Python 开发框架之

WxPython 桌面端

系统开发及部署说明书

V2.0

序号	修改人	修改日期	修改后版本	修改说明
1	伍华聪	2025-8-13	V1.0	初稿
2	伍华聪	2025-10-11	V2.0	增加 FastAPI 的打包部署方式

目　录

1. 引言

1.1. 背景

《Python 开发框架》是严格的前后端分离模式，该开发框架利用 Python 开发的最新、最广泛的技术，为客户提供最直接、高效的开发帮助。我们多年深耕.NET 开发框架领域，形成有自己独到开发框架思路和丰富的经验，我们把它们拓展到 Python 开发的领域，形成我们的《Python 开发框架》。

框架后端是基于 Python + FastApi + SqlAlchemy + Pydantic 的 Web API 服务，前端可以是基于 Web API 接入的任何前端，目前纯 Python 前端为基于 Python + WxPython 的桌面应用前端，基于 Python + PySide/PyQt 的桌面应用前端，以及基于 BS 的前端 Vue3+ElementPlus+TypeScript 前端。后面会继续完善和扩展其他前端等内容。

《Python 开发框架》使用的是前后端分离模式，因此除了基于常规的跨平台桌面前端（**WxPython 前端框架**）外，还可以接入其他前端，本文档介绍就是 **WxPython 前端框架**的相关开发过程和部署的说明。

1.2. 编写目的

本篇内容主要介绍如何准备 Vue + Element 的前端开发环境，以及项目开发完成后进行产品部署，所需要的发布产品步骤。

1.3. 参考资料

序号	名称	版本/日期	来源
1	《伍华聪的博客》		博客园
1	《伍华聪的 Python 开发随笔》		博客园

1.4. 术语与缩写

1 在本文件中出现的“系统”、“框架”一词，除非特别说明,均适用于《Python开发框架》。

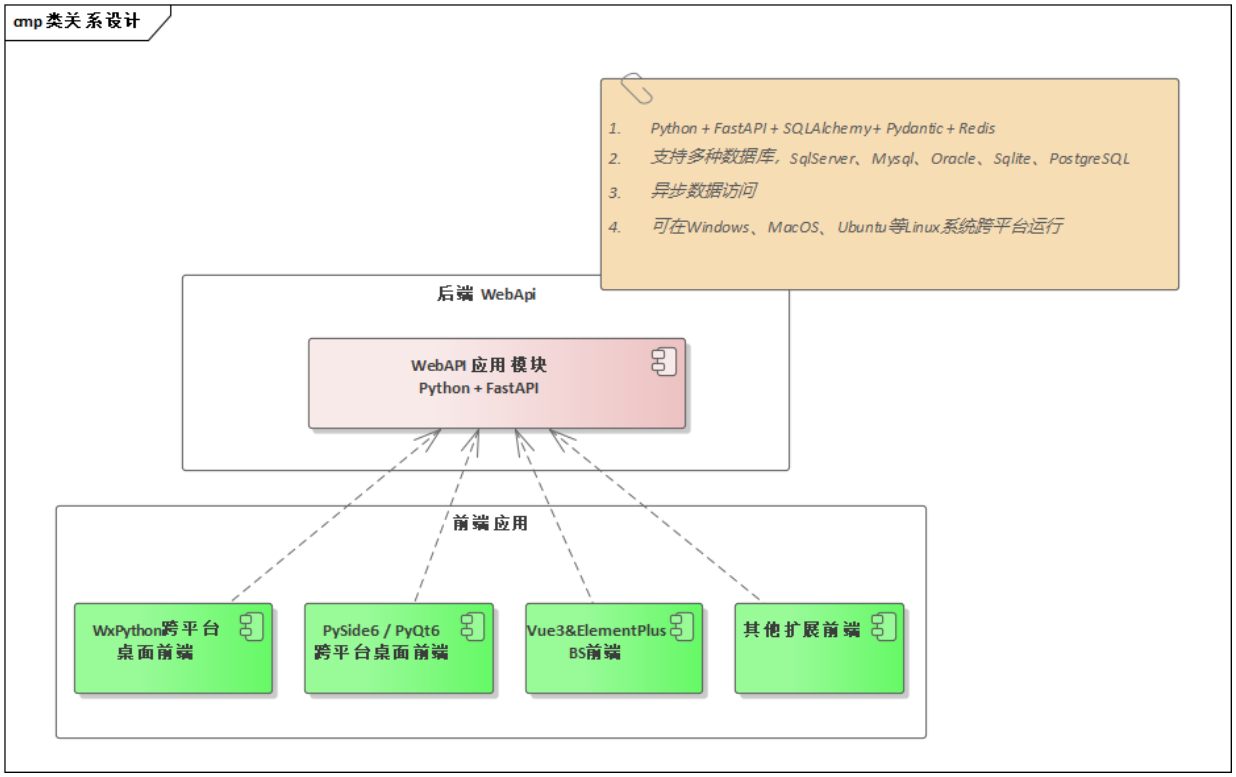
2 当前后端Web API基于Python的FastAPI,如无特殊说明，均指该后端。

2. 框架总体介绍

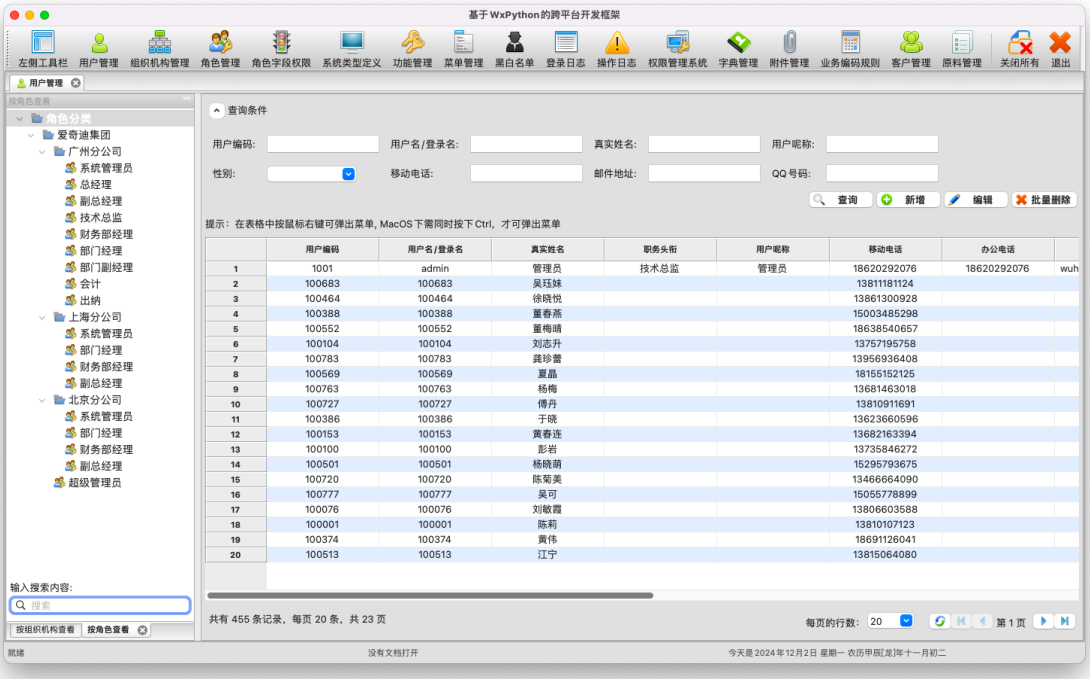
2.1. 总体架构介绍

《Python 开发框架》后端是基于 Python + FastApi 的 WebAPI 服务，前端应用目前有：纯 Python 前端的基于 Python + WxPython 的桌面应用前端，基于 Python + PySide/PyQt 的桌面应用前端，以及基于 BS 应用的 Vue3+ElementPlus+TypeScript 前端。

我们提供一些 UML 图示帮助了解架构和组件设计的细节。



2.2. 基于 Python + WxPython 桌面端特点



1、 基础功能完善，可重用、快速开发。基于我们的成熟框架，完善了用户管理、组织机构管理、角色管理、菜单管理、功能管理及权限分配，日志管理、字典管理、附件管理等管理功能，可实现用户功能权限、字段权限、数据权限的控制管理。

2、 集成通用字典管理模块，方便对字典大类及对应字典项目数据的管理。以及实现系统参数的维护管理，参数配置管理界面等。

3、 菜单动态展示和功能权限控制，集成菜单权限控制及全局菜单的显示及禁用，对界面按钮权限整合功能权限控制，对模块的字段权限进行设置可以控制修改、可见等。

4、 Python 开发框架后端基于 Python 开发，WebAPI 基于 Restful 接口，使用 FastAPI, SQLAlchemy, Pydantic, Pydantic-settings, Redis, JWT 构建的项目，异步处理数据库调用。

5、 前后端分离模式，可接入多种终端，具有良好的整合性。

6、 框架支持多数据库，可支持 Mysql、SqlServer、Postgresql、Sqlite、Oracle 等多种数据库接入，后端通过配置即可指定接入的数据库类型。

7、 通过代码生成工具 Database2sharp 配套工具，实现业务模块的快速开发，可以快速生成框架前后端代码，以及各个桌面端、Vue3 的 BS 端界面代码，可根据情况微调布局即可。

8、 生成的后端 Web API 代码，包括数据库模型层、DTO 层、数据访问层、Web API 控制

器层等；前端生成包含 Web API 的封装层、实体层、列表界面、编辑界面等内容，生成界面功能默认具有增删改查、分页、导出、打印、树列表展示等功能。

9、纯 Python 的跨平台开发框架，使用 VSCode 实现跨平台开发；跨平台运行，可在 Windows、MacOS、Ubuntu 等 Linux 系统上运行，实现多端界面效果一致的应用系统。

开发框架具有良好的抽象封装基类，数据访问层、API 控制器层、API 调用层、UI 层包括列表和编辑界面等，采用基类继承的方式减少重复代码，提高代码复用性。

2.3. 基于 Python +FastAPI 后端 WebAPI 特点

FastAPI 是一个用于构建 API 的现代、快速（高性能）web 框架，基于 Python 类型提示。它的主要特点包括自动生成 OpenAPI 和 JSON Schema 文档、快速代码编写、简洁的代码结构、高效的性能等。FastAPI 使用 Starlette 作为 Web 框架的核心，并使用 Pydantic 进行数据验证。

FastAPI 的主要特点

1. **快速：**
    - FastAPI 的性能非常接近于 NodeJS 和 Go 等速度较快的语言，并且比其他基于 Python 的框架如 Flask 和 Django 快得多。
2. **简洁：**
    - 通过类型提示和依赖注入，代码简洁易读。
    - 开发者可以更少的代码实现更多的功能。
3. **自动文档生成：**
    - FastAPI 自动生成符合 OpenAPI 规范的文档，这些文档可以通过内置的 Swagger UI 和 ReDoc UI 查看。
    - 自动生成 JSON Schema。
4. **数据验证：**
    - 基于 Pydantic，FastAPI 提供了强大的数据验证功能。
    - 支持复杂的数据验证和数据解析。
5. **类型提示：**
    - 充分利用 Python 的类型提示，帮助开发者编写和维护代码。

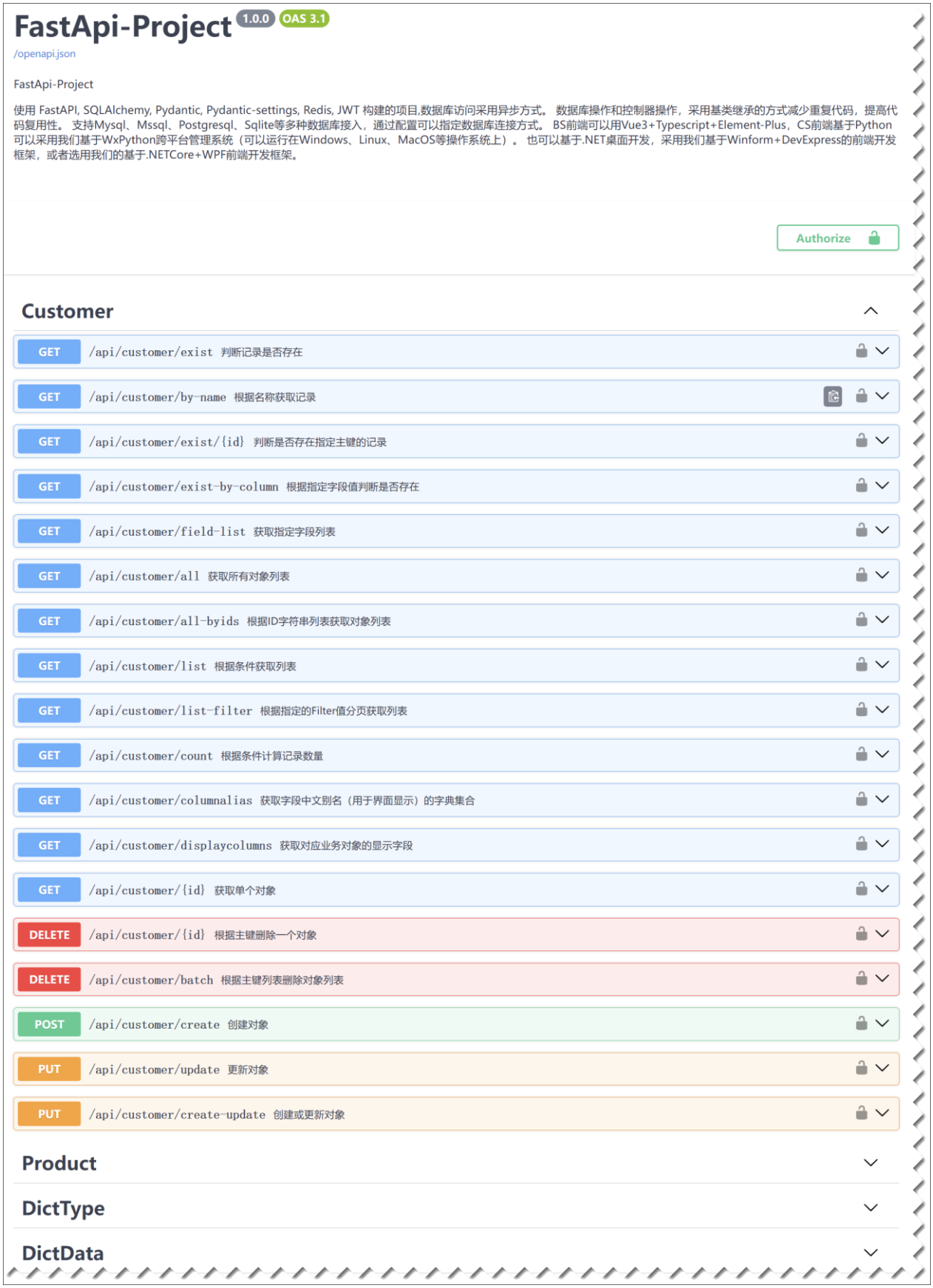
6. **依赖注入：**
    - FastAPI 提供了一个简单但功能强大的依赖注入系统，可以方便地管理依赖项。

FastAPI 还支持以下功能：

- 文件上传
- 安全性（OAuth2、JWT 等）
- 后台任务
- 流媒体响应
- GraphQL
- SQL（通过 SQLAlchemy 等）
- 数据库事务
- 后台任务

Python 开发框架后端基于 Python 开发，WebAPI 基于 Restful 接口，使用 FastAPI, SQLAlchemy, Pydantic, Pydantic-settings, Redis, JWT 构建的项目，异步处理数据库调用。

FastAPI 项目运行后，其界面自动整合 Swagger 的文档界面，如下所示。


FastApi-Project
1.0.0
OAS 3.1
/openapi.json
FastApi-Project
使用 FastAPI, SQLAlchemy, Pydantic, Pydantic-settings, Redis, JWT 构建的项目,数据库访问采用异步方式。 数据库操作和控制器操作，采用基类继承的方式减少重复代码，提高代码复用性。 支持Mysql、Mssql、Postgresql、Sqlite等多种数据库接入，通过配置可以指定数据库连接方式。 BS前端可以用Vue3+Typescript+Element-Plus，CS前端基于Python可以采用我们基于WxPython跨平台管理系统（可以运行在Windows、Linux、MacOS等操作系统上）。 也可以基于.NET桌面开发，采用我们基于Winform+DevExpress的前端开发框架，或者选用我们的基于.NETCore+WPF前端开发框架。
Authorize
Customer
GET /api/customer/exist 判断记录是否存在
GET /api/customer/by-name 根据名称获取记录
GET /api/customer/exist/{id} 判断是否存在指定主键的记录
GET /api/customer/exist-by-column 根据指定字段值判断是否存在
GET /api/customer/field-list 获取指定字段列表
GET /api/customer/all 获取所有对象列表
GET /api/customer/all-byids 根据ID字符串列表获取对象列表
GET /api/customer/list 根据条件获取列表
GET /api/customer/list-filter 根据指定的Filter值分页获取列表
GET /api/customer/count 根据条件计算记录数量
GET /api/customer/columnalias 获取字段中文别名（用于界面显示）的字典集合
GET /api/customer/displaycolumns 获取对应业务对象的显示字段
GET /api/customer/{id} 获取单个对象
DELETE /api/customer/{id} 根据主键删除一个对象
DELETE /api/customer/batch 根据主键列表删除对象列表
POST /api/customer/create 创建对象
PUT /api/customer/update 更新对象
PUT /api/customer/create-update 创建或更新对象
Product
DictType
DictData

3. WxPython 前端项目开发

纯前端的开发，一般不会再采用笨重的 VS 进行前端的开发，而改用 VS Code 或者 WebStorm 等其他轻型的开发工具来进行前端代码的开发和维护，虽然是轻型开发工具，不过功能也是非常强大的，而且开发环境可以在 Windows 系统，也可以在 Mac 系统等，实现多平台的开发环境。

本项目是基于 wxPython 开发的 GUI 应用，系统主要功能包括用户、角色、机构、权限、日志、菜单、字典、附件、通用编码规则、参数配置管理等基础框架内容。

我们通过利用其各种界面控件，结合 Python+WxPython 的跨平台运行的特性，为 Windows、MacOS、Ubuntu 等 Linux 系统，开发一套界面效果一致的桌面端应用系统。

我们可以基于 VSCode + wxpython + WebApi 组合实现桌面端的开发，并利用代码生成工具，快速实现前后端功能和界面代码的开发。

VSCode 是一个高效、快速的代码编辑器，启动速度快，不会占用过多资源，VSCode 支持 Windows、Linux 和 macOS，且在各平台上有一致的用户体验。

WxPython 是一个跨平台的 GUI 库，基于原生平台的控件实现，因此应用程序的界面与操作系统的原生应用界面高度一致，提升用户体验。 WxPython 支持 Windows、Linux 和 macOS，可以编写一次代码，并在多个操作系统上运行。 WxPython 提供了大量的控件，如按钮、文本框、列表框、树形控件、菜单、工具栏等，可以满足桌面应用大部分需求。

