《软件测试课程设计》讲稿（第 6 章）

第 7 周 · 2 学时（90 分钟）

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）】【板书】【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）＋板书＋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**；其后是板书、提问与预设回答、易错点/考点、过渡语。板书建议边讲边写。
>
> 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）

> 各位同学，我们开始第 6 章——**系统功能测试**。
>
> 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。那时被测对象很小，对错能用断言钉死。这一章把镜头往外拉：被测对象变成**一整个跑起来的系统**——有界面、有接口、有权限、有第三方支付。这时候很多"对错"没有现成答案，它取决于**用户怎么用**。

所以这一章换了个问题：**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**。

四个板块：**方法与过程、自动化、回归测试、精准测试**。前两块是主体，后两块解决"改完之后怎么测才够"。

带着一个问题听：**你手机上最常用的那个 App，明天上线一个新版本，你敢说它的主要功能都还对吗？凭什么是你来说这句话？**

板书：中间写"第 6 章 系统功能测试"；左侧竖写"第 5 章：代码内部（单元／集成／契约）"；右侧写"第 6 章：系统外部（功能／接口／UI／自动化／回归／精准）"；底部一行"功能测试＝按用户的用法验证系统行为"。

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"——这是最典型的误解。"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，还要补一句：看**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。功能测试的判据是**需求与用户预期**，不是"程序没报错"。

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

易错点 / 考点：① 功能测试 ≠ 界面点点点，接口层、数据层、平台层都是功能测试的对象；② 功能测试 ≠ 手工测试，它可以自动化（本章 6.2 整节讲这件事）；③ 判断题常见陷阱："功能测试只关心功能，不关心性能和安全"——错，功能测试关心的是"在正常与异常条件下功能是否仍然正确"，异常条件本身就包含并发、越权、脏数据。

过渡：先看这一章的四个板块怎么排，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）

这一页是本章的目录，四块，我按顺序点一遍，你记结构就行。

6.1 功能测试方法与过程——地基。回答三个问题：系统功能测试**究竟测什么**（七个维度）、**要考虑哪些问题**、**基本思路**是什么；然后分两条路：**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。其中**分层自动化**决定你把力气花在哪一层。

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，算出这次改动该跑哪些用例。

四块的关系：**6.1 定对象，6.2 定手段，6.3 定范围，6.4 缩范围**。

板书：竖排四块：6.1 方法与过程（测什么／考虑什么／思路／接口／UI）→ 6.2 自动化（接口・Web UI・Cypress・Android＋分层）→ 6.3 回归（为什么・范围・策略）→ 6.4 精准（覆盖率・映射・依赖）。右侧一条竖线标注："6.1 定对象，6.2 定手段，6.3 定范围，6.4 缩范围"。

提问：如果只给你一周时间学这一章，你会先学 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。因为 6.2 的脚本内容，本质上就是把 6.1 的用例翻译成代码。

- 追问：那 6.2 的价值在哪？答：在于**让同一批用例可以被重复执行一万次**，这是手工做不到的。

- 收口：先学会"设计"，再学"复制"，顺序别反。

易错点 / 考点：① 本章四个板块的名称要能默写，简答题常考"本章主要内容"；② 别把 6.3 回归测试和 6.4 精准测试当成两件事——**精准测试是回归测试的一种实施策略**，它解决的是"回归范围怎么定"；③ 考试若问"功能测试自动化包括哪几类"，答四类，不要只答 Selenium。

过渡：好，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）

这一页是 6.1 节的分隔，我用一分钟把这一节的路画出来，因为这一节内容最杂，没路线容易走丢。

这一节分两段。**第一段"想清楚"**：功能测试的要求与基本思路、测试需求怎么分析；以及探索式测试、SBTM 这类"边测边设计"的方法怎么用。

第二段"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）和**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？因为**接口是骨架，UI 只是外面那层皮**。逻辑绝大部分在服务端，从接口打直接打到本体；从界面点，中间隔着渲染、脚本、网络，出错还得先判断是皮还是骨头。

这也是行业惯例：**能测接口就不测界面**。这句话 6.2 还会再遇到。

板书：一条横线分两段。左段"想清楚"：要求与基本思路 → 测试需求分析 → 探索式测试／SBTM。右段"动手做"：面向接口（HTTP・WebService）｜面向 UI（分层・输入设计・角色路径・界面）。底部一行："接口是骨架，UI 是皮——能测接口就不测界面"。

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样式错位就红了，而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），这是界面层做不到的。

- 追问：那界面测试是不是可以省了？答：不能省，**它验证的是"用户能不能用"，这是接口替代不了的**。

- 收口：两者是互补，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**，不是测试类型（不是"接口测试"这个独立类型）；② 考试常问"为什么优先做接口测试"，标准答案三条：**更接近业务逻辑、构造输入更自由、执行更稳定更快**；③ 别把这一页当成纯目录跳过——它给出了本节的技术路线顺序。

过渡：进入 6.1 的第一页，先看这一节的方法清单里都放了什么。

【6.4】功能测试方法与应用（2 分钟）

这一页把 6.1 节要讲的方法摊开，六个词，我按"先想、再拆、后做"的顺序讲，你听完就知道每一块在整章里的位置。

第一块，功能测试要求和基本思路。这是接下来三页的内容：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。注意这个词："测试**需求**分析"，不是"需求分析"。区别在哪？需求分析是问"用户要什么"，测试需求分析是问"**我拿什么去验证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、优先级和风险**。这一步做不好，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。它的特点是不预先写完整用例，而是**边学习、边设计、边执行**，把测试过程当成一次有目的的探索。它的价值在于补"设计不出来的盲区"——很多缺陷不在任何人的用例里，而在"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**：把探索过程切成一段段有主题、有时限的"会话"，每段结束留下记录，这样探索式测试才可度量、可复盘、可交付。

第五块和第六块，回归测试及其策略、精准测试。这两块在本章后半 6.3、6.4 单独展开，这里先记住它们解决的是同一个问题：**改动之后，测多少才够**。

六块合起来是一条链：**想清楚（要求与思路）→ 拆出来（测试需求分析）→ 补盲区（探索式＋SBTM）→ 管改动（回归＋精准）**。

> **板书：**横排六块并标序号：① 要求与基本思路 ② 测试需求分析（产出：测试项・条件・优先级・风险）③ 探索式测试（边学边设计边执行）④ SBTM（会话化管理）⑤ 回归测试及策略 ⑥ 精准测试。底部一条链："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，你觉得差在哪？

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，这正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：一句话——**需求分析定义"应该是什么"，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，它是有**章程（charter）**、有**记录**、有**复盘**的结构化活动；③ SBTM 不是工具，是**管理方法**，别答成某个软件；④ 不要把"测试需求分析"和"测试设计"混为一谈，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）

这一页是本章最重要的一页，它给"功能测试"划边界。课件给了七个维度：**功能、逻辑、接口、界面、数据、操作、平台**。我一个一个讲，你对一下自己做过的东西。

功能——最直白：在不在、做没做全。下单要求能选商品、算价、用券、提交，少一个都不行，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、按钮该灰的时候灰不灰。

数据——最容易被忽略、后果最重的一维：存的字段与精度对不对、显示有没有脱敏、脏数据（空值、超长、特殊字符、重复提交）进来会怎样。

操作——键盘能不能操作、连点会不会重复下单、中途返回会不会丢数据、误操作能不能撤销。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。

七个维度齐了才叫"系统"功能测试；只测第一个，那叫"功能点核对"。

> **板书：**七行，每行"维度＋一句话追问"：

功能→**全不全**；逻辑→**规则对不对**；接口→**通不通、约定了吗**；界面→**看得懂、提示准吗**；数据→**存得对、脏数据扛得住吗**；操作→**人乱来会怎样**；平台→**换个环境还行吗**。底部一行："七维齐测才叫系统功能测试"。

> **提问：**一个"提交订单"的功能，你用七个维度各问一句，能问出几个可能的问题？

> **预设回答：**

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

- 答"连续点两下提交了两单"——操作维度，本质是**幂等与防重**没做，属于高频线上事故。

- 答"金额存进数据库变成了 99.99999"——数据维度，浮点精度问题，非常常见。

- 答"我只想到功能和界面"——很真实，这也正是这一页存在的理由：**想不到的维度，就是漏测的地方**。

- 收口：把七个词写在你的测试计划模板上，每次评审逐条对一遍，比凭经验想更可靠。

> **易错点 / 考点：** ① 七个维度的英文与中文要能对上（Function／Logic／API／UI／Data／Operation／Platform），选择题常给英文缩写；② 常考区分：**界面 ≠ 功能、数据 ≠ 数据库测试**（数据维度包含显示、精度、脱敏、脏数据）；③ 案例题若问"该功能应从哪些方面测试"，答题要按七维展开，只答"功能、性能、安全"会失分。

> **【思政落点 1】**：这七个维度讲的是一个朴素的道理：**系统是一个整体，任何一维塌了，用户感受到的都是"这软件不行"**。功能对、数据错，用户的钱就少了；逻辑对、平台不对，一半用户就打不开。工程上最危险的心态是"这不是我这一块的问题"。希望同学们做项目时习惯**从整体看自己那一块**——你写的接口，别人要拿它拼系统；你定的字段，后面有人要靠它算钱。**心中装着整个系统，才配得上"系统"这两个字。**

> **过渡：**七个维度告诉我们"测哪些面"，接下来要问的是"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）

这一页课件只给了一个设问，没有给答案——它是留给你想的。我把一个成熟测试人员在动手前会问自己的问题，归成七类，你以后拿到任何一个需求，都可以照这七问过一遍。

第一问：测到哪一层？ 是只验界面上的结果，还是要验接口返回、要验数据库落库？层次定得低，成本低但覆盖浅；定得高，覆盖全但投入大。这一问决定你的**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，权限缺陷就一定漏**。

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。边界在哪，就是第 3 章等价类与边界值要解决的问题。

第四问：状态怎么变？ 功能不是孤立的动作，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。**状态机没画出来，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，也都要问一句"对方挂了或超时了，我们怎么办"。

第六问：改动会波及谁？ 这一问直接通向 6.3 回归测试。每次改动前先问：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。要提前定：关键路径全过、缺陷收敛、覆盖率达标、遗留缺陷风险评估通过。

这七问不需要全答满分，但**必须都问过**。问过而决定不测，是判断；没问就漏掉，是失职。

板书： 七问竖排：① 测到哪层（界面／接口／数据）② 谁在用・怎么用（角色）③ 输入来源与边界 ④ 状态怎么变（状态机）⑤ 跟谁交互（外部依赖・故障兜底）⑥ 改动波及谁（→回归）⑦ 怎么算测完（准出标准）。右侧标注："问过而不测＝判断；没问就漏＝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"——这是最基本的三个，但还不够。

- 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，而不只是券本身。

- 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，而是**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：所以第四问的正确用法是**先画状态图，再测跳转**，而不是拿状态当标签。

易错点 / 考点： ① 这一页课件无文字，考法通常是案例题里"请说明你将从哪些方面考虑测试"，答题时按七问展开即可得分；② 最容易漏的两问是**状态**和**外部依赖故障**；③ 记住"准出标准要提前定"这个判断——所有测试计划类题目都可以引用。

过渡： 问题想清楚了，接下来是"按什么顺序、用什么骨架去测"，也就是基本思路。

【6.7】系统功能测试的基本思路（3 分钟）

这一页讲**基本思路**的前三条。思路这个词听起来虚，其实它决定了你先测什么、后测什么，以及什么时候可以停。我先给结论：**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。 需求文档给的是业务语言："用户可以下单并选择配送方式"。测试要把它翻译成测试语言：**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，就是一条条可执行的测试项。**树画不出来，说明需求你还没读懂**。

第二条，按层次分解，别一上来就测界面。 系统的层次是：数据层 → 服务层（接口）→ 展示层（UI）。测的顺序也应该这样：先确认接口契约对、数据落得对，再去测界面。反过来做，你会把大量时间花在"界面点不通"上，而点不通的原因可能是接口返回了 500——**从最上层往下排查，效率最低**。

第三条，沿主要路径推进，先主干后枝节。 用户使用产品有主路径：打开 → 登录 → 选商品 → 下单 → 支付 → 查看订单。先用最短时间把这条主干跑通，再回头补分支和异常。为什么？因为**主路径不通，其他测试都没有意义**；而且主路径跑通能最快暴露环境、数据、依赖的问题。

这三条合起来是一个很实用的顺序：**画树（拆）→ 分层（先接口后界面）→ 走主干（先通后全）**。你拿到任何一个陌生系统，都可以照这个顺序开工，不会乱。

板书：三行：

① **从需求出发**：功能 → 子功能 → 功能点（树的叶子＝可执行测试项）

② **按层次分解**：数据层 → 服务层（接口）→ 展示层（UI）——"从下往上测"

③ **沿主路径推进**：打开→登录→选商品→下单→支付→查订单，先通主干再补枝节。

底部："画树 → 分层 → 走主干"。

提问：为什么课件主张先测接口、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：界面报错时你无法判断是展示层、网络层还是服务层的问题。

- 答"接口测起来快，不用等前端"——对，这也是一条实打实的理由：接口可以独立于界面提前测，测试不必等开发全做完。

- 答"界面太脆，改一下就红了"——也很真实，UI 自动化维护成本高，先用接口把逻辑钉死，UI 层就只需要验"展示对不对"。

- 收口：三条理由合起来就是一句话——**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"这个三层分解是**简答高频点**，要能举例；② 常考判断："测试应该从用户界面开始，因为用户就是这么用的"——错，那是**验收测试**的视角，不是功能测试的实施顺序；③ 别把"主路径"理解成"只测正常流程"，它是**先通后全**，不是"只测正常"。

过渡： 还有三条思路，讲的是"怎么把测试做得可管理、可复用"。

【6.8】系统功能测试的基本思路（2 分钟）

这一页接着讲后三条思路，它们解决的是同一个问题：**怎么让测试不靠个人英雄主义**。

第四条，用输入域和组合方法把数据设计出来。 这就是第 3 章那套东西的落地：等价类划分把无穷输入压成有限几组，边界值盯住临界点，判定表和两两组合处理"多个条件同时起作用"。功能测试里真正难的从来不是"点按钮"，而是**决定点之前填什么**。记住一句话：**用例的质量，八成取决于数据的选取**。

第五条，用矩阵管住环境、角色、数据这三类变量。 平台维度（浏览器、机型、分辨率）、角色维度（普通用户、VIP、管理员、未登录）、数据维度（新数据、历史数据、脏数据）——它们是**相乘**的关系，不是相加。全组合跑不完，就用组合测试挑出代表性组合，再用风险判断补几条关键的。

第六条，可追溯、可闭环。 每条用例要能追到需求条目（正向追溯），每个缺陷要能追到用例和代码（反向追溯）。为什么强调这个？因为**没有追溯，你就无法回答"这个需求测完了吗"**，也无法在改动后算出"该跑哪些用例"——这正是 6.4 精准测试的技术基础。

再加一条实操建议：**测试过程中随时记录"想到了但没测"的点**。这些点往往就是风险清单，也是你写测试报告时最有价值的内容。

板书： 三行：

④ **输入设计**：等价类／边界值／判定表／两两组合——"用例质量八成在数据"

⑤ **三维矩阵**：平台 × 角色 × 数据（相乘关系，用组合测试取样＋风险补点）

⑥ **可追溯可闭环**：需求↔用例↔缺陷（正向／反向追溯）→ 精准测试的基础

底部："随时记录'想到未测'的点＝风险清单"。

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

预设回答：

- 答"因为要两两组合"——对，3 个浏览器 × 4 类角色 × 3 类数据就是 36 种，不是 10 种。

- 答"那就 36 种全测"——时间不允许，而且收益递减；正确做法是**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，但要在测试报告里**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**——取舍必须有依据、有记录。

易错点 / 考点：① "可追溯性"是**测试管理**的核心概念，简答题常问"什么是测试用例的可追溯性"；② 别把正向追溯和反向追溯混了：正向是**需求→用例**（覆盖了吗），反向是**缺陷→用例/代码**（哪条用例抓到的）；③ 组合测试的适用边界要清楚：它保证"任意两因素组合被覆盖"，不保证"三因素交互"，所以关键业务仍要手工补。

过渡：思路讲完，开始走第一条技术路线——**面向接口**的功能测试。

【6.9】面向接口的功能测试（1 分钟）

这一页是个转折，我们从"想"进入"做"。前面所有思路落到接口层，就变成三件事：**看懂协议、看懂契约、构造请求**。

为什么接口在 UI 前面讲？因为这个判断值钱：**逻辑绝大部分不在界面上，而在服务端**，界面只是调用者。从界面点，你只能输入"界面允许你输的"；从接口打，你能输入"协议允许你发的"——所以越权、缺字段、超大值、重复提交，界面测试抓不到。

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

带着一个问题听：**界面报"系统繁忙"时，你怎么判断是请求没发出去、被拒了、还是解析错了？**

板书：中间写"面向接口的功能测试"；左侧"看协议（HTTP 报文四段）"；右侧"看契约（SOAP・WSDL・UDDI）"；底部一行"界面能输的是界面允许的；接口能发的是协议允许的——后者更大"。

提问：界面报"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；测试时你应该**先有抓包**。

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

- 追问：如果没有抓包工具呢？答：那就用 curl 或 Postman 把同一个请求手工重放一遍，**能重放才能定位**。

- 收口：这就是为什么必须先懂 HTTP 报文结构——不懂结构，抓包给你看你也读不出来。

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

> **过渡：**先看 HTTP——所有接口测试的地基。

【6.10】HTTP 协议（3 分钟）

> 这一页讲 HTTP 报文结构。别小看它——接口测试里 90% 的"请求为什么不对"，答案都在这四段里。课件给的结构很干脆：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、该禁缓存的接口被缓存。**头**装的是元信息：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。**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，这是最直接的断言点：2xx 成功、3xx 重定向、4xx 客户端错、5xx 服务端错。注意：**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，只断状态码会漏掉全部业务错误。**响应头**给类型、长度、缓存、Cookie 设置；**响应体**才是业务结果。
>
> 为什么必须记结构？因为**你的断言就分布在这四段上**：状态行断状态码，头断内容类型与安全头（如 HttpOnly、X-Frame-Options），体断业务字段。结构不熟，断言只能写"不报错"。
>
> 课件还给了 W3C 的协议规范地址，要查标准语义就去那里看原文，比二手博客靠谱。

> **板书：**两列对照。

请求：**请求行**（方法・路径・版本）＋**头**（Content-Type・Cookie・鉴权・来源）＋**空行**＋**[体]**（业务数据）。

响应：**状态行**（状态码）＋**头**（类型・长度・缓存・Set-Cookie）＋**空行**＋**体**（JSON／XML）。

底部一行："断言分布：状态行→状态码；头→类型与安全头；体→业务字段"；再补一句红笔："200 ≠ 业务成功"。

> **提问：**一个接口返回 HTTP 200，但业务上其实失败了。这种情况你怎么测出来？

> **预设回答：**

- 答"那不可能，200 就是成功"——非常常见的误解。大量系统（尤其是国内常见的"统一响应体"风格）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，这就是为什么断言不能只写在状态行上。

- 追问：那是不是应该要求服务端改回用状态码表达业务错误？答：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**，而不是按自己的偏好。

- 收口：一句话——**先读契约，再写断言**。

> **易错点 / 考点：**① 请求与响应的四段结构、以及"空行是分隔符"这一点是**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 断言层次要分清：协议层断言（状态码、头）与业务层断言（响应体字段）**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——WebService。

【6.11】Webservice 接口（3 分钟）

> 这一页讲 WebService，三个关键词：**SOAP、WSDL、UDDI**。这套技术今天不算时髦，但银行、政务、ERP 这些领域活得很稳。
>
> 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**以及网络应用间的交互。三个要点：跨语言、跨平台、靠 XML 说话。为什么需要它？各家语言不通、系统不通，只能约定一种大家都能读的文本格式通过网线传。
>
> **SOAP**，XML Web Service 的**通信协议**，课件括号里写着"调用方法的规范"——这四个字关键：它不只传数据，还规定怎么表达"我要调哪个方法、参数是什么、返回什么、出错怎么报"。它还支持不同底层接口，如 HTTP(S)、SMTP。

WSDL，一份 **XML 文档，用于说明一组 SOAP 消息以及如何交换这些消息**——就是说明书、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，**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（标准 XML/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取回所需数据。

测试落点三处：**契约校验**（按 WSDL 校验请求结构，少必填、类型错该不该拒）、**错误处理**（SOAP Fault 是否规范返回，而不是抛 500 加一段 HTML）、**版本兼容**（WSDL 加字段后老客户端还能不能调）。

板书：四个词连带一句话：

WebService＝跨语言・跨平台・基于 HTTP・XML 描述

SOAP＝通信协议／调用方法的规范（可跑在 HTTP(S)、SMTP 上）

WSDL＝XML 描述文档：有哪些操作、参数、返回、类型（＝契约／说明书）

UDDI＝注册与查找机制（＝黄页，用 SOAP＋XML 发/查注册信息）

底部一句："SOAP 会说话，WSDL 是说明书，UDDI 是黄页，XML 是语言"；测试落点三点：契约校验・SOAP Fault・版本兼容。

提问：如果你拿到一个只有服务地址、没有 WSDL 的 WebService，你会怎么测?

