

CODEBASE · 代码理解

# 别人的代码 看不懂?

不是看不懂，是缺一张**能问的图**  
20 万行陌生代码，从哪一行开始读

# 版权与来源声明

## COPYRIGHT & ATTRIBUTION

### 本书性质

本书是**独立创作的学习资料**，基于公开的开源项目与作者自身的工程实践整理而成。书中所有章节结构、论述框架、判断标准、示例、图示与清单，均为原创组织与撰写；未逐字复制、未整段翻译任何第三方文档。

### 内容来源

本书涉及的代码图谱工具与方法，参考并讨论以下开源项目：

Understand-Anything ( [github.com/Egonex-AI/Understand-Anything](https://github.com/Egonex-AI/Understand-Anything) )

该项目以 **MIT License** 授权发布，版权归 **Yuxiang Lin and Infinite Universe, Inc.** 所有。

### MIT 许可摘要

MIT 协议授予任何人免费获取、使用、复制、修改、合并、发布、分发、再许可及**销售**软件副本的权利，条件是：在软件的所有副本或实质性部分中，保留上述版权声明与本许可声明。本书对来源项目的引用符合该条件，并在此完整列明版权归属。

### 独立创作范围

本书**不是**该开源项目的官方文档、官方教程或衍生版本，与该项目的作者、维护者及其所属机构**无隶属、无赞助、无背书关系**。书中的案例推演、数字结论、经验判断由作者独立给出，不代表上游项目立场。

### 商标声明

本书任何页面均未使用上述项目或其他任何第三方的商标、Logo 或标识图形。书中出现的产品名称仅用于客观指代与说明，相关权利归各自所有者。封面品牌标记「可乐书屋」为本书出品方自有标识。

### 免责声明

本书基于特定版本与特定条件下的实践整理，不构成技术、法律或投资建议；工具行为与数据结论可能随版本迭代而变化，请以实际环境为准并独立验证。

MIT License · Copyright © Yuxiang Lin and Infinite Universe, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

# 目录

|     |                            |    |
|-----|----------------------------|----|
| 序   | 这本书不教你装工具                  | 04 |
| 01  | 定位·先搞清楚你卡在哪一步              | 05 |
| 1.1 | 20 万行的第一小时：你缺的不是耐心         | 06 |
| 1.2 | 三种「看不懂」：没结构、没意图、没上下文       | 07 |
| 1.3 | 硬读是效率最低的路径                 | 08 |
| 1.4 | 什么时候该问人，什么时候该画图            | 09 |
| 02  | 建图·把代码变成一张能看的图             | 10 |
| 2.1 | 为什么是图，不是文档                 | 11 |
| 2.2 | 确定性的一半：解析器拿得准的东西           | 12 |
| 2.3 | 语义的另一半：解析器给不了的东西           | 13 |
| 2.4 | 分层：把 800 个文件压成 5 个色块       | 14 |
| 2.5 | 增量与指纹：代码改了，图怎么跟上           | 15 |
| 03  | 读图·四条按图索骥的路径               | 16 |
| 3.1 | 引导导览：按依赖顺序读，不是按目录顺序        | 17 |
| 3.2 | 领域视图：从业务流反推代码位置            | 18 |
| 3.3 | 影响分析：动手之前先看波及面             | 19 |
| 3.4 | 提问式阅读：把代码库当对话对象            | 20 |
| 04  | 落地·让这张图活下来                 | 21 |
| 4.1 | 过期的图，比没有图更危险               | 22 |
| 4.2 | 团队共享：一份图，省掉重复劳动            | 23 |
| 4.3 | 新人的第一周：把图谱嵌入 onboarding    | 24 |
| 4.4 | 图的边界：它解决不了什么               | 25 |
| A   | 附录 A 八个误读图谱的坑：现象 / 原因 / 修法 | 26 |
| B   | 附录 B 接手陌生代码库自检清单（24 条）     | 27 |
| 结   | 结语 图谱只是脚手架                 | 29 |

# 这本书不教你装工具

装一个插件、跑一条命令，二十分钟就能做完。这个时代获取工具的成本低到可以忽略。

所以这本书不打算在这上面花篇幅。**难的是装完之后。**

图出来了。你盯着屏幕上几百个节点和它们之间的连线，鼠标悬停在某个圆点上，弹出一段英文摘要。你看了两分钟，然后发现自己还是不知道该从哪个节点开始读——**这不是你的问题，是没人告诉过你该怎么读一张图。**

## 工具解决的是「有没有」，书要解决的是「准不准」

任何图谱工具都会告诉你：这是文件、这是函数、这是调用关系。它不会告诉你：

- 这条边是解析器**算出来的**，还是模型**猜出来的**？两者出错的方式完全不同。
- 图上没画关系的那两个模块，是真没关系，还是工具没识别出来？
- 同一个文件，昨天被标成 Service 层，今天变成 Utility——该信哪个？
- 哪些结论可以直接拿去改代码，哪些必须先人工确认？

这些问题的答案，决定了这张图对你是**资产**还是**干扰**。

## 这本书的组织方式

全书按「接手一个陌生代码库」的真实顺序展开，而不是按工具的功能列表展开：

| 章  | 解决的问题                   | 你会拿到      |
|----|-------------------------|-----------|
| 01 | 你卡在「没结构」还是「没意图」还是「没上下文」 | 一张自我诊断表   |
| 02 | 图上的东西哪些是事实、哪些是解释        | 一套可信度分级标准 |
| 03 | 先看什么、后看什么、怎么从问题倒推位置     | 四条阅读路径    |
| 04 | 怎么让图不烂在硬盘里，以及它到底解决不了什么  | 一份维护与边界清单 |

