

百战电子商城后台管理平台<Flask 项目开发>

1 项目介绍

随着信息化的发展，电商也随着互联网的发展日益壮大。如今在网上也有很多相应的产品如：Ebay 易趣、淘宝网、腾讯拍拍网、亚马逊、当当网、新蛋中国、京东商城、VANCL、乐淘网、鹏程万里贸易商城、红孩子、走秀网、唯品会、时尚起义、马萨玛索、麦包包、衣服网、戴维尼、钻石小鸟、乐友、麦网、多购、SHOPEX、BONO、EC Spyder、搜房家天下等



1.1 为什么有这么多公司做电商呢？

1．电子商务将传统的商务流程电子化、数字化，一方面以电子流代替了实物流，可以大量减少人力、物力，降低了成本;另一方面突破了时间和空间的限制，使得交易活动可以在任何时间、任何地点进行，从而大大提高了效率。

2．电子商务所具有的开放性和全球性的特点，为企业创造了更多的贸易机会。

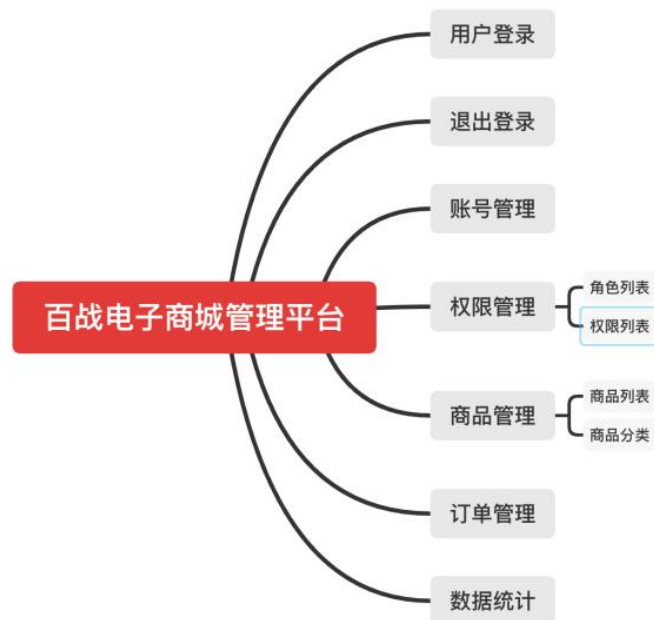
3. 电子商务使企业可以以相近的成本进入全球电子化市场,使得中小企业有可能拥有和大企业一样的信息资源,提高了中小企业的竞争能力。

4. 电子商务重新定义了传统的流通模式,减少了中间环节,使得生产者和消费者的直接交易成为可能,从而在一定程度上改变了整个社会经济运行的方式。

5. 互动性:通过互联网,商家之间可以直接交流,谈判,签合同,消费者也可以把自己的反馈建议反映到企业或商家的网站,而企业或者商家则要根据消费者的反馈及时调查产品种类及服务品质,做到良性互动。

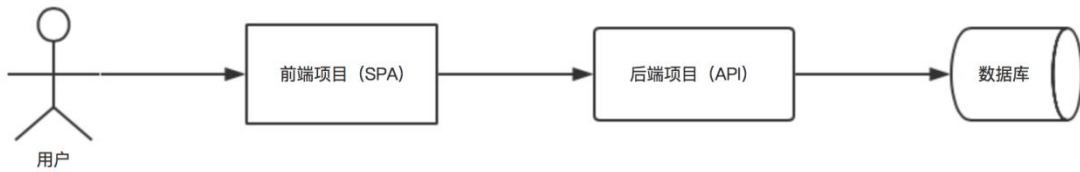
2 项目需求

百战电子商城后台管理平台包含账号管理、商品管理、订单管理、数据统计等业务功能

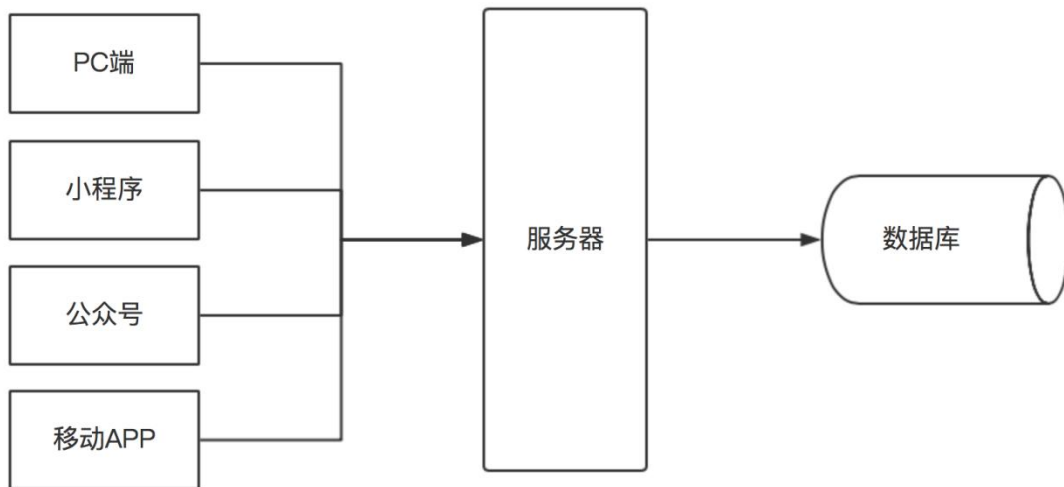


3 项目技术

百战电子商城后台管理平台采用前后端分离的开发模式



为什么选择这样的方式呢？



3.1 前端的技术

- Vue 前端框架
- Element-UI UI 框架
- Axios 发送请求
- Echarts 绘制图表

3.2 后端的技术

- Python 主流语言
- Flask Web 框架
- MySQL 存储主要数据

- Redis 存储缓存数据

项目实施

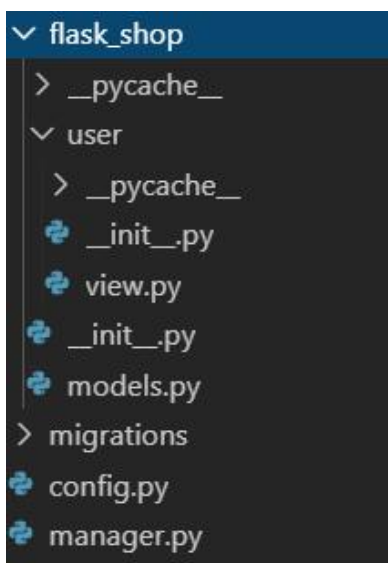
1 后端项目的搭建

1.1 配置 python 虚拟环境

```
mkvirtualenv shop_env
```

1.2 配置 Flask 项目结构

- app 项目目录
- app/__init__.py 模块初始化
- app/model 模型类
- app/blue_print 蓝图
- app/static 静态文件夹
- app/utils 工具文件夹
- log 日志目录
- manager.py 管理项目
- config.py 配置项目文件



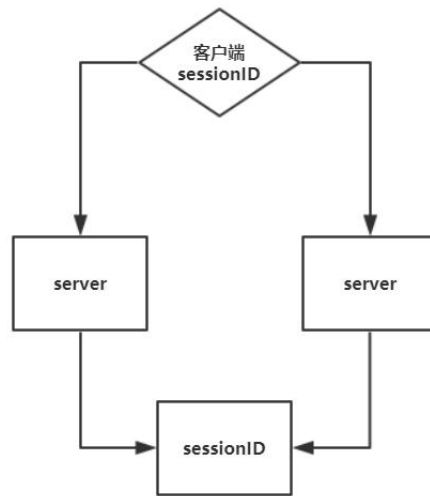
2 后端登录的实现

代码如下：

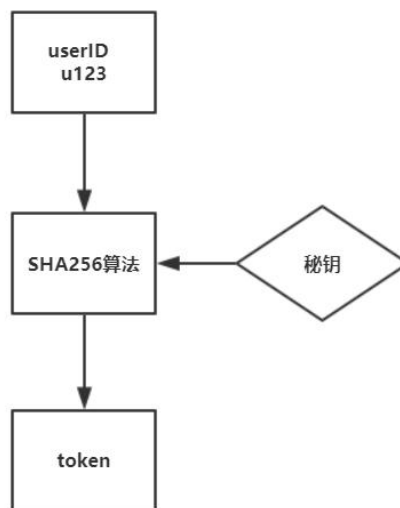
```
@user.route('/login',methods=['POST'])
def login():
    name = request.form.get('name')
    password = request.form.get('pwd')
    if not all([name,password]):
        return to_dict_msg(10000)
    if len(name) > 1:
        usr = UM.query.filter_by(name=name).first()
        if usr:
            if usr.check_passwd(password):
                token = tokens.generate_auth_token(usr.id,expiration=100000)
                return to_dict_msg(200,data={'token':token})
    return to_dict_msg(10001)
```

