前端分工

陈实立：

网站主页（功能添加）

- 热门商品
- 可以跳转到所有商品和商品详情
- 可以跳转到所有店铺页
- 有搜索框（搜商品或店铺）

所有商品页(用的分类)

- 展示所有商品
- 可以跳转到商品详情页
- 有搜索框（可以搜商品）

所有店铺页

- 展示所有店铺
- 可以跳转到店铺详情页
- 有搜索框（可以搜店铺）

商品详情页

- 展示商品的详细信息
- 猜你喜欢
- 可以添加到购物车

购物车

- 展示购物车中的商品
- 购物车的删除功能（支持批量删除和单个删除）
- 购物车商品失效（刷新页面即可）

柏露：

账户信息

- 展示用户的个人信息和账户余额
- 可以进行充值

我的商店

- 展示商店信息
- 展示商店中的商品

- 可以添加、修改或下架商品
- 可以申请删除商店

管理员账户

- 展示商城利润账户和商城购物中间账户的余额
- 可以进行充值

商品申请记录

- 展示当前商户的申请记录

管理员审核商品申请记录

- 展示记录
- 审核功能

后端分工

周训哲：

新的数据库的设计

- 添加商品表
- 添加申请表
- 添加账户表
- 添加账单表
- 添加购物车表

完成部分类的接口设计

- UserBehavior
- ShopperBehavior
- AdminBehavior
- ShoppingCartOperation
- ShopOperation
- GoodOperation

完成controller中部分函数的设计

- User
- Shopper
- Admin

张智雄：

后端架构的设计

- controller层负责前端和后端的数据交互
- service层负责进行数据库间接交互，整合数据库数据

- model层负责与数据库直接交互

后端的面向对象设计

- 数据库的每一张表都对应models中的一个具体的类
- 使用Operation类对数据库中的数据进行连接获取
- 使用Behavior类对controller层和model层的数据进行包装和处理

完成部分类的接口设计

- BillOperation
- AccountOperation
- RequestOperation
- ImageOperation

完成controller中部分函数的设计

- Image
- Visitor

实验设计

前端代码规范

我们在项目中启用 ESLint 规则检查，配置规则为 `eslint:recommentded`

```
require('@rushstack/eslint-patch/modern-module-resolution')

module.exports = {
  root: true,
  'extends': [
    'plugin:vue/vue3-essential',
    'eslint:recommended',
    '@vue/eslint-config-prettier/skip-formatting'
  ],
```

该规范的详细说明见 ESLint 官网，其中标有 ✅ 的规则均在 `eslint:recommentded` 中启用（我们遵守这些规则）

前端代码风格说明

我们前端的代码风格遵守 vue 官方文档的风格指南

组件名为多个单词

组件名应该始终是多个单词的，根组件 `App` 以及 `<transition>`、`<component>` 之类的 Vue 内置组件除外。

这样做可以避免跟现有的以及未来的 HTML 元素相冲突，因为所有的 HTML 元素名称都是单个单词的。

组件数据

组件的 `data` 必须是一个函数。

当在组件中使用 `data` property 的时候 (除了 `new Vue` 外的任何地方)，它的值必须是返回一个对象的函数。

Prop 定义必要

Prop 定义应该尽量详细。

在你提交的代码中，prop 的定义应该尽量详细，至少需要指定其类型。

<details>
<summary>详解</summary>

细致的 **prop 定义**有两个好处：

- 它们写明了组件的 API，所以很容易看懂组件的用法；
- 在开发环境下，如果向一个组件提供格式不正确的 prop，Vue 将会告警，以帮助你捕获潜在的错误来源。

</details>

为 `v-for` 设置键值

总是用 `key` 配合 `v-for`。

在组件上*总是*必须用 `key` 配合 `v-for`，以便维护内部组件及其子树的状态。甚至在元素上维护可预测的行为，比如动画中的对象固化 (object constancy)，也是一种好的做法。

