Alibaobao

1.项目简介

Alibaobao是一款模仿alibaba的oop实践项目，这个项目项目基于本机文件，提供了单机化的购物平台体验，当然，这种体验并不好，因为我们的能力和时间有限。Alibaibao的核心功能包括了 1.注册与登录，用户可以注册为三种不同类型的账号，admin，seller，consumer，而不同类型的账号登录之后都有各自不同的界面 2.客户界面可以在产品列表中对产品浏览信息，进行挑选并加入购物车，在个人信息中对账户充值之后即可进行结算，随后跳转到评价界面完成评价3.商户界面则提供了进货渠道，可以为商品进行分类后填写详细的商品属性并上架 4.管理者界面则简单划分为了五个功能功能按钮：查看用户信息、查看账单信息、查看日志、删除用户、修改产品价格。三种不同的账号的不同功能串联起来组成了一个较为完整的购物平台。接下来我将对该项目进行详细的技术介绍。

2.整体架构

该项目以python编程语言为基础，引进了tkinter作为用户接口；引进了json作为数据库；引进了美化文件sv\_ttk，使整体更加美观（美化文件并不是自己开发的，而是从github上下载的）；引进了os以便对文件进行操作；引进了datetime为日志提供实时时间信息

为了满足“使用装饰器”的要求，项目首先编写了日志记录装饰器（我没有办法想出能使用装饰器的地方，这个用途甚至是咨询AI后开发的）对项目内所有函数的调用进行记录，以方便管理者查看。

我们创建了核心类：user类。在这个类中，我们编写了进行界面初始化的方法：creat\_login\_page。在代码的末尾我们定义main()函数并执行，来对user类进行初始化属性编辑并调用其方法来初始化界面。初始化登录界面之后，我们添加了两个输入框和确认按钮，以及跳转至注册界面的链接，这里我们先来解释注册系统。注册界面以tk为接口，有两个输入框和一个选择框，这些输入在注册逻辑中被调用并储存在users.json的文件中。在登录界面调用了登录逻辑，调用用户的输入，并查找他们是否在users.json中存在对应关系。定义了加密方法\_\_get\_user\_tpye，以方便对其进行调用（但实际并没有应用的地方）。定义了log\_out方法，这方法将在后面被调用用于返回至登录界面。定义了加密方法\_\_is\_logged\_in，以便检查用户是否登录（但实际并没有应用的地方）。定义了初始化用户信息的方法，这个是因为后期频繁出现未找到文件报错而进行的处理，在登录逻辑中调用，使用户在登录后即创建个人信息。

定义了抽象方法creat\_first\_page，这个是user类中最重要的方法。这个方法将在它的三个子类中被重写，分别接入三种账号各自的用户界面，从而体现类的多态性。

我们定义了一个名为管理员的类，该类继承自User类。管理员类代表了一个具有管理员权限的用户，能够执行特定的管理操作。管理员类的构造函数接收三个参数：username、password和master。在构造函数中，首先调用父类User的构造函数，确保父类被正确初始化。接着，初始化一个名为\_\_管理员\_actions的私有列表，用于存储管理员执行的操作。我们在管理员类中使用了被装饰器@log\_function\_call修饰的create\_first\_page方法来创建管理员界面。在create\_first\_page方法中，首先创建一个全新的界面。然后创建一个标题为“管理员 Interface”（管理员界面），接下来，创建一个名为button\_frame的框架，用于放置按钮。我们创建了五个按钮，查看用户信息、查看账单数据、查看日志、删除用户、修改产品价格。同时我们创建了view\_logs 方法用于显示功能日志。然后创建新窗口、创建 Treeview：定义了四个列：时间戳（Timestamp）、功能（Function）、用户（User）和状态（Status）。 Treeview：使用 pack 方法将 Treeview 添加到窗口中，接着添加读取日志文件功能：检查名为 function\_logs.json 的日志文件是否存在。存在则使用 tree.insert 方法将日志信息添加到 Treeview 中。不存在则显示一个信息消息框。我们还是用、、使用添加滚动条处理大量日志数据时的滚动问题。最后调用 self.\_\_log\_admin\_action("Viewed function logs") 方法可能用于记录管理员查看日志的操作。管理员同时还有管理用户数据的权力，我们首先定义了一个名为view\_users的方法用于打开一个新的窗口，显示用户信息。该方法还会记录管理员查看用户信息的操作。接着定义了view\_bills方法用于，显示账单数据。此外，还为账单数据表格添加了垂直滚动条，以提高用户体验。该方法也会记录管理员查看账单数据的操作。prompt\_delete\_user和delete\_user方法用于删除用户。首先输入要删除的用户名。调用delete\_user方法来执行删除操作。此外，prompt\_modify\_product\_price和modify\_product\_price方法用于修改产品价格。管理员输入产品名称和新价格，调用modify\_product\_price方法来执行修改操作。最后定义了\_\_log\_admin\_action方法，用于记录管理员的操作日志。