2.1 token 的使用

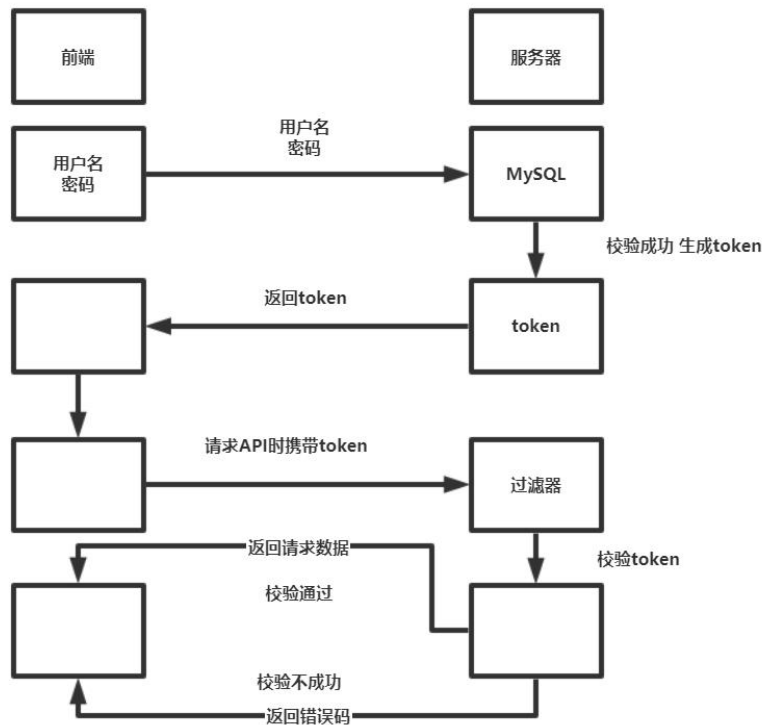
基于 Token 的身份验证是无状态的，我们不将用户信息存在服务器中。这种概念解决了在服务端存储信息时的许多问题。NoSession 意味着你的程序可以根据需要去增减机器，而不用去担心用户是否登录



Session 架构图 (图)



Token 产生方式 (图)



```

from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
from flask import current_app,request
from flask_shop.models import User
from flask_shop.utils import message
import functools

def generate_auth_token(uid,expiration):
    s = Serializer(current_app.config['SECRET_KEY'],expires_in = expiration)
    return s.dumps({'id':uid}).decode('utf-8')
    
```

```
def verify_auth_token(token):

    s = Serializer(current_app.config['SECRET_KEY'])

    try:

        data = s.loads(token)

    except Exception:

        return None

    usr = User.query.get(data['id'])

    return usr
```

2.2 登录装饰器

当有业务访问时，必须登录过才可以访问时，为了避免每次编写代码，我们可以编写一个装饰来验证用户是否登录

```
def login_required(view_func):

    @functools.wraps(view_func)

    def verify_token(*args,**kwargs):

        try:

            #在请求头上拿到 token

            token = request.headers["token"]

        except Exception:

            #没接收的到 token,给前端抛出错误

            return message.to_dict_msg(10002)
```



```
s = Serializer(current_app.config["SECRET_KEY"])

try:
    s.loads(token)

except Exception:
    return message.to_dict_msg(10003)

return view_func(*args,**kwargs)

return verify_token
```

3 前端项目的搭建

3.1 安装 vue 脚手架

- 安装 node.js

- <https://nodejs.org/zh-cn/>

- 安装 nrm

- 作用：提供了一些最常用的 NPM 包镜像地址，能够让我们快速的切换安装包时候的服务器地址；

- 什么是镜像：原来包刚开始是只存在于国外的 NPM 服务器，但是由于网络原因，经常访问不到，这时候，我们可以在国内，创建一个和官网完全一样的 NPM 服务器，只不过，数据都是从人家那里拿过来的，除此之外，使用方式完全一样；

- ◆ 运行 `npm i nrm -g` 全局安装 nrm 包；
- ◆ 使用 `nrm ls` 查看当前所有可用的镜像源地址以及当前所使用的镜像源地址；
- ◆ 使用 `nrm use npm` 或 `nrm use taobao` 切换不同的镜像源地址；

- 安装 vue-cli

- npm install -g @vue/cli

3.2 创建项目

3.2.1 命令创建

- 命令：vue create my-project
- 选择 Manually select features(选择特性以创建项目)
- 勾选特性：可以用空格进行勾选。
- 是否选用历史模式的路由：n
- ESLint 选择：ESLint + Standard config
- 何时进行 ESLint 语法校验：Lint on save babel , postcss
- 等配置文件如何放置：In dedicated config files(单独使用文件进行配置)
- 是否保存为模板：n

3.2.2 UI 创建

- 命令：vue ui
- 在自动打开的创建项目网页中配置项目信息

3.2.3 Vue-cli 创建项目结构介绍

- node_modules:依赖包目录
- public：静态资源目录
- src：源码目录
- src/assets:资源目录
- src/components：组件目录
- Plugins:三方组件配置目录

- src/views:视图组件目录
- src/App.vue:根组件
- src/main.js:入口 js
- src/router/index.js:路由 js
- babel.config.js:babel 配置文件
- .eslintrc.js: 校验配置 js

3.3 配置 vue 路由

3.4 配置 Element-UI 组件

3.5 配置 axios 库

4 前端登录的基础

4.1 ES6 语法介绍

4.1.1 默认导出 与 导入

默认导出语法 export default 默认导出的成员

默认导入语法 import 接收名称 from '模块标识符'

```
// mode1.js  
  
//定义私有成员 a b c  
  
let a = 10  
  
let b = 20  
  
let c = 30  
  
function show(){}
```

// 将本模块中的私有成员暴露出去，供其它模块使用（注意：每个模块中，只允许使用唯一的一次 export default，否则会报错！）

```
export default {  
  a,  
  b,  
  show  
}
```

// 导入模块成员

```
import m1 from './model1.js'
```

```
console.log(m1)
```

// 打印输出的结果为：

```
// { a: 10, b: 20, show: [Function: show] }
```

4.1.2 按需导出 与 按需导入

按需导出语法 export let s1 = 10

按需导入语法 import { s1 } from '模块标识符'

```
// 当前文件模块为 m1.js
```

```
// 向外按需导出变量 s1
```

```
export let s1 = 'aaa'
```

```
// 向外按需导出变量 s2

export let s2 = 'ccc'

// 向外按需导出方法 say

export function say = function() {}
```

```
// 导入模块成员

import { s1, s2 as ss2, say } from './m1.js'

console.log(s1) // 打印输出 aaa

console.log(ss2) // 打印输出 ccc

console.log(say) // 打印输出 [Function: say]
```

4.1.3 直接导入并执行模块代码

有时候，我们只想单纯执行某个模块中的代码，并不需要得到模块中向外暴露的成员，此时，可以直接导入并执行模块代码

```
// 当前文件模块为 m2.js

// 在当前模块中执行一个 for 循环操作

for(let i = 0; i < 3; i++) {

  console.log(i)

}
```

```
// 直接导入并执行模块代码

import './m2.js'
```

4.2 箭头函数的使用

箭头函数可以说是 es6 的一大亮点，使用箭头函数，可以简化编码过程，是代码更加的简洁

ES6 允许使用“箭头”（=>）定义函数

```
let f = a => a;

let f = function (a) { return a; };
```

```
let sum = (num1, num2) => num1 + num2;

let sum = function(num1, num2) { return num1 + num2; };
```

5 前端登录的实现

5.1 登录基本功能实现

因为是个登录功能，所以我们可在此创建一个独有的组件 Login.vue

```
<template>

  <div class="login_container">

    <div class="login_box">

      <div class="log">

      </div>

      <el-form ref="userref" :model="user" :rules="userRules" label-width="0px"
class="form-style">

        <el-form-item prop="name">

          <el-input v-model="user.name" prefix-icon="el-icon-user" placeholder="
```

```

用户名"></el-input>

</el-form-item>

<el-form-item prop="pwd">

  <el-input v-model="user.pwd" prefix-icon="el-icon-lock" placeholder="密
码"></el-input>

</el-form-item>

<el-form-item class="btns">

  <el-button type="primary" @click="login">登录</el-button>

  <el-button @click="resetForm">重置</el-button>

</el-form-item>

</el-form>

</div>

</div>

</template>

<script>

export default {

  data () {

    return {

      user: {

        name: "",

        pwd: ""

      },


```

```

userRules: {
  name: [
    { required: true, message: '请输入姓名', trigger: 'blur' },
    { min: 2, max: 5, message: '长度在 2 到 5 个字符', trigger: 'blur' }
  ],
  pwd: [
    { required: true, message: '请输入姓名', trigger: 'blur' },
    { min: 2, max: 5, message: '长度在 3 到 5 个字符', trigger: 'blur' }
  ]
}
},
methods: {
  resetForm () {
    console.log(this)
    this.$refs.userref.resetFields()
  },
  login () {
    this.$refs.userref.validate(async valid => {
      if (!valid) return
      const { data: res } = await this.$http.post('/user/login',
this.$qs.stringify(this.user))
      console.log(res)
    })
  }
}
}

```



```

    if (res.status !== 200) return this.$msg.error('登录失败')

    this.$msg.success('登录成功')

    // 1. 将登录成功之后的 token，保存到客户端的 sessionStorage 中

    // 1.1 项目中除了登录之外的其他 API 接口，必须在登录之后才能访问

    // 1.2 token 只应在当前网站打开期间生效，所以将 token 保存在
sessionStorage 中

    window.sessionStorage.setItem('token', res.data.token)

    // 2. 通过程式导航跳转到后台主页，路由地址是 /home

    this.$router.push('/home')

  })
}
}
}
</script>

<style lang="less" scoped>

.login_container {

  background-color:darksalmon;

  height: 100%;

}

.login_box {

  width: 450px;

  height: 300px;

