

项目结构学习指南

目标读者：正在学习专业开发实践的初学者 项目：Todo API (Python + FastAPI + SQLAlchemy) 最后更新：2026-02-14

目录

- [一、为什么项目结构很重要？](#)
- [二、完整项目结构一览](#)
- [三、逐层解析](#)
 - [3.1 根目录：项目的"门面"](#)
 - [3.2 src/ 目录：源代码的家](#)
 - [3.3 app/ 包：应用主体](#)
 - [3.4 models/ 包：数据库模型层](#)
 - [3.5 schemas/ 包：数据验证层](#)
 - [3.6 routers/ 包：路由层](#)
 - [3.7 tests/ 目录：测试代码](#)
- [四、核心设计原则](#)
- [五、专业开发者应当思考的问题](#)
- [六、常见错误与反模式](#)
- [七、进阶：项目结构的扩展](#)

一、为什么项目结构很重要？

想象你走进一个图书馆：

- **好的图书馆：**书按类别分区，有目录索引，你能在 30 秒内找到你要的书。
- **差的图书馆：**所有书堆在一起，从门口到角落没有任何分类。

项目结构就是你代码的“图书馆布局”。好的结构带来：

好处	说明
可读性	新人加入团队后，能快速理解项目组织方式
可维护性	需要修改某个功能时，能立即定位到对应的文件
可扩展性	添加新功能时，有明确的地方放新代码，不会混乱
可测试性	代码分层清晰，每一层都可以独立测试
团队协作	不同人可以同时修改不同模块，减少代码冲突

💡 专业开发者的思维

“代码是写给人看的，顺便能在计算机上运行。”

你写的代码，6个月后你自己回来看，就是“另一个人”了。如果连自己都看不懂，那项目就无法长期维护。

二、完整项目结构一览

todo-api/	← 项目根目录
└── .git/	← Git 版本控制数据（自动管理，不要手动修改）
└── .gitignore	← Git 忽略规则：哪些文件不要纳入版本控制
└── .env	← 环境变量文件（敏感配置，不提交到 Git）
└── .venv/	← Python 虚拟环境（自动生成，不提交到 Git）
└── requirements.txt	← 项目依赖清单（第三方库列表）
└── README.md	← 项目说明文档（项目的“名片”）
└── docs/	← 项目文档（你正在看的就是这里！）
└── 01-project-structure-guide.md	
└── 02-development-step-guide.md	
└── src/	← 所有源代码
└── app/	← 应用主包
└── __init__.py	← 包标识文件
└── main.py	← 应用入口（“总指挥”）
└── config.py	← 配置管理（环境变量读取）
└── database.py	← 数据库连接与会话管理
└── models/	← 数据库模型层（表结构定义）
└── __init__.py	
└── todo.py	← Todo 表的模型
└── schemas/	← 数据验证层（API 输入/输出格式）
└── __init__.py	
└── todo.py	← Todo 的请求/响应 Schema
└── routers/	← 路由层（API 端点定义）
└── __init__.py	
└── todo.py	← Todo 相关的所有端点
└── tests/	← 测试代码
└── __init__.py	
└── conftest.py	← 测试公共配置

三、逐层解析

3.1 根目录：项目的“门面”

根目录放的都是**项目级别的**配置和说明文件，而不是源代码。

.gitignore — 版本控制的“黑名单”

```
# Python 编译缓存 (运行时自动生成, 无需跟踪)
__pycache__/
*.pyc

# 虚拟环境 (每个开发者自己创建, 不纳入版本控制)
.venv/

# 环境变量 (包含敏感信息! )
.env

# 数据库文件 (开发数据, 不同环境不同)
*.db
```

为什么这些文件不应该提交到 Git?

文件	原因
.venv/	体积大, 且与操作系统相关。Windows 和 Mac 的虚拟环境不通用
.env	包含密码、API 密钥等敏感信息, 泄漏到 GitHub 会有安全风险
__pycache__/	Python 自动生成的编译缓存, 每个人的都不同
*.db	数据库文件是运行时数据, 不是代码

 **关键认知:** .gitignore 应该在项目创建之初就设置好, 而不是事后补救。一旦敏感文件被提交过一次, 即使后来删除, 它仍然存在于 Git 历史记录中。

.env — 环境变量文件

```
APP_NAME=Todo API
DATABASE_URL=sqlite+aiosqlite://./todo.db
DEBUG=true
```

核心理念: 配置与代码分离

这个原则来自 [The Twelve-Factor App](#) (十二要素应用), 它是现代 Web 应用开发的行业标准方法论。其中第三条就是:

