

🔧 项目规范化学习指南：Git、文档、测试与日志

目标读者：正在学习专业开发实践的初学者 **前置知识：**已完成项目结构搭建和核心代码编写（见前两篇指南）**最后更新：**2026-02-14

目录

- **一、Git 版本控制：你的代码"时光机"**
 - 1.1 为什么需要 Git
 - 1.2 Git 的核心工作流
 - 1.3 提交信息规范 (Conventional Commits)
 - 1.4 应当培养的习惯
- **二、README 与 .env.example：项目的"名片"与"说明书"**
 - 2.1 README.md 的标准结构
 - 2.2 .env.example 的作用
 - 2.3 应当培养的习惯
- **三、自动化测试：你的代码"安全网"**
 - 3.1 为什么测试如此重要
 - 3.2 测试基础设施详解
 - 3.3 AAA 测试模式
 - 3.4 我们写了哪些测试
 - 3.5 应当培养的习惯
- **四、日志系统：代码的"黑匣子"**
 - 4.1 为什么需要日志
 - 4.2 日志级别
 - 4.3 中间件模式
 - 4.4 应当培养的习惯
- **五、专业开发者的工作节奏**
- **六、自测清单**

一、Git 版本控制：你的代码"时光机"

1.1 为什么需要 Git

想象你在写一篇论文：

```
论文_v1.doc
论文_v2.doc
论文_v2_修改版.doc
论文_v2_修改版_最终版.doc
论文_v2_修改版_最终版_真的最终版.doc    ← 崩溃了吧?
```

Git 就是解决这个问题的工具。它帮你：

功能	说明	类比
版本记录	每次提交都是一个快照，可以随时回到任何历史版本	游戏存档
变更追踪	精确到每一行代码的增删改	Word 的修订模式
多人协作	多人同时修改同一个项目，不会互相覆盖	Google Docs 协作
分支管理	在不影响主代码的情况下实验新功能	平行宇宙

1.2 Git 的核心工作流

```

你修改了代码
|
↓
git add .           ← 第一步：把修改"放入暂存区"（选好要提交的文件）
|
↓
git commit -m "..." ← 第二步：正式提交（生成一个快照）
|
↓
(本地仓库保存了这个快照)

```

为什么分两步？为什么不直接提交？

因为你可能修改了 10 个文件，但只想提交其中 3 个。`git add` 让你精确选择“这次提交哪些修改”。

```

# 添加所有修改
git add .

# 只添加某个文件
git add src/app/main.py

# 查看当前状态（哪些文件被修改了、哪些在暂存区）
git status

```

1.3 提交信息规范 (Conventional Commits)

我们的项目使用了约定式提交规范，看看实际的提交记录：

```

e9f53bd feat: add request logging middleware      ← 新功能
3c7a1e2 test: add comprehensive api endpoint tests ← 测试
5eb8e65 docs: add README and .env.example       ← 文档
366cf8a feat: initialize todo api with full crud endpoints ← 新功能
eadb8bb chore: initial project setup            ← 杂务

```

格式：类型：简短描述

类型	含义	使用场景
feat	新功能	添加新的 API 端点、新的业务逻辑
fix	修复 Bug	修复了一个已知的错误
docs	文档	修改了 README、注释、指南
test	测试	添加或修改测试用例
chore	杂务	修改构建流程、更新依赖、调整配置
refactor	重构	改进代码结构，但不改变功能
style	代码风格	格式化、缩进、空行（不影响逻辑）

为什么要有这个规范？

1. **一眼看出改了什么**: 看到 fix: 就知道是修 Bug, 看到 feat: 就知道是新功能
2. **方便检索**: 想找某个 Bug 什么时候修的? 搜索 fix: 就行
3. **团队统一**: 所有人的提交记录风格一致, 可读性强
4. **自动化**: 很多工具能根据 feat: 和 fix: 自动生成版本日志