```

```
background-color: #ffffff;  
  
border-radius: 3px;  
  
position: absolute;  
  
left: 50%;  
  
top: 50%;  
  
transform: translate(-50%, -50%);  
  
}
```

```
.log {  
  
height: 80px;  
  
width: 200px;  
  
border: 1px solid #eee;  
  
border-radius: 10%;  
  
padding: 10px;  
  
position: absolute;  
  
left: 50%;  
  
transform: translate(-50%, -50%);  
  
box-shadow: 0 0 10px #ddd;  
  
img {  
  
width: 100%;  
  
height: 100%;  
  
border-radius: 15%;  
  
// background-color: #eee;
```

```

    }
  }
  .form-style {
    position: absolute;
    bottom: 0;
    width: 100%;
    padding: 0 10%;
    box-sizing: border-box;
  }
  .btns {
    display: flex;
    justify-content: flex-end;
  }
</style>

```

创建好组件，要记的将组件挂在到路由 router 上

```

// Index.js
const routes = [
  { path: '/', redirect: '/login' },
  { path: '/login', component: Login }
]

```

5.2 跨域问题

因为我们是前端一套系统，后端一套，并且我们在部署项目可能不是同一台服务器，所以我们在发送请求时，可能会产生跨越问题。因此，我们可以在 vue 时设置代理。

```
module.exports = {
  devServer: {
    proxy: {
      '/': {
        target: 'http://localhost:5000',
        changeOrigin: true, // 允许跨域
        pathRewrite: { // 重写路径
          '^/': '/'
        }
      }
    }
  }
}
```

5.3 发送请求前判断是否登录

在每次发送请求前，我们可以自己处理下看看是否登录，如果没有登录，就跳转登录页面，在 index.js 中编写

```
router.beforeEach((to, from, next) => {
  if (to.path === '/login') return next()
  const tokenStr = window.sessionStorage.getItem('token')
  if (!tokenStr) return next('/login')
  next()
})
```

5.4 发送请求携带 token

当我们登录后，我们每次访问后端时，我们应该携带后端传递给我们的 token，而我们如果每次都去写一遍从 storage 中获取 token 还是比较麻烦的，因此我们可以写一个拦截器，达到每次上访问时，获取 token,并放到 request 中，而我们的拦截器可以将他放到 main.js 中

```

axios.interceptors.request.use(
  config => {
    const tokenStr = window.sessionStorage.getItem('token')
    if (tokenStr) { // 判断是否存在 token，如果存在的话，则每个 http header 都加上 token
      config.headers.token = tokenStr
    }
    return config
  }
)

```

5.5 处理 token 失效

在每次发送请求后，后端会给我们响应，因为 token 有时间限制，所以我们应该在接收到响应后，判断下是否 token 失效,如果失效，我们可统一跳转 login 页面。所以在此我们可以写个拦截器

```

axios.interceptors.response.use(
  response => {
    if (response.data.status === 10016 || response.data.status === 10017) {
      window.sessionStorage.removeItem('token')
    }
  }
)

```

```
router.replace({
    path: '/login'
})

}

return response
}

)
```

6 后端主页的实现

6.1 菜单 Model 的编写

通过项目需求，我们可以分析出菜单 model 的可以设计字段与内容

```
class Menu(db.Model):

    __tablename__ = 't_menu'

    id = db.Column(db.Integer, primary_key=True)

    name = db.Column(db.String(32), unique=True, nullable=False)

    level = db.Column(db.Integer)

    path = db.Column(db.String(32))

    pid = db.Column(db.Integer, db.ForeignKey('t_menu.id'))

    children = db.relationship('Menu')
```

为了方便给前端发送数据，而传送数据为 json，所以我们编写

```
def to_dict(self):  
    return {  
        'id': self.id,  
        'name': self.name,  
        'level': self.level,  
        'path': self.path,  
        'pid': self.pid  
    }  
  
def get_child_list(self):  
    obj_child = self.children  
    data = []  
    for o in obj_child:  
        data.append(o.to_dict)  
    return data
```

6.2 菜单 Model 的接口实现

```
class Menu(Resource):  
    def get(self):  
        type_ = request.args.get('type')  
        menu_list = []  
        if type_ == 'list':
```

```
# 获取数据,并将数据填充到 menu_list

mu = models.Menu.query.filter(models.Menu.level != 0 ).all()

for m in mu:

    menu_list.append(m.to_dict())

else:

    mu = models.Menu.query.filter(models.Menu.level == 1 ).all()

    for m in mu:

        # 获取 1 级菜单转成 json

        first_mu = m.to_dict()

        # 给 2 级菜单创建保存容器

        first_mu['children'] = []

        for sm in m.children:

            # 获取 2 级菜单转成 json

            secd_dict = sm.to_dict()

            # 给 2 级菜单创建保存容器, 并把 3 级菜单数据加进来

            secd_dict['children'] = sm.get_child_list()

            # 把 2 级单级菜单加到 1 级的 children 列表中

            first_mu['children'].append(secd_dict)

        # 把 1 级单级菜单加到根列表中

        menu_list.append(first_mu)

return menu_list
```


7 前端主页的实现

7.1 页面布局

因为主页为独立的页面，所以我们可以创建 Home.vue 组件，在此我们的布局为上左右的布局,代码如下

```
<el-container>

  <el-header>Header</el-header>

  <el-container>

    <el-aside width="200px">Aside</el-aside>

    <el-main>Main</el-main>

  </el-container>

</el-container>
```

7.2 设置头部信息

在项目中一般项目中包含项目的名称，与用户功能菜单，在此我们在头部中加上项目名与退出功能

```
<el-header>

  <div>

    <span>电子后台管理系统</span>

  </div>

  <el-button type="primary" plain @click="logout">退出</el-button>

</el-header>
```

7.3 完成退出功能

因为前端保存登录的信息是 token 实现的，因此我们的退出方案比较简单。只需要清空 token 即可

```
logout () {  
  window.sessionStorage.clear()  
  this.$router.push('/login')  
}
```

7.4 左侧菜单结构编写

在 elementUI 中也为我们提供了左侧的样式，我们直接拿过来即可

```
<el-menu  
  default-active="2"  
  class="el-menu-vertical-demo"  
  @open="handleOpen"  
  @close="handleClose"  
  background-color="#545c64"  
  text-color="#fff"  
  active-text-color="#ffd04b">  
  <el-submenu index="1">  
    <template slot="title">  
      <i class="el-icon-location"></i>  
      <span>导航一</span>
```

```

</template>

<el-menu-item-group>

  <template slot="title">分组一</template>

  <el-menu-item index="1-1">选项 1</el-menu-item>

  <el-menu-item index="1-2">选项 2</el-menu-item>

</el-menu-item-group>

<el-menu-item-group title="分组 2">

  <el-menu-item index="1-3">选项 3</el-menu-item>

</el-menu-item-group>

<el-submenu index="1-4">

  <template slot="title">选项 4</template>

  <el-menu-item index="1-4-1">选项 1</el-menu-item>

</el-submenu>

</el-submenu>

<el-menu-item index="2">

  <i class="el-icon-menu"></i>

  <span slot="title">导航二</span>

</el-menu-item>

<el-menu-item index="3" disabled>

  <i class="el-icon-document"></i>

  <span slot="title">导航三</span>

</el-menu-item>

<el-menu-item index="4">

```

```
<i class="el-icon-setting"> </i>

<span slot="title">导航四</span>

</el-menu-item>

</el-menu>
```

7.5 菜单 Model 的数据获取

当我想要获取菜单数据时，我们可以直接通过 axios 发送请求即可

```
async getMenuList () {

  const { data: res } = await this.$axios.get('/menu')

  if (res.status !== 200) return this.$msg.error(res.msg)

  this.menuList = res.data

}
```

7.6 菜单 Model 的数据填充

填充数据我直接通过 v-for 在来遍历数据，并填充到 template 中即可

```
<el-submenu :index="item.id+''" v-for="item in menuList" :key="item.id">

  <template slot="title">

    <i :class="iconObj[item.id+' ']"> </i>

    <span>{{ item.name }}</span>

  </template>

  <el-menu-item :index="subItem.id+''" v-for="subItem in
item.children" :key="subItem.id">

    <template slot="title">

      <i :class="iconObj[subItem.id+' ']"> </i>

      <span>{{ subItem.name }}</span>
```

```
</template>

</el-menu-item>

</el-submenu>
```

为了根据不同的菜单换不同的图标，我们可以设置一个图标的对象

```
iconObj: {

  '1 ': 'el-icon-user-solid',

  '2 ': 'el-icon-s-tools',

  '3 ': 'el-icon-s-shop',

  '4 ': 'el-icon-s-order',

  '5 ': 'el-icon-s-data',

  '11 ': 'el-icon-user',

  '21 ': 'el-icon-setting',

  '22 ': 'el-icon-setting',

  '31 ': 'el-icon-goods',

  '32 ': 'el-icon-goods',

  '33 ': 'el-icon-goods'

}
```

7.7 主页默认欢迎页面

每次我们登录后，我们不应该让系统中心部分显示为了空的内容。我们应该让他有一个默认的面。如果要这样的操作，我们可以设置一个子路由。

首先我们创建一个欢迎组件：

```
<template>
```

```
<div>Welcome</div>

</template>
```

并将路由挂在到主页路由

```
path: '/home', component: Home, redirect: '/welcome', children: [{ path: '/welcome',
component: Welcome }] }
```

