

第一次作业 · 学生阅读手册

从骨架到 多轮对话

循序实现版 | Python + Flask + Requests

01

运行骨架

先看见网页与演示回复

02

逐步完善

接入模型、历史和会话

03

自己验证

检查行为并整理说明

从能运行的起点出发，理解每一处改动。本册提供起始骨架和局部示例，完整程序由你逐步完成。

阅读导航

先运行骨架，再按章节修改同一个 app.py。每次改完先验证，再进入下一节。目录和正文中的彩色链接均可点击。

准备 · 运行与配置	3
1. 读懂起始骨架	4
2. 接入真实模型	6
3. 为请求加入上下文	8
4. 显示整段历史	9
5. 新建并切换会话	10
6. 自查与提交	12
附录 A · 起始骨架	14
附录 B · 排错与资料	16
附录 C · 正式作业要求	17

阅读约定：代码框会区分 Python、HTML、终端命令和数据示例。HTML 需要放入 Python 字符串；局部片段按正文说明改入对应函数。附录 A 是起始骨架，尚未接入真实模型或保存历史。

建议同时打开配套 app.py。PDF 用于阅读，复制与运行代码优先使用资料包中的文件。

准备 · 运行与配置

程序在课程服务器运行，网页通过 SSH 转发在自己的浏览器打开。

准备个人目录

在已连接服务器的终端中执行：

终端命令 / BASH

```
mkdir -p ~/homework1  
cd ~/homework1
```

把资料包的 `app.py` 复制到这里。使用 VS Code 时，确认编辑的是服务器文件；在本机保存同名文件不会自动更新服务器。已有程序先留副本，再开始修改。

两个终端与浏览器

位置	用途
终端 A：服务器 Bash	设置配置、运行 Python、查看报错
终端 B：本机终端	保持 SSH 端口转发
本机浏览器	显示页面、发送问题

在终端 A 检查环境，再启动程序。若课程服务器提供以下 `pubpy` 路径，执行：

终端命令 / BASH

```
/opt/miniconda/envs/pubpy/bin/python -c \  
"import flask, requests; print('环境就绪')"  
/opt/miniconda/envs/pubpy/bin/python app.py
```

实际路径不同时按课程说明替换；导入失败先检查环境，不要自行修改公共环境。骨架使用 Flask，后续还会使用 Requests。

在终端 B 执行以下命令。如果你平时用 `ssh test_stu` 登录服务器，就把命令最后的 你的SSH别名 替换成 `test_stu`。请使用自己账号对应的连接名称：

终端命令 / BASH

```
ssh -N -L 127.0.0.1:18500:127.0.0.1:8500 你的SSH别名
```

浏览器打开 **`http://127.0.0.1:18500/`**。本机 18500 通过 SSH 转到服务器 8500。起始页面只能演示回复，此时不需要 API Key。

两个终端保持运行是正常现象。**修改 Python 后，在终端 A 按 Ctrl+C 停止旧程序，再重新启动。**仅刷新网页不会重载代码；端口不变时，终端 B 可以保持运行。

服务器 8500 被占用时，同时改 app.py 的 port 和转发命令最后的远端端口；本机 18500 被占用时，改前面的本机端口及浏览器地址。结束后，在两个终端分别按 Ctrl+C。

模型配置

接入模型前，在终端 A 停止程序，然后在**同一个服务器 Bash 终端**执行：

终端命令 / BASH

```
export LLM_API_URL="https://api.deepseek.com/chat/completions"
export LLM_MODEL="你实际使用的模型名称"
read -r -s -p '请输入 API Key: ' LLM_API_KEY
export LLM_API_KEY
printf '\n'
```

