

前言

你好，首先感谢您下载了这本pdf。因为图床不稳定的原因，所以特意转换成了pdf方便各位读者的阅读。这是我之前在公众号里写的FastAPI相关文章的汇总。该系列目前总共有6篇文章，能够帮助大家快速的入门FastAPI这个非常好用的框架。这份pdf推荐阅读时间是一个小时，所以也许你能在一个小时内入门这个框架。如果你想要深入学习这个框架，我是非常推荐你前往FastAPI官网进行学习。

OK，废话不多说，赶紧开始学起来！

（原创：Python进击者）

敲门砖：写一个demo

如果让我用一个句话来描述fastapi，我会说这是**Python里面最好的api框架！**

不比go、node.js差！

以往我们使用Python来写后端，基本上使用的是Django和Flask。

但是现在不一样了，fastapi不仅仅高效率而且还很适合产品级的开发。

本系列最后会带着大家做一个完整的产品级项目，所以大家一定要记得星标公众号！

最直观的了解一个框架，就是先写一个简单的demo。

在写demo之前，我们需要安装一下fastapi以及ASGI（ASGI是WSGI的升级版，支持异步调用）

```
pip install fastapi
pip install uvicorn
```

安装完毕之后，创建一个main.py，然后粘贴我们下面的代码跑一下：

```
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
def index():
    return {"msg": "hello world"}

@app.get("/items/{item_id}")
def read_item(item_id: int, keyword: str):
    return {"item_id": item_id, "keyword": keyword}
```

如何运行？

如何运行呢？

直接 python main.py ？

不是的，上面我们已经安装了uvicorn，我们就直接用它来运行

```
$ uvicorn main:app --reload --port 8000
$ curl http://127.0.0.1:8000
$ curl http://127.0.0.1:8000/items/1?keyword=book
```

这里估计也得跟大家介绍一下上面几个命令行的意思。

首先第一个 `uvicorn main:app --reload --port 8000`

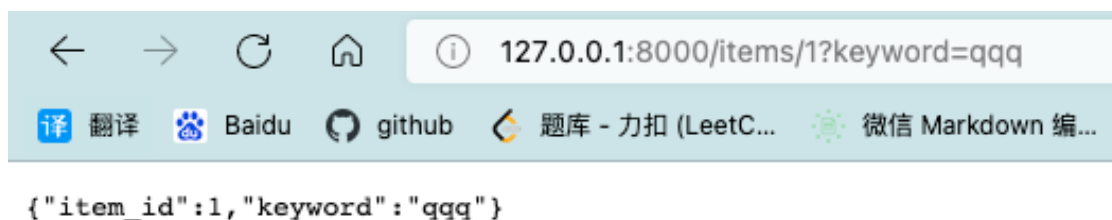
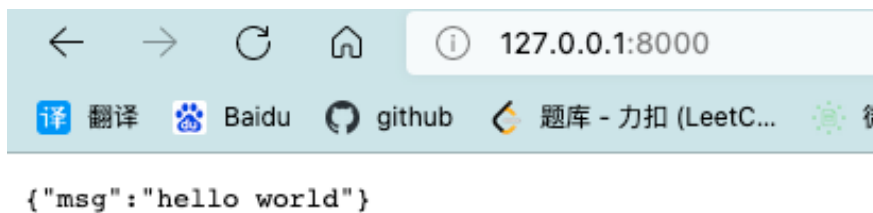
其中的main 指的是我们的文件名，app指的是我们在代码中FastAPI()这个类赋予的变量名，例如在上面`app = FastAPI()`，所以这里就是app。--reload是热部署的意思，热部署就是指我们改动代码服务器会自动更新代码文件然后重新运行。--port就不用多说了。

其次curl，你可以理解为一个请求web资源的命令行。

最后，运行完成，可以看到我们输出了一个json类型字符串

```
MacBook-Pro:~ kuls$ curl http://127.0.0.1:8000
{"msg":"hello world"}MacBook-Pro:~ kuls$
MacBook-Pro:~ kuls$ curl http://127.0.0.1:8000/items/1?keyword=book
{"item_id":1,"keyword":"book"}MacBook-Pro:~ kuls$
```

当然你也可以在浏览器中访问。



openAPI支持

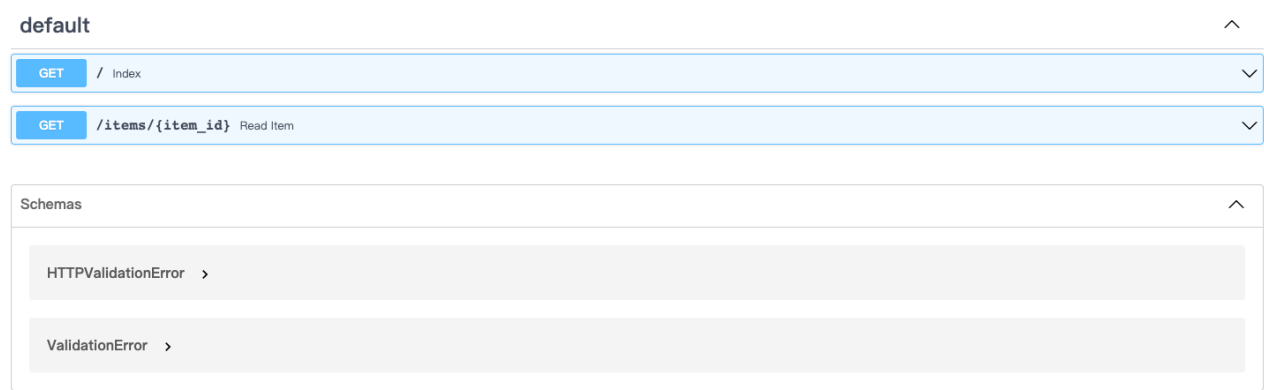
在之前介绍fastapi的文章中也讲了，fastapi是一个符合OpenAPI 和 JSON Schema的框架。

