第一节：FastAPI框架入门

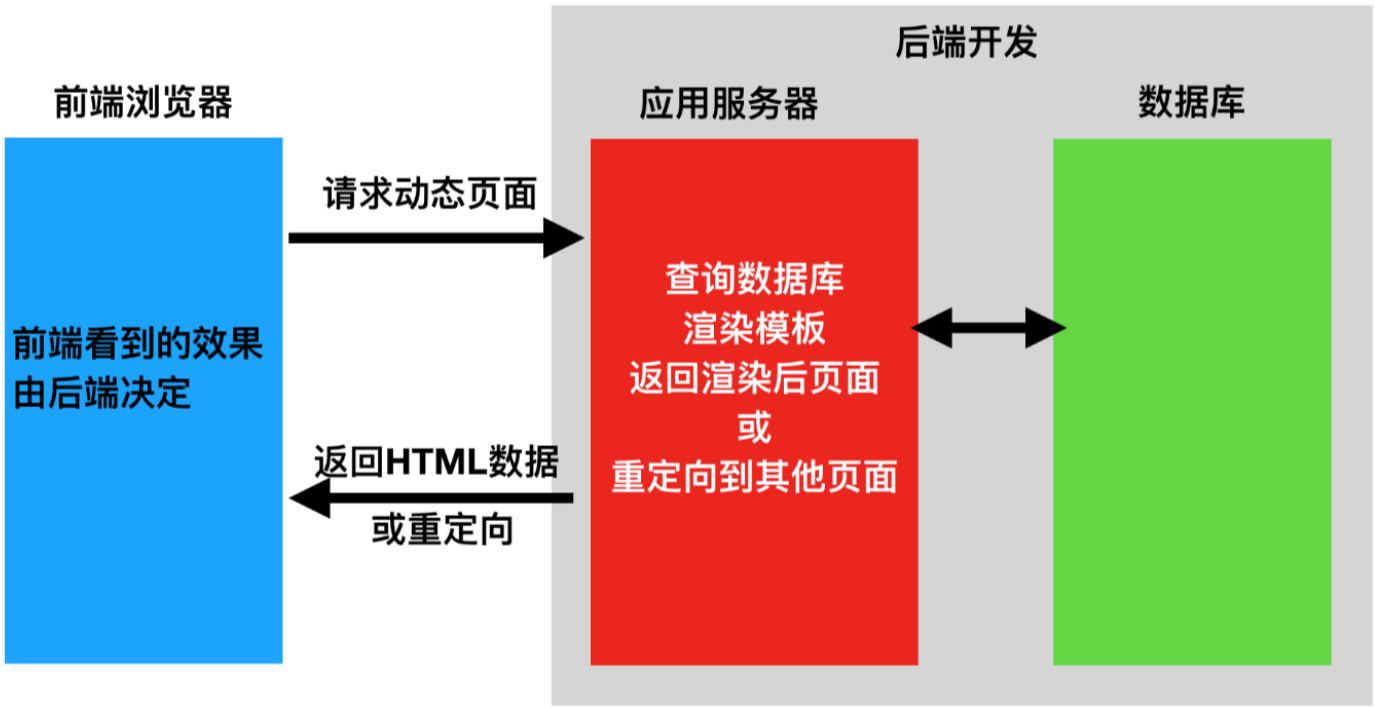
讲师：肖斌

一、FastAPI相关概念

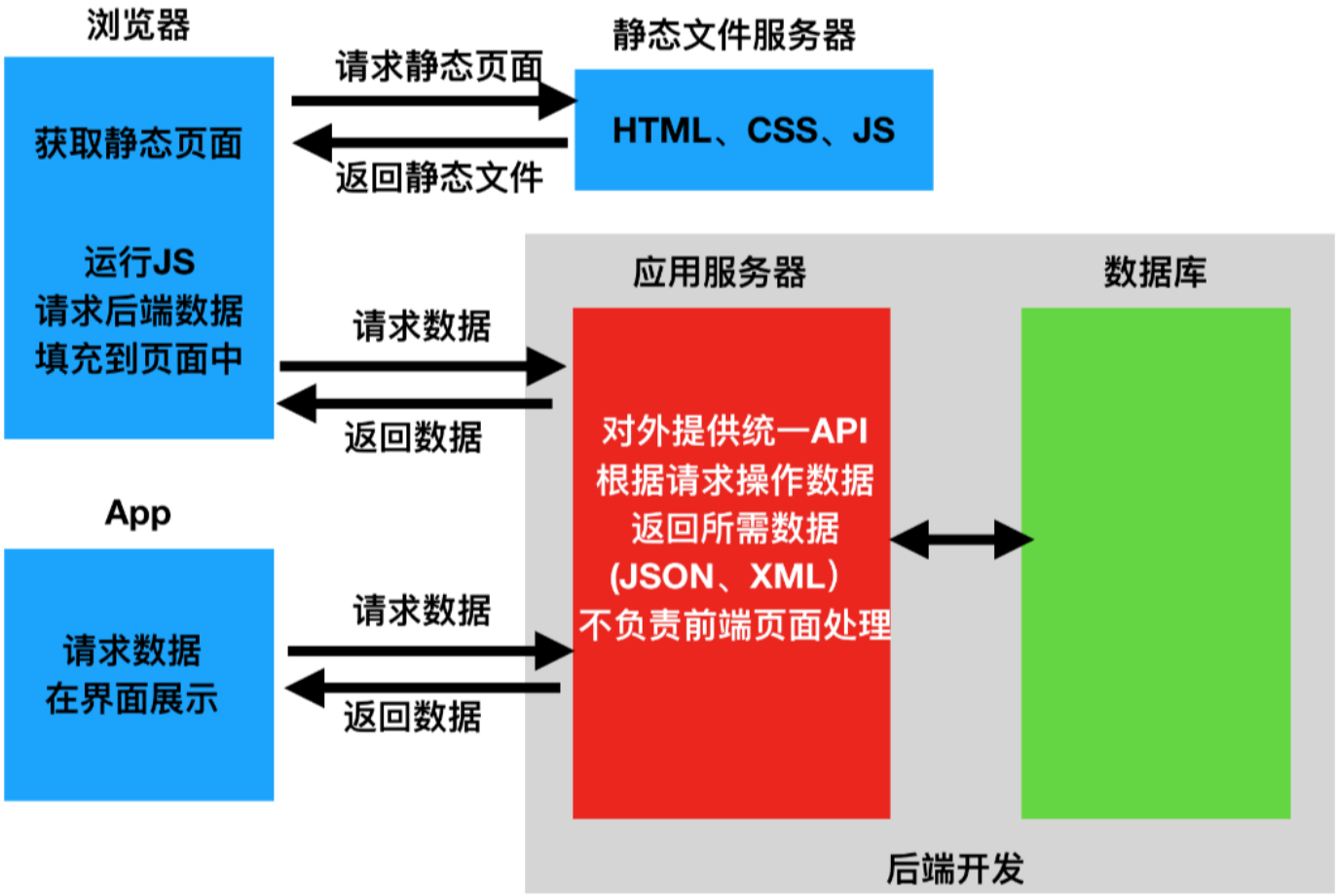
1、Web开发模式

在开发Web应用中，有两种应用模式：

1）前后端不分离：客户端看到的内容和所有界面效果都是由同一个服务端提供出来的。



2）前后端分离：前端的界面效果(html，css，js分离到另一个服务端，python服务端只需要处理业务逻辑和返回数据即可。



应用程序编程接口（Application Programming Interface，API接口）：就是应用程序对外提供了一个操作数据的入口，这个入口可以是一个函数或类方法，也可以是一个url地址或者一个网络地址。当客户端调用这个入口，应用程序则会执行对应代码操作，给客户端完成相对应的功能。

目前市面上大部分公司开发人员使用的接口实现规范主要有：restful、RPC。

REST全称是Representational State Transfer，RESTful是一种专门为Web 开发而定义API接口的设计风格，尤其适用于前后端分离的应用模式中。这是一种**面向资源开发的编程模式**。

这种风格的理念认为后端开发任务就是提供数据的，对外提供的是数据资源的访问接口，所以在定义接口时，客户端访问的URL路径就表示这种要操作的数据资源。

而对于数据资源分别使用POST、DELETE、GET、UPDATE等请求动作来表达对数据的增删查改。

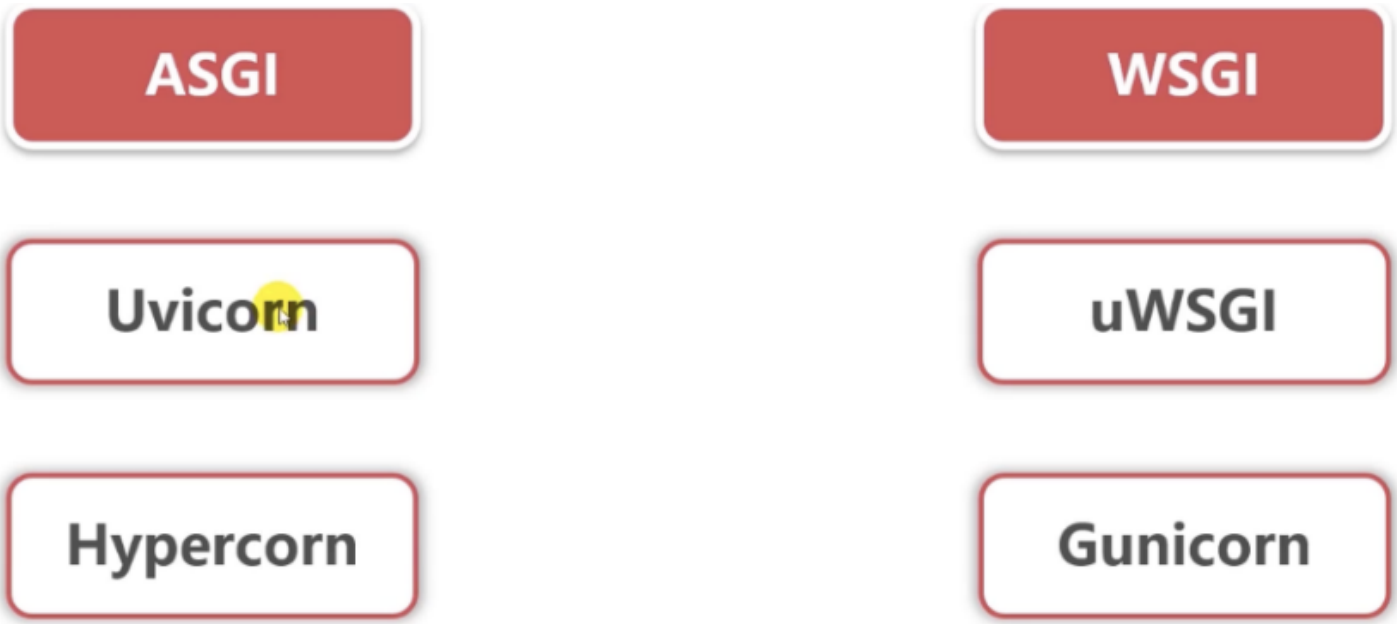
请求方法	请求地址	后端操作
POST	/student/	增加学生
GET	/student/	获取所有学生
GET	/student/1	获取id为1的学生
PUT	/student/1	修改id为1的学生
DELETE	/student/1	删除id为1的学生

