FastAPI 进阶

中间件、依赖注入、ORM

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

使用中间件为每个请求前后添加统一的处理逻辑

01

多个接口

都需要验证用户身份

02

多个接口

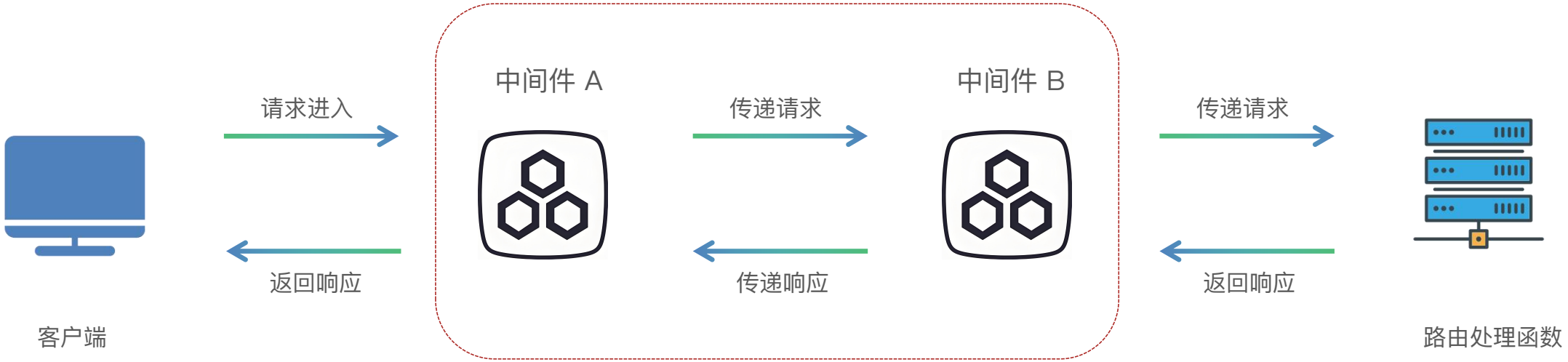
都需要记录日志、性能数据

中间件

中间件

中间件（Middleware）是一个在每次请求进入 FastAPI 应用时都会被执行的函数。

它在请求到达实际的路径操作（路由处理函数）之前运行，并且在响应返回给客户端之前再运行一次。



自下而上

中间件

中间件：函数的顶部使用装饰器 @app.middleware("http")

- 中间件作用是什么?

  为每个请求添加统一的处理逻辑（记录日志、身份认证、跨域、设置响应头、性能监控等）

- 中间件怎么定义?

  函数的顶部使用装饰器 @app.middleware("http")

```python
@app.middleware("http")
async def middleware(request, call_next):
    print('中间件开始处理 -- start')
    response = await call_next(request)
    print('中间件处理完成 -- end')
    return response
```

- 多个中间件的执行顺序是?

  自下而上

使用依赖注入系统来共享通用逻辑，减少代码重复

```python
@app.get('/news/news_list')
async def get_news_list(
  skip: int = Query(0, ge=0),
  limit: int = Query(10, le=60)
):
  # 分页逻辑...
  return {"list": "新闻列表"}
```

```python
@app.get("/users/user_list")
async def get_user_list(
  skip: int = Query(0, ge=0),
  limit: int = Query(10, le=60)
):
  # 分页逻辑...
  return {"users": "用户列表"}
```

```python
@app.get("/news/category")
async def get_category(
  skip: int = Query(0, ge=0),
  limit: int = Query(10, le=60)
):
  # 分页逻辑...
  return {"category": "新闻分类"}
```