"将配置存储在环境变量中"

为什么不直接写在代码里?

```
# ❌ 不好的做法: 硬编码
DATABASE_URL = "postgresql://admin:my_secret_password@prod-db.example.com/mydb"

# ✅ 好的做法: 从环境变量读取
DATABASE_URL = os.environ.get("DATABASE_URL")
```

如果你把密码写在代码里并推送到 GitHub：

1. 任何人都能看到你的数据库密码
2. 之后的每个人克隆代码都能看到
3. 即使你删除了这行代码，Git 历史中依然存在

requirements.txt — 依赖清单

```
fastapi==0.129.0
uvicorn==0.40.0
sqlalchemy==2.0.46
...
```

为什么要锁定版本号（==）？

假设你的项目依赖 fastapi：

- 你开发时用的是 0.129.0
- 三个月后，你的同事 pip install fastapi，装到了 0.135.0
- 新版本可能改了某些行为，导致你的代码出 Bug

锁定版本号确保每个人在任何时间安装，得到的都是完全一样的依赖版本。这叫“可复现的构建”。

README.md — 项目名片

一个好的 README 至少包含：

1. 项目是什么
2. 如何安装和运行
3. 如何运行测试
4. API 文档在哪里
5. 项目结构说明

💡 **关键认知：**很多开源项目，别人决定要不要用、要不要参与贡献，看的就是 README。它是你项目的第一印象。

3.2 src/ 目录：源代码的家

为什么不直接把代码放在根目录？

```
# ❌ 不推荐：源代码和配置文件混在一起
todo-api/
├── main.py
├── config.py
├── models.py
└── requirements.txt
└── .gitignore
```

```
└ ...  
# ✓ 推荐：源代码集中在 src/ 下  
todo-api/  
├── src/  
│   └── app/  
│       ├── main.py  
│       └── ...  
└── tests/  
└── requirements.txt  
└ ...
```

好处：

1. **清晰的边界**：一眼就能分辨哪些是源代码、哪些是配置、哪些是测试
2. **避免导入冲突**：src/ 下的代码不会意外地被当作根目录的模块导入
3. **打包友好**：将来如果要把项目发布为 Python 包，src/ 布局是推荐做法

3.3 app/ 包：应用主体

__init__.py — Python 包的“身份证”

```
# app/__init__.py
```

这个文件的存在告诉 Python：“这个目录是一个 Python 包（Package），可以被 import”。

没有它：

```
from app.config import settings # ✗ ModuleNotFoundError!
```

有了它：

```
from app.config import settings # ✓ 正常工作
```

注意：在 Python 3.3+ 中，__init__.py 技术上不是必须的（有“命名空间包”机制），但**显式总比隐式好**——这是 Python 之禅的一条原则。加上它是行业标准做法。

main.py — 应用入口（“总指挥”）

```
app = FastAPI(...)          # 创建应用实例  
app.include_router(...)     # 注册路由
```

main.py 的职责仅限于“组装”：

- 创建 FastAPI 实例
- 注册路由（把各模块连接起来）
- 管理应用生命周期（启动/关闭）

它不应该包含具体的业务逻辑（比如数据库查询）。这就是“关注点分离”。

config.py — 配置管理

```
class Settings(BaseSettings):
    APP_NAME: str = "Todo API"
    DATABASE_URL: str = "sqlite+aiosqlite:///./todo.db"
```

为什么用一个类，而不是直接用变量？

```
# ❌ 不好：散落各处的配置
DATABASE_URL = os.getenv("DATABASE_URL")
APP_NAME = os.getenv("APP_NAME")

# ✅ 好：集中管理的配置类
class Settings(BaseSettings):
    DATABASE_URL: str = "..."
    APP_NAME: str = "..."
```

类的好处：

1. **类型检查**：写了 `DEBUG: bool`，如果 `.env` 里写了 `DEBUG=abc`，启动时就会报错
2. **默认值**：没有设置环境变量时使用默认值
3. **集中管理**：所有配置一目了然，不用到处找
4. **IDE 支持**：用 `settings` 时，IDE 能自动提示所有可用配置

database.py — 数据库连接

这个文件有三个核心概念：

```
Engine (引擎) → Session Factory (会话工厂) → Session (会话)
"大门"           "制造机"           "工作台"
```

为什么用 `yield` 而不是 `return`？

```
async def get_db():
    async with async_session() as session:
        yield session # ← 注意这里是 yield, 不是 return
```