restful规范是一种通用的规范，不限制语言和开发框架的使用。事实上，我们可以使用任何一门语言，任何一个框架都可以实现符合restful规范的API接口。

2、ASGI协议和服务

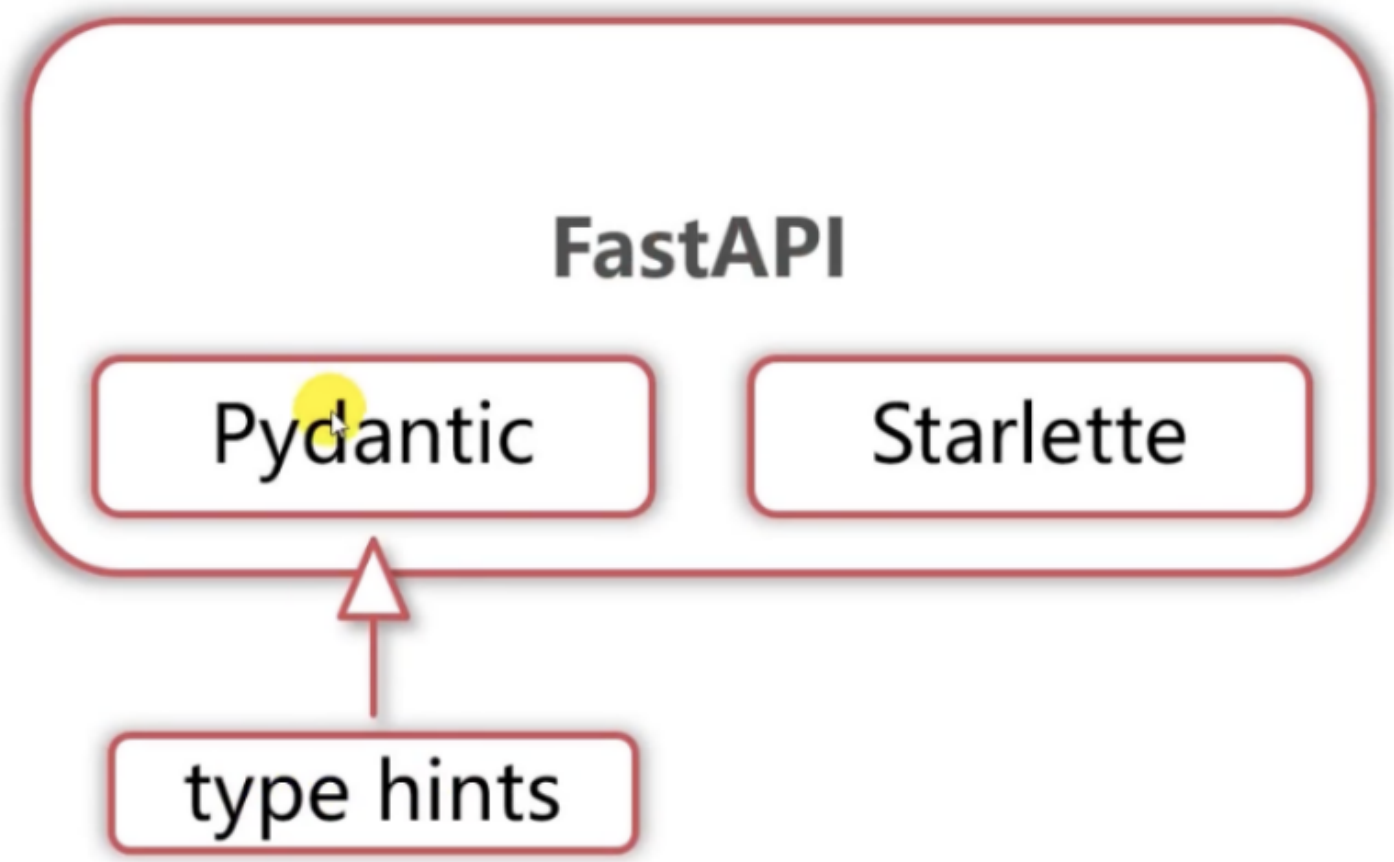
WSGI，（WEB SERVER GATEWAY INTERFACE），Web服务器网关接口，是一种Web服务器网关接口，它是一个Web服务器（如Nginx，uWSGI等服务器）与Web应用（如Flask框架写的程序）通信的一种规范。当前运行在WSGI协议之上的Web框架有**Flask，Django**。

ASGI：异步网关协议接口（*Asynchronous Server Gateway Interface*），一个介于网络协议服务和Python应用之间的标准接口，能够处理多种通用的协议类型，包括HTTP，HTTP2和WebSocket。当前运行在ASGI协议之上的Web框架有**FastAPI，Django-3.2以后**。



3、FastAPI框架介绍

FastAPI是一个现代、快速（高性能）的Web框架，用于构建API。是建立在Starlette和Pydantic基础上的。它基于Python 3.7+的类型提示（type hints）和异步编程（asyncio）能力，使得代码易于编写、阅读和维护。FastAPI具有自动交互式文档（基于OpenAPI规范和JSON Schema）、数据验证、依赖注入（Dependency Injection）等功能，这些功能使得API的开发速度更快、更可靠。FastAPI还支持WebSocket，可以轻松地扩展到更复杂的应用场景。



为什么要用FastAPI：

1、强大的性能：

FastAPI使用Pydantic和Starlette等现代Python库，可以实现比Flask和Django等传统框架更快的性能。FastAPI的异步支持使其能够处理大量并发请求，并在高负载下保持稳定的性能。

2、自动生成API文档：

FastAPI可以自动生成API文档，这是一个重要的优势。FastAPI使用OpenAPI标准，可以通过Swagger UI或Redoc自动生成交互式API文档。这使得API的使用和测试更加容易和直观。

3、类型提示和自动验证：

FastAPI使用Python的类型注解和Pydantic库来自动验证请求和响应的数据。这使得代码更加清晰和易于维护，并可以在运行时自动检查数据类型和格式。这也可以帮助开发人员更早地发现和解决错误。

4、快速开发：

FastAPI提供了一些快速开发的功能，如自动路由和依赖注入。这使得开发人员可以更快地编写API，并且可以更容易地管理和维护代码。FastAPI还提供了一些常见的功能，如身份验证和数据库集成，以便开发人员可以更快地构建功能完整的API。

5、 强大的交互式API文档：

FastAPI的交互式API文档不仅可以自动生成，而且还非常强大。开发人员可以在文档中执行请求和测试，以便更好地理解API的使用和行为。文档还提供了一些有用的工具，如请求和响应模型的可视化和自动代码生成。这使得API的使用和测试更加容易和直观。

二、第一个FastAPI程序

1、安装Python虚拟环境

为什么要使用虚拟环境

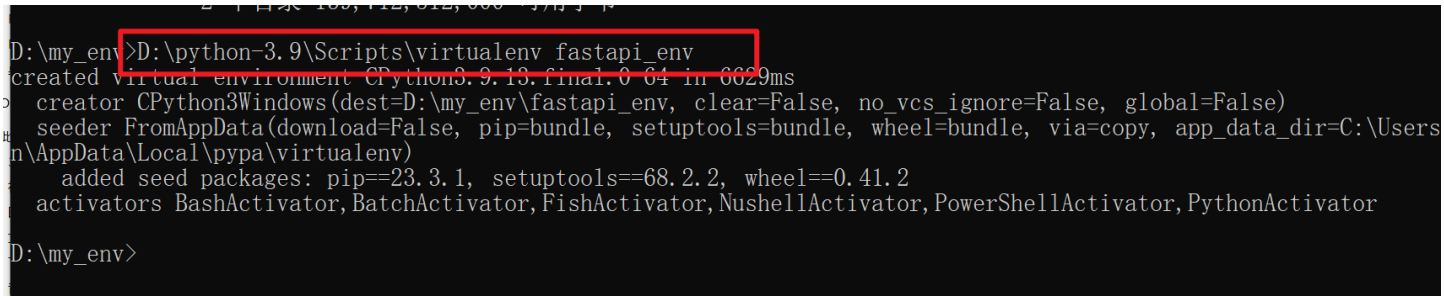
1. 项目部署时，直接导出项目对应的环境中的库就可以了；
2. 同时开发多个项目，各自项目使用的python版本不同，譬如一个是python2，另一个是python3，那么需要来回的切换python版本；
3. 当你同时开发多个项目时，特别是多个项目使用同一个库，譬如：django，但是各自项目使用的django的版本不一致时，那么你在开发这些项目时，需要来回的卸载和安装不同的版本，因为同一个python环境中，同名的库只能有一个版本。

虚拟环境的安装步骤

1. 安装好python环境
2. 安装虚拟环境库，在cmd中输入：

```
pip install virtualenv
```

3. 创建虚拟环境，在cmd中切换到需要创建虚拟环境的目录下，执行：




```
virtualenv env_name
```

4. 激活虚拟环境，在cmd中进入到 第三步创建的 env_name/Scripts 目录下，执行：

```
activate
```

执行成功后，在cmd中，当前输入行前面会有 (env_name) 的前缀

在当前状态下，使用 pip 就是在虚拟环境中安装第三方库了

5. 退出虚拟环境，cmd中输入：

```
deactivate
```

2、安装FastAPI

需要安装所有的可选依赖及对应功能，包括了 `uvicorn` ，你可以将其用作运行代码的服务器。

pip install fastapi[all] -i https://mirrors.aliyun.com/pypi/simple/

你也可以分开来安装：

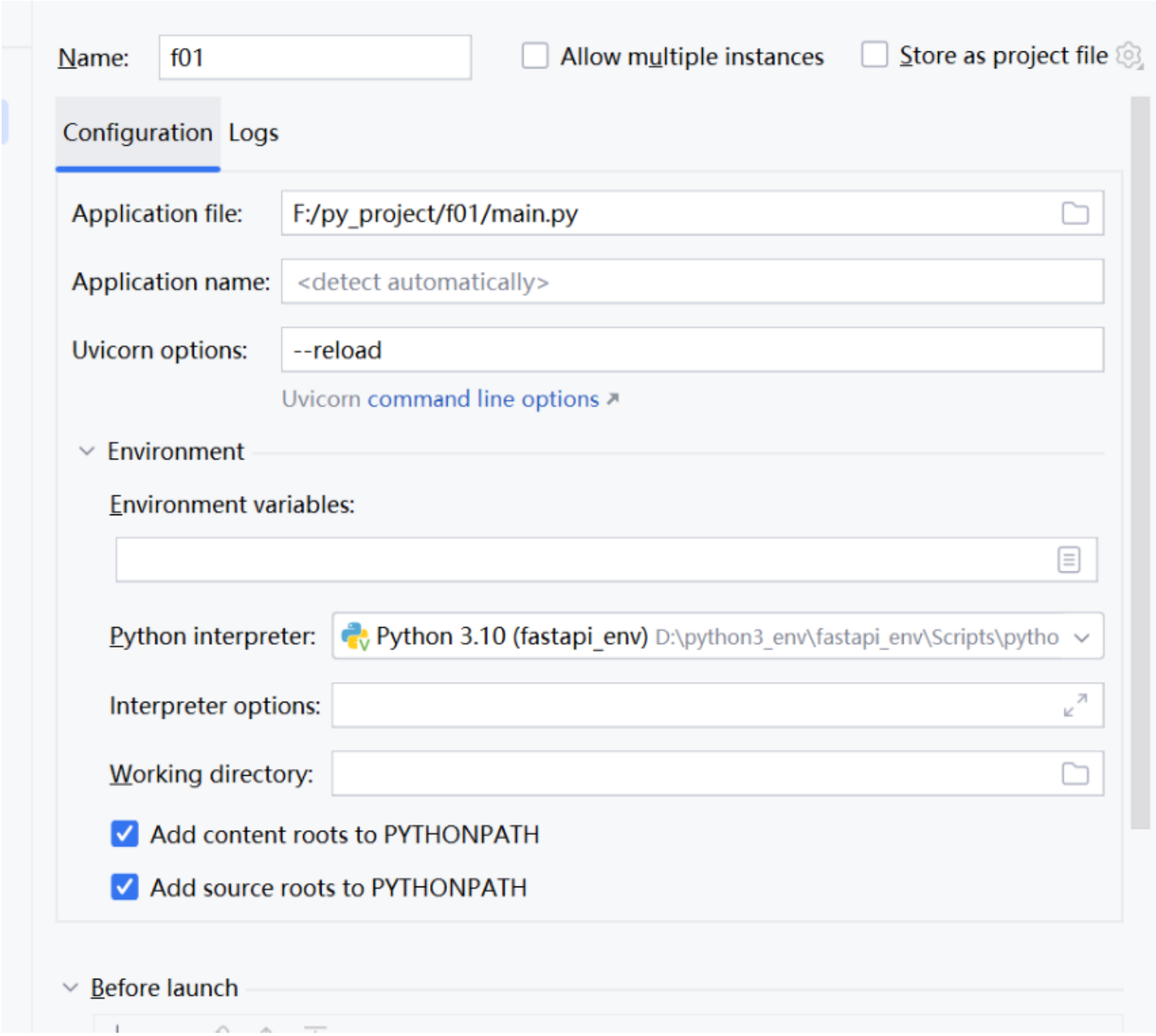
假如你想将应用程序部署到生产环境，你可能要执行以下操作：

```
pip install fastapi
```