所以fastapi也支持了API文档的自动生成，这一点对程序员来说真的是非常非常的舒服。

如何访问呢？

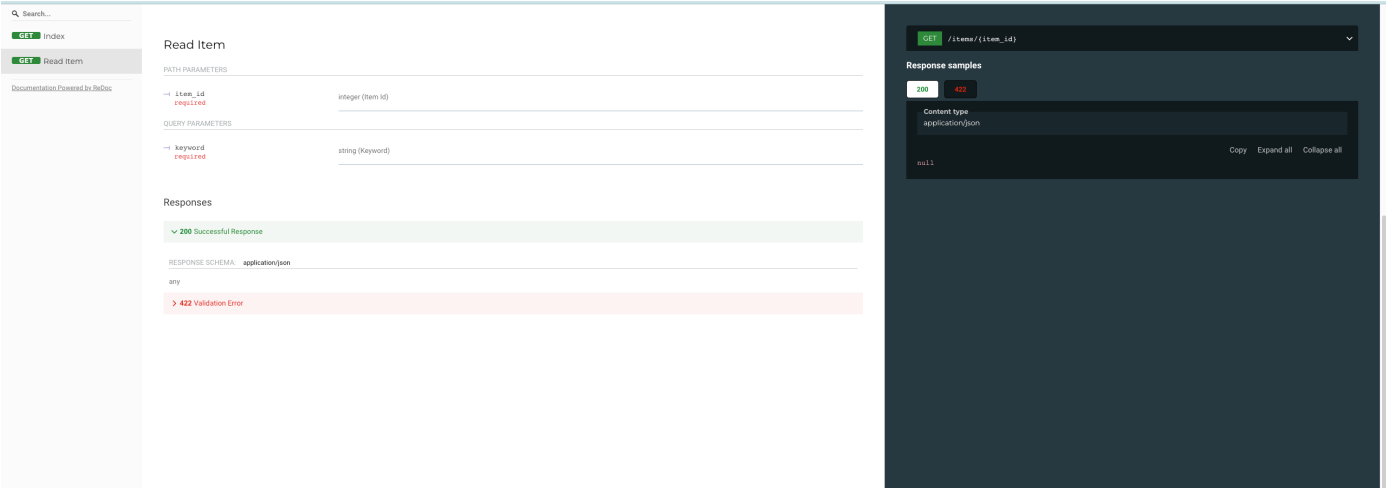
<http://127.0.0.1:8000/docs>

在后面加个docs就行了



<http://127.0.0.1:8000/redoc>

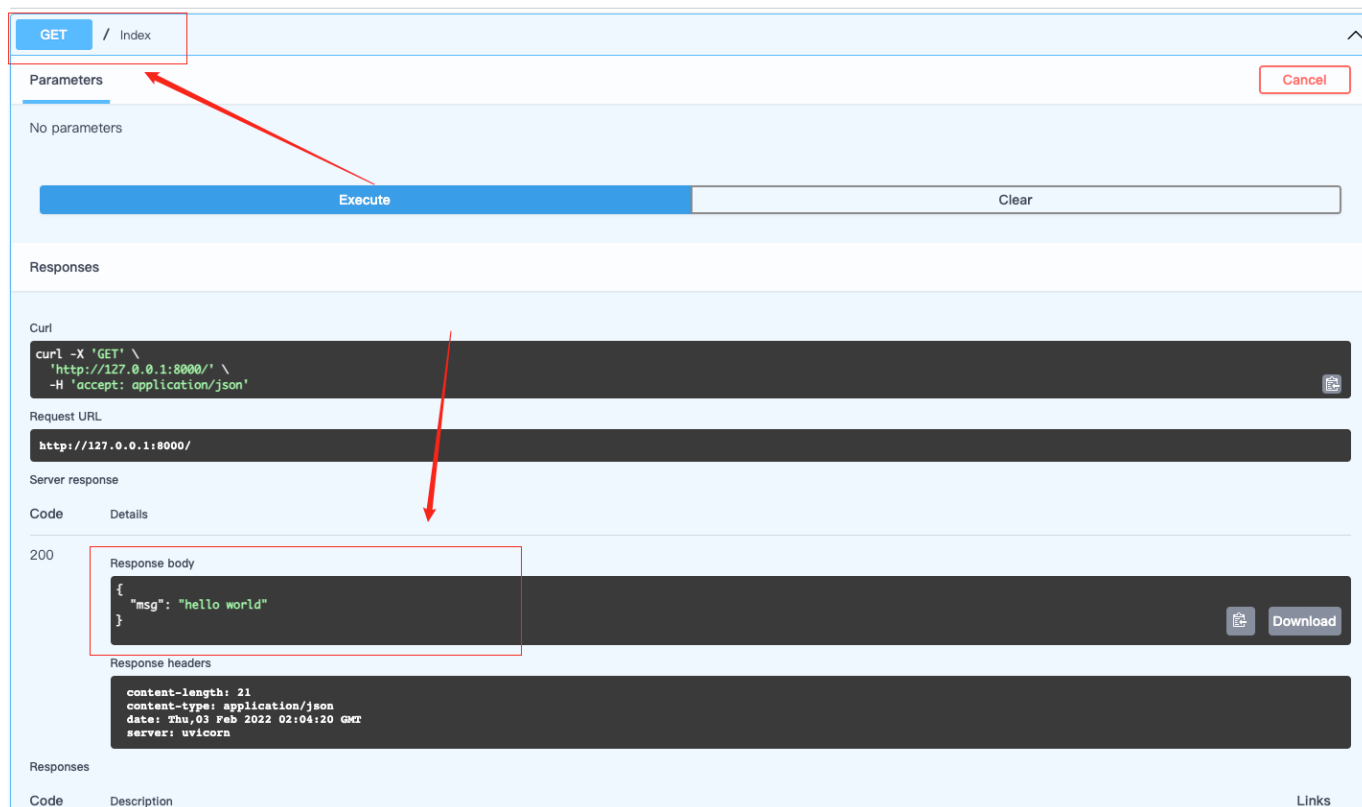
这样也能访问，只不过是换了一个api文档



可能很多读者不知道swagger是个啥，其实你可以理解为它是一个api管理和调试的工具箱。

我们编写的api接口都能够通过它来进行管理和调试。

例如下面，我可以直接调用之前写的/ 接口。



swagger的整体UI设计也是非常不错的!

新玩法的引入

有一些细心的朋友可能发现了上面我所写的代码，跟大家平时写的有一点点不同：

```
@app.get("/items/{item_id}")
def read_item(item_id: int, keyword: str):
    return {"item_id": item_id, "keyword": keyword}
```

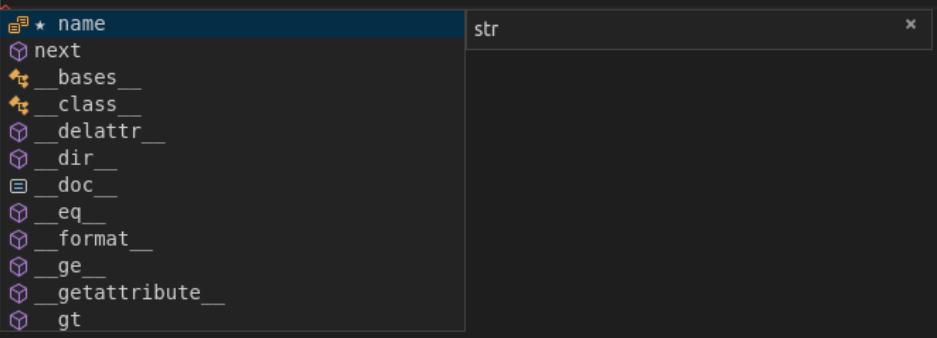
主要是在这一行 `def read_item(item_id: int, keyword: str):`

因为fastapi是一个比较年轻的api框架，创立于2018年12月，距今不到两年。

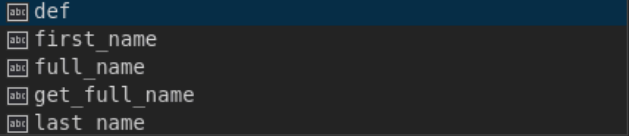
所以也引入了Python3.6+中比较新的 `类型提示` 功能。其实就是使用了pydantic这个库实现的。