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／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，**没有契约的接口测试只能靠猜**，猜出来的断言不可靠。

- 追问：如果开发说"WSDL 是自动生成的，不准"呢？答：那这本身就是个缺陷——**契约与实现不一致**，要让开发把 WSDL 固化为对外承诺。

- 收口：所以接口测试的第一个动作往往不是发请求，而是**要契约**。

易错点 / 考点：① 三个缩写的全称与作用必须能对上，**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：认为 WebService 只能用 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，这是它和 Word 文档的本质区别；④ 案例题问"企业系统对接怎么测"，答案要落到契约校验、异常返回、

版本兼容三点。

【思政落点 2】：WSDL 这个东西值得多说一句——它把"我能提供什么、你需要给我什么"写成了一份**双方都必须遵守的公开承诺**。这就是契约意识。现实里很多线上事故，根子上不是技术不够，而是**有人悄悄改了契约没说**：字段类型从整型改成字符串、必填改成选填、错误码换了含义，调用方一无所知。请同学们记住：**接口不是你一个人的事，它是你和一个陌生人的约定**。改之前先通知、先兼容、先留过渡期——这种"不给别人添麻烦"的自觉，就是职业素养。

过渡：接口这条路讲完了。换一条路——从用户看到的那一层切进去，面向 UI 的功能测试。

【6.12】面向 UI 的功能测试（3 分钟）

这一页讲面向 UI 的功能测试，五条，条条都是实操要求。

第一条，重点测试功能逻辑，按功能、子功能、功能点等层次展开。 注意"重点测**功能逻辑**"——面向 UI 不等于测界面好不好看，而是**通过界面验证背后的业务逻辑**。分层就是那棵树：功能（订单）→ 子功能（下单、支付、退款）→ 功能点（选配送、用券）。分层才能分优先级、分人力，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（第 3 章）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。特别提醒"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，从应用场景遍历主要操作路径。 这是 UI 测试独有的价值。接口能精确控制一次调用，但**用户不是这么用软件的**：他会搜索、筛选、比价、犹豫、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、错误提示是否指明怎么改、加载与空状态、极端数据（超长文本、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**。

板书：五行：

① 重点测**功能逻辑**：功能 → 子功能 → 功能点（分层定范围与优先级）

② 输入设计（第 3 章方法）→ 驱动执行 → **观察输出**（数据・金额・状态，不只看提示）

③ **扮演角色**，遍历主要操作路径（一条路走到黑，不中途重置）

④ 针对**不同设置**测（语言・时区・货币・通知・默认值）

⑤ 用户界面测试（布局・文案・必填・错误提示・空状态・极端数据）

底部："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所有功能点一遍？

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"没有权限的人试图操作"时暴露。

- 答"不同角色看到的菜单不一样"——对，这就是**角色可见性**：该看到的没看到是缺陷，不该看到的看到了更是缺陷。

- 答"场景不一样，路径不一样"——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：所以角色不是"换个账号登录"，而是**换一套前置条件和预期**。

易错点 / 考点：① 五条要能默写，简答题常问"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理解成"测界面"——课件写的是**重点测功能逻辑**；③ "观察输出"常被简化为"看有没有报错"，答题时要写出**数据、状态、提示**三类输出；④ 界面测试与功能测试的边界要清楚：界面测试查"表达"，功能测试查"行为"。

过渡：6.1 的方法讲完了。接下来是这一章的重头戏——怎么把这些测试**自动化**。

【6.13】6.2 功能测试自动化（1 分钟）

这一页进入 6.2——功能测试自动化。我先把立场说清，否则你会把它学成"工具课"。

最容易走偏的是：一头扎进工具，忘了为什么自动化。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——力气该往哪一层花；然后拆接口自动化的链路；最后是 UI 自动化。

这一节的落点不是让你成为工具高手，而是能回答三个问题：**哪些测试值得自动化、在哪一层自动化、怎么才不烂掉**。第三个最现实——大量项目不是写不出来，而是三个月后没人维护、全部失效。

带着一个问题听：**一个人、每周十小时做自动化，你会先做接口还是先做界面？**

> **板书：**中间"6.2 功能测试自动化"；左侧"分几类：接口／Web UI／Cypress／Android UI"；中间偏右"分层自动化（力气往哪花）"；右侧"接口链路：架构→流程→工具→Rest-Assured"；右下"UI 链路：Selenium→WebDriver→页面对象→Cypress／安卓"。底部："三个问题：值得自动化吗？在哪一层？怎么不烂掉？"

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，但代价最大：UI 脚本脆弱、维护费高、单条执行慢。

- 答"先做接口"——多数情况下的正确答案，理由下一页讲分层时给全。

- 追问：什么情况下先做界面？答：当**核心风险在界面交互本身**（比如复杂的表单校验、可视化编辑器）时，UI 才有优先权。

- 收口：先判断"风险在哪一层"，再决定"自动化在哪一层"，这个顺序别反。

易错点 / 考点：① 功能测试自动化包含的**四类**要能背；② 自动化的目标不是"替代人"，而是**替代重复**，探索性、易用性、一次性验证不适合自动化；③ 简答题常问"什么样的测试适合自动化"，答案要点：重复执行、结果可判定、输入可构造、需求相对稳定。

过渡：先看这一节的总类目，四类，我把边界划清。

【6.14】功能测试自动化（3 分钟）

这一页给出功能测试自动化的四条技术路线：**面向接口、Web 客户端 UI、基于 Cypress 的 UI、Android 应用 UI**。

接口自动化：被测对象是 HTTP 接口或 WebService，不经过界面。优势是**稳定、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层。

Web UI 自动化：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。价值是验证"用户真的能用"，接口全绿但按钮点不动，只有它能抓到。代价是脆弱、慢、维护贵。

基于 Cypress 的 UI 自动化：也测 Web 界面，但技术路线完全不同。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。这个差异既带来好处也带来限制。

Android UI 自动化：Robotium、UIAutomator、MonkeyRunner、Appium、Espresso、Selendroid 等等。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**，这决定了你能不能直接用控件 ID 操作元素，后面专门讲。

关于 iOS 和 Appium 的深入内容，这一节只到"知道它是什么、能做什么"：iOS 的自动化在后续课程里还会专门讨论；Appium 更细的用法，请同学按教材 6.2.5、6.2.6 两小节自己动手做一遍——**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**。

板书：四行：

① **接口自动化**：HTTP／WebService，稳定・快・可构造极端输入（主力）

② **Web UI 自动化**：Selenium 系，验"用户真能用"，代价是脆与慢

③ **Cypress**：测试跑在浏览器内部，与 WebDriver"外部指挥"路线不同

④ **Android UI 自动化**：Robotium／UIAutomator／MonkeyRunner／Appium／Espresso／Selendroid

底部："选择顺序：能接口不 UI；Web 看维护成本；移动端先看进程模型"。

提问：这四类里，哪一类最适合"每天跑一次"、哪一类只适合"发版前跑一次"？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 也可以每天跑，只要机器多"——可以，但要付出**维护成本**：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：那 UI 自动化真正的定位是什么？答：**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：自动化的分层不是技术偏好，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 归到"接口测试"里当干扰项——它是 **UI 自动化**；② iOS 与 Appium 在本节只作了解，考试不会深挖；③ 常见判断题："UI 自动化越多越好"——错，UI 层是**成本最高、收益最低**的一层，应尽量少；④ 记住 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"？这一页给了一张分层图，是本节最有价值的一张。

【6.15】分层自动化测试（3 分钟）

> 这一页讲分层自动化测试，我认为它是整个 6.2 节的**判断核心**。课件四句话，我先念一遍：**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（业务层）；概括起来——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。**
>
> 第一句讲**单元测试投入产出比最高**：它跑得最快，毫秒级，能每次提交全量跑；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。所以单位成本的缺陷发现数最高。
>
> 第二句讲 **API 自动化能接近 100%**。这里说的是**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"无法穷举，所以 UI 自动化的覆盖率天然上不去。
>
> 第三句最容易被漏掉：**适当加强手工端到端**。注意"手工"两个字。为什么自动化程度高了还要手工？因为端到端测的是**业务价值是否成立**——从搜索到收货这一整条链，"作为一个业务故事通不通"需要人来判断；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。分层的意义就是：**把大量验证压在"好测"的层，把少量验证留在"真实"的层**。
>
> 这张图给你的是成本账：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。

> **板书：** 画一个金字塔（从下往上）：**单元测试**（最宽：跑得快・定位准・最稳，RoI 最高）→ **API／接口测试**（可做到接近 100% 覆盖）→ **功能逻辑测试** → **UI 测试**（最窄：尽量避免）。左侧标注"越往下：缺陷更易定位、效率更高"，右侧标注"越往上：反映真实需求、更加接近业务"。底部一句："适当保留手工端到端

（业务层）"。

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"——这句很扎心但很真实。UI 自动化演示效果直观，单元测试写得再好，汇报时"什么都没看见"。

- 答"因为历史代码没有单元测试基础，补不起来"——非常现实的约束：遗留系统依赖太多，写一个单测要 Mock 半天，成本反而高。

- 追问：那这种团队该怎么办？答：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**成本最优化**，不是道德标准；说服别人靠的是算账，不是喊口号。

易错点 / 考点：① 四句话的顺序与关键词要能背，尤其"**RoI 最高**""**不必做全**""**适当加强手工端到端**""**尽量避免 UI**"；② 极易错：把"API 自动化 100%"理解成"API 测试能发现 100% 的缺陷"——课件说的是**自动化覆盖率**；③ 案例题问"给出该项目的自动化策略"，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 测试"不等于"不做 UI 测试"，关键路径仍要守。

【思政落点 3】：分层自动化背后是一种很朴素的工作方法：**把力气花在产出的地方**。同样一天时间，压在单元和接口层的收益，可能是压 UI 层的十倍——这不是技术高低之分，是**判断力**之分。同学们以后进了团队，一定会遇到"大家都说要做 UI 自动化"的氛围。这时候希望你敢做一件事：**把成本账算出来，再谈要不要做**。务实不是偷懒，恰恰相反，务实的人要为"不做"承担解释责任。**会算 ROI 的工程师，比会炫技术的工程师更难得。**

过渡：分层的底层和顶层都清楚了，现在把中间那层——接口自动化——拆开讲。

【6.16】API 自动化测试（2 分钟）

这一页是接口自动化的开场，接下来几页围绕它：架构、流程、工具、示例、DSL。

先用一句话概括它该有的样子：**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"。

为什么值得投入？**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。

难点也不在写脚本，而在**接口从哪来**和**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。

带着一个问题听：**一个系统 800 个接口，你怎么在两周内全部接入自动化？**

> **板书：**中间"API 自动化测试"；四周四个"自动"：自动**采集**接口 → 自动**生成**用例 → 自动**执行断言** → 自动**触发与报告**。底部三个理由："早（不等前端）・稳（不怕改版）・深（能造极端输入）"；再标一行："难点在接口来源与断言维护，不在脚本本身"。

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，而且写完还要维护。

- 答"先抓包，把真实请求存下来，批量生成"——这才是正解，也正是下一页架构里"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？答：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，靠的是**批量化流水线**，不是靠人力堆。

易错点 / 考点：① "接口自动化的优势"是简答高频题，三条要能说全；② 别把接口自动化和"性能测试"混谈，虽然工具可以复用，目的不同；③ 记住"接口来源"是工程化关键——这一点常被忽略，但正是架构题的得分点。

过渡：看架构。这张图信息量很大，我们一块一块拆。

【6.17】API 的 TA 架构（4 分钟）

这一页是接口自动化的**总架构**，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。 一是**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。二是**手工录入**：没文档或不可信时一条条填。三是**程序扫描**：扫代码里的注解或路由注册自动提取清单。四是 **HAR 导入**：把浏览器里真实发生的请求录下来导入。

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，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"，真正的业务是把多个接口串起来——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，而是**场景用例**。

第四段，回归。 触发四种：**按需回归**、**定时回归**、**即时触发**（提交就跑）、**事件触发**（如构建成功），之后执行与结果展示。

两个必记口径：**RAML 是 RESTful API 描述性语言**；**HAR 是基于 JSON、储存 HTTP 请求/响应信息的文件格式**。

板书：四段横排箭头：

来源：文档化工具｜手工录入｜程序扫描｜HAR 导入

定义：RAML（RESTful 描述语言）｜Swagger｜Markdown

用例：场景设定（选择＋组装 API）→ 验证（场景用例／手工组装／HAR 批量生成／代码迁移）

回归：触发（按需・定时・即时・事件）→ 执行 → 结果展示

底部两个口径：**RAML＝RESTful API 描述性语言**；**HAR＝基于 JSON 的 HTTP 请求/响应记录格式**。

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，但它是"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的节点上，这是 CI 的标准做法。

- 追问：那按需回归还要不要？答：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，而是**按场景组合**——高频的挂即时/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**填空与选择高频点**；② RAML 与 HAR 的定义要一字不差地记（一个是**描述语言**，一个是**文件格式**），这两者最容易被互换；③ 常错点：以为"接口自动

化＝单接口用例"——业务价值在**场景组装**；④ 架构题答题顺序按"来源→定义→用例→回归"讲，逻辑最清楚。

> **过渡：**架构讲的是"平台该有什么"，接下来看"具体接入一个接口，要走哪几步"。

【6.18】API 测试流程（3 分钟）

这一页讲**批量接入**的具体流程。课件标题后面写着"API 的接入步骤-批量生成"，六个环节，是一条务实的流水线。

第一步，HAR 文件。 从浏览器或抓包工具把真实请求导成 HAR。为什么从这开始？因为它是**免费拿到了完整信息**：URL、方法、头、Cookie、请求体，几秒钟的事。

第二步，过滤与保存。 HAR 里噪音很多：图片、样式、埋点、广告全在里面。要**按域名、路径、响应类型过滤**，只留业务接口，保存成结构化清单。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"的分水岭。**

第四步，Thrift code_gen。 针对使用 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。生成后调用接口就像调本地方法。

第五步，Dynamic Class / Loading 与 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"？因为**接口清单是变的**，把调用硬编码进脚本，接口一改就得重编；动态化让"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，形成可重复执行的用例。

六步的实质是：**把"人点出来的请求"变成"机器能重复执行的用例"**。注意每一步都在做同一件事——**去噪、参数化、解耦**。

> **板书：**六步竖排箭头：

① **HAR 文件**（抓真实请求：URL・方法・头・Cookie・体）

② **过滤与保存**（按域名/路径/类型筛，只留业务接口）

③ **脚本修改**（去噪・参数化・动态取前置数据）

④ **Thrift code_gen**（由接口描述生成客户端代码）

⑤ **Dynamic Class/Loading → Dynamic Call Method**（动态加载、动态调用，接口变了不用重编）

⑥ **动态测试**（组装参数→调用→断言）

底部一句："全程都在做三件事：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"——对，这是最典型的：不参数化，脚本第二天全部 401。

- 答"因为数据有依赖，比如下单要先有商品 ID"——也很关键，这类**前置数据依赖**如果写死，环境一换就全红。

- 答"那我干脆每次重新录一遍 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：所以这一步的产出不是"能跑的脚本"，而是**可维护的用例资产**——这才是自动化的价值所在。

> **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**（获取真实请求）、**参数化的目的**（可维护、可跨环境）；③ Thrift 属于 RPC 框架，注意别和 REST 混为一谈；④ "动态加载/动态调用"的动机是**应对接口变化**，这是架构设计题的高分点。

> **过渡：**理论讲完，看两个真实示例。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）

> 这一页是 Swagger UI 的实测截图，我要你看出三件事。
>
> **第一，Swagger UI 不只是文档，它本身就是个轻量测试工具。** 一个接口在页面上展开之后，你能看到它的方法、路径、参数表、每个参数的类型和是否必填、以及各种响应码对应的结构。更关键的是有那个"试一试"的入口：填参数、点执行，它就把请求发出去，把 curl 命令、请求 URL、响应体、响应头全部回显给你。**这意味着你可以在没有写一行代码的情况下，完成一次接口验证。**
>
> **第二，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，这四个动作就

能覆盖掉大部分参数校验缺陷。

第三，它暴露了"文档即契约"的脆弱性。 Swagger 文档常常是代码注解自动生成的，实现改了文档没同步，就会出现"文档说必填、实际不校验"或"文档没有这个字段、实际返回了"。所以**用 Swagger UI 测出问题的时候，先确认是实现的错还是文档的错**——这两类都要报，但性质完全不同。

课件把这一页放在"API 测试示例"之前，用意很明显：**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。

板书：三行：

① **Swagger UI ＝ 文档 ＋ 轻量测试器**（展开接口→填参→Execute→回显 curl／URL／响应体／响应头）

② 参数表＝**测试输入清单**（必填／选填・类型・范围 → 四招：全填・漏填・错类型・超范围）

③ **文档即契约的脆弱性**：实现与文档不一致时，两类缺陷分开报

底部："先摸一遍，再写脚本"。

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。要看这个字段在业务上是否真的必需：如果缺了它下游会算错，那就是**实现没校验**，属于功能缺陷。

