后端常见疑问解答

> 本章节摘自 hdu-cs.wiki。

这里我尽可能多地列举了小白可能想问的问题，简单的分了一下类别，列举在这里。

入门思考与自我定位

1. 为什么学习后端？

这里假设你是一个刚刚掌握了编程基础，知道类和对象等基本概念、可能完成过一些练习但缺乏完整项目经验的新手，想继续深入学习。

在如果有引导、有系统学习资源，从后端开发切入项目实践，是一个很不错的选择。我会讲讲优点和缺点。

学习后端会很好的**巩固你的编程内功**，因为它的核心是**逻辑，数据和架构**，会直接强迫你应用和深化函数、类、对象、数据结构等知识。你写的每一行代码都在处理具体的数据和逻辑流。入门后端之后，你也更容易进入云计算，DevOps，架构设计等等更进阶的领域，可以尝试“微服务”等更现代的系统构建。

后端的**技术保质期**更长。相较于前端，后端开发初期可以暂时不用理会UI设计、浏览器兼容性、页面布局以及五花八门，日新月异的前端框架。后端许多核心框架，如 Java 的 Spring、Python 的 Django，已经稳定存在十多年。近年来虽然像 Go, Rust 这种新兴语言的现代 Web 框架也开始流行，但整体来看，后端开发的核心思想变化并不大。真正变化较多的，其实是在基础设施和部署层面，比如容器化、服务网格、Serverless 等。而前端，三年前的主流技术可能放到现在已经不是最优解，导致开发者可能需要不断学习新的工具集。准确说，前端变化虽快，但并非完全“无用”，核心模型（组件化、响应式、虚拟DOM）是稳定演进的，不过也不可避免地带来开发者适应新技术的心智负担。

学习后端之后，面对不同语言或框架的后端，你的**技能可迁移性**会更强。API设计、数据库交互、用户认证，中间件等，这些概念在不同的语言和框架之间是通用的。而核心的业务逻辑甚至基本可以脱离特定的 Web 框架而存在。**你会学习到通用的“道”，而不是仅仅某一个工具的“术”**。在技能的迁移性这一点上，后端比前端好很多。后端的核心职责是**处理数据和执行业务规则**，前端的核心职责是**管理UI状态，响应用户交互**。前端的业务逻辑基本等同于 UI 交互逻辑，而如何高效地管理“状态”并在状态改变时更新UI，正是每个前端框架要解决的核心问题。对于状态管理，UI渲染更新，组件通信，可能每个框架作者都充满了不同的看法，每个框架的机制都可能不同，有时甚至是哲学上的不同（比如 React 是“函数式 + 声明式”，Vue 是“模板驱动 + 响应式”），会需要你在整个思维模式上做出改变。不同的前端框架代表了不同的“道”，它们的“术”也为了各自的“道”而服务。你会需要透过现象看本质的能力，理解组件化和现代 Web 应用的构建思想，才是前端里的可迁移技能。总之，后端在早期阶段，更容易让人接触到通用的设计思想，而前端则在深入后，才逐渐显现其模型哲学和架构价值。

从功利一点的角度来看，后端在很多团队中的确更容易产生直接的业务价值，并在产品形态与系统架构上拥有更多的话语权。

比如在很多公司中，尤其是以业务驱动为主的企业，产品设计往往是后端和产品一起深度参与完成的：他们会一起确定业务流程、字段结构、权限模型等核心内容，然后由后端定义出接口，告诉前端该怎么调用。

在这种分工下，前端更多是在已有接口和交互方案的基础上实现用户体验和交互逻辑，而后端则会相对而言更早介入产品设计，在架构，数据建模层面有更高参与度，包括数据模型如何抽象、流程怎么串联、接口怎么设计；系统的扩展性、可维护性、可复用性怎么保证；甚至未来支持新业务方向时的延展能力。

