写入多语言，例如：

**00001** = **登陆成功！**

## 控制器

### 添加主类

注意：请将主类的这两个文件放在com.ecmp这个包路径下。

### Jar包方式启动

@SpringBootApplication

//类名自定义  
**public class** WebApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(WebApplication.**class**, args);  
 }  
}

### War包方式启动

**//类名自定义**

**public class** WebServlet **extends** SpringBootServletInitializer {@Override  
 **protected** SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  
 **return** builder.sources(WebApplication.**class**);  
 }  
}

### 命名

命名遵循“实体”+“Controller”的方式，例如实体是Demo，控制器名称则是DemoController

### 类注解

@Controller用于标注控制层组件

@RequestMapping(“/demo”)表示该Controller的相对地址

### 继承类

BaseEntityController类包含了对以下几个相对请求地址的处理：

1. **show：页面展示**
2. **listAll：查询所有数据**
3. **listByPage：分页查询**
4. **save：保存**
5. **delete：删除**

### 方法注解

1. @RequestMapping(value = “listAll”)

表示该方法的请求相对路径

1. @ResponseBody

表示返回Json数据

### 编码实现

1. 都通过ApiClient创建接口代理的方式调用具体的方法，例如使用IDemoService的findAll方法

/\*\*

\* 查询所有数据

\*

\* @return 数据清单

\*/

@RequestMapping(value = "listAll")

@ResponseBody

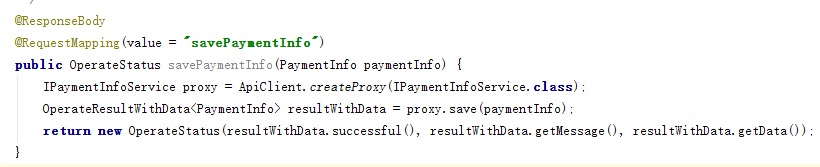
public List<Demo> listAll() {

IDemoService demoService = ApiClient.createProxy(IDemoService.class);

return demoService.findAll();

}

1. 若返回非List<T>类型的Json数据，则需要在返回前用OperateStatus进行构造



## Html页面

### 文件存放路径

#### HTML文件

在resources里面创建文件夹views，所有html文件都放在这个文件里面，可以自行对html文件进行分组。

#### JS、CSS等其他文件

在resources里面创建文件夹META-INF，在META-INF内创建resources文件夹，所有js，css文件都放在这个文件里面，可以自行对js,css文件进行分组。

### 页面命名

1. html、css、js文件命名均按照“实体名”+View的方式命名，例如：实体是General，html文件则是GeneralView.html
2. 多语言统一为lang\_zh\_CN.js

### 页面格式

<!DOCTYPE **html**>  
<**html xmlns:th="http://www.thymeleaf.org"**>  
<**head**>  
 <**th:block th:include="base/base :: base\_head"**/>  
 <**meta http-equiv="Content-Type" content="text/html; charset=UTF-8"**>  
 <**title**>演示</**title**>  
 <**link rel="stylesheet" type="text/css"  
 data-th-href="@{/static/styles/css/GeneralView.css}"**>  
 <**script type="text/javascript"  
 data-th-src="@{/static/scripts/main/GeneralView.js}"**></**script**>  
 <**script type="text/javascript"  
 data-th-src="@{/static/lang/lang\_zh\_CN.js}"**></**script**>  
 <**script type="text/javascript"**>  
 **var generalView**;  
 EUI.onReady(**function** () {  
 **generalView**= **new** EUI.General**View**({  
 **renderTo**: **"content"** });  
 });  
 </**script**>  
</**head**>  
<**body**>  
<**div id="content"**></**div**>  
</**body**>  
</**html**>

说明：页面格式基本固定，只需更改以下几点：

1. 导入的css、js和多语言文件名称及路径，这里需要注意先导入css，再导入js
2. js里的变量如上面的代码所示：
3. **var generalView**;推荐改为对应页面的变量名
4. **generalView**= **new** EUI.General**View,** new的对象应该为当前页面js
5. **renderTo**代表new的对象放入id为content的容器里面

