Python 并发基础

1. 并发和并行

先区分两个词：

并发：看起来同时处理多个任务，本质可能是任务快速切换
并行：真正同一时刻多个任务一起执行，通常需要多核 CPU

比如：

一个人轮流接多个电话：并发
多个人同时接多个电话：并行

面试说法：并发强调任务调度和切换，提高资源利用率；并行强调多个任务真正同时执行，通常依赖多核 CPU 或多进程。

2. 线程、进程、协程区别

对比	进程	线程	协程
资源单位	操作系统资源分配单位	CPU 调度单位	用户态轻量任务
内存	进程间内存隔离	同进程线程共享内存	同线程内切换
切换成本	高	中	低
适合任务	CPU 密集型	I/O 密集型	高并发 I/O
Python 常见问题	进程通信成本	受 GIL 影响	不能阻塞事件循环

面试说法：进程资源隔离强，适合 CPU 密集型任务；线程共享进程内存，适合 I/O 密集型任务；协程是用户态的轻量级并发方式，切换成本低，适合大量网络请求、数据库请求、模型 API 调用等 I/O 场景。

3. GIL 是什么？

GIL 是 Global Interpreter Lock，全局解释器锁。

在 CPython 中，同一时刻通常只有一个线程执行 Python 字节码。

所以：

Python 多线程不适合 CPU 密集型计算
Python 多线程仍然适合 I/O 密集型任务

为什么 I/O 密集型还能用线程？

因为线程在等待网络、磁盘、数据库、模型 API 响应时，会释放 CPU，其他线程可以继续执行。

面试说法：

GIL 是 CPython 的全局解释器锁，导致同一时刻通常只有一个线程执行 Python 字节码。所以 Python 多线程不能很好利用多核做 CPU 密集型计算。但对于 I/O 密集型任务，比如网络请求、数据库访问、调用大模型 API，线程等待 I/O 时可以切换，因此仍然有价值。CPU 密集型任务可以考虑多进程、C 扩展、NumPy/PyTorch 或任务队列。

4. I/O 密集型和 CPU 密集型

I/O 密集型

大量时间在等：

网络请求
数据库查询
文件读写
调用大模型 API
调用向量库
远程服务响应

适合：

async/await
线程池
异步 HTTP client
异步数据库驱动

CPU 密集型

大量时间在算：

图片处理
大规模文本预处理
Embedding 本地批量计算
模型推理
复杂排序
压缩解压

适合：

多进程
GPU
向量化计算
任务队列
独立推理服务

5. async / await 是什么？

async def 定义协程函数，await 用来等待一个异步操作完成，同时把控制权让出去，让事件循环调度其他任务。

例子：

```
import asyncio

async def fetch_data():
    print("start")
    await asyncio.sleep(1)
    print("end")

asyncio.run(fetch_data())
```

重点：

await 后面必须是可等待对象，比如协程、Task、异步 I/O 操作。

面试说法：async/await 是 Python 的协程语法。async def 定义协程函数，调用后返回协程对象；await 会等待异步任务完成，并在等待期间让出控制权，让事件循环可以执行其他任务。

6. 协程为什么适合高并发 I/O？

假设有 100 个请求都在等模型 API 返回。

同步方式：

一个请求等完，再处理下一个

异步方式：

请求 A 等模型响应时，让出控制权
请求 B、C、D 可以继续发起请求

所以对于大量等待型任务，协程很省资源。

面试说法：协程适合高并发 I/O，因为它在等待网络、数据库、API 响应时可以主动让出控制权，让事件循环调度其他任务。相比线程，协程切换成本更低，适合大量并发请求。

7. FastAPI 和 async

FastAPI 支持：

```
@app.get("/sync")
def sync_route():
    return {"msg": "sync"}

@app.get("/async")
async def async_route():
    return {"msg": "async"}
```

怎么选？

如果里面调用的是异步库，用 async def

如果里面是普通同步阻塞代码，用 def 或放线程池

比如：

适合 async：

httpx.AsyncClient

asyncpg

aioredis

异步向量库客户端

异步模型 API 调用

不适合直接 async：

time.sleep()

