天津大学

《智能数据分析》帮助文档

一、 WSL 安装

1.1 系统准备

（1）检查系统版本

按 Win + R 输入 winver 查看版本信息，确保系统为 Windows 10（版本 2004 及以上，内部版本 19041+）或 Windows 11。

（2）启用虚拟化（BIOS 设置）

重启电脑进入 **BIOS**（开机时按 F2、F10、Del 等键，具体因电脑型号而异）。在 Advanced 或 Configuration 选项中，找到 Intel Virtual Technology 或 AMD-V，设置为 **Enabled**。

保存设置并退出 BIOS。

1.2 启用 WSL 功能

（1）以管理员身份打开 **PowerShell** 或 **CMD**，依次执行以下命令：

```
# PowerShell
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart

dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```

分别启用 **WSL** 功能与虚拟机平台（WSL 2 必需）

（2）按 Win + R 输入 optionalfeatures，勾选以下两项，之后重启电脑：

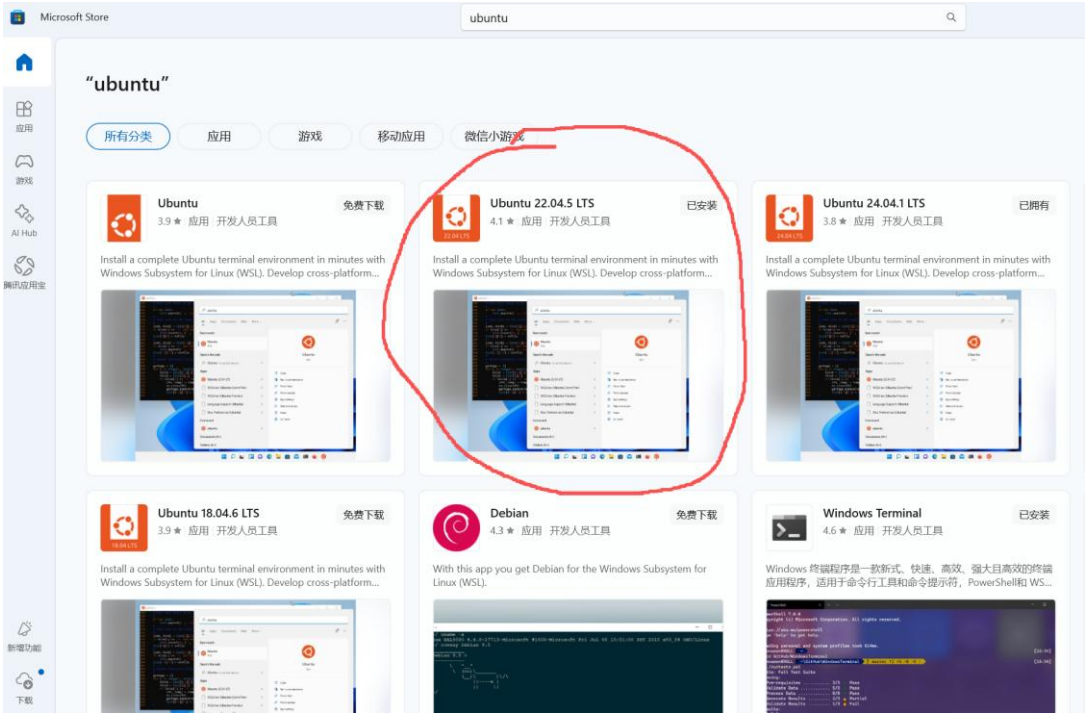
-- 适用于 Linux 的 Windows 子系统

-- 虚拟机平台

1.3 安装 Ubuntu 发行版

这里只介绍通过 Microsoft Store 安装。命令行安装容易因为网络问题安装不成功，手动安装较复杂。

（1）打开 Microsoft Store，搜索 **Ubuntu**



（2）选择 **Ubuntu 22.04.5 LTS**，点击获取并安装。
安装完成后，点击启动，首次运行会解压文件（约 1-2 分钟）。

（3）设置默认版本为 WSL 2，打开 Powershell 输入：

```
# Powershell
wsl --set-default-version 2
```

（4）验证安装，Powershell 下分别输入

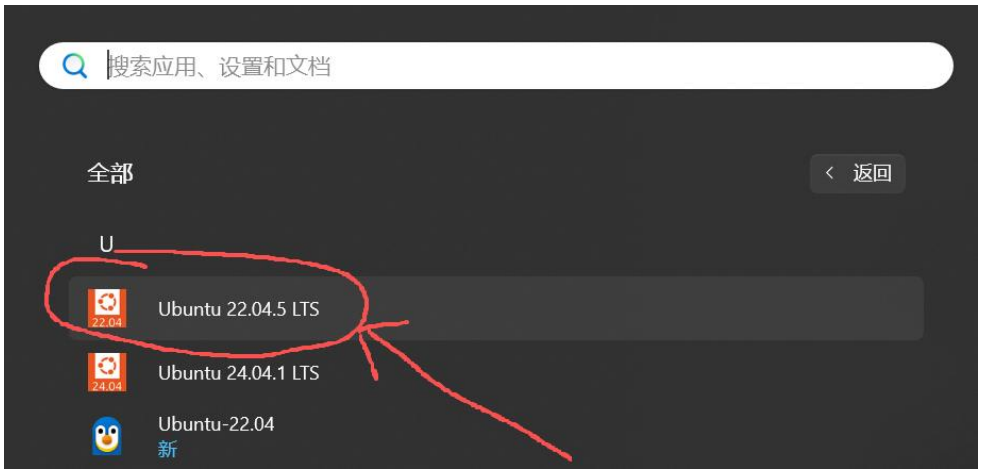
```
# Powershell
wsl --status
wsl -l -v
```

验证输出结果：

```
PS C:\Users\14767> wsl --status
默认分发: Ubuntu-22.04
默认版本: 2
PS C:\Users\14767> wsl -l -v
  NAME            STATE           VERSION
* Ubuntu-22.04    Running         2
```

1.4 启动 WSL

直接在开始菜单中打开：



或者按 win + s，搜索 Ubuntu 22.04

1.5 更新包

（1）打开 Ubuntu-22.04，按提示信息注册账户、设置密码（密码在输入时不可见），建议设的简短一点。

设置完成后出现类似以下信息

```
Installation successful!
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

Welcome to Ubuntu [redacted] LTS (GNU/Linux 6.6.87.2-microsoft-standard-WSL2 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro
```

（2）更新软件包列表，并安装解压包

```
# Bash
sudo apt update
sudo apt install unzip
```

二、前端运行

注意 以下操作都是在 Ubuntu 下进行

2.1 准备工作

（1）下载文件 **frontend.zip**，找到地址，对地址进行转换

```
# Eg. 地址转换
D:\BaiduNetdiskDownload\Edge\frontend.zip
转换为 /mnt/d/BaiduNetdiskDownload/Edge/frontend.zip
```

C 盘则转换为 /mnt/c/，注意"/"和"\"的方向

（2）打开 WSL 输入命令

```
# Bash
cp source ./
```

source 是转换后的地址。**cp** 是 copy 的简写，这条命令将 Windows 中的文件复制到 Ubuntu 中。

由于文件较大，等待时间可能较长。

（3）解压文件到当前文件夹（根目录）

```
# Bash
unzip frontend.zip
```

2.2 安装 Node.js 和 npm

(1) 换源 NodeSource，执行以下命令

