《软件测试课程设计》讲稿（第 6 章）

第 7 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 6 章　系统功能测试

第 7 周 · 2 学时（90 分钟） | 教材：朱少民《软件测试方法和技术（第 4 版）》

《软件测试课程设计》第 6 章 讲稿（第 7 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 6 章 系统功能测试 —— 从"单个函数对不对"走到"整个系统好不好用"
>
> 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 **71 页**（`lecture6/`，云端 http://111.231.0.226:2345/lecture6/index.html）
>
> 教案对应：第 7 次课「系统功能测试」
>
> 本讲重点：系统功能测试的七个维度（功能／逻辑／接口／界面／数据／操作／平台）；面向接口（HTTP／WebService）与面向 UI 两条主线；功能测试自动化的分层原则（单元 ROI 最高、少做 UI）；API 自动化架构与 Rest-Assured；UI 自动化三件套（Selenium/WebDriver、Cypress、Android 系）；回归测试的四种策略；精准测试的三块基础
>
> 本讲难点：为什么"分层自动化"要以接口层为主、UI 层为辅；Appium（跨进程）与 Instrumentation 系

> （同进程）两种机制的本质区别；回归测试"范围"与"策略"的取舍；精准测试如何靠"用例↔代码映射＋依赖分析"把回归范围收窄
>
> **本讲稿为可照读完整版**：每页含【口播（可直接朗读）】【提问＋预设回答】【易错点/考点】【过渡】；带 ★ 的页为时间紧张时可压缩页。

一、教学目标

维度	目标（课后可检验）
知识	① 能说出系统功能测试七个维度并各举一例；② 能画出 HTTP 请求/响应报文结构，说出 WebService 的 SOAP／WSDL／UDDI 各自作用；③ 能解释分层自动化的 ROI 排序及"尽量少做 UI 自动化"的理由；④ 能说出 API 自动化的四类 API 来源与四类回归触发方式；⑤ 能写出 Rest-Assured 的 given-when-then 结构；⑥ 能区分 Appium 与 Robotium／Espresso 的实现机制；⑦ 能列出回归测试四种策略及各自适用场景；⑧ 能说出精准测试的三块基础（覆盖率分析、用例与代码映射、依赖分析）
能力	① 能为一个业务系统写出"功能—子功能—功能点"三级测试清单；② 能用第 3 章的输入域／组合方法与第 4 章的 Charter 思路，设计探索式测试任务；③ 能把 10 条手工用例转成 Selenium＋WebDriver 的 Java 脚本并在 JUnit 下运行；④ 能针对一次代码改动，判断该用哪种回归策略、范围划到哪；⑤ 能读一份覆盖率报告，结合依赖分析判断"这次改动真正要重测哪些用例"
素质（思政）	① 由七个维度体会"系统思维"——不问单点对错，问整体是否成立；② 由接口契约体会"承诺必守"的职业信用；③ 由分层自动化体会"抓主要矛盾、按投入产出办事"的务实；④ 由回归缺陷体会"守护既有成果"的责任感；⑤ 由精准测试体会"用证据说话、不拍脑袋"的科学态度

本讲要建立的一个总印象：第 5 章把每个零件钉死了，但**零件都对，整体不一定对**。这一章解决两件新事：一是"**怎么从系统层面验证功能**"（接口与界面两条路），二是"**改动之后怎么既快又准地重测**"（回归与精准测试）。

二、时间分配总表

> ⚠ **容量提示（务必先读）**：本章课件 **71 页**，本讲稿是**素材全量版**（口播约 3.6 万字，≈150 分钟纯朗读），

> 课时只有 **90 分钟**。请按下表取用：
>
> 1. **重点页全文慢讲（2–3 分钟/页）**：
>
> 6.5、6.9、6.10、6.12、6.15、6.17、6.23、6.32、6.42、6.59、6.61、6.63、6.67。
>
> 2. **常规页快速带过（0.8–1.5 分钟/页）**：只讲口播第一段（结论）＋1 个提问。
>
> 3. **可课后阅读页（★）**：★6.6–6.8、★6.11、★6.16、★6.18–6.22、★6.24–6.41、★6.45–6.55、★6.58、★6.64–6.66、★6.70。
>
> 4. **必须当堂动手页**：6.12（列出"你的课程设计系统"的三级功能清单）、6.23（现场读一段 given-when-then）、6.61（给一次改动选回归策略）、6.68（三个开放问题讨论）。
>
> 5. **实验 3（6.69）**务必用 3 分钟交代：saucedemo.com → 需求分析＋20 条关键用例 → 选 10 条转 Selenium 脚本在 JUnit 跑。

90 分钟课堂推进表

时段	页码	内容	分钟	备注
第 1 节	6.1–6.4	开场、本章四部分、功能测试方法与应用	5	讲清"从零件到系统"
第 1 节	6.5–6.8	系统功能测试测什么、考虑什么、基本思路	8	**6.5 必讲**（七维度）
第 1 节	6.9–6.11	面向接口：HTTP 协议、WebService	8	**6.9、6.10 必讲**
第 1 节	6.12	面向 UI 的五条做法	4	**当堂列三级清单**
——	——	**课间休息 10 分钟**	——	——
第 2 节	6.13–6.22	6.2 功能测试自动化：分层原则、API 架构与流程、工具、Rest-Assured	14	**6.15、6.17、6.23 必讲**；工具清单可略
第 2	6.23–	UI 自动化：	16	**6.32、6.42 必讲**；其余按 ★

节	6.55	WebDriver、PageObject、Cypress、Android 系与两种机制		取舍
第 2 节	6.56–6.61	6.3 回归测试：回归缺陷、四种策略	10	**6.59、6.61 必讲**
第 2 节	6.62–6.66	6.4 精准测试：三块基础	6	**6.63 必讲**
第 2 节	6.67–6.71	小结、思考三问、实验 3、结束	5	6.68 三问开放讨论

> **压缩优先级**：先压缩 ★ 标记的工具截图页与 Android 各框架细节 → 再压缩 Cypress／MonkeyRunner／Robotium 的实现细节（保留 Appium vs Instrumentation 的对比）→ 最后才压缩 6.5、6.10、6.15、6.23、6.42、6.59、6.61、6.63。**这八页是考点密集页，不要动。**

三、逐页讲稿

> 使用说明：每页第一段是**可直接朗读的口播**；其后是提问与预设回答、易错点/考点、过渡语。
>
> 页码锚点与课件页一一对应（6.1–6.71），可逐页对照：http://111.231.0.226:2345/lecture6/index.html
>
> **两处口径提醒**：① 课件把 `Selendroid` 误写成 `Seledroid`，以正确拼写为准；② 6.14 页"iOS 和 Appium 内容可在第 9 章讨论／让学生参考教材 6.2.5、6.2.6 课外实践"是作者备课备注，已转成对学生的说明，不作为考点。

【6.1】软件测试方法和技术（1 分钟）

> 手机上最常用的那个 App，明天发一个新版本。你敢说它的主要功能都还对吗？这句话不好答。

>

> 第 5 章我们一直在代码内部干活。评审、单元测试、Mock、覆盖率、契约测试。被测的东西很小，对错能用一条断言钉死。写错一个判断，红灯一亮，改完再跑，心里有底。

>

> 现在镜头往外拉。被测的是一整个跑起来的系统：有界面、有接口、有权限、有第三方支付、有别人写的服务。对象一大，麻烦跟着变。很多"对不对"没有现成答案，它取决于用户怎么用。同一个按钮，学生点和你妈点，结果可能不一样。

>

> 还有一层变化容易被忽略。上一章出了问题，你打开 IDE 就能复现；这一章出了问题，得先把环境搭起来、数据造出来、账号配好。测一个系统功能，一半力气花在让它先跑起来。

>

> 所以本章换了个问题。装起来的整台机器，按用户真实的用法去用，它对不对、顺不顺、会不会崩。

>

> 四块内容排下去：方法与过程、自动化、回归测试、精准测试。前两块是主体。后两块回答另一件事——改完之后，测多少才够。

>

> 每块都有落点。七维度你得能背下来还能对着用；能说清为什么先测接口后测界面；能算出一次改动要跑哪些用例。

>

> 最后留一个问题。凭什么是你来说"主要功能都还对"？带着它听。

> **提问：**一个功能"点下去有反应、页面也跳了"，算不算功能正确？

> **预设回答：**

- 答"算，能用就行"——最典型的误解。"有反应"只说明链路通了，不说明结果对。金额算错、权限放开、数据落错表，页面照样会跳。

- 答"不算，还要看结果对不对"——对了一半，还得补一句：在什么输入、什么角色、什么环境下结果对不对。

- 答"要跟需求对一遍才算"——最接近。判据是需求与用户预期，不是"程序没报错"。

- 收口：这三种回答的差别，正好是本章要立的第一个观念——功能测试验的是行为，不是界面。

> **易错点 / 考点：**① 功能测试不等于界面点点点，接口层、数据层、平台层都在范围内；② 功能测试不等于手工测试，它可以自动化（6.2 整节讲这件事）；③ 判断题陷阱："功能测试只关心功能，不关心性能和安

全"——错，它关心的是正常与异常条件下功能是否仍然正确，异常条件本身就含并发、越权、脏数据。

> **过渡：**先看四块怎么排，你心里有个时间预算。

【6.2】6.1 功能测试方法与过程（1 分钟）

> 四个板块摆在一起，第一眼看不出先后。它们是四种问法。

>

> 6.1 功能测试方法与过程，回答三个问题：系统功能测试究竟测什么、要考虑哪些问题、基本思路是什么。然后分两条路走，面向接口和面向 UI。

>

> 6.2 功能测试自动化，把用例变成能反复跑的脚本。接口、Web UI、Cypress、Android UI 四类。里面有个判断比工具重要：分层自动化，它决定力气花在哪一层。

>

> 6.3 回归测试，改了一行代码，凭什么说别的地方没坏。

>

> 6.4 精准测试，回归的升级版。用代码与用例的映射关系，算出这次改动要跑哪些用例。

>

> 四块连成一条线：6.1 定对象，6.2 定手段，6.3 定范围，6.4 把范围缩到要跑的那几条。

>

> 实验课先做哪一块，也说清。先做接口那一层，别一上手就写 Selenium。接口脚本短、跑得快，红了就知道错在哪；界面脚本一红，你连是脚本坏了还是版本变了都分不清。

>

> 走偏的人通常是这么走的：一上来扑工具，用例设计随手糊，最后攒下一堆天天红的脚本，没人敢删也没人敢信。反过来走就顺：先会设计，再学复制。

>

> 教材是按节写的。课上按这条线走，你的笔记也照这条线记。

> **提问：**只给你一周时间学这一章，先学 6.1 还是先学 6.2 的自动化？

> **预设回答：**

- 答"先学自动化，工具最实用"——工具好学，但用例设计不对，自动化只是把错的用例跑得更快。这是行业里最常见的翻车方式。

- 答"先学 6.1"——稳。6.2 的脚本内容，就是把 6.1 的用例翻译成代码。

- 追问：那 6.2 的价值在哪？在于让同一批用例被重复执行一万次，手工做不到。

- 收口：先学会设计，再学复制，顺序别反。

> **易错点 / 考点：**① 四个板块名称要能默写，简答题常问"本章主要内容"；② 别把 6.3 和 6.4 当成两件事——精准测试是回归测试的一种实施策略，它解决"回归范围怎么定"；③ 问"功能测试自动化包括哪几类"，答四类，不要只答 Selenium。

> **过渡：**从 6.1 开始。先看这一节要解决的核心问题：功能测试到底在测什么。

【6.3】6.1 功能测试方法与过程（1 分钟）

> 内容最杂的一节，先把路线画出来再进，否则容易走丢。

>

> 分两段。前一段是"想清楚"：功能测试的要求与基本思路、测试需求怎么分析，还有探索式测试、SBTM 这类边测边设计的方法怎么用。

>

> 后一段是"动手做"。按被测对象的形态分两条技术路线。面向接口：HTTP 协议、WebService。面向 UI：分层展开、输入设计、角色路径、界面测试。

>

> 为什么先接口后 UI？逻辑绝大部分不在界面上，在服务端。从接口打，直接打到本体。从界面点，中间隔着渲染、脚本、网络，出了错还得先判断是皮还是骨头。

>

> 还有一层现实原因。接口能提前测，前端没写完就能动手；界面要等系统能跑起来才测得了。同样的缺陷，早一周发现和晚一周发现，修起来的价钱差好几倍。

>

> 行业里有句话：能测接口就不测界面。6.2 讲分层时会再遇到一次，那时候它变成一句关于成本的判断，不再只是习惯。

>

> 顺序也是考点。有人问为什么先讲过程、后讲自动化，答案就在这一页：过程决定用例长什么样，自动化只是把同一批用例搬到机器上跑。过程糊了，自动化只会更快地跑错。

>

> 两条路定下来，先看方法清单里放了什么。

提问：同一个"下单"功能，从接口测和从界面测，测到的东西一样吗？

> **预设回答：**

- 答"一样，都是下单"——不一样。界面测的是用户看到的流程，接口测的是服务端的契约。界面可能因为一个按钮错位就红了，业务算错反而未必被发现。

- 答"接口测得更深"——方向对。接口能构造界面造不出来的输入：越权 ID、超大金额、缺字段。

- 追问：那界面测试能不能省？不能省。它验的是用户能不能用，接口替代不了。

- 收口：两者互补，不是替代；但优先级上，接口在前。

易错点 / 考点：① "面向接口"和"面向 UI"是测试切入的层次，不是两种独立测试类型；② 常问"为什么优先做接口测试"，三条标准答案：更接近业务逻辑、构造输入更自由、执行更稳定更快；③ 别把这一页当纯目录跳过，它给出了本节的技术路线顺序。

过渡：进 6.1 的第一页，看方法清单里都放了什么。

【6.4】功能测试方法与应用（2 分钟）

六个词摊在课件上，看着平级，实际有先后。按先想、再拆、后做的顺序讲一遍。

>

第一块，功能测试要求和基本思路。接下来三页讲这些：要求是什么、要考虑哪些问题、基本思路有哪几条。它是后面所有方法的前提。思路错了，方法再熟也是白测。

>

第二块，测试需求分析方法。这个词要多看一眼：测试需求分析，不是需求分析。需求分析问"用户要什么"；测试需求分析问"我拿什么去证明用户要的东西被实现了"。前者产出功能列表，后者产出测试项、测试条件、优先级、风险。这一步空着，后面的用例就是无源之水。

>

第三块，探索式测试及其测试实践。它不预先写完用例，边学习、边设计、边执行，把测试当成一次有目的的探索。它补的是设计不出来的盲区。很多缺陷不在任何人的用例里，在没人想到要测的地方。

>

第四块，SBTM，基于会话的测试管理。它是探索式测试的管理外壳：把探索切成一段段有主题、有时限的会话，每段结束留下记录。切了、记了，探索才可度量、可复盘、可交付。

>

第五块第六块，回归测试及其策略、精准测试。这两块在 6.3、6.4 单独展开。它们解决同一个问题：改动之后，测多少才够。

>

六块连起来四个动作：想清楚、拆出来、补盲区、管改动。

提问："测试需求分析"和"需求分析"，就差两个字，差在哪？

预设回答：

- 答"差不多，都是看需求文档"——差得远。需求分析问要做什么，测试需求分析问怎么证明它做到了。

- 答"测试需求分析要出测试用例"——不准确，它出的是测试项和测试条件，用例是下一步的事。

- 答"要识别风险和优先级"——很关键，这是它比需求分析多出来的部分：哪些先测、哪些测不动、哪些测了也没用。

- 收口：需求分析定义"标准是什么"，测试需求分析定义"怎么验证它是什么"。

> **易错点 / 考点：** ① 六个词的名称要能默写；② 探索式测试常考判断："探索式测试就是随便乱点"——错，它有章程、有记录、有复盘，是结构化活动；③ SBTM 不是工具，是管理方法，别答成某个软件；④ 别把"测试需求分析"和"测试设计"混成一件事，前者定范围，后者定数据。

> **过渡：** 先回答最基础的那个问题——系统功能测试，究竟测什么？

【6.5】系统功能测试究竟测什么？（3 分钟）

> 点下去有反应，页面也跳走了，这就算功能对了吗？七个词里一个都对不上。课件给的是功能、逻辑、接口、界面、数据、操作、平台，一个一个说。

>

> 功能，最直白：在不在、做没做全。下单要能选商品、算价、用券、提交，少一个都不行。它测的是完备性。

>

> 逻辑，功能在，判断对不对。满减算几折、会员价和券能不能叠加、退货后积分扣不扣回。这一层测业务规则，也最容易错，因为它藏在代码里，界面上看不出来。

>

> 接口，模块之间、和外部系统之间的约定。支付回调、短信网关、第三方登录、微服务调用。接口不通，功能就是看着有、实际废。

>

> 界面，布局、提示语、必填标识、错误提示说不说得清、按钮在不能点的时候灰不灰。

>

> 数据，最容易被忽略、后果最重的一维。字段和精度对不对、显示有没有脱敏、空值、超长、特殊字符、重复提交进来会怎样。

>

> 操作，键盘能不能走完、连点会不会重复下单、中途返回会不会丢数据、误操作能不能撤。它测的是人的不规矩。

>

> 平台，同一套代码在不同浏览器、分辨率、系统、网络、机型上的表现。

>

> 七个都过一遍才叫"系统"功能测试。只测第一个，那叫功能点核对。

>

> 这七个词也在说一件事：系统是一个整体。功能对、数据错，用户的钱就少了；逻辑对、平台不对，一半用户打不开。别养成"这不是我这一块"的习惯。你写的接口，别人要拿它拼系统；你定的字段，后面有人靠它算钱。

提问：一个"提交订单"的功能，用七个维度各问一句，能问出几个可能的问题？

> **预设回答：**

- 答"优惠券叠加算错了"——逻辑维度。这类缺陷界面上完全看不出来，只能靠算。

- 答"连点两下提交了两单"——操作维度，就是幂等与防重没做，属高频线上事故。

- 答"金额存进库里变成 99.99999"——数据维度，浮点精度，很常见。

- 答"我只想到功能和界面"——很真实。想不到的维度，就是漏测的地方。

- 收口：把七个词写进测试计划模板，每次评审逐条对一遍，比凭经验想可靠。

易错点 / 考点：① 七个维度的中英文要能对上（Function／Logic／API／UI／Data／Operation／Platform），选择题常给英文；② 常考区分：界面不等于功能，数据不等于数据库测试（数据维度含显示、精度、脱敏、脏数据）；③ 案例题问"应从哪些方面测试"，按七维展开，只答功能、性能、安全会失分。

【思政落点 1】：把功能、逻辑、接口、界面、数据、操作、平台七个维度摆在一起，本身就是一种训练：**不要只盯着自己写的那一块**。做课设最常见的失误，是三个人各测各的，最后没人对"整个系统能不能用"负责。学会从七个角度看同一件事，是将来的测试工程师和普通程序员的第一个分水岭——前者关心系统，后者关心函数。

> **过渡：**七维度说的是测哪些面，接下来问：每个面上要考虑什么问题？

【6.6】系统功能测试应该考虑哪些问题？（3 分钟）

> 课件这一页只有一句设问，后面没有答案。答案要你自己填。

>

> 我把成熟测试人员动手前会问自己的问题归成七类。拿到任何一个需求，照这七问过一遍，不会空手。

>

> 第一问，测到哪一层。只验界面上的结果，还是要验接口返回、要验数据落库？层次定得低，成本低、覆盖浅；定得高，覆盖全、投入大。这一问定的是投入结构。

>

> 第二问，谁会用它、怎么用。同一个功能，普通用户、管理员、客服、外部合作方，看到的和能做的完全不同。角色没想到，权限缺陷一定漏。

>

> 第三问，输入从哪来、边界在哪。用户手输、文件导入、接口传入、定时任务生成，来源不同，脏数据的形态也不同。边界怎么定，就是第 3 章等价类和边界值那套东西。

>

> 第四问，状态怎么变。功能不是孤立动作，它有前置状态：未支付才能取消、已发货才能退货、审核中不能编辑。状态图没画出来，用例一定漏。

>

> 第五问，跟谁交互。内部模块、第三方服务、消息队列、缓存。凡是跨出去的地方都是故障高发区，也都要多问一句：对方挂了、超时了，我们怎么办。

>

> 第六问，改动会波及谁。这一问直接通到 6.3。改之前先问：这段代码被谁调用、这张表被谁读。

>

> 第七问，怎么算测完。没有准出标准，"测完了"就是一句感觉。关键路径全过、缺陷收敛、覆盖率达标、遗留缺陷风险有人认过，这才算。

>

> 七问不必全答满分，但都要问过。问过而决定不测，是判断；没问就漏掉，是失职。

提问：一个"优惠券领取"功能，按第四问（状态怎么变），你会想到哪些状态？

> **预设回答：**

- 答"已领取、已使用、已过期"——最基本的三个，还不够。

- 答"还有未开始、已抢光、已作废"——补充得对，说明你想到了活动层面的状态，不只是券本身。

- 答"状态之间怎么跳我没想过"——很常见。补一句：状态测试的关键不是枚举状态，是枚举非法跳转。已过期的券能不能用、已使用的券能不能再领，这些才是缺陷窝。

- 收口：第四问的正确用法是先画状态图，再测跳转，不是拿状态当标签。

易错点 / 考点：① 这一页课件无文字，考法多在案例题里"请说明你将从哪些方面考虑测试"，按七问展开就能得分；② 最容易漏的两问是状态和外部依赖故障；③ "准出标准要提前定"这个判断，所有测试计划类题目都能引用。

> **过渡：**问题想清楚了，接下来是先用什么顺序、什么骨架动手。

【6.7】系统功能测试的基本思路（3 分钟）

> 拿到一个陌生系统，第一小时你干什么？课件这一节就是回答它。前三条先给结论：从需求出发、按层次分解、沿主路径推进。

>

> 第一条，从需求出发，把"要测什么"拆成树。需求给的是业务语言：用户可以下单并选择配送方式。测试要把它翻成测试语言：功能、子功能、功能点。下单是功能，选商品、算价、选配送、提交是子功能，选配送下面又有普通快递、次日达、自提三个功能点。树的叶子，就是一条条能执行的测试项。树画不出来，说明需求你还没读懂。

>

> 第二条，按层次分解，别一上来就点界面。系统的层次是数据层、服务层、展示层。测的顺序也这样：先确认接口契约对、数据落得对，再去验界面。反过来做，时间全花在"界面点不通"上，而点不通的原因可能是接口回了 500。这里有个坑：从最上层往下排查，效率最低。