组件文件

只要有能够拼接文件的构建系统，就把每个组件单独分成文件。

当你需要编辑一个组件或查阅一个组件的用法时，可以更快速的找到它。

单文件组件文件名的大小写

单文件组件的文件名应该要么始终是单词大写开头 (PascalCase)，要么始终是横线连接 (kebab-case)。

单词大写开头对于代码编辑器的自动补全最为友好，因为这使得我们在 JS(X) 和模板中引用组件的方式尽可能的一致。然而，混用文件命名方式有的时候会导致大小写不敏感的文件系统的问题，这也是横线连接命名同样完全可取的原因。

基础组件名

应用特定样式和约定的基础组件 (也就是展示类的、无逻辑的或无状态的组件) 应该全部以一个特定的前缀开头，比如 `Base`、`App` 或 `V`。

单例组件名

只应该拥有单个活跃实例的组件应该以 `The` 前缀命名，以示其唯一性。

这不意味着组件只可用于一个单页面，而是*每个页面*只使用一次。这些组件永远不接受任何 prop，因为它们是为你的应用定制的，而不是它们在你的应用中的上下文。如果你发现有必要添加 prop，那就表明这实际上是一个可复用的组件，*只是目前*在每个页面里只使用一次。

上面只列出了我们代码中经常使用的风格，更多风格说明可以参考

https://github.com/vuejs/v2.cn.vuejs.org/blob/master/src/v2/style-guide/index.md

后端

实验设计

1、后端使用经典的类mvc架构设计。

2、我们将数据模型，路由控制，业务逻辑分离开来。数据模型在model模块中，路由控制在controller模块中，业务逻辑在service模块中。

3、数据库中的每张表都有一个自己的orm属性类，定义在model.models内，类内部定义了表的属性名，同时还有构造函数，和将属性转换为键值对的功能。

同时对于每张表的操作有一个Operation类，例如Users表有一个UserOperation类，其中定义了对User表的一些操作，如create_user(), modify_user()，get_user()，分别对应创建用户，修改用户信息，和获取用户信息三种数据库操作。

4、service模块中定义了一些行为类，我们认为项目的功能对应不同的用户行为，有管理员，普通用户，商户，游客四种不同的行为人，于是有AdminBehavior，UserBehavior，ShopperBehavior，VistorBehavior四种不同的行为类。行为类里定义了每种用户的行为函数，一个行为有多个数据库操作组成，如商户申请开店对应于在开店申请表创建开店申请，将注册资金转移至管理员中间账户两个数据库操作。两个操作构成一个事务。这样不仅体现了事务的一致性和原子性，同时大大降低了各个模块的耦合度，便于新增功能及代码维护。

5.以上3，4中的类均为单例模式，我们只需要一个对象既可以实现功能。

6.controller模块中用于处理前端不同的路由对应的功能，根据路由对参数进行处理，然后调用service模块中不同的行为完成操作，并将返回值处理包装后发回给前端。