3.1. VS code 的安装

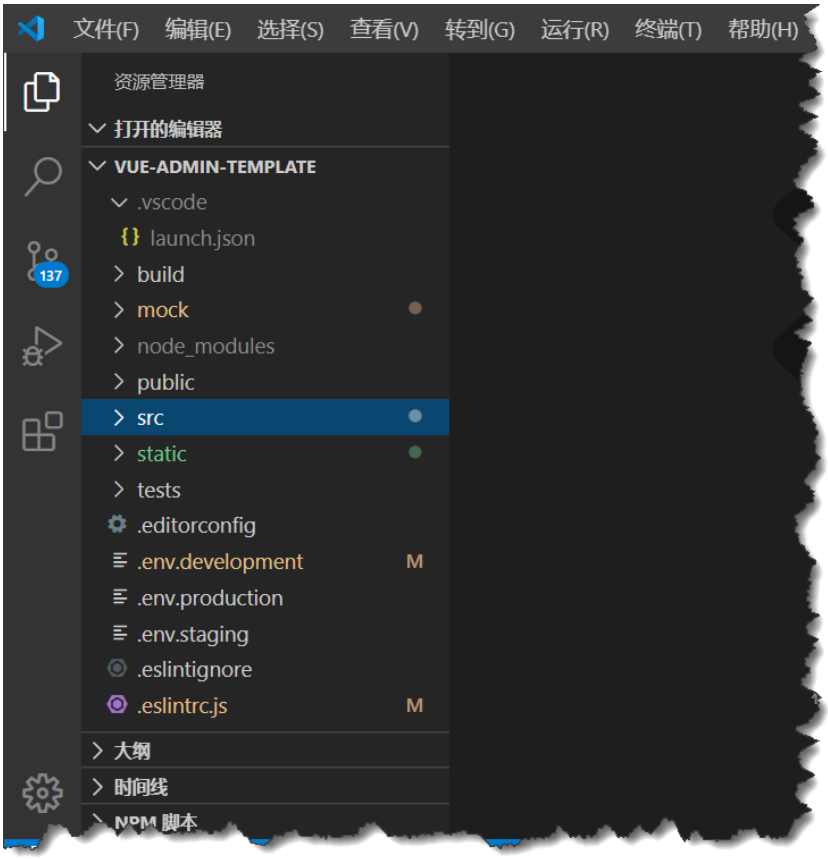
VS Code（Visual Studio Code）是由微软研发的一款免费、开源的跨平台文本（代码）编辑器。这是一个轻量级但功能强大的代码编辑器，支持丰富的扩展。你可以从 Visual Studio Code 官方网站 下载。

官网：https://code.visualstudio.com

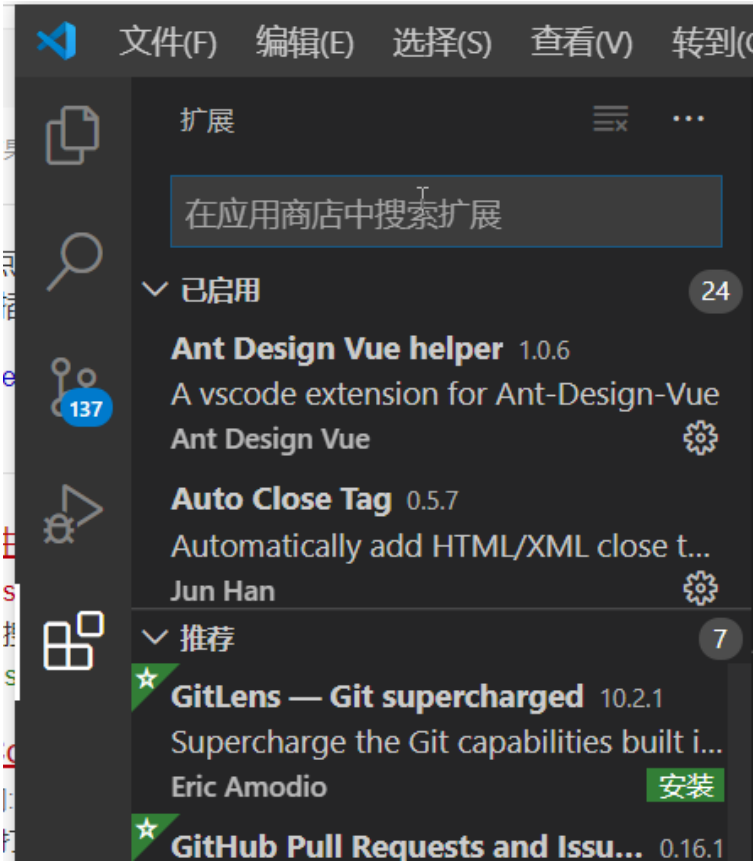
文档：https://code.visualstudio.com/docs

源码：https://github.com/Microsoft/vscode

VS Code 的界面大概如下所示，一般安装后，如果为英文界面，则安装它的中文包即可。



VS Code 安装后，我们一般还需要搜索安装一些所需要的插件辅助开发。安装插件很简单，在搜索面板中查找到后，直接安装即可。



一般我们需要安装这些 vs code 插件：

AI 编程助手，下面任选其一即可：

1、Fitten Code：

它是由非十大模型驱动的 AI 编程助手，它可以自动生成代码，提升开发效率，帮您调试 Bug，节省您的时间，另外还可以对话聊天，解决您编程碰到的问题。

Fitten Code 是由非十大模型驱动的 AI 编程助手，它可以自动生成代码，提升开发效率，帮您调试 Bug，节省您的时间。还可以对话聊天，解决您编程碰到的问题。免费且支持 80 多种语言：Python、C++、Javascript、Typescript、Java 等。

强烈推荐使用，自动补齐代码功能，可以节省很多手工键入代码的时间，减少错误。

2、GitHub Copilot：

GitHub Copilot 主要作为 Visual Studio Code（VS Code）的插件提供。你可以从 VS Code 的插件市场下载并安装它。最新的 Copilot 已经可以免费试用。

GitHub Copilot 是一种由 GitHub 与 OpenAI 合作开发的人工智能编程助手。它使用 GPT-3 或更高版本的语言模型，旨在帮助开发人员编写代码、提供代码片段、自动生成注释和文档，甚至根据上下文和已有代码提供建议。它集成在多个开发环境中，尤其是在 Visual Studio Code 中，作为插件使用。

主要功能：

1. **自动补全代码：** GitHub Copilot 会根据你正在编写的代码提供实时建议，自动完成代码行或函数。它可以根据函数名称、变量类型或你前面的代码片段来推测并生成合理的后续代码。
2. **生成代码片段和函数：** 你可以输入一个简短的注释或描述（例如："函数计算斐波那契数列"），然后 Copilot 会生成一个完整的代码实现。
3. **智能提示和补充：** Copilot 会根据上下文理解你的意图，提供相关的代码建议。它不仅能在你编写代码时给出补全建议，还能在你不确定如何开始时帮助你生成框架代码。
4. **跨语言支持：** GitHub Copilot 支持多种编程语言，包括 Python、JavaScript、TypeScript、Go、C++ 等。它能够根据你正在使用的编程语言提供相应的建议。
5. **重构代码和优化：** 在一些情况下，Copilot 能够识别出代码中的重复部分，提供优化或重构建议，帮助你改进代码质量。

3.2. 安装 Visual Studio Build Tools

Python 开发的时候，往往还需要安装 Visual Studio Build Tools，用来构建编译环境，因此需要在微软官方下载：https://visualstudio.microsoft.com/visual-cpp-build-tools/ ，安装时选择：

- **C++ Build Tools**
- 勾选 **Windows 10/11 SDK**

这样可以避免在安装依赖包或者编译 Python 项目的时候，提示缺少 wheel 等内容。

3.3. 配置 Python 开发环境

利用 VS Code 开发，我们很多时候，需要使用一些开发插件来方便开发各种类型的项目，Python 开发也一样。

在 VSCode 中安装 Python 扩展，在扩展市场搜索 Python 并安装。

下载 Python 3 安装包并安装。

Window 平台安装 Python: https://www.python.org/downloads/windows

Mac 平台安装 Python: https://www.python.org/downloads/mac-osx

配置 Python 环境变量：

打开系统环境变量，在 PATH 变量中添加 Python 目录。

Windows 平台：

右键“我的电脑” -> 属性 -> 高级系统设置 -> 环境变量 -> 系统变量 -> Path -> 编辑 -> 新建 -> 输入 Python 目录路径 -> 确定 -> 确定 -> 确定

Mac 平台：

打开终端，输入以下命令：

```
echo 'export PATH="/usr/local/bin:$PATH"' >> ~/.bash_profile
source ~/.bash_profile
```

测试 Python 环境

在命令行中输入 python，如果出现 Python 解释器版本信息，则表示 Python 环境配置成功。

安装 pip

打开命令行，输入 pip install --upgrade pip，升级 pip 到最新版本。

创建虚拟环境

在 Python 3.3 及以上版本中，venv 是内置模块，使用命令创建虚拟环境。

```
python -m venv myenv
```

激活虚拟环境

在 macOS/Linux，输入 source myenv/bin/activate，激活虚拟环境。

在 Windows，输入 myenv\Scripts\activate，激活虚拟环境。或进入虚拟环境目录 myenv\Scripts，输入 .\activate，激活虚拟环境。

如果无法进入虚拟环境，Windows 运行命令：

Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass

然后再次输入 myenv\Scripts\activate.bat，激活虚拟环境。

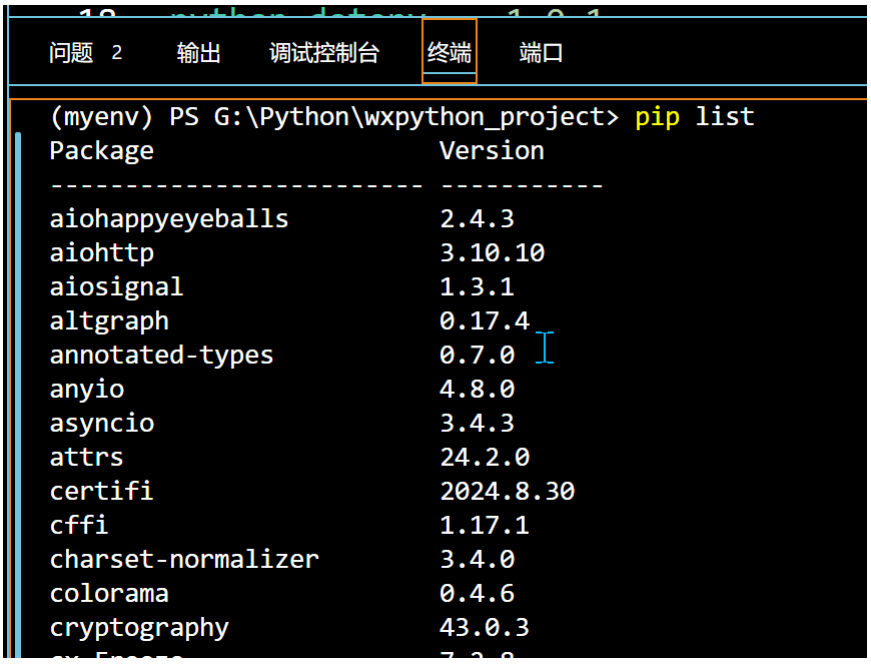
安装 Python 项目依赖库

requirements.txt 是项目的相关依赖库列表，确保 requirements.txt 文件在当前目录，然后运行以下命令：

在虚拟环境中，输入 pip install -r requirements.txt 安装项目依赖库。

在虚拟环境中，输入 pip install 库名称可以安装额外的类库，如：pip install requests 安装 requests 库。

如果需要查看已安装的项目类库，可以通过命令：pip list 进行查看，如下图所示。



3.4. 编译运行项目代码

使用 VSCode 打开 Python 项目。确保已经按照上面配置了 Python 开发环境，并把当前

控制台已经切换到虚拟状态中。

首次使用需要设置 VSCode 的 Python 项目解析器，在 VSCode 菜单的【查看】【命令面板】，选择 Python 解析器为当前虚拟环境的 Python 解析器即可。

在 Python 项目里，解析器（Python Interpreter）一般应该选择 **当前虚拟环境** 的，而不是全局的，原因是：

1. **依赖隔离**
   - 全局解析器使用的是系统 Python 环境，安装的库是共享的，容易导致不同项目之间依赖冲突。
   - 虚拟环境为项目单独提供一份 site-packages，不会影响其他项目。
2. **版本一致性**
   - 项目可能要求特定版本的 Python（比如 3.9、3.10），而全局环境可能不匹配。
   - 虚拟环境可以用精确的 Python 版本来运行和测试。
3. **可移植性**
   - 当别人拉取项目代码（比如用 requirements.txt 或 pyproject.toml）时，他们也会用对应的虚拟环境，确保运行结果一致。
4. **IDE 配合**
   - 如果你用 PyCharm、VS Code 等 IDE，选虚拟环境作为解析器能让智能提示、调试路径、代码运行环境和依赖保持一致。

设置好 Python 解析器后（首次设置），然后定位到项目源码 app/main.py 中，然后单击 VSCode 的右上侧，【运行 Python 文件】即可启动项目。

WxPython 的项目启动后的截图如下所示。

3.5. 项目目录结构

WxPython 的项目目录结构如下所示。

app/: 项目的主目录，包含所有应用相关代码。

main.py: 项目的入口文件。

.env: 环境变量文件，用于存储敏感信息，如数据库连接字符串。

README.md: 项目说明文件。

requirements.txt: 项目依赖列表。 #使用命令生成：pip freeze > requirements.txt

api/: 用于处理客户端 API 请求, 用于管理用户、角色、机构、权限等。

user.py : 用户 API，用于管理用户信息。

role.py : 角色 API，用于管理角色信息。

ou.py : 机构 API，用于管理机构信息。

menu.py : 菜单 API，用于管理菜单信息。

...

controls/: 自定义控件库， 用于扩展系统功能，包括文本框、数值框、日期选择、下拉框、树形控件、表格控件等。

my_textctrl.py : 文本框控件。

my_comobox.py : 下拉框控件。

my_dialog.py : 对话框控件。

core/: 核心功能，如配置、安全等。

config.py: 项目的配置信息，包括数据库连接信息、日志配置等。

system_app.py ： 系统应用，用于管理系统登录，以及获取用户相关信息，如用户、角色、机构、权限等。

system_splash_screen.py : 系统启动画面，用于显示启动信息。

system_taskbar_icon.py : 系统任务栏图标，用于显示系统状态。

core_images.py : 系统图片资源，用于显示系统图标等。

...

entity/: 数据模型，用于请求和响应的数据模型。

user.py : 用户实体。

role.py : 角色实体。

ou.py : 机构实体。

...

utils/: 工具函数和公用模块。

message_util.py : 用于处理系统消息。

filedialog_util.py : 用于处理文件选择对话框。

...

views/: 视图层，用于展现各个模块的界面，包括列表界面和编辑界面等。

frm_user.py : 用户视图。

frm_user_edit.py : 用户编辑视图。

...

images/: 图片资源。

myenv/: 虚拟环境目录，用于隔离项目依赖库。

logs/:生成日志文件。

static/: 静态文件。

3.6. 使用 PyInstaller 进行打包处理

使用 PyInstaller 打包 Python 项目是一个常见的需求，它可以将 Python 程序及其所有依赖项打包成一个独立的可执行文件或者安装文件，方便在没有安装 Python 环境的机器上运行。本随笔介绍 WxPython 跨平台开发框架中使用 PyInstaller 进行打包处理，包括在 WIndow 平台下生成独立的 exe 文件，松散结构的 exe 文件和目录，以及在 MacOS 上生成安装包的处理过程。

3.6.1. 安装和使用 PyInstaller 进行程序打包处理

PyInstaller 是目前最流行的 Python 打包工具之一。它可以将 Python 脚本打包成独立的可执行文件，支持 Windows、Linux 和 macOS 平台。

PyInstaller 有丰富的文档，提供了详细的使用说明和常见问题解答，你可以通过以下链接访问：

- PyInstaller 官方文档：https://pyinstaller.readthedocs.io
- GitHub 代码库：https://github.com/pyinstaller/pyinstaller

这些文档和资源能帮助你深入了解 PyInstaller 的使用方式，并解决在打包过程中可能遇到的问题。

打包后的可执行文件可以在没有 Python 环境的机器上运行。PyInstaller 会自动分析程序的依赖关系，并将所有必要的库和资源打包到一个文件或者一个文件夹中。

cx_Freeze 是另一个常用的 Python 打包工具，同样支持跨平台。

Py2exe 是专门用于将 Python 脚本打包成 Windows 平台可执行文件的工具。

首先，确保您的 Python 环境已经安装。然后，通过 pip 安装 PyInstaller。在 VSCode 或者其他编辑器的命令行（终端）中执行以下命令：

pip install pyinstaller

在项目目录下，使用 PyInstaller 命令来打包您的 Python 脚本。假设您的主程序文件名为 main.py，则可以使用以下命令：

pyinstaller --onefile main.py

这里的--onefile 选项指示 PyInstaller 生成一个单独的可执行文件，而不是一个包含多个文件的文件夹。PyInstaller 还支持许多其他选项，如--icon 来指定应用程序的图标，--windowed 或--noconsole 来避免在 Windows 上打开命令行窗口等。

PyInstaller 完成打包后，会在 dist 目录下生成可执行文件（或文件夹，如果您没有使用 --onefile 选项）。进入 dist 目录，您应该能看到一个名为 main（或您指定的名称，如果使用了--name 选项）的可执行文件。

如果一切正常，您的 Python 程序现在应该会在没有 Python 环境的情况下运行。

使用 PyInstaller 打包 Python 项目是一个简单而强大的方法，可以让您的程序更加便携和易于分发。通过遵循上述步骤，您可以轻松地将您的 Python 项目打包成一个可执行文件。

3.6.2. 使用.spec 文件进行定制打包处理

打包过程中，PyInstaller 会生成一个 .spec 文件。这个文件包含了 PyInstaller 的配置信息，其中包含了构建过程的所有配置信息。你可以修改这个文件来定制打包过程。

本项目提供了两个.spec 文件（main_my.spec、 main_mac.spec），分别为 Windows 和 MacOS 下的打包文件。main_my.spec 为 Windows 下的打包 exe 文件配置，main_mac.spec 为对应 macOS 下的打包 exe 文件配置。

你可以手动修改 .spec 文件来添加资源文件、修改导入模块、定制输出路径等。

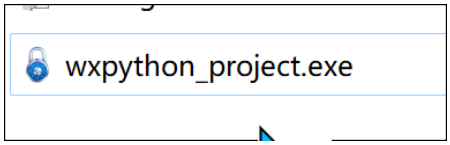
你可以通过编辑.spec 文件，在 EXE、COLLECT 和 BUNDLE 块下添加一个 name= ，为 *PyInstaller* 提供一个更好的名字，以便为应用程序（和 dist 文件夹）使用。

