《软件测试课程设计》讲稿（第 9 章）

第 11 周 · 2 学时

广东金融学院 计算机学院 | 周宇文

讲稿可照读版 | 由课件逐页撰写

第 9 章　软件测试自动化及其框架

第 11 周 · 2 学时（90 分钟）

《软件测试课程设计》第 9 章 讲稿（第 11 周 · 2 学时 90 分钟）

> 课程：软件测试课程设计（154442008） | 广东金融学院 计算机学院 | 专业选修
>
> 章节：第 9 章 软件测试自动化及其框架 —— 让机器替你反复验，你去做机器做不了的事
>
> 学时：**2 学时（90 分钟）**
>
> 教材：朱少民《软件测试方法和技术（第 4 版）》第 2 篇 软件测试技术
>
> 配套课件：逐页还原版 **75 页**（`lecture9/`，云端 http://111.231.0.226:2345/lecture9/index.html）
>
> 教案对应：第 11 次课「软件测试自动化及其框架」
>
> 本讲重点：自动化的价值边界与 ROI 判断；自动化层次与金字塔；测试框架类型（数据驱动／关键字驱动／混合／行为驱动／模块化）；主流框架与工具；脚本设计与可维护性（Page Object、断言、数据管理）；持续集成中的自动化流水线；自动化常见失败原因
>
> 本讲难点：什么该自动化、什么不该；框架选型的判断依据；脚本"越写越难维护"的根因与治理
>
> **本讲稿为可照读完整版**：每页含【口播】【板书】【提问＋预设回答】【易错点/考点】【过渡】；带 ★ 的

> 页为可压缩页。

一、教学目标

维度	目标（课后可检验）
知识	① 能说出自动化适合与不适合的测试类型，并给出理由；② 能画出自动化测试金字塔并解释各层 ROI；③ 能说出五类测试框架的核心机制与适用场景；④ 能列出脚本可维护性的四条设计原则；⑤ 能说明自动化在 CI 流水线中的位置与门禁作用
能力	① 能为一个项目写出"该自动化清单"与"暂不自动化清单"并说明依据；② 能为一个页面选型并搭出最小 Page Object 结构；③ 能诊断"脚本天天红"的三类根因
素质（思政）	① 由 ROI 判断体会务实、不迷信工具；② 由框架选型体会因事制宜；③ 由脚本可维护体会对后来者负责的工匠精神；④ 由 CI 门禁体会规则意识；⑤ 由失败复盘体会诚实反思

本讲要建立的一个总印象：自动化不是"把手工测试翻译成脚本"，而是**用工程手段把"重复验证"变成可复用资产**；它的敌人不是"写不出来"，而是"**维护不下去**"。

二、时间分配总表

> **容量提示**：本章课件 **75 页**，讲稿为素材全量版（口播约 3.5 万字，≈145 分钟），课时 90 分钟。请按下表取用：
>
> 1. **重点页慢讲**：自动化价值与 ROI、金字塔、五类框架对比、Page Object、CI 流水线、失败治理。
> 2. **快速带过**：各工具安装与截图页（★）。
> 3. **当堂动手**：写"该自动化清单"、画一个最小 Page Object。

时段	页码	内容	分钟	备注
第 1	9.1–9.20	自动化的价值与边界、ROI、层次与金字塔	26	**ROI 与金字塔必讲**

节				
第 1 节	9.21–9.40	框架类型：数据驱动、关键字驱动、混合、BDD、模块化	24	**五类对比表必讲**
——	——	**课间休息 10 分钟**	——	——
第 2 节	9.41–9.60	主流工具与框架、脚本设计（Page Object／断言／数据）	22	工具页可略，设计原则必讲
第 2 节	9.61–9.75	CI 流水线中的自动化、失败治理、小结与作业	18	**失败治理必讲**

> **压缩优先级**：先压缩工具安装截图 → 再压缩单一框架的 API 细节 → 最后才压缩 ROI、金字塔、五类框架、Page Object、CI 门禁、失败治理。

三、逐页讲稿

> 每页第一段是可直接朗读的口播；其后是板书、提问与预设回答、易错点/考点、过渡语。
>
> 页码锚点与课件页一一对应（9.1–9.75）：http://111.231.0.226:2345/lecture9/index.html

【9.1】软件测试方法和技术（1 分钟）

> 一个版本要跑三天回归，你手上只有三个人。

>

> 老板说明天开始每天跑一遍。三个人乘三，九个全职，人力先压不住。加人呢？第二遍开始人就走神，漏测往上走。所以卡住你的不是要不要自动化，是哪一段活可以交给机器反复做。人工回归还有个毛病：结果靠人判，今天宽一点、明天严一点，同一批用例两天的结论都能不一样。

>

> 上一章问的是换一种语言环境还能不能用。第 9 章，测试自动化及其框架，问的是同一批测试能不能不靠人一次一次地点。

>

> 先把话说准。自动化不是把手工用例翻译成代码。它是把测试变成能反复执行、能被机器判定的东西。翻成代码以后人还得一条条看结果，那不叫自动化。

>

> 这一章走四块。内涵与边界，说清什么算自动化、边界在哪儿。数据驱动和关键字驱动，讲脚本怎么组织：脚本不动、数据在变；引擎不动、关键字在变。xUnit，看单元框架的骨架。最后是 API 与 Web、移动 UI 框架，收在集成框架上。

>

> 四块不是并列的知识点，是一条线。先想清边界，再学怎么组织，再进框架，最后拼成一套天天能跑的。

>

> 这个顺序别倒。判断力在前，工具在后。先学工具，换一个库你就得从头再来。带着一个问题听：要求每天跑一遍全量回归，你怎么排班？

提问：老板要求"每天全量回归一遍"，而全套用例手工执行要三个人干三天，你怎么回答他？

预设回答：

- 答"加人"——三个人三天，每天一遍要九个全职，成本先崩；而且人会疲劳，第二遍开始漏测率就往上走。

- 答"全部做成自动化"——方向对，但太粗。哪一层值得自动化，才是关键判断；UI 层全量自动化往往是最贵、最脆的选择。

- 答"挑关键路径自动、剩下按周期手工"——最接近工程实际，第 5 页给分层策略时会给依据。

- 收口：这三答的差别，就是"想不想自动化"和"懂不懂自动化"的差别。

易错点 / 考点：① 章名是"测试自动化"，不是"自动化测试"，两个词宽窄不同，第 8 页专门辨析；② 判断题陷阱："自动化测试可以替代手工测试"——错，本章大量篇幅在讲边界；③ 四块的名称与顺序要记牢，简答常按这个结构给分。

过渡：先把路线摊开，你才知道每一页站在整章的什么位置。

【9.2】自动化测试的内涵与原理（1 分钟）

> 四块内容，只让你记一块，你记哪一块？

>

> 很多人选工具那一块，因为它马上能用。可工具的更替最快。你今年学的库，明年可能改名、换协议。

>

> 第一块，内涵与原理。为什么重视它、边界在哪、能分几类。看着最虚，方向却由它定。方向错了，工具越熟，浪费越大。边界那部分最容易被跳过。什么都想自动化的人，最后往往什么都没自动化好。

>

> 第二块，数据驱动、关键字驱动。脚本不动、数据在变；引擎不动、关键字在变。今天的框架几乎都能在这两条线上找到位置。为什么单讲它？因为这是脚本的组织方式，跟用哪个库关系不大。二十年里载体换了几轮，这两条没变。

>

> 第三块，xUnit。以 JUnit 为主线看单元框架的骨架。它要解决五件事：发现用例、准备环境、断言、清理、出报告。任何单元框架都躲不开这五件。为什么单看 JUnit？它是这一族里最典型的一份，看懂它，别的框架你上手就快。

>

> 第四块，API 与 Web、移动 UI 框架，收在集成框架。自动化不止一类、不止一个团队的时候，怎么装进同一套调度和报告。

>

> 串起来是一句话：想清楚，做出来，管起来。判断力在前，工具在后。顺序反了，学到的只是操作步骤。

> **提问：**这四块里，你认为哪一块在未来五年最不容易过时？

> **预设回答：**

- 答"框架和工具"——恰恰相反，工具更替最快，你今年学的库明年可能改名、换协议。

- 答"内涵与原理"——对。判断要不要自动化、自动化到哪一层，这份能力换任何工具都用得上。

- 答"数据驱动和关键字驱动"——也有道理，这两种组织思想二十年没变过，只是换了载体。

- 收口：所以讲课重心放在"为什么这么做"，命令和配置你课后照实验做就行。

> **易错点 / 考点：** ① 四块的名称与顺序；② 常考简答"自动化测试的分层与典型框架对应关系"——单元对 xUnit、接口对 Rest-Assured 或 SoapUI、UI 对 Selenium 或 Appium；③ 别把脚本技术和框架混成一件事：脚本技术是组织方式，框架是承载它的工程结构。

> **过渡：** 先看一组数据，看看这个行业到底把自动化做到了什么程度。

【9.3】自动化测试越来越受重视（2 分钟）

> 先看一组数字，出处是课件引的调查，2021 年。

>

> 自动化做到高水平，也就是覆盖率达到 90% 以上的团队，从 4% 提高到 9%。大部分已经实现自动化的，也就是超过 50% 水平的，从 32% 到 43.8%。这是调查数据，不是实验结论，读方向可以，别当定律。

>

> 这两个数字放一起，能读出三层。

>

> 第一层，方向是定的。过半团队已经不是试水，是大部分用例都在跑自动化。它从技术尝鲜变成了基本配置。

>

> 第二层，高水平还是少数。9% 这个数字更值得看。到了 2021 年，做到九成以上自动化的团队还不到十分之一。为什么这么难？不是工具难装，是维护难。脚本写了一千条，界面改一次废掉两百条，谁都不敢再加投入。维护难在三条：用例本身没设计好、脚本跟界面绑死、环境三天两头变。三条凑一起，改一次就够你修一周。

>

> 第三层，别拿别人的数字当自己的考核。把覆盖率 90% 写进考核，最省事的做法是什么？写一堆只断言"页面能打开"的空脚本。数字立刻好看，缺陷一个没抓住。所以考核要落在抓出多少缺陷上，不能落在覆盖率上。

>

> 所以这组数字想说的不是"要重视自动化"。它说的是：重视，还得看它抓不抓得住缺陷。

> **提问：**一个团队报"自动化覆盖率 92%"，你要追问哪两个问题，才能判断这个数字值不值钱？

> **预设回答：**

- 答"问脚本有多少条"——条数不等于价值，一千条空断言也是九百多条。

- 答"问这些脚本抓出过多少缺陷"——问到点上了，能抓缺陷的脚本才算资产。

- 答"问维护成本，一天要花多少时间修脚本"——很关键。维护成本高于收益的自动化，是负资产。

- 追问收口：还可以问断言是什么。只断言元素存在、不断言业务结果的脚本，只是"页面能打开"的探针。

> **易错点 / 考点：**① 两个百分比的方向别记反，4 到 9、32 到 43.8；② 数据出自调查、年份是 2021 年，答题写"据课件调查数据"，不要自己换数字；③ 高频概念题"为什么高水平的自动化比例这么低"——答维护成本、用例可测性、环境稳定性三条最稳。

> **过渡：**数据说的是行业趋势，再换一个更直接的观察角度。

【9.4】自动化测试越来越受重视 – 人才市场（2 分钟）

> 招聘网站上，测试岗的要求这几年被改写了一遍。

>

> 课件只给一句判断：自动化测试第一。意思是测试类岗位的能力要求里，自动化排在第一需求位。这句话信息量不小，拆三层。

>

> 第一层，岗位在变形。过去的描述是能设计用例、能执行、能提缺陷。现在几乎都写：熟悉至少一种自动化框架、有脚本开发能力、了解 CI/CD。这套要求本身没错，只是不够用了。不是用人单位变挑剔，是纯手工回归撑不住了。版本越来越快，一轮回归的时间预算被压到小时级。版本从月更变成周更，人跑一轮的时间可没缩短。

>

> 第二层，能力变成三件套。编程能力，至少一门语言能读能写。框架能力，能搭、能用、能改，不只是会点录制按钮。工程能力，版本管理、环境、流水线、报告。缺一样，你只能干跑脚本的活，干不了让脚本跑得住的活。还有一层隐性的：你得看得懂别人写的脚本，也改得动。

>

> 第三层，工具会过时，测试设计思维不会。今天最热的 UI 框架，五年后大概率被别的替掉。可这个场景要不要自动化、这条断言能不能发现回归、这次失败是真缺陷还是脚本脆。这些判断，换什么工具都用得上。前两层跟着工具走，这一层跟着你走。

>

> 所以课上我不会只教你调接口，我会教你怎么判断。市场要的是会写、会想、会兜住的人。

> **提问：**招聘要求写"熟悉 Selenium"，和真正能独立搭起一套自动化的人，差在哪？

> **预设回答：**

- 答"差在会不会写代码"——只是第一层。会写脚本的人不少，能设计出改版之后不用大改的结构的人少。

- 答"差在会不会搭框架"——对，框架意味着分层、对象库、数据管理、报告、调度这一整套。

- 答"差在会不会判断要不要自动化"——这一层最值钱，也是面试官最想听到的。

- 收口：这三层正好对上本章四块：判断力对第一块，框架对第四块，脚本组织对第二、三块。

> **易错点 / 考点：**① "自动化测试第一"是课件对人才市场的判断，简答可直接引用；② 不要理解成手工测试没用了，后面分层那一段会说手工 E2E 要适当加强；③ 面试类案例题问"你怎么评估一个候选人的自动化能力"，按三件套回答。

> **过渡：**既然重要，那自动化到底往哪一层投？下一页给本章最重要的一个判断。

【9.5】自动化测试分层策略（3 分钟）

> 最容易做错的一个决定，是把自动化全押在界面层。

>

> 课件给四句话。第一句：单元测试更容易自动化，回报也更高。三个原因。跑得极快，几千条几秒钟。定位

> 极准，挂了就指向那个函数。极稳定，不依赖浏览器、网络和界面改版。同样的投入，砸在单元层换来的检查次数最多。

>

> 第二句，API 自动化基本能做到 100%。接口是契约，输入输出结构稳定，定下来很少变。而且一个下单流程的核心规则全在后端接口里，界面只是把按钮摆出来。

>

> 第三句，适当加强手工 E2E，业务层那一段。适当和加强要一起读。端到端要保留、要做扎实，但别指望它扛全量回归。它贵、慢、脆，适合覆盖关键业务链路和新功能验收。为什么叫适当？因为它是唯一能验证用户整条路走不走得通的手段，砍掉就没人管整体了。

>

> 第四句是概括：以底层测试、接口测试、功能逻辑测试为主，尽量避免 UI 测试。避免不是禁止，是别把它当主力。一次界面改版，UI 脚本的维护量能顶它半年的执行时间。

>

> 课件还给四个好处：缺陷更易定位、效率更高、反映真实需求、更加接近业务。越往底层定位越准，越往业务越贴近用户价值。这个形状叫测试金字塔，底盘大、塔尖小。倒过来就是反模式，有人叫它冰淇淋甜筒。

>

> 一句话收住：投在哪一层，先看回报，不看你听说谁在用什么。

提问：一个团队把 80% 的自动化投入放在 UI 层，单元测试几乎没有，你能预测他们半年后遇到什么？

预设回答：

- 答"脚本维护不过来"——必然。界面一改版，几十条脚本一起红，团队天天在修定位器。

- 答"跑得太慢，没人愿意等"——也是必然。UI 脚本一条几十秒，上千条就是几小时，最后只能夜里跑，白天拿不到反馈。

- 答"这样更接近用户，所以更好"——最典型的错答。接近用户是价值判断，成本与稳定性同样要算。

- 追问收口：正确的做法是把断言往接口和单元下沉，UI 只留关键路径冒烟。

易错点 / 考点：① 四句的顺序与措辞，特别是"适当加强手工 E2E"不能记成"取消手工"；② 金字塔三层的比

例关系是考点，底层多、顶层少；③ 案例题问"为什么单元测试回报最高"，答快、准、稳；④ "尽量避免 UI 测试"要读成"避免把 UI 当主力"，不是不做 UI 测试。

【思政落点 1】：分层这一段算的是回报，落的是务实。行业里最常见的错，是把"先进"当"正确"：听说 UI 自动化时髦就全押 UI，听说某个框架火就全量迁移，钱花了，效果没出来。工程判断的第一步是先算投入产出：哪一层回报最高、维护成本扛不扛得住。不迷信工具、不追风口，按收益排序，这是工程师最基本的职业理性。

过渡：理论很漂亮，可现实往往不长这样。下一页看骨感在哪。

【9.6】现实是骨感的（2 分钟）

金字塔画得出来，长不出来。

>

课件只给一句：现实是骨感的。它接在分层策略后面，是打预防针。五条原因，你对号入座。

>

第一条，UI 自动化最好卖。演示的时候浏览器自己跑起来，视觉冲击最强。单元测试跑完只有一排绿点，没人鼓掌。资源自然往看得见的地方走。

>

第二条，脚本脆。定位器绑在会变的属性上，等待写成固定睡眠，用例之间互相污染数据。三样凑齐，红灯率能到三成。团队很快学会一件事：报告里的红，习惯性忽略。

>

第三条，环境不稳。缺数据、服务没起、环境被别人占了。脚本挂了，你分不清是产品坏了还是环境坏了。

>

第四条，用例本身不可自动化。需求含糊、界面没有稳定标识、流程依赖人工审批。这些工具解决不了，是可测性问题，得回到需求和设计去改。

>

> 第五条，指标虚荣。考核覆盖率、考核脚本条数，于是大家写能跑通但不断言的脚本。数字漂亮，缺陷照漏。
>
> 这五条有个共同点：没有一条是技术难题，全是投入方向和组织习惯的问题。

>

> 三条对策跟着来。先算回报再动手，按收益排序，不按时髦排序。先做单元和接口，把金字塔底盘垫起来。
>
> 把维护成本当一等公民，谁写红谁修，红过一天就下线。为什么先做底层？它便宜，改它也不牵动别人。

> **提问：**团队里有一批脚本连续红了两个月，大家跑报告时直接跳过。你怎么处理这批脚本？

> **预设回答：**

- 答"先修好"——要看值不值。修之前先判断：它测的那个场景还有价值吗？

- 答"删掉"——对一部分脚本这是正确答案。长期红灯的脚本等于没有脚本，还更糟，因为它训练团队忽略失败。

- 答"先隔离标记，评估后再决定修或删"——工程上最稳。给它们打上待处理标记，限期内要么修好要么删掉。

- 收口：关键不是修还是删，是不允许红灯长期存在，那会毁掉整套自动化的可信度。

> **易错点 / 考点：**① 这是本章的问题意识页，简答常问"自动化常见失败原因"，五条任答三条即可；② "长期红灯等于负资产"这个判断案例题高频；③ 别把"用例不可自动化"归到工具头上，它是可测性设计问题，第 6 章讲过。

> **过渡：**骨感的现实需要一个分类工具来对付。

【9.7】自动化测试四象限（2 分钟）

> 有个特别省事的判断法：两句话，四个格子。

>

> 两个维度。面向业务还是面向技术，适合手工还是适合自动化。两两组合，四格。为什么值得画？很多争论是在两个格子里各说各话，谁也说服不了谁。课件把详细讨论指向《软件质量报道》的公众号文章，图按业界通行的分法讲，你记用法就行。

>

> 面向技术又适合自动化的那一格，是主战场：单元测试、接口契约测试、静态分析、性能基线。投入产出最

> 高。

>

> 面向业务又适合自动化的那一格，做得了，但要挑：功能回归、验收用例的重复执行。挑稳定、关键的业务路径。

>

> 面向业务又适合手工的那一格：探索式测试、新功能第一次验收、易用性评估。自动化做不了，它的价值恰恰在人在场时临场发现问题。

>

> 面向技术又适合手工的那一格：渗透里的创造性部分、故障演练、性能定位分析。工具给数据，判断靠人。为什么这么分？自动化的强项是比对，人的强项是第一次见到东西时的判断。

>

> 四格不贴标签，它逼你先回答一个前提问题。这个格子最大的用处，是挡住"什么都自动化"的冲动。任何测试活动往里一放，先问两句：面向业务还是技术？需要人的判断还是只要机器比对？答完，要不要自动化基本就定了。

>

> 记住一句：自动化擅长重复确认已知，人擅长首次发现未知。

提问：把"新功能第一次验收"放进象限，它落在哪一格？为什么不能自动化？

预设回答：

- 答"面向业务加适合手工"——位置对了。

- 答"因为还没写脚本"——错答。不是来不及写，是这一轮的目的就是发现未知问题，你连预期结果都还不完全清楚，写不出可靠断言。

- 答"因为它要求人的判断，比如这个交互是不是别扭"——对，这就是首次发现未知的价值。

- 追问收口：功能稳定了、预期结果明确了、要反复回归了，它就移到面向业务加适合自动化那一格。同一个功能会随生命周期换格子。

易错点 / 考点：① 两个维度的名称；② 四格的典型活动各能举一例；③ 关键判断句"自动化擅长重复确认已

知，人擅长首次发现未知"，简答可直接用；④ 还有同一功能会换格子这个动态视角，案例题常问"新功能要不要立刻上自动化"。

> **过渡：**定了做什么，接下来要看"自动化"这个词到底指多大范围。

【9.8】自动化测试的内涵（3 分钟）

同一个词，在两个人嘴里可能不是一个意思。自动化就是这样。

>

> 课件四句话，一句一句抠。

>

> 第一句：通过平台、系统或工具自动完成测试的某类工作，都可以归为测试自动化。关键在"某类工作"四个字。环境自动部署算，用例自动生成算，缺陷自动分派也算。这是大概念，范围很宽。

>

> 第二句：自动化测试更侧重测试用例或测试数据的生成、测试执行、测试结果呈现。这是小概念，也是平时说的那个意思：用例怎么来、怎么跑、结果怎么给人看。三件事并列，很多团队只做了中间那件。两头都靠人，收益自然上不去。只做执行，用例靠人想、报告靠人看，这就是很多团队投了钱却算不出回报的原因。

