《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）

> 各位同学，我们进入第 9 章——**测试自动化及其框架**。
>
> 上一章讲本地化测试，问的是"换一种语言环境还能不能用"；这一章问的是"同一批测试，能不能不靠人一次一次地点"。它表面是工具问题，实质是**成本问题和纪律问题**。
>
> 一句话结论：自动化不是把手工用例翻译成代码，而是**把测试变成可以反复执行、可以被机器判定的资产**。
>
> 这一章走四块：内涵与边界 → **数据驱动、关键字驱动** → **xUnit** → **API 与 Web／移动 UI 框架**，最后收在集成框架上。
>
> 带着一个问题听：**如果要求每天跑一遍全量回归，你怎么排班？**

> **板书：**中间写"第 9 章 测试自动化及其框架"；左侧写"上一章：换环境还行不行"；右侧写"本章：换人不换

质量——让机器替人重复"；下方四块：内涵与边界 → DD／KD 脚本技术 → xUnit → API／UI 框架。

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

预设回答：

- 答"加人"——三个人三天，每天一遍要九个全职人力，成本先崩掉；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、其余按版本周期手工"——这是最接近工程实际的答案，第 5 页给分层策略时会给出依据。

- 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"，两个词在本章有宽窄之分，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，本章会用大量篇幅讲边界；③ 本章四块的名称与顺序要记牢，简答题常按这个结构给分。

过渡：先把这一章的路线摊开，你才知道每一页站在整章的什么位置上。

【9.2】自动化测试的内涵与原理（1 分钟）

这是本章路线图，一分钟记结构。

第一块，**内涵与原理**：为什么重视它、边界在哪、能分几类。这块最像务虚，却决定后面所有技术选择的方向——方向错了，工具越熟、浪费越大。

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。

第三块，**xUnit**：以 JUnit 为主线看单元框架的骨架——怎么发现用例、准备环境、断言、清理、出报告。

第四块，**API 与 Web／移动 UI 框架**，最后收在**集成框架**：自动化不止一类、不止一个团队时，怎么装进同一套调度与报告。

这条线是"想清楚 → 做出来 → 管起来"，判断力在前、工具在后，顺序别倒。

> **板书：**竖排四块并编号：① 内涵与原理（为什么・边界・分类）→ ② 数据驱动／关键字驱动（脚本组织）→ ③ xUnit（单元框架骨架）→ ④ API／Web／移动 UI 框架 → 集成框架（调度与报告）。右侧一行："判断力在前，工具在后"。

> **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，工具的更替最快，你今年学的库明年可能改名、换协议。

- 答"内涵与原理"——对。判断"该不该自动化、自动化到哪一层"的能力，换了任何工具都用得上。

- 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：所以我讲课的重心会放在"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／SoapUI、UI 对 Selenium／Appium；③ 别把"脚本技术"和"框架"混为一谈：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）

> 这一页给的是调查数据，我念一遍，然后带你读它。
>
> 课件说，根据最近几年的调查，**自动化测试是大家最为关注的**；自动化测试做到高水平的——也就是覆盖率达到 90% 以上的团队——从之前的 **4% 提高到 9%**；大部分已经实现自动化测试的，也就是超过 50% 水平的，从 **32% 提高到 43.8%**，这是 2021 年的数字。
>
> 三个数字放一起看，能读出三层意思。
>
> 第一层，**方向是确定的**。自动化水平在往上走，而且过半的团队已经不是"试水"，是"大部分用例都跑自动化"了。这说明它不再是少数团队的技术尝鲜，而是行业的基本配置。
>
> 第二层，**高水平依然是少数**。9% 这个数字很关键：它意味着即使到了 2021 年，做到 90% 以上自动化的团队仍不足十分之一。为什么这么难？不是工具难装，是**维护难**——脚本写了一千条，改一次界面废掉两百条，谁都不敢再投入。

第三层，**别拿别人的数字当自己的 KPI**。如果我把"覆盖率 90%"写进你的考核，最省事的做法是什么？写一堆只断言"页面能打开"的空脚本，覆盖率数字立刻好看，缺陷一个都没抓住。数字是可以刷的，**能抓住缺陷的自动化才有意义**。

所以这一页真正想告诉你的不是"要重视自动化"，而是"**重视自动化，但要看它抓不抓得住缺陷**"。

板书：三个数字：90%+ 从 4% → 9%；50%+ 从 32% → 43.8%（2021 年调查）。右侧三行解读：方向确定｜高水平仍稀缺（难在维护）｜别把覆盖率当 KPI（可刷）。

提问：如果一个团队报"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"——用例条数不等于价值，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"——这就问到点子上了，能抓到缺陷的脚本才是资产。

- 答"问维护成本，一天要花多少时间修脚本"——非常关键。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，本质上只是"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→9、32→43.8）；② 数据出自调查、年份是 2021 年，答题时写"据课件调查数据"，不要自己加年份或换数字；③ 概念题高频点："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——招聘市场。

【9.4】自动化测试越来越受重视 – 人才市场（2 分钟）

这一页只有一句话：**自动化测试第一**。意思是，在测试类岗位的能力要求里，自动化排在第一需求位。这句话信息量其实很大，我拆成三层讲。

第一层，**岗位在变形**。过去测试岗的典型描述是"能设计用例、能执行、能提缺陷"，今天你去翻招聘要求，几乎都会写"熟悉至少一种自动化框架""有脚本开发能力""了解 CI/CD"。这不是用人单位变挑剔了，是因为**纯手工回归的经济性已经撑不住了**——版本越来越快，一轮回归的时间预算被压缩到小时级。

第二层，**能力模型变成了三件套**：**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）。这三样缺一样，你在团队里就只能干"跑脚本"的活，干不了"让脚本跑得住"的活。

第三层，也是我要提醒的：**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取代；但"这个场景该不该自动化""这条断言能不能发现回归""这个失败是真缺陷还是脚本脆"——这些判断力，是你身上真正不会折旧的部分。所以这门课我不会只教你调 API，我要教你怎么判断。

一句话：市场要的是"**会写、会想、会兜住**"的测试工程师。

> **板书：**标题写"自动化测试第一"；下方三件套：编程能力 → 框架能力 → 工程能力（版本／环境／流水线／报告）；右侧一行："工具会过时，判断力不折旧"。

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"差在会不会判断该不该自动化"——这一层最值钱，也是面试官最想听到的。

- 收口：所以你看，这三层正好对应本章的四块内容：判断力对应第一块，框架对应第四块，脚本组织对应第二、三块。

易错点 / 考点：① "自动化测试第一"是课件对人才市场需求的判断，简答可直接引用；② 不要把它误解成"手工测试没用了"——下一章的分层策略会说，**手工 E2E 要适当加强**；③ 面试类案例题常问"你如何评估一个候选人的自动化能力"，按三件套回答即可。

过渡：既然重要，那自动化到底该往哪一层投？下一页给出本章最重要的一个判断。

【9.5】自动化测试分层策略（3 分钟）

这一页是本章的总纲，我讲慢一点。

第一句，**单元测试更容易实现自动化测试，RoI 也更高，收益显著**。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，砸在单元层能换来最多的"每次提交都被检查一遍"。

第二句，**API 自动化测试也基本能做到 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，UI 只是把按钮摆出来。

第三句，**适当加强手工 E2E 测试（业务层）**。"适当"和"加强"要一起读：端到端要保留、要做扎实，但别指望它承担全量回归——它贵、慢、脆，适合覆盖关键业务链路和新功能验收。

第四句是概括：**以底层测试、接口测试、功能逻辑测试等为主，尽量避免 UI 测试**。避免不是禁止，是别把它当主力。

课件还给了四个好处：**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。界把这个形状叫**测试金字塔**，底盘大、塔尖小；倒过来就是反模式。

板书：画三角形：底层"单元（多、快、稳、RoI 最高）"→ 中层"API（基本 100%）"→ 塔尖"UI／E2E（少而精，手工适当加强）"。右侧四个好处：缺陷易定位・效率高・反映真实需求・更接近业务。旁边写"倒三角＝冰淇淋甜筒＝反模式"。

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会遇到什么？

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"——这是最典型的错答。接近用户是价值判断，但**成本与稳定性**同样要算；UI 层应该是"少量、关键、稳定"的。

- 追问收口：那正确的做法是什么？把断言往接口和单元下沉，UI 只留"关键路径冒烟"。

【思政落点 1】：这一页讲的是 RoI，其实讲的是**务实**。行业里最容易犯的错，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，最后钱花了、效果没出来。工程判断的第一步永远是算账——投在哪一层回报最高、维护成本能不能扛住。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。

易错点 / 考点：① 四句话的顺序与措辞（尤其"适当加强手工 E2E"不能被记成"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 案例题常问"为什么单元测试 RoI 最高"，答"快、准、稳"三个字；④ "尽量避免 UI 测试"要理解成"避免把 UI 当主力"，不是"不做 UI 测试"。

> **过渡：**理论很漂亮，可现实往往不长这样。下一页我们看看"骨感"在哪。

【9.6】现实是骨感的（2 分钟）

这一页只有一句话：**现实是骨感的**。它接在分层策略后面，是给你打预防针——金字塔是对的，但绝大多数团队长出来的不是金字塔。

五条原因，你对号入座。**UI 自动化最好卖**：演示时浏览器自己跑起来，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。

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，团队很快就"习惯性忽略失败"。**环境不稳**：缺数据、服务没起、环境被占，脚本挂了你都分不清是产品坏了还是环境坏了。

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——工具解决不了，是**可测性**问题，要回到需求和设计去改。

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。

三条对策：**先算账再动手**，按 RoI 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。

> **板书：**中间写"现实是骨感的"；左侧五因：UI 好卖｜脚本脆｜环境不稳｜用例不可自动化｜指标虚荣；右侧三策：先算账 → 先底层 → 维护成本一等公民。

> **提问：**团队里有一批脚本已经连续红了两个月，大家跑报告时直接跳过它们。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"——这是工程上最稳的做法：给它们打上待处理标记，限期内要么修好要么删除。

- 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 "现实是骨感的"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 记住"长期红灯＝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，下一页给你一个二维框架。

【9.7】自动化测试四象限（2 分钟）

这一页给的是**自动化测试四象限**。课件在这里只留了标题和一句"详细讨论见公众号文章"，图我按业界通行的分法讲，你重点记**用法**，不记出处。

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**，两两组合出四格。

面向技术＋适合自动化：单元测试、接口契约测试、静态分析、性能基线——这是自动化的主战场，投入产出最高。**面向业务＋适合自动化**：功能回归、验收用例的重复执行——能做，但要挑稳定、关键的业务路径。

面向业务＋适合手工：探索式测试、新功能第一次验收、易用性评估——自动化做不了，因为它的价值恰恰在"人临场发现问题"。**面向技术＋适合手工**：渗透的创造性部分、故障演练、性能定位分析——工具给数据，判断靠人。

这个象限最大的用处，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需机器比对？答完，该不该自动化基本就定了。

记住一句话：**自动化擅长"重复确认已知"，人擅长"首次发现未知"**。

板书：画 2×2 矩阵。横轴：面向技术 ←→ 面向业务；纵轴：适合自动化 ←→ 适合手工。四格分别填：单元／契约／静态／性能基线｜功能回归／验收重复执行｜探索式／首次验收／易用性｜渗透创造部分／故障演练／性能定位。

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＋适合手工"——位置答对了。

- 答"因为还没写脚本"——这是错答。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：那什么时候可以自动化？等这个功能稳定了、预期结果明确了、要反复回归了——它就移动到"面向业务 + 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 四格的典型活动至少各能举一例；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 注意"同一功能会换格子"这个动态视角，案例题常考"新功能该不该立刻上自动化"。

> **过渡：** 定了该做什么，接下来要看"自动化"这个词到底指多大的范围。

【9.8】自动化测试的内涵（3 分钟）

这一页给"自动化"划范围，四句话，我们一句一句抠。

第一句：**通过平台、系统或工具自动地完成测试的某类工作，都可以归为测试自动化**。注意"某类工作"这四个字——它把范围放得很宽：环境自动部署算、用例自动生成算、缺陷自动分派也算。所以"测试自动化"是**大概念**。

第二句：**自动化测试更侧重测试用例或测试数据生成、测试执行和测试结果呈现等自动化**。这是"小概念"，也是我们平时说的那个意思：**用例怎么来、怎么跑、结果怎么给人看**。你注意它把三件事并列——生成、执行、呈现。很多团队只做了中间那件（执行），两头都靠人，自动化收益自然上不去。

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：不需要运行程序，用工具扫代码、扫配置、扫依赖，把评审从"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**。这句话很重要，它纠正一个常见误解：以为自动化就是"点点点"。实际上性能、安全、兼容这些专项，自动化程度可能比功能测试还高。

收口：这一页给你两把尺子——**大概念看范围，小概念看重点**。考试问"什么是测试自动化"，你把四句都写上，才完整。

> **板书：** 上写"测试自动化（大）＝ 用平台／系统／工具自动完成某类测试工作"；下写"自动化测试（小）＝ 用例与数据生成 ＋ 执行 ＋ 结果呈现"；右侧两条补充："＋ 评审辅助与静态分析"、"覆盖接口／UI 功能／专项测试"。

> **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"——只对了一半。按大概念算"测试自动化"，按小概念（生成／执行／呈现）它缺了最关键的**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"——这个区分很到位。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"——抓住了核心。**判定权交给机器**，才算真正的自动化测试。

- 收口：所以我给你一个更硬的判据：**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，你对号入座看自己团队缺哪块。

【9.9】自动化测试的分类（3 分钟）

> 这一页是本章的骨架图，六类，我按"投入从小到大、收益从高到低"的顺序走一遍。
>
> 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——装个工具接进流水线就开始了；收益很快——空指针、资源未关闭、硬编码密码这类问题，几分钟就能扫出一批。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。
>
> 第三类，**API（接口）自动化测试**。这是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 自动化测试**。Selenium 那一套，最直观也最脆。它负责"用户看到的这一层到底对不对"。
>
> 第五类，**移动 App UI 自动化测试**。Appium、Airtest 这一套，比 Web 多了一层设备管理：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**。前五类都是"零件"，这一类是"总装线"：把脚本、数据、调度、报告、环境管到一起。小团队可以先用开源拼，规模上来就必须有自己的集成层。

这六类不是并列关系，是**有先后的**：先静态分析和单元（便宜、快），再接口（性价比），最后 UI（贵、脆）。你如果只能做一个，做接口。

板书：六行表格式板书：① 代码静态分析（最小投入・最快见效）② 单元（快・准・稳）③ API（性价比最高）④ Web UI（直观・脆）⑤ 移动 UI（多一层设备管理）⑥ 集成框架（总装线）。右侧箭头："先便宜后昂贵：静态 → 单元 → 接口 → UI"。

提问：如果团队只允许你今年做一类自动化，你选哪一类？给出两个理由。

预设回答：

- 答"UI，因为最像用户"——直觉对，但性价比最低；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"——这是主流正确答案：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 RoI 最高"——也对，但有个前提：开发愿意写、代码结构支持。若团队没有单测习惯，接口是更现实的起点。

- 追问收口：选哪类还要看**你的痛点在哪**——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 每类至少能说出一个代表工具：静态分析（第 16 页那批）、单元（JUnit／pytest）、接口（Rest-Assured／SoapUI）、Web（Selenium／Cypress）、移动（Appium／Airtest）、集成（Robot Framework）；③ "哪类 RoI 最高"要分清：**单元 RoI 最高**（课件口径），**接口性价比最高**（工程口径），两者不矛盾。

过渡：从最便宜的一类开始——代码静态分析。

【9.10】代码静态分析（1 分钟）

这一页是静态分析这一小节的分隔，我交代它要解决什么问题。

一句话：**不运行程序，也能找出问题**。价值在"早"——代码刚写完、还没跑起来，工具就告诉你了。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。

这一小节按三条线走。**评审的辅助工具**：人评审最容易漏的是几百行 diff 里的那个边界判断——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）。

静态分析的原理：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"工具为什么会误报"。

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。

记住它的定位：**所有自动化里最便宜的一道闸门**。

> **板书：**中间写"代码静态分析＝不运行程序找问题"；下方三条线：评审辅助工具 → 分析原理（词法・语法・控制流・数据流・符号执行・值依赖）→ SAST。右侧写"价值在一个字：早"。

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"——这就说到原理层了。

- 答"靠猜"——这个说法粗糙但抓住了要害：静态分析本质是**近似推理**，近似就有误报，所以要人复核。

- 收口：下一页就先看人评审怎么被工具辅助。

易错点 / 考点：① 静态分析的定义（不运行程序）；② 它与第 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（并发、性能、资源竞争）它看不到。

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）

这一页给的是 GitHub 的 Pull Request 协作流程，作为"代码评审辅助工具"的实例。我不讲怎么点按钮，讲**它为什么能把评审做成可自动化的流程**。

传统评审是什么样？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。问题有三个：**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）、**拦不住**（说不改也能合）。

