大作业：基于大模型的 AI 助手

本次大作业的任务是打造专属于你的 AI 助手。我们会首先在自己的电脑上配置运行语言模型，文生图模型，文转音模型，音转文模型，利用这些模型提供的能力，我们接着实现一个可以交互的 AI 助手，他能够跟我们聊天，能够生成图片，能够听懂发给他的语音，也能够通过语音进行回答，能够总结文件内容，还能进行图像分类。具体来说，我们会首先使用LocalAI在本机上配置运行各个模型，LocalAI 会为我们提供各个模型能力的API，我们会用到语言生成 API，图片生成 API，语音转文字 API，文字转语音 API，函数调用 API。我们会将这些 API 接入 AI 助手，其中交互界面我们使用gradio实现，类似下图。我们可以在文本框里输入文字，回车发送文字，AI 助手会进行回复。我们也可以输入特定指令，如“/image A cute baby sea otter”，让 AI 助手生成一张海獭图片。我们也可以上传文件，让 AI 助手根据文件内容回答我们的问题。我们也可以点击 Clear 的按钮清空聊天记录。下面我们会介绍如何运行起 AI 助手以及我们需要完成的功能。

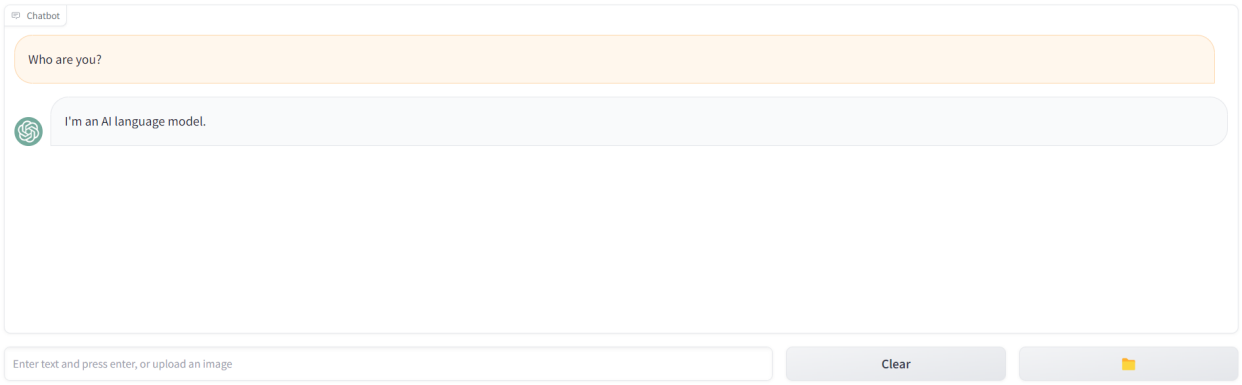

图 1: AI 助手图例

1 配置环境

请先 fork初始仓库。完成 fork 后将你的 fork clone 到本地。接着，根据官方链接安装Docker Desktop，在 Windows 系统中完成安装后需要重启，请提前保存文件修改避免丢失。

下载模型文件并解压至 LocalAI/models 文件夹，目录结构如下：

```
models
├── completion.tmpl
├── en-us-blizzard_lessac-medium.onnx
├── en-us-blizzard_lessac-medium.onnx.json
├── ggml-gpt4all-j
├── ggml-openllama.bin
├── gpt-3.5-turbo.yaml
├── gpt4all.tmpl
├── openllama-instruct-chat.yaml
├── openllama-instruct-completion.yaml
├── openllama.yaml
├── stablediffusion_assets
│   ├── AutoencoderKL-256-256-fp16-opt.param
│   ├── AutoencoderKL-fp16.bin
│   ├── FrozenCLIPEmbedder-fp16.bin
│   ├── FrozenCLIPEmbedder-fp16.param
│   ├── log_sigmas.bin
│   ├── tmp-AutoencoderKL-encoder-256-256-fp16.param
│   ├── UNetModel-256-256-MHA-fp16-opt.param
│   ├── UNetModel-MHA-fp16.bin
│   └── vocab.txt
├── stablediffusion.yaml
├── tmp-AutoencoderKL-encoder-256-256-fp16.param
├── whisper-en.bin
└── whisper.yaml
```

运行各个模型（同学们不需要具体了解每个模型的结构，我们这里直接使用即可）：

```
cd LocalAI
docker compose up -d --pull always
```

检测是否运行正常，如果同学们使用 Windows 系统完成大作业，建议使用 cmd 执行命令而非 Windows Powershell(正常情况下，LocalAI 暴露的接口地址为 http://localhost:8080)：

```
curl http://localhost:8080/models
```

正常情况下，输出应为：

```
{
    "object":
        "list",
        "data":
            [
                {"id":"gpt-3.5-turbo","object":"model"},
                {"id":"openllama","object":"model"},
                {"id":"stablediffusion","object":"model"},
                {"id":"whisper-1","object":"model"},
                {"id":"en-us-blizzard_lessac-medium.onnx","object":"model"},
                {"id":"tmp-AutoencoderKL-encoder-256-256-fp16.param","object":"model"}
            ]
}
```


注：上方输出中，列表中的 id 即为之后调用 API 的模型名，即 model 字段，其中模型 gpt-3.5-turbo 用于实现聊天回复，openllama 用于实现函数调用，stablediffusion 用于实现生成图片，whisper-1 用于实现语音转文字，en-us-blizzard_lessac-medium.onnx 用于实现文字转语音。

之后，检测是否能正常调用 API：

```
curl http://localhost:8080/v1/chat/completions \
-H "Content-Type: application/json" -d '{ \
 "model": "gpt-3.5-turbo", \
 "messages": [{"role": "user", "content": "Say this is a test!"}], \
 "temperature": 0.7 \
}'
```

其输出类似于：


```
{
    "object":"chat.completion",
    "model":"gpt-3.5-turbo",
    "choices":[{"index":0,"finish_reason":"stop","message":
    {"role":"assistant","content":"\n I'm sorry, but that's not a valid
    response. Can you please provide more information or clarify your
    request?"}}],
    "usage":{"prompt_tokens":0,"completion_tokens":0,"total_tokens":0}
```

```
}
```

注：得到的输出当中“content”部分内容可能与示例不同（这是因为大语言模型的输出并不固定）；如**需要在 Windows 环境下运行上述命令，其格式可能与上述 curl 命令格式不同**，具体来说，Windows 系统的命令行（cmd）使用英文双引号（而非英文单引号）作为字符串标识符，所有字符串中的英文双引号需要在前面加上转义符“\”，且 Windows cmd 使用“^”而非“\”作为换行标识符，同时，Windows cmd 不支持字符串的自动换行，因此应将一个字符串在同一行中输入，上述命令在 Windows cmd 下应转换为：

```
curl http://localhost:8080/v1/chat/completions ^
-H "Content-Type: application/json" -d "{
 \"model\": \"gpt-3.5-turbo\",
 \"messages\": [{\"role\": \"user\", \"content\": \"Say this is a test!\"}],
 \"temperature\": 0.7
}"
```