>

> 第三条，沿主路径推进，先主干后枝节。用户用产品有主路径：打开、登录、选商品、下单、支付、看订单。先用最短时间把主干跑通，再回头补分支和异常。为什么？主路径不通，别的测试都没有意义。主干跑通还能最快暴露环境、数据、依赖的问题。

>

> 三条合起来是个能用的顺序：画树、分层、走主干。

> **提问：**为什么主张先测接口、后测界面，而不是反过来？

> **预设回答：**

- 答"接口比较重要"——重要不是理由，定位成本才是：界面报错时你判断不出是展示层、网络层还是服务层的问题。

- 答"接口不用等前端"——对，这也是一条实打实的理由，测试不必等开发全做完。

- 答"界面太脆，改一下就红了"——很真实。先用接口把逻辑钉死，UI 层只验展示对不对。

- 收口：三条理由合成一句——从稳定的、接近逻辑的那一层开始测。

> **易错点 / 考点：**① "功能—子功能—功能点"三层分解是简答高频点，要能举例；② 常考判断："测试从界面开始才对，因为用户就是这么用的"——错，那是验收测试的视角；③ 别把主路径理解成只测正常流程，它是先通后全。

> **过渡：**还有三条思路，讲的是怎么让测试做得可管理、可复用。

【6.8】系统功能测试的基本思路（2 分钟）

> 前三条讲从哪下手，后三条讲怎么不靠个人英雄主义。

>

> 第四条，用输入域和组合方法把数据设计出来。等价类划分把无穷输入压成有限几组，边界值盯临界点，判定表和两两组合处理多个条件同时起作用。功能测试里难的从来不是点按钮，是点之前填什么。用例的质量，八成取决于数据怎么选。

>

> 第五条，用矩阵管住环境、角色、数据三类变量。平台上是浏览器、机型、分辨率；角色上是普通用户、VIP、管理员、未登录；数据上是新数据、历史数据、脏数据。它们是相乘的关系，不是相加。三个浏览器乘四类角色乘三类数据，三十六种，不是十种。全跑跑不完，就用组合测试挑代表性组合，再按风险补几条关键的。

>

> 第六条，可追溯、可闭环。每条用例能追到需求条目，叫正向追溯；每个缺陷能追到用例和代码，叫反向追溯。为什么盯着这个？没有追溯，你回答不了"这个需求测完了吗"，改动之后也算不出要跑哪些用例。6.4 精准测试的技术底子就在这儿。

>

> 再补一条实操：测试过程中随手记下"想到了但没测"的点。这些点往往就是风险清单，写测试报告时最值钱的就是它。

>

> 六条一起用，测试就从"某个人记得"变成"团队查得到"。

提问：为什么说"平台 × 角色 × 数据"是相乘而不是相加？这对测试计划意味着什么？

预设回答：

- 答"因为要两两组合"——对，三个浏览器乘四类角色乘三类数据就是三十六种，不是十种。

- 答"那就全测"——时间不允许，收益还递减。风险高的组合全测，其余用组合测试取样。

- 答"我只测主流浏览器"——可以，但要在报告里写明测了哪些、没测哪些、为什么，这才叫负责任的取舍。

- 收口：组合爆炸不是数学题，是取舍题，取舍要有依据、有记录。

易错点 / 考点：① 可追溯性是测试管理的核心概念，简答常问"什么是用例的可追溯性"；② 正向是需求到用例（覆盖了吗），反向是缺陷到用例和代码（哪条抓到的），别混；③ 组合测试只保证任意两因素组合被覆

盖，不保证三因素交互，关键业务仍要人工补。

> **过渡：**思路讲完，开始走第一条技术路线——面向接口。

【6.9】面向接口的功能测试（1 分钟）

从这里开始动手。

>

> 前面所有思路落到接口这一层，变成三件事：看懂协议、看懂契约、构造请求。

>

> 为什么接口排在 UI 前面？有个判断值钱：逻辑绝大部分不在界面上，在服务端，界面只是调用者。从界面点，你只能输界面允许你输的；从接口打，你能输协议允许你发的。越权、缺字段、超大值、重复提交，界面测试抓不到，接口这边一试就出来。

>

> 这一节讲两个东西。HTTP，大多数接口的运输层，报文什么结构、每段能放什么，得门儿清。
> WebService，企业系统之间打交道的老牌方式，三个关键词 SOAP、WSDL、UDDI。

>

> 学完这两块，你要能做到一件事：看见一个陌生接口，知道从哪下手去戳它，而不是干等前端把页面做出来。

>

> 还有个附加好处。接口测试逼你把契约读一遍。读契约的人，比只读页面的人更早知道系统该怎么工作，这份理解在定位缺陷时最省时间。

>

> 这一节两页半，HTTP 讲怎么发，WebService 讲怎么约，落到手上是同一件事：拿契约对照实现，一处一处对。

>

> 留个问题给你。页面只报"系统繁忙"四个字，你怎么判断是请求没发出去、被服务端拒了、还是响应解析错

> 了?

> **提问：**页面报"系统繁忙"，你怎么定位是请求没发出、被拒、还是返回解析失败?

> **预设回答：**

- 答"看后台日志"——可以，但那是事后。测试时你手上得先有抓包。

- 答"抓包看请求和响应"——对，标准动作。抓包同时回答三个问题：请求发出去了吗、状态码多少、响应体长什么样。

- 追问：没有抓包工具呢？用 curl 或 Postman 把同一个请求重放一遍，能重放才能定位。

- 收口：这就是为什么得先懂报文结构。不懂结构，抓包摆在你面前也读不出来。

> **易错点 / 考点：**① "面向接口"是切入层次，不等于"接口测试"这个独立类型；② 常问"接口测试相比 UI 的优势"，三条：更早介入、输入更自由、执行更稳定；③ 别忽略接口也要验数据落库，返回 200 不代表数据对。

> **过渡：**先看 HTTP，所有接口测试都绕不开它。

【6.10】HTTP 协议（3 分钟）

> 页面只给你四个字，系统繁忙。想弄清它，得能读懂那一次往返。

>

> 接口测试里九成的"请求为什么不对"，答案就在报文结构里。课件给得很干脆：请求是请求行、头、空行、请求体；响应是状态行、头、空行、响应体。

>

> 先说请求。请求行三段：方法、资源路径、协议版本。方法选错很常见，用 GET 改数据、不让缓存的接口被缓存。头里装元信息：内容类型、长度、Cookie、鉴权令牌、来源、期望格式。头里最常出事的是内容类型。发 JSON 却声明成表单，服务端解析出来就是空对象，你还以为是后端没接。空行是分隔符，告诉服务端"头到此为止"。请求体才是业务数据，GET 通常没有体。

>

> 再说响应。状态行第一段是状态码，最直接的断言点：2xx 成功、3xx 重定向、4xx 客户端错、5xx 服务端错。

> 注意一句：状态码是协议层的结论，不是业务层的结论。很多系统返回 200，业务码写在响应体里，你只断状态码，业务错误全漏。响应头给类型、长度、缓存、Cookie 设置；响应体才是业务结果。

>

> 为什么要把结构记到能画出来？因为你的断言就分布在这四段上。状态行断状态码，头断内容类型和安全头，比如 HttpOnly、X-Frame-Options，体断业务字段。结构不熟，断言只能写成"不报错"。

>

> 课件还给了 W3C 的规范地址，w3.org/Protocols。要查标准语义就看原文，比二手博客靠谱。

> **提问：**一个接口返回 HTTP 200，业务上并没有成功。这种情况你怎么测出来？

> **预设回答：**

- 答"那不可能，200 就是成功"——很常见的误解。大量系统用 200 承载一切，业务码放在响应体里。

- 答"要看响应体里的 code 字段"——对。这就是断言不能只写在状态行上的原因。

- 追问：要不要要求服务端改回用状态码表达业务错误？这是设计取舍。REST 风格主张用状态码，统一响应体风格主张用业务码。测试人员的任务，是按被测定下的契约去断言。

- 收口：先读契约，再写断言。

> **易错点 / 考点：**① 请求与响应的四段结构、空行是分隔符，属填空高频；② 状态码分类要能背：2xx 成功、3xx 重定向、4xx 客户端、5xx 服务端；③ 极易错：把 4xx 一律当成系统 bug，401、403、404 往往是正常的权限与存在性语义；④ 断言分两层，协议层断状态码和头，业务层断响应体字段，两层都要有。

> **过渡：**HTTP 是自己拼报文的风格。企业系统之间还有一套更重的方式。

【6.11】Webservice 接口（3 分钟）

> 银行、政务、ERP 里，接口还是另一副样子。今天不算时髦，但活得稳。

>

> 先看它是什么。课件说：WebService 是不依赖语言、不依赖平台的 SOA 架构，能实现基于 HTTP 协议、不同语言之间通过 XML 描述的相互调用，以及网络应用之间的交互。三个要点：跨语言、跨平台、靠 XML 说

> 话。语言不通、系统不通，只能约定一种大家都能读的文本。

>

> SOAP，全称 Simple Object Access Protocol，XML Web Service 的通信协议，课件括号里写着"调用方法的规范"。这几个字关键：它不只传数据，还规定怎么表达"调哪个方法、参数是什么、返回什么、出错怎么报"。底层也能换，HTTP(S)、SMTP 都行。

>

> WSDL，Web Services Description Language，一份 XML 文档，说明一组 SOAP 消息以及如何交换这些消息。它就是契约：服务地址、有哪些操作、每个操作的参数、返回结构与类型。有了它，工具能直接生成调用代码。

>

> UDDI，Universal Description, Discovery, and Integration，根据描述文档来引导系统查找相应服务的机制。相当于黄页：用 SOAP 消息机制，也就是标准 XML 与 HTTP，发布、编辑、浏览、查找注册信息；用 XML 封装数据，发到注册中心或从注册中心取回。

>

> 测试落点有三。契约校验：按 WSDL 校验请求结构，少必填、类型错要不要拒。错误处理：SOAP Fault 有没有规范返回，而不是抛 500 加一段 HTML。版本兼容：WSDL 加了字段，老客户端还能不能调。

>

> 现实里的事故，多半出在一处：契约改了没人通知，类型从整型变字符串、必填变选填。改之前要先通知。

提问：拿到一个只有服务地址、没有 WSDL 的 WebService，你怎么测?

预设回答：

- 答"没法测，结构都不知道"——太悲观。可以先按 SOAP 的固定信封结构，Envelope 和 Body，发一个最小请求，看错误信息反推结构。

- 答"找开发要 WSDL"——最正确的一步。WSDL 就是契约，没有契约只能靠猜，猜出来的断言不可靠。

- 追问：开发说"WSDL 是自动生成的，不准"呢？那本身就是缺陷——契约与实现不一致，要让开发把 WSDL 固化成对外承诺。

- 收口：接口测试的第一个动作往往不是发请求，是要契约。

> **易错点 / 考点：** ① 三个缩写的全称与作用必须对上：SOAP 是协议、WSDL 是描述文档、UDDI 是注册查找机制，选择题最爱在这里换位；② 常见错误：以为 WebService 只能跑 HTTP，课件明确写了支持 HTTP(S)、SMTP 等底层；③ 别把 WSDL 说成"接口文档"就完事，它是机器可读的契约，工具能据此生成代码；④ 案例题问"企业系统对接怎么测"，落到契约校验、异常返回、版本兼容三点。

> **【思政落点 2】**：WebService 的 SOAP、WSDL、UDDI 三件套，本质是把"两个系统怎么说话"写成公开的、可查的约定。这跟我们平时写接口文档的道理一样：**约定写清楚，别人才敢接**。以后你在团队里定接口，把字段、格式、异常都写明白，这不是形式主义，是让协作有据可依。守约定、留文档，是工程信用的一部分。

> **过渡：**接口这条路走完了。换一条，从用户看到的那一层切进去。

【6.12】面向 UI 的功能测试（3 分钟）

> 轮到用鼠标的那一层了。五条，条条是实操要求。

>

> 第一条，重点测功能逻辑，按功能、子功能、功能点等层次展开。这里的关键是"重点测功能逻辑"。面向 UI 不等于看界面好不好看，是通过界面验证背后的业务逻辑。分层就是那棵树：功能是订单，子功能是下单、支付、退款，功能点是选配送、用券。分了层才排得出优先级和人力，评审时才说得清这块测到什么程度。

>

> 第二条，基于输入域和组合测试这类方法（第 3 章那套）设计输入，驱动测试，并观察输出。这是一条动作链：设计输入、驱动执行、观察输出。哪一环松了都出事。特别说"观察输出"：别只看有没有报错提示，要看列表数据变没变、金额对不对、状态流转对不对。

>

> 第三条，扮演用户角色，从应用场景遍历用户使用产品的主要操作路径。这是 UI 独有的价值。接口能精确控制一次调用，但用户不这么用软件。他会搜、会筛、会比价、会犹豫、会返回、再下单。角色不同，主路径就不同。遍历的意思是一条路走到黑，不是每步都重置。

>

> 第四条，针对不同设置测试。设置项是组合缺陷的窝：语言、时区、货币、通知开关、隐私选项、默认地址。时区错了，订单日期就差一天，这是经典事故。

>

> 第五条，用户界面测试。布局、文案、必填标识、错误提示有没有说清怎么改、加载与空状态、极端数据比如超长文本和无头像会不会撑破页面。

>

> 五条合起来：分层定范围，输入定深度，角色定真实，设置定差异，界面定表达。

> **提问：**第三条说扮演用户角色，为什么不能只用一个管理员账号把功能点一遍？

> **预设回答：**

- 答"管理员权限最大，能点通所有功能"——恰恰是问题所在。权限最大的账号看不到权限缺陷，很多缺陷只在"没权限的人试图操作"时暴露。

- 答"不同角色看到的菜单不一样"——对，这是角色可见性。应看到的没看到是缺陷，越权看到内容更是缺陷。

- 答"场景不一样，路径也不一样"——很深。同一功能，新用户和老用户的前置数据不同，走出来的路径和结果都不同。

- 收口：角色不是换个账号登录，是换一套前置条件和预期。

> **易错点 / 考点：**① 五条要能默写，简答常问"面向 UI 的功能测试包括哪些内容"；② 最容易错的是把第一条读成"测界面"，课件写的是重点测功能逻辑；③ "观察输出"常被简化成看有没有报错，答题要写出数据、状态、提示三类；④ 界面测试查表达，功能测试查行为，边界别混。

> **过渡：**6.1 的方法讲完了。接下来是重头戏：怎么把这些测试自动化。

【6.13】6.2 功能测试自动化（1 分钟）

> 自动化这件事，先别急着装工具。立场不说清，你会把它学成一门工具课。

>

> 最容易走偏的路是这样：一头扎进 Selenium 或者 Cypress，写了几十个脚本，觉得学完了。三个月后脚本全红，没人会修，于是整批删掉。问题不在工具，在开始之前没想清三件事。

>

> 所以这一节的顺序是：先看四类路线，接口、Web UI、Cypress、Android UI；再看分层自动化这个判断，力气往哪一层花；然后拆接口自动化的链路，架构、流程、工具、Rest-Assured；最后是 UI 那一路，Selenium、WebDriver、页面对象，再到 Cypress 和安卓。

>

> 顺便交个底：这一节考试分值不低。四类名称、分层判断、六个接入步骤，都是填空和简答的常客。工具名要写准，Cypress 属于 UI 那一类，别塞进接口。

>

> 落点不是让你成为工具高手。是让你能回答三个问题：哪些测试值得自动化、在哪个层次自动化、怎么才不烂掉。第三个最现实。大量项目不是写不出来，是三个月后没人维护。

>

> 留个问题给你：一个人，每周十小时做自动化，先做接口还是先做界面？想清楚再往下翻。

提问：一个人、每周十小时做自动化，先做接口还是先做界面？

预设回答：

- 答"先做界面，用户看到的就是界面"——直觉，代价最大：脚本脆、维护贵、单条执行慢。

- 答"先做接口"——多数情况下的正确答案，理由在分层那一页给全。

- 追问：什么情况下先做界面？当核心风险就在界面交互本身时，比如复杂表单校验、可视化编辑器。

- 收口：先判断风险在哪一层，再决定自动化落在哪一层，顺序别反。

易错点 / 考点：① 功能测试自动化的四类要能背；② 自动化的目标是替代重复，不是替代人，探索性、易用性、一次性验证都不适合自动化；③ 常问"什么样的测试适合自动化"，要点：重复执行、结果可判定、输入可构造、需求相对稳定。

过渡：先看总类目，四类，把边界划清。

【6.14】功能测试自动化（3 分钟）

> 四条路摆出来，多数团队常动的只有两条。先把四条看清，再谈选哪条。

>

> 面向接口的自动化测试。被测对象是 HTTP 接口或 WebService，不经过界面。优势是稳、快、能构造极端输入，界面上线前就能写。多数系统的自动化主力在这一层。

>

> Web 客户端的 UI 自动化测试。代表是 Selenium 系列。它模拟真人在浏览器里操作：找元素、点击、输入、断言。价值在验证用户真的能用。接口全绿但按钮点不动，只有它抓得到。代价是脆、慢、维护贵。

>

> 基于 Cypress 的 UI 自动化测试。也测 Web 界面，技术路线完全不同。Selenium 通过驱动在外部指挥浏览器；Cypress 把测试代码跑在浏览器内部，和被测应用同一个运行环境。这个差别既带来好处，也带来限制。

>

> Android 应用的 UI 自动化测试。
> Robotium、UIAutomator、MonkeyRunner、Appium、Espresso、Selendroid 这些名字。它们的根本区别在于测试进程和被测应用是不是同一个进程。这一点决定你能不能直接用控件 ID 操作元素，后面专门讲。

>

> iOS 和 Appium 的深入内容，这一节只到"知道它是什么、能做什么"。iOS 的自动化在第 9 章还会专门讨论；Appium 更细的用法，请按教材 6.2.5 和 6.2.6 两小节在课外动手做一遍。这类工具光看不行，装一次、跑通一个用例，胜过看十页文档。

>

> 选择顺序很清楚：能用接口就不用 UI；Web 端先算维护成本；移动端先想清进程模型，再选框架。

> **提问：**这四类里，哪一类适合每天跑，哪一类只适合发版前跑？为什么？

> **预设回答：**

- 答"接口每天跑，UI 发版前跑"——基本正确，理由要说清：接口快而稳，适合高频；UI 慢而脆，高频跑会淹没在误报里。

- 答"UI 也能每天跑，只要机器多"——可以，但维护成本跑不掉。每天跑意味着每天都要处理失败的脚本，那不是加机器能解决的。

- 追问：UI 自动化的定位是什么？守关键路径。登录、下单、支付这几条最要紧的路，用最少的用例覆盖，其余交给接口层。

- 收口：分层不是技术偏好，是成本结构决定的。

> **易错点 / 考点：** ① 四类名称要能默写，选择题常把 Cypress 塞进"接口测试"当干扰项，它是 UI 自动化；② iOS 与 Appium 在本节只作了解；③ 判断题"UI 自动化越多越好"——错，UI 是成本最高、收益最低的一层；④ Appium 属于跨进程路线，这条是后面 6.42 对比题的基础。

> **过渡：** 为什么说 UI 要少？下面这张分层图给出理由。

【6.15】分层自动化测试（3 分钟）

> 金字塔这个形状，考试爱考，工程上也真在用。四句话先念一遍：单元测试更容易实现自动化，RoI 也更高，收益显著；API 自动化测试也基本能做到 100%；适当加强手工端到端测试，在业务层；概括起来，以底层测试、接口测试、功能逻辑测试为主，尽量避免 UI 测试。

>

> 第一句讲单元的投入产出比最高。它跑得最快，毫秒级，每次提交都能全量跑；定位最准，红了就是那个函数的问题；最稳，不依赖界面、网络、数据。同样的时间，压在这一层捞出来的缺陷最多。

>

> 第二句讲 API 自动化能接近 100%。说的是覆盖率。接口可以被穷举：参数组合、状态码分支、权限矩阵，都能用代码构造出来逐个打。界面上的用户行为穷举不了，所以 UI 自动化的覆盖率天然上不去。

>

> 第三句最容易被跳过：适当加强手工端到端。关键是"手工"两个字。自动化程度高了为什么还要手动？端到端验的是业务价值成不成立。从搜索到收货这一整条链，作为一个业务故事通不通，要人来判断。而且端到端用例少、变化慢，手工跑的成本可以接受。

>

> 第四句是结论：以底层、接口、功能逻辑为主，尽量避免 UI。课件图里还有四组标注说明理由。底层：缺陷更易定位、效率更高。上层：反映真实需求、更加接近业务。越往下越好测，越往上越真实。

>

> 分层的意义就一句：把大量验证压在好测的层，把少量验证留在真实的层。每往上一层，执行时间、维护成本、误报率都在涨，缺陷定位精度在降。会算这笔成本的人，比会炫工具的人难得。

> **提问：**既然单元测试 RoI 最高，为什么很多公司还是把人力主要投在 UI 自动化上?