并且安装 `uvicorn` 来作为服务器：

```
pip install "uvicorn[standard]"
```

3、创建项目



创建一个main.py

```
from fastapi import FastAPI

app = FastAPI()

```

```python

@app.get("/")
def root():
    return 'hello'


@app.get("/hello/{name}")
def say_hello(name: str):
    return {"message": f"Hello {name}"}

# 可以不要
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="127.0.0.1", port=8000)

```

4、启动

1）通过pycharm启动按钮（开发模式）

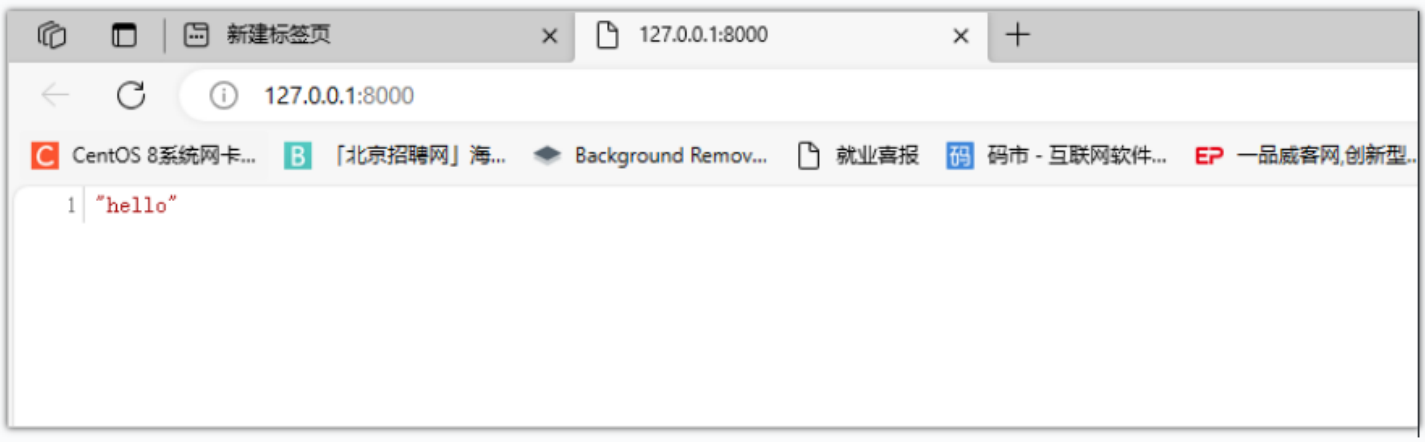
2）通过uvicorn命令启动（生产模式）

`uvicorn main:app` --reload 命令含义如下:

- `main`：`main.py` 文件（一个 Python「模块」）。
- `app`：在 `main.py` 文件中通过 `app = FastAPI()` 创建的对象。
- `--reload`：让服务器在更新代码后重新启动。仅在开发时使用该选项。

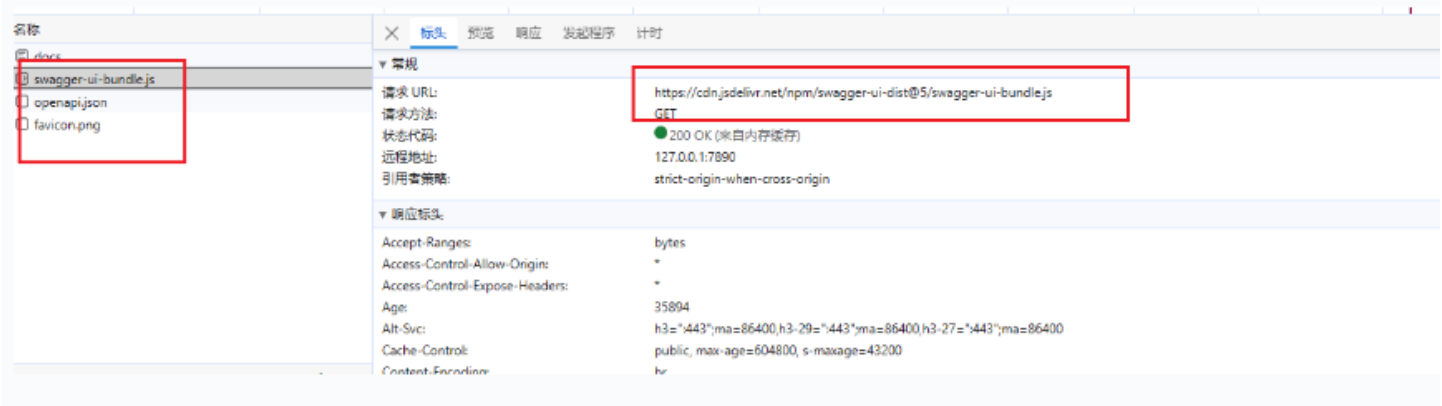
3)在main.py中定义main函数（备用）

5、访问接口和接口文档

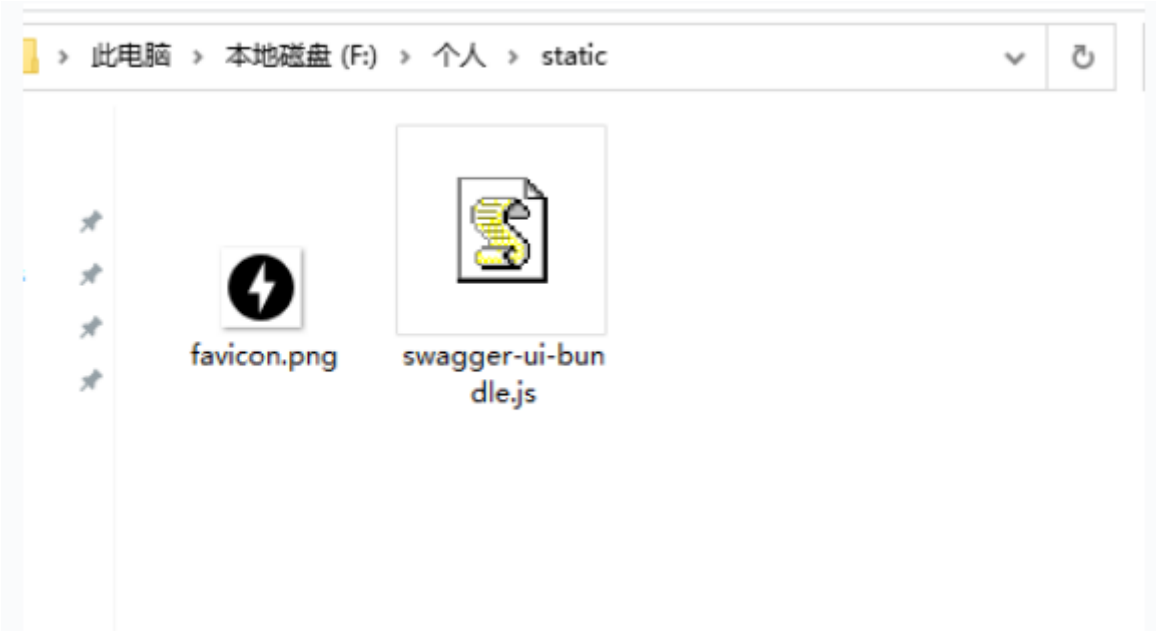


三、接口文档打不开的解决

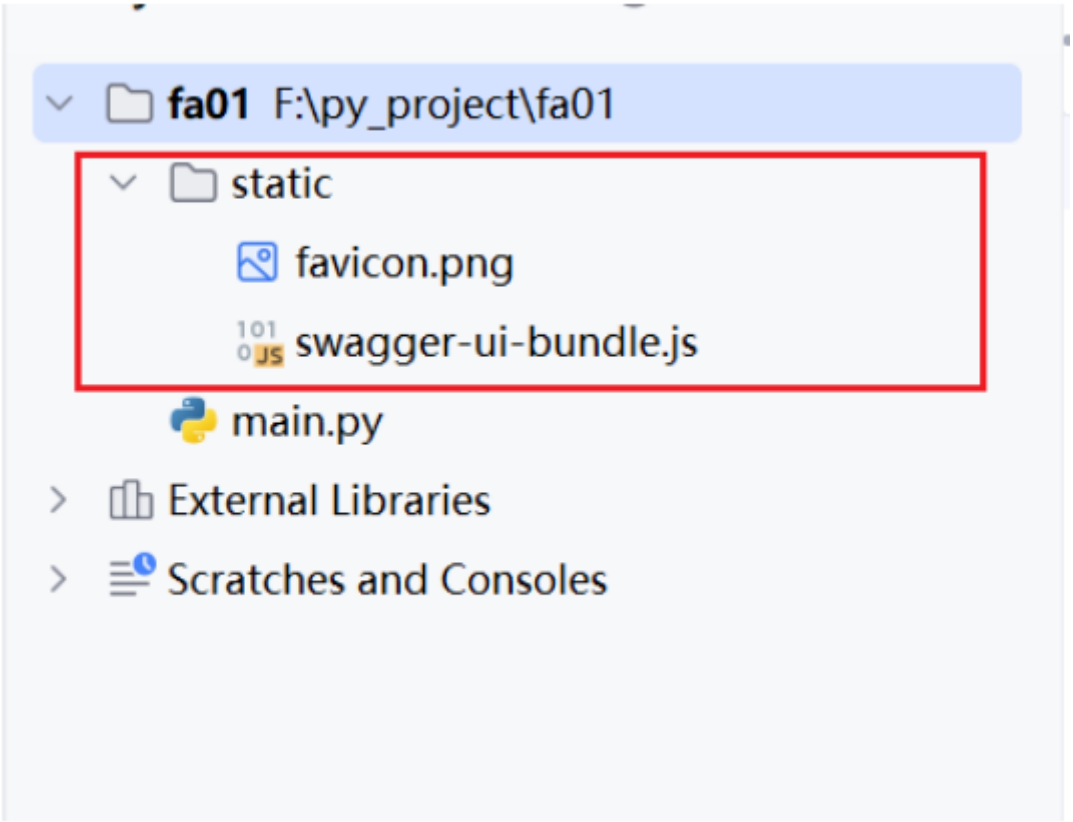
这个接口文档调用了一些css样式和 js脚本，这些脚本是部署在国外的，总之 就是因为这个原因导致我们没法访问了，由此我们需要把这个脚本从网上下载下来，放到本地，把此处调用国外的脚本变成调用我们自己本地的，即可。