- 答"实现的缺陷，必须校验"——多数情况是，但要区分"协议层必填"和"业务层条件必填"。

- 追问：那怎么定？答：**看契约的意图**——如果它在所有场景下都不可缺，就必须校验；如果是"某些场景才需要"，那文档要写清条件。

- 收口：所以这条缺陷报告要写清"文档怎么写的、实际怎么表现的、业务上应该怎样"，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，考试若问"能否用它做接口测试"要答能，但**不适合做批量回归**；② 别把 Swagger 与 SoapUI 混：前者是**定义与文档**为主，后者是**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：看一个更完整的接口测试示例，这次是带断言的。

【6.20】API 测试示例（2 分钟）

这一页是接口测试的完整示例，我按"一个合格的接口用例由哪几部分组成"来讲，你照这个骨架去写就不会漏。

第一部分，前置准备。 要有可用的环境和数据：服务起来了、数据库里有测试账号、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，就是因为把前置条件藏在了自己脑子里。

第二部分，请求构造。 方法、路径、参数、头、体，五样齐了才算一个完整请求。特别注意头和鉴权——接口测试里 401/403 类问题的根源往往不是逻辑，是令牌没带对或过期了。

第三部分，响应断言。 这是示例的核心。断言要分层：**协议层**断言状态码、内容类型；**结构层**断言响应体的字段是否存在、类型是否正确；**业务层**断言字段的值是否符合预期（金额、状态、条数）。只断第一层等于没断。

第四部分，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能发现。

第五部分，清理。 用例要能反复跑，就必须自己收尾：删掉造的数据、恢复状态。**不能重复执行的用例，在回归里就是负担**。

这五部分合起来才是"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**——以后每写一条接口用例，都问自己这五部分齐不齐。

板书：五段：

① **前置**（环境・账号・数据，写进用例不藏脑子里）

② **请求**（方法・路径・参数・头・体；鉴权别漏）

③ **断言**（协议层：状态码/类型 → 结构层：字段/类型 → 业务层：值）

④ **数据校验**（落库对不对、有没有重复）

⑤ **清理**（可重复执行的前提）

底部："五段齐了，才算一条能进回归的用例"。

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：订单表里堆满垃圾数据，查询变慢、统计失真，甚至触发业务规则（比如"每人限领一次"）。

- 追问：清理要怎么设计？答：**优先用独立的测试账号和隔离数据**，其次才是删除；删不掉的设计成"幂等可重复"的用例。

- 收口：一句话——**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点，五段要能说全；② 断言分层（协议／结构／业务）是评分点，只答"断言返回结果"不给满分；③ "数据校验"常被漏，凡涉及写操作的接口都必须验；④ 清理与幂等是工程化考点，论述题可以引用。

> **过渡：** 手工这样写效率太低，所以要有工具。下一页是接口测试工具的全景。

【6.21】基于 API 的 TA 工具（2 分钟）

这一页是接口测试工具的全景。我先给你一个分类框架，因为工具名字一大堆，没有框架记不住。

按用途分四类。**第一类，接口测试执行工具**——把请求发出去、断言、出报告。**第二类，API 定义与文档工具**——维护契约本身，Swagger 是代表。**第三类，Mock／桩工具**——被测系统依赖的第三方还没好，或者你想造异常响应，就用它来假装对方。**第四类，管理与协作平台**——管用例、管环境、管团队协作。

这一页课件列了四个偏"测试执行与平台"的工具：**SoapUI**，老牌的接口测试工具，对 SOAP 和 REST 都支持，界面化、能组织测试套件；**Rest-Assured**，Java 生态里的 REST 接口测试 DSL，写起来像自然语言，下一页重点讲；**SOAtest**，偏企业级的服务与接口测试平台，擅长复杂协议与消息格式；**APIfortress**，偏 API 的持续测试与安全／功能验证方向。

选工具的时候记住三条判断：**能不能进 CI**（不能自动触发就没有持续价值）、**断言能力够不够**（能不能做 JSON 路径断言、能不能校验 schema）、**维护成本高不高**（界面化工具好学但不好规模化，代码化工具上手慢但好维护）。

还有一条现实建议：**别为了工具而换工具**。团队用 Java，就优先 Rest-Assured；团队已经在用 Swagger 生态，就把它的用例生成打通。工具是给流程用的，不是给简历用的。

> **板书：**四类框架：**执行工具**（SoapUI・Rest-Assured・SOAtest・APIfortress）｜**定义文档**（Swagger・RAML）｜**Mock 桩**（下一页）｜**管理平台**（用例・环境・协作）。右侧三条选型判据：**能进 CI**｜**断言够强**｜**维护成本可控**。底部："别为了工具换工具"。

> **提问：**SoapUI 界面友好、上手快，为什么很多团队最后还是转向 Rest-Assured 这类代码化工具？

> **预设回答：**

- 答"因为代码化工具能进版本管理"——关键理由之一：脚本和被测代码放在同一个仓库，能评审、能分支、能追溯。

- 答"因为界面化工具不好做参数化和批量生成"——也对，几百条用例靠界面点，效率上不去。

- 追问：那 SoapUI 就没用了？答：不是，**探索阶段、临时验证、非技术同学参与**时它很好用。

- 收口：工具的选择取决于**规模**：几十条用例用界面工具最快，上千条就必须代码化。

> **易错点 / 考点：**① 四个工具的名称与定位要能对上，选择题常拿"Mockbin／wiremock"混进来当干扰（它们是 **Mock 工具**）；② 选型三判据可作论述题骨架；③ 记住"工具服务于流程"，这类表述在开放题里很加分。

> **过渡：**Mock 这一类工具单独一页，因为它解决的是接口测试里最头疼的问题——依赖。

【6.22】基于 API 的 TA 工具 -续（3 分钟）

这一页接着讲 Mock 与桩工具。重点讲 Mock，因为它是接口测试能不能独立跑起来的关键。

为什么需要 Mock? 两个场景。**一是依赖没就绪**：要测下单，支付网关还在开发，就用 Mock 假装它，返回"支付成功"。**二是依赖不可控**：第三方有时快、有时超时、有时返回错误码，这些异常你没法让它真实发生，只能自己造。

课件给了六个 Mock 工具。**Mockbin**：可以从 HAR 文件生成模拟桩——把真实响应直接变成桩。**RAP**：淘宝开发的 API 管理工具，文档、Mock、团队协作一体。**Swagger**：流行的 API 定义工具与规范，能按定义自动生成 Mock 响应，也常被当 Mock 用。**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。课件还列了两个工程师向的：**kristofa/mock-http-server**（轻量 HTTP Mock 服务）和 **wiremock**（Java 生态常用，能配请求匹配规则与响应模板，也支持录制回放）。

选 Mock 工具看三点：**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**（文件还是只能界面点）。

上一页四个执行工具——**SoapUI、Rest-Assured、SOAtest、APIfortress**——与 Mock 不是竞争关系：**一个假装对方，一个考问自己**，配合起来才完整。

板书：左栏"Mock／桩（假装对方）"：**Mockbin**（HAR→桩）｜**RAP**（淘宝，API 管理＋Mock）｜**Swagger**（定义→生成 Mock）｜**Easy Mock**（Swagger→假数据）｜**Doclever**（文档＋测试数据）｜**Mocky**（免登录快速生成 response）｜**mock-http-server**｜**wiremock**（Java 常用，可配规则/录制回放）。右栏"执行工具（考问自己）"：SoapUI・Rest-Assured・SOAtest・APIfortress。底部三条选型：**按请求匹配｜能造异常｜配置可进 CI**。

提问：如果第三方支付接口在生产上偶尔超时，你怎么用 Mock 把这种"偶尔"测出来?

预设回答：

- 答"让 Mock 返回 500"——只能测到一种异常，测不到"慢"。

- 答"用 Mock 固定延迟几秒再响应"——对了，这才是关键招：**延迟比错误更阴险**，因为很多系统对超时没有兜底，会一直等、把线程池占满。

- 追问：那超时之后的重试和幂等怎么测？答：Mock 第一次超时、第二次成功，验证系统**重试是否会重复扣款**——这是支付类系统的经典缺陷。

- 收口：所以 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：① 六个 Mock 工具与它的特点要能对上（尤其 **Mockbin 从 HAR 生成**、**Easy Mock 从 Swagger 生成数据**、**Mocky 免登录**）；② 极易错：把 Swagger 只当文档工具——它也能生成 Mock；③ 概念区分：**Stub 返回固定值，Mock 还能验证调用行为**，答题时别混；④ Mock 的测试价值在"造异常"，这一点是案例题的高分点。

过渡：工具讲完，看一个真正把接口测试写成"句子"的框架——Rest-Assured。

【6.23】Rest-Assured（3 分钟）

这一页讲 Rest-Assured。课件定位句：**它是一个用于测试和验证 RESTful 服务的 Java DSL，使得为基于**

HTTP 的 RESTful 服务编写测试变得更加简单。关键词 **DSL**——领域特定语言，就是为某件事专门设计的一套说法。

看课件那条断言：`when().` 接 `get("/lotto/{id}", 5)`，`then().` 接 `statusCode(200)`，再 `body("lotto.lottoId", equalTo(5), "lotto.winners.winnerId", hasItems(23, 54))`。没学过 Java 也能读懂：当……发 get；那么状态码应 200；lottoId 应等于 5，winnerId 应含 23 和 54。

这就是 DSL 的价值：**它把测试意图直接写成句子**。结构三段——**given（前提）／when（动作）／then（断言）**：given 放前提和参数，when 放请求，then 放断言。它与 BDD 一致，所以不只**好写**，还**好读**。

三个细节。**路径参数**：上面 `{id}` 被 5 替换，这就是参数化。**JSON 路径断言**：`"lotto.winners.winnerId"` 用点号往下走，**不必反序列化整个响应**就能断言。**Hamcrest 匹配器**：`equalTo`、`hasItems` 由 Hamcrest 提供。

课件给了官网 rest-assured.io、GitHub 仓库和示例地址，先把示例跑通，再故意让它失败——**故意失败是理解断言最有效的方式**。

板书：三段结构：

given() 前提（参数・头・体）→ **when()** 动作（get/post…）→ **then()** 断言（statusCode／body／header）

示例骨架：`when().get("/lotto/{id}", 5).then().statusCode(200).body("lotto.lottoId", equalTo(5), "lotto.winners.winnerId", hasItems(23, 54))`

右下三个词：**路径参数 {id}｜JSON 点号路径｜Hamcrest 匹配器（equalTo／hasItems）**；左下："Rest-Assured ＝ RESTful 服务的 Java DSL"。

提问：为什么说这套 DSL 让"测试可读"这件事变得重要？可读性对测试有什么实际好处？

预设回答：

- 答"好看一点而已"——不只是好看。**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"可读的测试能当文档用"——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：不可读的脚本三个月后就没人敢动，然后烂掉。

- 收口：所以 DSL 不是语法糖，它是**降低协作成本**的工程手段。

易错点 / 考点：① given-when-then 三段结构要能背，并说明各段放什么；② `body("a.b.c", matcher)` 的**点号路径**断言是常考点，注意它不是 XPath；③ 常错：以为 Rest-Assured 只能测 REST——它的定位就是 RESTful 服务；④ 记住它是 **Java DSL**，语言前提是 Java（或 Groovy/JVM 系），这条常出现在选择题里。

【思政落点 4】：given-when-then 这个结构值得多想一层：它要求你**把"前提、动作、预期"分开写清楚**。这恰恰是很多团队沟通出问题的根源——需求里只写了"要做个下单功能"，没写前提（什么角色、什么库存状态）、没写预期（成功是什么样、失败是什么样）。当测试用例被迫用三段式表达时，**模糊的地方就藏不住了**：你会立刻发现"这个需求没说要什么结果"。所以写好测试，本身就是一种**把话说清楚的能力**。希望同学们练出这个习惯：**任何结论都要能拆成前提、依据和预期**——这是工程师最硬的沟通本领。

过渡：三段式里最花功夫的是参数怎么传，下一页专门讲参数化。

【6.24】Rest-Assured 参数化（2 分钟）

这一页讲参数化。三条写法看着简单，却对应 HTTP 三种参数位置，最容易错。

第一种，表单参数 `formParam`。 例子同时用了两个：`given().formParam("formParamName","value1").queryParam("queryParamName","value2").when().post("/something")`。`formParam` 放**请求体**按表单编码，`queryParam` 拼在 **URL 后**。

第二种，通用 `param`。 例子是 `given().param("param1","value1").param("param2","value2").when().get("/something")`。`param` 会自动判断：GET 当查询参数，POST 表单当表单参数。方便，但**不精确**——需要明确位置时，还是用具体方法。

第三种，直接写在 URL 里。 `..when().get("/name?firstName=John&lastName=Doe")`。最直白，缺点是**值里有特殊字符（空格、&、中文、斜杠）就得自己编码**，忘了就出玄学问题。

三条写法背后是同一个知识点：**参数在报文里的位置，决定它怎么被服务端读**。放错位置服务端读不到，用例红得莫名其妙——被测系统没错，是**你的请求构造错了**。这种"假缺陷"占比不低。所以参数化前先确认契约里参数在哪（query、path、form、body、header），别用"万能 param"糊过去。

板书：三种写法对照：

formParam → 请求体（表单编码）｜**queryParam** → URL 查询串

param → 自动判断（GET→查询、POST 表单→表单），方便但不精确

直接拼 URL：`/name?firstName=John&lastName=Doe` → 需自行编码特殊字符

底部一句："参数位置由**契约**决定，不是由写法方便决定"；红字提醒："位置错了 → 假缺陷"。

> **提问：**一个接口你用 `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 URL 上它当然读不到。

- 答"参数名拼错了"——也是高频原因，尤其大小写和下划线／驼峰不一致。

- 追问：怎么快速区分"我的错"和"它的错"？答：**用 Swagger 或开发给的 curl 命令重放一次**——官方示例能通、你的不能通，就是你的构造问题。

- 收口：所以接口测试的第一条纪律是：**先让请求与契约完全一致，再去怀疑实现**。

> **易错点 / 考点：**① 三种参数写法的对应位置要能对上，这是**选择题高频**；② 极易错：以为 `param` 只能用于 GET——它对 POST 表单也可用；③ URL 直接传参的**编码问题**（中文、`&`、空格）是实战易错点，也是案例题的可写点；④ 记住"假缺陷"这个概念，它对应"测试代码的缺陷"，报告里要区分。

> **过渡：**Rest-Assured 单独跑还不够，它得挂到测试框架里才有组织，看下一页。

【6.25】Rest-Assured + JUnit（2 分钟）

这一页是 Rest-Assured 和 JUnit 的结合，也就是**把接口测试纳入单元测试框架的组织方式**。这件事看着是"拼装"，实际解决三个问题。

第一个问题，怎么组织。 JUnit 提供类和方法的结构、生命周期的钩子（每个用例前准备、后清理）、以及断言失败时的报告。Rest-Assured 只负责"发请求、取响应、做匹配"，它不管用例怎么排序、怎么统计、怎么出报告。**两者分工：JUnit 管组织，Rest-Assured 管交互。**

第二个问题，怎么复用。 一条接口用例里，`given()` 的前置部分（基础地址、公共头、鉴权令牌）几乎每条都要写。JUnit 的生命周期方法正好用来做这件事：在"所有用例之前"读配置、拿令牌、建好基础请求；在"每条用例之前"重置状态。这样用例本体就只剩下"这次要传什么、期望什么"，**可读性立刻上一个台阶**。

第三个问题，怎么进流水线。 JUnit 是 Maven／Gradle 生态里的一等公民：构建工具认识它、CI 认识它、报

> 告插件认识它。这意味着你写完的接口用例，**不需要额外开发一个执行器**就能接进持续集成——这一点是"能进 CI"这条选型判据的直接体现。
>
> 顺便说清一个容易混的地方：用 JUnit 跑接口测试，**被测对象不是"单元"**，我们借的只是它的**组织和执行机制**。所以别纠结"这算单元测试还是接口测试"——**按被测对象分类，它属于接口测试；按执行框架看，它跑在 JUnit 上**。这两件事不矛盾。

板书：三行：

① **分工**：JUnit 管组织（结构・生命周期・报告）｜Rest-Assured 管交互（请求・响应・匹配）

② **复用**：`@BeforeAll` 读配置／取令牌／建基地址 → `@BeforeEach` 重置状态 → 用例只写"这次传什么、期望什么"

③ **进流水线**：JUnit 被 Maven／Gradle／CI／报告插件原生识别 → 不用自研执行器

底部："按对象分类＝接口测试；按框架看＝跑在 JUnit 上"。

提问：把公共的令牌获取写在"所有用例之前"执行一次，会带来什么风险?