简直不敢相信

单击此处添加标题

使用依赖注入系统来共享通用逻辑，减少代码重复

```python
@app.get('/news/news_list')
async def get_news_list(
  skip: int = Query(0, ge=0),
  limit: int = Query(10, le=60)
):
  # 分页逻辑...
  return {"list": "新闻列表"}
```

```python
@app.get("/users/user_list")
async def get_user_list(
  skip: int = Query(0, ge=0),
  limit: int = Query(10, le=60)
):
  # 分页逻辑...
  return {"users": "用户列表"}
```

```python
@app.get("/news/category")
async def get_category(
  skip: int = Query(0, ge=0),
  limit: int = Query(10, le=60)
):
  # 分页逻辑...
  return {"category": "新闻分类"}
```

简直不敢相信

使用依赖注入系统来共享通用逻辑，避免代码重复

依赖项：可重用的组件（函数/类），负责提供某种功能或数据。

注入：FastAPI 自动帮你调用依赖项，并将结果"注入"到路径操作函数中。

优点：

- 代码复用：一次编写，多处使用
- 解耦：业务逻辑与基础设施代码分离
- 易于测试：轻松地用模拟依赖替换真实依赖进行测试

依赖注入应用场景

处理请求参数

从请求中提取和验证参数（路径参数、查询参数、请求体）

共享业务逻辑

抽取封装多个路由公用的逻辑代码

共享数据库连接

管理数据库会话的创建、使用、关闭

安全和认证

验证用户身份、检查权限和角色要求等

使用依赖注入系统来共享通用逻辑，避免代码重复

创建依赖项

```
async def common_parameters(
  skip: int = Query(0, ge=0),
  limit: int = Query(10, le=60)
):
  return {
    "skip": skip,
    "limit": limit
  }
```

```
from fastapi import Depends
```

导入 Depends

声明依赖项

```
@app.get("news/news_list'")
async def get_news_list(
  commons = Depends(common_parameters)
):
  return commons
```

- FastAPI中，依赖注入系统有什么用?

  抽取可复用的组件，实现代码复用、解耦且可轻松替换依赖项进行测试

- 怎么使用依赖注入系统?

  创建依赖项 → 导入 Depends → 声明依赖项

```python
from fastapi import Depends

async def common_parameters(
  skip: int = Query(0, ge=0),
  limit: int = Query(10, le=60)
):
  return { "skip": skip, "limit": limit }

@app.get("news/news_list'")
async def get_news_list(
  commons = Depends(common_parameters)
):
  return commons
```

ORM 简介

ORM（Object-RelationalMapping，对象关系映射）是一种编程技术，用于在面向对象编程语言和关系型数据库之间建立映射。它允许开发者通过操作对象的方式与数据库进行交互，而无需直接编写复杂的SQL语句。

优势：

- 减少重复的 SQL 代码
- 代码更简洁易读
- 自动处理数据库连接和事务
- 自动防止 SQL 注入攻击

ORM 分类

排名	ORM 工具	特点	适应场景
1	SQLAlchemy ORM	功能最强、最灵活、企业级	各类 API、微服务、数据应用
2	Django ORM	封装好、上手快	Django 项目、管理后台
3	Tortoise ORM	全异步	异步 Web 服务、高并发 API

ORM 使用流程

- sqlalchemy[asyncio]
- aiomysql（异步数据库驱动）

1. 安装

- run_sync(Base.metadata.create_all)

