软件工程化说明文档

根据需求分析报告完成软件工程化

1.项目概述

1.1 项目背景

在实际生产中，超市管理工作量大，内容庞杂，亟需一个系统来辅助管理超市的商品库存，我们团队敏锐地发现了这个问题并且致力于解决，最终搭建了这个超市库存管理系统。超市库存管理系统旨在高效管理超市的商品、库存、采购和销售订单，以提高库存管理的效率，减少库存积压和商品短缺现象。

1.2 项目目标

1. 提供便捷的库存管理功能；
2. 支持采购订单和销售订单的管理；
3. 实现账号注册和权限的管理；
4. 提供友好的用户界面，支持数据的插入、删除、修改和查询；
5. 限制一些非法数据和非法行为，例如负数的价格，负数的仓储量等等。

2.系统架构

系统采用前后端分离的架构：

前端：使用Vue.js框架，实现用户界面和与后端的交互。

后端：使用Django框架，实现业务逻辑和数据库操作。

数据库：设立四个表格管理数据，同时设置触发器管理四个表格之间的关系。

3. PB & SPB

3.1 腾讯文档（借鉴了部分Scrum的内容）

https://docs.qq.com/sheet/DRm9kd1NVQnZVYnFi?tab=BB08J2

3.2 微信沟通

内容过多，此处不一一展示，具体内容可找张序备调。

4. 协作平台

4.1 Github（Git Bash等）

4.2 VSCode

...

5.(协作化说明)协作性

5.0 团队协作

团队协作整体过程的详细内容可以参见团队报告部分，此处仅简述。

在这个超市库存管理项目中，我们小组通过分工负责不同的功能模块，通过微信交流群来进行沟通，并且利用Github的代码管理功能来进行协作。

Github具有强大的代码管理功能，可以便利地处理小组多个成员代码版本不同导致潜在的冲突问题。
与此同时，我们小组成员在本地一般通过VSCode来编写代码，在免费的代码编辑器功能相对强大，具有内置的Git支持，可以直接在编辑器中进行版本控制和代码对比。

5.1 版本控制

- 这次编写项目我们使用Git进行代码版本的控制，我们的Github仓库可以记录代码的每一次修改，并且在点击 `Commits` 按钮后可以查看之前的历次提交对项目的修改时间和具体文件。
- 同时，可以通过Git Bash进行内部版本管理控制。

5.2 代码对比

在上面访问Github项目的基础上，我们点击选择 `Browse repository at this point` 按钮可以回到修改时的那个时间点查看效果。
VSCode也可以方便地查看代码对比的效果，打开VSCode，在左侧活动栏中点击 `Source Control` 图标，在 `Changes` 列表中，点击一个修改的文件，可以在编辑器中看到该文件的当前版本与暂存区版本的差异。

5.3 拉取代码

我们在在GitHub上创建和管理分支，发起拉取请求进行代码合并。当代码经过团队中负责审阅代码的人审核通过之后方可通过请求提交代码。
每个小组成员完成自己的部分代码想要提交时，可以在GitHub网站上，选择对应的分支，然后点击 `New pull request` 按钮，可以查看该分支与目标分支之间的差异，并发起合并请求。

5.4 分支管理

Github支持创建和管理多个分支，可以为不同的功能或修复创建单独的分支。
同时我们也可以在VSCode中切换分支、创建分支、合并分支等。在底部状态栏中，点击当前分支名称，可以弹出分支管理菜单，可以切换分支、创建新分支、合并分支等。

我们小组成员通过通过熟练使用这些功能，可以更好地管理项目的版本控制，提高团队协作效率。

6.(自动化说明)自动化创建和部署

6.1 前端自动化创建

我们在安装好Vue，node.js和npm后，前端通过在命令行中输入 `npm create vue@latest` 这个命令，确认好项目名称后，就可以自动创建一个默认的但是配置完整的Vue项目，我们在这个的基础上添加自己的功能就可以保证自己的Vue项目配置完备，能满足基本的运行需求。

6.2 后端自动化创建

我们在安装好anaconda后在相应的虚拟环境里面 `pip install django` 后，可以直接通过命令 `django-admin startproject myproject` 来创建一个Django后端项目，`myproject` 是后端名称，可以改成自己喜欢的。
我们这样自动创建一个默认的但是配置完整的Django项目，在这个的基础上添加自己的功能就可以保证自己的Django项目配置完备，能满足基本的运行需求。