```
# Bash
curl -fsSL https://deb.nodesource.com/setup_lts.x | sudo -E bash -
```

（2）安装 Node.js 和 npm：

```
# Bash
sudo apt update
sudo apt install -y nodejs
```

（3）验证安装，检查 Node.js 版本与 npm 版本

```
# Bash
node -v
npm -v
```

出现类似以下输出 则安装成功

```
xt@ROG:~/vue_frontend$ node -v
v22.20.0
xt@ROG:~/vue_frontend$ npm -v
10.9.3
```

2.3 启动前端

cd 进入解压后的文件夹，**npm run dev** 开启前端：

```
# Bash
cd vue_frontend/
npm run dev
```

如果出现类似错误：

```
> deepseek-client@0.0.0 dev
> vite

node:internal/modules/esm/resolve:274
    throw new ERR_MODULE_NOT_FOUND(
          ^

Error [ERR_MODULE_NOT_FOUND]: Cannot find module '/home/xt/vue_frontend/node_modules/dist/
me/xt/vue_frontend/node_modules/.bin/vite
    at finalizeResolution (node:internal/modules/esm/resolve:274:11)
    at moduleResolve (node:internal/modules/esm/resolve:859:10)
    at defaultResolve (node:internal/modules/esm/resolve:983:11)
    at #cachedDefaultResolve (node:internal/modules/esm/loader:731:20)
    at ModuleLoader.resolve (node:internal/modules/esm/loader:708:38)
    at ModuleLoader.getModuleJobForImport (node:internal/modules/esm/loader:310:38)
    at onImport.tracePromise.__proto__ (node:internal/modules/esm/loader:664:36)
    at TracingChannel.tracePromise (node:diagnostics_channel:344:14)
    at ModuleLoader.import (node:internal/modules/esm/loader:663:21)
    at defaultImportModuleDynamicallyForModule (node:internal/modules/esm/utils:222:31) {
  code: 'ERR_MODULE_NOT_FOUND',
  url: 'file:///home/xt/vue_frontend/node_modules/dist/node/cli.js'
}
```

按如下方式解决，**一定要进入到 vue_front 目录下**

（1）删除 node_modules 和锁文件，并清除 npm 缓存：

```
# Bash
rm -rf node_modules package-lock.json
npm cache clean -force
```

（2）之后重新安装依赖以及最新版 vite

```
# Bash
npm install
npm install vite@latest --save-dev
```

（3）再尝试运行

```
# Bash
npm run dev
```

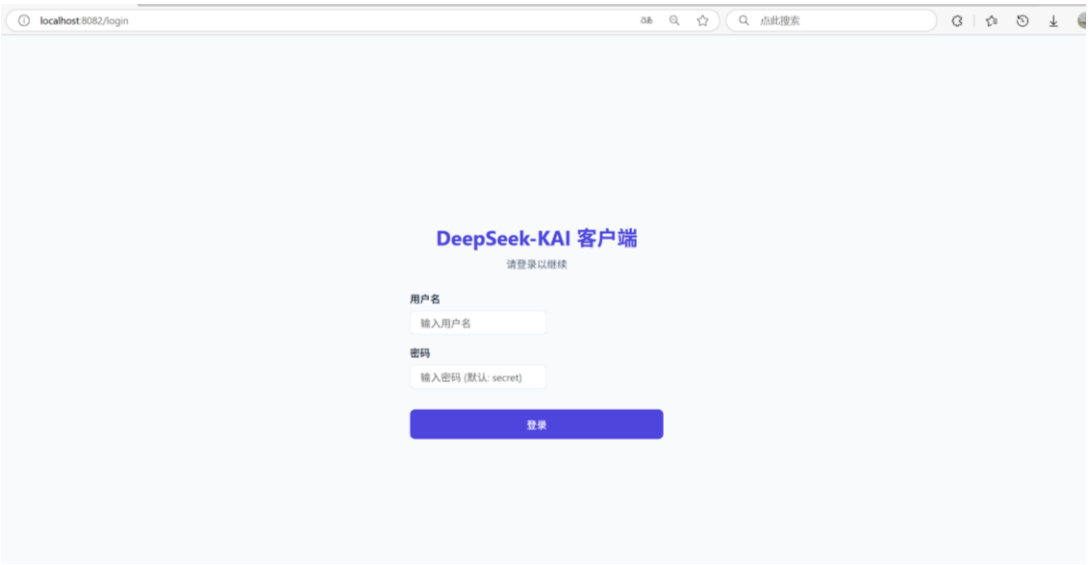
出现则运行成功：

```
VITE v7.1.7  ready in 188 ms

➜  Local:   http://localhost:8082/
➜  press h + enter to show help
```

按住 **ctrl** 鼠标单击 http://localhost:8082/，可以打开前端界面。

2.4 登录前端



用户名任意，密码为 **secret**

注意 这里需要启动后端，不然会出现登录失败的情况。

出现下面的界面则成功：

三、后端运行

3.1 下载 Miniconda

（1）进入到根目录，运行如下命令：

```
# Bash
mkdir -p ~/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O ~/miniconda3/miniconda.sh
bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3
rm ~/miniconda3/miniconda.sh
```

（2）等待 Miniconda 安装完成。安装完成后输入以下命令来刷新：

```
# Bash
source ~/miniconda3/bin/activate
```

此时用户名前会出现（base）字样，代表现在是在 conda 环境“base”下

（3）运行以下命令在所有可用的 shell 上初始化 conda：

```
# Bash
conda init --all
```

3.2 Miniconda 换源

（1）换源可以加速下载，解决下载错误的问题。使用以下命令加入源：

```
# Bash
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main/
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/r/
conda config --add channels
https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/msys2/
conda config --add channels
https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge/
```

（2）设置显示源地址

```
# Bash
conda config --set show_channel_urls yes
```

（3）查看当前配置，检查源是否已经加入：

```
# Bash
conda config --show-sources
```

3.3 在 Miniconda 中搭建 Python 环境

(1) 建立 conda 环境，版本选择 **python3.13**。将 **myenv** 替换为自己的命名。

```
# Bash
conda create --name myenv python=3.13
```

出现以下内容：

```
(base) linone@LAPTOP-LINONE:~$ conda create --name jiaoan1 python=3.10
2 channel Terms of Service accepted
Channels:
 - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/msys2
 - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/r
 - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
 - defaults
Platform: linux-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/linone/.conda/envs/jiaoan1

  added / updated specs:
    - python=3.10


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    python-3.10.18             |       h1a3bd86_0        26.5 MB  https://m
    setuptools-78.1.1          |  py310h06a4308_0         1.7 MB  https://m
    wheel-0.45.1               |  py310h06a4308_0         115 KB  https://m
    ------------------------------------------------------------
                                           Total:        28.4 MB
```

```
Proceed ([y]/n)? y
```

此时输入 **y** 并回车，出现以下内容即创建成功：

```
Downloading and Extracting Packages:

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate jiaoan1
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

（**2**）进入 conda 环境，将 **myenv** 替换为自己的名称，这里以 jiaoan1 为例

```
# Bash
conda activate myenv
```

```
(base) linone@LAPTOP-LINONE:~$ conda activate jiaoan1
(jiaoan1) linone@LAPTOP-LINONE:~$ cd Misc/
```