如果你希望在产品落地中扮演更主动的角色，希望不仅是“接接口”，而是参与“设计接口”的过程，那么学习后端会让你更有这种掌控感和产出感。

而从技术成长上看，后端也常常承担“架构演进”的主力，比如服务拆分、微服务架构设计、数据库优化、系统容错等，这些都往往是后端主导的领域。因此如果你对“做更深的技术”和“参与业务底层逻辑”的方向感兴趣，后端是一个天然更贴近这些目标的位置。

当然，这种角色差异也不是绝对的。在注重用户体验和复杂前端交互的项目中，比如低代码平台、图形编辑器、可视化大屏、Web IDE 等，前端也会拥有极大的设计空间和技术主导性。只是从平均情况和传统项目结构来看，后端更接近系统核心，因此也容易承担更大的设计责任和产出主力。

从后端入门同样有缺点。

显而易见的缺点是它更抽象，缺乏即时直观的反馈，入门门槛高。可能你写了好多代码，却只实现了一个简单的增删改查。如果缺乏兴趣，缺乏引导，大概率会让你感到枯燥迷茫。对于编程基础较弱的同学，可能前面所说的“巩固”会变成“卡住”。相比之下，前端哪怕只是复制粘贴一个按钮，都能马上看到结果。

并且想独立完成一个功能完整的后端项目，难度的确不可避免会很大。对于刚学会编程基础的新手，只写过小函数小脚本，要是看到那么复杂的项目结构，这真的令人望而却步。新手可能会花大量时间在搭环境配置，连不上库，理不清架构上。

后端调试错误对于一些新手可能很困难，原因在于后端错误很多时候相对更加抽象，偏向逻辑和网络协议的错误。这些错误经常只是说：“Internal Server Error”，“Database connection refused”，“NoneType has no attribute 'xyz'”，可能很难找到问题源头。而前端调试更偏UI/状态/异步交互类（虽说二者都不简单）。

不过，我们的 HDU-CS-Wiki 后端教程会用最新手友好的方式，让你无痛入门后端。

教程仍在施工，敬请期待，欢迎提供建议。

2. 我适合后端吗？

相比于一个按钮好不好看，你是否更关心它背后的系统是如何高效运转的？你是否享受分析一个问题的本质、理清逻辑结构？你是否对性能、稳定性、安全性等底层问题感兴趣？如果答案偏向于是，那么你就有成为优秀后端开发者的潜质。

如果你希望在产品落地中扮演更主动的角色，希望不仅是“接接口”，而是参与“设计接口”的过程，那么学习后端会让你更有这种掌控感和产出感。

如果你玩天际线，缺氧，异形工厂之类的规划类游戏，那你也很可能适合学习后端。~~(夹带私货了，因为本人也玩)~~

还有很多初学者，即使起初并不偏好后端，但希望通过项目实践掌握软件工程的通用技能，也很适合从后端开发入手，培养工程思维。也许后来你会发现后端开发的乐趣。

如果你仍不确定，欢迎来群里聊聊。

3. 后端比前端难学吗？

这个问题没有绝对的答案，有各自的难处，难点和节奏不同。

后端门槛略高，更抽象更逻辑化，但入门之后你会感到自己在稳定扎实地前进，路线会很清晰。前端可以相对更友好地入门，但可能需要你更多广泛的涉猎，时刻紧跟动态。而且当你成为熟练的前端开发者之后，如果你

想继续研究前端开发，可能你的关注点会延伸到美学、人文、可访问性等超出纯编程的领域。这需要你站在“用户视角”上去感受、思考，不是纯靠逻辑能解决的。后端则更多聚焦在数据与逻辑的构建，强调性能、可维护性、可靠性，偏向工程理性。

但你如果想问这个问题，大概率你目前只打算先选一个学。这是完全可以理解的，但也要意识到：**现代 Web 开发，前后端知识是有一定交叉的**。端开发者如果完全不懂前端，那和前端沟通起来会有障碍，可能不知道“这个接口怎么用才好”；前端开发者如果完全不懂后端，也可能会写出一些逻辑混乱、接口滥用的代码。真正理解前后端职责边界、协作模式的人，才能写出“干净、合理、可维护”的系统。