PR 这套流程把三件事都解决了。第一，**差异可视**：它以 diff 的形式把"改了什么"呈现出来，评审人按块看，不会淹没在整份文件里。第二，**讨论可追溯**：每条意见挂在具体某一行上，形成线程，谁提的、谁改的、改没改，全留痕。第三，**合并有闸门**：可以配置必须通过的检查（CI 跑单测、跑静态扫描）和必须签字的评审人，不满足就不许合。

这就是"辅助工具"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（格式、单测、静态告警）交给机器，机器查不了的（这段逻辑是否多余、这个抽象是否合理）留给人。

顺便说一句，这个流程本身就是一次"自动化闸门"的示范——下一页讲原理，最后第 71 页讲 CI 时你会再见到它。

板书：写"PR = 评审的工程化载体"；下面三列：差异可视（diff 分块）｜讨论可追溯（行级线程）｜合并有闸门（必过检查＋必签人）。右侧一句："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。规则去掉后，最先失守的就是"今天太忙，先合了再说"。

- 答"会，因为没有硬约束，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，但小团队不需要这么严"——这个观点可以讨论：小团队可以简化流程，但"改完必须被看过"这条底线不能没有。

- 收口：所以工具的价值不在"更聪明"，在"更可靠地执行纪律"。

易错点 / 考点：① 评审辅助工具的三大作用（可视、可追溯、有闸门）；② 案例题常问"如何提高代码评审的有效性"，答"缩小评审粒度＋机器先查＋强制闸门"；③ 别把 PR 当成"版本管理功能"，它的本质是**协作与准入控制**。

过渡：工具能查的那部分，靠的是静态分析技术。它是怎么"看"代码的？

【9.12】代码静态分析（3 分钟）

这一页讲原理，四层，一层比一层深。

最浅一层，**词法分析（Lexer）**：把源码这串字符切成 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"。

第二层，**语法分析（parser）**：把 Token 流按语法规则拼成**抽象语法树 AST**。到这一步工具知道"这是一个 if，条件是 a 大于 0"，AST 是多数工具的起点。

第三层，**控制流分析与数据流分析**：控制流图回答"程序可能从哪儿走到哪儿"，数据流分析回答"这个变量在哪儿赋值、能流到哪儿"。真问题的判断多在这层：未初始化的变量、没关闭的句柄、拼进 SQL 的用户输入。

第四层，课件特别强调的：**用模拟程序执行的技术**，如**符号执行、抽象解释、值依赖分析**，并用**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**。这句点出核心矛盾——**精度与成本的取舍**：全路径枚举精度高但跑不完，表面扫描跑得快但误报一片，约束求解就是中间的平衡木。

中间还有**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**一套引擎适配多种语言**。

一句话：**字符 → Token → AST → IR → 图 → 结论**，越往下理解越深，代价越高。

> **板书：**横向流水线：程序 →（词法分析）Token →（语法分析）AST → IR → 控制流图／数据流图 → 分析结论。下方补一行："符号执行・抽象解释・值依赖分析 ＋ 约束求解（路径约减・可达性分析）→ 降误报、提效率"。

提问：为什么静态分析工具总会报一些"其实没问题"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。根因在**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。比如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"——这也是原因之一，正则类工具尤其明显。

- 追问收口：所以成熟的用法是**分级**：高危的必须清零，中危的人工复核，低危的看趋势。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（词法→语法→AST→IR→分析）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：**路径约减、可达性分析**，且目的写"减少误报、提高效率"；④ 常考判断："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：**第四层里最出名的技术是符号执行，下一页用一个十行的小程序把它讲透。

【9.13】示例：符号执行（3 分钟）

这一页用十行代码把符号执行讲明白。方法 `CoverMe` 收一个整型数组 a：a 为 null 就返回；a 的长度大于 0；a 的第 0 个元素等于 1234567890 就抛异常，注释写着 bug。

问题在**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，而当符号**，沿控制流一条条走路径。

第一条：`a == null` 为真 → 直接 return。第二条：`a == null` 为假、`a.Length > 0` 为假 → 什么也不做。第三条：非空且长度大于 0，再判 `a[0] == 1234567890`，为假就结束，为真就抛异常——**bug 找到了**。

关键机制：每走完一条路径，就把这条路径上**条件的取反**作为新目标，让约束求解器解出一组能行的输入，再拿去跑下一轮。课件标的那两处就是"**取反后的条件**"和"**结束：没有其它路径了**"——所有可达路径都走完了。

代表作两个：**DART: directed automated random testing**（Godefroid、Klarlund、Sen，PLDI），思路是"具体执行＋符号约束"；**微软 Pex**，做 .NET 的白盒用例生成。

但它不万能：路径数是**指数级**的，每个 if 翻一倍。工程上必须**路径约减**，只挑有希望到达危险点的路径——这就是上一页说的精度与成本取舍。

> **板书：**左写代码三段判断；中间画三叉路径树：`a==null`（T→return／F）→ `a.Length>0`（T／F）→ `a[0]==1234567890`（T→抛异常／F）；右侧写"取反条件 → 约束求解 → 新输入 → 再执行"，底部写"DART（PLDI）・微软 Pex；路径爆炸 → 需路径约减"。

> **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，这是最直接的原因；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**：不可行的路径不该生成用例。

- 答"跳着判更快"——快是快，但会产出一堆跑不通的用例，误报率飙升。

- 追问收口：所以符号执行真正难的不是"解方程"，是**判断哪些路径是可行的**——这就是为什么要接约束求解器。

易错点 / 考点： ① 符号执行的核心："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／画图题）；③ 记住两个名字：DART、微软 Pex；④ 局限："路径爆炸"，靠路径约减缓解；⑤ 别把符号执行和"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"。下一页讲值依赖分析。

【9.14】值依赖分析（2 分钟）

这一页讲**值依赖分析**，它解决另一类问题：**这个值是从哪儿流过来的**。

课件定义：**数据流分析中的定值-使用连接的依赖关系，并通过值依赖图来表现**。两个术语拆开——"**定值**"是给变量赋值那一点，"**使用**"是读这个变量的那一点；定值-使用连接，就是"值赋完之后被哪些地方用了"。

第二句更具体：**值依赖图（Value Dependence Graph，VDG）是一种基于程序值依赖分析的、路径敏感的空指针解引用检测方法**，它**结合数据流分析中的到达定值分析、区间分析及指向分析**创建。

翻译成人话：空指针为什么难查？一个引用空不空取决于**它从哪条路径来**——A 分支来的是 new 出来的对象，B 分支来的是 null。全混着看分不清，只看一条又漏。VDG 做的就是**把值的来源和路径条件绑在一起记**，这就是"路径敏感"。

三样东西合起来才判得准：**到达定值分析**（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）。

课件引的文献是 Alan Christopher Lawrence 的《Optimizing compilation with the value state dependence graph》，2007 年——这套工具最早从**编译优化**里长出来，后来搬到缺陷检测上。

一句话：空指针检测的本质，是追**这个值从哪来、路上经过了什么判断**。

板书： 写"定值 → 使用（def-use）"；画一个小图：`x = new Obj()`（定值 1）→ 路径条件 `if (flag)` → `x.foo()`（使用）；旁边标三种分析：到达定值・区间分析・指向分析 → VDG → 路径敏感的空指针检测。

> **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 null 赋进来的。

- 答"看这条路径上有没有 null 的来源"——对了，这就是"到达定值"的思路：有没有某个 null 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：所以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 VDG 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 关键词"**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**判断某点上的值是否危险**。

> **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，**Static Application Security Testing**，静态应用安全测试。技术和刚才那套一样，目标换了——不看代码规范，看**安全漏洞**。
>
> 原理：**通过调用语言的编译器或解释器把高级语言代码转换成一种中间代码**，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、记录并分析数据传递过程**产生的安全问题，最重要——输入从 HTTP 参数进来，经几层方法拼进 SQL，这条链就是注入，工具要**把这条链画出来**。
>
> **控制流分析**：看**特定时间、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**配置文件中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、结构中的**问题，比如类暴露了不该暴露的公有方法。
>
> 最后收口：**结合多种分析结果，匹配规则库中的漏洞特征以发现漏洞**，并**形成包含漏洞信息的检测报告，包**

括具体代码行数以及修复建议。最值钱的是"**行号**"和"**修复建议**"——只说"你有 SQL 注入"的报告开发不看，说到第 87 行、建议参数化查询，他才会改。

板书：上写"SAST = 静态分析 + 安全规则库"；中间五行：语义（危险函数）·数据流（注入链）·控制流（权限绕过）·配置（密钥／开关）·结构（暴露面）；下方："规则库匹配 → 报告（行号 + 修复建议）"。

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"——正确做法。**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、低危挂账，并且**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，尤其**数据流分析**（注入类漏洞的根基）；③ 报告的两个要素：**代码行数 + 修复建议**；④ 常考判断："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，下一页就讲。

过渡：静态看代码是一路，还有一路是"打进去看反应"。下一页是 DAST。

【9.16】SAST 工具分类（3 分钟）

这一页按技术深度给 SAST 工具分四档，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 **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"绝不该出现的字面量"：私钥、内网地址、明文密码。

第二档，**基于语法树**，代表 **P3C、Fireline**：先把代码解析成 AST，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 规约扫描思路，它查的是"写法不地道"。

第三档，**Java 可以基于 class 文件**，代表 **FindBugs**：分析的是**字节码**而不是源码。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。

加分项：**SMT 求解器也被集成到一些大型工具中**，课件列了**HOL/Isabelle、ESC/Java2、ACL2、UCLID、BLAST、ureka、CUTE 和 PEX**——名字不必背全，但要懂它们代表"把程序能否走到坏状态变成一道数学题"。

板书：四行阶梯：① 正则匹配（Cobra、Raptor）→ ② 语法树（P3C、Fireline）→ ③ class 文件（FindBugs）→ ④ 控制流／数据流／调用关系（多数商业工具）。底部："＋ SMT 求解器（HOL/Isabelle、ESC/Java2、ACL2、UCLID、BLAST、ureka、CUTE、PEX）"；右侧写"精度↑ 成本↑"。

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：所以工具选型先问一句"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，尤其 P3C 属语法树档、FindBugs 属 class 档）；② SMT 求解器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 常考判断："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？下一页是 DAST。

【9.17】DAST（2 分钟）

这一页标题是 **DAST**。先说勘误：课件这页英文注释沿用了上一页的 "Static Application Security Testing"，是排版带过来的；DAST 的本义是 **Dynamic Application Security Testing，动态应用安全测试**。

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**（按漏洞类型发构造载荷）、**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**（生产开了调试模式、管理接口忘了鉴权）、**认证与会话**（Cookie 缺 HttpOnly）、**业务逻辑漏洞**（金额改负数、越权查别人订单），这些都没有"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**、**给不出代码位置**。一句话：**SAST 看"你写了什么"，DAST 看"你暴露了什么"**。

> **板书：**中间写"DAST = 跑起来 + 从外部打 + 看响应"；下方三步：爬 → 打 → 判；左侧列强项：配置与环境、认证与会话、业务逻辑漏洞；右侧列短板：覆盖面靠爬虫、无代码位置、可能污染数据。页首标注勘误："本页英文注释沿用了上一页，DAST 应为 Dynamic"。

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"这个订单是不是你的"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：所以安全测试的完整答案是"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（本页课件注释有笔误，按 Dynamic 记）；② 三步法：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 记住 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：IAST。

【9.18】DAST、SAST & IAST（3 分钟）

这一页把三种技术放一起比。SAST 与 DAST 讲过，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、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间怎么流，**既有运行时真实数据，又能定位到代码位置**。

所以 IAST 常叫"灰盒"：误报低（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（跑功能测试时顺手扫）。代价是**必须有人去跑**——测试没覆盖的代码它也看不见；还要装 agent，对语言与运行时版本有要求。

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。

一句话记：**SAST 早、IAST 准、DAST 真**。

板书：三列对比表：SAST（看图纸｜编码期｜白盒｜有行号｜误报偏高）／DAST（拆房试｜测试运行期｜黑盒｜无位置｜最可信但覆盖面靠爬虫）／IAST（装传感器｜测试执行期｜灰盒｜有位置｜必须有人跑）。底部一行："SAST 早、IAST 准、DAST 真"。

提问：如果团队只能先上一种，你选哪个？说出你的前提条件。

预设回答：

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

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

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

- 追问收口：所以答案取决于**你手上有什么**：有代码没测试 → SAST；有测试有环境 → IAST；要对外交付做验收 → DAST。这就是"因事制宜"。

易错点 / 考点：① 三种技术的定位词（看图纸／拆房试／装传感器）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"如何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 记住 IAST 的误报率最低这个结论，判断题常考。

过渡：静态这条线讲完了。下面进入本章第二块——真正意义上的"跑测试"：单元测试。

【9.19】单元测试（1 分钟）

这一页是单元测试的分隔，一句话交代任务：**把测试做到最小可测单元上**。

它的地位在第 5 页已经点过——**跑得最快、定位最准、维护最省，RoI 最高**。可它偏偏是学生最不愿写的一类，因为它不像 UI 脚本那样"看得见成果"。

这一小节三条线：**原理**，一个单元测试由哪几段组成——很多人只写"调用＋断言"，缺了准备和清理，用例互相污染，跑第二遍就红；**框架**，xUnit 家族以 JUnit 为主线，看它怎么用注解和反射把发现、执行、收集全自动，再看 JUnit 5 为什么要拆三层；**技巧与工具**，Mock 怎么替换不可控依赖，以及框架、智能生成、Mock、覆盖率这四类工具各管一段。

记一句：**单元测试不是"顺手写写"，它是有结构、有纪律的工程活动**。

> **板书：**中间写"单元测试＝最小可测单元上的自动化"；下方三条线：原理（结构四段）→ 框架（xUnit／JUnit 4／JUnit 5）→ 技巧与工具（Mock・用例生成・覆盖率）。右侧一行："快・准・稳 → RoI 最高"。

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，测试不该依赖随机。

- 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"——这是最常见的原因，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 技术出场。

- 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 单元测试的定位（最小可测单元、白盒为主）；② "RoI 最高"的结论要能给出三个理由；③ 常考判断："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）

这一页讲单元测试的结构。课件说得很到位：**如同代码实现，但有自己的结构**，然后给了四段：**初始化环境、环境清理、测试、断言**。

我按执行顺序重排：先**初始化环境**（造出被测对象、备好依赖与数据），再**执行测试**（调用被测方法），然后**断言**（比对实际与预期），最后**环境清理**（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，后一条就莫名其妙地红。这种"单独跑能过、一起跑就挂"的现象叫**测试间耦合**，是单元测试最常见的假失败来源。

再说**断言**：它是单元测试的**唯一判据**。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——测折扣就该断言"折后价 85 元"，不是断言"没抛异常"。

业界把这四段总结成 **AAA：Arrange、Act、Assert**，清理由框架钩子承担；还有 **FIRST 原则**：快速、独立、可重复、自验证、及时，可以当检查单用。一句话：**结构决定它能不能成批跑，断言决定它有没有价值**。

> **板书：**四段横排：初始化环境 → 测试（调用）→ 断言 → 环境清理；下方标"AAA：Arrange・Act・Assert（清理由框架钩子承担）"；再下一行"FIRST：Fast・Independent・Repeatable・Self-validating・Timely"；右侧红字："无断言＝只调用了一遍"。

提问：一条单元测试单独跑通过，和其他用例一起跑就失败。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，这是第一嫌疑：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，测试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：修法是两条——补上清理，或者让每条用例自己造自己的数据（独立），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"这个现象的识别与修法，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。先看 xUnit 这一家族。

【9.21】xUnit 框架（2 分钟）

> 这一页是 **xUnit 框架**。课件只给标题和图，我讲成一个家族的共同基因。
>
> x 是占位符：Java 是 JUnit，.NET 是 NUnit，Python 是 unittest，C++ 是 GoogleTest。语言不同、写法不同，**骨架是同一套**——学透一套，换语言只是换写法。
>
> 五个零件：**测试用例**，一个方法代表一个可独立判定的检查点；**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）；**测试套件**，把一批用例组织成批次；**运行器**，负责发现用例、执行、收集结果、出报告的发动机。
>
> 关系像一条生产线：**运行器是发动机，夹具是上下料，用例是工序，断言是质检，套件是批次**。懂了这条线，就懂为什么框架能"零配置发现用例"——靠**注解或命名约定加反射**。
>
> 还有一层价值常被忽略：**xUnit 把测试变成了代码的一等公民**——测试和产品代码同库、同构建，改代码就得改测试，测试不再是可以随手扔的东西。

> **板书：**中间写"xUnit ＝ 一个家族的共同基因"；五个零件：Test Case（用例）・Assertion（断言）・Fixture（夹具）・Suite（套件）・Runner（运行器）；下方一行类比："运行器＝发动机，夹具＝上下料，用例＝工序，断言＝质检，套件＝批次"。左侧列家族成员：JUnit／NUnit／unittest／GoogleTest／PHPUnit。