### 先说清楚这本书不是什么

它不是工具说明书。工具的命令、参数、安装步骤变化很快，写在书里半年就过期。这本书讲的是**不随版本变化的东西**：怎么判断一张图谱能不能用、怎么分辨事实与推测、什么时候该信图、什么时候该回去读代码。

### 怎么读最快

如果你手上正好有一个看不懂的代码库，直接从第 1 章读起，边读边给自己的困惑归类。如果你只是想建立判断力，先读第 2 章——它决定了后面所有内容的地基。



CH.01 定位

# 先搞清楚 你卡在**哪一步**

「看不懂」不是一种状态，是三种，而且解法完全不同

大多数人接手陌生代码的第一步就错了：打开 IDE，从 main 函数开始往下读。这不是勤奋，这是在用最慢的方式消耗你最宝贵的注意力。

## 先看一个例子

去医院看病，你不会跟医生说「我不舒服」。你会说「昨天开始胃疼，吃东西就疼，不吃就不疼」。把**症状说清楚，医生才能开对检查单。**

看不懂代码也是一样——「**看不懂**」是**症状，不是诊断**。你得先知道自己是哪一类看不懂，才谈得上选方法。

## 20 万行的第一小时：你缺的不是耐心

你坐在新工位上，clone 下一个仓库，数了数：4,200 个文件，20 万行。你的第一反应大概是——打开 IDE，找到入口文件，从第一行开始读。

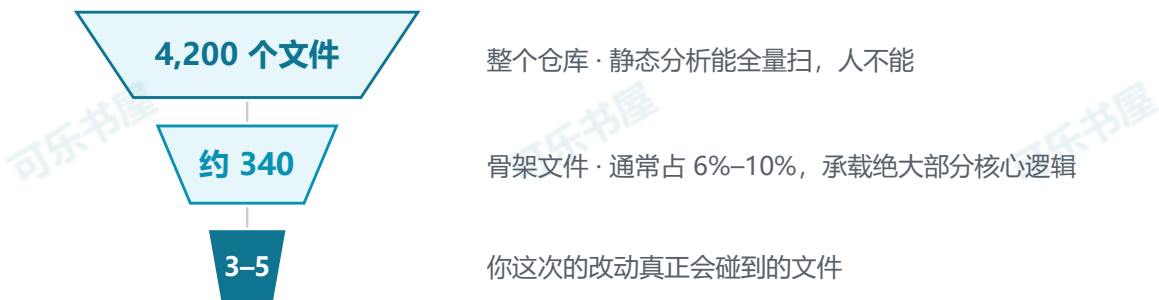
这个反应默认了一个前提：**这个代码库是有「顺序」的，而且它恰好和文件系统的顺序一致**。事实并非如此——一个维护了三年以上的项目长成现在这样，多半不是因为当初这么设计，而是因为几次紧急发版、几次人员变动、几个没人删掉的废弃模块。目录结构记录的是**历史**，不是**结构**。

| 顺序   | 由什么决定      | 适合拿来做什么  |
|------|------------|----------|
| 目录顺序 | 文件夹命名与字母排列 | 找已知路径的文件 |
| 创建顺序 | 历史上谁先被写出来  | 考古、追溯演进  |
| 依赖顺序 | 谁引用谁、谁调用谁  | 理解运行链路   |

按目录顺序读代码，等于按书架的摆放顺序读书——不是按章节顺序，更不是按你认为重要的顺序。

### 第一小时真正该做的三件事

1. **找到系统的对外入口**。不是 `main`，是别人调用你的那一层：HTTP 路由、CLI 命令、导出的公共 API。入口决定边界。
2. **找出「骨架文件」**。被引用次数最多的那批文件。它们通常不多，但决定了整个系统的形状。
3. **划定「这次可以不看」的区域**。这一条最省时间，也最少有人做。



把 4,200 个文件先压到 340 个，再压到 5 个——这是阅读顺序，不是删减代码

#### 把这句话记住

你缺的不是耐心，是把 **4,200 压缩成 5 的能力**。任何让你一张一张打开文件去读的方法，都是在浪费你最稀缺的资源——注意力。

## 三种「看不懂」：没结构、没意图、没上下文

「看不懂」是一句症状描述，不是诊断。就像「肚子疼」可能是胃炎、可能是阑尾、也可能只是饿——三种情况的处理方式完全不同。

接手陌生代码时，你遇到的困惑基本可以归进三类，而且它们的解法**互不通用**：

| 类型   | 你的典型症状                   | 你真正缺的 | 该用什么解            |
|------|--------------------------|-------|------------------|
| 没结构  | 「这东西到底在哪个文件里？」「我从哪儿开始看？」 | 一张地图  | 结构图谱 / 检索        |
| 没意图  | 「代码每行都读懂了，但不知道为什么要这么写」   | 一段上下文 | 语义摘要 / 问人 / 提交历史 |
| 没上下文 | 「我改了这行，但不知道会崩到哪儿去」       | 一张影响面 | 依赖追踪 / 影响分析      |

### 为什么必须先分类

因为最常见的错误，是用「没结构」的办法去解「没意图」的问题——把那段代码翻来覆去地读，读得越熟越困惑。

代码可以告诉你 `if (user.role === 'admin')`，但不会告诉你这个判断是十年前为了绕开某个 bug 加的，还是上周为了合规要求加的。

#### 识别信号

如果你已经能逐行解释一段代码，但被问「为什么要这样」时答不上来——你缺的是意图，不是结构。继续读源码是最没效率的选择。

### 三种缺口的典型触发场景

| 场景          | 主要缺口       | 表现                      |
|-------------|------------|-------------------------|
| 刚 clone 下仓库 | 没结构        | 不知道入口在哪，不知道该先看哪个目录      |
| 要改一个老模块     | 没意图        | 代码读得懂，但不敢动，怕破坏某个不为人知的约定 |
| 接手离职同事的代码   | 没意图 + 没上下文 | 双重缺口，也是「看不懂」感受最强烈的时候    |