同步 requests

同步数据库驱动

CPU 密集计算

错误示例：

```
import time

@app.get("/bad")
async def bad():
    time.sleep(5)  # 阻塞事件循环
    return {"ok": True}
```

正确示例：

```
import asyncio

@app.get("/good")
async def good():
    await asyncio.sleep(5)
    return {"ok": True}
```

面试说法：FastAPI 支持同步和异步接口。如果接口内部使用异步 I/O 库，比如异步 HTTP、异步数据库，可以用 async def 提高并发能力。如果在 async def 里调用同步阻塞代码，比如 time.sleep()、requests、同步数据库查询，会阻塞事件循环，反而影响性能。

8. 在 AI 应用开发里的选择

在 AI 应用中，调用大模型 API、向量库、数据库大多是 I/O 密集型，可以用 async 或线程池提高并发。如果是本地 embedding 批量计算、模型推理、图片处理这类 CPU/GPU 密集任务，我会考虑独立任务队列、多进程或单独推理服务，比如 vLLM，而不是直接阻塞 FastAPI 主服务。

FastAPI 基础、Pydantic、接口分层

1. FastAPI 是什么？

FastAPI 是一个 Python Web 框架，常用于构建 API 服务。

在 AI 应用开发里，它常作为：

前端 / 客户端请求入口

RAG 问答接口

Agent 调用接口

工单 CRUD 接口

模型服务代理层

管理后台 API

你的项目里可以这样讲：我的项目用 FastAPI 作为后端 API 层，对外提供登录注册、知识库问答、Agent 调

用、工单 CRUD、审计追踪等接口。FastAPI 负责请求参数校验、鉴权依赖注入、限流、路由分发和响应封装。

2. 一个最小接口

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/health")
def health():
    return {"status": "ok"}
```

这里：

@app.get("/health") 是路由装饰器
def health() 是接口处理函数
return 的 dict 会自动转成 JSON 响应

面试说法：FastAPI 通过装饰器把函数注册成 HTTP 路由，函数返回的 dict、Pydantic 对象等会被自动序列化成 JSON 响应。

3. 路径参数和查询参数

路径参数

```
@app.get("/tickets/{ticket_id}")
def get_ticket(ticket_id: int):
    return {"ticket_id": ticket_id}
```

请求：

GET /tickets/123

ticker_id 来自 URL 路径。

查询参数

```
@app.get("/tickets")
def list_tickets(status: str | None = None, page: int = 1):
    return {"status": status, "page": page}
```

请求：

GET /tickets?status=open&page=2

status 和 page 来自 query string。

面试说法：路径参数通常用于定位具体资源，比如 /tickets/{ticket_id}；查询参数通常用于过滤、分页、排序，比如 ?status=open&page=2。

4. 请求体和 Pydantic

POST / PUT 通常需要请求体。

```
from pydantic import BaseModel

class TicketCreate(BaseModel):
    title: str
    description: str
    priority: str = "medium"

@app.post("/tickets")
def create_ticket(data: TicketCreate):
    return data
```

请求 JSON：

```
{
  "title": "电脑无法开机",
  "description": "按电源没有反应",
```

```
  "priority": "high"
}
```

FastAPI 会自动：

读取 JSON 请求体
用 Pydantic 校验字段
把数据转换成 TicketCreate 对象
校验失败时返回 422

面试说法：FastAPI 通常用 Pydantic 模型定义请求体和响应体。Pydantic 负责运行时数据校验、类型转换和序列化。如果请求字段缺失或类型不匹配，FastAPI 会自动返回校验错误。

5. response_model

```
class TicketOut(BaseModel):
    id: int
    title: str
    status: str

@app.post("/tickets", response_model=TicketOut)
def create_ticket(data: TicketCreate):
    return {
        "id": 1,
        "title": data.title,
        "status": "open",
        "internal_note": "内部字段"
    }
```

实际响应只会包含：

```
{
  "id": 1,
  "title": "电脑无法开机",
  "status": "open"
}
```

internal_note 会被过滤掉。

面试说法：response_model 用于定义接口响应结构。它可以做响应数据校验、序列化和字段过滤，避免把内部字段、敏感字段直接暴露给前端。

6. Depends 依赖注入

FastAPI 的 Depends 常用于：

获取数据库 Session
获取当前用户
鉴权
权限校验
公共分页参数
限流

例子：