怎么写好提交信息？

```
# ❌ 不好的提交信息
git commit -m "修改了一些代码"
git commit -m "update"
git commit -m "fix bug"
git commit -m "asdfgh"

# ✅ 好的提交信息
git commit -m "feat: add pagination to todo list endpoint"
git commit -m "fix: return 404 when todo not found"
git commit -m "docs: add API usage examples to README"
```

原则：别人（或 6 个月后的你自己）看到这条提交信息，能不能立即明白改了什么？

1.4 应当培养的习惯

⌚ 习惯 1：小步提交，频繁提交

不要写了一整天代码才提交一次。每完成一个小功能、修复一个 Bug、写完一个测试，就应该提交。这样如果出了问题，你可以精确地回退到出问题之前的版本，而不是丢失一整天的工作。

⌚ 习惯 2：每次提交前先 git status

养成习惯：在 git add 之前先看看 git status，确认你要提交的是正确的文件。避免不小心把 .env（密码）或 todo.db（数据库）提交上去。

⌚ 习惯 3：提交的代码必须是“能跑的”

不要提交一个编译/运行都通不过的代码。每次提交应该是一个完整的、可工作的状态。如果你在做一个大功能写到一半，可以先不提交，等到能跑了再提交。

🔗 习惯 4：认真写提交信息

今天觉得 "fix bug" 足够了，三个月后你会后悔："到底修的哪个 Bug？" 多花 10 秒写清楚，省下将来 10 分钟的困惑。

二、README 与 .env.example：项目的"名片"与"说明书"

2.1 README.md 的标准结构

我们项目的 README 包含了以下部分：

```
# 项目名称      ← 是什么
## 技术栈       ← 用了什么技术
## 快速开始     ← 怎么在本地跑起来（最重要！）
    ### 克隆项目
    ### 创建虚拟环境
    ### 安装依赖
    ### 配置环境变量
    ### 启动服务
## API 端点     ← 有哪些接口
## 项目结构     ← 代码怎么组织的
## 运行测试     ← 怎么跑测试
## 许可证        ← 开源协议
```

一个好 README 的黄金标准：

一个从没见过你项目的开发者，按照 README 的步骤，能在 5 分钟内把项目跑起来。如果做不到，说明 README 不够好。

2.2 .env.example 的作用

我们创建了两个文件：

文件	提交到 Git?	作用
.env	✗ 不提交	真正的配置文件，可能包含密码
.env.example	<input checked="" type="checkbox"/> 提交	配置模板，告诉别人需要配置哪些变量

工作流程：

1. 新开发者克隆项目
2. 看到 .env.example，知道需要配置哪些环境变量
3. 复制一份：cp .env.example .env
4. 填入自己环境的值
5. 开始开发

为什么要这样做？

.env 里可能有数据库密码、第三方 API 密钥等敏感信息，绝对不能提交到 Git。但新成员需要知道“我该配置哪些变量”——.env.example 就是这个“模板”。

2.3 应当培养的习惯

⌚ 习惯 5：代码和文档同步更新

添加了新的 API 端点？立刻更新 README 的端点列表。添加了新的环境变量？立刻更新 .env.example。不要想着“等做完再补文档”——你会忘的，100% 会忘。

⌚ 习惯 6：站在“陌生人”的角度写 README

写完后问自己：如果我是第一次看到这个项目，能按照 README 跑起来吗？所有的步骤都写了吗？有没有遗漏的前置条件？

三、自动化测试：你的代码“安全网”

3.1 为什么测试如此重要

场景：你给 Todo 加了一个 priority 字段，改了 model、schema、router，然后运行——唉，创建 Todo 的功能正常了。部署上线。

结果第二天用户反馈：“删除 Todo 的功能挂了！”

