FastAPI ——web 框架

FastAPI 是一个用于构建 API 的现代、快速（高性能）的 web 框架，专为在 Python 中构建 RESTful API （是什么？）而设计。

FastAPI 使用 Python 3.8+ 并基于标准的 Python 类型提示。

FastAPI 建立在 Starlette 和 Pydantic 之上，利用类型提示进行数据处理，并自动生成 API 文档。

FastAPI 于 2018 年 12 月 5 日发布第一版本，以其易用性、速度和稳健性在开发者中间迅速流行起来。

FastAPI 极简实例

Fastapi 用于构建前端 web 模块如何绑定到后端的函数，那么自然就有 2 大部分，第一部分是了解前端页面的搭建，如何传递信息到后端，第二部分是后端的处理函数，核心自然是两者之间的通信，又可细分为如何建立通信（路由），通信时携带的数据是什么样的。

简单来说，以 upload 按钮为例，在构建前端时，会设置按钮触发的通信路径为 action='/upload'，点击按钮时，会把前端的信息封存为特定通信格式的数据发送请求到后端；

后端的函数 def upload 会通过装饰器函数@绑定到/upload 路径上，接收到前端的请求后，会解析出请求里的数据，然后调用函数处理。

两者通信的域名设置在 uvicorn.run(app,host='0.0.0.0',port=8000)里。

在此基础上，可进一步地探究异常处理，中间件，高并发等。

下面是一个极简的实现，结合这些介绍一下 fastapi 的核心架构和功能模块。

```
from fastapi import FastAPI

app = FastAPI()#实例化

# 路由装饰器绑定 GET / 到根路径处理函数

@app.get("/")

async def root():

    return {"message": "Hello World"}
```

核心功能模块详解

URI&URL&路径&路由

在 FastAPI 中，**路径（Path）**、**路由（Route）** 和 **URI（Uniform Resource Identifier）** 是紧密相关的概念，但它们有不同的具体含义和用途。以下是详细解释：

URI：统一资源标识符（Uniform Resource Identifier），是一个广义概念，表示网络上的唯一资源标识。例如：http://example.com/items/42。

URL：统一资源定位符（Uniform Resource Locator），是 URI 的子集，**表示资源的网络位置和访问方式**。例如：http://example.com/items/42 是 URL，包含了协议（http）、域名（example.com）和路径（/items/42）。

路径（Path）：在 Web 开发中，通常用 **路径（Path）** 指代 URL 中域名后的部分（如 /items/42），标识要访问的资源或端点（Endpoint）

举个简单的例子，比如 bilibili.com 是域名，可以访问到 bilibili，域名后面的 video/BV1or4y1B7o2?vd_source 等等信息则是指具体的视频所在的位置，也即路径。

路由：将 **HTTP 方法（GET、POST 等）** 和 **路径** 绑定到后端函数（Endpoint Function）的规则。

组成：HTTP 方法 + 路径，例如 GET /items 或 POST /users。

作用：定义客户端将请求以什么样的方式发送到服务器哪个端点，指定处理该请求的函数。

FastAPI 的路由系统（Routing）

通过操作前端界面的按钮等部件，会执行相应的操作，那么这个“按钮”和对应的执行函数是如何对应起来的呢？

在 FastAPI 中，路径（URL 路由）和执行函数的绑定是通过 **路由装饰器（Route Decorators）** 或 **显式注册** 的方式实现的。以下是两种方式的详细解释：

装饰器语法（常见写法）

FastAPI 最常用的方式是通过装饰器（前面加@符号）将路径和 HTTP 方法（如 POST、GET）与处理函数绑定。例如：

```
# 用装饰器绑定路径和函数

@app.post("/knowledge_base/upload_docs")

def upload_docs():

    return {"message": "File uploaded"}
```