这里我引入官网的几张图来给大家直观的看看

```
1 class Person:
2     def __init__(self, name: str):
3         self.name = name
4
5
6 def get_person_name(one_person: Person):
7     return one_person.name
8
```



```
1 def get_full_name(first_name, last_name):
2     full_name = first_name + last_name
```



我们可以通过声明变量的类型，编辑器和一些工具能给你提供更好的支持。

K哥认为引入这个的目的也是为了fastapi中的fast，让我们的开发更加快！

在官网中也说了高效编码：提高功能开发速度约 200% 至 300%。

关于fastapi的很多特性，我们都会从后面的文章中详细解答，例如它的安全和认证等等。

非常高兴你能够看到这里，其实写系列文章没有写其他文章流量那么大，但是K哥仍然还是想系统的来给大家讲讲某门技术，之前也写过Django和Flask系列文章。所以这里也希望各位看完的朋友能够帮忙转发转发！

初露头角：如何用最简单的方式理解一个FastAPI程序？

大家好，我是Kuls。

下面个程序就是我们上篇文章中所写的：

```
from fastapi import FastAPI
app = FastAPI()

@app.get("/items/{item_id}")
def read_item(item_id: int, keyword: str):
    return {"item_id": item_id, "keyword": keyword}
```

本文的目的就是通过这个程序来学习FastAPI中相关的知识点，下面我们一行一行来进行剖析。

1、`from fastapi import FastAPI`

其实这行代码大部分学过Python的读者都知道，但是因为有一些是没有Python基础的读者也在学习，所以这里我也简单说下。

这行代码的意思就是从 fastapi 库中导入一个叫做FastAPI的类，我们整个FastAPI程序都是基于这个类来实现的。

2、`app = FastAPI()`

这行也比较好理解，就是将FastAPI类进行实例化，并且将值赋给app。这里的app当然不是固定的，我们也可以命名为myapp，如果我们是这样写的：

```
myapp = FastAPI()
```

那么我们在运行的时候就得输入这样的命令行：

```
如果app没修过
$ uvicorn main:app --reload --port 8000
如果app修改成myapp
$ uvicorn main:myapp --reload --port 8000
```

其实前面的main代表的就是我们的文件名，如果我们将main.py修改成demo.py，那么我们将会这样执行

```
$ uvicorn demo:myapp --reload --port 8000
```

如果我们还是按照之前的命令行运行就会报如下错误

```
(venv) MacBook-Pro:demo1 kuls$ uvicorn main:app --reload
INFO:      Will watch for changes in these directories: ['/Users/kuls/Desktop/pythondemo/fastApiProject/demo1']
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:      Started reloader process [42710] using watchgod
ERROR:      Error loading ASGI app. Could not import module "main".
```

所以uvicorn也是非常灵活的，都是可配置化的。

3、`@app.get("/items/{item_id}")`

熟悉Python的朋友应该知道@代表的是装饰器的意思，如果你不懂装饰器，可以先去谷歌一下，其实你也可以理解它就是一个函数。

在这里我们引用app中的get方法，其含义就是发起一个 GET 请求，这里我们也可以换成post、put、delete....

```
m get(self, path, *, response_model=None, status_code=None, tags=None)
m put(self, path, *, response_model=None, status_code=None, tags=None)
m post(self, path, *, response_model=None, status_code=None, tags=None)
m delete(self, path, *, response_model=None, status_code=None, tags=None)
m options(self, path, *, response_model=None, status_code=None, tags=None)
m head(self, path, *, response_model=None, status_code=None, tags=None)
m patch(self, path, *, response_model=None, status_code=None, tags=None)
m trace(self, path, *, response_model=None, status_code=None, tags=None)
```

具体我们可以看相应的源码，对应的其实就是那几个很常见的网络请求操作。

在get函数内的参数，我们称之为路径参数，就是我们访问的 <http://127.0.0.1:8000/items/1?keyword=book>

其中{item_id}，代表的就是上面的 1，它会将 item_id 的值作为参数 item_id 传递给我们下面所写的函数。

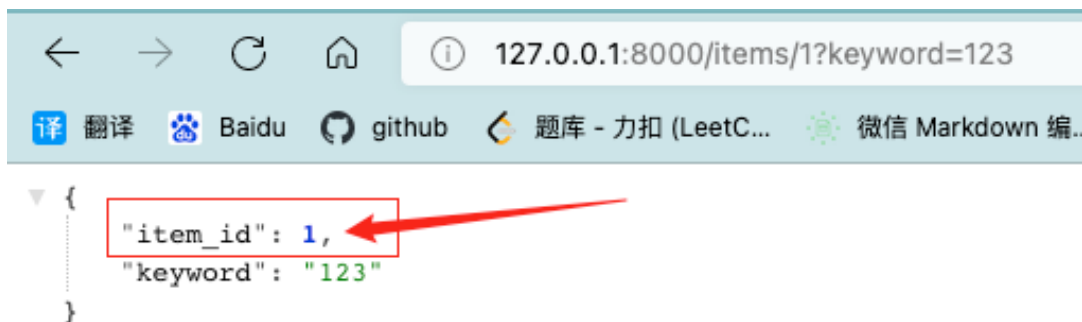
4、`def read_item(item_id: int, keyword: str):`

这个函数就是路径操作函数，我们来一步步拆解它。

首先def 函数名 这个大家没什么争议，主要是后面这一部分。

item_id: int，大家可能会发现这个与我们上面所讲的 @app.get("/items/{item_id}") 中定义的 {item_id} 是一样的名字，我们的路径操作函数就是通过这样来传输用户所输入的路径的。

其中int 就是定义这个传入的数据类型，这里给大家演示一下。



如果此时我们传输进去一个字符串，那会发生什么？



我们可以看到fastapi给我反馈出了错误的原因以及位置

```
"loc": [  
  "path",  
  "item_id"  
],
```

loc就代表着我们在path路径中的item_id出现了错误，那么具体是什么错呢？

```
"msg": "value is not a valid integer",  
"type": "type_error.integer"
```

我们传入的值不是一个int，所以在这里也可以看出fastapi的错误机制其实非常不错的，能够给我们一个比较清晰直观的错误提示。

聊完了前面那个参数，那我们来聊聊后面这个 `keyword: str`，这一部分跟前一个参数不同，前一个参数在我们的路径参数中有所体现，但是这个只出现在了函数的参数中。

FastAPI 其实非常聪明，能够分辨出参数 `item_id` 是路径参数而 `keyword` 不是，因此它会判断 `keyword` 是一个查询参数。

```
5、 return {"item_id": item_id, "keyword": keyword}
```