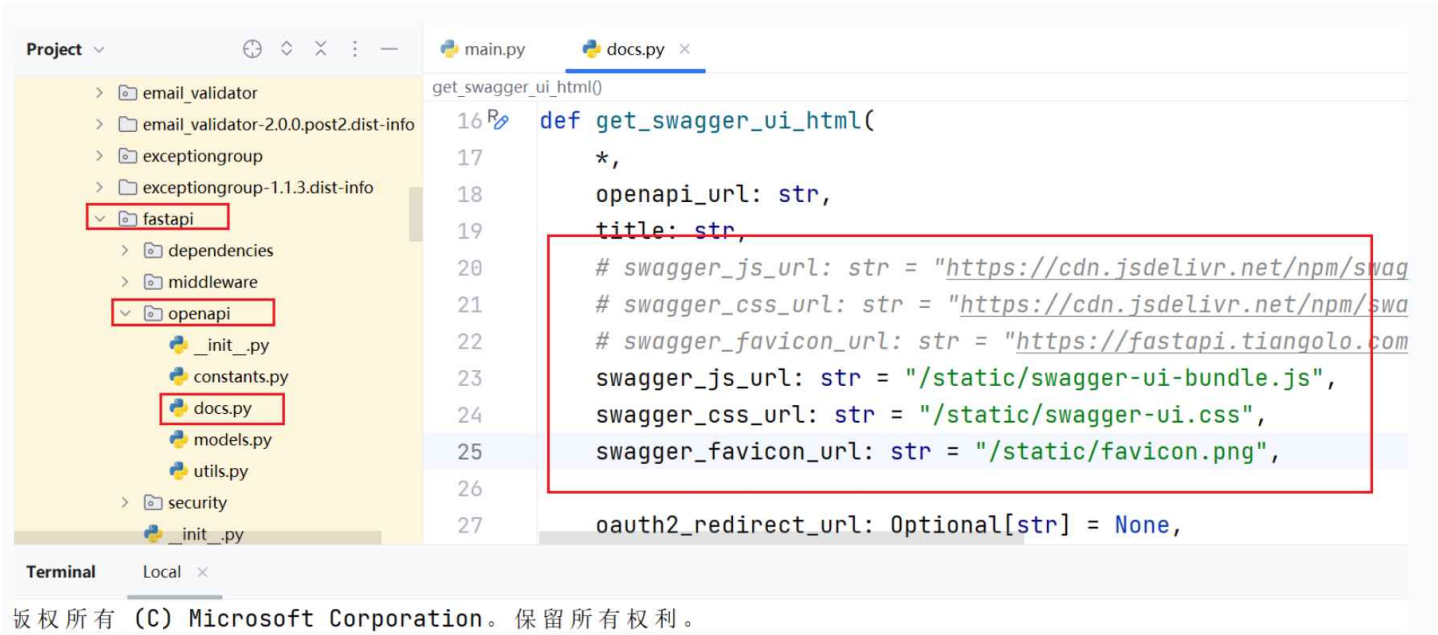
1、下载这些国外服务器的资源到一个static目录中



2、复制到项目中



3、修改FastAPI的源代码

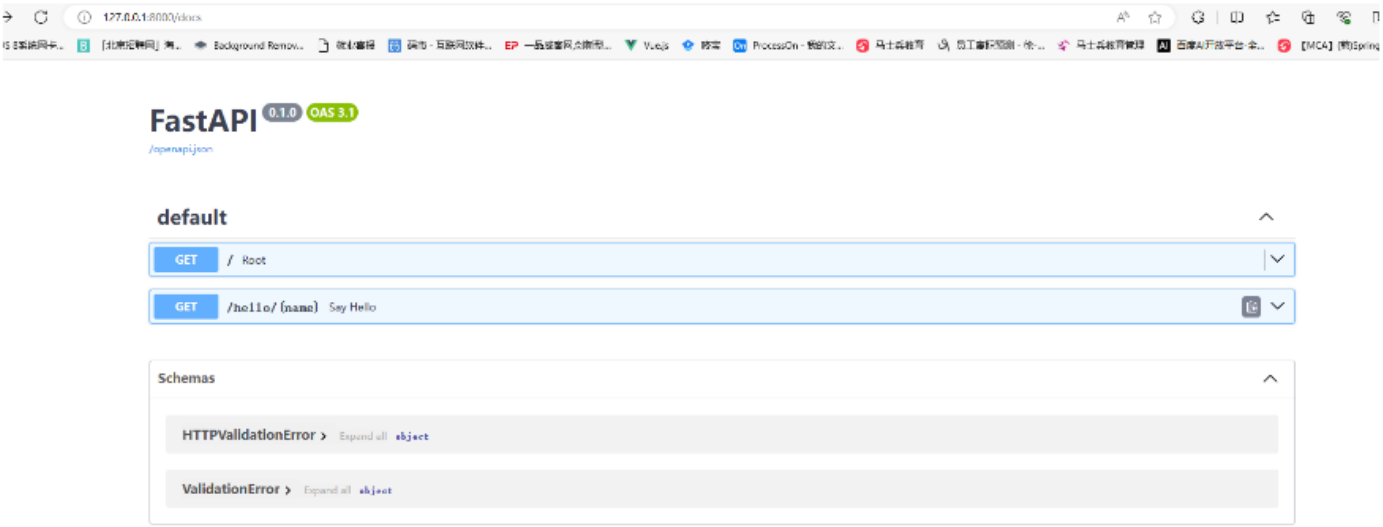


4、在app中注册static目录

main.py × docs.py

```python
from fastapi import FastAPI
from starlette.staticfiles import StaticFiles

app = FastAPI()
# 把项目下的static目录作为访问路径
app.mount('/static', StaticFiles(directory='static'), name='static')

```

5、打开接口文档



第二节：FastAPI的路由和请求参数

讲师：肖斌

一、FastAPI的路由

Route路由, 是一种映射关系！路由是把客户端请求的url路径与视图函数进行绑定映射的一种关系。

- **路径**：是 `/`
- **请求方法**：是 `get`
- **函数**：是位于「装饰器」下方的函数（位于 `@app.get("/")` 下方）

```python
# 路径操作装饰器参数
# tags 文档标题
# summary 文档总结
# description 文档描述
# response_description 响应详情内容
# deprecated 接口文档是否废弃
@app.post("/items", tags=["这是items测试接口"],
          summary="this is items测试 summary",
          description="this is items测试 description...",
          response_description="this is items测试 response_description",
          deprecated=False)
def test():
    return {"items": "items数据"}

```

1、路由分发

在main.py中的app作为主路由

```python
from fastapi import FastAPI
import uvicorn
from apps.app01.urls import shop
from apps.app02.urls import user

app = FastAPI()
```

```
7
8  # 路由分发include_router
9  app.include_router(shop, prefix="/shop", tags=["购物中心接口"])
10 app.include_router(user, prefix="/user", tags=["用户中心接口"])
11
```

其他的子路由：

```
1  from fastapi import APIRouter
2
3  shop = APIRouter()
4
5
6  @shop.get("/food")
7  def shop_food():
8      return {"shop": "food"}
9
10
11 @shop.get("/bed")
12 def shop_bed():
13     return {"shop": "bed"}
```

2、路由参数

请求参数，通过路由路径携带的方式；我们称之为路由传参。参数也叫： 路由参数。你可以使用与 Python 格式化字符串相同的语法来声明路径"参数"或"变量"：

```
1  @app.get("/items/{item_id}")
2  def read_item(item_id):
3      return {"item_id": item_id}
```

路径参数 `item_id` 的值将作为参数 `item_id` 传递给你的函数。所以，如果你运行示例并访问 [http://127.0.0.1:8000/items/foo]，将会看到如下响应：

```
1  {"item_id":"foo"}
```

1、定义参数的类型

你可以使用标准的 Python 类型标注为函数中的路径参数声明类型。

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

在这个例子中，`item_id` 被声明为 `int` 类型。FastAPI 通过上面的类型声明提供了对请求的自动"解析"。

同时还提供数据校验功能：

如果你通过浏览器访问 [http://127.0.0.1:8000/items/foo]，你会看到一个清晰可读的 HTTP 错误：

```
{
    "detail": [
        {
            "loc": [
                "path",
                "item_id"
            ],
            "msg": "value is not a valid integer",
            "type": "type_error.integer"
        }
    ]
}
```

因为路径参数 `item_id` 传入的值为 `"foo"`，它不是一个 `int`。

你可以使用同样的类型声明来声明 `str`、`float`、`bool` 以及许多其他的复合数据类型。

2、路由匹配的顺序

由于路由匹配操作是按顺序依次运行的，你需要确保路径 `/users/me` 声明在路径 `/users/{user_id}` 之前：

```python
@app.get("/users/me")
def read_user_me():
    return {"user_id": "the current user"}


@app.get("/users/{user_id}")
def read_user(user_id: str):
    return {"user_id": user_id}
```

否则，`/users/{user_id}` 的路径还将与 `/users/me` 相匹配，"认为"自己正在接收一个值为 `"me"` 的 `user_id` 参数。

3、预设值参数

如果你有一个接收路径参数的路径操作，但你希望预先设定可能的有效参数值，则可以使用标准的 Python `Enum` 类型。

Python中的枚举数据类型：是指列出有穷集合中的所有元素，即一一列举的意思。在Python中，枚举可以视为是一种数据类型，当一个变量的取值只有几种有限的情况时，我们可以将其声明为枚举类型。

```
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"}
```

二、请求URL传参

1、URL传参

url请求参数是通过url请求地址携带的，例如，在以下 url 中：

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

这些请求参数是键值对的集合，这些键值对位于 URL 的 `?` 之后，并以 `&` 符号分隔。

请求参数为：

- `skip`：对应的值为 `0`
- `limit`：对应的值为 `10`

当你为它们声明了 Python 类型（在上面的示例中为 `int`）时，它们将转换为该类型并针对该类型进行校验。

```
from fastapi import FastAPI

app = FastAPI()

fake_items_db = [{"item_name": "Foo"}, {"item_name": "Bar"}, {"item_name": "Baz"


@app.get("/items/")
def read_item(skip: int = 0, limit: int = 10):
    return fake_items_db[skip : skip + limit]
```

URL请求参数不是路径的固定部分，因此它们可以是可选的，并且可以有默认值。

在上面的示例中，它们具有 `skip=0` 和 `limit=10` 的默认值。你可以将它们的默认值设置为 `None`。

```
from typing import Union

from fastapi import FastAPI

app = FastAPI()


@app.get("/items/{item_id}")
async def read_item(item_id: str, q: Union[str, None] = None, short: bool = Fals
    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
```

Union类型是一种用于表示一个变量可以是多个不同类型之一的类型注解。它是typing模块中的一个类，可以与其他类型一起使用，以指定一个变量可以接受的多个类型。

Union类型的语法为：Union[type1, type2, ...]

这里的type1、type2等代表要包含在Union类型中的类型。

注意：当你想让一个查询参数成为必需的，不声明任何默认值就可以：

```python
@app.get("/items/{item_id}")
def read_user_item(
    item_id: str, needy: str, skip: int = 0, limit: Union[int, None] = None
):
    item = {"item_id": item_id, "needy": needy, "skip": skip, "limit": limit}
    return item
```

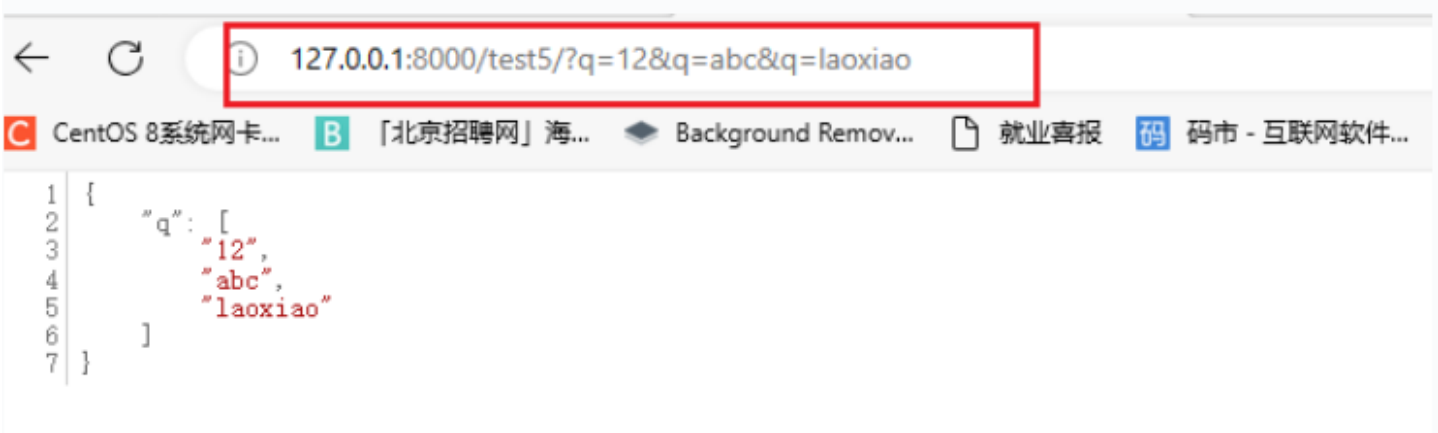
在这个例子中，有3个查询参数：

- `needy`，一个必需的 `str` 类型参数。
- `skip`，一个默认值为 `0` 的 `int` 类型参数，也是可选的。
- `limit`，一个可选的 `int` 类型参数。

2、一个参数名，多个值

我们还可以使用 Query 去接收一组值。使用 List[str] 来接收一组值，里面的str可以根据需求随意调换。

```python
@app.get("/test5/")
async def read_items(q: Union[List[str], None] = None):
    query_items = {"q": q}
    return query_items
```



这是因为参数 q中以一个 **Python list** 的形式接收到查询参数 **q** 的多个值。

3、参数校验

Query是 FastAPI专门用来装饰URL请求参数的类，也可以提供校验。

1）默认值设置，和参数接口描述

```
@app.get("/items/")
async def read_items(q: Union[str, None] = Query(default=None, description="参数(
    query_items = {"q": q}
    return query_items
```

description是 Query中的一个字段，用来描述该参数；

2）字符串长度校验

```
@app.get("/items2/")
async def read_items(q: Union[str, None] = Query(default=None, max_length=50, mi
    query_items = {"q": q}
    return query_items
```