示例是 DeepSeek Chat Completions 的完整请求地址。使用课程网关时，改用课程提供的对应地址、模型和 Key，不要只填官网或基础地址。

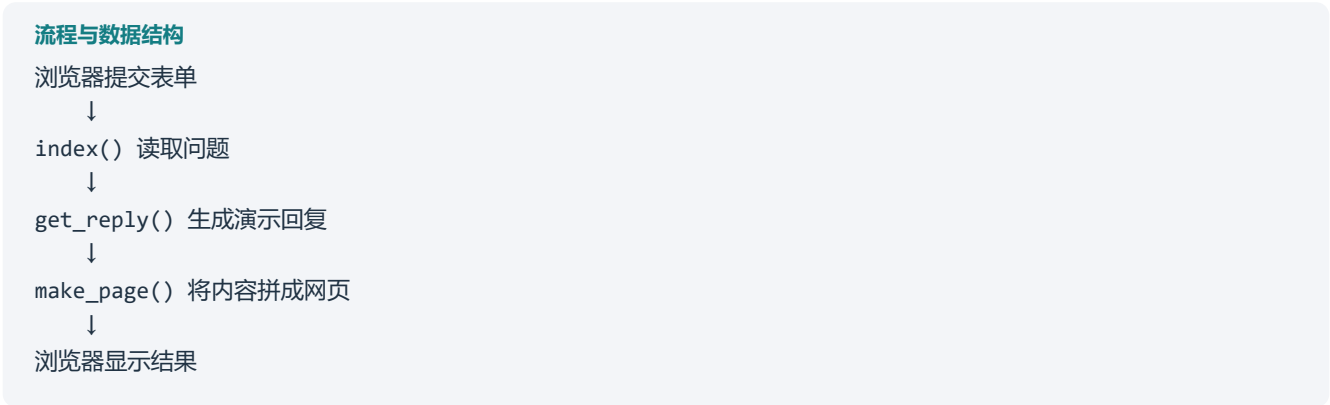
Key 在提示后输入，不显示字符。这些是 Bash 命令，不能直接在 Windows PowerShell 中执行，也不是 Python 代码。

配置好后，从同一终端启动 app.py；换终端需重新设置，修改配置需重启。只有按教程加入读取代码后，程序才会使用这些变量。不要将真实 Key 写入代码、README、截图或提交文件。

1. 读懂起始骨架

在网页输入“你好”，会看到“这是演示回复，我收到了：你好”。目前没有模型参与，我们先看清网页与 Python 如何配合。

一次发送经过了哪里



文件中只有三个自己定义的函数：

函数	作用
get_reply(question)	接收问题字符串，返回回复字符串

函数	作用
<code>make_page(question, answer, error_text)</code>	将问题、回复和提示拼成 HTML
<code>index()</code>	接收请求，组织前两个函数的调用

接入模型主要改 `get_reply()`，修改显示主要改 `make_page()`，控制调用和保存则在 `index()` 中完成。

顶部的 `Flask(__name__)` 建立应用。末尾的 `if __name__ == "__main__":` 表示直接运行文件时才执行启动语句；`app.run()` 启动服务，等待浏览器请求。无需自己写 `while` 循环等待输入。

表单与 Python 怎样对应

在 `make_page()` 中找到这段 HTML：

网页标记 / HTML

```
<form action="/" method="post">
```

`form` 是表单，`action="/"` 指定提交到首页，`method="post"` 指定提交方式。输入框的 `name="question"` 对应 `index()` 中的这行：

PYTHON

```
question = request.form.get("question", "").strip()
```

`get()` 按字段名取值，缺失时返回空字符串；`strip()` 去掉首尾空白。打开网页通常产生 GET，按发送产生 POST，因此只在 POST 分支中读取问题并回复。

文件末尾的 `app.add_url_rule()` 将 `/` 与 `index` 关联，并允许 GET、POST。`view_func=index` 传入函数本身；有请求时 Flask 才调用它，所以这里不写 `index()`。

Python 字符串怎样变成网页

`make_page()` 用 `page = page + ...` 拼接字符串，再 `return page`。浏览器将其中的 HTML 标记显示为页面：`h1` 是标题，`p` 是段落，`textarea` 是输入框，`button` 是按钮。`pre` 保留换行，`white-space: pre-wrap` 允许长行折行。

用户输入和模型回复显示前要经过 `html.escape()`。这样 `<b>你好</b>` 会显示为原文，而不会被解释为网页标签。固定的 HTML 标记不用转换，动态插入的文字需要转换。