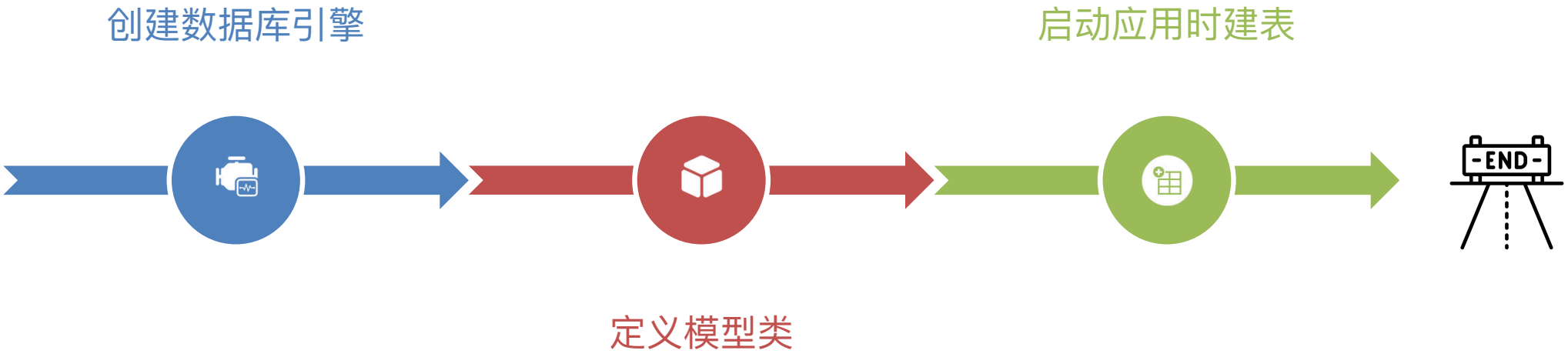
2. 建库、建表

- 查询
- 新增
- 修改
- 删除

3. 操作数据

```
create database xx;
```

ORM – 建表



ORM – 创建数据库引擎

使用 create_async_engine 创建异步引擎

```python
from sqlalchemy.ext.asyncio import create_async_engine

ASYNC_DATABASE_URL = "mysql+aiomysql://root:123456@localhost:3306/fastapi_test?charset=utf8"
# 创建异步引擎
async_engine = create_async_engine(
  ASYNC_DATABASE_URL,
  echo=True,        # 可选：输出SQL日志
  pool_size=10,     # 设置连接池中保持的持久连接数
  max_overflow=20   # 设置连接池允许创建的额外连接数
)
```

ORM – 定义模型类

1. 基类，继承 DeclarativeBase（包含通用属性和字段的映射）

2. 定义数据库表对应的模型类

```
class Base(DeclarativeBase):
  create_time: Mapped[datetime] = mapped_column(
    DateTime, insert_default=func.now(), default=datetime.now, comment="创建时间")
  update_time: Mapped[datetime] = mapped_column(
    DateTime, insert_default=func.now(), onupdate=func.now(), default=datetime.now, comment="修改时间")


class Book(Base):
  __tablename__="book"

  id: Mapped[int] = mapped_column(primary_key=True)
  bookname: Mapped[str] = mapped_column(String(255))
  author: Mapped[str] = mapped_column(String(255))
  ......
```

ORM – 创建数据库表

1. 从连接池获取异步连接，开启事务，执行 ORM 操作

2. FastAPI 应用启动时，创建数据库表

```python
async def create_tables():
  async with async_engine.begin() as conn:
    await conn.run_sync(Base.metadata.create_all)


@app.on_event("startup")
async def startup_event():
  await create_tables()
```

练习

- 创建数据表表 – 用户表

需求：使用 SQLAlchemy ORM 创建用户表，包含字段如下：用户 id、用户名、密码、创建时间、更新时间

- 用户 id：主键

ORM – 路由匹配中使用 ORM

ORM – 路由匹配中使用 ORM

核心：创建依赖项，使用 Depends 注入到处理函数

```python
# 创建异步会话工厂
AsyncSessionLocal = async_sessionmaker(
  bind=async_engine,  # 绑定数据库引擎
  class_=AsyncSession,  # 指定会话类
  expire_on_commit=False  # 会话对象不过期，不重新查询数据库
)
# 依赖项，用于获取数据库会话
async def get_database():
  async with AsyncSessionLocal() as session:
    try:
      yield session  # 返回数据库会话给路由处理函数
      await session.commit()  # 无异常，提交事务
    except Exception:
      await session.rollback()  # 有异常则回滚
      raise
    finally:
      await session.close()  # 关闭会话
```

```python
@app.get("/book/books")
async def get_book_list(
  db: AsyncSession = Depends(get_database)
):
  # 查询所有书籍
  result = await db.execute(select(Book))  # Book 模型类
  user = result.scalars().all()
  return user
```