可以看到用户名前的“(base)”变为了“(jiaoan1)”，说明进入了环境

“jiaoan1”。所有的项目包都会安装在这个环境下。

3.4 安装所需包

```
# Bash
# pip 后加上 -i https://pypi.tuna.tsinghua.edu.cn/simple/ 进行换源

# Django
conda install django
conda install django-ninja
conda install django-cors-headers

# LangChain
conda install langchain
conda install langchain-core
conda install langchain-community

# Llama-Index
conda install llama-index
conda install llama-index-core

# chromadb
conda install chromadb
pip install llama-index-vector-stores-chroma

# Ollama
conda install ollama
pip install -U langchain-ollama
pip install llama-index-embeddings-langchain
pip install llama-index-llms-langchain
```

3.5 Ollama 拉取模型

（1）安装 ollama

```
# Bash
curl -fsSL https://ollama.com/install.sh | sh
```

验证是否安装成功，安装成功则显示版本号：

```
# Bash
ollama -v
```

（2）启动 ollama 服务器，会占用当前终端窗口

```
# Bash
ollama serve
```

```
(base) linone@LAPTOP-LINONE:~$ ollama serve
time=2025-09-25T12:48:59.480+08:00 level=INFO source=routes.go:1332 msg="server config" env="map[CUDA_VISIBLE_DEVICES: G
PU_DEVICE_ORDINAL: HIP_VISIBLE_DEVICES: HSA_OVERRIDE_GFX_VERSION: HTTPS_PROXY: HTTP_PROXY: NO_PROXY: OLLAMA_CONTEXT_LENG
TH:4096 OLLAMA_DEBUG:INFO OLLAMA_FLASH_ATTENTION:false OLLAMA_GPU_OVERHEAD:0 OLLAMA_HOST:http://127.0.0.1:11434 OLLAMA_I
NTEL_GPU:false OLLAMA_KEEP_ALIVE:5m0s OLLAMA_KV_CACHE_TYPE: OLLAMA_LLM_LIBRARY: OLLAMA_LOAD_TIMEOUT:5m0s OLLAMA_MAX_LOAD
ED_MODELS:0 OLLAMA_MAX_QUEUE:512 OLLAMA_MODELS:/home/linone/.ollama/models OLLAMA_MULTIUSER_CACHE:false OLLAMA_NEW_ENGIN
E:false OLLAMA_NOHISTORY:false OLLAMA_NOPRUNE:false OLLAMA_NUM_PARALLEL:1 OLLAMA_ORIGINS:[http://localhost https://local
host http://localhost:* https://localhost:* http://127.0.0.1 https://127.0.0.1 http://127.0.0.1:* https://127.0.0.1:* ht
tp://0.0.0.0 https://0.0.0.0 http://0.0.0.0:* https://0.0.0.0:* app://* file://* tauri://* vscode-webview://* vscode-fil
e://*] OLLAMA_SCHED_SPREAD:false ROCR_VISIBLE_DEVICES: http_proxy: https_proxy: no_proxy:]"
time=2025-09-25T12:48:59.483+08:00 level=INFO source=images.go:477 msg="total blobs: 8"
time=2025-09-25T12:48:59.483+08:00 level=INFO source=images.go:484 msg="total unused blobs removed: 0"
time=2025-09-25T12:48:59.484+08:00 level=INFO source=routes.go:1385 msg="Listening on 127.0.0.1:11434 (version 0.11.11)"
time=2025-09-25T12:48:59.485+08:00 level=INFO source=gpu.go:217 msg="looking for compatible GPUs"
time=2025-09-25T12:49:02.426+08:00 level=INFO source=gpu.go:388 msg="no compatible GPUs were discovered"
time=2025-09-25T12:49:02.426+08:00 level=INFO source=types.go:131 msg="inference compute" id=0 library=cpu variant="" co
mpute="" driver=0.0 name="" total="7.6 GiB" available="7.2 GiB"
time=2025-09-25T12:49:02.426+08:00 level=INFO source=routes.go:1426 msg="entering low vram mode" "total vram"="7.6 GiB"
threshold="20.0 GiB"
```

（3）另开一个终端窗口，拉取 **bge-large:latest** 和 **deepseek-r1:7b**

```
# Bash
ollama pull model_name
```

```
(base) linone@LAPTOP-LINONE:~$ ollama pull bge-large:latest
pulling manifest
pulling 92b37e50807d: 100%  670 MB
pulling a406579cd136: 100%  1.1 KB
pulling 917eef6a95d7: 100%  337 B
verifying sha256 digest
writing manifest
success
(base) linone@LAPTOP-LINONE:~$ ollama pull deepseek-r1:7b
pulling manifest
pulling 96c415656d37: 100%  4.7 GB
pulling c5ad996bda6e: 100%  556 B
pulling 6e4c38e1172f: 100%  1.1 KB
pulling f4d24e9138dd: 100%  148 B
pulling 40fb844194b2: 100%  487 B
verifying sha256 digest
writing manifest
success
```

（4） 测试模型

```
# Bash
ollama run deepseek-r1:7b question
```

```
(base) linone@LAPTOP-LINONE:~$ ollama run deepseek-r1:7b "请用简单的语言解释量子计算"
Thinking...
嗯，用户问的是“请用简单的语言解释量子计算”，看来他们可能对这个概念不太了解，想找个简明
，句子通顺，让用户容易理解。

量子计算是什么？它涉及到量子力学，比如超导、超冷原子这些现象。所以，或许可以从这些基础
只需要简单的解释。
```

3.6 测试后端核心

(1) 使用 cd 命令进入 Django 后端代码所在目录 /django_backend 下

```
(jiaoan1) linone@LAPTOP-LINONE:~/Misc$ cd ..
(jiaoan1) linone@LAPTOP-LINONE:~$ cd Misc/
(jiaoan1) linone@LAPTOP-LINONE:~/Misc$ cd django_backend/
(jiaoan1) linone@LAPTOP-LINONE:~/Misc/django_backend$
```

保证你有一个 Ubuntu 窗口开启了 ollama 服务器，并且 activate 了对应的 conda 环境，在当前目录下运行文件 **topklogsystem.py**

```bash
# Bash
python topklogsystem.py
```

日志索引建立过程：

```
 POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
Generating embeddings:  79%|                                                  | 450/570 [02:05<00:34,  3.45it/s]
 POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
Generating embeddings:  81%|                                                  | 460/570 [02:08<00:31,  3.45it/s]
 POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
Generating embeddings:  82%|                                                  | 470/570 [02:11<00:27,  3.58it/s]
 POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
Generating embeddings:  84%|                                                  | 480/570 [02:13<00:24,  3.64it/s]
 POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
Generating embeddings:  86%|                                                  | 490/570 [02:16<00:21,  3.70it/s]
 POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
Generating embeddings:  88%|                                                  | 500/570 [02:19<00:18,  3.69it/s]
 POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
Generating embeddings:  89%|                                                  | 510/570 [02:21<00:16,  3.67it/s]
 POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
Generating embeddings:  91%|                                                  | 520/570 [02:24<00:13,  3.59it/s]
 POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
Generating embeddings:  93%|                                                  | 530/570 [02:27<00:11,  3.61it/s]
 POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
Generating embeddings:  95%|                                                  | 540/570 [02:30<00:08,  3.60it/s]
 POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
Generating embeddings:  96%|                                                  | 550/570 [02:33<00:05,  3.55it/s]
 POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
Generating embeddings:  98%|                                                  | 560/570 [02:36<00:02,  3.58it/s]
 POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
Generating embeddings: 100%|                                                  | 570/570 [02:38<00:00,  3.59it/s]
INFO:__main__:日志库索引构建完成，共 570 条日志
INFO:httpx:HTTP Request: POST http://127.0.0.1:11434/api/embed "HTTP/1.1 200 OK"
```

