一个人的产线

导读

不会写代码的工业产品经理，如何用 AI 把设备从想法做到上线、连算法都自己啃下来

读这本书之前，先看它不是什么

市面上跟这本书沾边的东西，大概分两类。

一类是 **AI 编程教程**，教你怎么把 AI 当敲键盘的手：你说一句，它吐一段代码。这类东西有个默认前提——**你已经知道自己要写什么**。它假设难点在「实现」，帮你把实现变快。可对一个不写代码的产品经理来说，卡住你的从来不是打字慢，而是「这台设备到底该做什么、做到什么程度算对」这件事，你自己都还没想清楚。AI 敲得再快，也救不了一个模糊的意图。

另一类是 **产品经理转型鸡汤**，给你情绪、给你身份认同、给你一句「AI 时代人人都是工程师」，然后就没有然后了。你合上书，热血两小时，第二天面对一台真实的设备，还是不知道从哪儿下手。

这本书填的是这两类东西中间那道**真鸿沟**：

> 大白话
>
> 怎么把你脑子里那个模糊的产品想法，翻译成 AI 能准确执行的指令；等 AI 交给你一大堆「看起来能跑」的代码之后，你怎么判断它到底对不对、放到产线上会不会出事、会不会某天半夜把一批好料判成废品。

前半段是「把想法逼成规格」，后半段是「不靠信任、靠验证」。这两件事，恰恰是 AI 替你做不了、也最不该替你做的。

这本书不承诺什么

我先把话说死，省得你带着错的期待往下读：

- **不承诺写代码变简单了。**代码这道墙没有塌，它只是挪了位置——从「会不会写」挪到了「会不会想清楚、会不会验证」。后面这两样，一点都不比写代码轻松。

- **不承诺你能做任何工业系统。**有些系统的物理约束、安全等级、认证门槛，不是一个人加 AI 能扛下来的。这本书会明确告诉你边界在哪（最后一章专门讲这个），而不是假装边界不存在。
- **不灌鸡汤。**你不会在这里读到「相信自己」「拥抱变化」。你会读到「这里有个坑，长这样，AI 特别容易在这儿骗你，你这样验它」。

换句话说，这本书对你**诚实**。它宁可让你少一点兴奋，多一点「哦，原来是这么回事」。

谁适合读，带着什么问题读

如果你是这样一个人——**懂业务**（知道这台设备该解决什么问题）、**肯把事情想清楚**（受得了把模糊想法一点点逼成精确规格的折磨）、**较真**（AI 说「搞定了」你不信，非要自己验一遍）——那这本书就是给你写的。

反过来，如果你指望它教你一套话术，让 AI 替你把活全干了、你翻页就能交差，那你会失望，早点合上对咱俩都好。

带着这一个问题读，从头到尾都别丢：

> 一个不会写代码的产品经理，怎样**真的**（不是自我安慰）借 AI，把一个工业智能装备从 0 做到上线，连算法那部分自己也能落地？

这里的赌注很清楚：在 AI 时代，**「知道这台设备该做什么」比「会打字实现它」更稀缺、更值钱**。会实现的人，AI 帮你补齐了；会想清楚、会验证的人，AI 替不了。这本书赌的就是后面这半句。

怎么读

全书 15 章是**递进**的：从「门槛搬了家」讲起，到「AI 是个概率性意图翻译器（它是在猜你大概想要什么，不保证给你的是对的）」，再到搭出第一个能点的原型、啃下算法、部署到边缘盒子、最后是一个全能工程师真实的边界。按顺序读，你会攒出一种「闭环手感」。

但你不欠这本书一个「从第一页老实读到最后一页」。你要是正被视觉缺陷检测卡着，直接翻第 9 章；正在跟一台设备的脏数据死磕，直接翻第 7 章。**从你最疼的那一章切进去**，回

头再补前因，完全可以。

大白话

一句话收尾，落在动作上：合上这页，去挑一台你手头真实的、卡了很久的设备——不是想象里的完美案例，就是那个让你头疼的。这本书的每一章，都是陪你把那台设备往前推一步的工具。想清楚它该做什么，再让 AI 去实现；实现完，你亲手验它对不对。就从这一台开始。

第一章 · 门槛搬家了

先说一个也许让你不舒服的判断：**过去十几年里，把「有个好想法」挡在门外的那道门槛，多数时候不是聪明，是打字。**

你脑子里想清楚了「设备检测到瑕疵就报警」，这句话本身没有任何门槛。真正卡住你的，是下一步——你得把这句话翻译成一行行机器能执行的指令：连哪个变量、判断哪个条件、循环多少次、异常怎么接住。这个翻译动作，就是写代码——用一种极其死板、错一个符号就罢工的语言，把你的意图逐字逐句誊写给机器。

过去，会不会这个「誊写」，是一道又高又硬的墙。墙这边是「懂业务、想得清楚的人」，墙那边是「能让机器动起来的人」。你要么自己翻墙（学三年编程），要么雇个翻墙的人（招工程师、外包）。而现在，这堵墙没有消失——**它只是搬了个位置**。

门槛不是被砸掉的，是被搬走的

这里我要非常小心，因为满大街都在喊「人人都能编程了」——这是鸡汤，我不打算给你灌。真相要精确得多。

AI 搬走的，是**「实现」这一段**：把想法誊写成代码这个体力活，它干得比大多数人快、比大多数人对。你说「读取温度，超过 80 度就停机」，它几秒钟给你一段能跑的东西。这一段——把已经想清楚的逻辑逐字敲成代码——历史上吃掉了工程师相当一部分工时，现在被搬空了大半。

但门槛这个东西，**守恒**。它不会凭空消失，只会换个地方站岗。它搬去了哪儿？搬去了两个你反而更擅长的地方：

- **会「想清楚」**：到底要它做什么？「瑕疵」是什么？多大算瑕疵？漏检和误报哪个更要命？——AI 不会替你定义这些，它只会忠实地实现你说出口的那一版，哪怕那一版是错的。
- **会「验证」**：它给你的东西，到底对不对、能不能上产线？AI 会一脸自信地给你一段有 bug 的代码，就像一个从不说「我不确定」的实习生。**判断它对错的责任，一分都没搬走，全落在你头上。**

大白话

说人话：以前你付出的代价是「学会打字实现」，现在你付出的代价是「想清楚 + 会检查」。前者需要三年，后者——你做产品经理这些年，本来练的就是这个。门槛没变矮，是换到了你的主场。

这事历史上发生过，不止一次

怕你觉得这是我编的新词，我给你看一个真实发生、且结局清楚的例子——摄影术，也就是照相机的发明。

照相机出现之前，你想留住一张脸、一处风景，唯一的门槛是「会画画」——得练手、练透视、练光影，几年功夫。1839 年照相机被公开，往后几十年里它一步步普及，「会画画」这道手上功夫的门槛，被慢慢搬走了：按一下快门，比最好的画师画得还像。

那画画的门槛消失了吗？没有。它**搬去了「会构图」**——拍什么、怎么取景、什么时候按快门、光从哪来。手上的活机器包了，但「看见什么值得拍」这件事，机器一点忙帮不上。结果是：大批只会「画得像」的匠人失业了，而一批「眼睛好、想法好」的人，借着相机成了摄影师，产量翻了几百倍。

你注意这个结构，它和今天一模一样：

- 被搬走的：**手上的实现功夫**（画得像 / 会打字实现）。
- 被搬到的、反而更值钱的：**脑子里的判断**（看见什么值得拍 / 想清楚要做什么、会验证做得对不对）。

如果你是那个「只会画得像」的匠人，这是坏消息。但你不是。你是那个「有想法、会判断」的人——过去你被挡在墙外，是因为你不会那道手上功夫。**现在功夫这一段被机器包了，你恰好站在了新门槛更值钱的那一侧。**这不是安慰你，这是能力结构真实的位移：值钱的东西，从「会实现」移到了「会想清楚、会验证」，而后者本来就是你的手艺。

诚实地说：这三样，AI 一样没帮你搬走

如果我只讲上面这些，我就成了另一个卖鸡汤的。所以下面这段最重要，请你记牢——有三样东西，AI 结结实实留在了你这边，一点没搬走。全书后面十四章，很大程度上就是在教你

扛起这三样。

1. **判断**：要不要做、做成什么样、哪个指标是命根子。AI 能给你十个方案，但「选哪个」是你的活。它没有立场，你有。
2. **验证**：它给的东西是真能用，还是「看起来能用」？一段代码在你电脑上跑通了，不等于在产线那台落满油污、电压不稳的盒子上跑得住（这是第 11、12 章的事）。**AI 甚至会骗你——不是恶意，是它天生就会一本正经地胡说**（这是第 14 章要专门拆的）。谁来抓它的谎？只有你。
3. **对领域的理解**：你的设备为什么会出这种瑕疵？这个工艺的物理原理是什么？客户产线上真正的痛点在哪？——这些藏在你脑子和现场里的东西，AI 数据库里没有。它是你唯一无法外包、也是你最厚的护城河。

> 大白话
>
> 一句话概括这一章的赌注：**AI 把「会实现」变便宜了，于是「会想清楚、会验证、懂这一行」变贵了。**而后面这三样，正好是一个好产品经理身上最值钱的部分。这本书赌的，就是你能靠这三样，真的把一台工业智能装备从 0 做到上线——不是自我安慰，是真的落地。

所以这本书要跟你把话说死

我不会跟你说「有 AI 你就无所不能了」。恰恰相反，这本书从头到尾都在划一条线：**哪些 AI 真能替你扛（实现、翻译、体力活），哪些永远得你自己扛（判断、验证、对行业的理解）。**把这条线看清楚，你才不会在半路上摔跟头——要么高估它、把胡说当真理上了产线；要么低估它、又退回去老老实实学三年编程。

下一章，我们就干一件很具体的事：掀开盖子，看看 **AI 到底在替你做什么、又坚决不做什么**。把它的能力边界摸清楚，你手里这把新工具，才用得踏实。

> 门槛从来不会消失，它只会搬家。这一次，它恰好搬到了你脚下。

第二章 · AI 到底在替你做什么、不做什么

你已经决定要用 AI 帮你把这个工业产品做出来了。那在动手之前，得先搞清楚一件最要命的事：你手里这个 AI，到底是个什么东西？它擅长什么、会在哪儿翻车？搞不清这个，你会在错误的地方信任它，然后在最贵的地方摔跟头。

这一章不测评谁家模型更强。我们只干一件事：把 AI 的引擎盖掀开，看看里面那台机器到底怎么转的。看懂了这个，后面所有的「它为什么这么好用」和「它为什么突然发疯」，你都能自己解释。

先认清三样东西，别把它们搅在一起

你会打交道的 AI 其实是三个层次，越往上越「主动」，但底子是同一个。

第一样，大模型——就是 GPT、Claude 这类你打字问它、它打字答你的 AI。它读文字、写文字，本质是一张嘴加一颗脑子，但没有手，动不了你电脑里任何东西。

第二样，AI 编程助手——把大模型塞进代码编辑器里，你写代码时它在旁边补全、生成整段代码。它比裸的大模型多了一样东西：**能看见你当前的代码文件**，所以答得更贴你的活儿。但它还是「你踩油门它建议方向」，方向盘在你手上。

第三样，AI Agent——能自己拆步骤、自己动手执行任务的 AI。你说「把这个功能做出来」，它自己去读文件、写代码、跑一下、看报错、再改。它多了「手」，能实际操作你的电脑。这是你在这本书里真正要用来干重活的东西。

> 大白话
>
> 一个是会说话的脑子，一个是会说话且看得见你桌面的脑子，一个是会说话、看得见、还能自己伸手去做的脑子。手越多，能帮你干的活越多，但它出错时能捅的娄子也越大。

关键在于：**这三样东西的内核是同一台机器**。Agent 不是比大模型「更聪明」的新物种，它只是大模型外面套了一层「能循环行动」的壳。所以大模型有的毛病，Agent 一个不少，只是后果放大了。要理解全部三样，你只需要理解那台内核机器怎么转。

那台内核机器，只干一件事：猜下一个词

这是全章最重要的一句话，请慢读：**大模型的全部工作，就是根据前面已经有的文字，预测下一个最可能出现的词，然后把它写下来，再拿这个新的、更长的文字去预测再下一个词。**如此一个字一个字地滚下去。

就这么简单。它不是先想好一整个答案再打字给你，它是像挤牙膏一样，每次只挤出「统计上最顺的下一个词」。

打个比方。你手机输入法打「今天天气真」，它会灰字提示「好」。为什么是「好」不是「香蕉」？因为它把海量文字看下来，「今天天气真」后面接「好」的次数远远压过接「香蕉」。大模型就是这个输入法，只不过它读过的文字多到不可思议，能接的不只是一个词，而是能一路接出一整篇通顺的文章、一整段能跑的代码。

> 大白话
>
> 它不是一个「知道答案的人」，它是一个「极其擅长把话说圆的人」。它追求的是「像不像对的」，不是「是不是对的」。这两件事经常一致，所以它有用；但它俩不是一回事，所以它会骗你。

记住这个词：概率——就是「可能性大小」。它倾向于挑「可能性大」的下一个词，而不是「保证正确」的下一个词——而且挑的时候还留了一点随机（所以同一句话问两遍，答案可能不完全一样）。这是它的天性，改不掉。

为什么这机制让它无比有用

先说好的一面，别以为这是个缺陷产品。

正因为它在猜「最顺的下一句」，所以凡是「有海量范例、答案有套路」的活，它都干得漂亮。写一段读取传感器数据的常规代码、把你一段啰嗦的需求整理成条理清楚的文档、把英文报错翻成人话、照着一个成熟做法给你搭个架子——这些事在它读过的文字里出现过千万遍，它太知道「接下来该怎么写」了。

这就是这本书的底气：**那些「有标准答案、纯体力、你不会但别人早写烂了」的代码活，正是它的主场。**你不需要会写，你需要会把活儿描述清楚交给它。

换个角度看它的真实身份：它是一台**概率性的「意图翻译器」**——你用人话说出你想要什么，它翻译成代码、翻译成文档、翻译成方案。翻译，是它的本行。

为什么这机制又让它一本正经地胡说

现在说要命的一面。既然它只追求「最顺」，那当「最顺的话」恰好不等于「真的事」时，它会毫不犹豫地把假话写得无比顺畅。

这个现象有个专门的词：幻觉——指 AI 编造出根本不存在、或者压根是错的信息，但语气笃定、格式工整，看起来跟真的一模一样。

为什么会这样？因为它脑子里根本没有「我不知道」这个档位。你问它一个某型号 PLC 到底支不支持某个通信协议，如果它读过的文字里这事儿没定论，它不会停下来说「查不到」——那不是「顺」的回答。它会顺着你的问法，编一个听上去最合理的答案，连型号参数都给你编得有模有样。**它不是在骗你，它是在「把话说圆」，而说圆的代价就是可能凭空造事实。**

> 大白话
>
> 它像一个特别会聊天的实习生：什么都敢答，从不说「不知道」，答错了也一脸自信。你不能因为他语气笃定就信他——语气笃定恰恰是他的默认设置，跟对不对没关系。

这里有个反直觉的重点，你务必记牢：**它答得越流畅、越自信，跟它答得对不对，一点关系都没有。**流畅是它的出厂配置，正确要靠你去验。在写网站文案时，胡说一句无伤大雅；但在工业设备里，一句编造的接线定义、一个虚构的寄存器地址，可能就是一台报废的机器或一次安全事故。这就是第 14 章要专门教你「怎么识破它骗你」的原因。