```
from fastapi import Depends, HTTPException

def get_current_user():
    return {"id": 1, "name": "张三"}

@app.get("/me")
def me(user=Depends(get_current_user)):
    return user
```

执行过程：

请求 /me
→ FastAPI 先执行 get_current_user()
→ 把返回值注入到 user
→ 执行 me(user)

面试说法：Depends 是 FastAPI 的依赖注入机制，可以把鉴权、数据库连接、当前用户、权限校验等公共逻辑抽离出来，避免每个接口重复写。

7. FastAPI 项目分层

推荐结构：

```
app/
├── main.py
├── api/
│   └── routes/
│       └── tickets.py
├── schemas/
│   └── ticket.py
├── services/
│   └── ticket_service.py
├── repositories/
│   └── ticket_repo.py
├── models/
│   └── ticket.py
└── db/
    └── session.py
```

层	职责
router	HTTP 路由、参数接收、依赖注入
schema	Pydantic 请求体 / 响应体
service	业务逻辑
repository / dao	数据库读写
model	ORM 数据库模型
db	数据库连接、Session 管理

面试说法：我一般会把 FastAPI 项目分成 router、schema、service、repository、model。router 负责 HTTP 层，schema 负责请求和响应模型，service 处理业务逻辑，repository 处理数据库访问，model 对应数据库表。这样业务逻辑不会堆在接口函数里，也方便测试和维护。

8. 结合你的项目怎么讲

你可以这样说：在我的 Policy KB Assistant 项目中，FastAPI 负责提供 /agent、/ask、/tickets、/audit_logs 等接口。接口层先做请求 schema 校验、用户鉴权和限流，然后把请求交给 service 层。比如创建工单时，router 接收 TicketCreate，service 处理字段校验、权限、默认状态和业务规则，repository 负责写入数据库，最后通过 response_model 返回安全字段。

第 8 课：CRUD + SQLAlchemy + 事务

1. CRUD 是什么？

CRUD 是业务系统最基础的数据库操作：

Create：创建
Read：读取 / 查询
Update：更新
Delete：删除

比如工单系统：

创建工单
查询工单详情
分页查询工单列表
更新工单状态
删除 / 取消工单

你的项目可以这样讲：我的工单模块本质上就是围绕 tickets、ticket_comments、ticket_drafts、audit_logs 等表做 CRUD。Agent 不直接操作数据库，而是通过 service 层调用受控的 CRUD 方法。

2. ORM 是什么？

ORM 是 Object Relational Mapping，对象关系映射。

简单说：

数据库表 → Python 类
表字段 → 类属性
一行记录 → 一个对象

例子：

```
class Ticket(Base):
    __tablename__ = "tickets"

    id = Column(Integer, primary_key=True)
    title = Column(String)
    status = Column(String)
```

这对应数据库表：

tickets
- id
- title
- status

面试说法：ORM 把数据库表映射成 Python 类，把一行数据映射成对象。这样可以用面向对象方式操作数据库，减少手写 SQL，但复杂查询和性能问题仍然需要理解 SQL。

3. ORM Model 和 Pydantic Schema 区别

对比	ORM Model	Pydantic Schema
作用	映射数据库表	定义 API 输入输出结构
面向	数据库	HTTP 请求 / 响应
是否直接暴露	不建议	可以暴露
常见例子	Ticket	TicketCreate,TicketOut

```
# ORM Model：数据库表
class Ticket(Base):
    __tablename__ = "tickets"
    id = Column(Integer, primary_key=True)
    title = Column(String)
    internal_note = Column(String)

# Pydantic Schema：请求/响应模型
    class TicketOut(BaseModel):
    id: int
    title: str
```

为什么不能直接返回 ORM Model？

因为 ORM Model 里可能有：

内部字段
敏感字段
数据库关系对象
不适合前端看到的数据

面试说法：ORM Model 负责数据库映射，Pydantic Schema 负责 API 输入输出。两者不要混用。Model 里可能包含内部字段或关系对象，直接暴露给前端不安全；Schema 可以控制哪些字段允许输入、哪些字段允许输出。

4. Session 是什么？

SQLAlchemy 的 Session 可以理解为：

一次数据库操作上下文 / 工作单元

它负责：

跟踪对象变化
执行 SQL
管理事务
commit 提交
rollback 回滚
close 释放连接

面试说法：Session 是 SQLAlchemy 和数据库交互的核心对象，可以理解为一次工作单元。它负责管理对象状态、执行查询、提交事务、回滚事务和释放连接。

5. commit / rollback / close

commit：提交事务，把修改真正写入数据库
rollback：回滚事务，撤销本次未提交修改
close：关闭 Session，释放数据库连接资源

典型结构：

