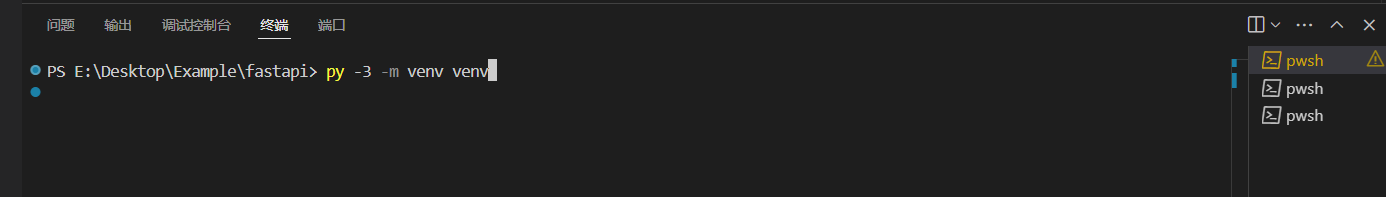
fastapi

vscode环境配置

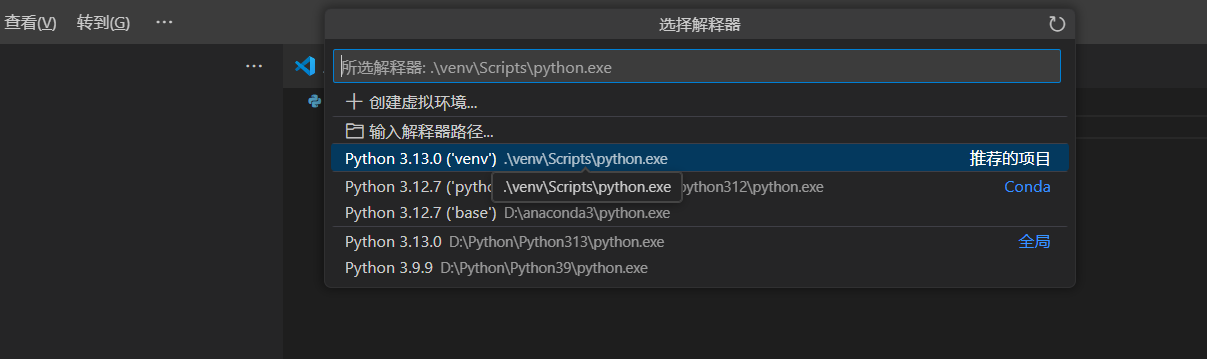
1. 创建虚拟环境

输入 （虚拟环境名字）

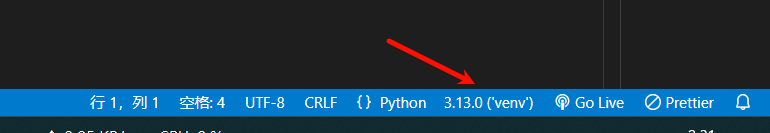


2. 选择默认解释器

点击：查看-》命令面板-》输入python select 选择上面创建的虚拟环境



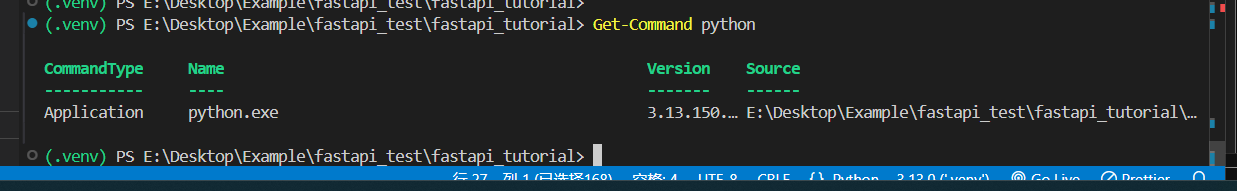
3. 注意当前的虚拟环境



4. 激活虚拟环境

在shell中输入：.venv\Scripts\Activate.ps1

5. 检查虚拟环境是否激活

在shell中输入：Get-Command python  


6. 停用虚拟环境

在shell中输入：deactivate

Starlette，Pydantic与FastAPI的关系

FastAPI 是直接从 Starlette 继承的类。

你可以通过 FastAPI 使用所有的 Starlette 的功能。

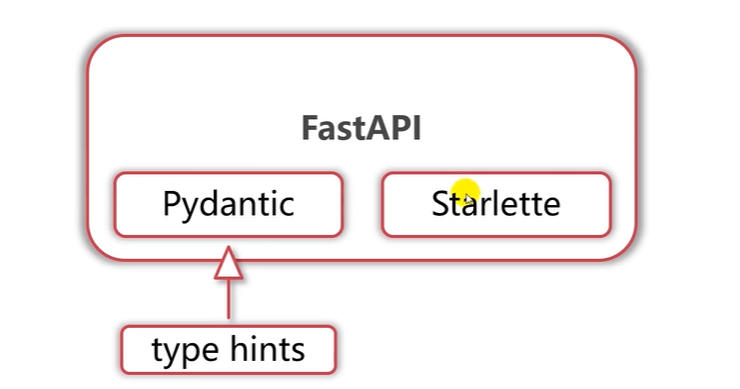
Python 的类型提示 type hints

Pydantic是一个基于Python类型提示来定义数据验证，序列化和

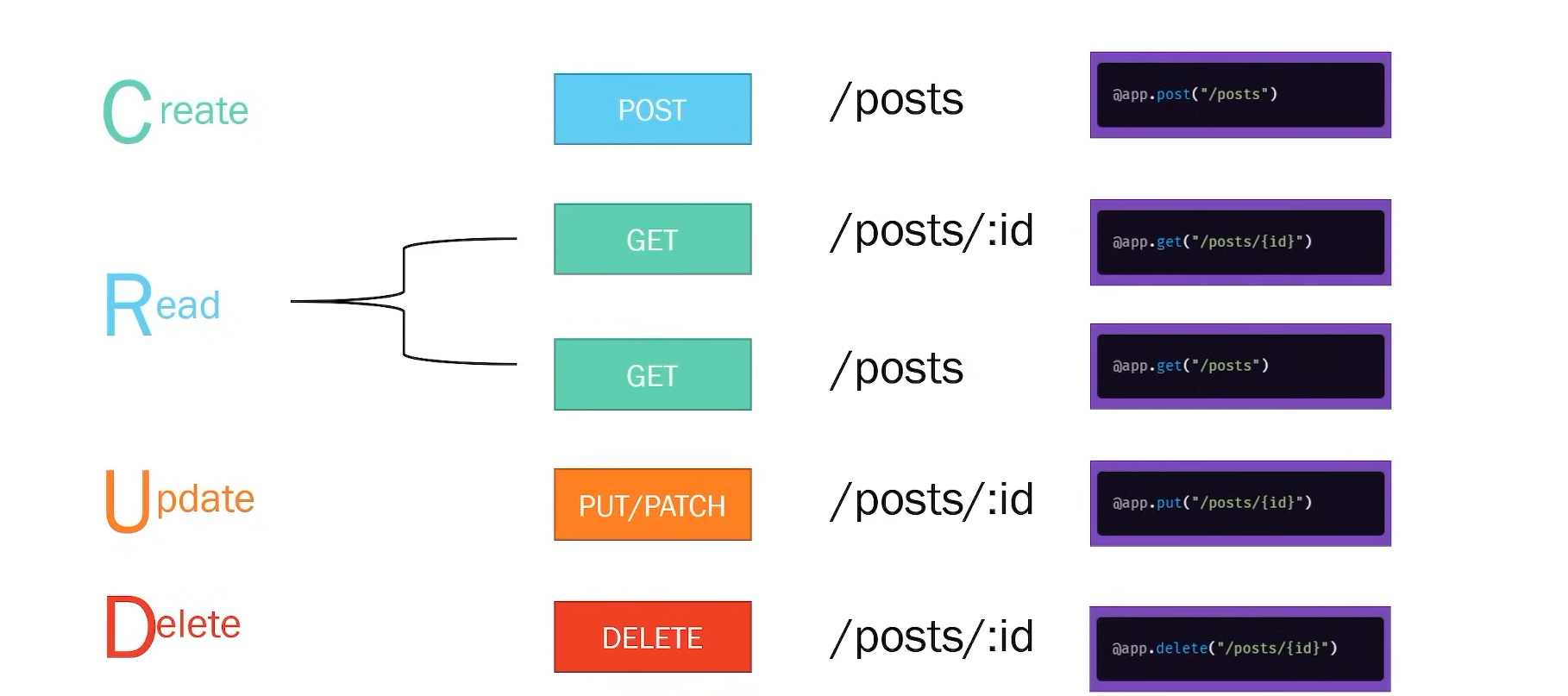
文档（使用JSON模式）库

Starlette是一种轻量级的ASGI框架/工具包，是构建高性能

Asyncio服务的理想选择







第一步

创建一个简单的API

创建main.py文件

from fastapi import FastAPI #导入 FastAPI

app = FastAPI() #创建fastapi的一个实例app

@app.get("/")

async def root():

return {"message": "Hello World"}

启动服务

在终端中输入：uvicorn main:app --reload

main代表创建的main.py名，app代表创建的app实例，--reload 代表修改代码后不必停止重新启动

路径（端点、路由）定义

这里的「路径」指的是 URL 中从第一个 / 起的后半部分。

所以，在一个这样的 URL 中：

https://example.com/items/foo

...路径会是：

/items/foo

操作（方法）定义

这里的「操作」指的是一种 HTTP「方法」。

下列之一：

* POST
* GET
* PUT
* DELETE

...以及更少见的几种：

* OPTIONS
* HEAD
* PATCH
* TRACE

在 HTTP 协议中，你可以使用以上的其中一种（或多种）「方法」与每个路径进行通信。

在开发 API 时，你通常使用特定的 HTTP 方法去执行特定的行为。

通常使用：

POST：创建数据。

GET：读取数据。

PUT：更新数据。

DELETE：删除数据。

因此，在 OpenAPI 中，每一个 HTTP 方法都被称为「操作」。

我们也打算称呼它们为「操作」。

创建一个路径操作

@app.get("/") #路径操作装饰器，app.get("/")路径操作

async def root():

return {"message": "Hello World"}

@app.get("/") 告诉 **FastAPI** 在它下方的函数负责处理如下访问请求：

* 请求路径为 /
* 使用 get 操作

你也可以使用其他的操作：

* @app.post()
* @app.put()
* @app.delete()

以及更少见的：

* @app.options()
* @app.head()
* @app.patch()
* @app.trace()

路径操作函数定义

「**路径操作函数**」：

* **路径**：是 /。
* **操作**：是 get。
* **函数**：是位于「装饰器」下方的函数（位于 @app.get("/") 下方）。

@app.get("/")

async def root(): #路径操作函数

return {"message": "Hello World"}

返回内容

@app.get("/")

async def root():

return {"message": "Hello World"}

你可以返回一个 dict、list，像 str、int 一样的单个值，等等。

总结

* 导入 FastAPI。
* 创建一个 app 实例。
* 编写一个**路径操作装饰器**（如 @app.get("/")）。
* 编写一个**路径操作函数**（如上面的 def root(): ...）。
* 运行开发服务器（如 uvicorn main:app --reload）。

路径参数

说明

https://fastapi.tiangolo.com/zh/tutorial/path-params/id?q=1&a=2&c=3

最后一个‘/’和‘？’中间的值称为路径参数（变量），没有‘？’全部是路径参数

FastAPI 支持使用 Python 字符串格式化语法声明**路径参数**（**变量**）

@app.get("/items/{item\_id}")

async def read\_item(item\_id):

return {"item\_id": item\_id}

这段代码把路径参数 item\_id 的值传递给路径函数的参数 item\_id。

运行示例并访问 <http://127.0.0.1:8000/items/foo>，可获得如下响应：

{"item\_id":"foo"}

声明路径参数的类型

@app.get("/items/{item\_id}")

async def read\_item(item\_id: int):

return {"item\_id": item\_id}

本例把 item\_id 的类型声明为 int。

类型声明将为函数提供错误检查、代码补全等编辑器支持。

数据转换

运行示例并访问 <http://127.0.0.1:8000/items/3>，返回的响应如下：

{"item\_id":3}

注意，函数接收并返回的值是 3（ int），不是 "3"（str）。

**FastAPI** 通过类型声明自动**解析**请求中的数据。

数据校验

**FastAPI** 使用 Python 类型声明实现了数据校验。

通过浏览器访问 <http://127.0.0.1:8000/items/foo>，接收如下 HTTP 错误信息：

{

"detail": [

{

"loc": [ #位置

"path", #代表路径参数

"item\_id"

],

"msg": "value is not a valid integer",

"type": "type\_error.integer"

}

]

}

这是因为路径参数 item\_id 的值 （"foo"）的类型不是 int。

值的类型不是 int 而是浮点数（float）时也会显示同样的错误，比如： [http://127.0.0.1:8000/items/4.2。](http://127.0.0.1:8000/items/4.2)

查看文档

访问 <http://127.0.0.1:8000/docs>，查看自动生成的 API 文档：

**FastAPI** 还内置了 ReDoc 生成的备选 API 文档，可在此查看 <http://127.0.0.1:8000/redoc>：

顺序很重要

有时，*路径操作*中的路径是写死的。

比如要使用 /users/me 获取当前用户的数据。

然后还要使用 /users/{user\_id}，通过用户 ID 获取指定用户的数据。

由于*路径操作*是按顺序依次运行的，因此，一定要在 /users/{user\_id} 之前声明 /users/me ：

[Python 3.8+](https://fastapi.tiangolo.com/zh/tutorial/path-params/#__tabbed_3_1)

from fastapi import FastAPI

app = FastAPI()

@app.get("/users/me")

async def read\_user\_me():

return {"user\_id": "the current user"}

@app.get("/users/{user\_id}")

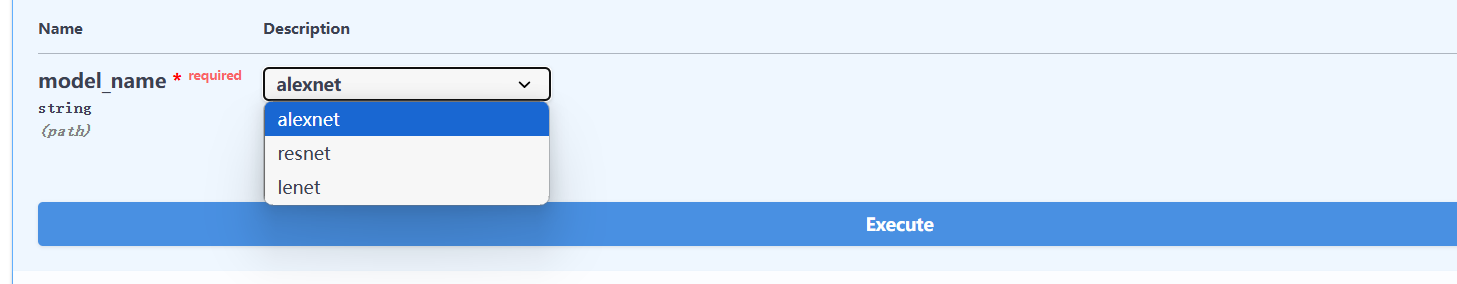
async def read\_user(user\_id: str):

return {"user\_id": user\_id}

否则，/users/{user\_id} 将匹配 /users/me，FastAPI 会**认为**正在接收值为 "me" 的 user\_id 参数。

预设值

路径操作使用 Python 的 Enum 类型接收预设的*路径参数*



创建 Enum 类[¶](https://fastapi.tiangolo.com/zh/tutorial/path-params/#enum)