显式注册语法

```
# 显式调用 app.post() 方法，并传入处理函数
app.post(
    "/knowledge_base/upload_docs",
    tags=["Knowledge Base Management"],
    response_model=BaseResponse,
    summary="上传文件到知识库，并/或进行向量化"
)(upload_docs)  # 将处理函数 upload_docs 作为参数传递
```

app.post() 方法本身返回一个 **路由注册函数**。

通过链式调用 (...)(upload_docs)，实际是将 upload_docs 函数作为参数传递给 app.post() 返回的函数。

参数	类型/值	作用说明	示例场景
路径 "/knowledge_base/upload_docs"	字符串	后端函数的路径，和域名一起组成服务器端点。	用户上传文件时访问 http://api.com/knowledge_base/upload_docs
tags	["Knowledge Base Management"]	**分组标签**，在 Swagger/Redoc 文档中归类显示。	在文档侧边栏生成 Knowledge Base Management 分类，包含所有相同标签的接口。
response_model	BaseResponse	**响应模型**：指定接口返回的数据结构（基于 Pydantic 模型），自动校验和序列化返回值。	确保返回的 JSON 包含 code、msg、data 字段，如：{"code":200, "msg":"success", "data":{}}
summary	"上传文件到知识库，并/或进行向量化"	**接口摘要**：说明接口的核心功能。	在文档中直接展示该描述。
处理函数 upload_docs	函数对象	**请求处理器**：实际处理 POST 请求的函数	用户上传文件时，该函数接收请求数据，保存文件到知识库，并触发向量化操作。

这种方式与装饰器语法完全等效，只是写法不同。方便将路径和处理函数分开写。

在某些场景下，显式注册更灵活：

动态路由：根据条件动态生成路径和绑定函数。

代码分离：将路由配置和处理函数定义分开（例如路由集中管理）。

避免循环导入：当处理函数和 FastAPI 应用实例不在同一文件时。

功能逻辑

无论是装饰器语法还是显式注册，其调用函数的逻辑是一样的，功能均是将待调用的函数 upload_docs 注册到路径 /knowledge_base/upload_docs 的 POST /GET 方法上。

（1）装饰器语法会将 post/get 和调用的函数写在一起，而显式注册则可以将其二者分开写到 2 个文件里，

（2）当客户端发送 POST 请求到 路径/knowledge_base/upload_docs 时，upload_docs 函数会被调用。注意这里的路径只是一个逻辑节点，并没有实际的文件等物理资源，用于触发调用函数执行，而调用函数内部则可以访问真正的物理资源。这里的路径在遵循命名规范的前提下，可以任意指定，不过尽可能清晰地描述绑定函数的作用。

（3）路径前的域名，在开发环境中，FastAPI 默认运行在本地主机的某个端口（如 8000），此时域名通常是 localhost 或 127.0.0.1。而在生产环境中，域名需要通过购买并配置 DNS 记录，将域名指向服务器的 IP 地址，然后通过反向代理将请求转发到 FastAPI 应用运行的端口。

匹配规则：静态路径优先：/items/fixed 比 /items/{item_id} 优先级高
路径参数：{item_id} 捕获动态值
类型转换：{item_id: int} 自动转换为整数类型
匹配顺序：按声明顺序从上到下匹配
绑定过程：装饰器 @app.get() 注册路径和 HTTP 方法
将函数 read_item 添加到路由表
路径参数自动映射到函数参数

底层原理

无论是装饰器语法还是显式注册，最终都会调用 FastAPI 的 add_api_route 方法，将路径、HTTP 方法和处理函数绑定。以下是伪代码解释：

```
# FastAPI 内部伪代码

class FastAPI:
```

```
    def post(self, path: str, **kwargs):
        # 1. 创建一个路由对象，记录路径和配置（如 tags、response_model）
        route = APIRoute(path=path, methods=["POST"], **kwargs)
        # 2. 返回一个函数，该函数接受处理函数并绑定到路由
        def register_handler(handler: Callable):
            self.routes.append(route(handler))  # 将路由添加到应用
            return handler
        return register_handler
# 显式注册时：
app.post("/path", ...)(upload_docs)  # 等价于装饰器语法
```

http 方法 post() 和 get()

app.post() 和 app.get() 都是前端与后端联系的方法，主要区别就是 get 告诉服务器想要获取什么资源，并且这些信息直接写在 URL 里，而 post 是将信息上传到服务器上，数据量大一点。

特性	app.post() 和 app.get() 的共性
路由注册	均用于将路径与后端处理函数绑定，定义客户端访问的 API 端点。
装饰器语法	都支持 FastAPI 的装饰器语法和显式注册语法
参数配置	共享相同的配置参数（如 tags、response_model、summary），用于生成 API 文档和校验响应格式。
请求处理	均能处理客户端请求，执行业务，并返回响应（支持同步和异步处理）。