> **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 的设计影响了几代框架。

- 答"因为它们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"——这是更准确的溯源：这套模式起源于 Smalltalk 的 SUnit，Kent Beck 与 Erich Gamma 把它带到了 Java。

- 收口：所以学框架要学"骨架"而不是"API 名"，骨架稳定，API 会变。

> **易错点 / 考点：**① xUnit 里 x 的含义（语言／框架前缀的占位符）；② 五个零件的名称与作用（简答常考"测

试夹具是什么"）；③ 常考判断："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 记住"注解／命名约定 + 反射"是自动发现的机制。

> **过渡：**以 Java 为标本，先看经典款的 JUnit 4 是怎么跑起来的。

【9.22】JUnit 4 工作原理/框架（2 分钟）

这一页讲 **JUnit 4 的工作原理**。我按"按下运行到看见绿条"的顺序讲，你就能把图对上。

第一步**发现**：方法上标着 `@Test`，运行器用**反射**扫描所有方法，有注解就登记成一条用例——这就是"零配置"的来源。

第二步**准备**：每条用例前跑 `@Before`，整个类开始前跑 `@BeforeClass`。区别是粒度与成本：前者负责"每条都要的干净环境"，后者整类只跑一次，负责很贵的事，比如启动内存数据库。

第三步**执行与判定**：调用方法、用 `Assert` 断言；失败会抛断言错误并被标记，但**不会中断整批执行**——剩下用例继续跑，一次拿到全部结果，这是框架的核心价值。第四步**清理**：`@After` 每条之后、`@AfterClass` 整类结束，顺序与准备相反。第五步**汇总**：结果对象交给 IDE 或构建工具，你看到的红绿条就是它的可视化。

再补一句：参数化运行器、套件运行器、Rule 都是"换个 Runner 或加个拦截器"。**理解 Runner，JUnit 4 就通了**。

> **板书：**五步流程：发现（@Test + 反射）→ 准备（@BeforeClass 一次／@Before 每条）→ 执行与断言（Assert，失败不中断）→ 清理（@After／@AfterClass，逆序）→ 汇总（结果对象 → IDE／构建工具）。右侧标"扩展点：Runner・Rule・参数化・Suite"。

> **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。粒度不同：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"——这一条很关键：它是类级别的，拿不到实例字段。

- 追问收口：还有一个纪律：`@Before` 里准备的东西，`@After` 里必须能拆干净，否则就是前面说的测试间耦合。

> **易错点 / 考点：**① 四个注解的执行次数与顺序（简答／判断高频）；② 断言失败**不会中断**整批执行，这是框架设计的要点；③ 常考判断："@BeforeClass 方法可以是实例方法"——错，必须是静态方法；④ 记住扩展点名词：Runner、Rule、参数化、Suite。

> **过渡：**JUnit 4 是经典款，JUnit 5 则做了一次架构级的重写，下一页看它为什么这么拆。

【9.23】JUnit 5 架构（3 分钟）

> 这一页是 **JUnit 5 的架构**，课件开头给了等式：**JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage（复古）**。这个等式是本节最重要的考点。
>
> **JUnit Platform**：**基于 JVM 的执行测试的基础框架，以及它的 Test-Engine API、控制台启动器（CLI）、支持 CI/CD、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**和一个**插座（Test-Engine API）**：谁来测，插上就行。
>
> **JUnit Jupiter**：**在 JUnit 5 中编写测试和扩展的新编程模型和扩展模型的组合**，负责**运行基于 Jupiter 的测试**。你写 JUnit 5 风格的测试，写的就是 Jupiter。
>
> **JUnit Vintage**：**提供 TestEngine 在平台上运行基于 JUnit 3 和 JUnit 4 的测试**。真实项目里有成千上万条老测试，没有它，升级就是"全改或全扔"；有了它，**老测试原地不动也能跑**，可以按模块慢慢迁。
>
> 三层拆开的意义一句话：**把"怎么执行"和"怎么写"解耦**，也就是插件化——测试框架可共存、执行入口可统一、新引擎可接入。

> **板书：**等式竖排：JUnit 5 ＝ Platform（执行底座＋Test-Engine API＋CLI＋CI/CD 支持＋JUnit 4 Runner）＋ Jupiter（新编程模型与扩展模型）＋ Vintage（跑 JUnit 3／4 老测试）。右侧一句："执行与写法解耦 ＝ 插件化"。

> **【思政落点 2】**：JUnit Platform 的意义，是让"跑测试"成为**有统一入口、硬性插座的流水线**。背后的道理是：**质量检查不能靠自觉，要靠机制**。最危险的不是没有测试，而是"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——比开十次动员会都有效。守规则不是束缚，是让每个人的努力都算数。

> **提问：**如果一个项目里 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，但要注意注解来自不同的包"——这是个实操细节，很到位：JUnit 4 的 `org.junit.Test` 和 Jupiter 的 `org.junit.jupiter.api.Test` 不是一个东西。

- 收口：所以架构拆层的价值是**让迁移变成渐进的**，而不是一次豪赌。

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

> **过渡：**单元测试最麻烦的事往往不是断言，而是"依赖不听话"。下一页讲 Mock。

【9.24】Mock 技术（3 分钟）

这一页讲 **Mock 技术**，课件给了三个关键词：**round-trip test**、**SUT**、**DOC**。

术语先立住。**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。

问题就出在 DOC 上：它**不可控**（改不了系统时间、不可能真去扣款）、**慢**（一接数据库就从毫秒变秒级）、**引入不确定性**（网络抖一下测试就红，产品其实没问题）。

Mock 的做法是**用一个假的 DOC 顶上去**，它要满足两点——**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：给定假的依赖行为，断言 SUT 的输出与交互。

课件还有两个概念要分清。**round-trip test（往返测试）**：走完整的"写进去—读出来"，看数据经一圈之后还是不是原样——验的是序列化、持久化、协议转换有没有丢信息，代价是慢，通常放集成层。**"front door"（前门）**：通过**公共接口**驱动 SUT，不去改私有字段——只从公共接口测，测试才不会绑在实现细节上，重构时就不该红。

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"我预期的调用顺序"，不是"系统做对了事"。

> **板书：**中间写"SUT（被测）— DOC（被依赖）"；右侧画一个"假 DOC"顶替真 DOC；下方三个关键词：Mock（行为可编程＋调用可验证）・round-trip test（写进去读出来，验序列化与持久化）・front door（从公共接口驱动，不碰私有字段）。底部三条纪律：能不用就不用｜只 mock 边界｜别过度断言交互。

> **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：会影响同机其它程序，CI 上还要权限。

- 答"把时间作为参数传进来，测试时传固定值"——这是最干净的做法：**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，这就是"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。**优先改设计**，其次才用 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构造状态"——错；④ Mock 与 Stub 的区别：Stub 只给返回值，Mock 还验证交互；⑤ 常考判断："Mock 越多测试越好"——错，会变成测假设。

> **过渡：**原理清楚了，下一页看市面上的工具箱里都有什么。

【9.25】最常用的单元测试工具（3 分钟）

> 这一页是工具全景图。课件按三类语言、四类工具来组织，我按类别讲，你对号入座。

> **第一类，单元测试框架**。Java 有 **JUnit、TestNG**——JUnit 是事实标准，TestNG 在分组、依赖、参数化上更"测试友好"；C++ 有 **GoogleTest**；Python 有 **pytest、unittest**——pytest 主流，因为断言直接写 `assert`；unittest 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**。它们解决"没人愿意写测试"的老大难：给一个类，自动生成一批能提高覆盖率的用例。定位要讲清——**生成的是起点不是终点**：机器不懂业务语义，生成的断言往往只保证"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**。注意覆盖率工具不是测试框架，它是**度量工具**——只告诉你哪些代码被跑过，不告诉你测得对不对。

课件的总结正好是这四类；考试按四类答，条理最清。**工具会换，四类分工不会变：写用例、造用例、替依赖、量覆盖**。

板书：四类分栏：① 框架（JUnit・TestNG｜GoogleTest｜pytest・unittest）② 智能生成（EvoSuite・Diffblue Cover）③ Mock（JMockit）④ 覆盖率（JaCoCo｜gcov・lcov・gcovr｜Coverage.py）。底部一句："工具会换，分工不变：写・造・替・量"。

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，但不等于测对了；机器不知道业务规则。

- 答"不能，它主要用来兜住回归和发现异常"——这个定位最准：自动生成适合做"广度覆盖"，帮你在重构时快速发现行为变化。

- 答"要看生成工具是否会生成断言"——很专业的补充：只生成调用不断言的工具，价值有限。

- 收口：所以正确用法是"机器铺面、人补点"——机器覆盖大量普通路径，人补关键业务规则和边界。

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

过渡：单元层往下是接口层。接口自动化的架构比单元复杂，我们先看它的整体形状。

【9.26】API 自动化测试（1 分钟）

这一页是 **API 自动化测试**的分隔。先说清它为什么是"性价比之王"。

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 UI 操作几秒钟，差两个数量级；**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，UI 只是把它们显示出来。

这一小节两条线。**架构**：API 从哪来、怎么变成定义、怎么组织成场景用例、怎么触发回归、结果怎么展示——考的是你能不能画出这张图。**工具**：一类做"桩"（还没有真接口时先造个假的顶上），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。

记一句：**接口自动化的核心资产不是脚本，是"API 定义＋场景用例"**。

板书：中间写"API 自动化＝性价比之王（稳・快・覆盖主干）"；下方两条线：架构（来源 → 定义 → 用例 → 回归 → 结果）｜工具（桩类／测类）。右侧写"核心资产：API 定义 ＋ 场景用例"。

提问：为什么很多团队做自动化，第一条真正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**：接口脚本写起来快、跑起来快、改起来少。

- 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，业务主干逻辑都在后端。

- 收口：所以业界的普遍路径是"先接口、后 UI"，这不是偏好，是算过账的。

易错点 / 考点：① 接口自动化的三个优势（稳定、快、覆盖主干）；② 常考判断："接口测试不需要关注数据"——错，接口测试最难的恰恰是测试数据与状态；③ 记住"桩"与"测"是两类工具，第 28 页会给清单。

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）

这一页是 **API 测试自动化（TA）的架构图**，课件标注"复习"。我按"输入—定义—用例—回归—展示"五段走。

第一段，API 的来源四条路：**文档化工具**（平台里已有定义）、**手工录入**（老接口只能人工补）、**程序扫描**（从代码里扫路由与参数）、**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）。意义在于：**有了机器可读的定义，用例生成和断言校验才能自动化**。

第三段，用例：**场景设定、选择和组装 API、验证**；生产方式有三种——**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**——这一段正是接口自动化与 CI 的接缝。**第五段**：执行与结果展示，报告要能定位到"哪个接口、哪一步、期望什么、实际什么"。

一句话：**把接口变成定义，把定义变成场景，把场景变成可重复执行的回归**。课件还注了一句：**HAR 是基于 JSON、储存 HTTP 请求／响应信息的文件格式**——浏览器开发者工具就能导出。

板书：五段横向流程：来源（文档工具・手工录入・程序扫描・HAR 导入）→ 定义（RAML・Swagger・Markdown）→ 用例（场景设定・选择组装・验证；手工／HAR 批量／代码迁移）→ 回归（按需・定时・即时・事件触发）→ 执行与结果展示。底部注："RAML＝RESTful API 描述语言；HAR＝JSON 格式的 HTTP 请求／响应记录"。

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；但对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"——这是最实用的捷径：真实请求里路径、头、参数、返回全都有。

- 追问收口：实际工程里通常是**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 五段的名称与顺序（画图题／简答高频）；② 回归的四种触发方式必须能默写；③ RAML 与 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"混为一谈：来源是"从哪拿信息"，定义是"用什么格式描述"。

过渡：图看懂了，下一页看具体有哪些工具能填进这些格子。

【9.28】基于 API 的 TA 工具（2 分钟）

这一页是 API 工具清单，课件分两组，我按"桩"和"测"讲，你记用途不记名字。

做桩的：**Mockbin，可从 HAR 文件生成一个模拟桩**——抓一次真实请求，它按那个响应回你，适合"后端没好、前端要联调"。**RAP，淘宝开发的 API 管理工具**。**Swagger，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。还有两个工程化的 **kristofa/mock-http-server** 和 **wiremock**——后者在 Java 生态

里几乎是契约测试标配。

做测的：**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，适合进代码库进 CI）、**SOAtest**（商业，偏企业级）、**APIfortress**（API 安全与质量一体化）。

两组关系讲清楚：**桩解决"依赖不存在"，测解决"行为对不对"**。桩先行让前后端并行；接口就绪后测接管，做回归。选型一句话：**Java 团队用 Rest-Assured 配 WireMock，要图形化快速上手就用 SoapUI 或 RAP**——别为了工具改技术栈。

板书：两栏。左栏"桩（Mock 服务）"：Mockbin（HAR→桩）・RAP（淘宝 API 管理）・Swagger（定义规范＋mock）・EasyMock（Swagger→数据）・Doclever（文档→测试数据）・Mocky（免登录）・mock-http-server・WireMock。右栏"测"：SoapUI・Rest-Assured・SOAtest・APIfortress。底部："桩解决依赖不存在，测解决行为对不对"。

提问：前后端并行开发时，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"——项目进度不允许，而且前端会把时间浪费在等待上。

- 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，白忙一场。

易错点 / 考点：① 桩／测两组工具的归属（选择、填空高频，尤其 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＋测试数据；④ 常考判断："有了 Mock 就不需要真实接口测试"——错，桩只能验证前端，不能验证后端。

过渡：接口之后是 UI，这一层最直观，也最容易让人误判它的价值。

【9.29】Web UI 自动化测试（1 分钟）

这一页是 **Web UI 自动化测试**的分隔。先把态度摆正——第 5 页说"尽量避免 UI 测试"，但"避免当主力"不等于"不做"。

三种必须做：**关键业务链路的冒烟**（登录、下单、支付，每次发布前真点一遍，它们是"整条链路能不能走

通"的最终判据）、**跨浏览器与兼容性**（同一页面在不同浏览器上表现可能不同，只有真跑才能发现）、**前端逻辑复杂的地方**（表单联动、富文本、拖拽，接口测不到）。

这一小节三步走：UI 自动化的三个基本动作（识别、操作、验证）；元素的本质 DOM，理解它才知道"定位"在定什么；最后三套主流路线——WebDriver 的"远程控制"、Cypress 的"跑在浏览器里"、SikuliX 的"看图点鼠标"。

一条纪律：**UI 脚本要少而精，断言要断业务结果，别断像素和文案**。

> **板书：**中间写"Web UI 自动化"；左侧三种必做场景：关键链路冒烟／跨浏览器／前端复杂逻辑；下方三步：识别 → 操作 → 验证；右侧三条技术路线：WebDriver（远程控制）・Cypress（浏览器内）・SikuliX（图像识别）。

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指数上升。

- 答"两者都有道理，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；不能，才放 UI。

易错点 / 考点：① UI 自动化适合的三类场景；② 常考判断："UI 自动化越多越好"——错；③ 三条技术路线的名字与差异（下一节展开）；④ 记住"少而精"这条纪律，案例题答到能加分。

过渡：先看 UI 自动化的三个基本动作，所有框架都是这三步的变体。

【9.30】Web UI 自动化测试（3 分钟）

这一页是 UI 自动化的三个基本动作，课件写得很简洁：**识别 UI 元素、操作 UI 元素、验证状态（比较）**。所有框架——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**，因为它是脚本与界面唯一的耦合点。识别方式从 id 到 xpath 到图像，稳定性依次下降。

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（要用显式等待，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 btn-primary、文案是"提交"）——前者稳定，后者一改版就红。

三步的关系打个比方：**识别是"看得见"，操作是"够得着"，验证是"说得清"**。只识别不操作是截图工具，只操作不验证是"点了一遍没报错"。还有一条工程纪律：**这三步要封装**——把"登录"封成一个方法，把定位收进对象库，这就是第 57 页讲的"对象库"思想。

板书：三步横排：识别（定位元素，最脆弱）→ 操作（点击・输入・选择・滚动，坑在时序：显式等待、iframe、遮挡）→ 验证（比较，断业务结果不断实现细节）。底部比喻："看得见・够得着・说得清"；右侧写"三步要封装：登录＝一个方法，定位进对象库"。

提问：一条 UI 脚本"点了提交按钮，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"，但证明不了"提交成功了"。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，"比较预期与实际"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 脚本的顺序应该是**先想断言，再写操作**——先问"我怎么知道它成了"，再问"我要点哪儿"。

易错点 / 考点：① 三步的名称与顺序（本节最高频考点）；② "识别"是最脆弱环节这个结论；③ 断言的两类对象（业务结果 vs 实现细节），案例题常考"为什么这条脚本改版就红"；④ 固定睡眠（sleep）与显式等待的区别，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 页以"对象库"表述。