6.3 数据库创建和连接后端

在我们的数据库软件PostgreSQL中，我们创建一个数据库，设置账号密码，同时后端在 `settings.py` 文件夹下面可以配置上数据库的信息，从而实现数据库和后端的自动化连接。

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'postgres',
        'USER': 'postgres',
        'PASSWORD': '1',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

6.4 前端自动化运行

在项目所在文件夹下，打开VSCode，输入 `npm run dev`，可以直接运行前端。

6.5 后端自动化运行

打开anaconda的命令行工具，进入安装了Django的虚拟环境，命令行一路 `cd` 进入项目所在地文件夹，输入 `python manage.py runserver` 即可直接运行后端，同时自动和数据库进行连接。

7.(软件过程说明)数据库设计

7.1 表格设计

我们超市管理系统一共拥有四个表格，每个表格负责不同的部分

Product_info表负责管理每种商品的信息

```
CREATE TABLE Product_info (
    product_id INTEGER PRIMARY KEY,
    product_name VARCHAR,
    specification VARCHAR,
    description TEXT,
    classification VARCHAR,
    price NUMERIC
);
```

Inventory表负责管理每个商品的仓储量

```
CREATE TABLE Inventory (
    product_id INTEGER PRIMARY KEY REFERENCES Product_info(product_id),
    stock_quantity INTEGER
);
```

Purchaseorder表负责管理商品购买记录

```
CREATE TABLE Purchaseorder (
    order_number VARCHAR PRIMARY KEY,
    purchase_date DATE,
    unit_price NUMERIC,
    quantity INTEGER,
    supplier VARCHAR,
    received BOOLEAN,
    product_id INTEGER REFERENCES Product_info(product_id)
);
```

Salesorder表负责管理商品销售记录

```
CREATE TABLE Salesorder (
    order_number VARCHAR PRIMARY KEY,
    sales_date DATE,
    unit_price NUMERIC,
    quantity INTEGER,
    product_id INTEGER REFERENCES Product_info(product_id)
);
```

7.2 触发器设计

为了提高系统的可用性和数据一致性，我们在数据库中设计了触发器。

我们的触发器要求满足进货销售和仓储里面的商品一定要在Product_info里面注册，进货商品增加仓储量，而销售商品减少仓储量，并且仓储量不能改成负数。同时我们的销售进货和仓储可以随时在合法数据范围内改变。

触发器1：当我们注册一个新商品后我们管理仓储量的表格自动更新，仓储值为0。

```
-- 创建触发器函数
CREATE OR REPLACE FUNCTION create_inventory_entry()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO Inventory(product_id, stock_quantity)
    VALUES (NEW.product_id, 0);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- 创建触发器
CREATE TRIGGER after_productinfo_insert_trigger
AFTER INSERT ON Product_info
FOR EACH ROW
EXECUTE FUNCTION create_inventory_entry();
```

触发器2：当进货记录发生改变时，仓储量也跟着改变。

```
-- 创建或替换触发器函数
CREATE OR REPLACE FUNCTION update_inventory_after_purchase()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        -- 增加库存
        UPDATE Inventory
        SET stock_quantity = stock_quantity + NEW.quantity
        WHERE product_id = NEW.product_id;
    ELSIF TG_OP = 'UPDATE' THEN
        -- 更新库存（例如，如果数量发生变化）
        UPDATE Inventory
        SET stock_quantity = stock_quantity + (NEW.quantity - OLD.quantity)
        WHERE product_id = NEW.product_id;
    ELSIF TG_OP = 'DELETE' THEN
        -- 减少库存
        UPDATE Inventory
        SET stock_quantity = stock_quantity - OLD.quantity
        WHERE product_id = OLD.product_id;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- 创建触发器
CREATE TRIGGER after_purchase_trigger
AFTER INSERT OR UPDATE OR DELETE ON Purchaseorder
FOR EACH ROW
EXECUTE FUNCTION update_inventory_after_purchase();
```

