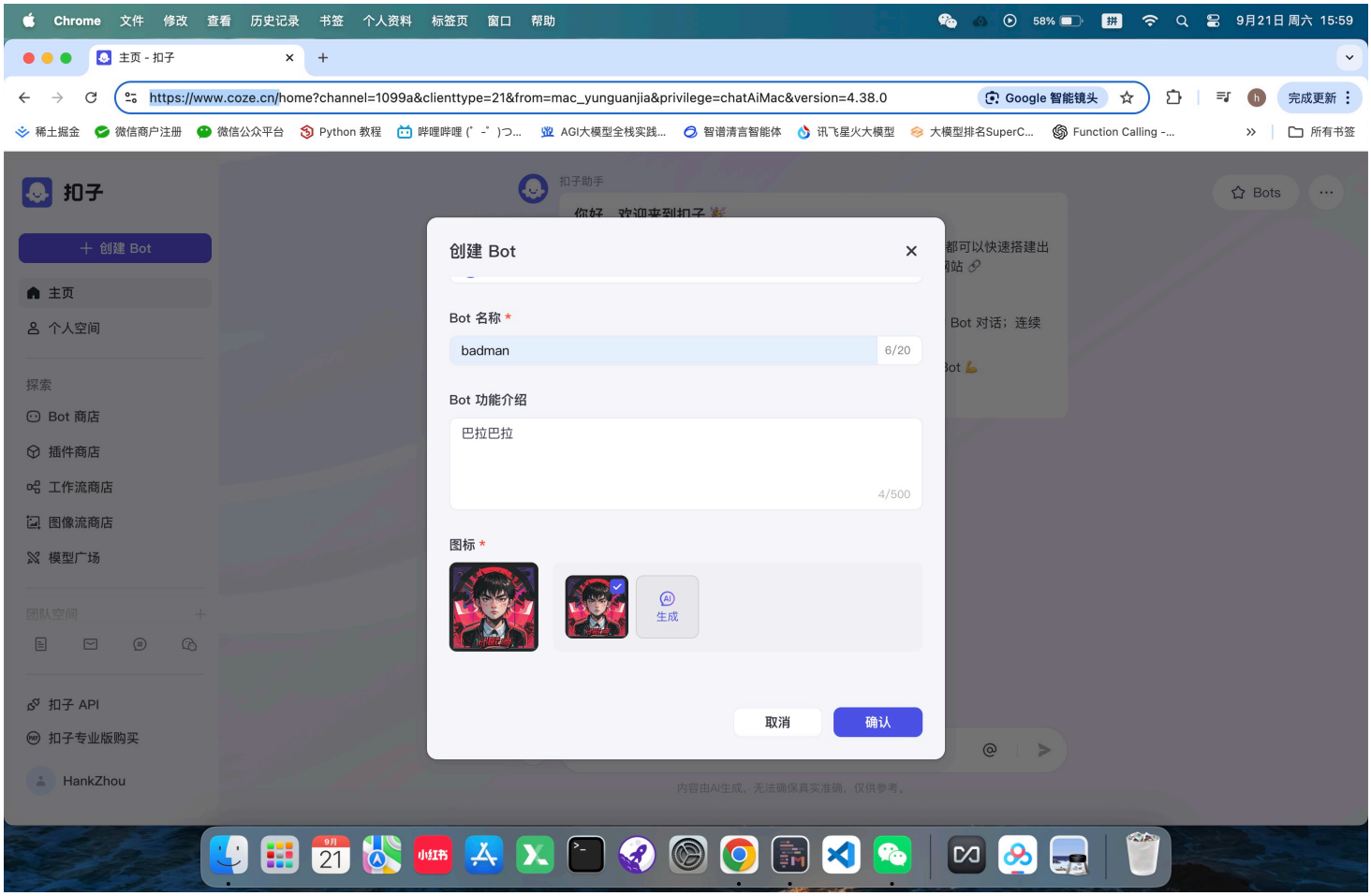
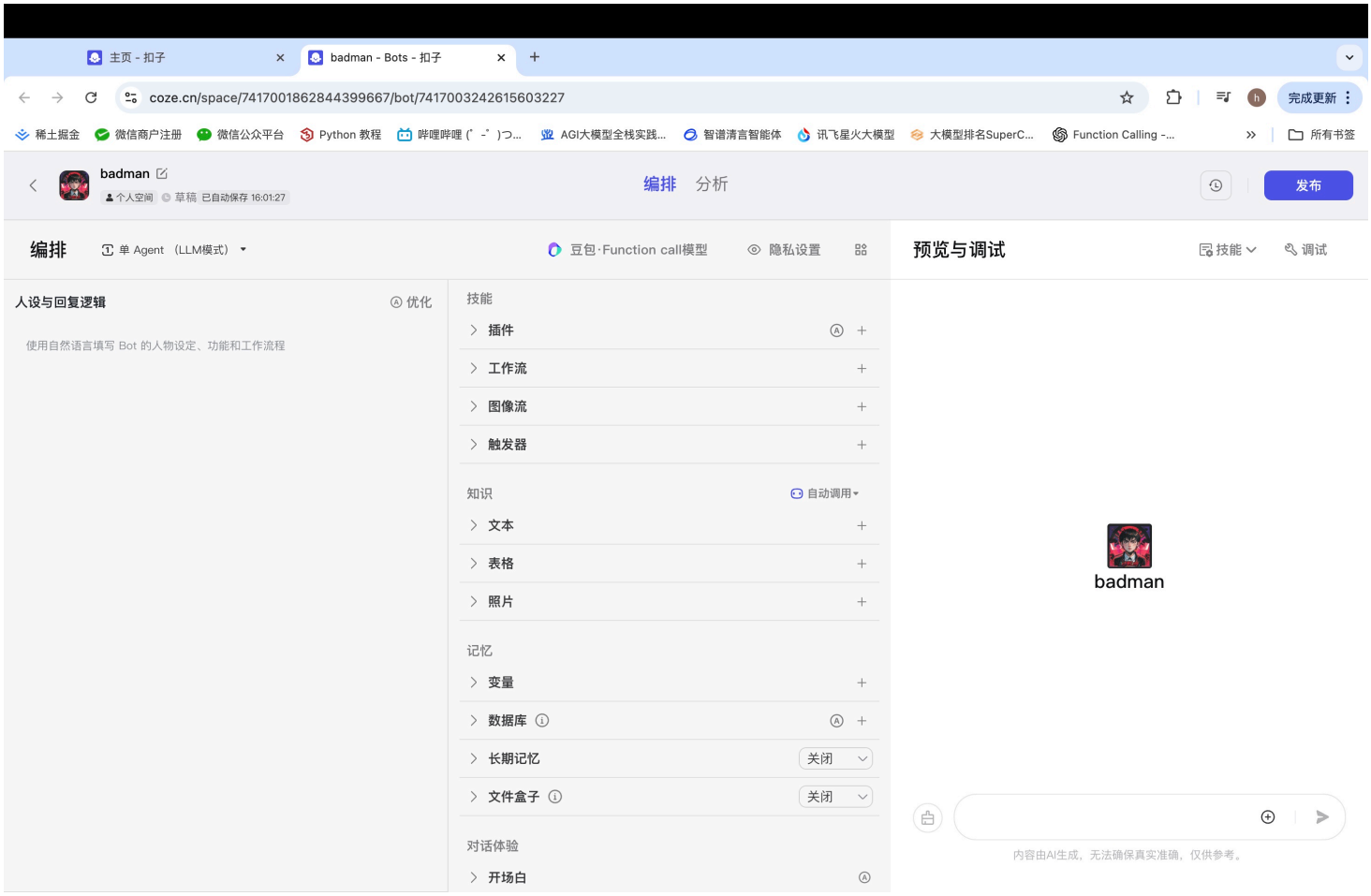
# GPTs