7.7 菜单路由的设置

在点击菜单时，我们应该让系统访问不同的路由，这时我们可以使用 element 里面的 router 功能

```
<el-menu

  default-active="2"

  class="el-menu-vertical-demo"

  @open="handleOpen"

  @close="handleClose"

  background-color="#303133"

  text-color="#fff"

  active-text-color="#409EFF"

  unique-opened="true"

  router="true">
```

8 后端用户管理的实现

8.1 根据 ID 查询用户的实现

为了方便用户快速查看信息，我们还应提供查询接口

请求参数：

参数名	参数说明
id	用户 ID

响应参数：

参数名	参数说明
id	用户 ID
role_id	角色 ID
mobile	手机号
email	邮箱

代码如下：

```
def get(self):
    try:
        id = int(request.args.get('id').strip())
        usr = UM.query.filter_by(id =id ).first()
        if usr:
            return to_dict_msg(200,usr.to_dict(),'获取用户成功！')
        else:
            return to_dict_msg(200,[],'获取用户成功！')
    except Exception as e:
        print(e)
        return to_dict_msg(10000)
```

8.2 用户数据列表的实现

用户可以通过请求获取用户的数据，方便查看，修改数据，而我们采用 restful 方式，所以我们应该采用 get 请求，而数据比较多，所以我们可以采用分页的形式返回。因此，我们可以建立以下几个参数：

参数名	参数说明	备注
query	查询参数	可以为空
pnum	当前页码	不能为空
psize	每页显示条数	不能为空

响应的数据为

参数名	参数说明
totalpage	总记录数
pagenum	当前页码
users	用户数据集合

实现代码如下：

```
def get(self):
    parser = reqparse.RequestParser()
    parser.add_argument('name',type=str)
    parser.add_argument('pnum',type=int)
    parser.add_argument('psize',type=int)
    args = parser.parse_args()
```



```
try:

    name = args.get('name')

    pnum = args.get('pnum') if args.get('pnum') else 1

    psize = args.get('psize') if args.get('psize') else 2

    if name:

        users = UM.query.filter(User.name == name).paginate(pnum,psize)

    else:

        users = UM.query.paginate(pnum,psize)

    data ={

        'pnum':pnum,

        'psize':psize,

        'users':[u.to_dict() for u in users.items]

    }

    return to_dict_msg(200,data,'获取用户列表成功！')

except Exception as e:

    return to_dict_msg(10000)
```

8.3 修改用户的实现

请求参数：

参数名	参数说明	备注
id	用户 id	不能为空
email	邮箱	可以为空

参数名	参数说明	备注
mobile	手机号	可以为空

响应参数：

参数名	参数说明
id	用户 ID
role_id	角色 ID
mobile	手机号
email	邮箱

代码如下：

```
def put(self):
    try:
        id = int(request.form.get('id').strip())
        email = request.form.get('email').strip() if request.form.get('email') else ""
        phone = request.form.get('phone').strip() if request.form.get('phone') else ""

        usr = models.User.query.get(id)

        if usr:
            usr.email = email
            usr.phone = phone
            db.session.commit()

        return to_dict_msg(200,msg='修改数据成功！')
```

```

else:

    return to_dict_msg(10051)

except Exception as e:

    print(e)

    return to_dict_msg(10000)
    
```

8.3 删除用户的实现

请求参数：

参数名	参数说明	备注
id	用户 id	不能为空参数是 url 参数:id

代码如下：

```

def delete(self):

    try:

        id = int(request.form.get('id').strip())

        usr = UM.query.get(id)

        if usr:

            db.session.delete(usr)

            db.session.commit()

        else:

            return to_dict_msg(10051)
    
```

```
except Exception:

    return to_dict_msg(10000)
```

8.4 重置密码的实现

请求参数：

参数名	参数说明	备注
id	用户 id	不能为空参数是 url 参数:id

代码如下：

```
@user.route('/reset',methods=['GET'])
def reset():
    try:
        id = int(request.args.get('id'))
        usr = models.User.query.get(id)
        usr.password = '123'
        db.session.commit()
        return to_dict_msg(200,msg = '重置密码成功！')
    except Exception as e:
        return to_dict_msg(20000)
```

9 前端用户管理的实现

首先我们定义一个用户的组件，并将其挂到组件上。因为是从主页做的跳转，因此推荐将路由挂载到主页的子路由中

```
{
```

```

path: '/home',
component: Home,
redirect: '/welcome',
children: [
  {
    path: '/welcome', component: Welcome
  },
  {
    path: '/user', component: User
  }
]
}

```

9.1 修改菜单记录状态

每次进入主页我们为了友好体验，我们可让浏览器的 sessionStorage 帮助我们记住上次访问的菜单，并让他自己帮助激活菜单。

```

saveNavState (activePath) {
  window.sessionStorage.setItem('activePath', activePath.index)
  this.activePath = activePath.path
}

```

9.2 用户界面 UI 布局

```

<template>
  <div>

```

```

<el-breadcrumb separator-class="el-icon-arrow-right">

  <el-breadcrumb-item :to="{ path: '/home' }">首页</el-breadcrumb-item>

  <el-breadcrumb-item>用户管理</el-breadcrumb-item>

  <el-breadcrumb-item>用户列表</el-breadcrumb-item>

</el-breadcrumb>

<el-card class="box-card">

  <div>

    <el-row :gutter="20">

      <el-col :span="6">

        <el-input placeholder="请输入内容">

          <el-button slot="append" icon="el-icon-search"></el-button>

        </el-input>

      </el-col>

      <el-col :span="2">

        <el-button type="primary" icon="el-icon-search">搜索</el-button>

      </el-col>

    </el-row>

    <template>

      <el-table :data="tableData" stripe border style="width: 100%">

        <el-table-column type="index"></el-table-column>

        <el-table-column prop="name" label="姓名"></el-table-column>

        <el-table-column prop="nick_name" label="昵称"></el-table-column>

        <el-table-column prop="email" label="邮箱"></el-table-column>

```

```

        <el-table-column prop="phone" label="电话"> </el-table-column>

        <el-table-column label="操作">

        </el-table-column>

    </el-table>

</template>

</div>

</el-card>

</div>

</template>

```

9.3 用户列表数据填充

```

async getUserList () {

    const { data: res } = await this.$http.get('/user/user_list')

    this.tableData = res.data.users

}

```

9.4 分页的实现

使用表格来展示用户列表数据，可以使用分页组件完成列表分页展示数据(复制分页组件代码，在 element.js 中导入组件 Pagination)

更改组件中的绑定数据

```

<!-- 分页导航区域

@size-change(pzie 改变时触发)

@current-change(页码发生改变时触发)

:current-page(设置当前页码)

```

:page-size(设置每页的数据条数)

:total(设置总页数) -->

```
<el-pagination      @size-change="handleSizeChange"      @current-
change="handleCurrentChange" :current-page="queryInfo.pnum" :page-sizes="[1, 2,
5, 10]" :page-size="queryInfo.psize" layout="total, sizes, prev, pager, next,
jumper" :total="total">
```

```
</el-pagination>
```

添加两个事件的事件处理函数@size-change , @current-change

```
handleSizeChange(newSize) {
```

```
  //psize 改变时触发，当 psize 发生改变的时候，我们应该
```

```
  //以最新的 psize 来请求数据并展示数据
```

```
  // console.log(newSize)
```

```
  this.queryInfo.psize= newSize;
```

```
  //重新按照 pagesize 发送请求，请求最新的数据
```

```
  this.getUserList();
```

```
},
```

```
handleCurrentChange( current ) {
```

```
  //页码发生改变时触发当 current 发生改变的时候，我们应该
```

```
  //以最新的 current 页码来请求数据并展示数据
```

```
  // console.log(current)
```

```
  this.queryInfo.pnum= current;
```

```
  //重新按照 pagenum 发送请求，请求最新的数据
```

```
  this.getUserList();
```



```
}
```

9.5 用户搜索的实现

添加数据绑定，添加搜索按钮的点击事件(当用户点击搜索按钮的时候，调用 getUserList 方法根据文本框内容重新请求用户列表数据) 当我们在输入框中输入内容并点击搜索之后，会按照搜索关键字搜索，我们希望能够提供一个 X 删除搜索关键字并重新获取所有的用户列表数据，只需要给文本框添加 clearable 属性并添加 clear 事件，在 clear 事件中重新请求数据即可

```
<el-col :span="7">

  <el-input placeholder="请输入内容" v-model="queryInfo.name" clearable
  @clear="getUserList">

    <el-button slot="append" icon="el-icon-search" @click="search"> </el-button>

  </el-input>

</el-col>
```