#### 先做这一步

下次打开陌生代码前，先花两分钟问自己：我现在卡的是「在哪」「为什么」还是「会怎样」？把答案写下来。写不出来的话，说明你还没定位到问题，这时候读代码只是在缓解焦虑。

## 硬读是效率最低的路径

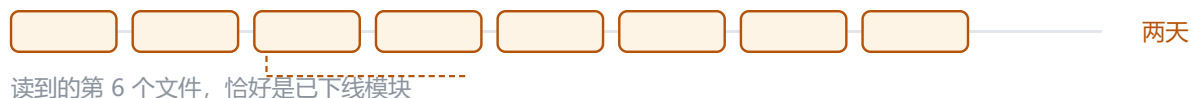
有一个真实的反例：某工程师接手一个老项目，花了两天时间把 `legacy-report/` 目录下的 12 个文件逐行读完，还认真写了一份笔记。第三天顺口问了同事一句，才知道这个模块三年前就下线了，代码一直没删。

这不是能力问题，是把「读完」当成了目标。读完是过程，能改才是目标。

### 硬读的三宗罪

1. **线性**。代码之间的关系是一张网，但你的阅读顺序是一条线。沿着线走，永远走不到网的中心。
2. **无优先级**。在文件系统里，一个每天被调用十万次的核心模块，和一个三年前就没跑过的脚本，长得一模一样。
3. **不可中断**。你读到第 400 行被叫去开会，回来之后要花时间重新找到「我刚才在哪儿、在想什么」。

**硬读：从第 1 行开始，按文件顺序推进**



**目标驱动：先写下要改什么，再决定读什么**



同样是「读懂了一个模块」，两条路径的投入产出比可能差十倍

### 把目标写在读之前

动手读之前，逼自己写下一句话：「我这次要完成的改动是 \_\_\_\_。」如果写不出来，说明任务本身还没定义清楚，这时候读代码只会让你更迷茫。

写完这句话，再追问一句：「为了完成它，我最少需要理解哪些东西？」答案通常短得让你意外——**三到五个文件，一到两条主链路。**

#### 12 个文件

硬读的产出  
两天 · 0 行改动

#### 3 个文件

目标驱动的范围  
两小时 · 1 个 PR

## 什么时候该问人，什么时候该自己建图

问人是所有手段里最快的——一句话可能省你两小时。但它也是最贵的：你打断了别人的工作，而对方的回答往往带着「我记得好像是……」的不确定性。

更麻烦的是，口头答案不会沉淀。三个月后另一个人遇到同样的问题，还得再问一遍。而**图是不同的**：它一次建好，可以被反复查询、被新人复用、被工具验证。

### 一条判断标准

#### 问「意图和历史」，查「结构和现状」

凡是关于**为什么这么设计、当初发生了什么、这事儿谁负责**——去问人，而且最好问当事人。

凡是关于**这个在哪、被谁调用、改动会波及谁、整个系统分几层**——别问人，这类问题有确定答案，应该由工具一次性回答。

| 你的困惑            | 问人 | 建图 | 理由             |
|-----------------|----|----|----------------|
| 「为什么当初这么设计？」    | ✓  |    | 历史意图只存在于人脑     |
| 「这个函数被谁调用？」     |    | ✓  | 有确定答案，人不该背这个   |
| 「这个模块还有人维护吗？」   | ✓  |    | 组织信息，代码里查不到    |
| 「改这个字段会影响哪些接口？」 |    | ✓  | 依赖关系可推导，且容易漏   |
| 「整个系统分几层？」      |    | ✓  | 需要全局视角，人给的容易片面 |

「这块代码是干嘛的？」这种问题，对方要么给你一段冗长的背景，要么说「你去看 XXX」。三个更高效的问题：

- 「如果我改了 X，你最担心我漏掉什么？」——逼对方给出风险点，而不是复述代码。
- 「这个模块有没有已经废弃的部分？」——一句话帮你砍掉一半阅读量。
- 「当初有没有考虑过另一种做法？为什么没选？」——这是唯一能拿到真实权衡的提问方式。

#### 本章小结

- ① 「看不懂」分三种，先分类再选方法；
- ② 目录顺序不是理解顺序；
- ③ 硬读最大的问题是没有目标；
- ④ 意图问人，结构查图。



# 把代码变成 一张能看的图

图上什么可信、什么只是参考，这一章给标准

一张图谱里同时住着两种东西：从语法算出来的事实，和模型读出来的解释。  
它们的出错方式完全不同，混着用，你就没法判断哪条结论能拿去改代码。

## 先看一个例子

化验单和病历，是两种性质完全不同的东西。**化验单是机器测出来的数**——白细胞 12.3，这个数不会因为换了个医生就变。**病历是医生的判断**——「疑似细菌感染」，换个医生可能写成「病毒感染可能」。

两者都要看，但你**不会拿医生的推测去跟化验单较真**。代码图谱也一样：哪部分是「测出来的」，哪部分是「读出来的」，心里得分得清。

## 为什么是图，不是文档

接手陌生代码时，多数人的第一个念头是：「让 AI 帮我写一份架构文档吧。」这个念头很自然，但它藏着一个前提——**文档能描述结构**。

实际上，文档只能描述**某一时刻、某个人理解到的**结构。这两者差别很大。

### 文档的三个死穴

1. **会过期**。代码每天都在变，文档只在写下的那一刻是准确的。三个月后再看，你无法判断哪些还成立。
2. **是线性的**。文档必须按顺序读，可代码之间的关系是一张网。你在第 3 节遇到的问题，答案可能写 在第 11 节。
3. **只给结论，不给证据**。文档说「A 模块调用了 B」，你没法验证，也没法追问「还有谁调了 B」。