我们定义了一个名为Seller的类，这个类继承User类。卖家类代表了一个具有卖家权限的用户，能够执行特定的销售操作。卖家类的构造函数接收三个参数：username、password和master。首先调用父类User的函数，确保父类被正确初始化。然后，初始化一个名为attribute\_entries的字典，用于存储卖家在进货时输入的产品属性。

在卖家类中，我们使用了被装饰器@log\_function\_call修饰的create\_first\_page方法来创建卖家的界面。在create\_first\_page方法中，创建了一个名为main\_frame的框架，用来放置进货表单和库存信息。

在左侧的进货框架中，卖家可以输入商品的类别、子类别、名称、数量和价格。商品类别可以通过下拉框选择，子类别会根据所选类别动态更新。卖家还可以输入与所选子类别相关的特定属性，这些属性通过attribute\_entries字典进行存储。我们定义了submit\_restock按钮，用于处理进货操作。当卖家点击提交按钮时，验证输入的数量和价格是否有效，并将新商品或更新现有商品的信息写入product\_data.json文件。

在右侧的库存框架中，卖家可以查看当前库存的商品信息。通过读取product\_data.json文件，卖家可以看到所有商品的名称、数量和价格。库存信息以表格的形式展示，使用Treeview组件来显示。

我们还定义了一个名为Consumer的类，该类继承自User类。Consumer类代表了一个消费者用户，具有特定的属性和方法，用于在一个基于图形用户界面(GUI)的购物应用程序中展示和操作商品信息。