试一试：修改标题和演示回复，保存、停止并重启程序。再发送空白和 `<b>你好</b>`，检查提示与原文显示。起始骨架只显示最近一次结果，输入框保留刚才的文字，还没有历史功能。

2. 接入真实模型

现在 `get_reply(question)` 接收一个字符串、返回一个字符串。先保持这个接口不变，只替换函数内部生成回答的方式。

读取三项配置

按[运行指南](#)设置完整请求地址、模型名称和 API Key。本教程使用 Chat Completions 格式的接口。

文件顶部增加 `import os` 和 `import requests`，然后在函数外读取配置：

PYTHON

```
api_url = os.environ.get("LLM_API_URL", "")
api_key = os.environ.get("LLM_API_KEY", "")
model_name = os.environ.get("LLM_MODEL", "")
```

这些值在启动时读取，修改配置后要重启。在 `get_reply()` **开头** 检查三项是否为空；缺少配置时，可用 `raise ValueError("请先配置 API 地址、模型和 Key。")` 表示本次调用失败。后面会在调用处处理它。

准备消息和请求

接口接收的是带身份的消息列表。在 `get_reply(question)` 内，先把当前问题表示为：

PYTHON

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

`[]` 是列表，`{}` 是字典。role 表示说话者，user 代表用户，content 是正文。

接着准备两个字典：headers 保存认证信息，data 保存请求内容。

字典	字段	对应的值
headers	Authorization	"Bearer " 加上 api_key，保留中间空格
data	model	model_name
data	messages	刚准备的消息列表
data	stream	Python 的 False，等待完整回答

变量准备好后，在 `get_reply()` 中使用下面的局部写法：

PYTHON

```
response = requests.post(
    api_url, headers=headers, json=data, timeout=60,
    allow_redirects=False
)
response.raise_for_status()
result = response.json()
```

`json=data` 将字典按 JSON 格式发送；`timeout` 设置网络等待超时；`allow_redirects=False` 不跟随地址跳转。
`raise_for_status()` 检查 HTTP 错误，`response.json()` 把正文解析为 Python 数据。

典型的返回数据如下，不要把它写死进程序：

返回数据示意 / JSON

```
{
  "choices": [
    {"message": {"role": "assistant", "content": "你好! "}}
  ]
}
```

用 `result["choices"][0]["message"]["content"]` 依次取出候选列表、第一项、消息和正文。把正文记为 `answer`，检查它是非空字符串，再用 `return` 返回。`isinstance(answer, str)` 可检查类型；确认是字符串后，再用 `answer.strip()` 检查是否只有空白。格式不对时抛出 `ValueError`。

原来的演示 `return` 要移除，否则函数会提前返回。完成后，`get_reply()` 应按顺序完成：检查配置、准备请求、发送请求、提取并返回回答。

调用失败时显示提示

在 `index()` 的非空输入分支中，把原来的单行调用改成下面的形式，保留它所在的缩进：

PYTHON

```
try:
    answer = get_reply(question)
except (
    requests.RequestException, ValueError,
    KeyError, IndexError, TypeError
):
    error_text = "未能取得回答，请检查配置、网络和返回数据。"
```

成功时得到 `answer`；出现指定类型的错误时执行 `except`，页面通过 `error_text` 显示提示。这些类型覆盖常见网络、配置、解析和返回结构错误；其他程序错误仍需查看终端中的报错行号。

试一试：重启后发送简单问题，确认收到真实模型回答，并把网页的演示说明改成实际状态。若失败，可在请求后临时打印 `response.status_code` 辅助排查，不要打印 Key 或认证头。

此时只有单轮问答。成功后直接显示 POST 结果，刷新可能重复发送；先避免刷新结果页，第 4 节会处理。

3. 为请求加入上下文

先说“样品甲的质量是 3 千克”，再问“它的两份总质量是多少”。现在第二次请求只有后一句，接口不会自动保存第一次问答。上下文需要由程序保存，并随下一次请求一起发送。

建立历史列表

在文件顶部、`app = Flask(__name__)` 后加入 `history = []`，保存已完成的问答。不要放进 `index()`，否则每次请求都会重新建立空列表。