你改 priority 的时候，不小心影响了删除逻辑，但你没有测试删除，所以没发现。

如果有自动化测试呢？

```
$ pytest
...
FAILED test_delete_todo_success ← 立刻发现问题！
```

修改后跑一遍测试，所有 15 个测试都通过，你才放心提交。这就是测试的价值——**它帮你检查你没想到的地方。**

专业开发者的铁律：没有测试的代码 = 不可信的代码。你敢修改没有测试覆盖的代码吗？你敢，但你怕。

3.2 测试基础设施详解

我们在 tests/conftest.py 中搭建了测试基础设施，理解它是理解测试的关键。

测试数据库隔离

```
# 生产/开发用的数据库：文件存储，数据持久保存  
DATABASE_URL = "sqlite+aiosqlite:///./todo.db"  
  
# 测试用的数据库：内存存储，测试结束自动消失  
TEST_DATABASE_URL = "sqlite+aiosqlite:///:memory:"
```

为什么用内存数据库？

1. **速度快**：不需要读写磁盘
2. **干净**：每次测试都是全新的空数据库，不会受之前测试的影响
3. **安全**：测试数据不会污染开发数据

依赖覆盖（Dependency Override）

```
# 关键的一行：  
app.dependency_overrides[get_db] = override_get_db
```

这行代码的意思是：

正常运行时：

API 请求 → get_db() → 连接真实数据库 → 操作开发数据

测试运行时：

API 请求 → override_get_db() → 连接测试数据库 → 操作测试数据

我们没有修改任何业务代码，只是“替换”了数据库连接，就实现了测试隔离。这就是依赖注入的威力——容易替换、容易测试。

每个测试自动重置数据库

```
@pytest.fixture(autouse=True)  
async def setup_database():  
    # 测试前：创建空表  
    async with test_engine.begin() as conn:  
        await conn.run_sync(Base.metadata.create_all)  
    yield  
    # 测试后：删除表（下次测试重新创建干净的表）  
    async with test_engine.begin() as conn:  
        await conn.run_sync(Base.metadata.drop_all)
```

这保证了：

- 测试 A 创建的 Todo 不会影响测试 B
- 每个测试都从“空白状态”开始

- 测试的执行顺序不影响结果

HTTP 测试客户端

```
@pytest.fixture
async def client():
    transport = ASGITransport(app=app)
    async with AsyncClient(transport=transport, base_url="http://test") as ac:
        yield ac
```

这个 `client` 可以像真正的浏览器一样向 API 发送请求，但不需要启动真正的服务器——它在内存中直接调用 FastAPI 应用。

使用方式：

```
response = await client.get("/todos/")          # 发送 GET 请求
response = await client.post("/todos/", json={"title": "测试"}) # 发送 POST 请求
```

3.3 AAA 测试模式

每个测试函数遵循 **AAA 模式** (Arrange-Act-Assert)：

```
async def test_create_todo_success(self, client):
    # Arrange (准备) — 准备测试需要的数据
    payload = {"title": "学习 Python", "description": "完成第一章"}

    # Act (执行) — 执行要测试的操作
    response = await client.post("/todos/", json=payload)

    # Assert (断言) — 检查结果是否符合预期
    assert response.status_code == 201
    assert response.json()["title"] == "学习 Python"
```

三步法让测试逻辑清晰：

- Arrange:** 准备好“演员和道具”
- Act:** 拍摄“正式镜头”
- Assert:** 检查“拍摄效果”是否符合预期

3.4 我们写了哪些测试

我们的 15 个测试覆盖了“正常情况”和“异常情况”：

端点	正常测试	异常测试
<code>POST /todos/</code>	有描述创建 <input checked="" type="checkbox"/> 无描述创建 <input checked="" type="checkbox"/>	空标题 <input checked="" type="checkbox"/> 缺标题 <input checked="" type="checkbox"/>