ORM 使用流程

- sqlalchemy[asyncio]
- aiomysql（异步数据库驱动）

1. 安装

- run_sync(Base.metadata.create_all)

2. 建库、建表

- 查询
- 新增
- 修改
- 删除

3. 操作数据

数据库操作

01 查询

- select()

02 新增

- add()

03 更新

- 先查再改（重新赋值）

04 删除

- delete()

数据库操作 – 查询

核心语句：await db.execute(select(模型类))，返回一个 ORM 对象

- 获取所有数据
  - ✓ scalars().all()

```python
@app.get("/book/get_books")
async def get_book_list(db: AsyncSession=Depends(get_database)):
  result = await db.execute(select(Book))
  book = result.scalars().all()
  return book
```

- 获取单条数据
  - ✓ scalars().first()
  - ✓ get(模型类，主键值)

```python
@app.get("/book/get_book")
async def get_book(db: AsyncSession=Depends(get_database)):
  # result = await db.execute(select(Book))
  # book=result.scalars().first()
  book = await db.get(Book, 1)
  return book
```

数据库操作 – 查询条件

```
select(Book).where(条件, 条件2, ...)
```

条件：

- ✓ 比较判断：==; >; <; >=; <= 等
- ✓ 模糊查询：like()
- ✓ 与非查询：&; |; ~
- ✓ 包含查询：in_()

查询条件 – 比较判断

比较判断：==; >; <; >=; <= 等

```
@app.get("/book/{book_id}")
async def get_book_list(book_id: int, db: AsyncSession = Depends(get_database)):
  result = await db.execute(select(Book).where(Book.id == book_id))
  book = result.scalar_one_or_none()
  return book
```

数据库操作 – 查询条件

```
select(Book).where(条件, 条件2, ...)
```

条件：

- ✓ 比较判断：==; >; <; >=; <= 等
- ✓ 模糊查询：like()
- ✓ 与非查询：&; |; ~
- ✓ 包含查询：in_()

查询条件 – 模糊查询

模糊查询：like()

- %：零个、一个或多个字符
- _：一个单个字符

```
@app.get("/book/get_books")
async def get_book_list(db: AsyncSession = Depends(get_database)):
  result = await db.execute(select(Book).where(Book.author.like("曹%")))
  book = result.scalars().all()
  return book
```

查询条件 – 与非查询

与非查询：

- &：与
- |：或
- ~：非

```
@app.get("/book/get_books")
async def get_book_list(db: AsyncSession = Depends(get_database)):
  result = await db.execute(select(Book).where((Book.author == "曹雪芹") & (Book.price == 200)))
  book = result.scalars().all()
  return book
```

查询条件 – 包含查询

包含查询：

- in_()

```
@app.get("/book/get_books")
async def get_book_list(db: AsyncSession = Depends(get_database)):
  id_list = [1, 2, 3, 4, 5, 6]
  result = await db.execute(select(Book).where(Book.id.in_(id_list)))
  book = result.scalars().all()
  return book
```

数据库操作 – 聚合查询

聚合计算：func.方法(模型类.属性)

- count：统计行数量
- avg：求平均值
- max：求最大值
- min：求最小值
- sum：求和

```python
@app.get("/book/count")
async def get_count(db: AsyncSession = Depends(get_database)):
    # result = await db.execute(select(func.count(Book.id)))
    # result = await db.execute(select(func.max(Book.price)))
    # result = await db.execute(select(func.sum(Book.price)))
    result = await db.execute(select(func.avg(Book.price)))
    count = result.scalar()
    return count
```

数据库操作 – 分页查询

分页查询：select().offset().limit()