EXE 下的名字是*可执行文件*的名字，BUNDLE 下的名字是应用程序包的名字。

修改完成后，执行以下命令来重新打包：

pyinstaller main_my.spec

最后在 dist 目录上生成一个文件，如下所示。

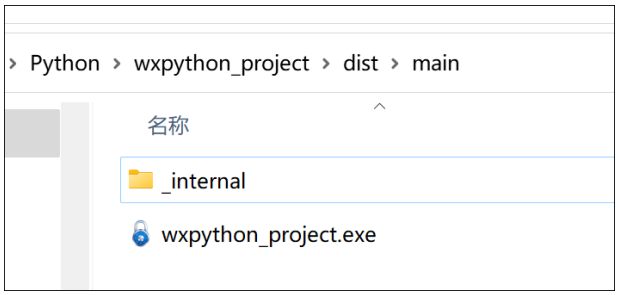


本文件用于 Window 平台下打包整个项目，生成一个独立的 exe 文件，包含了所有的依赖和资源文件，文件运行后会在临时目录中解压出来一个文件夹，程序退出后，该文件夹自动移除。

临时目录一般在 C:\Users\Administrator\AppData\Local\Temp。

如果我们想在 Windows 平台生成的 dist 目录中生成一个启动 exe，和其他相关的 Lib 依赖库目录，那么我们可以适当调整下.spec 文件，让它可以生成松散结构的文件目录包。项目提供了另外一个松散结构的打包配置文件 main_my_dir.spec

相当于之前在 exe 包中的 a.binaries 和 a.datas 从 EXE 构造函数中移到了 Collect 的构造函数里面了。这样会生成下面的目录结构。其中_internal 目录包含程序的相关依赖包和文件资源。



3.7. WxPython 前端项目代码生成

在经过对 WxPython 的深入研究后，并依据改造过的项目结构，整合在代码生成工具中，对项目的代码，包括列表界面，编辑界面、API 调用类、DTO 实体信息等进行统一的生成。

代码生成工具可以到地址下载：https://www.iqidi.com/database2sharp.htm，使用代码生成工具来快速开发项目代码，有很多好处。

减少重复工作：自动生成常用代码（如数据模型、CRUD 操作、API 接口等），减少手动编写的时间。

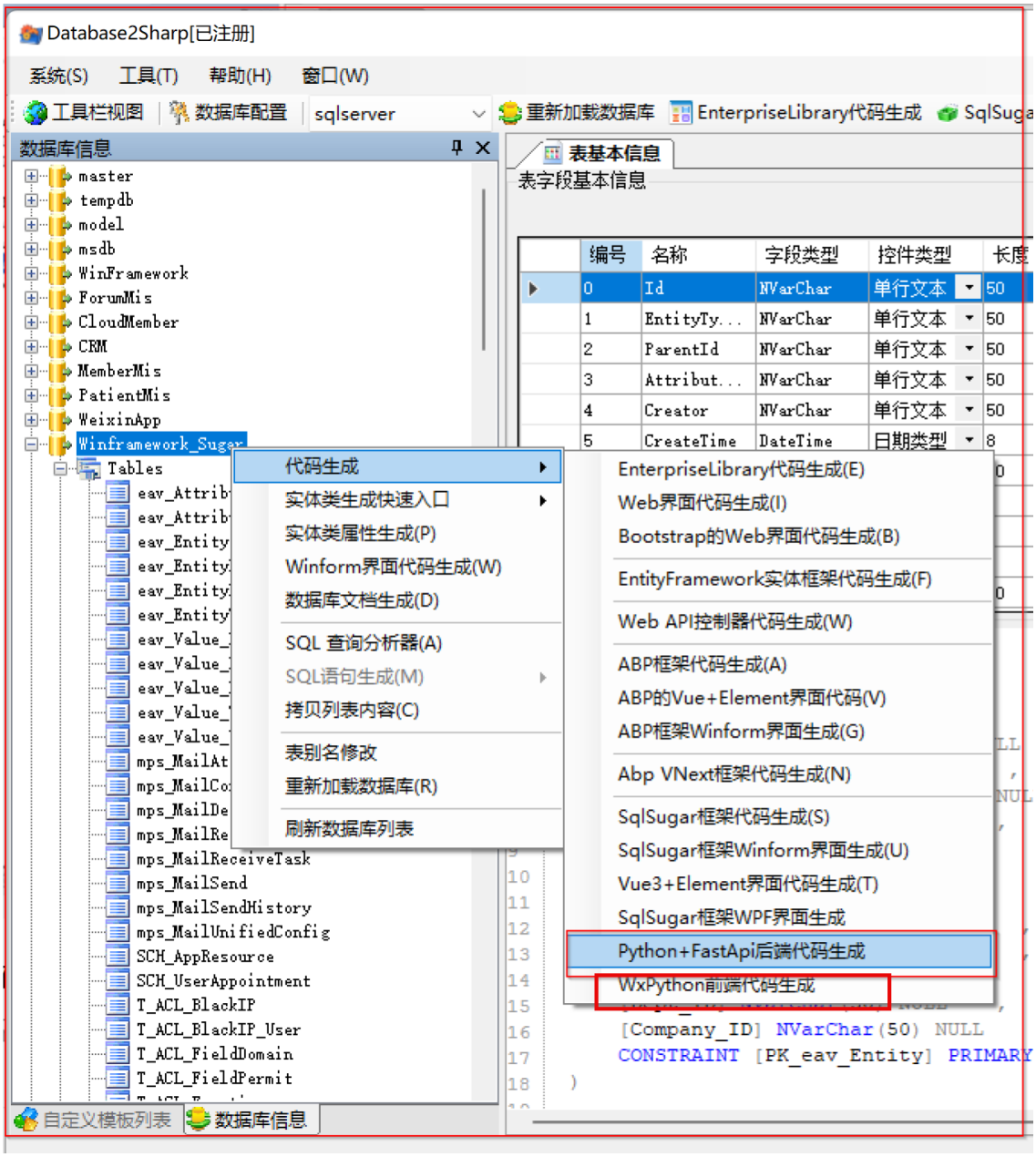
专注于核心逻辑：开发者可以将时间集中在业务逻辑和复杂问题的解决上，而非基础代

码的编写。

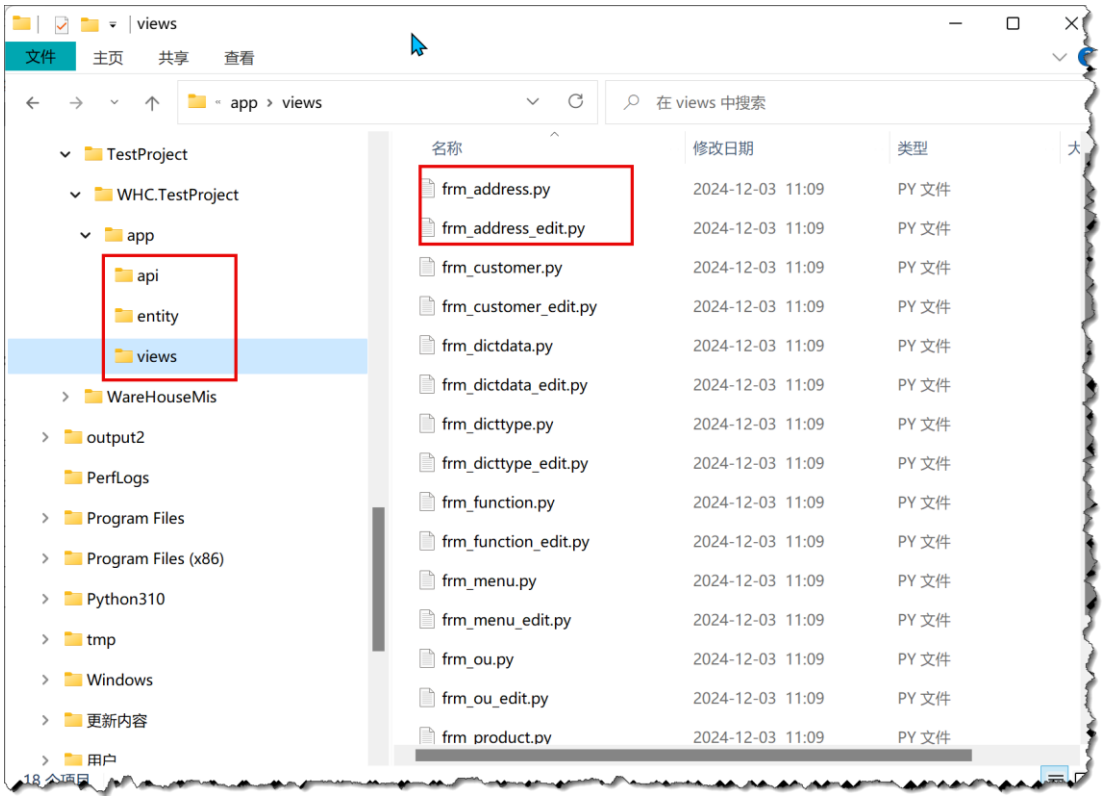
模块化和规范化：生成代码一般遵循既定的架构和风格，便于后续维护和扩展。

初学者友好：新手开发者可以通过代码生成工具快速上手，了解项目结构和基础代码。

代码生成工具在提升开发效率、降低出错率、标准化代码方面具有显著优势，尤其在重复性工作较多或团队合作时尤为适用。

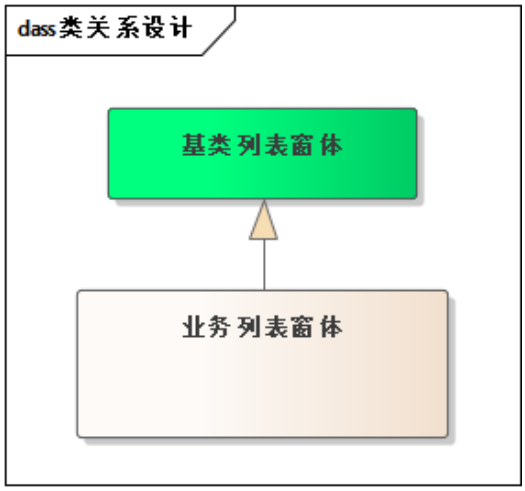


选择相关的数据表后，一键生成相关的代码，如下所示。

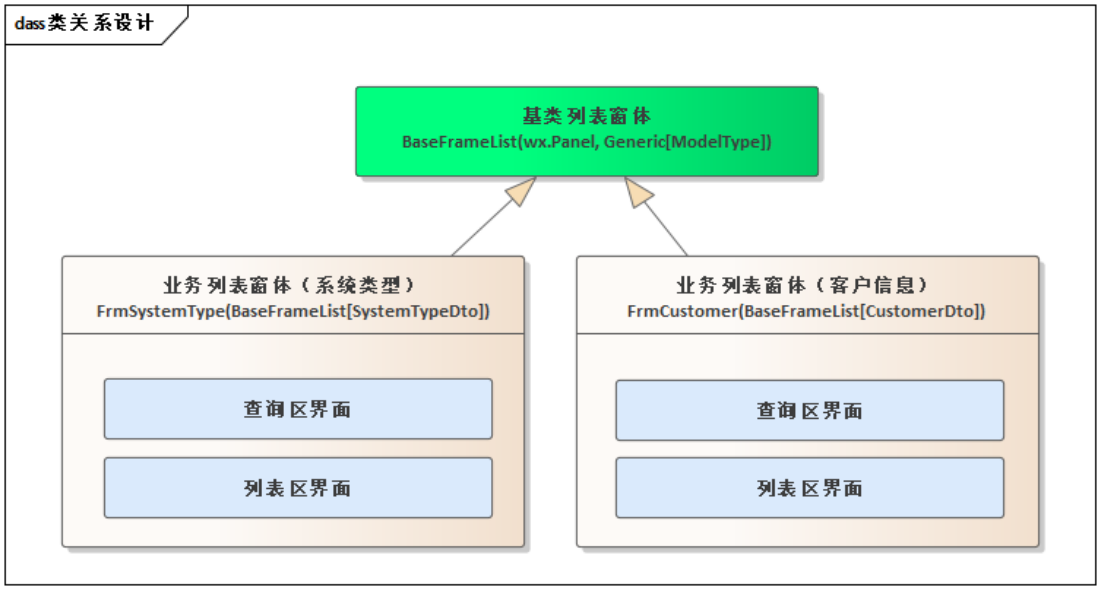


3.7.1. 列表界面和继承关系

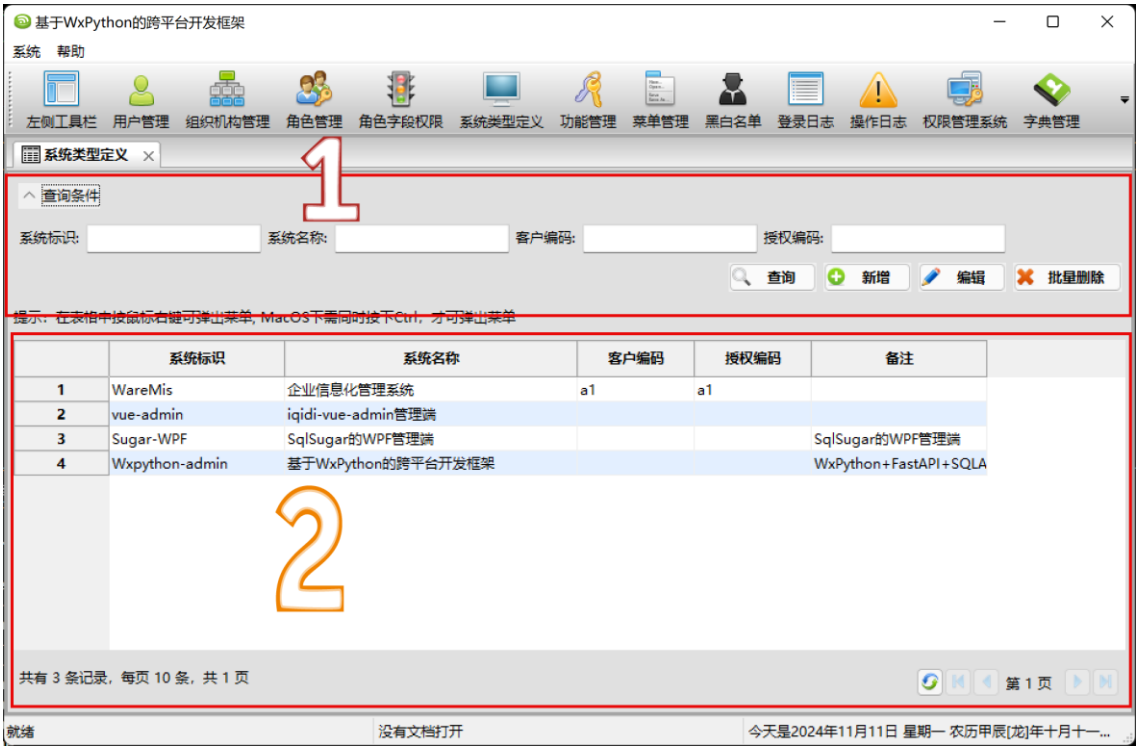
列表界面继承基类，从而可以大幅度的利用相应的规则和实现。



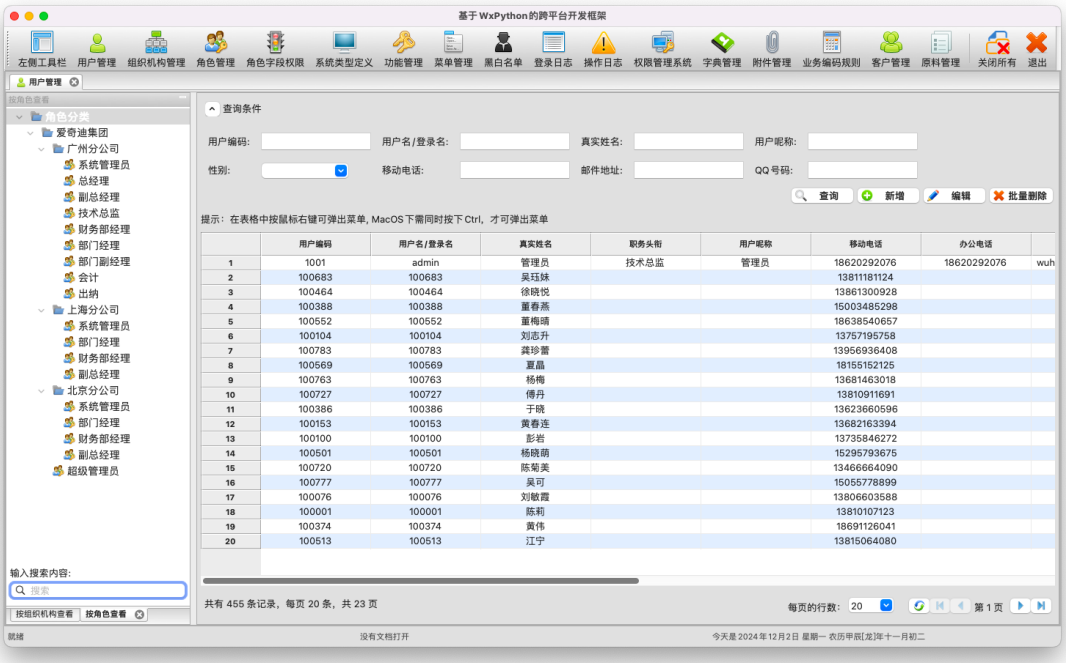
如对于两个例子窗体：系统类型定义，客户信息，其中传如对应的 DTO 信息和参数即可。



因为常规的列表界面一般分为查询区、列表界面展示区和分页信息区，我们把它分为两个主要的部分，如下界面所示。



当然如果有树形列表的，也整合在基类窗体中实现控制逻辑，具体实现放在子类处理即可。



同时，在树列表或者表格数据控件支持右键弹出菜单处理，包括常规的新增、编辑、删除、复制、刷新等常规功能，如果需要更多业务模块的功能，整合在右键菜单中，在窗体子类中重写某些只定义函数即可实现。

对于列表界面，生成的代码如下所示（以客户信息表为例）：

```python
# 继承BaseFrameList类，并传入实体类CustomerInfo，作为泛型类型
class FrmCustomer(ctrl.BaseListFrame[CustomerDto]):
    """客户信息"""

    # 显示的字段名称，逗号分隔
    display_columns = "id,name,age,creator,createtime"
    # 列名映射（字段名到显示名的映射）
    column_mapping = { "id": "编号", "name": "姓名", "age": "年龄", "creator": "创建人", "createtime": "创建时间" }

    def __init__(self, parent, **kwargs):
        # 初始化基类信息
        super().__init__(
            parent,
            model=CustomerDto,
            display_columns=self.display_columns,
            column_mapping=self.column_mapping,
            use_left_panel=False,  # 是否使用树面板
        )

    async def init_dict_items(self):…

    def CreateConditions(self, pane: wx.Window) -> List[wx.Window]:…

    async def OnQuery(self):…

    async def OnAdd(self, event: wx.Event) -> None:…

    async def OnEditById(self, entity_id: Any | str):…

    async def OnDeleteByIdList(self, id_list: List[Any | str]):…
```

