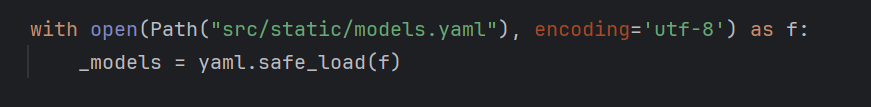
# Python基础

## Path类获取文件

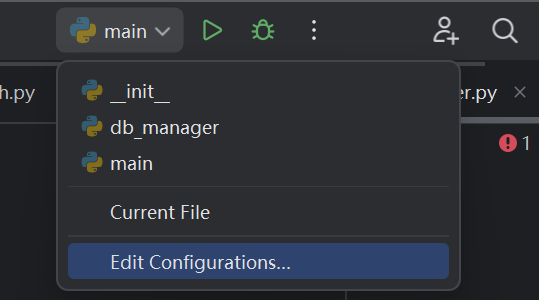


上面这是main.py中的一段代码。

上面的代码会在当前的工作目录下寻找src/static/models.yaml文件。

就比如你在控制台中执行python main.py，那么工作目录就是当前执行文件的目录

如果是在pycharm中点击启动按钮来启动main.py，那么工作目录可以Edit Configurations中可以设置并看到。



## \*与\*\*解包

## ‘海象运算符 :=

普通写法（没有 :=）

图形用户界面, 文本, 应用程序

AI 生成的内容可能不正确。

使用 := 的写法：



调用 self.\_get\_tools(conf.tools)，并将结果赋值给变量 tools

然后判断 tools 是否为真（非空、非 None 等）

如果为真，执行 model.bind\_tools(tools)

## 异步调用

图片包含 徽标

AI 生成的内容可能不正确。

在项目中对于网络请求、IO密集任务一般使用异步调用，这样在高并发情况下，当一个线程执行到此处时会暂停这个任务，转而去执行其它任务（比如处理其它用户发来的请求），当这个任务有结果返回时再接着这些这个任务，提高了线程的利用率。如果采用同步调用的话，在执行到此处时这个线程会暂停等待，也不会去执行其它任务仅仅只是等待，等待有返回结果了再继续往后执行。

下面是豆包的进一步解释：

同步调用：遇到 IO 密集任务（如网络请求）时，线程会被完全阻塞，期间什么都做不了，只能等待结果返回。

异步调用：遇到 IO 密集任务时，通过 await 让当前协程 “暂停”，但线程不会阻塞，而是由事件循环调度去执行其他就绪的协程（比如处理其他用户的请求），直到 IO 任务完成后再唤醒原协程继续执行。

这个对比是准确的，异步的核心价值就是在 IO 等待期间释放线程资源，提高线程利用率。

一、需要补充的细节

“线程暂停” 的准确含义

异步中，await 并不会让线程暂停，而是让当前协程暂停。线程本身一直在运行事件循环，事件循环会去调度其他协程。

例如：线程 A 运行事件循环，协程 X 执行到 await 网络请求 时，协程 X 暂停，事件循环会让线程 A 去运行协程 Y，直到网络请求完成后，协程 X 才会被唤醒继续执行。

（你的描述中 “线程会暂停这个任务” 可以更精确为 “协程暂停，线程继续运行事件循环”）

IO 任务的本质：不占用 CPU

IO 密集任务（如网络请求、文件读写）的核心操作是由操作系统内核处理的（比如网卡接收数据、磁盘读写），这个过程不占用 Python 线程的 CPU 时间。

同步调用：线程会傻傻地等待内核处理完成（期间什么都不做，浪费线程资源）。

异步调用：线程通过事件循环 “告知” 内核 “完成后通知我”，然后立刻去处理其他任务（充分利用 CPU 时间）。

这就是为什么异步在高并发 IO 场景下更高效 —— 它避免了线程在 IO 等待时的 “空转”。

同步调用的局限性

如果你在异步函数中用同步调用（比如 model.invoke() 而非 await model.ainvoke()），会导致：

线程被阻塞，事件循环无法调度其他协程，整个异步程序的并发能力退化到和同步一样。

甚至可能引发 “事件循环被阻塞” 的警告（如 RuntimeWarning: coroutine was never awaited），因为同步调用会打断事件循环的调度逻辑。