首先父类User的构造函数，传递了用户名、密码和主窗口（master）。然后，初始化了消费者的购物车为一个空字典。使用initialize\_user\_info方法用于初始化用户信息。create\_first\_page方法用于创建消费者的首页界面。它可以创建一个欢迎标签和构建一个包含产品信息的表格视图在表格下方，创建了一个包含多个按钮的底部框架。一个用于选择产品的下拉框（Combobox），一个用于输入购买数量的数字调节器（Spinbox），一个“添加到购物车”按钮，一个“查看产品信息”按钮，一个“购物车”按钮，以及一个“我的信息”按钮。这些按钮分别绑定了不同的方法，用于执行相应的操作。值得注意的是，代码中使用了ttk模块来创建GUI控件，这是Tkinter的一个扩展，提供了更现代和一致的外观。此外，还使用了json模块来处理JSON文件，以及os模块来检查文件的存在性。代码中的@log\_function\_call装饰器（尽管没有给出其定义）可能用于记录函数的调用情况，这在调试和监控应用程序行为时可能非常有用。定义了几个关键的功能，包括将商品添加到购物车、打开购物车查看已选商品以及查看个人信息。这些功能通过操作JSON文件来实现数据的持久化存储。首先，add\_to\_cart函数允许用户将选中的商品添加到他们的购物车中。open\_cart函数用于打开购物车界面，显示用户已添加到购物车中的所有商品。open\_my\_info函数用于打开个人信息界面，显示用户的个人信息和余额。定义了一个名为create\_top\_up\_page的方法，该方法用于创建一个充值页面。在这个页面中，首先会销毁主窗口（所有现有控件，确保页面干净。接着，创建头部框架（header\_frame），放置一个返回按钮（back\_button）和一个标签（label）“Top Up”（充值）。在头部框架下方，创建一个充值金额输入框（self.top\_up\_amount\_entry），允许用户输入充值金额。最后，添加了一个提交按钮（submit\_button），当用户点击该按钮时，会调用process\_top\_up方法来处理充值逻辑。process\_top\_up方法首先尝试从充值金额输入框中获取用户输入的金额，并将其转换为浮点数。如果金额有效，程序尝试读取用户信息文件，并根据文件中的余额更新充值金额。接下来，定义了create\_product\_review\_page方法，用于创建一个产品评价页面。在页面下方，创建了一个评价输入框架（review\_frame），允许用户输入产品名称、评分（1-5星）和评价内容。最后，定义了一个名为checkout的方法，用于处理购物车的结算逻辑。如果购物车不为空，程序会计算总账单金额，并尝试读取用户信息文件和账单文件。如果用户的余额足够支付账单，程序会更新用户信息文件中的余额，清空购物车文件，并保存新的账单记录到账单文件中。view\_product\_info方法的主要功能是显示选定产品的详细信息以及用户评论。它通过self.product\_combobox.get()获取用户从下拉列表中选择的产品名称。程序将创建一个新的顶层窗口（Toplevel），使用ttk.Notebook创建了一个标签页控件，用于组织产品信息和评论的显示。接下来，程序尝试从product\_data.json文件中读取产品数据。这个文件应该包含产品名称、类别、子类别、价格、库存、属性以及评论。程序将在Notebook的第一个标签页（info\_frame）中显示产品的详细信息，包括产品名称、类别、子类别、价格和库存。在Notebook的第二个标签页（reviews\_frame）中，程序使用ttk.Treeview控件显示产品的评论。每条评论包括用户名、评分和评论内容。get\_cart\_contents方法的功能是返回当前购物车内容的副本。购物车内容被存储在一个名为\_\_shopping\_cart的私有成员变量中，这个方法通过复制这个变量来返回购物车的内容，而不直接返回原始变量，以保护购物车数据不被外部修改。

我们定义了一个抽象类Product类，在Product类下面定义了三个大的子类 ElectronicProduct、ClothingProduct和FoodProduct，用于区分不同商品的大类，每个大的子类下面有若干个小的子类，用于进一步细分不同类型的商品。每个商品大类都继承并多态了display\_info和calculate\_discount方法，以便展示商品的牌子，价格等信息和计算折扣的功能，细分的子类也继承并多态了display\_info方法，可以展示商品库存等更精确的信息。其中我们设计了SmartWatch这件商品，它同时继承了电子产品和服装产品的类，让这件商品同时拥有品牌、尺码和操作系统等属性，更加贴近实际。整个Product类用于构建了产品分类管理系统，让不同类型的产品能够共享通用的属性和方法，同时又能根据产品不同的特点进行扩展。

3.缺点与不足

当然，我们的项目也存在漏洞和缺点，最主要的缺点就是在使用类时，并没有对对象进行实例化并直接直接调用其属性和方法，大部分情况下，我们选择直接从数据库中读取对应信息并调用。实际上当我们意识到要这么做的时候，再去对源代码进行修改，已经来不及了。同时，项目还存在很多需要优化和改进的地方，比如对数据库的初始化，当客户端被首次使用时被按照错误的顺序使用时（例如未进行进货就开始采购）应该给出更好的优化方案来提升用户体验，而不是直接弹出错误框。同时，由于专注于核心内容的开发，我们并没有花太多精力用于提升用户接口的美化，因此整个UI显得十分丑陋。同时，还有很多应该完善的功能尚未实现，例如商品折扣功能，我们写入核心代码中的方法是直接对对象的属性进行打折修改，而当实际应用时，我们因为并未实现在运行时的对象实例化，而很难写入结算逻辑，这迫使我们放弃（当然，对于产品类的多态化实现，我们使用display\_info方法来完成）。项目还有更多可以优化用户体验的地方，但是因为时间等各方面元素的影响，使本项目的表现结果并不那么理想，但我相信，只要时间充足，我们可以将它做的更加完美。