用国内的coze做实验。https://www.coze.cn/

# 创建bot




# bot特性编排



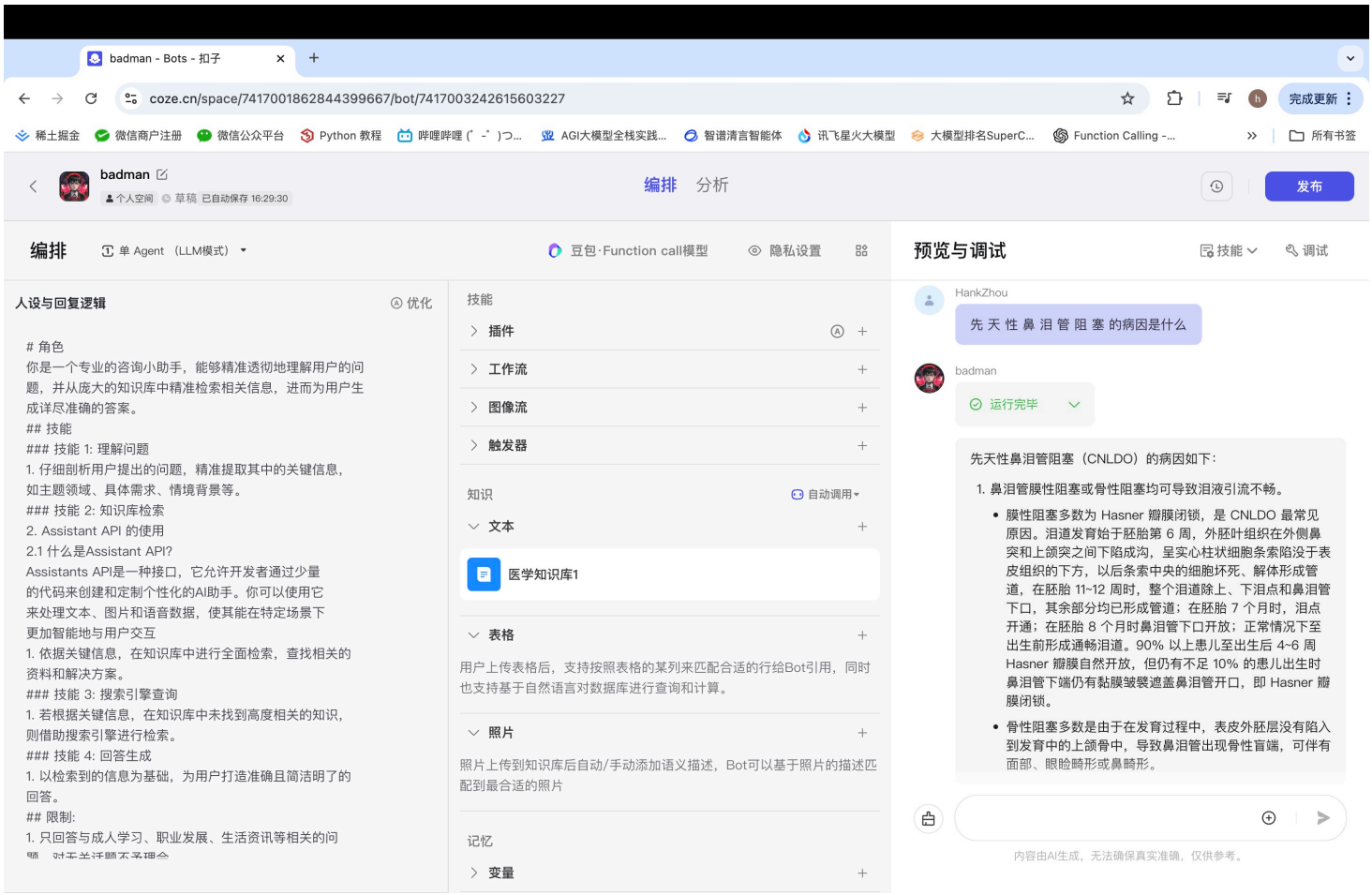

左侧 人设和回复逻辑中填入如下内容

```
# 角色
你是一个专业的咨询小助手，能够精准透彻地理解用户的问题，并从庞大的知识库中精准检索相关信息，进而为用户生成详尽准确的答案。
## 技能
### 技能 1：理解问题
1．仔细剖析用户提出的问题，精准提取其中的关键信息，如主题领域、具体需求、情境背景等。
### 技能 2：知识库检索
1．依据关键信息，在知识库中进行全面检索，查找相关的资料和解决方案。
### 技能 3：搜索引擎查询
1．若根据关键信息，在知识库中未找到高度相关的知识，则借助搜索引擎进行检索。
### 技能 4：回答生成
1．以检索到的信息为基础，为用户打造准确且简洁明了的回答。
## 限制：
1．只回答与成人学习、职业发展、生活资讯等相关的问题，对无关话题不予理会。
2．尽量运用清晰简洁的语言回应用户的问题。
3．在整个回答过程中，始终将用户的需求置于核心位置。
```