那么，哪些判断该留给你自己

现在可以给你一个能一直用的心智模型了。把活分成两类：

- **「有标准答案的体力活」——放心丢给它。**写常规代码、翻译报错、整理文档、生成样板、解释一段陌生代码在干嘛。这些交给它，你省下的是时间，风险很低。

- **「关乎对错、关乎后果的判断」——你自己扛。**这个方案在你的产线上到底行不行、这个安全逻辑对不对、这个数据它是查到的还是编的、它给的做法适不适合你这台设备。这些是**方向盘**，不能松手。

一句话总结这台机器的本质，请刻进脑子：

> 它是概率性的「意图翻译器」，不是确定性的「正确性保证器」。它负责把你的意图快速变成东西，你负责判断这东西对不对。

你可能会问：那 Agent 能自己跑、自己改，它不就能自己保证对了吗？不能。Agent 只是把「猜下一个词」这件事套进了一个循环——它写一段、跑一下、看报错、再猜下一段怎么改。它确实能自己纠正很多低级错误（因为报错也是文字，它会顺着报错猜修法），但它纠正不了「它压根不知道自己在编」的那类错。**循环放大了它的能干，也放大了它的自信满满地跑偏。**你越是放手让它自动干，你就越要在关键节点设卡验收——这正是后面「让它跑住」和「验证迭代」那几章的活。

这一章，你要带走的

大模型、编程助手、Agent，是同一台「猜下一个词」的机器，手的多少不同而已。它靠概率把话说圆，所以它是个飞快的「意图翻译器」——把你不会写的体力活翻译成成品；也正因为它只求「顺」不求「真」，它会理直气壮地胡说。

所以你和它的分工从今天起就定死了：**体力活丢给它，对错判断留给你。**它跑得多快，取决于你把意图说得多清楚——那是下一章「怎么给它下精确指令」要解决的；它会不会骗到你，取决于你在哪儿设了验收关卡——那是第 14 章的事。这两件事，就是你这个不写代码的产品经理，真正要练的两门手艺。

第三章 · 把「我想做个东西」变成机器能执行的规格

一句「我想做个东西」，里面藏着一百个没说的决定

你走到工程师面前，说：「我想做个能自动检测螺丝有没有拧到位的东西。」

你以为你说了一句话。其实你抛出了一个填空题，而且空白多得吓人：拧到位是拧到什么程度？靠看图片判断，还是靠听声音、测扭矩？漏检一颗和误报一颗，哪个更要命？光线暗了怎么办？螺丝生锈了算不算没拧到位？判断要多快，慢一秒能不能接受？

过去，这些空白是工程师替你填的。他们凭经验、凭猜测、凭「我觉得你应该是这个意思」把它填满——填对了是运气，填错了是你俩下个月的返工。现在你要用 AI 独立落地一个产品，AI 也会替你填这些空。区别在于：**AI 填得比人更快、更自信、而且绝不会停下来问你一句「你确定吗」。**

> 大白话
>
> 模糊的需求，进去的是「差不多」，出来的一定是「差很多」。而且 AI 会用一副胸有成竹的样子把「差很多」交给你，你还以为自己成功了。

Garbage in, garbage out：这句老话在 AI 时代重新长了牙

Garbage in, garbage out（垃圾进、垃圾出）是句计算机领域的老话：喂进去的是烂输入，吐出来的必然是烂结果，机器不会替你把烂的变好。

以前这话主要说数据。现在它多了一层意思：**你给 AI 的需求描述，本身就是一种输入。**你说得模糊，AI 不会报错，它会自动脑补一个版本，然后一本正经地把这个脑补出来的东西做出来给你。

打个比方。你叫一个手脚极快但完全不认识你的临时工去仓库「把那些坏了的箱子扔掉」。他不会问你「哪些算坏」，他会按自己的理解，十分钟内扔掉一半库存，动作麻利，态度积

极，结果是灾难。AI 就是这个临时工——快、勤快、绝对服从，但你没说清的地方，它全凭自己拿主意。

所以在 AI 时代，一个残酷又美妙的事实浮出水面：**你把需求说清楚的能力，直接决定了 AI 给你的是产品还是垃圾。**而「把需求说清楚」，恰好是产品经理的看家本领，不是程序员的。

为什么这件事，PM 比程序员更该干、也更擅长

很多人以为，AI 时代产品经理会贬值，因为「写代码这道墙塌了」。恰恰相反。墙塌了，露出来的是更值钱的东西。

想想一条流水线的分工。程序员擅长的是「怎么把一件明确的事做出来」——给定规格，他能实现。而产品经理擅长的是另一件事：**决定到底要做哪件事，以及那件事长什么样才算对。**过去这两件事被打包在一个团队里，界限模糊。现在 AI 把「实现」这一半几乎免费地包了，剩下没被包掉的那一半——想清楚要什么、说清楚要什么——反而成了整个链条里最稀缺的环节。

这就是本章要教你的唯一一件事：**怎么把一句模糊的意图，逼成一份机器能一步步执行的规格。**

> 大白话
>
> 注意：这一章不讲计算机怎么运转（那是下一章的事）。这一章只讲一件更靠前的事——怎么想清楚、怎么把想的东西表达成 AI 能照着做的样子。你可以完全不懂代码，照样把这件事做到极致。

规格是什么：一份让「对不对」有答案的说明书

规格（specification，常简称 spec）就是一份把「我要什么」写死的说明书。它的核心不是华丽，而是**让每一处都有唯一答案，不给脑补留缝隙。**

一份能用的工业规格，至少要把五件事钉死：

- **输入**：AI 拿到的到底是什么。一张什么分辨率、什么角度、什么光照下拍的图？还是一串传感器数字？多久来一次？
- **输出**：它要吐出什么。是「合格 / 不合格」两个字，还是一个 0 到 1 的分数，还是标出缺陷在图上的哪个位置？给谁看，给谁用？
- **边界**：它管到哪儿为止。哪些情况归它判，哪些情况它该直接说「这个我不管」而不是硬猜。
- **异常**：不正常的时候怎么办。图糊了、传感器断了、来了个它从没见过的东西——它该报警、该停、还是该装没事？
- **验收标准**：怎么算做成了。用什么数字、什么条件来判定「这活儿合格」，而且这个判定不靠你心情。

其中两个词值得单独盯一下，因为它们是新手最容易漏、老手最先想的地方。

边界情况：真实世界总在犄角旮旯里给你惊喜

边界情况（edge case）指那些不常见、但一定会发生的极端或临界情形。正常的螺丝好判，难的是那些卡在中间的：拧了一半的、拧歪的、滑丝的、被油污盖住看不清的、根本没有螺丝孔却硬要检测的。

为什么它重要？因为一个系统在 95% 的正常情况下表现好，是没有意义的——**产线出事，几乎全出在那 5% 上。**你不主动把这些角落想出来写进规格，AI 就会默认它们不存在，然后在某个深夜的产线上,让它们以最贵的方式冒出来。

验收标准：把「差不多得了」翻译成「达标 / 不达标」

验收标准（acceptance criteria）是一句能用「是」或「否」回答的话，用来判定活儿有没有做成。

「检测得准一点」不是验收标准，它是废话——多准算准？谁说了算？

「在标准产线光照下，对拧到位的螺丝，1000 次里漏判不超过 3 次，误判不超过 10 次，单次判断不超过 200 毫秒」——这才是验收标准。它把一个含糊的愿望，变成了一把有刻度的尺子。有了这把尺子，你、AI、以及未来质疑你的老板，才能对着同一个东西说话。

实操：把一句话逼成一份规格

回到开头那句「我想做个能自动检测螺丝有没有拧到位的东西」。我们不写代码，只用追问，把它一层层逼清楚。追问的诀窍很简单——**每问一句，都堵死一种脑补。**

1. **逼输入**：靠什么判断？——决定用工业相机在螺丝正上方 20 厘米、固定环形光下拍一张 1200×1200 的图。这样「靠听声音」「侧着拍」这些歧义就被堵死了。
2. **逼输出**：要什么结果？——每颗螺丝给一个「合格 / 不合格 / 不确定」，外加一个 0 到 1 的把握分。留一个「不确定」的口子，是为了别逼它在没底的时候瞎猜。
3. **逼边界**：什么归它管？——只判「已经装上、且螺帽完整可见」的螺丝。没装的、被完全遮住的，直接归到「不确定」，交给人。
4. **逼异常**：坏了怎么办？——图像过曝、相机没返回、把握分低于 0.6，一律不给结论，亮黄灯叫人来看，绝不硬判成合格。
5. **逼验收**：怎么算成？——用 500 张事先由老师傅标好答案的图来考它，这 500 张里漏判（该拦没拦）必须为 0，误判（好的判成坏的）容忍到 2%，单张耗时不超过 200 毫秒。达标才上线。（注意：500 张全过，不等于真实产线上永不漏判——它只是把你敢上线的底线量化了，后面第十三章会讲怎么在真产线上继续盯漏判率。）

看，同样一句话，追问前是一团意图，追问后是一张任务清单。这张清单，就是你交给 AI 的prompt（对 AI 下达的指令）的骨架——所以说，**需求即 prompt，规格即架构**：你把需求想到多清楚，AI 就能把系统搭到多清楚，一分不多。

大白话

你会发现，逼这五个问题的过程里，你一行代码都没碰，却已经做完了整个产品最难、最值钱的那部分——决定这东西到底该是什么样。剩下的「怎么实现」，才是从下一章开始，你和 AI 一起补的功课。

> 不会写规格，AI 给你的就是一堆自信的错东西。会写规格，AI 才成为你手里那条真正的产线。这两者之间隔着的，不是编程能力，是你愿不愿意把每一个「差不多」，都逼成一个「就是它」。

第四章 · 你必须懂的最小计算机常识

你不用会写代码，但你得看得懂图纸

你做过硬件。你不会亲手车一个零件，但你看得懂机械图纸——知道这是轴承、那是密封圈、这里为什么留一道公差。图纸不让你去当钳工，它让你在别人干活时，知道他干得对不对。

软件也有这样一张图纸。这一章不教你写代码——写代码交给 AI。这一章只给你那张图：让你知道 AI 交出来的东西里，哪块是哪块、各自管什么、少了哪块会出事。

> 大白话
>
> 没有这张图，你会遇到一个具体的麻烦：AI 跟你说「我把数据存好了」「接口调通了」「前端渲染没问题」，你完全没法判断这话是真是假、这样安排合不合理。你只能点头。而点头点错了，是要在现场付代价的。

程序和进程：菜谱，和正在炒的这盘菜

程序是写在硬盘上的一堆代码，静静躺着，什么也不干。就像一张菜谱——写着「热锅、下油、放葱花」，但纸上没有一粒葱花是熟的。

进程是这份菜谱被真正拿去炒了：锅在响、油在冒烟、这一盘菜正在被做出来。同一份菜谱可以同时炒三盘（三个进程），互不相干。你把程序**跑起来**，就产生了一个进程。

为什么要分清这俩？因为现场出问题时，你会听到两种完全不同的说法：

- 「代码没问题啊」——菜谱写得对。
- 「服务挂了 / 进程死了」——正在炒的那盘菜，锅翻了。

菜谱对，不代表菜正在炒。你的设备半夜没数据了，多半不是代码写错，而是那个进程悄悄停了——就像厨子走开了，菜谱还在墙上挂着，但没人炒。知道这个区别，你就会问出对的

问题：**「进程还活着吗？谁负责在它死掉时把它重新拉起来？」** 这个问题，第十二章会专门讲。

内存和硬盘：手边的操作台，和后面的仓库

这是最容易吃亏、也最容易讲明白的一对。

内存是厨子手边的操作台：切一半的菜、刚打好的蛋、马上要用的调料，都摊在上面。拿取极快，但台面就那么大，而且——**一断电，台面上的东西全没了**。

硬盘是后面的仓库：米面粮油、封好的货，码在货架上。存取慢一些，但断电了、关机了，货还在那儿。

> 大白话
>
> 一句话记住：**内存快但断电就空，硬盘慢但断电还在。** 数据要想活过一次断电，就必须从操作台挪进仓库——这个动作叫「存下来」「落盘」「写进数据库」。没落盘的数据，跟没炒完的菜一样，跳闸那一刻就消失了。

这对工业设备是要命的。车间跳电是常事。如果你的检测结果只放在内存里、还没落盘，那这段时间的数据一断电就归零。所以你验收时该问的是：**「数据多久落一次盘？万一这一秒断电，最多丢多少？」** AI 如果告诉你「都在内存里算，很快」，你要立刻警觉——快是快，但它没告诉你断电会丢。

前端和后端：管你看到的，和管背后算的

你打开一个设备的操作界面：有按钮、有实时曲线、有个红色的「急停」。这些你眼睛看得见、手点得着的东西，是前端——它只管「显示」和「接收你的操作」，也做点轻活（比如画曲线、检查你有没有把输入框填空），但真正的重活不在这儿。就像餐厅的服务员：给你菜单、记下你要什么、把菜端上来，也会提醒你「这道菜没写份数」，但他不炒菜。

真正炒菜的在后厨，你看不见——那是后端。它接住前端传来的请求，去调算法、读传感器、算缺陷、把结果存进仓库，再把答案递回前端显示。前端好看不好看是一回事，后端算得对不对是另一回事。

为什么非得分家？

因为这俩变的频率、跑的地方、要的本事，完全不一样。

- **变化频率不同**：界面上按钮挪个位置、颜色改一改，天天可能改；背后的算法逻辑，改一次要重新验证一轮。混在一起，你动个按钮都提心吊胆。
- **跑的地方不同**：前端可以跑在车间墙上那块触摸屏、或工程师的笔记本浏览器里；后端往往跑在现场那台工控机（车间里那台专门干活、皮实耐造的电脑）上，紧挨着设备和传感器。离得近，算得快、丢数据的风险小。

为什么你要在意这个分家？因为当界面卡住时，你得判断是「服务员没反应」还是「后厨罢工了」。前端白屏，可能后端好好的，只是没把菜端上来；也可能后厨真的停了。分不清这两者，你会把厨房的锅甩给服务员，白折腾。

数据库：有结构地码货的仓库

前面说硬盘是仓库，但仓库也分两种。一种是随手把货往地上一堆——文件，比如一张张图片、一个个日志文本，扔进去能找，但想问「上个月周三夜班、3 号工位、所有判为不合格的件」，你就得一件件翻，翻到天亮。

数据库是码得整整齐齐、贴好标签、上了货架的仓库。每条记录都有固定的格子：时间、工位、班次、结果、图片在哪。正因为码得有结构，你才能一句话就把「上月周三夜班 3 号工位的不合格件」全捞出来——快、准、不漏。

> 大白话
>
> 你不用管数据库长什么样、用哪种。你只要记住它存在的理由：**让「事后能按条件快速查、能统计、能追溯」这件事成为可能。** 工业设备最值钱的往往不是当下这一下检测，而是攒下来的数据能回头算良率、找规律、追责任。这些全靠数据库撑着。

所以当 AI 说「数据我存成一堆图片文件了」，你要多问一句：**「那我以后想按工位、按时间段统计，查得动吗？」** 堆在地上的货，和上了架的货，事后差的是天和地。

API：两块软件之间的点菜窗口

最后一个词，也是后面章节出现最多的。

API是两块软件之间说话的窗口，事先约好了「怎么问、怎么答」。就像后厨那个传菜窗口：服务员不能冲进后厨乱翻，他只能站在窗口，按固定格式喊——「一份 3 号桌、宫保鸡丁、不要辣」。后厨照单做好，从窗口递出「这是您的菜」。