4.开发历程与心得

这是我们第一次开发如此完整的项目，难免会出现因经验不足而产生的错误。在项目的早期策划上我们显得犹豫不决，从而浪费了大量时间。对于逻辑的开发我们比较了解，但是对接口的开发确实很生疏，这就导致在策划阶段的多种方案中我们只能选择了最简单的tkinter来进行接口的开发。然而即使选择了tk，我们仍然在接口开发上没有头绪，因为大一学年我们并没有接触到tk开发有关知识（大一期末项目py小游戏我们都是通过终端来作为用户接口的），所以我们只能从零开始进行学习，这就导致了我们在前期浪费了大量时间。在开发过程中，随着项目的复杂程度的提升，出现的bug也是越来越多，这也促使我们在排障能力上进行了一定程度的锻炼与提升。在处理数据上，我们最初愚昧地只想通过字典来储存关键数据，而后我们意识到json文件的优越之处，并改用json对数据进行保存。在项目的开发中，我们运用了本次课程学习的各类知识：类的继承，类的层次化，类的多态性，多重继承，对敏感信息的加密，抽象类，抽象方法，try expect语法的运用，装饰器的使用等等。这些知识的应用让我们对这些知识有了更深刻的印象，并让我们能够更好的理解他们各自的用途。在开发的过程中，底层逻辑，按钮，接口之间相互连接，存在着复杂的关系联系，在策划上也提升了我们的逻辑思维能力和统筹策划能力。这次项目的开发是艰难的，但也是充满收获的，我们在这个过程中学到了很多，这也是我们由学员成长为一个合格的程序员的必经之路。

\*\*Alibaobao\*\*

1. \*\*Project Overview\*\*

Alibaobao is an OOP practice project that mimics Alibaba. This project is based on local files and provides a standalone shopping platform experience. However, this experience is not ideal due to our limited capabilities and time. The core functions of Alibaobao include:

1. Registration and login: Users can register for three different types of accounts: admin, seller, or consumer. Each type of account has a different interface after logging in.

2. The consumer interface allows users to browse product information, select products, add them to the shopping cart, and proceed to checkout after recharging their account in the personal information section. After payment, users are redirected to a review page to complete product reviews.

3. The seller interface provides a supply channel where merchants can categorize products, fill in detailed product attributes, and list them for sale.

4. The admin interface is divided into five simple functional buttons: view user information, view billing information, view logs, delete users, and modify product prices. The different functionalities of the three types of accounts are interconnected to form a relatively complete shopping platform. Next, I will provide a detailed technical introduction to the project.

2. \*\*Overall Architecture\*\*

This project is based on the Python programming language and uses Tkinter for the user interface, JSON as the database, and sv\_ttk to enhance the visual design (this file was downloaded from GitHub, not self-developed). It also introduces the OS library for file operations and the datetime module to provide real-time timestamp information for logs.

To meet the requirement of "using decorators," the project first implemented a logging decorator (I couldn't think of where else to use a decorator, and this usage was developed after consulting AI). This decorator records the invocation of all functions in the project to facilitate admin monitoring.

We created the core class: the `user` class. In this class, we wrote a method for initializing the interface: `create\_login\_page`. At the end of the code, we defined the `main()` function and executed it to initialize the attributes of the `user` class and call its methods to initialize the interface. After initializing the login interface, we added two input fields, a confirmation button, and a link to the registration interface. Here, we will first explain the registration system. The registration interface uses Tk as the interface, with two input fields and a selection box. These inputs are used in the registration logic and stored in the `users.json` file. The login logic is called in the login interface, which takes the user input and checks whether there is a corresponding entry in `users.json`. We defined the encryption method `\_\_get\_user\_type` to facilitate calling it (though it's not used in practice). We also defined the `log\_out` method, which will be used later to return to the login interface. Another encryption method, `\_\_is\_logged\_in`, was defined to check if the user is logged in (but is not used in practice). We also defined a method to initialize user information, which was introduced to handle frequent file-not-found errors later in the process. This method is called in the login logic to create user information immediately after login.