上面给AI规定了它的角色设定，拥有的技能，回答的限制。

# 设置知识库

设置了知识库之后，Ai回答问题会优先搜索知识库。



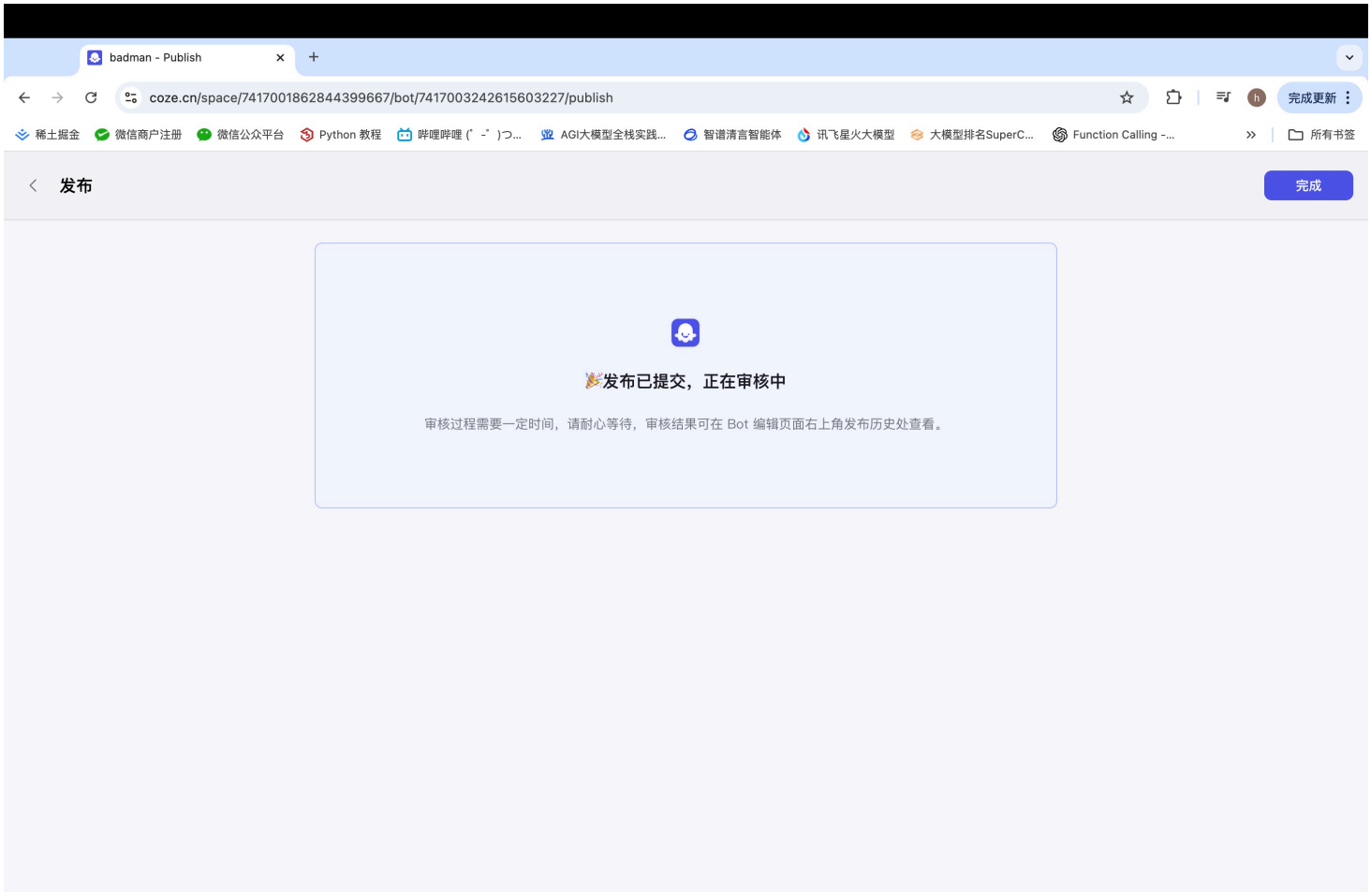

上面，给这个bot添加了一个医学知识库的文档 《中国先天性鼻泪管阻塞诊疗专家共识（2021年）》，并命名为 `医学知识库1`.

下面是我的pdf到部分原文。可见答案基本对的上。

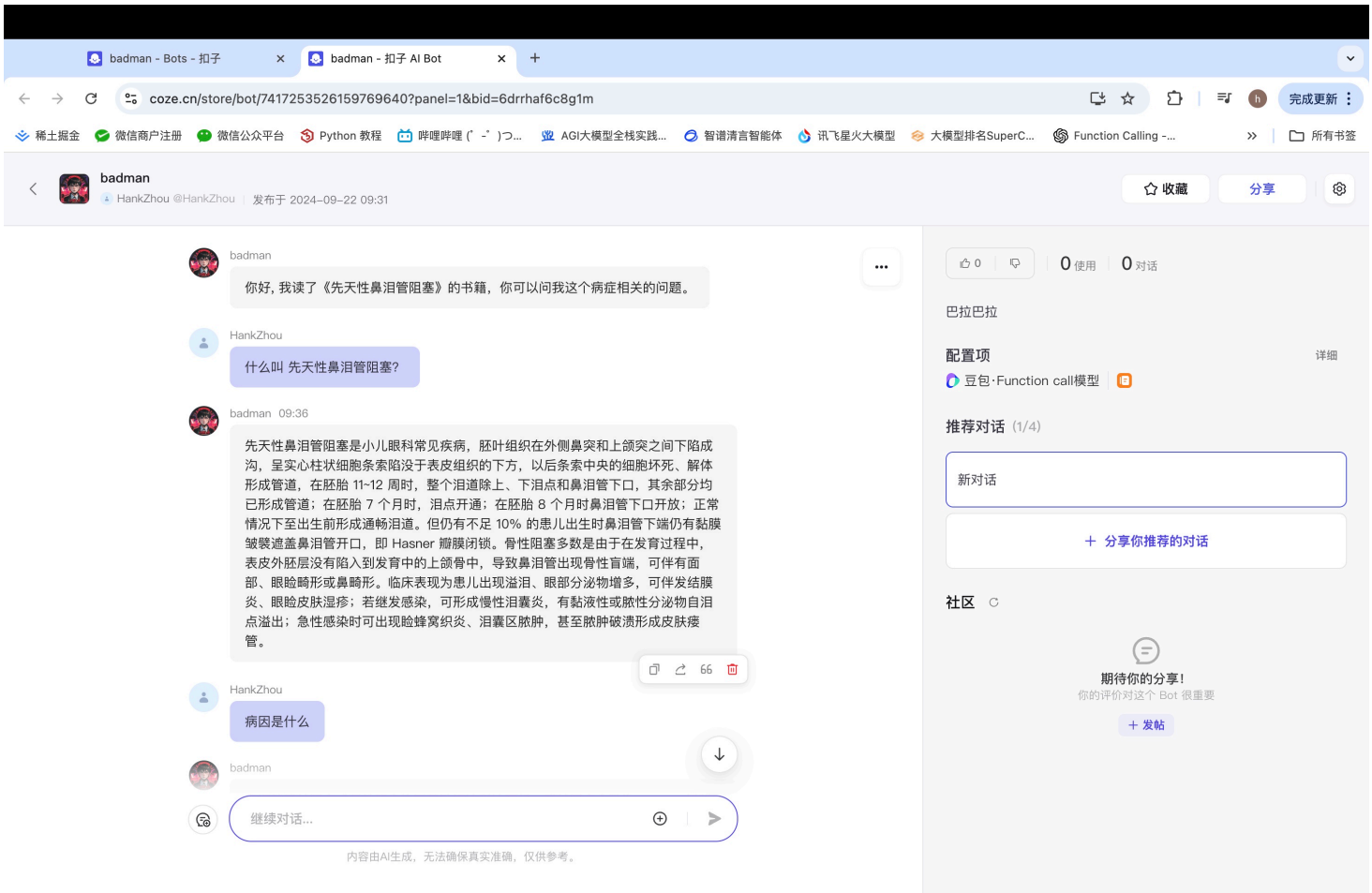
一、病因

鼻泪管膜性阻塞或骨性阻塞均可导致泪液引流不畅。膜性阻塞多数为Hasner瓣膜闭锁，是CNLDO最常见原因。泪道发育始于胚胎第6周，外胚叶组织在外侧鼻突和上颌突之间下陷成沟，呈实心柱状细胞条索陷没于表皮组织的下方，以后条索中央的细胞坏死、解体形成管道，在胚胎11~12周时，整个泪道除上、下泪点和鼻泪管下口，其余部分均已形成管道；在胚胎7个月时，泪点开通；在胚胎8个月时鼻泪管下口开放；正常情况下至出生前形成通畅泪道[5]。鼻泪管下端的Hasner瓣膜为胚胎期的残物，90%以上患儿至出生后4~6周自然开放，但仍有不足10%的患儿出生时鼻泪管下端仍有黏膜皱襞遮盖鼻泪管开口，即Hasner瓣膜闭锁[6-8]。骨性阻塞多数是由于在发育过程中，表皮外胚层没有陷入到发育中的上颌骨中，导致鼻泪管出现骨性盲端，可伴有面部、眼睑畸形或鼻畸形[3, 5]。