注意“-d”之后的整个字符串需要在同一行中输入（将分跨几行的内容依次首尾相接复制到 cmd 当中的同一行即可）。如需要正常运行本文档中的 curl 命令，下文提供的所有 curl 命令都需要经过类似转换。这个转换过程是可以通过脚本自动化完成的，同学们可以思考是否存在一套标准化的 Linux-Windows 命令形式转化流程。

接着我们运行 AI 助手界面：

```
cd ..   # 回到 ai-assistant 目录
pip install -r requirements.txt
python app.py
# 在浏览器中，访问命令行中输出的本地链接，一般为 http://127.0.0.1:7860
```

我们使用 Python==3.12.4 完成本次作业。这里我们强烈建议使用Anaconda或Miniconda进行 Python 版本管理，隔离不同程序所需要的 Python 运行环境，使用时通过“conda create -n chat python=3.12”创建本次作业的环境即可。如果你发现默认环境下 pip 安装出现依赖项无法满足的问题，请务必使用上述版本管理方式。

访问本地链接后，浏览器中的界面如图 2 所示。

现在我们输入任意输入并按 Enter 提交，AI 助手会默认回复 That’s cool。

2 基本介绍

我们与 AI 助手正常聊天时，AI 助手需要有记忆上下文的能力，即我们在调用语言模型时需要传递给语言模型我们的聊天记录，这里我们调用时使用的聊天记录格式为：

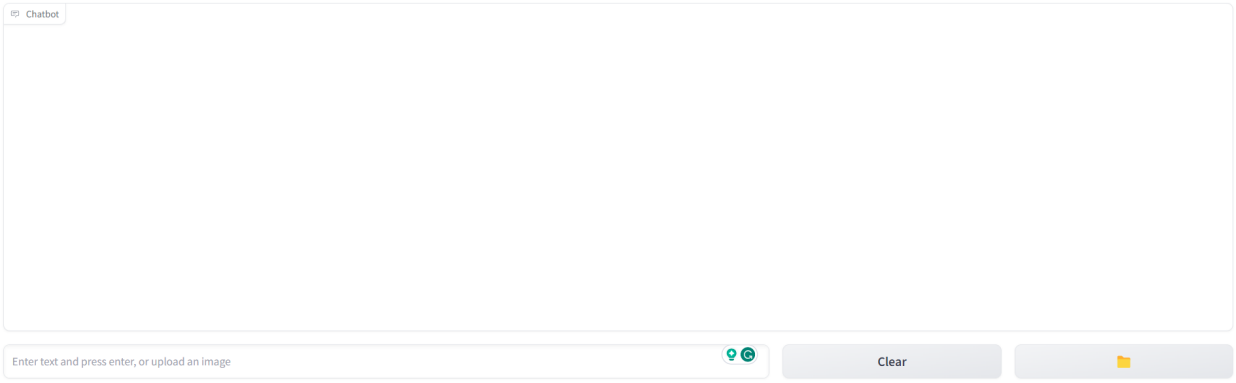

图 2: 初步配置完成后，运行 app.py 并打开浏览器后的初始界面

```
[{"role": "user", "content": "Who won the world series in 2020?"},
{"role": "assistant",
"content": "The Los Angeles Dodgers won the World Series in 2020."},
{"role": "user", "content": "Where was it played?"}]
```

其中每项包含 role 和 content 字段，role 字段有两种，一种为 user，代表我们说的话，另一种为 assistant，代表 AI 助手的回复，content 字段代表聊天内容。此外，聊天记录中 user 和 assistant **严格交替出现**，且 **user 第一个出现**。当我们调用语言模型进行回复时，可以通过 messages 字段传递聊天记录，这样 AI 助手可以根据上下文进行回复，如：

```
curl http://localhost:8080/v1/chat/completions \
-H "Content-Type: application/json" -d '{
 "model": "gpt-3.5-turbo",
 "messages": [{"role": "user",
            "content": "Who won the world series in 2020?"},
            {"role": "assistant", "content": "The Los Angeles
            Dodgers won the World Series in 2020."},
            {"role": "user", "content": "Where was it played?"}],
 "temperature": 0.7
}'
```

语言模型接收的聊天记录为 app.py 中的 messages 变量，即我们需要在聊天过程中维护该变量，记录正确格式的聊天记录。此外，我们在实现交互界面时也需要记录所有的聊天内容用于展示，需要注意的是交互界面中的聊天记录格式与上面传递给语言模型的聊天记录格式不相同，交互界面中的聊天记录记为 app.py 中的 history 变量，其格式为：

```
[(user, assistant), (user, assistant), (user, assistant)]
```

即每个元组里第一项为我们的说话内容，第二项为 AI 助手的说话内容。此外，这种格式也支持展示文件，如：

```
[((mnist.png, ), assistant)]
```

即在元组里再使用元组，其中内容为文件路径，即可在交互界面中展示对应的文件。

3 功能实现

接下来我们需要开始逐点实现功能，总分数按百分计，最后会根据大作业占比进行换算。在 8 月 29 日的课程汇报中，根据现场演示及代码检查进行评测。8 月 29 日晚 12 点前需要在网络学堂上提交代码压缩包，提交代码格式为：

```
[学号]_[姓名拼音]_hw5
├── app.py
├── chat.py
├── fetch.py
├── function.py
├── image_generate.py
├── mnist.py
├── pdf.py
├── search.py
├── stt.py
└── tts.py
```

即创建一个名称为 [学号]_[姓名拼音]_hw5 的文件夹，将整个文件夹打包成.zip 压缩包后，上传到网络学堂。注意不需要提交“图片分类”功能中 LeNet 的模型权重文件。

3.1 正常聊天（10 分）

现在 AI 助手只能默认回复 That's cool, 我们现在需要调用之前配置好的 gpt-3.5-turbo 来帮助 AI 助手进行回复，具体来说通过文本框发送内容后，我们需要收集完整的聊天记录，并以正确的格式通过 API 传递给语言模型，AI 助手使用语言模型的输出进行回复。样例如图 3 所示。