二、总结

你的核心理解是对的：异步通过 await 让协程在 IO 等待时 “让出” 线程，让线程去处理其他任务，而同步会阻塞线程直到 IO 完成。

补充细节后更完整的逻辑是：

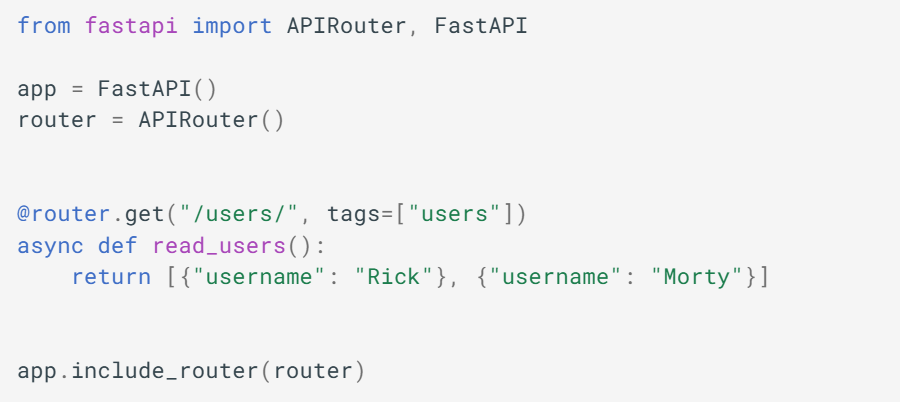
异步中，线程始终在运行事件循环，await 让当前协程暂停并将线程控制权交还给事件循环，事件循环调度其他协程；IO 任务由内核处理，完成后通知事件循环唤醒原协程。这一过程中线程从未阻塞，资源利用率更高。

因此，在高并发 IO 场景下，异步调用的优势是同步调用无法替代的。

# FastAPI学习

## APIRouter的使用

APIRouter 类，用于对 *路径操作* 进行分组，例如将应用程序结构化为多个文件。然后可以将其包含在 FastAPI 应用程序中，或者包含在另一个 APIRouter 中（最终都包含在应用程序中）。



## Ref的使用

ref 的作用是**创建一个响应式的引用（reactive reference）**，使得当这个引用的值发生变化时，Vue 能够自动追踪并更新相关的视图或副作用。

ref() 让我们能创造一种对任意值的 “引用”

# Sqlite学习

SQLite 是一个 嵌入式数据库，不同于 MySQL、PostgreSQL 等 客户端-服务器型数据库（如需提供 host、port、username、password 等连接信息），它直接以 文件形式 存储在本地磁盘中，所以连接时只需提供 数据库文件的路径 即可。

.db 文件是 SQLite 的“容器”，里面可以包含任意数量的表、视图、索引等数据库对象。相当于时mysql中的数据库。

## 连接sqlite数据库

可以使用sqlalchemy工具

from sqlalchemy import create\_engine

engine = create\_engine('sqlite:///server.db')

上面的代码就创建并连接了sqlite的server数据库了

## 创建表

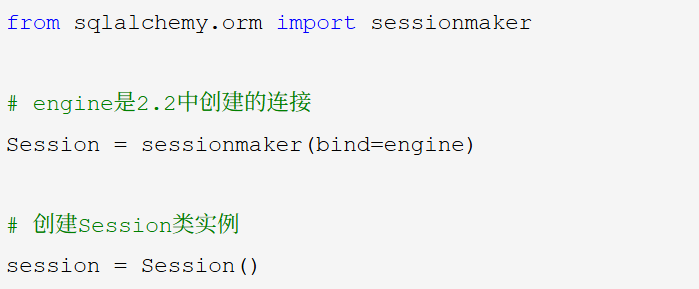
from sqlalchemy.ext.declarative import declarative\_base  
Base = declarative\_base()

Base.metadata.create\_all(engine)

在engine指定的server数据库中创建所有的表，扫描所有继承自 Base 的模型类，根据这些类的定义，自动在数据库中创建对应的表，如果表已经存在，则不会重复创建

## 建立会话

要对表进行增删改查就得通过会话



## 增

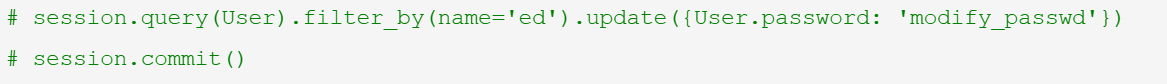


## 改



上边的操作，先查询再修改相当于执行了两条语句，和我们印象中的update不一致

# 可直接使用下边的写法，传给服务端的就是update语句



# 系统功能

## 登录功能

首先会先判断是否是第一次启动系统（通过查询数据库的user表，如果记录数count==0，说明是第一次），如果是第一次，那么会要求创建超级管理员账户

正常登录（不是第一次）

从请求中获取用户名和密码，查询数据库中的用户记录，验证密码是否匹配，匹配则更新user表对应记录的 last\_login 时间。生成 JWT access\_token。将登录日志保存到operation\_logs表中。返回 access\_token、用户信息等。

# 项目启动

## 前端

npm install pnpm

pnpm install

pnpm dev

# 项目简历

1. 使用JWT进行登录校验
2. AI安全之越狱风险