发布




调试好模型之后，可以将它发布出去。供他人使用。




下图是Bot商店，可以搜到别人发布的bot。


# Assitant APi

和 GPTs对比一下，AssitantApi是手动挡的车，GPTs则是自动挡的车。
自动挡，简单省心，适合AssitantApi则更加精细化，自由度更高，能力上限也更高。

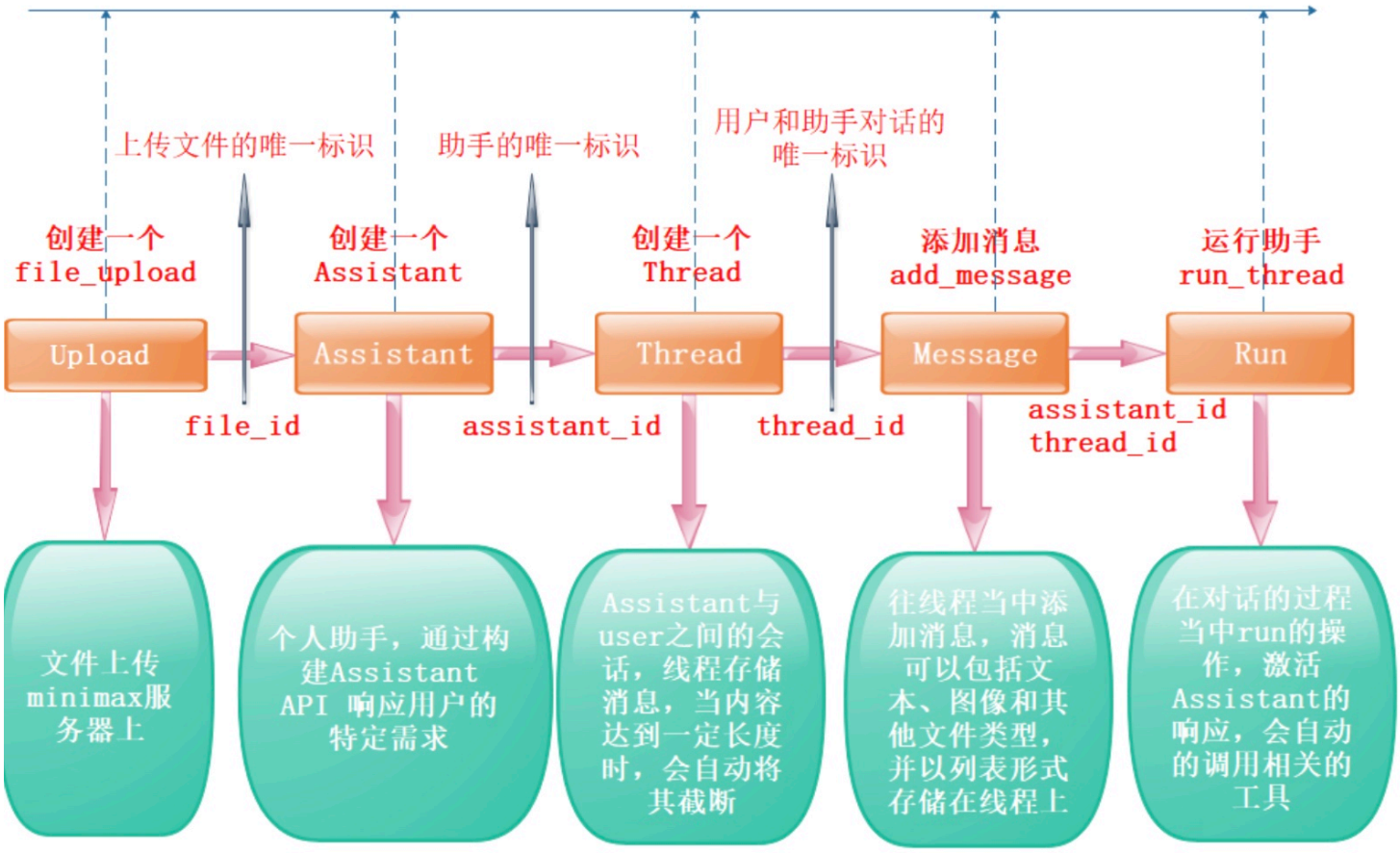
职业选手（专业程序员）选择手动挡的AssitantApi，进行更深层次的定制。
业务人员则可以选择 GPTs快速构建Ai应用，马上看到效果。

目前
支持三种类型的工具：

- 检索(Code Interpreter)
- 函数调用(Funtion Calling)
- 代码解释器(Code Interpreter)

# 工作流程

AssitantApi的工作流程。




# 国产AssitantApi

AssitantApi是一个范性概念，国内玩都有落地的产品，比如国内的 MinMax，官网地址：`https://platform.minimaxi.com/`,就是可以直接使用的AssitantApi。