> **预设回答：**

- 答"因为 UI 看得见，好交差"——扎心但真实。UI 自动化演示直观，单元测试写得再好，汇报时什么都没看见。

- 答"历史代码没单测基础，补不起来"——很现实的约束。遗留系统依赖太多，写一个单测要 Mock 半天，成本反而高。

- 追问：这种团队怎么办？从新增代码开始要求单测，老代码用接口自动化兜住主要逻辑，UI 只守关键路径。分层不等于平均用力。

- 收口：分层是成本最优，不是道德标准。说服别人靠算，不靠喊。

> **易错点 / 考点：**① 四句话的顺序和关键词要能背，特别是 RoI 最高、接近 100%、适当加强手工端到端、尽量避免 UI；② 极易错：把"API 100%"读成"API 能发现 100% 的缺陷"，课件说的是自动化覆盖率；③ 案例题问自动化策略，答题必须体现分层：底层多、接口全、UI 少、端到端手工保留；④ 尽量避免 UI 不等于不做 UI，关键路径仍要守。

> **【思政落点 3】**：分层自动化的取舍，说到底是**会算成本**：知道哪一层的投入产出比最高，就把资源压在那里。这和做任何工程决策一样——不是"哪个技术最先进就用哪个"，而是"在我这个项目、这个团队、这个时间点，什么最划算"。敢于对不划算的方案说不，并且说清理由，这种判断力比会用一个工具值钱得多。

> **过渡：**底层和顶层都清楚了，把中间那层拆开看。

【6.16】API 自动化测试（2 分钟）

> 八百个接口，两周接入自动化。这题目不夸张，很多团队真面临这个数。

>

> 先看它应有的样子。从接口的契约出发，自动采集或组装出请求用例，自动执行，自动断言，自动出报告，并且在代码变更时被自动触发。四个"自动"缺一个，就退化成半自动。

>

> 为什么值得投？三个字。早：后端接口定下来就能写用例，不必等前端。稳：不依赖浏览器渲染，不怕界面改版。深：越权、缺字段、超大值、异常编码，界面里造不出来，这里随手构造。

>

> 三个理由里，深这条最容易被低估。界面允许你输的，是产品经理设计过的输入；协议允许你发的，是没有边界的输入。缺陷最爱待在没有边界的地方。

>

> 早这一条还有个连带效果：接口先测，等于逼后端先把契约定下来。契约定得早，前端才好动工。

>

> 难点也不在写脚本。两个地方卡人：接口从哪来，断言靠什么维护。上千个接口不能手录；字段一改，断言就得跟着改，改不过来就烂掉。

>

> 所以这一节要讲架构、流程、工具、示例、DSL。架构回答"平台要有什么"，流程回答"一个接口怎么接进来"，工具和示例回答"具体怎么写"，DSL 回答"怎么写才让人看得懂"。

>

> 回到那八百个接口。两周接入，靠的不是加班手写，是流水线。

提问：800 个接口，两周接入自动化，你会怎么做？

预设回答：

- 答"一个个手写脚本"——两周写不完，写完还要维护。

- 答"先抓包，把真实请求存下来，批量生成"——这才是正解，也正是架构里 HAR 导入和批量生成存在的原因。

- 追问：批量生成的用例，断言从哪来？从契约来（字段、类型、状态码），从历史响应来（基线比对），关键的再人工补业务断言。

- 收口：规模化的接口自动化，靠批量化流水线，不靠人力堆。

> **易错点 / 考点：** ① "接口自动化的优势"是简答高频，三条要说全；② 别把接口自动化和性能测试混谈，工具能复用，目的不同；③ "接口来源"是工程化关键，常被忽略，但正是架构题的得分点。

> **过渡：** 看架构，这张图信息量大，一块一块拆。

【6.17】API 的 TA 架构（4 分钟）

> 一个接口自动化平台长什么样？从左到右四段：接口从哪来、怎么定义、用例怎么组装、回归怎么触发。

>

> 第一段，API 的来源，四条路。文档化工具：用 Swagger、RAP 这类平台管文档，定义机器可读，直接拿来用。手工录入：没文档或者文档不可信时一条条填。程序扫描：扫代码里的注解或路由注册，自动提出清单。HAR 导入：把浏览器里真实发生的请求录下来导进去。

>

> 第二段，API 定义，三种载体。RAML 是 RESTful API 的描述性语言，用文本写清资源、方法、参数、响应。Swagger 是当下最流行的 API 定义规范。Markdown 最土但实用，很多团队的文档就是 Markdown 表格，也能解析成用例输入。定义的意义是：契约被文本化，人和机器都能读。

>

> 第三段，用例。课件给两条：场景设定，选择和组装 API；验证侧四条路径，基于场景的测试用例、手工组装、HAR 批量生成、代码迁移。关键概念是组装。单接口调通只算可用，业务价值在把多个接口串起来：登录拿令牌、查商品、下单、支付回调。所以主体不是单接口用例，是场景用例。

>

> 第四段，回归。触发回归四种：按需回归、定时回归、即时触发（提交就跑）、事件触发（比如构建成功）。之后是执行和结果展示。

>

> 两个口径要一字不差地记：RAML 是 RESTful API 描述性语言；HAR 是基于 JSON、储存 HTTP 请求与响应信息的文件格式。

> **提问：** 四种触发方式里，哪一种最能体现持续测试的思想？为什么？

> **预设回答：**

- 答"定时回归，每晚跑一遍"——有价值，但那是批处理思维：问题第二天早上才知道。

- 答"即时触发，提交代码就跑"——最接近持续测试：反馈越快，修复越便宜。

- 答"事件触发，构建成功就跑"——也很典型，它把测试挂在了流水线节点上，是 CI 的标准做法。

- 追问：那按需回归还要不要？要，用于发版前的人工决策和临时验证。

- 收口：四种不是选一个，是按场景组合。高频的挂即时和事件，兜底的挂定时，决策的留按需。

> **易错点 / 考点：** ① 四个来源、三种定义载体、四种触发方式，都是填空与选择高频；② RAML 与 HAR 的定义要一字不差，一个是描述语言，一个是文件格式，最容易互换；③ 常错：以为接口自动化就是单接口用例，业务价值在场景组装；④ 架构题按来源、定义、用例、回归的顺序答，逻辑最清楚。

> **过渡：** 架构讲的是平台要有什么，接下来看一个接口接进来要走几步。

【6.18】API 测试流程（3 分钟）

> 一个真实请求，到变成天天能跑的用例，中间有六道手。课件标题后面写着"API 的接入步骤—批量生成"。

>

> 第一步，HAR 文件。从浏览器或抓包工具把真实请求导成 HAR。为什么从这儿起？它等于免费拿到完整信息：URL、方法、头、Cookie、请求体，几秒钟的事。

>

> 第二步，过滤与保存。HAR 里噪音多：图片、样式、埋点、广告全在里面。按域名、路径、响应类型过滤，只留业务接口，存成结构化清单。这一步决定用例质量，垃圾进，垃圾出。

>

> 第三步，脚本的修改。HAR 是当时那一刻的快照：令牌会过期、订单号是旧的、时间戳写死了。所以要做参数化：把会变的值抽成变量，把前置数据改成动态获取。这一步最花人力，也是"录制回放"能不能变成"可维护用例"的分界。

>

> 第四步，Thrift code_gen。针对用 Thrift 这类 RPC 框架的系统：用接口描述文件生成客户端代码，而不是手拼报文。生成之后，调接口像调本地方法。

>

> 第五步，Dynamic Class 与 Dynamic Loading，再到 Dynamic Call Method。动态加载生成的类，动态调用目标方法。为什么要动态？接口清单是变的。把调用硬编码进脚本，接口一改就得重编；动态化以后，接入新接口变成加载一个新类。

>

> 第六步，动态测试。组装参数、发起调用、断言结果，形成能重复执行的用例。

>

> 六步的实质是一件事：把人点出来的请求，变成机器能重复执行的用例。每一步都在做同一类动作：去噪、参数化、解耦。

> **提问：**第三步"脚本的修改"为什么最花人力？不做好会怎样？

> **预设回答：**

- 答"因为 token 会过期"——最典型。不参数化，脚本第二天全部 401。

- 答"因为数据有依赖，下单要先有商品 ID"——也很关键。前置数据写死，环境一换就全红。

- 答"那我每次重新录一遍 HAR"——短期可行，长期是灾难。用例没有沉淀，等于每次都从零开始。

- 收口：这一步的产出不是"能跑的脚本"，是可维护的用例。这才是自动化的价值。

> **易错点 / 考点：**① 六个步骤的名称与顺序要能复述；② 常考概念：HAR 的作用是获取真实请求，参数化的目的是可维护、可跨环境；③ Thrift 属 RPC 框架，别和 REST 混；④ 动态加载与动态调用的动机是对付接口变化，这是架构题的高分点。

> **过渡：**理论说完，看两个真例子。先看 Swagger UI 怎么直接当测试工具用。

【6.19】Swagger UI 测试示例（2 分钟）

> 很多人点开 Swagger 页面，看一眼接口列表就关了，当它是个文档站。这一下少了半件工具。

>

> 第一件事，Swagger UI 不只是文档，它本身就是个轻量测试工具。一个接口展开之后，你能看到方法、路径、参数表、每个参数的类型和是否必填，还有各种响应码对应的结构。更关键的是那个"试一试"的入口：填参数、点执行，它把请求发出去，再把 curl 命令、请求 URL、响应体、响应头全部回显给你。也就是说，没写一行代码，你就能完成一次接口验证。

>

> 第二件事，它告诉你契约长什么样。参数表里那些标注，必填还是选填、类型是整型还是字符串、取值范围，就是你现成的测试输入清单。必填项全填、必填项漏填、类型故意给错、超范围给一个大数，这四个动作能覆盖掉大部分参数校验缺陷。

>

> 第三件事，它也暴露了"文档即契约"的脆弱。Swagger 文档常由代码注解自动生成，实现改了文档没同步，就会出现文档说必填、实际不校验，或者文档没这个字段、实际返回了。所以用 Swagger UI 测出问题时，先分清是实现的错还是文档的错。两类都要报，性质完全不同。

>

> 课件把这一页放在"API 测试示例"之前，用意很清楚：先用最省力的方式把接口摸一遍，再去写自动化脚本。别一上来就写代码。

提问：Swagger 文档上写着某个字段必填，你故意不传，接口照样返回 200。这是谁的缺陷？

预设回答：

- 答"文档的缺陷，改了就行"——不一定。要看这个字段业务上是否真的必需：缺了它下游会算错，那就是实现没校验。

- 答"实现的缺陷，必须校验"——多数情况是，但要分清协议层必填和业务层条件必填。

- 追问：怎么定？看契约的意图。所有场景下都不可缺，就必须校验；只有某些场景需要，文档要写清条件。

- 收口：缺陷报告要写清文档怎么写的、实际怎么表现、业务上怎样才对，别只写"没校验"。

易错点 / 考点：① Swagger 属于 API 定义工具，同时提供 UI 供调试，问"能不能用它做接口测试"要答能，但不适合做批量回归；② 别把 Swagger 和 SoapUI 混，前者以定义与文档为主，后者以测试执行为主；③ 缺陷定性（文档错还是实现错）是案例题常见追问。

> **过渡：**看一个更完整的例子，这次带断言。

【6.20】API 测试示例（2 分钟）

> 一条接口用例，写全了是五段。照这个骨架写，漏不了。

>

> 第一段，前置准备。要有可用的环境和数据：服务起来了、库里有测试账号、有可用的商品或订单。很多同学写的用例在别人机器上跑不起来，就是把前置条件藏在了自己脑子里。

>

> 第二段，请求构造。方法、路径、参数、头、体，五样齐了才算一个完整请求。头和鉴权最容易出事。接口测试里 401、403 类问题，根源往往不是逻辑，是令牌没带对或者过期了。

>

> 第三段，响应断言，这是示例的核心。断言要分层：协议层断状态码和内容类型；结构层断响应体里的字段在不在、类型对不对；业务层断字段的值符不符合预期，比如金额、状态、条数。只断第一层，等于没断。

>

> 第四段，数据校验。接口返回对了，还要看库里落对了没有。返回 200 但没写库，或者写了两条重复记录，这类缺陷只有查数据才看得见。

>

> 第五段，清理。用例要能反复跑，就得自己收尾：删掉造的数据、恢复状态。不能重复执行的用例，在回归里是负担，不是资产。

>

> 五段齐了，才叫一条能进回归的接口用例。课件给这一页的定位是示例，我更希望你把它当模板。以后每写一条，就问自己这五段齐不齐。

> **提问：**为什么"清理"这一步经常被跳过，后果是什么？

> **预设回答：**

- 答"因为麻烦，而且当下能跑通"——这就是它被跳过的原因，也是它在回归里爆炸的原因。

- 答"数据越积越多，最后环境脏了"——很具体。订单表堆满垃圾，查询变慢、统计失真，甚至会触发业务规则，比如每人限领一次。

- 追问：清理怎么设计？优先用独立的测试账号和隔离数据，实在不行再删；删不掉的，设计成幂等可重复的用例。

- 收口：自动化的可持续性，一半靠清理。

> **易错点 / 考点：** ① 接口用例的组成是简答高频，五段要说全；② 断言分层（协议、结构、业务）是评分点，只答"断言返回结果"拿不到满分；③ 数据校验常被漏，凡写操作都要验；④ 清理与幂等是工程化考点，论述题可以引用。

> **过渡：** 手工这样写太慢，下一页看接口测试工具的全景。

【6.21】基于 API 的 TA 工具（2 分钟）

> 接口测试的工具，名字能列一屏。真到选的时候，很多人反而想不起用哪个。

>

> 原因不复杂，没有分类。名字是散的，用途是聚的。按用途分四类，一下就顺了。分不清的时候你只能背名字，背完就忘。

>

> 第一类，执行工具。把请求发出去，做断言，出报告。第二类，定义和文档。它管接口长什么样，Swagger 是代表。第三类，Mock 和桩。依赖还没好，或者你想造一个异常响应，靠它假装对方。第四类，管理和协作。管用例、管环境、管谁在改。

>

> 课件列了四个，偏执行和平台。SoapUI 是老牌，SOAP 和 REST 都认，有界面，能把用例组织成套件。Rest-Assured 是 Java 里的 REST 测试 DSL，写起来像句子，后面单独讲。SOAtest 偏企业级，擅长复杂协议和消息格式。APIfortress 偏持续测试，也管一部分安全验证。

>

> 选的时候看三条。能不能进 CI。不能自动触发，它就只是一次手工活。断言够不够细。能不能按 JSON 路径断言，能不能校验 schema。维护成本。界面化好学，可上千条用例没法靠手点；代码化上手慢，好在能评审、能分支。

>

> 还有一句实在话：别为了工具换工具。团队是 Java，就先用 Rest-Assured。工具是给流程用的，不是给简历用的。

> **提问：**SoapUI 上手快，为什么不少团队后来还是转去 Rest-Assured 这类代码化的工具？

> **预设回答：**

- 答"能进版本管理"——这是关键一条。脚本和被测代码搁同一个仓库，能评审，能分支，能查出是哪次改坏的。

- 答"界面点几百条用例太慢"——也对，参数化和批量生成靠手点做不出来。

- 追问：那 SoapUI 就没用了？不是。探索阶段、临时验一下、非技术同学参与，它很快。

- 收口：看规模。几十条用例，界面工具最快；上千条，只能代码化。

> **易错点 / 考点：**四个工具的名字和定位要对得上。选择题爱把 Mockbin、wiremock 混进来，它们是 Mock 工具。选型三判据能当论述题的骨架：能进 CI、断言够强、维护成本可控。

> **过渡：**Mock 这一类单独拎出来讲，因为接口测试最头疼的就是依赖。

【6.22】基于 API 的 TA 工具 -续（3 分钟）

> 要测下单，可支付网关那边还在写代码。你等它，还是先把下单测完？

>

> 等不起。这时候用 Mock，假装对方已经好了。Mock 解决两件事：依赖没就绪，还有一种更要命，依赖不可控。第三方有时快、有时超时、有时返 500。这种偶尔，你没法求它真发生，只能自己造。

>

> 课件列了六个。Mockbin 能从 HAR 文件生成桩，把抓到的真实响应直接变成桩。RAP 是淘宝做的 API 管理工具，文档、Mock、协作一体。Swagger 是流行的 API 定义工具和规范，按定义能生成 Mock 响应。Easy

> Mock 从 Swagger 生成数据。Doclever 写 REST 接口文档，顺手生成测试数据。Mocky 不用登录，直接给你一个 response。

>

> 另外两个偏工程师向。kristofa/mock-http-server 是轻量 HTTP 桩服务。wiremock 在 Java 圈用得最多，能配请求匹配规则和响应模板，也能录制回放。

>

> 选 Mock 看三点。能不能按请求匹配。能不能造异常，超时、5xx、空响应。配置能不能进 CI，还是一个界面点点点。

>

> 它和上一页那四个执行工具不是竞争。一个假装对方，一个考问自己。配合起来才是一条完整的链。

> **提问：**第三方支付在生产上偶尔超时，你怎么用 Mock 把这种"偶尔"测出来？

> **预设回答：**

- 答"让 Mock 返 500"——只测到一种异常，测不到慢。

- 答"让 Mock 固定延迟几秒再返回"——对，这才是关键一招。延迟比报错更阴，很多系统对超时没有兜底，会一直等，把线程池占满。

- 追问：超时之后的重试和幂等怎么测？让 Mock 第一次超时、第二次成功，看系统重试会不会重复扣款。支付类系统的经典缺陷就在这儿。

- 收口：Mock 不只是造数据，它的高级用法是造故障。延迟、超时、乱序、部分失败。

> **易错点 / 考点：**六个工具的特点要对上，特别是 Mockbin 从 HAR 生成、Easy Mock 从 Swagger 生成数据、Mocky 免登录。最容易错的是把 Swagger 只当文档工具，它也能生成 Mock。还有概念：Stub 返回固定值，Mock 还能验证调用行为，别混。

> **过渡：**工具说完了，看一个把接口测试写成句子的框架。

【6.23】Rest-Assured（3 分钟）

先看一条断言。没学过 Java 的人，能不能读懂它？

>

`when().get("/lotto/{id}", 5)`。后面接 `then().statusCode(200)`。再往后是 `body(...)`，两个断言都写在括号里。翻成中文：当我去取编号 5 的那期彩票；那么状态码要是 200；lottoId 要等于 5，中奖者编号里要有 23 和 54。

>

能读懂，这就是 DSL 的意思。领域特定语言，为这件事专门造的一套说法。课件给的定位是：测试和验证 RESTful 服务的 Java DSL。它让基于 HTTP 的 RESTful 服务写测试更容易。

>

结构只有三段。given 放前提，参数、头、体。when 放动作，发什么请求。then 放断言，状态码、响应体、响应头。它和 BDD 是一个路子，所以不光好写，还好读。

>

三个细节。路径参数，{id} 被 5 换掉，这就是最朴素的参数化。JSON 路径断言，用点号一层层往下走，不用把整个响应反序列化就能断。Hamcrest 匹配器，equalTo、hasItems 都是它提供的。

>

上面那段 JSON 就是它要断的对象。一期彩票，七个开奖号码，两个中奖者。

>

拿到官网 rest-assured.io 和那个 GitHub 仓库，先把示例跑通。再故意改错一个期望值，让它红一次。故意失败，是看懂断言最快的办法。

提问：这套 DSL 让测试好读，好读对测试有什么实际用处？

预设回答：

- 答"好看一点而已"——不只。测试代码是会被别人读的。开发要看懂你报的缺陷，新人要接手你的用例，评审要判断你测得对不对。

- 答"能当文档用"——很到位。一条 given-when-then 的用例，本身就是这个接口的用法示例。

- 答"读不懂的测试没人敢改"——这是最实际的后果。不可读的脚本放三个月，就没人敢动了，然后烂掉。

- 收口：好读不是装饰，它省的是协作成本。

> **易错点 / 考点：**given-when-then 三段要能背，说清各段放什么。`body("a.b.c", matcher)` 的点号路径断言是常考点，它不是 XPath。它只测 RESTful 服务，语言前提是 Java 或者 JVM 系。

> **过渡：**三段里最花功夫的是参数怎么传。

【6.24】Rest-Assured 参数化（2 分钟）

> 接口报参数缺失，你先别怀疑服务端。

>

> 三种写法对着 HTTP 三个位置。位置错了，参数就白传。第一种，formParam 和 queryParam 一起用。课件那条：given() 先挂 formParam，再挂 queryParam。然后 post 到 /something。formParam 进请求体，按表单编码。queryParam 拼在地址后面。

>

> 第二种，通用的 param。given() 挂两个 param，去 get /something。它会自己判断：GET 当查询参数，POST 表单当表单参数。省事，但不精确。位置要紧的时候，还是写具体方法。

>

> 第三种，直接写进地址。`get("/name?firstName=John&lastName=Doe")`。最直白。代价是值里有空格、&、中文、斜杠，你得自己编码。忘了就出玄学问题。

>

> 三条写法背后是一件事：参数在报文里的位置，决定服务端怎么读它。放错了，用例红得没道理。被测系统没错，是你的请求造错了。这种假缺陷，在真实项目里占比不低。

>

> 所以动手前先看契约。这个参数在 query、在 path、在表单、在 body，还是在头里。别拿万能 param 糊

> 过去。

> **提问：**你用 queryParam 传了参数，服务端说参数缺失。可能是什么原因?