7.我们还有一个utils模块和tests模块。

utils模块对应了一些实用的小工具，如请求参数的检验， 返回值的包装，密钥的生成与匹配等工具。

tests模块包含一些单元测试，每实现一个模块或功能便使用tests模块进行测试，以确保开发的顺利进行。

代码风格

1.变量名和函数名使用小写字母+下划线

2.二元运算符两端需要一个空格， （函数参数内的=号两端不需要空格）

3.类名和文件名用大驼峰法

4.不准把注释当作删除代码来用，即注释中只能由规定的功能说明，而不能存在无用的代码

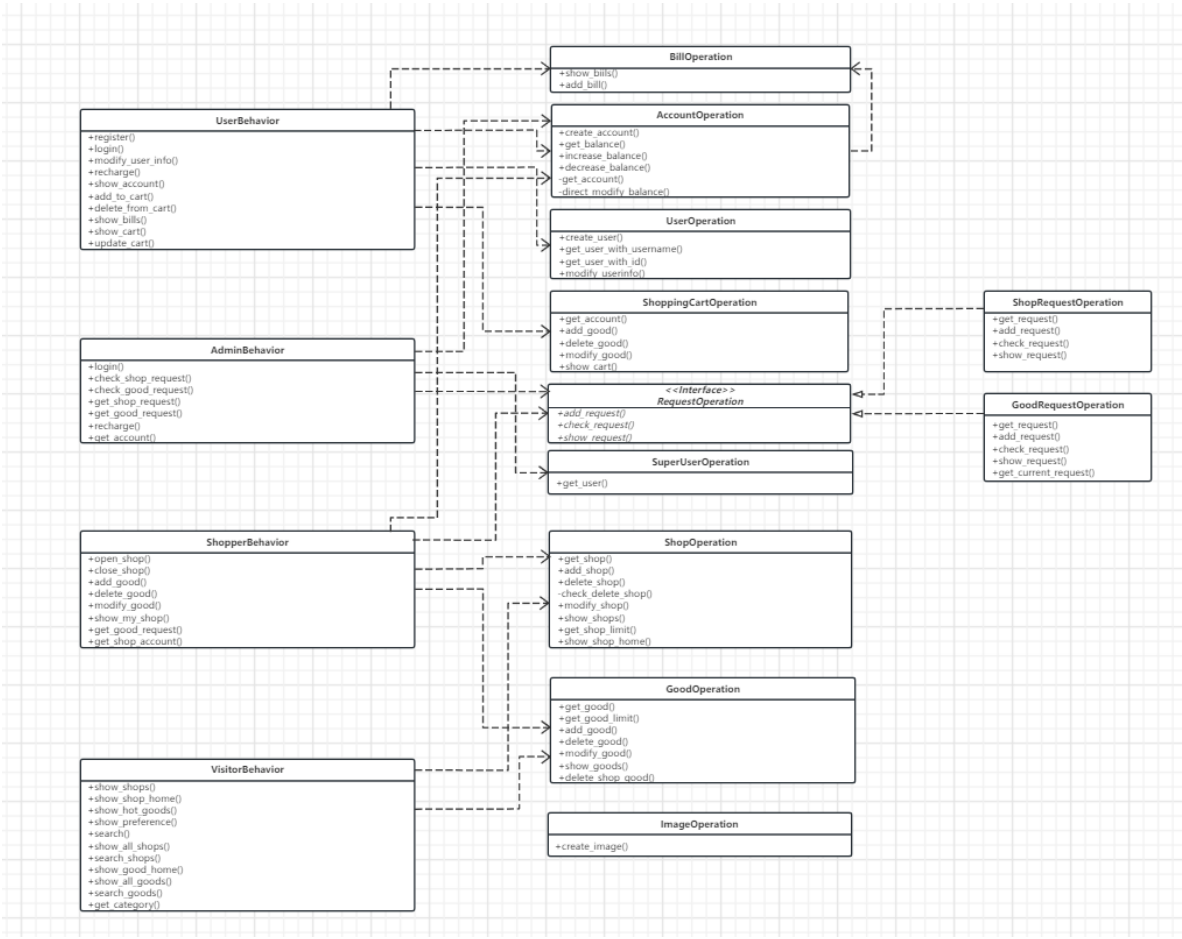
5.不允许出现魔法数字，所有常数必须使用有意义的变量名定义在规定的配置文件内

6.类之间的数据传输均使用`dict`类进行传输，方便数据管理

7.在数据传输过程中必须包含键`'err'`用于显示错误，为`None`则说明没有错误

面向对象设计

本次实验后端进行了大刀阔斧的改进，实现了面向对象设计，将操作包装成类，使用接口对数据进行调用，以下是我们设计的uml类图：



注释要求

1、每个函数都要在开头使用`"""`三个引号这种注释形式注明函数功能，参数类型及返回值类型，如下图

```
class AccountOperation:
    def create_account(self, data):
        """
        创建账户
        params:
        data(dict) {
            key: "account_type", value: AccountType
            [Optional]key: "id_num", value: str
            [Optional]key: "shop_id", value: int
        }
        return:
        dict {
            key: "result", value: Accounts
            key: "err", value: str
        }
        """
```