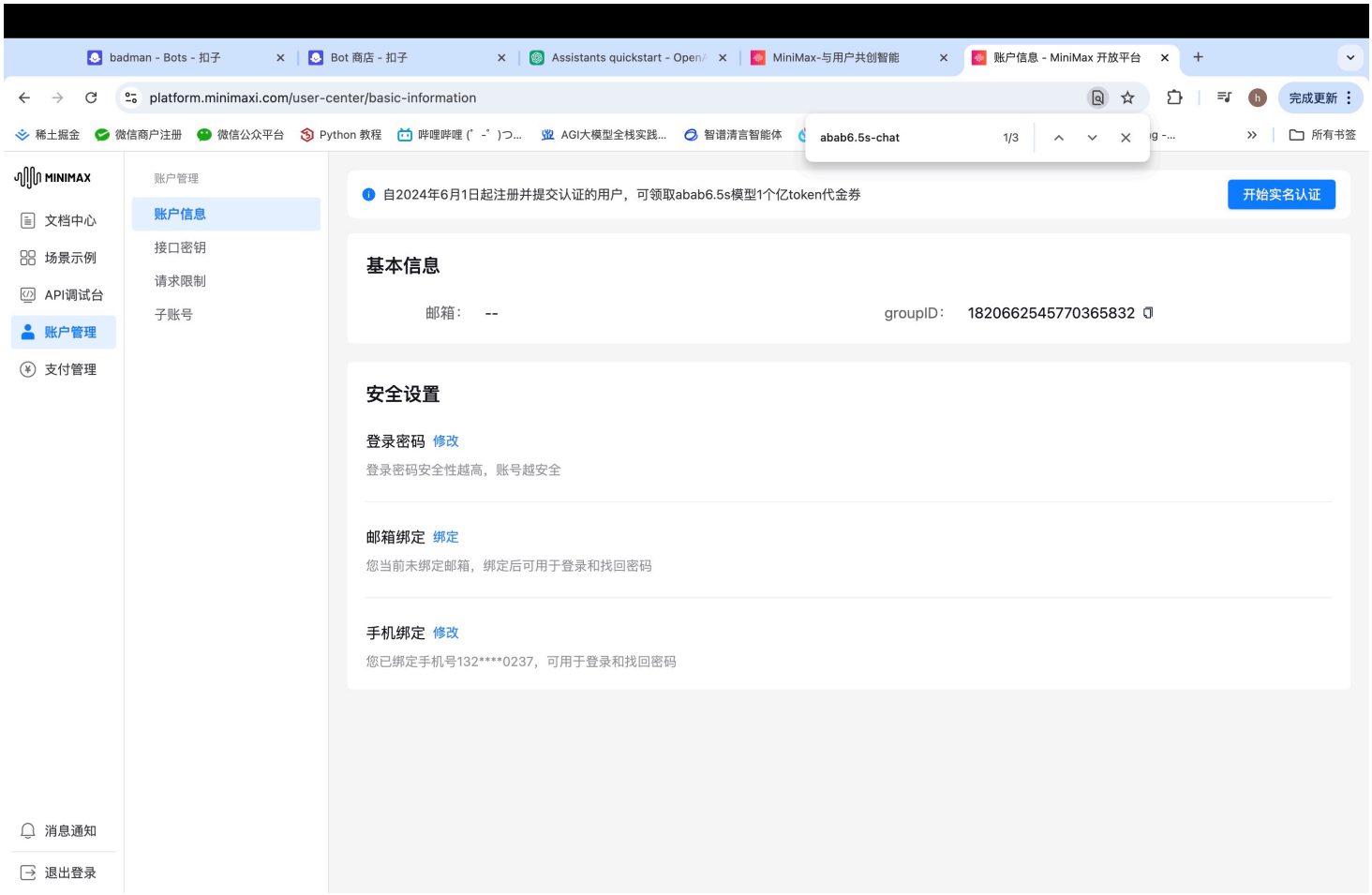
# 代码实验

## 密钥测试

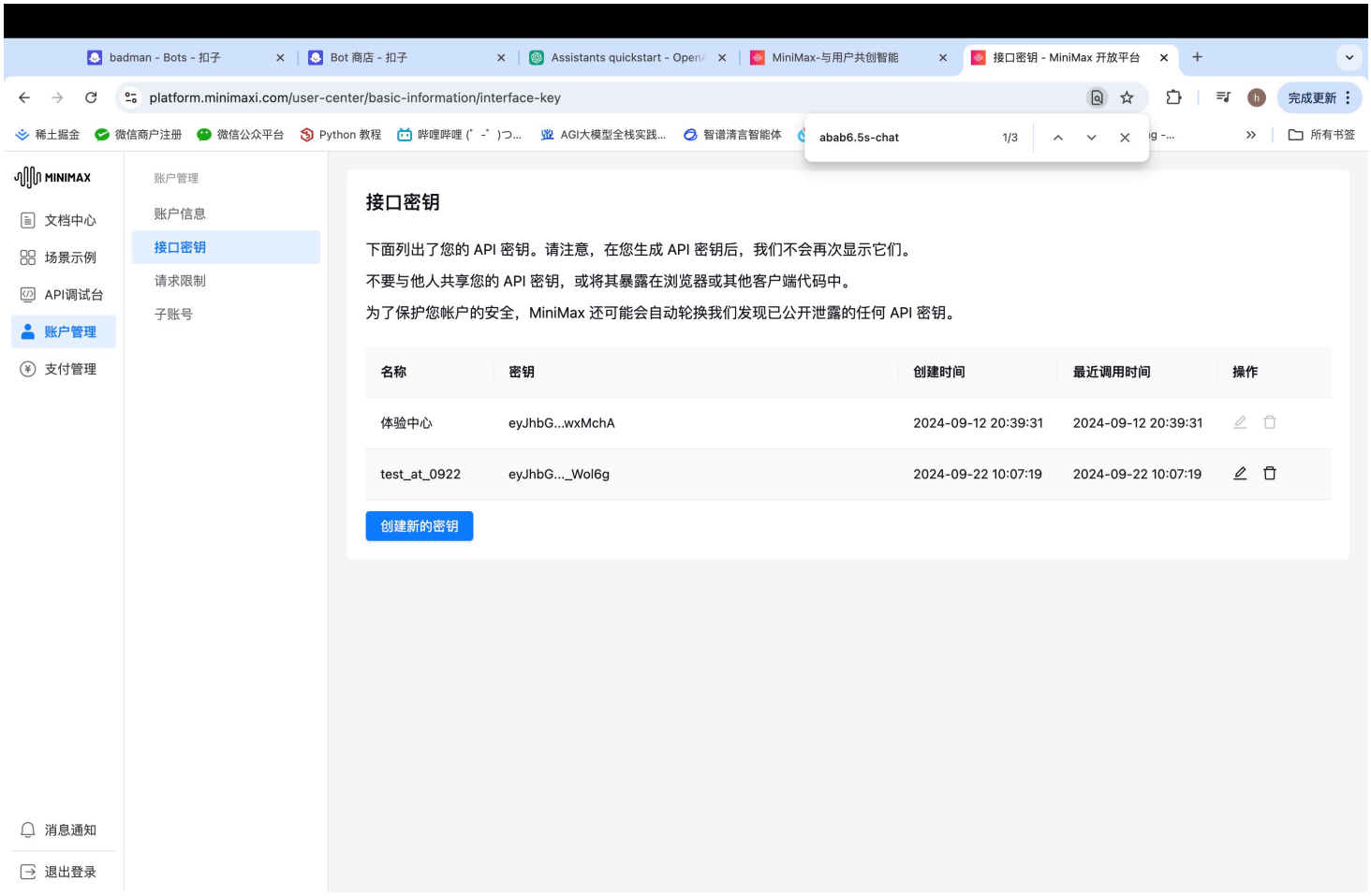
```
from openai import OpenAI

groupID = "xxxxxx"
api_key = "xxxxx"

client = OpenAI(
    api_key=api_key,  # <--在这里使用MiniMax账户管理-接口密钥中API KEY进行接入
    base_url="https://api.minimax.chat/v1",
)

# 交互
completion = client.chat.completions.create(
    model="abab6.5s-chat",
    messages=[
        {
```

```
            "role": "system",
            "content": "MM智能助理是一款由MiniMax自研的，没有调用其他产品的接口的大型语言模型。MiniMax是一家中国科技公司，一直致力于进行大模型相关的研究。",
        },
        {"role": "user", "content": "你好"},
    ],
)

print(completion)
print("****")
content = completion.choices[0].message.content
print(content)
print("****")
print("Trace-ID:", completion.id)
```