导入 Enum 并创建继承自 str 和 Enum 的子类。

通过从 str 继承，API 文档就能把值的类型定义为**字符串**，并且能正确渲染。

然后，创建包含固定值的类属性，这些固定值是可用的有效值：

[Python 3.8+](https://fastapi.tiangolo.com/zh/tutorial/path-params/#__tabbed_4_1)

from enum import Enum

from fastapi import FastAPI

class ModelName(str, Enum):

alexnet = "alexnet"

resnet = "resnet"

lenet = "lenet"

app = FastAPI()

@app.get("/models/{model\_name}")

async def get\_model(model\_name: ModelName):

if model\_name is ModelName.alexnet:

return {"model\_name": model\_name, "message": "Deep Learning FTW!"}

if model\_name.value == "lenet":

return {"model\_name": model\_name, "message": "LeCNN all the images"}

return {"model\_name": model\_name, "message": "Have some residuals"}

包含路径的路径参数

假设*路径操作*的路径为 /files/{file\_path}。

但需要 file\_path 中也包含*路径*，比如，home/johndoe/myfile.txt。

此时，该文件的 URL 是这样的：/files/home/johndoe/myfile.txt。

OpenAPI 不支持声明包含路径的*路径参数*，因为这会导致测试和定义更加困难。

不过，仍可使用 Starlette 内置工具在 **FastAPI** 中实现这一功能。

直接使用 Starlette 的选项声明包含*路径*的*路径参数*：

/files/{file\_path:path}

本例中，参数名为 file\_path，结尾部分的 :path 说明该参数应匹配*路径*。

用法如下：

[Python 3.8+](https://fastapi.tiangolo.com/zh/tutorial/path-params/#__tabbed_9_1)

from fastapi import FastAPI

app = FastAPI()

@app.get("/files/{file\_path:path}")

async def read\_file(file\_path: str):

return {"file\_path": file\_path}

小结

通过简短、直观的 Python 标准类型声明，**FastAPI** 可以获得：

* 编辑器支持：错误检查，代码自动补全等
* 数据**解析**
* 数据校验
* API 注解和 API 文档

只需要声明一次即可。

这可能是除了性能以外，**FastAPI** 与其它框架相比的主要优势。

查询参数

说明

https://fastapi.tiangolo.com/zh/tutorial/path-params/id?q=1&a=2&c=3

查询字符串是键值对的集合，这些键值对位于 URL 的 ? 之后，以 & 分隔。

声明的参数不是路径参数时，路径操作函数会把该参数自动解释为**查询**参数。

from fastapi import FastAPI

app = FastAPI()

fake\_items\_db = [{"item\_name": "Foo"}, {"item\_name": "Bar"}, {"item\_name": "Baz"}]

@app.get("/items/")

async def read\_item(skip: int = 0, limit: int = 10): #设置了默认值

return fake\_items\_db[skip : skip + limit]

例如，以下 URL 中：

http://127.0.0.1:8000/items/?skip=0&limit=10

……查询参数为：

* skip：值为 0
* limit：值为 10

这些值都是 URL 的组成部分，因此，它们的类型**本应**是字符串。

但声明 Python 类型（上例中为 int）之后，这些值就会转换为声明的类型，并进行类型校验。

默认值

查询参数不是路径的固定内容，它是可选的，还支持默认值。

如果访问：

http://127.0.0.1:8000/items/?skip=20

查询参数的值就是：

* skip=20：在 URL 中设定的值
* limit=10：使用默认值

可选参数

同理，把默认值设为 None 即可声明**可选的**查询参数：

也可以使用Optional[str] 可以帮助编辑器发现代码中的错误。

@app.get("/items/{item\_id}")

async def read\_item(item\_id: str, q: str | None = None):

if q:

return {"item\_id": item\_id, "q": q}

return {"item\_id": item\_id}

查询参数类型转换

@app.get("/items/{item\_id}")

async def read\_item(item\_id: str, q: str | None = None, short: bool = False):

item = {"item\_id": item\_id}

if q:

item.update({"q": q})

if not short:

item.update(

{"description": "This is an amazing item that has a long description"}

)

return item

本例中，访问：

http://127.0.0.1:8000/items/foo?short=1

short=1 short=on short=True short=true short=yes

或其它任意大小写形式（大写、首字母大写等），函数接收的 short 参数都是布尔值 True。值为 False 时也一样。

多个路径和查询参数

**FastAPI** 可以识别同时声明的多个路径参数和查询参数。

而且声明查询参数的顺序并不重要。

FastAPI 通过参数名进行检测：

@app.get("/users/{user\_id}/items/{item\_id}")

async def read\_user\_item(

user\_id: int, item\_id: str, q: str | None = None, short: bool = False

):

item = {"item\_id": item\_id, "owner\_id": user\_id}

if q:

item.update({"q": q})

if not short:

item.update(

{"description": "This is an amazing item that has a long description"}

)

return item

必选查询参数

如果只想把参数设为**可选**，但又不想指定参数的值，则要把默认值设为 None。

如果要把查询参数设置为**必选**，就不要声明默认值：

请求体

说明

请求体是html的body里的数据。

发送数据使用 POST（最常用）、PUT、DELETE、PATCH 等操作。

规范中没有定义使用 GET 发送请求体的操作，但不管怎样，FastAPI 也支持这种方式，只不过仅用于非常复杂或极端的用例。

我们不建议使用 GET，因此，在 Swagger UI 交互文档中不会显示有关 GET 的内容，而且代理协议也不一定支持 GET。

fastapi会把继承了Pydantic 的 BaseModel类的类当初请体规范，并把值传入该对象。

声明请求体参数

from fastapi import FastAPI

from pydantic import BaseModel

class Item(BaseModel): #创建请求体的类，并定义格式

name: str

description: str | None = None

price: float

tax: float | None = None

app = FastAPI()

@app.post("/items/")

async def create\_item(item: Item): #……此处，请求体参数的类型为 Item 模型

return item

使用模型，.dict（）提取请求体内容

@app.post("/items/")

async def create\_item(item: Item):

item\_dict = item.dict() #用.dict（）方法以字典形式返回请求体内容

if item.tax:

price\_with\_tax = item.price + item.tax

item\_dict.update({"price\_with\_tax": price\_with\_tax})

return item\_dict

请求体 + 路径参数 + 查询参数

**FastAPI** 支持同时声明**请求体**、**路径参数**和**查询参数**。

**FastAPI** 能够正确识别这三种参数，并从正确的位置获取数据。

@app.put("/items/{item\_id}")

async def update\_item(item\_id: int, item: Item, q: str | None = None):

result = {"item\_id": item\_id, \*\*item.dict()}

if q:

result.update({"q": q})

return result

函数参数识别规则

函数参数按如下规则进行识别：

* **路径**中声明了相同参数的参数，是路径参数
* 类型是（int、float、str、bool 等）**单类型**的参数，是**查询**参数
* 类型是 **Pydantic 模型**的参数，是**请求体**

查询参数和字符串校验

普通校验

@app.get("/items/")

async def read\_items(q: str | None = None):

results = {"items": [{"item\_id": "Foo"}, {"item\_id": "Bar"}]}

if q:

results.update({"q": q})

return results

查询参数 q 的类型为 str，默认值为 None，因此它是可选的。

Query校验

导入 Query

from typing import Union

from fastapi import FastAPI, Query #导入 Query

app = FastAPI()

@app.get("/items/")

async def read\_items(q: Union[str, None] = Query(default=None, max\_length=50)):

results = {"items": [{"item\_id": "Foo"}, {"item\_id": "Bar"}]}

if q:

results.update({"q": q})

return results

校验规则

@app.get("/items/")

async def read\_items(

q: Union[str, None] = Query(

default=None, min\_length=3, max\_length=50, pattern="^fixedquery$"

),

):

results = {"items": [{"item\_id": "Foo"}, {"item\_id": "Bar"}]}

if q:

results.update({"q": q})

return results

* default=None, #默认值为None

default选项填写或者等于...表示该参数为必须参数

* min\_length=3, #最小长度
* max\_length=50, #最大长度
* pattern="^fixedquery$" #正则匹配

查询参数列表 / 多个值

当你使用 Query 显式地定义查询参数时，你还可以声明它去接收一组值，或换句话来说，接收多个值用列表List[str]。

from typing import List, Union

from fastapi import FastAPI, Query

app = FastAPI()

@app.get("/items/")

async def read\_items(q: Union[List[str], None] = Query(default=None)):

query\_items = {"q": q}

return query\_items

然后，输入如下网址：

<http://localhost:8000/items/?q=foo&q=bar>

你会在*路径操作函数*的*函数参数* q 中以一个 Python list 的形式接收到*查询参数* q 的多个值（foo 和 bar）。

因此，该 URL 的响应将会是：

{

"q": [

"foo",

"bar"

]

}

你也可以直接使用 list 代替 List [str]：

async def read\_items(q: list = Query(default=[])):

更多选项

@app.get("/items/")

async def read\_items(

q: Union[str, None] = Query(

default=None,

title="Query string", #q参数标题

description="Query string for the items to search in the database that have a good match", #q参数描述

min\_length=3,

alias="item-query" #q参数别名

),

):

results = {"items": [{"item\_id": "Foo"}, {"item\_id": "Bar"}]}

if q:

results.update({"q": q})

return results

弃用参数

@app.get("/items/")

async def read\_items(

q: Union[str, None] = Query(

default=None,

alias="item-query",

title="Query string",

description="Query string for the items to search in the database that have a good match",

min\_length=3,

max\_length=50,

pattern="^fixedquery$",

deprecated=True, #弃用q参数

),

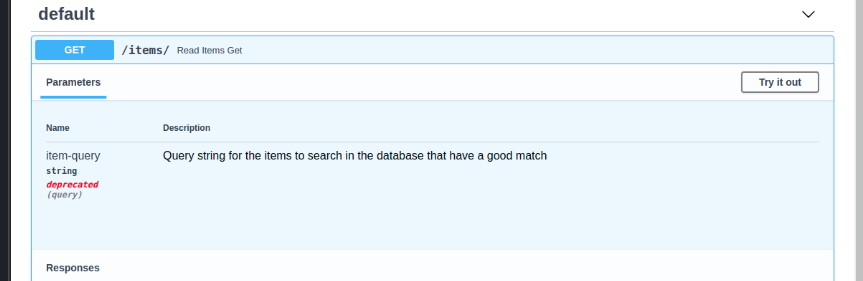
):

results = {"items": [{"item\_id": "Foo"}, {"item\_id": "Bar"}]}

if q:

results.update({"q": q})

return results



为查询参数声明额外的校验和元数据

通用的校验和元数据：

* alias
* title
* description
* deprecated

特定于字符串的校验：

* min\_length
* max\_length
* regex

路径参数和数值校验

普通校验

和查询参数一样

Path校验

导入 Path

from fastapi import FastAPI, Path, Query

声明元数据

你可以声明与 Query 相同的所有参数。

例如，要声明路径参数 item\_id的 title 元数据值，你可以输入：

from typing import Annotated

from fastapi import FastAPI, Path, Query

app = FastAPI()

@app.get("/items/{item\_id}")

async def read\_items(

item\_id: Annotated[int, Path(title="The ID of the item to get")],

q: Annotated[str | None, Query(alias="item-query")] = None,

):

results = {"item\_id": item\_id}

if q:

results.update({"q": q})

return results

对参数排序

如果你将带有「默认值」的参数放在没有「默认值」的参数之前，Python 将会报错。

解决办法：

1. 把不带默认值的参数排在带默认值参数的前面

2. 使用Qury和Path显示声明参数

3. 传递 \* 作为函数的第一个参数。

Python 不会对该 \* 做任何事情，但是它将知道之后的所有参数都应作为关键字参数（键值对），也被称为 kwargs，来调用。即使它们没有默认值。

async def read\_items(\*, item\_id: int = Path(title="The ID of the item to get"), q: str):

数值校验

async def read\_items(

\*, item\_id: int = Path(title="The ID of the item to get", ge=1), q: str

):

* gt：大于（greater than）
* ge：大于等于（greater than or equal）
* lt：小于（less than）
* le：小于等于（less than or equal）

总结

当你从 fastapi 导入 Query、Path 和其他同类对象时，它们实际上是函数。

当被调用时，它们返回同名类的实例。

如此，你导入 Query 这个函数。当你调用它时，它将返回一个同样命名为 Query 的类的实例。

因为使用了这些函数（而不是直接使用类），所以你的编辑器不会标记有关其类型的错误。

查询参数模型

定义查询参数模型

如果你有一组具有相关性的**查询参数**，你可以创建一个 **Pydantic 模型**来声明它们。

FastAPI 从 0.115.0 版本开始支持这个特性。

from typing import Annotated, Literal

from fastapi import FastAPI, Query

from pydantic import BaseModel, Field

app = FastAPI()

class FilterParams(BaseModel): #定义查询模型

limit: int = Field(100, gt=0, le=100)

offset: int = Field(0, ge=0)

order\_by: Literal["created\_at", "updated\_at"] = "created\_at"

tags: list[str] = []

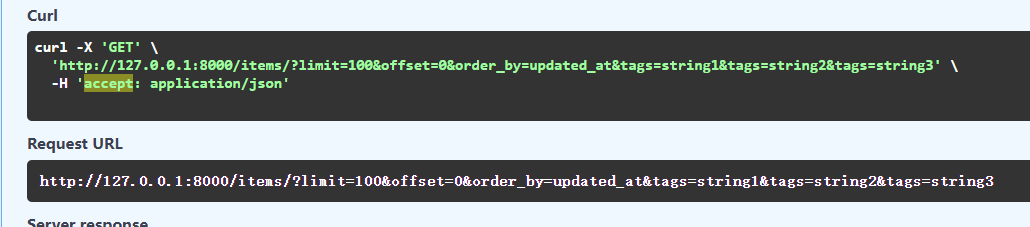
@app.get("/items/")

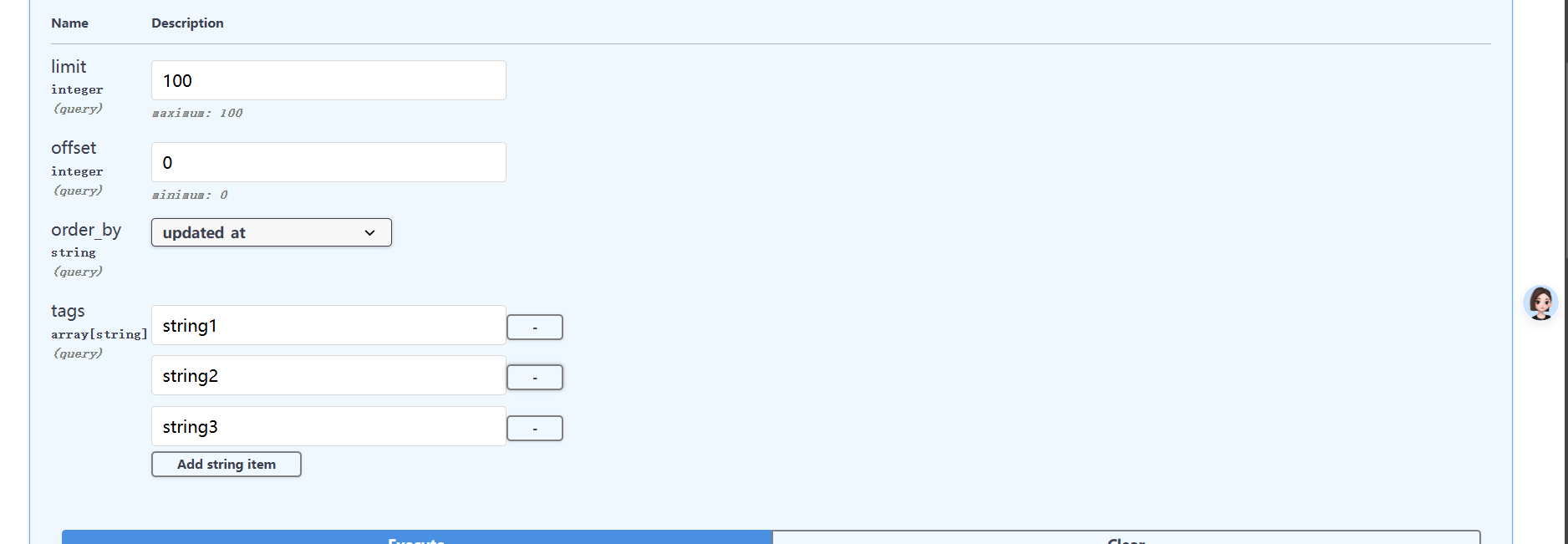
async def read\_items(filter\_query: Annotated[FilterParams, Query()]): #查询参数只能是FilterParams查询模型

return filter\_query

**FastAPI** 将会从请求的**查询参数**中**提取**出**每个字段**的数据，并将其提供给你定义的 Pydantic 模型。

输入的url：





禁止额外的查询参数

在一些特殊的使用场景中（可能不是很常见），你可能希望**限制**你要接收的查询参数。

你可以使用 Pydantic 的模型配置来 forbid（意为禁止 —— 译者注）任何 extra（意为额外的 —— 译者注）字段：

from typing import Annotated, Literal

from fastapi import FastAPI, Query

from pydantic import BaseModel, Field

app = FastAPI()

class FilterParams(BaseModel): #定义查询参数模型

model\_config = {"extra": "forbid"} #限制额外的查询参数

limit: int = Field(100, gt=0, le=100)

offset: int = Field(0, ge=0)

order\_by: Literal["created\_at", "updated\_at"] = "created\_at"

tags: list[str] = []