代码工具最大程度的提供常规方法的处理，如果需要特殊的操作，如在查询框中的条件，那么需要根据需要修改一下即可。

```python
    def CreateConditions(self, pane: wx.Window) -> List[wx.Window]:
        """创建折叠面板中的查询条件输入框控件"""
        # 创建控件，不用管布局，交给CreateConditionsWithSizer控制逻辑
        # 默认的FlexGridSizer为4*2=8列，每列间隔5px
        self.txtName = ctrl.MyTextCtrl(pane)
        self.txtAge = ctrl.MyNumericRange(pane)
        self.txtCustomerType = ctrl.MyComboBox(pane, style=wx.CB_READONLY)

        util = ControlUtil(pane)
        util.add_control("姓名:", self.txtName)
        util.add_control("年龄:", self.txtAge)
        util.add_control("客户类型:", self.txtCustomerType)  # 测试数据类型绑定

        return util.get_controls()
```

基本上就是不变的内容和规则，由基类处理，变化的内容，由子类来具体化即可。对于新增、编辑、删除的操作，我们根据表的不同，生成子类实现代码，一般不用修改。

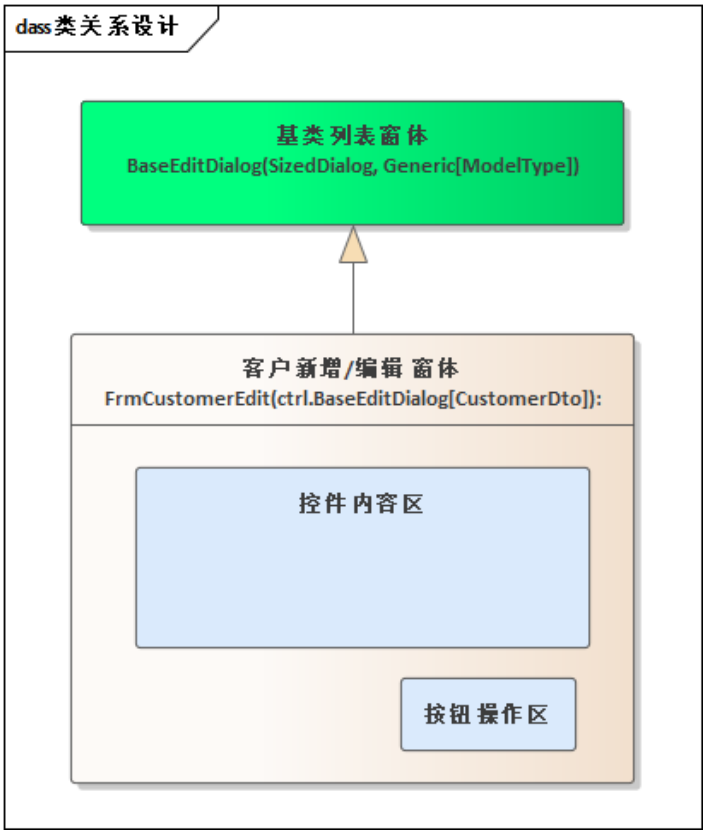
```
async def OnAdd(self, event: wx.Event) -> None:
    """子类重写-打开新增对话框"""
    dlg = FrmCustomerEdit(self)
    if await AsyncShowDialogModal(dlg) == wx.ID_OK:
        # 新增成功，刷新表格
        await self.update_grid()
    dlg.Destroy()
```

如果需要再查询框中初始化下拉列表的内容，我们重写初始化字典函数即可。

```
    async def init_dict_items(self):
        """初始化字典数据-子类重写"""
        await self.txtCustomerType.bind_dictType("客户类型")
```

通过上面我们构建的基类处理，以及提供一个界面辅助类来处理，可简化很多不必要的代码，而且还很灵活的控制布局处理，非常方便。

3.7.2. 编辑/新增界面继承关系



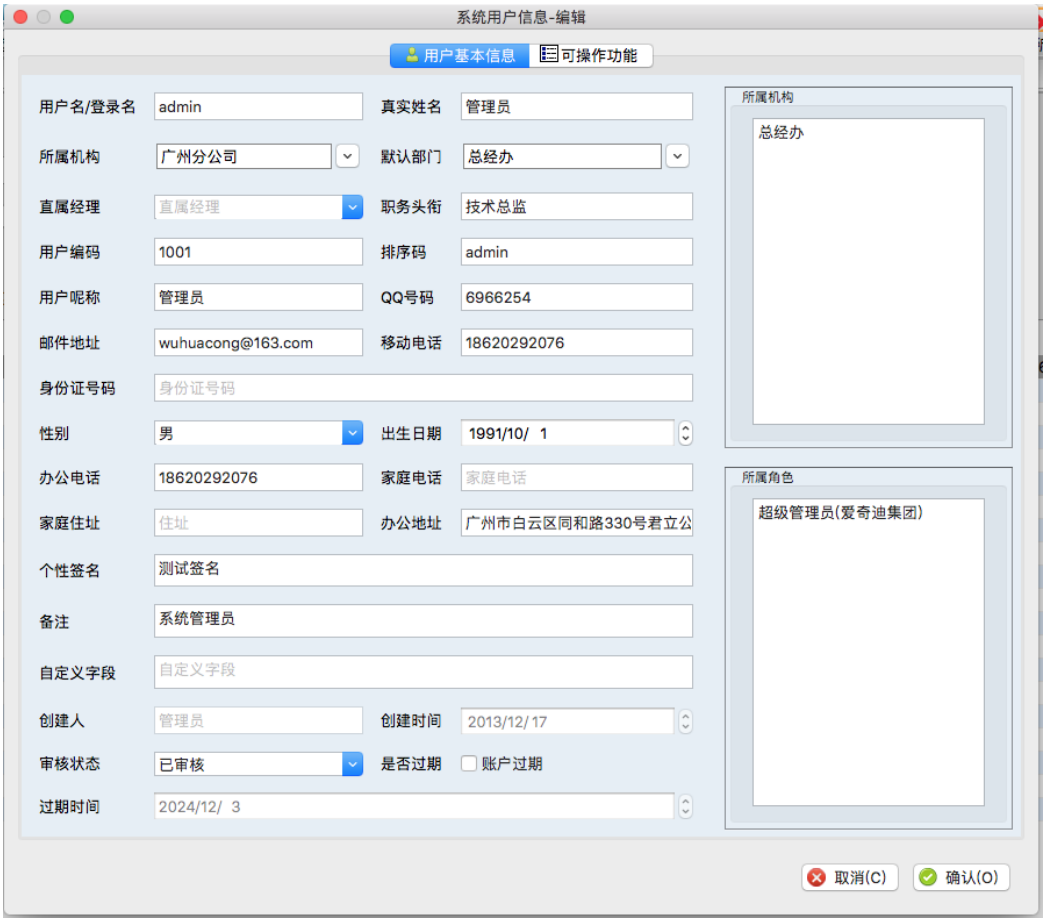
继承基类编辑对话框（通常用于创建模态对话框或自定义窗口的基类）有以下优点：

- **共享通用功能**：将所有对话框的共同行为（如按钮布局、事件处理、数据校验逻辑等）封装在基类中，子类可以直接继承使用，无需重复实现。
- **减少冗余代码**：对话框的通用结构只需在基类中实现一次，后续的功能扩展只需通过继承来实现，减少代码重复。
- **统一界面风格**：基类可以预定义窗口的样式和布局，确保项目中所有对话框的界面一致。
- **快速定制功能**：子类仅需实现或覆盖特定方法，即可快速实现自定义对话框的功能。

继承基类编辑对话框能够提高代码复用性、开发效率和维护性，特别适合在复杂系统中管理多个相似对话框。通过合理设计基类，开发者可以显著减少重复代码，实现更灵活的功能扩展。

常规的对话框中，业务表编码规则的新增、编辑界面如下所示。

当然，我们也可以增加更多的定制功能，稍作调整可以增加多页的功能。



一般我们在编辑框中，或者列表窗体中，我们都可能有树形列表的情况，我们提供标准的处理方法，用于对这些内容进行修改。

对于编辑界面来说，我们继承父类后，子类重写一些实现函数即可实现弹性化的处理了。

```python
class FrmCustomerEdit(ctrl.BaseEditDialog[CustomerDto]):
    """客户信息-新增/编辑界面"""

    module_name = "客户信息"
    entity_info = CustomerDto()

    def __init__(self, parent, entity_id=None):
        super().__init__(
            parent, model=CustomerDto, entity_id=entity_id, size=(500, 350)
        )

        # 使用业务实例化的api
        self.api = api

    def add_controls(self, panel: wx.Window) -> wx.GridBagSizer:…

    async def display_data(self):…

    async def check_input(self) -> bool:…

    async def init_dict_items(self):…

    async def LoadInfo(self, info: CustomerDto) -> None:…

    def SetInfo(self, info: CustomerDto) -> CustomerDto:…
```

上面这些函数就是各司其职，对界面的内容处理，显示编辑数据，校验输入，初始化字典、加载信息，保存对象信息，都是我们在编辑框中需要处理到的内容，我们根据不同的需求进行修改即可。

生成的界面代码中，对于输入控件的显示放在 add_controls 函数里面，如下代码所示。

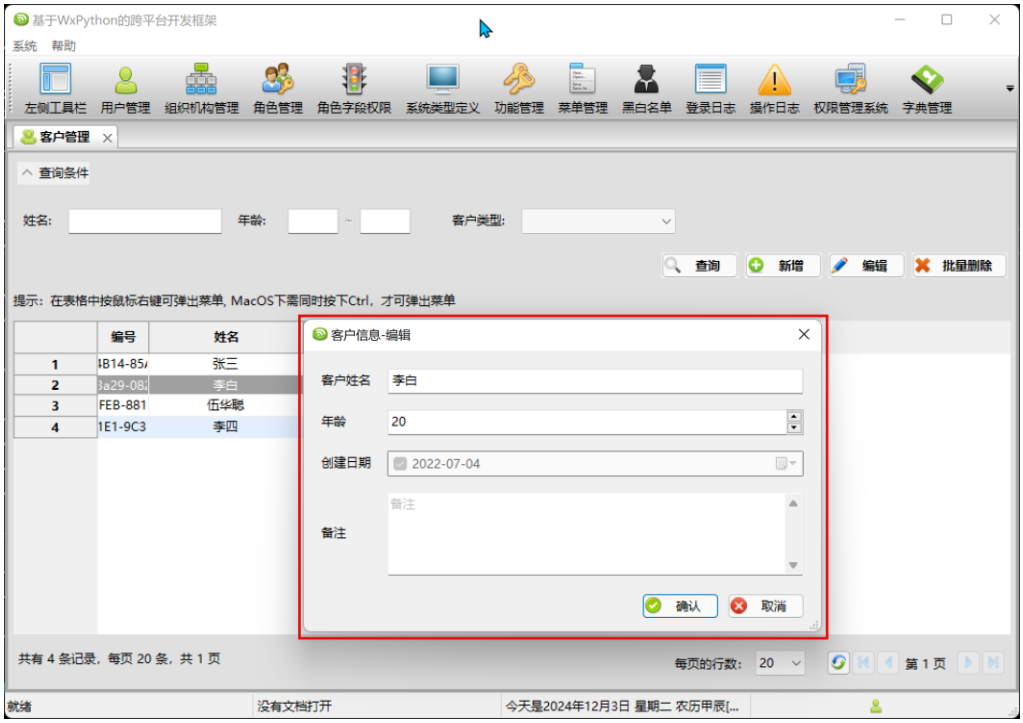
```
def add_controls(self, panel: wx.Window) -> wx.GridBagSizer:
    # 创建一个 GridBagSizer
    grid_sizer = wx.GridBagSizer(5, 5)  # 行间距和列间距为 5
    util = GridBagUtil(panel, grid_sizer, 2)  # 构建工具类，布局为2列

    self.txtName = ctrl.MyTextCtrl(panel, placeholder="客户姓名")
    self.txtAge = wx.SpinCtrl(panel)
    self.txtCreateTime = ctrl.MyDatePickerCtrl(panel)
    self.txtCreateTime.Disable()

    self.txtNote = ctrl.MyTextCtrl(panel, placeholder="备注", style=wx.TE_MULTILINE)

    util.add_control("客户姓名", self.txtName, is_expand=True)
    util.add_control("年龄", self.txtAge, is_expand=True)
    util.add_control("创建日期", self.txtCreateTime, is_expand=True)
    util.add_control("备注", self.txtNote, is_expand=True, is_span=True)

    # 让控件跟随窗口拉伸
    grid_sizer.AddGrowableCol(1)  # 允许第n列拉伸
    return grid_sizer
```

我们通过代码：

util = GridBagUtil(panel, grid_sizer, 2) # 构建工具类，布局为 2 列

构建了一个辅助类来处理布局的，添加的时候不用管布局，大概知道是几列的即可。

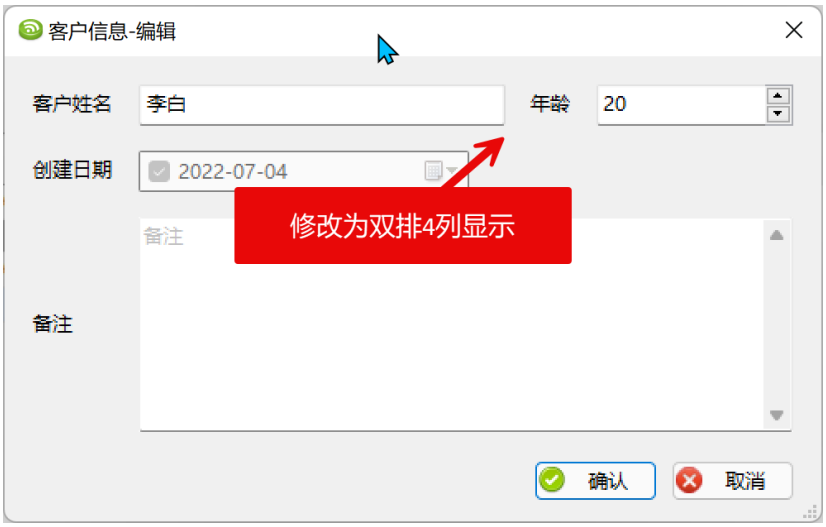
在 Windows 下，客户信息的编辑界面如下所示。



如果我们需要修改为双排的，那么修改下：

```
def add_controls(self, panel: wx.Window) -> wx.GridBagSizer:
    # 创建一个 GridBagSizer
    grid_sizer = wx.GridBagSizer(5, 5)  # 行间距和列间距为 5
    util = GridBagUtil(panel, grid_sizer, 4)  # 构建工具类，布局为4列

    self.txtName = ctrl.MyTextCtrl(panel, placeholder="客户姓名")
    self.txtAge = wx.SpinCtrl(panel)
    self.txtCreateTime = ctrl.MyDatePickerCtrl(panel)
    self.txtCreateTime.Disable()

    self.txtNote = ctrl.MyTextCtrl(panel, placeholder="备注", style=wx.TE_MULTILINE)

    util.add_control("客户姓名", self.txtName, is_expand=True)
    util.add_control("年龄", self.txtAge, is_expand=True)
    util.add_control("创建日期", self.txtCreateTime, is_span=True)
    util.add_control("备注", self.txtNote, is_expand=True, is_span=True, is_stretch=True)

    # 让控件跟随窗口拉伸
    grid_sizer.AddGrowableCol(1)  # 允许第n列拉伸
    grid_sizer.AddGrowableCol(3)  # 允许第n列拉伸
    return grid_sizer
```

界面效果如下所示。



稍作修改即可重新布局，非常方便。

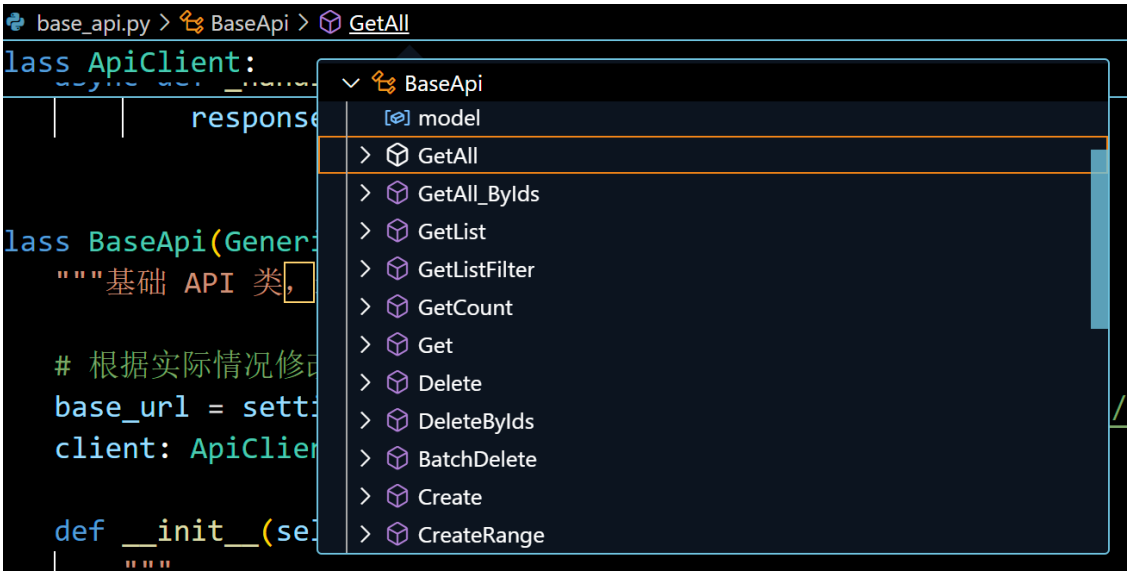
3.7.3. 前端对接 WebAPI 的接口封装类

利用代码生成工具，可以很好的利用现有基类的关系生成相关代码，包括界面代码，对 Web API 的调用代码也是一样，我们只需要做好继承关系，就具有基类一些的 CRUD 接口

了。

例如对于客户信息的 API 接口封装类，我们只需要增加一些个性化的自定义函数即可，默认对应基类有相关的 CRUD 接口。

```python
class Customer(BaseApi[CustomerDto]):
    """客户信息--API接口类"""

    api_name = "customer"

    def __init__(self):
        super().__init__(self.api_name, CustomerDto)

    async def exist(self, name: str = None, id: str = None) -> bool:
        """判断记录是否存在,如果指定ID，则判断ID不等于当前ID的记录是否存在"""
        url = f"{self.base_url}/exist"
        params = {"name": name, "id": id}
        data = await self.client.get(url, params=params)

        res = AjaxResponse[bool].model_validate(data)
        return res.result if res.success else False

    async def get_by_name(self, name: str) -> CustomerDto:
        """根据名称获取客户信息"""
        url = f"{self.base_url}/by-name"
        params = {"name": name}
        data = await self.client.get(url, params=params)

        res = AjaxResponse[CustomerDto].model_validate(data)
        return res.result if res.success else None


# 构建一个业务逻辑实例，方便调用
api_customer = Customer()
```



这个基类的函数，和后端的控制器接口一一对应。

利用代码生成工具，开发项目事半功倍，这就是工具的力量和魅力。

3.8. 复杂界面内容的分拆和重组处理

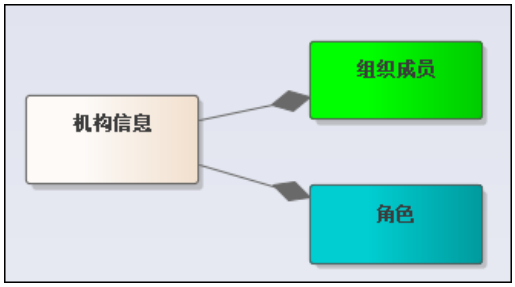
复杂界面内容的分拆和重组处理是现代软件开发中常见的做法，尤其在开发大型应用程序时，可以大幅提升开发效率、可维护性和用户体验。