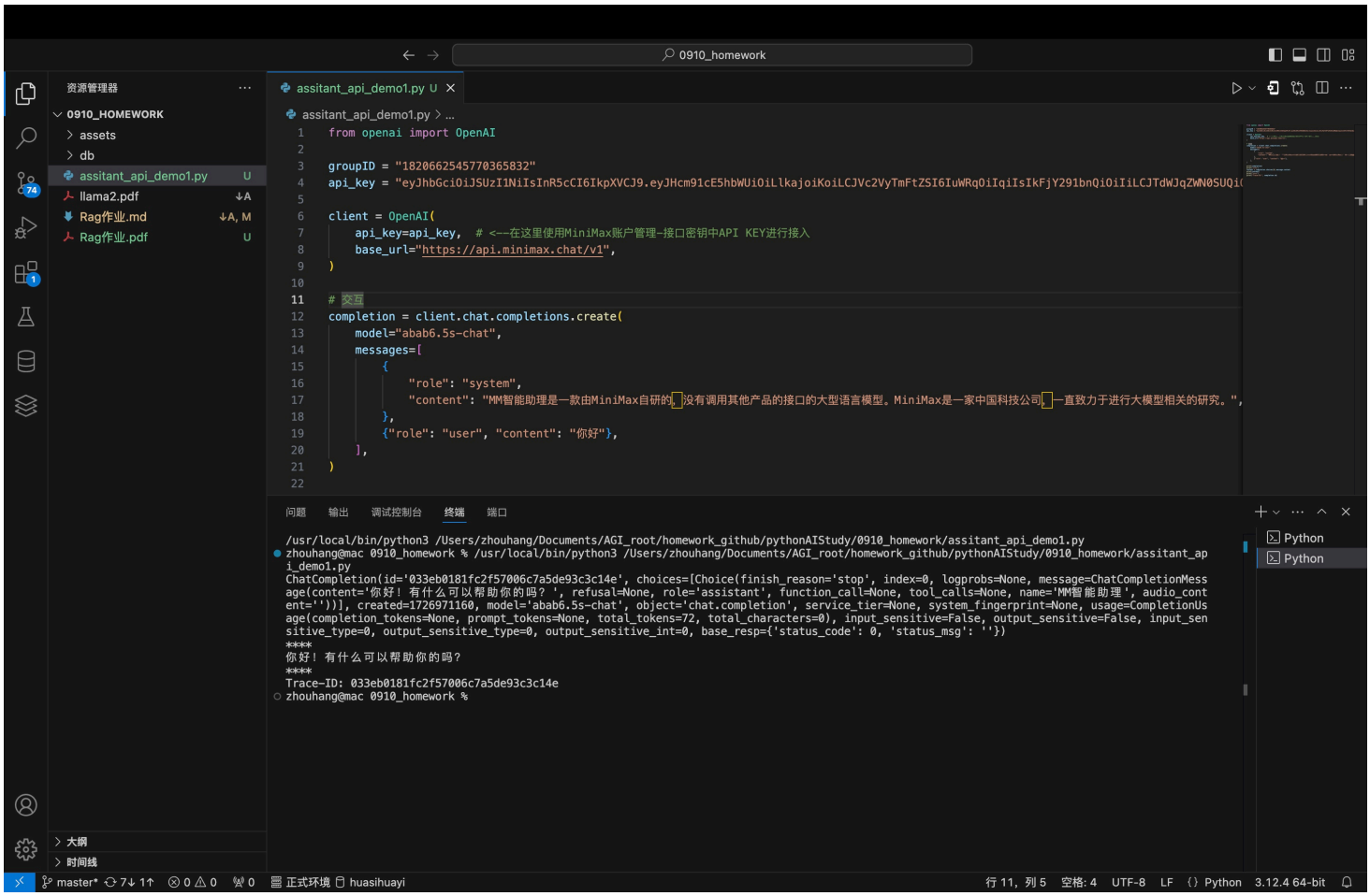
上面代码中的groupId和apiKey，都是来自minMax官网，等你用手机号注册之后就会自动给你。如下图：




接口密钥则需要你主动创建，注意创建一次之后记得保存，因为它不会显示第二次。




测试一下结果。




可见，代码运行之后，可以打印出来自Ai的回答。

由此可以证明我们的密钥和GroupId是正确的。
代码的写法和我们之前练习 OpenAi的时候一摸一样。

## 上传知识库文件

假如我们想要购买显卡，下面的表格是我们收集到的最新显卡参数和售价。

| 显卡型号 | 大致售价(人民币) | 核心数 | 显存 | 核心频率 | 显存带宽 |
|---|---|---|---|---|---|
| NVIDIA GeForce RTX 4090 | 13000 - 16000元 | 16384 CUDA核心 | 24 GB GDDR6X | 2.23 GHz（提升至 2.52 GHz） | 1008 GB/s |
| NVIDIA GeForce RTX 4080 | 8000 - 11000元 | 9728 CUDA核心 | 16 GB GDDR6X | 2.21 GHz（提升至 2.51 GHz） | 736 GB/s |
| NVIDIA GeForce RTX 4070 Ti | 5000 - 7000元 | 7680 CUDA核心 | 12 GB GDDR6X | 2.31 GHz（提升至 2.61 GHz） | 504 GB/s |
| NVIDIA GeForce RTX 4070 | 4000 - 6000元 | 5888 CUDA核心 | 12 GB GDDR6 | 2.48 GHz（提升至 2.61 GHz） | 504 GB/s |
| NVIDIA GeForce RTX 4060 Ti | 3000 - 4500元 | 4352 CUDA核心 | 8 GB GDDR6 | 2.67 GHz（提升至 2.85 GHz） | 288 GB/s |
| NVIDIA GeForce RTX 4060 | 2500 - 3500元 | 3072 CUDA核心 | 8 GB GDDR6 | 2.76 GHz（提升至 2.85 GHz） | 288 GB/s |
| AMD Radeon RX 7900 XTX | 9000 - 12000元 | 6144 流处理器 | 24 GB GDDR6 | 2.5 GHz | 960 GB/s |
| AMD Radeon RX 7900 XT | 7000 - 9000元 | 5376 流处理器 | 20 GB GDDR6 | 2.4 GHz | 800 GB/s |
| AMD Radeon RX 7800 XT | 4000 - 6000元 | 3840 流处理器 | 16 GB GDDR6 | 2.4 GHz | 512 GB/s |
| AMD Radeon RX 7700 XT | 3000 - 4000元 | 2560 流处理器 | 12 GB GDDR6 | 2.6 GHz | 384 GB/s |