2、在函数内部，对于每个较长的代码段需要用#这种注释形式说明代码段的功能，如下图：

```
if request_msg['result'].request_type == RequestType.open:
    if data['approval'] is True:
        # 获取注册资金
        shop_msg = ShopOperationObject.get_shop(data)
        if shop_msg['err'] is not None:
            session.rollback()
            return {'err': shop_msg['err']}
        # 将注册资金从中间账户转到商城利润账户
        middle_data = {
            'account_type': AccountType.middle,
            'amount': shop_msg['result'].register_capital
        }
        middle_msg = AccountOperationObject.decrease_balance(middle_data)
```

遇到问题及解决方案

前端

陈实立遇到的问题主要集中在：如何实现功能，如何设计组件，如何复用代码，以及 js 的语法细节

1. 搜索功能如何实现按下回车键发起搜索，如何通过路由参数传递搜索关键字?

按下回车键发起搜索需要在搜索框中添加监听案件时间的函数，然后判断按下的键是否是回车，如果是回车就发起搜索请求（跳转到对应界面，然后发起请求，并通过路由参数传递搜索关键字）。

```
<input v-if="type === `goods`" type="search" placeholder="搜商品" v-
model="keyword" @keydown.enter="searchEnterFun" />
```

```
function searchEnterFun(e){
      let keyCode = window.event ? e.keyCode : e.which;
      if(keyCode == 13){
        console.log(keyword.value)
        if(props.type === 'goods'){
          router.push({path:"/goods", query:{keyword : keyword.value}})
        }
        if(props.type === 'shops'){
          router.push({path:"/shops", query:{keyword : keyword.value}})
        }
      }
    }
```

goods 和 shops 中接受路由参数（监听路由参数的变化，一旦发生变化则发起搜索请求）。

```
created() {
    // watch 路由的参数，以便再次获取数据
    this.$watch(
      () => this.$route.query,
      () => {
        console.log("create watch")
        this.modifyKeywords(this.$route.query.keyword)
      },
      // 组件创建完后获取数据，
      // 此时 data 已经被 observed 了
      { immediate: true }
    )
  },
```

2. 如何对界面设置访问显示？使用路由守卫（对管理员和购物车界面进行了限制）

```
// 购物车，管理员这两个界面需要登录后才可以进入

import store from "@/store";
import Message from "@/components/library/Message";

/**
 * 登录拦截--路由守卫
 * @param to 新路由信息
 * @param from 原路由信息
 * @param next 放行
 */
export default function authGuard(to, from, next) {
  // 指明需要登录的地址
  const requireLogin = ["cart", "admin"];

  // 判断前往的页面是否需要拦截守卫
  if (requireLogin.includes(to.path.split("/")[1])) {
    // 判断用户是否登录
    if (!store.state.user.profile.token) {
      // 记录重定向地址
      store.commit("user/setRedirectURL", to.fullPath);
      // 未登录
      next({ path: "/login", query: { redirectURL: to.fullPath } });
      Message({ type: 'success', text: "请先登录" });
    } else {
```

```
      // 已登录
      next();
    }
  } else {
    // 访问的页面无需登录
    next();
  }

  if (from.path === "/login") {
    console.log(from);
    // 清空重定向地址
    // store.commit("user/setRedirectURL");
  }
}
```