>

> 第三句：自动化测试为评审提供辅助工具和代码静态分析。这一句把静态分析拉进来了。不用运行程序，用工具扫代码、扫配置、扫依赖。评审从凭经验看，变成先看机器报告、再看人的判断。

>

> 第四句：自动化测试可以覆盖系统的接口测试、UI 功能测试和专项测试。这句纠正一个误解，以为自动化就是点点点。性能、安全、兼容这些专项，自动化程度可能比功能测试还高。

>

> 要一个更硬的判据？结果判定是不是机器做的。是，算自动化测试；不是，最多算自动化的辅助。

> **提问：**"我们买了个测试管理平台，用例自动派单、报告自动汇总，所以我们实现了自动化测试。"这句话对

> 不对？

> **预设回答：**

- 答"对，都是自动的"——只对一半。按大概念算测试自动化，按小概念它缺了最关键的那一段：执行自动化。

- 答"不对，用例还是要人写、要人跑，那叫管理自动化"——区分到位。管理自动化省的是协作成本，不是执行成本。

- 答"要看它有没有让机器去判定结果"——抓住核心了。

- 收口：判据就是一句，结果判定是不是机器做的。是，才算自动化测试；不是，最多算自动化的辅助。

> **易错点 / 考点：** ① 大概念与小概念的区别是本章第一个辨析点；② 三件事生成、执行、呈现要能默写，简答常问"自动化测试包括哪些环节的自动化"；③ 常考判断"静态分析不属于自动化测试"——错，课件明确把它纳入。

> **过渡：** 范围划清了，下一页把它切成六类，你对号入座看自己缺哪块。

【9.9】自动化测试的分类（3 分钟）

> 六类，从便宜到贵排一遍。

>

> 第一类，代码静态分析。不运行程序，用工具扫源码、扫配置、扫依赖。投入最小，接进流水线就开始了。收益也快，空指针、资源没关、硬编码密码，几分钟扫出一批。它是所有自动化的第一道闸门。

>

> 第二类，单元测试自动化。开发写，跑得最快，定位最准。金字塔的底盘。为什么让开发写？只有他知道这段代码的边界条件在哪儿。

>

> 第三类，API 接口自动化。国内团队性价比最高的一块。接口稳定、执行快、能覆盖业务主干。第 27 页会给完整的架构。

>

> 第四类，Web UI 自动化。Selenium 那一套，最直观，也最脆。它管的是用户看到的那一层到底对不对。

>

> 第五类，移动 App UI 自动化。Appium、Airtest 这一套。比 Web 多一层设备管理：真机、模拟器、系统版本、权限弹窗。

>

> 第六类，自动化测试集成框架。前五类是零件，这一类是总装线：脚本、数据、调度、报告、环境管到一起。小团队先用开源拼，规模上来就必须有自己的集成层。脚本一多，谁跑、多久跑、失败了通知谁，全靠人记就乱了。

>

> 六类不是并列关系，有先后。先静态分析和单元，便宜快。再接口，性价比。最后 UI，贵且脆。今年只许你做一类，做接口。

> **提问：**如果团队只允许你今年做一类自动化，你选哪一类？给两个理由。

> **预设回答：**

- 答"UI，因为最像用户"——直觉对，性价比最低。除非你的系统就是纯前端、后端极简。

- 答"接口，因为稳定、快、覆盖主干"——主流正确答案。接口变更频率远低于界面，执行速度远高于 UI，业务逻辑又主要在后端。

- 答"单元，因为回报最高"——也对，但有前提：开发愿意写、代码结构支持。团队没有单测习惯时，接口是更现实的起点。

- 追问收口：还要看痛点在哪。回归太慢选接口，线上事故多选静态分析加单元，界面兼容问题多才轮到 UI。

> **易错点 / 考点：**① 六类的名称与顺序，简答高频；② 每类至少说一个代表工具：静态分析、单元 JUnit 或 pytest、接口 Rest-Assured 或 SoapUI、Web Selenium 或 Cypress、移动 Appium 或 Airtest、集成 Robot Framework；③ "哪类回报最高"要分清：单元回报最高是课件口径，接口性价比最高是工程口径，两者不矛盾。

> **过渡：**从最便宜的一类开始。

【9.10】代码静态分析（1 分钟）

> 代码刚写完，还没编译通过，问题就已经能挑出来了。编译器还没跑，一个只读源码的工具已经把可疑点列在屏幕上。

>

> 静态分析就干这件事：不运行程序，也能找出问题。价值在一个字，早。第 1 章讲过，缺陷越晚发现代价越高。静态分析就是把发现时间推到最前面的手段。越往后，改一行要动的测试、要回归的范围就越大。

>

> 这一小节按三条线走，三条线是同一个动作的三个角度。

>

> 第一条，评审的辅助工具。人评审最容易漏的，是几百行差异里那个边界判断。工具负责机械地看：格式、命名、空指针、硬编码密钥。人负责理解地看：逻辑对不对、设计合不合理。这个分工不能换。

>

> 第二条，分析原理。词法、语法、控制流、数据流，一直到符号执行和值依赖分析。它顺带解释了一个问题：工具为什么会误报。原理清楚了，你就不会把误报当成工具坏了。不然后果常见：整个工具被关掉，理由是太吵。

>

> 第三条，SAST。安全漏洞里有很大一部分不跑起来就能看出来。拼接的 SQL、没校验的输入、配置文件里的密码。这类问题不用跑也能看，所以它最便宜。

>

> 课件这页写的是原理、框架与工具。三条线合起来，定位只有一句：所有自动化里最便宜的一道闸门。闸门的意思是它挡在最前面，成本最低，挡不住才轮到后面更贵的手段。

> **提问：**静态分析不用跑程序，那它靠什么判断"这里可能有问题"？

> **预设回答：**

- 答"靠规则和经验"——对一半。正则和规则库确实是基础。

- 答"靠语法树和控制流，看数据能不能流到危险的地方"——说到原理层了。

- 答"靠猜"——说法粗糙，但抓住了要害：静态分析是近似推理，近似就有误报，所以要人复核。

- 收口：下一页先看人评审怎么被工具辅助。

> **易错点 / 考点：** ① 定义是不运行程序；② 与第 3 章静态测试的关系：静态分析是静态测试的自动化形式；③ 判断题"静态分析能发现所有缺陷"——错，运行时才暴露的问题它看不到，比如并发、性能、资源竞争。

> **过渡：** 先看最贴近日常开发的那一环，代码评审。

【9.11】代码评审辅助工具（2 分钟）

> 邮件加压缩包，这是十年前评审的样子。

>

> 发一封邮件，附一个压缩包，评审人打开对比，写几条意见回邮件，然后就没有然后了。三个毛病。看不见全貌，改动散在文件里。留不下痕迹，意见在邮件里，三个月后查不到。第三个是拦不住，说不改也能合。

>

> 课件给的入口是 GitHub 那份文档。标题是 Collaborating with pull requests。合并请求这套流程，把三件事都解决了。

>

> 差异可视。它以差异块的形式把改了什么摆出来，评审人按块看，不会淹没在整份文件里。为什么按块重要？人的注意力就那么多，摊到整份文件上等于没看。

>

> 讨论可追溯。每条意见挂在具体某一行，形成线程。谁提的、谁改的、改没改，全留痕。为什么要留痕？半年后有人问这行为什么这么写，邮件早找不着了。

>

> 合并有闸门。可以配必须通过的检查，比如 CI 跑单测、跑静态扫描。也可以配必须签字的评审人。不满足就不许合。

>

> 这就是辅助工具这四个字的准确含义。它不替你判断代码好不好，它保证判断这件事一定发生，并且留得下证据。机器能查的交给机器：格式、单测、静态告警。机器查不了的留给人：这段逻辑是不是多余、这个抽象合不合理。

>

> 一句话：机器查规范，人查意图。

提问：把"必须通过 CI 检查才能合并"这条规则去掉，评审质量会下降吗？

> **预设回答：**

- 答"不会，人看就行了"——人会累、会赶工期、会给面子。规则一去，最先失守的是"今天太忙，先合了再说"。

- 答"会，没有硬约束，评审会变成走过场"——对。闸门的价值在于它不依赖人的自觉。

- 答"会下降，但小团队不用这么严"——可以讨论。小团队能简化流程，可"改完必须被看过"这条底线不能没有。

- 收口：工具的价值不在更聪明，在更可靠地执行纪律。

易错点 / 考点：① 三大作用是可视、可追溯、有闸门；② 案例题问"怎么提高代码评审的有效性"，答缩小粒度、机器先查、强制闸门；③ 别把合并请求当成版本管理功能，它的实质是协作与准入控制。

> **过渡：**工具能查的那部分靠静态分析，它怎么"看"代码？

【9.12】代码静态分析（3 分钟）

工具凭什么敢说你这段代码有问题？

>

> 它不跑程序，靠的是一条流水线，四层，一层比一层深。

>

> 最浅一层，词法分析。把源码那串字符切成 Token：关键字、标识符、运算符、常量。它只认词长什么样，不问词是什么意思。所以这一层能查的，基本是字面量。

>

> 第二层，语法分析。把 Token 按语法规则拼成抽象语法树，也就是 AST。到这一步工具知道了：这是一个 if，条件是 a 大于 0。多数工具从这里起步。

>

> 第三层，控制流和数据流。控制流图回答程序可能从哪儿走到哪儿。数据流回答这个变量在哪儿赋值、能流到哪儿。真问题多在这一层：没初始化的变量、没关的句柄、拼进 SQL 的用户输入。

>

> 第四层，模拟程序执行的技术。符号执行、抽象解释、值依赖分析。再配数学约束求解工具，做路径约减或者可达性分析，目的是减少误报、增加效率。这句话点出核心矛盾：精度和成本要取舍。全路径枚举精度高但跑不完，表面扫描跑得快但误报一片。约束求解是中间那根平衡木。

>

> 中间还有一层中间表示，IR。源码先译成更规整、更接近机器语义的形式再分析。好处是一套引擎能适配多种语言。

>

> 串起来：字符、Token、AST、IR、图、结论。越往下理解越深，代价越高。

提问：为什么静态分析工具总会报一些"看着没问题"的告警？从这条流水线上找原因。

预设回答：

- 答"因为工具不够聪明"——太笼统。根子在近似：它不能真的把程序跑完所有输入。

- 答"因为它在某些分支上做了保守假设"——对。它无法确定一个变量运行时到底是不是空，就按可能为空报出来。

- 答"因为规则库太宽"——也是原因之一，正则类工具特别明显。

- 追问收口：所以成熟用法是分级。高危必须清零，中危人工复核，低危看趋势。要求零告警的团队，最后通常是把规则全关了。

易错点 / 考点：① 流水线顺序是简答高频；② 三种高级技术的名称要能写：符号执行、抽象解释、值依赖分析；③ 约束求解的两个用途是路径约减、可达性分析，目的写减少误报、提高效率；④ 判断"静态分析误报是工具缺陷"——更准确的说法是近似分析的固有代价。

> **过渡：**第四层里最出名的是符号执行，下一页用一个十行的小程序讲透。

【9.13】示例：符号执行（3 分钟）

> 十行代码，一个十位数，这就是全部。

>

> 方法 CoverMe 收一个整型数组。a 为空就返回。a 的长度大于 0，再判 a 的第 0 个元素是不是 1234567890。是，就抛异常，注释写着 bug。

>

> 难在哪？难在怎么构造出那个输入。a[0] 要正好等于这个十位数，随机试基本试不到。

>

> 符号执行的做法是：不把 a 当具体数组，当符号。沿控制流一条条走路径。

>

> 第一条，a 为空，直接返回。第二条，a 不为空、长度不大于 0，什么也不做。第三条，非空且长度大于 0，再判 a[0] 是否等于那个十位数，为真就抛异常。bug 找到了。

>

> 机制在这儿。每走完一条路径，就把这条路径上条件的取反当新目标。让约束求解器解出一组能行的输入，再拿去跑下一轮。课件在两处标了字，一处是取反后的条件，一处写着结束，所有路径都走完了。

>

> 两个代表作。DART 那篇论文的全称是 directed automated random testing。思路是具体执行加符号约束。还有微软的 Pex，做 .NET 的白盒用例生成。

>

> 它不万能。路径数是指数级的，每个 if 翻一倍。工程上必须做路径约减，只挑有希望到达危险点的那几条。这就是上一层说的精度和成本。

> **提问：**这段代码里，"a 长度大于 0"和"a[0] 等于某个值"，为什么工具必须按顺序判，不能跳着判？

> **预设回答：**

- 答"因为顺序执行"——对，最直接的原因。跳着判会构造出长度不大于 0 却取第 0 个元素的非法输入。

- 答"因为前一个条件是后一个的前提"——说得更准。这叫路径可行性，不可行的路径不能生成用例。

- 答"跳着判更快"——快是快，会产出一堆跑不通的用例，误报率飙升。

- 追问收口：符号执行难的不是解方程，是判断哪些路径可行。这就是为什么要接约束求解器。

> **易错点 / 考点：** ① 核心是符号代替具体值、沿路径收集约束、取反求新输入；② 三个路径分支要能画出来，简答或画图题常考；③ 两个名字：DART、微软 Pex；④ 局限是路径爆炸，靠路径约减缓解；⑤ 别把它和随机测试混了，它是有方向地找输入，不是瞎试。

> **过渡：** 符号执行看的是路径，还有一类技术看的是值从哪来。

【9.14】值依赖分析（2 分钟）

> 空指针难查，难在它不是一句话的事。

>

> 课件给的解法叫值依赖分析。定义分两半。前半句：数据流分析里定值到使用的那种依赖关系，用值依赖图来表现。定值是给变量赋值那一点，使用是读这个变量那一点。定值到使用的连接，就是值赋完之后被哪些地方用了。

>

> 后半句更具体。值依赖图是一种路径敏感的空指针解引用检测方法。它结合数据流分析里的到达定值分析、区间分析、指向分析建起来。

>

> 翻成人话。一个引用空不空，取决于它从哪条路径来。A 分支来的是新建的对象，B 分支来的是空。全混着看分不清，只看一条又漏。值依赖图做的事，是把值的来源和路径条件绑在一起记。这就是路径敏感。为什么这件事关键？空指针的答案不在某一行，在那条路上。

>

> 三样东西合起来才判得准。到达定值，哪些赋值能到达这个使用点。区间分析，取值范围。指向分析，可能指向哪些对象。少一样都不准：只看到达定值，你不知道取值范围，还是判不出危不危险。

>

> 课件引的文献是 Lawrence 那本，2007 年，讲用值状态依赖图做编译优化。这条线最早从编译器里长出来，后来搬到缺陷检测上。

>

> 一句话：查空指针，查的是这个值从哪来、路上经过了什么判断。

> **提问：**同一行 a.b.c()，什么情况下静态工具会报可能空指针，什么情况下它敢不报？

> **预设回答：**

- 答"看 a 有没有判空"——必要但不充分，还要看 a 是不是可能是空赋进来的。

- 答"看这条路径上有没有空的来源"——对了，这就是到达定值的思路：有没有某个空赋值能到达这个使用点。

- 答"工具永远都会报，宁滥勿缺"——这正是很多团队的现状，也是误报的来源。成熟工具会用区间和指向分析来收敛。

- 收口：评估一个空指针检测工具，看两个数，误报率和漏报率。只看检出多少没意义。

> **易错点 / 考点：**① 值依赖图的全称与定义，课件原文可背；② 三种分析的名字：到达定值、区间分析、指向分析；③ 路径敏感要能解释，考虑路径条件，不把不同分支混为一谈；④ 与符号执行的区别：符号执行生成输入去覆盖路径，值依赖分析判断某点上的值危不危险。

> **过渡：**把静态分析用到安全上，就是 SAST。它多做了哪些事？

【9.15】SAST：静态应用程序安全测试（3 分钟）

> SAST 这五个字母，考试里出现得不算少。

>

> 全称 Static Application Security Testing。中文是静态应用安全测试。技术还是刚才那套，目标换了：不看代码规范，看安全漏洞。

>

> 原理先交代。借语言的编译器或者解释器，把高级语言代码转成一种中间代码。再把调用关系、执行环境、上下文分析清楚。转中间代码，是为了让一套引擎统一处理多种语言。

>

> 五类分析是核心。

>

> 语义分析，查不安全的函数、方法使用。禁用算法、会拼字符串的 SQL 接口，看到调用就知道。

>

> 数据流分析，跟踪、记录、分析数据传递过程中产生的安全问题。这条最重要。输入从 HTTP 参数进来，过几层方法拼进 SQL，这条链就是注入。工具要能把这条链画出来。

>

> 控制流分析，看特定时间、特定状态下执行指令的问题。比如权限校验在某个分支被跳过。

>

> 配置分析，查配置文件里的敏感信息和配置缺失。明文密码、忘关的调试开关、允许星号的跨域。

>

> 结构分析，看上下文环境和结构里的问题。比如一个类的公有方法暴露得太宽。

>

> 最后一步：结合多种分析结果，匹配规则库里的漏洞特征，形成报告。报告里要有漏洞的具体代码行数和修复建议。行号和修复建议最值钱。只说你有 SQL 注入，开发不看；说到第 87 行、建议参数化查询，他才会改。

> **提问：** SAST 报出"第 87 行存在 SQL 注入风险"，开发回你"这里是内部调用，参数不可控"。你怎么处理这条告警？

> **预设回答：**

- 答"让他改，工具说了算"——工具是近似推理，会有误报，硬压会引发对抗。

- 答"那就关掉这条规则"——更糟，等于放弃这一类检测。

- 答"复核：确认调用链上参数是否真不可控，把结论记下来"——正确。误报要留痕关闭，不能静默关闭，否则下次同类问题没人再看。

- 追问收口：成熟团队会给告警分级，高危必修、中危复核、低危先记着，误报要有审批。

> **易错点 / 考点：** ① SAST 的英文全称与中文名；② 五类分析要能默写，特别是数据流分析，那是注入类漏洞的根基；③ 报告的两个要素是代码行数加修复建议；④ 判断"SAST 能发现运行时才出现的问题"——错，它不运行程序；⑤ 与 DAST 的区别下一页讲。

> **过渡：** 静态看代码是一路，还有一路是打进去看反应。

【9.16】SAST 工具分类（3 分钟）

> 扫描工具不是一类东西，深浅差得很远。

>

> 课件按技术深度分四档，你以后评估任何扫描工具，都能往里放。

>

> 第一档，正则匹配，代表是 Cobra、Raptor。拿文本规则去匹配源码。最快，规则最好写，也最容易误报。正则不懂语法，字符串里出现一个 select，它也可能报。这一档适合查绝不允许出现的字面量：私钥、内网地址、明文密码。

>

> 第二档，基于语法树，代表是 P3C、Fireline。先把代码解析成语法树，再在树上做规则匹配。好处是认得结构，误报明显下降。P3C 就是大家熟悉的 Java 规约扫描，它查的是写法地不地道。

>

> 第三档，Java 可以基于 class 文件，代表是 FindBugs。它分析字节码，不分析源码。好处是不需要源码也能扫，第三方 jar 也能查。坏处是拿不到注释和原始格式。

>

> 第四档，基于控制流、数据流、函数调用关系。课件说这是多数商业 SAST 工具的做法。它能查注入链，误报控制得最好，代价是慢、贵、配置复杂。

>

> 还有一个加分项。SMT 求解器被集成到一些大型工具里。课件列了一串名字。前面几个是 HOL/Isabelle、ESC/Java2、ACL2、UCLID。后面还有 BLAST、ureka、CUTE、PEX。不必背全，但要懂它们代表一件事：把程序能不能走到坏状态，变成一道数学题。

>

> 四档排下来，精度往上，成本也往上。

> **提问：**为什么"分析字节码"这一档，反而适合做第三方依赖的安全扫描？

> **预设回答：**

- 答"因为字节码跑得快"——速度是次要的，关键在可得性。

- 答"因为拿不到源码也能扫"——对，这是核心。你引用的一堆 jar 包，手里通常只有 class 文件。

- 答"字节码更接近真实执行，所以更准"——也有道理。泛型擦除、编译器优化之后的真实结构，字节码里才看得见。

- 收口：选型先问一句，我的代码以什么形态存在。有源码、有字节码、还是只有二进制，答案直接决定你能用哪一档。

> **易错点 / 考点：**① 四档的名称与代表工具，简答高频，特别是 P3C 属语法树档、FindBugs 属 class 档；② SMT 那一串名字不要求背全，但要知道它的作用是把路径可行性变成约束求解；③ 判断"基于正则的扫描属于 SAST"——是，但精度最低；④ 别把 Fireline 和 FindBugs 搞混，前者按语法树分析源码，后者分析字节码。

> **过渡：**静态是在不跑的前提下找问题，那跑起来再打的那一路叫什么？

【9.17】DAST（2 分钟）

> 先纠一个错。这页的英文注释沿用了上一页，写成静态应用安全测试那个全称。

>

> DAST 的本义是 Dynamic Application Security Testing。中文是动态应用安全测试。课件这里是排版带过来的笔误，按 Dynamic 记。

>

> 什么叫动态？把系统跑起来，从外部发请求，看它怎么反应。不看代码写了什么，只看它暴露了什么行为。这是黑盒视角。

>

> 做法三步：爬、打、判。顺序不能乱，爬不全就打不到，判不准就白打。先爬，把能到的入口和参数摸出来。再打，按漏洞类型发构造过的载荷。最后判，看回显、报错、延时、状态码。