触发器3：当销售记录发生改变时，仓储量也跟着改变。

```
-- 创建或替换触发器函数
CREATE OR REPLACE FUNCTION update_inventory_after_sale()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        -- 减少库存
        UPDATE Inventory
        SET stock_quantity = stock_quantity - NEW.quantity
        WHERE product_id = NEW.product_id;
    ELSIF TG_OP = 'UPDATE' THEN
        -- 更新库存（例如，如果数量发生变化）
        UPDATE Inventory
        SET stock_quantity = stock_quantity - (NEW.quantity - OLD.quantity)
        WHERE product_id = NEW.product_id;
    ELSIF TG_OP = 'DELETE' THEN
        -- 增加库存（在删除销售订单时，相当于撤销销售）
        UPDATE Inventory
        SET stock_quantity = stock_quantity + OLD.quantity
        WHERE product_id = OLD.product_id;
    END IF;
    RETURN NEW;
END;
```

```
$$ LANGUAGE plpgsql;

-- 创建触发器
CREATE TRIGGER after_sale_trigger
AFTER INSERT OR UPDATE OR DELETE ON Salesorder
FOR EACH ROW
EXECUTE FUNCTION update_inventory_after_sale();
```

8.(软件过程说明)前端设计

8.1 前端整体设计

前端采用Vue.js框架，主要界面包括登录界面，主界面和四个表格对应的界面。

登录界面通过定向路由实现，当我们正确输入账号密码后，我们可以路由跳转到超市管理系统的界面，但是当我们没有正确输入账号密码时，哪怕强行输入网址也会路由跳转到登录界面。

当我们成功登录系统之后，我们看到的就是系统的主界面了，我们采用嵌套Vue的方法，实现了点击主Vue界面左侧不同的按钮，即可切换到不同的子Vue界面。

```
        <el-container>
          <el-aside class="el-aside" width="220px" max-height="630px"
style="flex: 0 0 150px; width: 100%; border: 1px solid #ddd; border-radius:
10px;" >
            <el-row class="tac">
              <el-col class="el-col-12" :span="12" style="width: 100%">
                <el-menu
                    :default-active="$route.name"
                    class="el-menu-vertical-demo"
                    @open="handleOpen"
                    @close="handleClose"
                    :router="true"
                >
                  <el-menu-item index="1" route="/home">
                    <el-icon><location /></el-icon>
                    主界面
                  </el-menu-item>

                  <el-menu-item index="2" route="/PurchaseOrder">
                    <el-icon><document /></el-icon>
                    采购订单
                  </el-menu-item>

                  <el-menu-item index="3" route="/Products">
                    <el-icon><icon-menu /></el-icon>
<!--                    <RouterLink to="/Products" :class="{'active-link':
$route.path === '/Products', 'inactive-link': $route.path !== '/Products'}">商品信
息</RouterLink>-->
                    商品信息
                  </el-menu-item>
                  <el-menu-item index="4" route="/SalesOrder">
                    <el-icon><document /></el-icon>
<!--                    <RouterLink to="/SalesOrder" :class="{'active-link':
$route.path === '/SalesOrder', 'inactive-link': $route.path !== '/SalesOrder'}">
销售订单</RouterLink>-->
```

```
                    销售订单
                  </el-menu-item>
                  <el-menu-item index="5" route="/Inventories">
                    <el-icon><setting /></el-icon>
<!--                    <RouterLink to="/Inventories" :class="{'active-link':
$route.path === '/Inventories', 'inactive-link': $route.path !==
'/Inventories'}">库存   </RouterLink>-->
                    库存
                  </el-menu-item>
                </el-menu>
              </el-col>
            </el-row>
          </el-aside>

          <el-main style="flex: 1; padding: 20px; width: 1000px;">
<!--            <RouterView />-->
            <router-view></router-view>
          </el-main>
        </el-container>
```

我们考虑到实际生产需要，给商品仓库管理系统添加了一个预警功能，通过这个我们可以很方便地实现对商品仓储量的监管和反应。

具体实现代码如下：

```
<div v-if=" row.stock_quantity < 100 && row.stock_quantity > 10"><el-button
style="width: 60px" type="warning">Warning</el-button></div>
<div v-if=" row.stock_quantity <= 10"><el-button style="width: 60px"
type="danger">Danger</el-button></div>
```

我们预警的原理就是商品仓储量小于100大于10的时候会来一个Warning提醒我们，而商品仓储量小于等于10的时候就会直接警告Danger。

效果如下：

8.2 前端数据处理

这里我们以那个存储商品信息的表格为例子，来讲讲我们这个表格的4大功能：表格的删除、修改、插入以及搜索功能在前端的实现。

表格的删除功能模块

```
<template #default="{ row }">
    <div style="display: flex; align-items: center; justify-content: space-
between;">
        <el-button type="primary" @click="editProductInfo(row)">修改</el-button>
        <el-button type="danger" @click="deleteProductInfo(row.product_id)">注销
</el-button>
    </div>
</template>
```