LLM 返回对应的结果（不一定相同）：

```
#### 1. 立即采取的措施
- **检查并增加连接池的最大连接数**: 确保连接池能够支持预期的最高并发请求。
- **排查QuotaMgr配置**: 确认QuotaMgr是否正确释放连接，并允许配额重置。
- **优化数据库查询**: 使用更高效的查询策略，减少对数据库资源的消耗。

#### 2. 短期修复方案
- **增加连接池的最大连接数**: 将Alpha36和Alpha44的服务最大连接数从当前值提升到更高的值（例如100）
- **限制等待队列长度**: 设置合理的等待队列长度，避免长时间的阻塞。

#### 3. 长期预防措施
- **定期监控QuotaMgr状态**: 确保QuotaMgr能够正确释放连接池资源，并在需要时重置配额。
- **优化数据库查询**: 采用分页、缓存等技术减少对数据库的频繁连接请求。
- **使用更智能的数据库工具**: 如A+，以提高查询效率和优化资源使用。

### 预警建议
1. **监控指标建议**:
   - 监控Alpha36和Alpha44服务的当前连接数、等待队列长度等指标。
2. **告警阈值建议**:
   - 设置连接池当前连接数低于50时触发重置配额的操作。
3. **预防措施建议**:
   - 定期检查QuotaMgr的日志，确保其正常运行。
   - 使用性能分析工具（如A+）优化数据库查询。

通过以上措施，可以有效解决数据库连接池耗尽的问题，并提升系统的整体性能和稳定性。
```

四、运行前后端

4.1 运行 Ollama 服务器

打开一个 Ubuntu 窗口，启动 Ollama 服务器

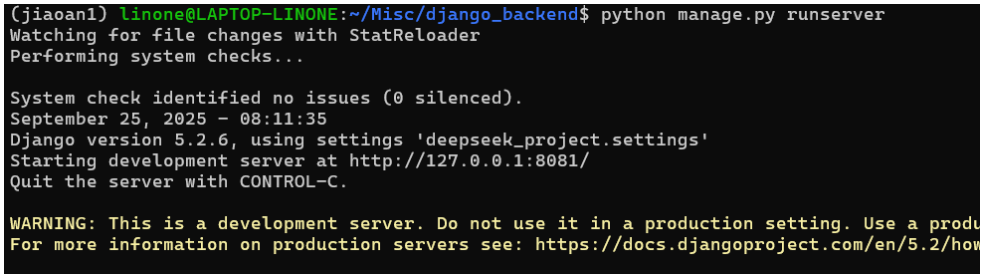
```
# Bash
ollama serve
```

4.2 运行 Django 后端

另开一个 Ubuntu 窗口，**确保进入 conda 环境并且正在运行 Ollama 服务器**，进入目录 **/django_backend** 下（manage.py 所在目录）输入运行指令

```
# Bash
python manage.py runserver
```