端点	正常测试	异常测试
GET /todos/	空列表 <input checked="" type="checkbox"/> 有数据 <input checked="" type="checkbox"/> 分页 <input checked="" type="checkbox"/>	—
GET /todos/{id}	正常获取 <input checked="" type="checkbox"/>	找不到 404 <input checked="" type="checkbox"/>
PATCH /todos/{id}	改标题 <input checked="" type="checkbox"/> 标记完成 <input checked="" type="checkbox"/>	找不到 404 <input checked="" type="checkbox"/>
DELETE /todos/{id}	正常删除 <input checked="" type="checkbox"/>	找不到 404 <input checked="" type="checkbox"/>
GET /health	正常返回 <input checked="" type="checkbox"/>	—

关键思维：不仅测试“能成功的情况”，还要测试“应该失败的情况”。

用户发来的数据千奇百怪，你的 API 必须优雅地处理每种情况，而不是直接崩溃。

3.5 应当培养的习惯

⌚ 习惯 7：先写代码，立刻写测试

不要把测试留到“以后”。写完一个功能就写对应的测试。专业团队甚至会先写测试再写代码（叫 TDD — 测试驱动开发）。作为初学者，至少做到“写完代码立刻补测试”。

⌚ 习惯 8：修 Bug 前先写一个能暴露 Bug 的测试

发现 Bug 后，先写一个测试来重现它（这个测试应该是失败的）。然后修复 Bug，让测试变绿。这样做的好处：确保这个 Bug 未来不会再次出现。

⌚ 习惯 9：每次提交前跑一遍测试

```
pytest -q      # 快速跑所有测试
```

只有全部通过才能提交。这个习惯能帮你避免大量“提交了坏代码”的尴尬时刻。

⌚ 习惯 10：测试既要测“成功”也要测“失败”

很多初学者只测试“传入正确数据，能正常返回”这一种情况。但现实中，用户可能传空值、传超长字符串、传不存在的 ID……你的 API 必须对每种错误情况都有合理的响应，而不是直接报 500 服务器错误。

四、日志系统：代码的“黑匣子”

4.1 为什么需要日志

你的 API 在服务器上 7×24 小时运行。凌晨三点有用户反馈“接口异常”，你不可能凌晨三点打开电脑调试。

有了日志，第二天上班打开日志文件，就能看到：

```
2026-02-14 03:15:22 | INFO    | POST /todos/ → 201 (5.3ms)
2026-02-14 03:15:45 | INFO    | GET /todos/42 → 404 (1.2ms) ← 这里！用户访问了
```

一个不存在的 Todo
 2026-02-14 03:16:01 | INFO | DELETE /todos/1 → 204 (3.8ms)

日志 = 出事后的唯一“目击证人”。

4.2 日志级别

```
logger.debug("变量 x 的值是 42")          # 只在开发调试时看
logger.info("POST /todos/ → 201 (5ms)")    # 正常运行信息
logger.warning("数据库连接池快满了")        # 需要关注但还没出错
logger.error("查询数据库失败: ...")         # 出错了
logger.critical("数据库完全无法连接!")       # 严重到可能影响整个服务
```

级别	什么时候用	类比
DEBUG	需要非常详细的调试信息时	检查每道菜的每个步骤
INFO	记录正常的操作流程	记录每桌客人点了什么
WARNING	不影响运行但需要注意	某道菜的食材快用完了
ERROR	某个操作失败了	一道菜做坏了要重做
CRITICAL	整个系统可能要挂了	厨房着火了

4.3 中间件模式

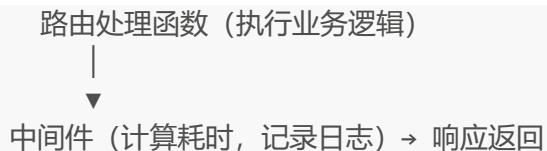
我们添加的日志功能使用了**中间件（Middleware）**模式：

```
@app.middleware("http")
async def log_requests(request: Request, call_next):
    start_time = time.time()                      # 记录开始时间
    response = await call_next(request)            # 让请求继续执行
    duration_ms = (time.time() - start_time) * 1000 # 计算耗时
    logger.info(
        "%s %s → %d (%.1fms)",
        request.method,                           # GET / POST / PATCH / DELETE
        request.url.path,                         # /todos/ 或 /todos/1
        response.status_code,                     # 200 / 201 / 404 等
        duration_ms,                            # 处理耗时
    )
    return response
```