过渡：识别这一步到底在"识别"什么？答案是 DOM。下一页看 UI 元素的本质。

【9.31】Web UI 元素（2 分钟）

这一页看 **Web UI 元素**。概念一句话：**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，本质上都是带标签、带属性、带文本的树节点。

常见的元素类型要能认：文本输入框、单选与复选、下拉选择、按钮、链接、表格与列表、文件上传、富文本编辑器、日期选择器、弹层与提示。它们的行为差别很大——下拉框有的能直接设值、有的必须模拟点击展开再选，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"，你得提前知道：**iframe**，页面里嵌页面，脚本必须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，你找得太早就找不到。这三样是 UI 脚本"时好时坏"的主要来源。

一句话：**元素不是图像，是节点；认清了节点，才知道该怎么找、怎么点、怎么验**。

板书：中间写"UI 元素 = DOM 节点（标签＋属性＋文本）"；下方列常见类型：输入框・单选复选・下拉・按钮・链接・表格・上传・日历・弹层；右侧三个"麻烦元素"：iframe（要切换）・Shadow DOM（穿不进去）・动态渲染（要等）。

提问：为什么同一条"点击按钮"的脚本，昨天能跑今天不行，而产品功能其实是好的？

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"——最常见的原因，要改用显式等待。

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

- 收口：所以排查顺序是：**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 与 Shadow DOM 要切换／穿不透的区别（判断题常考）；③ 动态渲染与显式等待的关系；④ 别把"元素可见"等同于"元素可点击"——被遮挡、被禁用都点不到。

过渡：既然元素是节点，那节点归谁管？归标准管。下一页看 DOM 标准。

【9.32】W3C DOM 标准（3 分钟）

这一页是 **W3C DOM 标准**（dom.spec.whatwg.org），课件列了一串 API。这些不要背，分四组看。

查找：`document.getElementById(id)`、`document.getElementsByTagName(name)`——按 id 找一个、按标签名找一批。

创建与读写：`document.createElement(name)`、`parentNode.appendChild(node)` 负责造节点、挂节点；`element.innerHTML`、`setAttribute()`、`getAttribute()`、`style.left` 负责读写。后者是**断言的主战场**：验文本看 innerHTML、验属性看 getAttribute；断言"按钮禁用"看 disabled，别看它灰不灰。

事件与窗口：`element.addEventListener()`、`window.onload`、`window.scrollTo()`、`window.content`、`window.dump()`。自动化里用得最多的是"**等页面加载完**"。

目的不是背 API，而是明白：**UI 自动化的一切动作最终都落到这些标准 API 上**。WebDriver 的 click 底下是协议，`executeScript` 则让你直接写 JS 调它们。

板书：四组分类：查找（getElementById・getElementsByTagName）｜创建挂载（createElement・appendChild）｜读写（innerHTML・setAttribute／getAttribute・style）｜事件与窗口（addEventListener・onload・scrollTo・content／dump）。底部："UI 自动化的一切动作最终落到 DOM API"。

提问：要断言"这个按钮处于禁用状态"，你应该读什么？

预设回答：

- 答"看它是不是灰色的"——这是视觉判断，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：所以原则是**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` 与 `window` 的区别：document 是文档对象，window 是浏览器窗口（判断题常考）；③ `innerHTML` 与 `textContent` 的差别：前者含标签，后者只要文本；④ 记住 `executeScript` 可以直接执行 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，下一页看怎么拿这些 API 去"测"。

【9.33】测试 DOM API（2 分钟）

这一页是 **测试 DOM API**，意思是：**把 DOM API 当作被测对象，也当作测试手段**。这两层意思都要讲。

第一层，**当作手段**。你在浏览器开发者工具的控制台里敲一句 `document.getElementById("submit").disabled`，立刻能看到结果。这个习惯非常重要：**写脚本之前，先在控制台里把定位器和断言验一遍**。控制台里能拿到的东西，脚本里才拿得到；控制台里拿不到的，别指望脚本能拿到。很多同学写完脚本跑失败，才开始猜，其实一开始在控制台验一句就省半小时。

第二层，**当作被测对象**。前端代码本身会操作 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？用断言去比对**操作前后的 DOM 状态**：点"展开"之后，某个节点的 class 里是否多了 expanded；提交失败之后，错误提示节点是否出现、文本是不是预期的那一句。

一句话总结这一页：**控制台是你的实验室，DOM 状态是你的断言对象**。

> **板书：** 左右两栏。左"DOM API 当手段"：控制台先验定位器与断言 → 再写脚本；右"DOM API 当对象"：断操作前后的节点状态（class・文本・显隐・属性）。底部一行："能用状态断言，就别用像素断言"。

> **提问：** 你要验证"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，为什么？

> **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、甚至抗锯齿都会让像素不同。

- 答"查节点文本"——正确。它断的是业务信息"提示说了什么"，不是"红到什么程度"。

- 答"两个都用，双保险"——听起来稳，实际是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：所以判断标准是：**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＋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 常考判断："截图比对是最可靠的 UI 断言方式"——错，它是最脆弱的方式之一。

> **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）

这一页是 **定位元素**。课件把 Selenium 的定位写法列全了，我按"稳定性"排序讲——这比记住语法更重要。

最稳的是 id 与 name：`By.id("id 的值")`、`By.name("name 的值")`。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("css 表达式")`，比 xpath 快、写法短，还能表达层级与属性。**第三档 xpath**：`By.xpath("xpath 表达式")`，最强也最容易写坏——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 与 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：`By.linkText("链接的全部文字")`、`By.partialLinkText("链接的部分文字")`——只能用于 `<a>`，靠可见文字定位，方便但**文字会变**。

最后是 jQuery：`Js.executeScript("return jQuery.find(...)")`——前提是页面加载了 jQuery，现在很多工程已不用，慎用。

记法：**id ＞ name ＞ css ＞ xpath ＞ 文本 ＞ 索引**。索引排最后，因为"第 3 个 div"定的是位置不是身份。

> **板书：**九种 By 按稳定性竖排：id ＞ name ＞ cssSelector ＞ xpath ＞ className／tagName ＞ linkText ／partialLinkText ＞ jQuery（executeScript）。右侧两条红字："绝对路径 xpath 是反模式"、"索引定位定的是位置，不是身份"。

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，比如按表单的 name 加按钮类型去限定。

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

- 追问收口：这就是为什么"定位"不只是测试的事——**测试人员提需求、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` 与 `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，且要通过 `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）

这一页是 **UI 元素定位示例**，我给一个判断流程，拿到任何页面都能照着做。

第一问：有没有稳定标识？ 有 id 或 `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。

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，因为改了会影响功能本身。

第三问：只能用结构吗？ 那就用"就近的稳定祖先＋相对路径"，别写绝对路径。**绕不开文案**也行，但要标风险——文案是多语言与迭代的高频变动区。

再给**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**。

一句话：**定位的功夫不在写表达式，在判断"哪个属性最不容易变"**。

板书： 四问流程：有稳定标识？→ 语义属性组合？→ 结构＋相对路径？→ 文案（标风险）。下方排查三步：在不在（渲染／iframe）→ 唯不唯一（命中多个）→ 可见不可见（遮挡／禁用／零尺寸）。

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"——这是最快的判别法：控制台能找到，说明脚本时机或上下文不对；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：所以诊断顺序是"控制台验选择器 → 看 iframe／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` 类测试专用属性的意义；④ 常考判断："用文案定位最稳"——错，多语言与迭代都会让它失效。

> **过渡：** 定位是方法层，真正把这些方法送到浏览器里的，是 WebDriver。

【9.36】Selenium + WebDriver（2 分钟）

> 这一页是 **Selenium + WebDriver**。它要回答：**脚本在测试机上跑、浏览器在另一处开，两边怎么对话？**
>
> 答案是一套协议。链路：**测试代码调客户端库 → 翻译成 HTTP 请求（WebDriver 协议）→ 发给浏览器驱动（如 ChromeDriver）→ 驱动通过浏览器的调试接口操作浏览器 → 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）。
>
> 再说家族：早期有 IDE（录制回放）和 RC（已废弃），今天是 **WebDriver** 当家、**Grid** 做分布式。听人说"Selenium 就是录制"，那是十几年前的印象——**录制解决"看"，不解决"维护"**，这也是第 72 页第一道思考题的由来。
>
> 一句提醒：既然一切走 HTTP，**网络与驱动的稳定性就是脚本稳定性的一部分**；驱动版本与浏览器版本不匹配，是最常见的环境故障。

> **板书：** 画链路：测试代码 → 客户端库 →（HTTP／WebDriver 协议）→ 浏览器驱动 → 浏览器；右侧三个好处：跨语言・跨浏览器・可远程。底部一句："驱动与浏览器版本不匹配＝最常见的环境类失败"。

> **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该浏览器的方式。

- 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，这正是 Selenium 4 之后协议统一的基础。

- 收口：所以"抽象一层协议"这个设计，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 常考判断："Selenium IDE 的录制可以作为主流开发方式"——错，见第 72 页；④ 驱动版本与浏览器版本必须匹配。

> **过渡：** 链路里有一个细节值得单独看——那个发 HTTP 的"客户端库"，它自己也是可替换的。

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

> 这一页讲一个很工程化的细节：**Selenium 的 HTTP 客户端可以替换**。课件给了两个 POM 依赖：`selenium-java` 与 `selenium-http-jdk-client`，版本都是 **4.6.0**。
>
> 先看它用 HTTP 与 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、根据网格模式让各网格组件互相通信、创建 ChromeDevTools 协议和 BiDi 协议会话**。四条说明一件事：Selenium 内部所有"对外说话"都走同一个通道。
>
> 课件还交代现状：**目前使用 AsyncHttpClient（建立在 Netty 之上，支持异步执行 HTTP 请求与 WebSocket）**。
>
> 那为什么还要引 `selenium-http-jdk-client`？因为它提供**基于 JDK 自带 HttpClient 的实现**：少一层第三方依赖（不用把 Netty 拖进依赖树）、与 JDK 的 TLS／代理／连接池配置天然一致、排查日志更贴近 JDK 体系；前提是 JDK 11 起才有 java.net.http。
>
> 教学价值不在记坐标，而在设计思想：**把通信抽象成可替换组件**，协议演进时上层 API 不用动——这就是第 57 页"接口稳定、实现可换"的设计要点。

> **板书：** 左写两个依赖：`selenium-java 4.6.0` ＋ `selenium-http-jdk-client 4.6.0`（POM）；中间四条用途：向 WebDriver 发命令・向 Grid 发命令・网格组件互通・建 CDP／BiDi 会话；右下写"默认 AsyncHttpClient（基于 Netty，支持异步 HTTP＋WebSocket）"；底部一句："通信抽象成可替换组件 → 上层 API 不动"。

> **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"——这是最实在的收益，尤其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"——很到位的补充，企业内网环境尤其明显。

- 收口：所以这类替换属于"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 两个依赖的坐标与版本（课件给的 4.6.0）；② 四条用途（简答可考）；③ 默认实现是 AsyncHttpClient（基于 Netty）；④ 常考判断："Selenium 只能用它自带的 HTTP 实现"——错，可替换；⑤ 别把 CDP（ChromeDevTools 协议）与 BiDi 混为一谈，两者都是浏览器调试／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。下一页看 Selenium Grid。

【9.38】Selenium Grid（2 分钟）

这一页是 **Selenium Grid**，解决的问题一句话：**让一批测试同时跑在多台机器、多个浏览器上**。

结构是**中心—节点**模型：**Hub** 接收测试请求、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。

好处三个：**时间**（一千条串行几小时，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账号的购物车就互相踩，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"，要用容器镜像固定；**失败归因**——报告一堆红，得能分清真缺陷、脚本脆、环境问题。

一句话：**Grid 放大吞吐量，也放大你的问题**——写得好是加速器，写得烂是放大器。

板书： 画 Hub—Node：Hub（接收・按能力分派）→ Node1（Chrome／Linux）・Node2（Firefox／Win）・Node3（Safari／macOS）。左侧三个好处：省时间・保覆盖・做隔离；右侧三个坑：数据隔离・环境一致・失败归因。底部："Grid 放大吞吐，也放大问题"。

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，尤其涉及登录、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：所以并行化的第一步不是加机器，是**让用例可独立执行**——数据自己造、环境自己清、顺序不依赖。

> **易错点 / 考点：** ① Hub／Node 的职责与"能力匹配"的分派机制；② 并行的三个收益与三个坑（简答高频）；③ 常考判断："并行度越高越好"——错，受数据隔离与环境一致性制约；④ 记住"用例必须可独立执行"这条前置条件。

> **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）

> 这一页是 **Cypress**，课件只给了官网地址，我讲它和 Selenium 的**根本差别**，因为差别决定了适用范围。
>
> 根本差别是**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。这就带来四个直接结果。
>
> **第一，不用等**。因为它在浏览器里，能感知网络请求与渲染的完成，自带**自动等待与重试**——你写"点这个按钮"，它会等到元素出现、可点击再点。这一条直接消灭了 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（`intercept` 这类能力），做桩非常自然，不必另起一个 mock 服务。
>
> **第四，限制也来自位置**。跑在浏览器里，就受浏览器的同源与标签页限制：跨域访问、多标签页协作、多个浏览器并行这类场景，它不如 WebDriver 顺手；另外它的语言生态以 JavaScript／TypeScript 为主，团队若不是前端栈，学习成本要考虑。
>
> 所以选型判断很清楚：**前端团队、单页应用、要快速反馈**——选 Cypress 很舒服；**要多浏览器矩阵、要跨系统端到端、团队是 Java／Python**——Selenium 更合适。这不是谁更强，是**谁更适合你的场景**。

> **板书：** 左右对比：Selenium（脚本在浏览器外・协议遥控・跨浏览器强・需自己处理等待）｜Cypress（脚本在浏览器内・同源运行・自动等待与重试・可拦截网络・时间旅行・受标签页／跨域限制）。底部："差

> 别在位置，选型看场景"。

> **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：重试机制让偶发的时序抖动不至于直接判失败。

- 收口：所以 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（脚本位置）；② Cypress 的四个特点与两条限制；③ 常考判断："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、异步任务仍需显式等待。

> **过渡：**它的能力从哪来？看它的架构就明白了。

【9.40】Cypress 架构（2 分钟）

> 这一页是 **Cypress 架构**。看懂它，上一页那些"能"与"不能"就都有解释了。
>
> 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、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。
>
> 这个双进程结构带来三个关键能力。**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"而不是"发协议"。
>
> 代价也在结构里：**测试和应用同源同上下文**，意味着跨域跳转、打开新标签页这类动作它会受限；也意味着**它不适合做"多系统串起来"的端到端**——你去访问另一个域名，它就出了自己的地盘。

> **板书：**两框：Node 进程（启动浏览器・起代理・打包测试・收集结果）⇄ 浏览器内运行器（测试代码与应

> 用同上下文）。三个能力：网络可控（代理）・上下文共享（window）・快（本地调用）。一个限制：同源同上下文 → 跨域／多标签受限。底部："架构定边界，边界定场景"。

> **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**——能力越大，越要自我约束。

> **易错点 / 考点：**① 双进程结构（Node 侧＋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 常考判断："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）

> 这一页是**基于图像识别的 UI 自动化**，课件给的是 **SikuliX → SikuliNG**，还有一条 REST API 的实现说明。
>
> 它填的是别人的空档：WebDriver、Appium 都要求**能拿到控件树**，可远程桌面里的老客户端、自绘界面、工控与医疗设备、游戏，在系统看来就是**一张图**——没有控件、没有 id、没有 DOM。
>
> 思路一句话：**你眼睛怎么找，它就怎么找**——按截图模板在屏幕上匹配，匹配到就按坐标去点。SikuliX 用 OpenCV 做模板匹配；演进到 **SikuliNG 后把 API 完整实现为 REST-API、由跑在目标机器上的服务支撑**，于是你能用任何语言远程驱动那台机器，不必在它上面装 Python。
>
> 优点直白：**通用**（屏幕上看得到就能操作）、**跨技术栈**（Web、桌面、移动、嵌入式一视同仁）。缺点同样直白：**脆**——分辨率、主题、字体、缩放一变就匹配失败；**无语义**——它认得"这张图长这样"，不知道"这是提交按钮"，断言只能停在"这块区域长得对"。
>
> 工程建议三条：**固定环境**（分辨率、缩放、主题锁死）、**只截稳定局部**（别截整窗）、**当兜底不当主力**——补死角很救命，当主力很折磨。

> **板书：**中间写"SikuliX → SikuliNG：按图找、按坐标点"；下方链路：截图模板 → OpenCV 匹配 → 得到坐标 → 鼠标键盘 → 断言（区域像素）；左侧优点：通用・跨技术栈；右侧缺点：脆（分辨率／主题／字体）・无语义。底部两条工程建议：固定环境・只当兜底。