运行效果如下：

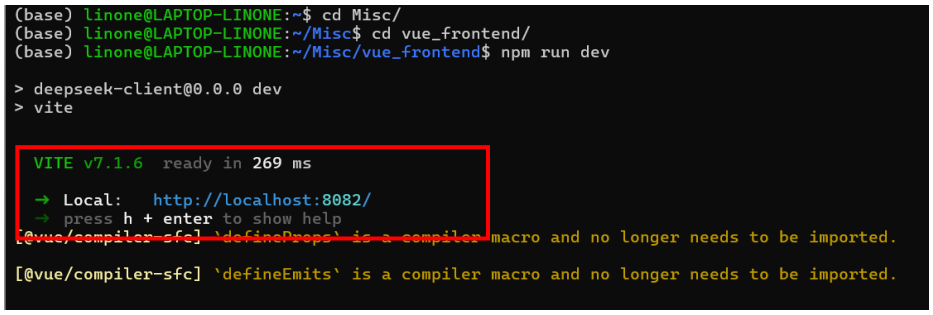


4.3 运行 vue 前端

另开一个 Ubuntu 窗口(总共开启 3 个)，进入目录 **/vue_front** 中，运行

```
# Bash
npm run dev
```

运行效果如下：

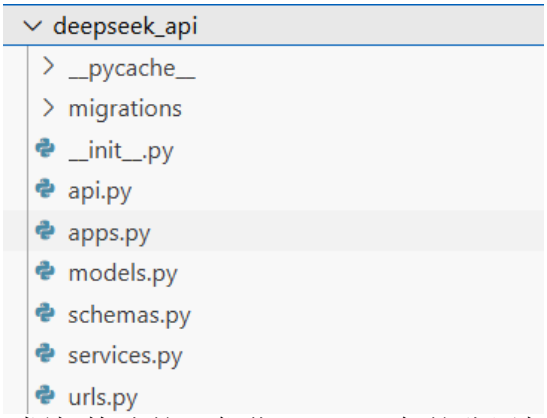


ctrl + 左键单击 URL，进入前端界面：

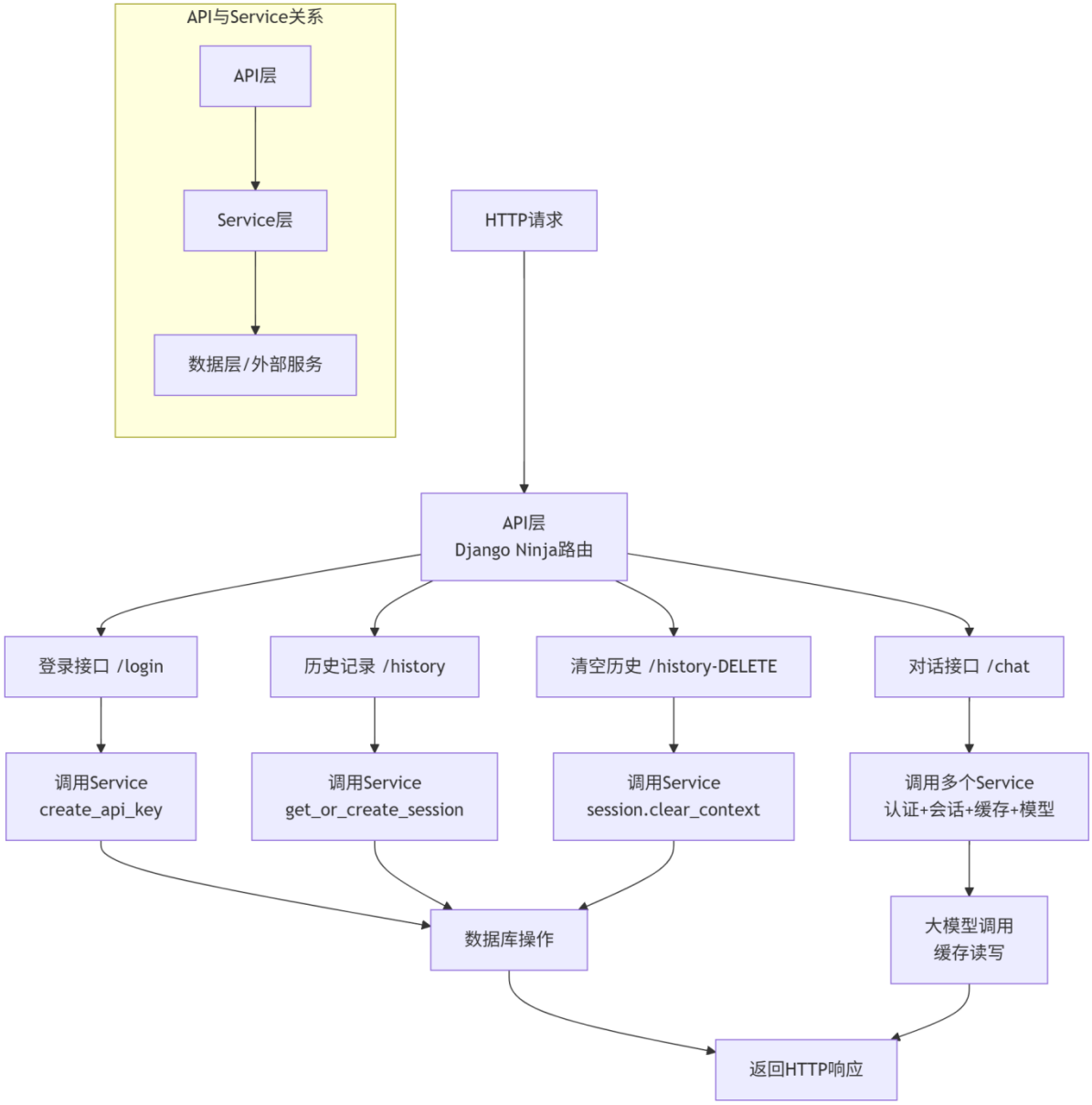
输入用户名（任意）和密码（默认 secret），登录系统

输入日志、问题，等待回复：

附录一　Django 框架主要文件说明



基于 Django Ninja 框架构建的现代化 Web 服务的分层架构和请求处理流程



一、api.py

模块一：api_key_auth 函数

功能：该函数作为自定义认证逻辑，被挂载到 router 上。当有请求访问该路由器下的任何接口（如 /chat）时，Django Ninja 会首先执行此函数.

工作流程：

（1）提取头信息：从 AuthorizationHTTP 头中获取值。

（2）解析格式：检查其是否符合 Bearer <api_key>的标准格式。

（3）验证有效性：在数据库的 APIKey 模型中查询该密钥是否存在且有效。

（4）返回结果：认证成功则返回 APIKey 对象（可通过 request.auth 访问），失败则返回 None（Django Ninja 将自动返回 401 未授权响应）

模块二：登录接口（/login）

此接口是用户获取访问凭证（API Key）的入口。

路径与方法：POST /api/login。它被直接注册在 api 实例上，因此无需认证即可访问。

工作流程：

（1）接收数据：通过 LoginInSchema（Pydantic 模型）接收用户名和密码，Django Ninja 会自动进行数据验证。

（2）业务逻辑验证：检查用户名和密码是否为空，并验证密码（示例中为固定的 "secret"）。（3）签发密钥：调用 services.create_api_key(username)生成一个与用户绑定的新 API Key 并返回，其中包含密钥和过期时间

模块三：核心对话接口（/chat）

这是整个服务的核心，实现了带上下文记忆的智能对话。

路径与方法：POST /api/chat。此接口位于 router 下，因此必须携带有效的 API Key 才能访问。

工作流程：

（1）认证检查：确认 request.auth 存在，即用户已登录。

（2）参数处理：清理 session_id 和 user_input，确保输入有效。

（3）会话管理：根据 session_id 和用户身份，获取或创建一个对话会话（Session）。这是实现多轮对话和上下文隔离的关键。

（4）上下文拼接：从会话中获取历史对话记录（pure_context），并将其与当前用户输入拼接成一个完整的提示（prompt）发送给大模型。这确保了模型能理解对话的上下文。

（5）调用大模型：在调用前，先检查是否有缓存回复，以提高响应速度并节约成本。若无缓存，则调用 deepseek_r1_api_call 函数获取模型回复。

（6）保存上下文：将本次的“用户输入-模型回复”对追加到会话的历史记录中，并保存到数据库，从而更新对话上下文。

模块四：历史记录管理接口（/history）

这两个接口用于管理用户的对话历史。

（1）查看历史（GET /api/history）：根据提供的 session_id，返回该会话的完整对话历史（即 session.context）。
（2）清空历史（DELETE /api/history）：根据提供的 session_id，清空该会话的对话历史（例如调用 session.clear_context()方法）。

模块五：路由注册
将定义好的路由器挂载到主 API 实例上，使所有接口生效。

二、service.py

模块一：deepseek_r1_api_call(prompt: str) -> str
核心功能：实际调用 DeepSeek-R1 模型的函数
工作流程：初始化日志系统，将拼接好的提示词发送给大模型，返回模型生成的文本回复

模块二： create_api_key(user: str) -> str
核心功能：为用户创建新的 API 密钥
工作流程：生成随机密钥字符串，设置过期时间，在数据库中创建 APIKey 记录，并为该密钥创建对应的速率限制记录
关联配置：使用 settings.TOKEN_EXPIRY_SECONDS 设置密钥有效期

模块三：validate_api_key(key_str: str) -> bool
核心功能：验证 API 密钥是否有效且未过期
工作流程：检查密钥是否存在，调用 api_key.is_valid()方法判断是否过期，自动删除过期密钥
安全机制：防止使用过期或无效的密钥访问服务

模块四： api_key_auth(request)
核心功能：Django Ninja 认证函数，验证请求头中的 API Key
工作流程：解析 Authorization: Bearer <api_key>格式，验证密钥有效性
返回值：认证成功返回 APIKey 对象，失败返回 None（触发 401 错误）

模块五：check_rate_limit(key_str: str) -> bool
核心功能：检查 API 密钥的请求频率是否超限
算法实现：使用令牌桶算法，每分钟最多 RATE_LIMIT_MAX 次请求
线程安全：使用 rate_lock 确保高并发下的计数准确性
重置机制：超过 RATE_LIMIT_INTERVAL 时间窗口后重置计数器

模块六：get_or_create_session(session_id: str, user: APIKey) -> ConversationSession
核心功能：获取或创建用户的专属会话，实现多轮对话上下文管理
隔离机制：根据 session_id 和 user 双重标识隔离不同用户的不同会话
上下文维护：返回的会话对象包含历史对话记录，保证对话连贯性

模块七：get_cached_reply(prompt: str, session_id: str, user: APIKey) -> str | None
核心功能：从缓存中获取之前相同请求的回复
缓存键设计：包含用户、会话 ID 和提示词哈希，避免跨会话缓存冲突
性能优化：避免重复调用大模型，减少响应时间和 API 成本