@app.get("/items/")

async def read\_items(filter\_query: Annotated[FilterParams, Query()]):

return filter\_query

假设有一个客户端尝试在**查询参数**中发送一些**额外的**数据，它将会收到一个**错误**响应。

例如，如果客户端尝试发送一个值为 plumbus 的 tool 查询参数，如：

<https://example.com/items/?limit=10&tool=plumbus>

他们将收到一个**错误**响应，告诉他们查询参数 tool 是不允许的：

{

"detail": [

{

"type": "extra\_forbidden",

"loc": ["query", "tool"],

"msg": "Extra inputs are not permitted",

"input": "plumbus"

}

]

}

请求体 - 多个参数

混合使用 Path、Query 和请求体参数

Annotated ：Annotated类型注解是 Python 3.9 及以上版本中引入的一个新特性，它可以为类型添加额外的元数据信息，就是可以添加Path或者Qury函数。

[Python 3.10+](https://fastapi.tiangolo.com/zh/tutorial/body-multiple-params/#__tabbed_1_1)

from typing import Annotated

from fastapi import FastAPI, Path

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):

name: str

description: str | None = None

price: float

tax: float | None = None

@app.put("/items/{item\_id}")

async def update\_item(

item\_id: Annotated[int, Path(title="The ID of the item to get", ge=0, le=1000)],

q: str | None = None,

item: Item | None = None,

):

results = {"item\_id": item\_id}

if q:

results.update({"q": q})

if item:

results.update({"item": item})

return results

多个请求体参数

[Python 3.10+](https://fastapi.tiangolo.com/zh/tutorial/body-multiple-params/#__tabbed_3_1)

from fastapi import FastAPI

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel): #声明请求体模型类

name: str

description: str | None = None

price: float

tax: float | None = None

class User(BaseModel): #声明请求体模型类

username: str

full\_name: str | None = None

@app.put("/items/{item\_id}")

async def update\_item(item\_id: int, item: Item, user: User): #多个请求体模型

results = {"item\_id": item\_id, "item": item, "user": user}

return results

在这种情况下，**FastAPI** 将注意到该函数中有多个请求体参数（两个 Pydantic 模型参数）。

因此，它将使用参数名称作为请求体中的键（字段名称），并期望一个类似于以下内容的请求体：

{

"item": {

"name": "Foo",

"description": "The pretender",

"price": 42.0,

"tax": 3.2

},

"user": {

"username": "dave",

"full\_name": "Dave Grohl"

}

}

请求体中的单一值

Body 同样具有与 Query、Path 以及其他后面将看到的类完全相同的额外校验和元数据参数。

用body当类型注解，fastapi会把该参数认为是html里的body里的单一键值对

from typing import Annotated

from fastapi import Body, FastAPI #导入body

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):

name: str

description: str | None = None

price: float

tax: float | None = None

class User(BaseModel):

username: str

full\_name: str | None = None

@app.put("/items/{item\_id}")

async def update\_item(

item\_id: int, item: Item, user: User, importance: Annotated[int, Body()] #引入body（）代表是请求体内的参数

):

results = {"item\_id": item\_id, "item": item, "user": user, "importance": importance}

return results

在这种情况下，**FastAPI** 将期望像这样的请求体：

{

"item": {

"name": "Foo",

"description": "The pretender",

"price": 42.0,

"tax": 3.2

},

"user": {

"username": "dave",

"full\_name": "Dave Grohl"

},

"importance": 5

}

嵌入单个请求体参数

假设你只有一个来自 Pydantic 模型 Item 的请求体参数 item。

默认情况下，**FastAPI** 将直接期望这样的请求体。

但是，如果你希望它期望一个拥有 item 键并在值中包含模型内容的 JSON，就像在声明额外的请求体参数时所做的那样，则可以使用一个特殊的 Body 参数 embed：

item: Item = Body(embed=True)

比如：

[Python 3.10+](https://fastapi.tiangolo.com/zh/tutorial/body-multiple-params/#__tabbed_9_1)

from typing import Annotated

from fastapi import Body, FastAPI

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):

name: str

description: str | None = None

price: float

tax: float | None = None

@app.put("/items/{item\_id}")

async def update\_item(item\_id: int, item: Annotated[Item, Body(embed=True)]):

results = {"item\_id": item\_id, "item": item}

return results

🤓 Other versions and variants

在这种情况下，**FastAPI** 将期望像这样的请求体：

{

"item": {

"name": "Foo",

"description": "The pretender",

"price": 42.0,

"tax": 3.2

}

}

而不是：

{

"name": "Foo",

"description": "The pretender",

"price": 42.0,

"tax": 3.2

}

请求体 - 字段

Query、Path 、Body 、Field

与在*路径操作函数*中使用 Query、Path 、Body 声明校验与元数据的方式一样，可以使用 Pydantic 的 Field 在 Pydantic 模型内部声明校验和元数据。

Field 的工作方式和 Query、Path、Body 相同，参数也相同。

实际上，Query、Path 都是 Params 的子类，而 Params 类又是 Pydantic 中 FieldInfo 的子类。

Pydantic 的 Field 返回也是 FieldInfo 的类实例。

Body 直接返回的也是 FieldInfo 的子类的对象。后文还会介绍一些 Body 的子类。

注意，从 fastapi 导入的 Query、Path 等对象实际上都是返回特殊类的函数。

导入 Field

from pydantic import BaseModel, Field

声明模型属性

from typing import Annotated

from fastapi import Body, FastAPI

from pydantic import BaseModel, Field

app = FastAPI()

class Item(BaseModel):

name: str

description: str | None = Field(

default=None, title="The description of the item", max\_length=300

)

price: float = Field(gt=0, description="The price must be greater than zero")

tax: float | None = None

@app.put("/items/{item\_id}")

async def update\_item(item\_id: int, item: Annotated[Item, Body(embed=True)]):

results = {"item\_id": item\_id, "item": item}

return results

Annotated[Item, Body(embed=True)]

表示期望请求体是这样的

键是"item"，值是Item类

{

"item": {

"name": "Foo",

"description": "The pretender",

"price": 42.0,

"tax": 3.2

}

}

小结

Pydantic 的 Field 可以为模型属性声明更多校验和元数据。

传递 JSON Schema 元数据还可以使用更多关键字参数。

请求体 - 嵌套模型

嵌套模型

from fastapi import FastAPI

from pydantic import BaseModel

app = FastAPI()

class Image(BaseModel): #定义子类

url: str

name: str

class Item(BaseModel):

name: str

description: str | None = None

price: float

tax: float | None = None

tags: set[str] = set()

image: Image | None = None #嵌套子类

@app.put("/items/{item\_id}")

async def update\_item(item\_id: int, item: Item):

results = {"item\_id": item\_id, "item": item}

return results

这意味着 **FastAPI** 将期望类似于以下内容的请求体：

{

"name": "Foo",

"description": "The pretender",

"price": 42.0,

"tax": 3.2,

"tags": ["rock", "metal", "bar"],

"image": {

"url": "http://example.com/baz.jpg",

"name": "The Foo live"

}

}

特殊的类型和校验

除了普通的单一值类型（如 str、int、float 等）外，你还可以使用从 str 继承的更复杂的单一值类型。

要了解所有的可用选项，请查看关于 [来自 Pydantic 的外部类型](https://docs.pydantic.dev/latest/concepts/types/) 的文档。你将在下一章节中看到一些示例。

例如，在 Image 模型中我们有一个 url 字段，我们可以把它声明为 Pydantic 的 HttpUrl，而不是 str：

from pydantic import BaseModel, HttpUrl

class Image(BaseModel):

url: HttpUrl #该字符串将被检查是否为有效的 URL，并在 JSON Schema / OpenAPI 文档中进行记录。

name: str

带有一组子模型的属性

class Image(BaseModel):

url: HttpUrl

name: str

class Item(BaseModel):

name: str

description: str | None = None

price: float

tax: float | None = None

tags: set[str] = set()

images: list[Image] | None = None #用列表的形式

深度嵌套模型

class Image(BaseModel):

url: HttpUrl

name: str

class Item(BaseModel):

name: str

description: str | None = None

price: float

tax: float | None = None

tags: set[str] = set()

images: list[Image] | None = None

class Offer(BaseModel):

name: str

description: str | None = None

price: float

items: list[Item]

纯列表请求体

如果你期望的 JSON 请求体的最外层是一个 JSON array（即 Python list），则可以在路径操作函数的参数中声明此类型，就像声明 Pydantic 模型一样：

images: List[Image]

@app.post("/images/multiple/")

async def create\_multiple\_images(images: list[Image]):

return images

模式的额外信息 - 例子

Pydantic schema\_extra

您可以使用 Config 和 schema\_extra 为Pydantic模型声明一个示例，如[Pydantic 文档：定制 Schema](https://docs.pydantic.dev/latest/concepts/json_schema/#schema-customization)中所述: 类似于注释

from fastapi import FastAPI

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):

name: str

description: str | None = None

price: float

tax: float | None = None

#类似于注释

model\_config = {

"json\_schema\_extra": {

"examples": [

{

"name": "Foo",

"description": "A very nice Item",

"price": 35.4,

"tax": 3.2,

}

]

}

}

@app.put("/items/{item\_id}")

async def update\_item(item\_id: int, item: Item):

results = {"item\_id": item\_id, "item": item}

return results

Field 的附加参数

类似于注释，给程序员看的

请记住，传递的那些额外参数不会添加任何验证，只会添加注释，用于文档的目的。

class Item(BaseModel):

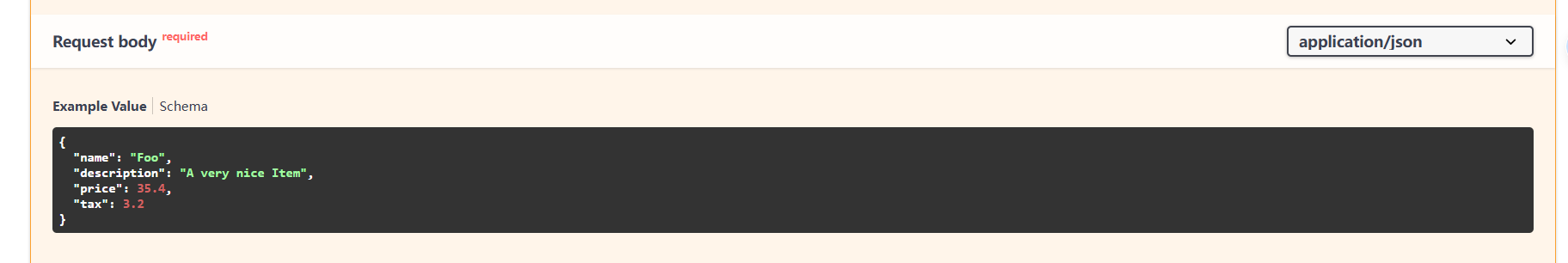
name: str = Field(examples=["Foo"])

description: str | None = Field(default=None, examples=["A very nice Item"])

price: float = Field(examples=[35.4])

tax: float | None = Field(default=None, examples=[3.2])

Body 额外参数



from typing import Annotated

from fastapi import Body, FastAPI

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):

name: str

description: str | None = None

price: float

tax: float | None = None

@app.put("/items/{item\_id}")

async def update\_item(

item\_id: int,

item: Annotated[

Item,

Body(

examples=[

{

"name": "Foo",

"description": "A very nice Item",

"price": 35.4,

"tax": 3.2,

}

],

),

],

):

results = {"item\_id": item\_id, "item": item}

return results

额外的数据类型

下面是一些你可以使用的其他数据类型:

* UUID:
  + 一种标准的 "通用唯一标识符" ，在许多数据库和系统中用作ID。
  + 在请求和响应中将以 str 表示。
* datetime.datetime:
  + 一个 Python datetime.datetime.
  + 在请求和响应中将表示为 ISO 8601 格式的 str ，比如: 2008-09-15T15:53:00+05:00.
* datetime.date:
  + Python datetime.date.
  + 在请求和响应中将表示为 ISO 8601 格式的 str ，比如: 2008-09-15.
* datetime.time:
  + 一个 Python datetime.time.
  + 在请求和响应中将表示为 ISO 8601 格式的 str ，比如: 14:23:55.003.
* datetime.timedelta:
  + 一个 Python datetime.timedelta.
  + 在请求和响应中将表示为 float 代表总秒数。
  + Pydantic 也允许将其表示为 "ISO 8601 时间差异编码", [查看文档了解更多信息](https://docs.pydantic.dev/latest/concepts/serialization/#json_encoders)。
* frozenset:
  + 在请求和响应中，作为 set 对待：
    - 在请求中，列表将被读取，消除重复，并将其转换为一个 set。
    - 在响应中 set 将被转换为 list 。
    - 产生的模式将指定那些 set 的值是唯一的 (使用 JSON 模式的 uniqueItems)。
* bytes:
  + 标准的 Python bytes。
  + 在请求和响应中被当作 str 处理。
  + 生成的模式将指定这个 str 是 binary "格式"。
* Decimal:
  + 标准的 Python Decimal。
  + 在请求和响应中被当做 float 一样处理。
* 您可以在这里检查所有有效的pydantic数据类型: [Pydantic data types](https://docs.pydantic.dev/latest/concepts/types/).