>

> 强项很明确：运行时才存在的问题，只有它看得见。三类典型。

>

> 配置与环境：生产环境开着调试模式，管理接口忘了鉴权。

>

> 认证与会话：Cookie 缺 HttpOnly，会话能被人拿走。

>

> 业务逻辑漏洞：金额改成负数，用别人的订单号查别人的单。这类问题没有危险函数，静态分析无从下手。

>

> 短板也清楚。覆盖面取决于爬虫，爬不到的地方测不到。给不出代码位置，开发拿着报告还得自己找。跑的时候还可能污染数据。

>

> 一句话对比：SAST 看你写了什么，DAST 看你暴露了什么。

> **提问：**一个"越权查看别人订单"的漏洞，为什么静态扫描几乎发现不了？

> **预设回答：**

- 答"因为代码里没有危险函数"——对。它用的全是正常接口、正常查询，只是少了一个"这单是不是你的"校验。

- 答"因为要登录两个账号去试"——这正是动态测试的做法。用 A 的凭证访问 B 的资源，看返回 200 还是 403。

- 答"静态也能扫，看有没有权限注解"——只能扫出忘了加注解这种形式问题，逻辑层面的越权它判断不了。

- 收口：完整答案是静态查写法、动态查行为、人工查逻辑，三层缺一不可。

> **易错点 / 考点：** ① DAST 的英文全称与中文名，这页注释有笔误，按 Dynamic 记；② 三步法是爬、打、判；③ 常考对比题"SAST 与 DAST 的区别"，按是否运行、白盒黑盒、能否给代码位置、覆盖范围四条答；④ DAST 能测业务逻辑漏洞，这是它相对 SAST 的最大优势。

> **过渡：** 静态、动态各有一半真相，于是有了把两者合起来的第三种。

【9.18】DAST、SAST&IAST（3 分钟）

> 三种扫描，摆在一起比，差别就出来了。

>

> SAST 和 DAST 讲过了，重点看 IAST，交互式应用安全测试。

>

> 三句话定位。SAST 是看图纸：编码期就能做，能指出第几行，但不知道运行时会不会真被利用。DAST 是拆房子试：真打一遍，结果可信，但不知道问题在哪。IAST 是给房子装传感器：应用里插一个 agent，一边跑测试一边看数据在方法之间怎么流。既有运行时的真实数据，又能定位到代码。

>

> 所以 IAST 常叫灰盒。误报低，因为看到的是真实执行路径。定位准，能报到方法甚至行号。能嵌进 CI，跑功能测试的时候顺手扫。为什么误报低？它报的是刚才真跑过的那条路，不是猜出来的。

>

> 代价也得说。必须有人去跑，测试没覆盖的代码它也看不见。还要装 agent，对语言和运行时版本有要求。

>

> 三者是时间轴上的三道岗。编码提交时用 SAST，几秒钟给反馈。测试执行时用 IAST，功能测一遍，顺带抓真实数据流上的漏洞。上线前后用 DAST，按攻击者视角再打一遍。为什么排三道？三种技术在时间上互补，

> 越早越便宜，越晚越可信。

>

> 一句话记：SAST 早、IAST 准、DAST 真。

> **提问：**如果团队只能先上一种，你选哪个？说出前提条件。

> **预设回答：**

- 答"SAST，最便宜、最快见效"——团队还没有任何安全扫描时，这是合理起点。

- 答"DAST，结果最可信"——但它需要可运行的环境和较专业的操作，起步门槛高。

- 答"IAST，又准又能进 CI"——技术上最优，但依赖已有的自动化测试覆盖，没测试就没有 IAST。

- 追问收口：答案取决于你手上有什么。有代码没测试先 SAST，有测试有环境再 IAST，要对外交付做验收才 DAST。

> **易错点 / 考点：**① 三种技术的定位词与白盒、黑盒、灰盒的对应；② IAST 的两个前提，必须有测试覆盖、必须能装 agent；③ 简答常问"怎么在一个研发流程里安排三种扫描"，按提交时 SAST、测试时 IAST、上线前 DAST 回答；④ IAST 误报率最低这个结论判断题常考。

> **【思政落点 2】**：三种扫描各有前提，选哪个不看你喜欢哪个名词，看你手上有什么。有代码没测试，先上 SAST；有测试有环境，才谈 IAST；要对外交付做验收，再上 DAST。技术上最优的方案，落到一个具体团队里可能跑不起来。承认条件、按条件做选择，这就是实事求是，也是工程师的分寸。

> **过渡：**静态这条线收尾，下面进入第二块，真正意义上的跑测试。

【9.19】单元测试（1 分钟）

> 回报最高的一类测试，恰恰是学生最不愿意写的。

>

> 单元测试，把测试做到最小可测单元上。它被单列一小节，不是因为难，是因为最容易被跳过。第 5 页点过它的位置：跑得最快、定位最准、维护最省，回报最高。可它不像 UI 脚本那样看得见成果。界面自己跑起来有人鼓掌，一排绿点没人看。

>

> 这一小节三条线。

>

> 第一条，原理。一个单元测试由哪几段组成。很多人只写调用加断言，缺了准备和清理。用例之间互相污染，跑第二遍就红。一段结构写歪了，后面几百条都跟着不可信。

>

> 第二条，框架。xUnit 家族，以 JUnit 为主线。看它怎么用注解和反射，把用例发现、执行、结果收集全自动。再看 JUnit 5 为什么要把结构拆成三层。拆开是用注解给准备、执行、清理各留一个位置，不用你手写一遍。这几件事手写一次两次还行，几百条用例全靠手写，一定会漏。

>

> 第三条，技巧和工具。Mock 怎么替换不可控的依赖。框架、智能生成、Mock、覆盖率，这四类工具各管一段，别指望一个工具全包。为什么要 Mock？依赖一不稳定，测试就不稳定，红了你都分不清怪谁。

>

> 课件这页写的是原理、框架与工具。三条线合起来一句话：单元测试不是顺手写写，它是有结构、有纪律的工程活动。

> **提问：**为什么很多团队的单元测试"跑第二遍就红"？

> **预设回答：**

- 答"因为有随机数据"——一类原因。测试不能依赖随机。

- 答"因为没有清理环境，上一条用例留下的数据影响了下一台"——最常见的原因，正是下一页要讲的环境清理。

- 答"因为依赖了外部服务，别人一改就挂"——也对，这就轮到 Mock 出场。

- 收口：三种原因都指向同一件事，单元测试必须有严格的结构和隔离，不能随手写。

> **易错点 / 考点：**① 定位是最小可测单元、白盒为主；② 回报最高的结论要能给出三个理由；③ 判断"单元测试只能由测试人员写"——错，工业实践中主要由开发写，测试提供方法与评审。

> **过渡：**先看一个单元测试到底由哪几段构成。

【9.20】单元测试（3 分钟）

单独跑是绿的，一起跑就红了。

>

> 这种场面在单元测试里最常见。课件给的结构是四段：初始化环境、环境清理、测试、断言。我按执行顺序重排一遍。先初始化，造出被测对象、备好依赖和数据。再执行测试，调用被测方法。然后断言，比对实际和预期。最后清理，释放资源、回滚数据。

>

> 学生最容易忽略的是首尾两段。为什么重要？因为单元测试是成批跑的。几百条用例在一个进程里依次执行，前一条留下的全局状态没清掉，后一条就无缘无故地红。单独跑能过、一起跑就挂，这个现象叫测试间耦合，是单元测试最常见的假失败来源。

>

> 再说断言。它是单元测试唯一的判据。没有断言的测试不叫测试，叫调用了一遍没报错。而且断言要断在业务结果上。测折扣，就断言折后价 85 元，不是断言没抛异常。没抛异常只证明它跑完了，没证明它算对了。

>

> 业界把这四段总结成 AAA：Arrange、Act、Assert。清理由框架的钩子承担。还有 FIRST 原则：快速、独立、可重复、自验证、及时。这两条可以当检查单用，不过要知道它们是业界惯例，不是课件原文。

>

> 收一句：结构决定它能不能成批跑，断言决定它有没有价值。

提问：一条单元测试单独跑通过，和别的用例一起跑就失败。你先怀疑什么？

预设回答：

- 答"怀疑被测代码有 bug"——先别急。这种单独过、一起挂的形态，八成不是产品缺陷。

- 答"怀疑测试之间共享了状态"——对，第一嫌疑：静态变量、单例、数据库里的残留数据、没关闭的文件句柄。

- 答"怀疑执行顺序"——顺着上一条来的。框架给的顺序不保证稳定，所以用例之间不允许有顺序依赖。

- 追问收口：修法两条。补上清理，或者让每条用例自己造自己的数据，别共用。

> **易错点 / 考点：** ① 四段的名称与顺序，简答高频；② AAA 与 FIRST 是业界惯例，可以引用，但要知道不是课件原文；③ 测试间耦合的识别与修法，案例题常考；④ 判断"有覆盖率就是好测试"——错，覆盖率高但断言弱的测试价值很低。

> **过渡：** 这四段如果每次手写太啰嗦，于是有了框架。

第 9 章讲稿（9.21–9.40，v3）

> 锚点与课件一一对应，不增删页。每页五个标记。口播按四拍：起 → 推 → 收。

【9.21】xUnit 框架（2 分钟）

> 同一个断言，Java 写 assertEquals。Python 写 assert a == b，C++ 是 EXPECT_EQ。写法全不一样。

>

> 可你把它们的骨架摊开看，几乎一模一样。这就是 xUnit 的意思。x 是占位符，语言换成什么，它就叫什么。Java 叫 JUnit，.NET 叫 NUnit。Python 叫 unittest，C++ 叫 GoogleTest，PHP 叫 PHPUnit。

>

> 为什么长得这么像？因为它们回答同一组问题：用例在哪、环境怎么备、结果怎么判。骨架相同，换语言只换写法，不换脑子。

>

> 五个零件先立住。测试用例，一个方法代表一个能独立判定的检查点。断言，判过不过的那一句。测试夹具，运行前要准备、运行后要拆掉的东西。连接、临时文件、mock 对象，都算夹具。测试套件，把一批用例组织成一批。运行器，负责发现、执行、收结果、出报告。

>

> 这五个怎么串起来？运行器先扫，扫到用例就交给夹具备环境。跑完调断言判，判完让夹具拆，最后把结果汇总。你只写了一个 @Test，剩下的都是这条线在动。框架凭什么不用你手写清单？靠注解或命名约定，加反射。它不认识你的业务，只认识标记。

>

> 还有一层常被漏掉。xUnit 让测试成了代码里的一等公民。测试和产品代码同库、同构建，改代码就得改测试。测试不再是能随手扔的附件。

提问：为什么不同语言的 xUnit 框架，结构会这么像？

预设回答：

- 答"因为互相抄"——粗糙，但有一半真相。JUnit 的设计影响了好几代框架。

- 答"因为解决的是同一类问题"——对。核心就三件：发现用例、准备环境、判定结果。

- 答"因为都是 SUnit 的后代"——更准的溯源。这套模式起于 Smalltalk 的 SUnit，Kent Beck 和 Erich Gamma 把它带到了 Java。

- 收口：学框架要学骨架，别背 API 名。骨架稳定，API 会变。

易错点 / 考点：① x 的含义（语言或框架前缀的占位符）；② 五个零件的名称与作用，简答常问"测试夹具是什么"；③ 判断题"xUnit 只能做单元测试"——错，它常被用来组织集成测试、接口测试；④ 自动发现的机制是"注解或命名约定加反射"。

过渡：以 Java 为标本，先看经典款的 JUnit 4 怎么跑起来。

【9.22】JUnit 4 工作原理/框架（2 分钟）

> 按下那个绿色三角之前，你大概没想过它做了什么。它没把你的类当成普通类看。它在找一个东西：Runner。

>

> 第一步发现。方法上标着 @Test，运行器用反射把类里方法扫一遍。有注解就登记成一条用例。这就是零配置的来处——你没写清单，标记就是清单。

>

> 第二步准备。每条用例之前跑 @Before，整类开始之前跑 @BeforeClass。差别在粒度和成本。前者负责每条都要的干净环境。后者整类只跑一次，用来干贵的事，比如起一个内存数据库。

>

> 第三步执行和判定。调方法，用 Assert 断言。失败会抛断言错误、被标红，但整批不会停。剩下用例继续跑，一次拿到全部结果。这是框架最值钱的设计。要是第一条红了就退，你一天只能修一个错。

>

> 第四步清理。@After 在每条之后，@AfterClass 在整类结束。顺序和准备相反。反着来有道理：依赖一层层叠上去，拆得从最上面拆。

>

> 第五步汇总。结果对象交给 IDE 或构建工具，红绿条就是它的可视化。

>

> 参数化运行器、套件运行器、Rule 是什么？都是换个 Runner，或者加一层拦截。理解 Runner，JUnit 4 就通了。

提问：@Before 和 @BeforeClass 你会在什么情况下分别使用？

预设回答：

- 答"随便用，反正都是准备"——不行。一个每条跑，一个整类跑一次。

- 答"@BeforeClass 放贵的，@Before 放便宜的"——抓住了成本这一维。

- 答"@BeforeClass 必须静态，所以只能放和实例无关的准备"——这条很关键。它是类级别的，拿不到实例字段。

- 追问收口：还有一条纪律。@Before 里备的东西，@After 里必须拆得干净，否则就是测试之间互相污染。

易错点 / 考点：① 四个注解的执行次数和顺序，简答、判断都高频；② 断言失败不会中断整批执行；③ 判断题"@BeforeClass 方法可以是实例方法"——错，必须静态；④ 扩展点名词：Runner、Rule、参数化、Suite。

过渡：JUnit 4 是经典款。JUnit 5 做了一次架构级重写，看它为什么这么拆。

【9.23】JUnit 5 架构（3 分钟）

> JUnit 5 装进项目，依赖里会多出好几个包。为什么不是一个？

>

> 因为它被拆成了三段。等式是 JUnit 5 等于 Platform 加 Jupiter 加 Vintage。括号里那个「复古」，说的就是 Vintage。

>

> Platform 是基于 JVM 执行测试的基础框架。它还带着 Test-Engine API、控制台启动器 CLI、CI/CD 支持。另外有基于 JUnit 4 的 Runner。关键词是基础框架。它自己不含任何测试写法，只提供执行能力，外加一个插座。谁来测，插上就行。

>

> Jupiter 是写测试和扩展的新编程模型与扩展模型的组合。它负责跑基于 Jupiter 的测试。你写 JUnit 5 风格的用例，写的就是它。

>

> Vintage 提供 TestEngine。它让基于 JUnit 3 和 JUnit 4 的测试，也能跑在平台上。真实项目里躺着成千上万条老用例。没有它，升级只有两条路：全改，或者全扔。有了它，老用例原地不动也能跑，可以按模块慢慢迁。

>

> 拆成三层图什么？把怎么执行和怎么写分开。分开之后，框架能共存，入口能统一，新引擎能接进来。这也是插件化最朴素的一次示范。先把插座定下来，剩下谁来插都行。

> **提问：**一个项目里 JUnit 4 和 JUnit 5 的测试都有，需要分成两个工程吗？

> **预设回答：**

- 答"需要，版本不兼容"——不必。Vintage 引擎就是为这个场景准备的。

- 答"不需要，Vintage 能让老的跑在 Platform 上"——对，这正是它存在的理由。

- 答"可以混，但注解来自不同的包"——很到位的实操细节。org.junit.Test 和 org.junit.jupiter.api.Test 不是一个东西。

- 收口：拆层的价值是让迁移变成渐进的，不是一次豪赌。

> **易错点 / 考点：** ① 三段等式的名称与职责，本节最高频，要能默写；② Platform 的四个关键词：Test-Engine API、CLI、CI/CD 支持、JUnit 4 Runner；③ Vintage 是兼容旧测试，不是复古风格；④ 判断题"JUnit 5 只能跑 JUnit 5 的测试"——错，靠 Vintage 能跑 3 和 4。

> **【思政落点 3】**：Platform 把"跑测试"做成了一条有统一入口、有硬插座的流水线。背后是一条工程常识：质量检查不能靠自觉，要靠机制。最危险的不是没有测试，是跑没跑、跑没跑过没人知道。把检查做成闸门，提交必过、失败必拦，比开十次动员会管用。守规则不是束缚，是让每个人的努力都算数。

> **过渡：** 单元测试最麻烦的地方常常不是断言，是依赖不听话。

【9.24】Mock 技术（3 分钟）

> 有个方法要读当前时间，判断优惠券过期没有。这条测试怎么写才能不飘？

>

> 先把两个词立住。SUT，System Under Test，被测系统。DOC，Depend-on Component，被依赖组件。数据库、支付网关、短信服务、当前时间，都算 DOC。

>

> 麻烦都出在 DOC 上。它不可控：你改不了系统时间，也不可能真去扣一笔款。它慢：一接数据库，毫秒就成了秒级。它还不确定：网络抖一下测试就红，产品没毛病。

>

> Mock 的做法是拿一个假的 DOC 顶上去。假货要满足两点：行为可编程，调用可验证。收过几次调用、参数是什么，都查得到。于是测试变成一句话：给定假的依赖行为，断言 SUT 的输出和交互。

>

> 课件还给了两个概念。round-trip test，往返测试：走完整的写进去、读出来。看数据转一圈还是不是原样。它验的是序列化、持久化、协议转换有没有丢信息。代价是慢，一般放集成层。

>

> front door，前门：从公共接口驱动 SUT，不碰私有字段。只走前门，测试才不会绑在实现细节上。重构时就不必红。

>

> 三条纪律。能不用就不用。只 mock 边界，外部服务、时间、随机数、文件系统。别过度断言交互，否则你测的是自己预期的调用顺序，不是系统做对了事。

提问：一个方法依赖"当前时间"来判断是否过期，你怎么把它测稳？

> **预设回答：**

- 答"把系统时间改掉"——能行，但很脏。会影响同机别的程序，CI 上还要权限。

- 答"把时间作为参数传进来，测试时传固定值"——最干净的做法。把不可控的依赖变成可注入的输入。

- 答"用 Mock 替换时间提供者"——对，这是只 mock 边界的典型场景。时间属于边界。

- 追问收口：三种做法里，第二种是设计层面的改进，第三种是测试层面的手段。先改设计，再用 Mock 兜。

易错点 / 考点：① SUT、DOC 的英文全称与含义，简答高频；② round-trip test 考的是往返一致性，别混成回归测试；③ front door 强调只用公共接口，判断题常考"可以直接访问私有成员来造状态"——错；④ Mock 与 Stub 的区别：Stub 只给返回值，Mock 还验交互；⑤ 判断题"Mock 越多越好"——错，会变成测假设。

> **过渡：**原理清楚了，看工具箱里都有什么。

【9.25】最常用的单元测试工具（3 分钟）

打开插件市场，测试相关的工具能翻十几页。别一个个记，它们只干四件事。

>

> 第一类，单元测试框架，负责写用例。Java 有 JUnit、TestNG。JUnit 是事实标准。TestNG 在分组、依赖、参数化上更贴测试的用法。C++ 有 GoogleTest。Python 有 pytest、unittest。pytest 是主流，断言直接写 assert。unittest 是标准库自带，胜在零依赖。

>

> 第二类，智能化的用例自动生成工具：EvoSuite、Diffblue Cover。它们治的是没人愿意写测试这个老毛病。

> 给它一个类，它自动生成一批能提覆盖率的用例。定位要讲清：生成的是起点不是终点。机器不懂业务语义，生成的断言常常只保证不抛异常。边界和规则还得人补。

>

> 第三类，Mock 工具，JMockit。Java 生态里能力很强，连静态方法和构造器都能替。

>

> 第四类，覆盖率工具。Java 用 JaCoCo。C/C++ 用 gcov、lcov、gcovr。Python 用 Coverage.py。它们不是测试框架，是度量工具。只告诉你哪些代码被跑过，不告诉你测得对不对。

>

> 四类分工一句话：写用例、造用例、替依赖、量覆盖。工具会换，这四件事不会变。课件那句总结也是按四类分的，考试照四类答，条理最清。

提问：自动生成的单元测试，能直接当验收依据吗？

预设回答：

- 答"能，覆盖率一下就上去了"——覆盖率上去了，不等于测对了。机器不知道业务规则。

- 答"不能，它主要用来兜回归、发现异常"——这个定位最准。自动生成适合做广度覆盖，重构时快速发现行为变化。

- 答"要看工具会不会生成断言"——很专业。只生成调用、不生成断言的工具，价值有限。

- 收口：正确用法是机器铺面、人补点。机器覆盖大量普通路径，人补关键规则和边界。

易错点 / 考点：① 四类工具的分工，简答高频；② 归属别搞错：JMockit 是 Mock 工具，JaCoCo 是覆盖率工具，EvoSuite 和 Diffblue 是生成工具；③ 覆盖率工具不做断言，只做度量；④ 判断题"覆盖率 100% 说明测试充分"——错，只说明代码被执行过。

过渡：单元层往下是接口层。接口自动化的架构比单元复杂，先看它的整体形状。

【9.26】API 自动化测试（1 分钟）

先给个结论：绝大多数团队第一条真正跑起来的流水线，是接口，不是界面。

>

> 三个理由。稳定，接口是契约，改的频率远低于界面。快，一次调用几十毫秒，一次界面操作几秒钟。覆盖主干，库存够不够、优惠能不能叠加、订单状态能不能流转。这些判断全在后端接口里，界面只是把它们显示出来。

>

> 差两个数量级不是玄学。界面操作要等渲染、等网络回来、等动画收尾。接口调用只等一次响应。跑一千条，差距就是几小时和几分钟。界面坏了，多半只是显示错了。接口坏了，才是业务坏了。

>

> 这一小节两条线。一条是架构：API 从哪来、怎么变成定义、怎么组织成场景用例。再往后是怎么触发回归、结果怎么展示。考的是你能不能把这张图画出来。另一条是工具：一类做桩，真接口还没有时先造个假的顶上。一类做测，发请求、断言、出报告。两类别混，它们解决的是不同阶段的问题。

>

> 还有一句要带走。接口自动化最值钱的资产不是脚本，是 API 定义加场景用例。脚本会随框架换代，定义和场景是团队的知识。写脚本的人走了，定义还在。定义写清了，换 Selenium 还是 Rest-Assured，都不影响用例本身。

提问：为什么很多团队做自动化，第一条跑起来的流水线是接口而不是 UI?