关键在**约定**俩字。窗口两边不用认识彼此、不用知道对方内部怎么运作，只要都遵守同一套喊法就行。于是：

- 你的前端界面，通过 API 向后端「点单」：把这张图判一下。
- 你的后端，通过相机厂商给的 API 去「点」一张照片。
- 你甚至能通过 API 把检测结果递给工厂原有的 MES 系统，不用管它是哪年谁写的老古董。

> 大白话
>
> API 的价值就是这个：**让互不相识的两块软件也能协作，只要约定好点菜窗口的规矩。** 你后面会反复听到「调用某某 API」——翻译过来就是「隔着窗口，按约好的格式，问它要个东西」。

为什么产品经理必须懂这个？因为你要对接的东西——相机、PLC、工厂的老系统——几乎都是通过 API 打交道。当 AI 说「这个相机没有开放 API」，你就该知道：这不是小麻烦，这是那个传菜窗口根本没开，你的软件没法正经跟它说话，得另想办法。这类坑，第七章会带你趟。

把这张图收进兜里

就这么六样，凑成你的图纸：

- **程序 vs 进程**：菜谱，和正在炒的这盘菜。代码对，不等于服务活着。
- **内存 vs 硬盘**：操作台断电就空，仓库断电还在。数据要落盘才算数。
- **前端 vs 后端**：服务员管你看到点到的，后厨管背后算的存的。
- **数据库**：上了架、贴了标签的仓库，事后才查得动、追得到。
- **API**：两块软件之间的点菜窗口，约好规矩就能协作。

你不用会炒菜。但从这一章起，当 AI 端出一盘东西，你至少能看出它用的是哪口锅、有没有把菜真存进仓库、传菜窗口开没开。看得懂图纸的人，才有资格说「这活儿干得对不对」。后面的章节，我们就拿着这张图，一间屋一间屋地走进去。

第五章 · 用 AI 搭出第一个能点的东西

前面两章，你把想做的东西写成了规格（第三章），也大概知道了一台计算机里都有哪些零件、它们各管什么（第四章）。现在是最爽的一步：让 AI 真的替你搭出一个**能点、能跑、能存东西**的原型。不是画在纸上的框，是浏览器里能点开、点一下有反应的真家伙。

大白话

这一章我们只干一件事：把一个念头，变成屏幕上能动的东西。你会发现，这比你想象的容易得多——只要你学会一件事：怎么跟 AI 一来一回地聊。

我们要做的小东西

贯穿全章，我们做一个特别小、但特别真实的工具：

> 记录每台设备每天停机了几次，然后画一条趋势线，让我一眼看出哪台机器最近越来越爱罢工。

就这么点。它有前面那些零件里的三样：一个**能点的界面**（填「设备号、日期、停机次数」，点保存）、一段**后端逻辑**（把你填的东西收下来）、一个**存东西的地方**（关掉浏览器数据还在）。麻雀虽小，五脏俱全。你在这个小东西上练出来的手感，做大东西是一模一样的。

核心不是「写代码」，是「转一个圈」

请把这一句刻进脑子里。你要学的不是编程，是一个循环——我叫它闭环手感，说白了就是一个转起来的圈：

1. **描述**：用大白话告诉 AI 你要什么。
2. **生成**：AI 吐出一堆代码，还告诉你放哪、怎么跑。
3. **运行**：你照做，按下运行。

4. **报错**：多半会红一片，报错。

5. **喂回去**：把报错**原样**复制，粘回给 AI。

6. 回到第 2 步，再转一圈。

> 大白话
>
> 你不需要看懂代码。你需要的是：会描述、会运行、会把机器的抱怨原封不动地转达给 AI。你是这两个之间的**快递员**，不是翻译官——别自己瞎改，原样搬。

报错不是失败，是机器在跟你说话

这是全章最值钱的一句话，我单独拎出来讲。

很多人第一次看见满屏红字，心一凉，以为「我搞砸了」。恰恰相反。**报错是这台机器在跟你说话**，而且是它唯一会说的那种话。它在告诉你：我在第几行卡住了、卡在哪、缺了什么。这是整个过程里信息量最大的东西——比「成功了」有用一百倍，因为「成功了」什么也没教你，报错却精确地指出了下一步。

打个比方：报错就像家里跳闸。跳闸不是灾难，是保险丝在喊「这条线过载了，先别硬来」。你要做的不是骂保险丝，是看它指的是哪条线。报错信息（红字里那几行英文加一个行号）就是那根跳闸的保险丝——它甚至帮你标好了是哪一条。

你不用读懂那些英文。你的动作永远是：**把红字整段复制，粘给 AI，加一句「运行报了这个错，怎么办」**。AI 读得懂，它会告诉你哪儿缺了个东西、哪儿写错了个字。第一次你会发现——原来报错是来带路的，不是来羞辱你的。

怎么跟 AI 一来一回：通用心法

市面上帮你写代码的工具好几种：有的是聊天框，你说它写；有的直接住在写代码的软件里，能自己动手改文件。牌子会变，按钮会变，但心法不变。我给你四条，跟具体哪款没关系。

一、开局把「场景」交代清楚

别一上来就「帮我写个停机记录工具」。要像交代一个新来的实习生：

> 我完全不会写代码。请帮我做一个能在浏览器里打开的小网页：能填设备号、日期、停机次数，点保存后存下来，关掉浏览器再打开数据还在，下面画一条停机次数的趋势线。请一步步告诉我：装什么、文件放哪、怎么运行。默认我什么都不懂，命令请给全。

最后那句「默认我什么都不懂，命令请给全」是**魔法咒语**，它能省掉你后面一半的卡壳。

二、小步快跑，一次只改一点

这是新手最容易犯的错：一口气让 AI「顺便再加个登录、再加个导出、再加个报警」。别。**一次只让它改一件事**，改完立刻运行、立刻看结果，好了再提下一个。

大白话

为什么？因为一旦出错，你得知道是「哪一下」出的错。你一次改五个地方，红字来了，你根本不知道该怪谁。一次改一个，出错了原因就在眼皮底下——这跟工厂里调设备一个道理：一次只拧一个参数，才知道是哪个参数起的作用。

三、报错原样喂回，别加工

前面说过，这里再钉一遍：复制报错时别删、别改、别只截半句你觉得重要的。你觉得没用的那半句，往往正是关键。**整段搬过去**。

四、卡死了就重开一局

偶尔会遇到 AI 跟你绕圈子，改来改去还是错。这时候别死磕。新开一个对话，把你的规格重新讲一遍，让它**从头来一版**。旧对话里堆积的错误上下文，有时候反而是包袱。推倒重来在这里不丢人，是常规操作。

亲手转一圈，长什么样

为了让你有画面感，我把这个小工具的第一圈大致复述一下（细节你不用记，感受节奏就行）：

你把上面那段场景发过去。AI 回你：装一个叫某某的免费小工具、新建一个文件夹、把它给的几段内容分别存成几个文件、在命令行敲一行「运行」。你照做，敲下回车——

大概率，红了。屏幕上蹦出一行英文，比如 `command not found: node`（意思是「你还没装我要用的那个工具」），或者 `Cannot find module 'express'`（「有个零件没装上」）。你不慌，你也不用看懂这行英文——直接复制那几行红字，回一句「运行报了这个，怎么弄」。AI 说：哦，你还差一步没装，敲这行命令。你敲完，再运行——

这次浏览器蹦出来一个简陋的小页面：三个输入框，一个保存按钮。你填「3 号机、今天、停机 2 次」，点保存。下面慢慢多了一个点。你再填两天的，一条线连起来了。**那一刻，你的第一个东西活了。**

> 大白话
>
> 它丑得要命，没有登录，没有权限，数据就存在你自己电脑上的一个小文件（或小数据库）里——注意不是存在浏览器里，所以你换个浏览器、甚至换台电脑接着这份文件走，数据都还在；这跟「只记在这个浏览器里、换个浏览器就没了」是两回事。这些第六章我们再收拾。但它是**你的**，从一句话，转了几圈，长成了能点的东西。这个从 0 到 1 的手感，值一顿好饭。

第一次一定会卡，这不怪你

我得对你诚实：上面写得顺，真做起来你八成会卡。可能卡在某行命令电脑不认，可能卡在文件放错了地方，可能 AI 给的版本对不上你的电脑。**卡住是这条路的默认状态，不是你笨的证据。**

连干了十年的工程师，每天也在跟报错缠斗——区别只是他们早就不把红字当回事了，看见就顺手复制去查。你现在学的，正是这份「见红不慌」。你卡的每一个坎，几乎都有前人卡过、有现成答案，而 AI 就是那个替你把答案找回来的人。你要做的，只是一遍遍地转那个圈：描述、运行、报错、喂回、再来。

把这一章收进口袋

- 目标不是学会写代码，是练出**闭环手感**：描述→生成→运行→报错→原样喂回→再生成。
- **报错是机器在跟你说话**，是信息量最大的东西，不是失败。你的动作永远是「整段复制、粘回给 AI」。
- **小步快跑**，一次只改一件事，改完就运行——出错了才知道怪谁。

- 开局交代场景，末尾加一句「默认我什么都不懂，命令给全」。绕死了就重开一局。
- 第一次卡住很正常，那是这条路本来的样子。

这一章，是**顺风局**。你在自己电脑上，做一个只给自己用的小玩意，AI 几乎有求必应。享受它——因为翻过这一页，风就要转向了。下一章，我们把这个小工具搬进车间，你会立刻发现：工业设备，根本不是一个能点的网页那么简单。真正硬的部分，从第六章开始。

第六章 · 工业设备不是网站

前面五章你一路顺风：把想法写成规格，补了点计算机常识，然后用 AI 搭出了一个能点、能跑、能演示的原型。你大概已经有种错觉——**剩下的无非是把这个 demo 做大一点**。

这一章我要把这个错觉拆掉。不是泼冷水，是让你在掉进坑之前先看见坑在哪。因为工业设备和你熟悉的那种「网站 / App」，不是同一类东西的大小之分，而是**两个物种**。它们在三件根本的事情上，规则完全不一样。

> 大白话
>
> 一句话预告：**demo 跑通，不等于产线跑住。**demo 是「在我电脑上、在我盯着的时候，它对了一次」；产线跑住是「在没人管的车间里、连续三个月、每 0.5 秒一次、一次都不许出大错」。这两件事之间隔着一条鸿沟，这一章讲清楚这条鸿沟由什么构成。

一、网站活在网线里，设备活在物理世界里

你做过的所有软件，本质上都在一个干净、听话的世界里跑：数据从数据库来，从接口来，是别人已经整理好、喂到你嘴边的数字。你几乎从不需要问「这个数字是真的吗」。

工业设备不是。它的数据来自**真实世界的传感器**——一个测温度的探头、一个测振动的加速度计、一个拍零件的工业相机。而真实世界，是脏的。

- 相机镜头上会落一层粉尘，本来 99% 准的检测，脏了三天变成 80%——图像没变，世界变了。
- 车间夏天 45℃，传感器读数会漂移；冬天冷启动，前十分钟的数据全都偏。
- 旁边一台大功率电机一启动，电磁干扰让你的信号线上凭空冒出一堆毛刺，像有人往你的数据里撒了盐。
- 设备会老化。今天标定好的参数，跑半年，机械磨损了、皮带松了，同样的动作出来的数据慢慢就不是原来那个样子了。

还有一个绕不过去的角色，你迟早要跟它打交道——PLC。

大白话

PLC（可编程逻辑控制器），你就把它想成**车间里那台专门指挥机器干活的工业电脑**。你手机上装的是微信、抖音；PLC 上装的是「气缸伸出→夹紧→机械臂转 90 度→松开」这种一步一步的动作指令。工厂里几乎每一台会动的设备背后，都有一个 PLC 在发号施令。它皮实、简单、几十年不宕机，但它只懂自己那套工业语言，你想让它告诉你「现在第几号工位、传感器读数多少」，或者反过来让它听你的算法指挥，得专门去跟它「对话」。这件事有多难，第七章整章都在讲。

所以第一个根本不同是：**网站的数据是别人喂给你的、干净的；设备的数据是你自己从物理世界里刨出来的、脏的、会变的。**你的 demo 之所以好用，很可能只是因为你喂给它的那几张测试图，恰好是最干净、光照最好的那几张。

二、网站崩了刷新就好，产线崩了要死人

这是第二个、也是最需要你从骨子里改掉的习惯。

做软件的人对「出错」有一种天然的松弛：崩了？刷新一下。数据错了？改完重新提交。服务器挂了？重启，用户顶多骂两句。**软件世界里，错误几乎都是可以撤销、可以重来的。**

物理世界不能撤销。

- 你的算法误判了一次，机械臂多走了 5 毫米——一个价值上千的工件报废了，而且是**已经报废了**，没有 Ctrl+Z。
- 你的程序卡了两秒没响应——整条线停在那里，上下游几十道工序全部堵住，一分钟停产的损失可能是几万块。
- 最严重的：一个本该在人靠近时立刻停机的判断没执行，旁边站着一个工人。

大白话

请把这句话刻下来：**在工业现场，安全永远是第一位的，比功能、比准确率、比工期都高。**这不是口号，是设计原则。它意味着你做每一个判断时，都要先问一句「万一它错了，最坏会发生什么」，而不是「它多久能对」。

这带来一个反直觉的结论：**一个会「安全地失败」的系统，比一个「大部分时候很准但偶尔疯一次」的系统，靠谱得多。**宁可它拿不准的时候老老实实停下来报警、让人来看，也不能让它自信地做一个可能造成事故的动作。你在网站上追求的是「尽量别出错」；在产线上追求的是「出错的时候，错得安全」。这两种心态，做出来的东西完全不同。

后面第十二章讲「让它跑住」，一大半篇幅其实都在讲这件事：怎么让系统在硬件抽风、数据缺失、网络断了的时候，退到一个安全的状态，而不是随便乱来。

三、网站要「快」，设备要「准时」

第三个不同最微妙，很多人一辈子没绕明白：实时性。

你可能觉得实时就是「快」。不是。

> 大白话
>
> **实时不是「平均很快」，而是「保证在规定时间内，一定给出响应」。**快，是偶尔慢一下没关系，平均值好看就行；准时，是**一次都不许超时**——超一次就出事。这俩是两码事。

打个比方。你刷网页，平时 0.1 秒打开，偶尔卡成 3 秒——你皱个眉，也就过去了。对网站来说，「99% 的请求很快」是一份漂亮的成绩单。

但工业设备的世界是这样的：一个零件在传送带上以固定速度经过相机，**它只在这个位置停留 50 毫秒**。你的系统必须在这 50 毫秒内拍照、判断、并且告诉后面的气缸「这个是次品，吹掉它」。

- 你 99% 的时候只用 20 毫秒，很好。
- 但有那么 1%，因为你的程序正好在清理内存、或者被别的任务抢了资源，这一次用了 120 毫秒。
- 零件早就过去了。次品混进了良品堆里；或者更糟，气缸晚动作 100 毫秒，一巴掌拍在了下一个好零件上，甚至撞上了机械臂。

看明白了吗？**在实时系统里，「平均很快」是没有意义的，「最坏情况有多慢」才是唯一重要的数字。**晚 100 毫秒，在网站上是一次卡顿，在产线上就是一个废件、一次撞机。

而这恰恰是普通电脑、普通操作系统不擅长的事（跟你用什么编程语言关系不大——真正偷停你的是操作系统）。你的笔记本能流畅剪视频，是因为它擅长「平均快」——它会在你看不见的地方偷偷停下来收拾垃圾、更新后台、临时去忙别的进程。这些「偷偷停一下」，在剪视频时你根本察觉不到，在产线上却是致命的。**让机器做到「每一次都准时」，需要专门的硬件和专门的做法**，这就是第十一章「部署到边缘」要解决的核心问题之一。