提问：图像识别方案为什么"看着最通用，实际最难维护"？

预设回答：

- 答"因为要不停重新截图"——这是最直观的痛：界面一改就得重截，工作量全在维护上。

- 答"因为它只认图案不认语义"——更本质：它没法判断"这个按钮是不是禁用""文案对不对"。

- 答"因为换了机器分辨率就全废"——很实际，这也是为什么必须先锁环境。

- 收口：所以它的定位是"**死角补丁**"，不是主力方案——用对了很救命，用错了很折磨。

易错点 / 考点：① 图像识别方案的适用前提（拿不到控件树）；② SikuliX → SikuliNG 的演进与服务化（REST API）；③ 优缺点对照（通用 vs 脆、无语义）；④ 常考判断："图像识别是最通用的 UI 自动化方案，应优先选用"——错，维护成本最高。

过渡：Web 讲完了，把同样的三要素搬到手机上，会遇到新的麻烦。

【9.42】移动 App UI 自动化测试框架（1 分钟）

这一页是**移动 App UI 自动化**的分隔，课件点了两个名字：**Appium** 和 **Airtest**，代表两条路线。

先讲移动端为什么比 Web 难：**设备多**（品牌、尺寸、分辨率）、**系统多**（Android 版本差异大，iOS 权限模型不同）、**干扰多**（来电、通知、弹窗）。

再加两条工程约束：**真机要管**（设备农场、充电、USB 稳定性、并发占用）、**安装包要管**（每版都要装，还要处理覆盖安装与卸载残留）。

所以移动自动化的重心往往不在写脚本，而在**环境与设备管理**——这也是第 43 页对比表要把"支持平台"和"设备管理"列成重要维度的原因。

记两条路线就够：**Appium 走控件识别（复用 WebDriver 协议），Airtest 走图像识别、并能用 Poco 走控件**。

板书：中间写"移动 UI 自动化的难点"；左侧三个"多"：设备多・系统多・干扰多；右侧两条工程约束：真

机要管·安装包要管；底部两条路线：Appium（控件·WebDriver 协议）｜Airtest（图像 + Poco 控件）。

提问：手机上跑自动化，脚本挂了，你怎么区分"产品 bug"和"被弹窗打断"？

预设回答：

- 答"看截图"——最直接，失败时保留截图是移动自动化的必备能力。

- 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工程纪律是：**先处理干扰，再执行主流程**——把弹窗、更新提示做成前置的"清场"动作。

易错点 / 考点：① 两个框架的名字与路线差异；② 移动端三个"多"；③ 常考判断："移动自动化就是把 Web 脚本搬到手机"——错，设备与干扰管理是主要工作量；④ 记住"失败留截图"是必备实践。

过渡：选型之前先看一张总表，把几条路线放在一起比。

【9.43】几款常见的测试工具比较（3 分钟）

这一页是**常见测试工具的比较表**。课件只给了表，我把"该比哪几列"讲清楚，你以后看到任何一张工具对比表，都知道怎么读。

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，但也往往意味着每一块都不够深。

第二列，**定位方式**：控件级（id／xpath／控件树）、图像级、还是代码级（直接调用页面脚本）。定位方式决定了**稳定性上限**——控件级最稳，图像级最脆。

第三列，**语言与生态**：能不能用你团队熟悉的语言写，社区活跃度如何，出问题有没有人答。这一列最容易被忽略，却最影响长期成本。

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。

把这几列填进同一张表，你会发现一个规律：**没有全能冠军，只有最合适的组合**——Selenium 覆盖广但时序要自己管；Cypress 反馈快但受限于前端场景；Appium 复用 WebDriver 生态但依赖设备管理；Airtest 通用

（连游戏都能测）但图像匹配脆。

所以选型方法给你五问：**我的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？谁来维护三年后？** 五问答完，答案基本自己浮出来了。

板书：表格五列：支持范围｜定位方式｜语言与生态｜并行与设备管理｜学习与维护成本。底部五问：形态？语言？矩阵？失败率？谁维护三年后？右侧一句："没有全能冠军，只有最合适的组合"。

【思政落点 3】：选型这一页讲的是"因事制宜"。行业里最常见的浪费，是用最流行的工具去解最不匹配的问题——明明团队是 Java 栈，偏要上 JavaScript 的框架；明明只有 Web，非要买一套移动设备农场。工程技术没有绝对先进，只有**与场景匹配**。先摸清自己的条件，再决定用什么武器，而不是让工具牵着项目走。

提问：如果团队只有两个人，还要维护三年，你会怎么选？

预设回答：

- 答"选最流行的"——流行意味着资料多，但也意味着坑被别人踩过、答案好找；不过两个人的团队最怕"学不动"。

- 答"选团队已经会的"——很务实的答案。**人力是最贵的成本**，两个人的团队没有余量去试错。

- 答"选生态稳、文档全、能进 CI 的"——也对，因为两个人最需要"自动化帮你干活"，而不是"你伺候自动化"。

- 收口：所以小团队选型的第一原则是**降低总拥有成本**，不是追求技术先进。

易错点 / 考点：① 比较的五个维度（简答可按此作答）；② 各工具的能力边界（Selenium／Cypress／Appium／Airtest）；③ 常考判断："工具越新越好"——错，看匹配度与维护成本；④ 记住选型五问，案例题很好用。

过渡：先看 Appium 这条"复用 WebDriver 协议"的路。

【9.44】Appium（2 分钟）

这一页是 **Appium**，定位一句话：**把 WebDriver 那套协议搬到移动端**。

结构三层：**客户端**（Java／Python／JS 写的代码，加一堆能力参数——平台、设备名、要装的包、自动化引擎）、**Appium 服务端**（接收命令、按平台分派）、**平台驱动**（Android 走 UIAutomator2，iOS 走 XCUITest，把 WebDriver 命令翻译成系统原生调用）。

好处直接：**一套 API 覆盖双端**，团队有 Selenium 经验的话迁移成本很低。定位方式是**控件级**——拿的是控件树（Android 视图层级、iOS 无障碍树），稳定性远高于图像识别，前提是开发给了有意义的标识。

成本也在结构里：**要装服务端、配驱动、管设备**。链路是"测试代码 → Appium 服务 → 设备驱动 → 真机／模拟器"，任何一环不稳都会红，所以通常放进设备农场或容器里跑。

一句提醒：**Appium 不管设备，只管驱动**——设备分配、安装包版本、并发占用要靠外部调度。

> **板书：**三层链路：客户端（capabilities：platformName・deviceName・app・automationName）→ Appium 服务端 → 平台驱动（UIAutomator2／XCUITest）→ 设备。右侧两行："一套 API 覆盖双端（复用 WebDriver 协议）"、"控件级定位，稳定性高于图像"；底部："Appium 管驱动，不管设备"。

提问：为什么说 Appium 的稳定性主要取决于"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"可测性需求"：关键控件加 `content-desc`／`accessibilityIdentifier`，这比任何脚本技巧都有效。

易错点 / 考点：① 三层的名称（客户端／服务端／平台驱动）；② 两个平台驱动的名字（UIAutomator2、XCUITest）；③ capabilities 的常见键；④ 常考判断："Appium 自带设备管理"——错；⑤ 记住"复用 WebDriver 协议"这一点，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）

这一页是 **Appium 示例**，课件给了官方文档里"运行测试"那一节的地址。我不逐行写代码，讲**一个最小脚本的骨架**，你照着就能跑通第一个 hello world。

骨架分五段。**第一段，准备能力参数**：告诉服务端"我要在哪个平台、哪台设备、装哪个包、用哪个自动化引擎"。这一段决定了脚本能不能连上设备，写错了就是连接超时。

第二段，建立会话：客户端拿这些参数向 Appium 服务端请求一个会话，服务端去启设备、装应用、拉起驱动。这一步最慢，也最容易失败——设备没连上、包没装成、驱动版本不匹配，都会卡在这里。

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。不要断坐标、不要断颜色。

第五段，收尾：关闭会话、释放设备。这一步经常被忘，后果是**设备被占住**，下一个任务排队等——在共享设备的环境里，这是很常见的"莫名超时"来源。

板书：五段骨架竖排：① 能力参数（平台・设备・包・引擎）→ ② 建会话（最慢、最易失败）→ ③ 定位与操作 → ④ 断言（断业务结果）→ ⑤ 收尾（关会话、放设备）。右侧两条经验：显式等待・失败留证据（截图／结构／日志）。

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"应用没卸载干净，下次装包冲突"——也可能，尤其覆盖安装场景。

- 收口：所以收尾要写进模板：**建会话就必须配对关会话**，最好放在 finally 里保证执行。

易错点 / 考点：① 五段骨架的顺序（简答／代码题）；② capabilities 的作用；③ 建会话是最易失败的环节；④ 常考判断："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 走图像但也能走控件，下一页看它的三件套。

【9.46】Airtest 框架（3 分钟）

这一页是 **Airtest 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，**Airtest**：**跨平台的、基于图像识别的 UI 自动化测试框架，提供了一系列跨平台 API 可供调用，覆盖各种常见的操作**。注意两个词——"跨平台"（多平台同一套 API）和"图像识别"（走的是第 41 页那条路）。

第二个，**Poco：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**。这是关键差异：**支持微信小程序**，不少团队做小程序自动化就冲这个来。

第三个，**AirtestIDE：跨平台的 UI 自动化测试编辑器，能够快速编写 Airtest 和 Poco 代码，提供脚本录制、一键回放、报告查看等功能**。它是把前两者的能力装进一个图形界面，降低上手门槛。

这三个组件的关系，我打个比方：**Airtest 是"眼睛"（看图），Poco 是"手"（抓控件），AirtestIDE 是"工作台"**。三者可以混用——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。

课件给了官网 airtest.netease.com。它的强项在于**覆盖面广**：原生 App、小程序、游戏、桌面应用都能上；弱项和图像方案一样——**图像部分脆**，所以工程上要尽量把断言和关键操作放在 Poco 侧。

板书：三框并列：Airtest（图像识别・跨平台 API・覆盖各种操作）｜Poco（控件识别・Android／iOS 原生＋微信小程序）｜AirtestIDE（编辑器・录制・一键回放・报告）。底部比喻："Airtest 是眼睛，Poco 是手，IDE 是工作台"；右侧写"能用控件就用 Poco，控件没有了再上图像"。

提问：同一个操作，既可以用图像点击，也可以用 Poco 定位控件点击，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：图像是兜底手段，不是首选。

- 收口：判断标准还是那条——**断言的稳定性取决于你抓的是"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 常考判断："Airtest 只能做图像识别"——错，Poco 走控件。

过渡：看一段 Poco 的写法，你就明白"控件识别"和"图像识别"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）

这一页是**基于 Poco 开发脚本的示例**，课件给了官网地址。我讲它的写法逻辑，你对照文档就能跑。

Poco 的核心是**控件树查询**。它的入口是一个 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，然后你调它的方法：点击、长按、滑动、读属性。

所以一段 Poco 脚本读起来像什么？像"**按名字找人，然后让他做动作**"：找到"登录"按钮、点它；找到输入框、输入账号；读"提示文字"控件的文本、断言它等于预期。**没有坐标、没有截图、没有像素**——这就是它比图像方案稳的原因。

但它有个前提，必须讲清楚：**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，小程序靠它自己的节点结构。如果开发一个标识都没给，Poco 就只能靠层级路径硬猜——那就退化成"结构定位"，一改版就废。

还有一个工程细节：**Poco 要先"连上"界面**。它需要初始化与目标平台对应的 Poco 实现（Android 用 uiautomation2 那套、iOS 用另一套、小程序又有自己的），连不上就查不到任何控件。这一步失败的表现往往是"控件树是空的"，而不是报错——新手最容易在这里卡住。

一句话：**Poco 让脚本用"语义"说话，代价是你必须先有"语义"**——也就是可测性设计。

> **板书：**中间写"Poco = 控件树查询 + 动作 + 属性断言"；左侧示例流：连上界面 → 选择器找控件 → click／settext → 读属性断言；右侧前提与坑：控件要有可识别属性・要先初始化对应平台的 Poco・控件树为空＝没连上。

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"，而不是"树是空的"。

- 答"查有没有连上目标应用／平台"——对，树为空基本是连接问题：包名不对、应用没在前台、Poco 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：所以排查顺序是**先确认连上、再确认在前台、最后才怀疑选择器**。

易错点 / 考点：① Poco 的工作方式（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 常考判断："Poco 靠截图找元素"——错，那是 Airtest 的图像部分。

过渡：图像方案里有一类特殊能力绕不开——从图里读字。这就是 OCR。

【9.48】OCR 实现（2 分钟）

这一页讲 **OCR 的实现**，课件一句话点明结构：**目前 OCR 的实现一般分成两个步骤：文本检测和文本识别**。

这两个步骤的分工要分清。**文本检测：找到图片上文本的位置和区域（文本框）**——它回答"字在哪儿"。**文本识别：识别每个文本框中的文字**——它回答"这些字是什么"。先定位再识别，是因为整张图直接识别既慢又乱；先切出小框，识别器只需处理一小块，准确率也更高。

为什么 UI 自动化要关心 OCR？三个场景。**第一，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。**第二，验证码与图片验证**：文字被渲染成图，控件属性里拿不到。**第三，跨技术栈的兜底**：远程桌面、老客户端、嵌入式屏幕——只要能截屏，OCR 就能读字。

但要提醒一句：**OCR 是"读"的能力，不是"测"的能力**。它把像素变成字符串，后面还得靠断言去比对。所以 OCR 在自动化里的位置是**感知层**，判断层仍然是断言。

还有一条工程经验：OCR 的准确率受**分辨率、字体、颜色对比、背景干扰**影响极大。工程上宁可**先截小区域**（只截那个数字所在的框），也不要把整屏丢进去——框越小、越干净，识别越稳。

板书：两步流程：文本检测（找位置／文本框）→ 文本识别（框内文字）。左侧三个应用场景：画布类界面・验证码／图片・跨技术栈兜底；右侧两条工程经验：先截小区域・OCR 只是感知层，判断仍靠断言。

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"：模型不知道哪块是文字、哪块是背景，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"，难度骤降。
- 答"因为要拿到位置，才能去点击"——很实用的一点：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**——这正是自动化需要的。

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 两者的先后顺序不能颠倒；③ OCR 在自动化中的定位（感知层）；④ 常考判断："OCR 可以直接替代控件定位"——错，它是兜底手段，且只能拿到文本、拿不到状态。

> **过渡：**这两步在工程上是怎么串起来的？下一页是完整的流水线。

【9.49】OCR 实现（3 分钟）

这一页是 OCR 的**工程流水线**，检测与识别各自的处理链都在这里。

检测的第一条路，是传统图像处理路线：多层 MSER 检测做**文本区域特征提取**（MSER：最大稳定极值区域）→ **特征过滤（CNN）**把不像文字的块筛掉 → **特征合并（规则）**把小块按规则拼成候选文本行 → **文本行过滤（CNN）**再筛一遍 → **多层结果合并（NMS + 规则，相邻及重叠文本框合并）**。

检测的第二条路，是深度学习路线：AdEAST 检测直接输出**文本区域特征图** → **后处理（框解析 + NMS）**得到最终文本框。

识别分三档：常用字识别（DenseNet + CTC）、繁体字识别（DenseNet + CTC）、高精度识别（DenseNet + BLSTM + CTC）。三档差别在于要不要上下文：DenseNet 提特征，CTC **免对齐**地把特征序列映射成字符序列——不用标出每个字的位置；高精度档再加 **BLSTM**（双向长短时记忆网络），让模型同时看左右上下文，对形近字更准，代价是更慢。

课件还列了几个开源实现（CTPN＋DenseNet＋CTC 的中文 OCR、CRNN 系列）；CRNN 是"卷积＋循环＋CTC"的统称。

做自动化取字，优先用现成的"检测＋识别"能力，别自己训模型——你的价值在"取哪块区域、断什么内容"，不在模型本身。

> **板书：**两行流程。检测①：多层 MSER（特征提取）→ 特征过滤（CNN）→ 特征合并（规则）→ 文本行过滤（CNN）→ 多层结果合并（NMS＋规则，相邻／重叠框合并）；检测②：AdEAST（特征图）→ 后处理（框解析＋NMS）。识别：常用字（DenseNet＋CTC）・繁体（DenseNet＋CTC）・高精度（DenseNet＋BLSTM＋CTC）。右侧术语：CRNN・BLSTM・CTC・MSER。

> **提问：**CTC 在 OCR 里解决了什么问题？

> **预设回答：**

- 答"提高准确率"——太笼统。

- 答"解决了对齐问题"——正确。不需要把每个字符和图像位置一一标注，模型输出序列后由 CTC 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 CTC 让 OCR 的训练数据只需要"图＋整行文字"，不需要"每个字的坐标"，这是它大规模可用的前提。

易错点 / 考点： ① 检测的两条路线与各自步骤（简答／填空高频）；② NMS 的作用（去重、合并重叠框）；③ 识别三档的名字与差别（是否加 BLSTM）；④ 术语全称：CRNN（卷积循环神经网络）、BLSTM（双向长短时记忆网络）、CTC（连接时序分类）、MSER（最大稳定极值区域）；⑤ 常考判断："OCR 识别不需要检测"——错，两步分工不同。