一轮完成后的数据形状类似：

PYTHON

```
[
    {"role": "user", "content": "样品甲的质量是 3 千克。"},
    {"role": "assistant", "content": "好的，我记住了。"}
]
```

程序应从空历史开始，存入实际问题和回答；不要把示例文字当成真实历史。模型回答的角色是 `assistant`。

同步修改函数与调用处

消息列表现在应由了解历史的 `index()` 准备，需要一起调整三处：

位置	修改方式
函数定义	将 <code>get_reply(question)</code> 改为 <code>get_reply(messages)</code>
函数内部	删除“只用当前问题重新建立 <code>messages</code> ”的语句，请求体使用传入列表
<code>index()</code> 调用处	先准备完整消息列表，再传给 <code>get_reply()</code>

返回值仍是回答字符串，配置、网络请求和结果解析可以沿用。

为本次请求另建列表

在 `index()` 的非空输入分支中，将当前问题表示为用户消息；另建一个请求列表，依次放入历史消息，再放入当前问题。

不要直接写 `messages = history`：这会让两个变量指向同一个列表，追加问题就提前改动了历史。先看这个独立例子：

PYTHON

```
saved = ["甲", "乙"]
working = []
for item in saved:
    working.append(item)
working.append("丙")
print(saved)    # ['甲', '乙']
print(working)  # ['甲', '乙', '丙']
```

将这个思路用于消息组织。这里复制外层列表，旧消息字典仍被引用；我们只追加新消息，不修改旧字典，因此可以这样做。

成功后再保存完整问答

请求列表是准备发送的内容，`history` 是已经完成的问答。只有 `get_reply()` 正常返回，才把本次用户消息及相应的 `assistant` 消息按顺序追加进历史。

在已有 `try/except` 后增加同一级缩进的 `else`：，把两条消息的保存放在这里。`else` 仅在 `try` 没有报错时执行。不要只存用户消息，也不要吧失败提示存成模型回答。

试一试：连续问两个依赖前文的问题，在调用前临时 `print(messages)`。不额外添加系统消息时，三次请求应分别包含 1、3、5 条消息。页面此时仍只显示最近一轮；先确认模型实际收到历史，再完善显示。

4. 显示整段历史

让页面从 `history` 读取内容，GET 就能重新显示整段会话，无需依赖刚才 POST 中的单条问题与回答。

改变页面函数的输入

将函数参数改为 `make_page(history, error_text, input_text)`，分别表示历史、错误提示和输入框中保留的文字。

同步把 `index()` 最后的调用改为 `make_page(history, error_text, question)`；若自己增加过其他返回分支，也要一起改。调用处的 `question` 传入后叫 `input_text`，名称可以不同，但位置和含义要对应。

在 `make_page()` 内完成两处替换：

- 删除原来的 `if answer != ""`：判断及其显示最近问答的整个代码块，在原位置改为遍历 `history`，不再使用单独的 `answer` 和 `question`。
- 输入框中原来的 `question` 改用 `input_text`。

标题、错误提示、表单及最后的 `return page` 可以保留。

用循环生成消息区域

在标题和错误提示后、表单前，用 `for message in history:` 逐条取出消息。读取 `message["role"]` 决定显示 “我” 还是 “AI”，读取 `message["content"]` 作为正文。

沿用骨架中的拼接方法：先追加身份段落，再追加正文的 `pre` 段落。正文仍需 `html.escape()`。每条消息的显示代码放在循环内；页面开头、表单和 `return page` 放在循环外，否则可能重复输入框或只显示第一条消息。

空历史会让循环执行零次，页面仍应有输入框。保存并重启，确认两轮问答都能显示。

成功后跳转，避免刷新重复发送

在 Flask 导入中增加 `redirect`。在 `index()` 成功保存问答的分支末尾，使用 `return redirect("/", code=303)`。浏览器随后 GET 首页，只读取历史，不再次调用模型。

函数末尾的 `return make_page(...)` 仍要保留，用于首次 GET、空输入和调用失败。失败时保留 `question` 供修改；成功跳转后的 GET 会把它初始化为空，输入框自然清空。不要同时清空 `history`。