预设回答：

- 答"因为接口简单"——简单是一方面。更关键是投入产出：写得快、跑得快、改得少。

- 答"因为前端老改，接口不常改"——对，这是稳定性的直接体现。

- 答"因为接口能测到业务逻辑"——也对。业务主干都在后端。

- 收口：所以业界普遍是先接口、后 UI。这不是偏好，是算过投入产出的。

易错点 / 考点：① 三个优势：稳定、快、覆盖主干；② 判断题"接口测试不需要关注数据"——错，最难的恰恰是测试数据和状态；③ 桩和测是两类工具，后面那页给清单。

过渡：先看这张架构图，它是本节的主干。

【9.27】API 的 TA 架构（复习）（3 分钟）

> 一个老系统，接口文档为零。你要给它补一套接口测试，从哪儿下手？

>

> 课件这张图画了五段。第一段，API 的来源，四条路。文档化工具，平台里已经有定义。手工录入，老接口只能人工补。程序扫描，从代码里扫路由和参数。HAR 导入，把浏览器抓到的一次真实请求导进来。四条路对应四种成熟度，手上有啥用啥。

>

> 第二段，API 定义，三种载体。RAML，RESTful API 描述性语言。Swagger，今天多称 OpenAPI，课件按 Swagger 讲。Markdown，最通用。意义在哪儿？有了机器可读的定义，用例生成和断言校验才谈得上自动。

>

> 第三段，用例。场景设定、选择和组装 API、验证。生产方式三种：手工组装、HAR 批量生成、代码迁移。

>

> 第四段，回归。四种触发：按需回归、定时回归、即时触发、事件触发。这一段是接口自动化接进 CI 的落点。

>

> 第五段，执行和结果展示。报告要能定位到哪个接口、哪一步、期望什么、实际什么。

>

> 连起来一句话：把接口变成定义，把定义变成场景，把场景变成能重复跑的回归。课件底部还注了一句。HAR 是基于 JSON、存 HTTP 请求和响应信息的文件格式。浏览器开发者工具就能导出。

> **提问：**一个老系统，接口没有任何文档，你会从哪条路把定义补出来？

> **预设回答：**

- 答"手工录入"——可行，但慢，而且容易漏参数。

- 答"从代码里扫路由"——效率高，适合框架规整的项目。动态拼出来的路径可能扫不全。

- 答"在浏览器里操作一遍，导出 HAR"——最实用的捷径。真实请求里路径、头、参数、返回全都有。

- 追问收口：工程里通常是组合。先用 HAR 快速铺一批，再用代码扫描补齐，最后人工核对定义。

> **易错点 / 考点：** ① 五段的名称和顺序，画图题、简答都高频；② 回归的四种触发要能默写；③ RAML 和 HAR 的定义，课件原文，判断题常考；④ 三种用例生成方式：手工、HAR 批量、代码迁移；⑤ 别把来源和定义混成一件事——来源是从哪拿信息，定义是用什么格式描述。

> **过渡：** 图看懂了，看有哪些工具能填进这些格子。

【9.28】基于 API 的 TA 工具（2 分钟）

> 后端接口还没写完，前端已经在催联调。这时候要的不是测试，是一个能回话的假后端。

>

> 课件把工具分两组。第一组做桩。Mockbin 能从 HAR 生成一个模拟桩。抓一次真实请求，它就按那个响应回你。RAP 是淘宝开发的 API 管理工具。Swagger 是流行的 API 定义工具、规范。Easy Mock 从 swagger 生成数据。Doclever 编写 REST 接口文档、生成测试数据。Mocky 无需登录，直接生成 response，救急最方便。还有两个偏工程化的：mock-http-server 和 wiremock。后者在 Java 生态里几乎是契约测试标配。

>

> 第二组做测。SoapUI，老牌，图形化，适合入门探查。Rest-Assured，Java 里最好写的接口测试 DSL。它适合进代码库、进 CI。SOAtest，商业，偏企业级。APIfortress，把 API 的安全和质量放一起看。

>

> 两组的分工一句话：桩解决依赖不存在，测解决行为对不对。桩先行，前后端能并行。接口就绪，测接管，做回归。

>

> 选型也别纠结。Java 团队用 Rest-Assured 配 WireMock。要图形化快速上手，就用 SoapUI 或 RAP。别为了一个工具去改技术栈。

> **提问：** 前后端并行开发，前端要联调，后端接口还没写完，你怎么办？

> **预设回答：**

- 答"等后端写完"——进度不允许，而且前端的时间会白等。

- 答"先用 Mock 服务按约定返回假数据"——标准做法。先定契约，桩按契约返回，前端照契约开发。

- 答"用 HAR 导一份真实响应做成桩"——更省事，前提是有可参照的老系统。

- 追问收口：桩的纪律是桩要跟着契约走。契约一改桩就得改，否则联调时两边对不上，白忙。

> **易错点 / 考点：** ① 两组的归属，选择、填空高频：Mockbin、WireMock 属桩，Rest-Assured 属测；② Mockbin 与 HAR 的关系，课件原文；③ RAP 是淘宝出品，Doclever 是文档加测试数据；④ 判断题"有了 Mock 就不需要真实接口测试"——错，桩只能验前端，验不了后端。

> **过渡：** 接口之后是 UI。这一层最直观，也最容易被误判价值。

【9.29】Web UI 自动化测试（1 分钟）

> UI 自动化最容易被高估，也最容易被一刀砍掉。两种态度都不对。

>

> 前面讲过尽量避免 UI 测试。避免当主力，不等于不做。三种情况必须做。第一，关键业务链路的冒烟：登录、下单、支付。每次发布前真点一遍，它是整条链路能不能走通的最终判据。第二，跨浏览器和兼容性。同一个页面在不同浏览器上可能不一样，只有真跑才看得见。第三，前端逻辑复杂的地方：表单联动、富文本、拖拽，接口测不到。

>

> 为什么链路冒烟非它不可？因为单元绿、接口绿，页面照样可能装配错。按钮绑错事件、接口返回了却没渲染出来，只有真点一遍才暴露。跨浏览器那条也省不掉。同一个样式算出来的位置能差几个像素，控件能不能点，各家也不一样。

>

> 这一小节走三步。先看三个基本动作：识别、操作、验证。再看元素的本质是 DOM，理解了它才知道定位在定什么。最后是三套主流路线。WebDriver 远程控制，Cypress 跑在浏览器里，SikuliX 看图点鼠标。

>

> 一条纪律先放着：UI 脚本要少而精，断言要断业务结果。别断像素，别断文案。

>

> 写得多不等于测得深。UI 脚本是最贵的那一类，每一行都要有人长期维护。判断标准不是能不能写，是这件事只有它能干。

提问：为什么 UI 自动化必须做但只能少做？

> **预设回答：**

- 答"因为它最接近用户"——这是必须做的理由。它是整条链路能否走通的最终判据。

- 答"因为它最脆最慢"——这是只能少做的理由。维护成本随界面复杂度往上涨。

- 答"两者都对，关键看覆盖什么"——对。覆盖关键路径和跨浏览器这类只有它才能覆盖的点。

- 收口：判断标准很简单。这件事，接口或单元能不能替代？能，就别放 UI。

易错点 / 考点：① UI 自动化适合的三类场景；② 判断题"UI 自动化越多越好"——错；③ 三条技术路线的名字和差异，后面展开；④ 少而精这条纪律，案例题答到能加分。

过渡：先看那三个基本动作，所有框架都是这三步的变体。

【9.30】Web UI 自动化测试（3 分钟）

一条脚本点了提交按钮，然后什么都没断言。这算测试吗？先放着这个问题，看三步。

>

> 识别 UI 元素、操作 UI 元素、验证状态。Selenium、Cypress、Appium，骨子里都是这三步。

>

> 第一步识别，让脚本知道要操作哪个东西。这是整个 UI 自动化最脆弱的一环。它是脚本和界面唯一的耦合点。识别方式从 id 到 xpath 到图像，稳定性依次往下掉。

>

> 第二步操作：点击、输入、选择、拖拽、滚动、上传。坑很多。元素还没渲染就点，得用显式等待，别用固定睡眠。元素被遮挡。元素在 iframe 里，要切进去。弹窗要等它出现。这些不是 API 问题，是时序问题。

> UI 自动化的失败，一大半是时序问题伪装成的功能缺陷。

>

> 第三步验证，这才是测试的本质。断言要断业务结果，订单状态显示已支付、列表里出现新记录。不要断实现细节：div 的 class 是不是 btn-primary、文案是不是提交。前者稳定，后者一改版就红。

>

> 三步的关系：识别是看得见，操作是够得着，验证是说得清。只识别不操作，那是截图工具。只操作不验证，那是点了一遍没报错。

>

> 还有一条工程纪律：三步要封装。把登录封成一个方法，把定位收进对象库。

> **提问：**一条 UI 脚本点提交按钮，但什么都没断言，它能算测试吗？

> **预设回答：**

- 答"算，至少验证了不报错"——它能证明点击没崩，证明不了提交成功。这是探针，不是测试。

- 答"不算，没有断言就没有判据"——对。测试的定义里，比较预期和实际是核心动作。

- 答"要看有没有后续检查"——如果后面还有"列表里出现新记录"的断言，那它算测试的一部分。

- 追问收口：所以写 UI 脚本的顺序是先想断言，再写操作。先问怎么知道它成了，再问要点哪儿。

> **易错点 / 考点：**① 三步的名称和顺序，本节最高频；② 识别是最脆弱环节这个结论；③ 断言的两类对象，案例题常问"为什么这条脚本一改版就红"；④ 固定睡眠和显式等待的区别，判断题常考，固定睡眠是反模式；⑤ Page Object 是业界模式名，课件在第 57 页写作对象库。

> **过渡：**识别这一步到底在识别什么？答案是 DOM。

【9.31】Web UI 元素（2 分钟）

> 同一条点击脚本，昨天过今天不过，产品没坏。这类事多半出在元素上。

>

> 先把认识立住：页面上你看到的一切，都是 DOM 节点。按钮、输入框、下拉框、表格、链接、图片，拆开

> 看都是树节点。带标签、带属性、带文本。

>

> 这三件东西决定你怎么找、怎么点、怎么验。标签决定选择器的类型，属性决定能不能唯一定位，文本决定断言写哪一句。还有一类东西不是元素，却常被当成元素。canvas 画出来的图、伪元素生成的装饰，它们没有独立节点，抓不到。只能靠坐标或者图像。

>

> 常见类型要能认。文本输入框、单选和复选、下拉选择、按钮、链接。还有表格和列表、文件上传、富文本编辑器、日期选择器、弹层和提示。它们的行为差别很大。下拉框有的能直接设值，有的必须点开再选。日期控件很多时候不让直接输入，必须走弹出日历。

>

> 还有三种麻烦元素，得提前知道。iframe，页面里嵌页面，脚本要先切进去才能找到里面的节点。Shadow DOM，组件封装出来的影子树，常规选择器穿不进去。动态渲染，节点是接口回来之后才生成的，找得太早就找不到。这三样是 UI 脚本时好时坏的主要来源。

>

> 所以那句「昨天能跑今天不能」，答案往往不在产品，在节点。还没生出来、藏在另一层文档里、或者换了结构。

提问：为什么同一条点击按钮的脚本，昨天能跑今天不行，而产品功能是好的？

预设回答：

- 答"产品有 bug"——先排除。这类时好时坏多半不是产品问题。

- 答"因为节点是动态生成的，脚本找得太早"——最常见。要改用显式等待。

- 答"因为页面结构改了，定位器失效"——也常见，特别是定位器绑在自动生成的 class 上。

- 收口：排查顺序是先看节点在不在，再看脚本找得对不对，最后才怀疑功能。

易错点 / 考点：① 元素即节点这个基本认识；② iframe 要切换、Shadow DOM 穿不透，这两个的区别判断题常考；③ 动态渲染和显式等待的关系；④ 别把元素可见等同于可点击——被遮挡、被禁用都点不到。

过渡：既然元素是节点，节点归谁管？归标准管。

【9.32】W3C DOM 标准（3 分钟）

> 浏览器里那个 document，你到底能对它做什么？课件列了一串 API，地址是 dom.spec.whatwg.org。这些不用背，分四组看。

>

> 查找。document.getElementById 按 id 找一个。document.getElementsByTagName 按标签名找一批。

>

> 创建和挂载。document.createElement 造节点。parentNode.appendChild 把节点挂上去。

>

> 读写。element.innerHTML 读写内容。setAttribute 和 getAttribute 读写属性。style.left 改位置。这一组是断言的主战场。验文本看 innerHTML，验属性看 getAttribute。要断言按钮禁用，就看 disabled，别看它灰不灰。

>

> 事件和窗口。element.addEventListener 挂事件，window.onload 等加载。window.scrollTo 滚到某个位置。还有 window.content 和 window.dump。自动化里用得最多的是等页面加载完。

>

> 目的不是背 API。要明白一件事：UI 自动化的一切动作，最后都落到这些标准 API 上。WebDriver 的 click 底下走的是协议。executeScript 让你直接写 JS 调它们。

>

> 会写两行 DOM 操作，等于给自己留了条后路。框架点不到的地方，你能绕进去点。

> **提问：**要断言"这个按钮处于禁用状态"，你要读什么？

> **预设回答：**

- 答"看它是不是灰色的"——视觉判断。颜色会因主题而变，不稳。

- 答"读 disabled 属性"——正确。这才是业务语义所在，和界面样式解耦。

- 答"试着点一下，看有没有反应"——能间接验证，但会产生副作用。有些禁用按钮点了会触发别的逻辑。

- 追问收口：原则是断属性、断文本、断状态，不断样式。样式是实现的皮，属性是行为的面。

> **易错点 / 考点：** ① DOM 的四类操作：查找、创建、读写、事件；② document 和 window 的区别，document 是文档对象，window 是浏览器窗口，判断题常考；③ innerHTML 和 textContent 的差别，前者含标签，后者只要文本；④ executeScript 能直接执行 JS 调这些 API，是 WebDriver 能力不足时的补位手段。

> **过渡：** 会调 API 只是手段，看怎么拿它去测。

【9.33】测试 DOM API（2 分钟）

> 控制台里先敲一句，再写脚本。这个习惯值半小时。

>

> 测试 DOM API 这句话有两层意思，两层都得讲。

>

> 第一层，DOM API 当手段。打开开发者工具的 Console，敲一行。getElementById 取到按钮，后面加 .disabled。结果立刻就出来。这个习惯很值钱。写脚本之前，先把定位器和断言在控制台里验一遍。控制台里拿得到的东西，脚本里才拿得到。控制台里拿不到的，别指望脚本能拿到。很多人写完脚本跑失败才开始猜。一开始验一句，半小时就省下来了。

>

> 第二层，DOM API 当对象。前端代码本身就在操作 DOM：渲染列表、切换状态、显示隐藏。这些操作对不对，就是要测的东西。怎么测？拿断言去比操作前后的 DOM 状态。点展开之后，看某个节点的 class 里有没有多出 expanded。提交失败之后，看错误提示节点在不在、文本对不对。

>

> 一句话收：控制台是你的实验室，DOM 状态是你的断言对象。控制台里验通的那一句，还能直接搬进 executeScript，不用重写。这不是取巧，是把证明过的东西再复用一次。

>

> 顺带一条原则：能用状态断言，就别用像素断言。

> **提问：**要验证"提交失败后出现红色错误提示"，两种断言方式：查节点文本，或截图比对。选哪个？

> **预设回答：**

- 答"截图比对，最直观"——直观但最脆。分辨率、字体、渲染引擎、抗锯齿，任何一样不同像素就不一样。

- 答"查节点文本"——正确。它断的是提示说了什么，不是红到什么程度。

- 答"两个都用，双保险"——听着稳，实际把不稳定因素引进来了。像素断言只在极少数视觉回归场景才值这个代价。

- 收口：判断标准是：这个断言会不会因为跟业务无关的变化而失败？会，就换掉。

> **易错点 / 考点：**① DOM API 的双重身份，手段加对象；② 先控制台、后脚本的排查习惯；③ 状态断言优于像素断言的理由；④ 判断题"截图比对是最可靠的 UI 断言方式"——错，它是最脆的方式之一。

> **过渡：**手段有了，接下来是 UI 自动化的第一道硬功夫：定位。

【9.34】定位元素（3 分钟）

> 登录按钮没有 id，class 是一串自动生成的乱码。你拿什么定位？

>

> 课件把 Selenium 的定位写法列全了。别按字母背，按稳定性排。

>

> 最稳的是 id 和 name：By.id、By.name。id 应当唯一，也不因样式调整而变。name 是表单标配。

>

> 第二档 css：By.cssSelector。它比 xpath 快，写法短，还能表达层级和属性。第三档 xpath：By.xpath，最强也最容易写坏。它能按文本、按轴找。但绝对路径是反模式，加一层 div 就全废。

>

> 第四档 class 和 tagName：By.className、By.tagName。class 往往不唯一，取回是列表。tagName 一般只用来取批量元素。

>

> 专用于链接的：By.linkText 取全部文字。By.partialLinkText 取部分文字。它只能用在 a 标签上。靠可见文字定位，方便，但文字会变。

>

> 最后是 jQuery。executeScript 里 return jQuery.find(...)。前提是页面加载了 jQuery。现在很多工程已经不用，慎用。

>

> 记法：id 优于 name，name 优于 css，css 优于 xpath。xpath 优于文本，文本优于索引。索引排最后。因为第 3 个 div 定的是位置，不是身份。位置天天变，身份不怎么变。

提问：一个登录按钮既没有 id，class 又是自动生成的乱码，你用什么定位？

预设回答：

- 答"用 xpath 按文本找"——可行，但多语言环境下文案一变就废。

- 答"用 css 按层级加属性组合"——更稳。比如用表单的 name 加按钮类型去限定。

- 答"让开发加一个稳定的测试属性"——最专业。可测性是设计出来的，加 data-testid 比任何技巧都稳。

- 追问收口：所以定位不只是测试的事。测试提需求，开发给标识，两边配合才有一劳永逸的方案。

易错点 / 考点：① 九种定位方式的写法，填空、代码题高频；② findElement 和 findElements 的区别；③ 绝对路径 xpath 的危害；④ jQuery 定位的前提是页面加载了 jQuery，而且要经 executeScript 执行；⑤ 稳定性排序 id、name、css、xpath。

过渡：光有方法不够，看一个真实页面上怎么选。

【9.35】UI 元素定位示例（2 分钟）

同一个页面，两个人写出来的定位器能差三个档次。差在哪儿？差在动手之前先问什么。

>

> 第一问，有没有稳定标识？有 id，或者 data-testid 这类专为测试加的属性，直接用。这是唯一不怕改版的定位方式。为什么它最稳？它不参与样式，也不参与布局，改版时没人动它。

>

> 第二问，能不能用语义化属性组合？没有 id，就用 name、type、aria-label 这些有业务含义的属性。组合出唯一选择器。它们改动概率低，因为一改就影响功能本身。举个例子。登录表单按 form 的 name、input 的 type 去限定。再叠上 button 的 type，比一串 div 稳得多。

>

> 第三问，只能用结构吗？那就用就近的稳定祖先加相对路径，别写绝对路径。绕不开文案也行，但要标风险。文案是多语言和迭代的高频变动区，产品最爱改的就是它。

>

> 再给一套排查三步法。看元素在不在：没渲染？在 iframe 里？看唯一性：命中多个，取到错的？看可见性：被遮挡、被禁用、零尺寸，都点不到。

>

> 这三步能分清八成定位失败：不在、不唯一、不可见。剩下那一成才真是选择器写错了。

>

> 定位的功夫不在写表达式。在判断哪个属性最不容易变。

提问：脚本报"元素找不到"，你怎么判断是没渲染还是选择器写错了？

> **预设回答：**

- 答"多跑几次，看是不是偶发"——偶发基本是时序问题，没渲染完；稳定失败更像选择器问题。

- 答"去控制台用同一个选择器查一遍"——最快的判别法。控制台能找到，说明脚本的时机或上下文不对；控制台也找不到，说明选择器写错了。

- 答"加个长 sleep 再看"——能验证猜想，但不能留在脚本里，只是诊断手段。

- 收口：诊断顺序是控制台验选择器、看 iframe 或 Shadow DOM、换显式等待。别一上来就改脚本。

易错点 / 考点：① 定位优先级的四问顺序；② 排查三步：不在、不唯一、不可见；③ data-testid 这类测试专用属性的意义；④ 判断题"用文案定位最稳"——错，多语言和迭代都会让它失效。

> **过渡：**定位是方法层。把这些方法送进浏览器的，是 WebDriver。

【9.36】Selenium + WebDriver（2 分钟）

装过 ChromeDriver 的人不少，能说清它在中间干什么的人不多。脚本和浏览器，中间到底怎么传话？

>

> 答案是一套协议。链路这样走。测试代码调客户端库。客户端库把命令翻译成 HTTP 请求，走 WebDriver 协议。请求发给浏览器驱动，比如 ChromeDriver。驱动通过浏览器的调试接口去操作浏览器。结果沿原路返回。

>

> 这条链路给三个好处。跨语言：Java、Python、C#、JS 说的是同一个协议。跨浏览器：换浏览器只是换驱动，脚本不动。可远程：走 HTTP，于是有了 Grid。为什么一层协议就能换来可远程？HTTP 请求能从任何一台机器发出，浏览器不必和脚本同机。

>

> 家族也说清。早期有 IDE，做录制回放。有 RC，已经废弃。今天是 WebDriver 当家，Grid 做分布式。有人说 Selenium 就是录制，那是十几年前的印象。录制解决看，不解决维护。