### 图不一样在哪

| 维度  | 文档       | 图           |
|-----|----------|-------------|
| 来源  | 人读完之后的总结 | 从源码解析出来的结构  |
| 时效  | 写完即开始过期  | 可以随时重新生成    |
| 读法  | 线性，从头到尾  | 可跳转，从任意节点出发 |
| 可验证 | 基本不能     | 每条边能指回具体文件  |

#### 一句话区分

文档是**别人嚼过的**，图是**源码的结构化投影**。前者告诉你结论，后者给你一条可以自己走的路——而且这条路随时可以重走一遍。

### 但图也不是万能的

图能表达「谁连着谁」，表达不了「为什么这么连」。前者是结构问题，后者是意图问题。把这两件事混在一起，是误读图谱最常见的原因——第 2.3 节会专门讲怎么把两者分开。

## 确定性的一半：解析器拿得准的东西

一张能用的图，必须有一部分是**可复现**的：同样的代码，今天跑和明天跑，得给出同样的边。这部分东西，不能交给大模型。

### 哪些属于「确定性的那一半」

| 结构事实   | 说明                       |
|--------|--------------------------|
| 文件间引用  | 谁 import 了谁、谁 require 了谁 |
| 符号定义位置 | 函数、类、接口分别定义在哪个文件的哪一行     |
| 调用关系   | 哪个函数调用了哪个函数              |
| 继承与实现  | 谁继承了谁、谁实现了哪个接口           |

这些都可以从语法树上直接读出来，做法是把源码解析成语法树，提取结构事实，并预先建立一张引用映射表——后续分析文件时直接查表，不必反复重新推导。

### 为什么必须由解析器来算

#### 可复现

解析器对同一份代码  
永远给同一个答案

#### 不幻觉

它不会脑补出  
一条不存在的依赖

#### 可追溯

每条关系都能指回  
具体的文件和行号

换模型来做这三件事，你会得到三种麻烦：同一份代码两次分析结果不一致；凭空多出一条根本没写过的调用；以及最要命的——你没法验证它说的对不对。

#### 把这条原则记下来

**凡是能从语法确定的东西，不要交给模型。**这不只是建图的规则，它是所有「AI + 工具」系统的通用分工原则：让确定的部分保持确定，模型只负责它真正擅长的模糊地带。

## 语义的另一半：解析器给不了的东西

解析器能告诉你 `auth.ts` 导出了 7 个函数、引用了 3 个模块。它没法告诉你，这个文件管的是「用户登录」还是「权限校验」——虽然你一眼就看得出来。

这个差距，就是语义层的全部价值。

### 只有模型能给的三样东西

1. **一句话摘要**。「这个文件负责把第三方登录回调转换成内部用户对象」——这种句子写不进语法树。
2. **分类标签**。它属于哪一层：对外接口、业务编排、数据访问、界面展示，还是通用工具。
3. **业务映射**。它落在哪个业务流程里——注册、支付、退款、对账。

这三样东西的共同点是：**它们都不在语法里**。你把语法树翻到底，也找不到「这是登录逻辑」这句话。意图是写代码的人脑子里的东西，只能靠读、靠推断。

### 但语义层有代价

| 风险  | 具体表现              | 应对方式          |
|-----|-------------------|---------------|
| 不稳定 | 同一个文件分析两次，标签可能不一样 | 把标签当参考，不当事实   |
| 会犯错 | 把工具类误判成业务层，或反之    | 关键路径人工复核一遍    |
| 难验证 | 摘要写错了，从字面上看不出来    | 抽查一批，建立对它的信任度 |

#### 最危险的用法

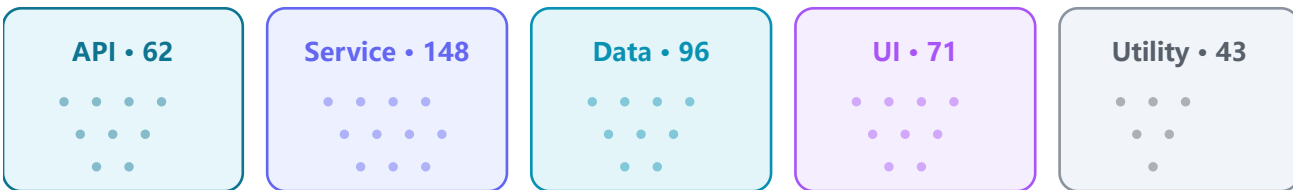
把语义层的结论当成事实直接使用。摘要和标签是**帮你缩小搜索范围**的，不是**替你做决定**的。

看见「这个文件是 Service 层」，你的正确反应是「那我去看看它」，而不是「那它一定不能有数据库调用」。

# 分层：把 800 个文件压成 5 个色块

人的工作记忆一次只能装下四五个组块。面对 800 个文件，你不可能同时思考它们之间的关系——这不是意志力问题，是硬件限制。

所以必须压缩。分层是目前最有效的一种：把 800 个节点按职责归成 5 类，先看清骨架，再决定往哪儿下钻。



每个圆点代表一个文件，共 800 个；方块内只画出示意密度

先按层看清骨架，再决定下钻到哪一堆——而不是从第一层第一个文件开始逐个读

## 五个层各是什么

| 层       | 职责               | 典型文件                         |
|---------|------------------|------------------------------|
| API     | 对外接口、路由、参数校验     | controller / route / handler |
| Service | 业务逻辑编排，一次请求的完整流程 | service / usecase / manager  |
| Data    | 持久化与外部依赖         | repository / dao / client    |
| UI      | 展示与交互            | component / view / page      |
| Utility | 不含业务含义的通用工具      | utils / helper / format      |