例子：

from datetime import datetime, time, timedelta

from typing import Annotated

from uuid import UUID

from fastapi import Body, FastAPI

app = FastAPI()

@app.put("/items/{item\_id}")

async def read\_items(

item\_id: UUID,

start\_datetime: Annotated[datetime, Body()],

end\_datetime: Annotated[datetime, Body()],

process\_after: Annotated[timedelta, Body()],

repeat\_at: Annotated[time | None, Body()] = None,

):

start\_process = start\_datetime + process\_after

duration = end\_datetime - start\_process

return {

"item\_id": item\_id,

"start\_datetime": start\_datetime,

"end\_datetime": end\_datetime,

"process\_after": process\_after,

"repeat\_at": repeat\_at,

"start\_process": start\_process,

"duration": duration,

}

Cookie 参数

说明

定义 Cookie 参数与定义 Query 和 Path 参数一样。

Cookie 、Path 、Query 是**兄弟类**，都继承自共用的 Param 类。

注意，从 fastapi 导入的 Query、Path、Cookie 等对象，实际上是返回特殊类的函数。

必须使用 Cookie 声明 cookie 参数，否则该参数会被解释为查询参数。

导入 Cookie

from fastapi import Cookie, FastAPI

声明 Cookie 参数

声明 Cookie 参数的方式与声明 Query 和 Path 参数相同。

第一个值是默认值，还可以传递所有验证参数或注释参数：

from typing import Annotated

from fastapi import Cookie, FastAPI

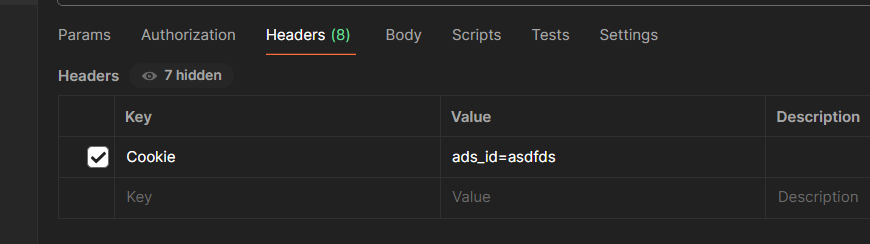
app = FastAPI()

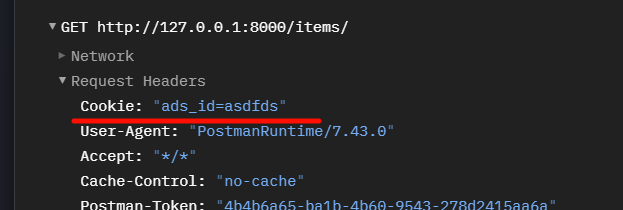
@app.get("/items/")

async def read\_items(ads\_id: Annotated[str | None, Cookie()] = None): #声明cookie参数

return {"ads\_id": ads\_id}

写Cookie参数的时候key：Cookie，value：ads\_id=xxx ，函数的ads\_id参数才能接收到值





带有 Pydantic 模型的 Cookie

如果您有一组相关的 **cookie**，您可以创建一个 **Pydantic 模型**来声明它们。🍪

这将允许您在**多个地方**能够**重用模型**，并且可以一次性声明所有参数的验证方式和元数据。

from typing import Annotated

from fastapi import Cookie, FastAPI #引入cookie

from pydantic import BaseModel

app = FastAPI()

class Cookies(BaseModel): #创建cookie类

session\_id: str

fatebook\_tracker: str | None = None

googall\_tracker: str | None = None

@app.get("/items/")

async def read\_items(cookies: Annotated[Cookies, Cookie()]): #声明cookie参数

return cookies

在postman的key里面输入

Cookie

在postman的value里面输入

session\_id=your\_session\_id\_value;fatebook\_tracker=your\_fatebook\_tracker\_value;googall\_tracker=your\_googall\_tracker\_value

返回结果

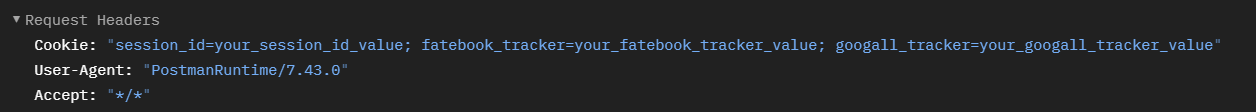
{

    "session\_id": "your\_session\_id\_value",

    "fatebook\_tracker": "your\_fatebook\_tracker\_value",

    "googall\_tracker": "your\_googall\_tracker\_value"

}



禁止额外的 Cookie

class Cookies(BaseModel):

model\_config = {"extra": "forbid"} #加入该参数可以禁止额外的cookie

session\_id: str

fatebook\_tracker: Union[str, None] = None

googall\_tracker: Union[str, None] = None

Header 参数

说明

定义 Header 参数的方式与定义 Query、Path、Cookie 参数相同。

不用担心变量中的下划线，**FastAPI** 可以自动转换。

导入 Header

from fastapi import FastAPI, Header

Header 是 Path、Query、Cookie 的**兄弟类**，都继承自共用的 Param 类。

注意，从 fastapi 导入的 Query、Path、Header 等对象，实际上是返回特殊类的函数。

必须使用 Header 声明 header 参数，否则该参数会被解释为查询参数。

声明 Header 参数

然后，使用和 Path、Query、Cookie 一样的结构定义 header 参数。

第一个值是默认值，还可以传递所有验证参数或注释参数：

from typing import Annotated

from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/items/")

async def read\_items(user\_agent: Annotated[str | None, Header()] = None): #声明header参数

return {"User-Agent": user\_agent}

自动转换

Header 比 Path、Query 和 Cookie 提供了更多功能。

大部分标准请求头用**连字符**分隔，即**减号**（-）。

但是 user-agent 这样的变量在 Python 中是无效的。

因此，默认情况下，Header 把参数名中的字符由下划线（\_）改为连字符（-）来提取并存档请求头 。

同时，HTTP 的请求头不区分大小写，可以使用 Python 标准样式（即 **snake\_case**）进行声明。

因此，可以像在 Python 代码中一样使用 user\_agent ，无需把首字母大写为 User\_Agent 等形式。

如需禁用下划线自动转换为连字符，可以把 Header 的 convert\_underscores 参数设置为 False：

[Python 3.10+](https://fastapi.tiangolo.com/zh/tutorial/header-params/#__tabbed_5_1)

from typing import Annotated

from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/items/")

async def read\_items(

strange\_header: Annotated[str | None, Header(convert\_underscores=False)] = None, #不会把strange\_header的下划线转换为‘-’

):

return {"strange\_header": strange\_header}

🤓 Other versions and variants

**警告**

注意，使用 convert\_underscores = False 要慎重，有些 HTTP 代理和服务器不支持使用带有下划线的请求头。

重复的请求头

有时，可能需要接收重复的请求头。即同一个请求头有多个值。

类型声明中可以使用 list 定义多个请求头。

使用 Python list 可以接收重复请求头所有的值。

例如，声明 X-Token 多次出现的请求头，可以写成这样：

@app.get("/items/")

async def read\_items(x\_token: Annotated[list[str] | None, Header()] = None):

return {"X-Token values": x\_token}

使用 Pydantic 模型的 Header 参数（可以定义多个header的kv）

如果您有一组相关的 **header 参数**，您可以创建一个 **Pydantic 模型**来声明它们。

这将允许您在**多个地方**能够**重用模型**，并且可以一次性声明所有参数的验证和元数据。😎

from typing import Annotated

from fastapi import FastAPI, Header

from pydantic import BaseModel

app = FastAPI()

class CommonHeaders(BaseModel): #声明一个header类

host: str

save\_data: bool

if\_modified\_since: str | None = None

traceparent: str | None = None

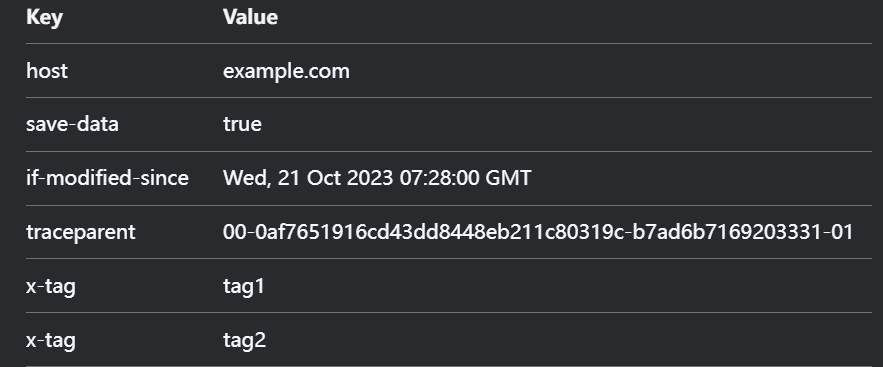
x\_tag: list[str] = []

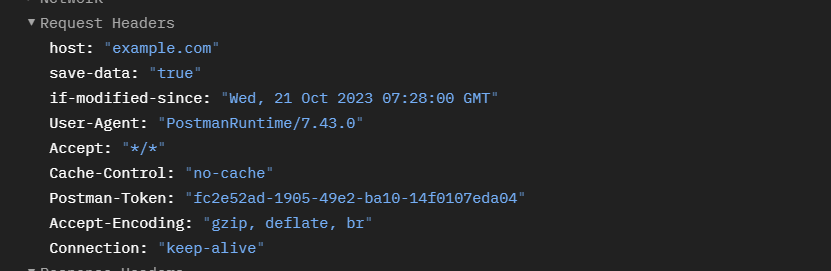
@app.get("/items/")

async def read\_items(headers: Annotated[CommonHeaders, Header()]): #声明header参数

return headers

在postman里面输入





禁止额外的 Headers

class CommonHeaders(BaseModel):

model\_config = {"extra": "forbid"} #加入该参数就能禁止额外的header参数

host: str

save\_data: bool

if\_modified\_since: str | None = None

traceparent: str | None = None

x\_tag: list[str] = []

响应模型

说明

设定响应的模型。

你可以在任意的*路径操作*中使用 response\_model 参数来声明用于响应的模型：

* @app.get()
* @app.post()
* @app.put()
* @app.delete()
* 等等。

response\_model设置相应模型

注意，response\_model是「装饰器」方法（get，post 等）的一个参数。不像之前的所有参数和请求体，它不属于*路径操作函数*。

FastAPI 将使用此 response\_model 来：

* 将输出数据转换为其声明的类型。
* 校验数据。
* 在 OpenAPI 的*路径操作*中为响应添加一个 JSON Schema。
* 并在自动生成文档系统中使用。

实例：

from typing import List, Union

from fastapi import FastAPI

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel): #设置相应模型的类

name: str

description: Union[str, None] = None

price: float

tax: float = 10.5

tags: List[str] = []

items = {

"foo": {"name": "Foo", "price": 50.2},

"bar": {"name": "Bar", "description": "The bartenders", "price": 62, "tax": 20.2},

"baz": {"name": "Baz", "description": None, "price": 50.2, "tax": 10.5, "tags": []},

}

@app.get("/items/{item\_id}", response\_model=Item, response\_model\_exclude\_unset=True) # response\_model=Item，设置相应模型是Item，response\_model\_exclude\_unset=True：排除未赋值的字段

async def read\_item(item\_id: str):

return items[item\_id]

相应编码的参数

如果include和exclude的参数相同，以exclude排除的参数为准。

response\_model\_include

class Item(BaseModel):

name: str

description: Union[str, None] = None

price: float

tax: float = 10.5

@app.get("/items/{item\_id}/public", response\_model=Item, response\_model\_inlude={"tax",”price”})

输出只包含tax和price字段

response\_model\_exclude

class Item(BaseModel):

name: str

description: Union[str, None] = None

price: float

tax: float = 10.5

@app.get("/items/{item\_id}/public", response\_model=Item, response\_model\_inlude={"tax",”price”})

输出排除tax和price字段

* response\_model\_exclude\_unset=True #排除未赋值的字段
* response\_model\_exclude\_defaults=True #排除有默认值的字段
* response\_model\_exclude\_none=True #排除None的字段

相应模型实例

Union相应模型

class CarItem(BaseItem): #创建CarItem类

type: str = "car"

class PlaneItem(BaseItem): #创建PlaneItem类

type: str = "plane"

size: int

@app.get("/items/{item\_id}", response\_model=Union[PlaneItem, CarItem]) #响应模型只能是PlaneItem,或者CarItem

async def read\_item(item\_id: str):

return items[item\_id]

list列表模型

class Item(BaseModel): #创建Item类

name: str

description: str

@app.get("/items/", response\_model=list[Item]) #响应模型是list列表，列表元素是Item类对象

async def read\_items():

return items

任意 dict 构成的响应

@app.get("/keyword-weights/", response\_model=dict[str, float]) #响应模型是dict

async def read\_keyword\_weights():

return {"foo": 2.3, "bar": 3.4}

响应状态码

说明

在装饰内的设置的状态码表明成功响应的状态码，

与指定响应模型的方式相同，在以下任意*路径操作*中，可以使用 status\_code 参数声明用于响应的 HTTP 状态码：

* @app.get()
* @app.post()
* @app.put()
* @app.delete()
* 等……

status\_code直接设定状态码

from fastapi import FastAPI

app = FastAPI()

@app.post("/items/", status\_code=201) # status\_code 参数接收表示 HTTP 状态码的数字。

async def create\_item(name: str):

return {"name": name}

关于 HTTP 状态码

在 HTTP 协议中，发送 3 位数的数字状态码是响应的一部分。

这些状态码都具有便于识别的关联名称，但是重要的还是数字。

简言之：

* 100 及以上的状态码用于返回**信息**。这类状态码很少直接使用。具有这些状态码的响应不能包含响应体
* **200** 及以上的状态码用于表示**成功**。这些状态码是最常用的
  + 200 是默认状态代码，表示一切**正常**
  + 201 表示**已创建**，通常在数据库中创建新记录后使用
  + 204 是一种特殊的例子，表示**无内容**。该响应在没有为客户端返回内容时使用，因此，该响应不能包含响应体
* **300** 及以上的状态码用于**重定向**。具有这些状态码的响应不一定包含响应体，但 304**未修改**是个例外，该响应不得包含响应体
* **400** 及以上的状态码用于表示**客户端错误**。这些可能是第二常用的类型
  + 404，用于**未找到**响应
  + 对于来自客户端的一般错误，可以只使用 400
* 500 及以上的状态码用于表示服务器端错误。几乎永远不会直接使用这些状态码。应用代码或服务器出现问题时，会自动返回这些状态代码

状态码名称快捷方式

从fastapi引入status后可以调用status的属性

from fastapi import FastAPI, status

实例：

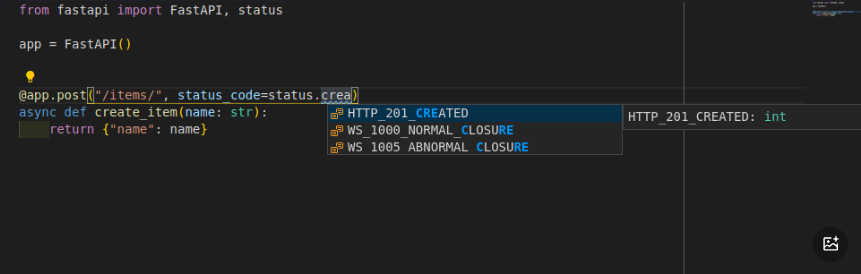
from fastapi import FastAPI, status

app = FastAPI()

@app.post("/items/", status\_code=status.HTTP\_201\_CREATED)

async def create\_item(name: str):

return {"name": name}



更改默认的响应状态码

使用 Response 参数，然后你可以在这个*临时*响应对象中设置status\_code。

实例：

from fastapi import FastAPI, Response, status #引入Response

app = FastAPI()

tasks = {"foo": "Listen to the Bar Fighters"}

@app.put("/get-or-create-task/{task\_id}", status\_code=200)

def get\_or\_create\_task(task\_id: str, response: Response):

if task\_id not in tasks:

tasks[task\_id] = "This didn't exist before"

response.status\_code = status.HTTP\_201\_CREATED #进入该条件设置返回的状态码

return tasks[task\_id]