预设回答：

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

- 答"应该每条用例都重新取"——更稳，但会慢；折中做法是**缓存＋失效重取**。

- 追问：那怎么判断是"令牌过期"还是"真的没权限"？答：看响应体和错误码，再单独造一条"用过期令牌访问"的用例把它固化下来。

- 收口：所以共享前置状态是把双刃剑——**省了时间，也引入了耦合**，要显式处理失效。

易错点 / 考点：① "JUnit 提供组织、Rest-Assured 提供交互"这个分工是简答得分点；② 常错：认为接口测试必须用专门的接口测试工具，**用单元测试框架完全可以**；③ 生命周期方法的用途（前置准备、清理、共享资源）是高频考点，与第 5 章内容呼应；④ 共享状态的失效处理是工程化细节，论述题可加分。

过渡：把前面这些拼起来，就是一个完整的 REST API 测试工程。

【6.26】REST API 测试（2 分钟）

这一页把前面几页拼成一个完整的 REST API 测试工程。我按"一个工程该有什么"清点一遍，这也是你做实验和大作业的检查单。

第一，工程结构。 至少要分层：**配置层**（环境地址、账号、超时）、**接口层**（每个接口一个封装，写好路径、方法、参数模板）、**用例层**（场景与断言）、**数据层**（准备与清理）。不分的后果是：换个环境要改几十个文件。

第二，请求封装。 别把 `given()` 散落在每个用例里，**把"怎么发"和"发什么"分开**：封装层管怎么发（基地址、公共头、日志、重试），用例层只给发什么（参数、期望）。这和后面第 34 页的 Page Objects 是同一个思想——**把操作细节从测试意图里剥出去**。

第三，断言策略。 三层：协议层、结构层、业务层。再补一条：**关键接口要加 schema 校验**——不只断言"字段在不在"，而是断言"结构是否符合定义"，这样字段类型被悄悄改掉时你能立刻发现。

第四，执行与报告。 要能一键跑全量、按标签挑子集（冒烟／全量）、失败重跑，报告里能看到请求与响应原文。**报告不好用等于没有自动化**，因为没人愿意去看。

第五，环境与数据治理。 每个环境一套配置；测试数据自动造、自动清；令牌密码不写进代码仓库。

这五条是"能不能长期活下去"的清单。你以后接手一个烂掉的接口自动化项目，多半能在这五条里找到死因。

板书： 五行：

① **工程结构**：配置层／接口封装层／用例层／数据层

② **请求封装**：怎么发（基地址・公共头・日志）与发什么（参数・期望）分离

③ **断言策略**：协议层 → 结构层（schema）→ 业务层

④ **执行与报告**：一键全量・按标签挑子集・失败重跑・请求响应原文

⑤ **环境与数据**：多环境配置・数据自造自清・敏感信息不入库

底部："这五条是自动化的'活命清单'"。

提问： 为什么说"报告不好用等于没有自动化"？

预设回答：

- 答"报告只是记录，跑完就行"——错。自动化的价值靠**有人看结果**才能兑现，报告难读就没人看，失败了也没人修。

- 答"报告要能直接看出是哪一步失败"——对，所以要在失败时把**请求原文、响应原文、断言表达式**都打出来。

- 追问：报告还要有什么？答：**趋势**——通过率随时间的变化、不稳定用例（时红时绿）的识别，这些比单次结果更有价值。

- 收口：一句话——**自动化的产出不是脚本，是"可信的信号"**。

> **易错点 / 考点：**① 分层结构（配置／封装／用例／数据）是设计题标准答案；② "怎么发与发什么分离"这个原则要能表述，它与 Page Objects 同源；③ schema 校验属于"结构层断言"，注意与业务断言区分；④ 报告与趋势是论述题加分项，别只答"生成 HTML 报告"。

> **过渡：**工程化的东西讲完了，下面看工具的另一半风景——先看接口测试工具的选型，再看 UI 自动化那条线。

【6.27】UI 自动化测试（2 分钟）

> 这一页从接口层转到界面层。课件给的是三条路线：**Web 客户端的 UI 自动化测试、基于 Cypress 的 UI 自动化测试、Android 应用的 UI 自动化测试**。
>
> 在往下讲工具之前，我先把"UI 自动化的本质"说清楚，否则你会把它学成"记命令"。UI 自动化的本质是三件事：**找到元素、操作元素、验证结果**。所有框架，不管是 Selenium 还是 Appium 还是 Robotium，都在解决这三个问题——只是"怎么找"和"怎么操作"的机制不同。
>
> 那么 UI 自动化的价值在哪？前面说过，它验证的是"用户真的能用"。这话落到实处是三类缺陷：**一，元素层面的缺陷**——按钮点不动、输入框被遮挡、弹窗关不掉；**二，交互流程的缺陷**——多步操作之后状态丢了、返回按钮行为不对；**三，展示的缺陷**——数据错位、文案错、金额格式错。这三类接口层都测不出来。
>
> 它的代价也要认清楚：**慢、脆、贵**。慢是执行慢；脆是界面一改就红；贵是维护成本高。所以业界的一致做法是：**UI 自动化只守关键路径，用例数量控制在很少的量级**。
>
> 这三条路线里，Web 端的两条（Selenium 与 Cypress）技术路线正好相反，我会放在一起对比；安卓端则是另一套生态，难点在进程模型。带着一个问题听：**它们各自凭什么"找到"那个按钮？**

> **板书：**三行：**Web 客户端 UI 自动化**｜**Cypress UI 自动化**｜**Android UI 自动化**。中间写一句"UI 自动化本质三步：**找到元素 → 操作元素 → 验证结果**"。右侧标注价值三类：元素缺陷・交互流程・展示正确；左侧标注代价：**慢・脆・贵 → 只守关键路径**。

提问：所有 UI 自动化框架都在解决"找元素、操作元素、验证结果"，那它们的差别到底在哪？

预设回答：

- 答"差别在语法"——语法是表象。真正的差别在**机制**：靠什么通道去操作界面，以及**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，但同一平台（比如都测 Web）也有完全不同的机制，比如 Selenium 走外部驱动、Cypress 跑在浏览器内部。

- 追问：为什么机制重要？答：因为机制决定了**你能看到什么、能做什么、会踩什么坑**——比如跨进程的框架拿不到应用内部对象，只能模拟点击。

- 收口：所以这一节的重点不是"记 API"，是**理解机制**。

易错点 / 考点：① 三类路线的名称要能背；② 常考判断："UI 自动化能替代手工测试"——错，它只覆盖可判定、可重复的部分；③ "找到元素、操作元素、验证结果"这三步是很通用的答题骨架；④ UI 自动化的"脆"来自**界面与脚本的耦合**，这个判断后面讲 Page Objects 时正好是解药。

过渡：先看 Web 端最经典的一条路——Selenium。

【6.28】Web UI 的自动化测试（3 分钟）

这一页是 Web UI 自动化的**起点**：用 Selenium IDE 做录制回放。课件给的是一个初始化界面，两个设置项——**Project name 和 Base URL**，以及一句"启动 Chrome 的插件 Selenium IDE"。

先说这条路的定位：**Selenium IDE 是浏览器里的一个录制插件**。你在页面上操作，它把每一步记下来；你点运行，它把步骤重放一遍。**零代码门槛**，这是它最大的价值——产品、运营、刚入门的同学都能用它做一次快速验证。

那两个设置项为什么要填？**Project name** 是给这个测试工程起个名字，方便管理多个录制集；**Base URL** 是"被测应用的根地址"。为什么根地址要单独设？因为**同一个脚本要能在不同环境跑**：今天在测试环境、明

天在预发环境，只要把 Base URL 一换，脚本里的相对路径就都跟着走。这就是最原始的"参数化"——**环境参数化**。

但我必须把它的局限讲透，否则你会以为学会了它就会了 UI 自动化。**第一，录制出来的脚本很脆**：它是按你点到的元素记定位的，界面结构一变就找不到。**第二，它不好维护**：几十条录制脚本之后，改一个公共步骤要改几十处。**第三，它难进流水线**：它是浏览器插件，不适合在 CI 服务器上无人值守地跑。**第四，它不可编程**：复杂逻辑（循环、条件、数据驱动）写不出来。

所以正确的定位是：**Selenium IDE 用来"入门、演示、快速验证、生成初始脚本骨架"**；真正要规模化，必须过渡到代码化的 WebDriver。这条演进路线，下一节就讲。

板书：三行：

① **Selenium IDE ＝ 浏览器插件，录制→回放，零代码**

② 两个设置：**Project name**（工程名）／**Base URL**（被测根地址 → 换环境只改这里＝环境参数化）

③ 四条局限：**脆（元素一变就红）・难维护（改一处要改多处）・难进 CI（插件形态）・不可编程（循环/条件/数据驱动写不了）**

底部："IDE 入门 → WebDriver 规模化"。

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

预设回答：

- 答"为了方便改"——对，但要说清是为了**跨环境**：测试、预发、生产各自的域名不同。

- 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：如果一个脚本里混着好几个服务的地址呢？答：那就要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"配置与脚本分离"的最小实践，后面所有自动化都在做这件事。

易错点 / 考点：① Selenium IDE 的定位（**录制回放插件**）与局限是简答高频点；② 常错：以为录制的脚本可以直接用于回归——**必须先改造（参数化、加断言、去硬编码）**；③ "环境参数化"这个概念要能举例；④ 记住 Selenium 家族的演进：**IDE → RC → WebDriver**（RC 已被淘汰），答题时别说反。

过渡：录制出来的脚本长什么样？我们看它录完之后的那一屏。

【6.29】Web UI 的自动化测试 –续（2 分钟）

> 这一页是 Selenium IDE 录完之后的样子。我不要你记界面，我要你从这一屏读出四件事，因为这四件事决定了你后面写 WebDriver 代码时的思路。
>
> **第一，脚本是"步骤序列"。** 每一步包含三样东西：**动作（command）、目标（target，也就是定位器）、值（value）**。比如"在某个输入框里输入某段文字"，动作是 type，目标是那个输入框的定位表达式，值是文字。理解这个三元组，你就理解了所有 UI 自动化脚本的最小单元——**WebDriver 里也是这三样，只是换成了方法调用**。
>
> **第二，定位器是脚本的命门。** 录制工具通常优先用 id、name 这类稳定属性，没有就用 CSS 或 XPath 路径。**路径式定位是最脆的**：界面加一层 div，路径就变了。所以你看录制脚本时，第一件事就是检查**定位器是不是稳定标识**，不是就手工换成 id、name、或稳定的业务属性。
>
> **第三，断言是要自己补的。** 录制只能记"我点了什么"，记不了"我应该看到什么"。所以录完的脚本**默认是没有断言的**——它只能证明"操作没报错"，不能证明"结果正确"。这是录制回放最容易被误用的地方：跑绿了，但什么都没验证。
>
> **第四，等待是要自己处理的。** 录制脚本里常出现"点完立刻找下一个元素"，而真实页面是异步加载的。所以你会看到脚本时快时慢、时红时绿。这不叫"环境不稳定"，这叫**没做等待策略**。
>
> 一句话总结这一屏的教学价值：**录制的价值是给你一个可运行的骨架；能不能用，取决于你有没有补上定位器、断言和等待这三件事。**

> **板书：**四行：

① 脚本单元＝**动作（command）＋目标（target／定位器）＋值（value）**（WebDriver 里同样是这三样）

② **定位器是命门**：优先 id／name／稳定业务属性；路径式（XPath 全路径）最脆

③ **断言要自己补**：录制只记操作，不记预期 → "跑绿"≠"验证过"

④ **等待要自己处理**：异步加载 → 显式等待，别用固定 sleep

底部："录制给你骨架，定位器・断言・等待才给你可靠性"。

> **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，说明流程通了"——只能说明**操作没抛异常**，不说明结果对。这就是"零断言"的假绿。

- 答"要加断言才说明问题"——对，而且断言要断**业务结果**（订单号出现、金额正确），不是只断"页面打开了"。

- 追问：那怎么判断一条脚本有没有价值？答：**故意改坏被测系统的一个地方，看它会不会红**——不会红，说明它没在验证任何东西。

- 收口：这个方法叫"变异自检"，是检验自动化有效性的土办法，但非常管用。

易错点 / 考点：① "动作＋目标＋值"这个三元组是理解 UI 脚本的钥匙，简答可用；② 常错：只重视"能跑通"，忽视断言——**没有断言的自动化是假自动化**；③ 定位器选择优先级（id＞name＞CSS＞XPath 绝对路径）是实操考点；④ 固定 sleep 与显式等待的区别，是面试与考试的双高频点。

过渡：录制只能应急。要真正写脚本，先看 Selenium 支持用什么语言写。

【6.30】Selenium 支持不同脚本语言（2 分钟）

这一页讲 Selenium 的一个很重要的特性：**它支持多种脚本语言**。你用 Java、Python、C#、Ruby、JavaScript 都能写同一套测试。这件事为什么重要？两个层面。

技术层面：Selenium 的核心不是某一门语言的库，而是**一套协议**——语言绑定（binding）只是把"发命令"这件事包成你熟悉的语法。所以同一套操作，在不同语言里长得不一样，但发出去的东西是一样的。**理解了这一点，你就不会把 Selenium 当成"Java 的某个库"**，而会把它看成一个"跨语言的自动化协议＋各语言的客户端"。

工程层面：语言可选的真正价值是**让测试跟着团队的技术栈走**。团队是 Java 背景，测试就用 Java，能复用工具链、能共用构建、开发也看得懂；团队是前端背景，就用 JavaScript，写起来顺手。**测试代码不是孤岛，它要和产品代码住在同一个工程里。**

但这里有个取舍要说清：**多语言支持不等于你可以随便混用**。一个项目里今天 Java 写一批、明天 Python 写一批，结果是环境要装两套、报告要合两处、谁也不敢重构。**统一语言，比"用我最熟的语言"更重要。**

另外提醒一句：语言绑定只是外壳，**真正决定脚本质量的是定位策略、等待策略、断言设计、以及页面封装**——这四件事换任何语言都一样。所以别在"学哪门语言"上纠结太久，**先把这四件事练明白**。

> **板书：**中间写"**Selenium = 协议 + 多语言绑定**"；四周列语言：Java｜Python｜C#｜Ruby｜JavaScript。左侧标注"同一套操作，语法不同、发出的命令相同"；右侧标注"选语言的依据＝团队技术栈，不是个人偏好"；底部一行"脚本质量取决于：定位策略・等待策略・断言设计・页面封装（与语言无关）"。

> **提问：**团队里两个人，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，反正 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"——最稳的判断依据：能复用构建与工具链，开发也能参与维护。

- 追问：如果产品是 Java，但团队只有一个人会 Java 呢？答：那就让另一个人学 Java 的基础——**测试脚本的语言水平要求远低于开发**，几周就够上手。

- 收口：所以这不是技术选型题，是**协作成本题**。

> **易错点 / 考点：**① "Selenium 支持多语言"是常识考点，但要注意它**不代表脚本可以跨语言复用**——要重写；② 常错：把 Selenium 当成"某个语言的测试框架"；③ 论述题可写"统一语言降低协作与维护成本"；④ 记住决定脚本质量的四要素，与语言无关。

> **过渡：**语言定了，接下来是 Selenium 真正干活的那一层——WebDriver。

【6.31】Selenium + WebDriver（3 分钟）

> 这一页是 Selenium 与 WebDriver 的组合，也就是今天做 Web UI 自动化的标准做法。讲三件事：**什么关系、怎么工作、怎么写**。
>
> **先说关系。** Selenium 是一个**项目家族**：IDE（录制回放）、RC（早期靠向页面注入 JavaScript 驱动浏览器，已淘汰）、**WebDriver**（今天的主力）、**Grid**（把用例分发到多台机器、多个浏览器并行执行）。所以"用 Selenium 做自动化"，今天基本等于**用 WebDriver 的 API 写脚本**。这一点要分清，考试常拿 RC 和 WebDriver 对比。
>
> **再说它怎么工作。** 测试代码调用 WebDriver 的 API（找元素、点击、输入），WebDriver 把调用**翻译成标准命令**，通过 HTTP 发给**浏览器驱动**，驱动再去操作**真实浏览器**。两个关键词：**标准命令**意味着脚本不依赖某家浏览器的私有接口；**真实浏览器**意味着你测的是用户真正在用的那个，不是模拟器。

> **最后说怎么写。** 一个最小脚本四步：**启动驱动（打开浏览器）→ 访问地址 → 找元素并操作 → 断言并关闭**。这四步和"找到元素、操作元素、验证结果"完全对应。其中"找元素"最花功夫，它决定稳定性；"断言"最容易被偷工减料，它决定价值。
>
> 一条纪律：**每条用例结束都要关浏览器**，或交给框架统一管生命周期。见过太多人写完不关，跑一晚上开了几百个窗口把机器拖死——**自动化脚本自己也要讲资源纪律**。

板书：三段：

① **家族**：IDE（录制）｜RC（注入 JS，已淘汰）｜**WebDriver（主力）**｜Grid（分布式并行）

② **链路**：测试代码 → WebDriver API → **标准命令（HTTP）** → 浏览器驱动 → **真实浏览器**

③ **四步脚本**：启动驱动 → 打开地址 → 找元素并操作 → 断言并关闭

底部纪律："每条用例都要收尾关浏览器／交给框架管生命周期"。

提问：为什么说 WebDriver 比早期的 RC 更可靠？

预设回答：

- 答"因为 WebDriver 更新"——时间新不是理由，**机制**才是。

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

- 追问：那 WebDriver 为什么能绕开这些？答：因为它**不是注入，而是通过驱动从"浏览器外部"下命令**——这就是"跨进程"的思路，下一页的架构图讲的就是它。

- 收口：所以机制决定上限：**注入式受页面约束，驱动式受协议约束**，后者稳定得多。

易错点 / 考点：① 家族成员与定位要能对上（尤其 **RC 已淘汰、Grid 做分布式**）；② "标准命令 + 真实浏览器"这两个关键词是简答得分点；③ 常错：以为 WebDriver 需要给浏览器装插件——它不是插件方案；④ 资源收尾这条虽是实操，但写进测试规范里很能体现工程素养。

过渡：这条链路画出来到底是什么样？看架构图。

【6.32】WebDriver 架构（3 分钟）

这一页是 WebDriver 的架构图。我按"三段边界"讲，因为这张图真正的价值不是记住方框，而是理解**边界**。

第一段，客户端（测试代码＋语言绑定）。 你的 Java 或 Python 脚本加客户端库，构成"发命令的一方"。它只做一件事：把"点击这个元素"翻译成一条标准命令。**它不直接碰浏览器。**

第二段，传输层（协议）。 客户端和驱动之间用 HTTP 通信，命令与结果是约定的 JSON 结构。这一层的意义是：**只要双方都遵守协议，客户端用什么语言、驱动是哪个浏览器，都能配上**——这就是"跨语言、跨浏览器"能成立的根因。

第三段，浏览器驱动＋浏览器。 驱动是独立进程，接收命令、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，它不知道有人在测试它。**

三段之间是**两个进程边界**：测试进程 ↔ 驱动进程 ↔ 浏览器进程。这带来三个直接后果：**一，问题能在协议层被抓到**——脚本报"元素找不到"，看那条命令返回了什么，错误信息来自驱动，很具体；**二，端口和版本必须匹配**——驱动要监听端口，版本要配合浏览器，否则连不上，报错还很隐晦，这是新手最常踩的坑；**三，跨进程意味着不能直接读内存里的对象**——你只能按协议提供的动作去操作界面，拿不到应用内部的变量和状态。**这既是限制，也是安全边界。**

这张图记住一句话：**WebDriver 是"外部指挥"，不是"内部注入"**。

板书：三段横排：

① 客户端（测试代码＋语言绑定：只翻译命令）→ **② 传输层**（HTTP＋约定的 JSON 命令／响应）→ **③ 浏览器驱动（独立进程）→ 浏览器（不知情）**

标出两个进程边界：测试进程 ↔ 驱动 ↔ 浏览器。右下三个后果：**问题可在协议层看到｜端口与版本必须匹配｜不能直接读内部对象（只能按协议操作）**。底部："外部指挥，不是内部注入"。

提问：架构里为什么要多一个"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"因为需要转换"——对，浏览器对外没有统一的自动化接口，各家的原生接口不同，驱动就是**统一协议的适配层**。

- 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，**边界清晰、出**

问题好定位。

- 追问：这个设计对测试人员有什么实际好处？答：升级浏览器通常只需要换驱动版本，**脚本不用改**——这就是分层解耦带来的稳定性。

- 收口：多一层不是麻烦，是**用一层隔离换来了可替换性**。

> **易错点 / 考点：** ① 三段结构（客户端／协议／驱动＋浏览器）与"两个进程边界"是本节最容易被考的结构题；② 常错：以为 WebDriver 直接控制浏览器内核——中间必须有驱动；③ "环境问题优先查驱动版本与端口"是实战口诀；④ 与后面安卓的"跨进程 vs 同进程"对比，这一页是铺垫。

> **【思政落点 5】**：这张架构图里藏着一个很值得学的设计思想：**用边界换安全**。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，谁都不能越界去读对方内部的东西。这既是技术上的解耦，也是**权限上的克制**。现实中的很多事故恰恰相反：为了方便，把数据库账号直接给到应用层、把管理员令牌塞进前端、让测试脚本去改生产数据——**每一次"抄近路"都在拆掉一道边界**。希望同学们以后设计系统时问自己一句：**这一步我是不是越界了？** 守住边界不是保守，是对别人数据的基本尊重。

> **过渡：**架构清楚了，接下来是写得好的关键——怎么组织页面元素，先看最基础的 UI Object。

【6.33】UI Object（2 分钟）

> 这一页讲 UI Object，也就是"把界面元素抽象成对象"。这是所有 UI 自动化工程化的第一步，概念不难，但它是后面 Page Objects 和 PageFactory 的地基。
>
> **什么是 UI Object？** 简单说：**一个界面元素，在代码里应该有一个名字、一套定位方式、一组可做的操作、以及一组可以断言的状态**。比如"登录按钮"，它不是一个 XPath 字符串，而是一个对象：它知道怎么被找到、知道自己能被点击、知道自己点完之后页面该变成什么样。
>
> 为什么必须抽象？因为**界面元素是会变的**。今天按钮的 id 是 `login-btn`，明天前端改成 `submit`，如果你的 XPath 散落在二十个用例里，你要改二十处；如果它被封装成一个对象，你只改一处。
>
> 第二个理由更本质：**抽象能把"怎么找"和"要干什么"分开**。用例里应该写"点击登录按钮"，而不是写"找到 id 为 login-btn 的元素然后调用 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。
>
> 实操上有三个要点：**一，定位器要集中管理**，一个元素一个定义，不重复；**二，等待要封装在操作里**——"找元素"这件事本身就应该带等待，页面慢的时候自动等，而不是让每个用例自己 sleep；**三，元素的行为要收

敛，不要给一个元素对象加一堆业务方法，那会让它变成"上帝对象"。

一句话：**UI Object 解决的是"元素级"的复用**，下一层的 Page Objects 解决"页面级"的复用。

板书：中间写"**UI Object = 元素的名字+定位方式+可做操作+可断言状态**"。

左侧三条理由：**元素会变（改一处 vs 改二十处）｜把"怎么找"与"要干什么"分开｜可读性**。

右侧三条实操：**定位器集中管理｜等待封装进操作｜行为收敛（别做上帝对象）**。

底部："UI Object 管元素级复用 → Page Objects 管页面级复用"。

提问：为什么说"找元素"这个动作本身就应该带等待?