>

> 还有一条容易被忽略。既然一切走 HTTP，网络和驱动的稳定性就是脚本稳定性的一部分。驱动版本和浏览器版本不匹配，是最常见的环境类故障。这种红不是代码错了，是环境错了。分不清这两类红，修一天也修不好。驱动还是独立进程，浏览器一升级，它就跟着升。

提问：为什么 Selenium 能做到"换浏览器不改脚本"？

预设回答：

- 答"因为它支持很多浏览器"——这是结果，不是原因。

- 答"因为中间有一层协议，浏览器差异被驱动吃掉了"——正确。脚本只说点这个元素，怎么点由驱动翻译成那个浏览器的方式。

- 答"因为浏览器都遵循同一套标准"——也对一半。现代浏览器都实现 WebDriver 标准，这是 Selenium 4 之后协议统一的基础。

- 收口：所以抽象一层协议，是跨浏览器能力的根源。这也是软件工程里最常用的手法。

易错点 / 考点： ① WebDriver 的链路，客户端库、协议、驱动、浏览器；② 三个好处：跨语言、跨浏览器、可远程；③ 判断题"Selenium IDE 的录制可以当主流开发方式"——错；④ 驱动版本和浏览器版本必须匹配。

过渡： 链路里有个细节值得单独看——那个发 HTTP 的客户端库，自己也是可替换的。

【9.37】Java HTTP Client + Selenium（3 分钟）

POM 里多写一行依赖，就能把 Selenium 的 HTTP 客户端整个换掉。两个坐标的版本都写着 4.6.0。第二个的作用，就是把默认的通信实现换掉。

>

两个 POM 依赖。一个是 selenium-java 4.6.0。另一个是 selenium-http-jdk-client 4.6.0。

>

先说 Selenium 拿 HTTP 和 WebSocket 客户端干什么，四条。向 WebDriver 发送命令。从 Selenium 客户端库向网格发送命令。按网格模式，让各种网格组件互相通信。创建 ChromeDevTools 协议和 BiDi 协议会话。四条说的是同一件事：Selenium 内部所有对外说话，都走同一个通道。

>

现状是什么？目前 Selenium 用 AsyncHttpClient。那是建立在 Netty 之上的开源库，支持异步执行 HTTP 请求和 WebSocket。

>

那为什么还要引 jdk-client？它提供基于 JDK 自带 HttpClient 的实现。好处三条。少一层第三方依赖，不用把 Netty 拖进依赖树。和 JDK 的 TLS、代理、连接池配置天然一致。排查日志更贴近 JDK 那套体系。前提是 JDK 11 起才有 java.net.http。

>

教学价值不在记坐标，在设计思想。把通信抽象成可替换的组件，协议演进时上层 API 不用动。

提问： 把 HTTP 客户端换成 JDK 自带实现，最直接的收益是什么？

预设回答：

- 答"跑得更快"——不一定。性能取决于场景，异步实现往往不慢。

- 答"少一个第三方依赖，依赖树更干净"——最实在的收益，特别是在有依赖合规要求的环境里。

- 答"代理和 TLS 配置跟 JDK 一致，排查方便"——很到位。企业内网环境里更明显。

- 收口：这类替换属于工程整洁度优化，不是性能优化。判断依据是环境约束，不是跑分。

> **易错点 / 考点：** ① 两个依赖的坐标和版本，课件给的是 4.6.0；② 四条用途，简答可考；③ 默认实现是 AsyncHttpClient，基于 Netty；④ 判断题"Selenium 只能用自带的 HTTP 实现"——错，可替换；⑤ 别把 CDP 和 BiDi 混成一件事，两者都是浏览器调试和双向通信协议。

> **过渡：** 一台机器跑得再多也有上限，要并行就得上网格。

【9.38】Selenium Grid（2 分钟）

> 一千条用例，串行跑三个小时。发布窗口就那么点，你只能砍覆盖。

>

> Grid 解决的就是这个：让一批测试同时跑在多台机器、多个浏览器上。

>

> 结构是中心加节点。Hub 收测试请求，按能力分派。若干 Node 各自注册能提供什么：浏览器、版本、操作系统。测试代码声明我要 Chrome 跑在 Linux 上，Hub 就派给满足条件的节点。Hub 自己不跑浏览器，它只做调度。真正开窗口的是 Node。

>

> 好处三个。时间：串行几小时，十路并行几十分钟。覆盖：多浏览器多版本同时跑，兼容性一次暴露。隔离：节点互不干扰，一条把浏览器跑崩也不影响别人。

>

> 但并行不免费，三个坑。数据隔离：两条用例同时改同一个账号的购物车，互相踩。解法是每条用例用自己的数据。环境一致：浏览器版本、分辨率、字体不同，会这台过那台不过。得用容器镜像固定。失败归因：报告一片红，得分清真缺陷、脚本脆、环境问题。

>

> 这三个坑有个共同点。都不是脚本写得对不对，是跑法的问题。所以并行化的第一步不是加机器，是让用例能独立执行。数据自己造，环境自己清，顺序不依赖别人。并行度这个数字，得由数据隔离和环境一致来定。

提问：并行跑之后，通过率从 95% 掉到 80%，你最先查什么？

预设回答：

- 答"产品被改坏了"——先等等。通过率是并行之后才掉的，变量是并行本身。

- 答"查数据冲突，是不是多条用例抢同一份数据"——第一嫌疑，特别涉及登录、下单、改状态的用例。

- 答"查节点环境差异"——第二嫌疑。分辨率、浏览器版本、时区不同都会让脚本行为不同。

- 追问收口：并行化的第一步不是加机器，是让用例可独立执行。

易错点 / 考点：① Hub 和 Node 的职责，以及按能力匹配的分派机制；② 并行的三个收益和三个坑，简答高频；③ 判断题"并行度越高越好"——错，受数据隔离和环境一致性制约；④ 用例必须可独立执行，这是前置条件。

【思政落点 4】：Grid 里最容易被忽视的坑是数据隔离——两条用例抢同一份数据，谁都没错，结果互相踩。这跟多人协作一个道理：自己的用例自己造数据、自己清环境，是不给别人添麻烦的基本功。工程里的规矩常常不写在需求里，写在"别人还要用"这件事上。守边界，团队才敢并行。

过渡：Selenium 走的是远程控制这条路。另有一条完全不同的路：脚本直接跑在浏览器里。

【9.39】基于 Cypress 的 UI 自动化测试（3 分钟）

写 UI 脚本最烦的一件事，是等。有人把这件事整个交给了框架。

>

> Cypress 和 Selenium 的差别，根子在一个词：位置。Selenium 的脚本在浏览器外面，通过协议遥控。Cypress 的脚本跑在浏览器里面，和应用共享同一个运行环境。位置一变，四个结果跟着来。

>

> 第一，不用等。它在浏览器里，能感知请求和渲染什么时候完成。自带自动等待和重试。你写点这个按钮，

> 它会等到元素出现、可点击再点。这一条消灭了 UI 自动化里最大的一类失败：时序问题。

>

> 第二，能看内部。它可以访问应用运行时的状态。可以拦截并伪造网络请求，做桩很自然，不必另起一个 mock 服务。

>

> 第三，反馈快。命令在本地执行，不用每条都往返一次协议。

>

> 第四，限制也来自位置。跑在浏览器里，就受同源和标签页的限制。跨域访问、多标签页协作、多浏览器并行，这些场景它不如 WebDriver 顺手。它的生态以 JavaScript 和 TypeScript 为主。团队不是前端栈，学习成本要考虑。

>

> 选型就清楚了。前端团队、单页应用、要快速反馈，选 Cypress 舒服。要多浏览器矩阵、要跨系统端到端、团队是 Java 或 Python，Selenium 更合适。不是谁更强，是谁更适合你的场景。

> **提问：**Cypress"自动等待"这个能力，为什么能消灭大量假失败?

> **预设回答：**

- 答"因为它跑得快"——速度不是关键。

- 答"因为它在浏览器里，能知道请求和渲染什么时候真完成"——对，这是根因。等待的依据从猜时间变成看状态。

- 答"因为它会重试命令"——补充得对。重试让偶发的时序抖动不至于直接判失败。

- 收口：Selenium 里我们靠显式等待手工补这件事，Cypress 把它做进了框架。这就是框架层面的设计取舍。

> **易错点 / 考点：**① 两条路线的根本差别是脚本位置；② Cypress 的四个特点和两条限制；③ 判断题"Cypress 可以完全替代 Selenium"——错，场景不同；④ 别把自动等待理解成不用管时序，跨系统、异步任务仍要显式等待。

> **过渡：**它的本事从哪来？看架构就明白了。

【9.40】Cypress 架构（2 分钟）

上一节说 Cypress 能自动等待、能拦截请求。这些本事不是魔法，来自它的结构。

>

它由两部分组成。一个是跑在操作系统里的 Node 进程。它负责启动浏览器、起一个代理、把测试文件打包送进浏览器。再收集结果。另一个是跑在浏览器里的运行器。测试代码和应用代码在同一个页面上下文里执行。

>

为什么要多一个代理？浏览器里的脚本没法自己接管全部网络流量。得有一个在浏览器外面、又能看见这些流量的东西。

>

这个双进程结构带来三个能力。

>

第一，网络可控。所有请求都经过那个代理。所以它能拦截、能改写、能伪造响应，做桩不用另起服务。

>

第二，上下文共享。测试代码和应用共享 window。所以能直接读应用的状态，能直接调用应用暴露的方法。断言可以做得非常内部。

>

第三，快。省掉了跨进程通信的开销。命令执行是本地调用，不是发协议。

>

代价也写在结构里。测试和应用同源同上下文，跨域跳转、打开新标签页这类动作会受限。它也做不了多系统串起来的端到端。你去访问另一个域名，就出了自己的地盘。

>

能力从哪来，限制就从哪来。认清了结构，选型就不用背结论。你要的能力它在结构里有，你就能用。结构里没有，配置再多也补不上。

> **提问：**为什么"测试和应用共享同一个 window"既是优点也是限制?

> **预设回答：**

- 答"优点是能读内部状态"——对，断言可以更贴近实现，调试也方便。

- 答"限制是耦合了实现细节"——很关键的洞察。能读内部，就容易被诱惑去断内部，重构时测试就红。

- 答"限制是跨域就不行了"——也对。出了自己的上下文，能力就断了。

- 收口：所以用它要守一条纪律：能用外部行为断言的，就别读内部状态。

> **易错点 / 考点：**① 双进程结构，Node 侧加浏览器侧；② 三个能力的来源：代理、共享上下文、本地调用；③ 能力边界：跨域、多标签；④ 判断题"共享上下文让测试更可靠"——要辩证看，它同时带来耦合风险。

> **过渡：**Web 之外还有一类更土但更通用的办法——直接看图。

【9.41】基于图像识别的 UI 自动化测试（3 分钟）

> 屏幕上只有一个按钮，可你拿不到它。控件树里没有它，XML 里找不到它的 id，DOM 更没有。远程桌面里的老客户端、自绘界面、工控和医疗设备的屏幕、游戏画面，在系统眼里只有一张图。

>

> 图像识别方案就是从这里进场的。思路很直白：你的眼睛怎么找，它就怎么找。先截一张模板图，再在屏幕上做匹配。匹配到就拿到坐标，然后鼠标键盘去点。SikuliX 用 OpenCV 做模板匹配。演进到 SikuliNG 以后，这套 API 被完整实现成 REST-API。背后靠一个服务撑着，那个服务跑在目标机器上。课件那行英文就一句：Implement the API completely as a REST-API。后半句是，backed by a server running on the target machine。这一步为什么管用？因为你不必在目标机上装 Python。任何语言写的脚本，都能远程驱动那台机器。

>

> 好话说完，说代价。它通用，屏幕上看得到就能操作。Web、桌面、移动、嵌入式一视同仁。它也脆，分辨率、主题、字体、缩放，随便变一样匹配就失败。它还没有语义。只认这块像素长得像，不知道这是提交按钮。更不知道按钮此刻是不是禁用的。

>

> 所以工程上就三条。第一，把环境锁死，分辨率、缩放、主题都定下来。第二，只截稳定的局部，别整窗截。

> 第三，把它当兜底，不当主力。它的断言也只能停在“这块区域长得对”。业务对不对，得靠别的手段。

> **提问：**图像识别这套办法，为什么看着最通用，实际最难维护？

> **预设回答：**

- 答“因为界面一改就得重新截图”——最直观的痛。工作量全压在维护上，脚本本身没多少。

- 答“因为它只认图案不认语义”——更关键。它判断不了这个按钮是不是禁用，也判断不了文案对不对。

- 答“因为换了机器、换了分辨率就全废”——很实际，所以第一条工程动作就是锁环境。

- 收口：它的位置是补死角的，不是主力。用对了很救命，用错了很折磨。

> **易错点 / 考点：**① 适用前提是拿不到控件树；② SikuliX 到 SikuliNG 的演进，重点是服务化、REST-API；③ 优缺点对照，通用对脆、有坐标对无语义；④ 判断题“图像识别最通用，应当优先选用”——错，维护成本最高。

> **过渡：**Web 那套搬到手机上，麻烦换了地方。

【9.42】移动 App UI 自动化测试框架（1 分钟）

> 同一份脚本，在你桌上跑得好好的，换一台手机就红了。移动端多出来的麻烦，多半不在写脚本，在别的地方。

>

> 先看三个多。设备多，品牌、尺寸、分辨率都不一样。系统多，Android 版本差异大，厂商 ROM 还各改一层。iOS 的权限模型又是另一套。干扰多，来一个电话、弹一条通知、跳一个更新提示，流程当场断掉。

>

> 同一份脚本在三种手机上跑，结论可能三样。设备多带来的不是工作量翻倍，是排查成本翻倍。你分不清是脚本太脆，还是这台机器本身就慢。所以移动端第一步不是写用例，是把设备拉到能复现的状态。

>

> 再看两条工程约束。真机要管：设备农场、充电、USB 稳定性、并发占用，一样都省不掉。安装包要管：每出一版都得装，还得处理覆盖安装和卸载残留。这两件事跟业务逻辑没关系，可它们吃掉大半工作量。云

> 真机能省一部分硬件钱，省不掉的是脚本里的防御。弱网、来电、切后台，这些在真机上比在模拟器上狠得多。

>

> 课件点了两个名字，代表两条路线。Appium 走控件识别，复用 WebDriver 协议，靠控件树找东西。Airtest 走图像识别，同时能用 Poco 走控件，两条腿走路。选谁，先问一句：这台设备上的控件树，你拿得到吗？

> **提问：**手机上脚本挂了，你怎么区分“产品真有 bug”和“被弹窗打断了”？

> **预设回答：**

- 答“看截图”——最直接。失败时留截图是移动自动化必备的能力。

- 答“看日志里有没有权限弹窗的记录”——对，权限弹窗是最高频的干扰源。

- 答“重跑一次，能过就是偶发”——可以，但要记下偶发率。偶发率高，说明脚本对环境没做防御。

- 收口：所以移动脚本的第一条纪律是先清场再主流程，把弹窗和更新提示做成前置动作。

> **易错点 / 考点：**① 两个框架的名字与路线差异，Appium 控件、Airtest 图像加 Poco；② 移动端三个多；③ 判断题“移动自动化就是把 Web 脚本搬到手机”——错，设备与干扰管理才是主要工作量；④ 失败留截图是必备实践。

> **过渡：**选型之前先看一张表，把几条路线摆在一起。

【9.43】几款常见的测试工具比较（3 分钟）

> 你们组要选工具了，会议室里第一种声音一定是——用最火的那个。

>

> 课件只给了一张对比表，我教你怎么读表。五列。第一列支持范围：Web、移动原生、桌面、还是混合。范围越宽越省心，代价是每一块都不够深。第二列定位方式：控件级，看 id、xpath、控件树；图像级；代码级，直接调页面脚本。定位方式决定稳定性的上限，控件级最稳，图像级最脆。第三列语言与生态：能不能用团队熟悉的语言写，社区活不活跃，出了问题有没有人答。这列最容易被跳过，可它最影响长期成本。第四列并行与设备管理：能不能多机并行，能不能接设备农场，报告能不能汇总。第五列学习与维护成本：上

> 手多久，脚本改造多不多，失败率高不高。

>

> 填完这张表，你会发现一个规律：没有全能冠军，只有最合手的组合。Selenium 覆盖广，可等待和时序要自己管。Cypress 反馈快，但受限于前端场景。Appium 复用 WebDriver 生态，前提是有人管设备。Airtest 连游戏都能测，图像那部分脆。

>

> 选型就五问：被测对象是什么形态？团队用什么语言？要不要多浏览器、多设备矩阵？失败率能不能接受？三年后谁来维护？五问答完，答案自己会浮出来。

> **提问：**团队只有两个人，还要维护三年，你会怎么选？

> **预设回答：**

- 答“选最流行的”——资料多、答案好找，可两个人的团队最怕学不动。

- 答“选团队已经会的”——很务实。人力是最贵的成本，两个没有余量试错。

- 答“选生态稳、文档全、能进 CI 的”——也对。两个人要的是自动化帮人干活，不是人伺候自动化。

- 收口：小团队选型的第一原则是压低总成本，不是追技术先进。

> **易错点 / 考点：**① 比较的五个维度；② 各工具的能力边界，Selenium、Cypress、Appium、Airtest；③ 判断题“工具越新越好”——错，看匹配度与维护成本；④ 选型五问，案例题好用。

> **【思政落点 5】**：选型考的是“因事制宜”四个字。行业里最常见的浪费，是拿最流行的工具去解最不匹配的问题。团队明明是 Java 栈，偏要上 JavaScript 的框架；明明只有 Web，非买一套移动设备农场。工程技术没有绝对先进，只有与场景相称。先看清自己的条件，再决定用什么，别让工具牵着项目走。

> **过渡：**先看 Appium 这条复用 WebDriver 协议的路。

【9.44】Appium（2 分钟）

> 手机上装个东西，让电脑去点它——中间到底过了几道手？

>

> 三道。Appium 的定位一句话就说完：把 WebDriver 那套协议搬到移动端。第一层是客户端。你写 Java、Python、JS，再配一堆能力参数。参数就那几样：平台、设备名、要装的包、自动化引擎。第二层是 Appium 服务端，接命令，按平台分派。第三层是平台驱动，Android 走 UIAutomator2，iOS 走 XCUITest。它负责把 WebDriver 命令翻成系统原生调用。

>

> 好处直接：一套 API 覆盖双端。团队有 Selenium 经验的话，迁过来几乎不用重学。定位是控件级，拿控件树——Android 的视图层级、iOS 的无障碍树。稳定性远高于图像。代价也长在结构里：要装服务端、配驱动、管设备。链路是测试代码到 Appium 服务，再到设备驱动，末了落到真机或模拟器。哪一环不稳都会红，所以通常塞进设备农场或者容器里跑。中间加这一层服务端，看着啰嗦，作用是解耦。客户端只管发命令，换平台不动代码。

>

> 顺带说一句，Appium 管驱动，不管设备。设备分配、安装包版本、并发占用，都得外部调度补上。

> **提问：**为什么说 Appium 稳不稳，主要看开发有没有给控件标识？

> **预设回答：**

- 答“因为它是按控件找元素”——对，找的依据就是控件树上的属性。

- 答“没有 id 就只能用坐标或层级，一改就废”——很实际，兜底方案都不稳。

- 答“因为图像识别更稳”——反了。图像识别更脆。

- 收口：所以移动端也要提可测性要求，关键控件加 content-desc 或 accessibilityIdentifier，比任何脚本技巧都管用。

> **易错点 / 考点：**① 三层名称，客户端、服务端、平台驱动；② 两个驱动名，UIAutomator2 与 XCUITest；③ capabilities 的常见键；④ 判断题“Appium 自带设备管理”——错；⑤ 复用 WebDriver 协议这条，是它与 Selenium 同源的关键。

> **过渡：**结构清楚了，看一个最小脚本长什么样。

【9.45】Appium 示例（2 分钟）

第一个脚本写完，跑起来红了，报的错你多半看不懂。

>

骨架五段，照着写就能跑通第一个 hello world。第一段准备能力参数：告诉服务端在哪个平台、哪台设备、装哪个包、用哪个自动化引擎。这段写错，表现就是连接超时。第二段建会话：客户端拿这批参数向服务端要一个会话。服务端去启设备、装应用、拉起驱动。这段最慢，也最容易失败。设备没连上、包没装成、驱动版本不匹配，全卡在这里。

>

第三段定位与操作：按控件属性找到目标元素，然后点击、输入、滑动。和 Web 一样，优先用稳定标识。第四段断言：断业务结果，列表里出现了新条目、页面标题变了、提示文案对了。别断坐标，别断颜色，那两样明天就变。第五段收尾：关会话、放设备。这段经常被忘。

>

忘了会怎样？服务端会话还挂着，设备被占住，下一个任务排队干等。在共享设备的环境里，这是很常见的莫名超时。所以建会话和关会话必须成对，最好写在 finally 里。另外两条经验顺手记下。等待用显式等待，别用固定睡眠。失败要留证据，截图、页面结构、日志，一样都不能省。入口在官方文档 writing-running-appium 下的 running-tests。

提问：脚本跑完没关会话，会有什么后果？

预设回答：

- 答“没什么后果，进程结束就没了”——不一定。服务端那边的会话可能还挂着，设备仍被占用。

- 答“设备被占住，后面的任务排队或超时”——对，这是共享设备环境里最常见的莫名失败。

- 答“应用没卸干净，下次装包冲突”——也可能，在覆盖安装的场景里。

- 收口：收尾要写进模板，建会话必须配对关会话，放 finally 里保证执行。

易错点 / 考点：① 五段骨架的顺序，代码题常考；② capabilities 的作用；③ 建会话是最易失败的环节；④ 判断题“脚本跑完不用管会话”——错，会导致设备占用；⑤ 显式等待优于固定睡眠。