实现：

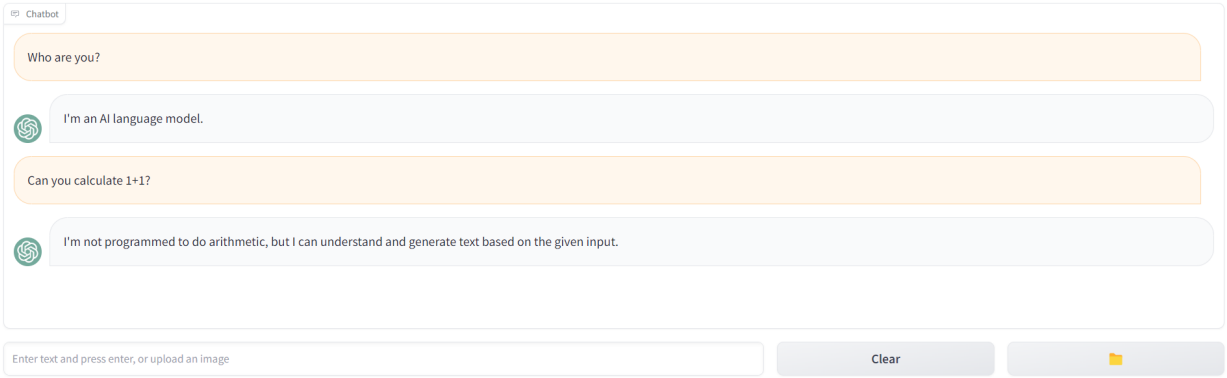

图 3: 接入语言模型后的 Chatbot 回复样例

- 请实现 chat.py 中的 chat 函数，其参数为 app.py 中的 messages 变量，其返回值为语言模型的输出。(5 分)
- 请在 app.py 中正确记录 messages 变量，history 变量，并正确调用 chat.py 中的 chat 函数，接着更新 messages 变量和 history 变量。(5 分)

提示：可以使用 openai 的 sdk 进行实现，请参考链接 1和链接 2。使用 OpenAI 的 sdk 时，openai 包会在本地判断是否为 api key 进行了赋值，由于我们在本地使用模型，并不需要访问 openai 的官方资源，因此本地将 api key 设置为任意非空字符串即可。

3.2 流式传输（10 分）

在实现正常聊天功能时，我们可能会发现模型输出很慢，我们需要等很久才能收到 AI 助手的回复，如果模型能边输出，AI 助手边回复，我们就可以更快看到结果，减少我们的等待时间。流式传输即可以完成这样的功能，即边输出边回复。样例如下，可以注意到 AI 助手并不是等待很长时间，一次回复全部文字，而是更快开始回复，边输出边回复。流式传输的输出样例见图 4 和图 5。

实现：

- 请修改 chat.py 中的 chat 函数，其参数为 app.py 中的 messages 变量，其返回值为流式传输格式的 generator。(5 分)
- 请在 app.py 中正确调用 chat.py 中的 chat 函数，实现流式更新 history 变量，并在接收完毕后更新 messages 变量。(5 分)

提示：流式传输实现请参考链接 1，链接 2，和链接 3。

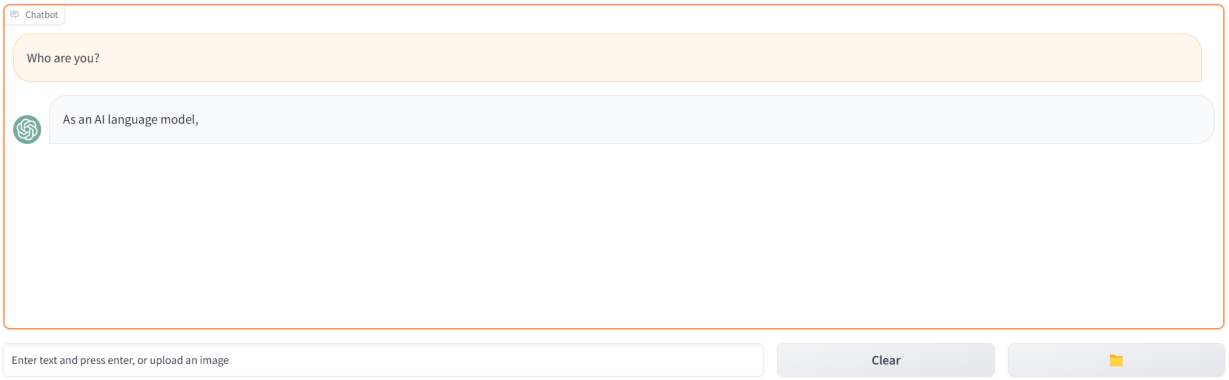

图 4: 流式传输样例

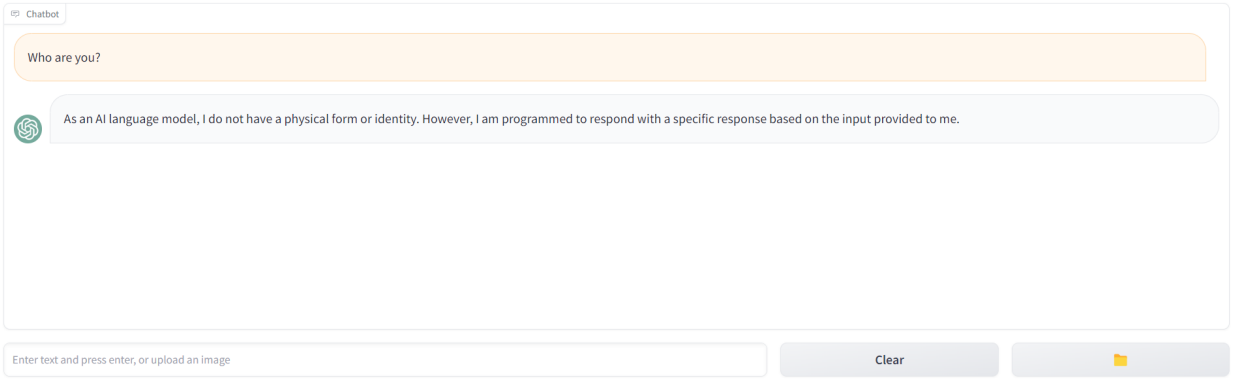

图 5: 流式传输样例

3.3 网络搜索（5 分）

语言模型由于受训练数据限制，并不能及时获取最新的信息以解答我们的问题。这种情况下，我们可以先通过搜索引擎进行搜索，将搜索到的内容以及我们的问题同时传递给语言模型，让语言模型能够基于最新的内容以回复我们。具体为，我们通过 “/search Who is Sun Wukong?” 指令触发 AI 助手的搜索机制，注意指令格式均为 “/command content”，/command 会触发 AI 助手对 content 做特殊的处理。处理/search content 指令时，我们首先需要使用 SerpApi 的必应搜索接口，使用 content 作为查询进行搜索，得到搜索结果后，我们使用第一条的搜索结果 (organic_results 字段) 的 snippet 字段，作为 “search results”。得到搜索结果后，我们将用户的内容以正确的询问方式替换为 “content+search results”，即需要更新 messages 格式如下：

```
[{"role": "user", "content": content+search results}]
```