### 关键认知：分层是「解释」，不是「事实」

同一个文件，不同的人、不同时期的模型，可能分到不同的层。这不是错误——**分层本身就是一种判断，而判断没有唯一解。**

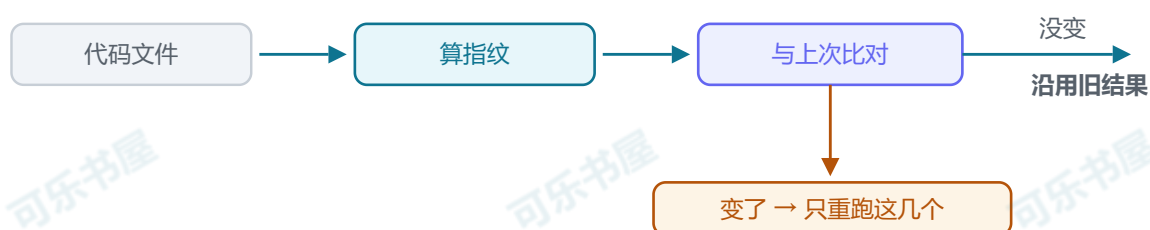
所以：用分层来**确定阅读顺序和范围**（先看哪一层），不要用它来**下结论**（「这个文件一定是 Service」）。

## 增量与指纹：代码改了，图怎么跟上

一张图最大的敌人不是画得不好，是**过期**。过期的图会以「权威」的姿态给你错误信息——这比没有图更危险，因为你会信它。

### 指纹：只重新分析变了的文件

做法很朴素：给每个文件的内容算一个指纹。下次分析时先比指纹，没变就直接沿用上次的事实，只有指纹变了的文件才重新解析。



首次全量分析很贵，之后的增量通常只涉及少数几个文件

### 让图跟着提交走

更省心的做法是挂一个提交后钩子：每次 commit 之后自动做一次增量更新。这样图和代码始终同步，团队里任何人拿到最新代码，图就是对的。

#### 但别太信自动更新

自动更新只能保证「**没变的部分确实没变**」，它不保证「**变了的那些分析对了**」。做过大范围重构（换目录结构、批量重命名、拆包）之后，增量更新的结果可能整体失真——这时候值得跑一次全量重建。

# 四条按图索骥的路径

有图不等于会读图——从哪儿进去，决定你能不能走出来

一张图上有几百个节点，你能问它的问题却有四种完全不同的类型。用错入口，你会在一堆无关的节点里绕圈。

## 先看一个例子

在图书馆找书，有四种找法：**不知道要看什么，就顺着分类架一层层逛**（导览）；**心里有个问题，去查主题索引**（领域视图）；**想知道这本书被谁引用过**，去查引文（影响分析）；**实在找不到，去问管理员**（提问）。

四种找法没有优劣，但**你得先知道自己是哪一种**——抱着一本具体书名去逛书架，是最笨的那一种。

## 引导导览：按依赖顺序读，不是按目录顺序

如果你完全不知道从哪儿开始，最省事的一步是：**先跟着导览走一遍。**

导览做的事情很简单——把「谁依赖谁」算出来，然后按依赖顺序给你排一条阅读路线。排在前面的是被依赖最多的基础模块，排在后面的是组装它们的上层逻辑。

### 为什么依赖顺序是对的

| 读法   | 你遇到的情况                         | 代价         |
|------|--------------------------------|------------|
| 目录顺序 | 读第 3 个文件时，它引用了第 17 个文件里还没见过的概念 | 反复来回跳，思路打断 |
| 依赖顺序 | 读每个文件时，它用到的东西你都已经看过了           | 几乎不用回跳     |

这也是为什么「按目录顺序读」体验那么差——你不断撞见还没建立认知的概念，每撞一次就要往回翻一次。

### 导览怎么用才有效

1. **第一遍只看不停。**不要打开任何一个文件，就把导览从头翻到尾，建立一个粗糙的全局印象。这一步的目标是「知道有这些东西」，不是「理解它们」。
2. **标记三处：**和你任务相关的、看起来最核心的、明显不懂的。剩下的暂时忘掉。
3. **第二遍只读你标的三处，**这时候再打开具体文件。

#### 别把导览当标准答案

导览是按依赖关系自动排的，它不知道你的任务是什么。**如果它推荐的顺序和你的目标冲突，以你的目标为准。**导览的价值是「降低开始的门槛」，不是「规定唯一的路径」。

## 领域视图：从业务流反推代码位置

结构视图回答的是「代码怎么连」，领域视图回答的是「业务怎么走」。这是两个不同的问题，很多人只用了前者。

典型场景：产品经理问你「退款流程现在走到哪一步会通知用户？」，而你手上只有一张模块依赖图。依赖图告诉你 `refund.ts` 引用了 `notify.ts`，但它不告诉你这两者构成的是「退款」这个业务流。

### 领域视图提供什么

| 层级 | 含义                | 例子            |
|----|-------------------|---------------|
| 领域 | 系统支撑的一整块业务        | 订单、支付、履约、售后   |
| 流程 | 某个业务目标下的一串动作      | 退款流程、对账流程     |
| 步骤 | 流程中的单个环节，通常落到具体代码 | 校验订单状态、调用支付网关 |