9.6 用户添加功能的实现

当我们点击添加用户按钮的时候，弹出一个对话框来实现添加用户的功能，首先我们需要复制对话框组件的代码并在 element.js 文件中引入 Dialog 组件

接下来我们要为“添加用户”按钮添加点击事件，在事件中将 addDialogVisible 设置为 true，即显示对话框

更改 Dialog 组件中的内容

```
<!-- 对话框组件 :visible.sync(设置是否显示对话框) width(设置对话框的宽度)
:before-close(在对话框关闭前触发的事件) -->

<el-dialog title="添加用户" :visible.sync="addDialogVisible" width="50%">

  <!-- 对话框主体区域 -->

  <el-form :model="addForm" :rules="addFormRules" ref="addFormRef" label-
```

```
width="70px">

  <el-form-item label="用户名" prop="username">

    <el-input v-model="addForm.username"> </el-input>

  </el-form-item>

  <el-form-item label="密码" prop="password">

    <el-input v-model="addForm.password"> </el-input>

  </el-form-item>

  <el-form-item label="邮箱" prop="email">

    <el-input v-model="addForm.email"> </el-input>

  </el-form-item>

  <el-form-item label="电话" prop="mobile">

    <el-input v-model="addForm.mobile"> </el-input>

  </el-form-item>

</el-form>

<!-- 对话框底部区域 -->

<span slot="footer" class="dialog-footer">

  <el-button @click="addDialogVisible = false">取消</el-button>

  <el-button type="primary" @click="addDialogVisible = false">确定</el-
button>

</span>

</el-dialog>
```

添加数据绑定和校验规则：

```
data() {
```

```
//验证邮箱的规则

var checkEmail = (rule, value, cb) => {

  const regEmail = /^\\w+@\\w+(\\.\\w+)+$/

  if (regEmail.test(value)) {

    return cb()

  }

  //返回一个错误提示

  cb(new Error('请输入合法的邮箱'))

}

//验证手机号码的规则

var checkMobile = (rule, value, cb) => {

  const regMobile = /^1[34578]\\d{9}$/

  if (regMobile.test(value)) {

    return cb()

  }

  //返回一个错误提示

  cb(new Error('请输入合法的手机号码'))

}

return {

  //获取查询用户信息的参数

  queryInfo: {

    // 查询的条件

    query: "",
```

```
// 当前的页数，即页码
pagenum: 1,

// 每页显示的数据条数
pagesize: 2
},

//保存请求回来的用户列表数据
userList: [],

total: 0,

//是否显示添加用户弹出窗
addDialogVisible: false,

// 添加用户的表单数据
addForm: {
  username: "",
  password: "",
  email: "",
  mobile: ""
},

// 添加表单的验证规则对象
addFormRules: {
  username: [
    { required: true, message: '请输入用户名称', trigger: 'blur' },
    {
      min: 3,
```

```

        max: 10,

        message: '用户名在 3~10 个字符之间',

        trigger: 'blur'

    },

    password: [

        { required: true, message: '请输入密码', trigger: 'blur' },

        {

            min: 6,

            max: 15,

            message: '用户名在 6~15 个字符之间',

            trigger: 'blur'

        }

    ],

    email: [

        { required: true, message: '请输入邮箱', trigger: 'blur' },

        { validator: checkEmail, message: '邮箱格式不正确，请重新输入', trigger: 'blur' }

    ],

    mobile: [

        { required: true, message: '请输入手机号码', trigger: 'blur' },

        { validator: checkMobile, message: '手机号码不正确，请重新输入', trigger: 'blur' }

    ]

}

```

```
}  
  
}
```

当关闭对话框时，重置表单 给 el-dialog 添加@close 事件，在事件中添加重置表单的代码

```
methods:{  
  
  ....  
  
  addDialogClosed(){  
  
    //对话框关闭之后，重置表达  
  
    this.$refs.addFormRef.resetFields();  
  
  }  
  
}
```

点击对话框中的确定按钮，发送请求完成添加用户的操作 首先给确定按钮添加点击事件，在点击事件中完成业务逻辑代码

```
methods:{  
  
  ....  
  
  addUser(){  
  
    //点击确定按钮，添加新用户  
  
    //调用 validate 进行表单验证  
  
    this.$refs.addFormRef.validate( async valid => {  
  
      if(!valid) return this.$msg.error("请填写完整用户信息");  
  
      //发送请求完成添加用户的操作  
  
      const {data:res} = await this.$http.post("users",this.addForm)
```

```
//判断如果添加失败，就做提示
if (res.status !== 200)
    return this.$msg.error('添加用户失败')

//添加成功的提示
this.$msg.success("添加用户成功")

//关闭对话框
this.addDialogVisible = false

//重新请求最新的数据
this.getUserList()
})
}
}
```

9.7 用户编辑功能的实现

9.8 用户删除功能的实现

9.9 重置密码功能的实现

10 后端角色管理的实现

通过项目需求，我们可以分析出菜单 model 的可以设计字段与内容

```
class Role(db.Model):
    __tablename__ = 't_role'

    id = db.Column(db.Integer,primary_key=True)
```

```
name = db.Column(db.String(32),unique=True,nullable=True)

desc = db.Column(db.String(32))

users = db.relationship('User', backref = 'role')

menu = db.relationship('Menu',secondary = trm)


def to_dict(self):

    return {

        'id':self.id,

        'name':self.name,

        'desc':self.desc

    }
```

10.1 角色列表

用户可以通过访问主页直接获取角色的所有数据，以及角色的权限,结构如下

```
角色:{

    "id": 1,

    "roleName": "管理员",

    "roleDesc": "技术负责人",

    "children": [

        {

            "id": 1,
```



```

    "name": "商品管理",
    "path": null,
    "children": [
      {
        "id": 1,1
        "name": "商品列表",
        "path": null,
        "children":
      }
    ]
  }

```

10.2 增加角色

请求参数

参数名	参数说明	备注
name	角色名称	不能为空
desc	角色描述	可以为空

10.3 删除角色

请求参数

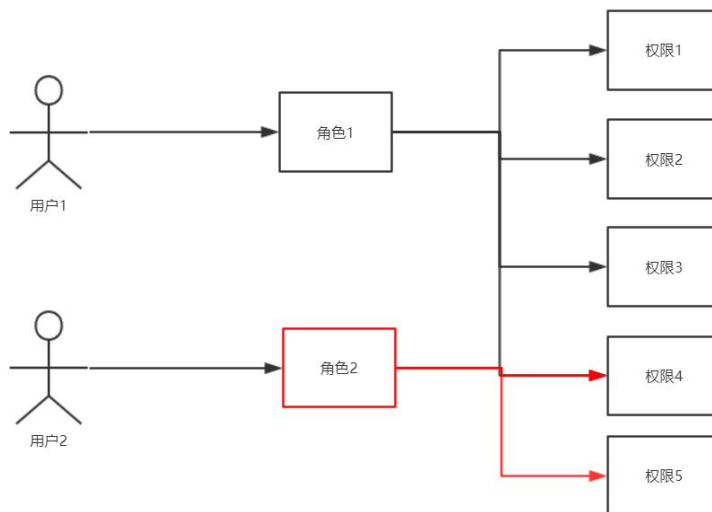
参数名	参数说明	备注
id	角色 ID	不能为空

10.4 修改角色

请求参数

参数名	参数说明	备注
id	角色 ID	不能为空
name	角色名称	不能为空
desc	角色描述	可以为空

10.5 用户修改角色



犹豫我们可以修改用户信息,所以可以直接在用户时,给予增加一个角色的角色即可。

首先在 User 的 模型类增加一个 rid, 建立用户与角色的关系

```
rid = db.Column(db.Integer,db.ForeignKey('t_role.id'))
```

在修改和增加用户时,直接增加 rid 字段即可

```
<el-form-item label="角色">

  <el-select v-model="addForm.role_name" placeholder="请选择角色">

    <el-option :label="r.name" :value="r.id" v-for="r in roles" :key="r.id"></el-
option>

  </el-select>

</el-form-item>
```

10.5 修改角色权限

```
@role.route('/set_menu/<int:rid>/', methods = ['POST'])
@author_required
def set_menu(rid):
    try:
        role = models.Role.query.get(rid)
        mids = request.form.get('mids')
        if role:
            role.menu = []
            for ml in mids.split(','):
                if ml:
                    menu = models.Menu.query.get(int(ml))
                    if menu:
                        role.menu.append(menu)
```

```

        db.session.commit()

        return to_dict_msg(200)

        return to_dict_msg(10051)
    except Exception as e:
        print(e)

        return to_dict_msg(20000)

```

11 前端角色管理的实现

Template 中：

```

<el-form-item label="角色">
    <el-select v-model="editForm.role_name" placeholder="请选择角色">
        <el-option :label="r.name" :value="r.id" v-for="r in roles" :key="r.id"></el-
option>
    </el-select>
</el-form-item>

```

Script

```

// 获取角色列表
async getRole () {
    const { data: res } = await this.$http.get('/role')
    if (res.status !== 200) this.$message.error(res.msg)
}

```

```
this.roles = res.data

}
```

12 前端权限管理的实现

Tmeplate:

```
<el-dialog title="分配权限" :visible.sync="dialogVisible" width="30%" :before-
close="handleClose">

  <el-tree

    show-checkbox

    :data="menuList"

    :props="menuProp"

    node-key="id"

    default-expand-all

    :default-checked-keys="keyList"

    ref="treeRef"

  > </el-tree>

  <span slot="footer" class="dialog-footer">

    <el-button @click="handleClose()">取消</el-button>

    <el-button type="primary" @click="editRM()">确定</el-button>

  </span>

</el-dialog>
```