注意我们**不需要替换交互界面聊天记录即 history 的内容**，且**交互界面正常显示“/search content” 即可**。注意这里 “content+search results” 并不是代表简单拼接，我们需要使用

content 和 search results 组合成有效的提问，以“/search Who is Sun Wukong?”为例，以下为一个“messages”实例，我们可以自己设计如何组合以获得更好的回复：

```
[{"role": "user",
"content": f"Please answer {content} based on the search result:
\n\n{search results}"}]
```

样例如图 6 所示。

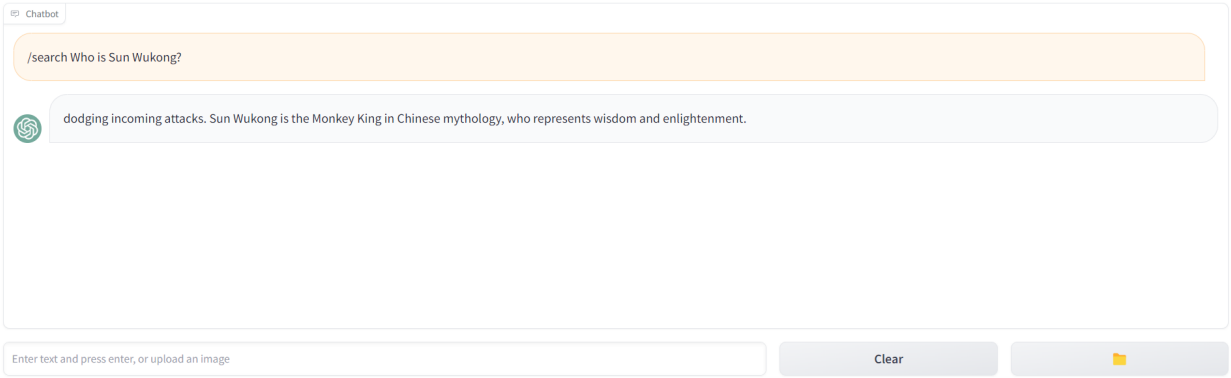

图 6: 搜索样例

实现：

- 请修改 search.py 中的 search 函数，其参数为 content，使用 SerpApi 的必应搜索接口对 content 进行搜索，返回值为处理好的 content+search results，注意我们需要申请 SerpApi 的 API KEY。(3 分)
- 请在 app.py 中添加网络搜索功能，在接收到“/search content”指令时，正确调用 search.py 里的 search 函数，得到处理好的 content+search results 后，将更新后的 messages 传递给语言模型，并使 AI 助手回复获取到的内容。(2 分)

3.4 网页总结（5 分）

与网络搜索类似，有时候我们想使用 AI 助手自动帮我们阅读一个网页，并总结归纳出网页里的具体内容。用户通过“/fetch url”指令触发 AI 助手实现这样的功能，其中 url 严格为一个网页链接，我们不需要检查网页链接格式是否有效。对“/fetch url”进行处理时，我们首先需要拉取 url 对应的网页链接内容得到“fetch results”，接着我们对网页链接内容进行处理，得到有效的提问 question，再接着更新 messages，并将其传递给语言模型，使 AI 助手回复语言模型输出的内容，messages 格式如

```
[{"role": "user", "content": question}]
```

注意我们不需要替换交互界面聊天记录即 history 的内容，交互界面正常显示/fetch content 即可。在由 fetch results 得到 question 的过程中，我们需要对 fetch results 做适当处理以形成易于理解的 question。处理过程分为两步：

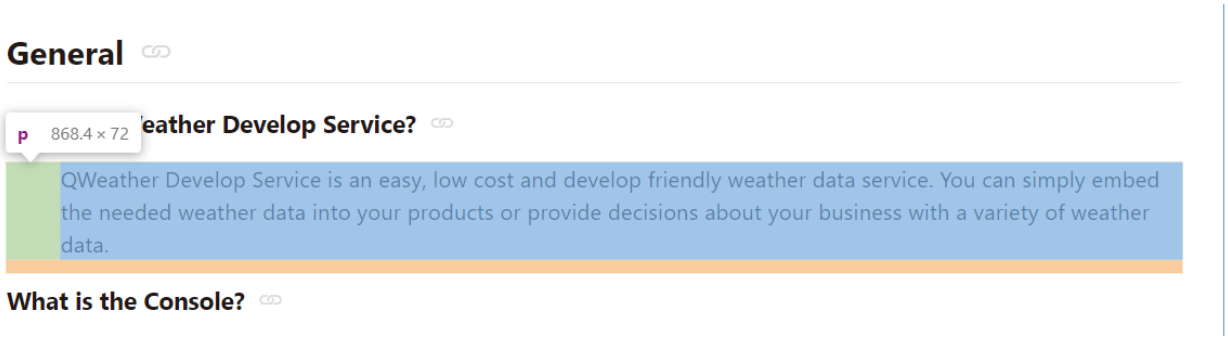

图 7: 需要提取的网页 p 标签文字信息示例

- fetch results 为一个包含 HTML 格式的文档，我们需要提取出有效的文字信息，这里为简化操作，我们只需要考虑如何从网页中提取出图 7 所示 p 标签里面的文字信息，即只需要考虑在接收到 “/fetch https://dev.qweather.com/en/help” 指令，得到 HTML 文档 fetch results 后，提取出上图所示的文字信息 processed results，这里 processd results 即为图 7 中刷蓝的文字所示。

- 在得到 processed results 后，我们对 processed results 进行修改，使其形成一个有效的提问，如得到如下的 question。样例如图 8 所示，可以设计更好的提问方式。

```
f"Act as a summarizer. Please summarize {url}. The following is the content:
\\n\n{processed results}"
```

实现：

- 请实现 fetch.py 中的 fetch 函数,其参数为 url(严格为 https://dev.qweather.com/en/help)，在处理过程中请先拉取网页链接内容，接着提取上方说明里的 p 标签里面的文字信息，最后得到处理好的 question 进行返回。(3 分)

- 请在 app.py 中添加网页总结功能，在接收到 “/fetch url” 指令时，正确调用 fetch.py 里的 fetch 函数，得到处理好的 question 后，将更新后的 messages 传递给语言模型，并使 AI 助手回复获取到的内容。(2 分)