表单数据

说明

接收的不是 JSON，而是表单字段时，要使用 Form。

使用 Form 可以声明与 Body （及 Query、Path、Cookie）相同的元数据和验证。

Form 是直接继承自 Body 的类。

声明表单体要显式使用 Form ，否则，FastAPI 会把该参数当作查询参数或请求体（JSON）参数。

例如，OAuth2 规范的 "密码流" 模式规定要通过表单字段发送 username 和 password。

该规范要求字段必须命名为 username 和 password，并通过表单字段发送，不能用 JSON。

什么是表单数据

与 JSON 不同，HTML 表单（<form></form>）向服务器发送数据通常使用「特殊」的编码。

**FastAPI** 要确保从正确的位置读取数据，而不是读取 JSON。

不包含文件时，表单数据一般用 application/x-www-form-urlencoded「媒体类型」编码。

但表单包含文件时，编码为 multipart/form-data。使用了 File，**FastAPI** 就知道要从请求体的正确位置获取文件。

可在一个*路径操作*中声明多个 File 和 Form 参数，但不能同时声明要接收 JSON 的 Body 字段。因为此时请求体的编码是 multipart/form-data，不是 application/json。

这不是 **FastAPI** 的问题，而是 HTTP 协议的规定。

导入form

要使用表单，需预先安装 [python-multipart](https://github.com/Kludex/python-multipart)

pip install python-multipart

from fastapi import FastAPI, Form

app = FastAPI()

@app.post("/login/")

async def login(username: str = Form(), password: str = Form()):

return {"username": username}

关于 "表单字段"

与 JSON 不同，HTML 表单（<form></form>）向服务器发送数据通常使用「特殊」的编码。

**FastAPI** 要确保从正确的位置读取数据，而不是读取 JSON。

表单数据的「媒体类型」编码一般为 application/x-www-form-urlencoded。

但包含文件的表单编码为 multipart/form-data

表单模型

from typing import Annotated

from fastapi import FastAPI, Form

from pydantic import BaseModel

app = FastAPI()

class FormData(BaseModel): #定义表单类

username: str

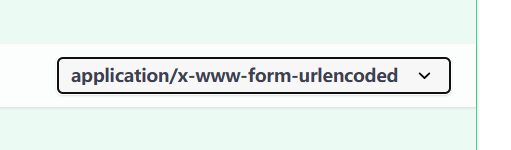
password: str

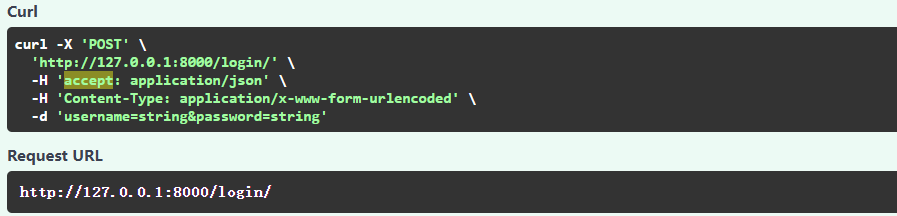
@app.post(“/login/”)

async def login(data: Annotated[FormData, Form()]): #显示声明为表单类

return data

输入的数据：





禁止额外的表单字段

class FormData(BaseModel):

username: str

password: str

model\_config = {"extra": "forbid"} #使用该参数输入额外的表单字段会报错

请求文件

说明

File 用于定义客户端的上传文件。

预先安装 pip install python-multipart

File 是直接继承自 Form 的类。

注意，从 fastapi 导入的 Query、Path、File 等项，实际上是返回特定类的函数。

导入File

从 fastapi 导入 File 和 UploadFile

from fastapi import FastAPI, File, UploadFile

app = FastAPI()

@app.post("/files/")

async def create\_file(file: bytes = File()):

return {"file\_size": len(file)}

@app.post("/uploadfile/")

async def create\_upload\_file(file: UploadFile):

return {"filename": file.filename}

定义 File 参数

File 是直接继承自 Form 的类。

注意，从 fastapi 导入的 Query、Path、File 等项，实际上是返回特定类的函数。

@app.post("/files/")

async def create\_file(file: bytes = File()):

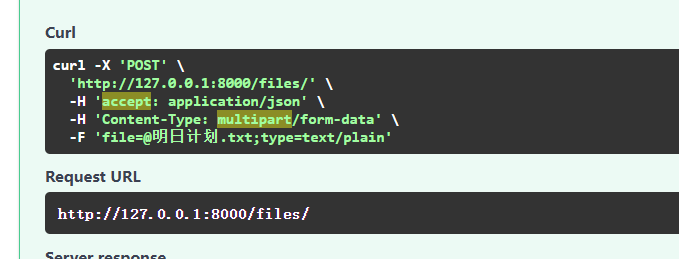
return {"file\_size": len(file)}

@app.post("/uploadfile/")

async def create\_upload\_file(file: UploadFile):

return {"filename": file.filename}

html的file输入框的name属性要与file: UploadFile的file名字相同



UploadFile和bytes参数区别

这里接收的文件是python的文件类型，可以用python的文件操作方法，write，read，seek，open，close都能用。

bytes：

如果把*路径操作函数*参数的类型声明为 bytes，**FastAPI** 将以 bytes 形式读取和接收文件内容。

这种方式把文件的所有内容都存储在内存里，适用于小型文件。

用bytes就必须显示声明是文件类型，例如下面这些

async def create\_file(file: bytes | None = File(default=None)):

UploadFile：

不过，很多情况下，UploadFile 更好用。

使用UploadFile不用显示声明文件类型

async def create\_upload\_file(file: UploadFile | None = None):

* 存储在内存的文件超出最大上限时，FastAPI 会把文件存入磁盘；
* 这种方式更适于处理图像、视频、二进制文件等大型文件，好处是不会占用所有内存

UploadFile 的属性如下：

* filename：上传文件名字符串（str），例如， myimage.jpg；
* content\_type：内容类型（MIME 类型 / 媒体类型）字符串（str），例如，image/jpeg；
* file： [SpooledTemporaryFile](https://docs.python.org/zh-cn/3/library/tempfile.html#tempfile.SpooledTemporaryFile)（ [file-like](https://docs.python.org/zh-cn/3/glossary.html#term-file-like-object) 对象）。其实就是 Python文件，可直接传递给其他预期 file-like 对象的函数或支持库。

UploadFile 支持以下 async 方法，（使用内部 SpooledTemporaryFile）可调用相应的文件方法。

* write(data)：把 data （str 或 bytes）写入文件；
* read(size)：按指定数量的字节或字符（size (int)）读取文件内容；
* seek(offset)：移动至文件 offset （int）字节处的位置；
  + 例如，await myfile.seek(0) 移动到文件开头；
  + 执行 await myfile.read() 后，需再次读取已读取内容时，这种方法特别好用；
* close()：关闭文件。

因为上述方法都是 async 方法，要搭配「await」使用。

例如，在 async *路径操作函数* 内，要用以下方式读取文件内容：

contents = await myfile.read()

在普通 def *路径操作函数* 内，则可以直接访问 UploadFile.file，例如：

contents = myfile.file.read()

可选文件上传

声明默认值为None

async def create\_upload\_file(file: UploadFile | None = None):

async def create\_file(file: bytes | None = File(default=None)):

带有额外元数据的 UploadFile

您也可以将 File() 与 UploadFile 一起使用，例如，设置额外的元数据:

@app.post("/uploadfile/")

async def create\_upload\_file(

file: UploadFile = File(description="A file read as UploadFile"),

):

return {"filename": file.filename}

多文件上传

上传多个文件时，要声明含 bytes 或 UploadFile 的列表（List）：

from fastapi import FastAPI, File, UploadFile

from fastapi.responses import HTMLResponse

app = FastAPI()

@app.post("/files/")

async def create\_files(files: list[bytes] = File()): #声明多文件上传

return {"file\_sizes": [len(file) for file in files]}

@app.post("/uploadfiles/")

async def create\_upload\_files(files: list[UploadFile]): ##声明多文件上传

return {"filenames": [file.filename for file in files]}

@app.get("/")

async def main():

content = """ #下面这段为html代码

<body>

<form action="/files/" enctype="multipart/form-data" method="post">

<input name="files" type="file" multiple>

<input type="submit">

</form>

<form action="/uploadfiles/" enctype="multipart/form-data" method="post">

<input name="files" type="file" multiple>

<input type="submit">

</form>

</body>

"""

return HTMLResponse(content=content)

请求表单与文件

导入 File 与 Form

可在一个*路径操作*中声明多个 File 与 Form 参数，但不能同时声明要接收 JSON 的 Body 字段。因为此时请求体的编码为 multipart/form-data，不是 application/json。

from fastapi import FastAPI, File, Form, UploadFile

app = FastAPI()

@app.post("/files/")

async def create\_file(

file: bytes = File(), fileb: UploadFile = File(), token: str = Form()

):

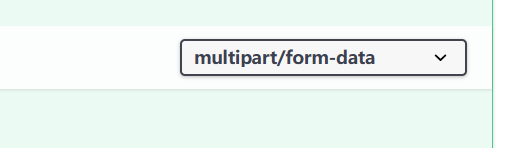
return {

"file\_size": len(file),

"token": token,

"fileb\_content\_type": fileb.content\_type,

}





路径操作配置

就是post里面不包括路径的的参数配置

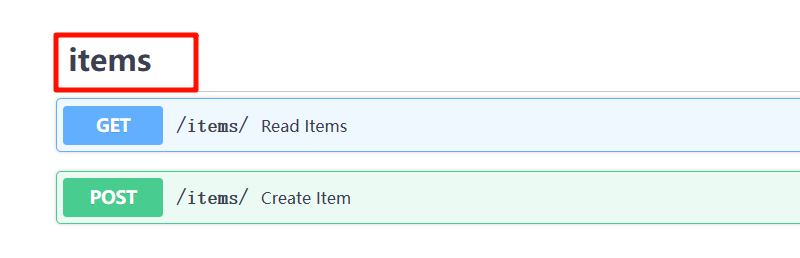
1. 响应模式和状态码

@app.post("/items/", response\_model=Item, status\_code=status.HTTP\_201\_CREATED)

2. tags

@app.post("/items/", response\_model=Item, tags=["items"]) #注意是列表，不是str

红框为tags



3. summary 和 description

@app.post(

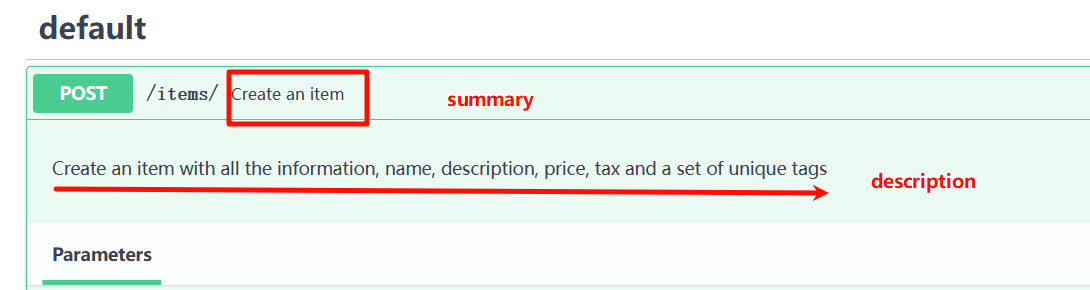
"/items/",

response\_model=Item,

summary="Create an item",

description="Create an item with all the information, name, description, price, tax and a set of unique tags",

)



4. 文档字符串（docstring）

就是在函数下面用3个引号括起来的内容 ‘’‘ 内容 ’‘’，这个内容是[Markdown](https://en.wikipedia.org/wiki/Markdown)写法，注意缩进。

描述内容比较长且占用多行时，可以在函数的 docstring 中声明*路径操作*的描述，**FastAPI** 支持从文档字符串中读取描述内容。

文档字符串支持 [Markdown](https://en.wikipedia.org/wiki/Markdown)，能正确解析和显示 Markdown 的内容，但要注意文档字符串的缩进。

from typing import Set, Union

from fastapi import FastAPI

from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):

name: str

description: Union[str, None] = None

price: float

tax: Union[float, None] = None

tags: Set[str] = set()

@app.post("/items/", response\_model=Item, summary="Create an item")

async def create\_item(item: Item): #

"""

Create an item with all the information:

- \*\*name\*\*: each item must have a name

- \*\*description\*\*: a long description

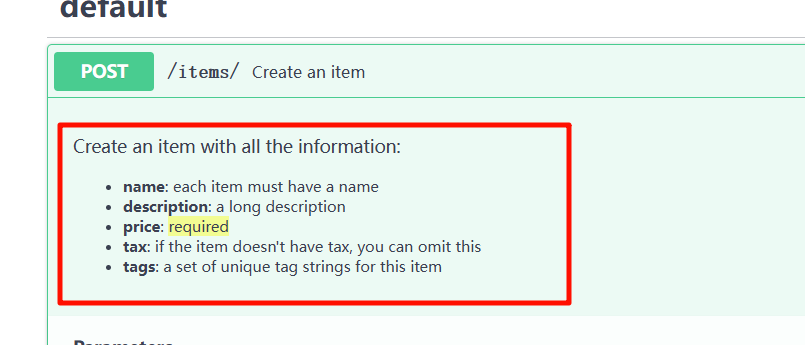
- \*\*price\*\*: required

- \*\*tax\*\*: if the item doesn't have tax, you can omit this

- \*\*tags\*\*: a set of unique tag strings for this item

"""

return item



5. response\_description响应描述

@app.post(

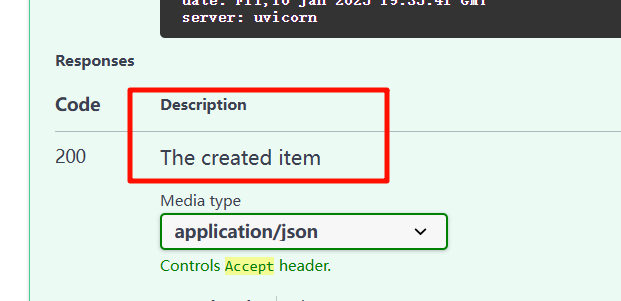
"/items/",

response\_model=Item,

summary="Create an item",

response\_description="The created item",

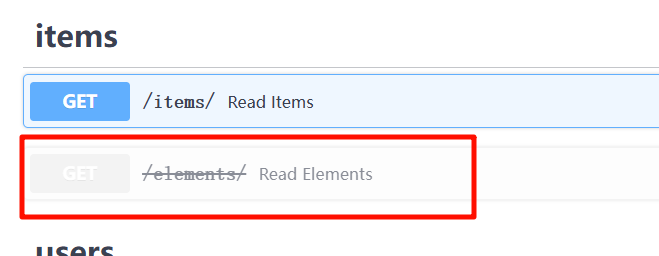
)



6. deprecated弃用路径操作

但还是可以请求

@app.get("/elements/", tags=["items"], deprecated=True)



元数据和文档 URL

API元数据

设置方式

在生成app对象时设置

示例：

from fastapi import FastAPI

description = """

ChimichangApp API helps you do awesome stuff. 🚀

## Items

You can \*\*read items\*\*.

## Users

You will be able to:

\* \*\*Create users\*\* (\_not implemented\_).

\* \*\*Read users\*\* (\_not implemented\_).