`yield`让这个函数变成一个生成器，它的特殊之处在于：

- `yield`之前的代码 → 在请求开始时执行（创建会话）
- `yield`返回的值 → 请求处理过程中使用
- `yield`之后的代码 + `async with` 的清理 → 在请求结束时执行（关闭会话）

这保证了无论请求成功还是失败，会话一定会被关闭，不会造成资源泄漏。

3.4 `models/` 包：数据库模型层

```
class Todo(Base):
    __tablename__ = "todos"
    id: Mapped[int] = mapped_column(Integer, primary_key=True)
    title: Mapped[str] = mapped_column(String(200), nullable=False)
    ...
```

这就是 ORM 的核心：一个 Python 类 = 一张数据库表

Python 世界	数据库世界
<code>class Todo</code>	<code>CREATE TABLE todos</code>
<code>id: Mapped[int]</code>	<code>id INTEGER</code>
<code>primary_key=True</code>	<code>PRIMARY KEY</code>
<code>nullable=False</code>	<code>NOT NULL</code>
<code>String(200)</code>	<code>VARCHAR(200)</code>
<code>todo = Todo(title="买菜")</code>	<code>INSERT INTO todos ...</code>
<code>todo.title</code>	<code>SELECT title FROM todos</code>

`created_at` 和 `updated_at` — 为什么每张表都应该有时间戳？

这是专业数据库设计的标准做法：

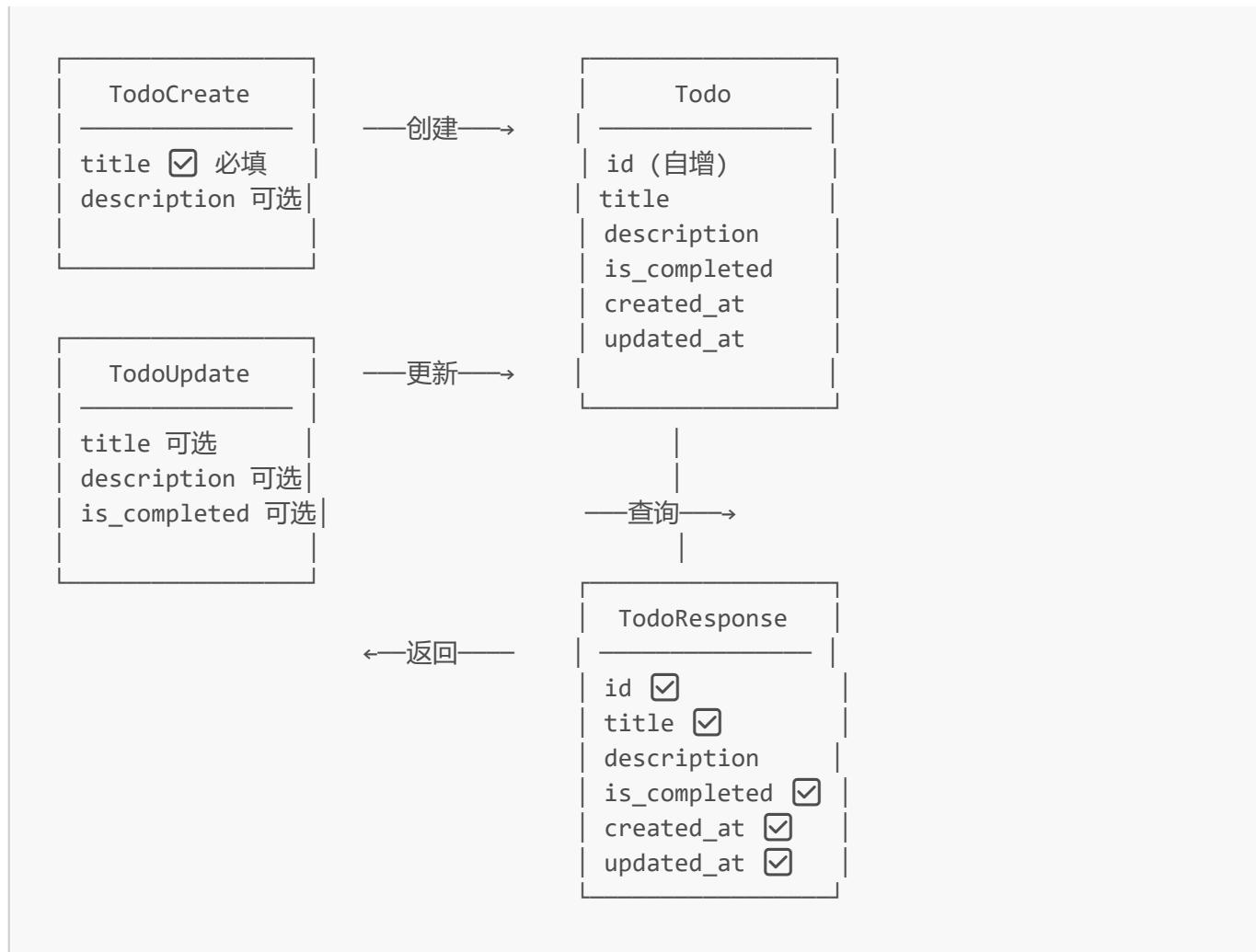
1. 审计追踪：知道每条数据何时创建、何时最后修改
2. 调试定位：出了 Bug，可以按时间线排查“这条数据是几点被改的？”
3. 业务需求：按创建时间排序、显示“最近更新”等功能都需要时间戳
4. 数据恢复：如果数据被误改，时间戳能帮你确定出问题的时间范围

