API Mock 服务器 —— Claude Code 演示方案

> **用途**：AI Coding 课程现场演示
> **时长**：约 20 分钟
> **难度**：中等（需要听众了解基本的 API 开发概念）

一、项目概述

1.1 一句话描述

根据 OpenAPI/Swagger 定义文件，自动生成能返回假数据的 HTTP Mock 服务器。

1.2 实际场景

- 前后端分离开发时，后端 API 还没写好，前端需要先联调
- 测试环境不稳定，需要稳定的 mock 数据
- 演示或原型阶段，快速搭建假后端

1.3 为什么适合演示?

维度	说明
业务易懂	只要写过 API 的人都秒懂
痛点真实	几乎每个团队都遇到过「等后端接口」的问题
复杂度适中	比 Todo 复杂，但 15-20 分钟能跑通核心
展示 AI 能力全面	文件解析、代码生成、数据模拟、错误处理
有翻车空间	复杂嵌套结构、循环引用等场景容易出问题
可扩展性强	听众可以继续加功能（延迟模拟、错误注入等）

二、技术栈

组件	选择	理由
语言	TypeScript	类型安全，主流
运行时	Node.js	生态丰富
HTTP 框架	Express	轻量，启动快
YAML 解析	js-yaml	成熟稳定
假数据生成	@faker-js/faker	功能丰富

三、功能范围

功能	演示版 ✅	完整版
解析 OpenAPI 3.0 YAML	✅	✅
生成 GET 端点	✅	✅
生成 POST/PUT/DELETE 端点	✅	✅
基础类型 mock（string/number/boolean）	✅	✅
数组类型 mock	✅	✅
嵌套对象 mock	✅	✅
$ref 引用解析	❌	✅
响应延迟模拟	❌	✅
错误状态码模拟	❌	✅
请求参数校验	❌	✅

四、演示用 OpenAPI 文件

将以下内容保存为 `api-spec.yaml`：

```yaml
openapi: 3.0.0
info:
  title: 用户管理 API
  version: 1.0.0

paths:
  /users:
    get:
      summary: 获取用户列表
      responses:
        '200':
          description: 成功
          content:
            application/json:
```

```yaml
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/User'
    post:
      summary: 创建用户
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/CreateUserRequest'
      responses:
        '201':
          description: 创建成功
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'

  /users/{id}:
    get:
      summary: 获取单个用户
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: string
      responses:
        '200':
          description: 成功
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
    put:
```

```yaml
      summary: 更新用户
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: string
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/CreateUserRequest'
      responses:
        '200':
          description: 更新成功
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
    delete:
      summary: 删除用户
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: string
      responses:
        '204':
          description: 删除成功

components:
  schemas:
    User:
      type: object
      properties:
```

```yaml
        id:
          type: string
          format: uuid
        name:
          type: string
        email:
          type: string
          format: email
        age:
          type: integer
          minimum: 0
          maximum: 150
        role:
          type: string
          enum: [admin, user, guest]
        isActive:
          type: boolean
        createdAt:
          type: string
          format: date-time
        address:
          type: object
          properties:
            street:
              type: string
            city:
              type: string
            country:
              type: string
            zipCode:
              type: string

    CreateUserRequest:
      type: object
      required:
        - name
```

```
      - email
    properties:
      name:
        type: string
      email:
        type: string
        format: email
      age:
        type: integer
      role:
        type: string
        enum: [admin, user, guest]
```

五、完整演示流程

阶段 0：开场铺垫（2 分钟）

讲师说：

> "大家有没有遇到过这种情况：后端说接口下周才能好，但你今天就要开始写前端？
>
> 今天我们用 Claude Code 来做一个工具：给它一个 API 定义文件，它自动生成一个能返回假数据的 Mock 服务器。
>
> 整个过程我们会走一遍 SDD 流程：需求 → Spec → 实现 → 测试。"

阶段 1：需求描述（2 分钟）

给 Claude Code 的 Prompt：

```
我想做一个 API Mock 服务器生成工具。

功能：
- 读取一个 OpenAPI 3.0 的 YAML 文件
- 解析其中的路径和数据模型定义
- 自动启动一个 HTTP 服务器
- 每个定义的端点都能返回符合 schema 的假数据

技术栈：Node.js + TypeScript + Express

请先帮我整理一份结构化的技术 Spec，不要急着写代码。
```

讲师旁白：

> "注意我这里强调了'不要急着写代码'——这是 SDD 的关键。我们先让 AI 帮我们理清楚要做什么。"