### Js文件

采用EUI框架，EUI在线示例<http://decmp.changhong.com/EUI-api/views/home/welcome.html>

### Css文件

自定义

### 多语言lang\_zh\_CN.js

1、**var *defaultLangs*** = {  
 **addText**: **"新增"**,  
 **hintText**: **"提示"**  
 };

这是通用的默认多语言，可以在里面新增通用的多语言

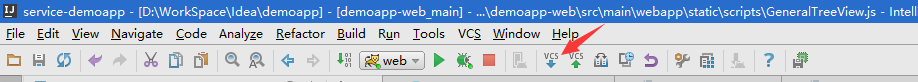
2、**if** (EUI.**BusinessActivityCategoryView**) {  
 EUI.**BusinessActivityCategoryView**.**prototype**.**lang** = EUI.applyIf({  
 **codeText**: **"代码"**,  
 **nameText**: **"名称"** }, ***defaultLangs***);  
 };

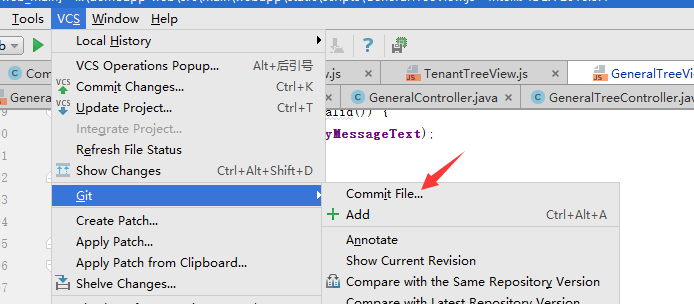
这是单独页面General**View**的多语言，同时引用通用多语言***defaultLangs***