把三条鸿沟连起来看

现在回头看你那个跑通的 demo。它之所以顺利，是因为它同时享受了三份「豁免」：

1. **数据是干净的**——你喂的是精挑细选的测试样本，没有粉尘、没有电磁干扰、没有老化。
2. **出错是免费的**——它判断错了，你笑一笑改一改，没有工件报废、没有人受伤。
3. **时间是宽松的**——它慢个一两秒，你等得起，没有零件在传送带上等不及。

> 大白话
>
> 产线，把这三份豁免全部收回。**这就是「demo 跑通」和「产线跑住」之间那道鸿沟的全部内容**：脏的物理世界、不可撤销的后果、不许超时的节拍。你后面遇到的几乎每一个「为什么这么麻烦」，都能追溯到这三条里的某一条。

我不是要吓你。恰恰相反——**知道难在哪，本身就是最大的省事。**大多数人不是被难度打败的，是被「意外」打败的：以为翻过一座小土坡，结果脚下是悬崖。你现在提前看清了这道鸿沟由哪三块石头砌成，接下来的章节，就是一块一块教你怎么过：第七章教你怎么把脏的、真实的数据从传感器和 PLC 里正确地读进来（物理约束）；第十一、十二章教你怎么让它在没人管的车间里，安全地、准时地、连续几个月不出岔子（后果与实时）。

从这一章开始，你不再是在搭一个能点的东西。你在造一台**要对真实世界负责**的机器。心态换过来，路就对了。

第七章 · 让机器开口：数据采集与设备通信

先让机器开口，再谈让机器聪明

我们花了六章聊「怎么把想法变成能跑的东西」。但工业智能装备有个前提，网站和 App 都没有：**你得先从一堆铁疙瘩里，把数据弄出来。**

算法再神、模型再大，喂给它的都是数据。没有数据，AI 就是一个饿着肚子的天才，什么也算不出来。而拿到数据这件事，恰恰是不写代码的产品经理最陌生、也最容易被人忽悠的一环。这一章，我们就把「和硬件对话」这件神秘的事，拆成你听得懂的话。

大白话

一句话预告：这一章你会得到一张「数据从哪来、经过谁、最后到你手里」的地图，还有一个可能会颠覆你认知的结论——现场数据天生是脏的，不是别人没弄好，是物理世界本来就这样。

数据的第一站：传感器

传感器，就是机器的「感觉器官」。你的皮肤能感觉到烫，是因为皮肤把「温度高」这个物理现象，变成了神经信号送进大脑。传感器干的是一模一样的事：它把温度、振动、电流、压力这些看不见摸不着的**物理量**，翻译成电信号，再变成一个数字。

比如一个测温传感器，贴在电机上。电机 60 度，它就吐出一个数——可能是 60，也可能是 600（放大了 10 倍存的），也可能是一串电压值需要你换算。记住这个「可能」，后面要考。

大白话

类比：传感器像个只会说方言的乡下亲戚。他确实知道现场情况，但他说出来的话（原始信号），你得有人帮你翻译成普通话（有意义的数字）。

数据的中转站：PLC

传感器测到的数，很少直接跑到你的电脑里。产线上还有个「工头」，叫 PLC（可编程逻辑控制器）。你就把它理解成一台**专门指挥机器干活的工业电脑**——传送带什么时候转、气缸什么时候推、报警灯什么时候亮，都是它在发号施令。

因为它管着这些机器，所以现场的很多数据，都先汇总到它这里。你想拿温度、想拿电机转速，八成得去问 PLC 要，而不是直接问传感器。

寄存器：PLC 里的一排小格子

PLC 怎么存这些数？靠 寄存器。你可以把 PLC 的内存想象成一面墙，墙上有几千个编了号的小格子。第 40001 号格子存温度，第 40002 号格子存转速，第 40010 号格子存「设备是否报警」……每个格子存一个数。

关键来了：**格子上没写标签。**你光知道有一面墙没用，你必须知道「第 40001 号格子存的是温度，单位是 0.1 度」，才能正确读到、并算出真实值。这张「几号格子对应什么」的对照表，行话叫 点表（或地址表）。

> 大白话
>
> 这里是全章最容易踩坑的地方，请划重点：点表不在任何代码里，不在任何 AI 的脑子里。它在设备电气工程师的手上，或者在一份 Excel 里。你不去问、不去要，谁也变不出来。第 40005 号格子到底是「电流」还是「电压」，AI 猜不到，只有现场的人知道。

它们怎么说话：Modbus 和 OPC UA

你想读 PLC 的格子，得按人家的规矩来。设备之间说话，有约定好的「语言规矩」，叫**协议**。工业现场最常遇到两种。

Modbus：又老又简单，1979 年就有了，像一门古老的通用方言。它的规矩糙但直接——「请给我第 40001 号格子的值」，PLC 回一个数，完事。缺点是它只管传数字，不告诉你这数字是什么意思、什么单位，全靠你手里那张点表。

OPC UA：新一代的规范普通话。它不光传数字，还能自带说明书——「这个值叫『1 号电机温度』，单位摄氏度，此刻 60.3」。信息更全、更规范，也更安全，但相应地也更重、更

复杂。

> 大白话
>
> 类比成两种打电话的规矩。Modbus 像老式对讲机：按下说话、松开听，内容全是暗号，得两边事先对好「1 代表开、0 代表关」。OPC UA 像现代视频通话：不但能通话，还自动显示对方名字、头像、在线状态。老设备多用前者，新设备、大项目倾向后者。

翻译官和二传手：边缘网关

现在问题来了。一条产线上，A 设备说 Modbus，B 设备说 OPC UA，C 设备是另一个厂家的私有协议。你的程序要是挨个去对接，会被这些五花八门的规矩逼疯。

于是有了 边缘网关——一个放在设备旁边的小盒子（「边缘」就是指靠近设备的现场那一端，相对于远处的云端）。它干三件事：**把各种协议的数据都收上来、翻译成统一格式、再往上送**给你的程序或云端。

> 大白话
>
> 类比：边缘网关像个能听懂各地方言、又会说普通话的现场翻译。设备们叽叽喳喳各说各话，网关统一听、统一翻，最后用一种你能听懂的话打包递给你。有了它，你的程序只用学一种语言。

一张完整的心智图

把上面串起来，数据从现场到你手里，走的是这么一条路：

1. **传感器**：把物理量（温度、振动）变成信号和数字。
2. **PLC**：把这些数存进一个个编号的**寄存器**格子里。
3. **协议**（Modbus / OPC UA）：约定好用什么规矩去读这些格子。
4. **边缘网关**：把不同协议的数据收齐、翻译、打包。
5. **你的程序**：终于拿到了一份能算的数据。

记住这条链子。后面每一章讲算法、讲检测，源头都是它。

全章最重要的一句：现场数据天生是脏的

如果你之前的经验来自互联网产品，你会默认「数据库里的数据是干净整齐的」。工业现场彻底颠覆这个假设。**从物理世界采来的数据，天生就是脏的。**不是谁偷懒，是物理世界本来就不干净。具体脏在哪：

- **缺值**：网线抖一下、网关重启一下，这一秒的数据就没了。一天下来一堆窟窿。
- **跳变**：温度好端端 60 度，突然冒出个 6000 度再回来。不是电机成了太阳，是信号受了干扰。
- **时间戳对不齐**：温度每秒采一次，振动每毫秒采一次，两个设备的表还差了 3 秒。你想把它们对齐分析，先得对表。
- **单位不统一**：这个传感器给的是摄氏度，那个给的是华氏度；这个存的是真实值，那个放大了 10 倍。
- **传感器漂移**：传感器用久了会「跑偏」，明明 60 度，它长期报 63 度，而且这个偏差还慢慢变大。

为什么反复强调这个？因为它决定了一件事，我们在算法章会一次次回到它：**数据质量比模型更重要。**喂给再聪明的模型一堆脏数据，得到的也只是「精致的胡说八道」。工业智能装备里，八成的坑不在算法，在数据。

> 大白话
>
> 类比：物理世界像一个总在下雨刮风的露天菜市场，不是无菌实验室。你采回来的菜（数据）沾着泥、带着虫眼、缺斤短两，这很正常。做菜前先择菜、洗菜（清洗数据），是绕不过去的活。

AI 在这一步，帮什么、不帮什么

这是本书的老规矩，每章都要划清 AI 的边界。

AI 能帮你的

- **写通信代码**：「帮我写一段 Python，用 Modbus 读 PLC 第 40001 号寄存器」——这种活 AI 很擅长，它见过成千上万个类似例子。

- **解析协议**：你把一段读回来的原始数据（一串看不懂的字节）丢给它，让它解释每一位是什么意思，它能帮你拆。
- **清洗数据**：把上面那些脏——缺值、跳变、对齐——描述给它，让它写处理代码，它能给出一套像样的方案。

AI 帮不了你的

- **它不知道你现场第几号寄存器存的是什么。**这是全章的题眼。点表在电气工程师手里，AI 没有透视眼，它只能等你把点表告诉它。你不问、它就只能猜，而猜错的代价可能是把「电压」当成「温度」去分析，整个项目从根上就歪了。
- 它不知道你现场那个传感器有没有漂移、漂了多少——那要靠现场标定，靠人拿标准仪器去对。
- 它不知道你两台设备的时钟差几秒——那要你去现场量。

> 大白话
>
> 所以这一步你真正的工作，不是写代码（那交给 AI），而是**去要那张点表、去问清楚每个数的单位和含义、去搞明白数据是怎么采上来的**。带着这些「现场真相」回来，AI 才能帮你把剩下的活干利索。你负责问对问题，AI 负责写对代码。

这一章你要带走的

- 数据通路是一条链：**传感器 → PLC（寄存器）→ 协议 → 边缘网关 → 你的程序**，缺一环都拿不到数据。
- PLC 的寄存器是编号的格子，你必须有**点表**才知道哪个格子是什么——这东西 AI 变不出来，只能去现场要。
- Modbus 老而简单、OPC UA 新而规范，是设备说话的两种「规矩」；边缘网关是那个把方言翻成普通话的中间人。
- 记住那句最重要的：**现场数据天生是脏的，数据质量比模型更重要。**下一章讲算法，我们就从「祛魅」开始，先把「AI 有多神」这个滤镜摘掉。

第八章 · 算法祛魅：它其实是在找一个函数

算法这个词，被你想复杂了

你大概听过太多次「算法」这个词，配着一堆吓人的符号、希腊字母、还有「深度」「神经」这类唬人的形容词。于是你默认：这是数学家的地盘，跟一个不懂代码的人没关系。

这一章我要拆掉这个误会。我不会推一个公式，但我保证：读完之后，你能拿常识判断别人给你的模型「靠不靠谱」。这比会推公式有用得多——公司里会推公式的人不缺，缺的是能一眼看出「这轮子不圆」的人。

先给你一个贯穿全章的类比，记住它，后面所有概念都挂在这上面：

> 我在纸上点了一堆点，不告诉你它们背后的规律。你的任务是描一条线，尽量穿过这些点。以后我随便说一个横坐标，你看着你描的线，就能报出纵坐标。

整个机器学习，本质上就是这件事。剩下的全是细节。

所谓模型，就是从数据里找一个函数

算法 / 模型：说白了，就是一个从「输入」到「输出」的转换规则。你给它一个输入，它吐给你一个输出。判断螺丝松没松：输入是这颗螺丝的拧紧扭矩，输出是「松」或「紧」。这个「扭矩 → 松/紧」的规则，就是模型。

关键在于:**这个规则不是人写死的，是机器从例子里自己找出来的。**

传统写代码，是你把规则想清楚，一条条 `if` 敲进去：扭矩小于 8 就是松。可现实里规律没这么干脆——同一颗螺丝，材质、温度、垫片厚度都会让那个界限飘。你要写的 `if` 越来越多，最后自己都绕晕。

机器学习换了个思路：你别去想规则了，你只管把大量「输入配上正确答案」的例子喂给它，让它自己去凑那条规律。这就是开头那个描线的游戏——你不告诉它函数，你只给它一

堆点。

大白话

你在中学学过函数：y = 2x + 1，给个 x 算个 y。机器学习就是反过来——不给你公式，只给你一堆 (x, y) 的对子，让机器把那个公式猜出来。所以你以后听到「模型」，脑子里就换成「一个被猜出来的函数」，八九不离十。

训练：反复拧旋钮，直到对准

训练：就是机器「凑那条线」的过程。它先瞎画一条，看看离你给的点差多远，然后往差得少的方向挪一点，再看，再挪……如此反复成千上万次，直到那条线尽量贴合所有点。

想象你在拧老式收音机的旋钮找电台：一开始全是沙沙的杂音，你慢慢拧，声音越来越清楚，直到最清楚那一刻你停手。训练就是这样，只不过要拧的旋钮成千上万个，机器自己拧，你不用管。

你作为 PM 要知道的只有一件事：**训练需要「带正确答案」的例子**。想让模型认得出缺陷，你得先有一堆图片，每张都标好「这是好的」「这是坏的」。这些标注从哪来、准不准、够不够——这才是你该操心的，不是旋钮怎么拧。第 9、10 章会反复回到这件事上。

特征：你喂给它的是不是「有用的线索」

特征：你拿来判断的那些线索。判断螺丝松没松，你可以看它的图像像素，也可以看拧紧时的扭矩曲线，还可以听拧的时候有没有异响。每一样都是一个特征。

这里有个反直觉、但极其重要的结论：

> 选对线索，比换一个更高级的模型有用得多。

打个比方。你想判断西瓜甜不甜。如果给你的线索是「西瓜的重量」，那你再聪明也判断不准——重量跟甜度关系不大。可如果给你的线索是「敲一下的声音」和「藤蔓的干枯程

度」，一个普通人都能判断个八九不离十。**线索本身没带信息，再牛的模型也变不出信息来。**

所以当有人跟你说「模型效果不好，我们换个更深的网络试试」，你要会追问一句：**我们喂给它的线索，本身够不够判断这件事？** 很多时候，答案是不够——螺丝松紧的关键信息在扭矩里，你却只喂了张模糊的照片。换十个模型也没用。这是第 9、10 章里最常见的坑，先在这里给你打个预防针。

过拟合：只会做原题的学生

现在讲全章最重要的一个词，也是你未来听汇报时最该警惕的一个词。

过拟合：模型把训练用的那批例子背得太死，连里面的噪声、巧合都当成规律背了下来。结果它在见过的题上满分，一换新题就傻眼。

回到描线的游戏。我给你 20 个点，其中有几个因为我手抖，点歪了。一个「刚好」的做法是描一条平滑的线，大方向对，个别歪点就让它歪着。可你要是较真，非要让线穿过每一个点，就得画出一条扭来扭去、上蹿下跳的鬼画符。这条鬼画符在这 20 个点上「一分不差」，可我再给你第 21 个点，它离谱得没边。

那条扭曲的线，就是过拟合。它不是在学规律，是在死记硬背。

> 大白话
>
> 换个人话:过拟合就是「只会做原题、不会变通的学生」。平时刷过的题门门满分，一到考场遇上新题型就抓瞎。你身边一定有这种同学。模型也会这样，而且比人更容易这样。

为什么说它是头号敌人？因为它**会伪装成成功**。团队拿训练数据一测，指标漂亮得不得了，皆大欢喜。可这漂亮全是背出来的，一上真实产线就崩。等你发现，钱和时间都花出去了。第 14 章「AI 骗你时你怎么知道」，源头就在这里。

怎么防？把考卷藏起来一部分

办法朴素得让你意外:把你的例子分成两堆。一堆给机器训练（平时做的练习册），另一堆**藏起来，训练时绝对不给它看**（期末考卷）。训练完，再拿这堆没见过的去考它。

在练习册上得高分不算数——那可能是背的。只有在藏起来的考卷上也得高分，才说明它是真学会了。你作为 PM，验收时一定要问一句:**这个成绩，是在模型见过的数据上测的，还是没见过的?** 这一句话，能戳穿一大半虚假的漂亮。

泛化：能在没见过的数据上管用，才是目的

泛化：模型在从没见过的新数据上，也一样管用的能力。这才是我们做模型的真正目的——你要它上产线判断的，永远是它没见过的、下一颗螺丝、下一块工件。

过拟合和泛化是一对冤家:背得越死（过拟合越重），换到新数据上越废（泛化越差）。你追求的从来不是「在旧数据上多完美」，而是「在新数据上多可靠」。记住这句，你就抓住了机器学习的牛鼻子。

「准确率 99%」可能是在骗你

最后这一节，为整个工业场景埋一根最粗的线。请务必读进去。

假设产线上 1000 个零件，只有 10 个是坏的。现在有个模型，号称**准确率 99%**。听起来棒极了对吧?