我们在界面上会显示一个可以调用具有删除功能的函数的按钮，点击按钮后启动 `deleteProductInfo(row.product_id)` 函数。

```
deleteProductInfo(product_id) {
    axios.delete(`http://172.18.56.143:8000/your_app_name/api/product-
info/${product_id}/`)
    .then(response => {
        if (response.status === 204) {
            console.log('Success: Purchase Order deleted');
            this.loadProductInfo();
        } else {
            console.error('Error:', response.status);
        }
    })
    .catch(error => {
        console.error('Error:', error);
    });
},
```

启动该函数后我们会向后端发送删除指令，包含需要删除的表项(使用主键进行标识)以及表项所在的表格(在这里为product_info)。后端在接收到前端的请求之后，就会接着给数据库发送指令从而实现数据库内容的删除。前段在收到后端删除操作的返回后，将再次请求最新的数据项，重新传输一次数据，我们就可以看到删除指定数据项后的数据了。

删除界面如下

表格的修改功能模块

```
<template #default="{ row }">
    <div style="display: flex; align-items: center; justify-content: space-
between;">
        <el-button type="primary" @click="editProductInfo(row)">修改</el-button>
        <el-button type="danger" @click="deleteProductInfo(row.product_id)">注销
</el-button>
    </div>
</template>
```

我们在界面上会显示一个可以调用具有修改功能的函数的按钮，点击按钮后启动`editProductInfo(row)`函数。

```
editProductInfo(products) {
    // 将订单信息加载到编辑表单中
    this.editedProduct = { ...products };
    this.state=1;
},
```

启动该函数后，表格中要被修改的那一行的数据内容将被加载到一个结构体变量上，该变量将接收用户的修改值，并实时显示用户当前的输入值。

以下是我们修改输入框的交互设计

```
<div v-if="state===1">  //v表示实时同步，保证当用户点击修改按钮后才显示修改框
    <h3>编辑商品信息</h3>
    <el-form @submit.prevent="updateProductInfo">
        <el-row>
        <el-col :span="8">
        <el-form-item label="商品编号" prop="product_id">
```

```
        <el-input v-model="editedProduct.product_id" type="number" :autosize="{
minRows: 1, maxRows: 4 }" required />       #v-model用于使用户输入值和变量进行绑定
    </el-form-item>
        </el-col>
        <el-col :span="8">
        <el-form-item label="商品名称" prop="product_name">
        <el-input v-model="editedProduct.product_name" type="text" :autosize="{
minRows: 1, maxRows: 4 }" required />
    </el-form-item>
        </el-col>
        <el-col :span="8">
        <el-form-item label="规格" prop="specification">
        <el-input v-model="editedProduct.specification" type="text" :autosize="{
minRows: 1, maxRows: 4 }" required />
    </el-form-item>
        </el-col>
        </el-row>
        <el-row>
        <el-col :span="8">
        <el-form-item label="描述" prop="description">
        <el-input v-model="editedProduct.description" type="text" :autosize="{
minRows: 1, maxRows: 4 }" required />
    </el-form-item>
        </el-col>
        <el-col :span="8">
        <el-form-item label="分类" prop="classification">
        <el-input v-model="editedProduct.classification" type="text" :autosize="
{ minRows: 1, maxRows: 4 }" required />
    </el-form-item>
        </el-col>
        <el-col :span="8">
        <el-form-item label="价格" prop="price">
        <el-input v-model="editedProduct.price" type="number" :autosize="{
minRows: 1, maxRows: 4 }" required />
    </el-form-item>
        </el-col>
        </el-row>
    <el-form-item>
        <el-button color="#626aef" :dark="isDark" type="primary"
@click="updateProductInfo">上传</el-button>
    </el-form-item>
    </el-form>
</div>
```