"""

app = FastAPI(

title="ChimichangApp",

description=description,

summary="Deadpool's favorite app. Nuff said.",

version="0.0.1",

terms\_of\_service="http://example.com/terms/",

contact={

"name": "Deadpoolio the Amazing",

"url": "http://x-force.example.com/contact/",

"email": "dp@x-force.example.com",

},

license\_info={

"name": "Apache 2.0",

"url": "https://www.apache.org/licenses/LICENSE-2.0.html",

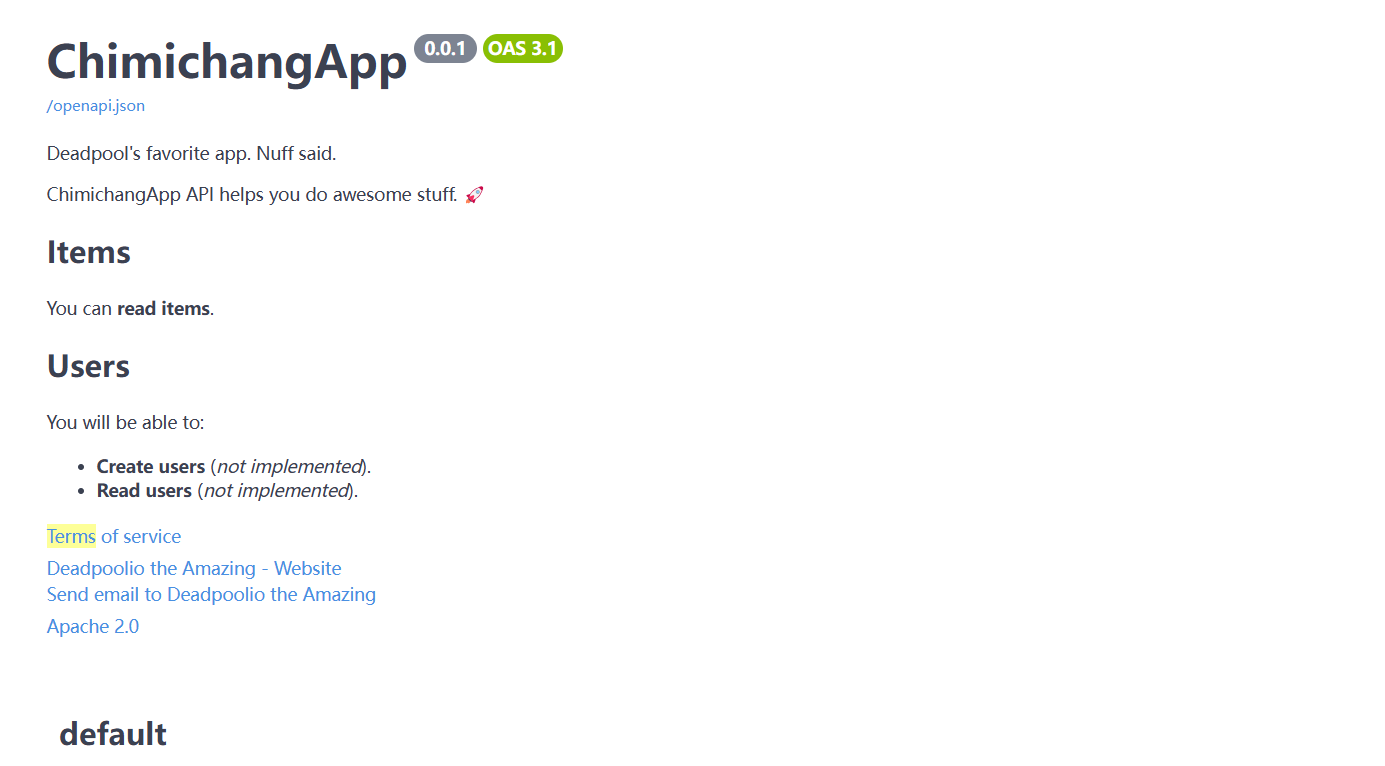
},

)

@app.get("/items/")

async def read\_items():

return [{"name": "Katana"}]





标签元数据

设置方式

from fastapi import FastAPI

# 定义标签元数据

tags\_metadata = [

{

"name": "users",

"description": "Operations with users. The \*\*login\*\* logic is also here.",

},

{

"name": "items",

"description": "Manage items. So \_fancy\_ they have their own docs.",

"externalDocs": {

"description": "Items external docs",

"url": "https://fastapi.tiangolo.com/",

},

},

]

app = FastAPI(openapi\_tags=tags\_metadata) #引入标签元数据

@app.get("/users/", tags=["users"]) #引入标签元数据

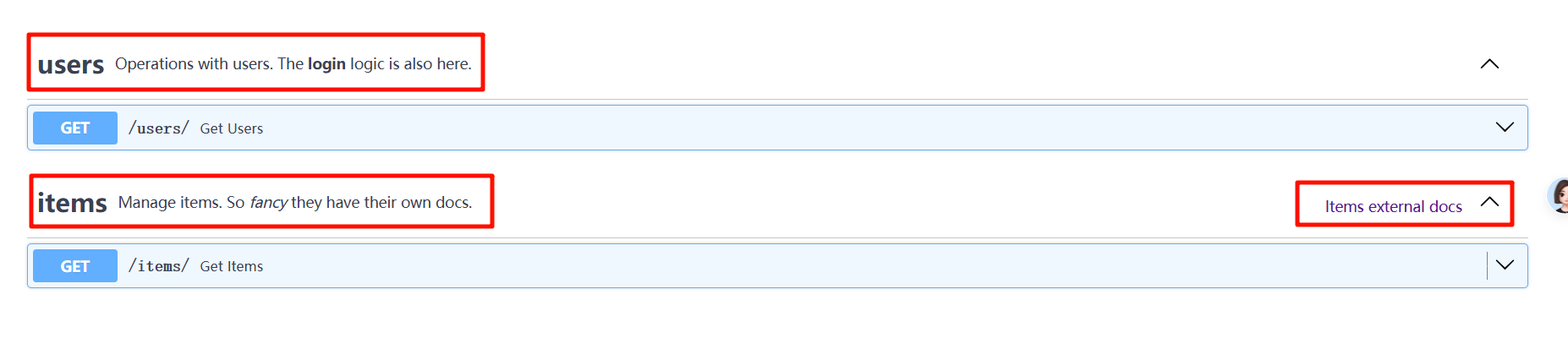
async def get\_users():

return [{"name": "Harry"}, {"name": "Ron"}]

@app.get("/items/", tags=["items"]) #引入标签元数据

async def get\_items():

return [{"name": "wand"}, {"name": "flying broom"}]



标签顺序

每个标签元数据字典的顺序也定义了在文档用户界面显示的顺序。

例如按照字母顺序，即使 users 排在 items 之后，它也会显示在前面，因为我们将它的元数据添加为列表内的第一个字典。

OpenAPI URL

默认情况下，OpenAPI 模式服务于 /openapi.json。

但是你可以通过参数 openapi\_url 对其进行配置。

例如，将其设置为服务于 /api/v1/openapi.json：

禁用

如果你想完全禁用 OpenAPI 模式，可以将其设置为 openapi\_url=None，这样也会禁用使用它的文档用户界面。

app = FastAPI(openapi\_tags=tags\_metadata, openapi\_url=None)

文档 URLs

* **Swagger UI**：服务于 /docs。
  + 可以使用参数 docs\_url 设置它的 URL。
  + 可以通过设置 docs\_url=None 禁用它。
* ReDoc：服务于 /redoc。
  + 可以使用参数 redoc\_url 设置它的 URL。
  + 可以通过设置 redoc\_url=None 禁用它。

例如

app = FastAPI(docs\_url="/documentation", redoc\_url=None)

静态文件、子应用、挂载

挂载和子应用

**挂载**是指在特定路径中添加完全**独立**的应用，然后在该路径下使用*路径操作*声明的子应用处理所有事务。

子应用：子应用只是另一个标准 FastAPI 应用，但这个应用是被**挂载**的应用：

例如：

app = FastAPI()

subapi = FastAPI() #子应用

app.mount("/subapi", subapi) #挂载子应用到/subapi这个路径

挂载子应用示例

在顶层应用 app 中，挂载子应用 subapi。

本例的子应用挂载在 /subapi 路径下：

from fastapi import FastAPI

app = FastAPI() #创建父应用

@app.get("/app")

def read\_main():

return {"message": "Hello World from main app"}

subapi = FastAPI() #创建子应用

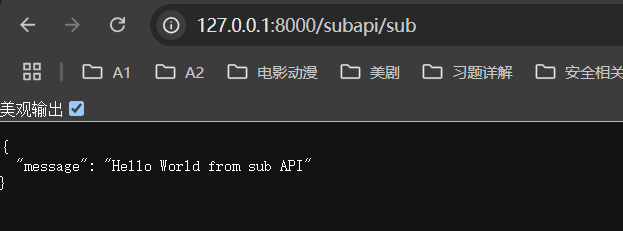
@subapi.get("/sub") #

def read\_sub():

return {"message": "Hello World from sub API"}

app.mount("/subapi", subapi) #挂载子应用

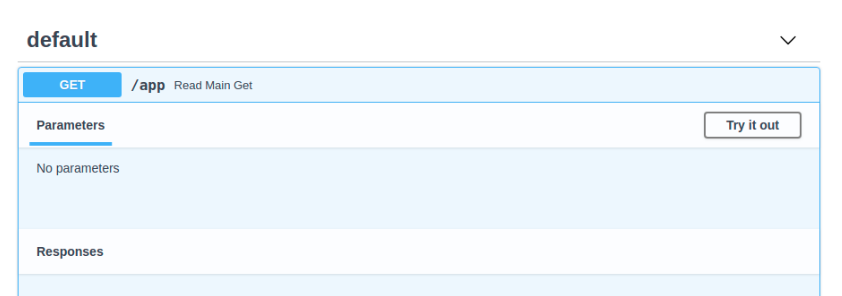
访问<http://127.0.0.1:8000/subapi/sub>地址效果



查看文档效果

查看文档 [http://127.0.0.1:8000/docs。](http://127.0.0.1:8000/docs)

下图显示的是主应用 API 文档，只包括其自有的*路径操作*。



然后查看子应用文档 [http://127.0.0.1:8000/subapi/docs。](http://127.0.0.1:8000/subapi/docs)

下图显示的是子应用的 API 文档，也是只包括其自有的*路径操作*，所有这些路径操作都在 /subapi 子路径前缀下。



注意路径操作的先后顺序

from fastapi import FastAPI

app = FastAPI()

@app.get("/subapi/") #这里的路径和子应用的路径相同

def read\_main():

    return {"message": "Hello World from main app"}

subapi = FastAPI()

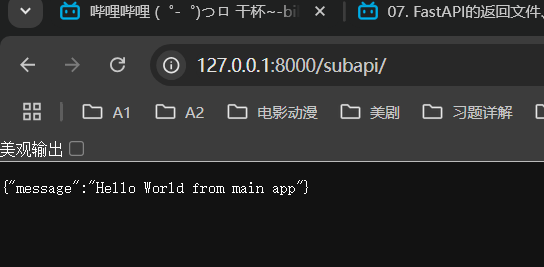
@subapi.get("/") #子应用的路径和父应用路径相同

def read\_sub():

    return {"message": "Hello World from sub API"}

app.mount("/subapi", subapi)

访问相同url，从上到下执行，第一个路由匹配到了就不再执行了



挂载静态文件

使用StaticFiles

* 导入StaticFiles。
* "挂载"(Mount) 一个 StaticFiles() 实例到一个指定路径。

[Python 3.8+](https://fastapi.tiangolo.com/zh/tutorial/static-files/#__tabbed_1_1)

from fastapi import FastAPI

from fastapi.staticfiles import StaticFiles #导入StaticFiles

app = FastAPI()

@app.get(“/subapi/”)

def read\_main():

return {“message”: “Hello World from main app”}

subapi = StaticFiles(directory=”./static”, html=True) #创建StaticFiles应用实例

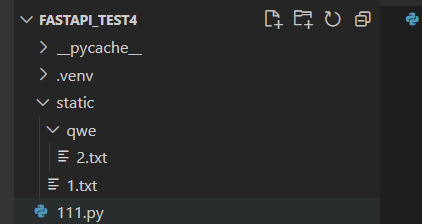
app.mount("/subapi", subapi) #挂载StaticFiles应用实例

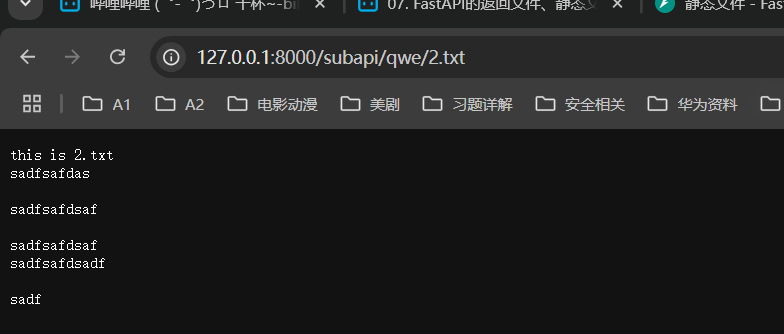
解释说明

StaticFiles(directory=”./static”, html=True)

directory=”./static”表示挂载在当前目录的static文件夹下

效果





处理错误

使用 HTTPException返回 HTTP 错误响应

向客户端返回 HTTP 错误响应，可以使用 HTTPException。

导入 HTTPException

from fastapi import FastAPI, HTTPException #导入HTTPException函数

app = FastAPI()

items = {"foo": "The Foo Wrestlers"}

@app.get("/items/{item\_id}")

async def read\_item(item\_id: str):

if item\_id not in items:

raise HTTPException(status\_code=404, detail="Item not found") #达到条件抛出异常

return {"item": items[item\_id]}

触发 HTTPException

HTTPException 是额外包含了和 API 有关数据的常规 Python 异常。

因为是 Python 异常，所以不能 return，只能 raise。

如在调用*路径操作函数*里的工具函数时，触发了 HTTPException，FastAPI 就不再继续执行*路径操作函数*中的后续代码，而是立即终止请求，并把 HTTPException 的 HTTP 错误发送至客户端。

触发 HTTPException 时，可以用参数 detail 传递任何能转换为 JSON 的值，不仅限于 str。

还支持传递 dict、list 等数据结构。

**FastAPI** 能自动处理这些数据，并将之转换为 JSON。



添加自定义响应头

from fastapi import FastAPI, HTTPException

app = FastAPI()

items = {"foo": "The Foo Wrestlers"}

@app.get("/items-header/{item\_id}")

async def read\_item\_header(item\_id: str):

if item\_id not in items:

raise HTTPException(

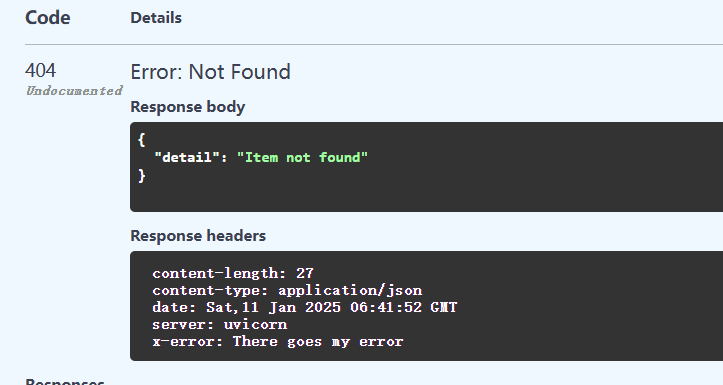
status\_code=404,

detail="Item not found",

headers={"X-Error": "There goes my error"}, #添加自定义响应头

)

return {"item": items[item\_id]}



自定义异常处理器

from fastapi import FastAPI, Request

from fastapi.responses import JSONResponse #导入异常相关对象

class UnicornException(Exception): #定义异常类

def \_\_init\_\_(self, name: str):

self.name = name

app = FastAPI()