阶段 2：审核 Spec（3 分钟）

预期 AI 生成的 Spec 结构：

```
## API Mock Server - 技术规格

### 输入
- OpenAPI 3.0 YAML 文件路径

### 输出
- 运行在指定端口的 HTTP 服务器
- 每个路径返回符合 schema 的 mock 数据

### 核心模块
1. YAML 解析器：读取并解析 OpenAPI 文件
2. Schema 解析器：将 OpenAPI schema 转换为内部格式
3. Mock 数据生成器：根据 schema 生成假数据
```

```
4. 路由生成器：为每个 path 创建 Express 路由
5. 服务器：启动 HTTP 服务

### 数据类型映射
| OpenAPI 类型 | Mock 策略 |
|-------------|----------|
| string | faker.lorem.word() |
| string + format:email | faker.internet.email() |
| string + format:uuid | faker.string.uuid() |
| string + format:date-time | faker.date.recent().toISOString() |
| string + enum | 随机选择枚举值 |
| integer | faker.number.int({ min, max }) |
| boolean | faker.datatype.boolean() |
| array | 生成 1-5 个元素 |
| object | 递归生成各属性 |
```

讲师操作：

> "我来审一下这个 Spec...
>
> 有几个问题我想改：
>
> 1. 数组元素数量应该可配置，默认 3 个就好
> 2. 需要处理 $ref 引用——但演示版本我们先跳过复杂的循环引用
> 3. 加一个启动端口的配置
>
> 我把修改意见告诉它..."

修改 Prompt：

```
Spec 整体不错，我有几点修改：

1. 数组默认生成 3 个元素
2. 需要支持 $ref 引用解析（但暂不处理循环引用）
3. 端口号通过命令行参数指定，默认 3000
4. 启动时在控制台打印所有注册的路由

请更新 Spec。
```

阶段 3：确定项目结构（2 分钟）

Prompt：

```
好的，Spec 我确认了。现在请设计项目目录结构，然后生成初始化命令和配置文件。

要求：
- 目录结构清晰，职责分离
- 使用 pnpm
- 配置 TypeScript 严格模式
- 添加 npm scripts: dev, build, test
```

预期输出：

```
api-mock-server/
├── src/
|   ├── index.ts              # CLI 入口
|   ├── server.ts             # Express 服务器
|   ├── types.ts              # 类型定义
|   ├── parser/
|   |   ├── yaml-loader.ts    # 加载 YAML
|   |   └── schema-resolver.ts # 解析 schema 和 $ref
|   └── generator/
|       ├── mock-data.ts      # 生成假数据
|       └── routes.ts         # 生成路由
```

```
├── tests/
|   └── mock-data.test.ts      # 单元测试
├── api-spec.yaml              # 示例 API 定义
├── package.json
├── tsconfig.json
└── README.md
```

阶段 4：实现核心模块（5 分钟）

Prompt：

```
现在开始实现。请按以下顺序：

1. 先实现 types.ts - 定义内部数据结构
2. 再实现 parser/ - YAML 加载和 schema 解析（包括 $ref 解析）
3. 然后 generator/mock-data.ts - 假数据生成（这是核心）
4. 接着 generator/routes.ts - 路由生成
5. 最后 server.ts 和 index.ts

每个文件实现完，简要说明设计决策。
```

讲师旁白（边看生成边说）：

> "看这里，AI 在处理嵌套对象时用了递归...
>
> 这个 enum 处理得不错，随机选一个...
>
> 注意它怎么处理 format:email 的，用了 faker 的 internet.email()...
>
> $ref 解析这里，它会去 components/schemas 里查找对应的定义..."

阶段 5：运行测试（3 分钟）

安装依赖并启动:

```
# 安装依赖
pnpm install

# 启动服务（使用我们准备的 api-spec.yaml）
pnpm dev -- ./api-spec.yaml --port 3000
```

预期控制台输出:

```
🚀 API Mock Server starting...

Registered routes:
  GET    /users
  POST   /users
  GET    /users/:id
  PUT    /users/:id
  DELETE /users/:id

✅ Server running at http://localhost:3000
```

验证请求:

```
# 获取用户列表
curl http://localhost:3000/users | jq

# 预期输出类似:
# [
#   {
#     "id": "550e8400-e29b-41d4-a716-446655440000",
#     "name": "lorem",
#     "email": "john.doe@example.com",
#     "age": 42,
#     "role": "user",
#     "isActive": true,
```

```
#     "createdAt": "2024-01-15T10:30:00.000Z",
#     "address": {
#       "street": "123 Main St",
#       "city": "Springfield",
#       "country": "USA",
#       "zipCode": "12345"
#     }
#   },
#   ...
# ]

# 获取单个用户
curl http://localhost:3000/users/123 | jq

# 创建用户（POST 请求）
curl -X POST http://localhost:3000/users \
  -H "Content-Type: application/json" \
  -d '{"name":"test","email":"test@example.com"}' | jq
```