注意：max_length和 min_length仅可用于 str 类型的参数。

3）正则表达式校验

```python
@app.get("/items3/")
async def read_items(
    q: Union[str, None] = Query(default=None, regex="^laoxiao$")
):
    query_items = {"q": q}
    return query_items
```

也就是说该接口的查询参数只能是" laoxiao"，不然就会报错，当然其他的正则表达式都是可以拿来用的！

4）数值大小校验

如果参数的类型是int，float就可以采用Query进行大小校验，或者范围校验。

```python
@app.get("/test4/")
async def read_items4(id: int = Query(description="id参数必须是0到1000之间", gt=0,
    results = {"item_id": id}
    if q:
        results.update({"q": q})
    return results
```

- 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）

三、请求体传参

当你需要将数据从客户端(例如浏览器)发送给API时,你将其作为[请求体] （request body）发送,请求体是客户端发送给API的数据.响应体是API发送给客户端的数据。

```python
class Addr(BaseModel):
    province: str
    city: str


class User(BaseModel):
    # name: str = Field(regex="^a")  # name参数值必须是a开头
    name: str
```

```python
    age: int = Field(default=0, ge=0, lt=100)  # age参数给默认值0,数值区间范围约束为
    birth: Union[date, None] = None  # birth约束类型为date或者None,默认值为None
    friends: List[int] = []  # friends给默认值为[]
    description: Optional[str] = None  # description约束类型为str或者None,默认值为N
    addr: Addr   # 嵌套类型

    @field_validator("name")  # 校验name字段
    def name_must_alpha(cls, value):
        assert value.isalpha(), 'name must be alpha'
        # 断言语句,检查value是否全部由字母组成,如果value的值不满足条件则引发AssertionEr
        # 并且异常错误消息是:"name must be alpha"
        return value


@app.post("/data")
async def data(user: User):
    print(user.name, user.age)
    return user

```

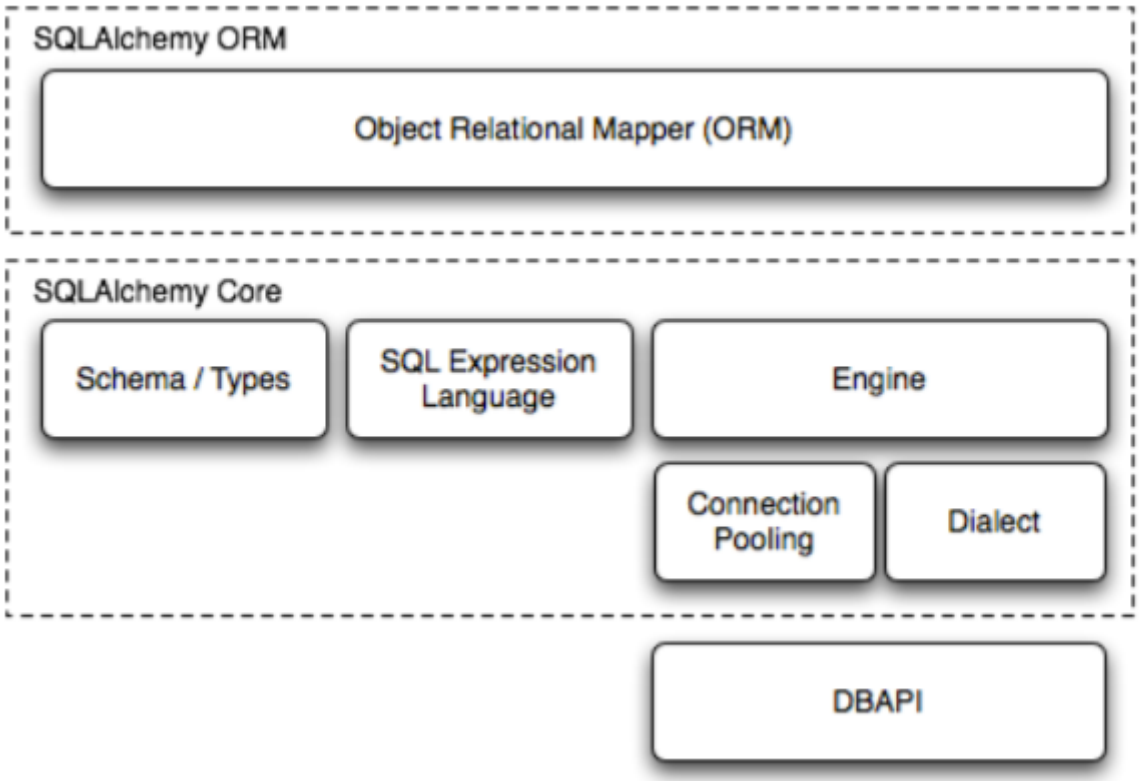
注意： `Field` **的工作方式和** `Query`**、** `Path` **和** `Body` **相同，包括它们的参数等等也完全相同。**

第三节：SQLAlchemy-2.0中模型定义

讲师：肖斌

一、SQLAlchemy的介绍

SQLAlchemy 是 Python 生态系统中最流行的 ORM。SQLAlchemy 设计非常优雅，分为了两部分——底层的 Core 和上层的传统 ORM。在 Python 乃至其他语言的大多数 ORM 中，都没有实现很好的分层设计， 比如 django 的 ORM，数据库链接和 ORM 本身完全混在一起。

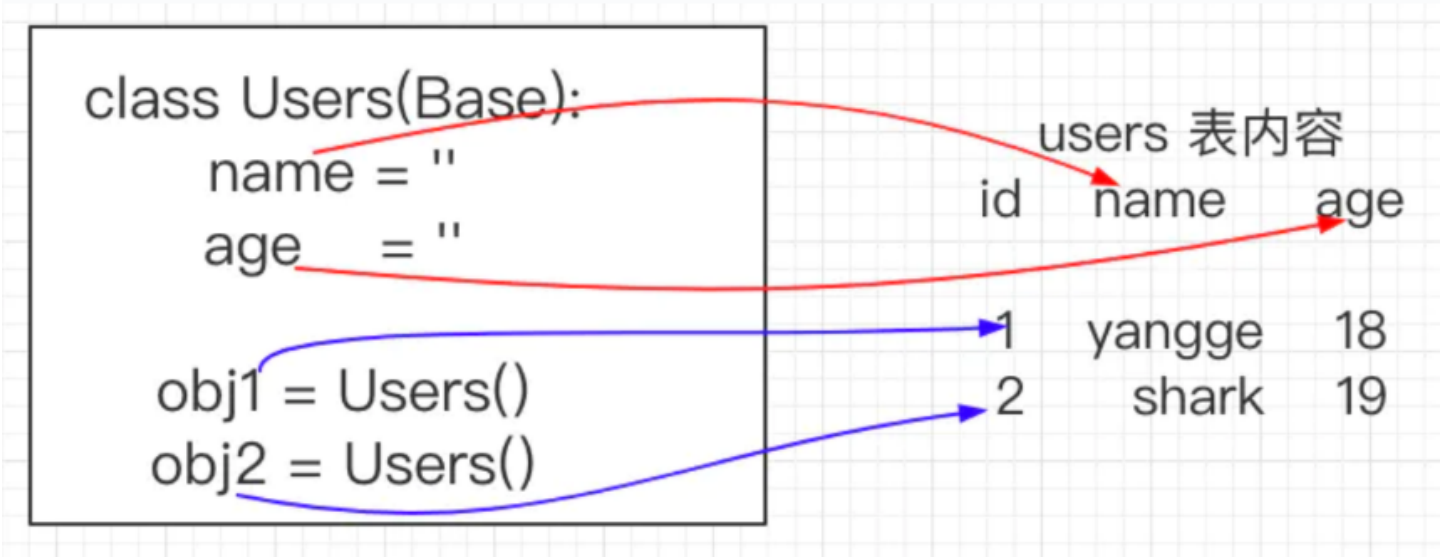


SQLAlchemy 是 Python 中一个通过 ORM 操作数据库的框架。

SQLAlchemy对象关系映射器提供了一种方法，用于将用户定义的Python类与数据库表相关联，并将这些类（对象）的实例与其对应表中的行相关联。它包括一个透明地同步对象及其相关行之间状态的所有变化的系统，称为*工作单元*，以及根据用户定义的类及其定义的彼此之间的关系表达数据库查询的系统。

可以让我们使用类和对象的方式操作数据库，从而从繁琐的 sql 语句中解脱出来。

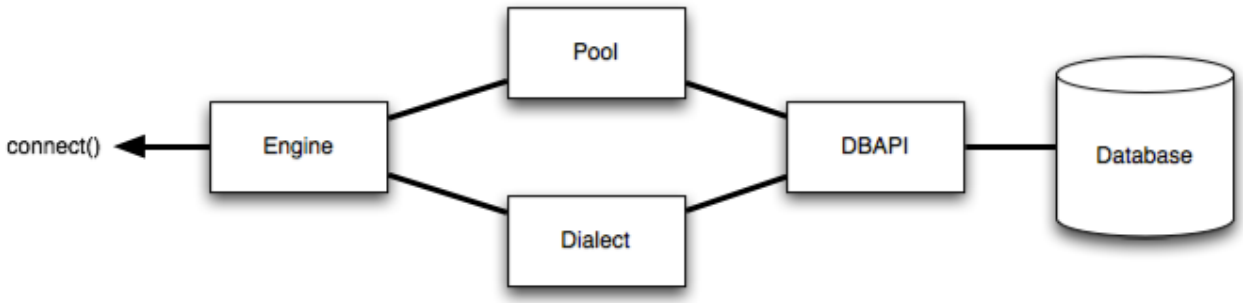
ORM 就是 Object Relational Mapper 的简写，就是关系对象映射器的意思。



二、数据库引擎

任何SQLAlchemy应用程序的开始都是一个名为 `Engine` . 此对象充当连接到特定数据库的中心源，提供工厂和称为 connection pool 对于这些数据库连接。引擎通常是一个只为特定数据库服务器创建一次的全局对象，并使用一个URL字符串进行配置，该字符串将描述如何连接到数据库主机或后端。

总体结构如下：



sqlalchemy使用 create_engine() 函数从URL生成一个数据库引擎对象。比如:

```
1 engine = create_engine(r'sqlite:///C:\path\to\foo.db')
```

1、支持的数据库

URL通常可以包括用户名、密码、主机名、数据库名以及用于其他配置的可选关键字参数。主题格式为：

```
1 dialect+driver://username:password@host:port/database
```

注意：你必须安装，你想要的数据库驱动库

1) SQLite数据库

sqlite使用python内置模块连接到基于文件的数据库 `sqlite3` 默认情况下。

```
# Unix/Mac - 4 initial slashes in total
engine = create_engine('sqlite:////absolute/path/to/foo.db')

# Windows
engine = create_engine('sqlite:///C:\\path\\to\\foo.db')

# Windows alternative using raw string
engine = create_engine(r'sqlite:///C:\path\to\foo.db')

# 当前目录下的test.db
engine = create_engine(r'sqlite:///test.db', echo=True, future=True)
```

2) MySQL数据库

mysql方言使用mysql python作为默认dbapi。mysql dbapis有很多，包括pymysql 和 mysqlclient：

```
# default
engine = create_engine('mysql://scott:tiger@localhost/foo?charset=utf8')

# mysqlclient (a maintained fork of MySQL-Python)
engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo?charset=utf8')

# PyMySQL
engine = create_engine('mysql+pymysql://scott:tiger@localhost/foo?charset=utf8')
```

3) Microsoft SQL数据库

SQL Server方言使用pyodbc作为默认dbapi。PYMSSQL也可用：

```
# pyodbc
engine = create_engine('mssql+pyodbc://scott:tiger@mydsn')

# pymssql
engine = create_engine('mssql+pymssql://scott:tiger@hostname:port/dbname')
```

2、数据库引擎的参数

```
    ),
)
def create_engine(url: Union[str, _url.URL], **kwargs: Any) -> Engine:
    """Create a new :class:`_engine.Engine` instance.

    The standard calling form is to send the :ref:`URL <database_urls>` as t
```