我们最后来看看这行代码，这行代码其实就是返回一个json字符串。

我们在以后的项目当中，这一部分就是我们api返回的结果。

好了，到这里相信大家能够对这段代码有所理解了，这里K哥给大家留下一个作业。

如果我们想要制作一个博客，我们对于博客的首页、某一个博客内容页面、关于我的页面、搜索功能... 该如何设置路径参数呢？该返回哪些内容呢？

作业非常简单，大家可以尝试写一写！

非常高兴你能够看到这里，其实写系列文章没有写其他文章流量那么大，但是K哥仍然还是想系统的来给大家讲讲某门技术，之前也写过Django和Flask系列文章。所以这里也希望各位看完的朋友能够帮忙转发转发！

小试牛刀：FastAPI快速搭建一个博客系统

大家好，我是Kuls。

下面主要讲的是FastAPI快速搭建一个博客系统。

这里可能有些小伙伴就懵了，我啥都不会的，怎么就让我来写个博客？

别慌，K哥这里是想通过搭建一个博客系统的api框架来给大家具体讲讲FastAPI里面的相关知识。

我们将会实现博客的几个功能：

- 博客首页
- 单篇博客页面
- 某篇博客评论内容
- 获取未发布状态的博文
- 发布博客

这里大家需要注意，我们编写的只是博客的后端api的大致框架，如果有数据库等操作的内容，我们会放在后面说的。

一般我们写项目，首先可以先把简单的部分给写出来，这样我们整体项目的雏形也就可以慢慢形成。

在上述的几个功能里，博客首页其实是最简单的。

当然在我们编写代码之前，我们需要去创建虚拟环境、创建main.py、安装相关的依赖。

不管我们要写啥功能，导入fastapi，创建fastapi实例的操作是必备的：

```
from fastapi import FastAPI
app = FastAPI()
```

博客首页

博客首页其实是最简单就可以实现的。

```
# 博客首页
@app.get('/blog')
def index():
    return {'data': '我是博客首页'}
```

直接一个get请求，路径我们设置为 /blog，return 返回的内容我们先随便模拟一些数据，后续我们会继续完善。

实现的效果：



但是大家有没有发现一个问题。

因为我们有一个功能是 获取未发布状态的博文，这说明我们每一篇博文是有属于它自己的状态的，也就是 发布 和 未发布。

所以这里我们需要有一个参数来代表我们是想要获取发布的还是未发布的。

为了更加完善，我们也可以加入一些limit、sort等参数

http://127.0.0.1:8000/blog?published=true&&limit=10&&sort=publish_at

最终我们是通过上述的访问得到相应的博文列表。

代码实现：

```
from fastapi import FastAPI
from typing import Optional
app = FastAPI()

# 博客首页
@app.get('/blog')
def index(limit: int = 10, published: bool = True, sort: Optional[str] = None):
    return {'data': f'我是博客首页，显示{limit}篇内容，并且发布状态为{published}，排序顺序是根据{sort}字段'}
```

def index(limit: int = 10, published: bool = True, sort: Optional[str] = None):

这行代码中 limit: int = 10，其实就是将limit的默认值设为10，published同理。

sort: Optional[str] = None，其中的Optional是我们从typing第三方库中导入的，意思就是可选的，也就意味着sort这个参数我们也可以不设置。

<http://127.0.0.1:8000/blog?published=true&&limit=10> 这样访问也是正确的。

最后，实现的效果：



单篇博客页面

单篇博客页面这里就涉及到了fastapi中的路径参数。

当你去观察很多的博客时，你会发现大多数都是通过 `/blog/数字` 这种形式来定义的。

所以在这里我们也给每一篇博客一个id，然后直接通过 <http://127.0.0.1:8000/blog/1> 就能够访问到id为1的博客。

```
@app.get('/blog/{id}')
def showblog(id: int):
    return {'data': f'这是id为 {id} 的博文'}
```

这里可以看到，我们设置了一个路径参数 {id}，并且我们把它设置成了int类型。

实现的效果：



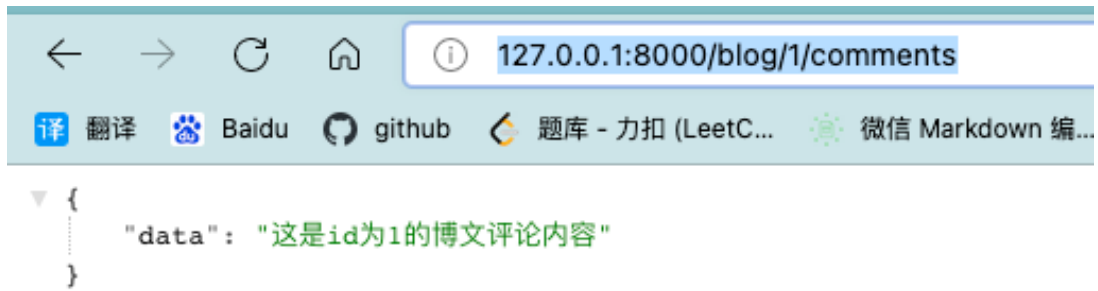
某篇博客评论内容

这个其实也很简单，跟我们单篇博客页面其实差不很多。

```
@app.get('/blog/{id}/comments')
def comments(id: int):
    return {'data': f'这是id为{id}的博文评论内容'}
```

其中主要的区别就在于路径参数，我们后面加了一个comments来代表评论。

最后，访问的效果：



获取未发布状态的博文

想要拿到未发布状态的博文，其实逻辑很简单，我们在具体逻辑中判断一下就行了，这里会在后面文章中详细写。

具体函数：

```
@app.get('/blog/unpublished')
def unpublished():
    return {'data': '这里是没有发布的博文列表'}
```

当我们写完这个函数之后

```
1 from fastapi import FastAPI
2
3 from typing import Optional
4 app = FastAPI()
5
6
7 # 博客首页
8 @app.get('/blog')
9 def index(limit: int = 10, published: bool = True, sort: Optional[str] = None):
10     return {'data': f'我是博客首页，显示{limit}篇内容，并且发布状态为{published}，排序顺序是根据{sort}字段'}
11
12
13 @app.get('/blog/{id}')
14 def showblog(id: int):
15     return {'data': f'这是id为 {id} 的博文'}
16
17 @app.get('/blog/{id}/comments')
18 def comments(id: int):
19     return {'data': f'这是id为{id}的博文评论内容'}
20
21 @app.get('/blog/unpublished')
22 def unpublished():
23     return {'data': '这里是没有发布的博文列表'}
```

去访问 <http://127.0.0.1:8000/blog/unpublished>

竟然报错了！？

为啥，我们具体看一下报错信息



是因为我们在上面写了一个访问单篇博文的功能

```
@app.get('/blog/{id}')
def showblog(id: int):
    return {'data': f'这是id为 {id} 的博文'}
```

大家可以发现

@app.get('/blog/{id}')

@app.get('/blog/unpublished')

这两个路由非常的相似，只是后面一部分不同，可能有些读者就发现不对了，报错信息说我们传入的值不是一个合法的整型。

这说明fastapi它每次去查看路径时，是按照代码的从上到下的顺序来的。

所以它首先访问的是@app.get('/blog/{id}')的路由，而不是@app.get('/blog/unpublished')。

我们特别需要注意这一点！这在fastapi的编写当中非常重要！

那怎么解决这个问题？

```

from fastapi import FastAPI
from typing import Optional
app = FastAPI()

# 博客首页
@app.get('/blog')
def index(limit: int = 10, published: bool = True, sort: Optional[str] = None):
    return {'data': f'我是博客首页, 显示{limit}篇内容, 并且发布状态为{published}, 排序顺序是根据{sort}字段'}

@app.get('/blog/unpublished')
def unpublished():
    return {'data': '这里是没有发布的博文列表'}

@app.get('/blog/{id}')
def showblog(id: int):
    return {'data': f'这是id为 {id} 的博文'}

@app.get('/blog/{id}/comments')
def comments(id: int):
    return {'data': f'这是id为{id}的博文评论内容'}

```

放在上面

我们直接将@app.get('/blog/unpublished')路径下的函数放在上面, 这样fastapi首先就会访问到它, 同时也不会影响@app.get('/blog/{id}')。

这个问题就完美的解决了!