试一试：发送两轮后刷新页面，历史应保留，且不产生新模型请求。等待回复时只点一次发送；303 跳转解决的是成功后刷新，没有防止连续点击的功能。

5. 新建并切换会话

新建时应保留旧历史，再创建一段空历史。直接清空原来的 `history` 会丢失旧记录。

用外层列表保存多段会话

将文件顶部的 `history = []` 替换为 `conversations = [[]]`。外层列表保存各段会话，内层列表保存该会话的消息：

流程与数据结构

`conversations`

- [0] 第一段对话的消息列表
- [1] 第二段对话的消息列表
- [2] 第三段对话的消息列表

先在 `index()` 中用 `history = conversations[0]` 选择第一段，后面的请求、保存和显示继续使用 `history`。运行确认原有功能正常，再加入编号选择。

这里故意引用原列表，追加消息才能存回所选会话；第 3 节准备请求时另建列表，是为了避免提前改变历史。两处目的不同。

从网址读取当前编号

约定 `?chat=0` 选择第一段，`?chat=1` 选择第二段。网页显示可以从 “对话 1” 开始，内部下标仍从 0 开始。

在 `index()` 开头，用 `request.args.get("chat", "0")` 读取字符串，再用 `int()` 转为整数 `chat_number`。
`request.args` 读取网址参数，`request.form` 读取表单输入。

转换放进 `try/except ValueError` 中。转换失败或编号不满足 $0 \leq \text{chat_number} < \text{len}(\text{conversations})$ 时，调用 `abort(404)` 拒绝访问；记得在 Flask 导入中增加 `abort`。先验证，再取列表。Python 接受负数下标，但我们的编号约定不接受。

验证后，将固定取第一段的语句改为按 `chat_number` 取出 `history`。请求只能使用这段历史，不能合并整个 `conversations`。

增加新建按钮

在 `index()` 后、路由注册前定义 `new_chat()`。它不接收参数，也不调用模型，只做三件事：追加新的空列表，取得其下标，再跳转到对应地址。

用 `append()` 添加新的 `[]`，末项下标是 `len(conversations) - 1`。每次都建立新列表，不能复用旧列表。编号拼进地址前用 `str()` 转为字符串；跳转仍使用 `redirect(..., code=303)`。

保留首页路由，在它附近注册新地址：

PYTHON

```
app.add_url_rule("/new", view_func=new_chat, methods=["POST"])
```

在 `make_page()` 中增加独立表单，放在发送表单之外，不要嵌套：

网页标记 / HTML

```
<form action="/new" method="post">
  <button type="submit">新建对话</button>
</form>
```

这是 HTML。仿照骨架，把每行作为带引号的字符串追加进 `page`，不要将标签直接当作 Python 语句粘贴。点击后会 POST 到 `/new`，执行新建并跳转。

在页面与请求之间保留编号

页面需要知道当前编号。在已有三个参数前加上 `chat_number`，得到 `make_page(chat_number, history, error_text, input_text)`；`index()` 中所有调用也在相同位置传入编号。

在标题后循环生成旧会话入口。用 `range(len(conversations))` 得到各个下标，将下标放进链接地址，将下标加一作为显示编号。例如：

网页标记 / HTML

```
<a href="/?chat=1">对话 2</a>
```

用 `str()` 将整数转成字符串后再拼接。这个循环只生成入口；消息区域仍只遍历所选 `history`。

发送与跳转也要保留编号，否则第二段会话可能把问题发回默认的第一段。检查以下位置是否一致：

位置	应使用的信息
index()	读取、验证 chat_number，选择该段历史
make_page() 及其调用处	接收同一个编号和所选历史
发送表单的 action	带当前编号，例如 /?chat=1
保存成功后的 303 跳转	回到相同的带编号地址

新建表单仍提交到 /new，不要跟着改。可以先实现新建，再添加旧会话链接，最后检查发送与跳转地址。

试一试：在对话 1 说“样品代号是 Q73”，再新建对话 2。检查其首次请求不含 Q73；回到对话 1，旧记录仍在且能继续提问。以实际请求内容为准，不只看模型是否碰巧答对。