过渡： 到这里，单点的自动化能力都讲完了。最后一节把它们装进一个框架。

【9.50】自动化测试集成框架（1 分钟）

这一页是**自动化测试集成框架**这一小节的分隔。我先交代它要解决什么——前面讲的全是"零件"：静态分析、单元、接口、UI。

零件齐了，问题才刚开始。你会遇到四件事：**脚本放哪儿、谁来跑、跑完结果给谁看、环境谁来管**。四个团队各写各的脚本，最后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。

这一小节我按四个问题走：**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、脚本语言、代理、工具、调度、报告）、**框架应该提供哪些服务**、**设计时要抓哪些要点**（控制中心、多层次、分布式、对象库、关键字库、数据共享）。

记一句话：**集成框架不是"更高级的工具"，它是"让自动化可持续的工程结构"**。

板书： 中间写"集成框架＝把零件装成产线"；左侧四个现实问题：脚本放哪・谁来跑・结果给谁看・环境谁管；下方四个主题：脚本组织（KD・数据分离）→ 部件构成 → 提供服务 → 设计要点 → 以 RF 为实例。

提问： 团队的单元测试、接口脚本、UI 脚本各用一套工具，最直接的麻烦是什么？

预设回答：

- 答"要学三套东西"——是麻烦，但不是最致命的。

- 答"报告没法汇总，没人说得清这次发布到底测了什么"——这才是最致命的：**质量结论无法收敛**。

- 答"资源重复投入，机器和设备各占一套"——也很实际，成本翻倍。

- 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 vs 能力）；③ 常考判断："集成框架必须有商业产品"——错，开源可拼，也可自研；④ 记住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）

> 这一页是 **Selenium 关键字驱动**。什么叫关键字驱动？一句话：**把"做什么"从代码里抽出来，变成一行行的"关键字＋参数"**。
>
> 这样做的好处有三个。**第一，脚本与实现分离**：业务人员看得懂表格，不用看代码；改流程改表格，不动引擎。**第二，复用**：同一个"点击"关键字，被几千条用例共用，改一次全生效。**第三，可组合**：关键字可以套关键字，形成"业务关键字"——比如"登录"这个关键字内部由五个基础关键字组成。
>
> 代价也要讲清楚。**第一，引擎复杂**：你要写解析器、执行器、对象库、日志，工程量不小。**第二，调试难**：出问题在表格层看不出来，得进引擎里查。**第三，表达能力受限**：遇到复杂逻辑（循环、条件、异常），表格会变得很别扭，最后还是得写代码。
>
> 所以工程上的常见做法是**混合**：**关键帧用表格（稳定、可读），复杂逻辑用代码（灵活）**——这就是"混合驱动"这个名字的由来。课件在 Selenium 上演示，是因为它的 API 天然适合封装成"动作词"。

> **板书：** 左画一张三列表：关键字｜对象｜数据（例：打开浏览器｜—｜Chrome；输入｜用户名框｜admin；点击｜登录按钮｜—；断言文本｜提示区｜登录成功）。右列三利：分离・复用・可组合；三弊：引擎复杂・调试难・表达受限。底部："混合＝关键帧用表格，复杂逻辑用代码"。

> **提问：** 一条用例里有"如果登录失败就重试三次"这种逻辑，你放在表格里还是代码里？

> **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"很快就会把表格撑爆。

- 答"代码里，写成自定义关键字"——这是标准做法：**把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：所以关键字驱动的边界很清楚：**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（表格编排＋代码实现）；④ 常考判断："关键字驱动不需要写代码"——错，引擎与自定义关键字都是代码。

> **过渡：** 把这套思想做成一个成熟框架的，是 Robot Framework。

【9.52】RobotFramework 关键字驱动（2 分钟）

这一页是 **Robot Framework 的关键字驱动**——同一套思想，做成了完整框架。

组织形式是**表格文件**，四类区段：**Settings**（导入库、变量文件、前后置）、**Variables**（定义变量）、**Test Cases**（用例，一行一步）、**Keywords**（用户自定义关键字）。

它比"Selenium＋自制表格"强在三点：**关键字分层**（库、资源、用户三层关键字）、**变量与数据管理内建**（天然支持一套脚本多套数据）、**报告与日志内建**（跑完自动出 HTML 报告，每步的关键字、参数、耗时、截图都在里面）。

定位上的差别：RF 是**通用的、ATDD 的自动化测试框架，易于使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、调试要靠日志、关键字设计不好会变成"关键字地狱"**。所以 RF 项目的成败往往取决于关键字库设计——这正是第 57 页的"设计要点"。

> **板书：** 四区段：Settings（导入库・变量文件・前后置）｜Variables｜Test Cases（一行一步）｜Keywords（用户关键字）。右侧三层复用：库关键字 → 资源关键字 → 用户关键字。底部两句："ATDD：先写验收用例再写实现"、"关键字设计不好＝关键字地狱"。

> **提问：** RF 项目最容易失败在哪里？

> **预设回答：**

- 答"安装配置"——一次性的，不算真问题。

- 答"关键字太多太乱，没人知道该用哪个"——这是最典型的失败模式，"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该是**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 常考判断："RF 只能做 UI 测试"——错，它靠库扩展做接口、数据库、桌面等；⑤ 记住"关键字地狱"这个典型风险。

> **过渡：** 关键字驱动的底层逻辑，其实是"数据与脚本分离"。这一页专门讲它。

【9.53】分离数据与脚本（3 分钟）

这一页讲**分离数据与脚本**，课件给的原话是：**使用表（Table）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，我逐个拆。

第一个，**测试输入**。用例要的数据不能写死在脚本里，要放到表或数据文件里。为什么？因为**同一个流程往往要用几十组数据跑**——正常的、边界的、异常的。写死在脚本里，你要复制几十份脚本；放到数据文件里，脚本一份就够。

第二个，**可验证的输出**。这一条最容易被忽略：**期望结果也要数据化**。很多团队只把输入抽出去了，期望结果还写在断言代码里——结果加一组数据就得改一段代码，等于没分离。正确做法是数据文件里明确写"这组输入的期望输出是什么"，断言只负责比对。

第三个，**测试环境设置和控制**。连哪个环境、用哪个账号、开什么开关，这些也不该硬编码。放到配置文件里，一套脚本就能在开发、测试、预发三套环境上跑。

这三样合起来，就是**数据驱动（Data-Driven）**的完整含义：**脚本是引擎，数据是内容，配置是环境**。加用例只加数据行，不改代码——这是数据驱动唯一的价值判据。

再说它和关键字驱动的关系：**两者不冲突，常常叠加**。关键字驱动管"步骤怎么表达"，数据驱动管"数据从哪来"。你把关键字表格的每一行参数换成变量，再从数据文件读值——这就是"关键字＋数据"的混合驱动。

> **板书：** 三栏：测试输入（表／数据文件）｜可验证的输出（期望结果也数据化）｜环境设置与控制（配置文件）。底部公式："脚本＝引擎，数据＝内容，配置＝环境"；右侧一句："加用例只加数据行、不改代码＝分离成功"。

> **提问：** 把输入抽到 Excel 里，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，说明耦合还在。

- 答"不算，期望结果也得数据化"——正确，这才是完整的数据驱动。

- 答"看情况，期望结果简单的可以写死"——可以妥协，但要有意识：这是技术债，不是设计。

- 收口：判断标准很硬：**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（加用例不改代码）；③ 与关键字驱动的关系（可叠加成混合驱动）；④ 常考判断："数据驱动就是把数据放 Excel"——不完整，缺了期望输出与环境。

> **过渡：** 数据要放在文件里，那用什么格式、要满足什么要求？下一页专门讲。

【9.54】测试数据的管理（3 分钟）

这一页讲**测试数据的管理**。课件先给七种载体：

Excel、CSV、Property、XML、YAML、Database、JSON。**Excel 与 CSV** 适合表格化数据；**Property** 适合键值配置；**XML 与 YAML** 适合层级结构；**JSON** 是接口测试的事实标准，因为接口本身就是 JSON；**Database** 适合数据量大、多用例共享的场景。

然后是七条要求，这是重点。**处理业务实体／数据**：数据要能表达业务对象（订单、用户），不是散字段。**容易描述特定的需求**：看一眼就知道这条用例验什么。**读写操作容易（～测试框架）**：框架要能方便地读写。**容易修改／维护测试数据**：改数据不该要写代码。**如果需要，创建 DSL 来描述**：复杂到一定程度就该定义领域专用语言。**不能局限于输入**：期望输出、环境、前置条件都要能表达。**容易实现比对（Outputs）**：数据要方便与"实际结果"自动比对。

这七条的重心：**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，最后还是人工判断，自动化就断了。

再一条实践：**数据要能被"造"和"清"**。数据不是静态资产，会被用例改；数据管理必须包含准备与回滚，否则并行一跑就互相污染。

一句话：**测试数据是资产，不是附件**——它的质量直接决定可维护性。

> **板书：** 七种载体：Excel・CSV・Property・XML・YAML・Database・JSON（旁注：表格／键值／层级／

> 接口／大数据量）。七条要求竖排：业务实体・易描述需求・读写容易・易改易维护・可建 DSL・不局限于输入・易比对输出。右侧两条经验：好写 vs 好用・数据要能造也要能清。

提问：接口测试的数据，你会选 Excel 还是 JSON？为什么？

预设回答：

- 答"Excel，测试人员熟悉"——熟悉是优点，但嵌套结构（请求体里的对象数组）在 Excel 里表达很别扭。

- 答"JSON，和接口格式一致，能直接当请求体"——这是主流答案：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：所以选格式的标准是：**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 七种载体至少能列五种；② 七条要求的重心（好写＋好用，尤其"不能局限于输入"和"易比对"）；③ DSL 什么时候需要；④ 常考判断："测试数据只需要管输入"——错，期望输出与环境也算数据；⑤ 记住"数据要能造也能清"这条工程实践。

过渡：数据、脚本都有了，接下来是容器——一个 TA 框架由哪些部件组成？

【9.55】TA 框架的构成（3 分钟）

这一页是 **TA 框架的构成**，课件给六个部件：**Harness/IDE、脚本语言、代理（Agents）、工具（Tools）、任务安排（Scheduler）、报告（Report）**，外加被测对象 **SUT**。我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/IDE** 是开发与调试的工作台——Harness 负责加载脚本、组织执行、收集结果，IDE 是你写脚本的地方。

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，接收任务、启动工具、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。

这条链是：**脚本 → 任务安排 → 代理 → 工具 → SUT → 报告**。像不像生产线？**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，调度不合理机器闲着或挤爆，报告不清楚前面的努力没人看得见。

板书：环形流水：脚本（Script Language）→ 任务安排（Scheduler）→ 代理（Agents）→ 工具（Tools）→ SUT → 报告（Report）；左侧竖写"Harness／IDE＝开发与调试工作台"。底部："每一环都是

瓶颈候选"。

提问：六个部件里，哪一个最容易被团队忽略，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：报告只给一堆日志，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，自动化就退化成"半自动"。

- 收口：我的答案是**报告与调度**这两个"非技术"部件——它们不产生测试能力，但决定自动化能不能持续被使用。

易错点 / 考点：① 六个部件的名称（简答高频，英文名也要能写）；② 各部件职责不要混：Scheduler 管"何时跑"，Agents 管"在哪跑"；③ Harness 与 IDE 的区别；④ 常考判断："框架就是工具"——错，工具只是六分之一。

过渡：部件清楚了，再看一个成熟框架应该提供哪些"服务"。

【9.56】TA 框架应提供的服务（3 分钟）

这一页是 **TA 框架应提供的服务**，课件列了八条。我把它们分成三组，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。这一条听起来平淡，但它决定了"新人多久能上手""三个月后还找不找得到那条用例"。

第二组，管人干活：测试脚本开发调试（TIDE）、测试机／资源的管理、任务安排、测试执行启动与调度。这一组是"人怎么写、机器怎么跑"：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、按什么顺序、并发多少"。

第三组，管结果：系统监控、Log 收集、测试结果分析、测试报告查询。这一组决定"跑完之后能不能得出可信的结论"——注意这里有个关键：**监控与日志收集**必须与执行同步，事后补是补不出来的。

课件最后留了省略号——**这八条是底线，不是上限。**真实框架还会提供：环境自动部署、用例依赖管理、失败重试、与缺陷系统联动。

这八条里，最容易被做成"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。而自动化的价值恰恰在"让人更快下判断"，不在"生成更多数据"。

一句话：**框架的服务清单，本质是"把测试当工程来管"的清单**。

板书：三组竖排。管资产：测试件存储与管理。管干活：TIDE（开发调试）・资源／测试机管理・任务安排・执行启动与调度。管结果：监控与 Log 收集・结果分析・报告查询。右侧一句："八条是底线不是上限"；底部红字："结果分析最容易做成半成品"。

提问：八条服务里，哪一条做不好，会让"自动化"退化成"手工的电子版"？

预设回答：

- 答"任务安排"，没有调度就得人点 —— 对，这就是"半自动"的典型症状。

- 答"资源管理"，机器抢不到、环境不稳——也很关键，直接导致失败率飙升。

- 答"结果分析"，报告没人看得懂——最有洞察力：数据不等于结论。

- 收口：三条都致命，但优先级不同：**先解决"能自动跑"，再解决"跑得稳"，最后解决"看得懂"**。

易错点 / 考点：① 八条服务的名称（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 常考判断："框架只要能执行测试就行"——错，八条缺一不可；⑤ 记住"资源管理"与"任务安排"的分工。

过渡：知道了要提供什么，接下来是设计时抓什么。这是本节最关键的一页。

【9.57】TA 框架设计要点（3 分钟）

这一页是 **TA 框架设计要点**，课件给了九个关键词：**控制中心、多层次结构、分布式架构、通讯问题、接口定义、对象库、关键字库、数据共享、集成环境**。

先看**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**——从用例层、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展，多机、多浏览器、多设备并行。

再看**协作与契约**这三个：**通讯问题**——控制中心与代理之间怎么通信（消息、心跳、超时、断线重连），这是分布式框架最容易出故障的地方；**接口定义**——各层之间、各组件之间的接口要稳定，否则换一个实现就

牵一片；**集成环境**——与 CI、缺陷系统、测试管理平台的对接方式。

最后是**资产复用**这三个，也是最影响长期成本的：**对象库**——把所有元素的定位方式集中管理，界面改了只改一处；**关键字库**——把业务动作封装成可复用关键字，越用越省；**数据共享**——不同用例、不同层级能共用同一份数据定义，避免"每个脚本抄一份"。

九个要点背后是同一个原则：**把易变的东西集中起来，把稳定的东西抽象出来**。界面会变，所以定位收进对象库；业务会变，所以动作收进关键字库；环境会变，所以配置收进数据层。**框架的价值，就是把"变化"关进少数几个笼子里**。

板书：三组九词。结构：控制中心・多层次结构・分布式架构；协作：通讯问题・接口定义・集成环境；复用：对象库・关键字库・数据共享。底部一句："把易变的东西集中起来，把稳定的东西抽象出来"。

【思政落点 4】：对象库、关键字库、命名规范，讲到底是在为"后来的人"负责。你写脚本时图省事，把定位写死在三百条用例里，自己这周很爽，接手的人要改一次界面就得通读三百个文件。工匠精神不是把代码写得多漂亮，而是**让别人能接着干下去**：留清楚的分层、留可查的对象库、留能读懂的命名。这份自觉，就是职业素养的样子。

提问：为什么"对象库"能显著降低维护成本？不用它会怎样？

预设回答：

- 答"因为定位只写一次"——对，这是核心：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"——很准确，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：**界面改了你还是要改对象库**，只是改一处而不是改两百处。
- 收口：所以对象库不是"免维护"，是"把维护成本从 O(用例数) 降到 O(元素数)"。

易错点 / 考点：① 九个要点的名称（简答可按三组作答，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 常考判断："多层次结构会增加复杂度，所以小项目不该分层"——错，分层是为了隔离变化，规模小可以层少但不能混。

过渡：这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）

这一页讲**框架如何支持测试的设计、管理与执行**。上一页是"设计要点"，这一页是"功能落点"——把九个抽象要点，落到三个具体环节上。

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。它的产物是**可执行的测试件**，不是文档。这里有个观念要转过来：**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：管什么？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）。管理的目标是**让"跑什么"这件事可配置**——而不是每次都靠人回忆。

执行环节：调度、并发、重试、失败取证（截图、日志、页面结构）、结果回传。执行环节的设计目标是**可重复**——同一份代码、同一份数据，跑十次结论应当一致；不一致就要查原因，那是脆弱性的信号。

一句话：**框架支持的不是"跑测试"这一个动作，而是"测试这件事的全流程"**。

板书：三段竖排。设计：用例＝代码资产・步骤与元素分离・期望结果显式；管理：用例集・数据・资源・执行历史（可配置）；执行：调度・并发・重试・失败取证・结果回传（可重复）。底部："设计好→管理轻→执行稳→结果可信"。

提问：同一份脚本连跑三次，一次过两次失败，这说明框架哪个环节有问题？

预设回答：

- 答"执行环节不稳定"——表面看是执行，但根因可能在设计（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，尤其"单独过、一起挂"的那类。
- 答"管理环节：数据没隔离"——也很可能，并行或连续跑时数据被改。
- 收口：所以"不稳定"是**症状**，不是原因；要按执行→设计→管理倒着查根因。

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 失败取证的三个内容（截图、日志、页面结构）；④ 常考判断："执行阶段的问题一定出在执行代码里"——错，常见根因在设计与管理。

过渡：把框架放大到"多类型、多对象"的规模，就需要统一规划。

【9.59】统一规划:不同的测试类型和对象（3 分钟）

这一页是**统一规划**，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：**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**——脚本调库里已有的方法。**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，出统一格式报告。**Load Test Engine** —— **Based on Platform Type**，按平台类型划分引擎：**Mobile Engine、Application Engine、Web Engine**。

还有 **Composited Tested System**（组合的被测系统）——出现两次说明同一系统既被功能测试驱动、也被负载测试驱动。

统一规划的核心不是"用同一个工具"，而是"共用同一套资产"——同一份数据、同一份报告格式、同一个执行入口，这样加一类测试，成本才低。

> **板书：**中心写"Driven Script-Execution Module（统一执行入口）"；下方四个支撑：Functional Libraries（脚本表达式调库中已有方法）・Classified Test Data・Test Report Generator（接 Test Suite Batch Execution）・Load Test Engine；再下一行"按平台类型的引擎：Mobile／Application／Web Engine"；两侧标 Composited Tested System（被测系统，被功能与负载两条路径驱动）。

提问：为什么"统一报告格式"比"统一测试工具"更重要?