Script

```

async showMenuDialog(row) {

  this.dialogVisible = true

  this.roleId = row.id

  const { data: resp } = await this.$http.get('/menu')

  if (resp.status !== 200) return this.$message.error(resp.msg)

  this.menuList = resp.data

  this.getKeys(row.menu)

},

getKeys(menu) {

  menu.forEach(element => {

    element.children.forEach(e => {

      this.keyList.push(e.id)

    })

  })

},

handleClose() {

  this.keyList = []

  this.dialogVisible = false

},

async editRM(rid) {

```

```
const allMid = [
  ...this.$refs.treeRef.getHalfCheckedKeys(),
  ...this.$refs.treeRef.getCheckedKeys()
]

const allMidStr = allMid.join(',')

const { data: resp } = await this.$http.post(
  `/set_menu/${this.roleId}/`, this.$qs.stringify({ mids: allMidStr })
)

if (resp.status !== 200) return this.$message.error(resp.msg)

this.$message.success(resp.msg)

this.getRoleList()

this.handleClose()
}
```

13 后端商品分类管理的实现

Model 的设计

```
class Category(db.Model):
    __tablename__ = 't_category'

    id = db.Column(db.Integer, primary_key=True)

    name = db.Column(db.String(32), nullable=True)

    pid = db.Column(db.Integer, db.ForeignKey('t_category.id'))
```

```
level = db.Column(db.Integer)

children = db.relationship('Category')

def to_dict(self):
    return {
        'id': self.id,
        'name': self.name,
        'level': self.level,
        'pid': self.pid
    }
```

13.1 商品分类数据列表

请求方法：get

请求参数

参数名	参数说明	备注
level	[1,2,3]	值：1，2，3 分别表示显示一层二层三层分类列表 【可选参数】如果不传递，则默认获取所有级别的分类
pnum	当前页码值	【可选参数】如果不传递，则默认获取所有分类
psize	每页显示多少条数据	【可选参数】如果不传递，则默认获取所有分类

响应参数

参数名	参数说明	备注
id	分类 ID	
name	分类名称	
pid	分类父 ID	
level	分类当前层级	

13.2 增加商品分类

请求参数

参数名	参数说明	备注
pid	分类父 ID	不能为空
name	分类名称	不能为空
level	分类层级	不能为空，1 表示二级分类；2 表示三级分类

13.3 根据 ID 查询商品分类

请求方法：get

请求参数

参数名	参数说明	备注
-----	------	----

参数名	参数说明	备注
id	分类 ID	不能为空

13.4 修改商品分类数据

请求方法：put

请求参数

参数名	参数说明	备注
id	分类 ID	不能为空
name	分类名称	不能为空

13.5 删除商品分类数据

请求方法：delete

请求参数

参数名	参数说明	备注
id	分类 ID	不能为空

14 前端商品分类管理的实现

14.1 商品分类列表的显示

一般情况下，我们在显示列表时，是通过表格显示，但由于分类菜单有层级结构，在显示时，我们又想能看出层级关系，所以我们可让树显示在列表里。

但是 elementUI 默认是不支持这样显示，所以我们可以安装三方依赖 (vue-table-with-tree)

地址：<https://github.com/MisterTaki/vue-table-with-tree-grid>

代码如下：

```
<tree-table
border
show-index
index-text="#"
:expand-type="false"
:selection-type="false"
:data="cateList"
:columns="columns"
class="tree-table"
>

<template slot="level" slot-scope="scope">
  <el-tag v-if="scope.row.level === 1">一级菜单</el-tag>
  <el-tag v-else-if="scope.row.level === 2" type="success">二级菜单</el-tag>
  <el-tag v-else type="warning">三级菜单</el-tag>
</template>

<template slot="opt">
```

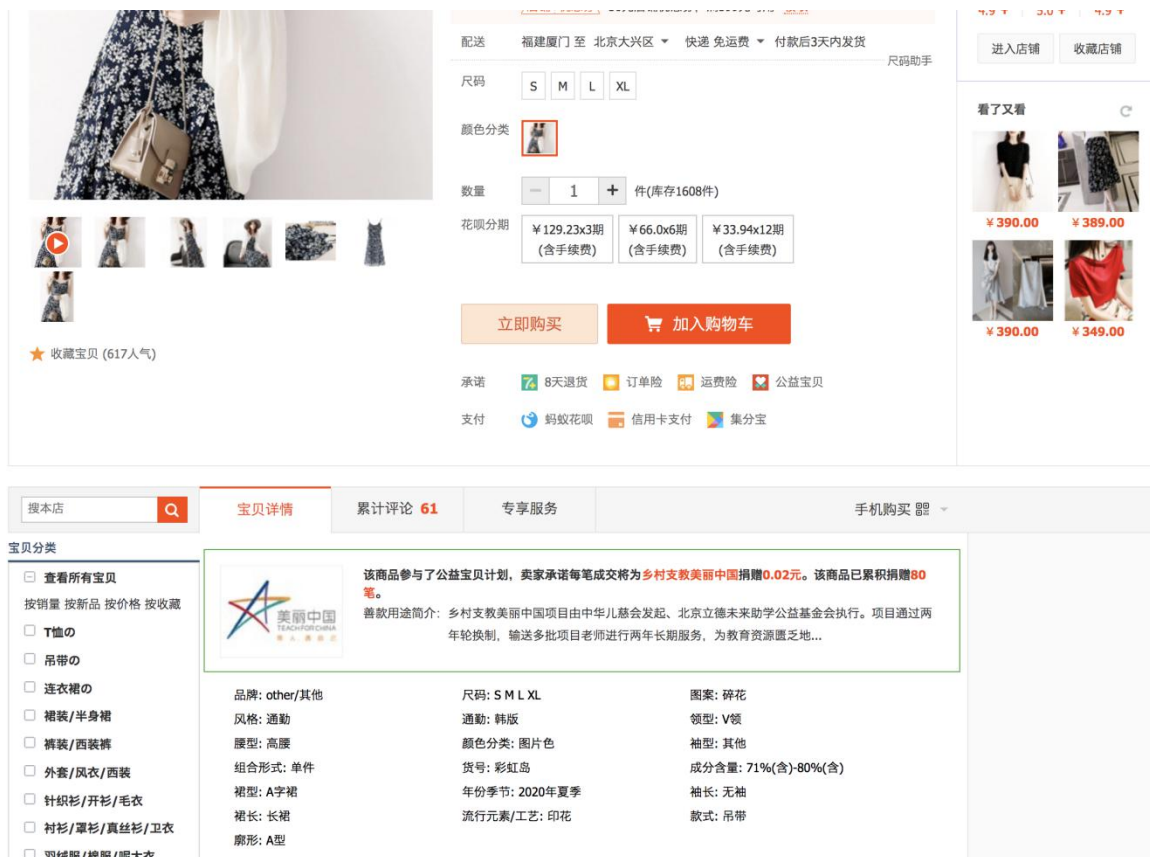
```
<el-button size="mini" type="primary" icon="el-icon-edit">编辑</el-button>

<el-button size="mini" type="danger" icon="el-icon-delete"> 删除 </el-
button>

</template>

</tree-table>
```

15 后端分类参数管理的实现



model 的设计

```
class Attribute(db.Model):
```

```
__tablename__ = 't_attribute'

id = db.Column(db.Integer, primary_key=True)

name = db.Column(db.String(32))

_type = db.Column(db.Enum('static', 'dynamic'))

val = db.Column(db.String(255))

cid = db.Column(db.Integer, db.ForeignKey('t_category.id'))


def to_dict(self):

    return {

        'id': self.id,

        'name': self.name,

        'type': self._type,

        'values': self.val,

        'cid': self.cid

    }
```

15.1 分类参数数据列表

请求方法：get

请求参数

参数名	参数说明	备注
-----	------	----

参数名	参数说明	备注
cid	分类 ID	不能为空
_type	[static,dynamic]	不能为空,通过 static 或 dynamic 来获取分类静态参数还是动态参数

响应参数

参数名	参数说明	备注
id	分类参数 ID	
name	分类参数名称	
cid	分类参数所属分类	
_type	静态参数或动态参数	
val	静态：单值 动态：该值以逗号分隔	

15.2 添加动态参数或者静态属性

请求方法：post

请求参数

参数名	参数说明	备注
cid	分类 ID	不能为空

参数名	参数说明	备注
name	参数名称	不能为空
_type	[static,dynamic]	不能为空
val	静态：单值 动态：该值以逗号分隔	【可选参数】