```
try:
    db.add(ticket)
    db.commit()
    db.refresh(ticket)
    return ticket
except Exception:
    db.rollback()
    raise
finally:
    db.close()
```

面试说法：写操作成功后需要 commit；如果中间出错，需要 rollback，避免部分修改污染数据库；最后要 close 释放连接。FastAPI 中通常用依赖注入统一管理 Session 生命周期。

6. refresh 是什么？

```
db.add(ticket)
db.commit()
db.refresh(ticket)
```

refresh 会从数据库重新读取对象最新状态。

为什么需要？

因为数据库可能生成：

自增 id
默认值
更新时间
触发器字段

面试说法：refresh 用来把数据库提交后的最新字段同步回 ORM 对象，比如自增 id、默认状态、创建时间等。

7. 查询单条数据

```
def get_ticket(db: Session, ticket_id: int):
    return db.query(Ticket).filter(Ticket.id == ticket_id).first()
```

如果不存在：

```
ticket = get_ticket(db, ticket_id)
if not ticket:
    raise HTTPException(status_code=404, detail="Ticket not found")
```

面试注意：

first() 找不到返回 None
one() 找不到或多条会抛异常
all() 返回列表

8. 分页查询

常见分页：

```
def list_tickets(db: Session, page: int = 1, page_size: int = 20):
    offset = (page - 1) * page_size
    return (
        db.query(Ticket)
        .offset(offset)
        .limit(page_size)
        .all()
    )
```

解释：

page=1, page_size=20 → offset=0
page=2, page_size=20 → offset=20
page=3, page_size=20 → offset=40

面试说法：常见分页方式是 limit + offset。小数据量简单好用，但深分页时 offset 很大可能变慢。数据量大时可以考虑游标分页，比如基于 id 或 created_at 做下一页查询。

9. 更新数据

```
def update_ticket(db: Session, ticket_id: int, title: str):
    ticket = db.query(Ticket).filter(Ticket.id == ticket_id).first()
    if not ticket:
        raise HTTPException(status_code=404, detail="Ticket not found")

    ticket.title = title
    db.commit()
    db.refresh(ticket)
    return ticket
```

关键点：

先查是否存在
再修改字段
commit
refresh
返回

10. 删除数据：硬删除 vs 软删除

硬删除：

```
db.delete(ticket)
db.commit()
```

记录真的没了。

软删除：

```
ticket.deleted_at = datetime.utcnow()
```

```
ticket.is_deleted = True
db.commit()
```

记录还在，只是标记删除。

区别：

删除方式	优点	缺点
硬删除	简单，数据彻底删除	不方便恢复和审计
软删除	可恢复，可审计	查询时要过滤删除数据

在企业系统里，软删除更常见。

面试说法：硬删除是直接删除数据库记录，软删除是增加 is_deleted 或 deleted_at 字段标记删除。企业系统通常更偏软删除，因为方便恢复、审计和追踪历史。

11. CRUD 分层怎么写？

推荐：

```
router：接收 HTTP 请求
service：业务校验、权限、状态流转
repository：纯数据库操作
model：数据库表
schema：请求和响应结构
```

例如取消工单：

```
router
→ service 检查当前用户是否有权限取消
→ service 检查工单状态是否允许取消
→ repository 更新状态
→ audit service 写审计日志
→ 返回 response_model
```

这就是你项目要讲的关键。

第 9 课：JWT、鉴权、权限校验

1. 认证和授权的区别

两个词要分清：