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**，可能是网络慢、可能是异步渲染。

- 答"那我在每条用例前面加 sleep 就行"——能跑，但代价大：sleep 短了不稳定、长了整体变慢，而且**它把不确定性转移给了每条用例**。

- 追问：那正确的做法是什么？答：**智能等待**——等条件满足（元素可见、可点击）就立刻继续，超时才失败。

- 收口：所以等待不是一个技巧，是**封装该承担的责任**。

易错点 / 考点：① UI Object 的四个组成（名字／定位／操作／断言）可作简答；② 常错：把 UI Object 理解成"就是一个 XPath 变量"——它是有行为的对象；③ "定位器集中管理"是维护性考点，与 Page Objects 呼应；④ 别把等待写成固定 sleep，这是实操评分点。

过渡：元素抽象完，往上走一层——页面。

【6.34】Page Objects（2 分钟）

这一页讲 Page Objects，也就是"页面对象"模式。这是 UI 自动化里最值得学的一个设计模式，学明白它，脚本寿命能长好几倍。

核心思想一句话：一个页面（或页面里的一块区域），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，只调用页面类的方法。

举个具体的：登录页对应 `LoginPage` 类，里面有用户名框、密码框、登录按钮，以及一个 `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，所有用例自动跟着好。

它解决三个真实痛点：**定位器分散**（一个按钮的定位散在几十个用例里，现在集中一处）、**脚本不可读**（以前满屏 `findElement(By.xpath(...))`，现在全是业务动作）、**重复代码**（多个用例都要登录，现在只有一份实现）。

三条实践纪律：**一，页面对象不写断言**——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同，写进去就锁死了。**二，页面对象不写跨页流程**——"登录后跳首页再点购物车"属于用例或更上层的流程对象。**三，一个页面一个类，别搞超大基类**——继承层次一深，改一处崩一片。

课件给的参考是 Selenium 官方 wiki 的 PageObjects 页，值得看，因为这套约定是社区长期总结出来的。

板书：中间"**一个页面 = 一个类**"，类里两样东西：**元素（定位器）+ 操作（方法）**。

右侧三个收益：**定位器集中（改一处）｜用例变业务语言（可读）｜消除重复（登录只写一份）**。

左侧三条纪律：**页面对象不写断言｜不写跨页流程｜一个页面一个类、别搞巨型继承**。

底部："用例只调用页面方法，不碰元素"。

提问：为什么说"页面对象里不要写断言"？断言写进去不是更方便吗？

预设回答：

- 答"方便啊，一个方法连点带验"——短期方便，长期是坑：同一个页面在不同用例里的预期不同，写死就冲突。

- 答"因为断言属于测试意图"——很到位：**断言是"我要验证什么"，页面对象只该回答"我能做什么、现在是什么状态"**。

- 追问：那"取状态"和"断言"怎么分？答：页面对象提供 `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：一句话——**页面对象负责"提供事实"，用例负责"判断对错"**。

易错点 / 考点：① Page Objects 的思想与三条纪律是简答高频点；② 极易错：把断言写进页面对象——这是模式误用的典型；③ "一个页面一个类"要能举例说清元素与方法的归属；④ 与 UI Object 的层次关系要清楚：**UI Object 管元素，Page Objects 管页面**。

过渡：页面对象要手写初始化代码很啰嗦，Selenium 给了一个工厂来简化，看下一页。

【6.35】PageFactory（2 分钟）

这一页讲 PageFactory，它是 Selenium 提供的、用来**简化 Page Objects 实现**的工厂类。抓住两点就够：**声明式地定义元素**，和**初始化时统一注入**。

传统写法里每个元素都要现找：`driver.findElement(By.id("username"))`，元素一多，代码里全是这行东西。PageFactory 换了个思路：**在字段上写注解声明"这个元素怎么找"，构造时调用一次初始化，框架就把元素准备好**。于是页面类读起来变成一份"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，一目了然。

三个好处：**声明与使用分离**（定位写在字段声明上，方法里直接用）、**可读**（类开头就是"这个页面有哪些元素"，新人看得懂）、**支持延迟查找**（不必在页面创建时立刻找到，等到真正使用时再找，对异步渲染页面友好，能减少"还没渲染完就找不到元素"的失败）。

两条注意：**一，它不替代等待策略**——延迟查找解决"什么时候找"，但元素出现了能不能点（比如被遮罩挡住），仍要靠显式等待；**二，别把它当银弹**——它只是减少样板代码，**页面对象的纪律（不写断言、不写流程）照旧要守**。

课件给的链接是官方 wiki 的 PageFactory 页。建议你按"先手写一版 Page Objects，再用 PageFactory 改写"的顺序练一遍——**只有手写过，你才知道工厂帮你省掉了什么**。

板书：三行：

① **写法**：字段上**注解声明定位** → 构造时**统一初始化注入** → 方法里直接用字段

② **三个好处**：声明与使用分离｜可读（类开头就是元素清单）｜**延迟查找**（用时才找，适配异步渲染）

③ **两条注意**：不替代**显式等待**｜只是减样板，**页面对象纪律照旧**

底部："先手写 Page Objects，再用 PageFactory 改写"。

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"，**找不到时它还是失败**，它不会帮你等。

- 答"不能，还得显式等待'可点击'"——准确：**存在 ≠ 可见 ≠ 可点击**，这三个状态要靠等待条件区分。

- 追问：那两者怎么配合？答：定位交给 PageFactory，状态判断交给显式等待，**各管一段**。

- 收口：所以"能不能点"这个判断，永远是等待策略的活儿。

> **易错点 / 考点：**① PageFactory 的两个动作（**注解声明 + 初始化**）要能说清；② 常见错误判断："用了 PageFactory 就不用处理等待"——错；③ 延迟查找的适用场景（异步渲染、元素多）是加分点；④ 三层关系要能讲：UI Object（元素）→ Page Objects（页面）→ PageFactory（页面实现的简化）。

> **过渡：**Selenium 这条线讲完，看另一条思路完全不同的工具——Cypress。

【6.36】基于 Cypress 的 UI 自动化测试（2 分钟）

> 这一页进入 Cypress，课件给的是官网 cypress.io。我把它的定位、和它跟 Selenium 的根本差异讲清。
>
> **先看定位。** Cypress 是面向**现代 Web 前端**的自动化测试工具，它把自己定位成"**跑在浏览器里的测试工具**"，这句话是理解它的钥匙。Selenium 站在浏览器**外面**，通过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。所以它不需要驱动、不需要协议转译，**它就是页面里的另一段 JavaScript**。
>
> 三个直接好处：**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，它都能感知，很多场景**不需要手写等待**，脚本里几乎没有 sleep；**二，它能看到应用内部状态**——网络请求、定时器、本地存储都能在调试视图里看到，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，接入成本低。
>
> 代价也说清：**受浏览器约束**（同源策略、多标签页这类场景不如 Selenium 灵活）、**主要面向 Web**（原生移动应用不是它的战场）、**不是 Selenium 的替代品**（Selenium 覆盖广、生态老、能测多浏览器多种类；Cypress 在 Web 前端场景开发体验更好）。**选哪个，取决于你测什么、团队是什么背景。**
>
> 后面两页讲它的架构、安装与脚本写法。现在只要记住一句话：**Selenium 从外面指挥浏览器，Cypress 从里面和应用一起跑**。

> **板书：**中间"**Cypress：跑在浏览器内部的 Web 自动化工具**"。

左侧对比：**Selenium**＝外部指挥（驱动＋协议，覆盖广）｜**Cypress**＝内部同栈（无驱动，开发体验好）

右侧三个好处：**自带等待（少写 sleep）｜可见内部状态＋时间旅行调试｜前端生态一致、接入快**

下方三个边界：**受浏览器约束（同源／多标签页）｜主打 Web｜不是替代 Selenium**

底部："外指挥 vs 内同栈"。

> **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，但要理解它为什么能省：因为它**与应用在同一环境里，能感知状态变化**，不是靠魔法。

- 答"隐患是让人不知道等待的存在"——这话有水平：一旦遇到框架等不到的场景（比如跨域跳转），不会写等待的人就懵了。

- 追问：所以学 Cypress 要不要先学等待机制？答：要，**理解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——**方便的背后是你要更懂原理**。

> **易错点 / 考点：**① 常考判断：Cypress 属于 **UI 自动化**（不是接口测试工具），且与 Selenium 的机制差异是"浏览器内 vs 浏览器外"；② 别把 Cypress 说成"只能测前端组件"——它能做主流的端到端 Web 测试；③ 与 Selenium 的取舍要能答成"场景与团队决定"，而不是"谁更先进"；④ 后面 6.37 会讲它的架构，这一页先把定位立住。

> **过渡：**Cypress 和 Selenium 到底差在哪，光靠定位说不清，下一页用一张架构图把它摊开。

【6.37】Cypress 架构（3 分钟）

> 上一页把 Cypress 的定位立住了——它跑在浏览器里面。这一页看它的架构，也就是"跑在里面"具体长什么样。
>
> 一张图上看两条链路，一条是 Selenium 的，一条是 Cypress 的，对照着看最省力。Selenium 是四段式：测试脚本在 JVM 或 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 进程负责把测试代码打包、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，也是这门课要你记住的三个词。

第一个词，同源。 因为测试代码就在页面里，它能直接读写 DOM、直接摸到应用挂在 window 上的东西、直接看到页面发出的每一个请求。所以拦截和伪造网络响应，对它来说只是改一个函数，不需要中间架代理。

第二个词，自动等待。 它的每条命令不是"执行一次、失败就报错"，而是"反复重试直到超时"。所以你在断言里写"这个列表应该有五条"，页面两秒后才补齐，它不会立刻红，会等你。

第三个词，时间旅行。 每一步命令执行完，它给当时的 DOM 拍一张快照。用例失败时，你在调试器里把鼠标悬停到某一步，右边就回到那一刻的页面——不是截图，是可查的 DOM。

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

> **板书：**上方两条链路对照——**Selenium：脚本 → WebDriver 协议（HTTP）→ 驱动 → 浏览器**（四段，跨进程跨网络）；**Cypress：Node 进程打包注入 → 浏览器内［测试代码 ‖ 被测应用］**（同源、同上下文）。

下方三个关键词横排：**同源（可直接摸 DOM／可拦网络）｜自动等待（命令重试到超时）｜时间旅行（每步 DOM 快照）**。

右下角红字："能力边界由执行位置决定。"

> **提问：**有个场景——在 A 标签页登录甲账号，再开 B 标签页用乙账号操作，验证两边能看到同一笔订单。用 Cypress 写会卡在哪一步？为什么？

> **预设回答：**

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

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

- 答"那就换 Selenium 或 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：所以架构不是谈资，它直接决定"这个场景你能不能做"。下一页讲安装时你会看到它有多轻，但轻不等于全能。

> **易错点 / 考点：**Cypress 与 Selenium 的本质差别是必考项，答题要从**执行位置**入手：Selenium 是进程外遥控（客户端—驱动—浏览器，跨进程跨网络）；Cypress 是与应用同源同上下文运行（测试代码在浏览器内）。由这一条派生四个差别：要不要显式等待、能否时间旅行调试、能否直接拦改网络、能否多标签多会

话。判断题" Cypress 支持多标签页测试"——错。另有一个细节考点：**自动等待不是"睡一会儿"，而是命令重试到超时**，所以脚本里不该出现 `cy.wait(5000)`。

过渡：架构讲完，我们来验证它那句宣传语——"安装和使用简单"，到底简单在哪。

【6.38】安装和使用简单（2 分钟）

这一页只有两条命令，但里面有三个知识点。

第一条，安装：`npm install cypress --save-dev`。注意后面这个 `--save-dev`，它把 Cypress 记进开发依赖，不记进生产依赖。这是常识但常写错：测试工具不该被带进发布包。真正装的时候还有一步隐形成本——它会下载一个自带浏览器的运行时，包不小。国内网络在这一步卡住是常态，需要配镜像或走代理，这是你们动手时的第一个坑。

第二条，打开：`npx cypress open`。`npx` 会优先用项目本地的可执行文件，而不是全局的那一份。这条命令拉起图形界面的测试运行器，第一次运行还会顺手给你建好目录骨架：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。这就是"约定优于配置"：你不用告诉它用例在哪，放对地方它自己找。

第三条，学习路径。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 → 再谈写复杂的。

我要多说一句的是：**"简单"这两个字要靠得住，靠的不是命令少，而是失败信息清楚**。Cypress 在这一点上做得确实好——用例红了，它告诉你哪一步、当时页面长什么样。这才是"简单"的真正含义。

板书：两行命令＋一棵目录树：

`npm install cypress --save-dev`（开发依赖，不进生产包）→ `npx cypress open`（本地可执行，不污染全局）→ 生成 `e2e / fixtures / support / cypress.config`。

右侧一行："约定优于配置：放对目录，它自己找。"

【思政落点 6】：这一页两条命令就能跑起来，"简单"是它最大的卖点。但我要提醒一句：**工具越容易上手，越容易让人跳过思考**。两条命令跑通之后，很多人的第一反应是"赶紧录个脚本交差"，而不是"我到底要验证什么、断言什么"。真正的门槛从来不在安装，而在设计用例、选稳定定位器、让脚本在别人的机器上也跑得

绿。技术的易用性降低的是入门成本，替代不了判断力。以后你们用任何"开箱即用"的东西，都先问一句：它替我做了什么，又替我想掉了什么。

提问：团队里为什么强调用 `npx cypress open`，而不是 `npm install -g cypress` 之后直接敲 `cypress open`？

预设回答：

- 答"`npx` 命令短"——只是表象，短不是理由。

- 答"`npx` 用的是项目本地的版本"——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"——听上去省，实际是把多个项目的工具版本绑在一起，升级一个项目就动了所有项目。

- 收口：依赖管理的原则就一句：**工具版本跟着项目走**，锁文件和 node_modules 才是唯一事实。

易错点 / 考点：直接考命令的概率不高，但实验必踩，评分会看。要点：① `--save-dev` 与 `--save` 的区别；② `npx` 与全局安装的区别；③ `cypress open`（交互式运行器，开发调试用）与 `cypress run`（命令行无头，CI 用）的配对——这两个常出选择题。

过渡：装好了，脚本到底怎么写？下一页给骨架，并且和 Selenium 的写法对照着看。

【6.39】Cypress 脚本（3 分钟）

这一页是脚本骨架。我按"结构—三个写法要点—三条禁忌"讲，你照着敲一遍就懂了。

结构：`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('选择器')`，输入接 `.type()`，点击接 `.click()`，断言接 `.should(...)`。就这么几样，能覆盖八成场景。

第一个要点：命令是异步的，但你看不到 Promise。 里面没有 await、没有回调地狱。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"而红。

第二个要点：断言是挂在命令链上的。 `cy.get('.item').should('have.length', 5)` ——`should` 失败会把前面的 `get` 重新执行一遍，直到超时。这是它区别于 Selenium 最要紧的一处：**等待不是一件事，而是命令**

的固有属性。

第三个要点：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，慢的时候三秒还不够。正确做法是等条件——等某个元素出现、等某个请求回来。

三条禁忌跟着说：一是不要试图把命令的返回值赋给普通变量带出链外，因为那时它还没执行；要取值必须在 `.then()` 里面。二是不要只写操作不写断言，没有断言的用例跑绿了也不能说明任何问题。三是不要在一条用例里做太多事，一条用例失败应该能立刻指向一个原因。

> **板书：** 骨架四行：`describe` → `it` → `cy.visit` / `cy.get(选择器)` / `.type` / `.click` / `.should(断言，自动重试)`。

红字三条禁忌：**不写 cy.wait(毫秒)｜跨命令不共享普通变量（要在 .then 里取）｜没断言的用例等于没测。**

> **提问：** 用例里写 `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，因为当时只有 3 条"——这是拿 Selenium 的思维在答。`should` 会重试。

- 答"会通过，它重试到补齐为止"——对，前提是在默认超时窗口内。

- 追问 8 秒的情况：超时窗口内等不到就失败。这时正确做法是调大这条命令的 timeout，或者先等列表接口返回，而不是塞一个 `cy.wait(10000)`。

- 收口：自动等待的本质是"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：** ① Cypress 命令不是 Promise，不能 await，也不能跨命令赋给普通变量——高频错点；② `should` 会自动重试，`expect` 不会（同步断言，一般写在 `.then` 里），这一条最常考；③ `cy.wait(数字)` 是坏味道，评审要拦。记忆口诀：**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？下一页分两种场合讲。

【6.40】Cypress 执行（2 分钟）

执行分两种场合，这两条命令千万别混。

开发时用 `npx cypress open`。 拉起测试运行器，所有用例文件列在一侧，点哪个跑哪个。跑起来之后，面板里是每一步命令的清单，旁边的窗口是应用的实时画面。把鼠标移到某一步上，那一侧的页面就回到当时的 DOM——这就是上一页说的时间旅行。用例红了，你直接看到失败那一步页面长什么样，不用靠猜、不用加打印。

CI 里用 `npx cypress run`。 无头执行，终端出日志，同时**默认录视频、失败自动截图**，产物落在固定的两个目录里。这条才是自动化的形态：它可以指定只跑某个文件、指定浏览器、把结果上传到云端面板看历史趋势。

两条命令的差别不只是"有没有界面"。`run` 的配置更严格，比如基准地址没配好，相对路径就会直接报错；在容器里跑还要保证依赖齐全，缺共享库会起不来。这些都是工程问题，不是框架问题。

最后一句是我要强调的：**跑起来只是及格，跑完能定位问题才叫能用**。你们的实验交付里，截图、日志、报告和脚本一样重要。

板书： 两条路对照——

open：交互运行器｜时间旅行回放｜开发调试｜看"每一步长什么样"

run：无头｜CI｜产出 videos・screenshots｜可上传云端面板｜看"整体过没过"