@app.exception\_handler(UnicornException) #定义异常处理结果

async def unicorn\_exception\_handler(request: Request, exc: UnicornException):

return JSONResponse(

status\_code=418,

content={"message": f"Oops! {exc.name} did something. There goes a rainbow..."},

)

@app.get("/unicorns/{name}")

async def read\_unicorn(name: str):

if name == "yolo":

raise UnicornException(name=name) #返回异常对象

return {"unicorn\_name": name}

覆盖默认异常处理器

JSON 兼容编码器

在某些情况下，您可能需要将数据类型（如Pydantic模型）转换为与JSON兼容的数据类型（如dict、list等）。

使用jsonable\_encoder[¶](https://fastapi.tiangolo.com/zh/tutorial/encoder/#jsonable_encoder)

让我们假设你有一个数据库名为fake\_db，它只能接收与JSON兼容的数据。

例如，它不接收datetime这类的对象，因为这些对象与JSON不兼容。

因此，datetime对象必须将转换为包含[ISO格式化](https://en.wikipedia.org/wiki/ISO_8601)的str类型对象。

同样，这个数据库也不会接收Pydantic模型（带有属性的对象），而只接收dict。

对此你可以使用jsonable\_encoder。

它接收一个对象，比如Pydantic模型，并会返回一个JSON兼容的版本：

from datetime import datetime

from fastapi import FastAPI

from fastapi.encoders import jsonable\_encoder

from pydantic import BaseModel

fake\_db = {}

class Item(BaseModel):

title: str

timestamp: datetime

description: str | None = None

app = FastAPI()

@app.put("/items/{id}")

def update\_item(id: str, item: Item):

json\_compatible\_item\_data = jsonable\_encoder(item)

fake\_db[id] = json\_compatible\_item\_data

在这个例子中，它将Pydantic模型转换为dict，并将datetime转换为str。

依赖项

名词解释

依赖项

* **定义**：可以理解为一个软件组件（比如一个类或者一个函数）在运行过程中所需要的其他组件、对象、数据等。简单说，就是某个东西要正常工作所离不开的其他东西。

**依赖项在fastapi里就是函数和类**

* **示例**：假如有一个做饭的函数，做饭需要锅、食材等，那么锅和食材就是做饭这个函数的依赖项，没有它们，做饭函数就无法正常执行。

依赖注入

* **定义**：是一种设计模式，主要目的是让组件之间的依赖关系更加灵活和可管理。简单来讲，就是把一个组件所需要的依赖项，以某种方式传递（注入）给这个组件，而不是让组件自己去创建或获取这些依赖项。

**依赖注入方式在fastapi里就是采用depends 实现**

* **示例**：还是以做饭为例，以前是做饭函数自己去买锅、找食材，现在通过依赖注入，由外部把已经准备好的锅和食材传递给做饭函数，这样做饭函数就不用关心锅和食材是从哪里来的，只需要专注于做饭的过程。这样一来，如果要换不同的锅或者食材，只需要在外部进行更换后再注入给做饭函数就行，而不用去修改做饭函数内部的代码，使代码的可维护性和可扩展性更强。

Depends

虽然，在路径操作函数的参数中使用 Depends 的方式与 Body、Query 相同，但 Depends 的工作方式略有不同。

这里只能传给 Depends 一个参数。

且该参数必须是可调用对象，比如函数。

该函数接收的参数和*路径操作函数*的参数一样。

对于

@app.get("/items/")

async def read\_items(commons: CommonQueryParams = Depends(CommonQueryParams)):

FastAPI 不会使用commons: CommonQueryParams进行数据转换、验证等 (它使用 =Depends(CommonQueryParams))。

会使用依赖项的参数进行数据转换、验证等.

要不要使用 async

**FastAPI** 调用依赖项的方式与*路径操作函数*一样，因此，定义依赖项函数，也要应用与路径操作函数相同的规则。

既可以使用异步的 async def，也可以使用普通的 def 定义依赖项。

在普通的 def *路径操作函数*中，可以声明异步的 async def 依赖项；也可以在异步的 async def *路径操作函数*中声明普通的 def 依赖项。

上述这些操作都是可行的，**FastAPI** 知道该怎么处理。

函数作为依赖项

from typing import Union

from fastapi import Depends, FastAPI #导入Depends

app = FastAPI()

async def common\_parameters( #创建依赖项函数

q: Union[str, None] = None, skip: int = 0, limit: int = 100

):

return {"q": q, "skip": skip, "limit": limit}

@app.get("/items/")

async def read\_items(commons: dict = Depends(common\_parameters)): #引入依赖

return commons

@app.get("/users/")

async def read\_users(commons: dict = Depends(common\_parameters)): #引入依赖

return commons

类作为依赖项

示例

from fastapi import Depends, FastAPI #导入Depends

app = FastAPI()

fake\_items\_db = [{"item\_name": "Foo"}, {"item\_name": "Bar"}, {"item\_name": "Baz"}]

class CommonQueryParams: #创建类依赖项

def \_\_init\_\_(self, q: str | None = None, skip: int = 0, limit: int = 100):

self.q = q

self.skip = skip

self.limit = limit

@app.get("/items/")

async def read\_items(commons: CommonQueryParams = Depends(CommonQueryParams)): #依赖注入

response = {}

if commons.q:

response.update({"q": commons.q})

items = fake\_items\_db[commons.skip : commons.skip + commons.limit]

response.update({"items": items})

return response

在两个例子下，都有：

* 一个可选的 q 查询参数，是 str 类型。
* 一个 skip 查询参数，是 int 类型，默认值为 0。
* 一个 limit 查询参数，是 int 类型，默认值为 100。

在两个例子下，数据都将被转换、验证、在 OpenAPI schema 上文档化，等等。

依赖注入语法

commons: CommonQueryParams = Depends(CommonQueryParams) #推荐使用这种语法，这样更清晰

commons = Depends(CommonQueryParams)

commons: CommonQueryParams = Depends()

子依赖项

示例

from typing import Union

from fastapi import Cookie, Depends, FastAPI #导入Depends

app = FastAPI()

def query\_extractor(q: Union[str, None] = None): #创建父依赖项

return q

def query\_or\_cookie\_extractor(

q: str = Depends(query\_extractor), #创建子依赖项，依赖query\_extractor

last\_query: Union[str, None] = Cookie(default=None),

):

if not q:

return last\_query

return q

@app.get("/items/")

async def read\_query(query\_or\_default: str = Depends(query\_or\_cookie\_extractor)): #依赖注入，引用子依赖项Depends(query\_or\_cookie\_extractor)

return {"q\_or\_cookie": query\_or\_default}

多次使用同一个依赖项

如果在同一个*路径操作* 多次声明了同一个依赖项，例如，多个依赖项共用一个子依赖项，**FastAPI** 在处理同一请求时，只调用一次该子依赖项。

FastAPI 不会为同一个请求多次调用同一个依赖项，而是把依赖项的返回值进行「缓存」，并把它传递给同一请求中所有需要使用该返回值的「依赖项」。

在高级使用场景中，如果不想使用「缓存」值，而是为需要在同一请求的每一步操作（多次）中都实际调用依赖项，可以把 Depends 的参数 use\_cache 的值设置为 False :

async def needy\_dependency(fresh\_value: str = Depends(get\_value, use\_cache=False)):

return {"fresh\_value": fresh\_value}

路径操作装饰器依赖项

有时，我们并不需要在*路径操作函数*中使用依赖项的返回值。

或者说，有些依赖项不返回值。

但仍要执行或解析该依赖项。

对于这种情况，不必在声明*路径操作函数*的参数时使用 Depends，而是可以在*路径操作装饰器*中添加一个由 dependencies 组成的 list。

示例

from fastapi import Depends, FastAPI, Header, HTTPException

app = FastAPI()

async def verify\_token(x\_token: str = Header()):

if x\_token != "fake-super-secret-token":

raise HTTPException(status\_code=400, detail="X-Token header invalid")

async def verify\_key(x\_key: str = Header()):

if x\_key != "fake-super-secret-key":

raise HTTPException(status\_code=400, detail="X-Key header invalid")

return x\_key

@app.get("/items/", dependencies=[Depends(verify\_token), Depends(verify\_key)])

async def read\_items():

return [{"item": "Foo"}, {"item": "Bar"}]

路径操作装饰器依赖项（以下简称为**“路径装饰器依赖项”**）的执行或解析方式和普通依赖项一样，但就算这些依赖项会返回值，它们的值也不会传递给*路径操作函数*。

全局依赖项

有时，我们要为整个应用添加依赖项。

通过与定义[*路径装饰器依赖项*](https://fastapi.tiangolo.com/zh/tutorial/dependencies/dependencies-in-path-operation-decorators/) 类似的方式，可以把依赖项添加至整个 FastAPI 应用。

这样一来，就可以为所有*路径操作*应用该依赖项：

示例

from fastapi import Depends, FastAPI, Header, HTTPException

async def verify\_token(x\_token: str = Header()):

if x\_token != "fake-super-secret-token":

raise HTTPException(status\_code=400, detail="X-Token header invalid")

async def verify\_key(x\_key: str = Header()):

if x\_key != "fake-super-secret-key":

raise HTTPException(status\_code=400, detail="X-Key header invalid")

return x\_key

app = FastAPI(dependencies=[Depends(verify\_token), Depends(verify\_key)]) #引入全局依赖项

@app.get("/items/")

async def read\_items():

return [{"item": "Portal Gun"}, {"item": "Plumbus"}]

@app.get("/users/")

async def read\_users():

return [{"username": "Rick"}, {"username": "Morty"}]

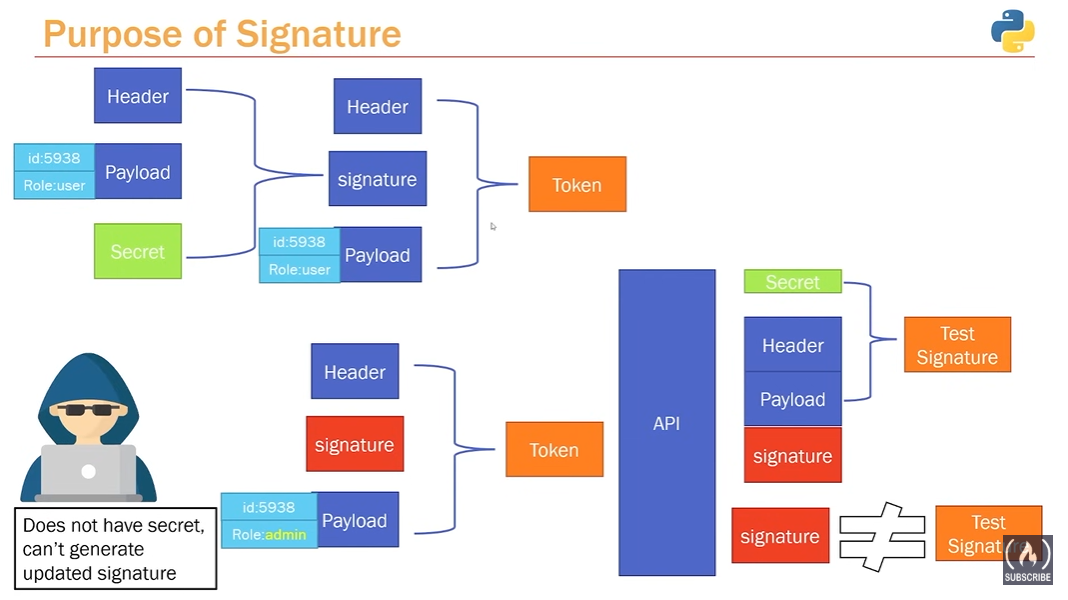
OAuth2、JWT、Token

Token

介绍

Token 是一种**令牌**，通常用于身份验证和授权。它可以理解为一个“通行证”，用来证明用户的身份或权限。常见的 Token 类型包括：

Token 的组成



Header（头部）：

描述 Token 的类型和签名算法。

示例：

json

{

"alg": "HS256", // 签名算法

"typ": "JWT" // Token 类型

}

Payload（载荷）：

包含实际的数据（如用户 ID、角色、过期时间等）。

示例：

json

{

"sub": "1234567890", // 用户 ID

"name": "John Doe", // 用户名

"iat": 1516239022 // 签发时间

}

Signature（签名）：

用 Header 和 Payload 加上一个密钥（Secret）生成，用来验证 Token 的真实性。

示例：

HMACSHA256(

base64UrlEncode(header) + "." + base64UrlEncode(payload),

secret

)

密码hash与校验

为了安全，数据库保存明文密码，只保存经过hash后的字符串。

案例：

先安装pip install passlib[bcrypt]

from passlib.context import CryptContext

pwd\_context = CryptContext(schemes=["bcrypt"], deprecated="auto")   #加密方式为bcrypt，自动选择最新的加密方式

def hash\_password(password: str):   #hash密码，参数：明文密码

    return pwd\_context.hash(password)

def verify\_password(plain\_password, hashed\_password):   #验证密码，参数1：明文密码，参数2：加密后的密码

    return pwd\_context.verify(plain\_password, hashed\_password)

OAuth2 是什么

OAuth2 是一种**授权框架**，用来让一个应用程序（比如一个网站或手机 App）安全地访问用户在另一个应用程序（比如 Google、Facebook）中的资源（比如你的照片、邮箱、联系人等），而不用直接告诉它你的密码。

举个例子：

* 你想用 Google 账号登录一个第三方网站（比如 Medium）。
* 你不想把 Google 密码告诉 Medium。
* OAuth2 就是让 Medium 通过 Google 的授权，安全地访问你的 Google 信息（比如你的名字和邮箱），而不需要知道你的密码。

OAuth2 的工作流程

OAuth2 的工作流程可以简单理解为以下几步：

1. **用户同意授权**：
   * 客户端（Medium）告诉用户：“嘿，我想访问你的 Google 信息，可以吗？”
   * 用户同意后，会被重定向到 Google 的登录页面。
2. **获取授权码**：
   * 用户在 Google 登录页面输入账号和密码。
   * Google 验证成功后，会给客户端（Medium）一个**授权码**。
3. **换取访问令牌**：
   * 客户端（Medium）用授权码向 Google 换取一个**访问令牌（Access Token）**。
4. **访问资源**：
   * 客户端（Medium）用访问令牌向 Google 请求用户的信息（比如名字和邮箱）。
   * Google 验证令牌后，返回用户的信息。

**举个例子**

假设你想用 Google 账号登录 Medium：

1. Medium 会跳转到 Google 的登录页面。
2. 你输入 Google 账号和密码，同意 Medium 访问你的邮箱和名字。
3. Google 给 Medium 一个授权码。
4. Medium 用授权码向 Google 换取访问令牌。
5. Medium 用访问令牌向 Google 请求你的邮箱和名字。
6. Google 返回你的邮箱和名字，Medium 完成登录。

OAuth2 的常见术语

* **Access Token（访问令牌）**：客户端用来访问用户资源的“钥匙”。
* **Refresh Token（刷新令牌）**：用来获取新的访问令牌，避免用户频繁登录。
* **Scope（权限范围）**：定义客户端可以访问哪些资源（比如只读、读写等）。
* **Authorization Code（授权码）**：客户端用来换取访问令牌的临时代码。

JWT 简介

JWT 即**JSON 网络令牌**（JSON Web Tokens）。

JWT 是一种将 JSON 对象编码为没有空格，且难以理解的长字符串的标准。

JWT 的工作流程

JWT 的工作流程可以简单理解为以下几步：

1. **用户登录**：
   * 用户输入用户名和密码，发送给服务器。
   * 服务器验证成功后，生成一个 JWT 并返回给用户。
2. **携带 JWT 访问资源**：
   * 用户每次访问其他页面时，都会在请求头中带上这个 JWT。
   * 示例：