过渡：Appium 走控件，Airtest 走图像，也能走控件。

【9.46】Airtest 框架（3 分钟）

有团队专门为了测小程序，去装了一整套框架。

>

> 课件一句话定性：支持 Python 语言的测试脚本，由三个开源组件构成。第一个是 Airtest。它是跨平台的、基于图像识别的 UI 自动化测试框架。它提供一系列跨平台 API，覆盖各种常见操作。两个词要抓住，跨平台，图像识别。走的就是前面那条按图找的路。第二个是 Poco。它是基于 UI 控件识别的自动化测试框架，支持 Android、iOS 原生 app 和微信小程序。小程序这一条是差异化，不少团队冲它来。第三个是 AirtestIDE。它是跨平台的 UI 自动化测试编辑器，能快速编写 Airtest 和 Poco 代码。脚本录制、一键回放、报告查看，都在里面。

>

> 三者的关系打个比方。Airtest 是眼睛，负责看。Poco 是手，负责抓控件。AirtestIDE 是工作台，把两者的能力装进一个界面，降低上手门槛。它们可以混用。同一个脚本里，能用控件的地方用 Poco，稳。控件拿不到的地方用图像，通用。这才是它的正确用法。

>

> 入口是 airtest.netease.com。强项是覆盖面，原生 App、小程序、游戏、桌面应用都能上。弱项和图像方案一样，图像那部分脆。所以断言和关键操作尽量压在 Poco 这边。

提问：同一个操作，图像点击和 Poco 定位控件点击都能做，你选哪个？

预设回答：

- 答“图像，直观”——直观，但脆。分辨率与主题一变就失败。

- 答“Poco，因为控件有唯一标识”——对，稳定性优先。

- 答“看情况，控件拿不到才用图像”——最完整的答案。图像是兜底，不是首选。

- 收口：判断标准就一条，你抓的是语义还是像素。

易错点 / 考点：① 三个组件的名称与职责，简答高频；② Poco 支持微信小程序这一点是差异化考点；③ AirtestIDE 的四个功能，编写、录制、回放、报告；④ 判断题“Airtest 只能做图像识别”——错，Poco 走

控件。

> **过渡：**看一段 Poco 的写法，就明白两种办法在代码层差在哪。

【9.47】示例：基于 Poco 开发自动化测试脚本（2 分钟）

脚本里写的东西，和你眼睛看到的东西，是不是一回事？

>

> Poco 的核心是控件树查询。入口是一个 Poco 对象，代表当前界面。你给它一个选择器，控件的名字、类型、或者层级路径，它返回那个控件。然后你调方法：点击、长按、滑动、读属性。

>

> 所以一段 Poco 脚本读起来像“按名字找人，再让他做动作”。找到登录按钮，点它。找到输入框，输入账号。读提示文字那个控件的文本，断言等于预期。没有坐标，没有截图，没有像素。稳就稳在这。定位等不到元素时，它还能等你几秒，这比死等三秒强。

>

> 读属性这句要多说一句。能拿到按钮是不是可点、文本是不是空，断言才落得住。比像素比对结实得多。

>

> 前提得讲清：控件必须有可识别的属性。Android 原生靠 resource-id 或文本。iOS 靠无障碍标识。小程序靠自己的节点结构。一个标识都没给，Poco 只能按层级路径硬猜。那就退化成结构定位，一改版就废。

>

> 还有个工程细节，Poco 要先连上界面。它得初始化与平台对应的实现，Android 一套，iOS 一套，小程序又一套。连不上就查不到任何控件，表现是控件树是空的，而不是报错。新手多半卡在这。

提问：Poco 脚本报“控件树是空的”，你先查什么？

> **预设回答：**

- 答“查选择器写错没有”——选择器错通常报找不到控件，不是树为空。

- 答“查有没有连上目标应用或平台”——对。树为空基本是连接问题：包名不对、应用不在前台、实现没初

始化。

- 答“查应用是不是 WebView 或自绘界面”——很专业的补充。WebView 里的内容不在原生控件树里，得另走一套。

- 收口：排查顺序是先确认连上，再确认在前台，末了才怀疑选择器。

> **易错点 / 考点：** ① Poco 的工作方式是控件树查询，不是图像匹配；② 前提条件是控件有可识别属性；③ 控件树为空这个典型故障的归因；④ 判断题“Poco 靠截图找元素”——错，那是 Airtest 的图像部分。

> **过渡：** 图像方案里有一类能力绕不开，从图里读字。

【9.48】OCR 实现（2 分钟）

> 一张图上的字，机器是怎么读出来的？

>

> 课件点明结构：目前 OCR 的实现一般分成两个步骤，文本检测和文本识别。文本检测，找到图片上文本的位置和区域，也就是文本框，它回答字在哪儿。文本识别，识别每个文本框里的文字，它回答这些字是什么。

>

> 为什么不能跳过检测？整张图直接识别又慢又乱。模型分不清哪块是字、哪块是背景，输出会串成一片。切成小框，识别器只处理一行字，难度骤降，准确率也上来。检测还顺手给了框的坐标。要点哪儿，位置就有了。

>

> UI 自动化关心 OCR，三个场景。画布类界面，图表、地图、游戏画面，控件树里什么都没有，字全在画出来的像素里。验证码和图片验证，文字渲染成图，属性里拿不到。跨技术栈兜底，远程桌面、老客户端、嵌入式屏幕，只要能截屏就能读字。三个场景的共性就一句：字是画出来的，不是控件给的。

>

> 但 OCR 是读的能力，不是测的能力。它把像素变成字符串，判断还得靠断言。它在自动化里的位置是感知层。再补一条经验：识别准不准，跟分辨率、字体、颜色对比、背景干扰关系很大。宁可先截小区域，只截那个数字所在的框，也别整屏丢进去。框越小越干净，识别越稳。

> **提问：**为什么不能跳过文本检测，直接把整张图丢给识别模型？

> **预设回答：**

- 答“因为整张图识别不出来”——不准确。是识别得乱，输出会串行。

- 答“因为要先切小框，识别才准”——对，切框把问题简化成一行字是什么。

- 答“因为要拿到位置，才能去点击”——很实用。检测给出的框坐标，可以直接用来点。

- 收口：所以检测不只是预处理，它同时产出了可供操作的位置信息。

> **易错点 / 考点：**① 两步的名称与各自回答的问题，在哪儿与是什么；② 顺序不能颠倒；③ OCR 在自动化里的定位是感知层；④ 判断题“OCR 可以替代控件定位”——错，它只能拿到文本，拿不到状态。

> **过渡：**工程上这两步怎么串起来？下面看完整流水线。

【9.49】OCR 实现（3 分钟）

> 一堆缩写砸下来，CTC、BLSTM、MSER，第一眼没人看得懂。

>

> 先看检测的两条路。传统路线走多层 MSER 检测，做文本区域特征提取，MSER 是最大稳定极值区域。接着特征过滤，用 CNN 把不像文字的块筛掉。再特征合并，按规则把小块拼成候选文本行。然后文本行过滤，CNN 再筛一遍。末了一层是多层结果合并，NMS 加规则，把相邻及重叠的文本框合起来。

>

> 深度学习路线短得多。AdEAST 检测直接输出文本区域特征图。再做后处理，框解析加 NMS，得到最终文本框。

>

> 识别分三档。常用字识别，DenseNet 加 CTC。繁体字识别，同样 DenseNet 加 CTC。高精度识别，DenseNet 加 BLSTM 加 CTC。差别在要不要上下文。DenseNet 提特征，CTC 免对齐地把特征序列映射成字符序列，不用标出每个字在哪儿。高精度档再加 BLSTM，双向长短时记忆网络。左右上下文一起看，形近字更准，代价是更慢。

>

> 术语记全称：CRNN 是卷积循环神经网络，卷积加循环加 CTC 的统称。MSER 刚说过，最大稳定极值区域。课件还列了一批开源实现。CTPN 加 DenseNet 加 CTC 的中文 OCR 是一种，crnn_self_attetion 和 CRNN_Attention_OCR 也算。做自动化取字，用现成的检测加识别能力就行，别自己训模型。你的价值在取哪块区域、断什么内容。

> **提问：**CTC 在 OCR 里到底解决了什么问题？

> **预设回答：**

- 答“提高准确率”——太笼统，等于没说。

- 答“解决了对齐问题”——正确。不用把每个字符和图像位置一一标注，模型输出序列，CTC 去做映射。

- 答“能处理不定长文本”——也是它的作用，输出长度不必预先固定。

- 收口：所以训练数据只要图加整行文字，不需要每个字的坐标，这是它能量产的前提。

> **易错点 / 考点：**① 检测两条路线与各自步骤，填空高频；② NMS 的作用，去重、合并重叠框；③ 识别三档的差别，在不在 BLSTM；④ 术语全称要能写；⑤ 判断题“OCR 不需要检测”——错，两步分工不同。

> **过渡：**单点的自动化能力到这儿讲完了，往后把它们装进一个框架。

【9.50】自动化测试集成框架（1 分钟）

> 零件都齐了，为什么还要再搭一层？

>

> 前面讲的都是零件：静态分析、单元测试、接口测试、UI 自动化。零件齐了，问题才刚开始。你会撞上四件事：脚本放哪儿、谁来跑、跑完结果给谁看、环境谁来管。四个团队各写各的，末了就是四套互不相通的东西。报告没法汇总，资源没法复用，新人接手要学四遍。

>

> 这四件事各自不解决，会怎样？脚本散在各自电脑里，换个人就跑不起来。没人调度，半夜没人点那个运行按钮。报告格式各家一套，发布前没人说得清到底测了什么。环境靠口头传，下一个人连数据库地址都要问三天。四件事凑一起，测试就成了靠人盯的体力活。

>

> 这一小节按四个问题走。怎么组织脚本，关键字驱动、数据与脚本分离。框架由哪些部件构成，Harness 与 IDE、脚本语言、代理、工具、任务安排、报告。框架要提供哪些服务。设计时抓哪些要点，控制中心、多层次结构、分布式架构、对象库、关键字库、数据共享。末了拿 Robot Framework 当实例，看这套思想长成的样子。

>

> 集成框架不是更高级的工具，它是让自动化能持续的工程结构。工具解决能不能做，结构解决能不能一直做下去。

提问：单元、接口、UI 各用一套工具，最直接的麻烦是什么？

> **预设回答：**

- 答“要学三套东西”——是麻烦，但不是最要命的。

- 答“报告没法汇总，没人说得清这次发布测了什么”——这才最要命，质量结论收不拢。

- 答“资源重复投入，机器和设备各占一套”——也实际，成本直接翻倍。

- 收口：所以集成框架的第一价值不是技术先进，是让质量信息能汇总、能判断。

易错点 / 考点：① 集成框架要解决的四个现实问题；② 它与测试工具的区别，结构对能力；③ 判断题“集成框架必须有商业产品”——错，开源可拼，也能自研；④ 本节四个主题的顺序。

过渡：先看脚本层最经典的两种组织方式，第一种是关键字驱动。

【9.51】Selenium 关键字驱动（2 分钟）

一行行的“打开浏览器、输入、点击”，不看代码也能读懂，怎么做到的？

>

> 一句话，把做什么从代码里抽出来，变成一行行的关键字加参数。表格三列：关键字、对象、数据。打开浏览器配 Chrome。输入作用在用户名框上，值 admin。点击登录按钮。断言提示区的文本是登录成功。四行下来，一条用例写完了。写的人不用会编程，看表格像看一张操作单。

>

> 好处三条。脚本与实现分离，业务人员看得懂表格，改流程改表格，不动引擎。复用，同一个点击关键字被几千条用例共用，改一次全生效。可组合，关键字可以套关键字，登录这个关键字内部由五个基础关键字组成。关键字写得好不好，直接决定表格读起来像流程还是像代码。

>

> 代价三条。引擎复杂，解析器、执行器、对象库、日志，工程量不小。调试难，问题在表格层看不出来，得钻进引擎里查。表达受限，遇到循环、条件、异常，表格会变得很别扭，末了还是写代码。

>

> 所以现实做法是混合：关键帧用表格，稳定又易读；复杂逻辑用代码，灵活。这就是混合驱动这个名字的由来。课件用 Selenium 演示，因为它的 API 天然适合封成动作词。

> **提问：**一条用例里有“登录失败就重试三次”，你放表格里还是代码里？

> **预设回答：**

- 答“表格里，加个重试关键字”——可以，但重试几次、什么条件重试，很快会把表格撑爆。

- 答“代码里，写成自定义关键字”——标准做法。复杂逻辑封在关键字内部，表格层只写登录并确保成功。

- 答“两者都不写，砍掉这个需求”——不现实，重试是真实需求。

- 收口：边界很清楚，表格负责编排，代码负责实现。

易错点 / 考点：① 关键字驱动的三要素，关键字、对象、数据；② 三个优点与三个代价；③ 混合驱动的含义；④ 判断题“关键字驱动不需要写代码”——错，引擎和自定义关键字都是代码。

> **过渡：**把这套思想做成成熟框架的，是 Robot Framework。

【9.52】RobotFramework 关键字驱动（2 分钟）

同一个“登录”，在十个项目里被写了十遍。

>

> Robot Framework 是同一套思想做成的完整框架。组织形式是表格文件，四类区段。Settings 放导入库、

> 变量文件、前后置。Variables 定义变量。Test Cases 写用例，一行一步。Keywords 放用户自定义关键字。一行就是一步，从上往下读，读完就是流程。

>

> 具体长什么样？Settings 里导入 SeleniumLibrary。Test Cases 里写一行 Open Browser，配地址和浏览器名。Keywords 里把那五步登录动作封成一行。用例里只剩一句“登录成功”，五步藏在关键字里。

>

> 它比 Selenium 加自制表格强在三点。关键字分层，库关键字、资源关键字、用户关键字，三层复用。变量与数据管理内建，天然支持一套脚本配多套数据。报告与日志内建，跑完自动出 HTML 报告，每一步的关键字、参数、耗时、截图都在里面。出问题不用重跑，翻报告就能定位。

>

> 定位上有个差别要说清。RF 是通用的、ATDD 的自动化测试框架，易于用表格来组织测试过程和测试数据。ATDD 是验收测试驱动开发，先写验收用例，再写实现。

>

> 代价三条。表格不适合表达复杂逻辑。调试得靠日志。关键字设计不好，会变成关键字地狱。所以 RF 项目成不成，多半取决于关键字库怎么设计，也就是后面那九个设计要点要管的事。

提问：RF 项目最容易在哪里失败？

预设回答：

- 答“安装配置”——一次性的麻烦，不算真问题。

- 答“关键字太多太乱，没人知道要用哪个”——最典型的失败模式，关键字地狱。

- 答“用例写成了一堆底层操作，业务看不懂”——也对，说明抽象层次没设计好。

- 收口：所以 RF 的第一份设计文档应当是关键字分层规范，不是用例。

易错点 / 考点：① 四个区段的名称与作用，简答高频；② 三层关键字复用；③ ATDD 的含义；④ 判断题“RF 只能做 UI 测试”——错，它靠库扩展做接口、数据库、桌面；⑤ 关键字地狱这个风险要能点名。

过渡：关键字驱动的底层，是数据与脚本分开。下面专门讲这一层。

【9.53】分离数据与脚本（3 分钟）

脚本里的 test_001 和 test_002，除了三行数据，剩下的一个字不差。

>

> 课件原话：使用表或数据文件作为测试输入和可验证的输出，以及测试环境设置和控制，不采用硬编码的过程。这句话里有三个宾语。

>

> 第一个是测试输入。用例要的数据不能写死在脚本里。为什么？同一套流程往往要用几十组数据跑，正常的、边界的、异常的。写死就得复制几十份脚本。放进数据文件，脚本一份就够。

>

> 第二个是可验证的输出。这条最容易被漏掉：期望结果也要数据化。很多团队只把输入抽出去，期望结果还写在断言代码里。加一组数据就得改一段代码，等于没分离。对的做法是数据文件里写明这组输入的期望输出是什么，断言只负责比对。

>

> 第三个是测试环境设置和控制。连哪个环境、用哪个账号、开什么开关，都不写死，放配置文件里。一套脚本就能在开发、测试、预发三套环境上跑。

>

> 三样合起来才是数据驱动的完整含义：脚本是引擎，数据是内容，配置是环境。跟关键字驱动的关系也不冲突，常叠加。关键字管步骤怎么表达，数据驱动管数据从哪来。把关键字表格每行的参数换成变量，再从数据文件读值，就是关键字加数据的混合驱动。

提问：把输入抽到 Excel，期望结果还写在断言代码里，算不算做到了数据驱动？

> **预设回答：**

- 答“算，输入已经分离了”——只做了一半。加一组数据还得改代码，耦合还在。

- 答“不算，期望结果也得数据化”——正确，这才是完整的。

- 答“看情况，期望结果简单的可以写死”——可以妥协，但心里有数，那是技术债不是设计。

- 收口：判据很硬，新增一组用例要不要动代码。要动，就没分离干净。

易错点 / 考点： ① 三个分离对象，输入、期望输出、环境；② 数据驱动的判据；③ 与关键字驱动的关系，可叠加成混合驱动；④ 判断题“数据驱动就是把数据放 Excel”——不完整，缺期望输出和环境。

过渡： 数据要落在文件里，那用什么格式？

【9.54】测试数据的管理（3 分钟）

你接手一个项目，打开数据目录，十二个 Excel，名字叫副本、副本 2、最终版。名字里带副本和最终版的数据文件，多半没人敢删。

>

课件先给七种载体。表格类的 Excel 和 CSV。键值的 Property。层级的 XML 和 YAML。接口的 JSON，它是接口测试的事实标准，因为接口本身就是 JSON。数据量大的 Database，适合多用例共享。

>

然后七条要求。处理业务实体与数据，数据要能表达订单、用户这样的业务对象，不是一堆散字段。容易描述特定的需求，看一眼知道这条用例验什么。读写操作容易，要配得上测试框架。容易修改和维护，改数据不用写代码。需要的话创建 DSL，复杂到一定程度就得有领域专用语言。不能局限于输入，期望输出、环境、前置条件也要能表达。容易实现比对，数据要方便和实际结果自动比。

>

七条的重心不一样：前四条讲好写，后三条讲好用。很多团队的数据文件好写不好用。人能看懂，机器比对不了，末了还得人工判断，自动化就断在这儿。

>

还有一条实践：数据要能造，也要能清。数据不是静态资产，会被用例改。管理必须包含准备与回滚，否则并行一跑就互相污染。

提问： 接口测试的数据，你选 Excel 还是 JSON？

预设回答：

- 答“Excel，测试人员熟悉”——熟悉是优点，可嵌套结构在 Excel 里表达得很别扭。

- 答“JSON，和接口格式一致，能直接当请求体”——主流答案，零转换、易比对。

- 答“两个都用，用例清单用 Excel，请求数据用 JSON”——很务实的组合，也是不少团队的现状。

- 收口：选格式的标准是能不能无损表达你的数据结构，能不能被框架直接读写。

> **易错点 / 考点：** ① 七种载体至少列五种；② 七条要求的重心，特别是不能局限于输入、容易比对输出；③ DSL 什么时候需要；④ 判断题“测试数据只需要管输入”——错，期望输出与环境也算数据；⑤ 数据要能造也要能清。

> **过渡：** 脚本、数据都有了，接下来是装它们的容器。

【9.55】TA 框架的构成（3 分钟）

> 测试跑了一整夜，早上你打开电脑，先看哪儿?

>

> 六个部件，按一次测试的生命周期串一遍。脚本语言决定谁能写、生态多大。Harness 负责加载脚本、组织执行、收集结果。IDE 是你写脚本的地方。两个合起来，是开发与调试的工作台。任务安排决定什么时候跑、跑哪些、跑在哪儿，它把用例集合变成执行计划。代理是真正干活的执行器，跑在目标机器上，接任务、启工具、执行脚本、回传结果。工具是能力提供者，Selenium、Appium 这些。报告把结果变成人能判断的东西：通过率、失败分布、耗时、截图。跑视频的团队，还会把录屏挂上去。整夜跑完，报告就是第一现场。

>

> 链路是：脚本到任务安排，到代理，到工具，到 SUT，末了到报告。SUT 就是被测系统，System under test。

>

> 每一环都有输入输出，每一环都可能是瓶颈。脚本烂，后面全白搭。调度不合理，机器要么闲着要么挤爆。报告不清楚，前面的努力没人看得见。出问题时，这一串也是排查顺序。你要知道是哪台机器、哪个代理、哪一步挂的，报告里得答得上来。所以六个部件里，能力集中在中段，能不能用下去往往看两头。

> **提问：** 六个部件里，哪一个最容易被忽略，却最影响日常效率?