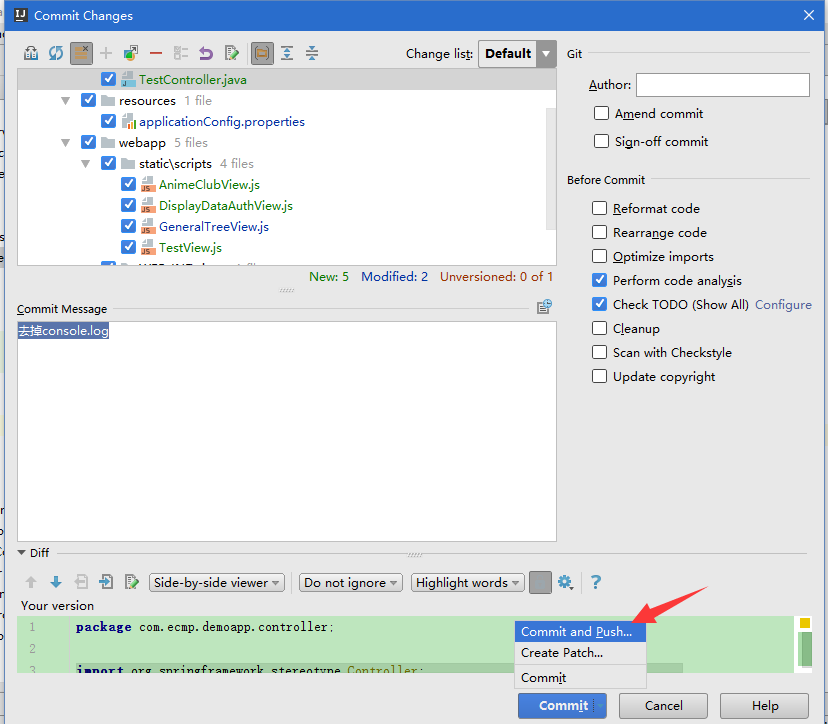
# 签入和发布

## 代码签入

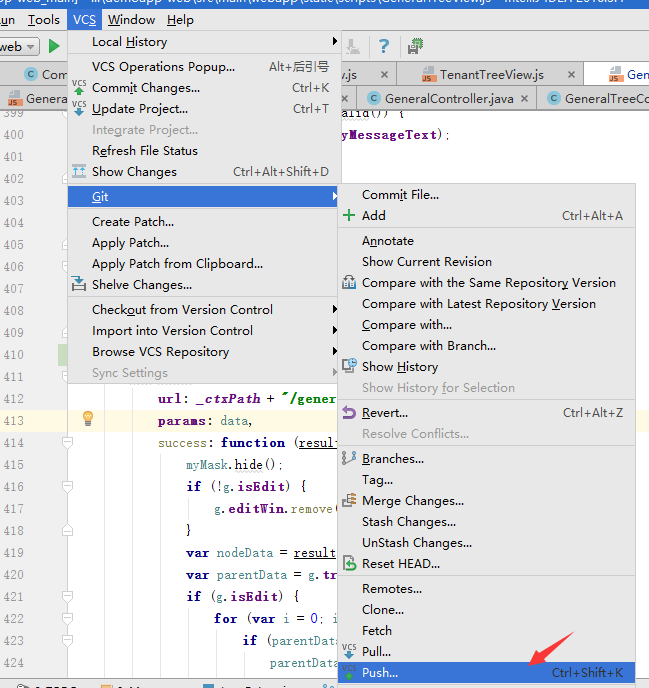
1. 上传时点击上传箭头按钮或者点击VCS-->Git-->Commit File，然后可以依次点开列表里的文件来确认修改内容，确认无误后选中Commit按钮里的Commit and Push上传到服务器的Git里， 选择Commit只是添加到本地Git







1. 若不小心选择了Commit，可以通过如下顺序上传到服务器： VCS-->Git-->Push



## 代码发布

平台代码由Jenkins统一管理，配置管理员在Jenkins中配置了相应的Git地址，可以设置间隔一定时间扫描Git，获取有变动的最新代码，然后进行发布。发布完成后需要在平台配置功能项和菜单



# 注册功能项和菜单

## 概述

平台菜单最多有三层，根节点就是我们的项目，下面的一层节点就是我们的功能项组，最后一层就是具体的功能项了。因此，我们在创建功能项组及功能项的时候一定要规划好我们整个的菜单结构。

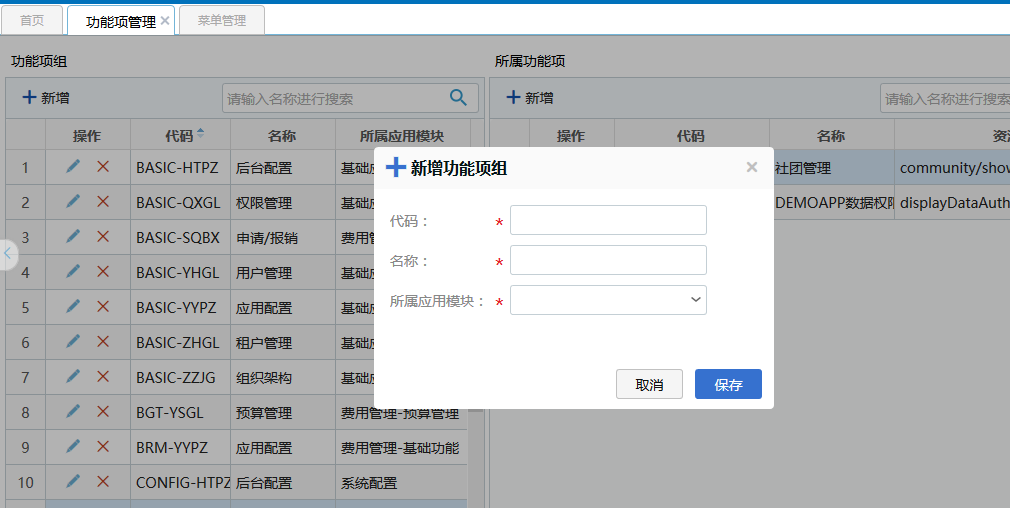
项目（IDEA里的单个项目）首次配置功能项和菜单的时候要新建功能项组和根节点，当我们的项目已经配置过功能项组和根节点的时候，我们只需要在对应的功能项组和菜单里面配置就可以了，或者根据需要新增功能项组，但根节点不再新增