```
Authentication：认证，确认“你是谁”
Authorization：授权，确认“你能做什么”
```

例子：

```
登录成功，知道你是用户 A → 认证
判断用户 A 是否能取消工单 123 → 授权
```

面试说法：认证解决身份问题，授权解决权限问题。登录、token 校验属于认证；判断用户能不能访问某个资源、执行某个操作属于授权。

2. JWT 是什么？

JWT 是 JSON Web Token，常用于前后端分离项目的登录状态传递。

一个 JWT 通常由三部分组成：

- **Payload 仅经过 Base64Url 编码**，这是一种可逆的编码方式，**不是加密**。
- 任何人都可以轻松解码 Payload，直接读取其中的内容（例如使用 atob 或在线 JWT 解码工具）。

```
Header.Payload.Signature
```

Payload 里可以放：

```
{
  "sub": "user_id",
  "exp": "过期时间",
  "role": "user"
```

```
}
```

注意：JWT 的 Payload 默认只是 Base64URL 编码，不是加密，不能放密码、身份证号、密钥等敏感信息。

面试说法：JWT 是一种自包含 token，服务端签发后，客户端后续请求带上 token，服务端通过签名验证 token 是否被篡改，并解析出用户身份和过期时间。JWT 适合前后端分离和无状态认证，但 Payload 不应该放敏感信息。

3. 登录流程

典型流程：

```
用户提交用户名和密码
→ 后端校验密码哈希
→ 校验成功，生成 access token
→ 前端保存 token
→ 后续请求在 Authorization Header 携带 token
→ 后端解析 token，得到当前用户
```

请求头：

```
Authorization: Bearer <access_token>
```

面试说法：登录时服务端验证账号密码，成功后签发 JWT。客户端后续请求在 Authorization Header 里携带 Bearer token，后端验证签名和过期时间后解析用户身份。

4. 密码不能明文存储

数据库里不能存：

```
password = "123456"
```

应该存：

```
password_hash = hash("123456" + salt)
```

常见做法：

```
bcrypt
argon2
passlib
```

面试说法：密码不能明文存储，应该使用 bcrypt、argon2 等算法加盐哈希存储。登录时不是解密密码，而是把用户输入的密码用同样算法校验哈希是否匹配。

5. access token 和 refresh token

Token	作用	生命周期
access token	访问接口	短
refresh token	换新的 access token	长

面试说法：access token 用于访问接口，过期时间较短；refresh token 用于在 access token 过期后换取新的 access token，过期时间较长。这样既能减少频繁登录，又能降低 access token 泄露后的风险。

6. FastAPI 中获取当前用户

常见做法：

```
def get_current_user(token: str = Depends(oauth2_scheme)):
    payload = decode_jwt(token)
    user_id = payload["sub"]
    user = get_user_by_id(user_id)
    if not user:
        raise HTTPException(status_code=401)
    return user
```

接口中使用：

```
@app.get("/me")
def me(current_user=Depends(get_current_user)):
    return current_user
```

执行流程：

请求进入接口
→ Depends 先执行 get_current_user
→ 从 Header 取 token
→ 验证 token
→ 查用户
→ 注入 current_user
→ 执行业务函数

面试说法：FastAPI 里一般通过 Depends 封装当前用户解析逻辑。依赖函数负责读取 Bearer token、校验 JWT、查询用户，如果失败就返回 401，成功则把 current_user 注入到接口或 service 层。

7. 权限校验怎么做？

认证之后，还要授权。

例如取消工单：

用户 A 登录成功
但不代表用户 A 可以取消所有工单

要检查：

工单是否存在
当前用户是否创建者 / 负责人 / 管理员
当前工单状态是否允许取消
操作是否需要确认

面试说法：鉴权不只是 token 有效，还要做资源级权限校验。比如工单系统中，用户只能查看和操作自己有权限的工单。取消、删除、关闭这类危险操作还需要状态校验和确认机制。

8. RBAC 是什么？

RBAC 是 Role-Based Access Control，基于角色的访问控制。

常见角色：

普通用户
客服 / 运维人员
管理员

权限例子：

普通用户：创建工单、查看自己的工单
运维人员：处理分配给自己的工单
管理员：查看全部工单、分配工单、关闭工单

面试说法：RBAC 是基于角色的权限控制。系统给用户分配角色，再给角色分配权限。这样比给每个用户单独配置权限更容易维护。

9. Agent 场景下的权限安全

这是你的重点。

Agent 不能因为用户一句话：

帮我取消所有工单

就直接执行。

安全设计：

Agent 只生成计划
后端校验工具白名单
校验参数 schema
校验当前用户权限
校验资源归属
危险操作进入确认态
写审计日志

面试说法：在 Agent 系统里，我不会让模型直接执行数据库操作。模型只负责生成结构化 ToolPlan，后端再做工具白名单、参数校验、权限校验、资源归属校验和危险操作确认。真正执行后还要写审计日志，保证可追踪。

10. 常见 HTTP 状态码

你至少要记这些：