这三层的价值在于：它把代码节点重新组织成了业务的形状。你可以从「退款」这个业务词出发，直接落到几个文件上，而不必先猜它可能在哪个目录。

### 什么时候最该用它

- **需求评审之前。**先把改动涉及的业务流走一遍，你会提前发现「这个需求要动三个领域」。
- **写交接文档的时候。**业务流比模块依赖更容易被非技术同事理解。
- **排查线上问题时。**从「用户下单失败」这条业务流入手，比从报错堆栈里猜模块要快。

#### 和分层一样，领域划分也是「解释」

同一个功能的代码，不同的分析可能归进不同的领域。这不是错误——**业务边界本身就是人为划的**。领域视图的正确用法是**给你一个搜索入口**，不是**给你一份权威的业务边界定义**。

# 影响分析：动手之前先看波及面

改代码最贵的不是写，是漏。漏掉一个调用方，测试环境可能照样过，上线之后才炸。

影响分析要回答的就是这一件事：**我动了这里，还有谁会被牵连？**

## 先看入边，再看出边

| 方向       | 含义            | 该关心什么        |
|----------|---------------|--------------|
| 入边（谁依赖我） | 别的模块引用了我要改的东西 | 会不会破坏它们的假设   |
| 出边（我依赖谁） | 我要改的东西引用了别的模块 | 我的改动会不会被下游抵消 |

顺序很重要：**入边的风险远大于出边**。改自己依赖的模块，最坏情况是功能不对，你很快会发现；改被别人依赖的模块，最坏情况是别的功能悄悄坏掉，而你在几天后才知道。

## 三种改动的风险等级

低

新增文件 / 新增函数  
没人依赖你，影响面为零

中

修改函数内部实现  
入边不变，只要行为兼容就安全

高

改签名 / 改公共类型  
所有入边都可能断

### 图谱给的是「静态影响面」

它能算出代码层面谁引用了谁，但算不出**运行时的**间接影响：反射调用、依赖注入容器、消息订阅、配置文件里的类名。

所以规则是：**静态影响面用图看，动态影响面靠经验和测试兜**。越靠近框架边界的地方，越不能只信图。

## 提问式阅读：把代码库当对话对象

前三节都是「看图」，这一节是「问图」。当你心里已经有了一个具体问题，直接问比在图上找更快。

### 好问题和坏问题

| 问了等于没问       | 能直接拿到线索                 |
|--------------|-------------------------|
| 「这个项目是干嘛的？」  | 「对外提供哪几个接口？分别由哪个文件处理？」  |
| 「这里为什么这么写？」  | 「这个文件最近三个月被改过几次？谁改的？」   |
| 「帮我解释一下这段代码」 | 「这个函数的入参在哪些地方可能是空的？」    |
| 「有没有坑？」      | 「这个模块被引用了多少次？引用方分布在哪儿？」 |

规律很清楚：**好问题是可验证的**。「哪几个文件处理登录」有确定答案，你可以去核对；「这个项目是干嘛的」听起来重要，但拿到的答案你没法验证，也没法直接用。

### 什么时候该停下来

提问式阅读有个陷阱：它太舒服了。你一句一句地问，每句都能得到看起来合理的答案，于是你一直问下去，却始终没有读一行真实代码。

#### 停下来的信号

当答案开始重复，或者你发现自己在问「那这个呢」「那那个呢」——**停下来，打开那个文件**。提问的作用是**帮你决定读哪个文件，不是替代读文件**。

#### 本章小结

① 不知道从哪儿开始时用导览；② 从业务词出发用领域视图；③ 动手改之前用影响分析，且先看入边；④ 有问题就直接问，但问题要可验证，问完要落到具体文件上。

# 让这张图 活下来

图不是画完就完了——它要么被维护，要么变成负担

建一张图只要几十分钟，让它在一个季度后还有用，需要一套习惯。这一章讲怎么维护、怎么共享，以及最难的——什么时候该承认它帮不上忙。

## 先看一个例子

纸质地图和导航 App 的区别，不在精度，在**更新机制**。纸质地图印出来那一刻就固定了，路改了它不知道；导航 App 会更新，所以它更可信。

但导航也会错——新修的路还没收录时，它会把你带到一条不存在的路口。**图谱和导航一样：默认它是对的，但在关键决策前，值得自己看一眼真实路况。**

## 过期的图，比没有图更危险

没有图的时候，你知道自己在摸黑，所以会格外小心。有一张**看起来还很完整**的过期图时，你会放心地照着它做决定——这才是问题。

### 图是怎么腐化的

| 腐化方式 | 表现           | 危害               |
|------|--------------|------------------|
| 文件消失 | 代码删了，图上的节点还在 | 你会去研究一个不存在的模块    |
| 路径改变 | 文件重命名或移动了位置  | 点开节点找不到文件        |
| 关系漂移 | 依赖被重构了，边还是老的 | 影响分析给出的结论是错的     |
| 新增缺失 | 新写的模块没进图     | 你以为改完了，其实漏了新的调用方 |

前两种你很快能发现——点开一个节点报错就知道了。真正麻烦的是后两种：**它们不会报错，只会安静地给你错误答案。**

### 三个判断信号

1. **节点数和文件数对不上。**差得越多，图的可用性越低。
2. **导览里出现了你确定删掉的东西。**说明至少有一段时间没更新过。
3. **你最近改过的东西，图上找不到。**这是最直接的信号——说明图落后于当前代码。