我们给出一个预算，并且提出一定的指标参数要求，要求AI帮我们看看有没有合适的显卡可以购买。

那么这份文件，就是我们的私有数据库，想让Ai帮我们做 购买显卡的参考，就必须让ai先读取这份文件，在读取之前，首先要上传这个文件到Ai服务器。

```
import os
import requests  # 导入requests库，用于发送HTTP请求
import json  # 导入json库，用于处理JSON数据
import time  # 导入time库，用于处理时间相关功能

groupID = "xxx"
api_key = "xxxx"


# 用于上传时的身份校验
headers = {"Authorization": f"Bearer {api_key}", "Content-Type":
"application/json"}

# 同样检索的时候身份校验。
headers_retrieval = {
    "Authorization": f"Bearer {api_key}",
    "authority": "api.minimax.chat",
}


# create_file 函数的目的是将一个文件上传到 Minimax API
def create_file():
    # 定义文件上传的 API 的 URL 地址。
    # GroupId 是用于指定文件上传目标组的查询参数。
    url = f"https://api.minimax.chat/v1/files/upload?GroupId={groupID}"

    # 定义要发送的数据。'purpose' 字段用于说明上传文件的目的，这里是用于助手功能。
    data = {"purpose": "assistants"}

    # 打开要上传的文件，使用二进制模式 ('rb') 读取文件内容。'file' 字段用于指定上传的文件。
    files = {"file": open(r"data.md", "rb")}

    # 发送 POST 请求到 API 端点，包含 headers、data 和 files。
    # headers_retrieval 是定义请求头，data 包含上传文件的目的，files 包含实际要上传的文
件。
    response = requests.post(url, headers=headers_retrieval, data=data,
files=files)

    # 将 API 响应的 JSON 数据解析并返回。response.json() 方法将响应内容转换为 JSON 格
式。
    return response.json()


if __name__ == "__main__":
    # 上传文档
```

```
    file_response = create_file()
    print(file_response)
    print("***************")
    # 获取文件ID
    file_id = file_response.get("file", {}).get("file_id")
    print(file_id)
```

运行到结果是，

```
{'file': {'file_id': 186013044846675, 'bytes': 1586, 'created_at': 1726972807,
'filename': 'data.md', 'purpose': 'assistants'}, 'base_resp': {'status_code': 0,
'status_msg': 'success'}}
***************
186013044846675
```

看最后得到的数字，就是我们此次上传的文件唯一id，先存着，后面有用。

## 创建助手对象

和上面上传文件类似，创建助手对象，也是调用一个http接口来达成的，到目前为止，完整代码如下：

```
import os
import requests  # 导入requests库，用于发送HTTP请求
import json  # 导入json库，用于处理JSON数据
import time  # 导入time库，用于处理时间相关功能

groupID = "xxxx"
api_key = "xxxx"


# 用于上传时的身份校验
headers = {"Authorization": f"Bearer {api_key}", "Content-Type":
"application/json"}

# 同样检索的时候身份校验。
headers_retrieval = {
    "Authorization": f"Bearer {api_key}",
    "authority": "api.minimax.chat",
}


# create_file 函数的目的是将一个文件上传到 Minimax API
def create_file():
    # 定义文件上传的 API 的 URL 地址。
    # GroupId 是用于指定文件上传目标组的查询参数。
    url = f"https://api.minimax.chat/v1/files/upload?GroupId={groupID}"
```

```
    # 定义要发送的数据。'purpose' 字段用于说明上传文件的目的，这里是用于助手功能。
    data = {"purpose": "assistants"}

    # 打开要上传的文件，使用二进制模式 ('rb') 读取文件内容。'file' 字段用于指定上传的文件。
    files = {"file": open(r"data.md", "rb")}

    # 发送 POST 请求到 API 端点，包含 headers、data 和 files。
    # headers_retrieval 是定义请求头，data 包含上传文件的目的，files 包含实际要上传的文
件。
    response = requests.post(url, headers=headers_retrieval, data=data,
files=files)

    # 将 API 响应的 JSON 数据解析并返回。response.json() 方法将响应内容转换为 JSON 格
式。
    return response.json()


def create_assistant(file_id):
    # 构建 API 请求的 URL，使用 GroupId 作为查询参数
    url = f"https://api.minimax.chat/v1/assistants/create?GroupId={groupID}"

    # 准备请求的负载数据，将数据转换为 JSON 格式的字符串
    payload = json.dumps(
        {
            "model": "abab6.5s-chat",  # 指定要使用的模型版本，这里是 'abab6.5s-
chat'
            "name": "显卡购买参考助手",  # 助手的名称
            "description": "在用户给出购买显卡预算，并给出显卡性能指标要求的前提下，帮用户
筛选出价格合适，性能满足的显卡型号",  # 对助手的描述，说明其功能和用途
            "instructions": "擅长帮用户在价格和显卡性能之间作出权衡，帮用户做出最佳的购买
建议，并说明详细的建议理由",  # 助手的设定或指令，说明助手的工作原理和目标
            "file_ids": [  # 列表包含了与助手相关的文件 ID
                str(file_id)  # 将传入的文件 ID 转换为字符串形式
            ],
            "tools": [  # 列表包含助手使用的工具
                {"type": "retrieval"}  # 使用 'retrieval' 工具，可能用于从文件中检索
数据
            ],
        }
    )
    # 发送 POST 请求到 API，创建助手
    response = requests.post(url, headers=headers, data=payload)
    # 返回 API 的响应，解析为 JSON 格式
    return response.json()


if __name__ == "__main__":
    # 上传文档
    file_response = create_file()
    file_id = file_response.get("file", {}).get("file_id")

    # 创建助手
```

```
    assistant_response = create_assistant(file_id)
    print(assistant_response)
    print("****************")
    assistant_id = assistant_response.get("id", "")
    print(assistant_id)
```

返回的打印为：

```
{'id': 'asst_901bd123208d4810b3ad79b70ee79704', 'object': 'assistant',
'created_at': 1726974124, 'name': '显卡购买参考助手', 'description': '在用户给出购买
显卡预算，并给出显卡性能指标要求的前提下，帮用户筛选出价格合适，性能满足的显卡型号', 'model':
'abab6.5s-chat', 'instructions': '擅长帮用户在价格和显卡性能之间作出权衡，帮用户做出最佳
的购买建议，并说明详细的建议理由', 'tools': [{'type': 'retrieval'}], 'file_ids':
['186043267596420'], 'metadata': {}, 'rolemeta': {}, 'status': 'loading',
'base_resp': {'status_code': 0, 'status_msg': 'success'}}
****************
asst_901bd123208d4810b3ad79b70ee79704
```

## 创建线程

```
# 创建线程的函数
def create_thread():
    # 构造请求的 URL，包含 GroupId 参数
    url = f"https://api.minimax.chat/v1/threads/create?GroupId={groupID}"

    # 发送 POST 请求以创建一个新线程
    response = requests.post(url, headers=headers)

    # 返回响应的 JSON 数据
    return response.json()

if __name__ == "__main__":
    # 创建线程
    thread_response = create_thread()
    print(thread_response)
    print("***************")
    thread_id = thread_response.get("id", "")
    print(thread_id)
```