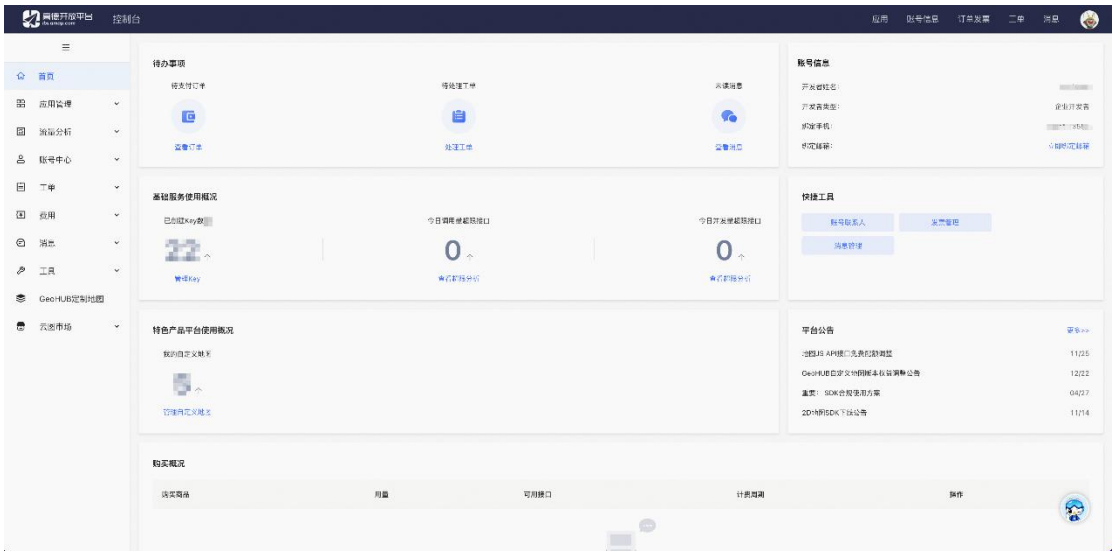
模块八：set_cached_reply(prompt: str, reply: str, session_id: str, user: APIKey, timeout=3600)
核心功能：将模型回复缓存指定时间（默认 1 小时）
缓存策略：使用 Django 缓存框架，支持 Redis 或内存缓存后端

模块九：generate_cache_key(original_key: str) -> str
核心功能：生成安全的缓存键
技术实现：使用 SHA256 哈希算法将任意字符串转换为固定长度安全标识
优势：避免缓存键过长或包含不安全字符，确保键的唯一性和安全性

附录二　LLM 工作流实现

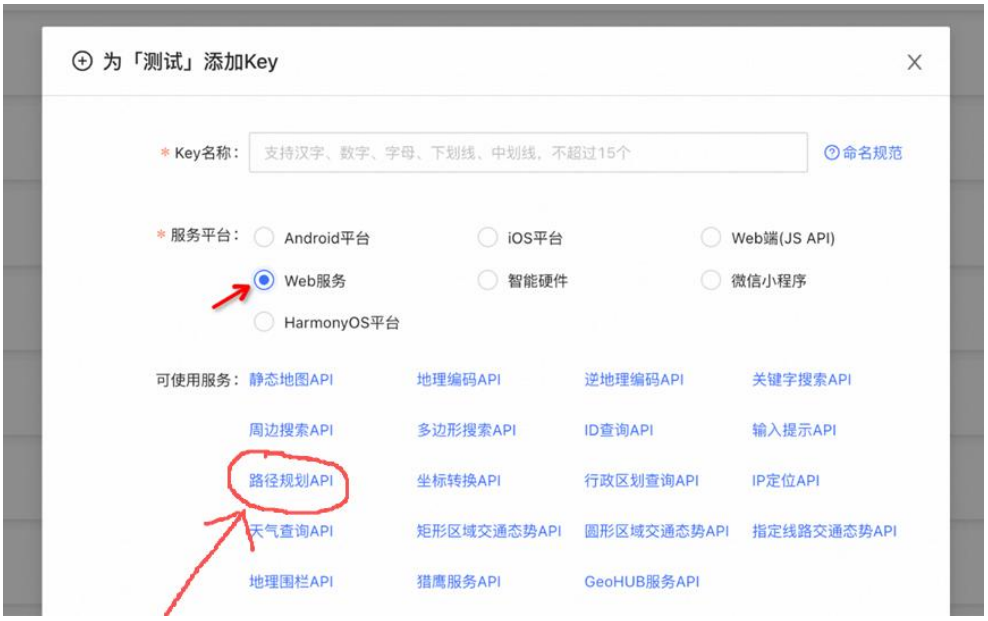
一、　申请高德开放平台 API key

(1) 登录高德开放平台控制台，如果没有开发者账号，请注册成为开发者。

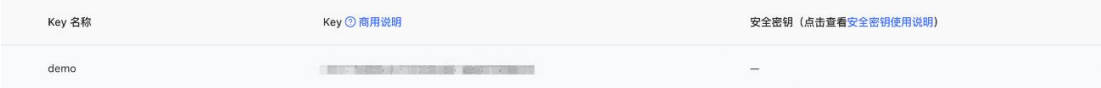


（2）进入**应用管理**，点击页面右上角**创建新应用**，填写表单。

（3）创建 Key：进入**应用管理**，在我的应用中选择需要创建 Key 的应用，点击**添加 Key**，表单中的服务平台选择 **Web 服务、路径规划 API**。



(4) 创建成功后，可获取 **Key** 和**安全密钥。**

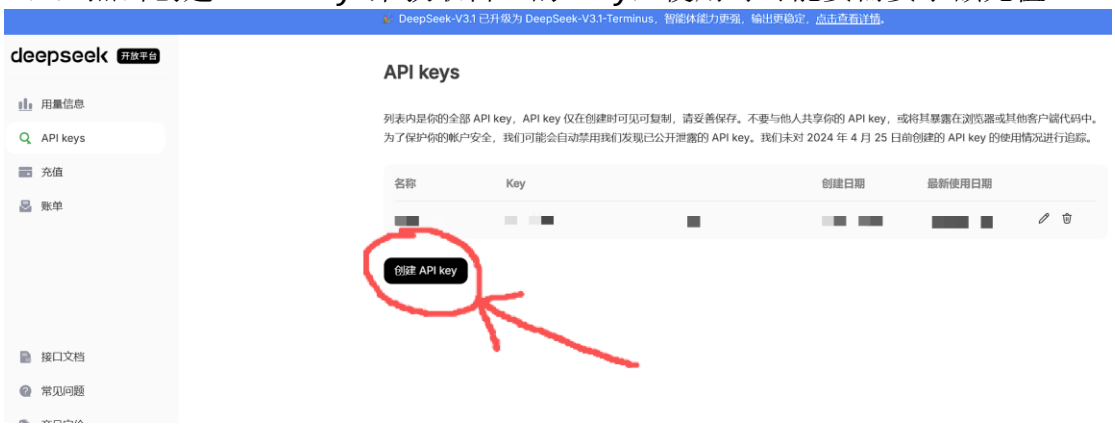


请妥善保管你的 Key。

二、 申请 DeepSeek API

（1）访问 [DeepSeek 开放平台](#)， 注册后点击左边的 **API keys**：

（2）点击创建 API key 来获取自己的 key，使用时可能要需要小额充值。



（3）DeepSeel API 的帮助文档：[首次调用 API | DeepSeek API Docs](#)
DeepSeek API 基础调用方法：

```
# Please install OpenAI SDK first: `pip3 install openai`
from openai import OpenAI

client = OpenAI(api_key="<DeepSeek API Key>",
base_url="https://api.deepseek.com")

response = client.chat.completions.create(
    model="deepseek-chat",
```

```
    messages=[
        {"role": "system", "content": "You are a helpful assistant"},
        {"role": "user", "content": "Hello"},
    ],
    stream=False
)

print(response.choices[0].message.content)
```