讲师展示：

> "看，返回的数据完全符合我们定义的 schema：
>
> - id 是 UUID 格式
> - email 看起来像真实邮箱
> - role 是枚举值之一（admin/user/guest）
> - address 是嵌套对象，各字段都有值
>
> 前端拿到这些数据就可以开始开发了，不用等后端。"

阶段 6：翻车演示（3 分钟）

翻车场景 1：模糊需求

给一个不带 Spec 的模糊指令：

```
帮我写一个 mock server
```

展示 AI 的"自由发挥"：

> "看，它自己定义了数据结构、自己选了端口、甚至自己编了几个 API...
>
> 这和我们的需求完全对不上。它可能生成了一个 /products 接口，但我们要的是 /users。
>
> 这就是为什么我们要先写 Spec —— **没有契约，AI 就在赌博**。"

翻车场景 2：循环引用

在 api-spec.yaml 中加一个复杂的 schema：

```yaml
Comment:
  type: object
  properties:
    id:
      type: string
    content:
      type: string
    author:
      $ref: '#/components/schemas/User'
    replies:
      type: array
      items:
        $ref: '#/components/schemas/Comment'  # 循环引用自己!
```

预期翻车：

> "看，这里有循环引用：Comment 的 replies 引用 Comment 自己。

> 如果 AI 不特殊处理，会无限递归导致栈溢出...
>
> （运行后可能报错或卡死）
>
> 这就是为什么我们需要测试、需要 Review ——**AI 写的代码不一定考虑到所有边界情况**。
>
> 解决方案是加一个递归深度限制，但这个我们留给听众课后练习。"

阶段 7：补充测试（2 分钟）

Prompt：

```
请为 generator/mock-data.ts 的核心函数编写单元测试。

覆盖场景：
1. 基础类型（string, number, boolean）
2. 带 format 的 string（email, uuid, date-time）
3. enum 类型
4. 嵌套对象
5. 数组类型
6. 空 schema 或 undefined 的边界情况

使用 Jest 框架，文件放在 tests/mock-data.test.ts
```

讲师旁白：

> "注意我没有让 AI 自己决定测什么，而是明确告诉它测试场景。
>
> 这也是 SDD 的一部分——**测试用例也是 Spec 的一部分**。"

运行测试：

```
pnpm test
```

阶段 8：总结复盘（2 分钟）

讲师总结：

> "回顾一下，整个过程中：
>
> **人做的事**：
>
> - 定义需求边界（演示版不做循环引用处理）
> - 审核和修改 Spec
> - 决定项目结构和技术选型
> - 设计测试场景
> - Review 关键实现
>
> **AI 做的事**：
>
> - 整理结构化 Spec 草稿
> - 生成项目骨架和配置
> - 实现所有代码
> - 编写测试
>
> 整个项目从 0 到能运行，大概 15 分钟。如果我自己写，至少要 2-3 小时。
>
> 但关键是：这不是'AI 随便生成的代码'，而是**在 Spec 约束下、经过 Review 的代码**。"

六、问题应对预案

问题	应对策略
AI 生成的代码有语法错误	"这很正常，我们让它修。这就是为什么要有测试。"
运行时报错	把错误信息喂给 AI："这个错误怎么修？"
生成的假数据不够"真实"	"可以继续优化 faker 的配置，比如用中文 locale"
时间不够	提前准备好"检查点"代码，关键时刻切换到预置版本
网络问题装不了依赖	提前装好依赖，或准备离线 node_modules
AI 理解错需求	"看，它理解偏了，这就是为什么要 Review Spec"

七、课后延伸作业

给听众的练习建议（难度递增）：

难度	任务	提示
⭐ 初级	支持更多 string format（uri、hostname、ipv4）	扩展 mock-data.ts 的 format 映射
⭐⭐ 中级	添加响应延迟模拟（`?_delay=1000`）	用 setTimeout 包装响应
⭐⭐ 中级	添加错误状态码模拟（`?_status=500`）	检查 query 参数，返回对应状态码
⭐⭐⭐ 高级	处理循环 $ref 引用（加递归深度限制）	传入 depth 参数，超过阈值返回 null
⭐⭐⭐ 高级	支持从 URL 加载远程 OpenAPI 文件	用 fetch 获取，支持 JSON 和 YAML