15.3 根据 ID 查询分类参数

请求方法：get

请求参数

参数名	参数说明	备注
id	参数 ID	不能为空

15.4 修改分类数据

请求方法：put

请求参数

参数名	参数说明	备注
cid	分类 ID	不能为空
id	属性 ID	不能为空
name	新属性的名字	不能为空

参数名	参数说明	备注
val	参数的属性值	可选参数

15.5 删除分类参数数据

请求方法：delete

请求参数

参数名	参数说明	备注
id	分类参数 ID	不能为空

16 前端分类参数管理的实现

因为参数分为固定参数与动态参数。所以在此我们可以通过使用 Tabs、TabPane 实现同一个分类的参数与值的显示

```
<el-tabs v-model="activeName" @tab-click="handleClick">
  <el-tab-pane label="静态参数" name="static">
    <el-button
      type="primary"
      size="mini"
      :disabled="isBtnDisable"
      @click="addDialogVisible = true">
```



```
>增加参数</el-button>

<el-table :data="staticAttr">

  <el-table-column type="expand">

    <template slot-scope="scope">

      <el-tag>{{ scope.row.val}}</el-tag>

    </template>

  </el-table-column>

  <el-table-column type="index"></el-table-column>

  <el-table-column label=" 参 数 名 称 " prop="name"></el-table-
column>

  <el-table-column label="操作">

    <template slot-scope="scope">

      <el-button type="success" size="mini">编辑</el-button>

      <el-button type="danger" size="mini">删除</el-button>

    </template>

  </el-table-column>

</el-table>

</el-tab-pane>

<el-tab-pane label="动态参数" name="dynamic">

  <el-button

    type="primary"
```

```

size="mini"

:disabled="isBtnDisable"

@click="addDialogVisible = true"
>增加参数</el-button>

<el-table :data="dynamicAttr">

  <el-table-column type="expand">

    <template slot-scope="scope">

      <el-tag

        @click="removeTag(scope.row, i)"

        closable

        v-for="(v,i) in scope.row.val"

        :key="i"

      >{{v}}</el-tag>

      <el-input

        class="input-new-tag"

        v-if="scope.row.inputVisible"

        v-model="scope.row.inputValue"

        ref="saveTagInput"

        size="small"

        @keyup.enter.native="handleInputConfirm(scope.row)"

        @blur="handleInputConfirm(scope.row)"

```

```

        > </el-input>

        <el-button

        v-else

        class="button-new-tag"

        size="small"

        @click="showInput(scope.row)"

        > + New Tag </el-button>

    </template>

</el-table-column>

<el-table-column type="index"> </el-table-column>

<el-table-column label=" 参 数 名 称 " prop="name"> </el-table-
column>

<el-table-column label="操作">

    <template slot-scope="scope">

        <el-button type="success" size="mini">编辑</el-button>

        <el-button type="danger" size="mini">删除</el-button>

    </template>

</el-table-column>

</el-table>

</el-tab-pane>

</el-tabs>

```

在获取参数后，我们可以通过 v-for 来遍历出值来，但是在显示值时，静态的参数还是好显示的，但是在动态的参数值时，是在同一个字段上，是通过逗号分割开成多个值。所以在显示前需要对动态参数的值做下处理。

```
async getAttribute() {
  const { data: resp } = await this.$axios.get('/category/attr_list', {
    params: { cid: this.selectKeys[2], _type: this.activeName }
  })
  if (resp.status !== 200) return this.$msg.error(resp.msg)
  console.log(resp.data)
  if (this.activeName === 'static') {
    this.staticAttr = resp.data
    this.staticFlag = false
  } else {
    resp.data.forEach(item => {
      item.val = item.val ? item.val.split(',') : []
      item.inputVisible = false
      item.inputValue = ''
    })
    this.dynamicAttr = resp.data
    this.dynamicFlag = false
  }
}
```

```
}
```

17 后端商品管理的实现

17.1 商品数据列表

请求方法：get

请求参数

参数名	参数说明	备注
name	查询参数	可以为空

17.2 增加商品

请求方法：post

请求参数

参数名	参数说明	备注
name	商品名称	不能为空
price	价格	不能为空
number	数量	不能为空
weight	权重	不能为空

参数名	参数说明	备注
introduce	介绍	可以为空
pics	上传的图片临时路径（对象）	可以为空
attr_static	商品的静态参数（数组）	可以为空
attr_dynamic	商品的动态参数	
cid_one	1 级分类	不能为空
cid_two	2 级分类	不能为空
cid_three	3 级分类	不能为空

17.3 删除商品数据

请求方法：delete

请求参数

参数名	参数说明	备注
id	分类参数 ID	不能为空

17.4 图片上传

请求方法：post

请求参数

参数名	参数说明	备注
-----	------	----

参数名	参数说明	备注
file	上传文件	

18 前端商品管理的实现

对于商品的管理有 CRUD，在之前的功能中基本都一一实现了，但是商品的增加时字段过于多，在一个 dialog 中显示内容过长，因此可以通过进度条（Step、Steps）加标签页(Tabs,TabPane)来实现

```
<el-tabs      v-model="active"      :tab-position="'left'"      :before-leave="beforeLeave">

  <el-tab-pane label="基本信息" name="0">

    <el-form-item label="商品名称" prop="name">

      <el-input v-model="addForm.name"> </el-input>

    </el-form-item>

    <el-form-item label="商品价格" prop="price">

      <el-input v-model="addForm.price"> </el-input>

    </el-form-item>

    <el-form-item label="商品数量" prop="number">

      <el-input v-model="addForm.number"> </el-input>

    </el-form-item>

    <el-form-item label="商品权重" prop="weight">
```

```

        <el-input v-model="addForm.weight"> </el-input>

    </el-form-item>

    <el-form-item label="商品分类" prop="cid">

        <el-cascader

            v-model="selectKeys"

            :options="cateIdList"

            :props="{ expandTrigger: 'hover', label:'name', value:'id'}"

            clearable

            separator=" > "

            @change="changeSeletor"

        > </el-cascader>

    </el-form-item>

</el-tab-pane>

<el-tab-pane label="商品静态参数" name="1">

    <el-form-item :label="s.name" v-for="s in attr_static" :key="s.id">

        <el-input v-model="s.val"> </el-input>

    </el-form-item>

</el-tab-pane>

<el-tab-pane label="商品动态参数" name="2">

    <el-form-item          :label="d.name"          v-for="d          in
attr_dynamic" :key="d.id">

```



```

        <el-checkbox-group v-model="d.val">
            <el-checkbox :label="dv" v-for="(dv,i) in d.val" :key="i"
border> </el-checkbox>

        </el-checkbox-group>

    </el-form-item>

</el-tab-pane>

<el-tab-pane label="商品图片" name="3">

    <el-upload

        class="upload-demo"

        action="/upload_img"

        :on-preview="handlePreview"

        :on-remove="handleRemove"

        :on-success="handleSuccess"

        list-type="picture"

    >

        <el-button size="small" type="primary">点击上传</el-button>

    </el-upload>

</el-tab-pane>

<el-tab-pane label="商品内容" name="4">

    <quill-editor v-model="addForm.introduce"> </quill-editor>

    <el-button type="primary" @click="goodsAdd" class="btnAdd">

```

添加商品</el-button>

</el-tab-pane>

</el-tabs>

19 后端订单管理的实现

19.1 订单数据列表

请求方法：get

请求参数

参数名	参数说明	备注
id	订单 ID	可以为空

19.2 查看物流信息

请求方法：get

请求参数

参数名	参数说明	备注
iod	订单 ID	可以为空

20 前端订单管理的实现

在订单中想要获取订单的物流信息时，一般都为某个时间到了哪个地方。这时，我们可以通过时间线(TimeLine)来实现显示

```
<el-dialog title="物流信息" :visible.sync="expressVisible">
  <el-timeline :reverse="reverse">
    <el-timeline-item
      v-for="(activity, index) in expressList"
      :key="index"
      :timestamp="activity.update_time"
    >{{activity.content}}</el-timeline-item>
  </el-timeline>
</el-dialog>
```

21 后端数据统计的实现

一般在项目中可以存在着数据的统计行为，这时就会就需要一个聚合函数，在此我们统计物品分类各等级有多少分类

```
@category.route('/cate_group_level')

def get_cate_group_by_level():

    # group_data
    =models.Category.query.group_by(models.Category.level).having(models.C
    ategory.level > 0).all()

    group_data =
    db.session.query(models.Category.level,func.count(1).label('count')).group_
    by(models.Category.level).having(models.Category.level >0).all()

    data={

        'name':'数量',

        'xAxis':[f'{g[0]}级分类' for g in group_data],

        'series_data':[g[1] for g in group_data]

    }

    return to_dict_msg(200,data,'获取统计数据成功！！')
```

22 前端数据统计的实现

在前端为了更好的显示出统计的数据一般会选择用图表的形式，但是在 vue 与 elementUI 中没有图表的直接。所以我们可以安装 Echarts 的依赖来实现图表的显示。

具体显示图表的代码如下：

```
<template>
```

```

<div>

  <el-breadcrumb separator-class="el-icon-arrow-right">

    <el-breadcrumb-item :to="{ path: '/home' }"> 首页 </el-breadcrumb-
item>

    <el-breadcrumb-item>数据统计</el-breadcrumb-item>

    <el-breadcrumb-item>商品统计</el-breadcrumb-item>

  </el-breadcrumb>

  <el-card>

    <!-- 为 ECharts 准备一个具备大小（宽高）的 DOM -->

    <div id="main" style="width: 600px;height:400px;"> </div>

  </el-card>

</div>

</template>

<script>

import echarts from 'echarts'

export default {

  async mounted() {

    const { data: resp } = await this.$axios.get('/cate_group_level')

    if (resp.status !== 200) return this.$msg.error(resp.msg)

    // 基于准备好的 dom，初始化 echarts 实例
  }
}

```

```
var myChart = echarts.init(document.getElementById('main'))

// 指定图表的配置项和数据

var option = {

  title: {

    text: 'ECharts 入门示例'

  },

  tooltip: {},

  legend: {

    data: [resp.data.name]

  },

  xAxis: {

    data: resp.data.xAxis

  },

  yAxis: {},

  series: [

    {

      name: resp.data.name,

      type: 'bar',

      data: resp.data.series_data

    }

  ]

}
```

```

    }

    // 使用刚指定的配置项和数据显示图表。
    myChart.setOption(option)
  }
}
</script>

```

问题：

1 前端代码验证十分严格如何解决？

1.1 关闭 eslint 验证

在创建项目时，不开启 eslint 功能即可 (Use ESLint to lint your code?)