3.5 `schemas/` 包：数据验证层

Models vs Schemas —— 最容易混淆的概念

`schemas/todo.py`
(面向 API 用户)

`models/todo.py`
(面向数据库)



为什么需要三个不同的 Schema?

因为同一个资源 (Todo) , 在不同操作中需要的数据不同:

操作	用户需要发送	系统返回
创建	title (必填) , description (可选)	完整的 Todo + id + 时间戳
更新	任意字段 (都可选)	更新后的完整 Todo
查询	无	完整的 Todo

如果只用一个 Schema, 那创建时 `id` 就变成必填了 (因为 `TodoResponse` 里 `id` 是必填的) , 但创建时 `id` 应该是数据库自动生成的。所以必须分开。

3.6 routers/ 包: 路由层

```

router = APIRouter(prefix="/todos", tags=["Todos"])

@router.post("/")          # POST /todos
@router.get("/")           # GET /todos
@router.get("/{todo_id}") # GET /todos/123
@router.patch("/{todo_id}") # PATCH /todos/123
@router.delete("/{todo_id}") # DELETE /todos/123
  
```

RESTful API 设计规范

REST (Representational State Transfer) 是一种 API 设计风格，核心思想是：

用 URL 表示资源，用 HTTP 方法表示操作

HTTP 方法	含义	URL 示例	操作
GET	获取资源	GET /todos	获取所有 Todo
GET	获取资源	GET /todos/1	获取 id=1 的 Todo
POST	创建资源	POST /todos	创建新 Todo
PATCH	部分更新	PATCH /todos/1	更新 id=1 的 Todo (部分字段)
PUT	完整替换	PUT /todos/1	替换 id=1 的 Todo (需要全部字段)
DELETE	删除资源	DELETE /todos/1	删除 id=1 的 Todo

为什么 routers/ 是一个目录而不是一个文件？

现在只有 `todo.py`，看起来放一个目录有点“杀鸡用牛刀”。但想想未来，你的 API 可能会扩展：

```
routers/
├── todo.py      # /todos/*
├── user.py      # /users/*
└── tag.py       # /tags/*
   └── auth.py    # /auth/*           ← 未来添加用户系统
                           ← 未来添加标签功能
                           ← 未来添加认证功能
```

提前按目录组织，就不需要在扩展时重构文件结构。这就是“为扩展设计”。

3.7 tests/ 目录：测试代码

```
tests/
└── __init__.py
   └── conftest.py      # pytest 的共享配置文件
```

为什么 tests/ 和 src/ 分开？

- 部署时不需要测试代码：生产服务器上只需要 `src/` 里的代码
- 关注点分离：测试代码和业务代码各占各的地方
- 不同的依赖：测试可能需要额外的库（如 `pytest`, `httpx`），但生产环境不需要

conftest.py 是什么？

这是 `pytest` 的特殊文件——“共享工具箱”。里面定义的 fixture（测试工具）可以被所有测试文件自动使用，无需 `import`。

四、核心设计原则

4.1 关注点分离 (Separation of Concerns)

每个模块只负责一件事

```
用户请求 → Router (处理路由) → Schema (验证数据) → Model (操作数据库) → 返回响应  
"接待员"           "安检员"           "仓库管理员"
```

好处：修改数据库结构时，只需要改 `models/`，不需要改 `routers/`。

4.2 依赖注入 (Dependency Injection)

```
async def create_todo(  
    todo_in: TodoCreate,                      # ← FastAPI 自动解析请求体  
    db: AsyncSession = Depends(get_db),          # ← FastAPI 自动注入数据库会话  
):
```