1. echo=False -- 如果为真，引擎将记录所有语句以及 repr() 其参数列表的默认日志处理程序
2. future -- 使用2.0样式 Engine 和 Connection API。
3. logging_name -- 将在“sqlalChemy.engine”记录器中生成的日志记录的“name”字段中使用的字符串标识符。
4. pool_size=5 # 连接池的大小默认为 5 个，设置为 0 时表示连接无限制
5. pool_recycle=3600, # 设置时间以限制数据库自动断开
6. pool_timeout：连接超时时间，默认为30秒，超过时间的连接都会连接失败。

三、定义模型类

这种模型类结构称为声明性映射，它同时定义了 Python 对象模型，以及描述的数据库元数据 在特定数据库中存在或将要存在的真实 数据库 表。

映射从一个基类开始，并且是 通过对类的继承来创建一个简单的子类。这里的父类是：Base 模型类。

1、定义模型

```python
# 定义所有模型的父类
class Base(DeclarativeBase):
    """每一种表都拥有的字段映射"""

    create_time: Mapped[datetime] = mapped_column(insert_default=func.now(), com
    update_time: Mapped[datetime] = mapped_column(insert_default=func.now(), onu
                                                  comment='记录最近修改的时间')
```

```python
# 枚举类
class SexValue(enum.Enum):
    MALE = '男'
    FEMALE = '女'


```

```python
# 定义员工模型类
class Employee(Base):
    """员工的模型类"""
    __tablename__ = 't_emp'

    # id属性， 字段。主键，自增
    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    name: Mapped[str] = mapped_column(String(20), unique=True, nullable=False)

    # DECIMAL(10, 2) ,10代表总位数，2代表小数点的位数。
    sal: Mapped[Decimal] = mapped_column(DECIMAL(10, 2), nullable=True, comment=
    bonus: Mapped[Optional[int]] = mapped_column(SmallInteger, default=0, commen
    entry_date: Mapped[date] = mapped_column(insert_default=func.now(), comment=
    is_leave: Mapped[bool] = mapped_column(Boolean, default=False)

    gender: Mapped[SexValue]
```

关键字	对应SQL中的类型
Integer	int
SmallInteger	smallint
BigInteger	bigint
Float(m,d)	float(m,d)
DECIMAL(m,d)或Numeric(m,d)	decimal(m,d)
String(n)	varchar(n)
Text	text
Datetime	datetime
Date	date
Time	time
TIMESTAMP	timestamp
Boolean	boolean
Unicode	可变长度的Unicode字符串
UnicodeText	可变长度字符串，用于存储较长的字符串
LargeBinary	二进制文件

常用约束

关键字	说明
primary_key	主键
autoincrement	自增
nullable	是否可以为Null，默认nullable=True
unique	唯一，默认unique=False
default	默认值，支持字段引用
index	是否建立索引

2、engine负责数据库迁移

```
# 所有的表都重新创建
Base.metadata.drop_all(engine)
Base.metadata.create_all(engine)


# 单独把某个表创建一下
Employee.__table__.drop(engine)
Employee.__table__.create(engine)
```

四、alembic数据库迁移工具

Alembic 使用 SQLAlchemy 作为底层引擎，为关系数据库提供*变更管理*脚本的创建、管理和调用。

1、安装alembic

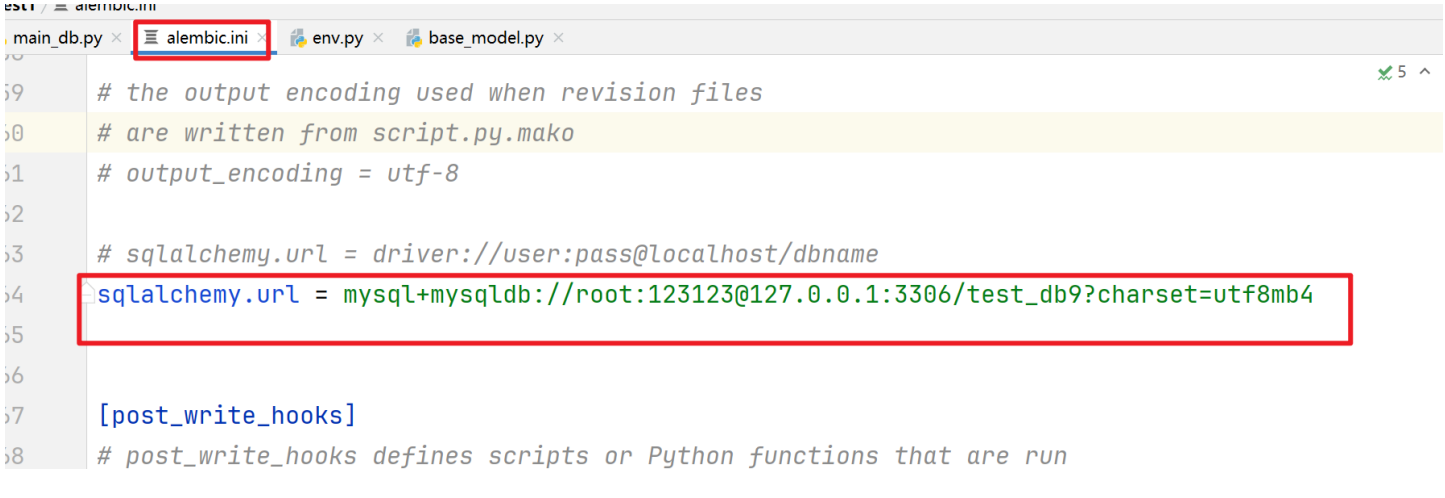
```
pip install alembic
```

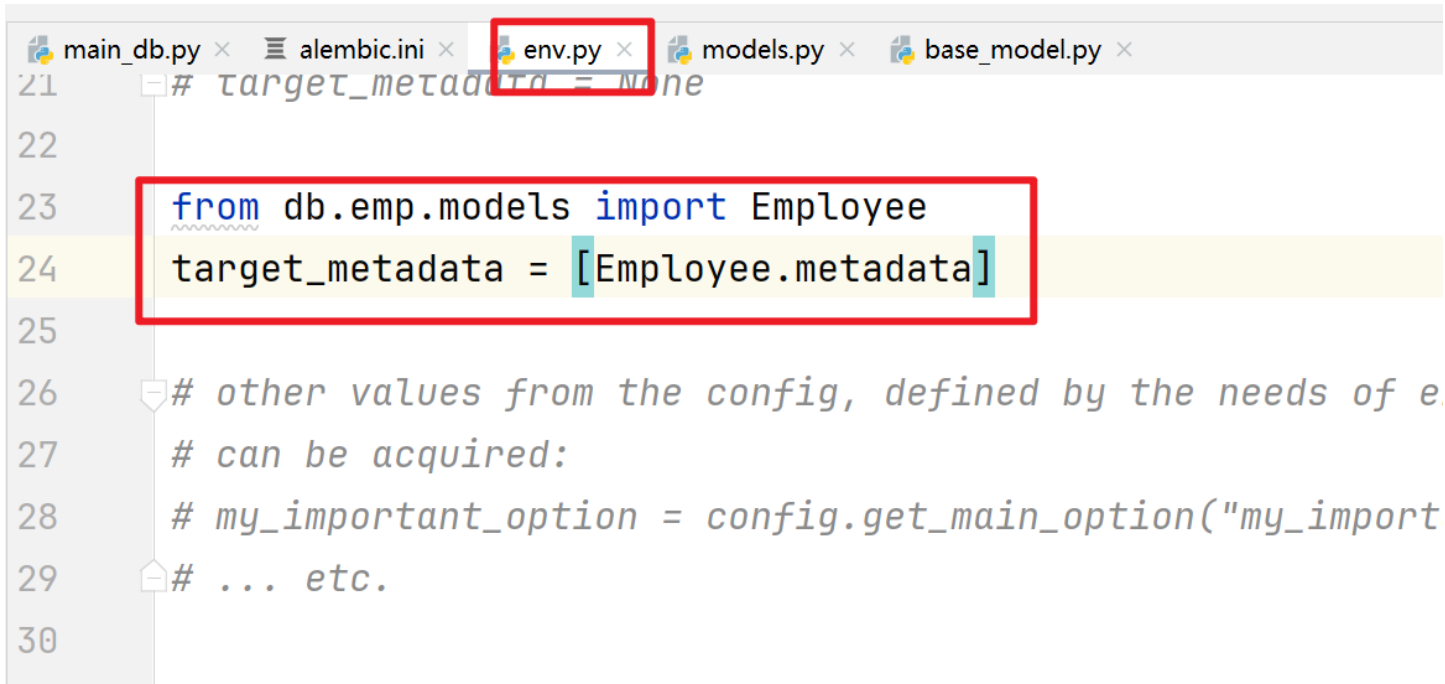
2、初始化alembic环境

命令： alembic init alembic

```
# 查看alembic结构
├── alembic # 二级目录
│   ├── README  # readme
│   ├── env.py  # 环境配置
│   ├── script.py.mako
│   └── versions    # 版本
├── alembic.ini # 配置文件
......
```

3、修改配置和环境





4、执行命令

```
# 自动生成迁移脚本
alembic revision --autogenerate -m "init commit"  # 注意修改了orm之后，修改-m后迁移

# 数据库迁移命令
alembic upgrade head
```

```
6
7
```

- `alembic upgrade head`：将数据库升级到最新版本。
- `alembic downgrade base`：将数据库降级到最初版本。
- `alembic upgrade <version>`：将数据库升级到指定版本。
- `alembic downgrade <version>`：将数据库降级到指定版本。

第四节：Session的相关操作

讲师：肖斌

一、Session对象

session用于创建程序和数据库之间的会话，所有对象的载入和保存都需通过session对象 。在Web项目中，一个请求共用一个session对象。

1、创建Session对象的两种方式

```
# 第一种，需要自己提交事务
with Session(bind=engine) as session:
    session.begin()
    try:
        session.add(some_object)
        session.add(some_other_object)
    except:
        session.rollback()
        raise
    else:
        session.commit()

# 第二种， 不需要自己提交事务
with sessionmaker(bind=engine).begin() as session:
    sess.execute()
```

2、新增模型对象操作

对模型对象进行新增，有两种方式：

```
<!--第一种：类SQL方式-->
insert_stmt = insert(User).values(name='name1')
with Session() as sess:
    sess.execute(insert_stmt)
```

```
    sess.commit()

<!--未绑定参数-->
insert_stmt2 = insert(User)
with Session() as sess:
    sess.execute(insert_stmt2,{'name':'name1'})
    sess.commit()

<!--批量-->
with Session() as sess:
    sess.execute(insert_stmt2,[{'name':'name1'},{'name':'name2'}])
    sess.commit()
```

```
<!--第二种:  面向对象方式-->
obj=User(name='name2')
with Session() as sess:
    sess.add(obj)
    sess.commit()

<!--批量-->
obj=User(name='name2')
obj2=User(name='name2')
with Session() as sess:
    sess.add(obj)
    sess.add(obj2)
    # 或者 s.add_all([obj,obj2])
    sess.commit()

 # 批量添加对象
with sessionmaker(engine).begin() as session:
    emp1 = Employee(name='zs', sal=2000, bonus=500, gender=SexValue.MALE)
    emp2 = Employee(name='ls', sal=3000, bonus=400, gender=SexValue.MALE)
    session.add_all((emp1, emp2))
```

3、简单查询操作

1）根据主键查询

```
emp = session.get(Emp, 1)
```

2) 查询整张表的数据

a、返回模型对象

```
statement = select(Employee)
list_emp = session.scalars(statement).all()
for o in list_emp:
    print(o)


# 需要在Employee模型类中增加一个__str__函数
def __str__(self):
    return f'{self.name}, {self.gender.value}, {self.sal}, {self.entry_date}'
```

b、返回row对象，一般用于指定返回的字段

```
statement = select(Employee.name, Employee.sal, Employee.gender)
# list_emp中的元素不是模型对象， 而是包含多个字段只的row对象
list_emp = session.execute(statement).all()
for row in list_emp:
    print(row.name, row.sal, row.gender.value)
```

c、执行原生的SQL，并返回row对象

```
# 执行原生sql
sql_text = text('select id, name, sal, gender from t_emp')
# list_emp中的元素不是模型对象， 而是包含多个字段只的row对象
list_emp = session.execute(sql_text).all()
for row in list_emp:
    print(row)
    print(row.name, row.sal, row.gender)
```

d、执行原生的sql，并返回模型对象

```
# 执行原生sql
sql_text = text('select id, name, sal, gender from t_emp')
# 手动建立映射关系
new_sql = sql_text.columns(Employee.id, Employee.name, Employee.sal,
Employee.gender)
# 得到orm的查询语句
orm_sql = select(Employee).from_statement(new_sql)
```

```
# list_emp中的元素是模型对象
list_emp = session.execute(orm_sql).scalars()
for o in list_emp:
    print(type(o))
    print(o)
```