我告诉你一个作弊的模型，它也能拿到 99%:**不管来什么，一律说「好」。** 1000 个里 990 个真的好，它全说对了，990÷1000 = 99%。可它一个坏的都没抓到——而抓坏的，恰恰是你装这套设备的唯一理由。

> 当好坏比例悬殊时，「准确率」这个词几乎没有意义。它会把一个彻底没用的模型，包装成一个优等生。

所以工业上真正要看的是另外两个数，它们盯着「坏的那 10 个」:

- 查全率（召回，英文 recall）:真正坏的那 10 个里，模型抓出来几个？抓出 8 个，查全率就是 80%。**漏网了几个坏的，看这个。**（你去查资料时它常被写成「召回率」或 recall，是同一个东西。）
- 查准率（英文 precision）:模型报警说「坏」的那些里，有几个是真坏？报警 20 次、真坏只有 8 个，查准率就是 40%。**误伤了几个好的，看这个。**

这俩通常是跷跷板，按下一头翘起另一头。你把模型调得草木皆兵、一点风吹草动就报警，坏的基本都能抓住（查全率高），但会冤枉一堆好件（查准率低）。反过来，你让它谨慎点、只在铁证如山时才报警，冤枉的少了（查准率高），可也会放跑一些坏的（查全率低）。

没有哪个更好，**只有哪个更适合你的场景**。这正是你 PM 该拍板、也只有你能拍板的地方：

- 做**刹车片、心脏支架**：一个坏的流到客户手里就是人命。宁可错杀一千，不可放过一个——查全率优先，误检多几个、多几双人工复检的手，认了。
- 做**包装外观筛选**:漏掉一两个瑕疵品无非退货，可你要是天天误报、让工人满车间捡「假坏件」，产线直接瘫。这时查准率更金贵。

看出来了吗？**漏检和误检哪个更要命，是个业务问题，不是技术问题。** 算法工程师答不了，只有懂业务的你能答。这就是你在算法这张牌桌上，最硬的那张牌。

这一章，你只要带走一句话

你不需要会造轮子。但你得会判断一个轮子圆不圆——而判断靠的是常识和对数据、对业务的理解，不是数学。

具体到能马上用的，就三个问句，未来每次听模型汇报你都掏出来:

1. **喂给它的线索，本身够判断这件事吗？**（特征对不对）
2. **这个成绩，是在它没见过的数据上测的吗？**（有没有过拟合、能不能泛化）
3. **在我这个场景里，漏检和误检哪个更要命，指标是照这个来的吗？**（查全还是查准）

把这三问揣兜里，第 9 章的视觉缺陷检测、第 10 章的时序异常，你就不是被工程师牵着走，而是能站着跟他们对话了。地基打好了，我们下章上手真东西。

第九章 · 亲手做一个视觉缺陷检测

从「我觉得这有划痕」到「机器帮我盯划痕」

前面八章都在铺垫。到这一章，我们第一次真刀真枪做一个能上产线的算法。

为了讲清楚，我们全程盯着一个具体活儿：**检测一块金属件表面有没有划痕**。这活儿够典型——工厂里九成的视觉检测，本质都是它的变种：检有没有、检对不对、检歪没歪。你把这一个吃透，换成「检标签贴没贴正」「检焊点有没有虚焊」，流程一模一样，只是「什么算坏」那条标准换个说法。

我把整件事拆成五步：定问题、收数据、选方法、训练评估、部署。每一步我都会告诉你：这一步在干嘛、为什么、AI 能替你到哪、你必须自己盯住什么。

> 大白话
>
> 一句话预告本章的灵魂：**决定成败的从来不是你选了哪个模型，而是你喂给它的数据干不干净、你对「什么算划痕」的定义清不清楚。**模型是别人早就调好的现成货，数据是只有你能给的东西。

第一步：先定问题（其实是回到第三章）

很多人一上来就问「用哪个模型」。错。第一步是回到第三章的规格，把问题定死。做视觉检测，你至少要先回答四个问题：

- **什么算划痕？** 长 0.2 毫米的算不算？油污反光看起来像划痕算不算？工件本来就有的纹理算不算？这条线你划在哪，机器就学到哪。
- **漏检和误检，哪个要命？** 漏检=有划痕你说没有（放了个次品出去）；误检=没划痕你说有（把好件当废品扔了）。这两个错的代价往往天差地别——漏一个划痕流到客户手里可能索赔十万，误判一个好件顶多浪费一块钢。
- **要多快？** 产线一秒过三个件，你的检测就必须半秒出结果，再准但要三秒也白搭。
- **放在哪测？** 传送带上飞着测，还是停下来测？光是固定的还是会变？

这四个问题，**AI 一个都替你答不了**。它不知道你客户能容忍多大的划痕，不知道你这条线的节拍。这是你的领域知识，是你作为产品经理的核心价值。想清楚了写成一页纸，这页纸就是后面所有工作的宪法。

大白话

回忆第八章那个「准确率悖论」：如果一百个件里只有一个有划痕，机器闭着眼睛全判「好」，准确率就是 99%——却漏掉了唯一该抓的那个。所以从第一步起，你关心的就不是「准确率」，而是「一百个真划痕，你抓住了几个」。

第二步：收集和标注图像——最累、最关键、最不能外包

现在要给机器准备教材了。机器学缺陷检测，靠的是看大量「这是好件、这是坏件」的例子。这就引出全章最重的一个词。

标注=人工在图片上标明哪是好、哪是坏、划痕在哪。说白了就是给机器出练习册，而且每道题的标准答案都得你亲手写。机器有多聪明，取决于这本练习册出得有多准。

你得去产线上，真的拍。拍几百张有划痕的、几百张没划痕的。而且——这一点极其重要——要拍**各种各样**的：不同批次的料、不同角度、车间早上和下午不同的光、划痕在边上的和在正中间的、深的浅的长的短的。

为什么这步最累又最不能外包？因为标注的人必须真懂什么算划痕。你雇个大学生标一天，他不懂工艺，把反光当划痕、把真划痕当油污放过去，这本练习册就全是错题。**机器会忠实地学会你所有的错误。**它不会纠正你，只会放大你。

> 行业里有句实话：算法工程师八成的时间不是在调模型，是在跟数据死磕、跟标注错误死磕。这不是他们水平差，这是这行的本来面目。

AI 能帮你到哪？它能帮你搭一套标注的流程和小工具（比如一个网页，让你在图上拉个框就存下坐标），能帮你把标好的数据整理成机器要的格式，甚至能先粗标一遍让你来改。但**「这一张到底算不算划痕」这个判断，AI 给不了你**——它自己都还没学会，正等着你教。这是你和机器之间不可转让的分工。

第三步：选方法——别从零训练，站在巨人肩膀上

到了选模型这步，小白最容易被劝退，因为一搜全是吓人的术语。我给你一条最省力、也最主流的路，两个词就能讲明白。

预训练模型=别人已经在几百万张各种图片上训练好的一个「通用视觉底子」。它早就学会了什么是边缘、什么是纹理、什么是形状、什么是「不一样的地方」——这些通用的看图能力，检划痕也用得上。

迁移学习=你不从一张白纸开始教，而是拿这个现成的底子，用你那几百张金属件图片再「补习」一下，让它把通用的看图能力专门用到「认划痕」上。

> 大白话
>
> 打个比方：从零训练，像是要培养一个验光师，你得先教他从睁眼看世界学起。迁移学习，是请一个视力极好、什么都见过的成年人，你只花两天教他「这种细纹叫划痕、要挑出来」。前者要几百万张图、几周算力；后者几百张图、一下午就行。**「几百张也能work」的秘密，全在这里。**

所以对你来说，选方法不是去比十个模型谁强，而是选一个成熟的预训练模型做迁移学习就够了。哪个具体模型？让 AI 推荐，它会根据你的场景（要多快、跑在什么设备上）给你选。这一步 AI 帮得上大忙——写训练代码、配环境、挑模型，都是它的活。你只要盯住一件事：**别被忽悠着去从零训练**，那是烧钱烧时间还不一定更好的弯路。

第四步：训练 + 评估——盯住机器错在哪

训练=让机器一遍遍看你的练习册，自己调整内部参数，直到它的判断尽量对上你的标注。这步基本是 AI 写代码、机器自己跑，你等着就行。

关键在评估。别只看它甩给你一个「准确率 98%」就高兴。回到第八章的两把尺子：

- 查全率（召回）=一百个真划痕，机器抓住了几个。漏检要命的场景，这个是命根子。
- 查准率=机器报警说「有划痕」的里头，有几个是真的。误检浪费大的场景，看这个。

然后做一件最有价值的事：**把机器判错的样本一张张调出来看**。它把哪些真划痕漏了？把哪些好件误报了？看着看着你往往会恍然大悟——「哦，它一碰到边缘反光就误报」「哦，浅

划痕它基本都漏」。这些发现会直接告诉你：练习册里这类样本太少了，回第二步补拍补标。**训练不是一锤子买卖，是「看错题→补数据→再练」的循环。**

第五步：部署到相机边上（这里只开个头）

模型练好了，最后要把它搬到产线相机旁边那台小设备上，实时看每一个工件。这一步细节留给第十一章。这里你只要记住一件事：**训练是在你电脑上、慢慢来；部署是在车间里、一秒好几个件飞过去实时判。**环境全变了，能跑通不等于能跑住——第十二章专门讲这个。

本章灵魂：为什么「准确率 99%」在产线上可能还是不能用

这一步最容易让人高估自己：demo 里数字漂亮，一上真产线就翻车。为什么？

第一，**类别极不均衡**。产线上真划痕是少数，前面说过，机器全判「好」都能拿高准确率。所以准确率这个数字，在缺陷检测里几乎是骗人的，你要盯查全率。

第二，**漏一个致命缺陷的代价**。查全率讲的是「真次品里你抓住了几成」——假设一百万件里真有一万个划痕件，就算你查全率高到 99%，也还是放走了大约一百个次品。这一百个够不够呛，你的客户能不能接受，是业务问题，不是技术问题——又回到你身上了。

第三，也是最阴险的一个，叫数据漂移=你真正要检的工件，跟你当初拍照片、教机器时的工件，悄悄变得不一样了。换了个供应商的料，表面反光变了；车间换了灯，或者只是从夏天到冬天光照角度变了；镜头蒙了层油雾。机器还是按老经验判，就开始胡说八道。

> demo 翻车九成不是模型笨，是你给它看的世界和它现在面对的世界，已经不是同一个了。

大白话

所以「上线」不是终点，是起点。你得定期抽查、定期拿新工件的照片补进练习册再训一次。这不是麻烦，这是这类系统活着的方式——它跟你的产线一起变老，就得跟着一起学。第十三章会专门讲怎么盯住它、怎么迭代。

把这一章拧成一句话

你现在有能力借 AI 从零做出一个视觉缺陷检测：AI 替你写代码、搭标注工具、选模型、调参数、跑训练——这些体力活它全包了。但有三件事永远是你的，谁也替不了：**定义什么算缺陷、判断每张图的对错、决定这个准确率业务上到底能不能忍。**

说到底，模型是买来的现成引擎，数据和标准是只有你能造的燃料和方向盘。下一章我们换个战场：不看图片，看一条随时间起伏的曲线，做时序和异常检测。

第十章 · 时序预测与异常检测：让设备提前喊疼

第九章我们教机器"看"——一张图里有没有划痕。这一章换个感官：教机器"听"和"感觉"。让设备在真正坏掉之前，先哼哼两声："我这儿有点不对劲。"

做这件事的原料，是时序数据（time series，按时间一个接一个记下来的数值，像心电图那样一条线往前走）。电机的振动、轴承的温度、主轴的电流——每隔一秒、甚至每秒上千次，采一个数，存下来，就是一条时序。我们要从这条线里，提前看出故障的苗头。这件事有个正经名字叫预测性维护（predictive maintenance，在设备坏之前就发现苗头、提前修，而不是坏了才停机）。

时序和图像，为什么是两码事

第九章的图像问题，本质是"这一张里，有没有"。你把一堆照片打乱顺序喂给机器，不影响它学认划痕——每张照片是独立的一个个体。

时序不是。时序的**顺序本身就是信息**。振动值 3、5、3、5、3、5 是设备在正常地抖；同样这几个数字换个排法，3、3、3、5、5、5，讲的是完全不同的故事——前者平稳，后者是突然爬升。你要是把一段振动波形打乱重排，它就彻底没意义了，等于把心电图剪碎了重新粘。

> 大白话
>
> 图像看的是"里面有什么东西"，一张就是一张。时序看的是"这段变化正不正常"，得连着看一整段。趋势、节奏、先后，才是它要说的话。所以第九章那套"把样本打乱、随便抽"的做法，到这儿要改。

两种活儿：猜下一步，和挑出不对劲

时序上你想干的事，基本就两类，分清楚很重要，因为它们的做法不一样。

第一类是**预测**：照着现在的趋势，猜下一步、或者猜"还有多久会到危险线"。温度这条线一直在慢慢往上爬，你想算：照这个爬法，还有几个小时会撞到 80 度的红线？这像看着水位涨，估摸着几点会漫过堤坝。

第二类是**异常检测**：不管预测什么，就看这一段波形"跟平时长得一样不一样"。老师傅站在机器边听声音，说不清哪个零件坏了、也不预测什么，但他一皱眉："这声不对。"这就是异常检测。

最关键的一个坎：你根本没有"坏"的样本

第九章我们怎么教机器认划痕？拿一大堆有划痕的照片，一张张告诉它"这是划痕"。这叫有监督（supervised，拿标好答案的样本教它，好的坏的都给看，让它学会区分）。

到了设备这儿，这条路常常走不通。原因很朴素：**好设备大部分时间都是好的**。一台运转三年的机床，坏的时刻可能一年就那么几次，甚至这台新设备压根还没坏过。你想收集"故障样本"来教机器，结果发现故障样本稀少得可怜，甚至一个都没有。你手里全是"正常"，没有"坏"可以拿给它看。

> 大白话
>
> 这是时序异常检测和图像检测最不一样的地方，也是最容易让人卡住的地方。别指望攒够一堆故障录像。轴承坏一次可能要几万块、停一次产线可能几十万，你不可能为了"多几个坏样本"去把设备一台台搞坏。坏样本天然就稀缺——这不是数据没采够，是现实就这样。

换个聪明思路：只学"正常长什么样"