> **预设回答：**

- 答“脚本语言”——不太会忽略，选型时就要定。

- 答“报告”——很常见。报告只甩一堆日志，没人看得懂，等于白跑。

- 答“任务安排”——也常被忽略。没有调度，脚本只能手工点，自动化退化成半自动。

- 收口：我的答案是报告和调度这两个非技术部件。它们不产生测试能力，却决定自动化能不能持续被用。

> **易错点 / 考点：** ① 六个部件的名称，英文名也要能写；② 职责别混，Scheduler 管何时跑，Agents 管在哪跑；③ Harness 与 IDE 的区别；④ 判断题“框架就是工具”——错，工具只占六分之一。

> **过渡：** 部件清楚了，再看一个成熟框架要提供哪些服务。

【9.56】TA 框架应提供的服务（3 分钟）

> 八条服务一条不缺，框架还是没人用——这种场面你见过吗？

>

> 八条我分三组，你记组不记条。第一组管资产：测试件的存储与管理。用例、脚本、数据、配置都得有地方放、有版本、能检索。听着平淡，可它决定新人多久上手、三个月后还找不找得到那条用例。

>

> 第二组管干活。测试脚本开发调试，也就是 TIDE。测试机与资源的管理。任务安排。测试执行启动与调度。TIDE 是开发环境。资源管理解决用哪台机器、哪个设备。任务安排和执行调度解决什么时候跑、按什么顺序、并发多少。举个具体的：同一份用例集，今天跑测试环境，明天跑预发环境，环境从哪来，就是资源管理要回答的。

>

> 第三组管结果：系统监控与 Log 收集，测试结果分析，测试报告查询。监控和日志必须与执行同步，事后补是补不出来的。跑崩那一下没抓到，第二天只有一堆猜。

>

> 课件末尾留了省略号，这八条是底线，不是上限。真实框架还会给环境自动部署、用例依赖管理、失败重试、与缺陷系统联动。最容易做成半成品的是结果分析。很多框架能跑、能出报告，但不会分析，不聚类原因，不分环境问题与真缺陷，不给趋势。报告很厚，判断很难。

提问：八条里哪条做不好，会让自动化退化成手工的电子版？

预设回答：

- 答“任务安排”——对。没有调度就得人点，这是半自动的典型症状。

- 答“资源管理”——机器抢不到、环境不稳，失败率直接飙升。

- 答“结果分析”——报告没人看得懂，最有洞察力的一句，数据不等于结论。

- 收口：三条都致命，但顺序有先后，先能自动跑，再跑得稳，末了才谈看得懂。

易错点 / 考点：① 八条服务名称，简答高频；② TIDE 的含义，测试集成开发环境；③ 监控与日志必须与执行同步；④ 判断题“框架只要能执行测试就行”——错；⑤ 资源管理与任务安排的分工。

过渡：知道要提供什么，接下来是设计时抓什么。

【9.57】TA 框架设计要点（3 分钟）

界面改了一个按钮的 class，两百条用例全红。

>

九个关键词，分三组。结构与部署这三个：控制中心，整个框架要有一个统一的大脑，负责任务分派、状态跟踪、结果汇总。多层次结构，用例层、场景层、关键字层、驱动层分开，越往下越技术、越往上越业务。分布式架构，执行能力要能横向扩，多机、多浏览器、多设备并行。

>

协作与契约这三个：通讯问题，控制中心和代理之间怎么通信，消息、心跳、超时、断线重连。这是分布式框架最容易坏的地方。接口定义，各层之间、各组件之间的接口要稳，换一个实现不能牵一片。集成环境，与 CI、缺陷系统、测试管理平台怎么对接。

>

资产复用这三个最影响长期成本：对象库，把元素定位集中管理，界面改了只改一处。关键字库，把业务动作封成可复用关键字，越用越省。数据共享，不同用例、不同层级共用同一份数据定义，别每个脚本抄一份。

>

> 九个要点背后是同一个原则：把易变的集中起来，把稳定的抽象出来。界面会变，定位收进对象库。业务会变，动作收进关键字库。环境会变，配置收进数据层。界面改版是每周的事，元素定位也就那几百个，收在一处才可能一次改完。

提问：为什么对象库能明显压低维护成本？不用它会怎样？

预设回答：

- 答“因为定位只写一次”——对，一处定义、处处引用。

- 答“不用它就得每条用例都写定位，界面一改全废”——准确，这就是改一次界面废两百条脚本的由来。

- 答“有对象库就不用改脚本了”——要修正。界面改了你还是得改对象库，只是改一处，不是改两百处。

- 收口：所以对象库不是免维护，它把维护量从用例数降到了元素数。

易错点 / 考点：① 九个要点名称，按三组作答条理最清；② 对象库、关键字库、数据共享的复用价值；③ 通讯问题是分布式框架的高频故障点；④ 判断题“小项目分层就是自找麻烦”——错，分层是为了隔离变化，可以层少，不能混着写。

【思政落点 6】：对象库、关键字库、命名规范，说白了是在替后来的人着想。图省事把定位写死在三百条用例里，自己这一周很舒服，接手的人改一次界面就得通读三百个文件。工匠精神不在代码写得多漂亮，在于让别人能接着干下去：留清楚的分层，留可查的对象库，留读得懂的命名。这份自觉就是职业素养。

过渡：九个要点落到功能上，就是支持设计、管理、执行三件事。

【9.58】TA 框架支持测试设计/管理/执行（2 分钟）

有人问你，框架到底帮你做了什么？答不上来的人不少。

>

落到三个环节。设计环节，框架要能让用例以可维护的方式被表达。用例与数据分离，步骤与元素分离，期望结果显式化。产物是可执行的测试件，不是文档。有个观念要转过来，在自动化框架里，用例本身就是代码资产。它要被评审、被版本管理、被重构。

>

管理环节管四样。管用例集，哪些属于冒烟、哪些属于全量。管数据，造与清。管资源，机器与设备。管执

> 行历史，哪次跑、跑出什么。目标是让跑什么这件事可配置，不再靠人回忆。冒烟集跑一次十分钟，全量跑三小时，这得写在配置里，不能靠人挑。

>

> 执行环节管调度、并发、重试、失败取证、结果回传。失败取证就三样：截图、日志、页面结构。执行的设计目标只有一个，可重复。同一份代码、同一份数据，跑十次结论得一致。不一致就去查原因，那是脆弱性的信号。

>

> 举个卡住的场景。失败取证只留了一张截图，没留页面结构。隔天再看，你不知道当时弹的是权限框还是更新框。三样证据少一样，排查就得重跑。

>

> 三个环节里，最难自动的是管理，因为它管的是人和机器。设计好，管理轻，执行稳，结果才可信。

> **提问：**同一份脚本连跑三次，一次过两次挂，说明哪个环节有问题？

> **预设回答：**

- 答“执行环节不稳定”——表面看是执行，根因可能在设计，等待写死、用例互相污染。

- 答“设计环节，用例之间有耦合”——很可能，特别是单独过、一起挂的那类。

- 答“管理环节，数据没隔离”——也很可能，并行或连跑时数据被改。

- 收口：不稳定是症状不是原因，按执行、设计、管理的顺序倒着查。

> **易错点 / 考点：**① 三个环节各自的目标，可维护、可配置、可重复；② 用例是代码资产这个观念；③ 失败取证的三样内容；④ 判断题“执行阶段的问题一定在执行代码里”——错，常见根因在设计与管理。

> **过渡：**把框架放大到多种测试类型、多个被测对象，就得统一规划。

【9.59】统一规划:不同的测试类型和对象（3 分钟）

> 功能测试一套工具，性能测试一套工具，报告两套格式。

>

> 课件给的是英文结构图，按一个引擎、四个支撑、三类平台讲。一个引擎是 Driven Script-Execution Module，脚本执行驱动模块。所有测试类型都由它驱动执行，这是统一入口。

>

> 四个支撑。Functional Libraries 功能库，Depend on the script expression to call existed methods in libraries。意思是靠脚本表达式去调库里已有的方法。方法早就写好了，脚本只负责挑和调。Classified Test Data 分类的测试数据，按类型和场景分好，供不同测试复用。

>

> Test Report Generator 报告生成器，接 Test Suite Batch Execution 套件批量执行的结果，出统一格式的报告。Load Test Engine 负载测试引擎，Based on Platform Type，按平台类型划分。往下分 Mobile Engine、Application Engine、Web Engine。

>

> 图上还有个 Composited Tested System，组合的被测系统。它出现两次，说明同一个系统既被功能测试驱动，也被负载测试驱动。两条路径压在一个被测对象上，这才是真实的项目。

提问：为什么统一报告格式比统一测试工具更要紧？

> **预设回答：**

- 答“因为报告是给领导看的”——不全面。报告是给判断用的。
- 答“因为工具各有所长，很难统一，但结论必须能汇总”——正确。工具可以异构，结论必须同构。
- 答“因为报告统一了就能比较不同测试的结果”——很到位，可比性才是统一的价值。
- 收口：所以抓手是资产与格式，不是工具品牌。

易错点 / 考点：① 各英文部件的名称与职责，读图题常考；② 按平台类型的三类引擎，Mobile、Application、Web；③ 统一规划的核心是共用资产；④ 判断题“统一规划就是全公司用一个工具”——错。

> **过渡：**这些部件里有两个最接地气，数据和对象识别。

【9.60】数据管理与对象识别组件（2 分钟）

脚本报的错有两类，一类说找不到元素，一类说数据不对。这两件事不住在同一个地方。

>

> 课件画得简单。一边是 Classified Test Data，分类的测试数据。一边是 Test Object identification，测试对象识别。中间一句关键话，Scripts and Methods need test data or test object or both。这句话点出脚本的两类输入，数据和对象。喂什么进去，对谁动手。三种情况：只要数据、只要对象、两者都要。算一个金额，只要数据。点一个按钮，只要对象。登录后校验余额，两样都要。

>

> 所以框架得有两个独立的管理组件。数据组件负责分类：按业务实体、场景、环境分好，按需取用。它要的是复用与隔离。同一份用户数据能被登录、下单、查询共用，测试数据与生产数据必须隔开。对象识别组件负责定位：集中管理定位方式，对外只暴露逻辑名。定位表达式改在组件里，脚本不用动。

>

> 一个只有对象库的框架会怎样？数据只能写死在脚本里，加一组数据就复制一份脚本。所以两个组件的分工是刚性的：对象解决找得到，数据解决用得上。

提问：框架只有对象库、没有数据管理，会出什么问题？

预设回答：

- 答“数据只能写死在脚本里”——对，于是加一组数据就得复制一份脚本。
- 答“环境和生产数据混在一起，容易误用”——很实际，特别在没有隔离机制的时候。
- 答“不影响，数据放 Excel 就行”——Excel 是载体，可没有分类与取用的接口，复用和隔离还是靠人自觉。
- 收口：分工是刚性的，对象解决找得到，数据解决用得上。

易错点 / 考点：① 两个组件的名称与职责；② 那句英文的三种情况要能各举一例；③ 数据分类的两个目的，复用与隔离；④ 判断题“对象库就是定位表达式的集合”——不完整，它还是对外逻辑名的抽象层。

过渡：部件、服务、要点、组件都过完了。往后再看这套结构怎么落到真实项目里。

第 9 章讲稿（9.61–9.75，v3）

> 锚点与课件一一对应，不增删页。每页五个标记，口播按四拍：起 → 推 → 收。

【9.61】测试报告组件（2 分钟）

报告这东西，跑完自动生成，多数人只看一眼通过率就关掉。这一眼其实不够用。

>

往下拆，一份自动化报告是四个环节。第一个是 Test Plan，测试计划，管要测什么。哪些用例、什么顺序、什么环境、判定标准写在哪，都在这里。报告里那句"应测多少条"，数字就来自它。

>

第二个是 Test Script Execution Log。脚本执行日志，管实际发生了什么。每条用例几点开始几点结束，每一步做了什么，断言结果、异常堆栈、当时的截图。

>

这一层最容易被做烂。日志写得含糊，后面所有统计和归因都是猜。到最后你手上只有一句"失败了"，没人知道失败在哪。

>

第三个是 Test Report Generator，报告生成器，把计划和日志合在一起算。通过率、失败归类、几次之间的趋势，都是它算出来的。它的价值在聚合，不在排版。

>

第四个是 HTML Report Template，模板，决定这些信息怎么摆给人看。同样一堆数据，摆得清不清楚，读的人差十分钟。

>

四段连成一条数据流。计划说应测什么，日志记实际发生什么，生成器算差在哪，模板把差在哪摆明白。

>

> 判断一份报告够不够用，只看一件事：拿着它，能不能定下这次发不发版。

> **提问：**一份自动化报告，够不够支撑"发不发版"这个决定，看哪三样？

> **预设回答：**

- 答"通过率和失败列表"——必要，但不够。光有失败列表，人还得一条条点开看。

- 答"通过率、失败聚类、失败证据"——这是完整答案，正好对上分层呈现。

- 答"执行时间和机器占用"——工程上确实要看，属第四层信息，别拿它当主菜。

- 收口：报告好不好，就看没参与执行的人能不能五分钟后做出发版决定。

> **易错点 / 考点：**① 四个环节的名称和顺序，简答高频；② 生成器的价值在聚合，不在排版；③ 分层呈现的原则，结论、聚类、证据；④ 判断"报告越详细越好"——错，信息过载会让人干脆不读。

> **过渡：**抽象说完，拿两份真实报告摆一起，差别立刻看得出来。

【9.62】实例：自动生成用户化的报告（3 分钟）

> 同一批用例，两份报告摆在桌上。一份三十秒能判断完，另一份看十分钟还抓不到重点。

>

> 课件把这点差别说得很直白。传统那份，可读性很差，不能直观体现整个测试过程。用户化那份，能充分还原整个过程，分析效率提上去，分析难度降下来。

>

> 翻成工程话，传统报告的问题是信息密度错配。它把原始日志整段铺出来，几千行"步骤 12 通过、步骤 13 通过"。真失败夹在中间，没人看得见。人得自己做统计，而人最不擅长在大段文本里数数。

>

> 用户化报告做四件事。第一，结构化，按用例、按模块、按失败类型分组，一眼看出失败集中在支付模块。第二，可视化，通过率、耗时分布、趋势曲线，用眼睛判断。

>

> 第三，可还原。失败那一步的截图、当时的页面结构、请求和响应，都留在原位。第四，可下钻。从总览点

> 到用例，从用例点到步骤，从步骤点到证据。

>

> 这里有个坑，先说明白。自动化的失败，有一半不是产品缺陷。脚本脆、环境问题、数据冲突，全算进失败里。报告只说一句失败，你就得重跑一遍去复现。它把现场留住，你三十秒能判断"这是被弹窗挡住了"。这就是报告对效率的真实贡献。

>

> 所以做报告别比谁功能多，比谁判断快。新人拿到它，不用问人就知道从哪查起，这份报告就成了。

> **提问：**为什么"能还原失败现场"比"能统计通过率"更有价值？

> **预设回答：**

- 答"因为通过率一眼就能看出来"——对，统计是低价值劳动，机器算完就完事。

- 答"因为失败要人判断原因，判断要有证据"——正确，这正是报告的价值所在。

- 答"因为截图看着直观"——直观是手段，目的是支撑归因。

- 收口：所以报告的设计重心是证据链，不是仪表盘。

> **易错点 / 考点：**① 两类报告的差别，背课件那两句原话；② 用户化报告的四个特征；③ 可还原和可下钻不是一回事，一个给证据，一个给路径；④ 判断"报告能看通过率就够"——错。

> **【思政落点 7】**：讲报告能还原现场，讲到的是证据。数据不好看就换一张好看的图，工程里代价极高，因为它把判断建在假象上。把失败原样留住，让接手的人自己看、自己判，这是这行最基本的诚实。

> **过渡：**抽象说完，落到一个真框架上。先看 Robot Framework 的定位。

【9.63】Robot Framework（3 分钟）

> 你们有没有写过那种表格？一行一步，写完不用培训，做业务的人自己也能读懂。

>

> 课件给 Robot Framework 的定性就一句话。通用的、ATDD 的自动化测试框架。它易于用表格来组织测试过程和测试数据。三个词，逐个看。

>

> "通用的"，指它不绑任何一种被测对象。Web、接口、数据库、桌面、移动，全靠扩展库接进来。它提供一套统一的组织方式和执行方式，具体能力外包给库。所以它是框架的框架。

>

> "ATDD 的"，验收测试驱动开发。先写验收用例，再写实现。用例由业务、开发、测试三方一起写，写清楚才算需求达成一致。

>

> 这一层的价值不在技术，在沟通。把"到底要什么"提前吵完，而不是等测出来才发现两边理解不一样。

>

> "用表格组织"，意思是表格里写的是关键字，关键字由库实现。一行一步，表格管编排，库管实现，两件事分开。

>

> 课件旁边还标了 Tools 和 Data。工具通过库引入，数据通过变量和数据文件引入。两样都不写死在用例里，换环境、换数据，用例不动。

>

> 工程价值三条。入口统一，一个命令跑所有类型。报告内建，不用自己写报告模块。扩展容易，写个 Python 库就能接新能力。

>

> 边界也说清。适合流程清晰、步骤可枚举的测试；不适合需要复杂逻辑和算法的测试，那种场景回代码里写更省事。前面讲过的"关键字地狱"不是它不好，是用错了地方。

>

> 所以它不万能，它把"重复的编排"标准化了。这一句想清楚，选不选它就有依据。

提问：它说自己"通用"，可核心是表格。表格怎么做到通用？

> **预设回答：**

- 答"表格能写任何步骤"——表面如此，机制在库。表格里写的是关键字，关键字由库实现。

- 答"靠扩展库把能力接进来，表格只负责编排"——正确，通用性来自编排和实现分离。

- 答"支持 Python，什么都能写"——也对，自定义关键字就是 Python。

- 收口：通用的代价是必须有好库。库不行，框架再通用也做不了事。

> **易错点 / 考点：** ① 三个定性词，通用、ATDD、表格组织；② Tools 和 Data 分离的含义；③ 适用边界，流程清晰对复杂逻辑；④ 判断"RF 只能做 Web UI 测试"——错，靠库覆盖多类型。

> **过渡：** 它的报告长什么样？这是它最常被人夸的一点。

【9.64】RF 报告（2 分钟）

> 跑完一次，报告自动躺在那儿，一行排版代码都不用写。这是它最常被人夸的地方。

>

> 一份报告通常给两类视图。汇总视图：总量、通过率、按套件分组的统计、耗时。打开它，五秒内知道这次跑成什么样。

>

> 详细日志视图：每条用例展开成一步步的关键字调用。每步带参数、耗时、状态，失败那一步高亮，还挂着当时的截图或错误信息。

>

> 两类视图的分工，正好对上上一页说的分层。汇总给判断，日志给归因。

>

> 而且它们是同一份数据的两种展开，不是两份独立文件。所以从汇总点到失败用例，能直接跳到日志的对应位置，不用另开一个文档去找。

>

> 还有一层设计值得学。报告的层级跟关键字的层级是对齐的。你写用例时用了几层关键字，报告里就能展开几层。

>

> 于是关键字设计得好，报告自然好读。业务关键字"登录成功"在报告里显示成一步，而不是底下二十步操作摊开给人看。

>

> 这也说明关键字库不只是可维护性的事，它同时决定可读性。写关键字的粒度，按业务能不能读懂来定，不按技术实现来定。

>

> 还有一条常被跳过的动作：报告要归档。这次的和上一次的摆在一起，趋势才看得出来。半年后有人问"这个模块是一直这么慢，还是最近才慢"，只有归档过的报告答得上来。

提问：为什么说"关键字设计得好，报告自然好读"？

预设回答：

- 答"因为关键字名字起得好"——只到一半，命名清楚是必要条件。

- 答"因为报告层级跟着关键字走，封装得高就显示得简洁"——正确，业务关键字把二十步收成一步。

- 答"因为关键字越多报告越详细"——反了，关键字越细碎，报告越啰嗦。

- 收口：粒度按业务可读来定，不按技术实现来定。

易错点 / 考点：① 两类视图的分工；② 报告层级和关键字层级的对应；③ 归档做趋势对比的实践；④ 判断"RF 报告要自己开发"——错，内建。

过渡：报告能自动生成，靠的是标准库。先看装完就有哪些。

【9.65】RF 自带的标准库（3 分钟）

装完框架，先打开库列表，看手上已经有什么。这一步能省掉后面很多白干的时间。

>

> 标准库的意义很简单：装完就能用，不用额外找库。按用途分四组，你记组就行。

>

> 第一组，测试基本功：BuiltIn。断言、变量操作、日志、循环与条件、失败与跳过。所有用例都会用到，等于这门语言自带的运行时。

>

> 它同时给断言和日志，所以哪怕一个扩展库都不装，你也能写出"输入—断言—记录"的完整测试。

>

> 第二组，数据处理。Collections 管列表和字典，String 管字符串。DateTime 管日期时间的计算和格式转换。测试里大量工作是在造数据、比数据，这三个库撑起这件事。

>

> 第三组，系统交互。OperatingSystem 管文件和目录、环境变量。Process 起子进程、取返回值。准备数据、读配置、清临时文件都要用。

>

> 第四组，界面与外部。Screenshot 截屏取证，XML 做报文解析和校验。Telnet 连远程设备。

>

> 清单地址在 robotframework.org。那个页面里 standard-libraries 一节就是列表。装之前扫一眼，比搜教程快。

>

> 有一处分界要分清。标准库随框架一起发布。Selenium、Appium、Database、Http 这些是扩展库，要单独装。Image 和 Sikuli 也一样。标准库解决基本功，扩展库解决接什么系统。

>

> 搭环境时找错东西，多半是把这两类混成了一类。

提问：一条用例要断言"优惠券三天后过期"，会用哪两个库？

预设回答：

- 答"BuiltIn 就够，直接断言"——断言用它没错，可"三天后"这个时间得先算出来。

- 答"DateTime 算时间，BuiltIn 断言"——完整答案，一个负责造期望，一个负责判等。

- 答"String 拼字符串就行"——拼得出文本，算不准日期，跨月、闰年、时区都会错。

- 收口：测试里的时间永远用日期库算，别手拼字符串。

> **易错点 / 考点：** ① 标准库的分组和代表库名，简答高频；② BuiltIn 的定位，断言加日志加控制流；③ 标准库和扩展库的分界；④ 判断"Database 是标准库"——错，属扩展库。

> **过渡：** 标准库之外还有两层生态，先看插件这一层。

【9.66】RF 插件（2 分钟）

> 编辑器里能不能补全，加个插件的事。可插件在 RF 里不只有这一类。

>

> 第一类，编辑器和 IDE 插件。RF 的脚本是纯文本表格。所以各种编辑器都能靠插件拿到语法高亮、关键字补全。跳到定义、直接运行、看结果也有。

>

> 这一类不改测试能力，改的是写脚本的体验。有没有补全，一行一行敲下来的效率差得很远。

>

> 第二类，运行时扩展，最常见的是 listener。它挂在执行过程上，能在用例开始结束、关键字开始结束这些事件点插入自己的逻辑。

>

> 插进去干什么？打自定义日志、采性能数据、失败时额外截图、把结果推到别的系统。这是框架留的钩子，不改框架就能加能力。

>

> 第三类，报告与集成插件。把结果推到测试管理平台、缺陷系统、CI 看板，或者换一种报告样式。

>

> 三类有个共同点：都不改"测什么"，只改"怎么用"。所以选插件的标准是有没有减少摩擦，不是功能多不多。

>

> 还有一句实用的：插件版本跟着框架版本一起锁。不锁，升级那天你会花半天找一个不兼容的插件。

>

> 工具链这一层到这儿算清楚了。再往下，提供测试能力的才轮到库。这两层别混。混了的结果是装了两天插件，测试能力一点没涨。

> **提问：**为什么 listener 这种扩展方式，比改框架源码更好？

> **预设回答：**

- 答"不用改源码"——对，升级框架时不会被自己的改动拖住。

- 答"能在事件点插逻辑，够灵活"——也对，钩子覆盖了大部分需求。

- 答"性能更好"——不是重点，甚至可能略慢。

- 收口：核心是可升级，加能力不动框架本体。

> **易错点 / 考点：**① 三类插件的划分；② listener 的作用和事件点；③ 插件版本和框架版本的耦合风险；④ 判断"插件能替代扩展库"——错，一个管工具链，一个管测试能力。

> **过渡：**提供测试能力的那一层，下一页过一遍清单。

【9.67】RF 扩展库（3 分钟）

> 框架本身什么都不会。这句话不是贬义，是它的设计。

>

> 课件列了一串：Selenium、AutoIT、Appium、Android。还有 Database、Http、Image、Sikuli。按"管什么"分类，就跟前面几节对上了。

>

> 界面类。Selenium 管 Web UI，Appium 管移动原生。Sikuli 靠图像识别找控件，Image 做图像比对、跑视觉回归。这一组是把 UI 自动化能力接进 RF。

>

> 桌面类：AutoIT 操作 Windows 原生窗口和控件。Web 测不到的系统对话框、"另存为"、证书弹窗，靠它。

>

> 移动类：Android 库通过 adb 直接操作设备，装包、起应用、取日志。它和 Appium 的分工要说清。

>

> Appium 走 UI 控件，断言界面。Android 库走设备命令，管设备状态。一个测界面，一个管设备，不是替代关系。

>

> 数据和接口类。Database 连库做校验和造数，Http 发请求做接口测试。接口层自动化的主力就这两个。

>

> 看清这一圈，才算看清 RF 的架构。它自己不实现任何测试能力，全压在库上。好处是一套编排通吃所有类型，代价是库的质量决定你的上限。库写得烂，你在表格里怎么调都是烂的。

>

> 所以挑库的标准是"这件事在哪一层做"，不是"别人装了我也装"。装得越多，依赖越重，维护越累。

> **提问：**既然装了 Appium，为什么有时还要装 Android 这个库?