We defined the abstract method `create\_first\_page`, which is the most important method in the `user` class. This method is overridden in the three subclasses to connect to the respective user interfaces of the three types of accounts, thus demonstrating the polymorphism of the class.

We defined a class called \*\*Admin\*\*, which inherits from the \*\*User\*\* class. The \*\*Admin\*\* class represents a user with administrative privileges and can perform specific management operations. The constructor of the \*\*Admin\*\* class takes three parameters: `username`, `password`, and `master`. In the constructor, we first call the parent class \*\*User\*\*'s constructor to ensure that the parent class is properly initialized. Then, we initialize a private list named `\_\_admin\_actions` to store the operations performed by the admin.

In the \*\*Admin\*\* class, we use the `create\_first\_page` method, decorated with `@log\_function\_call`, to create the admin interface. In the `create\_first\_page` method, we first create a brand-new interface. Then, we create a title labeled "Admin Interface" and proceed to create a frame named `button\_frame` to hold the buttons. We create five buttons: View User Info, View Bill Data, View Logs, Delete User, and Modify Product Price.

Additionally, we define the `view\_logs` method to display function logs. In this method, a new window is created, and a \*\*Treeview\*\* is set up with four columns: Timestamp, Function, User, and Status. The \*\*Treeview\*\* is added to the window using the `pack` method. Next, we add functionality to read log files by checking if a log file named `function\_logs.json` exists. If it exists, the log information is added to the \*\*Treeview\*\* using the `tree.insert` method. If it does not exist, an information message box is displayed. We also implement a scrollbar to handle the issue of scrolling through large amounts of log data. Finally, the method `self.\_\_log\_admin\_action("Viewed function logs")` is called to log the action of the admin viewing logs.

Admins also have the authority to manage user data. We first define a method called `view\_users` to open a new window displaying user information. This method also logs the admin's action of viewing user information. Then, we define the `view\_bills` method to display bill data. A vertical scrollbar is added to the bill data table to enhance the user experience, and the admin’s action of viewing bill data is logged.

The `prompt\_delete\_user` and `delete\_user` methods are used to delete a user. First, the admin inputs the username to be deleted. The `delete\_user` method is then called to execute the deletion. Additionally, the `prompt\_modify\_product\_price` and `modify\_product\_price` methods are used to modify product prices. The admin inputs the product name and the new price, and the `modify\_product\_price` method is called to carry out the modification. Lastly, we define the `\_\_log\_admin\_action` method to log the admin’s actions.

---

We also defined a class named \*\*Seller\*\*, which inherits from the \*\*User\*\* class. The \*\*Seller\*\* class represents a user with seller privileges, enabling them to perform specific sales-related operations. The constructor of the \*\*Seller\*\* class takes three parameters: `username`, `password`, and `master`. First, we call the parent class \*\*User\*\*'s constructor to ensure that the parent class is properly initialized. Then, we initialize a dictionary named `attribute\_entries` to store the product attributes input by the seller during restocking.

In the \*\*Seller\*\* class, we use the `create\_first\_page` method, decorated with `@log\_function\_call`, to create the seller’s interface. In the `create\_first\_page` method, we create a frame named `main\_frame` to hold the restocking form and inventory information.

On the left side of the restocking frame, the seller can input product details such as category, subcategory, name, quantity, and price. The product category can be selected from a dropdown list, and the subcategory dynamically updates based on the selected category. The seller can also input specific attributes related to the selected subcategory, which are stored in the `attribute\_entries` dictionary. We define a `submit\_restock` button to handle restocking operations. When the seller clicks the submit button, the input quantity and price are validated, and the new product or updated product information is written to the `product\_data.json` file.