八、完整 Prompt 模板

以下是可以直接复制使用的完整 Prompt：

8.1 初始需求 Prompt

```
# API Mock Server 生成器 - 需求文档

## 目标
创建一个工具，读取 OpenAPI 3.0 YAML 定义，自动生成返回 mock 数据的 HTTP 服务器。

## 技术栈
- Node.js + TypeScript
- Express
- js-yaml（YAML 解析）
- @faker-js/faker（假数据生成）
- Jest（测试）

## 功能需求

### 输入
- OpenAPI 3.0 YAML 文件路径
- 可选：端口号（默认 3000）

### 输出
- 运行中的 HTTP 服务器
- 控制台打印所有注册的路由

### Mock 数据规则

| Schema 类型 | 生成策略 |
|------------|---------|
| string | faker.lorem.word() |
| string + format:email | faker.internet.email() |
| string + format:uuid | faker.string.uuid() |
| string + format:date-time | new Date().toISOString() |
| string + enum:[...] | 随机选择一个枚举值 |
| integer | faker.number.int({ min: 0, max: 100 }) |
| integer + minimum/maximum | 使用指定范围 |
| boolean | faker.datatype.boolean() |
| array | 生成 3 个元素 |
```

```
| object | 递归生成各属性 |

### 约束
- 支持 $ref 引用解析
- 暂不处理循环 $ref 引用（可能导致无限递归）
- 所有端点返回 200 状态码（DELETE 返回 204）
- POST/PUT 请求忽略请求体，直接返回 mock 数据

## 项目结构
```

```
src/
├── index.ts            # CLI 入口
├── server.ts           # Express 服务器
├── types.ts            # 类型定义
├── parser/
│   ├── yaml-loader.ts    # 加载 YAML
│   └── schema-resolver.ts # 解析 schema 和 $ref
└── generator/
    ├── mock-data.ts      # 生成假数据
    └── routes.ts         # 生成路由
tests/
└── mock-data.test.ts
```

```
## 请按以下顺序实现

1. 类型定义 (types.ts)
2. YAML 加载 (parser/yaml-loader.ts)
3. Schema 解析 (parser/schema-resolver.ts)
4. Mock 数据生成 (generator/mock-data.ts)
5. 路由生成 (generator/routes.ts)
6. 服务器入口 (server.ts, index.ts)
7. 单元测试 (tests/mock-data.test.ts)

每完成一个模块，简要说明设计决策。
```

8.2 测试生成 Prompt

```
请为 generator/mock-data.ts 的 generateMockData 函数编写单元测试。

覆盖以下场景：

1. **基础类型**
   - string 类型返回字符串
   - integer 类型返回整数
   - boolean 类型返回布尔值

2. **String 格式**
   - format: email 返回邮箱格式
   - format: uuid 返回 UUID 格式
   - format: date-time 返回 ISO 日期格式

3. **Enum 类型**
   - 返回值必须是枚举数组中的一个

4. **复合类型**
   - object 类型递归生成各属性
   - array 类型生成指定数量的元素
```

```
5. **边界情况**
   - 空 schema 返回 null 或 undefined
   - 未知类型的处理

使用 Jest 框架，每个测试要有清晰的描述。
```

8.3 Review Prompt

```
请 Review 以下代码，重点关注：

1. **正确性**
   - 所有 OpenAPI 数据类型是否都处理了？
   - $ref 引用解析是否正确？
   - 边界条件是否处理？

2. **安全性**
   - 文件读取是否有路径遍历风险？
   - 是否有潜在的无限递归？

3. **可维护性**
   - 函数职责是否单一？
   - 命名是否清晰？
   - 是否有重复代码可以抽取？

4. **性能**
   - 是否有不必要的重复计算？

请按以下格式输出：
- 🔴 必须修复：...
- 🟡 建议改进：...
- 🟢 做得好的地方：...
```

九、演示检查清单

演示前确认以下事项：

- ☐ Node.js 已安装（v18+）
- ☐ pnpm 已安装
- ☐ api-spec.yaml 文件已准备
- ☐ 网络正常（或依赖已预装）
- ☐ 终端字体足够大（现场投影）
- ☐ curl 和 jq 已安装（用于测试请求）
- ☐ 备用代码已准备（以防现场翻车太严重）

十、时间分配参考

阶段	内容	时长
0	开场铺垫	2 分钟
1	需求描述	2 分钟
2	审核 Spec	3 分钟
3	项目结构	2 分钟
4	实现代码	5 分钟
5	运行测试	3 分钟
6	翻车演示	3 分钟
7	补充测试	2 分钟
8	总结复盘	2 分钟
总计		**约 24 分钟**

> 可根据实际情况压缩阶段 6-7，控制在 20 分钟内。

祝演示顺利！