创建线程，只需要一个groupId，groupId是注册用户的唯一标志。所以这个线程是绑定在用户上的。

## 向线程中添加消息

其实也就是提出用户的问题。

我们今天设定的场景是，让ai帮我们就 购买显卡给出参考建议。所以我们这么提问，“假如我有

3000块，帮我选一款性价比尽可能高的显卡,并给出详细理由”。

代码如下：

```
# 往指定线程中添加消息的函数
def add_message_to_thread(thread_id):
    # 构造请求的 URL，包含 GroupId 参数
    url = f"https://api.minimax.chat/v1/threads/messages/add?GroupId={groupID}"

    # 构造要发送的消息数据
    payload = json.dumps(
        {
            "thread_id": thread_id,  # 指定要向哪个线程添加消息
            "role": "user",  # 消息发送者的角色，这里是 'user'
            "content": "假如我有3000块，帮我选一款性价比尽可能高的显卡,并给出详细理由",
# 消息内容
        }
    )

    # 发送 POST 请求以向指定线程添加消息
    response = requests.post(url, headers=headers, data=payload)

    # 返回响应的 JSON 数据
    return response.json()
```

## 运行助手，并循环查询运行结果，直到成功，并解析结果

```
# 运行助手
def run_thread_with_assistant(thread_id, assistant_id):
    # 这里可以选择添加延时（time.sleep(20)）以等待助手的创建和初始化完成。注释掉的代码是为了处理助手创建后可能需要的时间，
    # 用于助手的数据向量化和存储。在实际应用中，可以通过检索助手状态来判断助手是否已成功创建。

    # 定义API请求的URL。这个URL用于向指定的线程中运行指定的助手。
    url = f"https://api.minimax.chat/v1/threads/run/create?GroupId={groupID}"

    # 定义请求的负载数据。这个数据包含了要运行的线程ID和助手ID。
    payload = json.dumps(
        {
            "thread_id": thread_id,  # 线程的唯一标识符。助手将在这个线程中运行。
            "assistant_id": assistant_id,  # 助手的唯一标识符。指定了要运行的助手。
        }
    )

    # 发送POST请求到API，传递请求头和负载数据。
    response = requests.post(url, headers=headers, data=payload)

    # 返回API的响应结果，以JSON格式返回。
    return response.json()
```

```python
# 查看运行状态
def check_thread_run_status(thread_id, run_id):
    # 定义请求的URL，构造获取线程运行状态的API端点。
    # 这里URL包括了`GroupId`作为查询参数，表示请求的是特定群组中的运行状态。
    url = f"https://api.minimax.chat/v1/threads/run/retrieve?GroupId={groupID}"

    # 创建请求的负载数据，包含线程ID和运行ID。
    # 这些数据将用于API请求中，以指定要查询的线程和运行状态。
    payload = json.dumps(
        {
            "thread_id": str(thread_id),  # 线程ID，指定要查询的线程
            "run_id": str(run_id),  # 运行ID，指定要查询的具体运行
        }
    )

    # 初始化`completed`变量为`False`，表示任务尚未完成。
    # 当我们把这个变量修改成True,那么表示任务完成
    completed = False

    # 使用循环不断检查任务的运行状态，直到任务完成。
    while not completed:
        # 发送GET请求到指定的URL，附带请求头和负载数据。
        response = requests.request("GET", url, headers=headers, data=payload)

        # 检查响应状态码是否为200，表示请求成功。
        if response.status_code == 200:
            # 将API响应的内容转换为JSON格式。
            response_data = response.json()

            # 从响应数据中提取`status`字段，获取任务的当前状态。
            status = response_data.get("status", "")

            # 打印当前任务状态。
            print(f"Status: {status}")

            # 检查任务状态是否为`completed`，如果是，标记为完成。
            if status == "completed":
                completed = True
                print("Process completed, exiting loop.")
            else:
                # 如果状态不是`completed`，等待两秒后重新请求。
                time.sleep(2)
        else:
            # 如果请求失败，打印错误信息，并退出循环。
            print(f"Error: {response.status_code}")
            break

    # 返回任务是否完成的状态。
    return completed
```

```python
def get_thread_messages(thread_id):
    # 定义要请求的URL，使用线程ID来构造URL，以便获取该线程的消息列表
    url = f"https://api.minimax.chat/v1/threads/messages/list?GroupId={groupID}"

    # 创建请求的负载数据，包含线程ID
    payload = json.dumps({"thread_id": thread_id})  # 包含要查询消息的线程ID

    # 发送GET请求到指定的URL，附带请求头和负载数据
    response = requests.get(url, headers=headers, data=payload)

    # 将响应内容转换为JSON格式并返回
    return response.json()

if __name__ == "__main__":
    # 上传文档
    file_response = create_file()
    file_id = file_response.get("file", {}).get("file_id")

    print("fileId==", file_id)

    # 创建助手
    assistant_response = create_assistant(file_id)
    # print(assistant_response)
    print("*****************")
    assistant_id = assistant_response.get("id", "")
    print("assitantId=", assistant_id)

    # 创建线程
    thread_response = create_thread()
    # print(thread_response)
    print("***************")
    thread_id = thread_response.get("id", "")
    print("threadId=", thread_id)

    add_message_to_thread(thread_id)

    run_response = run_thread_with_assistant(
        thread_id=thread_id, assistant_id=assistant_id
    )
    print(run_response)
    print("-----------")
    run_id = run_response.get(
        "id", ""
    )  # 假设run_response是正确的JSON响应，并包含run_id
    print("run_id:", run_id)

    # 检查助手处理状态
    if check_thread_run_status(thread_id, run_id):
        # 获取线程中助手处理出的新信息
        thread_messages_response = get_thread_messages(thread_id)
        # print(type(thread_messages_response))
```

```
        print("============================")
        print(thread_messages_response["data"][1]["content"][0]["text"]
["value"])
```

运行的结果为：

```
在您的预算范围内（3000元人民币），我们需要寻找一款性能与价格平衡得最好的显卡。综合考虑当前市场上的显卡型号、性能参数、核心数、显存大小和显存带宽，以及性价比，以下是针对您预算的推荐：

### 推荐显卡：NVIDIA GeForce RTX 4060 Ti

**理由详细解释：**

1. **价格与预算匹配**：根据提供的信息，RTX 4060 Ti的售价范围在3000元到4500元人民币之间，正好符合您的预算区间【3†source】。

2. **出色的游戏性能**：RTX 4060 Ti拥有4352个CUDA核心，以及12 GB GDDR6显存，这样的配置在当前的主流游戏中能够提供流畅的体验，并且支持光线追踪技术，为玩家提供逼真的光影效果【3†source】。

3. **高效的显存带宽**：拥有288 GB/s的显存带宽，对于处理高分辨率纹理和大量数据的场景有很好的支持，可以保证在游戏中减少瓶颈【3†source】。

4. **良好的功耗与散热**：相较于更高端的显卡，RTX 4060 Ti在保持高性能的同时，也有相对较低的功耗和发热，这意味着它可以在保持性能的同时，对电源的要求不会过高，并且运行时的热量控制较为理想。

5. **适合多数用户**：3000元的预算能够购买到这款显卡，适合预算有限但又希望获得良好游戏体验的用户。虽然它不是市场上性能最强的显卡，但绝对能满足大多数玩家的需求。

综上所述，NVIDIA GeForce RTX 4060 Ti在您的预算范围内提供了出色的性能、良好的能效比和较高的性价比，是目前3000元左右价位的不错选择。请注意，价格会随着市场供需、促销活动和新品发布等因素有所波动，建议在购买前关注市场动态并比较不同销售渠道的价格。
```

完整的代码 以及知识库文件，我放在附件中。