> **预设回答：**

- 答"服务端有 bug"——先别急。很可能是契约要求这个参数在请求体里，你放在地址上，它读不到。

- 答"参数名拼错了"——也是高频，特别是大小写、下划线和驼峰不一致。

- 追问：怎么快速分清是我的错还是它的错？用 Swagger 或者开发给的 curl 重放一次。官方示例通、你的不通，就是你的构造问题。

- 收口：先让请求和契约完全一致，再去怀疑实现。

> **易错点 / 考点：**三种写法的位置要能对上，选择题爱考。param 不只用于 GET，POST 表单也能用。URL 直接传参的编码问题——中文、`&`、空格——是实战易错点。假缺陷这个概念要会，报告里要分清"被测系统的缺陷"和"测试代码的缺陷"。

> **过渡：**Rest-Assured 单独跑不够，它得挂进测试框架。

【6.25】Rest-Assured + JUnit（2 分钟）

> 用例越写越多，它们散在几个 main 方法里，跑全量靠手点。

>

> 把 Rest-Assured 挂到 JUnit 上，看着是拼装，实际解决三个问题。

>

> 第一个，怎么组织。JUnit 给类和方法的结构，给用例前后的钩子，给失败时的报告。Rest-Assured 只管发请求、取响应、做匹配，它不管排序、统计、出报告。分工很清楚：JUnit 管组织，Rest-Assured 管交互。

>

> 第二个，怎么复用。一条接口用例，前面那截几乎每条都一样：基础地址、公共头、令牌。JUnit 的生命周期方法正好接这段。所有用例之前读配置、取令牌、建好基础请求；每条用例之前把状态重置。用例本体就只剩这次传什么、期望什么。

>

> 第三个，怎么进流水线。JUnit 是 Maven、Gradle 里的老熟人。构建工具认识它，CI 认识它，报告插件也认识它。你不用自己写一个执行器。这就把前面那条选型判据落到了地上。

>

> 有一处容易混。用 JUnit 跑接口测试，被测对象不是单元。借的只是它的组织和执行机制。按被测对象分，它是接口测试；按执行框架看，它跑在 JUnit 上。两件事不打架。

> **提问：**令牌只在所有用例之前取一次，会带来什么风险？

> **预设回答：**

- 答"省时间，没风险"——有风险。令牌有有效期，跑久了会过期，后面的用例集体 401。

- 答"每条都重新取"——更稳，但慢。折中是缓存加失效重取。

- 追问：怎么分清令牌过期和真的没权限？看响应体和错误码，再单独造一条用过期令牌访问的用例把它钉住。

- 收口：共享前置状态省时间，也带来耦合，失效要显式处理。

> **易错点 / 考点：**JUnit 管组织、Rest-Assured 管交互，这个分工是简答题的得分点。别以为接口测试必须用专门的接口工具，用单元测试框架完全可以。生命周期方法的用途是高频考点，和第 5 章接着。共享状态的失效处理，论述题能加分。

> **过渡：**把这些拼起来，就是一个完整的接口测试工程。

【6.26】REST API 测试（2 分钟）

> 工具都认识了，把它们拼成一个工程，要有什么？

>

> 五条，也是你大作业的检查单。

>

> 第一，工程结构要分层。配置层管地址、用户名、超时。接口层一个接口一个封装。用例层放场景和断言。数据层负责造和清。不分的后果很具体：换个环境，你要改几十个文件。

>

> 第二，把怎么发和发什么分开。given() 别散在每条用例里。封装层管基地址、公共头、日志、重试；用例层只给参数和期望。这和后面讲 Page Objects 是同一个想法，把操作细节从测试意图里剥出去。

>

> 第三，断言分三层。协议层看状态码和头。结构层看字段、类型、schema。业务层看这笔单子的金额对不对。关键接口要加 schema 校验，不然字段类型被悄悄改掉，你断不出来。

>

> 第四，执行和报告。一键跑全量，按标签挑子集，失败能重跑，报告里看得到请求和响应原文。报告难用等于没自动化，因为没人愿意看。

>

> 第五，环境和数据。一个环境一套配置。数据自己造、自己清。令牌和口令不进代码仓库。

>

> 这五条管的是能不能长期活下去。你以后接手一个烂掉的接口自动化项目，死因多半就在里面。接手那天先看两样：有没有分层，有没有人在看报告。

提问：为什么说报告不好用等于没有自动化？

预设回答：

- 答"跑完就行，报告只是记录"——错。自动化的价值要有人看结果才兑现，报告难读就没人看，红了也没人修。

- 答"要能看出哪一步失败"——对，所以失败时要把请求原文、响应原文、断言表达式都打出来。

- 追问：报告还缺什么？趋势。通过率随时间怎么走，哪些用例时红时绿。这比单次结果有用。

- 收口：自动化的产出不是脚本，是能信的信号。

易错点 / 考点：分层结构是设计题的标准答案。怎么发和发什么分离，这个原则要能表述，它和 Page Objects 同源。schema 校验属于结构层断言，别和业务断言混。报告和趋势是加分项，别只答生成一份 HTML。

过渡：工程化讲完，转到界面那一头。

【6.27】UI 自动化测试（2 分钟）

> 按钮点不动，弹窗关不掉，金额显示成 12.5 元。这类毛病，接口层永远测不出来。

>

> 课件给三条路线。Web 客户端的 UI 自动化。基于 Cypress 的 UI 自动化。Android 应用的 UI 自动化。

>

> 讲工具之前，先把这件事的内核说清，否则你会把它学成一堆命令。所有 UI 框架都在做三件事：找到元素、操作元素、验证结果。Selenium 也好，Appium 也好，区别只在怎么找、怎么操作。

>

> 它的价值落在三类缺陷上。元素层：按钮点不动、输入框被遮罩挡住、弹窗关不掉。流程层：五步操作走完状态丢了，返回键行为不对。展示层：数据错位、文案错、金额格式错。这三类，接口断言全看不见。

>

> 代价要认。慢，执行慢。脆，界面一改就红。贵，维护成本高。这三样放在一起，就注定了它站得高、占得少。所以行内的做法很一致：UI 自动化只守关键路径，用例数量压到很少。放到你们的课设里，就是三五条，把下单主流程走通，别去点每一个按钮。

>

> Web 这两条路子技术方向正好相反，稍后并排看。安卓是另一套生态，难点在进程模型。带着一个问题听：它们各自靠什么找到那个按钮？

提问：所有框架都在做找元素、操作元素、验证结果，那它们的差别在哪？

> **预设回答：**

- 答"差别在语法"——语法是表面。差别在机制：靠什么通道操作界面，测试进程和被测程序是什么关系。

- 答"差别在支持的平台"——也对。可同一平台也有完全不同的机制，Selenium 走外部驱动，Cypress 跑在浏览器里。

- 追问：机制为什么重要？它决定你能看到什么、能做什么、会踩什么坑。跨进程的框架拿不到应用内部对象，只能模拟点击。

- 收口：这一节不是记 API，是弄懂机制。

> **易错点 / 考点：**三条路线要能背。判断题"UI 自动化能替代手工测试"——错，它只覆盖可判定、可重复的那部分。三步骨架很通用。UI 自动化脆，根源是界面和脚本耦合，解药在 Page Objects。

> **过渡：**先看 Web 端最老的那条路。

【6.28】Web UI 的自动化测试（3 分钟）

> 装个浏览器插件，点几下鼠标，脚本就有了。听着太美。

>

> 这是 Web UI 自动化最老的入口：Selenium IDE。你在页面上操作，它把每一步记下来；点运行，它重放一遍。装上 Chrome 的插件，打开页面就能录，不用配环境。零代码门槛，这是它最大的好处。产品、运营、刚入门的同学，都能拿它做一次快速验证。

>

> 课件给的是初始化界面，两个设置项。Project name 给这个工程起个名字，方便管多个录制集。Base URL 是被测应用的根地址。根地址为什么要单独拎出来？因为同一个脚本要在不同环境跑。今天测试环境，明天预发环境，换掉 Base URL，脚本里的相对路径跟着走。这就是最原始的环境参数化。

>

> 局限得讲透。第一，录出来的脚本脆。它按你点到的元素记定位，界面结构一变就找不到。第二，不好维护。几十条脚本之后，改一个公共步骤要改几十处。第三，难进流水线。浏览器插件不适合在 CI 机器上无人值守地跑。第四，不可编程。循环、条件、数据驱动，写不出来。

>

> 所以它的位置很清楚：入门、演示、快速验证、生成初始骨架。要规模化，必须走到代码化的 WebDriver。

> **提问：**Base URL 为什么要单独做成一个设置项，而不是把完整地址写进每一步？

> **预设回答：**

- 答"方便改"——对，但要说清是为跨环境。测试、预发、生产，域名不一样。

- 答"因为路径会变"——路径一般不变，变的是域名。这正是拆开的意义。

- 追问：一个脚本里混着几个服务的地址呢？那就按服务拆成多个配置项，一个地址一个变量，别硬编码在步骤里。

- 收口：这就是配置和脚本分离的最小实践，后面所有自动化都在做这件事。

> **易错点 / 考点：**Selenium IDE 的定位是录制回放插件，它的四条局限是简答高频点。录出来的脚本不能直接拿去回归，得先参数化、加断言、去硬编码。Selenium 家族的顺序是 IDE、RC、WebDriver，RC 已经淘汰，别答反。

> **过渡：**录完的脚本长什么样？看它录完的那一版。

【6.29】Web UI 的自动化测试 –续（2 分钟）

> 录完的脚本跑绿了，你敢不敢签收？

>

> 从这条脚本里要读出四件事，它们决定你后面写 WebDriver 时的思路。

>

> 第一，脚本是步骤序列。每一步三样东西：动作、目标、值。动作是 type，目标就是那个输入框的定位表达式，值是要输的文字。这个三元组是所有 UI 脚本的最小单元。WebDriver 里也是这三样，只是换成了方法调用。

>

> 第二，定位器决定它能活多久。录制工具一般优先用 id、name，没有就退到 CSS 或 XPath。路径式最脆，界面加一层 div，路径就变了。所以看录制脚本，头一件事是查定位器是不是稳定标识。不是，就手工换成 id、name，或者稳的业务属性。

>

> 第三，断言要自己补。录制只记你点了什么，记不了你期望看到什么。所以录完的脚本默认没有断言。它只能证明操作没报错，证明不了结果对。这就是录制回放最容易被误用的地方：一遍绿，什么都没验证。

>

> 第四，等待要自己处理。脚本里常是点完立刻找下一个元素，真页面是异步加载的。于是它时快时慢、时红

> 时绿。这不叫环境不稳，这叫没做等待策略。

>

> 录制的价值是给你一个能跑的骨架。能不能用，看定位器、断言、等待这三件事补没补上。

> **提问：**一条录制脚本跑绿了，能不能说明这个功能是好的？

> **预设回答：**

- 答"能，流程通了"——它只说明操作没抛异常，不说明结果对。这叫零断言的假绿。

- 答"要加断言才算"——对，而且断言要断业务结果，比如订单号出现、金额正确，不是只断页面打开了。

- 追问：怎么判断一条脚本有没有价值？故意改坏被测系统的一处，看它会不会红。不红，就说明它什么都没验证。

- 收口：这招叫变异自检，土，但很管用。

> **易错点 / 考点：**动作、目标、值这个三元组是理解 UI 脚本的入口。只重视能跑通、不管断言，是常见错法，没有断言的自动化是假自动化。定位器的优先顺序是 id、name、CSS、XPath 绝对路径。固定 sleep 和显式等待的区别，考试和面试都爱问。

> **过渡：**录制只能应急。真要写脚本，先看能用什么语言写。

【6.30】Selenium 支持不同脚本语言（2 分钟）

> 同一套测试，Java 能写，Python 也能写。这件事本身在说明什么？

>

> 技术上看，Selenium 的核心不是某门语言的库，是一套协议。语言绑定只是把发命令这件事，包成你熟悉的语法。同一套操作，在各语言里长得不一样，发出去的东西是一样的。

>

> 所以别把 Selenium 当成 Java 的某个库。它是跨语言的自动化协议，加上各语言的客户端。

>

> 工程上，多语言的价值是让测试跟着团队的技术栈走。团队是 Java，测试就用 Java，工具链能复用，构建

> 能共用，开发也看得懂你写的用例。团队是前端背景，就用 JavaScript。测试代码不是孤岛，它要和产品代码住在一个工程里。反过来说，一个人会两门语言，也比不过一个把定位和等待做扎实的人。

>

> 但多语言不等于可以随便混。一个项目里今天 Java 一批、明天 Python 一批，结果是环境装两套、报告合两处。谁都不敢重构。统一语言，比用我最熟的语言更要紧。

>

> 还有一句。语言绑定是外壳，决定脚本质量的是定位策略、等待策略、断言设计、页面封装。这四件事换哪门语言都一样。别在学哪门语言上纠结太久，先把这四件事练明白。

提问：组里一个只会 Python，一个只会 Java，你建议统一用哪个？

预设回答：

- 答"各写各的，Selenium 都支持"——技术上可行，工程上是灾难。两套环境、两套依赖、报告分裂。

- 答"跟产品代码保持一致"——最稳的依据。构建和工具链能复用，开发也愿意维护。

- 追问：产品是 Java，可只有一个人会 Java？那就让另一个人补 Java 基础。测试脚本的语言要求远低于开发，几周够上手。

- 收口：这不是技术选型，是协作成本。

易错点 / 考点：Selenium 支持多语言是常识考点，但它不代表脚本能跨语言复用，换语言要重写。别把 Selenium 当成某门语言的测试框架。决定脚本质量的是定位、等待、断言、页面封装这四件事，跟语言无关。

过渡：语言定了，看干活的那一层。

【6.31】Selenium + WebDriver（3 分钟）

Selenium 这个名字，是一家人，不是一个人。

>

> 家里有四个成员。IDE 负责录制回放。RC 靠往页面注入 JavaScript 驱动浏览器，已经淘汰。WebDriver 是今天的主力。Grid 把用例分发到多台机器、多个浏览器并行跑。所以今天说用 Selenium 做自动化，基本就

> 是用 WebDriver 的 API 写脚本。RC 和 WebDriver 的对比，考试爱出。

>

> 它怎么工作？测试代码调 WebDriver 的 API，找元素、点击、输入。WebDriver 把这些调用翻成标准命令，用 HTTP 发给浏览器驱动。驱动再去操作真浏览器。两个词要紧：标准命令，脚本就不依赖某家浏览器的私有接口；真浏览器，你测的是用户真在用的那个，不是模拟器。

>

> 最小脚本四步。启动驱动，打开地址，找元素并操作，断言再关闭。这和找到元素、操作元素、验证结果是对上的。找元素最费功夫，它决定稳不稳。断言最容易被偷工减料，它决定有没有价值。

>

> 一条纪律：每条用例结束都要关浏览器，或者交给框架统一管生命周期。有人写完不关，跑一晚上开了几百个窗口，把机器拖死。脚本自己也要讲资源纪律。

> **提问：**为什么说 WebDriver 比早期的 RC 可靠？

> **预设回答：**

- 答"因为它新"——时间新不是理由，机制才是。

- 答"RC 靠往页面注入 JS，会被浏览器的安全策略挡住"——很准。注入式方案受同源策略和页面脚本环境影响，跨域和复杂页面经常失灵。

- 追问：WebDriver 凭什么绕开？它不注入，它通过驱动从浏览器外部下命令。这就是跨进程的思路，后面那张架构图讲的就是它。

- 收口：注入式受页面约束，驱动式受协议约束，后者稳得多。

> **易错点 / 考点：**家族四个成员和定位要对上，RC 已淘汰，Grid 做分布式。标准命令加真浏览器这两个词是简答得分点。别以为 WebDriver 要给浏览器装插件，它不是插件方案。

> **过渡：**这条链路画出来是什么样？看那张架构图。

【6.32】WebDriver 架构（3 分钟）

> 客户端下了一条命令，最后是谁真的点了那个按钮？

>

> 这张图分三段。值钱的地方不是方框，是边界。

>

> 第一段，客户端：你的脚本加语言绑定。它只做一件事，把点击这个元素翻成一条标准命令。它不碰浏览器。

>

> 第二段，传输层：HTTP，命令和结果是约定好的 JSON 结构。这一层的意思很直接。双方都守协议，客户端用哪门语言、驱动是哪家浏览器，都能配上。跨语言、跨浏览器就成立在这儿。

>

> 第三段，浏览器驱动加浏览器。驱动是独立进程，收命令、转成浏览器原生操作、再把结果回传。浏览器本身是被操作的对象，它不知道有人在测它。

>

> 三段之间是两个进程边界：测试进程、驱动进程、浏览器进程。这带来三个后果。一，问题能在协议层看到。脚本报元素找不到，那条命令返回了什么，错误信息来自驱动，很具体。二，端口和版本必须对得上。驱动要监听端口，版本要配浏览器，配错了连不上，报错还很不友好，新手最爱在这儿卡住。三，跨进程读不到内存里的对象。你只能按协议给的动作去操作界面，拿不到应用内部的变量和状态。

>

> 一句话：它是外部指挥，不是内部注入。它后面所有特点，都从这条边界推出来。

> **提问：**为什么要多一个浏览器驱动进程，让客户端直接连浏览器不行吗？

> **预设回答：**

- 答"因为要转换"——对。浏览器对外没有统一的自动化接口，各家原生接口不一样，驱动就是那个适配层。

- 答"因为权限和安全"——很关键。驱动是唯一被授权操作浏览器的东西，客户端隔着协议说话，出了事好定位。

- 追问：这对测试人员有什么好处？升级浏览器通常只换驱动版本，脚本不用改。这就是分层带来的稳定。

- 收口：多一层不是麻烦，是用隔离换来了可替换。

> **易错点 / 考点：**三段结构和两个进程边界，是最容易被考的结构题。别以为 WebDriver 直接控制浏览器内核，中间必须有驱动。环境出问题，先查驱动版本和端口。这一页还给后面安卓那条线做了铺垫。

> **【思政落点 4】**：这张图里藏着一个值得学的设计：用一个进程边界换安全。驱动是唯一被允许操作浏览器的角色，客户端只能按协议提请求。现实里的事故常常反着来：图方便把数据库口令直接给应用层，把管理员令牌塞进前端。每抄一次近路，就拆掉一道边界。做系统时先问自己：这一步我是不是越界了。守住边界，是对别人数据的基本尊重。

> **过渡：**架构看清了，往下看怎么组织页面元素。

【6.33】UI Object（2 分钟）

> 界面元素会变。今天 id 叫 login-btn，明天前端改成 submit。

>

> 如果你的定位散在二十条用例里，那天你要改二十处。

>

> UI Object 就是把一个界面元素当对象看。它有名字、有定位方式、有一组能做的操作、有一组能断言的状态。登录按钮不再是一个 XPath 字符串。它知道怎么被找到，知道自己能被点击，知道点完之后页面要变成什么样。名字也跟着业务走，loginButton 就比 btn1 好得多。

>

> 第一个理由就是刚才那个：元素会变，改一处和改二十处的差别。

>

> 第二个理由更要紧。它把怎么找和要干什么分开。用例里写点击登录按钮，不写找到 id 为 login-btn 的元素再调 click。前者是测试意图，后者是实现细节。混在一起，读的人累，改的人也累。

>

> 实操三点。定位器集中管理，一个元素一个定义，别重复。等待封在操作里，找元素这件事本身就带等待，

> 页面慢它自己等。元素的行为要收敛，别给一个元素对象挂一堆业务方法，那会把它养成一个什么都管的大对象。

>

> UI Object 管的是元素级的复用。往上一层，页面级的复用，是下一页的事。两层的分工只有一个：越小的东西，复用越广。

> **提问：**为什么找元素这个动作本身就要带等待?

> **预设回答：**

- 答"因为页面加载慢"——对，但更准确的说法是元素出现的时机不确定，可能网络慢，可能异步渲染。

- 答"那我在每条用例前面加 sleep"——能跑，代价大。短了不稳，长了整体变慢，还把不确定性摊给了每条用例。

- 追问：正确做法？智能等待。条件满足就立刻继续，超时才失败。

- 收口：等待不是一个技巧，是封装要背的责任。

> **易错点 / 考点：**UI Object 的四个组成能作简答：名字、定位、操作、断言。别把它理解成一个 XPath 变量，它是有行为的对象。定位器集中管理是维护性考点，和 Page Objects 接着。等待别写成固定 sleep。

> **过渡：**元素抽象完，往上走一层。

【6.34】Page Objects（2 分钟）

> 登录这一步，二十条用例都要做。代码抄了二十遍。

>

> Page Objects 就一句话：一个页面，或者页面里的一块区域，对应代码里的一个类。类里放两样东西，这个页面上的元素，和在这个页面上能做的操作。用例不直接碰元素，只调页面类的方法。

>

> 登录页对应 LoginPage。里面有用户名框、密码框、登录按钮，还有一个 login(user, pass) 方法。用例里写 loginPage.login("tom","123456")，读起来就是业务语言。前端把按钮 id 改了，你只改 LoginPage 里

> 那一行，所有用例跟着好。

>

> 它治三个真痛点。定位器分散，一个按钮的定位散在几十条用例里，现在收到一处。脚本不可读，以前满屏 findElement(By.xpath(...))，现在全是业务动作。重复代码，多条用例都要登录，现在只有一份实现。

>

> 三条纪律。一，页面对象里不写断言。断言属于用例，页面对象只负责做动作、取状态。同一个页面在不同用例里预期不一样，写进去就锁死了。二，不写跨页流程。登录完跳首页再点购物车，那是用例或者上层流程对象的事。三，一个页面一个类，别搞超大基类。继承一深，改一处崩一片。

>

> 课件给的参考是 Selenium 官方 wiki 的 PageObjects 页，值得看。这套约定是社区磨了很多年磨出来的。

> **提问：**页面对象里不写断言，凭什么？写进去不是更方便？

> **预设回答：**

- 答"一个方法连点带验，多省事"——短期省事，长期是坑。同一个页面在不同用例里预期不同，写死就打架。

- 答"断言属于测试意图"——很到位。断言是我要验证什么，页面对象只回答我能做什么、现在是什么状态。

- 追问：取状态和断言怎么分？页面对象给 getErrorMessage() 这样的取值方法，断言写在用例里：assertThat(loginPage.getErrorMessage()).contains("密码错误")。

- 收口：页面对象提供事实，用例判断对错。

> **易错点 / 考点：**Page Objects 的思想和三条纪律是简答高频点。把断言写进页面对象，是模式误用的典型。一个页面一个类，要能举例说清元素和方法归谁。层次关系要清楚：UI Object 管元素，Page Objects 管页面。

> **过渡：**页面对象要手写初始化，太啰嗦。Selenium 给了个工厂。

【6.35】PageFactory（2 分钟）

> 元素一多，页面类开头全是同一行代码。

>

> driver.findElement(By.id("username"))，一行一个。抄到第十个就想放弃。这种重复最耗耐心，也最容易抄错一个字。

>

> PageFactory 换了个写法。在字段上写注解，声明这个元素怎么找；构造的时候调一次初始化，框架把元素准备好。于是页面类开头变成一份元素清单。用户名对应哪个定位，密码对应哪个定位，登录按钮对应哪个定位，一眼看完。

>

> 三个好处。声明和使用分开，定位写在字段上，方法里直接拿来用。可读，新人打开类就知道这个页面有哪些元素。延迟查找，不必创建页面时立刻找到，真用到的时候再找。对异步渲染的页面友好，能少掉一批还没渲染完就找不到的失败。

>

> 有两条得说清。一，它不替代等待策略。延迟查找只解决什么时候去找，元素出现了能不能点、会不会被遮罩挡住，还得靠显式等待。二，它不是银弹。它只减样板代码，页面对象那几条纪律照旧要守。

>

> 建议按这个顺序练：先手写一版 Page Objects，再用 PageFactory 改写一遍。手写过，你才知道工厂替你省掉了什么。

提问：PageFactory 的延迟查找，能替代显式等待吗？

> **预设回答：**

- 答"能，反正都是等元素出现"——不能。延迟查找只决定什么时候去找，找不到它照样失败，它不会替你等。

- 答"不能，还得等可点击"——准确。存在、可见、可点击是三个状态，要靠等待条件分开。

- 追问：两者怎么配合？定位交给 PageFactory，状态判断交给显式等待，各管一段。

- 收口：能不能点，永远是等待策略的活。

> **易错点 / 考点：**两个动作要能说清：注解声明定位，构造时统一注入。常见错误判断是"用了 PageFactory 就不用管等待"。延迟查找适合异步渲染、元素多的页面，这是加分点。三层关系要能讲：UI Object、Page Objects、PageFactory。

> **过渡：**Selenium 这条线走完，换一条思路完全不同的。

【6.36】基于 Cypress 的 UI 自动化测试（2 分钟）

有的工具站在浏览器外面，有的工具直接住进浏览器里。

>

> Cypress 属于后者。它把自己定位成跑在浏览器里的测试工具，这句话是理解它的入口。Selenium 在浏览器外面，靠驱动和协议往里下命令。Cypress 钻进浏览器里面，和被测应用跑在同一个运行环境里。它不需要驱动，也不需要协议转译，它就是页面里的另一段 JavaScript。