通过将复杂的界面内容分拆成更小的模块，每个模块都专注于单一功能或组件，代码更容易理解和维护。模块化的界面组件可以在多个地方复用，减少了重复开发的工作。通过将复杂的界面分拆为多个小模块，开发者可以更专注于每个模块的细节，优化每个部分的用户体验。

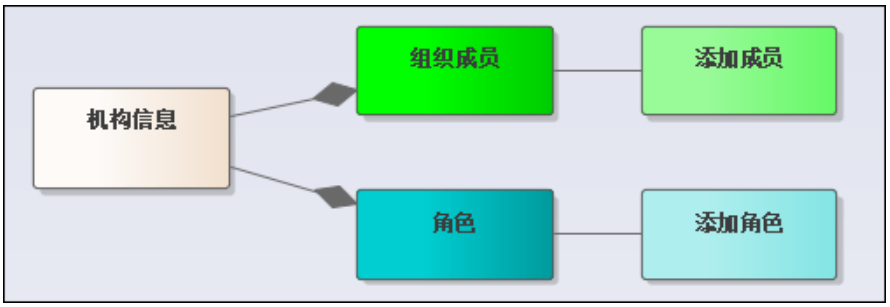
3.8.1. 一般界面项目的相关实现

如在一般的机构或角色信息中，界面内容比较多，可以进行拆分，根据内容的展示不同，拆分为各自的组件模块，然后合并使用即可，如下所示。

在对象 UML 的图例中，如下所示的效果图，组织机构包含组织成员和角色的内容。



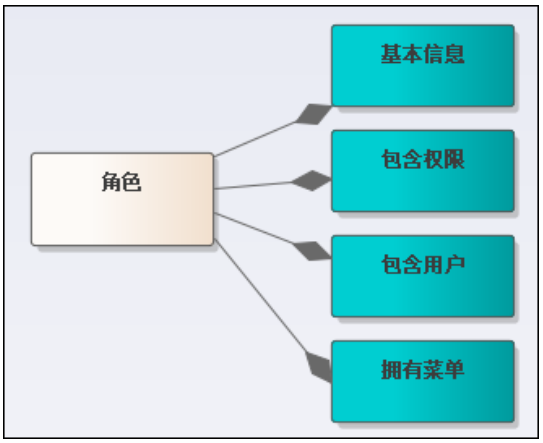
在界面上，组织成员还需要添加成员的功能，同理角色也需要添加角色的处理，如下 UML 图示。



其中每个角色，除了包含基本信息外，还会包含拥有的权限（功能控制点）、包含用户，以及拥有的菜单，其中权限是用来控制界面元素，如操作按钮的显示的，而拥有的菜单，则

是用户以指定账号登录系统后，获得对应角色的菜单，然后构建对应的访问入口的。

角色界面模块 UML 类图如下所示。



在 Vue+Element 的 BS 前端项目上， 我们以不同的 Tab 来展示这些信息，如下所示。 其中可以看到不同的 Tab 显示不同的内容。

另外用户信息和角色信息类似，也需要展示复杂的相关信息，用户本身拥有的权限功能点和拥有的菜单，如下界面所示。

上面介绍是基于 Vue+Element 的前端开发界面处理，其实对于 Python 的 WxPython 组件的分拆和组合来说一样非常有必要，也更加方便。

3.8.2. WxPython 中复杂界面的分拆和重组

在 Python 的 wxPython 框架中，复杂界面内容的分拆和重组同样是非常重要的做法。由于 wxPython 提供了许多丰富的控件和布局管理器，能够帮助开发者高效地构建界面，分拆和重组组件不仅有助于提高代码的可维护性和可复用性，还能够提升界面的响应性和用户体验。

1）**组件化设计**

和前面提到的模块化设计类似，wxPython 的组件化设计就是将界面拆分成多个小的、功能明确的控件或子窗口，这些控件可以是：

- **基本控件**：例如按钮、文本框、标签、选择框等。
- **容器控件**：如 Panel、BoxSizer、SplitterWindow，用于管理其他控件的布局和显示。

- **自定义控件**：通过继承 wxPython 的基本控件来创建自己的控件（例如，自定义表格、树状视图等）。

常见做法：

- 将一个复杂的窗口分解成多个 Panel 或子窗口（子界面），每个 Panel 专注于处理特定的任务（例如，数据输入区、数据展示区、按钮区等）。
- 创建可复用的组件类（如自定义控件），将不同的 UI 元素封装在类内部，使得这些控件可以在不同地方重用。

2）**布局管理器**

wxPython 提供了多种布局管理器（如 BoxSizer、GridSizer、FlexGridSizer 等）来帮助自动调整控件的大小和位置，适应不同的窗口大小，提升界面响应性。

常见做法：

- **BoxSizer**：常用于垂直或水平布局。

  通过 wx.BoxSizer(wx.HORIZONTAL) 或 wx.BoxSizer(wx.VERTICAL) 将控件按顺序排列。
- **GridSizer**：用于将控件组织成网格布局。
- **FlexGridSizer**：类似于 GridSizer，但支持单元格自适应大小，适合处理复杂的布局需求。
- **SplitterWindow**：允许在多个子窗口之间进行分割，以便显示多个视图或控件。

通过合理地使用布局管理器，可以有效地分拆界面并控制各个控件的布局和展示。

3）**事件绑定与回调函数**

在 wxPython 中，界面控件的交互通过事件机制来处理。通过事件绑定，可以将用户操作（如按钮点击、文本框输入等）与特定的回调函数（事件处理函数）关联起来。

4）**使用自定义对话框**

对于复杂的交互逻辑，使用自定义对话框（wx.Dialog）是一个有效的分拆方法。可以将一些特定功能（如设置、用户登录等）封装到单独的对话框中，避免主界面过于复杂。

5）**分拆大型界面为多个窗口或视图**

在复杂应用中，往往需要处理多个界面或视图。例如，可以将主界面和设置、帮助等内容分开，使用 wx.Notebook、wx.SplitterWindow 或多文档界面（MDI）来实现不同界面之间的切换和组织。

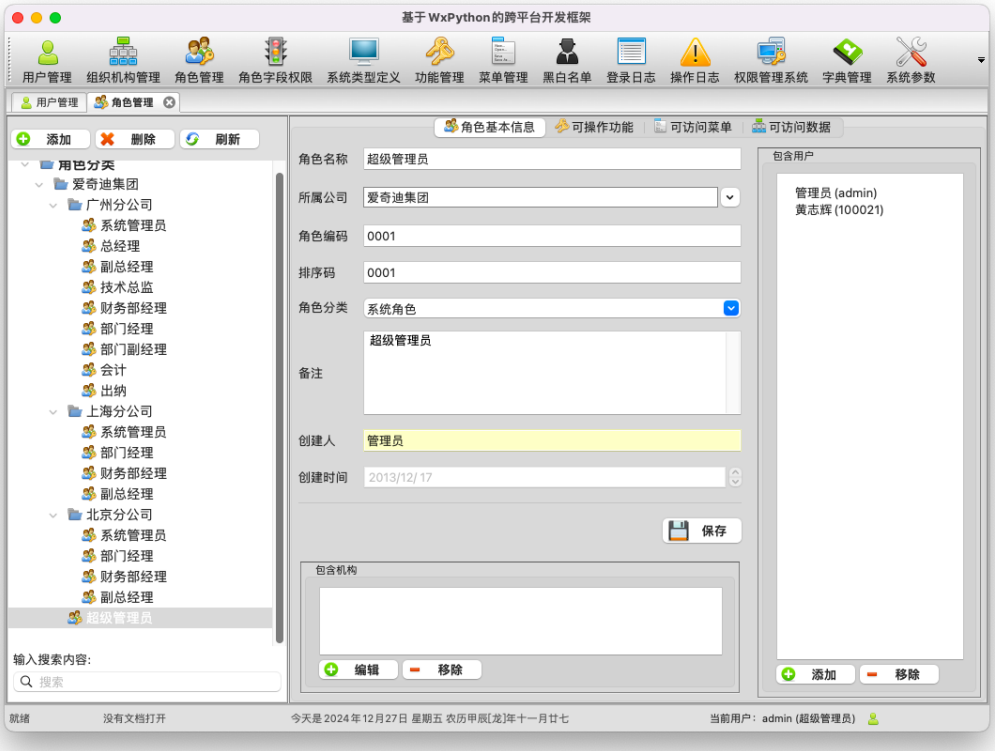
常见做法：

- 使用 wx.Notebook 来实现标签页式布局，每个标签页对应一个不同的功能模块。
- 使用 wx.SplitterWindow 来分隔主界面的上下或左右区域，使得界面更加灵活和适应不同的操作需求。

这些做法能够帮助开发者有效组织复杂的 wxPython 界面，使其更加灵活、易于维护并提供良好的用户体验。

介绍完理论知识后，我们来实际看看在 WxPython 跨平台开发框架中是如何实现界面的分拆和重组的。

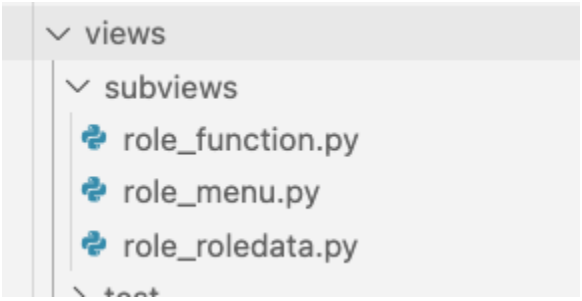
在 WxPython 的项目中，角色信息的展示也是类似的多选项卡来实现分来展示的。其中包括：角色基本信息、可操作功能、可访问菜单、可访问数据等等内容。如下界面所示。



如果把这些内容的控件布局，和具体的事件整合在一个文件里，那么界面代码比较臃肿，也很难修改维护。因此把它们分门别类的进行独立分拆。

初步我们可以把这些内容按选项卡的界面来分别创建多个面板（Panel)的创建和实现，然后在主界面上整合它们即可，进一步，我们还可以把相同的列表或者局部界面进行独立出来重用，多次重构后，可以使得我们的界面维护起来更加方便。

如我们先根据界面的不同部分进行创建不同的文件，用来构建局部内容的展示处理。



然后在主界面 frm_role.py 进行汇总整合即可，如下所示，先引入相关的局部面板界面。

frm_role.py

app > views > frm_role.py > ...

```
33
34  from core.config import settings
35  from views.frm_ou_select import FrmOuSelect
36  from views.frm_user_select import FrmUserSelect
37
38  from views.subviews.role_roledata import Role_RoleData
39  from views.subviews.role_menu import Role_Menu
40  from views.subviews.role_function import Role_Function
41
42
43  class FrmRole(wx.Panel):
44      """角色管理"""
45
46      def __init__(self, parent):
47          super().__init__(parent, id=wx.ID_ANY, size=(1000, 800))
48
```

然后我们通过编写一个函数，让他们组合到 wx.NoteBook 控件上即可，如下代码所示。

```
def create_content(self, parent: wx.Panel):
    """创建列表面板"""

    # 为NotBook 准备图标
    img_list = get_image_list(["group_key", "key", "menu", "organ"], 16)

    # 添加NoteBook
    self.notebook = wx.Notebook(parent)
    self.notebook.AssignImageList(img_list)
    self.notebook.AddPage(self.create_main_basic(self.notebook), "角色基本信息", imageId=0)

    self.role_function_panel = Role_Function(self.notebook)
    self.notebook.AddPage(self.role_function_panel, "可操作功能", imageId=1)

    self.role_menu_panel = Role_Menu(self.notebook)
    self.notebook.AddPage(self.role_menu_panel, "可访问菜单", imageId=2)

    self.role_data_panel = Role_RoleData(self.notebook)
    self.notebook.AddPage(self.role_data_panel, "可访问数据", imageId=3)

    return self.notebook
```

这样各司其职，各个部分负责自己内部的界面展示和数据初始化的逻辑即可，如果我们需要调用某个组件的处理，那么公布对应的方法来操作数据即可。

如我们在数据显示的时候，让它们组件各自负责相关的更新处理即可，如下代码所示是在对具体角色的数据进行显示的时候，调用各个组件对象进行更新显示。

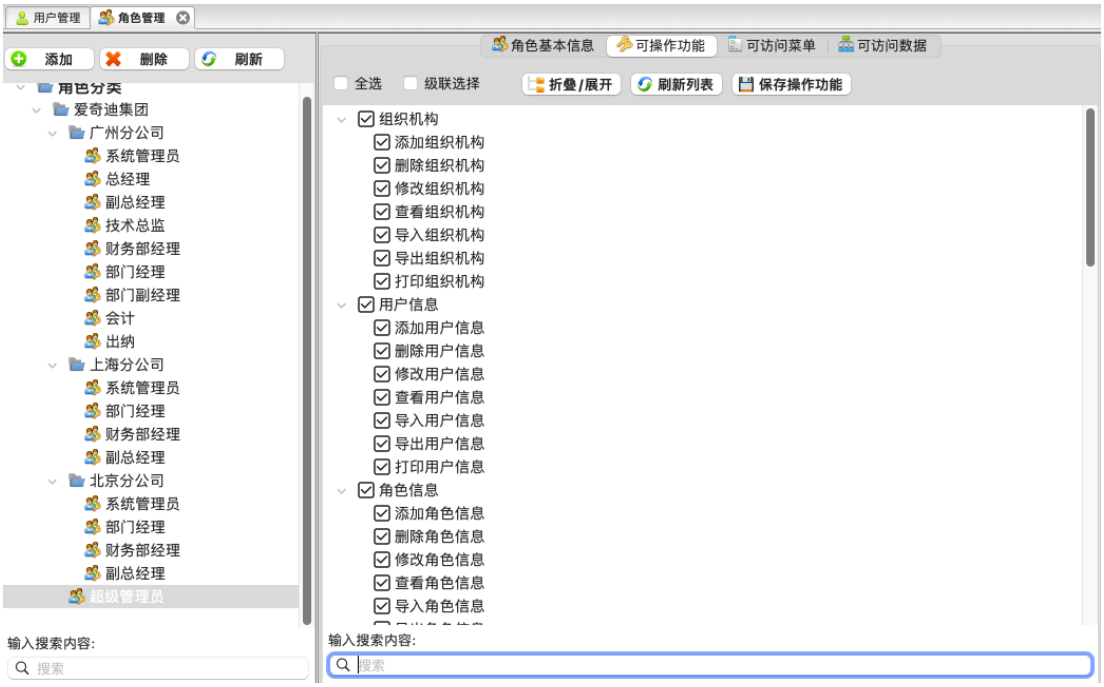
```python
class FrmRole(wx.Panel):
    async def display_data(self):

        if self.entity_id is None:…

        else:
            # 显示当前实体信息
            busy = MessageUtil.show_busy(self)
            info = await api_role.Get(self.entity_id)
            if info.success:
                self.entity_info = info.result
                await self.LoadInfo(info.result)

                # 刷新用户列表和机构列表
                await self.RefreshUsers()
                await self.RefreshOus()

                await self.role_function_panel.refresh_role_functions(self.entity_id)
                await self.role_menu_panel.refresh_role_menus(self.entity_id)
                await self.role_data_panel.refresh_role_orgdata(self.entity_id)
            else:
                MessageUtil.show_warning(self, "获取数据失败")

            MessageUtil.close_busy(busy)
```

类似这样处理，我们就可以控制界面组件的有限操作即可，其他交由内部的事件进行相关数据的保存操作即可。

同样，对于其他模块，我们也可以采用同样的方式来简化界面的处理逻辑，同时也是分拆关注点，便于我们更加快捷的处理问题。

如用户管理模块的信息展示中，也是包含 很多内容。

我们把常用到的树形列表，封装为一个独立的界面控件，这样可以在很多场合上使用，如只需要展示类别，而不需要勾选的时候，如上图所示，如果需要勾选，我们通过一个变量来控制显示复选框即可，如下界面所示。上面和下面的两个树形列表的展示，是基于一个自定义的树形控件再封装的。



以上就是 WxPython 跨平台开发框架之复杂界面内容的分拆和重组处理的一些经验分享，希望你喜欢，可以在分析复杂界面的时候，运用这些分而治之的理念，通过分拆内容，实现关注点的聚焦，从而能够快捷的处理一些复杂的问题。

4. FastApi 后端框架开发

《Python 开发框架》的后端是基于 Python+FastApi+SqlAlchemy + Pydantic+Redis 的技术路线。

FastAPI 是一个现代、快速（高性能）的 Web 框架，用于构建 API。它基于 Python 3.7+ 的类型提示，并且依赖于 Starlette（用于 web 服务器和路由）和 Pydantic（用于数据验证和序列化）。FastAPI 的设计目标是提供与 Flask 和 Django 类似的开发体验，但在性能、类型安全和开发者友好性方面做出更大的提升。GitHub 地址：https://github.com/fastapi/fastapi

FastAPI 的主要特性

- **极高的性能**：基于 ASGI 的异步支持，使得 FastAPI 在性能上接近 Node.js 和 Go

的水平，适合处理高并发。

- **自动生成 API 文档**：使用 OpenAPI 和 JSON Schema 自动生成交互式的 API 文档（如 Swagger UI 和 ReDoc）。
- **基于类型提示的自动验证**：利用 Python 的类型提示和 Pydantic，自动进行数据验证和解析。
- **异步支持**：原生支持 async 和 await，能够处理异步任务，适合与数据库、第三方 API、WebSocket 等交互。
- **内置依赖注入系统**：使得依赖的声明和管理变得简洁而强大，便于模块化设计。
- **开发者友好**：提供了详细的错误信息和文档，支持自动补全，极大提升了开发效率。

当你运行 FastAPI 应用时，它会自动生成交互式文档：

- **Swagger UI**：访问 http://127.0.0.1:8000/docs
- **ReDoc**：访问 http://127.0.0.1:8000/redoc

这两个文档界面可以让你查看 API 的结构，甚至可以直接在界面中进行 API 调用。

FastAPI 是一个非常现代化和高效的框架，非常适合用于构建高性能的 API。其自动文

档生成、数据验证和依赖注入等特性，使得开发者能够更快、更安全地编写代码，并提供出色的用户体验。

4.1. 常规开发工具的安装

《Python 开发框架》的后端 Web API 是基于 FastAPI 的框架应用，开发工具一般采用 VSCode；数据库默认采用 MySql，而登录部分也采用了 Redis 缓存的数据库。

开发框架支持多种数据库，SqlServer、MySql、Oracle、SQLite、PostgreSQL、达梦数据库都可以支持，可以再配置文件中修改指定即可。

VS 开发工具可以在微软官方下载即可：https://visualstudio.microsoft.com/zh-hans/ ，默认下载社区版本即可。

MySQL 或其他数据库则自行获取地址进行下载，数据库表管理推荐使用 Navicat Premium 17 或以上版本进行维护。

框架后端登录部分采用了 Redis 缓存的数据库，因此也需要 Redis 的环境支持，Redis 数据库下载地址：https://github.com/MSOpenTech/redis/releases，Redis 管理工具下载地址：http://redisdesktop.com/download。