## 配置功能项

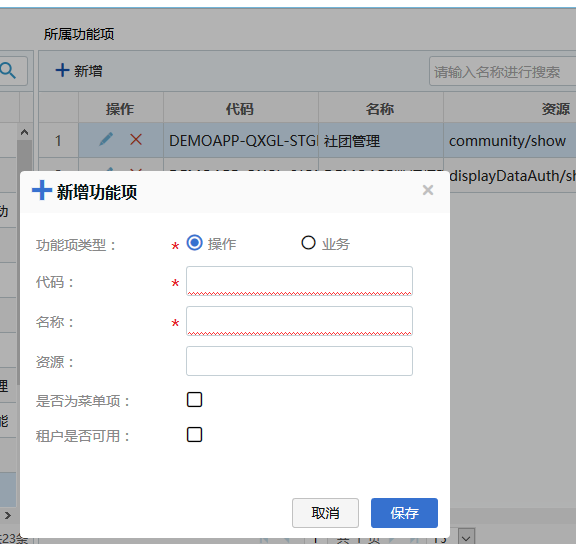
### 创建功能项组

代码格式：项目-功能项组名称的首字母简写

例：DEMOAPP项目要创建名称为演示的功能项组，功能项组的代码为：DEMOAPP-YS



### 创建对应的功能项



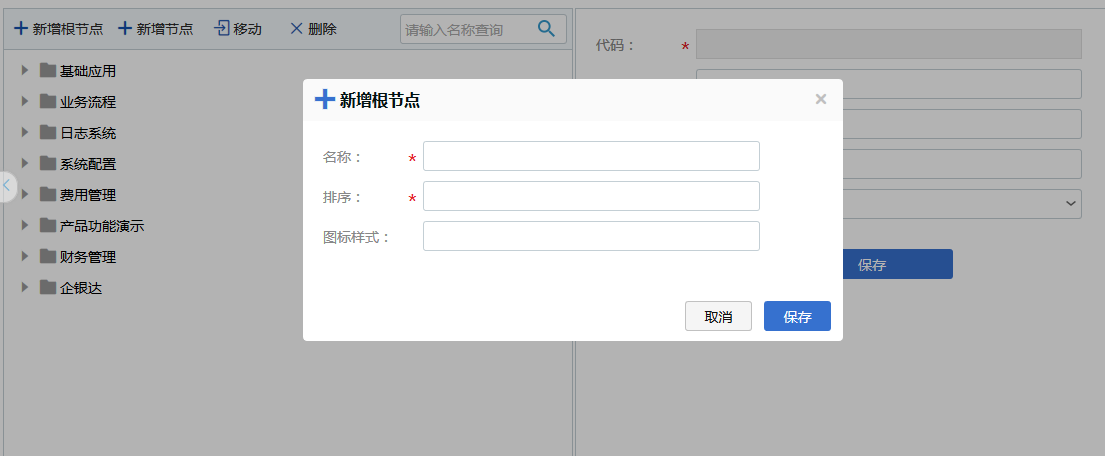
这里如果是要用来显示页面的话，就选择操作，配置资源（页面的相对路径），选择是菜单项、租户可用