>

> 三个好处。一，它和应用同呼吸。应用什么时候渲染完、请求什么时候结束，它都感知得到。很多场景不用手写等待，脚本里几乎没有 sleep。二，它看得见应用内部状态。网络请求、定时器、本地存储都在调试视图里，每一步还能回放当时页面的样子，这叫时间旅行调试。三，它跑在前端熟悉的生态里，同一套语言、同一个工程，接进来很轻。

>

> 代价也说清。受浏览器约束，同源策略、多标签页这类场景，它不如 Selenium 灵活。主打 Web，原生移动应用不是它的战场。它不是 Selenium 的替代品，Selenium 覆盖广、生态老、能测多浏览器多种类。测什么、团队什么背景，决定选哪个。

>

> 一句话先立住：Selenium 从外面指挥浏览器，Cypress 从里面和应用一起跑。

提问：Cypress 几乎不用写等待，这是优点还是隐患？

> **预设回答：**

- 答"当然是优点，省事"——省事是真的。它为什么能省？因为测试代码和应用在一个环境里，能感知状态变化，不是靠魔法。

- 答"隐患是让人忘了等待这件事"——这话有水平。一旦碰上框架等不到的场景，比如跨域跳转，不会写等待的人就懵了。

- 追问：那学 Cypress 要不要先学等待机制？要。弄懂机制，才知道它什么时候会失效。

- 收口：所有自动帮你搞定的框架都一样，方便的背后是你得懂原理。

> **易错点 / 考点：**判断题：Cypress 属于 UI 自动化，不是接口测试工具。它和 Selenium 的机制差别是浏览器内还是浏览器外。别把它说成只能测前端组件，它能做端到端的 Web 测试。取舍按场景和团队答，别答谁更先进。

> **过渡：**差别光靠说说不清，下一页用一张图摊开。

【6.37】Cypress 架构（3 分钟）

> 把两条链路并排画出来，差别不用讲，一眼就看完了。

>

> Selenium 是四段。测试脚本在 JVM 或者 Python 进程里，通过 WebDriver 协议发 HTTP 请求。浏览器驱动接单，驱动再去操作浏览器。中间隔着进程、隔着网络、隔着一次协议翻译。

>

> Cypress 把这条链砍成两段。一个 Node 进程负责打包测试代码、注入浏览器，剩下的事全在浏览器里完成。测试代码和被测应用并排跑在同一个运行环境、同一个域名上下文里。

>

> 这个结构带来三个后果，也就是三个词。

>

> 第一个词，同源。测试代码就在页面里，能直接读写 DOM。能直接摸到应用挂在 window 上的东西，能直接看到页面发出的每一个请求。要拦截和伪造一个响应，对它来说只是改一个函数，不用在中间架代理。

>

> 第二个词，自动等待。它的命令不是执行一次、失败就报错，而是反复重试直到超时。你在断言里写这个列表一共五条，页面两秒后才补齐，它不会立刻红，它会等。

>

> 第三个词，时间旅行。每一步执行完，它给当时的 DOM 拍一张快照。用例失败时，鼠标悬停到某一步，右边就回到那一刻的页面。不是截图，是能查的 DOM。

>

> 这些好处都有代价，代价恰好长在同源这两个字上。测试代码依附在一个浏览器上下文里，多标签页、多会话并行这类场景，做起来就别扭。所以它不是 Selenium 的升级版，是另一种取舍。

提问：A 标签页登录甲用户，B 标签页用乙用户操作，验证两边看到同一笔订单。用 Cypress 写会卡在哪一步？

> **预设回答：**

- 答"cy.visit 两次就行"——不行。两次 visit 是同一个标签页重新加载，不是两个并存的会话。

- 答"用 window.open 开第二个"——也不行。应用自己开出来的标签页，Cypress 不去接管。这是它明确的能力边界，不是 bug。

- 答"换 Selenium 或者 Playwright"——正解。选工具先看场景。单页单会话、前端为主，Cypress 舒服；多标签、多会话、跨浏览器矩阵，Selenium 更合适。

- 收口：架构不是谈资，它直接决定这个场景你做不做得出来。

> **易错点 / 考点：**本质差别从执行位置答：Selenium 是进程外遥控，跨进程跨网络；Cypress 和应用同源同上下文运行。由这一条派生出四个差别：要不要显式等待、能不能时间旅行、能不能直接拦改网络、能不能多标签多会话。判断题"Cypress 支持多标签页测试"——错。自动等待不是睡一会儿，是命令重试到超时，所以脚本里别出现 `cy.wait(5000)`。

> **过渡：**架构看完了，去验一下它那句宣传语。

【6.38】安装和使用简单（2 分钟）

> 两条命令，装完就能跑。这么便宜的事，为什么还要讲？

>

> 第一条，npm install cypress --save-dev。后面这个 --save-dev 把它记进开发依赖，不记进生产依赖。测试工具别被打进发布包。安装还有一步隐形成本：它会下载一个自带浏览器的运行时，包不小。国内网络卡在这一步是常态，要配镜像或者走代理，这是你们动手的第一个坎。装不动就先换源。

>

> 第二条，npx cypress open。npx 优先用项目本地的那份可执行文件，不动全局的。这条命令拉起图形界面的运行器。第一次跑还会顺手建好目录骨架。用例放 e2e，测试数据放 fixtures，自定义命令和钩子放 support。配置在 cypress.config 里。这叫约定优于配置，你不用告诉它用例在哪，放对地方它自己找。

>

> 课件还给了官方指南的入口，安装文档里挂了段演示视频。路径很清楚：装，打开，照着模板改一条，跑起来，再谈复杂的。

>

> 我要多说一句。简单这两个字靠得住，靠的不是命令少，是失败信息清楚。用例红了，它告诉你哪一步、当时页面长什么样。这才是简单的真意思。

提问：为什么强调用 npx cypress open，而不是全局装一份再敲 cypress open?

预设回答：

- 答"npx 命令短"——只是表象。

- 答"npx 用项目本地的版本"——这才是关键。多人协作、CI 上跑，必须锁同一版本。全局安装就变成我这能跑、你那报错。

- 答"全局装一份多个项目共用，省空间"——听着省，实际是把几个项目的工具版本绑在一起，升一个动全部。

- 收口：工具版本跟着项目走，锁文件和 node_modules 才是事实。

易错点 / 考点：命令本身考得不多，但实验必踩，评分会看。--save-dev 和 --save 的区别；npx 和全局安装的区别；cypress open 是交互式运行器，cypress run 是命令行无头跑 CI，这一对爱出选择题。

【思政落点 5】：两条命令就能跑起来，这是它最大的卖点。可得留一句：工具越容易上手，越容易让人跳过思考。跑通之后，很多人的第一反应是赶紧录个脚本交差，不是问我要验证什么、断什么。门槛不在安装，在设计用例、选稳的定位器、让脚本在别人机器上也绿。易用性降的是入门成本，替不了判断力。

过渡：装好了，脚本怎么写？下一页给骨架。

【6.39】Cypress 脚本（3 分钟）

> 一条用例，三行代码，能覆盖多少场景？

>

> 先看骨架。describe 包一组用例，it 是一条用例，里面是一串链式的 cy 命令。访问页面用 cy.visit，定位元素用 cy.get 加选择器。输入接 .type()，点击接 .click()，断言接 .should()。就这么几样，够覆盖八成场景。

>

> 三个写法要点。一，命令是异步的，但你看不到 Promise。里面没有 await，也没有回调套回调。Cypress 把命令排进一条队列，按顺序执行，失败重试。所以你写完点击立刻写断言，也不会因为还没渲染完就红。

>

> 二，断言挂在命令链上。cy.get('.item').should('have.length', 5)。should 失败会把前面的 get 重新跑一遍，直到超时。这是它和 Selenium 最要紧的一处分岔：等待不是一件单独的事，是命令自带的东西。

>

> 三，别写 cy.wait(3000)。这是初学者最爱干的事，等于把自动等待退化成死等。快的时候白等三秒，慢的时候三秒又不够。正确做法是等条件：等元素出现，等某个请求回来。

>

> 三条禁忌跟着说。一，别把命令的返回值赋给普通变量带出链外，那时候它还没执行；要取值就在 .then() 里取。二，别只写操作不写断言，没有断言的用例绿了也不说明任何事。三，别在一条用例里做太多事，它失败时要能立刻指到一个原因。

> **提问：**`cy.get('.list-item').should('have.length', 5)`，此刻页面只渲染了 3 条，2 秒后补齐到 5 条。这条用例通过还是失败？如果 8 秒才补齐呢？

> **预设回答：**

- 答"立刻失败，当时只有 3 条"——这是拿 Selenium 的思维在答。should 会重试。

- 答"会通过，它重试到补齐"——对，前提是在默认超时窗口内。

- 追问 8 秒：窗口内等不到就失败。这时把这条命令的 timeout 调大，或者先等列表接口返回，别塞一个 cy.wait(10000)。

- 收口：自动等待是给条件一个窗口，不是无条件等这么久。这个区别决定用例稳不稳。

> **易错点 / 考点：**Cypress 命令不是 Promise，不能 await，也不能跨命令赋给普通变量，这是高频错点。should 会自动重试，expect 不会，它一般写在 .then 里，这一条最常考。cy.wait 加数字是坏味道，评审要拦。口诀：get 定位、type 输入、click 动作、should 断言，全程自动重试。

> **过渡：**脚本写完，怎么跑，在哪跑，跑完看什么。

【6.40】Cypress 执行（2 分钟）

> 写完脚本，第一次跑，敲哪条命令？

>

> 开发的时候用 npx cypress open。拉起运行器，用例文件列在一侧，点哪个跑哪个。跑起来，面板里是每一步命令的清单，旁边窗口是应用的实时画面。鼠标移到某一步，页面就回到当时的 DOM，这就是上一页说的时间旅行。用例红了，你直接看见失败那一步页面什么样，不用猜，不用加打印。

>

> CI 里用 npx cypress run。无头执行，终端出日志，默认录视频、失败自动截图，产物落在固定两个目录里。这条才是自动化的形态。它能指定只跑某个文件，指定浏览器，把结果传到面板上看历史趋势。

>

> 有人把它记成开发用 open、上线用 run。这样记也行，只是漏了一半。差别另有一处，在产物：open 让你看见过程，run 给你留下证据。评审要看的是证据。

>

> run 的配置也比 open 严。基准地址没配好，相对路径直接报错。在容器里跑还要依赖齐全，缺共享库就起不来。这些是工程问题，不是框架问题。

>

> 最后一句。跑起来只是及格，跑完能定位问题才叫能用。会跑说明你会用工具，会定位说明你会做测试。你们交实验的时候，截图、日志、报告和脚本一样重要。

> **提问：**团队要求每次提交后自动跑一遍 UI 用例，还要保留失败现场。用哪条命令？还要配什么？

> **预设回答：**

- 答"用 open，让机器开着界面跑"——CI 机器通常没有显示器，open 是交互式的，不适合自动化。

- 答"用 run，截图和视频它自己存"——对。产物路径固定，CI 直接归档成构建产物。

- 答"还要把结果接到面板里"——加分。只有终端日志，失败要人工复现；有视频和截图，定位能省一大半时间。

- 收口：自动化的产出物设计和用例本身一样重要。

> **易错点 / 考点：**配对题：open 对交互调试和时间旅行，run 对无头和 CI。判断题"Cypress 只能在 Chrome 上跑"——错，可以指定浏览器。还有一个隐性考点：交付物包含日志、截图、视频、报告，不只是脚本。

> **过渡：**Web 这条线到这里。下一段把视线移到手机上。

【6.41】Android App UI 自动化测试（3 分钟）

> 一个安卓应用刚装到手机上，你要让机器替你点一下，用什么？先把名字摆出来。
> Robotium、UIAutomator、MonkeyRunner、Appium。另外四个：
> Espresso、Selendroid、ATAF、TestComplete。光背没用，得知道它们各站哪条线。

>

> 第一个维度，测试代码跑在哪一侧。Robotium、Espresso、Selendroid 走仪器化路线。测试代码打成测试包，和被测应用一起装进设备，在设备内部跑。UIAutomator 也在设备上，但它走系统的控件树通道，所以能跨应用。MonkeyRunner 不在设备上。它是电脑侧的 Python 脚本，靠 adb 遥控。Appium 也在电脑侧，能力靠借。安卓上复用 UIAutomator，iOS 上复用苹果的测试框架。对外统一成一套 WebDriver 协议。

>

> 第二个维度，能不能跨出自己这个应用。只测自己家的，是前面那三个。能把我的应用、系统弹窗、别人的应用串起来测的，只有 UIAutomator。

>

> 第三个维度，单平台还是跨平台。Appium 是一套 API 同时盖住 Android 和 iOS 的那个。TestComplete 是

> 商业工具，主打录制回放、低代码。ATAF 是腾讯开源的，把 UIAutomator 包了一层，更好用。Robotium 还多一条：原生界面和内嵌网页都能操作。Selendroid 给老版本安卓补位，现在基本退场。

>

> 交叉一下，选型就出来了。要快、要白盒、只测自己，选 Espresso。要跨应用，选 UIAutomator。要跨平台、要黑盒，选 Appium。电脑侧遥控，选 MonkeyRunner。

>

> 这张名单不是排行榜，是分工表。

> **提问：**需求是"从我们 App 点一个链接，跳到系统浏览器，再返回 App，验证页面状态还在"。这个场景用 Espresso 能测吗？

> **预设回答：**

- 答"能，Espresso 是官方推荐的"——官方推荐也有边界。它跑在被测应用进程内，看不见系统浏览器的控件。

- 答"不能，它只能在自己应用里操作"——对。跨应用要靠 UIAutomator，或者用 Appium，底层也是 UIAutomator。

- 追问"那 Espresso 还有什么用"——应用内快速、稳定的细粒度验证。常见组合是 Espresso 测内部流程，UIAutomator 测跨应用链路。

- 收口：所以工具谱系不是谁替代谁，是谁负责哪一段路。

> **易错点 / 考点：**常考"工具—机制—能力"三列匹配。两个高频错：把 MonkeyRunner 当成 Monkey，前者是写脚本的 API，后者是灌随机事件的命令行工具；把 Appium 理解成另一种 Robotium，错在它是外部驱动加协议统一。口诀：仪器化三个在应用里，UIAutomator 跨应用，MonkeyRunner 在电脑上，Appium 两头通吃。Selendroid 是给老版本安卓补位的方案，现在基本退出使用。

> **过渡：**这些差别看着零散，根子只有一条：测试进程和被测应用，是不是同一个进程。

【6.42】两种基本的自动化测试机制（4 分钟）

> 上一页那八个名字，其实一刀就能分开。刀口在哪？

>

> 第一类，Appium 这一类。课件写得准：Appium 基于 UIAutomator 框架实现。测试进程与目标应用进程是分开的。分开意味着什么？测试端拿不到应用里的任何对象引用。改不了一个控件的属性，读不到一个内部变量，调不了一个私有方法。它能做的只有一件事：往系统里注入一个点击、一次滑动、一段文本。然后通过控件树看界面变成了什么样。所以它只能模拟触发事件。

>

> 两个进程，不是两个线程。这个说法得掰清楚，否则后面全乱。

>

> 第二类，Robotium、Espresso、Selendroid。它们基于 Instrumentation 框架。测试进程和目标应用在同一个进程里，作为两个不同的线程运行。同进程就共享地址空间，对象引用就是能直接用的指针。所以测试代码能调 Activity 的方法。能断言某个控件此刻的文本，也能在界面线程上同步执行。

>

> 落到写用例的手感上，差别很直观。第一类只能"点一下、看一眼"。你没法问这个按钮的 enabled 到底是真是假，只能看它长得像不像禁用。第二类可以直接构造对象、直接调业务方法。课件说 Espresso 能写出类似白盒测试那样、更美观的脚本，根子就在这儿。

>

> 代价也得摆平。同进程观测能力强，可应用一崩测试跟着崩，还要求两边签名一致。跨进程隔离好、抗崩溃、能测别人的应用，但慢、脆，控件树一变就失效。

>

> 一句话，进程边界就是能力边界。

> **提问：**为什么"同进程"就一定能摸到应用内部的对象属性，而"跨进程"就一定不行？背后是操作系统的什么机制？

> **预设回答：**

- 答"同进程共享内存"——方向对了。同一进程内所有线程共享地址空间，对象引用就是能直接用的指针。跨进程只能靠进程间通信，传过去的是序列化后的数据，不是对象。

- 答"跨进程也能读界面啊，不是有控件树吗"——对，但控件树是系统愿意给你看的公开视图，只有它肯暴露的属性。应用内部的私有字段、临时状态，树里没有。

- 追问"那安卓为什么设计成跨进程也能测"——要测别人的应用、要测系统弹窗，还要防测试代码把被测应用搞崩。安全与隔离的代价，就是观测能力被裁剪。

- 收口：能不能看到内部，不是工具好坏，是进程边界定的。

> **易错点 / 考点：**极高频，几乎必考。标准问法：Appium 与 Robotium 在实现机制上的根本区别？答法：Appium 基于 UIAutomator，测试进程与被测应用分离，只能模拟触发事件；Robotium 基于 Instrumentation，同进程两线程，可直接访问应用元素属性。再记一组代价对照：同进程等于能力强、耦合紧、容易被带崩；跨进程等于隔离好、通用性强、只能黑盒。判断题"Appium 可以读取被测应用内部变量"——错。

> **过渡：**同进程这件事听着像魔法，其实靠一个很具体的手段实现。

【6.43】Instrumentation 框架（3 分钟）

两个安装包，系统凭什么把它们当成一个进程？答案在签名上。

>

> 课件那句话是关键：通过签名来实现，让系统认为测试 App 和被测 App 是同一个进程。为什么签名能做到？安卓靠用户 ID 做进程和数据隔离，而这个 ID 来自应用签名。

>

> 往下推一步。测试包和被测应用用同一张证书签名，并在清单里声明共享用户 ID。系统于是把它们分到同一个用户、同一个 Linux 进程、同一份数据目录。于是测试线程和应用线程真的住进了同一个屋子。

>

> 这不是巧合，是前提条件。开发环境里新建一个仪器化测试，它自动拿调试签名签出来，正好和被测应用的调试签名一致。换成正式签名不一样，装上去就跑不起来。

>

> 第二个角色是执行器。测试包装进设备之后，谁去跑它？系统里有一个专门的运行器，负责加载测试类。它按 setUp、测试方法、tearDown 的顺序调用，再收集结果。它靠清单登记项里的目标包名，知道要测的是哪个应用。这个字段填错，运行时会直接告诉你找不到目标应用。这是新手第一号错误，每年都能见到。

>

> 第三个层面是意义。正因为同进程，测试代码才能操纵应用内部对象，才能在界面线程上同步执行，才能精确断言内部状态。上一页说的白盒能力，根就在这里。

>

> 签名看着只是个打包细节，它其实是这套机制的前提。

> **提问：**如果测试包和被测应用用了不同的签名，adb 装上去会发生什么？系统是怎么判断的？

> **预设回答：**

- 答"装不上，报签名冲突"——同包名不同签名确实冲突；这里更常见的情况是能装上、跑起来却说找不到目标。

- 答"能装上，但测试跑不起来，因为共享用户 ID 不成立"——准确。系统按签名判用户身份，身份不同就不共享进程，仪器化的前提就断了。

- 追问"那 CI 上怎么保证签名一致"——用同一份调试或测试签名配置，写进构建脚本，别让人手工签。

- 收口：签名不一致是环境问题里最难查的一类，它的表现是测试莫名跑不动，而不是一条清楚的报错。

> **易错点 / 考点：**① 同进程的手段是同签名加共享用户 ID，不是把两个包合并；② 清单里的目标包名填的是被测应用的包名，不是测试包的；③ 执行器负责加载与调度测试，它不负责定位控件。判断题"仪器化测试与被测应用是两个独立进程"——错。

> **【思政落点 6】**：整套机制能成立，靠的是系统相信签名相同就是同一个应用。签名在这儿不是一张图，是信任凭证：签同一个名，就共享进程、共享数据、共享权限。开发时随手共享、随手用调试签名，等于自己拆掉隔离墙的一块砖。私钥泄露，别人拿到的不是一份文件，是以你的身份运行的能力。技术上的边界，也是责任上的边界。

> **过渡：**执行器具体怎么用、有哪些参数，下面把命令摊开。

【6.44】InstrumentationTestRunner（4 分钟）

> 跑一条安卓用例，命令只有一行，参数却有三套口径。

>

> 先说要写什么。继承哪个基类，看你测什么。测界面、要真实启动的，用 ActivityInstrumentationTestCase2。只测生命周期回调的，用 ActivityUnitTestCase。测服务的用 ServiceTestCase。测内容提供者的用 ProviderTestCase。测应用级对象的用 ApplicationTestCase。纯逻辑、什么都不依赖的，用 AndroidTestCase。整个测试类只启动一次界面的，用 SingleLaunchActivityTestCase。还有一个通用的 InstrumentationTestCase。它能注入事件、能拿到仪器化实例，很多场景落在它身上。

>

> 八个基类，规律只有一条：按被测对象挑。

>

> 再说配什么。测试包的清单里，把 instrumentation 元素的目标包名设成被测应用的包名。AUT 指被测应用。绑错了，运行时就找不到目标。

>

> 再说怎么跑。adb shell am instrument -w，那个 -w 是等测试跑完再返回结果。不带任何可选参数时，跑所有测试，但排除性能测试。三个常用参数记准。-e func true 跑所有功能测试。口径就一条：源自 InstrumentationTestCase。-e unit true 跑所有单元测试。指的是不从 InstrumentationTestCase 派生的、非性能测试。-e class 指定单个测试类，写全限定类名。

>

> 为什么默认跳过性能测试？因为性能测试要单独控环境、单独看数据。混在功能用例里跑，量出来的是排队时间，不是执行时间。

>

> 所以选基类的时候就决定了它会被哪条命令选中。

> **提问：**你写了 30 条用例，25 条继承通用仪器化基类，5 条继承 AndroidTestCase。执行 am instrument -w -e unit true，会跑几条？为什么？剩下 25 条怎么跑？