4、修改操作

1) 先查询出来，再修改属性

```
 # 查询，再修改
 old_emp = session.get(Emp, 1)
 print(old_emp.name)
 old_emp.name = '李四'
```

2) 直接根据主键修改

```
# 注意where语句就是条件的意思，其中id前面的Employee必须有
session.execute(update(Employee).where(Employee.id ==
1).values(sal=Decimal(6000), bonus=Decimal(500)))
```

3) 直接根据主机批量修改

```
# 批量修改，必须根据主键来修改
session.execute(update(Employee), [
    {'id': 1, 'name': '张三三'},
    {'id': 2, 'name': '李四四'},
])
```

5、删除操作

1） 先查询，再删除

```
# 先查询出来，再删除
emp = session.get(Employee, 2)
session.delete(emp)
```

2) 直接删除

```
# 删除的数据，由where条件决定
session.execute(delete(Employee).where(Employee.id == 6))
```

6、查询条件

过滤是数据提取的一个很重要的功能，以下对一些常用的过滤条件进行解释，并且这些过滤条件都是只能通过where方法实现的：

1. equals： ==，或者.is_ 函数
2. not equals ：!= 或者 isnot函数
3. like & ilike [不区分大小写]：
4. 在某个集合中存在，in_函数， 或者notin_函数（不存在）
5. And 多条件组合

```
# 查询员工的名字中包含”四“ ，并且基本薪资大于3000的所有员工
where(Employee.name.like('%四%'), Employee.sal > 3000))

where(and_(Employee.name.like('%四%'), Employee.sal > 3000))
```

6. or多条件组合

```
where(or_(Employee.name.like('%四%'), Employee.sal > 3000))
```

7. 聚合函数
   - func.count：统计行的数量。
   - func.avg：求平均值。
   - func.max：求最大值。
   - func.min：求最小值。
   - func.sum：求和。

```
# 查询所以员工的数量
result = session.execute(select(func.count(Employee.id)))
```

8. 分页查询
    a. limit函数：可以限制查询的时候只查询前几条数据。 属top-N查询
    b. offset：可以限制查找数据的时候过滤掉前面多少条。可指定开始查询时的偏移量。

```
result = session.scalars(select(Employee).offset(0).limit(2))
```

9. 排序： order_by函数

```
result =
session.scalars(select(Employee).order_by(Employee.sal.desc()).offset(0).limit(
2))
```

10. 分组查询：group_by 和过滤函数 having

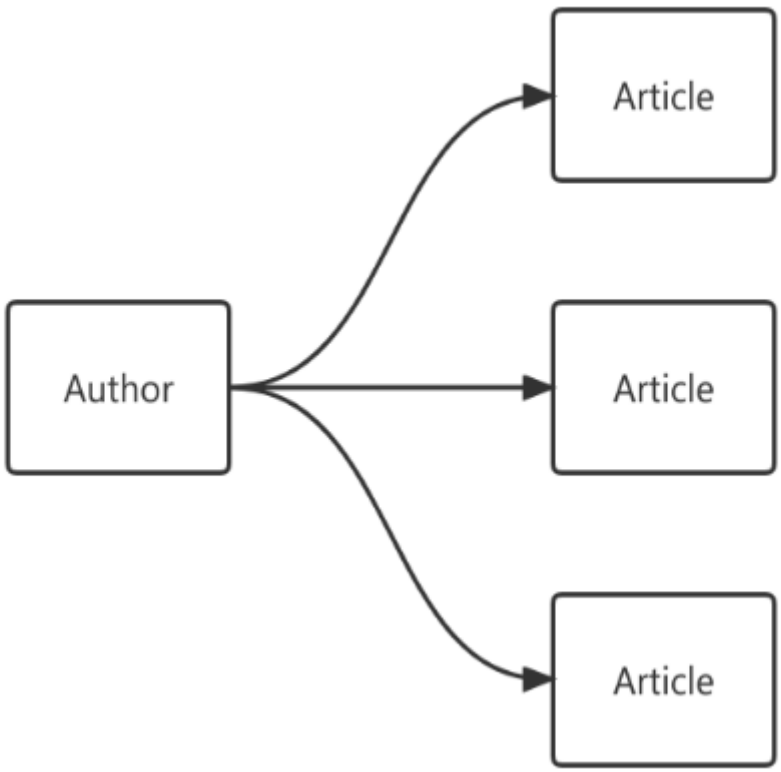
```
# 查询每个性别，各有多少员工
result = session.execute(select(Employee.gender,
func.count(Employee.id)).group_by(Employee.gender))
for o in result:
    # print(type(o))
    print(o)
```

第五节：模型类之间的关联关系

讲师：肖斌

一、一对多和多对一关联

比如：作者和文章之间， 部门和员工之间都是一对多的关联关系。反过来就是：多对一的关联关系。

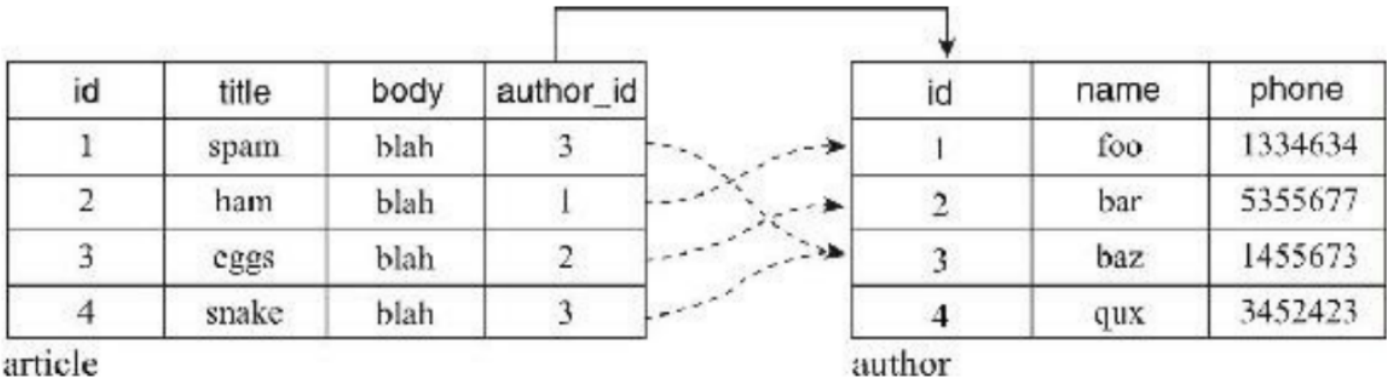


1、定义外键约束

定义关系的第一步是创建外键。外键是（foreign key）用来在 A 表存储 B 表的主键值以便和 B 表建立联系的关系字段。

因为外键只能存储单一数据（标量），**所以外键总是在 “多” 这一侧定义**，多篇文章属于同 一个作者，所以我们需要为每篇文章添加外键存储作者的主键值以指向 对应的作者。在 Article 模型中，我们定义一个 author_id 字段作为外键：

注意ForeginKey的参数是<表名>.<键名>，而不是<类名>.<字段名>



id	title	body	author_id
1	spam	blah	3
2	ham	blah	1
3	eggs	blah	2
4	snake	blah	3

article

id	name	phone
1	foo	1334634
2	bar	5355677
3	baz	1455673
4	qux	3452423

author

```
# 多对一关联的外键
dept_id: Mapped[Optional[int]] = mapped_column(ForeignKey('t_dept.id'))
```

2、定义关联属性和双向关联

我们在 `Author` 类中定义了集合关系属性 `articles`，用来获取某个作者拥有的多篇文章记录。在某些情况下，你也许希望能在 `Article` 类中定义 一个类似的 `author` 关系属性，当被调用时返回对应的作者记录，而这种两侧都添加关系属性获取对方记录的关系我们称之为 **双向关系。双向关系并不是必须的，但在某些情况下会非常方便。**

```
# 多对一关联的属性， back_populates写对方模型类中的关联属性名字
dept: Mapped[Optional['Dept']] = relationship(back_populates='emp_list')
```

```
 # 一对多的关联属性
 emp_list: Mapped[List['Employee']] = relationship(back_populates='dept',
cascade='save-update')
```

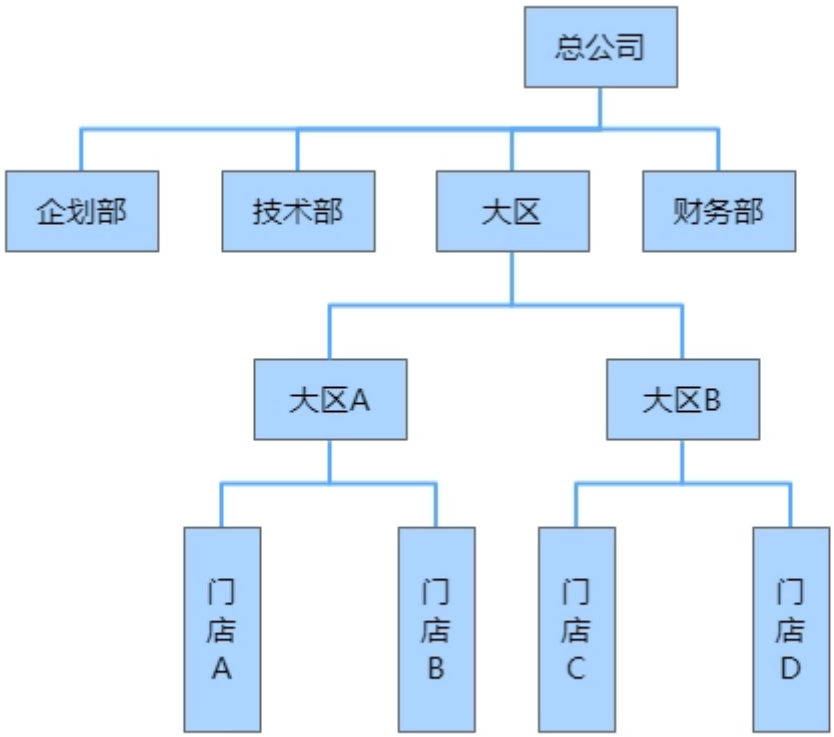
3、级联操作

cascade，默认选项为save-update：

- 一：save-update：默认选项，在添加一条数据的时候，会把其他和次数据关联的数据都添加到数据库中，这种行为就是save-update属性决定的
- 二：delete：表示当删除某一个模型中的数据的时候，也删除掉使用relationship和此数据关联的数据

- 三：delete-orphan：表示当对一个ORM对象解除了父表中的关联对象的时候，自己便会被删除，如果父表的数据被删除，同样自己也会被删除，这个选项只能用在一对多上，不能用在多对多和多对一上，并且使用的时候还需要在子模型的relationship中增加参数：single_parent=True
- 四：merge(合并)：默认选项，当在使用session.merge合并一个对象的时候，会将使用了relationship相关联的对象也进行merge操作
- 五：expunge：移除操作的时候，会将相关联的对象也进行移除，这个操作只是从session中移除，并不会正则从数据库删除
- 六：all：对 save-update、merge、refresh-expire、expunge、delete 这几种的缩写

4、树形结构的自关联




```
class Dept(Base):
    """部门的模型类"""
    __tablename__ = 't_dept'

    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    name: Mapped[str] = mapped_column(String(20), unique=True, nullable=False)
    address: Mapped[Optional[str]] = mapped_column(String(50))

    # 一对多的关联属性
```

```
    emp_list: Mapped[List['Employee']] = relationship(back_populates='dept',
cascade='save-update')

    # 自己和自己外键
    pid: Mapped[Optional[int]] = mapped_column(ForeignKey('t_dept.id'))
    # 自己和自己一对多的关联的属性
    children: Mapped[List['Dept']] = relationship(back_populates='parent')
    # 自己和自己多对一的关联的属性 remote_side = [id], 写到多的一端。（非列表）
    parent: Mapped[Optional['Dept']] = relationship(back_populates='children',
remote_side=[id])
```