Authorization: Bearer <JWT>

1. **服务器验证 JWT**：
   * 服务器收到请求后，验证 JWT 的签名是否有效。
   * 如果有效，服务器会从 Payload 中提取用户信息，然后允许用户访问资源。

OAuth2PasswordRequestForm获取 username 和 password

使用该方法需导入

 from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm

OAuth2 规范要求使用**密码流**时，客户端或用户必须以表单数据形式发送 username 和 password 字段。

并且，这两个字段必须命名为 username 和 password ，不能使用 user-name 或 email 等其它名称。

OAuth2PasswordRequestForm 是用以下几项内容声明表单请求体的类依赖项：

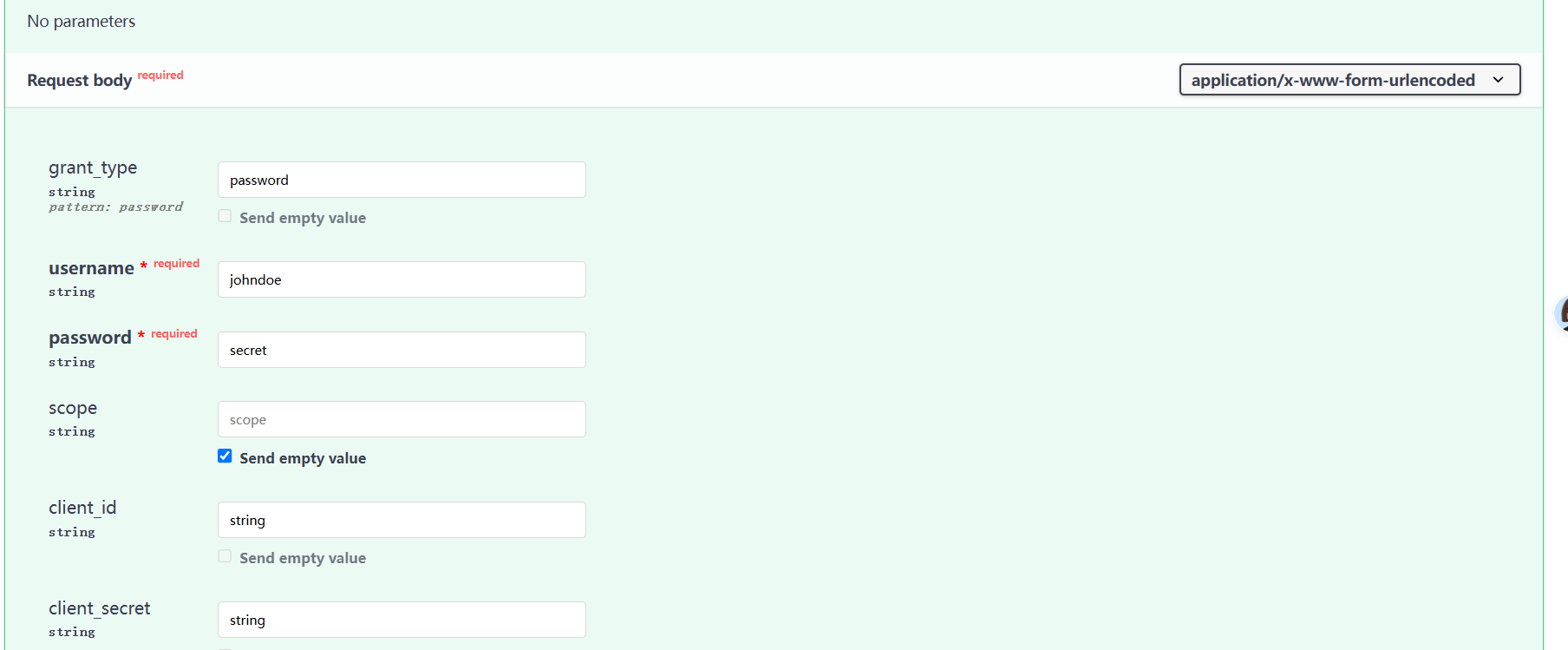
* username
* password
* 可选的 scope 字段，由多个空格分隔的字符串组成的长字符串
* 可选的 grant\_type
* 可选的 client\_id（本例未使用）
* 可选的 client\_secret（本例未使用）

例如

@app.post("/token")

async def login(form\_data: OAuth2PasswordRequestForm = Depends()):

user\_dict = fake\_users\_db.get(form\_data.username)





返回 Token

token 端点的响应必须是 JSON 对象。

响应返回的内容应该包含 token\_type。本例中用的是**Bearer**Token，因此， Token 类型应为**bearer**。

返回内容还应包含 access\_token 字段，它是包含权限 Token 的字符串。

@app.post("/token")

async def login(form\_data: OAuth2PasswordRequestForm = Depends()):

    user\_dict = fake\_users\_db.get(form\_data.username)

    if not user\_dict:

        raise HTTPException(status\_code=400, detail="Incorrect username or password")

    user = User(\*\*user\_dict)

    if not form\_data.password == user.password:

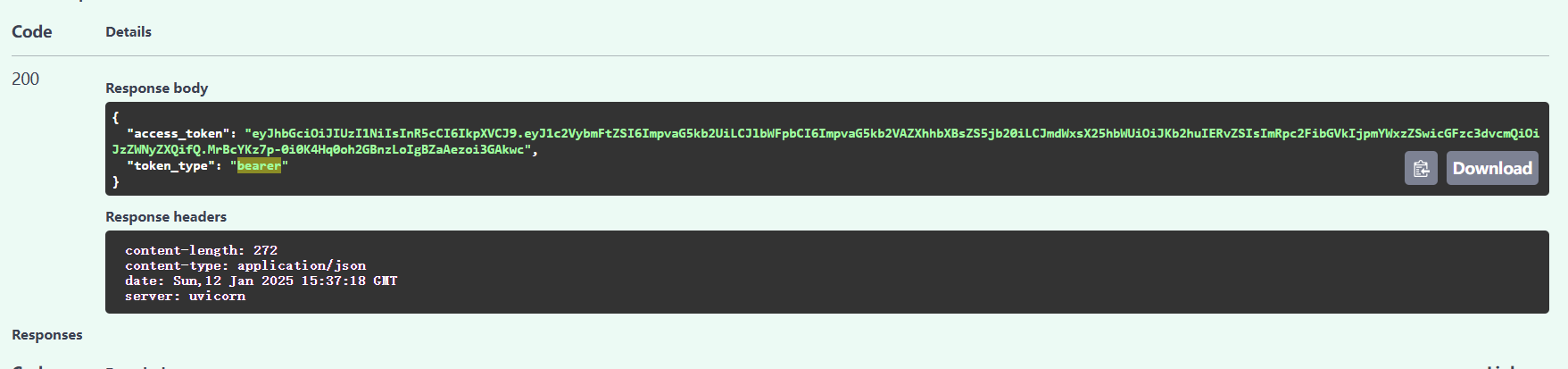
        raise HTTPException(status\_code=400, detail="Incorrect username or password")

    token\_data = user.dict()

token = jwt.encode(token\_data, SECRET\_KEY, algorithm=ALGORITHM)

#返回的boby必须是这种形式

    return {"access\_token": token, "token\_type": "bearer"}



OAuth2PasswordBearer获取请求头的token

使用该方法需导入

 from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm

在此，tokenUrl="token" 指向的是暂未创建的相对 URL token。这个相对 URL 相当于 ./token。

因为使用的是相对 URL，如果 API 位于 https://example.com/，则指向 https://example.com/token。但如果 API 位于 https://example.com/api/v1/，它指向的就是https://example.com/api/v1/token。

简而言之就是获取token的url

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token") #创建oauth2\_scheme

async def get\_current\_user(token: str = Depends(oauth2\_scheme)): #获取token

    user\_data = jwt.decode(token, SECRET\_KEY, algorithms=[ALGORITHM])

    user = User(\*\*user\_data)

    print(user)

    if not user:

        raise HTTPException(

            status\_code=status.HTTP\_401\_UNAUTHORIZED,

            detail="Invalid authentication credentials",

            headers={"WWW-Authenticate": "Bearer"},

        )

return user

请求头的token数据



jwt.encode（）和jwt.decode（）编解码token

使用该方法需导入

import jwt

SECRET\_KEY = "09d25e094faa6ca2556c818166b7a9563b93f7099f6f0f4caa6cf63b88e8d3e7"

ALGORITHM = "HS256"

ACCESS\_TOKEN\_EXPIRE\_MINUTES = 30

# encoded\_jwt = jwt.encode(token\_data, SECRET\_KEY, algorithm=ALGORITHM) #编码token

# payload = jwt.decode(token, SECRET\_KEY, algorithms=ALGORITHM) #解码token

第一个参数是token数据

第二个参数是密钥

第三个参数是加密方法

jwt自动验证时间

1. **JWT 中的时间相关声明**
   * **exp（Expiration Time）声明**：这是最常用的时间相关声明。它定义了 JWT 的过期时间。例如，一个exp的值可以是一个 Unix 时间戳，表示从 1970 年 1 月 1 日 UTC 开始到令牌过期的秒数。当验证 JWT 时，服务器会检查当前时间是否超过了exp所指定的时间。如果超过了，那么该 JWT 将被视为无效。
   * **nbf（Not Before Time）声明**：规定了 JWT 在这个时间之前是无效的。这个时间同样可以是一个 Unix 时间戳。这可以用于确保 JWT 在某个特定时间之后才开始生效，比如在用户完成某个操作（如激活账户）之后。
   * **iat（Issued At）声明**：表示 JWT 的发布时间，也是以 Unix 时间戳的形式出现。虽然这个声明本身不是用于验证时间有效性的主要依据，但它可以在一些场景下用于审计或者计算令牌的有效期范围等。
2. **验证时间的实现方式**
   * **在服务器端验证**：在大多数 JWT 的应用场景中，服务器端会对 JWT 进行验证。在验证过程中，服务器会获取当前时间（通常是服务器的系统时间），然后与 JWT 中的exp和nbf声明进行比较。

案例：

import jwt

import datetime

from jwt.exceptions import ExpiredSignatureError, InvalidTokenError

SECRET\_KEY = 'your\_secret\_key'

def create\_jwt():

payload = {

"user\_id": 1,

"iat": datetime.datetime.utcnow(), #时间key，注意要用utcnow

"exp": datetime.datetime.utcnow() + datetime.timedelta(minutes = 30) #时间key，注意utcnow

}

token = jwt.encode(payload, SECRET\_KEY, algorithm="HS256")

return token

def verify\_jwt(token):

try:

payload = jwt.decode(token, SECRET\_KEY, algorithms=["HS256"])

print("JWT验证成功，有效负载：", payload)

except ExpiredSignatureError:

print("JWT已过期")

except InvalidTokenError:

print("JWT无效")

token = create\_jwt()

verify\_jwt(token)

* 在这个示例中，首先创建了一个 JWT，其中包含了iat（发布时间）和exp（过期时间）声明。在验证函数verify\_jwt中，如果当前时间超过了exp时间，就会抛出ExpiredSignatureError，表示 JWT 已过期。

案例

from typing import Union

import jwt   #导入jwt

from fastapi import Depends, FastAPI, HTTPException, status

from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm #导入OAuth2

from pydantic import BaseModel

# 伪造数据库

fake\_users\_db = {

    "johndoe": {

        "username": "johndoe",

        "full\_name": "John Doe",

        "email": "johndoe@example.com",

        "password": "secret",

        "disabled": False,

    },

    "alice": {

        "username": "alice",

        "full\_name": "Alice Wonderson",

        "email": "alice@example.com",

        "password": "secret2",

        "disabled": True,

    },

}

app = FastAPI()

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="token") #生成OAuth2PasswordBearer(tokenUrl="token")对象，指明认证tokenUrl

SECRET\_KEY = "09d25e094faa6ca2556c818166b7a9563b93f7099f6f0f4caa6cf63b88e8d3e7"

ALGORITHM = "HS256"

ACCESS\_TOKEN\_EXPIRE\_MINUTES = 30

# encoded\_jwt = jwt.encode(to\_encode, SECRET\_KEY, algorithm=ALGORITHM)

# payload = jwt.decode(token, SECRET\_KEY, algorithms=[ALGORITHM])

class User(BaseModel):

    username: str

    email: Union[str, None] = None

    full\_name: Union[str, None] = None

    disabled: Union[bool, None] = None

    password: str

def get\_user(db, username: str):

    if username in db:

        user\_dict = db[username]

        return User(\*\*user\_dict)

async def get\_current\_user(token: str = Depends(oauth2\_scheme)):

    user\_data = jwt.decode(token, SECRET\_KEY, algorithms=ALGORITHM) #获取token数据

    user = User(\*\*user\_data)

    print(user)

    if not user:

        raise HTTPException(

            status\_code=status.HTTP\_401\_UNAUTHORIZED,

            detail="Invalid authentication credentials",

            headers={"WWW-Authenticate": "Bearer"},

        )

    return user

async def get\_current\_active\_user(current\_user: User = Depends(get\_current\_user)):

    if current\_user.disabled:

        raise HTTPException(status\_code=400, detail="Inactive user")

    return current\_user

@app.post("/token")

async def login(form\_data: OAuth2PasswordRequestForm = Depends()): #获取username和password

    user\_dict = fake\_users\_db.get(form\_data.username)

    if not user\_dict:

        raise HTTPException(status\_code=400, detail="Incorrect username or password")

    user = User(\*\*user\_dict)

    if not form\_data.password == user.password:

        raise HTTPException(status\_code=400, detail="Incorrect username or password")

    token\_data = user.dict()

    token = jwt.encode(token\_data, SECRET\_KEY, algorithm=ALGORITHM)

    return {"access\_token": token, "token\_type": "bearer"} #返回token

    # return token(token\_data=token, token\_type="bearer")

@app.get("/users/me")

async def read\_users\_me(current\_user: User = Depends(get\_current\_active\_user)):

    return current\_user

APIRouter

作用和主app相同

用法：

posts.py文件

from fastapi import APIRouter

router = APIRouter(prefix='/posts', tags=['posts'])

# 获取所有文章的API端点

@router.get('/', response\_model=list[PostResponse])

def get\_posts(db: Session = Depends(get\_db), user:models.User = Depends(auth2.get\_current\_user)):

    print(user.email)

    # 使用SQLAlchemy查询所有文章

    posts = db.query(models.Post).all()

return posts  # 返回文章列表

main.py文件

from fastapi import FastAPI

app = FastAPI()

app.include\_router(post.router)

app.include\_router(user.router)

app.include\_router(auth.router)

配置环境变量

from pydantic\_settings import BaseSettings #引入模块

class Setting(BaseSettings): #继承

    DATABASE\_HOST:str

    DATABASE\_PORT:str

    DATABASE\_PASSWORD:str

    DATABASE\_NAME:str

    DATABASE\_USERNAME:str

    SECRET\_KEY:str

    ALGORITHM:str

    ACCESS\_TOKEN\_EXPIRE\_MINUTES:str

    class Config:

        env\_file = '.env'

settings = Setting()

继承BaseSettings的类属性，会查找本地计算机同名的环境变量，在创建实例的时候生成相同的实例属性。

如果配置了class Config: env\_file = '.env'，会把'.env'文件里的变量值以字符串传进来。

.env文件

DATABASE\_HOST=192.168.75.20

DATABASE\_PORT=5432

DATABASE\_PASSWORD=p-0p-0p-0

DATABASE\_NAME=fastapi

DATABASE\_USERNAME=postgres

SECRET\_KEY= 02435013be0c2268bb9bab21f8e89e8819bf259b591b2df106ce921a523e6df9

ALGORITHM= HS256

ACCESS\_TOKEN\_EXPIRE\_MINUTES= 30