三、 工作流的实现

3.1 高德地图 API 初步使用

这是一个高德地图的应用示例，打印一个包含了从(117.314542,38.997899)到(117.170393,39.110190)路径的 json

源代码

```
import requests
import json

def main():
    api_key = 'your_api_key'
    origin = '117.314542,38.997899'
    destination = '117.170393,39.110190'
    url =
f"https://restapi.amap.com/v3/direction/walking?origin={origin}&destination
={destination}&key={api_key}"

    payload = ""
    headers = {
        'Content-Type': 'application/json',
        'Accept': 'appliction/json'
    }

    access_token = requests.request("GET", url, headers=headers,
data=payload)
    print(access_token.text)

if __name__ == '__main__':
    main()
```

3.2 get_route() 获取路径

将上面的操作包装为一个函数，得到

源代码

```
import json
import requests

def get_route(origin="117.314542,38.997899",
destination="117.170393,39.110190"):
    api_key = 'your_api_key'
    url =
f"https://restapi.amap.com/v3/direction/walking?origin={origin}&destination
```

```
={destination}&key={api_key}"

    payload = ""
    headers = {
        'Content-Type': 'application/json',
        'Accept': 'appliction/json'
    }

    access_token = requests.request("GET", url, headers=headers,
data=payload)
    result = json.loads(access_token.text)['route']
    return result

result = eval(f"get_route()")
print(result)
```

3.3 get_location() 获取经纬度

get_route() 需要传入的参数是经纬度，很不方便，采用另一个调用来获取地点的经纬度：

源代码

```
import json
import requests

def get_location(address="天津大学北洋园校区"):
    api_key = 'your_api_key'
    url =
f"https://restapi.amap.com/v3/geocode/geo?address={address}&key={api_key}"

    payload = ""
    headers = {
        'Content-Type': 'application/json',
        'Accept': 'appliction/json'
    }

    access_token = requests.request("GET", url, headers=headers,
data=payload)
    result = json.loads(access_token.text)
    answer = result['geocodes'][0]['location']
    return answer

result = eval(f"get_location()")
# answer = json.loads(result)['geocodes'][0]['location']
print(result)
```

运行结果

117.314542,38.997899

3.4 工具流的初步

编写 prompt 来对 LLM 进行知识输入，使用思维链来给大模型提供对应的思考路线

源代码

```
from openai import OpenAI
from ds_api_settings import *
import json
import requests

client = OpenAI(api_key=KEY, base_url=URL)        # DeepSeek 的 API key

def send_messages(messages):
    response = client.chat.completions.create(
    model='deepseek-chat',
    messages=messages
    )
    return response.choices[0].message

def get_route(origin="119.300057,26.089245",
destination="119.306711,26.087856"):
    api_key = 'your_api_key'
    url =
f"https://restapi.amap.com/v3/direction/walking?origin={origin}&destination
={destination}&key={api_key}"

    payload = ""
    headers = {
        'Content-Type': 'application/json',
        'Accept': 'appliction/json'
    }

    access_token = requests.request("GET", url, headers=headers,
data=payload)
    result = json.loads(access_token.text)['route']
    return result

system_prompt = """
你在运行一个“思考” “工具调用” “响应”循环。每次只运行一个阶段

1. “思考”阶段：你要仔细思考用户的问题。
2. “工具调用”阶段：选择可以调用的工具，并且输出对应工具需要的参数。
3. “响应”阶段：根据工具调用返回的影响，回复用户问题。

已有的工具如下：
get_route:
e.g. get_route: 位置坐标，位置坐标
返回路径

Example:
question: 从 119.300057,26.089245 到 119.306711,26.087856 怎么走？
thought: 我应该调用工具查询这个路径
Action:
{
    "function_name": "get_route",
    "function_params": "'119.300057,26.089245', '119.306711,26.087856'"
}
Answer:
{'origin': '119.300057,26.089245', 'destination': '119.306711,26.087856',
'paths': [{'distance': '1056', 'duration': '845', 'steps': [{'instruction': '向北步行 28 米右转
', 'orientation': '北',
```

'road': [], 'distance': '28', 'duration': '22', 'polyline':
'119.300004,26.089223;119.300004,26.08924;119.299991,26.089479',
'action': '右转', 'assistant_action': [], 'walk_type': '0'}, {'instruction': '沿鼓东路向东步行
670 米右转',
'orientation': '东', 'road': '鼓东路', 'distance': '670', 'duration': '536',
'polyline':
'119.299987,26.089479;119.300074,26.089492;119.300074,26.089492;119.300681,26.089553;
119.300681,26.089553;119.301324,26.089644;119.301324,26.089644;119.301606,26.089683;119.301606
,26.089683;119.301719,26.089696;
119.301719,26.089696;119.302305,26.089757;119.302305,26.089757;119.302821,26.089813;119.302821
,26.089813;119.303299,26.08987;
119.303299,26.08987;119.303806,26.089931;119.303806,26.089931;119.303872,26.089935;119.303872,
26.089935;119.30408,26.089935;
119.304484,26.089974;119.304484,26.089974;119.304653,26.089991;119.304653,26.089991;119.304774
,26.090004;119.304774,26.090004;
119.305126,26.090048;119.305126,26.090048;119.305694,26.090152;119.305694,26.090152;119.305846
,26.090178;119.305846,26.090178;
119.306098,26.090221;119.306098,26.090221;119.306693,26.090343', 'action': '右转',
'assistant_action': [], 'walk_type': '0'},
{'instruction': '沿五四路向南步行 187 米右转', 'orientation': '南', 'road': '五四路', 'distance':
'187', 'duration': '150',
'polyline':
'119.306693,26.090343;119.30681,26.089965;119.30681,26.089965;119.306858,26.089818;119.306858,
26.089818;
119.306875,26.08977;119.306875,26.08977;119.306901,26.089679;119.306901,26.089679;119.306949,2
6.089553;119.306949,26.089553;
119.30707,26.089249;119.30707,26.089249;119.307148,26.089076;119.307148,26.089076;119.307309,2
6.088711', 'action': '右转',
'assistant_action': [], 'walk_type': '0'}, {'instruction': '向西南步行 27 米左转',
'orientation': '西南', 'road': [], 'distance': '27',
'duration': '22', 'polyline':
'119.307309,26.088707;119.30717,26.088641;119.30717,26.088641;119.307062,26.088598', 'action':
'左转',
'assistant_action': [], 'walk_type': '0'}, {'instruction': '向东南步行 77 米右转',
'orientation': '东南', 'road': [], 'distance': '77',
'duration': '62', 'polyline': '119.307057,26.088594;119.30737,26.087956', 'action': '右转',
'assistant_action': [], 'walk_type': '0'},
{'instruction': '向西步行 67 米到达目的地', 'orientation': '西', 'road': [], 'distance': '67',
'duration': '54',
'polyline':
'119.30737,26.087951;119.30727,26.087951;119.30727,26.087951;119.306836,26.087969;119.306697,2
6.08796;119.306697,26.08796;
119.306697,26.08796', 'action': [], 'assistant_action': '到达目的地', 'walk_type': '0'}]}]}
"""

```
question = """从 119.286612,26.062221 到 119.297300,26.083639 怎么走? """

messages = [
    {'role': 'system', 'content': system_prompt},
    {'role': 'user', 'content': question}
]

message = send_messages(messages)
response = message.content
try:
    action = response.split("Action:")[1]
except:
    action = response
```

```
action = json.loads(action)
print(f"ModelResponse:\n {action}")

function_name = action["function_name"]
function_params = action["function_params"]
code = f"{function_name}({function_params})"
print(code)

result = eval(code)
print(result)
```