6. 自查与提交

回头读自己的程序，确认各部分分工清楚：

位置	职责
文件顶部	导入库、读取配置、建立会话存储
get_reply(messages)	调用模型，返回回答字符串
make_page(...)	显示历史、会话入口和表单
index()	选会话、收输入、组装请求、保存成功结果
new_chat()	建立空会话并跳转
文件末尾	注册地址、启动服务

按完成后的自查实际操作，再参考 README 写作提示写自己的说明。提交代码和 README 到服务器个人目录的 homework1 文件夹。

本路线用内存列表保存记录，重启程序后会丢失；同一进程的多个浏览器窗口共享记录。请说明自己的实现范围，再决定是否进一步改进。正式要求以课程发布内容为准。

完成后自查

用自己的程序实际操作一次，把结果记进自己的 README。下面是帮助你查错的场景，不是额外的评分表。

先检查作业的主要功能

操作	观察什么
发送一个简单问题	得到真实模型回答，已不再使用演示函数
先给出一个新信息，再追问它	实际请求包含此前问答；页面也能显示整段历史
新建对话后发送问题	新会话从空历史开始，请求不含旧会话的消息
回到旧会话并继续提问	原来的记录仍在，后续请求属于这段会话
在第二段会话中连续发送两次	页面、表单提交和跳转始终指向同一个编号

调试时可以临时打印发送的消息列表，确认实际内容；不要只凭模型“好像记住了”判断，也不要打印 Key 或认证头。完成检查后去掉不需要的调试输出。

再检查容易出错的情况

- 发送空白：应给提示，不调用模型，也不增加历史。
- 输入 `<b>你好</b>` 或带多行的文字：页面应显示原文，不把它当成 HTML 执行。
- 成功发送后刷新：不会重新调用模型或重复追加问答。
- 模型调用失败：已有历史不变，页面有提示。可以临时使用无效模型名做一次检查，随后恢复；也可以在自己的本地调试中模拟失败。
- 打开 `?chat=abc`、`?chat=-1` 或不存在的编号：程序拒绝这个地址，不选错会话。
- 停止后重启：确认历史是否仍在，并在 README 如实说明。教程中的内存列表实现会丢失记录。

能解释这些地方，就更容易自己改动

浏览器里的 `name="question"` 与 Python 的哪一行相对应？为什么历史不能在 `index()` 中每次初始化？为什么新建会话要追加空列表？为什么成功后才保存问答？为什么选中历史可以直接引用列表，而准备请求要另建列表？

不必另交这些问题的书面答案。它们用于帮你沿着代码梳理自己的实现。

README 写作提示

在自己的 `homework1` 中写一份简短的 `README.md`，说明以下内容，以实际实现和验证结果为准。

- **运行方法**：Python 环境、必要依赖、入口文件、API 配置，以及启动和浏览器访问方式。不要写真实 Key。
- **实现思路**：消息如何表示，历史在哪里保存，怎样选中和新建会话，为什么不会混入旧内容。
- **保存范围**：刷新、切换、新建和重启分别怎样影响记录。
- **验证结果**：写实际操作、结果和仍存在的问题，不要把教程的预期结果当成自己的记录。
- **AI 辅助开发**：写帮助编程的模型具体型号、平台、消耗的额度或 token，以及自己做过的验证；多个模型或平台分别记录。这与程序聊天接口使用哪个模型不是同一件事。