一行总结："能不能跑"和"跑完能不能定位"是两件事。

提问： 团队要求每次提交后自动跑一遍 UI 用例，并保留失败现场。你用哪条命令？还要额外配什么？

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，而且 open 是交互式的，不适合自动化。

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

- 答"还要把结果接到报告或面板里"——加分。只有终端日志的话，失败原因要人工复现；有视频和截图，定位时间能省一大半。

- 收口：所以自动化测试的**产出物设计**和用例本身一样重要，这是工程素养。

易错点 / 考点： 常考配对题：open ↔ 交互调试 ↔ 时间旅行；run ↔ 无头 ↔ CI。判断题" Cypress 只能在 Chrome 上跑"——错，可指定浏览器。另一个隐性考点：自动化测试的交付物包含日志、截图、视频、报告，不只脚本。

> **过渡：**Web 端这条线讲到这里。接下来把视线移到手机上，那里的工具谱系复杂得多。

【6.41】Android App UI 自动化测试（3 分钟）

这一页是安卓 UI 自动化的工具谱系，先认名字，再建结构。八个名字：Robotium、UIAutomator、MonkeyRunner、Appium、Espresso、Selendroid、ATAF、TestComplete。光背没有意义，我按三个维度给你归一下类。

维度一：代码跑在哪一侧。 Robotium、Espresso、Selendroid 是"仪器化"路线——测试代码被打成一个测试包，和被测应用一起装到设备上，在设备内部跑。UIAutomator 也跑在设备上，但走的是系统的控件树通道，能跨应用。MonkeyRunner 干脆不在设备上，它是电脑侧的 Python 脚本，通过 adb 遥控。Appium 也在电脑侧，但它把底层能力复用了——安卓上复用 UIAutomator，iOS 上复用苹果自己的测试框架，对外统一成一套 WebDriver 协议。

维度二：能不能跨应用。 只能测自己家应用的是 Robotium、Espresso、Selendroid；能把"我的应用—系统弹窗—别的应用"整条链路串起来测的，是 UIAutomator。

维度三：单平台还是跨平台。 Appium 是主流方案里唯一一套 API 同时覆盖 Android 和 iOS 的；TestComplete 是商业工具，主打录制回放和低代码；ATAF 是腾讯开源的方案，本质是把 UIAutomator 包了一层，做得更好用。

三个维度交叉一下，选型表就出来了：快＋白盒＋只测自己 → Espresso；要跨应用 → UI Automator；要跨平台、要黑盒 → Appium。

> **板书：**一列名字（Robotium／UIAutomator／MonkeyRunner／Appium／Espresso／Selendroid／ATAF／TestComplete）。

右边三个维度轴：**设备内 or 电脑侧 → 单应用 or 跨应用 → 单平台 or 跨平台**。

底部一行选型：**快＋白盒＋单应用＝Espresso｜跨应用＝UIAutomator｜跨平台＋黑盒＝Appium｜电脑侧遥控＝MonkeyRunner**。

> **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。Espresso 跑在被测应用进程内，它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"：用于应用内快速、稳定的细粒度验证。实践中常见组合是 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。两个高频错误：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，它是外部驱动＋协议统一）。记忆口诀：**仪器化三个在应用里，UIAutomator 跨应用，MonkeyRunner 在电脑上，Appium 两头通吃**。另需记住：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，其实根子只有一条——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）

这一页是整节的理论核心，把上一页那八个工具一刀切成两类。这一刀切下去，后面所有细节都顺了。

第一类，Appium 这一类。 课件说得很准：Appium 基于 UIAutomator 框架实现，**测试进程与目标应用进程是分开的**，所以不能直接访问目标应用的元素属性来操作，只能**模拟触发事件**。注意"分开"这两个字——是两个进程，不是一个进程里的两个线程。分开意味着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。这是**黑盒式的外部驱动**。

第二类，Robotium、Espresso、Selendroid 这一类。 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。同进程意味着测试代码能直接拿到应用内部的对象——可以调 Activity 的方法、可以断言某个控件此刻的文本、可以在界面线程上同步执行。这是**白盒式的内部观察**。

落到写用例的手感上，差别非常直观。第一类只能"点一下、看一眼"：你没法问"这个按钮的 enabled 到底是真还是假"，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——这就是课件说

> Espresso 能写出"类似白盒测试那样、更美观的脚本"的原因。
>
> 代价也要摆平说。同进程观测能力强，但**应用一崩测试跟着崩**，而且要求两边签名一致（下一页讲）。跨进程隔离好、抗崩溃、能测第三方应用，但慢、脆，控件树一变就失效。

> **板书：**左右两栏对照——

左｜Appium：跨进程 → 只能注入事件＋读控件树 → 黑盒 → 抗崩溃、通用、慢

右｜Robotium・Espresso・Selendroid：同进程两线程（Instrumentation）→ 可直接访问对象属性 → 白盒 → 快、精确、同生共死

中间一行大字："**进程边界＝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？这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，你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，但控件树是系统给你看的**公开视图**，只有系统愿意暴露的属性；应用内部的私有字段、临时状态，树里根本没有。

- 追问"那安卓为什么要设计成跨进程也能测"：因为要测别人的应用、要测系统弹窗、还要防止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：所以"能不能看到内部"不是工具好坏，是进程边界决定的。这句话请写进笔记。

> **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？"标准答法："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＝能力强、耦合紧、易被带崩；跨进程＝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：**同进程这件事听起来像魔法，其实靠一个很具体的手段实现——下一页就讲 Instrumentation 怎么做到。

【6.43】Instrumentation 框架（3 分钟）

这一页解释"两个不同的安装包，为什么能被系统当成同一个进程"。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**。展开讲：安卓靠用户 ID 做进程和数据隔离，而这个 ID 来自应用签名。你写出来的测试包和被测应用，如果用**同一个签名证书**签名，并在清单里声明共享用户 ID，系统就会把它们分配成同一个用户、同一个 Linux 进程、同一份数据目录。于是"测试线程"和"应用线程"就真的住在同一个屋子里了。

这就是为什么你在开发环境里新建一个仪器化测试，它是自动用调试签名签出来的，而且和被测应用的调试签名一致——**这不是巧合，是前提条件**。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、按 `setUp → 测试方法 → tearDown` 的顺序调用、收集结果。它靠清单里那个登记项里的目标包名，知道"我要测的是哪个应用"。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，我每年都能见到。

第三个层面是它的意义：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，根子就在这里。

板书：三格竖排——

① **同签名 → 同一用户 ID → 同一进程（清单里声明共享）**

② **清单登记项：目标包名 = 被测应用包名**（填错＝找不到目标）

③ **运行器：加载测试类 → setUp → 测试方法 → tearDown → 收集结果**

【思政落点 7】：这一页的技术细节里藏着一个值得停一下的问题：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**"。签名在这里不是一张图、一个标识，它是信任凭证——签了同一个名字，就共享进程、共享数据、共享权限。这意味着两件事：一是开发时随手共享、随手用调试签名，等于自己拆掉隔离墙的一块砖；二是签名私钥一旦泄露或被滥用，对方拿到的不是"一个文件"，而是"以你的身份运行"的能力。请记住：**技术上的边界，本质上是责任上的边界**；你手上那串密钥，对应的是用户对你的信任。

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"——如果同包名不同签名，确实会冲突；但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"——这才是准确答案。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：所以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 同进程的实现手段是"同签名＋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 执行器负责的是"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。

> **过渡：** 执行器具体怎么用、有哪些参数，下一页把命令摊开。

【6.44】InstrumentationTestRunner（4 分钟）

这一页信息密度最高，我拆成三条线：写什么、配什么、怎么跑。

写什么。 编写测试用例，对软件包里的类做单元、功能、性能测试。课件列了一串基类，别被吓住，它们的分工有规律：测界面要走真实启动的，用带启动能力的 Activity 基类；只测生命周期回调、不真显示界面的，用 Activity 单元测试基类；测服务的用服务基类；测内容提供者的用提供者基类；测应用级对象的用应用基类；什么都不依赖的纯逻辑，用最普通的安卓测试基类。还有一个"整个测试类只启动一次界面"的基类，专门用来省掉每条用例重启的开销。最后一个通用的仪器化基类，能注入事件、能拿到仪器化实例，是很多场景的落脚点。

配什么。 在测试包的清单里，把登记项的目标包名设成被测应用的包名。这一步等于把测试包"绑"到目标上，绑错了运行时就找不到。

怎么跑。 `adb shell am instrument -w`，`-w` 的意思是等测试跑完再返回结果。这里有个最容易漏的细节：**不带任何可选参数时，跑所有测试，但排除性能测试**。三个常用参数要记准：`-e func true` 跑所有功能测试，也就是源自仪器化测试基类的那些；`-e unit true` 跑所有单元测试，指的是不从那个基类派生、且不是性能测试的；`-e class` 指定单个测试类跑，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制环境、单独看数据。混在一堆功能用例里跑，测出来的

> 耗时没有意义——它测的是"排队时间"，不是"执行时间"。

> **板书：**三段式——

写：基类按被测对象分（界面／Activity 单元／服务／提供者／应用／单次启动／通用仪器化）

配：目标包名 = 被测应用包名

跑：`am instrument -w` → 默认「全部，但排除性能测试」；`-e func true`（功能）／`-e unit true`（单元）／`-e class`（单个）

> **提问：**你写了 30 条用例，25 条继承通用仪器化基类，5 条继承最普通的安卓测试基类。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` 是有筛选口径的。

- 答"5 条，因为 unit 只跑不从仪器化基类派生的"——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：所以"跑全部"和"按类跑"的边界，是**继承关系**决定的。你选基类的时候就决定了它会被哪条命令选中——基类不能乱挑。

> **易错点 / 考点：**三个参数的口径必背，而且极易混淆。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：① 不带参数时**排除性能测试**，这个"排除"最常被漏；② 目标包名填被测应用，不是测试包。判断题"am instrument 默认会执行性能测试"——错。

> **过渡：**仪器化这条线讲完，回头看一个更早、思路完全不同的工具——电脑侧遥控的 MonkeyRunner。

【6.45】示例 1：Monkeyrunner（3 分钟）

> MonkeyRunner 是安卓开发工具包里自带的老工具，年代早，但思路非常清楚：**用 Python 写脚本，在被测应用的代码之外控制设备**。
>
> 先把最容易错的一件事分清：**Monkey 和 MonkeyRunner 不是一回事**。Monkey 是命令行工具，往应用里灌随机的点击和按键，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、发哪个按键、截哪张图，然后自己判断对错。一个随机、一个可控，这是两件完全不同的事。
>
> 课件的定义很准确：基于 Python 脚本实现复杂测试用例的界面测试工具，**通过坐标、控件 ID 来操作应用的界面元素，截取测试执行的界面，进行图像比较分析来发现问题**。这一句里既有它的三个能力，也有它的三个软肋。
>
> 能力一，坐标与控件定位。控件 ID 定位还算稳，但很多场景只能靠坐标——按某个位置按下去。坐标的问题不用我多说：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。
>
> 架构位置上也要说清：脚本跑在电脑上，通过 adb 和设备通信，所以它天然属于"外部驱动"那一类，和 Appium 同侧，而不是仪器化那一侧。

> **板书：**中间写"**MonkeyRunner = Python 脚本 + 电脑侧 + adb**"。

左侧能力三连：**坐标／控件 ID 操作 → 截图 → 图像比对**

右侧软肋三连：**分辨率敏感｜界面改版即失效｜没有内部断言**

右上角红字："**Monkey ≠ MonkeyRunner：一个随机，一个可控**"。

> **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"；一旦团队形成"这条用例老红但没事"的共识，整套自动化就废了。

- 收口：所以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 vs 脚本 API）——选择题高频；② MonkeyRunner 属于电脑侧外部驱动，不是仪器化系列；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"。记忆口诀：**Monkey 乱点，MonkeyRunner 照单点**。

> **过渡：**它整套 API 只有三个类，分工干净得很，下一页讲。

【6.46】示例 1：Monkeyrunner - 续（3 分钟）

MonkeyRunner 的 API 只有三个核心类，分工像一把三节棍，一节比一节具体。

第一节，总控类。 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），以及为脚本程序创建界面、显示内置帮助。你可以把它理解成"总控台"——先在这儿把设备和脚本接上。

第二节，设备类。 它代表一台设备或一个模拟器，方法是"对设备做什么"：安装和卸载应用包、启动某个界面、发送键盘或轻触事件、运行测试包。你要执行的所有操作，都从这里发出去。

第三节，图像类。 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。注意"比较"这个动作——比较的结果是要你自己给阈值的，它不替你做判断。

三个类串起来的典型链条是：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。

课件还补了一句很关键的话：这套 API 可以**跨多个设备或模拟器**应用一个或多个测试套件，可以基于它和 Python 开发一整套自动化系统。这句话的意思是，它的价值不在"单个脚本"，而在"你能用 Python 生态自己搭平台"——读配置、遍历设备、出报告，全是 Python 的事，它只负责和设备说话。

最后提醒一句：它是 Python 2 时代的产物，官方早已不维护，新项目不要选它。我们讲它，是因为它把"截图＋比对"这条思路展示得最干净，而且这三个类的划分，是后来很多框架的雏形。

> **板书：**三节棍竖排，箭头串成链条——

总控类（连接设备／建界面／帮助）→ **设备类**（装包・启动界面・按键与点击・输入・运行测试包）→ **图像类**（截屏・格式转换・比较・写文件）

下方一行："能力不在单个脚本，在'能用 Python 搭一套'"。

提问：如果用这三个类写一个"回归检查"，最关键的设计点应该放在哪里？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，但阈值不是最脆的地方。

- 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：所以图像比对这条路，难点从来不在"截图"和"比较"这两个动作，而在**等待**和**基线管理**。

易错点 / 考点：三个类的职责常做连线题：装包／发事件→设备类；截屏／比较→图像类；连接设备／建界面→总控类。注意"比较"返回的是布尔值且要自己给阈值，它不是断言，只是比较——这一点常被误答成"它自带断言"。另外要能说出它的时代局限：Python 2、不再维护。

过渡：看一段真实脚本，把三个类的用法一次串起来。

【6.47】示例 1：Monkeyrunner - 续（2 分钟）

这一页是一段脚本骨架，我按五拍讲，你重点看四件事：连接、等待、操作、判据。

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。这里有个细节：装包是会返回成功与否的，**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，因为 MonkeyRunner **没有控件级等待**，它不知道界面渲染完了没有。这是它和 Cypress、Espresso 手感上最大的差别——前面两个框架是"自动等"，这里是"你自己估"。

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、进行图像比较分析来发现问题"，落到脚本里就是这两三行。

这段脚本最大的教学价值，是让你看清它的判据有多"软"：不是"断言等于什么"，而是"两张图像不像"。判据软，维护成本就硬。

板书：五拍竖排——

导入 → 等连接（阻塞等设备）→ 装包＋启动（判返回值）→ 睡（它没有自动等待）→ 点／输／按 → 截图＋写文件＋比对（判据＝图像相似度）

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，换机型就点到别处去了。

- 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：有没有判装包返回值、有没有等待、坐标有没有按屏幕尺寸做适配。考试层面只需记住一句：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"。这就是 Robotium 那条路。

【6.48】示例 2： Robotium（3 分钟）

Robotium 是仪器化路线的代表，国内不少老项目用过。课件给了四个特点，我逐个讲清"这意味着什么"。

第一，支持对原生界面和网页视图的操作。 这一条很实用。很多应用是"原生外壳＋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，能自动支持多个界面（Activity）。 老框架有个通病：界面一跳转，就得重新拿一次界面引用。它帮你管这件事，跳转之后你还能继续用同一个操作对象。

第三，有单独的录制回放工具。 录你的操作生成脚本。但记住我前面说过的：录制生成的脚本脆、难维护，这是行业共识，第 68 页会专门讨论这个现象。

第四，可以和 Maven、Gradle、Ant 这些构建工具集成。 这条是工程化的命门——能进构建、能进流水线，才算真能用。跑不进去的框架，最后都变成"某个人电脑上的脚本"。

它的 API 风格很集中，几乎都在一个"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，这是它的设计取向——**让脚本读起来像一份操作说明书**。

课件那句总结很到位：能够对各种控件进行操作，模拟各种手势操作，还有查找和断言机制的 API。

最后要交代清楚：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。学它，是为了理解"同进程白盒"这条路线怎么落地，不是为了今天还去用它。

> **板书：**四特点竖排——**① 原生＋网页视图双支持 → ② 多界面自动管理 → ③ 自带录制回放 → ④ 与 Maven／Gradle／Ant 集成**

中间写核心操作对象；下方一行："路线：仪器化 → 同进程 → 白盒"。

> **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，而且生成的代码通常又长又不可读。

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

- 收口：工具没有对错，看你把它当**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，这一点容易被忽略。判断题"Robotium 支持网页视图测试"——对。还要能说清它属于仪器化、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？下一页给一个典型例子。

【6.49】Robotium 脚本示例（2 分钟）

这一页是 Robotium 的典型脚本，看"三段式"结构和它跟 MonkeyRunner 的差别，比看语法更有收获。

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。 用标准的断言方法，或者断言当前界面类。注意这里的断言是**结构化的**——不是"图像像不像"，而是"现在是不是这个界面""这段文字在不在"。

和上一页的 MonkeyRunner 脚本比，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"等文字出现""等界面出现"这类条件等待，虽然不如 Cypress 那种命令级重试智能，但至少是"等条件"而不是"睡死"；它的判据是断言，能精确指出是哪一条期望没满足。

这三条红利，全部来自"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？"

> **板书：**三段式——**准备（拿仪器化实例 → 构造操作对象）→ 操作（等界面／按文字点击／按序号输入／等文字）→ 断言（断言当前界面／标准断言）**

右侧小对照表：**坐标 vs 文字定位｜睡 vs 等条件｜图像比对 vs 断言**

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"的角度说清楚。

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"——这是关键。控件是结构，位置是像素；结构相对稳定，像素随分辨率、语言、字号而变。

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

- 收口：定位策略的稳定性排序记一句话——**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，导致操作对象初始化失败。考试层面牢记"定位依据的稳定性排序"，这一条几乎能套用到所有界面自动化题上。另注意：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——UI Automator。

【6.50】UI Automator 测试框架（3 分钟）

UI Automator 是官方的界面测试框架，它的定位和前面几个都不一样：**它面向整台设备，不是面向某一个应用**。

课件给了三个组件，正好对应它的三层能力。

第一层，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，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。 这是它最本质的特点。因为走的是系统的无障碍通道，它能看到**所有**应用画出来的控件树——包括系统设置、通知栏、权限弹窗、别人的应用。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。

第三层也带来代价：**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。

工程定位上补一句：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。

> **板书：**三件套竖排——**查看器（导出控件树，看得到）→ 设备与元素操作 API（找・点・输・滑，动得了）→ 跨应用 API（无障碍通道，测得到别人的界面）**

下方一行定位："**在设备内、在应用外，本质是黑盒**"。

> **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，不少团队被这个坑卡过。

- 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，但那叫"绕过"，不叫"测试"。真要做兼容性验证，还是要能真实处理弹窗。

- 收口：所以选框架之前先问一句：**我要测的流程里，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"这一条。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，它导出控件树，截图只是附带）。另需记住：它的测试跑在设备上，MonkeyRunner 跑在电脑上。