发布博客

在之前的功能中, 我们使用的都是fastapi中的get请求, 在这个功能里, 我们将会使用post来实现。

在实现发布博客前, 我们得首先了解一下 **模型** 这个概念。

学过flask或者Django的读者应该知道这个玩意, 没学过也没关系, 你可以理解为就是一个class类, 继承了一个基类BaseModel。

其中的BaseModel, 我们是从pydantic第三方库导入进来的。

下面就是我们实现的一个简单的Blog的基类

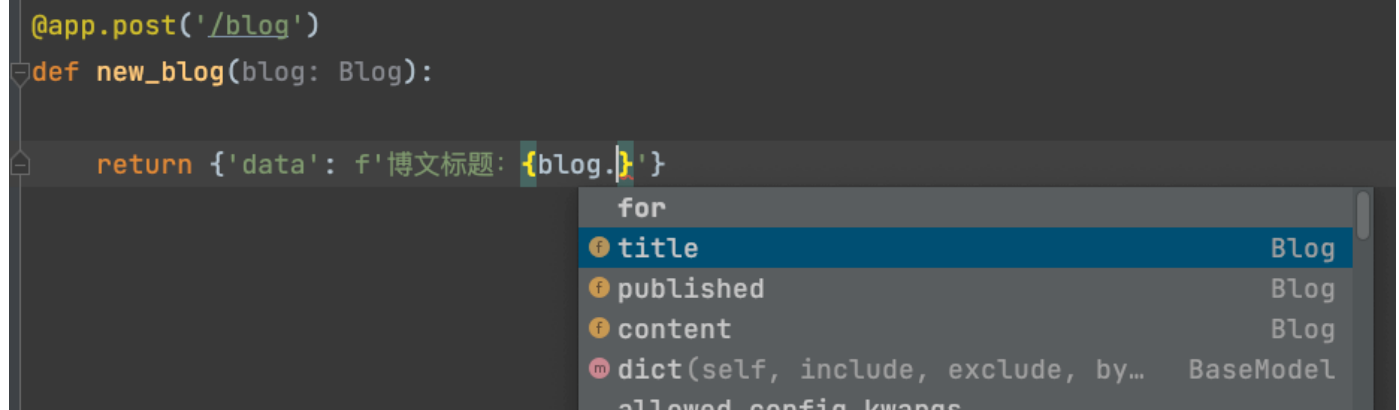
```
class Blog(BaseModel):  
    title: str  
    content: str  
    published: Optional[bool]
```

title就是我们博文的标题，content就是内容，published表示的状态。

下面我们把基本的框架写好，看看能不能获取到post提交过来的值。

```
@app.post('/blog')  
def new_blog(blog: Blog):  
    return {'data': f'博文标题: {blog.title}, 博文内容: {blog.content}, 博文发表状态: {blog.published}'}
```

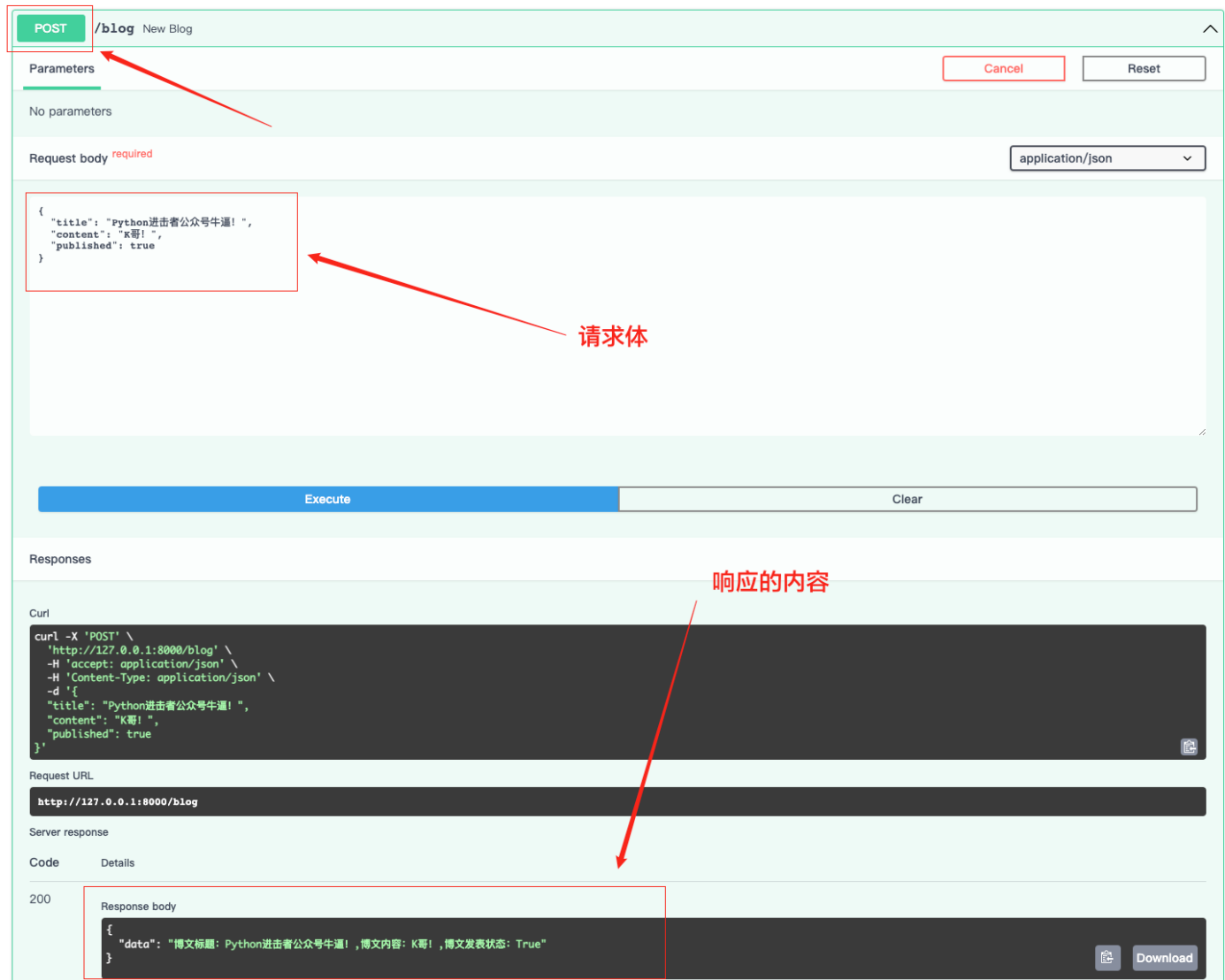
大家可以看到@app.post('/blog')，此处是post方法了。并且在函数的参数中，我们使用的是上面建的class Blog在编写的时候，fastapi的特性也体现出来了，IDE给了我们相应的提示，这也加快了我们的开发速度。