维护一对多自关联关系的时候（relationship），必须加入：remote_side= [id]

二、一对一关联

一对一关系实际上是通过建立双向关系的一对多关系的基础上转化而来。

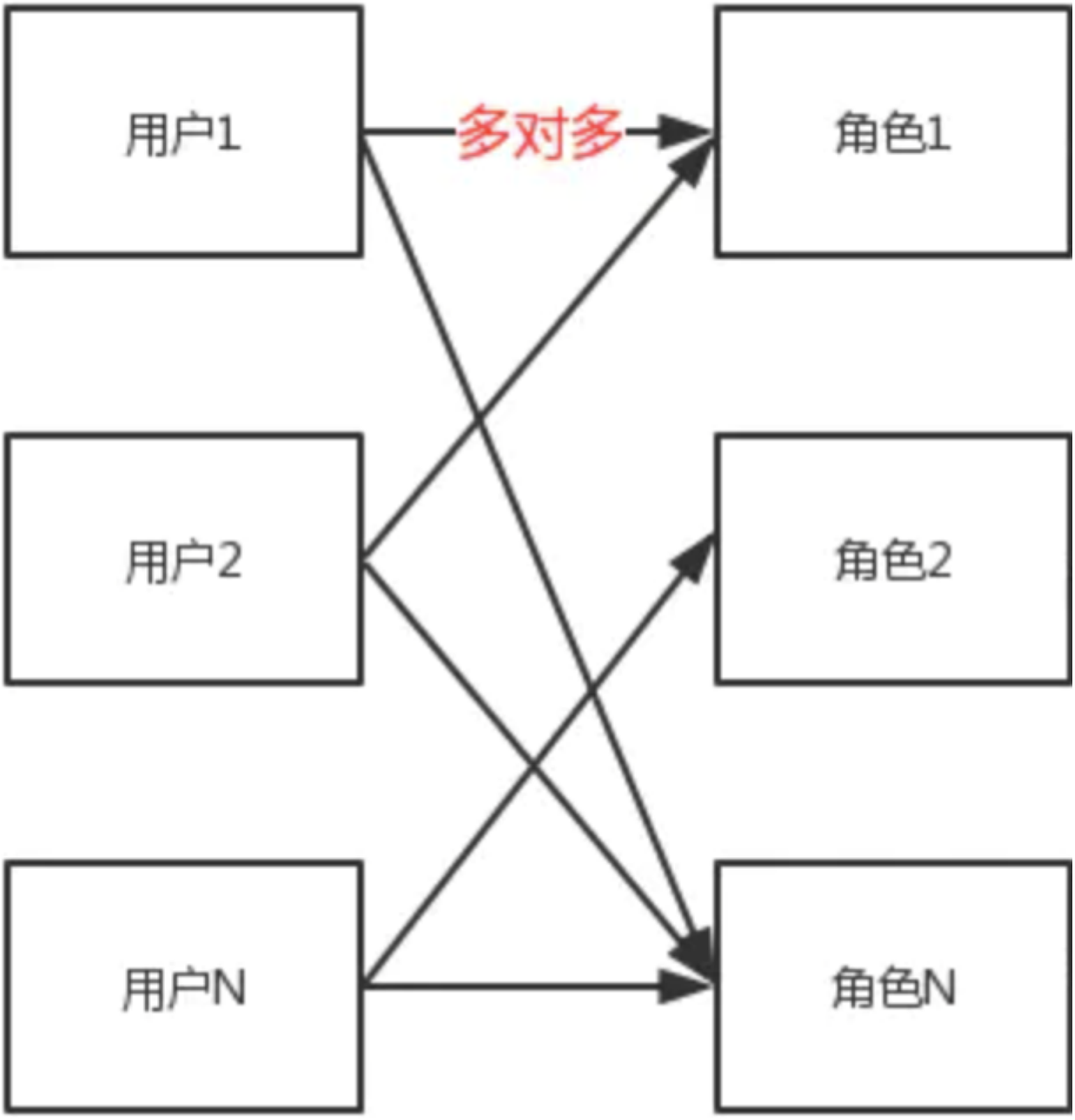
比如：一个用户对应一张身份证，一张身份证属于一个用户。

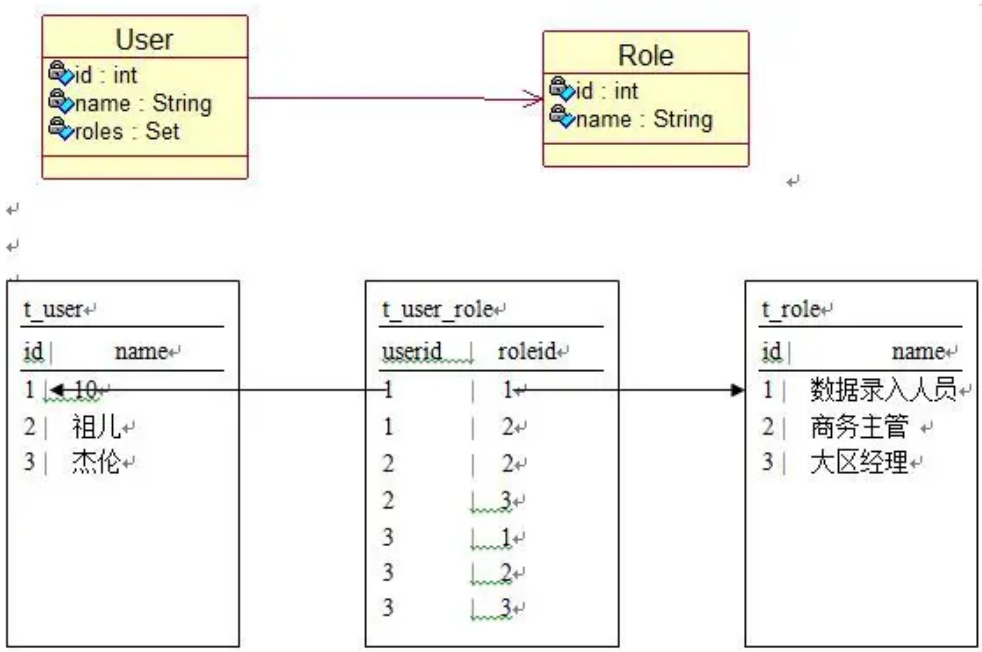

```
class IdCard(Base):
    """省份证的模型类， 它和员工之间是一对一的关联关系"""
    __tablename__ = 't_id_card'
    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    card_number: Mapped[str] = mapped_column(String(18), unique=True,
nullable=False, comment='省份证号码')

    origin: Mapped[Optional[str]] = mapped_column(String(50), comment='籍贯')

    # 外键
    emp_id: Mapped[int] = mapped_column(ForeignKey('t_emp.id'))
    # 和员工的关联属性
    emp: Mapped['Employee'] = relationship(single_parent=True,
back_populates='idc')
```

其中： `single_parent=True` **表示子表，只关联父表的一行记录。**

```
    # 和省份证的关联属性
    idc: Mapped[Optional['IdCard']] = relationship(back_populates='emp')
```

三、多对多关联





在 `SQLAlchemy` 中，要想表示多对多关系，除了关系两侧的模型外，我们还需要创建一个关联表（middle_table）。关联表不存储数据，只用来存储关系两侧模型的外键对应关系。

```
# 多对多关联，先定义中间表（没有对应的模型类）
middle_table = Table(
    't_user_role',
    Base.metadata,
    Column('user_id', ForeignKey('t_user.id'), primary_key=True),  # 联合主键
    Column('role_id', ForeignKey('t_role.id'), primary_key=True),
)


class User(Base):
    # 用户和角色之间是多对多关系， 一个用户可以拥有多个角色，一个角色可以所属多个用户
    """用户的模型类"""
    __tablename__ = 't_user'

    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    username: Mapped[str] = mapped_column(String(20), unique=True,
nullable=False)
    password: Mapped[str] = mapped_column(String(20), nullable=False)

    # 一个用户有多个角色
    roles: Mapped[Optional[List['Role']]] =
relationship(secondary=middle_table, back_populates='users')


class Role(Base):
    """角色的模型类"""
```

```
    __tablename__ = 't_role'

    id: Mapped[int] = mapped_column(primary_key=True, autoincrement=True)
    name: Mapped[str] = mapped_column(String(20), unique=True, nullable=False,
comment='角色的名字')

    # 一个角色被多有用户所拥有
    users: Mapped[Optional[List['User']]] =
relationship(secondary=middle_table, back_populates='roles')
```

四、关联查询

```
# 关联查询
# 1、查询2020年入职的员工姓名以及该员工的所在部门名称
result = session.execute(select(Employee.name, Dept.name).join(Dept,
isouter=True).where(extract('year', Employee.entry_date) == 2018))
# 2、 查询省份号码是：11111111111。 的员工的姓名，以及该员工所属的部门
result = session.execute(select(Employee.name, Dept.name).join(Dept,
isouter=True).join(IdCard).where(IdCard.card_number == '111111111111'))
# 3、 查询，每个部门名字、city以及该部门下面的员工个数
result = session.execute(select(Dept,
func.count(Employee.dept_id)).join(Employee, isouter=True).group_by(Dept.id))

# 4、查询拥有角色数量超过1的 用户名以及他所拥有的角色数量。
sub_stmt = select(User.username.label('username'),
func.count(Role.id).label('role_count')).join(User.roles).group_by(User.id).sub
query()
result = session.execute(select(sub_stmt.c.username,
sub_stmt.c.role_count).where(sub_stmt.c.role_count > 1))

# 5、 查询每个部门的名字以及它里面的员工数量，并且按照员工数量的个数降序排序。
result = session.execute(select(Dept,
func.count(Employee.dept_id).label('ec')).join(Employee,
isouter=True).group_by(Dept.id).order_by(desc('ec')))
```

第六节：FastAPI和SQLAlchemy的整合

讲师：肖斌

一、定义Pydantic模型

一般情况下，每一个功能模块都需要自己的数据模式（schemas）。数据模式（schemas）是FastAPI模块用于数据传递的对象，通过继承pydantic中的类建立。多个pydantic模型类组成了完整的schemas。

一旦我们定义了Pydantic模型，我们可以使用 `orm_mode` 参数将ORM模型转换为Pydantic模型。`orm_mode` 参数告诉Pydantic将ORM模型视为数据库记录（dict）而不是普通的Python对象。

我们可以将Pydantic模型应用到FastAPI中的路径参数、请求体和响应模型上。FastAPI会自动根据Pydantic模型生成文档，并进行输入数据的验证和输出数据的编码。

```
class EmpBase(BaseModel):
    id: Optional[int] = Field(description='员工的ID,可以不传', default=None)
    name: str = Field(description='员工的名字')
    gender: SexValue = Field(description='员工的性别', default=SexValue.MALE)
    sal: Optional[Decimal] = Field(description='基本薪资', default=0)
    bonus: Optional[int] = Field(description='奖金和津贴', default=0)
    is_leave: Optional[bool] = Field(description='是否离职', default=False)
    entry_date: Optional[date] = Field(description='入职时间')
    dept_id: Optional[int] = Field(description='员工所属的部门ID,可以不传',
default=None)

    class Config:
        from_attributes = True


class DeptBase(BaseModel):
    id: Optional[int] = Field(description='部门的ID,可以不传', default=None)
    name: str = Field(description='部门的名字')
    city: str = Field(description='部门所在的城市', default=None)


class EmpBo(EmpBase):
```

```
    dept: Optional[DeptBase] = Field(default=None)

    class Config:
        from_attributes = True
```

1、Config 类

此类 **Config** 用于为 **Pydantic** 提供配置。

```
    from_attributes = True
```

将告诉 **Pydantic** 模型读取数据，即它不是一个 **dict**，而是一个 **ORM** 模型。这样该 **Pydantic** 模型就会尝试从属性中获取它，如 id = db_model.id。

二、FastAPI和SQLAlchemy整合

通过注入，把session对象注入到视图函数中去

```
    @app.get("/test", response_class=HTMLResponse)
    def test(request: Request, name: Union[str, None], session: Session =
    Depends(get_session)):
        all_list = session.query(Employee).all()
        return templates.TemplateResponse("result.html", {"request": request,
    'emp_list': all_list})
```

```
    def get_session():
        session = Session(bind=engine)
        try:
            yield session
        finally:
            session.close()
```