> **过渡：**那个查看器用起来是什么感觉？下一页看界面。

【6.51】UI Automator Viewer（2 分钟）

> 这一页是查看器的使用界面，它的作用只有一句话：**让你知道该拿什么去定位一个控件**。
>
> 界面上三块信息，正好对应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、要钻几层。
>
> **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，全在这儿。**写定位器就是从这张表里挑一个稳定的属性**。优先级还是那句话：标识优先，其次内容描述和文本，层级路径次之，坐标最后。
>
> **第三块，截图与控件的联动。** 点一下树上的节点，图上对应的区域就高亮。这一步在排查"我明明点了它却没反应"时特别有用——你会发现你要点的那个小图标，实际可点击的区域是它外面那层没名字的容器。
>
> 这里有个实战细节值得说：很多控件点不动，不是定位错了，而是**它本身不可点击**，可点击的是它的父节点。属性表里"是否可点击"那一栏就是答案，看一眼就明白，比瞎试快一百倍。

> **板书：**三块——**控件树（在哪一层）→ 属性表（用什么定位：标识／内容描述／文本／类型／可用／可点击／区域）→ 截图联动（真正可点击的是谁）**

旁注一句："**可点击属性是排查'点不动'的钥匙**。"

> **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：这三条其实也是优先级：先看能不能借父节点，再考虑提需求，最后才退回坐标。

> **易错点 / 考点：**这一页不直接考，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——这就要一个代表整台设备的对象。

【6.52】访问设备状态（3 分钟）

> 这一页讲那个代表"整台设备"的对象。你把它想成手里那台手机——所有跟设备本身有关的动作，都从它发出去。
>
> **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，因为你的滑动距离、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。
>
> **再看它能做什么，课件列了五条。**
>
> 第一，**改变设备的旋转**。横竖屏切换是最容易暴露布局问题的场景之一，一行代码就能覆盖，性价比极高。
>
> 第二，**按硬件键**，比如音量键。这覆盖了实体按键这条路径。
>
> 第三，**按返回、主屏幕、菜单键**。测"连续点返回能不能正确退出"这类场景全靠它。注意这类测试在仪器化路线里很别扭——返回可能导致界面销毁，测试进程跟着没了；而它在应用外面，不受影响。
>
> 第四，**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，**截取当前窗口的屏幕截图**。注意它截的是"整个屏幕"，包括状态栏和别的应用——这跟应用内截图不是一回事。

五条合起来说明它的定位：**它操作设备，不操作应用**。所以它做的都是系统级动作：转屏、按键、拉通知栏、截屏。

板书：中间写"**设备对象 = 整台设备**"。

四周五个动作：**读属性（方向・尺寸）→ 转屏 → 硬件键 → 导航键（返回・主屏・菜单）→ 拉通知栏 → 全屏截图**

下方一句："设备级动作，不是应用级动作。"

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，测试就断了。

- 答"因为它能按系统键"——也对，但更根本的原因还是进程边界。

- 追问"那它有什么做不了的"：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**。这条主线在本节已经出现三次了，请一定记牢。

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。另需记住它截的是全屏，包含状态栏。

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——Espresso。

【6.53】UI Automator（4 分钟）

这一页标题还挂着 UI Automator，内容已经转到 Espresso 了——因为 Espresso 正是在这套基础设施之上、专门为应用内测试设计的那一层。

课件先给了一句总评：**可以写出类似白盒测试那样、更美观的自动化脚本，可充分利用被测应用实现的程序代码，而且能够实现界面线程同步，解决了可能存在的并发问题，能够改进测试的可靠性**。这句话里有三个关键词，我逐个拆。

第一个，"类似白盒"。 因为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，而不是靠截图或控件树去猜。所以脚本写得像单元测试一样精确。

第二个，"界面线程同步"。 这是它最有价值的设计，也是它跟别的框架最大的区别。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——正在跑的动画、排队的消息、后台任务——在执行每个操作和断言前**自动等到它们都空闲**再动手。这就是"解决并发问题"的意思：不是让你去处理并发，而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。

课件第二段给的就是三件套：**界面匹配器**，为控件构建的一套灵活 API，借助"在视图上"这个方法定位布局里的控件，且**支持多层次定位**——也就是"找谁"；**动作 API**，一系列扩展的、完成界面交互操作的方法集，借助"执行"方法完成操作——也就是"对它做什么"；**视图断言**，借助"检查"方法判定当前所选控件的状态——也就是"它现在该是什么样"。

三段串起来就是一句：找到登录按钮、点它、确认它还在屏幕上——这就是它说的"美观"。

板书：三件套＋三问——

界面匹配器（在视图上定位，支持多层次）→ 找谁

动作 API（执行）→ 做什么

视图断言（检查）→ 该是什么样

右侧红字："**界面线程同步＝自动等空闲，不写睡**"。

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**。这比"完全不用等"精确得多。

易错点 / 考点：三件套的名称与职责必考，常做连线或填空：匹配器↔定位、动作↔执行、断言↔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，它的核心类是五个，各有分工。

【6.54】UI Automator API（3 分钟）

UI Automator 的核心 API 是五个类，课件每个都给了定义。我按"从外到内"的顺序讲，你更容易记成一条链。

第一个，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："这个列表里应该有 20 条商品""找到文本是'已售罄'的那个子项"。它是**容器视角**。

第二个，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**。

第三个，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。它提供"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**。

第四个，选择器类。 它表示对设备上一个或多个目标元素的**查询**。注意"查询"这个词——它本身不操作、不持有元素，它只是一份筛选条件，比如"标识是某某，且文本包含某某"。元素和集合，都是拿它来构造的。它是**条件视角**。

第五个，配置类。 它设置运行测试的关键参数，比如查找超时时间、是否启用压缩、界面旋转如何处理。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）拿着条件去找到目标 → 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**。

板书：五个框从外到内——**配置（参数）→ 选择器（条件）→ 集合／可滚动（容器、滚动）→ 元素（单个）→ 动作**

旁注一句："**选择器不持有元素，只是条件**。"

提问：一个商品列表有 50 条，你要点第 38 条上的"加入购物车"。你会用到哪几个类？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：这条链说明了它的思维定式——**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）。记忆口诀：**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？下一页看例子。

【6.55】UI Automator 代码示例（2 分钟）

> 这一页是一段测试代码，你对照上一页的五个类看，能一一对上号。
>
> **开头拿设备对象**：通过仪器化实例取到设备。注意这里还是能拿到仪器化实例——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。这一点很多人搞混，我特意点出来：**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？因为设备对象面向的是设备，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。如果元素在滚动区域里，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。
>
> **最后收尾**：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，但它是"用例能独立跑"的前提。
>
> 整段看下来你会发现一个特点：**它不关心应用内部怎么实现**，只关心"界面上有没有这个东西、点了变成什么样"。这就是黑盒的写法。

> **板书：**五步——**取设备 → 起应用 → 选择器写条件 → 找元素／必要时滚动 → 点 · 读文本 · 断言 → 按键清理现场**

右侧一行："不碰内部实现，只看界面。"

> **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。那它为什么不干脆直接用同进程的能力？

> **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"——这个说法很准确，也最好记。

- 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，但操作走设备与元素对象的控件树通道，因此跨应用可行，且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了。换个角度问一个问题：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）

> 进入 6.3，回归测试。这一节只有几页，但它是前面所有自动化技术存在的**经济学理由**。
>
> 先把它要回答的问题摆出来。前面讲的那些工具——Selenium、Cypress、Appium——你可能一直有个疑问：我手工点一遍也就十分钟，为什么要花两周写脚本？答案就在这一节里：**一次性的验证不需要自动化，反复执行的验证才需要**。而这世界上绝大多数测试都是反复执行的：每一次版本迭代，你都要把老功能再验一遍。
>
> 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，也是本节的难点，因为"全部重测"在稍大的系统上根本跑不完。
>
> 最后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**。

还有一个前提要说清：这一节讲的"回归"，不是"把功能再测一遍"这么简单。它是一套有明确对象（回归缺陷）、明确代价（带缺陷发布）、明确方法（四种策略）的工程活动。学完这一节，你应该能在评审会上回答两个问题：这个版本为什么必须做回归，以及为什么回归只做这些而不是全部。

板书：中间写"**6.3 回归测试**"。

左侧：**定义 → 成因（涟漪）**

右侧：**范围？策略？**

下方箭头一条："**回归测试（选什么）→ 精准测试（凭什么选）**"。

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新做一遍的操作"。

- 答"因为用例是现成的，脚本只需要写一次"——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：本节的结构常出填空：回归测试（定义・成因・必要性・策略）＋ 精准测试（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：以前都正常的，新版本反而不正常了。

【6.57】为什么需要进行回归测试？（3 分钟）

这一页只有一个场景，但它是每个测试人员都经历过的：**以前都正常的，新的版本反而不正常了**。课件给它起了名字——**回归缺陷**。

把这个现象拆开，它有三个特征，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"，而是"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。

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。

打个生活化的比方。一家餐厅的红烧肉是招牌，某天换了厨师，做法一个字没改，但新厨师把调料罐的位置挪了，结果糖放成了盐。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"的功能是没有容忍度的。新功能崩了，用户会想"再看看"；天天用的功能突然不能用了，用户当场卸载。

板书：中间写"**回归缺陷**"。

三条判定竖排：**① 曾经正常 → ② 需求未变 → ③ 受其它改动影响**

下方一行类比："**菜谱没变，糖放成了盐**"。

提问：一个功能上线三年没出过问题，这次版本因为它挂了。这算不算回归缺陷？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，那往往不是回归缺陷，而是环境、数据或依赖变了——这种反而更难查。

- 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：所以判定要走三步：先确认旧行为当时是对的，再确认需求没动，最后确认是被别的改动牵连的。

易错点 / 考点：回归缺陷的定义必背，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。记忆口诀：**旧的好、需求没动、被别人带坏**。

过渡：定义清楚了，那下一个问题就来了——为什么非得专门为它做一轮测试？

【6.58】为什么要有回归测试？（3 分钟）

这一页回答"不带回归缺陷发布，行不行"。答案是：不行，而且后果比你想的严重。

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，一次改动的波及面越大。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，因为它藏在"已验证过"的信任里。 新功能大家会重点测，老功能大家默认是好的。测试资源永远优先给新东西，于是老功能的验证被压缩成"点一下看看"。回归缺陷恰恰就活在这个缝隙里。

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"。而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**。注意它用的是"就"——意思是这不是概率事件，是必然事件。

所以回归测试的定位不是"再多测一遍"，而是**一道闸门**：每次改动都要过这道闸，过了才能发布。

> **板书：**三层理由竖排——**① 波及面随规模增长（必然有一些坏）→ ② 藏在"已验证"的信任里（最难发现）→ ③ 落在主链路上（代价最高）**

下方红字："**没有回归测试＝带着回归缺陷发布**"。

> **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，也就是下一页要讲的"基于风险选择"。

- 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。这也是团队必须记录回归缺陷分布的原因：它是最好的说服材料。

- 收口：所以跟产品、开发谈回归范围，靠的不是"我觉得要测"，而是"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——而不是"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，这一节的难题来了——怎么做？

【6.59】回归测试（4 分钟）

这一页给出正式定义，我们逐句读。

"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，导致受影响的区域出现回归缺陷。"这句讲的是机制：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——这句话为下一节精准测试埋了伏笔。

"回归缺陷：原来正常工作的功能，没有发生需求变化，而由于受其它改动影响而产生的问题。"这是定义，和上一页那三条完全对齐。

"回归测试就是为了发现回归缺陷而进行的测试。"最简洁的一句：目的单一，找退步，不找缺失。

"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。

合起来，回归测试可以这么记：**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。注意第三个——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，**失败率应该很低**。跑一千条，可能只有三五条红。如果红了一大片，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。

> **板书：**四句拆成四行——**修改有涟漪 → 回归缺陷定义（旧的好・需求没动・被别人带坏）→ 回归测试＝为发现回归缺陷而测 → 不做就带着缺陷发布**

右侧三个关键词："**修改之后・重新执行・已有用例**"。

> **【思政落点 8】**：回归测试本质上是在守一个承诺——**"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；对他来说，那个能用的版本是我们已经许下的诺。工程上的很多事故不是"新东西没做"，而是"把旧东西弄坏了还发出去"。所以做回归不是保守，是守信：**对已经交付的东西负责，**

比对新东西负责更难，也更见品格。发布前多跑一遍回归，是对用户的一次交代。

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。所以一个版本发布前要做两件事：**新功能的验证 + 老功能的回归**，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：所以回归测试只保证"已覆盖的部分没退步"，不保证"没覆盖的部分没问题"。这也解释了为什么覆盖率重要。

易错点 / 考点：定义必背，三个关键词（修改之后、重新执行、已有用例）是答题骨架。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 vs 找缺失）。

过渡：定义清楚了，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）

这一页只提了两个词，但这两个词就是回归测试全部的实操难点：**范围和策略**。

先说范围。 这次改动之后，我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；范围定大了，测试跑不完、项目延期，最后还是"没测完就发"。所以范围问题的本质是：**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。 确定了范围，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？按代码改动的位置？

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

举个具体的例子。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？策略问题：按风险，优惠券和订单金额都必须测，因为跟钱有关；按操作剖面，就看用户实际最常走哪条路径，如果大多数用户只用满减，那满减优先；按改动的部分，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，而是"在这个项目、这个版本、这个风险水平下，哪个合适"的问题。

板书：中间两个大字"**范围？策略？**"。

下方一行箭头："**策略（凭什么选）→ 范围（测哪些）**"。

右侧三个圈：**优惠券（改动点）→ 订单金额（风险）→ 常用路径（操作剖面）**。

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：所以"改动很小"不是缩小范围的理由，"调用链很短"才是。

易错点 / 考点：这一页本身不考具体知识，但它是策略页的引子。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，策略是用来平衡它们的"，就已经到位。

过渡：那到底有哪几种策略？下一页给四种。

【6.61】回归测试策略（4 分钟）

这一页是本节的重点。四种策略，我每一种都讲清三件事：怎么做、成本多少、风险在哪。

第一种，再测试全部用例。 把已有的用例全部重跑一遍。成本最高，风险最低——理论上没有漏网之鱼。什么时候用？两种场合：一是安全关键系统，漏一个的代价远大于多跑几天的成本；二是版本临近发布、时间还够的收尾阶段。现实里它很难长期执行，因为用例会随时间膨胀，全量跑的时间会从几小时涨到几天。

第二种，基于风险选择测试。 按"出问题的概率乘以后果"给各区域排序，优先测高风险的。哪些算高风险？改动涉及的、历史缺陷密集的、在主链路上的、复杂度高的、多人同时改过的。成本中等，风险可控，这是

工业界最常用的思路。难点在"风险怎么量化"——不能凭感觉，要拿历史缺陷数据、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选，用得最多的路径优先。这个逻辑很硬气：**用户绝大多数时间在走那少数几条路径，那几条路径坏了才是真灾难**。成本低、性价比高。盲区也很清楚：冷门路径几乎不测，一旦坏了往往没人发现，直到某个客户投诉。

第四种，再测试修改的部分。 只测这次改动直接影响的部分。成本最低，风险最高——因为涟漪是往外传的，你只测了圆心。它适合的场景是：改动非常局部、依赖清晰、而且有自动化能快速验证调用方。

课件最后还留了个省略号，意思是策略不止这四种，还有基于影响分析、基于变更覆盖等等。

那实际怎么组合？ 我给你们一个工业界常见的结构：改动点周围用"再测试修改的部分"做快速验证；主链路和高风险区用"基于风险"补齐；发布前用"基于操作剖面"覆盖用户高频路径；安全关键模块或者大版本，才上"全部重测"。

板书：四行表——

全部重测：成本最高／风险最低／关键系统与大版本

基于风险：成本中／概率×后果／工业常用

基于操作剖面：成本低／按使用频率抽样／高频优先，冷门易漏

再测试修改部分：成本最低／风险最高／局部改动

底部一行："**实际＝分层组合：改动点快验 ＋ 风险补齐 ＋ 高频冒烟 ＋ 关键全量**"。

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？说明理由，并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，但下单是最高频的主链路，必须验。

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

- 收口：所以答案是"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。另一易错点：以为"再测试修改的部分"是最优解——它是成本最低，不是风险最低。记忆口诀：**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：这四种策略都有一个共同前提——你得知道"改动影响了谁"。靠猜不行，得靠证据。这就是 6.4 要解决的问题。

【6.62】6.4 精准测试（2 分钟）

进入 6.4，精准测试。这一节回答上一节留下的那个问题：**凭什么说"这些该测、那些不用测"**？

先给个定位。精准测试不是一种新的测试方法，它是**一套让回归范围"有据可依"的技术体系**。它要解决的核心矛盾就是上一页说的那个：范围大了跑不完，范围小了漏缺陷。传统做法靠人的经验判断"改这儿会影响到那儿"，精准测试要把这个判断变成**可计算、可追溯的证据**。

它的基本思路只有一句话：**把测试用例和代码建立映射，再分析代码之间的依赖，那么"代码改了什么"就能反推出"哪些用例必须重跑"**。听起来简单，做起来有三个基础工作，下一页展开。

为什么这一节重要？因为在持续交付的今天，一天发几次已经不稀奇。如果每次发布都要靠人判断回归范围，那发布频率就被人的判断速度卡死了。精准测试是让"高频发布"在质量上站得住的前提。

顺带说一句它的边界：它解决的是"选得准"，不解决"测得好"。用例本身写得烂，选得再准也没用。

还要提前打个预防针：这一节的技术名词听着唬人——覆盖率、映射、依赖图，但它们的共同目标非常朴素，就是回答一句话："我改了这里，哪些老用例必须再跑一遍，凭什么"。你把这句话记住，后面三个基础就不会乱。

板书：中间写"**6.4 精准测试**"。

上方一句问话："**凭什么选这些用例？**"

下方三个词：**覆盖率分析 · 用例与代码映射 · 依赖分析 → 精准回归**

右下角一行："**把经验判断变成证据**"。

提问：精准测试要靠工具采集覆盖率、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，只是"靠人记得住"。

- 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，这也是精准测试的雏形。

- 答"等出过事故就知道要用了"——这句很现实，但代价太大。

- 收口：所以精准测试的核心不是工具，是**"改动与用例的对应关系"这件事有没有被显式记录下来**。工具只是把它自动化。

> **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：**三个基础工作，一个一个看。

【6.63】精准测试（3 分钟）

这一页列出精准测试的三个基础，我讲清每个基础"解决什么问题"。

第一，测试覆盖率分析。 它回答"现有用例到底测到了哪些代码"。注意方向：这是**从用例到代码**的测量。手段有代码插桩、字节码分析、运行时探针。测出来的结果是"每一行、每个分支被哪些用例覆盖过"。没有这一步，你既不知道自己的用例网有多密，也不知道哪些代码在裸奔。

第二，代码和测试用例之间的关系。 覆盖率分析天然产出这份映射：某条用例执行时经过了哪些类、哪些方法。把这份数据反过来建索引，就得到双向的映射——用例到代码，代码到用例。这是精准测试的**核心数据结构**。为什么关键？因为回归时要回答的问题是"我改了这段代码，哪些用例必须跑"，这正好是"代码到用例"那个方向的查询。