4.1.1. Redis 的安装使用

Redis 是一个开源的使用 ANSI C 语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库，和 Memcached 类似，它支持存储的 value 类型相对更多，包括 string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和 hash（哈希类型）。在此基础上，redis 支持各种不同方式的排序。与 memcached 一样，为了保证效率，数据都是缓存在内存中。区别的是 Redis 会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了 master-slave(主从)同步。

Redis 的代码遵循 ANSI-C 编写，可以在所有 POSIX 系统（如 Linux, *BSD, Mac OS X, Solaris 等）上安装运行。而且 Redis 并不依赖任何非标准库，也没有编译参数必需添加。

1）Redis 支持两种持久化方式：

(1) :snapshotting(快照)也是默认方式.(把数据做一个备份，将数据存储到文件)

(2) Append-only file(缩写 aof)的方式

快照是默认的持久化方式，这种方式是将内存中数据以快照的方式写到二进制文件中，默认的文件名称为 dump.rdb.可以通过配置设置自动做快照持久化的方式。我们可以配置 redis 在 n 秒内如果超过 m 个 key 键修改就自动做快照.

aof 方式:由于快照方式是在一定间隔时间做一次的，所以如果 Redis 意外 down 掉的话，就会丢失最后一次快照后的所有修改。aof 比快照方式有更好的持久化性，是由于在使用 aof 时，redis 会将每一个收到的写命令都通过 write 函数追加到文件中，当 redis 重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。

2）Redis 数据结构

Redis 的作者 antirez 曾称其为一个数据结构服务器（**data structures server**），这是一个非常准确的表述，Redis 的所有功能就是将数据以其固有的几种结构保存，并提供给用户操作这几种结构的接口。我们可以想象我们在各种语言中的那些固有数据类型及其操作。

Redis 目前提供四种数据类型：**string**,**list**,**set** 及 **zset**(sorted set)和 **Hash**。

- string 是最简单的类型，你可以理解成与 Memcached 一模一个的类型，一个 key 对应一个 value，其上支持的操作与 Memcached 的操作类似。但它的功能更丰富。
- list 是一个链表结构，主要功能是 push、pop、获取一个范围的所有值等等。操作中 key 理解为链表的名字。
- set 是集合，和我们数学中的集合概念相似，对集合的操作有添加删除元素，有对多个集合求交并差等操作。操作中 key 理解为集合的名字。
- zset 是 set 的一个升级版本，他在 set 的基础上增加了一个顺序属性，这一属性在添加修改元素的时候可以指定，每次指定后，zset 会自动重新按新的值调整顺序。可以理解了有两列的 mysql 表，一列存 value，一列存顺序。操作中 key 理解为 zset 的名字。
- Hash 数据类型允许用户用 Redis 存储对象类型,Hash 数据类型的一个重要优点是,当你存储的数据对象只有很少几个 key 值时,数据存储的内存消耗会很小.更多关于 Hash 数据类型的说明请见：http://code.google.com/p/redis/wiki/Hashes

3）Redis 数据存储

Redis 的存储分为内存存储、磁盘存储和 log 文件三部分，配置文件中有三个参数对其进行配置。

save seconds updates，save 配置，指出在多长时间内，有多少次更新操作，就将数据同步到数据文件。这个可以多个条件配合，比如默认配置文件中的设置，就设置了三个条件。

appendonly yes/no ，appendonly 配置，指出是否在每次更新操作后进行日志记录，如果不开启，可能会在断电时导致一段时间内的数据丢失。因为 redis 本身同步数据文件是按上面的 save 条件来同步的，所以有的数据会在一段时间内只存在于内存中。

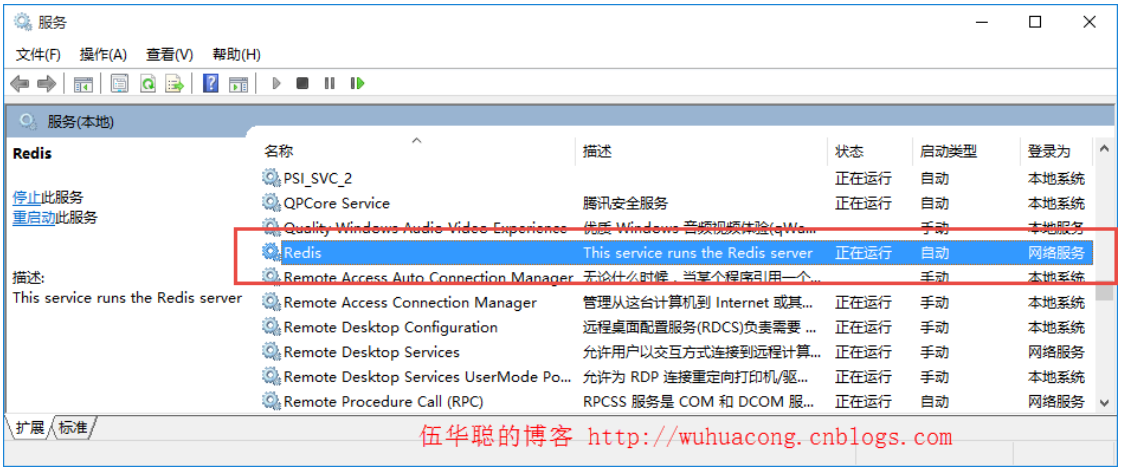
appendfsync no/always/everysec ，appendfsync 配置，no 表示等操作系统进行数据缓存同步到磁盘，always 表示每次更新操作后手动调用 fsync()将数据写到磁盘，everysec 表示每秒同步一次。

4）Redis 的安装

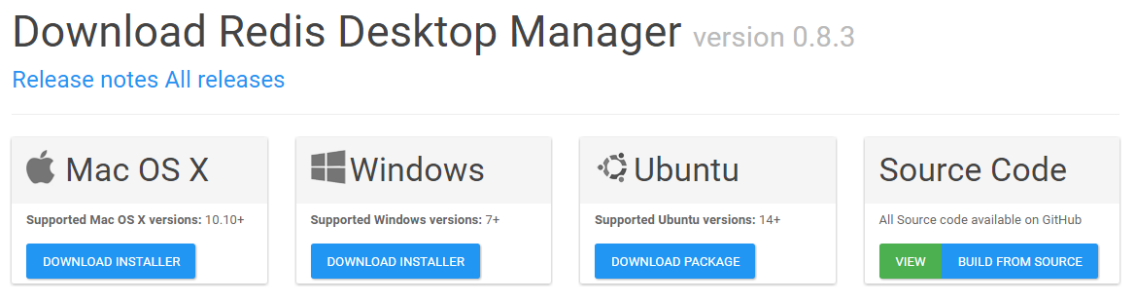
Redis 可以在不同的平台运行，不过我主要基于 Windows 进行开发工作，所以下面主要是基于 Windows 平台进行介绍。

Redis 可以安装以 DOS 窗口启动的，也可以安装为 Windows 服务的，一般为了方便，我们更愿意把它安装为 Windows 服务，这样可以比较方便管理。下载地址：https://github.com/MSOpenTech/redis/releases 下载，安装为 Windows 服务即可。

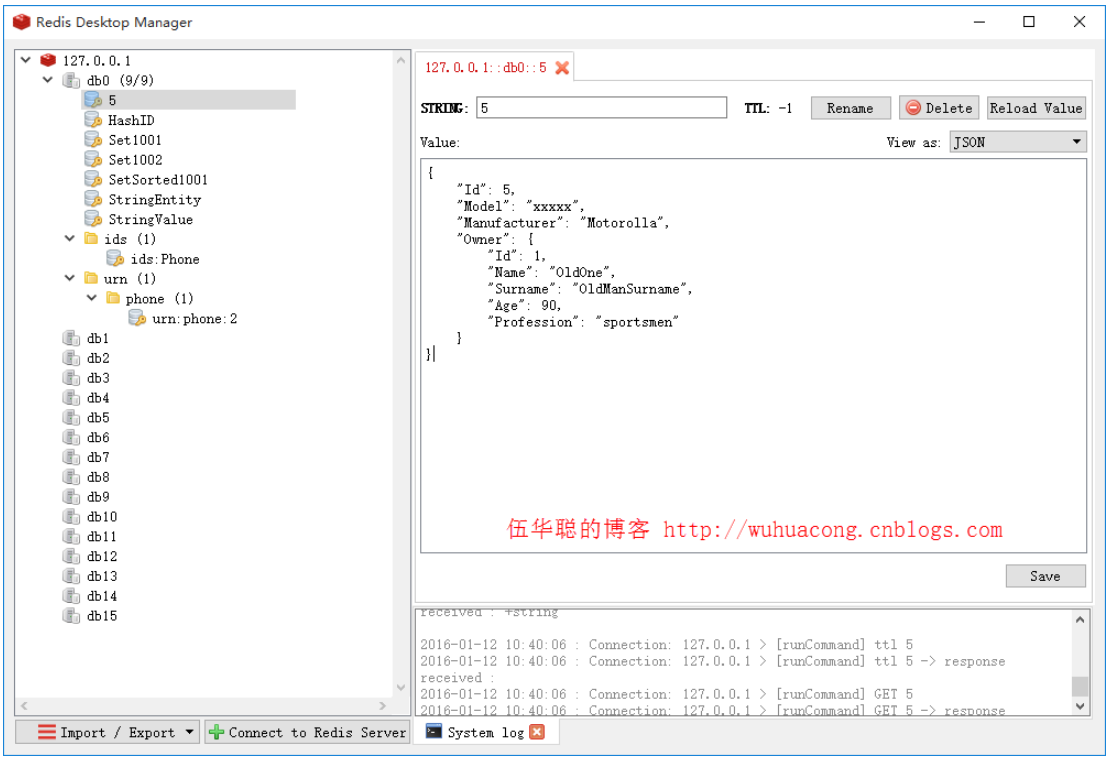
当前可以下载到最新的 Windows 安装版本为 3.0，安装后作为 Windows 服务运行，安装后可以在系统的服务里面看到 Redis 的服务在运行了，如下图所示。



安装好 Redis 后，还有一个 Redis 伴侣 Redis Desktop Manager 需要安装，这样可以实时查看 Redis 缓存里面有哪些数据，具体地址如下：http://redisdesktop.com/download 下载属于自己平台的版本即可

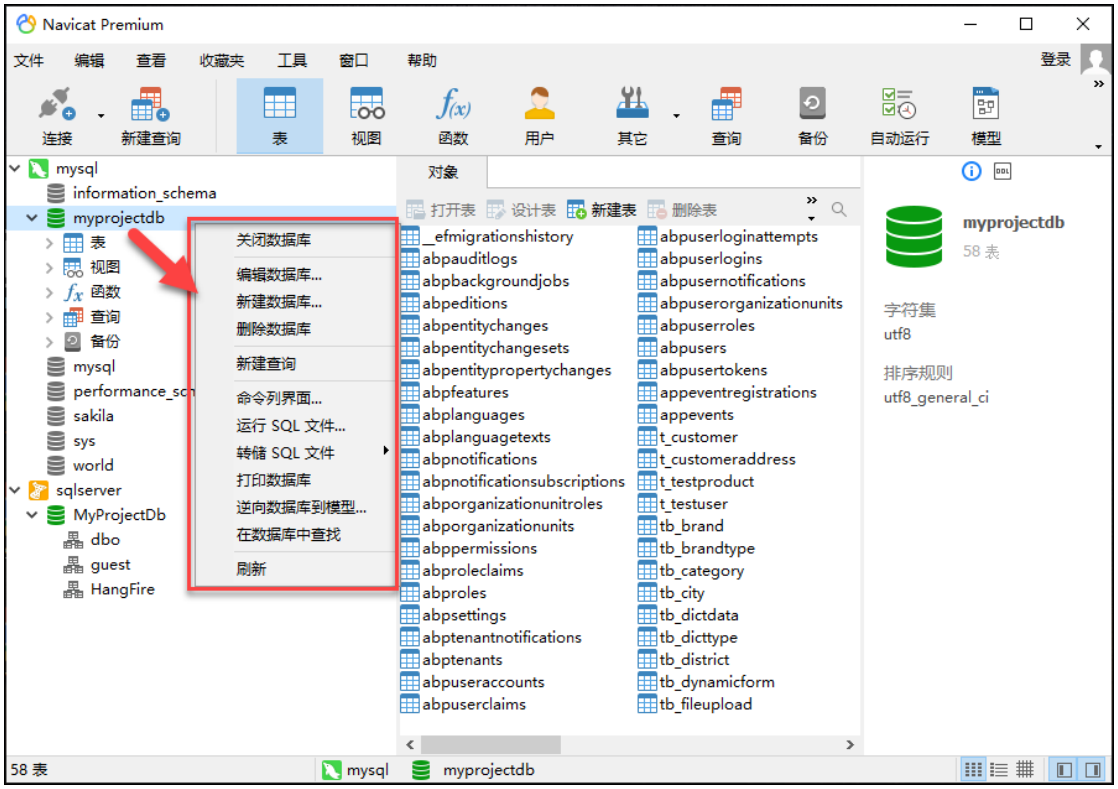


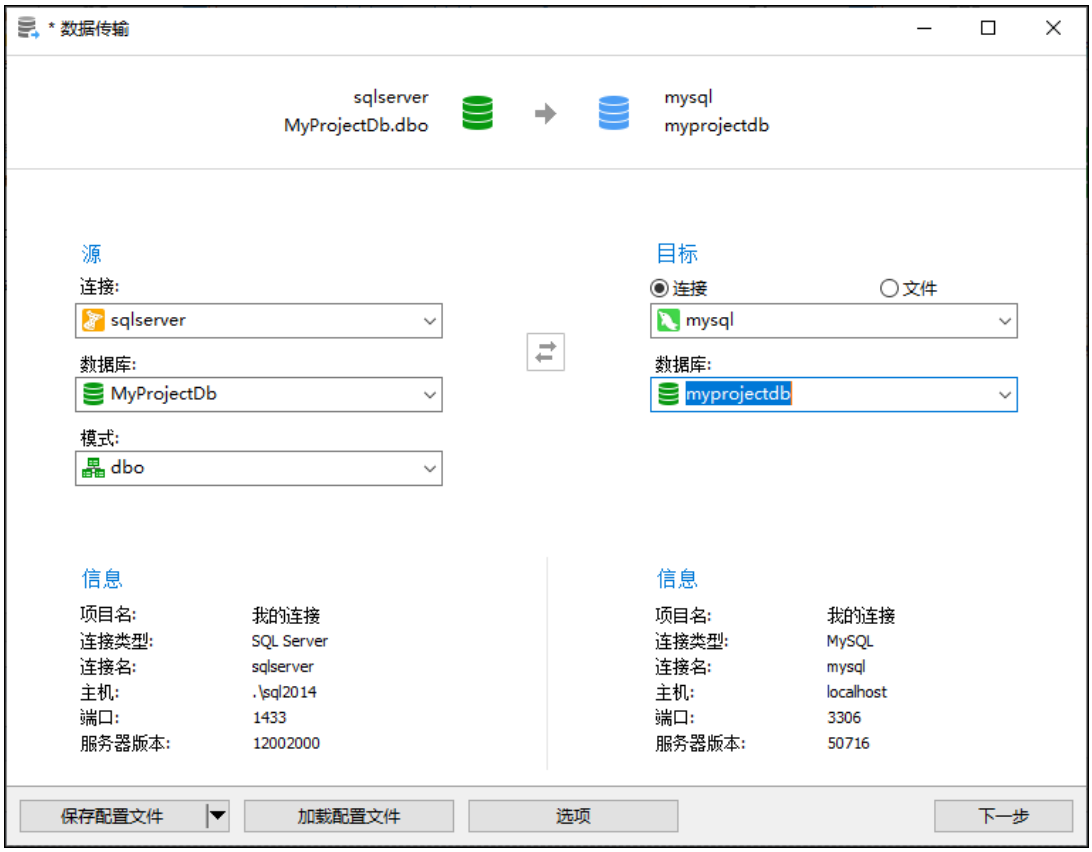
下载安装后，打开运行界面，如果我们往里面添加键值的数据，那么可以看到里面的数据了。



4.2. MySQL 数据库及管理工具

开发框架后端可以切换不同的数据库，如切换为 MySQL 数据库，那么你需要准备好 MySQL 的开发环境，一般需要 MySQL 5.7 或以上版本，而 MySQL 管理工具则推荐使用 Navicat Premium15 或以上版本，Navicat Premium 是一个非常方便、强大的 MySQL 数据库管理工具，可以生成/执行 SQL 脚本，从不同类型的数据库中导入数据库等。


Navigator Premium
文件 编辑 查看 收藏夹 工具 窗口 帮助
登录
连接 新建查询 表 视图 函数 用户 其它 查询 备份 自动运行 模型
mysql
information_schema
myprojectdb
表
视图
函数
查询
备份
mysql
performance_sch
sakila
sys
world
sqlserver
MyProjectDb
dbo
guest
HangFire
关闭数据库
编辑数据库...
新建数据库...
删除数据库
新建查询
命令列界面...
运行 SQL 文件...
转储 SQL 文件
打印数据库
逆向数据库到模型...
在数据库中查找
刷新
对象
打开表 设计表 新建表 删除表
_efmigrationshistory
abpauditlogs
abpbackgroundjobs
abpeditions
abpentitychanges
abpentitychangesets
abpentitypropertychanges
abpfeatures
abplanguages
abplanguagetexts
abpnotifications
abpnotificationsubscriptions
abporganizationunitroles
abporganizationunits
abppermissions
abproleclaims
abproles
abpsettings
abptenantnotifications
abptenants
abpuseraccounts
abpuserclaims
abpuserloginattempts
abpuserlogins
abpusernotifications
abpuserorganizationunits
abpuserroles
abpusers
abpusertokens
appeventregistrations
appevents
t_customer
t_customeraddress
t_testproduct
t_testuser
tb_brand
tb_brandtype
tb_category
tb_city
tb_dictdata
tb_dicttype
tb_district
tb_dynamicform
tb_fileupload
myprojectdb
58 表
字符集
utf8
排序规则
utf8_general_ci
58 表
mysql
myprojectdb


* 数据传输
sqlserver
MyProjectDb.dbo
mysql
myprojectdb
源
连接:
sqlserver
数据库:
MyProjectDb
模式:
dbo
目标
连接
文件
mysql
数据库:
myprojectdb
信息
项目名: 我的连接
连接类型: SQL Server
连接名: sqlserver
主机: .\sql2014
端口: 1433
服务器版本: 12002000
信息
项目名: 我的连接
连接类型: MySQL
连接名: mysql
主机: localhost
端口: 3306
服务器版本: 50716
保存配置文件
加载配置文件
选项
下一步

4.3. 复制方式部署基于 FastApi 的 WebAPI 服务端

开发框架的后端是基于 Python 的 FastAPI 的后端，遵循 Restful 的 Web API 标准，部署方式直接复制项目在服务上运行即可，运行前需要在服务器安装好 Python 运行环境，以及项目的依赖引用即可。