特性	app.get()	app.post()
HTTP 方法	处理 **HTTP GET 请求**（请求数据通过 URL 参数传递）。	处理 **HTTP POST 请求**（请求数据通过请求体传递）。
主要用途	用于 **获取资源**（如查询数据、加载页面），操作是幂等的（多次请求结果相同）。	用于 **提交数据**（如创建、修改资源），操作通常非幂等（多次请求可能产生不同结果）。
数据传递方式	数据通过 **URL 路径参数** 或 **查询参数** 传递（如 /items/1?limit=10）。	数据通过 **请求体（Body）** 传递（支持 JSON、表单、文件上传等格式）。
数据安全性	参数暴露在 URL 中，可能被浏览器历史记录或日志记录（不适合敏感数据）。	数据在请求体中，更安全（适合传输密码、文件等敏感信息）。

特性	app.get()	app.post()
数据长度限制	受 URL 长度限制（通常不超过 2048 字符）。	无严格长度限制，适合传输大量数据（如文件上传）。
缓存机制	可被浏览器缓存（适用于频繁查询的静态数据）。	默认不被缓存（每次请求可能触发服务端状态变更）。
FastAPI 参数声明	使用 Query、Path 声明参数：def get_item(item_id: int = Path(...))	使用 Body、Form、File 声明参数：def create_item(item: Item = Body(...))

关键选择建议

遵循 RESTful 设计原则，根据操作类型选择对应的 HTTP 方法（如 GET 查、POST 增、PUT 改、DELETE 删）

条件	app.get()	app.post()
操作类型	查询、读取数据（不修改服务器状态）	创建、更新、删除数据（修改服务器状态）
数据敏感性	非敏感数据（如搜索关键词）	敏感数据（如密码、支付信息）
数据大小	少量数据（参数简单）	大量数据（如文件、复杂 JSON）
幂等性要求	需要幂等（多次请求结果一致）	允许非幂等（每次请求可能产生不同结果）

FastAPI 的数据传递

数据格式

FastAPI 使用 Python 类型提示来定义数据类型，主要分为以下几类：

路径参数

```
@app.get("/items/{item_id}")
async def read_item(item_id: int):   # int 类型
    return {"item_id": item_id}
```

查询参数

```
@app.get("/items/")
async def read_items(skip: int = 0, limit: int = 10):   # 默认值表示查询参数
    return {"skip": skip, "limit": limit}
```

请求体参数（Pydantic 模型）

```
from pydantic import BaseModel
class Item(BaseModel):
    name: str；description: Optional[str] = None；price: float；is_offer: bool = False
```

List[str]: 字符串列表；Dict[str, int]: 字典；datetime: 日期时间；EmailStr: 邮箱格式字符串

数据类型的 json 到底是指什么

在 FastAPI 中，当我们说“JSON”时，通常指的是通过 HTTP 请求发送和接收的数据，这些数据以 JSON（JavaScript Object Notation）格式进行编码。JSON 是一种轻量级的数据交换格式，基于 JavaScript 对象表示法的子集，但独立于编程语言。JSON 使用人类可读的文本来传输由键值对和数组组成的数据对象，易于人阅读和编写，同时也易于机器解析和生成。

JSON 的基本结构

Json 支持的数据类型：**字符串，数字**（整数或浮点数），**布尔值**，**null**，**对象**：键值对的集合，**数组**：值的有序集合

键值对集合：类似于 Python 中的字典

{"name": "John", "age": 30, "city": "New York"}

有序值列表：类似于 Python 中的列表

["apple", "banana", "orange"]

嵌套结构：可以嵌套对象和数组

```
{
  "person": {
    "name": "Alice",
    "hobbies": ["reading", "swimming"]
  },
  "id": 12345}
```

在 FastAPI 中，JSON 数据类型主要对应于 Pydantic 模型和 Python 的基本数据类型（如 dict、list、str、int、float、bool 等）。

与 python 基本类型的关系

FastAPI 会自动将请求中的 JSON 数据解析为相应的 Python 对象（json 的反序列化），并将 Python 对象转换为 JSON 作为响应返回（JSON 的序列化）。