你不需要手动创建数据库会话、手动关闭——FastAPI 帮你管理。这让你的路由函数**只关注业务逻辑**，不关心基础设施。

4.3 约定优于配置 (Convention over Configuration)

- `__init__.py` → Python 约定的包标识
- `main.py` → Python 约定的入口文件
- `conftest.py` → pytest 约定的配置文件
- `requirements.txt` → pip 约定的依赖文件

遵循约定意味着：别人看到你的项目，不需要任何说明就知道每个文件该干嘛。

五、专业开发者应当思考的问题

⌚ Level 1：基础理解（必须掌握）

- 能否不看参考，画出项目的目录结构？
- 能否解释每个文件的作用（一句话概括）？
- `models/` 和 `schemas/` 的区别是什么？
- 为什么 `.env` 不能提交到 Git？
- `requirements.txt` 的作用是什么？
- `__init__.py` 为什么存在？

⌚ Level 2：设计思维（应该理解）

- 如果要增加一个“用户”功能，需要添加哪些文件？（答案：`models/user.py`, `schemas/user.py`, `routers/user.py`，然后在 `main.py` 注册路由）

- 为什么 config.py 用类而不是普通变量?
- database.py 中的 yield 有什么特殊含义?
- 为什么 TodoUpdate 的所有字段都是可选的?
- REST API 中 PATCH 和 PUT 的区别是什么?

㊂ Level 3：专业判断（逐步培养）

- 什么时候应该把一个文件拆分成一个目录?
- 如果有 50 个 API 端点，你会如何组织代码?
- 除了 models/, schemas/, routers/，还可能需要哪些层？（提示：services/ 业务逻辑层、middleware/ 中间件层、utils/ 工具函数层）
- 如何在不改动代码的情况下，让同样的代码在开发环境和生产环境使用不同的数据库?
- 如果数据库表结构需要修改（比如给 Todo 加一个 priority 字段），应该怎么做？（提示：数据库迁移工具 Alembic）

六、常见错误与反模式

✗ 反模式 1：所有代码塞在一个文件里

```
# ✗ 一个 main.py 写了 1000 行，包含模型、路由、数据库连接...
```

问题：难以阅读、难以测试、多人协作时必然冲突。

✗ 反模式 2：硬编码配置

```
# ✗ 数据库密码写死在代码里
engine = create_engine("postgresql://root:password123@localhost/mydb")
```

问题：安全隐患，环境切换困难。

✗ 反模式 3：不使用虚拟环境

```
# ✗ 直接全局安装
pip install fastapi
```

问题：不同项目的依赖互相干扰，离开你的电脑就无法复现。

✗ 反模式 4：先写代码，最后补结构

```
# ✗ 先在根目录写一大堆 .py 文件，后来才想起来要整理
```

问题：重构文件结构后，所有 import 语句都要改，非常痛苦。 **正确做法：**项目开始时就规划好结构。

✖ 反模式 5：提交 .venv/ 或 .env 到 Git

问题：.venv/ 动辄几十 MB，.env 包含敏感信息。

七、进阶：项目结构的扩展

随着项目复杂度增长，你可能需要添加以下目录：

```
src/app/
└── services/          # 业务逻辑层（复杂的业务规则）
    └── todo.py        # 例如：自动给过期的 Todo 发提醒

└── middleware/        # 中间件（请求/响应的“拦截器”）
    └── logging.py    # 例如：记录每个请求的耗时

└── utils/             # 工具函数（通用的辅助函数）
    └── pagination.py # 例如：统一的分页处理

└── exceptions/       # 自定义异常
    └── todo.py       # 例如：TodoNotFoundException

└── migrations/        # 数据库迁移脚本（Alembic 管理）
    └── versions/     # 每次表结构变更生成一个迁移脚本
```

什么时候添加 services/ 层？

当 routers/ 里的函数变得越来越复杂时：

```
# ✖ Router 里塞满了业务逻辑
@router.post("/")
async def create_todo(todo_in: TodoCreate, db = Depends(get_db)):
    # 验证用户权限...
    # 检查是否有重复标题...
    # 创建 Todo...
    # 发送通知邮件...
    # 记录操作日志...
    pass

# ✓ Router 只做“分发”，业务逻辑放在 Service 里
@router.post("/")
async def create_todo(todo_in: TodoCreate, db = Depends(get_db)):
    return await todo_service.create(db, todo_in)
```

❖ **记住：**好的项目结构不是一次性设计出来的，而是随着项目成长逐步演化的。但**基础结构**（models, schemas, routers 的分离）从一开始就应该做对。这就像盖房子——地基打好了，后面加层就容易；地基歪了，整栋楼都得推倒重来。