1.2 配置合适的规则

规则的含义：

“off” or 0 - 关闭(禁用)规则

“warn” or 1 - 将规则视为一个警告（并不会导致检查不通过）

“error” or 2 - 将规则视为一个错误（退出码为 1，检查不通过）

1.2.1 常用规则

属性	含义
Possible	Errors 可能的错误或逻辑错误

属性	含义
no-cond-assign	禁止条件表达式中出现模棱两可的赋值操作符
no-console	禁用 console
no-constant-condition	禁止在条件中使用常量表达式
no-debugger	禁用 debugger
no-dupe-args	禁止 function 定义中出现重名参数
no-dupe-keys	禁止对象字面量中出现重复的 key
no-duplicate-case	禁止出现重复的 case 标签
no-empty	禁止出现空语句块
no-ex-assign	禁止对 catch 子句的参数重新赋值
no-extra-boolean-cast	禁止不必要的布尔转换
no-extra-parens	禁止不必要的括号
no-extra-semi	禁止不必要的分号
no-func-assign	禁止对 function 声明重新赋值
no-inner-declarations	禁止在嵌套的块中出现变量声明或 function 声明
no-irregular-whitespace	禁止在字符串和注释之外不规则的空白
no-obj-calls	禁止把全局对象作为函数调用
no-sparse-arrays	禁用稀疏数组
no-prototype-builtins	禁止直接使用 Object.prototype 的内置属性
no-unexpected-multiline	禁止出现令人困惑的多行表达式
no-unreachable	禁止在 return、throw、continue 和 break 语句之后出现不可达代码
use-isnan	要求使用 isNaN() 检查 NaN
valid-typeof	强制 typeof 表达式与有效的字符串进行比较

1.2.2 Best Practices 最佳实践

属性	含义
array-callback-return	强制数组方法的回调函数中有 return 语句
block-scoped-var	强制把变量的使用限制在其定义的作用域范围内
complexity	指定程序中允许的最大环路复杂度
consistent-return	要求 return 语句要么总是指定返回的值，要么不指定
curly	强制所有控制语句使用一致的括号风格
default-case	要求 switch 语句中有 default 分支
dot-location	强制在点号之前和之后一致的换行
dot-notation	强制在任何允许的时候使用点号
eqeqeq	要求使用===和!==
guard-for-in	要求 for-in 循环中有一个 if 语句
no-alert	禁用 alert、confirm 和 prompt
no-case-declarations	不允许在 case 子句中使用词法声明
no-else-return	禁止 if 语句中有 return 之后有 else
no-empty-function	禁止出现空函数
no-eq-null	禁止在没有类型检查操作符的情况下与 null 进行比较
no-eval	禁用 eval()
no-extra-bind	禁止不必要的.bind()调用
no-fallthrough	禁止 case 语句落空
no-floating-decimal	禁止数字字面量中使用前导和末尾小数点
no-implicit-coercion	禁止使用短符号进行类型转换
no-implicit-globals	禁止在全局范围内使用 var 和命名的 function 声明
no-invalid-this:	禁止 this 关键字出现在类和类对象之外
no-lone-blocks	禁用不必要的嵌套块

属性	含义
no-loop-func	禁止在循环中出现 function 声明和表达式
no-magic-numbers	禁用魔术数字
no-multi-spaces	禁止使用多个空格
no-multi-str	禁止使用多行字符串
no-new	禁止在非赋值或条件语句中使用 new 操作符
no-new-func	禁止对 Function 对象使用 new 操作符
no-new-wrappers	禁止对 String , Number 和 Boolean 使用 new 操作符
no-param-reassign	不允许对 function 的参数进行重新赋值
no-redeclare	禁止使用 var 多次声明同一变量
no-return-assign	禁止在 return 语句中使用赋值语句
no-script-url	禁止使用 javascript:url
no-self-assign	禁止自我赋值
no-self-compare	禁止自身比较
no-sequences	禁用逗号操作符
no-unmodified-loop-condition	禁用一成不变的循环条件
no-unused-expressions	禁止出现未使用过的表达式
no-useless-call	禁止不必要的.call()和.apply()
no-useless-concat	禁止不必要的字符串字面量或模板字面量的连接
vars-on-top	要求所有的 var 声明出现在它们所在的作用域顶部

1.2.3 严格模式

属性	含义
Strict	Mode 使用严格模式和严格模式指

属性	含义
strict	要求或禁止使用严格模式指令

1.2.4 Variables 变量声明

属性	含义
init-declarations	要求或禁止 var 声明中的初始化
no-catch-shadow	不允许 catch 子句的参数与外层作用域中的变量同名
no-restricted-globals	禁用特定的全局变量
no-shadow	禁止 var 声明与外层作用域的变量同名
no-undef	禁用未声明的变量，除非它们在/global/注释中被提到
no-undef-init	禁止将变量初始化为 undefined
no-unused-vars	禁止出现未使用过的变量
no-use-before-define	不允许在变量定义之前使用它们

1.2.5 Nodejs and CommonJS Node.js, CommonJS

属性	含义
global-require	要求 require() 出现在顶层模块作用域中
handle-callback-err	要求回调函数中有容错处理

属性	含义
no-mixed-requires	禁止混合常规 var 声明和 require 调用
no-new-require	禁止调用 require 时使用 new 操作符
no-path-concat	禁止对 dirname 和 filename 进行字符串连接
no-restricted-modules	禁用指定的通过 require 加载的模块

1.2.6 Stylistic issues 风格指南

属性	含义
array-bracket-spacing	强制数组方括号中使用一致的空格
block-spacing	强制在单行代码块中使用一致的空格
brace-style	强制在代码块中使用一致的大括号风格
camelcase	强制使用骆驼拼写法命名约定
comma-spacing	强制在逗号前后使用一致的空格
comma-style	强制使用一致的逗号风格
computed-property-spacing	强制在计算的属性的方括号中使用一致的空格
eol-last	强制文件末尾至少保留一行空行
func-names	强制使用命名的 function 表达式
func-style	强制一致地使用函数声明或函数表达式

属性	含义
indent	强制使用一致的缩进
jsx-quotes	强制在 JSX 属性中一致地使用双引号或单引号
key-spacing	强制在对象字面量的属性中键和值之间使用一致的间距
keyword-spacing	强制在关键字前后使用一致的空格
linebreak-style	强制使用一致的换行风格
lines-around-comment	要求在注释周围有空行
max-depth	强制可嵌套的块的最大深度
max-len	强制一行的最大长度
max-lines	强制最大行数
max-nested-callbacks	强制回调函数最大嵌套深度
max-params	强制 function 定义中最多允许的参数数量
max-statements	强制 function 块最多允许的的语句数量
max-statements-per-line	强制每一行中所允许的最大语句数量
new-cap	要求构造函数首字母大写
new-parens	要求调用无参构造函数时有圆括号
newline-after-var	要求或禁止 var 声明语句后有一行空行

属性	含义
newline-before-return	要求 return 语句之前有一空行
newline-per-chained-call	要求方法链中每个调用都有一个换行符
no-array-constructor	禁止使用 Array 构造函数
no-continue	禁用 continue 语句
no-inline-comments	禁止在代码行后使用内联注释
no-lonely-if	禁止 if 作为唯一的语句出现在 else 语句中
no-mixed-spaces-and-tabs	不允许空格和 tab 混合缩进
no-multiple-empty-lines	不允许多个空行
no-negated-condition	不允许否定的表达式
no-plusplus	禁止使用一元操作符++和--
no-spaced-func	禁止 function 标识符和括号之间出现空格
no-ternary	不允许使用三元操作符
no-trailing-spaces	禁用行尾空格
no- whitespace-before-property	禁止属性前有空白
object-curly-newline	强制花括号内换行符的一致性
object-curly-spacing	强制在花括号中使用一致的空格

属性	含义
object-property-newline	强制将对象的属性放在不同的行上
one-var	强制函数中的变量要么一起声明要么分开声明
one-var-declaration-per-line	要求或禁止在 var 声明周围换行
operator-assignment	要求或禁止在可能的情况下要求使用简化的赋值操作符
operator-linebreak	强制操作符使用一致的换行符
quote-props	要求对象字面量属性名称用引号括起来
quotes	强制使用一致的反勾号、双引号或单引号
require-jsdoc	要求使用 JSDoc 注释
semi	要求或禁止使用分号而不是 ASI
semi-spacing	强制分号之前和之后使用一致的空格
sort-vars	要求同一个声明块中的变量按顺序排列
space-before-blocks	强制在块之前使用一致的空格
space-before-function-paren	强制在 function 的左括号之前使用一致的空格
space-in-parens	强制在圆括号内使用一致的空格
space-infix-ops	要求操作符周围有空格
space-unary-ops	强制在一元操作符前后使用一致的空格

属性	含义
spaced-comment	强制在注释中//或/*使用一致的空格

1.3 使用 prettier 插件

在项目的根目录下，新建 .prettierrc.json 配置文件

配置格式化方式

```
{  
  "singleQuote":true,  
  "semi":false,  
  "tabWidth":2,  
  "bracketSpacing":true,  
  "jsxBracketSameLine":true  
}
```

配置属性：<https://prettier.io/docs/en/options.html>