> **预设回答：**

- 答"两个都要，一个测界面一个管设备"——正确，Appium 做 UI 断言，Android 库做装包、清数据、取日志。

- 答"因为 Appium 不稳定，拿它兜底"——不是主因，稳定性要用工程手段解决。

- 答"重复了，装一个就够"——看需求。只测界面可以只装 Appium，准备环境收日志还得用设备命令。

- 收口：选库的标准是这件事在哪一层做。UI 的事交给 UI 库，设备的事交给设备库。

> **易错点 / 考点：**① 扩展库和标准库的分界，本节最高频；② 各库对应什么能力；③ Appium 和 Android 库的分工；④ 判断"装得越多越好"——错，依赖膨胀是维护负担。

> **过渡：**网页和 Windows 两套界面混在一个业务里，一套框架怎么管住？下一页讲一体化。

【9.68】Web 和 Windows 一体化自动化测试框架（2 分钟）

一个业务流程，前面在网页里点，中间弹出一个系统的"另存为"，后面又回到桌面客户端。这种系统不少。

>

做法是分层接管。Web 层用 Selenium 管浏览器里的页面。桌面层用 AutoIT 或系统 API 管原生窗口和对话框。中间靠统一的脚本和报告层连起来。

>

Robot Framework 恰好适合做这个连接层。它的表格编排不关心底层是哪个库，只要关键字对得上。所以网页步骤和桌面步骤能写在同一条用例里。

>

三个工程要点得提前处理。第一，焦点和窗口句柄。切换应用时焦点会跑，脚本要显式激活目标窗口。不激活，输入打到别处，你还看不出为什么失败。

>

第二，时序。桌面应用响应慢，弹窗出现的时间不定。要等窗口出现再动手，不能靠固定睡眠。睡短了抢跑，睡长了整批变慢，两头都难受。

>

第三，环境隔离。桌面应用依赖特定机器和本地配置，换台机器就跑不了，这是常态。要用镜像或专用执行机把环境固定住。

>

这三条不难懂。可漏一条，整套框架在别人机器上就跑不起来。

>

所以这种框架的价值不在技术统一，在流程统一。同一份报告里能看到网页步骤和桌面步骤，端到端才立得住。

提问：Selenium 为什么处理不了"另存为"对话框？

预设回答：

- 答"它是系统弹窗，不是网页元素"——正确，它不属于浏览器渲染的 DOM。

- 答"因为它太快了，来不及"——不是速度问题，是能力边界问题。

- 答"用键盘模拟解决"——能绕，但不稳。正规做法是交给桌面层工具。

- 收口：先问这个界面是谁渲染的。浏览器渲染归 Selenium，操作系统渲染归桌面库。

> **易错点 / 考点：** ① 三段式分层，Web、桌面、连接层；② Selenium 的能力边界；③ 三个工程要点，焦点、时序、环境隔离；④ 判断"一个框架只能测一种界面"——错。

> **过渡：** 企业里做框架，往往不从零写，先做一套模板。下一页看这种思路。

【9.69】基于 SpringBoot 的高效模板化自动化测试框架（2 分钟）

新建一个项目，头两天你在干什么？多数时间不是在写业务用例。

>

> 在写那些每条用例都得写的东西。课件里五样。依赖注入与配置加载、日志与断言库、HTTP 客户端与鉴权。还有数据准备与清理、报告与通知。

>

> 依赖注入和配置加载管连环境、读参数。日志和断言库管统一格式和失败信息。HTTP 客户端和鉴权管带 token、处理签名。

>

> 数据准备和清理管造数、回滚。报告和通知管出报告、发消息。这五样是每个项目都在重复的那一层。

>

> 模板化的思路就一句：把它们抽出来做成基础设施，新增用例只填业务部分。用 SpringBoot 的工程结构落这件事。配置走配置文件，公共能力做成 Bean。用例只写调接口加断言。

>

> 好处三条。上手快，模板就是规范，新人照着填。一致性好，日志、断言、报告全项目一个样。易维护，公共能力改一处，全项目生效。

>

> 代价也得说。模板会僵化。覆盖不到的场景，要么改模板，改一下影响所有人；要么绕过去，绕一次多一个例外。例外多了，模板就名存实亡。

>

> 所以模板必须留扩展点。留了口子，特殊情况走口子，主线不乱。

>

> 模板化不是限制自由，是把重复劳动一次解决掉。这一句反过来读也成立：不解决重复劳动，你的时间就一直耗在那五样上。

> **提问：**模板化框架最容易出的问题是什么？

> **预设回答：**

- 答"模板太复杂，新人看不懂"——常见。为了通用，把模板做得过于抽象。

- 答"模板僵化，特殊场景绕不过去"——结构性风险，只能靠扩展点解决。

- 答"模板一改，所有用例都要回归"——也真实，所以要控制模板的变更频率。

- 收口：三条是同一个问题的三面，通用性和灵活性怎么平衡。

> **易错点 / 考点：**① 模板化的核心思想，抽公共、留业务；② 五样公共能力；③ 模板化的代价和扩展点；④ 判断"模板化只适合大团队"——不一定，小团队更缺时间去重复。

> **过渡：**模板说完了，看一条真用例怎么写。用 RF 加 Selenium 测一个页面。

【9.70】RF+Selenium 完成某页面测试（2 分钟）

> 一条用例从打开浏览器到关掉浏览器，写出来是什么样？

>

> 第一步，准备。在 Settings 区导入 Selenium 库，声明浏览器和地址这类变量。用例开始前打开浏览器，设隐式等待。

>

> 第二步，操作。一行一个动作：打开地址、在用户名框输入、在密码框输入、点登录按钮。每行是关键字加定位器加数据，读起来接近自然语言。

>

> 第三步，断言。断业务结果，比如首页显示了当前用户名。别断某个 div 的 class 变了，那是实现细节，改一次样式就红一次。

>

> 第四步，清理。关闭浏览器。这一步要写在用例的后置里，保证失败也会执行。写在正文里，一断言失败就跳过去，进程留在那儿，下次跑抢资源。

>

> 第五步，报告。跑完自动生成，每一步的关键字、耗时、状态都在，失败那一步带截图。

>

> 这条链里有三种自动化。执行，不用手点。数据参数化，账号密码来自变量。结果判定，断言加报告。

>

> 三样齐了才算完整。只有执行没有断言，那是帮忙点鼠标，不叫自动化测试。缺判定最常见，脚本跑完显示通过，一个断言都没写。那种绿灯最贵，它给你假的安全感。

>

> 写的时候先跑通一条，再批量复制。一上手铺二十条，红起来你分不清是脚本问题还是产品问题。

提问：断言为什么放在清理之前，清理为什么又要放后置？

预设回答：

- 答"顺序不能乱"——顺序还在后面，关键是失败也得清理。

- 答"清理放后置，失败时也会执行"——正确，这是跑多少次都干净的关键。

- 答"断言放前面能看到结果"——对，断言要在业务动作之后、关浏览器之前，否则页面没了。

- 收口：一条用例的可靠结构是前置准备、业务动作、断言、后置清理，四段都不能省。

易错点 / 考点：① 一条完整用例的五步结构；② 断言断业务结果，不断样式；③ 清理放后置，保证失败也执

行；④ 判断"断言可以放在关浏览器之后"——错，页面已销毁。

> **过渡：**技术讲完了，把这一章收成几句话。

【9.71】小结（3 分钟）

整章的东西摊在桌上，会不会觉得碎？六条小结就是把碎片收口。

>

> 全 SDLC：自动化不只发生在系统测试阶段。静态分析在编码期，单元测试在提交前。接口测试在集成期，UI 和端到端在验收前。性能和安全放在专项阶段。它贯穿整个生命周期。

>

> 高度集成：脚本、数据、调度、报告、环境装进一套体系。前面讲集成框架那一大段，说的就是这件事。

>

> 跨平台：Web、移动、桌面、嵌入式，一套编排通吃，靠的是扩展库。

>

> 过程透明：谁跑了什么、什么时候跑的、结果如何，全都留痕。报告组件的价值就在这儿。

>

> 可测试性保障：这条最容易漏，也最深。做自动化的过程会反过来逼产品变得可测，加稳定标识、拆开紧耦合、配置外置、逻辑和界面分开。你在测的同时，也在改被测对象。

>

> 需求可执行：ATDD 那层意思。验收用例写成可执行的，需求就不再是模糊的一段话，而是跑得过还是跑不过。

>

> 支持敏捷、CD、DevOps：自动化是持续交付的前提。没有自动化，每天发布就是每天赌一次。

>

> 这六条不是并列的六件事。前四条讲自动化铺到哪儿，后两条讲它反过来改了产品什么。

>

> 六条收成一句很土的话：改完你敢发。这一句立住，前面六条就有意义。

> **提问：**六条里哪一条最容易被团队忽略，却最有长期价值？

> **预设回答：**

- 答"可测试性保障"——我的答案也是这条，它改的是产品本身，收益远超测试。

- 答"过程透明"——也重要，没有透明就没有信任。

- 答"支持敏捷、CD"——这是结果，不是原因。前面几条做到了，它自然成立。

- 收口：你在测的同时，也在把系统改造得更可测，这层收益要提前意识到。

> **易错点 / 考点：**① 六条的名称，简答高频，可当全章总结题；② 可测试性保障的双向含义，测产品加改产品；③ 判断"自动化只服务测试团队"——错，它服务整条交付链。

> **过渡：**最后留三个问题，它们比任何工具都值得想。

【9.72】提问与思考（3 分钟）

> 三道题，一道都不给标准答案。它们考的是你想清楚没有。

>

> 第一问，录制脚本轻松搞定，为什么没成为功能测试自动化脚本开发的主流？

>

> 录制的产物是一次操作的坐标记录。它记的是点了这个位置、输了这个值，不记意图。意图没记下来，就没法维护。挪个位置、加个等待、换个账号，全废。

>

> 而且录制脚本读不懂、没法复用、也不好参数化，天生一次性。它合适的地方是快速探测，拿它摸清陌生系统的操作路径，然后手写成能维护的脚本。

>

> 第二问，开源框架还是自己开发好？三个问题定方向。需求有多特殊，开源能不能覆盖八成。有没有人长期养框架，自研等于养一个团队。差异化在哪，框架本身几乎从不构成竞争力。

>

> 结论通常是先开源、后扩展。开源做骨架，特有部分做成插件或库。

>

> 第三问，大家都重视自动化，为什么效果不好？四条原因。目标错，把覆盖率当目标，而不是抓缺陷、缩短反馈。分层错，全押在 UI 上，下面那层空着。

>

> 维护崩，脚本脆，红灯长期挂着没人管。组织缺，没人负责、没有门禁、不做复盘。

>

> 四条里三条不是技术问题。自动化失败，多数死在工程管理上。

>

> 答辩场上遇到这三题，答案的形状是给判断、给理由、给边界。只给态度，拿不到分。

> **提问：**如果给"自动化效果不好"排一个责任顺序，你把哪条放第一？

> **预设回答：**

- 答"工具不行"——最常见的甩锅方向，可工具几乎从不是根因。

- 答"目标错了，为了覆盖率而自动化"——我会把这条放第一，目标错了后面全错。

- 答"没人维护，脚本烂了"——也对，这是最直接的死因。

- 收口：无论排哪条，都要拿得出证据，失败率多高、维护工时多少、抓了多少缺陷。

易错点 / 考点：① 三题本身就是论述题，答法是给判断、给理由、给边界；② 录制脚本的局限，无意图、不可维护、不可复用；③ 开源对自研的三问；④ 效果不好的四类原因，三条属工程管理。

【思政落点 8】：第三问值得诚实面对。做自动化最常见的失败，不是工具选错，是不肯承认它没起作用：脚本天天红，报告照样发；复盘只怪"环境不稳"，不查自己目标定错。诚实是先承认，再找原因。复盘追的从

来不是谁的责任。

过渡：思考题留给你们，动手的部分在实验里。

【9.73】实验 6 部署自动化测试框架（3 分钟）

实验六要交的东西，不是一堆截图。

>

六步，我按装、配、跑、集成、扩展、交付走一遍。

>

第一步，下载安装 Robot Framework。验收标准很低也很硬：命令行能跑通，能执行一个最小用例。

>

第二步，装 RF 的 GUI 界面。编辑器加运行加看报告，写脚本不必靠记事本。

>

第三步，装第三方库，课件点了四个。SeleniumLibrary 做 Web UI。HTTP RequestsLibrary 做接口。AppiumLibrary 做移动。RESTinstance 是接口的另一种风格，偏 JSON 校验。

>

这一步的验收是四个库都能 import 成功，而且各跑一个最小示例。装上了不代表能用，也要跑一遍才知道。

>

第四步，启动 RF，用内建库完成基本操作，写 RF 自身的关键字驱动脚本。考的是只用 BuiltIn，能不能写出前置、动作、断言、清理四段齐全的用例。

>

第五步，结合实验 3 实现 RF 集成，生成测试报告。这是重点。把前面实验做的测试接进 RF 统一编排，一份报告里全看到。

>

> 第六步，基于 RF 和第三方库完成接口测试脚本，生成报告，断言状态码和业务字段。

>

> 验收看三样。能跑，六步都通。能读，报告让人看得懂。能交，脚本进版本库，有说明文档。

>

> 第三样我最看重。实验报告不是截图堆砌，要写清你遇到的坑和怎么解决的。那才是工程能力的证据。

> **提问：**第三步那四个库，为什么建议一次装完，而不是用到哪个装哪个？

> **预设回答：**

- 答"一次装完省事"——省事，但容易踩依赖冲突，四个库的依赖版本可能互相打架。

- 答"用哪个装哪个，先跑通再逐步加"——更稳，先最小可用，再往外扩。

- 答"一次装完便于验证环境完整"——也有道理，课程实验要求统一环境。

- 收口：两种都行，关键是装完逐个验证 import 和最小示例，别等写用例时才发现库不可用。

> **易错点 / 考点：**① 六步的名称和顺序，实验报告必写；② 四个库各自的用途；③ 验收三样，能跑、能读、能交；④ 判断"实验报告只需贴通过截图"——错，要写清问题和解法。

> **过渡：**想往深走，下面四本书是建议的路径。

【9.74】学习资源推荐（2 分钟）

> 技术书买回来，翻三章就放下了，这事普遍。所以先排序，比先买齐重要。

>

> 第一本，《从零开始学 Selenium 自动化测试》，机械工业出版社，2020。入门用。WebDriver 的常用接口、定位方式、等待机制讲得透。显式等待还没用熟，从这本开始。

>

> 第二本，《Robot Framework 自动化测试精解》，人民邮电出版社，2020.4。这本对本章最直接。关键字设计、库的用法、报告与集成，正好对应前面那一大段。建议边看边做实验六。

>

> 第三本，《接口自动化测试项目实战》，清华大学出版社，2021.11。接口是性价比最高的一层。它给的是项目怎么做，不是接口怎么调。

>

> 第四本，《前端自动化测试框架——Cypress 从入门到精通》。电子工业出版社，2020.4。偏前端栈，或者想弄明白脚本跑在浏览器里这条路线，看它。

>

> 路径这么排。先 Selenium 打通一条 UI 链路，再 RF 学会组织和报告。接着用接口那本补主干，最后按需看 Cypress。

>

> 四本一起买、一起看，多半一本都进不去。书的价值不在读完，在你当下那个问题被解答。

>

> 还有一处口径要守住。版本相关的问题，一律以官方文档为准。JUnit、Selenium、Appium、RF、Airtest 的官方文档，比二手教程准。

>

> 遇到版本疑问，先别翻博客，去官网那一页看。

提问：为什么建议"先打通一条链路，再学框架"，而不是反过来？

预设回答：

- 答"框架太抽象，没跑通过听不懂"——很准。没有具体经验，抽象概念落不了地。

- 答"先学框架更系统"——系统是好事，但容易学完不会用。

- 答"两条路都行，看人"——也对，多数人的有效路径是先窄后宽。

- 收口：学习顺序跟着反馈速度走，能快速看到结果的先学。

易错点 / 考点：① 四本书的书名和大致定位，了解即可；② 学习路径的合理性，先具体后抽象；③ 官方文档优先于二手教程；④ 判断"工具书要通读"——不必，按需查阅效率更高。

> **过渡：**这一章到这儿，最后几句话送给你们。

【9.75】感 谢 聆 听（1 分钟）

> 一年以后，这一章你还记得几句？

>

> 我们从为什么重视自动化，讲到自动化的活放在哪一层。从静态分析讲到单元、接口、UI，从脚本技术讲到集成框架和报告。细节很多，只留三句。

>

> 第一句，自动化是投资，不是采购。买工具不等于有自动化，成本在设计和维护。算不清投入产出的自动化，越做越亏。

>

> 第二句，分层决定成败。最底下是静态分析和单元，中间是接口，最上面才是 UI。倒过来做，早晚推倒重来。

>

> 第三句，自动化的落点是信任。一条用例绿了不算什么。一千条每次提交都绿，才换来那句可以发。让你改完敢发版、敢上线、敢对用户负责。

>

> 这三句都不新，可你走完这一章再回头看，它们每一句背后都压着一堆具体做法。

>

> 三句连起来是一条线：钱花在设计和维护上，功夫下在下面几层，结果落在敢不敢发那一下。三句都对不上，工具再新也只是多一个要维护的东西。

>

> 下一章换一个维度。当软件要长期演进、要被人接手、要反复修改，测试还能怎么帮上忙。带着这三句去听，你会发现后面的内容都在回答同一个问题：质量怎么持续得下去。

>

> 这门课在广东金融学院开，讲稿署名周宇文。今天到这儿，谢谢大家。

> **提问：**这一章里，哪一个概念你觉得自己还没吃透？课后拿它去查一个真实项目。

> **预设回答：**

- 答"投入产出怎么算"——拿你手上的项目，估一次回归的工时，再估脚本维护工时，一对比就清楚了。

- 答"框架怎么选"——把前面那几问对着你们团队的情况答一遍。

- 答"脚本为什么老红"——按识别、操作、验证三步逐段排查。

- 收口：带着一个真问题去查，比再读一遍课件有用得多。

> **易错点 / 考点：**① 三句话是本章收束，论述题可当结论段；② 复习按分层、分类、框架、报告四条主线串；③ 别把本章记成工具清单，考点在判断依据上。

> **过渡：**本章结束，进入下一章。

四、思政落点清单（共 8 处）

序号	所在页	落点主题
1	9.5	分层这一段算的是回报，落的是务实。行业里最常见的错，是把"先进"当"正确"：听说
2	9.18	三种扫描各有前提，选哪个不看你喜欢哪个名词，看你手上有什么。有代码没测试，先上 S
3	9.23	Platform 把"跑测试"做成了一条有统一入口、有硬插座的流水线。背后是一条工
4	9.38	Grid 里最容易被忽视的坑是数据隔离——两条用例抢同一份数据，谁都没错，结果互相
5	9.43	选型考的是“因事制宜”四个字。行业里最常见的浪费，是拿最流行的工具去解最不匹配的
6	9.57	对象库、关键字库、命名规范，说白了是在替后来的人着想。图省事把定位写死在三百条用
7	9.62	讲报告能还原现场，讲到的是证据。数据不好看就换一张好看的图，工程里代价极高，因为
8	9.72	第三问值得诚实面对。做自动化最常见的失败，不是工具选错，是不肯承认它没起作用：脚