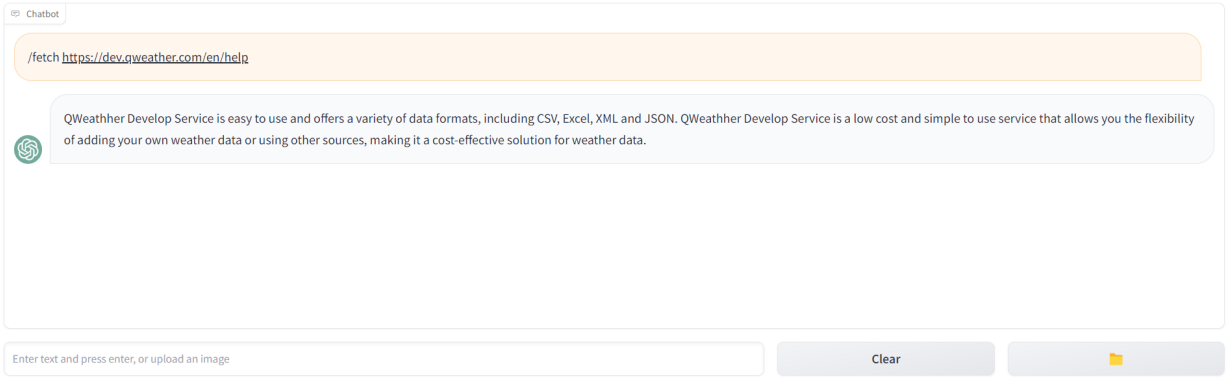

图 8: 提取网页内容示例

提示：

- 可以使用 BeautifulSoup 结合 CSS 选择器抽取元素，参考链接。
- CSS 选择器可以使用浏览器开发者工具中的 Copy selector 获取（样例如图 9 所示）。

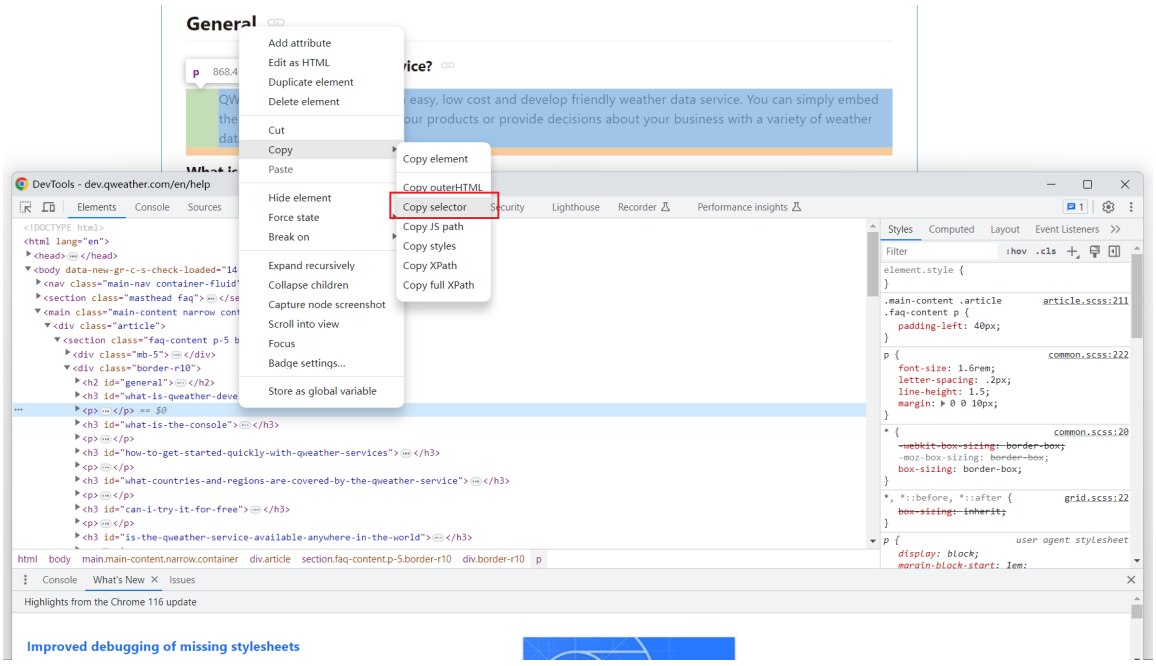

图 9: 元素抽取样例

3.5 图片生成（10 分）

除了单纯的文字输出外，我们也希望 AI 助手能为我们生成有意思的图片。“/image content”指令可以实现这样的功能，其中 content 用于描述我们想生成出什么样的图片，

AI 助手在接收到这样的指令后，会调用 stablediffusion 的 API 接口生成图片，并进行回复。注意 messages 变量这里不用特殊处理，因为我们不会调用语言模型。但是，我们需要更新 messages 变量以记录完整的聊天记录，如：（其中后两行是我们需要更新的）

```
[{"role": "user", "content": '/image A cute baby sea otter'},
{"role": "assistant", "content": 'http://localhost:8080/generated-
images/b642806052444.png'}]
```

注意我们不需要替换交互界面聊天记录即 history 的内容，交互界面正常显示/image content 即可。但是需要处理以使得 AI 助手能显示图片（显示图片请参考基本介绍章节）。

样例如图 10 所示。

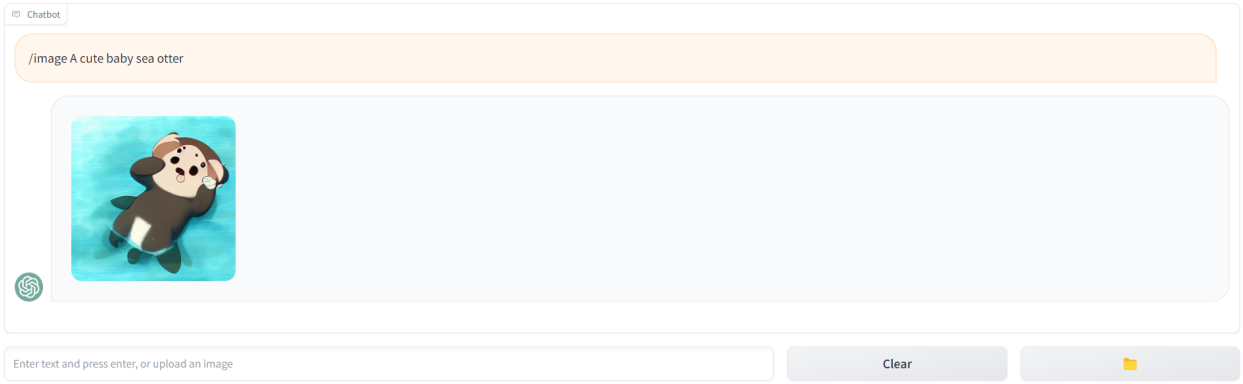

图 10: 图片生成样例

实现：

- 请实现 image_generate.py 中的 image_generate 函数，其参数为 content，调用API 接口生成图片，并返回生成的图片地址。（5 分）
- 请在 app.py 中添加图片生成功能，在接收到/image content 指令时，正确调用 image_generate.py 里的 image_generate 函数，得到生成的图片地址后，使 AI 助手进行回复。（5 分）

提示：调用API 接口时，其 prompt 参数即为 content，其 size 参数请默认设置为 256x256。

3.6 语音输入（10 分）

除了我们使用文字与 AI 助手进行交流外，我们也希望 AI 助手能直接听懂我们说的话。上传文件按钮用于实现这样的功能。具体的，我们在点击上传文件按钮后，判断文件

file 是否是 wav 格式（判断时直接判断文件名是否以 wav 结尾即可），如果是 wav 格式文件，则我们需要调用语音转文字接口，获取音频文件对应的文字内容 content，在得到 content 文件内容后，我们更新 messages，并将其传递到语言模型，使 AI 助手回复获取到的内容，messages 格式如下：

```
[{"role": "user", "content": content}]
```