3. 如何实现吸顶头部组件（页面滚动到78px以上，显示吸顶组件）？

该组件的完成主要有两个个步骤：

第一步：监听滚动距离超过78px后隐藏原有的头部导航栏。

第二步：为了吸顶头部的内容不遮住不吸顶的头部，在滚动距离超过78px后才显示该组件。

其中滚动距离的监听是通过 useWindowScroll() （@vueuse/core提供的api）可返回当前页面滚动时候蜷曲的距离。

```
<template>
  <div class="app-header-sticky" :class="{ show: scrollTop > 78 }">
    <div class="container" v-show="scrollTop > 78">
      <RouterLink to="/" class="logo" />
      <AppHeaderNav />
      <div class="right">
        <RouterLink to="/">品牌</RouterLink>
        <RouterLink to="/">专题</RouterLink>
      </div>
    </div>
  </div>
</template>

<script>
import AppHeaderNav from "@/components/AppHeaderNav";
import useScrollTop from "@/hooks/useScrollTop";
// import { useWindowScroll } from "@vueuse/core";
export default {
  name: "AppHeaderSticky",
  components: { AppHeaderNav },
  setup() {
    const scrollTop = useScrollTop();
    // const { y: scrollTop } = useWindowScroll();

    console.log("top", scrollTop);
    return { scrollTop };
  },
};
</script>

<style scoped lang="less">
.app-header-sticky {
```

```
  width: 100%;
  height: 80px;
  position: fixed;
  left: 0;
  top: 0;
  z-index: 999;
  background-color: #fff;
  border-bottom: 1px solid #e4e4e4;
  transform: translateY(-100%);
  opacity: 0;
  &.show {
    transform: none;
    opacity: 1;
    transition: all 0.3s linear;
  }
  .container {
    display: flex;
    align-items: center;
  }
  .logo {
    width: 200px;
    height: 80px;
    background: url(../assets/images/logo.png) no-repeat right 2px;
    background-size: 160px auto;
  }
  .right {
    width: 220px;
    display: flex;
    text-align: center;
    padding-left: 40px;
    border-left: 2px solid @xtxColor;
    a {
      width: 38px;
      margin-right: 40px;
      font-size: 16px;
      line-height: 1;
      &:hover {
        color: @xtxColor;
      }
    }
  }
}
</style>
```

show类是通过动态绑定的方式添加到外层div上的，当scrollTop变量的值大于78时，show类会被添加到外层div上，从而显示导航栏。

4. 如何图片懒加载?

当图片进入可视区域内去加载图片，且处理加载失败，封装成指令。

基于vue3.0和IntersectionObserver封装懒加载指令（请求图片信息）

观察dom进入可视区离开可视区都会触发 IntersectionObserver

```
import { ref } from "vue";
import { useIntersectionObserver } from "@vueuse/core";
```

```
export default function useLazyData(apiFn) {
  // 1.创建对象元素
  const target = ref(null);
  // 2.存储数据
  const result = ref(null);
  // 3.监听元素进入可视区
  const { stop } = useIntersectionObserver(
    target,
    ([{ isIntersecting }]) => {
      console.log("懒加载", isIntersecting); //@log
      if (isIntersecting) {
        // 停止监听懒加载
        console.log("懒加载2"); //@log
        stop();
        // 调用API 获取数据
        apiFn().then((res) => {
          console.log("layData:", res); //@log
          if(res.goods) result.value = res.goods;
          if(res.result) result.value = res.result;
          console.log("layData:", result.value); //@log
          console.log(typeof result.value); //@log
        });
      }
    },
    { threshold: 0 }
  );
  // 返回数据
  return { target, result };
}
```

在显示页面时判断图片是否已经加载完毕，否则只显示骨架

```
        <ul class="goods-list" v-if="goods">
          <li v-for="item in goods" :key="item.id">
            <RouterLink :to="`/good/${item.id}`">
              <view v-for="(picture,index) in item.images" :key="index">
                <view v-if="index==0">
                  <img :src="picture" alt="item.goodname" />
                </view>
              </view>
              <p class="name ellipsis">{{ item.goodname }}</p>
              <p class="price">&yen;{{ item.price }}</p>
            </RouterLink>
          </li>
        </ul>
        <HomeSkeleton v-else />
```