On the right side of the inventory frame, the seller can view the current inventory information. By reading the `product\_data.json` file, the seller can see all products’ names, quantities, and prices. The inventory information is displayed in a table format using the \*\*Treeview\*\* component.

We also defined a class named \*\*Consumer\*\*, which inherits from the \*\*User\*\* class. The \*\*Consumer\*\* class represents a consumer user with specific attributes and methods designed to display and interact with product information in a graphical user interface (GUI)-based shopping application.

First, the constructor of the \*\*User\*\* parent class is called, passing the username, password, and main window (`master`). Then, the consumer's shopping cart is initialized as an empty dictionary. The `initialize\_user\_info` method is used to initialize the user's information. The `create\_first\_page` method creates the homepage interface for the consumer. It includes a welcome label and a table view to display product information. Below the table, a bottom frame is created, containing multiple buttons: a product selection dropdown (Combobox), a numeric spinner (Spinbox) for inputting purchase quantity, an "Add to Cart" button, a "View Product Info" button, a "Shopping Cart" button, and a "My Information" button. Each button is bound to a different method to perform the corresponding operation.

Notably, the `ttk` module is used to create the GUI controls, which is an extension of Tkinter that provides a more modern and consistent appearance. Additionally, the `json` module is used to handle JSON files, and the `os` module is used to check file existence. The `@log\_function\_call` decorator (although its definition is not provided) is likely used to log function calls, which can be useful for debugging and monitoring application behavior.

Several key functions are defined, including adding products to the shopping cart, opening the cart to view selected items, and viewing personal information. These functions handle persistent data storage by working with JSON files. First, the `add\_to\_cart` function allows users to add selected products to their cart. The `open\_cart` function opens the shopping cart interface, displaying all items the user has added to the cart. The `open\_my\_info` function opens the personal information interface, showing the user's details and balance.

A method named `create\_top\_up\_page` is defined to create a top-up page. On this page, the main window is cleared (destroying all existing controls to ensure a clean page). Then, a header frame is created, containing a back button (`back\_button`) and a label ("Top Up"). Below the header frame, an input box (`self.top\_up\_amount\_entry`) is created to allow the user to enter a top-up amount. Finally, a submit button is added, and when the user clicks this button, the `process\_top\_up` method is called to handle the top-up logic. The `process\_top\_up` method first attempts to retrieve the amount entered by the user and convert it into a float. If the amount is valid, the program tries to read the user information file and update the balance with the new top-up amount.

Next, a method called `create\_product\_review\_page` is defined to create a product review page. At the bottom of the page, a review input frame (`review\_frame`) is created, allowing users to input the product name, a rating (1-5 stars), and review content. Finally, a method named `checkout` is defined to handle the shopping cart's checkout logic. If the shopping cart is not empty, the program calculates the total bill and attempts to read the user information and billing files. If the user has enough balance to cover the bill, the program updates the balance in the user information file, clears the shopping cart file, and saves the new billing record in the billing file.

The `view\_product\_info` method primarily displays detailed information about the selected product and user reviews. It retrieves the product name selected from the dropdown list using `self.product\_combobox.get()`. The program creates a new top-level window (`Toplevel`) and uses `ttk.Notebook` to create tabbed views for organizing product information and reviews. The program then attempts to read the product data from the `product\_data.json` file, which should contain details such as product name, category, subcategory, price, stock, attributes, and reviews. In the first tab (`info\_frame`) of the notebook, detailed product information is displayed, including the product name, category, subcategory, price, and stock. In the second tab (`reviews\_frame`), the program uses the `ttk.Treeview` component to display user reviews. Each review includes the username, rating, and review content.

The `get\_cart\_contents` method returns a copy of the current shopping cart contents. The cart contents are stored in a private member variable named `\_\_shopping\_cart`, and this method returns a copy of the cart’s contents to protect the original data from external modification.