后端的前置知识

4. 学后端之前，我需要先学什么？

1. **掌握至少一门编程语言基础**。Python、JavaScript、Java、Go 都可以，但不只是会语法，更重要的是理解函数，理解类和对象，能灵活使用基础数据结构（列表、字典、集合等），理解控制流（循环、分支）、递归、异常处理。这里推荐一个十分权威的神课 **CS61A**。本人也是通过 CS61A 打开了编程的大门。

2. **基础的命令行使用能力**。后端学习过程中会频繁接触命令行（CLI），比如：创建/进入目录、运行脚本；安装软件包、运行服务器；使用 Git 进行版本控制；修改配置文件等。虽然不是必须“精通”，但你至少要不排斥和恐惧命令行、能基本操作它。

3. **基本的英文阅读能力**。后端世界几乎一切文档、错误提示、API参考资料……全是英文的。你不需要能流利写作或口语，但你需要：能读懂开发文档；能大致看懂报错信息；能 Google/Stack Overflow 查英文资料。如果你暂时没有练成，也不用担心，善用翻译工具就行。比如**沉浸式翻译**浏览器插件。

4. **畅通的网络环境**。后端开发离不开 GitHub、Stack Overflow、官方文档、npm/pip 等包管理平台。如果你不能流畅访问这些网站，找学长私聊琢磨下。

5. **提问的智慧**。你可能听说过**《提问的智慧》**这篇经典文章。这个文章的语气对新手有点压迫感，有种高高在上令人生畏的感觉，至少我是新手时是这么想的，看完感觉自己都不配提问了。但它确实讲的很好。虽然标题是提问的智慧，但实际上它不只是告诉你怎么问问题，更是在潜移默化地教你怎么分析问题、定位问题，最终自己解决问题，而后者我认为反而是更重要的。你可以在无聊的时候看一看这篇文章。

5. 我需要先学习前端吗？

不需要先学前端，完全可以直接学后端。

关于这个问题其实前面的问题已经部分回答过了。这里继续梳理一遍。

后端的核心任务是：处理数据、执行业务逻辑、提供接口。这套能力可以完全独立于 UI 层被开发和测试，不依赖页面渲染、按钮交互、DOM 等前端技术。也就是说，你哪怕一个 HTML 标签都没写过，也可以搭建一个能处理请求、连接数据库、执行业务逻辑的后端服务。

这正是后端适合作为编程进阶路径的原因之一 —— 它能直接锤炼你的数据处理、结构设计、逻辑思维等“编程内功”。

但懂一点前端，有时会帮你更好地理解前后端协作的边界。稍微了解一些 HTML/HTTP/前端如何调用 API 的方式，你会更容易理解前后端接口怎么对接，明确哪些逻辑该由后端做，哪些放在前端，或者自己写一些简单页

面来测试接口，提升效率。这些都不是前置条件，而是顺带掌握、越学越通透的部分。因此 Web 前后端也不是完全分家的。你大概率两者都要涉猎一点点。

6. 学后端要会很多算法吗？

不需要。大多数后端工作不需要你掌握大量复杂的算法，不需要你再像高中那样天天刷算法题，大多数时候用到些常规算法就行了，尤其现在各种高质量的算法库和框架早已实现了大多数通用算法，很多时候你只需要正确调用并理解其输入输出即可。

因为，后端开发面对的日常问题更多的是在于工程能力、架构思维、业务理解能力，而不是实现某个图算法，手写红黑树。不过，基本的算法和数据结构知识是进阶必需的。排序、查找，哈希表、栈、队列、树、图的基本概念，时间复杂度之类的。你需要知道“这些算法大概是干什么的”，知道“何时使用它们”，而不一定要从头造轮子。