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：所以统一规划的抓手是**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／简答常考）；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 常考判断："统一规划意味着全公司用同一个测试工具"——错。

过渡：这些部件里有两个最"接地气"的——数据和对象识别，下一页专门看。

【9.60】数据管理与对象识别组件（2 分钟）

这一页讲**数据管理与对象识别组件**。课件画得很简：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。

所以框架必须有**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——意义是**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离。**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——这就是第 57 页的对象库。

一句话：**数据组件管"喂什么"，对象组件管"对谁动手"**，两者都是脚本的左膀右臂。

板书：左写"Classified Test Data（分类测试数据：按实体／场景／环境；复用＋隔离）"，右写"Test Object identification（对象识别：集中定位、对外给逻辑名）"，中间写"Scripts and Methods ← need test data or test object or both"。底部："数据管喂什么，对象管对谁动手；两者对外接口要稳定"。

提问：如果一个框架只有对象库、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，尤其没有隔离机制时。

- 答"不影响，数据放 Excel 就行"——Excel 是载体，但没有**分类与取用接口**，复用和隔离还是靠人自觉。

- 收口：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"**。

易错点 / 考点：① 两个组件的名称与职责；② "脚本需要数据、对象或两者"这句的三种情况要能举例；③ 数据分类的两个目的（复用、隔离）；④ 常考判断："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：跑完之后，最后一道关口是报告。下一页看报告组件是怎么搭的。

【9.61】测试报告组件（2 分钟）

这一页是**测试报告组件**，课件给了四个环节：**Test Plan（测试计划）**、**Test Script Execution Log（脚本执行日志）**、**Test Report Generator（报告生成器）**、**HTML Report Template（HTML 报告模板）**。这四个是一条数据流，我顺着走一遍。

Test Plan 是"要测什么"：哪些用例、什么顺序、什么环境、判定标准。它是报告的**骨架来源**——报告里的"应测多少条"就来自这里。

Test Script Execution Log 是"实际发生了什么"：每条用例开始结束时间、每步动作、断言结果、异常堆栈、截图。它是报告的**事实来源**，也最容易被做烂——日志写得含糊，后面所有分析都是猜。

Test Report Generator 是"聚合与格式化"：把计划和日志合起来算通过率、归类失败、生成趋势。它的价值在**聚合**，不在排版。

HTML Report Template 是"长什么样"：决定信息怎么呈现给人看。

板书：四段流程：Test Plan（应测什么）→ Execution Log（实际发生什么）→ Report Generator（聚合与计算）→ HTML Template（呈现）。底部两层原则："报告先给结论，再给聚类，最后给证据"；红字："读完报告还要翻日志才知道发生了啥＝报告失败"。

提问：一份自动化报告，最重要的三样信息是什么？

预设回答：

- 答"通过率和失败列表"——必要，但不够：光有失败列表，人还是得一条条看。

- 答"通过率、失败聚类、失败证据（截图／日志）"——这是完整答案，正好对应分层呈现。

- 答"执行时间和机器占用"——工程上确实重要，属第四层信息。

- 收口：判断报告好不好，就看它能不能支撑"**发不发版**"这个决策。

易错点 / 考点：① 四个环节的名称与顺序（简答高频）；② 报告生成器的价值在"聚合"而非"排版"；③ 分层

呈现原则（结论→聚类→证据）；④ 常考判断："报告越详细越好"——错，信息过载会让人放弃阅读。

过渡：抽象讲完，看一个具体对比——传统报告和用户化报告差在哪。

【9.62】实例：自动生成用户化的报告（3 分钟）

这一页是**自动生成用户化报告的实例**，课件给了两句很实在的对比：**传统的自动化测试报告可读性很差，不能直观地体现整个测试过程**；而**用户化的测试报告可以非常直观地充分还原整个测试过程，极大提升测试结果的分析效率，降低分析难度**。

我把这两句话翻译成工程语言。**传统报告的问题**，本质是"信息密度错配"：它把原始日志直接铺出来——几千行"步骤 12 通过、步骤 13 通过"，真正的失败淹没在中间。人读它需要自己聚合，而人最不擅长的就是在大段文本里做统计。

用户化报告做的是四件事。**第一，结构化**：按用例、按模块、按失败类型分组，一眼看到"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，让人用眼睛就能判断。**第三，可还原**：失败那一步的截图、当时的页面结构、请求响应都留在原位，不用去别处找。**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。

注意第三点"可还原"为什么关键：**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。如果报告只告诉你"失败"，你就得重跑一遍去复现；如果它能还原现场，你三十秒就能判断"这是弹窗挡住了"。**这就是报告对效率的真实贡献**。

所以做报告别追求"功能多"，追求**"判断快"**。一份好报告的标准很朴素：**新人拿到它，不用问人，也知道该从哪儿查起**。

板书：左右对比。传统：原始日志铺满・失败被淹没・需人工聚合。用户化：结构化（按模块／失败类型分组）・可视化（通过率・耗时・趋势）・可还原（截图・页面结构・请求响应）・可下钻（总览→用例→步骤→证据）。底部："失败有一半不是缺陷，报告要能还原现场"。

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，机器算完就完了。

- 答"因为失败要人判断原因，判断需要证据"——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应当是**证据链**，而不是仪表盘。

> **易错点 / 考点：** ① 传统报告与用户化报告的差别（课件原文两句话）；② 用户化报告的四个特征；③ "可还原"与"可下钻"的区别：一个给证据，一个给路径；④ 常考判断："报告只要能看通过率就够"——错。

> **过渡：** 抽象讲完，落到一个真实框架上。先看 Robot Framework 的整体定位。

【9.63】Robot Framework（3 分钟）

> 这一页是 **Robot Framework**，课件一句话定性：**一个通用的、ATDD 的自动化测试框架，易于使用表格来组织测试过程和测试数据**。这句话里有三个关键词，我逐个拆。
>
> **"通用的"**：它不绑定任何一种被测对象。Web、接口、数据库、桌面、移动，全靠扩展库接进来。所以它是"框架的框架"——提供一个统一的组织与执行方式，具体能力外包给库。
>
> **"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。这个流程的价值不在技术，在**沟通**——把"到底要什么"提前吵完，而不是等测出来再吵。
>
> **"易于使用表格来组织测试过程和测试数据"**：表格是它的表达形式，一行一步，业务人员能读。课件旁边还标了 **Tools** 与 **Data** 两个词——意思是它把"工具"和"数据"分开管：工具通过库引入，数据通过变量与数据文件引入，两者都不写死在用例里。
>
> 它的工程价值，我总结三条：**入口统一**（一个命令跑所有类型的测试）、**报告内建**（不用自己写报告模块）、**扩展容易**（写一个 Python 库就能接入新能力）。
>
> 边界也要说清楚：**适合"流程清晰、步骤可枚举"的测试**；**不适合"需要复杂逻辑与算法"的测试**——那种场景，回代码里写更省事。这也解释了第 52 页说的"关键字地狱"：不是 RF 不好，是用错了地方。

> **板书：** 中间写"Robot Framework ＝ 通用 ＋ ATDD ＋ 表格驱动"；左侧三关键词解读：通用（不绑被测对象・靠库接能力）｜ATDD（先写验收用例・三方共识）｜表格（一行一步・业务可读）。右侧 **Tools／Data 分离**；底部三条工程价值：入口统一・报告内建・扩展容易；右侧红线："适合流程清晰的测试，不适合复杂算法逻辑"。

提问：RF 说自己"通用"，可它的核心是表格。表格怎么做到"通用"？

预设回答：

- 答"因为表格能写任何步骤"——表面看是这样，但真正的机制在库：表格里写的是关键字，关键字由库实现。

- 答"因为它靠扩展库把能力接进来，表格只负责编排"——正确。通用性来自"编排与实现分离"。

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

- 收口：所以"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 RF 的三个定性词（通用、ATDD、表格组织）；② Tools 与 Data 分离的含义；③ 适用边界（流程清晰 vs 复杂逻辑）；④ 常考判断："RF 只能做 Web UI 测试"——错，靠库扩展覆盖多类型。

过渡：它的报告长什么样？这是它最受人称赞的一点。

【9.64】RF 报告（2 分钟）

这一页是 **RF 报告**。RF 最被称道的一点就是"报告不用自己写"——跑完自动生成，而且结构是经过设计的。

一份 RF 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。你打开它，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，失败的那一步高亮，还挂着当时的截图或错误信息。

这两个视图的分工，正好对应上一页讲的"分层呈现"：**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方式**——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。

还有一层设计值得学：**报告的层级与"关键字层级"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。这意味着**关键字设计得好，报告自然就好读**——业务关键字"登录成功"在报告里显示成一步，而不是二十步底层操作。这也是为什么第 57 页强调"关键字库"是设计要点：**它不只影响可维护性，还影响可读性**。

板书：两类视图：汇总（总量・通过率・分组统计・耗时 → 给判断）｜详细日志（逐个关键字・参数・耗时・状态・失败高亮・截图 → 给归因）。中间箭头："同一份数据两种展开，可互相跳转"。底部两条：报告层级与关键字层级对齐・报告要归档做趋势对比。

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"——只是一半：命名清楚是必要条件。

- 答"因为报告的层级跟着关键字走，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：所以写关键字的粒度要按"业务可读"来定，不是按"技术实现"来定。

易错点 / 考点：① 两类视图的分工；② 报告与关键字层级的对应关系；③ 归档与趋势对比的实践；④ 常考判断："RF 报告需要自己开发"——错，内建。

过渡：报告之所以能自动生成，靠的是标准库。它们是 RF 的地基。

【9.65】RF 自带的标准库（3 分钟）

这一页是 **RF 自带的标准库**，课件给了官网清单地址。标准库的意义是：**装完就能用，不用额外找库**。我按用途分四组，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。这是所有用例都会用到的一组，等于"语言运行时"；它同时提供**断言**与**日志**，所以哪怕什么都不装，也能写出"输入—断言—记录"的完整测试。

第二组，数据处理：**Collections**（列表与字典）、**String**（字符串）、**DateTime**（日期时间计算与格式转换）。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**（文件目录、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**（报文解析校验）、**Telnet**（连远程设备）。

这里有个概念考试常考：**标准库随框架一起发布**，而 **Selenium、Appium、Database、HTTP、Image、Sikuli 这些是扩展库**，要单独装。**标准库解决"基本功"，扩展库解决"接什么系统"**——这个分工记住了，搭环境时就不会找错东西。

板书：四组竖排：基本功（BuiltIn：断言・变量・日志・循环・跳过）｜数据（Collections・String・DateTime）｜系统（OperatingSystem・Process）｜界面与外部（Screenshot・XML・Telnet）。右侧

> 分界："标准库＝随框架发布；扩展库＝Selenium／Appium／Database／HTTP／Image／Sikuli，需另装"。

> **提问：**一条用例要断言"优惠券三天后过期"，你会用到哪两个库？

> **预设回答：**

- 答"BuiltIn 就够，直接断言"——断言用它没错，但"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：所以测试里的"时间"永远要用日期库算，不要手拼字符串——这是很常见的隐藏坑。

> **易错点 / 考点：**① 标准库的分组与代表库名（简答高频）；② BuiltIn 的定位（断言＋日志＋控制流）；③ 标准库与扩展库的分界（Selenium、Appium、Database、HTTP 属扩展库）；④ 常考判断："Database 是 RF 标准库"——错，属扩展库。

> **过渡：**标准库之外，RF 还有插件与扩展两层生态。先看插件。

【9.66】RF 插件（2 分钟）

> 这一页是 **RF 插件**。在 RF 的语境里，"插件"主要指**工具链层面**的扩展，不是测试能力本身，我分三类讲。
>
> **第一类，编辑器／IDE 插件**。RF 的脚本是纯文本表格，所以各种编辑器都能通过插件获得"语法高亮、关键字补全、跳转到定义、直接运行、看结果"的能力。这一类插件不改变测试能力，但极大改变**写脚本的体验**——有没有补全，效率差得很远。
>
> **第二类，监听器（listener）这类运行时扩展**。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：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。这一类是**框架层的钩子**，让你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。
>
> 这三类的共同点：**它们都不改变"测什么"，只改变"怎么用"**。所以选插件时的标准是"是否减少摩擦"，不是"是否功能多"。

> **板书：**三类插件：编辑器／IDE（高亮・补全・跳转・运行）｜运行时钩子（listener：事件点插入日志、截图、上报）｜报告与集成（推送到测试管理／缺陷／CI 看板）。底部一句："插件不改变'测什么'，只改变'怎么用'"；右侧提醒："插件版本要与框架版本一起锁定"。

提问：为什么"监听器"这种扩展方式比"改框架源码"更好？

预设回答：

- 答"因为不用改源码"——对，升级框架时不会被自己的改动拖住。

- 答"因为它能在事件点插入逻辑，够灵活"——也对，钩子覆盖了大部分需求。

- 答"因为性能更好"——不是重点，甚至可能略慢。

- 收口：核心是**可升级性**：扩展点让你"加能力不动地基"，这是所有成熟框架的共同设计。

易错点 / 考点：① 三类插件的划分；② 监听器的作用与事件点；③ 插件版本与框架版本的耦合风险；④ 常考判断："插件能替代扩展库"——错，一个管工具链，一个管测试能力。

过渡：真正提供测试能力的是扩展库，下一页看清单。

【9.67】RF 扩展库（3 分钟）

这一页是 **RF 扩展库**，课件列了一串：

Selenium、AutoIT、Appium、Android、Database、Http、Image、Sikuli……我按"管什么"分类，你就能把它和前面几节对上号。

界面类：**Selenium**（Web UI，对应第 36 页那套）、**Appium**（移动原生，对应第 44 页）、**Sikuli**（图像识别，对应第 41 页）、**Image**（图像比对，做视觉回归）。这一组是把 UI 自动化能力接进 RF。

桌面类：**AutoIT**（Windows 原生窗口与控件操作，对应第 68 页要讲的一体化框架）。Web 测不到的系统对话框、"另存为"、证书弹窗，靠它。

移动类：**Android**（通过 adb 直接操作设备：装包、起应用、取日志）。它和 Appium 的区别要讲清楚：**Appium 走 UI 控件，Android 库走设备命令**——一个测界面，一个管设备。

数据与接口类：**Database**（连数据库做校验与造数）、**Http**（发 HTTP 请求，做接口测试）。这两个是接口层自动化的主力。

讲这一页的真正目的，是让你看清 RF 的架构价值：**它自己不实现任何测试能力，全部靠库**。好处是"一套编排方式，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。

板书：四类扩展库：界面（Selenium・Appium・Sikuli・Image）｜桌面（AutoIT）｜移动设备（Android：adb 装包起应用取日志）｜数据与接口（Database・Http）。右侧两条：Appium 走控件 vs Android 库走设备命令；底部纪律："够用就好，锁定版本"。

提问：已经有了 Appium，为什么有时还要装 Android 这个扩展库？

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，用 Android 库兜底"——不是主因，稳定性问题应靠工程手段解决。

- 答"重复了，装一个就够"——看需求：只测界面可以只装 Appium；但准备环境、收日志时还是会用到设备命令。

- 收口：所以选库的标准是"**这件事该在哪一层做**"——UI 的事交给 UI 库，设备的事交给设备库。

易错点 / 考点：① 扩展库与标准库的分界（本节最高频）；② 各库的对应能力（Selenium／Appium／Sikuli／AutoIT／Database／Http）；③ Appium 与 Android 库的分工；④ 常考判断："装得越多越好"——错，依赖膨胀是维护负担。

过渡：Web 和 Windows 两套界面混在一个业务里，怎么用一个框架管住？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）

这一页是 **Web 和 Windows 一体化自动化测试框架**。它解决的是很现实的一类系统：**一个业务流程里，既要点网页，又要点桌面客户端，还要处理系统对话框**。

做法是**分层接管**：**Web 层用 Selenium**（浏览器内的页面），**桌面层用 AutoIT 或系统 API**（原生窗口、菜单、对话框），**中间靠统一的脚本与报告层连接**。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。

三个工程要点必须提前处理。**第一，焦点与窗口句柄**：切换应用时焦点会变，脚本要显式激活目标窗口，否则输入会打到别处。**第二，时序**：桌面应用响应慢、弹窗出现时间不定，必须等窗口出现再操作，别靠固定

睡眠。**第三，环境隔离**：桌面应用依赖特定机器与本地配置，"换台机器就跑不了"是常态，要用镜像或专用执行机固定。

一句话：**一体化框架的价值不在"技术统一"，而在"流程统一"**——同一份报告里能看到网页步骤和桌面步骤，才叫端到端。

板书：三层接管：Web 层（Selenium）｜桌面层（AutoIT／系统 API）｜连接层（RF 表格编排 + 统一报告）。左侧场景举例：网页发起 → 系统"另存为"对话框 → 桌面客户端上传。右下三个要点：焦点与窗口句柄・时序（等窗口而非 sleep）・环境隔离（镜像／专用执行机）。

提问：为什么 Selenium 处理不了"另存为"对话框?