JSON 类型	Python 类型	示例
string	str	"hello" → "hello"
number	int/float	42 → 42 / 3.14 → 3.14
boolean	bool	true → True / false → False
null	None	null → None
array	list	[1, 2, 3] → [1, 2, 3]
object	dict	{"key": "value"} → {"key": "value"}

什么是 Pydantic 模型？

Pydantic 是一个用于数据验证和设置管理的 Python 库，它使用 Python 类型注解来验证数据，继承自 pydantic.BaseModel。在 FastAPI 中，Pydantic 模型是核心组件，用于定义 API 的输入和输出数据结构，还提供了强大的数据验证、序列化和文档生成功能。

1. 数据验证：自动验证输入数据是否符合定义的字段类型和约束条件。
2. 数据转换：自动将输入数据转换为正确的 Python 类型。
3. 序列化：将 Python 对象转换为 JSON 或其他格式。
4. 文档生成：自动生成 API 文档（在 FastAPI 中特别有用）。
5. 设置管理：管理应用程序的配置和设置。
6. 默认值处理：为字段提供默认值。
7. 复杂嵌套支持：支持复杂的数据结构和嵌套模型。

1. 定义基本模型

```
from pydantic import BaseModel
class User(BaseModel):
    id: int；  name: str；  signup_date: datetime = datetime.now()
    is_active: bool = True；  tags: List[str] = []；  bio: Optional[str] = None
```

2. 创建模型实例

```
#  使用字典创建
user_data = {
    "id": 1,
    "name": "John Doe",
    "email": "john@example.com",
    "tags": ["developer", "python"]}
user = User(**user_data)
#  直接创建
user = User(
    id=1,
    name="John Doe",
    email="john@example.com",
    tags=["developer", "python"])
```

3. 访问模型数据

```
print(user.id)              # 1
print(user.name)            # "John Doe"
print(user.email)           # "john@example.com"
print(user.tags)            # ["developer", "python"]
```

4. 转换为字典或 JSON

```
#  转换为字典
user_dict = user.dict()
#  转换为 JSON
user_json = user.json()
```

我们定义了一个 User 模型，它包含了多种基本数据类型（字符串、整型、浮点型、布尔型、列表等）。当我们通过 Streamlit 前端提交表单时，数据被组

织成一个字典，然后通过 requests.post 以 JSON 格式发送到 FastAPI 后端。FastAPI 后端自动验证并转换这些数据为 User 模型实例。

Fastapi 支持的数据类型

FastAPI 使用 Python 的类型提示（type hints）来定义 API 的参数和模型。这些类型提示包括 Python 的基本数据类型（如 int, str, float, bool, dict, list 等），

FastAPI 会自动将请求中的 JSON 数据解析为相应的 Python 对象（json 的反序列化），并将 Python 对象转换为 JSON 作为响应返回（JSON 的序列化）这些基本类型在 FastAPI 中会被用于自动的请求解析和响应序列化。例如，如果一个端点期望一个 int 类型的参数，FastAPI 会尝试将接收到的字符串参数转换为整数，如果转换失败则返回错误。

对于复杂结构，FastAPI 通常使用 Pydantic 模型（继承自 BaseModel 的类）来定义请求体和响应模型，而不是直接使用 Python 的 dict。Pydantic 模型提供了数据验证、序列化和文档生成。

FastAPI 还支持使用 List、Set、Tuple 等来自 typing 模块的泛型类型，以指定集合中元素的类型。

示例：
在 FastAPI 中，你可以这样定义一个路径参数和查询参数：

```
@app.get("/items/{item_id}")
async def read_item(item_id: int, q: str = None):
    return {"item_id": item_id, "q": q}
```

这里，item_id 被声明为 int 类型，FastAPI 会自动将 URL 路径中的字符串转换为整数。如果无法转换，将返回 422 错误。

而对于请求体，我们通常使用 Pydantic 模型：

```
class Item(BaseModel):
    name: str
    description: str = None
@app.post("/items/")
async def create_item(item: Item):
  return item
```

这里，Item 是一个 Pydantic 模型，它定义了请求体的结构。FastAPI 会自动验证传入的 JSON 对象是否匹配这个模型，并将 JSON 字段转换为相应的类型。