怎么去测试post呢？

你可能忘记了fastapi中的swagger！

我们下面用swagger来测试一下这个接口



大家可以看到成功的返回了！

总结

好了，前面两篇文章没有写过总结，今天还是给大家写一个总结吧。

这篇文章的目的其实就是教大家熟练掌握路径参数、查询参数、模型、请求方法这几个方面，整篇文章中也给大家讲述了一些细节问题。K哥还是希望大家能够认真阅读文章并且自己动手写一下，代码我也会放在文末。

非常高兴你能够看到这里，其实写系列文章没有写其他文章流量那么大，但是K哥仍然还是想系统的来给大家讲讲某门技术，之前也写过Django和Flask系列文章。所以这里也希望各位看完的朋友能够帮忙转发转发！

本系列代码仓库：<https://github.com/hellokuls/fastapi/tree/master>

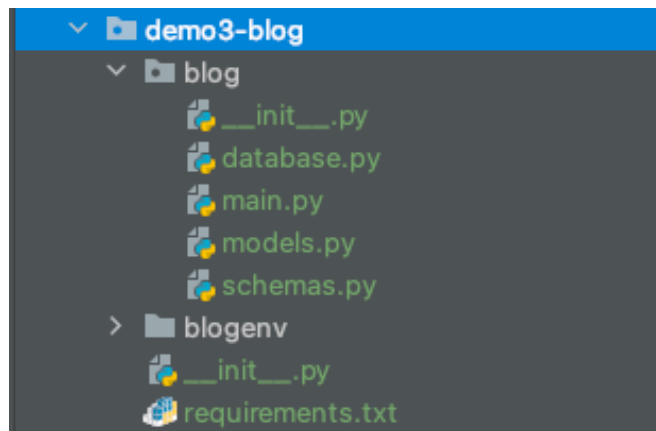
更进一步：FastAPI如何优雅的连接数据库？

大家好，我是Kuls。

在前面一篇文章中，我们有去写一个简单的博客框架，对它的路径、查询参数及路径参数函数等进行了学习。

今天我们来学习FastAPI的数据库连接，当然我们不仅仅只是为了学习这个知识点而学习。

今天K哥也来给大家说说如何去优雅的写一个FastAPI项目。



这是我今天所要讲解所编写项目的文件格式。

大家可以看到我们会在里面新建一个blog库来存放我们编写博客相关的操作，而不是像前面我们所写的，直接写一个main.py就行了。

那样对于我们编写一个项目是不太好的。

我们编写一个项目可以先创建一个虚拟环境，如果你还不懂虚拟环境是什么，可以去看下我的这篇文章 [虚拟环境真的太重要了，很多人还不知道！](#)

```
$ cd demo3-blog/
$ ls
__init__.py          blog                  requirements.txt

$ python -m venv blogenv
$ ls
__init__.py          blog                  blogenv
requirements.txt

$ source blogenv/bin/activate
```

随后创建一个requeriment.txt，里面填写这个项目要使用的第三方库。

```
fastapi
uvicorn
sqlalchemy
pymysql
```

执行

```
pip install -r requeriment.txt
```

好了，上面其实算是题外话。

今天我们的主角其实是SQLAlchemy，可能之前学过Django或Flask的同学应该接触过SQLAlchemy。

我们看下官网的解释：SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.

简答理解，它是一个sql工具箱，是一个ORM框架。

ORM：对象关系映射，你可以简单理解为 Python 中的一个类映射一张数据表。

其实关于SQLAlchemy，里面有很多的知识，我也把它的详细的基础使用方法链接给大家。

官方文档：<https://docs.sqlalchemy.org/en/14/orm/tutorial.html>

本篇文章中我们也会学习SQLAlchemy。

database.py

还记得我们创建的database.py文件吗？我们将会在这里面编写数据库相关的内容：

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

SQLALCHEMY_DATABASE_URL = 'mysql+pymysql://user:password@localhost/fastapi'
engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False)
Base = declarative_base()
```

这里我们一行一行来进行分析

```
engine = create_engine(SQLALCHEMY_DATABASE_URL)'
```

这一行创建了我们的数据库引擎，我们就是通过这个引擎来创建表等各种操作的。其中的参数就是我们数据库的连接url，fastapi支持的数据库非常的多。

这里提醒一下，如果我们使用的是sqlite数据库，我们需要在create_engine的参数中新增connect_args={"check_same_thread": False}

下面也是官网给出的支持的数据库，这些我们都可以通过sqlalchemy来进行连接。

- PostgreSQL
- MySQL
- SQLite
- Oracle
- Microsoft SQL Server, etc.

关于具体数据库的url是啥，这里我给出官网，大家可以进行查阅

<https://docs.sqlalchemy.org/en/14/core/engines.html>

```
SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False)
```

这里我们创建的是SessionLocal，后续我们用到会进行讲解。

```
Base = declarative_base()
```

通过declarative_base()方法生成的类，该类是我们后面要用到的ORM 模型所要继承的父类。

models.py

既然数据库相关的配置都弄好了，接下来我们编写一个模型。

其实这个模型在我们上一篇文章中就已经编写过了。

但是还是有一些不同

```
from sqlalchemy import Column, Integer, String, Boolean
from .database import Base

class Blog(Base):
    __tablename__ = 'blog'

    id = Column(Integer, primary_key=True, index=True) # 设置主键和索引
    title = Column(String(32))
    content = Column(String(32))
    pulished = Column(Boolean)
```

大家可以发现，其实非常的直观和简单，无非就是创建一个类，里面有一些字段变量。

其中的**tablename**表示的是我们待会创建数据表的名称。

schemas.py

这个其实很简单，就是将我们在main.py中所编写的

```
from pydantic import BaseModel
from typing import Optional

class Blog(BaseModel):
    title: str
    content: str
    published: Optional[bool]
```

这部分内容搬到了一个新的文件当中，方便我们统一管理。

main.py

其实main.py跟我们上一篇文章的中的main.py差不多，只是多了几行代码：

```
from fastapi import FastAPI
from .database import engine
from . import schemas, models
from typing import Optional
app = FastAPI()

models.Base.metadata.create_all(engine)