- offset：跳过的记录数
- limit：返回的记录数

当前页码	每页数量（limit）	跳过数量(offset)
1	10	0
2	10	10
3	10	20
4	10	30

offset 值 = (当前页码 – 1) * 每页数量 limit

```python
@app.get("/book/get_books")
async def get_book_list(
  page: int = 1,
  page_size: int = 3,
  db: AsyncSession = Depends(get_database)
):
  skip = (page-1) * page_size
  stmt = select(Book).offset(skip).limit(page_size)
  result = await db.execute(stmt)
  books = result.scalars().all()
  return {"books": books}
```

ORM – 查询 – 总结

核心思路：

- select() → db.execute() → 从 ORM 对象获取数据 → 响应结果
- db.get(模型类, 主键值)

条件查询 where()

- ✓ 比较判断
- ✓ 模糊查询 like()
- ✓ 与非查询：& | ~
- ✓ 包含查询：in_()

聚合查询 func

- ✓ count()
- ✓ max() 和 min()
- ✓ avg()
- ✓ sum()

分页查询

- ✓ select().offset().limit()
- ✓ offset值 = （当前页码 – 1）* 每页数量

ORM – 查询 – 总结

从 ORM 对象获取数据的方式

- 获取所有数据
  - ✓ `scalars().all()`

- 获取单条数据
  - ✓ `scalars().first()`: 提取第一个数据
  - ✓ `scalar_one_or_none()`: 提取一个或 null
  - ✓ `scalar()`: 提取标量值（配合聚合查询使用）

数据库操作

01 查询

- select()

02 新增

- add()

03 更新

- 先查再改（重新赋值）

04 删除

- delete()

数据库操作 – 新增

核心步骤：定义 ORM 对象 → 添加对象到事务：add(对象) → commit 提交到数据库

```python
@app.post("/book/add_book")
async def add_book(book: BookBase, db: AsyncSession = Depends(get_database)):
    # 获取 book 参数，创建图书对象（__dict__ 返回 book 对象的属性字典）
    book_obj = Book(**book.__dict__)
    db.add(book_obj)
    await db.commit()
    return book
```

数据库操作

01 查询

- select()

02 新增

- add()

03 更新

- 先查再改（重新赋值）

04 删除

- delete()

数据库操作 – 更新

核心步骤：查询 get → 属性重新赋值 → commit 提交到数据库

```python
@app.put("/book/update_book/{book_id}")
async def update_book(book_id: int, data: BookUpdate, db: AsyncSession = Depends(get_database)):
    # 1. 查询
    book = await db.get(Book, book_id)
    if book is None:
        raise HTTPException(status_code=404, detail="Book not found")
    # 2. 修改属性（重新赋值）
    book.bookname = data.bookname
    book.author = data.author
    book.price = data.price
    # 3. 提交
    await db.commit()
    return book
```

数据库操作

01 查询

- select()

02 新增

- add()

03 更新

- 先查再改（重新赋值）

04 删除

- delete()

数据库操作 – 删除

核心步骤：查询 get → delete 删除 → commit 提交到数据库

```python
@app.delete("/book/delete_book/{book_id}")
async def delete_book(book_id: int, db: AsyncSession = Depends(get_database)):
  db_book = await db.get(Book, book_id)
  if db_book is None:
    raise HTTPException(status_code=404, detail="Book not found")
  await db.delete(db_book)
  await db.commit()
  return {"message": "Book deleted"}
```

面向对象的方式操作数据库

不写 SQL 啦

企业应用最多、社区最活跃

SQLAlchemy ORM

- sqlalchemy[asyncio]
- aiomysql（异步数据库驱动）

安装

1. create_async_engine 创建异步引擎
2. 定义模型基类（继承 DeclarativeBase）和 模型类
3. run_sync(Base.metadata.create_all) 建表

建表

1. 路由处理函数注入数据库会话依赖项 Depends
2. 查询数据：select()
3. 增加数据：add()
4. 更新数据：重新赋值
5. 删除数据：delete()

操作数据


小意思


看好你奥

传智教育旗下高端IT教育品牌