5. 购物车的设计：购物车模块大部分信息会存放在后端，但是选中商品和价格计算的工作需要在前端完成，如何实现这种功能？

前端的购物车状态通过 vuex 持久化来维护。

这部分代码实在是太长了，这里不做展示，具体可以参看前端目录下的 `store/cart.js` 文件。

后端

问题1：

原先的代码结构过于混乱，模块之间耦合度过高且逻辑上不够清晰，不便于后续的维护

解决方案：

将原先的代码结构推翻，并使用经典的mvc结构进行设计，将数据模型，路由控制，业务逻辑分离。

问题2：

使用orm框架提供的session与数据库进行交互，发现可能存在并发问题。

解决方案：

在使用的框架官网查到如何创造线程安全的session，并进行修改。

问题3：

在代码仓库中发现很多配置文件和二进制文件

解决方案：

.gitignore编写有误，上网查询并修改之后解决。

问题4：

对于每个数据库操作都进行了单独的commit或rollback，如开店申请需要同时进行转账和添加申请两部操作，将其分开则违背了事务的原子性和一致性。

解决方案：

重新设计代码，将所有功能抽象为用户行为，一个用户行为视为一次事务，由多个数据库操作组成，只进行一次commit或rollback。

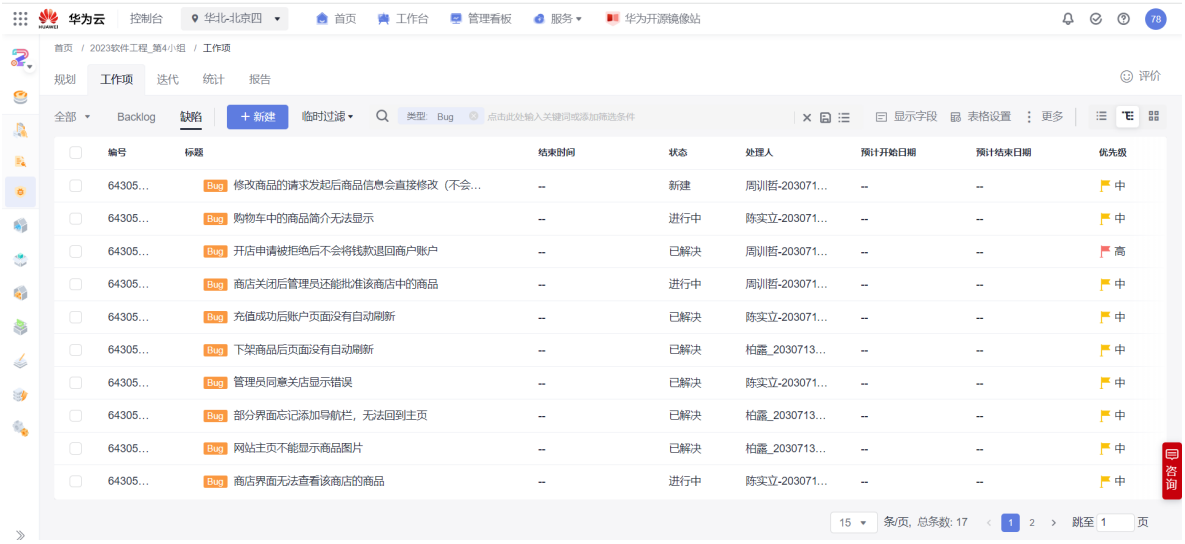
问题5：

图片的传输使用本地自增编号进行存储，在数据库中存储的是编号的值的字符串，传输到前端时则用编号进行获取相应文件，然后直接传输即可。

代码检查结果

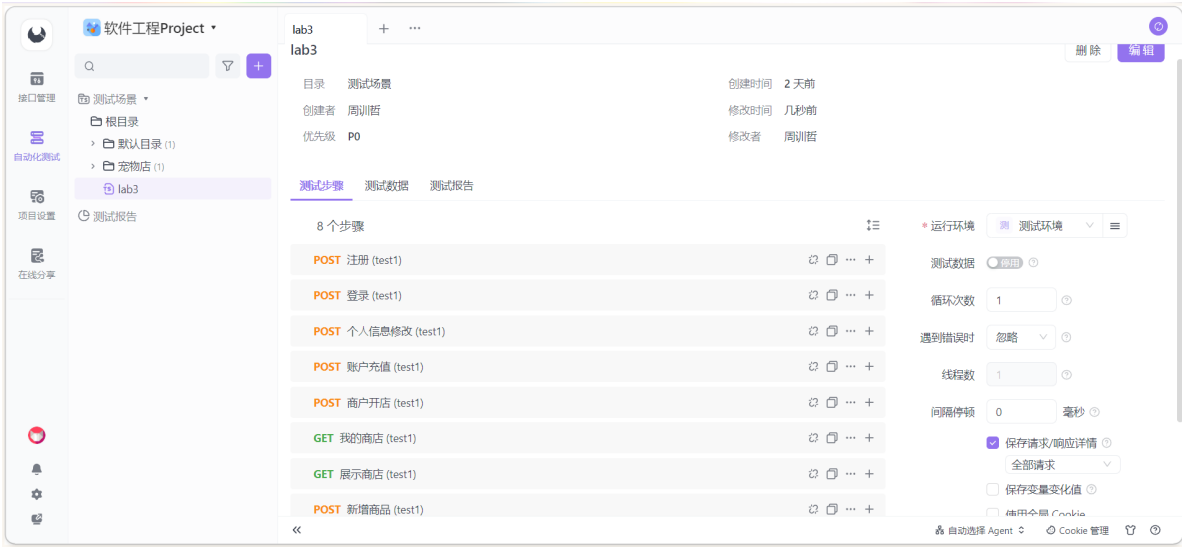
缺陷管理

我们按照 Lab3 文档中的要求对项目开发过程中进行了缺陷记录和管理：对每个缺陷进行了描述、分析、和优先级评定以及跟踪等，整个缺陷管理过程我们使用了华为云自带的缺陷管理功能，如下图：