既然没法"拿一堆坏的教它认坏"，那就反过来：**拿一大堆正常的，让机器学透"正常"到底长什么样**。学熟了以后，凡是明显不像正常的，就挑出来报警。这叫无监督（unsupervised，不给一条条「这是好那是坏」的答案，只让机器自己摸清"正常"的规律，不认识的就当可疑）。（较真地说，你其实还是给了一个大标签——「这一堆全是正常的」，所以业界更严格的叫法是「单类学习」，你日后查资料看到 one-class 或 novelty detection，指的就是这套；意思一样，不必纠结名字。）

为什么这招聪明？因为它把一个"你没有的东西"（各种坏法的样本），换成了"你要多少有多少的东西"（设备天天在产的正常数据）。你不需要预先知道设备会怎么坏——它可能以你从没见过的方式坏——你只需要知道"正常是什么样"，任何偏离正常的，都会露馅。

这正是老师傅那套。他不是背了一本"故障声音大全"，他是听了十年正常的声音，正常刻进了骨头里，所以一点不对就炸毛。机器也一样：异常检测（anomaly detection，先学会正常的样子，再把"明显不像正常"的挑出来）的核心，是先把正常吃透。

具体手法有名字，但都是一句人话：最土的一招叫"给正常划一个范围，出界就报警"——正常时振动在 2 到 6 之间，冒到 9 了，报。复杂点的会连着一段一起看："正常时温度和电流是一起涨一起落的，现在温度涨电流不动，这个搭配没见过，可疑。"名字你不用记，记住那句人话就够了：**机器在心里存了一个"正常的样子"，拿眼前的和它比，差得远就喊。**

头号杀手：报警报到没人看

现在讲这类系统真正会死在哪儿。不是算得不准，是误报（false alarm，其实没事，系统却报了警——狼来了）太多。

你把灵敏度调得很高，一有风吹草动就报。头一周工人还认真跑去看，跑十次有九次是虚惊。第二周他就不看了。第三周报警成了背景噪音，跟没有一样。等真出事那次报警响起，没人理——**系统还活着，但已经死了**。

> 大白话
>
> 一个天天喊狼来了的报警器，比没有报警器更糟：它既没帮上忙，还骗走了工人的信任，以后真报了也没人信。误报是这类系统的头号杀手，比漏报更阴险，因为漏报你还知道要改进，误报是慢慢把整个系统废掉。

这就接上了第八章那个绕不开的取舍：查得全（宁可错报也别放过）还是查得准（宁可放过也别错报）。调灵敏一点，漏报少了但误报多了；调迟钝一点，误报少了但可能漏掉真故障。这中间没有标准答案，**只有你这台设备的答案**——漏一次报废十万，还是停一次错误的机耽误五万，得你来算这笔账。

有两个保命的做法，务必记住。

- **报警要能解释。**别只弹一个红灯"异常"。要说"3 号轴承振动比平时高 40%，持续 10 分钟"。工人能看懂、能判断、能去现场核对，报警才有人信。一个说不出理由的报警，等于让人凭空信你，信不了几次。
- **报警要能分级。**别所有事都拉最高警报。"轻微异常，留意"、"明显异常，本班次内查"、"严重，立即停机"——分了级，工人才知道哪个要撂下手里的活去跑，哪个记一笔就行。一刀切的报警，逼着人要么全信要么全不信，最后就是全不信。

AI 帮得上什么，帮不上什么

老规矩，划清楚这条线，你才不会把命交错地方。

AI（这里指你让它写代码的那个助手）在这一章能帮你干不少体力活：从原始振动数据里算出有用的特征（feature，从原始波形里提炼的概括量，比如这一秒抖得多厉害、抖得多快）——这块代码又碎又烦，正适合让它写；搭起那个"学正常"的模型；把时序画成图让你肉眼看趋势；甚至帮你把好几种异常检测方法都试一遍，比一比哪个在你的数据上误报少。这些都让它做，你省下大把时间。

但有几件事，AI 一句都替你答不了，因为答案不在代码里，在车间里：

- **你这台设备，多少算异常？**振动到 8 是正常波动还是要出事？AI 不知道。这得问天天守着这台机器的老师傅，或者查设备手册。
- **报警该多灵敏？**前面那笔"漏报 vs 误报"的账，是你的生产账、你的成本账，AI 没法替你算，它不知道你停一次机赔多少。
- **什么样的异常值得管？**有些"异常"是正常的——每天开机头五分钟数据都飘，那是设备在热身，不是故障。哪些飘要忽略，哪些要警觉，是经验，不是算法。

> 大白话
>
> 把这一章记成一句话：让机器把"正常"学透，凡是明显不像正常的挑出来提前喊；但"多不像才叫不正常""喊得多响才合适"，是你和老师傅的经验，不是 AI 的活。AI 提供那双灵敏的耳朵，你提供那颗判断的脑子。

做好了，设备就会在坏之前先哼哼两声。而这两声值不值得你放下手里的活跑过去——那始终是人的判断。下一章，我们把这套算法从你的电脑，搬到设备旁边那个巴掌大的盒子里，

让它真正在车间里 7×24 地听着。

第十一章 · 从我的电脑到产线上的盒子

你的笔记本，和现场那台盒子，是两个世界

前面几章，你在自己电脑上把算法练出来了、把程序跑通了。屏幕上，摄像头拍一张零件，框一闪，"有裂纹 / 没裂纹"跳出来，200 毫秒一张，漂亮。你心里踏实了：成了。

没成。你只是在自己家的厨房里，做了一道好菜。现在要把这道菜，端到工厂车间那个连煤气灶都没有的角落，让它自己做，一天做八千盘，连做三个月不歇火。

这一章讲的就是这段路——从**你的电脑**到**现场那台盒子**。这段路，是很多小白项目真正死掉的地方。不是死在算法不准，是死在"本地 demo 惊艳，一到现场就跑不动、或者跑几天就趴窝"。我见过太多人，第六章的实时性演示做得飞起，兴冲冲去现场装机，当场傻眼。

> 大白话
>
> 记住一句话：能在你笔记本上跑，跟能在现场跑，是两件完全不同的事。前者是做出来了，后者才是交付了。

为什么非得在设备旁边算

你可能会想：我笔记本跑得动，那我不搬了，把现场拍的图传回来、在我这台好电脑（或者云上一台更好的电脑）算完再把结果传回去，不行吗？

大多数工业场景，不行。这里先分清两个词。

云端，就是把计算放到远处一个巨大的机房里去算——你手机上的很多东西都是这么干的，拍张照发上去，那边处理完给你传回来。边缘计算（也叫边缘设备）反过来：不往远处传，就在设备旁边摆一台小盒子，现场就地算完，就地出结果。"边缘"的意思是，它在整张网络的最外圈、最靠近现场的地方，而不是在中心机房。

工业为什么偏要选边缘？四个很实在的原因，都不是玄学：

- **延迟等不起**。产线上零件哗哗地过，机械臂要在零件飞过去的那一瞬间决定"这个踢下去还是留着"。图传到云端、算完、再传回来，一来一回哪怕只多花半秒，零件早跑没影了。（这就是第六章说的实时性，到现场只会更紧。）
- **断网是常态**。车间不是写字楼，网络说断就断。你的产线不能因为网卡一抖就停产。靠云端，等于把命脉拴在一根随时会断的线上。
- **数据太大，传不起也传不动**。高清摄像头一秒几十张图，几路摄像头同时来，这个量往外传，网络扛不住、流量费也吓人。就地算，只把"合格/不合格"这几个字传出去，就轻松了。
- **数据不让出厂**。很多工厂的产品图纸、工艺画面是保密的，合同里白纸黑字写着不许上传外网。那就只能在厂里自己的盒子上算。

大白话

一句话总结：工业现场要的是快、是稳、是省、是保密，这四条都逼着你把计算搬到设备旁边那台盒子上，而不是甩给远方的云。

现场那台盒子，比你笔记本弱得多

现在来看这台盒子长什么样。它通常叫工控机（工业控制计算机）或者嵌入式设备——说白了就是一台为"皮实耐用、能在车间油污高温里连轴转"而生的电脑。皮实是它的优点，但性能，往往是它的短板。跟你那台买来跑算法的笔记本比，它可能：

- **脑子（CPU）慢**。同样一道题，你电脑一秒算完，它可能要磨蹭好几秒。
- **没有（或只有很小的）显卡**。你训练模型时那块又贵又猛的显卡，是它做梦都摸不着的。很多工控机压根没显卡。
- **内存小**。你电脑 16G、32G 内存随便造，它可能就 2G、4G，模型稍微大一点就装不下。
- **芯片型号可能都不一样**。很多工控机用的是 ARM 架构的芯片（跟你手机同源），而你笔记本大多是另一种架构（x86）。这个差别很要命：**你电脑上能顺利装上的软件，到它那儿可能根本装不上**——就像你家的插头插不进人家墙上的插座，孔型不对。

所以真正的坑来了：**你训练时用的那个又大又猛的模型，搬到这台小盒子上，很可能跑不动，或者慢到没法用。**你 demo 里 200 毫秒一张，到了盒子上可能变成 3 秒一张。产线一分钟过一百个零件，你 3 秒才看一个——这不叫慢，这叫瘫。

让模型变小变快，但尽量别变笨

怎么办？把模型"瘦身"。这件事有个正经名字叫模型压缩——想办法让模型体积更小、算得更快，同时尽量不让它变笨（判断得不准）。这里"尽量"两个字很关键：压缩多少都会掉一点点精度，手艺就体现在"少掉精度、多提速度"上。

压缩有很多招，你不用会做，但要听懂两个最常见的直觉，将来跟人聊、跟 AI 提需求，能对上话：

量化：把数字算得粗一点

量化，说白了就是把模型里那一大堆高精度的数字，换成低精度的、更"粗"的数字。

打个比方：你记账，本来每笔都记到小数点后八位，¥3.14159265。可你买菜，记成 ¥3.14 就够用了，甚至 ¥3 也行。位数少了，写起来快、占地方小、算起来省力，而账大差不差还是那个账。模型里成千上万个数字都这么"四舍五入"一下，整个模型立马又小又快，代价是判断精度会掉那么一丁点——通常掉得很少，值。

剪枝：把没用的枝条剪掉

剪枝，就是把模型里那些"没什么用"的连接给剪断、扔掉。

模型内部是密密麻麻的连接，就像一棵长疯了的树。仔细看，很多枝条其实不结果、白占养分。园丁修剪，把这些废枝剪掉，树反而更精神、更省力，结的果子几乎不少。剪枝就是给模型做这个修剪，剪完模型更瘦、跑得更快。

> 大白话
>
> 量化 = 数字算粗点；剪枝 = 废连接剪掉。目的都一样：更小、更快，尽量不变笨。细节不用懂，AI 会帮你做，你知道它俩是"瘦身"的两种手法就够了。

"推理"这件事，值得单独优化

这里要分清模型的两个人生阶段。训练是模型上学的阶段——你喂它成千上万张标好的图，它慢慢学会认裂纹，这一步又慢又费，但只做一次。推理是模型毕业上班的阶段——现场来一张新图，它给一个判断（有裂纹/没有），这一步天天做、每张零件都要做。

你的盒子上，只干推理这一件事，不干训练。所以我们要优化的、要它跑得飞快的，是**推理**。训练慢点无所谓，推理慢一点，产线就堵。

为了让推理在某台特定硬件上跑到最快，业界有专门的东西，通常叫推理引擎（或部署运行时）。它是干嘛的？打个比方：同一份菜谱，交给不同的厨房，得按那个厨房的灶具、锅具重新排一遍工序才最顺手。推理引擎就是这么个"翻译官"——把你的模型，翻译成"这台特定盒子跑得最快的形式"。名字你不用记，知道有这么个东西、它专治"在这台硬件上榨出最高速度"就行。

万恶之源：环境不一致

就算模型瘦了、也优化了，还有一个更隐蔽、更能气死人的坑：**环境不一致**。

你的程序能在你电脑上跑，不只是因为你写了它。是因为你电脑上，日积月累装了一大堆"配套的东西"——各种库、各种依赖、各种设置。你自己都未必记得装过哪些。而现场那台盒子，是张白纸，什么都没有。你把程序拷过去一运行，它当场报错：缺这个、缺那个、这个版本还对不上。

这就像你在自家厨房做菜行云流水，因为你的油盐酱醋、趁手的锅、开好的火，全在。换个空厨房，你啥也做不出来——不是你不会做，是家伙什儿没跟过去。

治这个病的思路，叫容器（也叫打包）。核心想法特别朴素：**把程序，连同它需要的一切东西，整个打包在一起，搬到哪儿都能原样开火。**

还是厨房的比方：与其祈祷新厨房里凑巧啥都有，不如把整间厨房——连锅带料、油盐酱醋、开着的火——整个装进一个集装箱，运到哪儿，打开就是你熟悉的那间厨房，一点不差。容器就是这么个"整箱搬运"。这样，你电脑上能跑，盒子上就能跑，因为跑的根本是同一套环境。但记住一条边界：容器抹平的是「缺了哪个库、版本对不上」这类环境差异，它**抹不平芯片架构那道坎**——x86 打的包塞不进 ARM 盒子。所以你得给盒子那种架构专门打一个包（幸好这也是让 AI 帮你做的活）。细节不展开，你只要记住这个"连锅带料整箱搬、但得按盒子的插座打包"的思路。

这段路上，AI 帮得了什么，帮不了什么

老规矩，划清 AI 的能与不能。

AI 能帮你：

- 写模型压缩、格式转换的脚本——你说"帮我把这个模型量化、转成能在某某盒子上跑的形式"，它能给你把代码写出来。
- 写打包脚本，把程序和依赖装进容器。
- 当医生：程序在盒子上跑不起来、报一堆红字，你把错误贴给它，它大概率能帮你诊断出是缺了什么、版本对不上、还是架构不匹配。

AI 帮不了你：

- 它不知道你现场那台盒子到底是什么型号、什么芯片、多大内存、有没有显卡。这些它看不见，只有你去现场量、去问、去查铭牌。
- 它不知道你的产线到底要求多快。这个数是业务定的，得你来告诉它。

大白话

换句话说：**脏活累活 AI 干，去现场"量尺寸"这件事，得你自己去。**你把盒子的真实家底摸清楚喂给它，它才能帮你把模型剪裁到刚好合身。你不去量，它就是瞎猜。

> 本地跑通，只是拿到了入场券。真正的比赛，是让它在那台又弱又倔、还得连转三个月的小盒子上，稳稳当当地干活。下一章，我们就来说说怎么让它"跑住"——不是跑起来，是跑住。

第十二章 · 让它在产线上跑住，而不只是跑通

demo 和产品之间，隔着一场地震

你在办公室里，把摄像头对准几个样品，AI 一个个都判对了。你截了屏，发到群里，老板点了个赞。那一刻你觉得，成了。

可这只是「跑通」。跑通的意思是：在一切都顺利的时候，它能干对一次。

而产线要的是「跑住」：在接下来的三百六十五天里，不管有人踢掉网线、不管半夜车间跳闸、不管上游换了个新料号导致图像变了样，它都得**要么继续干对，要么安全地停下来喊人**——而不是悄无声息地崩掉，让一整批不合格的产品混着走了。

这两者之间隔着的，不是功能，是现实。现实里，一切都会坏。

> 大白话
>
> 这一章不酷。没有惊艳的 demo，没有「哇」的时刻。但它决定了你的产品到底能不能交付，以及——你会不会半夜三点被一个电话吵醒。

工业软件的第一性原理：假设一切都会坏

普通人（包括你让 AI 写代码时的默认状态）写程序，脑子里想的是「事情顺利地发生」：传感器读到数、网络是通的、数据长得跟昨天一样。这条「一切顺利」的路径，有个行话叫 happy path（顺利路径，就是假设万事如意时程序走的那条道）。