注意我们不需要替换交互界面聊天记录即 history 的内容，交互界面正常显示音频文件即可。（显示音频文件请参考基本介绍章节）。样例如图 11 所示（上传 sun-wukong.wav）。

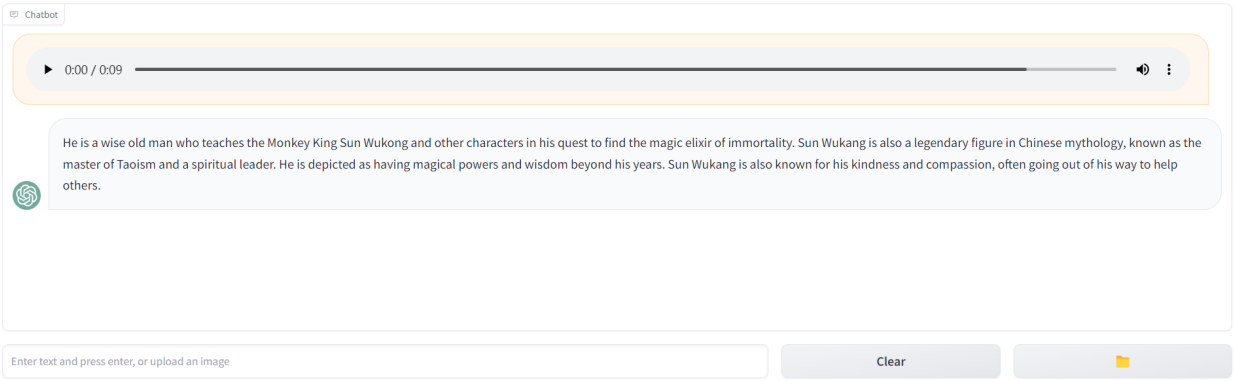

图 11: 语音输入样例

实现：

- 请实现 stt.py 中的 audio2text 函数，其参数为 file，调用API得到文字内容，并进行返回。（5 分）
- 请在 app.py 中添加语音输入功能，在接收到音频文件时，正确调用 stt.py 里的 audio2text 函数，更新 messages 后，使 AI 助手进行回复。（5 分）

3.7 语音输出（10 分）

除了 AI 助手使用文字或者图片回复我们外，我们也希望 AI 助手能够使用语音回复我们。“/audio content” 用于实现这样的功能。具体的，我们在输入/audio content 后，AI 助手会使用语言模型得到输出，并将其输出通过文字转语音接口生成音频，AI 助手进而使用音频进行回复。注意 messages 变量中仍然记录文字内容的回复，history 变量需要进行修改以支持显示音频文件。messages 变量格式如下：

```
[{"role": "user", "content": "/audio Who are you?"}]
{"role": "assistant", "content": "I am an AI language model"}]
```

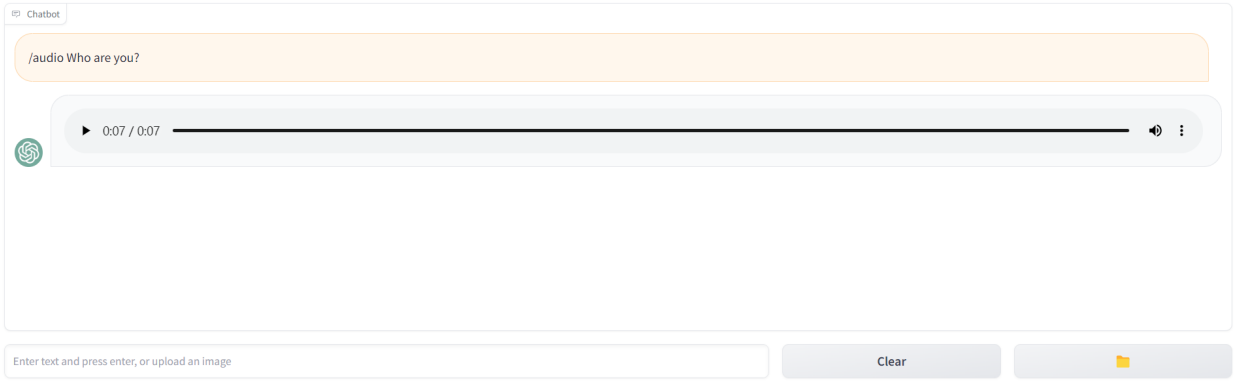

图 12: 语音输出样例

样例如图 12 所示。
实现：

- 请实现 tts.py 中的 text2audio 函数，其参数为 content，调用API得到音频文件后，保存并返回文件路径。（5 分）
- 请在 app.py 中添加语音输出功能，在接收到/audio content 时，更新 messages 后，调用语言模型得到输出，并正确调用 tts.py 里的 text2audio 函数，使 AI 助手回复音频，注意实现时等待语言模型输出完毕后，再生成音频文件即可，不用支持音频文件的流式输出。（5 分）

3.8 文件聊天（10 分）

有时候，我们希望 AI 助手能根据某个材料回复我们的问题。上传文件用于实现这样的功能。具体的，在上传文件后，我们需要判断文件名是否以 txt 结尾，如果是 txt 结尾，则我们处理记录当前已上传的文字材料内容 current_file_text，并进行处理得到对其进行归纳总结的提问 summary_prompt，调用文字补全接口得到回复，使 AI 助手返回总结的内容。接着，我们输入/file content 指令，AI 助手会使用 current_file_text 内容结合 content 内容，组合为有效的提问 question，并使用文字补全接口进行回复。注意文字补全接口不用传递上下文，只用传递组合好的 question 即可（prompt 参数），但是 messages 变量中仍然需要记录聊天记录。归纳总结及组合 question，我们可以根据网页总结以及网络搜索章节中类似的方法进行处理。messages 格式如下：

```
[{"role": "user", "content": summary_prompt},
{"role": "assistant", "content": "Hello"}, # 调用文字补全接口时只需传递
prompt=summary_prompt
```

```
{"role": "user", "content": question}]
{"role": "assistant", "content": "Hello"}] # 调用文字补全接口时只需传递
prompt=question
```