第三，代码依赖性分析。 它回答"改了 A，还可能影响谁"。依赖有很多种：方法调用、类继承、接口实现、数据库表、配置文件、消息主题、前端组件引用。分析出来的结果是一张依赖图。有了它，"改动点"就能沿着依赖边往外扩散，算出影响范围。

三者合起来是一条链：**改动代码 → 依赖分析扩散出影响范围 → 用"代码到用例"的映射查出覆盖这些代码的用例 → 这些就是必须重跑的最小回归集**。这就是课件那句"基于上述工作，可以精准地完成回归测试"。

> 它的价值有两条，必须同时成立才叫"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**，不在影响范围内的用例不用跑，时间省下来了。

> **板书：**三个基础——**① 覆盖率分析（用例→代码，测得有多密）→ ② 双向映射（核心索引）→ ③ 依赖分析（改 A 影响谁）**

下方公式一行："**改动 → 依赖扩散 → 映射反查 → 最小回归集（不漏也不多）**"。

> **提问：**如果团队用例很少、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"——表面上是，实际上这是最危险的情况。

- 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。精准测试会诚实地告诉你"这里没有用例"，但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。**精准测试的结论必须配合覆盖率数据一起读**。

- 收口：所以它的前提是"用例网足够密"。网破的地方它救不了你，只会如实告诉你网破了。

> **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"改了影响谁"。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？下一页专门讲。

【6.64】测试用例与代码映射关系（3 分钟）

> 这一页只有两个箭头：**用例到代码**，和**代码到用例**。看着像同一件事的两种说法，其实是两种不同的查询，用途完全不同。
>
> **先说用例到代码。** 方向是"跑这条用例会执行哪些代码"。用途是**评估用例质量**：一条用例只覆盖三行，价值有限；一条用例覆盖两百行、跨五个类，那是主干用例。这个方向还用来找**冗余**——两条用例覆盖的代码完全一样，它们大概率重复，可以合并；以及找**缺口**——某些代码从来没被任何用例走过，那就是裸奔区。
>
> **再说代码到用例。** 方向是"这段代码被哪些用例覆盖"。用途是**回归选型**：这次改了这段代码，把这批用例挑

出来重跑，就是最小回归集。这个方向还辅助**缺陷定位**——某个方法报错，能立刻知道哪些用例该红。

为什么必须双向？ 因为两个方向回答的是两个不同的问题，而且**不能互相推导**。你有完整的正向数据，理论上能建倒排索引得到反向结果，但工程上这两份数据往往来自不同通道：正向靠运行时插桩，反向靠静态分析与索引；更新频率也不同——用例天天加，索引要跟着重建。

再深一层：**映射的粒度决定精准度**。粒度细到方法级甚至行级，精准度高但数据量大、开销大；粗到类级或文件级，开销小但会放大回归范围——改一行，整个类都被判为受影响。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。

这一页的技术本质，其实是件很朴素的事：**说"这段代码没问题"，得有证据**。

板书：两个箭头对着写——

用例 → 代码：评质量・找冗余・找缺口

代码 → 用例：回归选型・缺陷定位

中间一行："同一份数据，两个方向的查询，**不能互相替代**。"

下方："粒度——行级最准最贵，类级最省最粗。"

【思政落点 9】：这一页的技术本质是一件很朴素的事：说"这段代码没问题"，得有证据。映射表就是证据链——某条用例跑过这段代码，这段代码才敢说"被验证过"。工程上最容易出问题的地方，恰恰是"我觉得没问题"：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。精准测试的价值，就是把这些"我觉得"换成"数据显示"。**用数据说话，不拍脑袋**，这既是测试的职业素养，也是任何工程判断该有的态度。

提问：团队想做精准测试，但只采集了"用例到代码"这个方向的数据。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：如果只记了"这条用例跑过这个类"，最多反查到类级，做不到方法级。

- 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"那干脆两个方向都采"——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，而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匹配，常做配对题：用例→代码＝评估用例质量／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"——它们的采集方式、更新频率、使用场景都不同。

过渡：映射解决了"谁覆盖了谁"，但"改了 A 会不会影响 B"，还得靠依赖分析。

【6.65】代码依赖性分析（3 分钟）

这一页讲依赖分析，它要回答的问题是：**改动一个地方，涟漪能传多远**。

先看依赖有哪几种，这决定你的分析要覆盖哪些边。**调用依赖**：方法 A 调方法 B，最直接，改了 B，A 的行为可能变。**继承与实现依赖**：改父类一个方法，所有子类都受影响——这是最容易漏的一类，因为你改的地方和出问题的地方在代码里可能隔得很远。**数据依赖**：共用同一张表、同一个字段、同一个缓存键，改字段类型，所有读写它的地方都危险。**配置与资源依赖**：共用配置文件、共用同一个开关，改一个默认值，影响面可能横跨多个模块。**消息与事件依赖**：发布订阅关系，改消息体结构，所有消费者都受影响——这类最隐蔽，因为代码里没有直接调用，是运行时连起来的。

再看分析怎么做，两条路。**静态分析**：解析代码，把调用关系、继承关系抽出来建图。优点是全量、快、不用跑；缺点是有动态性盲区——反射、依赖注入、动态代理、脚本化配置，静态分析往往看不出来。**动态分析**：跑起来，记录实际执行轨迹。优点是准，能看到真实调用链；缺点是只覆盖跑到的路径。

工业界的做法是**两者结合**：静态给全量骨架，动态修正和补充。

最后是这套分析怎么用：从改动点出发，沿依赖边做**传递闭包**——一层调用、两层调用、三层调用，扩散出影响范围。实践中会设一个"扩散层数上限"，再往外就认为影响可忽略。这个层数，就是精准度和成本之间的那个旋钮。

板书：左边五种依赖边：**调用・继承实现・数据・配置资源・消息事件**

右边两条分析路：**静态（全量・快・有盲区）／动态（准・只覆盖跑到的）**

下方一行："**改动点 → 沿依赖边传递闭包（设层数上限）→ 影响范围**"。

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。这会让精准测试出现什么问题？

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 就不会被选进回归集，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：依赖扩散要设层数上限，这是精准度与成本的平衡点。

> **过渡：**三个基础里还剩一个——覆盖率分析，它是所有证据的源头。

【6.66】测试覆盖率分析（3 分钟）

这一页讲覆盖率分析，它在精准测试里扮演"数据源头"的角色。

先说它测什么。 覆盖率有几档粒度，从粗到细：代码行覆盖，这一行被执行过没有，最粗也最容易做；分支或判定覆盖，判定的真假两条路都走过没有；再往细是条件覆盖、路径覆盖。在精准测试的语境里，最常用的是行覆盖和方法覆盖——因为你要建立的是"用例与代码"的对应关系，粒度太细数据量会爆炸。

再说它怎么采。 主流手段是**插桩**：在编译期或运行期往代码里插记录点，跑测试时记下"这个点被谁走过"。Java 生态里常见的是字节码插桩，前端常见的是构建期注入统计代码。插桩的代价是性能开销和产物污染，所以生产环境一般不插桩，只在测试环境采。还有第三种，**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，但会漏掉短小的调用。

最后说它在精准测试里的两个作用。 第一，**建立映射**：覆盖率数据天然就是"用例到代码"的原始记录，按用例维度切开，映射表就有了。第二，**评估精准回归的可信度**——这一点最容易被忽略。如果改动落在"零覆盖区"，精准测试选出来的回归集就是空集，它会告诉你"没有用例需要跑"。这个结论必须被识别成**高风险**，而不是"没问题"。

所以覆盖率数据要和回归选型**一起读**：选出来的用例少，可能是"改动影响小"，也可能是"这段代码根本没人测"。这两种情况长得一模一样，只能靠覆盖率区分。

> **板书：**三档粒度：**行 • 分支 • 路径**

三种采集：**插桩 • 字节码插桩 • 运行时采样**

两个作用：**建映射／评估回归可信度**

红字一行："**零覆盖区 → 回归集为空 ≠ 没问题**"。

> **提问：**精准测试工具报告"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，没有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。如果是零覆盖，那就不是"没影响"，而是"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**，判断还得人来做。**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**。另一易错点：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：**到这里 6.1 到 6.4 全部讲完，我们把这一讲收一下。

【6.67】本讲小结（3 分钟）

这一页是本讲小结，五条。我按"分析—执行—守护"的顺序串一遍，你别当成五个孤立的知识点背。

第一条，基于需求和启发式测试策略进行功能测试需求分析。 这是所有测试工作的起点——不是拿到需求就开始写用例，而是先把"测什么、为什么测、测到什么程度"分析清楚。方法是分解需求，配合启发式策略，比如按功能、场景、数据、状态、时序去切。

第二条，先要明确测试目标，再进行测试范围、测试项和风险等分析。 注意这个顺序：目标在前。目标不清，范围一定失控。课件强调用**分解方法**，内容涉及**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。 这部分讲的是"人怎么在不确定的情况下高效地找缺陷"：四要素是框架，IAPE 是执行模型，SBTM 是基于会话的测试管理——把探索式测试变得可计划、可度量、可汇报。它的价值在于：不是所有测试都能提前写成脚本，探索式测试是脚本之外的那一半。

第四条，回归测试策略。 四种：全部重测、基于风险、基于操作剖面、再测试修改的部分。核心是理解成本

> 与风险的取舍，以及实践中怎么分层组合。
>
> **第五条，精准测试。** 三个基础——覆盖率分析、用例与代码的映射、依赖分析，合起来支撑精准回归。它把"凭经验选范围"变成"凭证据选范围"。
>
> 这五条其实是一条线：**分析需求定目标（一、二）→ 用方法找缺陷（三）→ 改动之后守住成果（四、五）**。前面回答"怎么找问题"，后面回答"怎么不把已经解决的问题再弄坏"。

> **板书：**五条竖排——**① 需求＋启发式 → 测试需求分析｜② 目标优先 → 范围・测试项・风险（功能・场景・数据）｜③ 探索式四要素・IAPE・生态・SBTM｜④ 回归四策略｜⑤ 精准测试三基础**

右侧一条纵线箭头："**找问题 → 守住成果**"。

> **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？为什么？

> **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"，而是"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，后面所有工作都是在错误的靶子上努力。

- 答"选精准测试"——前提是你已经有回归体系了，它是在优化，不是在建。

- 收口：这三个答案分别对应"防退步""防错方向""防跑不完"。如果团队连回归都没有，先做回归；回归有了但跑不完，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：**最后留三个问题，没有标准答案，但值得你认真想。

【6.68】提问与思考（4 分钟）

> 这一页三个问题，我都不给标准答案，给的是"讨论的抓手"，你们可以带着去查资料、去问实习单位的师兄。
>
> **第一个，录制脚本轻松搞定，为什么没成为脚本开发的主流？** 抓手在这里：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"，却不知道"我在验证登录成功"。界面一调整脚本就失效，而且失

效原因还不明确。更深一层：录制的脚本**没有断言**，能回放操作，不能判断对错；而真正有价值的脚本，工作量的大头恰恰在断言和定位策略的设计上，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**，不适合做**长期资产**。再想一层：团队说"我们用录制"，真正的成本不在录制，而在后期维护——只是这笔账被推到了后面。

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，这些是通用能力，开源早做完了，自研等于重新发明轮子，还要吃下所有兼容性成本。自研的价值只在两种情况：一是有开源解决不了的**特殊对象**，比如自绘引擎、特殊协议、嵌入式界面；二是要把它做成**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，底层仍用开源。所以不是二选一，而是**分层**：底层开源，平台自研。连自动化都没跑顺就上自研，通常结局是造了个只有自己能用的半成品。

第三个，都重视自动化，效果却不够好，问题在哪？ 给四个怀疑方向。一，**目标错了**——把"用例数量""脚本行数"当成绩，而不是"发现了多少缺陷""省了多少人工"。二，**维护成本失控**——定位器太脆，界面一改大面积红，最后没人敢跑。三，**只做界面层**——界面自动化最贵最慢，真正该自动化的是接口层和单元层。四，**没进流程**——脚本跑在某人电脑上，没接流水线，红了三天没人管。最致命的是第四条：**没人看的自动化等于没有自动化**。

板书：三问并排，每问下面挂抓手关键词——

录制：记操作不记意图・无断言・维护成本后置

自研：先问差异化・底层开源＋平台自研

效果差：目标错・维护崩・只做界面层・没进流程

【思政落点 10】：第三个问题值得多说一句。面对"自动化效果不好"，最容易的反应是找一个外部理由——工具不行、时间不够、需求变太快。但工程上的进步，几乎都始于一句诚实的自我追问：**我当初做这件事，到底想解决什么问题？** 如果答案是"因为别人都在做"，那效果不好就是必然。技术判断上的诚实比技术本身更难。愿同学们以后带项目时，敢在复盘会上说"这一项我们做错了"，而不是把它包装成"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，死因更接近上面哪一条？

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，观察它三个月后还能不能跑。

- 答"死了，因为界面老变"——对应第二条，维护成本失控，这也是最常见的死因。

- 答"活着但没人看"——这其实是"假活"，对应第四条。指标上它每天在跑，实际没有任何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**它红了以后会不会有人停下来**。没人停，就是死的。

> **易错点 / 考点：**三问是开放题，闭卷考一般不直接考，但课程设计答辩常被追问。答这类题要有立场、有理由、有边界——比如"我选自研，因为我们的对象是自绘控件，开源框架定位不到；但底层驱动还是用开源的"。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 就是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）

> 实验 3 的题目是：针对某 web 应用系统进行功能测试。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、下单流程都齐全，而且它本身就是给学习和测试用的公开靶场——这一点很重要，做测试一定要用授权的对象，别拿真实业务系统练手。
>
> 实验要求三条，我逐条说清"做到什么程度算合格"。
>
> **第一条，基于本模块所学的测试方法，进行测试需求分析，按优先级列出主要的测试项以及识别出测试风险。** 三个关键词：测试需求分析、优先级、风险。合格的产出是一张表——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 个以上相对关键的测试用例，以发现更多的缺陷，并能说明设计思路和方法。** 两个关键词：**关键**和**思路**。不是让你把"用户名空／密码空／用户名超长"这种凑数用例写满，而是要覆盖不同的设计方法——等价类、边界值、判定表、场景法、错误猜测，每条用例都要标出"我用的是哪种方法、为什么这么设计"。这一条是评分重点，因为它考的是"你会不会设计"，不是"你会不会点"。
>
> **第三条，基于 Selenium 加 WebDriver，把之前设计的 10 个测试用例转化为脚本，如 Java 格式，在 JUnit 框架上执行和调试。** 注意是"转化 10 个"，不是新写——用例的可自动化程度会影响你挑哪 10 个。合格的脚本要有：显式等待（不是死等）、稳定的定位器（优先标识）、明确的断言、用例之间互不依赖、执行后能看到结果。JUnit 上用准备和清理方法管浏览器生命周期，用测试注解放用例。

> **板书：**三行任务——① **需求分析 → 测试项表（项 · 依据 · 优先级 · 风险）** | ② **20+ 关键用例（标注设计方法）** | ③ **挑 10 条 → Selenium＋WebDriver → Java 脚本 → JUnit 执行**

右下角红字三条验收标准："**显式等待 · 稳定定位 · 独立可跑**"。

> **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。但只看"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（核心流程）、执行频率高的（每次发布都验）、判定明确的（能断言）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：自动化的选型标准记三条：**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——① 用死等代替显式等待；② 定位器写成绝对路径（一改就废）；③ 用例之间有依赖（A 依赖 B 创建的订单）；④ 只有操作没有断言；⑤ 交上来的 10 条和那 20 条对不上号。这些都不是技术难度问题，是工程习惯问题，也正是这门课要练的。

> **过渡：**想再往深学的，下一页有几本书可以挑。

【6.70】学习资源推荐（2 分钟）

推荐四本书，我按"补哪块短板"介绍，你别按顺序买，按需要买。

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**。我前面反复说过：界面自动化最贵最慢，性价比最高的是接口层。如果你的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。补的是 **Web 界面自动化的基本功**——元素定位、等待策略、页面对象、测试框架集成。这本和本讲 6.2 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。补的是**新一代前端测试的思路**——就是我们讲过的同源运行、自动等待、时间旅行那一套。前端方向的同学优先看这本。

第四本，《Robot Framework 自动化测试精解》，人民邮电出版社，2020 年 4 月。补的是**关键字驱动**这条

路——它的思路和写代码的框架完全不同，用表格和关键字组织测试，适合业务测试人员参与自动化。团队里"测试同学不写代码"是个真实问题，这本书给的就是这个答案。

最后一句建议：书不用全看，**挑一本，把里面的例子在你的实验项目上跑通**。跑通一本的价值，大于翻完四本。

板书：四本书＋各自补的短板——**接口实战（接口层）｜Selenium 从零（Web 界面基本功）｜Cypress 入门到精通（前端新思路）｜Robot Framework 精解（关键字驱动·低代码）**

下方一行："**挑一本跑通 > 四本翻完**"。

提问：如果你们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，复杂逻辑仍需要编码能力。

- 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——这是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，我把整条主线收一下。

【6.71】感 谢 聆 听（2 分钟）

这一讲我们从 Web 端的 Selenium、Cypress，走到移动端的 Appium、Espresso、UI Automator，再回到"改完之后怎么守住成果"的回归测试和精准测试。

只带走三句话就够了。**第一，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。**第二，自动化不是"多测一遍"，是"让反复执行的验证变便宜"**——它省下的不是第一次的时间，而是第 N 次的时间。**第三，回归测试守的是承诺，精准测试靠的是证据**——前者是态度，后者是方法；两者都有了，质量才不是一句口号。

下一讲进入性能测试：功能测试问"对不对"，性能测试问"扛不扛得住"。谢谢大家。

板书：三句话竖排——**① 场景定工具｜② 自动化＝让重复变便宜｜③ 回归守承诺・精准靠证据**

右下角："下一讲：性能测试"。

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，目的清楚。

- 答"就是防止改 A 坏 B"——很生动，也很好记。

- 收口：三句都对，差别在"有没有说清目的"。以后你们写测试报告、做答辩，也请养成这个习惯：**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，也是答辩时最好用的一句话总结。

过渡：本讲到此结束，下一讲性能测试见。

四、思政落点清单（共 10 处）

序号	所在页	落点主题
1	6.5	这七个维度讲的是一个朴素的道理：系统是一个整体，任何一维塌了，用户感受到的都是"
2	6.11	WSDL 这个东西值得多说一句——它把"我能提供什么、你需要给我什么"写成了一份双
3	6.15	分层自动化背后是一种很朴素的工作方法：把力气花在产出的地方。同样一天时间，压在单
4	6.23	given-when-then 这个结构值得多想一层：它要求你把"前提、动作、预期
5	6.32	这张架构图里藏着一个很值得学的设计思想：用边界换安全。驱动是唯一被允许操作浏览器
6	6.38	这一页两条命令就能跑起来，"简单"是它最大的卖点。但我要提醒一句：工具越容易上手

7	6.43	这一页的技术细节里藏着一个值得停一下的问题：整套机制能成立，靠的是"系统相信签名
8	6.59	回归测试本质上是在守一个承诺——"你昨天能用的功能，今天还能用"。用户不会因为我
9	6.64	这一页的技术本质是一件很朴素的事：说"这段代码没问题"，得有证据。映射表就是证据
10	6.68	第三个问题值得多说一句。面对"自动化效果不好"，最容易的反应是找一个外部理由——