代码格式：功能项组代码加上“-”功能项名称首字母简写

例：**演示**功能项组下的功能项**实体演示**，代码为DEMOAPP-YS-STYS

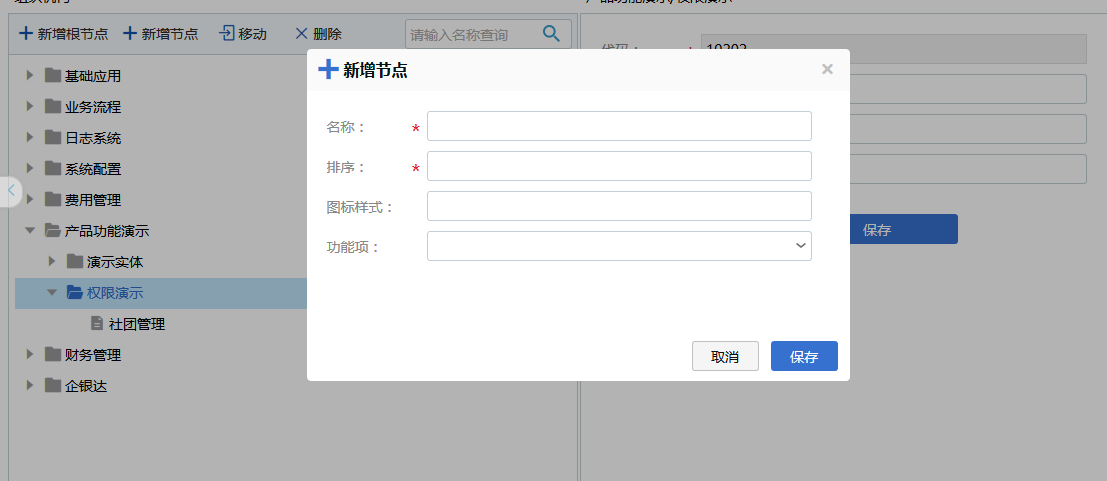
## 配置菜单

### 创建根节点



根节点的名称就是项目的中文名称

### 创建子节点



菜单里的第二层对应的是功能项组，第三层就是具体的功能项了

# 异常日志

## 服务端的异常日志

添加依赖

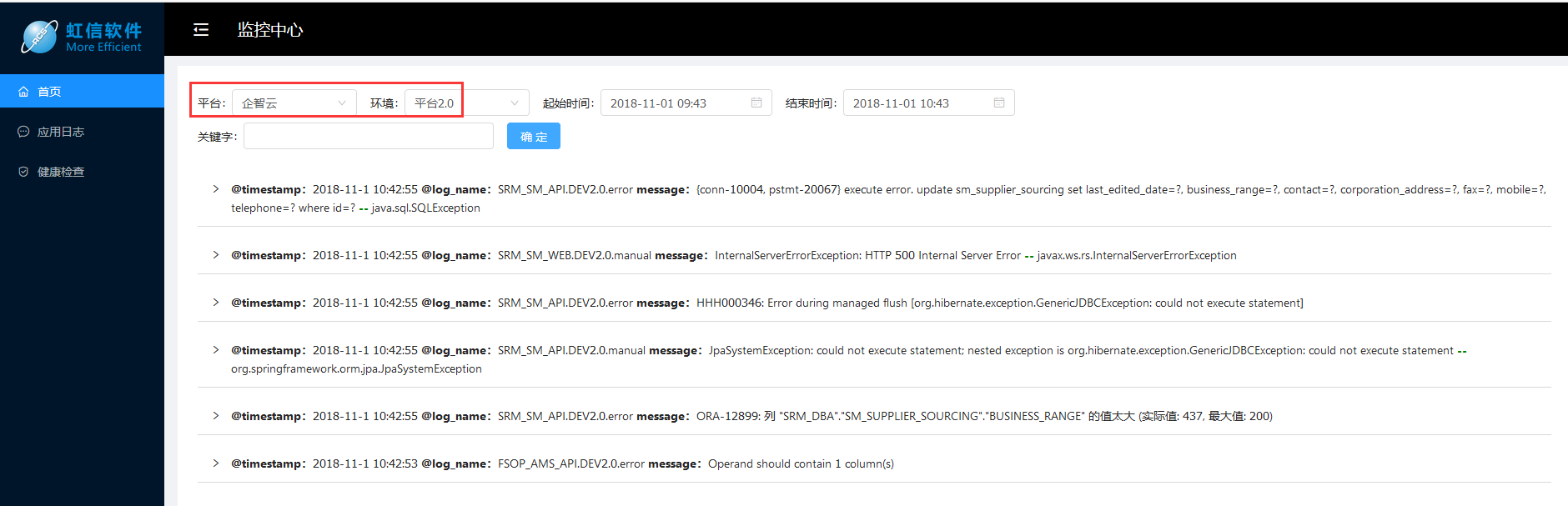
compile(,  
 **"com.ecmp:log-ecmp-spring-boot-starter:**$ecmp\_version**"**)

## 需要特殊记录的异常日志

使用工具类 LogUtil 的 LogUtil.error(String message, Throwable throwable) 方法

## 查看异常日志

登录<http://decmp.changhong.com/mc-web>，选择相应的平台与环境，即可查看异常日志 。



# 附件

|  |  |
| --- | --- |
| MySQL | <https://www.mysql.com/downloads/> |
| JDK | <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html> |
| Tomcat | <http://tomcat.apache.org/> |
| IDEA | <http://www.jetbrains.com/idea/> |
| Gradle | [https://gradle.org](http://link.zhihu.com/?target=https%3A//gradle.org/) |
| Maven | <https://maven.apache.org/> |
| Git | <https://www.git-scm.com/downloads> |
| PowerDesigner | <http://powerdesigner.de/> |