算法当然是值得学的，但仅对于后端开发角度而言，是不必精通的。你对算法本身感兴趣，当然可以深入。最重要的在于你的兴趣。

后端的学习路线，方法，资源

7. 后端要学些啥？

一开始只需要掌握一门语言 + 一个 Web 框架 + 一个数据库，能写出一个最基础的 API 服务，就已经能入门了。

你只需要学会：

- 怎么写接口（GET、POST、PUT、DELETE）；
- 怎么存数据（增删改查）；
- 怎么把你的服务部署起来让别人访问。

这就已经是后端开发者了，真的没那么难。

在此基础上，你可以逐步学地深一点：

- 更复杂的数据库模型（比如多表关联）；
- 项目结构的组织（如何让代码更清晰、可维护, 设计模式）；
- 基础的安全（鉴权、密码加密等）；
- 容器化部署（Docker）；
- 团队协作开发（Git, Code Review）；
- 性能优化、缓存、消息队列；
- 微服务，容器集群编排(kubernetes);
- ...

这些内容是可以逐步补上的，不是一开始就要全掌握。

如果你想要一份更系统的大纲，欢迎关注 HDU-CS-Wiki 的后端教程。

8. 后端的相关教程资源？

坦白说，如果你认真读了前面的内容，你基本已经知道你大概要学什么内容，按照什么顺序学了。

但是我怕新手犯的一个错误，就是打开 B 站，搜了个后端入门，打开黑马程序员三十天速通 Spring Boot 一声不吭在那看，看着视频敲啥就照着敲啥。结果教程看完，发现离了教程就不会做了。又或者看到一半发现实在太枯燥无聊，中道崩殂。

这种“填鸭式看剧”学习法听起来很安心，但实际上效率极低。你可能一晚上看了两小时，却啥也没留下来，第二天一行代码都写不出来。然后你会想，我大概需要重新看一遍教程，或者找点别的教程。这时你就陷入了教程漩涡，不能说没有进步，但是可能相对还是有点低效。而且后端更重要的不是“学某个框架”，而是学会看文档，学会调试，学会融会贯通。这也是为什么我的教程倾向于尽可能多地演示官方文档的材料的目的。

读一些教学书籍未尝不可，但它需要高度的投入。一般这种书会非常详细地解释每个概念，这很棒，但也意味着进步是缓慢的，这不适用于每个人。而且大多数人不会“阅读第 x 章并完成章节后的练习”。

正确的后端学习姿势是，主动探索，文档优先。

我很推荐你根据自己的兴趣，用一个简单的语言和框架，跟着文档一步步直接上手尝试，尝试实现你自己想做的有趣东西。想象不到如何实现，那就直接问 AI 或者查文档。

这里也推荐一些靠谱的后端课程/资源：

- MDN Web Doc - Server-side Mozilla 出品，浅显易懂，涵盖后端全貌，业界权威。
- FastAPI 官方文档 它虽然是框架文档，但文档作者很有开源精神，在作为框架说明文档的同时，还普及了通用的后端知识和编程知识，已经是广受好评的后端教程。Python 同时也是一个很好上手的语言，从 FastAPI 开始入门后端是相当新手友好的。
- SQLModel 官方文档 SQLModel 和 FastAPI 同作者。同样是新手友好的库，同样在文档中教学 SQL 的基础知识。
- FastAPI 全栈模板 在你感觉越写越乱时，可以模仿它来组织你的代码结构。拉取仓库，跑一跑玩一玩，也能顺便让你上手 Git 和 Github。
- freeCodeCamp 有很多不错的免费教程。
- SQL Bolt 交互式 SQL 教学，很有趣，很推荐。

如果你听说过 Go 语言并且想从 Go 语言上手(这会比较有挑战性)：

- Learn Go with tests 从 Go 语言基础，到 简单的 http 应用开发，并且还教会你测试驱动开发的方法，不完全是后端教程，更多还是 Go 语言基础。
- Go Web Examples 从基础到RESTful API开发