同样，当返回一个 Pydantic 模型实例或任何可转换为 JSON 的对象（如字典、列表）时，FastAPI 会自动将其转换为 JSON 并设置适当的 Content-Type 头（application/json）。

此外，FastAPI 还支持一些额外的类型，例如：

UploadFile：用于文件上传

Form：用于表单字段

Body、Query、Path、Header：用于明确指定参数来源并添加额外验证

在 Streamlit 中，我们使用 requests 库向 FastAPI 后端发送 HTTP 请求，其中：

发送数据时，我们使用 json 参数，它会自动将 Python 字典转换为 JSON 并设置适当的请求头。

接收响应时，我们使用.json()方法将响应内容解析为 Python 字典或列表。

数据流动与变化

明确整个流程：Streamlit 前端构建表单，用户输入数据，然后通过 HTTP 请求（比如使用 requests 库）发送到 FastAPI 后端，FastAPI 后端接收数据并处理，最后返回响应。

在这个过程中，数据类型的传递和转换涉及以下几个方面：

Streamlit 前端：用户输入的数据在 Streamlit 中通过不同的组件（如 st.text_input, st.number_input 等）获取，这些组件返回的数据类型通常是 Python 基本类型（str, int, float, bool, list 等）。

请求发送：使用 requests 库发送请求时，需要将数据转换为 JSON 格式（即字典和列表的组合），因为 HTTP 请求体通常使用 JSON 格式。

FastAPI 后端：FastAPI 使用 Pydantic 模型来接收数据，自动将 JSON 数据转换为 Python 对象，并进行验证。

常用基本数据类型及变化形式

1. 字符串 (String)
前端输入：st.text_input(), st.text_area()
传递形式：JSON 字符串
后端接收：str 类型或带验证的字符串类型（如 EmailStr）
2. 数字 (Number)
前端输入：st.number_input(), st.slider()
传递形式：JSON 数字（整数或浮点数）
后端接收：int, float 或带验证的数字类型（如 PositiveInt）
3. 布尔值 (Boolean)
前端输入：st.checkbox(), st.radio()
传递形式：JSON 布尔值 (true/false)
后端接收：bool 类型
4. 列表 (List)

前端输入：st.multiselect(), 多选框组
传递形式：JSON 数组
后端接收：List[T] 类型（如 List[str], List[int]）
5. 字典/对象 (Dictionary/Object)
前端输入：多个相关字段组合
传递形式：JSON 对象
后端接收：Pydantic 模型或 Dict[str, Any] 类型
6. 日期时间 (DateTime)
前端输入：st.date_input(), st.time_input()
传递形式：ISO 格式字符串（如 "2023-05-15T10:00:00"）
后端接收：datetime 类型

传递方式

首先，回顾 FastAPI 支持的数据传递方式。根据之前的知识，FastAPI 主要支持以下几种方式：

路径参数（Path Parameters）

直接在 URL 路径中传递（例如/items/42 中的 42），支持自动类型转换（如将字符串转换为整数）。

```
@app.get("/items/{item_id}")  # 定义路径参数 item_id
async def read_item(item_id: int):  # item_id 自动转换为整数类型
    return {"item_id": item_id}  # 输出: {"item_id": 42}
```

测试请求：

```
curl http://localhost:8000/items/42
```

输出：

```
{"item_id": 42}
```

查询参数（Query Parameters）

在 URL 问号后传递（如/items?skip=0&limit=10），用于过滤、分页或可选参数，可设置默认值和可选性。

```
@app.get("/items/")
async def read_items(
    skip: int = 0,          # 默认值 0 (int)
```

```
    limit: int = 10          # 默认值 10 (int)
):
    return {"skip": skip, "limit": limit}
```