# 博客首页
@app.get('/blog')
def index(limit: int = 10, published: bool = True, sort: Optional[str] = None):
    return {'data': f'我是博客首页，显示{limit}篇内容，并且发布状态为{published}，排序顺序是根据{sort}字段'}

@app.get('/blog/unpublished')
def unpublished():
    return {'data': '这里是没有发布的博文列表'}

@app.get('/blog/{id}')
def showblog(id: int):
    return {'data': f'这是id为 {id} 的博文'}

@app.get('/blog/{id}/comments')
def comments(id: int):
    return {'data': f'这是id为{id}的博文评论内容'}

@app.post('/blog')
def new_blog(blog: schemas.Blog):
    return {'data': f'博文标题: {blog.title},博文内容: {blog.content},博文发表状态: {blog.published}'}
```

其中的create_all()方法就帮助我们创建了数据表

运行

到了这里基本就大功告成。

我们直接运行

```
INFO: Started server process [12513]
INFO: Waiting for application startup.
INFO: Application startup complete.
```



查看数据库，发现我们的数据表已经成功创建。

```
[mysql> show tables
+-----+
| Tables_in_fastapi |
+-----+
| blog                |
+-----+
1 row in set (0.00 sec)
```

总结

好了，今天主要讲了数据库相关的操作以及编写项目时的注意事项。
数据库具体的读写删操作，我们会在后面继续写，也会基于这个blog来写。
因为公众号改版，所以大家一定要记得星标公众号。
整个系列的代码我都放在了github中，大家可以访问下面链接：

<https://github.com/hellokuls/fastapi/tree/master>

玩点骚的：如何操作数据库？

大家好，我是Kuls。
今天我们要讲的是如何去操作数据库，我们还是通过我们的博客案例来给大家讲解。
在上篇文章中，我们已经将博客项目的大体框架都写好了，包括数据库引擎相关的内容。
那么我们该如何实际去操作数据库呢？
也就是如何对数据库进行 `增删改查`。
其实也非常的简单。

SessionLocal

还记得我们在database.py中创建的SessionLocal吗？

```
SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False)
```

我们的操作都是基于它来实现的！

首先，我们从database.py中导入它

```
from .database import SessionLocal

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

然后编写函数get_db()来用于获取sessionLocal以及及时关闭连接。

大家还记得我们之前所编写的新增_blog()函数吗？也就是新增一篇博客的函数。

今天我们将它与数据库进行具体实现。

新增博客

```
from fastapi import Depends

@app.post('/blog')
def new_blog(blog: schemas.Blog, db: Session = Depends(get_db)):
    newBlog = models.Blog(title=blog.title, content=blog.content,
published=blog.published)
    db.add(newBlog)
    db.commit()
    db.refresh(newBlog)
    return newBlog
```

这个地方可能有些人会困惑Depends()是个啥玩意？

其实你可以简单理解成依赖注入，我们点进去看源码

```
def Depends( # noqa: N802
    dependency: Optional[Callable[..., Any]] = None, *, use_cache: bool = True
) -> Any:
    return params.Depends(dependency=dependency, use_cache=use_cache)
```

我们可以发现Depends()里的参数有两个dependency、use_cache，我们主要关注的就是dependency，英文含义就是依赖，可以发现我们可以传入函数以及类。

也就是说这个参数它会依赖于这个函数或者类来生成。

我们这里的db也就是依赖于get_db()这个函数来生成的。

编写好函数，我们前往swagger进行测试

<http://127.0.0.1:8000/docs>

Request body:

```
{
  "title": "文章标题",
  "content": "文章内容",
  "published": true
}
```

CodeDetails

200

Response body

```
{
  "published": true,
  "id": 2,
  "title": "文章标题",
  "content": "文章内容"
}
```

Download

Response headers

```
content-length: 73
content-type: application/json
date: Thu, 17 Feb 2022 06:41:06 GMT
server: uvicorn
```

Objectsblog@fastapi (root)

id	title	content	published
1	文章标题	文章内容	1
2	文章标题	文章内容	1

可以看到插入成功了！

但是这里也有一个小小的问题，我们可以看到swagger当中给我们返回的响应码是200，熟悉响应码的朋友可能知道，如果是新建xxx，响应码应该返回201，而不是200。

那么fastapi能不能实现呢？

其实也是非常的简单，我们只需要在@app.post('/blog')，加上一个参数@app.post('/blog', status_code=201)

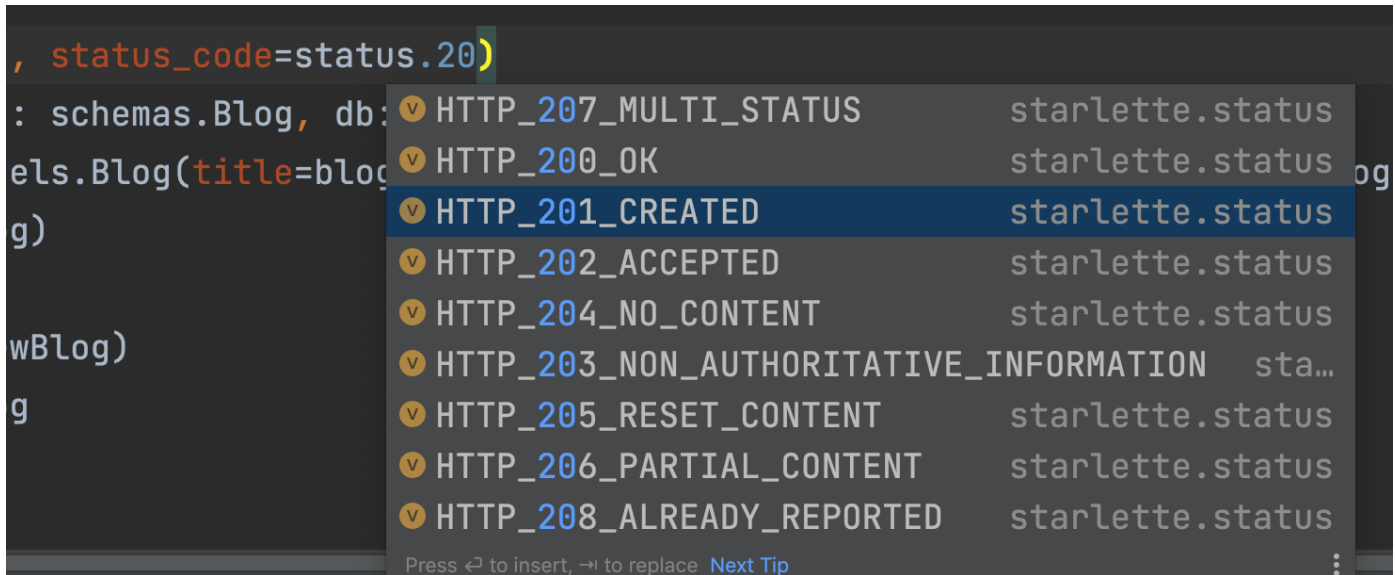


可以看到，返回的是201。

如果你想要更加详细了解响应码，可以前往官网查看

<https://docs.python.org/3/library/http.html#http.HTTPStatus>

fastapi也给我们提供了各种响应码的选择，只需要导入fastapi.status库



获取所有博客

聊完了新建博客，下面我们聊聊如何去获取所有的博客。

这个函数我们在之前也有设计过，这里我给你打印出来


```
# 博客首页
@app.get('/blog')
def index(limit: int = 10, published: bool = True, sort: Optional[str] = None):
    return {'data': f'我是博客首页，显示{limit}篇内容，并且发布状态为{published}，排序顺序是根据{sort}字段'}
```