```
200 OK：请求成功
201 Created：创建成功
400 Bad Request：请求参数错误
401 Unauthorized：未认证 / token 无效
403 Forbidden：已认证但无权限
404 Not Found：资源不存在
409 Conflict：状态冲突，比如重复操作
422 Unprocessable Entity：请求体校验失败
500 Internal Server Error：服务端未知错误
```

重点区分：

401：你是谁都不知道，没登录或 token 错

403：知道你是谁，但你没权限

第 10 课：接口测试、统一异常、中间件、限流

1. 为什么要写接口测试？

接口测试的目标不是测试函数细节，而是测试完整 HTTP 行为。

例如测试：

```
POST /tickets
```

要确认：

```
传入合法数据 → 返回 201
缺少必填字段 → 返回 422
未登录 → 返回 401
访问别人资源 → 返回 403
重复提交 → 返回 409 或幂等返回已有结果
```

面试说法：接口测试用于验证 API 的请求、响应、状态码、鉴权、参数校验和异常处理是否符合预期。FastAPI 通常使用 pytest + TestClient 来写接口测试。

2. pytest 是什么？

pytest 是 Python 常用测试框架。

常见测试函数：

```
def test_create_ticket_success():
    ...
```

断言：

```
assert response.status_code == 201
assert response.json()["title"] == "无法登录"
```

面试说法：pytest 是 Python 常用测试框架，可以用 assert 编写测试断言，也支持 fixture 做测试数据准备和依赖复用。

断言（Assert） 是测试中的核心概念，简单说就是：**“我断言某个条件必须为真，如果不为真，就说明程序出错了。**

3. TestClient 是什么？

FastAPI 提供 TestClient，可以像真实 HTTP 请求一样测试接口。

示例：

```
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

def test_health():
```

```
    response = client.get("/health")
    assert response.status_code == 200
```

它的好处是：

- 不用真的启动 uvicorn
- 可以直接在测试里请求 FastAPI app
- 可以测试状态码、JSON 响应、Header、鉴权等

面试说法：TestClient 可以在测试环境中模拟 HTTP 请求，不需要真正启动服务，就能测试 FastAPI 路由、依赖、异常处理和响应结构。

4. fixture 是什么？

fixture 是 pytest 用来准备测试环境和测试数据的机制。

例如：

```
@pytest.fixture
def client():
    return TestClient(app)
```

或者：

```
@pytest.fixture
def auth_headers():
    return {"Authorization": "Bearer test-token"}
```

面试说法：fixture 用于复用测试准备逻辑，比如创建测试数据库、初始化客户端、创建测试用户、生成登录 token，避免每个测试重复写 setup 代码。

5. 什么是统一异常处理？

项目里不能到处随便返回错误格式。

不推荐：