测试请求：

```
curl "http://localhost:8000/items/?skip=5&limit=20"
```

输出：

```
{"skip": 5, "limit": 20}
```

请求体（Request Body）

通过 POST/PUT 请求发送 JSON 数据，使用 Pydantic 模型验证和序列化，处理复杂嵌套数据（如 JSON、XML）。

- **传输位置：**在请求正文（Body）中，不在 URL 中显示。
- **数据格式：**支持复杂结构（如嵌套字典、列表），FastAPI 默认解析 JSON 格式（application/json）。
- **适用场景：**传递创建 / 更新资源的详细信息（如注册用户时的用户名、密码、邮箱等结构化数据）。
- **关联技术：**通常与 Pydantic 模型配合使用，用于数据校验和类型转换。

```
# 定义 Pydantic 数据模型
class Item(BaseModel):
    name: str           # 必填字符串字段
    price: float        # 必填浮点数字段
    tax: float = None   # 可选浮点数字段

@app.post("/items/")
async def create_item(item: Item):    #item 就是请求体，自动从 JSON 请求体解析为 Item 对象
    if item.tax:
        total = item.price + item.tax
    else:
        total = item.price
    return {**item.dict(), "total": total}    # 输出包含计算后的总价
```

测试请求：

```
curl -X POST "http://localhost:8000/items/" \
     -H "Content-Type: application/json" \
     -d '{"name":"Laptop", "price": 999.99, "tax": 99.99}'
```

输出：

```
{
   "name": "Laptop",
   "price": 999.99,
   "tax": 99.99,
   "total": 1099.98
}
```

表单数据（Form Data）

通过 HTML 表单提交（application/x-www-form-urlencoded 或 multipart/form-data 编码），本质上也是请求体的一种，但格式与 JSON 不同，适用于传统表单提交场景，需显式声明 Form(...)。

- **传输位置**：在请求正文（Body）中，但格式为 application/x-www-form-urlencoded（普通表单）或 multipart/form-data（含文件的表单）。
- **数据格式**：键值对结构，与查询参数类似，但在请求体中传递（而非 URL），支持简单类型（无法直接传递嵌套结构）。
- **适用场景**：HTML 表单提交（如登录表单的用户名、密码）、需要兼容传统表单的场景。
- **注意**：Form 数据与 JSON 请求体不能同时使用（因为两者都是请求体，FastAPI 默认只能解析一种格式）。

```
@app.post("/login/")
async def login(
    username: str = Form(...),   # 必填表单字段 (str)
    password: str = Form(...)    # 必填表单字段 (str)
):
    return {"username": username, "logged_in": True}    # 输出：{"username": "admin", "logged_in": true}
```

测试请求：

```
curl -X POST "http://localhost:8000/login/" \
     -d "username=admin&password=secret"
```

输出：

```
{"username": "admin", "logged_in": true}
```

文件上传（File Upload）

通过 multipart/form-data 上传文件，支持多个文件和大文件流式传输，通常使用 multipart/form-data 编码，使用 UploadFile 类型处理元数据。

- **传输位置**：在请求正文（Body）中，使用 multipart/form-data 格式（专为二进制数据设计）。
- **数据格式**：二进制文件内容，附加文件名、类型等元信息。
- **适用场景**：上传图片、文档、视频等文件。
- **优势**：UploadFile 提供了便捷的文件操作方法（如 read()、filename、content_type 等），支持大文件流式处理。

```
from fastapi import FastAPI, UploadFile, File#需导入 UploadFile, File 数据类型
@app.post("/upload/")
async def upload_file(
    file: UploadFile = File(...)  # 上传的文件对象 (UploadFile)
):
    contents = await file.read()  # 读取文件内容 (bytes)
    return {
        "filename": file.filename,              # 原始文件名 (str)
        "content_type": file.content_type,  # MIME 类型 (str)
        "size": len(contents)                   # 文件大小 (int)
    }  # 输出包含文件信息
```

测试请求：

```
curl -X POST "http://localhost:8000/upload/" \
     -F "file=@document.pdf"
```

输出：

```
{  "filename": "document.pdf",
  "content_type": "application/pdf",
  "size": 24576}
```

Header 参数

从 HTTP 头中提取参数，常用于身份验证（如 Authorization），自动转换命名格式（如 user-agent → user_agent）。

```
from fastapi import FastAPI, Header
@app.get("/headers/")
async def read_headers(
    user_agent: str = Header(None)  # 从 User-Agent 头获取值 (str)
):
    return {"user_agent": user_agent}  # 输出: {"user_agent": "curl/7.68.0"}
```

测试请求：

```
curl -H "User-Agent: MyBrowser/1.0" http://localhost:8000/headers/
```