#### 维护的节奏

**日常靠增量：**挂上提交后钩子，每次 commit 自动补一次，成本很低。

**重构靠全量：**换目录结构、批量重命名、拆包分包之后，增量更新的结果可能整体失真，这时候跑一次全量重建更省事。

## 团队共享：一份图，省掉重复劳动

图谱本质上就是一个 JSON 文件。这意味着一件很划算的事：**一个人分析一次，整个团队可以直接用**，不需要每个人都跑一遍。

### 提交什么，不提交什么

| 内容          | 是否提交 | 原因               |
|-------------|------|------------------|
| 主图谱文件       | 提交   | 这就是共享的那份成果       |
| 配置文件（如语言选择） | 提交   | 保证大家看到的内容一致      |
| 中间产物目录      | 不提交  | 本机临时文件，体积大且无复用价值 |
| 本地差异覆盖文件    | 不提交  | 只对当前工作区有意义       |

另外提醒一句：如果图谱文件本身就超过十兆，直接提交会让仓库变得很沉。这种情况用 Git LFS 托管，使用方式和普通文件一样，只是背后走了单独的大文件通道。

### 共享之后，谁最受益

#### 新人

入职第一天就有全景图  
不必拉着老同事问三小时

#### 评审

PR 里能直接看到  
这次改动波及了哪些模块

#### 文档

架构图可以随代码更新  
不再靠人自觉维护

#### 注意使用方式

图谱没有 LLM 也能看——它自带一个只读的本地查看器，**不需要每个人都会用命令行工具，也不需要联网**。数据全部从本地磁盘读取，不会上传到任何地方。

## 新人的第一周：把图谱嵌进 onboarding

如果你带过新人，应该熟悉这个场景：第一周大部分时间花在「找人问」和「对着代码发呆」上。图谱能把这两件事压缩掉大半。

### 五天的安排

| 时间    | 做什么                  | 目标              |
|-------|----------------------|-----------------|
| 第 1 天 | 跟一遍引导导览，不打开任何文件      | 知道系统里有哪些东西，不必理解 |
| 第 2 天 | 挑一条和自己职责相关的业务流，走完    | 把业务词和文件对应起来     |
| 第 3 天 | 接一个真实的小 bug，改之前先看影响面 | 体验一次「先看后改」的完整流程 |
| 第 4 天 | 写下：这个系统分几层，我负责哪一块    | 把输入变成输出，暴露没懂的地方 |
| 第 5 天 | 把自己的理解讲给同事听          | 讲不清楚的，就是还没懂透的   |

第 4 天那一步最容易被跳过，但它最关键。**看图和讲图是两种能力**——看图是接收，讲图是重构。很多人看完图觉得懂了，一开口就发现全是缺口。

#### 对带人的一方

别把图当成「可以少讲两句」的理由。图谱能替代的是「**这个在哪儿**」这类问题，替代不了「**我们为什么这么做**」。后者才是老同事真正不可替代的部分——而且恰好是你最该讲的那部分。

## 图的边界：它解决不了什么

前面四章都在讲图谱能做什么。这一节讲它不能做什么——**知道边界在哪，比多掌握一个功能重要得多。**

| 图谱擅长回答       | 图谱回答不了      |
|--------------|-------------|
| 这个文件在哪、属于哪一层 | 哪段代码是性能瓶颈   |
| 这个函数被谁调用     | 多线程下会不会出竞态  |
| 改动会波及哪些模块    | 业务规则为什么这么定  |
| 系统整体分成几大块    | 当初为什么选了这套方案 |
| 新写的模块有了合理位置吗 | 运行时数据实际怎么流转 |

右边这一列有个共同点：**它们要么依赖运行时信息，要么依赖人的意图，两者都不在静态结构里。**

### 三条使用原则

1. **结构问题信图，行为问题信测试。**「谁调用了它」看图谱，「它跑起来对不对」看测试和监控。
2. **事实信图，意图信人。**图谱告诉你 A 调用了 B，但只有当事人能告诉你为什么。
3. **图是起点，不是终点。**它的价值在于把你送到该读的那几个文件面前，读不读、怎么读，还是你自己的事。

#### 一个有用的心态

把图谱当成**脚手架**：它的作用是让你够得着高处，楼盖好了就该拆掉。如果三个月后你还在天天盯着同一张图看，可能说明你对这个系统仍然没有形成自己的理解——那才是真正要解决的问题。

## 八个误读图谱的坑：现象 / 原因 / 修法

| 现象                       | 原因                | 修法                      |
|--------------------------|-------------------|-------------------------|
| 把模型写的摘要当成定论              | 没分清语义层和结构层        | 摘要只用来缩小范围，关键结论回到源码确认    |
| 「图上没连线，说明两者没关系」          | 把「没解析出来」当成「不存在」   | 图谱只覆盖静态引用；反射、配置、消息要另找   |
| 「它标了 Service 层，肯定不能碰数据库」 | 把分层的解释当成了硬约束      | 分层是阅读提示，不是架构规范；以实际代码为准  |
| 只看模块依赖，答不上业务问题           | 只用了结构视图，没用领域视图    | 业务问题从领域视图进，技术问题从结构视图进   |
| 照着一个月前的图做重构              | 没有更新机制，图已过期       | 动手前先跑一次更新，对不上就重建        |
| 把调用边当成执行顺序               | 混淆了「静态依赖」和「运行时流程」 | 执行顺序看日志和链路追踪，图谱只给依赖     |
| 改完自测通过，上线还是出问题           | 只看了入边，漏了动态调用方     | 影响分析要叠加「谁通过配置/注入拿到了我」   |
| 有了图之后反而更少读代码             | 把工具当成了替代品         | 把图谱定位成「导航」，读代码仍是不可省略的一步 |

### 这八条的共性

它们都不是工具的问题，而是把「解释」当成了「事实」。图谱里只有一半是事实，另一半是推断——分清这两半，上面八条会自动消失一大半。

### 怎么用这份对照表

不要在读完之后一次性对照。更有效的做法是：**每次接手新项目时把这八条摆在旁边**，当你在「这个结论能不能直接拿去用」上犹豫时，回来扫一眼。八条里总有一条正好说中你当下的处境。