```
{"error": "not found"}
```

另一个接口又返回：

```
{"message": "权限不足"}
```

应该统一成：

```
{
  "code": "TICKET_NOT_FOUND",
  "message": "工单不存在",
  "trace_id": "xxx"
}
```

FastAPI 可以用 exception handler 统一处理异常。

面试说法：统一异常处理可以保证所有接口返回一致的错误结构，方便前端处理，也方便日志排查。FastAPI 可以通过 exception_handler 捕获自定义业务异常、HTTPException 和 ValidationError，并统一转换成标准响应。

6. 业务异常 vs 系统异常

业务异常是可预期错误：

- 工单不存在
- 无权限
- 状态不允许取消
- 重复提交
- 参数不合法

系统异常是非预期错误：

- 数据库连接失败
- 代码 bug
- 第三方服务超时
- 未知异常

面试说法：业务异常应该返回明确的错误码和状态码，比如 403、404、409；系统异常一般返回 500，并记录详细日志，但不把内部堆栈暴露给前端。

7. Middleware 是什么

Middleware 是中间件，请求进入接口前、响应返回前都会经过它。

流程：

请求进入
→ Middleware 前置处理
→ 路由函数
→ Middleware 后置处理
→ 返回响应

常见用途：

记录请求日志
生成 trace_id
统计耗时
CORS
限流
鉴权预处理
异常兜底

面试说法：中间件可以在请求进入路由前和响应返回前做统一处理，比如生成 trace_id、记录请求日志、统计耗时、处理 CORS、做限流等。

8. trace_id 是什么？

trace_id 是一次请求的唯一标识。

作用：

前端报错给 trace_id
后端通过 trace_id 查日志
串起 API、Agent、RAG、工具调用、数据库操作

你的项目里尤其重要，因为有：

用户请求
→ Agent Planner
→ RAG 检索
→ ToolPlan
→ 工单操作
→ 审计日志

面试说法：trace_id 用于追踪一次请求的完整链路。每个请求进入系统时生成唯一 trace_id，后续日志、Agent 调用、RAG 查询和审计记录都带上它，方便排查问题和审计。

9. CORS 是什么？

CORS 是跨域资源共享。

当前端和后端不在同一个域名、端口或协议时，就可能出现跨域。

例如：

前端：http://localhost:3000
后端：http://localhost:8000

端口不同，也算跨域。

FastAPI 可以用 CORSMiddleware。

面试说法：CORS 是浏览器的跨域安全机制。当前端页面和后端 API 的协议、域名或端口不一致时，需要后端配置允许的 origins、methods 和 headers，否则浏览器会拦截请求。

10. 限流是什么？

限流是限制用户或 IP 在一段时间内的请求次数。

例如：

同一个 IP 每分钟最多请求 60 次
同一个用户每分钟最多调用 Agent 10 次

目的：

防刷
防止接口被打爆
控制 LLM 调用成本
保护数据库和向量库

面试说法：限流用于控制单位时间内的请求次数，防止恶意刷接口、资源耗尽和 LLM 成本失控。常见做法是基于 IP、用户 ID 或 API Key 在 Redis 中计数，并设置过期时间。

11. 幂等性是什么？

幂等性指同一个请求执行多次，结果和执行一次一致。

例如创建工单：

用户点击一次：

创建工单 123

用户网络卡顿，重复点击三次。

不希望产生：

工单 123
工单 124
工单 125

而是希望：

第一次创建成功
后面重复请求返回同一个工单

常见做法：

前端传 Idempotency-Key
后端保存 key 和执行结果
重复 key 直接返回之前结果

面试说法：幂等性用于防止重复提交导致重复创建或重复扣费等问题。常见做法是客户端传 Idempotency-Key，服务端以用户 ID + key 作为唯一约束，第一次请求执行并保存结果，后续相同 key 的请求直接返回已有结果。

第 11 课：配置管理、后台任务、文件上传、同步/异步数据库

1. 配置管理是什么？

项目里不要把配置硬编码在代码里。

不推荐：

```
DATABASE_URL = "postgresql://user:pass@localhost:5432/db"
SECRET_KEY = "abc123"
```

应该从环境变量或配置文件读取。

常见配置：

DATABASE_URL
REDIS_URL
JWT_SECRET_KEY
OPENAI_API_KEY
QDRANT_URL
ENV
LOG_LEVEL

面试说法：配置管理是把数据库地址、Redis 地址、JWT 密钥、LLM API Key 等环境相关配置从代码里抽离出来，通过环境变量或 .env 管理。这样开发、测试、生产环境可以使用不同配置，也避免把密钥写死在代码里。

2. Pydantic Settings 是什么？

FastAPI 项目常用 Pydantic Settings 管理配置。

示例：

```
from pydantic_settings import BaseSettings


class Settings(BaseSettings):
```

```
    database_url: str
    redis_url: str
    jwt_secret_key: str
    env: str = "dev"

    class Config:
        env_file = ".env"

settings = Settings()
```

面试说法：我一般会用 Pydantic Settings 统一管理配置，它可以从环境变量或 .env 文件读取配置，并做类型校验。业务代码不直接读取环境变量，而是从 settings 对象中获取配置。

3. 为什么密钥不能提交到 GitHub？

比如：