复制方式部署相当于在服务器端准备相关的 Python 开发环境和 requirement.txt 的依赖包，环境准备会比较繁琐，建议生产环境使用 PyInstaller 打包方式部署。如果需要使用 PyInstaller 打包方式部署，可以参考下一节内容介绍。

Web API 的后端内容，采用 基于 Python+FastApi +SqlAlchemy+Pydantic+Redis 的后端框架。Python 项目具有跨平台运行的特性，可以在 Windows、MacOS、Ubuntu 等 Linux 系统上运行。为了正常运行 Python 开发框架的后端 Web API 项目，我们需要在服务器安装好 Python 运行环境，以及项目的依赖引用。

4.3.1. 下载 Python 3 安装包并安装。

Window 平台安装 Python: https://www.python.org/downloads/windows/

MacOS 平台安装 Python: https://www.python.org/downloads/mac-osx/

下载对应版本的 Python 安装程序，根据您的系统是 32 位还是 64 位，选择对应的安装包，一般服务器上使用 64 位的 Python 安装程序，安装的时候记得**勾选 “Add Python to PATH”**（非常重要，避免后续手动配置环境变量）。

对于大多数 Linux 发行版，您可以通过包管理器直接安装 Python。例如，在 Ubuntu 上，您可以使用以下命令：

```
sudo apt update
sudo apt install python3
```

MacOS 通常也会自动配置好环境变量。如果需要手动配置，可以编辑 ~/.bash_profile、~/.zshrc 或其他 shell 配置文件，添加 Python 的安装路径到 PATH 变量中。

安装完成后，务必验证 Python 是否成功安装，并检查版本信息是否正确，在命令行中验证 Python 版本是否正常安装：

```
python --version
```

在安装库之前，建议将 pip 更新到最新版本：

python -m pip install --upgrade pip

4.3.2. 上传后端项目到服务器中并安装虚拟环境

在 Python 3 中，创建和使用虚拟环境可以帮助你管理项目的依赖，避免系统级别的包和项目之间的冲突。以下是如何在 Python 3 中创建和使用虚拟环境的步骤。

1. 安装 venv 模块（如果尚未安装）

从 Python 3.3 起，venv 模块已经被包含在 Python 的标准库中。你不需要额外安装任何东西，只需要确保你的 Python 版本是 3.3 或更高。

你可以使用以下命令来检查 Python 版本：

python3 --version

2. 创建虚拟环境

使用 venv 模块来创建虚拟环境。首先，进入你想要放置项目的目录，然后运行以下命令：

python3 -m venv myenv

其中，myenv 是你想要创建的虚拟环境的名字，你可以根据需要自定义。此命令会在当前目录下创建一个名为 myenv 的子目录，里面包含了虚拟环境的所有文件。

3. 激活虚拟环境

虚拟环境创建完成后，你需要激活它。

在 Windows 上：

myenv\Scripts\activate

在 macOS 和 Linux 上：

source myenv/bin/activate

当虚拟环境激活时，你会看到命令行提示符前面有 (myenv)，表示当前正在使用 myenv 虚拟环境。

4. 安装依赖

激活虚拟环境后，你可以使用 pip 来安装项目所需的库和依赖。在项目中，我们通常会使用 requirements.txt 文件来记录所有依赖包，这样其他人可以通过 pip 安装所有依赖。

pip install -r requirements.txt

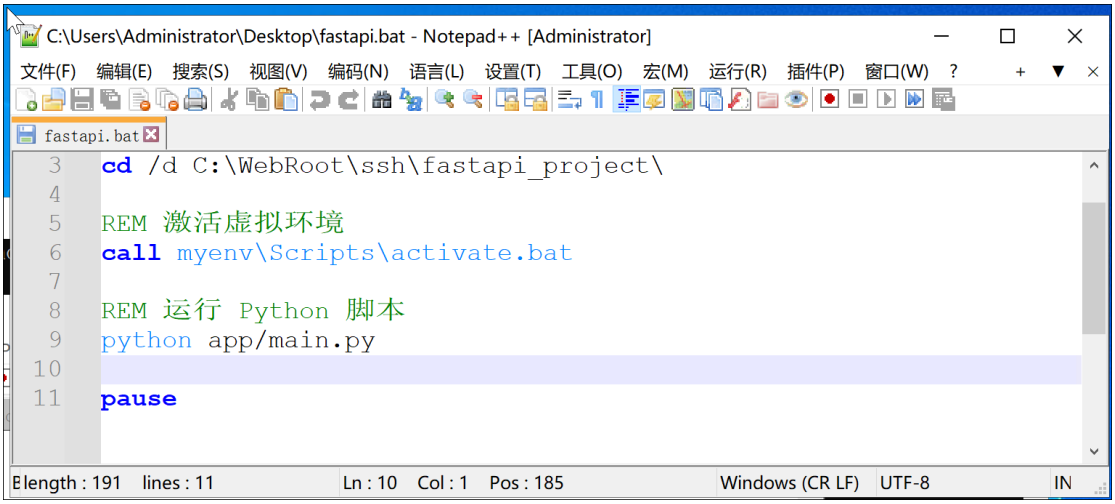
如果顺利安装，那么所有项目的依赖将在虚拟环境上安装完成。

5. 启动项目

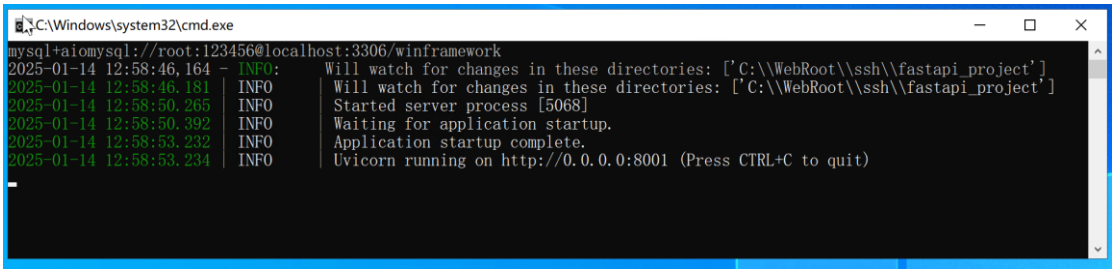
通过上面的步骤完成了虚拟环境和项目引用环境的安装后，就可以通过 Python 命令启动项目的 main.py 入口文件了。

python app/main.py

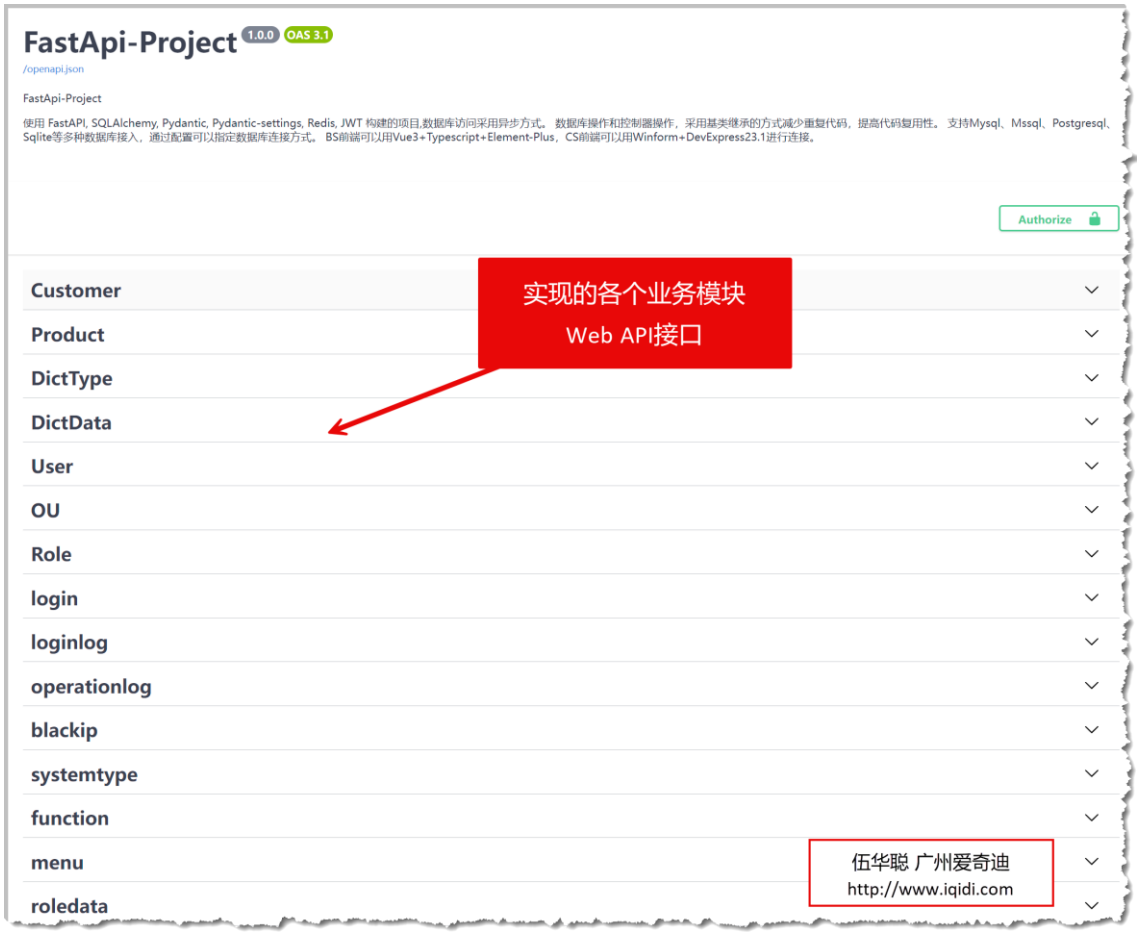
为了方便，在 Window 服务器中我们往往通过编写一个 bat 文件（Window 下批处理文件）来快速启动后端 Web API 项目。



启动项目过程中，可以看到命令行的提示结果。



正常启动项目后，可以看到 WebAPI 主页中有详细的 Swagger 文档介绍，非常方便参考使用。



有了统一 Web API 的后端，我们可以根据需要扩展实现自己的系统业务终端了。

4.3.3. 如何查看和终止正在运行的 Python 进程或端口

如何查看和终止正在运行的 Python 进程：

要查看和终止正在运行的 Python 进程，你可以根据不同的操作系统使用不同的命令和方法。以下是具体步骤：

输入以下命令并执行,查看正在运行的 Python 进程：

Mac/Linux 下命令：

ps -ef | grep python

其中，PID 是进程号，kill 命令用于终止进程。

kill -9 [PID]

windows 下命令：

查看所有 Python 进程：

tasklist | findstr python

结束所有 python 进程，可以使用 taskkill 命令：

taskkill /f /im python.exe

如何查看和终止正在运行的端口

要查看和终止正在运行的端口，你可以使用不同的命令来查找哪些进程占用了哪些端口，并根据需要终止它们。以下是根据不同操作系统的具体方法。

Windows 下命令：

查看正在运行的端口

netstat -ano | findstr :8000

终止正在运行的端口：

taskkill /PID <PID> /F

示例：

taskkill /PID 12345 /F

Mac 下命令：

查看正在运行的端口：

lsof -i:8000

终止正在运行的端口：

kill -9 [进程号]

4.4. PyInstaller 打包方式部署 FastApi 的 WebAPI 服务端

我们基于 Python 的 FastAPI 后端应用，在实际开发的时候，直接运行 main.py 进行调试即可，但是部署的时候，我们就需要把它们进行打包处理，这里首选 PyInstaller 进行打包。

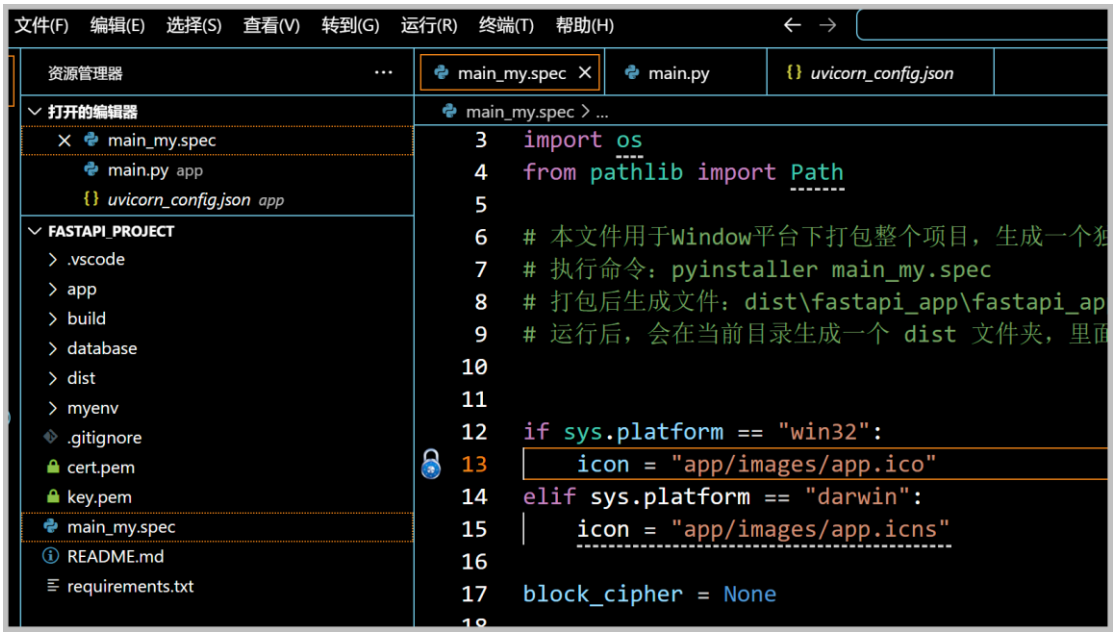
本节详细介绍了 **如何使用 PyInstaller 对基于 Python 的 FastAPI 后端项目进行打包与部署**，使其能够在目标环境中以独立可执行文件的形式运行，无需安装 Python 解释器或额外依赖。

4.4.1. 打包环境准备和了解

如果开发环境没有安装 PyInstaller，那么需要先安装才能使用打包操作。

```
pip install pyinstaller
```

我们需要注意几个文件，一个是 main_my.spec，一个是 app/main.py, 一个是 app/uvicorn_config.jsons，如下所示。



我们来详细介绍一下前面提到的三个文件在 **FastAPI + PyInstaller 项目**中的用途，以及它们之间的关系。理解清楚之后，你在打包和部署时就不会迷糊了。

---

1 app/main.py

用途：

- 这是你的 **FastAPI 应用的入口**，程序运行的起点。
- 定义了 FastAPI 的 app 实例、路由、业务逻辑，以及启动 Uvicorn 服务的逻辑。
- 在开发环境下，你直接用 python app/main.py 就能运行。
- 在打包环境下，这个文件会被 PyInstaller 分析并打包到可执行文件或目录中。

```python
from fastapi import FastAPI
import uvicorn

app = FastAPI()

@app.get("/")
async def root():
    return {"msg": "Hello FastAPI"}

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

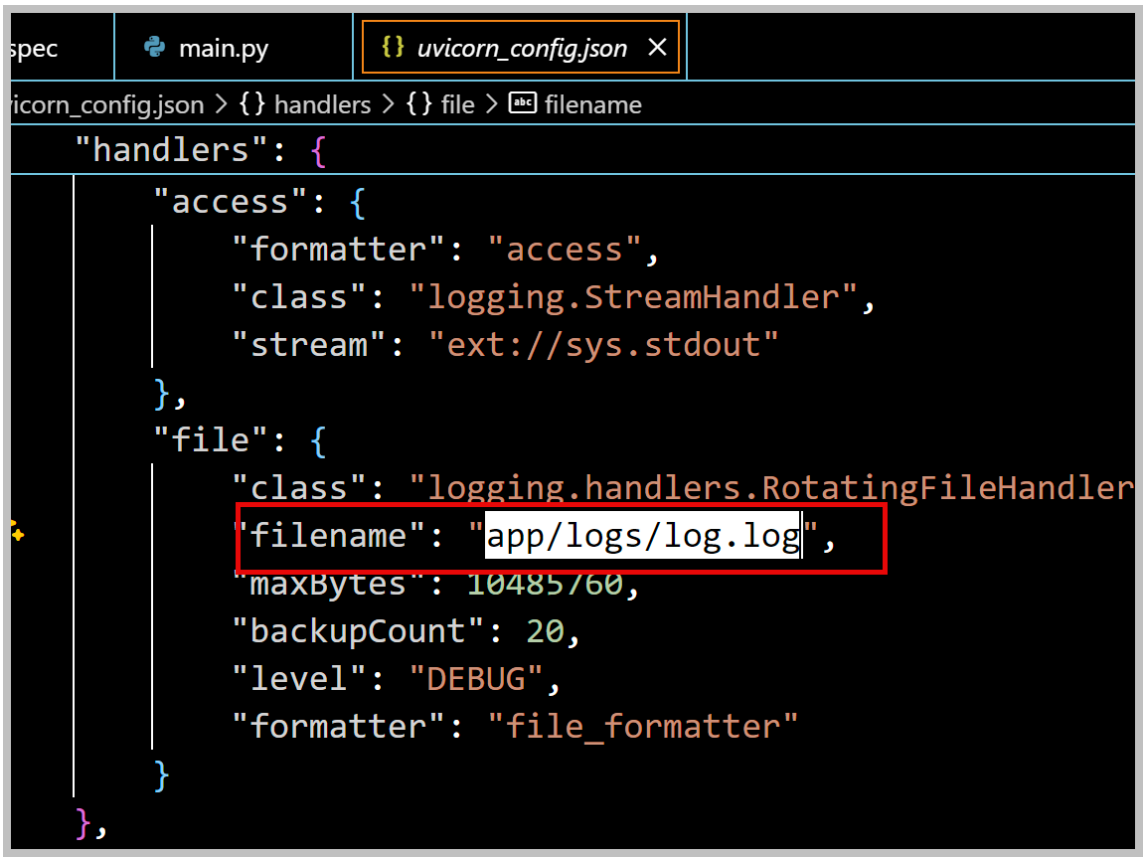
注意：

- 如果使用 PyInstaller 打包，main.py 是 Analysis 中被引用的 “script”，PyInstaller 会从这里开始分析依赖。
- 它也包含了 **日志目录创建、资源路径处理** 等逻辑，保证打包后的 exe 可以正常运行。

Main.py 文件启动 FastAPI 的时候，往往通过配置文件方式进行运行，那么需要解决相对路径的问题。

```
if __name__ == "__main__":

    # 日志配置路径
    config_path = resource_path("app/uvicorn_config.json")
    # 运行 uvicorn
    try:
        config = uvicorn.Config(
            app = socket_app,
            reload=True,
            host=settings.SERVER_IP,
            log_config = config_config,  # 日志配置
        )
        server = uvicorn.Server(config)
        server.run()
    except Exception as e:
        raise e
```

如果 app/uvicorn_config.json 配置了日志目录，运行的时候，你可能会发现 app/uvicorn_config.json 里面配置的日志文件路径和实际不对。



那么还需要动态进行调整下。

```python
if __name__ == "__main__":