遇到问题时，问老登是很好的。

不要被教程框住你该走的路。你要成为构建知识的人，而不是单纯的知识接受者。

最后，欢迎关注 HDU-CS-Wiki 的后端教程。教程预计将分为两个部分，第一部分会用 Python FastAPI 带你简单入门后端，第二部分会由浅入深地用 Go 语言带你搭建自己的第一个大型微服务架构应用。

欢迎你推荐你认为同样不错的教程。

9. 我要选择哪一门语言？ 我要选择什么框架？

大多数时候你纠结的不是语言，而是害怕选错、走弯路。但放心讲一句：新手根本不可能选错。对于后端新手，选择简单的语言和框架即可，Python FastAPI，或者 node.js Express 就不错。但是入门之后，也不用担心自己的技术栈不够深，而把自己定位在某一个语言，专注同一个框架。你可以拥抱各种新领域，广泛探索。因为对于后端，真正的“深入”不在于语言，而在于你是否理解背后的原理和模式。

关于AI

10. 如何正确利用 AI 辅助学习？

在你完全接触一个新领域时，把 AI 当作一个全天候，有耐心的私人助教。这个阶段的 AI 会极大节省你到处翻看文档查找资源的时间。并且好处在于，你永远不需要担心自己的问题是不是太蠢。你可以说：用一个生活中的例子，解释什么是RESTful API？帮我看看这段代码有什么可以优化的地方？有没有明显的不良实践？

但是！如果你还没学会基础编程的各种概念，除非你已经绞尽脑汁，否则请不要将 AI 用于你的任何 **代码练习** 中。这个阶段你需要的是一砖一瓦地亲手搭建最基础的知识框架。

如果你已经度过了编程小白时期，开始尝试写一些自己的小项目，那么我会提出一个完全相反的观点：**尽可能多地，放心大胆地，让 AI 来生成你的项目代码**。

不过这里的前提是，你需要**阅读 AI 的每一行代码**，确保自己读懂它。并且**调试和 debug 时不要使用 AI**！

使用 AI 快速实现需求能极大提高你的信心，能迅速把你领进现代开发的工作流，熟悉多样的技术。同时这个阶段也完全没必要怕 AI 写的坏代码，我反而要说这是不得不品鉴的一环。只有自己第一次把坏代码写出来，你当你想改一个小功能却要动十几个地方时，你会发觉“原来这就是坏味道”，才能切实体会到代码为什么坏，才能理解某种所谓的“最佳实践”为什么是最佳实践，某些设计模式或原则为什么是好的，能借此学习如何重构自己的代码。相反，如果直接去硬啃那么多“最佳实践”，这就像是你不知道敌人是谁就举起了武器，想必是会脱靶。

当你度过了这种“后新手”时期，你一定也会有自己对于 AI 工具的独特理解了。在这之前，你会从“判断 AI 写的代码好不好”开始，逐渐成为一个真正独立的开发者。

11. AI 会取代后端开发吗？

短期内不会。经常看到视频说完全用 AI 写了一个大项目，但目前来看， AI 的代码虽然能跑，却往往有不少问题。经常看起来 90% 正确，实际就是差那 10% 。经常缺乏架构意识和维护性考量（可以通过提示词工程来改善），或者安全性、健壮性差。

但是这样来看，这也意味着 AI 的进步速度飞快，也许再过几年，AI 就能独立负责一整个项目，写出又快又好的代码。

因此，后端开发者大概会需要具有架构师和产品经理的思维。

AI 发达了之后，后端开发的时间成本和人力成本都会下降，更有价值的问题不再是技术本身。后端开发者可能更需要对业务有整体的把握能力。因为技术不是最终目的，**技术的本质是为了产品服务的**。什么样的产品是用户所需要的，产品针对的用户群体是什么，产品的商业模式是什么，等等，可能才是更有价值的问题。