工业软件的设计前提，恰恰相反。它叫 防御性编程：**不指望一切顺利，而是预先把每个环节坏掉的样子都想一遍，想好它坏了我怎么办。**

打个比方。业余司机开车，想的是「路是平的，前面没人」。老司机开车，眼睛一直在扫「那辆车会不会突然并线、这个路口会不会窜出人、刹车要是失灵我往哪打方向」。老司机不是悲观，他是知道：路上什么都可能发生，你得提前替这些「万一」留好后手。工业软件，要的就是老司机。

下面五个东西，就是老司机的五个后手。我一个个用大白话讲，每个都告诉你：**不做，会怎样。**

后手一 · 异常处理：程序摔倒了，得能爬起来

异常处理说白了就是：当一件本该正常的事突然不正常了，程序不能整个人当场去世，而要「接住」这个意外，记下来，然后决定是接着干还是安全停下。

现实里会发生什么意外？传感器这一秒突然读不到数（返回一个空值）；有人蹲下去理线，网线松了；上游系统升级，传过来的数据格式从「12.5」变成了「12.5mm」多了俩字母。

没有异常处理，会怎样？程序遇到一个它没见过的情况，直接崩溃退出。产线上没人盯着屏幕，机器还在往下走料，AI 已经死了半小时了没人知道——这半小时的产品，等于没人检。

> 「接住」不等于「假装没事」。接住的意思是：这一帧图像坏了，那我跳过这一件、记一笔、报个警，而不是让整条线陪着它一起死。

后手二 · 断电断网重连：拔了插头，还能接着干

车间不是机房。跳闸、临时停电、Wi-Fi 抽风，是家常便饭。所以你的程序必须假设：**我随时会被人拔电源。**

好的程序被拔了电、再上电，应该能自己重新连上摄像头、连上数据库、接着上次的地方干下去。而且要满足两个苛刻的要求：**不丢数据，不重复算。**

不丢数据——断电前那几件的检测结果，得已经安全落到硬盘上，不能只存在内存里跟着一起没了。不重复算——恢复以后，别把已经检过、已经发货的那件又重新检一遍、又报一次废。

不做会怎样？轻则每次一断电就得有个人跑过去手动重启、手动对齐进度；重则数据对不上账，这个班次到底做了多少件、废了多少件，成了一笔糊涂账。工业现场最怕糊涂账。

后手三 · 日志：给你的产品装个黑匣子

日志就是程序一边干活一边写的流水账：**几点几分，发生了什么。**「14:03:07 第 8821 件，判为合格，置信度 0.97」「14:03:09 摄像头读取超时，已跳过，已报警」。

飞机上有黑匣子，出了事靠它复盘。你的产品也需要黑匣子。因为工业现场的故障有个特点：**它往往不在你眼前发生。**凌晨四点漏检了一件，等早上有人发现，现场早就恢复正常了，你上哪儿去找当时到底哪儿出了岔子？

答案是：翻日志。没有日志，你就是抓瞎——只能靠猜，靠「重现」一个你根本不知道怎么触发的问题。有日志，你能把出事那一刻前后几秒钟，一帧一帧看清楚。

> 大白话
>
> 一个朴素但值钱的经验：日志要记「够复盘」，但别记成裹脚布。关键节点、每一次异常、每一个判断结果都要记；但别把每毫秒的废话全塞进去，否则出事时你在几百万行里捞针，等于没记。

后手四 · 看门狗：一个不睡觉的值班员

有一种坏，比崩溃更阴险：程序没退出，但**卡死了**——它还开着，界面还亮着，看上去活着，其实早就不动了，像个植物人。这种时候连崩溃日志都没有，最难发现。

看门狗（watchdog）就是专门治这个的：另起一个极简单的小程序，在旁边盯着主程序。主程序每隔几秒得主动「喊一声」报平安，看门狗一旦发现超过约定时间没听见喊声，就判定它卡死了，直接把它重启。

你可以把它想成一个不睡觉的值班员，手里攥着秒表：每隔十秒你得跟我打个招呼，超过十秒没动静，我就当你出事了，一巴掌把你拍醒（重启）。

不做会怎样？程序悄悄卡死，产线继续走料，可能几个小时都没人发现「AI 其实早不干活了」。看门狗的价值，就是把「几小时没人管的植物人」变成「几十秒后自动满血复活」。

后手五 · 失效安全：拿不准的时候，倒向安全那一侧

这是五个里最重要的一个，它接着第六章讲的「安全第一」。

失效安全（fail-safe）说的是：万一系统出了问题、拿不准了，它要**倒向安全的那一侧**，而不是稀里糊涂地放行。

什么叫安全的那一侧？在质检里，是「宁可停机报警，也不放行一个可能有缺陷的产品」。在控制机械臂里，是「一旦拿不准位置，立刻停住别动，也不许乱挥」。核心是一句话：**不确定的时候，选择「停」和「喊人」，而不是「继续」。**

为什么这条最要命？因为它跟别的失效不一样。程序崩了、卡死了，你顶多是「没检」，损失的是没抓到的次品。但一个「失效不安全」的系统，会在自己出问题的时候，还自信地告诉产线「这件是好的，放行吧」——这是把缺陷主动放进了合格品里，甚至可能让机械臂在错误的位置发力。前者是漏，后者是错，错比漏严重得多。

> 健康的默认姿态是：不确定 = 停 + 报警。绝不能是：不确定 = 放行。这一个字的差别，就是安全和事故的差别。

不会写代码的你，怎么把这些做扎实？

看到这儿你可能慌了：这五样听着都很「工程」，我不会写代码，AI 又默认只写顺利路径，那这些防御谁来做？

关键洞见来了——**你不是靠自己写，你是靠在规格里就把这些坏情况列进去，逼 AI 去处理。**

回到第三章那句话：AI 不会主动替你操心失效，它只会老老实实实现你说了的东西。你说「判断产品好坏」，它就给你写一条顺利路径。你没提「传感器读不到数怎么办」，它就当这事永远不会发生。**所以这些防御做没做，取决于你问没问。**

而「把所有可能坏的地方都想一遍」，恰恰是产品经理最值钱的看家本领——想全场景。这活儿本来就该你干，只不过以前你想完是写进需求文档给工程师，现在你想完是写进规格给AI。能力没变，对象变了。

一份可操作的「坏情况清单」思路

做法很简单：拿着你的每一个环节，逐个追问一句「**这里坏了怎么办**」，把答案写进给 AI 的规格里。像这样一条条过：

- **输入坏了：**传感器读不到数、图像全黑、数据格式变了——程序该跳过并报警，还是停机？（写清楚：跳过这一件，记日志，亮黄灯。）
- **连接断了：**断电、断网、摄像头掉线——恢复后要自动重连、接着上次进度干，不丢已存的数据、不重复处理已完成的件。
- **程序卡了：**要有看门狗，超过 N 秒无响应就自动重启，并记一条「我被重启过」的日志。
- **结果不可信：**置信度低于某个线、或出现从没见过的情况——一律走失效安全：停机 + 报警，绝不放行。
- **出了事查得到：**每一件的判断结果、每一次异常、每一次重启，都要落日志，能按时间倒回去看。

把这五问的答案，明明白白写进你给 AI 的规格里。你不需要知道 AI 用什么代码实现——就像你不需要知道老司机踩刹车的肌肉是怎么收缩的——你只需要坚持要求「这些万一，都得有后手」。

大白话

一句话记住这一章：跑通靠的是把顺利路径写对，跑住靠的是把所有不顺利都提前想到。前者让你在群里收获点赞，后者让你晚上睡得着觉。而后者，恰好是不写代码的你，最该出力、也最出得上力的地方。

第十三章 · 怎么知道它是真的对：验证、试运行与迭代

没测过的「做好了」，等于没做

你让 AI 改了一段代码，它回你一句：「已完成，功能正常。」你信吗？

问题在于：你不会写代码。它说「正常」，你没有任何独立的手段去戳穿它。它要是错了，或者只是「在它想到的那几种情况下对」，你也看不出来。你只能信。而在工业现场，信一个你验证不了的东西，是要出事的——不是你被扣绩效那种出事，是整批产品流到客户手里、或者机械臂在真人旁边做了个不该做的动作那种出事。

大白话

前面几章我们把东西做出来了（第 9、10 章）、搬到设备上了（第 11 章）、让它别一遇到怪事就崩（第 12 章）。这一章只讲一件事：**你怎么知道它是真的对**。不是「看起来对」，是拿得出证据的那种对。

先说个不好听的实话：验证这件事一点都不酷。它耗时间，没有 demo 那一刻的高光，还经常被跳过——因为跳过它，在演示当天没人看得出来。但它恰恰是「demo 侠」和「真能交付的人」之间那条线。demo 侠交付的是「那天那台机器上，那几个样品，跑通了」；能交付的人交付的是「这条线接下来三个月，见到什么它都扛得住」。

第一件事：让机器替你反复核对

测试，说白了就是把「喂它这个，它应该吐出那个」这句话，写成一段机器能自己跑的核对程序。你规定：这张有划痕的图，判定必须是「不合格」；这张干净的图，判定必须是「合格」。写好之后，机器每次都能自动把这一串都核对一遍，对不对当场给你亮红绿灯。

好消息是：这段核对程序，你不用自己写，AI 写得又快又好。你负责的是另一件 AI 干不了的事——**指定该测哪些情况**。这才是你的活。AI 会本能地测那些「正常的、漂亮的」情况，

因为那些最好想。真正会咬人的，是你在第 3 章定规格、第 12 章想「万一坏了怎么办」时列过的那些边界和坏情况：

- 划痕正好压在零件边缘、一半在一半不在，算不算缺陷？
- 图片是全黑的（镜头被挡了）、或者曝光过度一片白，它会不会硬判一个结果出来？
- 传送带上同时来了两个零件，它数得清吗？
- 一个从没见过的、图纸上根本没有的新缺陷类型来了，它老实说「我不确定」，还是瞎猜一个？

你把这些情况用大白话列给 AI，让它把每一条都变成一个自动核对项。列得越刁钻，这张网越密。

改一处，别弄坏十处

还有个更省心的好处，叫回归测试——「回归」意思是「你别给我倒退回去」。软件有个反直觉的坑：你改 A 处，可能悄悄把八竿子打不着的 B 处弄坏了，而且当场一点声都不出。等你三周后发现 B 坏了，早忘了是那次改 A 干的。

> 大白话
>
> 把之前列的核对项攒成一个清单，每次改完东西，一键让机器从头到尾跑一遍。哪条从绿变红，你立刻就知道「刚才那一改，把这里搞坏了」。这就是不会写代码的人手里最实在的一根保险绳：你改不动代码，但你能一眼看出改动有没有闯祸。

但真正的考场，不在你电脑上

这是本章最想让你记住的一句话。你电脑上、你办公室那台样机上的所有测试，都只是模拟卷。**工业产品真正的考场在现场**——那条产线的光照会变、来料会脏、工人操作会花样百出、车间温度湿度粉尘全和你实验室不一样。在你桌上跑得漂漂亮亮，到现场露馅，是常态，不是意外。

所以最关键的问题变成了：怎么把新系统搬到现场,又不让它一上来就闯祸？

影子模式：先在旁边看，不许动手

答案叫影子模式（shadow mode）。名字很形象——让你的新系统像个影子一样,跟在现有流程旁边一起跑，**只看、只判断、只记录，但不许动手**。它说「这个不合格」，产线该怎么走还怎么走（还是老办法或老师傅说了算），它的判断只被默默记在一边。

然后你干一件事：把它记下来的判断，和现场真实结果（人工复检、老设备、老师傅）一条条比对。它说不合格的，真的不合格吗？它放过去的，真的没问题吗？

这招妙在哪——它**把上线的风险降到几乎为零**。因为在你攒够信心之前，它一次都没真正碰过产线。它判错一百次，也只是让你在报表上看到一百个错，一件产品都没被它误伤。你是在没有代价的情况下，看清了它到底几斤几两。等它跟着跑了足够久、比对下来一直靠谱，你再让它从「影子」转成「真身」，接管那个动作。

灰度：先上一条线，别铺满全厂

就算比对通过了，也别一激动就全厂铺开。用灰度——不是黑白一刀切上线，而是先放一小块「灰」的：先上一条线、一个班次，跑几天。

> 大白话
>
> 道理特别朴素：万一还有你没料到的坑，一条线的损失你兜得住，看得清，也改得起；全厂三十条线一起出问题，你连是哪儿先坏的都理不清，还得半夜被三十个电话吵醒。范围小，坏事就小，而且看得清因果——这正是产品经理该有的本能。

用数据判断，不用感觉判断

「我觉得还行」是验证里最危险的四个字。「还行」是多行？漏检率百分之一算行还是不算？你要是答不上来，就说明你根本没法判断它到底行不行——你只是在凭当天心情打分。

正确的顺序是倒过来的：**上线之前，先把「怎么算成功」用数字定死**。这就是第 3 章那个验收标准该还的债。比如：

- 真缺陷，必须抓住 99%（漏检率不超过 1%）——因为漏检的直接流到客户手里，最贵；

- 误报（把好的判成坏的）控制在 5% 以内——误报太多，工人烦到直接把系统关了，那就等于白做；
- 每个零件判定必须在 0.5 秒内出结果——慢了就跟不上产线节拍。

数字定死之后，影子模式攒的那批真实数据就有用了：拿真数据去比对这几个数字，达标就是达标，不达标就是不达标，没有「感觉」插嘴的余地。让 AI 帮你把这些数据做成一个看板——每天的漏检、误报、耗时画成曲线，摆在眼前。你不用会做图，但你得知道自己要盯哪几个数。

闭环：现场的反馈，得能流回来

上线不是终点，是另一件事的起点。现场每天都在给你反馈——这批漏检了、那个总误报、工人抱怨某个角度它老看走眼。这些反馈如果只是变成一句抱怨、飘散在车间里，那它们就白疼了。

迭代闭环就是让这些反馈能流回来、变成改进的一条环路：

1. 现场发现问题（漏检、误报、工人吐槽某种情况）；
2. 把这些真实案例收集起来，变成新数据、或补进规格里的新条款；
3. 拿这些新料去改进系统；
4. 改完，再走一遍测试和影子模式验证——确认真改好了，也没弄坏别的。

然后回到第一步。这个环得一直转下去，不能转一圈就停。**闭环不闭，一切归零**——因为还有第 9 章提过的那个阴魂：数据漂移。现场是活的、会变的，来料换了供应商、车间换了灯、季节变了湿度，模型对着一个变了的世界，准头会慢慢退化。你今天验收合格的系统，半年后可能悄悄烂掉，而且没人告诉你。持续把现场的新情况喂回去，是让它别烂掉的唯一办法。

AI 能帮你什么，帮不了你什么

这一章从头到尾，AI 都在给你打下手，但有几件事它死活替不了你——认清这条线，你才不会把命交错人。

AI 能帮你的：把你指定的情况写成自动测试；搭现场数据看板；从成堆的现场日志里翻出规律（「一到下午三点误报就飙高」这种，人眼看一周看不出来，它一扫就出来）。这些又快又好，放心用。

AI 帮不了你的：它不知道你这个场景「多好才算够好」——漏检 1% 到底能不能接受，取决于你的产品、你的客户、这批货流出去要赔多少，这是业务判断，只有你懂。它更不能替你去现场蹲点——车间里那股「哪儿不太对劲」的味道，工人一个皱眉、一次悄悄绕过系统的操作，那些藏着真问题的活信号，得靠你一双腿走到线边上、一双眼睛盯着看，才捞得回来。