输出：

{"user_agent": "MyBrowser/1.0"}

Cookie 参数

从 Cookie 中提取参数，常用于会话管理，需显式声明 Cookie(...)

```
from fastapi import FastAPI, Cookie
@app.get("/cookies/")
async def read_cookies(
    session_id: str = Cookie(None)  # 从 Cookie 获取 session_id (str)
):
    return {"session_id": session_id}  # 输出: {"session_id": "abc123"}
```

测试请求：

curl -H "Cookie: session_id=abc123" http://localhost:8000/cookies/

输出：

{"session_id": "abc123"}

总结

方式	传输位置	数据类型	典型场景
路径参数	URL 路径	基础类型	获取唯一资源（如用户 ID）
查询参数	URL 查询字符串	基础类型	分页、过滤
请求体	HTTP Body (JSON)	复杂对象	创建/更新资源
表单数据	HTTP Body (Form)	键值对	HTML 表单提交
文件上传	HTTP Body (Multipart)	二进制文件	上传图片、文档
Header 参数	HTTP 头	字符串	认证令牌、客户端信息
Cookie 参数	HTTP Cookie	字符串	会话管理

接下来是另一个关键的问题，前面都是使用命令行指令访问的，在实际网页端上的数据形式是如何传递的呢？比如我在网页端填写了用户名和密码，这些数据以什么样的形式，怎么传递到后端的处理函数上呢？以上传文件为例：

在 FastAPI 中，当通过前端网页上传文件并调用 upload_docs 处理函数时，数据传递遵循 **HTTP multipart/form-data** 格式。以下是整个数据传递过程的详细说明：

HTTP 请求格式

前端上传文件时，浏览器会自动构造一个 **multipart/form-data** 类型的请求，包含以下部分：

文件数据：通过 <input type="file"multiple> 选择的文件内容（二进制流），这部分是通过浏览并上传本地文件，每个文件作为独立的 multipart/form-data 部分传输，FastAPI 自动将文件数据包装为 UploadFile 对象列表

表单字段：其他文本参数（如知识库名称、配置选项等），这些就是在前端界面的表格中填的信息，比如知识库名称，基本介绍等。前端通过 JavaScript 将数据序列化为 JSON。发送 POST 请求，设置 Content-Type: application/json

通过 Form(...) 声明的参数从表单字段中提取：

参数名	数据类型	来源示例（HTML 表单）
knowledge_base_name	str	<input type="text" name="knowledge_base_name">
override	bool	<input type="checkbox" name="override">
chunk_size	int	<input type="number" name="chunk_size" value="500">
docs	Json	<input type="hidden" name="docs" value='{"test.txt": [...]}'>

Fastapi 和 uvicorn

uvicorn.run(app, host='0.0.0.0', port=8000) 这行代码是使用 Uvicorn 作为 ASGI（异步服务器网关接口）服务器来运行一个基于 ASGI 规范的应用程序 app。Uvicorn 是一个快速的 Python ASGI 服务器，常用于运行基于 FastAPI、Starlette 等框架开发的 Web 应用。

- app：这是一个遵循 ASGI 规范的应用程序实例。ASGI 是一种用于 Python Web 服务器和应用程序之间通信的标准接口，允许异步处理请求。例如，使用 FastAPI 框架创建的应用程序实例就可以作为 app 传入。
- host='0.0.0.0'：指定服务器监听的 IP 地址。0.0.0.0 表示服务器将监听所有可用的网络接口，意味着无论是本地访问还是通过网络上的其他设备访问，都可以连接到这个服务器。
- port=8000：指定服务器监听的端口号。客户端可以通过这个端口号与服务器进行通信。

当执行 uvicorn.run(app, host='0.0.0.0', port=8000) 时，Uvicorn 服务器开始启动。在控制台中，你会看到类似以下的输出：

```
INFO: Started server process [1234]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

- Started server process [1234]：表示服务器进程已经启动，[1234] 是进程的 ID。
- Waiting for application startup.：服务器正在等待应用程序启动。
- Application startup complete.：应用程序已经成功启动。
  Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)：服务器已经开始监听 0.0.0.0:8000 地址，提示可以通过该地址访问应用程序，按 CTRL+C 可以停止服务器。