> **预设回答：**

- 答"30 条全跑"——不对，unit true 有筛选口径。

- 答"5 条，因为 unit 只跑不从 InstrumentationTestCase 派生的"——对，这就是课件的口径。

- 答"剩下 25 条用 -e func true 跑，或者干脆不带参数跑全部"——正确。不带参数是全部，但排除性能测试。

- 收口：跑全部和按类跑的边界，是继承关系定的。选基类的时候就决定了它会被哪条命令选中。

> **易错点 / 考点：**三个参数的口径必背，也极易混。口诀：func 看继承（源自仪器化基类的功能测试），unit 看不继承（非派生的非性能测试），class 点名（单个用例）。另两个高频点：不带参数时排除性能测试，这个排除最常被漏；目标包名填被测应用，不是测试包。判断题"am instrument 默认执行性能测试"——错。

> **过渡：**仪器化这条线讲完，回头看一个更早、思路相反的工具。

【6.45】示例 1：Monkeyrunner（3 分钟）

> 名字只差半截的两个东西，年年有人把它们当成一个。

>

> 先把容易混的这层剥掉。Monkey 是命令行工具，往应用里灌随机点击和按键，做压力测试。你控制不了它点哪儿、点几下。MonkeyRunner 是一套 API，你写脚本。指定点哪个坐标、发哪个键、截哪张图，然后自己判断对错。一个随机，一个可控，两件事。

>

> 课件的定义很准：基于 Python 脚本实现复杂测试用例的界面测试工具。它通过坐标、控件 ID 操作应用的界面元素，截取测试执行的界面，做图像比较分析来发现问题。这一句里有三个能力，也埋着三个软肋。

>

> 能力一，坐标与控件定位。控件 ID 定位还算稳，很多场景只能靠坐标，按某个位置按下去。坐标的问题不用多说：换个分辨率就废，换个机型也废。

>

> 能力二，截图。它能把当前屏幕存成图片对象。

>

> 能力三，图像比较。这才是它的判据。把这一步的截图和基线图比，差异超过阈值就认为界面变了。用图像比对代替断言，好处是不用了解内部结构，坏处是界面一改版就全崩，动画差一帧就误报。

>

> 位置也说清。脚本跑在电脑上，通过 adb 和设备通信。所以它属于外部驱动那一类，和 Appium 同侧，不在仪器化那一侧。

>

> 工具箱里还有几件东西值得认清。InstrumentationTestRunner 是针对被测应用运行测试脚本的执行器。Test tools 是与 Eclipse IDE 集成的、构建测试的 SDK tools。测试包放在测试项目里，遵守命名空间。用例类从基类继承。MonkeyRunner 给自己的定位是：提供 API 开发测试脚本。在 Android 代码之外控制设备。

> **提问：**用图像比对做断言的界面测试，为什么在真实项目里很难长期维护？请给两个具体原因，并说明最要命的那个后果。

> **预设回答：**

- 答"界面一改就失败"——第一号原因。改个间距、换个字体、加载慢一拍，像素就不同。

- 答"不同设备渲染不一样"——第二号。分辨率、像素密度、系统字体、抗锯齿都会让同一界面像素不同，基线图没法跨设备复用。

- 答"误报多了就没人看了"——这句最要命。测试的价值在于红了就一定是问题。团队一旦形成"这条老红但没事"的共识，整套自动化就废了。

- 收口：现代做法是把断言放到结构层：控件属性、控件树、接口返回。图像比对只留给必须看外观的少数场景。

> **易错点 / 考点：**① Monkey 与 MonkeyRunner 的区别，随机事件工具对脚本 API，选择题高频；② MonkeyRunner 属于电脑侧外部驱动，不在仪器化系列；③ 图像比对的适用边界，答题要点是分辨率敏感、改版即失效、误报伤信任。口诀：Monkey 乱点，MonkeyRunner 照单点。

> **过渡：**它整套 API 只有三个类，分工干净。

【6.46】示例 1：Monkeyrunner - 续（3 分钟）

> MonkeyRunner 的接口面很小，统共三个类。

>

> 第一个，MonkeyRunner 类。它管两件事：把脚本连到设备或模拟器，等连接、指定设备序列号；还有为脚本创建界面、显示内置帮助。你可以把它当总控台，先在这儿把设备和脚本接上。

>

> 第二个，MonkeyDevice 类。它代表一台设备或一个模拟器，方法是"对设备做什么"。装和卸载软件包、启动某个 Activity、发键盘或轻触事件、运行测试软件包。你要做的操作，都从这里发出去。

>

> 第三个，MonkeyImage 类。它代表一张图，方法是"对图像做什么"。截屏、把位图转成各种格式、比较两个图像对象、把图片写进文件。比较这个动作要留意，结果要你自己给阈值，它不替你做判断。

>

> 典型链条是：连设备，装包，启动应用，点几下，截图，和基线图比。不一致就把图存下来当证据。

>

> 课件还补了一句关键的话：这套 API 能跨多个设备或模拟器跑一个或多个测试套件。也能基于它和 Python 开发一整套自动化系统。这句的意思是，它的价值不在单个脚本，在你用 Python 生态自己搭平台。读配置、遍历设备、出报告，全是 Python 的事，它只管和设备说话。

>

> 交代一句时代。它是 Python 2 时代的产物，官方早已不维护，新项目不要选。我们讲它，是因为它把"截图加比对"这条思路摆得最干净，这三个类的划分也是后来很多框架的雏形。

>

> 三个类，一条链，收在一张图上。

提问：如果用这三个类写一个"回归检查"，最关键的设计点放在哪里？为什么？

预设回答：

- 答"放在比较的阈值上"——部分对。阈值决定灵敏度，调太松会漏，太紧会误报，可它不是最脆的地方。

- 答"放在截图前的等待上"——很实在。截图早一拍，界面还没渲染完，后面比什么都白搭。这正是图像比对的隐形难点：它没有自动等待。

- 答"放在基线图的管理上"——也关键。基线图就是期望值，期望值要版本化、要按设备分组，否则整套比对没

有意义。

- 收口：图像比对这条路，难点不在截图和比较这两个动作，在等待和基线管理。

> **易错点 / 考点：**三个类的职责常做连线：装包、发事件归设备类；截屏、比较归图像类；连接设备、建界面归总控类。比较返回的是布尔值，阈值要自己给，它不是断言，只是比较，这一点常被误答成"自带断言"。另外要能说出时代局限：Python 2、不再维护。

> **过渡：**看一段真实脚本，把三个类串起来。

【6.47】示例 1：Monkeyrunner - 续（2 分钟）

> 脚本第一行往往最不起眼，也最容易漏。

>

> 第一拍，导入与连接。从包里导入三个类，然后等连接，这一步会阻塞到设备就绪。不写它，后面全是空引用。

>

> 第二拍，装包与启动。先装应用包，再按包名和界面名启动。装包会返回成功与否，必须判返回值。很多人直接往下写，装包其实失败了，脚本还在那儿点空屏幕。

>

> 第三拍，等界面稳定。这里必须睡一下。为什么？因为 MonkeyRunner 没有控件级等待，它不知道界面渲染完没有。这是它和 Cypress、Espresso 手感上最大的差别：那两个是自动等，这里是靠你估。

>

> 第四拍，操作。按坐标按下、输入文本、按返回键。全部是事件注入。

>

> 第五拍，截图与比对。截一张图写进文件，然后和基线图比，或者留给人工看。课件说的截取界面、做图像比较分析来发现问题，落到脚本里就是这两三行。

>

> 五拍里，睡得准不准，决定这条用例是稳定还是随机失败。而睡多久只能靠估，估不出一个跨机型都成立的数。

>

> 这段骨架最大的教学价值，是让你看清它的判据有多软。不是断言等于什么，是两张图像不像。判据软，维护成本就硬。

>

> 所以要盯的四件事：连接、等待、操作、判据。

> **提问：**这段脚本里，哪一行最容易在别人的机器上跑不通？为什么？

> **预设回答：**

- 答"按坐标点的那一行"——最典型。这个坐标是特定分辨率下的位置，换机型就点到别处。

- 答"睡的时长"——也很典型。机器快了两秒够，慢了不够；写长了整体变慢，写短了随机失败。

- 答"应用包的路径"——属于工程配置问题，好修。

- 收口：前两个不是写得不好，是这类脚本的结构性缺陷。修的方向只有两个：少用坐标多用控件标识，把睡换成等条件。

> **易错点 / 考点：**实验里若用 MonkeyRunner，评分会看三处：有没有判装包返回值、有没有等待、坐标有没有按屏幕尺寸适配。考试层面记一句就够：这类脚本的判据是图像比对，稳定性依赖等待时长与分辨率适配，所以它注定是过渡技术。

> **过渡：**新一代的做法是把断言做进应用内部，让测试能直接问"你现在是什么状态"。

【6.48】示例 2： Robotium（3 分钟）

> 官方早就劝你别用的框架，为什么还占十分钟？

>

> 先看它留下的东西。四条特点。第一，支持对原生界面和 WebView 的操作。这条很实用。很多应用是原生外壳加内嵌网页的混合结构，只能测原生的框架，一进网页就抓瞎，它两边都能摸。第二，能自动支持多个

> Activity。老框架有个通病：界面一跳转就得重新拿一次引用，它帮你管这件事，跳转之后你还用同一个操作对象。第三，有单独的录制回放工具，录操作生成脚本。第四，可以和 Maven、Gradle、Ant 集成。这条是工程化的门槛，能进构建、能进流水线，才算真能用。进不去的框架，都变成某个人电脑上的脚本。

>

> API 风格也很集中，几乎都挂在一个操作对象上。按按钮上的文字点，按序号往输入框里填，等某个界面出现。用自然语言式的名字定位控件，它想让脚本读起来像一份操作说明书。课件的总结是：各种控件都能操作，能模拟手势，还有查找和断言的 API。

>

> 再看它为什么退场。它建立在已被废弃的旧测试基类之上，官方早就建议新项目别用。这不是它做得不好，是底下的基类先没了。

>

> 学它的理由只有一个：同进程白盒这条路线怎么落地，它是最好懂的样本。录出来的脚本当草稿很好，当交付物会拖垮项目。

> **提问：**既然录制回放这么省事，为什么我既说它是优点、又说它是陷阱？

> **预设回答：**

- 答"录制省事，做演示很合适"——确实。做样例、做冒烟、给不写代码的同事演示，都好用。

- 答"录出来的脚本没法维护"——对。录制器记的是当时的层级路径和位置，界面一调整就找不到元素，生成的代码通常又长又不可读。

- 答"那为什么还要有这功能"——它降低了入门门槛，也适合做录一遍、再人工改造成稳定定位的第一步。

- 收口：工具没有对错，看它当草稿还是当交付物。当草稿很好，当交付物会拖垮项目。

> **易错点 / 考点：**四个特点常出多选题；核心操作对象只有一个，这一点容易漏。判断题"Robotium 支持网页视图测试"——对。还要说清它属于仪器化、同进程机制。另有一个高分点：能指出它依赖废弃基类，属于历史上的正确答案。

> **过渡：**脚本长什么样，下面给一个典型例子。

【6.49】Robotium 脚本示例（2 分钟）

> 把两段脚本并排看，差别不在语法，在判据。

>

> 第一段，初始化。测试类继承针对某个界面的仪器化测试基类，构造时把被测界面的类传进去。在准备方法里先调父类的准备，再拿仪器化实例和当前界面，构造出操作对象。后面所有操作都靠这个对象。

>

> 第二段，用例本体。先等目标界面出现，断言当前就在这个界面；然后按按钮上的文字点击，按序号往输入框里填内容；再等某段文字出现。

>

> 第三段，断言。用标准断言方法，或者断言当前界面类。这里的断言是结构化的，不是图像像不像，是现在是不是这个界面、这段文字在不在。

>

> 和 MonkeyRunner 那段比，三处差别一目了然。它用文字和控件定位，不用坐标，可读性和稳定性高一个量级。它有等界面出现、等文字出现这类条件等待。虽然不如 Cypress 那种命令级重试聪明，至少是等条件，不是睡死。它的判据是断言，能精确指出哪条期望没满足。它还常用资源标识定位，比文字更稳，因为文字会跟着市场改。

>

> 这三条红利，全来自同进程白盒。它能直接问应用：你现在是哪个界面，这个控件上写的什么字。

>

> 所以自动化脚本的水平，先看判据，再看语法。

> **提问：**为什么"按按钮文字点击"比"按坐标点击"更稳？请从"定位依据"的角度说清。

> **预设回答：**

- 答"因为按钮名字不会变"——大致对，只要文案不改。文案改了也会失败，但失败原因是明确的找不到这个按钮，不是点偏了。

- 答"因为它找的是控件不是位置"——这才关键。控件是结构，位置是像素；结构相对稳定，像素随分辨率、语言、字号变。

- 答"文案也会改啊"——那就用资源标识定位，更稳。现代框架都主推按标识定位，原因在这。

- 收口：稳定性排序记一句：标识优于文本，文本优于层级路径，层级路径优于坐标。

> **易错点 / 考点：**实验里最常见的错是忘了先调父类准备、忘了拿当前界面，操作对象初始化就失败。考试层面牢记定位依据的稳定性排序，这条几乎能套到所有界面自动化题上。另一点：这类脚本必须用仪器化方式运行，不能像 MonkeyRunner 那样从电脑侧直接跑。

> **过渡：**Robotium 之后，官方给出了面向整台设备的方案。

【6.50】UI Automator 测试框架（3 分钟）

有的框架不管你的应用，它管整台手机。

>

> UI Automator 的位置就在这里：面向整台设备，不是面向某一个应用。课件给了三个组件，正好对上它的三层能力。

>

> 第一层，用于检查布局层次结构的查看器。它把当前屏幕的控件树导出来，每个节点的标识、文本、类型、区域都能看到。写定位器全靠它。这是看得见的能力。

>

> 第二层，用于检索状态信息并在目标设备上执行操作的 API。找控件、点击、输入、滑动、读属性。入口是两个对象：一个代表设备，一个代表元素。这是动得了的能力。

>

> 第三层，支持跨应用界面测试的 API。这条是它最要紧的一条。它走系统的无障碍通道，能看到所有应用画出来的控件树。系统设置、通知栏、权限弹窗、别人的应用，都在里面。所以"从我的应用跳到系统浏览器再回来"这类流程，只有它能测。

>

> 第三层也带代价。跨应用就意味着慢。每次查找都要向系统要一次控件树快照，比同进程直接读内存慢得多。而且它拿不到应用内部的私有状态，本质仍是黑盒。

>

> 工程位置上补一句。它的测试代码编译成独立的测试包，跑在设备上，不是电脑侧。所以它是第三条路：在设备内、在应用外。这一句能把它和另外两类彻底分开。

> **提问：**应用首次启动会弹"是否允许通知"的系统弹窗，这个弹窗不属于我们的应用。Robotium 和 UI Automator 分别能不能处理?

> **预设回答：**

- 答"Robotium 不行，它在自己进程里，看不见系统弹窗"——对。这是同进程路线的天然盲区，不少团队被卡过。

- 答"UI Automator 可以，因为它跨应用"——对。它能导到系统弹窗的控件，然后点允许。

- 答"还能用 adb 命令提前授权，绕过这个框"——工程上常用，可那叫绕过，不叫测试。真要做兼容性验证，还得能真实处理弹窗。

- 收口：选框架之前先问一句：要测的流程里，有没有跳出我应用的界面?

> **易错点 / 考点：**三个能力常出简答，关键是跨应用那一条。两个易错：以为 UI Automator 能读应用内部状态，不能，它是黑盒；以为那个查看器是截图工具，不是，它导的是控件树，截图只是附带。另一点：它的测试跑在设备上，MonkeyRunner 跑在电脑上。

> **过渡：**那个查看器用起来什么感觉，下面看界面。

【6.51】UI Automator Viewer（2 分钟）

> 你盯着一屏控件，想知道拿什么去点它。这个查看器就干一件事：把界面结构翻成一张能查的表，让你挑得出定位依据。

>

> 界面上三块信息，正好对上写用例时的三步判断。

>

> 第一块，控件树的层级。它告诉你这个元素挂在哪一层：是列表的子项，还是某个自定义容器的孩子。层级决定你的定位路径要不要往下钻、钻几层。

>

> 第二块，节点的属性表。资源标识、文本、类型、内容描述、是否可用、是否可点击、所在区域，全在这张表里。写定位器就是从这张表里挑一个稳定的属性。优先级：标识优先，然后是内容描述和文本，层级路径再往后，坐标垫底。

>

> 第三块，截图与控件的联动。点一下树上的节点，图上对应区域就高亮。排查"我明明点了它却没反应"时特别有用。你会看到，要点的那个小图标，真正能接收点击的是它外面那层没名字的容器。

>

> 有个实战细节值得说。很多控件点不动，不是定位错了，是它本身不可点击，可点击的是父节点。属性表里"是否可点击"那一栏就是答案，看一眼就明白，比瞎试快得多。

>

> 所以控件点不动，先查它能不能点，再查你点得对不对。三块信息合起来，就是一张能照着走的定位清单。

> **提问：**你在属性表里看到一个节点，文本为空、标识为空、内容描述也为空，只有一块区域信息。这时候怎么办？

> **预设回答：**

- 答"用坐标点"——能跑通，但最脆，换分辨率就废。

- 答"往上找父节点"——好思路。很多自定义控件的父容器有标识，或者整体就可点击。

- 答"让开发加个标识"——最正确的做法。可测性是设计出来的，跟开发提"给这个控件加个标识"，是测试的正当需求，不是求人。

- 收口：这三条也是优先级：先看能不能借父节点，再考虑提需求，才退回坐标。

> **易错点 / 考点：**这一块不直接考，但实验必用。要建立的观念是：定位器是测试代码的接口，接口不稳定，上面盖什么都塌。所以跟开发约定标识命名规范属于测试工程化的一部分，也是答辩时的加分点。

> **过渡：**定位到控件只是第一步，还得能对设备本身做操作。

【6.52】访问设备状态（3 分钟）

手里这台手机，代码里也有一个对象代表它。

>

> UiDevice 就是那个对象。课件说它在运行目标应用的设备上访问和执行操作。先看它能读什么：设备属性，屏幕方向、显示屏尺寸。这两个在写适配性用例时必须先读。为什么？你的滑动距离、点击位置都得按尺寸算，不能写死。写死的那一刻，这条用例就只在你那台机器上有效。换一台平板，它全红。

>

> 再看它能做什么，五条。

>

> 第一条，改变设备的旋转。横竖屏切换最容易暴露布局问题，一行代码就能覆盖，性价比很高。

>

> 第二条，按硬件键，比如音量调高按钮。实体按键这条路径就被盖住了。

>

> 第三条，按返回、主屏幕、菜单按钮。测"连续点返回能不能正确退出"全靠它。这类测试在仪器化路线里很别扭：返回可能让界面销毁，测试进程跟着没了；它在应用外面，不受影响。

>

> 第四条，打开通知栏。通知是很多应用的重要入口，点通知能不能跳到正确页面，是必测项。

>

> 第五条，截取当前窗口的屏幕截图。它截的是整个屏幕，包括状态栏和别的应用，跟应用内截图不是一回事。

>

> 五条合起来说明它的位置：它操作设备，不操作应用。转屏、按键、拉通知栏、截屏，全是系统级动作。

提问：为什么"连续按返回键直到退出应用"这个场景，用 UI Automator 写比用 Robotium 容易？

> **预设回答：**

- 答"因为它在应用外面，应用退出了它还在"——完全正确。仪器化的测试进程和应用同生共死，界面一销毁、进程一退出，测试就断。

- 答"因为它能按系统键"——也对，更根本的原因还是进程边界。

- 追问"那它有什么做不了"——应用内部状态的精确断言。它看不见内存里的对象。

- 收口：又一次回到那条主线，进程边界决定能力边界。

> **易错点 / 考点：**设备对象与元素对象的职责常混。一句话区分：设备对象管设备与全局动作，元素对象管某一个控件。判断题"设备对象可以读取应用内部变量"——错。另一点：它截的是全屏，含状态栏。

> **过渡：**官方后来在这套基础设施之上，又给了一层更适合应用内测试的东西。

【6.53】UI Automator（4 分钟）

> 教材给了一句很高的评价：能写出类似白盒测试那样、更美观的自动化测试脚本。这句话落在 Espresso 身上。

>

> 它正好搭在 UI Automator 那套基础设施之上，专为应用内测试做的一层。三个关键词，一个个拆。

>

> 第一个，类似白盒。它走仪器化同进程路线，能直接拿到界面控件对象，能读属性、能断言状态，不用靠截图或控件树去猜。所以脚本写得像单元测试一样精确。课件还说它可充分利用被测应用实现的程序代码，说的就是这一条。

>

> 第二个，界面线程同步。这是它最有价值的设计。安卓的界面只能在主线程改。测试代码如果在别的线程发操作，就会出现点了但界面还没更新的竞态。传统做法是睡一会儿，或者轮询。它的做法是：框架知道当前有没有待处理的界面任务。动画在跑、消息在排队、后台任务没完，它都算得出来。每个操作和断言之前，它自动等到这些活都干完再动手。所以课件说它解决了可能存在的并发问题、能改进测试的可靠性。不是让你去处理并发，是框架把并发挡在外面。

>

> 第三个，更美观。因为它的 API 是三段式。

>

> 三段式就是课件给的三个组件。ViewMatchers，界面匹配器。它为 View 构建了一套灵活的 API。借助 onView 定位布局里的控件，还支持多层次 View 的定位。这一层管找谁。ViewActions，动作 API。这是一系列扩展的、完成界面交互的方法集。借助 ViewInteraction.perform 完成操作。这一层管对它做什么。ViewAssertions，视图断言。它借助 ViewInteraction.check 判定当前所选 View 的状态。这一层管它现在是不是那个样子。

>

> 串起来就是一句：找到登录按钮，点它，确认它还在屏幕上。

> **提问：**它既然能自动做线程同步，是不是意味着用例里再也不用考虑等待了？

> **预设回答：**

- 答"对，不用等了"——最常见的误解。它等的是它知道的那些界面任务，动画、消息队列、后台任务之类。

- 答"网络请求回来的那条链它未必全知"——对。接口返回之后在主线程更新界面，这个链条它不一定看得到。

- 答"所以有闲置资源登记的机制"——正解。你自己告诉框架，我这儿有个异步任务，没完之前别动手。

- 收口：准确说法是，它把框架知道的并发处理掉了，剩下框架不知道的要你显式登记。

> **易错点 / 考点：**三段式的名称与职责必考，常做连线或填空：匹配器对定位、动作对执行、断言对检查。另一高频点：Espresso 属于仪器化同进程路线，和 Robotium 同类，不是跨进程。判断题"Espresso 可以测试跨应用流程"——错，跨应用要用 UI Automator。