运行结果

```
ModelResponse:
 {'function_name':            'get_route',            'function_params':
"'119.286612,26.062221', '119.297300,26.083639'"}
get_route('119.286612,26.062221', '119.297300,26.083639')
...具体的路径
```

可以看出 LLM 自动的调用 get_route() 方法来获取两个坐标间的路径

3.5 更进一步

prompt 中涉及多个函数的嵌套调用，给出相应的例子方便 LLM 进行理解。

源代码

```
from openai import OpenAI
from ds_api_settings import *
import json
import requests

client = OpenAI(api_key=KEY, base_url=URL)

def send_messages(messages):
    response = client.chat.completions.create(
    model='deepseek-chat',
    messages=messages
    )
    return response.choices[0].message

def get_location(address="福州东街口"):
    api_key = '65285d19de67b34debd054ff18c5c266'
    url =
f"https://restapi.amap.com/v3/geocode/geo?address={address}&key={api_key}"

    payload = ""
    headers = {
        'Content-Type': 'application/json',
        'Accept': 'appliction/json'
    }

    access_token = requests.request("GET", url, headers=headers,
data=payload)
    result = json.loads(access_token.text)
    answer = result['geocodes'][0]['location']
```

```
    return answer

def get_route(origin="119.286612,26.062221",
destination="119.297300,26.083639"):
    api_key = '65285d19de67b34debd054ff18c5c266'
    url =
f"https://restapi.amap.com/v3/direction/walking?origin={origin}&destination
={destination}&key={api_key}"

    payload = ""
    headers = {
        'Content-Type': 'application/json',
        'Accept': 'appliction/json'
    }

    access_token = requests.request("GET", url, headers=headers,
data=payload)
    result = json.loads(access_token.text)['route']
    return result


system_prompt = """
你在运行一个“思考”“工具调用”“响应”循环。每次只运行一个阶段

1. “思考”阶段：你要仔细思考用户的问题。
2. “工具调用”阶段：选择可以调用的工具，并且输出对应工具需要的参数。
3. “响应”阶段：根据工具调用返回的影响，回复用户问题。

已有的工具如下：
1. get_location:
e.g. get_location: 地点名
返回路径

Example:
question: 福州东街口的位置坐标是什么？
thought: 我应该调用工具查询这个位置坐标
Action:
{
    "function_name": "get_location",
    "function_params": "'福州东街口'"
}
Answer:
119.300057,26.089245


2. get_route:
e.g. get_route: 位置坐标，位置坐标
返回路径

Example:
question: 从福州东街口到福建省立医院怎么走？
thought: 我应该调用工具查询这个路径
Action:
{
    "function_name": "get_route",
    "function_params": "eval('get_location("福州东街口")'),
eval('get_location("福建省立医院")')"
}
```

```
Answer:
{'origin': '119.300057,26.089245', 'destination': '119.306711,26.087856',
'paths': [{'distance': '1056', 'duration': '845', 'steps': [{'instruction': '向北步行 28 米右转
', 'orientation': '北',
'road': [], 'distance': '28', 'duration': '22', 'polyline':
'119.300004,26.089223;119.300004,26.08924;119.299991,26.089479',
'action': '右转', 'assistant_action': [], 'walk_type': '0'}, {'instruction': '沿鼓东路向东步行
670 米右转',
'orientation': '东', 'road': '鼓东路', 'distance': '670', 'duration': '536',
'polyline':
'119.299987,26.089479;119.300074,26.089492;119.300074,26.089492;119.300681,26.089553;
119.300681,26.089553;119.301324,26.089644;119.301324,26.089644;119.301606,26.089683;119.301606
,26.089683;119.301719,26.089696;
119.301719,26.089696;119.302305,26.089757;119.302305,26.089757;119.302821,26.089813;119.302821
,26.089813;119.303299,26.08987;
119.303299,26.08987;119.303806,26.089931;119.303806,26.089931;119.303872,26.089935;119.303872,
26.089935;119.30408,26.089935;
119.304484,26.089974;119.304484,26.089974;119.304653,26.089991;119.304653,26.089991;119.304774
,26.090004;119.304774,26.090004;
119.305126,26.090048;119.305126,26.090048;119.305694,26.090152;119.305694,26.090152;119.305846
,26.090178;119.305846,26.090178;
119.306098,26.090221;119.306098,26.090221;119.306693,26.090343', 'action': '右转',
'assistant_action': [], 'walk_type': '0'},
{'instruction': '沿五四路向南步行 187 米右转', 'orientation': '南', 'road': '五四路', 'distance':
'187', 'duration': '150',
'polyline':
'119.306693,26.090343;119.30681,26.089965;119.30681,26.089965;119.306858,26.089818;119.306858,
26.089818;
119.306875,26.08977;119.306875,26.08977;119.306901,26.089679;119.306901,26.089679;119.306949,2
6.089553;119.306949,26.089553;
119.30707,26.089249;119.30707,26.089249;119.307148,26.089076;119.307148,26.089076;119.307309,2
6.088711', 'action': '右转',
'assistant_action': [], 'walk_type': '0'}, {'instruction': '向西南步行 27 米左转',
'orientation': '西南', 'road': [], 'distance': '27',
'duration': '22', 'polyline':
'119.307309,26.088707;119.30717,26.088641;119.30717,26.088641;119.307062,26.088598', 'action':
'左转',
'assistant_action': [], 'walk_type': '0'}, {'instruction': '向东南步行 77 米右转',
'orientation': '东南', 'road': [], 'distance': '77',
'duration': '62', 'polyline': '119.307057,26.088594;119.30737,26.087956', 'action': '右转',
'assistant_action': [], 'walk_type': '0'},
{'instruction': '向西步行 67 米到达目的地', 'orientation': '西', 'road': [], 'distance': '67',
'duration': '54',
'polyline':
'119.30737,26.087951;119.30727,26.087951;119.30727,26.087951;119.306836,26.087969;119.306697,2
6.08796;119.306697,26.08796;
119.306697,26.08796', 'action': [], 'assistant_action': '到达目的地', 'walk_type': '0'}]}]}
"""


question = """从天津大学北洋园校区到天津大学卫津路校区怎么走？"""

messages = [
    {'role': 'system', 'content': system_prompt},
    {'role': 'user', 'content': question}
]

message = send_messages(messages)
```

```python
response = message.content
try:
    action = response.split("Action:")[1]
except:
    action = response
action = json.loads(action)
print(f"ModelResponse:\n {action}")

function_name = action["function_name"]
function_params = action["function_params"]
code = f"{function_name}({function_params})"
print(code)

result = eval(code)
print(result)
```

运行结果

```
ModelResponse:
 {'function_name': 'get_route', 'function_params': 'eval(\'get_location("天津大学北洋园校区")\'), eval(\'get_location("天津大学卫津路校区")\')'}
get_route(eval('get_location("天津大学北洋园校区")'), eval('get_location("到天津大学卫津路校区")'))
...具体的路径
```

LLM 学习了我们的范例，根据具体的地名先调用 get_location() 再调用 get_route()，并且是用嵌套的方式去实现调用，最终得到具体的路径.