当用户点击修改按钮后，我们的修改窗口将按下面样式被加载出来。

前端调用后端API的方法

```
updateProductInfo() {
    // 发起更新订单的请求
    axios.patch(`http://172.18.56.143:8000/your_app_name/api/product-
info/${this.editedProduct.product_id}/`, this.editedProduct)
    .then(response => {
        console.log('Success:', response.data);
```

```
            this.loadProductInfo(); // 重新加载订单列表
            this.editedProduct = { // 重置编辑状态
                product_id: null,
                product_name: null,
                specification: null,
                description: null,
                classification: null,
                price: null,
            };
        })
        .catch(error => {
            console.error('Error:', error);
        });
    },
```

当我们把状态修改到我们满意的程度后我们点击按钮启动 `updateProductInfo` 函数实现数据的上传更新，并且最后清空这个存储编辑信息的结构体，方便后续更新其他行的数据。启动该函数后我们会向后端发送修改指令，后端就会接着给数据库发送指令从而实现数据库内容的修改，然后前段再次请求新的数据，重新传输一次数据，我们就可以看到修改后的数据了。

表格的插入功能模块

```
<el-button color="#626aef" :dark="isDark" round @click="addProductInfopre()">注册新商品</el-button>
```

我们在界面上会显示一个可以调用具有插入功能的函数的按钮，点击按钮后启动 `addProductInfopre()` 函数。

```
addProductInfopre() {
    this.ifadd = false;
    if(this.state!==2){
        this.state=2
    }
    else{
        this.state=0
    }
},
```

启动该函数后我们会修改状态值以便修改界面加载出来。

```
<div v-if="state === 2">
    <h3>注册商品信息</h3>
    <el-form @submit.prevent="addProductInfo">
        <el-row>
        <el-col :span="8">
        <el-form-item label="商品编号" prop="product_id">
        <el-input v-model="newProduct.product_id" type="number" :autosize="{
minRows: 1, maxRows: 4 }" required />
    </el-form-item>
        </el-col>
        <el-col :span="8">
        <el-form-item label="商品名称" prop="product_name">
        <el-input v-model="newProduct.product_name" type="text" :autosize="{
minRows: 1, maxRows: 4 }" required />
```

```
    </el-form-item>
        </el-col>
        <el-col :span="8">
        <el-form-item label="规格" prop="specification">
        <el-input v-model="newProduct.specification" type="text" :autosize="{
minRows: 1, maxRows: 4 }" required />
    </el-form-item>
        </el-col>
        </el-row>
        <el-row>
        <el-col :span="8">
    <el-form-item label="描述" prop="description">
        <el-input v-model="newProduct.description" type="text" :autosize="{
minRows: 1, maxRows: 4 }" required />
    </el-form-item>
        </el-col>
        <el-col :span="8">
        <el-form-item label="分类" prop="classification">
        <el-input v-model="newProduct.classification" type="text" :autosize="{
minRows: 1, maxRows: 4 }" required />
    </el-form-item>
        </el-col>
        <el-col :span="8">
        <el-form-item label="价格" prop="price">
        <el-input v-model="newProduct.price" type="number" :autosize="{ minRows:
1, maxRows: 4 }" required />
    </el-form-item>
        </el-col>
        </el-row>
    <el-form-item>
        <el-button color="#626aef" :dark="isDark" type="primary"
@click="addProductInfo">上传</el-button>
        </el-form-item>
    </el-form>
</div>
```

这里状态改变后我们的数据插入界面加载出来。

当我们填完数据并且点击提交后，我们上传信息，并且把填好的数据值赋给一个存储新插入行的结构体里面去。

```
addProductInfo() {
    axios.post('http://172.18.56.143:8000/your_app_name/api/product-info/',
this.newProduct)
    .then(response => {
        console.log('Success:', response.data);
        this.loadProductInfo();
        this.newProduct= {
            product_id: null,
            product_name: null,
            specification: null,
            description: null,
            classification: null,
```

```
            price: null,
        };
    })
    .catch(error => {
        console.error('Error:', error);
    });
},
```

当我们点击按钮后我们启动 `addProductInfo` 函数，实现数据的插入。然后清空存储新插入行的结构体，方便下一次数据插入。启动该函数后我们会向后端发送插入指令，后端就会接着给数据库发送指令从而实现数据库内容的插入，然后再次刷新，重新传输一次数据，我们就可以看到插入后的数据了。

表格的搜索功能模块

这里，我们先在 `style` 里面放一个列表来存放所有可以搜索的词条

```
availableFields: ["product_id", "product_name", "specification", "description",
"classification", "price"],
```