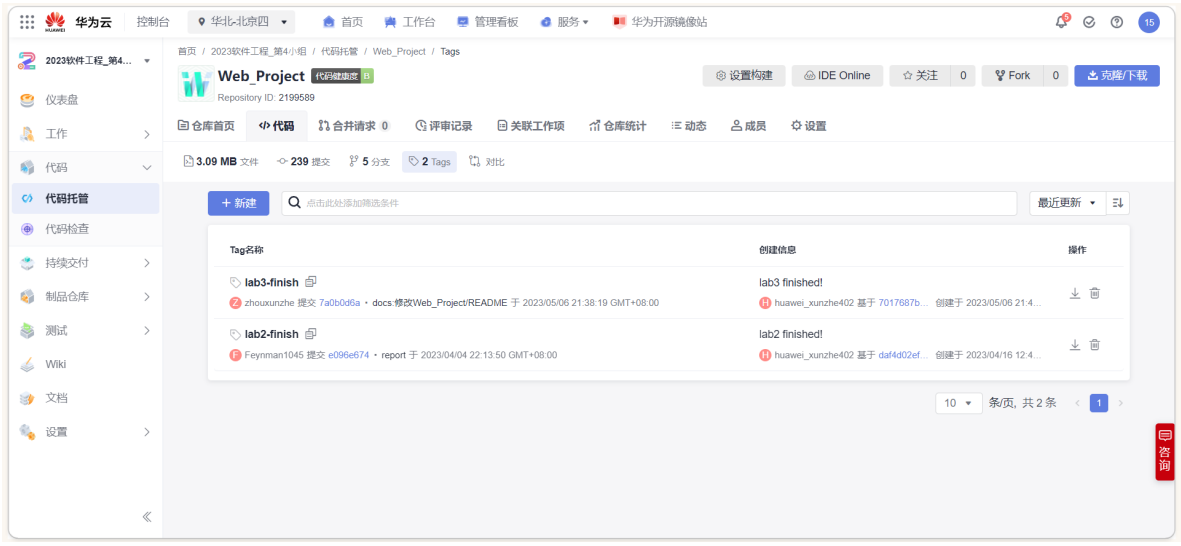


选做项完成情况

我们不是使用华为云，而是选用了apifox的自动化测试作为我们小组的代码评审实现测试驱动开发，一方面在api的url中加入测试样例进行测试，另一方面使用自动化测试实现网站完整操作流程的模拟。测试截图如下：



tag提交记录



心得体会

陈实立

在第二次软工实验中，我们团队开发的效率明显高出不少，大部分接口都在开发前确定下来，很少有开发过程中修改接口的情况；我们在沟通的过程中增加了线下开会的次数，事实证明线下的沟通才是最高效的。

在这次实验中，需求显著增加，任务量也明显加大，前端工作比较繁重，而且我在前端部分学习了一些新的技术，这使得我花了非常长的时间来完成这次的实验。希望在之后的实验中任务量可以稍微减少一点，为我们留出更多的时间来优化我们的代码，完善我们的开发流程。

此外，我认为我们在本次实验中我们的缺陷管理做得还不够完善，希望可以在后续的实验中进行改进。

周训哲

在第二次的软工实验中，后端工作量十分大，有很多新增的功能需求，对于数据库的设计和整个代码结构的维护都是一个巨大的挑战。于是我们推翻了第一次lab中简陋的代码结构，使用了经典的mvc架构，将数据模型，业务逻辑，路由控制分离开来，学习了面向对象设计的思想，体会到了高内聚，低耦合带来的遍历，对于软件工程的设计模式，代码维护有了更深的理解。

但是在这次实验过程中仍然出现出现前后端沟通不顺利，导致数据类型等传输错误。同时我们在开发过程中对于模块的测试还不够充分，经常都是开发了很多之后，才发现之前编写的代码有bug，导致很多重复的工作量。

在下次实验中应该继续增加团队的沟通，尽量每周一次会讨论后续工作方向以尽可能协调工作。

张智雄

在第二次的软工实验中，后端工作量十分大，有很多新增的功能需求，对于数据库的设计和整个代码结构的维护都是一个巨大的挑战。于是我们推翻了第一次lab中简陋的代码结构，使用了经典的mvc架构，将数据模型，业务逻辑，路由控制分离开来，学习了面向对象设计的思想，体会到了高内聚，低耦合带来的遍历，对于软件工程的设计模式，代码维护有了更深的理解。

但是，我们在开发过程中对于模块的测试还不够充分，经常都是开发了很多之后，才发现之前编写的代码有bug，导致很多重复的工作量。

柏露

在第二次软工实验中，我们团队开发的效率明显高出不少，大部分接口都在开发前确定下来，很少有开发过程中修改接口的情况。

在这次实验中，出现上传图片的功能，让我学会了前端如何上传图片，收益很多。这次实验的工作量比较大，由于我们只有最开始和最后面线下开了会，导致我们对后端的完成情况不了解，出现了最后两天后端赶代码的情况，这也导致了前端和后端的联调只有在最后一天才能完成，导致了最后一天的工作量比较大，我们应在增加开会的次数，交流工作完成情况，自定一些分阶段的目标，在完成分阶段的目标后就进行线下联调，这样才可能不出现这种前后端完成情况不一致导致最后一天才能完成最后测试的情况。