这里我们也需要加入db的参数：

```
# 博客首页
@app.get('/blog')
def index(limit: int = 10, published: bool = True, sort: Optional[str] = None, db: Session = Depends(get_db)):
    blogs = db.query(models.Blog).all()
    return blogs
```

通过db.query(models.Blog).all()，就能够查询到所有的结果，我们去swagger里面进行测试：

Response body

```
[
  {
    "published": true,
    "id": 1,
    "title": "文章标题",
    "content": "文章内容"
  },
  {
    "published": true,
    "id": 2,
    "title": "文章标题",
    "content": "文章内容"
  }
]
```

接下来我们继续改造之前的函数，还记得我们是如何获取单篇文章内容的吗？

获取单篇博客

这里我已经把它改造好了：

```
@app.get('/blog/{id}')
def showblog(id: int, db: Session = Depends(get_db)):
    blog = db.query(models.Blog).filter(models.Blog.id == id).first()
    return blog
```

通过filter函数，我们可以筛选出相应要求的数据，first()则代表我们只想获取到符合要求的第一条数据。显然这里是只有一条的。

同样，我们也去测试一下。

Name

Description

id * required
integer
(path)

1

Execute

Responses

Curl

```
curl -X 'GET' \
'http://127.0.0.1:8000/blog/1' \
-H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/blog/1
```

Server response

Code	Details
200	<div><div>Response body</div><pre>{ "published": true, "id": 1, "title": "文章标题", "content": "文章内容" }</pre></div>

非常的简单！

删除某篇博客

相信看到这里，大家都知道如何去删除一篇博客了。

```
@app.delete('/blog/{id}', status_code=status.HTTP_204_NO_CONTENT)
def delete(id: int, db: Session = Depends(get_db)):
    db.query(models.Blog).filter(models.Blog.id ==
id).delete(synchronize_session=False)
    db.commit() # 别忘了哦
    return {"msg": "已经删除"}
```

首先通过id查找到对应的博客，然后调用delete方法将其删除。

DELETE /blog/{id} Delete

Parameters

Name	Description
id * required integer (path)	<input type="text" value="1"/>

Execute

	id	title	content	published	
	2	文章标题	文章内容	1	
	3	第三篇文章	文章	1	

前往swagger，发现id=1已经被成功删除了。

更新博客

我们首先新建一条博客

id	title	content	published
4	文章1	文章1	1

如何更新一条博客呢？

```
@app.put('/blog/{id}')
def update(id: int, blog: schemas.Blog, db: Session = Depends(get_db)):
    db.query(models.Blog).filter(models.Blog.id == id).update(blog.dict())
    db.commit()
    return {"msg": "成功更新!"}
```

我们用到了put方法。特别需要注意的是update(blog.dict()), 我们需要把blog对象转化为dict, 不然会报错的哦。

Name	Description
id * required integer (path)	<input type="text" value="4"/>
Request body required	
<pre>{ "title": "文章2", "content": "string", "published": true }</pre>	
<div>Execute</div> <div>Clear</div>	

执行后成功！

id	title	content	published
4	文章2	string	1

总结

今天我们实现了以下的基本数据库操作

GET	/blog	Index
POST	/blog	New Blog
GET	/blog/{id}	Showblog
PUT	/blog/{id}	Update
DELETE	/blog/{id}	Delete

大家最好要动手操作一下！这个非常重要。

新手毕业：FastAPI你不得不知道的响应模型

你好，我是Kuls。

我们主要来聊一聊FastAPI里的响应模型。

可能很多读者朋友不太清楚响应模型是啥，但是我可以举一个很简单的例子让大家明白。

例如在我们的用户表中有以下几个字段：

username、password、name、age、gender

依次的意思就是用户名、密码、昵称、年龄、性别。

当我们想要去获取所有的用户，也就是做一个用户列表时，我们其实并不想把密码给展示出来。

因为在实际的项目中，密码都是加密存在的，我们也不可能会给别人展示出密码。

我们只需要username、name、age、gender这几个字段的信息，但是如何去除掉password的呢？

这就是我们今天要讲的响应模型，也就是响应用户请求的模型。

首先我们建立一个User模型：

models.py

```
class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True, index=True) # 设置主键和索引
    username = Column(String(32))
    password = Column(String(32))
    age = Column(Integer)
    gender = Column(String(32))
```

schemas.py

```
class User(BaseModel):
    username : str
    password : str
    age : int
    gender : str
```

然后在main.py中编写获取所有用户的函数

```
@app.get('/users')
def get_users( db: Session = Depends(get_db)):
    users = db.query(models.User).all()
    return users
```

我们自己手动添加几条数据，跑一下。

	id	username	password	age	gender
	1	kuls	12312312	11	1
	2	admin	admin	123	1

```
unvicorn blog.main:app --reload
```

打开swagger，执行方法后

Server response	
Code	Details
200	<div>Response body</div> <pre>[{ "id": 1, "username": "kuls", "age": 11, "password": "12312312", "gender": "1" }, { "id": 2, "username": "admin", "age": 123, "password": "admin", "gender": "1" }]</pre>

数据是正常打印出来了。

那么此时我们如何去让password消失掉呢？

其实也比较简单，我们只需要编写schemas.py里面：

添加一个ShowUser的schemas

```
class ShowUser(BaseModel):
    username : str
    age : int
    gender : str
    class Config(): # 注意此处
        orm_mode = True
```

添加完成之后，我们去main.py里面，刚刚编写的函数

```
from typing import List
@app.get('/users',response_model=List[schemas.ShowUser])
def get_users( db: Session = Depends(get_db)):
    .....
```

因为我们返回的是一个List，所以我们需要从typing中导入List。并且在@app.get修饰器中添加了,response_model=List[schemas.ShowUser]。

除了get的修饰器，其他的例如@app.get()、@app.post()、@app.put()、@app.delete()都是可以添加的。

我们可以看到，password没有显示出来了。

200

Response body

```
[
  {
    "username": "kuls",
    "age": 11,
    "gender": "1"
  },
  {
    "username": "admin",
    "age": 123,
    "gender": "1"
  }
]
```

可能很多读者会觉得这有啥用呢？放心，之后你会在很多场景中用到。这里就不一一给大家展示了。

关于响应模型我建议大家看完本文后，继续阅读官方文档中给出的教程：

<https://fastapi.tiangolo.com/zh/tutorial/response-model/>