> **过渡：**回到 UI Automator 的 API 本身，核心类是五个。

【6.54】UI Automator API（3 分钟）

> 五个类，一条链，从配置走到一次点击。

>

> UiCollection，容器集合类。它枚举容器里的界面元素，目的是计数，或者按可见文本、内容描述属性去定

> 位子元素。典型场景：这个列表里应当有 20 条商品；找到文本是"已售罄"的那个子项。它是容器视角。

>

> UiObject，元素类。它表示设备上可见的一个界面元素，是所有操作的基本单位。点击、长按、输入、读属性，都发在它身上。它是元素视角。

>

> UiScrollable，可滚动类。它支持在可滚动的界面容器里搜索项目。这一类解决一个很具体的痛：目标在屏幕外，你得先滑到它出现。它给了"滚动直到目标可见""滚到顶部"这类方法。没有它，你就得手写滑一次、找一次、再滑的循环。它是滚动视角。

>

> UiSelector，选择器类。它表示对设备上一个或多个目标元素的查询。查询这个词得留意：它本身不操作、不持有元素。它只是一份筛选条件，比如标识是某个值，且文本包含某个词。元素和集合，都拿它来构造。它是条件视角。

>

> Configurator，配置类。它设置运行测试的关键参数：查找超时时间、是否启用压缩、界面旋转怎么处理。它是配置视角。

>

> 五个串起来就是一次完整操作。配置定规矩，选择器写条件，元素或集合、可滚动容器拿着条件找到目标。找到之后，再对目标执行动作。

>

> 这条链说明一件事：它的 API 是条件驱动的，不是坐标驱动的。

提问：一个商品列表有 50 条，你要点第 38 条上的"加入购物车"。会用到哪几个类？顺序是什么？

> **预设回答：**

- 答"直接用选择器找第 38 条"——找不到。屏幕外的内容不在可见范围里，先得让它出现。

- 答"用可滚动类滚到它出现，再用元素类点"——正解。可滚动类负责让它可见，元素类负责操作它。

- 答"顺便用集合类数一下总数做前置断言"——加分。先断言数量再定位，失败原因会清楚得多。

- 收口：这条链说明它的思维定式，先让它可见，再谈操作。

> **易错点 / 考点：**五个类的职责常考连线。最易混两组：选择器与元素，前者是查询条件，后者是找到的元素；集合与可滚动，前者是容器内枚举计数，后者是滚动搜索。口诀：配置定规矩，选择器提条件，可滚动找得到，集合数个够，元素点下去。

> **过渡：**五个类怎么落到代码里，下面看例子。

【6.55】UI Automator 代码示例（2 分钟）

> 代码摊开，五个类都能在上一页找到出处。

>

> 开头拿设备对象：通过仪器化实例取到设备。这里还能拿到仪器化实例，说明它的测试同样是一个仪器化测试包。只是操作界面走的是控件树通道，不是应用内部对象。很多人把这两件事搞混：安装方式和操作方式是两回事。

>

> 然后启动被测应用：通过上下文拿包管理器，取出启动意图，再启动。为什么不拿设备对象去启动？设备对象面向整台设备，启动某个具体应用走系统接口更直接。

>

> 接着是定位与操作，这段是重点。拿选择器写条件，比如按资源标识；用找到对象拿到元素，然后点击。元素在滚动区域里，就先构造一个可滚动容器，再按文本找子项，允许它边滚边找。

>

> 断言部分：要么断言元素存在，要么读文本比对。

>

> 收尾：按返回或回主屏，把现场清干净，保证用例之间互不污染。这一步最容易被省略，可它是用例能独立跑的前提。

>

> 整段看下来一个特点：它不关心应用内部是怎么实现的。代码里没有一句去断言内部的字段，全部落在界面上，只问有没有这个东西、点了变成什么样。

>

> 这就是黑盒的写法。收尾那一步把现场清了，下一条用例才敢从干净的状态开始跑。

> **提问：**代码开头拿仪器化实例这一句，说明它的测试包也走仪器化的安装流程。那它为什么不干脆用同进程的能力？

> **预设回答：**

- 答"是走同一个安装流程"——对。测试包还是仪器化测试包，要装，要登记目标包名。

- 答"但它不用同进程能力，因为它要跨应用"——这是关键答案。同进程只能看自己，跨应用必须走控件树通道。

- 答"所以它是借了仪器化的壳，用了无障碍的芯"——这个说法准确，也好记。

- 收口：框架的安装方式和操作方式是两件事，别因为都叫仪器化，就以为能力一样。

> **易错点 / 考点：**代码题常问，给一段这样的代码，它能不能测跨应用，为什么。答题要点：测试包形式是仪器化的，操作走设备与元素对象的控件树通道，所以跨应用可行，也不受被测应用进程崩溃影响。另一考点：用例末尾的清理动作属于用例独立性要求，缺了会被扣分。

> **过渡：**界面自动化这一大块讲完。换个问法：代码改完之后，怎么保证原来的功能没坏？

【6.56】6.3 回归测试（2 分钟）

> 自动化讲了这么多，得回答一个更朴素的问题：为什么要一遍遍重跑？

>

> 手工点一遍也就十分钟，写脚本要两周。这笔投入天天有人算，答案就藏在 6.3 回归测试这一节里。

>

> 一次性的验证不需要自动化，反复执行的验证才需要。自动化不为了让手工活变高级，是让重复的那部分不再靠人记。而这世上绝大多数测试都是反复执行的：每一次版本迭代，老功能都要再验一遍。

>

> 这一节解决三个层次的问题。第一，回归缺陷到底是什么，为什么它必然发生。这是定义与成因。第二，为什么必须做回归，不做会怎样。这是代价。第三，怎么做才做得起。这是范围与策略，也是难点，因为全部重测在稍大的系统上根本跑不完。

>

> 它还会跟下一节连起来。策略决定测哪些，精准测试回答凭什么这么选。一句话，回归测试选范围，精准测试给证据。

>

> 还有个前提得说清。这里的回归，不是把功能再测一遍这么简单。它有一套明确的对象，回归缺陷；明确的代价，带着缺陷发布；明确的方法，四种策略。

>

> 学完这节，评审会上两个问题你得答得上：这个版本为什么必须做回归；为什么回归只做这些，不是全部。答不上来，回归就只能靠加班顶。

提问：既然回归测试跑的是现成用例、不用重新设计，为什么它反而是自动化投入产出最高的一块？

预设回答：

- 答"因为它省钱"——太笼统。省的是哪一笔？省的是每次版本迭代都要重做一遍的操作。

- 答"因为用例是现成的，脚本只写一次"——对。新功能测试要现设计用例，回归测试的输入已经存在，自动化只要把它变成可重复执行的。

- 答"因为人做回归会偷懒"——这句很真实。人做重复劳动到第三轮一定开始跳步，机器不会。

- 收口：所以回归测试是自动化的主战场，界面自动化只是它的手段之一。

易错点 / 考点：本节结构常出填空：回归测试含定义、成因、必要性、策略；精准测试含三基础、双向映射。易错：把回归测试理解成把所有测试重跑一遍，那只是四种策略里最贵的一种，不是定义。

过渡：从一个最朴素的场景开始。

【6.57】为什么需要进行回归测试？（3 分钟）

> 去年这个功能还是好的，这个版本一上线，它莫名其妙就坏了。

>

> 课件给它起了名字：回归缺陷，Regression bug。拆开看，三个特征，缺一不可。

>

> 第一，它曾经是好的。上一版测过、上过线、用户在用，功能是通的。所以它不是新功能没做好，是老功能被弄坏了。

>

> 第二，需求没有变。没有人要求改它，需求文档一个字没动。

>

> 第三，它是被别的改动带坏的。改 A 把 B 弄坏了；修一个缺陷，引出两个新缺陷。

>

> 所以这个词说的不是功能坏了，是本来好的被弄坏了。

>

> 为什么非要强调这三条？因为它决定责任归属和防范手段。新功能有缺陷是没做到；回归缺陷是做到了又丢了。后者对用户的伤害更大，因为用户已经建立了预期。

>

> 打个生活里的比方。一家餐厅的红烧肉是招牌菜。某天换了厨师，菜谱一个字没改，可新厨师把调料罐的位置挪了，糖放成了盐。菜谱没变、客人要求没变，就是别的环节动了，招牌菜毁了。客人不会说你们新菜不行，客人会说你们家不行了。

>

> 软件里这个逻辑更狠。用户对已经能用的功能，基本上没有容忍度。新功能崩了，用户会想再看看；天天用的功能突然不能用了，当场卸载。

> **提问：**一个功能上线三年没出过问题，这次版本因为它挂了。算不算回归缺陷？判断依据是什么？

> **预设回答：**

- 答"算，因为它以前是好的"——只满足一条，不够。还要看需求有没有变、是不是被别的改动带坏的。

- 答"要看这次有没有人动过相关代码"——对，这是第三条。如果没人动它却坏了，往往不是回归缺陷，而是环境、数据或依赖变了，这种更难查。

- 答"如果需求变了呢"——那就不叫回归缺陷，叫需求变更引起的预期变化，测的是新需求，不是旧行为。

- 收口：判定走三步：先确认旧行为当时是对的，再确认需求没动，再看是不是被别的改动牵连。

> **易错点 / 考点：**定义必背，三个条件一个都不能少。选择题常给四五个场景让你判断，最容易错的是需求变了的那个，那不是回归，是变更。口诀：旧的好、需求没动、被别人带坏。

> **过渡：**定义清楚了，下一个问题就来了。

【6.58】为什么要有回归测试？（3 分钟）

> 有人说，改一处小地方，为什么非得再测一轮？

>

> 第一层，波及面随系统规模非线性增长。一个系统有一百个功能点，改一处，理论上所有和它有数据或调用关系的功能都可能受影响。系统越大、耦合越紧，一次改动能碰到的地方就越多。这不是可能出问题，是必然有一些地方出问题，只是你不知道在哪。

>

> 第二层，回归缺陷最难被发现，因为它的藏身之处就是"已经验过"这份信任。新功能大家会重点测，老功能默认是好的。测试资源永远优先给新东西，于是老功能的验证被压成点一下看看。回归缺陷就活在这个缝隙里。

>

> 第三层，它的线上代价最高。新功能出问题，影响的是想用新功能的人；老功能出问题，影响的是每天都要用的人。老功能往往在业务主链路上，登录、下单、支付、查询，一挂就是直接的经济损失和口碑损失。

>

> 课件那句话说得极重：如果没有回归测试，产品就带着回归缺陷被发布出去了，造成严重后果。它用的是"就"，意思是这不是概率事件，是必然事件。

>

> 所以回归测试的位置不是再多测一遍，它是一道闸门。每次改动都要过这道闸，过了才能发布。闸不开，代码就出不去，这一版就压着不发。

> **提问：**团队说"人手不够，这个版本就不做回归了，反正只改了一个小地方"。你会怎么回应？要给一个有说服力的理由，不能只说"不行"。

> **预设回答：**

- 答"只改一处也会影响别处"——方向对，但太笼统，容易被"我改的是独立模块"顶回来。

- 答"那至少把和它有关联的模块测一遍"——这是可执行的让步，也就是基于风险选择范围。

- 答"上个版本就是因为小改动出的问题"——用历史事实说话最有力。这也是团队要记录回归缺陷分布的原因，它是最好的说服材料。

- 收口：跟产品、开发谈范围，靠的不是"我觉得要测"，是这里出过问题、这里耦合紧、这里在支付链路上。

> **易错点 / 考点：**考点是回归测试的必要性理由，答三条即可：波及面、隐蔽性、代价。另外要能说出回归测试是发布闸门这个定位，不是重复劳动。这两句的区别，是理解与否的分界。

> **过渡：**定义和理由都清楚了，难题来了：怎么做。

【6.59】回归测试（4 分钟）

> 回归这两个字，容易被读成回头再测一遍。课件给的是四句话。

>

> 第一句，一旦程序某些区域被修改了，就可能影响原来正常工作的区域。受影响的区域于是出现回归缺陷。这句讲机制：修改是有涟漪的。涟漪靠什么传？调用关系、数据依赖、全局状态、配置、共享的库。所以影响范围不是猜出来的，是可以分析的。这句话为下一节的精准测试埋了伏笔。

>

> 第二句，回归缺陷：原来正常工作的功能，没有发生需求变化，而由于受别的改动影响产生的问题。这是定义，和上一页那三条对得上。

>

> 第三句，回归测试就是为了发现回归缺陷而做的测试。目的单一：找退步，不找缺失。

>

> 第四句，如果没有回归测试，产品就带着回归缺陷被发布出去了，造成严重后果。这是代价。英文名是Regression Testing。

>

> 合起来记：在软件被修改之后，重新执行已有的测试，确认修改没有破坏原有功能。三个关键词：修改之后、重新执行、已有测试。第三个要划重点，回归测试原则上不设计新用例，它复用现成的。这也正是它天然适合自动化的原因：用例是现成的，只是要反复跑。

>

> 再说一个反直觉的点。正常的回归测试，失败率应当很低。跑一千条，可能只有三五条红。红了一大片，要么这次改动太大，要么用例本身不稳定。所以回归的通过率本身就是一个质量指标。

> **提问：**回归测试原则上不设计新用例。那如果这次改动引入了一条"从没测过的新路径"，回归测试能不能发现它？

> **预设回答：**

- 答"不能，回归只跑老用例"——对，这就是它的能力边界。老用例没覆盖到的路径，它看不见。

- 答"那要靠新功能测试补充"——对。一个版本发布前要做两件事：新功能验证加老功能回归，两者不能互相替代。

- 追问"怎么知道新路径有没有牵连老路径"——靠代码依赖分析，也就是下一节的内容。

- 收口：回归只保证已覆盖的部分没退步，不保证没覆盖的部分没问题。这也解释了覆盖率为什么重要。

> **易错点 / 考点：**定义必背，三个关键词是答题骨架：修改之后、重新执行、已有用例。判断题"回归测试需要重新设计全部用例"——错。另一高频点：回归测试与新功能测试的目的区别，找退步对找缺失。

> **【思政落点 7】**：回归测试守的是一个承诺：你昨天能用的功能，今天还能用。用户不会因为我们加了十个新

功能，就原谅老功能坏掉。那套能用的行为，是我们已经许出去的话。工程上的事故常常不是新东西没做，是把旧东西弄坏了还发出去。所以做回归不是保守，是守信；对自己交付过的东西负责，比对新东西负责更难，也更见人。

过渡：实操上的两个问题立刻浮出来。

【6.60】如何进行回归测试？（3 分钟）

只剩两个词：范围，策略。这两个词就是回归测试全部的实操难点。

>

先说范围。这次改动之后，到底要重测哪些东西？范围定小了，漏掉的区域带着缺陷上线；定大了，测试跑不完、项目延期，结果还是没测完就发。所以范围问题的实质是：在有限的时间和人力下，怎么把风险最高的那部分圈进来。

>

再说策略。范围定了，用什么方法去选？全部重跑，还是抽一部分？抽的话按什么抽，按风险，按用户实际使用频率，还是按代码改动的位置？

>

这两个问题不是并列的，是一条决策链：先有策略，才有范围。策略回答凭什么选，范围是策略的输出。

>

举个具体的。一个电商系统这个版本改了优惠券计算。范围问题：只测优惠券，还是连订单金额、发票、退款一起测？策略不同，答案不同。按风险，优惠券和订单金额都得测，因为跟钱有关。按操作剖面，看用户实际最常走哪条路径，如果大多数人只用满减，满减优先。按改动的位置，就顺着优惠券的调用链往外扩一层。

>

三种策略给出三个不同的范围，成本差好几倍。所以这不是哪个对的问题，是在这个项目、这个版本、这个风险水平下，哪个合适的问题。

>

> 改动小不是缩小范围的理由，调用链短才是。

> **提问：**产品经理说"这次改动很小，回归只测改动的那个页面就行"。你同意吗？为什么？

> **预设回答：**

- 答"同意，改动小影响小"——危险。改动大小和影响范围不成正比。改一行公共工具方法，可能影响几十个页面。

- 答"不同意，要看它被谁调用了"——正解。判断依据是依赖关系，不是改了多少行。

- 答"折中，按调用链测一层"——很务实。先测直接调用方，再看有没有跨模块的共享状态。

- 收口：改动小不是缩小范围的理由，调用链短才是。

> **易错点 / 考点：**这一块本身不考具体知识，它是策略页的引子。要建立的核心思维：回归范围是风险决策，不是工作量妥协。答题时能说出范围与成本是一对矛盾、策略用来平衡它们，就已经到位。

> **过渡：**到底有哪几种策略，下面给四种。

【6.61】回归测试策略（4 分钟）

> 版本要发，回归得跑。真正让人头疼的不是跑不跑，是跑多少。用例从几百条涨到几千条，全量跑一遍要两天，你手上只剩半天。

>

> 课件给四种策略。第一种，再测试全部用例。把老用例一条不落地重跑。成本最高，风险最低，理论上不留漏网。什么时候值得？安全关键系统，漏一个的代价远大于多跑几天。或者临近发布、时间还够的收尾阶段。它难长期坚持，因为用例会膨胀，全量从几小时变成几天。

>

> 第二种，基于风险选择测试。按出问题的概率乘后果排序，先测高风险的。哪儿算高风险？这次改动碰到的、历史上缺陷扎堆的、在主链路上的、复杂度高的、好几个人同时改过的。成本中等，工业界最常用。难在风险怎么量化，不能凭感觉，要拿历史缺陷数、变更频率、复杂度当依据。

>

> 第三种，基于操作剖面选择测试。按用户实际使用的频率分布来挑，用得最多的路径先测。为什么这么干？用户绝大多数时间就走那几条路，那几条坏了才是真事故。成本低，性价比高。盲区也清楚：冷门路径几乎不测，坏了一直没人发现，等到某个客户投诉。

>

> 第四种，再测试修改的部分。只测这次改动直接影响到的。成本最低，风险最高。为什么最高？影响是往外传的，你只测了圆心。它适合改动很局部、依赖清楚、调用方又有自动化能快速验的情况。

>

> 课件末尾留了个省略号。意思是策略不止这四种，还有基于影响分析的、基于变更覆盖的。

>

> 那实际怎么用？分层。改动点用第四种快验。高风险区和主链路用第二种补齐。发布前用第三种把高频路径走一遍。关键模块或者大版本，才上全量。四种不是四选一，是四层。

> **提问：**一个日活百万的购物应用，这次只改了下单页一个按钮的文案。你选哪种策略？说清理由，再指出你的方案可能漏掉什么。

> **预设回答：**

- 答"再测试修改的部分"——合理，改动确实局部。但只做这一项，就漏了"下单链路整体还通不通"。

- 答"基于操作剖面，把下单主路径跑一遍"——更好。文案虽小，下单是最高频的主链路，必须验。

- 答"再加基于风险：支付接口有没有被牵连"——加分。改文案本来不该碰支付，可这次顺手动了公共组件，风险就在那儿。

- 收口：答案是组合——改动点直接验，下单主链路冒烟，支付接口快速回归。可能漏的是冷门路径和兼容性场景，比如老版本客户端。

> **易错点 / 考点：**四种策略的名称、成本与风险对照必背。最易混的是**基于风险**与**基于操作剖面**：前者看"坏了多严重"，后者看"用得多不多"，两者正交、可以叠加。另一处易错：以为"再测试修改的部分"是最优解——它成本最低，不是风险最低。口诀：全量最稳最贵，风险看后果，剖面看频率，改哪测哪最省也最险。

> **过渡：**这四种策略有个共同前提——你得知道改动到底影响了谁。靠猜不行，得有证据。

【6.62】6.4 精准测试（2 分钟）

> 刚才那个前提，卡住过很多人：凭什么说这几条必须跑，那几条可以不跑？

>

> 精准测试回答的就是这一问。它不是一种新的测试方法，是一套让回归范围有据可依的技术体系。要解决的矛盾还是那一个：范围大了跑不完，范围小了漏缺陷。过去靠人的经验去判断"改这儿会牵连那儿"，它要把这个判断换成能算、能追溯的证据。

>

> 基本思路一句话。先把测试用例和代码建立映射，再分析代码之间的依赖。于是"代码改了什么"，就能反推出"哪些用例必须重跑"。

>

> 为什么这事过去做不到？因为改动的记录散在聊天记录、提交说明和某个人的脑子里。现在工具能读代码、能读执行轨迹、能建索引，判断才有地方落脚。

>

> 为什么这门课要单开一节？因为一天发几次版本已经不稀奇，发布窗口常常只留一两个小时，三千条用例全跑谁也等不起。每次发布都靠人拍脑袋定范围，发布频率就被人的判断速度卡死了。

>

> 边界也要说清。它管选得准，不管测得好。用例本身写得烂，选得再准也没用，它也不会替你把用例补上。

>

> 后面的名词听着唬人：覆盖率、映射、依赖图。它们合起来只回答一句话——我改了这里，哪些老用例必须再跑，凭什么。

> **提问：**精准测试要工具采覆盖率、建映射、算依赖，那它是不是只适合大公司、大系统，小项目用不上？

> **预设回答：**

- 答"是，小项目人少改动少，凭经验就够"——前半句成立，后半句危险。改动少不等于没影响，只等于"人还记得住"。

- 答"小项目可以用轻量的办法"——正解。轻到什么程度？把改了哪个模块、要跑哪几条用例记成一张表，人工维护，这就是精准测试的雏形。

- 答"等出过事故就知道要用了"——很现实，代价太大。

- 收口：所以它的核心不是工具，是"改动与用例的对应关系有没有被显式记下来"。工具只把这件记录自动化。

> **易错点 / 考点：**本节三基础（覆盖率分析、用例与代码映射、依赖分析）与精准回归的关系必背。最易混：把精准测试当成一种"测试方法"——它是一套**范围决策技术**，本身不产生新的用例设计方法。

> **过渡：**三个基础，一个一个来。

【6.63】精准测试（3 分钟）

先说个常见场面。团队买了工具，导出一份覆盖率报告，翻两页就放下了，因为不知道这份报告该喂给谁。

>

> 三个基础就是这条流水线的三段。第一段，测试覆盖率分析。它回答：现有用例到底走到了哪些代码。方向是从用例到代码。手段有插桩、字节码分析、运行时探针。结果是每一行、每个分支被哪些用例覆盖过。少了这一段，你既不知道自己的用例网有多密，也不知道哪块代码在裸奔。