## 接手陌生代码库自检清单 (24 条)

按时间顺序排列。每一项都能独立勾选，不需要按顺序全做——但漏掉的那几项，通常就是后来出问题的地方。

### A • 接手之前 (5 条)

- ☐ 1 确认这次的任务边界：要改什么，改到什么程度算完成
- ☐ 2 问清楚这个模块是否仍在使用，有没有已废弃的部分
- ☐ 3 搞清楚谁是这个模块的负责人，关键时刻找谁
- ☐ 4 确认当前分支和版本，别在旧代码上做判断
- ☐ 5 先跑一遍项目，确认本地环境能正常启动

### B • 建图时 (5 条)

- ☐ 6 确认分析范围覆盖了整个仓库，不是只扫了一个子目录
- ☐ 7 确认输出语言设置正确（否则摘要全变成英文）
- ☐ 8 首次全量分析尽量用订阅额度或本地模型，避免额度超支
- ☐ 9 检查节点总数和文件总数是否大致对得上
- ☐ 10 抽查三个你熟悉的文件，看摘要和分层对不对

### C • 读图时 (5 条)

- ☐ 11 先用导览建立全局印象，第一遍不打开任何文件
- ☐ 12 标记出「与任务相关」「最核心」「完全不懂」三类节点
- ☐ 13 对存疑的结论，回到源码确认，别直接采信
- ☐ 14 涉及业务问题时，切换到领域视图再看一遍
- ☐ 15 图上没有的关系，不等于不存在——记下来单独确认

### D • 动手改动之前 (5 条)

- ☐ 16 查入边：还有谁依赖我要改的东西
- ☐ 17 查出边：我依赖的东西，改了会不会被下游抵消
- ☐ 18 列出动态调用方：配置、注入容器、消息订阅、反射
- ☐ 19 确认公开签名和公共类型是否需要保持兼容
- ☐ 20 想清楚回滚方案，再动手

### E • 长期维护 (4 条)

- ☐ 21 挂上提交后自动更新，让图和代码同步
- ☐ 22 每季度或大重构后，跑一次全量重建
- ☐ 23 把图谱提交进仓库，让新人第一天就能拿到
- ☐ 24 如果三个月没打开过这张图，重新评估它还有没有价值

# 把清单变成习惯

24 条全记住是不可能的，也没必要。真正需要长期留住的只有两件事：一份使用顺序，和一条每周花五分钟做的检查。

## 怎么用这份清单

**不要一次全做完。**第一次接手新项目时，先做 A 组和 B 组；等到真正要动代码那天，再做 D 组。C 组的五条应该贯穿整个过程，E 组是团队层面的，需要有人认领。

最有价值的其实是第 24 条——**一张没人看的图，和不存在没有区别。**定期检查这一条，比认真勾完其他 23 条更能省时间。

## 每周五分钟自检

不用每次都翻清单。每周固定花五分钟，做这三件事，就足以判断手上的图还值不值得信赖：

| 动作   | 看什么               | 红灯信号        |
|------|-------------------|-------------|
| 数一数  | 节点总数和仓库文件总数差多少    | 差距比上周明显变大   |
| 点三个  | 随机点开三个节点，能跳到真实文件吗 | 有任意一个跳不过去   |
| 回一次头 | 想一下这周改过的文件，图上有没有  | 改过的东西一个都找不到 |

这三件事加起来不超过五分钟，但它们分别对应了图谱腐烂的三种方式：**规模对不上、路径失效、新增缺失**。任何一条亮红灯，就该跑一次更新了。

## 清单之外，三条经验

**一、先看被引用最多的，不看最早写的。**历史遗留的顺序里，最核心的模块往往不在最前面——创建时间和重要性没有关系。

**二、每读完一个模块，用一句话写下它的职责。**写不出来，说明还没读懂。这比任何笔记方法都有效，因为它逼你从「读过」走到「理解」。

**三、把「我猜的」和「我确认的」分开记。**过三天再回来看，你会发现当时以为确定的事，有相当一部分其实只是推测。

# 图谱只是脚手架

回头看这四章，其实只讲了四件事：搞清楚自己卡在哪、分清图上的事实和解释、知道从哪儿进去、让图活下来。

没有一件是「学会某个工具」。工具会换，命令会改，今天好用的插件明年可能就没人维护了。但这四件事不会变——只要你还在一张别人写的代码网里找路，它们就有用。

## 关于「看不懂」这件事

最后想说的是：「看不懂别人的代码」不是能力缺陷，它是一种**正常状态**。

任何一个十万行以上的系统，都不存在「全部看懂」的人。维护它五年的人，也只熟悉其中一部分。真正专业的表现不是「什么都懂」，而是**知道自己在哪一块不懂，并且知道怎么快速补上**。

这本书提供的所有方法，本质都在服务这一件事：把「我不懂」这三个字，变成「我不懂哪个具体的东西」。前者是情绪，后者是待办事项。

## 下一步

如果你刚读完，最好的动作不是再读一遍，而是：**打开你手上那个最让你头疼的仓库，按第 1 章的分类表给自己定位一次**。哪怕只是写出「我卡在没上下文」，也比读完整本书但不动手要有用。

### 最后一句

脚手架搭起来是为了盖楼，楼盖好了就该拆。**当你不再需要这张图的那一天，说明你真的读懂了。**



可 乐 书 屋

# 别人的代码看不懂？

四章 · 十七个专题 · 一本讲透  
不是看不懂，是缺一张能问的图

工程实战系列 · 第二册  
可乐书屋 出品 · A4 图文版