及时记录 AI 用量，以平台实际信息为准。平台不显示时如实说明，并向助教确认登记方式，不要编造数值。完全没有使用 AI 时，按要求备注“纯手搓”。

附录 A • 起始骨架

下面仅是最初的演示程序，尚未实现 API、上下文和新建会话。代码与配套 app.py 行为一致，少数长行仅作等价分行。

APP.PY / 起始骨架

```
"""教程的起始骨架：能打开网页并演示回复，还没有模型调用或历史功能。"""
```

```
import html
from flask import Flask, request

app = Flask(__name__)

def get_reply(question):
    # 先用演示回复走通网页流程，随后在这里接入真实模型。
    return "这是演示回复，我收到了：" + question

def make_page(question, answer, error_text):
    page = '<!doctype html><html lang="zh-CN"><head>'
    page = page + '<meta charset="utf-8">'
    page = page + '<meta name="viewport" content="width=device-width, initial-scale=1">'
    page = page + '<title>我的对话程序</title></head><body>'
    page = page + '<h1>我的对话程序</h1>'
    page = page + '<p>起始骨架：目前只有演示回复，尚未接入 AI，也不保存历史。</p>'

    if error_text != "":
        page = page + '<p>' + html.escape(error_text) + '</p>'

    if answer != "":
        page = page + '<h2>这次的输入</h2>'
        page = page + '<pre style="white-space: pre-wrap">' + html.escape(question) + '</pre>'
        page = page + '<h2>这次的回复</h2>'
        page = page + '<pre style="white-space: pre-wrap">' + html.escape(answer) + '</pre>'

    page = page + '<form action="/" method="post">'
    page = page + '<p><label for="question">输入内容</label></p>'
    page = page + '<textarea id="question" name="question" rows="4" cols="30">'
    page = page + html.escape(question) + '</textarea><br>'
    page = page + '<button type="submit">发送</button></form>'
    page = page + '</body></html>'
    return page
```

APP.PY / 起始骨架 / 续

```
def index():
    question = ""
    answer = ""
    error_text = ""

    if request.method == "POST":
        question = request.form.get("question", "").strip()
        if question == "":
            error_text = "请先输入内容。"
        else:
            answer = get_reply(question)

    return make_page(question, answer, error_text)

app.add_url_rule("/", view_func=index, methods=["GET", "POST"])

if __name__ == "__main__":
    app.run(host="127.0.0.1", port=8500, debug=False, threaded=False)
```

运行：在服务器个人目录的 homework1 中，使用运行指南给出的 Python 解释器执行 app.py。沿第 1 至第 5 节逐步完善它。

附录 B • 排错与资料

常见问题

现象	优先检查
网页打不开或仍是旧页面	两个终端、端口、服务器文件是否已保存和重启
语法或缩进错误	报错行及前一行的引号、括号、冒号和缩进
参数数量错误	函数定义与所有调用是否同步修改
提交后出现 405	路由是否允许 POST，表单地址是否正确
仍显示演示回复	是否删除了演示用的提前 return
401、403、404	认证、权限、完整请求地址和模型
429 或超时	额度、频率、网络和服务状态
第二段会话发回第一段	action 和成功跳转是否带当前 chat 编号

程序错误要到终端 A 查看具体行号，再结合正在修改的函数排查。

查阅资料

- [Flask 快速入门](#)：网页、表单、请求与跳转。
- [Requests 快速入门](#)：JSON 请求、超时与错误处理。
- [DeepSeek 多轮对话](#)：消息格式与历史传递。

官方资料用于查具体用法。本文只采用完成作业所需的基础功能，评分与提交要求以课程发布内容为准。

附录 C · 正式作业要求

以下按当前作业文件收录，未增加评分条件。

作业目标

使用 Python 编写一个基于 LLM API 的网页多轮对话程序，实现在浏览器中打开网页进行对话。

一、程序要求

- 接入至少一个课程支持的 LLM API。
- 支持保留上下文的连续对话。
- 支持新建对话和保留历史记录；新建后，旧会话内容不得进入后续请求。
- 对话以网页的形式打开。

二、提交内容

- **程序代码**：包含运行入口及必要的额外依赖说明（最好是能够在服务器上的pubpy环境下直接运行）。
- **README.md**：说明运行方法、实现思路，保持简洁。

允许使用 AI 辅助开发。需要在 README 中简要说明使用了什么模型（具体型号）来帮助编程、在什么平台上进行的、消耗了多少额度/token，以及自己进行了哪些验证。

如果完全没有使用AI可以备注“纯手搓”。

三、评价重点

主要评价程序功能和对话界面的实现。

提交位置

请在服务器上的个人目录下建立一个“homework1”文件夹，然后将你的作业代码全部放在该文件夹。

作业截止日期：10月3号23:59