    # 动态解析日志配置路径
    config_path = resource_path("app/uvicorn_config.json")

    # 加载并修改日志配置，主要对日志文件路径进行修正
    with open(config_path, "r", encoding="utf-8") as f:
        log_config = json.load(f)

    # 找到其中的 file handler，改写 filename 为绝对路径
    for handler in log_config.get("handlers", {}).values():
        if "filename" in handler:
            log_file = handler["filename"]
            abs_log_path = resource_path(log_file)
            os.makedirs(os.path.dirname(abs_log_path), exist_ok=True)
            handler["filename"] = abs_log_path  # 替换为绝对路径

    # 运行 uvicorn（传入已修改的 log_config dict）
    try:
        config = uvicorn.Config(
            app = socket_app,
            reload=True,
            host=settings.SERVER_IP,
            log_config = log_config,  # 日志配置,修正方式见上
        )

        server = uvicorn.Server(config)
        server.run()
    except Exception as e:
        raise e
```

2 app/uvicorn_config.json

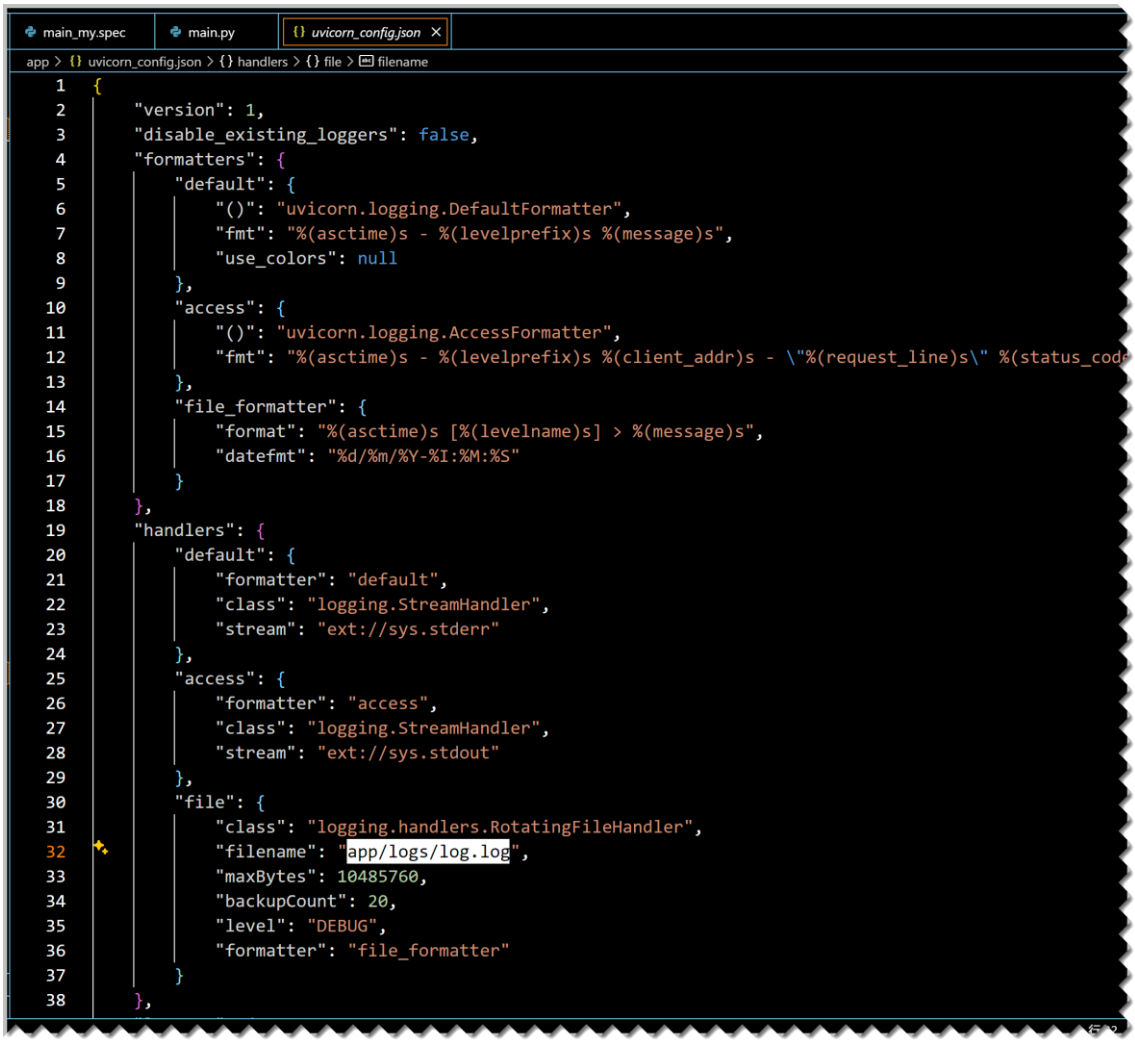
用途：

- Uvicorn 的 **日志配置文件**，用于配置日志格式、日志级别、日志输出位置（控制台或文件）。
- 文件中可以定义多个 **handler**（StreamHandler、FileHandler 等），以及日志文件名和路径。

- 在开发环境下，uvicorn.Config(log_config="app/uvicorn_config.json") 会读取这个文件。
- 在打包环境下，通常需要通过 resource_path() 或动态修改 filename 来保证路径正确，因为 PyInstaller 的 exe 会改变目录结构。

注意：

- filename 相对路径在 PyInstaller 打包后可能失效，需要动态替换成绝对路径或者使用 _internal/logs 统一目录。
- 这个文件通常不需要改动，只要代码在运行前处理路径即可。



3 main_my.spec

用途：

- PyInstaller 的 **打包说明文件**（spec 文件），控制 **如何把 Python 项目打包成可执**

行文件或目录。

- 里面指定了：
    - Analysis：哪些脚本需要分析，依赖哪些库
    - datas：哪些静态文件需要打包（templates、static、JSON、images 等）
    - hiddenimports：有些库 PyInstaller 静态分析不到，需要显式导入
    - EXE：生成 exe 文件的配置
    - COLLECT：生成松散目录结构的配置（多文件模式）

```
a = Analysis(
    ['app/main.py'],
    datas=[('app/uvicorn_config.json','app'), ('app/templates/*','app/templates')
    hiddenimports=['uvicorn','fastapi'],
)

exe = EXE(a.scripts, ...)
coll = COLLECT(exe, a.binaries, a.datas, ...)
```

注意：

- .spec 文件是 **打包流程的核心配置文件**，修改后需要用 pyinstaller main_my.spec 来打包。
- datas 指定的文件路径会影响 exe 运行时资源的加载路径（比如 _internal/app/uvicorn_config.json）。
- 它不会执行 Python 逻辑，只是描述打包规则。

当 PyInstaller 打包时，它默认只会分析 Python 代码的依赖模块，而不会自动包含图片、HTML 模板、配置文件等静态资源。

这时，就需要用 `--add-data`（或在 `.spec` 文件的 `datas` 中定义）告诉它要额外打包哪些文件或目录。

这里不介绍命令行的方式，只介绍`.spec` 文件写法：

```
a = Analysis(
    ["app/main.py"],
    pathex=[],
    binaries=[],
    datas=[ #要额外打包哪些文件或目录
        ("app/uvicorn_config.json", "app"),
        ("app/.env", "."),
        ("app/images/*", "app/images"),
        ("app/templates/*", "app/templates"),
        ("app/uploadfiles/*", "app/uploadfiles"),
        ("app/logs/*", "app/logs"),
    ],
    hiddenimports=[
        "uvicorn", "fastapi", "pydantic", "aiomysql", 'asyncio',    # 确保依赖被正确包含
    ],
    hookspath=[],
```

PyInstaller 在打包时，会分析你的 Python 源代码（AST）来判断使用了哪些模块。但有些模块是**动态导入的**（例如通过 importlib 或字符串导入），它就可能漏掉。

解决办法：用 --hidden-import 或者.spec 文件中指定 hiddenimports 集合，告诉 PyInstaller 把这些模块也打包进去，如上所示。

三者关系总结:

文件	类型	作用	与其他文件关系
`app/main.py`	Python 脚本	应用入口、定义路由和启动逻辑	PyInstaller 分析打包的核心脚本，会加载 `uvicorn_config.json`
`app/uvicorn_config.json`	JSON 配置	日志配置文件	被 `main.py` 加载，控制 Uvicorn 日志输出；路径可能需要通过 spec 或 resource_path 处理
`main_my.spec`	PyInstaller 配置文件	打包规则	告诉 PyInstaller 哪些脚本和数据文件打包，以及 exe/目录生成方式

理解重点：

1. main.py → 运行时入口
2. uvicorn_config.json → 日志配置资源
3. .spec → 打包规则、控制最终 exe/目录结构

4.4.2. PyInstaller 打包生成的目录结构

你可以手动修改 `.spec` 文件来添加资源文件、修改导入模块、定制输出路径等。

你可以通过编辑`.spec` 文件，在 EXE、COLLECT 和 BUNDLE 块下添加一个 `name=`，为

PyInstaller 提供一个更好的名字，以便为应用程序（和 `dist` 文件夹）使用。

EXE 下的名字是*可执行文件*的名字，BUNDLE 下的名字是应用程序包的名字。

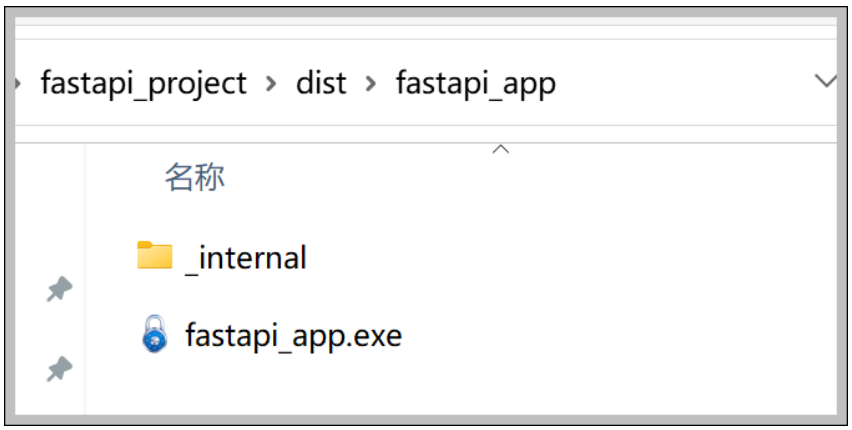
修改完成后，执行以下命令来重新打包：

pyinstaller main_my.spec

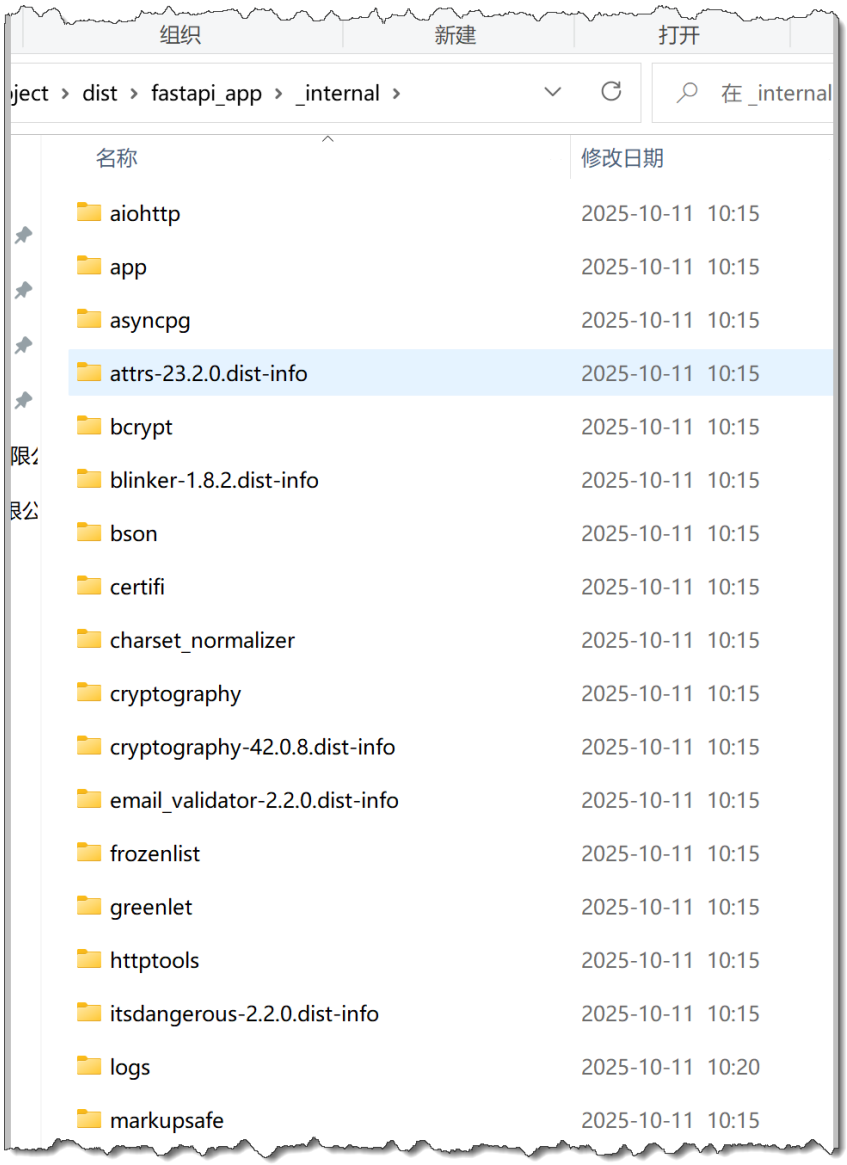
如果我们想在 Windows 平台生成的 dist 目录中生成一个启动 exe，和其他相关的 Lib 依赖库目录，那么我们可以适当调整下.spec 文件，让它可以生成松散结构的文件目录包。

```
exe = EXE(
    pyz,
    a.scripts,
    [],
    exclude_binaries=True,
    name="fastapi_app",
    debug=False,
    bootloader_ignore_signals=False,
    strip=False,
    upx=True,
    upx_exclude=[],
    runtime_tmpdir=None,
    console=True,           # True = 有控制台输出（调试方便），False = 静默运行
    onefile=False,  # <-- False取消、True使用 onefile 模式
    icon=icon,  # <-- 图标路径
    disable_windowed_traceback=False,
    argv_emulation=False,
    target_arch=None,
    codesign_identity=None,
    entitlements_file=None,
)

coll = COLLECT(
    exe,
    a.binaries,
    a.zipfiles,
    a.datas,
    strip=False,
    upx=True,
    name='fastapi_app'
)
```

相当于之前在 exe 包中的 a.binaries 和 a.datas 从 EXE 构造函数中移到了 Collect 的构造函数里面了。这样会生成下面的目录结构。

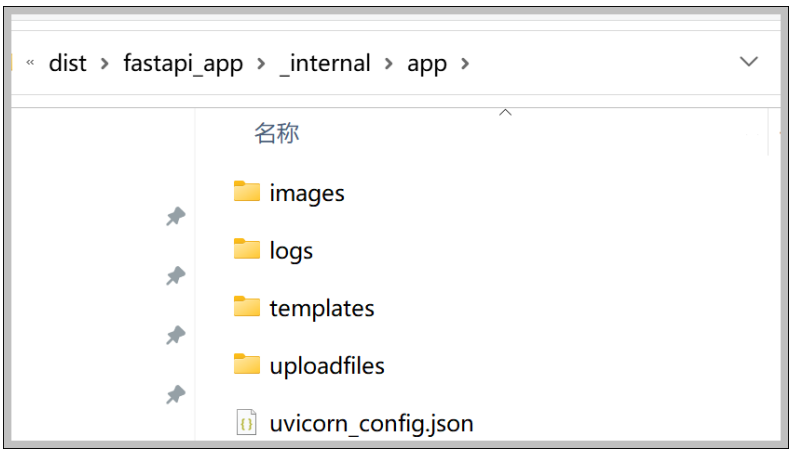


其中_internal 目录包含程序的相关依赖包和文件资源。



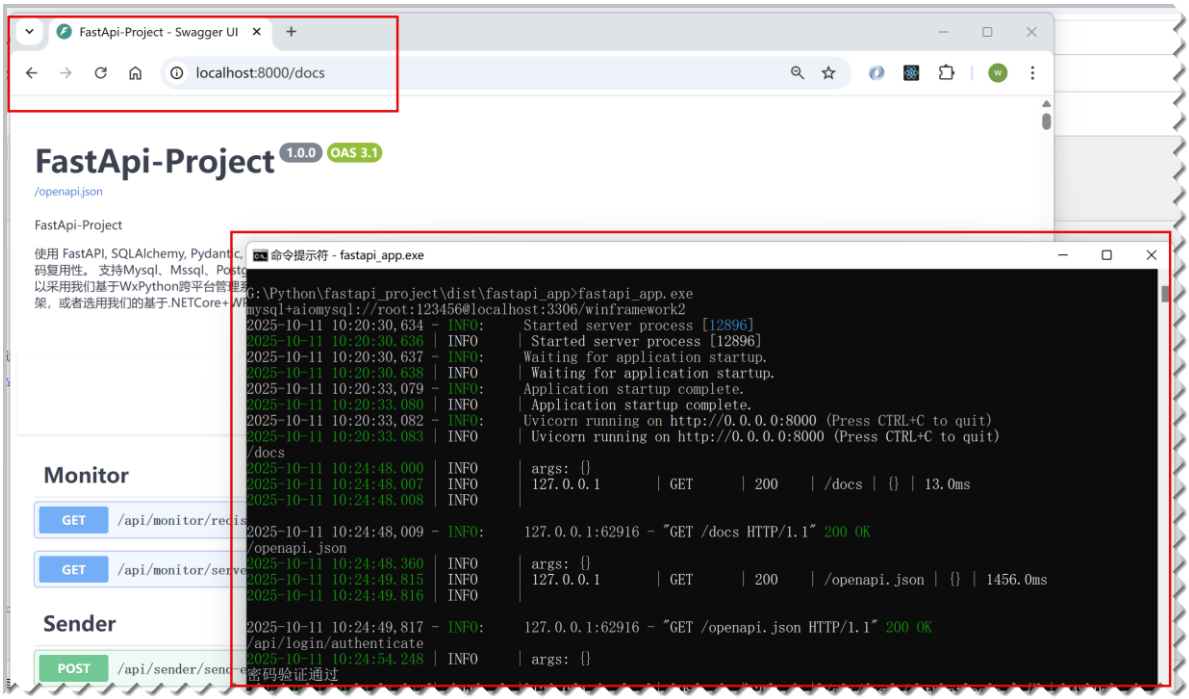
由于打包的.spec 文件指定的目录结构为松散结构（使用了 COLLECT 构造），那么可以看到

_internal / app 目录下有下面的目录结构。



也就是我们前面通过 Analysis 模块指定的 datas 集合路径的内容。

我们来看看 FastAPI 顺利启动后的效果。复制松散文件夹到服务器上双击运行即可，需要也可以修改配置文件.env 实现相关修改。



这样，我们在 dist 目录下生成的 exe 和松散目录一并复制到服务器上，直接双击或者命令行下运行 exe 文件就可以启动 FastAPI 服务，实现服务器端的更好部署方式了。