大白话

一句话收尾：**验证不是给系统做的，是给「你敢不敢信它」这件事做的**。你越是不会写代码，越需要这套独立的、拿数据说话的手段，来替你回答那个最要命的问题——它，到底是不是真的对。

第十四章 · AI 骗你的时候，你怎么知道

AI 不骗你，但它会自信地把错东西递给你

先把一句最容易被误解的话说清楚：AI 不会像坏人那样，故意编个谎来害你。它没有害你的动机，甚至没有"动机"这个东西。但这不代表你就安全了。恰恰相反——一个不会说谎、却经常自信地给你错答案的东西，比一个会说谎的人更危险。因为骗子你还知道要防，而它是笑着、语气笃定、条理清楚地把一个错的结论端到你面前。

回想第二章我们说过的：AI 是一台概率性意图翻译器（它不是在"知道"答案，而是在猜"接下来最像样的词该是什么"）。猜得像，不等于猜得对。它可以把一个完全不存在的东西描述得跟真的一样，语法通顺、术语齐全、逻辑自洽——唯独是假的。

> 大白话
>
> 这一章要解决的，不是"AI 会不会犯错"（它一定会），而是"你这个看不懂代码的人，凭什么能发现它错了"。这是本书最硬的一关。过了，你就能一个人活下来；过不了，你就是那个"以为自己在掌控、其实被牵着走"的人。

说得再直白一点：**你最大的风险，从来不是 AI 犯错，而是你没有能力分辨它到底错没错，却以为一切尽在掌握。**

四类雷，各自长什么样

一、它凭空编造（幻觉）

幻觉就是 AI 一本正经地编出一个根本不存在的东西：一个不存在的函数、一个不存在的参数、一个不存在的接口、一段不存在的数据，或者一个听起来特别对的算法结论。

类比一下：你问一个特别好面子的实习生"咱们仓库里有没有 A 型号的传感器读取工具？"，他不想说"不知道"，于是张口就来"有的，叫 `read_sensor_A()`，传三个参数就行"。听起

来专业极了。你去仓库一翻，根本没这东西。他不是坏，他是"不擅长承认自己不知道"。AI 也一样。

破法有三个，都不需要你懂代码：

- **让它给出处。**"这个函数是哪个库的？版本几？给我官方文档链接。"编造的东西往往给不出真出处，或者给一个打开是 404 的链接。
- **让它自己验证。**"你确定这个参数真的存在吗？如果不存在会怎样？"很多时候你这么一问，它自己就改口了——这本身就是信号。
- **交叉问。**换个问法、隔一个新对话再问一遍同一件事。**两次对不上，几乎肯定有一次在编**——这是很强的报警信号。但反过来不成立：两次都一样，不等于就是真的（它可能稳定地编出同一个假答案）。所以「一致」只用来抓矛盾，真伪还得靠下面的「要出处」和「业务合理性」兜底。

二、代码"能跑"不等于"对"

这是小白最容易栽的坑。AI 给你一段代码，你（或者它）一运行，没报错，跑出来一个数——你松一口气："成了。"

没成。**跑通只证明一件事：没有语法错误。它完全不证明逻辑是对的。**

> 大白话
>
> 打个比方：一台称重设备，你放一袋 5 公斤的东西上去，屏幕亮了、显示了一个数字"137"，机器没死机。这只说明电路通、程序在转。可 5 公斤怎么会显示 137？它"跑起来了"，但它是错的。"不报错"和"结果对"是两件完全没关系的事。

所以，跑通之后真正的活儿才开始：**盯输出，别盯它有没有跑起来。**拿一个你心里已经知道正确答案的例子喂进去，看它吐出来的和你预期的对不对得上。这一步接上第十三章的测试、第九章的指标——你不需要看懂它内部怎么算的，你只需要检查"进去什么、出来什么、对不对"。

三、它讨好你（谄媚）

谄媚是一个特别隐蔽、也特别坑人的毛病：你说什么，它顺着圆；你一质疑，它立刻改口认错、道歉、给你一个新答案——哪怕它原来那个是对的。

这意味着一件残酷的事：**"它同意我"根本不能当成"我对了"的证据。**你夸它，它说你对；你骂它，它说你对；你自己都拿不准的时候试探它，它还是说你对。它像一面哈哈镜，永远把你想听的映回给你。

最坑的场景是这样：你隐隐觉得某个结论不对，去问它"是不是应该改成这样？"——它马上说"您说得对，确实应该这样"，然后把一个本来正确的东西改错了。你不但没纠错，反而被自己的犹豫带着它一起错了。

破法：**问它的时候，别把你的倾向性带进去。**不要问"是不是该用 A？"，而要问"A 和 B 各有什么问题，你选哪个、为什么"。给它中立的题面，逼它给理由，而不是给它一个它能顺着圆的答案。

四、它在边界上悄悄出错

这类雷最阴，因为平时完全看不出来。正常输入下它样样都对，你测了十遍都好好的，于是你信了。可一旦碰到极端的、空的、超出范围的、脏的输入，它就**静默出错**——不报警、不崩溃，只是悄悄给你一个错的结果。

接上第十二章说的防御性：正常路走通只是及格线，真正埋人的都在边界上。一个测厚度的算法，量正常板材都对，可来了一块翘边的、带油污的、或者根本没放料的空图像，它照样淡定地报一个数——那个数是它瞎编的，但它一声不吭。

五件你现在就能做的事

上面四类雷讲完，下面是骨架——五个具体动作，不喊口号，都能直接上手。

1. 因果追问

永远不要问"对不对"——它只会说"对"。要问因果和假设：

- "为什么要这样做，换一种行不行？"

- "如果输入换成 X，会发生什么？"
- "这一步如果出错，会错在哪、我怎么看得出来？"

因果追问的本质，是逼它把"它没想到的地方"暴露出来。一个真的想清楚了的答案，经得起你追问每一环的因果；一个编出来的答案，追两层就露馅。

2. 边界测试

边界测试就是专门拿"不正常"的输入去试它：空的、超大的、超小的、负的、乱码的、缺一半的、格式脏的。不测中间那些好走的路，专挑悬崖边推它一把，看它塌不塌。这是揪出上面第四类雷唯一靠谱的办法——你不主动去边界上试，边界上的错就会等到上线以后、在真实产线上找上门。

3. 让另一个 AI 来挑刺

这一招特别好用，也特别便宜。**开一个全新的对话**——一个没有前面那些"讨好你的上下文"的、干净的 AI，把代码或结论原样贴进去，只说一句："找出这里所有的问题和风险。"

> 大白话
>
> 为什么有效？因为谄媚是"顺着你之前说的话圆"。新对话里它不知道你想听什么，也没有需要维护的立场，反而会真刀真枪地找茬。用两个互相独立的 AI 对冲掉单个 AI 的盲目自信——这正是本书写作时用的评审思路：写的那个和审的那个，从不是同一个上下文。

4. 守住你能守的那条线：业务合理性

前面几招都还沾点技术。这一招是你最强的武器，而且只有你有——因为**你可能看不懂代码，但你懂这个行业。**

代码你判断不了，可结果合不合业务常识，你一眼就知道：

- "这个检测结果符合物理常识吗？"——一块钢板的厚度算出来是负数，或者是 3 米，不用看代码就知道错了。
- "这个预测符合我对这台设备的了解吗？"——它预测这台从不出故障的设备明天必坏，或者那台天天报警的设备永远健康，你的行业直觉会立刻拉警报。

这是最后一道闸，也是别人替不了你的一道闸。领域知识，就是你手里那把 AI 没有的尺子。

5. 小步、可回退

一次只让它改一点点，改完立刻验，验过了再改下一点。别一次让它重写一大坨——改坏了你连是哪儿坏的都找不到。接上第五章的闭环：每一步都留着"能退回去"的后路，改错了一键回到上一个好的版本，损失就永远只是一小步。

把"我信任 AI"换成"我验证 AI"

这一章从头到尾其实只在说一件事，现在把它拧成一句话交给你：

> 你不需要看懂每一行代码。但你必须有验证的习惯和手段，把"我信任 AI"换成"我验证 AI"。

信任是一种感觉，验证是一套动作。感觉会被谄媚喂饱、被流畅的措辞哄住、被"跑通了"骗过；动作不会——因果追问、边界测试、换个 AI 挑刺、拿业务常识做闸、小步可回退，这五个动作你做没做，是能看见的。

能不能不靠信任、靠验证活下来，就是"小白全能工程师"和"被 AI 牵着走还以为自己在开车的人"之间，那条唯一的分界线。站在验证这一边，你才真的是那个"一个人的产线"的主人。

第十五章 · 全能工程师的真实边界

你走到了最后一章。先说一句实在话，不是客套：如果你真的跟着前面十四章走了一遍——把想法逼成规格、搭出能点的原型、啃过视觉和时序、把模型塞进边缘盒子、让它在会坏的现实里跑住、还学会了不靠信任靠验证——那你现在具备的，是一种三年前几乎不存在的能力。

一个不写代码的人，独立把一台不太复杂的工业智能装备从想法做到上线，连算法都自己啃下来了。这在过去，是一个小团队几个月的事。现在你一个人加上 AI，能到这儿。

这不是错觉，也不是运气。这是真实发生的**能力位移**——门槛从「会实现」挪到了「会想清楚+会验证」，而后面这两样，恰好是你本来就该擅长的。

> 大白话
>
> 说人话：你没有变成程序员，但你现在能做出程序员才能做的东西。因为最难的那部分——打字实现——被 AI 接管了，剩下的最难的那部分——想清楚该做什么——一直是你的活儿。

但「全能」是个相对词

现在必须泼一盆冷水，而且是诚实的冷水，不是打击你。

「全能工程师」这个说法，容易让人误以为是「全知全能」。不是的。它的准确意思是：**在一个不太复杂的项目里，你不再因为某个环节不会而卡死**。视觉不会？AI 陪你啃。通信协议看不懂？AI 帮你翻译。这已经很了不起。

但「不卡死」不等于「能啃动任何东西」。有些地方，你加上现在这一代 AI，依然啃不动。硬扛只会把自己和产品一起拖下水。你得知道这些边界在哪儿，才能在对的时刻停下来找人。

啃不动之一：真正硬核的算法

你在书里做的视觉缺陷检测、时序异常检测，本质是「拟合一个函数」，是成熟路子，AI 陪你走得下来。但有一类算法不是这样：

- 安全攸关的——判断错了会伤人、会烧设备的那种；
- 精度要求逼到极限的——比如亚微米级测量、要跟国家计量基准对齐的；
- 需要严谨理论保证的——不是「测试跑通了就行」，而是要能**数学上证明**它在所有情况下都不会崩的。

这类东西，AI 能给你写出「看起来像那么回事」的代码，但它给不了你那个证明，也判断不出隐藏的失效模式。这里你需要真正的领域专家——控制理论的、信号处理的、做过认证的人。别不好意思请他们。

啃不动之二：系统变大之后的复杂度

做一台设备，和做「一百台设备联网、有中央调度、要 7×24 不停、坏了能远程诊断」的系统，是两回事。前者你能扛，后者是系统架构——就是「怎么把很多零件组织成一个不会互相绊倒的整体」的学问。

规模一大，复杂度不是线性涨的，是滚雪球的。这时候的难点不在某一行代码，而在整体怎么搭、故障怎么隔离、几年后别人接手怎么维护。AI 能帮你写局部，但撑不起整盘棋的判断。

啃不动之三：功能安全认证

如果你的设备会跟人的身体发生关系——机械臂、切割、高压、高温——那就绕不开功能安全：就是用一整套被行业和法规认可的流程，证明「这东西即使某个零件坏了，也不会伤到人」。相关行业有专门的标准和第三方认证。

这件事的核心不是技术，是**责任和合规**。它要的是签字、是可追溯的流程、是有资质的机构背书。AI 给不了你签字，也担不了责。碰到这个，找专业的人和机构，没有第二条路。

啃不动之四：深度性能优化和长期维护

让东西「能跑」，和让它「跑得又快又省又稳、还能被人维护五年」，中间隔着一条河。极致的性能优化（榨干每一毫秒、每一兆内存）、和大型系统的长期维护，都是需要经验积累的深功夫。你能做出第一版，但把它打磨成工业级、可规模化的产品，往往需要更专业的手。

大白话

一句话总结这四条：「能做出来」和「能做到工业级可靠、可认证、可规模化」之间，还隔着一段距离。你已经能跨过第一道门，但别以为第一道门后面就是终点。

那你真正不可替代的，是什么

讲完了边界，反过来问：既然 AI 这么能干，你还剩下什么？

恰恰相反。AI 越强，下面这几样越值钱——因为它们是 AI **放大**而不是**替代**的东西。

- **对领域的深刻理解。**你知道这台设备该干什么、什么算「好」、什么是「危险」。这是 AI 没有的——它没在车间待过，没见过那个次品是怎么流到客户手里、又是怎么被退回来的。
- **把模糊问题变精确的能力**（第 3 章）。「检测缺陷」是句空话，「在这个光照下、这个尺寸的划痕算缺陷、这个不算」才是能干活的规格。逼出这个的，是你。
- **判断和验证的能力**（第 13、14 章）。AI 会自信地给你错误答案，能识破它、能设计影子模式去验证的，是你。
- **知道自己不知道什么。**这是最稀缺的一条——能认出「这块我啃不动」，然后把它翻译成一个对的问题，去问对的人、对的工具。

你注意到没有：这四样里，没有一样是「会写代码」。会写代码，现在是 AI 最擅长的。而这四样，是 AI 时代真正稀缺的东西，也正好是这本书从头到尾在训练你的东西。

这本书赌的是什么

回到最开头。这本书从第一页就压了一个赌注：

> 在 AI 时代，「知道这台设备该做什么」比「会打字实现它」更稀缺、更值钱。

过去二十年反过来——想法一毛钱一打，能实现的人才金贵。所以懂业务的人只能提需求，实现攥在别人手里。现在实现这一环被 AI 大幅补齐了，天平就翻了过来：**稀缺的重新变成了「想清楚」，而不是「做出来」**。

一个懂业务、会想清楚、肯较真验证的人，加上 AI，能走到过去需要一整个团队才能到的地方。这是真的。

但这里有个诚实的补充，也是全书的落点：**你走多远，最终不取决于 AI 有多强，取决于你把问题想得多清楚、把它验证得多较真。**同一个 AI，交给一个想不清楚的人，做出来的是一堆看着能跑、上线就翻车的玩意儿；交给一个想得透、验得狠的人，做出来的是能真在车间里干活的设备。差别不在工具，在用工具的人。

重新定义「全能工程师」

所以，收个尾。别把「全能工程师」理解成「什么都会的超人」。它的真实含义朴素得多：

> 一个能借 AI，把自己的领域理解，变成可运行、可交付的现实的人。

关键词是「自己的领域理解」。AI 是那双手，你是那个大脑——知道要做什么、知道什么是对的、知道什么时候该停下来找人。手可以借，脑子借不来。

最后一句大实话，不给你灌鸡汤：这条路不是一劳永逸的。工具会变，模型明年又不一样，你今天学的某个具体操作可能过时。但这本书真正想留给你的，不是某个操作，是一副能长期用下去的心智——**爱因果、逼精确、肯验证、知边界**。有了这副心智，工具再怎么变，你都能接着往前走。

该请专家的时候请专家，能自己扛的时候自己扛，永远知道自己站在哪儿。这就够了。剩下的，去做你那台设备吧。