```
OPENAI_API_KEY
JWT_SECRET_KEY
数据库密码
云服务器密码
```

不能提交到 GitHub。

原因：

```
泄露后别人可以调用你的接口
产生费用
访问数据库
伪造 token
攻击系统
```

面试说法：密钥不能提交到代码仓库，应该放在环境变量、部署平台的 Secret 配置或 .env 文件中，并把 .env 加入 .gitignore。如果密钥泄露，需要立即轮换。

4. BackgroundTasks 是什么？

FastAPI 的 BackgroundTasks 可以在响应返回后执行一些轻量后台任务。

例如：

```
发送邮件
写通知
记录非关键日志
异步生成报告
```

示例：

```
from fastapi import BackgroundTasks

@app.post("/tickets")
def create_ticket(background_tasks: BackgroundTasks):
    ticket = create_ticket_service()
    background_tasks.add_task(send_notice, ticket.id)
    return ticket
```

面试说法：BackgroundTasks 适合处理轻量、非关键、耗时但不需要立即返回结果的任务，比如发送通知或写辅助日志。它不是完整任务队列，不适合长时间、强可靠性的任务；这类任务更适合 Celery、RQ 或消息队列。

5. 文件上传怎么做？

FastAPI 用 UploadFile 和 File 处理文件上传。

示例：

```
from fastapi import UploadFile, File
```

```
@app.post("/upload")
async def upload(file: UploadFile = File(...)):
    content = await file.read()
    return {"filename": file.filename}
```

常见校验：

- 文件大小
- 文件类型
- 文件后缀
- MIME type
- 是否病毒扫描
- 是否保存到对象存储

面试说法：FastAPI 可以用 UploadFile 处理文件上传。实际项目中不能只接收文件，还要校验文件大小、类型、后缀，避免恶意文件上传。大文件一般不直接读入内存，而是流式保存到磁盘或对象存储。

6. 同步数据库和异步数据库怎么选？

FastAPI 支持 async，但不代表所有代码都必须 async。

同步数据库：

- SQLAlchemy sync session
- psycopg2
- 普通 CRUD 项目常用

异步数据库：

- SQLAlchemy AsyncSession
- asyncpg
- 高并发 I/O 场景更适合

面试说法：FastAPI 支持异步接口，但是否使用异步数据库要看项目复杂度和并发需求。普通后台管理、工单 CRUD 项目用同步 SQLAlchemy 也可以，简单稳定；如果系统有大量并发 I/O，比如高并发查询、长连接或异步调用链，可以考虑 AsyncSession + asyncpg。但要避免在 async 接口里直接调用阻塞式 I/O，否则会阻塞事件循环。

7. async def 和 def 路由有什么区别？

```
@app.get("/sync")
def sync_route():
    ...

@app.get("/async")
async def async_route():
    ...
```

区别：

- def：同步路由，适合普通阻塞式代码
- async def：异步路由，适合 await 异步 I/O

关键点：

- async def 里面应该 await 异步函数
- 不要在 async def 里直接做大量阻塞操作

面试说法：如果接口内部主要调用同步数据库或普通阻塞函数，用 def 就可以；如果调用的是异步数据库、异步 HTTP 客户端或异步 LLM SDK，可以用 async def 并配合 await。async 不是自动变快，只有在 I/O 可等待且调用链是异步时才有收益。

8. 依赖覆盖是什么？

测试时，可能不想真的连生产数据库，也不想真的调 LLM。

FastAPI 可以覆盖 Depends。

例如：

```
app.dependency_overrides[get_current_user] = fake_current_user
```

用途：

测试时 mock 当前用户
替换测试数据库
跳过真实鉴权
mock 外部服务

面试说法：FastAPI 的 dependency_overrides 可以在测试中替换依赖，比如把真实数据库替换成测试数据库，把真实用户鉴权替换成 mock 用户，把真实 LLM 调用替换成 fake service。这样测试更稳定，也不会影响生产环境。