>

> 第二段，代码和测试用例之间的关系。覆盖率数据天然带着这份对应：某条用例跑的时候经过了哪些类、哪些方法。把它反过来建索引，就得到双向映射。这一段为什么算核心？回归要问的是"我改了这段代码，哪些用例必须跑"，正好是代码到用例那个方向的查询。

>

> 第三段，代码依赖性分析。它回答：改了 A，还可能牵连谁。依赖不止调用一种，还有继承与实现、数据库表和字段、配置和开关、消息主题、前端组件引用。分析完得到一张依赖图。

>

> 三段串起来是一条链。改动代码，依赖分析往外扩出影响范围。拿代码到用例的映射，查出覆盖这些代码的用例。剩下的就是必须重跑的最小回归集。课件底部那句"基于上述工作，可以精准地完成回归测试"，说的就是这条链。

>

> 它要同时满足两条才叫精准。一是不漏，被影响的代码一定有对应用例被选中；二是不多，影响范围外的用例不跑，时间才省得下来。缺一条，另一条就没意义。

提问：要是团队用例很少、覆盖率只有三成，精准测试还能不能起到"精准"的作用？

预设回答：

- 答"能，选出来的用例更少、更省时间"——表面如此，实际上这是最危险的情形。

- 答"不能，七成代码没有用例覆盖，改到那些地方时映射查不到任何用例"——这是关键答案。它会诚实告诉你"这里没有用例"，系统不会因此变安全。

- 追问"那怎么办"：先把覆盖缺口补上，或者把无覆盖区降级为人工评估。它的结论必须配合覆盖率一起读。

- 收口：所以前提是用例网够密。网破的地方它救不了你，只会如实说网破了。

易错点 / 考点：三个基础的名称与作用必背，常出简答。最易混：把覆盖率分析与依赖分析搞反——前者看"测到了没有"，后者看"改了牵连谁"。另一考点：双向映射里回归用的是"代码到用例"这个方向，常被忽略。

过渡：映射为什么必须双向？下一页专门说这件事。

【6.64】测试用例与代码映射关系（3 分钟）

两个箭头，方向相反。写反了，工具买回来也使不上劲。

>

> 先看用例到代码。方向是：跑这条用例会执行哪些代码。用途有三个。评质量，一条用例只碰三行，价值有限；一条跨五个类、走上两百行，那是主干。找冗余，两条用例覆盖的代码完全一样，大概率重复，可以合并。找缺口，有些代码从来没被任何用例走过，那就是裸奔区。

>

> 再看代码到用例。方向是：这段代码被哪些用例盖着。主用途是回归选型——这次改了这里，把这批用例挑出来重跑，就是最小回归集。它还帮缺陷定位：某个方法报错，马上知道哪几条用例该红。

>

> 为什么非做双向？两个方向回答的是两个不同问题，而且推不出来。你有完整的正向数据，理论上能倒排一份索引。可工程上这两份数据常来自不同通道。正向靠运行时插桩，反向靠静态分析加索引。更新节奏也不一样，用例天天在加，索引得跟着重建。

>

> 再深一层：粒度决定精准度。细到方法级、行级，准，但数据量大、开销大。粗到类级、文件级，省，但会把回归范围放大——改一行，整个类都被判成受影响。工业界的取舍通常是：核心模块做到方法级，外围模块做到类级。

>

> 所以这张映射表不是填给老师看的文档，它是"这段代码没问题"这句话的凭据。说没问题，得拿得出谁跑过它。

> **【思政落点 8】**：一张映射表，本质是把"我觉得"换成"数据显示"。工程上最容易出事的地方，恰恰是那些自我说服：我觉得这个小改动不影响别的模块，我觉得这个功能上次测过了。测试这个职业最值钱的习惯，就是任何判断都要求一份能摆出来的证据。愿同学们以后写报告、做答辩、给结论，都先问自己一句：这话的凭据在哪。

> **提问：**团队想做精准测试，但只采了"用例到代码"一个方向的数据。他们能不能算出"改了某个方法要重跑哪些用例"？

> **预设回答：**

- 答"能，反查一下就行"——逻辑上对，工程上不一定。要看粒度：只记了这条用例跑过这个类，最多反查到类级，到不了方法级。

- 答"不能，得重建倒排索引，而且数据要及时更新"——对。新加的用例不在索引里，反查就是空的。

- 答"那就两个方向都采"——这正是工业界的做法：正向采集加反向索引，两份数据互相校验。

- 收口：所以双向不是重复劳动，是两套用途不同的数据资产。

> **易错点 / 考点：**核心考点是两个方向与用途的配对：用例→代码＝评估用例质量、找冗余、找缺口；代码→用例＝回归选型、缺陷定位。易错：以为两者只是"正反说法"——采集方式、更新频率、使用场景都不同。

> **过渡：**映射解决了谁盖住谁，可"改了 A 会不会牵连 B"，还得看依赖。

【6.65】代码依赖性分析（3 分钟）

> 一个字段改类型，你猜要连带看多少处？猜不出来，这就是依赖分析要解决的事。

>

> 先看边有哪几种，这决定你的图要画多宽。调用依赖最直接：A 调 B，改了 B，A 的行为可能变。继承与实现依赖最容易被漏。改父类一个方法，所有子类都受影响。你改的地方和出事的地方，在代码里隔得很远。数据依赖：共用一张表、一个字段、一个缓存键，改类型，所有读它的地方都危险。配置与资源依赖：共用一个开关，改个默认值，影响面横跨几个模块。消息与事件依赖最隐蔽。发布订阅关系，改消息体结构，所有消费者都中招。代码里没有一处直接调用，它是运行时连起来的。

>

> 再看怎么分析，两条路。静态分析解析代码，把调用和继承关系抽出来建图。好处是全量、快、不用跑。坏处有盲区：反射、依赖注入、动态代理、脚本化配置，它往往看不出来。动态分析跑起来记真实轨迹。好处是准，坏处是只覆盖跑到的那几条路。工业界合起来用：静态给骨架，动态修正补充。

>

> 图建好了怎么用？从改动点出发，沿依赖边一层层往外走，这就是传递闭包。一层调用，两层，三层，越远越淡。实践中会设一个扩散层数上限，再往外就当影响可忽略。这个层数，就是精准度和成本之间那个旋钮。

提问：一个项目大量使用依赖注入和反射，静态分析出来的依赖图很稀疏。这会让精准测试出什么问题？

预设回答：

- 答"会漏掉影响范围"——对。图稀疏意味着"改了 A 却查不到 B"，B 不会被选进回归集，这是最危险的情形。

- 答"所以要用动态分析补"——正解。跑一遍集成测试，把真实调用链记下来补进图里。

- 答"还有一种办法：把注入关系写成配置，让静态分析能读"——工程上可行，属于提高可分析性的设计改进。

- 收口：所以"代码可分析性"本身就是一种设计质量。框架用得越动态，工具越难帮你。

易错点 / 考点：五种依赖类型常考简答；静态与动态分析的优缺点对照是高频选择题——**静态＝全量但不准，动态＝准但不全**，别背反。另一考点：扩散要设层数上限，这是精准度与成本的平衡点。

过渡：三个基础里还剩一个，覆盖率，它是所有证据的源头。

【6.66】测试覆盖率分析（3 分钟）

工具给了一句很干脆的结论：本次改动影响 3 个方法，需要重跑的用例，0 条。你信不信？

>

> 先看它测什么。覆盖率有粗细几档：行覆盖，这行执行过没有，最粗也最好做；分支或判定覆盖，判定的真假两条路走过没有；再细是条件覆盖、路径覆盖。建映射这件事上，最常用的是行覆盖和方法覆盖。为什么不用最细的？粒度太细，数据量会爆炸。

>

> 再看怎么采。主流是插桩：编译期或运行期往代码里放记录点，跑测试时记下这个点被谁走过。Java 生态多是字节码插桩，前端多是构建期注入统计代码。插桩有性能开销，还会污染产物，所以生产环境一般不插，只在测试环境采。另一条路是运行时采样分析：按时间间隔采调用栈，反推哪些代码被执行过。开销小，但短小的调用会漏掉。

>

> 它在精准测试里干两件事。一是建映射，覆盖率数据本身就是用例到代码的原始记录，按用例切开，表就有了。二是给回归结论定可信度，这一步最容易被忽略。改动落在零覆盖区，选出来的回归集就是空集，工具会说没有用例需要跑。这个结论得当成高风险看，不能当成没问题。

>

> 为什么？因为"改动影响小"和"这段代码没人测过"，在报告上长得一模一样。区别只能靠覆盖率数据认出来。所以两样要一起读。

提问：精准测试工具报"本次改动影响 3 个方法，需要重跑的用例 0 条"。这个结论你怎么处理？

> **预设回答：**

- 答"直接发布，没用例要跑说明影响很小"——最危险的处理方式。

- 答"先看这 3 个方法的覆盖率"——正解。要是零覆盖，那就不是没影响，是没人测过。

- 答"还要看改动本身是什么性质"——加分。改注释、改日志格式和改金额计算，风险完全不同。

- 收口：工具给的是数据，判断还得人做。自动化能算出范围，算不出该不该担心。

易错点 / 考点：覆盖率的粒度与采集方式常考。最核心的判断是：**覆盖率低时，精准回归的结论不可信**。另一

处易错：把覆盖率当成质量分数——它是测量，不是评价；跑过不等于断言过。

过渡：6.1 到 6.4 走完了，把这一讲收一下。

【6.67】本讲小结（3 分钟）

五条。别当五条孤立的知识点背，它们是一条线。

>

第一条，基于需求和启发式测试策略做功能测试需求分析。起点在这儿：不是拿到需求就写用例，而是先把测什么、为什么测、测到什么程度想清楚。方法是分解需求，配上启发式策略，按功能、场景、数据、状态、时序去切。

>

第二条，先明确测试目标，再做测试范围、测试项和风险的分析。顺序别倒，目标在前。目标不清，范围一定失控。内容是功能、场景、数据三个维度。功能是有哪些能力，场景是用户在什么情况下用，数据是什么样的输入会出问题。

>

第三条，探索式测试的四要素、IAPE、生态建设以及 SBTM。这部分讲人在不确定的情况下怎么高效找缺陷。四要素是框架，IAPE 是执行模型。SBTM 是基于会话的管理，把探索式测试变得能计划、能度量、能汇报。它的位置很清楚：不是所有测试都能提前写成脚本，它管脚本之外那一半。

>

第四条，回归测试策略。全部重测、基于风险、基于操作剖面、再测试修改的部分。核心是成本与风险的取舍，以及实践中怎么分层组合。

>

第五条，精准测试。覆盖率分析、用例与代码的映射、依赖分析三个基础撑起精准回归，把凭经验选范围换成凭证据选范围。

>

一条线串下来：先分析需求定目标，再用方法找缺陷，改动之后守住已经拿到的结果。前半段回答怎么找问

> 题，后半段回答怎么不把修好的东西再弄坏。

> **提问：**这五条里，只能给团队留一条"最能减少线上事故"的，你选哪条？

> **预设回答：**

- 答"选回归测试"——很有说服力。绝大多数线上事故不是新功能没做好，是老功能被改坏了。

- 答"选测试需求分析"——也说得通。缺陷在需求阶段埋下，后面全是在错误的靶子上使劲。

- 答"选精准测试"——前提是你已经有回归体系，它是在优化，不是在建。

- 收口：三个答案分别对应防退步、防错方向、防跑不完。连回归都没有，先做回归；回归有了却跑不完，才轮到精准测试。

> **易错点 / 考点：**小结页常出"本章包含哪几部分"的结构题。要能一句话说出 6.1 到 6.4 各自的主题：功能测试方法与过程、功能测试自动化、回归测试、精准测试。别把自动化和精准测试混成一类——前者是执行手段，后者是范围决策。

> **过渡：**留三个问题，没有标准答案，但值得认真想。

【6.68】提问与思考（4 分钟）

> 三个问题，都不给标准答案。给的是往下查的抓手。

>

> 第一个，录制脚本那么轻松，为什么没成主流？抓手在这儿：录制器记的是操作序列，不是测试意图。它记下"在某个位置点了一下"，不知道"我在验登录成功"。界面一调，脚本就失效，失效原因还说不清。更麻烦的是它没有断言，能回放操作，不能判断对错。而脚本的大头工作量，恰恰在断言和定位策略上，这部分录制器一点忙都帮不上。所以录制适合做原型和冒烟，不适合当长期资产。团队说"我们用录制"，成本并没消失，只是被推到后面去了。

>

> 第二个，开源框架还是自研？先问你的差异化在哪。定位控件、点击、断言，这些都是通用能力。开源早做完了，自研等于重造一遍，还得吃下所有兼容性成本。自研只在两种情况下值：一是开源搞不定的特殊对象，比如自绘引擎、特殊协议、嵌入式界面；二是你要做平台，多项目共享、可视化编排、统一报告，这时自研

> 的是平台层，底层照样用开源。所以不是二选一，是分层。自动化都没跑顺就上自研，通常结局是造出一个只有自己能用的半成品。

>

> 第三个，都重视自动化，效果却不好，问题在哪？四个方向查。目标错了，拿用例数量、脚本行数当成绩，不拿发现多少缺陷、省多少人工当成绩。维护失控，定位器太脆，界面一改一片红，最后没人敢跑。只做界面层，界面最贵最慢，真正该自动化的是接口层和单元层。没进流程，脚本跑在某个人电脑上，没接流水线，红了三天没人管。四条里最致命的是最后一条：没人看的自动化，等于没有自动化。

> **【思政落点 9】**：第三个问题多说一句。面对效果不好，最容易的反应是找外部理由：工具不行、时间不够。可工程上的进步，几乎都始于一句诚实的自我追问——我当初做这件事，想解决什么？答案是"因为别人都在做"，效果不好就是必然。技术判断上的诚实比技术本身难。愿你们以后带项目，敢在复盘会上说我这项做错了，别包装成阶段性成果。

> **提问：**你们自己或实习时见过的自动化测试项目，最后是活着还是死了？如果死了，死因更接近哪一条？

> **预设回答：**

- 答"没见过"——也正常，那就拿这门课的实验当第一个样本，观察三个月后它还能不能跑。

- 答"死了，因为界面老变"——对应第二条，维护失控，这也是最常见的死因。

- 答"活着但没人看"——这是假活，对应第四条。指标上它每天在跑，实际没人因为它的结果改变行为。

- 收口：判断一个自动化项目活没活，别看它跑不跑，看它红了以后有没有人停下来。没人停，就是死的。

> **易错点 / 考点：**三问是开放题，闭卷一般不直接考，答辩常被追问。答这类题要有立场、有理由、有边界——比如"我选自研，因为对象是自绘控件，开源定位不到，但底层驱动还用开源"。只会说"开源好"或"自研好"，是没答到点上。

> **过渡：**说了这么多，该动手了。实验 3 把这一讲的东西全用一遍。

【6.69】实验 3: 针对某 web 应用系统进行功能测试（3 分钟）

> 这次要交的是能跑的脚本，不是几张截图。题目是：针对某 web 应用系统做功能测试。被测系统用课件给的那个公开电商演示站 saucedemo，登录、商品列表、排序、购物车、结算、下单都有，它本身就是给学习和

测试准备的公开靶场。这一条也别忽略：练手要用授权的对象，别拿真实业务系统开刀。

>

要求三条，我逐条说清做到什么程度算过关。

>

第一条，基于本模块学过的方法做测试需求分析，按优先级列出主要测试项，识别出测试风险。三个关键词：分析、优先级、风险。合格产出是一张表——测试项、依据来自哪条需求、优先级、风险。比如结算流程依赖第三方支付沙箱，可能不可用。不合格的产出是一句话"我们要测登录、购物车、下单"，那不是分析，那是列标题。

>

第二条，设计 20 个以上相对关键的用例，能发现更多缺陷，还要说明设计思路和方法。关键词是关键和思路。不是把用户名空、密码空、用户名超长这类凑数用例写满。要覆盖不同设计方法：等价类、边界值、判定表、场景法、错误猜测。每条用例标出用的是哪种方法、为什么这么设计。这一条是评分重点，考的是你会不会设计，不是会不会点。

>

第三条，基于 Selenium 加 WebDriver，把之前设计的 10 条用例转成脚本。Java 格式，在 JUnit 上执行和调试。是转化 10 条，不是新写，所以用例的可自动化程度会影响你挑哪 10 条。脚本要有显式等待，不是死等；定位器要稳，优先用标识；断言要明确；用例之间互不依赖。JUnit 里用准备和清理方法管浏览器生命周期，用测试注解放用例。

提问：你从 20 条用例里挑 10 条去做自动化，挑的标准是什么？

预设回答：

- 答"挑最好写的"——大家都会这么想。只看好写，自动化最后覆盖的都是简单场景，价值不大。

- 答"挑会反复跑的、结果明确的"——正解。回归价值高的、执行频率高的、判定明确的优先。

- 答"挑最容易出缺陷的"——方向也好，但要权衡稳定性，探索性强的场景手工更合适。

- 收口：选型记三条——高频、稳定、可判定。三条都满足的优先，只满足一条的先放放。

易错点 / 考点：实验常见扣分点：用死等代替显式等待；定位器写成绝对路径，一改就废；用例之间有依赖，

A 靠 B 创建的订单；只有操作没有断言；交上来的 10 条和那 20 条对不上号。这些不是技术难度问题，是工程习惯问题，也正是这门课要练的。

过渡：想再往深学的，下面几本书可以挑。

【6.70】学习资源推荐（2 分钟）

四本，按补哪块短板挑，别按顺序买。

>

第一本，《接口自动化测试项目实战》，清华大学出版社，2021 年 11 月。补的是接口层自动化。前面反复说过，界面自动化最贵最慢，性价比最高的是接口层。项目还没做接口自动化，从这本开始。

>

第二本，《从零开始学 Selenium 自动化测试》，机械工业出版社，2020 年。补 Web 界面的基本功：元素定位、等待策略、页面对象、测试框架集成。这本和 6.2 讲的内容对得上，实验 3 会直接用到。

>

第三本，《前端自动化测试框架——Cypress 从入门到精通》。电子工业出版社，2020 年 4 月。补新一代前端测试的思路，就是同源运行、自动等待、时间旅行那一套。前端方向的同学先看这本。

>

第四本，《Robot Framework 自动化测试精解》。人民邮电出版社，2020 年 4 月。补关键字驱动这条路。它用表格和关键字组织测试，思路和写代码的框架完全不同，门槛低，业务测试同学也能参与。团队里"测试同学不写代码"是个真问题，这本给的就是这个答案。

>

最后一句：书不用全看。挑一本，把里面的例子在你的实验项目上跑通。跑通一本，比翻完四本管用。

提问：如果团队里测试同学不写代码，你会推荐哪本给团队？为什么？

预设回答：

- 答"Robot Framework 那本"——合理。关键字驱动门槛最低，业务同学能读懂、能改。

- 答"但长期还是要会写代码"——也得说。关键字驱动能做常规场景，复杂逻辑仍要编码能力。

- 答"先看团队现状，再看哪本"——这就是工程判断：工具选型要和团队能力匹配。

- 收口：选型时"团队接得住"比"技术更先进"重要得多。

> **易错点 / 考点：**这一页一般不考。但要知道"接口层优先于界面层"这个投入产出判断——简答和答辩的常见加分点，也是很多团队走弯路的地方。

> **过渡：**这一讲到这里结束，我把主线收一句。

【6.71】感 谢 聆 听（2 分钟）

> 课到这里，我不总结全书，留三句话。

>

> 第一句，工具的选择由场景决定，不由流行度决定。同源还是跨进程、单应用还是跨应用、界面层还是接口层，先想清楚要测什么，再挑工具。反过来，先有工具再找场景，就是买了锤子看什么都像钉子。

>

> 第二句，自动化不是把测试多做一遍，是让反复执行的验证变便宜。它省的不是第一次的时间，是第 N 次的时间。所以你第一次写脚本一定比手工慢，这是正常的，别在这个阶段放弃。

>

> 这三句你答辩时能直接用。评委最爱问的就是那一句：为什么用这个工具，为什么不用手工。答得出场景和成本，这道题就过去了。

>

> 第三句，回归测试守的是承诺，精准测试靠的是证据。一个是态度，一个是方法。两样都有，质量才不是一句口号。这三句有先后：场景决定工具，成本决定值不值得做，证据决定敢不敢发。

>

> 落到你的课程设计上，具体要求只有一条：说得出哪几条用例是回归用的，为什么是这几条。

>

> 一圈走下来。Web 端有 Selenium、Cypress。移动端有 Appium、Espresso、UI Automator。再到改完之后怎么守住成果。下一讲进性能测试：功能测试问对不对，性能测试问扛不扛得住。谢谢大家。

提问：如果让你用一句话向没上过这门课的同学解释"什么是回归测试"，你会怎么说？

预设回答：

- 答"就是把以前的测试再跑一遍"——对，但只说到动作，没说目的。

- 答"就是确认新改动没把老功能弄坏"——这才是完整的说法。

- 答"就是防止改 A 坏 B"——很生动，也好记。

- 收口：三句都对，差别在有没有说清目的。以后写测试报告、做答辩也这样：先说目的，再说动作。

易错点 / 考点：这一页不考。但"回归测试守承诺、精准测试靠证据"这两句，是整讲的收口，答辩时最好用。

过渡：本讲到此结束，下一讲性能测试见。

四、思政落点清单（共 9 处）

序号	所在页	落点主题
1	6.5	把功能、逻辑、接口、界面、数据、操作、平台七个维度摆在一起，本身就是一种训练：不
2	6.11	WebService 的 SOAP、WSDL、UDDI 三件套，本质是把"两个系统怎么
3	6.15	分层自动化的取舍，说到底是会算成本：知道哪一层的投入产出比最高，就把资源压在那里
4	6.32	这张图里藏着一个值得学的设计：用一个进程边界换安全。驱动是唯一被允许操作浏览器的
5	6.38	两条命令就能跑起来，这是它最大的卖点。可得留一句：工具越容易上手，越容易让人跳过
6	6.43	整套机制能成立，靠的是系统相信签名相同就是同一个应用。签名在这儿不是一张图，是信
7	6.59	回归测试守的是一个承诺：你昨天能用的功能，今天还能用。用户不会因为我们加了十个新
8	6.64	一张映射表，本质是把"我觉得"换成"数据显示"。工程上最容易出事的地方，恰恰是那
9	6.68	第三个问题多说一句。面对效果不好，最容易的反应是找外部理由：工具不行、时间不够。