中间件的工作方式：

请求进来 → 中间件（记录开始时间）





中间件就像一个“门卫”——所有人进出都要经过它。我们用它来给所有请求统一打上日志，而**不需要在每个路由函数里写日志代码。**

这就是中间件的价值：**一次编写，所有端点自动生效。**

4.4 应当培养的习惯

⌚ 习惯 11：日志要有结构化信息

```
# ❌ 不好：缺少关键信息  
logger.info("请求处理完毕")  
  
# ✅ 好：包含方法、路径、状态码、耗时  
logger.info("POST /todos/ → 201 (5.3ms)")
```

好的日志应该回答：**谁做了什么操作？结果是什么？花了多久？**

⌚ 习惯 12：不要在日志里打印敏感信息

```
# ❌ 绝对不要！  
logger.info("用户登录：密码是 %s", password)  
  
# ✅ 安全的做法  
logger.info("用户登录：user_id=%d", user.id)
```

五、专业开发者的工作节奏

现在你已经体验了完整的开发循环，让我们总结一下专业开发者的工作节奏：

1. 设计和规划 ← 想清楚要做什么
2. 搭建基础设施 ← 项目结构、数据库、配置
3. 实现核心功能 ← 写业务代码
4. ★ Git 提交 ← 存档！
5. 编写测试 ← 验证代码正确性
6. ★ Git 提交 ← 再次存档！

7. 完善文档 ← README、注释
- |
8. ★ Git 提交 ← 又一个存档!
- |
9. 添加增强功能 ← 日志、错误处理等
- |
10. ★ Git 提交 ← 最终存档!
- |
11. 代码审查 ← 让别人帮你看
- |
12. 部署上线 ← 让用户使用

注意到了吗？Git 提交贯穿了整个流程。 每完成一个有意义的步骤，就提交一次。这就是为什么我们第一件事是做 Git 提交——它是整个工作流的节奏基础。

六、自测清单

④ Git 相关

- 能说出 `git add` 和 `git commit` 分别做了什么吗？
- 能解释为什么 `.env` 不能提交到 Git 吗？
- 能写出规范的提交信息吗？（比如给 Todo 添加了一个优先级字段，提交信息怎么写？）
- 如果你想看看项目的所有提交历史，用什么命令？（提示：`git log --oneline`）
- 你刚把一个密码文件 `git add` 了但还没 commit，怎么撤回？（提示：`git reset HEAD 文件名`）

④ 测试相关

- 能说出 AAA 模式的三个步骤吗？
- 为什么测试用内存数据库而不是文件数据库？
- `app.dependency_overrides` 做了什么？为什么需要它？
- 为什么 `test_create_todo_empty_title_fails` 期望状态码 422 而不是 400？
- 如果你给 Todo 添加了一个 `priority` 字段，需要写哪些新测试？

④ 日志相关

- 能说出 5 个日志级别从低到高的顺序吗？
- 什么是中间件？它的优势是什么？
- 为什么不在每个路由函数里单独写 `logger.info(...)` 而是用中间件统一处理？

④ 综合理解

- 从修改代码到提交，你的标准流程应该是什么？（提示：修改 → 跑测试 → `git status` → `git add` → `git commit`）
- 如果测试没通过，你应该提交吗？为什么？
- 一个完全没接触过这个项目的人，需要从哪个文件开始看你的项目？

❖ 核心记忆：专业开发不只是“写出能跑的代码”。一个真正的专业项目：

- **有版本控制**记录每一步变更 (Git)
- **有文档**让任何人能快速上手 (README)
- **有测试**保证代码的正确性 (pytest)
- **有日志**帮你排查线上问题 (logging)

这四样东西和业务代码同等重要。缺少任何一个，项目都称不上“专业”。