预设回答：

- 答"因为它是系统弹窗，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"可以用键盘模拟解决"——能绕，但不稳；正规做法是桌面层工具接管。

- 收口：所以判断"该用哪个工具"，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 能力边界（只管浏览器渲染的内容）；③ 三个工程要点（焦点、时序、环境隔离）；④ 常考判断："一个框架只能测一种界面"——错，一体化框架可以接管多层。

过渡：企业里做框架，往往不是从零写，而是先做一套模板。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）

这一页是**基于 SpringBoot 的高效模板化自动化测试框架**。核心思想一句话：**把"每条用例都要写的那些东西"做成模板，让新增用例只填业务部分**。

什么叫"每条都要写的东西"？五样：**依赖注入与配置加载**（连环境、读参数）、**日志与断言库**（统一格式与失败信息）、**HTTP 客户端与鉴权**（带 token、处理签名）、**数据准备与清理**（造数、回滚）、**报告与通知**（出报告、发消息）。这些是脚手架，每个项目都在重复写。

用 SpringBoot 的工程结构把公共能力做成基础设施——配置走配置文件、公共能力做成 Bean、用例只写"调接口＋断言"。

好处：**上手快**（模板即规范）、**一致性好**（日志、断言、报告全项目统一）、**易维护**（公共能力改一处、全项目生效）。代价是**模板会僵化**：覆盖不到的场景，要么改模板（影响所有人），要么绕过（产生例外）——所以模板必须留**扩展点**。

一句话：**模板化不是"限制自由"，是"把重复劳动一次性解决"**。

板书：中间写"模板化＝把每条用例都写的东西抽出来"；左侧五样脚手架：依赖注入与配置・日志与断言・HTTP 客户端与鉴权・数据准备与清理・报告与通知；右侧三好处：上手快・一致性好・易维护；右下提醒：模板会僵化，必须留扩展点。

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"——很常见：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"——这是结构性风险，必须靠扩展点解决。
- 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：三条其实是同一个问题的三面：**通用性与灵活性的平衡**。

易错点 / 考点：① 模板化的核心思想（抽公共、留业务）；② 五样"脚手架"内容；③ 模板化的代价与扩展点；④ 常考判断："模板化框架只适合大团队"——不一定，小团队更需要减少重复。

过渡：抽象讲完，看一个具体例子：用 RF ＋ Selenium 测一个页面。

【9.70】RF+Selenium 完成某页面测试（2 分钟）

这一页是 **RF ＋ Selenium 完成某页面测试**的实例，我把一条完整用例的骨架走一遍，你就知道表格驱动在真实场景里长什么样。

第一步，准备：在 Settings 区导入 Selenium 库、声明浏览器与地址这类变量；在用例前打开浏览器、设置隐式等待。

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 + 定位器 + 数据"，读起来就是自然语言。

第三步，断言：断言页面出现了预期元素或预期文本。这里要强调：**断业务结果**，比如"首页显示当前用户名"，而不是"某个 div 的 class 变了"。

第四步，清理：关闭浏览器。这一步必须写在用例的后置里，保证失败也会执行——否则浏览器进程残留，下次跑就抢资源。

第五步，报告：跑完自动生成报告，每一步的关键字、耗时、状态都在里面，失败那步带截图。

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＋报告）。三者齐了才算完整。

板书：五步：准备（导入 SeleniumLibrary・变量・打开浏览器）→ 操作（一行一动作：输入・点击）→ 断言（断业务结果，不断样式）→ 清理（后置关浏览器，失败也执行）→ 报告（自动生成，失败带截图）。底部三条"自动化"：执行・数据参数化・结果判定；右下："先跑通一条，再批量复制"。

提问：为什么"断言"这一步要放在清理之前，而且清理要放在后置里？

预设回答：

- 答"因为顺序不能乱"——顺倒是其次，关键是**失败也要清理**。

- 答"清理放后置，失败时也会执行"——正确，这是保证"跑多少次都干净"的关键。

- 答"断言放前面能看到结果"——对，断言必须在业务动作之后、关闭浏览器之前，否则页面没了就没法断。

- 收口：所以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；③ 清理必须放后置（保证失败也执行）；④ 常考判断："断言可以放在关浏览器之后"——错，页面已销毁。

过渡：全章的技术讲完了，我们把它收束成几句话。

【9.71】小结（3 分钟）

这一页是**本章小结**，课件给了六条：**全 SDLC、高度集成、跨平台、过程透明、可测试性保障、需求可执行、**

支持敏捷、CD、DevOps。我逐条落到你学过的内容上。

全 SDLC：自动化不是只在系统测试阶段做。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 与端到端在验收前，性能与安全在专项阶段——**它贯穿整个生命周期**，这是第 9 页那六类分类的真正含义。

高度集成：脚本、数据、调度、报告、环境要装进一套体系。这就是第 50 到 60 页那一大段讲的集成框架。

跨平台：Web、移动、桌面、嵌入式，一套编排通吃（第 67 页的扩展库就是干这个的）。

过程透明：谁跑了什么、什么时候跑的、结果如何，全都留痕。这是第 61、62 页报告组件的价值。

可测试性保障：这一条最容易被忽略，但它最深刻——**做自动化的过程，会反过来逼着产品变得可测**：加稳定标识、拆开紧耦合、把配置外置、把逻辑与界面分离。**自动化不只是"测"，它在改善"被测对象"**。

需求可执行：ATDD 那一层意思——验收用例写成可执行的，需求就不再是模糊的一段话，而是"跑得过或跑不过"的判据。

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

收束：**自动化测试的终极产物不是脚本，是"可重复的信任"**——每次改动之后，你敢不敢发。

> **板书：**六条竖排：全 SDLC・高度集成・跨平台・过程透明・可测试性保障・需求可执行；下一行"支持敏捷／CD／DevOps"。每条右侧标对应页段（9 页六类／50–60 集成／67 扩展库／61–62 报告）。底部一句："自动化的终极产物是可重复的信任"。

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：它改变的是产品本身，收益远超测试。

- 答"过程透明"——也很重要，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：所以做自动化要有一个"溢出效应"的意识：**你在测的同时，也在把系统改造得更可测**。

易错点 / 考点：① 六条的名称（简答高频，可作全章总结题）；② "可测试性保障"的双向含义（测产品＋改产品）；③ 常考判断："自动化只服务于测试团队"——错，它服务整个交付链。

过渡：最后留三个问题给你，它们比任何工具都值得想。

【9.72】提问与思考（3 分钟）

这一页是三道思考题，课件问得很准。我一个一个说，但**不给标准答案**——这三题只有"想清楚了没有"。

第一问：录制脚本轻松搞定，为何没有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**"，记录的是"点了这个位置、输了这个值"，而不是**意图**。意图没被记录，就无法维护：挪个位置、加个等待、换个账号，全废；而且录制脚本不可读、不可复用、不可参数化，天生是"一次性"的。它真正适合的场景是**快速探测**——用它摸清陌生系统的操作路径，然后手写成可维护的脚本。

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**（自研意味着养一个团队）、**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**——用开源做骨架，把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"抓住缺陷、缩短反馈"）、**分层错**（全押 UI，塔基没有）、**维护崩**（脚本脆、红灯长期挂着）、**组织缺**（没人负责、没门禁、没复盘）。**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。

> **板书：**三问三答要点：① 录制＝记录坐标不记录意图 → 不可维护，只适合探测；② 开源 vs 自研 → 三问：需求多特殊・有没有人养・差异化在哪 → 先开源后扩展；③ 效果不好四因：目标错・分层错・维护崩・组织缺（三条非技术）。

> **【思政落点 5】**：第三问尤其值得诚实面对。做自动化最常见的失败，不是工具选错，而是**不肯承认失败**：脚本天天红，报告照样发；复盘只说"环境不稳"，不说自己目标定错了。诚实是先承认"这套东西没起作用"，再查为什么。**复盘的价值不在追究谁，而在让下一次不再重演**；一个团队能不能进步，就看它敢不敢把问题说透。

> **提问：**如果让你给"自动化效果不好"排一个责任顺序，你把哪一条放第一？

> **预设回答：**

- 答"工具不行"——最常见的甩锅方向，但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，因为目标错了，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，关键是要能**拿出证据**：失败率多高、维护工时多少、抓了多少缺陷——用数据说话，复盘才不是空谈。

> **易错点 / 考点：**① 三道思考题本身就是论述题，答题要"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 vs 自研的三问框架；④ 效果不好的四类原因（其中三条属工程管理）。

> **过渡：**思考题留给你们，动手的部分在实验里。看实验 6 要做什么。

【9.73】实验 6 部署自动化测试框架（3 分钟）

这一页是**实验 6：部署自动化测试框架**，课件给了六步，我按"装—配—跑—集成—扩展—交付"讲，并交代验收标准。

第一步，下载安装 Robot Framework：能跑通命令行、能执行一个最小用例。**第二步，安装 RF 的 GUI 界面**：编辑器＋运行＋看报告，让写脚本不必靠记事本。

第三步，安装第三方库（RF 插件），课件点了四个：**SeleniumLibrary**（Web UI）、**HTTP RequestsLibrary**（接口）、**AppiumLibrary**（移动）、**RESTinstance**（接口另一种风格，偏 JSON 校验）。验收标准是四个库都能 import 成功，且各跑一个最小示例。

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 能不能写出四段齐全（前置、动作、断言、清理）的用例。**第五步，结合实验 3 实现 RF 集成，生成测试报告**——这是本实验重点：把前面实验的测试接进 RF 统一编排。**第六步，基于 RF 和第三方库完成接口测试脚本，生成报告**，断言状态码与业务字段。

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）。我最看重第三样——**实验报告不是截图堆砌，要写清"我遇到的坑和怎么解决的"**，这才是工程能力的证据。

> **板书：**六步：① 装 RF ② 装 GUI ③ 装四个库（SeleniumLibrary・HTTP RequestsLibrary・AppiumLibrary・RESTinstance）④ 内建库写关键字脚本 ⑤ 结合实验 3 集成并出报告 ⑥ 接口测试脚本＋报告。右下验收三样：能跑・能读・能交（版本库＋说明文档）。

> **提问：**第三步装四个库，为什么要一次装完，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"应该用哪个装哪个，先跑通再逐步加"——更稳的工程做法：先最小可用，再逐步扩展。

- 答"一次装完便于验证环境完整性"——也有道理，尤其课程实验要求统一环境。

- 收口：两种做法都行，关键是**装完要逐个验证 import 与最小示例**，别等到写用例时才发现库不可用。

易错点 / 考点：① 六步的名称与顺序（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 常考判断："实验报告只需贴通过截图"——错，要写清问题与解决过程。

过渡：想继续深入，这四本书是我建议的路径。

【9.74】学习资源推荐（2 分钟）

这一页是**学习资源**，课件给了四本，我按"先学哪本"给你排个序，并说明每本解决什么问题。

第一本，《从零开始学 Selenium 自动化测试》（机械工业出版社，2020）。适合入门：把 WebDriver 的常用 API、定位方式、等待机制讲透。你要是连"显式等待"都还没用熟，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）。这本对本章最直接——关键字设计、库的使用、报告与集成，正好对应 50 到 70 页那一大段。建议边看边做实验 6。

第三本，《接口自动化测试项目实战》（清华大学出版社，2021）。接口是性价比最高的一层，这本书给的是"项目怎么做"而不是"API 怎么调"，适合你把第 27 页那张架构图落地。

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）。如果你偏前端栈，或者想理解"脚本跑在浏览器里"这条路线，看这本。

我的建议是**别四本一起看**。路径应该是：**先 Selenium 打通一条 UI 链路 → 再 RF 学会组织与报告 → 再接口那本补上主干 → 最后按需看 Cypress**。技术书的价值不在读完，在**解决你当下那个问题**。

另外别忘了官方文档——JUnit、Selenium、Appium、RF、Airtest 的官方文档永远比二手教程准。**遇到版本相关的疑问，一律以官方文档为准**。

板书：四本书按学习顺序竖排：① Selenium 入门（WebDriver・定位・等待）② RF 精解（关键字・库・

> 报告・集成）③ 接口实战（架构落地）④ Cypress（浏览器内路线）。底部路径："UI 链路 → RF 组织 → 接口主干 → 按需 Cypress"；右侧一句："版本问题以官方文档为准"。

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

- 答"因为先学框架更系统"——系统是好事，但容易"学完不会用"。

- 答"两条路都行，看人"——也对，但多数人的有效路径是"先窄后宽"。

- 收口：所以学习顺序应当匹配**反馈速度**：能快速看到结果的先学，抽象的后学。

易错点 / 考点：① 四本书的书名与大致定位（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 常考判断："工具书要通读"——不必，按需查阅效率更高。

过渡：这一章就到这里，最后几句话送给你们。

【9.75】感 谢 聆 听（1 分钟）

第 9 章讲完了。

我们从"为什么重视自动化"讲到"自动化到底该做在哪一层"，从静态分析讲到单元、接口、UI，从脚本技术讲到集成框架与报告。细节很多，但只让你带走三句话。

第一句：自动化是投资，不是采购。买工具不等于有自动化，真正的成本在设计与维护；算不清 ROI 的自动化，越做越亏。

第二句：分层决定成败。塔基是静态分析与单元，中间是接口，塔尖才是 UI。倒过来做，早晚推倒重来。

第三句：自动化的终极产物是信任。让你在每次改动之后敢发版、敢上线、敢对用户负责——这才是它真正的价值。

下一章我们会走进另一个维度：当软件要长期演进、要被人接手、要反复修改时，测试还能怎么帮上忙。带着这三句话去，你会发现后面的内容都在回答同一个问题——**怎么让质量持续得下去**。

谢谢大家。

> **板书：**三句话竖排：① 自动化是投资不是采购（成本在设计维护）② 分层决定成败（静态／单元 → 接口 → UI）③ 终极产物是信任（敢发版、敢负责）。底部："质量要成为一件可持续的事"。

提问：这一章里，哪一个概念你觉得自己还没吃透？课后用它去查一个真实项目。

预设回答：

- 答"ROI 怎么算"——很好的切入点：拿你手上的项目，估一次回归的工时，再估脚本维护工时。

- 答"框架怎么选"——用第 43 页那五问，对着你们团队的情况答一遍。

- 答"脚本为什么老红"——用第 30 页的三步（识别、操作、验证）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 三句话是本章的收束，论述题可用作结论段；② 复习时按"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 5 处）

序号	所在页	落点主题
1	9.5	这一页讲的是 RoI，其实讲的是务实。行业里最容易犯的错，是把"先进"当"正确"：
2	9.23	JUnitPlatform 的意义，是让"跑测试"成为有统一入口、硬性插座的流水线
3	9.43	选型这一页讲的是"因事制宜"。行业里最常见的浪费，是用最流行的工具去解最不匹配的
4	9.57	对象库、关键字库、命名规范，讲到底是在为"后来的人"负责。你写脚本时图省事，把定
5	9.72	第三问尤其值得诚实面对。做自动化最常见的失败，不是工具选错，而是不肯承认失败：脚