然后我们在前端显示一个可以选择搜索词条的搜索框。

```
<div>
    <el-input v-model="searchKeyword" placeholder="输入关键字进行搜索" style="width:
60%"  />
    <el-select v-model="selectedField" placeholder="请选择搜索字段" style="width:
30%">
        <el-option v-for="field in availableFields" :key="field" :label="field"
:value="field" />
    </el-select>
    <el-button type="primary" @click="searchPurchaseOrders" >搜索</el-button>
</div>
```

当我们写好关键字，选好搜索字段后，我们点击按钮启动搜索功能，调用 `searchPurchaseOrders` 函数。

```
searchPurchaseOrders() {
    axios.get(`http://172.18.56.143:8000/your_app_name/api/product-info/search/?
query=${this.searchKeyword}&field=${this.selectedField}`)
    .then(response => {
        const sortedData = response.data.sort((a, b) => {
            if (new Date(a.purchase_date) < new Date(b.purchase_date)) {
                return -1;
            } else if (new Date(a.purchase_date) > new Date(b.purchase_date)) {
                return 1;
            } else {
                if (a.order_number < b.order_number) {
            return -1;
            } else {
                return 1;
            }
        }
    });
    this.productInformation = sortedData;
    })
```

```
    .catch(error => {
        console.error('Error:', error);
    });
},
```

我们通过指令从约定好的地址那里接收到了搜索回来的数据，并且顺带排了个序，这样子可以保证搜索结果按我们的要求顺序排序。我们实现显示框内容变化的原理是覆盖，通过修改 `productInformation` 的值让前端原本显示所有数据的界面只显示搜索结果。这样我们就可以看到搜索后的数据了。此时点击一下显示全部数据的按钮，我们又会调用一个函数来修改这个列表的值让我们能够重新看到所有数据。

搜索功能界面如下

这样超市管理系统数据库的四个主要功能在前端的实现就讲完了。

9.(软件过程说明)后端设计

后端采用Django框架，主要实现了登录数据库系统和数据库连接，处理来自前端的请求并且返回给指定网址。

首先使用标准操作创建Django项目，接着创建Django应用

我们将以Product_info作为例子展示后端的构建，其他表格也只是大致重复类似的过程

1. 在创建出来的 `models.py` 文件中定义自己的模型，实际上表格的设计就是在这里定型的

```
class Product_info(models.Model):
    product_id = models.IntegerField(primary_key=True, unique=True)
    product_name = models.CharField(max_length=255)
    specification = models.CharField(max_length=255)
    description = models.TextField()
    classification = models.CharField(max_length=50)
    price = models.DecimalField(max_digits=10, decimal_places=2)

    def __str__(self):
        return self.product_name
```

2. 在 `setting.py` 文件夹配置Django连接数据库的配置文件

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'SuperMarket_backsides',
        'USER': 'xxx',
        'PASSWORD': 'xxx',
        'HOST': '172.18.xxx.xxx',
        'PORT': 'xxxx',
    }
}
```

3. 在项目目录下运行以下命令，将模型应用到数据库

```
python manage.py makemigrations
python manage.py migrate
```

至此，数据库方面的表格就建立好了，接下来需要设置操作表格的API接口

4. 首先创建序列化容器，通过json的格式来向前端传递数据

```
from rest_framework import serializers  #利用了DRF中的序列化类
from .models import Product_info,

class ProductInfoSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product_info    #指定模型
        fields = '__all__'  #fields设置为所有属性
```

5. 利用DRF中的视图集类，来提供API

```
class ProductInfoViewSet(viewsets.ModelViewSet):
    queryset = Product_info.objects.all()
    serializer_class = ProductInfoSerializer
    #... 后续可以继续加入自定义的API
```

6. 在 `urls.py` 中借助DRF建立路由，设置路径

```
from rest_framework.routers import DefaultRouter    #
from .views import PurchaseOrderViewSet, SalesOrderViewSet, InventoryViewSet,
ProductInfoViewSet
from django.urls import path, include

router = DefaultRouter()
router.register(r'product-info', ProductInfoViewSet, basename='product-info')

urlpatterns = [
    path('api/', include(router.urls)),
]
```

除了app内的路由，还需要设置项目文件的路由和路径，一切设置完毕之后，就可以使用API来访问后端了。

以上便是我们软件工程化说明文档的全部内容了。