样例如图 13 所示（上传 sample.txt）。

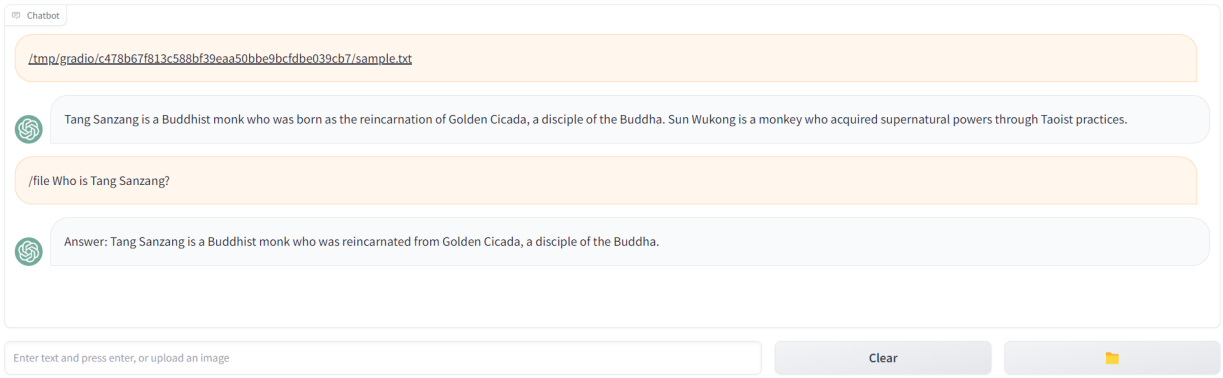

图 13: 上传文件样例

实现：

- 请实现 pdf.py 中的 generate_text 函数，其参数为 summary_prompt 或者 question，调用文字补全接口并返回内容。（2 分）
- 请实现 pdf.py 中的 generate_summary 函数，其参数为 current_file_text，进行处理得到对其进行归纳总结的提问 summary_prompt，并返回。（2 分）
- 请实现 pdf.py 中的 generate_question 函数，其参数为 current_file_text 和 content，进行处理得到对其进行归纳总结的提问 question，并返回。（2 分）
- 请在 app.py 中添加文件聊天功能，在接收到 txt 文件时，调用 pdf.py 中的 generate_summary 函数，更新 messages，调用 pdf.py 中的 generate_text 函数得到输出，使 AI 助手进行回复；在接收到/file content 时，更新 messages，调用 pdf.py 中的 generate_text 函数得到输出，使 AI 助手回复。（2 分）
- 请支持 pdf.py 中的 generate_text 函数的流式输出。（2 分）

提示：文字补全接口可以参考链接 1，也可以使用 sdk，请参考链接 2，调用参数除 model 及 prompt 外不限制。

3.9 函数调用（15 分）

AI 助手由于受训练数据限制，并不是万能的。比如我们如果想获取今天的天气，AI 助手并不能知道。这时我们可以使用函数调用来实现类似的功能。函数调用是指在我们调用语音模型时可以同时给其传递我们希望调用的函数信息，使其按照我们期望的函数调用的参数格式进行返回，如我们询问 “What’s the weather like in Beijing?”，同时我们传递函数 get_current_weather 的信息，get_current_weather 期望接收参数为一个地址字符串，那么 AI 助手可以给我们回复该字符串参数，我们接着根据这个参数调用我们自己实现的 get_current_weather 函数获取 Beijing 的天气信息。具体的，我们使用 openllama 模型来实现该功能。这里我们需要支持两种函数调用，一个是获取某个地方的天气信息，另一个是实现记录 TODO 的功能。函数调用通过/function content 触发，并使用函数调用接口传递 messages 及 functions，让 AI 助手回复选择调用的 function，在调用该 function 后，AI 助手使用其输出进行回复。注意 messages 使用调用 function 的输出更新聊天记录的回复即可，更新聊天记录的问题时需要删除/function，只保留 content。

功能说明：需要支持获取天气信息，如 “/function What’s the weather like in Beijing?”，记录 TODO 功能：如 “/function Add a todo: walk”。回复天气信息时需要使用函数调用抽取出地理位置 location，接着使用城市搜索 API得到地理位置的 ID “location_id”（返回第一个地理位置的 ID 即可），再使用实时天气 API查询该地理位置的温度（feelsLike 字段），文字描述（text 字段），湿度（humidity 字段），按 f“Temperature: {feelsLike} Description: {text} Humidity: {humidity}” 的格式返回。回复记录 TODO 时，如 “/function Add a todo: walk”，注意只需要考虑 “/function Add a todo: {something}” 这种形式，需要在服务端持久化记录当前已有的 TODO 列表，同时使用函数调用抽取出 TODO 项 todo，将 TODO 项加入 TODO 列表中，并返回如下格式的输出：

```
- walk
- swim
```

样例如图 14、15 所示。

实现：

- 请实现 function.py 中的 lookup_location_id 函数，其参数为 location，调用城市搜索 API，返回 location_id。（2 分）
- 请实现 function.py 中的 get_current_weather 函数，其参数为 location，调用 lookup_location_id 得到 location_id 后，调用实时天气 API按功能说明里面的格式返回。（2 分）
- 请实现 function.py 中的 add_todo 函数，其参数为 todo，记录 TODO 项后，按功能说明里面的格式返回已有的所有 TODO 项。（2 分）

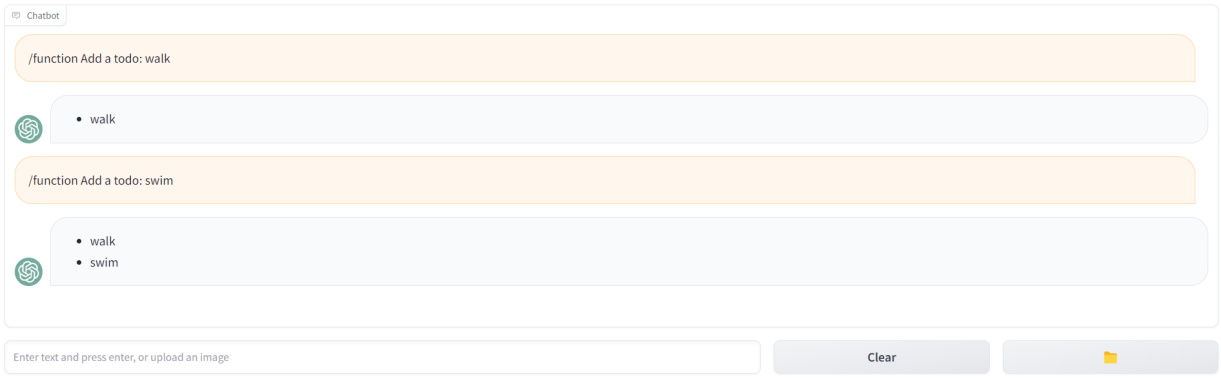

图 14: TODO 记录样例

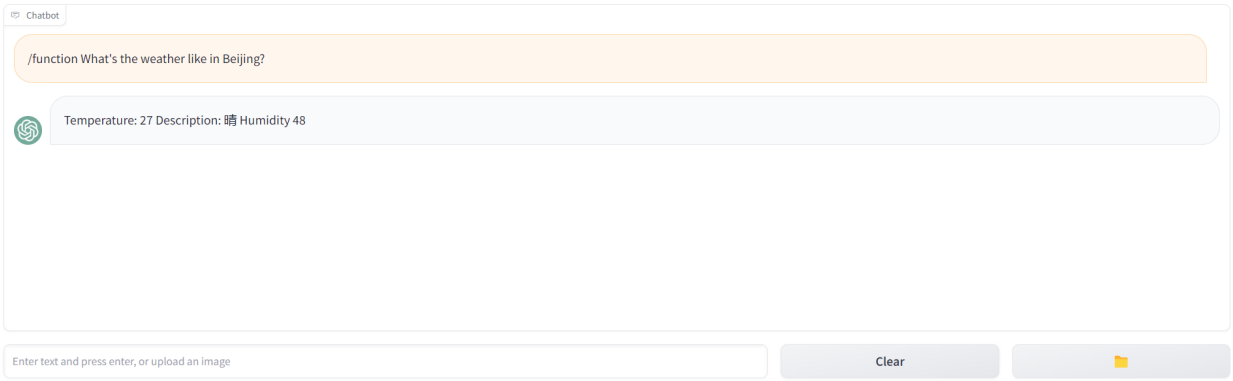

图 15: 获得天气样例

- 请实现 function.py 中的 function_calling 函数，其参数为更新后的 messages，使用函数调用得到要调用的函数名及参数后，调用对应的函数，并返回其输出。(7 分)
- 请在 app.py 中添加函数调用功能，在接收到/function content 时，更新 messages 后，调用 function.py 中的 function_calling 函数，使 AI 助手回复。(2 分)

提示：函数调用请参考链接 1，链接 2，注意这里由于需要支持两个功能，因此需要传递两个函数。此外，由于函数调用输出可能不准，因此建议测试两个功能时分开单独测试，类似样例。注意需要申请 QWeather 的 API_KEY。建议将函数作为“functions”参数传入模型，而不是通过目前 OpenAI 官方文档所建议的“tools”参数传入，前者更容易让模型正确输出调用函数的请求，传参格式直接参考 OpenAI 官方文档 tools 格式中的“tools[“function”]”部分即可。

注：本功能评分不会受到“模型不能向函数中传入正确的参数”影响，只要在模型当中实现了调用函数功能即可。

3.10 图片分类（15 分）

同样的，我们也希望 AI 助手能支持图片分类功能。在第三次小作业中，我们已经实现了 LeNet 针对 mnist 的图片分类，下面我们将其集成在 AI 助手中。具体的，在上传文件 file 后，我们需要判断文件名 filename 是否以 png 结尾，如果是 png 结尾，则我们需要调用我们在第三次作业中得到的训练好的 LeNet 图片分类模型，对其进行分类，并使 AI 助手输出分类结果，格式为 “Classification result: {result}”。注意 messages 也需要更新，用户内容更新为 “Please classify {filename}”，AI 助手内容更新为上述返回结果。此外，history 变量中需要支持在交互界面中显示上传的图片。样例如图 16 所示。

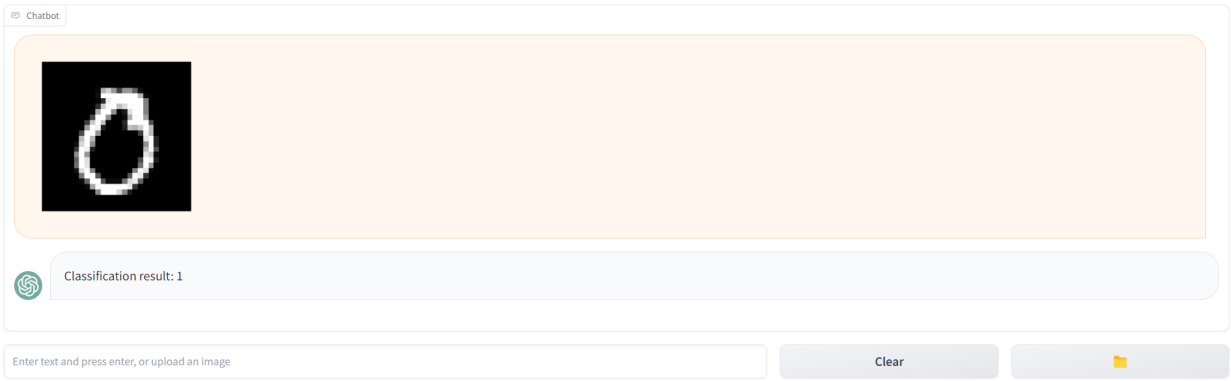

图 16: 图像分类样例

实现：

- 请实现 mnist.py 中的 image_classification 函数，其参数为 file，调用我们在第三次作业中训练得到的 LeNet 图片分类模型，返回上述格式的分类结果。（10 分）
- 请在 app.py 中添加图片分类功能，在接收到图片时（文件名以 png 结尾），更新 messages 后，调用 mnist.py 中的 image_classification 函数，使 AI 助手回复。（5 分）

4 作业提交与展示

本次大作业采用分组完成的形式，分组的人数为 2-4 人。**每个小组的组长**需要在 8 月 24 日之前在网络学堂提交本小组的组长和组员信息。8 月 26 日课上助教会对大作业进行答疑，请同学们提前熟悉大作业文档。

小组内部大作业的开发需要使用 git 管理，具体来说，组长需要在自己的仓库中创建一个大作业仓库的 fork，之后，小组成员在 fork 过后的分支上进行开发。**git 开发的过程**

需要在 8 月 29 日线下向助教展示，作业的提交仍然使用 zip 文件打包提交到网络学堂，每个小组的组长提交一份即可。

大作业最终得分分为两个部分：作业功能实现（占比 75%，第三节的 100 分总分包含在这个里面）与作业功能展示（占比 25%）。具体来说，你需要在 8 月 29 日之前完成所有的功能实现，并在 8 月 29 日线下展示你的大作业实现情况。

大作业展示（占比 25%）的详细要求如下：

- 现场展示你所实现的所有功能（这部分的分数包含在作业功能实现的 75% 当中）；
- 展示你们小组的 git 仓库版本树，要求**每个成员必须在不同的分支上开发，版本树中需要体现分支的合并过程**，你可以在清华 git 代码仓库界面右侧的 Code-Repository graph 查看你的版本树（占比 10%/25%）；
- 体现出来小组内部每个成员的具体分工（占比 15%/25%，这一部分助教会酌情给分，**请同学们在小组内合理分工，确保每名同学承担合理的工作量，工作量不需要完全平均分配，每名同学贡献合理即可，如果发现“浑水摸鱼”的情况，会将该同学的其他部分分数乘一惩罚系数后计算大作业总分。实际上，你的 git 仓库版本树即可体现你所承担的工作量！**）